CS 4440 A

Final Review

04/22/26

Final Logistics

Final will be released Thursday Apr 30 at 5:30PM and due Friday May 1 at 5:30PM

- 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.

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

Transaction (CC)

Two-phase locking (2PL)

”Proper” use of locks

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

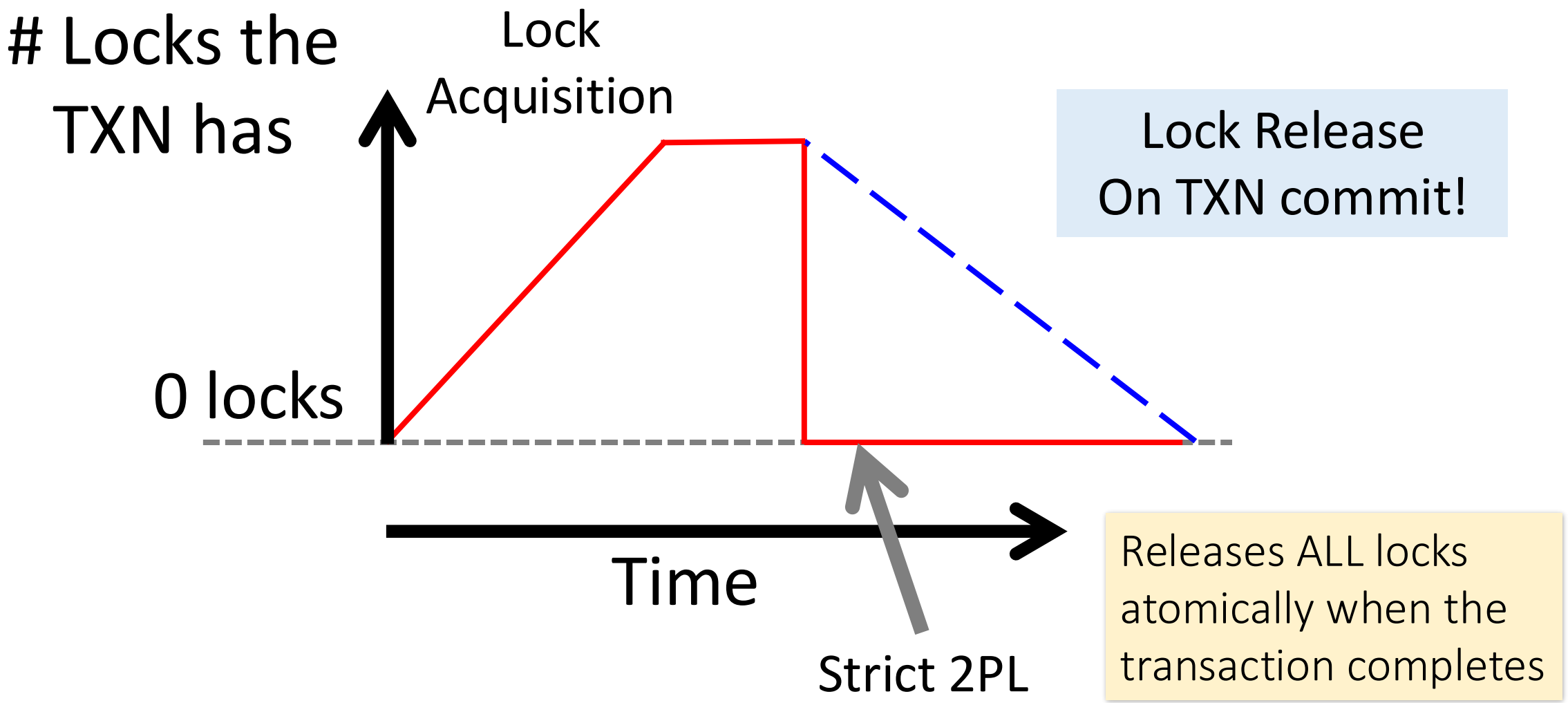
Two-phase locking (2PL)

In every transaction, all lock actions precede all unlock actions

- Growing phase: Acquire locks as needed, cannot release any locks
- Shrinking phase: Release locks as needed, cannot acquire any new locks

Guarantees a legal schedule of consistent transactions is conflict serializable

Picture of 2PL



Practice Question

cs4400_sp25_final

- 5. 2PL Feasibility

Locking with several modes

Using one type of lock is not efficient when reading and writing

Instead, use shared locks for reading and exclusive locks for writing

$sl_i(X)$: Ti requests shared lock on X
$xl_i(X)$: Ti requests exclusive lock on X

Requirements: analogous notions of consistent transactions, legal schedules, and 2PL

Locking with several modes

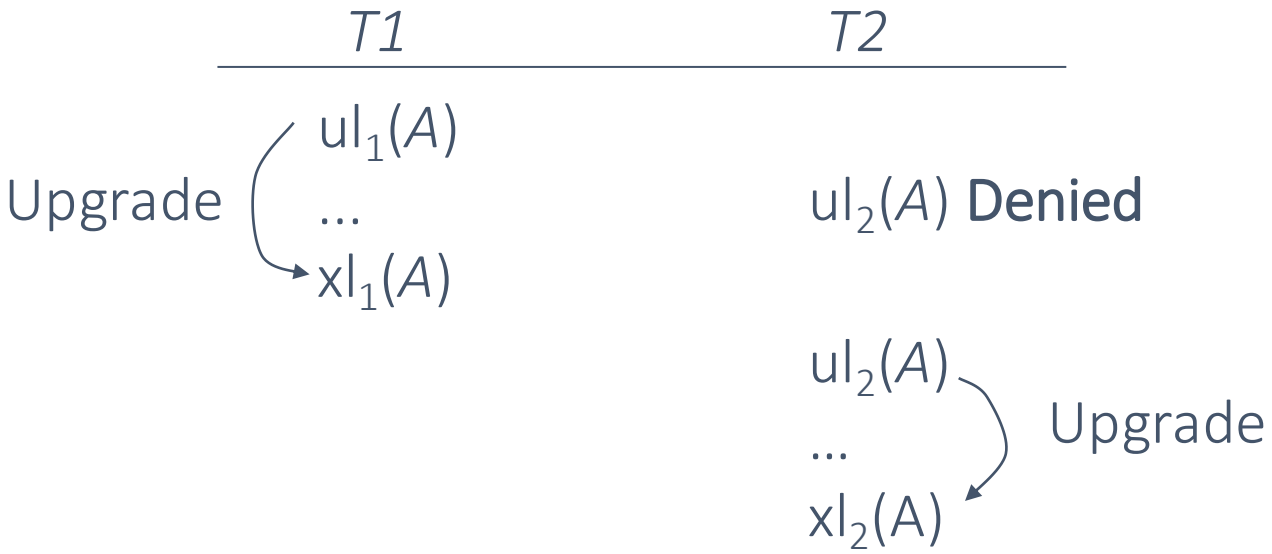
- Compatibility matrix

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

Lock Modes Beyond S/X: Update locks

$ul_i(X)$: Ti requests an update lock on X

- Solution: introduce new type called update locks
- Only an update lock can be updated to an exclusive lock later

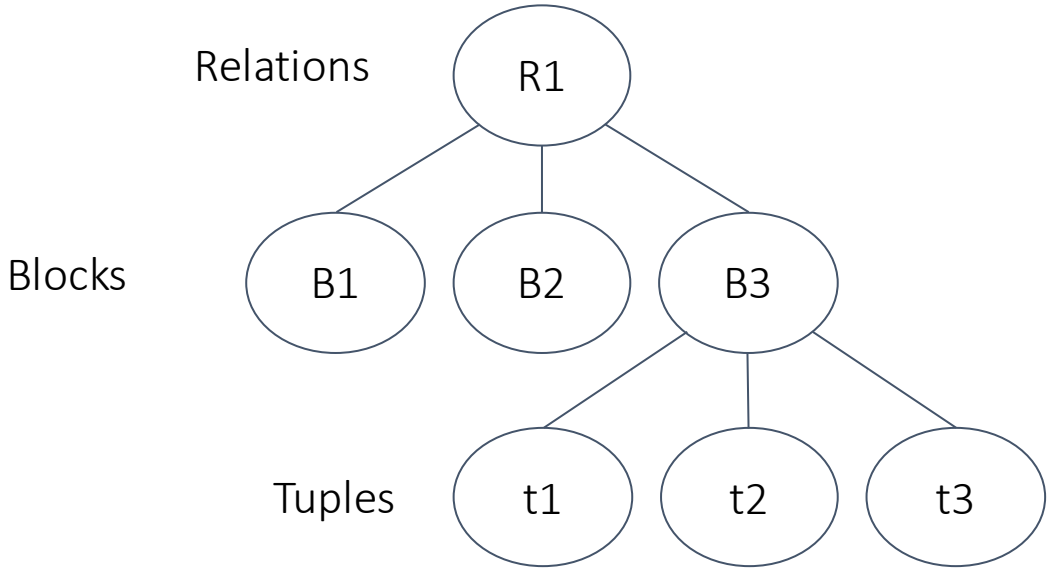


Compatibility matrix

	S	X	U
S	Yes	No	Yes
X	No	No	No
U	No	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

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 reduces 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.

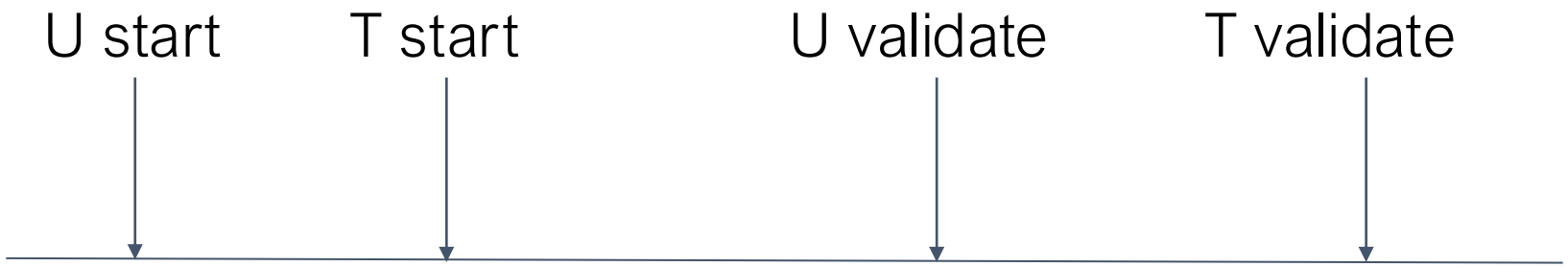
Validation rules (assume U validated)

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



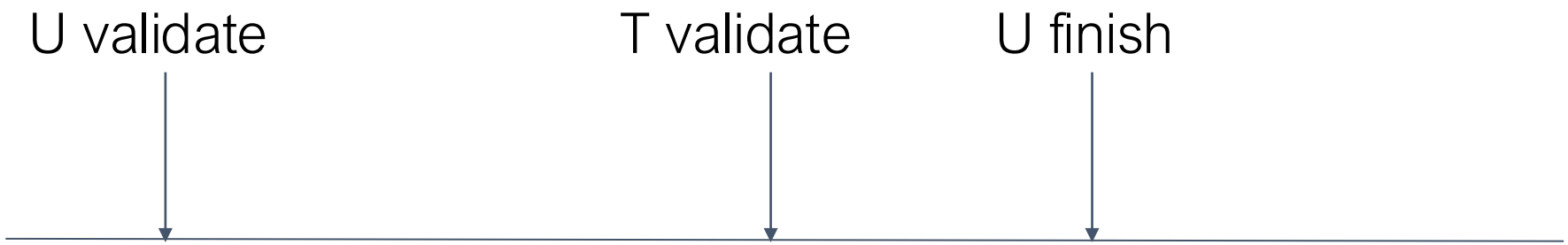
Validation rules (assume U validated)

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

WS(U) = {A, B}

WS(T) = {B, C}

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



Practice Question

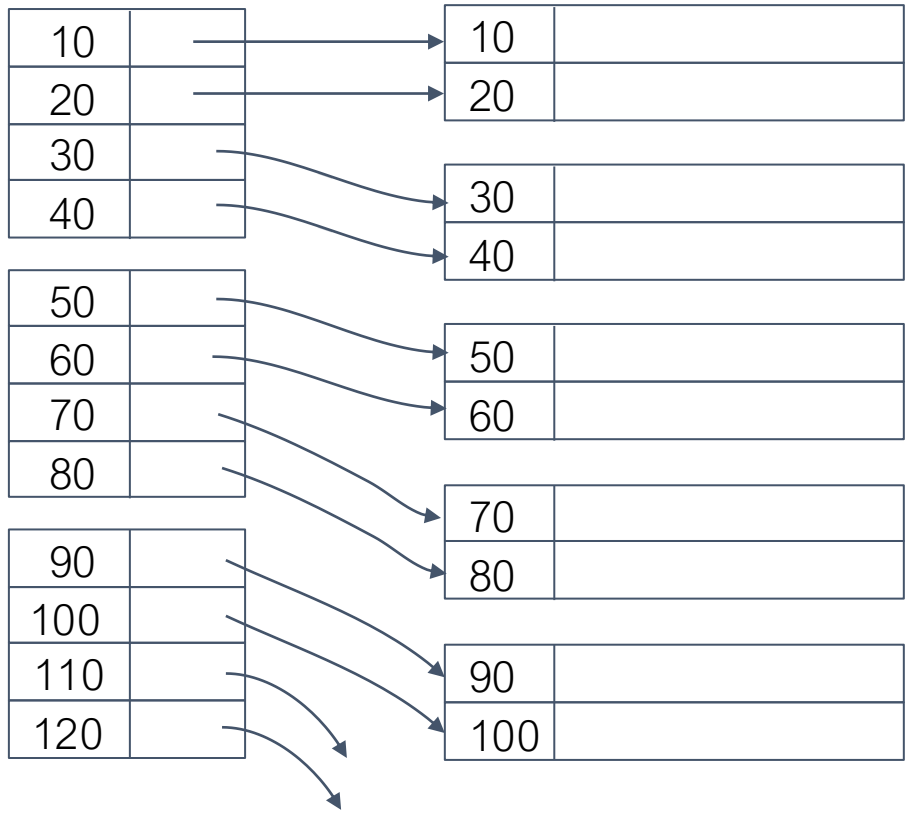
cs4400_sp25_final

- 6. Optimistic Concurrency Control

Index Basics

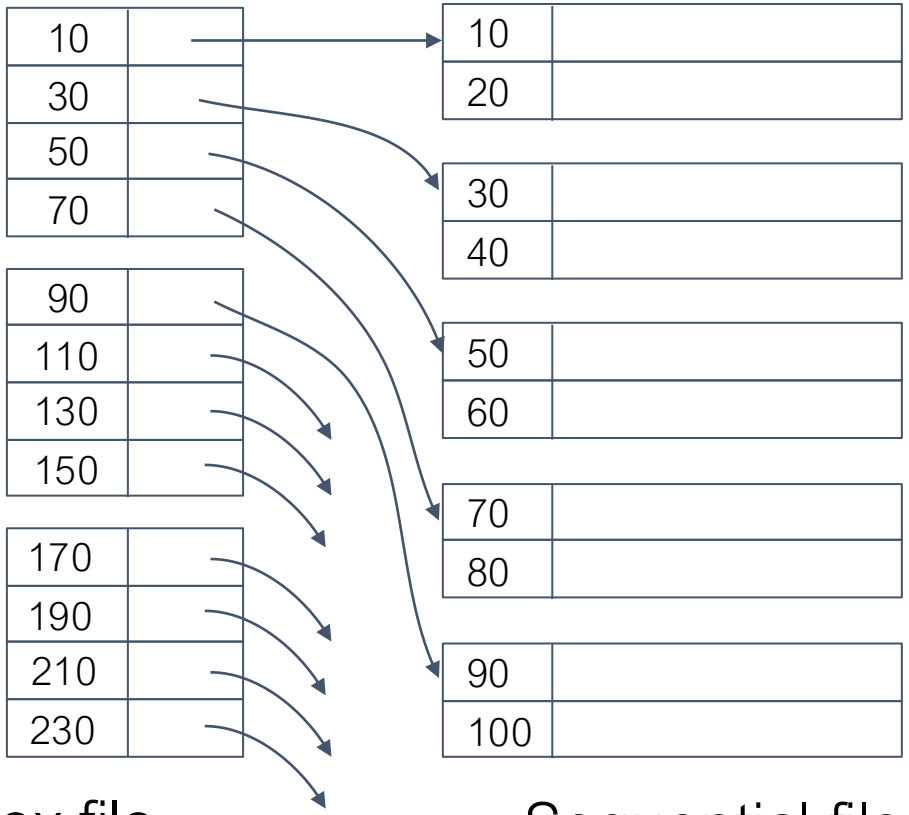
Dense index

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



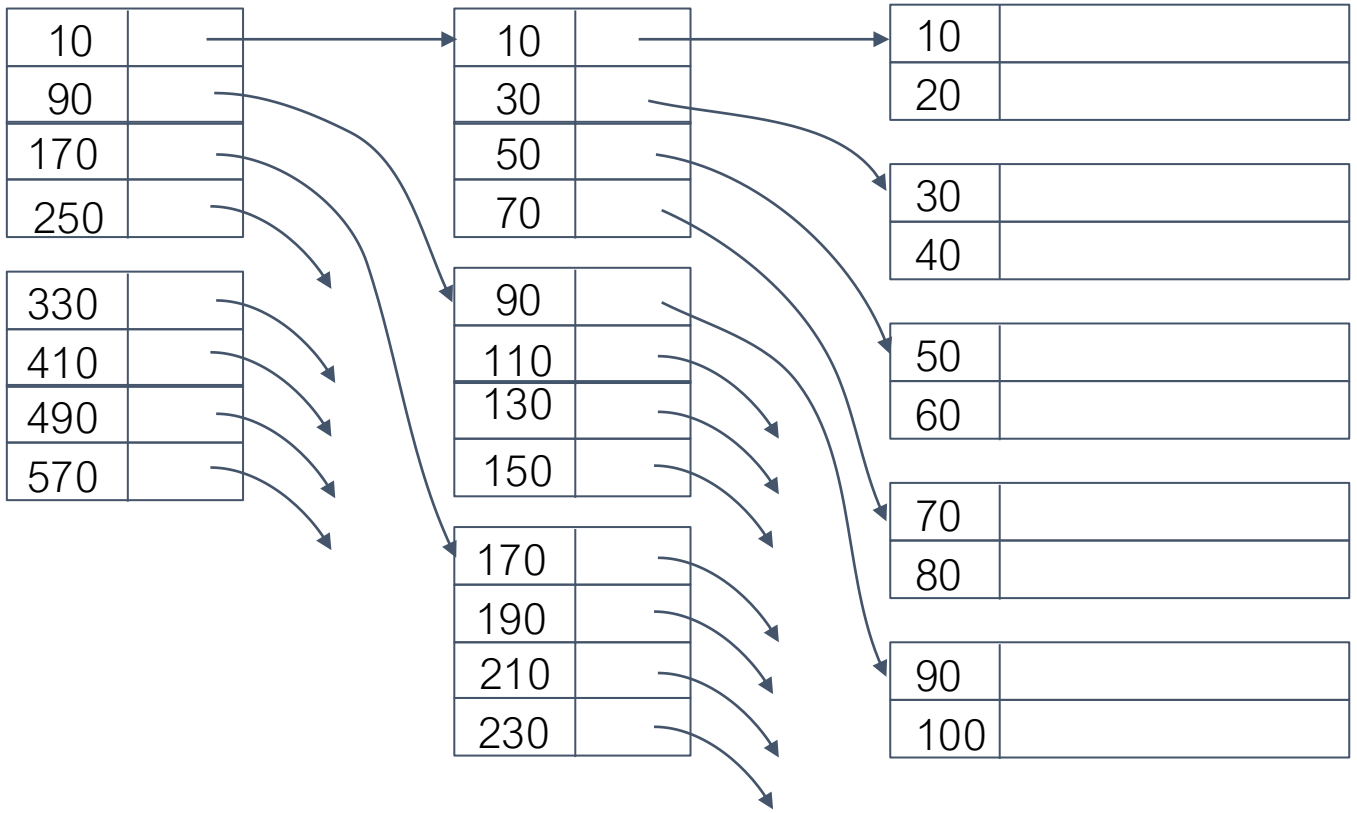
Sparse index

- 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

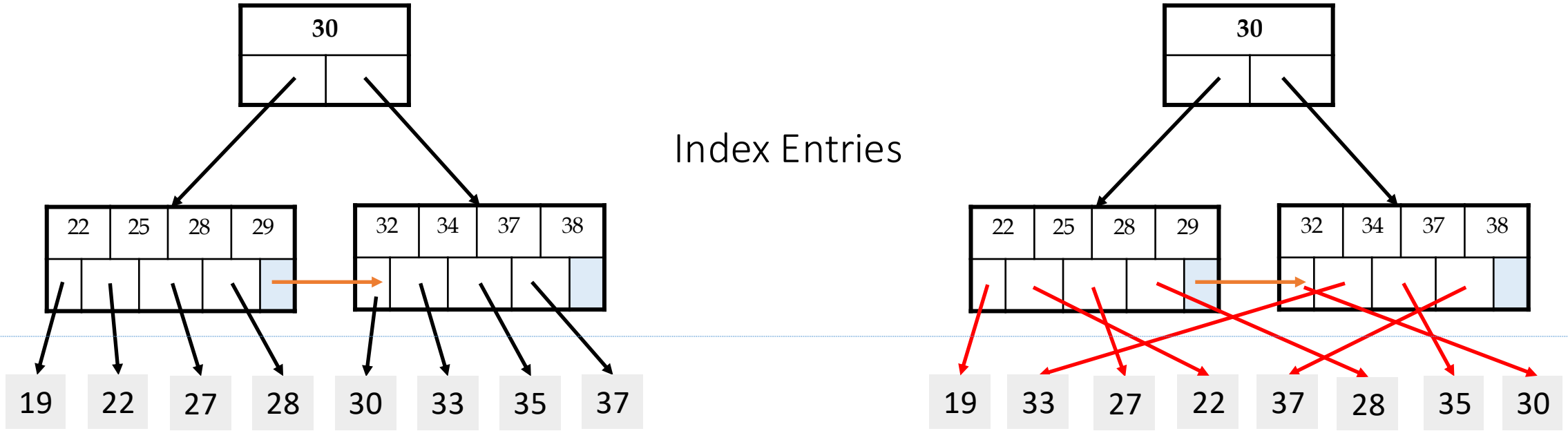


Multiple levels of index

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



Clustered vs. Unclustered Index



Clustered

Unclustered

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!

Practice Question

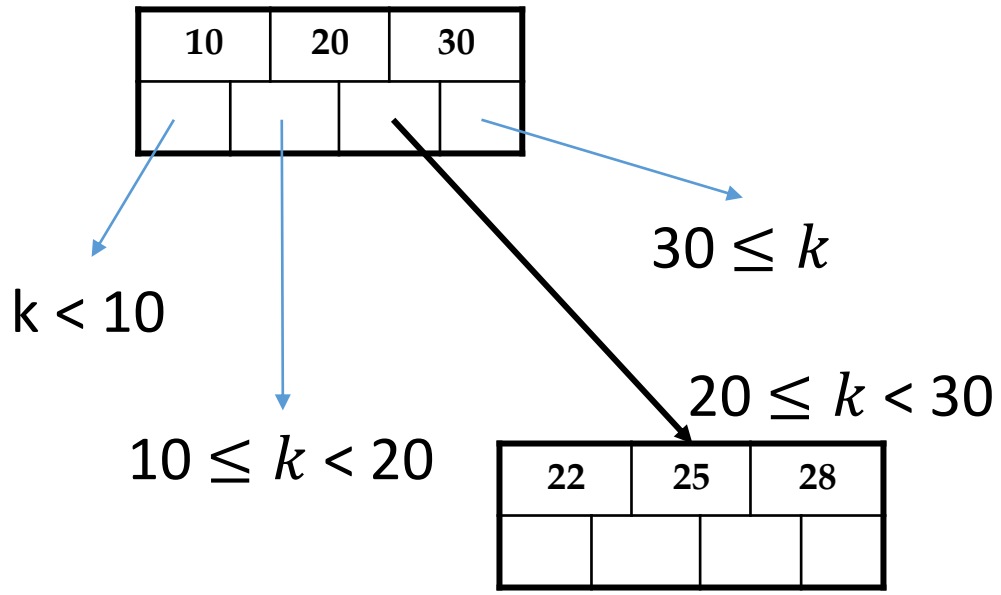
cs4440_sp25_midterm

4. How many blocks?

B+ Tree

B+ Tree Basics

Non-leaf or *internal* node



Parameter ***d*** = the degree

Each non-leaf ("interior") node has node has ≥ d and ≤ 2d keys*

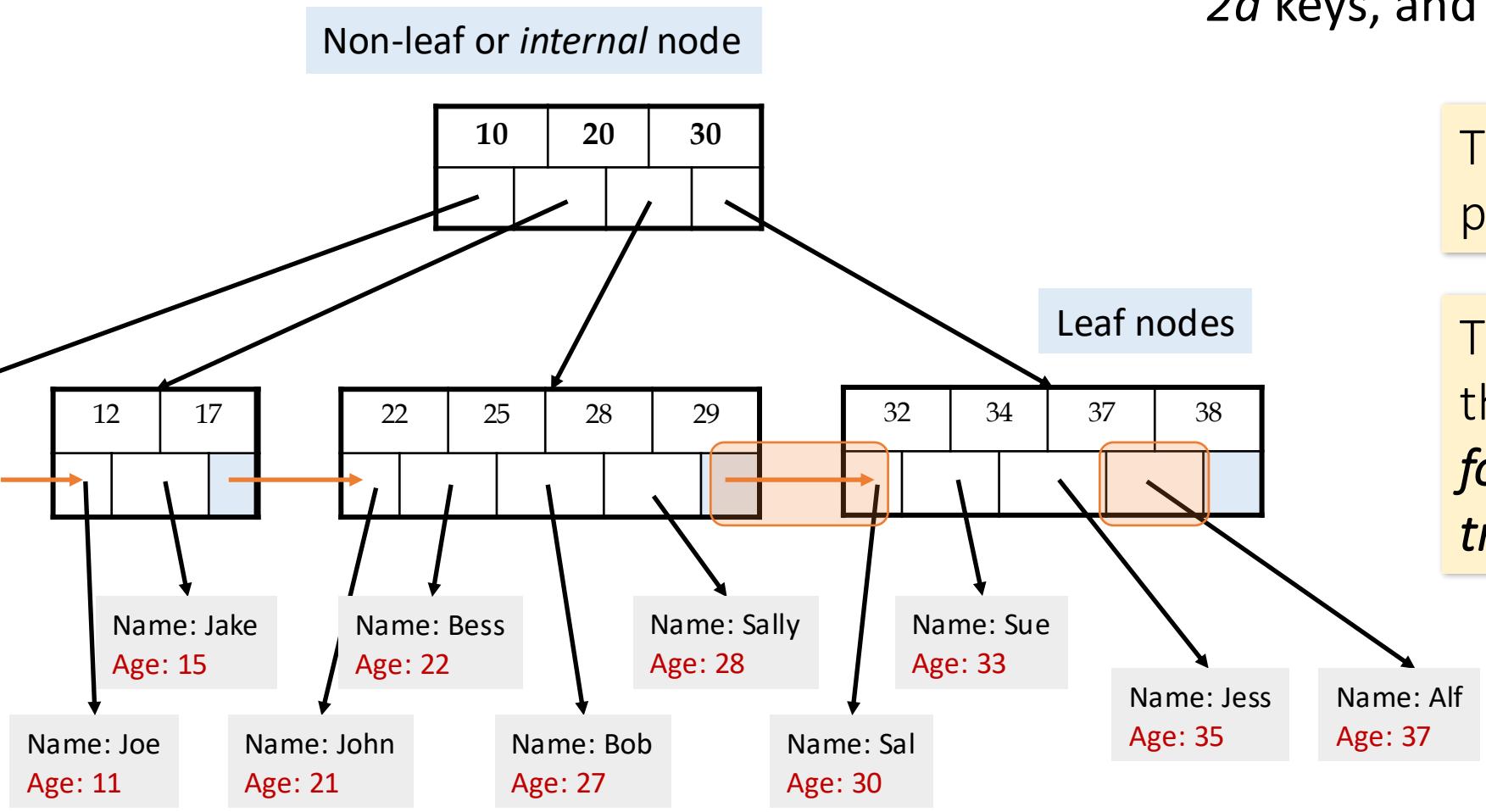
The *k* keys in a node define *k+1* ranges

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

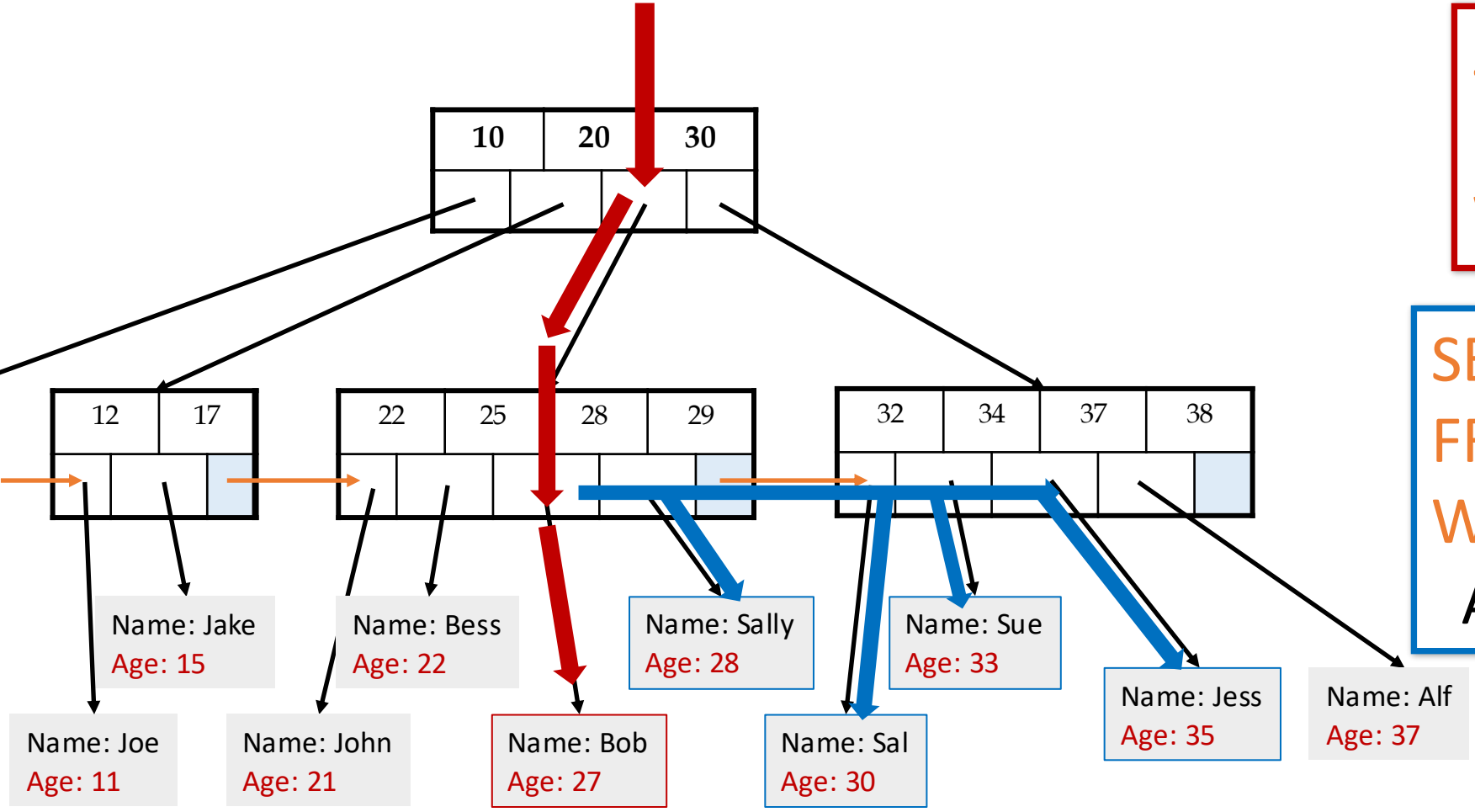
Leaf nodes also have between d and *2d* keys, and are different in that:



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 ~= 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 $\mathbf{f}^{\mathbf{h}}$ pages and has $\mathbf{f}^{\mathbf{h-1}}$ 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

cs4440_sp25_midterm.pdf

- 5. Indexing Problem
  - 5.2. B+ Trees

Hashing

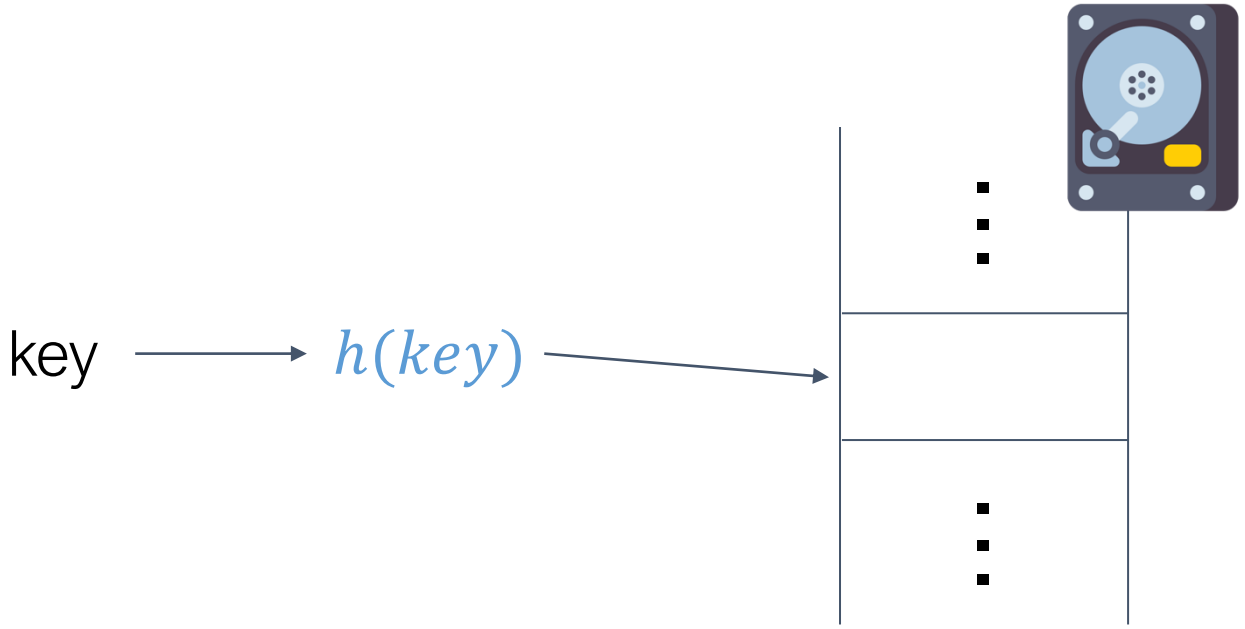
Hash table basics

- A hash function *h* takes a key and returns a block number from 0 to B - 1
- Blocks contain records and are stored in **secondary storage**

Space complexity: O(n)

Time Complexity:

- Average: O(1)
- Worst: O(n)



Hash table: Design Decisions

Hash Function

- How to map a large key space into a smaller domain of array offsets
- Trade-off between fast execution vs. collision rate

Hashing Scheme

- How to handle key collisions after hashing
- Trade-off between allocating a large hash table vs. extra steps to location/insert keys
- Static vs dynamic schemes

Slides adapted from CMU CS 15-445/645 by Andy Pavlo

Static hash table

- The number of buckets is fixed
- Often used during query execution because they are faster than dynamic hashing schemes.
- If the DBMS runs out of storage space in the hash table, it has to rebuild a larger hash table (usually 2x) from scratch, which is very expensive!

Examples

- Linear Probing Hashing
- Cuckoo Hashing

Dynamic hash table

The previous hash tables require the DBMS to know the number of elements it wants to store; otherwise it needs to rebuild the table to resize

Dynamic hash tables incrementally resize the hash table on demand without needing to rebuild the entire table at once.

- **Key Trade-off:** Eliminates massive rebuild costs in exchange for more complex maintenance overhead during normal operations

Examples:

- Chained Hashing
- Extensible Hashing
- Linear Hashing

Practice Question

cs4440_sp25_final.pdf

- 1. Hashing

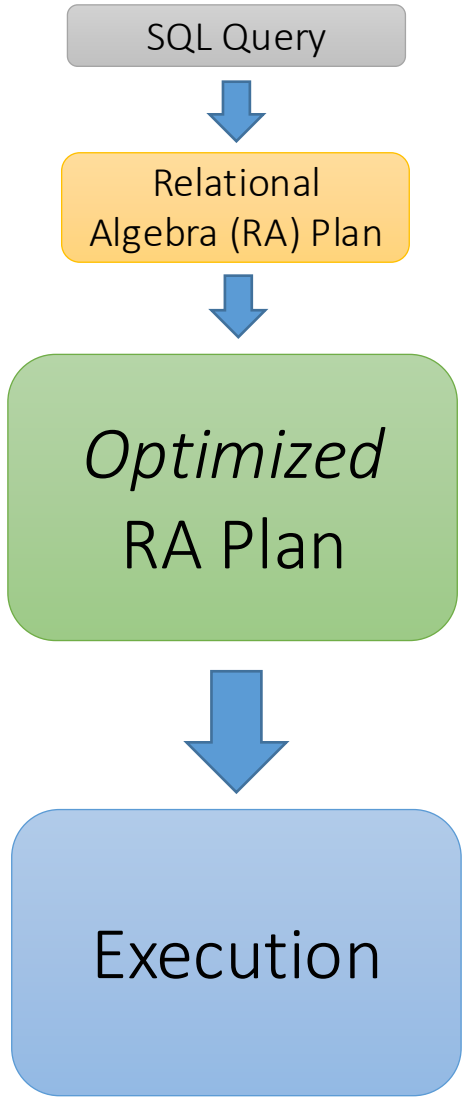
* We will not test on specific hashing schemes

Query Optimization

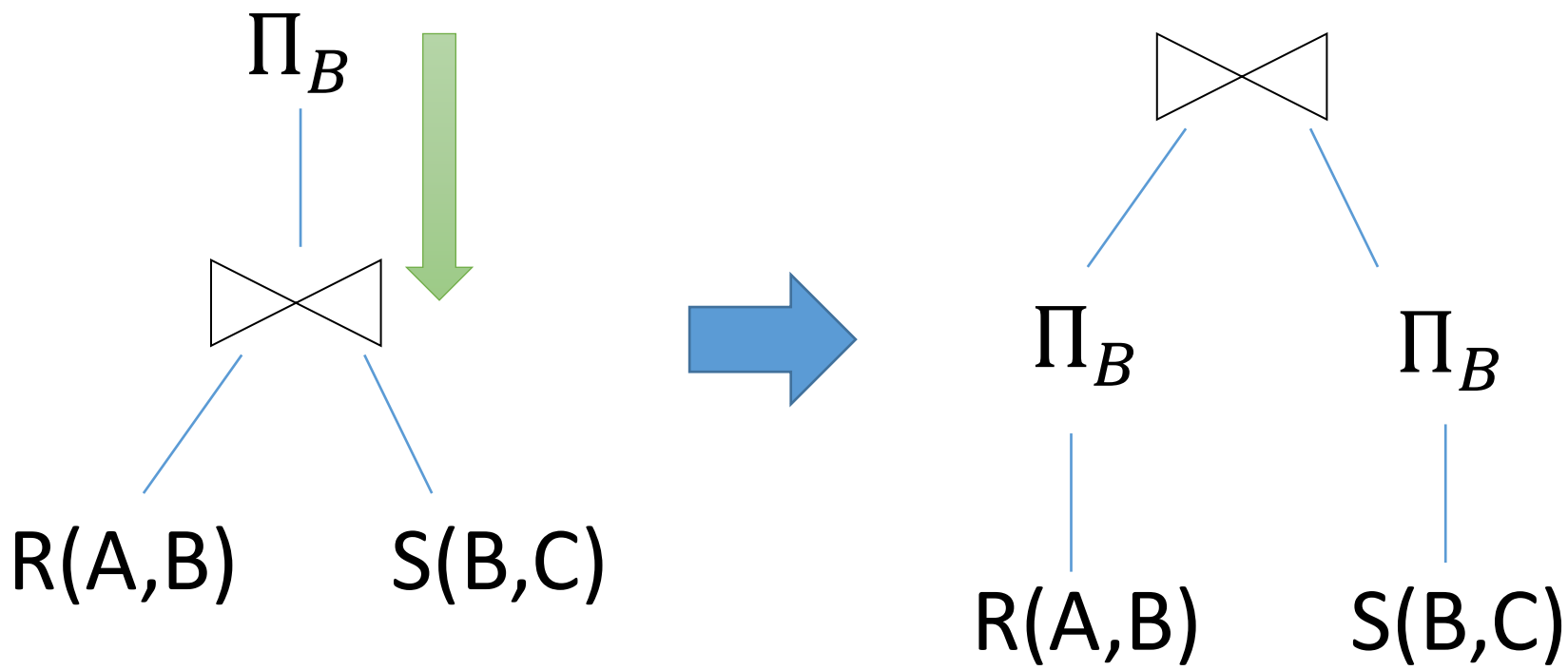
Logical vs. Physical Optimization

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

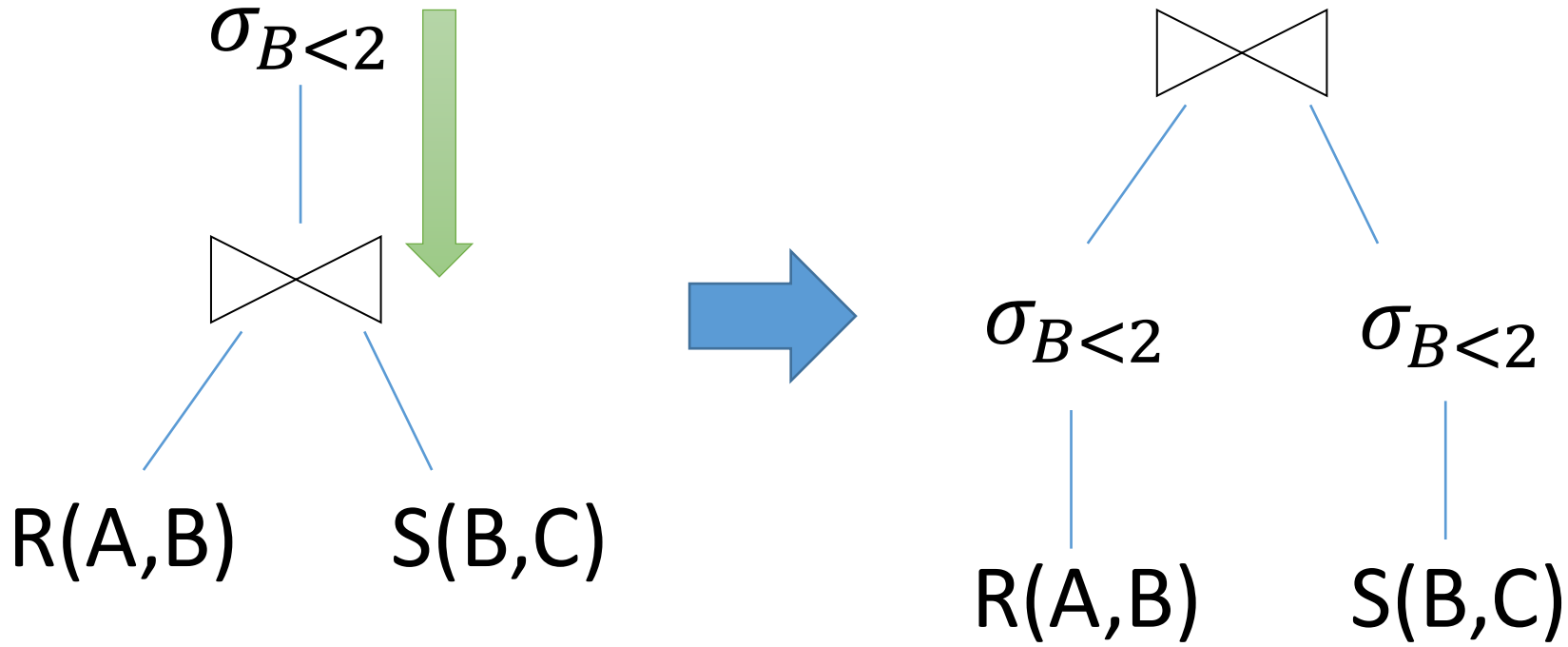
- **Physical optimization:**
  - Find algorithm with lowest IO cost to execute our plan
  - *Intuition: Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)*



Logical Optimization: “Pushing down” projection



Logical Optimization: “Pushing down” selection



Commutative and associative laws

$$R \bowtie S = S \bowtie R$$
$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

- Same holds for $\times$, $\cup$, $\cap$
- Holds for both set and multi-set semantics

Example ($R \bowtie S \bowtie T$):

- $|R| = 1{,}000$ tuples
- $|S| = 10{,}000$ tuples
- $|T| = 100$ tuples
- $|R \bowtie S| \approx 50{,}000$ tuples (many matches)
- $|S \bowtie T| \approx 500$ tuples (few matches)

Which join order is better?

$(R \bowtie S) \bowtie T$

$R \bowtie (S \bowtie T)$

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*

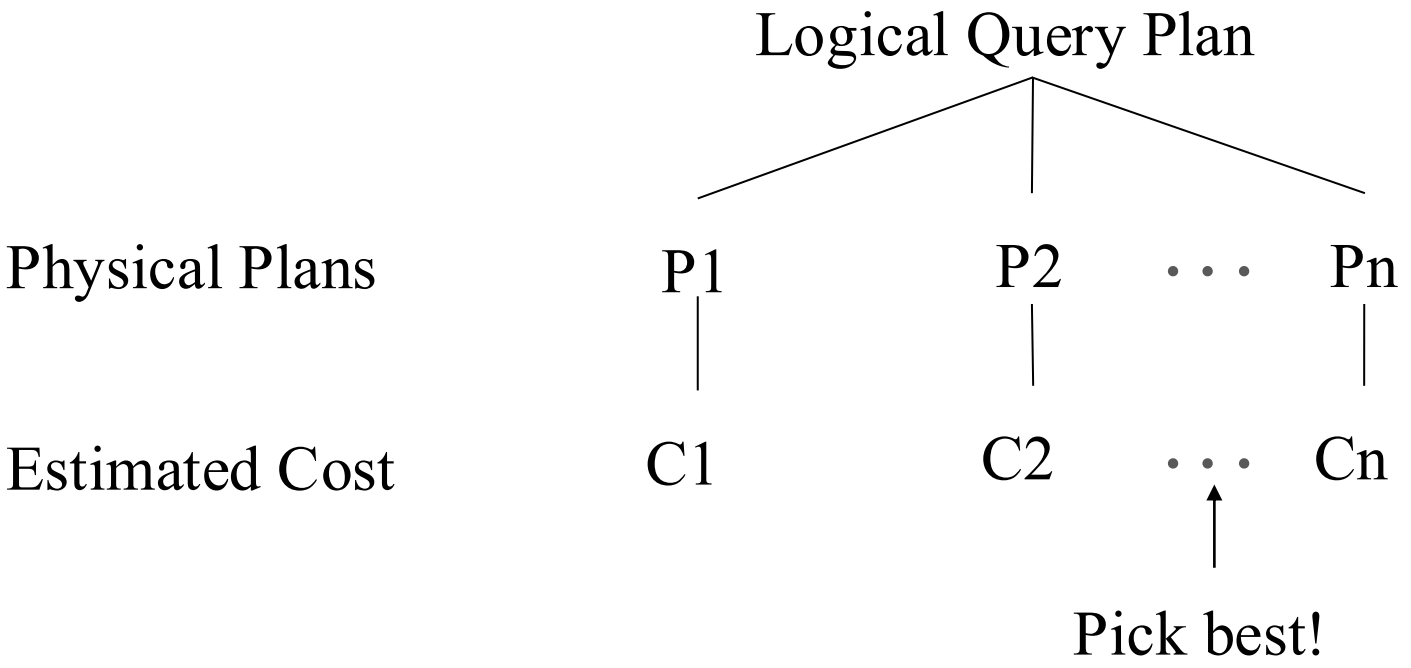
This simple set of tools allows us to greatly improve the execution time of queries by optimizing RA plans!

Practice Question

cs4400_sp25_final

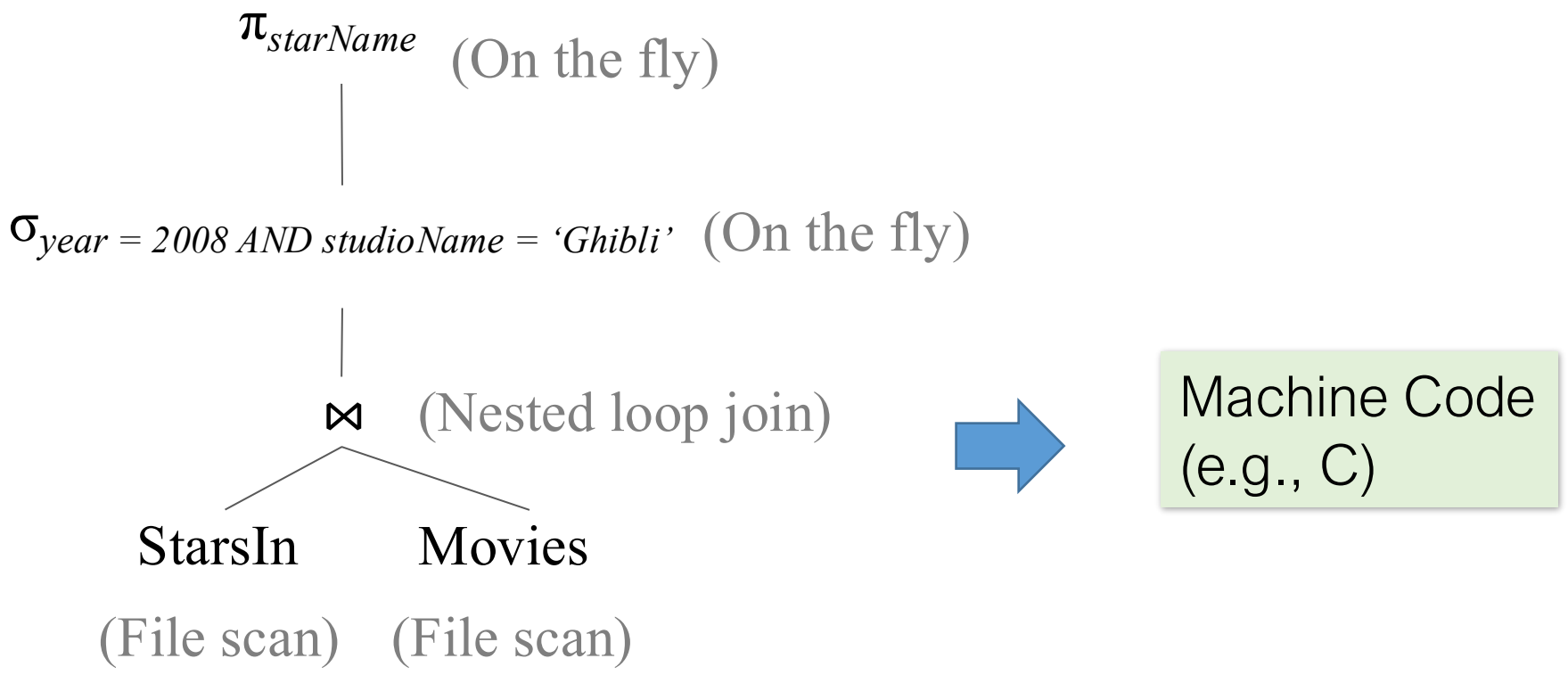
2. Logical Optimization

Select physical query plan



In general, there can be many possible physical plans

Query execution



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: $V(R \bowtie S, X) = V(R, X)$

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

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

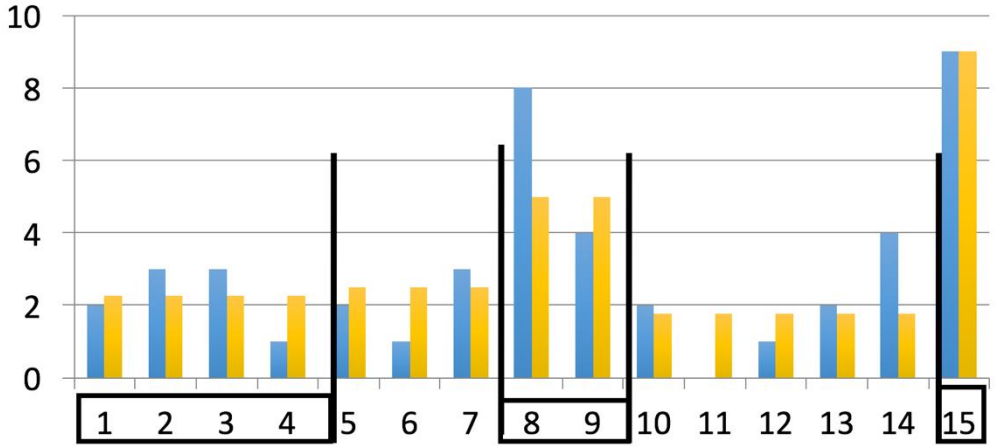
For each pair (r, s), we know that the Y-value of S is one of the Y-values of R by containment of value sets, so the probability of r having the same Y-value is 1/V(R,Y)

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

Histogram types

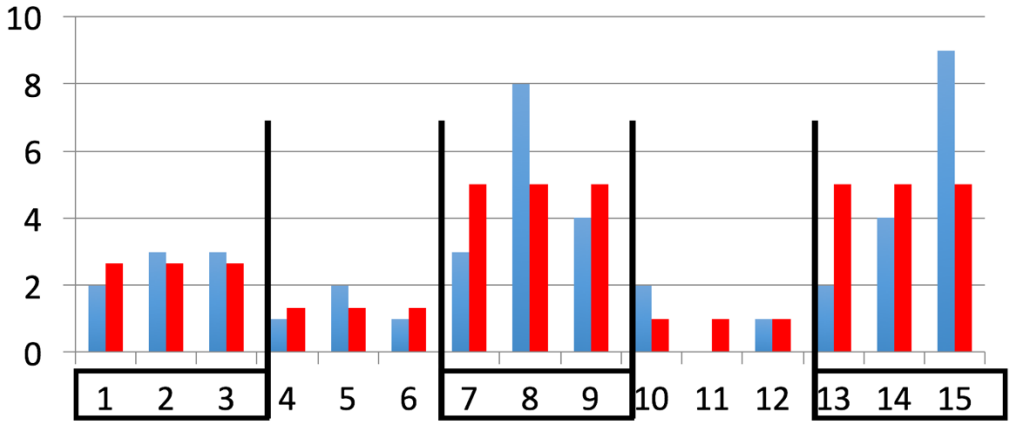
Equi-depth

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



Equi-width

All buckets roughly the same width



EMS and Join Algorithms

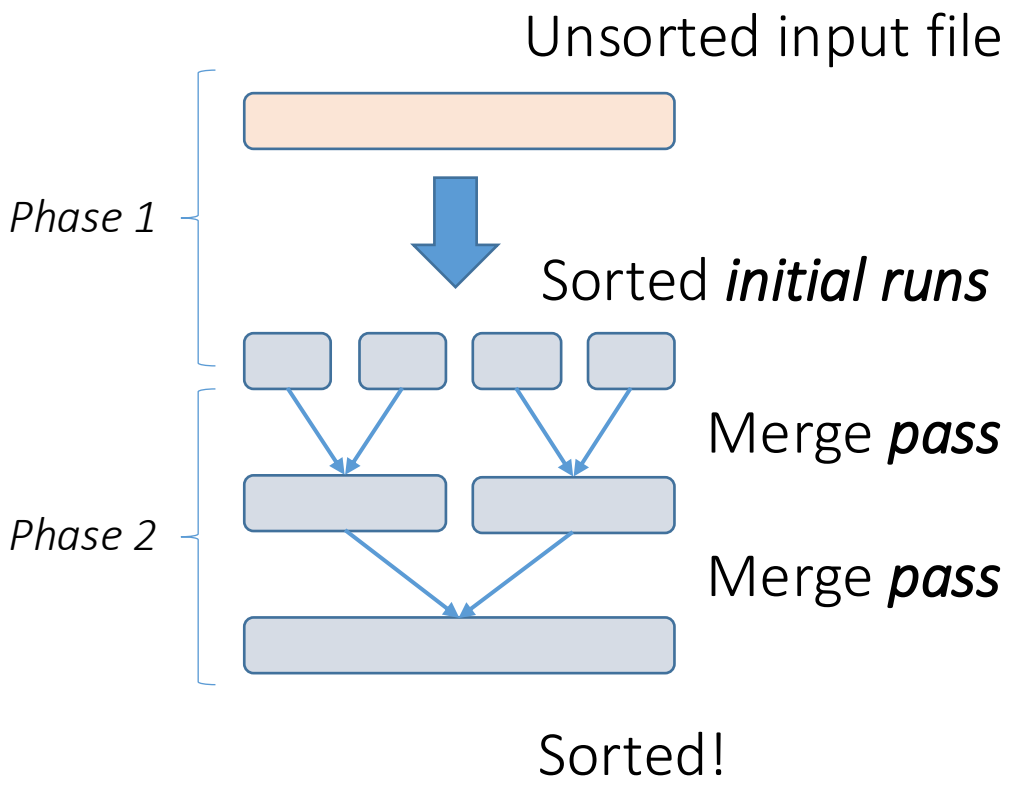
External Merge Sort Algorithm

- ***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

- Optimizations on Basic algorithm
  - (B+1)-length initial runs
  - B-way merging

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

Starting with runs of length 1

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

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

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

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:	$2N(\lceil \log_B \lceil \frac{N}{B+1} \rceil \rceil + 1)$	***Phase 1:*** Initial runs of length B+1 are created • There are $\lceil \frac{N}{B+1} \rceil$ of these • The IO cost is 2N ***Phase 2:*** We do passes of B-way merge until fully merged • Need $\lceil \log_B \lceil \frac{N}{B+1} \rceil \rceil$ passes • The IO cost is 2N per pass

Join Algorithms: Overview

For $R \bowtie S$ *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 $R \bowtie S \; on \; A$:

```
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!

Cost:

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

```
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)
```

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

Cost:

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

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**

Basic SMJ: Total cost

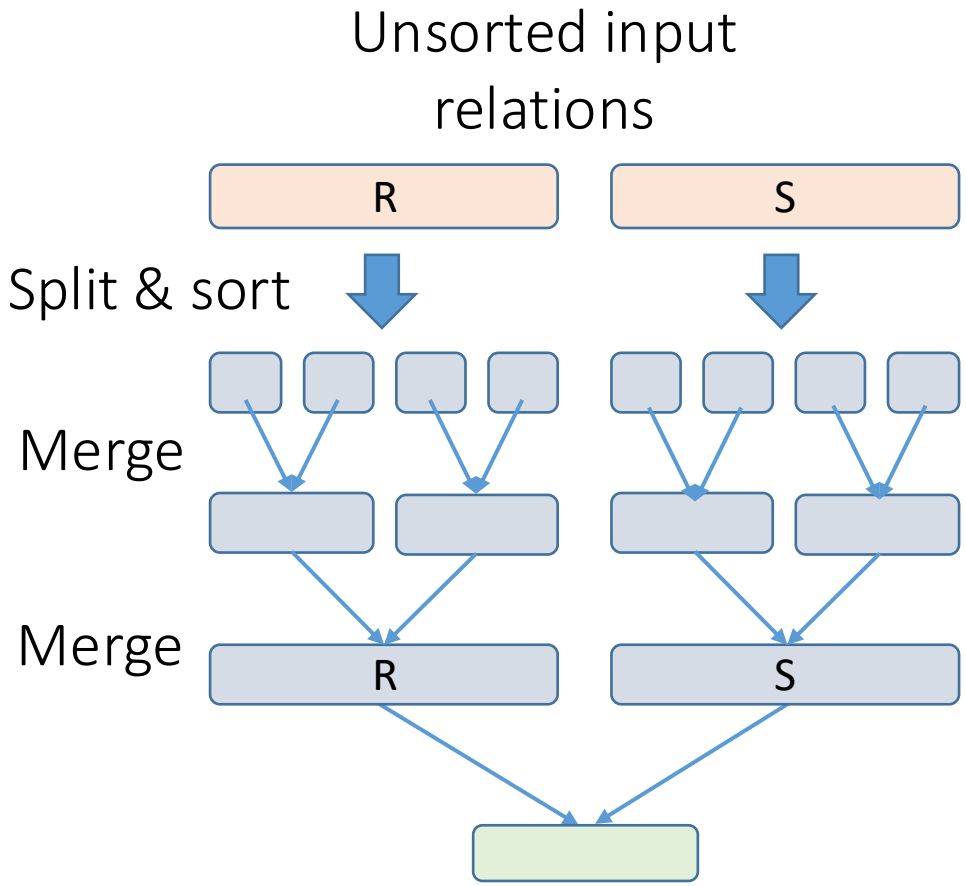
Cost of SMJ is **cost of sorting** R and S…

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**

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

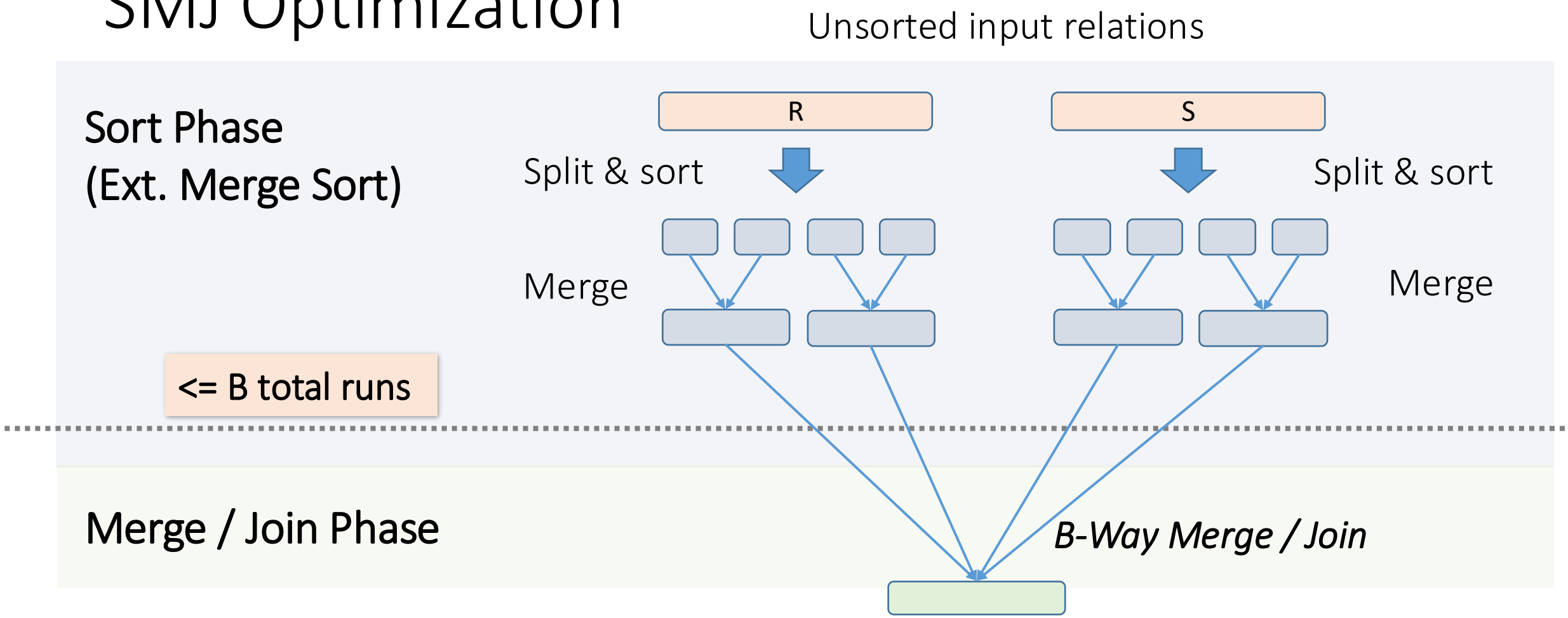


Recall: $\text{Sort(N)} \approx 2N\left(\left\lceil \log_B \frac{N}{B+1} \right\rceil + 1\right)$

Note: this is WITHOUT the repacking optimization

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

SMJ Optimization



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

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.

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

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

cs4400_sp25_final

3. Query Optimization