CS 564 Final Review

The Best Of Collection (Master Tracks), Vol. 2

Course Announcements

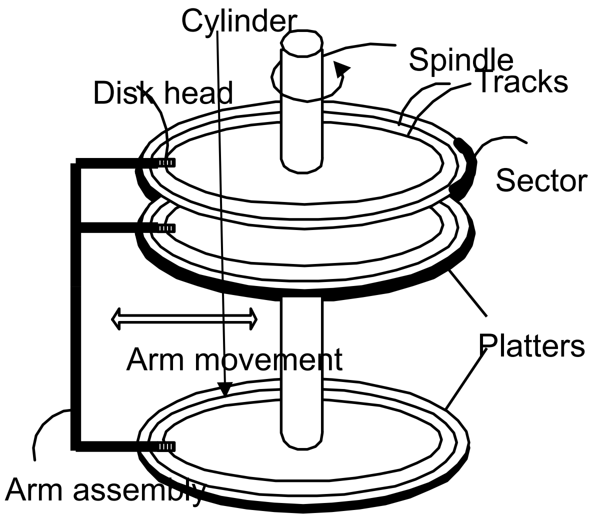
- Last day for course evaluations -please fill out ☺
  - We want your feedback to improve the course!
  - Tell us what you liked and didn’t!
  - I take **every** evaluation very seriously.

- Project 4 due today—No late days!

High-Level: Lectures 9 - 11

- The ***buffer*** & simplified filesystem model
- Shift to ***IO Aware*** algorithms
- The ***external merge algorithm***

High-level: Disk vs. Main Memory



Disk:

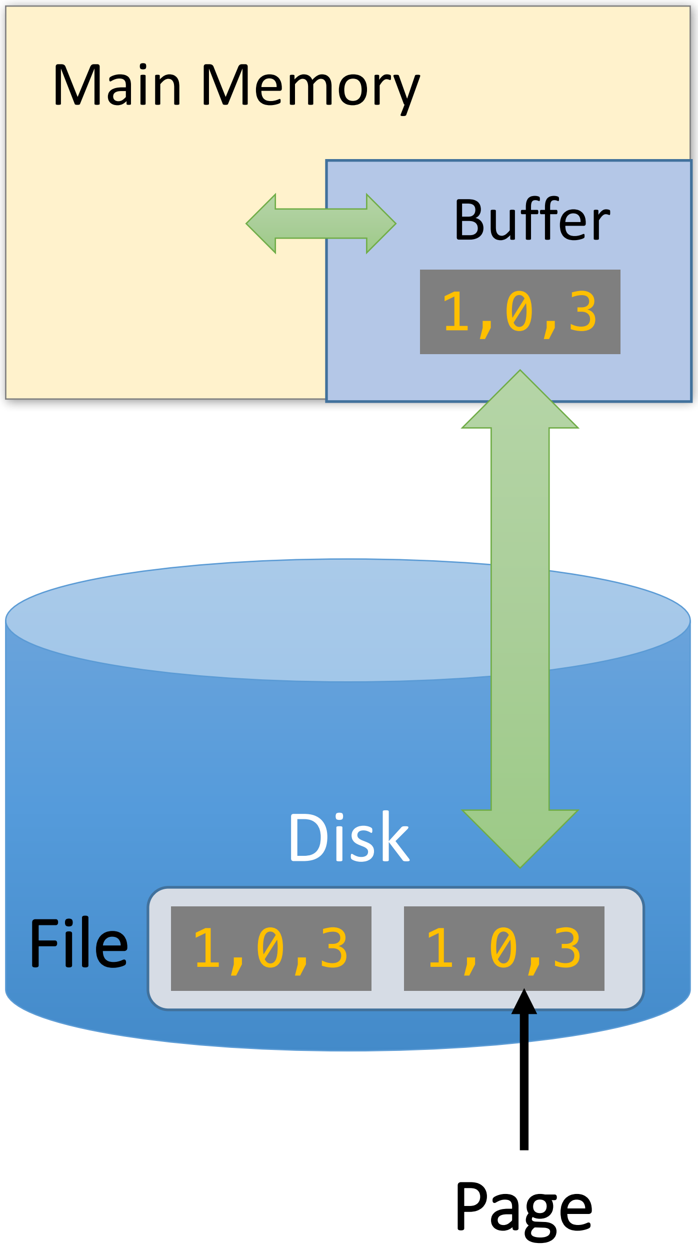
- ***Slow:*** Sequential *block* access
  - Read a blocks (not byte) at a time, so sequential access is cheaper than random
  - **Disk read / writes are expensive!**
- ***Durable:*** We will assume that once on disk, data is safe!
- ***Cheap***

Random Access Memory (RAM) or Main Memory:

- ***Fast:*** Random access, byte addressable
  - ~10x faster for sequential access
  - ~100,000x faster for random access!
- ***Volatile:*** Data can be lost if e.g. crash occurs, power goes out, etc!
- ***Expensive:*** For $100, get 16GB of RAM vs. 2TB of disk!

The Buffer

- A **buffer** is a region of physical memory used to store *temporary data*
  - *Key Idea:* Reading / writing to disk is SLOW, need to cache data in main memory
  - Can **read** into buffer, **flush** back to disk, **release** from buffer
- DBMS manages its own buffer for various reasons (better control of eviction policy, force-write log, etc.)
- We use a simplified model:
  - A ***page*** is a fixed-length array of memory; **pages are the unit that is read from / written to disk**
  - A ***file*** is a variable-length list of pages on disk



IO Aware

- Key idea: Reading from / writing to disk- e.g. ***IO operations-*** is ***thousands*** of times slower than any operation in memory
  - → We consider a class of algorithms which try to minimize IO, and *effectively ignore cost of operations in main memory*

"IO aware" algorithms!

External Merge Algorithm

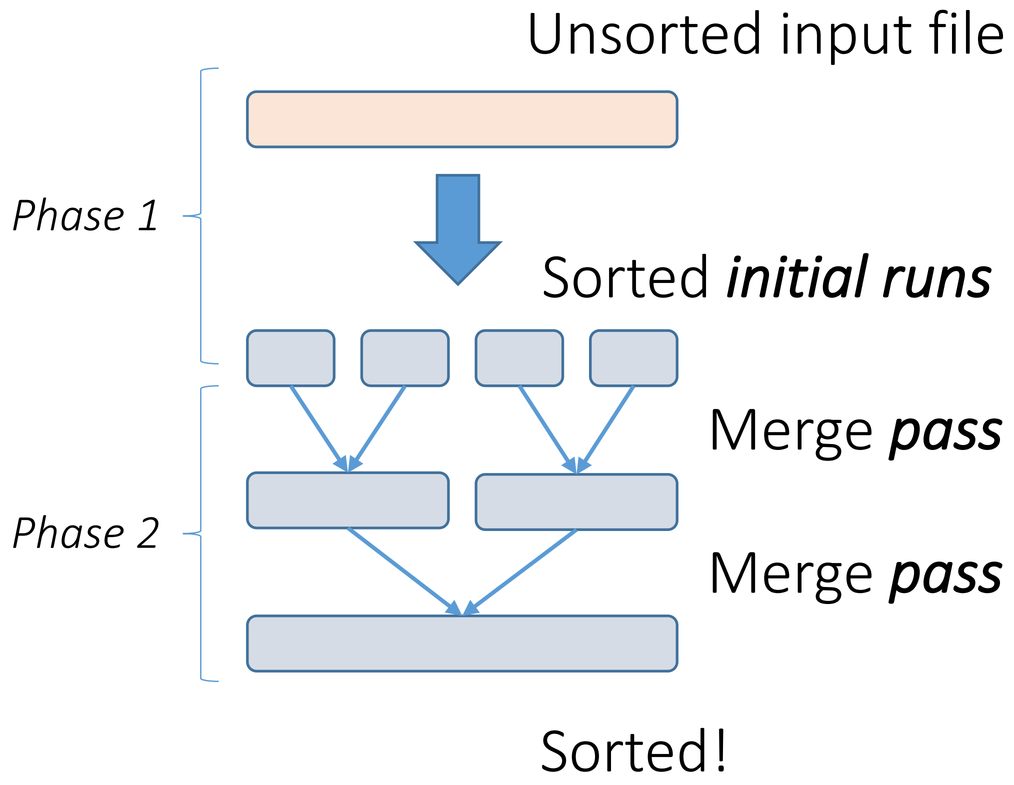
- ***Goal:*** Merge sorted files that are much bigger than buffer
- ***Key idea:*** Since the input files are sorted, we always know which file to read from next!
- ***Details:***

Given:	***B+1*** buffer pages
Input:	***B*** sorted files, $\boldsymbol{F_1,\ldots,F_B}$, where F_i has $P(F_i)$ pages
Output:	One merged sorted file
IO COST:	$\mathbf{2} * \sum_{i=1}^{B} \boldsymbol{P(F_i)}$ *(Each page is read & written once)*

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 Sort Algorithm

Given:	***B+1*** buffer pages	
Input:	Unsorted file of length ***N*** pages	
Output:	The sorted file	
IO COST:	$2N(\left\lceil \log_B \left\lceil \frac{N}{B+1} \right\rceil \right\rceil + 1)$	***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

Repacking Optimization for Ext. Merge Sort

- ***Goal:*** Create larger initial runs
- ***Key Idea:*** Keep loading unsorted pages, writing out next-largest values, and "repacking" for as long as possible!
  - *Guaranteed to do at least as well as our previous method of loading & doing quicksort*
- ***IO Cost:*** On average, we will create initial runs of size **~2(B+1)**

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

High-Level: Lectures 12 - 14

- Indexes Part I: Basics
- B+ Trees
- Clustered vs. unclustered
- Hash Indexes

Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Where the *search key* could be any subset of fields, and does ***not*** need to be the same as *key of a relation*

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

Note this is the logical setup, not how data is actually stored!

By_Author_Title_Index

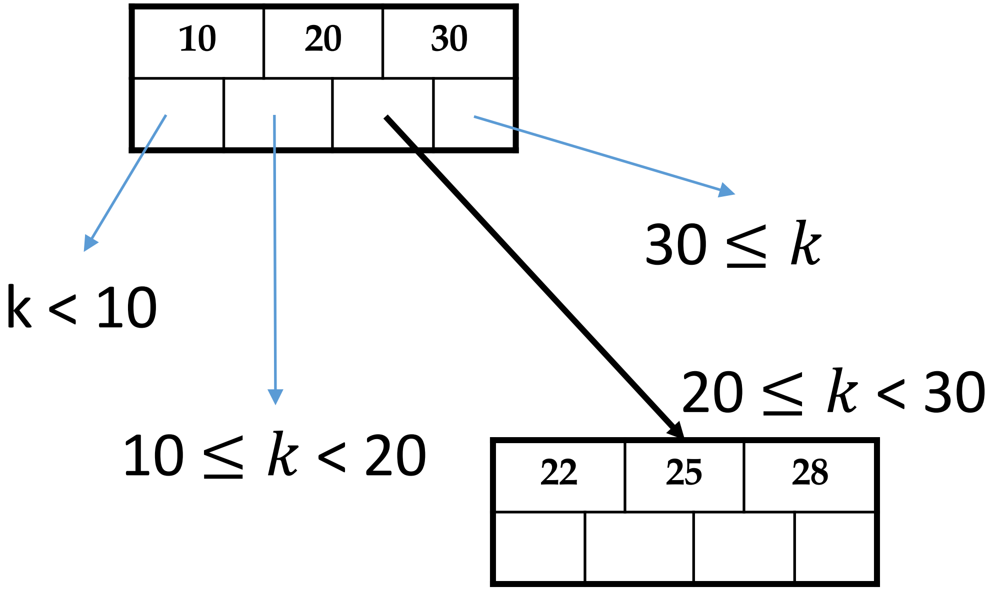
Author	Title	BID
Dostoyevsky	Crime and Punishment	002
Tolstoy	Anna Karenina	003
Tolstoy	War and Peace	001

An index is <u>**covering**</u> *for a specific query* if the index contains all the needed attributes

B+ Tree Basics

Parameter ***d*** = the order

Non-leaf or *internal* node



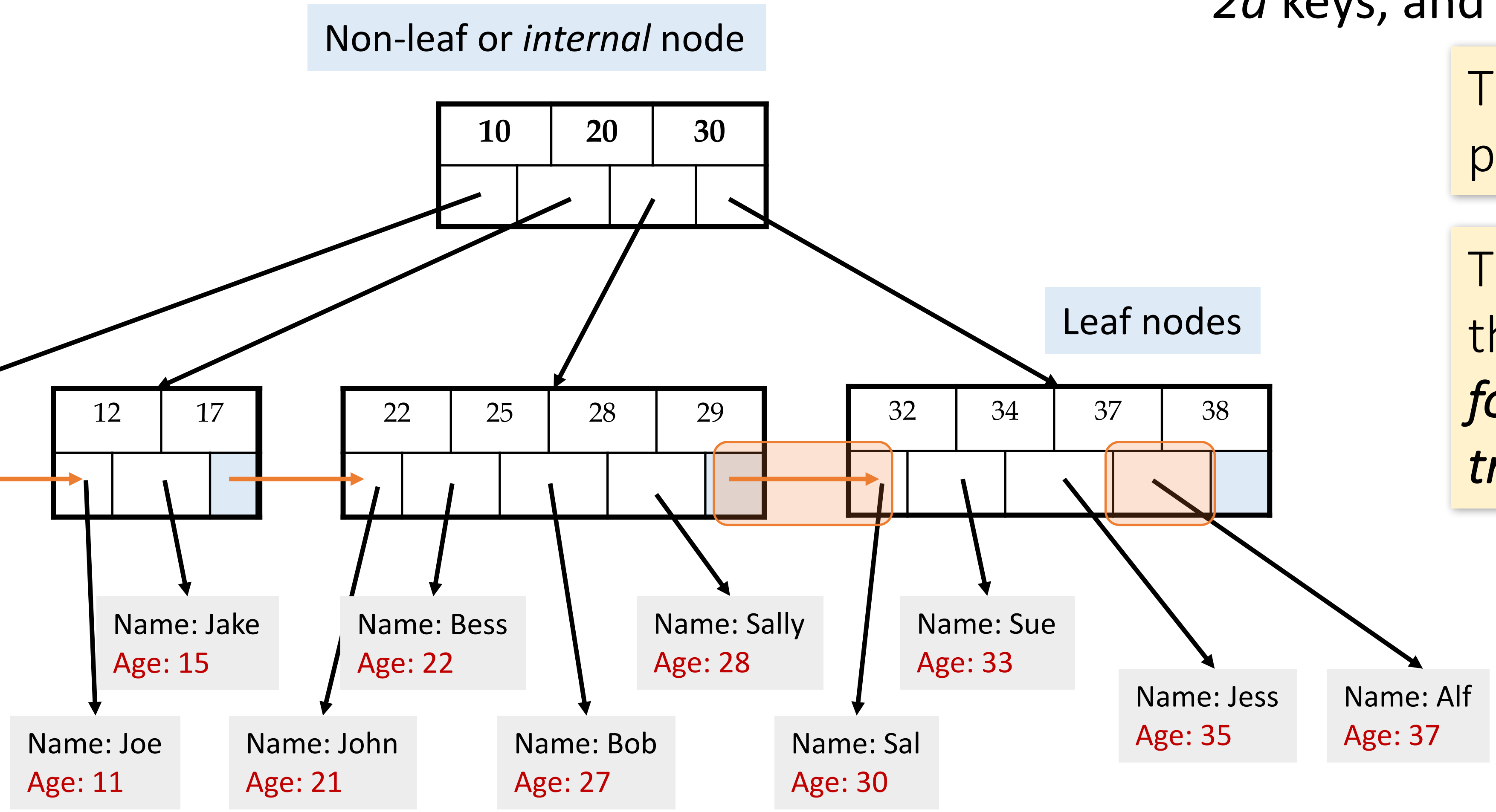
Each *non-leaf ("interior")* ***node*** has ≥ d and ≤ 2d ***keys****

The *n* keys in a node define *n+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

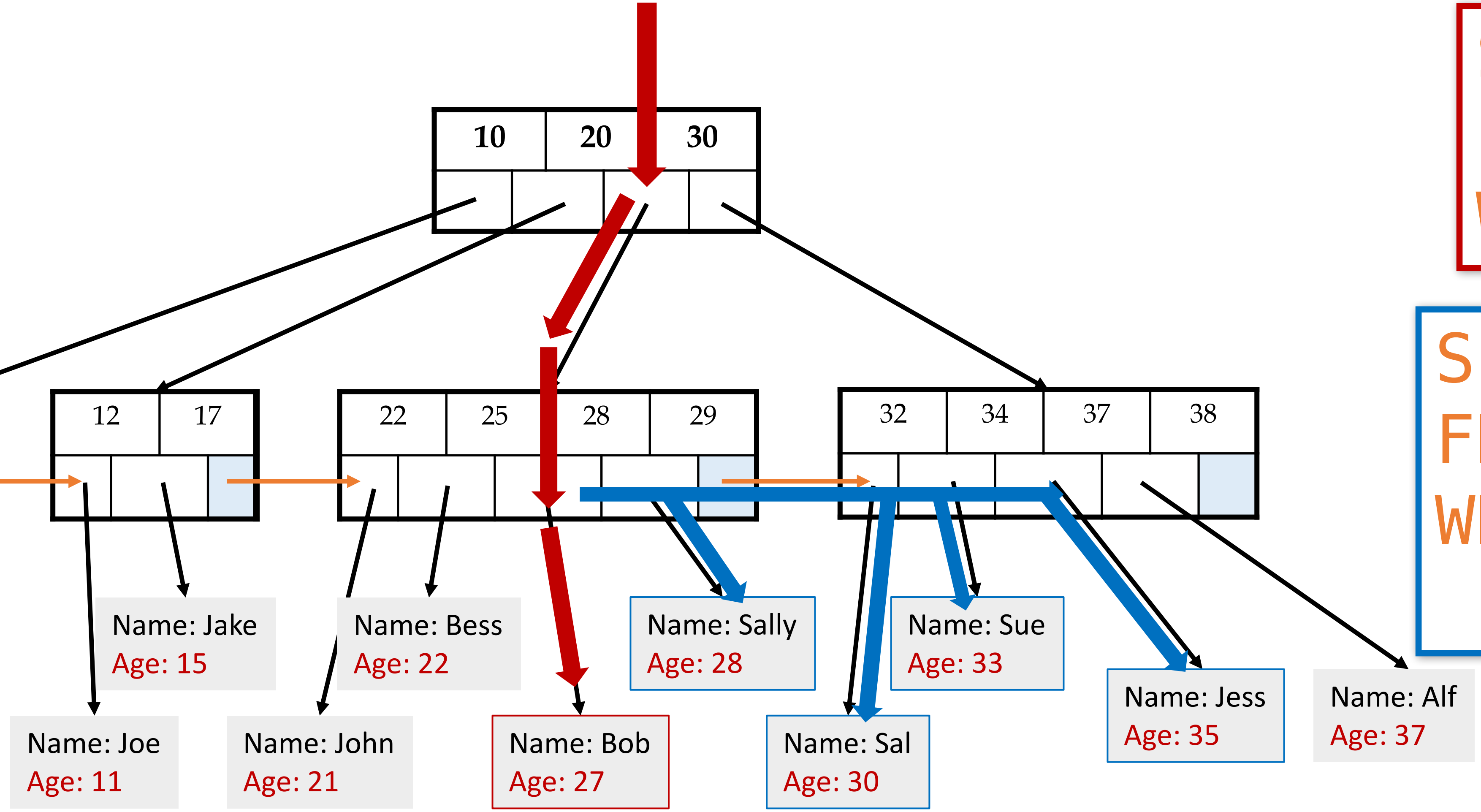


Leaf nodes also have between *d* and *2d* keys, and are different in that:

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 Range Search

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:

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

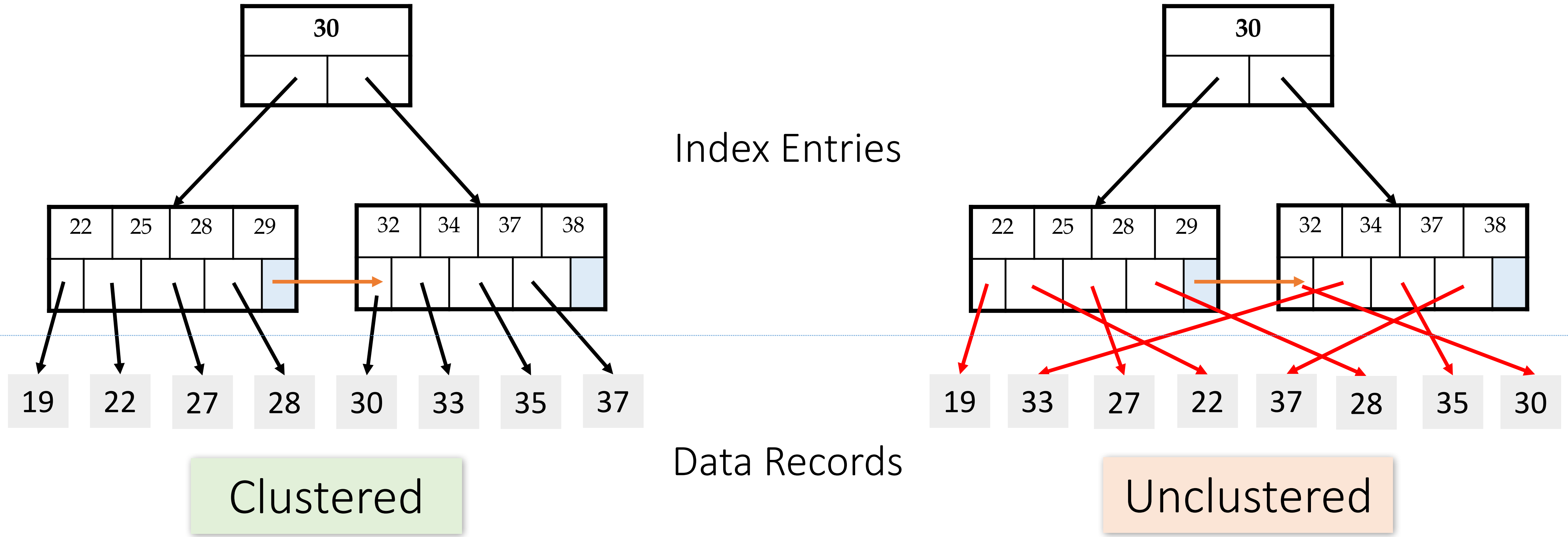
Note that we will often approximate f as constant across nodes!

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

B+ Tree Range Search

Given:	• Parameter ***d*** • Fill-factor ***F*** • ***B*** available pages in buffer • A B+ Tree over ***N*** pages • **f** is the fanout [d+1,2d+1]	
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

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!

Hash Indexes

- A hash index is:
  - good for equality search
  - not so good for range search (use tree indexes instead)

- An *ideal* hash function must be uniform: each bucket is assigned the same number of key values
- A *bad* hash function maps all search key values to the same bucket

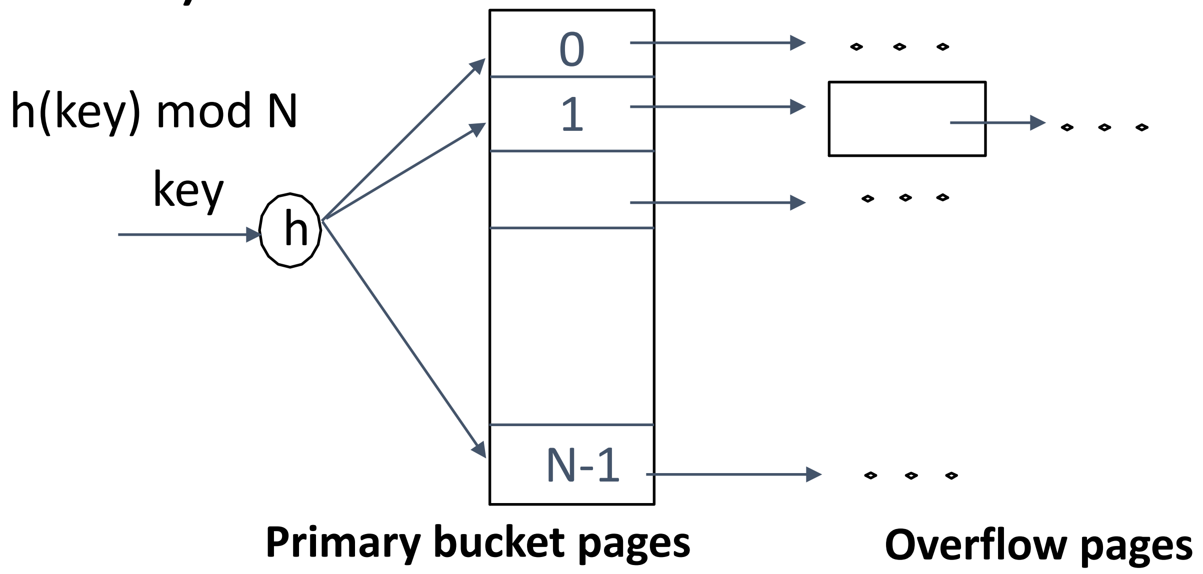
Hash Indexes

- A hash index is:
  - good for equality search
  - not so good for range search (use tree indexes instead)

- An *ideal* hash function must be uniform: each bucket is assigned the same number of key values
- A *bad* hash function maps all search key values to the same bucket

Static Hashing

- # primary bucket pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h(*k*) mod N** = bucket to which data entry with key *k* belongs. (N = # of buckets)



Extendible Hashing

See examples in L14!

- **Extendible hashing** is a type of *dynamic* hashing
- It keeps a directory of pointers to buckets
- On overflow, it reorganizes the index by doubling the directory (and not the number of buckets)

High-Level: Lectures 16-17

- Projection and Selection
- Join Algorithms:
  - Nested Loop Join Variants: NLJ, BNLJ, INLJ
  - SMJ
  - Hash Join

Selection

access path = way to retrieve tuples from a table

- **File Scan**
  - scan the entire file
  - I/O cost: O(N), where N = #pages
- **Index Scan:**
  - use an index available on some predicate
  - I/O cost: it varies depending on the index

Index Scan Cost

I/O cost for index scan

- Hash index: O(1)
  - but we can only use it with equality predicates
- B+ tree index: $O(\log_F N) + X$
  - X depends on whether the index is clustered or not:
    - *unclustered*: X = # selected tuples
    - *clustered*: X = (#selected tuples)/ (#tuples per page)

Index Matching

- We say that an index *matches* a selection predicate if the index can be used to evaluate it
- Consider a conjunction-only selection. An index matches (part of) a predicate if
  - Hash: only equality operation & the predicate includes *all* index attributes
  - B+ Tree: the attributes are a prefix of the search key (any ops are possible)

Projection

Simple case: **`SELECT`** `R.a, R.d`

- scan the file and for each tuple output R.a, R.d

Hard case: **`SELECT DISTINCT`** `R.a, R.d`

- project out the attributes
- eliminate *duplicate tuples* (this is the difficult part!)

Projection: Sort-based

We can improve upon the naïve algorithm by modifying the sorting algorithm:

1. In Pass **0** of sorting, project out the attributes
2. In subsequent passes, eliminate the duplicates while merging the runs

Projection: Hash-based

2-phase algorithm:

- **partitioning**
  - project out attributes and split the input into B-1 partitions using a hash function *h*
- **duplicate elimination**
  - read each partition into memory and use an in-memory hash table (with a *different* hash function) to remove duplicates

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

→

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

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 ⋈ 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 R ⋈ 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

<u>Cost:</u>

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

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

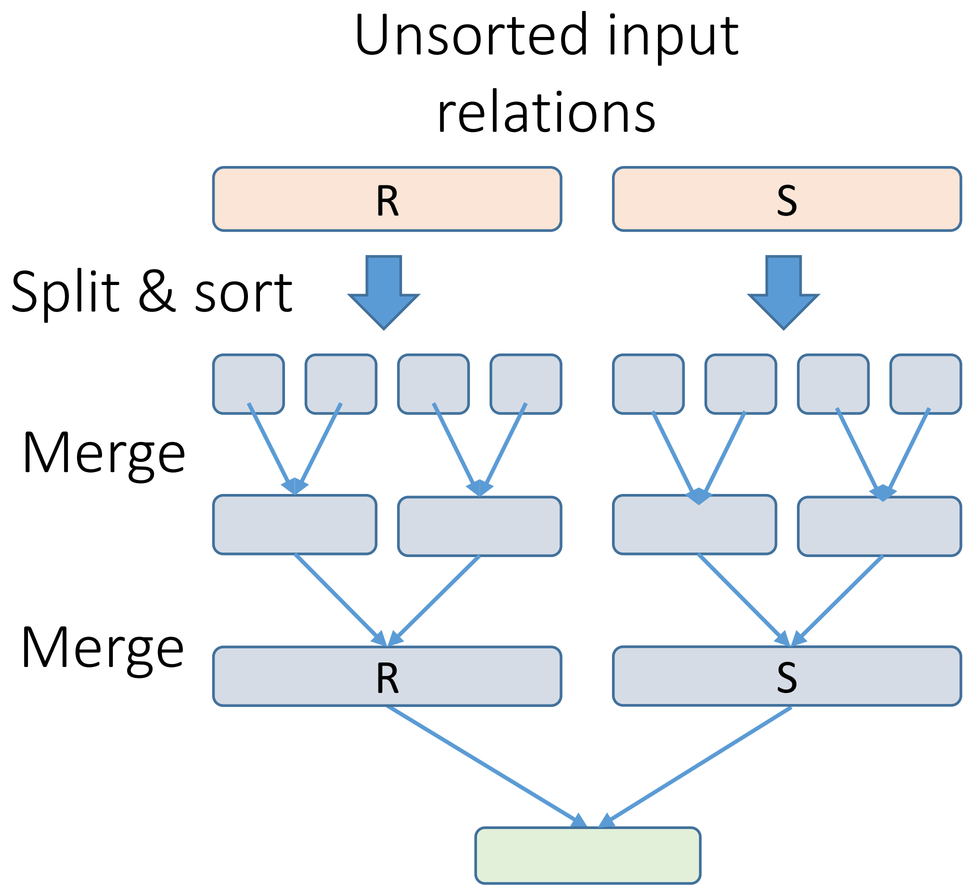
4. **Write out**

Sort Merge Join (SMJ)

- ***Goal:*** Execute R ⋈ S on A

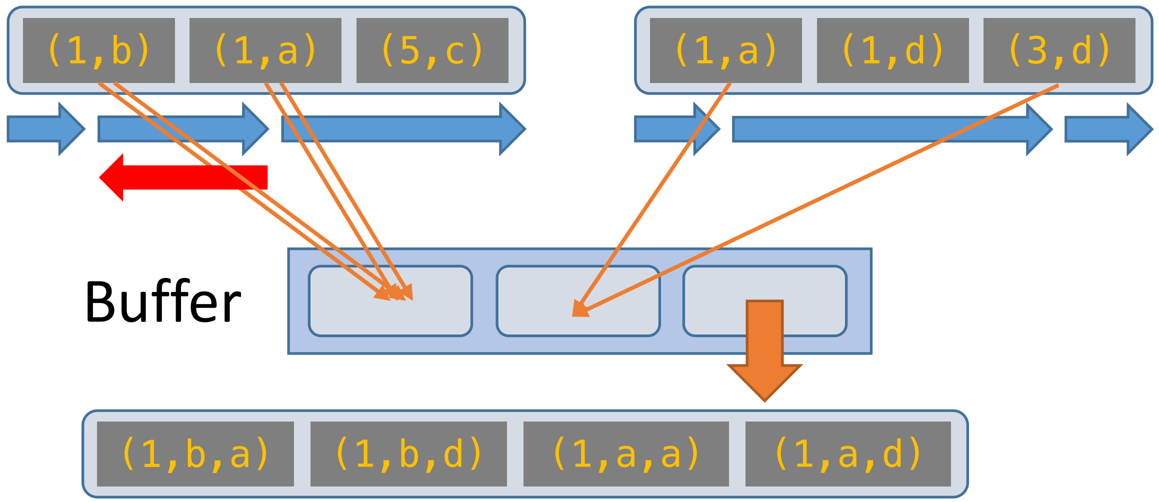
- ***Key Idea:*** We can sort R and S, then just scan over them!

- ***IO Cost:***
  - *Sort phase:* Sort(R) + Sort(S)
  - *Merge / join phase:* ~ P(R) + P(S) + OUT
    - ***Can be worse though- see next slide!***



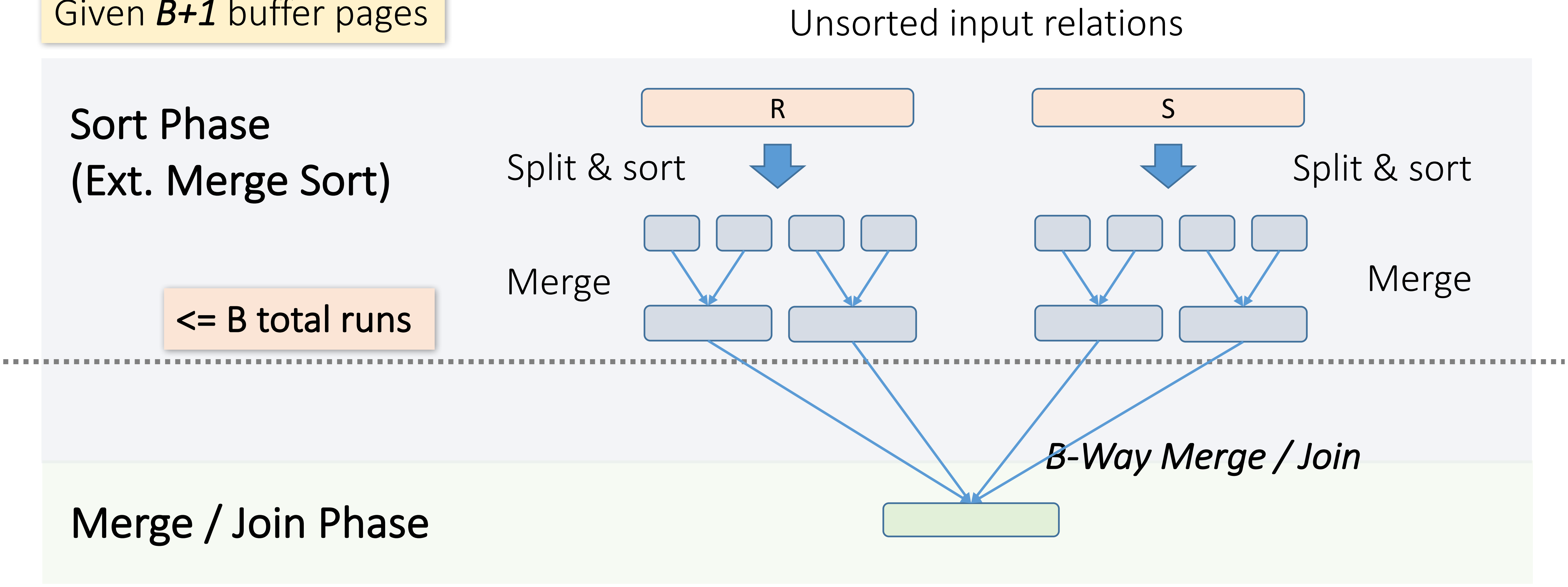
SMJ: Backup

- Without any duplicates:
  - We just scan over R and S once each → P(R) + P(S)
- However, if there are duplicates, we may have to ***back up*** and re-read parts of the file
  - In worst case have to read in P(R)*P(S)!
  - In worst case, output is T(R)*T(S)
  - *Usually not that bad…*



Simple SMJ Optimization

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



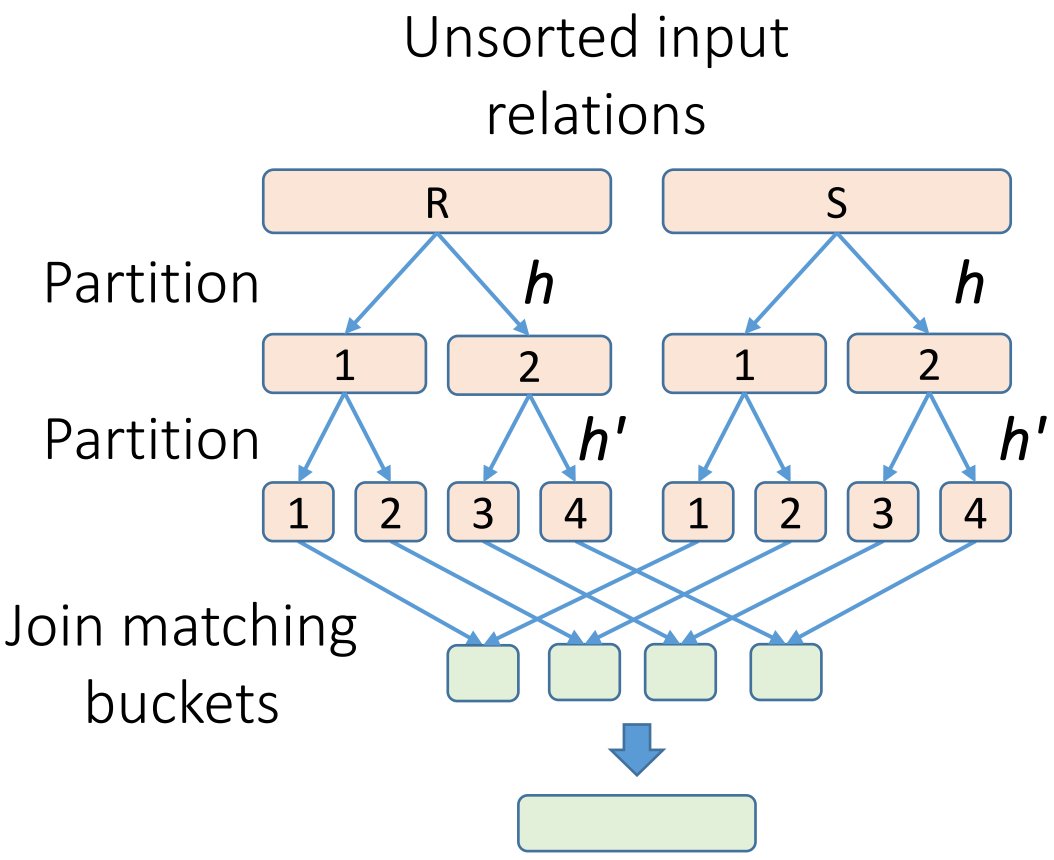
This allows us to "skip" the last sort & save 2(P(R) + P(S))!

Hash Join

- ***Goal:*** Execute R ⋈ S on A

- ***Key Idea:*** We can partition R and S into buckets by hashing the join attribute- then just join the pairs of (small) matching buckets!

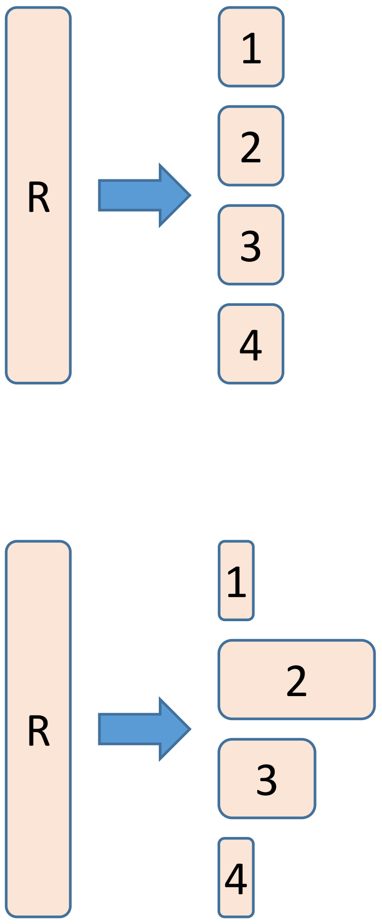
- ***IO Cost:***
  - *Partition phase:* 2(P(R) + P(S)) each pass
  - *Join phase: Depends on size of the buckets… can be* ~ P(R) + P(S) + OUT if they are small enough!
    - ***Can be worse though- see next slide!***



HJ: Skew

- Ideally, our hash functions will partition the tuples *uniformly*

- However, hash collisions and *duplicate join key attributes* can cause ***skew***
  - For hash collisions, we can just partition again with a new hash function
  - Duplicates are just a problem… (Similar to in SMJ!)



R ⋈ S on A

Overview: SMJ vs. HJ

SMJ

Note: Ext. Merge Sort!

- We create ***initial sorted runs***
- We keep ***merging*** these runs until we have one sorted merged run for R, S
- We scan over R and S to complete the ***join***

HJ

- We keep ***partitioning*** R and S into progressively smaller buckets using hash functions h, h', h''…
- We ***join*** matching pairs of buckets (using BNLJ)

How many of these passes do we need to do?

R ⋈ S on A

How many passes do we need?

SMJ

# of passes	Length of runs	# of runs
0	1	N
1	B+1	$\left\lceil \frac{N}{B+1} \right\rceil$
2	B(B+1)	$\frac{1}{B}\left\lceil \frac{N}{B+1} \right\rceil$
…	…	…
k+1	$B^k(B+1)$	$\frac{1}{B^k}\left\lceil \frac{N}{B+1} \right\rceil$

Initial sorted runs

HJ

# of passes	Avg. bucket size	# of buckets
0	N	1
1	$\left\lceil \frac{N}{B} \right\rceil$	B
2	$\frac{1}{B}\left\lceil \frac{N}{B} \right\rceil$	B^2
…	…	…
k+1	$\frac{1}{B^k}\left\lceil \frac{N}{B} \right\rceil$	B^{k+1}

Each pass, we get:

Fewer, longer runs by a factor of ***B***

More, smaller buckets by a factor of ***B***

Each pass costs $2(P(R) + P(S))$

R ⋈ S on A

How many passes do we need?

SMJ

# of passes	Length of runs	# of runs
k+1	$B^k(B+1)$	$\frac{1}{B^k}\left\lceil\frac{N}{(B+1)}\right\rceil$

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

$$B \geq \frac{\mathrm{P(R)}}{\mathrm{B^k(B+1)}} + \frac{\mathrm{P(S)}}{\mathrm{B^k(B+1)}}$$

$$B^{k+1}(B+1) \geq \mathrm{P(R)} + \mathrm{P(S)}$$

**Using the 'optimization' on slide 25*

HJ

# of passes	Avg. bucket size	# of buckets
k+1	$\frac{1}{B^k}\left\lceil\frac{N}{B}\right\rceil$	B^{k+1}

If *one of* the relations has bucket size $\leq B-1$, then we have partitioned enough to complete the join with single-pass BNLJ:

$$\mathrm{B}-1 \geq \frac{\min\{\mathrm{P(R)}, P(S)\}}{B^{k+1}}$$

$$B^{k+1}(\mathrm{B}-1) \geq \min\{\mathrm{P(R)}, P(S)\}$$

R ⋈ S on A

How many buffer pages for nice behavior?

Let's consider what B we'd need for k+1 = 1 passes (plus the final join):

SMJ

$$B(B+1) \geq \mathrm{P(R)} + \mathrm{P(S)}$$

If we use repacking, then we can satisfy the above if approximately:

$$B^2 \geq \max\{\mathrm{P(R)}, \mathrm{P(S)}\}$$

HJ

$$B(B-1) \geq \min\{\mathrm{P(R)}, \mathrm{P(S)}\}$$

So approximately:

$$B^2 \geq \min\{\mathrm{P(R)}, \mathrm{P(S)}\}$$

→ *Total IO Cost = **3(P(R) + P(S)) + OUT**!*

Overview: SMJ vs. HJ

- HJ:
  - PROS: Nice linear performance is dependent on the *smaller relation*
  - CONS: Skew!
- SMJ:
  - PROS: Great if relations are already sorted; output is sorted either way!
  - CONS:
    - Nice linear performance is dependent on the *larger* relation
    - Backup!

High-Level: Lecture 18

- Overall RDBMS architecture
- The Relational Model
- Relational Algebra

Check out the Relational Algebra practice exercises notebook!!

RDBMS Architecture

How does a SQL engine work ?

SQL Query → Relational Algebra (RA) Plan → *Optimized* RA Plan → Execution

Declarative query (from user)

Translate to relational algebra expression

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!

The Relational Model: Data

Student

sid	name	gpa
001	Bob	3.2
002	Joe	2.8
003	Mary	3.8
004	Alice	3.5

An **attribute** (or **column**) is a typed data entry present in each tuple in the relation

The number of tuples is the **cardinality** of the relation

A **tuple** or **row** (or *record)* is a single entry in the table having the attributes specified by the schema

The number of attributes is the **arity** of the relation

A **relational instance** is a ***set*** of tuples all conforming to the same *schema*

Relational Algebra (RA)

- <u>Five **basic** operators:</u>
  1. Selection: σ
  2. Projection: Π
  3. Cartesian Product: $\times$
  4. Union: $\cup$
  5. Difference: -
- <u>Derived or auxiliary operators:</u>
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: ρ
  - Division

1. Selection (σ)

- Returns all tuples which satisfy a condition
- Notation: $\sigma_c(R)$
- The condition c can be =, <, >, <>

```
Students(sid,sname,gpa)
```

SQL:

```sql
SELECT *
FROM Students
WHERE gpa > 3.5;
```

RA:

$$\sigma_{gpa > 3.5}(Students)$$

2. Projection (Π)

- Eliminates columns, then removes duplicates
- Notation: $\Pi_{A1,\ldots,An}(R)$

```
Students(sid,sname,gpa)
```

SQL:

```
SELECT DISTINCT
   sname,
   gpa
FROM Students;
```

RA:

$$\Pi_{sname,gpa}(Students)$$

3. Cross-Product (×)

- Each tuple in R1 with each tuple in R2
- Notation: $R1 \times R2$
- Rare in practice; mainly used to express joins

```
Students(sid,sname,gpa)
People(ssn,pname,address)
```

SQL:

```
SELECT *
FROM Students, People;
```

RA:

$$Students \times People$$

Renaming (ρ)

`Students(sid,sname,gpa)`

- Changes the schema, not the instance
- A 'special' operator- neither basic nor derived
- Notation: $\rho_{B1,\ldots,Bn}(R)$
- **Note: this is shorthand for the proper form (since names, not order matters!):**
  - $\rho_{A1 \rightarrow B1,\ldots,An \rightarrow Bn}(R)$

SQL:

```sql
SELECT
  sid AS studId,
  sname AS name,
  gpa AS gradePtAvg
FROM Students;
```

RA:

$$\rho_{studId,name,gradePtAvg}(Students)$$

We care about this operator *because* we are working in a *named perspective*

Natural Join ($\bowtie$)

- Notation: $R_1 \bowtie R_2$

- Joins R_1 and R_2 on *equality of all shared attributes*
  - If R_1 has attribute set A, and R_2 has attribute set B, and they share attributes A⋂B = C, can also be written: $R_1 \bowtie_C R_2$

- Our first example of a *derived* RA operator:
  - Meaning: $R_1 \bowtie R_2 = \Pi_{A \cup B}(\sigma_{C=D}(\rho_{C \to D}(R_1) \times R_2))$
  - Where:
    - The rename $\rho_{C \to D}$ renames the shared attributes in one of the relations
    - The selection $\sigma_{C=D}$ checks equality of the shared attributes
    - The projection $\Pi_{A \cup B}$ eliminates the duplicate common attributes

```
Students(sid,name,gpa)
People(ssn,name,address)
```

SQL:

```
SELECT DISTINCT
  ssid, S.name, gpa,
  ssn, address
FROM
  Students S,
  People P
WHERE S.name = P.name;
```

RA:

$$Students \bowtie People$$

Converting SFW Query -> RA

```
SELECT DISTINCT A1,…,An
FROM            R1,…,Rm
WHERE           c1 AND … AND ck;
```

$$\Pi_{A_1,\dots,An}(\sigma_{c_1} \dots \sigma_{c_k}(R_1 \bowtie \cdots \bowtie R_m))$$

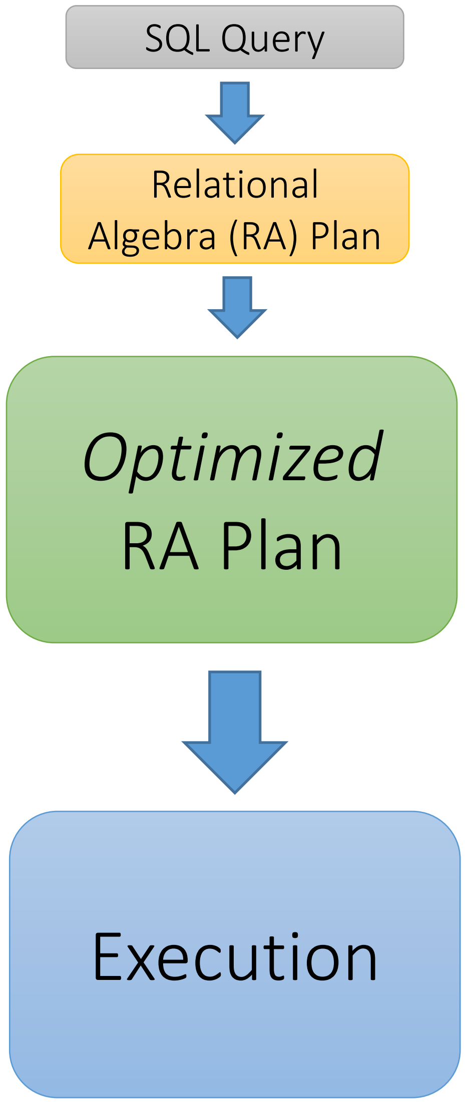
Why must the selections "happen before" the projections?

High-Level: Lecture 19

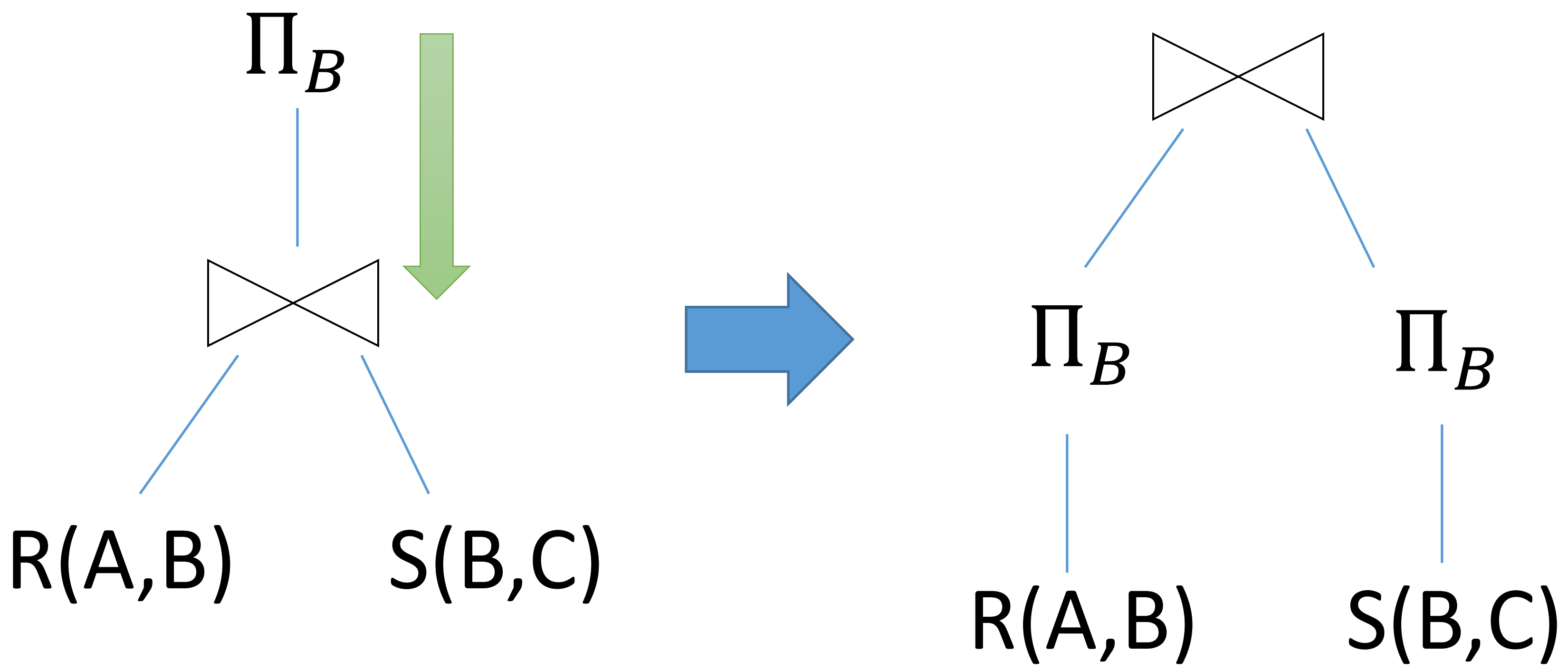
- Logical optimization
- Physical optimization
    - Index selections
    - IO cost estimation

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)*

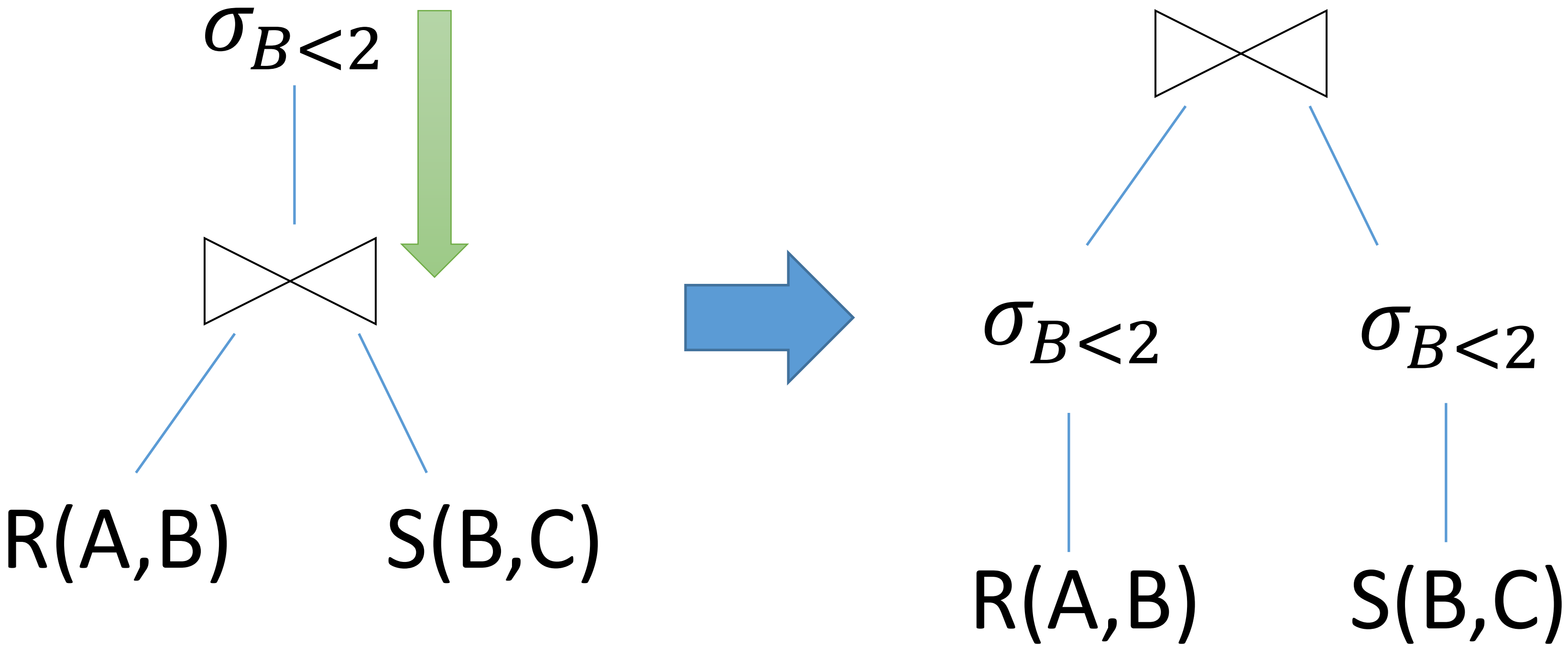


Logical Optimization: "Pushing down" projection



Why might we prefer this plan?

Logical Optimization: "Pushing down" selection



Why might we prefer this plan?

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!

Index Selection

Input:

- Schema of the database
- **Workload description:** set of (query template, frequency) pairs

Goal: Select a set of indexes that minimize execution time of the workload.

- Cost / benefit balance: Each additional index may help with some queries, but requires updating

This is an optimization problem!

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 $\boldsymbol{\sigma_{A>10}(\mathrm{R}) \bowtie \sigma_{B=1}(S)}$?**

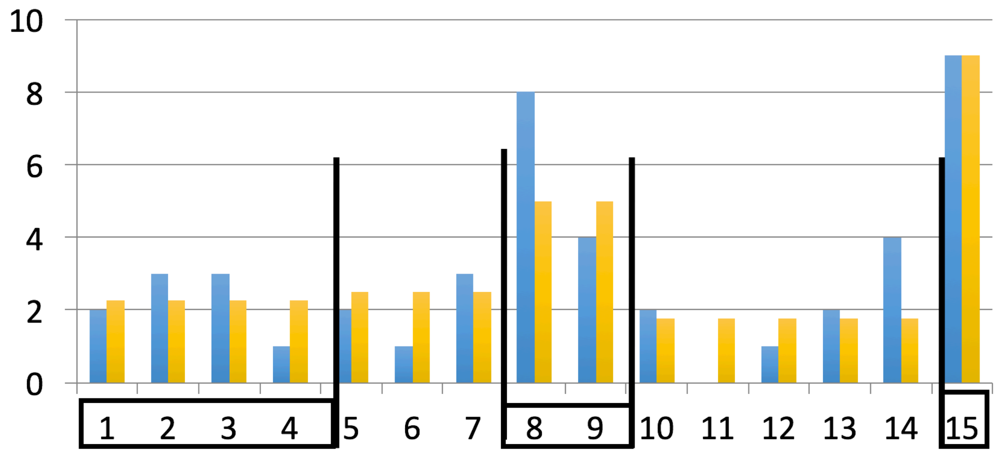
- In general, we will need some way to ***estimate intermediate result set sizes***

Histograms provide a way to efficiently store estimates of these quantities

Histogram types

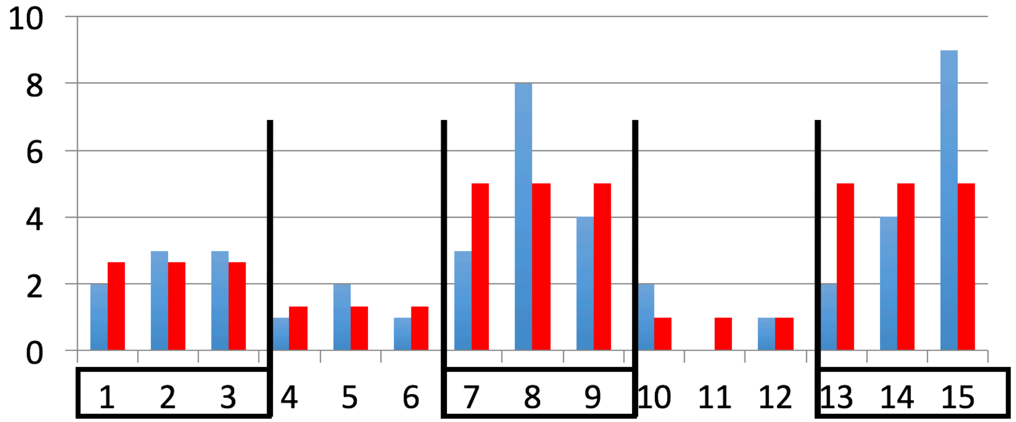
Equi-depth

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



Equi-width

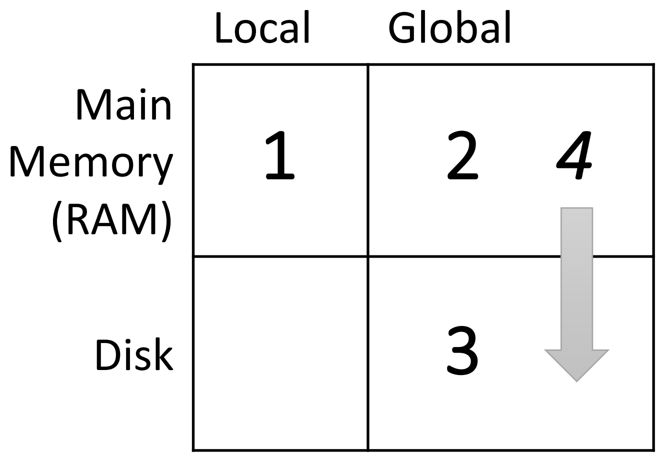
All buckets roughly the same width



High-Level: Lecture 20

- Our model of the computer: Disk vs. RAM, local vs. global
- Transactions (TXNs)
- ACID
- Logging for Atomicity & Durability
  - Write-ahead logging (WAL)

Our model: Three Types of Regions of Memory



1. **Local:** In our model each process in a DBMS has its own local memory, where it stores values that only it "sees"

2. **Global:** Each process can read from / write to shared data in main memory

3. **Disk:** Global memory can read from / flush to disk

4. ***Log:*** *Assume on stable disk storage- spans both main memory and disk…*

Log is a *sequence* from main memory -> disk

"<u>**Flushing**</u> to disk" = writing to disk + erasing ("evicting") from main memory

Transactions: Basic Definition

A **<u>transaction ("TXN")</u>** 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
```

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

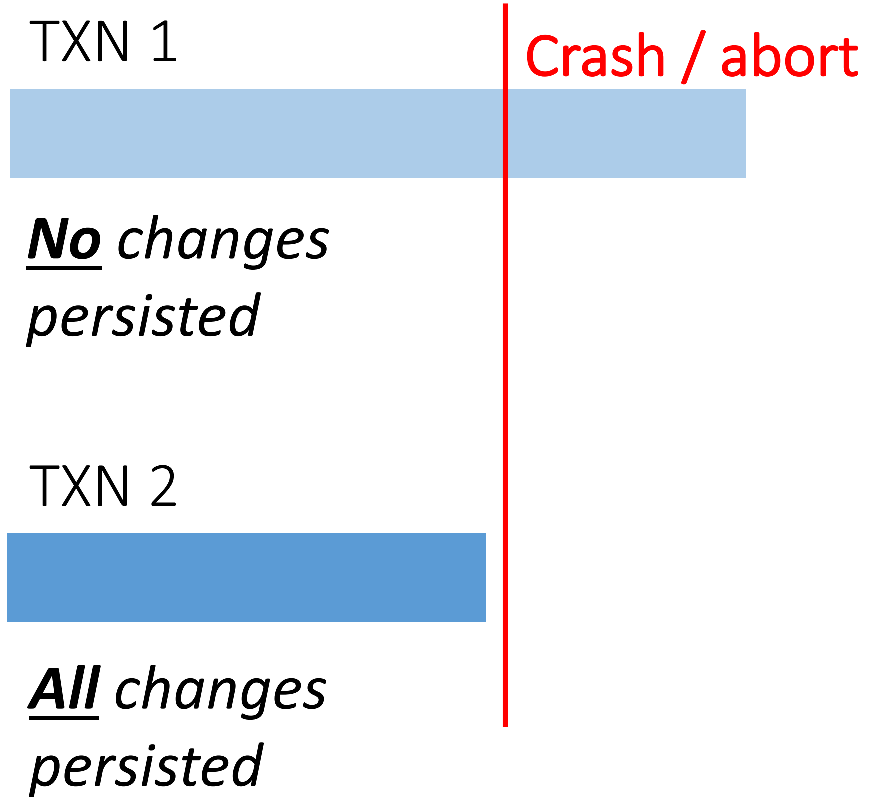
ACID is/was source of great debate!

Goal of LOGGING: Ensuring Atomicity & Durability

ACID

- **Atomicity:**
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen

- **Durability:**
  - If DBMS stops running, changes due to completed TXNs should all persist
  - *Just store on stable disk*



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:

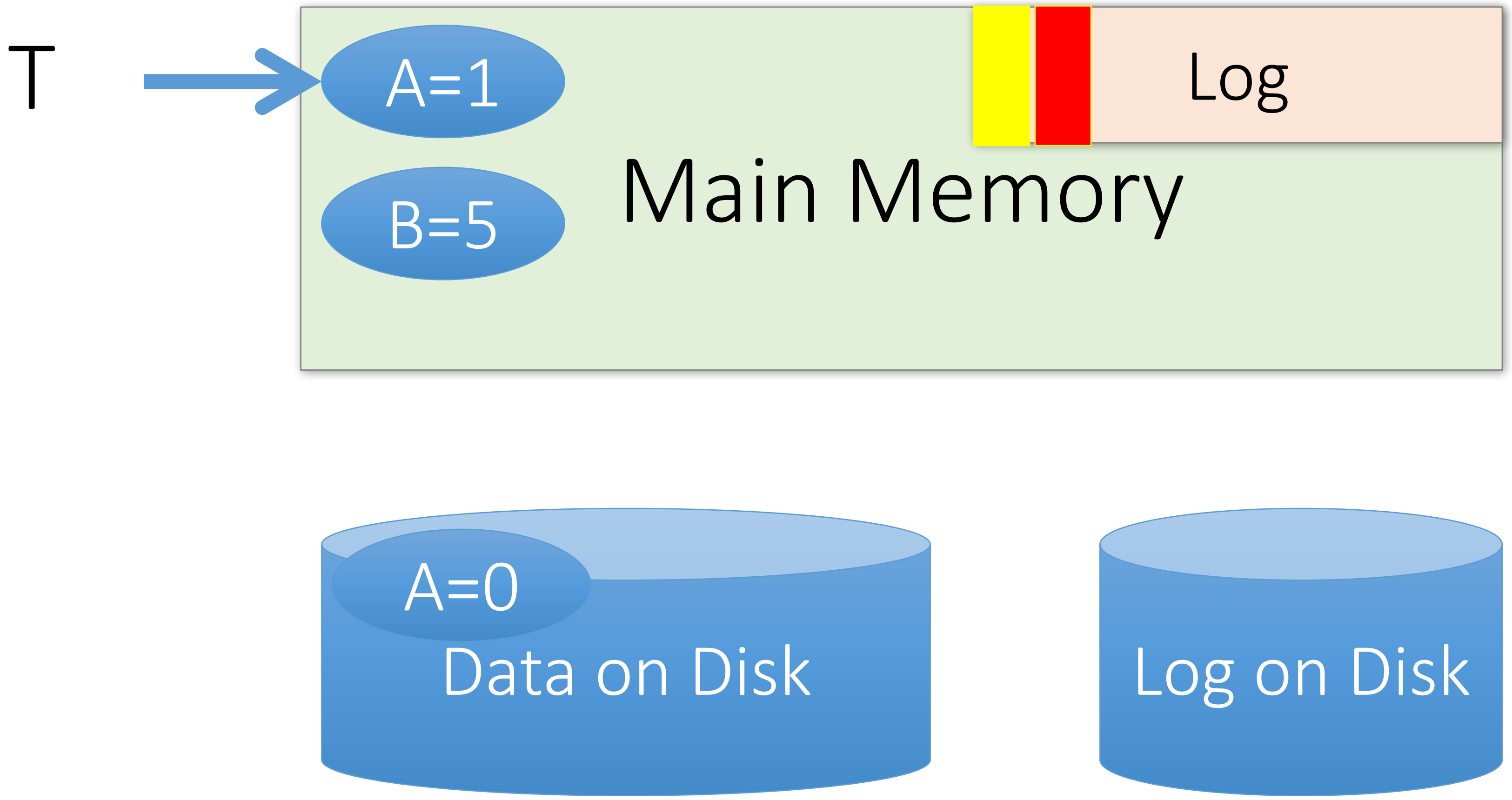
    <XID, location, old data, new data>

This is sufficient to UNDO any transaction!

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

A: 0→1



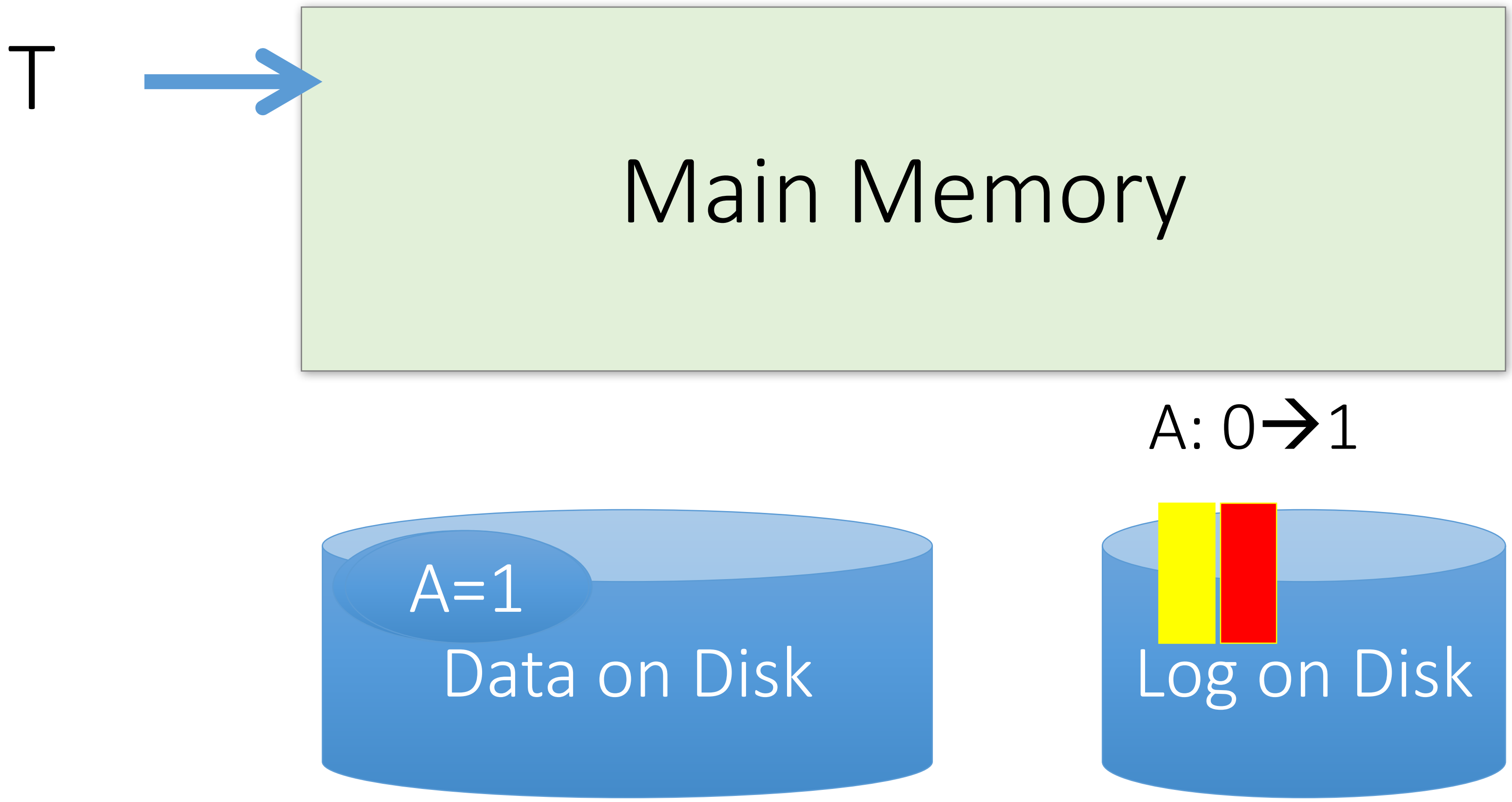
This time, let's try committing ***after* we've written log to disk but *before* we've written data to disk**... this is WAL!

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

High-Level: Lecture 21

- Motivation: Concurrency with Isolation & consistency
  - Using TXNs…

- Scheduling

- Serializability

- Conflict types & classic anomalies

Concurrency: Isolation & Consistency

- The DBMS must handle concurrency such that…

1. **Isolation** is maintained: Users must be able to execute each TXN **as if they were the only user**
   - DBMS handles the details of *interleaving* various TXNs

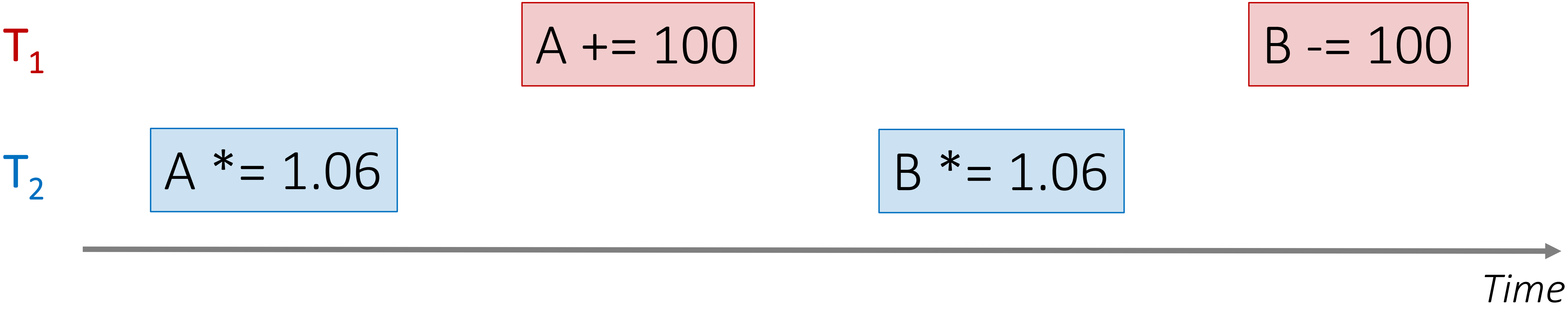
ACID

2. **Consistency** is maintained: TXNs must leave the DB in a **consistent state**
   - DBMS handles the details of enforcing integrity constraints

ACID

Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



What goes / could go wrong here??

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule $T_1 \rightarrow T_2$:

T_1: A += 100, B -= 100

T_2: A *= 1.06, B *= 1.06

A	B
$159	$106

Interleaved schedule B:

T_1: A += 100, B -= 100

T_2: A *= 1.06, B *= 1.06

A	B
$159	$112

Different result than serial $T_1 \rightarrow T_2$!

Scheduling Definitions

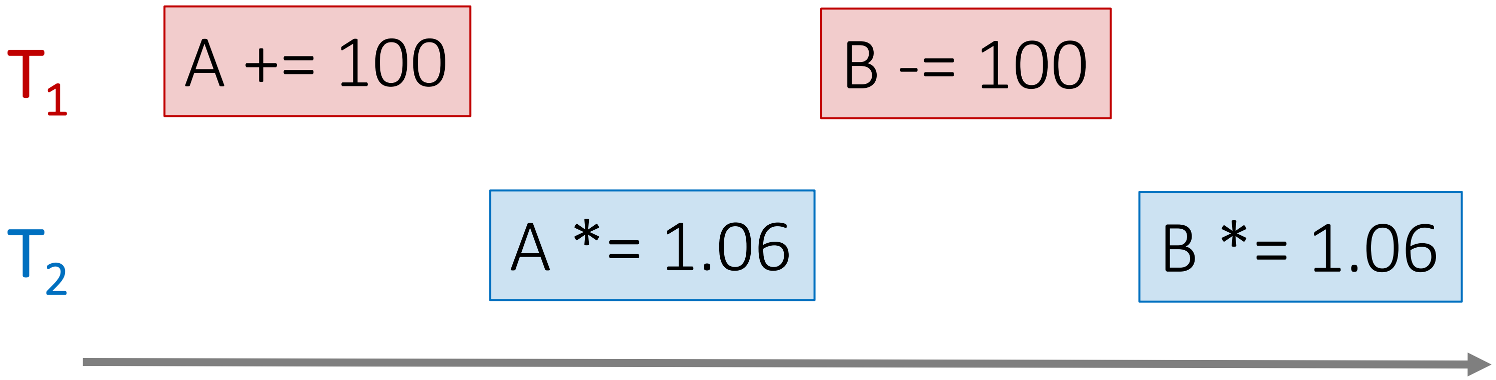
- A **<u>serial schedule</u>** is one that does not interleave the actions of different transactions

- 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 def powerful and tricky!

Serializable?



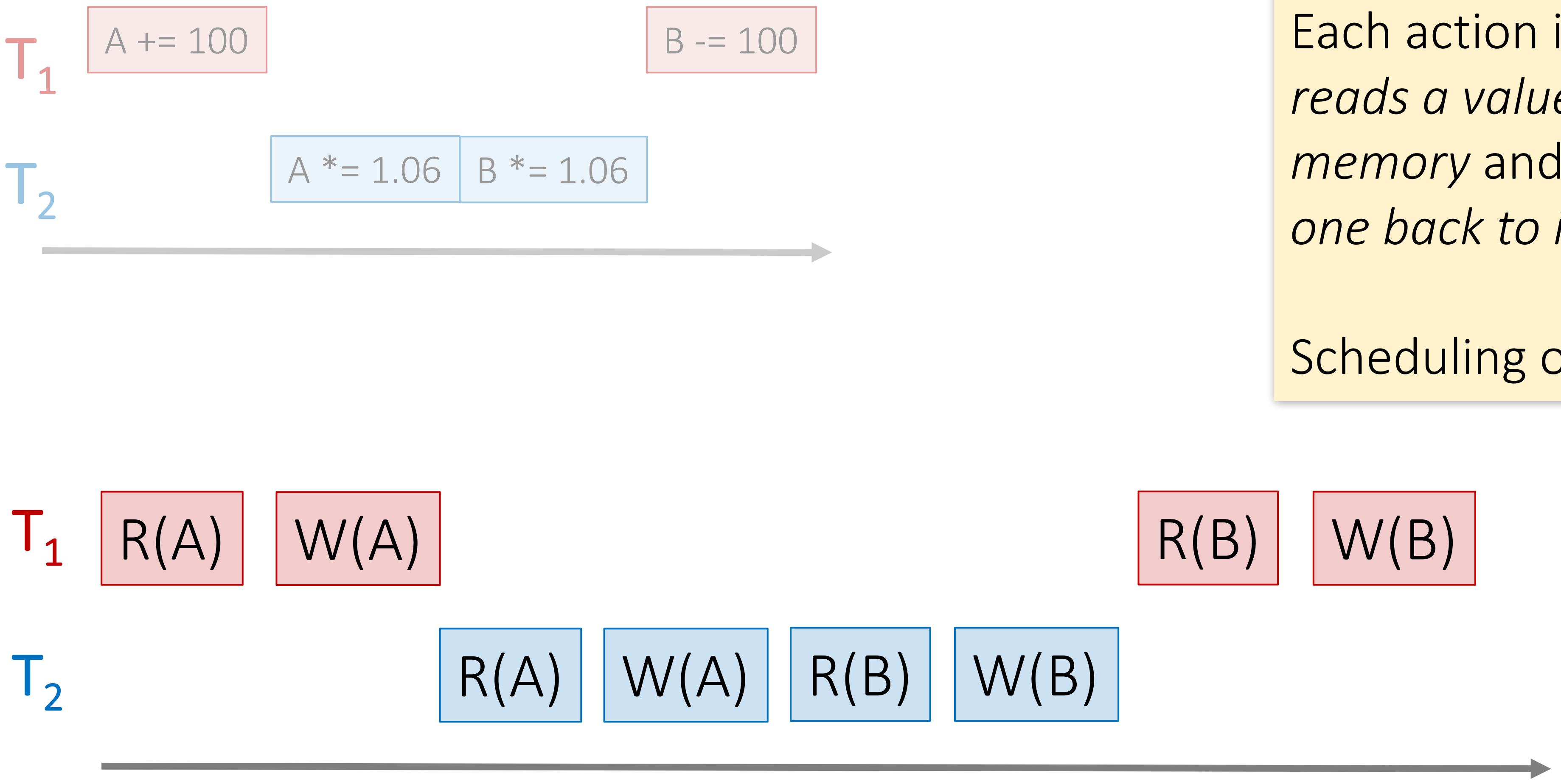
Serial schedules:

	A	B
$T_1 \rightarrow T_2$	1.06*(A+100)	1.06*(B-100)
$T_2 \rightarrow T_1$	1.06*A + 100	1.06*B - 100

A	B
1.06*(A+100)	1.06*(B-100)

Same as a serial schedule *for all possible values of A, B* = <u>serializable</u>

The DBMS's view of the schedule



Each action in the TXNs *reads a value from global memory* and then *writes one back to it*

Scheduling order matters!

Conflict Types

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write

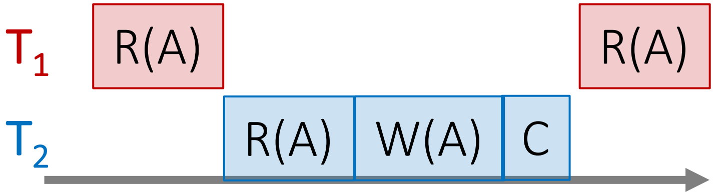
- Thus, there are three types of conflicts:
  - Read-Write conflicts (RW)
  - Write-Read conflicts (WR)
  - Write-Write conflicts (WW)

Why no "RR Conflict"?

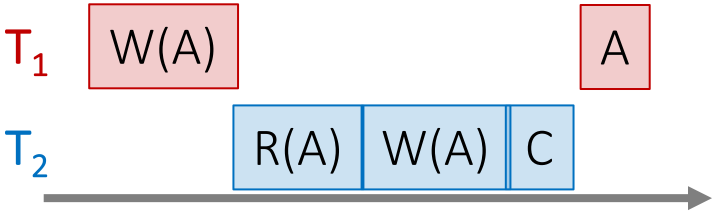
Interleaving anomalies occur with / because of these conflicts between TXNs *(but these conflicts can occur without causing anomalies!)*

Classic Anomalies with Interleaved Execution

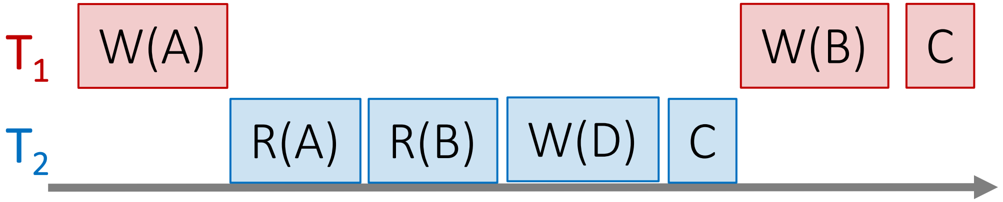
"Unrepeatable read":



"Dirty read" / Reading uncommitted data:



"Inconsistent read" / Reading partial commits:



Partially-lost update: