CS 145 Final Review

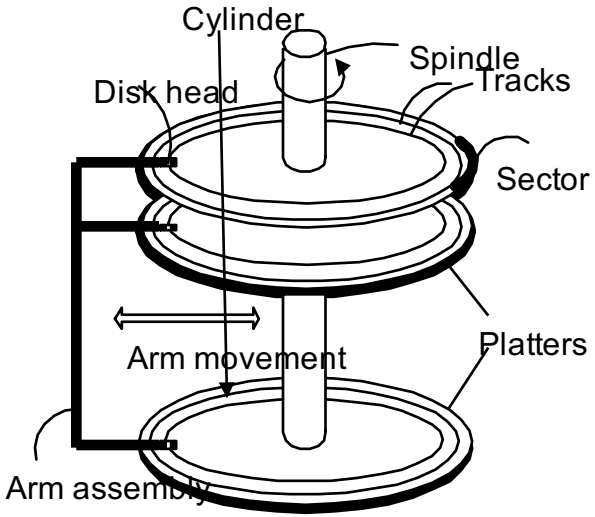
The Best Of Collection (Master Tracks), Vol. 2

= requested on piazza / in linked voting spreadsheet (@1253)

High-Level: Lecture 12

- *Note: Content from 12-1 (conflict serializability, deadlock, etc) will* ***NOT*** *be covered on the final*
- The ***buffer*** & simplified filesystem model
- Shift to ***IO Aware*** algorithms
- The ***external merge algorithm***

High-level: Disk vs. Main Memory



Disk:

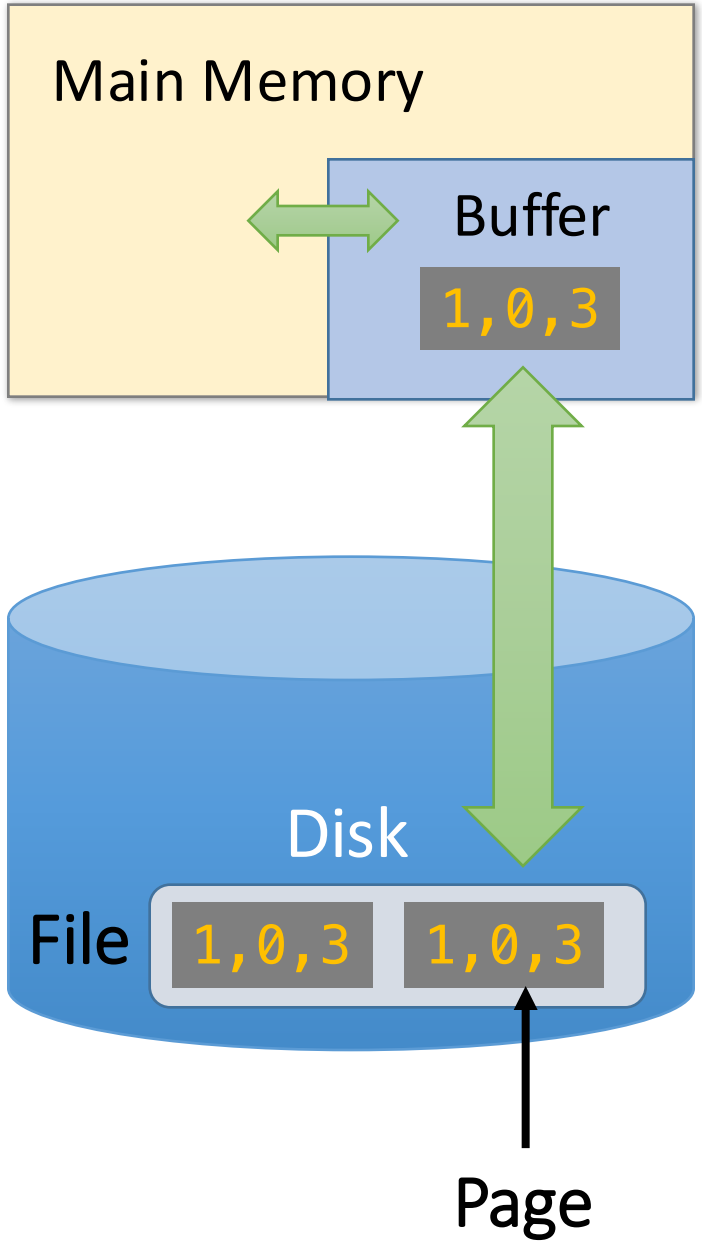
- ***Slow:*** Sequential *block* access
  - Read a blocks (not byte) at a time, so sequential access is cheaper than random
  - **Disk read / writes are expensive!**
- ***Durable:*** We will assume that once on disk, data is safe!
- ***Cheap***

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!

See L12:54-66!

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

High-Level: Lecture 13

- External Merge Sort Algorithm
  - Basic algorithm (including (B+1)-length initial runs & B-way merging)
  - Repacking optimization for longer initial runs
- Indexes Part I: Basics

See L13:11-27!

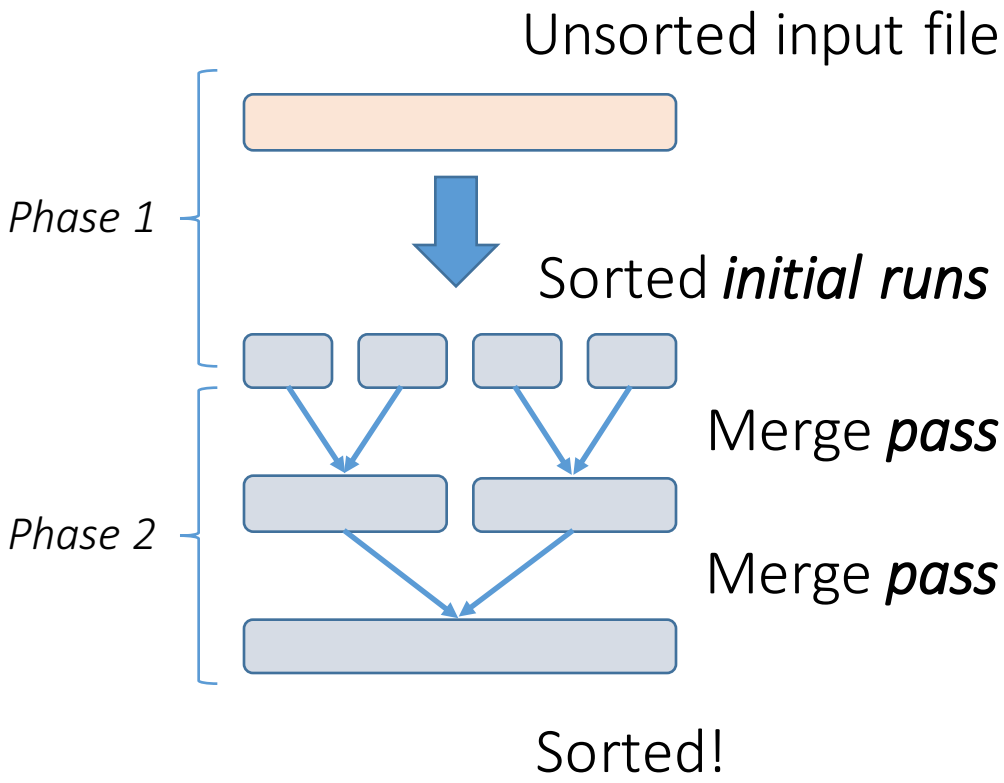
External Merge Sort Algorithm

- ***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!



See L13:11-27!

External Merge Sort Algorithm

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

See L13:28-40!

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)**

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

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

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

By_Author_Title_Index

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

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	…

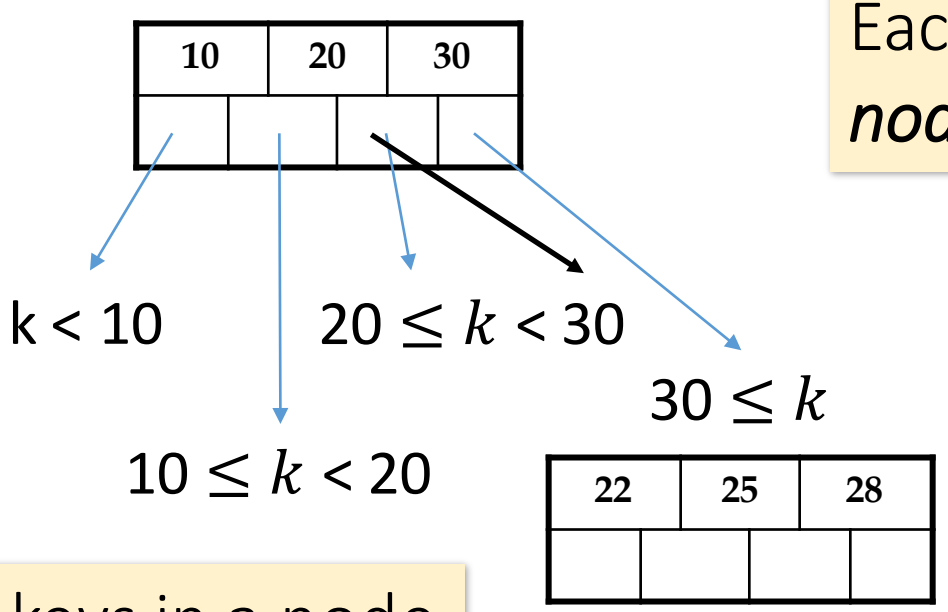
An index is **covering** *for a specific query* if the index contains all the needed attributes

High-Level: Lectures 14-15

- Indexes Pt. 2:
  - B+ Trees
  - Clustered vs. unclustered
- Join Algorithms:
  - Nested Loop Join Variants: NLJ, BNLJ, INLJ
  - SMJ
  - Hash Join

B+ Tree Basics

Non-leaf or *internal* node



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

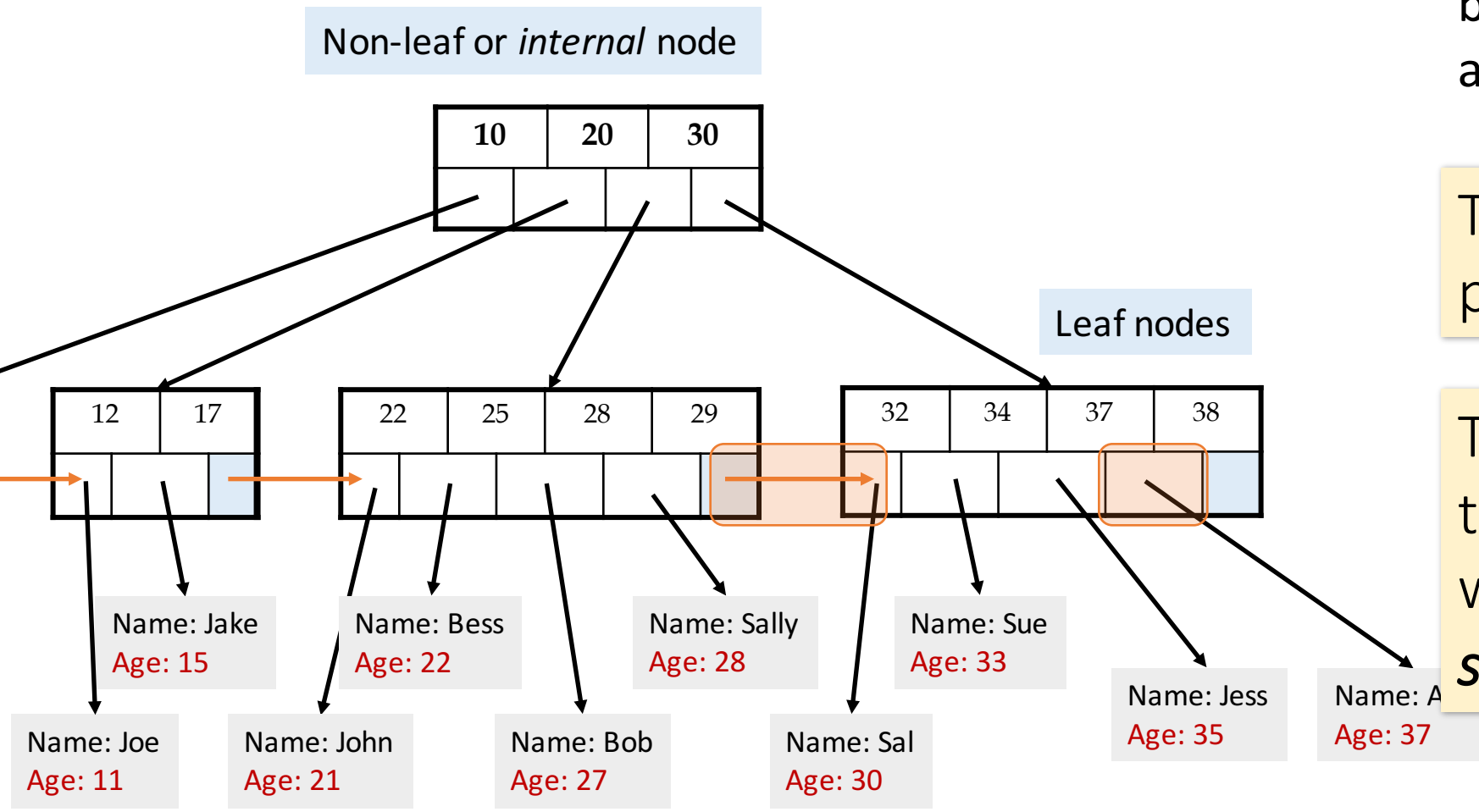
Parameter ***d*** = the degree

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

except for root node, which can have between **2 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



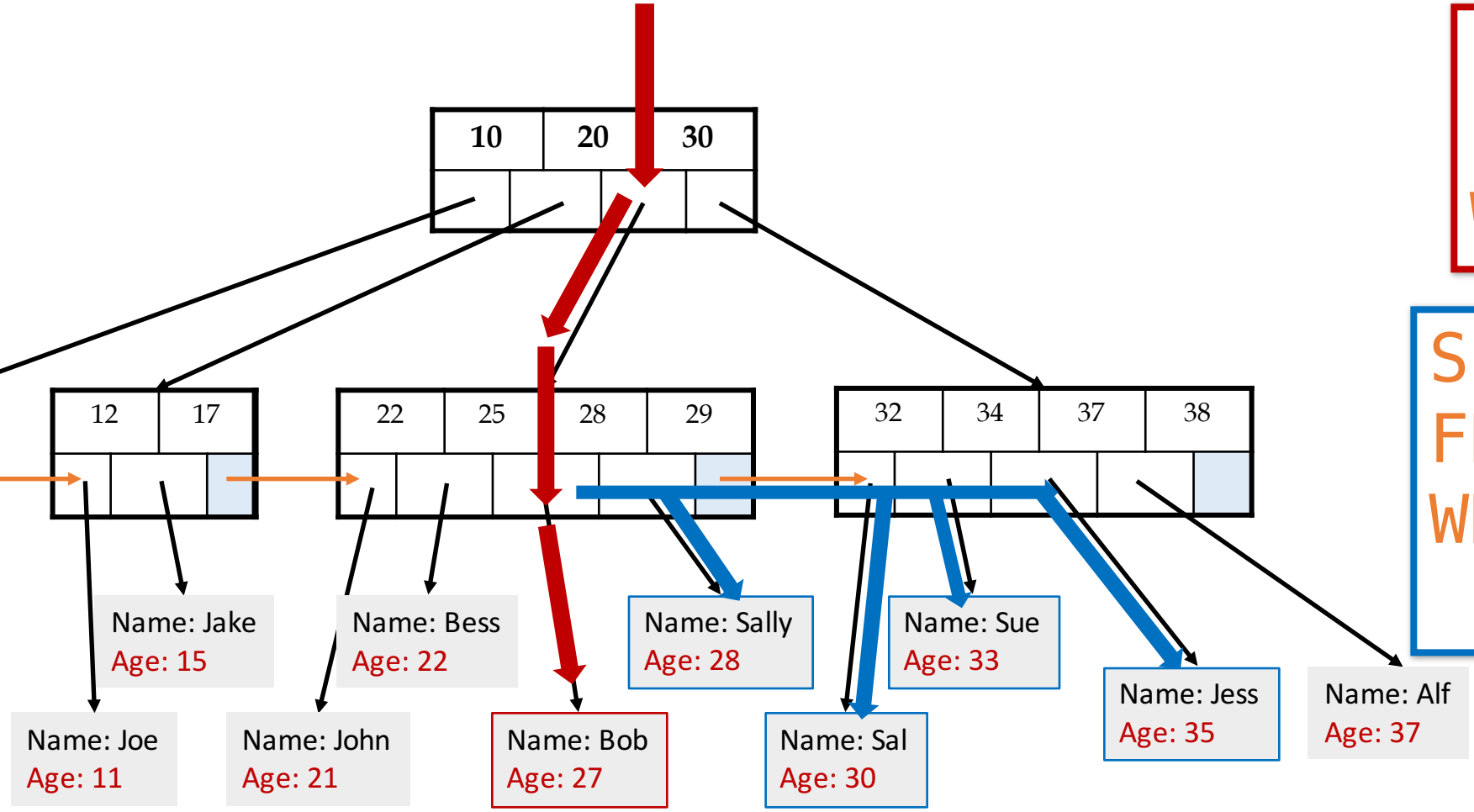
Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

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

See L14-15:17-18!

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

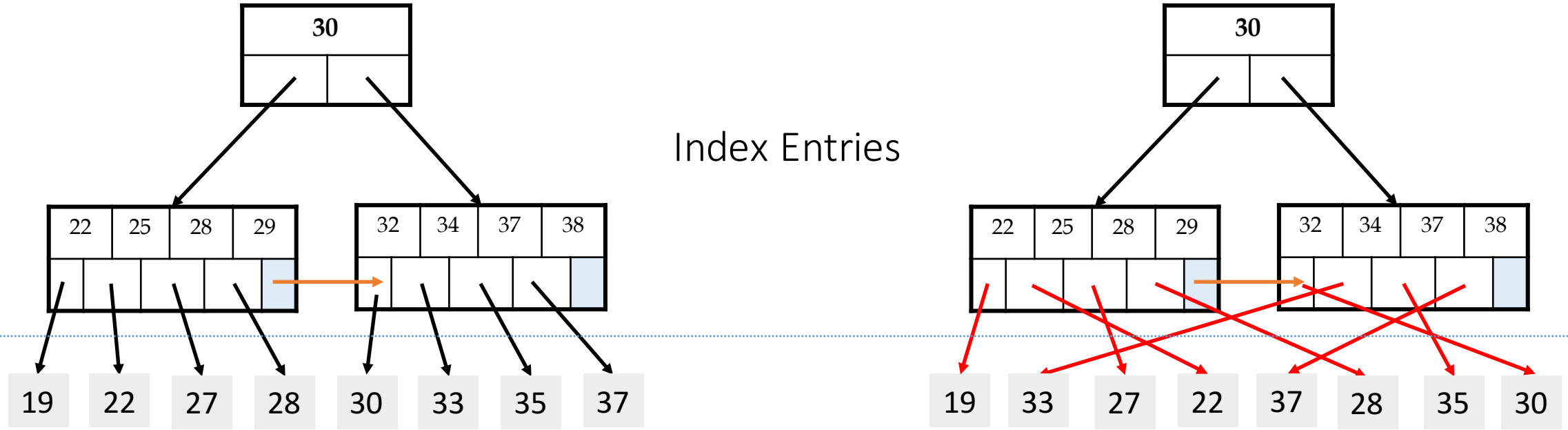
- ***Details:***
  - One node per page (thus page size determines *d*)
  - Fill only some of each node's slots (the ***fill-factor***) to leave room for insertions
  - We can keep some levels of the B+ Tree in memory!

See L14-15:22-24!

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:	$\left\lceil \log_f \frac{N}{F} \right\rceil - \left\lfloor \log_f B \right\rfloor + \mathbf{Cost}(Out)$	***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!

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!

Joins: Example

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

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

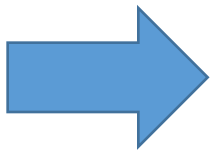
Example: Returns all pairs of tuples $r \in R, s \in S$ *such that* $r.A = s.A$

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

Cost:

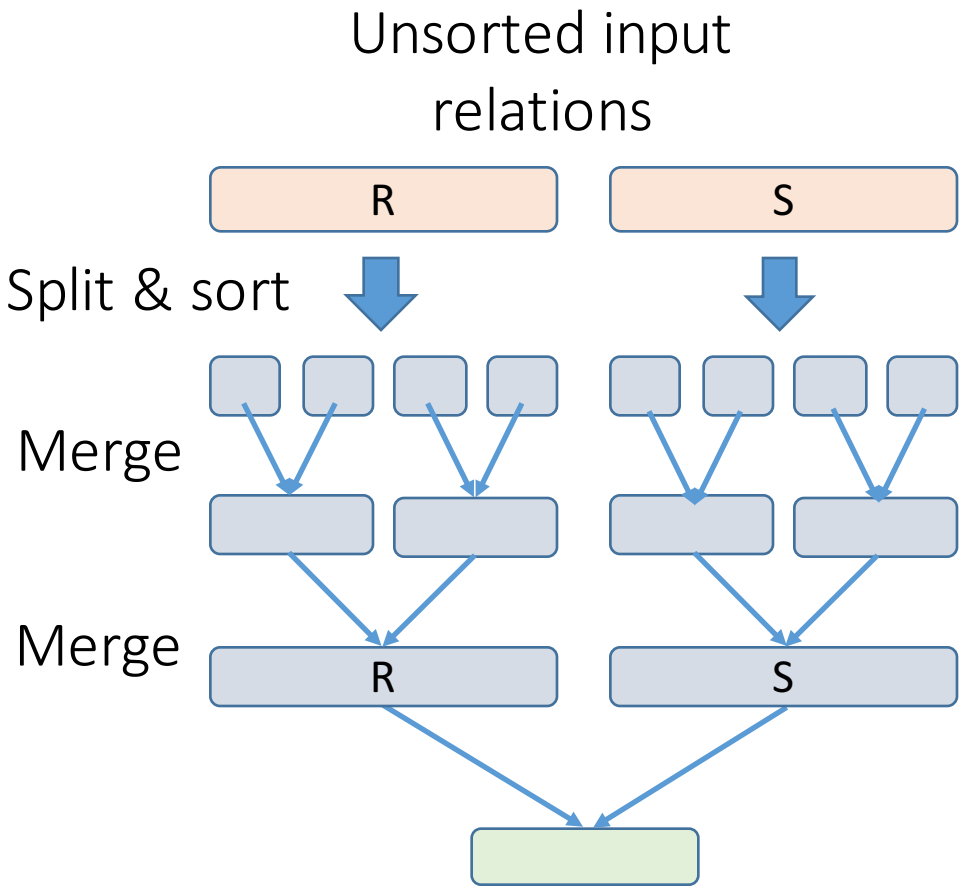
$$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$$

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**

See L14-15:63-69!

Sort Merge Join (SMJ)

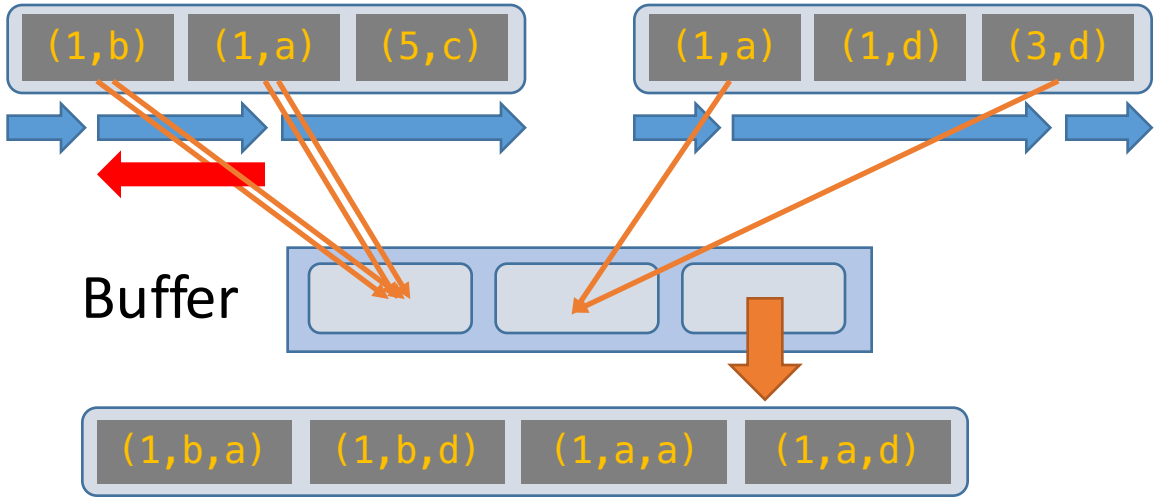
- ***Goal:*** Execute R ⋈ S on A
- ***Key Idea:*** We can sort R and S, then just scan over them!
- ***IO Cost:***
  - *Sort phase:* Sort(R) + Sort(S)
  - *Merge / join phase:* ~ P(R) + P(S) + OUT
    - ***Can be worse though- see next slide!***



See L14-15:70-75!

SMJ: Backup

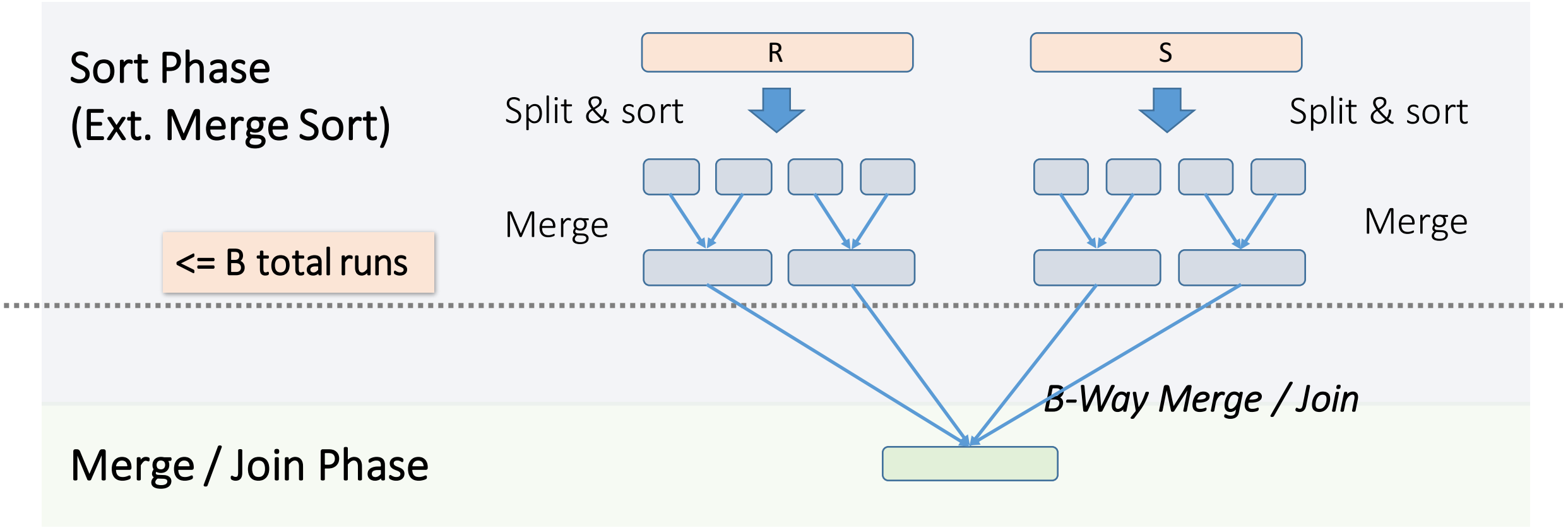
- Without any duplicates:
  - We just scan over R and S once each → P(R) + P(S)
- However, if there are duplicates, we may have to ***back up*** and re-read parts of the file
  - Up to P(R)*P(S)!
  - *Usually not that bad…*



See L14-15:78-81!

Simple SMJ Optimization

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

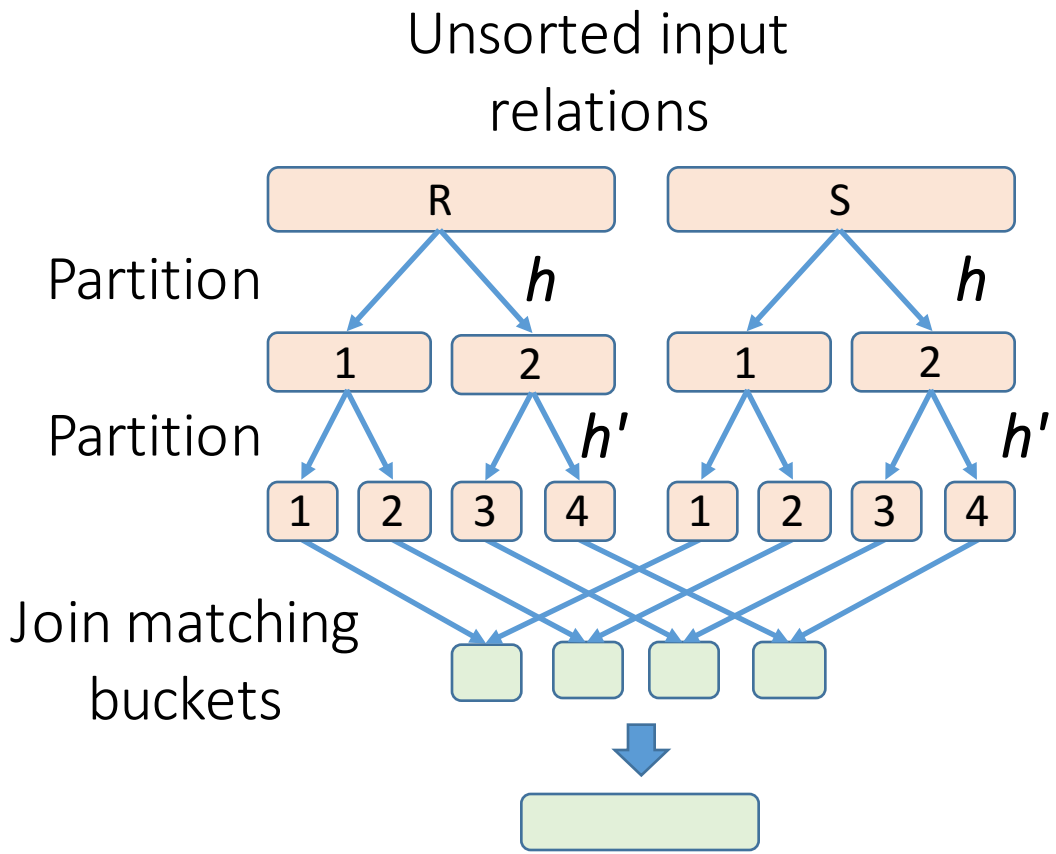


This allows us to "skip" the last sort & save 2(P(R) + P(S))!

See L14-15:88-!

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!***

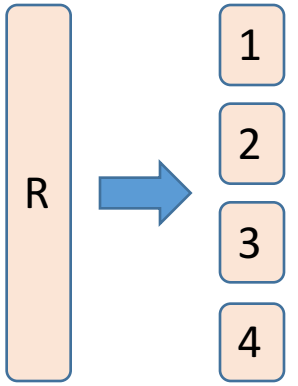


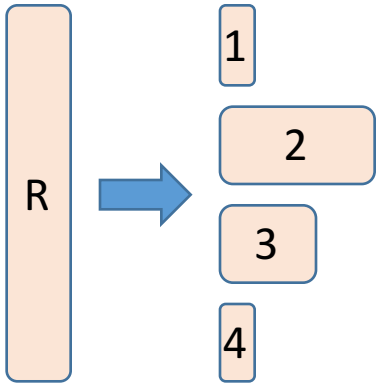
See L14-15:109-112!

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

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	$\frac{1}{B^k}\left\lceil \frac{N}{B} \right\rceil$	B^k

Note:
- *Exact* initial number
- *Approximate* scaling factor

Each pass, we get:

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

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

R ⋈ S on A

How many passes do we need?

SMJ

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

R and S each fully merged once:

$B^k(B+1) >= max\{P(R), P(S)\}$

HJ

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

R and S partitioned small enough to do single-pass BNLJ if *one of them* has bucket size <= B-1, thus we need:

$B^k(B-1) >= min\{P(R), P(S)\}$

Note:

- *Exact* initial number
- *Approximate* scaling factor

R ⋈ S on A

How many buffer pages for nice behavior?

SMJ

- **If <u>max</u>{P(R), P(S)} < B^2…**
- Then we can create <= B initial sorted runs
- Which means we can then immediately B-way merge / join these*

HJ

- **If <u>min</u>{P(R), P(S)} < B^2…**
- Then we can get *pairs of* buckets where the smaller one is < B-1
- And then, we can do BNLJ on these pairs in linear time!

I.e. $k = 1$ from previous slide!

**With optimization described in lecture*

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

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!

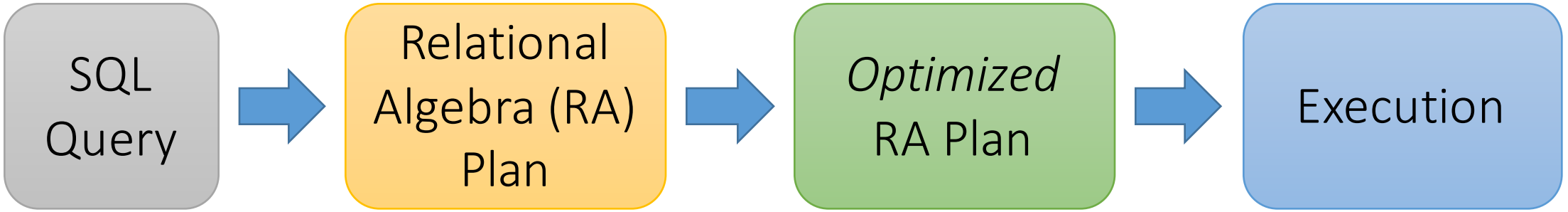
High-Level: Lecture 16

- Overall RDBMS architecture
- The Relational Model
- Relational Algebra

Check out the Relational Algebra practice exercises notebook!!

RDBMS Architecture

How does a SQL engine work ?



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

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

Student

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

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

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

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

Relational Algebra (RA)

- Five **basic** operators:
  1. Selection: σ
  2. Projection: Π
  3. Cartesian Product: $\times$
  4. Union: $\cup$
  5. Difference: -
- Derived or auxiliary operators:
  - 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:

```
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 × R2
- Rare in practice; mainly used to express joins

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

SQL:

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

RA:

$$Students \times People$$

Natural Join (⋈)

- Notation: $R_1 \bowtie R_2$
- Our first example of a *derived* RA operator:
  - Meaning: $R_1 \bowtie R_2 = \Pi_A(\sigma_C(R_1 \times R_2))$
- Where:
  - The selection σ_C checks equality of all common attributes
  - The projection 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:

$$Students \bowtie People$$

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:

```
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*

Converting SFW Query -> RA

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

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

Why must the selections “happen before” the projections?

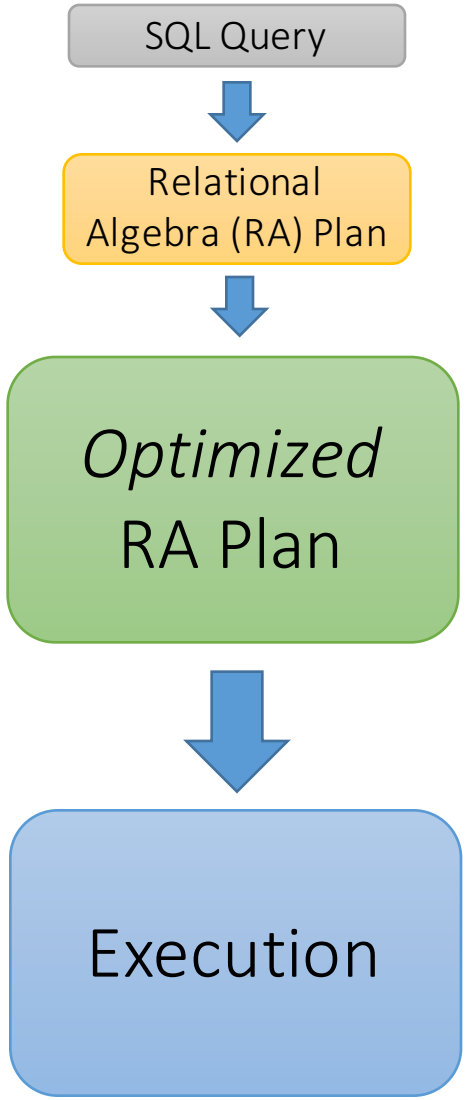
High-Level: Lecture 17

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

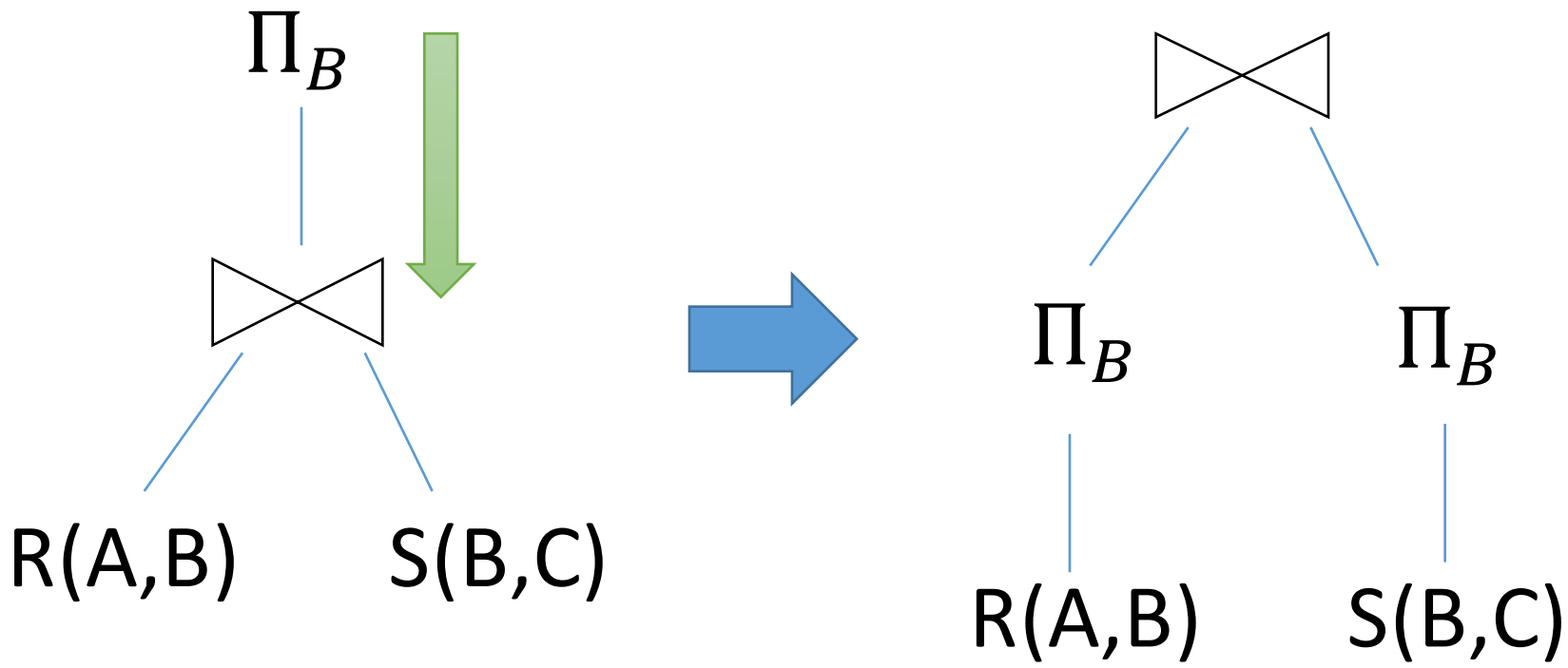
Logical vs. Physical Optimization

- **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" operators



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

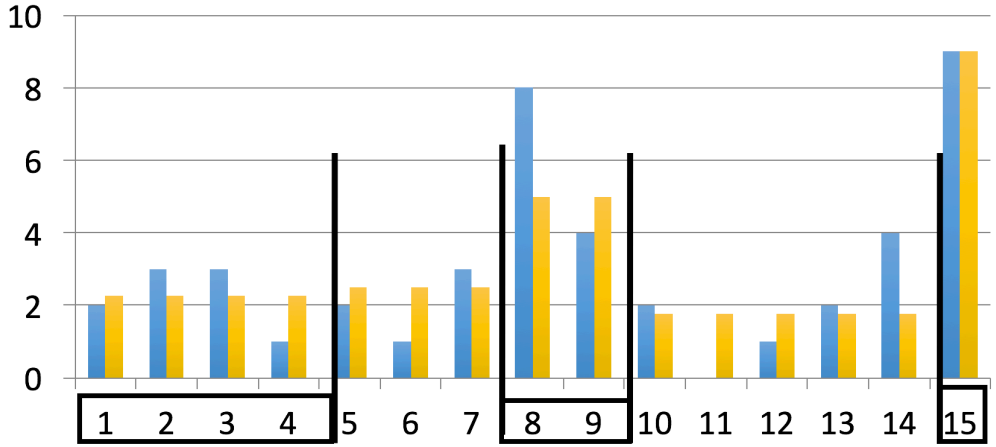
- For **index selection**:
  - What is the cost of an index lookup?
- Also for **deciding which algorithm to use**:
  - Ex: To execute $\mathrm{R} \bowtie S$, which join algorithm should DBMS use?
  - **What if we want to compute $\boldsymbol{\sigma_{A>10}(\mathrm{R}) \bowtie \sigma_{B=1}(S)}$?**
- In 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