# Cs 145

Exam 1 review

## Exam Reminders

- 24 hours
  - 12:01am PDT 10/28 - 11:59pm PDT 10/26
- Private ED posts only (TAs will share updates on doc)
- Open book and open internet
  - Honor code still applies
- Murphy’s law → leave time for scan/upload

## Exam Reminders

Exam is **~25 pages** (including space for answers)

8 questions, 90 points/90 minutes

- 3 on SQL and SQL optimization
- 1 on systems primer
- 4 on sorting, hashing, b+trees, joins

[Goal of test:

- Test understanding of principles. Not blindly applying formulae. Read questions carefully for assumptions.
- If not obvious, ask. State any “reasonable” assumptions. (e.g., “unreasonable” = infinite speed]

## Review

Recap of

- Systems design
- SQL
- Scale, optimization

Catch up with full lectures for details

## Review

At this point…

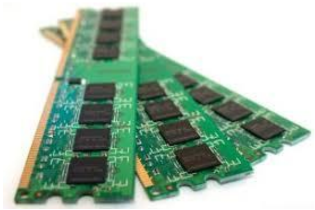
1. You have tools, techniques, equations for scaling for large datasets

2. Key next steps: Pickup data problems, practice, change assumptions. Repeat.

   (Goal of HWs, projects, midterm etc.)

# RAM, Disks, Clouds


(Example [datacenter] from 2:50 min)



*Fast:* Random access vs disks, byte addressable

- ~10x faster for sequential access
- ~100,000x faster for random access!

*Volatile:* Lose Data, if e.g. crash occurs, power goes out

*Expensive:* For $100, 16GB of RAM vs. 2TB of disk!

disk rotation [video]

*Slow:* Sequential *block* access

- Disk read / writes are slow/expensive!

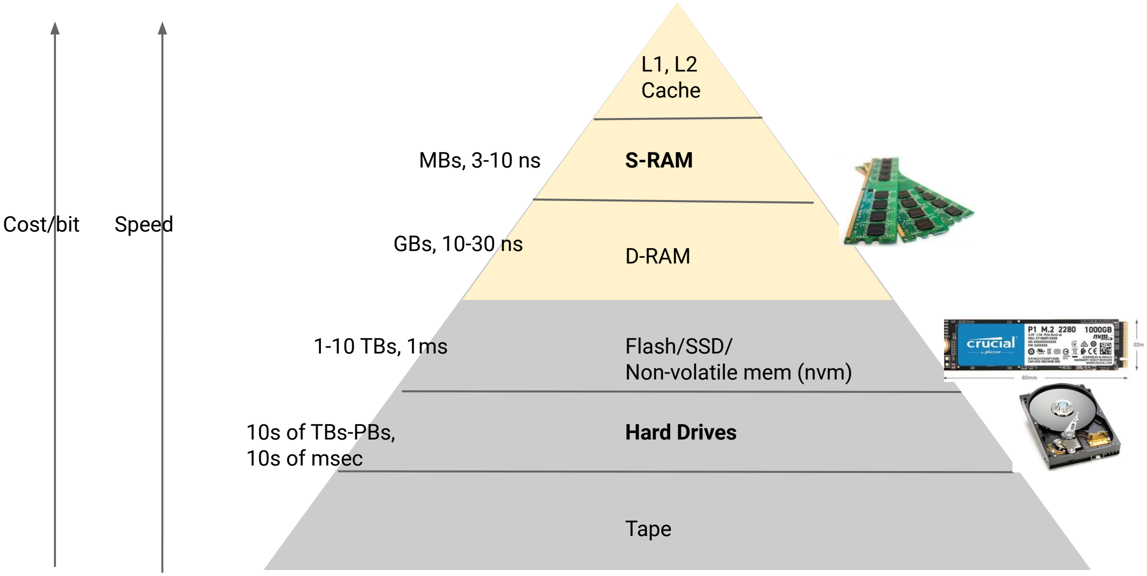
*Durable:* Data is safe* (assume for this class)!

*Cheap*

# IO Hierarchy

Volatile -- data lost when you turn off power

Non-Volatile




**Rough rule of thumb**

<1-10 GBs
Usual CS algorithms
Pandas + SQL

> 10 GBs - 10 TBs
SQL on a cluster
Store on SSDs

> 10 TBs - PBs
SQL + Cs145 algorithms
Store on HDs

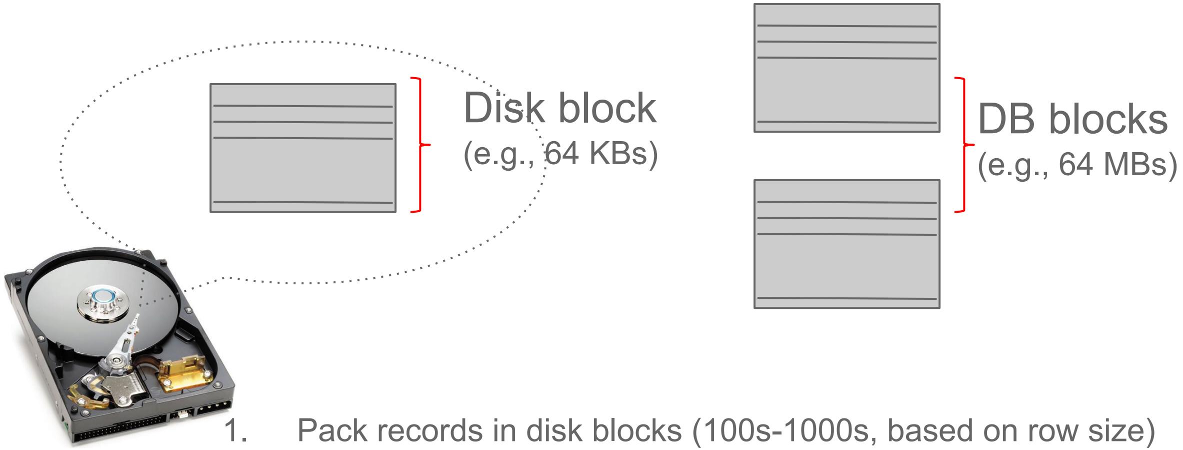
⇒ Rest of cs145: Focus on simplified RAM + Disk model
(learn tools for other IO models)

Disk blocks

& DB blocks

# After all the hard work to seek, get a big Block?

(not just a byte)




1. Pack records in disk blocks (100s-1000s, based on row size)
2. When you seek and read, you get a full disk block (i.e., you get 64 KBs, not just a byte)
3. Even better? Create a DB block with 1000 contiguous disk blocks, I.e., get 64 MBs per seek

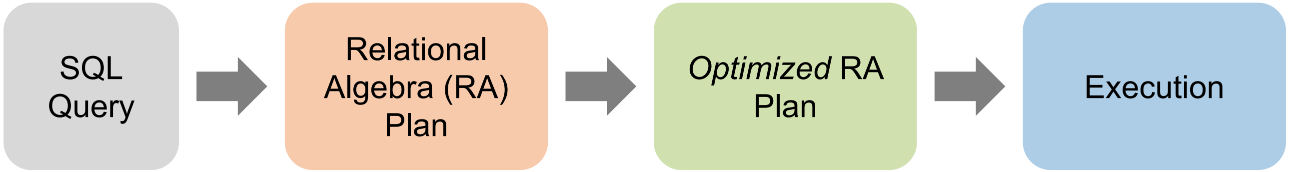
Example: To store a 1 TB table on a disk (with 64MB DB blocks)

⇒ We'd need 15,625 DB blocks

⇒ Each seek will get you back a full 64MB block

# RDBMS Architecture

How does a DB 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!

Key concept

**Data model & SQL**

# Relational model (aka tables)

Simple, popular algebra (E.F. Codd et al)

Every relation has a schema

- Logical Schema: describes types, names
- Physical Schema: describes data layout
- Virtual Schema (Views): derived tables

Data model

Organizing principle of data + operations

Schema

Describes blueprint of table (s)

# SQL to express queries declaratively

World's most successful parallel programming language

SQL

Data definition and data manipulation language

# The Relational Model: Data

**Student**

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

A <u>**column**</u> (or <u>**attribute**</u> is a typed data entry present in each tuple in the relation

The number of rows is the <u>**cardinality**</u> of the relation

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

A <u>**relational instance**</u> is a ***set*** of rows all conforming to the same *schema*

The number of columns is the <u>**arity**</u> of the relation

# General form of Grouping and Aggregation

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition $C_1$ on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
3. Apply HAVING condition $C_2$ to each group (may have aggregates)
4. Compute aggregates in SELECT, S, and return the result

# Reminder: Sets vs. Multisets

- In SQL, relations (i.e. tables) are multisets, meaning you can have duplicate tuples

- If you get confused: just state your assumptions!

# Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
            COUNT(day) AS nbd
FROM precipitation,
   (SELECT day, MAX(precip)
    FROM precipitation
    GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

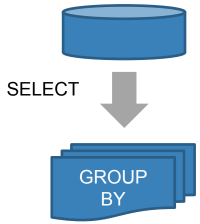
Think about **order*!**

**of the semantics, not the actual execution*

# Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
            COUNT(day) AS nbd
FROM precipitation,
    SELECT day, MAX(precip)
    FROM precipitation
    GROUP BY day
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

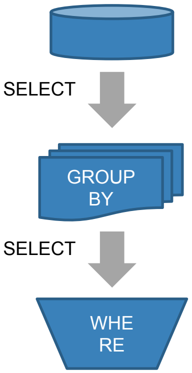



Get the max precipitation **by day**

# Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
            COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip)
     FROM precipitation
     GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```



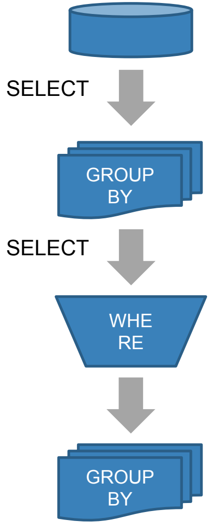

Get the max precipitation **by day**

Get the station, day pairs where / when this happened

# Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
            COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip)
     FROM precipitation
     GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```




Get the max precipitation **by day**

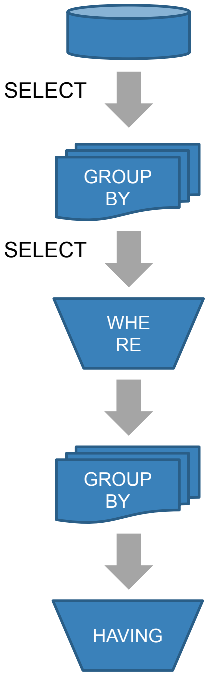
Get the station, day pairs where / when this happened

Group by stations

# Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
            COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip)
     FROM precipitation
     GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```




Get the max precipitation **by day**

Get the station, day pairs where / when this happened

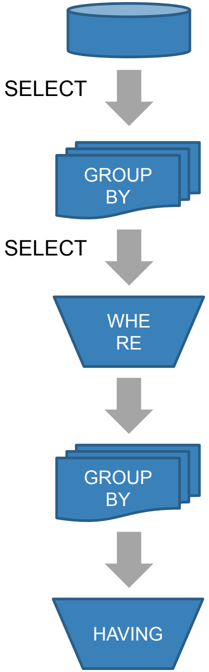
Group by stations

Having > 1 such day

# Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
           COUNT(day) AS nbd
FROM precipitation,
   (SELECT day, MAX(precip)
    FROM precipitation
    GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```




Get the max precipitation **by day**

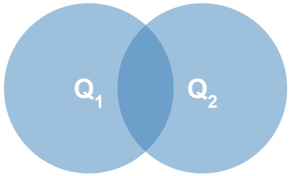
Get the station, day pairs where / when this happened

Group by stations

Having > 1 such day

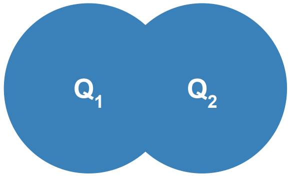
## INTERSECT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
INTERSECT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```



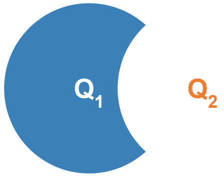

## UNION

```
SELECT  R.A
FROM   R, S
WHERE  R.A=S.A
UNION
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```




## EXCEPT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
EXCEPT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```




Reminder: Operate on Sets!
What if you want multi-sets? [Use ALL → try it out]

# Nested queries: Sub-queries Returning Relations

Company(<u>name</u>, city)
Product(<u>name</u>, maker)
Purchase(<u>id</u>, product, buyer)

```
SELECT c.city
FROM   Company c
WHERE  c.name  IN (
    SELECT pr.maker
    FROM   Purchase p, Product pr
    WHERE  p.product = pr.name
           AND  p.buyer = 'Joe Blow')
```

"Cities where one can find companies that manufacture products bought by Joe Blow"

# Nested queries: Sub-queries Returning Relations

Product(name, price, category, maker)

**ALL**

```
SELECT name
FROM   Product
WHERE  price > ALL(X)
```

Price must be > *all* entries in multiset X

**ANY**

```
SELECT name
FROM   Product
WHERE  price > ANY(X)
```

Price must be > *at least one* entry in multiset X

**EXISTS**

```
SELECT name
FROM   Product p1
WHERE  EXISTS (X)
```

X must be non-empty

**Note that p1 can be referenced in X (correlated query!)*

# Example

Product(name, price, category, maker)

**ALL**

```
SELECT name
FROM   Product
WHERE  price > ALL(
  SELECT price
  FROM   Product
  WHERE  maker = 'G')
```

Find products that are more expensive than ***all products*** produced by "G"

**ANY**

```
SELECT name
FROM   Product
WHERE  price > ANY(
  SELECT price
  FROM   Product
  WHERE  maker = 'G')
```

Find products that are more expensive than ***any one product*** produced by "G"

**EXISTS**

```
SELECT name
FROM   Product p1
WHERE  EXISTS (
  SELECT *
  FROM   Product p2
  WHERE  p2.maker = 'G'
   AND  p1.price = p2.price)
```

Find products where ***there exists some*** product with the same price produced by "G"

# Null Values

- *For numerical operations,* NULL -> NULL:
  - If x = NULL then 4*(3-x)/7 is still NULL

- *For boolean operations,* in SQL there are three values:

**FALSE = 0**
**UNKNOWN = 0.5**
**TRUE = 1**

  - If x= NULL then x="Joe" is UNKNOWN

# Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM   Person
WHERE (age < 25)
 AND (height > 6 AND weight > 190)
```

Won't return e.g.
(age=20
height=NULL
weight=200)!

Rule in SQL: include only tuples that yield TRUE / 1.0

# Null Values

Unexpected behavior:

```
SELECT *
FROM   Person
WHERE  age < 25
   OR  age >= 25
```

Some Persons are not included !

```
SELECT *
FROM   Person
WHERE  age < 25
   OR  age >= 25
   OR  age IS NULL
```

Now it includes all Persons!

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

# RDBMS Architecture

How does a SQL engine work ?

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

| 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! |

# Relational Algebra (RA)

Five **basic** operators:

1. Selection: $\sigma$
2. Projection: $\Pi$
3. Cartesian Product: $\times$
4. Union: $\cup$
5. Difference: -

Derived or auxiliary operators:

- Intersection
- Joins: $\bowtie$ (natural, equi-join, semi-join)
- Renaming: $\rho$

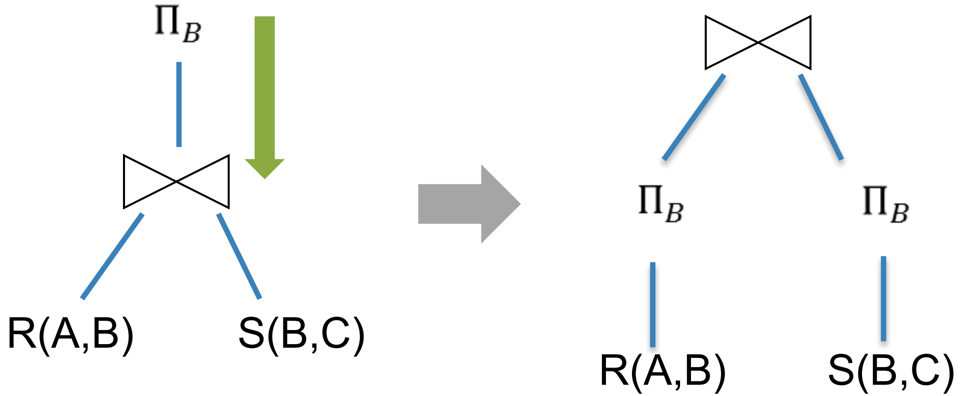
# Converting SFW Query -> RA

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

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

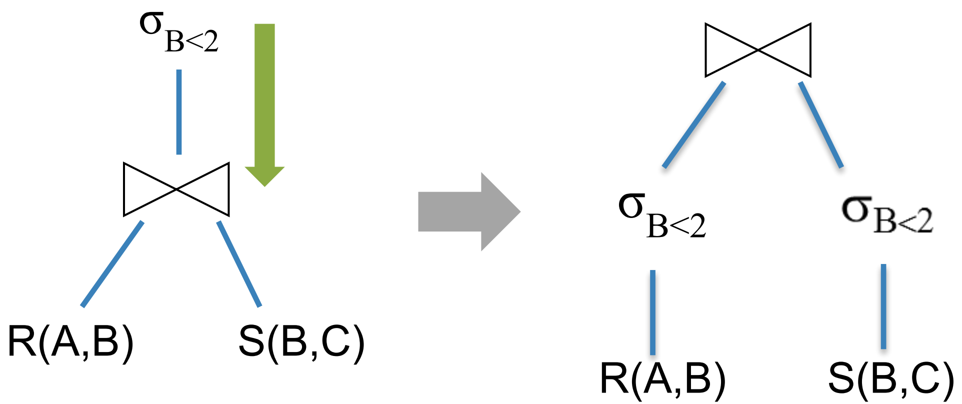
Why must the selections "happen before" the projections?

# Logical Optimization: "Pushing down" projection




Why might we prefer this plan?

# Logical Optimization: “Pushing down” selection




Why might we prefer this plan?

# Basic RA 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!

# RDBMS Architecture

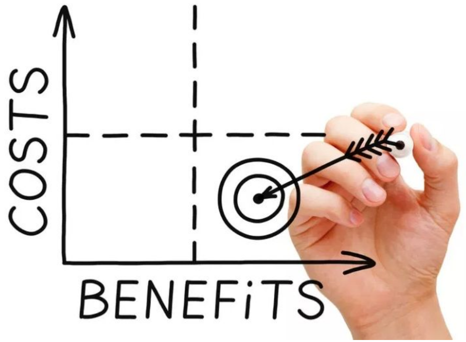
How does a SQL engine work ?

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

| SQL Query | Relational Algebra (RA) Plan | *Optimized* RA Plan | Execution |
|---|---|---|---|
| Declarative query (from user) | Translate to relational algebra expresson | *Find logically equivalent- but more efficient- RA expression* | Execute each operator of the optimized plan! |

# Optimization Roadmap

Build Query Plans




Analyze Plans

1. For SFW, Joins queries
   a. Sort? Hash? Count? Brute-force?
   b. Pre-build an index? B+ tree, Hash?

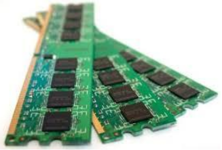
2. What statistics can I keep to optimize?
   a. E.g. Selectivity of columns, values

Cost in I/O, resources?
To query, maintain?

# Big Scaling (with Indexes)

# Roadmap

Primary data structures/algorithms

| | Hashing | Sorting |
| --- | --- | --- |
|  | HashTables<br>($hash_i(x)$) | BucketSort, QuickSort<br>MergeSort |
|  | Hashes for disk location<br>($hash_i(x)$) | MergeSortedFiles<br>SortFiles |
|  | Hashes for machines, shards<br>($hash_i(x)$) | MergeSortedFiles<br>SortFiles |

# IO Aware algorithms

A class of algorithms which try to minimize IO, and

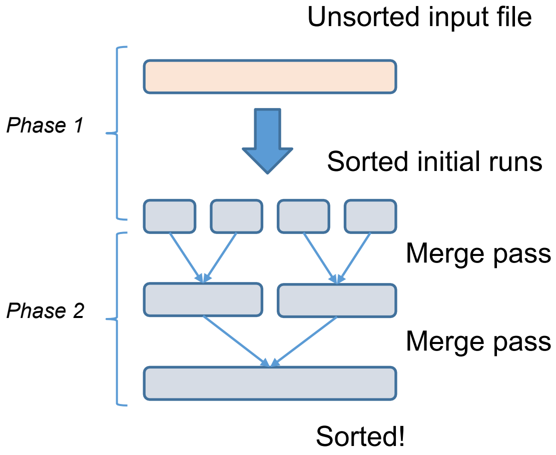
*effectively ignore cost of operations in main memory*

# 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<br>• There are $\left\lceil \frac{N}{B+1} \right\rceil$ of these<br>• The IO cost is 2N<br><br>***Phase 2:*** We do passes of B-way merge until fully merged<br>• Need $\left\lceil \log_B \left\lceil \frac{N}{B+1} \right\rceil \right\rceil$ passes<br>• The IO cost is 2N per pass |

Here we use cost = 1 IO for read and 1 IO for write.
Alternative IO model (e.g, SSDs in HW#2): 1 IO for read and 8 IOs for write?

# 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!
- HPJ: 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!

<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!*

What happens if R and S are swapped?

# 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 each page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Cost:

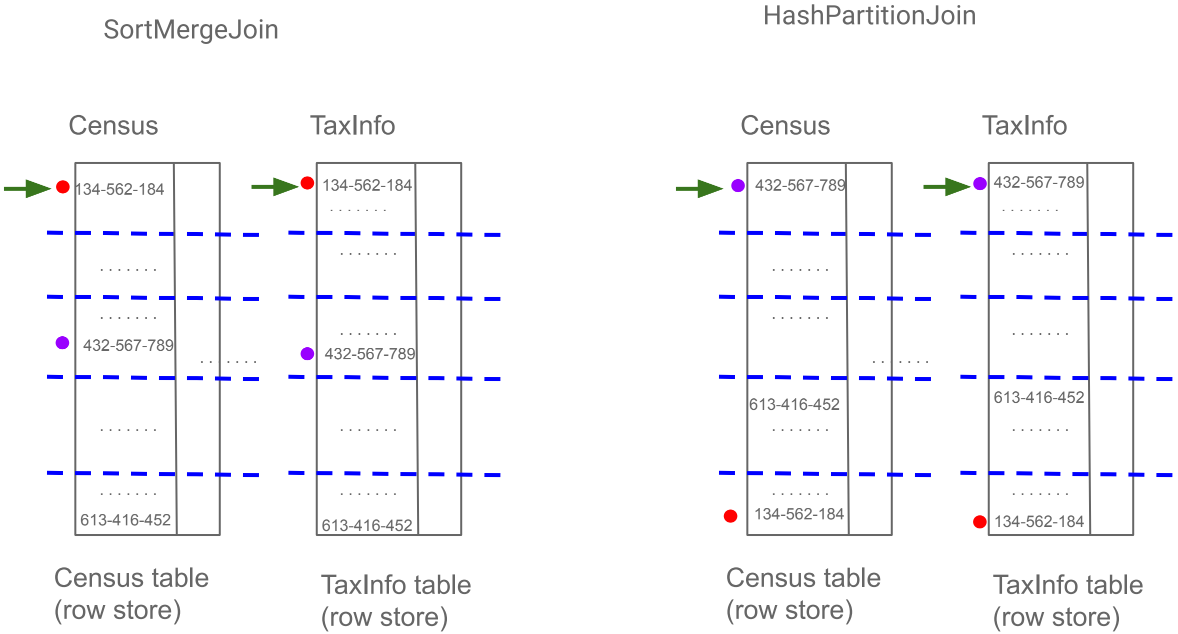
$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S) + \mathrm{OUT}$$

How many R pages to read?
P(R)

How many times is each S page read?
P(R)/(B-1)

Preview of smarter joins

# Pre-process data before JOINing




-- Sort(Census), Sort(TaxInfo) on SSN
-- Merge sorted pages

-- Hash(Census), Hash(TaxInfo) on SSN
-- Merge partitioned pages

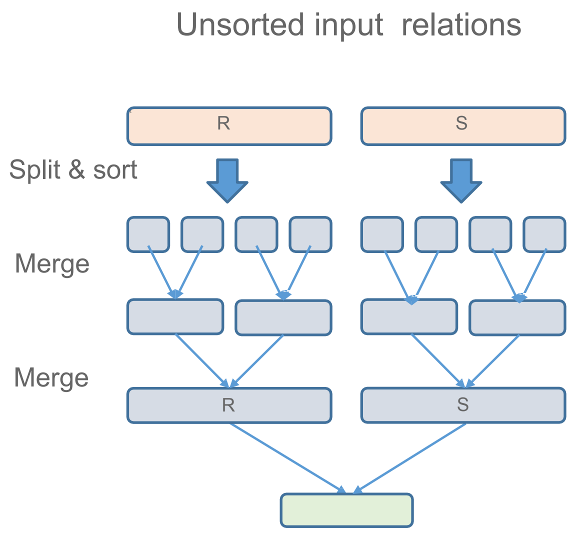
# Sort Merge Join (SMJ)

***Goal:*** Execute R ⋈ S on A

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

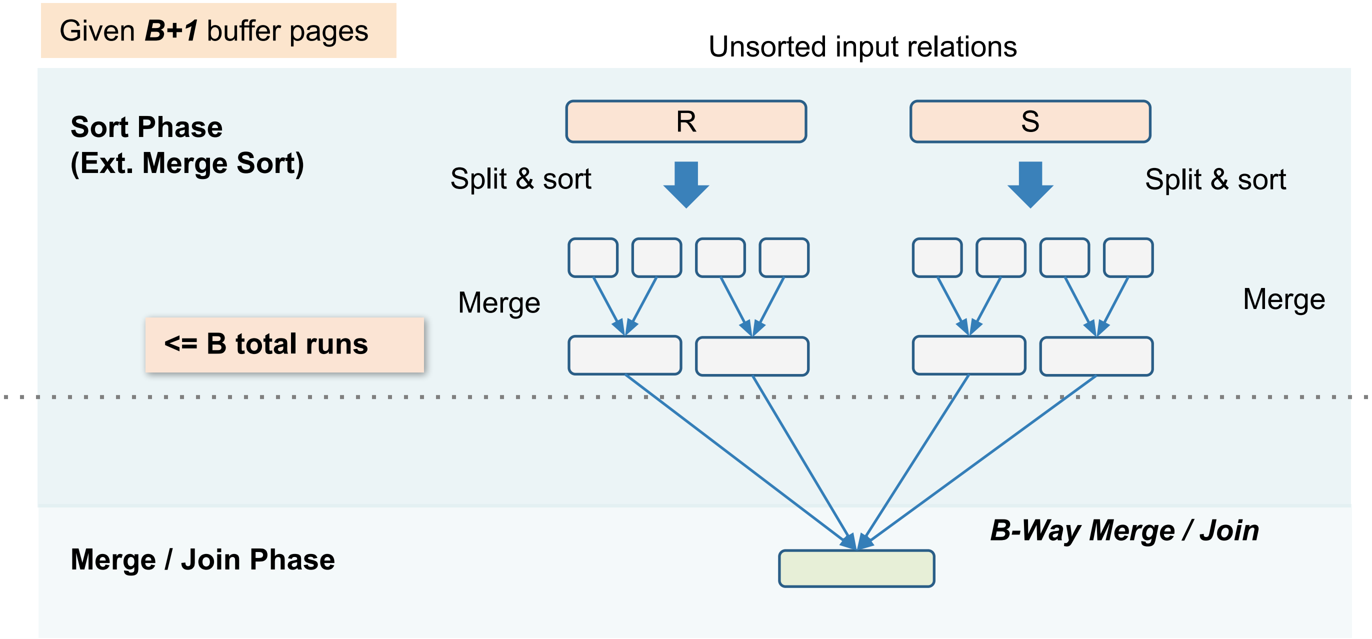
***IO Cost:***

- *Sort phase:* Sort(R) + Sort(S)
- *Merge / join phase:* ~ P(R) + P(S) + OUT




[Reminder: Above assume 1 R = 1 W cost.
IO system could have different Read Cost vs Write Cost] (HW2)

# Simple SMJ Optimization



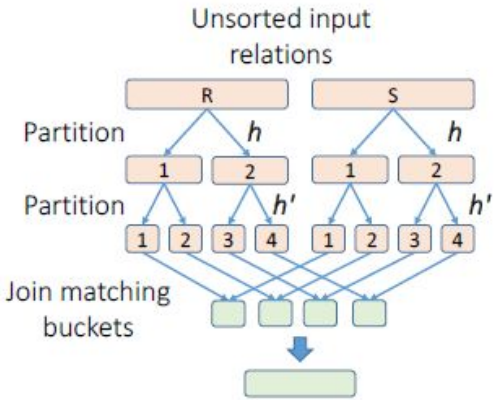

This allows us to “skip” the last sort & save

# 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:***
    - *Hash Partition phase:* 2(P(R) + P(S)) each pass
    - *Partition Join phase: Depends on size of the buckets… can be* ~ P(R) + P(S) + OUT if they are small enough!
        - ***Can be worse due to skew!***



# 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!

# IO analysis

# Recap

Sorting of relational T with N pages

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

Sort N pages with B+1 buffer size

*(vs n log n, for n tuples in RAM. Negligible for large data, vs IO -- much, much slower)*

~ 2 N

Sort N pages when N ~= B

*(because ($Log_B$ 0.5) < 0)*

~ 4 N

Sort N pages when N ~= 2*B^2

*(because ($Log_B$ B) = 1)*

SortMerge and HashJoin for R & S

~3 * (P(R) + P(S)) + OUT

Where P(R) and P(S) are number of pages in R and S, when you have enuf RAM

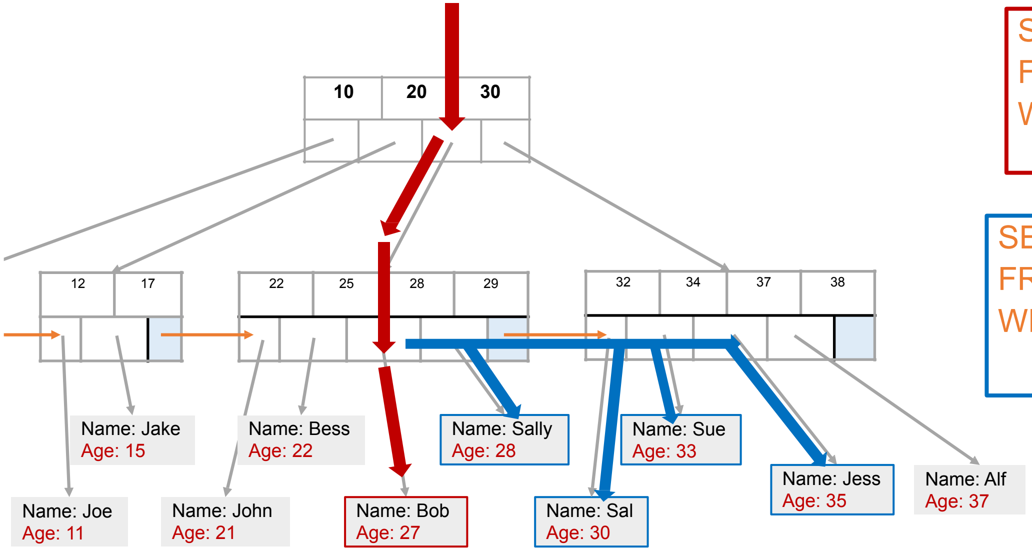
~1 * (P(R) + P(S)) + OUT

For SortMerge, if already sorted

For HashJoin, if already partitioned

We assume cost = 1 IO for read and 1 IO for write.
Alternative IO model (e.g, SSDs in HW#2): 1 IO for read and 8 IOs for write?

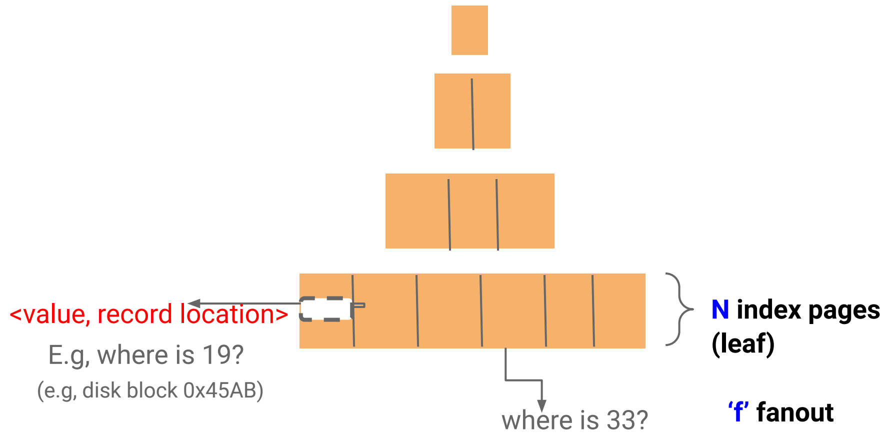
# Searching a B+ Tree

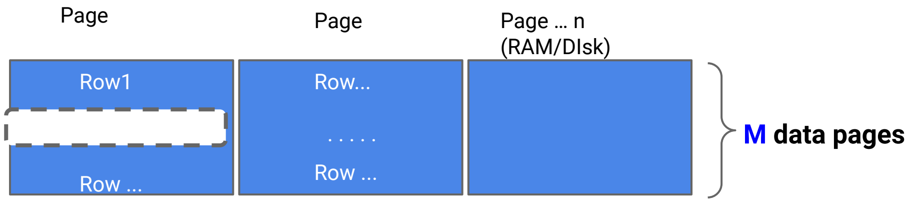



```
SELECT   name
FROM     people
WHERE    age = 27
```

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

# Cost Model for Indexes -- [Baseline simplest model]




"Real" data layout, with full records (including cname, prices, etc.)

Question: What's physical layout? What are costs?

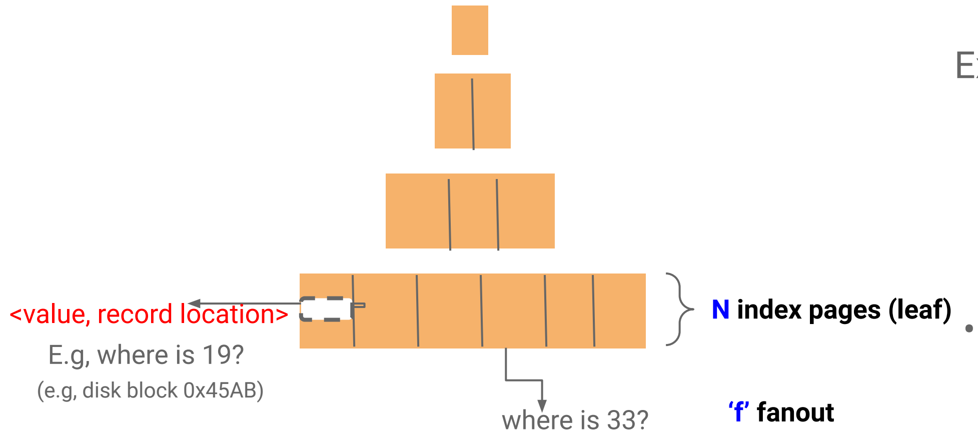
Let:

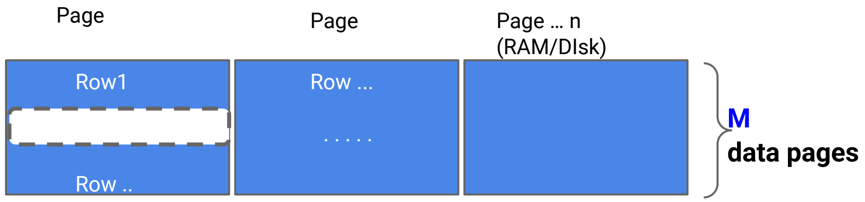
- *f* = fanout *(we'll assume it is constant for our cost model for simplicity…)*
- *N* = number of *pages* we need to index
- Height of tree = $\lceil \log_f N \rceil$

Key intuition

- 'M' depends on Table size
- 'N' depends on number of index values (e.g., <cname> or <cname, price, …> search keys)
- 'f' depends on key size and pointer size

# Cost Model for Indexes -- [Baseline simplest model]




"Real" data layout, with full records
(including cname, prices, etc.)

Example 1:

- N = 2^40 index pages (~1 Trillion pages of 64KBs each)
- Value (or "search key") size = 4 bytes,
- "Location Pointer" size = 8 bytes

- We store one *node* per *page*

  f x 4 + (f+1) x 8 <= 64K → *f ~= 5460*

→ *h = 4* (*i.e.,* $5460^h = 2^{40}$ )

AMAZING, for big 'f'!! What about small 'f?'

# Example 2 -- What about small ‘f = 100’?

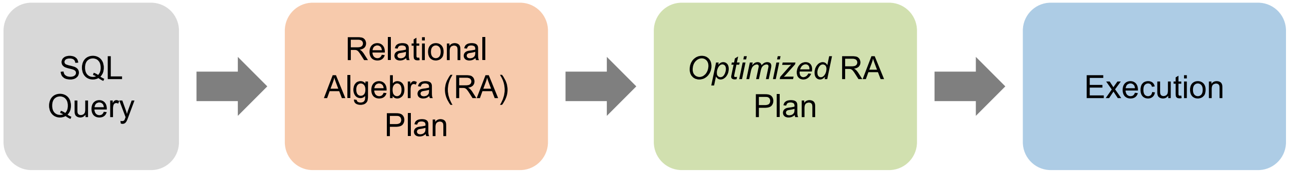
| Level | Number of pages (Size) | Num of Index records |
|---|---|---|
| 1 | 1 (64KB) | 100 |
| 2 | 100 (6.4MB) | 100^2 |
| 3 | 100^2 (0.64GB) | 100^3 |
| 4 | 100^3 (64GB) | 100^4 = 100 Million |
| 5 | ... | ... |

Which levels will be in RAM, if you had
[a] 32 GB of RAM?
[b] 64 GB of RAM?

Other levels? Will (likely) cost a disk IO

# DBMS Architecture

How does a DB 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!