



# Systems Primer

## Bits to disks to clouds

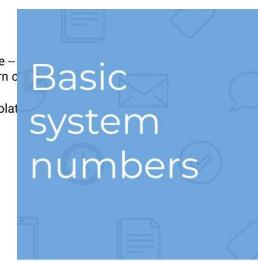
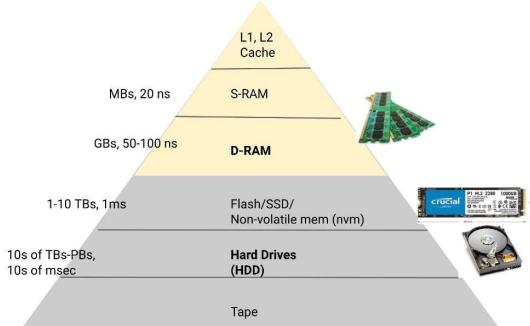
Why? ‘Full stack’ engineer?

# Key Questions

1. Why a rich variety of DBs?
2. How to eval DBs for my data apps?

Data Size	Hardware/ Storage	Algorithms	Languages + Libraries
'Tiny' (< 10 GBs)	Store in RAM	'Usual' CS Algorithms (sort, hash, ..)	Pandas (Python) + SQL  (Spreadsheets for < 100 MBs)
'Small' (10GB - 10TB)			
'Big' (> 10 TBs)			

**Goal:** Gain intuition – What are Speed and Cost tradeoffs?



1 CPU cycle	1 s
Level 1 cache access	3 s
Level 2 cache access	9 s
Level 3 cache access	43 s
Main memory access	6 min
Solid-state disk I/O	2-6 days
Rotational disk I/O	1-12 months
Internet: SF to NYC	4 years
Internet: SF to UK	8 years
Internet: SF to Australia	19 years
OS virtualization reboot	423 years
SCSI command time-out	3000 years
Hardware virtualization reboot	4000 years
Physical system reboot	32 millennia

	Latency (secs)	Scan Throughput (GBs/sec)	Time to move 1 MBs (to CPU)	What you get for ~100\$ (Jun' 22)
RAM	~10 nanosec	~100 GB/sec	10 + 10000 nsec	32 GBs
High-end SSD	~10 microsec	~5 GB/sec	10 + 200 microsec	640GB
HDD Seek (Hard Disk)	~10 millisec	~100 MB /sec	10 + 10 millisec	4 TB
Machine 2 Machine (Network)	~ 1 microsec	~ 5 GB/sec	1 + 200 microsec	

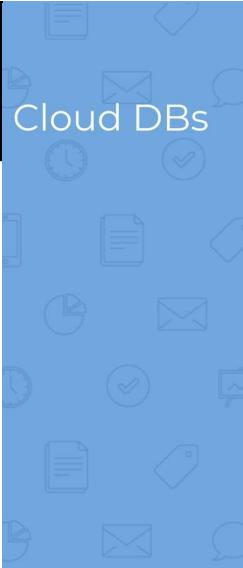
Typical dedicated (non-shared) machine assumptions (unless problem states otherwise):

- 64 GB RAM
- Block sizes: 64 KB (for disk block, RAM page size), 64 MB (for DB block)
- Example: AWS/GCP offer machine instances (e.g. [ec2.r5](#) offers 1-3Gbps network bandwidth, 2CPU/16GB RAM to 96 CPU/768GB RAM for \$\$\$\$ in Nov'21)

## Clouds of machines



(Example [datacenter](#) from 2:50 min)



### Ballpark Cloud DB costs

Data size	Size	Storage Cost	Compute Cost	Cost Range (per yr)
Small	~5 TB	~\$1300	~\$50,000	~\$25k-\$75k
Medium	~50 TB	~\$14000	~\$160,000	~\$100k-\$200k
Large	~200TB	~\$55,000\$	~\$400,000	~\$300k-\$500k

'10-100x' Lower costs in past 10 yrs.

Primary Driver? "db admin" team → Cloud management tools

Reference Snowflake [Jun'22]

(Google for full pricing calculators for BigQuery, RedShift, etc.)

### Rough rule of thumb for data sizes

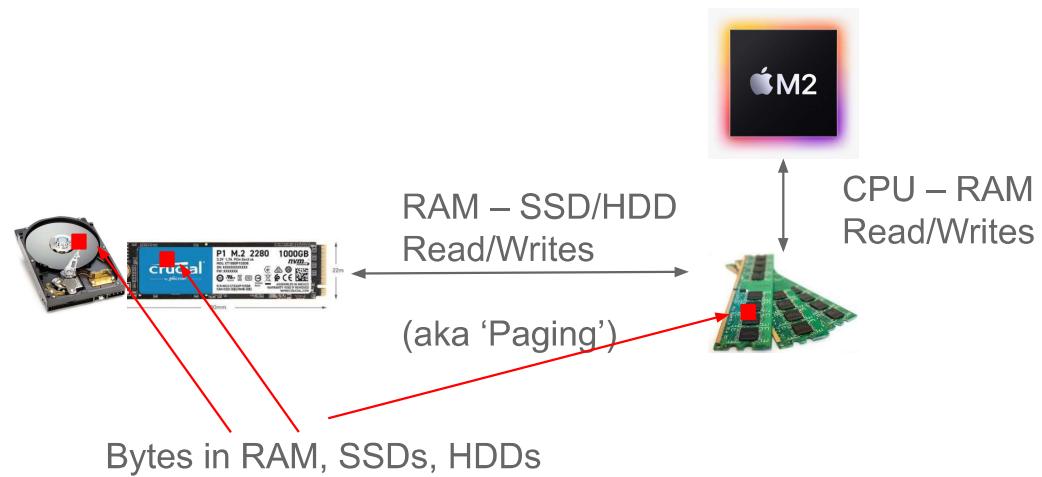
Data size	Size	Algorithms	Storage and Tools
Tiny	< 10 GBs	'Usual' CS Algorithms (sort, hash, ... in RAM)	Pandas (Python) + SQL; Store in RAM
Small	10 GB - 10 TBs	CS 145 Algorithms	SQL on cloud; Store in SSDs
Medium, Large	> 10 TBs	CS 145 Algorithms	SQL on cloud; Store in HDDS (or SSDs for \$\$)

(\*Above is for structured data. Some apps use a

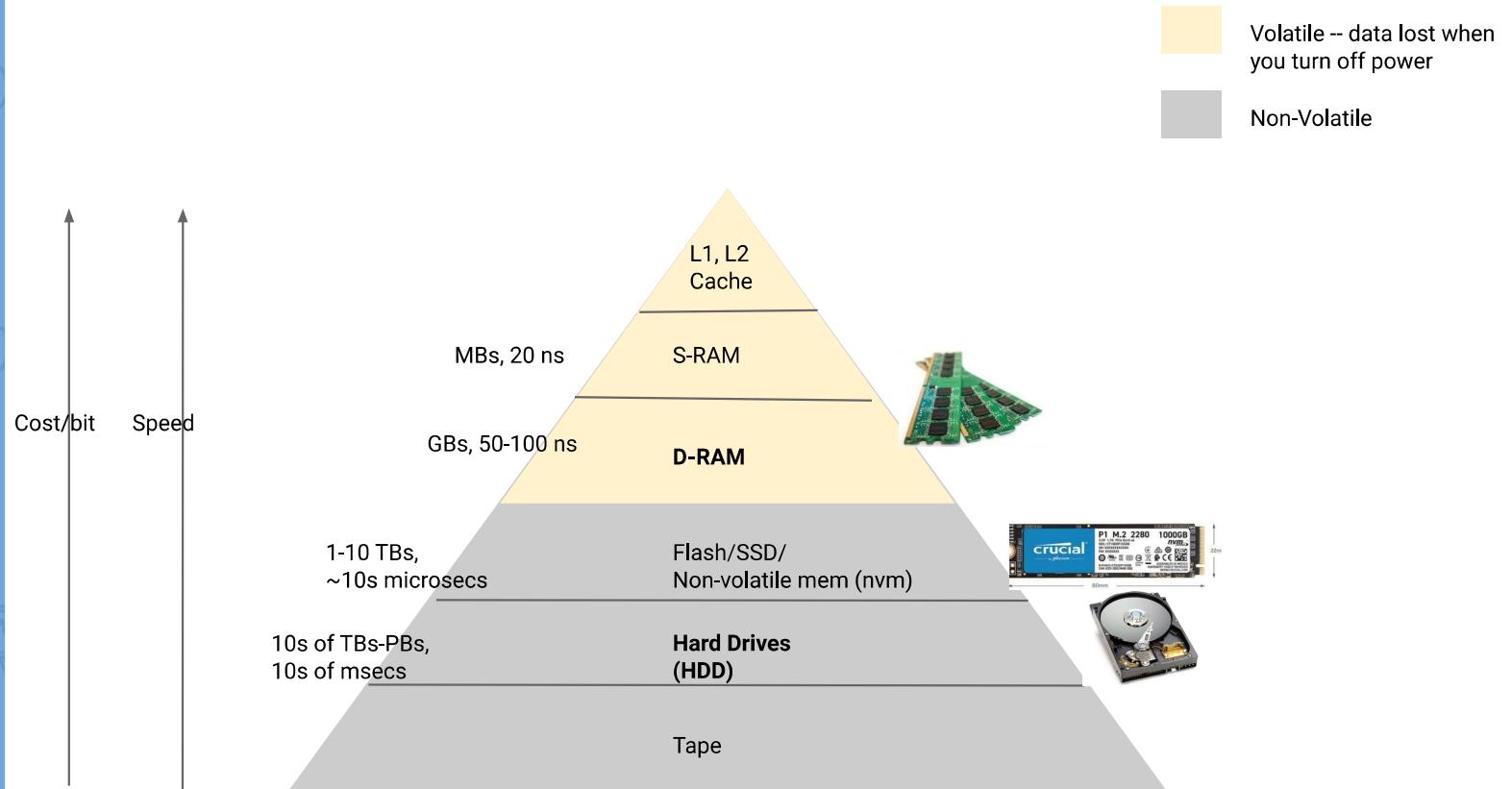
# Basics

## Basics

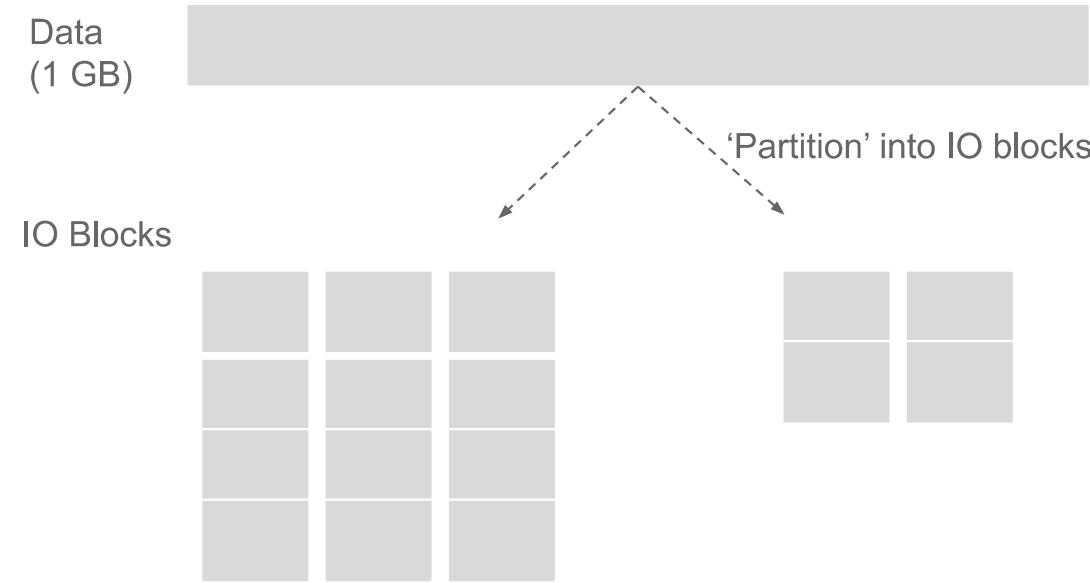
- INT32: 4 bytes, INT64: 8 bytes
- $2^{10} = 1024$ ,  $2^{20} \approx 1\text{ Million}$ ,  $2^{30} \approx 1\text{ Billion}$  ( $10^9$ ),  $2^{40} \approx 1\text{ Trillion}$  ( $10^{12}$ )
- To store int32 records: 1 Million records = 4MB, 1 Billion records = 4 GB
- [Often use 1000 vs 1024, for quick approximations]



# IO Systems



# IO Blocks For Efficiency



E.g., to store 1GB data, we partition the data, and store in IO Blocks

# IO Blocks For Efficiency

## Key Concepts

1. Data is stored in **Blocks** (aka “**partitions**”)
2. Sequential IO is 10x-100x+ faster than ‘many’ random IO
  - E.g., 1 MB (located sequentially) versus 1 Million bytes in random locations
3. HDDs/SSDs copy sequential ‘big’ blocks of bytes to/from RAM



1 Package

1000 Packages  
(1 Trailer)

3000 Packages  
(3 Trailers)

‘M’ Trucks

1 Byte

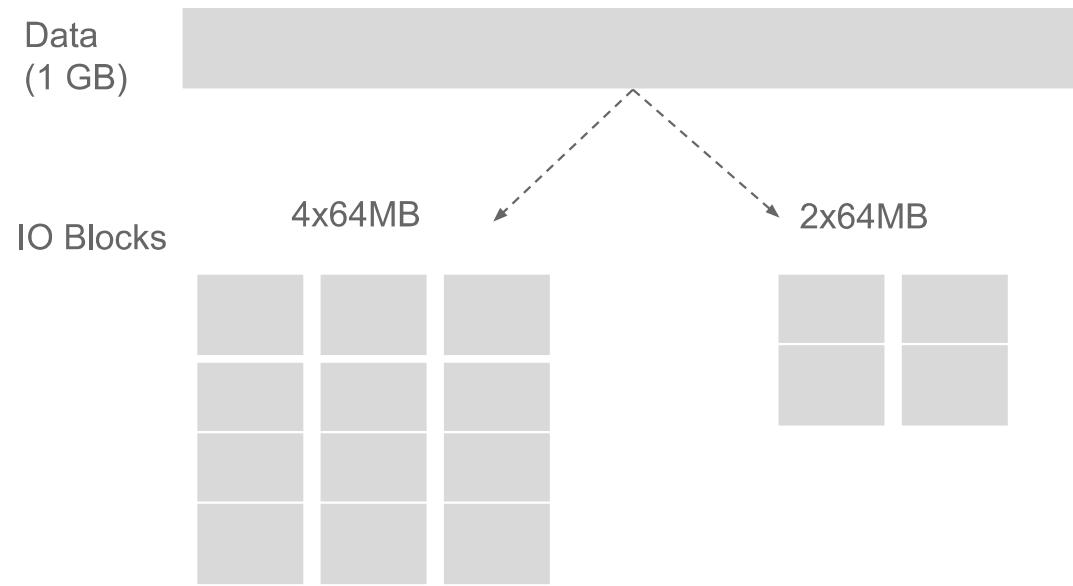
**64MB-Blocks**  
(64MBs of  
sequential, aka  
contiguous bytes)

**Nx64MB-Blocks**

(‘N’ sequential  
64MB-Blocks)

Set of ‘M’  
**Nx64MB-Blocks**

# IO Blocks For Efficiency



E.g., to store 1GB data, we need 15.625 **64MB-Blocks** ( $= 1\text{GB}/64\text{MB}$ ) in

- Sixteen 64MB-Blocks, ( $1 \times 64\text{MB} = 64\text{MB}$ ) OR
- Eight 2x64MB Blocks, (i.e.,  $M = 8, N = 2$ ), OR
- Four 4x64MB Blocks, OR Two 8x64MB Blocks, OR One 16x64B Block
- (OR some combo)

# Read, Write IO Blocks



```
// Read N contiguous Blocks from startLocation
ReadBlocks(startLocation, N Blocks)
1. Access Block's startLocation in IO device
2. Scan N Blocks in one efficient operation. That is
   a. Read N*64MB (or multiple) contiguous bytes to RAM
```

```
// Read M Nx64MB-Blocks for Data
ReadData(M) For i:1...M Blocks
   a. ReadBlocks(startLocation[ith Block], N)
```

**WriteBlocks** is similar to **ReadBlocks**, except  
2b. Write N\*64MB (or multiple) contiguous bytes from RAM

# Basic system numbers



Srigi  
@srigi

"Latency Numbers Every Programmer Should Know"

It is hard for humans to get the picture until you translate it to "human numbers":

1 CPU cycle	1 s
Level 1 cache access	3 s
Level 2 cache access	9 s
Level 3 cache access	43 s
Main memory access	6 min
Solid-state disk I/O	2-6 days
Rotational disk I/O	1-12 months
Internet: SF to NYC	4 years
Internet: SF to UK	8 years
Internet: SF to Australia	19 years
OS virtualization reboot	423 years
SCSI command time-out	3000 years
Hardware virtualization reboot	4000 years
Physical system reboot	32 millennia



	Access Latency (secs)	Scan Throughput (GBs/sec)	What you get for ~100\$ (Jun' 22)
RAM (D-RAM)	~100 nanosec	~100 GB/sec	32 GBs
High-end SSD	~10 microsec	~5 GB/sec	640GB
HDD Seek (Hard Disk)	~10 millisec	~100 MB /sec	4 TB
Machines M1 to M2 (Network)	~ 1 microsec	~ 5 GB/sec	N/A

**Access Latency (secs) = Time to access Block's start location**

**Scan Throughput (GB/sec) = Speed to Scan + Copy data to RAM**

Note: Why are they different speeds?

- Digital (e.g. SSDs and RAM) vs Analog (e.g. [HDD seeks](#))
- Distance from CPU (e.g. RAM is on same chip as CPU vs SSD)

# IO Cost Model



	Access Latency (secs)	Scan Throughput (GBs/sec)	Total Time to read one 64MB-Block
RAM (D-RAM)	~100 nanosec	~100 GB/sec	$100 + 64\text{MB}/100\text{GB/s} = 100 + 640000 \text{nsec}$
High-end SSD	~10 microsec	~5 GB/sec	10 + 12800 microsec
HDD Seek (Hard Disk)	~10 millisec	~100 MB /sec	10 + 640 millisec
Machine 2 Machine (Network)	~ 1 microsec	~ 5 GB/sec	1 + 12800 microsec

**Total Time to ReadData =**  
**AccessLatency \* M + DataSize/ScanThroughput**

Where

- DataSize = data size (in bytes)
- M = Number of non-contiguous Blocks
- AccessLatency = Time to access Block
- ScanThroughput = Speed to copy/scan to RAM

# Example

## Data size

Students			
<b>SID</b>	<b>Name</b>	<b>Address</b>	<b>Bio</b>
40001	Mickey	43 Toontown	Mickey is a Sophomore in CS. He is...
40002	Daffy	147 Main St	Daffy is part of the Orchestra. He was...
50003	Donald	312 Escondido	Donald is a 1st year MS in EE. He was...
50004	Minnie	451 Gates	
10008	Pluto	97 Packard	

Q1: What's **row size** (i.e., size of each record)?

SID: `int32` ⇒ 4 bytes

Student: `char[100]` ⇒ 100 bytes

Address: `char[200]` ⇒ 200 bytes

Bio: `char[720]` ⇒ 720 bytes

## Note: Picking so row size=1024

⇒ Each row is 1024 bytes

Q2: What's **table size**

- with 1000 student rows?  $1000 * 1024$  bytes = 1 MB
- with 1M student rows?  $1M * 1024$  bytes = 1 GB
- With 1B student rows?  $1B * 1024$  bytes = 1 TB

# Example

## Data speed

Q3: For 1 Billion student table of size 1TBs (stored in M=1)



	Access Latency for 1 Block (secs)	Scan Time for 1 TB (sec)
RAM (D-RAM)	~100 nanosec	1TB/100 GBps = ~10 sec
High-end SSD	~10 microsec	1TB/5GBps = ~200 sec
HDD	~10 millisec	1TB/100MBps =~10,000sec
Copy to RAM on Machine Z1 from RAM on Machine Z2	(Network) 1 microsec + (RAM on Z1) 100 nsec + (RAM on Z2) 100 nsec ~ 1.2 microsec [10,000x faster vs reading from Z1's HDD Or 10x faster vs reading from Z1's SSD]	(Network) 1 TB/5GBps + (RAM on Z1) 1 TB/100GBps + (RAM on Z2) 1 TB/100GBps ~= 220 secs [50x faster vs reading from Z1's HDD (~10,000 secs) Or 10% slower vs reading from Z1's SSD]

[Recall Total Time to ReadData =  

$$\text{AccessLatency} * M + \frac{\text{DataSize}}{\text{ScanThroughput}}$$
 ]

# Example Data speed

Q4: With 100 parallel machines? (100x RAM, 100x disks)



	Access Latency for 1 Block (secs) (Same as Q3)	Scan Time for 1 TB (sec) with 100x more capacity? (Same numbers as Q3, divided by 100)
RAM (D-RAM)	~100 nanosec	10 sec/100
High-end SSD	~10 microsec	200 sec/100
HDD	~10 millisec	~10,000/100sec
Copy to RAM on Machines $Z_{1..100}$ from RAM on $Z_{101..200}$	~= 1.2 microsec	~= 220/100 secs

# Example: Find a student, by scanning data

	Design Choices	Storage Cost	Time	Find 'Daffy' from a DB of billion students (1 TB)
Design1	<p><b>Data in RAM</b>            (Scan sequentially &amp; filter)</p>	(@100\$/32GB) <u>3000\$</u>	(AccessLatency) 100 nsec + (Scan) 1000 GB / 100 GBps $\approx \underline{10 \text{ secs}}$	
Design2	<p><b>Data in HDD (no IO Blocks; rows in random spots)</b>            (Access each row. Look for Daffy. Repeat for every row)</p>	(@25\$/TB of disk) <u>25\$</u>	(AccessLatency) 10 msec * 1 Billion rows + (Scan) 1 TB /100 MBps $= 10^7 \text{ secs} + 10^4 \text{ secs}$ $\approx \underline{115 \text{ days}}$	
Design3	<p><b>Data in 64MB-Blocks in HDD</b>            (Access each block. Scan rows, look for Daffy. Repeat per 64MB-block)</p>	(@25\$/TB of disk) <u>25\$</u>	Number of 64MB-blocks = $1 \text{ TB}/64\text{MB} = 15625$ (AccessLatency) 10 msecs *15625 blocks + (Scan) 1 TB /100 MB-sec $= 10.15^4 \text{ secs} \approx \underline{3 \text{ hrs}}$	

In 2 weeks, we'll see how to do this a lot faster (in msec) with good **Indexes** (e.g, hashing)

# Example: Find a student, by scanning data

	Design Choices	Storage Cost	Time	Find 'Daffy' from a DB of billion students (1 TB)
Design4	<p>Data in SSD (no IO Blocks; rows in random spots) (Access each row, look for Daffy; Repeat for every row)</p>	(@150\$/TB of SSD) <u>150\$</u>	(AccessLatency) $10 \text{ microsec} * 1 \text{ Billion rows} + (\text{Scan}) 1 \text{ TB} / 5\text{GBps}$ $= 10^4 \text{ secs} + 200 \text{ secs}$ $\approx \underline{1200 \text{ secs}}$	
Design5	<p>Data in SSD 64MB-Blocks (Access each block, scan rows, look for Daffy; Repeat per 64MB-block)</p>	(@150\$/TB of SSD) <u>150\$</u>	Number of 64MB-blocks = $1 \text{ TB} / 64\text{MB} = 15625$  (AccessLatency) $10 \text{ microsecs} * 15625 \text{ blocks} + (\text{Scan}) 1 \text{ TB} / 5\text{GBps}$ $= 10.15^4 \text{ secs}$ $\approx \underline{200 \text{ secs}}$	

In 2 weeks, we'll see how to do this a lot faster (in msec) with good **Indexes** (e.g. hashing)

# Clouds of machines



(Example [datacenter](#)  
from 2:50 min)



Typical dedicated (non-shared) machine assumptions (unless problem states otherwise):

- 64 GB RAM
- Block sizes: 64 MB-DB block (and RAM page size)
- Example: AWS/GCP offer machine instances  
(e.g. [ec2.r5](#) offers 1-3Gbps network bandwidth, 2CPU/16GB RAM to 96 CPU/768GB RAM for \$\$\$\$ in Nov'21)

# Cloud DBs

[Optional]

## Ballpark Cloud DB costs

Data size	Size	Storage Cost	Compute Cost	Cost Range (per yr)
Small	<b>~5 TB</b>	~\$1300	~\$50,000	~\$25k-\$75k
Medium	<b>~50 TB</b>	~\$14000	~\$160,000	~\$100k-\$200k
Large	<b>~200TB</b>	~\$55,000\$	~\$400,000	~\$300k-\$500k

'10-100x' Lower costs in past 10 yrs.

Primary Driver? "db admin" team → Cloud management tools

[Reference](#) Snowflake [Jun'22]  
(Google for full pricing calculators  
for BigQuery, RedShift, etc.)

## Rough rule of thumb for data sizes

Data Size	Hardware/ Storage	Algorithms	Languages + Libraries
'Tiny' (< 10 GBs)	Store in RAM	'Usual' CS Algorithms (sort, hash, ..)	Pandas (Python) + SQL  (Spreadsheets for < 100 MBs)
'Small' (10GB - 10TB)	Store in SSD	CS145 Algorithms	SQL on Cloud
'Big' (> 10 TBs)	Store in HDD	CS145 Algorithms	SQL on Cloud

(\*For popular Tables data.  
Some apps use JSON  
(hierarchical data) in  
nonSQL DBs – see later)

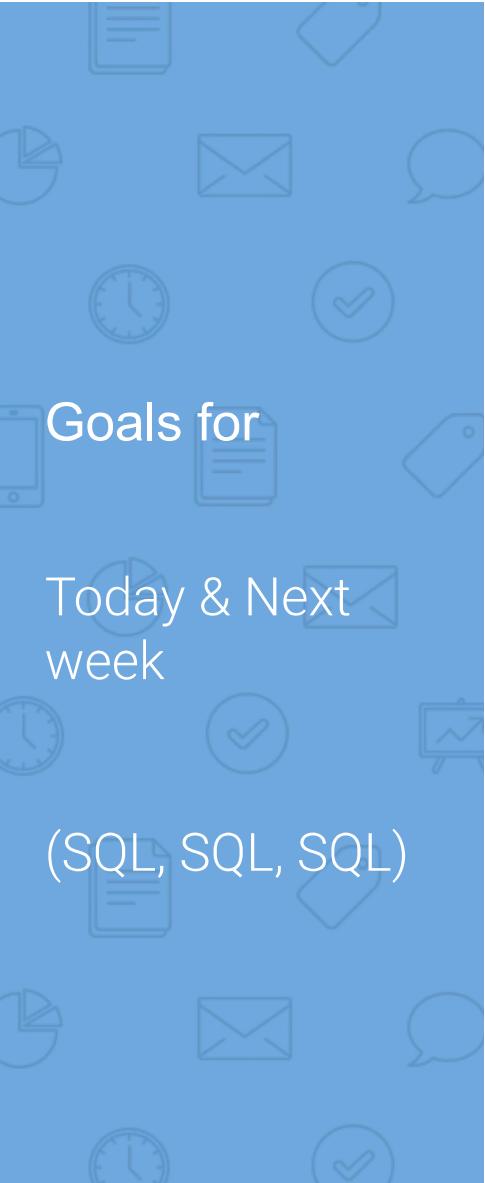


## In this Section

- Systems – Size, Cost, and Speeds of GB/TBs
  - On RAM, HDDs/SSDs, clouds
- IO Cost model
  - Simple IO model for Total Time (for RAM, SSD, HDD)
  - Good estimate – Reality is a bit more complex to fully model (buses, cache hit rates, etc.)
- Examples analyzing system design choices



# Intuition for SQL



## Phase I: Intuition for SQL (1st half of today)

Basic Relational model (aka tables)

Example SQL (exploring real datasets)

## Phase II: Formal description

SQL concepts we'll study (similar to Python map-reduce)

Schemas, Query structure of SELECT-FROM-WHERE, JOINs, etc



## A Motivating Example

A basic Course Management System (CMS):

*Entities (e.g., Students, Courses, Professors)*

*Relationships (e.g., Who takes what, Who teaches what?)*

# Simple DB == Spreadsheets

	C	D	E	F	G	H	I	
1								
2								
3								
4	Students							
5	SID	Name	GPA					
6	40001	Mickey	3.2					
7	40002	Daffy	3.6					
8	50003	Donald	3.3					
9	50004	Minnie	3.9					
10	10008	Pluto	4					
11								
12								
13	Courses							
14	CID	C-Name	Room					
15	1012	CS145 - Toon systems	Nvidia					
16	1017	CS161 - Animation art	Gates 300					
17	1019	CS245 - Painting town red	Packard 45					
18								
19								

## Tables

Student(sid: string, name: string, gpa: float)

Courses(cid: string, c-name: string, room: string)

Enrolled(sid: string, cid: string, grade: string)

sid: Connects Enrolled to Students

cid: Connects Enrolled to Courses

## Queries ["compute" over tables]

- Minnie's GPA?
- AVG student GPA?
- Mickey's classes?
- AVG student GPA in CS145?



# Example SQL

Part I

# Example 1

## Sun Roof potential

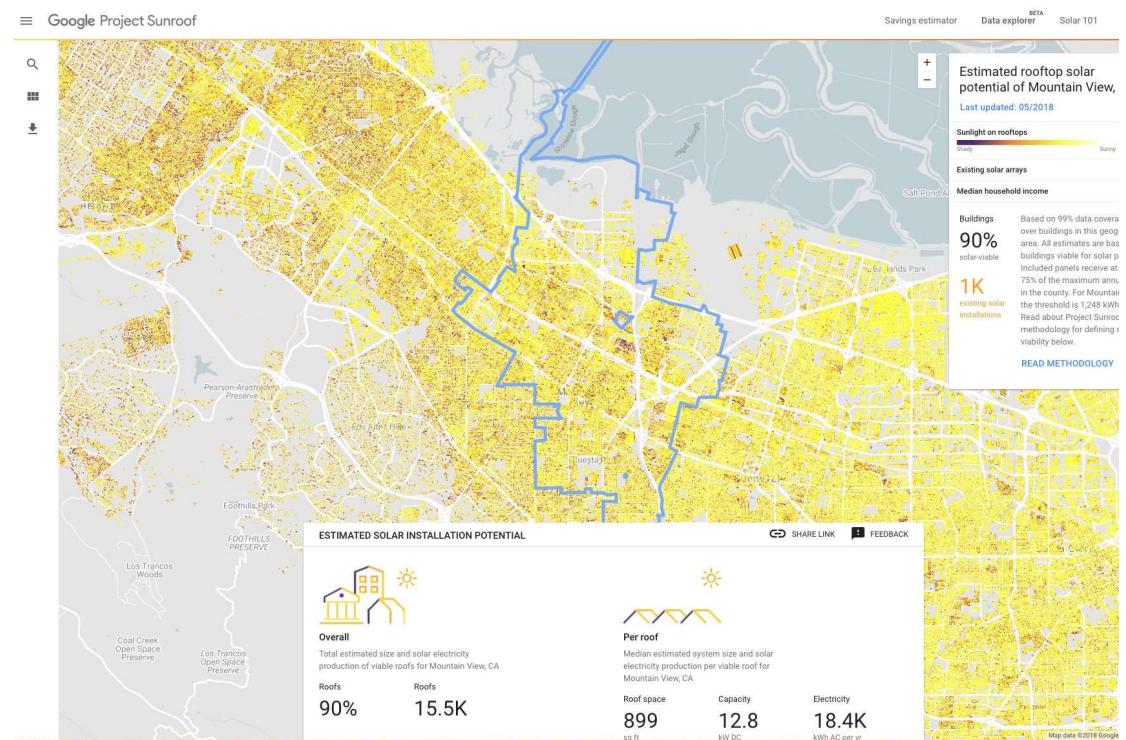
from Satellite images



# Example 1

SunRoof potential

## SunRoof explorer



## Example 1

Public dataset

Public Dataset: Solar potential by postal code

region_name	percent_covered	kw_total	carbon_offset_metric_tons
94043	97.79146031321109	215612.5	84929.00985071347
94041	99.05200433369447	56704.25	22189.34823862318

Public Dataset: USA.population by zip2010

zipcode	population
99776	124
38305	49808
37086	31513
41667	720
67001	1676

# Example 1

SunRoof

On BigQuery  
Public dataset

What is the solar potential of Mountain View, CA? [\[Run query\]](#)

## Saved Query: MTV sunroof [edited]

```
1 #StandardSQL
2 SELECT
3     region_name,
4     percent_covered,
5     kw_total,
6     carbon_offset_metric_tons
7 FROM `bigquery-public-data.sunroof_solar.solar_potential_by_postal_code`
8 WHERE
9     region_name = '94040'
10    OR region_name = '94041'
11    OR region_name = '94043'
```

Standard SQL Dialect X

Ctrl + Enter: run

**RUN QUERY** ▾

Save Query

Save View

Format Query

Schedule Query

Show Options

Query complete (1.6s elapsed, 346 KB processed)

Results Details

Download as CSV

Download as JSON

Save

Row	region_name	percent_covered	kw_total	carbon_offset_metric_tons
1	94043	97.79146031321109	215612.5	84929.00985071347
2	94041	99.05200433369447	56704.25	22189.34823862318
3	94040	98.9440337909187	139745.5	55039.74974407879

## Example 2

SunRoof

Public dataset  
On BigQuery

How many metric tons of carbon would we offset, if building in communities with 100% coverage all had solar roofs? [\[Run query\]](#)

Saved Query: CO2 offset in 100percent zips

```
1 #StandardSQL
2 SELECT
3   ROUND(SUM(s.carbon_offset_metric_tons),2) total_carbon_offset_possible_metric_tons
4 FROM `bigquery-public-data.sunroof_solar.solar_potential_by_postal_code` s
5 JOIN `bigquery-public-data.census_bureau_usa.population_by_zip_2010` c
6 ON s.region_name = c.zipcode
7 WHERE
8   percent_covered = 100.0
9   AND c.population > 0
10
11
12
13
```

Ctrl + Enter: run qu

Standard SQL Dialect

RUN QUERY

Save View

Format Query

Schedule Query

Show Options

Query com

Results

Details

Download as CSV

Download as JSON

Save as

Row	total_carbon_offset_possible_metric_tons
-----	--

1	3689508.33
---	------------

## Example 2

How many metric tons of carbon would we offset, per zipcode?

↪ Saved Query: CO2 offset in 100percent zips [edited] [?](#)

```
1 #StandardSQL
2 SELECT
3     zipcode, ROUND(SUM(s.carbon_offset_metric_tons),2) total_carbon_offset_possible_metric
4 FROM `bigquery-public-data.sunroof_solar.solar_potential_by_postal_code` s
5 JOIN `bigquery-public-data.census_bureau_usa.population_by_zip_2010` c
6 ON s.region_name = c.zipcode
7 WHERE
8     percent_covered = 100.0
9     AND c.population > 0
10 GROUP BY c.zipcode
11
12
13
```

Standard SQL Dialect [X](#) Ctrl + Enter: run c

[RUN QUERY](#) [Save Query](#) [Save View](#) [Format Query](#) [Schedule Query](#) [Show Options](#)

Query complete (2.5s elapsed, 23.5 MB processed)

Results	Details	<a href="#">Download as CSV</a>	<a href="#">Download as JSON</a>	<a href="#">Save a</a>
Row	zipcode	total_carbon_offset_possible_metric_tons		
1	35119	3417.26		
2	10165	162.1		
3	21810	6650.09		
4	74078	61515.66		
5	47876	5544.3		
6	10170	831.22		



## Example 2

SunRoof

How many metric tons of carbon would we offset, per zipcode sorted?

↪ Saved Query: CO2 offset in 100percent zips [edited] [?](#)

Query Editor UDF

```
1 #StandardSQL
2 SELECT
3     zipcode, ROUND(SUM(s.carbon_offset_metric_tons),2) total_carbon_offset_possible_metric_tons
4     FROM `bigquery-public-data.sunroof_solar.solar_potential_by_postal_code` s
5     JOIN `bigquery-public-data.census_bureau_usa.population_by_zip_2010` c
6     ON s.region_name = c.zipcode
7 WHERE
8     percent_covered = 100.0
9     AND c.population > 0
10    GROUP BY c.zipcode
11    ORDER BY total_carbon_offset_possible_metric_tons
12    DESC
13
14
```

Ctrl + Enter: run query, Tab or Ctrl + Space

Standard SQL Dialect [X](#)

RUN QUERY [▼](#)

Save Query

Save View

Format Query

Schedule Query

Show Options

Query complete (2.8s elapsed, 23.5 MB processed)

Results Details

Download as CSV

Download as JSON

Save as Table

Save to

Row	zipcode	total_carbon_offset_possible_metric_tons
1	18503	715700.55
2	44243	271861.55
3	38677	266787.12
4	96860	225850.35
5	47809	141087.91

## Reminder

## Special Databases



# Query with SQL, universally over 'all' DBs

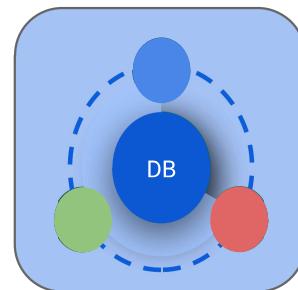
SQL

```
= QUERY(T,  
"SELECT c1, c2  
FROM T  
WHERE condition;")
```

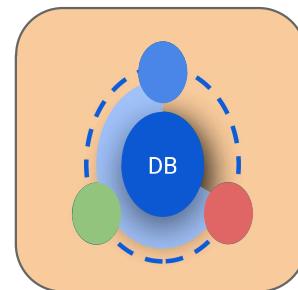
```
SELECT c1, c2  
FROM T  
WHERE condition;
```

```
results =  
spark.SQL(  
"SELECT c1, c2  
FROM T  
WHERE condition;")
```

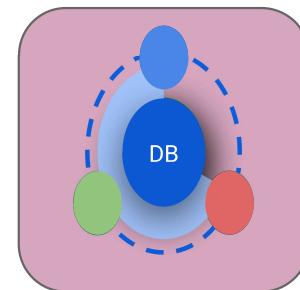
DB



'Spreadsheets'



GCP BigQuery, AWS Redshift,  
MySQL, PostgreSQL, Oracle



Spark, Hadoop

Data



100s of Scaling algorithms/systems? [Weeks 3..]

- Data layout? [Row vs columns...]
- Data structs? [Indexing...]

# Preview

## SQL queries

[sqltutorial.org/sql-cheat-sheet](http://www.sqltutorial.org/sql-cheat-sheet)

## SQL CHEAT SHEET <http://www.sqltutorial.org>

### QUERYING DATA FROM A TABLE

**SELECT c1, c2 FROM t;**

Query data in columns c1, c2 from a table

**SELECT \* FROM t;**

Query all rows and columns from a table

**SELECT c1, c2 FROM t**

**WHERE condition;**

Query data and filter rows with a condition

**SELECT DISTINCT c1 FROM t**

**WHERE condition;**

Query distinct rows from a table

**SELECT c1, c2 FROM t**

**ORDER BY c1 ASC [DESC];**

Sort the result set in ascending or descending order

**SELECT c1, c2 FROM t**

**ORDER BY c1**

**LIMIT n OFFSET offset;**

Skip offset of rows and return the next n rows

**SELECT c1, aggregate(c2)**

**FROM t**

**GROUP BY c1;**

Group rows using an aggregate function

**SELECT c1, aggregate(c2)**

**FROM t**

**GROUP BY c1**

**HAVING condition;**

Filter groups using HAVING clause

### QUERYING FROM MULTIPLE TABLES

**SELECT c1, c2**

**FROM t1**

**INNER JOIN t2 ON condition;**

Inner join t1 and t2

**SELECT c1, c2**

**FROM t1**

**LEFT JOIN t2 ON condition;**

Left join t1 and t2

**SELECT c1, c2**

**FROM t1**

**RIGHT JOIN t2 ON condition;**

Right join t1 and t2

**SELECT c1, c2**

**FROM t1**

**FULL OUTER JOIN t2 ON condition;**

Perform full outer join

**SELECT c1, c2**

**FROM t1**

**CROSS JOIN t2;**

Produce a Cartesian product of rows in tables

**SELECT c1, c2**

**FROM t1, t2;**

Another way to perform cross join

**SELECT c1, c2**

**FROM t1 A**

**INNER JOIN t2 B ON condition;**

Join t1 to itself using INNER JOIN clause

### USING SQL OPERATORS

**SELECT c1, c2 FROM t1**

**UNION [ALL]**

**SELECT c1, c2 FROM t2;**

Combine rows from two queries

**SELECT c1, c2 FROM t1**

**INTERSECT**

**SELECT c1, c2 FROM t2;**

Return the intersection of two queries

**SELECT c1, c2 FROM t1**

**MINUS**

**SELECT c1, c2 FROM t2;**

Subtract a result set from another result set

**SELECT c1, c2 FROM t1**

**WHERE c1 [NOT] LIKE pattern;**

Query rows using pattern matching %, \_

**SELECT c1, c2 FROM t**

**WHERE c1 [NOT] IN value\_list;**

Query rows in a list

**SELECT c1, c2 FROM t**

**WHERE c1 BETWEEN low AND high;**

Query rows between two values

**SELECT c1, c2 FROM t**

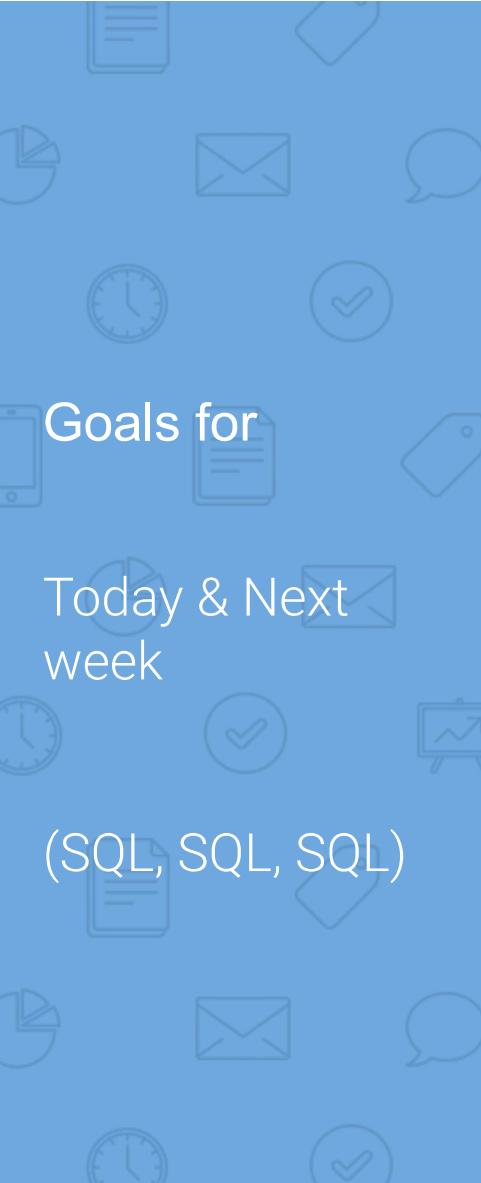
**WHERE c1 IS [NOT] NULL;**

Check if values in a table is NULL or not



# Introduction to SQL

Part I



## Phase I: Intuition for SQL (1st half of today)

Basic Relational model (aka tables)

Example SQL (exploring real datasets)

## Phase II: Formal description

SQL concepts we'll study (similar to Python map-reduce)

Schemas, Query structure of SELECT-FROM-WHERE, JOINs, etc



## Concepts In this section

1. SQL introduction & schema definitions
2. Basic single-table queries
3. Multi-table queries



# SQL Introduction

- SQL is a standard language for querying and manipulating data
- SQL is a **very high-level** programming language  
This works because it is optimized well!
- Many standards out there:  
ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ....

*NB:* Probably the world's most successful **parallel** programming language (multicore?)

SQL stands for  
Structured  
Query  
Language



# Tables

Product

A relation or table is a multiset of rows, with columns.

The schema of a table is the table name, its columns, and their types.

PName	Price	Manuf
Gizmo	\$19.99	GizmoWorks
Powergizmo	\$29.99	GizmoWorks
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

Each row (or tuple or record) has attributes

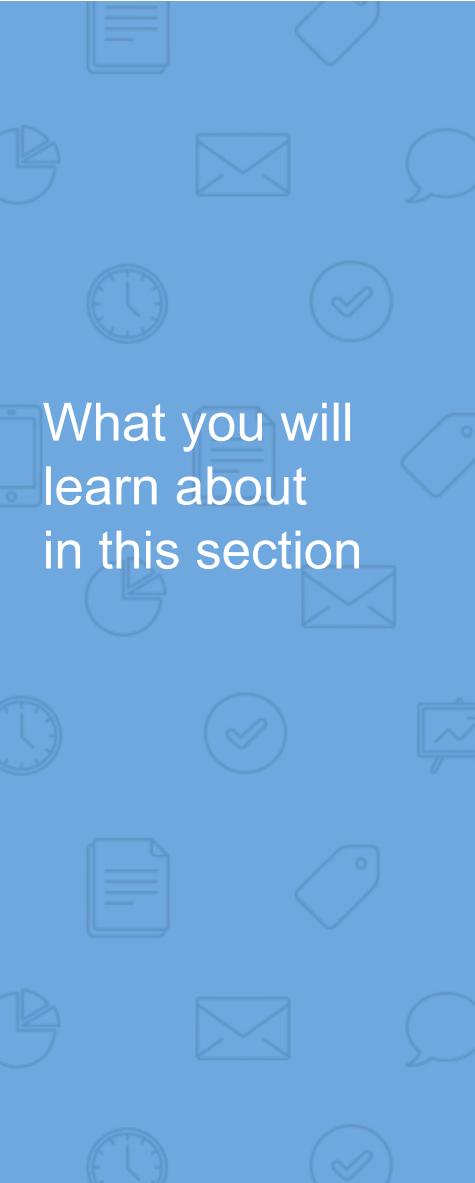
The number of tuples is the cardinality of the relation

Each column (or attribute) has a type

The number of columns is the arity of the relation.



# Single - table queries



What you will  
learn about  
in this section

1. The SFW query
2. Other useful operators: LIKE, DISTINCT, ORDER BY



# SQL Query

Basic form (there are many many more bells and whistles)

**SELECT <attributes>**

**FROM <one or more relations>**

**WHERE <conditions>**

Call this a **SFW** query.



# Simple SQL Query: Selection

**Selection** is the operation of filtering a relation's tuples on some condition

```
SELECT *  
FROM Product  
WHERE Category =  
'Gadgets'
```

PName	Price	Category	Manuf
Gizmo	\$19.99	Gadgets	GWorks
Powergizmo	\$29.99	Gadgets	GWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi



PName	Price	Category	Manuf
Gizmo	\$19.99	Gadgets	GWorks
Powergizmo	\$29.99	Gadgets	GWorks



# Simple SQL Query: Projection

**Projection** is the operation of producing an output table with tuples that have a subset of their prior attributes

```
SELECT Pname, Price,  
Manufacturer  
  
FROM Product  
  
WHERE Category = 'Gadgets'
```

PName	Price	Category	Manuf
Gizmo	\$19.99	Gadgets	GWorks
Powergizmo	\$29.99	Gadgets	GWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi



PName	Price	Manuf
Gizmo	\$19.99	GWorks
Powergizmo	\$29.99	GWorks



# Notation

Input Schema

Product(PName, Price, Category, Manufacturer)

```
SELECT Pname, Price, Manufacturer  
FROM Product  
WHERE Category = 'Gadgets'
```

Output Schema

Answer(PName, Price, Manufacturer)





# LIKE: Simple String Pattern Matching

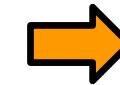
```
SELECT *
FROM Products
WHERE PName LIKE '%gizmo%'
```

- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - \_ = any single character



# DISTINCT: Eliminating Duplicates

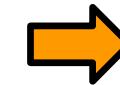
```
SELECT DISTINCT Category  
FROM Product
```



Category
Gadgets
Photography
Household

Versus

```
SELECT Category  
FROM Product
```



Category
Gadgets
Gadgets
Photography
Household



# ORDER BY: Sorting the Results

```
SELECT PName, Price, Manufacturer  
FROM Product  
WHERE Category='gizmo' AND Price > 50  
ORDER BY Price, PName
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.



## A Few Details (1/3)

1. **SQL commands** are case insensitive:
  - a. SELECT = Select, Product = product
2. **Values** are **not**: 'Seattle' vs 'seattle'
3. Use single quotes for constants:
  - a. 'abc' - best practice (versus "abc" with mixed support)
4. To say "don't know the value" we use **NULL**
  - a. Common, but annoying (more detail later)
  - b. E.g., Student GPA in 1st quarter = **NULL**, not zero



## A Few Details (2/3)

Atomic types for columns:

Characters: CHAR(20), VARCHAR(50)

Numbers: INT, BIGINT, SMALLINT, FLOAT

Others: MONEY, DATETIME...

(Most SQL dialects support **record** types, e.g. json-like. Works in a similar fashion. Optional: for cs145.)



# Declaring Schema

```
CREATE TABLE Product (
    Pname CHAR(20),
    Manufacturer VARCHAR(50),
    price float,
    Category VARCHAR(50),
    PRIMARY KEY (Pname, Manufacturer))
```

.....  
Product(Pname: string, Price: float, Category: string, Manufacturer: string)



# Keys

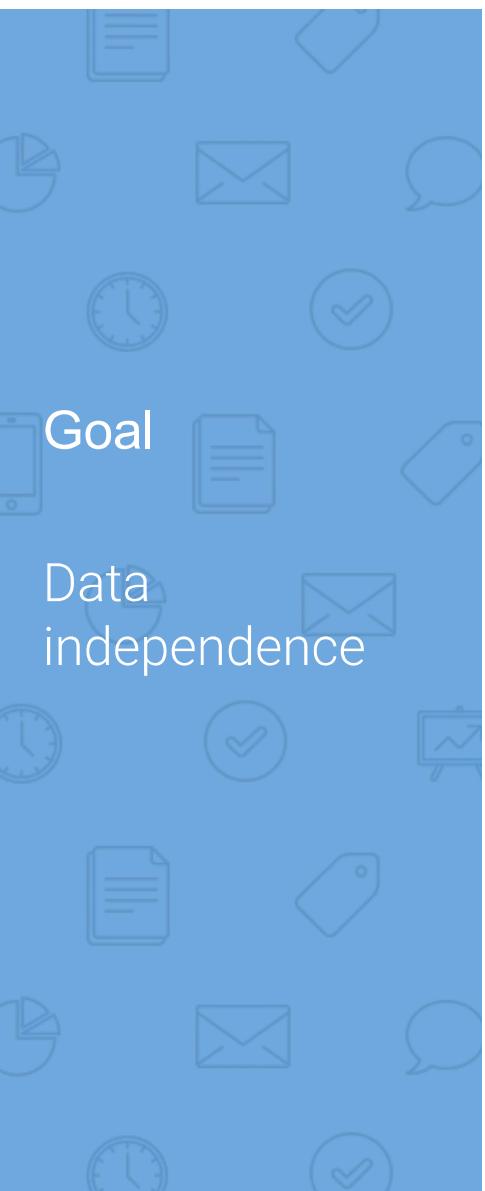
A **key** is a **minimal subset of columns** that acts as a unique identifier for tuples in a relation

- i.e. if two tuples agree on the values of the key, then they must be the same tuple!

Product(Pname: *string*, Price: *float*, Category: *string*, Manufacturer: *string*)

Design choices?

1. Which would you select as a key?
2. Is a key always guaranteed to exist?
3. Can we have more than one key?



# Data independence

1. Can we add a new column or attribute without rewriting the application?
  - a. YES! Logical Data Independence
2. Should you care which disks/machines are the data stored on?
  - a. NO! Physical Data Independence

# Preview

## SQL queries

[sqltutorial.org/sql-cheat-sheet](http://www.sqltutorial.org/sql-cheat-sheet)

## SQL CHEAT SHEET <http://www.sqltutorial.org>

### QUERYING DATA FROM A TABLE

`SELECT c1, c2 FROM t;`  
Query data in columns c1, c2 from a table

`SELECT * FROM t;`  
Query all rows and columns from a table

`SELECT c1, c2 FROM t  
WHERE condition;`  
Query data and filter rows with a condition

`SELECT DISTINCT c1 FROM t  
WHERE condition;`  
Query distinct rows from a table

`SELECT c1, c2 FROM t  
ORDER BY c1 ASC [DESC];`  
Sort the result set in ascending or descending order

`SELECT c1, c2 FROM t  
ORDER BY c1  
LIMIT n OFFSET offset;`  
Skip offset of rows and return the next n rows

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1;`  
Group rows using an aggregate function

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1  
HAVING condition;`  
Filter groups using HAVING clause

### QUERYING FROM MULTIPLE TABLES

`SELECT c1, c2  
FROM t1  
INNER JOIN t2 ON condition;`  
Inner join t1 and t2

`SELECT c1, c2  
FROM t1  
LEFT JOIN t2 ON condition;`  
Left join t1 and t2

`SELECT c1, c2  
FROM t1  
RIGHT JOIN t2 ON condition;`  
Right join t1 and t2

`SELECT c1, c2  
FROM t1  
FULL OUTER JOIN t2 ON condition;`  
Perform full outer join

`SELECT c1, c2  
FROM t1  
CROSS JOIN t2;`  
Produce a Cartesian product of rows in tables

`SELECT c1, c2  
FROM t1, t2;`  
Another way to perform cross join

`SELECT c1, c2  
FROM t1 A  
INNER JOIN t2 B ON condition;`  
Join t1 to itself using INNER JOIN clause

### USING SQL OPERATORS

`SELECT c1, c2 FROM t1  
UNION [ALL]`  
Combine rows from two queries

`SELECT c1, c2 FROM t1  
INTERSECT`  
`SELECT c1, c2 FROM t2;`  
Return the intersection of two queries

`SELECT c1, c2 FROM t1  
MINUS`  
`SELECT c1, c2 FROM t2;`  
Subtract a result set from another result set

`SELECT c1, c2 FROM t1  
WHERE c1 [NOT] LIKE pattern;`  
Query rows using pattern matching %, \_

`SELECT c1, c2 FROM t  
WHERE c1 [NOT] IN value_list;`  
Query rows in a list

`SELECT c1, c2 FROM t  
WHERE c1 BETWEEN low AND high;`  
Query rows between two values

`SELECT c1, c2 FROM t  
WHERE c1 IS [NOT] NULL;`  
Check if values in a table is NULL or not



# Notes on Language and Data Model

[Optional]



## Marketing Billboard on 101

Aaron Levie (@levie) · Sep 24  
Silicon Valley is back!

**zesty** @zestular · Sep 24  
i watched two tech bros get kicked out of a bar last night for getting in a fight over MongoDB. you really can't beat san francisco  
[Show this thread](#)

8:49 AM · Sep 25, 2022 · Twitter for iPhone

56 Retweets 6 Quote Tweets 948 Likes

[Reply](#)

**Brian Armstrong** @brian\_armstrong · Sep 26  
Replies to @levie  
It's web scale!

10 Retweets 1 Quote Tweet 40 Likes

[Reply](#)

**Suhail** @Suhail · Sep 25  
Replies to @levie  
Meanwhile, the postgres guy was like 😊

1 Retweet 4 Likes

[Reply](#)

**Tinfoil** @tinfoil\_globe · Sep 25  
Replies to @levie  
You're going to look at me and tell me that I'm wrong? Mongo is WEBSCALE, Doug. WEB. SCALE.

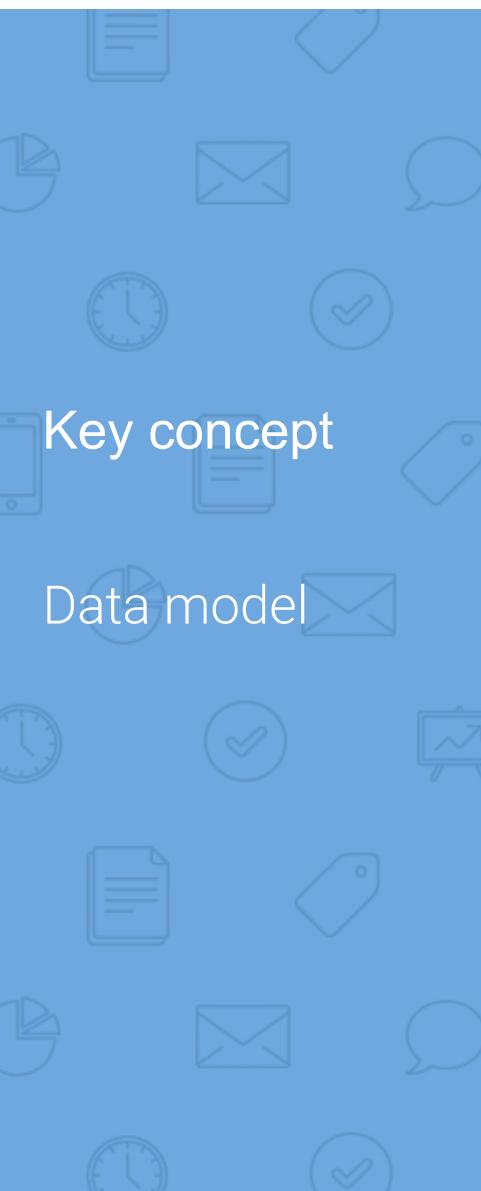
1 Retweet 6 Likes

[Reply](#)

**Aaron Levie** @levie · Sep 25  
Replies to @tinfoil\_globe  
Yeah Doug!

[Reply](#)

## Bar fights



## Data model: How to organize data + operations?

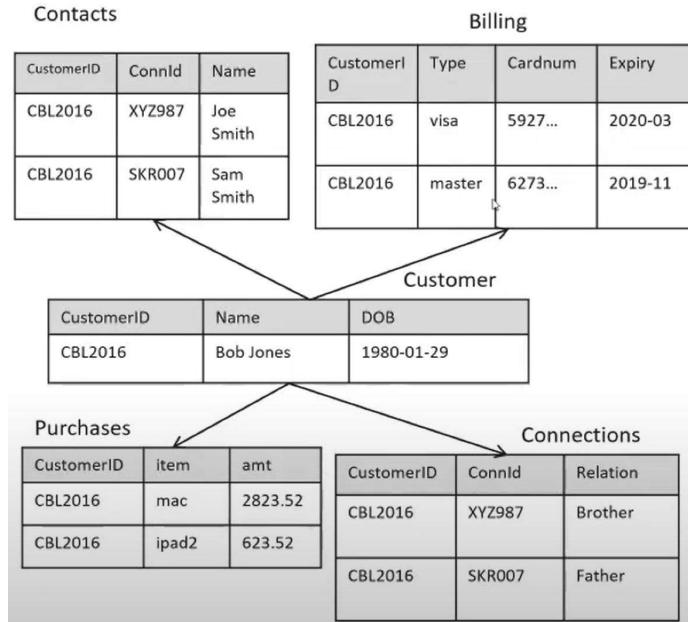
1. Relational model (aka **Tables**)  
Simple and most popular  
Elegant algebra (E.F. Codd et al)  
Structured data (e.g., a typed Schema)
  
2. Hierarchical model (aka JSON-like **Trees**)  
Semi-structured data (e.g., [Google's protobufs](#))

```
message Result {  
    string url = 1;  
    optional string title = 2;  
    optional string snippets = 3;  
}  
message SearchResponse {  
    repeated Result results = 1;  
}
```

## Example on Tradeoffs

### Data model

#### Relational



#### Hierarchical

DocumentKey: **CBL2016**

```
{
  "Name": "Bob Jones",
  "DOB": "1980-01-29",
  "Billing": [
    {
      "type": "visa",
      "cardnum": "5927-2842-2847-3909",
      "expiry": "2020-03"
    },
    {
      "type": "master",
      "cardnum": "6273-2842-2847-3909",
      "expiry": "2019-11"
    }
  ],
  "Connections": [
    {
      "CustId": "XYZ987",
      "Relation": "Brother"
    },
    {
      "CustId": "PQR823",
      "Relation": "Father"
    }
  ],
  "Purchases": [
    {"id":12, item: "mac", "amt": 2823.52},
    {"id":19, item: "ipad2", "amt": 623.52}
  ]
}
```

Error: Contacts should also be in Hierarchical's JSON.

1. Example of Table vs JSON view (Ignore the marketing)
2. Key "CS" ideas
  - a. Structured, or semi-structured with lots of optional fields?
  - b. How deep is the tree structure?
  - c. What kinds of queries and updates do you want to run? (e.g., customer-oriented queries? Purchase-oriented queries?)
3. Rough rule of thumb
  - a. **Relational**: Google Ads, Amazon Products, Bank Transactions. Structured? <10 levels deep? Few optional fields? Flexibility in running queries on all Levels of the tree?
  - b. **Hierarchical**. Document search. Lots of optional fields? Many levels deep. Mostly search oriented around the top level of doc.

# Python operating on Lists [reminder]

## BASIC TYPES

Int, long int  
string ...

## COMPOUND TYPES

Arrays, Lists, ...

## MAP + FILTER

`map(function, list of inputs)`

`filter(function, list of inputs)`

- Map applies function to input list
- Filter returns sub-list that satisfies a filter condition

## REDUCE/AGGREGATE

`reduce (...)`

- Reduce runs a computation on a list and returns a result.  
E.g., SUM, MAX, MIN

*For review, check out your favorite python tutorial (e.g, [https://book.pythontips.com/en/latest/map\\_filter.html](https://book.pythontips.com/en/latest/map_filter.html))*

# SQL Queries on Tables (Lists of rows)

## BASIC TYPES

Int32, int64  
Char[n] ...  
Float32, float64

## COMPOUND TYPES

Arrays, JSON\*, ...  
Tables

## MAP + FILTER

Single table query

```
SELECT c1, c2  
FROM T  
WHERE condition;
```

## Multi table JOIN

```
SELECT c1, c2  
FROM T1, T2  
WHERE condition;
```

## REDUCE/AGGREGATE

```
SELECT SUM(c1*c2)  
FROM T  
WHERE condition  
GROUP BY c3;
```

1. General “**Map-Filter-Reduce**” programming pattern – simple and powerful pattern used in SQL, and in nonSQL systems (e.g., MapReduce, Hadoop, key-value stores, Mongo) with different language nuances
2. \*Many modern SQL dialects [support JSON](#).



# Multi-table queries



What you will  
learn about  
in this section

1. Foreign key constraints
2. Joins: basics
3. Joins: SQL semantics



# Foreign Key constraints

- Suppose we have the following schema :

```
Students(sid: string, name: string, gpa: float)
```

```
Enrolled(student_id: string, cid: string, grade: string)
```

- And we want to impose the following constraint:  
a student must exist in the Students table to enroll in a class

**Students**

sid	name	gpa
102	Bob	3.9
123	Mary	3.8

**Enrolled**

Student _id	cid	grade
123	564	A
123	537	A+

We say that **student\_id** is a foreign key that refers to **Students**



# Declaring Foreign Keys

`Students(sid: string, name: string, gpa: float)`

`Enrolled(student_id: string, cid: string, grade: string)`

```
CREATE TABLE Enrolled (
    student_id CHAR(20),
    cid CHAR(20),
    grade CHAR(10),
    PRIMARY KEY (student_id, cid),
    FOREIGN KEY (student_id) REFERENCES Students(sid)
)
```



# Foreign Keys and update operations

`Students(sid: string, name: string, gpa: float)`

`Enrolled(student_id: string, cid: string, grade: string)`

- What if we insert a tuple into Enrolled, but no corresponding student?  
INSERT is rejected (foreign keys are constraints)!
- What if we delete a student? Design choices
  1. Disallow the delete
  2. Remove all of the courses for that student
  3. SQL allows a *third via NULL (not yet covered)*

*DBA chooses*



# Keys and Foreign Keys

## Company

<u>CName</u>	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

What is a foreign key vs. a key here?

## Product

<u>PName</u>	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi



## Concepts In this section

1. SQL introduction & schema definitions
2. Basic single-table queries
3. Multi-table queries

Preview

SQL queries

[sqltutorial.org/sql-cheat-sheet](http://www.sqltutorial.org/sql-cheat-sheet)

## SQL CHEAT SHEET <http://www.sqltutorial.org>

### QUERYING DATA FROM A TABLE

`SELECT c1, c2 FROM t;`  
Query data in columns c1, c2 from a table

`SELECT * FROM t;`  
Query all rows and columns from a table

`SELECT c1, c2 FROM t  
WHERE condition;`  
Query data and filter rows with a condition

`SELECT DISTINCT c1 FROM t  
WHERE condition;`  
Query distinct rows from a table

`SELECT c1, c2 FROM t  
ORDER BY c1 ASC [DESC];`  
Sort the result set in ascending or descending order

`SELECT c1, c2 FROM t  
ORDER BY c1  
LIMIT n OFFSET offset;`  
Skip offset of rows and return the next n rows

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1;`  
Group rows using an aggregate function

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1  
HAVING condition;`  
Filter groups using HAVING clause

### QUERYING FROM MULTIPLE TABLES

`SELECT c1, c2  
FROM t1  
INNER JOIN t2 ON condition;`  
Inner join t1 and t2

`SELECT c1, c2  
FROM t1  
LEFT JOIN t2 ON condition;`  
Left join t1 and t2

`SELECT c1, c2  
FROM t1  
RIGHT JOIN t2 ON condition;`  
Right join t1 and t2

`SELECT c1, c2  
FROM t1  
FULL OUTER JOIN t2 ON condition;`  
Perform full outer join

`SELECT c1, c2  
FROM t1  
CROSS JOIN t2;`  
Produce a Cartesian product of rows in tables

`SELECT c1, c2  
FROM t1, t2;`  
Another way to perform cross join

`SELECT c1, c2  
FROM t1 A  
INNER JOIN t2 B ON condition;`  
Join t1 to itself using INNER JOIN clause

### USING SQL OPERATORS

`SELECT c1, c2 FROM t1  
UNION [ALL]`  
Combine rows from two queries

`SELECT c1, c2 FROM t1  
INTERSECT`  
`SELECT c1, c2 FROM t2;`  
Return the intersection of two queries

`SELECT c1, c2 FROM t1  
MINUS`  
`SELECT c1, c2 FROM t2;`  
Subtract a result set from another result set

`SELECT c1, c2 FROM t1  
WHERE c1 [NOT] LIKE pattern;`  
Query rows using pattern matching %, \_

`SELECT c1, c2 FROM t  
WHERE c1 [NOT] IN value_list;`  
Query rows in a list

`SELECT c1, c2 FROM t  
WHERE c1 BETWEEN low AND high;`  
Query rows between two values

`SELECT c1, c2 FROM t  
WHERE c1 IS [NOT] NULL;`  
Check if values in a table is NULL or not



# SQL Part II



# Reminder

- ▶ **Homework 1 out**
  - ▷ Material: Lecture 1 (Systems)
  - ▷ Details: Ed Post on Oct 4. Section on 10/7
  - ▷ **Useful practice** pre Week 3
    - Weeks 3 and 4 – Focus on Scale
- ▶ **Project 1 out**
  - ▷ Material: Lectures 2-4 (SQL)
  - ▷ Details: Ed post on Oct 3
  - ▷ Good practice for SQL, and set up for Projects 2 and 3

# Preview

## SQL queries

[sqltutorial.org/sql-cheat-sheet](http://www.sqltutorial.org/sql-cheat-sheet)

## SQL CHEAT SHEET <http://www.sqltutorial.org>

### QUERYING DATA FROM A TABLE

`SELECT c1, c2 FROM t;`  
Query data in columns c1, c2 from a table

`SELECT * FROM t;`  
Query all rows and columns from a table

`SELECT c1, c2 FROM t  
WHERE condition;`  
Query data and filter rows with a condition

`SELECT DISTINCT c1 FROM t  
WHERE condition;`  
Query distinct rows from a table

`SELECT c1, c2 FROM t  
ORDER BY c1 ASC [DESC];`  
Sort the result set in ascending or descending order

`SELECT c1, c2 FROM t  
ORDER BY c1  
LIMIT n OFFSET offset;`  
Skip offset of rows and return the next n rows

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1;`  
Group rows using an aggregate function

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1  
HAVING condition;`  
Filter groups using HAVING clause

### QUERYING FROM MULTIPLE TABLES

`SELECT c1, c2  
FROM t1  
INNER JOIN t2 ON condition;`  
Inner join t1 and t2

`SELECT c1, c2  
FROM t1  
LEFT JOIN t2 ON condition;`  
Left join t1 and t2

`SELECT c1, c2  
FROM t1  
RIGHT JOIN t2 ON condition;`  
Right join t1 and t2

`SELECT c1, c2  
FROM t1  
FULL OUTER JOIN t2 ON condition;`  
Perform full outer join

`SELECT c1, c2  
FROM t1  
CROSS JOIN t2;`  
Produce a Cartesian product of rows in tables

`SELECT c1, c2  
FROM t1, t2;`  
Another way to perform cross join

`SELECT c1, c2  
FROM t1 A  
INNER JOIN t2 B ON condition;`  
Join t1 to itself using INNER JOIN clause

### USING SQL OPERATORS

`SELECT c1, c2 FROM t1  
UNION [ALL]`  
Combine rows from two queries

`SELECT c1, c2 FROM t1  
INTERSECT`  
`SELECT c1, c2 FROM t2;`  
Return the intersection of two queries

`SELECT c1, c2 FROM t1  
MINUS`  
`SELECT c1, c2 FROM t2;`  
Subtract a result set from another result set

`SELECT c1, c2 FROM t1  
WHERE c1 [NOT] LIKE pattern;`  
Query rows using pattern matching %, \_

`SELECT c1, c2 FROM t  
WHERE c1 [NOT] IN value_list;`  
Query rows in a list

`SELECT c1, c2 FROM t  
WHERE c1 BETWEEN low AND high;`  
Query rows between two values

`SELECT c1, c2 FROM t  
WHERE c1 IS [NOT] NULL;`  
Check if values in a table is NULL or not



# Key concepts

- ▶ NULLs
- ▶ JOINs
- ▶ Aggregation & GROUP BY



# NULL

- To say “don’t know the value” we use **NULL**  
NULL has (sometimes painful) semantics, more detail later

Students(sid:string, name:string, gpa: float)

sid	name	gpa
123	Bob	3.9
143	Jim	NULL

*Say, Jim just enrolled in his first class.*

In SQL, we may constrain a column to be NOT NULL,  
e.g., “name” in this table

# NULLs in SQL

- Whenever we don't have a value, we can put a NULL
- Can mean many things:
  - Value does not exist
  - Value exists but is unknown
  - Value not applicable, Etc.
- The schema specifies for each attribute if it can be null (*nullable* attribute)

# How SQL handles Null Values

- Numerical
  - If  $x = \text{NULL}$  then  $4*(3-x)/7$  is still  $\text{NULL}$
- Logical (FALSE = 0, TRUE = 1, NULL = 0.5)
  - $C1 \text{ AND } C2 = \min(C1, C2)$
  - $C1 \text{ OR } C2 = \max(C1, C2)$
  - $\text{NOT } C1 = 1 - C1$

Rule in SQL: include only tuples that yield TRUE (1.0)

# Examples

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

Test on example tables

Name	Age	Height
Alice	22	170
Bob	27	180
ET	Unknown	120

- Numerical
  - If  $x = \text{NULL}$  then  $4*(3-x)/7$  is still  $\text{NULL}$
- Logical (FALSE = 0, TRUE = 1, NULL = 0.5)
  - $C1 \text{ AND } C2 = \min(C1, C2)$
  - $C1 \text{ OR } C2 = \max(C1, C2)$
  - $\text{NOT } C1 = 1 - C1$

Rule in SQL: include only tuples that yield TRUE (1.0)

Rules from previous slide

Q1  
SELECT \*  
FROM Person  
WHERE age < 25 OR age >= 25

Ans: No ET

Q2  
SELECT \*  
FROM Person  
WHERE age < 25 OR height >= 100

Ans: Alice, Bob, ET

Q3  
SELECT \*  
FROM Person  
WHERE age < 25 AND height >= 100

Ans: Alice

# Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

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

Now it includes all Persons!



# Key concepts

- ▶ NULLs
- ▶ JOINs
- ▶ Aggregation & GROUP BY



# Set algebra (reminder)

List: [1, 1, 2, 3]

Set: {1, 2, 3}

Multiset: {1, 1, 2, 3}

A **multiset** is an unordered list (or: a set with multiple duplicate instances allowed)

## UNIONs

Set: {1, 2, 3}  $\cup$  {2} = {1, 2, 3}

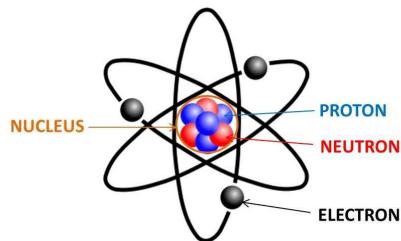
Multiset: {1, 1, 2, 3}  $\cup$  {2} = {1, 1, 2, 2, 3}

## Cross-product

$$\{1, 1, 2, 3\} * \{y, z\} =$$

$$\begin{aligned} & \{<1, y>, <1, y>, <2, y>, <3, y> \\ & <1, z>, <1, z>, <2, z>, <3, z> \\ & \} \end{aligned}$$

# Reminder on schemas



Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Students(sid: string, name: string, gpa: float)

Enrolled(student\_id: string, cid: string, grade: string)

We'll use different  
Tables/tuples, for  
examples  
to build ideas

Data about local areas (for real-world examples)

SolarPanel(region\_name: string, kw\_total: float, carbon\_offset\_ton\_metrics: float, ...)

Census(zipcode: string, population: int, ...)

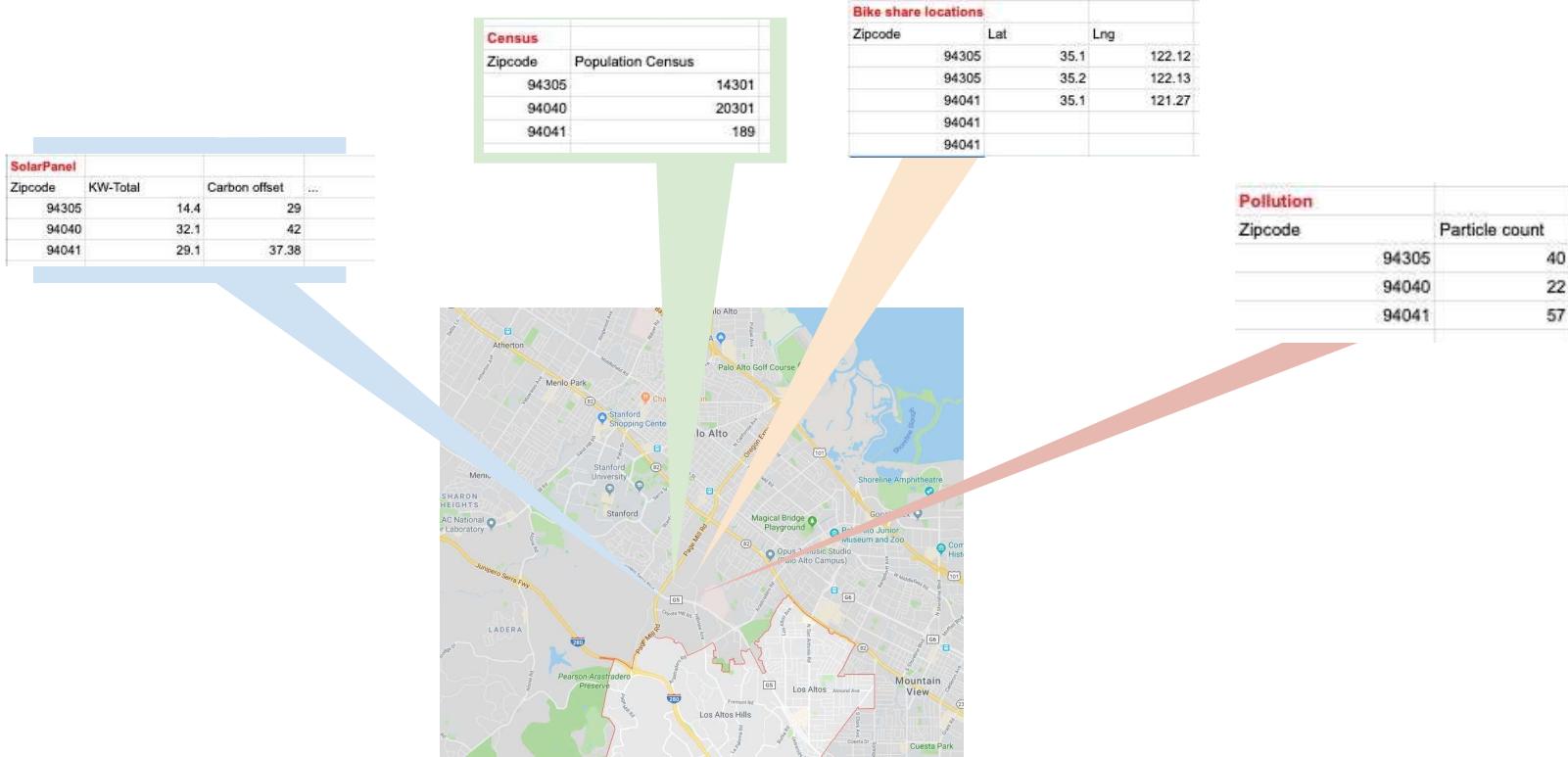
Pollution(zipcode: string, Particle\_count: int...)

BikeShare(zipcode: string, trip\_origin: float, trip\_end: float, ...)

...



## Option 1: 'Good' tables, with 10s-100s of columns

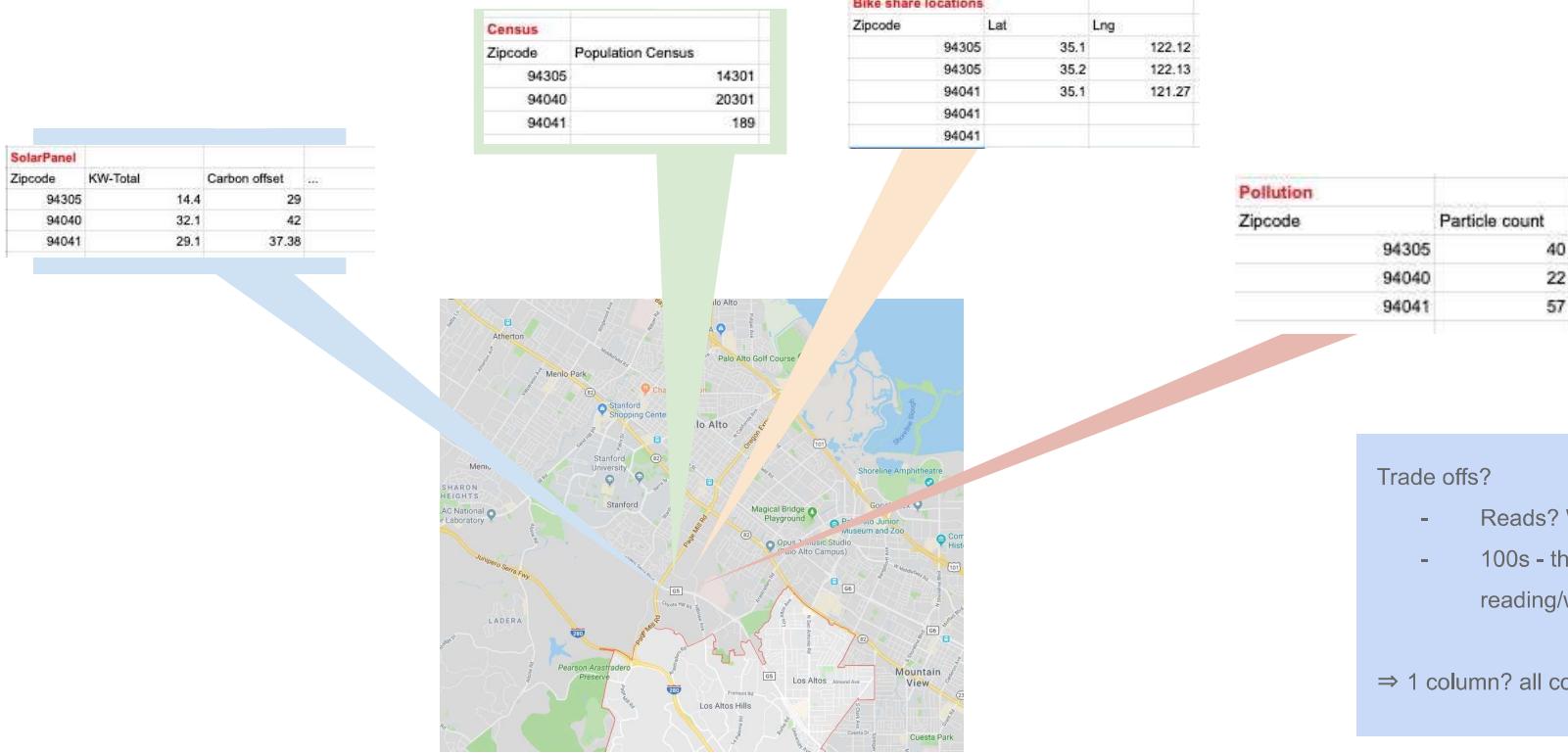


## Option 2: 'Frankenstein Table' (with 1000s of columns)

**Omnidata**

SolarPanel			Census		Pollution		Bike share locations	
Zipcode	KW-Total	Carbon offset	...	Population	Particle count	Lat	Lng	
94305	14.4	29	?????	14301	40	35.1	122.12	
94305						35.2	122.13	
94305						35.2	122.1	
94305						35.1	122.12	
94305						...	...	

## Option 1: A few “good” tables (with 10s of columns)



## Option 2: ‘FrankenTable’ (with 1000s of columns)

Omnidata			SolarPanel			Census			Pollution			Bike share locations		
Zipcode	KW-Total	Carbon offset	...			Population			Particle count	Lat	Lng			
94305	14.4	29				14301			40	35.1	122.12			
94305	14.4	29				14301			40	35.2	122.13			
94305	14.4	29				14301			40	35.2	122.1			
94305	14.4	29				14301			40	35.1	122.12			

# Split and Join

1. Split a “Franken” table into smaller tables
2. Keep extra join keys in both tables (e.g., zipcode)
3. Join tables
  - ▷ “connect” tables for queries
  - ▷ drop join keys

[Steps 1 and 2 ⇒ design “good” schema

- ▷ How? Study in Week 8, 9
- ▷ Assume (for next 2 weeks): We have pre-split tables.

# Joins

```
Product(PName, Price, Category, Manufacturer)  
Company(CName, StockPrice, Country)
```

A **join** between tables returns all unique combinations of their tuples **which meet specified join condition**

Ex: Find all products under \$200 manufactured in Japan; return their names and prices.

```
SELECT PName, Price  
FROM Product, Company  
WHERE Manufacturer = CName  
      AND Country='Japan'  
      AND Price <= 200
```

Note: we will often omit column types in schema definitions for brevity, but assume columns are always atomic types

# Joins

```
Product(PName, Price, Category, Manufacturer)  
Company(CName, StockPrice, Country)
```

Several equivalent ways to write a basic join in SQL:

```
SELECT PName, Price  
FROM Product, Company  
WHERE Manufacturer = CName  
      AND Country='Japan'  
      AND Price <= 200
```

```
.....  
SELECT PName, Price  
FROM Product  
JOIN Company  
ON Manufacturer = Cname  
WHERE Price <= 200  
      AND Country='Japan'
```

A few more later on

# Joins

**Product**

PName	Price	Category	Manufacturer
Gizmo	\$19	Gadgets	GizmoWorks
Powergizmo	\$29	Gadgets	GizmoWorks
SingleTouch	\$149	Photography	Canon
MultiTouch	\$203	Household	Hitachi

**Company**

CName	Stock Price	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer = CName
AND Country='Japan'
AND Price <= 200
```

PName	Price
SingleTouch	\$149

# Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)  
Company(name, address)

1. SELECT DISTINCT name, address
2. FROM Person, Company
3. WHERE worksfor = name

Which “address”  
does this refer to?

Which name”s??

# Tuple Variable Ambiguity in Multi-Table

Both equivalent  
ways to resolve  
variable  
ambiguity

```
Person(name, address, worksfor)  
Company(name, address)
```

```
SELECT DISTINCT Person.name, Person.address  
FROM Person, Company  
WHERE Person.worksfor = Company.name
```

```
SELECT DISTINCT p.name, p.address  
FROM Person p, Company c  
WHERE p.worksfor = c.name
```

# Semantics of JOINS

```
SELECT x1.a1, x1.a2, ..., xn.ak
FROM R1 AS x1, R2 AS x2, ..., Rn
AS xn
WHERE Conditions(x1, ..., xn)
```

```
Answer = {}
for x1 in R1 do
    for x2 in R2 do
        ....
            for xn in Rn do
                if Conditions(x1, ..., xn)
                    then Answer = Answer U {(x1.a1, x1.a2, ..., xn.ak)}
return Answer
```

**Note:**  
This is a  
*multiset*  
union

# Semantics of JOINS (2 tables)

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

- Take **cross product**

$$V = R \times S$$

Recall: Cross product ( $R \times S$ ) is the set of all unique tuples in  $R, S$

Ex:  $\{a,b,c\} \times \{1,2\}$   
 $= \{(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)\}$

- Apply **selections/conditions**

$$Y = \{(r, s) \text{ in } V \mid r.A == s.B\}$$

= **Filtering!**

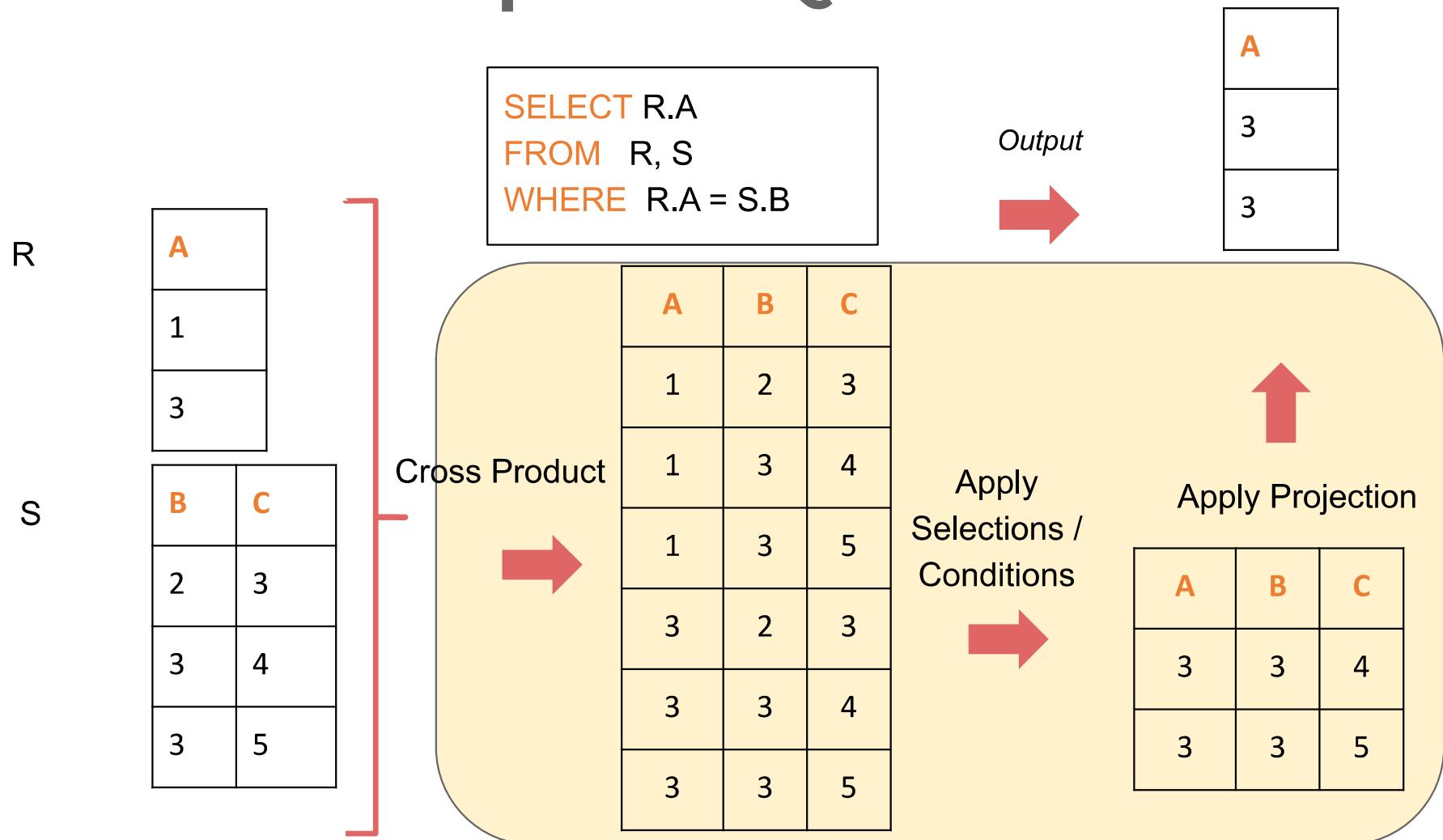
- Apply **projections** to get final output

$$Z = (y.A) \text{ for } y \text{ in } Y$$

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries  
(see later on...)

# An example of SQL semantics



## Note: we say “semantics” not “execution order”

- The preceding slides show *what a join means*
- Not how DB executes under the covers (Week 3+)

# A Subtlety about Joins

```
Product(PName, Price, Category, Manufacturer)  
Company(CName, StockPrice, Country)
```

Find all countries that manufacture some product  
in the ‘Gadgets’ category.

```
SELECT Country  
FROM Product, Company  
WHERE Manufacturer=CName AND  
Category='Gadgets'
```

# A Subtlety about Joins

**Product**

PName	Price	Category	Manuf
Gizmo	\$19	Gadgets	GWorks
Powergizmo	\$29	Gadgets	GWorks
SingleTouch	\$149	Photography	Canon
MultiTouch	\$203	Household	Hitachi

**Company**

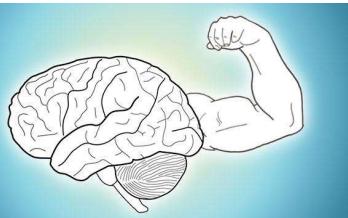
Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT Country
FROM Product, Company
WHERE Manufacturer=Cname
AND Category='Gadgets'
```

Country
USA
USA

What is the Problem? What is the Solution?

Ans? USA is duplicated. Use DISTINCT



# So far, we've seen **Inner Joins**

By default, joins in SQL are “**inner joins**”:

```
Product(name, category)  
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store  
FROM Product, Purchase  
WHERE Product.name = Purchase.prodName
```

```
.....  
SELECT Product.name, Purchase.store  
FROM Product  
JOIN Purchase ON Product.name = Purchase.prodName
```

```
.....  
SELECT Product.name, Purchase.store  
FROM Product  
INNER JOIN Purchase  
ON Product.name = Purchase.prodName
```

Both equivalent:  
Both INNER JOINS!  
[Like Below]

# Inner Joins

By default, joins in SQL are “**inner joins**”:

```
Product(name, category)  
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store  
FROM Product  
JOIN Purchase ON Product.name = Purchase.prodName
```

Product	
Name	Category
Iphone	Media
Roomba	Cleaner
Ford Pinto	Car
Tesla	Car

Purchase	
ProdName	Store
Iphone	Apple Store
Tesla	Tesla Store

What about Products that never sold (with no Purchase tuple)?

# LEFT OUTER JOIN

Product

name	category
iphone	media
Tesla	car
Ford Pinto	car

Purchase

prodName	store
iPhone	Apple store
Tesla	car
iPhone	Apple store

```
SELECT Product.name, Purchase.store  
FROM Product  
LEFT OUTER JOIN Purchase  
ON Product.name = Purchase.prodName
```



name	store
iPhone	Apple store
iPhone	Apple store
Tesla	car
Ford Pinto	NULL

# Outer Joins

- An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations
  - I.e. If we join relations A and B on  $a.X = b.X$ , and there is an entry in A with  $X=5$ , but none in B with  $X=5\dots$
  - A LEFT OUTER JOIN will return a tuple (a, NULL)!
- Left outer joins in SQL:

```
SELECT Product.name, Purchase.store  
FROM Product  
LEFT OUTER JOIN Purchase ON  
Product.name = Purchase.prodName
```

Now we'll get products even if they didn't sell

# Other Outer Joins

- Left outer join:
  - Include the left tuple even if there's no match
- Right outer join:
  - Include the right tuple even if there's no match
- Full outer join:
  - Include the both left and right tuples even if there's no match

# Key concepts

- ▶ JOINs
- ▶ Aggregation & GROUP BY

# Aggregation

```
SELECT AVG(price)  
FROM Product  
WHERE maker = "Toyota"
```

```
SELECT COUNT(*)  
FROM Product  
WHERE year > 1995
```

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

# Aggregation: COUNT

COUNT counts all tuples (including duplicates), unless otherwise stated

```
SELECT COUNT(category)
FROM Product
WHERE year > 1995
```

versus

```
SELECT COUNT(DISTINCT category)
FROM Product
WHERE year > 1995
```

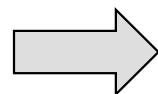
# Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

Example 1

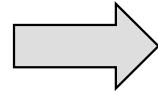
```
SELECT SUM(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```



50  
 $(= 1*20 + 1.50*20)$

Example 2

```
SELECT SUM(price * quantity)
FROM Purchase
```



65  
 $(= 1*20 + 1.50*20 + 0.5*10 + 1*10)$

# Grouping and Aggregation



What GROUPings are possible?

- Type, Size, Color
- Number of holes
- Combination?

# What GROUPings are possible?

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

## Possible Groups

- Product? (e.g. SUM(quantity) by product) # product units sold
- Date? (e.g., SUM(price\*quantity) by date) # daily sales
- Price?
- Product, Date?
- <various column combinations>

# Grouping and Aggregation

Purchase(product, date, price, quantity)

```
SELECT product,  
       SUM(price * quantity) AS TotalSales  
FROM Purchase  
WHERE date > '10/1/2005'  
GROUP BY product
```

Find total sales  
after 10/1/2005  
per product.

Let's see what this means...

# Grouping and Aggregation

```
SELECT product,  
       SUM(price * quantity) AS TotalSales  
FROM Purchase  
WHERE date > '10/1/2005'  
GROUP BY product
```

## Semantics of the query:

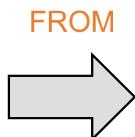
1. Compute the **FROM** and **WHERE** clauses
2. Group by the attributes in the **GROUP BY**
3. Compute the **SELECT** clause: grouped attributes and aggregates

(Why is Select on top? Similar to Functions --  $f \Rightarrow$  output on top)

# Step 1.

## Compute the **FROM** and **WHERE** clauses

```
SELECT product, SUM(price*quantity) AS TotalSales  
FROM Purchase  
WHERE date > '10/1/2005'  
GROUP BY product
```

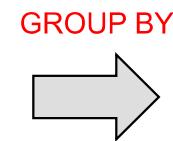


Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

## Step 2. Group by the attributes in the **GROUP BY**

```
SELECT product, SUM(price*quantity) AS TotalSales  
FROM Purchase  
WHERE date > '10/1/2005'  
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10



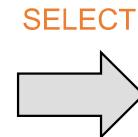
Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

## Step 3.

# Compute the **SELECT** clause: grouped attributes and aggregates

```
SELECT product, SUM(price*quantity) AS TotalSales  
FROM Purchase  
WHERE date > '10/1/2005'  
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10



Product	TotalSales
Bagel	50
Banana	15

(Why is Select on top? Similar to Functions ⇒ output on top)

# HAVING Clause

```
SELECT product, SUM(price*quantity)
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers

HAVING clauses contains conditions on **aggregates**

Whereas WHERE clauses condition on *individual tuples*...

# General form of Grouping and Aggregation

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

Why?

- S = Can ONLY contain attributes  $a_1, \dots, a_k$  and/or aggregates over other attributes
- $C_1$  = is any condition on the attributes in  $R_1, \dots, R_n$
- $C_2$  = is any condition on the aggregate expressions

# 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  $R_1, \dots, R_n$
2. **GROUP BY** the attributes  $a_1, \dots, a_k$
3. **Apply condition  $C_2$  to each group (may need to compute aggregates)**
4. Compute aggregates in  $S$  and return the result

# Common Table Expressions (CTEs)

Give it a name

```
WITH ProductSales AS
  (SELECT product,
           SUM(price * quantity) AS TotalSales
    FROM Purchase
   WHERE date > '10/1/2005'
  GROUP BY product)

SELECT * FROM ProductSales
```

Useful for readability -- e.g., sub-queries, chaining queries

# Key concepts

- ▶ JOINs
- ▶ Aggregation & GROUP BY

# Preview

## SQL queries

[sqltutorial.org/sql-cheat-sheet](http://sqltutorial.org/sql-cheat-sheet)

## SQL CHEAT SHEET <http://www.sqltutorial.org>

### QUERYING DATA FROM A TABLE

`SELECT c1, c2 FROM t;`  
Query data in columns c1, c2 from a table

`SELECT * FROM t;`  
Query all rows and columns from a table

`SELECT c1, c2 FROM t  
WHERE condition;`  
Query data and filter rows with a condition

`SELECT DISTINCT c1 FROM t  
WHERE condition;`  
Query distinct rows from a table

`SELECT c1, c2 FROM t  
ORDER BY c1 ASC [DESC];`  
Sort the result set in ascending or descending order

`SELECT c1, c2 FROM t  
ORDER BY c1  
LIMIT n OFFSET offset;`  
Skip offset of rows and return the next n rows

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1;`  
Group rows using an aggregate function

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1  
HAVING condition;`  
Filter groups using HAVING clause

### QUERYING FROM MULTIPLE TABLES

`SELECT c1, c2  
FROM t1  
INNER JOIN t2 ON condition;`  
Inner join t1 and t2

`SELECT c1, c2  
FROM t1  
LEFT JOIN t2 ON condition;`  
Left join t1 and t2

`SELECT c1, c2  
FROM t1  
RIGHT JOIN t2 ON condition;`  
Right join t1 and t2

`SELECT c1, c2  
FROM t1  
FULL OUTER JOIN t2 ON condition;`  
Perform full outer join

`SELECT c1, c2  
FROM t1  
CROSS JOIN t2;`  
Produce a Cartesian product of rows in tables

`SELECT c1, c2  
FROM t1, t2;`  
Another way to perform cross join

`SELECT c1, c2  
FROM t1 A  
INNER JOIN t2 B ON condition;`  
Join t1 to itself using INNER JOIN clause

### USING SQL OPERATORS

`SELECT c1, c2 FROM t1  
UNION [ALL]`  
`SELECT c1, c2 FROM t2;`  
Combine rows from two queries

`SELECT c1, c2 FROM t1  
INTERSECT`  
`SELECT c1, c2 FROM t2;`  
Return the intersection of two queries

`SELECT c1, c2 FROM t1  
MINUS`  
`SELECT c1, c2 FROM t2;`  
Subtract a result set from another result set

`SELECT c1, c2 FROM t1  
WHERE c1 [NOT] LIKE pattern;`  
Query rows using pattern matching %, \_

`SELECT c1, c2 FROM t  
WHERE c1 [NOT] IN value_list;`  
Query rows in a list

`SELECT c1, c2 FROM t  
WHERE c1 BETWEEN low AND high;`  
Query rows between two values

`SELECT c1, c2 FROM t  
WHERE c1 IS [NOT] NULL;`  
Check if values in a table is NULL or not



Wrap up SQL for cs145 (Practice with 3 Projects)

- ▶ SET operators, Sub-queries
- ▶ When are two queries equivalent?
- ▶ How does SQL work?
  - ▷ Intro to Relational Algebra
  - ▷ A basic RDBMS query optimizer

Next 2 weeks – Scale, Scale, Scale

# Preview

## SQL queries

[sqltutorial.org/sql-cheat-sheet](http://sqltutorial.org/sql-cheat-sheet)

## SQL CHEAT SHEET <http://www.sqltutorial.org>

### QUERYING DATA FROM A TABLE

`SELECT c1, c2 FROM t;`  
Query data in columns c1, c2 from a table

`SELECT * FROM t;`  
Query all rows and columns from a table

`SELECT c1, c2 FROM t  
WHERE condition;`  
Query data and filter rows with a condition

`SELECT DISTINCT c1 FROM t  
WHERE condition;`  
Query distinct rows from a table

`SELECT c1, c2 FROM t  
ORDER BY c1 ASC [DESC];`  
Sort the result set in ascending or descending order

`SELECT c1, c2 FROM t  
ORDER BY c1  
LIMIT n OFFSET offset;`  
Skip offset of rows and return the next n rows

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1;`  
Group rows using an aggregate function

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1  
HAVING condition;`  
Filter groups using HAVING clause

### QUERYING FROM MULTIPLE TABLES

`SELECT c1, c2  
FROM t1  
INNER JOIN t2 ON condition;`  
Inner join t1 and t2

`SELECT c1, c2  
FROM t1  
LEFT JOIN t2 ON condition;`  
Left join t1 and t2

`SELECT c1, c2  
FROM t1  
RIGHT JOIN t2 ON condition;`  
Right join t1 and t2

`SELECT c1, c2  
FROM t1  
FULL OUTER JOIN t2 ON condition;`  
Perform full outer join

`SELECT c1, c2  
FROM t1  
CROSS JOIN t2;`  
Produce a Cartesian product of rows in tables

`SELECT c1, c2  
FROM t1, t2;`  
Another way to perform cross join

`SELECT c1, c2  
FROM t1 A  
INNER JOIN t2 B ON condition;`  
Join t1 to itself using INNER JOIN clause

### USING SQL OPERATORS

`SELECT c1, c2 FROM t1  
UNION [ALL]`  
Combine rows from two queries

`SELECT c1, c2 FROM t1  
INTERSECT`  
`SELECT c1, c2 FROM t2;`  
Return the intersection of two queries

`SELECT c1, c2 FROM t1  
MINUS`  
`SELECT c1, c2 FROM t2;`  
Subtract a result set from another result set

`SELECT c1, c2 FROM t1  
WHERE c1 [NOT] LIKE pattern;`  
Query rows using pattern matching %, \_

`SELECT c1, c2 FROM t  
WHERE c1 [NOT] IN value_list;`  
Query rows in a list

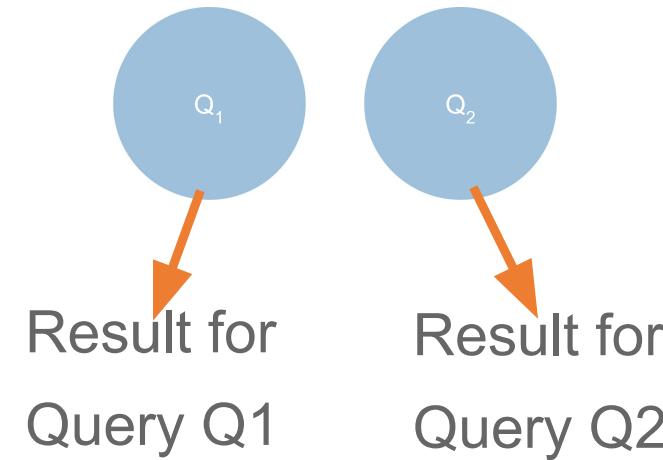
`SELECT c1, c2 FROM t  
WHERE c1 BETWEEN low AND high;`  
Query rows between two values

`SELECT c1, c2 FROM t  
WHERE c1 IS [NOT] NULL;`  
Check if values in a table is NULL or not

What you will  
learn about in  
this section

1. Set and Multiset operators in SQL
2. Nested queries

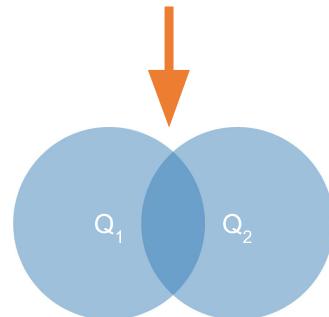
Notation



# Explicit Set Operators: INTERSECT, UNIONs on results of Queries Q1, Q2

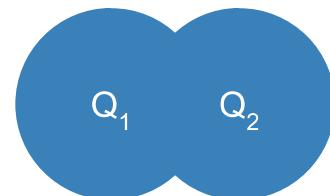
```
SELECT R.A  
FROM R, S  
WHERE R.A=S.A  
INTERSECT  
SELECT R.A  
FROM R, T  
WHERE R.A=T.A
```

Q1      Q2

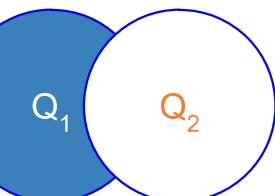


```
SELECT R.A  
FROM R, S  
WHERE R.A=S.A  
UNION  
SELECT R.A  
FROM R, T  
WHERE R.A=T.A
```

Q1      Q2



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



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

1. For Set Operators (UNION, INTERSECT, EXCEPT\* or MINUS\*), SQL uses Set semantics (i.e., no dups)
2. To keep Multisets, use **UNION ALL**, **INTERSECT ALL\***  
(\* footnote – differences in SQL dialects for specific DBs. Check DB's docs)

# Key concept

Like other programming languages . . .

- ▶ SQL is **Compositional**
  - a. Output of one query can be input to another (nesting)!
    - ⇒ Q1 -> Q2 -> ... Qn -> Result Table
    - ⇒ Including on same table (e.g., self correlation)
  - b. How?
    - i. Common Table Expressions (CTEs)
    - ii. **Sub-queries** offer another tool
    - iii. (Trade offs: [Read blog](#) on CTE vs sub-queries)
- ▶ But, **Declarative**. (Specify what you want)

# Nested queries (aka Sub-queries) Return Relations (or Columns)

Company(name, city)  
Product(name, maker)  
Purchase(id, product, buyer)

Company		Product		Purchase	
Name	City	Name	Maker	Product	Buyer
Tesla	Palo Alto	Model X	Tesla	Kindle	Mickey
Amazon	Seattle	Kindle	Amazon	Model X	Mickey
		Kindle Fire	Amazon	Kindle Fire	Mickey
		Books	Amazon	Book	Mickey

Q: Mickey buys products. Where (cities) are those made?

```
SELECT pr.maker  
FROM Purchase p, Product pr  
WHERE p.product = pr.name  
AND p.buyer = 'Mickey')
```

Outer SQL =  
- City of companies returned by inner SQL?  
Inner SQL =  
- Companies making products Mickey bought"

(Ans = Palo Alto, Seattle)

# Subqueries Return Relations

You can also use operations of the form:

- $s > \text{ALL } R$
- $s < \text{ANY } R$
- $\text{EXISTS } R$

Ex: Product(name, price, maker)

Maker	Name	Price
Apple	Ipad	1000
Apple	Iwatch	500
Grapes	Ipad	700
Grapes	Civic	20000
Tesla	Model3	50000

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

Q1: Find products that are more expensive than all those produced by "Apple"

(Ans = Civic, Model3)

```
SELECT p1.name  
FROM Product p1  
WHERE p1.maker = 'Apple'  
AND EXISTS(  
    SELECT p2.name  
    FROM Product p2  
    WHERE p2.maker <> 'Apple'  
    AND p1.name = p2.name)
```

Q2: Find 'copycat' products of Apple, i.e. products made by competitors with the same names as Apple's products (e.g., Ipad by Grapes)

(Ans = IPad)

$\neq$  means !=

Imagine "double FOR loop" (Scoped variables!)

# Example: Complex Correlated Query

Q: Makers focusing on more expensive products?

Find makers and products that are more expensive than all products made by the same maker before 2020

Product(name, price, maker, year)

```
SELECT DISTINCT x.name, x.maker
FROM Product AS x
WHERE x.price > ALL(
    SELECT y.price
    FROM Product AS y
    WHERE x.maker = y.maker
        AND y.year < 2020)
```

Maker	Name	Price	Year
Apple	Ipad	1000	2019
Apple	Iwatch	500	2019
Apple	Ipad	1200	2022
Apple	Iwatch	1100	2022
Tesla	Model3	50000	2019
Tesla	ModelX	100000	2019
Tesla	Model3	60000	2022
Tesla	ModelX	100000	2022

(Ans = <Apple, Ipad>  
<Apple, Iwatch>  
)

Can be very powerful (also much harder to optimize)

# Key concept

## Equivalent SQL queries

Can write different SQL queries to solve same problem

Key:

- Be careful with sets and multisets
- Go back to semantics (1st principles)

Errors fixed – 10/6/22 Lecture

## Example1: Two equivalent queries?

```
Product(pname, price, maker)  
Company(name, city)
```

Company	
Name	city
Amzn	Seattle
Msft	Seattle
Apple	Cupertino

Product		
pname	maker	price
Windows	Msft	55
Xbox	Msft	300
Echo	Amzn	55
Kindle	Amzn	75

Find all companies with  
products having price < 100

vs

Find all companies that don't  
make products with price >= 100

'Similar' but  
non-equivalent'

```
SELECT DISTINCT Company.name  
FROM Company, Product  
WHERE Company.name = Product.maker  
AND Product.price < 100
```

(Ans = Amazon, Msft  
)

```
SELECT DISTINCT Company.name  
FROM Company  
WHERE Company.name NOT IN (  
    SELECT Product.maker  
    FROM Product  
    WHERE Product.price >= 100)
```

(Ans = Amazon, Apple  
)

## Errors fixed – 10/6/22 Lecture

### Example 2: Headquarters of makers with factories in US AND China

Company(name, city)  
Product(pname, maker, factory\_loc)

Company	
Name	city
Amazon	Seattle
Microsoft	Seattle
Apple	Cupertino

Product		
pname	maker	factory_loc
Xbox	Microsoft	US
Echo	Amazon	China
ipad	Apple	US
Mac	Apple	China

Option 1: With Nested queries

```
SELECT DISTINCT city
FROM Company, Product
WHERE maker = name
AND maker IN (
    SELECT maker
    FROM Product
    WHERE factory_loc = 'US')
AND maker IN (
    SELECT maker
    FROM Product
    WHERE factory_loc = 'China')
```

(Ans = Cupertino  
)

Option 2: With Intersections

```
SELECT city
FROM Company, Product
WHERE maker = name
AND factory_loc='US'
INTERSECT
SELECT city
FROM Company, Product
WHERE maker = name
AND factory_loc='China'
```

(Ans = Cupertino, Seattle  
)

Microsoft has a factory in the US (but not China)

Amazon has a factory in China (but not US)

But Seattle is returned by the query!

⇒ Option 1 and Option 2 are **NOT** equivalent

## Example3: Are these equivalent? [harder version]

```
SELECT c.city  
FROM Company c, Product pr, Purchase p  
WHERE c.name = pr.maker  
AND pr.name = p.product  
AND p.buyer = 'Mickey'
```

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

Step 1:  
Construct some  
examples

Company		Product		Purchase	
Name	City	Name	Maker	Product	Buyer
Tesla	Palo Alto	Model X	Tesla	Kindle	Mickey
Amazon	Seattle	Kindle	Amazon	Model X	Mickey
		Kindle Fire	Amazon	Kindle Fire	Mickey
		Books	Amazon	Book	Mickey

Step 2: Test  
examples

Seattle  
Palo Alto  
Seattle  
Seattle

Palo Alto  
Seattle

Beware of duplicates!

## Example3: Are these equivalent?

```
SELECT c.city  
FROM Company c, Product pr, Purchase p  
WHERE c.name = pr.maker  
AND pr.name = p.product  
AND p.buyer = 'Mickey'
```

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

Fix duplicates!

```
SELECT DISTINCT c.city  
FROM Company c, Product pr, Purchase p  
WHERE c.name = pr.maker  
AND pr.name = p.product  
AND p.buyer = 'Mickey'
```

```
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 = 'Mickey')
```

Now they are equivalent (both use set semantics)

## Example4: Are these equivalent?

Students(sid, name, gpa)  
Enrolled(student\_id, cid, grade)

- Find students enrolled in > 5 classes

### Attempt 1: with nested queries

```
SELECT DISTINCT Students.sid  
FROM Students  
WHERE (  
    SELECT COUNT(cid)  
    FROM Enrolled  
    WHERE Students.sid = Enrolled.student_id  
) > 5
```

SQL by  
a novice

### Attempt 2: with GROUP BYs

```
SELECT Students.sid  
FROM Students, Enrolled  
WHERE Students.sid = Enrolled.student_id  
GROUP BY Students.sid  
HAVING COUNT(Enrolled.cid) > 5
```

1. SQL by an expert
2. No need for **DISTINCT**: automatic from **GROUP BY**

# Group-by vs. Nested Query

Which way is more efficient?

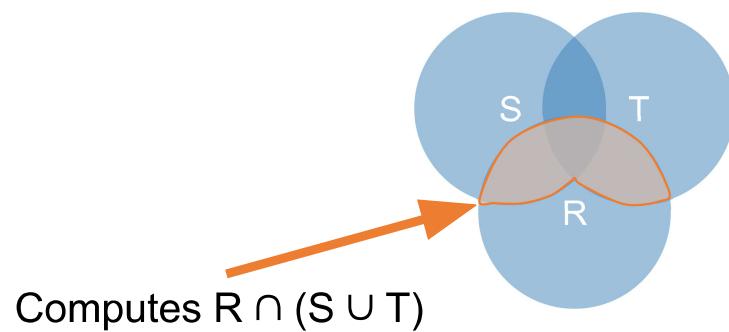
- Attempt #1- *With nested*: How many times do we do a SFW query over all of the Enrolled relations?
- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be **much** more efficient!

## Example 5: An Unintuitive Query

```
SELECT DISTINCT R.A  
FROM R, S, T  
WHERE R.A=S.A OR R.A=T.A
```

What does it compute?



But what if  $S = \emptyset$ ?

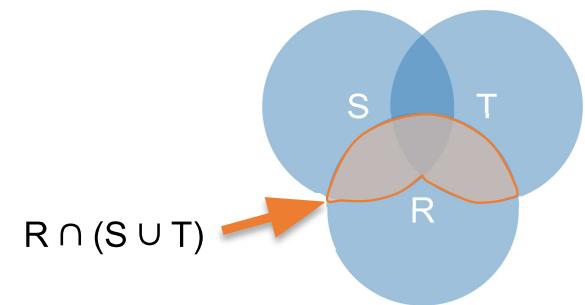
Go back to the semantics!

# What does this look like in Python?

```
SELECT DISTINCT R.A  
FROM R, S, T  
WHERE R.A=S.A OR R.A=T.A
```

Semantics:

1. Take cross-product
2. Apply selections / conditions
3. Apply projection

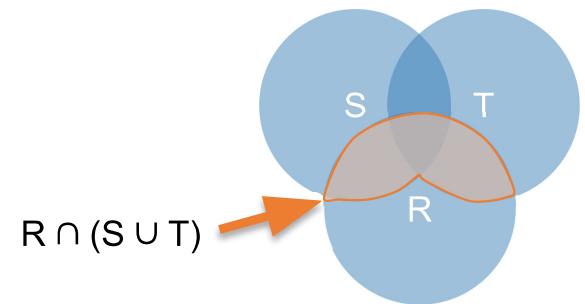


*Joins / cross-products are just nested for loops  
(in simplest implementation)!*

*If-then statements!*

# What does this look like in Python?

```
SELECT DISTINCT R.A  
FROM R, S, T  
WHERE R.A=S.A OR R.A=T.A
```



```
output = []  
  
for r in R:  
    for s in S:  
        for t in T:  
            if r['A'] == s['A'] or r['A'] == t['A']:  
                output.append(r['A'])  
return list(output)
```

Can you see now what happens if  $S = []$ ?

# Key concept

## Equivalent SQL queries

Can write different SQL queries to solve same problem

Key:

- Be careful with sets and multisets
- Go back to semantics (1st principles)

Preview

SQL queries

[sqltutorial.org/sql-cheat-sheet](http://sqltutorial.org/sql-cheat-sheet)

## SQL CHEAT SHEET <http://www.sqltutorial.org>

### QUERYING DATA FROM A TABLE

`SELECT c1, c2 FROM t;`  
Query data in columns c1, c2 from a table

`SELECT * FROM t;`  
Query all rows and columns from a table

`SELECT c1, c2 FROM t  
WHERE condition;`  
Query data and filter rows with a condition

`SELECT DISTINCT c1 FROM t  
WHERE condition;`  
Query distinct rows from a table

`SELECT c1, c2 FROM t  
ORDER BY c1 ASC [DESC];`  
Sort the result set in ascending or descending order

`SELECT c1, c2 FROM t  
ORDER BY c1  
LIMIT n OFFSET offset;`  
Skip offset of rows and return the next n rows

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1;`  
Group rows using an aggregate function

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1  
HAVING condition;`  
Filter groups using HAVING clause

### QUERYING FROM MULTIPLE TABLES

`SELECT c1, c2  
FROM t1  
INNER JOIN t2 ON condition;`  
Inner join t1 and t2

`SELECT c1, c2  
FROM t1  
LEFT JOIN t2 ON condition;`  
Left join t1 and t2

`SELECT c1, c2  
FROM t1  
RIGHT JOIN t2 ON condition;`  
Right join t1 and t2

`SELECT c1, c2  
FROM t1  
FULL OUTER JOIN t2 ON condition;`  
Perform full outer join

`SELECT c1, c2  
FROM t1  
CROSS JOIN t2;`  
Produce a Cartesian product of rows in tables

`SELECT c1, c2  
FROM t1, t2;`  
Another way to perform cross join

`SELECT c1, c2  
FROM t1 A  
INNER JOIN t2 B ON condition;`  
Join t1 to itself using INNER JOIN clause

### USING SQL OPERATORS

`SELECT c1, c2 FROM t1  
UNION [ALL]`  
Combine rows from two queries

`SELECT c1, c2 FROM t1  
INTERSECT`  
`SELECT c1, c2 FROM t2;`  
Return the intersection of two queries

`SELECT c1, c2 FROM t1  
MINUS`  
`SELECT c1, c2 FROM t2;`  
Subtract a result set from another result set

`SELECT c1, c2 FROM t1  
WHERE c1 [NOT] LIKE pattern;`  
Query rows using pattern matching %, \_

`SELECT c1, c2 FROM t  
WHERE c1 [NOT] IN value_list;`  
Query rows in a list

`SELECT c1, c2 FROM t  
WHERE c1 BETWEEN low AND high;`  
Query rows between two values

`SELECT c1, c2 FROM t  
WHERE c1 IS [NOT] NULL;`  
Check if values in a table is NULL or not

## [Optional]

### Example non-SQL syntax for hierarchical data models

```
SELECT * FROM inventory WHERE status = "A" OR qty < 30
```

versus

```
db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

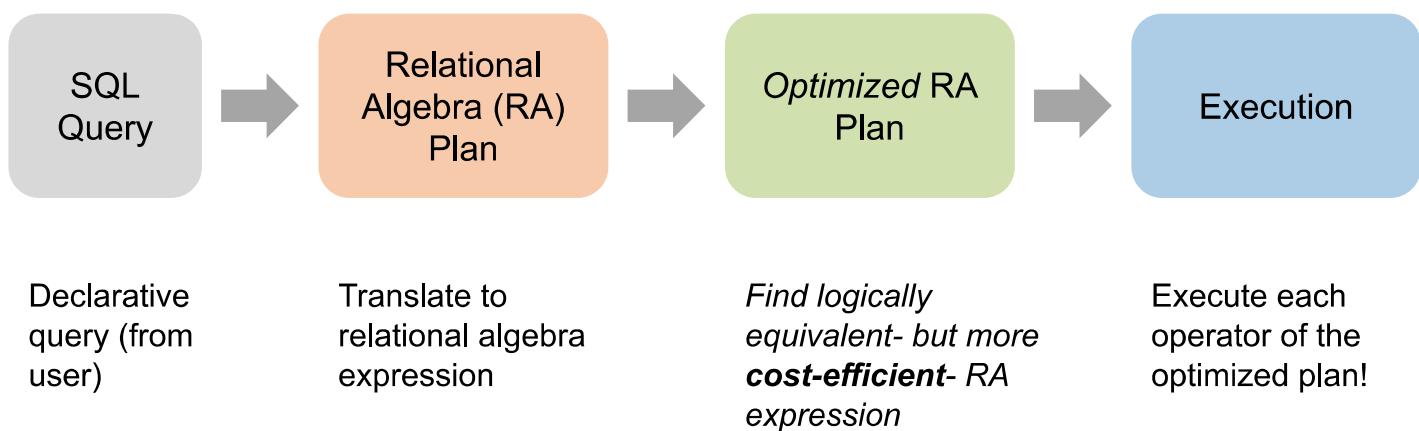


# How does it work?



# RDBMS Architecture

How does a SQL engine work ?



# RDBMS Architecture

How does a SQL engine work ?



Relational Algebra allows us to translate declarative (SQL) queries into precise and optimizable expressions!



# Relational Algebra (RA)

## Five basic operators:

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

## Derived or auxiliary operators:

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

## What's an Algebra? Why?

- For ex, in Math
    - a)  $(x + y) + z$  vs  $x + y + z$
    - b)  $(x + y) + 2*x$  vs  $(x + y + 2)*x$
  - Operators and rules
    - Basic notation for operators ('+', ' $\cdot$ ', '\*', ' $\wedge$ ', ' $\wedge\wedge$ ' etc.)
    - Association, commutative, ...
- ⇒ Why?
- What can you reorder, simplify?
  - Express complex equations and expressions, and reason about them

# Converting SFW Query to RA

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

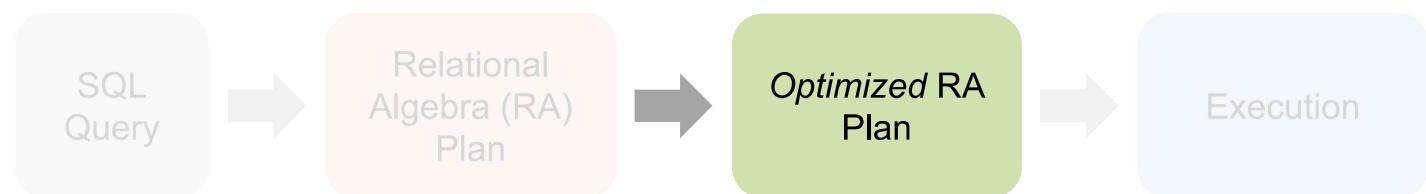
```
SELECT DISTINCT gpa, address  
FROM Students S, People P  
WHERE gpa > 3.5 AND  
      sname = pname;
```


$$\Pi_{gpa,address}(\sigma_{gpa>3.5}(S \bowtie P))$$

How do we represent this query in RA?

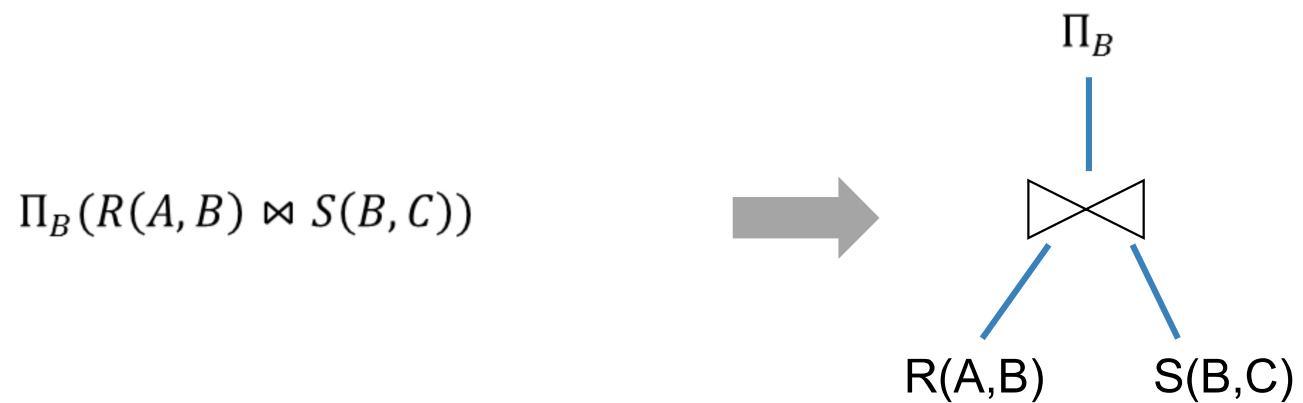
# RDBMS Architecture

How does a SQL engine work ?



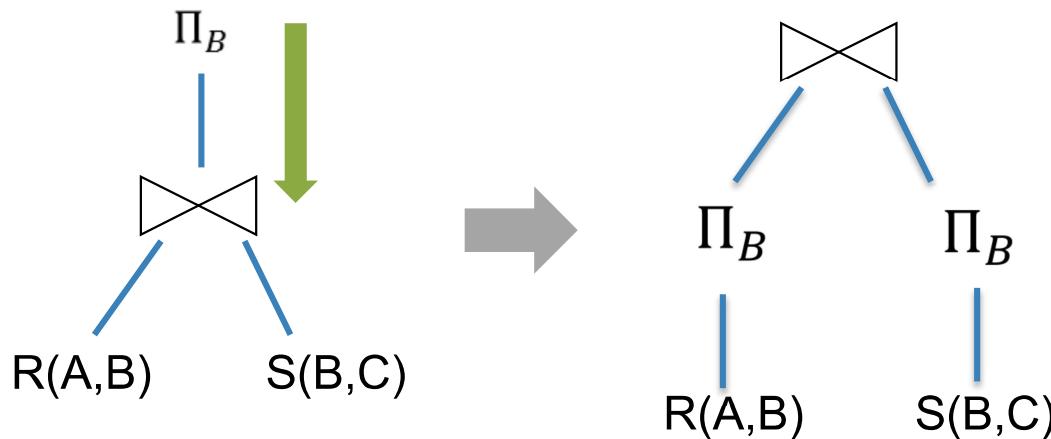
We'll look at how to then optimize these plans now

# Visualize the plan as a tree



Bottom-up tree traversal = order of operation execution!

## One simple plan -- “Push down” projection



What SQL query does this correspond to?

Are there any logically equivalent RA expressions?

Notes:

1. For the LHS tree, note that after JOIN, you are retaining only ‘B’
2. For the RHS tree, you are retaining ‘B’ (and dropping A and C early)

Why might we prefer this plan?



# Optimizing the SFW RA Plan



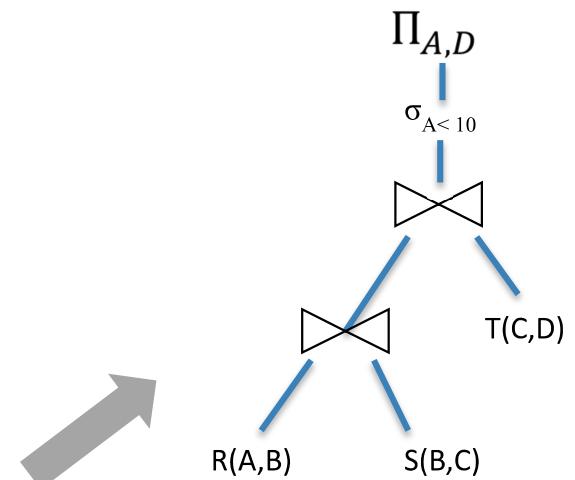
# Logical Optimization

- Heuristically, we want selections and projections to occur as early as possible in the plan
  - Terminology: “push down **selections and projections**”
- **Intuition:** We will usually have fewer tuples in a plan.  
Exceptions
  - Could fail if the selection condition is very expensive (e.g., run image processing algorithm)
  - Projection could be a waste of effort, but more rarely

⇒ Cost-based Query Optimizers (e.g., Postgres/ BigQuery/ MySQL optimizers, SparkSQL’s Catalyst)

# Example1: Translating to RA

R(A,B) S(B,C) T(C,D)
<b>SELECT R.A,T.D FROM R,S,T WHERE R.B = S.B AND S.C = T.C AND R.A &lt; 10;</b>


$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$


# Example1: Optimizing RA Plan

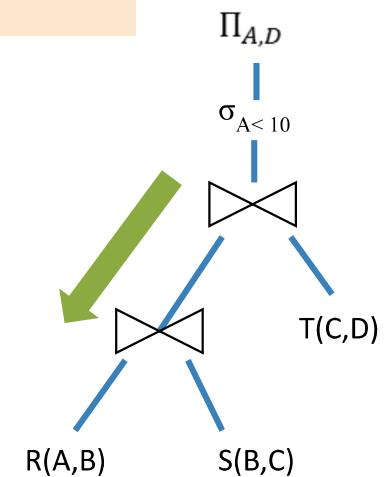
R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D  
FROM R,S,T  
WHERE R.B = S.B  
AND S.C = T.C  
AND R.A < 10;
```



$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$

Push down selection  
on A so it occurs  
earlier



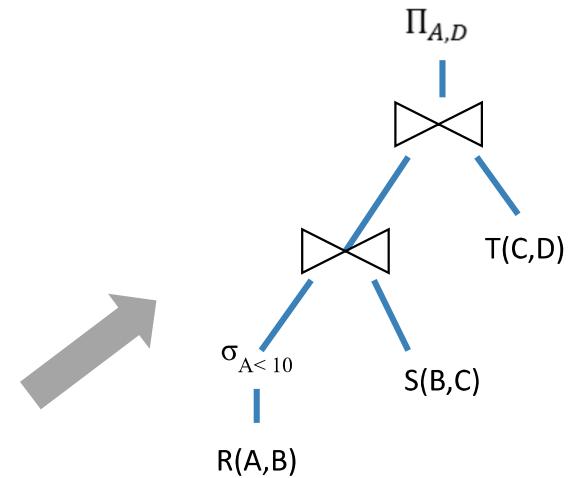
# Example1: Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D  
FROM R,S,T  
WHERE R.B = S.B  
AND S.C = T.C  
AND R.A < 10;
```

Push down selection  
on A so it occurs  
earlier

$\Pi_{A,D}(T \bowtie (\sigma_{A<10}(R) \bowtie S))$



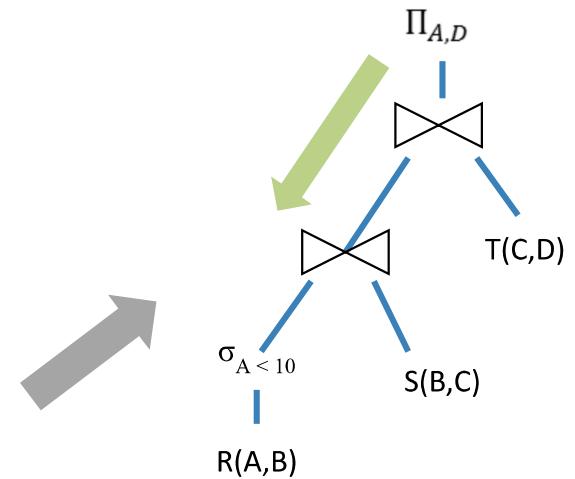
# Example1: Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D  
FROM R,S,T  
WHERE R.B = S.B  
AND S.C = T.C  
AND R.A < 10;
```

Push down  
projection so it  
occurs earlier

$$\Pi_{A,D}(T \bowtie (\sigma_{A<10}(R) \bowtie S))$$



# Example1: Optimizing RA Plan

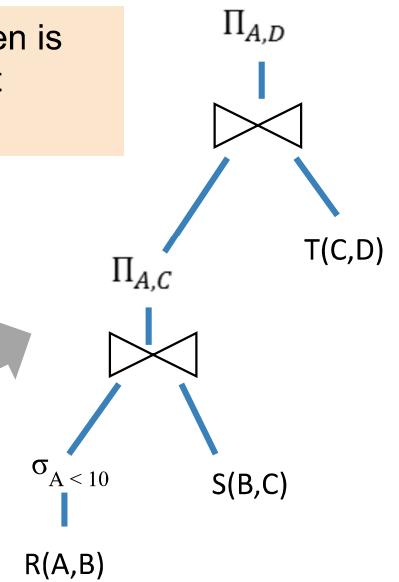
R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D  
FROM R,S,T  
WHERE R.B = S.B  
AND S.C = T.C  
AND R.A < 10;
```


$$\Pi_{A,D} \left( T \bowtie \Pi_{A,C} (\sigma_{A<10}(R) \bowtie S) \right)$$

We eliminate B earlier!

In general, when is an attribute not needed...?



# Basic RA commutators

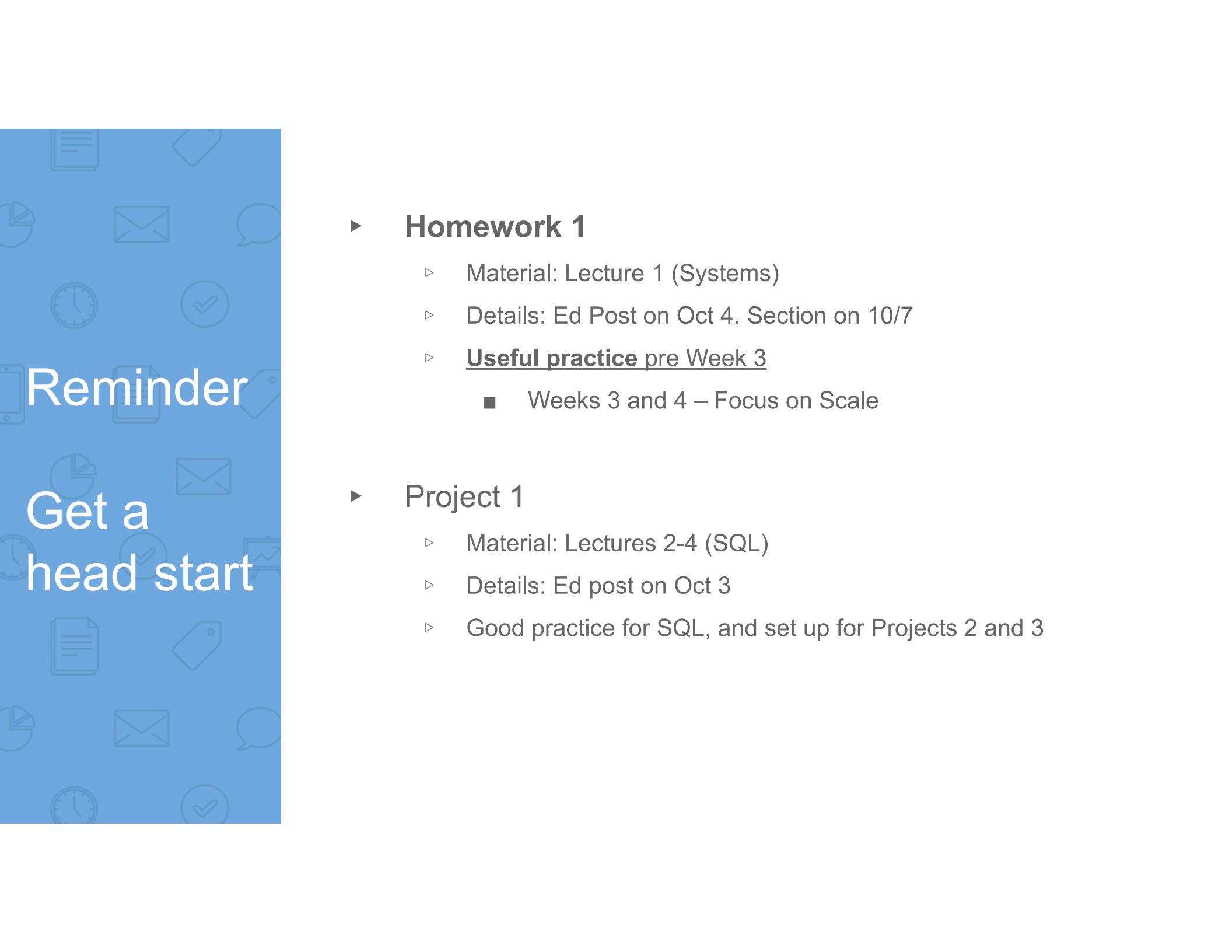
- Push **projection** through (1) **selection**, (2) **join**
  - Push **selection** through (3) **selection**, (4) **projection**, (5) **join**
  - *Also:* Joins can be re-ordered!
- ⇒ Note that this is not an exhaustive set of operations  
This covers *local re-writes*; *global re-writes possible but much harder*

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



# Takeaways

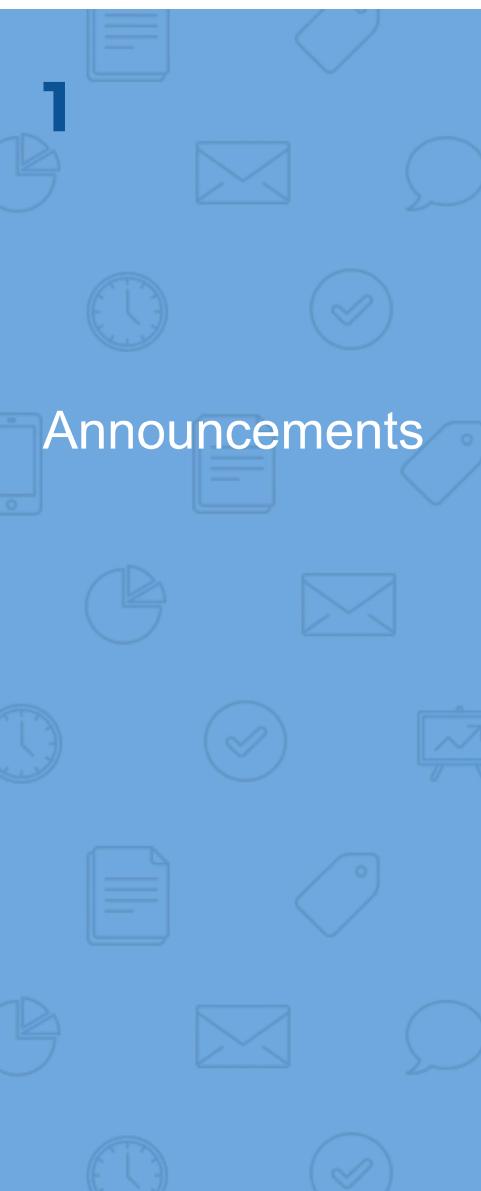
- This process is called logical optimization
- Many equivalent plans used to search for “good plans”
- Relational algebra is a simple and elegant abstraction



**Reminder**

**Get a head start**

- ▶ **Homework 1**
  - ▷ Material: Lecture 1 (Systems)
  - ▷ Details: Ed Post on Oct 4. Section on 10/7
  - ▷ **Useful practice** pre Week 3
    - Weeks 3 and 4 – Focus on Scale
- ▶ **Project 1**
  - ▷ Material: Lectures 2-4 (SQL)
  - ▷ Details: Ed post on Oct 3
  - ▷ Good practice for SQL, and set up for Projects 2 and 3



1

Announcements

1. Project 1 due Friday

2. My Office Hours

- ▷ Thursdays, right after lecture
- ▷ Friday on zoom

2

How?

Example  
Game App

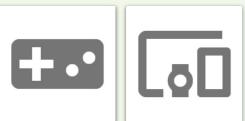
DB v0

(Recap lectures)

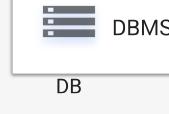
App designer

- Q1: 1000 users/sec writing?
- Q2: Offline?
- Q3: Support v1, v1' versions?

Mobile Game



Real-Time User Events



Report & Share  
Business/Product Analysis



Systems designer

- Q7: How to model/evolve game data?
- Q8: How to scale to millions of users?
- Q9: When machines die, restore game state gracefully?

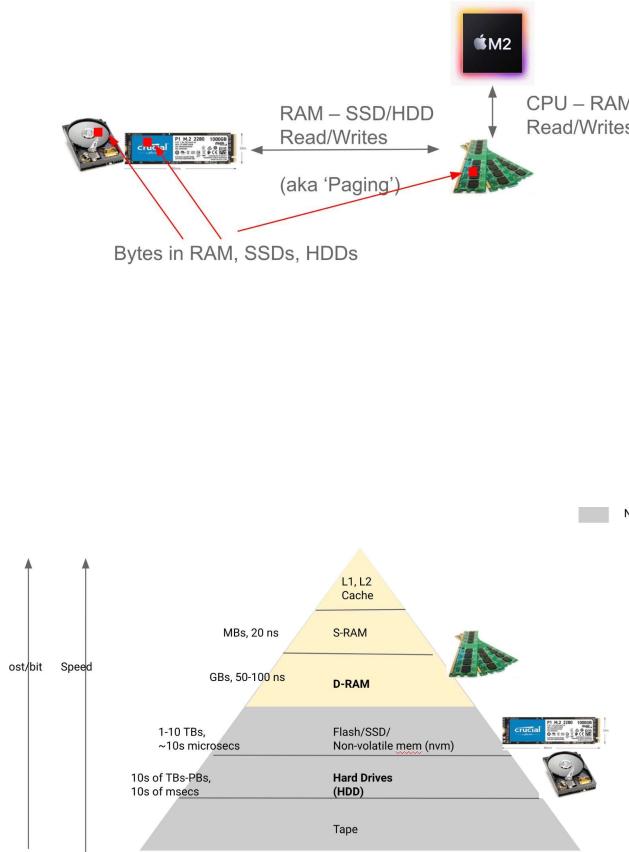
- Q4: Which user cohorts?
- Q5: Next features to build?
- Experiments to run?
- Q6: Predict ads demand?

Product/Biz designer

# Recall from Systems Primer

(Week 1 and HW1)

3



## Key Concepts

1. Data is stored in **Blocks** (aka “**partitions**”)
2. Sequential IO is 10x-100x+ faster than ‘many’ random IO
  - E.g., 1 MB (located sequentially) versus 1 Million bytes in random locations
3. HDDs/SSDs copy sequential ‘big’ blocks of bytes to/from RAM



Rough rule of thumb for data sizes

Data Size	Hardware/Storage	Algorithms	Languages + Libraries
‘Tiny’ (< 10 GBs)	Store in RAM	‘Usual’ CS Algorithms (sort, hash, ..)	Pandas (Python) + SQL (Spreadsheets for < 100 MBs)
‘Small’ (10GB - 10TB)	Store in SSD	CS145 Algorithms	SQL on Cloud
‘Big’ (> 10 TBs)	Store in HDD	CS145 Algorithms	SQL on Cloud

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



# Scale: Logical → Physical DB

So far... how to run SQL on “logical tables?”

5

## In this Section

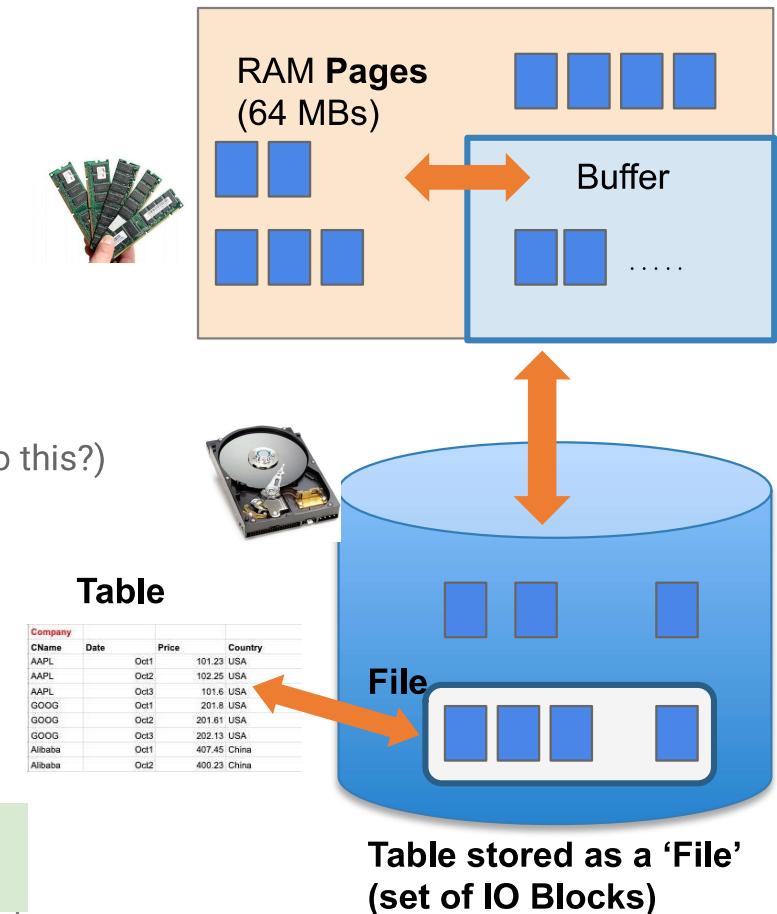
(Next weeks:  
Scale, Scale, Scale)

1. IO Model
2. Data layout
3. Indexing
4. Organizing Data and Indices

## 6

# DB Buffer in RAM

A buffer is a part of physical memory used to store intermediate data between disk and processes



DBs typically set RAM Page size = IO Block size (e.g, 64 MBs)  
 ⇒ In this class, we'll use Page, IO Block, Disk Block interchangeably

# Logical → Physical?

Logical Table

	Col3			
	CName	Date	Price	Country
Row1	AAPL	Oct1	101.23	USA
	AAPL	Oct2	102.25	USA
Row3	AAPL	Oct3	101.6	USA
	GOOG	Oct1	201.8	USA
Row5	GOOG	Oct2	201.61	USA
	GOOG	Oct3	202.13	USA
Row8	Alibaba	Oct1	407.45	China
	Alibaba	Oct2	400.23	China

Company(CName, StockPrice, Date, Country)

Next: How to store table in physical storage ‘files’?  
How to access rows/columns?  
(e.g., disk, RAM, clusters)



So far... how to run SQL on “logical tables?”

# Data Layout

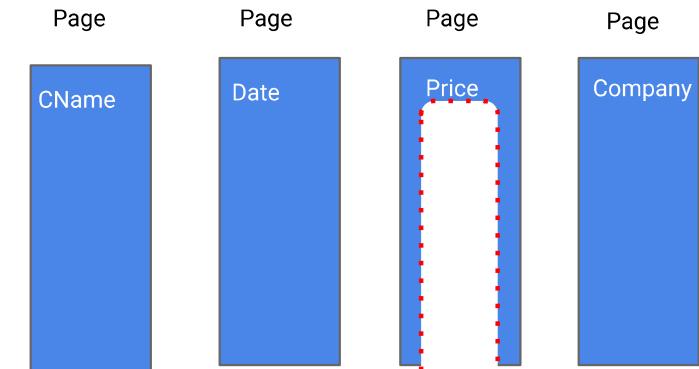
Logical Table

	Col3			
	CName	Date	Price	Country
Row1	AAPL	Oct1	101.23	USA
Row2	AAPL	Oct2	102.25	USA
Row3	AAPL	Oct3	101.6	USA
Row4	GOOG	Oct1	201.8	USA
Row5	GOOG	Oct2	201.61	USA
Row6	GOOG	Oct3	202.13	USA
Row7	Alibaba	Oct1	407.45	China
Row8	Alibaba	Oct2	400.23	China
Row Billion				

Company(CName, StockPrice, Date, Country)



Row based storage  
(aka Row Store)



Column based storage  
(aka Column Store)

# Example Origin Story of BigQuery (Dremel)

WebPage(URL, PageRank, Language, NumVisits, HTML)

Google index of Web Pages (~2005)

URL: 100 bytes  
PageRank: 8 bytes  
Language: 4 bytes  
Number of visitors: 4 bytes  
HTML: 2 MBs \* 5 versions ← **(the big column)**

⇒ Overall size = ~10 MBs/URL, stored in row format

Use case: What's PageRank of popular pages?

- E.g., select AVG(PageRank) ... where NumVisits > 100
- **Hours** to run query over 1 billion URLs. Why?
  - ⇒ Row based layout: Processing 10 MB\*1 billion urls
  - ⇒ Column based layout: Need only *PageRank* (8 bytes)+*NumVisits* (4 bytes) data. I.e., process only (4+8) bytes \* 1 billion urls (~1 million times faster)
- **Core idea: Analytical** (aka exploratory) queries usually focus on a few columns

[Optional: [Award](#) paper (2005-2020)]

# Data Layout

	Col3			
	CName	Date	Price	Country
Row1	AAPL	Oct1	101.23	USA
Row2	AAPL	Oct2	102.25	USA
Row3	AAPL	Oct3	101.6	USA
Row4	GOOG	Oct1	201.8	USA
Row5	GOOG	Oct2	201.6	USA
Row6	GOOG	Oct3	202.13	USA
Row7	Alibaba	Oct1	407.45	China
Row8	Alibaba	Oct2	400.23	China

Company(CName, StockPrice, Date, Country)

Row based storage  
(aka Row Store)

- Easy to retrieve and modify full tuple/row
- Classic way to organize data

Column based storage  
(aka Column Store)

- Aggregation queries -- e.g., AVG(Price)
- Compression – e.g., see Date column
- Scale to machine clusters – distribute columns to different machines
- Only retrieve columns you need for query
- Cons: Updates are more work

Tradeoffs on 'Workloads':

1. **Analytical:** Lots of data, many exploratory queries on few columns, e.g., Youtube analytics
2. Transactional: Good combination of reads and writes, e.g., Delta Airlines

## In this Section

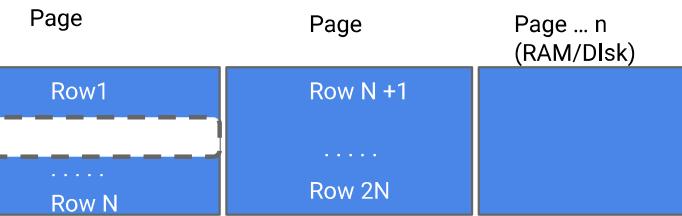
(Next weeks:  
Scale, Scale, Scale)

1. IO Model
2. Data layout
3. Indexing
4. Organizing Data and Indices

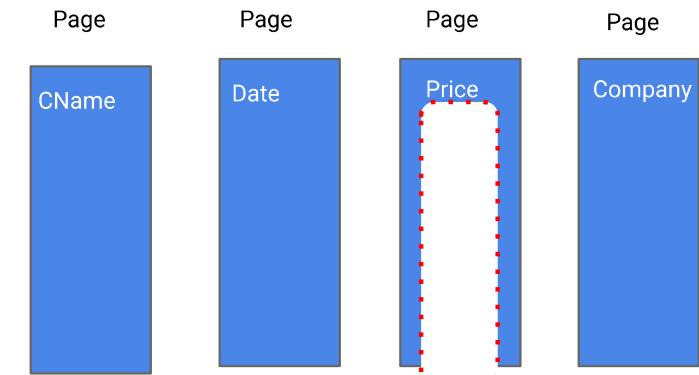
# How to find the right data fast?

Company				
CName	Date	Price	Country	
AAPL	Oct1	101.23	USA	
AAPL	Oct2	102.25	USA	
AAPL	Oct3	101.6	USA	
GOOG	Oct1	201.8	USA	
GOOG	Oct2	201.61	USA	
GOOG	Oct3	202.13	USA	
Alibaba	Oct1	407.45	China	
Alibaba	Oct2	400.23	China	

Company(CName, StockPrice, Date, Country)



Row based storage  
(aka Row Store)



Column based storage  
(aka Column Store)

Next: How to find AAPL Prices?

## Why study Indexes?

1. Fundamental unit for DB performance
2. Core indexing ideas have become **stand-alone systems**
  - E.g., search in google.com
  - Data blobs in noSQL, Key-value stores
  - Embedded join processing

14

## Example

### Find Book in Library



#### Design choices?

- Scan through each aisle
- Lookup pointer to book location, with librarian's organizing scheme

15

Example

Find Book  
in Library  
With Index

# the DEWEY DECIMAL SYSTEM

The grid consists of two rows of five cards each. Each card features a color-coded background, an icon, and a category number followed by its name.

Main Class	Division	Section	Classification
000	GENERAL KNOWLEDGE		
100	PHILOSOPHY & PSYCHOLOGY		
200	RELIGION		
300	SOCIAL SCIENCES		
400	LANGUAGES		
500	SCIENCE		
600	TECHNOLOGY		
700	ARTS & RECREATION		
800	LITERATURE		
900	HISTORY & GEOGRAPHY		

**Understanding the Dewey Decimal System**

746.43

UVICSEA EAST CAMPUS LIBRARY. © MADE BY MAGGIE APPLETON  
amyallender.com

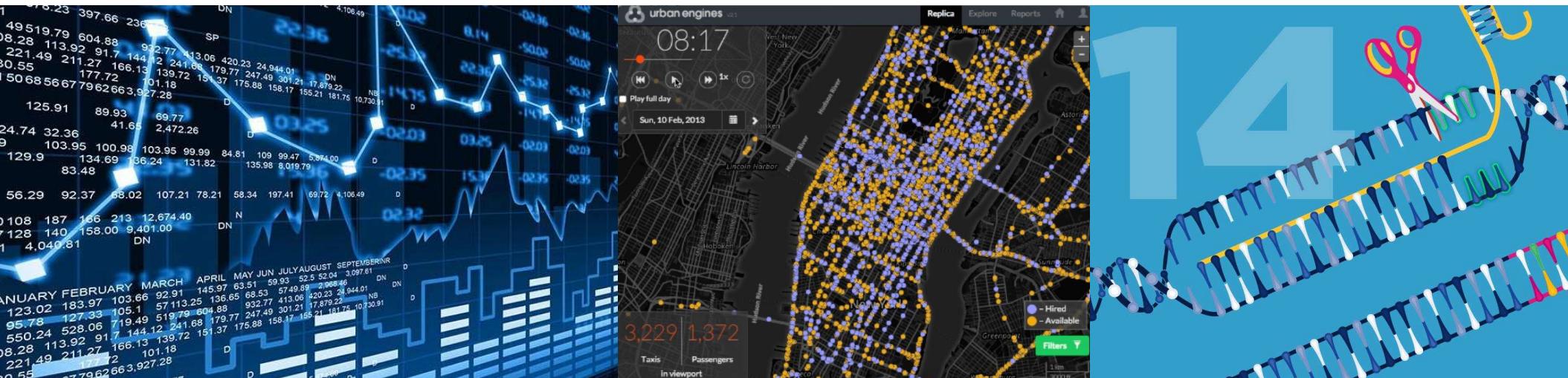
Index Cards

Algorithm for book titles

- Find right category
- Lookup Index, find location
- Walk to aisle. Scan book titles. Faster if books are sorted

16

# Kinds of Indexes (different data types)



## Index for

- Strings, Integers
- Time series, GPS traces, Genomes, Video sequences
- Advanced: Equality vs Similarity, Ranges, Subsequences

Composites of above

# Example: Search on stocks

Company(CName, StockPrice, Date, Country)

Company			
CName	Date	Price	Country
AAPL	Oct1	101.23	USA
AAPL	Oct2	102.25	USA
AAPL	Oct3	101.6	USA
GOOG	Oct1	201.8	USA
GOOG	Oct2	201.61	USA
GOOG	Oct3	202.13	USA
Alibaba	Oct1	407.45	China
Alibaba	Oct2	400.23	China

```
SELECT *
FROM Company
WHERE CName like 'AAPL'
```

```
SELECT CName, Date
FROM Company
WHERE Price > 200
```

Q: On which attributes would you build indexes?

A: On as many subsets as you'd like. Look at query workloads.

# Example



**CName\_Index**

CName
AAPL
AAPL
AAPL
GOOG
GOOG
GOOG
Alibaba
Alibaba

Block #

Company				
CName	Date	Price	Country	
AAPL	Oct1	101.23	USA	
AAPL	Oct2	102.25	USA	
AAPL	Oct3	101.6	USA	
GOOG	Oct1	201.8	USA	
GOOG	Oct2	201.61	USA	
GOOG	Oct3	202.13	USA	
Alibaba	Oct1	407.45	China	
Alibaba	Oct2	400.23	China	

Block #

**PriceDate\_Index**

Date	Price	Block #
Oct1	101.23	
Oct2	102.25	
Oct3	101.6	
Oct1	201.8	
Oct2	201.61	
Oct3	202.13	
Oct1	407.45	
Oct2	400.23	

1. Index contains <search key> → <Block #>: e.g., IO block number.
  - o In general, “pointer” to where the record is stored (e.g., RAM page, IO block number or even machine + IO block)
  - o **ReadBlock(Block #)** // Pseudocode from Systems Primer
  - o Index is conceptually a table. In practice, implemented very efficiently (see how soon)
2. Can have multiple indexes to support multiple search keys



# Indexes (definition)

Maps search keys to sets of rows in table

- Provides efficient lookup & retrieval by search key value (much faster than scanning all rows and filtering, usually)
- Key operations: Lookup, Insert, Delete

An index can be

- Primary: Index on a set of columns that includes unique primary key. And no duplicates
- Secondary: Non-primary index (on any set of columns) and may have duplicates [much of our focus -- primary is an easier version]
- Advanced: build across rows, across tables

# Covering Indexes

PriceDate\_Index

Date	Price	Block #
Oct1	101.23	
Oct2	102.25	
Oct3	101.6	
Oct1	201.8	
Oct2	201.61	
Oct3	202.13	
Oct1	407.45	
Oct2	400.23	

An index covers for a specific query if the index contains all the needed attributes

I.e., query can be answered using the index alone!

The “needed” attributes are the union of those in the SELECT and WHERE clauses...

Example:

```
SELECT Date, Price  
FROM Company  
WHERE Price > 101
```

## Why study Indexes?

1. Fundamental unit for DB performance
2. Core indexing ideas have become **stand-alone systems**
  - E.g., search in google.com
  - Data blobs in noSQL, Key-value stores
  - Embedded join processing

## In this Section

(Next weeks:  
Scale, Scale, Scale)

1. IO Model
2. Data layout
3. Indexing
4. Organizing Data and Indices

23

Big Scale

Roadmap

Hashing

Sorting

Hashing, Sorting solves “all” known data scale problems

- + Boost with a few patterns -- Cache, Parallelize, Pre-fetch



T H E   B I G   I D E A

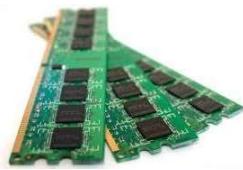
Navigation icons: back, forward, search, etc.

Note: Works for Relational, noSQL  
(e.g. mySQL, postgres, BigQuery, BigTable, MapReduce, Spark, Mongo)

24

Big Scale

Roadmap



Primary data structures/algorithms

Hashing

HashTables  
(hash(key) --> value)  
(e.g., hash("Apple") --> 5348)

Sorting

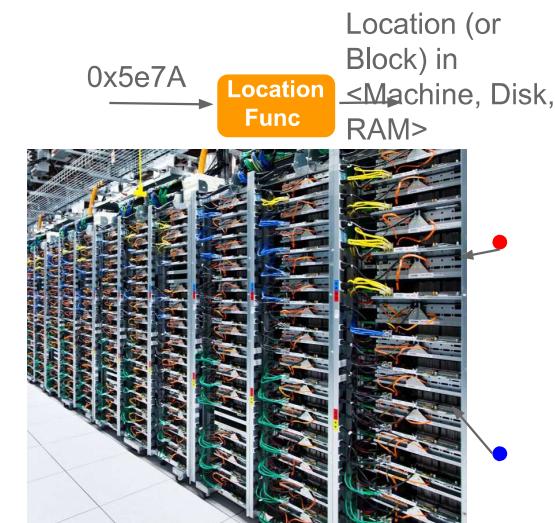
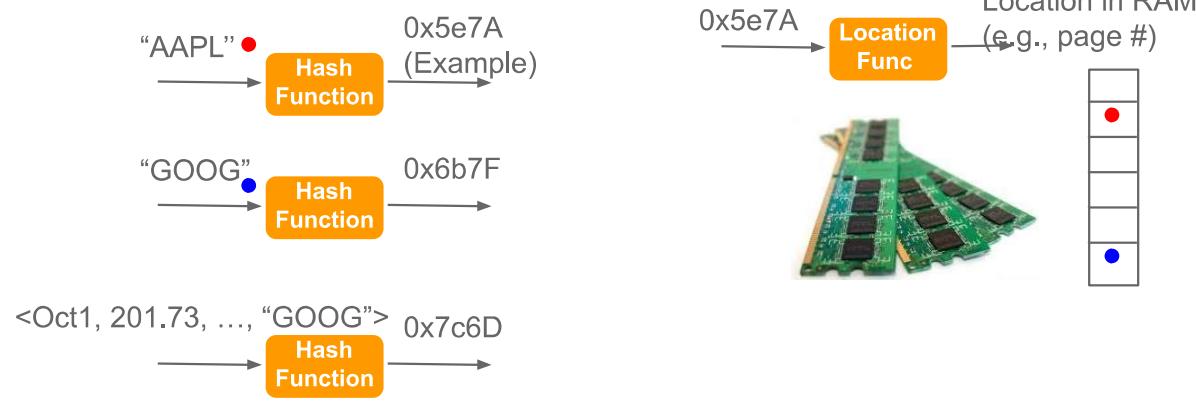
BucketSort, QuickSort  
MergeSort

?????



# Hashing

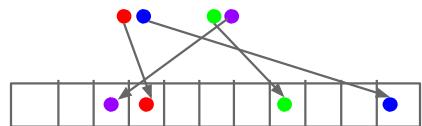
# Recall: Hashing



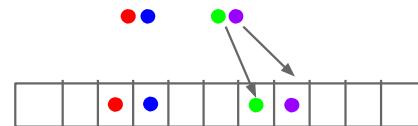
- Magic of hashing:
  - A hash function  $h_B$  maps into  $[0, B-1]$ , nearly uniformly
  - Also called sharding function
- A hash collision is when  $x \neq y$  but  $h_B(x) = h_B(y)$ 
  - Note however that it will never occur that  $x = y$  but  $h_B(x) \neq h_B(y)$

# Hashing ideas for scale

- Idea: Multiple hash functions (uncorrelated to spread data)
  - $h_i(x), h_{i+1}(x), h_{i+2}(x), h_{i+3}(x), \dots$
- Idea: Locality sensitive hash functions (for k-dimensional data)
  - Special class of hash functions to keep spread ‘local’



Regular hash functions  
(spread all over)



Locality Sensitive Hash (LSH) functions  
(spread in closer buckets, with high probability)

28

## Big Scaling (with Indexes)

## Roadmap



Primary data structures/algorithms

Hashing

Sorting

HashTables  
( $\text{hash}_i(\text{key}) \rightarrow \text{value}$ )

BucketSort, QuickSort  
MergeSort2Lists,  
MergeSort

MergeSortedFiles,  
MergeSort

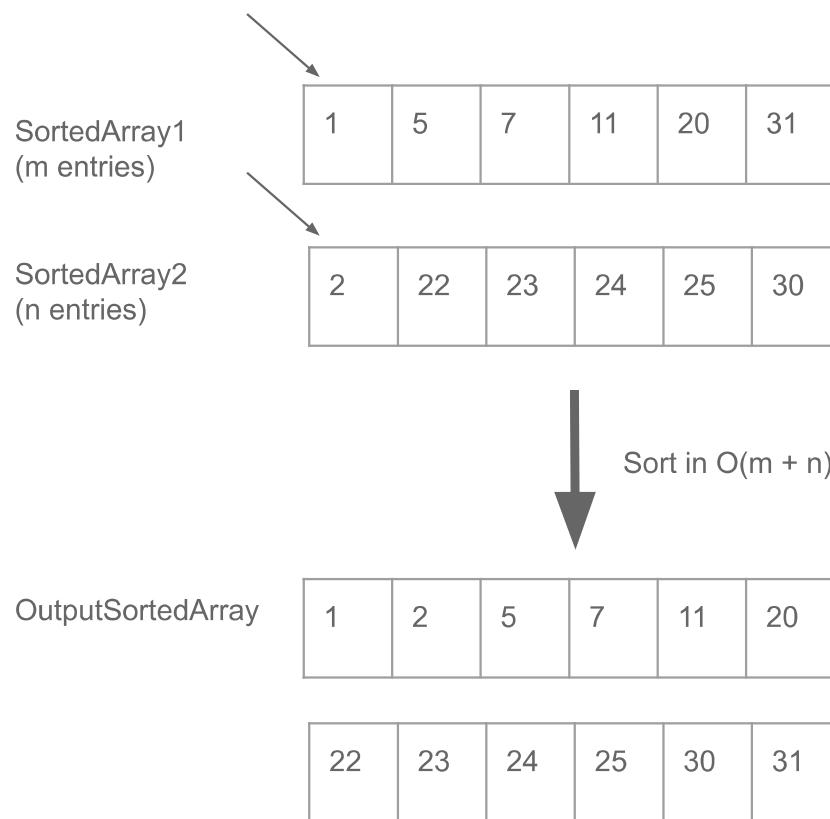


# Sorting 1: External Merge Algorithm

Given two Sorted Files...

30

## MergeSortedFiles (in RAM)



Idea: To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list.

## Challenge: Merging Big Files with Small Memory



How do we *efficiently* merge two sorted files when both are much larger than our RAM buffer?

Key point: **Disk IO (R/W) dominates the algorithm cost**

Our first example of an “**IO aware**” algorithm / cost model



## External Merge Algorithm

- Input: 2 sorted files of length  $M$  and  $N$  ( $\text{length} = \# \text{ of IO blocks}$ )
- Output: 1 sorted list of length  $M + N$
- Required: At least 3 Buffer Pages
- IOs:  $2(M+N)$

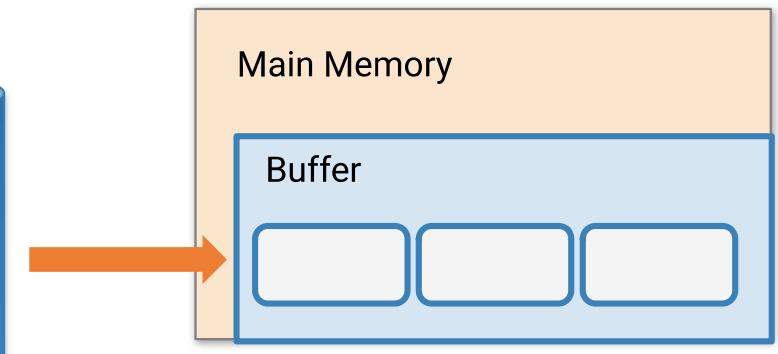
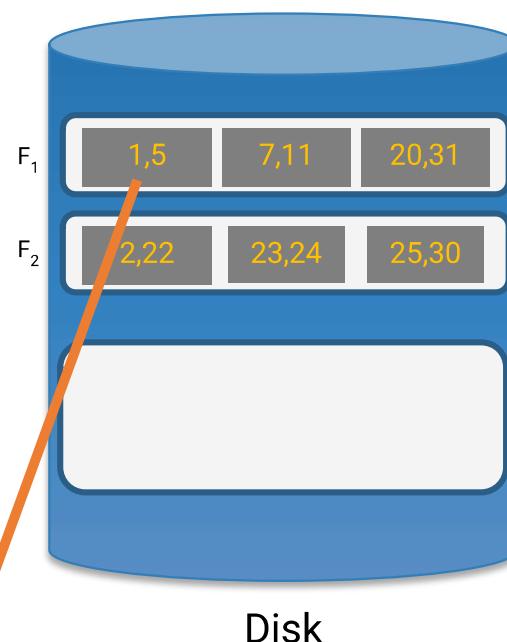
# External Merge Algorithm



Input:  
Two sorted  
files F1, F2

Output:  
One merged  
sorted file

Notation:  
F1 has 3 pages. F1's Page1 has 2 values  
(1, 5).



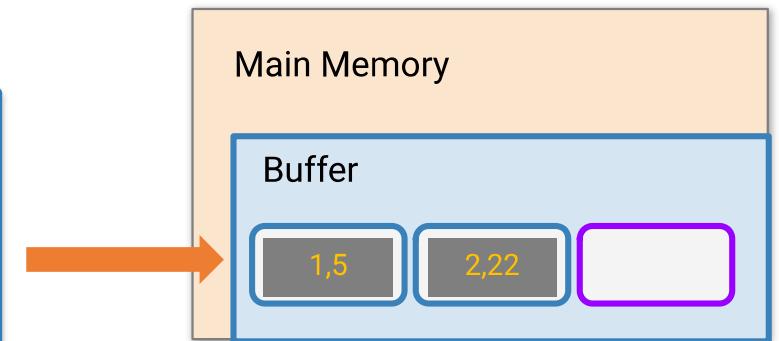
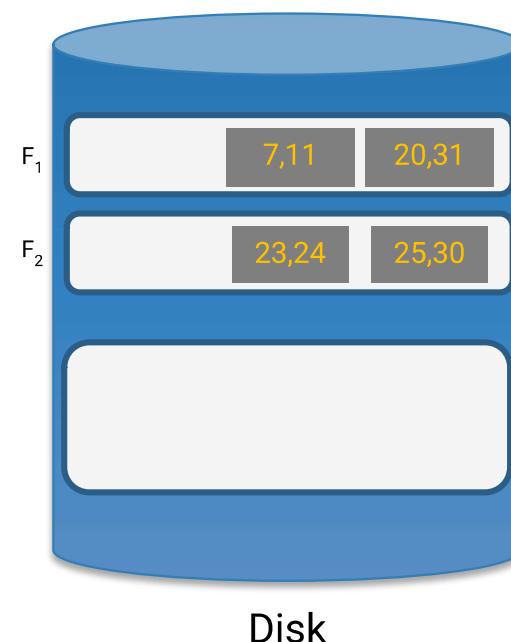
Example:  
B = 3  
Input: Keep 2 pages  
Output: Keep 1 page

# External Merge Algorithm



Input:  
Two sorted  
files

Output:  
One *merged*  
sorted file

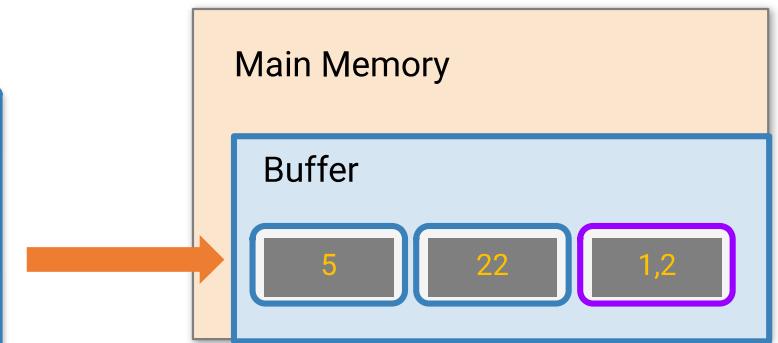
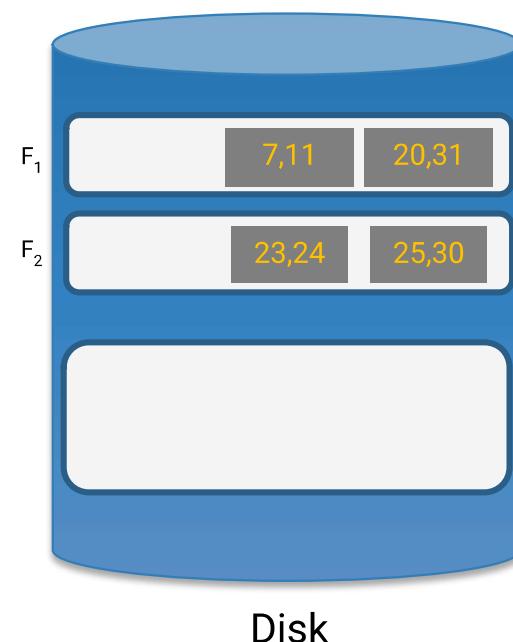


# External Merge Algorithm



Input:  
Two sorted  
files

Output:  
One *merged*  
sorted file

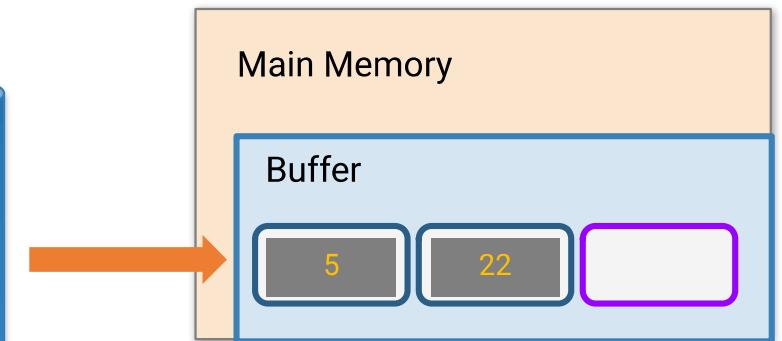
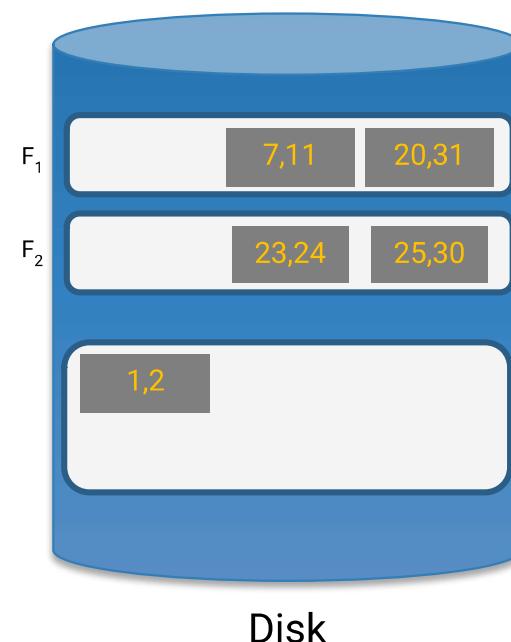


# External Merge Algorithm



Input:  
Two sorted  
files

Output:  
One *merged*  
sorted file

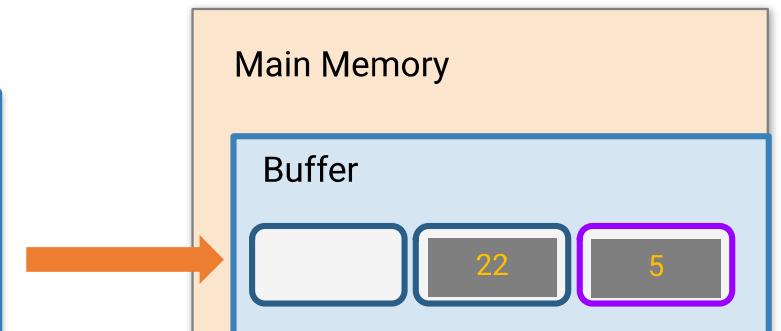
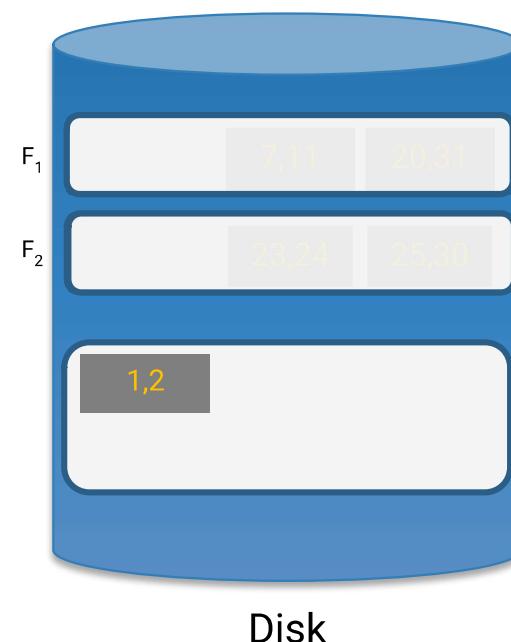


# External Merge Algorithm



Input:  
Two sorted  
files

Output:  
One *merged*  
sorted file



Main Memory

Buffer



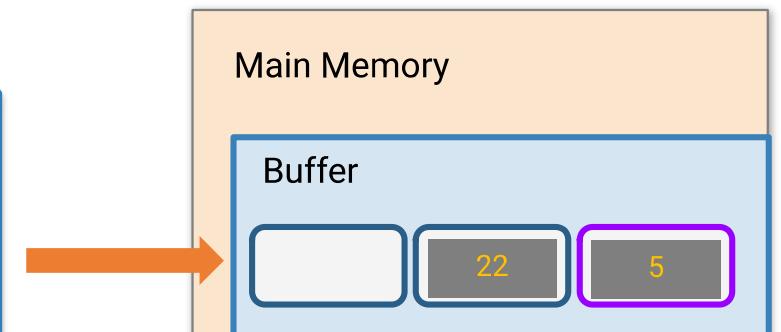
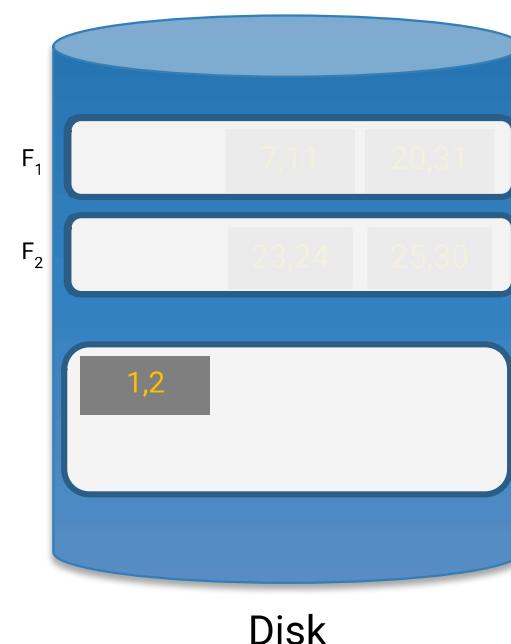
This is all the algorithm “sees”...  
Which file to load a page from next?

# External Merge Algorithm



Input:  
Two sorted  
files

Output:  
One *merged*  
sorted file



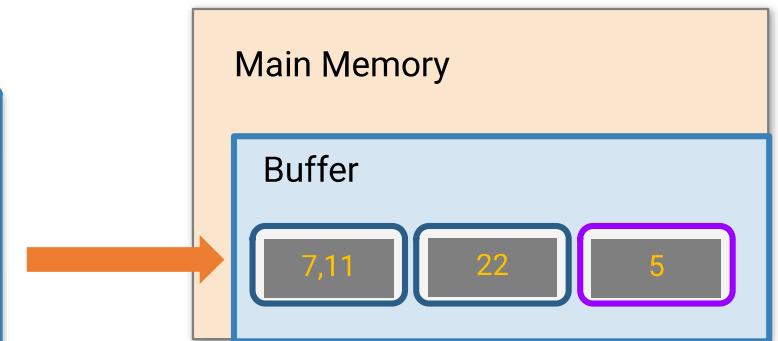
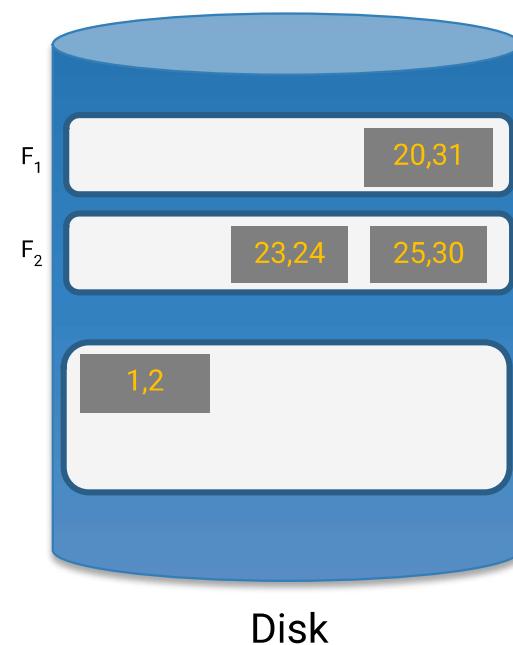
We know that  $F_2$  only contains values  $\geq 22$ ... so we should load from  $F_1$ !

# External Merge Algorithm



Input:  
Two sorted  
files

Output:  
One *merged*  
sorted file

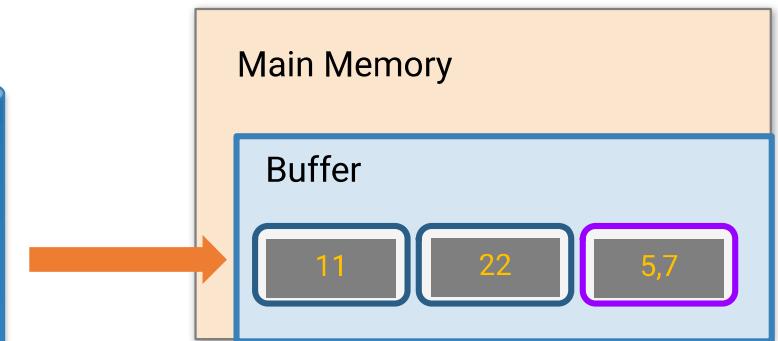
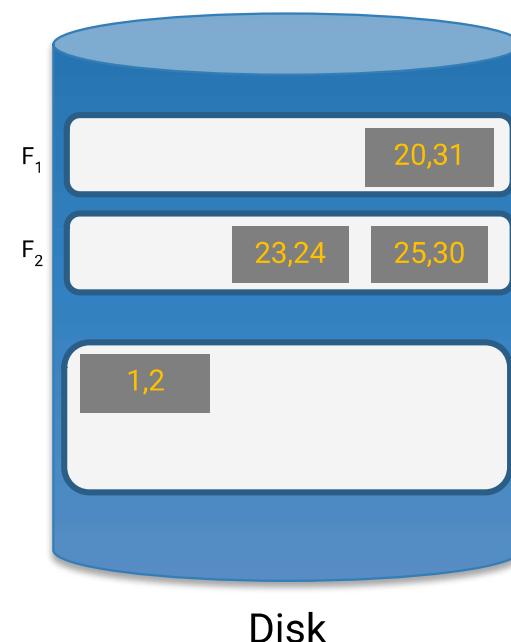


# External Merge Algorithm



Input:  
Two sorted  
files

Output:  
One *merged*  
sorted file

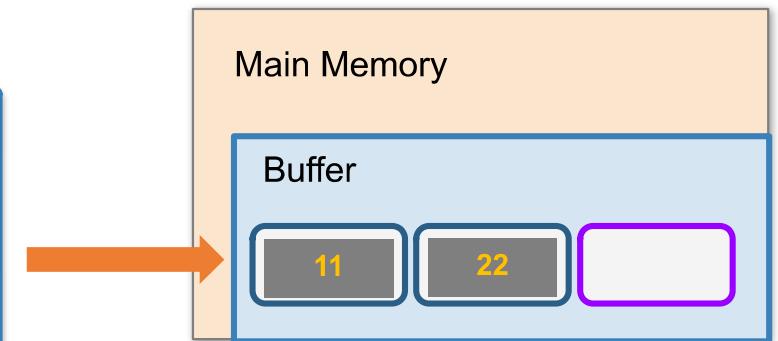


# External Merge Algorithm



Input:  
Two sorted  
files

Output:  
One *merged*  
sorted file

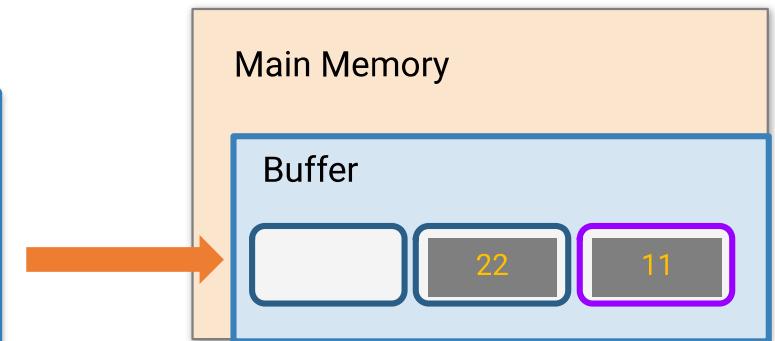
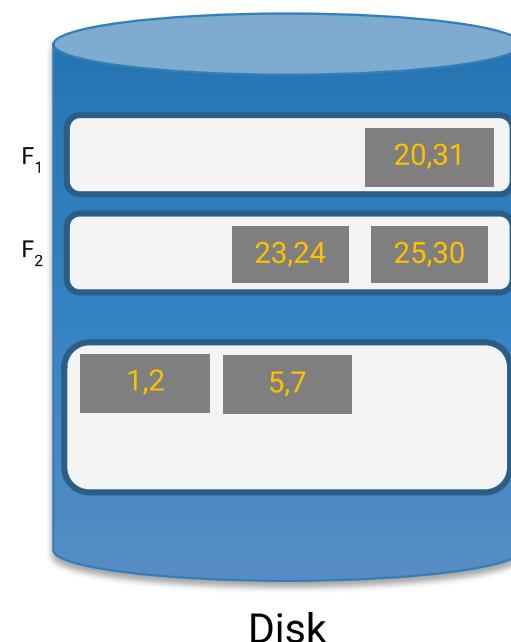


# External Merge Algorithm



Input:  
Two sorted  
files

Output:  
One *merged*  
sorted file



Main Memory

Buffer

22

11

And so on...



We can merge lists of arbitrary length with only 3 buffer pages.

If lists of size M and N, then

Cost:  $2(M+N)$  IOs

Each **page** is read once, written once

1. Recall:  $n \log n$  for sorting in RAM still true. Negligible vs IO costs from disks.
2. With  $B+1$  buffer pages, can merge  $B$  lists. How?

## Recap: External Merge Algorithm

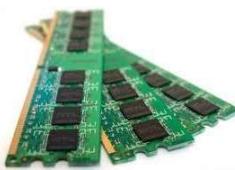
- Suppose we want to merge two sorted files both much larger than main memory (i.e. the buffer)
- We can use the external merge algorithm to merge files of *arbitrary length* in  $2*(N+M)$  IO operations with only 3 buffer pages!

Our first example of an “IO aware”  
algorithm / cost model

45

## Big Scaling (with Indexes)

## Roadmap



Primary data structures/algorithms

Hashing

HashTables  
 $(hash_i(key) \rightarrow value)$

Sorting

BucketSort, QuickSort  
MergeSort

[ExternalMerge](#)

?????

## In this Section

(Next weeks:  
Scale, Scale, Scale)

1. IO Model
2. Data layout
  - ▷ row vs column storage
3. Indexing
4. Organizing Data and Indices
  - ▷ Hashing, Sorting, Counting

1

## Plan for the day

1. How to sort **10 TB** files with **1 GB** of RAM?

A classic problem in computer science!

2. How to Search over **1 Trillion** items in **< 1 second**, with **Index**?
  - a. (e.g., Youtube's catalog, Amazon's products, etc.)

A classic problem in industry!

# 2

## Big Scale

## Roadmap

Hashing

Sorting

Hashing-Sorting solves “all” known data scale problems :=)

- + Boost with a few patterns -- Cache, Parallelize, Pre-fetch



**T H E   B I G   I D E A**

Note

Works for Relational, noSQL

(e.g. mySQL, postgres, BigQuery, BigTable, MapReduce, Spark)

# 3

## Big Scale Lego Blocks

## Roadmap



Primary data structures/algorithms

Hashing

Sorting

HashTables  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

HashFunctions  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

HashFunctions  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

BucketSort, QuickSort  
MergeSort

ExternalMerge

ExternalMergeSort

ExternalMergeSort

# 4

## Why are Sort Algorithms Important?

Why not just use quicksort in main memory??

- How to Sort **10TB - 100 TB** of data?
- E.g., with 16GB of RAM, i.e., 0.1%-0.01% of data size...

A classic problem in computer science!

Example use cases

1. Query results in sorted order is extremely common
  - e.g., find students in increasing GPA order
2. Core building block for data compression, indexing, joins



# ExternalMergeSort() Algorithm



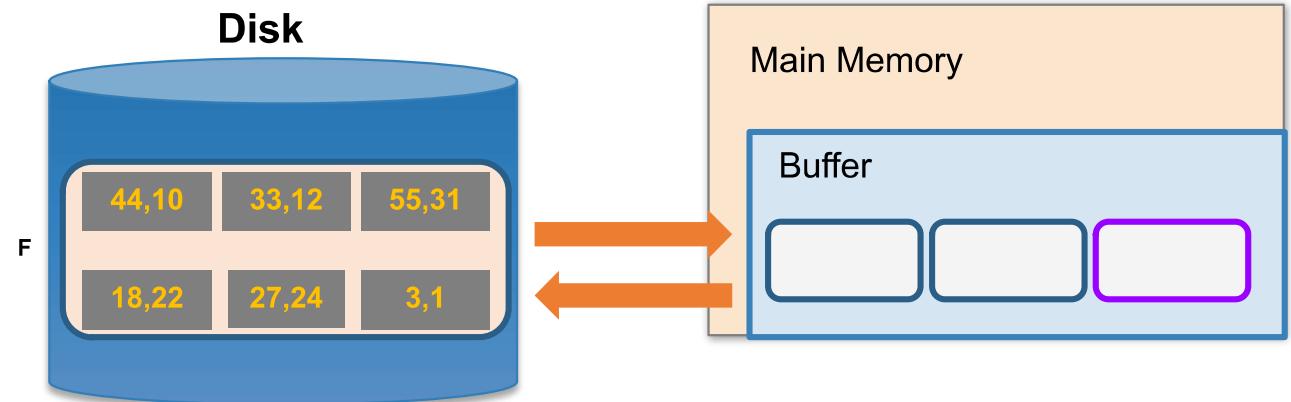
## So how do we sort big files?

1. Split into chunks small enough to sort in memory (“SortRuns”)
2. *ExternalMerge()* [from previous lecture] to merge runs from Step 1
3. Keep merging the resulting runs (each time = a “pass”) until left with one sorted file!

# ExternalMergeSort()

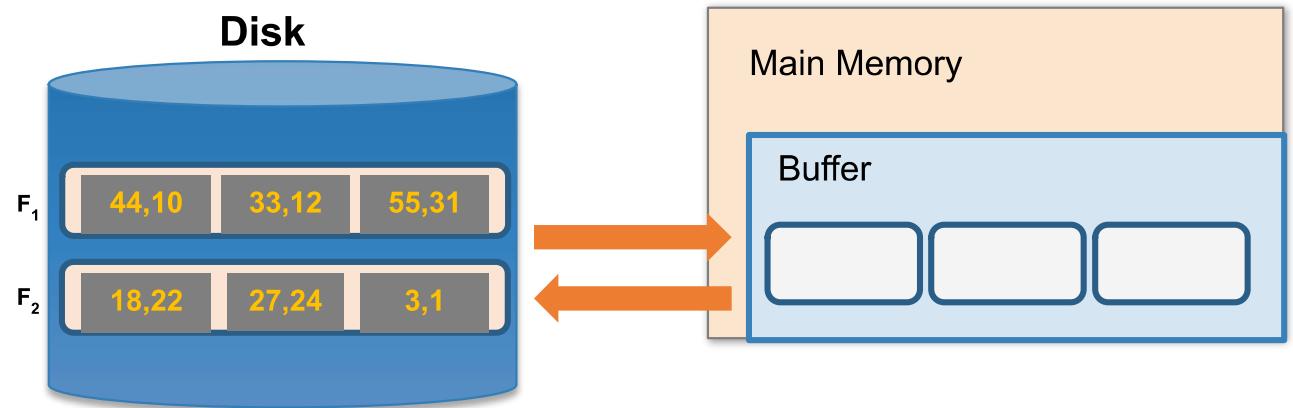
- Example
- 3 Buffer pages
  - 6-page file

Orange file  
= unsorted



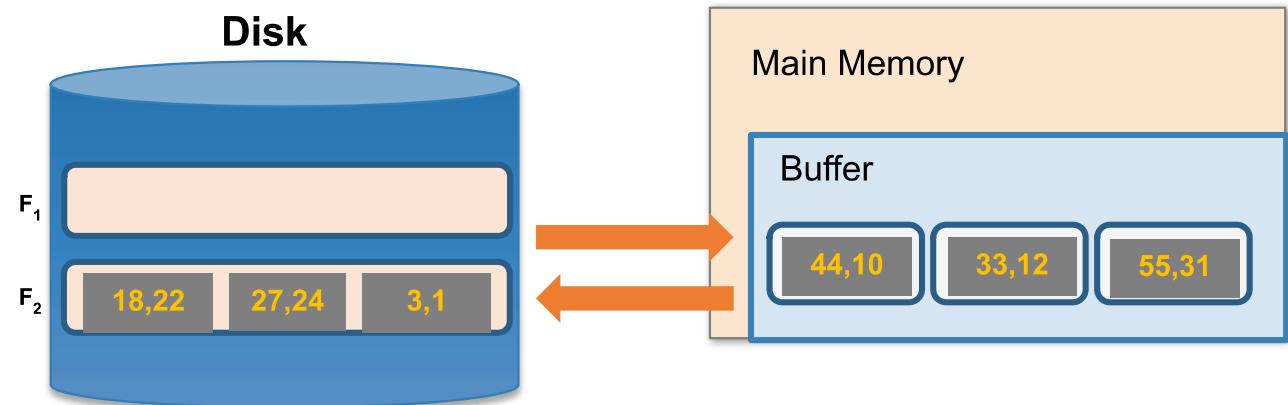
1. Split into chunks small enough to **sort in memory**

# ExternalMergeSort()



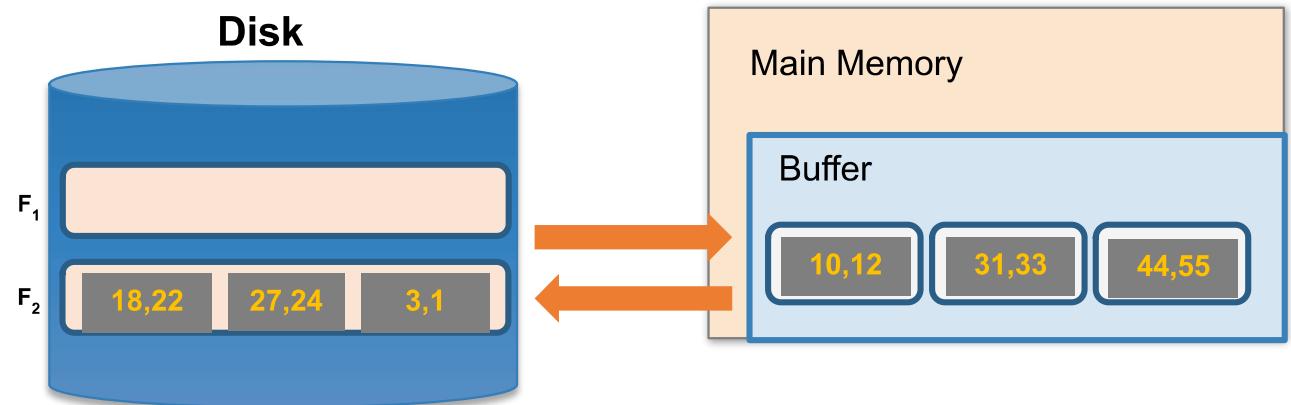
1. Split into chunks small enough to **sort in memory**

# ExternalMergeSort()



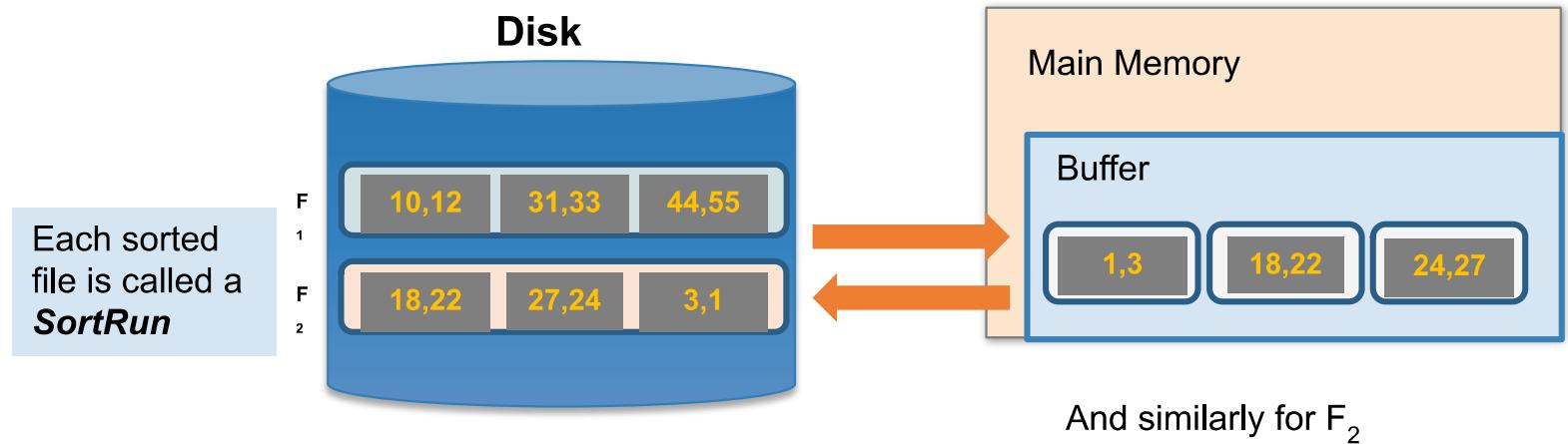
1. Split into chunks small enough to **sort in memory**

# ExternalMergeSort()



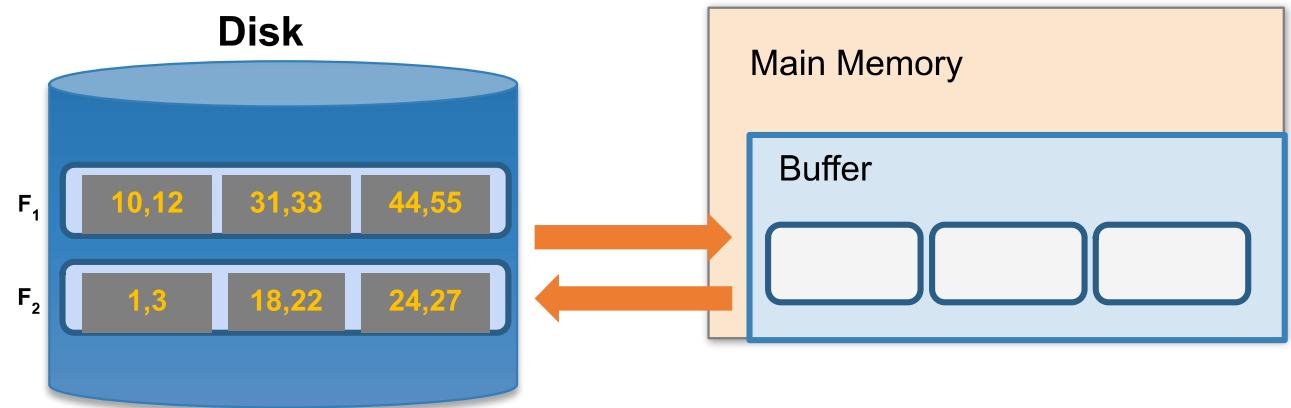
1. Split into chunks small enough to **sort in memory**

# ExternalMergeSort()



1. Split into chunks small enough to **sort in memory**

# ExternalMergeSort()



2. Now just run the **ExternalMerge()** algorithm [from last lecture] & we're done!



## Example: Calculating IO Cost

For 3 buffer pages, 6 page file: **Let cost(Read) = R, cost(Write) = W**

1. Split into two 3-page files and sort in memory  
 $= 1R + 1W \text{ per page} = 6*(1R + 1W)$
2. Merge each pair of sorted chunks with *external merge algorithm*  
*Recall: ExtMerge costs  $[1R+1W]$  per page (We have  $3 + 3$  pages after Step 1)*  
 $= [1R + 1W] * (3 + 3)$
3. Total cost =  $12R + 12W$  IO      **Often, we use 1 IO for R and W. I.e.,  $R = W = 1$  IO  
(Alternate examples in HMK2)**

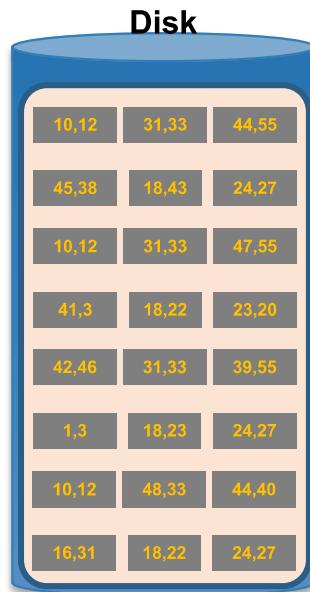
Note: Why “IO”? How does it map to access (e.g., HDD seek) and scan?

- Handles “Are blocks contiguous for read (or write)?”
  - No? ‘3 R’  $\Rightarrow$  Cost = 3 access + scan 3 blocks
  - Yes? ‘3 R’  $\Rightarrow$  Cost = 1 access + time to scan 3 blocks

⇒ For such problems, we’ll use IO units for simplicity

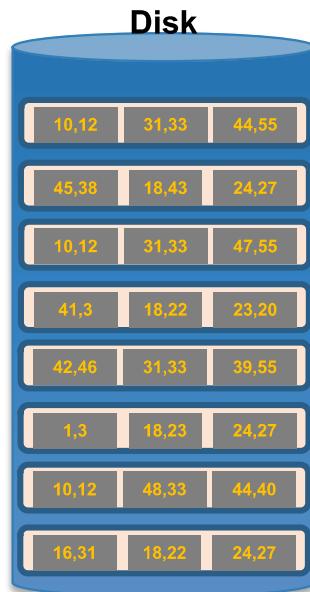
14

# ExternalMergeSort() on Larger Files



Assume we still only have 3 buffer pages  
(Buffer not pictured)

# ExternalMergeSort() on Larger Files



1. Split into files small enough to sort in buffer...

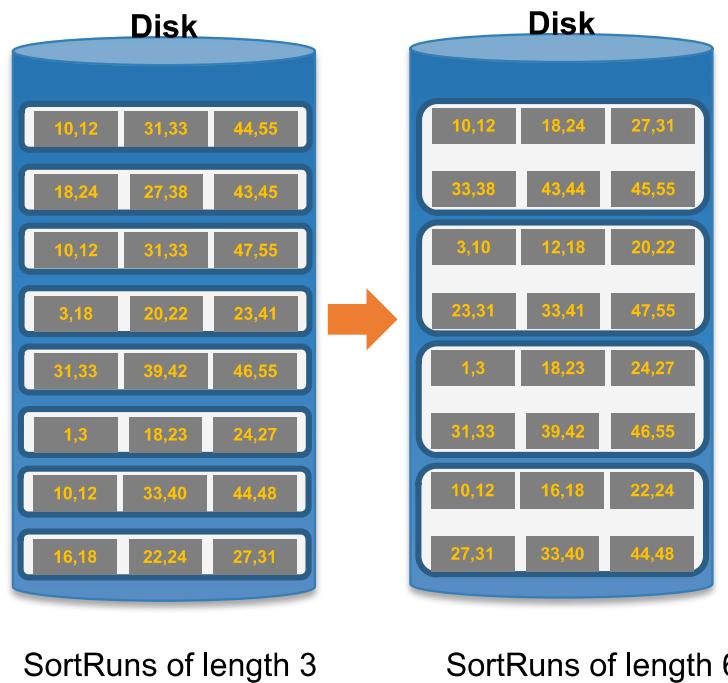
# ExternalMergeSort() on Larger Files



1. Split into files small enough to sort in buffer... and sort

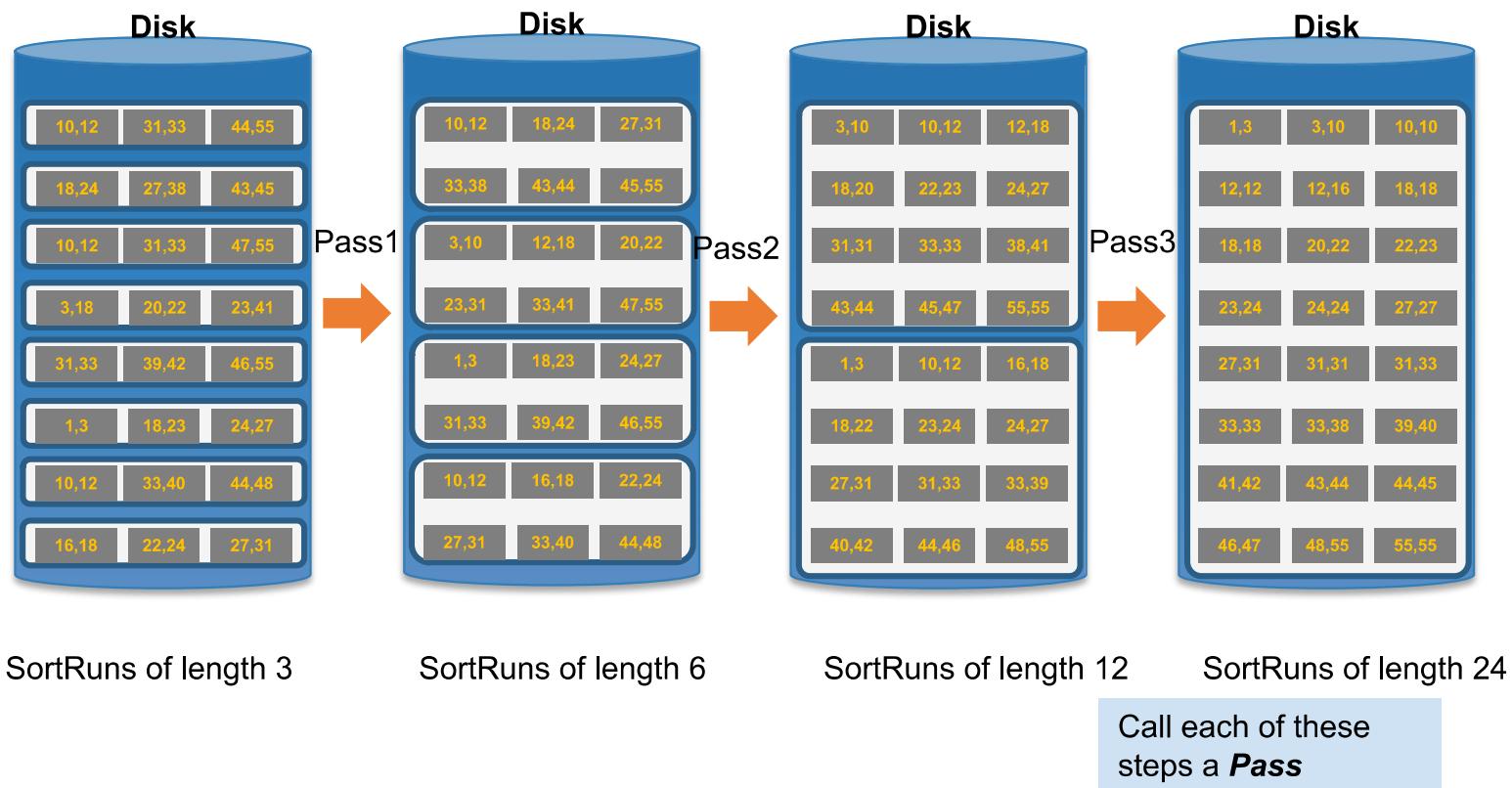
Each sorted file is a **SortRun**

# ExternalMergeSort() on Larger Files



2. Now merge pairs  
of **SortRuns**...to  
produce more  
**SortRuns**

# ExternalMergeSort() on Larger Files



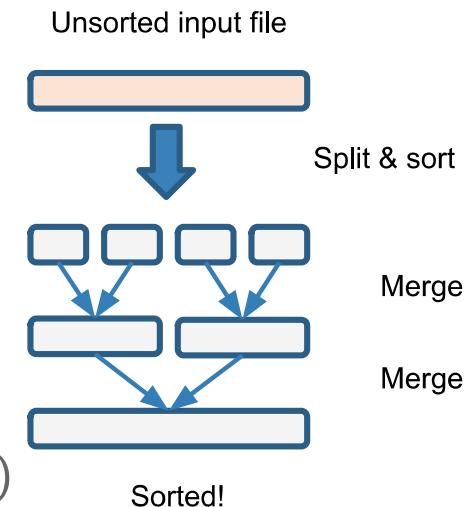


## ExternalMergeSort() for N pages

[Note: We assume  $R = W = 1$  IO here. That is,  $(R+W)N = 2N$  IO]

For N page file, we do

- Sort small chunks in  $2N$  IOs
- Merge:
  - $\lceil \log_2 N \rceil$  passes
  - $2N$  IOs/pass  
(each page is read+write once)



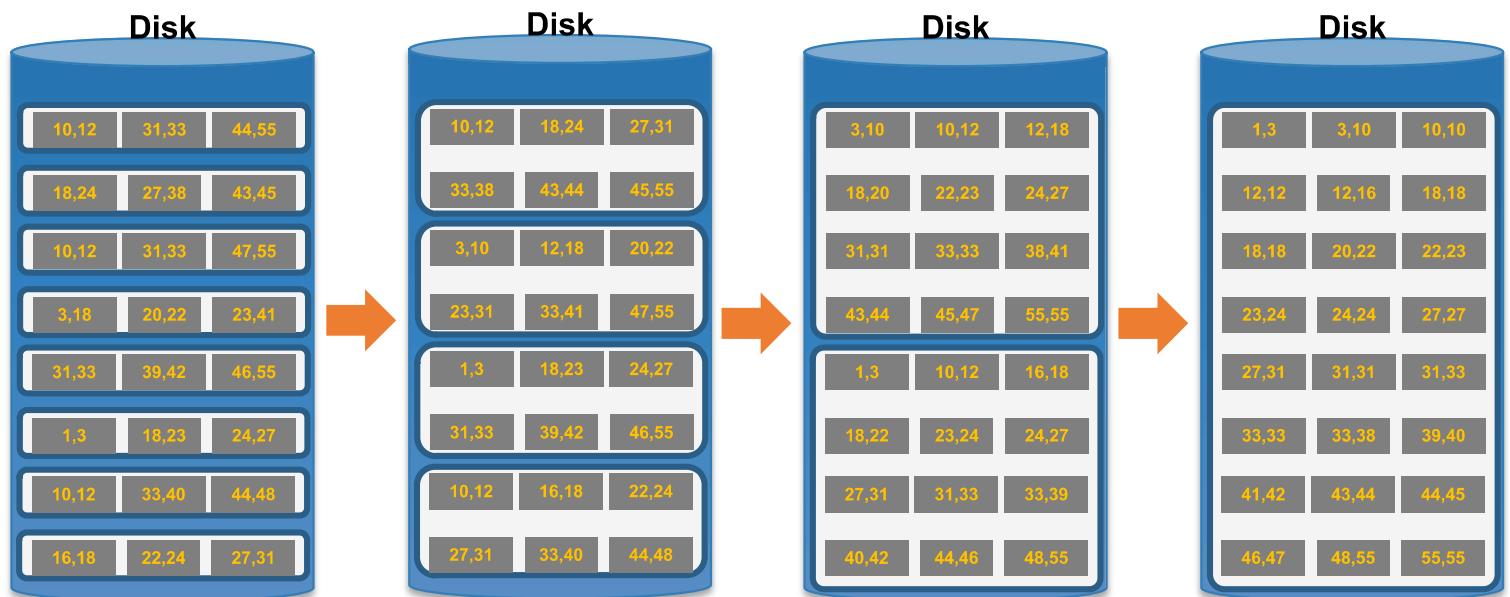
IO Cost  $\sim$

$$2N \lceil \log_2 N \rceil + 2N$$

Note: We'll do much better in next 10 slides. I.e., this is only a partial formula.

20

## Example Optimization – ‘Repacking’



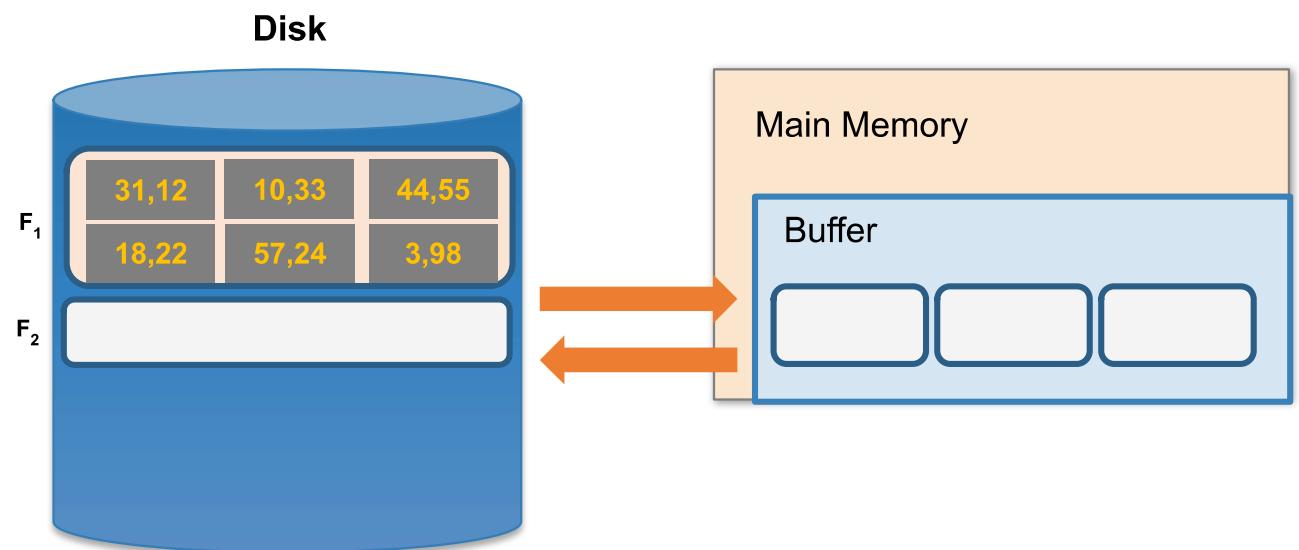
Idea: What if it's already 'nearly' or 'partly' sorted?

Can we be smarter with buffer? **Optimistic sorting**

21

# Rearranging Example: 3 page buffer

Start with unsorted single input file, and load 2 pages

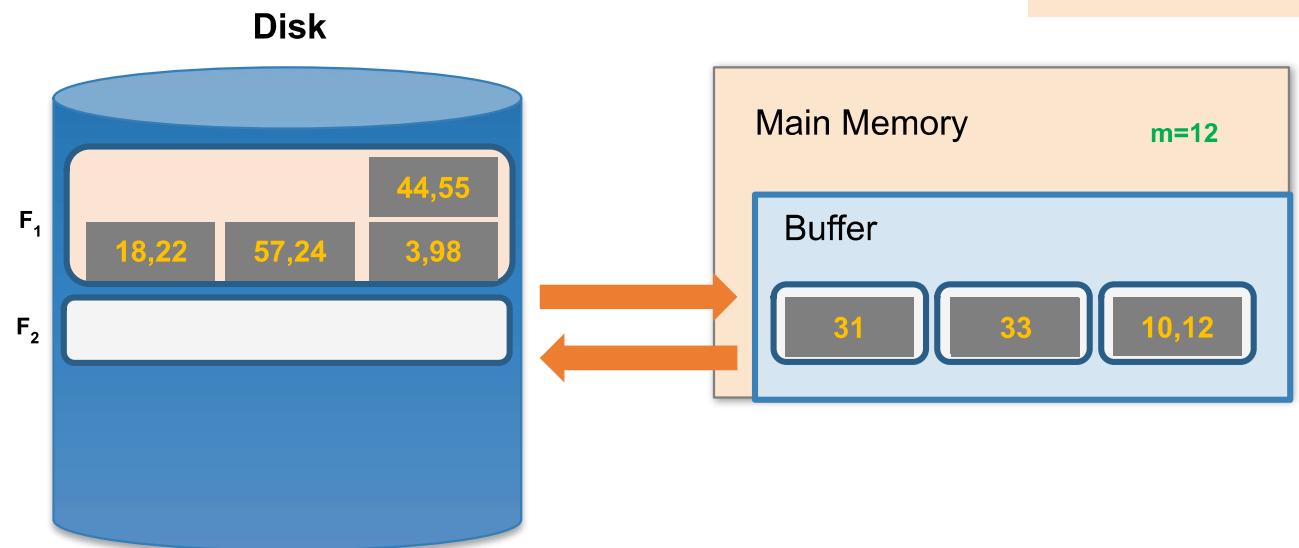


22

# Repacking Example: 3 page buffer

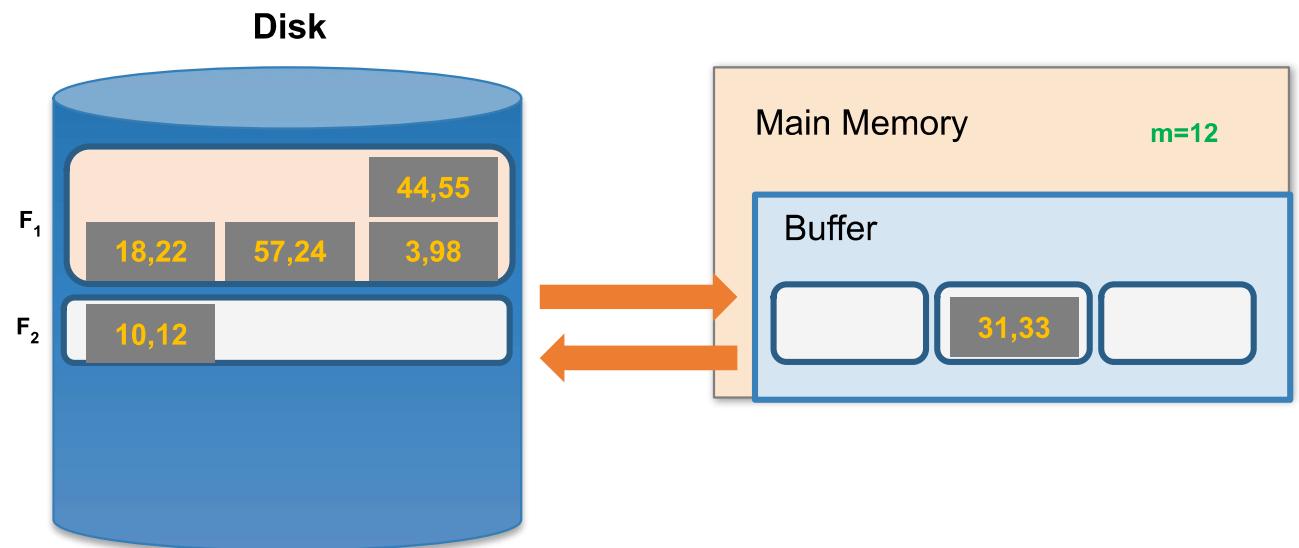
Take the minimum two values, and put in output page

Also keep track of  
**max (last) value** in  
current SortRun...



# Rpacking Example: 3 page buffer

- Next, **repack**

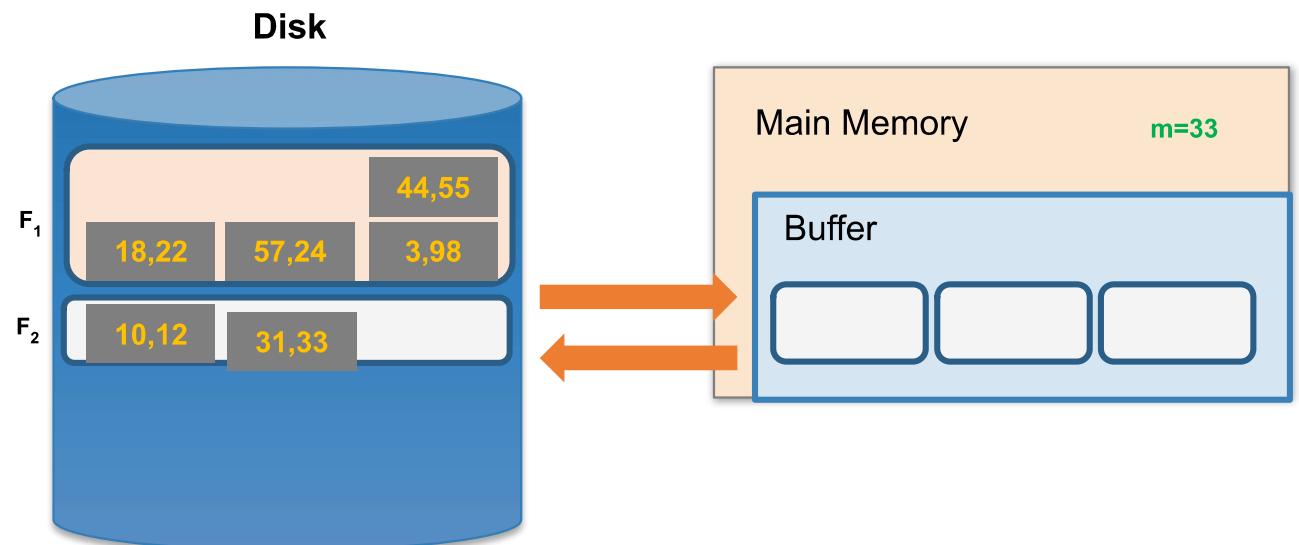


(Notice  $m=12$  – indicate the largest value written to F2 so far)

24

# Rpacking Example: 3 page buffer

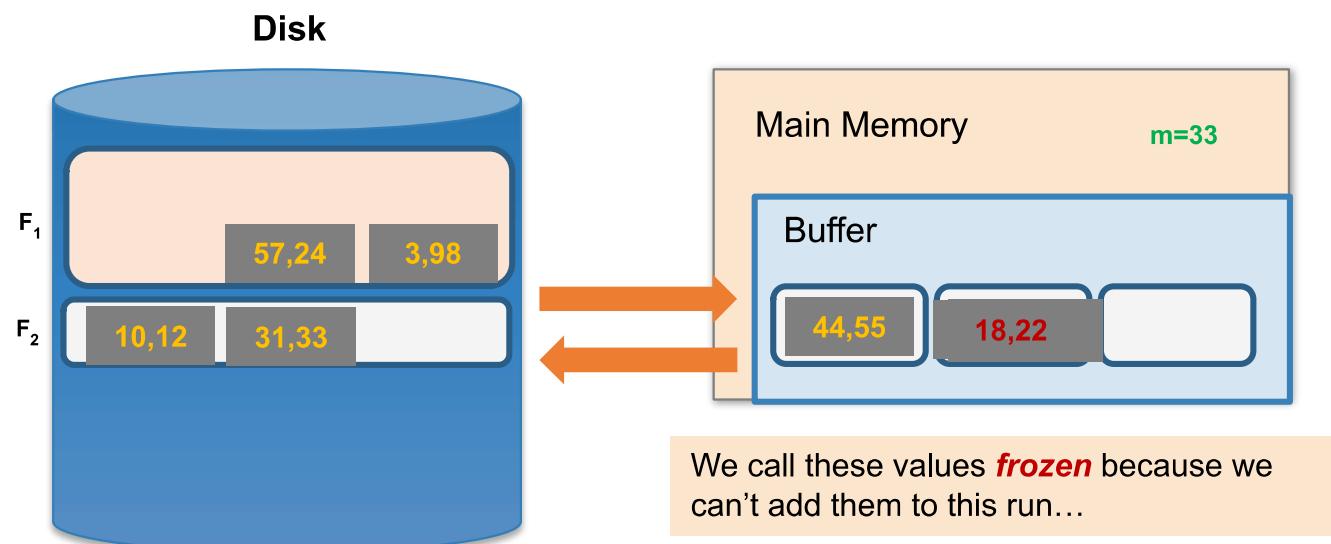
- Next, **repack**, then load another page and continue!



(Notice  $m=33$  – indicate the largest value written to  $F_2$  so far.)

# Repacking Example: 3 page buffer

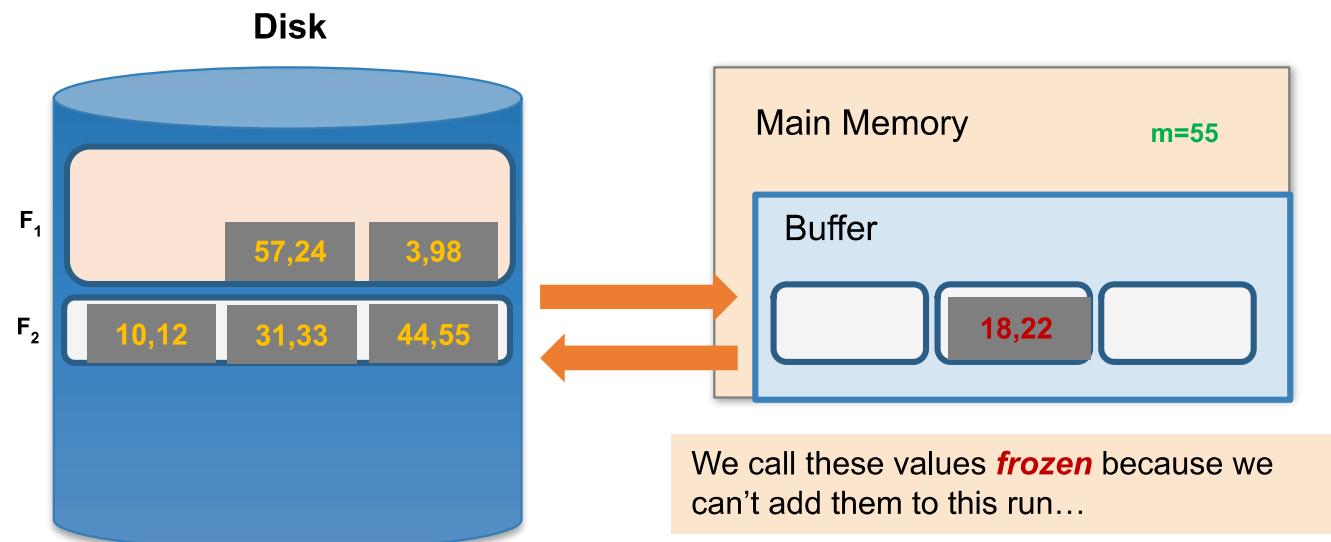
- Now, however, ***the smallest values are less than the largest (last) in the sortrun...***



(Notice can you write 18,22 to F2? No. m=33.)

# Repacking Example: 3 page buffer

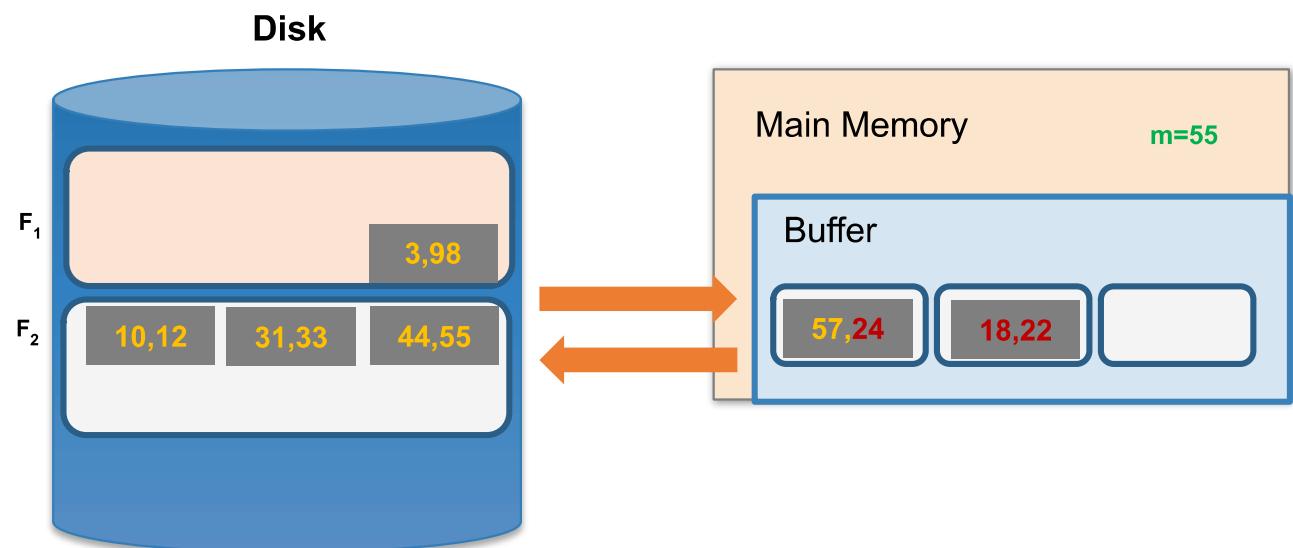
- Now, however, ***the smallest values are less than the largest (last) in the sortrun...***



(Note: you write 44, 55 to F2. Set m = 55)

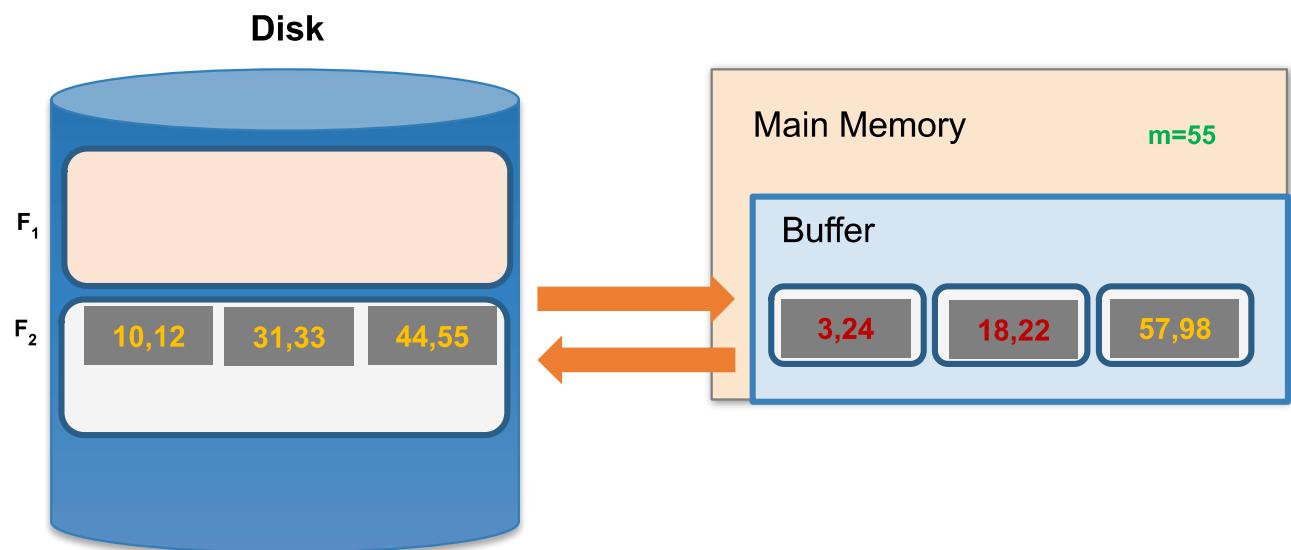
# Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sortrun...***



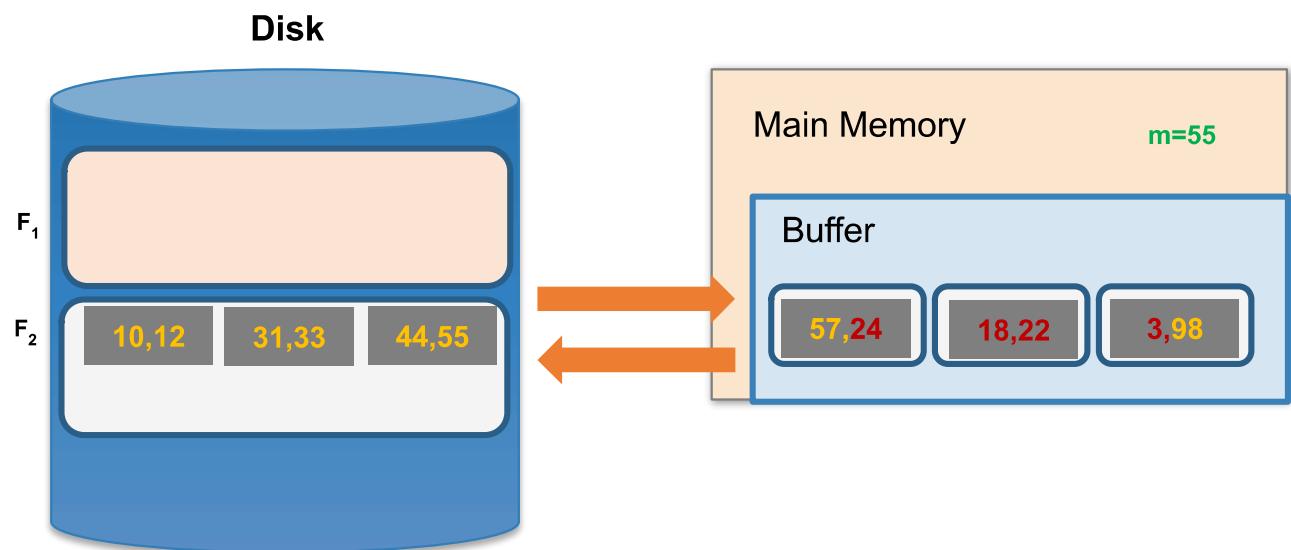
# Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sortrun...***



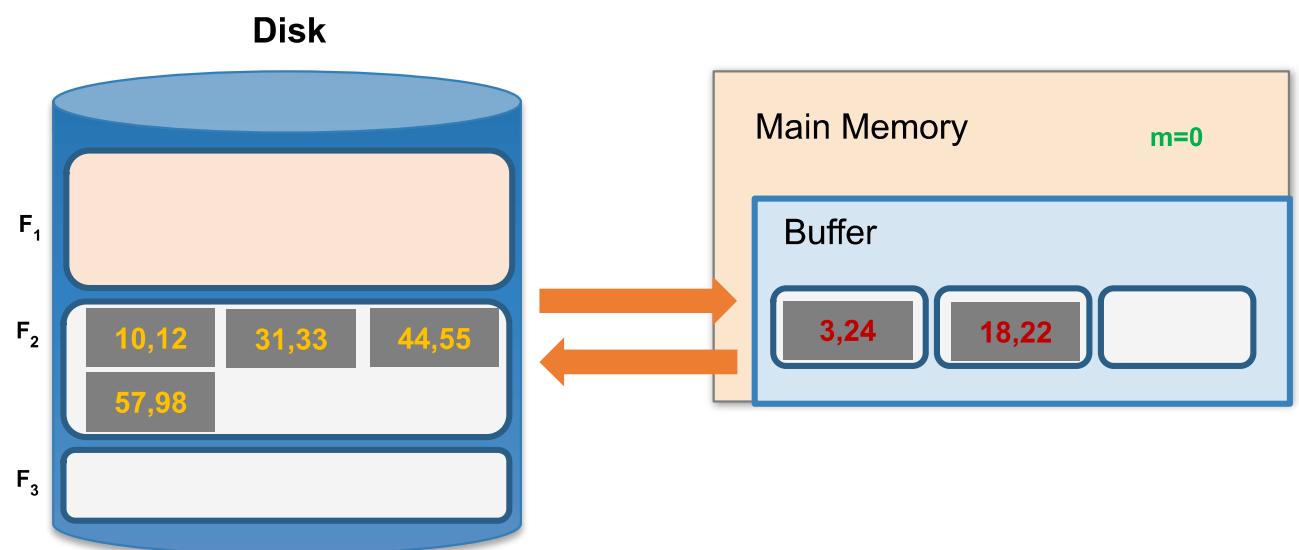
# Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run...***



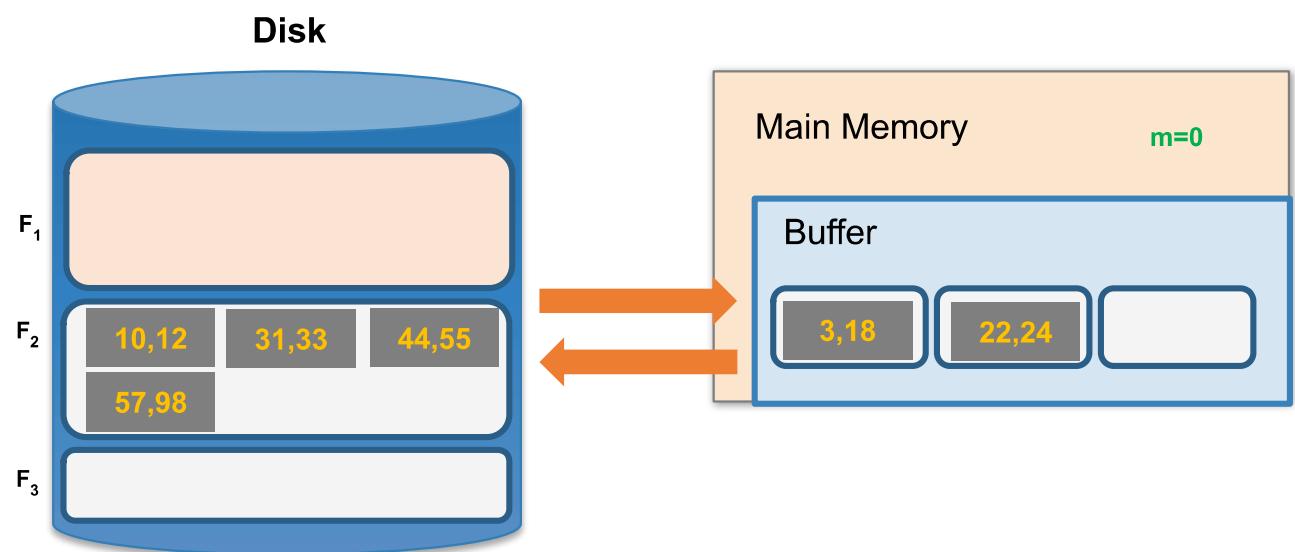
# Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value***, or input file is empty, start new run with the frozen values



# Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value***, or input file is empty, start new run with the frozen values





## Why Repacking helps?

- **Best case:** If input file is sorted → nothing frozen → we get a single SortRun!
- **Worst case:** If input file is reverse sorted → becomes as bad as basic ExternalMergeSort()

⇒ In general, with repacking we do no worse than without it!

# General ExternalMergeSort(): Optimizations

So far...in examples, we used  $B=3$  pages for simplicity

- Today's 64GB RAM machines,  $\mathbf{B = 1024}$  (i.e., 1024 RAM Pages of 64MBs each)

Three optimizations for large  $B$

1. Repacking (last few slides)
2. Increase the length of initial runs
3. B-way merges

IO Cost ~=  
(sort  $N$  pages)

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

*Note: This is MUCH better than  
our basic version ~10 slides back*

34

# 10 TB Sorting Example

Sort 10 TB file with 10 GB of RAM

- I.e., File has 156,250 64MB-Blocks, RAM  $\approx 156$  Pages
- I.e.,  $N = 156,250$ ,  $B = 156$

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

$$\Rightarrow \log_{156} (156250/[2*156]) \approx 1.23$$

$$\Rightarrow \text{ExternalMergeSort() Cost} = 2N (\text{roundUp}[1.24]+1) \approx 6*N \text{ IOs}$$

That's AMAZING!!!

Algorithm sorts BIG files ( $\sim 1000x$  bigger than RAM) with a small constant factor (6x) on data size  
[What if B is  $\sim 1\text{GB}$ ?  $\log_{15.6} (156250/[2*15.6]) \approx 3.1$ . **ExternalMergeSort() Cost**  $\approx 10N$  IOs.  
That is, for 1/10th the B, we're only  $\sim 10/6$  worse off]

## Sorting

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

 $\sim 2N$  $\sim 4N$  $\sim 6N$ 

Sort N pages with B buffer size

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

Sort N pages when  $N \approx B$ 

(because  $(\log_B 0.5) < 0$ )

Sort N pages when  $N \approx 2^*B^2$ 

(because  $(\log_B B) = 1$ )

Sort N pages when  $N \approx 2^*B^3$ 

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?



## Sorting, with insertions?

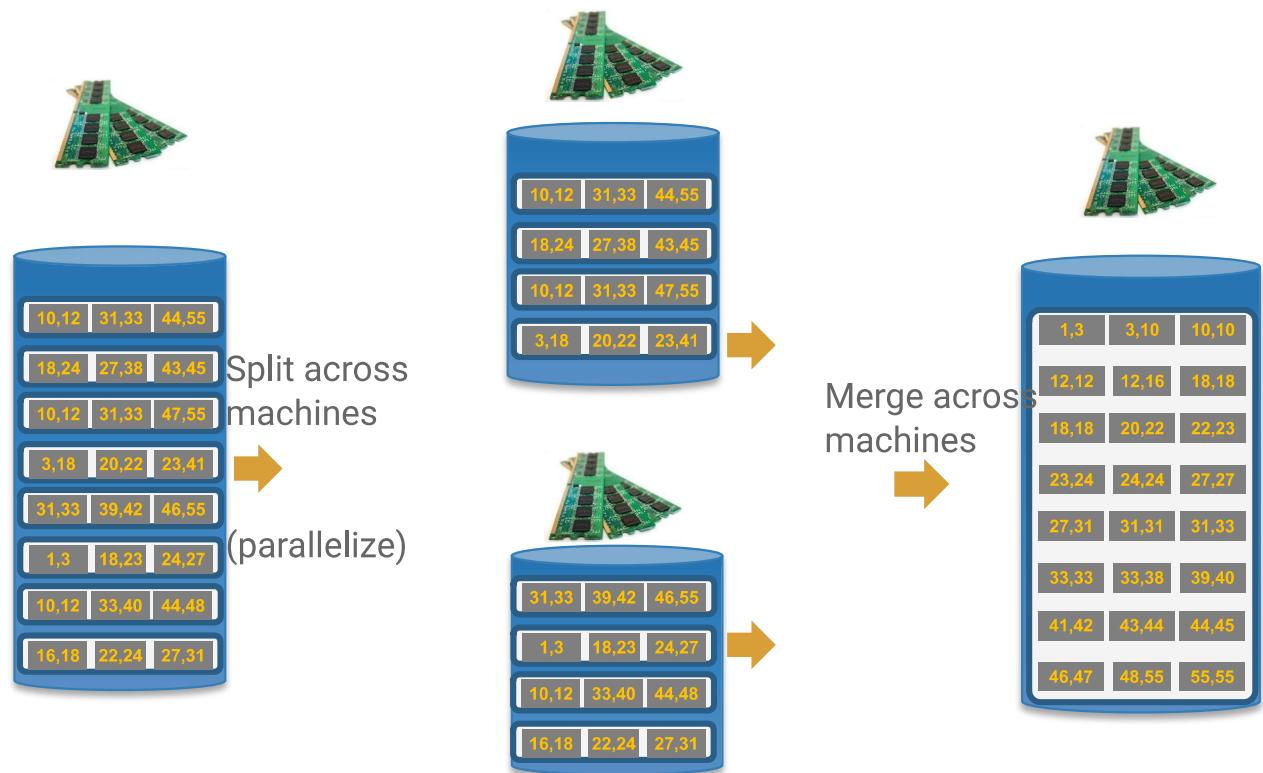


- We would have to potentially shift  $N$  records, requiring  $\sim 2*N/P$  IO (worst case) operations (where  $P = \#$  of records per page)!
  - We could leave some “slack” in the pages...

Could we get faster insertions?  
(next section)

# 37

## Scaling, Speeding Sort (in Cluster)



**ExternalMergeSort**  
locally in each machine  
(in parallel)

Notes

- Use N machines ( $N \geq 2$ )
- Could reuse machines
- Speedup at cost of network bandwidth (especially with current data centers)

# 38

## Big Scale Lego Blocks

## Roadmap



Primary data structures/algorithms

Hashing

Sorting

HashTables  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

HashFunctions  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

HashFunctions  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

BucketSort, QuickSort  
MergeSort

ExternalMerge

ExternalMergeSort  
(also  
ExternalSort)

ExternalMergeSort

## Plan for the day

1. How to sort **10 TB** files with **1 GB** of RAM?

A classic problem in computer science!

2. How to Search over **1 Trillion** items in **< 1 second**, with **Index**?
  - a. (e.g., Youtube's catalog, Amazon's products, etc.)

A classic problem in industry!



# Let's build Indexes

# 2

## Example [Reminder]



**CName\_Index**

CName	Block #
AAPL	.....
AAPL	.....
AAPL	.....
GOOG	.....
GOOG	.....
GOOG	.....
Alibaba	.....
Alibaba	.....

Block #

Company				
CName	Date	Price	Country	
AAPL	Oct1	101.23	USA	→
AAPL	Oct2	102.25	USA	→
AAPL	Oct3	101.6	USA	→
GOOG	Oct1	201.8	USA	→
GOOG	Oct2	201.61	USA	→
GOOG	Oct3	202.13	USA	→
Alibaba	Oct1	407.45	China	→
Alibaba	Oct2	400.23	China	→

**PriceDate\_Index**

Date	Price	Block #
Oct1	101.23	.....
Oct2	102.25	.....
Oct3	101.6	.....
Oct1	201.8	.....
Oct2	201.61	.....
Oct3	202.13	.....
Oct1	407.45	.....
Oct2	400.23	.....

1. Index contains search values + Block #: e.g., DB block number.
  - o In general, "pointer" to where the record is stored (e.g., RAM page, DB block number or even machine + DB block)
  - o Index is conceptually a table. In practice, implemented very efficiently (see how soon)
2. Can have multiple indexes to support multiple search keys

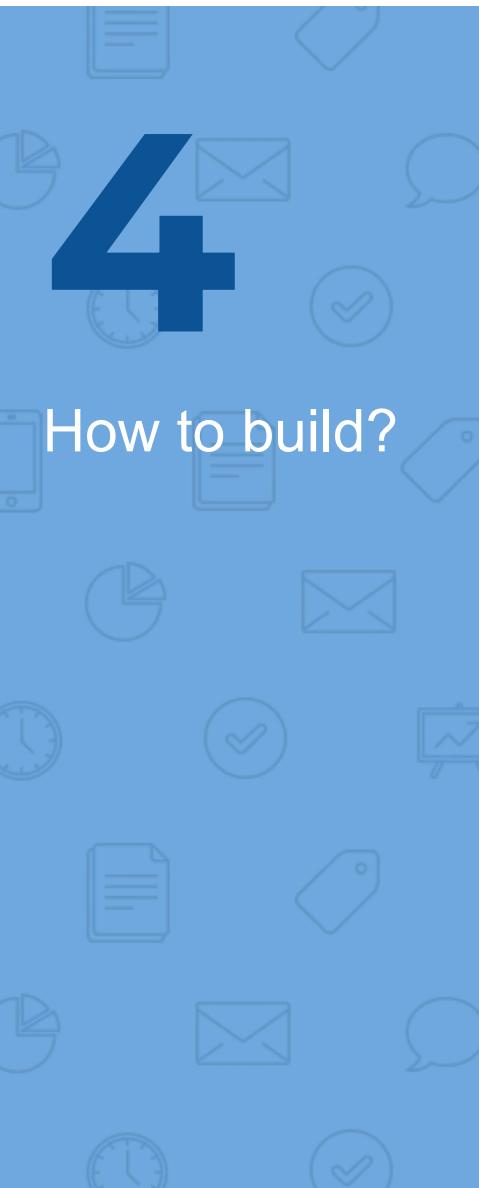
How?



IO Aware

For big files (bigger than RAM), we need efficient data structures that work with HDD/SSD IO systems

- ▷ An ***IO aware*** algorithm! (and data structures)



1. How is data organized?
  - ▷ Is data in Row or Column store?
  - ▷ Is data sorted or not?
2. How do we organize search values?

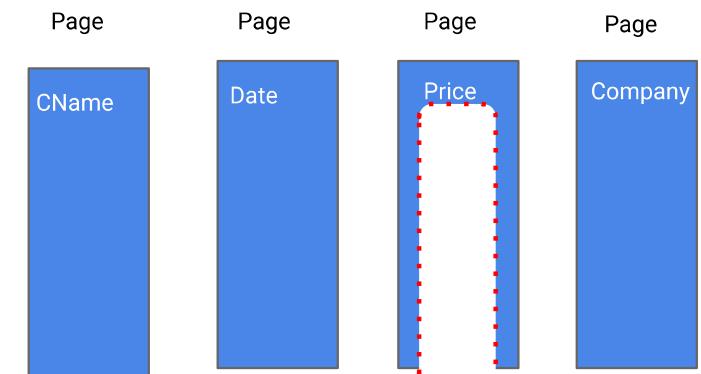
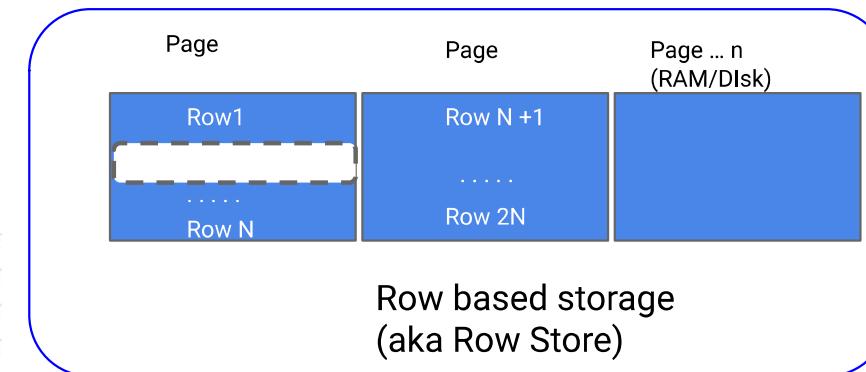
# 5

## Recall Data Layout

Company				
CName	Date	Price	Country	
AAPL	Oct1	101.23	USA	Col3
AAPL	Oct2	102.25	USA	
AAPL	Oct3	101.6	USA	
GOOG	Oct1	201.8	USA	
GOOG	Oct2	201.61	USA	
GOOG	Oct3	202.13	USA	
Alibaba	Oct1	407.45	China	
Alibaba	Oct2	400.23	China	

Company(CName, StockPrice, Date, Country)

Logical Table



Column based storage  
(aka Column Store)

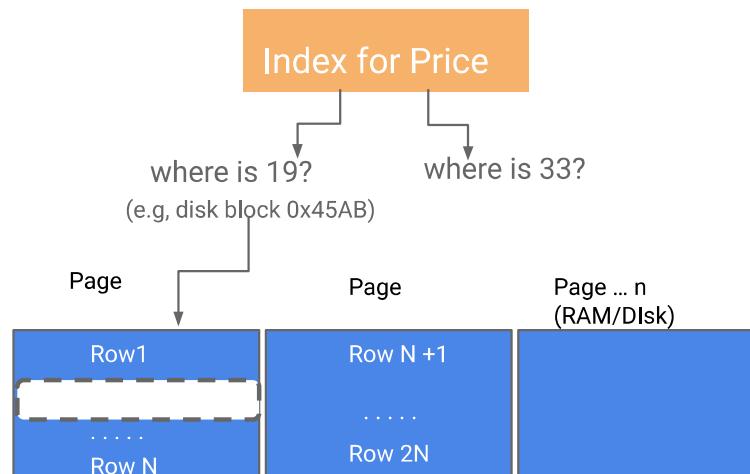
# 6

## Index on row store

Note: **search key** does not mean it's unique. It's what are you searching for. (vs Primary KEYs in SQL that are unique)

### Query: Search for cname with specific price?

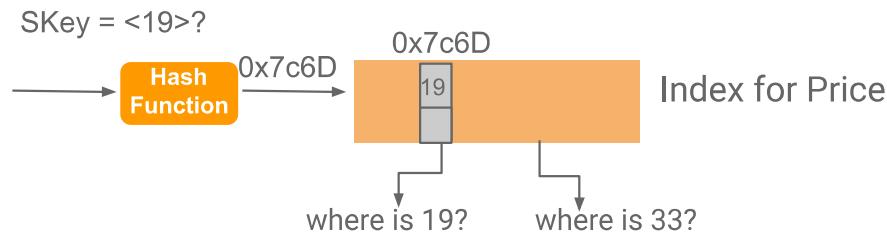
- ⇒ If 'price' is an indexed column, query will be fast.
- ⇒ 'Price' is search key(SKey). Values in price column are search values. (e.g. 'price' == 19?)



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

# 7

## Our 1st Hashing index

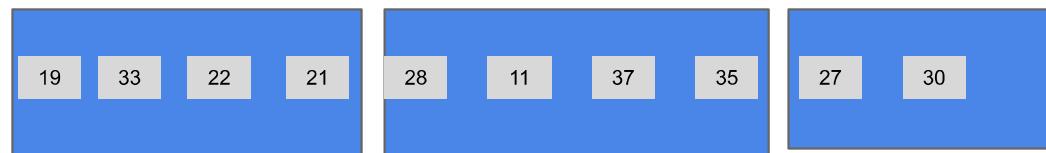


Goal of Index: where is location of each SValue

For simplicity, we'll only show the SValues for Price index

Example row: <'goog', price=19, date=Oct 1, ...> as <price=19> or <19>

Option1:  
Unsorted



If unsorted, maintain locations of all SValues in each block.

Option2:  
Sorted



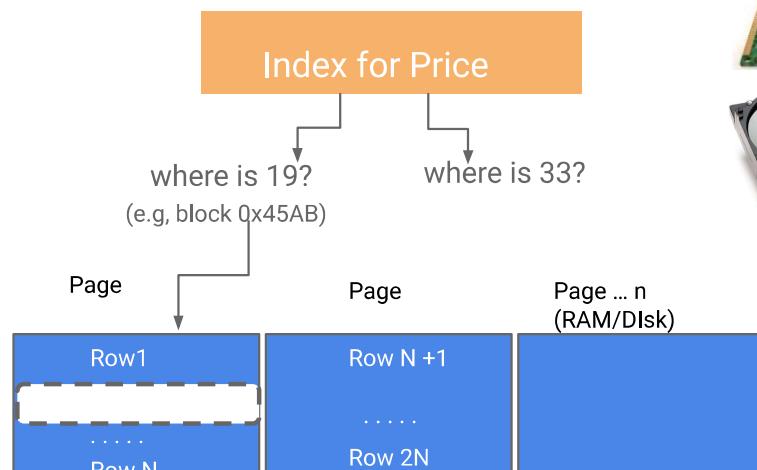
If sorted, can do better. Maintain locations only of smallest SValue in each block. (e.g., 11, 27, 33)

How it works in practice?

1. Schema designer picks a column to keep data sorted by (e.g., price). Index for that column is cheap.
2. For other columns, index will be bigger (e.g., CName)

# Index on row store

Query: Search for cname with specific price?



How do we store Index?  
⇒ Idea: Index is just a table (rows/columns). Same ideas

- Store in pages
- Persist on disk
- Page into RAM buffer

If Index fits in RAM?

- Lookups are fast

If Index does not fit in RAM?

- Could page at random
- Can we organize index pages better? (e.g. Index the index)

# 9



## Index Types

- Hash Tables
  - IO-aware hashing (e.g., *linear* or *extendible hashing*)
- B-Trees (*covered next*)
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will look at a variant called B+ Trees*

These data structures  
are “IO aware”

**Real difference between structures:**  
costs of ops *determines which index you pick and why*

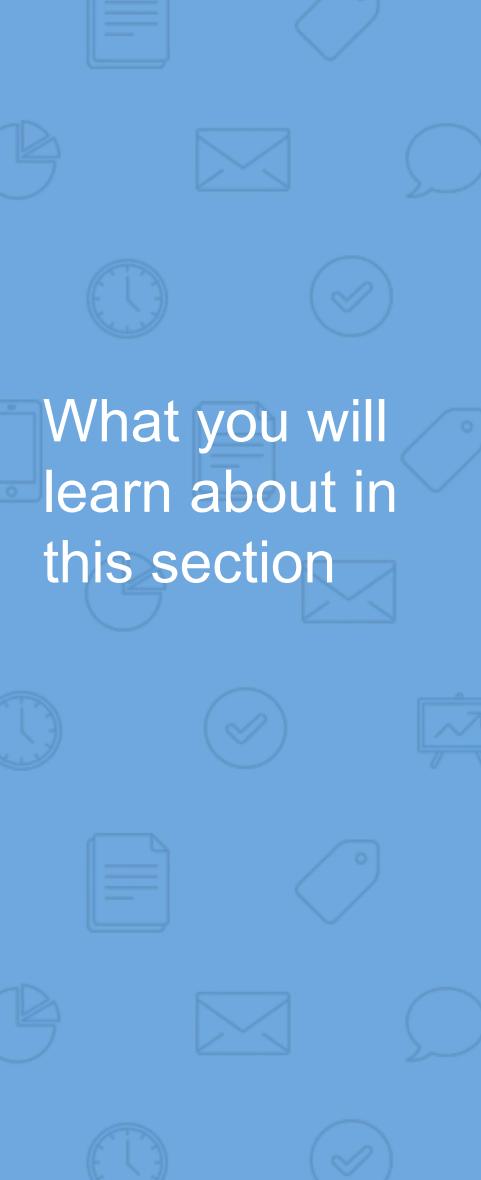


# B+ Trees

# Idea in B+ Trees

Search trees that are IO aware

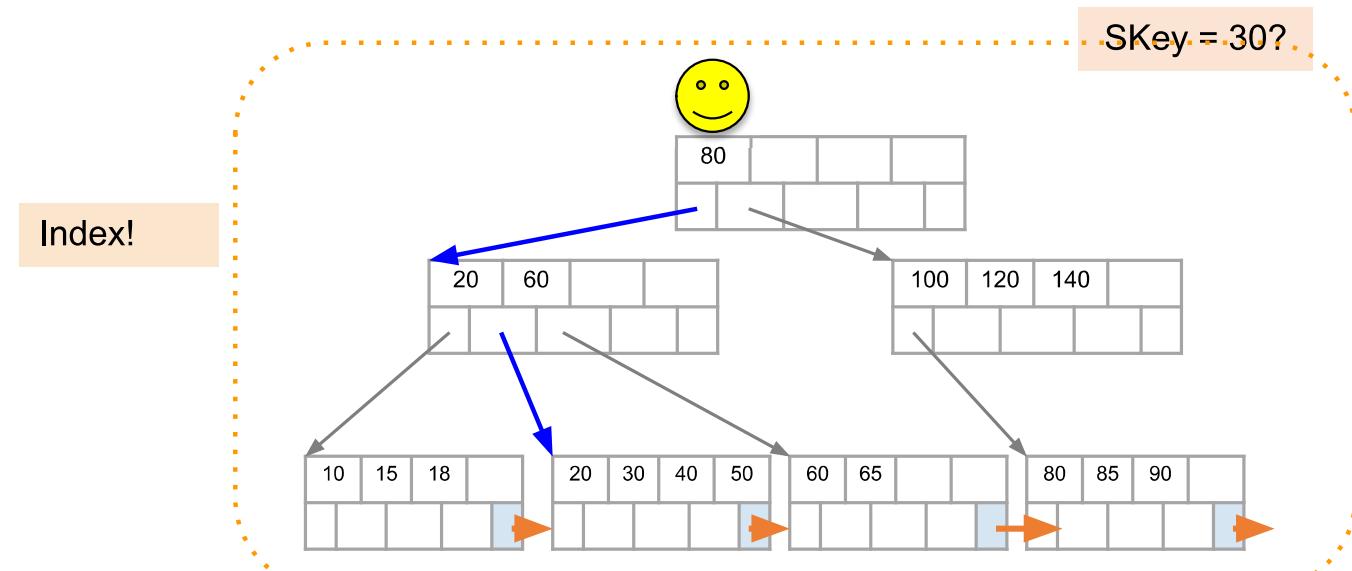
- Store each tree-node in 1 page
- Balanced, height adjusted tree
- Make leaves into a linked list (for range queries)



What you will learn about in this section

1. B+ Trees: Basics
2. B+ Trees: Design & Cost
3. Clustered Indexes

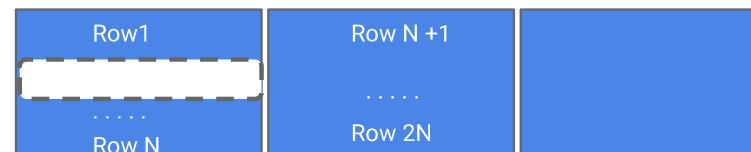
# B+ Tree Exact Search



Note: the pointers at the leaf level will be to the actual data records (rows).

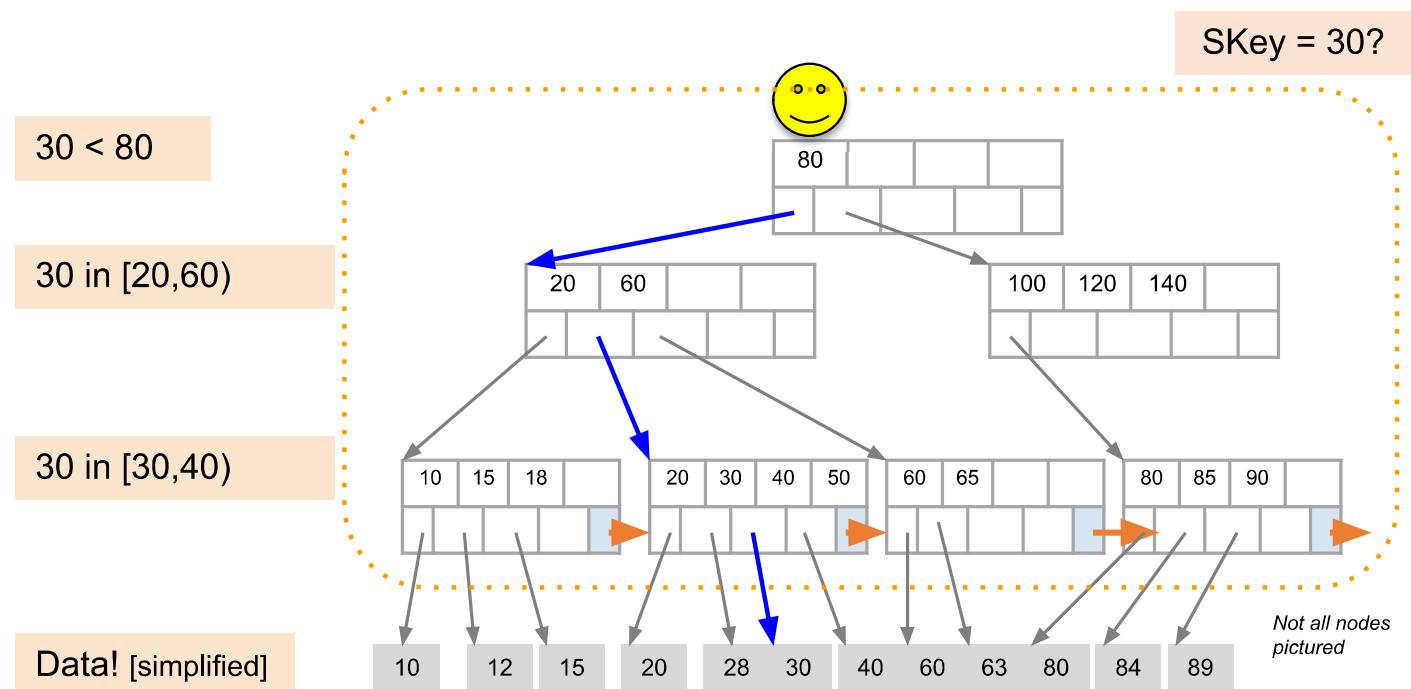
We truncate and only display SValues for simplicity (as before)...

Data!



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

# B+ Tree Exact Search



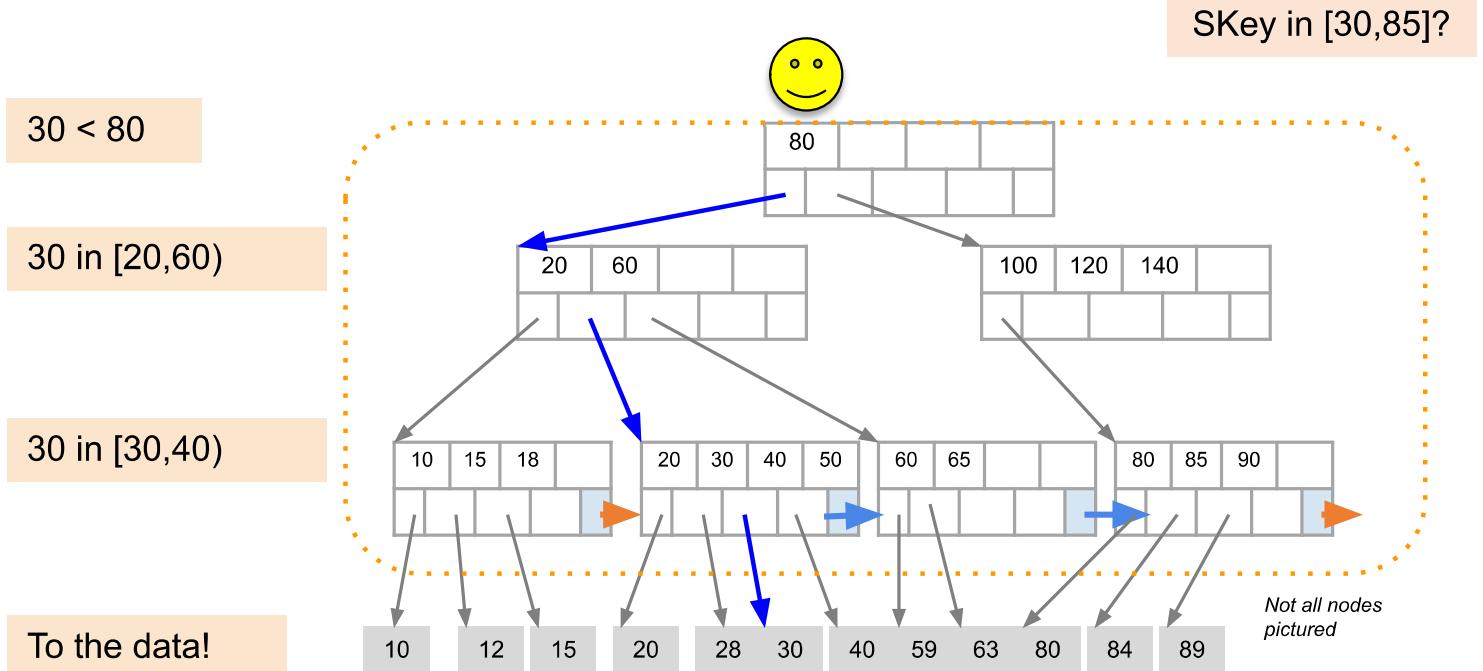
# Searching a B+ Tree

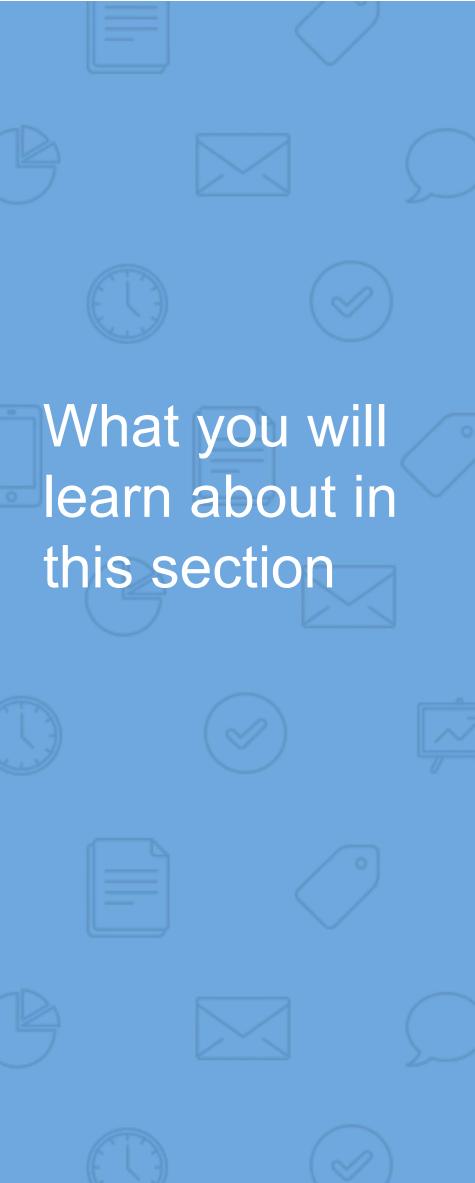
- For exact SKey values:
  - Start at the root
  - Proceed down, to the leaf
- For range queries:
  - As above
  - *Then sequential traversal*

```
SELECT cname  
FROM Company  
WHERE price = 25
```

```
SELECT cname  
FROM Company  
WHERE 20 <= price  
AND price <= 30
```

# B+ Tree Range Search





What you will learn about in this section

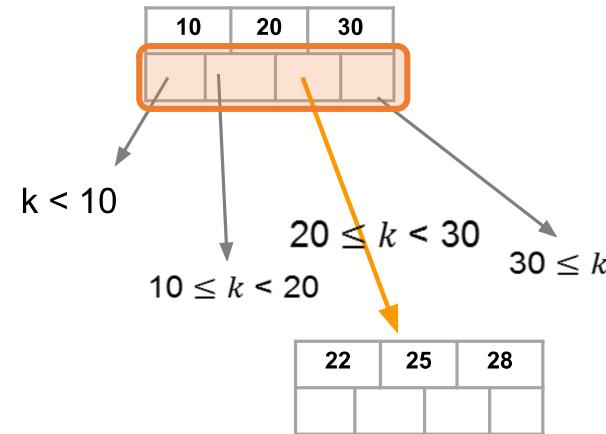
1. B+ Trees: Basics
2. B+ Trees: Design & Cost
  - ▷ How many search values per page?
  - ▷ How many levels in tree?
3. Clustered Indexes



# B+ Tree Basics -- Root and non-leaf nodes

Parameter  $f = \text{fanout}$   
(e.g.  $f = 3$ )

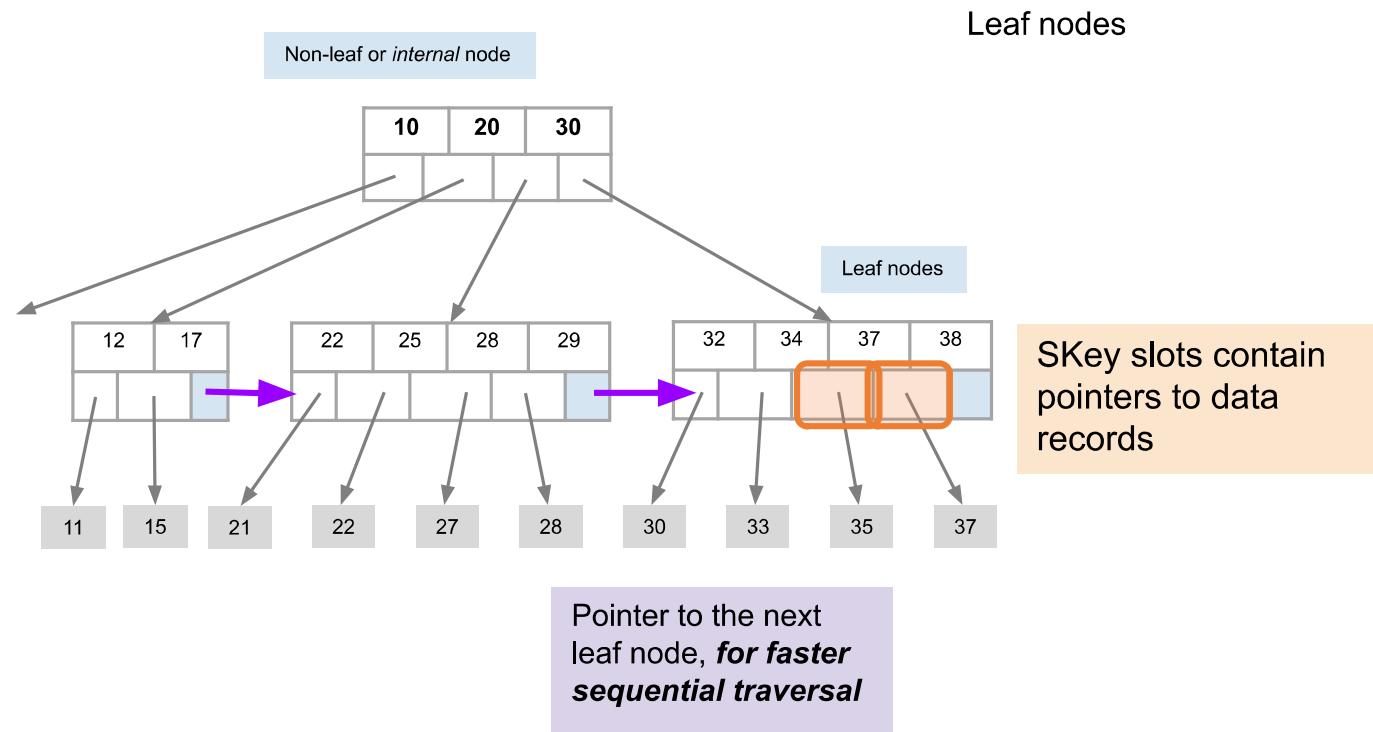
Each non-leaf node has  
 $\leq f$  SKeys



11    15    21    22    27    28    30    33    35    37



## B+ Tree Basics -- Leaf nodes





# Costs of B+ trees



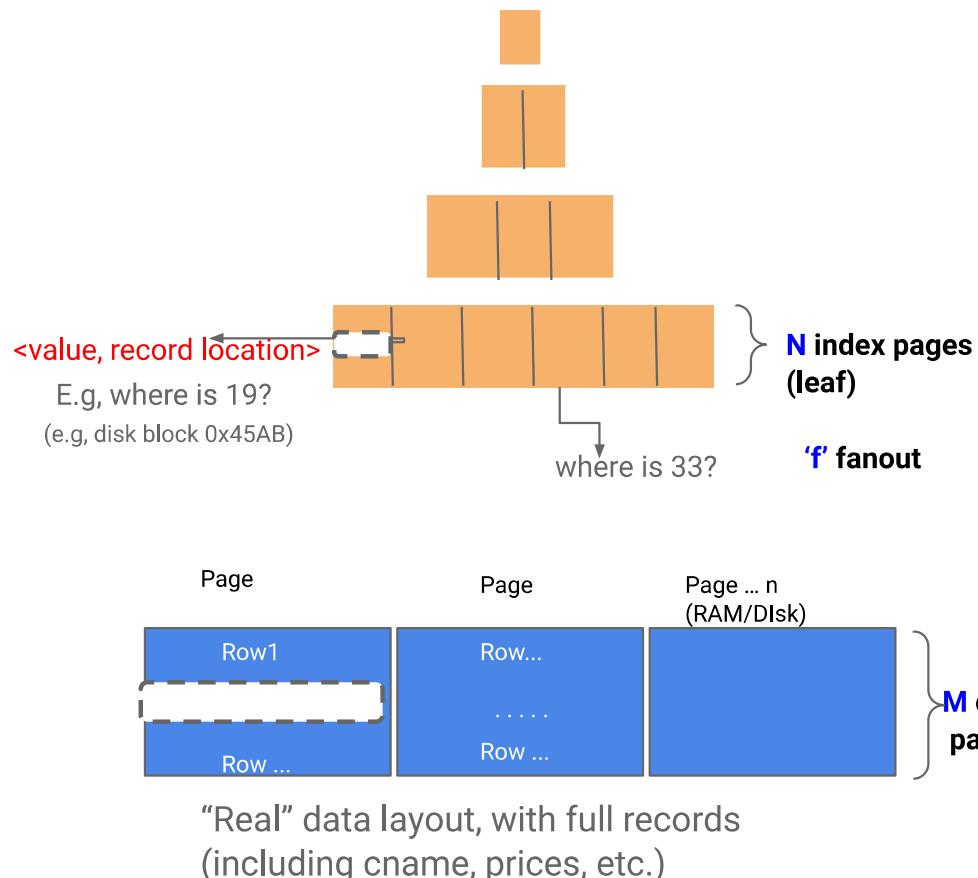
# B+ Tree: High Fanout = Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high fanout**
- Hence the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most/all of B+ Tree in RAM!

The fanout is defined as the number of pointers to child nodes coming out of a [leaf] node

*Note that fanout is dynamic- we'll often assume it's constant just to come up with approximate eqns!*

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



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

Let us build an **index for an SKey** (e.g., CName) in data

- **NumSKeys** = number of SValues for that SKey (e.g., 5000 cnames)
- **SKeySize** = size of SKey (e.g. 4 bytes)
- **PointerSize** = size of pointer (e.g. 8 bytes)

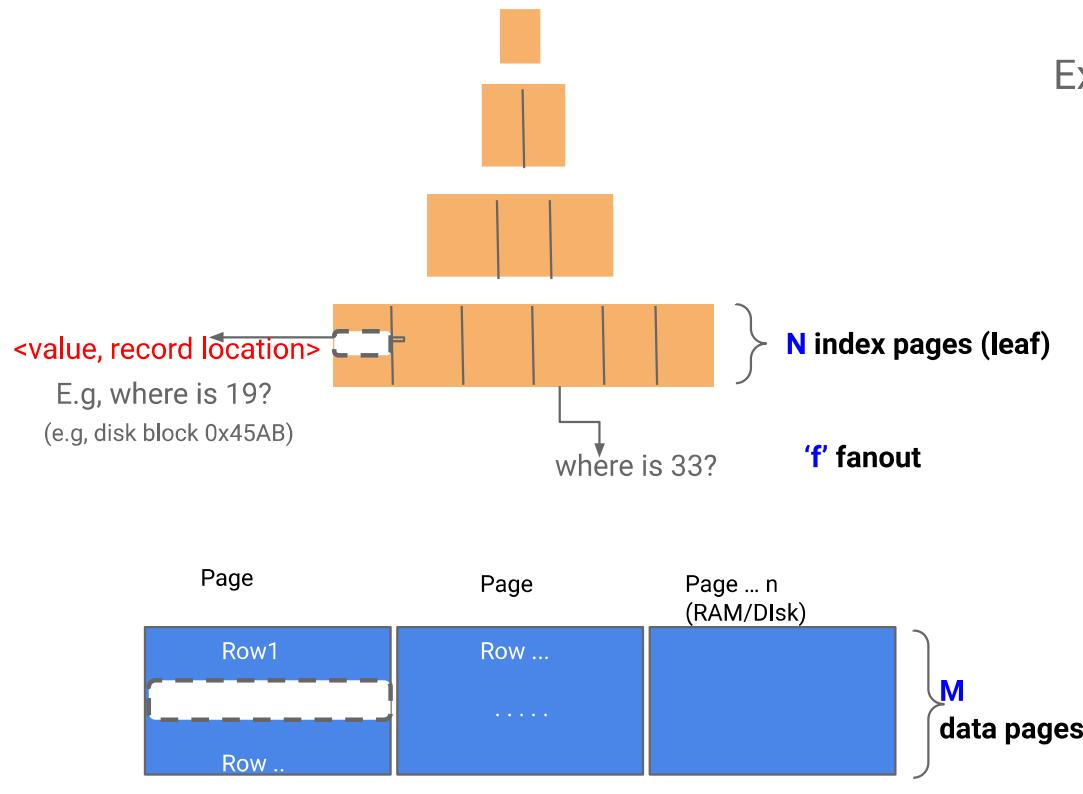
For B+ tree

- $f$  = fanout (we'll assume it is constant for simplicity...)
- $h$  = height of tree (e.g., 1, 2, ...)
- $N$  = number of index pages

Simplified cost model

$$f \leq \frac{\text{PageSize}}{\text{SKeySize} + \text{PointerSize}} // \text{ Fit upto } f \text{ (SKey, Pointer)}$$
$$f^h \geq \text{NumSKeys} // \text{ Leaf nodes should point to all SKeys}$$

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



Example 1: Search Amazon's Product Catalog

- SKeySize = 8 bytes,
- PointerSize = 8 bytes
- NumSKeys = 1 Trillion

PageS ize	64KB	64MB
f	~4000	~4 million
h	~4	~2

**AMAZING:** Worst-case for 1 Trillion SKeys

- 2 IOs (for 64MB) for index
- 1 IO for data page



# Simple Cost Model for Range Search

- To do range search, we just follow the horizontal pointers
- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each *page* of the results- we phrase this as “Cost(OUT)”



# Fast Insertions & Self-Balancing

- We won't go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:
  - ~ Same cost as exact search
  - *Self-balancing*: B+ Tree remains balanced (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!  
*However, can become bottleneck if many insertions (if fill-factor slack is used up...)*

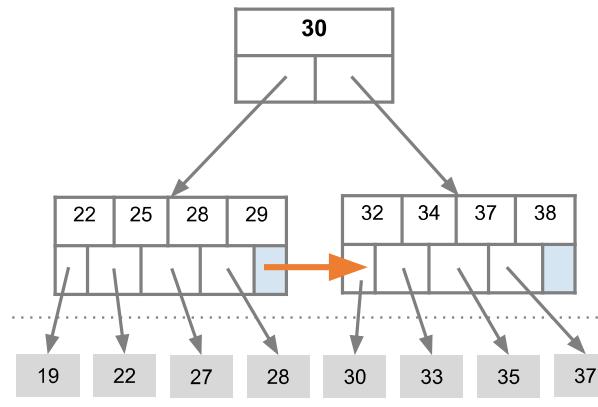


What you will  
learn about in  
this section

1. B+ Trees: Basics
2. B+ Trees: Design & Cost
3. Clustered Indexes



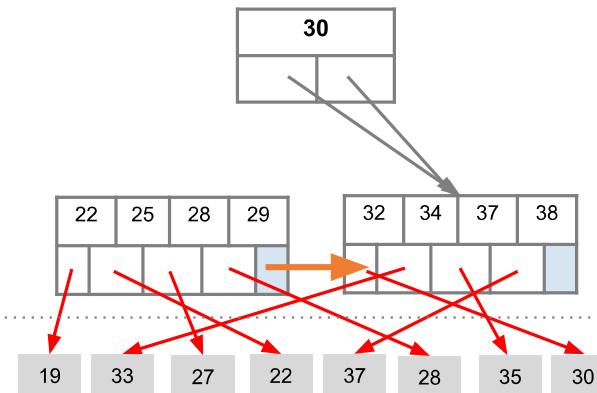
# Clustered vs. Unclustered Index



Clustered  
(sorted data)

Index Entries

Data Records



Unclustered

An index is clustered if the underlying data is ordered in the same way as the index's data entries.



# Clustered vs. Unclustered Index

- Recall that sequential disk block IO is much faster than random IO
- For exact search, no difference between clustered / unclustered
- For range search over R values: difference between
  - [a] 1 random IO + R sequential IO and [b] R random IO:
    - A random IO costs ~ 10ms (sequential much much faster)
    - For R = 100,000 records- difference between ~10ms and ~17min!



# Summary

- We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
  - An *IO aware* algorithm!
- We create indexes over tables in order to support *fast* (*exact and range*) search and *insertion* over *multiple search keys*
- B+ Trees are one index data structure which support very fast exact and range search & insertion via *high fanout*
  - *Clustered* vs. *unclustered* makes a big difference for range queries too



# Let's build Indexes

# 2

## Example [Reminder]



**CName\_Index**

CName	Block #
AAPL	.....
AAPL	.....
AAPL	.....
GOOG	.....
GOOG	.....
GOOG	.....
Alibaba	.....
Alibaba	.....

Block #

Company				
CName	Date	Price	Country	
AAPL	Oct1	101.23	USA	→
AAPL	Oct2	102.25	USA	→
AAPL	Oct3	101.6	USA	→
GOOG	Oct1	201.8	USA	→
GOOG	Oct2	201.61	USA	→
GOOG	Oct3	202.13	USA	→
Alibaba	Oct1	407.45	China	→
Alibaba	Oct2	400.23	China	→

**PriceDate\_Index**

Date	Price	Block #
Oct1	101.23	.....
Oct2	102.25	.....
Oct3	101.6	.....
Oct1	201.8	.....
Oct2	201.61	.....
Oct3	202.13	.....
Oct1	407.45	.....
Oct2	400.23	.....

1. Index contains search values + Block #: e.g., DB block number.
  - o In general, "pointer" to where the record is stored (e.g., RAM page, DB block number or even machine + DB block)
  - o Index is conceptually a table. In practice, implemented very efficiently (see how soon)
2. Can have multiple indexes to support multiple search keys

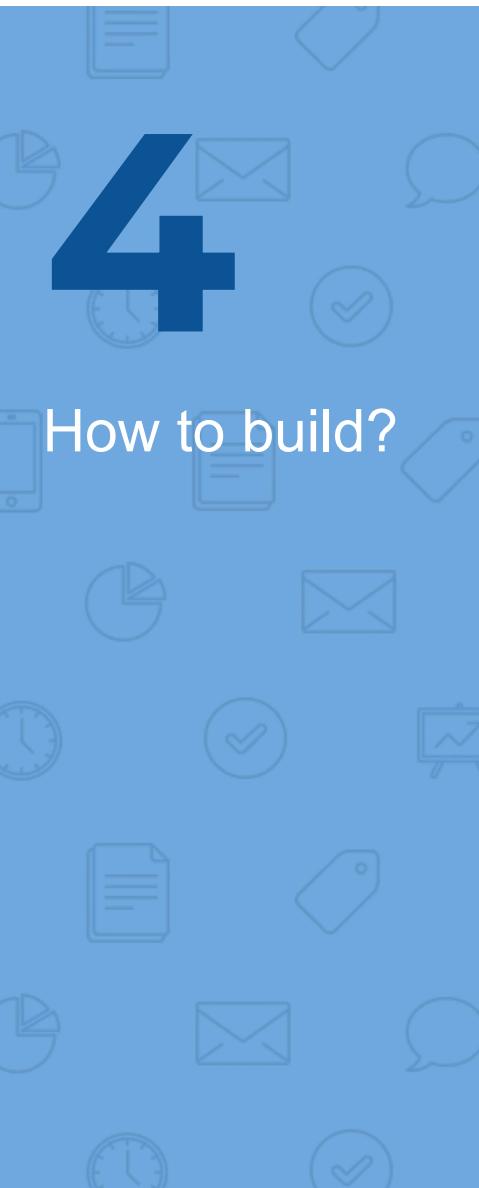
How?



IO Aware

For big files (bigger than RAM), we need efficient data structures that work with HDD/SSD IO systems

- ▷ An ***IO aware*** algorithm! (and data structures)



1. How is data organized?
  - ▷ Is data in Row or Column store?
  - ▷ Is data sorted or not?
2. How do we organize search values?

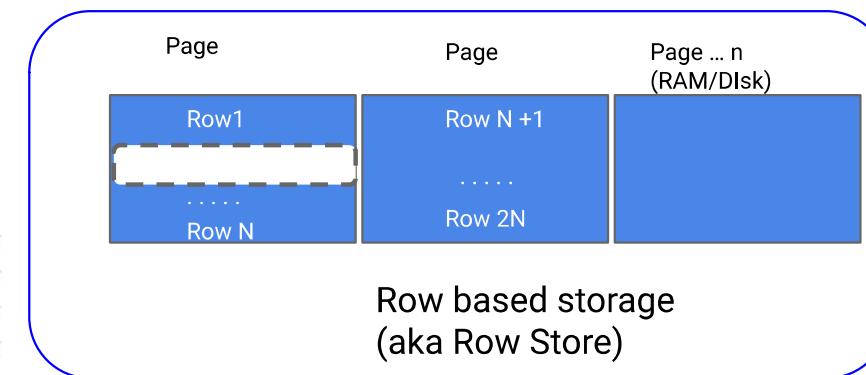
# 5

## Recall Data Layout

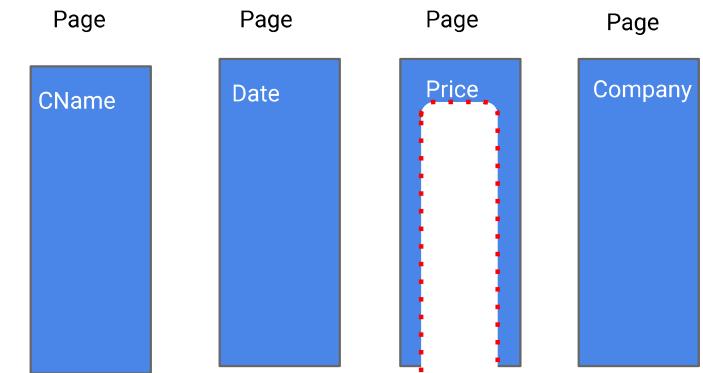
Company				
CName	Date	Price	Country	
AAPL	Oct1	101.23	USA	Col3
AAPL	Oct2	102.25	USA	
AAPL	Oct3	101.6	USA	
GOOG	Oct1	201.8	USA	
GOOG	Oct2	201.61	USA	
GOOG	Oct3	202.13	USA	
Alibaba	Oct1	407.45	China	
Alibaba	Oct2	400.23	China	

Company(CName, StockPrice, Date, Country)

Logical Table



Row based storage  
(aka Row Store)



Column based storage  
(aka Column Store)

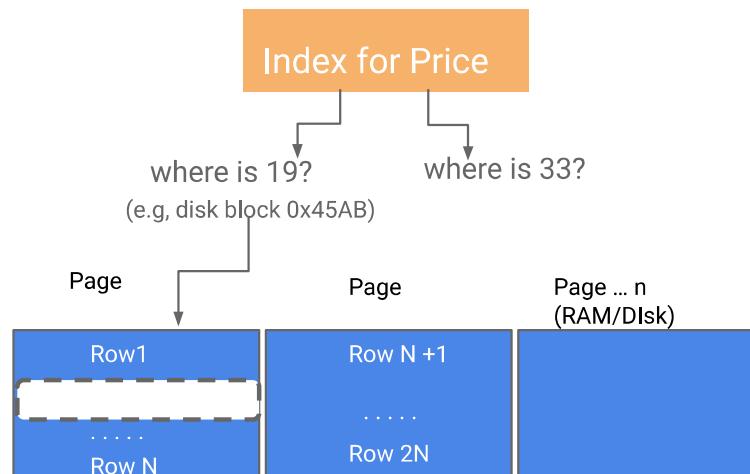
# 6

## Index on row store

Note: **search key** does not mean it's unique. It's what are you searching for. (vs Primary KEYs in SQL that are unique)

### Query: Search for cname with specific price?

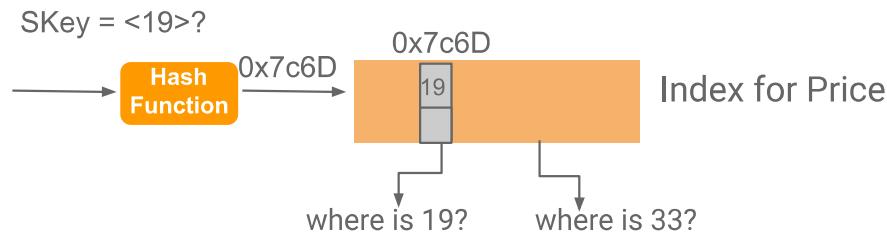
- ⇒ If 'price' is an indexed column, query will be fast.
- ⇒ 'Price' is search key(SKey). Values in price column are search values. (e.g. 'price' == 19?)



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

# 7

## Our 1st Hashing index

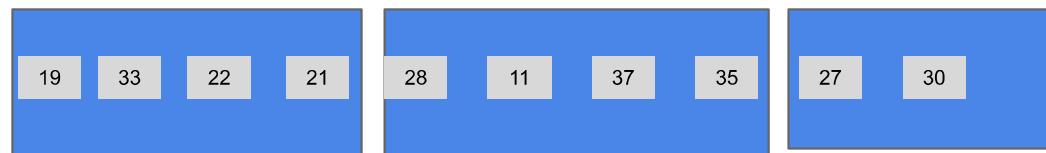


Goal of Index: where is location of each SValue

For simplicity, we'll only show the SValues for Price index

Example row: <'goog', price=19, date=Oct 1, ...> as <price=19> or <19>

Option1:  
Unsorted



If unsorted, maintain locations of all SValues in each block.

Option2:  
Sorted



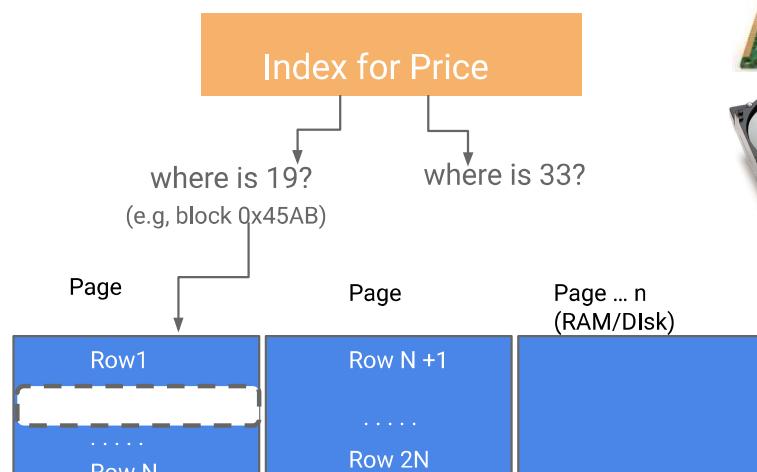
If sorted, can do better. Maintain locations only of smallest SValue in each block. (e.g., 11, 27, 33)

How it works in practice?

1. Schema designer picks a column to keep data sorted by (e.g., price). Index for that column is cheap.
2. For other columns, index will be bigger (e.g., CName)

# Index on row store

Query: Search for cname with specific price?



How do we store Index?  
⇒ Idea: Index is just a table (rows/columns). Same ideas

- Store in pages
- Persist on disk
- Page into RAM buffer

If Index fits in RAM?

- Lookups are fast

If Index does not fit in RAM?

- Could page at random
- Can we organize index pages better? (e.g. Index the index)

# 9



## Index Types

- Hash Tables
  - IO-aware hashing (e.g., *linear* or *extendible hashing*)
- B-Trees (*covered next*)
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will look at a variant called B+ Trees*

These data structures  
are “IO aware”

**Real difference between structures:**  
costs of ops *determines which index you pick and why*



# B+ Trees

# Idea in B+ Trees

Search trees that are IO aware

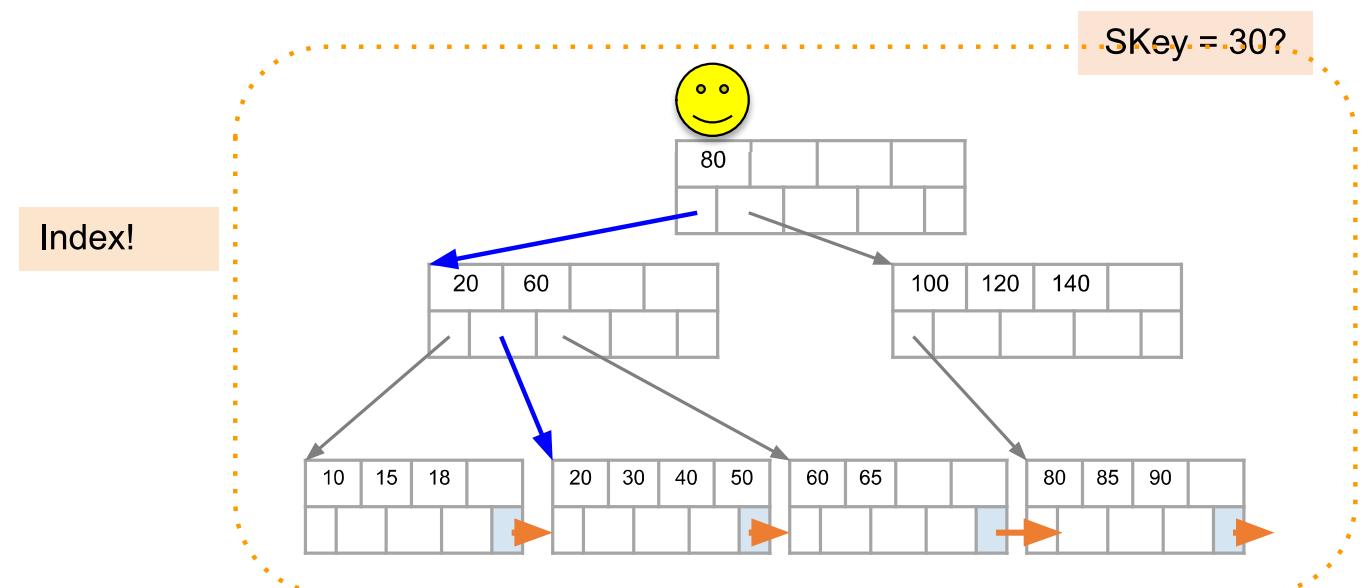
- Store each tree-node in 1 page
- Balanced, height adjusted tree
- Make leaves into a linked list (for range queries)



What you will  
learn about in  
this section

1. B+ Trees: Basics
2. B+ Trees: Design & Cost
3. Clustered Indexes

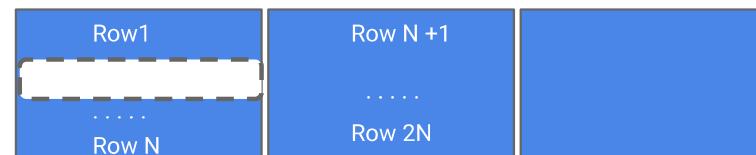
# B+ Tree Exact Search



Note: the pointers at the leaf level will be to the actual data records (rows).

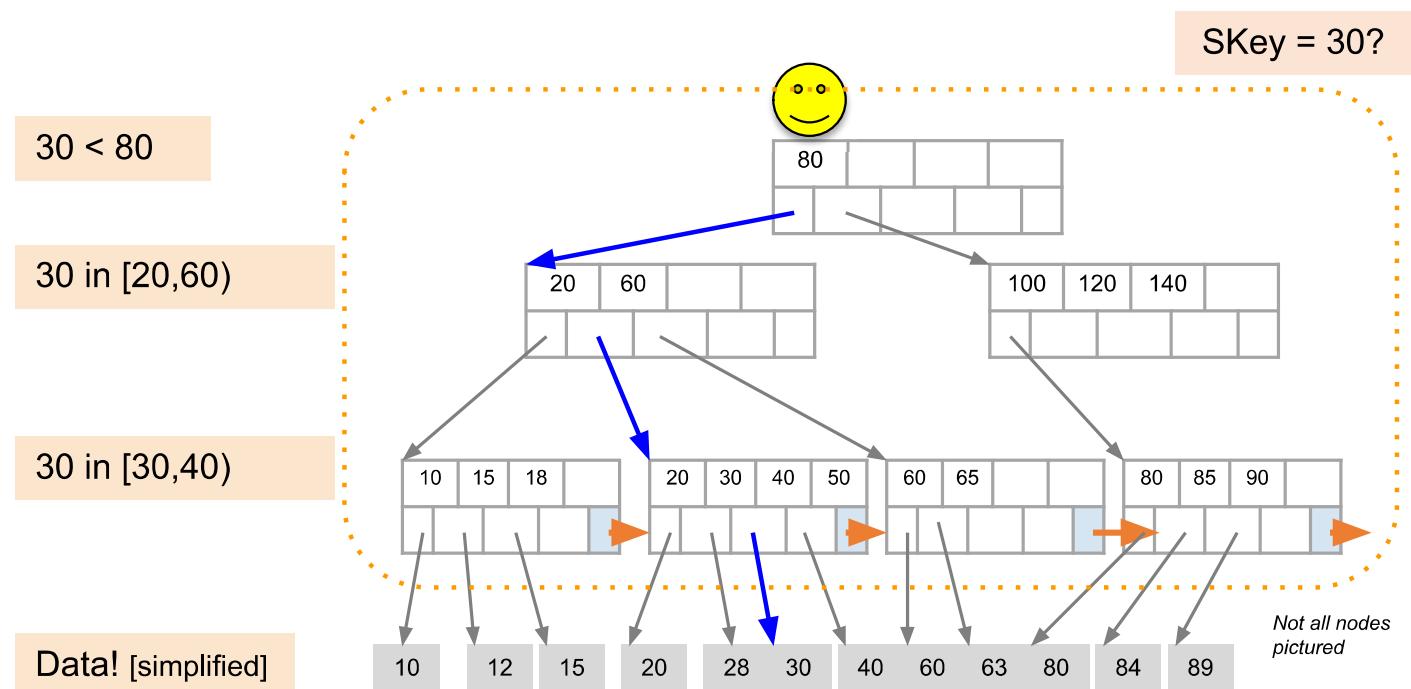
We truncate and only display SValues for simplicity (as before)...

Data!



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

# B+ Tree Exact Search



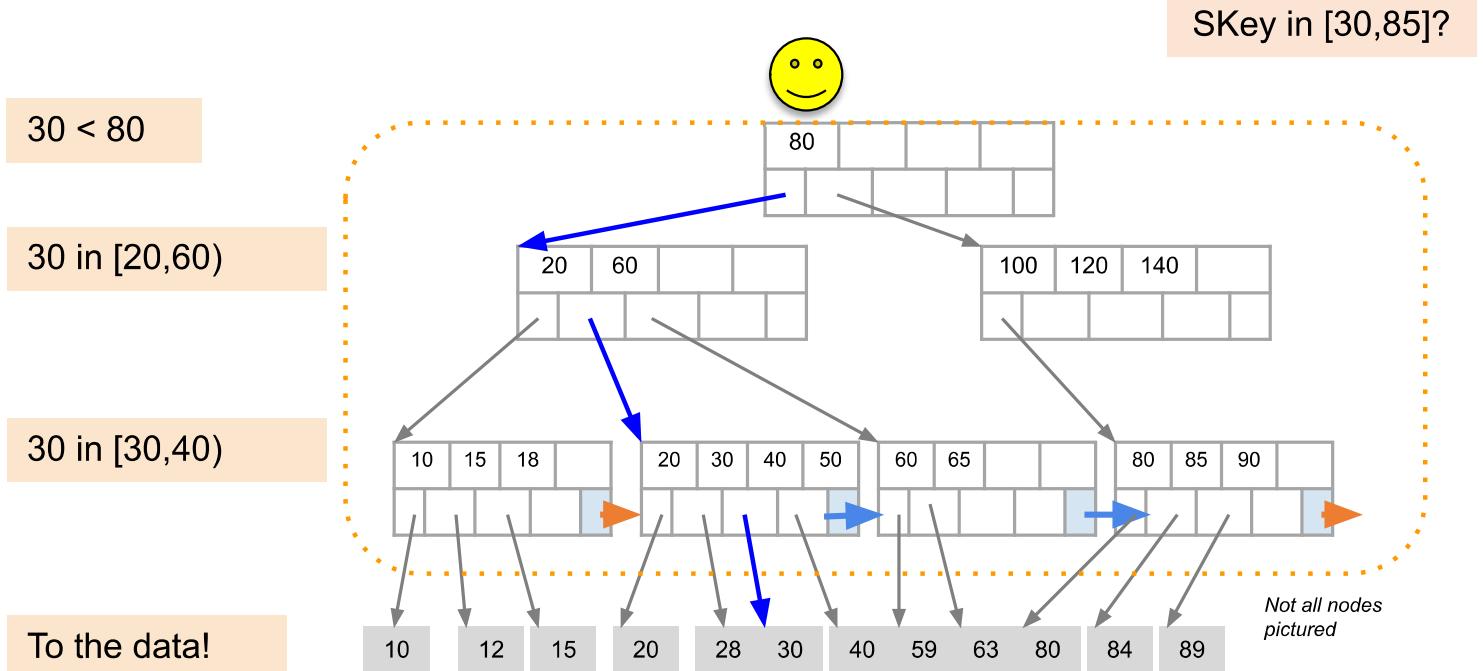
# Searching a B+ Tree

- For exact SKey values:
  - Start at the root
  - Proceed down, to the leaf
- For range queries:
  - As above
  - *Then sequential traversal*

```
SELECT cname  
FROM Company  
WHERE price = 25
```

```
SELECT cname  
FROM Company  
WHERE 20 <= price  
AND price <= 30
```

# B+ Tree Range Search





What you will learn about in this section

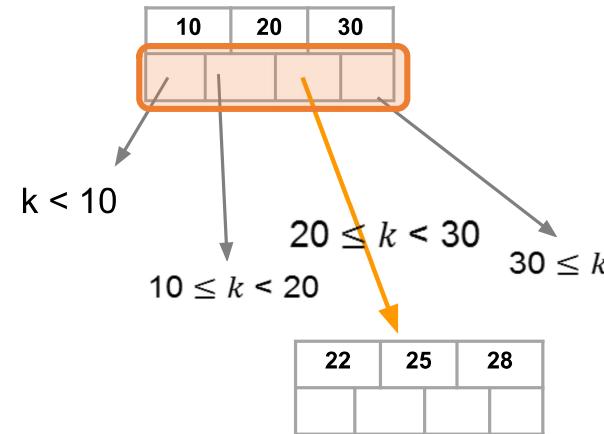
1. B+ Trees: Basics
2. B+ Trees: Design & Cost
  - ▷ How many search values per page?
  - ▷ How many levels in tree?
3. Clustered Indexes



# B+ Tree Basics -- Root and non-leaf nodes

Parameter  $f = \text{fanout}$   
(e.g.  $f = 3$ )

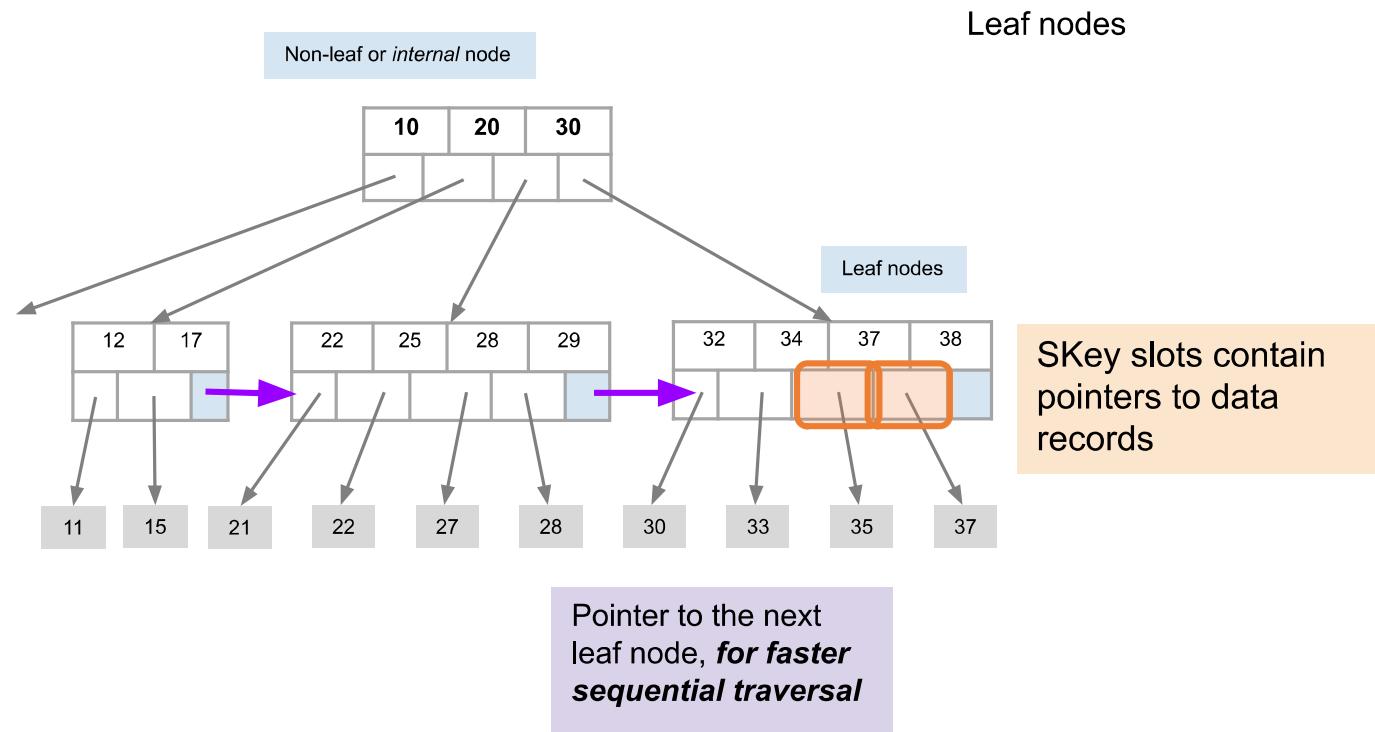
Each non-leaf node has  
 $\leq f$  SKeys



11    15    21    22    27    28    30    33    35    37



## B+ Tree Basics -- Leaf nodes





# Costs of B+ trees



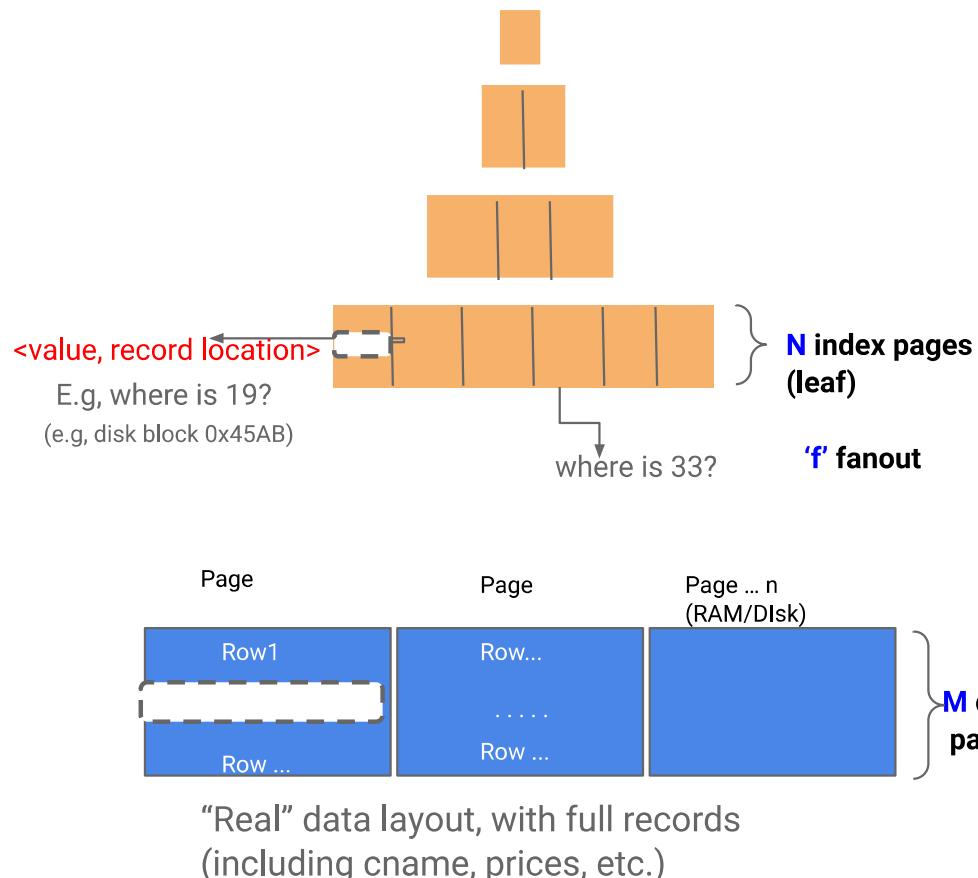
# B+ Tree: High Fanout = Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high fanout**
- Hence the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most/all of B+ Tree in RAM!

The fanout is defined as the number of pointers to child nodes coming out of a [leaf] node

*Note that fanout is dynamic- we'll often assume it's constant just to come up with approximate eqns!*

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



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

Let us build an **index for an SKey** (e.g., CName) in data

- **NumSKeys** = number of SValues for that SKey (e.g., 5000 cnames)
- **SKeySize** = size of SKey (e.g. 4 bytes)
- **PointerSize** = size of pointer (e.g. 8 bytes)

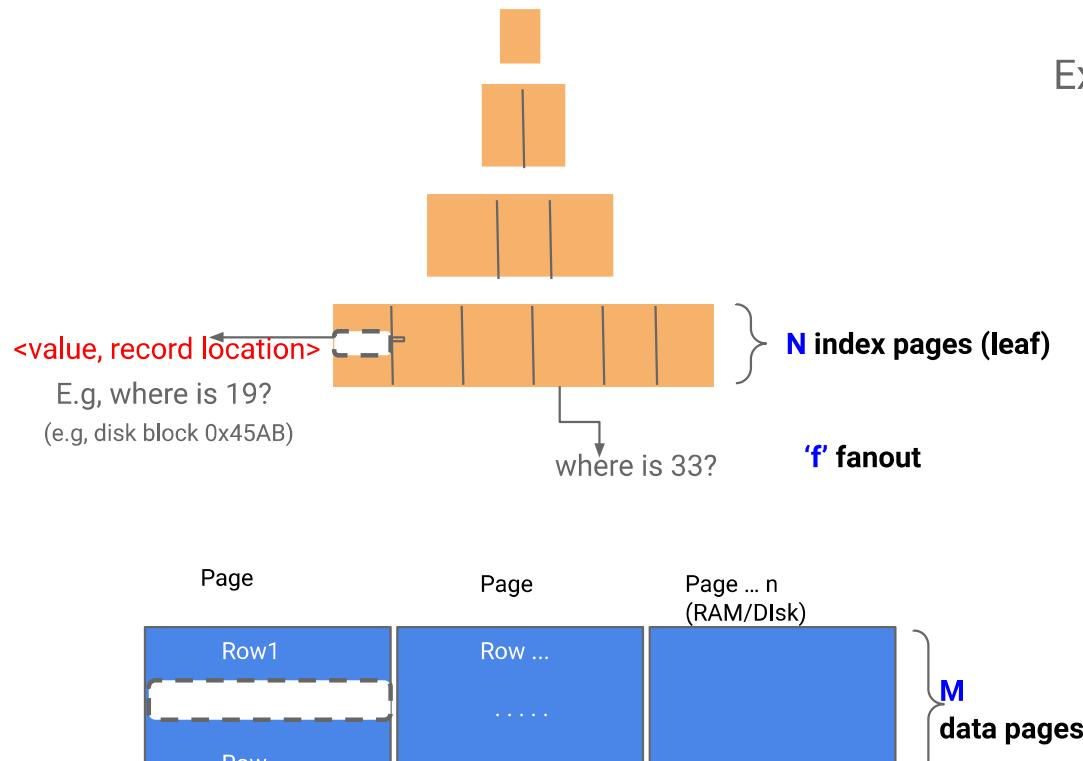
For B+ tree

- $f$  = fanout (*we'll assume it is constant for simplicity...*)
- $h$  = height of tree (e.g., 1, 2, ...)
- $N$  = number of index pages

Simplified cost model

$$f \leq \frac{\text{PageSize}}{\text{SKeySize} + \text{PointerSize}} // \text{ Fit upto } f \text{ (SKey, Pointer)}$$
$$f^h \geq \text{NumSKeys} // \text{ Leaf nodes should point to all SKeys}$$

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



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

Example 1: Search Amazon's Product Catalog

- SKeySize = 8 bytes,
- PointerSize = 8 bytes
- NumSKeys = 1 Trillion

PageS ize	64KB	64MB
f	~4000	~4 million
h	~4	~2

**AMAZING:** Worst-case for 1 Trillion SKeys

- 2 IOs (for 64MB) for index
- 1 IO for data page



# Simple Cost Model for Range Search

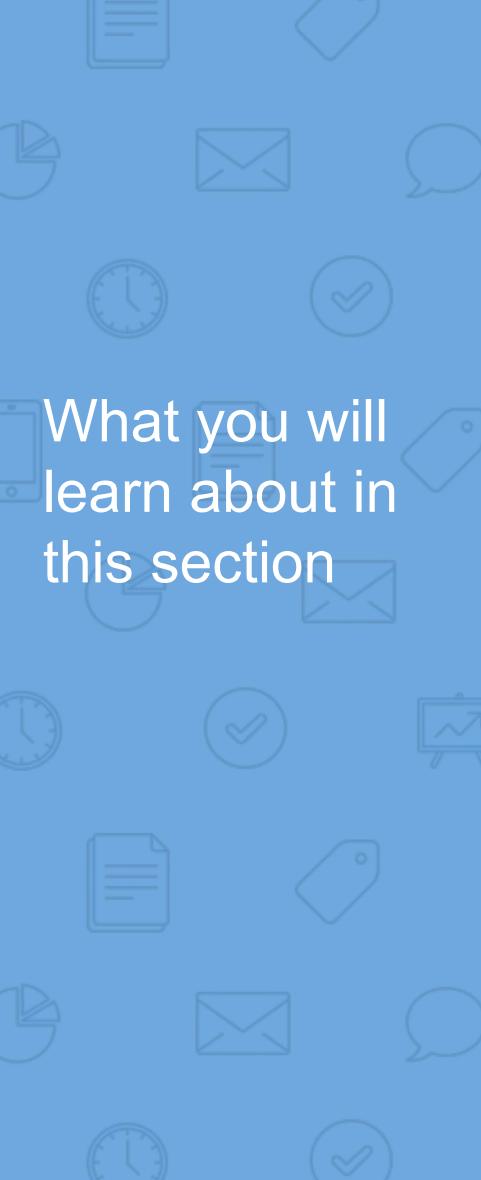
- To do range search, we just follow the horizontal pointers
- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each *page* of the results- we phrase this as “Cost(OUT)”



# Fast Insertions & Self-Balancing

- We won't go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:
  - ~ Same cost as exact search
  - *Self-balancing*: B+ Tree remains balanced (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!  
*However, can become bottleneck if many insertions (if fill-factor slack is used up...)*

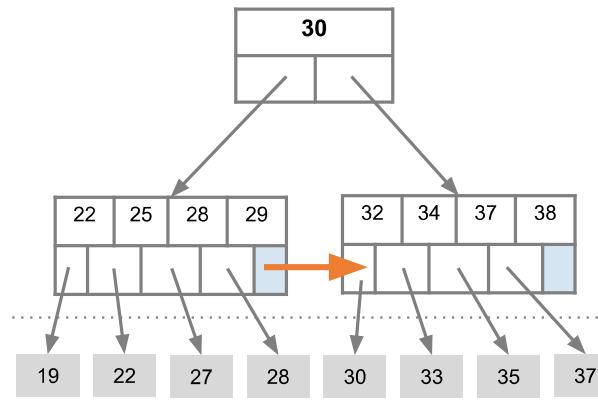


What you will  
learn about in  
this section

1. B+ Trees: Basics
2. B+ Trees: Design & Cost
3. Clustered Indexes



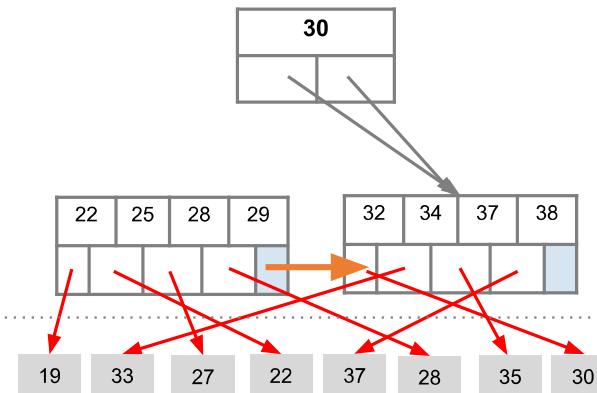
# Clustered vs. Unclustered Index



Clustered  
(sorted data)

Index Entries

Data Records



Unclustered

An index is clustered if the underlying data is ordered in the same way as the index's data entries.



# Clustered vs. Unclustered Index

- Recall that sequential disk block IO is much faster than random IO
- For exact search, no difference between clustered / unclustered
- For range search over R values: difference between
  - [a] 1 random IO + R sequential IO and [b] R random IO:
    - A random IO costs ~ 10ms (sequential much much faster)
    - For R = 100,000 records- difference between ~10ms and ~17min!



# Summary

- We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
  - An *IO aware* algorithm!
- We create indexes over tables in order to support *fast* (*exact and range*) search and *insertion* over *multiple search keys*
- B+ Trees are one index data structure which support very fast exact and range search & insertion via *high fanout*
  - *Clustered* vs. *unclustered* makes a big difference for range queries too



## OCTOBER 2022

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	HW2 <sup>19</sup> out	HW2 <sup>20</sup> section		22
23	24	25	HW2 <sup>26</sup> due	Review	27	28
30	31					29

## NOVEMBER 2022

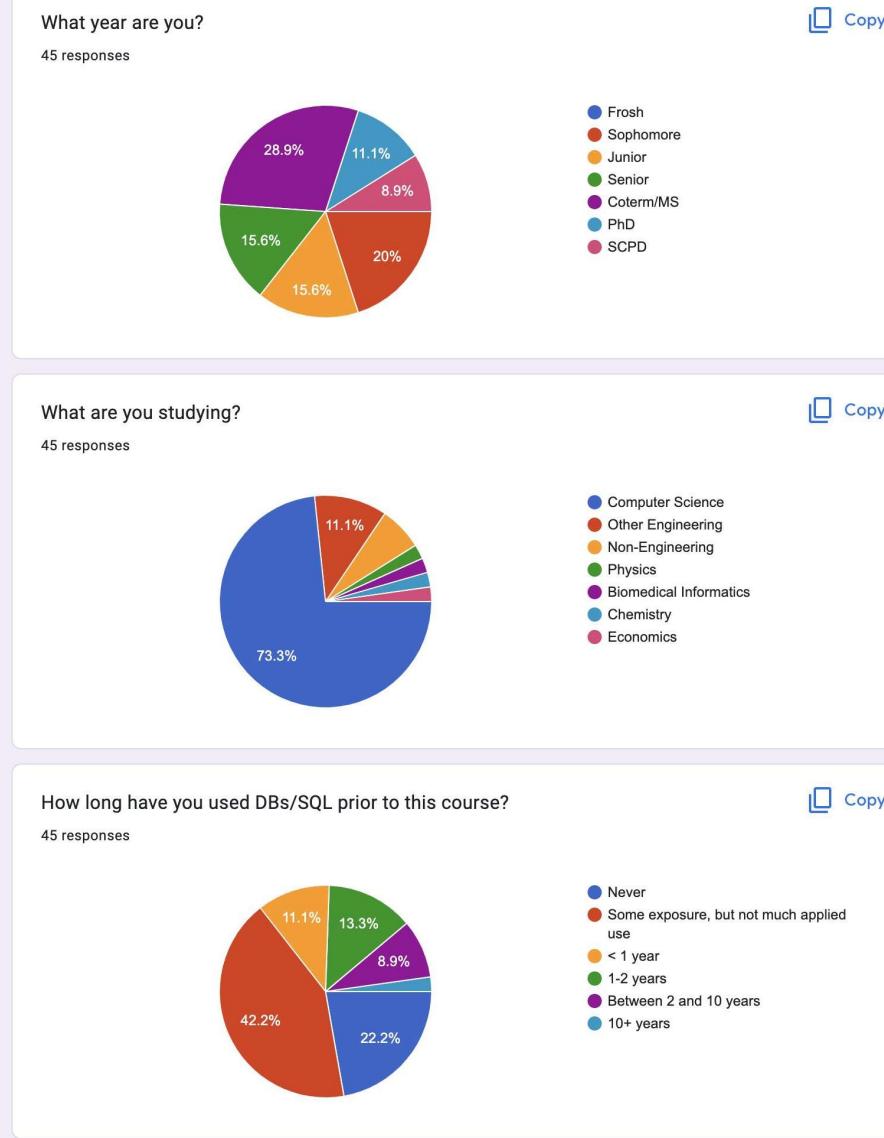
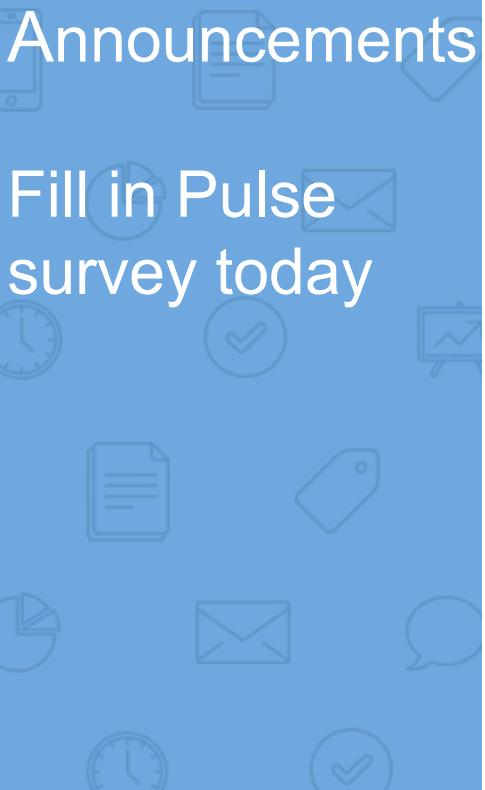
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
			1 <u>Midterm</u>	2	3	4
5						
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

1.

### Nov 1st Midterm (dates/details on cs145 website)

- a. Material covered till **Oct 20th** lecture (Oct 25th lecture tested in Finals, not in Midterm)
- b. Exam format: ~5 questions (SQL, Systems, materials in 1st 8 lectures)
- c. Similar questions to HW1, HW2, Project1 (sql). I.e., problems are the best prep. +1 prep-test out ~Oct 27
- d. **HW2 (Oct 19) - due Oct 26th HW2** (no late days) 11:59 pm due
  - i. Shorter than HW1 (moved some problems to HW3)
  - ii. Oct 21 Section for HW2
  - iii. Oct 27 HW2 11:59 pm solutions released
  - iv. For OAE students who need an extra day, please plan ahead now
- e. Oct 27th Test prep in class

2.



## Wide class mix

- 85%+ CS+ Engg
- 60% Junior-> CoTerm
- Good CS exposure, but no intermediate classes yet

Need more help on RAM Algorithms +OS concepts? (~15%). Goto OH with questions. E.g., if are

- non-CS?
- pre-Junior?

If you find the material basic (~10%)

- Use Project3
- CS245 a better fit?
- Stop by my OH

# Where we are

## Week 3

- Learnt basics of IO-aware algorithms and indexes

## Week 4

- Problem 1: How to search Amazon's product catalog in < 1sec?
- Problem 2: JOIN queries are slow and expensive.
  - How do we speed up by **1000x?** (Today)
  - How do we speed up by another **100x?** (Thursday)

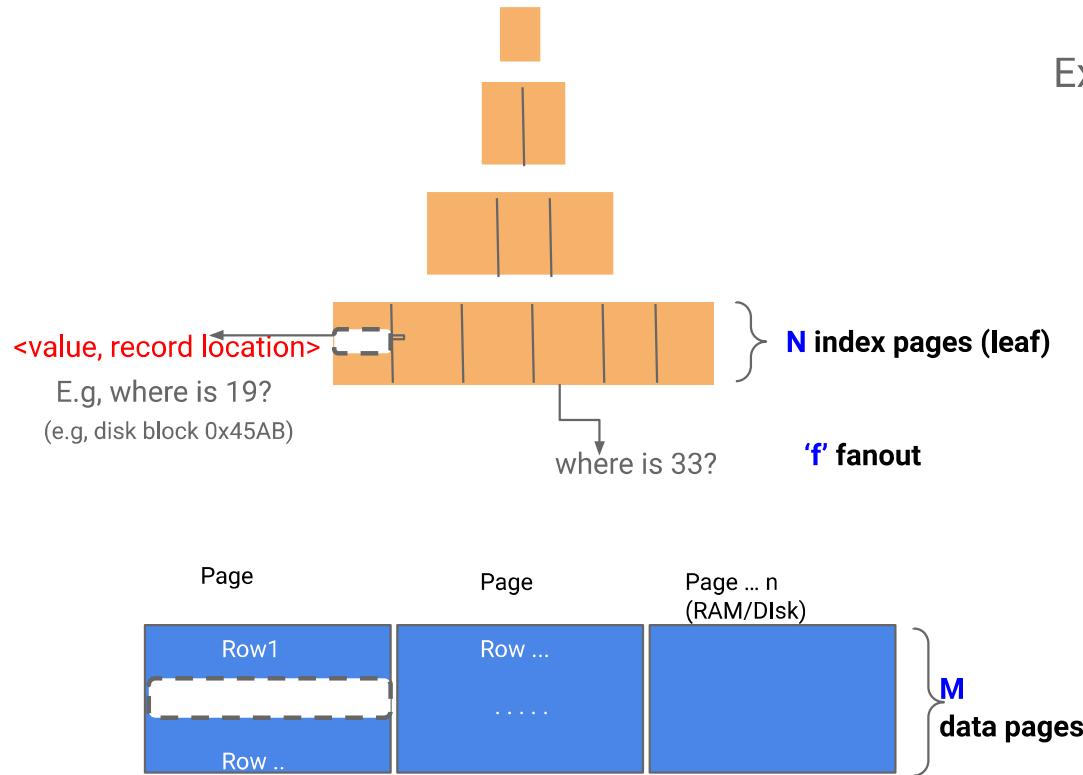
## Week 5

- Problem 3: Build a Product Recommendation system (e.g., Amazon's)



**Q:** What's index size for Amazon's product catalog?  
**(Wrapping up B+ trees example)**

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



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

Example 1: Search Amazon's 1 trillion Products

- SKeySize = 8 bytes,
- PointerSize = 8 bytes
- NumSKeys = 1 Trillion

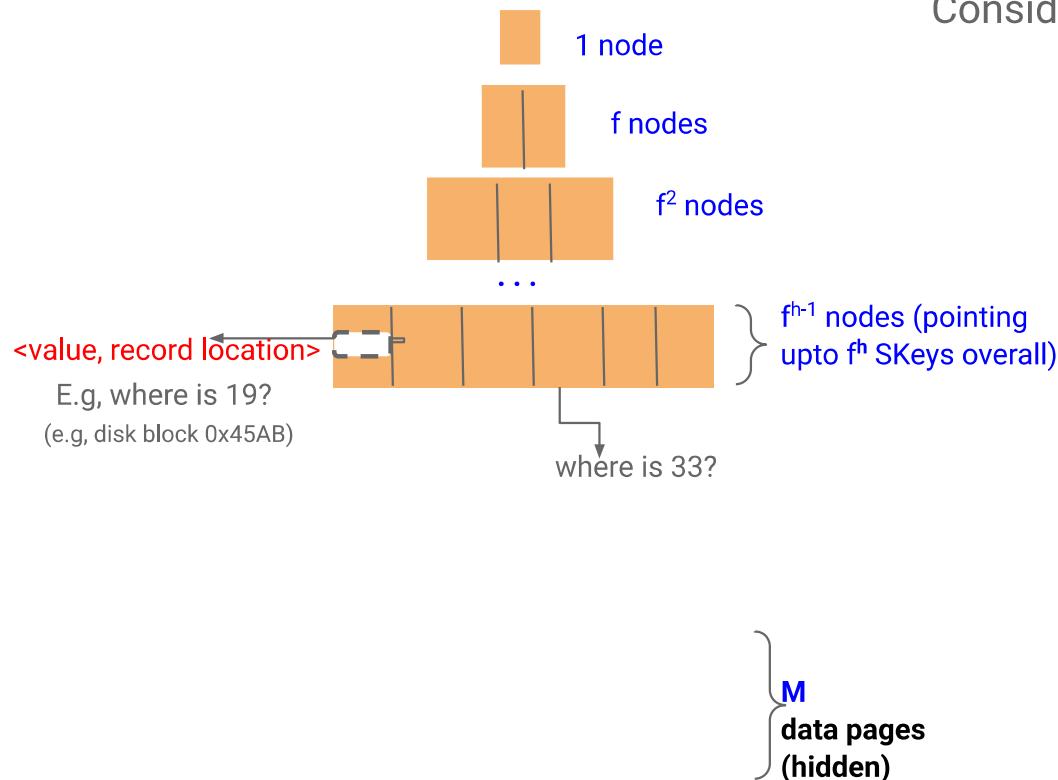
PageS ize	64KB	64MB
f	~4000	~4 million
h	~4	~2

**AMAZING:** Worst-case for 1 Trillion SKeys

- 2 IOs (for 64MB) for index
- 1 IO for data page

# B+ tree Levels and node sizes

$f \leq \frac{\text{PageSize}}{\text{SKeySize} + \text{PointerSize}}$  // Fit upto 'f' (SKey, Pointer)  
 $f^h \geq \text{NumSKeys}$  // Leaf nodes should point to all SKeys  
 $h \geq \log_f \text{NumSKeys}$  // From previous equation



Consider Example with  $f \approx 4000$  and  $\text{PageSize} = 64\text{KB}$

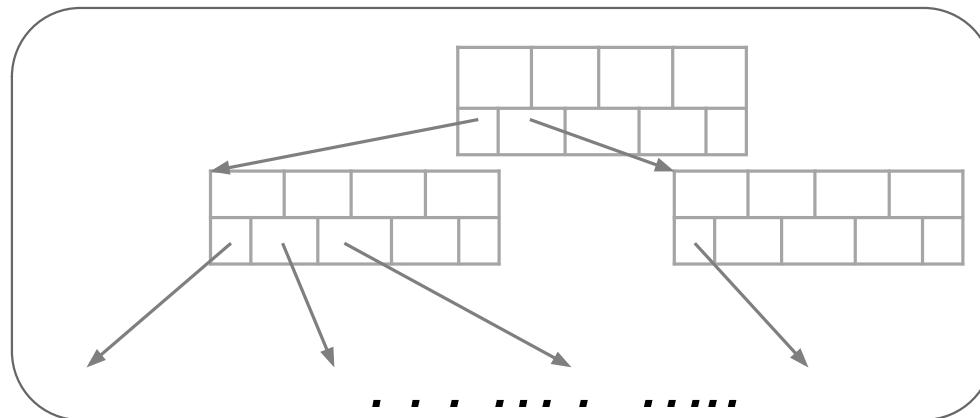
Level	Num Nodes (size)	Size
0 (root)	1	64 KB
1	$\sim 4000$ ( $f$ )	$4000^*$ 64KB
2	$\sim 4000^2$ ( $f^2$ )	$\sim 4000^2*$ 64KB
3	$\sim 4000^3$ ( $f^3$ )	$\sim 4000^3*$ 64KB
$h-1$	$4000^{h-1}$ ( $f^{h-1}$ nodes) [Collectively pointing at upto $f^h$ SKeys]	$4000^{h-1} * 64\text{KB}$

Recall We use fanout 'f' as a constant, for simplicity. The algorithm uses 'f' for keys, and 'f+1' for pointers. Our engineering approximation uses 'f' for both. (i.e.,  $f \approx f + 1$ )

# Search cost of B+ Tree (on RAM + Disk)

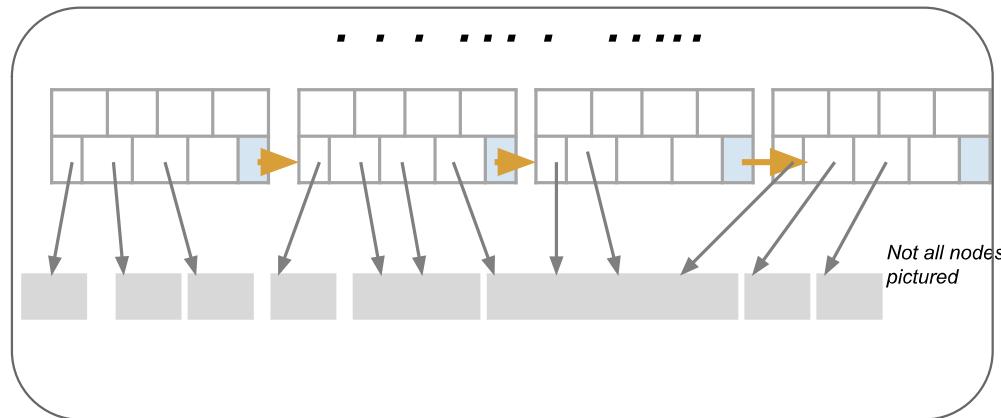


Read 1st levels  
Into RAM buffer



Use **B** (RAM size) to  
keep top few levels  
(e.g., 1st 2-3 levels.  
Overall, as many nodes  
as will fit in B – because  
they're 'zero' cost and  
speed up queries)

Rest of index on  
disk





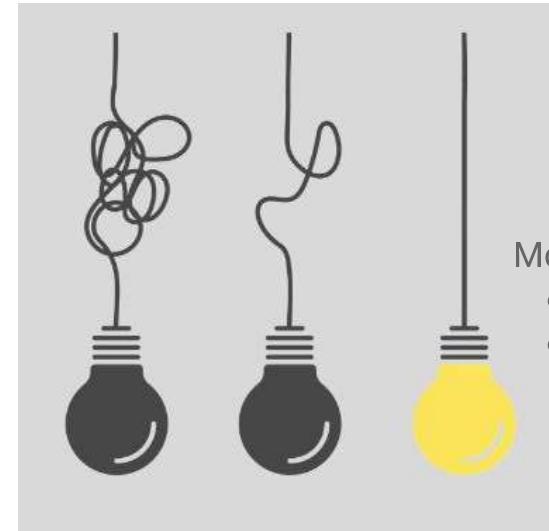
# Problem: How to make SQL queries fast?

## Optimize!

# Approximations

## Complexity

$O(n)$  vs  $O(n^2)$  – Big “O” notation for algorithms



Models of real world systems are complex.  $\Rightarrow$   
• IO cost equations are often ‘dense.’  
• We will approximate and simplify (with most of the needed accuracy), so we focus on the **core insights**.

## Engineering Approximations

Similar goal. We often approximate  $B \approx B+1$ , or  $f \approx f+1$  for simpler equations.

Is that OK?

- Yup, for “modern”  $\sim 2010+$  machines
- When  $B \approx 100$ ,  $B+1/B \approx 1.01$ .

## Example:

# Basic SFW queries

Workload description

200 times/sec

```
SELECT pname  
FROM Product  
WHERE year = ? AND category = ?
```

100 times/sec

```
SELECT pname  
FROM Product  
WHERE year = ? AND Category = ?  
      AND manufacturer = ?
```

Lower cost  
(query and  
update cost)

1. How to execute? Sort, Hash first ...?
2. Maintain indexes for Year? Category? Manufacturer?
3. For query, check multiple indexes?
4. What's cost of maintaining index?
5. Use multiple machines? ...

Intuition

Manufacturers likely most **Selective**.

Few Categories. Many more manufacturers. Maintain index, if this query happens a lot.

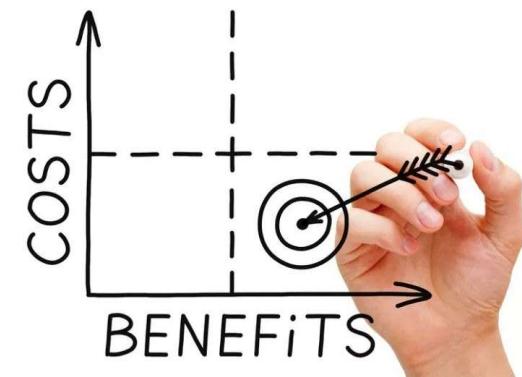
# Optimization

## Roadmap



Build Query 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



Analyze Plans

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



What you will learn about in this section

1. RECAP: Joins
2. Nested Loop Join (NLJ)
3. Block Nested Loop Join (BNLJ)
4. Index Nested Loop Join (INLJ)



**Problem:** JOINS are slow. How to make them fast?

Optimize!

# Semantically: A Subset of the Cross Product

$R \bowtie S$

```
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	

Cross Product



...



Filter by conditions  
( $r.A = s.A$ )

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

Can we actually implement a join in this way?

# Joins: Example

$R \bowtie S$

```
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			S	
A	B	C	A	D
1	0	1	3	7
2	3	4	2	2
2	5	2	2	3
3	1	1		



A	B	C	D
2	3	4	2

# Joins: Example

$R \bowtie S$

```
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			S	
A	B	C	A	D
1	0	1		
2	3	4	3	7
2	5	2	2	2
3	1	1	2	3



A	B	C	D
2	3	4	2
2	3	4	3

# Joins: Example

$R \bowtie S$

```
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

# Joins: Example

$R \bowtie S$

```
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			S	
A	B	C	A	D
1	0	1		
2	3	4	3	7
2	5	2	2	2
3	1	1	2	3



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

# Joins: Example

$R \bowtie S$

```
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  $r.A = s.A$

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

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



# Simple way to execute JOINS – Nested Loop Joins



# Notation for JOIN cost estimates

We consider “IO aware” algorithms: *care about IO*

Given a relation R, let:

- $T(R)$  = # of tuples in R
- $P(R)$  = # of pages in R

Recall that we read / write entire pages (blocks)

We'll assume **B** buffer for input ( $B$  usually  $\ll P(R), P(S)$ )

+ **1** for output ← Imagine there's always 1 extra page

We'll see lots of formulae from now

⇒ Hint: Focus on how it works. Much easier to derive from 1st principles (vs recalling formula soup)

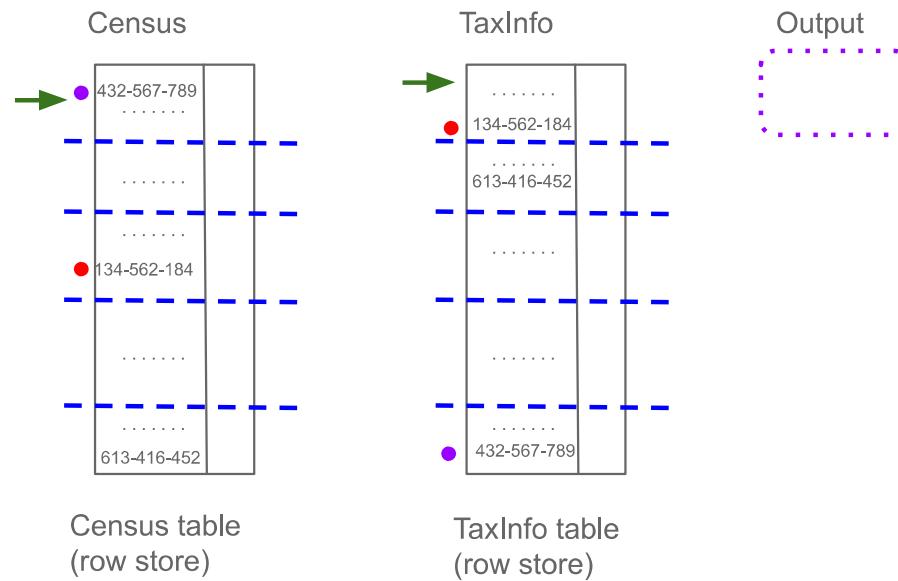
## Recall

Census (SSN, Address, ...)  
TaxInfo (SSN, TaxPaid, ...)

For 1 Billion people

Goal: Compute Census JOIN TaxInfo

Data stored in RowStores, 1000 tuples/page (million pages)



Steps: Repeat till done  
Read tuples from Census and TaxInfo into Buffer  
Output matching tuples



# 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)
```

Is this

- (a) Correct, (b) Incorrect?
- (c) Fast, (d) Slow?

Note: This is our 1st basic algorithm. We use this baseline to intro cost models. We'll see much better ones soon.

# 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)
```

Cost:

$P(R)$

1. Loop over the tuples in R

We read all  $T(R)$  tuples. But they are in  $P(R)$  pages.Cost to read =  $P(R)$

# 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)
```

Cost:

$$\mathbf{P}(R) + \mathbf{T}(R)^*\mathbf{P}(S)$$

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S

**It's  $\mathbf{T}(R)^*\mathbf{P}(S)$  because**

- Per tuple in R (i.e.,  $\mathbf{T}(R)$ ), we read every page of S (i.e.,  $\mathbf{P}(S)$ ).

# 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)
```

Cost:

$$\mathbf{P(R)} + \mathbf{T(R)*P(S)}$$

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

Note that NLJ can handle things other than equality constraints... just check in the *if* statement!

# 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)
```

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)

What would **OUT** be if our join condition is trivial  
(if TRUE)?

**OUT** =  $T(R)*T(S)$  tuples split into multiple pages.  
Could be bigger than  $P(R)*P(S)$ ... but usually not that bad

# 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)
```

Cost:

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

What if R ("outer") and S ("inner") switched?



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

Outer vs. inner selection makes a huge difference-  
DBMS needs to know which relation is smaller!



# IO-Aware Approach



**Problem: JOINS are slow. How to make them fast?**

**Optimize – IO aware**  
**(1000x faster than NLJ?)**

# Intuition: Block Nested Loop Joins

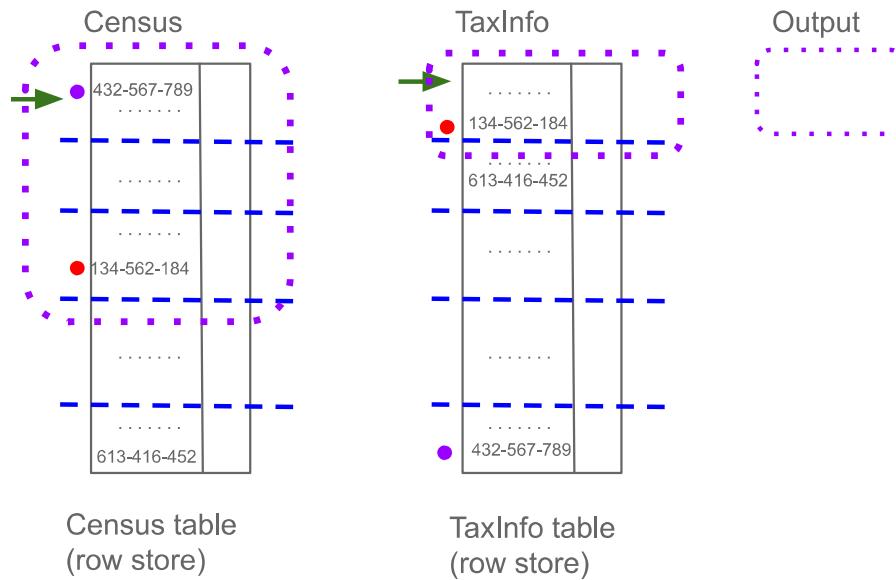
Census (SSN, Address, ...)  
TaxInfo (SSN, TaxPaid, ...)

For 1 Billion people

Goal: Compute Census JOIN TaxInfo

Data stored in RowStores, 1000 tuples/page (million pages)

Given: B buffer space for input. + 1 for output  
Idea: Use **B-1** pages for Census, **1** page for TaxInfo



Steps: Repeat till done  
Read B-1 pages from Census into Buffer  
Read 1 page from TaxInfo  
Partial Join into 1 output page

# Block Nested Loop Join (BNLJ)

Assume  $B$  buffer for input  
+ 1 for output (For  $B \ll P(R), P(S)$ )

Cost:

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

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

$P(R)$

1. Load in  $B-1$  pages of  $R$  at a time (leaving 1 page each free for  $S$  & output)

*Why  $P(R)$ ?*

1. Overall, read all  $P(R)$  pages, even if it's  $B-1$  at a time
2. *There could be some speedup here, if we're reading in multiple pages sequentially however we'll ignore this here!*

# Block Nested Loop Join (BNLJ)

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

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

Cost:

$$P(R) + P(R)*P(S)/B$$

How many  $R$  pages to read?  
 $P(R)$

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

1. Load in  $B-1$  pages of  $R$  at a time (leaving 1 page for  $S$ )
2. For each  $(B-1)$ -page segment of  $R$ , load each page of  $S$

Notes:

1. Faster to iterate over the *smaller* relation first!
2. We use  $1/B$  in denominator, for our engg approximation (vs  $1/(B-1)$ ).

# Block Nested Loop Join (BNLJ)

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

Cost:

$$P(R) + P(R)*P(S)/B$$

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

BNLJ can also handle non-equality constraints

# Block Nested Loop Join (BNLJ)

Cost:

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

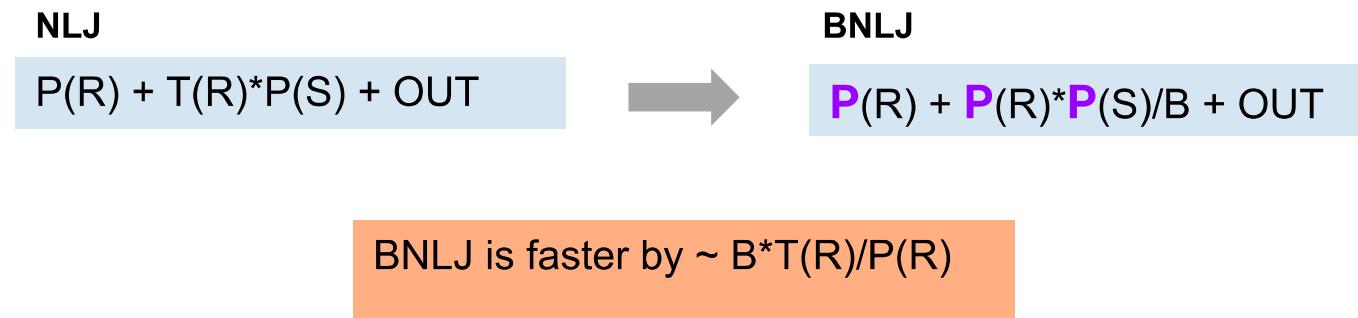
$P(R) + P(R)*P(S)/B + OUT$

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

# BNLJ vs. NLJ: Benefits of IO Aware

In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S

- Still the full cross-product, but more done only *in memory*



# Example NLJ vs. BNLJ: Steel Cage Match

Example:  $P(R) = 1000$ ,  $P(S) = 500$ ,  
100 tuples/page  $\Rightarrow T(R) = 1000 * 100, T(S) = 500 * 100]$

	$B = 100$ (+ 1 for output)	$B = 20$ (+ 1 for output)
NLJ	$(1000 + 1000 * 100 * 500 + OUT)$ $\Rightarrow IO = \sim 5,001,000 + OUT$	$(1000 + 1000 * 100 * 500 + OUT)$ $\Rightarrow IO = \sim 5,001,000 + OUT$
BNLJ	$(1000 + 1000 * 500 / 100)$ $\Rightarrow IO = \sim 6000 + OUT$	$(1000 + 1000 * 500 / 20)$ $\Rightarrow IO = \sim 26,000 IOs + OUT$

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

$P(R) + P(R) * P(S) / B + OUT$

Small change in algorithm  $\Rightarrow$  Big speedup in JOINs (**~1000x faster**)  
Also, notice if we swap R and S, we can save an extra **500 IOs** in BNLJ



# **Problem: JOINS are slow. How to make them fast?**

**(Can we do 100x faster than BNLJ?)**

Idea:

## **Smarter cross-products**



What you will  
learn about in  
this section

0. Index Joins
1. Sort-Merge Join
2. HashPartition Joins



# Smarter than Cross-Products: From Quadratic to Nearly Linear

All joins computing the *full cross-product* have a quadratic term

- For example we saw:

$$\begin{array}{ll} \text{NLJ} & P(R) + \boxed{T(R)*P(S)} + \boxed{OUT} \\ & \downarrow \quad \downarrow \\ \text{BNLJ} & P(R) + \boxed{P(R)*P(S)/B} + \boxed{OUT} \end{array}$$

Now we'll see some (nearly) linear joins:

- $\sim O(P(R) + P(S) + OUT)$

We get this gain by *taking advantage of structure*- moving to equality constraints ("equijoin") only!

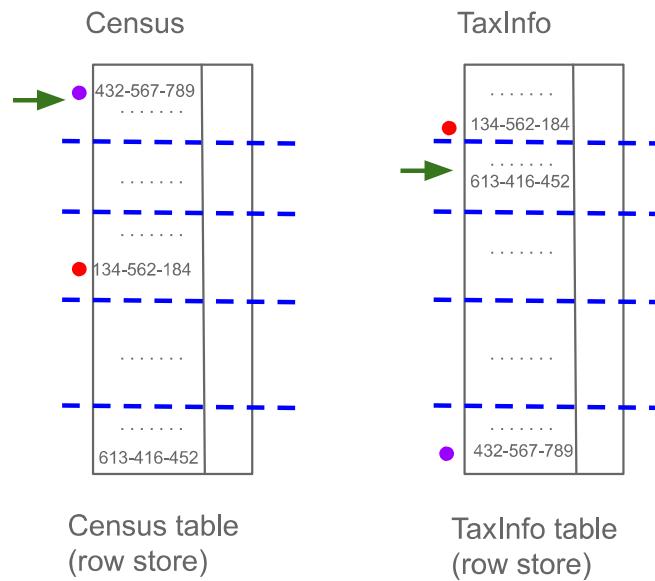
## Example

Census (SSN, Address, ...)  
TaxInfo (SSN, TaxPaid, ...)

For 1 Billion people

Goal: Compute Census JOIN TaxInfo

Data stored in RowStores, 1000 tuples/page (million pages)



BNLJ -- See all the extra work BNLJ is doing  
to JOIN for 432-567-789, ...



# Index Nested Loop Join (INLJ)

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

Given index  $\text{idx}$  on  $S.A$ :  
for  $r$  in  $R$ :  
   $s$  in  $\text{idx}(r[A])$ :  
    yield  $r,s$

Cost:

$$P(R) + T(R)^*L + OUT$$

Where  $L$  is the IO cost to access each distinct value in index

Recall:  $L$  is usually small (e.g., 2-5)

→ We can use an **index** (e.g. B+ Tree) to **avoid full cross-product!**

→ Much better than quadratic. But what if  $T(R) = 1$  billion?



## OCTOBER 2022

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	HW2 <sup>19</sup> out	HW2 <sup>20</sup> section		22
23	24	25	HW2 <sup>26</sup> due	Review	27	28
30	31					29

## NOVEMBER 2022

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
			1 <u>Midterm</u>	2	3	4
5						
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

1.

### Nov 1st Midterm (dates/details on cs145 website)

- a. Material covered till **Oct 20th** lecture (Oct 25th lecture tested in Finals, not in Midterm)
- b. Exam format: ~5 questions (SQL, Systems, materials in 1st 8 lectures)
- c. Similar questions to HW1, HW2, Project1 (sql). I.e., problems are the best prep. +1 prep-test out ~Oct 27
- d. **HW2 (Oct 19) - due Oct 26th HW2** (no late days) 11:59 pm due
  - i. Shorter than HW1 (moved some problems to HW3)
  - ii. Oct 21 Section for HW2
  - iii. Oct 27 HW2 11:59 pm solutions released
  - iv. For OAE students who need an extra day, please plan ahead now
- e. Oct 27th Test prep in class

2.

# FAQ: Why do we focus on IO cost?

From Lecture1



Srigi  
@srigi

"Latency Numbers Every Programmer Should Know"

It is hard for humans to get the picture until you translate it to "human numbers":

1 CPU cycle	1 s
Level 1 cache access	3 s
Level 2 cache access	9 s
Level 3 cache access	43 s
Main memory access	6 min
Solid-state disk I/O	2-6 days
Rotational disk I/O	1-12 months
Internet: SF to NYC	4 years
Internet: SF to UK	8 years
Internet: SF to Australia	19 years
OS virtualization reboot	423 years
SCSI command time-out	3000 years
Hardware virtualization reboot	4000 years
Physical system reboot	32 millenia

## [1] CPU Cost

(e.g., sort in RAM or check tuple equality in NLJ in RAM)

Typical algorithms in RAM  
(e.g., quicksort in  $n \log n$ )

## [2] IO Cost from HDD/SSDs

(# of Pages we read or write fromHDD/SSD)

E.g., For ExternalSort

$$\text{IO Cost } \approx \text{ (sort } N \text{ pages)} \quad 2N \left[ \log_B \frac{N}{2B} \right] + 2N$$

⇒ For big data, focus on **IO cost** (i.e., it's the primary factor)  
(For tiny data < 10 GBs, just use RAM)

# FAQ: Why do we focus on scale? How does it relate to SQL?

1. How to search Amazon's product catalog?
  - a. Consumers want answers in < 1 sec
  - b. Data layout and Indexing –
    - i. Lecture 1 vs Lecture 6 (speedup “search” queries from **hours** to < 1sec)

---

2. How to run JOINs fast?
  - a. Improved by 1000x from NLJ to BNLJ
  - b. Will do another 100x today

⇒ For big data, how to scale is the primary driver

(For tiny data < 10 GBs, just use RAM)

## Agenda

1. Systems Primer (Week 1)
  - Refresher on how data flows in hardware, between CPU and IO (RAM, SSDs, HDDs). Data access in RAM vs in SSDs – is it 10x, 1000x, or 100,000x faster? There's a big demand in industry for 'full stack' engineers. Especially folks who understand systems, and have good intuition for speed and scale tradeoffs at every layer of the stack (frontend apps, backend servers, DBs).
  - **Goal:** Learn how to analyze speed and cost tradeoffs in hardware.
2. SQL (Weeks 1 and 2)
  - SQL is the world's most popular parallel programming language for data. Works everywhere, from tiny CPUs embedded in sensors, and CPU cores on your smartphones, to clouds with millions of CPU cores.
  - SQL was popular for decades. For a few years, ceded some mind share to "NoSQL" and "NewSQL" systems (e.g., Hadoop, Mongo, etc.). In the past 5 years, SQL has surged as the lingua franca for "modern data stacks."
  - **Goal:** Learn SQL basics. Apply in 3 projects on Google's BigQuery.
3. Scaling Queries and Analytics (i.e., reads) (Week 3 and 4)
  - Amazon had a peak of ~[105 million queries per sec](#) on Prime day! Google, Youtube, TikTok scale to similarly big numbers. How?
  - **Goal:** Learn key algorithms and data structures, independent of SQL and nonSQL systems. Extend **Sorting** and **Hashing** algorithms when data does not fit in RAM.
  - **Goal:** See how DBs run parallel queries on 1,000x bigger data sets (versus spreadsheets and Pandas).
4. Scaling Transactions (i.e., writes) (Weeks 6 and 7)
  - VISA's credit card systems process ~100k transactions/second, across consumers and merchants. How?
  - When you get \$100 from an ATM machine, imagine the ATM crashes or loses its network connection. How does the bank update your balance?
  - **Goal:** Learn how to use **Locks** and **Logs** to build a scalable Transaction system.
5. Schemas (Weeks 8 and 9)
  - **Goal:** Design good tables. Communicate good designs (e.g., DAG and ER diagrams)

**Reminder: reread “[Why cs145](#)” from Week 1.**



# **Problem: JOINS are slow. How to make them fast?**

**(Can we do 100x faster than BNLJ?)**

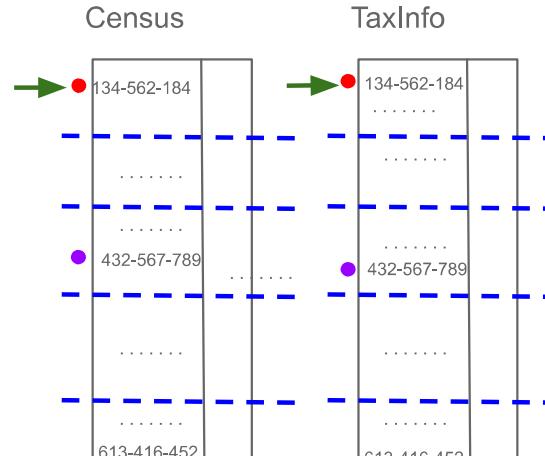
Idea:

## **Smarter cross-products**

Preview of  
doing better?

# Pre-process data before JOINing

SortMergeJoin

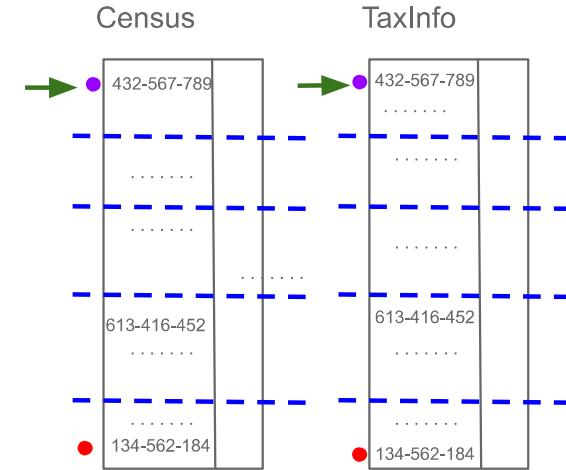


Census table  
(row store)

TaxInfo table  
(row store)

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

HashPartitionJoin



Census table  
(row store)

TaxInfo table  
(row store)

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

# Join Algorithms: Summary

For  $R \bowtie S$  on column 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!

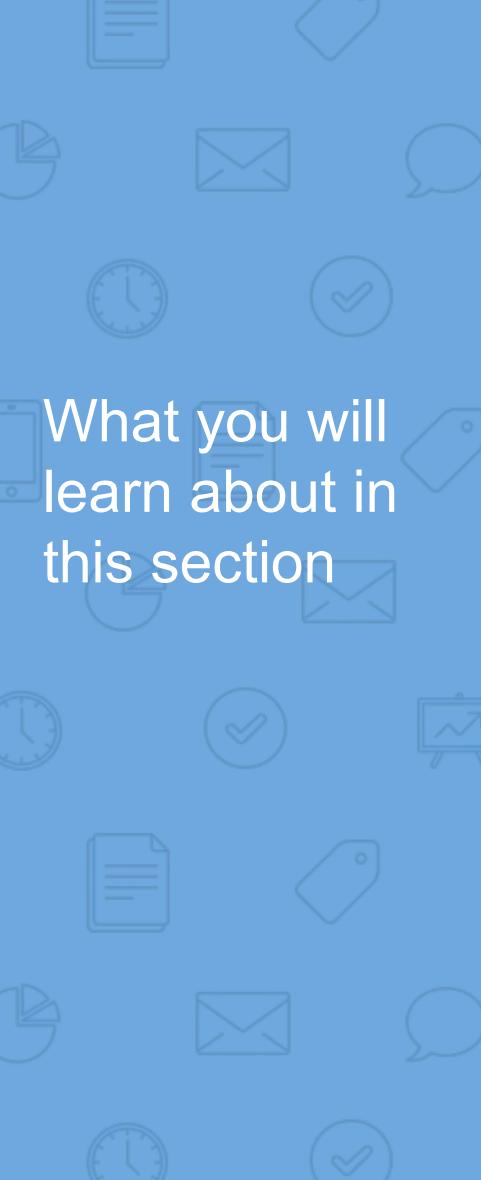
- HPJ: Partition R and S into buckets using a hash function, then join the (much smaller) matching buckets

Given sufficient buffer space, linear in  $P(R)$ ,  $P(S)$   
I.e.  $\sim O(P(R)+P(S))$

By only supporting equijoins & taking advantage of this structure!



# Sort-Merge Join (SMJ)



What you will learn about in this section

1. Sort-Merge Join
2. “Backup” & Total Cost
3. Optimizations

# Sort Merge Join (SMJ)

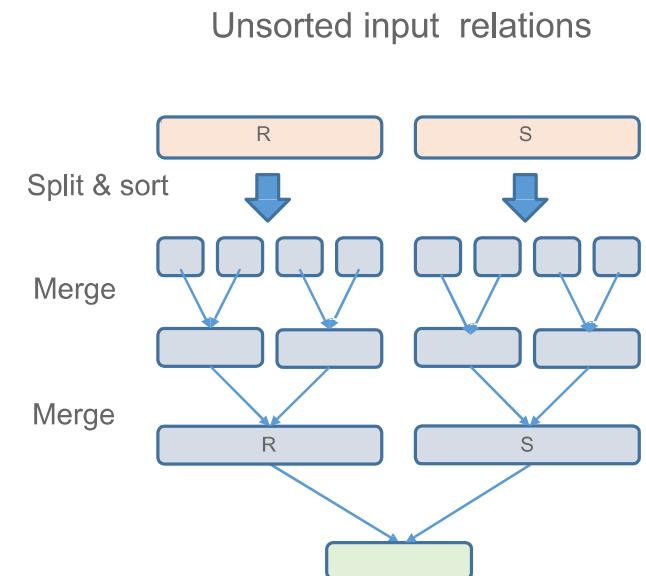
**Goal:** Execute  $R \bowtie S$  on A

## Key Idea:

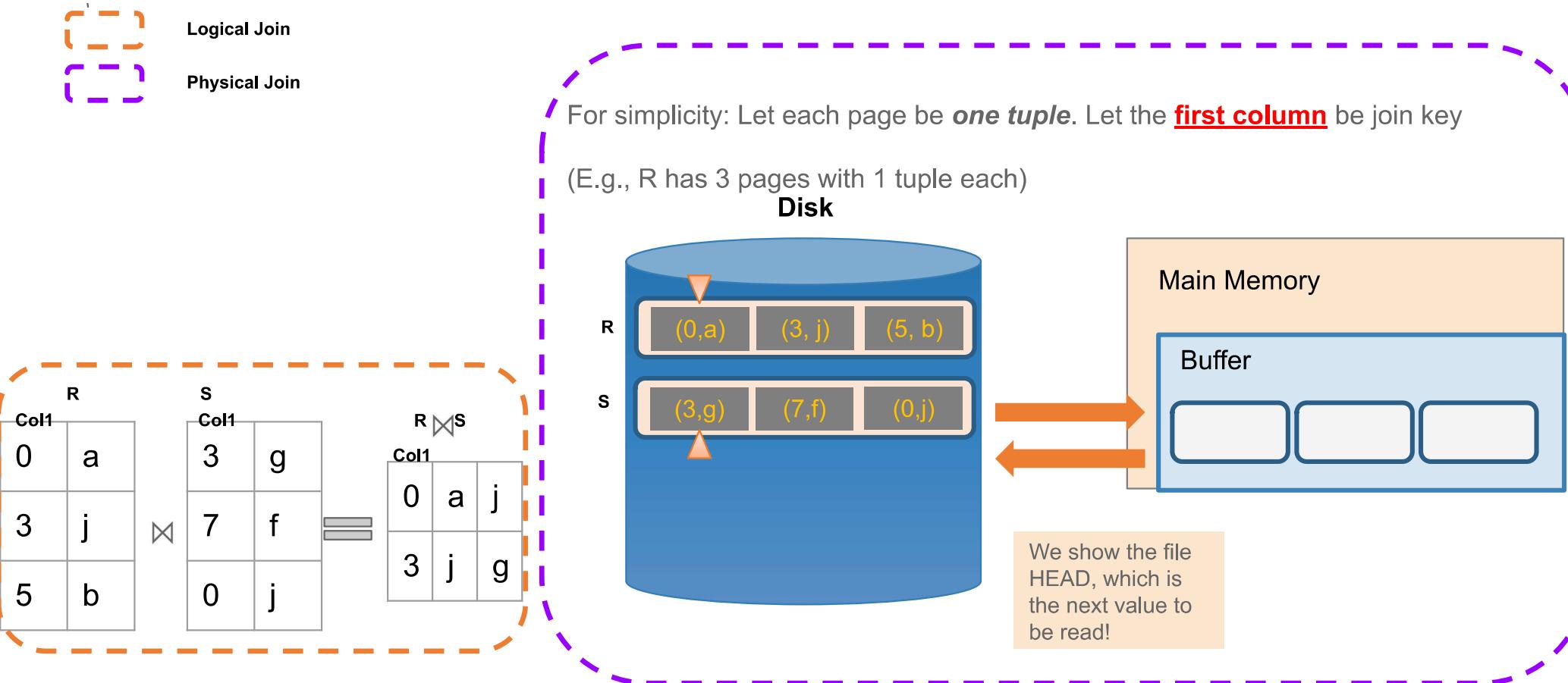
Sort R and S [with external sort]  
Merge-scan (aka merge-join) over them!

## IO Cost:

- Sort phase:  $\text{Sort}(R) + \text{Sort}(S)$
- Merge-join phase:  $\sim P(R) + P(S) + \text{OUT}$

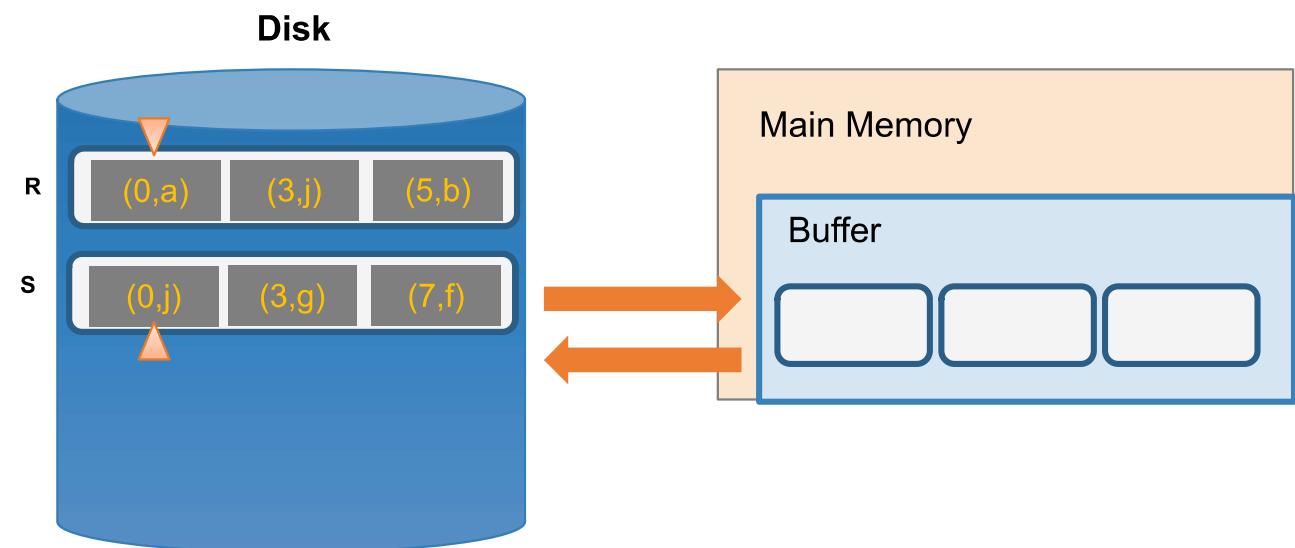


# Example1: SMJ: $R \bowtie S$ with 3 page buffer



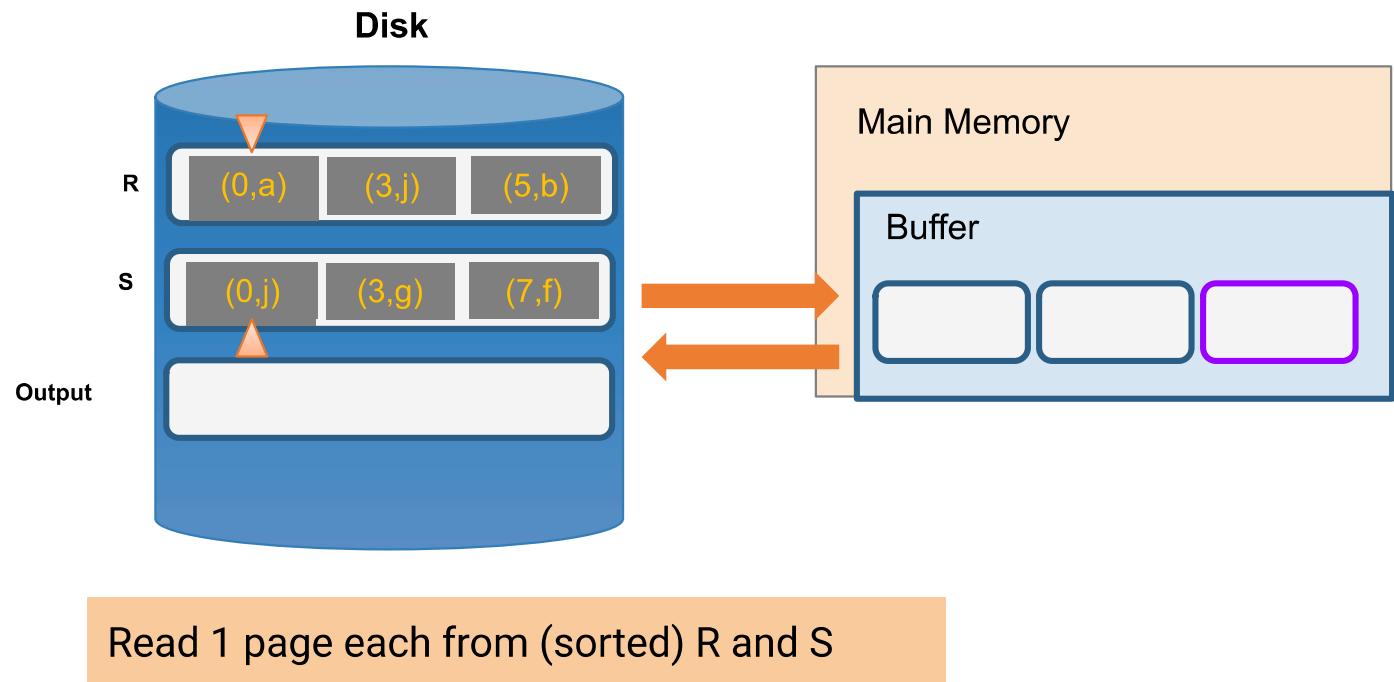
# SMJ Example: $R \bowtie S$ with 3 page buffer

1. Sort( $R$ ) and Sort( $S$ ) (on 1st column)



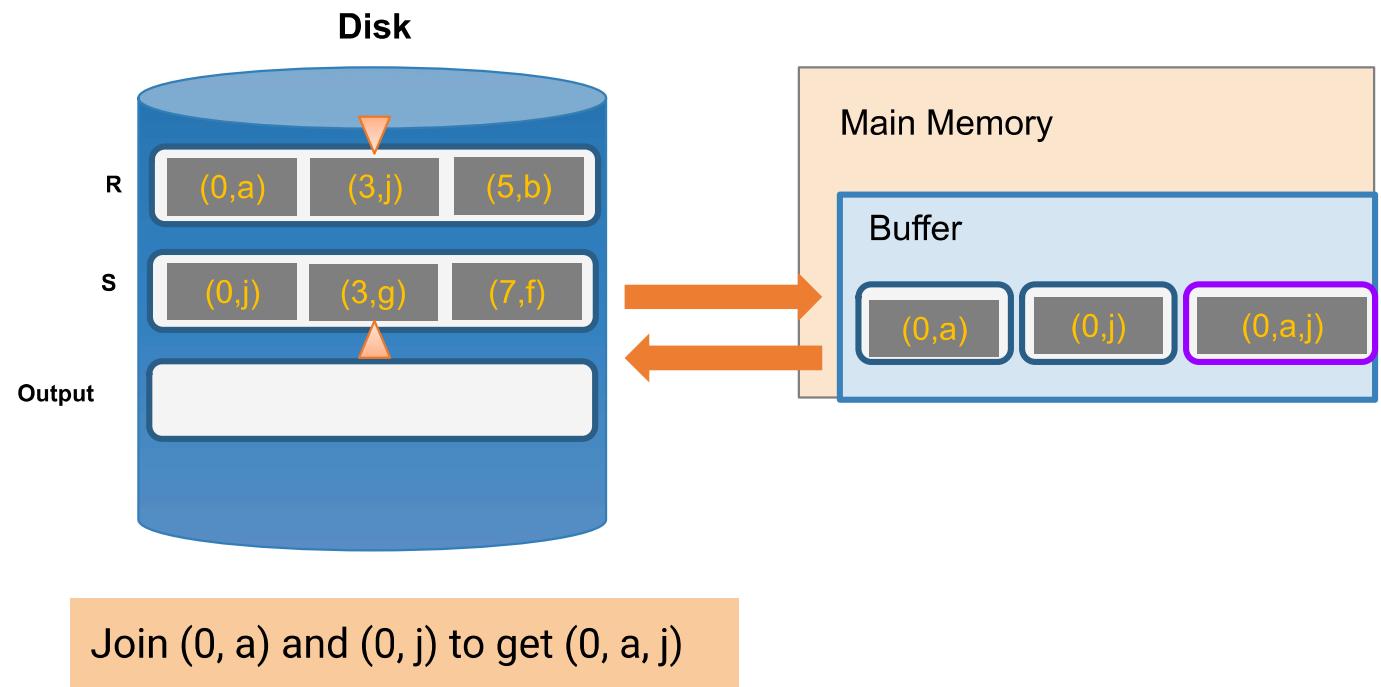
# SMJ Example: $R \bowtie S$ with 3 page buffer

2. Merge-scan (or merge-join) – Scan and “merge” on join key!



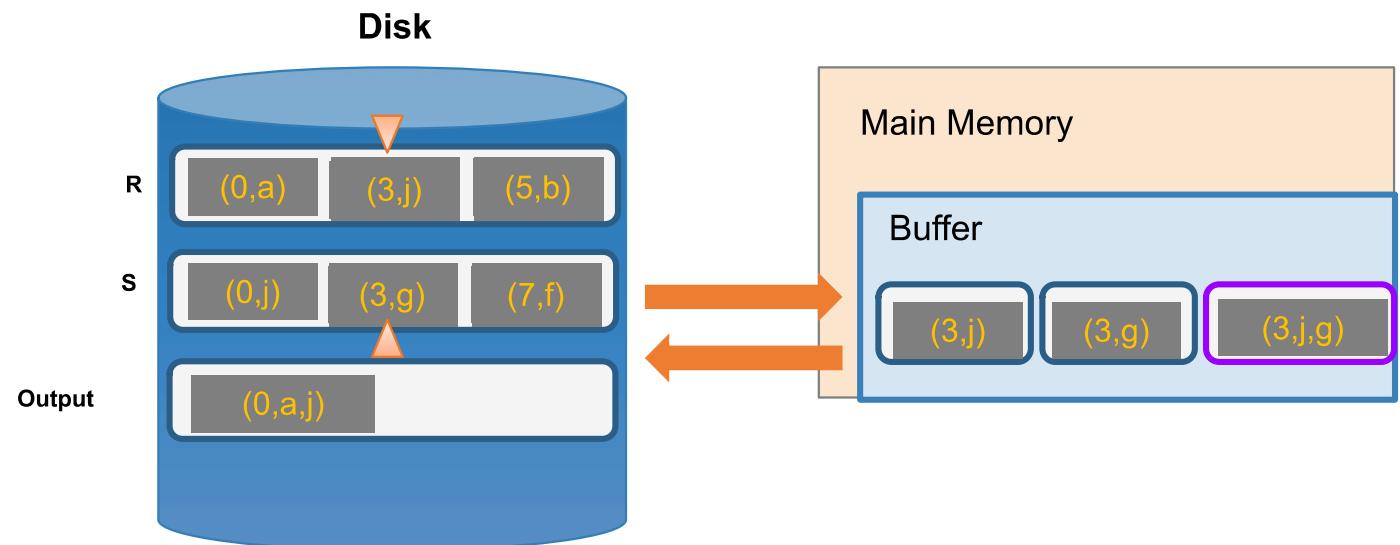
# SMJ Example: $R \bowtie S$ with 3 page buffer

2. Scan and “merge” on join key!



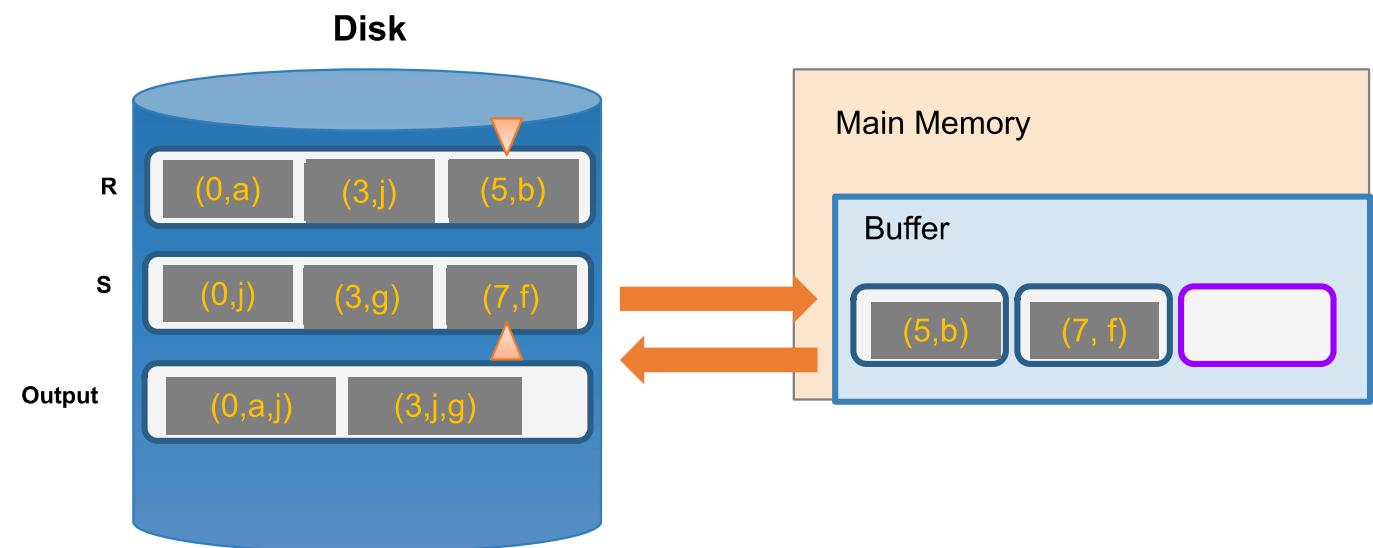
# SMJ Example: $R \bowtie S$ with 3 page buffer

2. Scan and “merge” on join key!



# SMJ Example: $R \bowtie S$ with 3 page buffer

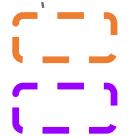
2. Done!





# What happens with duplicate join keys?

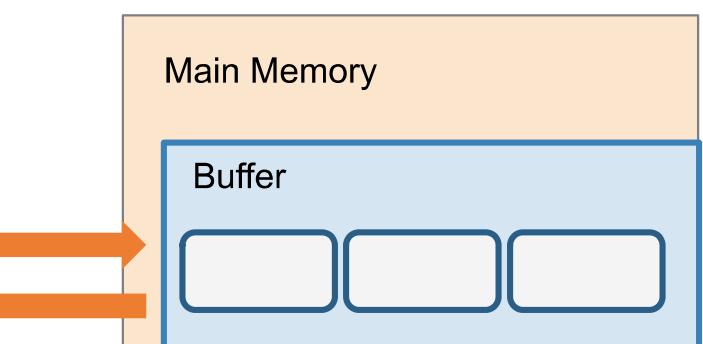
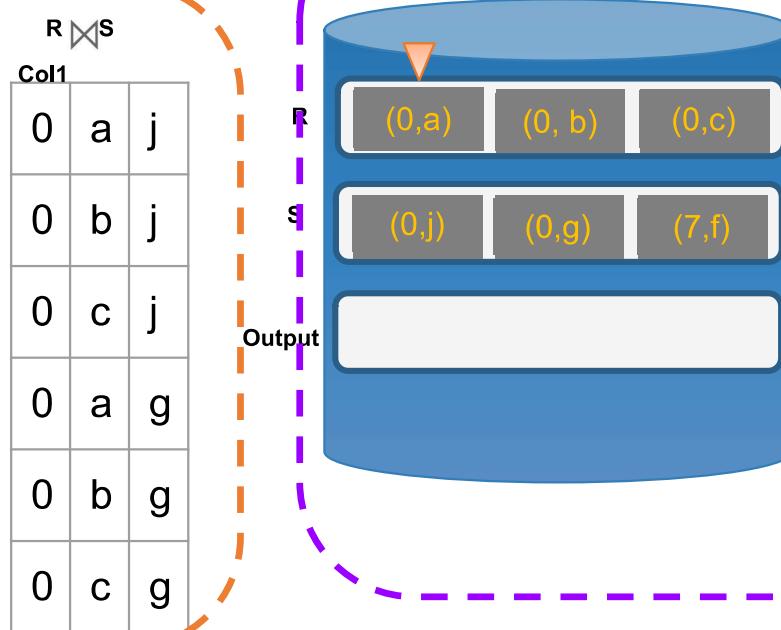
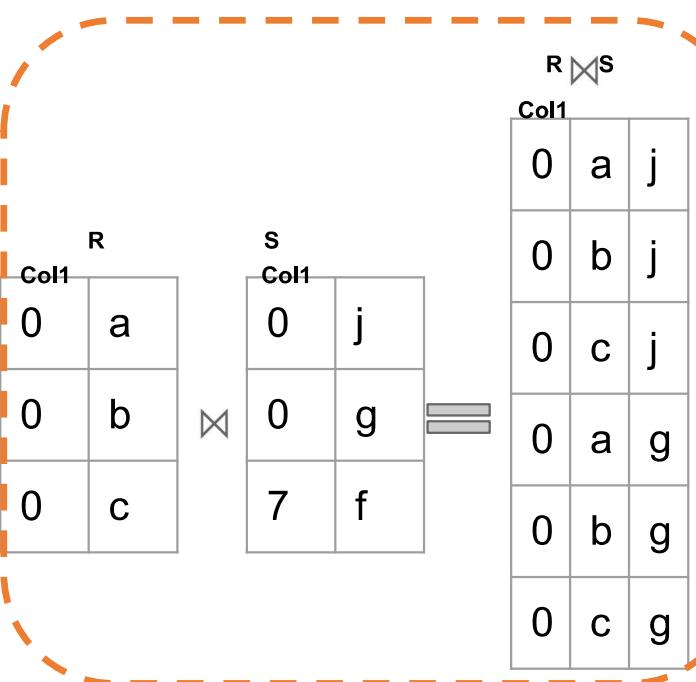
## Example2: Multiple tuples with Same Join Key: “Backup”



Logical Join

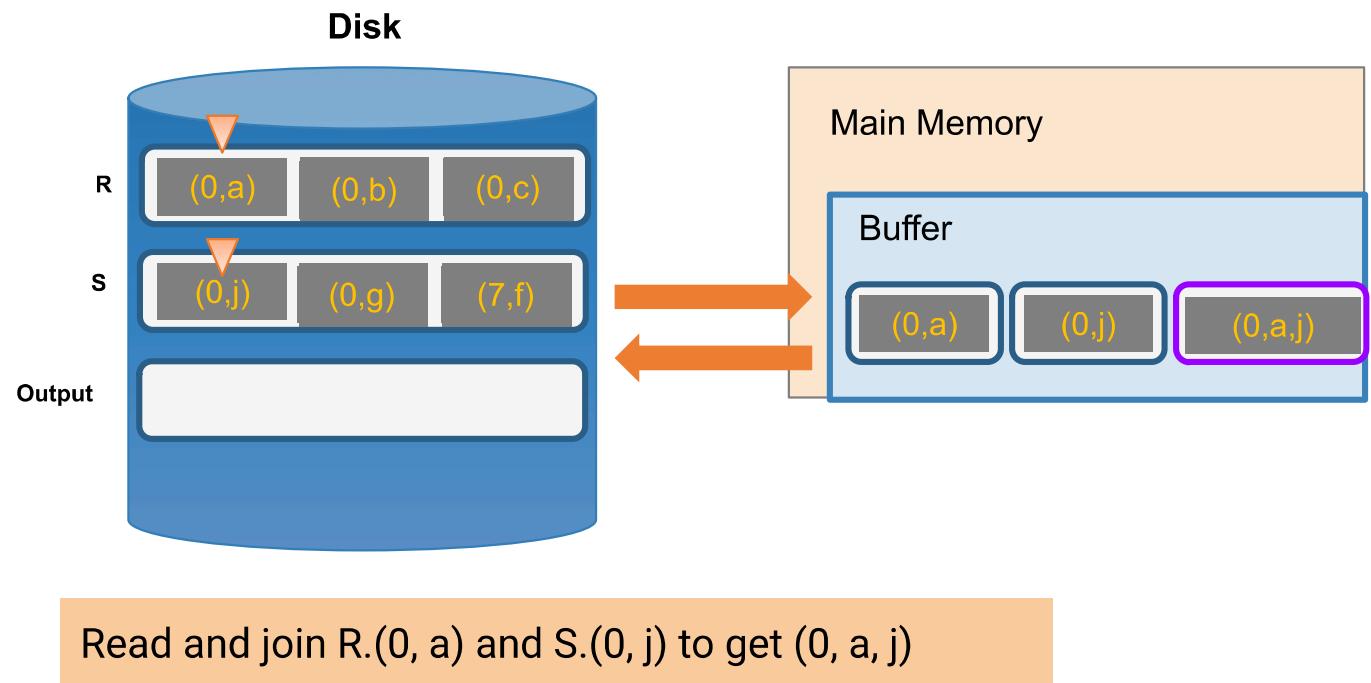
Physical Join

1. Start with sorted relations, and begin merge-join...



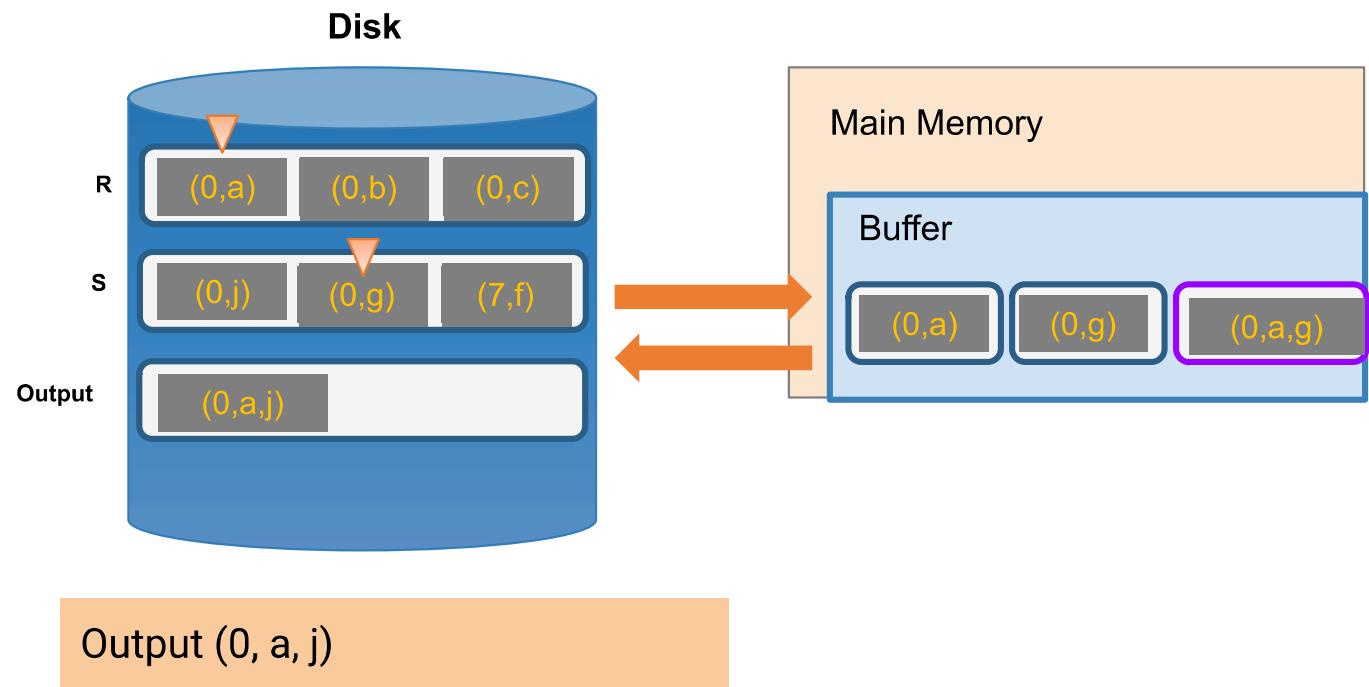
# Multiple tuples with Same Join Key: “Backup”

1. Start with sorted relations, and begin merge-join...



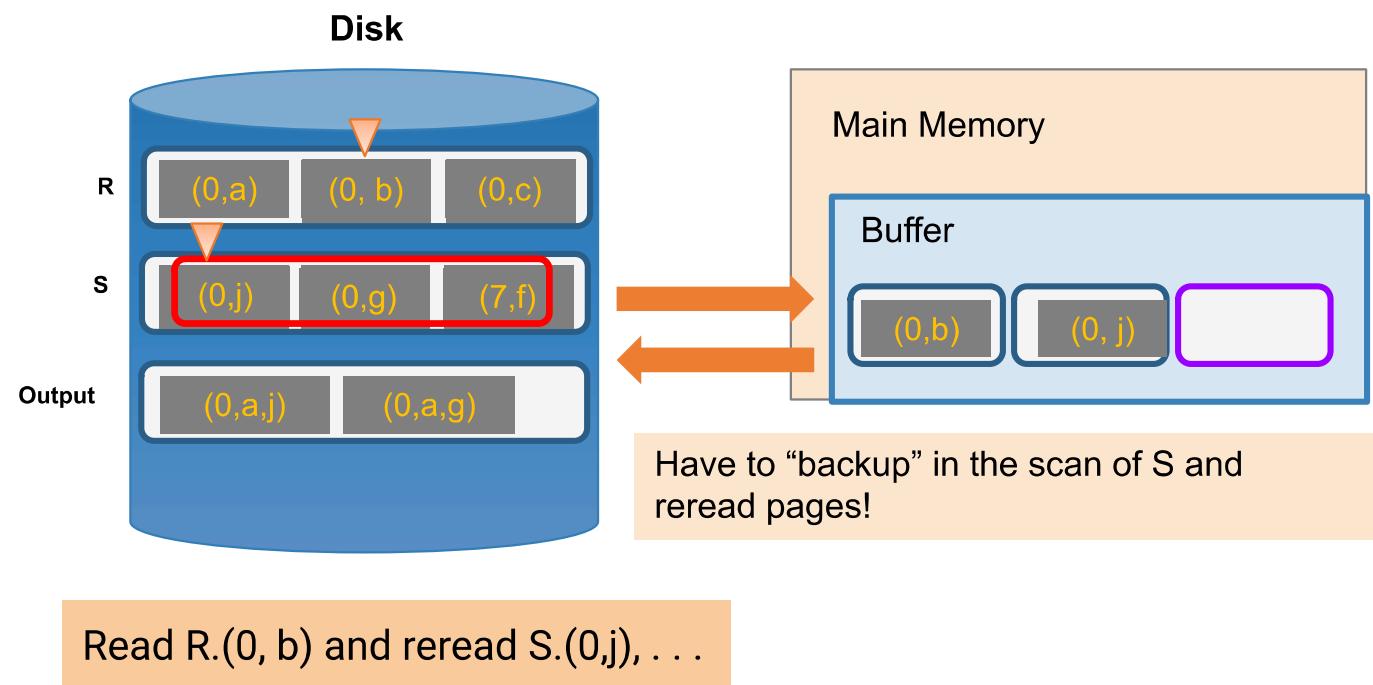
# Multiple tuples with Same Join Key: “Backup”

1. Start with sorted relations, and begin merge-join...



# Multiple tuples with Same Join Key: “Backup”

1. Start with sorted relations, and begin merge-join...



# Backup

- At best, no backup → merge-scan takes  $P(R) + P(S)$  reads
  - For ex: if no duplicate values in join attribute
- At worst (e.g. full backup each time), merge-scan could take  $P(R) * P(S)$  reads!
  - For ex: if *all* duplicate values in join attribute, i.e. all tuples in R and S have the same value for the join attribute
  - Roughly: For each page of R, we'll *back up* and read each page of S...
- Often not that bad however, plus we can:
  - Leave more data in buffer (for larger buffers)
  - Can design other algorithms



## SMJ: Total cost

- **Cost of SMJ** is
  - **Cost of sorting R and S...**
  - Plus the **cost of merge-join**:  $\sim P(R) + P(S)$ 
    - Because of *backup*: in worst case  $P(R)*P(S)$ ; but unlikely
- Plus the **cost of writing out**: OUT

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

IO Cost  $\sim$   
(sort N pages)

NumPasses

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

## Example: SMJ NumPasses

Consider  $P(R) = 1000$ ,  $P(S) = 500$

IO Cost ~=  
(sort N pages)

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

Case1: **NumPasses** for R (for  $B = 100$ ) =  
 $K = \lceil \log_{100} 1000/(2*100) \rceil = 1$

Case2: **NumPasses** for R (for  $B = 20$ )  
 $K = \lceil \log_{20} 1000/(2*20) \rceil = 2$

(Repeat for S, and you get k = 1 and 2)

Reminder: More Buffer? Fewer passes for Sorting

# Example SMJ vs. BNLJ: Steel Cage Match

Error fixed 10/20 after lecture.

1. Let's use full formulae pre-optimization in next 2 slides.
2. Note: with optimization in next 2 Slides, we'll need fewer IOs (we skip the last merge steps).

Consider  $P(R) = 1000, P(S) = 500$

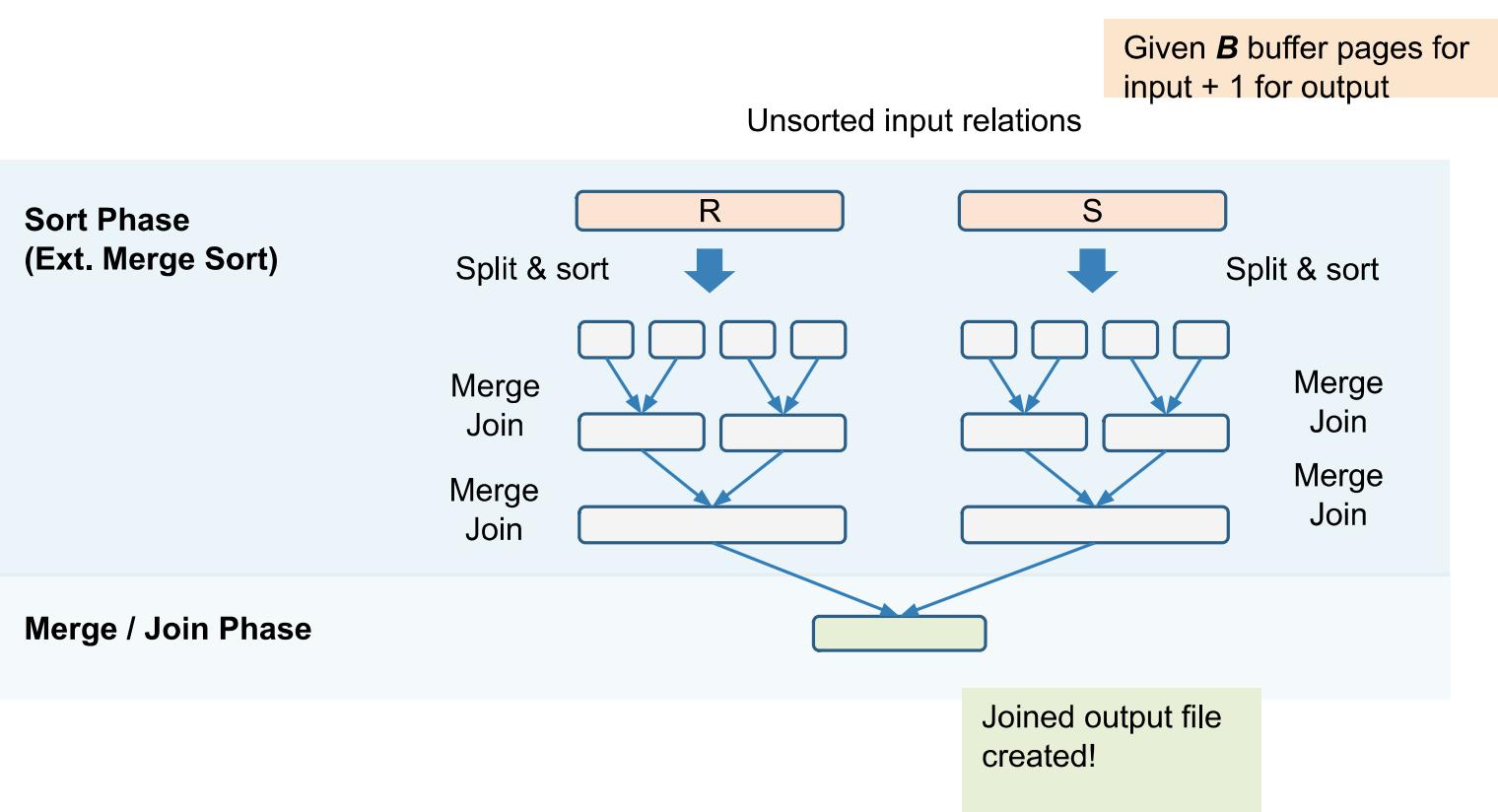
IO Cost ~=  
(sort N pages)

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

		$B = 100$	$B = 20$
$\sim \text{Sort}(P(R)) + \text{Sort}(P(S)) + P(R) + P(S) + \text{OUT}$	SMJ	<p>(Sort R and S in NumPasses (np) = 1  <math>2*1000*(np+1) + 2*500*(np+1) = 6000</math></p> <p>MergeJoin: <math>1000 + 500 = 1500</math> IOs)</p> <p><math>\Rightarrow \text{IO} = 7500 \text{ IOs} + \text{OUT}</math></p>	<p>(Sort R and S in NumPasses (np)=2  <math>2*1000*(np+1) + 2*500*(np+1) = 9000</math></p> <p>MergeJoin: <math>1000 + 500: 1500</math> IOs)</p> <p><math>\Rightarrow \text{IO} = 10,500 \text{ IOs} + \text{OUT}</math></p>
	BNLJ	<p><math>(500 + 1000*500/100)</math></p> <p><math>\Rightarrow \text{IO} = 5500+\text{OUT}</math></p>	<p><math>(500 + 1000*500/20 )</math></p> <p><math>\Rightarrow \text{IO} = 25500 \text{ IOs} + \text{OUT}</math></p>

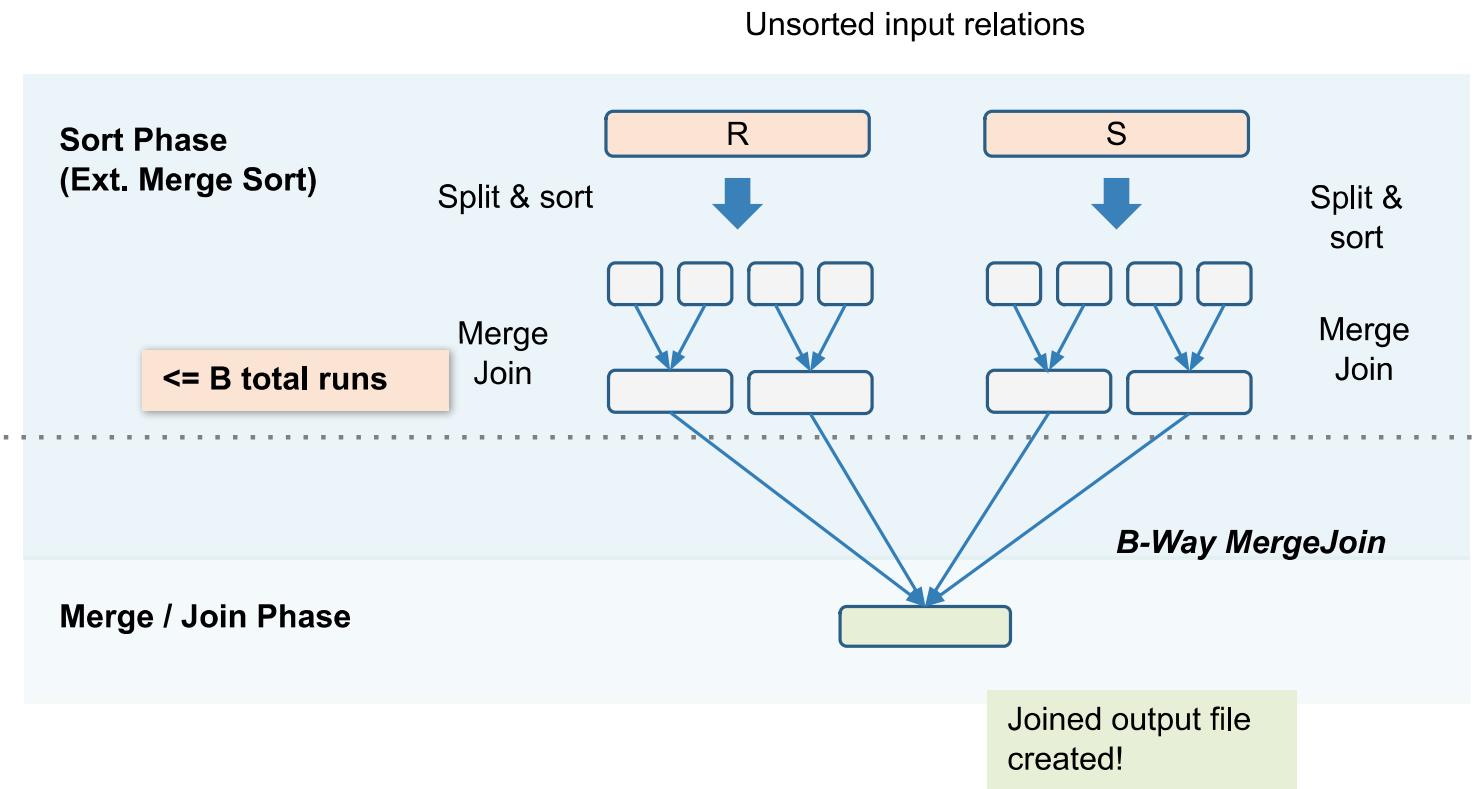
SMJ is ~ linear vs. BNLJ is quadratic...  
 Redo the same with 10x? SMJ much faster.

# Un-Optimized SMJ





# Simple SMJ Optimization





## Takeaway points from SMJ

SMJ needs to sort **both** relations

- SMJ is basically linear.
- Nasty but unlikely case: Many duplicate join keys.

If relation(s) already sorted on join key, no more work.



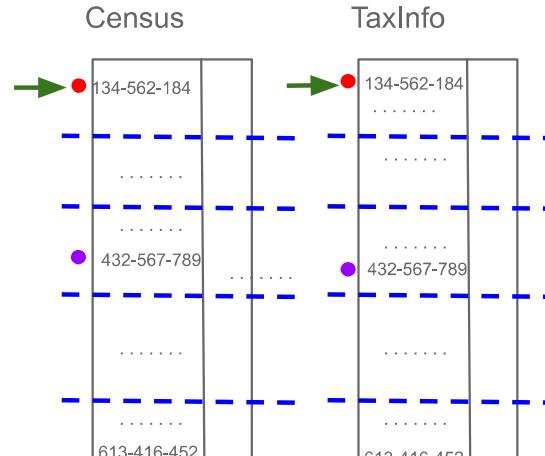
What you will  
learn about in  
this section

0. Intuition for smarter joins
1. SortMergeJoin
2. HashPartition Joins

## Preview of smarter joins

# Pre-process data before JOINing

SortMergeJoin

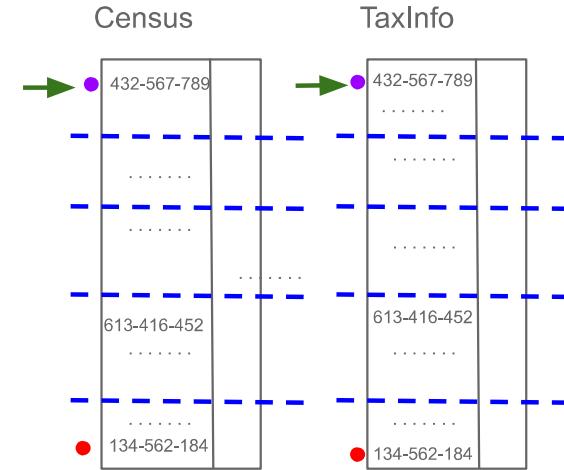


Census table  
(row store)

TaxInfo table  
(row store)

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

HashPartitionJoin



Census table  
(row store)

TaxInfo table  
(row store)

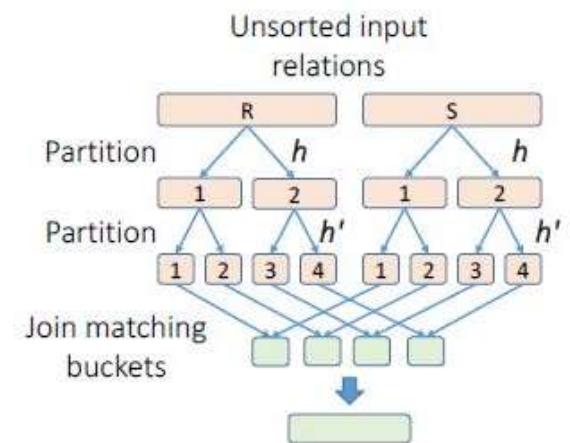
```
-- Hash(Census), Hash(TaxInfo) on SSN  
-- Merge partitioned pages
```



# Hash Join (HJ) or Hash Partition Join (HPJ)

# Hash Join

- **Goal:** Execute  $R \bowtie S$  on A
- **Key Idea:**
  - Partition R and S into buckets by hashing the join attribute
  - Join the pairs of (small) matching buckets!





# HPJ Phase 1: Hash Partitioning

**Goal:** For each relation, partition relation into **buckets** such that if  $h_B(t.A) = h_B(t'.A)$  they are in the same bucket

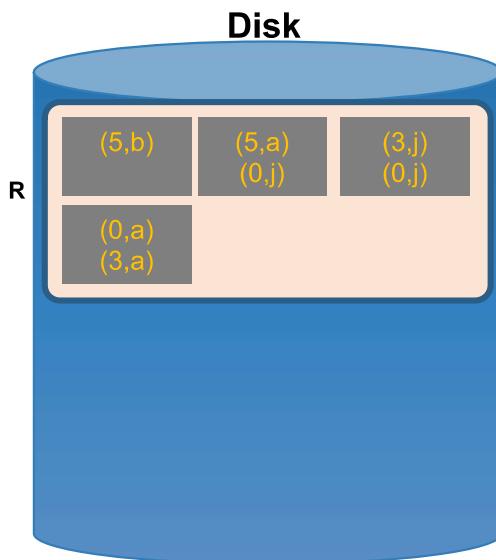
Given  $B+1$  buffer pages, we partition into  $B$  buckets:

- We use  $B$  buffer pages for output (one for each bucket), and 1 for input
  - For each tuple  $t$  in input, copy to buffer page for  $h_B(t.A)$
  - When page fills up, write to disk.

# HPJ Phase 1: Partitioning

We partition into  $B = 2$  buckets using hash function  $h_2$  so that we can have one buffer page for each partition (and one for input)

Given  $B+1 = 3$  buffer pages



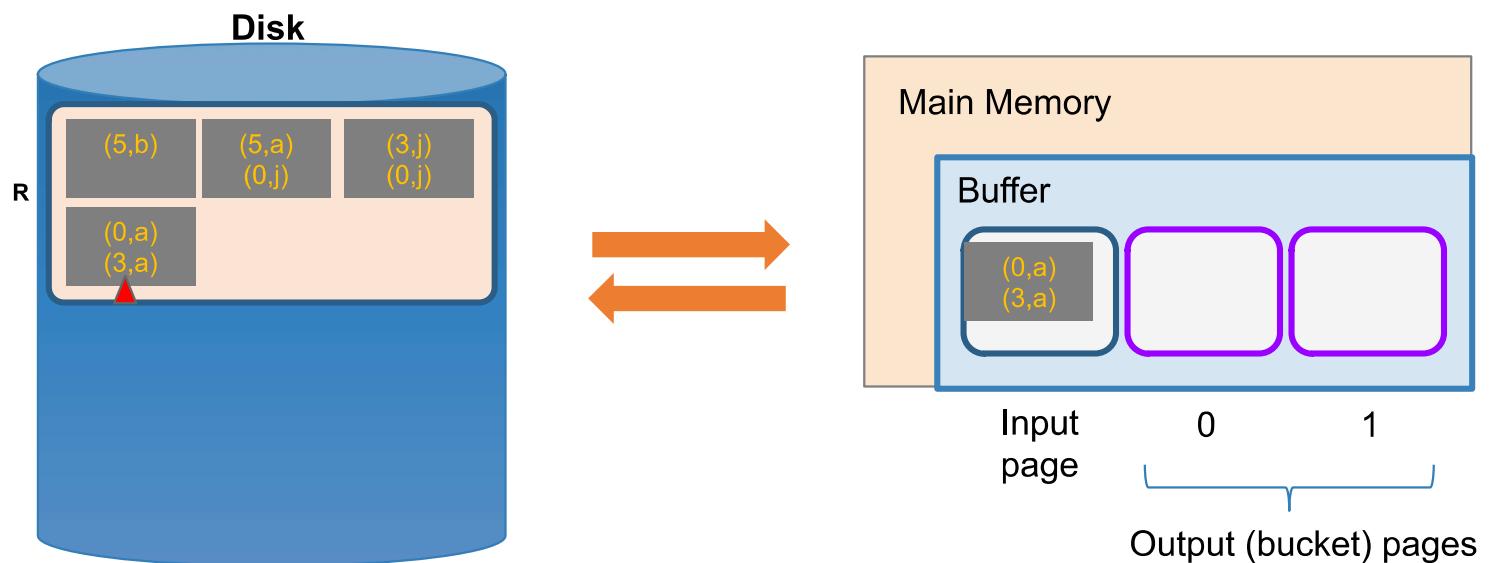
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

Note our new convention for this example: pages each have **two** tuples (one per row)

# HPJ Phase 1: Partitioning

1. We read pages from R into the “input” page of the buffer...

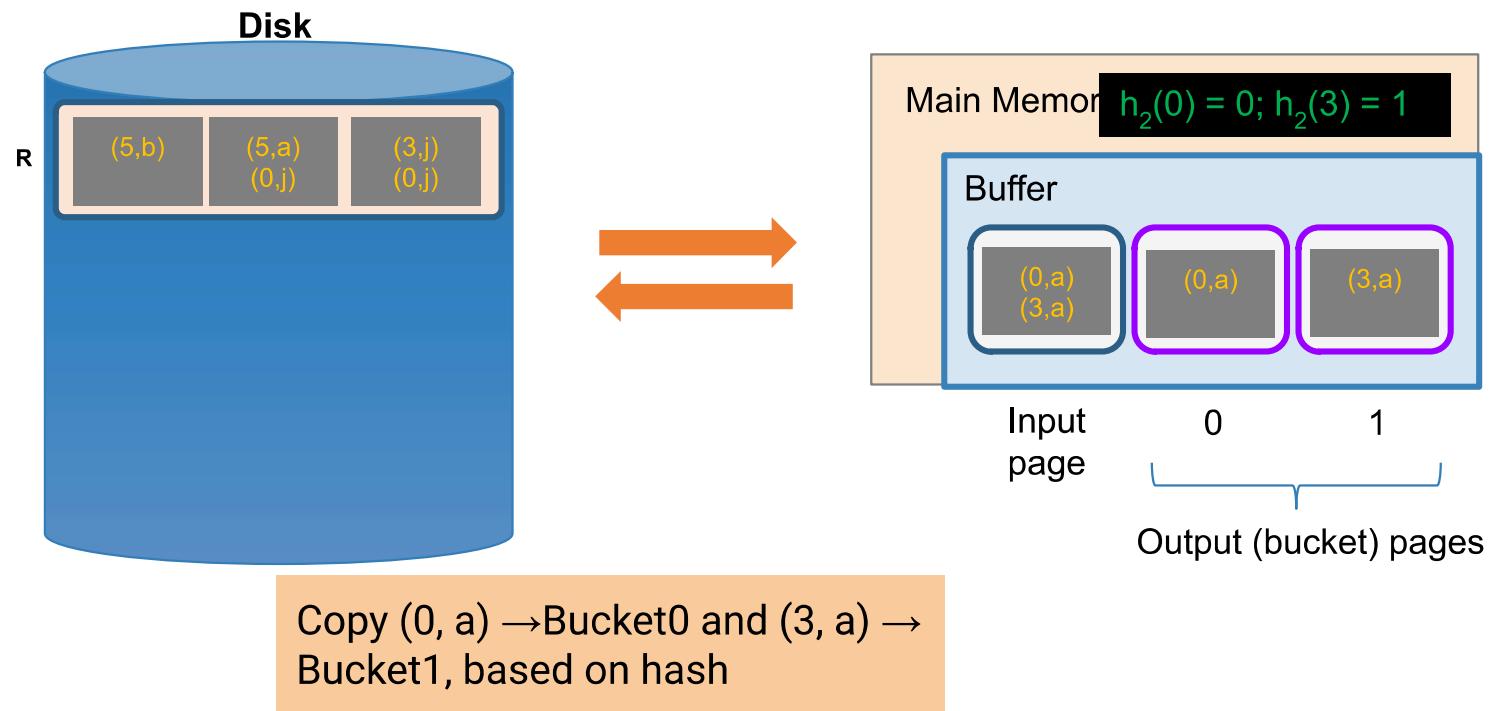
Given  $B+1 = 3$   
buffer pages



# HPJ Phase 1: Partitioning

2. Then we use **hash function  $h_2$**  to sort into the buckets, which each have one page in the buffer

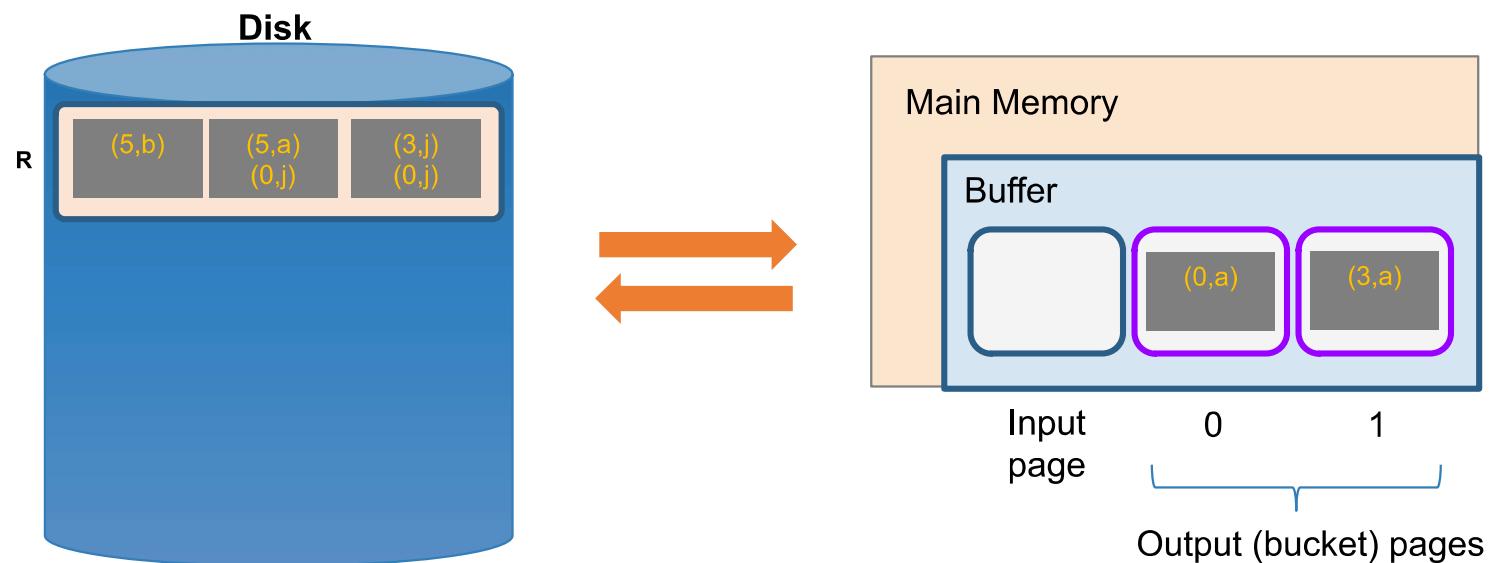
Given  $B+1 = 3$   
buffer pages



# HPJ Phase 1: Partitioning

3. We repeat until the buffer bucket pages are full...

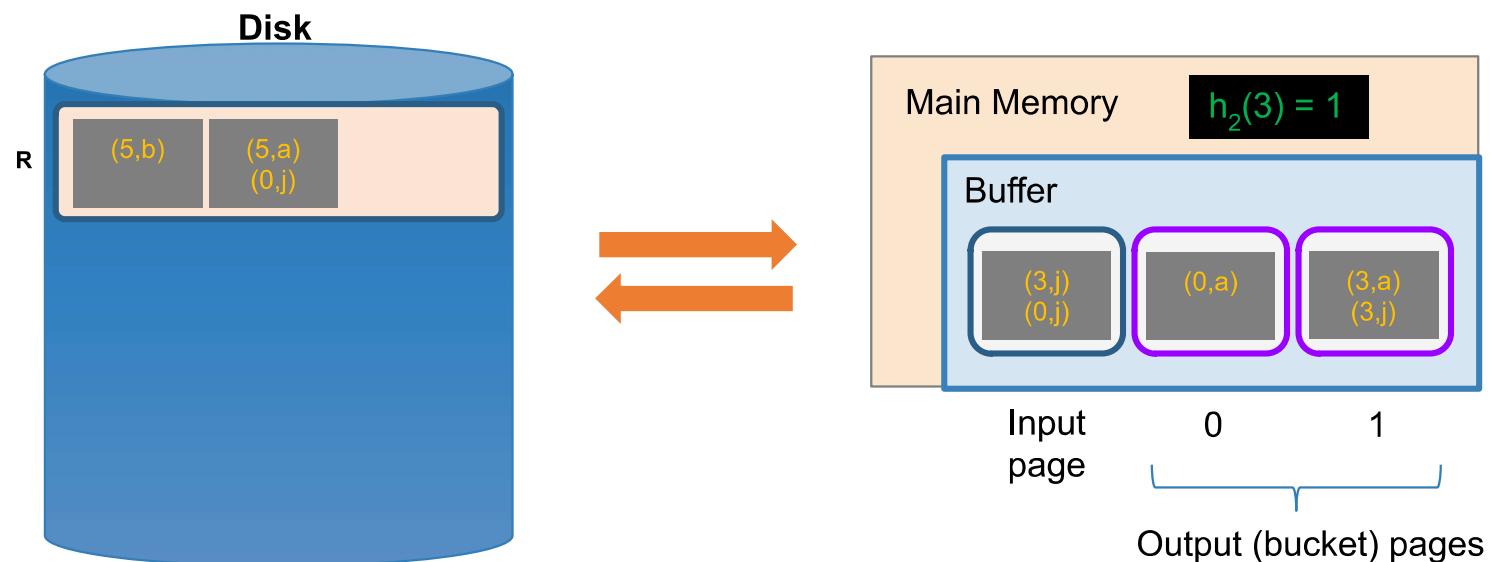
Given  $B+1 = 3$   
buffer pages



# HPJ Phase 1: Partitioning

3. We repeat until the buffer bucket pages are full...

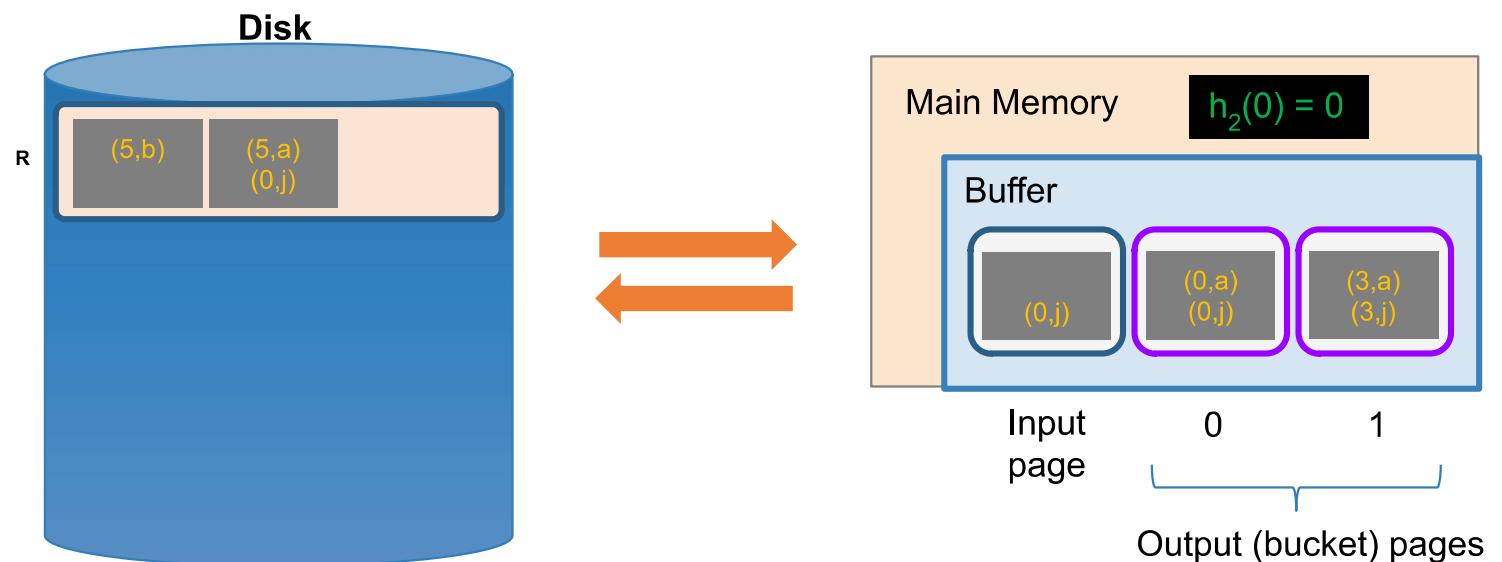
Given  $B+1 = 3$   
buffer pages



# HPJ Phase 1: Partitioning

3. We repeat until the buffer bucket pages are full...

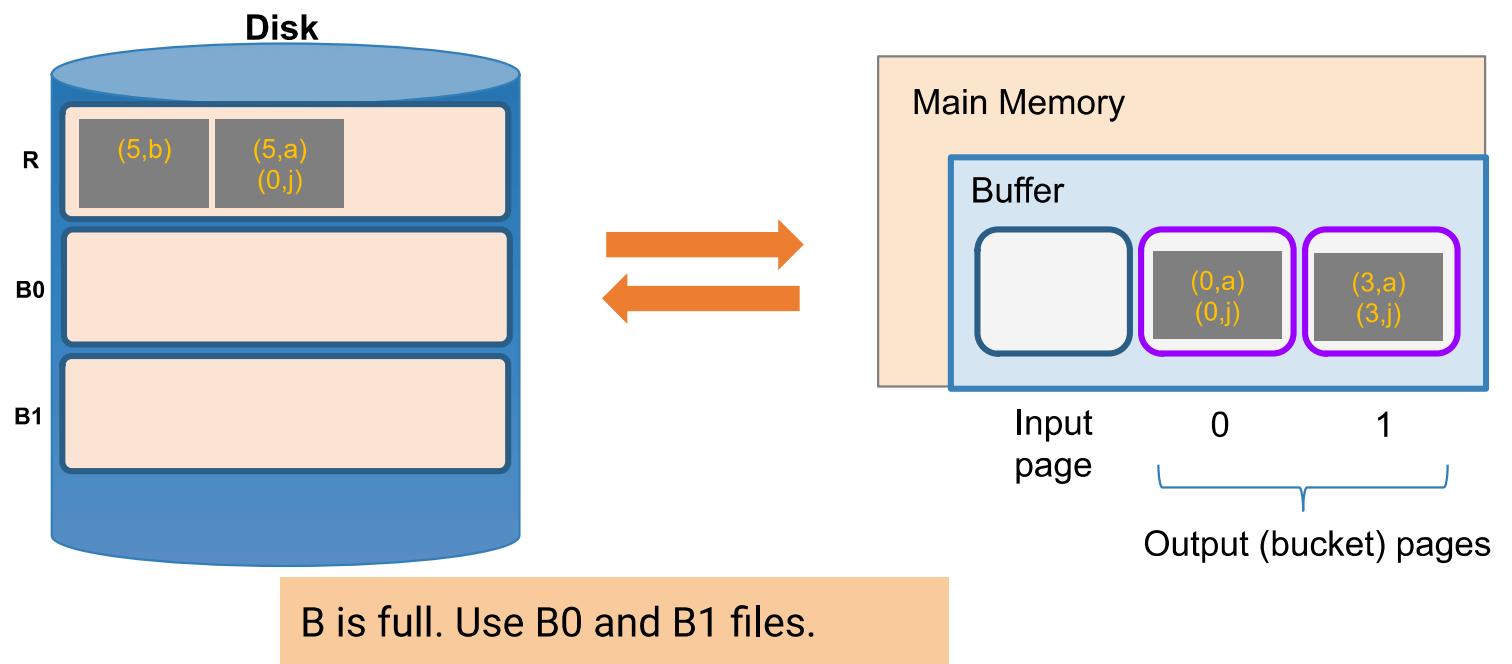
Given  $B+1 = 3$   
buffer pages



# HPJ Phase 1: Partitioning

3. We repeat until the buffer bucket pages are full... then flush to disk

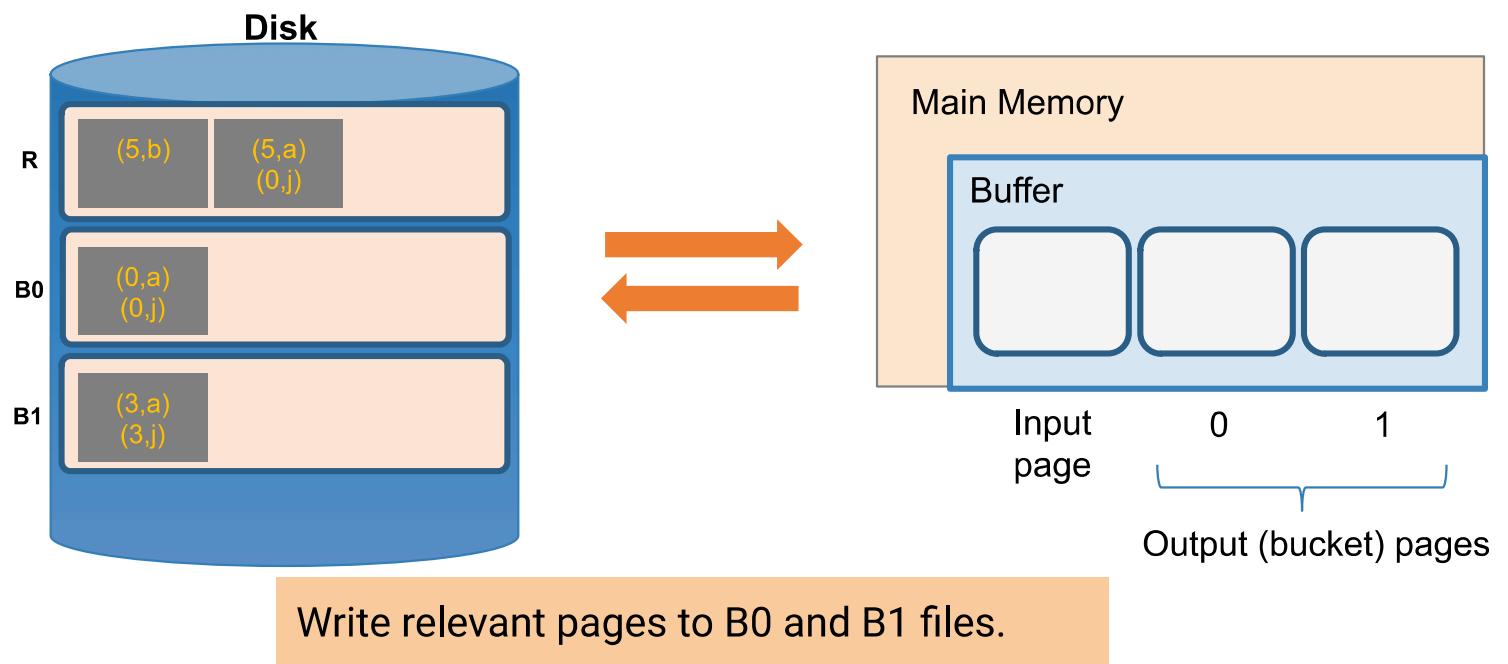
Given  $B+1 = 3$   
buffer pages



# HPJ Phase 1: Partitioning

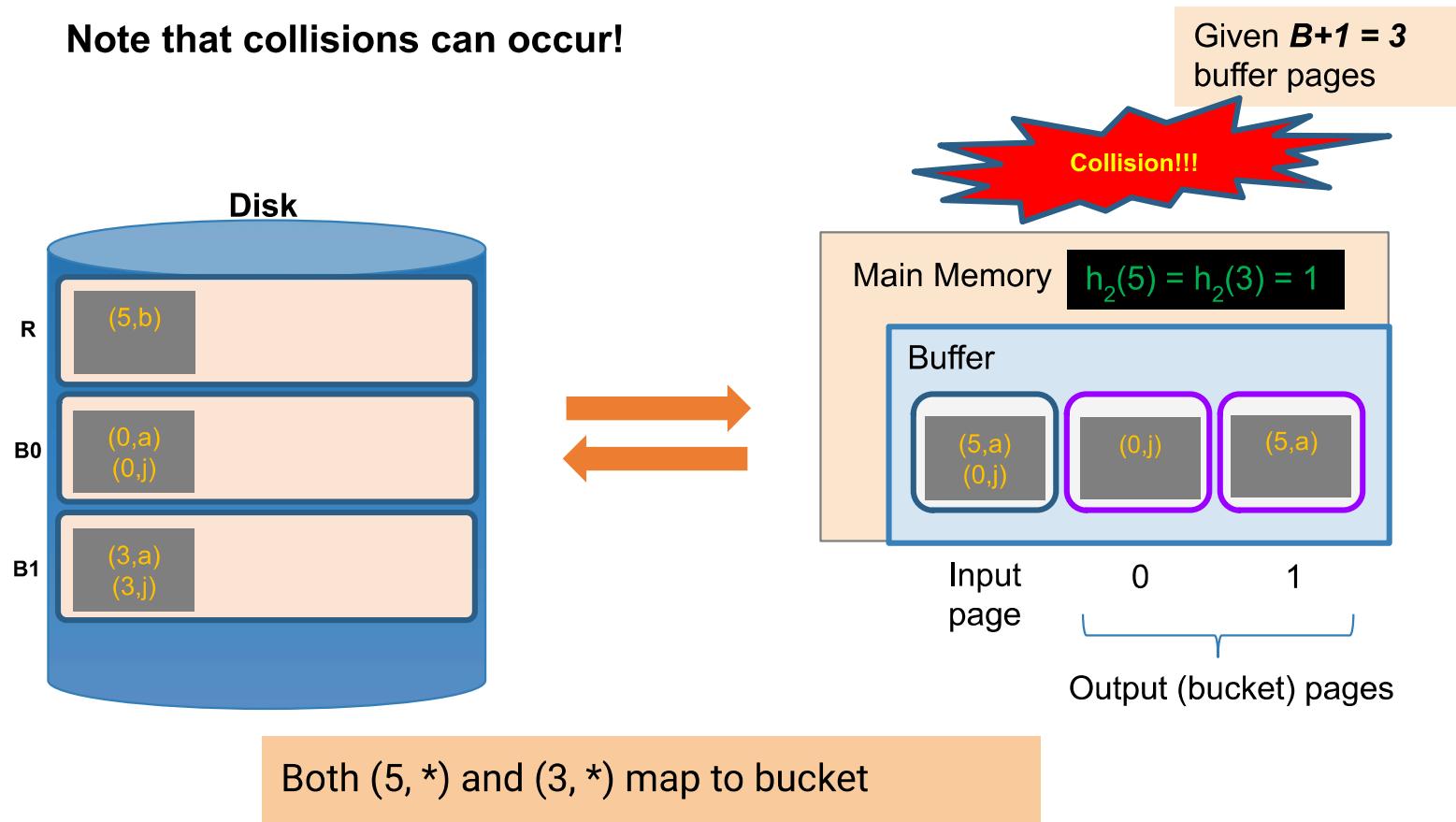
3. We repeat until the buffer bucket pages are full... then flush to disk

Given  $B+1 = 3$   
buffer pages



# HPJ Phase 1: Partitioning

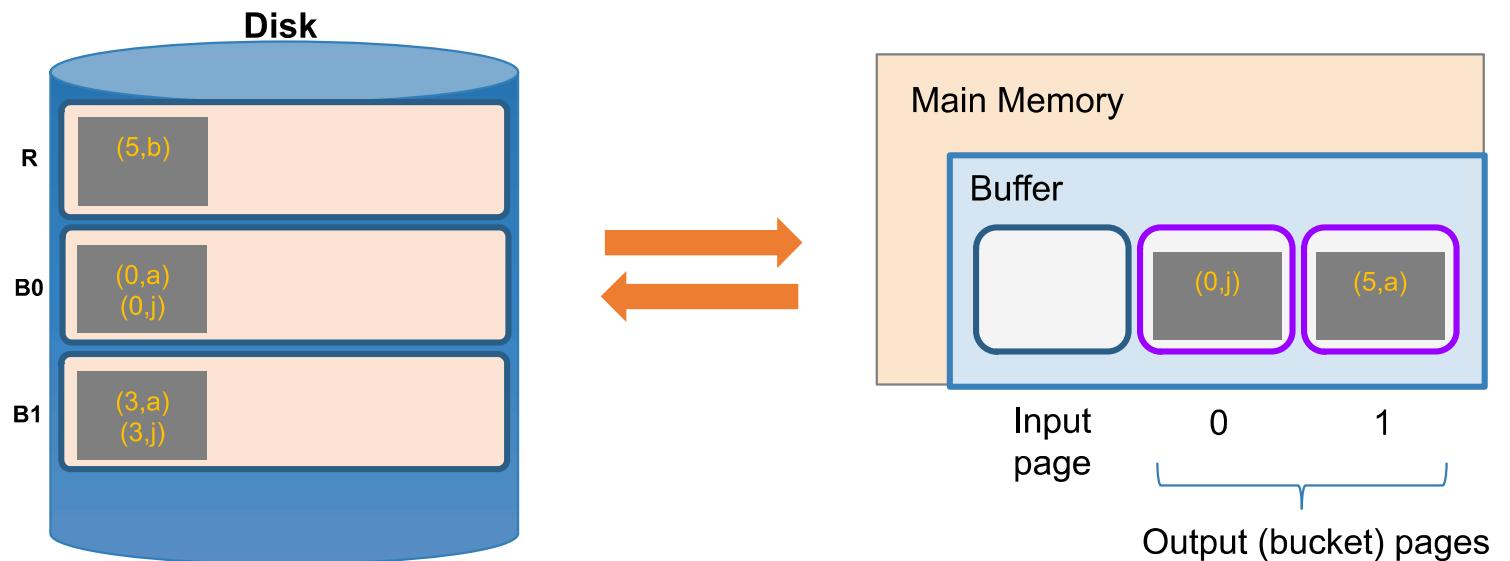
Note that collisions can occur!



# HPJ Phase 1: Partitioning

Finish this pass...

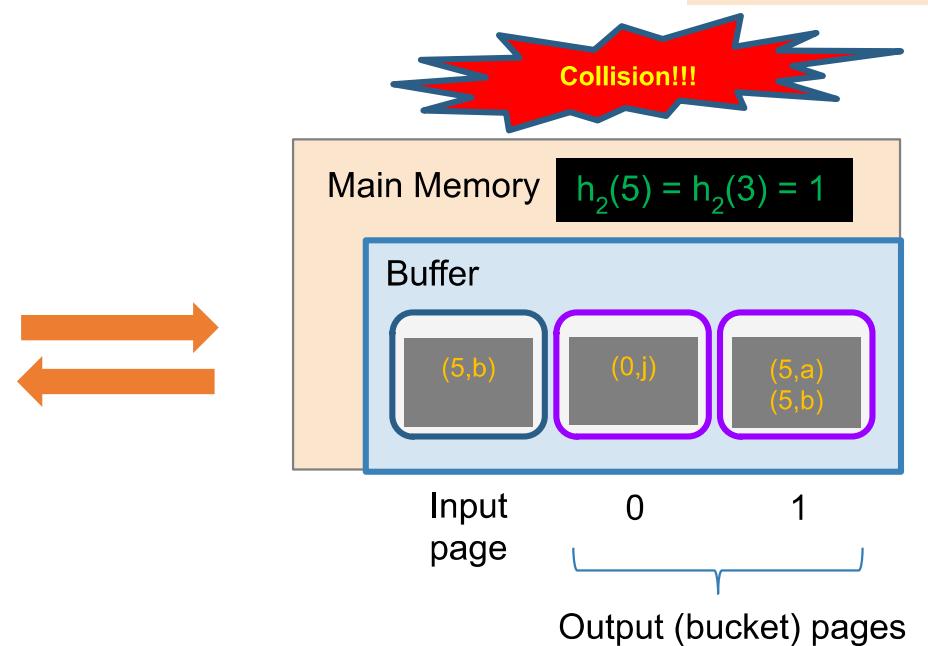
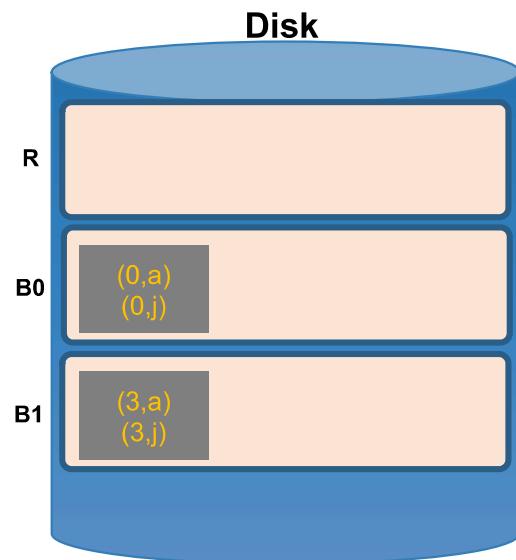
Given  $B+1 = 3$   
buffer pages



# HPJ Phase 1: Partitioning

Finish this pass...

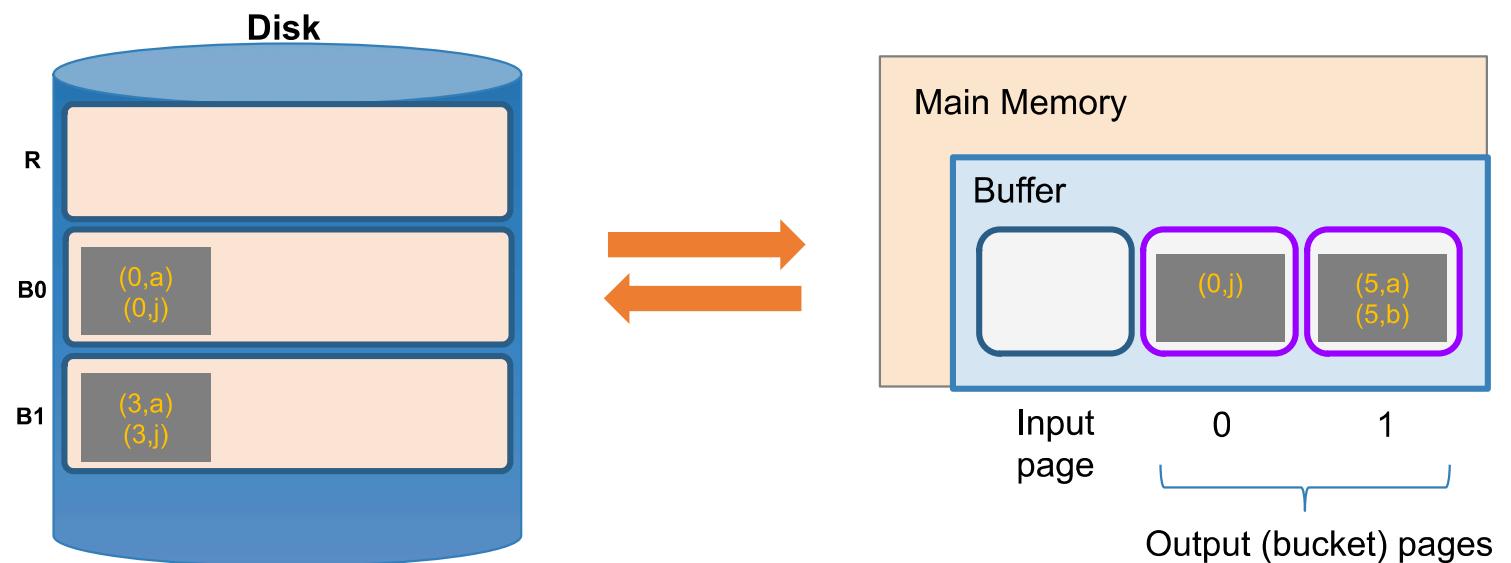
Given  $B+1 = 3$   
buffer pages



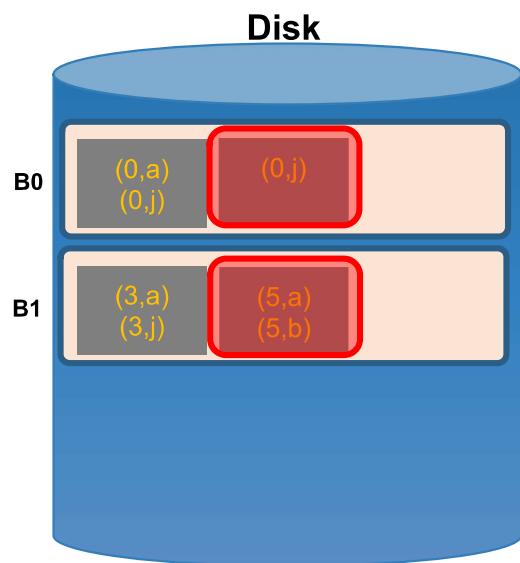
# HPJ Phase 1: Partitioning

Finish this pass...

Given  $B+1 = 3$   
buffer pages



# HPJ Phase 1: Partitioning



We wanted buckets of size  
 $B-1 = 1 \dots$  however we got  
larger ones due to:

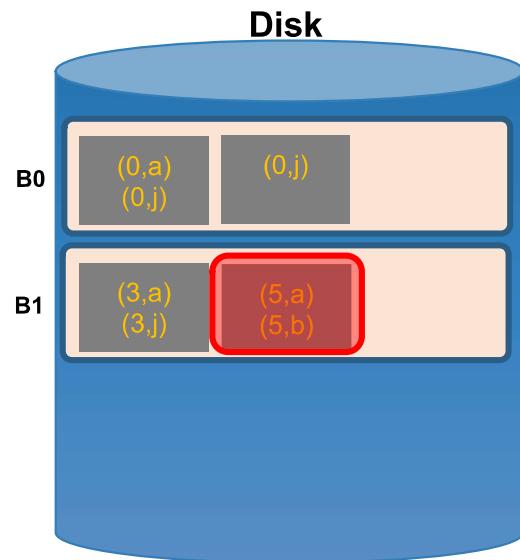
Given  $B+1 = 3$   
buffer pages

(1) Duplicate join keys

(2) Hash collisions

# HPJ Phase 1: Partitioning

Given  $B+1 = 3$   
buffer pages



To take care of larger buckets caused by (2) hash collisions, we can just do another pass!

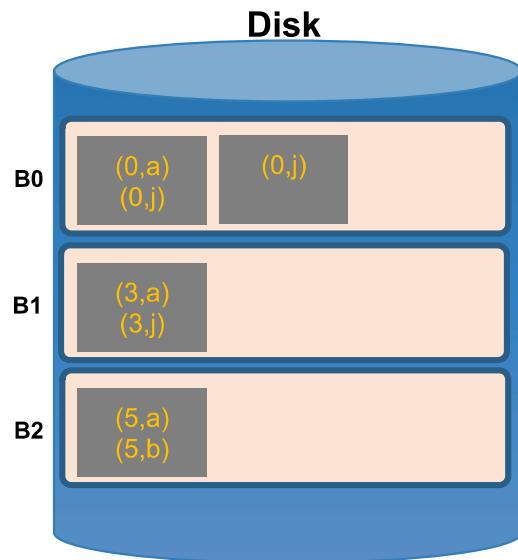
What hash function should we use?

Do another pass with a different hash function,  $h'_2$ , ideally such that:

$$h'_2(3) \neq h'_2(5)$$

# HPJ Phase 1: Partitioning

Given  $B+1 = 3$   
buffer pages



To take care of larger buckets caused by (2) hash collisions, we can just do another pass!

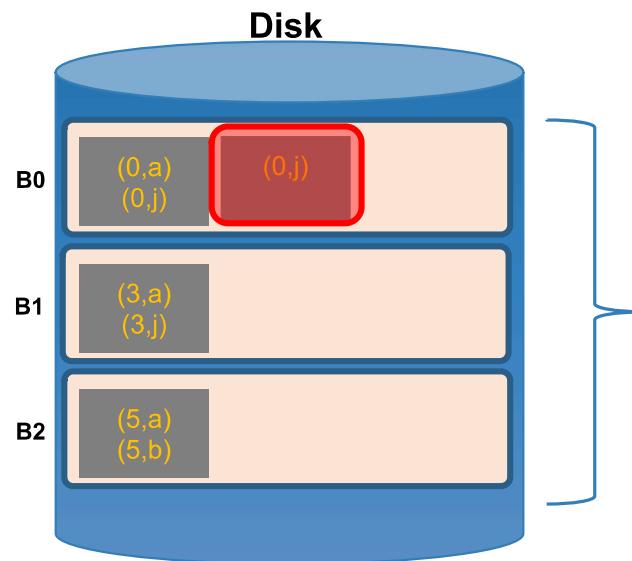
What hash function should we use?

Do another pass with a different hash function,  $h'_2$ , ideally such that:

$$h'_2(3) \neq h'_2(5)$$

# HPJ Phase 1: Partitioning

Given  $B+1 = 3$   
buffer pages



What about duplicate join keys? Unfortunately this is a problem... but usually not a huge one.

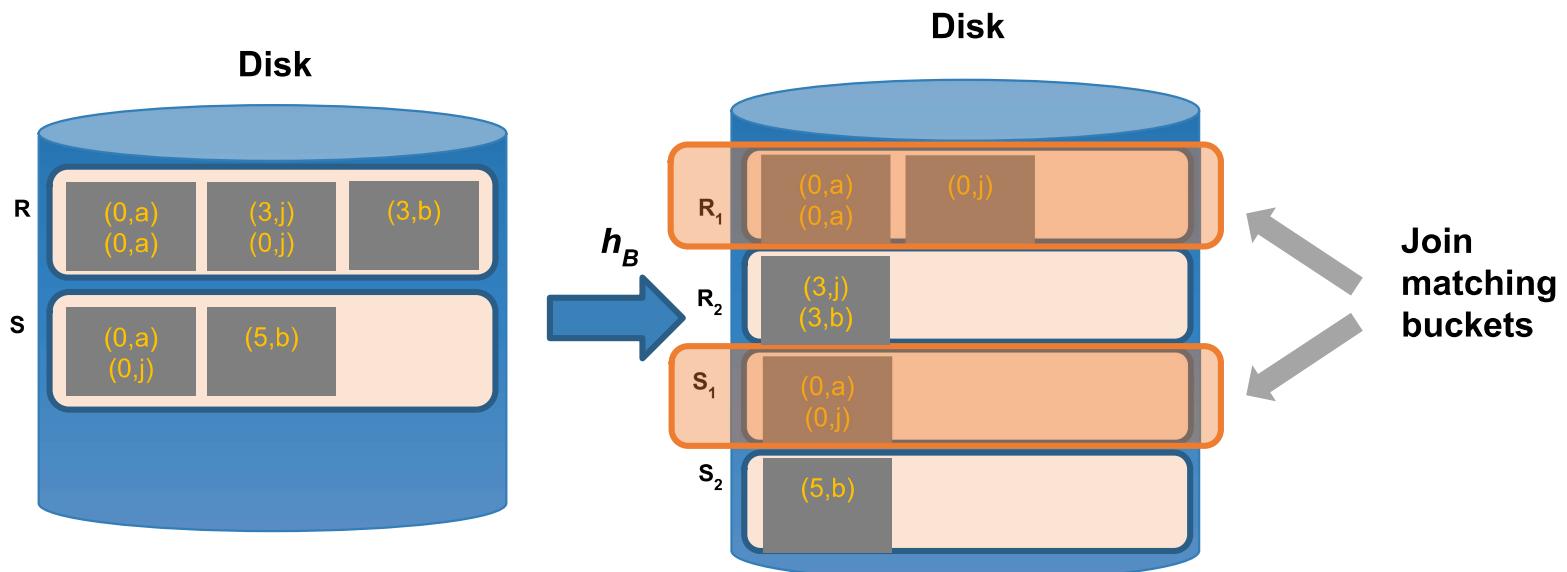
We call this unevenness in the bucket size **skew**



**Now that we have  
partitioned R and S...**

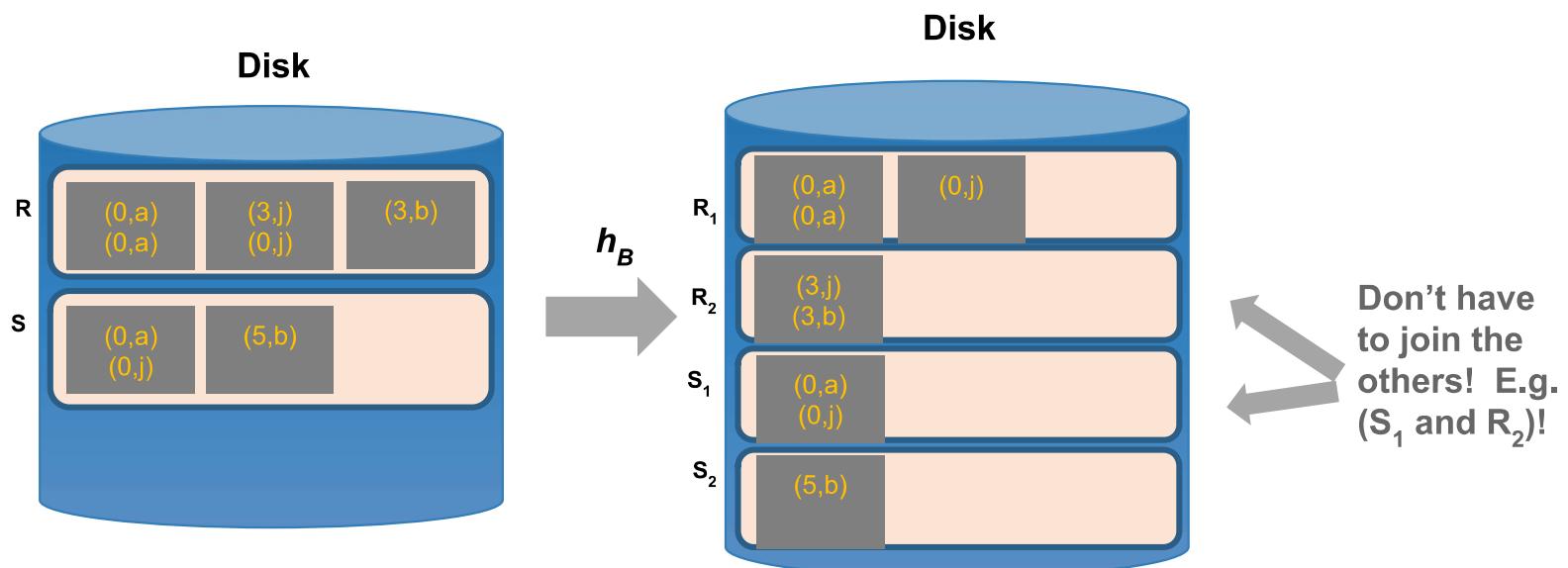
## HPJ Phase 2: Partition Join

Now, we just join pairs of buckets from R and S that have the same hash value to complete the join!



# HPJ: High-level procedure

## 2. Per-Partition Join: JOIN tuples in same partition





# HPJ Summary

*Given enough buffer pages...*

- **Hash Partition** requires reading + writing each page of R,S
  - $\rightarrow 2(P(R)+P(S))$  IOs
- **Partition Join** (with BNLJ) requires reading each page of R,S
  - $\rightarrow P(R) + P(S)$  IOs

HJ takes  $\sim 3(P(R)+P(S)) + OUT$  IOs!

# Join Algorithms: Summary

For  $R \bowtie S$  on column 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.  $\sim O(P(R)+P(S))$

By only supporting equijoins & taking advantage of this structure!

## IO analysis

## Recap

Sorting of relation  $R$  with  $N$  pages (i.e.  $P(R) = N$ )

IO Cost ~=  
(sort  $N$  pages)

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

$\sim 2N$

$\sim 4N$

Sort  $N$  pages with  $B$  buffer size (+1 for output)

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

Sort  $N$  pages when  $N \approx B$

(because  $(\log_B 0.5) < 0$ )

Sort  $N$  pages when  $N \approx 2^B$

(because  $(\log_B B) = 1$ )

SortMerge and HashJoin for  $R$  &  $S$

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

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

$$\sim 1 * (P(R) + P(S)) + OUT$$

Special cases: 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?

# Where we are

## Week 3

- Learnt basics of IO-aware algorithms and indexes

## Week 4

- Problem 1: How to search Amazon's product catalog in < 1sec?
- Problem 2: JOIN queries are slow and expensive.
  - How do we speed up by **1000x?** (Tuesday)
  - How do we speed up by another **100x?** (Tuesday)

## Week 5

- Problem 3: Build a Product Recommendation system (e.g., Amazon's)



# Putting it all together

Systems design



## In this lecture

1. Quick recap of scale
2. A full systems example putting things together
  - ▷ How to build Amazon's Products and Recommendations page

# Big Scale Lego Blocks

## Roadmap



Primary data structures/algorithms

### Hashing

HashTables  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

HashFunctions  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

### Sorting

BucketSort, QuickSort  
MergeSort

### MergeSortedFiles

MergeSort

MergeSort

## IO analysis

## Recap

Sorting of relational T with N pages

IO Cost ~=  
(sort N pages)

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

$\sim 2N$

$\sim 4N$

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)

Sort N pages when  $N \approx B$

(because  $(\log_B 0.5) < 0$ )

Sort N pages when  $N \approx 2*B^2$

(because  $(\log_B B) = 1$ )

SortMerge and HashJoin for R & S

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

$$\sim 1 * (P(R) + P(S)) + OUT$$

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

For SortMerge, if already sorted

For HashJoin, if already partitioned

# Systems Design Example:

## User Cohorts

Billion products

User searches for “coffee machine”



Nespresso Vertuo Coffee and Espresso Machine Bundle with Aeroccino Milk Frother by Breville, Red

by Breville

★★★★★ 980 customer reviews

| 259 answered questions

Amazon's Choice for "nespresso machine red"

List Price: \$249.95

Price: \$189.96 | FREE One-Day

You Save: \$59.99 (24%)

Your cost could be \$179.96. Eligible customers get a \$10 bonus when reloading \$100.

Free Amazon product support included

Style Name: Nespresso by Breville

Nespresso Nespresso by Breville

Color: Red



Customers who viewed this item also viewed these products



Dualit Food XL1500 Processor  
\$560

Add to cart



Kenwood kMix Manual Espresso Machine  
★★★★★  
\$250

Select options



Weber One Touch Gold Premium Charcoal Grill-57cm  
\$225

Add to cart



NoMU Salt Pepper and Spice Grinders  
\$3

View options



SMJs then  
GROUP BYs?

```
// Query: Demographic split of users, by product interest
SELECT      ... , count (*) ...
FROM        Product p, UserViews u
WHERE       p.productid = u.productid
GROUP BY    u.age, u.gender, u.city
```

In SMJ, what do you sort/merge on?

- ▶ (a) `productid`? (b) `<u.age, u.gender, u.city >`?



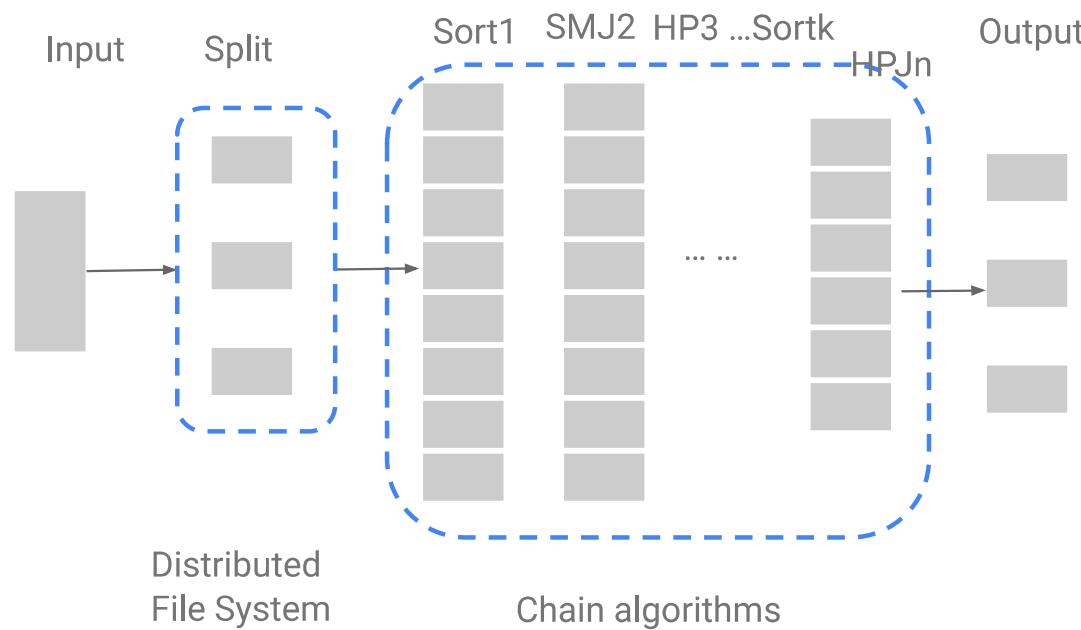
# Algorithms can be chained (like functions)

(e.g., SMJ, Sorts (aka ExternalSorts), HPs, HPJs)

- ▶ Step1: SMJ( $p.\text{productid} = u.\text{productid}$ ). Specifically,
  - a. Sort(Product by  $\text{productid}$ ), Sort(UserViews by  $\text{productid}$ )
  - b. MergeJoin and produce Temp=< $\text{productid}, \text{u.age}, \text{u.gender}, \text{u.city}$ >
- ▶ Step2: // Handle groupbys
  - a. Sort(Temp, < $\text{u.age}, \text{u.gender}, \text{u.city}$ >)
  - b. Scan sortedTemp, and make GROUPBYs for  $\text{u.age}, \text{u.gender}, \text{u.city}$
  - c. [Notice: in Step2(a), you could also use HP(Temp, by < $\text{u.age}, \text{u.gender}, \text{u.city}$ >)]
- ▶ Step3: Further process... (e.g., for indexing or for a sub-select, etc.)

Chaining Sorts,  
SMJs, HPs

## General Composition



## Special cases

1. **MapReduce:** Input → [Split → HP(keys) → HJP] → Output
2. **Shuffle:** HP (e.g., on next step's GROUP BYs)
  - a. Typical optimization – save time by using only RAM, without writing OUT to HDD/SSD.



1. Quick recap of scale
2. A full systems example putting things together
  - ▷ How to build Amazon's Products and Recommendations page

# Systems Design Example:

## Product Search & CoOccur

### Billion products

User searches for “coffee machine”



Nespresso Vertuo Coffee and Espresso Machine Bundle with Aeroccino Milk Frother by Breville, Red

by Breville  
★★★★★ 980 customer reviews  
259 answered questions

Amazon's Choice for "nespresso machine red"

List Price: \$249.95  
Price: \$189.96 prime | FREE One-Day  
You Save: \$59.99 (24%)

Your cost could be \$179.96. Eligible customers get a \$10 bonus when reloading \$100.

Free Amazon product support included

Style Name: Nespresso by Breville

Nespresso | Nespresso by Breville

Color: Red



### Product recommendations

Customers who viewed this item also viewed these products



Dualit Food XL1500 Processor  
\$560

Add to cart



Kenwood kMix Manual Espresso Machine  
★★★★★ 250  
\$250

Select options



Weber One Touch Gold Premium Charcoal Grill-57cm  
\$225

Add to cart



NoMU Salt Pepper and Spice Grinders  
\$3

View options

# Systems Design Example:

## Product Search & CoOccur

### Counting popular product-pairs

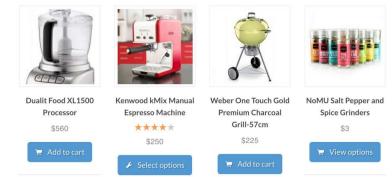
Story: Amazon/Walmart/Alibaba (AWA) want to sell products

1. Problem1: AWA wants fast user searches for product
  2. Problem2: AWA shows 'related products' for all products
    - Using collaborative filtering ('wisdom of crowds') from historical website logs.
    - Each time a user views a set of products, those products are related (co-occur)
- ⇒ Goal: compute product pairs and their co-occur count, across all users

#### Data input:

- AWA has **1 billion products**. Each product record is ~1MB (descriptions, images, etc.).
- AWA has **10 billion UserViews** each week, from 1 billion users. Stored in **UserViews**, each row has <userID, productID, viewID, viewTime>.

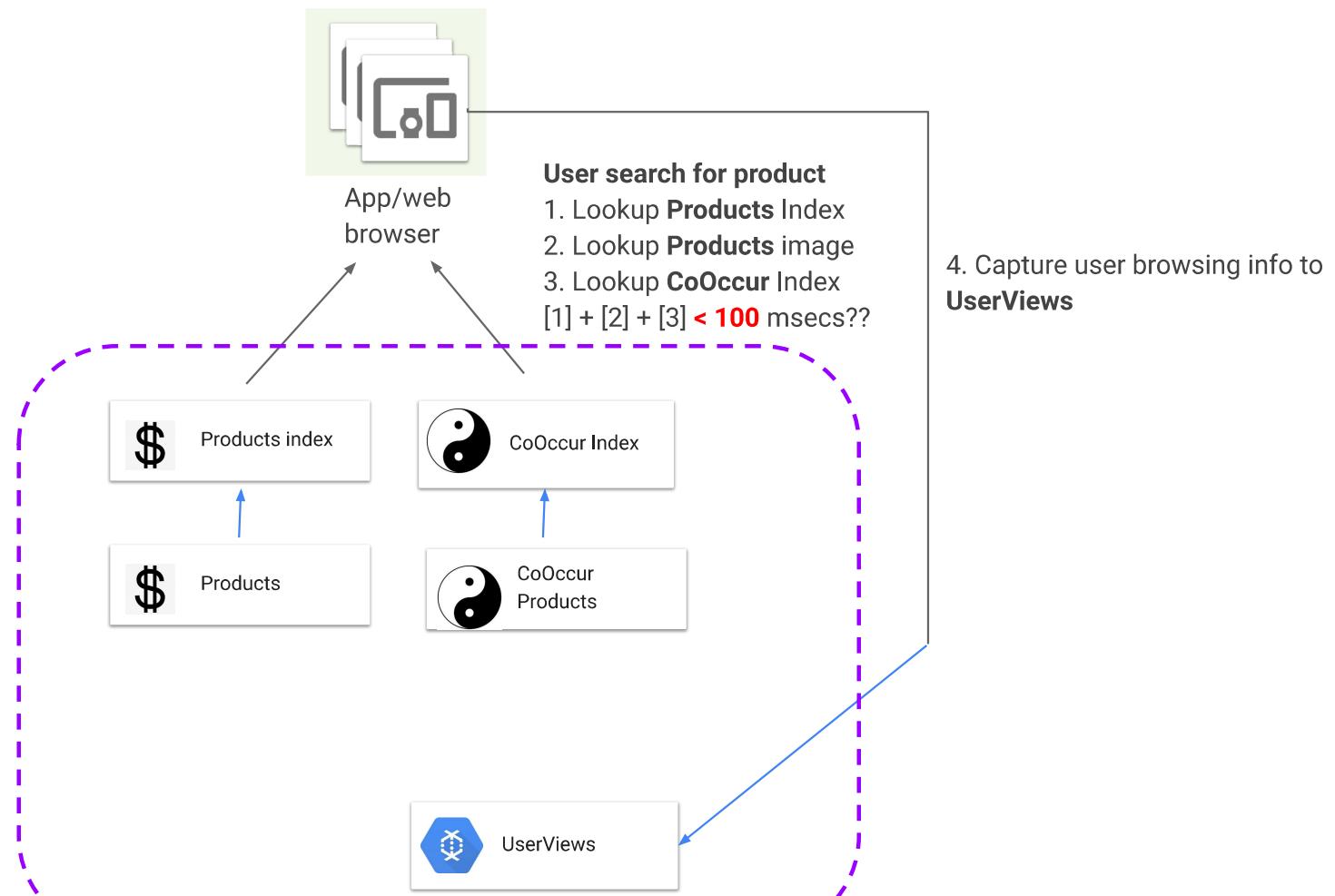
Customers who viewed this item also viewed these products



# Data Systems Design Example:

## Product Search & CoOccur

### Goal



# Plan #1 Counting in RAM

Counting product views for billion product PAIRs

UserViews(UserId, ProductID, ..., ...)

	Nespresso Coffee
	Bread maker
	Kenwood Espresso
	...

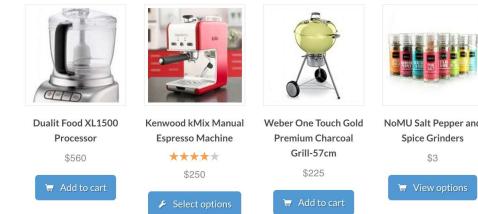
CoOccur(ProductID1, ProductID2, count)

Nespresso Coffee	Bread Maker	301
Bread maker	Kenwood Espresso	24597
Kenwood Espresso	Bike	22
		...

Algorithm: For each user, product  $p_i$   $p_j$   
 $\text{CoOccur}[p_i, p_j] += 1$

Idea: If  $\text{CoOccur}[p_i, p_j] > 1 \text{ million}$  (e.g.), 'related.' Classic idea. Tweak for AI inferencing/personalization

Customers who viewed this item also viewed these products



Pre-cs145

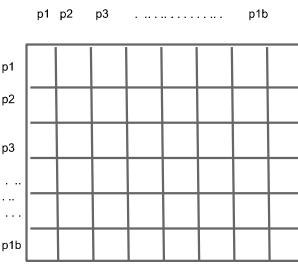
Compute in  
RAM

(Engg approximations)

### Counting product views

Input size (4 bytes for user, 4 bytes for productid)	$\sim 1\text{Bil} * [4 + 4] = 8 \text{ GB}$
Output size (4 bytes for productid, 4 bytes for count)	$\sim 1\text{Bil} * [4 + 4] = 8 \text{ GB}$

Trivial



### Counting product pair views

Plan #1:  $\sim P * P/2$  matrix for counters in RAM (4 bytes for count)

- RAM size =  $1 \text{ Billion} * 1\text{Billion}/2 * 4 = 2 \text{ Million TBs}$ 
  - $// (count(a, b) = count(b, a))$
- Trivial? If you have **~\$6 Billion** (RAM at  $\sim \$100/32\text{GB}$  RAM)

Plan #1 (on disk): Let OS page into memory as needed

- Worst case #1 > 1 million years on HDD  
(if you seek to update each counter)

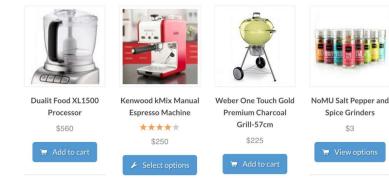
**⇒ Need to do some design work, with cs145**

# Systems Design Example:

## Product CoOccur

### Counting popular product-pairs

Customers who viewed this item also viewed these products



Your mission: Compute a table – CoOccur <productID1, productID2, count>, with co-occur counts from last week's data

1. AWA's data quality magicians recommend
  - a. Keep only top three recommendations per product, and (b) drop pairs with co-occur counts less than million.
  - b. On avg, users view **ten products each week** (*User is interested in ~10 products/week, not 1000s*). (For compactness, using u1 for a userID, and p1..p10 for productids)
    - $\Rightarrow \{<\text{u1}, \text{p1}, \dots>, <\text{u1}, \text{p2}, \dots>, <\text{u1}, \text{p3}, \dots> \dots <\text{u1}, \text{p10}>\}$ ; 10 rows/user in UserViews
    - We'd need to count the following  $10*9/2$  ( $n$  choose 2) product pairs, per u1
      - $<\text{p1}, \text{p2}>, <\text{p1}, \text{p3}>, \dots <\text{p1}, \text{p10}>$
      - $<\text{p2}, \text{p3}>, <\text{p2}, \text{p4}> \dots <\text{p2}, \text{p10}>$
      - $<\text{p3}, \text{p4}>, <\text{p3}, \text{p5}>, \dots <\text{p3}, \text{p10}>$
      - ...
      - $<\text{p9}, \text{p10}>$
    - Alternately, we say the blowup-factor = 4.5 (=45/10 for the cross-product)
      - (i.e.. for 'k' avg views, the blowup-factor =  $(k-1)/2$ )

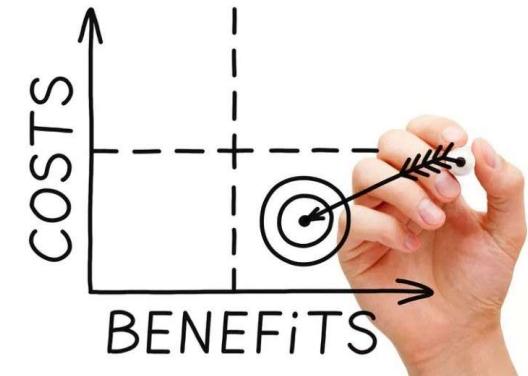
# Optimize,

# Evaluate design plan 1, plan2, plan 3, ...



Build Query Plans

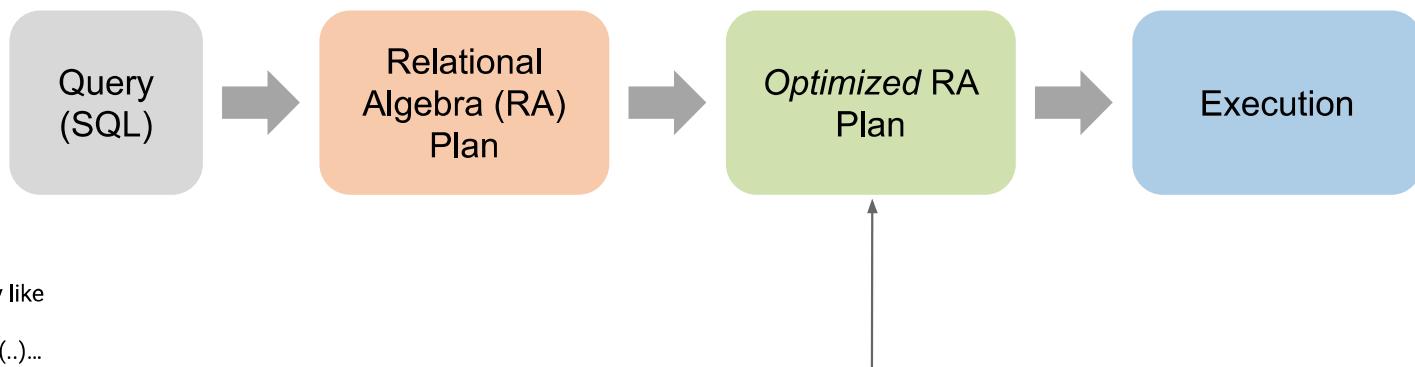
1. For SFW, Joins queries
  - a. Sort? Hash? Count? Brute-force?
  - b. Pre-build an index? B+ tree, Hash?
2. What statistics can I keep to optimize?
  - a. E.g. Selectivity of columns, values



Analyze Plans

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

# Post CS 145



```
SELECT TOP(..)  
FROM UserViews v1, UserViews v2  
WHERE ...  
GROUP BY v1.productId, v2.productId  
HAVING count(*) > 1 million
```

## Query plan#2

1. **SMJ**(UserViews, UserViews) on **userid**
2. **Sort**(SortedUsersViews) on <**v1.productId, v2.productId**> for group by & HAVING

# Systems Design Example:

## Product CoOccur

### Pre-design

	Size	Why?
ProductId	4 bytes	1 Billion products $\Rightarrow$ Need at least 30 bits ( $2^{30} \approx 1$ Billion) to represent each product uniquely. So use 4 bytes.
UserID	4 bytes	"
ViewID	8 bytes	10 Billion product views.
Products	$P(\text{Product}) = 15.625\text{M pages}$	1 Billion products of 1 MB each. Size = <b>1 PB</b> = <b>15.625M pages</b> $\text{NumSearchKeys}(\text{Product})=1$ Billion
UserViews	$P(\text{UserViews}) = 3750 \text{ pages}$	Each record is <userID, productID, viewID, viewtime>. Assume: 8 bytes for viewTime. So that's 24 bytes per record. $10 \text{ Billion} * [4+4+8+8] \text{ bytes} = 240 \text{ GBs} = 3750 \text{ Pages.}$
ProductPairs for all userViews (We'll see how to use this in 3 slides)	<b>5625 pages</b>	Given: blowup-factor = 4.5, and 10 Billion UserViews. Each record is <productID, productID>. At 8 bytes per record, $45 \text{ Billion} * 8 \text{ bytes} = 360 \text{ GBs} = 5625 \text{ Pages.}$
CoOccur (for top 3 Billion)	$P(\text{CoOccur}) = 563 \text{ pages}$	The output should be <productID, productID, count> for the co-occur counts. That is, 12 bytes per record (4 + 4 + 4 for the two productIDs and 4 bytes for count). For three billion product pairs, you need $3 \text{ billion} * 12 \text{ bytes} = 36 \text{ GBs} = 563 \text{ pages.}$ $\text{NumSearchKeys}(\text{CoOccur}) = 1$ Billion.

## Systems Design Example:

## Product CoOccur

## Plan #2

# Computing CoOccurs [B = 3000]

- ▶ Step1: SMJ( $v1.\text{userid} = v2.\text{userid}$ ). Specifically,
  - a. SMJ(UserViews, UserViews by  $\text{userid}$ )
    - // Note: Sort and merge UserViews **once**
    - // Normally, SMJ Cost  $\sim= 3*(P(R)+P(S))$ .
    - // So SMJ Cost\*  $\sim= 3*P(\text{UserViews})$ , due to self-join in UserViews.
    - // You could optimize further. Let's use above IO Costs for simplicity
- ▶ Step2: // Handle groupbys
  - a. Sort(Temp,  $\langle v1.\text{productid}, v2.\text{productid} \rangle$ )
  - b. Scan sortedTemp, and make GROUPBYs for  $\langle v1.\text{productid}, v2.\text{productid} \rangle$

# Systems Design Example:

## Product CoOccur

### Plan #2

[See [Notebook](#) for impact of B values]

## Engg Cost Approximation [B = 3000]

Typo: In 2[b], should be Sort(Temp), not Sort(UserViews).

- ▶ Step1:  $\text{SMJ}(v1.\text{userid} = v2.\text{userid})$ . Specifically,
  - a.  $\text{SMJ}(\text{UserViews}, \text{UserViews by } \text{userid})$ 
    - // Note: Sort and merge UserViews **once**
    - // Normally, SMJ Cost  $\sim= 3*(P(R)+P(S))$ .
    - // So SMJ Cost  $\sim= 3*P(\text{UserViews})$ , due to self-join in UserViews.
    - OUTPUT Temp= $\langle v1.\text{productid}, v2.\text{productid} \rangle$
  - IO Cost =  $\text{SMJ}(\text{UserViews}, \text{UserViews})$   
 $\sim= 3*P(\text{UserViews}) + P(\text{Temp})$
- ▶ Step2: // Handle groupbys
  - a.  $\text{Sort}(\text{Temp}, \langle v1.\text{productid}, v2.\text{productid} \rangle)$
  - b. Scan sortedTemp, and make GROUPBYS for  $\langle v1.\text{productid}, v2.\text{productid} \rangle$ 
    - IO Cost =  $\text{Sort}(\text{Temp})$   
 $\sim= 2*P(\text{Temp}) + P(\text{CoOccur})$

# Systems Design Example:

## Product CoOccur

## Plan #2

[See [Notebook](#) for impact of B values]



$$P(\text{UserViews}) = 3750$$

$$P(\text{Temp}) = 5625$$

$$P(\text{CoOccur}) = 563$$

	Access Latency (secs)	Scan Throughput (GBs/sec)	What you get for ~100\$ (Jun '22)
RAM (D-RAM)	~100 nanosec	~100 GB/sec	32 GBs
High-end SSD	~10 microsec	~5 GB/sec	640GB
HDD Seek (Hard Disk)	~10 msec	~100 MB/sec	4 TB
Machines M1 to M2 (Network)	~ 1 microsec	~ 5 GB/sec	N/A

Steps	Cost (IOs)	Why?
SMJ(UserViews, UserViews) on UserID	$3*3750 + 5625$	$3*P(\text{UserViews}) + P(\text{Temp})$
Sort, GroupBY -> CoOccur	$2*5625 + 563$	$2*P(\text{Temp}) + P(\text{CoOccur})$
Total IO cost	28688 IOs	

Recall: HDD Scan at 100 MBps, Access = 10 msec

- Time = Access ( $28688 * 0.01$  secs) + Scan ( $28688 * 64\text{MB} / 100\text{MBps}$ )  
 $\approx 18.647 \text{ secs}$
- Total cost  $\approx \$250$ 
  - Cost (B = 64GB RAM at 100\$/32GB) = ~200\$
  - Cost(HDD = 2 TB at 100\$/4 TB) ~= 50\$

⇒ Could optimize by ~2x with more tweaks. Good enough for cs145. Went from 6B\$ or 1 million years to above)

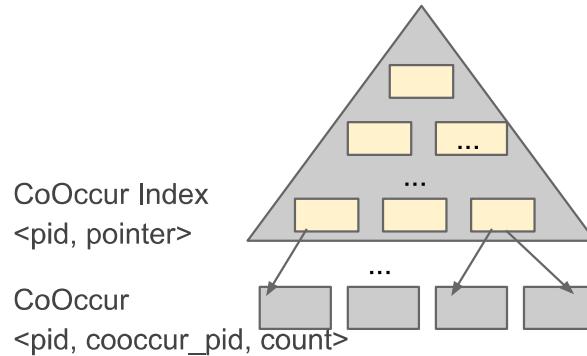
# Systems Design Example:

## Product CoOccur

### B+ tree index

Build indexes with search key=productId. (Assume: data not clustered)

[[Typo Fixed Oct26 'h' from 1 to 2]]



$\text{NumSearchKeys(CoOccur)} = 1 \text{ Billion}$   
 $P(\text{CoOccur}) = 563$

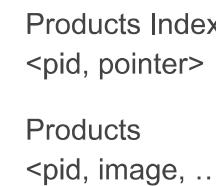
What's f and h?

- $f = 5.6 \text{ million}$   
 $(4 \text{ byte productId} + 8 \text{ byte pointer} @ 64\text{MB/page})$
- $h = 2$

Cost of lookup? (Worst case, with only root in RAM)

- 1 IO (for 1st level) + 1 IO for CoOccur data  
 (assume: clustered for 3 recommendations)
- **2 IOs**

$f \leq \frac{\text{PageSize}}{\text{SKeySize} + \text{PointerSize}}$  // Fit upto 'f' (SKey, Pointer)  
 $f^h \geq \text{NumSKeys}$  // Leaf nodes should point to all SKeys  
 $h \geq \log_f \text{NumSKeys}$  // From previous equation



$\text{NumSearchKeys(Products)} = 1 \text{ Billion}$   
 $P(\text{Products}) = 15.6M$

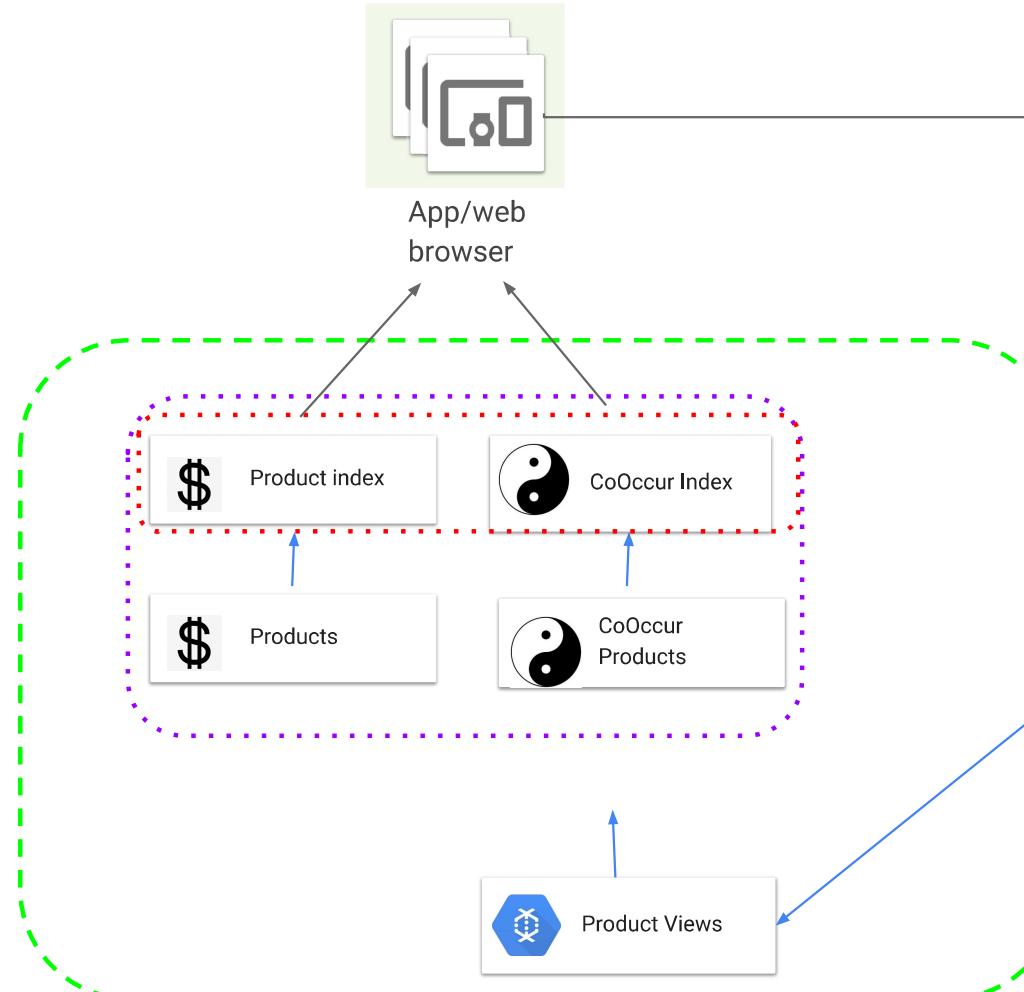
What's f and h?

- $f = 5.6 \text{ million}$   
 $(4 \text{ byte productId} + 8 \text{ byte pointer} @ 64\text{MB/page})$
- $h = 2$

Cost of lookup? (Worst case, with only root in RAM)

- 1 IO (for 1st level) + 1 IO for data
- **2 IOs**

# Data Systems Design Example: Product CoOccur



## User latency?

1. Lookup Product Index
  2. Lookup Products image
  3. Lookup CoOccur Index
- [1] + [2] + [3] < 100 msecs

Ans:

1. 2 IOs for [1] + [2]
2. 2 IOs for [3]
3. (+ ~3 \*2 IOs for the 3 recommendations)

Overall,  $\approx 40$  (+60) msecs even with HDD

- In RAM
- Fast access
- In datacenter

# Data Systems Design

## Popular Systems design pattern

1. Efficiently compute 'batch' of data (sort, hash, count)
2. Build Lookup index on result (b+ tree, hash table)
3. For 'streaming' data, update with 'micro batches'

## Popular problems

1. Related videos (youtube), people (Facebook), pages (web)
2. Security threats, malware (security), correlation analysis



# Histograms & IO Cost Estimation

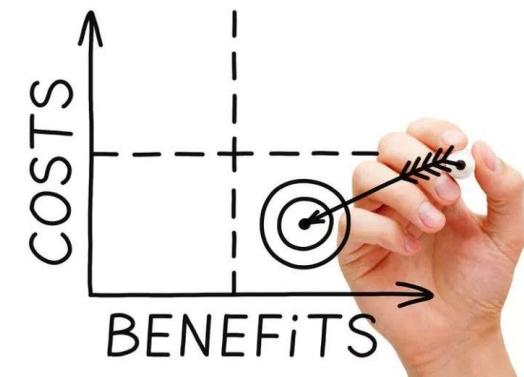
# Optimization

## Roadmap



Build Query Plans

1. For SFW, Joins queries
  - a. Brute-force? Sort? Hash? Count?
  - b. Pre-build an index? B+ tree, Hash?
2. What **statistics** can I keep to optimize?
  - a. E.g. Selectivity of columns, values

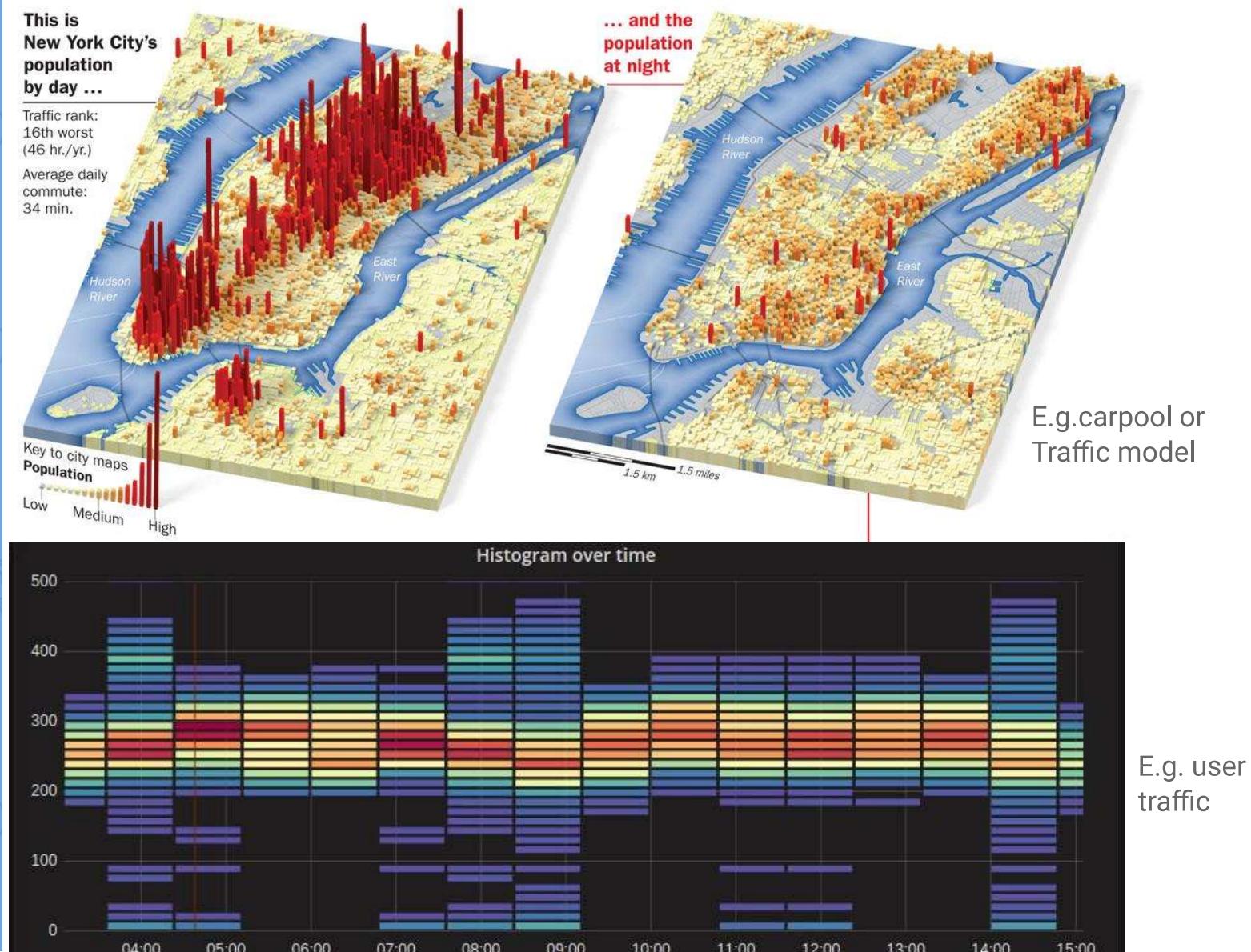


Analyze Plans

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

# Example

## Stats for spatial and temporal data



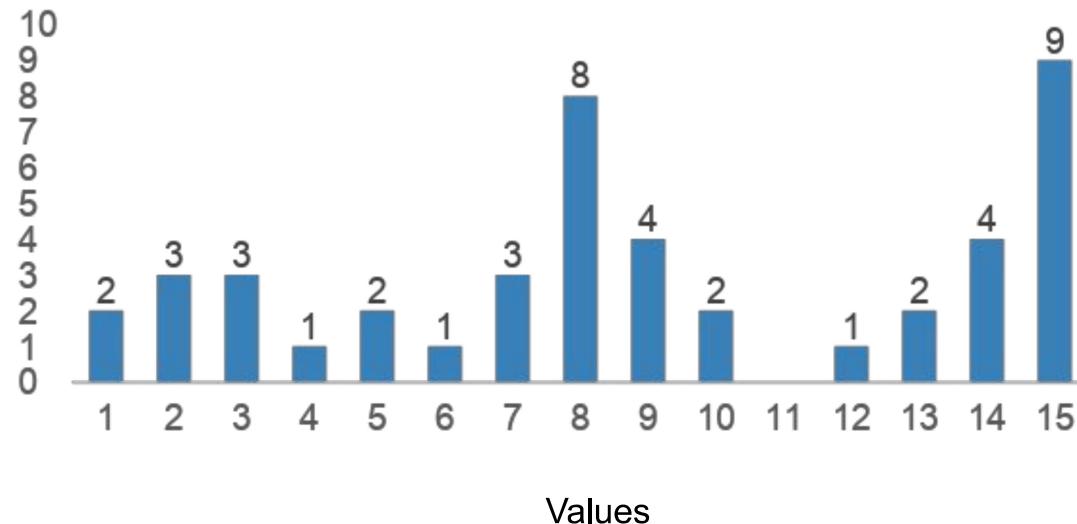


# Histograms

- A histogram is a set of value ranges (“buckets”) and the frequencies of values in those buckets
- How to choose the buckets?
  - Equi-width & Equi-depth
- High-frequency values are very important(e.g, related products)

# Example

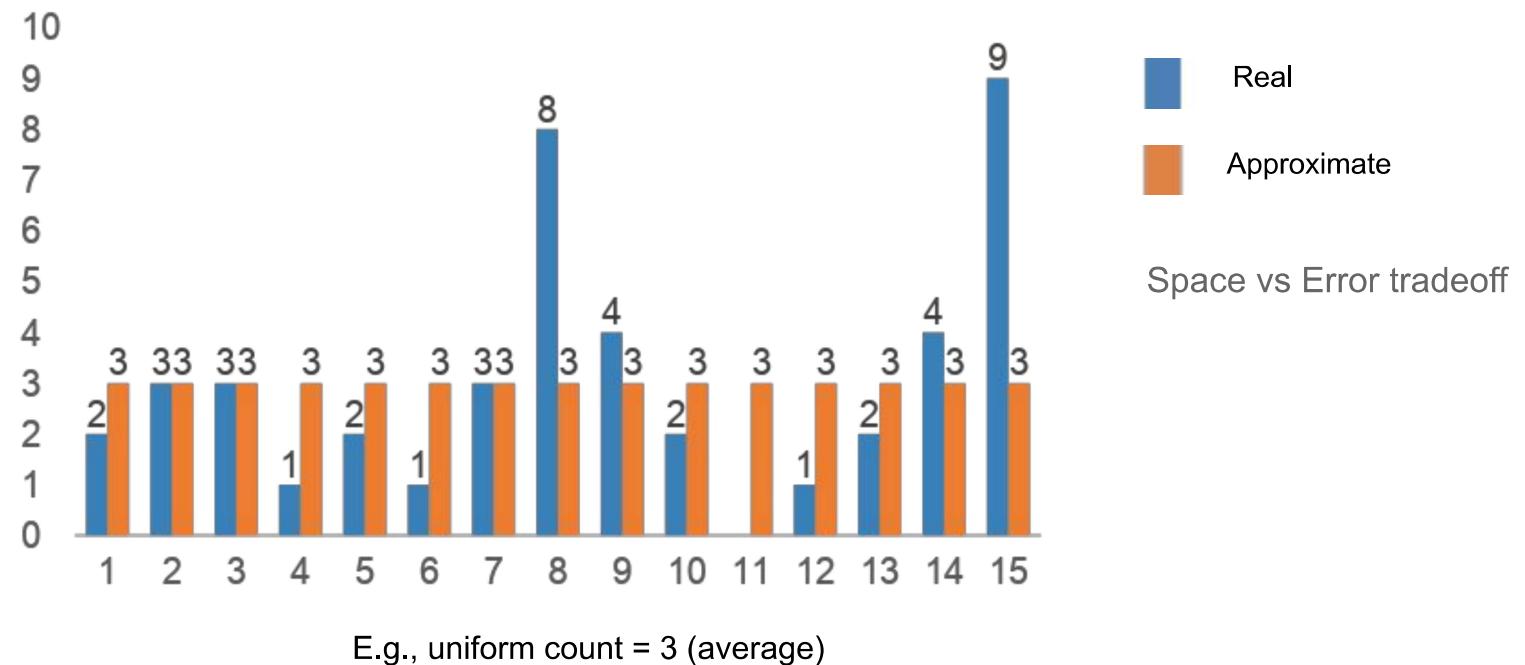
Frequency



How do we  
compute how  
many values  
between 8 and 10?  
(Yes, it's obvious)

Problem: counts take up too much space!

# What if we kept average only?

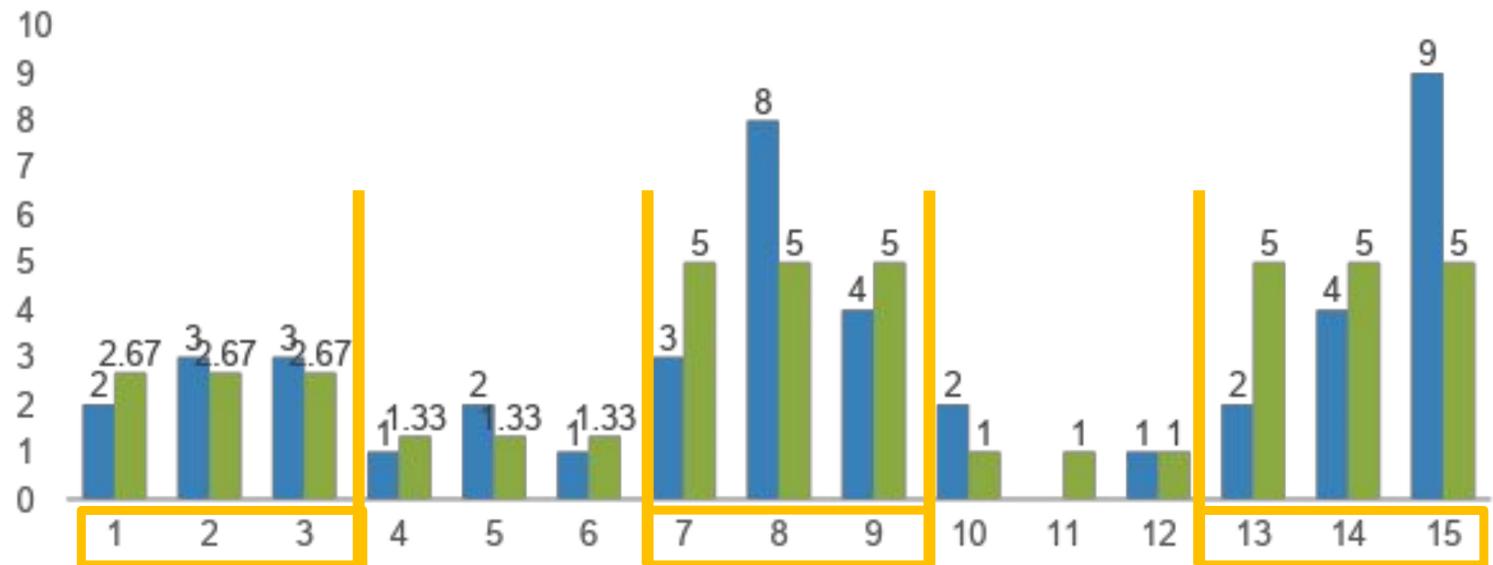


# Fundamental Tradeoffs

- Want high resolution (like the full counts)
- Want low space (like uniform)
- Histograms are a compromise!

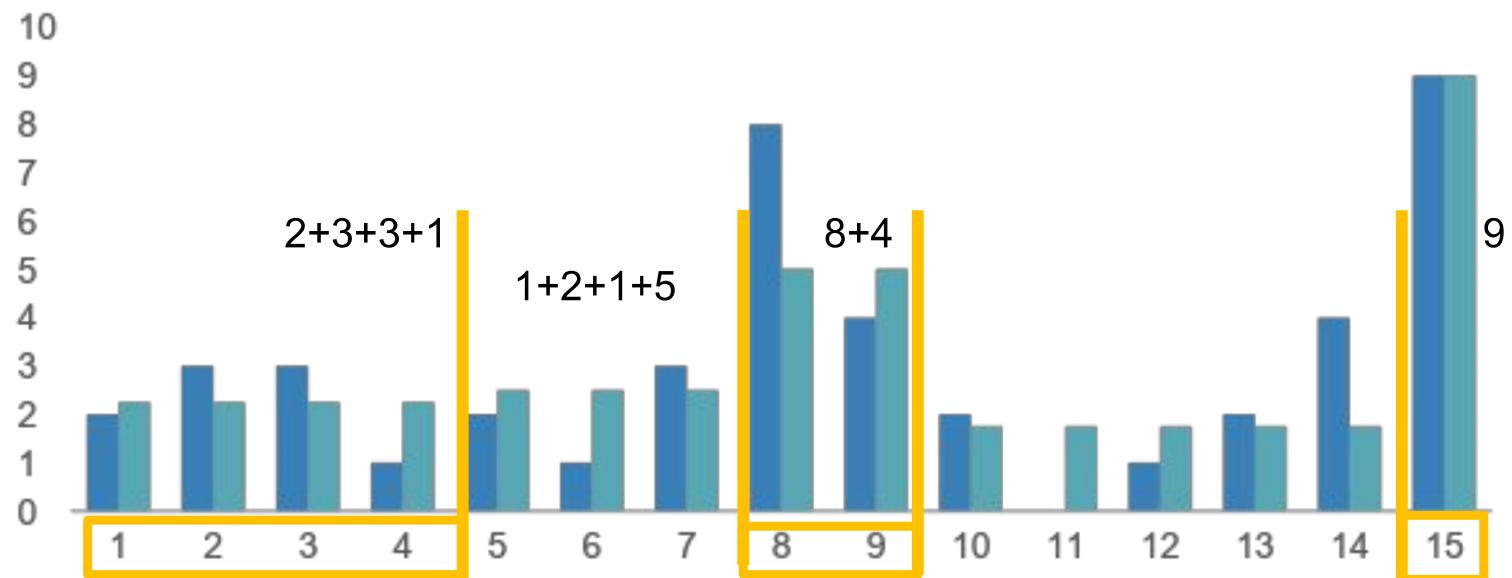
So how do we compute the “bucket” sizes?

# Equi-width



Partition buckets into roughly same width (value range)  
(E.g., 1st 3 values have average = 2.67, vs next 3 with avg = 1.33. Keep (2.67, 1.33, 5, 1, 5) as the averages for the 5 buckets)

# Equi-depth



Partition buckets for roughly same number of items (total frequency)



# Histograms

- Simple, intuitive and popular
- Parameters: # of buckets and type
- Can extend to many attributes (multidimensional)

# Maintaining Histograms

- Histograms require that we update them!
  - Typically, you must run/schedule a command to update statistics on the database
  - Out dated histograms can be bad!
- Research on self-tuning histograms and the use of query feedback

# Compressed Histograms

One popular approach

1. Store the most frequent values and their counts explicitly
2. Keep an equiwidth or equidepth one for the rest of the values

People continue to try fancy techniques here *wavelets*,  
*graphical models*, *entropy models*, ...

# Optimization

## Roadmap



Build Query Plans

1. For SFW, Joins queries
  - a. Brute-force? Sort? Hash? Count?
  - b. Pre-build an index? B+ tree, Hash?
2. What **statistics** can I keep to optimize?
  - a. E.g. Selectivity of columns, values

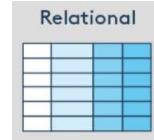


Analyze Plans

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

# Data models

**Structured**  
(e.g. ads, purchases, product tables)  
[aka relational tables]



first_name	last_name	cell	city	year_of_birth	location_x	location_y
'Mary'	'Jones'	'516-555-2048'	'Long Island'	1986	'-73.9876'	'40.7574'

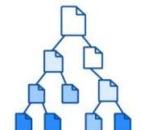
ID	user_id	profession
10	1	'Developer'
11	1	'Engineer'

ID	user_id	name	version
20	1	'MyApp'	1.0.4
21	1	'DocFinder'	2.5.7

ID	user_id	make	year
30	1	'Bentley'	1973
31	1	'Rolls Royce'	1965

**Semi-structured**  
(e.g. user profile, web site activity)  
[aka JSON documents]

Document



```
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
    type: "Point",
    coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
    { name: "MyApp",
        version: 1.0.4 },
    { name: "DocFinder",
        version: 2.5.7 }
],
cars: [
    { make: "Bentley",
        year: 1973 },
    { make: "Rolls Royce",
        year: 1965 }
]
```



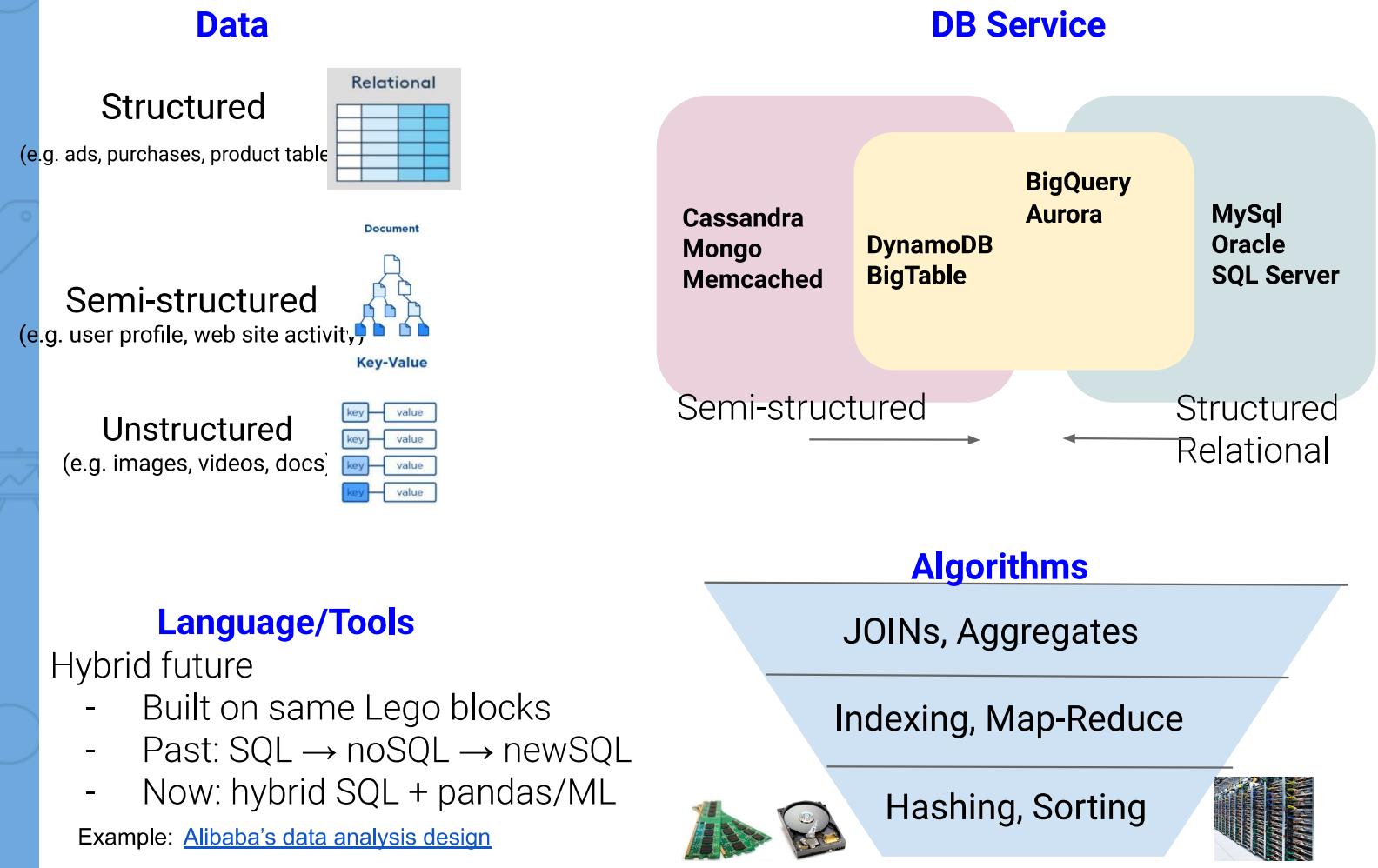
Key Starter  
Questions for:

Scale, Scale,  
Scale

1. How to scale to large data sets?
  - ▷ Is data relational, or unstructured or ...?
  - ▷ Is data in Row or Column store?
  - ▷ Is data sorted or not?
2. How do we organize search values?
  - ▷ E.g., Hash indices, B+ trees
3. How to JOIN multiple datasets?
  - ▷ E.g., SortMerge, HashJoins

# Hybrid Data Systems

In reality, a mix of systems -- e.g., Amazon/e-commerce site



# Cs 145

## Exam 1 review

## Exam Reminders

### What to expect on content?

1. Still finalizing pre-flight, but something close to this **+/-5 points** per section
2. Test understanding of principles. Not blindly applying formulae. Read questions carefully for assumptions.
3. If not obvious, ask. State any “reasonable” assumptions. (e.g., “unreasonable” = infinite speed. Reasonable = “not applying SMJ last-step optimization, because the problem didn’t state.” Or “ $B \approx B+1$  for  $B \geq 100$ ”)

This exam contains **15 pages** (including this cover page).

#	Question	Points
1	Instructions/Honor Code	0
2	Multiple Choice	15
3	Systems	26
4	SQL	20
5	Scaling/Indexing/Sorting	19
6	Feedback	2
Total		82/80

## Exam Reminders

### For Midterm (not for Example Tests and HWKs)

**Our Goal:** Test understanding of principles, and of IO cost estimates. Reduce exam time stress, and pesky off-by-one errors. Specifically,

1. **Can I use  $B \approx B+1$ ,  $f \approx f+1$  (e.g., for Systems, Indexing, Sorting questions)**
  - a.  $\Rightarrow$  YES! For the Test, we're picking  $B \geq 100$ , and large ' $f$ '.
2. **What about arithmetic computations?**
  - a.  $\Rightarrow$  We'll supply answer choices for (most) numerical options, as a quick 'self-test.' (something like below). You'll get X points for choosing the right option, and Y points for showing how you got it

The database is stored in an HDD with rows stored randomly. (Choose x if your answer lies within 5% of x)

- (a) 1 million seconds
- (b) 100k seconds
- (c) 10k seconds
- (d) 1024 seconds
- (e) 512 seconds
- (f) 256 seconds
- (g) 128 seconds
- (h) 64 seconds
- (i) < 64 seconds



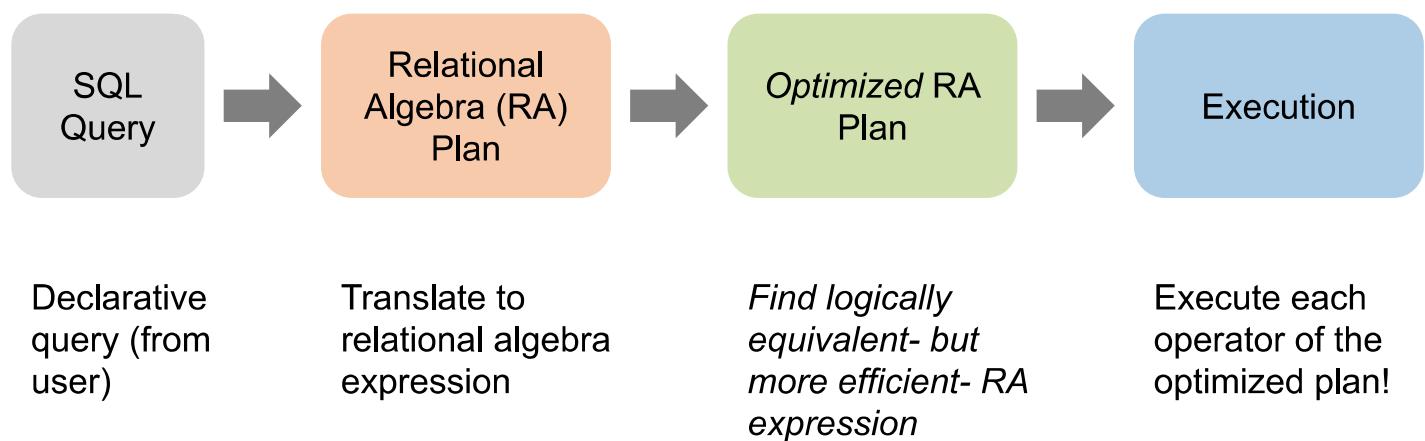
Review

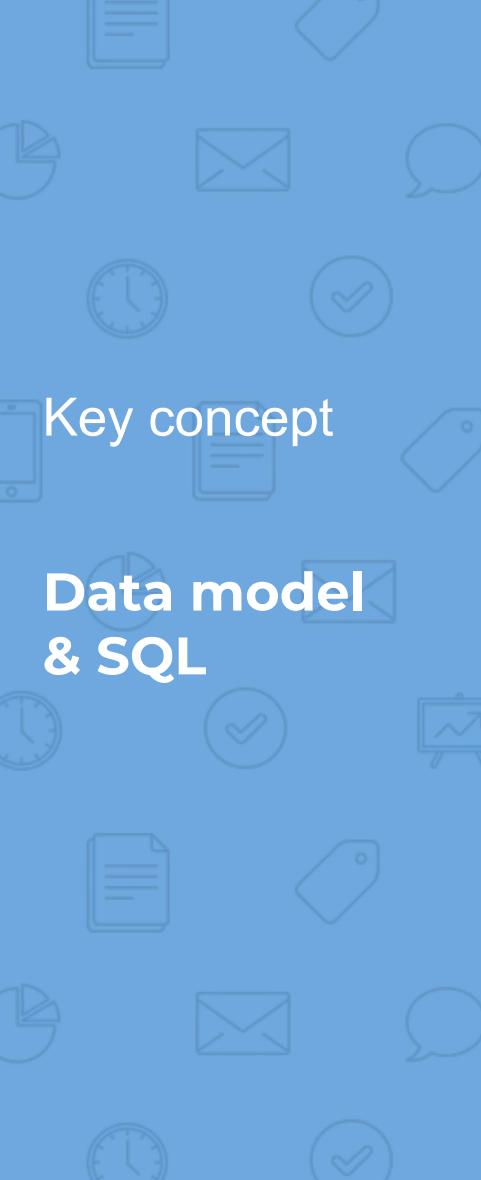
Recap of  
Systems design  
SQL  
Scale, optimization

Catch up with full lectures for details

# RDBMS Architecture

How does a DB engine work ?





## 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

A column or  
attribute

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

A table or relation is a  
multi-set of rows

The number of  
columns is the  
arity of the  
relation

The number of  
rows is the  
cardinality of the  
table

A tuple or row (or  
record)

⇒ In SQL, **tables are multisets**, meaning you can have duplicate tuples



# Examples

Students(sid: string, name: string, gpa: float)

Enrolled(student\_id: string, cid: string, grade: string)

**Students**

sid	name	gpa
102	Bob	3.9
123	Mary	3.8

**Enrolled**

student_id	cid	grade
123	145	A
123	161	A+

// Schema Declaration in SQL Example

```
CREATE TABLE Enrolled (
    student_id CHAR(20),
    cid        CHAR(20),
    grade      CHAR(10),
    PRIMARY KEY (student_id, cid),
    FOREIGN KEY (student_id) REFERENCES Students(sid)
)
```

# Preview

## SQL queries

[sqltutorial.org/sql-cheat-sheet](http://www.sqltutorial.org/sql-cheat-sheet)

## SQL CHEAT SHEET <http://www.sqltutorial.org>

### QUERYING DATA FROM A TABLE

`SELECT c1, c2 FROM t;`  
Query data in columns c1, c2 from a table

`SELECT * FROM t;`  
Query all rows and columns from a table

`SELECT c1, c2 FROM t  
WHERE condition;`  
Query data and filter rows with a condition

`SELECT DISTINCT c1 FROM t  
WHERE condition;`  
Query distinct rows from a table

`SELECT c1, c2 FROM t  
ORDER BY c1 ASC [DESC];`  
Sort the result set in ascending or descending order

`SELECT c1, c2 FROM t  
ORDER BY c1  
LIMIT n OFFSET offset;`  
Skip offset of rows and return the next n rows

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1;`  
Group rows using an aggregate function

`SELECT c1, aggregate(c2)  
FROM t  
GROUP BY c1  
HAVING condition;`  
Filter groups using HAVING clause

### QUERYING FROM MULTIPLE TABLES

`SELECT c1, c2  
FROM t1  
INNER JOIN t2 ON condition;`  
Inner join t1 and t2

`SELECT c1, c2  
FROM t1  
LEFT JOIN t2 ON condition;`  
Left join t1 and t2

`SELECT c1, c2  
FROM t1  
RIGHT JOIN t2 ON condition;`  
Right join t1 and t2

`SELECT c1, c2  
FROM t1  
FULL OUTER JOIN t2 ON condition;`  
Perform full outer join

`SELECT c1, c2  
FROM t1  
CROSS JOIN t2;`  
Produce a Cartesian product of rows in tables

`SELECT c1, c2  
FROM t1, t2;`  
Another way to perform cross join

`SELECT c1, c2  
FROM t1 A  
INNER JOIN t2 B ON condition;`  
Join t1 to itself using INNER JOIN clause

### USING SQL OPERATORS

`SELECT c1, c2 FROM t1  
UNION [ALL]`  
Combine rows from two queries

`SELECT c1, c2 FROM t1  
INTERSECT`  
`SELECT c1, c2 FROM t2;`  
Return the intersection of two queries

`SELECT c1, c2 FROM t1  
MINUS`  
`SELECT c1, c2 FROM t2;`  
Subtract a result set from another result set

`SELECT c1, c2 FROM t1  
WHERE c1 [NOT] LIKE pattern;`  
Query rows using pattern matching %, \_

`SELECT c1, c2 FROM t  
WHERE c1 [NOT] IN value_list;`  
Query rows in a list

`SELECT c1, c2 FROM t  
WHERE c1 BETWEEN low AND high;`  
Query rows between two values

`SELECT c1, c2 FROM t  
WHERE c1 IS [NOT] NULL;`  
Check if values in a table is NULL or not

# General SQL query

```
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  $R_1, \dots, R_n$
2. **GROUP BY** the attributes  $a_1, \dots, a_k$
3. **Apply HAVING condition  $C_2$  to each group (may have aggregates)**
4. Compute aggregates in **SELECT**,  $S$ , and return the result

# 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;
```

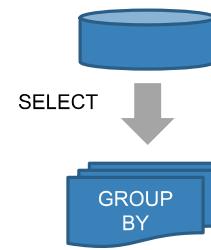
Think about **order\***!

*\*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;
```

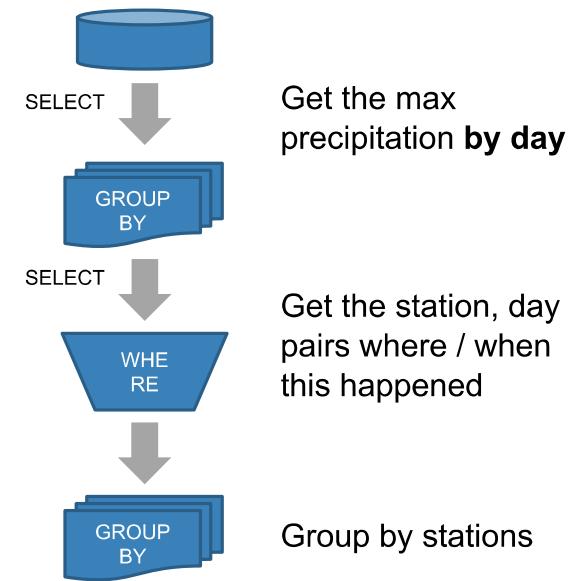


Get the max precipitation **by day**

# Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

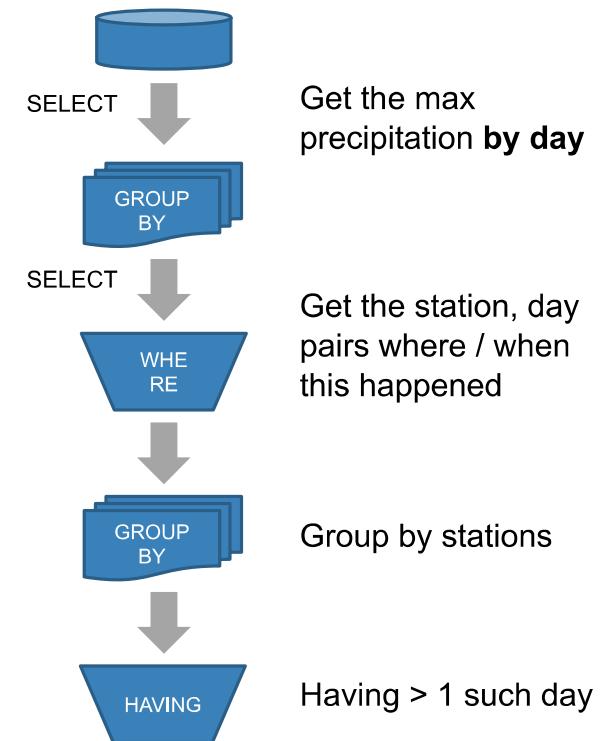
```
SELECT station_id,  
       COUNT(day) AS nbd  
  FROM precipitation,  
       (SELECT day, MAX(precip)  
        FROM precipitation  
        GROUP BY day) AS m  
 WHERE day = m.day AND precip = m.precip  
 GROUP BY station_id  
 HAVING COUNT(day) > 1  
 ORDER BY nbd DESC;
```



# Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

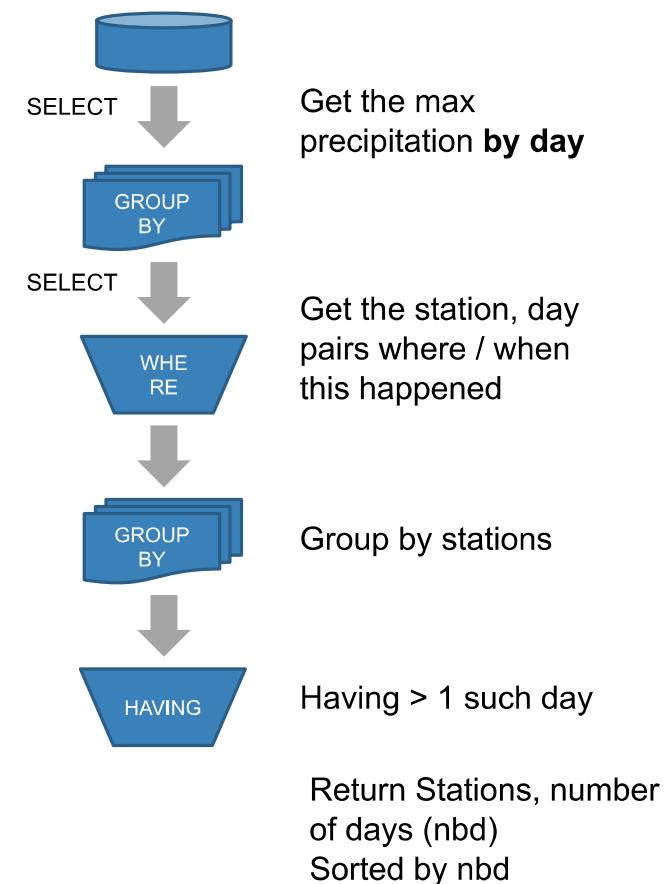
```
SELECT station_id,  
       COUNT(day) AS nbd  
  FROM precipitation,  
       (SELECT day, MAX(precip)  
        FROM precipitation  
       GROUP BY day) AS m  
 WHERE day = m.day AND precip = m.precip  
 GROUP BY station_id  
 HAVING COUNT(day) > 1  
 ORDER BY nbd DESC;
```



# 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;
```



## INTERSECT

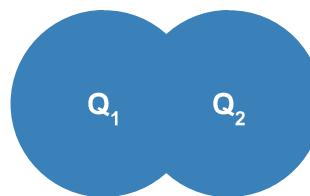
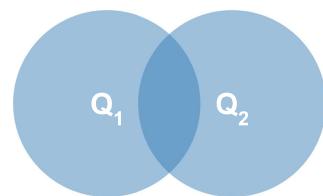
```
SELECT R.A  
FROM R, S  
WHERE R.A=S.A  
INTERSECT  
SELECT R.A  
FROM R, T  
WHERE R.A=T.A
```

## 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: SET operators operate on Sets!  
What if you want multi-sets? [Use ALL → try it out]

# 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')
```

**ANY**

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

**EXISTS**

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

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

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

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

# How SQL handles Null Values

- Numerical
  - If  $x = \text{NULL}$  then  $4*(3-x)/7$  is still  $\text{NULL}$
- Logical (FALSE = 0, TRUE = 1, NULL = 0.5)
  - $C1 \text{ AND } C2 = \min(C1, C2)$
  - $C1 \text{ OR } C2 = \max(C1, C2)$
  - $\text{NOT } C1 = 1 - C1$

Rule in SQL: include only tuples that yield TRUE (1.0)

# Null Values

Unexpected behavior:

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



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

Some Persons are not included !

Now it includes all Persons!

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

# Common Table Expressions (CTEs)

Give it a name

```
WITH ProductSales AS
  (SELECT product,
           SUM(price * quantity) AS TotalSales
    FROM Purchase
   WHERE date > '10/1/2005'
  GROUP BY product)

SELECT * FROM ProductSales
```

Useful for readability -- e.g., sub-queries, chaining queries

# Semantically: A Subset of the Cross Product

$R \bowtie S$

```
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	



Cross Product

...

Filter by conditions  
( $r.A = s.A$ )

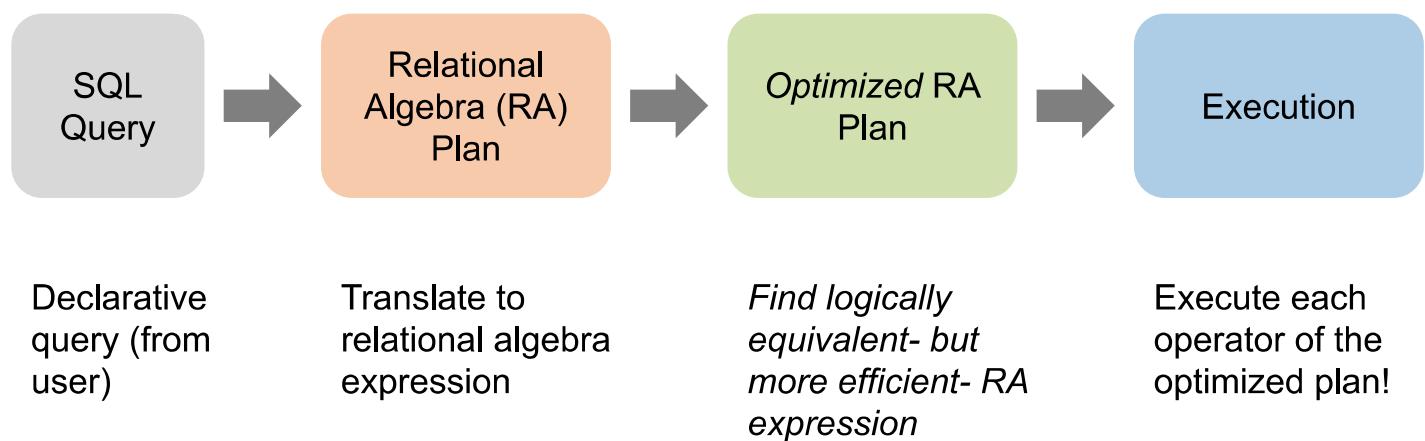
A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Can we actually implement a join in this way?

# How to execute SQL?

# RDBMS Architecture

How does a SQL engine work ?



# Optimization

## Roadmap



Build Query 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



Analyze Plans

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

# FAQ: Why do we focus on scale? How does it relate to SQL?

1. How to search Amazon's product catalog?
  - a. Consumers want answers in < 1 sec
  - b. Data layout and Indexing –
    - i. Lecture 1 vs Lecture 6 (speedup “search” queries from **hours** to < 1sec)

---

2. How to run JOINs fast?
  - a. Improved by 1000x from NLJ to BNLJ
  - b. Another 100x with SMJ/HPJ

⇒ For big data, how to scale is the primary driver

(For tiny data < 10 GBs, just use RAM)

# How to execute SQL?

⇒ Understand IO Cost models for big data

# FAQ: Why do we focus on IO cost?

From Lecture1



Srigi  
@srigi

"Latency Numbers Every Programmer Should Know"

It is hard for humans to get the picture until you translate it to "human numbers":

1 CPU cycle	1 s
Level 1 cache access	3 s
Level 2 cache access	9 s
Level 3 cache access	43 s
Main memory access	6 min
Solid-state disk I/O	2.6 days
Rotational disk I/O	1-12 months
Internet: SF to NYC	4 years
Internet: SF to UK	8 years
Internet: SF to Australia	19 years
OS virtualization reboot	423 years
SCSI command time-out	3000 years
Hardware virtualization reboot	4000 years
Physical system reboot	32 millenia

## [1] CPU Cost

(e.g., sort in RAM or check tuple equality in NLJ in RAM)

Typical algorithms in RAM  
(e.g., quicksort in  $n \log n$ )

## [2] IO Cost from HDD/SSDs

(# of Pages we read or write from HDD/SSD)

E.g., For ExternalSort

IO Cost  $\sim=$   
(sort N pages)

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

⇒ For big data, focus on **IO cost** (i.e., it's the primary factor)  
(For tiny data < 10 GBs, just use RAM)



ONE MILLION DOLLARS



ONE HUNDRED MILLION BILLION DOLLARS!

## Engineers (and interns) need sense of **cost/time tradeoffs**

In real interviews and projects,

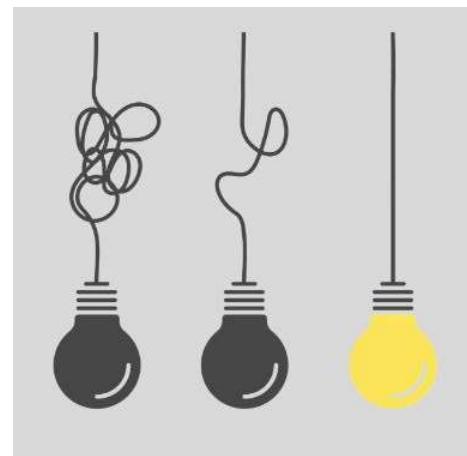
- Shockers ⇒ Proposed solutions would cost ~10s of Billion\$ or years to run
- Not after cs145 :-)
- Is something going to run in milliseconds vs seconds? Hours? Years?

(Interviewed 5000+ engineers, built teams of 1000s of engineers)

# Approximations

## Complexity

$O(n)$  vs  $O(n^2)$  – Big “O” notation for algorithms



Models of real world systems are complex. ⇒

- IO cost equations are often ‘dense.’
- We will approximate and simplify (with most of the needed accuracy), so we focus on the **core insights**.

## Engineering Approximations

Similar goal. We often approximate  $B \approx B+1$ , or  $f \approx f+1$  for simpler equations.

Is that OK?

- Yup, for “modern” ~2010+ machines
- When  $B \approx 100$ ,  $B+1/B \approx 1.01$ .

# IO Blocks For Efficiency

## Key Concepts

1. Data is stored in **Blocks** (aka “**partitions**”)
2. Sequential IO is 10x-100x+ faster than ‘many’ random IO
  - E.g., 1 MB (located sequentially) versus 1 Million bytes in random locations
3. HDDs/SSDs copy sequential ‘big’ blocks of bytes to/from RAM



1 Package

1000 Packages  
(1 Trailer)

3000 Packages  
(3 Trailers)

‘M’ Trucks

1 Byte

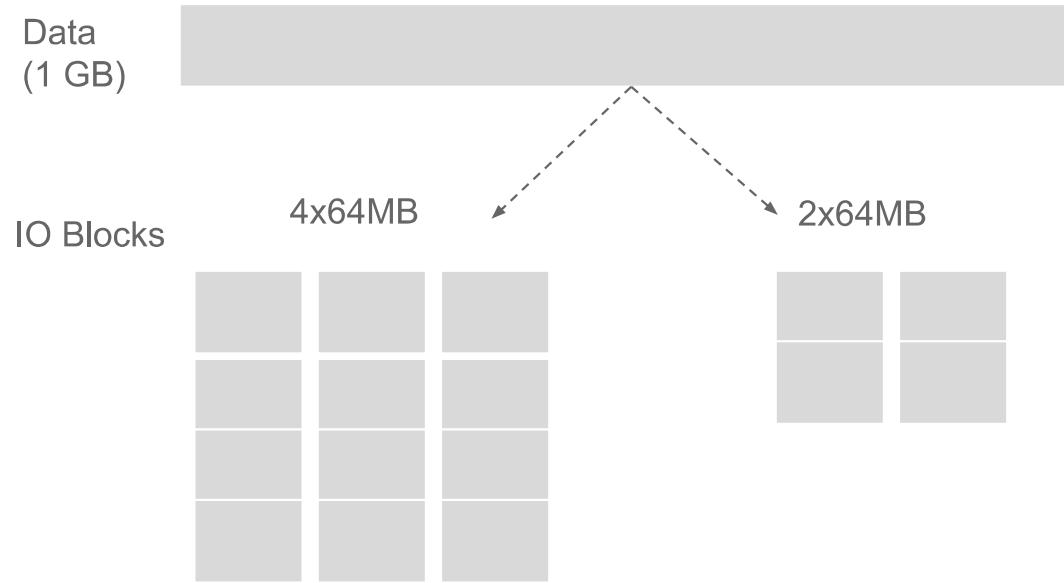
**64MB-Blocks**  
(64MBs of  
sequential, aka  
contiguous bytes)

**Nx64MB-Blocks**

(‘N’ sequential  
64MB-Blocks)

Set of ‘M’  
**Nx64MB-Blocks**

# IO Blocks For Efficiency



E.g., to store 1GB data, we need 15.625 **64MB-Blocks** ( $= 1\text{GB}/64\text{MB}$ ) in

- Sixteen 64MB-Blocks, ( $1 \times 64\text{MB} = 64\text{MB}$ ) OR
- Eight 2x64MB Blocks, (i.e.,  $M = 8, N = 2$ ), OR
- Four 4x64MB Blocks, OR Two 8x64MB Blocks, OR One 16x64B Block
- (OR some combo)

# Basic system numbers

 Srigi  
@srigi

"Latency Numbers Every Programmer Should Know"

It is hard for humans to get the picture until you translate it to "human numbers":

1 CPU cycle	1 s
Level 1 cache access	3 s
Level 2 cache access	9 s
Level 3 cache access	43 s
Main memory access	6 min
Solid-state disk I/O	2-6 days
Rotational disk I/O	1-12 months
Internet: SF to NYC	4 years
Internet: SF to UK	8 years
Internet: SF to Australia	19 years
OS virtualization reboot	423 years
SCSI command time-out	3000 years
Hardware virtualization reboot	4000 years
Physical system reboot	32 millenia



	Access Latency (secs)	Scan Throughput (GBs/sec)	What you get for ~100\$ (Jun' 22)
RAM (D-RAM)	~100 nanosec	~100 GB/sec	32 GBs
High-end SSD	~10 microsec	~5 GB/sec	640GB
HDD Seek (Hard Disk)	~10 millisec	~100 MB /sec	4 TB
Machines M1 to M2 (Network)	~ 1 microsec	~ 5 GB/sec	N/A

**Access Latency (secs) = Time to access Block's start location**

**Scan Throughput (GB/sec) = Speed to Scan + Copy data to RAM**

Note: Why are they different speeds?

- Digital (e.g. SSDs and RAM) vs Analog (e.g. [HDD seeks](#))
- Distance from CPU (e.g. RAM is on same chip as CPU vs SSD)

# IO Cost Model



	Access Latency (secs)	Scan Throughput (GBs/sec)	Total Time to read one 64MB-Block
RAM (D-RAM)	~100 nanosec	~100 GB/sec	$100 + 64\text{MB}/100\text{GB/s} = 100 + 640000 \text{nsec}$
High-end SSD	~10 microsec	~5 GB/sec	10 + 12800 microsec
HDD Seek (Hard Disk)	~10 millisec	~100 MB /sec	10 + 640 millisec
Machine 2 Machine (Network)	~ 1 microsec	~ 5 GB/sec	1 + 12800 microsec

**Total Time to ReadData =**  
**AccessLatency \* M + DataSize/ScanThroughput**

Where

- DataSize = data size (in bytes)
- M = Number of non-contiguous Blocks
- AccessLatency = Time to access Block
- ScanThroughput = Speed to copy/scan to RAM

# How to execute SQL?

- ⇒ Extend Sort and Hash for big data?
- ⇒ So we can execute JOINS, build indices

# 36

## Big Scale Lego Blocks

## Roadmap



Primary data structures/algorithms

### Hashing

HashTables  
 $(hash_i(key) \rightarrow \text{location})$

### Sorting

BucketSort, QuickSort  
MergeSort

HashPartitions(R)

HPJoins(R, S, ...)  
(aka HashPartitionJoins  
or HashJoins or HPJ)

ExternalMerge(R)  
Sort(R) (or ExternalMergeSort or ExternalSort)

SortMergeJoins (R, S, ...)  
(aka SMJs or ExternalSortMergeJoin)

The big idea – with ~5 algorithms, we can efficiently solve most big data problems that don't fit in RAM.



# Notation for IO cost estimates

We consider “IO aware” algorithms: *care about IO*

Given a relation R, let:

- $T(R)$  = # of tuples in R
- $P(R)$  = # of pages in R

Recall that we read / write entire pages (blocks)

We'll assume **B** buffer for input ( $B$  usually  $\ll P(R), P(S)$ )

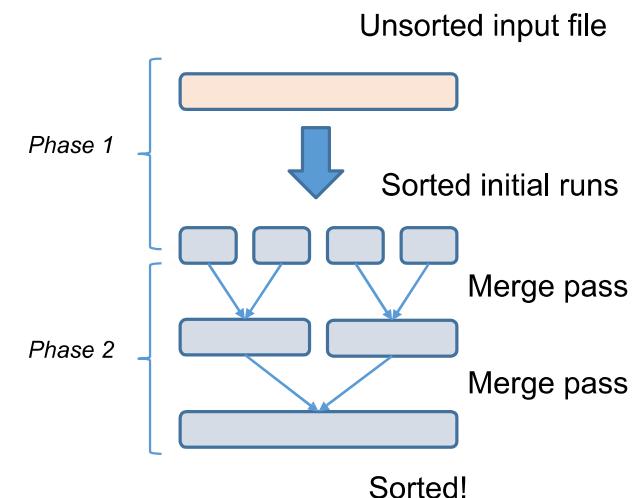
+ **1** for output ← Imagine there's always 1 extra page

# ExternalSort 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!



# 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

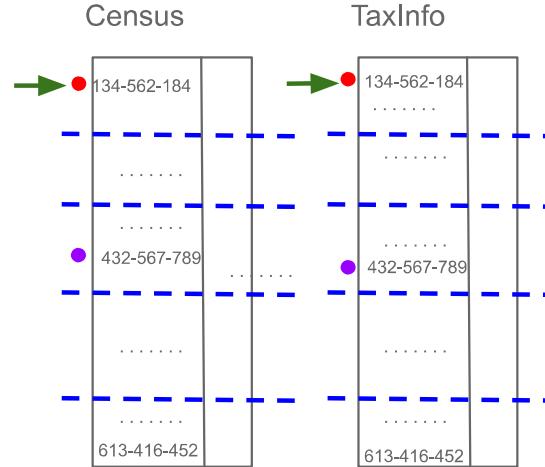
Given sufficient buffer space, linear in  $P(R)$ ,  $P(S)$   
I.e.  $\sim O(P(R)+P(S))$

By only supporting equijoins & taking advantage of this structure!

## Preview of smarter joins

# Pre-process data before JOINing

SortMergeJoin

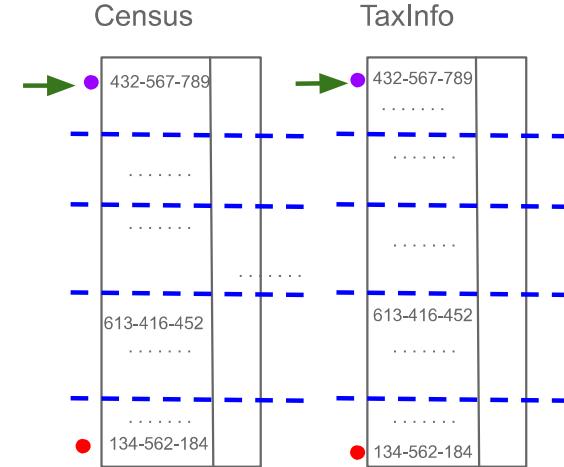


Census table  
(row store)

TaxInfo table  
(row store)

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

HashPartitionJoin



Census table  
(row store)

TaxInfo table  
(row store)

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

# Sort Merge Join (SMJ)

**Goal:** Execute  $R \bowtie S$  on A

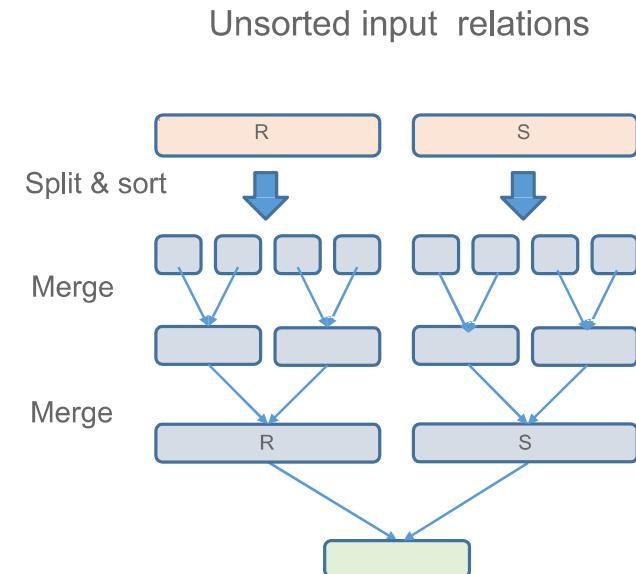
**Key Idea:** We can sort R and S, then just scan over them!

**IO Cost:**

- Sort phase:  $\text{Sort}(R) + \text{Sort}(S)$
- Merge / join phase:  $\sim P(R) + P(S) + \text{OUT}$

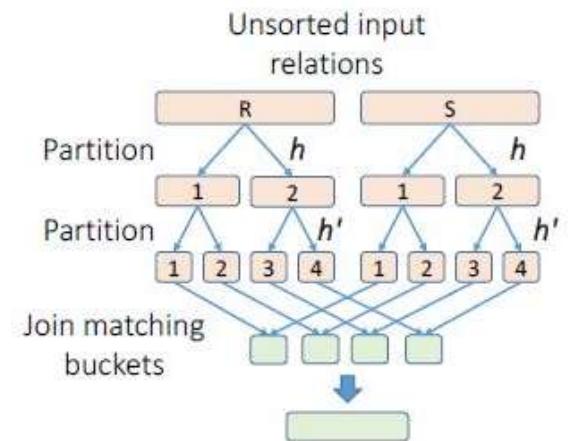
[Reminder: Above assume 1 R = 1 W cost.

IO system could have different Read Cost vs Write Cost] (HW2)



# Hash Join

- **Goal:** Execute  $R \bowtie S$  on A
- **Key Idea:** We 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  $\sim P(R) + P(S) + OUT$  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 Costs

### Sort(R) with N pages

IO Cost ~=

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

$\sim 2N$

$\sim 4N$

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?

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)

when  $N \approx B$

(because  $(\log_B 0.5) < 0$ )

when  $N \approx 2*B^2$

(because  $(\log_B B) = 1$ )

**SMJ(R, S)**

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

Case 1. For enuf B ( $N \leq \sim 2B$ ) and no backup

$\sim 3 * (P(R) + P(S)) + \text{OUT}$

Case 2. If R and S pre-sorted and no backup

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

**HPJ(R, S)**

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

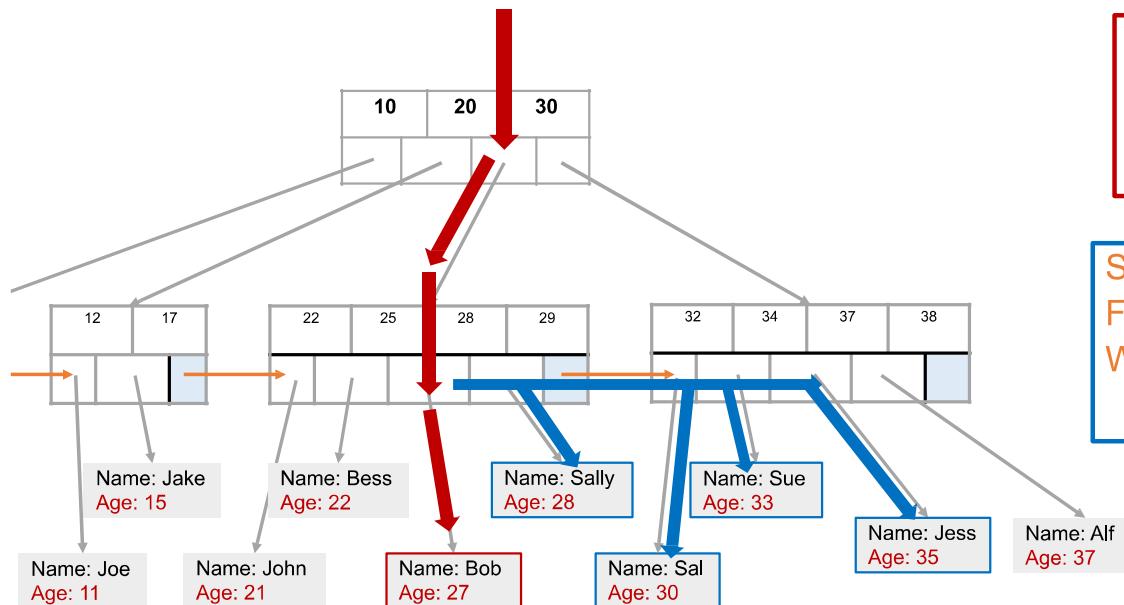
Case 1. For not much data skew and enuf B

$\sim 3 * (P(R) + P(S)) + \text{OUT}$

Case 2. If R and S pre-HPed

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

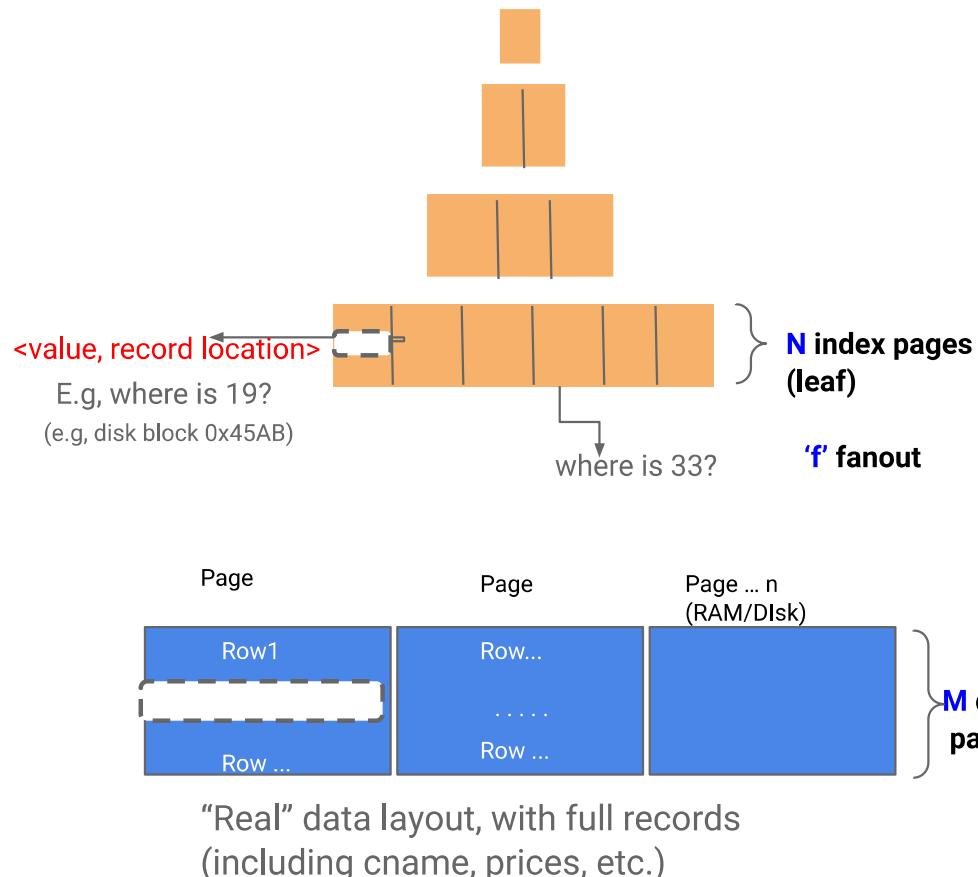
# Searching a B+ Tree



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

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

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



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

Let us build an **index for an SKey** (e.g., CName) in data

- **NumSKeys** = number of SValues for that SKey (e.g., 5000 cnames)
- **SKeySize** = size of SKey (e.g. 4 bytes)
- **PointerSize** = size of pointer (e.g. 8 bytes)

For B+ tree

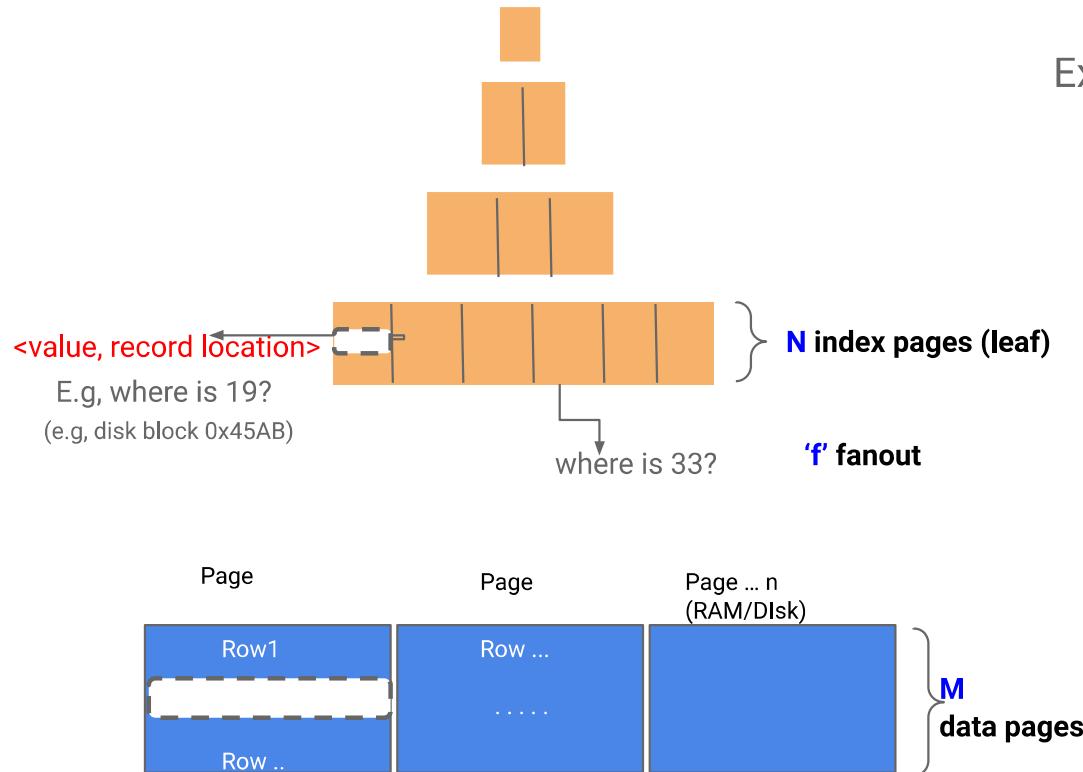
- $f$  = fanout (*we'll assume it is constant for simplicity...*)
- $h$  = height of tree (e.g., 1, 2, ...)
- $N$  = number of index pages

Simplified cost model

$$f \leq \frac{\text{PageSize}}{\text{SKeySize} + \text{PointerSize}} // \text{ Fit upto } f \text{ (SKey, Pointer)}$$
$$f^h \geq \text{NumSKeys} // \text{ Leaf nodes should point to all SKeys}$$

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

$f \leq \frac{\text{PageSize}}{\text{SKeySize} + \text{PointerSize}}$  // Fit upto 'f' (SKey, Pointer)  
 $f^h \geq \text{NumSKeys}$  // Leaf nodes should point to all SKeys  
 $h \geq \log_f \text{NumSKeys}$  // From previous equation



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

Example 1: Search Amazon's 1 trillion Products

- SKeySize = 8 bytes,
- PointerSize = 8 bytes
- NumSKeys = 1 Trillion

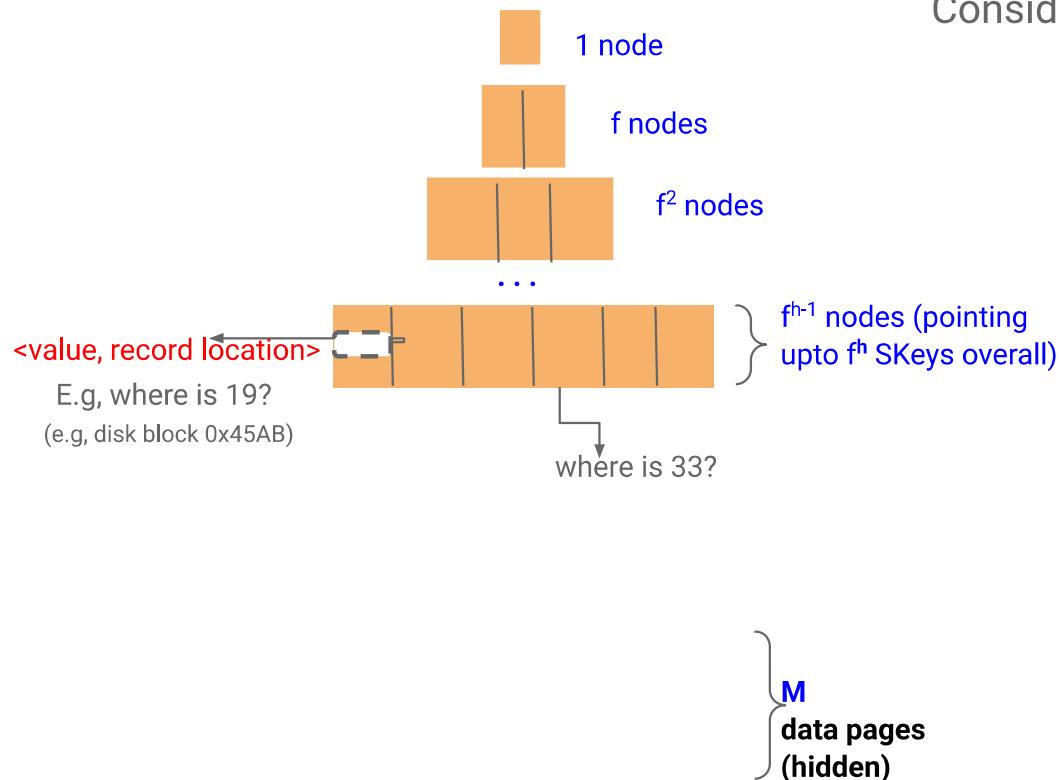
PageSize	64KB	64MB
f	~4000	~4 million
h	~4 $\text{ceil}(\log 1\text{Trillion})$ = $\text{ceil}(3.33)$	~2 $\text{ceil}(1.87)$

**AMAZING:** Worst-case for 1 Trillion SKeys

- 2 IOs (for 64MB) for index
- 1 IO for data page

# B+ tree Levels and node sizes

$f \leq \frac{\text{PageSize}}{\text{SKeySize} + \text{PointerSize}}$  // Fit upto 'f' (SKey, Pointer)  
 $f^h \geq \text{NumSKeys}$  // Leaf nodes should point to all SKeys  
 $h \geq \log_f \text{NumSKeys}$  // From previous equation



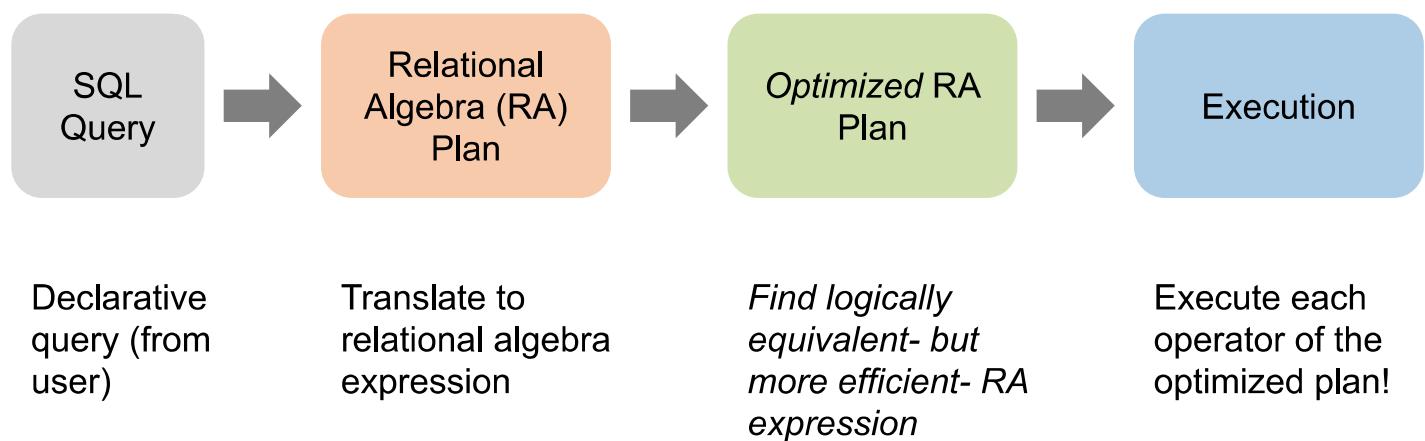
Consider Example with  $f \approx 4000$  and  $\text{PageSize} = 64\text{KB}$

Level	Num Nodes (size)	Size
0 (root)	1	64 KB
1	$\sim 4000$ ( $f$ )	$4000^*$ $64\text{KB}$
2	$\sim 4000^2$ ( $f^2$ )	$\sim 4000^2*$ $64\text{KB}$
3	$\sim 4000^3$ ( $f^3$ )	$\sim 4000^3*$ $64\text{KB}$
$h-1$	$4000^{h-1}$ ( $f^{h-1}$ nodes) [Collectively pointing at upto $f^h$ SKeys]	$4000^{h-1} * 64\text{KB}$

Recall We use fanout ' $f$ ' as a constant, for simplicity. The algorithm uses ' $f$ ' for keys, and ' $f+1$ ' for pointers. Our engineering approximation uses ' $f$ ' for both. (i.e.,  $f \approx f + 1$ )

# DBMS Architecture

How does a DB engine work ?



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.)
3. After midterm, ...scaling WRITES and OLTP (aka Transactions)



# Transactions



# SQL Writes

## SQL CHEAT SHEET <http://www.sqltutorial.org>



### MANAGING TABLES

```
CREATE TABLE t (
    id INT PRIMARY KEY,
    name VARCHAR NOT NULL,
    price INT DEFAULT 0
);
```

Create a new table with three columns

```
DROP TABLE t;
```

Delete the table from the database

```
ALTER TABLE t ADD column;
```

Add a new column to the table

```
ALTER TABLE t DROP COLUMN c;
```

Drop column c from the table

```
ALTER TABLE t ADD constraint;
```

Add a constraint

```
ALTER TABLE t DROP constraint;
```

Drop a constraint

```
ALTER TABLE t1 RENAME TO t2;
```

Rename a table from t1 to t2

```
ALTER TABLE t1 RENAME c1 TO c2;
```

Rename column c1 to c2

```
TRUNCATE TABLE t;
```

Remove all data in a table

### USING SQL CONSTRAINTS

```
CREATE TABLE t(
    c1 INT, c2 INT, c3 VARCHAR,
    PRIMARY KEY (c1,c2)
);
```

Set c1 and c2 as a primary key

```
CREATE TABLE t1(
    c1 INT PRIMARY KEY,
    c2 INT,
    FOREIGN KEY (c2) REFERENCES t2(c2)
);
```

Set c2 column as a foreign key

```
CREATE TABLE t(
    c1 INT, c2 INT,
    UNIQUE(c2,c3)
);
```

Make the values in c1 and c2 unique

```
CREATE TABLE t(
    c1 INT, c2 INT,
    CHECK(c1 > 0 AND c1 >= c2)
);
```

Ensure c1 > 0 and values in c1 >= c2

```
CREATE TABLE t(
    c1 INT PRIMARY KEY,
    c2 VARCHAR NOT NULL
);
```

Set values in c2 column not NULL

### MODIFYING DATA

```
INSERT INTO t(column_list)
VALUES(value_list);
```

Insert one row into a table

```
INSERT INTO t(column_list)
VALUES (value_list),
       (value_list), ...;
```

Insert multiple rows into a table

```
INSERT INTO t1(column_list)
SELECT column_list
FROM t2;
```

Insert rows from t2 into t1

```
UPDATE t
SET c1 = new_value;
```

Update new value in the column c1 for all rows

```
UPDATE t
SET c1 = new_value,
    c2 = new_value
WHERE condition;
```

Update values in the column c1, c2 that match the condition

```
DELETE FROM t;
```

Delete all data in a table

```
DELETE FROM t
WHERE condition;
```

Delete subset of rows in a table

# SQL Writes

```
UPDATE Product  
SET Price = Price - 1.99  
WHERE pname = 'Gizmo'
```

```
INSERT INTO SmallProduct(name, price)  
SELECT pname, price  
FROM Product  
WHERE price <= 0.99
```

```
DELETE Product  
WHERE price <= 0.99
```

How?

## Example Game App

DB v0

(Recap lectures)

App designer

- Q1: 1000 users/sec?
- Q2: Offline?
- Q3: Support v1, v1' versions?

Mobile Game



Real-Time User Events



Systems designer

- Q7: How to model/evolve game data?
- Q8: How to scale to millions of users?
- Q9: When machines die, restore game state gracefully?

Product/Biz designer

Report & Share  
Business/Product Analysis



- Q4: Which user cohorts?
- Q5: Next features to build?
- Experiments to run?
- Q6: Predict ads demand?



How?

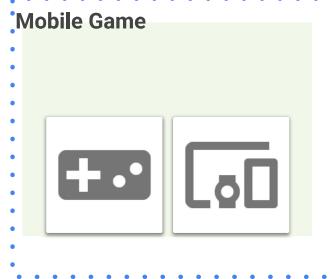
## Example Game App

DB v0

(Recap lectures)

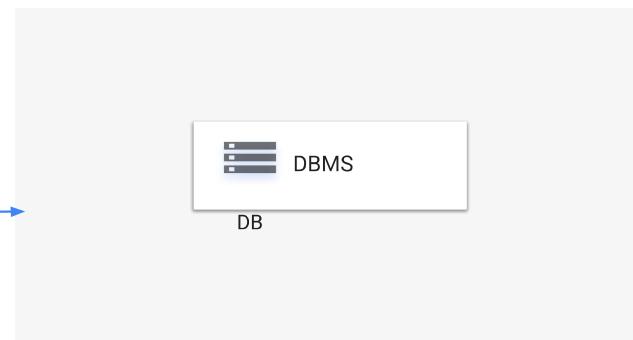
App designer

- Q1: 1000 users/sec?
- Q2: Offline?
- Q3: Support v1, v1' versions?



Real-Time User Events

Systems designer



- Q7: How to model/evolve game data?
- Q8: How to scale to millions of users?
- Q9: When machines crash, restore game state gracefully?

Product/Biz designer

Report & Share  
Business/Product Analysis



- Q4: Which user cohorts?
- Q5: Next features to build?
- Experiments to run?
- Q6: Predict ads demand?

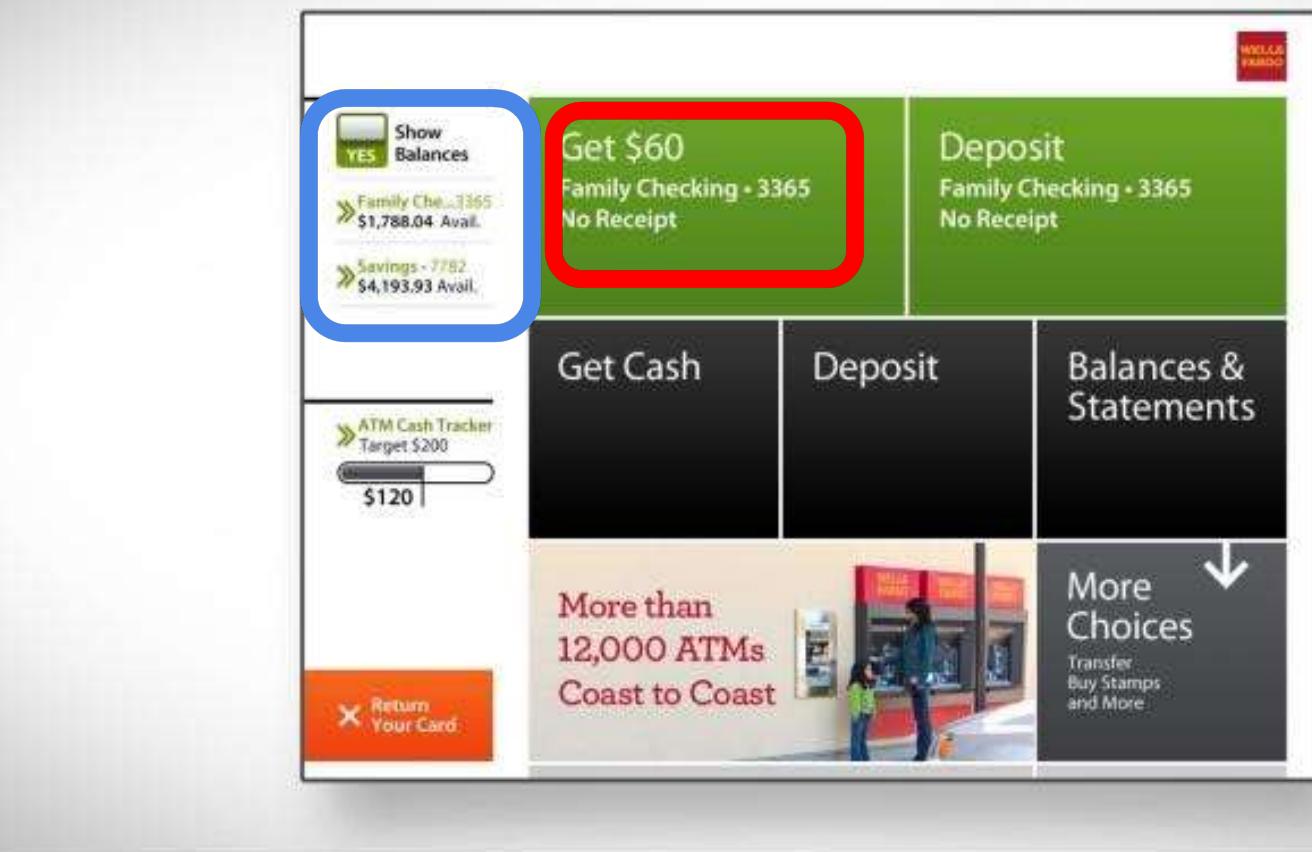




# Today's Lecture

1. Why Transactions?
2. Transactions
3. Properties of Transactions: ACID
4. Logging

Example  
Unpack  
ATM DB:  
Transaction



Read Balance  
Give money  
Update Balance

vs

Read Balance  
Update Balance  
Give money

WELLS  
FARGO



Visa does > 60,000 TXNs/sec with users & merchants

Want your 4\$ Starbucks transaction to wait for a stranger's 10k\$ bet in Las Vegas ?  
⇒ Transactions can (1) be quick or take a long time, (2) unrelated to you



Transactions are at the core of

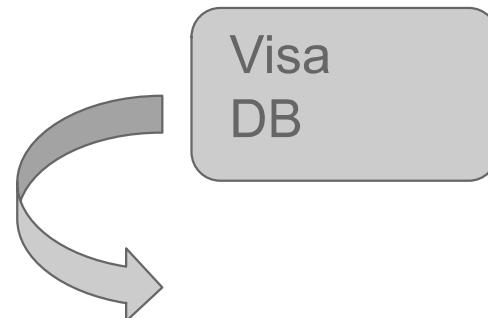
- payment, stock market, banks, ticketing
- Gmail, Google Docs (e.g., multiple people editing)

## Example Visa DB



### Transaction Queue

- 60000 user TXNs/sec
- Monthly 10% Interest TXN



Account	...	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20



### Design#1 [RAM only]

For each **Transaction** in Queue

- For relevant records
  - Read, Update records in RAM

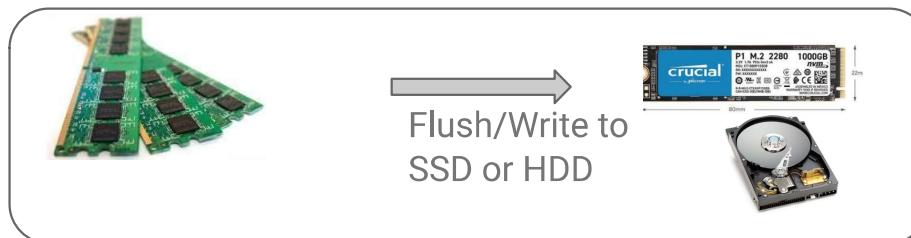
[Obviously bad → e.g., turn off power, lose \$\$s]

### Design#2 [RAM + Write/Flush on HDD/SSD]

For each **Transaction** in Queue

- For relevant records
  - Read, Update records in RAM
  - Flush/Write updates on SSD/HDD

[Let's study tradeoffs in next few slides]



# Example

Monthly  
Visa bank  
interest transaction

Money

Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...	...	
30108		-110
40008		110
50002		22

'T-Monthly-423'

Monthly Interest 10%

4:28 am Starts run on 100M bank accounts

Takes 24 hours to run

```
UPDATE Money  
SET Balance = Balance * 1.1
```

# Example

Monthly  
bank  
interest  
transaction  
With crash

Money		
Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@10:45 am)		
Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-200
5002		320
...	...	
30108		-110
40008		110
50002		22

'T-Monthly-423'

Monthly Interest 10%

4:28 am Starts run on 100M bank accounts

Takes 24 hours to run

Network outage at 10:29 am,  
System access at 10:45 am

??

??

??

Did T-Monthly-423 complete?  
Which tuples are bad?

Case1: T-Monthly-423 crashed  
Case2: T-Monthly-423 completed  
4002 deposited 20\$ at 10:45 am

Problem 1: Wrong :(

# Example

Monthly  
Visa bank  
interest  
transaction

## Performance

Money

Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...	...	
30108		-110
40008		110
50002		22

Cost to update all data

100M bank accounts → 100M updates?

Worst case

(@10 msec/HDD seek, that's 1 million secs)

(@10 musec/SSD access, ~1000 secs)



Problem2: SLOW to run :(

Problem3: How does user access money in this time?

# Problems

Problem1: How can a {Bank, Visa, Fintech, NASDAQ} update 100 million records correctly?

Problem2: How to update 100 million records in **Seconds** (not 1000s to 1 million secs)?

Problem3: How to let users have access their accounts?

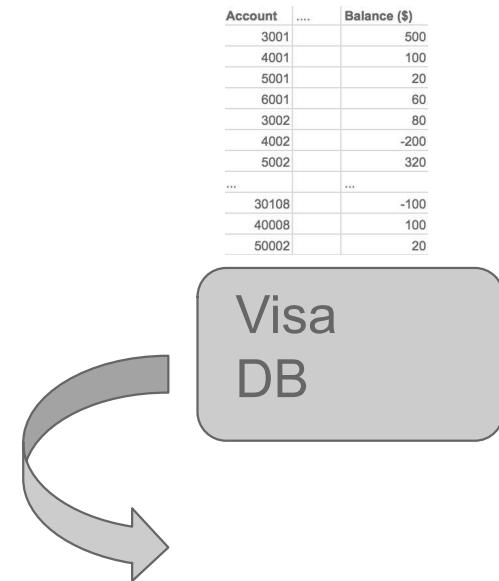
Problem4: Make it easy for the application engineer!!!

## Preview – Our eventual solution (after 2 weeks)



### Transaction Queue

- 60000 user TXNs/sec
- Monthly 10% Interest TXN



Account	...	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

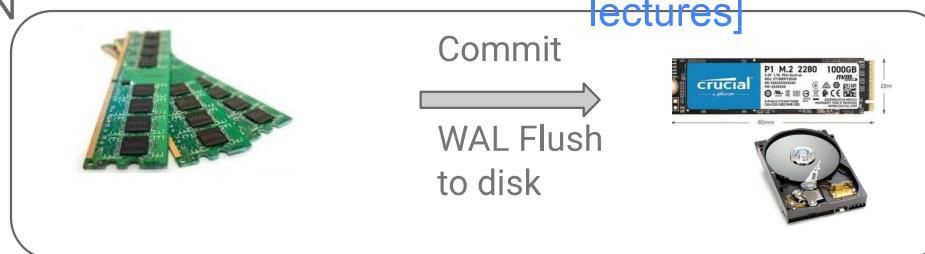


### Design#3 VisaDB

For each **Transaction** in Queue

- For relevant records
  - Use **2PL** to acquire/release locks
  - Read, Process, Write records
  - **WAL** Logs for updates
- Commit or Abort

[Note: We'll focus on the BLUE parts in next lectures]



# Motivation for Transactions

Group user actions (reads & writes) into *Transactions* (**TXNs**) helps with two goals:

1. Recovery & Durability: Keep the data consistent and durable.  
*Despite system crashes, user canceling TXN part way, etc.*

This lecture!

**Idea:** Use **LOGS**. Support to “commit” or “rollback” TXNs

2. Concurrency: Get better performance by parallelizing TXNs  
*without creating ‘bad data.’ Despite slow disk writes and reads.*

Next lecture

**Idea:** Use **LOCKS**. Run several user TXNs concurrently.



# Today's Lecture

1. Why Transactions?
2. Properties of Transactions: ACID
3. Logging

# Transactions: Basic Definition

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

In the real world, a TXN either happened completely or not at all (e.g., you withdrew 100\$ from bank. Or not.)

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

# Transactions in SQL

- In “ad-hoc” SQL, each statement = one transaction
- In a program, multiple statements can be grouped together as a transaction

```
START TRANSACTION
    UPDATE Bank SET amount = amount - 100
    WHERE name = 'Bob'
    UPDATE Bank SET amount = amount + 100
    WHERE name = 'Joe'
    COMMIT
```

# Example 1: Protection against crashes / aborts

Scenario: Make a CheapProducts table, from a Products table

Client 1:

```
INSERT INTO CheapProduct(name, price)  
SELECT pname, price  
FROM Product  
WHERE price <= 0.99
```

Crash / abort!

```
DELETE Product  
WHERE price <=0.99
```

What goes wrong?

Client 1:

START TRANSACTION

INSERT INTO CheapProduct(name, price)

SELECT pname, price

FROM Product

WHERE price <= 0.99

DELETE Product

WHERE price <=0.99

COMMIT

Now we'd be fine! We'll see how / why this lecture

## Example 2: Multiple users: single statements

Client 1: [at 10:01 am]

```
UPDATE Product  
SET Price = Price – 1.99  
WHERE pname = ‘Gizmo’
```

Client 2: [at 10:01 am]

```
UPDATE Product  
SET Price = Price*0.5  
WHERE pname=‘Gizmo’
```

Two managers attempt to change prices ***at same time*** -

What could go wrong?

Client 1: START TRANSACTION

    UPDATE Product

    SET Price = Price – 1.99

    WHERE pname = ‘Gizmo’

    COMMIT

Client 2: START TRANSACTION

    UPDATE Product

    SET Price = Price\*0.5

    WHERE pname=‘Gizmo’

    COMMIT

Now works like a charm- we'll see how / why next lecture...



### 3. Properties of Transactions



What you will  
learn about in  
this section

1. Atomicity
2. Consistency
3. Isolation
4. Durability



# ACID: Atomicity

- TXN is all or nothing
  - *Commits*: all the changes are made
  - *Aborts*: no changes are made



## **ACID: Consistency**

- The tables must always satisfy user-specified *integrity constraints*  
(E.g., Account number is unique, Sum of *debits* and of *credits* is 0)
- How DBs support consistency? Split responsibility
  - i. Programmer knows needed logic. Will assert a TXN will go from one consistent state to a consistent state
  - ii. System asserts the TXN is atomic (e.g., if EXCEPTION, rolls back)



# ACID: Isolation

- A TXN executes **concurrently** with other TXNs
- Effect of TXNs is the same as TXNs running one after another

Conceptually,

- similar to OS “sandboxes”
- E.g. TXNs can’t observe each other’s “partial updates”



## ACID: Durability

- The effect of a TXN must **persist** after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
- ⇒ Write data to durable IO (e.g., disk)



# ACID Summary

- **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
- **Durable**
  - Once a TXN has committed, its effects remain in the database

# A Note: ACID is one popular option!

- Many debates over ACID, both **historically** and **currently**
- Some “NoSQL” DBMSs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs...

⇒ Usually, depends on what consistency and performance your application needs



ACID is an extremely important & successful paradigm,  
but still debated!



## 4. Atomicity & Durability via Logging

Conceptual Idea:  
Plan a trip with friends.

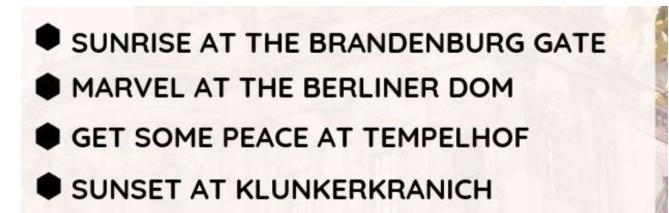
⇒ TODOs in NY, LA, SF, and Berlin

Execute your TODO list in random sequence, as soon as the ideas flow in from friends?

OR

Plan your TODOs in NY, in LA, in Berlin?  
And commit by buying Tickets?

Done	Name	Category
<input type="checkbox"/>	Download Travel Checklist	Preparation
<input type="checkbox"/>	Take an umbrella for traveling to	Packing
<input type="checkbox"/>	Take a French dictionary for traveling	Packing
<input type="checkbox"/>	Take sunglasses for traveling to	Packing
<input type="checkbox"/>	Walk up the Eiffel Tower	Paris
<input type="checkbox"/>	Have a coffee at a cafe	Paris
<input type="checkbox"/>	Go and see the Queen	London
<input type="checkbox"/>	Listen to Big Ben rings	London
<input type="checkbox"/>	Drink tea at 5 pm	London
<input type="checkbox"/>	Visit Opera House	Odessa
<input type="checkbox"/>	Go down the Patyomlin Stairs	Odessa



## Big Idea

LOGS!  
(aka TODO/  
ledger)

Recall (on disks)

- ▷ For SMJ/Indexing, we exploited – **Block reads FASTER** than random reads
- ▷ Now – **Appends are FASTER** than random writes
- ▷ [ See [Notebook](#) – ]

## Big Idea: LOGs (or log files or ledger)

- ▷ Any value that changes? Append to LOG!
  - LOG is a compact “todo” list of data updates
- ▷ Intuition:
  - Data pages: (a) Update in RAM (fast) (b) Update on disk later (slow)
  - LOGs: ( c) Append “todo” in LOGs and (d) control when you Flush LOGs to disk

Many kinds of LOGs. We'll study a few key ones!



# Basic Idea: (Physical) Logging

Idea:

- Log consists of an ordered list of Update Records
- Log record contains UNDO information for every update!  
 $\langle \text{TransactionID}, \&\text{reference}, \text{old value}, \text{new value} \rangle$   
(e.g., key)

What DB does?

- Owns the log “service” for all applications/transactions.
- Appends to log. **Flush** when necessary – force writes to disk

This is sufficient to UNDO any transaction!

Couple of changes,

Planning ahead for next 4 weeks

## Bigger CS class

- More diverse in backgrounds
- Ed posts are growing faster than we can search, or respond to
  - On k topics, we now have  $\sim 2^k$  threads (various combos)
- Recalibrating before Transactions

## Transactions are important, and tricky

1. Released short videos on focused topics [two on Logging, ~10 mins each]
2. Goal:
  - a. Set up as videos you can view separately on focused sub-topics
  - b. We'll use these examples in class to spark questions
  - c. Set up focused Ed posts – one on LOGGING, one on LOCKING, etc.  
Easier for students to find, navigate, connect similar questions others have or are exploring, and help peers on this tricky topic. (And for us to answer connected questions)

## Why are Transactions tricky, and important?

1. **Old, New, and Evolving**
  - a. Old: Classic Jim Gray text book on Transaction Processing on HDDs [e.g., cs 345]
  - b. Evolving: Machine architectures, HDDs vs SSDs performance
  - c. Industry + Research – new problems in OLAP → OLTP
    - i. Past 10 yrs lots on OLAP - e.g., BQ/RedShift, etc.
    - ii. Next 10 yrs – Big investments are happening in OLTP now

# Problems

Problem1: How can a {Bank, Visa, Fintech, NASDAQ} update 100 million records correctly?

Problem2: How to update 100 million records in **Seconds** (not 1000s to 1 million secs)?

Problem3: How to let users have access their accounts?

Problem4: Make it easy for the application engineer!!!

# Example1: “TXN-INC5”

1 TXN, 1 value

numLikes = 30

// Stored in an IO block in DB

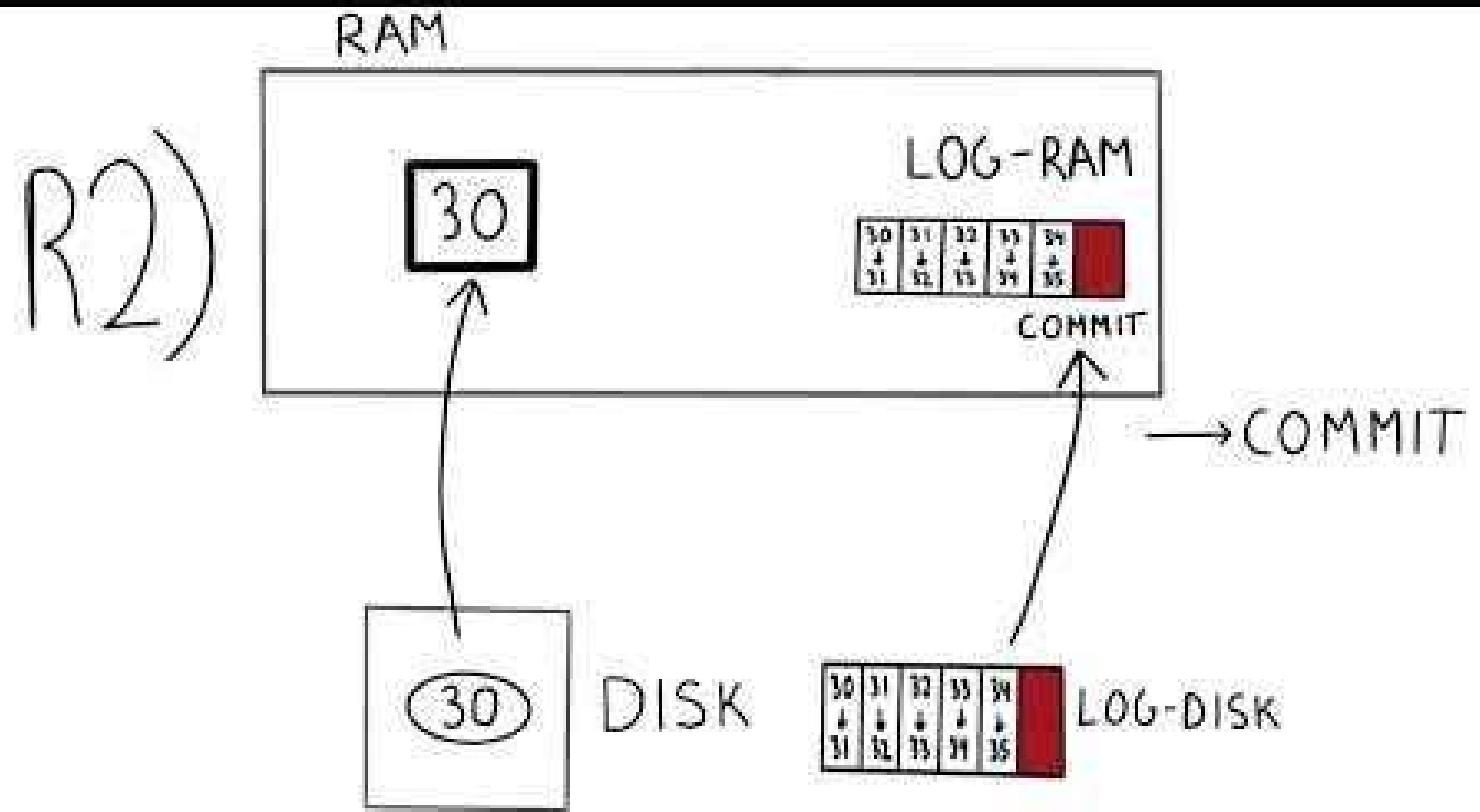
numLikes++ 5 times

Possible values for numLikes: // Updated in IO block in DB

TXN Aborts (fails/crashes)  $\Rightarrow$  30

TXN Commit (aka succeeds)  $\Rightarrow$  35

## End to end Example



## Key questions

For 1 TXN, 1 value, we saw a full run. But . . .

1. Is it **correct**? What happens if there's a crash?
2. Why is there a speedup?
3. How to extend?

⇒ Let's focus on A and D in ACID.

**Recall:** A TXN can either COMMIT or ABORT. DB guarantees below, whether the TXN succeeds or fails.

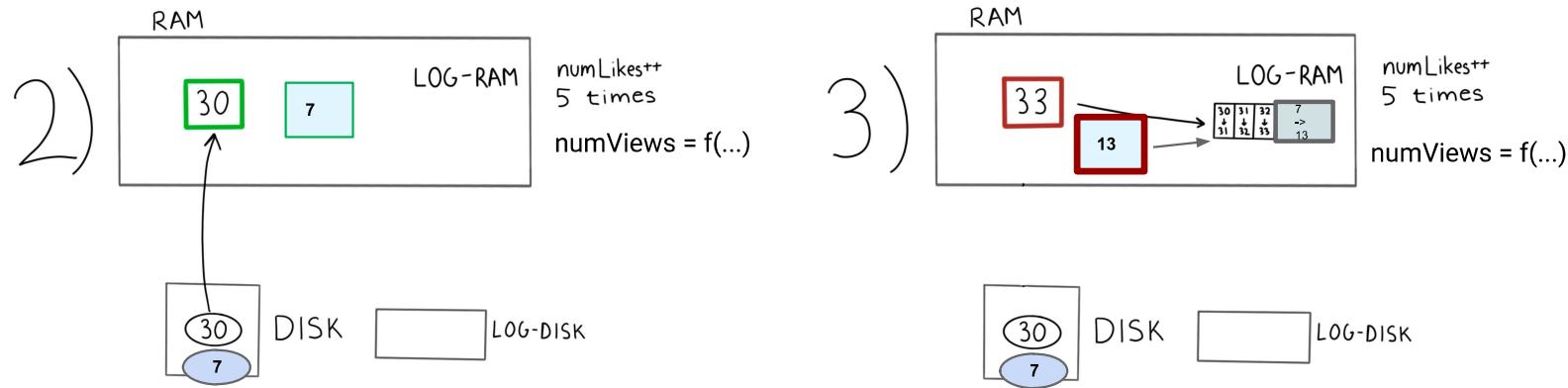
1. **A**tomicity
2. **C**onsistency
3. **I**solation
4. **D**urability

# Example1

Review TXN if there's a crash  
How to **recover**?

Extending to 1 TXN, 2 values

Extension #1



Consider updating <numLikes, numViews>.

⇒ Same process as earlier. (2) Read each value from Blocks, (3) update in LOG.



# General Algorithm. . .

## WAL Logging



## Write-Ahead Logging (WAL)

### Algorithm: WAL

For each tuple update, write Update Record into LOG-RAM

Follow two **Flush** rules for LOG

- Rule1: Flush Update Record into LOG-Disk before corresponding data page goes to storage
- Rule2: Before TXN commits,
  - Flush all Update Records to LOG-Disk
  - Flush COMMIT Record to LOG-Disk

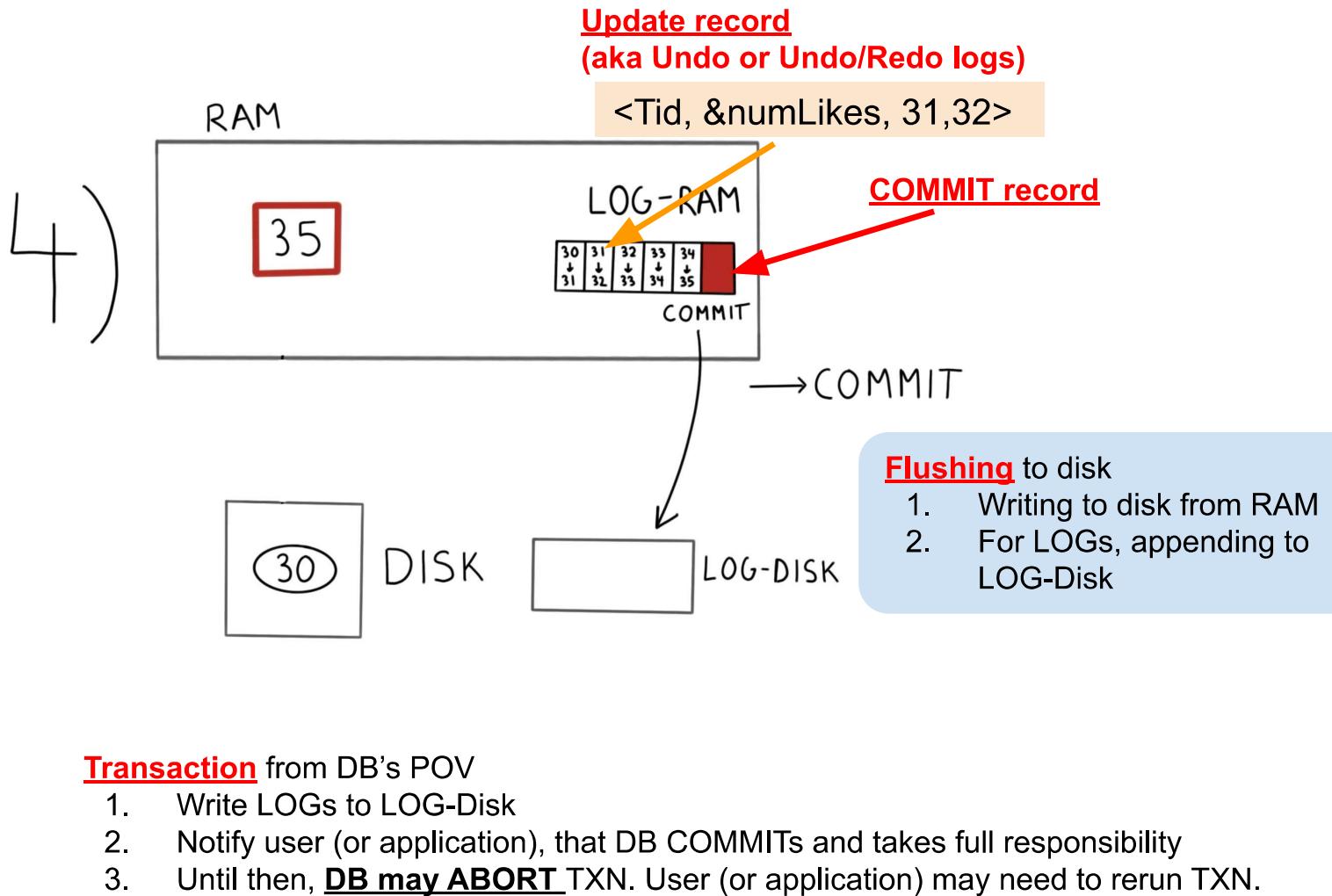
→ **Durability**

→ **Atomicity**

Transaction is committed once COMMIT record is on stable storage

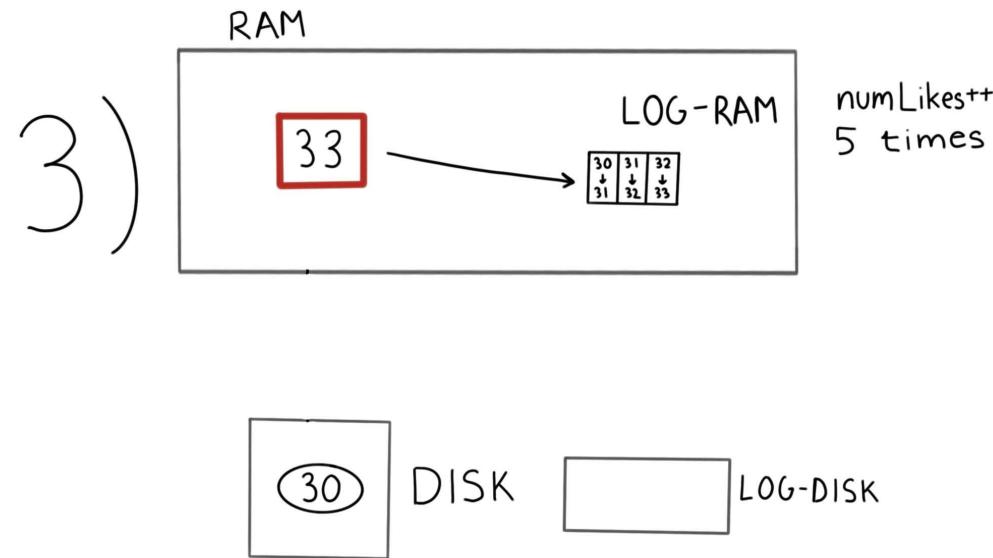


## Definitions



What if no more space in RAM?  
Or in LOG-RAM?

Extension #2



We may **need to append partial results of TXNs** on LOG-DISK due to

- Memory constraints (e.g., billions of updates)
- Time constraints (what if one TXN takes very long?)

⇒ We could use the same LOG-DISK for UNDOing partial TXNs. That is, we use the LOG for Atomicity (in addition to Durability).



# Formalizing

# Write-Ahead Logging (WAL) TXN Commit Protocol



# Write-Ahead Logging (WAL)

Note: fixing an unintended visual association. We get the combo of Durability + Atomicity, from Rules 1 and 2. Adding a 'curly brace'

## Algorithm: WAL

For each tuple update, **write** Update Record into LOG-RAM

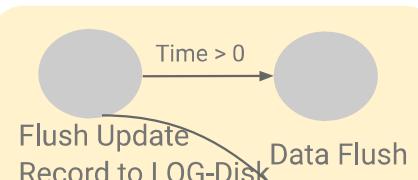
Follow two **Flush** rules for LOG

- Rule1: **Flush** Update Record *into LOG-Disk before corresponding data page goes to storage*
- Rule2: Before TXN commits,
  - **Flush** all Update Records to LOG-Disk
  - **Flush** COMMIT Record to LOG-Disk

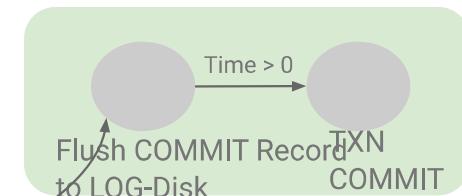
→ **Durability**

→ **Atomicity**

Transaction is committed once **COMMIT record is on stable storage**



Rule1: For each tuple update



Rule2: Before TXN commits

## Example WAL Sequence

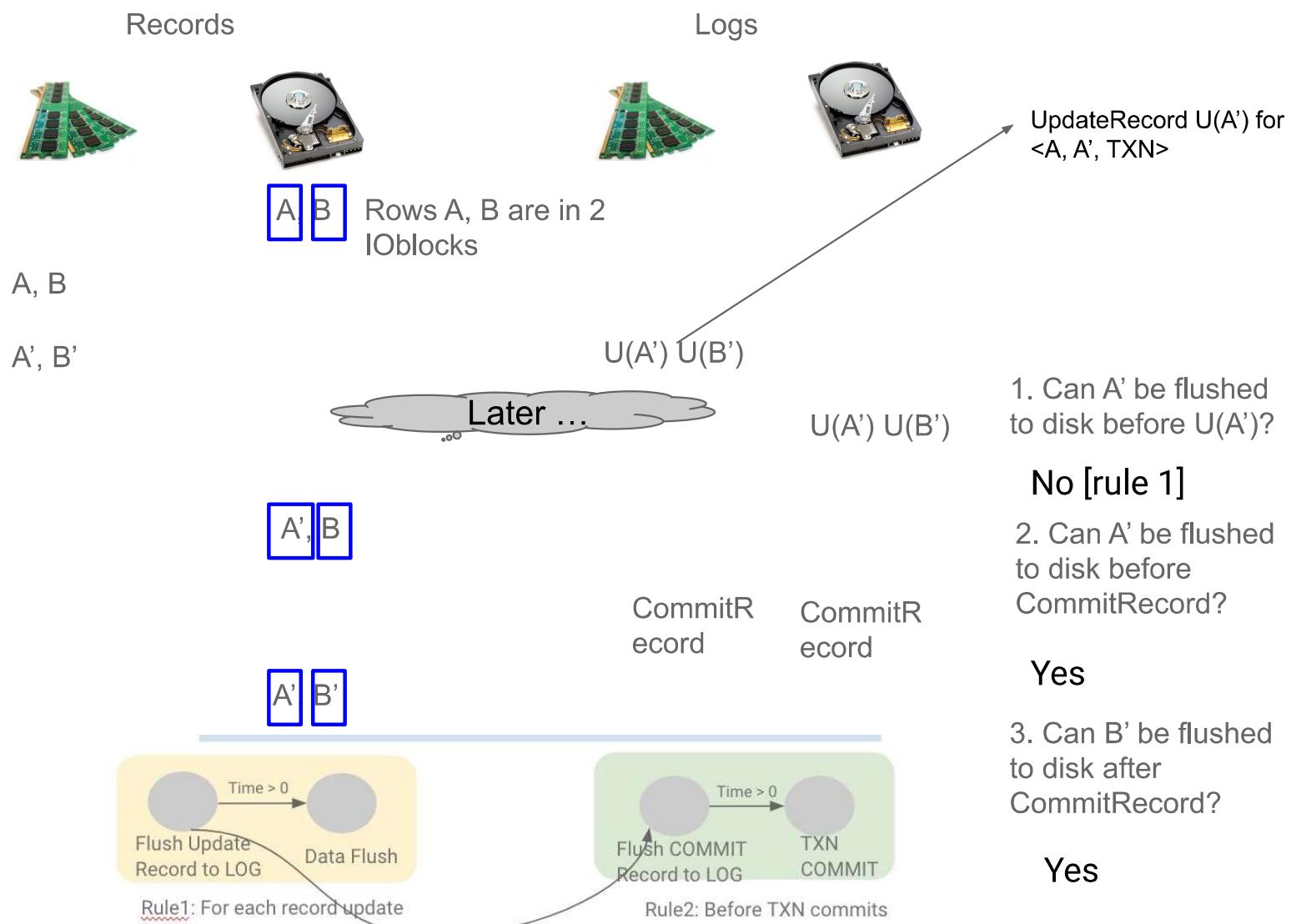
Start TXN

$R(A), R(B)$

$W(A), W(B)$

...

COMMIT TXN





# Example: Bank Transaction

# Example

Monthly  
bank  
interest  
transaction

## Full run

Money

Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...	...	
30108		-110
40008		110
50002		22

WA Log (@4:29 am day+1)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	COMMIT		

Update  
Records

Commit  
Record

'T-Monthly-423'

Monthly Interest 10%  
4:28 am Starts run on 100M bank accounts  
Takes 24 hours to run

START TRANSACTION  
UPDATE Money  
SET Amt = Amt \* 1.10  
COMMIT

# Example

Monthly  
bank  
interest  
transaction

With crash

Money

Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@10:45 am)

Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-200
5002		320
...	...	
30108		-110
40008		110
50002		22

WA Log (@10:29 am)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352

### TXN 'T-Monthly-423'

Monthly Interest 10%  
4:28 am Starts run on 100M bank accounts  
Takes 24 hours to run

Network outage at 10:29 am,  
System access at 10:45 am

?? ?? ?? ?? ??

Did T-Monthly-423 complete?  
When T-Monthly-423 was crashed  
Case2: T-Monthly-423 completed. 4002 deposited 20\$ at 10:45 am

Can you infer from **RED Update logs**?

⇒ Yup. There is NO TXN Commit. As part of recovery, undo any data updates already flushed.

# Example

Monthly  
bank  
interest  
transaction

## Recovery

Money (@10:45 am)

Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-200
5002		320
...		
30108		-110
40008		110
50002		22

Money (after recovery)

Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		
30108		-100
40008		100
50002		20

WA Log (@10:29 am)

START TRANSACTION			
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22

System recovery (after 10:45 am)

1. Rollback uncommitted TXNs as ABORTed
  - Restore old values from Log-Disk (if any)
  - Notify developers about ABORTed TXN
2. Redo Recent COMMIT-ed TXNs (w/ new values)
3. Back in business

# Example

Monthly bank interest transaction

## Performance

Money

Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...	...	
30108		-110
40008		110
50002		22

WAL (@4:29 am day+1)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	COMMIT		

### Cost to update all data

100M bank accounts → 100M seeks? (worst case)

(@10 msec/seek, that's 1 Million secs)



### Speedup for TXN Commit

1 Million secs vs 10 sec!!!

(Try other examples in [Notebook](#))

### Cost to Append to log

- + 1 access to get 'end of log'
- + write 100M log entries sequentially (fast!!! < 10 sec)

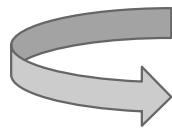
[Lazily update data on disk later, when convenient.]



# Extensions of TXNs

# LOG on Cluster model

## A popular alternative (with tradeoffs)



COMMIT by flushing log and/or data to  
'n' other machines (e.g. n = 2 )

[On same rack, different rack or  
different datacenter]

## Example

## Cluster LOG model

## Performance

### Failure model

Main model: RAM could fail, Disk is durable

VS

Cluster LOG model:

RAM on different machines don't fail at same time  
Power to racks is uncorrelated



# Example: Youtube DB

The image shows a screenshot of the YouTube mobile interface. At the top, there are two search bars: one with "funny cats" and another with "cats funny". Below each search bar is a list of video thumbnails. A blue box highlights the view count "809,337 views" for the first video in the "funny cats" search. A red box highlights the like button with "4.7K" likes and the dislike button with "768" dislikes. In the top right corner, there is a red-bordered box containing the "Upload video" and "Go live" buttons. On the far right, there is a sidebar with a "DressLily" advertisement featuring a colorful patchwork shirt and a "Shop Now" button. Below the sidebar, there are "Up next" suggestions for videos like "Top Cats Vs. Cucumbers Funny Cat Videos Compilation" and "Funny Elias play with the wheel on the bus and another toys".

YouTube

funny cats

About 12,100,000 results

How to Get Rid of Cat Pee Stains Ad BISSELL • 2M views

Your cat had an accident on the carpet. BISSELL is here to help!

REMOVE CAT PEE STAINS 1:46

CATS make us LAUGH ALL THE TIME! - Ultra FUNNY CAT 1 Tiger FunnyWorks 100K views • 3 days ago

Ultra funny cats and kitten that will make you cry with laughter! Cats are the best laugh all the time! This is ...

New

You will LAUGH SO HARD that YOU WILL FAINT - FUNNY C compilation Tiger FunnyWorks 19M views • 8 months ago

Well well well, cats for you again. But this time, even better, even funnier, even more you like these furries the ...

10:02

Have you EVER LAUGHED HARDER? - Ultra FUNNY CATS Tiger FunnyWorks 124K views • 1 week ago

Super funny cats and kitten that will make you scream with laughter! This is the LAUGH challenge ever!

10:02

809,337 views

Baby Cats - Funny and Cute Baby Cat Videos Compilation (2018) Gatitos Bebes Video Recopilación

Animal Planet Videos Published on May 23, 2018

4.7K 768 SHARE

SUBSCRIBE 337K

Baby Cats - Funny and Cute Baby Cat Videos Compilation (2018) Gatitos Bebes Video Recopilación | Animal Planet Videos

Subscribe Here: <https://goo.gl/qor4XN>

SHOW MORE

Upload video

Go live

Shop Now > DressLily

Up next

Top Cats Vs. Cucumbers Funny Cat Videos Compilation Animal Planet Videos 23M views

Funny Elias play with the wheel on the bus and another toys Elias Adventures 291 watching LIVE NOW

LIVE: Rescue kitten nursery TinyKittens HQ 1.3K watching LIVE NOW

Puss in boots and the three diablos [HD] Mathew Garcia 7.6M views



Example: Increment number of LIKES per Video

Design 1: WAL Log for Video Likes

<videoid, numLikescount, new numLikes count>

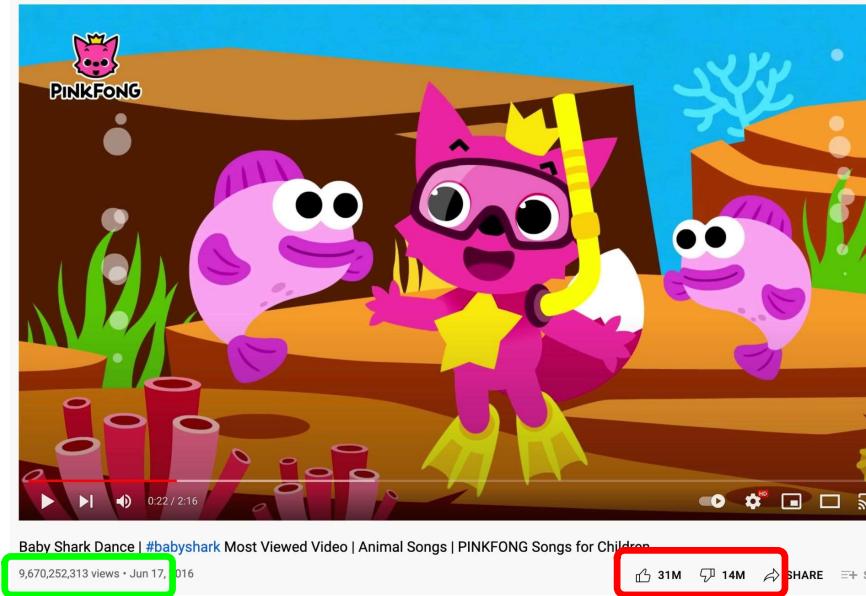
WAL for Video likes			
T-LIKE-4307	<b>START TRANSACTION</b>		
T-LIKE-4307	3001	537	538
T-LIKE-4307	<b>COMMIT</b>		
T-LIKE-4308	<b>START TRANSACTION</b>		
T-LIKE-4308	5309	10001	10002
T-LIKE-4308	<b>COMMIT</b>		
T-LIKE-4309	<b>START TRANSACTION</b>		
T-LIKE-4309	3001	538	539
T-LIKE-4309	<b>COMMIT</b>		
...	...	...	...
T-LIKE-4341	5309	10002	10003
T-LIKE-4351	5309	10003	10004
...	...	...	...
T-LIKE-4383	<b>START TRANSACTION</b>		
T-LIKE-4383	5309	10004	10005
T-LIKE-4383	<b>COMMIT</b>		

Critique?

Correct?

Write Speed? Cost? Storage?

Bottlenecks?





### Design 2:

- Replicate numLikes count in cluster.
- Batch updates in Log (e.g. batch increase counts by 100 or 1000 to amortize writes, based on video popularity)

Update RAM on  
n=3 machines  
(<videoid, #likes>)



Critique?

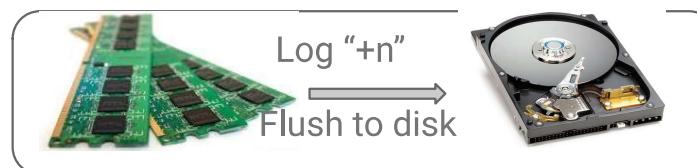
Correct?

Write Speed? Cost? Storage?

Bottlenecks?

System recovery?

Micro-batch updates		
Txn (e.g, Timestamp)	Videoid	Batch Increment
1539893189	3001	100
1539893195	5309	5000
1539893225	3001	200
	..	400
	5309	100
	...	5000
	5309	100000
	...	10
1539893289	5309	10



## Example

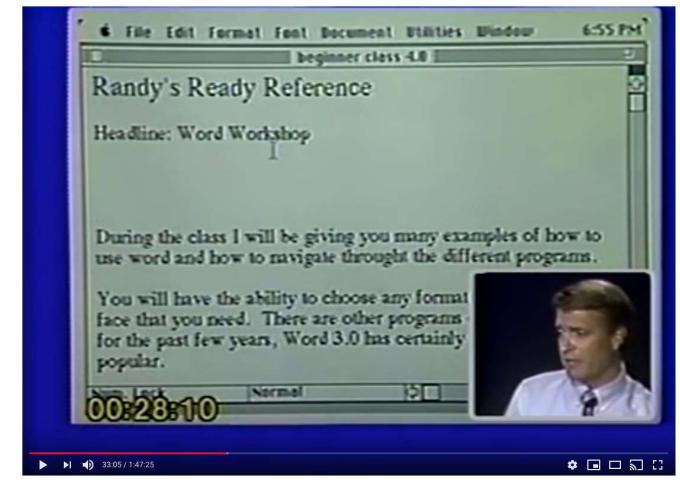
## Youtube writes

## Performance

## Vs Cost



Popular video



World's most boring video (Unpopular)

## Design #3

For most videos, Design 1 (full WAL logs)

For popular videos, Design 2

## Critique?

Correct?

Write Speed? Cost? Storage?

Bottlenecks?

System recovery?

# Summary

## Design Questions?

Correctness: Need true ACID? Pseudo-ACID? What losses are OK?

Design parameters:

Any data properties you can exploit? (e.g., '+1', popular vs not)

How much RAM, disks and machines?

How many writes per sec?

How fast do you want system to recover?

Choose: WAL logs, Replication on n-machines, Hybrid? (More in cs345)

# Transactions

## Summary

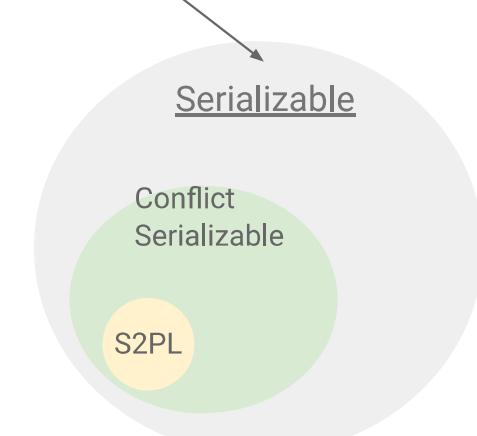
### Why study Transactions?

Good programming model for parallel applications on shared data !

- Atomic
- Consistent
- Isolation
- Durable



### Next Lecture



Note: this is an intro  
Next: Take 346 (Distributed Transactions) or read [Jim Gray's](#) classic



# Logging Summary

- If DB says TX commits, TX effect remains after database crash
- DB can undo actions and help us with atomicity
- This is only half the story...



# Lecture: Concurrency & Locking for Transactions

# Problems

Problem1: How can a {Bank, Visa, Fintech, NASDAQ} update 100 million records correctly?

Problem2: How to update 100 million records in **Seconds** (not 1000s to 1 million secs)?

Problem3: How to let users have access their accounts?

Problem4: Make it easy for the application engineer!!!

# Example

Monthly bank interest transaction

Money

Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...	...	
30108		-110
40008		110
50002		22

'T-Monthly-423'

Monthly Interest 10%  
4:28 am Starts run on 10M bank accounts  
Takes 24 hours to run

UPDATE Money  
SET Balance = Balance \* 1.1



## Other Transactions

- 10:02 am Acct 3001: Wants 600\$
- 11:45 am Acct 5002: Wire for 1000\$
- .....
- .....
- 2:02 pm Acct 3001: Debit card for \$12.37

Q: How do I not wait for a day to access my \$\$\$s?

# Big Idea LOCKS!

## Big Idea: LOCKs

- ▷ **Intuition:**
  - ‘Lock’ each record for shortest time possible
    - (e.g, Locking Money Table for a day is not good enough)
- ▷ **Key questions:**
  - Which records? For how long? What’s algorithm?



Many kinds of LOCKs. We'll study some simple ones!



What you will learn about in this section

## 1. Concurrency

- ▷ Interleaving & scheduling (Examples)
- ▷ Conflict & Anomaly types (Formalize)

## 2. Locking: 2PL, Deadlocks (Algorithm)



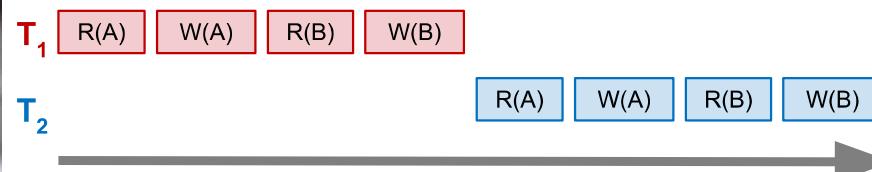
2 TXNs, 2 Values



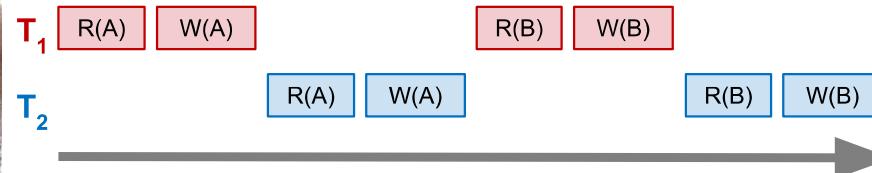
# Speed up TXNs – Interleaving TXNs for Concurrency

## Serial Schedule

We use **R(A)** and **W(A)** for a TXN reading and writing value 'A' in RAM (e.g., money or numLikes)



## Interleaved Schedule



We call the particular order of interleaving a **schedule**

**Scenario:** DB gets T1 and T2 at roughly the same time. {T1, T2} are **Concurrent TXNs**.

How will DB execute TXNs in **RAM + CPU**?

1. DB guarantees **Isolation** in ACID
2. DB does not guarantee which order T1 and T2 are executed in. Must look as if the TXNs executed serially!
3. DB **interleaves** R/Ws for performance.

(Why? E.g., T1's R(A) and R(B) may take IO time. While waiting for T1's R(B), do T2's R(A) and W(A), because A's page already in RAM.)

# Example- consider two TXNs:

T1: START TRANSACTION

    UPDATE Accounts

    SET Amt = Amt + 100

    WHERE Name = 'A'

    UPDATE Accounts

    SET Amt = Amt - 100

    WHERE Name = 'B'

    COMMIT

T1 transfers \$100 from B's account to A's account

T2: START TRANSACTION

    UPDATE Accounts

    SET Amt = Amt \* 1.06

    COMMIT

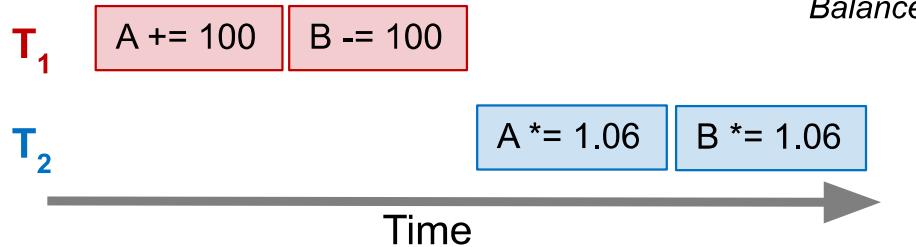
T2 credits both accounts with a 6% interest payment

## Note:

1. DB does not care if T1 → T2 or T2 → T1 (which TXN executes first)
2. If developer does, what can they do? (Put T1 and T2 inside 1 TXN)

# Scheduling examples

Serial schedule S1 ( $T_1$  runs then  $T_2$ )



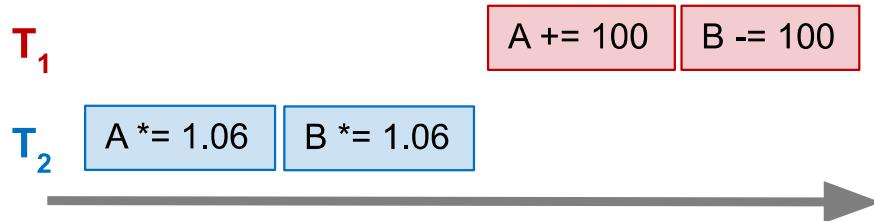
Starting  
Balance

A	B
\$50	\$200

A	B
\$159	\$106

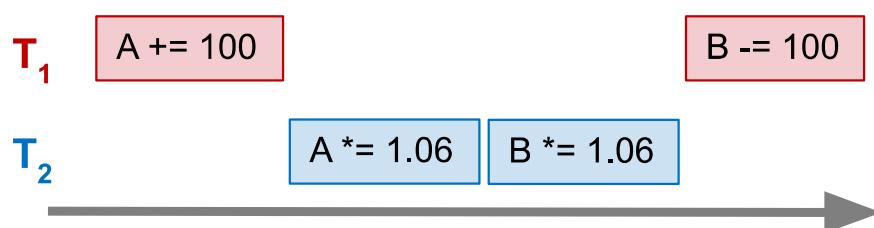
S1 and S2 give two results, but OK!  
ACID not focused on TXN sequence.

Serial schedule S2 ( $T_2$  runs then  $T_1$ )



A	B
\$153	\$112

Interleaved schedule S6:



Different result than S1 or S2.  
Not OK! ACID needs Isolation.

A	B
\$159	\$112



# Scheduling Definitions

- A **serial schedule** does not interleave TXNs
- S1 and S2 are **equivalent schedules**
  - if the effect of executing S1 is **identical** to executing S2, **for any database state**
- A **serializable schedule** is a schedule equivalent to **some** serial schedule.

The word “**some**” makes this definition powerful & tricky!

Serial Schedules

T1		A += 100	B -= 100		
T2				A * = 1.06	B * = 1.06

S1

T1				A += 100	B -= 100
T2			A * = 1.06	B * = 1.06	

S2

Interleaved Schedules

T1			A += 100		B -= 100	
T2				A * = 1.06		B * = 1.06

S3

T1					A += 100	B -= 100
T2			A * = 1.06		B * = 1.06	

S4

T1				A += 100	B -= 100	
T2			A * = 1.06			B * = 1.06

S5

T1			A += 100			B -= 100
T2				A * = 1.06	B * = 1.06	

S6

Serial Schedules	S1, S2
Serializable Schedules	S3, S4 (And S1, S2)
Equivalent Schedules	<S1, S3> <S2, S4>
Non-serializable (Bad) Schedules	S5, S6



What you will learn about in this section

## 1. Concurrency

- ▷ Interleaving & scheduling (Examples)
- ▷ Conflict & Anomaly types (Formalize)

## 2. Locking: 2PL, Deadlocks (Algorithm)



# Conflicts and Anomalies



# Conflicting Actions

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

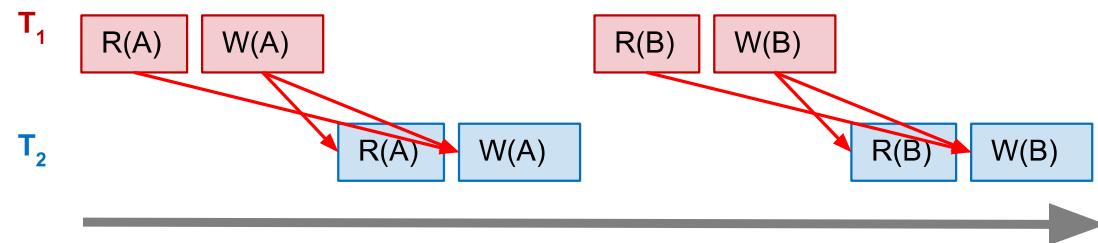
Three types of conflicts:

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

*Why no “RR Conflict”?  
See definition above. RRs  
don’t change values.*

**Conflicts** may happen for concurrent TXNs

- Can be in both “good” and “bad” schedules





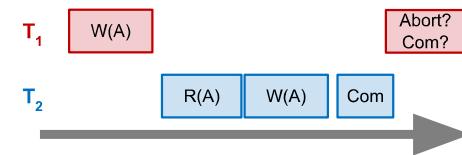
## But need to Avoid Anomalies

When interleaving TXNs, do not create “bad” (anomalous) schedules  
(i.e., break isolation and/or consistency)

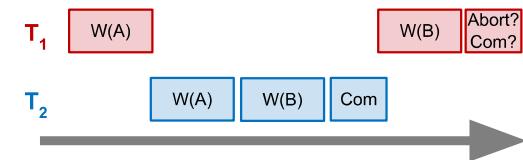
**“Unrepeatable read”:**



**Reading uncommitted data:**  
("Dirty read")



**Partially-lost update:**

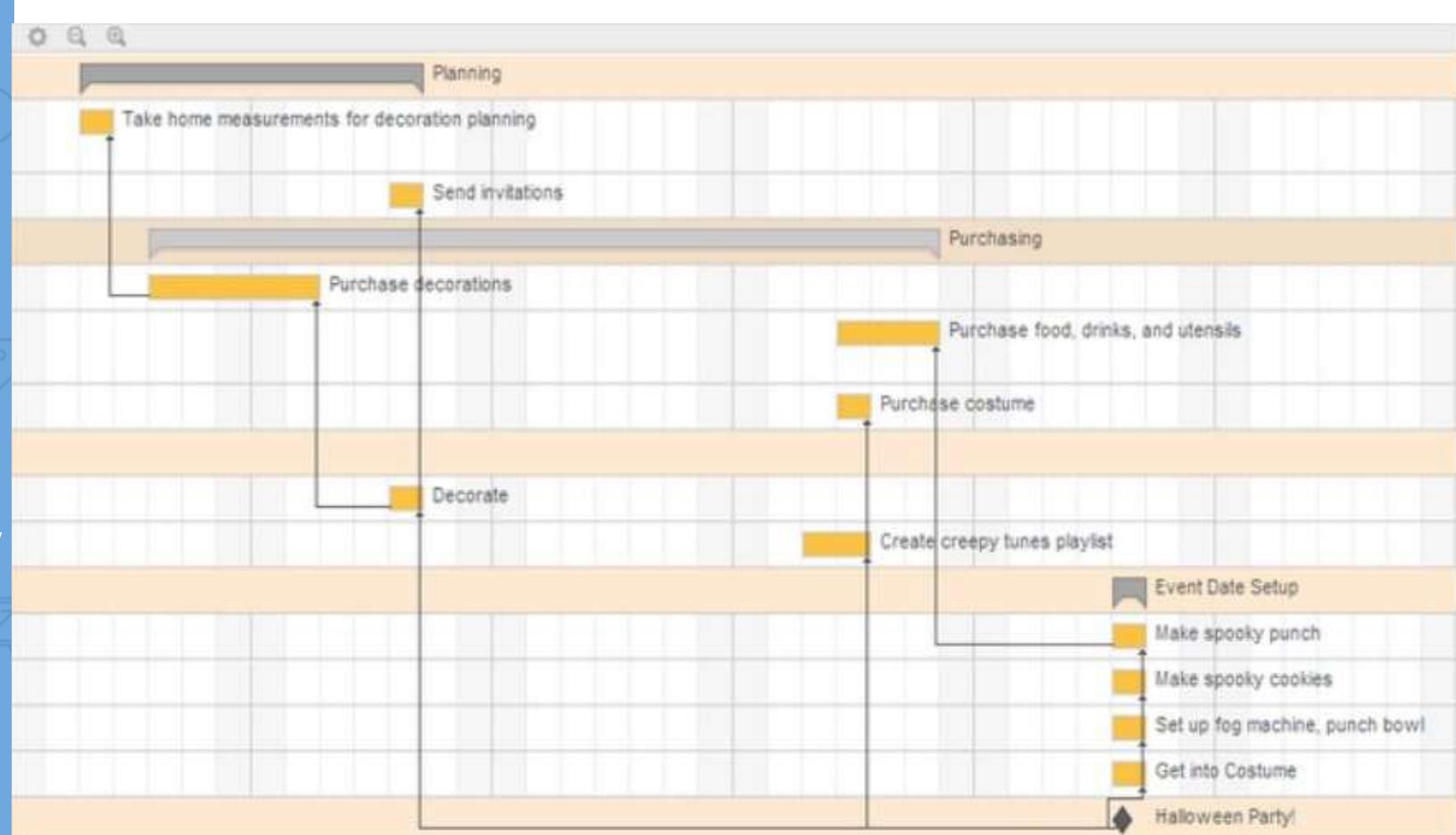




# Conflict Serializability

## Example TODO list dependencies

(Intuition for DAGs/  
TopoSort)



How would you plan?  
What if there are cycles? (dependencies)

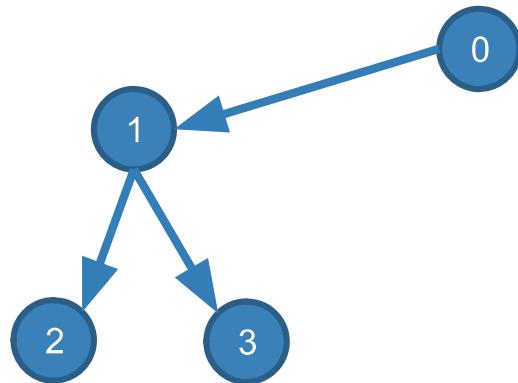


# CS Concept Reminder: DAGs & Topological Orderings

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges
- A directed **acyclic** graph (DAG) always has one or more **topological orderings**
  - (And there exists a topological ordering *if and only if* there are no directed cycles)

# DAGs & Topological Orderings

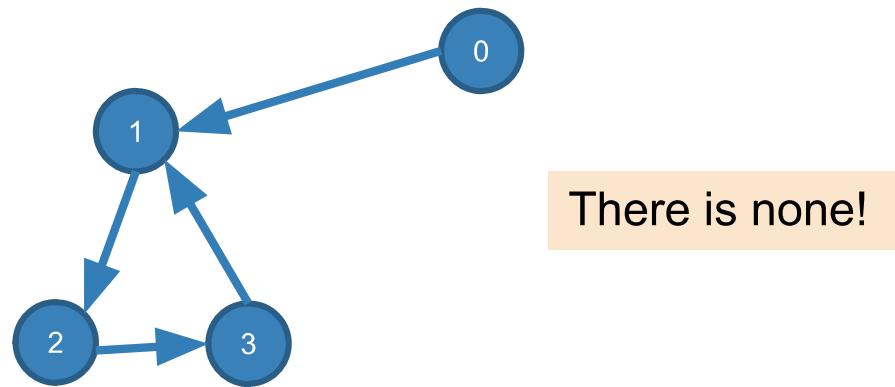
- Ex: What is one possible topological ordering here?



Ex: 0, 1, 2, 3 (or: 0, 1, 3, 2)

# DAGs & Topological Orderings

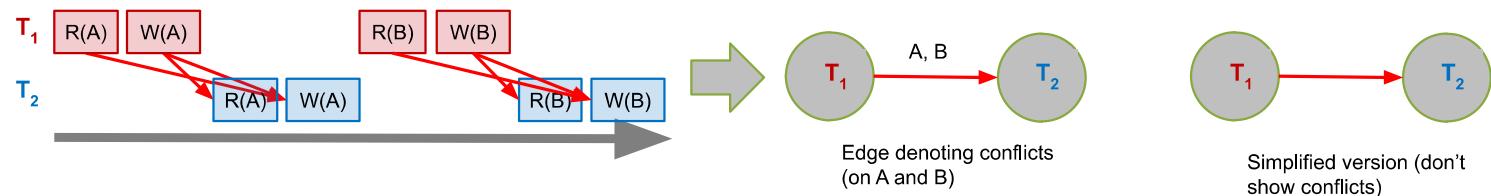
- Ex: What is one possible topological ordering here?



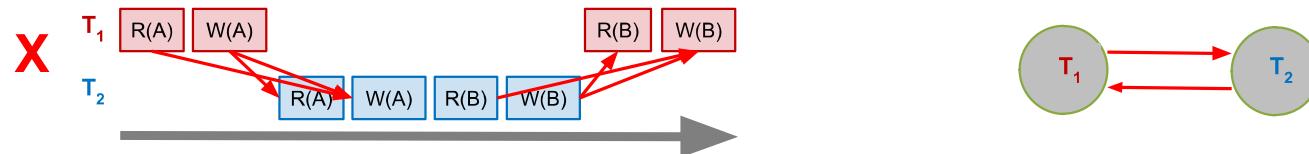


# The Conflict Graph

- For a given schedule, compute Conflict Graph
  - TXNs as **nodes**, and
  - Edge from  $T_i \rightarrow T_j$  if any actions in  $T_i$  precede and conflict with any actions in  $T_j$



**'Bad' Schedule**



⇒ Given an acyclic Conflict Graph, a **topological** ordering of TXNs corresponds to a **serial schedule of TXNs**

### Example with 5 Transactions

Given: Schedule S1

	w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)
--	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

E.g., w3(C) is short for "T3 Writes on C"

Good or Bad schedule?  
Conflict serializable?

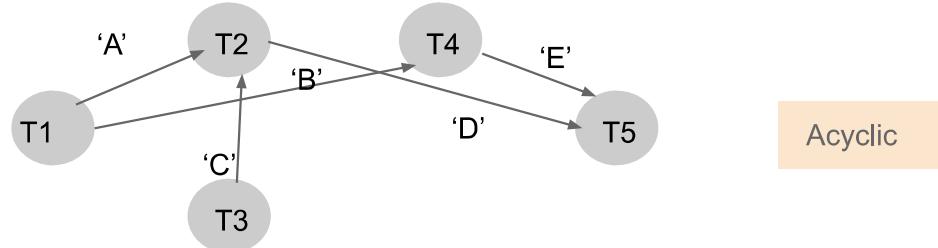
### Step1

Find conflicts  
(RW, WW, WR)

T1	w1(A)		w1(B)							
T2		r2(A)			r2(C)		w2(D)			
T3				w3(C)						
T4					r4(B)			w4(E)		
T5									r5(D)	w5(E)

### Step2

Build Conflict graph  
Acyclic? Topo Sort



### Step3

Example serial schedules  
Conflict Equiv to S1

	T3	T1	T4	T2	T5		SerialSched (SS1)			
	w3(C)	w1(A)	w1(B)	r4(B)	w4(E)	r2(A)	w2(D)	r5(D)	w5(E)	
	T1	T3	T2	T4	T5		SerialSched (SS2)			
	w1(A)	w1(B)	w3(C)	r2(A)	r2(A)	w2(D)	r4(B)	w4(E)	r5(D)	w5(E)



# Conflict Serializability for a concurrent set of TXNs

1. Two schedules S1 and S2 are **conflict equivalent** if:
  - a. Their Conflict Graphs have the same directed edges, for the same nodes and actions
  - b. I.e., every pair of *Conflicting Actions* is *ordered in the same way* (for the *same actions of the same TXNs*)
2. Special case: If a schedule S1 is **conflict equivalent to serial schedule**, SS1
  - a. S1 is **conflict serializable**. (That is, conflict equivalent and serializable)
  - b. ConflictGraph(SS1) has no cycles.  $\Rightarrow$  ConflictGraph(S1) also has no cycles.
  - c. That is, S1 is conflict serializable to a serial schedule, i.e., serializable



# Conflict Serializability Theory and Practice

**Theorem:**

Schedule is conflict serializable if-and-only-if Conflict Graph is acyclic

## Good vs Bad schedule

1. Build Conflict Graph for concurrent TXNs (not for past or future TXNs)
  - a. Check if Conflict Graph is acyclic. Conflict serializable  $\Rightarrow$  Serializable to serial schedule  $\Rightarrow$  Isolation
  - b. Else, anomalies. Avoid!

\* Later, we'll see a Locking algorithm to produce good schedules

# Summary

- Concurrency achieved by **interleaving TXNs** such that **isolation & consistency** are maintained
  - We formalized a notion of **serializability** that captured such a “good” interleaving schedule
  - We defined **conflict serializability**



Put all this  
machinery  
together NEXT  
week

Quick intuition for use cases ?

1. Construction

Locking algorithms to produce good schedules

2. Optimization

Optimizer may take a schedule and reorder (if disk is slow, etc.)



# 2 TXNs 2 Values . . .Locking and Concurrency

(now that we understand properties of schedules we want)

# Big Idea LOCKS!

## Big Idea: LOCKs

- ▷ **Intuition:**
  - ‘Lock’ each record for shortest time possible
    - (e.g, Locking Money Table for a day is not good enough)
- ▷ **Key questions:**
  - Which records? For how long? What’s algorithm?

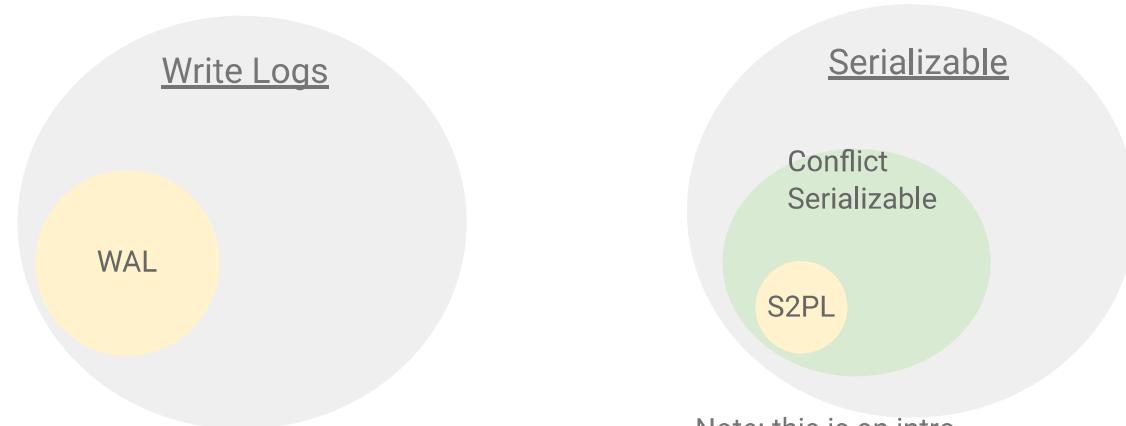


We now have the tools to BUILD such locks.



## Strict 2 PL (S2PL) Locking

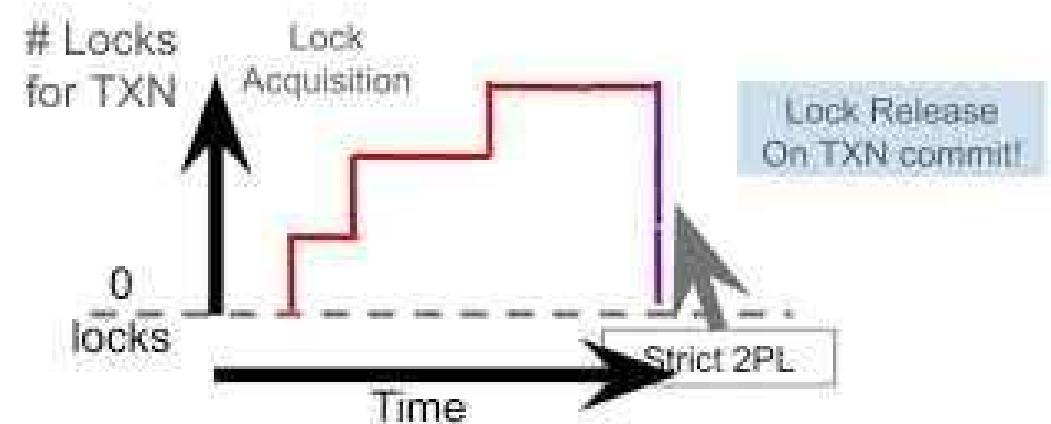
## Putting it all together -- ACID Transactions



Note: this is an intro  
Next: Take 245/346 (Distributed Transactions) or read  
[Jim Gray's](#) classic



## Strict 2-Phase Locking (S2PL)



**2-Phase Locking:** A transaction can not request additional locks once it releases any locks. [Phase 1: "growing phase" to get more locks. Phase 2: "shrinking phase"]

**Strict 2-PL:** Release locks only at COMMIT (COMMIT Record flushed) or ABORT.



# Locks

TXNs obtain:

1. An **X (exclusive) lock** on object before **writing**.  
⇒ No other TXN can get a lock (S or X) on that object.  
(e.g, X('A') is an exclusive lock on 'A')
2. An **S (shared) lock** on object before **reading**  
⇒ No other TXN can get an X lock on that object
3. All locks held by a TXN are released when TXN completes.

Waits-for graph:

- Nodes are transactions
- There is an edge from  $T_i \rightarrow T_j$  if  $T_i$  is *waiting for  $T_j$  to release a lock*

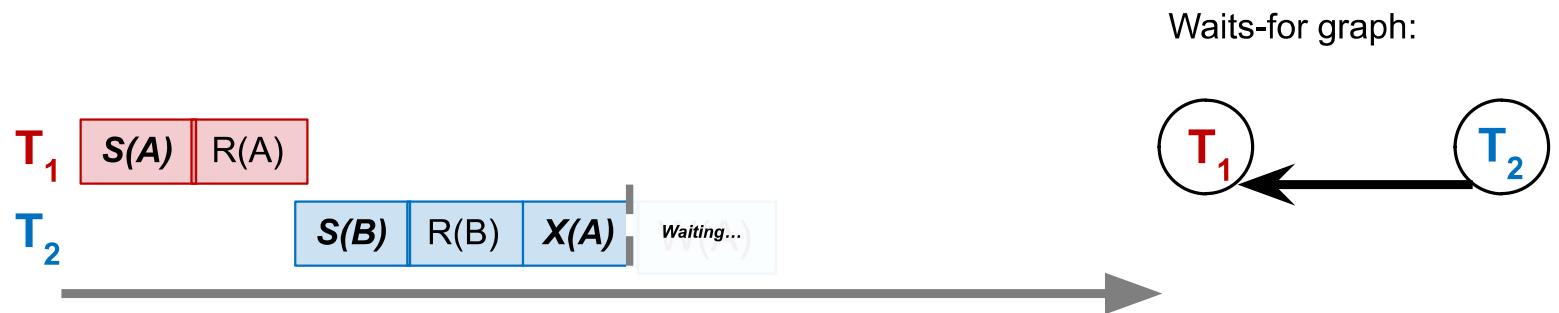


# Deadlock Detection

Create the **waits-for** graph:

- Nodes are transactions
- There is an edge from  $T_i \rightarrow T_j$  if  $T_i$  is *waiting for  $T_j$  to release a lock*

## 2 TXN, 2 Values example

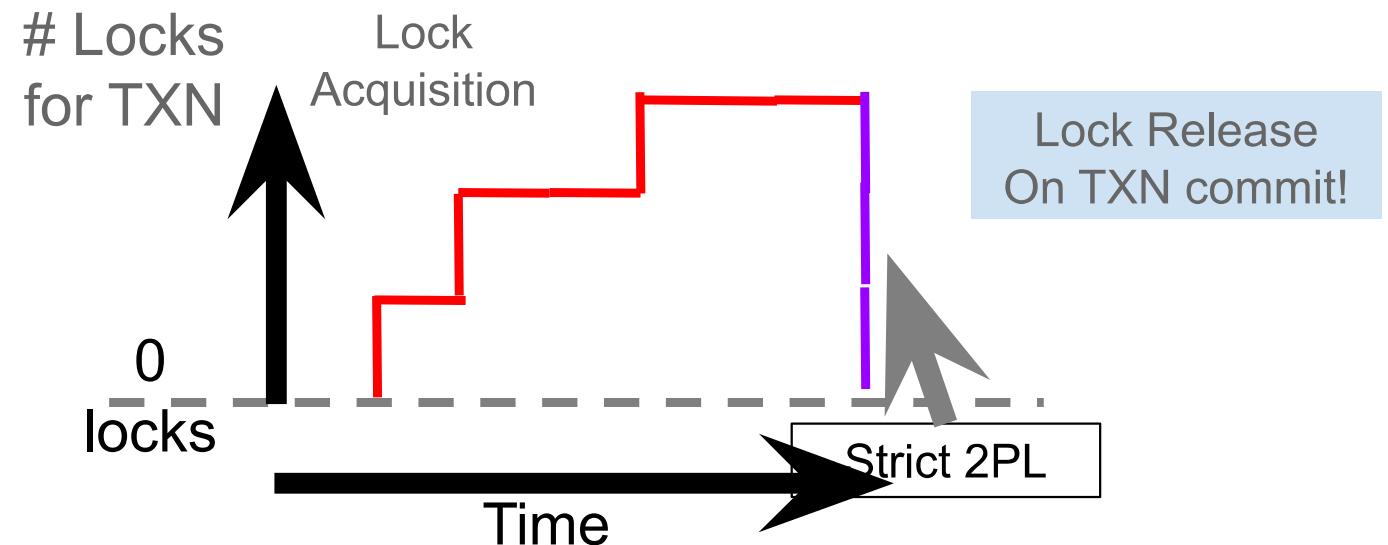


1. First,  $T_1$  requests a shared lock on A to read from it
2.  $T_2$  requests a shared lock on B to read from it
3.  $T_2$  requests an exclusive lock on A to write to it- **now  $T_2$  is waiting on  $T_1$ ...**

Waits-For graph: Track which Transactions are waiting  
IMPORTANT: WAITS-FOR graph different than CONFLICT graph we learnt earlier !



# Strict 2-Phase Locking (S2PL)



**2-Phase Locking:** A transaction can not request additional locks once it releases any locks. [Phase1: “growing phase” to get more locks. Phase2: “shrinking phase”]

**Strict 2-PL:** Release locks only at COMMIT (COMMIT Record flushed) or ABORT



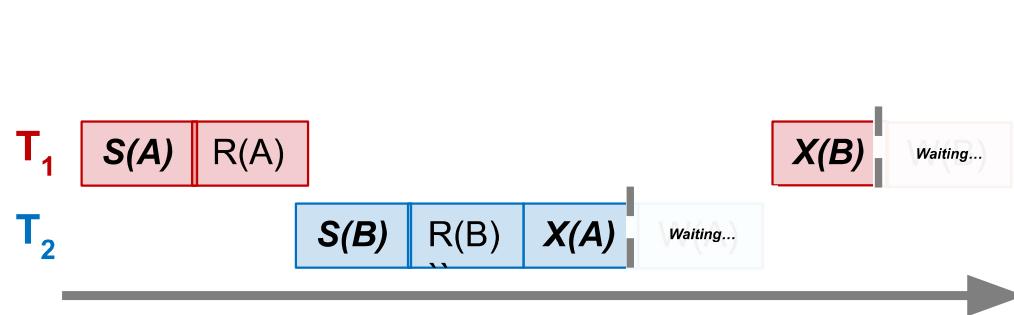
## Strict 2PL

S2PL produces **conflict serializable** schedules...  
...and we get isolation & consistency!

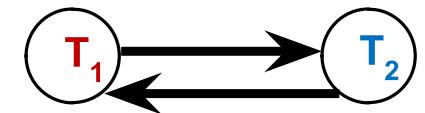
Popular implementation

- Simple !
- Produces subset of \*all\* conflict serializable schedules
- There are MANY more complex LOCKING schemes with better performance. (See CS 245/ CS 345)
- One key, subtle problem (next)

# Deadlock Detection: Example



Waits-for graph:



Cycle =  
DEADLOCK

4. Later  $T_1$  requests an exclusive lock on B to write to it-  
**now  $T_1$  is waiting on  $T_2$ ... DEADLOCK!**



# Deadlocks

**Deadlock:** Cycle of transactions waiting for locks to be released by each other.

Options to handle Deadlocks

- Deadlock prevention
- Deadlock detection: Periodically check for (**and break**) cycles in the waits-for graph

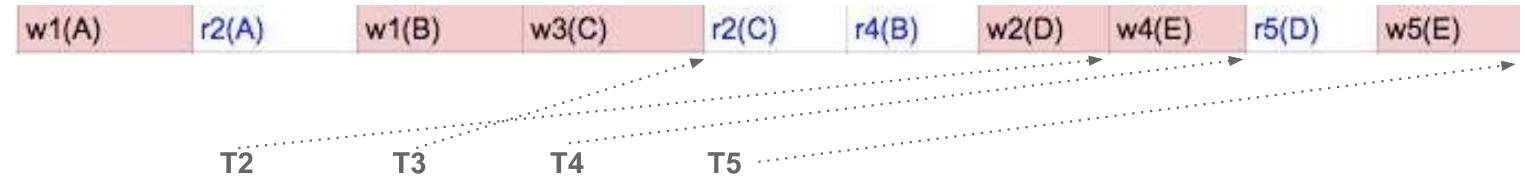


# 5 TXNs 5 Values Example

## . . . Locking and Concurrency

Example with 5 Transactions (S2PL)

Given: Schedule S1



Execute with S2PL

T1

T2

T4

T1

T3

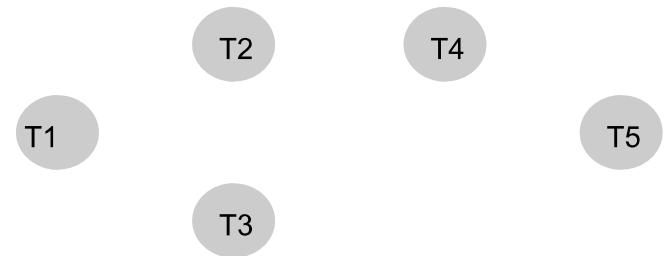
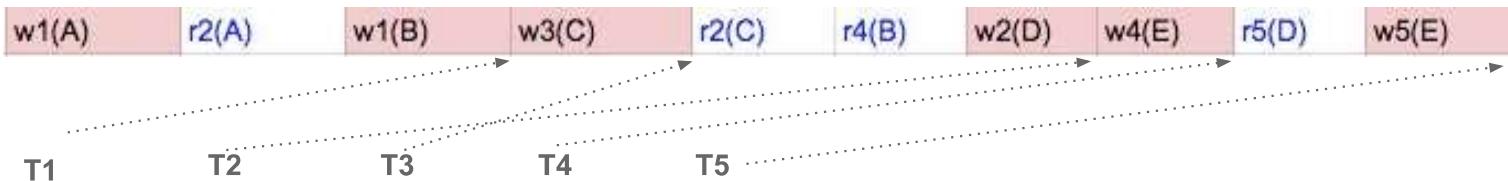
T5

Waits- For Graph

Example with 5 Transactions (S2PL)

Given: Schedule S1

Execute with S2PL



Waits- For Graph

### Example with 5 Transactions (S2PL)

Given: Schedule S1

Execute with S2PL

Step 0

X (A)  
**w1(A)**

Req S(A)

Step 1

X (B)  
**w1(B)**

Unl B, A

Step 2

Step 3

Step 4

Step 5

Step 6

Step 7

Step 8

Step 9

Step 10

Get S(A)  
**r2(A)**

X (C)  
**w3(C)**  
Unl C

S(C)  
**r2(C)**

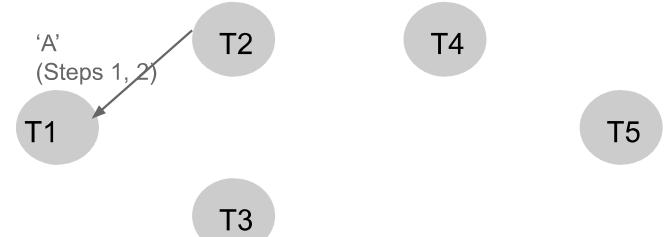
S(B)  
**r4(B)**

X(D)  
**w2(D)**  
Unl A, C, D

X(E)  
**w4(E)**  
Unl B, E

S (D)  
**r5(D)**

X (E)  
**w5(E)**  
Unl D, E



Waits- For Graph

### Example with 5 Transactions (S2PL)

Schedule S1

Execute with S2PL

Step 0

Step 1

Step 2

Step 3

Step 4

Step 5

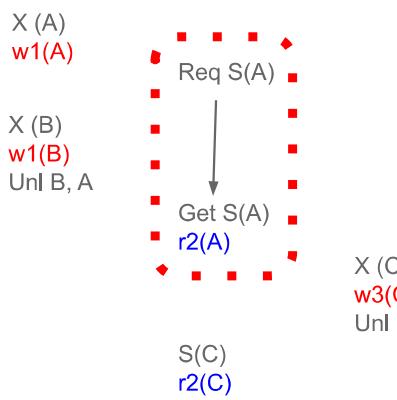
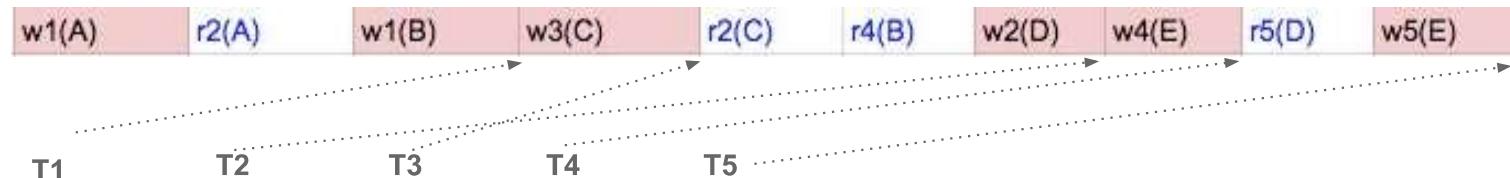
Step 6

Step 7

Step 8

Step 9

Step 10



S(B)  
r4(B)

X(D)  
w2(D)

Unl A, C, D

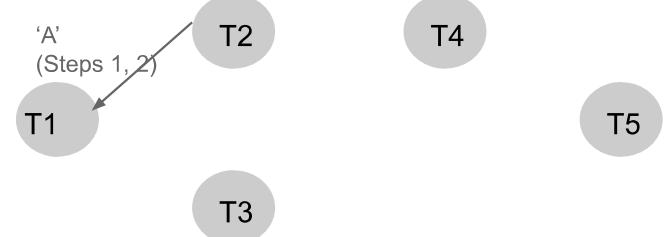
X(E)  
w4(E)

Unl B, E

S (D)  
r5(D)

X (E)  
w5(E)

Unl D, E



Waits- For Graph

# Example: What happened?

Input Schedule



S2PL's Reordered Schedule Executed



In general, given a set of TXNs, i.e., schedules  
S2PL() produces conflict-serializable schedules.



# Many TXNs and Values

...Locking and Concurrency

# DB Scheduler

## How does it all work?

```
// Scheduler keeps a Locks table for Waits-For-Graph
// Plans and re-plans Schedules to handle correctness and performance

DBScheduler(TXN set with all R/W actions)
    1. Make a list of S and X locks to request. (e.g., W(A) ⇒ Request X(A))
    2. Estimate costs. Plan a rough schedule S1 (e.g., data in RAM? JOINs?)
    3. Start executing next action in S1.

        // Requests and updates Locks table
        a. If a Lock is available for next action, take it. Execute action
        b. If a Lock is unavailable, add to Waits-For-Graph.
        c. If Deadlock in Waits-For-Graph, break cycles. (e.g., abort one of the TXNs
           and redo by adding the TXN set)
    4. At the end of a TXN, release relevant locks.
        a. [If S2PL, release Locks only when TXN Commits/Aborts]
        b. Unblock the next action in Waits-For.
        c. Repeat 3 and 4 until done.
```

Broader  
picture on  
Concurrency

[Take  
cs245/cs345  
next for more]

## Locks

S, X

Increment

Upgrade Locks

Intention Locks

## Locking Protocol

2 PL

S2PL

Tree-based

## Scheduling

Serializability

Conflict-Serializability

View Serializability

Timestamp-based

Validation-based

## Tradeoffs

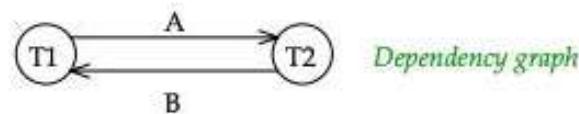
1. S2PL easiest, also slowest. E.g., 2PL is faster, but subtle problems (read: cascading rollbacks in 245/345)
2. (In red) Richer locks, faster protocols, more flexible definitions of correctness for speed

For cs145 in Fall'22, mostly focus on Strict 2PL for our tests, homeworks

## Example2

- A schedule that is not conflict serializable:

T1:	R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)	



If you Input above schedule into S2PL(), what would happen?

[Ans: R2(A) blocked until after W1(B). Therefore, conflict serialized. i.e. $T1 \rightarrow T2$ ]

# Transactions

## Summary

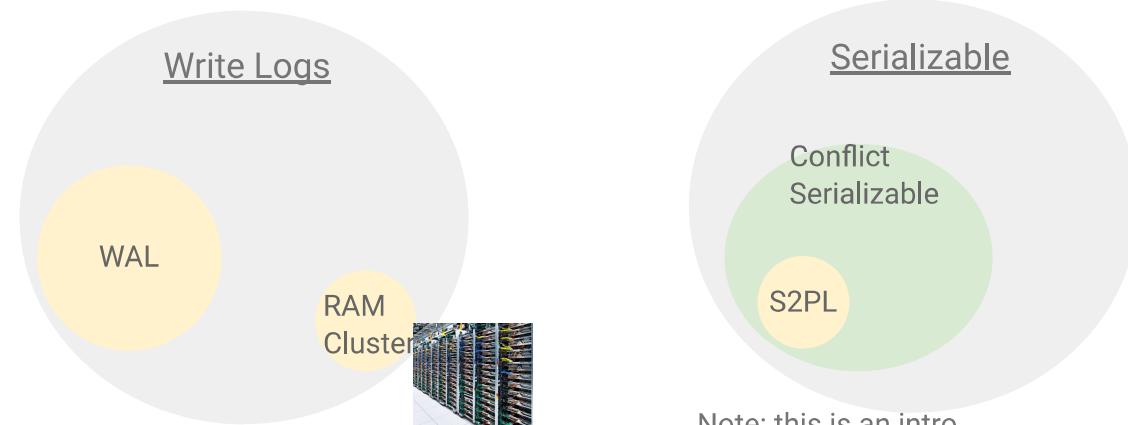
### Why study Transactions?

Good programming model for parallel applications on shared data !

Atomic  
Consistent  
Isolation  
Durable

### Design choices?

- Write update Logs (e.g., WAL logs)
- Serial? Parallel, interleaved and serializable?



Note: this is an intro  
Next: Take 346 (Distributed Transactions) or read [Jim Gray's](#) classic



# Putting it all together

# Example

Monthly  
bank  
interest  
transaction

Money

Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...	...	
30108		-110
40008		110
50002		22

'T-Monthly-423'

Monthly Interest 10%  
4:28 am Starts run on 10M bank accounts  
Takes 24 hours to run

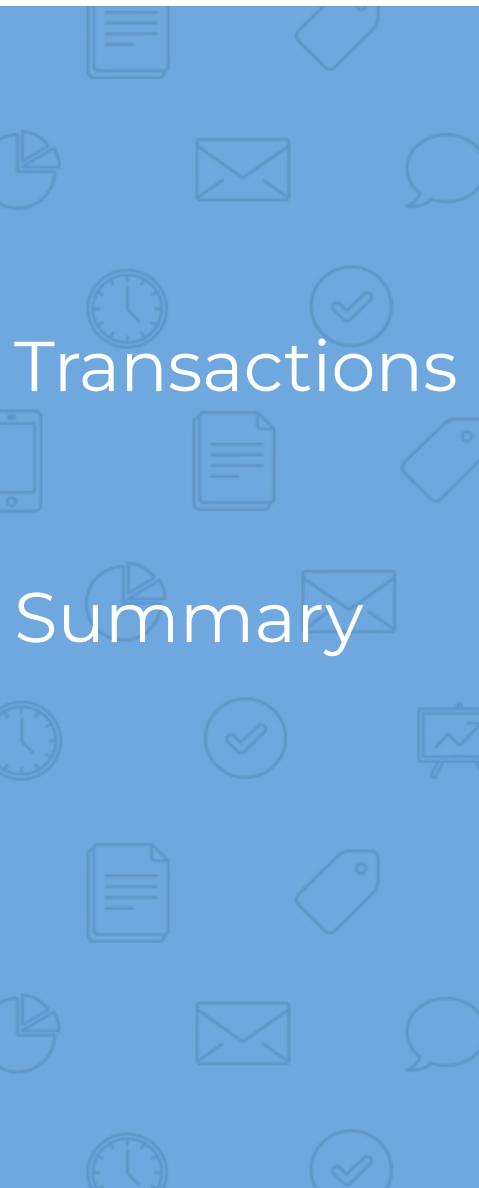
UPDATE Money  
SET Balance = Balance \* 1.1



## Other Transactions

- 10:02 am Acct 3001: Wants 600\$
- 11:45 am Acct 5002: Wire for 1000\$
- .....
- .....
- 2:02 pm Acct 3001: Debit card for \$12.37

Q: How do I not wait for a day to access \$\$\$s?



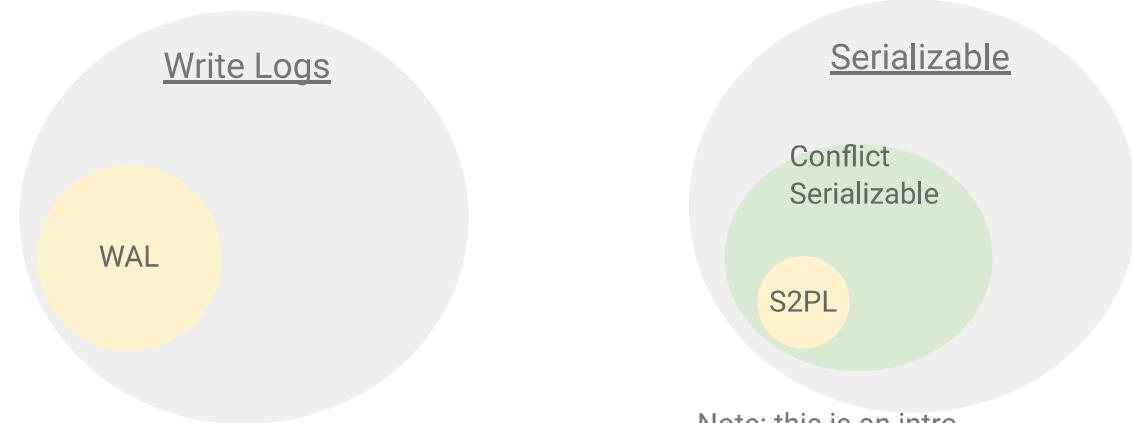
### Why study Transactions?

Good programming model for parallel applications on shared data !

Atomic  
Consistent  
Isolation  
Durable

### Design choices?

- Write update Logs (e.g., WAL logs)
- Serial? Parallel, interleaved and serializable?



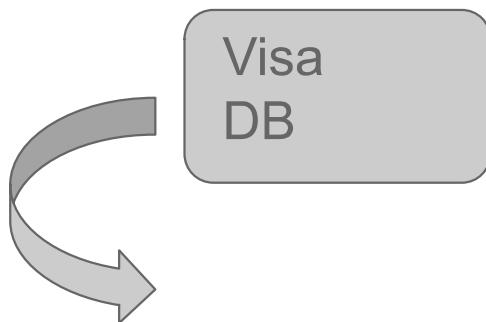
Note: this is an intro  
Next: Take 245/346 (Distributed Transactions) or read  
[Jim Gray's](#) classic

## Example Visa DB



### Transaction Queue

- 60000 user TXNs/sec
- Monthly 10% Interest TXN



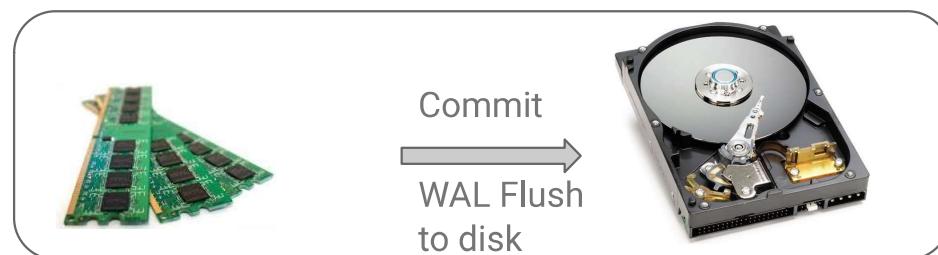
Account	...	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20



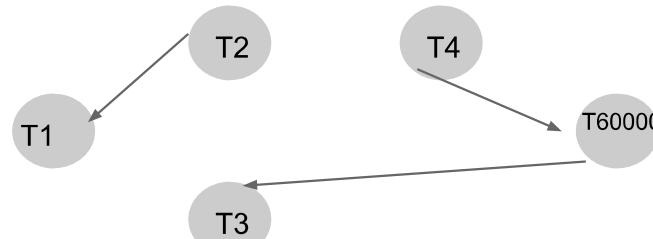
### Design#1 VisaDB

For each Transaction in Queue

- For relevant records
  - Use **2PL** to acquire/release locks
  - Read, Process, Write records
  - **WAL** Logs for updates
- Commit or Abort



### Example Waits-For Graph



### Example WAL Logs

- for 'T-Monthly-423'

WAL (@4:29 am day+1)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	COMMIT		

Update  
Records

Commit  
Record



# CS 145: Data Management and Data Systems

## Homework Section 3



## Topics Covered Today

- ▶ 

### Transactions

  - ▷ Brief review (key points to know to tackle the HW):
    - ACID properties
    - Logging (Write-ahead)
    - Conflicts/serializability/scheduling
    - Locking (2PL)
  - ▷ Some example problems along the way
  - ▷ Any questions you may have



## Review: ACID Properties

- ▶ **Atomicity:**
  - ▷ Key Idea: Every TXN has 2 outcomes:
    - Commit: All changes completed
    - Abort: No changes made
- ▶ **Consistency:**
  - ▷ Key Idea: After every TXN, DB satisfies constraints



## Review: ACID Properties

- ▶ **Isolation**
  - ▷ Key Idea: Concurrent TXNs are not affected by each other
  - ▷ The transactions proceed as if in isolation from other transactions. (\*)
- ▶ **Durability**
  - ▷ Key Idea: Results of committed transactions are permanent
  - ▷ Even in case of system failure!



## Example Question: ACID Properties

Imagine that we have a database of soccer (football) teams. Each team **must** consist of 11 players. Team A recently traded one of its players, George, for a new player, Alex. We wish to execute a transaction to update the database to reflect this change. In that case, we are forced to add Alex to team A and remove George from Team A in the same transaction.

Which ACID property requires us to do this?  
Why?



## Example Question: ACID Properties

Imagine that we have a database of soccer (football) teams. Each team **must** consist of 11 players. Team A recently traded one of its players, George, for a new player, Alex. We wish to execute a transaction to update the database to reflect this change. In that case, we are forced to add Alex to team A and remove George from Team A in the same transaction.

Which ACID property requires us to do this?  
Why?

**Answer:** Consistency. This is because before and after the transaction, we must satisfy the constraint that each team has 11 players.

## Review: Logging

### ► Logging setup:

T: R(A), W(A)



Log is a file (like any data table)  
1. A set of Pages updated in RAM  
2.Flushed as DB blocks on disk (sequential I/O)



**“Flushing** to disk” = writing to disk from main memory



## Review: Logging

- ▶ What to know about logging:
  - ▷ Only information on disk is durable
  - ▷ So, in the event of a crash, we need all information required to rollback changes to be stored **on disk**.



## Review: Logging

- ▶ Write-ahead logging (WAL):
  - ▷ Key Idea: Before updating data on disk, **record in the log first (flush the log record(s) to disk)**
  - ▷ Rule 1: Flush log record to disk **before** updating data on disk
  - ▷ Rule 2: Flush all log records and commit record to disk **before** committing TXN



## **Example Question: Logging and ACID**

Let's put the logging technique in a larger context - what ACID properties does logging guarantee?

- A. Atomicity
- B. Consistency
- C. Isolation
- D. Durability



## Example Question: Logging and ACID

Let's put the logging technique in a larger context - what ACID properties does logging guarantee?

- A. Atomicity ✓
- B. Consistency
- C. Isolation
- D. Durability ✓

Now we introduce a new protocol: instead of WAL, we write the data to disk, then commit and write corresponding log records. Which ACID property is lost?

**Atomicity!**



## Example Question: Logging

Let's imagine we have a DB table storing bank balances. In this table we have a record R containing John's bank balance. Initially  $R = 1000$ . John just deposited his paycheck of 100 dollars. We therefore execute a transaction to update record R and increase the balance by 100 dollars.

Our database supports write-ahead logging. List/describe each of the steps which will occur in this transaction, making reference to 4 possible areas of interest:

- ▶ Data in memory
- ▶ Log in memory
- ▶ Data on disk
- ▶ Log on disk



## Example Question: Logging

### Answer:

1. Read R from [Data on disk] into [Data in memory]
2. Modify R ( $R += 100$ ) in [Data in memory]
3. Create a log record ( $R: 1000 \rightarrow 1100$ ) in [Log in memory]
4. Create a commit record (to indicate end of transaction) in [Log in memory]
5. Write/flush [Log in memory] to [Log on disk] (both logs, update and commit)
6. Write/update record R from [Data in memory] to [Data on disk]



## Review: Logging Performance

- ▶ Logging is fast because it is sequential, while the records being updated may not be
- ▶ NOTE: Once the commit log is on disk, the TXN is committed
  - ▷ i.e. To do a TXN, only need to (sequentially) write to log, can update data later lazily as needed



## Example Question: Logging Performance

Assume that we are not using a log and have to deal with a million transactions per day where each transaction affects one record. How long does it take to update these records if our data is scattered randomly on disk?

Assume seek time is 10 ms/seek and the time it takes to write the update is 1 ms for simplicity.



## Example Question: Logging Performance

Assume that we are not using a log and have to deal with a million transactions per day where each transaction affects one record. How long does it take to update these records if our data is scattered randomly on disk?

Assume seek time is 10 ms/seek and the time it takes to write the update is 1 ms for simplicity.

**ANSWER:**

Time = 1 million seeks \* 11 msec = about 3 hours



## Example Question: Logging Performance

Now assume that we are using a log on disk to record all transactions. We still get 1 million transactions a day.

Assume sequential IO is 200 MB/sec and that the log ends up being 100 GB. How long did logging take?



## Example Question: Logging Performance

Now assume that we are using a log on disk to record all transactions. We still get 1 million transactions a day.

Assume sequential IO is 200 MB/sec and that the log ends up being 100 GB. How long did logging take?

ANSWER:

$$\text{Time} = 100 \text{ GB} / 200 \text{ MB/sec} = 500 \text{ seconds}$$

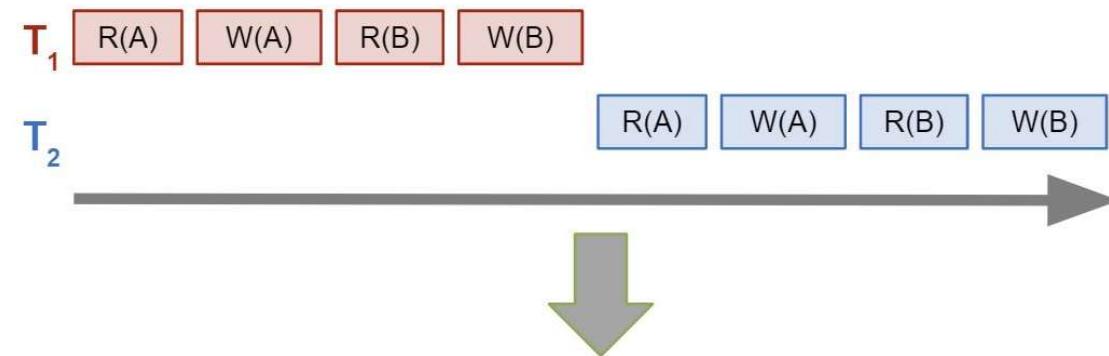


## Review: Scheduling/ Interleaving

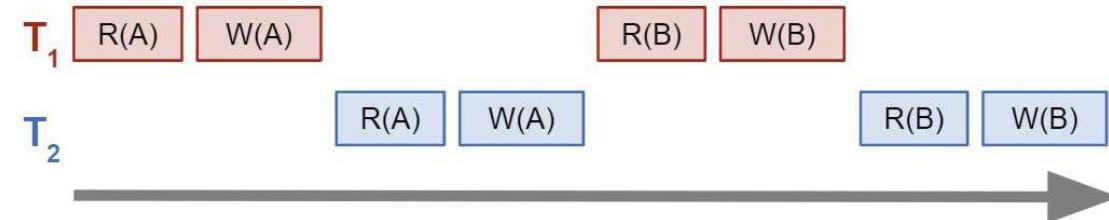
- ▶ **Scheduling/Interleaving:**
  - ▷ Change order of how parts of concurrent transactions are executed
- ▶ **Why?**
  - ▷ Improve performance (vs. sequential execution)
- ▶ **Problem:**
  - ▷ Need to maintain:
    - Isolation
    - Consistency

## Review: Scheduling/ Interleaving

### Serial Schedule



### Interleaved Schedule





## Review: Scheduling/ Interleaving

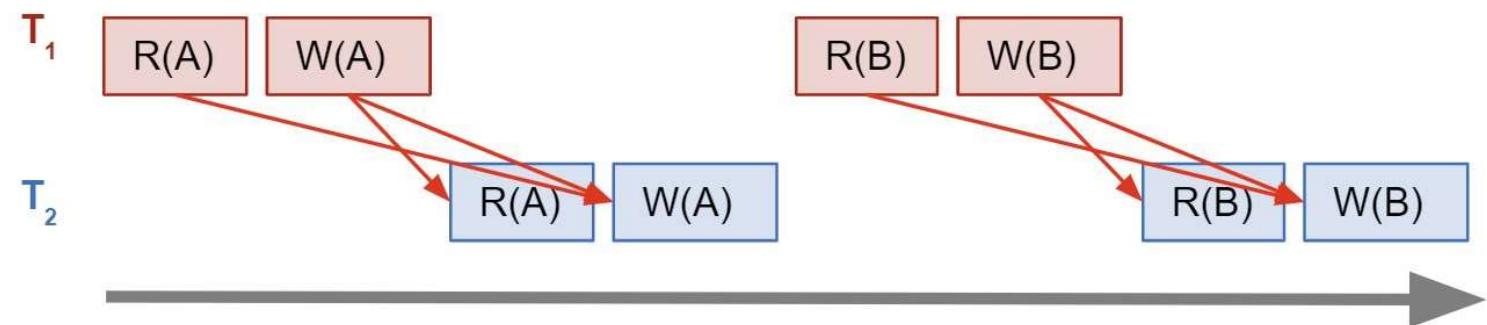
- ▶ How to ensure our interleaved schedule preserves **Isolation + Consistency?**
- ▶ Make sure our schedule is **equivalent to a serial execution of the transactions => same result**
  - ▷ Why? Serial execution respects:
    - Isolation
    - Consistency
- ▶ We want a **serializable schedule**



## Review: Conflicts

- ▶ How do we know if a schedule is **Serializable**?
- ▶ **Idea:** Look at **conflicts**
- ▶ **Conflict** = A pair of actions from two concurrent TXNs which 1) are on the same record and 2) at least one is a **write**
  - ▷ RW
  - ▷ WR
  - ▷ WW

## Review: Conflicts



All “conflicts”!

- ▶ NOTE: Conflicts are unavoidable, and having conflicts doesn't imply that the transactions can't be scheduled. We just need to be careful how we handle them.



## Review: Conflict Serializability

- ▶ How do we make use of **conflicts** to determine if a schedule is **Serializable**?
- ▶ Definition: A schedule is **conflict serializable** if there exists a sequential schedule such that:
  - ▷ For every **conflict**:
    - The **ordering** of the conflicting actions is the same
- ▶ If a schedule is **conflict serializable**, it is **Serializable**

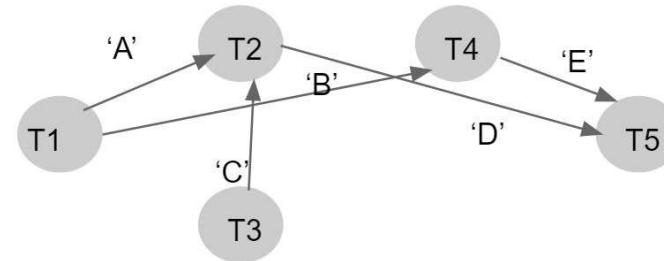


## Review: Conflict Graph

- ▶ One easy way to find if a schedule w/ many TXNs is **conflict serializable**: **Build a conflict graph:**
  - ▷ Each **node** is a TXN
  - ▷ There is an **edge** from  $T_i \rightarrow T_j$  if any actions in  $T_i$  precede and conflict with any actions in  $T_j$
- ▶ If the conflict graph is **acyclic**, the schedule is **conflict serializable**

## Review: Conflict Graph

T1	w1(A)		w1(B)						
T2		r2(A)			r2(C)		w2(D)		
T3				w3(C)					
T4						r4(B)		w4(E)	
T5								r5(D)	w5(E)



Acyclic  
⇒ Conflict serializable!  
⇒ Serializable



## Review: Conflict Serializability

- ▶ To sum up: Is a schedule  $S$  serializable?
- ▶ **Procedure:**
  - ▷ Find all **conflicts** in  $S$
  - ▷ Make **conflict graph**
    - For every conflict from  $T_i$  to  $T_j$ , add a directed edge from node  $T_i$  to  $T_j$
  - ▷ Check if graph is **acyclic (serializable)**
  - ▷ If so, **TopoSort** to get an equivalent serial schedule

## Example Question: Conflict Serializability

Is the following schedule conflict serializable?

T1	R(A)		R(B)		W(B)	
T2		W(B)		R(A)		R(A)
Timestamp	1	2	3	4	5	6

## Example Question: Conflict Serializability

Is the following schedule conflict serializable?

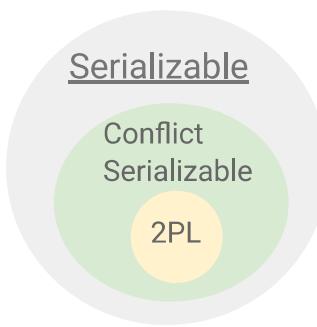
T1	R(A)		R(B)		W(B)	
T2		W(B)		R(A)		R(A)
Timestamp	1	2	3	4	5	6

**Yes.** Why? The only conflicts are between W(B)[2] and R(B)[3] and W(B)[2] and W(B)[5].

Both of these start at T2 and go to T1. So the conflict graph will only have an arrow from T2 to T1 (no cycle). Therefore the schedule is serializable.

## Review: 2-Phase Locking

- ▶ Before **writing**, a TXN gets an **exclusive (X) lock**
  - ▷ No other TXN can read or write
- ▶ Before reading, a TXN gets a **shared (S) lock**
  - ▷ No other TXN can write, but can read
- ▶ TXN releases all locks at once after commit/abort
- ▶ Any schedule **produced** by 2PL will be conflict serializable
- ▶ **Important distinction:**
  - ▷ If a schedule “breaks the rules” of 2PL described above, it cannot be **executed in exact order** by a 2PL DB
  - ▷ However, such a schedule could still be **executed** by a 2PL DB! It would just be executed in a modified order that satisfies 2PL
  - ▷ Just because a schedule cannot be **executed in exact order** by a 2PL DB, it doesn’t mean it can’t be executed (it is not **deadlocked**)



## Example Question: 2PL

Can the following schedule be produced by S2PL?

T1	R(A)		R(B)		W(B)	
T2		W(B)		R(A)		R(A)
Timestamp	1	2	3	4	5	6

## Example Question: 2PL

Can the following schedule be produced by S2PL?

T1	R(A)		R(B)		W(B)	
T2		W(B)		R(A)		R(A)
Timestamp	1	2	3	4	5	6

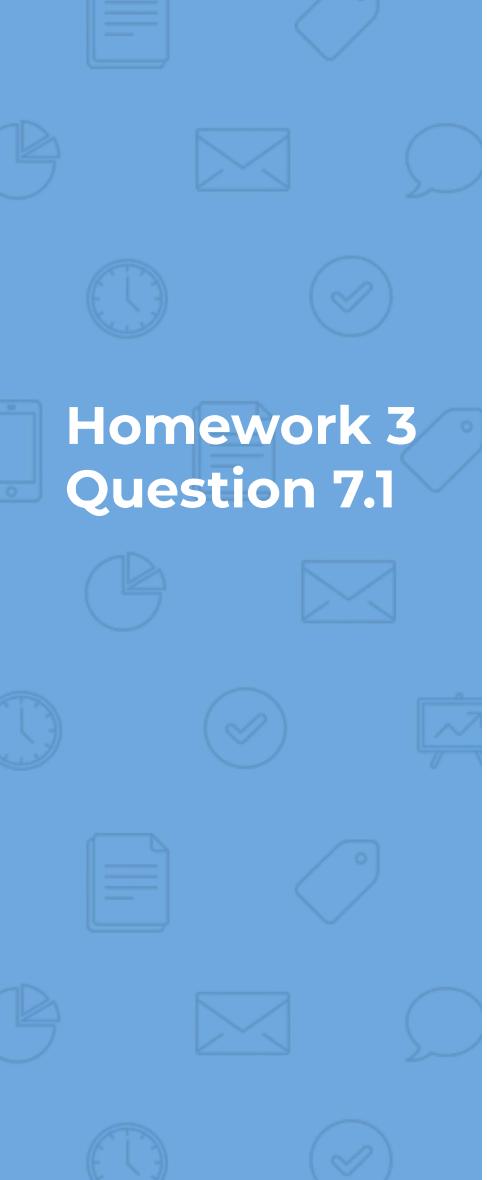
**No.** Why? At time 2, T2 acquires an exclusive lock on B. Then, at time 3, T1 will fail to acquire a shared lock on B to read it.

## Homework 3 Question 7.1

Is the following schedule:

1. Conflict serializable?
2. Producible by strict 2PL?

T1	R(A)	W(A)		R(B)	
T2			W(A)		W(B)
timestamp	0	1	2	3	4



## Homework 3 Question 7.1

Is the following schedule:

1. Conflict serializable?
2. Producible by strict 2PL?

T1	R(A)	W(A)		R(B)	
T2			W(A)		W(B)
timestamp	0	1	2	3	4

1. **Yes.** It is equivalent to the serial schedule: T1, then T2.
  - a. **Why is it not equivalent to: T2, then T1?**
2. **No.** T1 acquires S/X locks on resource A, but has not yet committed (still needs S on B at t=3), so T2 cannot obtain an X lock on resource A at t=2.

# Questions?



# Scale for Big Schemas

For 10s-1000s of Tables, Columns, and Flows

# Example 1: NCAA Basketball -- schema for 1 table in BigQuery

The screenshot shows the BigQuery UI interface. On the left, there is a tree view of datasets and tables. A table named 'mbb\_games\_sr' is selected and highlighted with a blue background. To the right of the tree view is a large, empty gray area.

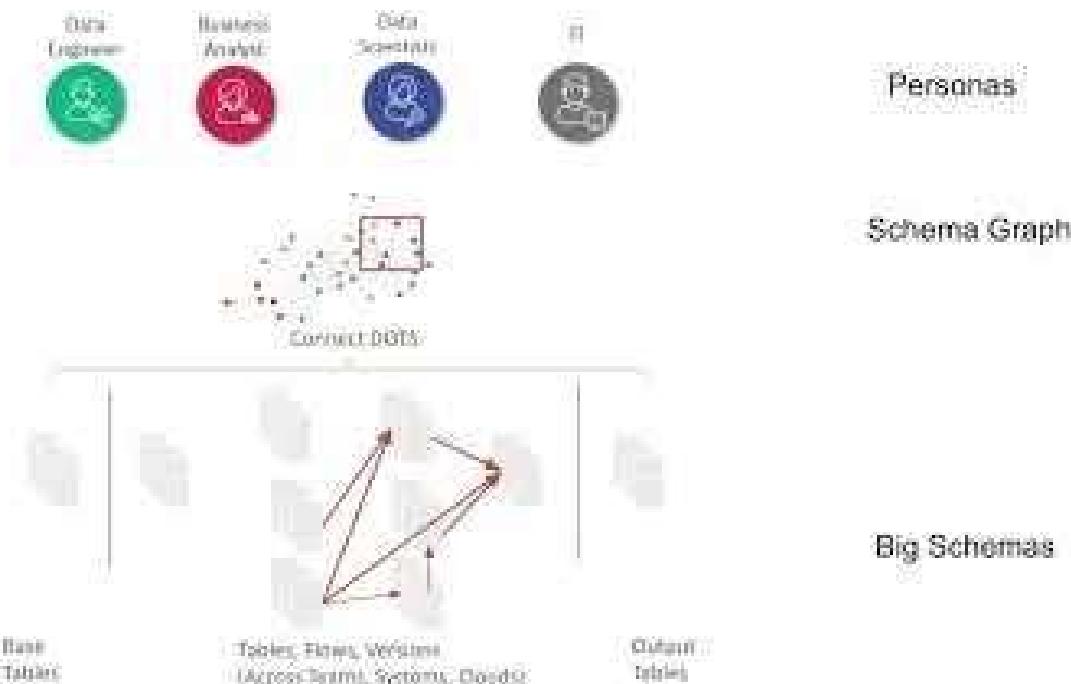
Field name	Type	Mode	Policy Tags	Description
game_id	STRING	NULLABLE		[Game data] Unique identifier for the game
season	INTEGER	NULLABLE		[Game data] Season the game was played in
status	STRING	NULLABLE		[Game data] Indicates the last state of Sportradar's game file
coverage	STRING	NULLABLE		[Game data] Type of coverage provided by Sportradar
neutral_site	BOOLEAN	NULLABLE		[Game data] Indicator of whether the game was played on a neutral court
scheduled_date	DATE	NULLABLE		[Game data] Date the game was played
gametime	TIMESTAMP	NULLABLE		[Game data] Date and time the game was played
conference_game	BOOLEAN	NULLABLE		[Game data] Indicator of whether the two teams were in the same conference at the time the game was played
tournament	STRING	NULLABLE		[Game data] Whether the game was played in a post-season tournament
tournament_type	STRING	NULLABLE		[Game data] Type of post-season tournament a game was in played
tournament_round	STRING	NULLABLE		[Game data] Tournament round
tournament_game_no	STRING	NULLABLE		[Game data] Tournament game number
attendance	INTEGER	NULLABLE		[Game data] Attendance of the game
lead_changes	INTEGER	NULLABLE		[Game stats] Number of lead changes in the game
times_tied	INTEGER	NULLABLE		[Game stats] Number of ties in the game
periods	INTEGER	NULLABLE		[Game stats] Number of periods the game
possession_arrow	STRING	NULLABLE		[Game stats] The unique identifier of the team that would receive the ball the next time a jump ball is called, see <a href="https://en.wikipedia.org/wiki/Jump_ball">https://en.wikipedia.org/wiki/Jump_ball</a> for more information
venue_id	STRING	NULLABLE		[Game data] Unique identifier for the venue where the game was played
venue_city	STRING	NULLABLE		[Game data] City where the game was played

The screenshot shows the BigQuery UI interface. On the left, there is a tree view of datasets and tables. A table named 'mbb\_games\_sr' is selected and highlighted with a blue background. To the right of the tree view is a large, empty gray area.

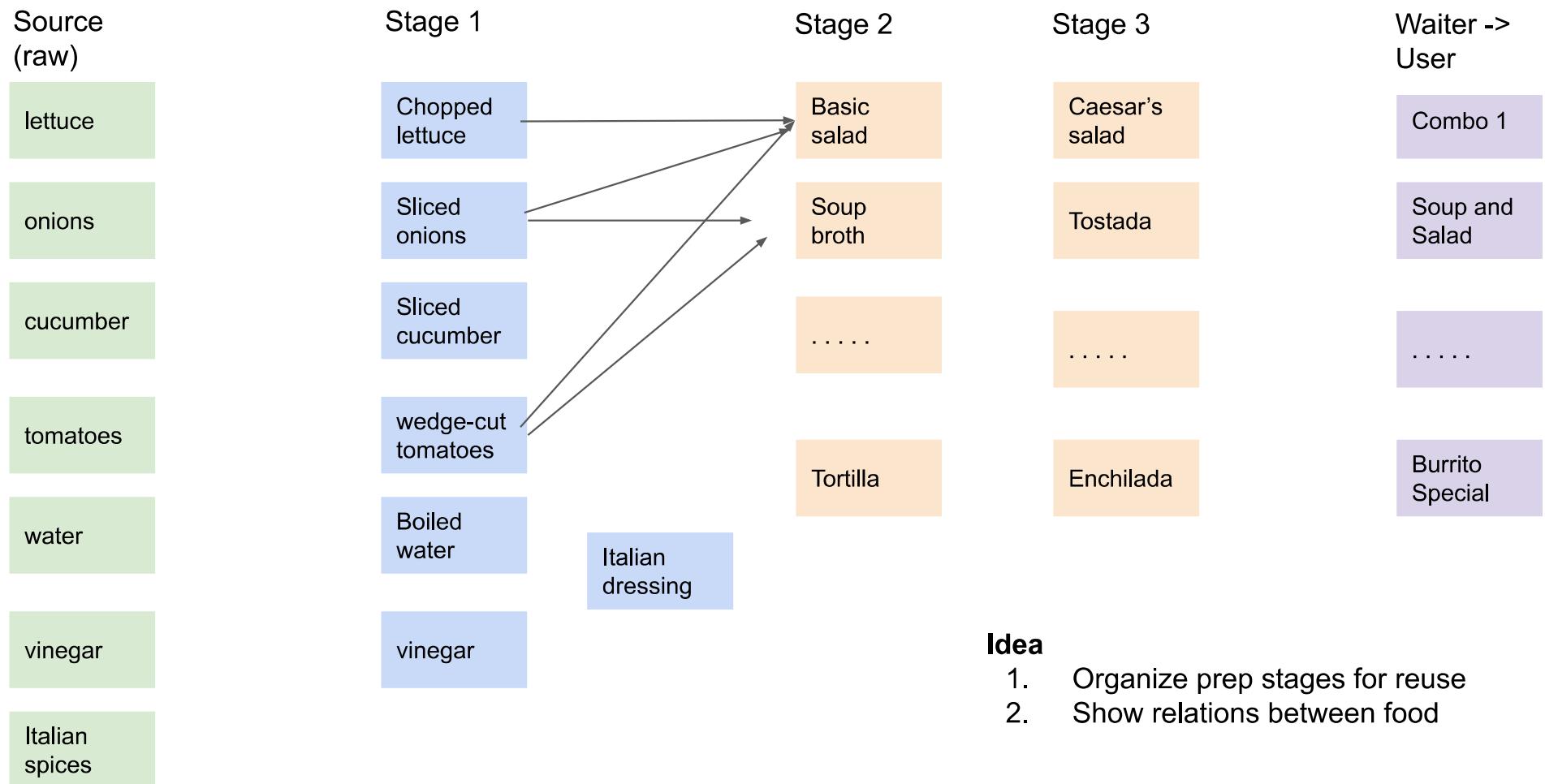
Field name	Type	Mode	Policy Tags	Description
game_id	STRING	NULLABLE		[Game data] Unique identifier for the game
season	INTEGER	NULLABLE		[Game data] Season the game was played in
status	STRING	NULLABLE		[Game data] Indicates the last state of Sportradar's game file
coverage	STRING	NULLABLE		[Game data] Type of coverage provided by Sportradar
neutral_site	BOOLEAN	NULLABLE		[Game data] Indicator of whether the game was played on a neutral court
scheduled_date	DATE	NULLABLE		[Game data] Date the game was played
gametime	TIMESTAMP	NULLABLE		[Game data] Date and time the game was played
conference_game	BOOLEAN	NULLABLE		[Game data] Indicator of whether the two teams were in the same conference at the time the game was played
tournament	STRING	NULLABLE		[Game data] Whether the game was played in a post-season tournament
tournament_type	STRING	NULLABLE		[Game data] Type of post-season tournament a game was in played
tournament_round	STRING	NULLABLE		[Game data] Tournament round
tournament_game_no	STRING	NULLABLE		[Game data] Tournament game number
attendance	INTEGER	NULLABLE		[Game data] Attendance of the game
lead_changes	INTEGER	NULLABLE		[Game stats] Number of lead changes in the game
times_tied	INTEGER	NULLABLE		[Game stats] Number of ties in the game
periods	INTEGER	NULLABLE		[Game stats] Number of periods the game
possession_arrow	STRING	NULLABLE		[Game stats] The unique identifier of the team that would receive the ball the next time a jump ball is called, see <a href="https://en.wikipedia.org/wiki/Jump_ball">https://en.wikipedia.org/wiki/Jump_ball</a> for more information
venue_id	STRING	NULLABLE		[Game data] Unique identifier for the venue where the game was played
venue_city	STRING	NULLABLE		[Game data] City where the game was played

1. How to find relationships between columns?
2. How about 10x-100x tables, Columns?

# Problems



# Intuition: Cooking Prep



# Example2: Connected data flows

## Production ML

An on-call engineer's biggest nightmare

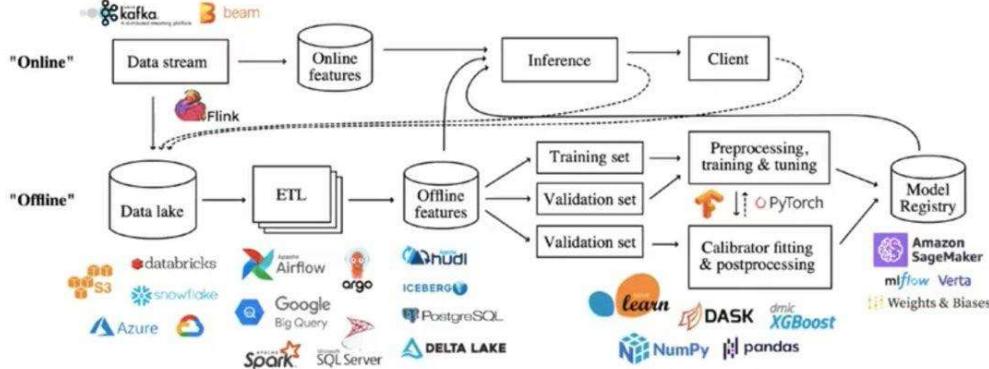
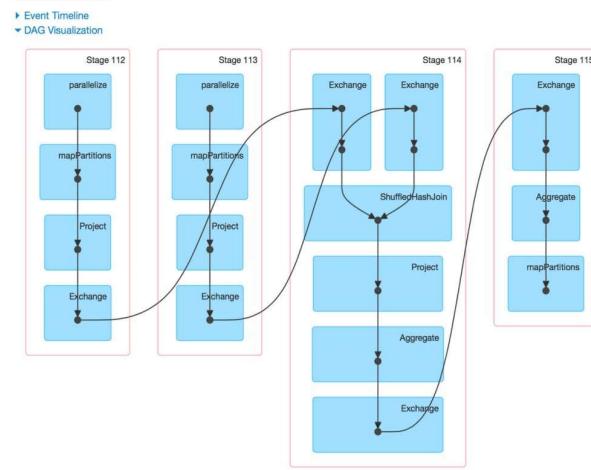


Figure 1: High-level architecture of a generic end-to-end machine learning pipeline. Logos represent a sample of tools used to construct components of the pipeline, illustrating heterogeneity in the tool stack. Shankar et al. 2021

### Details for Job 8

Status: SUCCEEDED  
Completed Stages: 4  
Event Timeline  
DAG Visualization



Note: 115 Stages in sample  
Spark pipeline !!!

Basics: Which “ids”, which versions, which tools?

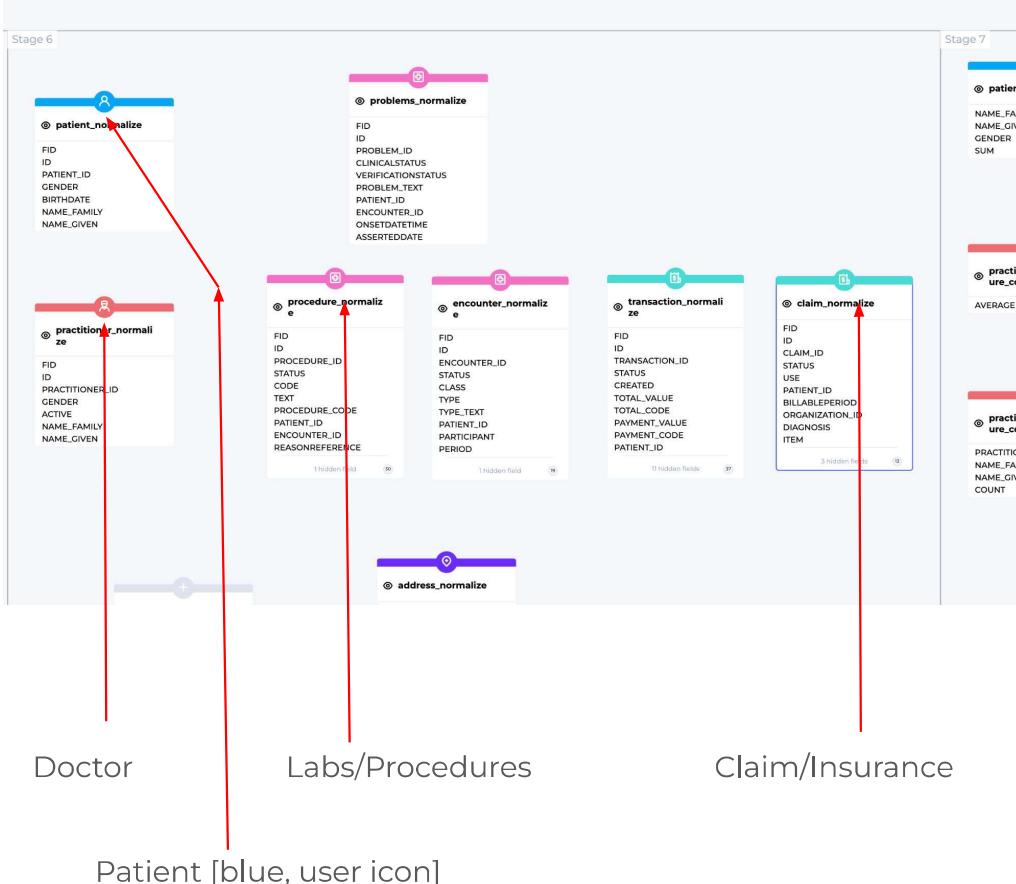
# Problems

1. How to connect schemas across
  - a. 10s-1000s of Tables, Columns, Relationships, and Flows?
2. How teams collaborate on Big Schemas?
  - a. Different subsets useful to App team, Data Analysts, Data Eng...
3. When schemas change?

# Problems



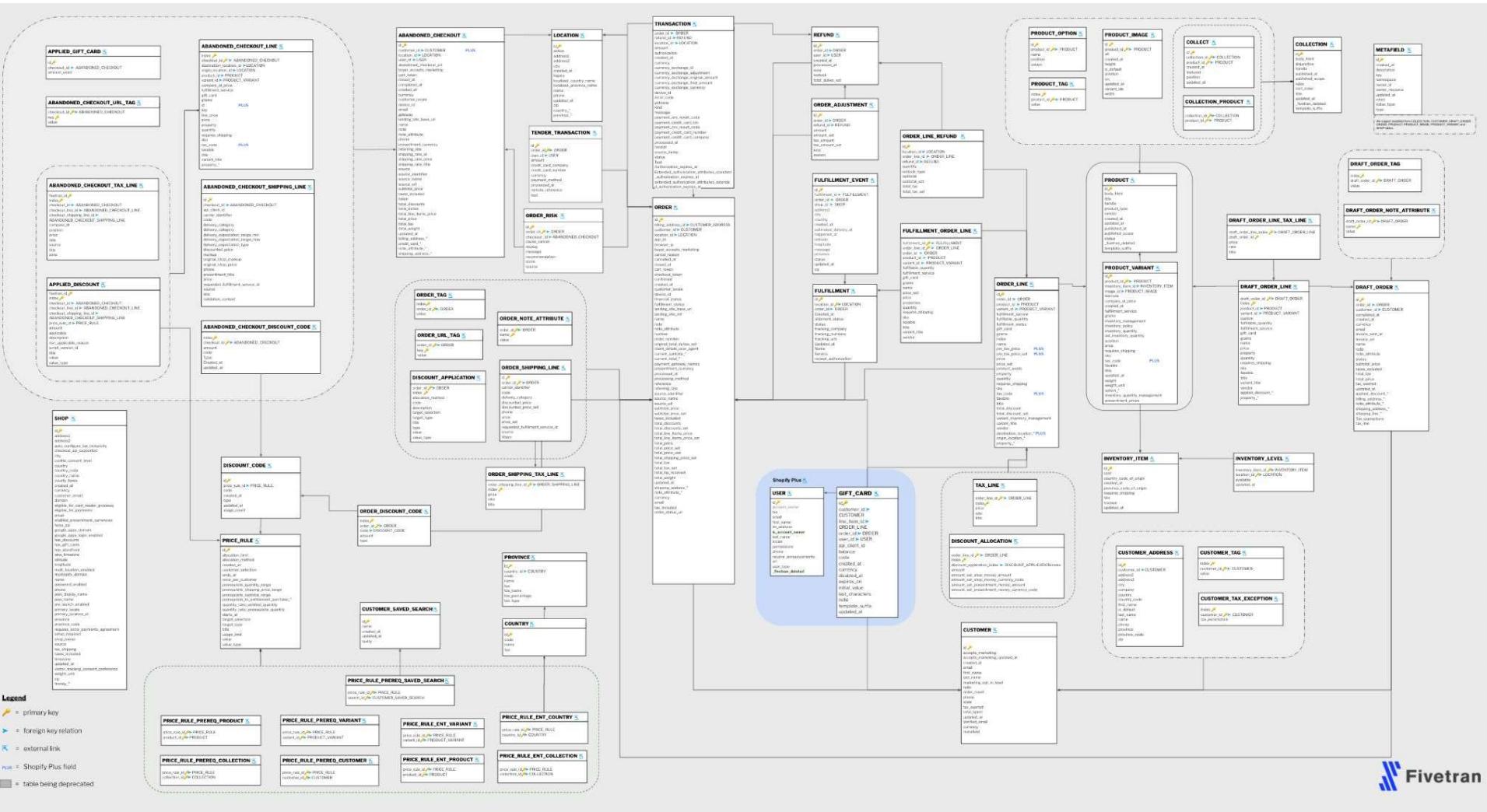
# Example: Health data app



## Scenario

1. Patient/Doctor PII data → Cost and Quality
2. Multiple teams producing/consuming
  - a. Biz analysts, app eng, data eng, product
  - b. Fast-paced app changes in prod/staging
3. Transforms
  - a. ~600 tables –
    - i. 20 input tables → 100s of Views/CTEs for downstream applications
  - b. Versions in Jan/Feb – Edit/add ~12 models

# Example2: Shopify's simplified ER (Entity Relation Diagram)





# Lecture: Design Theory – Part 1

# Conceptual Design

For a "mega" table

- Search for "bad" dependencies
- If any, keep decomposing the table into sub-tables until no more bad dependencies
- When done, the database schema is normalized

Note: there are several "good" (normal) forms...

## Example Enrollment table - “v0”

~375  
cs145  
students

<b>SID</b>	<b>Class</b>	<b>Room</b>	<b>Time</b>	<b>Lat</b>	<b>Lng</b>
4749732	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
2720942	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
4823984	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
4287594	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2984994	cs 145	Nvidia Aud	T/R 4:30-6	...	...
8472374	cs 145	Nvidia Aud	T/R 4:30-6	...	...
4723663	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2478239	cs 145	Nvidia Aud	T/R 4:30-6	...	...
4763268	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2364532	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2364573	cs 145	Nvidia Aud	T/R 4:30-6	...	...
3476382	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2347623	cs 145	Nvidia Aud	T/R 4:30-6	...	...
...	...	...	...	...	...
~300 cs245 students	2364579	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N
	3476343	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N
	2322232	cs 245	Nvidia Aud	T/R 3-4:30	122.1742° W



Problems  
Repeats?  
Room/time change?  
Deletes?

Properties  
Class → Room/time  
Room → Lat, Lng  
(more compact)

## Example Enrollment table - “v1”

375  
cs145  
students

<b>SID</b>	<b>Class</b>
4749732	cs 145
2720942	cs 145
4823984	cs 145
4287594	cs 145
2984994	cs 145
8472374	cs 145
4723663	cs 145
2478239	cs 145
4763268	cs 145
2364532	cs 145
2364573	cs 145
3476382	cs 145
2347623	cs 145
...	...
2364579	cs 245
3476343	cs 245
2322232	cs 245

300  
cs245  
students

<b>Class</b>	<b>Room</b>	<b>Time</b>
cs 145	Nvidia Aud	T/R 4:30-6
cs 245	Nvidia Aud	T/R 3-4:30
cs 246	Nvidia Aud	M/W 3-4:30

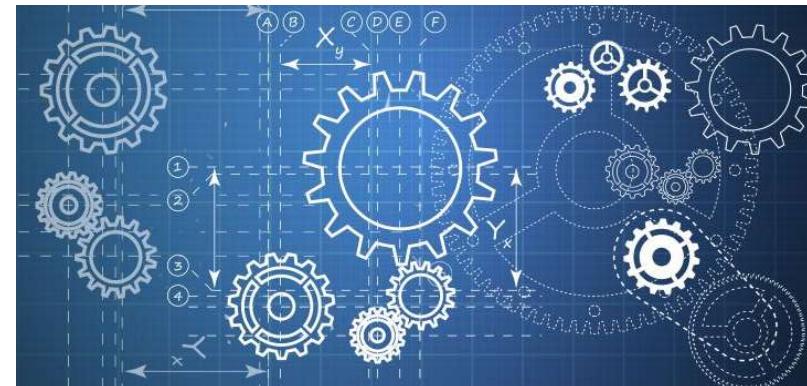
<b>Room</b>	<b>Lat</b>	<b>Lng</b>
Nvidia Aud	37.4277° N	122.1742° W





# Design Theory

- Design theory: how to represent your data to avoid ***anomalies***.
  - Note: different than concurrency anomalies.
- Simple algorithms for “best practices”





# Data Anomalies & Constraints

# Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

If every course is in only one room, contains **redundant** information!

# Constraints Prevent (some) Anomalies in the Data

A poorly designed DB can have *anomalies*:

Student	Course	Room
Mary	CS145	B01
Joe	CS145	C12
Sam	CS145	B01
..	..	..

If we update the room number for one tuple, we get inconsistent data = an **update anomaly**

# Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

Student	Course	Room
..	..	..

If everyone drops the class, we lose what room the class is in! = a **delete anomaly**

# Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:



Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Similarly, we can't reserve a room without students = an **insert anomaly**

# Constraints Prevent (some) Anomalies in the Data

Student	Course
Mary	CS145
Joe	CS145
Sam	CS145
..	..

Course	Room
CS145	B01
CS229	C12

Is this form better?

- Redundancy?
- Update anomaly?
- Delete anomaly?
- Insert anomaly?

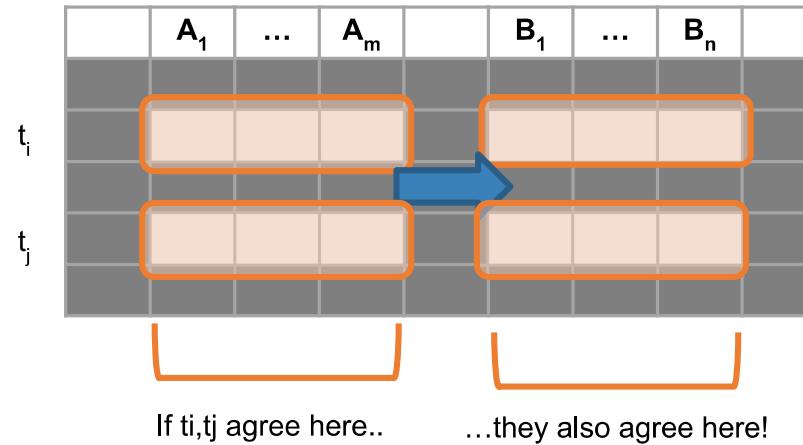
What are “good” decompositions?



# Functional Dependencies



# A Picture Of FDs



## Definition:

Given attribute sets  $A = \{A_1, \dots, A_m\}$  and  $B = \{B_1, \dots, B_n\}$  in  $R$ ,

The ***functional dependency***  $A \rightarrow B$  on  $R$  holds if for **any**  $t_i, t_j$  in  $R$ :

**if**  $t_i[A_1] = t_j[A_1] \text{ AND } t_i[A_2] = t_j[A_2] \text{ AND } \dots \text{ AND } t_i[A_m] = t_j[A_m]$

**then**  $t_i[B_1] = t_j[B_1] \text{ AND } t_i[B_2] = t_j[B_2] \text{ AND } \dots \text{ AND } t_i[B_n] = t_j[B_n]$

$A \rightarrow B$  means that  
“whenever two tuples agree on  $A$  then they agree on  $B$ .”



# FDs for Relational Schema Design

High-level idea: why do we care about FDs?

1. Start with some relational *schema*
2. Find *functional dependencies (FDs)*
3. Use these to *design a better schema*  
One which minimizes the possibility of anomalies

# Functional Dependencies as Constraints

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Note: The FD {Course}  $\rightarrow$  {Room} **holds on this table instance**

However, cannot prove that the FD {Course}  $\rightarrow$  {Room} holds on all instances. That is, FDs are for an instance and not for **schema**

# Functional Dependencies as Constraints

Note that:

- You can check if an FD is **violated** by examining a single instance;
- However, you **cannot prove** that an FD is part of the schema by examining a single instance.
  - *This would require checking every valid instance*

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

# More Examples

An FD is a constraint which holds, or does not hold on an instance:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

# More Examples

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876 ←	Salesrep
E1111	Smith	9876 ←	Salesrep
E9999	Mary	1234	Lawyer

$\{\text{Position}\} \rightarrow \{\text{Phone}\}$

# More Examples

EmplID	Name	Phone	Position
E0045	Smith	1234 →	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234 →	Lawyer

but *not* {Phone} → {Position}

# Example (mega) Enrollment table - “v0”

~375  
cs145  
students

<b>SID</b>	<b>Class</b>	<b>Room</b>	<b>Time</b>	<b>Lat</b>	<b>Lng</b>
4749732	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
2720942	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
4823984	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
4287594	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2984994	cs 145	Nvidia Aud	T/R 4:30-6	...	...
8472374	cs 145	Nvidia Aud	T/R 4:30-6	...	...
4723663	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2478239	cs 145	Nvidia Aud	T/R 4:30-6	...	...
4763268	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2364532	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2364573	cs 145	Nvidia Aud	T/R 4:30-6	...	...
3476382	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2347623	cs 145	Nvidia Aud	T/R 4:30-6	...	...
...	...	...	...	...	...
~300 cs245 students	2364579	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N
	3476343	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N
	2322232	cs 245	Nvidia Aud	T/R 3-4:30	122.1742° W

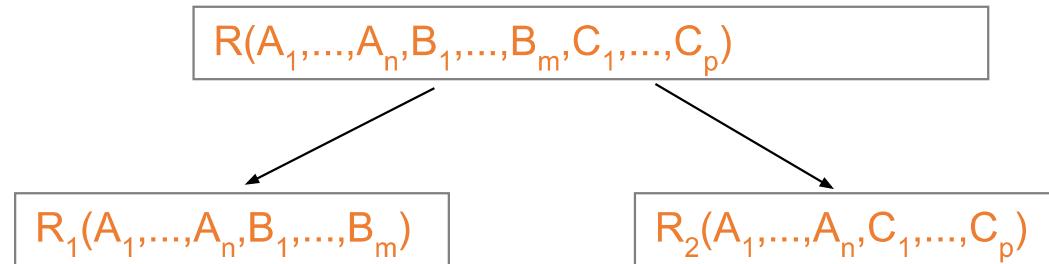


Problems  
Repeats?  
Room/time change?  
Deletes?

FDs  
Class → Room, Time  
Room → Lat, Lng  
(more compact)



# Table Decomposition



$R_1$  = the *projection* of  $R$  on  $A_1, \dots, A_n, B_1, \dots, B_m$

$R_2$  = the *projection* of  $R$  on  $A_1, \dots, A_n, C_1, \dots, C_p$

# Conceptual Design

For a “mega” table



- Search for “bad” dependencies
- If any, *keep decomposing the table into sub-tables* until no more bad dependencies
- When done, the database schema is normalized

Note: there are several “good” (normal) forms...

# Finding Functional Dependencies

Example:

**Products**

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

**Provided FDs:**

1.  $\{Name\} \rightarrow \{Color\}$
2.  $\{Category\} \rightarrow \{Department\}$
3.  $\{Color, Category\} \rightarrow \{Price\}$

Given the provided FDs,  
 $\{Name, Category\} \rightarrow \{Price\}$  must also hold on **any instance...**

Which / how many other FDs do?!?



# Inferring FDs

Given a set of FDs,  $F = \{f_1, \dots, f_n\}$ , does an FD  $g$  hold?

(Similar idea to Logic inferencing. E.g, A implies B, B implies C, so A implies C)

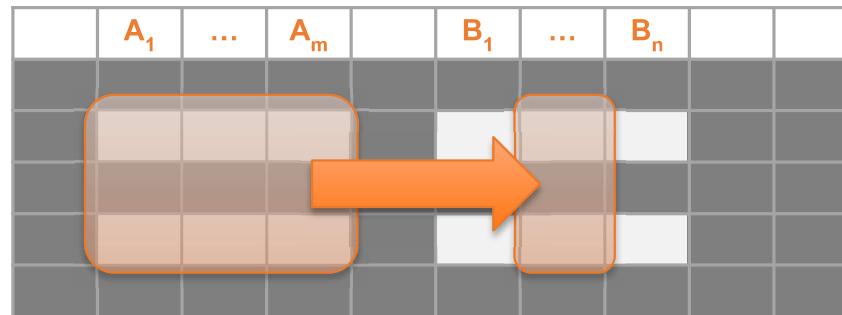
## Armstrong's Rules.

1. Split/Combine
2. Reduction
3. Transitivity





# 1. Split/Combine

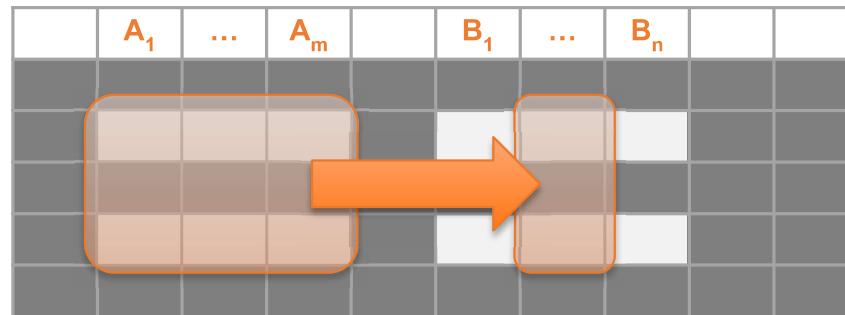


$A_1, \dots, A_m \rightarrow B_1, \dots, B_n \quad \dots$  is equivalent to the following  $n$  FDs...

$A_1, \dots, A_m \rightarrow B_i \text{ for } i=1, \dots, n$



# 1. Split/Combine

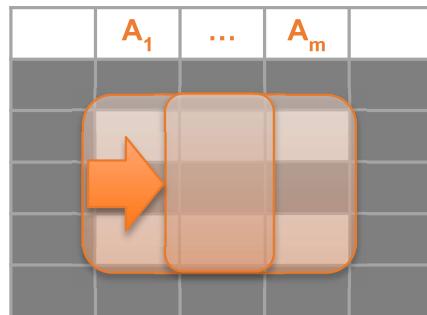


**And vice-versa,**  $A_1, \dots, A_m \rightarrow B_i$  for  $i=1, \dots, n$  ... is equivalent to ...

$$A_1, \dots, A_m \rightarrow B_1, \dots, B_n$$



## 2. Reduction/Trivial

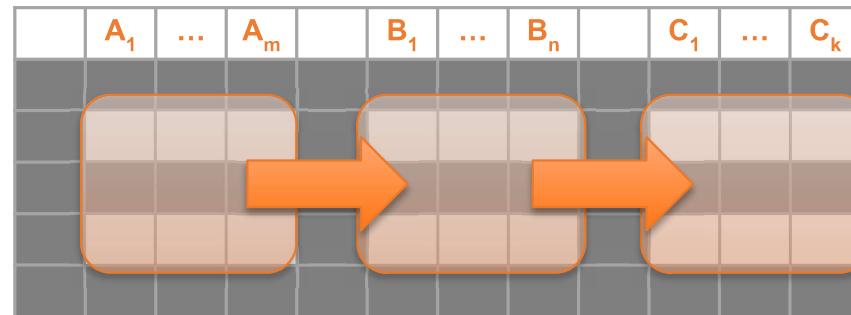


If B is subset of A ( $=A_1, \dots, A_m$ ) for any  $j=1, \dots, m$

$$A \rightarrow B$$



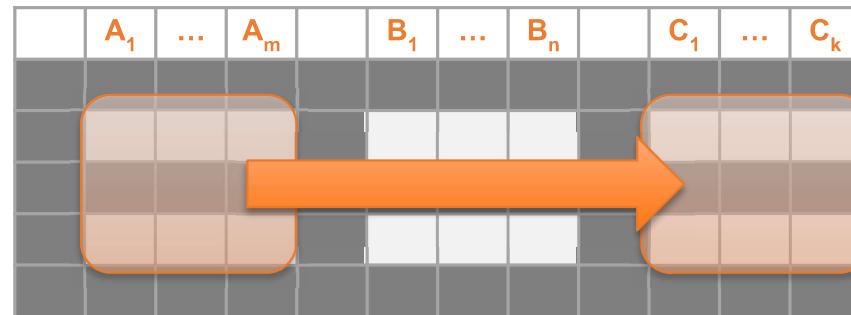
### 3. Transitive Closure



$A_1, \dots, A_m \rightarrow B_1, \dots, B_n$  and  
 $B_1, \dots, B_n \rightarrow C_1, \dots, C_k$



### 3. Transitive Closure



$A_1, \dots, A_m \rightarrow B_1, \dots, B_n$  and  
 $B_1, \dots, B_n \rightarrow C_1, \dots, C_k$

implies

$A_1, \dots, A_m \rightarrow C_1, \dots, C_k$

# Finding Functional Dependencies

Example:

**Products**

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

**Provided FDs:**

1.  $\{Name\} \rightarrow \{Color\}$
2.  $\{Category\} \rightarrow \{Department\}$
3.  $\{Color, Category\} \rightarrow \{Price\}$

Which / how many other FDs hold?

# Finding Functional Dependencies

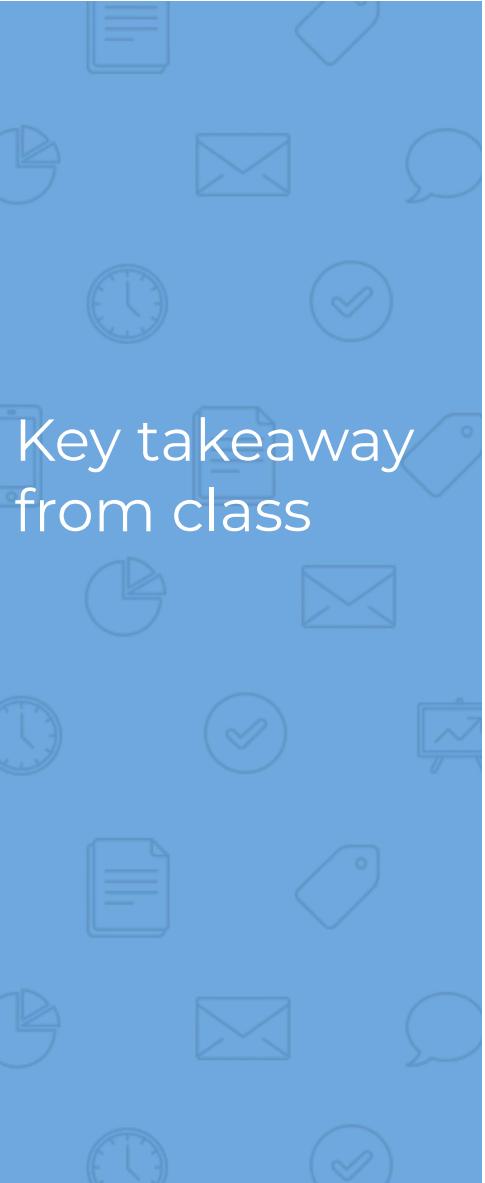
## Example:

Inferred FD	Rule used	Provided FDs:
4. {Name, Category} $\rightarrow$ {Name}	Trivial	1. {Name} $\rightarrow$ {Color}
5. {Name, Category} $\rightarrow$ {Color}	Transitive (4 $\rightarrow$ 1)	2. {Category} $\rightarrow$ {Dept.}
6. {Name, Category} $\rightarrow$ {Category}	Trivial	3. {Color, Category} $\rightarrow$ {Price}
7. {Name, Category} $\rightarrow$ {Color, Category}	Split/Combine (5 + 6)	
8. {Name, Category} $\rightarrow$ {Price}	Transitive (7 $\rightarrow$ 3)	

What's an algorithmic way to do this?



# Final Review



Key takeaway  
from class

## ⇒ How to apply CS concepts at scale to solve big problems?

- How do we scale for
  - “Big systems” by 10-100x,
  - “Big queries” by 10-100x,
  - “Big writes” (transactions) by 10-100x
- How to work with real data sets?
  - “Big schemas” for 100s-1000s of Tables, Flows
  - 3 projects

**CONGRATULATIONS!**



# Case Study 1

**Amazon's Product  
Recommendations**

[see Lecture 9]

# Product Search & CoOccur

## Billion products

User searches for “coffee machine”



Nespresso Vertuo Coffee and Espresso Machine Bundle with Aeroccino Milk Frother by Breville, Red

by Breville

★★★★★ 980 customer reviews

| 259 answered questions

Amazon's Choice for "nespresso machine red"

List Price: \$249.95

Price: \$189.96 | FREE One-Day

You Save: \$59.99 (24%)

Your cost could be \$179.96. Eligible customers get a \$10 bonus when reloading \$100.

Free Amazon product support included

Style Name: Nespresso by Breville

Color: Red



## Product recommendations

Customers who viewed this item also viewed these products



Dualit Food XL1500 Processor  
\$560

Add to cart



Kenwood kMix Manual Espresso Machine  
★★★★★ 250  
\$250

Select options



Weber One Touch Gold Premium Charcoal Grill-57cm  
\$225

Add to cart



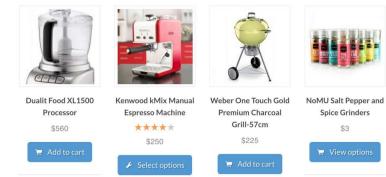
NoMU Salt Pepper and Spice Grinders  
\$3

View options

# Product Search & CoOccur

## Counting popular product-pairs

Customers who viewed this item also viewed these products



Story: Amazon/Walmart/Alibaba (AWA) want to sell products

1. Problem1: Amazon wants fast user searches for product
  2. Problem2: Amazon shows 'related products' for all products
    - Using collaborative filtering ('wisdom of crowds') from historical website logs.
    - Each time a user views a set of products, those products are related (co-occur)
- ⇒ Goal: compute product pairs and their co-occur count, across all users

Data input:

- AMZN has **1 billion products**. Each product record is ~1MB (descriptions, images, etc.).
- AMZN has **10 billion UserViews** each week, from 1 billion users. Stored in **UserViews**, each row has <userID, productID, viewID, viewTime>.

# Pre-cs145

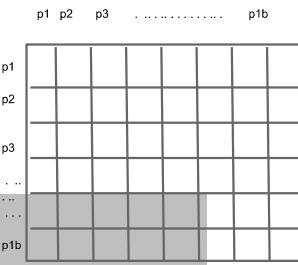
## Compute in RAM

(Engg approximations)

### Counting product views

Input size (4 bytes for user, 4 bytes for productid)	$\sim 1\text{Bil} * [4 + 4] = 8 \text{ GB}$
Output size (4 bytes for productid, 4 bytes for count)	$\sim 1\text{Bil} * [4 + 4] = 8 \text{ GB}$

Trivial



### Counting product pair views

#### Plan #1: $\sim P * P/2$ matrix for counters in RAM (4 bytes for count)

- RAM size =  $1 \text{ Billion} * 1\text{Billion}/2 * 4 = 2 \text{ Million TBs}$ 
  - $// (count(a, b) = count(b, a))$
- Trivial? If you have **~\$6 Billion** (RAM at  $\sim \$100/32\text{GB}$  RAM)

#### Plan #1 (on disk): Let OS page into memory as needed

- Worst case #1 > 1 million years on HDD  
(if you seek to update each counter)

**⇒ Need to do some design work, with cs145**

## Systems Design Example:

### Product CoOccur

#### Pre-design

## Systems Design Example:

### Product CoOccur

#### Plan #2

[See Notebook for impact of B values]

	Size	Why?
ProductId	4 bytes	1 Billion products $\Rightarrow$ Need at least 30 bits ( $2^{30} \approx 1$ Billion) to represent each product uniquely. So use 4 bytes.
UserID	4 bytes	"
ViewID	8 bytes	10 Billion product views.
Products	$P(\text{Product}) = 15.625\text{M pages}$	1 Billion products of 1 MB each. Size = 1 PB = 15.625M pages NumSearchKeys(Product)=1 Billion
UserViews	$P(\text{UserViews}) = 3750 \text{ pages}$	Each record is <userId, productId, viewID, viewType>. Assume: 8 bytes for viewType. So that's 24 bytes per record. 10 Billion * [4+4+8+8] bytes = 240 GBs = 3750 Pages.
ProductPairs for all userViews (We'll see how to use this in 3 slides)	5625 pages	Given: blowup-factor = 4.5, and 10 Billion UserViews. Each record is <productId, productId>. At 8 bytes per record, 45 Billion * 8 bytes = 360GB = 5625 Pages.
CoOccur (for top 3 Billion)	$P(\text{CoOccur}) = 563 \text{ pages}$	The output should be <productId, productId, count> for the co-occur counts. That is, 12 bytes per record (4 + 4 + 4 for the two productIDs and 4 bytes for count). For three billion product pairs, you need 3 billion * 12 bytes = 36 GBs = 563 pages. NumSearchKeys(CoOccur) = 1 Billion.

$$P(\text{Temp}) = 5625$$

$$P(\text{CoOccur}) = 563$$



Steps	Cost (IOs)	Why?
SMJ(UserViews, UserViews) on UserID	$3*3750 + 5625$	$3*P(\text{UserViews}) + P(\text{Temp})$
Sort, GroupBY -> CoOccur	$2*5625 + 563$	$2*P(\text{Temp}) + P(\text{CoOccur})$
Total IO cost	28688 IOs	

Recall: HDD Scan at 100 MBps, Access = 10 msecs

- Time = Access (28688\*0.01 secs) + Scan (28688\*64MB/100MBps)  
 **$\approx 18.647 \text{ secs}$**
- Total cost  **$\approx \$250$** 
  - Cost (B = 64GB RAM at 100\$/32GB) = ~200\$
  - Cost(HDD = 2 TB at 100\$/4 TB) = ~50\$

⇒ Could optimize by ~2x with more tweaks. **Good enough for cs145.** Went from 6B\$ or 1 million years to above)

## Systems Design Example:

### Product CoOccur

#### Plan #2

[See Notebook for impact of B values]

## Engg Cost Approximation [B = 3000]

- ▶ Step1:  $\text{SMJ}(v1.\text{userid} = v2.\text{userid})$ . Specifically,
  - $\text{SMJ}(\text{UserViews}, \text{UserViews by userid})$ 
    - // Note: Sort and merge UserViews **once**
    - // Normally, SMJ Cost  $\approx 3*(P(R)+P(S))$ .
    - // So SMJ Cost  $\approx 3*P(\text{UserViews})$ , due to self-join in UserViews
  - OUTPUT Temp=<v1.productId, v2.productId>

IO Cost =  $\text{SMJ}(\text{UserViews}, \text{UserViews})$   
 $\approx 3*P(\text{UserViews}) + P(\text{Temp})$
- ▶ Step2: // Handle groupbys
  - Sort(Temp, <v1.productId, v2.productId>)
  - Scan sortedTemp, and make GROUPBYS for <v1.productId, v2.productId>

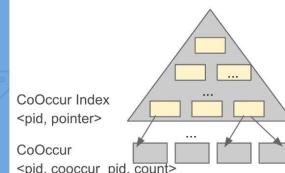
IO Cost = Sort( $\text{UserViews}$ )  
 $\approx 2*P(\text{Temp}) + P(\text{CoOccur})$

Build indexes with search key=productId. (Assume: data not clustered) [[Type Fixed Oct(28 'h' from 1 to 28)]]

$$\frac{\text{PageSize}}{\text{SKeySize} + \text{PointerSize}} \quad // \text{Fit upto } f \text{ (SKey, Pointer)}$$

$$f^h = \text{NumSKeys} // \text{Leaf nodes should point to all SKeys}$$

$$h \geq \log_f \text{NumSKeys} // \text{From previous equation}$$



$$\text{NumSearchKeys(CoOccur)} = 1 \text{ Billion}$$

$$P(\text{CoOccur}) = 563$$

What's f and h?

- f = 5.6 million (4 byte productId + 8 byte pointer @ 64MB/page)
- h = 2

Cost of lookup? (Worst case, with only root in RAM)

- 1 IO (for 1st level) + 1 IO for CoOccur data (assume: clustered for 3 recommendations)
- **2 IOs**

What's f and h?

- f = 5.6 million (4 byte productId + 8 byte pointer @ 64MB/page)
- h = 2

Cost of lookup? (Worst case, with only root in RAM)

- 1 IO (for 1st level) + 1 IO for data
- **2 IOs**



## Big Takeaways

### Case Study 1: Amazon Product Recommendations

- Pre-cs145: Cost ~6 Billion \$\$\$s (or 1 million years)
- Post-cs145 design
  - 8 lines of SQL code
  - ~250\$ and ~18k secs for recommendations
  - Index < 100 msecs for user queries

### Case Study 2: UberEats product launches [Dr. Girish Baliga]

- New launches in 2-3 weeks vs 6 months
- ~100 lines of code in Presto SQL (optimized) + bunch of UI
- Index < 100 msecs for user queries



**1st half**  
**(Rapid version, Review lectures)**

**How to scale queries  
10-100x on large data sets?**



CS145

Goals

# Course Summary

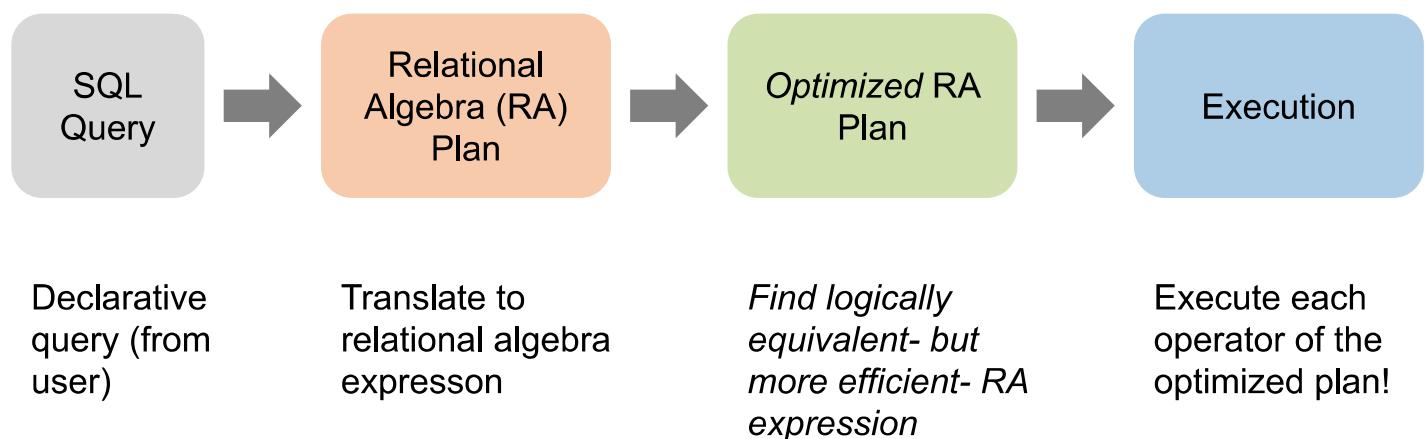
We'll learn How To...

- **Query** over small-med-large data sets with **SQL?** [Weeks 1 and 2]
  - On relational engines, and "big data" engines
- **Scale** for "big queries"? On Clusters? [Weeks 3, 4, 5]
  - OLAP/Analytics, 1st principles of scale, query optimizers
- **Scale** for "big writes"? [Weeks 6, 7, 8]
  - Writes, Transactions, Logging, ACID properties
- **Design** "good" databases? [Weeks 9, 10]
  - Big Schemas, design, functional dependencies

**Project:** Query-Visualize-Learn on GB/TB scale data sets on a Cloud [sql + python]  
**Industry Talks:** Real world talks from Google, Uber, Coinbase

# RDBMS Architecture

How does a SQL engine work ?



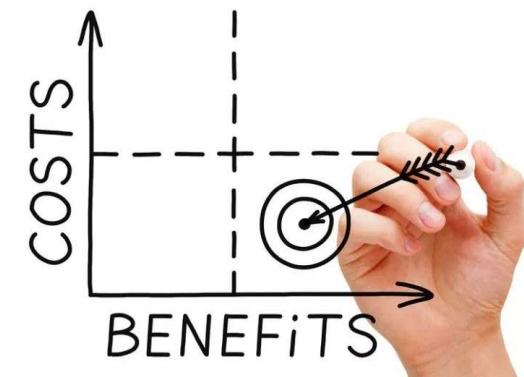
# Optimization

## Roadmap



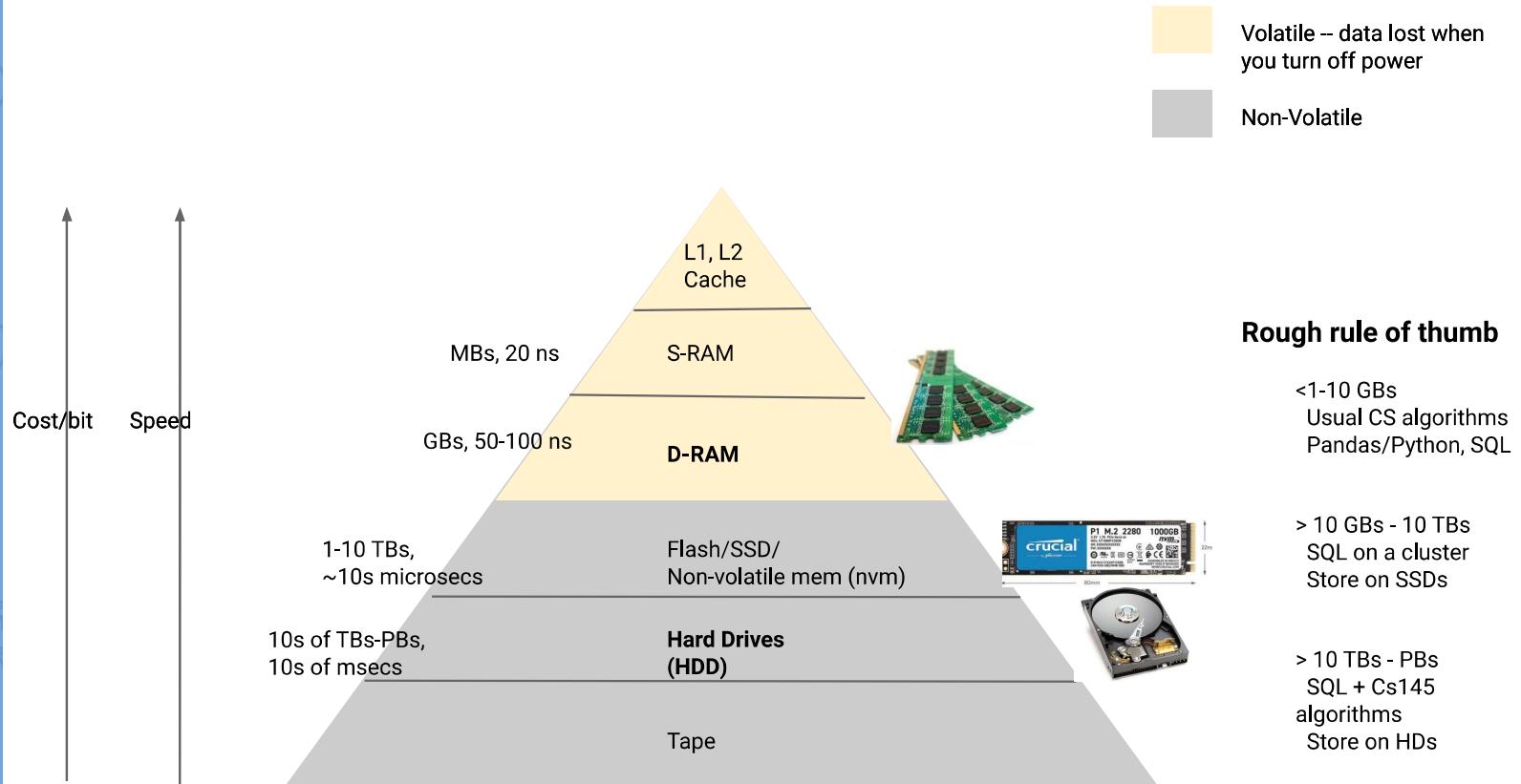
Build Query Plans

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



Analyze Plans

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



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

# Big Scale Lego Blocks

Tech  
Takeaway  
[2/4]



Primary data structures/algorithms

## Hashing

HashTables  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

HashFunctions  
 $(\text{hash}_i(\text{key}) \rightarrow \text{location})$

## Sorting

BucketSort, QuickSort  
MergeSort

## MergeSortedFiles

MergeSort

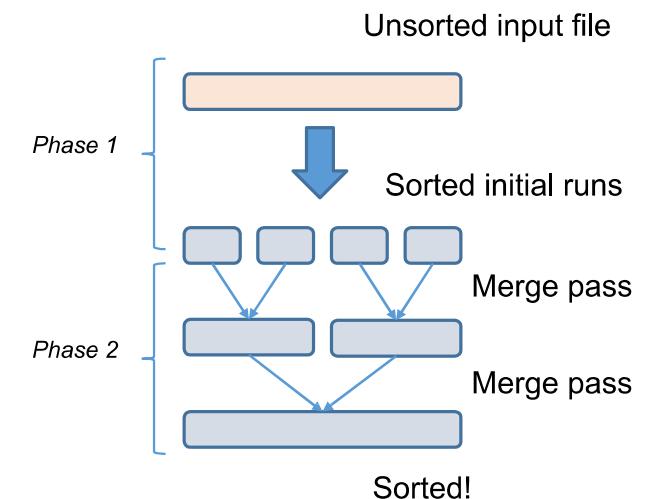
MergeSort

# External Merge Sort Algorithm

**Goal:** Sort a file that is much bigger than RAM

**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!



**IO-Aware algorithms** – try to minimize IO, and *effectively ignore cost of operations in RAM*

# 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!

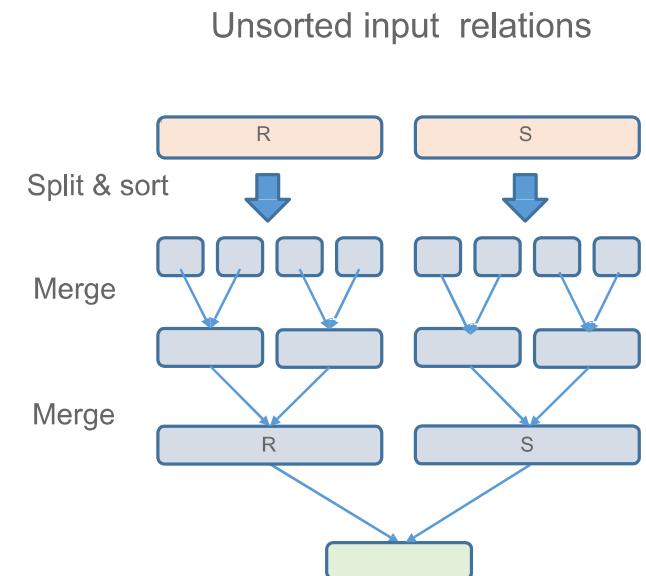
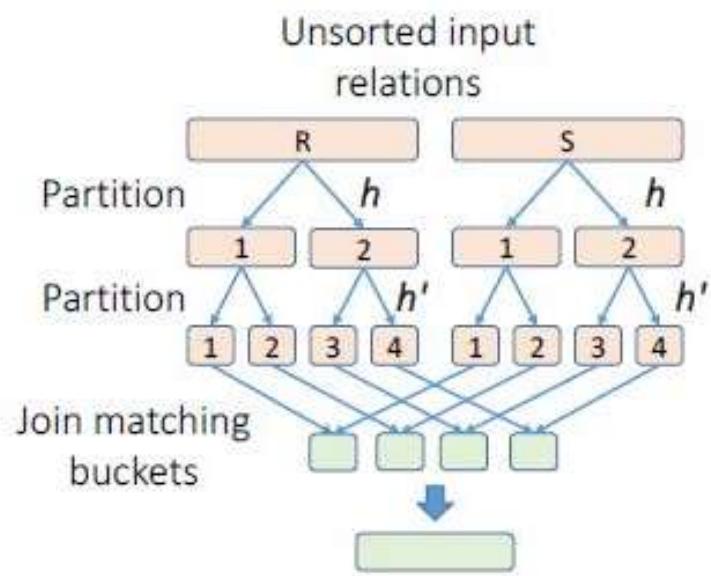
- HPJ: Partition R and S into buckets using a hash function, then join the (much smaller) matching buckets

Given sufficient buffer space, linear in  $P(R)$ ,  $P(S)$   
I.e.  $\sim O(P(R)+P(S))$

By only supporting equijoins & taking advantage of this structure!

# HPJ vs SMJ

*Goal:* Execute  $R \bowtie S$  on A





## 2nd half Highlights (Rapid version, Review lectures)

**How to scale transactions by  
10-100x for large data?**



CS145

Goals

# Course Summary

We'll learn How To...

- **Query** over small-med-large data sets with **SQL?** [Weeks 1 and 2]
  - On relational engines, and “big data” engines
- **Scale** for “big queries”? On Clusters? [Weeks 3, 4, 5]
  - OLAP/Analytics, 1st principles of scale
- **Scale** for “big writes”? [Weeks 6, 7, 8]
  - Writes, Transactions, Logging, ACID properties
- **Design** “good” databases? [Weeks 9, 10]
  - Big Schemas, design, functional dependencies, query optimizers

**Project:** Query-Visualize-Learn on GB/TB scale data sets on a Cloud [sql + python]



## 3 Case Studies for TXNs

- Taylor Swift and Ticketmaster
  - Visa Payments
  - Amazon's Black Friday event

"Need to Master Extreme Transactions" ([Forbes \(Insights\)](#))

# Problems

Problem1: How can a {Bank, Visa, Fintech, NASDAQ} update 100 million records correctly?

Problem2: How to update 100 million records in **Seconds** (not 1000s to 1 million secs)?

Problem3: How to let users have access their accounts?

Problem4: Make it easy for the application engineer!!!



# 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

# Transactions in SQL

- In “ad-hoc” SQL, each statement = one transaction
- In a program, multiple statements can be grouped together as a transaction

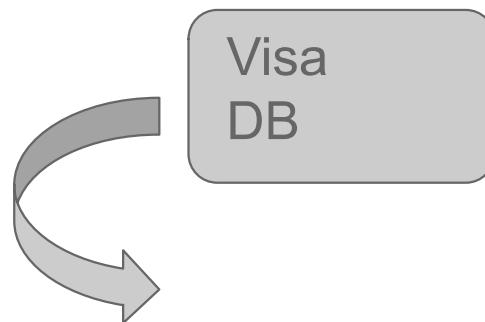
```
START TRANSACTION
    UPDATE Bank SET amount = amount - 100
    WHERE name = 'Bob'
    UPDATE Bank SET amount = amount + 100
    WHERE name = 'Joe'
    COMMIT
```

## Example Visa DB [Takeaway 3 / 4]



### Transaction Queue

- 60000 user TXNs/sec
- Monthly 10% Interest TXN



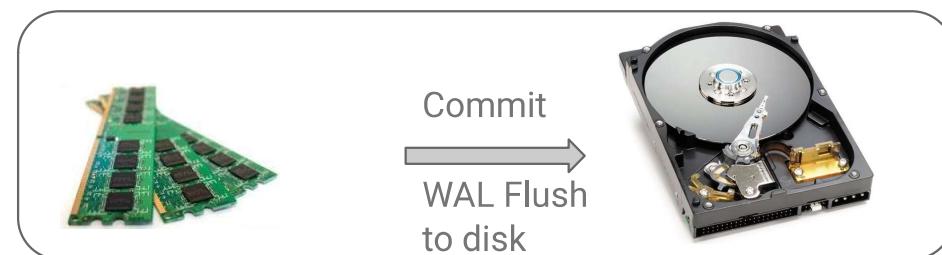
Account	...	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20



### Design#1 VisaDB

For each Transaction in Queue

- For relevant records
  - Use **2 PL** to acquire/release **Locks**
  - Process record
  - **WAL Logs** for updates
- Commit or Abort



# Example Problem 1

Monthly bank interest transaction

With crash

Money		
Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@10:45 am)		
Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-200
5002		320
...	...	
30108		-110
40008		110
50002		22

'T-Monthly-423'

Monthly Interest 10%  
4:28 am Starts run on 10M bank accounts  
Takes 24 hours to run

Network outage at 10:29 am,  
System access at 10:45 am

??

??

??

??

Did T-Monthly-423 complete?  
Which tuples are bad?

Case1: T-Monthly-423 was crashed  
Case2: T-Monthly-423 completed. 4002 deposited 20\$ at 10:45 am



# Write-Ahead Logging (WAL)

## Algorithm: WAL

For each tuple update, **write** Update Record into LOG-RAM

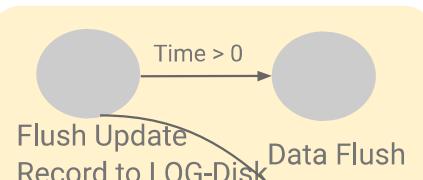
Follow two **Flush** rules for LOG

- Rule1: **Flush** Update Record *into LOG-Disk before corresponding data page goes to storage*
- Rule2: Before TXN commits,
  - **Flush** all Update Records to LOG-Disk
  - **Flush** COMMIT Record to LOG-Disk

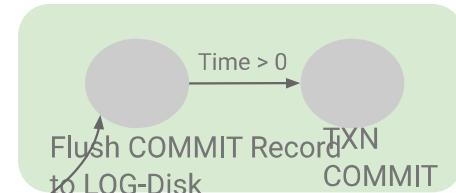
→ **Durability**

→ **Atomicity**

Transaction is committed once **COMMIT record is on stable storage**



Rule1: For each tuple update



Rule2: Before TXN commits

# Example

Monthly  
bank  
interest  
transaction

Money

Account	....	Balance (\$)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...	...	
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance (\$)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...	...	
30108		-110
40008		110
50002		22

WAL (@4:29 am day+1)

T-Monthly-423	<b>START TRANSACTION</b>		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	<b>COMMIT</b>		

'T-Monthly-423'

Monthly Interest 10%  
4:28 am Starts run on 10M bank accounts  
Takes 24 hours to run

```
START TRANSACTION
UPDATE Money
SET Balance = Balance * 1.1
COMMIT
```

# Example- consider two TXNs:

T1: START TRANSACTION

    UPDATE Accounts

    SET Amt = Amt + 100

    WHERE Name = 'A'

    UPDATE Accounts

    SET Amt = Amt - 100

    WHERE Name = 'B'

    COMMIT

T1 transfers \$100 from B's account to A's account

T2: START TRANSACTION

    UPDATE Accounts

    SET Amt = Amt \* 1.06

    COMMIT

T2 credits both accounts with a 6% interest payment

## Note:

1. DB does not care if T1 → T2 or T2 → T1 (which TXN executes first)
2. If developer does, what can they do? (Put T1 and T2 inside 1 TXN)

Serial Schedules

T1		A += 100	B -= 100	
T2				A * = 1.06 B * = 1.06

S1

T1				A += 100	B -= 100
T2		A * = 1.06	B * = 1.06		

S2

Interleaved Schedules

T1			A += 100		B -= 100	
T2				A * = 1.06		B * = 1.06

S3

T1					A += 100		B -= 100
T2		A * = 1.06				B * = 1.06	

S4

T1					A += 100	B -= 100	
T2		A * = 1.06					B * = 1.06

S5

T1			A += 100				B -= 100
T2				A * = 1.06	B * = 1.06		

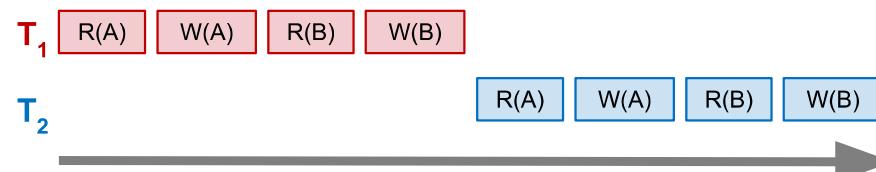
S6

Serial Schedules	S1, S2
Serializable Schedules	S3, S4 (And S1, S2)
Equivalent Schedules	<S1, S3> <S2, S4>
Non-serializable (Bad) Schedules	S5, S6



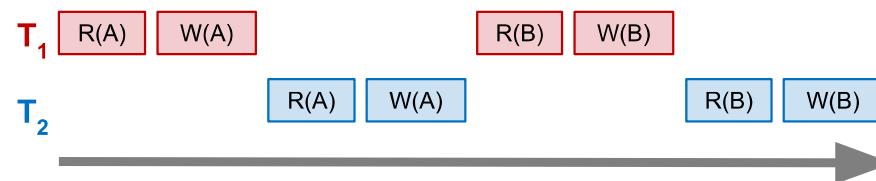
# General DBMS model: Concurrency as Interleaving TXNs

## Serial Schedule



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

## Interleaved Schedule



For our purposes, having TXNs  
occur concurrently means  
**interleaving their component  
actions (R/W)**

We call the particular order  
of interleaving a **schedule**



# 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”?*

Note: **conflicts** happen often in many real world transactions. (E.g., two people trying to book an airline ticket)



# Conflict Serializability

Two schedules are **conflict equivalent** if:

- They involve *the same actions of the same TXNs*
- Every *pair of conflicting actions* of two TXNs are *ordered in the same way*

Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

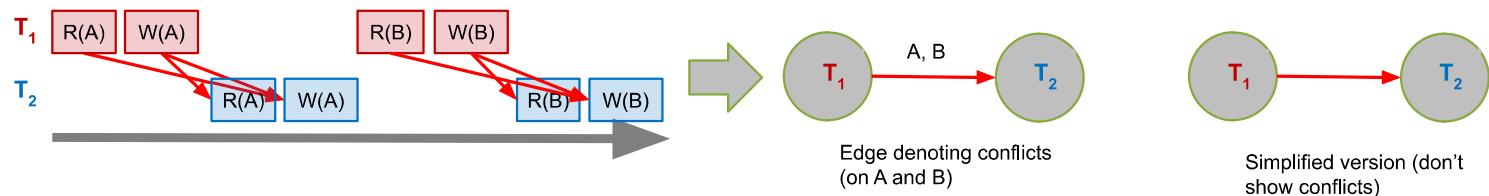
**Conflict serializable  $\Rightarrow$  serializable**

So if we have conflict serializable, we have consistency & isolation!

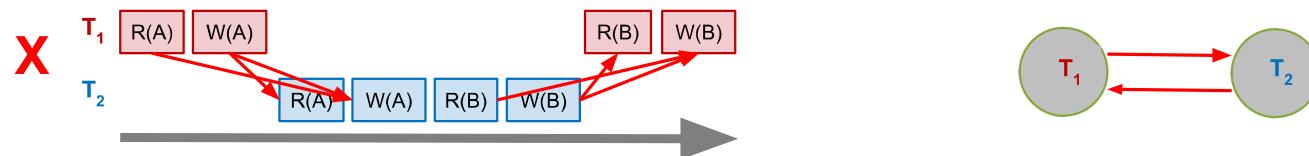


# The Conflict Graph

- For a given schedule, compute Conflict Graph
  - TXNs as **nodes**, and
  - Edge from  $T_i \rightarrow T_j$  if any actions in  $T_i$  precede and conflict with any actions in  $T_j$



**'Bad' Schedule**



⇒ Given an acyclic Conflict Graph, a **topological** ordering of TXNs corresponds to a **serial schedule of TXNs**

### Example with 5 Transactions

Schedule S1

w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

Good or Bad schedule?  
Conflict serializable?

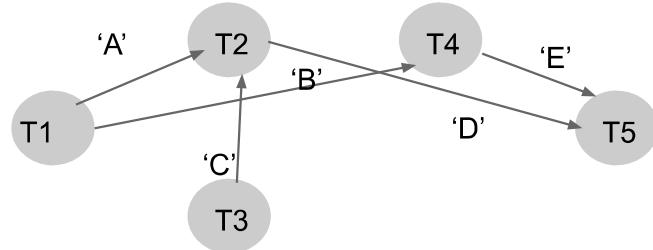
Step1

Find conflicts  
(RW, WW, WR)

T1	w1(A)		w1(B)						
T2		r2(A)			r2(C)		w2(D)		
T3				w3(C)					
T4					r4(B)		w4(E)		
T5								r5(D)	w5(E)

Step2

Build Conflict graph  
Acyclic? Topo Sort



Acyclic  
⇒ Conflict serializable!  
⇒ Serializable

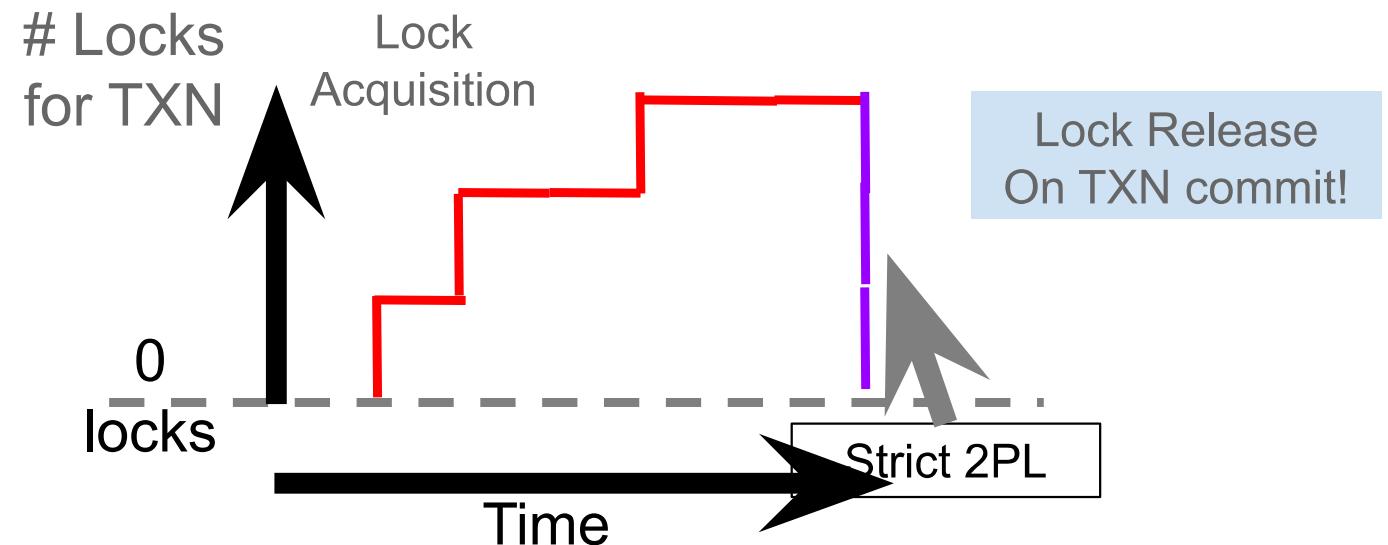
Step3

Example serial schedules  
Conflict Equiv to S1

T3	T1	T4	T2	T5	SerialSched (SS1)
w3(C)	w1(A)	w1(B)	r4(B)	w4(E)	r2(A)
					w2(D) r5(D) w5(E)
T1	T3	T2	T4	T5	SerialSched (SS2)
w1(A)	w1(B)	w3(C)	r2(A)	r2(A)	w2(D) r4(B) w4(E) r5(D) w5(E)



# Strict 2-Phase Locking (S2PL)



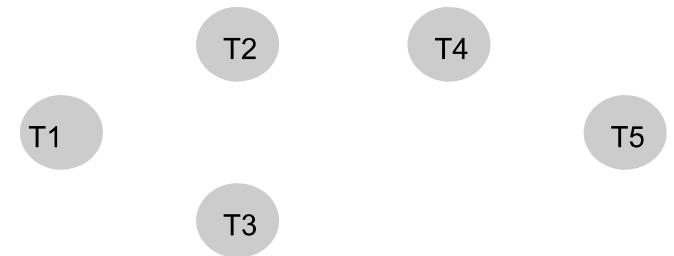
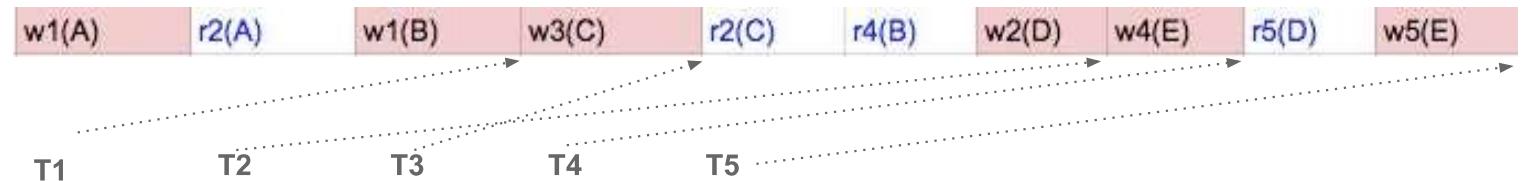
**2-Phase Locking:** A transaction can not request additional locks once it releases any locks. [Phase1: “growing phase” to get more locks. Phase2: “shrinking phase”]

**Strict 2-PL:** Release locks only at COMMIT (COMMIT Record flushed) or ABORT

Example with 5 Transactions (2PL)

Schedule S1

Execute with S2PL

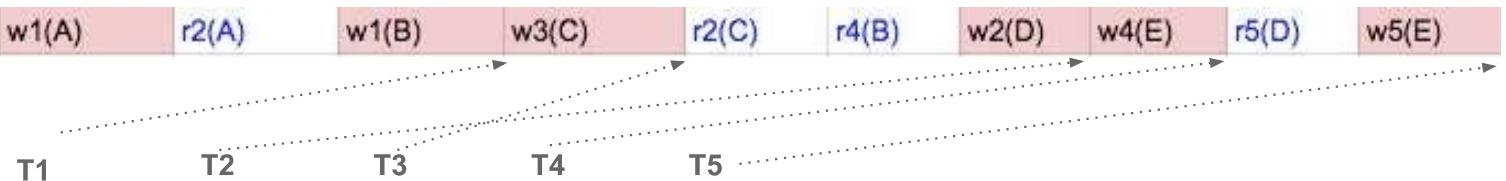


Waits- For Graph

### Example with 5 Transactions (2PL)

Schedule S1

Execute with S2PL



Step 0

X (A)  
**w1(A)**

Req S(A)

Step 1

X (B)

**w1(B)**

Unl B, A

Step 2

Step 3

Step 4

Step 5

Step 6

Step 7

Step 8

Step 9

Step 10

Get S(A)  
**r2(A)**

X (C)  
**w3(C)**  
Unl C

S(C)  
**r2(C)**

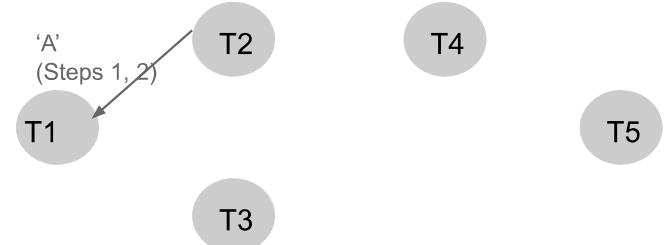
S(B)  
**r4(B)**

X(D)  
**w2(D)**  
Unl A, C, D

X(E)  
**w4(E)**  
Unl B, E

S (D)  
**r5(D)**

X (E)  
**w5(E)**  
Unl D, E

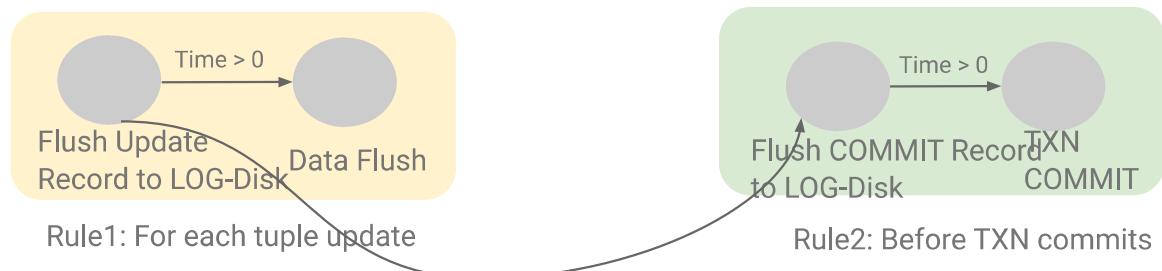


Waits- For Graph



# Example WAL +S2PL scenarios

TXN commit before <u>COMMIT Record</u> on disk?	No
Data page flushed before its <u>Update Record</u> flushed?	No
TXN commit before modified data page flushed?	Yes, often. Especially for large transactions. For TXN Commit, should have Flushed... - All Update Records to Log - COMMIT record to Log
TXN updated “Bob” and committed. TXN2 needs committed data record for ‘Bob’; record still in RAM, not flushed to disk	TXN2 requests/gets LOCK to get latest from memory.
TXN3 updated “John” but not yet committed. TXN4 needs updated record for “John”	TXN4 requests LOCK. Waits for TXN3.



## Example Visa DB -- Need Higher Performance?

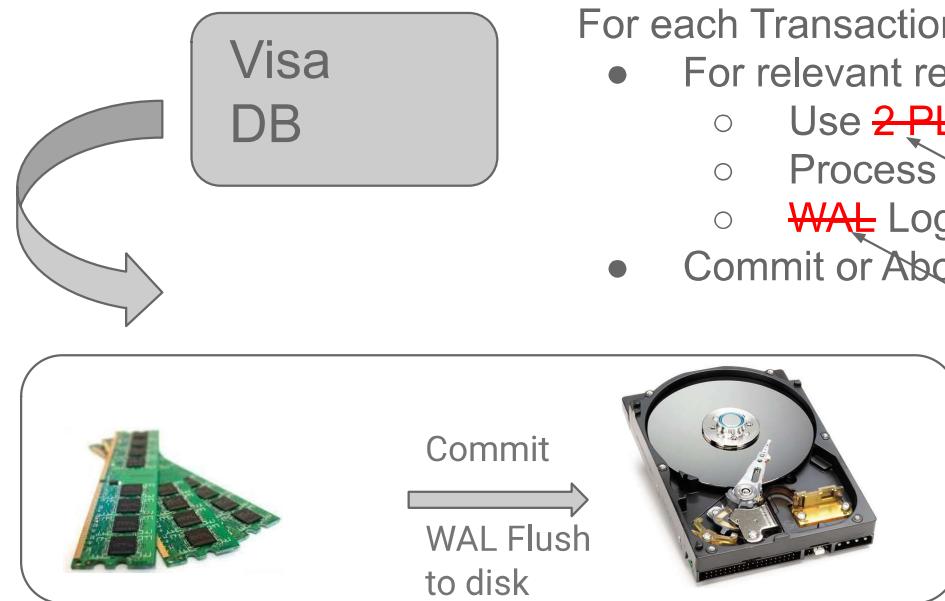


### Transaction Queue

- 60000 TXNs/sec
- Monthly Interest TXN

### 'T-Monthly-423'

Monthly Interest 10%  
4:28 am Starts run on 10M visa accounts  
Takes 24 hours to run



### Design#2 VisaDB

For each Transaction in Queue

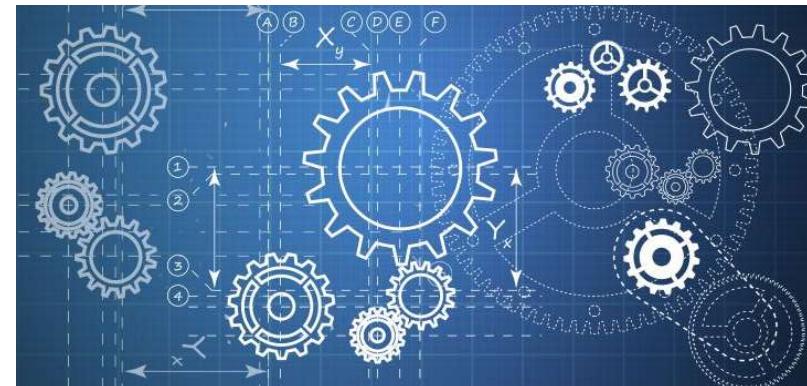
- For relevant records
  - Use **2PL** to acquire/release locks
  - Process record
  - **WAL** Logs for updates
- Commit or Abort

Replace with more sophisticated algorithms  
(cs245/cs345)

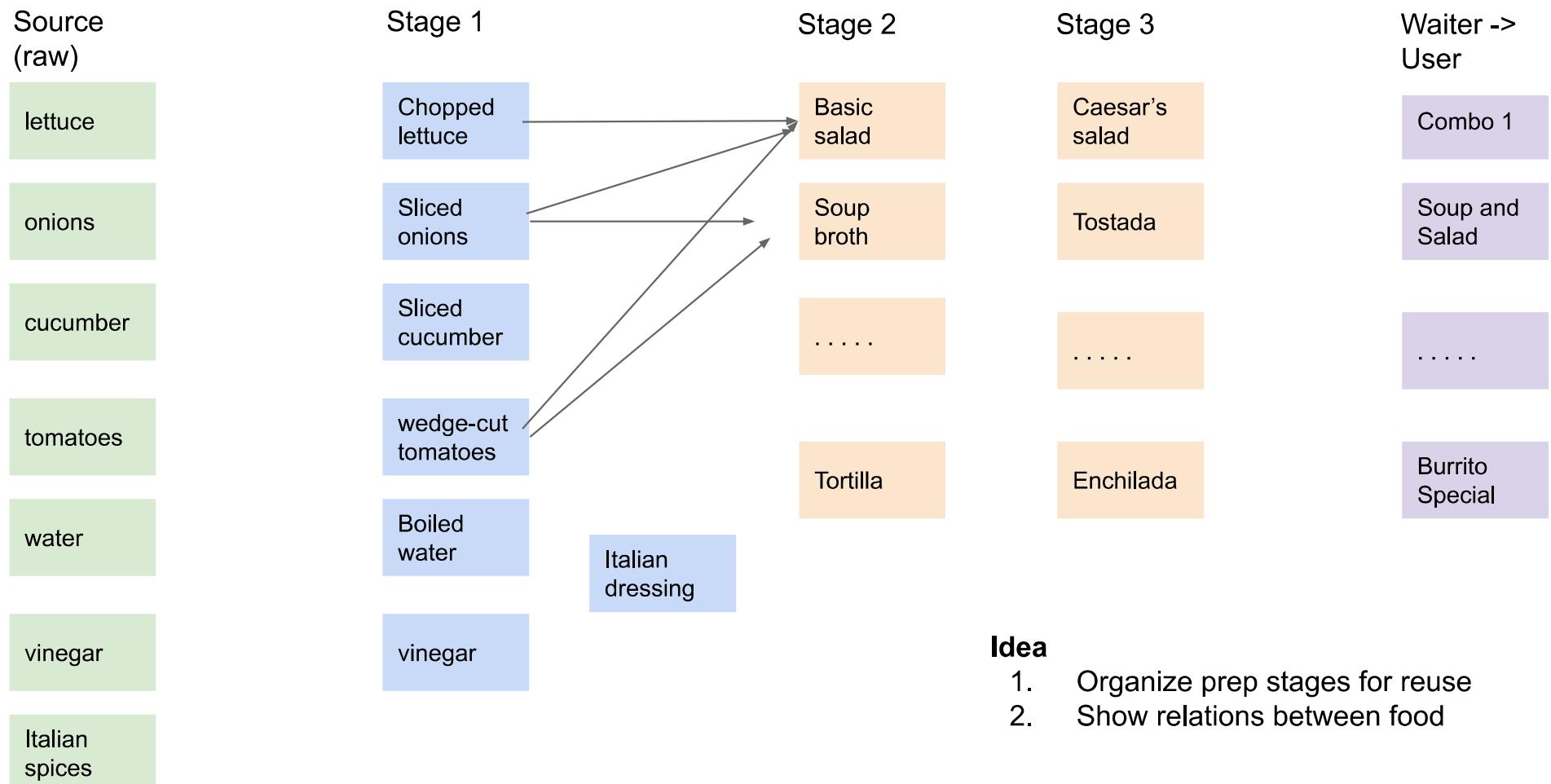


# Design Theory

- Design theory is about how to represent your data to avoid ***anomalies***.
- Simple algorithms for “best practices”



# Intuition: Cooking Prep



## Idea

1. Organize prep stages for reuse
2. Show relations between food



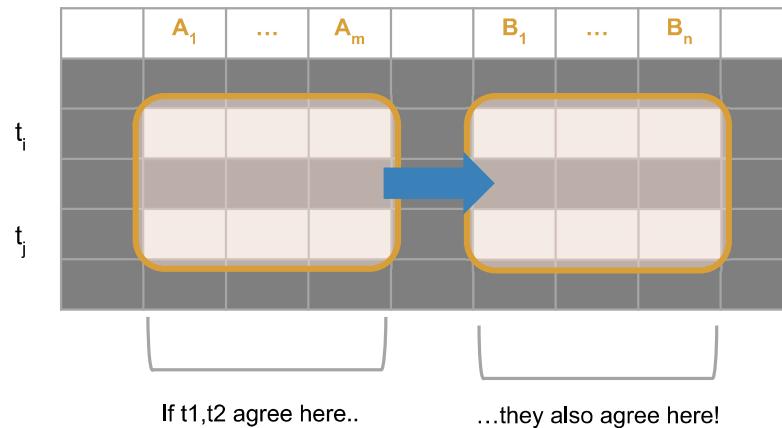
# Relational Schema Design

High-level idea

1. Start with some relational *schema*
2. Find out its *functional dependencies (FDs)*
3. Use these to *design a better schema*  
One which minimizes the possibility of anomalies



# A Picture Of FDs



## Defn (again):

Given attribute sets  $A = \{A_1, \dots, A_m\}$  and  $B = \{B_1, \dots, B_n\}$  in  $R$ ,

The ***functional dependency***  $A \rightarrow B$  on  $R$  holds if for **any**  $t_i, t_j$  in  $R$ :

**if**  $t_i[A_1] = t_j[A_1] \text{ AND } t_i[A_2] = t_j[A_2] \text{ AND } \dots \text{ AND }$   
 $t_i[A_m] = t_j[A_m]$

**then**  $t_i[B_1] = t_j[B_1] \text{ AND } t_i[B_2] = t_j[B_2] \text{ AND } \dots$   
 $\text{AND } t_i[B_n] = t_j[B_n]$



# Finding Functional Dependencies

Given a set of FDs,  $F = \{f_1, \dots, f_n\}$ , does an FD  $g$  hold?

**Inference problem:** How do we decide?

Answer: Three simple rules called  
**Armstrong's Rules.**

1. Split/Combine
2. Reduction
3. Transitivity



# Finding Functional Dependencies

## Example:

### Inferred FDs:

Inferred FD	Rule used
4. {Name, Category} $\rightarrow$ {Name}	Trivial
5. {Name, Category} $\rightarrow$ {Color}	Transitive (4 $\rightarrow$ 1)
6. {Name, Category} $\rightarrow$ {Category}	Trivial
7. {Name, Category} $\rightarrow$ {Color, Category}	Split/Combine (5 + 6)
8. {Name, Category} $\rightarrow$ {Price}	Transitive (7 $\rightarrow$ 3)

### Provided FDs:

1. {Name}  $\rightarrow$  {Color}
2. {Category}  $\rightarrow$  {Dept.}
3. {Color, Category}  $\rightarrow$  {Price}

What's an algorithmic way to do this?



# Keys and Superkeys

A **superkey** is a set of attributes  $A_1, \dots, A_n$  s.t.  
for *any other* attribute  $B$  in  $R$ ,  
we have  $\{A_1, \dots, A_n\} \rightarrow B$

I.e. all attributes are  
*functionally determined* by  
a superkey

A **key** is a *minimal* superkey

Meaning that no subset of a  
key is also a superkey

**Superkey Algorithm:**  
For each set of attributes  $X$

1. Compute  $X^+$
2. If  $X^+ =$  set of all attributes then  
 $X$  is a **superkey**
3. If  $X$  is minimal, then it is a **key**

# Conceptual Design

For a “mega” table



- Search for “bad” dependencies
- If any, *keep decomposing (lossless) the table into sub-tables* until no more bad dependencies
- When done, the database schema is normalized

Recall: there are several normal forms...

## Key takeaway from class

⇒ How to apply CS concepts at scale to solve big problems?

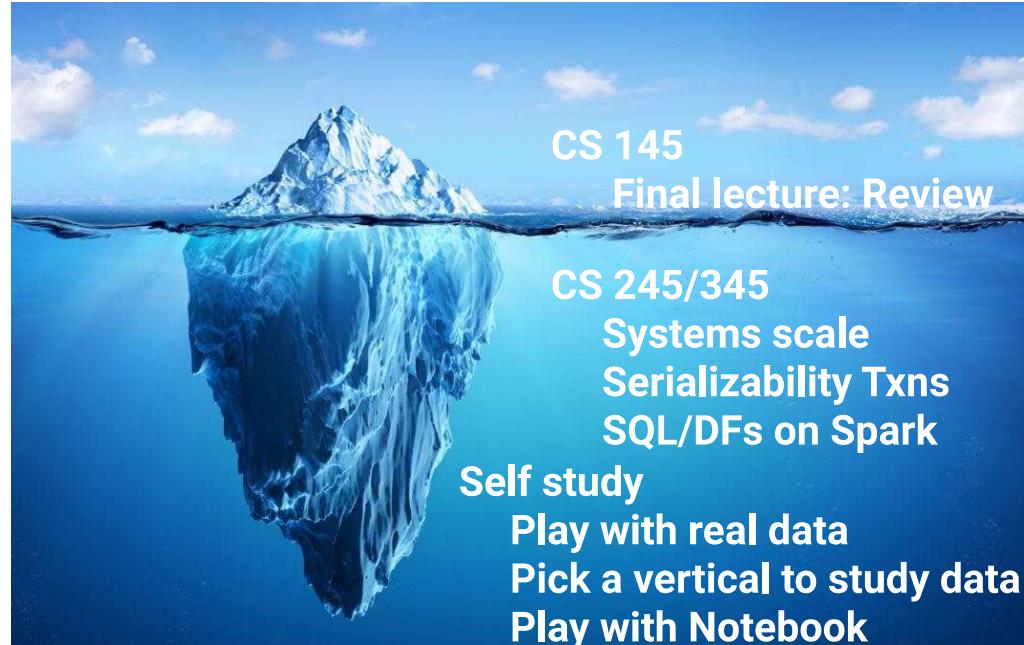
- How do we scale for
  - “Big systems” by 10-100x,
  - “Big queries” by 10-100x,
  - “Big writes” (transactions) by 10-100x
- How to work with real data sets?
  - “Big schemas” for 100s-1000s of Tables, Flows
  - 3 projects

⇒ Why does it matter?

- Examples from Case Studies →
  - 8 lines of code, save billions :-)
  - Launch new product experiences 10x faster
- Data products expect to grow from 100b\$/yr to 1 Trillion \$/yr

# Next Steps

## Where to go next?



# Final project

## Notes

1. Due date today; All extensions expire 12/12 at 11:59 pm
2. If you're asking for extensions, expect not to get additional ones
3. CAs will be grateful (and happy) if you submit before the weekend
4. CAs have ~3 days to grade Finals and Projects by the 15th

Good luck with the Project

# Finals

## Tips

1. Read ED post with logistics
2. Point distribution +/-5 points, per category
3. CAs took ~2 hours to pre-flight the test. However, your mileage may vary! Here are some tips we hope will be helpful to you:
  - If you're confident in your material from lectures and HWs, you'll complete the test in around the same amount of time as the CAs.
  - If you find yourself needing to refer to your notes frequently, you might have a harder time finishing the test in the allotted time.
  - Keep in mind that the test is **open book/notes, NO INTERNET**, so you can use them to your advantage. But it's a good idea to have a strategy for what information you keep in your notes, what you know well, and which problems you want to tackle first.
4. **Review the Schema.** Print/save. <Drumrolls..>

Good luck with the test!

#	Question	Points
1	Instructions/Honor Code	0
2	SQL	27
3	DBSim	23
4	Transactions I	37
5	Transactions II	25
6	Schemas	18
7	Feedback	2
Total		132/130



**CONGRATULATIONS!**

**And**

**THANK YOU**  
**for a fun quarter!**