1. Project 1 due Friday

2. My Office Hours
   - Today, right after lecture
   - Friday on zoom
Common Table Expressions (CTEs)

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

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

Give it a name

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

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

Example Game App

DB v0

(Recap lectures)

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

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
Q6: Predict ads demand?

Report & Share

Business/Product Analysis

Mobile Game

Real-Time User Events

DBMS

DB

App designer

Systems designer

Product/Biz designer
Scale:
Logical $\rightarrow$ Physical DB
Logical → Physical?

Logical Table

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?”
Small tables (e.g., < 1 GB)? “Easy” with tools you already know
- CS Principles
  - Data structures? Linked lists, arrays, trees, hash tables
  - Algorithms? Sorting, Hashing, Dynamic Programming, Graph algorithms, etc.
- How?
  - I/O model ⇒ Work with data in RAM;
  - Language/library? python/c++, pandas libraries, or build your own

Big tables? (e.g., TBs, PetaBytes?)

- How? Expand our tool set with cs145 + advanced systems classes!
  - I/O model ⇒ RAM + HD/SSDs + Clusters
  - Language/library? “Data language” + “I/O-scaling libraries”
IO Hierarchy

- **L1, L2 Cache**: MBs, 3-10 ns
- **S-RAM**: GBs, 10-30 ns
- **D-RAM**: 1-10 TBs, 1ms
- **Flash/SSD/Non-volatile mem (nvm)**: 10s of TBs-PBs, 10s of msec
- **Hard Drives**: Volatile – data lost when you turn off power
- **Tape**: Non-Volatile

**Rough rule of thumb**

- **<1-10 GBs**
  - Usual CS algorithms
  - Pandas + SQL
- **> 10 GBs - 10 TBs**
  - SQL on a cluster
  - Store on SSDs
- **> 10 TBs - PBs**
  - SQL + Cs145 algorithms
  - Store on HDs

⇒ Rest of cs145: Focus on simplified RAM + Disk model (+Clouds)
(learn tools for other IO models)
1. IO Model
2. Data layout
3. Indexing
4. Organizing Data and Indices
Basic IO Model for Reads

- **RAM** Pages (each 64 KB)
- **Disk Blocks** (each 64 KB, like Page)
- **DB Blocks** (contiguous set of Disk Blocks) (e.g., each 64 MBs)

Example RAM size: 64 GB RAM
⇒ 1 million 64 KB pages

Example Disk size: 1 TB
⇒ 15.6M Disk Blocks (= 1 TB/64 KB)
⇒ 15.6K DB Blocks (@ 64 MB/DB block)

In DBs, Page and Disk Block are usually same size.
⇒ In this class, we'll use them interchangeably
A buffer is a part of physical memory used to store intermediate data between disk and processes.

Database maintains its own buffer (Why? Doesn’t the OS already do this?)

- DB knows more about access patterns
- Recovery and logging require ability to flush to disk
- **Buffer Manager** handles page replacement policies
In this Section

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

Company(CName, StockPrice, Date, Country)

Logical Table

How to store table in physical storage ‘files’? (e.g., disk, RAM)
Data Layout

Company(CName, StockPrice, Date, Country)

Logical Table

<table>
<thead>
<tr>
<th>Company</th>
<th>CName</th>
<th>Date</th>
<th>Price</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAPL</td>
<td>Oct1</td>
<td>101.23</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>AAPL</td>
<td>Oct2</td>
<td>102.25</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>AAPL</td>
<td>Oct3</td>
<td>101.6</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>GOOG</td>
<td>Oct1</td>
<td>201.8</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>GOOG</td>
<td>Oct2</td>
<td>201.6</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>GOOG</td>
<td>Oct3</td>
<td>202.13</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>Alibaba</td>
<td>Oct1</td>
<td>407.45</td>
<td>China</td>
<td></td>
</tr>
<tr>
<td>Alibaba</td>
<td>Oct2</td>
<td>400.22</td>
<td>China</td>
<td></td>
</tr>
</tbody>
</table>

Row based storage (aka Row Store)

Column based storage (aka Column Store)
Data Layout

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. Analytics: Lots of data, many exploratory queries on few columns, e.g., Youtube analytics
2. Transactions: Good combination of reads and writes, e.g., Delta Airlines
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 \( \rightarrow \) (the big column)

\[ \Rightarrow \text{Overall size} = \sim 10 \text{ MBs/URL, stored in row format} \]

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?
  - \( \Rightarrow \text{Row based layout: Processing 10 MB*1 billion urls} \)
  - \( \Rightarrow \text{Column based layout: Need to process only 12 bytes * 1 billion urls} \)
  - (~1 million times faster)

- **Core idea**: Exploratory queries usually focus on a few columns
1. IO Model
2. Data layout
3. Indexing
4. Organizing Data and Indices
How to find the right data fast?

Company(CName, StockPrice, Date, Country)

Row based storage (aka Row Store)

Column based storage (aka Column Store)

Next: How to find AAPL Prices?
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
Example

Find Book in Library

Design choices?
- **Scan** through each aisle
- **Lookup** pointer to book location, with librarian's organizing scheme
Algorithm for book titles
- Find right category
- Lookup Index, find location
- Walk to aisle. Scan book titles. Faster if books are sorted
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)

<table>
<thead>
<tr>
<th>CName</th>
<th>Date</th>
<th>Price</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAPL</td>
<td>Oct1</td>
<td>101.23</td>
<td>USA</td>
</tr>
<tr>
<td>AAPL</td>
<td>Oct2</td>
<td>102.25</td>
<td>USA</td>
</tr>
<tr>
<td>AAPL</td>
<td>Oct3</td>
<td>101.6</td>
<td>USA</td>
</tr>
<tr>
<td>GOOG</td>
<td>Oct1</td>
<td>201.8</td>
<td>USA</td>
</tr>
<tr>
<td>GOOG</td>
<td>Oct2</td>
<td>201.61</td>
<td>USA</td>
</tr>
<tr>
<td>GOOG</td>
<td>Oct3</td>
<td>202.13</td>
<td>USA</td>
</tr>
<tr>
<td>Alibaba</td>
<td>Oct1</td>
<td>407.45</td>
<td>China</td>
</tr>
<tr>
<td>Alibaba</td>
<td>Oct2</td>
<td>400.23</td>
<td>China</td>
</tr>
</tbody>
</table>

Q: On which attributes would you build indexes?

A: On as many subsets as you’d like. Look at query workloads.
1. Index contains search key + Block #: e.g., DB block number.
   ○ In general, “pointer” to where the record is stored (e.g., RAM page, DB block number or even machine + DB block)
   ○ 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
External Merge Algorithm

Input: Two sorted files
Output: One merged sorted file

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

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

<table>
<thead>
<tr>
<th>Date</th>
<th>Price</th>
<th>Block #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct1</td>
<td>101.23</td>
<td></td>
</tr>
<tr>
<td>Oct2</td>
<td>102.25</td>
<td></td>
</tr>
<tr>
<td>Oct3</td>
<td>101.6</td>
<td></td>
</tr>
<tr>
<td>Oct1</td>
<td>201.8</td>
<td></td>
</tr>
<tr>
<td>Oct2</td>
<td>201.61</td>
<td></td>
</tr>
<tr>
<td>Oct3</td>
<td>202.13</td>
<td></td>
</tr>
<tr>
<td>Oct1</td>
<td>407.45</td>
<td></td>
</tr>
<tr>
<td>Oct2</td>
<td>400.23</td>
<td></td>
</tr>
</tbody>
</table>
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

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

(Next weeks: Scale, Scale, Scale)
Hashing-Sorting solves “all” known data scale problems :=)

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

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

Roadmap

Primary data structures/algorithms

Hashing

HashTables
(hash(key) -> value)

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)
  - $h_i(x), h_{i+1}(x), h_{i+2}(x), h_{i+3}(x), \ldots$

- Idea: Locality sensitive hash functions (for high 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)
Big Scaling (with Indexes)

Roadmap

Primary data structures/algorithms

Hashing
- HashTables
  \( \text{hash}(\text{key}) \rightarrow \text{value} \)

Sorting
- BucketSort, QuickSort
- MergeSort2Lists, MergeSort
- MergeSortedFiles, MergeSort
Sorting 1: External Merge Algorithm
MergeSortedFiles
(in RAM)

SortedArray1
(m entries)

1 5 7 11 20 31

SortedArray2
(n entries)

2 22 23 24 25 30

OutputSortedArray

1 2 5 7 11 20

22 23 24 25 30 31

Sort in $O(m + n)$
Key (Simple) 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.

If:

\[ A_1 \leq A_2 \leq \cdots \leq A_N \]
\[ B_1 \leq B_2 \leq \cdots \leq B_M \]

Then:

\[ Min(A_1, B_1) \leq A_i \]
\[ Min(A_1, B_1) \leq B_j \]

for \( i=1 \ldots N \) and \( j=1 \ldots M \)
Challenge: Merging Big Files with Small Memory

How do we efficiently merge two sorted files when both are much larger than our main memory 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 lists of length M and N
- 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

**Output:** One *merged* sorted file

<table>
<thead>
<tr>
<th>Disk</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Buffer</td>
</tr>
</tbody>
</table>

- $F_1$: 1, 5, 7, 11, 20, 31
- $F_2$: 2, 22, 23, 24, 25, 30
External Merge Algorithm

Input: Two sorted files

Output: One merged sorted file

Disk

Main Memory

Buffer

F₁

F₂

7,11  20,31

23,24  25,30

1,5  2,22
External Merge Algorithm

Input: Two sorted files

Output: One merged sorted file

- Disk
- Main Memory
- Buffer

Input Files:
- $F_1$: 7,11, 20,31
- $F_2$: 23,24, 25,30
External Merge Algorithm

Input: Two sorted files
Output: One merged sorted file
External Merge Algorithm

Input: Two sorted files

Output: One merged sorted file

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

Disk

Main Memory
Buffer

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

Disk

Main Memory
Buffer

F₁
1,2 20,31
F₂
23,24 25,30

Input:
Two sorted files
Output:
One merged sorted file
External Merge Algorithm

Input:
Two sorted files

Output:
One merged sorted file

Disk

Main Memory

Buffer

11
22
5,7
External Merge Algorithm

Input: Two sorted files

Output: One *merged* sorted file
External Merge Algorithm

Input:
Two sorted files

Output:
One merged sorted file

Disk

Main Memory

Buffer

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
Big Scaling (with Indexes)

Roadmap

Primary data structures/algorithms

Hashing

HashTables
(hash(key) --> value)

Sorting

BucketSort, QuickSort
MergeSort

MergeSortedFiles

??????
1. IO Model

2. Data layout
   ▶ row vs column storage

3. Indexing

4. Organizing Data and Indices
   ▶ Hashing, Sorting, Counting

(Next weeks: Scale, Scale, Scale)