CS 145 Final Review

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

• We learned...

  1. How to design a database

• We got a sense (as the old joke goes) of the three most important topics in DB research:

  • Performance, performance, and performance
Course Summary

• We learned...

1. How to design a database

2. How to query a database, even with concurrent users and crashes / aborts
Course Summary

• We learned...

1. How to design a database

2. How to query a database, even with concurrent users and crashes / aborts

3. How to optimize the performance of a database
Course Summary

• We learned...

  1. How to design a database
  2. How to query a database, even with concurrent users and crashes / aborts
  3. How to optimize the performance of a database

• We got a sense (as the old joke goes) of the three most important topics in DB research:
  • Performance, performance, and performance
High-level: Disk vs. Main Memory

- **Disk:**
  - **Slow**
    - Sequential access
    - (although fast sequential reads)
  - **Durable**
    - We will assume that once on disk, data is safe!
  - **Cheap**
High-level: Disk vs. Main Memory

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

A **transaction (“TXN”)** is a sequence of one or more **operations** (reads or writes) which reflects a **single real-world transition**.

```
START TRANSACTION
  UPDATE Product
  SET Price = Price - 1.99
  WHERE pname = 'Gizmo'
COMMIT
```

In the real world, a TXN either happened completely or not at all.
Motivation for Transactions

Grouping user actions (reads & writes) into transactions helps with two goals:

1. **Recovery & Durability**: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

2. **Concurrency**: Achieving better performance by parallelizing TXNs without creating anomalies
Transaction Properties: ACID

- **Atomic**
  - State shows either all the effects of txn, or none of them
- **Consistent**
  - Txn moves from a state where integrity holds, to another where integrity holds
- **Isolated**
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **Durable**
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!
Basic Idea: (Physical) Logging

• Record UNDO information for every update!
  • Sequential writes to log
  • Minimal info (diff) written to log

• The log consists of an ordered list of actions
  • Log record contains:
    <XID, location, old data, new data>

This is sufficient to UNDO any transaction!
Why do we need logging for atomicity?

• Couldn’t we just write TXN to disk **only** once whole TXN complete?
  • Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  • *With unlimited memory and time, this could work*...

• However, we **need to log partial results of TXNs** because of:
  • Memory constraints (enough space for full TXN??)
  • Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!
...And so we need a **log** to be able to **undo** these partial results!
Transaction Commit Process

1. FORCE Write **commit** record to log

2. All log records up to last update from this TX are FORCED

3. Commit() returns

Transaction is committed *once commit log record is on stable storage*
Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)   A: 0→1

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

OK, Commit!

If we crash now, is T durable?
Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

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

Main Memory

A: 0 → 1

If we crash now, is T durable?

USB THE LOG!
Write-Ahead Logging (WAL)

• DB uses **Write-Ahead Logging (WAL)** Protocol:

1. Must *force log record* for an update *before* the corresponding data page goes to storage

2. Must *write all log records* for a TX *before commit*

Each update is logged! Why not reads?

→ **Atomicity**

→ **Durability**
Why Interleave TXNs?

• Interleaving TXNs might lead to anomalous outcomes... why do it?

• Several important reasons:
  • Individual TXNs might be slow- don’t want to block other users during!
  • Disk access may be slow- let some TXNs use CPUs while others accessing disk!

All concern large differences in performance
Scheduling Definitions

• A **serial schedule** is one that does not interleave the actions of different transactions.

• A and B are **equivalent schedules** if, for any database state, the effect on DB of executing A is identical to the effect of executing B.

• A **serializable schedule** is a schedule that is equivalent to some serial execution of the transactions.

The word “some” makes this definition powerful & tricky!
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)

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

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

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

All “conflicts”!
Conflicts

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

All “conflicts”!
What can we say about “good” vs. “bad” conflict graphs?

**Serial Schedule:**

![Serial Schedule Diagram]

- $T_1$ → $T_2$

**Interleaved Schedules:**

![Interleaved Schedules Diagram]

- $T_1$ → $T_2$
- $T_2$ → $T_1$

**Theorem:** Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**.
DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?

There is none!
Strict Two-phase Locking (Strict 2PL) Protocol:

TXNs obtain:

• An \textit{X (exclusive) lock} on object before \textit{writing}.
  • If a TXN holds, no other TXN can get a lock (S or X) on that object.

• An \textit{S (shared) lock} on object before \textit{reading}
  • If a TXN holds, no other TXN can get \textit{an X lock} on that object.

• All locks held by a TXN are released when TXN completes.

Note: Terminology here- “exclusive”, “shared”- meant to be intuitive- no tricks!
Picture of 2-Phase Locking (2PL)

# Locks the TXN has
0 locks

Lock Acquisition

Lock Release On TXN commit!

Time

Strict 2PL
Deadlocks

• **Deadlock**: Cycle of transactions waiting for locks to be released by each other.

• Two ways of dealing with deadlocks:
  1. Deadlock prevention
  2. Deadlock detection
High-Level: Lecture 11

• The buffer & simplified filesystem model

• Shift to IO Aware algorithms

• The external merge algorithm
High-level: Disk vs. Main Memory

**Disk:**

- **Slow:** Sequential *block* access
  - Read a blocks (not byte) at a time, so sequential access is cheaper than random
  - Disk read / writes are expensive!

- **Durable:** We will assume that once on disk, data is safe!

- **Cheap**

**Random Access Memory (RAM) or Main Memory:**

- **Fast:** Random access, byte addressable
  - ~10x faster for sequential access
  - ~100,000x faster for random access!

- **Volatile:** Data can be lost if e.g. crash occurs, power goes out, etc!

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

- A **buffer** is a region of physical memory used to store temporary data
  - **Key Idea:** Reading / writing to disk is SLOW, need to cache data in main memory
  - Can read into buffer, **flush** back to disk, **release** from buffer

- DBMS manages its own buffer for various reasons (better control of eviction policy, force-write log, etc.)

- We use a simplified model:
  - A **page** is a fixed-length array of memory; **pages are the unit that is read from / written to disk**
  - A **file** is a variable-length list of pages on disk
IO Aware

• Key idea: Reading from / writing to disk- e.g. **IO operations**- is **thousands** of times slower than any operation in memory

• → We consider a class of algorithms which try to minimize IO, and **effectively ignore cost of operations in main memory**
External Merge Algorithm

• **Goal:** Merge sorted files that are much bigger than buffer

• **Key idea:** Since the input files are sorted, we always know which file to read from next!

• **Details:**

<table>
<thead>
<tr>
<th>Given:</th>
<th>( B+1 ) buffer pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>( B ) sorted files, ( F_1, \ldots, F_B ), where ( F_i ) has ( P(F_i) ) pages</td>
</tr>
<tr>
<td>Output:</td>
<td>One merged sorted file</td>
</tr>
<tr>
<td>IO COST:</td>
<td>( 2 \times \sum_{i=1}^{B} P(F_i) ) (Each page is read &amp; written once)</td>
</tr>
</tbody>
</table>
High-Level: Lecture 12

• External Merge Sort Algorithm
  • Basic algorithm (including (B+1)-length initial runs & B-way merging)
  • Repacking optimization for longer initial runs

• Indexes Part I: Basics
External Merge Sort Algorithm

- **Goal**: Sort a file that is much bigger than the buffer

- **Key idea:**
  - *Phase 1*: Split file into smaller chunks ("initial runs") which can be sorted in memory
  - *Phase 2*: Keep merging (do "passes") using external merge algorithm until one sorted file!
# External Merge Sort Algorithm

<table>
<thead>
<tr>
<th>Given:</th>
<th>(B+1) buffer pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>Unsorted file of length (N) pages</td>
</tr>
<tr>
<td>Output:</td>
<td>The sorted file</td>
</tr>
<tr>
<td>IO COST:</td>
<td>(2N(\left\lfloor \log_B \left( \frac{N}{B+1} \right) \right\rfloor + 1))</td>
</tr>
</tbody>
</table>

**Phase 1:** Initial runs of length \(B+1\) are created
- There are \(\left\lfloor \frac{N}{B+1} \right\rfloor\) of these
- The IO cost is \(2N\)

**Phase 2:** We do passes of B-way merge until fully merged
- Need \(\left\lfloor \log_B \left( \frac{N}{B+1} \right) \right\rfloor\) passes
- The IO cost is \(2N\) per pass
Indexes

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

### By_Yr_Index

<table>
<thead>
<tr>
<th>Published</th>
<th>BID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1866</td>
<td>002</td>
</tr>
<tr>
<td>1869</td>
<td>001</td>
</tr>
<tr>
<td>1877</td>
<td>003</td>
</tr>
</tbody>
</table>

### By_Author_Title_Index

<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
<th>BID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dostoyevsky</td>
<td>Crime and Punishment</td>
<td>002</td>
</tr>
<tr>
<td>Tolstoy</td>
<td>Anna Karenina</td>
<td>003</td>
</tr>
<tr>
<td>Tolstoy</td>
<td>War and Peace</td>
<td>001</td>
</tr>
</tbody>
</table>

### Russian_Novels

<table>
<thead>
<tr>
<th>BID</th>
<th>Title</th>
<th>Author</th>
<th>Published</th>
<th>Full_text</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td><em>War and Peace</em></td>
<td>Tolstoy</td>
<td>1869</td>
<td>...</td>
</tr>
<tr>
<td>002</td>
<td><em>Crime and Punishment</em></td>
<td>Dostoyevsky</td>
<td>1866</td>
<td>...</td>
</tr>
<tr>
<td>003</td>
<td><em>Anna Karenina</em></td>
<td>Tolstoy</td>
<td>1877</td>
<td>...</td>
</tr>
</tbody>
</table>

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

An index is *covering* for a specific query if the index contains all the needed attributes.
High-Level: Lectures 14-15

• Indexes Pt. 2:
  • B+ Trees
  • Clustered vs. unclustered

• Join Algorithms:
  • Nested Loop Join Variants: NLJ, BNLJ, INLJ
  • SMJ
  • Hash Join
**B+ Tree Basics**

Each non-leaf ("interior") node has $\geq d$ and $\leq 2d$ keys*

Parameter $d =$ the degree

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

*except for root node, which can have between 1 and $2d$ keys

For each range, in a non-leaf node, there is a pointer to another node with keys in that range
B+ Tree Basics

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

- Their key slots contain pointers to data records
- They contain a pointer to the next leaf node as well, for faster sequential traversal.
Searching a B+ Tree

SELECT name
FROM people
WHERE age = 27

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

Name: John
Age: 21

Name: Jake
Age: 15

Name: Bess
Age: 22

Name: Sally
Age: 28

Name: Bob
Age: 27

Name: Sue
Age: 33

Name: Sal
Age: 30

Name: Jess
Age: 35

Name: Alf
Age: 37

Name: Joe
Age: 11

Name: Bess
Age: 22

Name: Sal
Age: 30

Name: Alf
Age: 37

See L14-15:17-18!
B+ Tree Range Search

- **Goal:** Get the results set of a range (or exact) query with minimal IO

- **Key idea:**
  - A B+ Tree has high **fanout** \((d \approx 10^2 - 10^3)\), which means it is very shallow → we can get to the right root node within a few steps!
  - Then just traverse the leaf nodes using the horizontal pointers

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

---

**Note that exact search is just a special case of range search \((R = 1)\)**

The **fanout** \(f\) is the number of pointers coming out of a node. Thus:

\[
d + 1 \leq f \leq 2d + 1
\]

*Note that we will often approximate \(f\) as constant across nodes!*

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

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

Input: A a range query.

Output: The $R$ values that match

IO COST:

$$\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + \text{Cost(Out)}$$

where $B \geq \sum_{l=0}^{L_B} f^l$

**Depth of the B+ Tree:** For each level of the B+ Tree we read in one node = one page

**# of levels we can fit in memory:** These don’t cost any IO!

**This equation** is just saying that the sum of all the nodes for $L_B$ levels must fit in buffer

See L14-15:22-24
Clustered vs. Unclustered Index

Clustered

Index Entries

Clustered can make a huge difference for range queries!

Unclustered

Index Entries

1 Random Access IO + Sequential IO
(# of pages of answers)

Random Access IO for each value
(i.e. # of tuples in answer)
Joins: Example

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

\[ R \bowtie S \]

**Example**

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

**R**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**S**

<table>
<thead>
<tr>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Final Review > Lectures 14-15**
Join Algorithms: Overview

• NLJ: An example of a non-IO aware join algorithm

• BNLJ: Big gains just by being IO aware & reading in chunks of pages!

• SMJ: Sort R and S, then scan over to join!

• HJ: Partition R and S into buckets using a hash function, then join the (much smaller) matching buckets

For R $\bowtie$ S on A

- Quadratic in $P(R)$, $P(S)$
  - I.e. $O(P(R) \times P(S))$

- 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!
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:

$P(R) + T(R) \times P(S) + OUT$

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)

Note that IO cost based on number of *pages* loaded, not number of tuples!

Have to read *all of $S$* from disk for *every tuple in $R$*!
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 $(r,s)$

Again, $OUT$ could be bigger than $P(R) \times P(S)$... but usually not that bad

Cost:

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

Given $B+1$ pages of memory

1. Load in $B-1$ pages of $R$ at a time (leaving 1 page each free for $S$ & output)
2. For each $(B-1)$-page segment of $R$, load each page of $S$
3. Check against the join conditions
4. Write out
Sort Merge Join (SMJ)

- **Goal:** Execute $R \bowtie S$ 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:* $\sim P(R) + P(S) + \text{OUT}$
    - *Can be worse though - see next slide!"
Simple SMJ Optimization

Given \( B+1 \) buffer pages

Sort Phase  
(Ext. Merge Sort)

Unsorted input relations

Merge Phase

\( \leq B \) total runs

This allows us to “skip” the last sort & save \( 2(P(R) + P(S)) \)!
Hash Join

• **Goal:** Execute \( R \bowtie S \) on \( A \)

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

• **IO Cost:**
  - *Partition phase:* \( 2(P(R) + P(S)) \) each pass
  - *Join phase:* Depends on size of the buckets... can be \( \sim P(R) + P(S) + \text{OUT} \) if they are small enough!
    - *Can be worse though—see next slide!*

Unsorted input relations

\[ \begin{array}{c}
R \\
1 & 2 & 3 & 4 \\
\end{array} \hspace{1cm} \begin{array}{c}
S \\
1 & 2 \\
\end{array} \]

Join matching buckets

\[ \begin{array}{c}
\text{Join result} \\
\end{array} \]
HJ: Skew

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

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

SMJ

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

HJ

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

How many of these passes do we need to do?
### How many passes do we need?

**SMJ**

<table>
<thead>
<tr>
<th># of passes</th>
<th>Length of runs</th>
<th># of runs</th>
<th>Avg. bucket size</th>
<th># of buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$N$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$B+1$</td>
<td>$\left\lfloor \frac{N}{B+1} \right\rfloor$</td>
<td>$\frac{N}{B}$</td>
<td>$B$</td>
</tr>
<tr>
<td>2</td>
<td>$B(B+1)$</td>
<td>$\frac{1}{B} \left\lfloor \frac{N}{B+1} \right\rfloor$</td>
<td>$\frac{N}{B}$</td>
<td>$B^2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>$k+1$</td>
<td>$B^k(B+1)$</td>
<td>$\frac{1}{B^k} \left\lfloor \frac{N}{B+1} \right\rfloor$</td>
<td>$\frac{N}{B}$</td>
<td>$B^{k+1}$</td>
</tr>
</tbody>
</table>

**HJ**

<table>
<thead>
<tr>
<th># of passes</th>
<th>Avg. bucket size</th>
<th># of buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$N$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$\frac{N}{B}$</td>
<td>$B$</td>
</tr>
<tr>
<td>2</td>
<td>$\frac{N}{B}$</td>
<td>$B^2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$k+1$</td>
<td>$\frac{N}{B}$</td>
<td>$B^{k+1}$</td>
</tr>
</tbody>
</table>

Each pass, we get:

- **Fewer, longer** runs by a factor of $B$
- **More, smaller** buckets by a factor of $B$

Each pass costs $2(P(R) + P(S))$
How many passes do we need?

<table>
<thead>
<tr>
<th># of passes</th>
<th>Length of runs</th>
<th># of runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k+1$</td>
<td>$B^k(B+1)$</td>
<td>$\frac{1}{B^k} \left\lceil \frac{N}{B+1} \right\rceil$</td>
</tr>
</tbody>
</table>

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

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

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

*Using the ‘optimization’ on slide 25

<table>
<thead>
<tr>
<th># of passes</th>
<th>Avg. bucket size</th>
<th># of buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k+1$</td>
<td>$\frac{1}{B^k} \left\lceil \frac{N}{B} \right\rceil$</td>
<td>$B^{k+1}$</td>
</tr>
</tbody>
</table>

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

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

$$B^{k+1}(B - 1) \geq \min\{P(R), P(S)\}$$
How many buffer pages for nice behavior?

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

**SMJ**

\[
B(B + 1) \geq P(R) + P(S)
\]

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

\[
B^2 \geq \max\{P(R), P(S)\}
\]

**HJ**

\[
B(B - 1) \geq \min\{P(R), P(S)\}
\]

So approximately:

\[
B^2 \geq \min\{P(R), P(S)\}
\]

\[\rightarrow \text{Total IO Cost} = 3(P(R) + P(S)) + \text{OUT}\!\]
Overview: SMJ vs. HJ

• HJ:
  • PROS: Nice linear performance is dependent on the *smaller relation*
  • CONS: Skew!

• SMJ:
  • PROS: Great if relations are already sorted; output is sorted either way!
  • CONS:
    • Nice linear performance is dependent on the *larger relation*
    • Backup!
High-Level: Lecture 16

• Overall RDBMS architecture

• The Relational Model

• Relational Algebra

Check out the Relational Algebra practice exercises notebook!!
RDBMS Architecture

How does a SQL engine work?

1. **SQL Query**: Declarative query (from user)
2. **Relational Algebra (RA) Plan**: Translate to relational algebra expression
3. **Optimized RA Plan**: Find logically equivalent but more efficient RA expression
4. **Execution**: Execute each operator of the optimized plan!
The Relational Model: Data

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

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Bob</td>
<td>3.2</td>
</tr>
<tr>
<td>002</td>
<td>Joe</td>
<td>2.8</td>
</tr>
<tr>
<td>003</td>
<td>Mary</td>
<td>3.8</td>
</tr>
<tr>
<td>004</td>
<td>Alice</td>
<td>3.5</td>
</tr>
</tbody>
</table>

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

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

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

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

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

• **Derived or auxiliary operators:**
  - Intersection, complement
  - Joins (natural, equi-join, theta join, semi-join)
  - Renaming: $\rho$
  - Division
1. Selection ($\sigma$)

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

SQL:
```
SELECT *
FROM Students
WHERE gpa > 3.5;
```

RA:
$$\sigma_{gpa > 3.5}(Students)$$
2. Projection ($\Pi$)

- Eliminates columns, then removes duplicates
- Notation: $\Pi_{A_1,\ldots,A_n}(R)$

**SQL:**
```sql
SELECT DISTINCT sname, gpa
FROM Students;
```

**RA:**
$$\Pi_{sname, gpa}(Students)$$
3. Cross-Product (×)

• Each tuple in R1 with each tuple in R2
• Notation: R1 × R2
• Rare in practice; mainly used to express joins

SQL:
SELECT *
FROM Students, People;

RA:
Students × People
Renaming ($\rho$)

- Changes the schema, not the instance
- A ‘special’ operator- neither basic nor derived
- Notation: $\rho_{B_1,\ldots,B_n}(R)$

**Note:** this is shorthand for the proper form (since names, not order matters!):

- $\rho_{A_1\to B_1,\ldots,A_n\to B_n}(R)$

**SQL:**

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

**RA:**

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

We care about this operator *because* we are working in a *named perspective*.
Natural Join (⋈)

• Notation: $R_1 \bowtie R_2$

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

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

SQL:

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

RA:

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

Final Review > Lectures 16
Converting SFW Query -> RA

\[
\text{SELECT DISTINCT } A_1, \ldots, A_n \\
\text{FROM } R_1, \ldots, R_m \\
\text{WHERE } c_1 \text{ AND } \ldots \text{ AND } c_k;
\]

\[
\Pi_{A_1, \ldots, A_n} (\sigma_{c_1} \ldots \sigma_{c_k} (R_1 \bowtie \cdots \bowtie R_m))
\]

Why must the selections “happen before” the projections?
High-Level: Lecture 17

- Logical optimization

- Physical optimization
  - Index selections
  - IO cost estimation
Logical vs. Physical Optimization

- **Logical optimization:**
  - Find equivalent plans that are more efficient
  - *Intuition:* Minimize # of tuples at each step by changing the order of RA operators

- **Physical optimization:**
  - Find algorithm with lowest IO cost to execute our plan
  - *Intuition:* Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)
Logical Optimization: “Pushing down” projection

Why might we prefer this plan?
Logical Optimization: “Pushing down” selection

Why might we prefer this plan?
RA commutators

- The basic commutators:
  - Push projection through (1) selection, (2) join
  - Push selection through (3) selection, (4) projection, (5) join
  - Also: Joins can be re-ordered!

- Note that this is not an exhaustive set of operations
  - This covers local re-writes; global re-writes possible but much harder

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

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

**Goal:** Select a set of indexes that minimize execution time of the workload.
- Cost / benefit balance: Each additional index may help with some queries, but requires updating

This is an optimization problem!
IO Cost Estimation via Histograms

• For **index selection**:
  • What is the cost of an index lookup?

• Also for **deciding which algorithm to use**:
  • Ex: To execute $R \bowtie S$, which join algorithm should DBMS use?

  • **What if we want to compute** $\sigma_{A>10}(R) \bowtie \sigma_{B=1}(S)$?

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

Histograms provide a way to efficiently store estimates of these quantities
Histogram types

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

Equi-width
All buckets roughly the same width