CS 145 Final Review
The Best Of Collection (Master Tracks), Vol. 2
Course Announcements

• Course evaluations are online soon—please fill out 😊
  • We want your feedback to improve the course!
  • Tell us what you liked and didn’t!
  • I read every evaluation.

• Project 3 is due on Thursday—No late days! We grade that night!
  • PS#3 is being graded right now!

• Thursday class is Matei Zaharia. *Mandatory guest lecture!*
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

“IO aware” algorithms!
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>
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>$2N(\lceil \log B \lfloor N/B+1 \rfloor / +1)$</td>
</tr>
</tbody>
</table>

**Phase 1:** Initial runs of length $B+1$ are created
- There are $\lceil N/B+1 \rceil$ of these
- The IO cost is $2N$

**Phase 2:** We do passes of B-way merge until fully merged
- Need $\lceil \log B \lfloor N/B+1 \rfloor / \rceil$ passes
- The IO cost is $2N$ per pass
Repacking Optimization for Ext. Merge Sort

• **Goal:** Create larger initial runs

• **Key Idea:** Keep loading unsorted pages, writing out next-largest values, and “repacking” for as long as possible!
  • Guaranteed to do at least as well as our previous method of loading & doing quicksort

• **IO Cost:** On average, we will create initial runs of size $\sim 2(B+1)$
High-Level: Lecture 12

• Indexes Part I: Basics

• B+ Trees

• Clustered vs. unclustered
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>War and Peace</td>
<td>Tolstoy</td>
<td>1869</td>
<td>...</td>
</tr>
<tr>
<td>002</td>
<td>Crime and Punishment</td>
<td>Dostoyevsky</td>
<td>1866</td>
<td>...</td>
</tr>
<tr>
<td>003</td>
<td>Anna Karenina</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
B+ Tree Basics

Each non-leaf (“interior”) node has $\geq d$ and $\leq 2d$ keys* except for root node, which can have between 1 and 2d keys.

Parameter $d = \text{the degree}$

For each range, in a non-leaf node, there is a pointer to another node with keys in that range.

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

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

Non-leaf or *internal* node

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Leaf nodes

<table>
<thead>
<tr>
<th>12</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>Jake</td>
</tr>
<tr>
<td>Age: 11</td>
<td>Age: 15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>22</th>
<th>25</th>
<th>28</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bess</td>
<td>Bob</td>
<td>Sally</td>
<td>Sue</td>
</tr>
<tr>
<td>Age: 22</td>
<td>Age: 27</td>
<td>Age: 28</td>
<td>Age: 33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32</th>
<th>34</th>
<th>37</th>
<th>38</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jess</td>
<td>Sal</td>
<td>Sue</td>
<td>Alf</td>
</tr>
<tr>
<td>Age: 35</td>
<td>Age: 30</td>
<td>Age: 33</td>
<td>Age: 37</td>
</tr>
</tbody>
</table>
Searching a B+ Tree

SELECT name FROM people WHERE age = 27

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

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* \( d \approx 10^2 \cdot 10^3 \), 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:

\[
\lceil \log f \frac{N}{F} \rceil - LB + \text{Cost}(Out)
\]

where $B \geq \sum l=0^{LB-1} f l$

**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

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

Unclustered

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

Clustered can make a huge difference for range queries!
High-Level: Lectures 14-15

• Join Algorithms:
  • Nested Loop Join Variants: NLJ, BNLJ, INLJ
  • SMJ
  • Hash Join

• Lecture 15 walks through in detail on the board.
  • Board discussion in the review slides, here...
Joins: Example

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$
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 \( R \bowtie S \text{ on } A \)

- 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:

$P(R) + T(R) \cdot P(S) + \text{OUT}$

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 $(r, s)$

Given $B+1$ pages of memory

Cost:

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

1. Load in B-1 pages of $R$ at a time (leaving 1 page each free for $S$ & output)

2. For each (B-1)-page segment of $R$, load each page of $S$

3. Check against the join conditions

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

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:* Sort($R$) + Sort($S$)
  - *Merge/join phase:* $\sim P(R) + P(S) +$ OUT
    - *Can be worse though- see next slide!*

---

Unsorted input relations

Split & sort

Merge

Unsorted input relations

Merge

Unsorted input relations
SMJ: Backup

• Without any duplicates:
  • We just scan over R and S once each \( \rightarrow P(R) + P(S) \)

• However, if there are duplicates, we may have to **back up** and re-read parts of the file
  • In worst case have to read in \( P(R)*P(S)! \)
  • In worst case, output is \( T(R)*T(S) \)
  • **Usually not that bad...**
Simple SMJ Optimization

Given $B+1$ buffer pages

Sort Phase (Ext. Merge Sort)
- Split & sort
- Merge
- $\leq B$ total runs

Unsorted input relations
- Split & sort
- Merge

Merge / Join Phase
- $B$-Way Merge / Join

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!*

![Diagram showing partition and join process](Diagram.png)
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)

Note: Ext. Merge Sort!
How many passes do we need?

### SMJ

<table>
<thead>
<tr>
<th># of passes</th>
<th>Length of runs</th>
<th># of runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>B+1</td>
<td>[N/B+1]</td>
</tr>
<tr>
<td>2</td>
<td>B(B+1)</td>
<td>[1/B \cdot N/B+1]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>k+1</td>
<td>B^k(B+1)</td>
<td>[1/B^k \cdot N/B+1]</td>
</tr>
</tbody>
</table>

**Initial sorted runs**

- Each pass, we get:
  - **Fewer, longer** runs by a factor of B

### 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>[N/B]</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>[1/B \cdot N/B]</td>
<td>B^2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>k+1</td>
<td>[1/B^k \cdot N/B]</td>
<td>B^{k+1}</td>
</tr>
</tbody>
</table>

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

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

### SMJ

<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>$1/B^k \left\lceil N/(B+1) \right\rceil$</td>
</tr>
</tbody>
</table>

If $(\text{# of runs of } R) + (\text{# of runs of } S) \leq B$, 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

### HJ

<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>$1/B^k \left\lceil N/B \right\rceil$</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:

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

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

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

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

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

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

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.

### Student

<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>

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

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

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

The number of attributes is the **arity** of the relation.
Relational Algebra (RA)

- Five **basic** operators:
  1. Selection: $\sigma$
  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: $\Pi_{A_1,...,A_n}(R)$

SQL:

```
SELECT DISTINCT sname, gpa
FROM Students;
```

RA:

\[ \Pi_{\downarrow sname,gpa}(Students) \]
3. Cross-Product (×)

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

\[ \text{Students}(\text{sid}, \text{sname}, \text{gpa}) \times \text{People}(\text{ssn}, \text{pname}, \text{address}) \]

**SQL:**

\[
\text{SELECT * FROM Students, People;}
\]

**RA:**

\[ \text{Students} \times \text{People} \]
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!):

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

### SQL:

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

### RA:

$$\rho\downarrow \text{studId, name, gradePtAvg}(\text{Students})$$

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:

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

RA:

\( \text{Students} \bowtie \text{People} \)
Converting SFW Query -> RA

\[
\begin{align*}
\text{SELECT DISTINCT} & \quad A_1, \ldots, A_n \\
\text{FROM} & \quad R_1, \ldots, R_m \\
\text{WHERE} & \quad c_1 \text{ AND } \ldots \text{ AND } c_k;
\end{align*}
\]

\[\Pi A_1, \ldots, A_n (\sigma c_1 \ldots c_k (R_1 \bowtie \ldots \bowtie R_m))\]

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