CSE 132C
Database System Implementation

Arun Kumar

Topic 4: Query Processing; Operator Implementation

Chapters 12.1-12.3 and 14 of Cow Book

Slide ACKs: Jignesh Patel
Lifecycle of a Query

Database Server

Parser → Optimizer → Query Scheduler → Execute Operators

Syntax Tree and Logical Query Plan  Physical Query Plan

Segments

Query Result

Query
Recall the Netflix Schema

<table>
<thead>
<tr>
<th>RatingID</th>
<th>Stars</th>
<th>RateDate</th>
<th>UID</th>
<th>MID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.5</td>
<td>08/27/15</td>
<td>79</td>
<td>20</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UID</th>
<th>Name</th>
<th>Age</th>
<th>JoinDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>Alice</td>
<td>23</td>
<td>01/10/13</td>
</tr>
<tr>
<td>80</td>
<td>Bob</td>
<td>41</td>
<td>05/10/13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MID</th>
<th>Name</th>
<th>Year</th>
<th>Director</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Inception</td>
<td>2010</td>
<td>Christopher Nolan</td>
</tr>
<tr>
<td>16</td>
<td>Avatar</td>
<td>2009</td>
<td>Jim Cameron</td>
</tr>
</tbody>
</table>
### Example SQL Query

```sql
SELECT M.Year, COUNT(*) AS NumBest
FROM Ratings R, Movies M
WHERE R.MID = M.MID
    AND R.Stars = 5
GROUP BY M.Year
ORDER BY NumBest DESC
```

Suppose, we also have a B+Tree Index on Ratings (Stars)
Logical Query Plan

SELECT R.stars = 5
Ratings Table

JOIN R.MID = M.MID
Movies Table

GROUP BY AGGREGATE
M.Year, COUNT(*)

SORT On NumBest

Called “Logical” Operators
From extended RA
Each one has alternate “physical” implementations
Called “Physical” Operators

Specifies exact algorithm/code to run for each logical operator, with all parameters (if any)

This is one of many physical plans possible for a query!
Logical-Physical Separation in DBMSs

Logical = Tells you “what” is computed
Physical = Tells you “how” it is computed

Declarative “querying” (logical-physical separation) is a key system design principle from the RDBMS world:

Declarativity often helps improve user productivity
Enables behind-the-scenes performance optimizations

People are still (re)discovering the importance of this key system design principle in diverse contexts…
(MapReduce/Hadoop, networking, file system checkers, interactive data-vis, graph systems, large-scale ML, etc.)
Operator Implementations

Select
Project
Join
Set Operations
Group By Aggregate

Need scalability to larger-than-memory (on-disk) datasets and high performance at scale!
But first, what metadata does the RDBMS have?
System Catalog

❖ Set of pre-defined relations for metadata about DB (schema)
❖ For each **Relation**:
  Relation name, File name
  File structure (heap file vs. clustered B+ tree, etc.)
  Attribute names and types; Integrity constraints; Indexes
❖ For each **Index**:
  Index name, Structure (B+ tree vs. hash, etc.); IndexKey
❖ For each **View**:
  View name, and View definition
Statistics in the System Catalog

- RDBMS periodically collects stats about DB (instance)
- For each **Table R**: 
  - Cardinality, i.e., number of tuples, $N_{\text{Tuples}}(R)$
  - Size, i.e., number of pages, $N_{\text{Pages}}(R)$, or just $N_R$ or $N$
- For each **Index X**: 
  - Cardinality, i.e., number of distinct keys $I_{\text{Keys}}(X)$
  - Size, i.e., number of pages $I_{\text{Pages}}(X)$ (for a B+ tree, this is the number of leaf pages only)
  - Height (for tree indexes) $I_{\text{Height}}(X)$
  - Min and max keys in index $I_{\text{Low}}(X)$, $I_{\text{High}}(X)$
Operator Implementations

Select

Project

Join

Set Operations

Group By Aggregate

Need scalability to larger-than-memory (on-disk) datasets and high performance at scale!
Selection: Access Path

\[ \sigma_{SelectCondition} (R) \]

- Access path: **how exactly is a table read** ("accessed")
- Two common access paths:
  - **File scan:**
    Read the heap/sorted file; apply SelectCondition
    I/O cost: \(O(N)\)
  - **Indexed:**
    Use an index that **matches** the SelectCondition
    I/O cost: Depends! For equality check, \(O(1)\) for hash index, and \(O(\log(N))\) for B+-tree index
Indexed Access Path

\[ \sigma SelectCondition (R) \]

- An Index **matches** a predicate if it can avoid accessing most tuples that violate the predicate (reduces I/O!)

- **Examples:**
  
  \[ \sigma_{\text{Stars}=5} (R) \]

  Hash index on R(Stars) matches this predicate

  Cl. B+ tree on R(Stars) matches too

  What about uncl. B+ tree on R(Stars)?
Selectivity of a Predicate

$$\sigma_{SelectCondition}(R)$$

- Selectivity of SelectionCondition = percentage of number of tuples in R satisfying it (in practice, count pages, not tuples)

\[
\begin{align*}
\sigma_{\text{Stars}=5}(R) & \\
\text{Selectivity} & = \frac{2}{7} \sim 28\% \\
\sigma_{\text{Stars}=2.5}(R) & \\
\text{Selectivity} & = \frac{3}{7} \sim 43\% \\
\sigma_{\text{Stars}<2}(R) & \\
\text{Selectivity} & = \frac{1}{7} \sim 14\%
\end{align*}
\]
Selectivity and Matching Indexes

- An Index matches a predicate if it brings I/O cost very close to 
  \(N \times \text{predicate’s selectivity}\); compare to file scan!

\[
\sigma_{Stars=5}(R)
\]

\[N \times \text{Selectivity} = 2\]

Hash index on \(R(\text{Stars})\)

Cl. B+ tree on \(R(\text{Stars})\)

Uncl. B+ tree on \(R(\text{Stars})\)?

<table>
<thead>
<tr>
<th>RatingID</th>
<th>Stars</th>
<th>RateDate</th>
<th>UID</th>
<th>MID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.0</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>39</td>
<td>5.0</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>12</td>
<td>2.5</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>402</td>
<td>5.0</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>293</td>
<td>2.5</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>49</td>
<td>1.0</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>66</td>
<td>2.5</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Assume only one tuple per page
Matching an Index: More Examples

<table>
<thead>
<tr>
<th>R</th>
<th>RatingID</th>
<th>Stars</th>
<th>RateDate</th>
<th>UID</th>
<th>MID</th>
</tr>
</thead>
</table>

\[ \sigma_{Stars \geq 4} (R) \]

Hash index on R(Stars) does not match! Why?
Cl. B+ tree on R(Stars) still matches it! Why?
Cl. B+ tree on R(Stars,RateDate)?
Cl. B+ tree on R(Stars,RateDate,MID)?
Cl. B+ tree on R(RateDate,Stars)?
Uncl. B+ tree on R(Stars)?

B+ tree has a nice “prefix-match” property!
Prefix Matching for CNF Predicates

- Express SelectionCondition in **Conjunctive Normal Form (CNF)**, i.e., Pred1 AND Pred2 AND … (each is a “conjunct”)
- Given IndexKey k of B+ tree index, if any prefix subset of k appears in any conjunct, it matches the predicate
- **Example:**

\[ \sigma \text{UID} = 123 \land \text{Stars} = 5 (R) \]

IndexKey (UID, Stars)? (Stars, UID)?
IndexKey (UID, Stars, MID)?
IndexKey (Stars, UID, MID)?
IndexKey (MID, UID, Stars)?
IndexKey UID? IndexKey Stars?

**Conjunct is a prefix of IndexKey**
**IndexKey is a subset of Conjunct:**
“Primary Conjunct”
More Examples for Index Matching

<table>
<thead>
<tr>
<th>R</th>
<th>RatingID</th>
<th>Stars</th>
<th>RateDate</th>
<th>UID</th>
<th>MID</th>
</tr>
</thead>
</table>

\[ \sigma_{UID = 123 \land Stars = 5}(R) \]

Cl. B+ tree index on \( R(UID, \text{Stars}, \text{MID}) \)?
Cl. B+ tree index on \( R(\text{Stars}, \text{MID}, \text{UID}) \)?

Hash index on \( R(UID, \text{Stars}) \)?
Hash index on \( R(UID, \text{Stars}, \text{MID}) \)?
Hash index on \( R(\text{Stars}, \text{MID}, \text{UID}) \)?
Hash index on \( R(UID) \)? On \( R(\text{Stars}) \)?

Hash index does not have the “prefix-match” property of a B+ tree index!

Primary conjuncts!
Matching an Index: Multiple Matches

<table>
<thead>
<tr>
<th>R</th>
<th>RatingID</th>
<th>Stars</th>
<th>RateDate</th>
<th>UID</th>
<th>MID</th>
</tr>
</thead>
</table>

\[ \sigma \text{UID} < 123 \land \text{Stars} > 2.5 \land \text{MID} = 93 \left( R \right) \]

Cl. B+ tree index on R(UID,Stars)?

What if we also have an index (hash or tree) on MID?

*Multiple indexes match non-identical portions of predicate*

*We can use both indexes and intersect the sets of RecordIDs!*

*Sometimes, unions of RecordID sets for disjunctions*
Matching an Index: More Examples

❖ Given hash index on <a> and hash index on <b>
   Predicate: (a = 7 OR b < 5)
   Which index matches? *Neither! Recall CNF!*

❖ Given hash index on <a> and cl. B+ tree index on <b>
   Predicate: (a = 7 AND b < 5)
   Which index matches? *Both! Can intersect RecordIDs!*

❖ Given hash index on <a> and cl. B+ tree index on <b>
   Predicate: (a = 7 OR c > 10) AND (b < 5)
   Which index matches? *Only B+ tree on b*
Operator Implementations

Select

Project

Join

Set Operations

Group By Aggregate

Need scalability to larger-than-memory (on-disk) datasets and high performance at scale!
SELECT R.MID, R.Stars FROM Ratings R

Trivial to implement! Read R and discard other attributes

I/O cost: \( N_R \), i.e., \( \text{Npages}(R) \) (ignore output write cost)

SELECT DISTINCT R.MID, R.Stars FROM Ratings R

Relational Project! \( \pi_{MID, Stars}(R) \)

Need to deduplicate tuples of \((MID, Stars)\) after discarding other attributes; but these tuples might not fit in memory!
**Project: 2 Alternative Algorithms**

\[ \pi_{\text{ProjectionList}}(R) \]

- **Sorting-based:**
  - **Idea:** Sort R on ProjectionList (External Merge Sort!)
  1. In Sort Phase, discard all other attributes
  2. In Merge Phase, eliminate duplicates
  Let T be the temporary “table” after step 1
  - **I/O cost:** \( N_R + N_T + \text{EMSMerge}(N_T) \)

- **Hashing-based:**
  - **Idea:** Build a hash table on R(ProjectionList)
To build a hash table on $R(\text{ProjectionList})$, read $R$ and discard other attributes on the fly.

If the hash table fits entirely in memory:

Done!

I/O cost: $N_R$

Needs $B \geq F \times N_T$

If not, 2-phase algorithm:

Partition

Deduplication

Q: What is the size of a hash table built on a $P$-page file?

$F \times P$ pages

("Fudge factor" $F \sim 1.4$ for overheads)
Hashing

Assuming uniformity, size of a T partition = \( N_T / (B-1) \)
Size of a hash table on a partition = \( F \times N_T / (B-1) \)
Thus, we need: \((B-2) \geq F \times N_T / (B-1)\)
Rough: \( B > \sqrt{F \times N_T} \)
I/O cost: \( N_R + N_T + N_T \)
If \( B \) is smaller, need to partition **recursively**!
Sorting-based vs. Hashing-based:

1. Usually, I/O cost (excluding output write) is the same:
   \[ N_R + 2N_T \] (why is EMSMerge(N_T) only 1 read?)

2. Sorting-based gives sorted result ("nice to have")

3. I/O could be higher in many cases for hashing (why?)

In practice, sorting-based is popular for Project

If we have any index with ProjectionList as a subset of IndexKey
Use only leaf/bucket pages as the "T" for sorting/hashing

If we have tree index with ProjectionList as a prefix of IndexKey
Leaf pages are already sorted on ProjectionList (why?)
Just scan them in order and deduplicate on-the-fly!
Operator Implementations

Select
Project
Join
Set Operations
Group By Aggregate

Need scalability to larger-than-memory (on-disk) datasets and high performance at scale!
This course: we focus primarily on **equi-join** (the most common, important, and well-studied form of join)

<table>
<thead>
<tr>
<th>R</th>
<th>RatingID</th>
<th>Stars</th>
<th>RateDate</th>
<th>UID</th>
<th>MID</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>UserID</td>
<td>Name</td>
<td>Age</td>
<td>JoinDate</td>
<td></td>
</tr>
</tbody>
</table>

\[
U \bowtie_{UserID=UID} R
\]

We study 4 major (equi-) join implementation algorithms:
- Page/Block Nested Loop Join (PNLJ/BNLJ)
- Index Nested Loop Join (INLJ)
- Sort-Merge Join (SMJ)
- Hash Join (HJ)
Nested Loop Joins: Basic Idea

“Brain-dead” idea: nested *for loops* over the tuples of R and U!

1. For each tuple in Users, \( t_U \):
2. For each tuple in Ratings, \( t_R \):
3. If they match on join attribute, “stitch” them, output

*But we read *pages* from disk, not single tuples!*
Page Nested Loop Join (PNLJ)

“Brain-dead” nested *for loops* over the pages of R and U!

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>For each page in Users, ( p_U ) :</td>
</tr>
<tr>
<td>2.</td>
<td>For each page in Ratings, ( p_R ) :</td>
</tr>
<tr>
<td>3.</td>
<td>Check each pair of tuples from ( p_R ) and ( p_U )</td>
</tr>
<tr>
<td>4.</td>
<td>If any pair of tuples match, stitch them, and output</td>
</tr>
</tbody>
</table>

**U** is called “**Outer** table”

**R** is called “**Inner** table”

*I/O Cost:* \( N_U + N_U \times N_R \)

**Q:** How many buffer pages are needed for PNLJ?

Outer table should be the smaller one:

\[ N_U \leq N_R \]
Block Nested Loop Join (BNLJ)

Basic idea: More effective usage of buffer memory (B pages)!

1. For each sequence of B-2 pages of Users at-a-time:
2. For each page in Ratings, \( p_R \):
3. Check if any \( p_R \) tuple matches any \( U \) tuple in memory
4. If any pair of tuples match, stitch them, and output

I/O Cost: \( N_U + \left[ \frac{N_U}{B - 2} \right] \times N_R \)

Step 3 ("brain-dead" in-memory all-pairs comparison) could be quite slow (high CPU cost!)
In practice, a hash table is built on the U pages in-memory to reduce #comparisons (how will I/O cost change above?)
Index Nested Loop Join (INLJ)

Basic idea: If there is an index on R or U, why not use it?

Suppose there is an index (tree or hash) on R (UID)

1. For each sequence of B-2 pages of Users at-a-time:
2. Sort the U tuples (in memory) on UserID
3. For each U tuple $t_U$ in memory:
4. Lookup/probe index on R with the UserID of $t_U$
5. If any R tuple matches it, stitch with $t_U$, and output

I/O Cost: $N_U + NTuples(U) \times I_R$

Index lookup cost $I_R$ depends on index properties (what all?)

A.k.a Block INLJ (tuple/page INLJ are just silly!)

Q: Why does step 2 help? Why not buffer index pages?
Sort-Merge Join (SMJ)

Basic idea: Sort both R and U on join attr. and merge together!

1. Sort R on UID
2. Sort U on UserID
3. Merge sorted R and U and check for matching tuple pairs
4. If any pair matches, stitch them, and output

I/O Cost: \( \text{EMS}(N_R) + \text{EMS}(N_U) + N_R + N_U \)

If we have “enough” buffer pages, an improvement possible: 
No need to sort tables fully; just merge all their runs together!
Sort-Merge Join (SMJ)

**Basic idea:** Obtain runs of R and U and merge them together!

1. Obtain runs of R sorted on UID (only Sort phase)
2. Obtain runs of U sorted on UserID (only Sort phase)
3. Merge all runs of R and U together and check for matching tuple pairs
4. If any pair matches, stitch them, and output

**I/O Cost:** $3 \times (N_R + N_U)$

**How many buffer pages needed?**

- # runs after steps 1 & 2 $\sim N_R/2B + N_U/2B$
- So, we need $B > (N_R + N_U)/2B$
- Just to be safe: $B > \sqrt{N_R}$
- $N_U \leq N_R$
Given tables R and U with $N_R = 1000$, $N_U = 500$, $NTuples(R) = 80,000$, and $NTuples(U) = 25,000$. Suppose all attributes are 8 bytes long (except Name, which is 40 bytes). Let $B = 400$. Let UID be uniformly distributed in R. Ignore output write costs.

1. What is the I/O cost of projecting R on to Stars (with deduplication)?
2. What are the I/O costs of BNLJ and SMJ for a join on UID?
3. What are the I/O costs of BNLJ and SMJ if $B = 50$ only?
4. Which buffer replacement policy is best for BNLJ, if $B = 800$?
Hash Join (HJ)

Basic idea: Partition both on join attr.; join each pair of partitions

1. Partition U on UserID using \( h1() \)
2. Partition R on UID using \( h1() \)
3. For each partition of \( U_i \):
4. Build hash table in memory on \( U_i \)
5. Probe with \( R_i \) alone and check for matching tuple pairs
6. If any pair matches, stitch them, and output

\[ I/O \text{ Cost: } 3 \times (N_U + N_R) \]

\( U \) becomes “Inner table”
\( R \) is now “Outer table”

This is very similar to the hashing-based Project!
Hash Join

Similarly, partition R with same h1 on UID

\[ N_U \leq N_R \]

Memory requirement:

\[ (B-2) \geq F \times \frac{N_U}{(B-1)} \]

Rough:

\[ B > \sqrt{F \times N_U} \]

I/O cost:

\[ 3 \times (N_U + N_R) \]

Q: What if B is lower?

Q: What about skews?

Q: What if \( N_U > N_R \)?

“Hybrid” Hash Join algorithm exploits memory better and has slightly lower I/O cost
Join: Comparison of Algorithms

❖ Block Nested Loop Join vs Hash Join:
    Identical if \((B-2) > F \times N_U\)! Why? I/O cost? Otherwise, BNLJ is potentially much higher! Why?

❖ Sort Merge Join vs Hash Join:
    To get I/O cost of \(3 \times (N_U + N_R)\), SMJ needs: \(B > \sqrt{N_R}\)
    But to get same I/O cost, HJ needs only: \(B > \sqrt{F \times N_U}\)
    Thus, HJ is often more memory-efficient and faster

❖ Other considerations:
    HJ could become much slower if data has skew! Why?
    SMJ can be faster if input is sorted; gives sorted output

❖ Query optimizer considers all these when choosing phy. plan
Join: Crossovers of I/O Costs

We plot the I/O costs of BNLJ, SMJ, and HJ.

8GB memory; 8KB pages
(So, B = 1m)

|U| = 5m; N_\text{U} \sim 195K

|U| = 5m; N_\text{U} \sim 195K

|R| = 500m; N_\text{R} \sim 19.5M

NTuples(R) / 5m

I/O cost in pages

B > N_\text{R} + N_\text{U}

fails

Usually, HJ dominates!
More General Join Conditions

\[ A \Join_{\text{JoinCondition}} B \]

\( N_A \leq N_B \)

- If JoinCondition has only equalities, e.g., A.a1 = B.b1 and A.a2 = B.b2
  - HJ: works fine; hash on \((a1, a2)\)
  - SMJ: works fine; sort on \((a1, a2)\)
  - INLJ: use (build, if needed) a *matching* index on A
  - What about disjunctions of equalities?

- If JoinCondition has inequalities, e.g., A.a1 > B.b1
  - HJ is useless
  - SMJ and INLJ are also mostly unhelpful! Why?
  - Inequality predicates might lead to large outputs!
Operator Implementations

Select
Project
Join
Set Operations
Group By Aggregate

Need scalability to larger-than-memory (on-disk) datasets and high performance at scale!
Set Operations

- **Cross Product**: $A \times B$
  
  Trivial! BNLJ suffices!

- **Intersection**: $A \cap B$
  
  Logically, an equi-join with JoinCondition being a conjunction of all attributes; same tradeoffs as before

- **Union**: $A \cup B$

- **Difference**: $A - B$
  
  Similar to intersection, but need to deduplicate upon matches and output only once!
  Sounds familiar?
Union/Difference Algorithms

- **Sorting-based**: Similar to a SMJ A and B. Twists:
  A ∪ B: *deduplicate* matching tuples during merging
  A – B: *exclude* matching tuples during merging

- **Hashing-based**: Similar to HJ of A and B. Twists:
  Build hash table (h.t.) on Bi
  A ∪ B: probe h.t. with Ai; if pair matches, discard tuple
  else, *insert* Ai tuple into h.t.; h.t. holds output!
  A – B: probe h.t. with Ai; if pair matches, discard tuple
  else, *output* Ai tuple directly
Operator Implementations

Select
Project
Join
Set Operations

Need scalability to larger-than-memory (on-disk) datasets and high performance at scale!
Group By Aggregate

\[ \bigcap_{X, Agg(Y)}(R) \]

“Grouping Attributes” (Subset of \( R \)’s attributes)
“A numerical attribute in \( R \)”

“Aggregate Function” (SUM, COUNT, MIN, MAX, AVG)

❖ Easy case: \( X \) is empty!
Simply aggregate values of \( Y \)

\[ Q: \text{How to scale this to larger-than-memory data?} \]

❖ Difficult case: \( X \) is not empty
“Collect” groups of tuples that match on \( X \), apply \( Agg(Y) \)

3 algorithms: sorting-based, hashing-based, index-based
Group By Aggregate: Easy Case

- All 5 SQL aggregate functions computable incrementally, i.e., one tuple at-a-time by tracking some “running information”

<table>
<thead>
<tr>
<th>RatingID</th>
<th>Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.0</td>
</tr>
<tr>
<td>39</td>
<td>5.0</td>
</tr>
<tr>
<td>12</td>
<td>2.5</td>
</tr>
<tr>
<td>402</td>
<td>5.0</td>
</tr>
<tr>
<td>293</td>
<td>2.5</td>
</tr>
<tr>
<td>49</td>
<td>1.0</td>
</tr>
<tr>
<td>66</td>
<td>2.5</td>
</tr>
</tbody>
</table>

SUM: Partial sum so far
3.0; 8.0; 10.5; 15.5; 18.0; 19, 21.5

COUNT is similar

MAX: Maximum seen so far
3.0; 5.0

MIN is similar
3.0; 2.5; 1.0

**Q:** What about AVG?

Track both SUM and COUNT!
In the end, divide SUM / COUNT
Collect groups of tuples (based on X) and aggregate each

\[ \gamma_{MID, \text{AVG} (Stars)}(R) \]

<table>
<thead>
<tr>
<th>MID</th>
<th>UID</th>
<th>Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>3</td>
<td>3.0</td>
</tr>
<tr>
<td>55</td>
<td>294</td>
<td>5.0</td>
</tr>
<tr>
<td>80</td>
<td>12</td>
<td>2.5</td>
</tr>
<tr>
<td>21</td>
<td>32</td>
<td>5.0</td>
</tr>
<tr>
<td>55</td>
<td>24</td>
<td>2.0</td>
</tr>
<tr>
<td>55</td>
<td>19</td>
<td>1.0</td>
</tr>
<tr>
<td>21</td>
<td>11</td>
<td>4.0</td>
</tr>
<tr>
<td>55</td>
<td>123</td>
<td>4.0</td>
</tr>
</tbody>
</table>

AVG for 21 is 4.0
AVG for 55 is 3.0
AVG for 80 is 2.5

Q: How to collect groups? Too large?
Group By Agg.: Sorting-Based

1. Sort R on X (drop all but X U {Y} in Sort phase to get T)
2. Read in sorted order; for every distinct value of X:
3. Compute the aggregate on that group ("easy case")
4. Output the distinct value of X and the aggregate value

**I/O Cost:** $NR + NT + EMSMerge(NT)$

**Q:** Which other sorting-based op. impl. had this cost?

**Improvement:** Partial aggregations during Sort Phase!

**Q:** How does this reduce the above I/O cost?
1. Build h.t. on X; bucket has X value and running info.
2. Scan R; for each tuple in each page of R:
3. If h(X) is present in h.t., update running info.
4. Else, insert new X value and initialize running info.
5. H.t. holds the final output in the end!

**I/O Cost:** $N_R$

**Q:** What if h.t. using X does not fit in memory
(Number of distinct values of X in R is too large)?
Group By Agg.: Index-Based

- Given B+ Tree index s.t. $X \cup \{Y\}$ is a subset of IndexKey: Use leaf level of index instead of $R$ for sort/hash algo.

- Given B+ Tree index s.t. $X$ is a prefix of IndexKey: Leaf level already sorted! Can fetch data records in order. If AltRecord approach used, just one scan of leaf level!

**Q:** What if it does not use AltRecord?

**Q:** What if $X$ is a non-prefix subset of IndexKey?
1. Suppose we have infinite buffer memory. Which join algorithm will have the lowest I/O cost? What about Project?
2. Given tables A and B such that they are both sorted on the joining attributes. Which join algorithm is preferable?
3. Why does SMJ not suffer from the skew problem HJ does?
4. How does SMJ give sorted outputs? Why not HJ?
5. Given a B+ Tree on Ratings(UID,MID) with AltRecord, what is the I/O cost of computing the average rating for each user? For each movie?
6. How to impl. VARIANCE aggregate efficiently? MEDIAN?