CS 6530: Advanced Database Systems Fall 2023

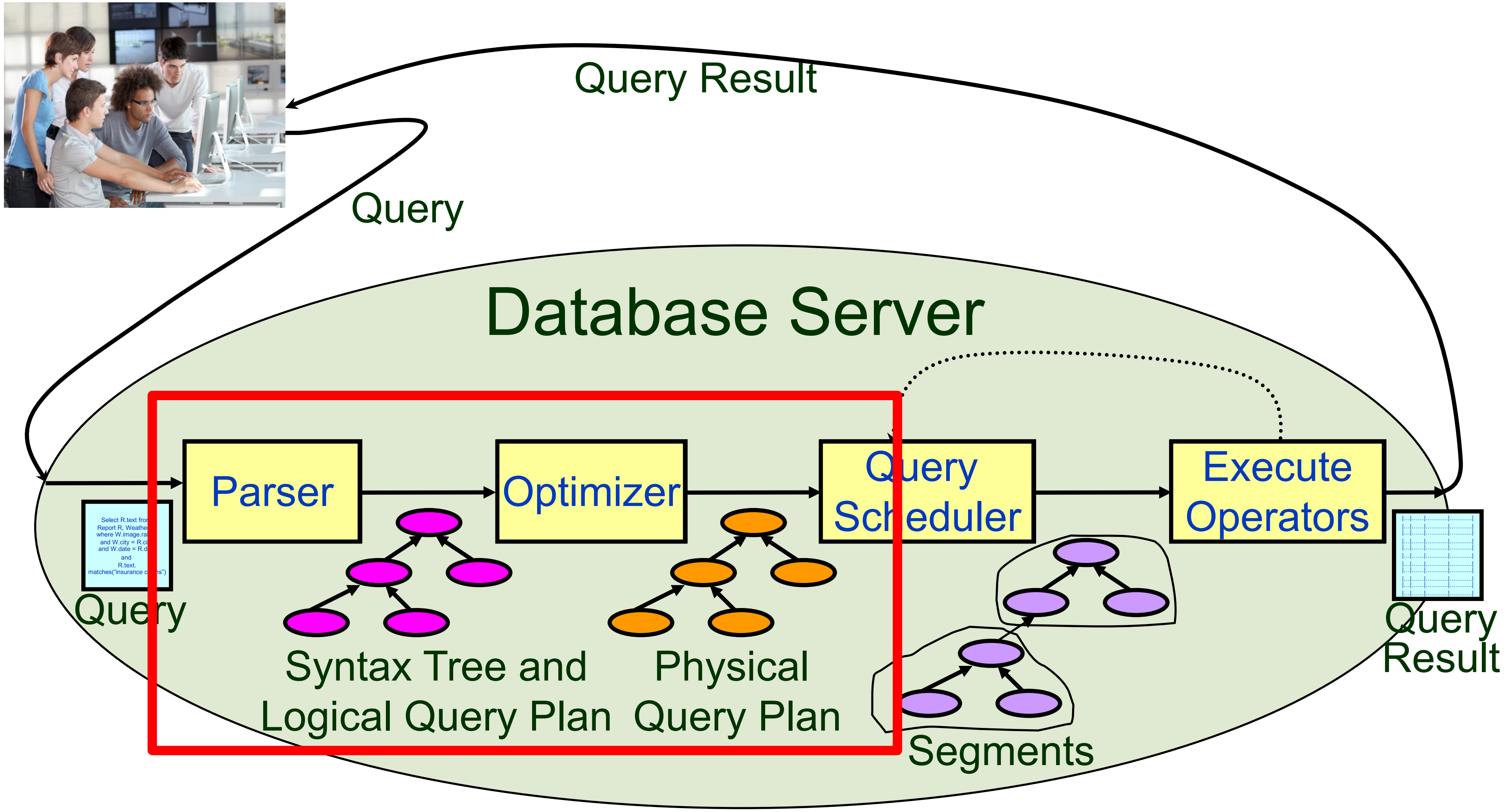
Lecture 13
Query processing

Prashant Pandey
prashant.pandey@utah.edu

Acknowledgement: Slides taken from Prof. Arun Kumar, UCSD

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Lifecycle of a Query



The Netflix Schema

Ratings

1	3.5	08/27/15	79	20
…	…	…	…	…

Users

UID	Name	Age	JoinDate
79	Alice	23	01/10/13
80	Bob	41	05/10/13

Movies

MID	Name	Year	Director
20	Inception	2010	Christopher Nolan
16	Avatar	2009	Jim Cameron

Example SQL Query

RatingID	Stars	RateDate	UID	MID

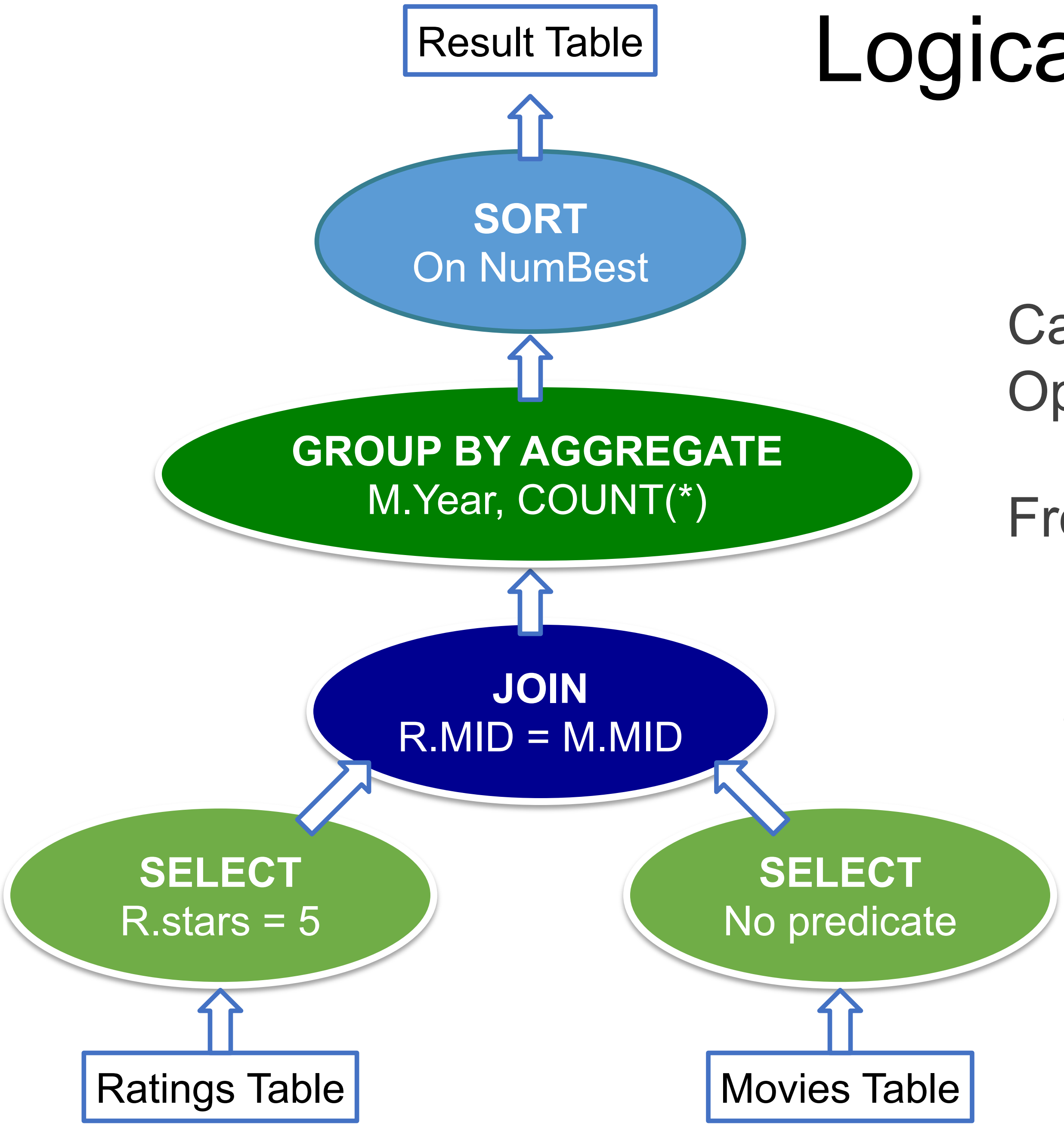
UID	Name	Age	JoinDate

MID	Name	Year	Director

```
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

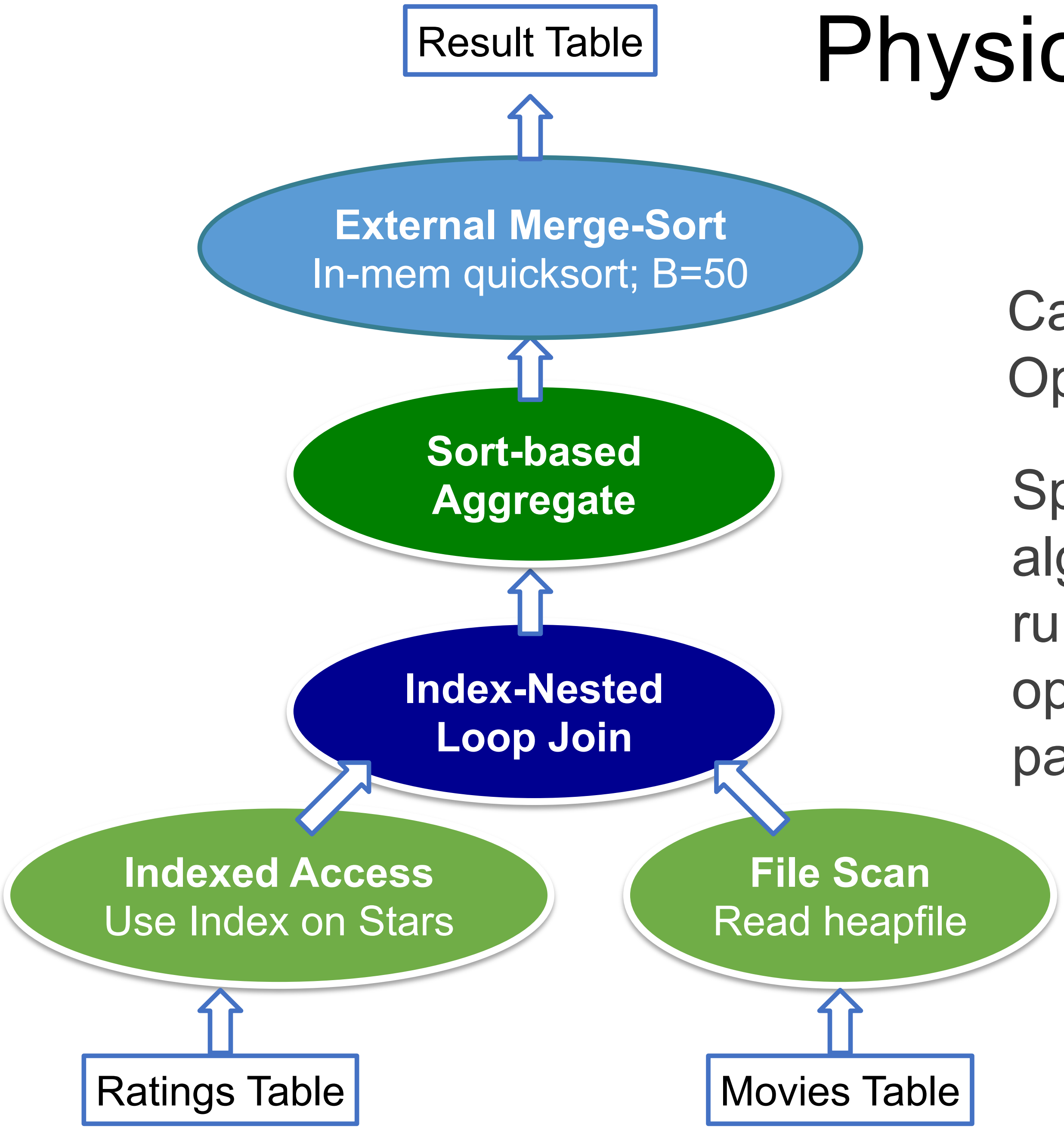


Called "**Logical**" Operators

From extended RA

- Each one has alternate "physical" implementations

Physical Query Plan

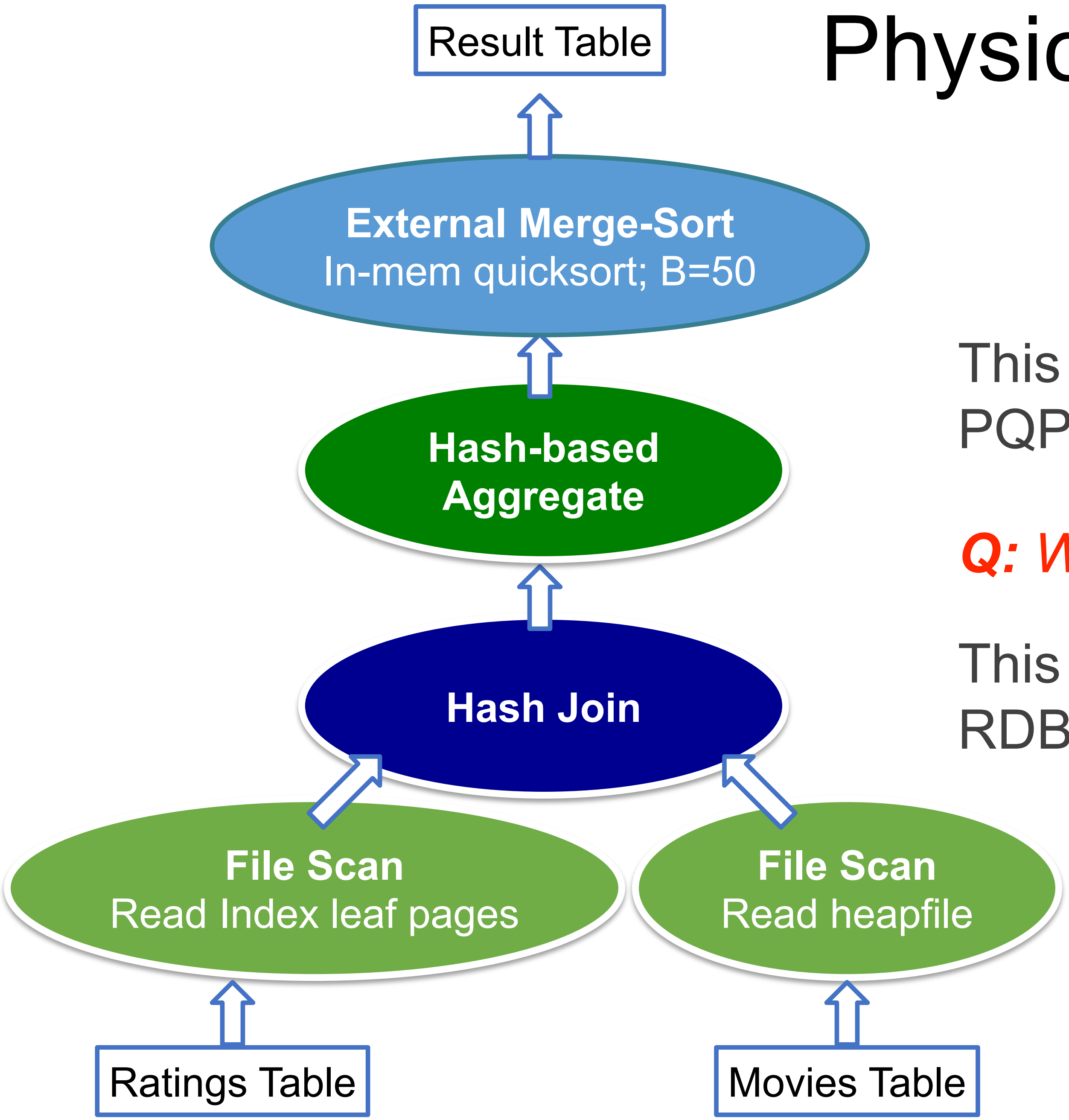


Called “**Physical**” Operators

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

Aka “**Query Evaluation Plan**”

Physical Query Plan



This is also a correct PQP for the given LQP!

***Q:** Which PQP is faster?*

This is a key job of the RDBMS Query Optimizer!

Logical-Physical Separation in DBMSs

Logical = Tells you "what" is computed

Physical = Tells you "how" it is computed

Declarativity!

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

Group By Aggregate

(Optional) Set Operations

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, **NTuples (R)**

    Size, i.e., number of pages, **NPages (R)**, or just $\mathbf{N_R}$ or **N**
- For each **Index X**:

    Cardinality, i.e., number of distinct keys **IKeys (X)**

    Size, i.e., number of pages **IPages (X)** (for a B+ tree, this is the number of leaf pages only)

    Height (for tree indexes) **IHeight (X)**

    Min and max keys in index **ILow (X)**, **IHigh (X)**

Operator Implementations

Select

Project

Join

Group By Aggregate

(Optional) Set Operations

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

Selection: Access Path

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

- 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

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

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

R

RatingID	Stars	RateDate	UID	MID

$$\sigma_{\text{Stars}=5}(\mathbf{R})$$

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

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

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

$$\sigma_{Stars=5}(\mathbf{R})$$

Selectivity = 2/7 ~ 28%

$$\sigma_{Stars=2.5}(\mathbf{R})$$

Selectivity = 3/7 ~ 43%

$$\sigma_{Stars<2}(\mathbf{R})$$

Selectivity = 1/7 ~ 14%

R

2	3.0	…	…	…
39	5.0	…	…	…
12	2.5	…	…	…
402	5.0	…	…	…
293	2.5	…	…	…
49	1.0	…	…	…
66	2.5	…	…	…

Selectivity and Matching Indexes

- An Index matches a predicate if it brings I/O cost very close to (N * predicate's selectivity); compare to file scan!

$$\sigma_{Stars=5}(\mathbf{R})$$

N x Selectivity = 2

Hash index on R(Stars)

Cl. B+ tree on R(Stars)

Uncl. B+ tree on R(Stars)?

R

2	3.0	…	…	…
39	5.0	…	…	…
12	2.5	…	…	…
402	5.0	…	…	…
293	2.5	…	…	…
49	1.0	…	…	…
66	2.5	…	…	…

Assume only one tuple per page

Matching an Index: More Examples

R	RatingID	Stars	RateDate	UID	MID

$$\sigma_{Stars>4}(\mathbf{R})$$

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!

Operator Implementations

Select

Project

Join

Group By Aggregate

(Optional) Set Operations

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

Project

R	RatingID	Stars	RateDate	UID	MID

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

  Trivial to implement! Read R and discard other attributes

  I/O cost: N_R, *i.e., Npages(R) (ignore output write cost)*

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

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

Need to deduplicate tuples of (MID,Stars) after discarding other attributes; but these tuples might not fit in memory!

Project: 2 Alternative Algorithms

$$\pi_{ProjectionList}(\mathbf{R})$$

- **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: $N_R + N_T + EMSMerge(N_T)$

- **Hashing-based**:

  Idea: Build a hash table on R(ProjectionList)

Hashing-based Project

$$\pi_{ProjectionList}(\mathbf{R})$$

- To build a hash table on R(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 >= F x N_R
- If not, 2-phase algorithm:

  Partition

  Deduplication

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

F x P pages

("**Fudge factor**" F ~ 1.4 for overheads)

Hashing

Assuming uniformity, size of a T partition $= N_T / (B-1)$

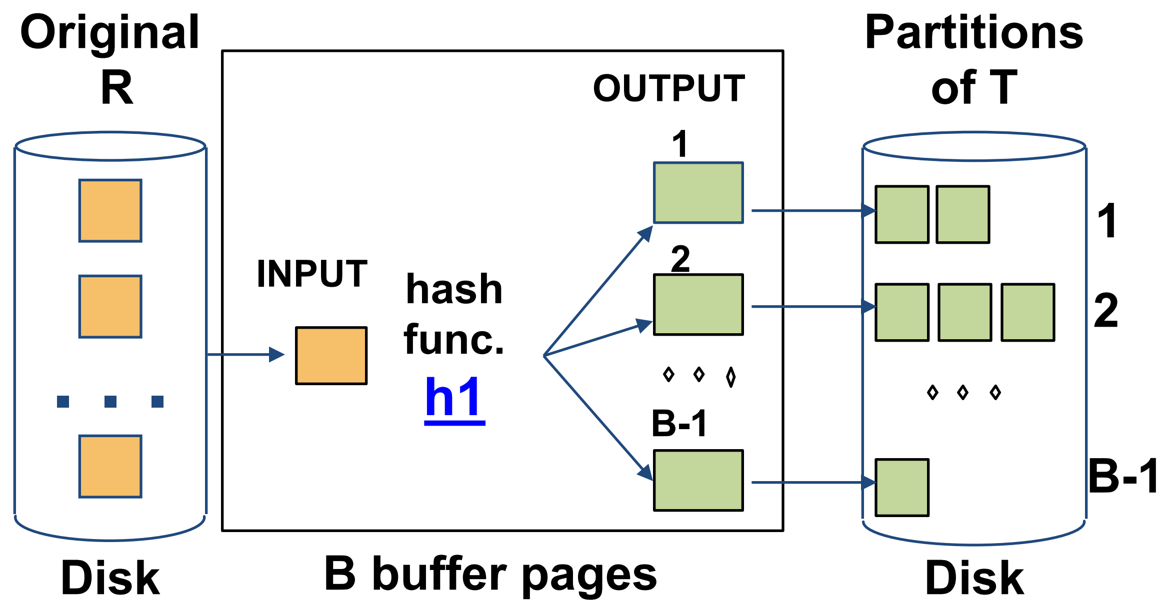
Size of a hash table on a partition $= F \times N_T / (B-1)$

Thus, we need: $(B-2) >= F \times N_T / (B-1)$

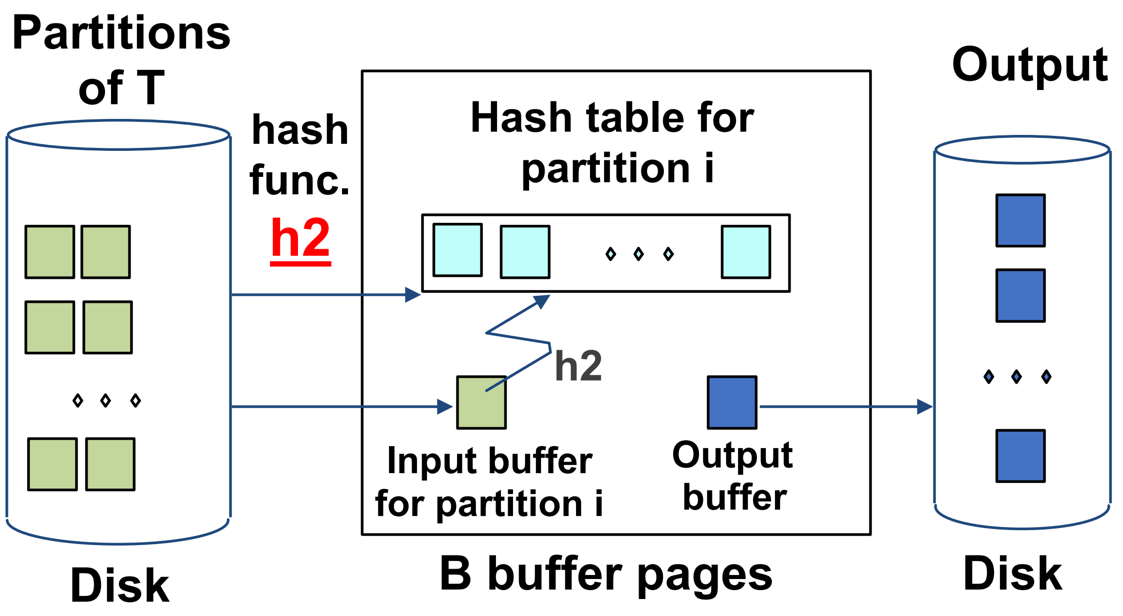
Rough: $B > \sqrt{F \times N_T}$

I/O cost: $N_R + N_T + N_T$

If B is smaller, need to partition recursively!



Partition phase



Deduplication phase

Project: Comparison of Algorithms

- 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 subset of IndexKey

  Use only leaf/bucket pages as the "T" for sorting/hashing
- If we have tree index with ProjectionList as 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

Group By Aggregate

(Optional) Set Operations

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

Join

This course: we focus primarily on equi-join
(the most common, important, and well-studied form of join)

R	RatingID	Stars	RateDate	UID	MID

U	UserID	Name	Age	JoinDate

$$\mathbf{U} \bowtie_{UserID=UID} \mathbf{R}$$

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 <u>pages</u> from disk, not single tuples!

Page Nested Loop Join (PNLJ)

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

1. For each page in Users, p_U :
2. For each page in Ratings, p_R :
3. Check each pair of tuples from p_R and p_U
4. If any pair of tuples match, stitch them, and output

U is called “Outer table”

R is called “Inner table”

I/O Cost: $N_U + N_U \times N_R$

Outer table should be the smaller one:

$N_U \leq N_R$

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

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: $N_U + \left\lceil \frac{N_U}{B-2} \right\rceil \times N_R$

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) x I_R

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

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

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: $EMS(N_R) + EMS(N_U) + N_R + N_U$

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 ~ $N_R/2B + N_U/2B$

So, we need $B > (N_R + N_U)/2B$

Just to be safe: $B > \sqrt{N_R}$

$N_U \leq N_R$

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 Ui :
4. Build hash table in memory on Ui
5. Probe with Ri alone and check for matching tuple pairs
6. If any pair matches, stitch them, and output

$N_U \leq N_R$

I/O 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!

Join: Comparison of Algorithms

$N_U \leq N_R$

B buffer pages

- 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

More General Join Conditions

$$A \bowtie_{JoinCondition} B$$

$N_A \leq N_B$

- 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 also mostly unhelpful! Why?
  - INLJ: build a B+ tree index on A
  - Inequality predicates might lead to large outputs!

Operator Implementations

Select

Project

Join

Group By Aggregate

(Optional) Set Operations

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

Group By Aggregate

$$\gamma_{X, Agg(Y)}(\mathbf{R})$$

"**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:** 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

Operator Implementations

Select

Project

Join

Group By Aggregate

(Optional) Set Operations

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

Set Operations

- **Cross Product**: A × B

  Trivial! BNLJ suffices!

- **Intersection**: A ∩ B

  Logically, an equi-join with JoinCondition being a conjunction of all attributes; same tradeoffs as before

- **Union**: A ∪ 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