CS 6530: Advanced Database Systems Fall 2022

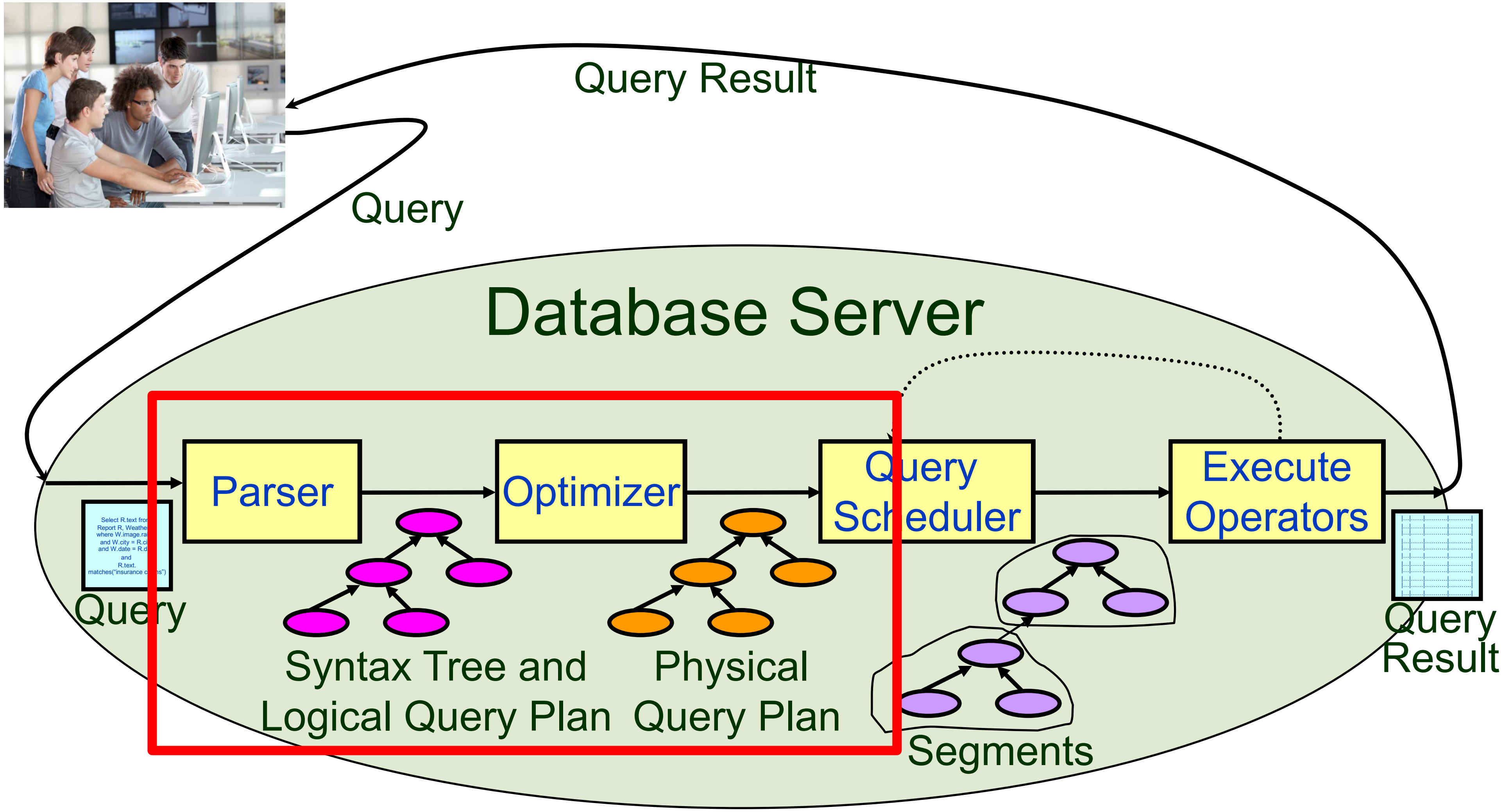
Lecture 14
Query processing and optimization

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

<u>RatingID</u>	Stars	RateDate	UID	MID

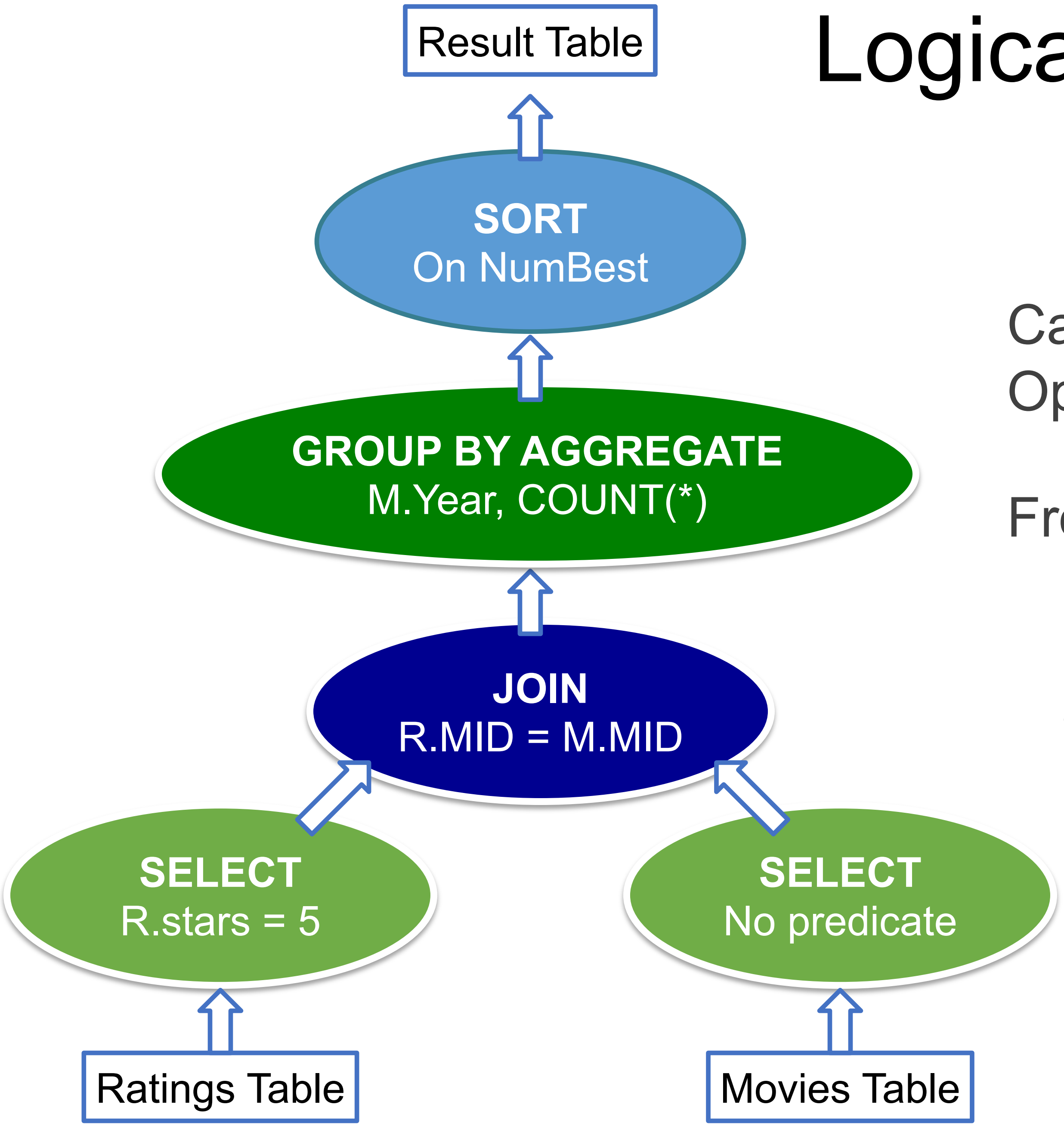
<u>UID</u>	Name	Age	JoinDate

<u>MID</u>	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

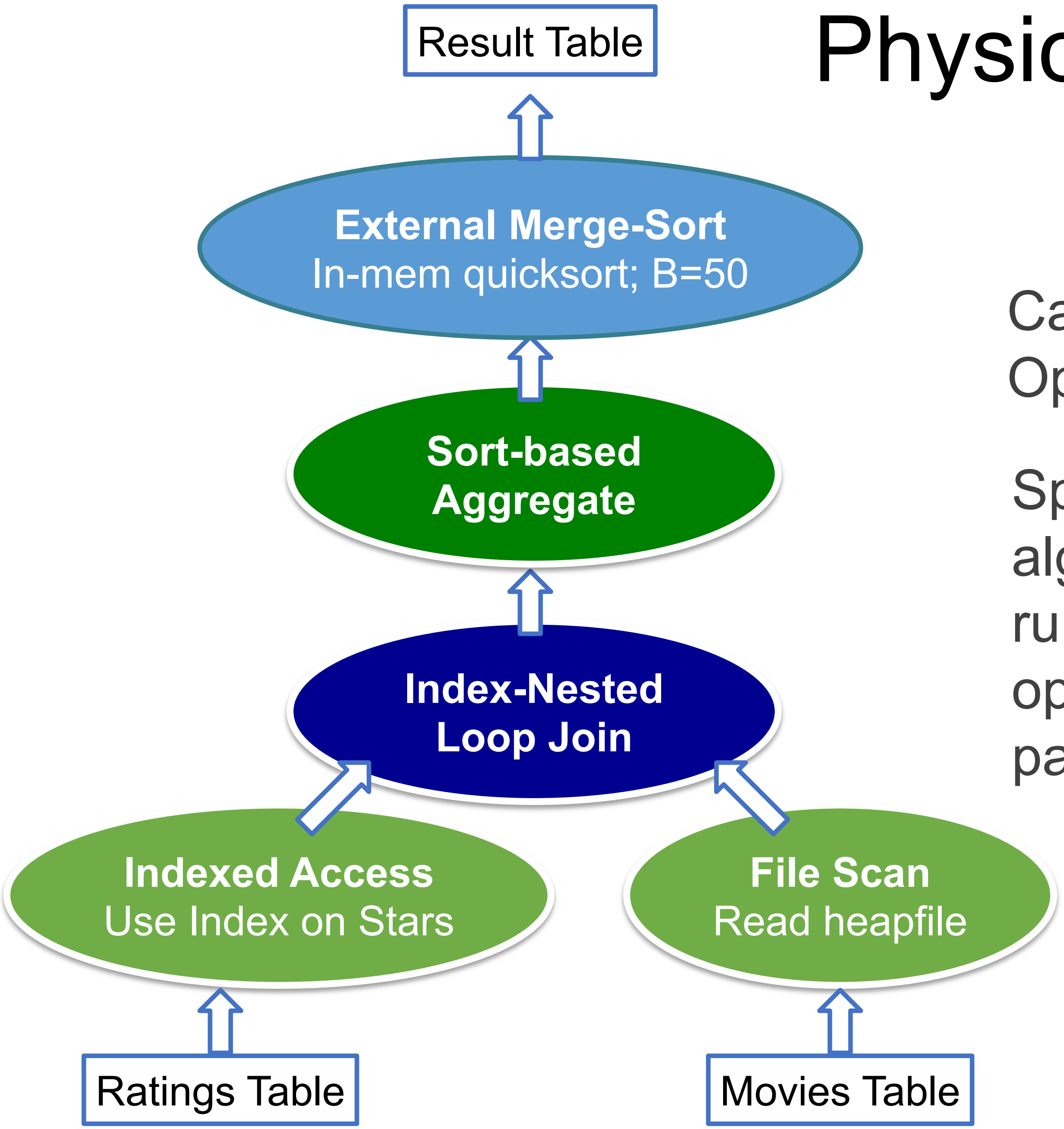


Called “**Logical**” Operators

From extended RA

Each one has alternate “physical” implementations

Physical Query Plan

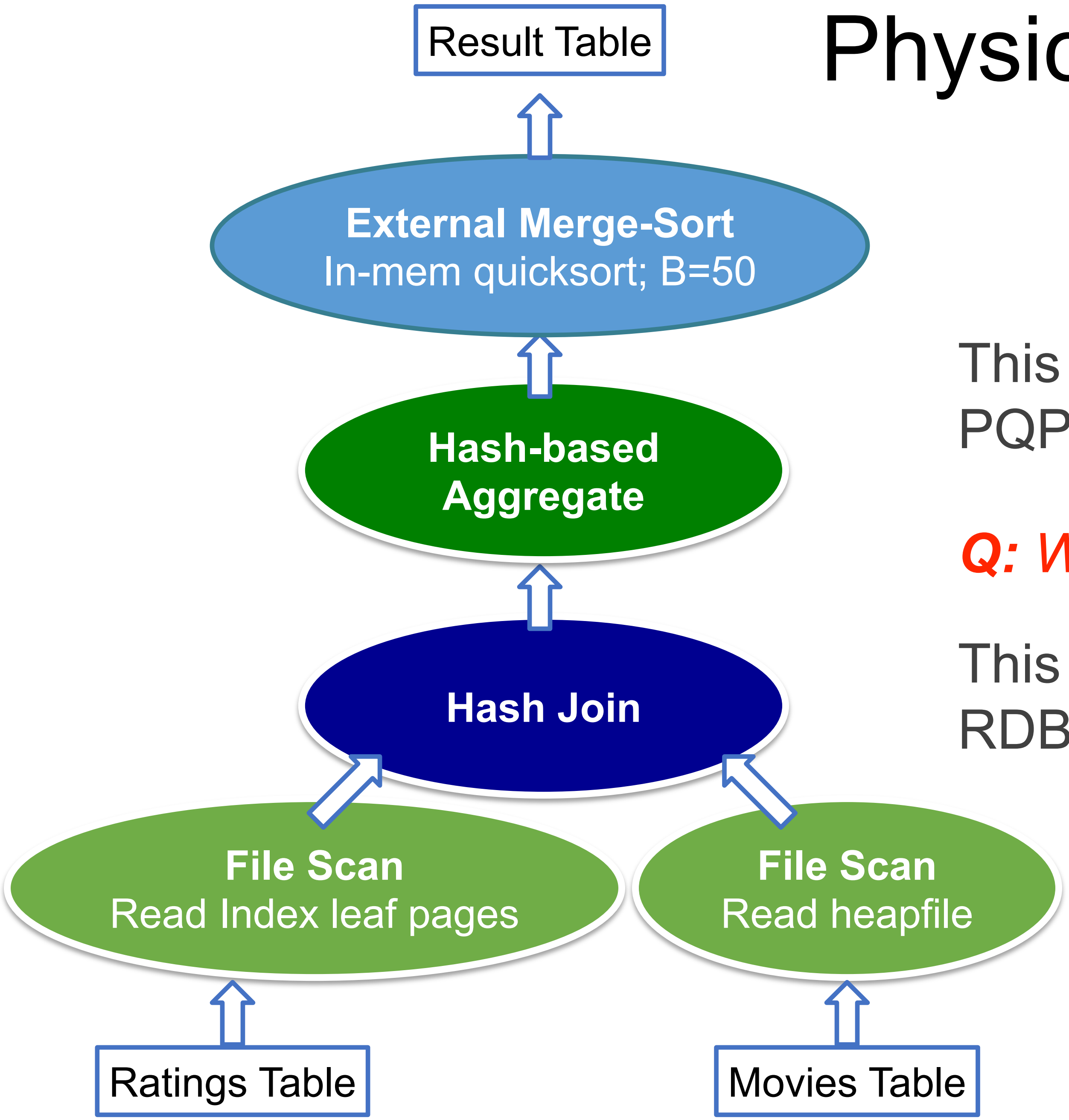


Called "**Physical**" Operators

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

$\boxed{\text{Select}}$

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

R

<u>RatingID</u>	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 <u>matches</u> 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

$\boxed{\textbf{Project}}$

Join

Group By Aggregate

(Optional) Set Operations

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

Project

R	<u>RatingID</u>	Stars	RateDate	UID	MID

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

  Trivial to implement! Read R and <u>discard</u> other attributes

  <u>I/O cost:</u> 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 <u>deduplicate</u> 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: $N_R + N_T + \text{EMSMerge}(N_T)$

- **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 $= N_T / (B-1)$

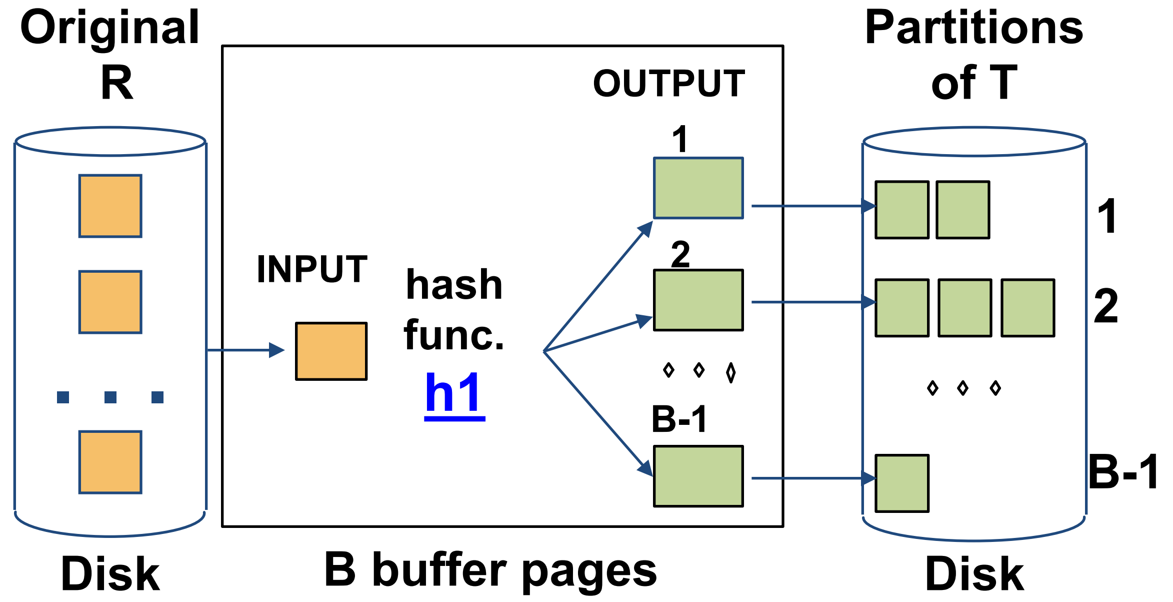
Size of a hash table on a partition $= F \times N_T / (B-1)$

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

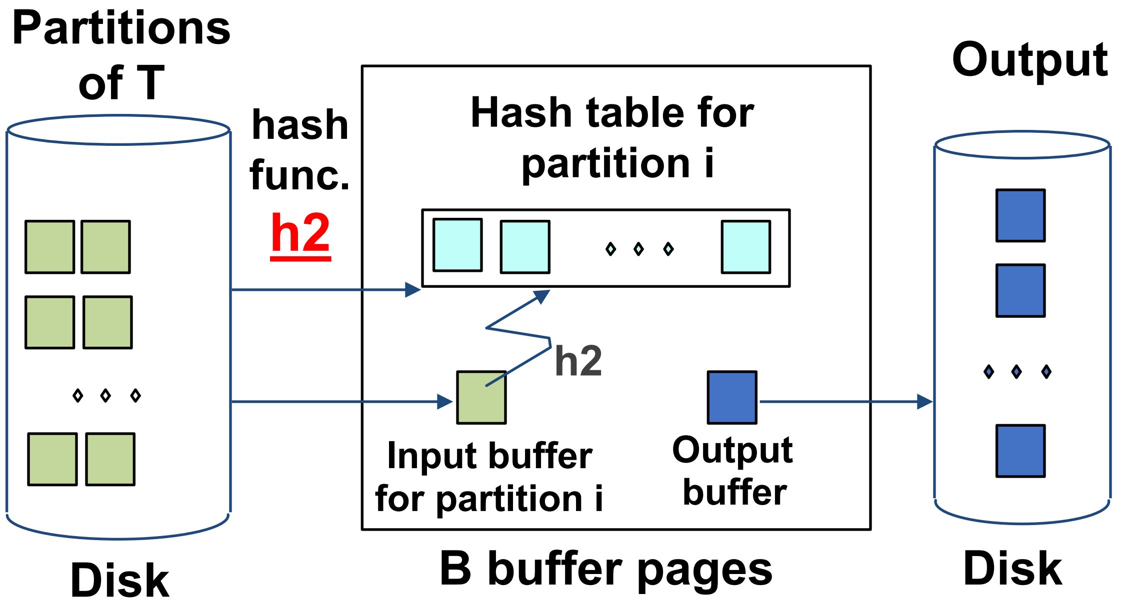
Rough: $B > \sqrt{F \times N_T}$

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 <u>subset</u> of IndexKey

  Use only leaf/bucket pages as the "T" for sorting/hashing
- If we have <u>tree</u> index with ProjectionList as <u>prefix</u> of IndexKey

  Leaf pages are already sorted on ProjectionList (why?)!

Just scan them in order and deduplicate on-the-fly!

Operator Implementations

Select

Project

$\boxed{\textbf{Join}}$

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?*

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

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, pR :
3. Check if any pR 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!)

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

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: $B > \sqrt{N_R}$

$N_U \leq N_R$

Review Questions!

R	<u>RatingID</u>	Stars	RateDate	UID	MID
U	<u>UID</u>	Name	Age	JoinDate	

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 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

$N_U \leq N_R$

U becomes "Inner table"

R is now "Outer table"

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

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) >= F x NU / (B-1)

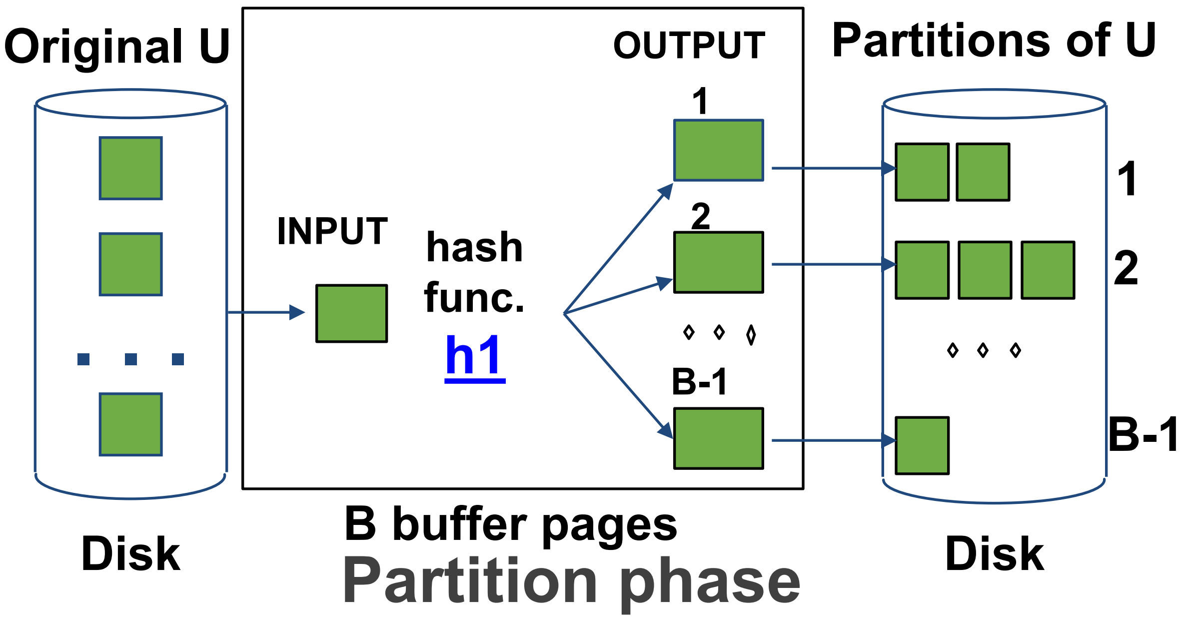
Rough: $B > \sqrt{F \times N_U}$

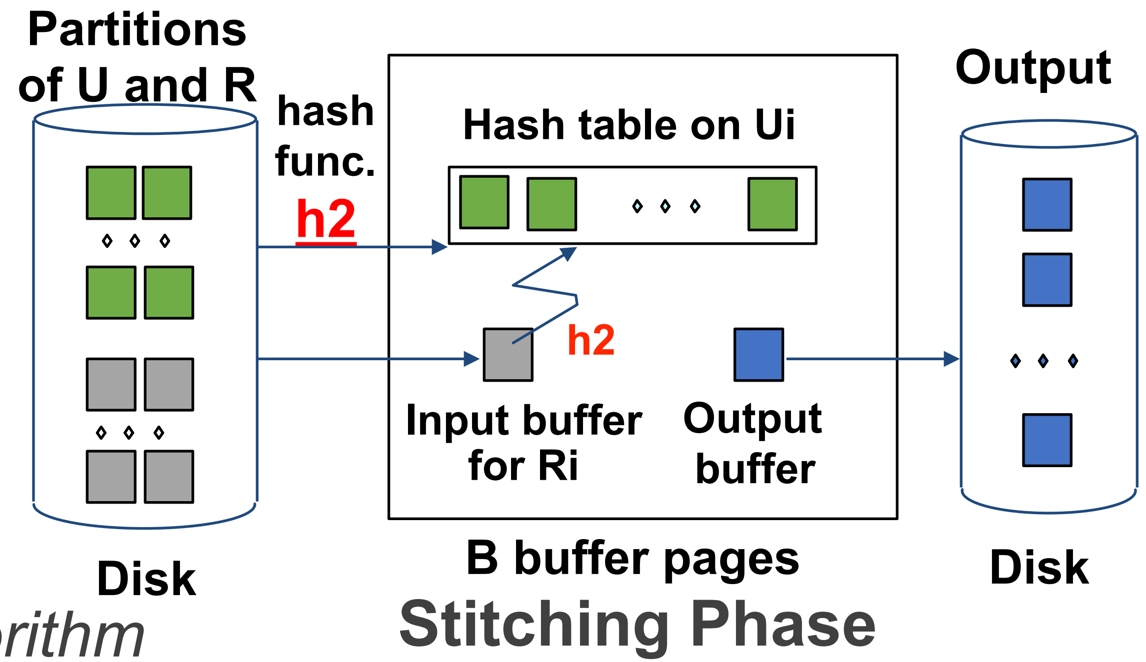
I/O cost: $3 \times (N_U + N_R)$

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

$N_U \leq N_R$

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

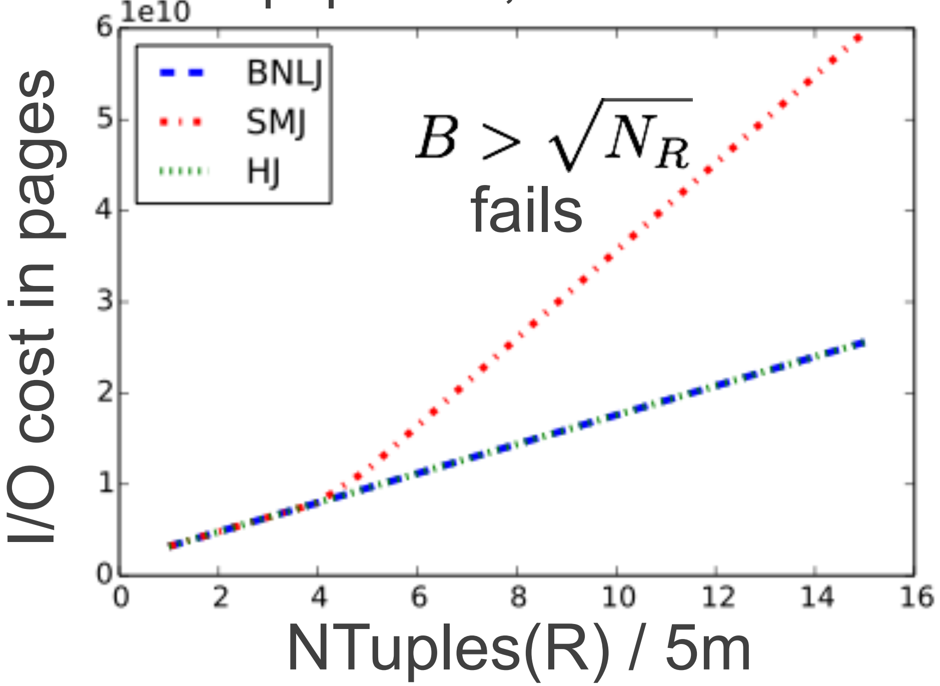
- Query optimizer considers all these when choosing phy. plan

Join: Crossovers of I/O Costs

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

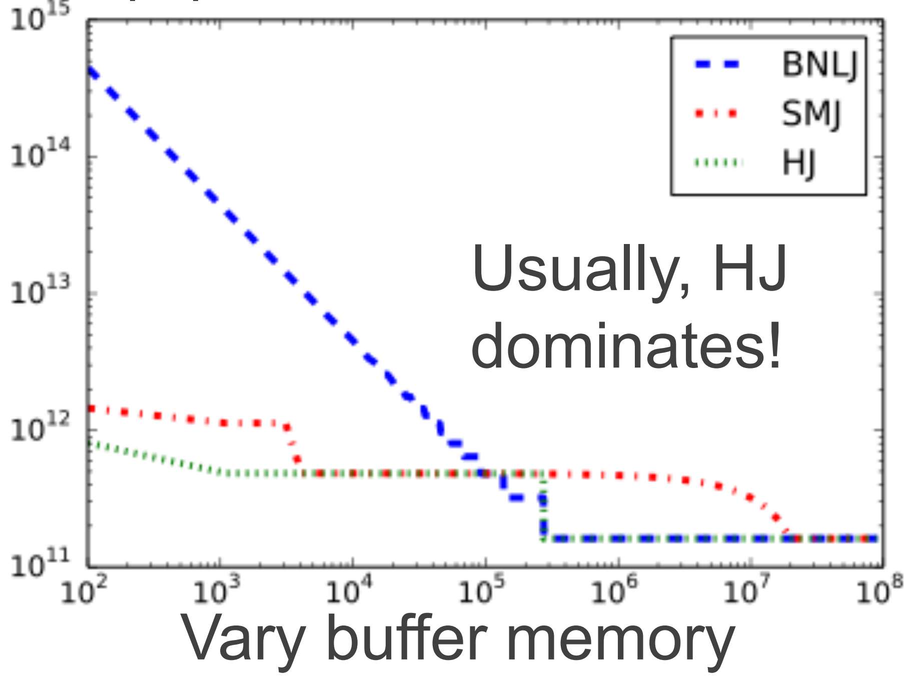
8GB memory; 8KB pages
(So, B = 1024)

|U| = 5m; NU ~ 195K



Arity of both R and U = 40

|U| = 5m; NU ~ 195K
|R| = 500m; NR ~ 19.5M



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

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

"**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

Group By Aggregate: Easy Case

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

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

SUM: Partial sum so far

COUNT is similar

3.0; 8.0; 10.5; 15.5; 18.0; 19, 21.5

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

Group By Aggregate: Difficult Case

- Collect groups of tuples (based on X) and aggregate each

$$\gamma_{MID,AVG(Stars)}(\mathbf{R})$$

21	3	3.0
55	294	5.0
80	12	2.5
21	32	5.0
55	24	2.0
55	19	1.0
21	11	4.0
55	123	4.0

21	123	3.0
21	294	5.0
21	11	4.0

AVG for 21 is 4.0

55	294	5.0
55	24	2.0
55	11	1.0
55	123	4.0

AVG for 55 is 3.0

80	123	2.5

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: $N_R + N_T + \text{EMSMerge}(N_T)$

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?*

Group By Agg.: Hashing-Based

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 U {Y} is a <u>subset</u> of IndexKey:

  Use leaf level of index instead of R for sort/hash algo.!

- Given B+ Tree index s.t. X is a <u>prefix</u> 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?*

Review Questions!

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?

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**: $A \times B$

  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

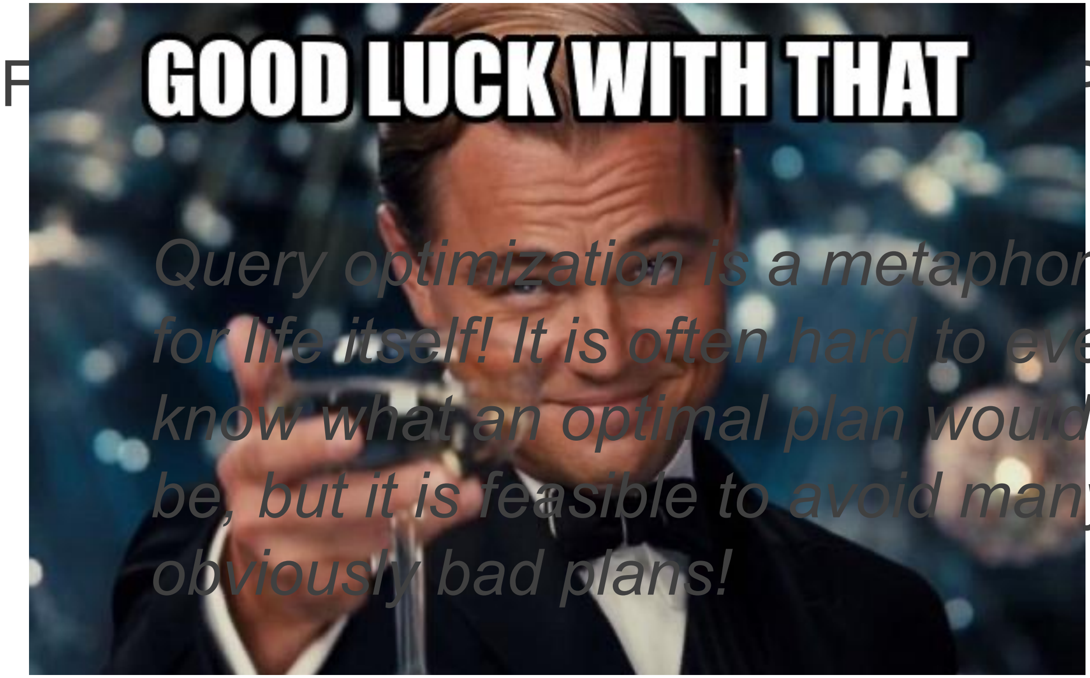
So, what is query optimization and how does it work?

Meet Query Optimization

Basic Idea: A given LQP could have several possible PQPs with very different runtime performance

Goal (Ideal): Get the optimal (fastest) PQP for a given LQP

Goal (Realistic): F Ps!



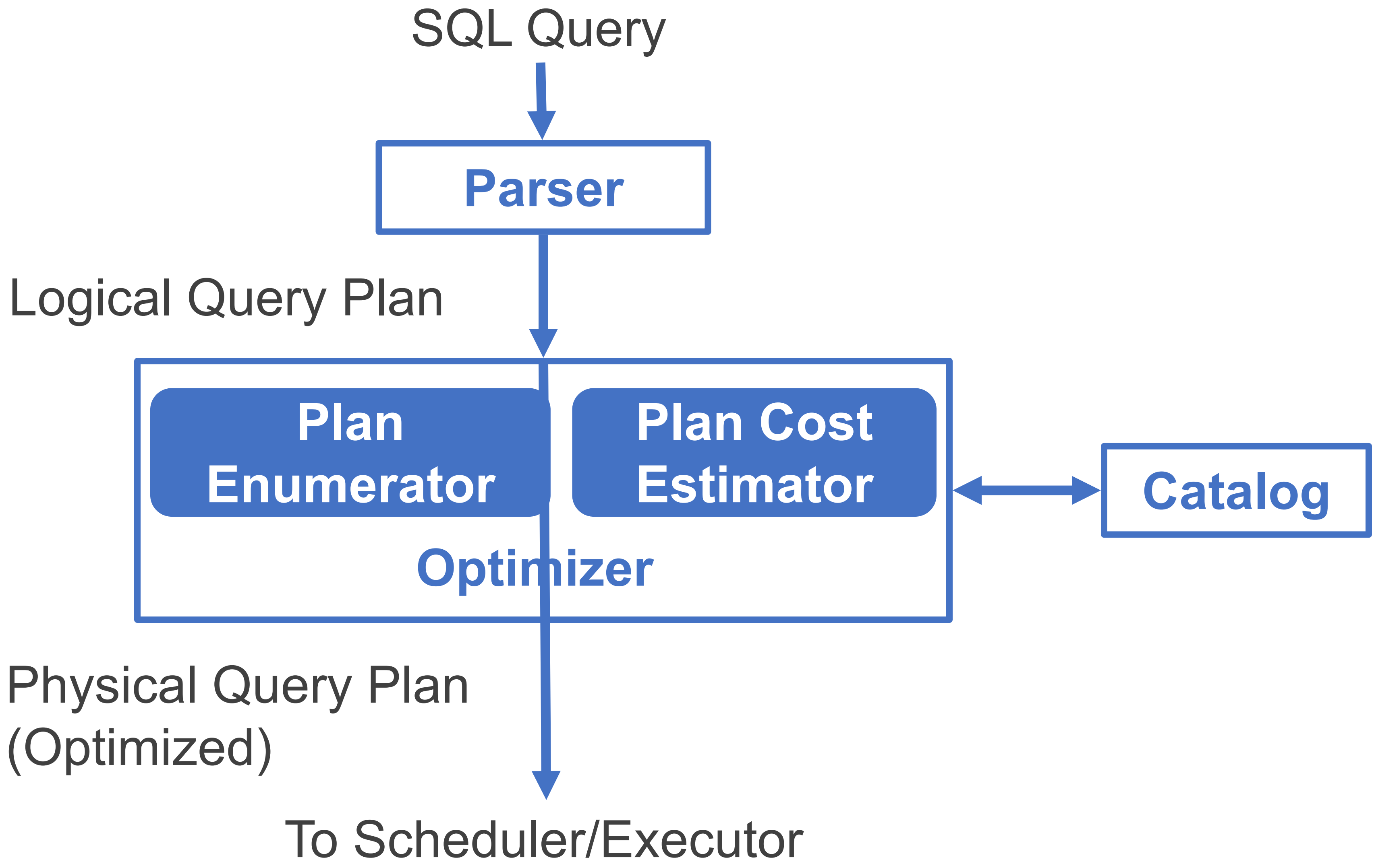

Jeff Naughton

Query optimization is a metaphor for life itself! It is often hard to even know what an optimal plan would be, but it is feasible to avoid many obviously bad plans!

Query Optimization

- Overview of Query Optimizer
- Physical Query Plan (PQP)
  - Concept: Pipelining
  - Mechanism: Iterator Interface
- Enumerating Alternative PQPs
  - Logical: Algebraic Rewrites
  - Physical: Choosing Phy. Op. Impl.
- Costing PQPs
- Materialized Views

Overview of Query Optimizer



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.); Index key
- 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 $\mathbf{N_R}$
- 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)

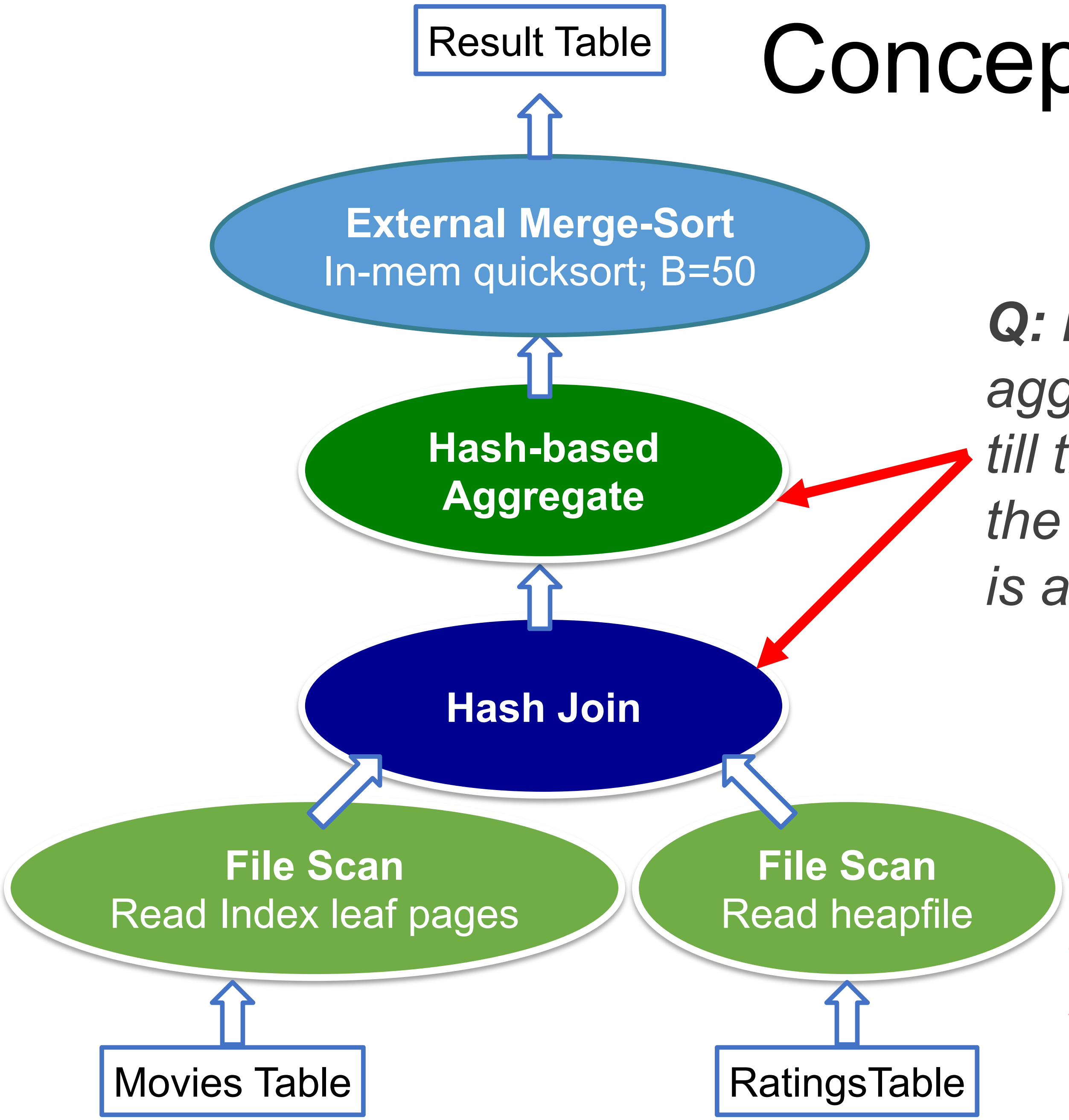
  Height (for tree indexes) **IHeight (X)**

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

Query Optimization

- Overview of Query Optimizer
- Physical Query Plan (PQP)
  - Concept: Pipelining
  - Mechanism: Iterator Interface
- Enumerating Alternative PQPs
  - Logical: Algebraic Rewrites
  - Physical: Choosing Phy. Op. Impl.
- Costing PQPs
- Materialized Views

Concept: Pipelining



***Q:** Does the hash-based aggregate have to wait till the entire output of the “upstream” hash join is available?*

No! We can “**pipeline**” the output of the join – pass on a join output tuple as soon as it is obtained!

Concept: Pipelining

Basic Idea: Do not force "downstream" physical operators to wait till the entire output is available

Benefits: Display output to the user incrementally
<u>CPU Parallelism in multi-core systems!</u>

Tuples

File Scan

Hash Join

Hash-based Aggregate

Concept: Pipelining

- Crucial for PQPs with workflow of many phy. ops.
- Common feature of almost all RDBMSs
- Works for many operators: SCAN, HASH JOIN, etc.

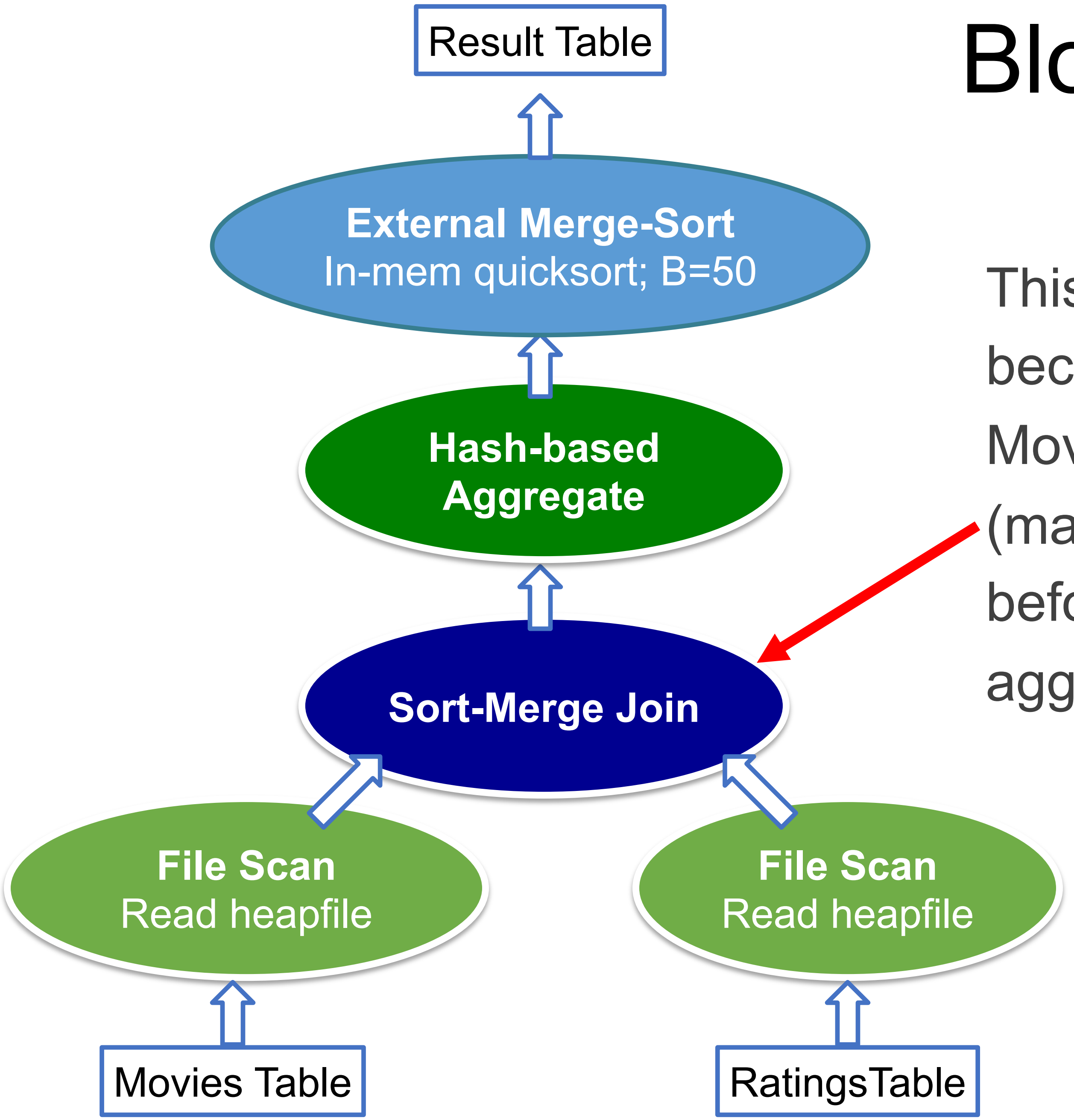
***Q:** Are all physical operators amenable to pipelining?*

No! Some may “stall” the pipeline: “**Blocking Op**”

A blocking op. requires its output to be **Materialized** as a temporary table

Usually, any phy. op. involving <u>sorting</u> is blocking!

Blocking Op



This phy. op. is blocking because we need to sort Movies and sort Ratings (materialize the output) before we can start any aggregate computations!

Query Optimization

- Overview of Query Optimizer
- Physical Query Plan (PQP)
  - Concept: Pipelining
  - Mechanism: Iterator Interface
- Enumerating Alternative PQPs
  - Logical: Algebraic Rewrites
  - Physical: Choosing Phy. Op. Impl.
- Costing PQPs
- Materialized Views

Mechanism: Iterator Interface

- Software API to process PQP; makes pipelining easy to impl.
- Enables us to abstract away individual phy. op. impl. details
- Three main functions in usage interface of each phy. op.:

Open(): **Initialize the phy. op. "state", get arguments**

Allocate input and output buffers

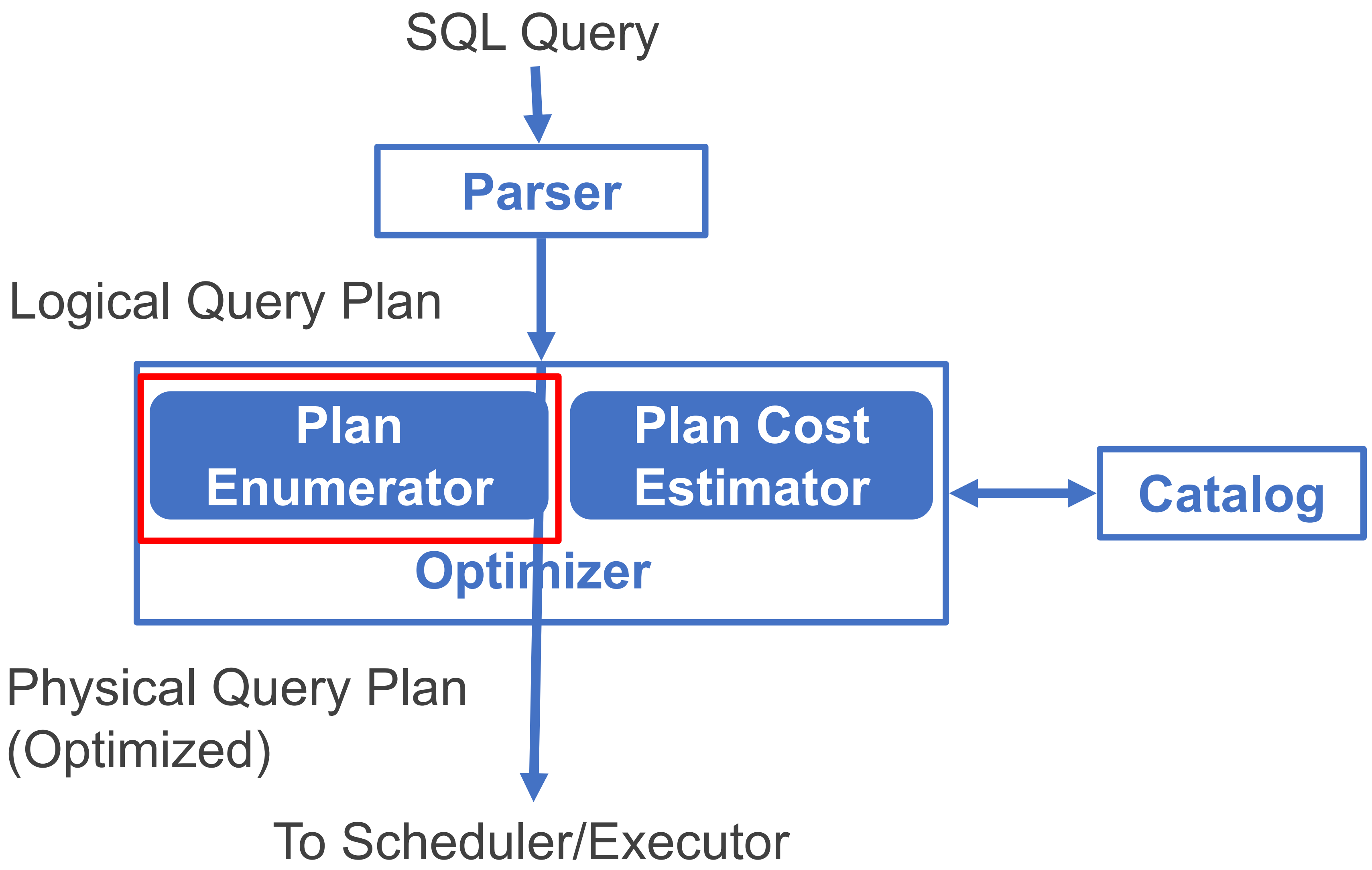
GetNext(): **Ask the phy. op. impl. to "deliver" next output tuple; pass it on; if blocking, wait**

Close(): **Clear phy. op. state, free up space**

Query Optimization

- Overview of Query Optimizer
- Physical Query Plan (PQP)
  - Concept: Pipelining
  - Mechanism: Iterator Interface
- Enumerating Alternative PQPs
  - Logical: Algebraic Rewrites
  - Physical: Choosing Phy. Op. Impl.
- Costing PQPs
- Materialized Views

Overview of Query Optimizer



Enumerating Alternative PQPs

- Plan Enumerator explores various PQPs for a given LQP
- **Challenge: Space of plans is huge! How to make it feasible?**
- RDBMS Plan Enumerator has **Rules** to help determine what plans to enumerate, and also consults **Cost models**
- Two main sources of Rules for enumerating plans:

    Logical: Algebraic Rewrites:

    Use relational algebra <u>equivalence</u> to rewrite LQP itself!

    Physical: Choosing Phy. Op. Impl.:

    Use different phy. op. impl. for a given log. op. in LQP

Query Optimization

- Overview of Query Optimizer
- Physical Query Plan (PQP)
  - Concept: Pipelining
  - Mechanism: Iterator Interface
- Enumerating Alternative PQPs
  - Logical: Algebraic Rewrites
  - Physical: Choosing Phy. Op. Impl.
- Costing PQPs
- Materialized Views

Algebraic Rewrite Rules

- Rewrite a given RA query in to another that is <u>equivalent</u> (a logical property) but might be <u>faster</u> (a physical property)
- RA operators have some formal properties we can exploit
- We will cover only a few rewrite rules:

  Single-operator Rewrites

    Unary operators

    Binary operators

  Cross-operator Rewrites

Unary Operator Rewrites

- Key unary operators in RA: $\sigma \;\; \pi$
- Commutativity of σ

$$\sigma_{p_1}(\sigma_{p_2}(\mathbf{R})) = \sigma_{p_2}(\sigma_{p_1}(\mathbf{R}))$$

- Cascading of σ

$$\sigma_{p_1}(\sigma_{p_2}(\ldots\sigma_{p_n}(\mathbf{R})\ldots)) = \sigma_{p_1 \wedge p_2 \wedge \cdots \wedge p_n}(\mathbf{R})$$

- Cascading of π $\qquad A_i \subseteq A_{i+1} \forall i = 1 \ldots (n-1)$

$$\pi_{A_1}(\pi_{A_2}(\ldots\pi_{A_n}(\mathbf{R})\ldots)) = \pi_{A_1}(\mathbf{R})$$

***Q:** Why are cascading rewrites beneficial?*

Binary Operator Rewrites

- Key binary operator in RA: $\bowtie$
- Commutativity of $\bowtie$ $$R \bowtie S = S \bowtie R$$
- Associativity of $\bowtie$ $$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Q: *Why are these properties beneficial?*

Q: *What other binary operators in RA satisfy these?*

Cross-Operator Rewrites

- Commuting σ and π

$$A \subseteq B$$

$$\sigma_{p(A)}(\pi_B(R)) = \pi_B(\sigma_{p(A)}(R))$$

- Combining σ and $\times$

$$\sigma_p(R \times S) = R \bowtie_p S$$

- "Pushing the select"

$$A \subseteq R.*$$

$$\sigma_{p(A)}(R \bowtie S) = \sigma_{p(A)}(R) \bowtie S$$

$$\sigma_{p(A)}(R \times S) = \sigma_{p(A)}(R) \times S$$

- Commuting π with $\times$ and $\bowtie$

$$\pi_A(R \times S) = \pi_{A \cap R.*}(R) \times \pi_{A \cap S.*}(S) \qquad B \subseteq A$$

$$\pi_A(R \bowtie_{p(B)} S) = \pi_{A \cap R.*}(R) \bowtie_{p(B)} \pi_{A \cap S.*}(S)$$

Review Question

- **Which of the following hold?**

$$A \subseteq R$$

$$\pi_A(R \times S) = \pi_A(R) \times S$$

$$C = A \cup B$$

$$\pi_A\left(R \bowtie_{p(B)} S\right) = \pi_A(\pi_{C\cap R}(R) \bowtie_{p(B)} \pi_{C\cap S}(S))$$

$$\sigma_{p_1 \wedge p_2 \vee p_3}(R) = \sigma_{p_1}(R) \cap \sigma_{p_2}(R) \cup \sigma_{p_3}(R)$$

$A \subseteq R$ and $B \subseteq S$

$$\sigma_{p(A) \wedge q(B)}(R \bowtie S) = \sigma_{p(A)}(R) \bowtie \sigma_{q(B)}(S)$$

Query Optimization

- Overview of Query Optimizer
- Physical Query Plan (PQP)
  - Concept: Pipelining
  - Mechanism: Iterator Interface
- Enumerating Alternative PQPs
  - Logical: Algebraic Rewrites
  - $\boxed{\text{Physical: Choosing Phy. Op. Impl.}}$
- Costing PQPs
- Materialized Views

Choosing Phy. Op. Impl.

- Given a (rewritten) LQP, pick phy. op. impl. for each log. op.
- Recall various RA op. impl. with their I/O (and CPU costs)

σ	File scan vs Indexed (B+ Tree vs Hash)
π	Hashing-based vs Sorting-based vs Indexed
$\bowtie$	BNLJ vs INLJ vs SMJ vs HJ
	etc.

$$\pi_B(\sigma_{p(A)}(R) \bowtie S)$$

3 options   3 options   4 options   = **36** PQPs!

Q: *With algebraic rewrites?!*

Phy. Op. Impl.: Other Factors

- Are the indexes clustered or unclustered?
- Are there multiple matching indexes? Use multiple?
- Are index-only access paths possible for some ops?
- Are there “interesting orderings” among the inputs?
- Would sorted outputs benefit downstream ops?
- Estimation of <u>cardinality</u> of intermediate results!
- How best to reorder multi-table joins?

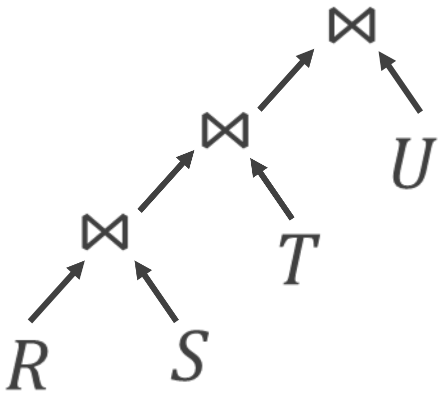
Query optimizers are complex beasts!

Still a hard, open research problem!

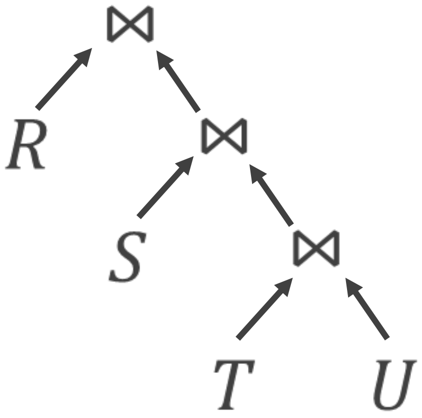
Phy. Op. Impl.: Join Orderings

- Since joins are associative, exponential number of orderings!

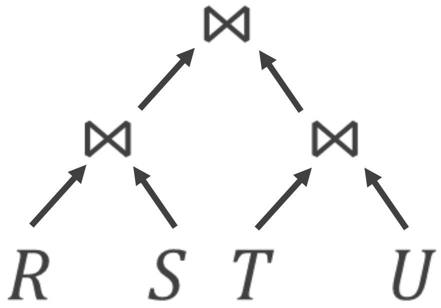
$$R \bowtie S \bowtie T \bowtie U$$



Left Deep tree



Right Deep tree



"Bushy" tree

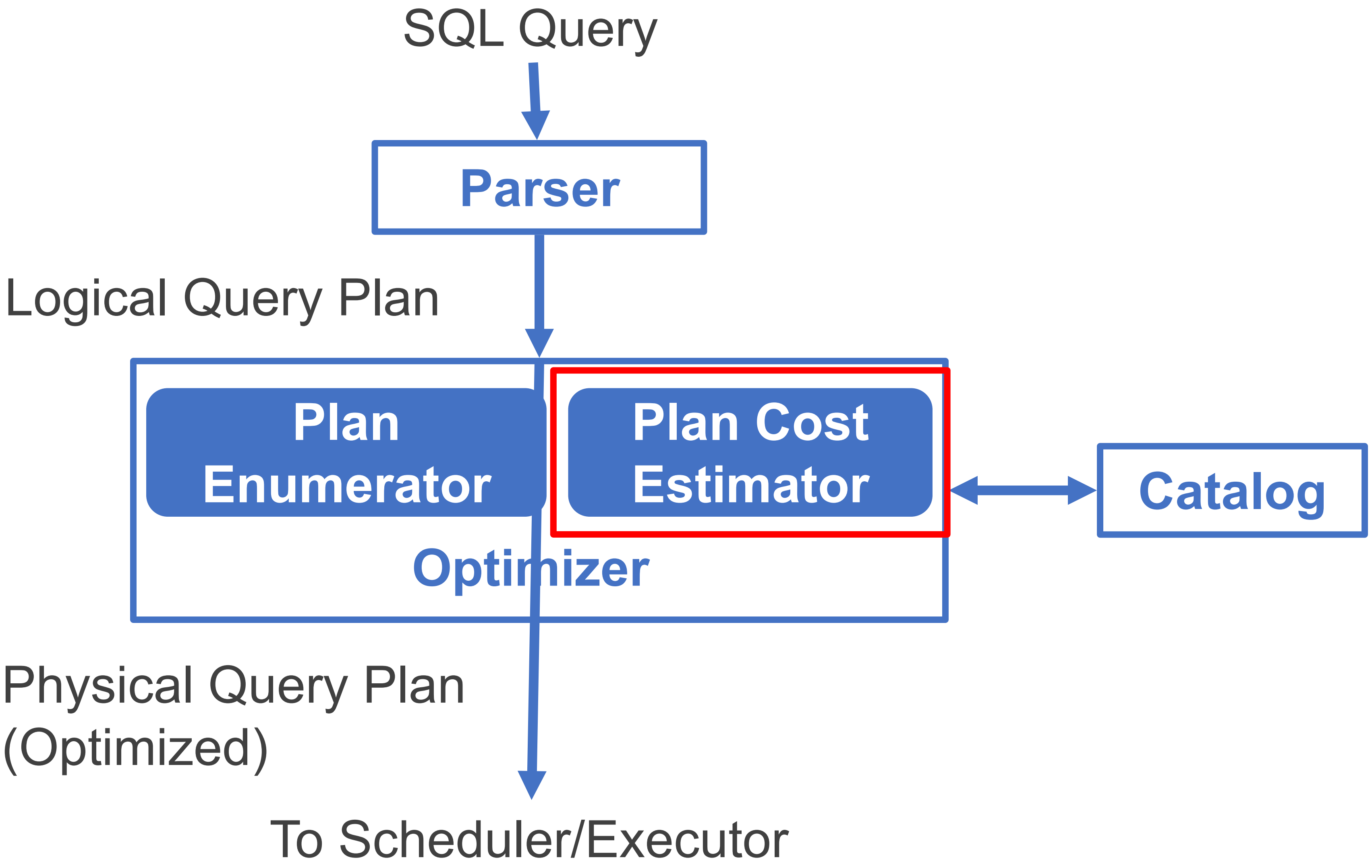
- Almost all RDBMSs consider only left deep join trees
  - Enables easy pipelining! Why?
- "Interesting orderings" idea from System R optimizer paper
- Dynamic program to combine enumeration and costing

"Access Path Selection in a Relational Database Management System" SIGMOD'79

Query Optimization

- Overview of Query Optimizer
- Physical Query Plan (PQP)
  - Concept: Pipelining
  - Mechanism: Iterator Interface
- Enumerating Alternative PQPs
  - Logical: Algebraic Rewrites
  - Physical: Choosing Phy. Op. Impl.
- $\boxed{\text{Costing PQPs}}$
- Materialized Views

Overview of Query Optimizer



Costing PQPs

- For each PQP considered by the Plan Enumerator, the Plan Cost Estimator computes "**Cost**" of the PQP
  - Weighted sum of I/O cost and CPU cost
  - (Distributed RDBMSs also include Network cost)
- **Challenge: Given a PQP, compute overall cost**
- **Issues to consider:**
  - Pipelining vs. blocking ops; cannot simply add costs!
  - **Cardinality estimation** for intermediate tables!
  - ***Q:*** *What statistics does the catalog have to help?*

Costing PQPs

- Most RDBMSs use various heuristics to make costing tractable; so, it is approximate!
- **Example: Complex predicates**

$$\sigma_{p_1 \wedge p_2}(R)$$

Suppose selectivity of p_1 is 5%
and selectivity of p_2 is 10%

Q. What is the selectivity of $p_1 \wedge p_2$*?*

Not enough info!

But, most RDBMSs use the **independence** heuristic!

Selectivity of conjunction = Product of selectivities

Thus, ≈ 0.05 * 0.1 = 0.005, i.e., 0.5%

Query Optimization: Summary

- Plan Enumerator and Cost Estimator work in lock step:
  - **Rules** determine what PQPs are enumerated
    - Logical: Algebraic rewrites of LQP
    - Physical: Op. Impl. and ordering alternatives
  - **Cost models** and **heuristics** help cost the PQPs
- Still an active research area!
  - Parametric Q.O., Multi-objective Q.O.,
  - Multi-objective parametric Q.O., Multiple Q.O.,
  - Online/Adaptive Q.O., Dynamic re-optimization, etc.

Review Question

<u>RatingID</u>	Stars	RateDate	UID	MID

10m pages

Page size 8KB; Buffer memory 4GB; 8B for each field

```
SELECT COUNT(DISTINCT UID) FROM Ratings
```

Propose an efficient physical plan and compute its I/O cost.

***Q:** What if there was an unclustered B+ tree index on UID?*

(RecordID pointers can be assumed to be 8B too)

Review Question

<u>RatingID</u>	Stars	RateDate	UID	MID	10m pages

<u>MID</u>	Name	Year	Director	100k pages

Page size 8KB; Buffer memory 4GB

```
SELECT AVG(Stars) FROM Ratings R, Movies M
WHERE  R.MID = M.MID AND
       M.Director = "Christopher Nolan" AND
       R.UID = 1234;
```

Propose an efficient physical plan that does not materialize any intermediate data (fully pipelined) and compute its I/O cost.

Query Optimization

- Overview of Query Optimizer
- Physical Query Plan (PQP)
  - Concept: Pipelining
  - Mechanism: Iterator Interface
- Enumerating Alternative PQPs
  - Logical: Algebraic Rewrites
  - Physical: Choosing Phy. Op. Impl.
- Costing PQPs
- $\boxed{\text{Materialized Views}}$

Introducing Materialized Views

- A **View** is a "virtual table" created with an SQL query
- A **Materialized View** is a physically instantiated/stored view

Example:

<u>RatingID</u>	Stars	RateDate	UID	MID

<u>UID</u>	Name	Age	JoinDate	<u>MID</u>	Name	Year	Director

```
SELECT AVG(Stars)
FROM   Ratings R, Movies M, Users U
WHERE  R.MID = M.MID AND R.UID = U.UID
       M.Director = "Christopher Nolan" AND
       U.Age >= 20 AND U.Age < 30;
```

$$\gamma_{AVG(Stars)}(R \bowtie \sigma_{Director=\text{"Christopher Nolan"}}(M) \bowtie \sigma_{20 \leq Age < 30}(U))$$

Requires file scans of R, M, and U and, say, hash joins

Materialized Views Example

Example:

RatingID	Stars	RateDate	UID	MID

UID	Name	Age	JoinDate	MID	Name	Year	Director

$$\gamma_{AVG(Stars)}(R \bowtie \sigma_{Director=\text{“}Christopher\ \ Nolan\text{”}}(M) \bowtie \sigma_{20 \leq Age < 30}(U))$$

```
CREATE MATERIALIZED VIEW NolanRatings AS
SELECT RatingID, Stars, UID, MID
FROM   Ratings R, Movies M
WHERE  R.MID = M.MID AND
       M.Director = "Christopher Nolan";
```

Creates a subset of R with ratings for only Nolan's movies

$$V \leftarrow \pi_{RatingID, Stars, UID, MID}(R \bowtie \sigma_{Director=\text{“}Christopher\ \ Nolan\text{”}}(M))$$

Materialized Views Example

Example:

<u>RatingID</u>	Stars	RateDate	UID	MID

<u>UID</u>	Name	Age	JoinDate	<u>MID</u>	Name	Year	Director

$$\gamma_{AVG(Stars)}(R \bowtie \sigma_{Director=\text{“}Christopher\ \ Nolan\text{”}}(M) \bowtie \sigma_{20 \leq Age < 30}(U))$$

Given the materialized view V, RDBMS optimizer can automatically *rewrite* to use V to avoid scans of R and M

$$V \leftarrow \pi_{RatingID,Stars,UID,MID}(R \bowtie \sigma_{Director=\text{“}Christopher\ \ Nolan\text{”}}(M))$$

$$\gamma_{AVG(Stars)}(V \bowtie \sigma_{20 \leq Age < 30}(U))$$

Likely much faster since V is likely much smaller than R, but this depends on data statistics; leave it to optimizer!

***Q:** How did DBA know to materialize a view for Nolan ratings?*

Materialized View Maintenance

Example:

<u>RatingID</u>	Stars	RateDate	UID	MID

<u>UID</u>	Name	Age	JoinDate

<u>MID</u>	Name	Year	Director

We are given this materialized view V over R and M

$$V \leftarrow \pi_{RatingID,Stars,UID,MID}(R \bowtie \sigma_{Director=\text{``}Christopher\ \ Nolan\text{''}}(M))$$

Q: *What if new ratings are inserted to R for Nolan's movies?*

- RDBMS will automatically "trigger" updates to V
- Such updates are called **Materialized View Maintenance**
- 2 alternatives: Recompute whole view from scratch vs **Incremental View Maintenance** (IVM)

Incremental View Maintenance (IVM)

Basic Idea: Recomputing V from scratch may be an overkill
Try to *incrementally* update parts that change

$$V = Q(D) \qquad V' = Q(D')$$

- D' can be the outcome of inserts and/or deletes to D
- Q can be a unary query or involve multiple tables
- Computing V' may require inserts and/or deletes to V; realized as ***algebraic rewrite rules*** at LQP level
- Whether or not IVM of V is feasible and/or efficient depends on form of Q, nature of updates to D, data statistics, etc.
- We will focus only on inserts to D triggering inserts to V

Incremental View Maintenance (IVM)

Unary IVM for insertions:

$$R' = R \cup \Delta R$$ ← Newly inserted tuples

Select:

$$V \leftarrow \sigma_{SelectCondition}(R)$$
$$V' = V \cup \sigma_{SelectCondition}(\Delta R)$$

Can be just an *append* (union with “bag” semantics)

Project:

$$V \leftarrow \pi_{ProjectionList}(R)$$
$$V' = V \cup \pi_{ProjectionList}(\Delta R)$$

Requires full set union with V for deduplication

Select and Project can be composed and reordered as before

Incremental View Maintenance (IVM)

Unary IVM for insertions:

$$R' = R \cup \Delta R$$ ← Newly inserted tuples

Group By Agg: $V \leftarrow \gamma_{AggList, Agg(Y)}(R)$

Feasibility of IVM Depends on Agg() function!

Rewrite rules exist for SUM, COUNT, and MIN/MAX over bags

AVG not possible in general; needs deeper system changes

$$V' = \gamma_{AggList, SUM(Y)}(V \cup \gamma_{AggList, SUM(Y)} \Delta R)$$

$$V' = \gamma_{AggList, SUM(Y)}(V \cup \gamma_{AggList, COUNT(Y)} \Delta R)$$

$$V' = \gamma_{AggList, MIN(Y)}(V \cup \gamma_{AggList, MIN(Y)} \Delta R)$$

Incremental View Maintenance (IVM)

Join IVM for insertions:

$$V \leftarrow A \bowtie B$$

Assume no duplicate inserts

$$A' = A \cup \Delta A$$
$$B' = B \cup \Delta B$$

$$V' = V \cup (\Delta A \bowtie B') \cup (A' \bowtie \Delta B)$$

Alternatively, we can just append the output of the following query to V (union below is just append too):

$$(\Delta A \bowtie B') \cup (A' \bowtie \Delta B) - (\Delta A \bowtie \Delta B)$$

IVM for complex queries compose such op-level rewrites

Query Optimization

- Overview of Query Optimizer
- Physical Query Plan (PQP)
  - Concept: Pipelining
  - Mechanism: Iterator Interface
- Enumerating Alternative PQPs
  - Logical: Algebraic Rewrites
  - Physical: Choosing Phy. Op. Impl.
- Costing PQPs
- Materialized Views