CS 6530: Advanced Database Systems Fall 2024

Lecture 16
Query optimization

Prashant Pandey

prashant.pandey@utah.edu

Acknowledgement: Slides taken from Prof. Arun Kumar, UCSD

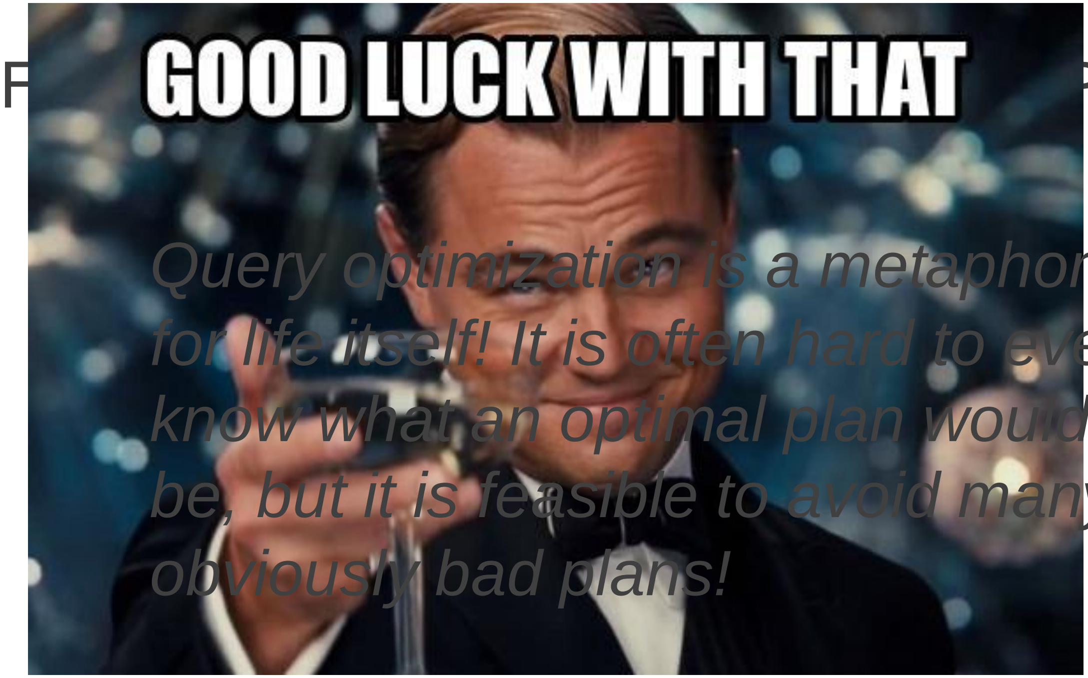
So, what is query optimization and how does it work?

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!



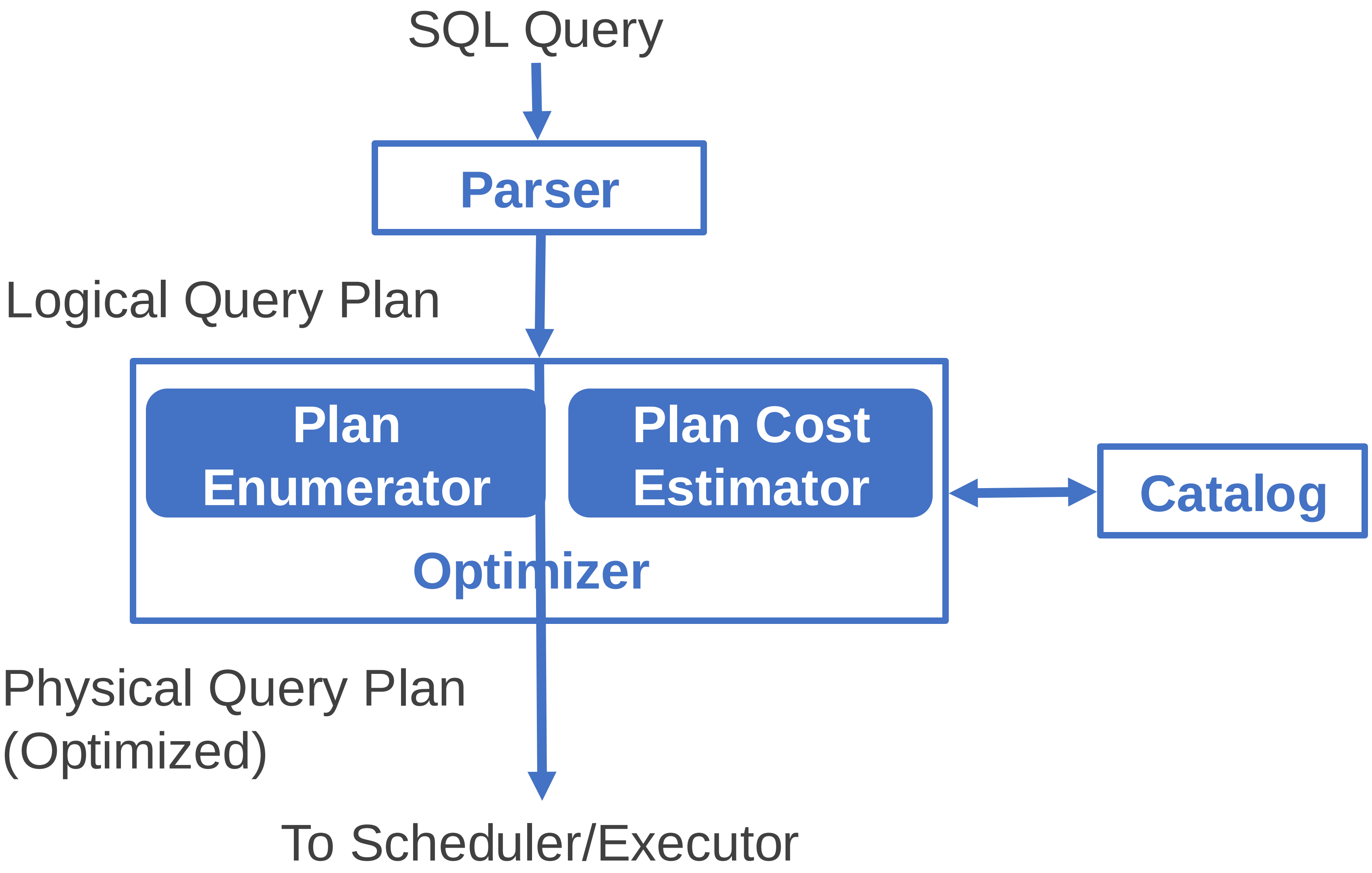
Query optimization is a metaphor for life itself! It is often hard to even know what an optimal plan would be, but it is feasible to avoid many obviously bad plans!

Jeff Naughton

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

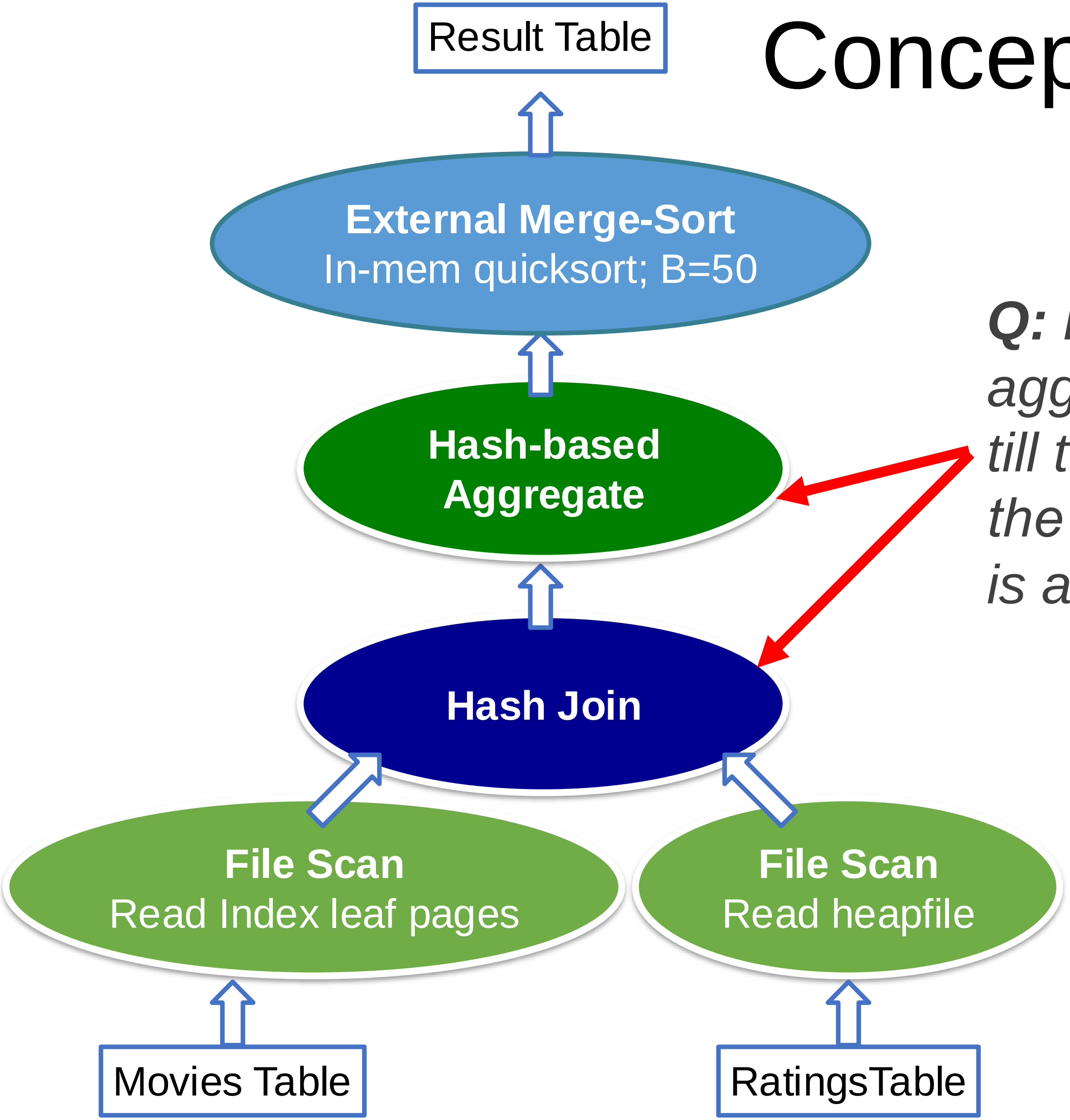
Statistics in the System Catalog

- 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>

File Scan

Hash Join

Hash-based Aggregate

Tuples

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.

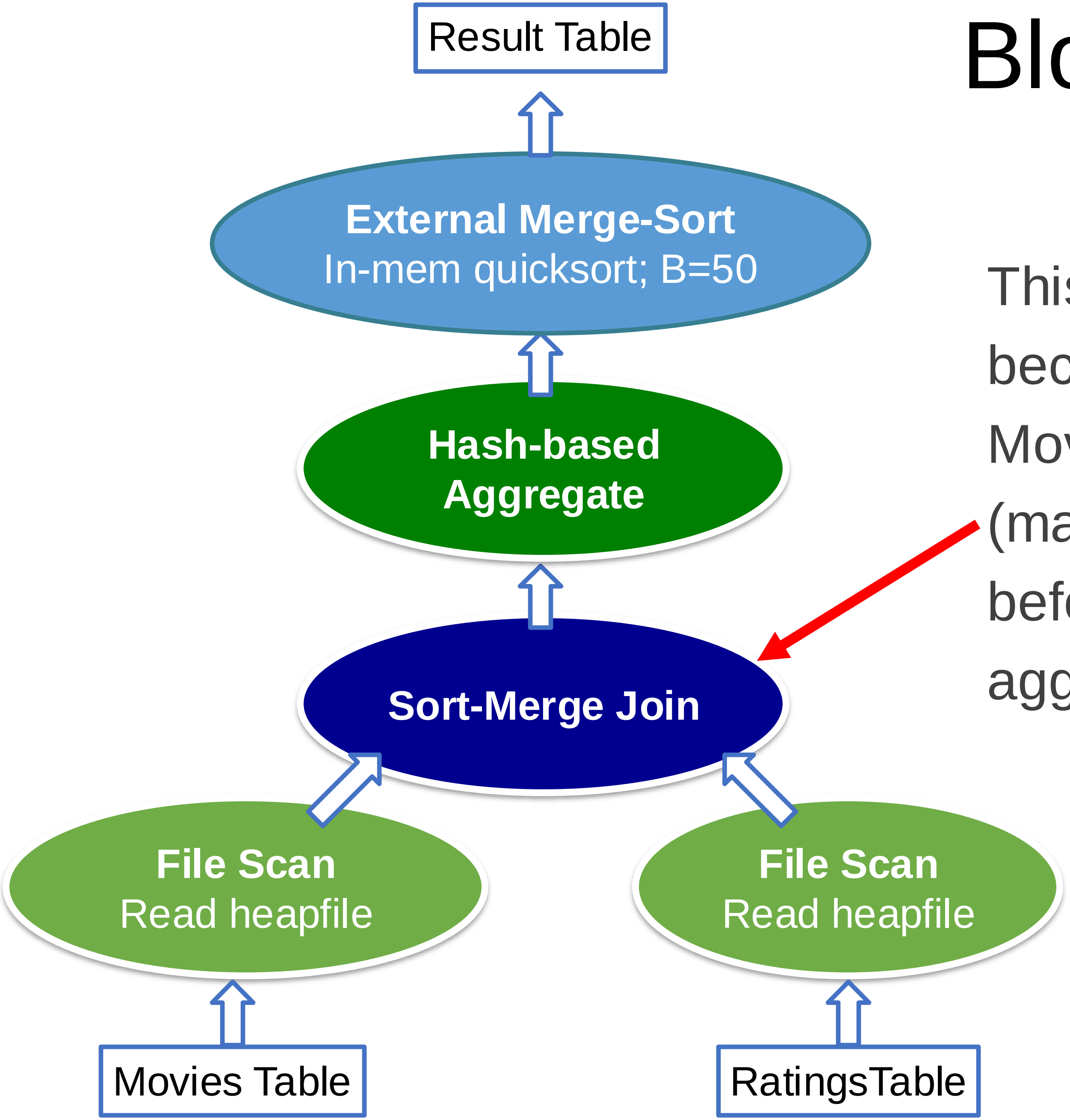
***Q:** Are all physical operators amenable to pipelining?*

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
- Enumerat 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

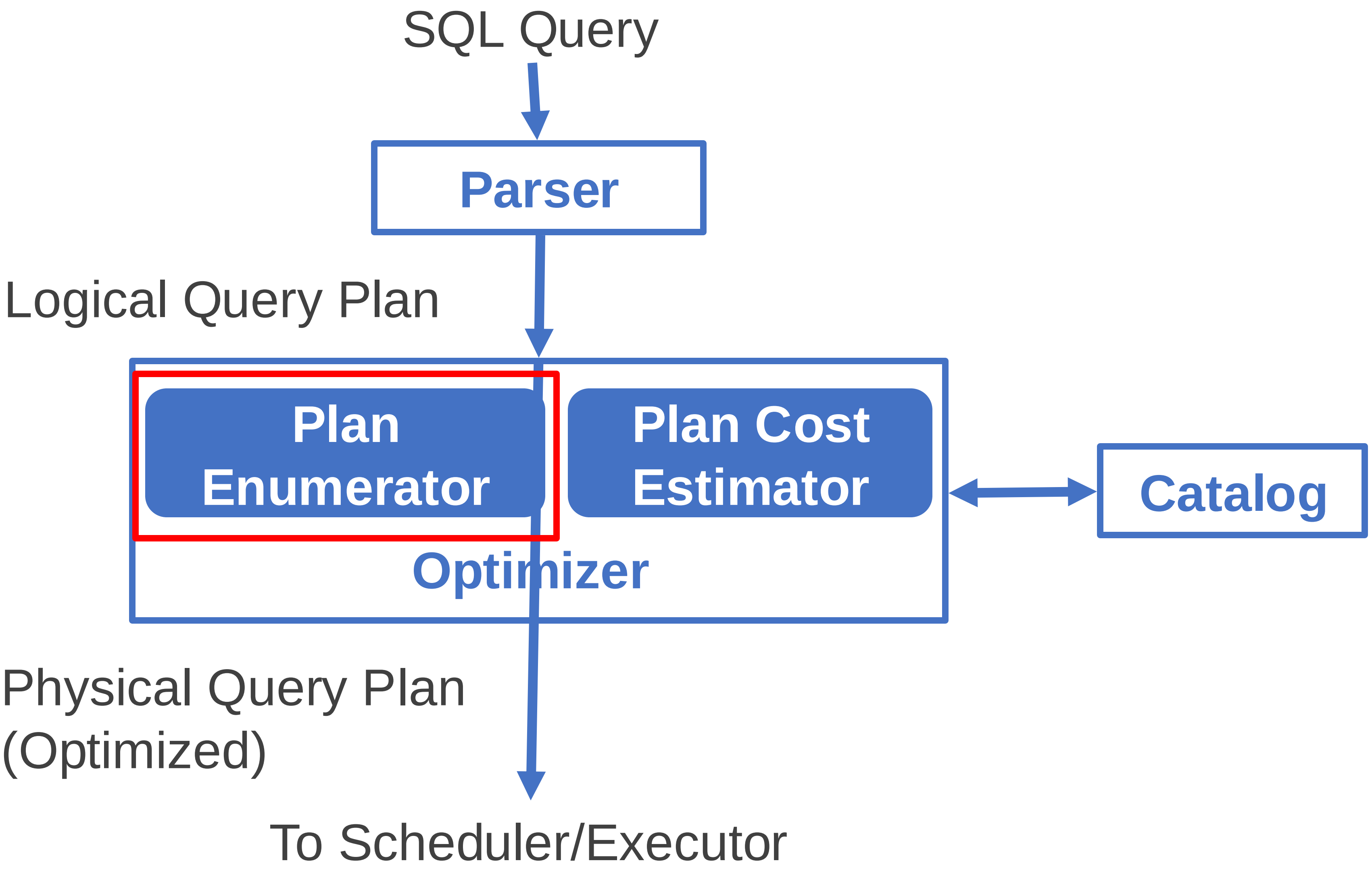
GetNext(): **Ask the phy. op. impl. to “deliver” next**
output tuple; pass it on; if blocking, wait

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



SQL Query

Parser

Logical Query Plan

Plan Enumerator

Plan Cost Estimator

Optimizer

Catalog

Physical Query Plan
(Optimized)

To Scheduler/Executor

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 equivalent (a logical property) but might be faster (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

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

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

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 $\bowtie$

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

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

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

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.

$$\underbrace{\pi_B}_{\text{3 options}}(\underbrace{\sigma_{p(A)}}_{\text{3 options}}(R) \underbrace{\bowtie}_{\text{4 options}} S)$$

= **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 cardinality of intermediate results!
- How best to reorder multi-table joins?

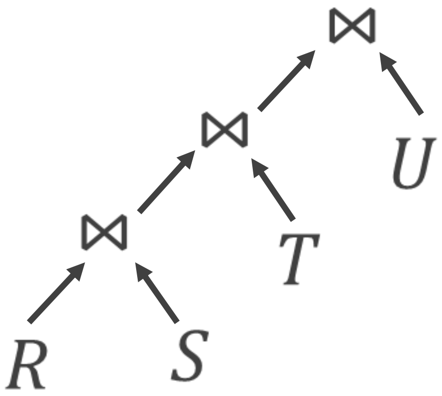
Query optimizers are complex beasts!

Still a hard, open research problem!

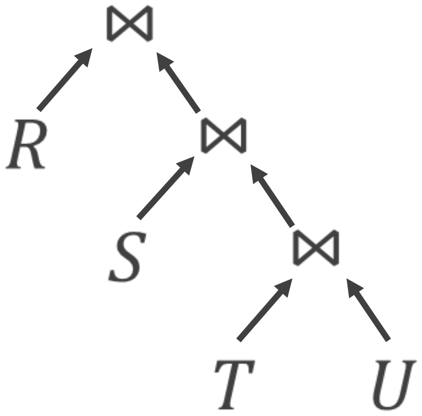
Phy. Op. Impl.: Join Orderings

- Since joins are associative, exponential number of orderings!

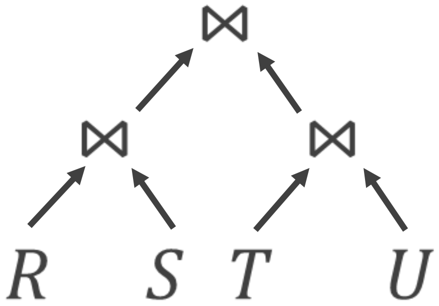
$$R \bowtie S \bowtie T \bowtie U$$



Left Deep tree



Right Deep tree



“Bushy” tree

- Almost all RDBMSs consider only left deep join trees

  Enables easy pipelining! Why?

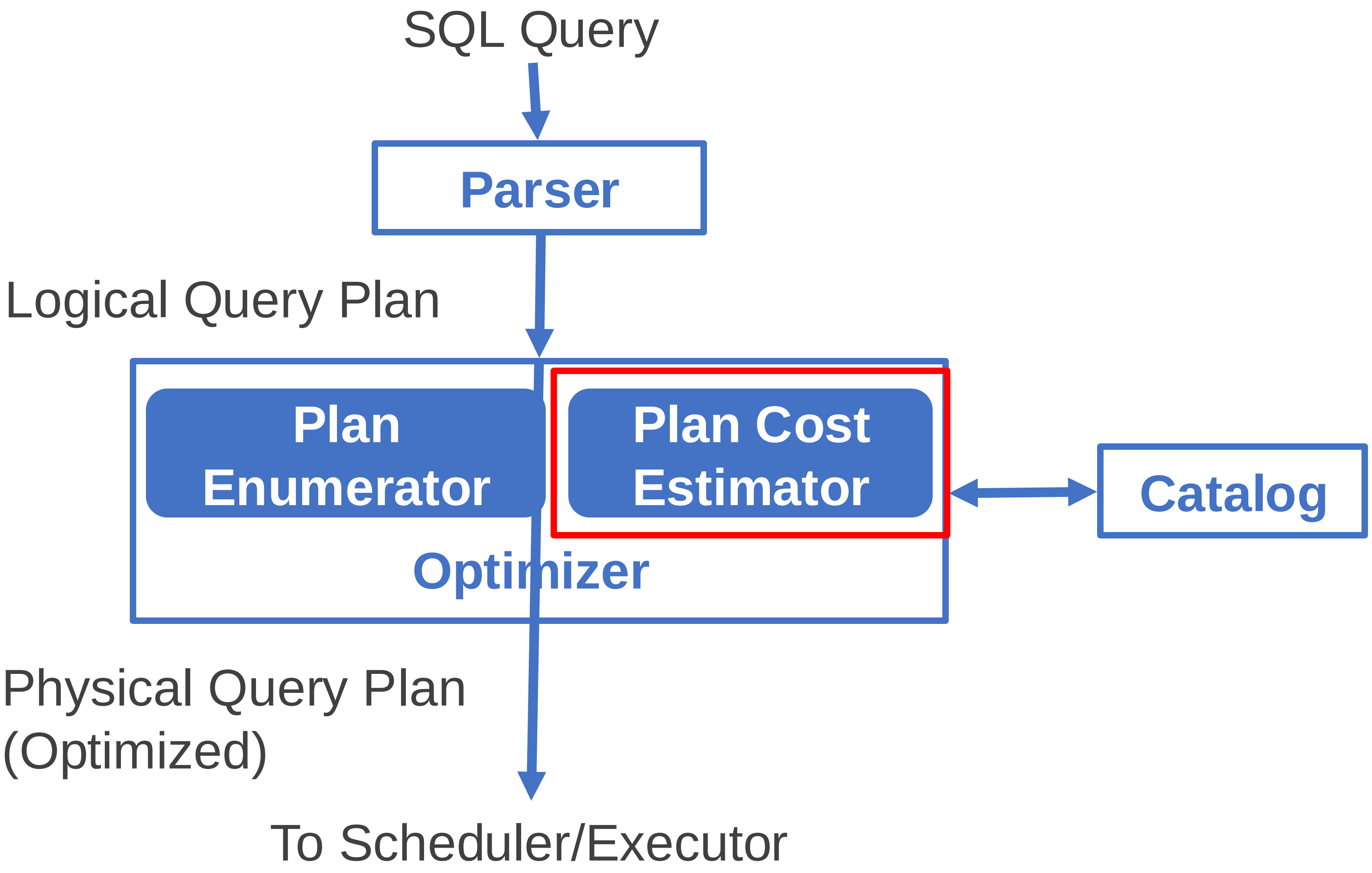
- “Interesting orderings” idea from System R optimizer paper
- Dynamic program to combine enumeration and costing

“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.
- Costing PQPs
- Materialized Views

Overview of Query Optimizer



SQL Query

Parser

Logical Query Plan

Plan Enumerator

Plan Cost Estimator

Catalog

Optimizer

Physical Query Plan
(Optimized)

To Scheduler/Executor

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

RatingID	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

RatingID	Stars	RateDate	UID	MID

10m pages

MID	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
- 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:

RatingID	Stars	RateDate	UID	MID

UID	Name	Age	JoinDate	MID	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;
```

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

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:

RatingID	Stars	RateDate	UID	MID

UID	Name	Age	JoinDate	MID	Name	Year	Director

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

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:

RatingID	Stars	RateDate	UID	MID

UID	Name	Age	JoinDate

MID	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:

Assume no duplicate inserts

$$V \leftarrow A \bowtie B$$

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

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

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