CS525: Advanced Database Organization

Notes 6: Query Processing
Part IV: Logical Optimization

Yousef M. Elmehdwi

Department of Computer Science

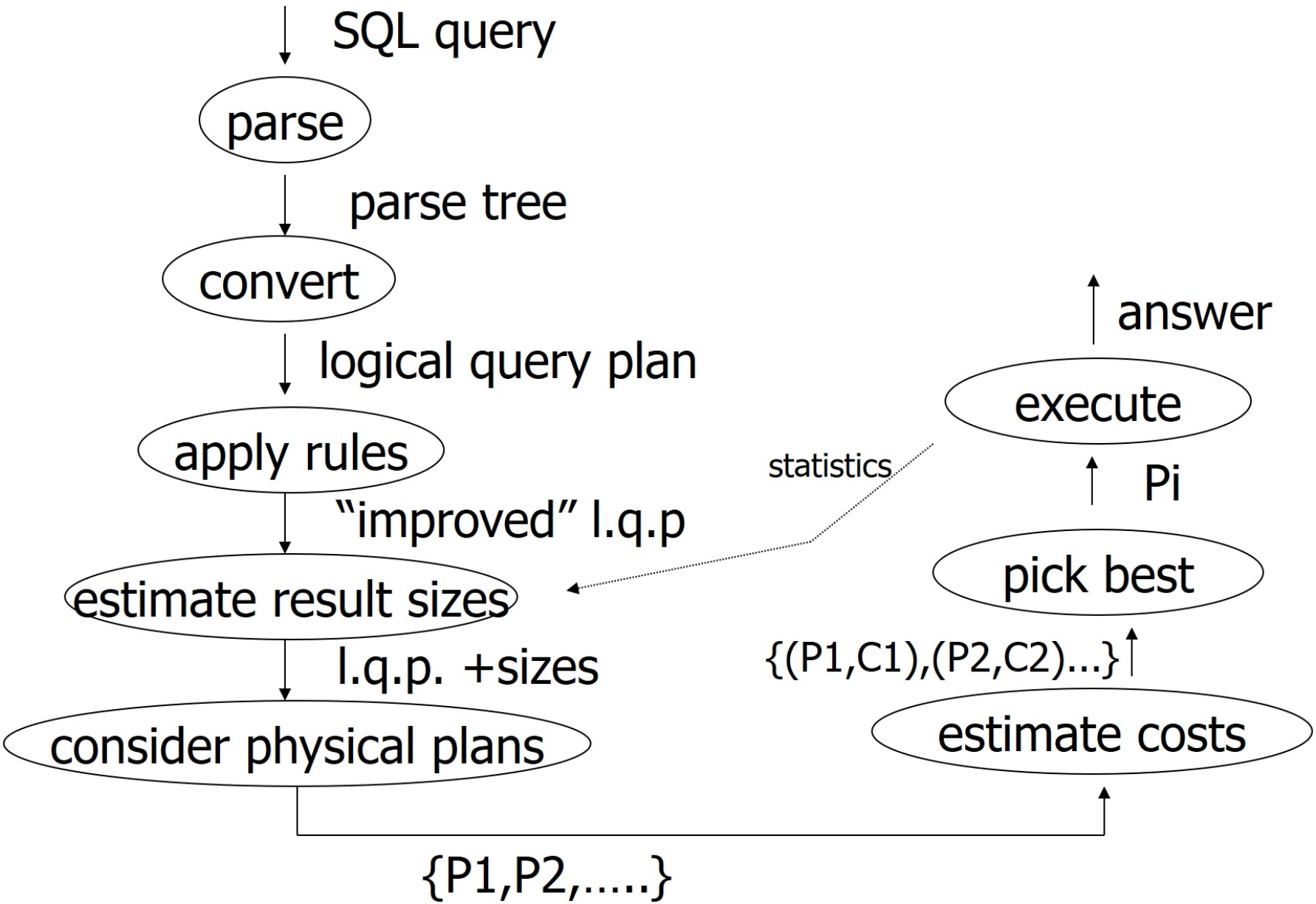
Illinois Institute of Technology

yelmehdwi@iit.edu

November 15, 2023

Slides: adapted from courses taught by Hector Garcia-Molina, Stanford, Shun Yan Cheung, Emory University, & Jennifer Welch, Texas A&M, Ramon Lawrence, Andy Pavlo, Carnegie Mellon University & Introduction to Database Systems by ITL Education Solutions Limited

Basic Steps in Processing an SQL Query



Query Plan

- Query plan is a list of instructions that the database needs to follow in order to execute a query on the data.
- The DBMS converts a SQL statement into a query plan.
- Operators are arranged in a tree.
- Data flows from the leaves towards the root.
- The output of the root node in the tree is the result of the query.
- Typically operators are binary (1-2 children).
- The same query plan can be executed in multiple ways.
- Most DBMSs will want to use an index scan as much as possible.

Query Optimization

- Remember that SQL is declarative.
  - The query only tells the DBMS what to compute, but not how to compute it
  - Thus, the DBMS needs to translate a SQL statement into an executable query plan.
- There can be a big difference in performance based on plan is used
  - There are different ways to execute a query (e.g., join algorithms) $\Rightarrow$ there will be differences in performance for these plans
- Thus, the DBMS needs a way to pick the "best" plan for a given query. This is the job of the DBMS's optimizer.

Query Optimization

- A single query can be executed through different algorithms or re-written in different forms and structures.
- *Q: Which of these forms or pathways is the most optimal?*
- `Query Optimization`
  - The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans
  - i.e., find the "cheapest" execution plan for a query
- *The role of query optimizer goal: Transform SQL queries into an efficient execution plan*

IBM SYSTEM R

- First implementation of a query optimizer from the 1970s.
- Prior to this, people did not believe that a DBMS could ever construct a query plan better than a human.
  - People argued that the DBMS could never choose a query plan better than what a human could write.
- Remains the hardest part of building a DBMS 30+ years later

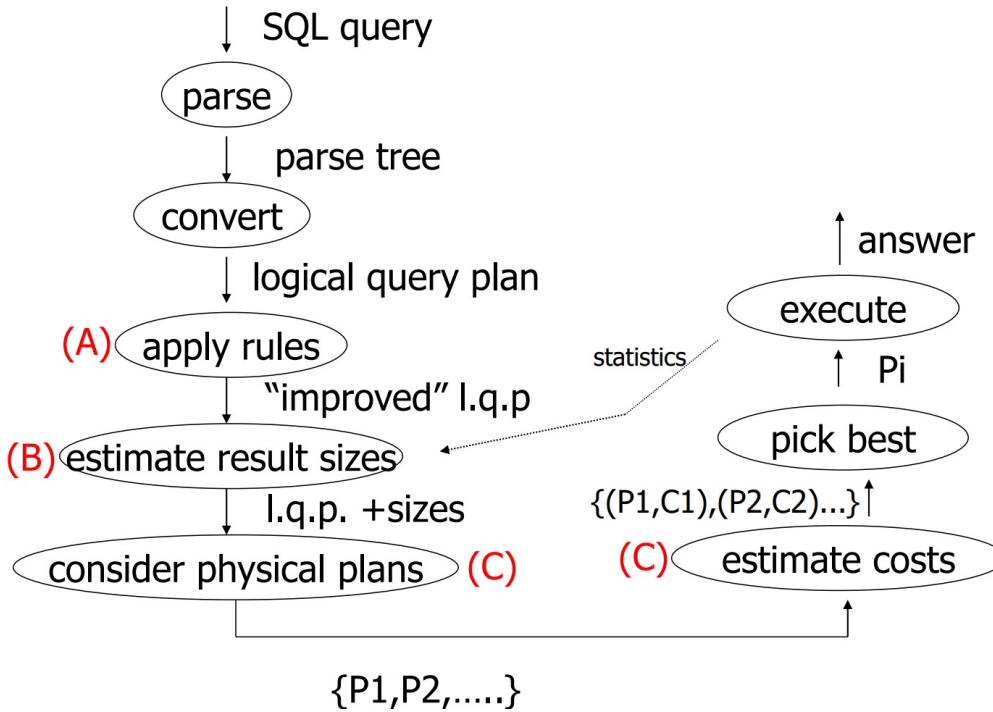
- *Many concepts, design decisions and assumptions made in the System R (became DB2) optimizer, such as cost models and statistical information, are still relevant today. Optimizers use statistics about data distribution and table sizes to estimate the cost of different query plans.*

Query Optimization

- There are two types of optimization strategies:
  - Heuristics/Rules
    - Use static rules, or heuristics.
    - Heuristics match portions of the query with known patterns to assemble a plan.
    - These rules transform the query to remove inefficiencies.
      - i.e., rewrite the query to remove stupid/inefficient things.
    - These techniques may need to examine catalog to understand the structure of the data, but they do not need to examine data.
    - *Use heuristics or rules written by humans to fix some obvious inefficiencies and apply some simple optimizations.*
  - Cost-based Search
    - Use a model to estimate the cost of executing equivalent plans.
    - Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.
      - The cost model chooses the plan with the lowest cost (in terms of performance).

Query Optimization

- Relational algebra level (A)
- Detailed query plan level
  - Estimate Result Size (B)
  - Estimate Costs (C)
    - without indexes
    - with indexes
  - Generate and compare plans (C)



Logical vs. Physical Plans

- The optimizer generates a mapping of a logical algebra expression to the optimal equivalent physical algebra expression.
- The logical plan is roughly equivalent to the relational algebra expressions in the query.
- Physical operators define a specific execution strategy using an access path for the different operators in the query plan.
  - Physical plans may depend on the physical format of the data that they process (i.e., sorting, compression).
  - Not always a 1:1 mapping from logical to physical plan.
- Logical operators: *what they do*
  - e.g., union, selection, project, join, grouping
- Physical operators: *how they do it*
  - e.g., nested loop join, sort-merge join, hash join, index join

Query Optimization is NP-Hard

- This is the hardest part of building a DBMS.
- If you are good at this, you will get paid $$$.
- People are starting to look at employing ML to improve the accuracy and efficacy of optimizers.

- Anonymous Quote
  - "*Query optimization is not rocket science. When you flunk out of query optimization, we make you go build rockets.*"

Query Optimization: The Main Steps

1. Enumerate logically equivalent plans by applying equivalence rules
2. For each logically equivalent plan, enumerate all alternative physical query plans
3. Estimate the cost of each of the alternative physical query plans
4. Run the plan with lowest estimated overall cost

Where we are

- `SQL` $\Rightarrow$ `parse tree` $\Rightarrow$ `expression of relational algebra (initial logical query plan)`
- The translation rules converting a parse tree to a logical query tree do not always produce the best logical query tree.
- Today: consider ways of transformations to improve the query plan
  - Algebraic laws for improving query plans

Optimizing/Improving the Logical Query Plan

- The translation rules converting a parse tree to a logical query tree do not always produce the best logical query tree.
- It is possible to optimize the logical query tree by applying relational algebra laws to convert the original tree into a more efficient logical query tree.
  - possibly giving smaller temporary relations
  - possibly reducing the number of disk I/Os
- Optimizing a logical query tree using relational algebra laws is called heuristic optimization because the optimization process uses common conversion techniques that result in more efficient query trees in most cases, but not always.
  - Applying algebraic laws turning the expression into an equivalent expression that will have a more efficient physical query plan
- We begin with a summary of relational algebra laws/Equivalence Rules.

Relational Algebra Optimization

- What are transformation rules?
  - preserve equivalence
- What are good transformations?
  - reduce query execution costs

Query Equivalence

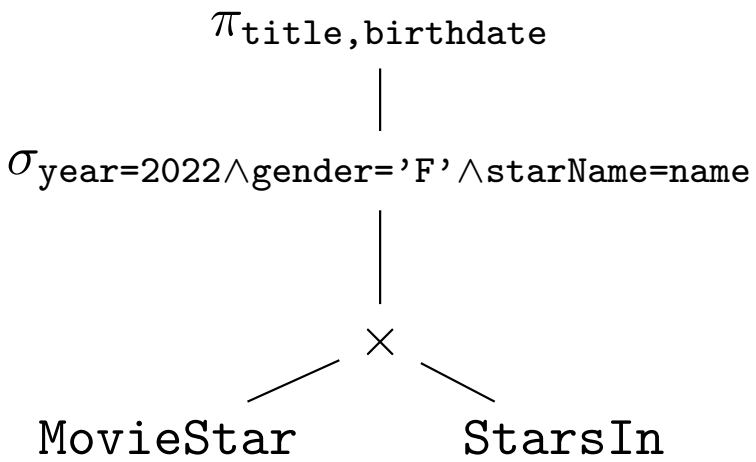
- Much of query optimization relies on the underlying concept that the high level properties of relational algebra are preserved across equivalent expressions.
- Two relational algebra expressions are equivalent if they generate the same set of tuples.
- The DBMS can identify better query plans without a cost model.
- Two queries $\texttt{q}_1$ and $\texttt{q}_2$ are equivalent:
  - If for every database instance `I` (contents of all the tables)
  - Both queries have the same result
- $\texttt{q}_1 \equiv \texttt{q}_2$ `iff` $\forall$ `I:` $\texttt{q}_1\texttt{(I)} = \texttt{q}_2\texttt{(I)}$

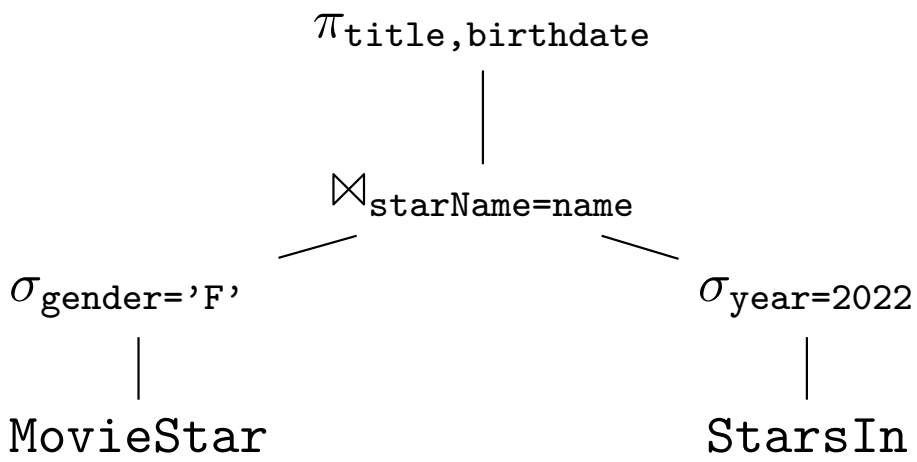
Query Equivalence

StarsIn(title,year,startName)

MovieStar(name,address,gender,birthdate)

```
SELECT    title, birthdate
FROM      MovieStar, StarsIn
WHERE     year=2022 AND gender='F' AND starName=name;
```





Algebraic Laws/Equivalence Rules

- Just like there are laws associated with the mathematical operators, there are laws associated with the relational algebra operators.
- The most common laws used for simplifying expressions are:
  - the commutative law allowing operators to be performed in any sequence, e.g.,
    - `x ○ y = y ○ x`
      `(where ○ is an operator)`
  - the associative law allowing operators to be grouped either from left or right, e.g.,
    - `x ○ (y ○ z) = (x ○ y) ○ z`

Algebraic Laws: Joins and Products

- Natural joins and product are both associative and commutative
  - `Join` ($\bowtie$) is commutative: $\texttt{R} \bowtie \texttt{S} = \texttt{S} \bowtie \texttt{R}$
  - `Join` ($\bowtie$) is associative: $\texttt{R} \bowtie (\texttt{S} \bowtie \texttt{T}) = (\texttt{R} \bowtie \texttt{S}) \bowtie \texttt{T}$
  - `Cross product` ($\times$) is commutative: $\texttt{R} \times \texttt{S} = \texttt{S} \times \texttt{R}$
  - `Cross product` ($\times$) is associative: $\texttt{R} \times (\texttt{S} \times \texttt{T}) = (\texttt{R} \times \texttt{S}) \times \texttt{T}$
  - will give the same attributes and concatenated tuples regardless of order (the attributes are named so the order of these does not matter)
- What about theta-join?
  - Commutative ($\texttt{R} \bowtie_c \texttt{S} = \texttt{S} \bowtie_c \texttt{R}$), but not always associative, e.g.,
    - `R(a,b)`, `S(b,c)`, and `T(c,d)`
    - $(\texttt{R} \bowtie_{\texttt{R.b<S.b}} \texttt{S}) \bowtie_{\texttt{a<d}} \texttt{T}) \neq \texttt{R} \bowtie_{\texttt{R.b<S.b}} (\texttt{S} \bowtie_{\texttt{a<d}} \texttt{T})$

Algebraic Laws: Joins and Products

- Q: Does it matter in which order join or product are performed with respect to performance, e.g., $\texttt{R} \times \texttt{S} \times \texttt{T} \times \ldots$?
- YES, it *may* be very important
  - if only one of the relations fits in memory, we should perform the operation using this relation first - one-pass operation reducing the number of disk I/Os
  - if joining or taking product of two of the relations in a large expression give a temporary relation which fits in memory, one should join these first to save both memory and disk I/Os
  - one should try to make the temporary result as small as possible to save memory
  - if we can estimate (using statistics) the amount of tuples being joined, we can save a lot of operations by joining the two relations giving fewest tuples first
- BUT, the final result will be the same

Note

- Carry attribute names in results, so order is not important
- Can also write as trees, e.g.:

- Different ordering in the execution of the ⋈ operations can produce different intermediate results (often with large difference in size of result sets)
- So one of the topics (problems) in query optimization will be:
  - `Find the optimal join ordering of a set of ⋈ operations`

Algebraic Laws: Union and Intersect

- Union and intersection are both associative and commutative:
  - `Union` (∪) is commutative: `R` ∪ `S` = `S` ∪ `R`
  - `Union` (∪) is associative: `R` ∪ (`S` ∪ `T`) = (`R` ∪ `S`) ∪ `T`
  - `Intersection` (∩) is commutative: `R` ∩ `S` = `S` ∩ `R`
  - `Intersection` (∩) is associative: `R` ∩ (`S` ∩ `T`) = (`R` ∩ `S`) ∩ `T`

Laws for Bags and Sets Can Differ

- Example of an Algebraic Law that holds for set, but not for bags (e.g., distributive law of intersection over union)
- We know from `Set Theory` that

$$\texttt{A} \cap_{\texttt{set}} (\texttt{B} \cup_{\texttt{set}} \texttt{C}) = (\texttt{A} \cap_{\texttt{set}} \texttt{B}) \cup_{\texttt{set}} (\texttt{A} \cap_{\texttt{set}} \texttt{C})$$

- But, this law does not hold for bags:
  - Suppose bags `A`, `B`, and `C` were each $\{x\}$

$$\begin{aligned}
\texttt{A} \cap_{\texttt{bag}} (\texttt{B} \cup_{\texttt{bag}} \texttt{C}) &= \{x\} \cap_{\texttt{bag}} (\{x\} \cup_{\texttt{bag}} \{x\}) \\
&= \{x\} \cap_{\texttt{bag}} \{x, x\} \\
&= \{x\}
\end{aligned}$$

$$\begin{aligned}
(\texttt{A} \cap_{\texttt{bag}} \texttt{B}) \cup_{\texttt{bag}} (\texttt{A} \cap_{\texttt{bag}} \texttt{C}) &= (\{x\} \cap_{\texttt{bag}} \{x\}) \cup_{\texttt{bag}} (\{x\} \cap_{\texttt{bag}} \{x\}) \\
&= \{x\} \cup_{\texttt{bag}} \{x\} \\
&= \{x, x\}
\end{aligned}$$

Algebraic Laws: Selection

- Selection is a very important operator in terms of query optimization
  - reduces the number of tuples (size of relation)
- Some selection optimizations include:
  - Performing filters as early as possible.
    - push selects as far down the tree as possible
  - Reordering predicates so that the DBMS applies the most selective one first.
  - Breaking up a complex predicate and pushing it down.
- Selection is `idempoten`. (multiple applications of the same selection have no additional effect beyond the first one)
  - σ_p`(R)` $=$ $\sigma_p\sigma_p$`(R)`
- Select operations are `commutative`. (the order selections are applied in has no effect on the eventual result)
  - $\sigma_p\sigma_q$`(R)` $=$ $\sigma_q\sigma_p$`(R)`

Algebraic Laws: Selection

- Break a complex predicate, and push down
  - The selection condition involving conjunction of two or more predicates can be deconstructed into a sequence of individual select operations.
    - $\sigma_{p_1 \wedge p_2}(\mathtt{R}) = \sigma_{p_1}\left(\sigma_{p_2}(\mathtt{R})\right) = \sigma_{p_2}\left(\sigma_{p_1}(\mathtt{R})\right)$
    - $\sigma_{p_1 \vee p_2}(\mathtt{R}) = \left(\sigma_{p_1}(\mathtt{R})\right) \cup_{set} \left(\sigma_{p_2}(\mathtt{R})\right)$

Algebraic Laws: Selection

- Push selections through the binary operators: product, union, intersection, difference, and join.
- if pushing select, select ...
  1. must be pushed through both arguments for union:
     - Union: $\sigma_p(\texttt{R} \cup \texttt{S}) = \sigma_p(\texttt{R}) \cup \sigma_p(\texttt{S})$
  2. must be pushed through first arguments, optionally second for difference:
     - Difference: $\sigma_p(\texttt{R - S}) = \sigma_p(\texttt{R}) \texttt{ - S}$
     - $\sigma_p(\texttt{R - S}) = \sigma_p(\texttt{R}) \texttt{ - } \sigma_p(\texttt{S})$
  3. For the other operators it is only required that the selection be pushed to one argument.
     - product, natural join, theta join, intersection
     - e.g., $\sigma_p(\texttt{R} \times \texttt{S}) = \sigma_p(\texttt{R}) \times \texttt{S}$, if p contains only attributes from R

Algebraic Laws: Selection

- If the selection condition `p` involves only the attributes of `R` and `q` involves the attributes of `S`, then the select operation distributes.
  - $\sigma_{p \wedge q}(\texttt{R} \bowtie \texttt{S}) = \sigma_p(\texttt{R}) \bowtie \sigma_q(\texttt{S})$
- Let
  - p = predicate with only `R` attributes
  - q = predicate with only `S` attributes
  - m = predicate with `R,S` attributes
  - $\sigma_p(\texttt{R} \bowtie \texttt{S}) = \sigma_p(\texttt{R}) \bowtie \texttt{S}$
  - $\sigma_q(\texttt{R} \bowtie \texttt{S}) = \texttt{R} \bowtie \sigma_q(\texttt{S})$
  - Some Rules can be Derived:
    - $\sigma_{p \wedge q}(\texttt{R} \bowtie \texttt{S}) = \sigma_p(\texttt{R}) \bowtie \sigma_q(\texttt{S})$
    - $\sigma_{p \wedge q \wedge m}(\texttt{R} \bowtie \texttt{S}) = \sigma_m\left(\sigma_p(\texttt{R}) \bowtie \sigma_q(\texttt{S})\right)$
    - Derivation for first one

$$\begin{aligned}\sigma_{p \wedge q}(R \bowtie S) &= \sigma_p\left(\sigma_q(R \bowtie S)\right) \\ &= \sigma_p\left(R \bowtie \sigma_q(S)\right) \\ &= \sigma_p(R) \bowtie \sigma_q(S)\end{aligned}$$

Algebraic Laws: Selection

Example

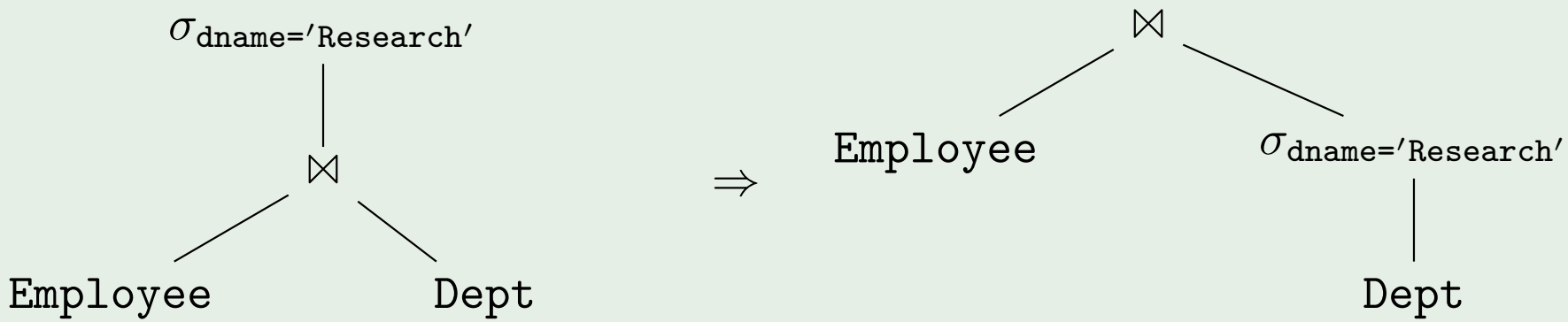
- `R(a,b)`
- `S(c,d)`

$$\begin{aligned}\sigma_{a=3 \wedge c=4}(R \cap S) &= \sigma_{a=3}\big(\sigma_{c=4}(R \cap S)\big) \\ &= \sigma_{a=3}\big(R \cap \sigma_{c=4}(S)\big) \\ &= \sigma_{a=3}(R) \cap \sigma_{c=4}(S)\end{aligned}$$

Algebraic Laws: Selection: Pushing Selections

Example

- `Employee(fname, salary, dno)`
- `Dept(dname, dno)`
- $\sigma_{\texttt{dname='Research'}}(\texttt{Employee} \bowtie \texttt{Dept}) = \texttt{Employee} \bowtie \sigma_{\texttt{dname='Research'}}(\texttt{Dept})$
- "`Pushing down`" a selection (σ) will result in a smaller intermediate result set



Algebraic Laws: Selection: Pushing Selections

Example

- `Employee(fname, salary, dno)`
- `Dept(dname, dno)`
- $\sigma_{dname='Research' \wedge fname='John'}(\texttt{Employee} \bowtie \texttt{Dept})$
  $= \sigma_{fname='John'}\left(\sigma_{dname='Research'}(\texttt{Employee} \bowtie \texttt{Dept})\right)$
  $= \sigma_{fname='John'}\left(\texttt{Employee} \bowtie \sigma_{dname='Research'}(\texttt{Dept})\right)$
  $= \sigma_{fname='John'}(\texttt{Employee}) \bowtie \sigma_{dname='Research'}(\texttt{Dept})$

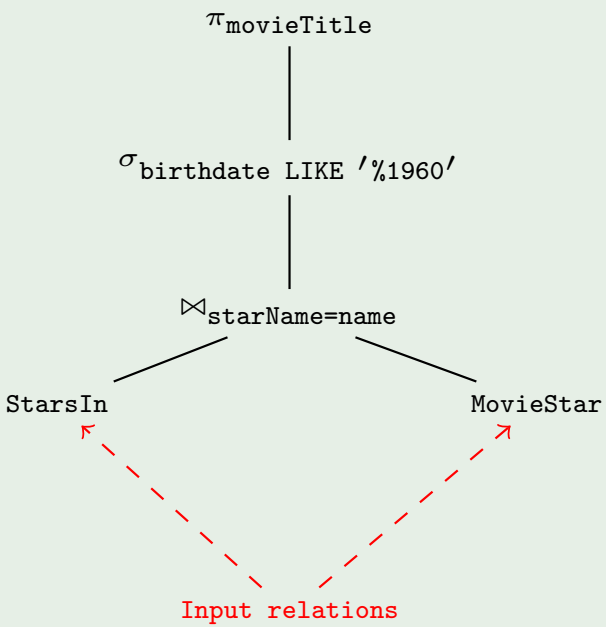
Common used query optimization technique involving σ

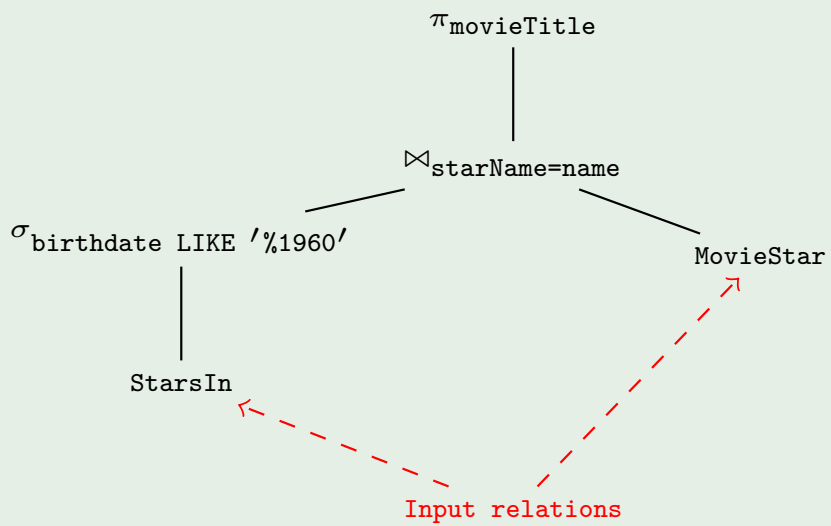
- Simple query optimization
  - The running time of database operations depends on:
    - The size of the input relations (operands)
  - Therefore: It is always beneficial (for running time) to reduce the size of the input relation(s)

Reducing the size of input relation using σ

- The selection operator σ can reduce the size of the input relation of some operators

Example



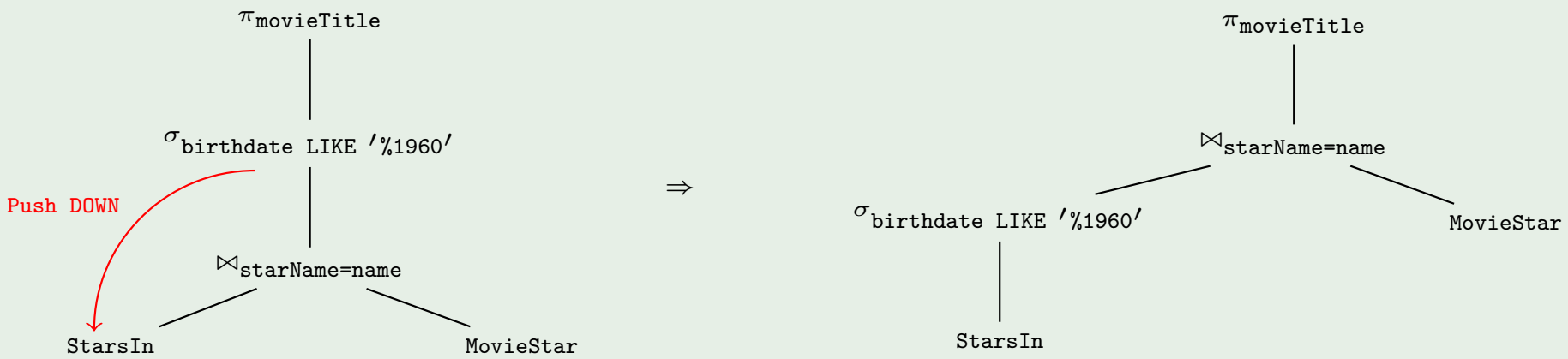


- The input relation of $\bowtie$ in the second case $\sigma_{\texttt{birthday LIKE '\%1960'}}(\texttt{StarsIn})$ can be much smaller than the input relation StarsIn

Simple query optimization technique: "push select down"

- One of the many query optimization techniques used by the DBMS is execute a σ_p as soon as possible.
- In terms of a `query tree`, it means that the σ_p operation is push as far down the `logical query tree` as possible

Example



Note: "push select down" query optimization technique

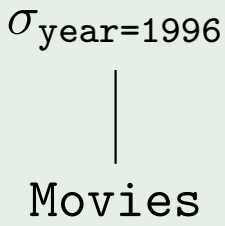
- When a query contains a virtual table, then the σ_p operation is **pushed down** the **logical query tree** as far as possible is not sufficient
- Sometimes it is useful to push selection the other way, i.e., up in the tree, using the law $\sigma_p(R \bowtie S) = R \bowtie \sigma_p(S)$ backwards

Example

- Relations:
  - **StarsIn**(title, year, starName, birthday) // Movie stars
  - **Movies**(title, year, gender, studioName) // Movies
- View:

```
CREATE VIEW Movies96 AS {
  SELECT  *
  FROM  Movies
  WHERE year = 1996
 }
```

- Corresponding logical query plan



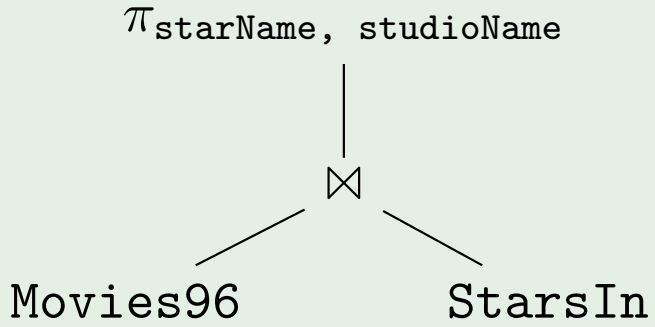
Note: “push select down” query optimization technique

Example (Continue)

- Query: Find all movie stars and their studio name in movies of 1996

```
SELECT    starName,  studioName
FROM      Movies96, StarsIn
WHERE     Movies96.title = StarsIn.title;
```

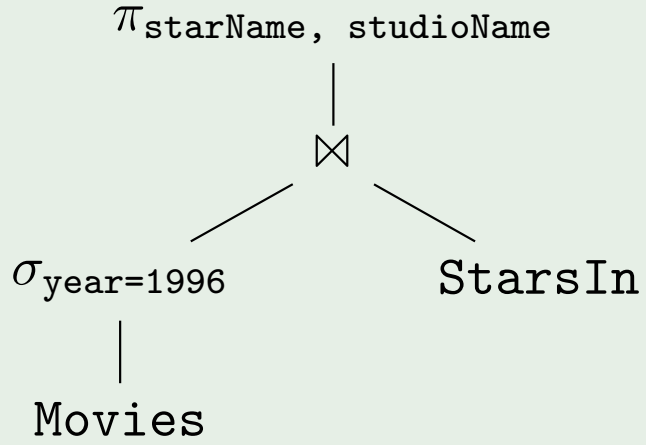
- initial logical query plan



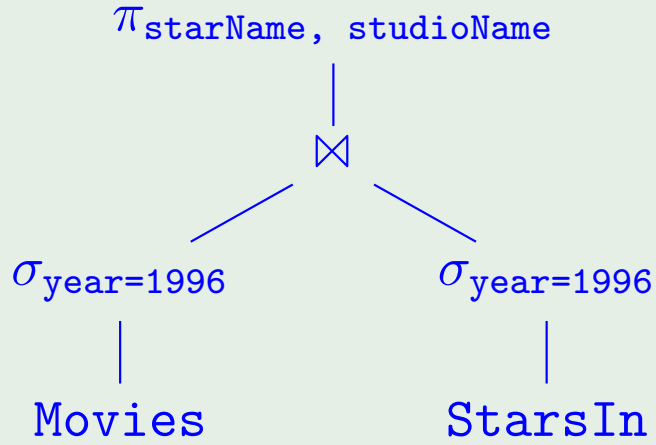
Note: "push select down" query optimization technique

Example (Continue)

- After replacing the `virtual table` with the corresponding query:



- However, the optimal query plan is as follows:

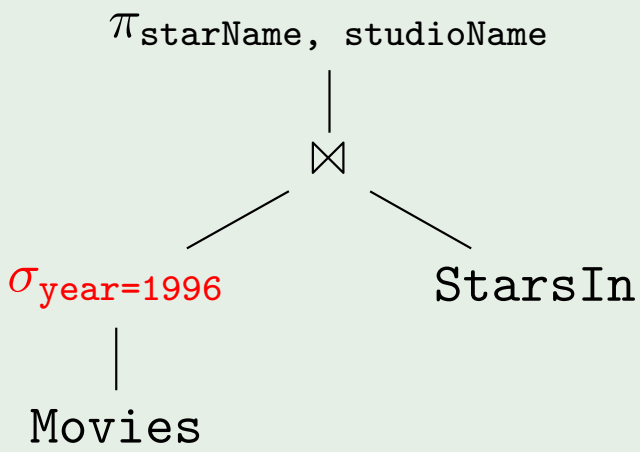


Amendment to the simple query optimization technique

- If there are virtual table in the query plan, then to find the optimal query plan, we must
  - Push any selection σ operators in the virtual table as far up the query tree as possible
  - Push every selection σ operators in the resulting query tree as far down the query tree as possible

Example

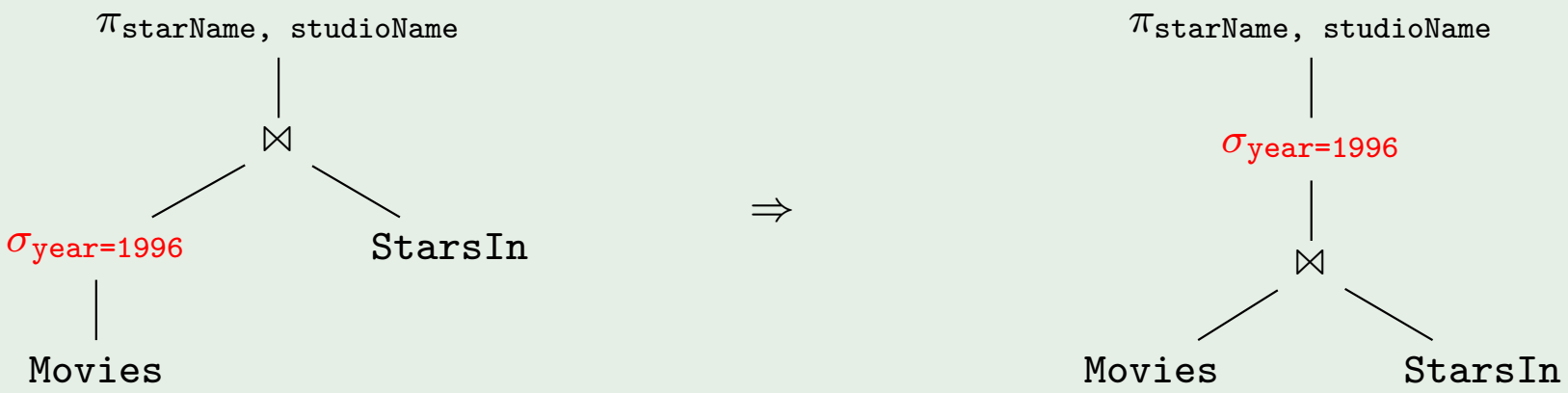
- Query plan after incorporating the virtual table query:



Amendment to the simple query optimization technique

Example (Continue)

- Use this algebraic law in the reverse order:$\sigma_p(\texttt{R} \bowtie \texttt{S}) = \sigma_p(\texttt{R}) \bowtie \texttt{S}$ to push the $\sigma_{year=1996}$ operation **up the tree**

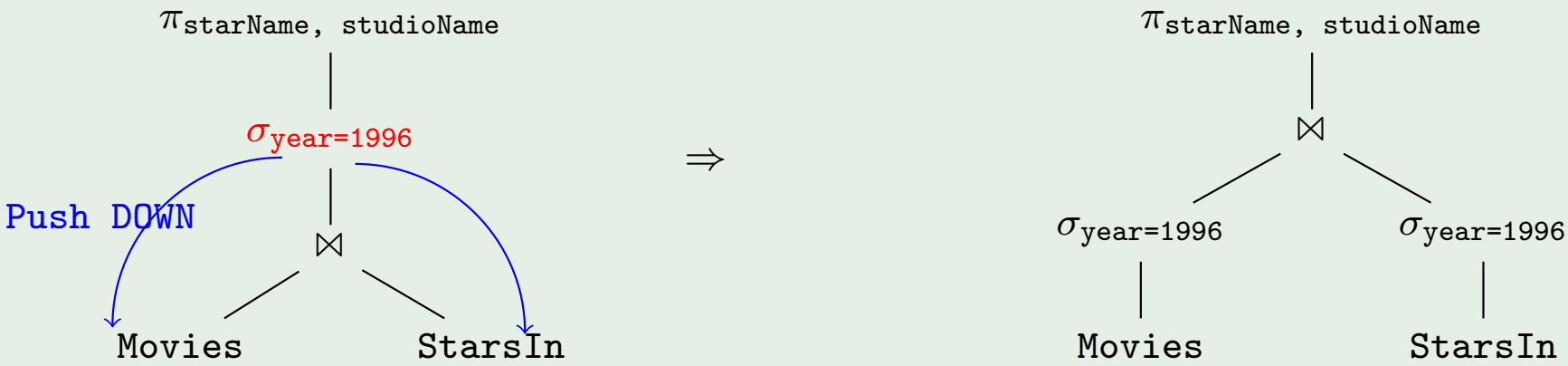


Amendment to the simple query optimization technique

Example (Continue)

- Both relations have the attribute `year`
- Use this algebraic law in the `forward order`:

  $\sigma_p(\mathtt{R} \bowtie \mathtt{S}) = \sigma_p(\mathtt{R}) \bowtie \sigma_p(\mathtt{S})$ to **push** the $\sigma_{\mathtt{year=1996}}$ operation **down the tree**



Relational Algebra Equivalences: Selections: Summary

- Perform filters as early as possible.
- Reorder predicates so that the DBMS applies the most selective one first.
- Break a complex predicate, and push down

$$\sigma_{p_1 \wedge \sigma_{p_2} \wedge \ldots \wedge \sigma_{p_n}}(\mathtt{R}) = \sigma_{p_1}\left(\sigma_{p_2}\left(\ldots \sigma_{p_n}(\mathtt{R})\right)\right)$$

Algebraic Laws: Projection

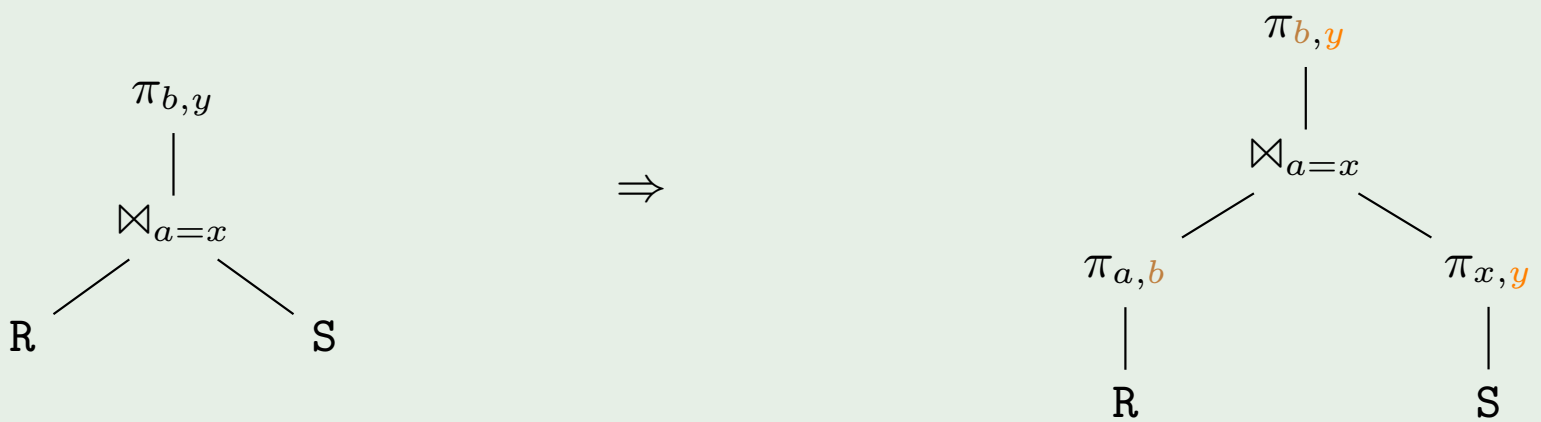
- The projection operator π can remove unnecessary attributes from intermediate results
- Some projection optimizations include:
  - Perform projections as early as possible to create smaller tuples and reduce intermediate results (projection pushdown).
  - Project out all attributes except the ones requested or requires (e.g., joining keys)

Algebraic Laws: Projection

- Projections can be pushed down through many operators:
  - The projection operator π may be introduced anywhere as long as it does not remove any attributes used above in the tree
  - The projection operator is thus often not moved, we introduce a new
  - Adding a projection lower in the tree can improve performance, since often tuple size is reduced

Example

R(a, b, c), S(x, y, z)



Algebraic Laws: Project

- If a query contains a sequence of project operations, only the final operation is needed, the others can be omitted.
  - $\pi_{L_1}\left(\pi_{L_2}\left(\ldots\left(\pi_{L_n}\texttt{(R)}\right)\ldots\right)\right) = \pi_{L_1}\texttt{(R)}$, where $L_i \subseteq L_{i+1}$ for $\texttt{i}\in\texttt{[1,n)}$
  - This transformation is called cascading of project operator.

Rules: Projections

Let:

- $\mathtt{X}$ = set of attributes
- $\mathtt{Y}$ = set of attributes
- $\mathtt{XY} = \mathtt{X} \cup \mathtt{Y}$
- $\pi_{\mathtt{XY}}(\mathtt{R}) = \pi_{\mathtt{X}}\left(\pi_{\mathtt{Y}}(\mathtt{R})\right)$ Is this correct?

Algebraic Laws: Project

Let:

- $\mathtt{X}$ = set of attributes
- $\mathtt{Y}$ = set of attributes
- $\mathtt{XY} = \mathtt{X} \cup \mathtt{Y}$
- $\pi_{\mathtt{XY}}(\mathtt{R}) = \pi_{\mathtt{X}}\left(\pi_{\mathtt{Y}}(\mathtt{R})\right)$

Rules: π, σ combined

- It is also possible to push a projection below a selection.
- If the selection condition p involves only the attributes $a_1, a_2, \ldots, a_n$ that are present in the projection list, the two operations can be commuted.
  - $\pi_{a_1,a_2,\ldots,a_n}\big(\sigma_p(\texttt{R})\big) = \sigma_p\big(\pi_{a_1,a_2,\ldots,a_n}(\texttt{R})\big)$
- Rule: $\pi_L\big(\sigma_p(\texttt{R})\big) = \pi_L\Big(\sigma_p\big(\pi_M(\texttt{R})\big)\Big)$, where M is all attributes used by L Union p

Rules: π, σ combined

- Let
  - x = subset of `R` attributes
  - z = attributes in predicate p (subset of `R` attributes)
  - $\pi_x\big(\sigma_p(\texttt{R})\big) = \sigma_p\big(\pi_x(\texttt{R})\big)$

Rules: π, σ combined

- Let
  - `x` = subset of `R` attributes
  - `z` = attributes in predicate `p` (subset of `R` attributes)
  - $\pi_x\big(\sigma_p(\texttt{R})\big) = \sigma_p\big(\overline{\pi_x}(\texttt{R})\big)$

Rules: π, σ combined

- Let
  - $\mathtt{x}$ = subset of $\mathtt{R}$ attributes
  - $\mathtt{z}$ = attributes in predicate $\mathtt{p}$ (subset of $\mathtt{R}$ attributes)
  - $\pi_x\left(\sigma_p(\mathtt{R})\right) = \pi_x\left(\sigma_p\left(\pi_{xz}(\mathtt{R})\right)\right)$

Rules: π, $\bowtie$ combined

- Let
  - x = subset of `R` attributes
  - y = subset of `S` attributes
  - z = intersection of `R`,`S` attributes
  - $\pi_{xy}(\texttt{R} \bowtie \texttt{S}) = \pi_{xy}\left(\pi_{xz}(\texttt{R}) \bowtie \pi_{yz}(\texttt{S})\right)$

Rules: π, $\bowtie$ combined

- Let
  - x = subset of `R` attributes
  - y = subset of `S` attributes
  - z = intersection of `R`,`S` attributes
  - $\pi_{xy}\big(\sigma_p(\texttt{R} \bowtie \texttt{S})\big) = \pi_{xy}\Big(\sigma_p\big(\pi_{xz'}(\texttt{R}) \bowtie \pi_{yz'}(\texttt{S})\big)\Big)$, where $z' = z \cup \{attributes\ used\ in\ p\}$

Replace Cartesian Products

- There are two laws that are important with respect to performance following from the definition of join
  - $(\texttt{R} \bowtie_p \texttt{S}) = \sigma_p(\texttt{R} \times \texttt{S})$ (theta join)
  - $(\texttt{R} \bowtie \texttt{S}) = \pi_L\left(\sigma_p(\texttt{R} \times \texttt{S})\right)$ (natural join)
    - where p equates each pair of tuples from R and S with the same name
    - L is a list of attributes including all distinct attributes from R and S
- *If one has the opportunity to apply these rules, it generally will increase performance, because the algorithms for computing a join are much faster than computing the product followed by a selection on a very large relation*

Algebraic Laws: Duplicate Elimination

- Moving Duplicate elimination δ down the tree is potentially beneficial as it can reduce the size of intermediate relations
- Can be eliminated if argument has no duplicates
  - a relation with a primary key
  - a relation resulting from a grouping operator
- Push the δ operation through product, join, theta-join, selection, and bag intersection
  - Ex: $\delta(\texttt{R} \times \texttt{S}) = \delta(\texttt{R}) \times \delta(\texttt{S})$
  - The result of δ is always a set (i.e.: no duplicates)
- Cannot push δ through bag union, bag difference or projection
- The cost saving resulting from pushing down δ is usually small.

  $\rightarrow$ this optimization step is often not implemented

Duplicate Elimination Pitfalls

Example

- `R` has two copies of tuple `t`
- `S` has one copy of `t`
- `T(a,b)` contains only `(1,2)` and `(1,3)`
- **Bag Union**
  - δ`(R` $\cup_{bag}$ `S)` has one copy of `t`
  - δ`(R)` $\cup_{bag}$ δ`(S)` has two copies of `t`
- **Bag difference**
  - δ`(R` $-$ `S)` has one copy of `t`
  - δ`(R)` $-$ δ`(S)` has no copies of `t`
- **Bag projection**
  - $\delta\big(\pi_a$`(T)`$\big) = \{$`1`$\}$
  - $\pi_a\big(\delta$`(T)`$\big) = \{$`1,1`$\}$

Algebraic Laws: Grouping and Aggregation

- Whether or not the grouping operator can be pushed depends on details of the aggregate operator used
  - cannot state general rules
- The grouping operator only interact with very few relation algebra operations:
  1. γ_L produces a set, therefore the δ operation is unnecessary
     - $\delta\left(\gamma_L(\texttt{R})\right) = \gamma_L(\texttt{R})$
  2. You can project out some attributes as long as you keep the grouping attributes:
     - $\gamma_L(\texttt{R}) = \gamma_L\left(\pi_M(\texttt{R})\right)$, where M must contain all attributes used by γ_L
  3. The aggregate functions `max` and `min` are not dependent on duplicates
     - $\gamma_L(\texttt{R}) = \gamma_L\left(\delta(\texttt{R})\right)$, where γ_L =max or min
     - max(5,5,3)=max(5,3)
  4. The aggregate functions `sum`, `count`, and `avg` are dependent on duplicates.

Nested Sub-Queries

- The DBMS treats nested sub-queries in the WHERE clause as functions that take parameters and return a single value or set of values.
- Two Approaches:
  - Rewrite to de-correlate and/or flatten them
  - Decompose nested query and store result to temporary table

Rewrite to De-correlate and/or Flatten

- De-correlation
  - In many cases, nested sub-queries involve correlated sub-queries, where the inner query references a column from the outer query.
  - De-correlation involves rewriting the query to eliminate this dependency, making it independent of the outer query.
  - This can improve performance.
- Flatten
  - If a sub-query returns multiple rows, flattening involves rewriting the query to use JOINs or other techniques to achieve the same result without a nested sub-query.

Rewrite to De-correlate and/or Flatten

```
SELECT *
FROM employees e
WHERE salary > (SELECT AVG(salary) FROM employees WHERE
    department_id = e.department_id);
```

```
FROM employees e
JOIN (
    SELECT department_id, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department_id
) AS subquery
ON e.department_id = subquery.department_id
WHERE e.salary > subquery.avg_salary;
```

Decomposing Queries

- For harder queries, the optimizer breaks up/decompose the nested sub-queries into blocks and then concentrates on one block at a time.
- Sub-queries are written to a temporary table that are discarded after the query finishes.

Decomposing Queries

```
SELECT *
FROM employees e
WHERE salary > (SELECT AVG(salary) FROM employees WHERE
    department_id = e.department_id);
```

```
SELECT department_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id;

SELECT e.*
FROM employees e
JOIN temp_avg_salary subquery
ON e.department_id = subquery.department_id
WHERE e.salary > subquery.avg_salary;
```

Expression Rewriting

- An optimizer transforms a query's expressions (e.g., WHERE clause predicates) into the optimal/minimal set of expressions.
- Implemented using if/then/else clauses or a pattern-matching rule engine.
    - Search for expressions that match a pattern.
    - When a match is found, rewrite the expression.
    - Halt if there are no more rules that match.

- *if/then/else Clauses: These statements evaluate various conditions based on the characteristics of the query and choose the most appropriate optimization strategies accordingly.*

More transformations

- Another optimization that a DBMS can use is to remove impossible or unnecessary predicates.
- That is, remove the evaluation of predicates those result does not change per tuple in a table.
- Bypassing these predicates will reduce computation cost

More transformations

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

More transformations

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

```
-- Predicate will always be false and can be disregarded
```

More transformations

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 1;

-- Predicate will always return true and can also be
   bypassed.
```

More transformations

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Impossible / Unnecessary Predicates

```
SELECT * FROM A;
```

More transformations

- Similarly, unnecessary joins can be eliminated a

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Join Elimination

```
SELECT A1.*
FROM   A AS A1 JOIN A AS A2 ON A1.id = A2.id;

-- Wasteful- every tuple in A must exist in A.
```

More transformations

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Join Elimination

```
SELECT *
FROM   A;
```

More transformations

- Another optimization is ignoring projections that are unnecessary.

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Ignoring Projections

```
SELECT * FROM A AS A1
WHERE EXISTS(SELECT val FROM A AS A2
             WHERE A1.id = A2.id);

-- contains a wasteful join and projection because it
    checks that any output found exists.
```

More transformations

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Ignoring Projections

```
SELECT *
FROM A;
```

More transformations

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Merging Predicates

```
SELECT * FROM A
WHERE val BETWEEN 1 AND 100
      OR val BETWEEN 50 AND 150;
```

More transformations

Example

```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT NOT NULL );
```

- Merging Predicates

```
SELECT * FROM A
WHERE val BETWEEN 1 AND 150;
```

Heuristic Query Optimization

- We can use static rules and heuristics to optimize a query plan without needing to understand the contents of the database.
- Heuristic query optimization takes a logical query tree as input and constructs a more efficient logical query tree by applying equivalence preserving relational algebra laws.
- Equivalence preserving transformations insure that the query result is identical before and after the transformation is applied. Two logical query trees are equivalent if they produce the same result.

- Note that heuristic optimization does not always produce the most efficient logical query tree as the rules applied are only heuristics!

Rules of Heuristic Query Optimization

- The described relational algebraic laws are used to improve - or rewrite - the LQPs generated from the parse tree to improve performance
- The most commonly used in query optimizers are:
  - push selects as far down as possible
    If the select condition consists of several parts, we often split the operation in several selects and push each select as far down as possible in tree
  - push projects as far down as possible
    Projects can be added anywhere as long as attributes used above in the tree is included
  - duplicate eliminations can sometimes be removed (e.g., if on key)
  - if possible, combine select with cartesian products to form a type of join
- But, no transformation is always good

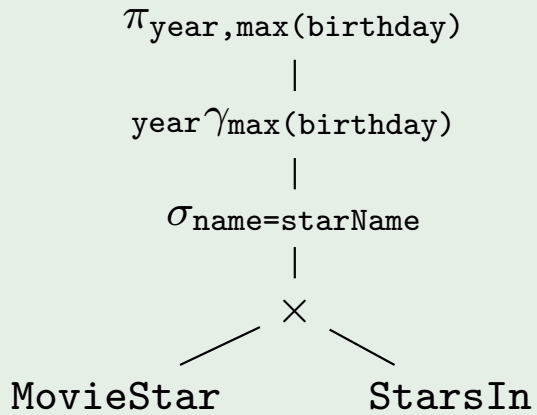
Applying the Algebraic laws for query optimization

Example

- **MovieStar**(name, addr, gender, birthdate)
- **StarsIn**(title, year, starName)
- Query: For each (movie) year, find the earliest birthday (youngest movie star) in that (movie) year

```
SELECT      year, max(birthday)
FROM        movieStar, StarsIn
WHERE       name = starName
GROUP BY    year
```

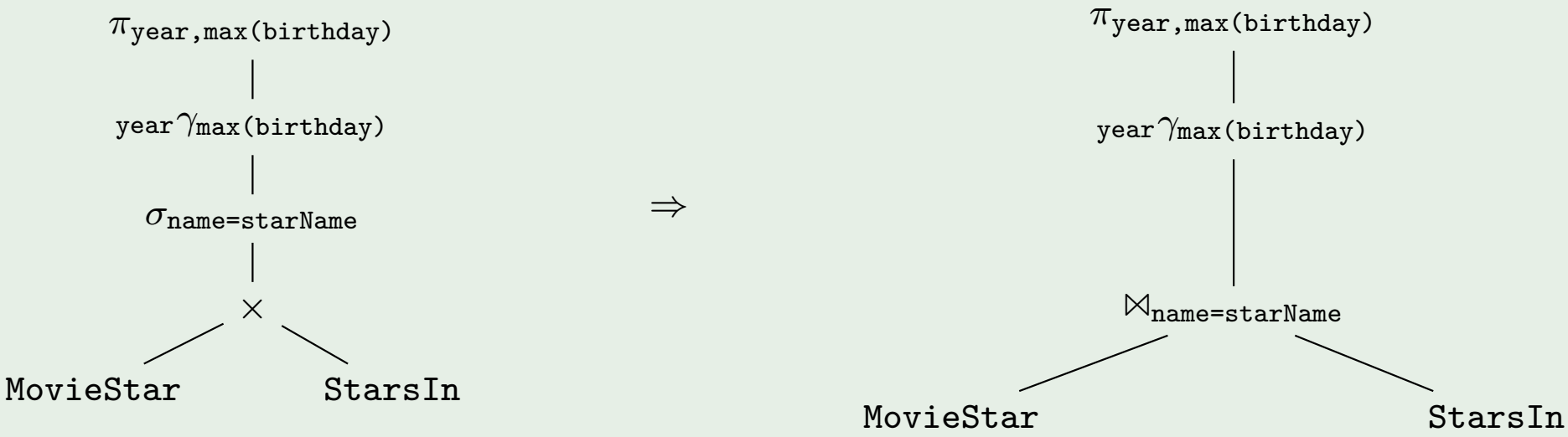
- The initial logical query plan is as follows:



Applying the Algebraic laws for query optimization

Example (Continue)

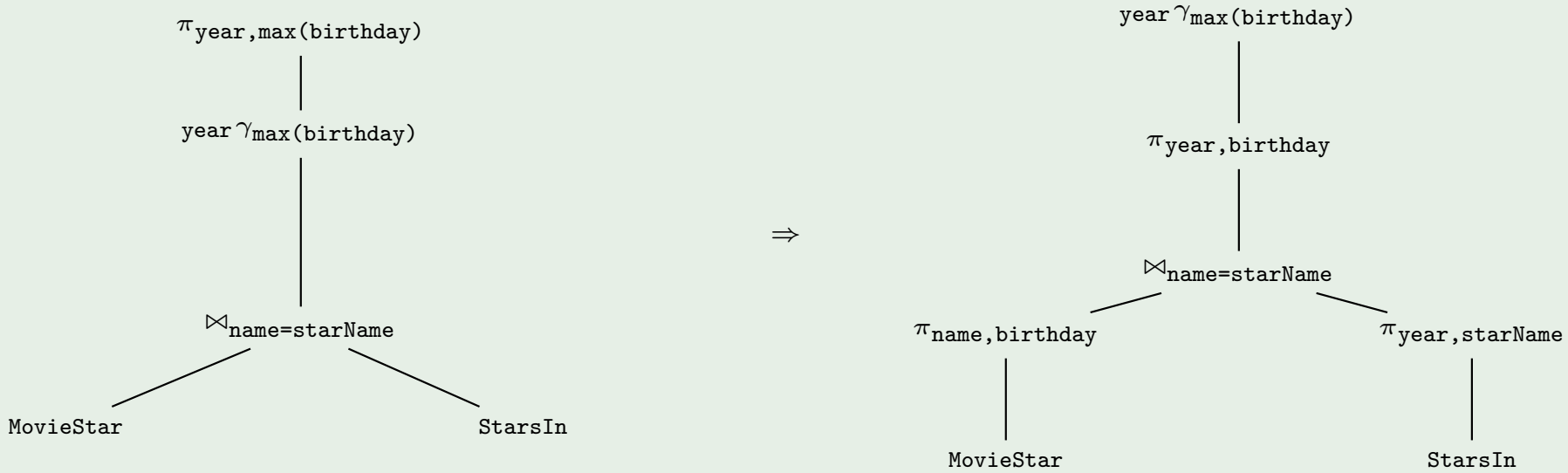
- Apply: $\texttt{R} \bowtie_p \texttt{S} = \sigma_p(\texttt{R} \times \texttt{S})$, where $p =$ "$name = starName$"



Applying the Algebraic laws for query optimization

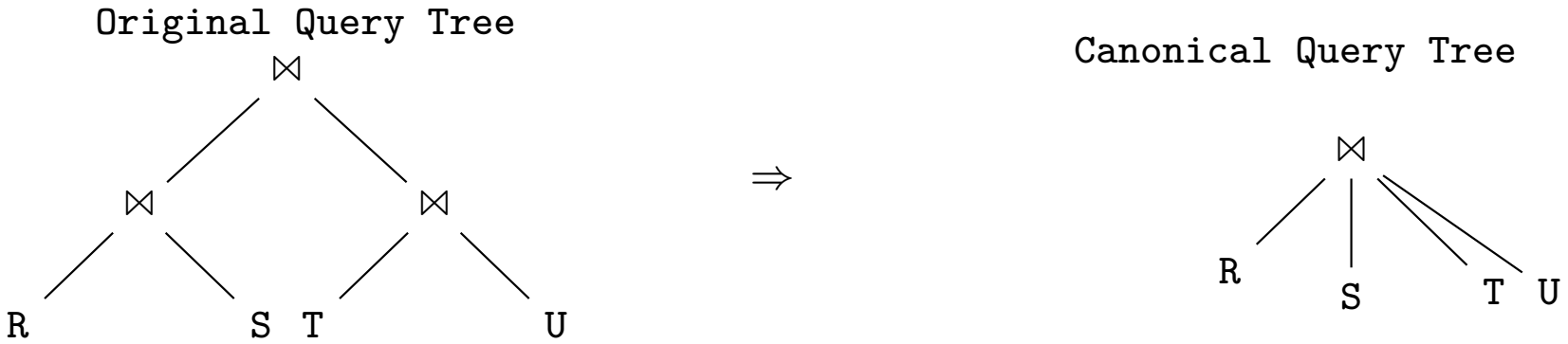
Example (Continue)

- Add harmless projections to remove unused attributes



Canonical Logical Query Trees

- A canonical logical query tree (Grouping Operators) is a logical query tree where all associative and commutative operators with more than two operands are converted into multi-operand operators.
  - i.e., group nodes that have the same operator into one node with many children
  - This makes it more convenient and obvious that the operands can be combined in any order.
- This is especially important for joins as the order of joins may make a significant difference in the performance of the query.



Summary

- No transformation is always good at the LQP level
- Selections
  - push down tree as far as possible
  - if condition is an AND, split and push separately
  - sometimes need to push up before pushing down
- Projections
  - can be pushed down
  - new ones can be added (but be careful)
- Duplicate elimination
  - sometimes can be removed
- Selection/product combinations
  - can sometimes be replaced with join
- Many transformations lead to “promising” plans

Query Optimization

- Heuristics / Rules
  - Rewrite the query to remove stupid/inefficient things.
  - These techniques may need to examine catalog, but they do not need to examine data.
- Cost-based Search
  - Use a model to estimate the cost of executing a plan.
  - Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.

Outline - Query Processing

- Relational algebra level
  - transformations
  - good transformations
- Detailed query plan level
  - estimate costs
  - generate and compare plans

Evaluating Logical Query Plans

- The transformations discussed so far intuitively seem like good ideas
- But how can we evaluate them more scientifically?
- Estimate size of relations, also helpful in evaluating physical query plans

Steps in query optimization

1. We start with an initial logical query plan (obtained by transforming the parse tree into a relational algebra tree)
2. We transform this initial logical query plan into optimal logical query plan using Algebraic Laws
3. We choose the `best feasible algorithm` for `each relational operator` in the optimal logical query plan to obtain the optimal physical query plan

- we will learn to find optimal logical query plan

Comparing different logical query plans

- Before we can improve a query plan, we must have a **measure** to determine the **cost** of the **logical query plans**
- Measuring the cost of logical query plans
  1. The **ultimate cost measure** is the **execution time (#disk IOs performed)** of the **query plan**
     - However, **Execution time** is a measure used for implementation algorithms
       - i.e.: the physical query plan
     - We are comparing different **logical query plan**
  2. A **good approximation** of the **execution time (# disk IOs) measure** is **the size (# tuples)** of the result produced by the **operations**

Which query plan is better?

- The answer to the question is determined by:
  - The size (# tuples) of the intermediate result relations produced by each logical query plan
  - Because, the size (# tuples) will determine the number of disk IO performed by the relational operators (algorithms) further up in the query tree
- We need a method to compute (estimate) the size of the intermediate results of the relational operators on the logical query plan

Important fact:

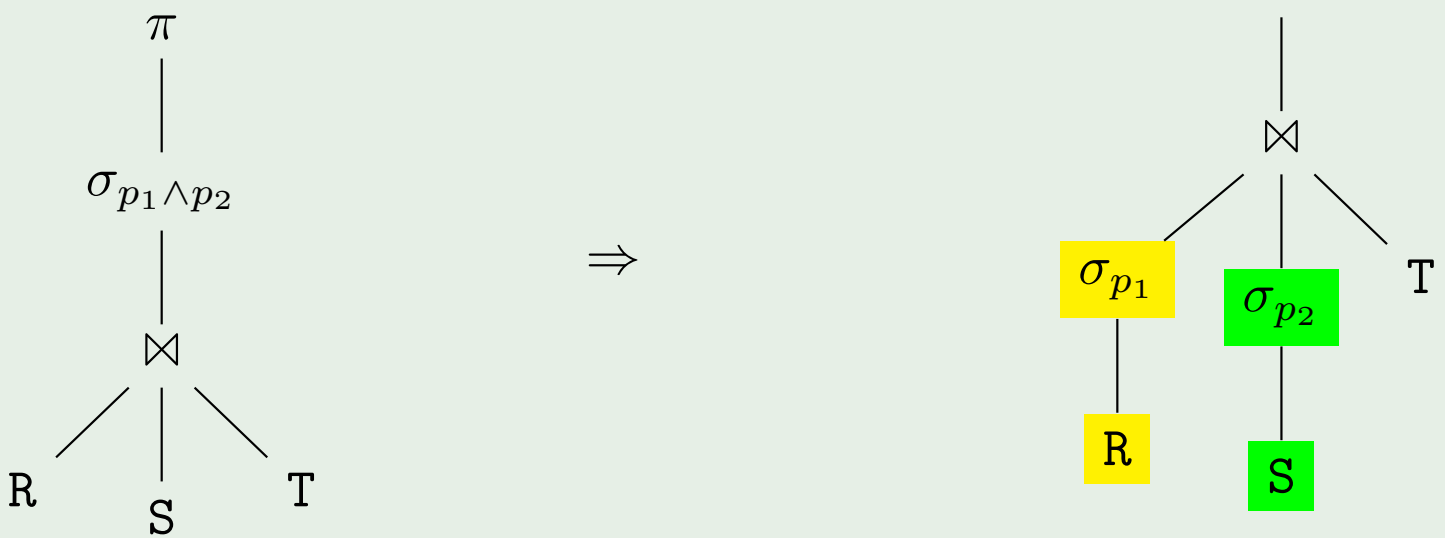
- The `size (# tuples)` of the result set of a `relational operator` is not dependent on the implementation algorithm.
- The differences between the algorithms are
  - `running time`
  - `memory requirement`
- ⇒ The `size` of the result in the `intermediate outputs` will
  - Depend only on the `order` of the operations in the `logical query plan`
  - Does not depend on `algorithm` used to compute the result
- *Thus, `# tuples` in the `intermediate result` of the `query plan` is a good estimate for the cost of the `logical query plan`*

Steps to find optimal (logical) query plan

1. Use the `relational algebra Laws` to find `least cost logical query plan` without considering the ordering of the join operations .

   i.e., the query plan has a smaller # tuples in the intermediate results

Example



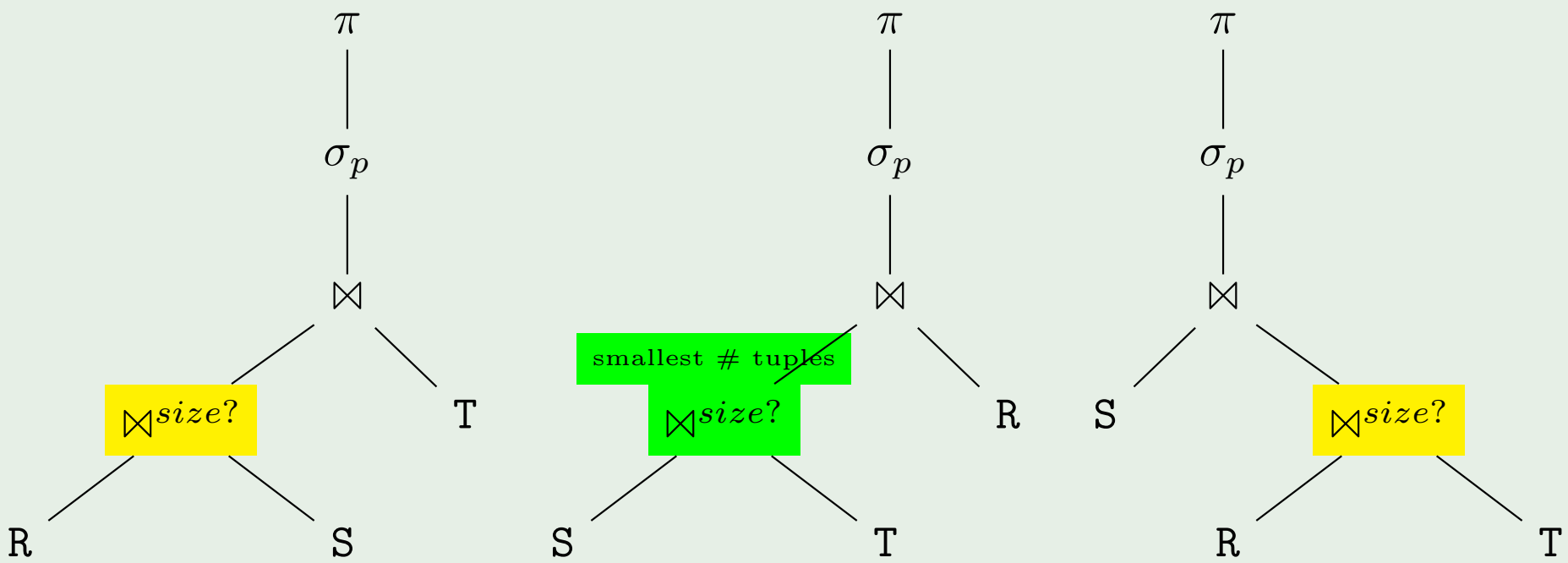
smaller intermediate results

Steps to find optimal (logical) query plan

2. If there are more than `2` input relations, then, find the `ordering` of the `join operations` that results in the `smallest # tuples` in the `intermediate results` in the `join tree`

Example (If we have 3 input relations:)

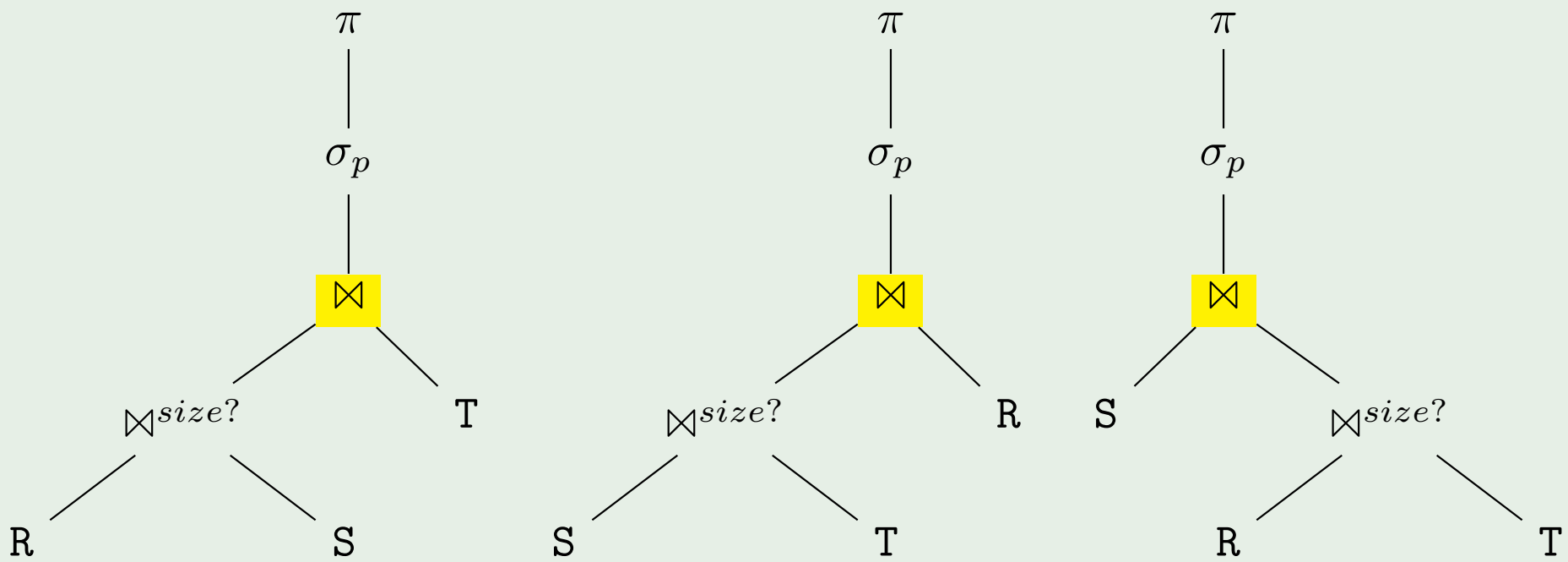
- We must find a join ordering that produce the smaller # tuples in the intermediate results in the join tree:



Steps to find optimal (logical) query plan

- Notice that the `end result` of all the joins are `equal`

Example (Continue)



- The only difference is the intermediate result sets

Estimating cost of query plan

- Estimates of cost are essential if the optimizer is to determine which of the many query plans is likely to execute fastest
  - Estimating size of results (Operation Cost)
  - Estimating # of IOs
- Note that the query optimizer will very rarely know the exact cost of a query plan because the only way to know is to execute the query itself!
  - Since the cost to execute a query is much greater than the cost to optimize a query, we cannot execute the query to determine its cost!
- How many tuples will be read/written?
- It is too expensive to run every possible plan to determine this information, so the DBMS need a way to derive this information

Prerequisite knowledge to performing query optimization

- The pre-requisite knowledge needed in finding the least cost logical query plan
  1. We need statistical information on the input relations to:
     - Estimate the size (# tuples) of the intermediate results in the relational algebra operations in the logical query plan

     The more detailed statistical information we have on the input relations, the more accurate (better) the estimates of the intermediate result sets
  2. We must also learn:
     - How to estimate the # tuples in the intermediate result sets using the statistical information

Statistics

- The DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog.
- Different systems update the statistics at different times.
- Commercial DBMS have way more robust and accurate statistics compared to the open source systems.
- These are estimates and thus the cost estimates will often be inaccurate.
- Manual invocations:

```
Postgres/SQLite: ANALYZE
Oracle/MySQL: ANALYZE TABLE
SQL Server: UPDATE STATISTICS
DB2: RUNSTATS
```

Statistics

- Statistics for base relations and attributes
- For each relation `R`, the DBMS maintains the following information:
  - T(R) = number of tuples in the relation `R`
  - S(R) = size (# of bytes) of a tuple of `R`
  - B(R) = number of blocks used to hold all tuples of relation `R`
  - V(R,A) = number of distinct values for attribute `A`
  - With the information listed above, the optimizer can derive the selection cardinality SC(A,R) statistics.
  - The selection cardinality is the average number of records with a value for an attribute `A` given.
    - Note that this assumes *data uniformity*
    - SC(A,R) = $\frac{T(R)}{V(R,A)}$
    - *average number of tuples of R that satisfy an equality condition on A*
    - `10,000` students, `10` colleges - how many students in `CS`?
- For now, we will not count record headers, but when storing tuples on blocks, the size of these must be added to the size of each tuple

Estimating result size (Operation Cost)

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

- `A`: 20 bytes `String`
- `B`: 4 bytes `integer`
- `C`: 8 bytes `date`
- `D`: 5 bytes `String`
- `T(R)` $= 5$
- `S(R)` $= 37$ bytes
- `V(R,A)` $= 3$, `V(R,B)` $= 1$, `V(R,C)` $= 5$, `V(R,D)` $= 4$

Estimating the (size of the) result set of a projection (π)

- Calculating the size of a relation after the projection operation is easy because we can compute it directly
- Recall: π does not remove duplicate values
- Every tuple changes size by a known amount.
- Estimating $\texttt{U} = \pi_{\texttt{A,B,...}}(\texttt{R})$
  - $\texttt{T(U)=T}\left(\pi_{\texttt{A,B,...}}(\texttt{R})\right) = \texttt{T(R)}$
    - Number of tuples is unchanged
  - $\texttt{S(U)=S}\left(\pi_{\texttt{A,B,...}}(\texttt{R})\right) = \texttt{sizeof(R.A)} + \texttt{sizeof(R.B),...}$
    - change the size of the tuple only, removing attributes (or adding new components that are combinations of other)

  $\Rightarrow \texttt{sizeof}\left(\texttt{U} = \pi_{\texttt{A,B,...}}(\texttt{R})\right) = \texttt{T(U)} * \texttt{S(U)}$

Estimating the (size of the) result set of a projection (π)

Example

- `R(A,B,C)` is a relation with `A` and `B` integers of `4` bytes each; `C` a string of `100` bytes.
- Tuple headers are `12` bytes.
- Blocks are `1024` bytes and have headers of `24` bytes.
- `T(R) = 10,000` and `B(R) = 1250`.
- How many blocks do we need to store $U = \pi_{\texttt{A,B}}(\texttt{R})$?

Answer

- `T(U) =T(R)=10,000`
- `S(U) =12+4+4=20 bytes`
- We can hence store $\frac{(1024-24)}{20} = 50$ tuples in one block.
- $\therefore$ `B(U)`$=\frac{T(U)}{50} = \frac{10{,}000}{50} = 200$ blocks
- This projection shrinks the relation by a factor slightly more than `6`

Result size estimation: $R_1 \times R_2$

- As with projections, we can exactly compute the size of a cartesian product ($\times$)
  - produces one tuple for each possible combination of each tuple in relation R_1 and R_2
    - $T(R_1 \times R_2) = T(R_1) * T(R_2)$
  - the size of each new tuple is the sum of the size of each original tuple
    - $S(R_1 \times R_2) = S(R_1) + S(R_2)$

$\Rightarrow$ $sizeof(R_1 \times R_2) = T(R_1 \times R_2) * S(R_1 \times R_2) = T(R_1) * T(R_2) * \big(S(R_1) + S(R_2)\big)$

Result size estimation: Selection σ_p(R)

- A Select (σ) reduces the number of tuples, but the size of each tuple is the same:
  - sizeof$\left(\sigma_{\mathtt{p}}(\mathtt{R})\right)$=T$(\sigma_{\mathtt{p}}(\mathtt{R}))$*S(R), where p is the condition selecting tuples
  - how to estimate the number of tuples depends on
    - value distribution of attribute A - we assume a uniform distribution where we use V(R,A) to estimate the number of tuples returned by the selection
    - condition upon which the tuples are selected
- General formula
  - T$\left(\sigma_p(\mathtt{R})\right)$=T(R)$\times$sel$_p$(R), where the selectivity (sel$_p$(R)) is the estimated fraction of tuples in R that satisfy predicate p
  - i.e., sel$_p$(R) is the estimated probability that a tuple in R satisfies p.

- How we calculate sel$_p$(R) depends on what p is.
- *Observe that the selectivity of a predicate is equivalent to the probability of that predicate. This allows probability rules to be applied in many selectivity computations.*

Result size estimation: 1. Equality selection: $\sigma_{\mathtt{A=c}}(\mathtt{R})$

- `Equality selection`: $\sigma_{\mathtt{A=c}}(\mathtt{R})$, for attribute `A` and constant `c`:
  - $sel_{\mathtt{A=c}}(\mathtt{R}) = \frac{1}{\mathtt{V(R,A)}}$
  - Intuition:
    - There are `V(R,A)` distinct `A`-values in `R`.
    - Assuming that `A`-values are uniformly distributed, the probability that a tuple has `A`-value `c` is $\frac{1}{\mathtt{V(R,A)}} = \frac{\mathtt{SC(R,A=c)}}{\mathtt{T(R)}}$

$\Rightarrow \mathtt{T}\left(\sigma_{\mathtt{A=c}}(\mathtt{R})\right) = \frac{\mathtt{T(R)}}{\mathtt{V(R,A)}}$, i.e., original number of tuples divided by number of different values of `A`

Result size estimation: 1. Equality selection: $\sigma_{\texttt{A=c}}(\texttt{R})$

Example

- `R(A,B,C)` is a relation.
- `T(R) = 10,000`
- `V(R,A) = 50`
- Estimate $\texttt{T}\big(\sigma_{\texttt{A=10}}(\texttt{R})\big)$
- $\texttt{T}\big(\sigma_{\texttt{A=10}}(\texttt{R})\big) = \frac{\texttt{T(R)}}{\texttt{V(R,A)}} = \frac{\texttt{10,000}}{\texttt{50}} = 200$

Result size estimation: 2. Inequality selection: $\sigma_{\mathtt{A<c}}(\mathtt{R})$

- `Inequality selection`: $\sigma_{\mathtt{A<c}}(\mathtt{R})$, for attribute `A` and constant `c`:
- $sel_{\mathtt{A<c}}(\mathtt{R}) = \frac{1}{3}$
  - estimate the fraction of `R` having tuples satisfying the condition
  - usually the fraction is small - one third of all tuples frequently used
    - Queries involving an inequality tend to retrieve a small fraction of the possible tuples (usually you ask about something that is true of less than half the tuples)

$$\Rightarrow \mathtt{T}\left(\sigma_{\mathtt{A<c}}(\mathtt{R})\right) = \frac{\mathtt{T(R)}}{3}$$

Example

- `R(A,B,C)` is a relation.
- `T(R) = 10,000`
- $\mathtt{T}\left(\sigma_{\mathtt{A<10}}(\mathtt{R})\right) = \mathtt{T(R)} \times \frac{1}{3} \approx 3334$

Estimate values in range

Example

- `R(A,B,C)` is a relation.
- `T(R) = 10,000`
- The `DBMS` statistics show that the values of the `A` attribute lie within the range `[8,57]`, uniformly distributed.
- Question: what would be a reasonable estimate of $sel_{\texttt{A<10}}(\texttt{R})$?

Answer

- We see that `57-8+1` different values of `A` are possible
- however only records with values `A=8` or `A=9` satisfy the filter `A<10`.
- Therefore, $sel_{\texttt{A<10}}(\texttt{R}) = \frac{2}{(57-8+1)} = \frac{2}{50} = 0.04$
- And hence, $\texttt{T}\left(\sigma_{\texttt{A<10}}(\texttt{R})\right) = \texttt{T(R)} \times sel_{\texttt{A<10}}(\texttt{R}) = 400$

Result size estimation: 3. Not-equal selection: $\sigma_{A \neq c}(R)$

- `Not-equal selection`: $\sigma_{A \neq c}(R)$, for attribute `A` and constant `c`:
  - can usually use $T(\sigma_{A \neq c}(R)) = T(R)$ for simplicity
  - more accurately, subtract a fraction $\frac{1}{V(R,A)}$
  - Fact:
    - $\sigma_{A \neq c}(R) \cup \sigma_{A=c}(R) = R$
    - $\Leftrightarrow \sigma_{A \neq c}(R) = R - \sigma_{A=c}(R)$
  - Therefore,
    - $sel_{A \neq c}(R) = \frac{V(R,A)-1}{V(R,A)}$
    - $T\left(\sigma_{A \neq c}(R)\right) = T(R) * \frac{V(R,A)-1}{V(R,A)}$

Result size estimation:4. Selection conditions with NOT

- Selection conditions with NOT/Negation query: $\sigma_{\neg p}(R)$
  - $sel_{\neg p}(R) = 1 - sel_{p}(R)$

  $\Rightarrow T(\sigma_{\neg p}(R)) = T(R) - T(\sigma_{p}(R))$
- *Observation: Selectivity* $\approx$ *Probability*

Result size estimation: 5. Selection using several conditions with AND

- `Selection using several conditions with AND`: $\sigma_{P_1 \wedge P_2}$(R)
  - treat selection as a cascade of several selections
    - Treat $\sigma_{P_1 \wedge P_2}$(R) as $\sigma_{P_1}\left(\sigma_{P_2}(\mathtt{R})\right)$
    - the order does not matter, treating this as $\sigma_{P_2}\left(\sigma_{P_1}(\mathtt{R})\right)$ gives the same results.
  - Estimated size is original size multiplied by the selectivity factor
    - $sel_{P_1 \wedge P_2}$(R)= sel_{P_1}(R) $*$ sel_{P_2}(R)

$\Rightarrow$ T($\sigma_{P_1 \wedge P_2}$(R)) = T(R) $*$ sel_{P_1}(R) $*$ sel_{P_2}(R)

- *This assumes that the predicates are independent.*

Result size estimation: 5. $\sigma_{P_1 \wedge P_2}(\mathrm{R})$

Example

- R(A,B,C) is a relation. T(R) = 10,000. V(R,A) = 50
- estimate the size of the result set U = $\sigma_{\mathtt{A=10} \wedge \mathtt{B<20}}(\mathtt{R})$

Answer

- $sel_{\mathtt{A=10}}(\mathtt{R}) = \frac{1}{50}$
- $sel_{\mathtt{B<20}}(\mathtt{R}) = \frac{1}{3}$
- T(U) = $sel_{\mathtt{A=10}} * sel_{\mathtt{B<20}} *$ T(R) = $\frac{1}{50} * \frac{1}{3} *$ 10,000 = 66.67

Result size estimation: 6. Selection using several conditions with OR

- Selection using several conditions with OR: $\sigma_{P_1 \vee P_2}(\mathtt{R})$
  - $P_1 \vee P_2 = \neg(\neg P_1 \wedge \neg P_2)$
  - Treat $\sigma_{P_1 \vee P_2}(\mathtt{R})$ as $\sigma_{\neg(\neg P_1 \wedge \neg P_2)}(\mathtt{R})$
  - $sel_{P_1 \vee P_2}(\mathtt{R}) = 1 - \Big(\big(1 - sel_{p_1}(\mathtt{R})\big) * \big(1 - sel_{P_2}(\mathtt{R})\big)\Big)$

$\Rightarrow \mathtt{T}\big(\sigma_{P_1 \vee P_2}(\mathtt{R})\big) = \mathtt{T(R)} * sel_{P_1 \vee P_2}(\mathtt{R})$

Example

- R(A,B,C) is a relation. T(R) = 10,000. V(R,A) = 50
- Estimate the size of the result set U = $\sigma_{\mathtt{A=10} \vee \mathtt{B<20}}(\mathtt{R})$

Answer

- $sel_{\mathtt{A=10}}(\mathtt{R}) = \frac{1}{50}$
- $sel_{\mathtt{B<20}}(\mathtt{R}) = \frac{1}{3}$
- $\mathtt{T(U)} = \mathtt{T(R)} * \big(1 - (1 - \frac{1}{50})(1 - \frac{1}{3})\big)$

Result size estimation: R ⋈ S

- In our size estimations for join, we will look at natural join (⋈), but other joins is managed similarly
  - equi-join as natural join
  - theta-joins as a cartesian product followed by a selection
- Assume: R(X,Y) and S(Y,Z), we join on Y: R(X,Y)⋈S(Y,Z).
- Question: Estimate the size of $\big($R(X,Y) ⋈ S(Y,Z)$\big)$
- The challenge is we do not know how the set of values of Y in R relate to the values of Y in S. There are some possibilities:
  - If Y attribute values in R(X,Y) and S(Y,Z) are disjoint - empty join:
    - T$\big($R(X,Y) ⋈ S(Y,Z)$\big)$ = 0
  - If Y attribute is a key in S and a foreign key of R, so each tuple in R joins with exactly one tuple in S:
    - T$\big($R(X,Y) ⋈ S(Y,Z)$\big)$ = T(R)
  - If almost every tuple in R and S has the same Y attribute value - combine all tuples of each relation:
    - T$\big($R(X,Y) ⋈ S(Y,Z)$\big)$ = T(R) * T(S)
- Range of T(R ⋈ S): 0 ≤ T(R ⋈ S) ≤ T(R) * T(S)

Result size estimation: R ⋈ S: Simplifying Assumptions

- For our calculations, we will make two assumptions:

1. containment of value sets assumption
   - An attribute `Y` in a relation `R(...,Y)` always takes on a **`prefix`** of a fixed list of values: y_1 y_2 y_3 y_4 ...
     - i.e., if attribute `Y` appears in several relations, the values are chosen from the front of a given list of values
     - thus, if `V(R,Y)` $\leq$ `V(S,Y)`, then every `Y-value` in `R` will match a `Y-value` in `S`

Example

Relations:

- `R(...,Y)`
- `S(...,Y)`
- `U(...,Y)`
- Attr values of `Y` in `R` can be one of: y_1 y_2 ..... y_R
- Attr values of `Y` in `S` can be one of: y_1 y_2 ..... y_S
- Attr values of `Y` in `U` can be one of: y_1 y_2 ... y_U

Result size estimation: R ⋈ S: Simplifying Assumptions

2. preservation of value sets assumption
   - The join operation `R(X,Y)` ⋈ `S(Y,Z)` will preserve all the possible values of the non-joining attributes.
   - i.e., non-join attributes will not lose any values from its set of possible values.
     - The attribute values taken on by `X` in `R(X,Y)` ⋈ `S(Y,Z)` and `R(X,Y)` are same. Thus, $\texttt{V}\left(\texttt{R(X,Y)} \bowtie \texttt{S(Y,Z)}, \texttt{X}\right) = \texttt{V(R,X)}$
     - The attribute values taken on by `Z` in `R(X,Y)` ⋈ `S(Y,Z)` and `S(Y,Z)` are same. Thus, $\texttt{V}\left(\texttt{R(X,Y)} \bowtie \texttt{S(Y,Z)}, \texttt{Z}\right) = \texttt{V(S,Z)}$

Result size estimation: R ⋈ S, joining on 1 attribute

- The size of R(X,Y)⋈S(Y,Z) in number of tuples can now be estimated as follows:
- Case 1: assume V(R,Y)≥V(S,Y)
  - The tuples in relations R and S take on the following attribute values for the Y attribute:
    - Attribute values of Y in R: y_1 y_2 ..... $y_{V(R,Y)}$
    - Attribute values of Y in S: y_1 y_2 ... $y_{V(S,Y)}$

    ⇒ every tuple t of S has a chance $\frac{1}{V(R,Y)}$ of joining with a given tuple of R.
  - There are T(R) tuples in R, therefore, one tuple t ∈ S will produce $\frac{T(R)}{V(R,Y)}$ number of matches
  - There are T(S) tuples in S, then estimated size of R⋈S is $\frac{T(R)\times T(S)}{V(R,Y)}$
- Case 2: if V(S,Y)≥V(R,Y)
  - estimated size of R⋈S is $\frac{T(R)\times T(S)}{V(S,Y)}$

Result size estimation: R ⋈ S, joining on 1 attribute

- In general, we divide by whichever of V(R,Y) and V(S,Y) is larger:
- $\mathtt{T(R \bowtie S)} = \frac{\mathtt{T(R)} \times \mathtt{T(S)}}{\max\left(\mathtt{V(R,Y)}, \mathtt{V(S,Y)}\right)}$

Result size estimation: R ⋈ S, joining on 1 attribute

Example

R(a,b)	S(b,c)	U(c,d)
T(R)=1000	T(S)=2000	T(U)=5000
V(R,b)=20	V(S,b)=50	
	V(S,c)=100	V(U,c)=500

- Find $T(R \bowtie S \bowtie U)$?

Result size estimation: R ⋈ S, joining on 1 attribute

Method 1: (ordering 1)

- $\texttt{R(a,b)}\bowtie\texttt{S(b,c)}\bowtie\texttt{U(c,d)} = \big(\texttt{R(a,b)}\bowtie\texttt{S(b,c)}\big)\bowtie\ \texttt{U(c,d)}$
- $\texttt{T}\big(\texttt{R(a,b)}\bowtie\texttt{S(b,c)}\big) = \frac{\texttt{T(R)}\times\texttt{T(S)}}{\max\big(\texttt{V(R,b)},\texttt{V(S,b)}\big)} = \frac{1000\times2000}{max\{20,50\}} = 40,000$
- $\texttt{T}\Big(\big(\texttt{R(a,b)}\bowtie\texttt{S(b,c)}\big)\bowtie\texttt{U(c,d)}\Big) = \frac{T\big(\texttt{R(a,b)}\bowtie\texttt{S(b,c)}\big)\times\texttt{T(U)}}{\max\Big(\texttt{V}\big(\texttt{R(a,b)}\bowtie\texttt{S(b,c)},\texttt{c}\big),\texttt{V(U,c)}\Big)}$
- From the `preservation of value sets` assumption, we have:
  $\texttt{V}\big(\texttt{R(a,b)}\bowtie\texttt{S(b,c)},\texttt{c}\big) = \texttt{V(S,c)}$, where $\texttt{V(S,c)}$=100 according to data
- $\therefore\ \texttt{T(R}\bowtie\texttt{S}\bowtie\texttt{U)} = \frac{T\big(\texttt{R(a,b)}\bowtie\texttt{S(b,c)}\big)\times\texttt{T(U)}}{\max\Big(\texttt{V}\big(\texttt{R(a,b)}\bowtie\texttt{S(b,c)},\texttt{c}\big),\texttt{V(U,c)}\Big)} = \frac{40{,}000\times5{,}000}{\max(100,500)} = 400{,}000$

Result size estimation: R ⋈ S, joining on 1 attribute

Method 2: (ordering 2)

- $\mathtt{R(a,b)} \bowtie \mathtt{S(b,c)} \bowtie \mathtt{U(c,d)} = \mathtt{R(a,b)} \bowtie \big(\mathtt{S(b,c)} \bowtie \mathtt{U(c,d)}\big)$
- $\mathtt{T}\big(\mathtt{S(b,c)} \bowtie \mathtt{U(c,d)}\big) = \frac{\mathtt{T(S)} \times \mathtt{T(U)}}{\max\big(\mathtt{V(S,c)}, \mathtt{V(U,c)}\big)} = \frac{2000 \times 5000}{max\{100,500\}} = 20,000$
- $\mathtt{T}\Big(\mathtt{R(a,b)} \bowtie \big(\mathtt{S(b,c)} \bowtie \mathtt{U(c,d)}\big)\Big) = \frac{\mathtt{T(R)} \times \mathtt{T}\big(\mathtt{S(b,c)} \bowtie \mathtt{U(c,d)}\big)}{\max\Big(\mathtt{V(R,b)}, \mathtt{V}\big(\mathtt{S(b,c)} \bowtie \mathtt{U(c,d)}, \mathtt{b}\big)\Big)}$
- From the `preservation of value sets` assumption, we have:
  $\mathtt{V}\big(\mathtt{S(b,c)} \bowtie \mathtt{U(c,d)}, \mathtt{b}\big) = \mathtt{V(S,b)}$, where $\mathtt{V(S,b)}$=50 according to data
- $\therefore \mathtt{T(R} \bowtie \mathtt{S} \bowtie \mathtt{U)} = \frac{\mathtt{T(R)} \times \mathtt{T}\big(\mathtt{S(b,c)} \bowtie \mathtt{U(c,d)}\big)}{\max\Big(\mathtt{V(R,b)}, \mathtt{V}\big(\mathtt{S(b,c)} \bowtie \mathtt{U(c,d)}, \mathtt{b}\big)\Big)} \frac{1,000 \times 20,000}{\max(20,50)} = 400,000$

Result size estimation: $\mathtt{R} \bowtie \mathtt{S}$, joining on 2 attributes

- Assume the relation schema $\mathtt{R(X,Y_1,Y_2)}$ and $\mathtt{S(Y_1,Y_2,Z)}$, i.e., we join on $\mathtt{Y_1}$ and $\mathtt{Y_2}$.
- General formula:

$$\mathtt{T}\big(\mathtt{R(X,Y_1,Y_2)} \bowtie \mathtt{S(Y_1,Y_2,Z)}\big) = \frac{\mathtt{T(R)} \times \mathtt{T(S)}}{\max\big(\mathtt{V(R,Y_1)},\mathtt{V(S,Y_1)}\big) \times \max\big(\mathtt{V(R,Y_2)},\mathtt{V(S,Y_2)}\big)}$$

Result size estimation: R ⋈ S, joining on 2 attributes

Example

R(a,b)	S(b,c)	U(c,d)
T(R)=1000	T(S)=2000	T(U)=5000
V(R,b)=20	V(S,b)=50	
	V(S,c)=100	V(U,c)=500

- Find T(R⋈S⋈U)?
- Computed using this ordering:

  $R(a,b) \bowtie S(b,c) \bowtie U(c,d) = \big(R(a,b) \bowtie U(c,d)\big) \bowtie S(b,c)$

- A join operation with no common attributes will degenerates into a Cartesian product
- Example: $R(a,b) \bowtie U(c,d) \Rightarrow R(a,b) \times U(c,d)$

Result size estimation: R ⋈ S, joining on 2 attributes

Method 3: (ordering 3)

- $\texttt{R(a,b)}\bowtie\texttt{S(b,c)}\bowtie\texttt{U(c,d)} = \big(\texttt{R(a,b)}\bowtie\texttt{U(c,d)}\big)\bowtie\texttt{S(b,c)}$
- $\texttt{T}\big(\texttt{R(a,b)}\bowtie\texttt{U(c,d)}\big)=\texttt{T}\big(\texttt{R(a,b)}\times\texttt{U(c,d)}\big)=1000\times 5000=5{,}000{,}000$
- $$\texttt{T}\Big(\big(\texttt{R(a,b)}\bowtie\texttt{U(c,d)}\big)\bowtie\texttt{S(b,c)}\Big) = \frac{\texttt{T}\big(\texttt{R(a,b)}\bowtie\texttt{U(c,d)}\big)\times\texttt{T(S)}}{\max\Big(\texttt{V}\big(\texttt{R(a,b)}\bowtie\texttt{U(c,d),b}\big),\texttt{V(S,b)}\Big)\times\max\Big(\texttt{V}\big(\texttt{R(a,b)}\bowtie\texttt{U(c,d),c}\big),\texttt{V(S,c)}\Big)}$$
- From the **preservation of value sets** assumption, we have:

  $\texttt{V}\big(\texttt{R(a,b)}\bowtie\texttt{U(c,d),b}\big)$ = $\texttt{V(R,b)}$, $\texttt{V(R,b)}=20$ according to data

  $\texttt{V}\big(\texttt{R(a,b)}\bowtie\texttt{U(c,d),c}\big)$= $\texttt{V(U,c)}$, $\texttt{V(U,c)}=500$ according to data
- $\therefore \texttt{T(R}\bowtie\texttt{S}\bowtie\texttt{U)} = \frac{5{,}000{,}000\times 2{,}000}{\max(20,50)\times\max(500,100)}=400{,}000$

The 2 assumptions (**containment and preservation of value sets**) allows us to re-order the **join-order** without affecting the size of the result set estimation

Estimating Join Sizes

- So far, we have only calculated the number of tuples, but the size of a join is given by
- `sizeof(R⋈S) = T(R⋈S) * S(R⋈S)`
- However, the size of the tuples from a join is dependent on which kind of join we perform, e.g.,
  - in a natural join, the join attributes only appear once
  - in a theta-join, all attributes from all relations appear

- thus, before calculating the total size in number of bytes, we must find the correct size of each tuple

Result size estimation: Size of a Union: $\mathtt{R} \cup \mathtt{S}$

- The number of tuples of a union ($\cup$) is dependent of whether it is a **set**- or **bag**-version:
- Bag-based:
  - the result is exactly the sum of the tuples of all the arguments:
    $\mathtt{T(R} \cup_{bag} \mathtt{S)} = \mathtt{T(R)+T(S)}$
- Set-based:
  - Range of the result set of $\mathtt{R} \cup_{set} \mathtt{S}$: $\mathtt{max}\big(\mathtt{T(R)},\mathtt{T(S)}\big) \leq \mathtt{T(R} \cup_{set} \mathtt{S)} \leq \mathtt{T(R)+T(S)}$
    - $\mathtt{max}\big(\mathtt{T(R)},\mathtt{T(S)}\big)$: $\mathtt{R} \subseteq \mathtt{S}$ or $\mathtt{S} \subseteq \mathtt{R}$
    - as bag-version if disjoint relations: $\mathtt{T(R)+T(S)}$: $\mathtt{R} \cap \mathtt{S} = \emptyset$
  - Recommended estimate for $\mathtt{R} \cup_{set} \mathtt{S}$ usually somewhere between sum of both and the number of the larger relation
  - may for example use:
    $$\mathtt{T(R} \cup_{set} \mathtt{S)} = \mathtt{max}\big(\mathtt{T(R)},\mathtt{T(S)}\big) + \tfrac{1}{2} \times \mathtt{min}\big(\mathtt{T(R)},\mathtt{T(S)}\big)$$
    i.e.: $\mathtt{maximum} + \frac{1}{2} \times \mathtt{(smaller\ size)}$

Result size estimation: Size of an Intersection: R ∩ S

- Range of the result set of R ∩ S
  - $0 \leq T(R \cap S) \leq \min\big(T(R), T(S)\big)$
    - 0: $R \cap S = \emptyset$ - if disjoint relations
    - $\min\big(T(R), T(S)\big)$: $R \subseteq S$ or $S \subseteq R$ - if one relation contains only a subset of the other
- Recommended estimate for R ∩ S usually somewhere in-between
  - may for example use average: $T(R \cap S) = \frac{1}{2} \times \min\big(T(R), T(S)\big)$

Result size estimation: Size of a Difference R−S

- Range of the result set of R−S:
  - $\max\{0, T(R)-T(S)\} \leq T(R-S) \leq T(R)$
    - T(R)−T(S): all tuples in S also is in R
    - T(R): $R \cap S = \emptyset$ - disjoint relations
- Recommended estimate for R−S
  - usually somewhere in-between
  - may for example use: $T(R-S) = T(R) - \frac{1}{2} \times T(S)$
  - Note: if $T(R) - \frac{1}{2} \times T(S) \leq 0$, then $T(R-S) = 0$ (estimate)

Result size estimation: $\delta(R,A)$

- The number of tuples of a duplicate elimination (δ) is the same as the number of distinct tuples
- Range of the result set of $\delta(R,A)$
  - $1 \leq T\big(\delta(R,A)\big) \leq T(R)$
    - 1: all tuples have same attribute value
    - T(R): all tuples have different attribute values
- Recommended estimate for $\delta(R,a_1,a_2,\ldots,a_n)$:
  - given $V(R,a_i)$ for all n attributes, the maximum number of different tuples are $T\big(\delta(R,a_1,a_2,\ldots,a_n)\big) = V(R,a_1) * V(R,a_2) *\ldots* V(R,a_n)$
  - let estimated number of tuples be the smaller of this number and the number of tuples in the relation
  - Otherwise, if no statistics available, then we use this estimate: $T\big(\delta(R,a_1,\ldots,a_n)\big) = \min\left(\left[(\frac{1}{2})^n \times T(R)\right], V(R,a_1)*\ldots*V(R,a_n)\right)$
  - Reason
    - $V(R,a_1)*\ldots* V(R,a_n)$: The upper bound limit on the number of distinct tuples that could exist
    - $(\frac{1}{2})^n * T(R)$: The size can be as small as 1 or as big as T(R)

Result size estimation: $\gamma_L(\mathtt{R})$

- The number of tuples of a grouping (γ) is the same as the number of groups
- Range of the result set of $\gamma_L(\mathtt{R})$
  - $\mathtt{1} \leq \mathtt{T}\big(\gamma_L(\mathtt{R})\big) \leq \mathtt{T(R)}$
    - `1: all tuples have same attribute value`
    - `T(R): all tuples have different attribute values for attribute L`
- Recommended estimate for $\mathtt{T}\big(\gamma_L(\mathtt{R})\big)$
  - If the database maintains statistics on the attribute values:
    $\mathtt{T}\big(\gamma_L(\mathtt{R})\big) = \mathtt{V(R,L)}$
  - If no statistics available, then we use this estimate:
    $\mathtt{T}\big(\gamma_L(\mathtt{R})\big) = (\frac{1}{2})^n \times \mathtt{T(R)}$, where n = number of attributes in the attribute list `L`

Histogram

- Modern DBMS often maintain a number of statistical information to help the Query Optimizer decide on the optimal query plan
- Better selectivity estimates are possible if we have more detailed statistics
- A `DBMS` typically collects `histograms` that detail the distribution of values.
- A `histogram` can be of two types:
  - `equi-width histogram`:
    - All buckets contain the same number of values
    - Easy, but inaccurate
    - The range of values is divided into equal-sized subranges.
  - `equi-depth histograms`: (used by most DBMS)
    - All buckets contain the same number of tuples
    - Better accuracy, need to sort data to compute
    - The sub ranges are chosen in such a way that the number of tuples within each sub range is equal.
- Such `histograms` are only available for base relations, however, not for sub-results.
- If a histogram is available for the attribute `A`, the number of tuples can be estimated with more accuracy.

Histogram to estimate result set of $\sigma_{A=c}(\mathrm{R})$

- If a histogram is available for the attribute `A`, the number of tuples can be estimated with more accuracy.
- The range in which the value `c` belongs is first located in the `histogram`.
- `|B|`: number of values per bucket (# distinct values appearing in that range)
- `#B`: number of records in bucket
- $\mathtt{T}\left(\sigma_{A=c}(\mathtt{R})\right) = \frac{\#B}{|B|}$

Histogram to estimate result set of $\sigma_{A=c}(\mathrm{R})$

Example

- R(A,B,C) is a relation.
- `T(R) = 10,000`
- `V(R,A) = 50`
- Estimate $\mathtt{T}\big(\sigma_{A=10}\mathtt{(R)}\big)$
- The `DBMS` has collected the following `equi-width` histogram on `A`

range	[1,10]	[11,20]	[21,30]	[31,40]	[41,50]
tuples in range	50	2000	2000	3000	2950

- $\mathtt{T}\big(\sigma_{A=10}\mathtt{(R)}\big) = \frac{\#B}{|B|} = \frac{50}{10} = 5$

Join Size using Histograms

- R ⋈ S
- Use:
- $\mathtt{T(R \bowtie S)} = \dfrac{\mathtt{T(R)} \times \mathtt{T(S)}}{\max\left(\mathtt{V(R,A)}, \mathtt{V(S,A)}\right)}$
- Apply for each bucket

Join Size using Histograms

- V(R,A) = V(S,A) = bucket size |B|
- $\Rightarrow$ T(R$\bowtie$S) = $\sum_{buckets} \frac{\#B(R) \times \#B(S)}{|B|}$

Advanced Techniques

- Wavelets
- Approximate Histograms
- Sampling Techniques
- Compressed Histograms

Summary

- As should be clear by now, result size estimation is not an exact art
- To estimate the size of the intermediate relations, we have used parameters like `T(R)` and `V(R,A)`
- The DBMS keeps statistics from previous operations to be able to provide such parameters
- However, computing statistics are expensive and should be recomputed periodically only:
  - statistics usually have few changes over a short time
  - even inaccurate statistics are useful
  - statistics recomputation might be triggered after some period of time or after some number of updates

Outline

- Estimating cost of query plan
  - Estimating size of results
  - Estimating # of IOs (next)
  - Operator Implementations (next)
- Generate and compare plans