CS525: Advanced Database Organization

Notes 6: Query Processing
Logical Optimization

Yousef M. Elmehdwi

Department of Computer Science

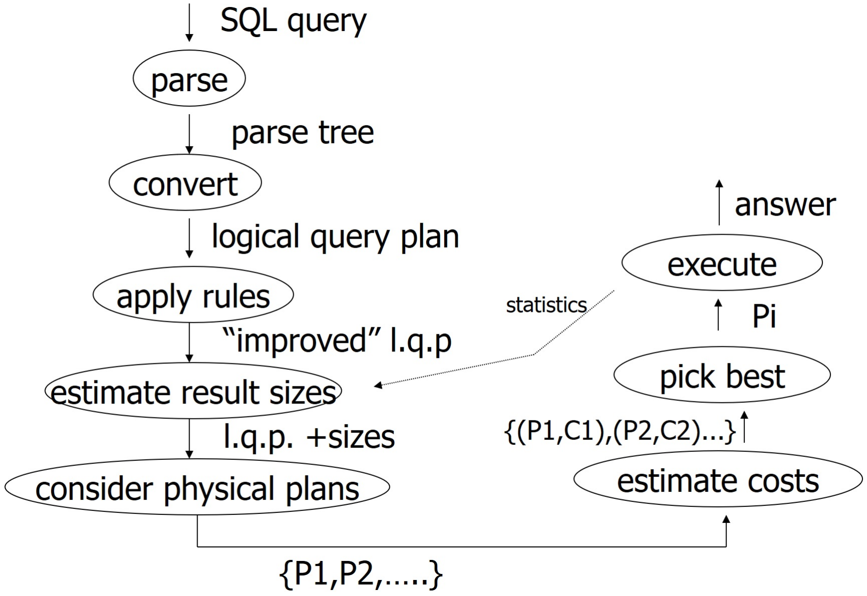
Illinois Institute of Technology

yelmehdwi@iit.edu

October 11, 2018

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

Basic Steps in Processing an SQL Query



Where we are

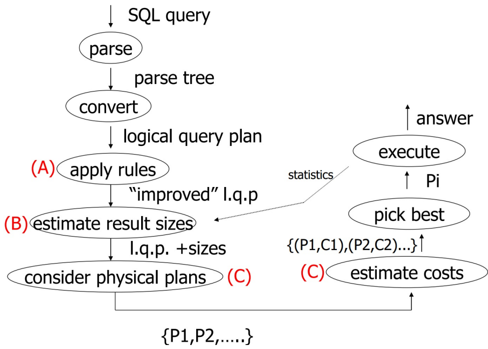
- `SQL` $\Rightarrow$ `parse tree` $\Rightarrow$ `expression of relational algebra (initial logical query plan)`
- Today: consider ways of transformations to improve the query plan
    - Algebraic laws for improving query plans

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 often possible to optimize the logical query tree by applying relational algebra laws to convert the original tree into a more efficient logical query tree.
- Next we'll survey some of these laws
- Optimizing a logical query tree using relational algebra laws is called heuristic optimization

Query Optimization

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



Relational Algebra Optimization

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

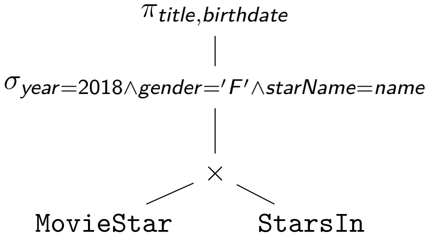
Query Equivalence

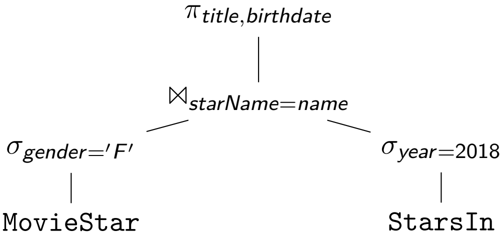
- Two queries q_1 and q_2 are equivalent:
  - If for every database instance `I` (contents of all the tables)
  - Both queries have the same result
- $q_1 \equiv q_2$ `iff` $\forall$ `I:` q_1`(I) =` q_2`(I)`

Query Equivalence

StarsIn`(title.  year, startName)`

MovieStar`(name, address, gender, birthdate)`





Rules: Natural joins & cross products & union

- `Join` ($\bowtie$) is commutative: $\texttt{R} \bowtie \texttt{S} = \texttt{S} \bowtie \texttt{R}$
- `Join` ($\bowtie$) is associative: $(\texttt{R} \bowtie \texttt{S}) \bowtie \texttt{T} = \texttt{R} \bowtie (\texttt{S} \bowtie \texttt{T})$

Note

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

$$\begin{tikzcd}[column sep=small]
 & & \bowtie \arrow[dl, no head] \arrow[dr, no head] & & & & & \bowtie \arrow[dl, no head] \arrow[dr, no head] & & \\
 & \bowtie \arrow[dl, no head] \arrow[dr, no head] & & \mathtt{T} & \equiv & & \mathtt{R} & & \bowtie \arrow[dl, no head] \arrow[dr, no head] & \\
\mathtt{R} & & \mathtt{S} & & & & & \mathtt{S} & & \mathtt{T}
\end{tikzcd}$$

- Different ordering in the execution of the $\bowtie$ operation 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` $\bowtie$ `operations`

Rules: Natural joins & cross products & union

- `Cross product` $(\times)$ is commutative: $\texttt{R} \times \texttt{S} = \texttt{S} \times \texttt{R}$
- `Cross product` $(\times)$ is associative: $(\texttt{R} \times \texttt{S}) \times \texttt{T} = \texttt{R} \times (\texttt{S} \times \texttt{T})$

Rules: Natural joins & cross products & union

- `Union` `(`$\cup$`)` is commutative: `R` $\cup$ `S` $=$ `S` $\cup$ `R`
- `Union` `(`$\cup$`)` is associative: `(R` $\cup$ `S)` $\cup$ `T` $=$ `R` $\cup$ `(S` $\cup$ `T)`

Rules: Selections

- Selections usually reduce the size of the relation (decrease the number of rows)
- Usually good to do selections early, i.e., “push them down the tree”
  - Perform selection as early as possible (but take existing indexes on relations into account)
- Also can be helpful to break up a complex selection into parts

Rules: Selections

- Selection is `idempoten`. (multiple applications of the same selection have no additional effect beyond the first one)
    - $\sigma_p(\mathtt{R}) = \sigma_p\sigma_p(\mathtt{R})$
- Select operations are `commutative`. (the order selections are applied in has no effect on the eventual result)
    - $\sigma_p\sigma_q(\mathtt{R}) = \sigma_q\sigma_p(\mathtt{R})$

Rules: Selection Splitting

- The selection condition involving conjunction of two or more predicates can be deconstructed into a sequence of individual select operations.
  - $\sigma_{p_1 \wedge p_2}(\mathtt{R}) = \sigma_{p_1}\left(\sigma_{p_2}(\mathtt{R})\right) = \sigma_{p_2}\left(\sigma_{p_1}(\mathtt{R})\right)$
  - This transformation is called cascading of select operator.

Bags vs. Sets

- R = {a,a,b,b,b,c}
- S = {b,b,c,c,d}
- R ∪ S = ?
- Option 1: `SUM`
  - R ∪ S = {a,a,b,b,b,b,b,c,c,c,d}
- Option 2: `MAX`
  - R ∪ S = {a,a,b,b,b,c,c,d}

“SUM” is implemented

- Use `‘‘SUM’’` option for bag unions
- CAREFUL!. Some rules cannot be used for bags

Laws for Bags and Sets Can Differ

- Example of an Algebraic Law that holds for set, but not for bags
- We know from `Set Theory` that

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

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

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

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

Rules σ, $\cup$, $-$ combined

- Push selections through the binary operators: product, union, intersection, difference, and join.
  1. Must push selection to both arguments:
     - $\sigma_p(\texttt{R} \cup \texttt{S}) = \sigma_p(\texttt{R}) \cup \sigma_p(\texttt{S})$
  2. Must push to first argument, optional for second:
     - $\sigma_p(\texttt{R - S}) = \sigma_p(\texttt{R}) \texttt{ - } \sigma_p(\texttt{S})$
     - $\sigma_p(\texttt{R - S}) = \sigma_p(\texttt{R}) \texttt{ - S}$
  3. Push to at least one argument with all attributes mentioned in `p`:
     - 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`

Rules: Selections

- If the condition `p` in σ_p(`R ∩ S`) is compound (`p = p`$_1$ `and p`$_2$), to split `p` up, we can use:
  - $\sigma_{p_1 \wedge p_2}(\texttt{R}) = \sigma_{p_1}\big(\sigma_{p_2}(\texttt{R})\big) = \sigma_{p_2}\big(\sigma_{p_1}(\texttt{R})\big)$

Example

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

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

Rules σ, $\bowtie$ combined

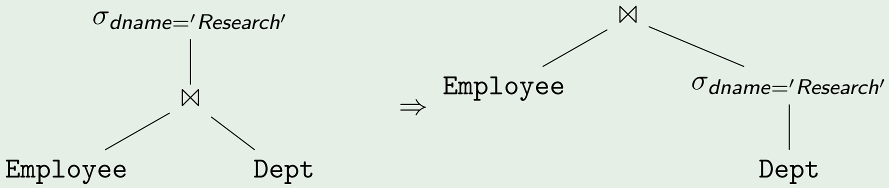
- If the selection condition p involves only the attributes of R and q involves the attributes of S, then the select operation distributes.
  - $\sigma_{p\wedge q}(\mathtt{R} \bowtie \mathtt{S}) = \sigma_p(\mathtt{R}) \bowtie \sigma_q(\mathtt{S})$
- Let
  - p = predicate with only R attributes
  - q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

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

Pushing Selections

Example

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



Pushing Selections

Example

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

  $= \sigma_{fname='John'}\big(\sigma_{dname='Research'}(\texttt{Employee} \bowtie \texttt{Dept})\big)$

  $= \sigma_{fname='John'}\big(\texttt{Employee} \bowtie \sigma_{dname='Research'}(\texttt{Dept})\big)$

  $= \sigma_{fname='John'}(\texttt{Employee}) \bowtie \sigma_{dname='Research'}(\texttt{Dept})$

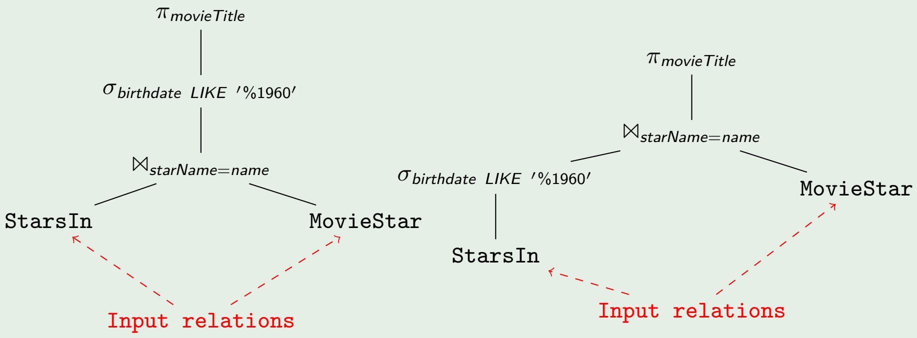
Commonly used query optimization technique involving σ

- 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



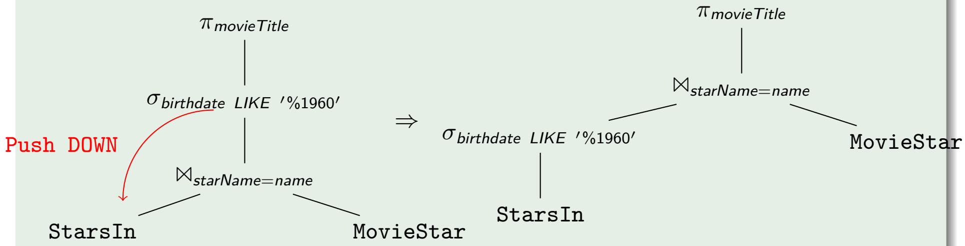
$\pi_{movieTitle}$
$\sigma_{birthdate\ LIKE\ '\%1960'}$
$\bowtie_{starName=name}$
StarsIn
MovieStar
Input relations
$\pi_{movieTitle}$
$\bowtie_{starName=name}$
$\sigma_{birthdate\ LIKE\ '\%1960'}$
StarsIn
MovieStar
Input relations

- The input relation of $\bowtie$ in the second case $\sigma_{birthday\ LIKE\ '\%1960'}$(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


$\pi_{movieTitle}$
$\sigma_{birthdate\ LIKE\ '\%1960'}$
Push DOWN
$\bowtie_{starName=name}$
StarsIn
MovieStar
$\Rightarrow$
$\pi_{movieTitle}$
$\bowtie_{starName=name}$
$\sigma_{birthdate\ LIKE\ '\%1960'}$
MovieStar
StarsIn

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

Example

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

- View:

```
CREATE VIEW
MoviesOf1996  AS {
  SELECT  *
  FROM   Movies
  WHERE  year = 1996
 }
```

- Corresponding logical query plan

$$\begin{tikzcd}[row sep=large] \sigma_{year=1996} \arrow[d, no head] \\ \text{Movies} \end{tikzcd}$$

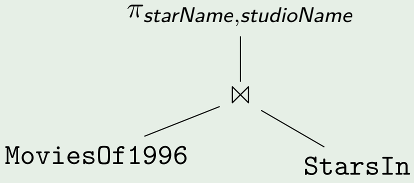
Note: “push select down” query optimization technique

Example (Continue)

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

```
SELECT    starName,   studioName
FROM      MoviesOf1996,  StarsIn
WHERE     MoviesOf1996.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:

$$\begin{tikzcd}
 & \pi_{starName,studioName} \arrow[d, dash] & \\
 & \bowtie \arrow[dl, dash] \arrow[dr, dash] & \\
\sigma_{year=1996} \arrow[d, dash] & & \texttt{StarsIn} \\
\texttt{Movies} & &
\end{tikzcd}$$

- However, the optimal query plan is as follows:

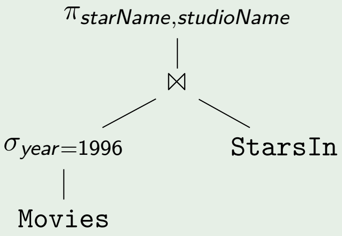
$$\begin{tikzcd}
 & \pi_{starName,studioName} \arrow[d, dash] & \\
 & \bowtie \arrow[dl, dash] \arrow[dr, dash] & \\
\sigma_{year=1996} \arrow[d, dash] & & \sigma_{year=1996} \arrow[d, dash] \\
\texttt{Movies} & & \texttt{StarsIn}
\end{tikzcd}$$

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

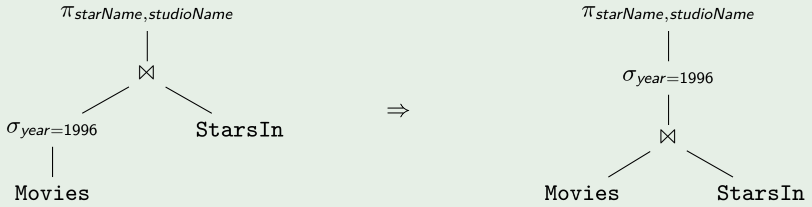
- Query plan after incorporating the virtual table query:



Amendment to the simple query optimization technique

Example (Continue)

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

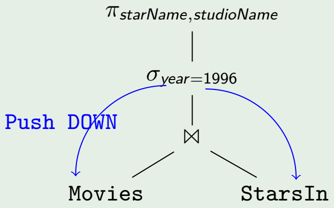


Amendment to the simple query optimization technique

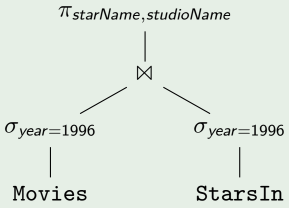
Example (Continue)

- 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 $\sigma_{year=1996}$ operation **down the tree**



$\Rightarrow$

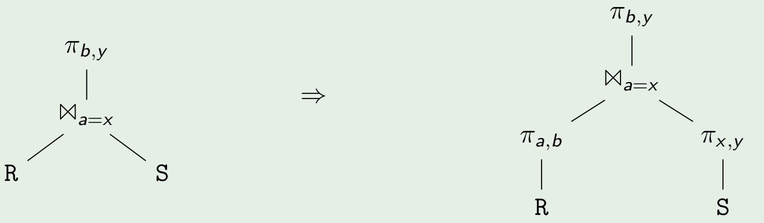


Laws Involving Projections: Use of π in query optimization

- The `projection` operation π can remove unnecessary attributes from intermediate results
- Common use the `project` operation π in query optimization:
  - The `projection` operator π can be added anywhere in the relational algebra expression (= logical query plan/tree), as long as:
    - π will only eliminate attributes that are not used by an operator that is located high up the tree

Example

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



Laws Involving Projections

- Consider adding in additional projections
- Adding a projection lower in the tree can improve performance, since often tuple size is reduced
    - Usually not as helpful as pushing selections down
- If a projection is inserted in the tree, then none of the eliminated attributes can appear above this point in the tree

Rules: Projections

- If a query contains a sequence of project operations, only the final operation is needed, the others can be omitted.
  - $\pi_{L_1}\Big(\pi_{L_2}\big(\ldots\big(\pi_{L_n}(\texttt{R})\big)\ldots\big)\Big) = \pi_{L_1}(\texttt{R})$, 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:

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

Rules: Projections

Let:

- $\texttt{X}$ = set of attributes
- $\texttt{Y}$ = set of attributes
- $\texttt{XY = X} \cup \texttt{Y}$
- $\pi_{XY}(\texttt{R}) =$ ~~$\pi_X\left(\pi_Y(\texttt{R})\right)$~~

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 or 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
  - $\mathtt{x}$ = subset of `R` attributes
  - $\mathtt{z}$ = attributes in predicate `p` (subset of `R` attributes)
  - $\pi_x\big(\sigma_p(\mathtt{R})\big) \; = \; \sigma_p\big($[redacted]$(\mathtt{R})\big)$

Rules: π, σ combined

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

Rules: π, $\bowtie$ combined

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

Rules: π, $\bowtie$ combined

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

Push Projection Below Selection?

- Is it a good idea?

Example

- Relations:
  - **StarsIn**`(title, movieYear, starName, birthday)` // Movie stars
  - `SELECT starName FROM StarsIn WHERE movieYear = 1996;`

$$\begin{tikzcd}
\pi_{starName} \arrow[d, no head] \\
\sigma_{movieYear=1996} \arrow[d, no head] \\
\texttt{StarsIn}
\end{tikzcd}$$

$$\begin{tikzcd}
\pi_{starName} \arrow[d, no head] \\
\sigma_{movieYear=1996} \arrow[d, no head] \\
\pi_{starName,movieYear} \arrow[d, no head] \\
\texttt{StarsIn}
\end{tikzcd}$$

`Extra work to scan through StarsIn twice`

Rules: Joins and Products

- Laws by definition: These are not really laws, but they are the definition of the $\bowtie$ operator:
  - $(\texttt{R} \bowtie_p \texttt{S}) = \sigma_p(\texttt{R} \times \texttt{S})$ (theta join)
  - $(\texttt{R} \bowtie \texttt{S}) = \pi_L\big(\sigma_p(\texttt{R} \times \texttt{S})\big)$ (natural join)
    - where `p` equates same-name attributes in `R` and `S`, and L includes all attributes of `R` and `S` dropping duplicates
- To improve a logical query plan, replace a product followed by a selection with a join
  - Join algorithms are usually faster than doing product followed by selection on the (very large) result of the product

Rules: Duplicate Elimination

- Moving 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: $\delta(\mathtt{R} \times \mathtt{S}) = \delta(\mathtt{R}) \times \delta(\mathtt{S})$
  - 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. Therefore, 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**
  - $\delta(R \cup_{bag} S)$ has one copy of t
  - $\delta(R) \cup_{bag} \delta(S)$ has two copies of t
- **Bag difference**
  - $\delta(R - S)$ has one copy of t
  - $\delta(R) - \delta(S)$ has no copies of t
- **Bag projection**
  - $\delta\big(\pi_a(T)\big) = \{1\}$
  - $\pi_a\big(\delta(T)\big) = \{1,1\}$

Rules: Grouping and Aggregation

- The grouping operator only interact with very few relation algebra operations:
  1. γ_L produces a set, therefore the δ operation is unnecessary
     - $\delta\Big(\gamma_L(\mathtt{R})\Big) \;=\; \gamma_L(\mathtt{R})$
  2. You can project out some attributes as long as you keep the grouping attributes:
     - $\gamma_L(\mathtt{R}) \;=\; \gamma_L\Big(\pi_M(\mathtt{R})\Big)$, where M must contain all attributes used by γ_L
  3. The aggregate functions `max` and `min` can tolerate removal of duplicates:
     - $\gamma_L(\mathtt{R}) \;=\; \gamma_L\Big(\delta(\mathtt{R})\Big)$, where γ_L =max or min
     - `max(5,5,3)=max(5,3)`

More transformations

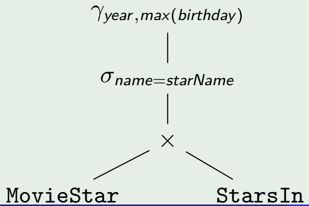
- Eliminate common sub-expressions
- Detect constant expressions

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

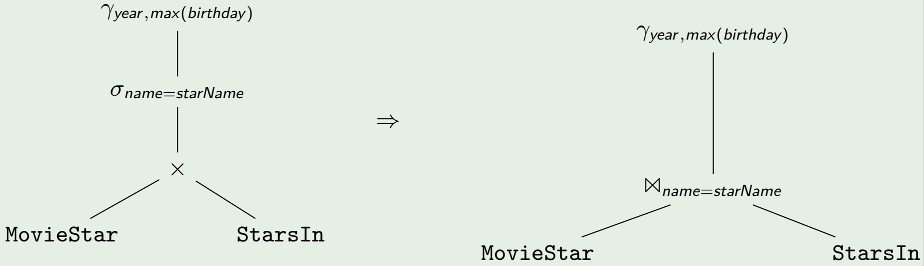
-
  ```
  SELECT      year, max(birthday)
  FROM        movieStar, StarsIn
  WHERE       name = starName
  GROUP BY    year
  ```
- The initial logical query plan is as follows:



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*"

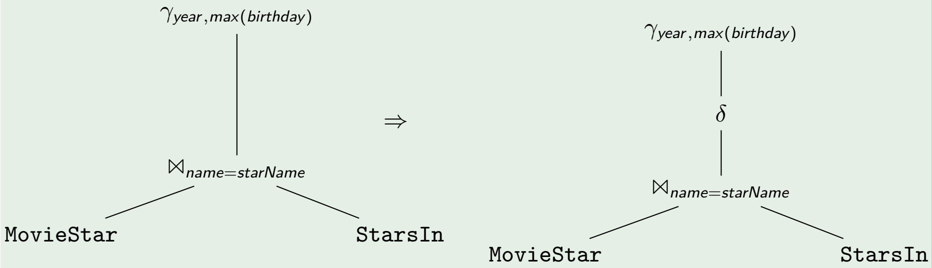


$\gamma_{year,max(birthday)}$
$\sigma_{name=starName}$
$\times$
MovieStar
StarsIn
$\Rightarrow$
$\gamma_{year,max(birthday)}$
$\bowtie_{name=starName}$
MovieStar
StarsIn

Applying the Algebraic laws for query optimization

Example (Continue)

- Apply: $\gamma_L(\texttt{R}) \; = \; \gamma_L\big(\delta(\texttt{R})\big)$, where $\gamma_L = \text{max} \;\; \textit{or} \;\; \text{min}$

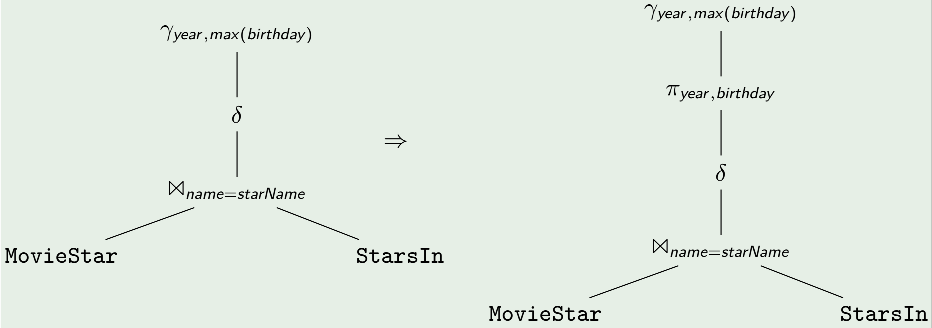


$\gamma_{year,max(birthday)}$
$\bowtie_{name=starName}$
MovieStar
StarsIn
$\Rightarrow$
$\gamma_{year,max(birthday)}$
δ
$\bowtie_{name=starName}$
MovieStar
StarsIn

Applying the Algebraic laws for query optimization

Example (Continue)

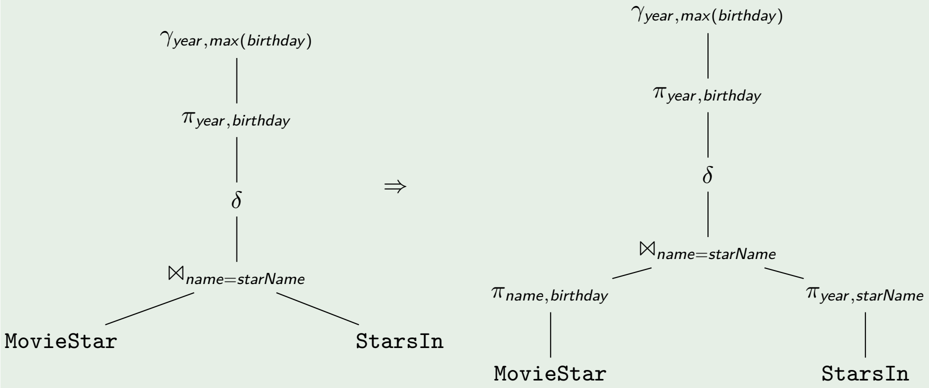
- Optionally, you can insert a projection at the top


$\gamma_{year,max(birthday)}$
δ
$\bowtie_{name=starName}$
MovieStar
StarsIn
$\Rightarrow$
$\gamma_{year,max(birthday)}$
$\pi_{year,birthday}$
δ
$\bowtie_{name=starName}$
MovieStar
StarsIn

Applying the Algebraic laws for query optimization

Example (Continue)

- Optionally, you can insert a couple of projections at the bottom



Heuristic Query Optimization

- 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

1. Deconstruct conjunctive selections into a sequence of single selection operations.
2. Move selection operations down the query tree for the earliest possible execution.
3. Replace Cartesian product operations that are followed by a selection condition by join operations.
4. Execute first selection and join operations that will produce the smallest relations.
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed

Summary

- No transformation is always good at the l.q.p 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

Outline - Query Processing

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

Canonical Query Trees

- A canonical logical query tree is a logical query tree where all associative and commutative operators with more than two operands are converted into multi-operand operators.
  - 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

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

Overview of the Query Optimization process

- `Logical query plan`
  - a `query tree` where the nodes consist of relational algebra operators
- `Physical query plan`
  - a `query tree` where the nodes consist of relational algebra algorithms
    - There are different (implementation) algorithms for a relational algebra operator
    - Each with different cost (# disk IOs) and memory requirement

Query Optimization Physical Query Plan

- `Physical query plan` is derived from a logical query plan by:
  1. Selecting an order and grouping for operations like joins, unions, and intersections.
  2. Deciding on an algorithm for each operator in the logical query plan.
     - e.g. For joins: Nested-loop join, sort join or hash join
  3. Adding additional operators to the logical query tree such as sorting and scanning that are not present in the logical plan.
  4. Determining if any operators should have their inputs materialized for efficiency.
- Whether we perform cost-based or heuristic optimization, we eventually must arrive at a physical query tree that can be executed by the evaluator.

Query Optimization Heuristic versus Cost Optimization

- To determine when one physical query plan is better than another, we must have an estimate of the cost of the plan.
- Heuristic optimization is normally used to pick the best logical query plan.
- Cost-based optimization is used to determine the best physical query plan given a logical query plan.
- Note that both can be used in the same query processor (and typically are). Heuristic optimization is used to pick the best logical plan which is then optimized by cost-based techniques

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 let us tell the `difference (in cost)` between the different `logical query plans`
- Measuring the cost of logical query plans
  1. The `ultimate cost measure` is `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 `excution 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

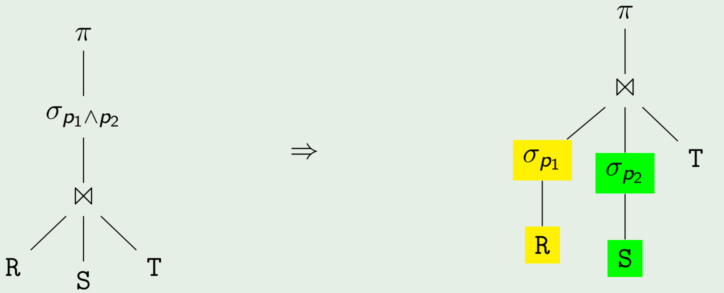
Note

- 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 (the query plan has a smaller # tuples in the intermediate results)

Example

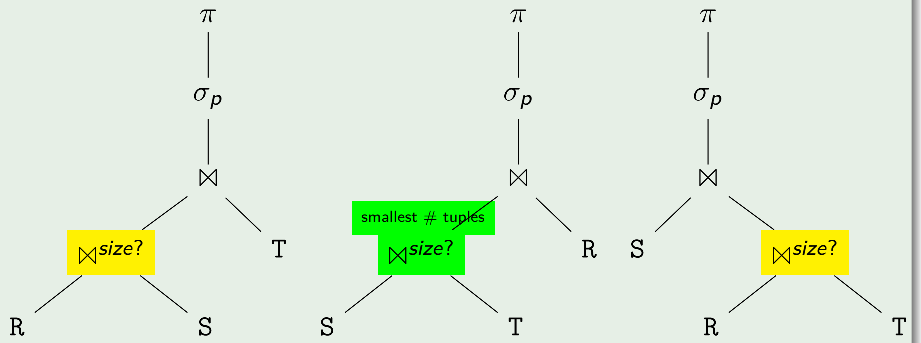


smaller intermediate results

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`

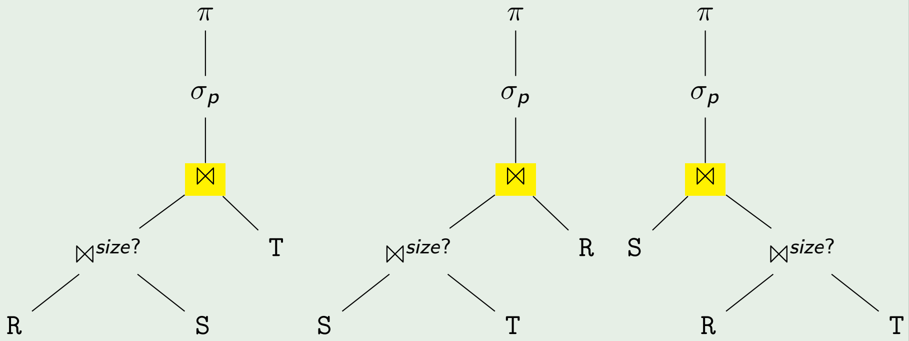
Example (Continue)



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!
- It is important to be able to estimate the cost of a query plan without executing it based on statistics and general formulas.

Estimating result size (Operation Cost)

- Statistics/Information about relations and attributes
  - **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`

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
- This can be exactly computed
- Every tuple changes size by a known amount.
- Estimating `S = ` π_a`(R)`
  - $\mathtt{T}\big(\pi_a(\mathtt{R})\big) = \mathtt{T}(\mathtt{R})$
    - Number of tuples is unchanged

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 $U = \pi_{A,B}(\mathtt{R})$?

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: $R_1 \times R_2$

- $T(R_1 \times R_2) = T(R_1) \times T(R_2)$
- $S(R_1 \times R_2) = S(R_1) + S(R_2)$

Result size estimation: Selection $\sigma_p(\mathtt{R})$ with $\mathtt{p}$ a predicate

- Generally reduce the number of tuples, although the sizes of tuples remain the same
- General formula
  - $\mathtt{T}\big(\sigma_p(\mathtt{R})\big) = \mathtt{T(R)} \times \mathtt{sel}_p\mathtt{(R)}$, where $\mathtt{sel}_p\mathtt{(R)}$ is the estimated fraction of tuples in $\mathtt{R}$ that satisfy predicate $\mathtt{p}$
  - i.e., $\mathtt{sel}_p\mathtt{(R)}$ is the estimated probability that a tuple in $\mathtt{R}$ satisfies $\mathtt{p}$.
- How we calculate $\mathtt{sel}_p\mathtt{(R)}$ depends on what $\mathtt{p}$ is.

Result size estimation: 1. $\sigma_{A=c}$(R) with c a constant

- $sel_{A=c}(\mathtt{R}) = \frac{1}{V(R,A)}$
- 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 $\frac{1}{V(R,A)}$
- $\therefore \mathtt{T}\big(\sigma_{A=c}(\mathtt{R})\big) = \frac{T(R)}{V(R,A)}$, i.e., original number of tuples divided by number of different values of `A`

Result size estimation: 1. $\sigma_{A=c}(R)$ with c a constant

Example

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

Alternate Assumption

- Assumption:
  - Values in select expression `A=constant` are uniformly distributed over possible `V(R,A)` values.
- Alternate Assumption:
  - Values in select expression `A=constant` are uniformly distributed over domain with `DOM(R,A)` values.

Result size estimation: 1. $\sigma_{A=c}(R)$ with c a constant

- 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`: the range of values is divided into equal-sized subranges.
  - `equi-depth histograms`: 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.

Result size estimation: 1. $\sigma_{A=c}(R)$ with c a constant

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

Result size estimation: 1. $\sigma_{A=c}$(R) with c a constant

Example

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

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

Result size estimation: 2. $\sigma_{A<c}$(R) with c a constant

- $sel_{A<c}(\mathtt{R}) = \frac{1}{3}$
- Intuition:
  - On average, you would think that the value should be $\frac{T(R)}{2}$. However, queries with inequalities tend to return less than half the tuples, so the rule compensates for this fact.
  - i.e., 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)

Example

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

Result size estimation: 2. $\sigma_{A<c}$(R) with c a constant

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 $sel_{A<10}$(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, $sel_{A<10}(R) = \frac{2}{(57-8+1)} = \frac{2}{50} = 0.04$
- And hence, $\mathtt{T}\big(\sigma_{A<10}(\mathtt{R})\big) = \mathtt{T(R)} \times sel_{A<10}(\mathtt{R}) = 400$

Result size estimation: 3. $\sigma_{A \neq c}$(R) with c a constant

- S=$\sigma_{A \neq c}$(R)
- 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(S) = T(R) $\times$ $\frac{V(R,A)-1}{V(R,A)}$

Result size estimation: 4. $\sigma_{\neg p}$(R)

- $sel_{\neg p}$(R)= 1 - sel_p(R)

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

- $sel_{P_1 \wedge P_2}(\texttt{R}) = sel_{p_1}(\texttt{R}) \times sel_{P_2}(\texttt{R})$
- Assumption: The conditions P_1 and P_2 are (statistically) independent
- Treat $\sigma_{P_1 \wedge P_2}(\texttt{R})$ as $\sigma_{P_1}(\sigma_{P_2}(\texttt{R}))$
- The order does not matter, treating this as $\sigma_{P_2}(\sigma_{P_1}(\texttt{R}))$ gives the same results.

Example

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

Answer

- $sel_{A=10}(\texttt{R}) = \frac{1}{50}$
- $sel_{B<20}(\texttt{R}) = \frac{1}{3}$
- $\texttt{T(S)} = sel_{A=10} \times sel_{B<20} \times \texttt{T(R)} = \frac{1}{50} \times \frac{1}{3} \times 10{,}000 = 66.67$

Result size estimation: 6. $\sigma_{P_1 \vee P_2}(R)$

- $P_1 \vee P_2 = \neg(\neg P_1 \wedge \neg P_2)$
- Treat $\sigma_{P_1 \vee P_2}(\mathtt{R})$ as $\sigma_{\neg(\neg P_1 \wedge \neg P_2)}(\mathtt{R})$
- $sel_{P_1 \vee P_2}(\mathtt{R}) = 1 - \left(1 - sel_{p_1}(\mathtt{R})\right) \times \left(1 - sel_{P_2}(\mathtt{R})\right)$
- Assumption: The conditions P_1 and P_2 are (statistically) independent

Example

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

Answer

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

Result size estimation: $\mathtt{R} \bowtie \mathtt{S}$

- Assume the relation schema R(X,Y) and S(Y,Z), we join on Y $\big(\mathtt{R(X,Y)} \bowtie \mathtt{S(Y,Z)}\big)$.
- Question: Estimate the size of $\big(\mathtt{R(X,Y)} \bowtie \mathtt{S(Y,Z)}\big)$
- Possible outcomes of $\mathtt{R(X,Y)} \bowtie \mathtt{S(Y,Z)}$
  - If the Y attribute values in R(X,Y) and S(Y,Z) are disjoint
    - $\mathtt{T}\big(\mathtt{R(X,Y)} \bowtie \mathtt{S(Y,Z)}\big) = 0$
  - If Y attribute is a key in S and a foreign key of R, so each tuple of R joins with exactly one tuple of S
    - $\mathtt{T}\big(\mathtt{R(X,Y)} \bowtie \mathtt{S(Y,Z)}\big) = \mathtt{T(R)}$
  - If almost every tuple in R and S has the same Y attribute value
    - $\mathtt{T}\big(\mathtt{R(X,Y)} \bowtie \mathtt{S(Y,Z)}\big) = \mathtt{T(R)} \times \mathtt{T(S)}$
- Range of $\mathtt{T(R \bowtie S)}$: $0 \leq \mathtt{T(R \bowtie S)} \leq \mathtt{T(R)} \times \mathtt{T(S)}$

Result size estimation: R ⋈ S: Simplifying Assumptions

- Without any assumptions on the joining attribute values, it is not possible to provide an estimation on the result `T(R ⋈ S)`
- Assumptions that helps use find an estimate of `R(X,Y) ⋈ S(Y,Z)`

1. The `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 ...

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
- `Containment` of value sets assumption will help to estimate the size of `T(R ⋈ S)`

Result size estimation: `R` $\bowtie$ `S`: Simplifying Assumptions

- Assumptions that helps use find an estimate of `R(X,Y)` $\bowtie$ `S(Y,Z)`

2. The `preservation of value sets assumption`
    - The join operation `R(X,Y)` $\bowtie$ `S(Y,Z)` will preserve all the possible values of the non-joining attributes
    - In other words
        - The attribute values taken on by `X` in `R(X,Y)` $\bowtie$ `S(Y,Z)` and `R(X,Y)` are same
        - The attribute values taken on by `Z` in `R(X,Y)` $\bowtie$ `S(Y,Z)` and `S(Y,Z)` are same
        - `preservation of value sets assumption` will help to estimate the size of `T(R` $\bowtie$ `S` $\bowtie$ `U)`

Result size estimation: R $\bowtie$ S when joining on 1 attribute

- We can estimate the size of R(X,Y) $\bowtie$ S(Y,Z) as follows:
- Case 1. V(R,Y)$\geq$V(S,Y)
  - The tuples in relations R and S take on the following attribute values for the Y attribute:
    - Attr values of Y in R: y_1 y_2 ...... $y_{V(R,Y)}$
    - Attr values of Y in S: y_1 y_2 ... $y_{V(S,Y)}$
  - Then 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 $\in$ S will produce $\frac{T(R)}{V(R,Y)}$ number of matches
  - There are T(S) tuples in S, then estimated size of R $\bowtie$ S is $\frac{T(R) \times T(S)}{V(R,Y)}$
- Case 2. V(S,Y)$\geq$V(R,Y)
  - estimated size of R $\bowtie$ S is $\frac{T(R) \times T(S)}{V(S,Y)}$

Result size estimation: $R \bowtie S$ when joining on 1 attribute

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

Result size estimation: R $\bowtie$ S when 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

- Estimate the size of R $\bowtie$ S $\bowtie$ U?

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

Method 1: (ordering 1)

- R(a,b) ⋈ S(b,c) ⋈ U(c,d) $=$ $\big($R(a,b) ⋈ S(b,c)$\big)$ ⋈ U(c,d)
- T$\big($R(a,b) ⋈ S(b,c)$\big)$ = $\frac{T(R)\times T(S)}{\max\big(V(R,b),V(S,b)\big)} = \frac{1000\times 2000}{max\{20,50\}} = 40,000$
- The estimate of the size the join $\big($R(a,b) ⋈ S(b,c)$\big)$ ⋈ U(c,d) is
$$=\frac{T\Big(R(a,b)\bowtie S(b,c)\Big)\times T(U)}{\max\Big(V\Big(R(a,b)\bowtie S(b,c),c\Big),V(U,c)\Big)}$$
- From the preservation of value sets assumption, we have: $V\big(R(a,b)\bowtie S(b,c),c\big)$ = V(S,c), where V(S,c)=100 according to data
- ∴ T(R⋈S⋈U)= $\frac{T\Big(R(a,b)\bowtie S(b,c)\Big)\times T(U)}{\max\Big(V\Big(R(a,b)\bowtie S(b,c),c\Big),V(U,c)\Big)} = \frac{40{,}000\times 5{,}000}{\max(100{,}500)}$=400,000

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

Method 2: (ordering 2)

- $\texttt{R(a,b)} \bowtie \texttt{S(b,c)} \bowtie \texttt{U(c,d)} = \texttt{R(a,b)} \bowtie \big(\texttt{S(b,c)} \bowtie \texttt{U(c,d)}\big)$
- $\texttt{T}\big(\texttt{S(b,c)} \bowtie \texttt{U(c,d)}\big) = \frac{T(S)\times T(U)}{\max\big(V(S,c),V(U,c)\big)} = \frac{2000\times 5000}{max\{100{,}500\}} = 20,000$
- The estimate of the size the join $\texttt{R(a,b)} \bowtie \big(\ \texttt{S(b,c)} \bowtie \texttt{U(c,d)}\big)$ is

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

  $V\big(S(b,c)\bowtie U(c,d),b\big) = \texttt{V(S,b)}$, where $\texttt{V(S,b)}=50$ according to data
- $\therefore \texttt{T(R}\bowtie\texttt{S}\bowtie\texttt{U)} = \frac{T(R)\times T\big(S(b,c)\bowtie U(c,d)\big)}{\max\Big(V(R,b),V\big(S(b,c)\bowtie U(c,d),b\big)\Big)} = \frac{1{,}000\times 20{,}000}{\max(20{,}50)} = 400{,}000$

Result size estimation: $\mathtt{R} \bowtie \mathtt{S}$ when 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:

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

Result size estimation: R $\bowtie$ S when 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

- Estimate the size of R $\bowtie$ S $\bowtie$ U?
- Computed using this ordering:

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

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

Result size estimation: $R \bowtie S$ when joining on 2 attributes

Method 3: (ordering 3)

- $R(a,b) \bowtie S(b,c) \bowtie U(c,d) = \big(R(a,b) \bowtie U(c,d)\big) \bowtie S(b,c)$
- $T\big(R(a,b) \bowtie U(c,d)\big) = T\big(R(a,b) \times U(c,d)\big) = 1000 \times 5000 = 5{,}000{,}000$
- The estimate of the size the join $\big(R(a,b) \bowtie U(c,d)\big) \bowtie S(b,c)$ is

$$= \frac{T\Big(R(a,b) \bowtie U(c,d)\Big) \times T(S)}{\max\Big(V\big(R(a,b) \bowtie U(c,d), b\big), V(S,b)\Big) \times \max\Big(V\big(R(a,b) \bowtie U(c,d), c\big), V(S,c)\Big)}$$

- From the `preservation of value sets` assumption, we have:

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

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

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

Result size estimation: R ∪ S

- Bag-based: `T(R)+T(S)`
- Set-based:
  - Range of the result set of R ∪ S:
  - $\max\big(T(R), T(S)\big) \le T(R \cup S) \le T(R) + T(S)$

    $\max\big(T(R), T(S)\big)$: $R \subseteq S$ or $S \subseteq R$

    $T(R) + T(S)$: $R \cap S = \emptyset$
  - Recommended estimate for R ∪ S

    $T(R \cup S) = \max\big(T(R), T(S)\big) + \frac{1}{2} \times \min\big(T(R), T(S)\big)$

    i.e.: maximum + $\frac{1}{2}$ × (smaller size)

Result size estimation: $\texttt{R} \cap \texttt{S}$

- Range of the result set of $\texttt{R} \cap \texttt{S}$
  - $0 \leq \texttt{T(R} \cap \texttt{S)} \leq \texttt{min}\big(\texttt{T(R),T(S)}\big)$
    - 0: $\texttt{R} \cap \texttt{S} = \emptyset$
    - $\texttt{min}\big(\texttt{T(R),T(S)}\big)$: $\texttt{R} \subseteq \texttt{S}$ or $\texttt{S} \subseteq \texttt{R}$
- Recommended estimate for $\texttt{R} \cap \texttt{S}$
  - $\texttt{T(R} \cap \texttt{S)} = \frac{1}{2} \times \texttt{min}\big(\texttt{T(R),T(S)}\big)$
  - i.e.: `the average of the min and max`

Result size estimation: R$-$S

- Range of the result set of R$-$S
  - $\max\big(0, T(R)-T(S)\big) \leq T(R-S) \leq T(R)$
    - $\max\big(0, T(R)-T(S)\big)$: $R \subseteq S$ or $S \subseteq R$
    - $T(R)$: $R \cap S = \emptyset$
- Recommended estimate for R$-$S
  - $T(R-S) = T(R) - \frac{1}{2} \times T(S)$
  - (Probabilistically speaking: 50–50 chance that a tuple in S is also in R)
  - Note: if $T(R) - \frac{1}{2} \times T(S) \leq 0$, then $T(R-S) = 0$ (estimate)

Result size estimation: δ(R,A)

- Range of the result set of δ(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 δ(R,A)
  - If the database maintains statistics on the attribute values:
    $T\big(\delta(R,A)\big) = V(R,A)$
  - If no statistics available, then we use this estimate:
    $T\big(\delta(R,A)\big) = \frac{1}{2} \times T(R)$
- Recommended estimate for δ(R,A,B)
  - If the database maintains statistics on the attribute values:
    $T\big(\delta(R,A,B)\big) = V(R,A) \times V(R,B)$
  - If no statistics available, then we use this estimate:
    $T\big(\delta(R,A,B)\big) = \frac{1}{4} \times T(R)$

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

- Range of the result set of $\gamma_L(\mathtt{R})$
  - $1 \leq \mathtt{T}\big(\gamma_L(\mathtt{R})\big) \leq \mathtt{T(R)}$
    - `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:
    $\mathtt{T}\big(\gamma_L(\mathtt{R})\big) = \frac{1}{2}^{n(L)} \times \mathtt{T(R)}$, where $n(L) =$ number of attributes in the attribute list L

Summary

- As should be clear by now, result size estimation is not an exact art
- Don't forget: Statistics must be kept up to date. (cost?)

Outline

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