IN3020/4020 – Database Systems
Spring 2020, Week 6.1

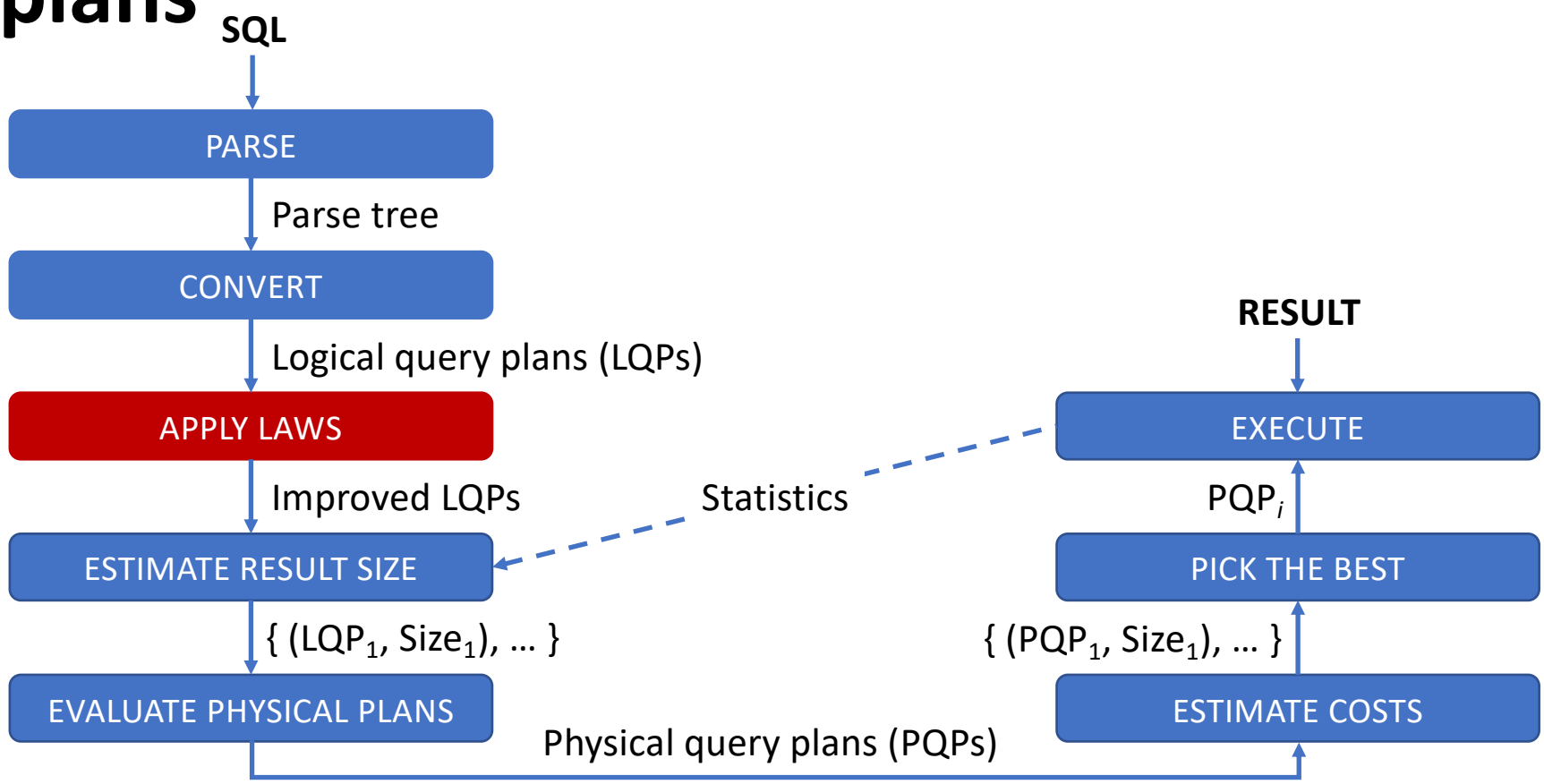
Query Compilation – Part 2

Dr. M. Naci Akkøk, Chief Architect, Oracle Nordics

Based upon slides by E. Thorstensen from Spring 2019

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Algebraic laws for improving logical query plans



Commutativity and associativity

- An operator ω is commutative if the order of the arguments does not matter:

  $x \ \omega \ y = y \ \omega \ x$

- An operator is associative if the grouping of arguments has no significance:

  $x \ \omega \ (y \ \omega \ z) = (x \ \omega \ y) \ \omega \ z$

Algebraic laws for product and join

- Natural join and product are both commutative and associative:
  - $R \bowtie S = S \bowtie R$ and $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
  - $R \times S = S \times R$ and $R \times (S \times T) = (R \times S) \times T$
- What about theta-join?
  - Commutative: $R \bowtie_c S = S \bowtie_c R$
  - But not always associative.
    - Example: Given the relationships R (a, b), S (b, c), T (c, d).
    - Then: $(R \bowtie_{R.b < S.b} S) \bowtie_{a < d} T \neq R \bowtie_{R.b < S.b} (S \bowtie_{a < d} T)$

Order of produkt and join

Does the order and grouping of a product or join have an effect on efficiency?

- If only one of the relations fits into the memory, this should be the first argument -- a one-pass operation that reduces the number of disk Ios
- If the product or join of two of the relations result in a temporary relation that fits into the memory, these joins should be taken first to save both memory space and disk IO
- Temporary relations (intermediate results) should be as small as possible to save memory space
- If we can estimate (using statistics) the number of tuples to be joined, we can save many operations by joining the relations that give the fewest tuples first (does not apply to product)

Algebraic laws for union and intersection

- Union and intersection are commutative and associative:
  - $R \cup S = S \cup R$; $R \cup (S \cup T) = (R \cup S) \cup T$
  - $R \cap S = S \cap R$; $R \cap (S \cap T) = (R \cap S) \cap T$

- Note: Some laws may be different for sets and bags, e.g. distribution of intersection across union:
  - $R \cap_S (S \cup_S T) = (R \cap_S S) \cup_S (R \cap_S T)$
  - $R \cap_B (S \cup_B T) \neq (R \cap_B S) \cup_B (R \cap_B T)$

Algebraic laws for selection

- Selection is a very important operator for optimization
  - Reduces the number of tuples (the size of the relationship)
  - General rule: push selections as far down the tree as possible
- Conditions with AND and OR can be split:
  - $\sigma_{a \text{ AND } b}(R) = \sigma_a(\sigma_b(R))$
  - $\sigma_{a \text{ OR } b}(R) = (\sigma_a(R)) \cup_S (\sigma_b(R))$, where R is a set and $\cup_S$ is a set union
    - Splitting OR works only when R is a set. If R is a bag and both conditions are met, bag union will include a tuple twice - once in each selection.
- The order of subsequent selections has no bearing on the resulting set:
  - $\sigma_a(\sigma_b(R)) = \sigma_b(\sigma_a(R))$

Pushing selection

REMEMBER
Join: $\sigma_a(R \bowtie S) = \sigma_a(R) \bowtie \sigma_a(S) = R \bowtie \sigma_a(S) = \sigma_a(R) \bowtie S$

- When selection is pushed down the tree, then ...
- It must be pushed to both arguments for
  - **Union:** $\sigma_a(R \cup S) = \sigma_a(R) \cup \sigma_a(S)$
  - **Cartesian product:** $\sigma_a(R \times S) = \sigma_a(R) \times \sigma_a(S)$
- It must be pushed to the first argument (optionally the second argument too) for
  - **Difference:** $\sigma_a(R - S) = \sigma_a(R) - S = \sigma_a(R) - \sigma_a(S)$
- It can be pushed to one or both arguments for
  - **Intersection:** $\sigma_a(R \cap S) = \sigma_a(R) \cap \sigma_a(S) = R \cap \sigma_a(S) = \sigma_a(R) \cap S$
  - **Join:** $\sigma_a(R \bowtie S) = \sigma_a(R) \bowtie \sigma_a(S) = R \bowtie \sigma_a(S) = \sigma_a(R) \bowtie S$
  - **Theta join:** $\sigma_a(R \bowtie_b S) = \sigma_a(R) \bowtie_b \sigma_a(S) = R \bowtie_b \sigma_a(S) = \sigma_a(R) \bowtie_b S$

Pushing selection – Example

Assume that each attribute is 1 byte

$\sigma_{A=2}$ (R ⋈ S)

- **join:** compare 4 * 4 items = 16 operations
  store (cache) the relation: R ⋈ S yields 52 i.e., ((23 + 3) x 2) bytes
- **selection:** check each tuple: 2 operations

$\sigma_{A=2}$ (R) ⋈ S

- **selection:** check each tuple in R: 4 operations
  store (cache) the relation: $\sigma_{A=2}$(R) gives 24 bytes
- **join:** compare 1 x 4 items = 4 operations

Relation R

A	B	C	...	X
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4

Relation S

X	Y	Z
2	f	c
3	t	b
7	g	c
9	e	c

Relation **R ⋈ S**

A	B	C	...	X	Y	Z
2	c	6	...	2	f	c
3	r	8	...	7	g	c

Relation **$\sigma_{A=2}$ (R)**

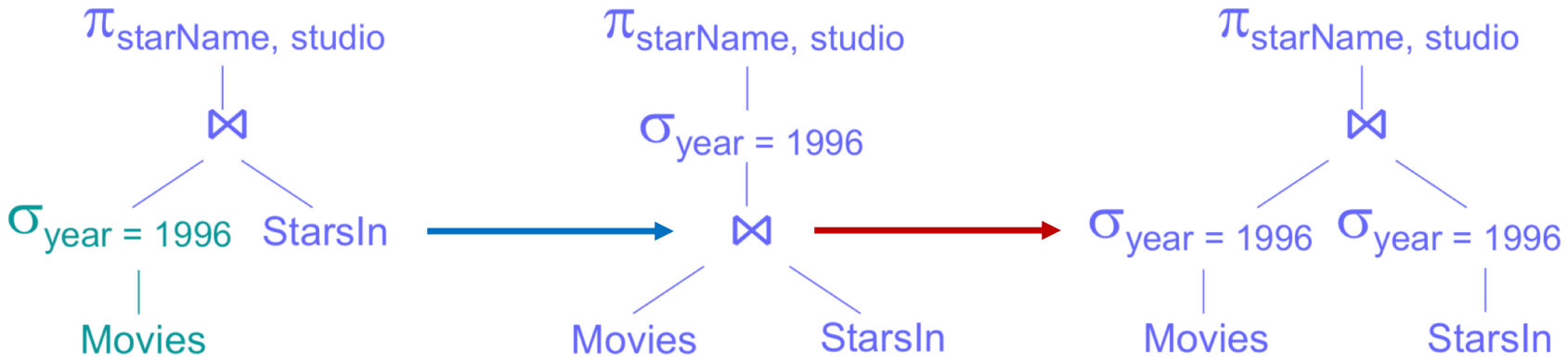
A	B	C	...	X
2	c	6	...	2

Pushing selection upwards in the tree

Sometimes it is useful to push selection the other way, ie upwards in the tree, using the law $\sigma_a(R \bowtie S) = R \bowtie \sigma_a(S)$ «backwards».

EXAMPLE: StarsIn(title, year, starName); Movies(title, year, studio ...)

```
CREATE VIEW Movies96 AS
      SELECT * FROM Movies WHERE year = 1996;
SELECT starName, studio FROM Movies96 NATURAL JOIN StarsIn;
```



REMEMBER Join: $\sigma_a(R \bowtie S) = \sigma_a(R) \bowtie \sigma_a(S) = R \bowtie \sigma_a(S) = \sigma_a(R) \bowtie S$

Algebraic laws for projection #1

Projection can be pushed through join and cross product (i.e., new projections can be introduced) as long as we do not remove attributes used further up the tree:

- $\pi_L(R \bowtie S) = \pi_L(\pi_M(R) \bowtie \pi_N(S))$ if
  - M contains the join attributes and that of L which is in R
  - N contains the join attributes and that of L which is in S
- $\pi_L(R \bowtie_C S) = \pi_L(\pi_M(R) \bowtie_C \pi_N(S))$ if
  - M contains the attributes in C and L that are in R
  - N contains the attributes in C and L that are in S
- $\pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S))$ if
  - M contains the attributes of L that are in R
  - N contains the attributes of L that are in S

Algebraic laws for projection #2

Projection can be pushed through bag union:

$$\pi_L(R \cup_B S) = \pi_L(R) \cup_B \pi_L(S))$$

Note: The same rule does not apply to set union, set intersection, bag intersection, set difference or bag difference because projection can change the multiplicity of the tuples:

- R being a set does not necessarily mean that $\pi_L(R)$ is a set
- If R is a bag and a tuple **t** occurs *k* times in R, then the projection of **t** on L may occur more than *k* times in $\pi_L(R)$.

Algebraic laws for projection #3

Projection can be pushed through selection (new projections are introduced) as long as we do not remove attributes used further up the tree:

- $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$
  if M contains the attributes in C and L
- NOTE: If R has an index on some of the attributes in selection condition C, then this index will not be possible to use during selection if we first do a projection on M.

Algebraic laws for join, product. Selection and projection

Two important laws that follow from the definition of join:

- $\sigma_C(R \times S) = R \bowtie_C S$
- $\pi_L(\sigma_C(R \times S)) = R \bowtie S$ if
  - C compares each pair of tuples from R and S with the same name
  - L = all attributes of R and S (without duplicates)

Examples

- $\pi_L(\sigma_{R.a = S.a}(R \times S))$ vs. $R \bowtie S$
  - R(a,b,c,d,e,..., k), T(R) = 10.000, S(a,l,m,n,o,...,z), T(S) = 100
  - Each tuple is 1 byte, a is a candidate key in both R and S
  - Assumes that all doubles in S find a match in R, i.e., 100 doubles in the result
- $\pi_L(\sigma_C(R \times S))$:
  - Cross product:
    combine 10,000 * 100 items = 1,000,000 operations
    temp storage, relation $R \times S$ = 1,000,000 * (11 + 16) = 27,000,000 bytes
  - Selection:
    Check each tuple: 1.000.000 operations
    temp storage, relation $\sigma_{R.a = S.a}(R \times S)$ = 100 * 27 = 2700 bytes
  - Projection:
  - Check each tuple: 100 operations
- $R \bowtie S$:
  - join: check 10.000 * 100 elements = 1.000.000 operations

Algebraic laws for duplicate elimination

Duplication elimination can reduce the size of temporary relations by pushing through

- Cartesian product: $\delta(R \times S) = \delta(R) \times \delta(S)$
- Join: $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$
- Theta join: $\delta(R \bowtie_C S) = \delta(R) \bowtie_C \delta(S)$
- Selection: $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$
- Bag intersection: $\delta(R \cap_B S) = \delta(R) \cap_B \delta(S) = \delta(R) \cap_B S = R \cap_B \delta(S)$

- Note: duplicate elimination cannot be pushed through
  - **set** operations
  - **bag** union and difference
  - **projections**

Algebraic laws for grouping operation

This one is easy: **No general rules!**

Improving (optimizing) logical query plans

The most common LQP optimizations are:

- Push selections as far down as possible.
- If the selection condition consists of several parts, split into multiple selections and push each one as far down the tree as possible.
- Push projections as far down as possible.
- Combine selection and Cartesian product to an appropriate type of join
- Duplicate eliminations can sometimes be removed.
- But don't ruin indexing: Pushing projection past a selection can ruin the use of indexes in the selection!