

IN3020/4020 – Database Systems

Spring 2020, Week 5.2

Query Compilation – Part 1

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

Query Compilation (in two parts)

Part 1:

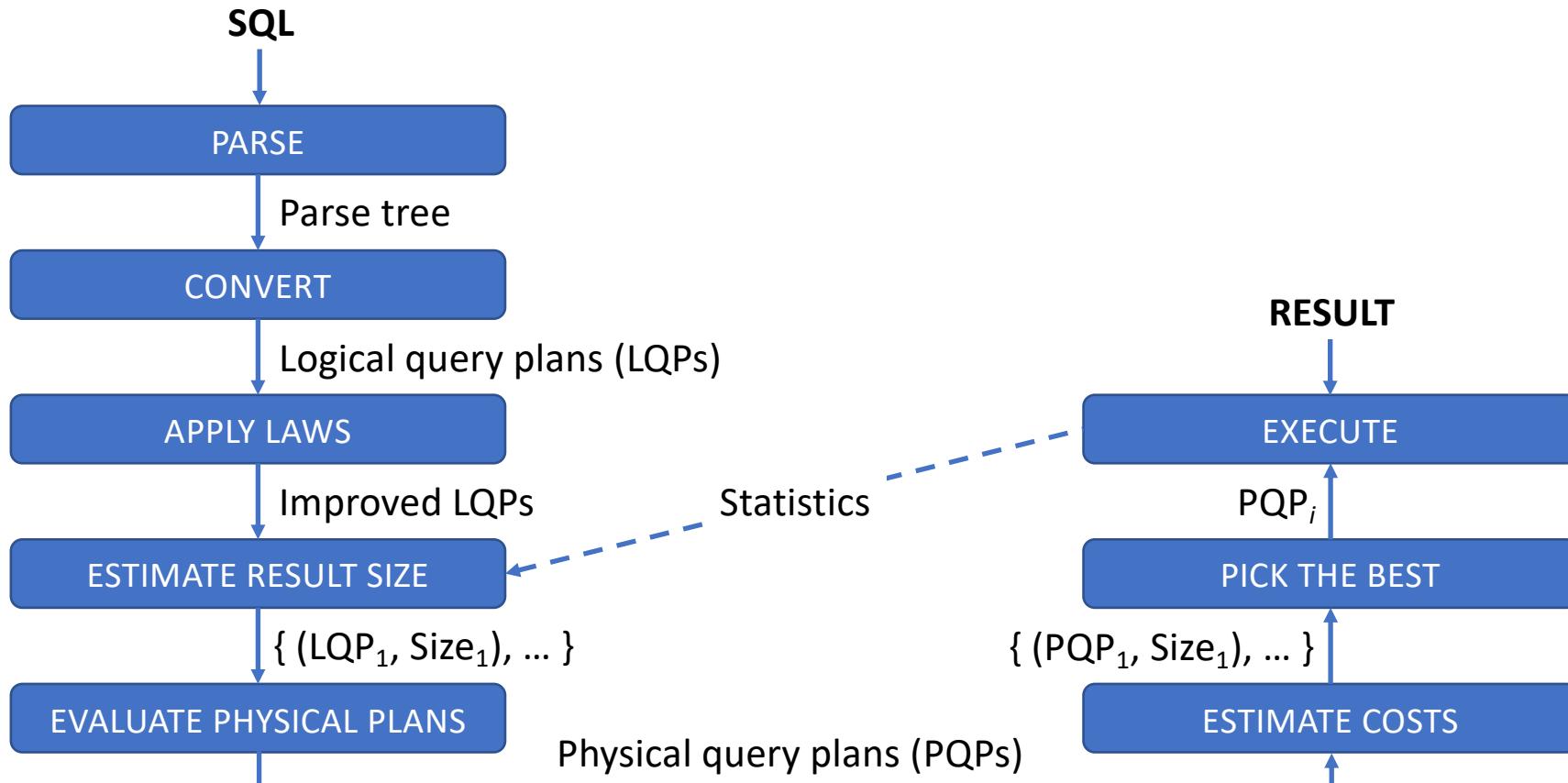
- Parsing
- Logical query plans (expressed in relational algebra)
- Optimization (using algebraic laws)

Part 2:

- Estimate the size/cost of the intermediary results
- Evaluate physical query plans

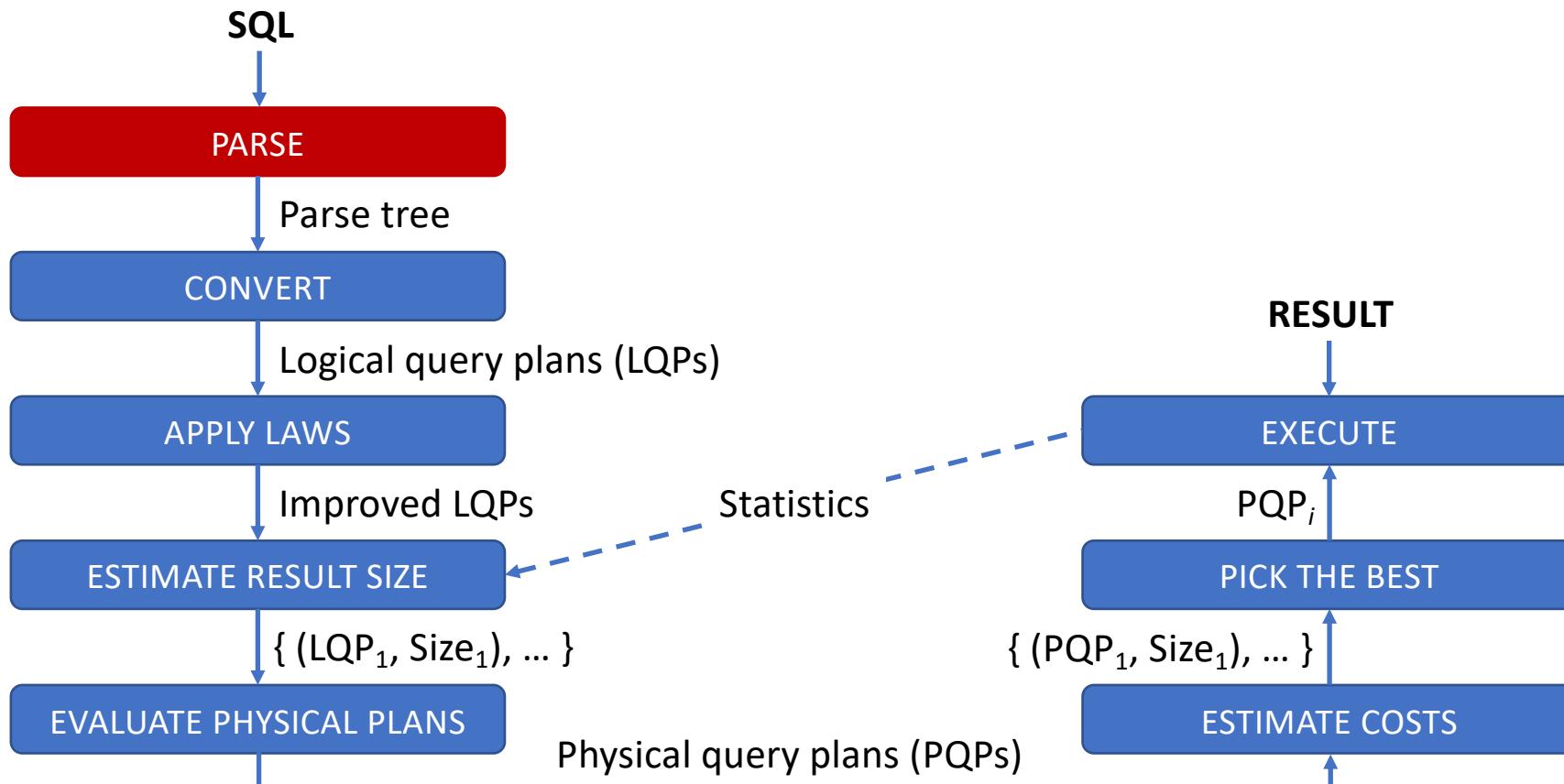


OVERVIEW: The (typical) journey of a query

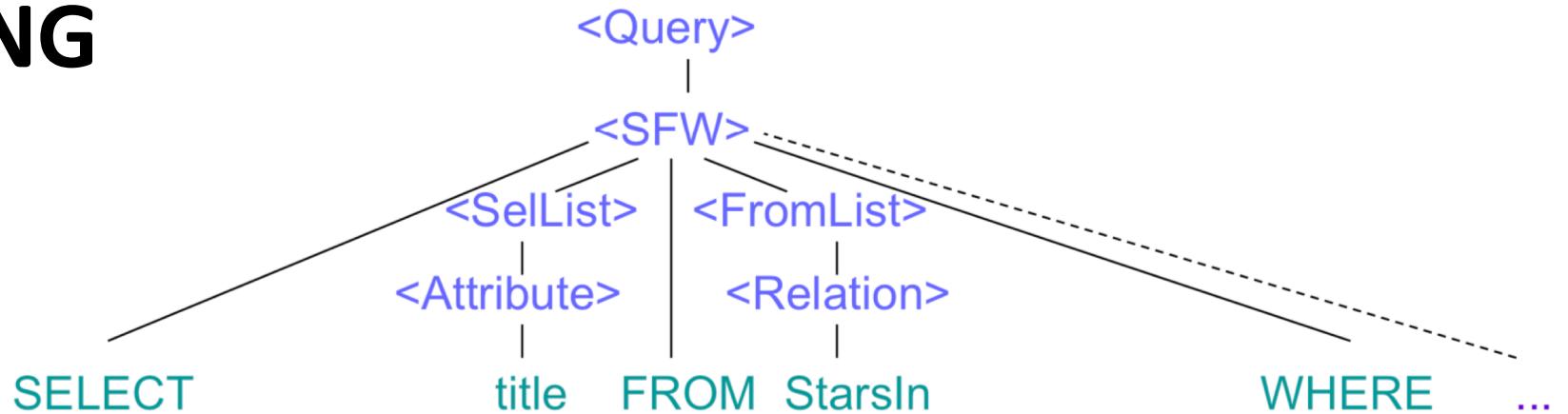


PARSING

The goal is to convert an **SQL-query** to a **parse tree**



PARSING



- Each node in a parse tree is either:
- an *atom (primitive)* – i.e., a lexical element like a keyword, name, constant, parentheses or operators ... (leaf node)
- a *syntactic category* – part of the query ... (inner node)



Simple Grammar #1

- Query:
 - $\langle \text{Query} \rangle ::= \langle \text{SFW} \rangle$
 - $\langle \text{Query} \rangle ::= (\langle \text{Query} \rangle)$
 - The complete grammar would also include operators like UNION, JOIN, ...
 - Rule 2 is typically used in sub-queries



Simple Grammar #2

- Select-From-Where:
 - $\langle SFW \rangle ::= \text{SELECT } \langle SelList \rangle \text{ FROM } \langle FromList \rangle \text{ WHERE } \langle Condition \rangle$
 - The complete grammar must include GROUP BY, HAVING, ORDER BY, ...
- Select-list:
 - $\langle SelList \rangle ::= \langle Attribute \rangle$
 - $\langle SelList \rangle ::= \langle Attribute \rangle, \langle SelList \rangle$
 - The complete grammar must include expressions and aggregate functions
- From-list:
 - $\langle FromList \rangle ::= \langle Relation \rangle$
 - $\langle FromList \rangle ::= \langle Relation \rangle, \langle FromList \rangle$
 - The complete grammar must include aliasing and expressions like R JOIN S



Simple Grammar #3

- Condition:
 - $\langle \text{Condition} \rangle ::= \langle \text{Condition} \rangle \text{ AND } \langle \text{Condition} \rangle$
 - $\langle \text{Condition} \rangle ::= \langle \text{Tuple} \rangle \text{ IN } \langle \text{Query} \rangle$
 - $\langle \text{Condition} \rangle ::= \langle \text{Attribute} \rangle = \langle \text{Attribute} \rangle$
 - $\langle \text{Condition} \rangle ::= \langle \text{Attribute} \rangle \text{ LIKE } \langle \text{Pattern} \rangle$
 - The complete grammar must include operators like OR, NOT, etc., and all comparison operators
- Tuple:
 - $\langle \text{Tuple} \rangle ::= \langle \text{Attribute} \rangle$
 - The complete grammar must include tuples with multiple attributes, ...
- Basic syntactic categories like $\langle \text{Relation} \rangle$, $\langle \text{Attribute} \rangle$, $\langle \text{Pattern} \rangle$ etc. do not have own rukles, but are replaced with a name or a text string



Simple Grammar: Example

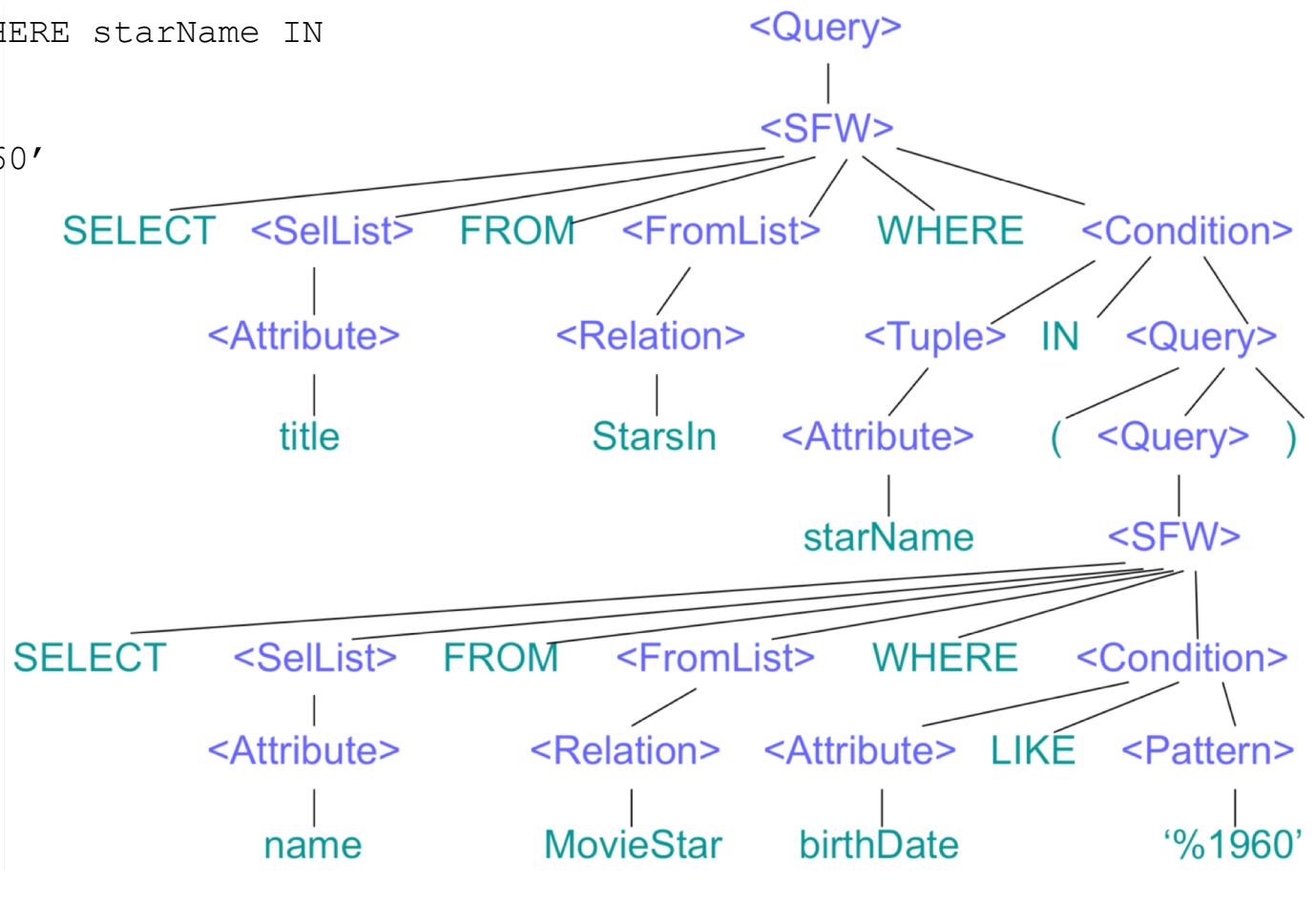
Find films with actors born in 1960:

```
SELECT title FROM StarsIn WHERE starName IN  
( SELECT name  
    FROM MovieStar  
   WHERE birthDate LIKE '%1960'  
);
```



Simple Grammar: Example

```
SELECT title FROM StarsIn WHERE starName IN  
( SELECT name  
  FROM MovieStar  
 WHERE birthDate LIKE '%1960'  
) ;
```

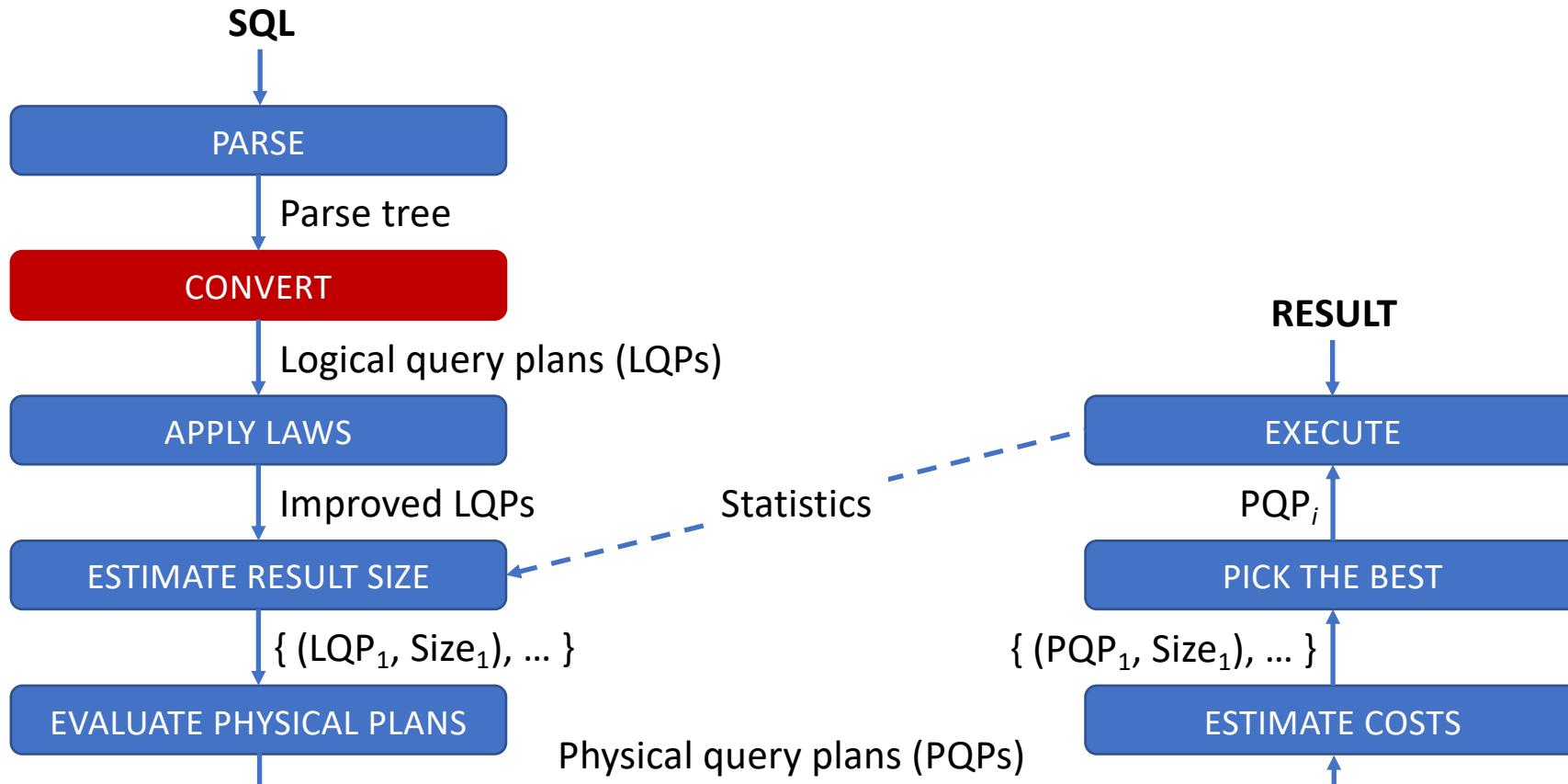


Pre-processor

- Check that the queries are semantically correct:
 - *relations* – each relation in FROM must be a relation or a view in the schema the query is executed
Hvert view må erstattes av et parseringstre.
 - *attributes* – each attribute must exist in one of the relations within the scope of the query
 - *types* – all usage of attributes must be in accordance with the given types



Generating the logical query plan



Logical and Physical query-plans

- These parse-tree is converted into «execution plans» in several stages
- Logical plan: Relational algebra expressions *Relational alg*
- Physical plan: Actual algorithms *alg*
- These are supposed to be two distinct stages, but, the two stages often overlap in reality



Logical query-plans – Example

SELECT B, C, Y
FROM R, S
WHERE W = X AND A = 3 AND Z = 'a'
(attribute list) | (attribute list) <--> select)

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c



Logical query-plans – Strategy 1

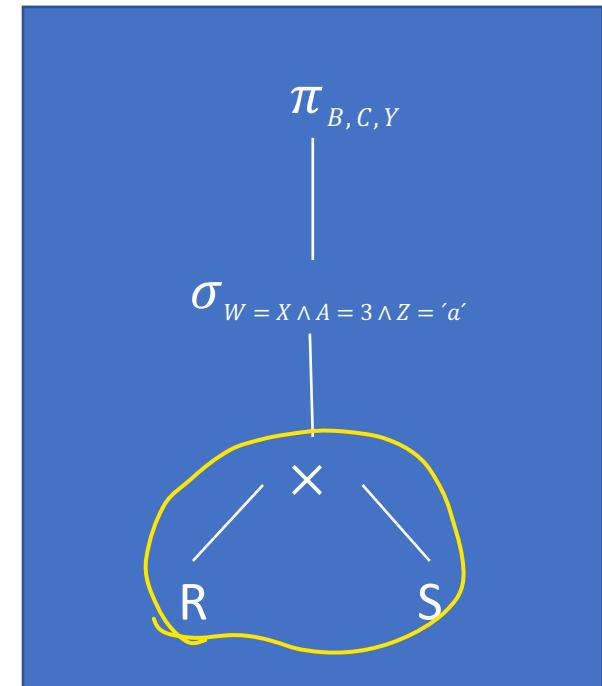
1. Take the cross product of R and S,
2. Select tuples,
3. Project the attributes.

$$\pi_{B,C,Y} (\sigma_{W = X \wedge A = 3 \wedge Z = 'a'} (R \times S))$$

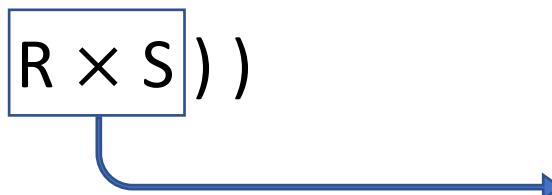
NOTE:

$$\# \text{ attributes} = \#R\text{-attributes} + \#S\text{-attributes} = 23 + 3 = 26$$

$$\# \text{ tuples} = \#R\text{-tuples} \times \#S\text{-tuples} = 9 \times 9 = 81$$



Logical query-plans – Strategy 1

$$\pi_{B,C,Y}(\sigma_{W=X \wedge A=3 \wedge Z='a'}(R \times S))$$


A	B	C	...	W	X	Y	Z
1	z	1	...	4	1	a	a
...	2	f	c
...
2	c	6	...	2	1	a	a
...	2	f	c
...
3	r	8	...	7	1	a	a
...	2	f	c
...
3	r	8	...	7	7	k	a
4	n	9	...	4	1	a	a
...	2	f	c
...	v	
2	i	0	...	3	1	a	a
...	2	c	c
...
3	t	5	...	9	1	a	a
...	2	f	c
...
7	e	3	...	3	1	a	a
...	2	f	c
...	v

```
SELECT B, C, Y
FROM R, S
WHERE W = X AND A = 3 AND Z = 'a'
```

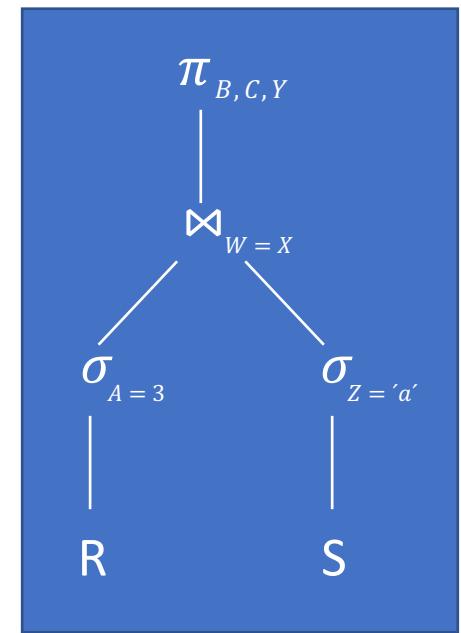
RESULT

B	C	Y
r	8	k



Logical query-plans – Strategy 2

1. Select tuples,
2. Do an equi-join,
3. Project the attributes.

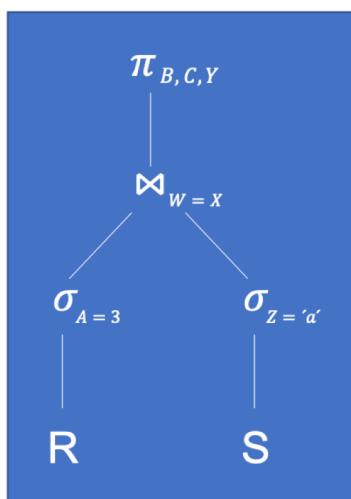
$$\pi_{B,C,Y} ((\sigma_{A=3} (R)) \bowtie_{W=X} (\sigma_{Z='a'} (S))$$


Logical query-plans – Strategy 2

1. Select tuples,
2. Do an equi-join,
3. Project the attributes.

$$\pi_{B,C,Y}((\sigma_{A=3}(R)) \bowtie_{W=X} (\sigma_{Z='a'}(S)))$$

3 1 2 1



A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π

RESULT

B	C	Y
r	8	k

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

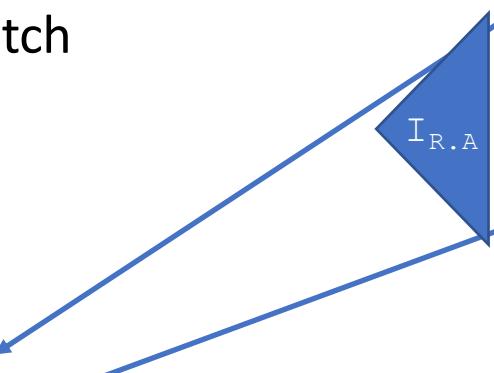


Logical query-plans – Strategy 3

USE INDICES!

1. Use the index on R.A to select tuples where R.A = 3
2. Use the index on S.X to select tuples where S.X = R.W
3. Select the S-tuples where S.Z = 'a'
4. Join the tuples from R and S that match
5. Project the attributes B, C and Y.

r ₁				
A	B	C	...	W
3	r	8	...	7
3	t	5	...	9



```
SELECT B, C, Y  
FROM R, S  
WHERE W = X AND A = 3 AND Z = 'a'
```

R					
A	B	C	...	W	
1	z	1	...	4	
2	c	6	...	2	
3	r	8	...	7	
4	n	9	...	4	
2	j	0	...	3	
3	t	5	...	9	
7	e	3	...	3	
8	f	5	...	8	
1	h	7	...	5	



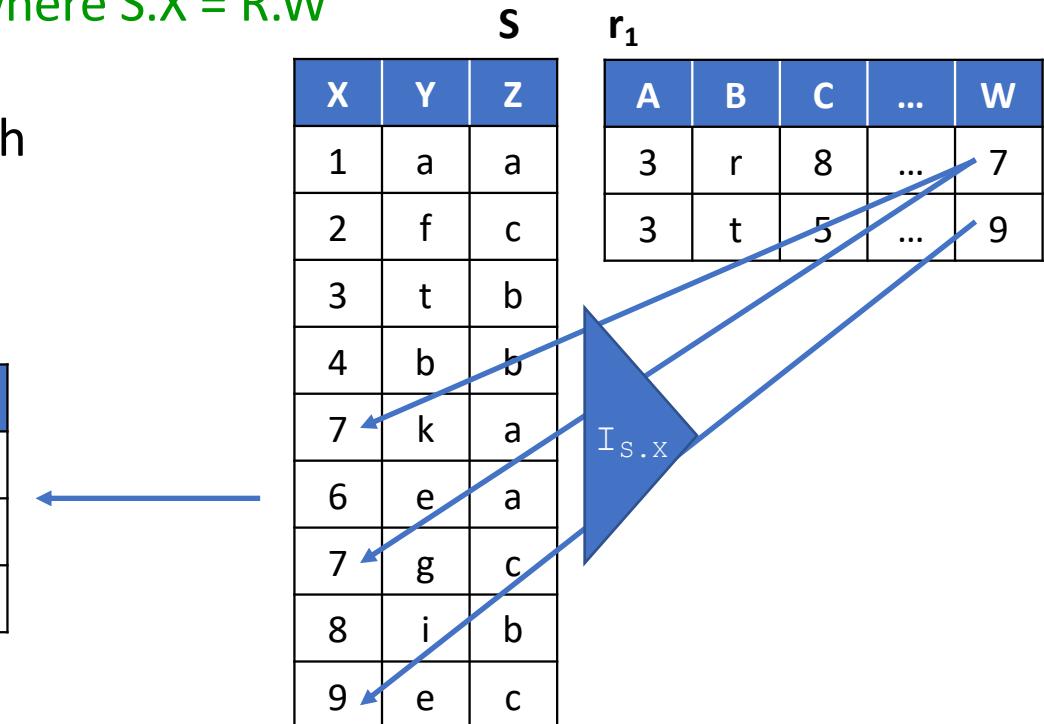
Logical query-plans – Strategy 3

```
SELECT B, C, Y  
FROM R, S  
WHERE W = X AND A = 3 AND Z = 'a'
```

USE INDICES!

1. Use the index on R.A to select tuples where R.A = 3
2. Use the index on S.X to select tuples where S.X = R.W
3. Select the S-tuples where S.Z = 'a'
4. Join the tuples from R and S that match
5. Project the attributes B, C and Y.

X	Y	Z
7	k	a
7	g	c
9	e	c

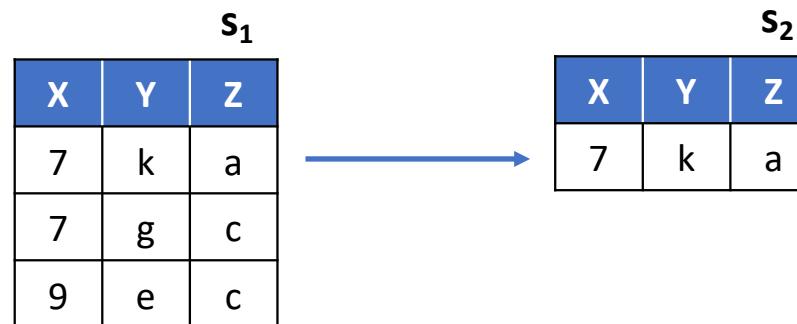


Logical query-plans – Strategy 3

```
SELECT B, C, Y  
FROM R, S  
WHERE W = X AND A = 3 AND Z = 'a'
```

USE INDICES!

1. Use the index on R.A to select tuples where R.A = 3
2. Use the index on S.X to select tuples where S.X = R.W
3. Select the S-tuples where S.Z = 'a'
4. Join the tuples from R and S that match
5. Project the attributes B, C and Y.



Logical query-plans – Strategy 3

```
SELECT B, C, Y  
FROM R, S  
WHERE W = X AND A = 3 AND Z = 'a'
```

USE INDICES!

1. Use the index on R.A to select tuples where R.A = 3
2. Use the index on S.X to select tuples where S.X = R.W
3. Select the S-tuples where S.Z = 'a'
4. Join the tuples from R and S that match
5. Project the attributes B, C and Y

r ₁				
A	B	C	...	W
3	r	8	...	7
3	t	5	...	9

s ₂		
X	Y	Z
7	k	a



A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a
3	t	5	...	9			

Result

B	C	Y
r	8	k



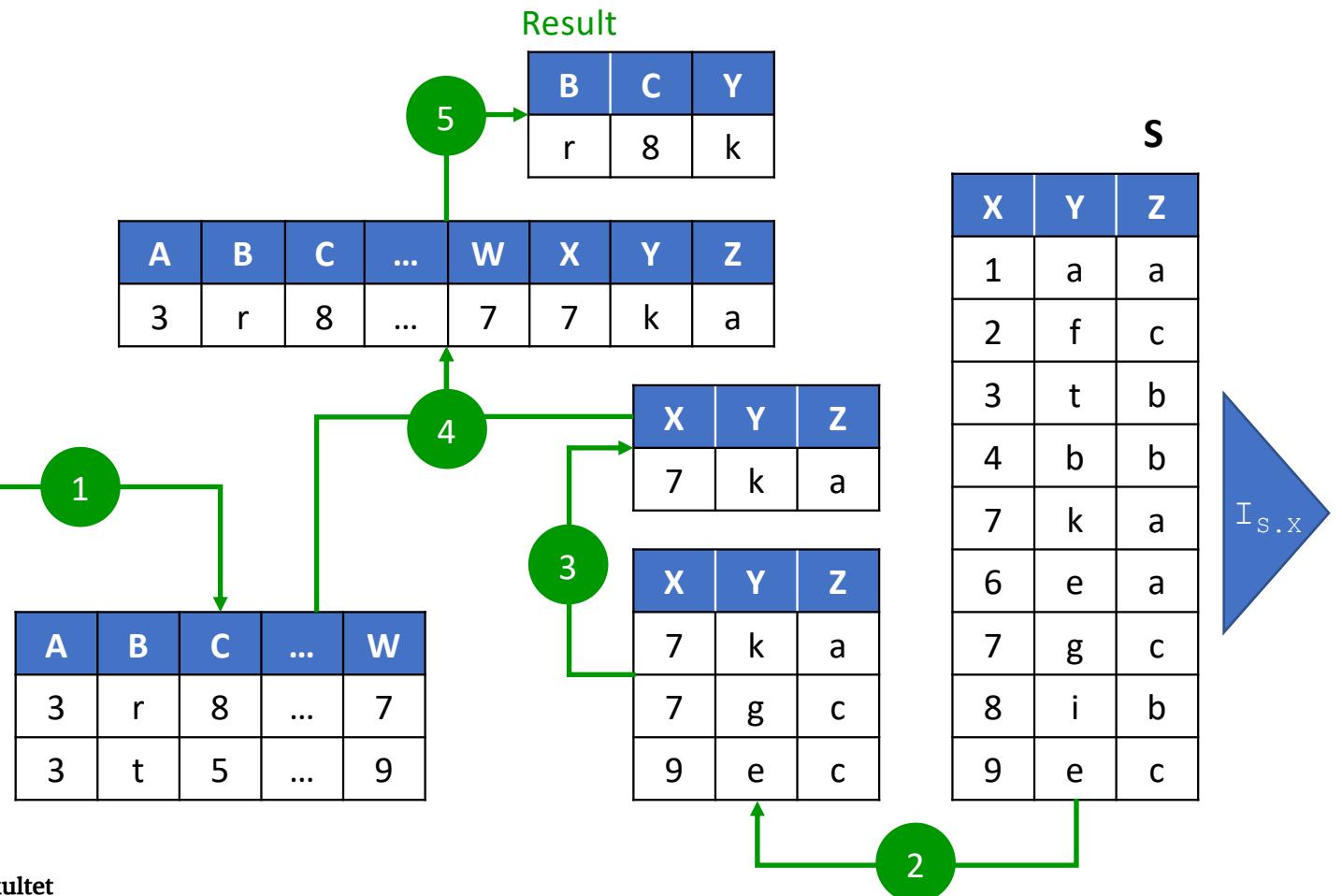
Strategy 3 Summary

1. Use the index on R.A to select tuples where R.A = 3
2. Use the index on S.X to select tuples where S.X = R.W
3. Select the S-tuples where S.Z = 'a'
4. Join the tuples from R and S that match
5. Project the attributes B, C and Y.

R

$I_{R.A}$

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5



Converting Select-From-Where (SFW)

SELECT <SelList> FROM <FromList> WHERE <Condition>

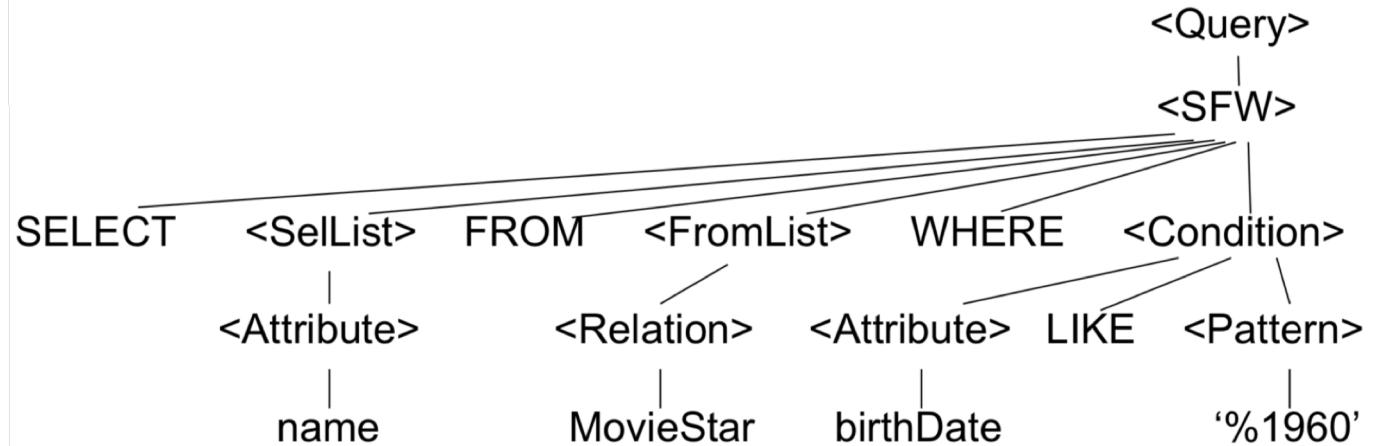
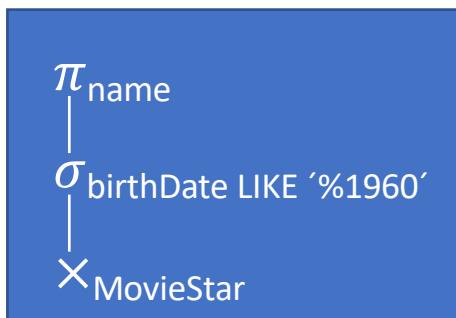
- Replace the relations in <FromList> with the **product** (\times) of all the relations
- This product is the argument for the **selection** (σ_C) where C is the <Condition>
- This selection is the argument for the **projection** (π_L) where L is the list of attributes in <SelList>



SFW conversion example

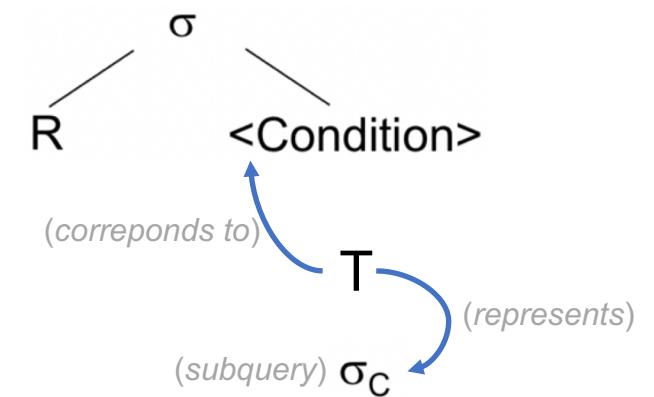
Product (\times) of all relations in <FromList>
Selection (σ_C), with C as <Condition>
Projection (π_L), with L as attributes in <SelList>

```
SELECT name FROM MovieStar WHERE birthDate LIKE '%1960'
```



Converting sub-queries

- For subqueries, we use an **auxiliary operator**, the **two-argument selection $\sigma(R, T)$** , where T represents the subquery (i.e., that which corresponds to $\langle \text{Condition} \rangle$).
- Further processing depends on the type of the subquery.

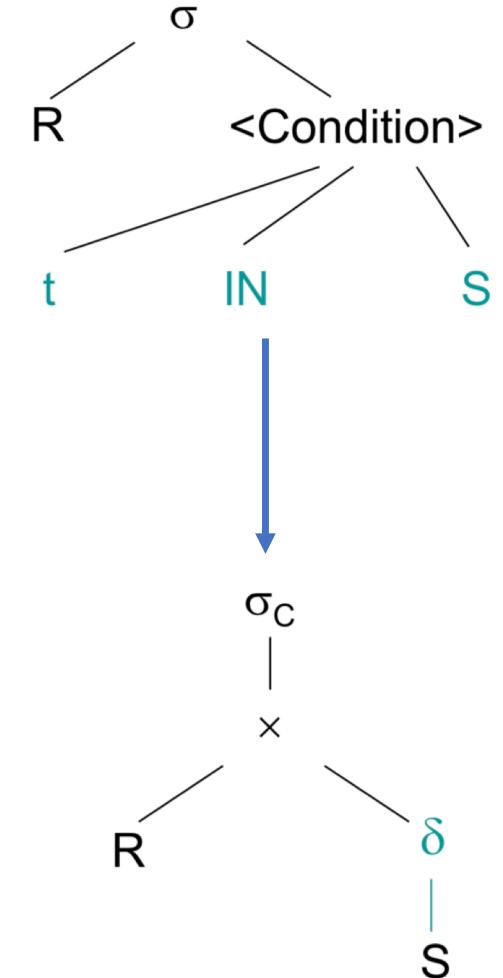


Converting sub-queries

Example

Let us look at $t \text{ IN } S$ as one example:

- Replace $\langle\text{Condition}\rangle$ with the tree for S .
- If S can contain duplicates, we need a δ operator above S .
- Replace two-argument selection with one-argument selection σ_C , where C compares each component in t with the corresponding attribute in S .
- Let σ_C get the product of R and S as argument

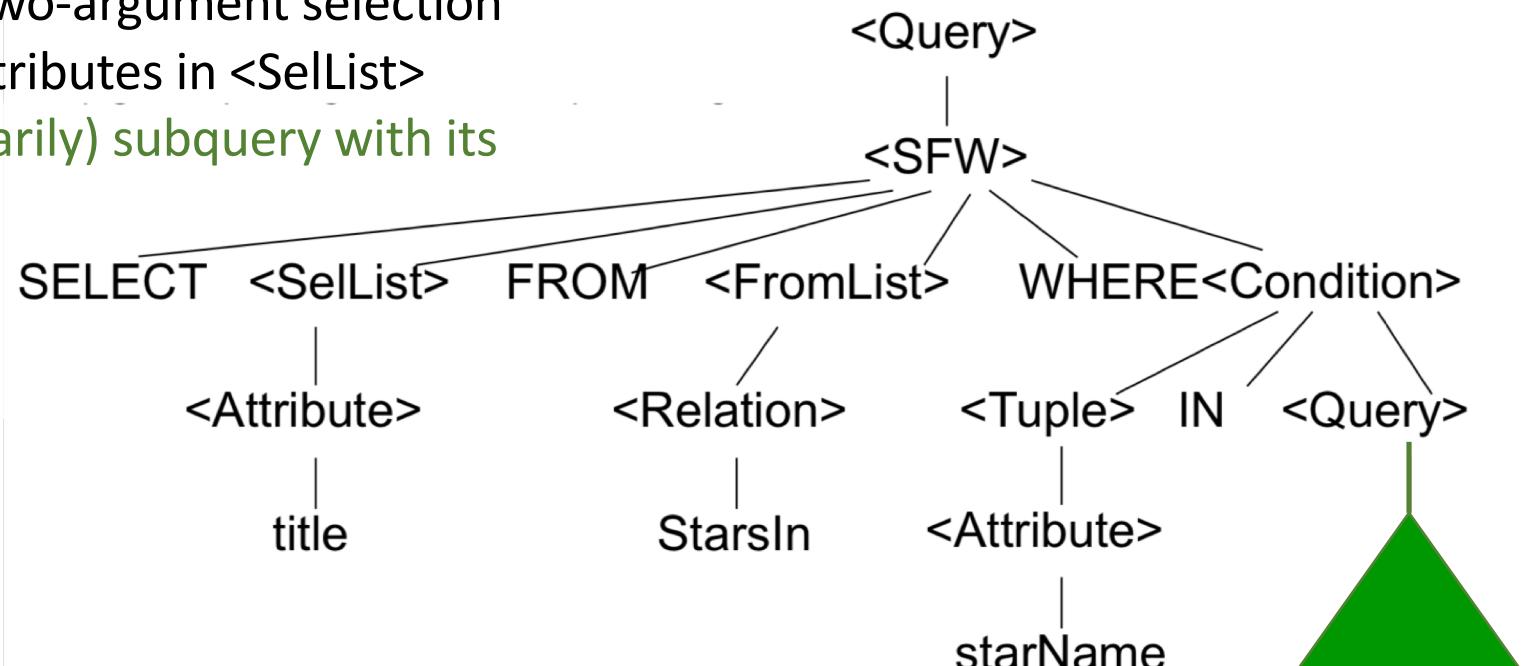


Converting sub-queries

Example

- Product of the relations in <FromList>
- Select based upon <Condition>, represented by two-argument selection
- Project on the attributes in <SelList>
- Replace (temporarily) subquery with its parse tree

```
SELECT title FROM StarsIn  
WHERE starName IN  
(SELECT name FROM MovieStar  
WHERE birtDate LIKE '%1960');
```

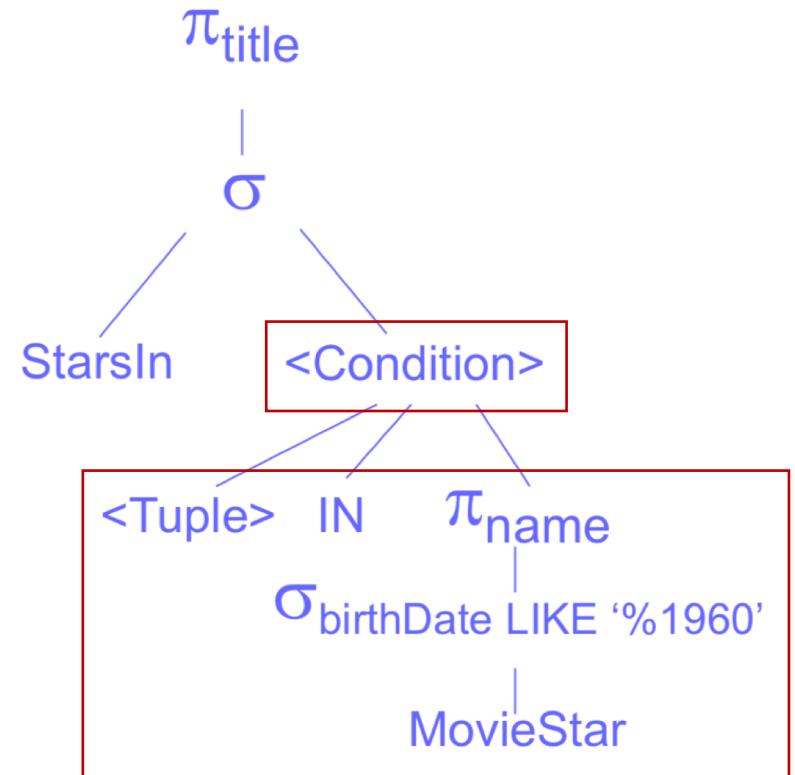


Converting sub-queries

Example

- Product of the relations in `<FromList>`
- Select based upon `<Condition>`,
represented by two-argument selection
- Project on the attributes in `<SelList>`
- Replace (temporarily) **subquery with its
parse tree**

```
SELECT title FROM StarsIn
WHERE starName IN
  (SELECT name FROM MovieStar
   WHERE birtDate LIKE '%1960');
```

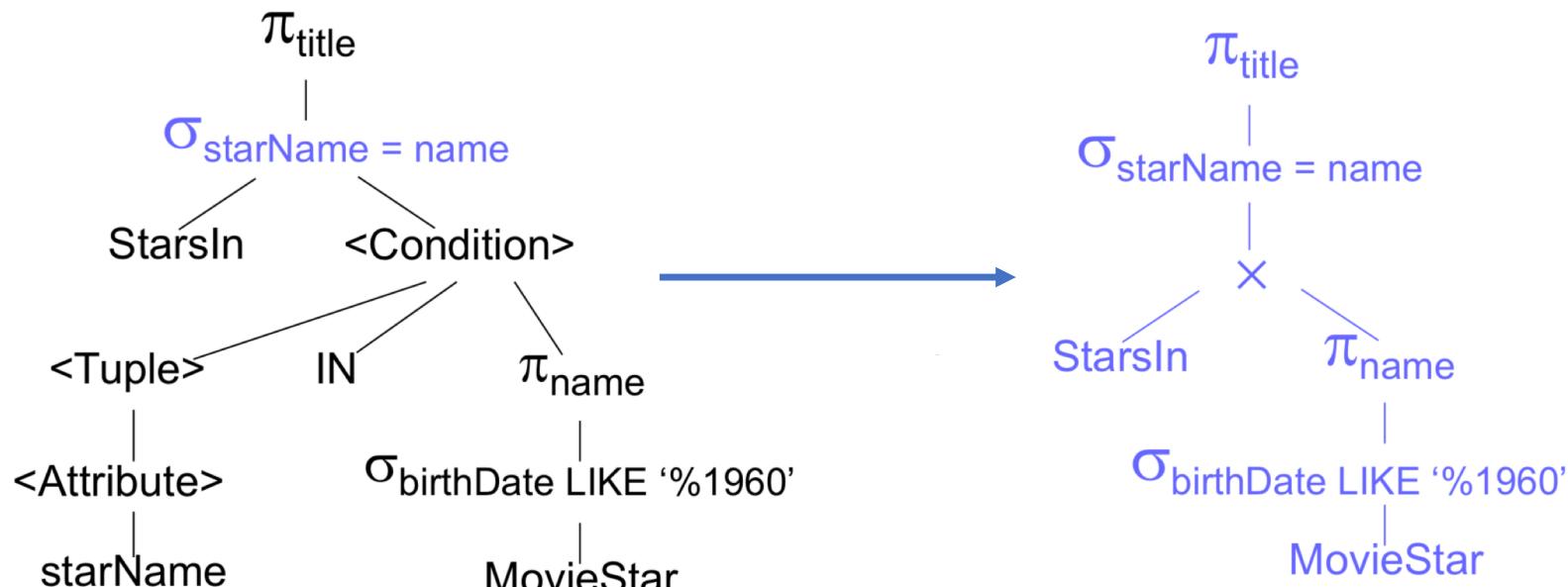


Converting sub-queries

Example

- Replace <Condition> with the subquery tree
- Replace two-argument selection with one-argument selection σ_C , where C is $\text{starName} = \text{name}$
- Let σ_C operate on the product of StarsIn and MovieStar as an argument

```
SELECT title FROM StarsIn  
WHERE starName IN  
(SELECT name FROM MovieStar  
WHERE birthDate LIKE '%1960');
```

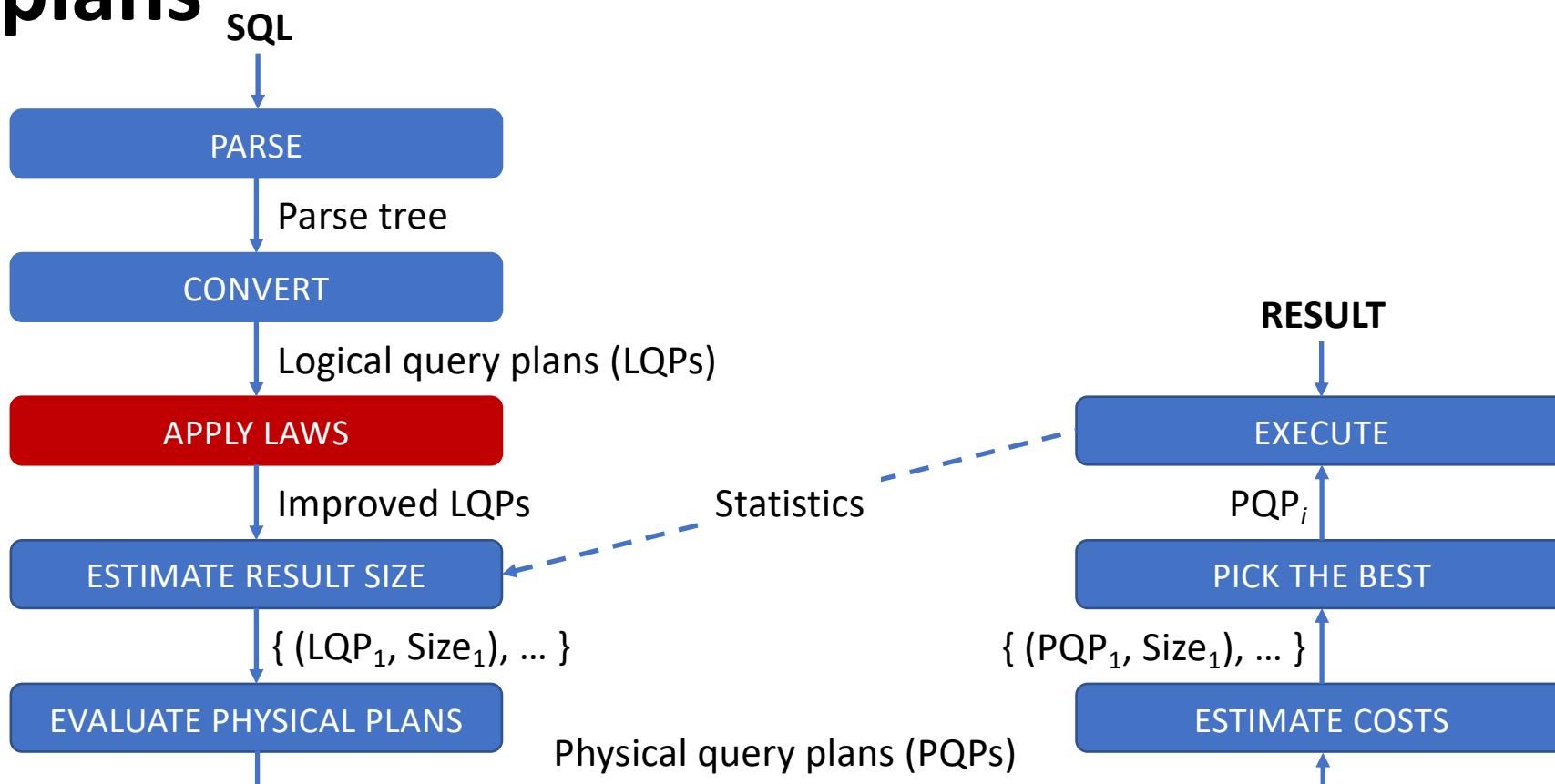


Converting sub-queries – some notes

- Sub-query conversion becomes more complicated if the sub-query is related to values defined outside the scope of the sub-query
- We must then create a relation with extra attributes for comparison with the external attributes
- The extra attributes are later removed using projections
- In addition, all duplicate tuples must be removed



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 I/Os
- 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 I/O
- 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

$\sigma_{A=2}(R \bowtie S)$

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

$\sigma_{A=2}(R) \bowtie S$

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

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

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

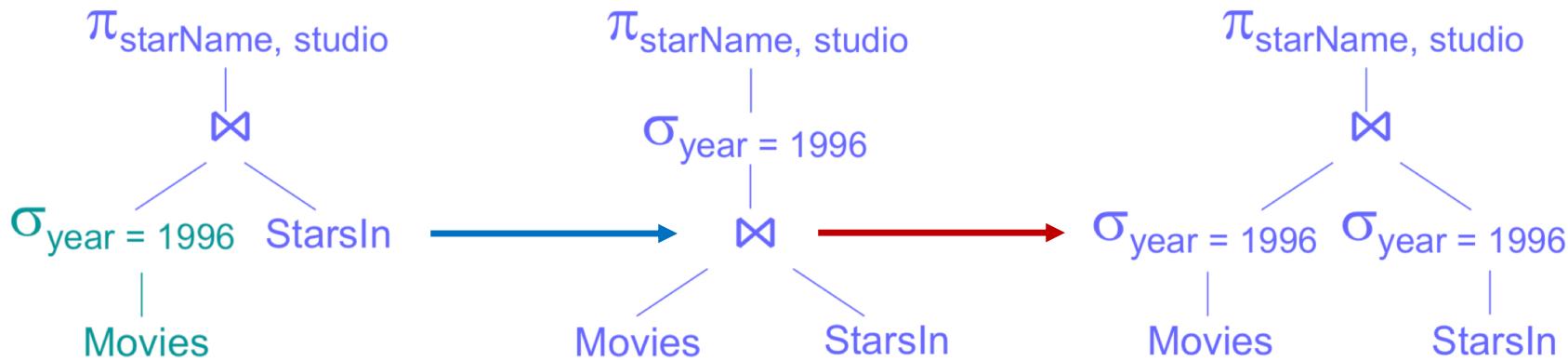
A	B	C	...	X
2	c	6	...	2



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;



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. $R \bowtie S$
 - $R(a,b,c,d,e,\dots, k)$, $T(R) = 10.000$, $S(a,l,m,n,o,\dots,z)$, $T(S) = 100$
 - 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!

