IN3020/4020 – Database Systems
Spring 2021, Weeks 4.2-5.2

Query Compilation – Parts 1-3

Egor V. Kostylev

Based upon slides by E. Thorstensen and M. Naci Akkøk

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Query Compilation: Two Parts

Part 1:

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

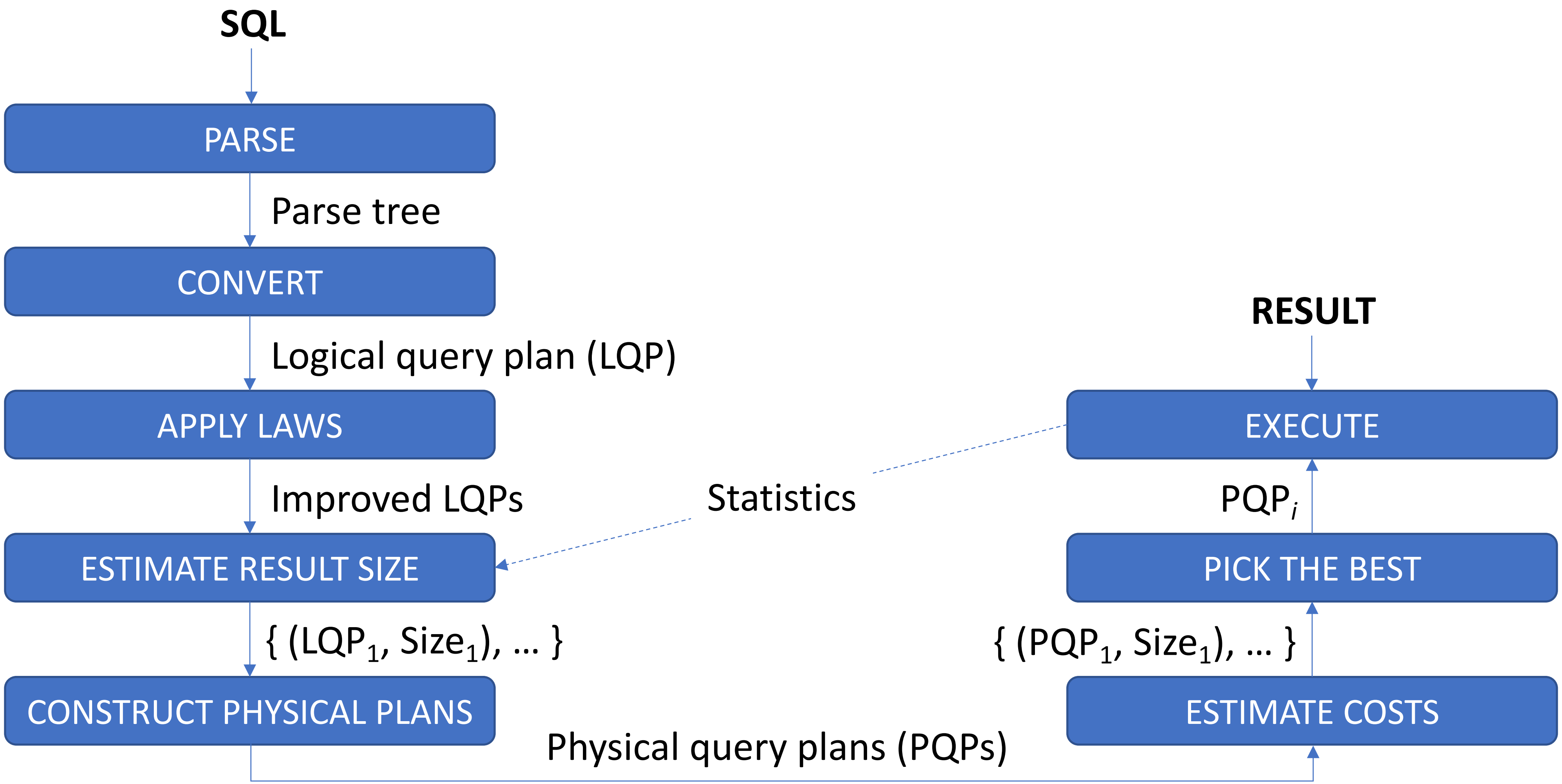
Part 2:

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

Materials to Read

- Part 8, Chapter 19 (and parts of 18) of the Book (Elmasri & Navathe, «Fundementals of Database Systems»)
- Parsing: not covered in the Book, can be read in any book on Compilers (or Wikipedia)
- NOTE: I do not follow any of them line by line

Overview: The (Typical) Journey of a Query



SQL

PARSE

Parse tree

CONVERT

Logical query plan (LQP)

APPLY LAWS

Improved LQPs

ESTIMATE RESULT SIZE

{ (LQP$_1$, Size$_1$), … }

CONSTRUCT PHYSICAL PLANS

Physical query plans (PQPs)

ESTIMATE COSTS

{ (PQP$_1$, Size$_1$), … }

PICK THE BEST

PQP$_i$

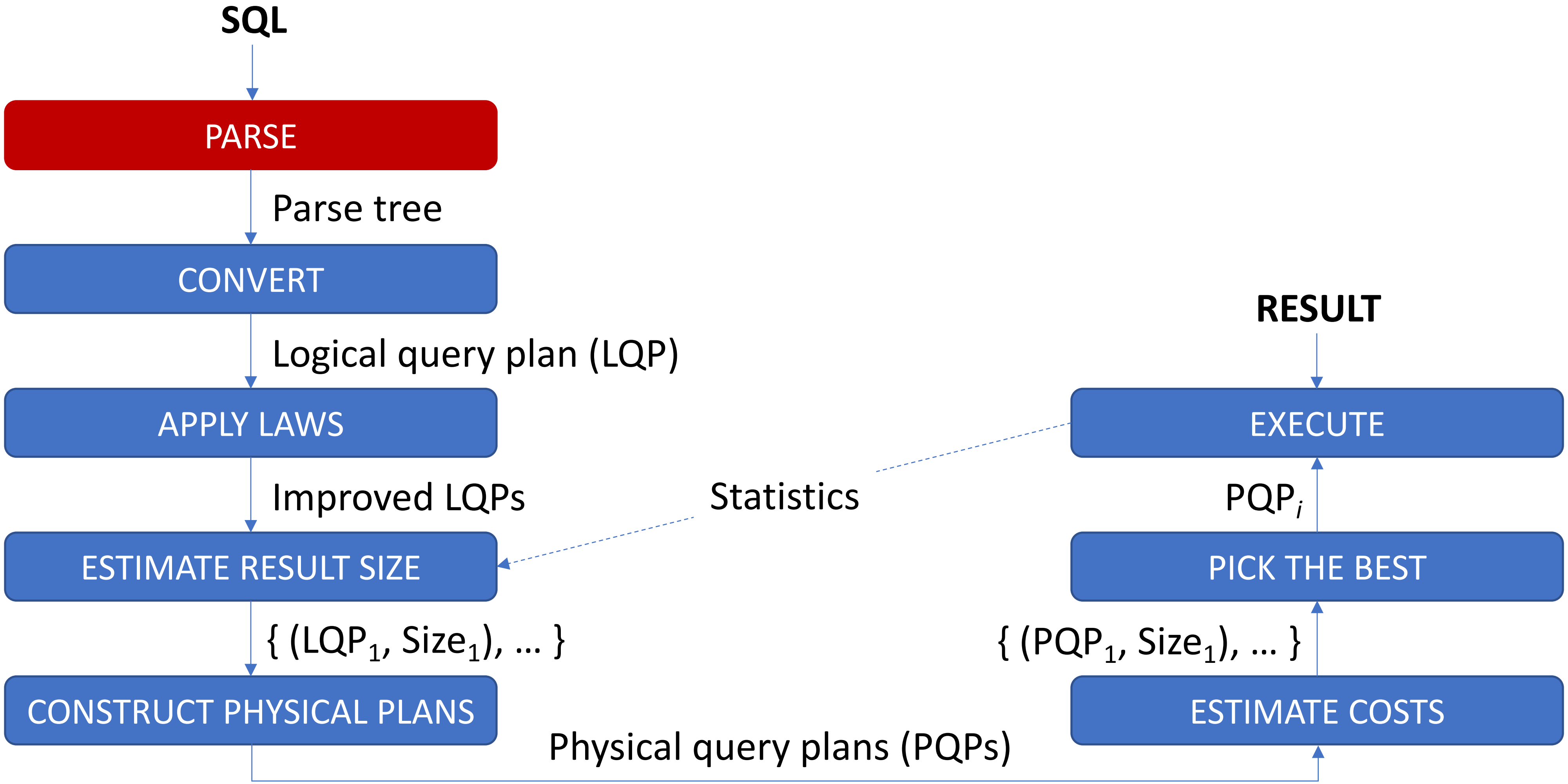
EXECUTE

RESULT

Statistics

Parsing

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



SQL

PARSE

Parse tree

CONVERT

Logical query plan (LQP)

APPLY LAWS

Improved LQPs

ESTIMATE RESULT SIZE

{ (LQP$_1$, Size$_1$), … }

CONSTRUCT PHYSICAL PLANS

Physical query plans (PQPs)

ESTIMATE COSTS

{ (PQP$_1$, Size$_1$), … }

PICK THE BEST

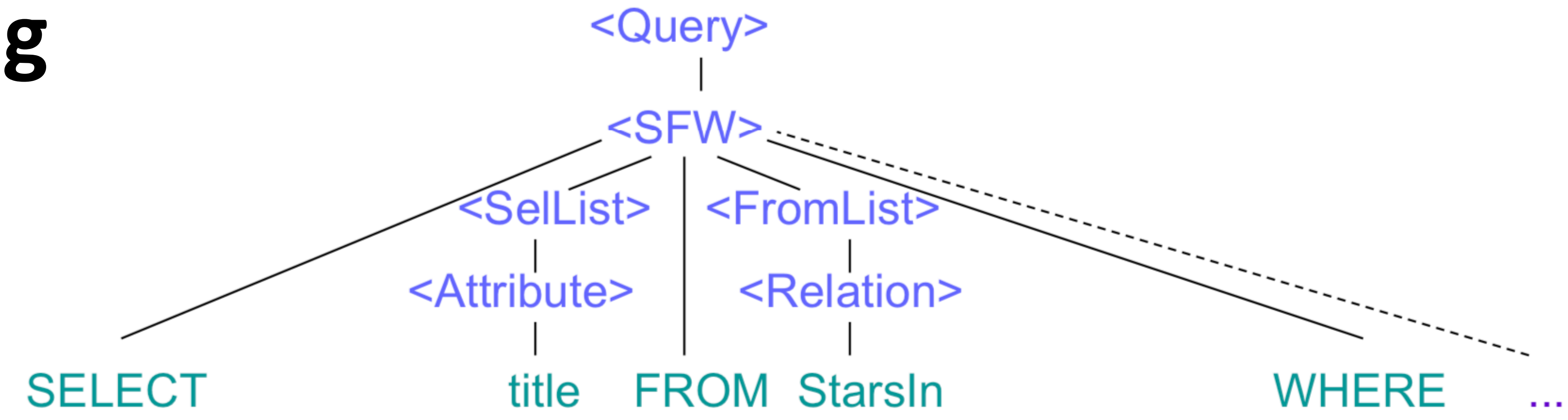
PQP$_i$

EXECUTE

Statistics

RESULT

Parsing



- Each node in a parse tree is
- 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:
  - <Query> ::= <SFW>
  - <Query> ::= **(** <Query> **)**
  - <Query> ::= ... (e.g., rules with UNION)

  - Rule 2 is typically used in sub-queries

Simple Grammar #2

- Select-From-Where:
  - <SFW> ::= SELECT <SelList> FROM <FromList> WHERE <Condition> [...]
  - [...] includes productions for GROUP BY, HAVING, ORDER BY, etc.
- Select-list:
  - <SelList> ::= <Attribute>
  - <SelList> ::= <Attribute>, <SelList>
  - <SelList> ::= ... (e.g., rules for expressions and aggregate functions)
- From-list:
  - <FromList> ::= <Relation>
  - <FromList> ::= <Relation>, <FromList>
  - <FromList> ::= ... (e.g., rules for aliasing and expressions R JOIN S)

Simple Grammar #3

- Condition:
    - <Condition> ::= <Condition> AND <Condition>
    - <Condition> ::= <Tuple> IN <Query>
    - <Condition> ::= <Attribute> = <Attribute>
    - <Condition> ::= <Attribute> LIKE <Pattern>
    - <Condition> ::= ... (e.g., rules for OR, NOT, comparison)
- Tuple:
    - <Tuple> ::= <Attribute>
    - <Tuple> ::= ... (e.g., rules for tuples with multiple attributes)
- Basic syntactic categories like <Relation>, <Attribute>, <Pattern> etc. do not have own rules, but are replaced with a name or a text string

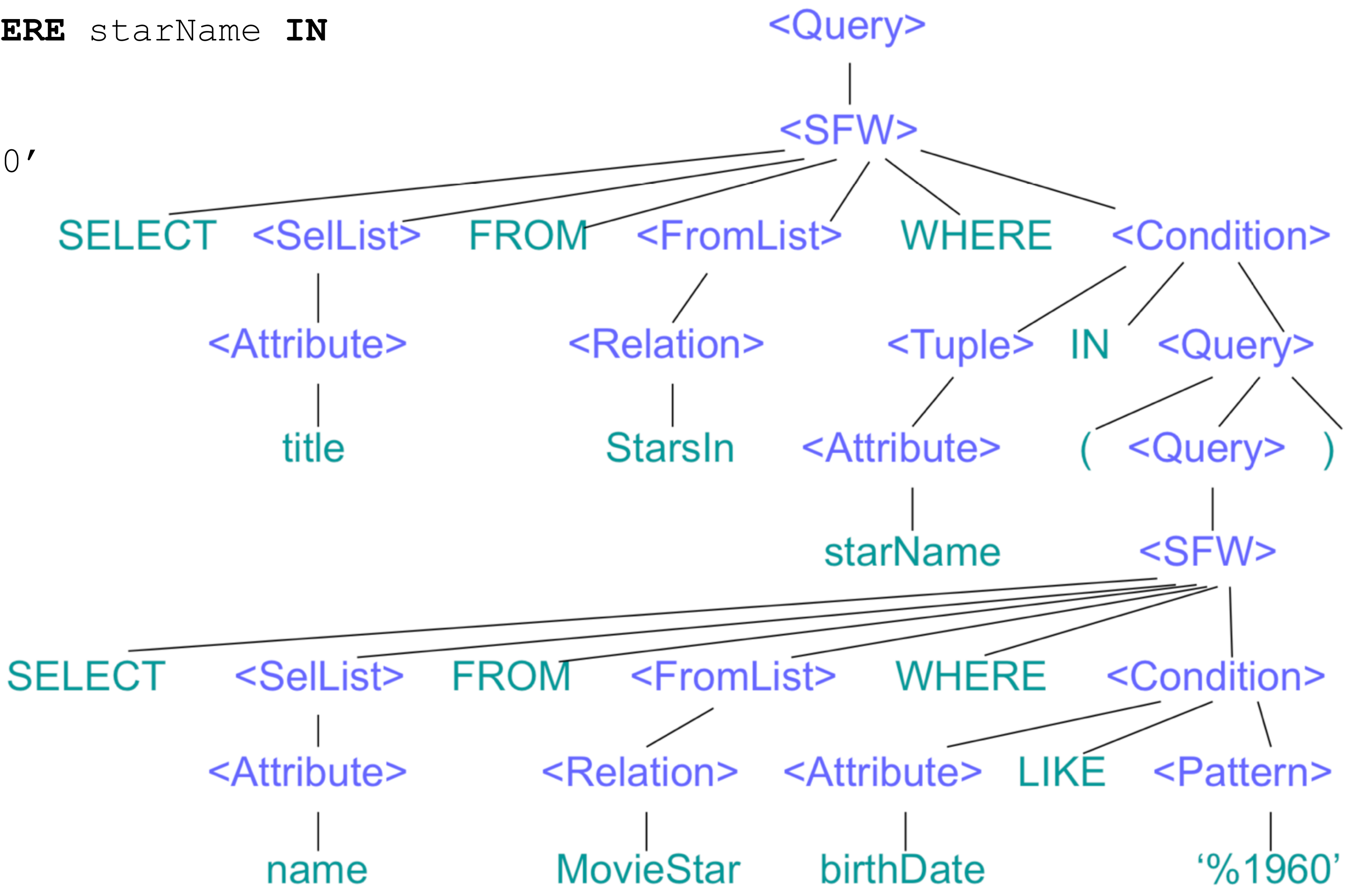
Simple Grammar: Example

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’
);
```



<Query>

<SFW>

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

<Attribute> <Relation> <Tuple> IN <Query>

title StarsIn <Attribute> (<Query>)

starName <SFW>

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

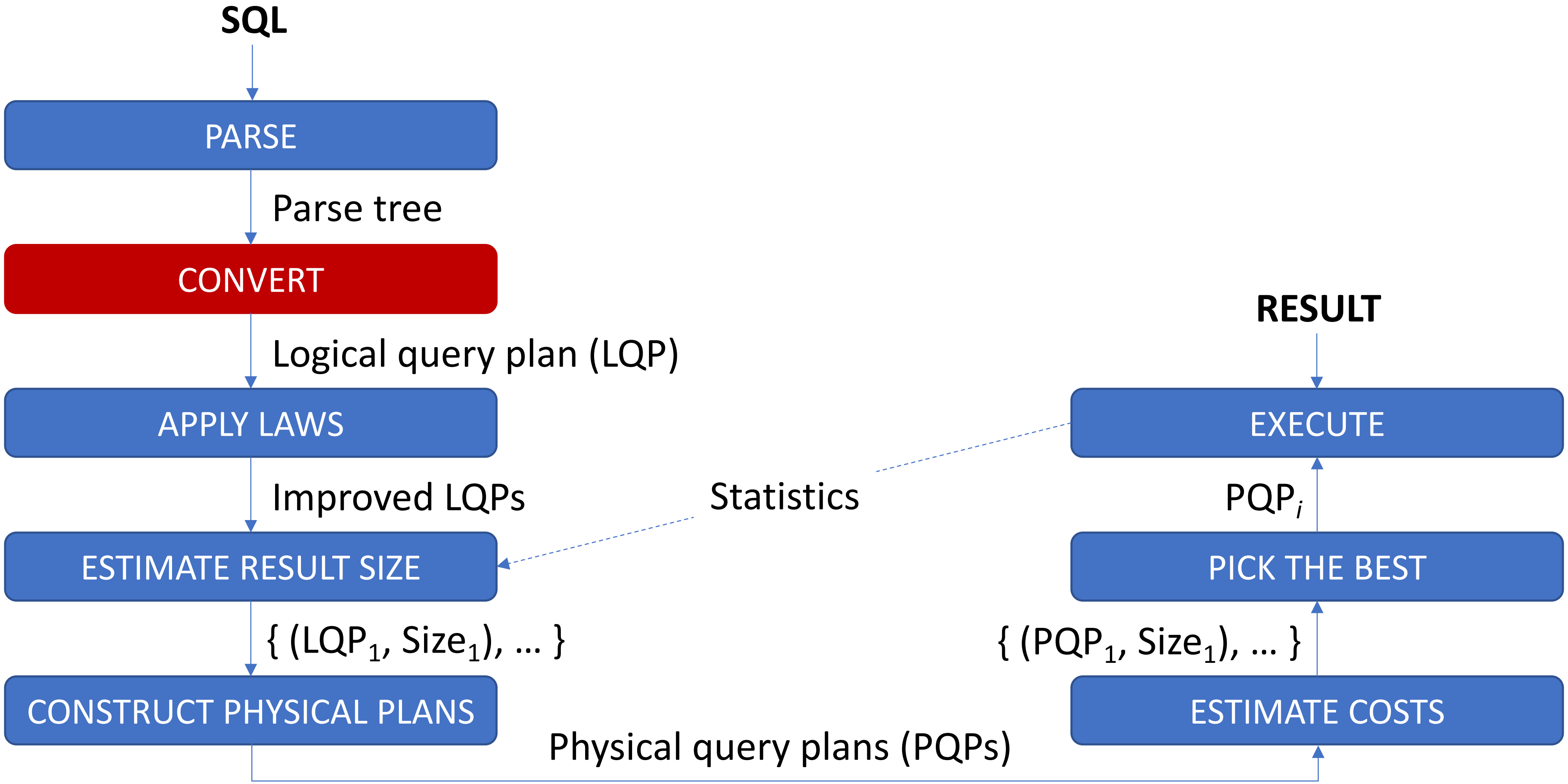
<Attribute> <Relation> <Attribute> LIKE <Pattern>

name MovieStar birthDate ‘%1960’

Pre-processor

- Checks that the query are **syntactically** correct (i.e., parses)
- Checks that the query are **semantically** correct:
  - *relations* – each relation in FROM must be a relation or a view in the schema the query is executed
    Each view must be replaced by a parsing tree.
  - *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



Converting Select-From-Where (SFW)

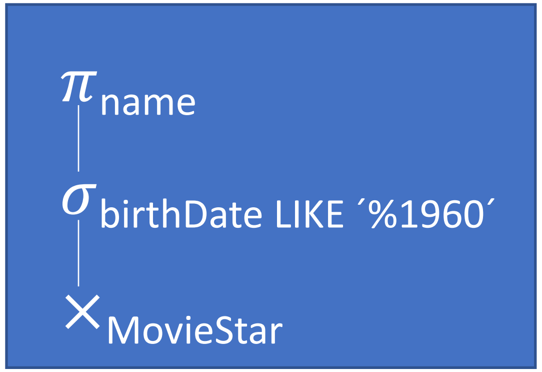
`SELECT` <SelList> `FROM` <FromList> `WHERE` <Condition>

- Replace the relations in <FromList> with the product (×) 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>

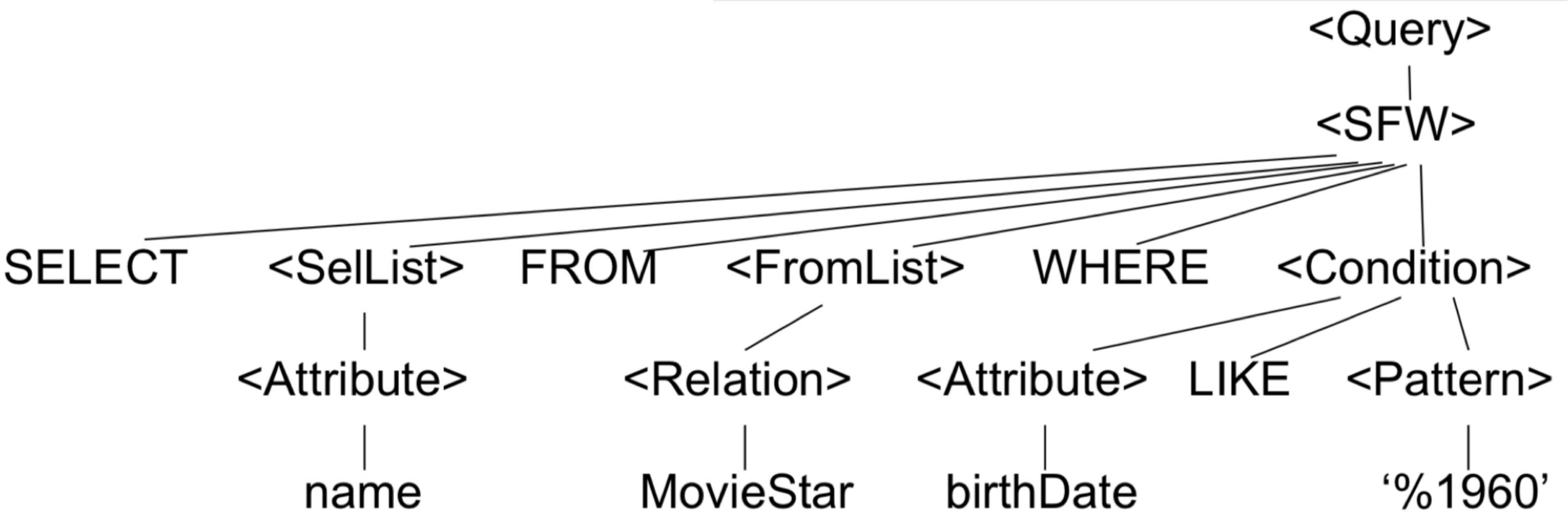
SFW conversion example

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'
```

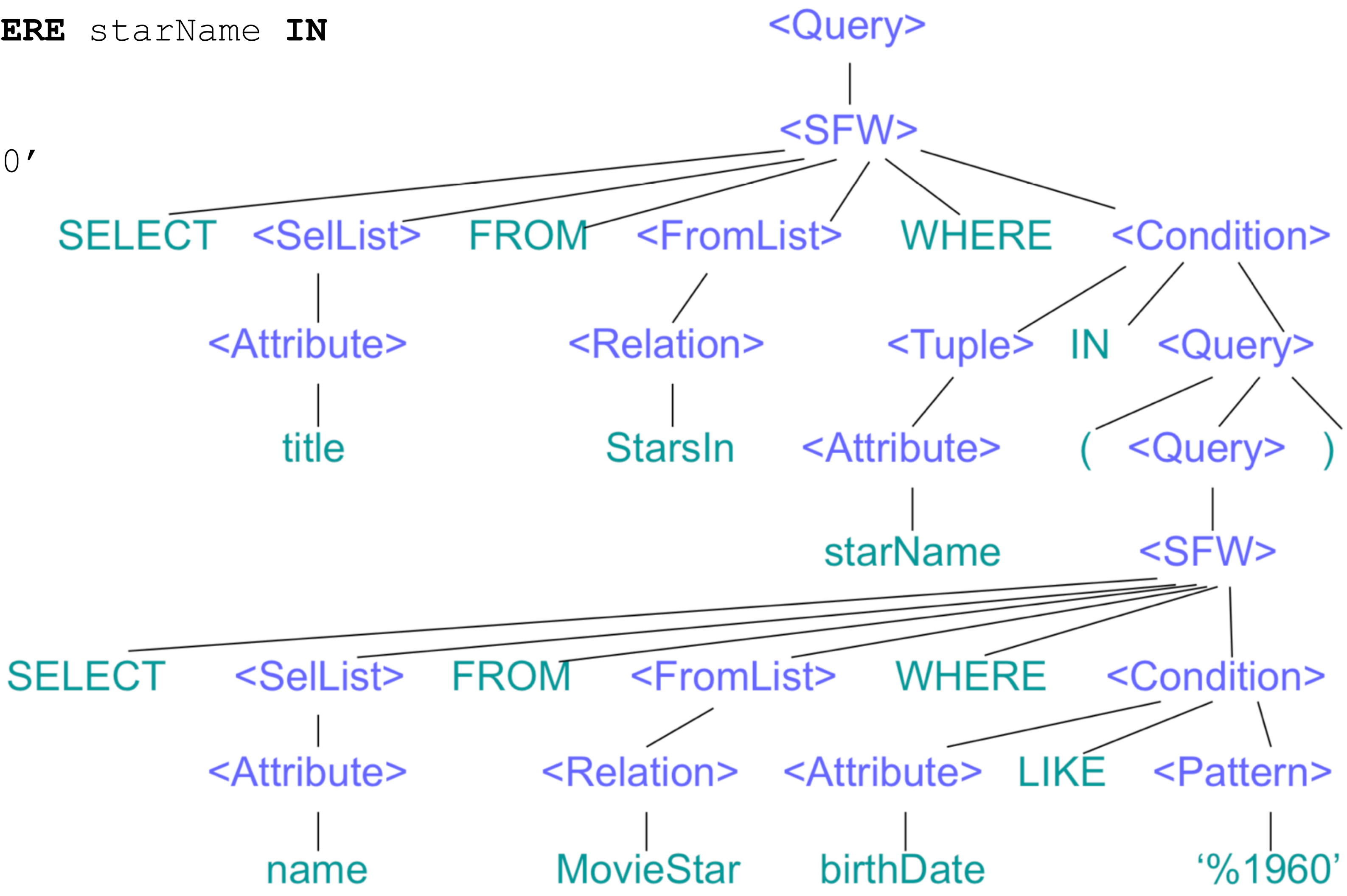


π_{name}
$\sigma_{\text{birthDate LIKE ´\%1960´}}$
$\times_{\text{MovieStar}}$



Simple Grammar: Example

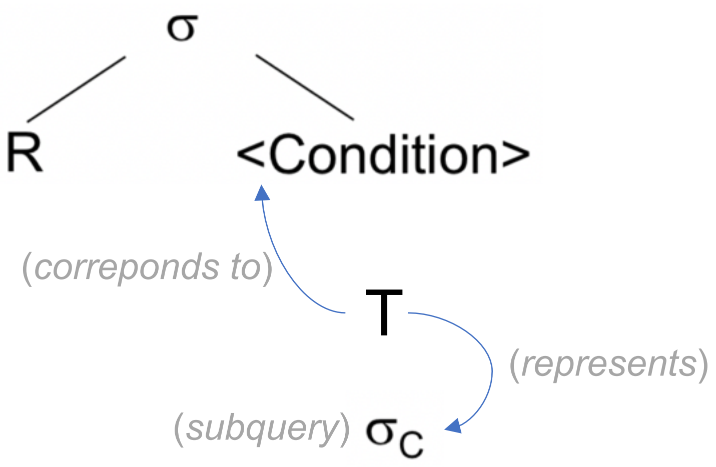
```
SELECT title FROM StarsIn WHERE starName IN
( SELECT name
  FROM MovieStar
  WHERE birthDate LIKE ‘%1960’
);
```



Converting sub-queries

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

- For subqueries,
  we use an auxiliary operator,
  the two-argument selection $\sigma(R, T)$,
  where T represents the subquery
  (i.e., that corresponds to <Condition>)

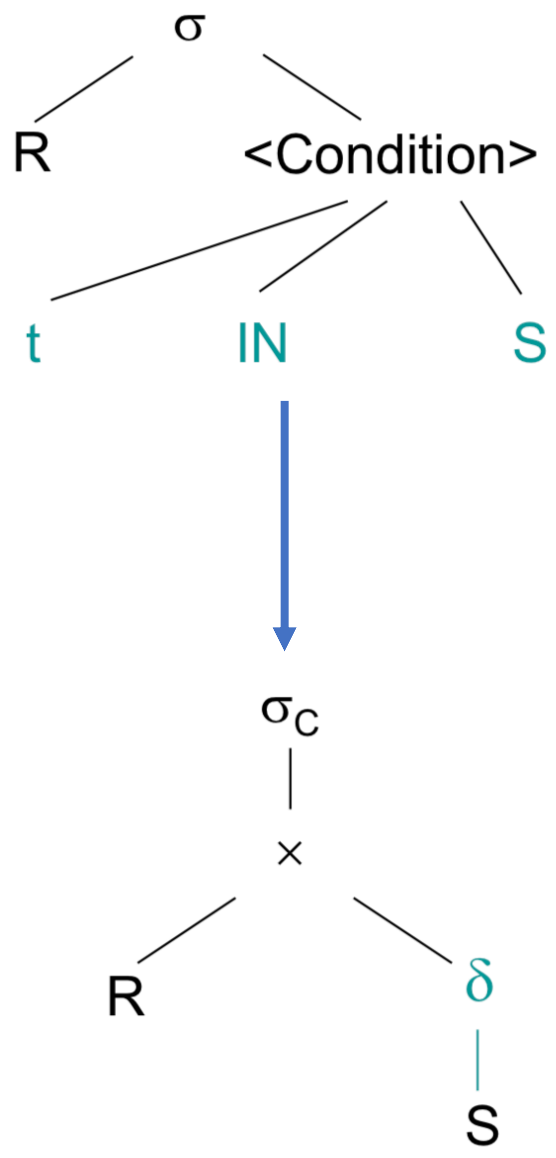


- Further processing depends on the type of the subquery

Converting sub-queries Example

Let us look at **t IN S** as one example:

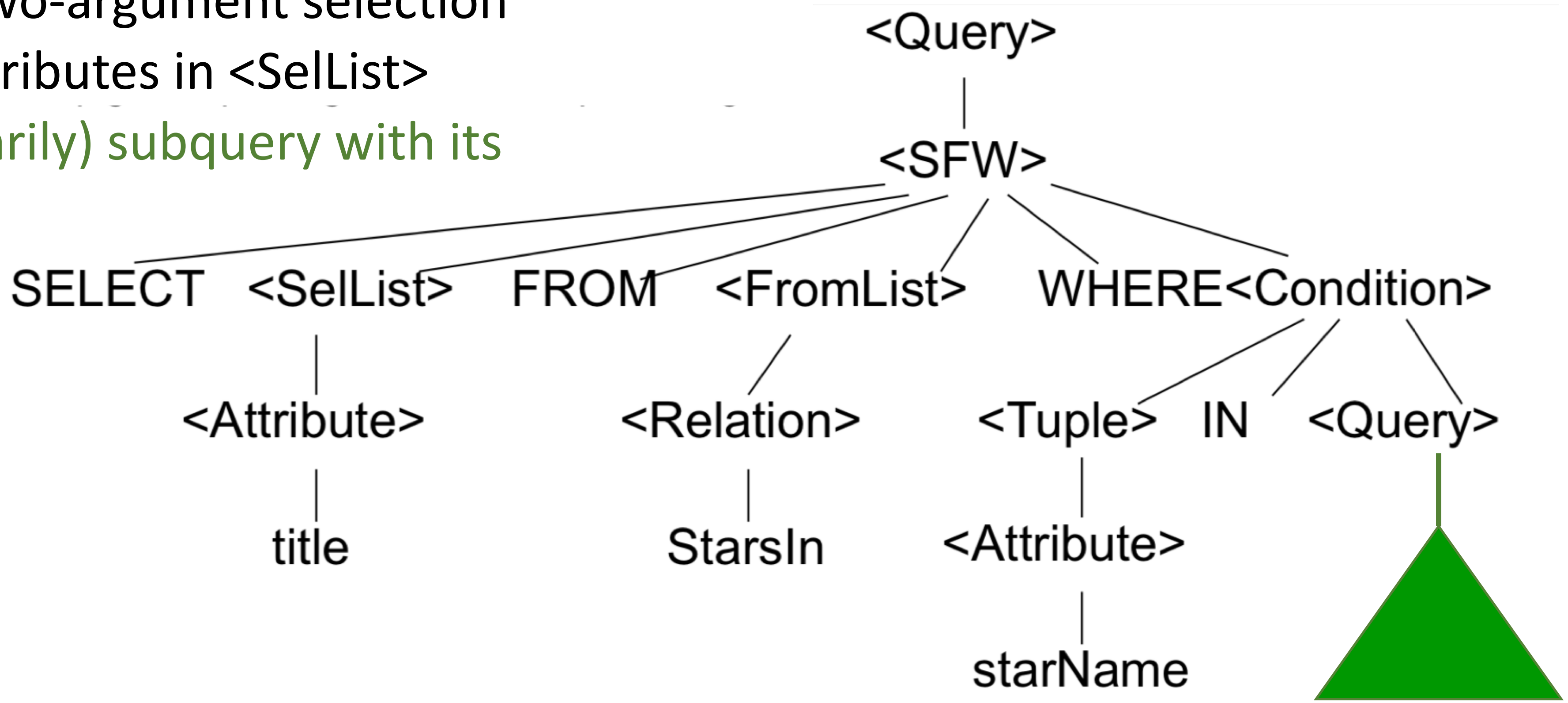
- Replace <Condition> with the tree for S
- If S can contain duplicates, we need a δ operator above S
- Replace two-argument selection with one-argument selection σ_C, where C compares each component in t with the corresponding attribute in S
- Let σ_C get the R × S as argument



Converting sub-queries Example

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

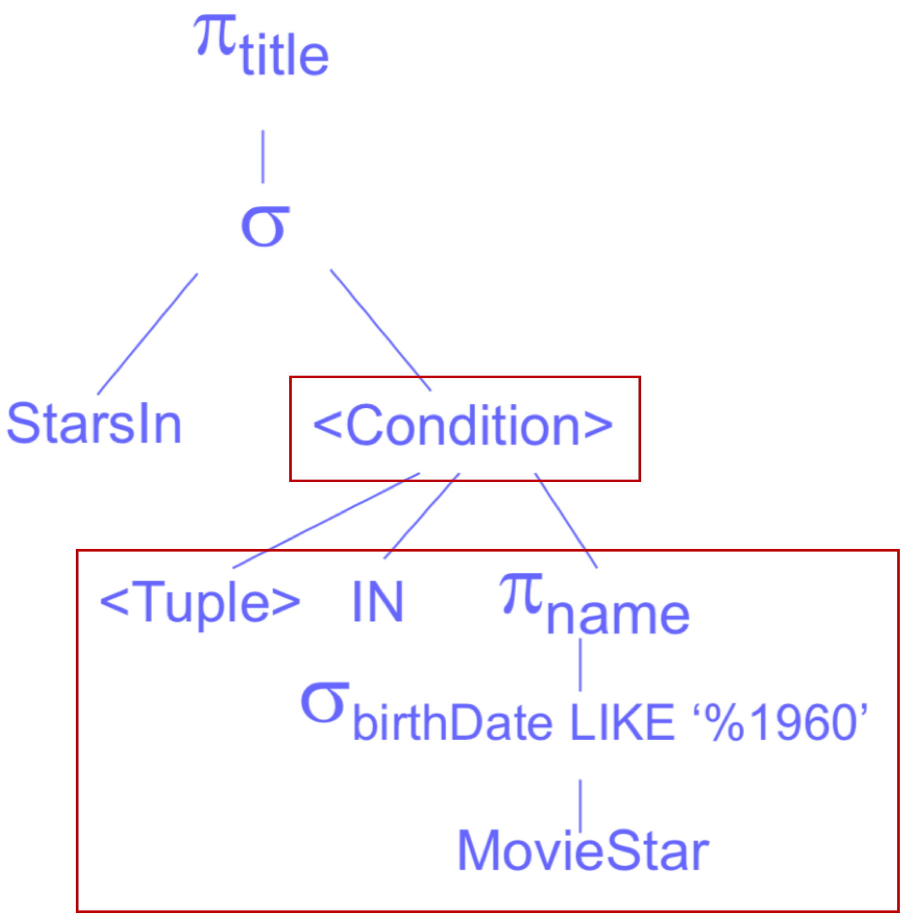
- Product of the relations in <FromList>
- Select based upon <Condition>, represented by two-argument selection
- Project on the attributes in <SelList>
- Replace (temporarily) subquery with its parse tree



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 birthDate LIKE '%1960'
);
```



π_{title}

σ

StarsIn

<Condition>

<Tuple> IN π_{name}

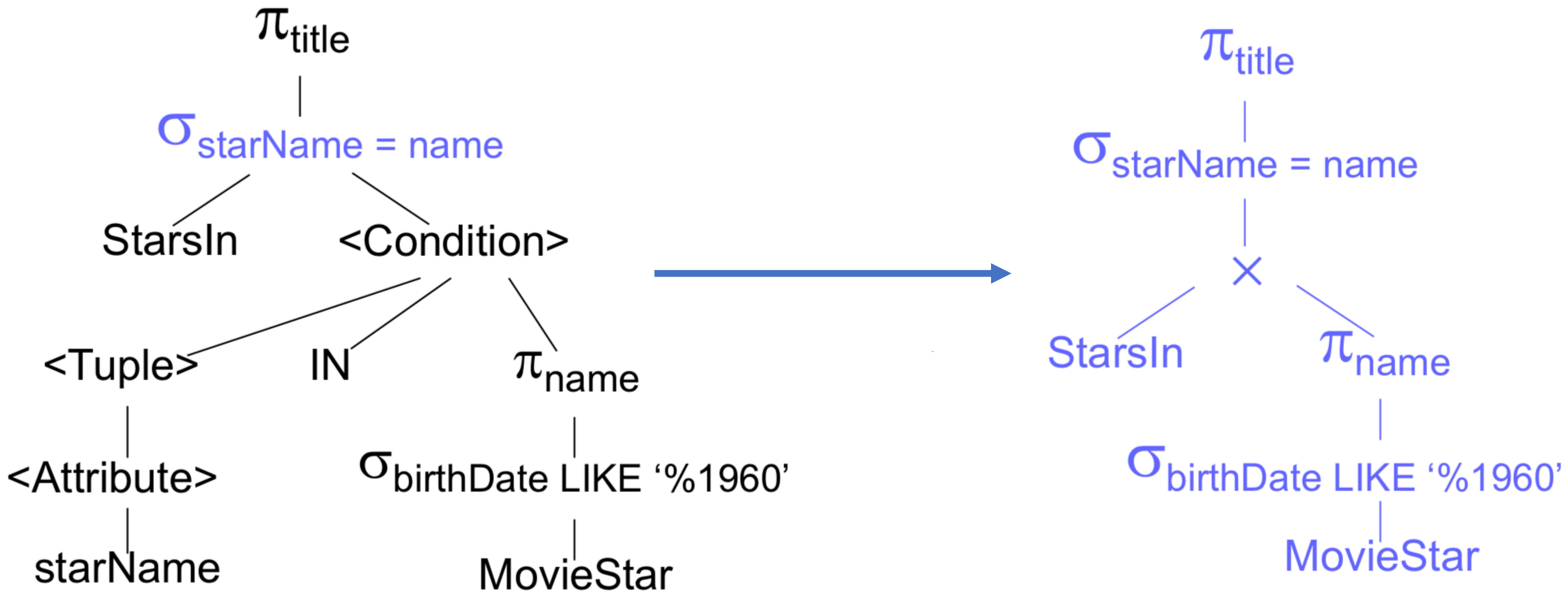
$\sigma_{birthDate\ LIKE\ '\%1960'}$

MovieStar

Converting sub-queries Example

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

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



Converting sub-queries – some notes

- Sub-query conversion becomes more complicated if the sub-query is related to values defined <u>outside</u> 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

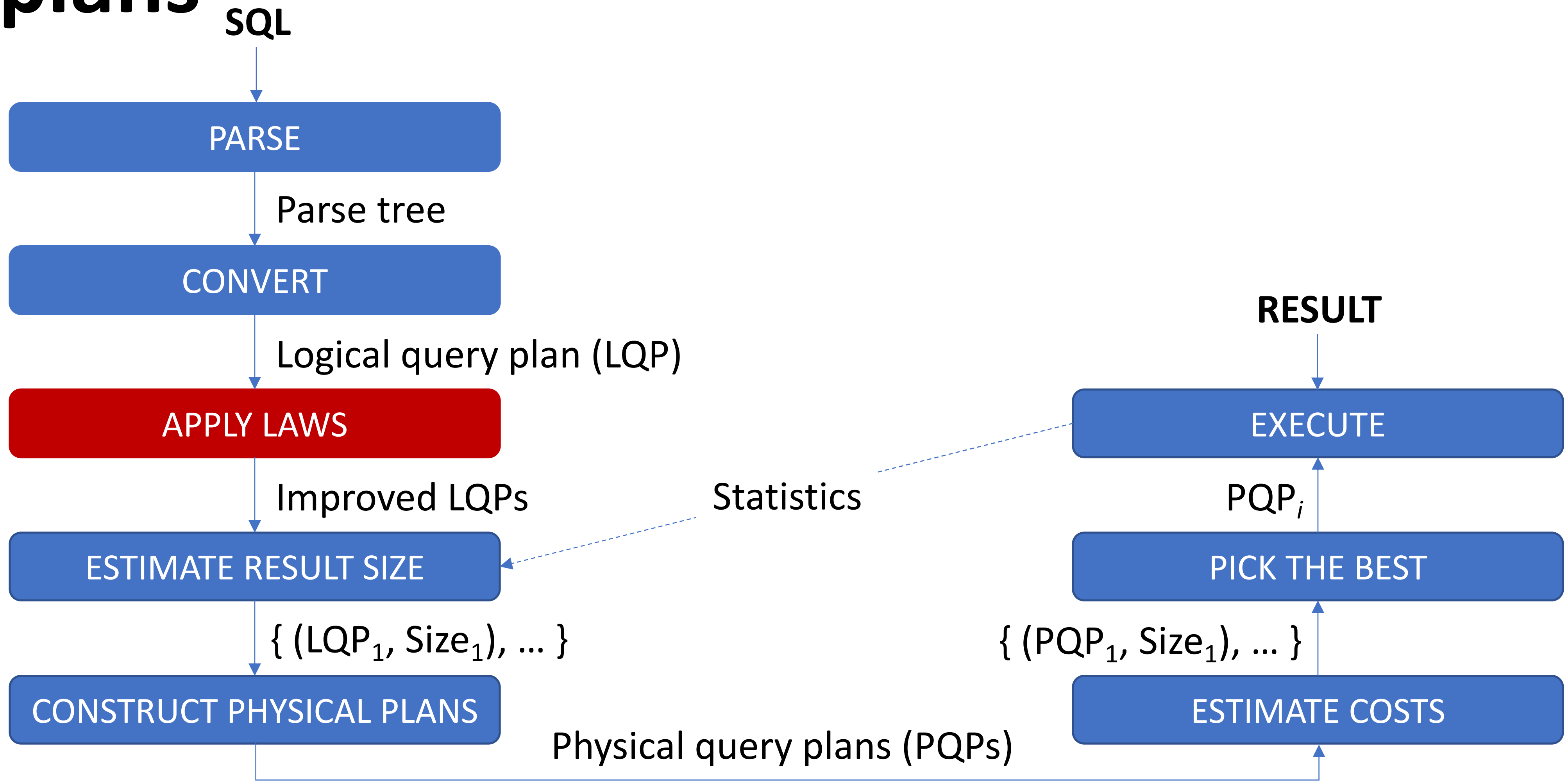
Logical and Physical query plans

- These parse trees are converted into «execution plans» in several stages

  - Logical plan: Relational algebra expressions
  - Physical plan: Actual algorithms

- These are supposed to be two distinct stages, but, the two stages often overlap in reality

Algebraic laws for improving logical query plans



SQL

PARSE

Parse tree

CONVERT

Logical query plan (LQP)

APPLY LAWS

Improved LQPs

ESTIMATE RESULT SIZE

{ (LQP$_1$, Size$_1$), … }

CONSTRUCT PHYSICAL PLANS

Physical query plans (PQPs)

ESTIMATE COSTS

{ (PQP$_1$, Size$_1$), … }

PICK THE BEST

PQP$_i$

EXECUTE

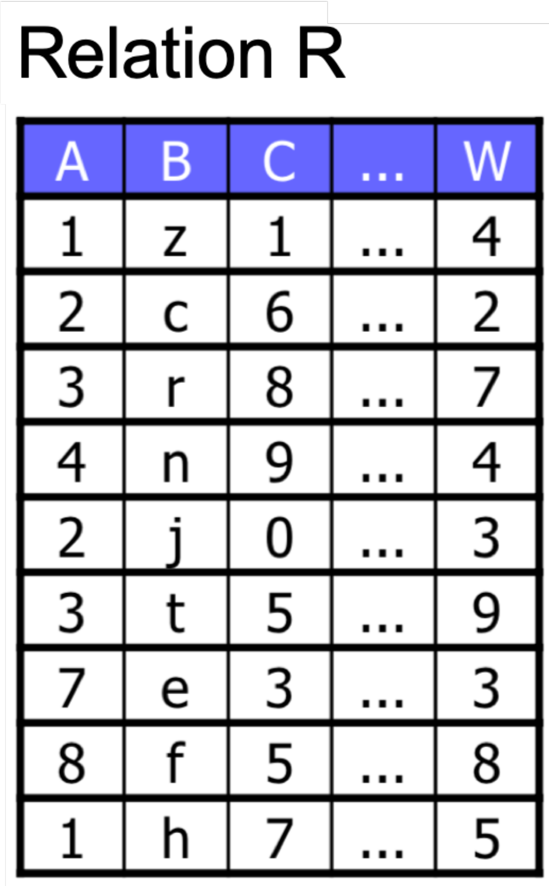
RESULT

Statistics

Logical query plans – Example

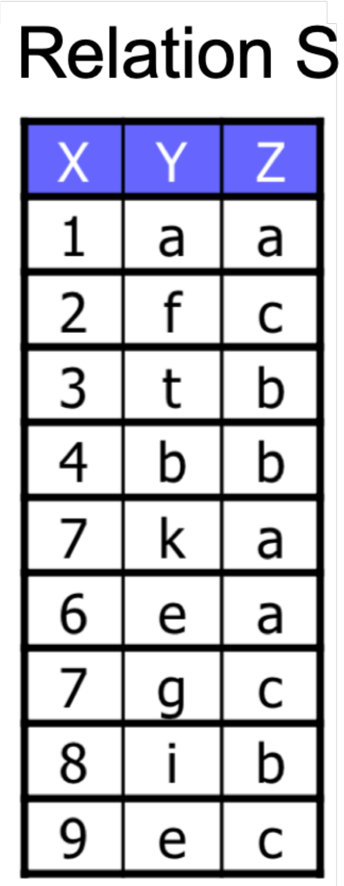
```
SELECT B, C, Y
FROM R, S
WHERE W = X AND A = 3 AND Z =´a´
```

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



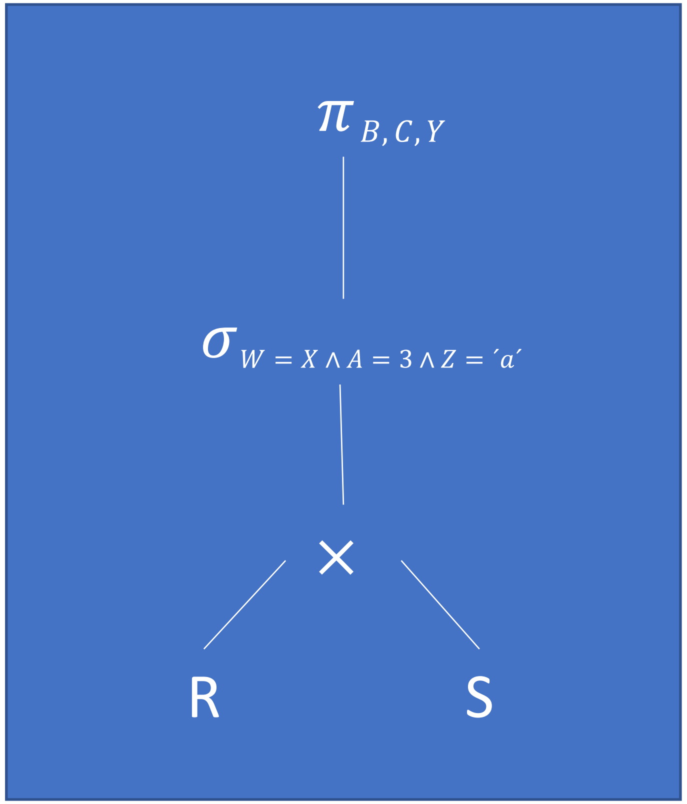
X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

Logical query plans – Strategy 1

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

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:

\# attributes = #R-attributes + #S-attributes = 23 + 3 = 26

\# tuples = #R-tuples x #S-tuples = 9 x 9 = 81

Logical query plans – Strategy 1

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

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	j	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

RESULT

B	C	Y
r	8	k

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

Logical query plans – Strategy 2

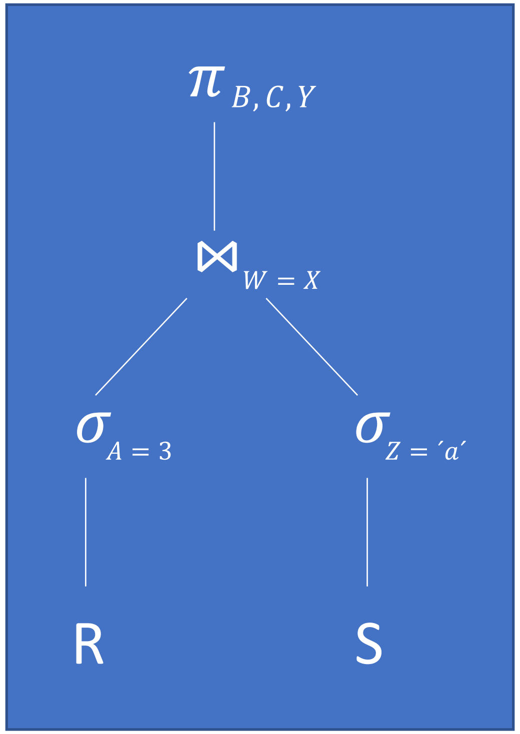
```
SELECT B, C, Y
FROM R, S
WHERE W = X AND A = 3 AND Z =´a´
```

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

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

Strategy 1 for comparison:

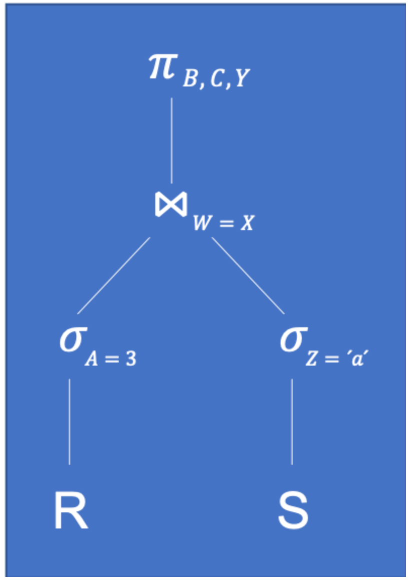
$$\pi_{B,C,Y}(\sigma_{W=X \wedge A=3 \wedge Z=´a´}(R \times S))$$



Logical query plans – Strategy 2

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

$$\underset{③}{\pi_{B,C,Y}}\left(\left(\underset{①}{\sigma_{A=3}}(R)\right)\underset{②}{\bowtie_{W=X}}\left(\underset{①}{\sigma_{Z='a'}}(S)\right)\right)$$

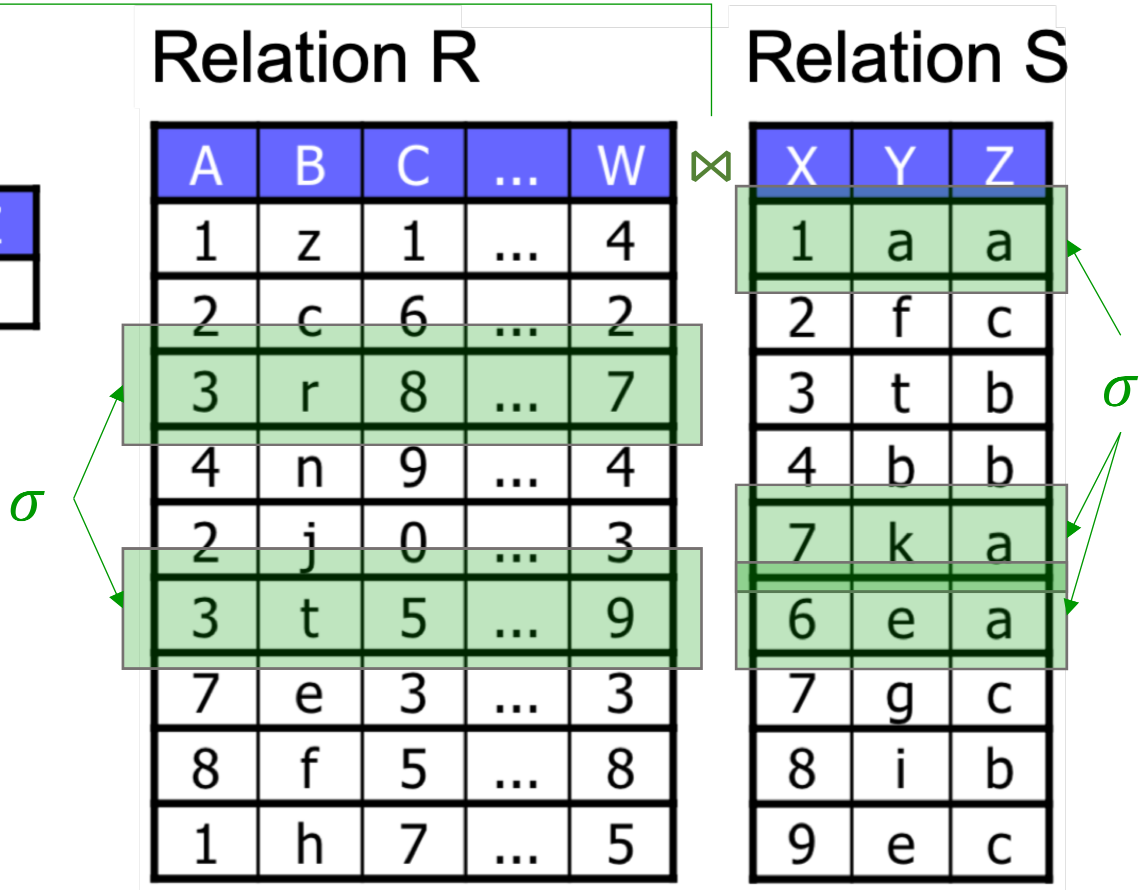


A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

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

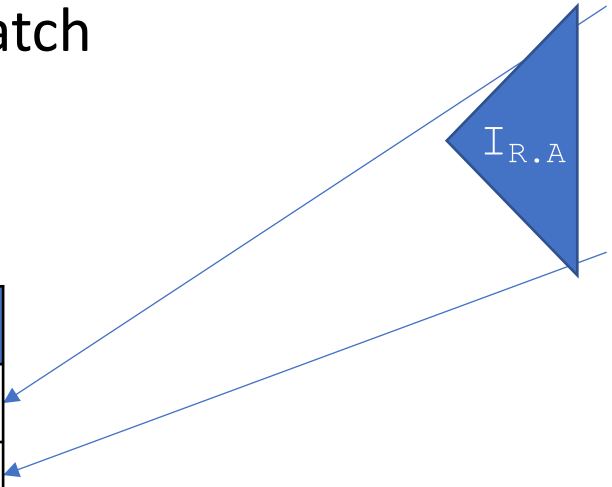
```
SELECT B, C, Y
FROM R, S
WHERE W = X AND A = 3 AND Z =´a´
```

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

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



r_1

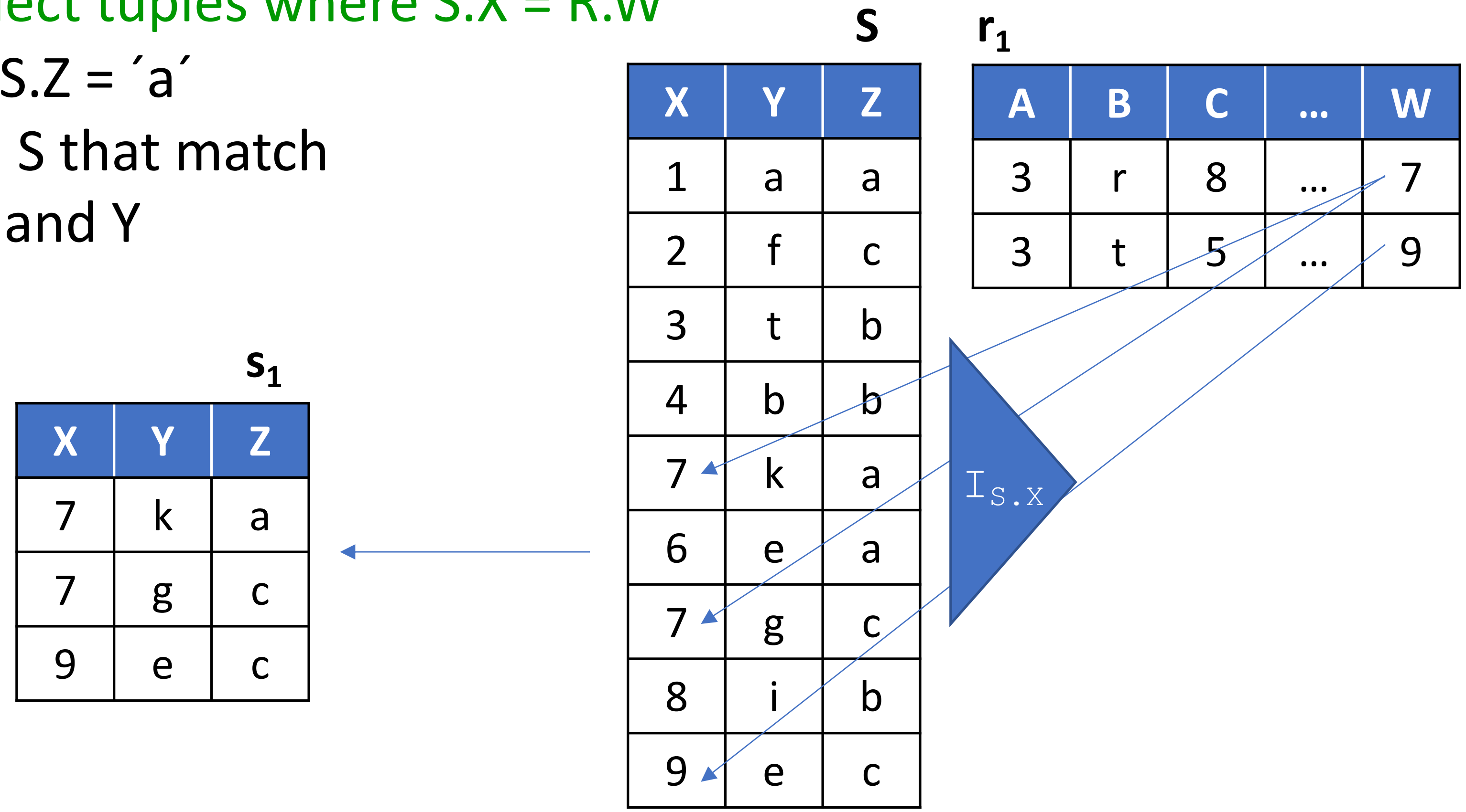
A	B	C	…	W
3	r	8	…	7
3	t	5	…	9

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



s_1

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

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

r_1

A	B	C	…	W
3	r	8	…	7
3	t	5	…	9

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.

S_1

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

→

S_2

X	Y	Z
7	k	a

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

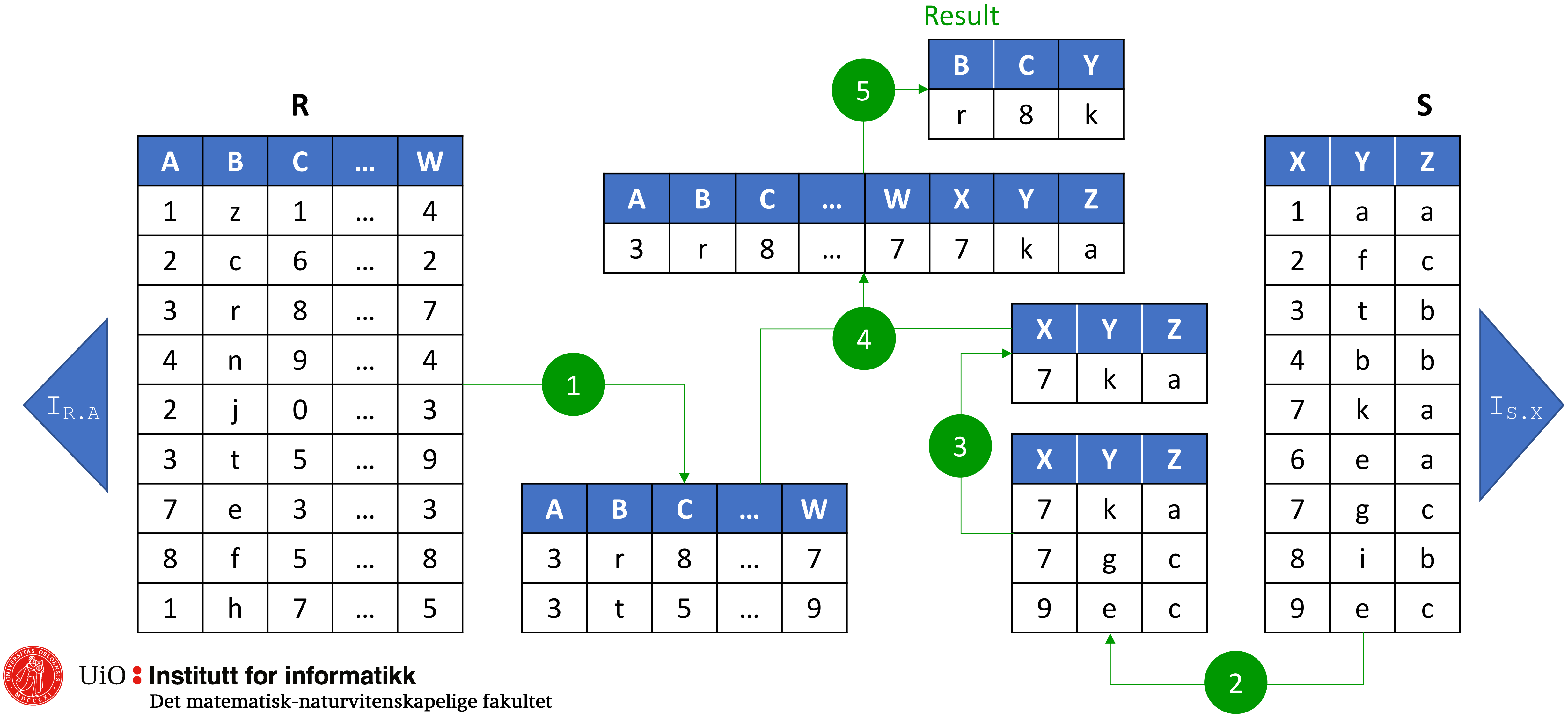
A	B	C	…	W	X	Y	Z
3	r	8	…	7	7	k	a

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

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

A	B	C	…	W
3	r	8	…	7
3	t	5	…	9

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

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

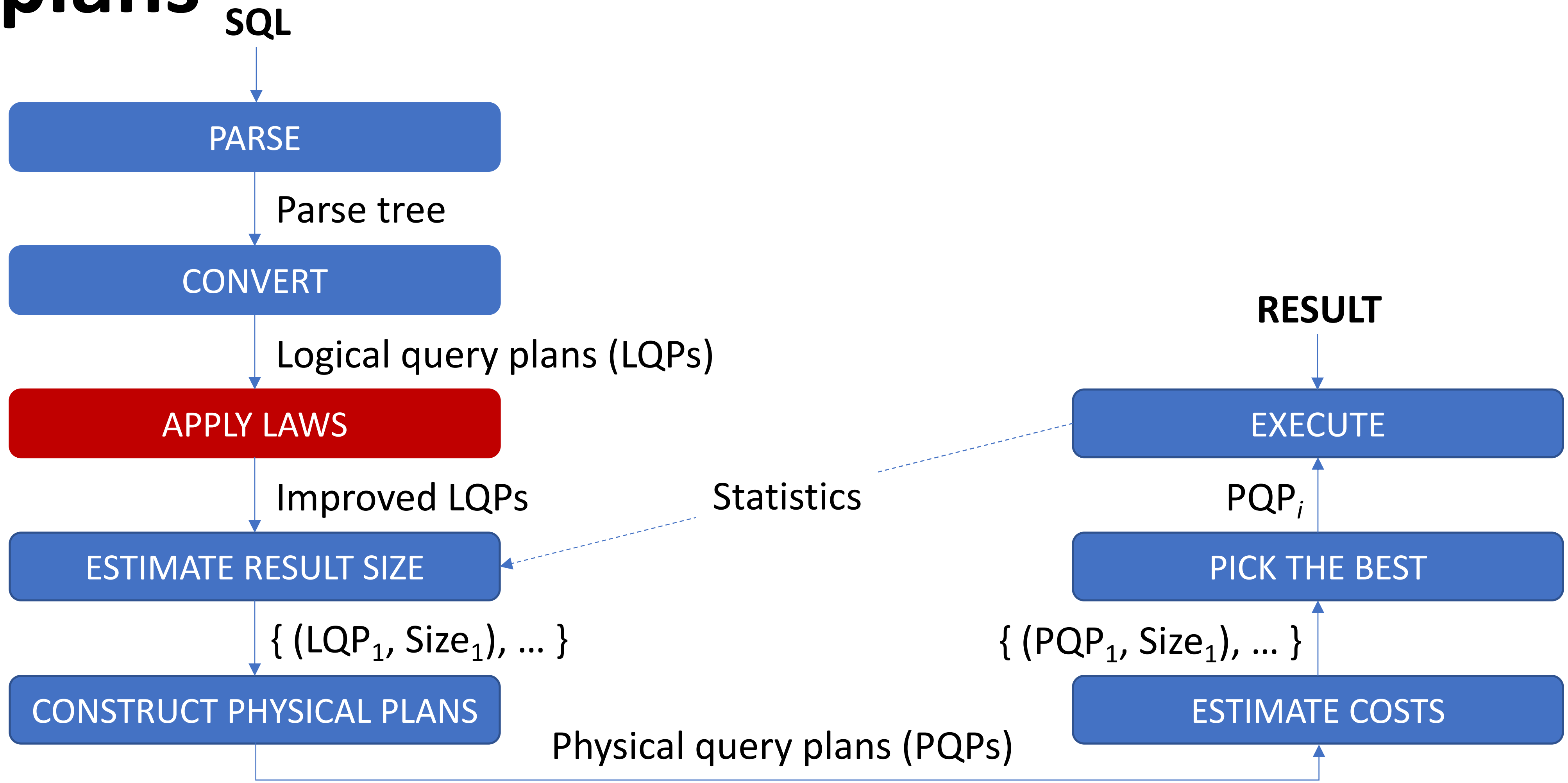
X	Y	Z
7	k	a

A	B	C	…	W	X	Y	Z
3	r	8	…	7	7	k	a

Result

B	C	Y
r	8	k

Algebraic laws for improving logical query plans



SQL

PARSE

Parse tree

CONVERT

Logical query plans (LQPs)

APPLY LAWS

Improved LQPs

Statistics

ESTIMATE RESULT SIZE

{ (LQP$_1$, Size$_1$), … }

CONSTRUCT PHYSICAL PLANS

Physical query plans (PQPs)

ESTIMATE COSTS

{ (PQP$_1$, Size$_1$), … }

PICK THE BEST

PQP$_i$

EXECUTE

RESULT

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 order of applications 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 (formally)
    - Example: Given the relationships R (a, b), S (b', c), T (c, d), we have $(R \bowtie_{b<b'} S) \bowtie_{a<d} T \neq R \bowtie_{b<b'} (S \bowtie_{a<d} T)$

Order of joins

Does the order of joins (incluing products) 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

Algebraic laws for union and intersection

- Union and intersection are commutative and associative:
  - $R \cup S = S \cup R$ $\quad$ $R \cup (S \cup T) = (R \cup S) \cup T$
  - $R \cap S = S \cap R$ $\quad$ $R \cap (S \cap T) = (R \cap S) \cap T$
- Union distributes over intersection:
  - $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$
- Intersection distributes over union only for sets(!):
  - $R \cap_S (S \cup_S T) = (R \cap_S S) \cup_S (R \cap_S T)$
  - $R \cap_B (S \cup_B T) \neq (R \cap_B S) \cup_B (R \cap_B T)$

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) = σ_a (σ_b (R))
  - $\sigma_{a \text{ OR } b}$ (R) = (σ_a (R)) $\cup_S$ (σ_b (R)), where R is a set and $\cup_S$ is a set union
    - Splitting OR works only when R is a set, but 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:
  - σ_a (σ_b (R)) = σ_b (σ_a (R))

Pushing selection

- When selection is pushed down the tree, then ...
- It must be pushed to both arguments for
  - **Union:** $\sigma_a(R \cup S) = \sigma_a(R) \cup \sigma_a(S)$
- 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:** $\sigma_a(R \bowtie S) = \sigma_a(R) \bowtie \sigma_a(S) = R \bowtie \sigma_a(S) = \sigma_a(R) \bowtie S$ (only if make sence!)
  - **Theta join:** $\sigma_a(R \bowtie_b S) = \sigma_a(R) \bowtie_b \sigma_a(S) = R \bowtie_b \sigma_a(S) = \sigma_a(R) \bowtie_b S$ (same!)
  - **Cartesian product:** similarly, but be careful with renaming

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
  ((23 + 3) x 2) = 52 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 x 4 items = 4 operations

R

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

S

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

$R \bowtie S$

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

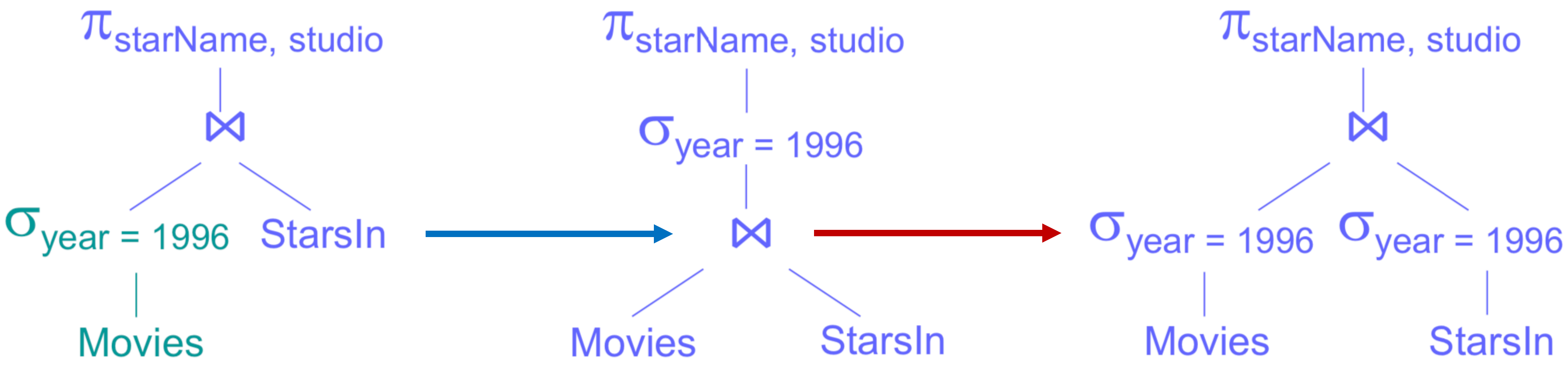
$\sigma_{A=2}(R)$

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;
```



$\pi_{starName, studio}$
$\bowtie$
$\sigma_{year = 1996}$ StarsIn
Movies

$\pi_{starName, studio}$
$\sigma_{year = 1996}$
$\bowtie$
Movies StarsIn

$\pi_{starName, studio}$
$\bowtie$
$\sigma_{year = 1996}$ $\sigma_{year = 1996}$
Movies 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 those of L that are in R
  - N contains the join attributes and those of L that are in S
- $\pi_L(R \bowtie_C S) = \pi_L(\pi_M(R) \bowtie_C \pi_N(S))$ if
  - M contains the join attributes and the attributes in C and L that are in R
  - N contains the join attributes and the attributes in C and L that are in S
- $\pi_L(R \times S) = \pi_M(R) \times \pi_N(S)$ if
  - M contains the attributes of L that are in R (appropriately renamed)
  - N contains the attributes of L that are in S (appropriately renamed)

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:

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

Examples

- $\pi_L(\sigma_{R.a = S.a}(R \times S))$ vs. $R \bowtie S$
  - R(a,b,c,d,e,..., k), #Tuples(R) = 10,000, S(a,l,m,n,o,...,z), #Tuples(S) = 100
  - Each attribute takes 1 byte, a is a candidate (e.g., primary) key in both R and S
  - Assumes that each tuple in S find a single match in R, i.e., 100 tuples 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
  - Bag union and difference
  - Projection

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 (AND or OR), split into multiple selections and push each one as far down the tree as possible
- Push projections as far down as possible
- Combine selections and Cartesian products to an appropriate 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!

Query Compilation (in two parts)

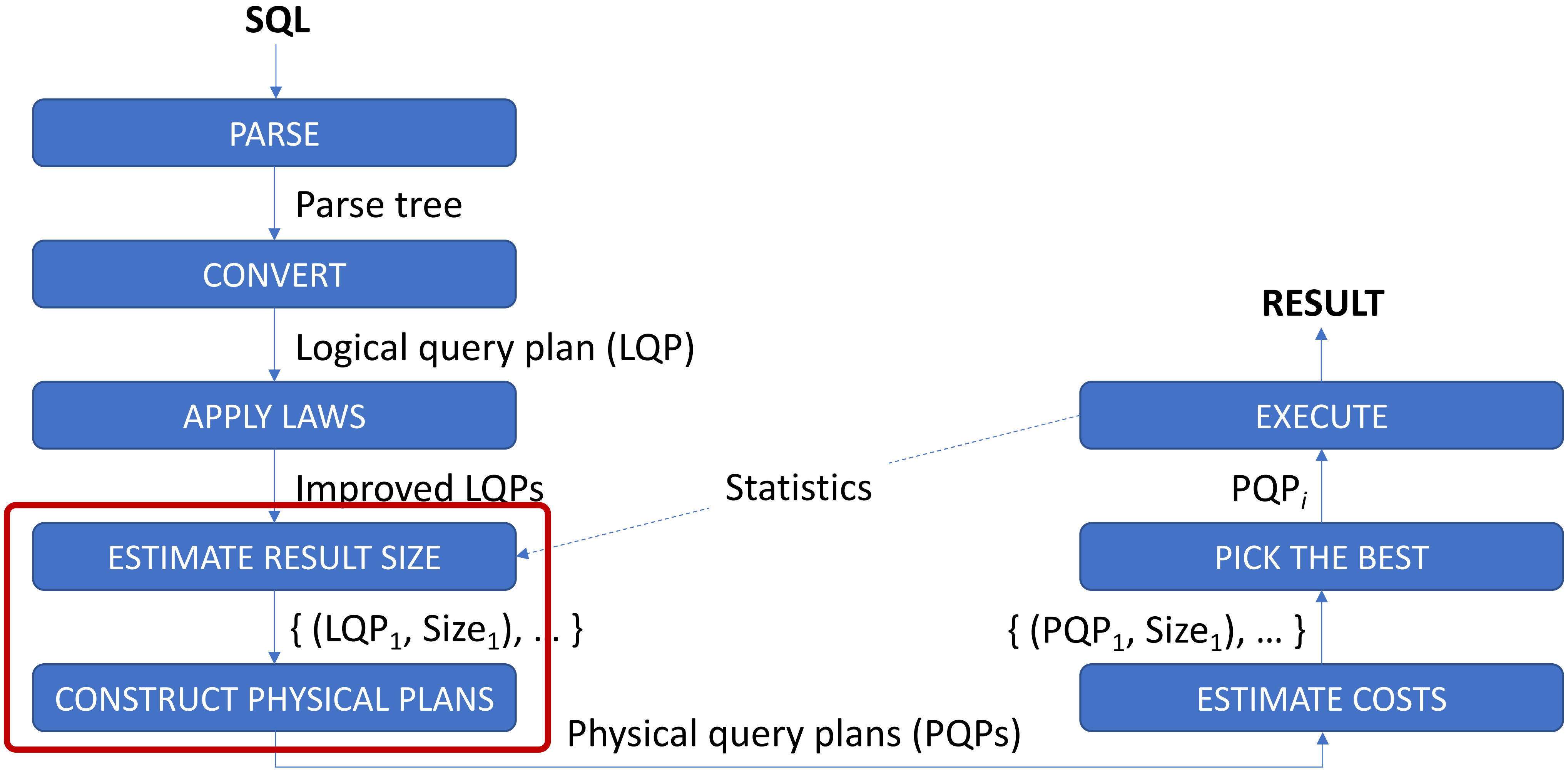
Part 1 (done in part 1):

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

Part 2 (now in part 2):

- Estimate the size/cost of the intermediary results
- (Construct physical query plans)

OVERVIEW: Our focus in part 2



OVERVIEW

- To assess both logical and physical plans, we need some way to calculate cost
- These can not be calculated exactly (depending on the data we have), so the DBMS estimates the costs
- We want a cost function C that can be calculated locally: $C(R \bowtie_c S)$ should be possible to calculate from C(R) and C(S)
- What kind of costs exist? What do we mean by cost? What should we choose as cost?

Costs: IO!

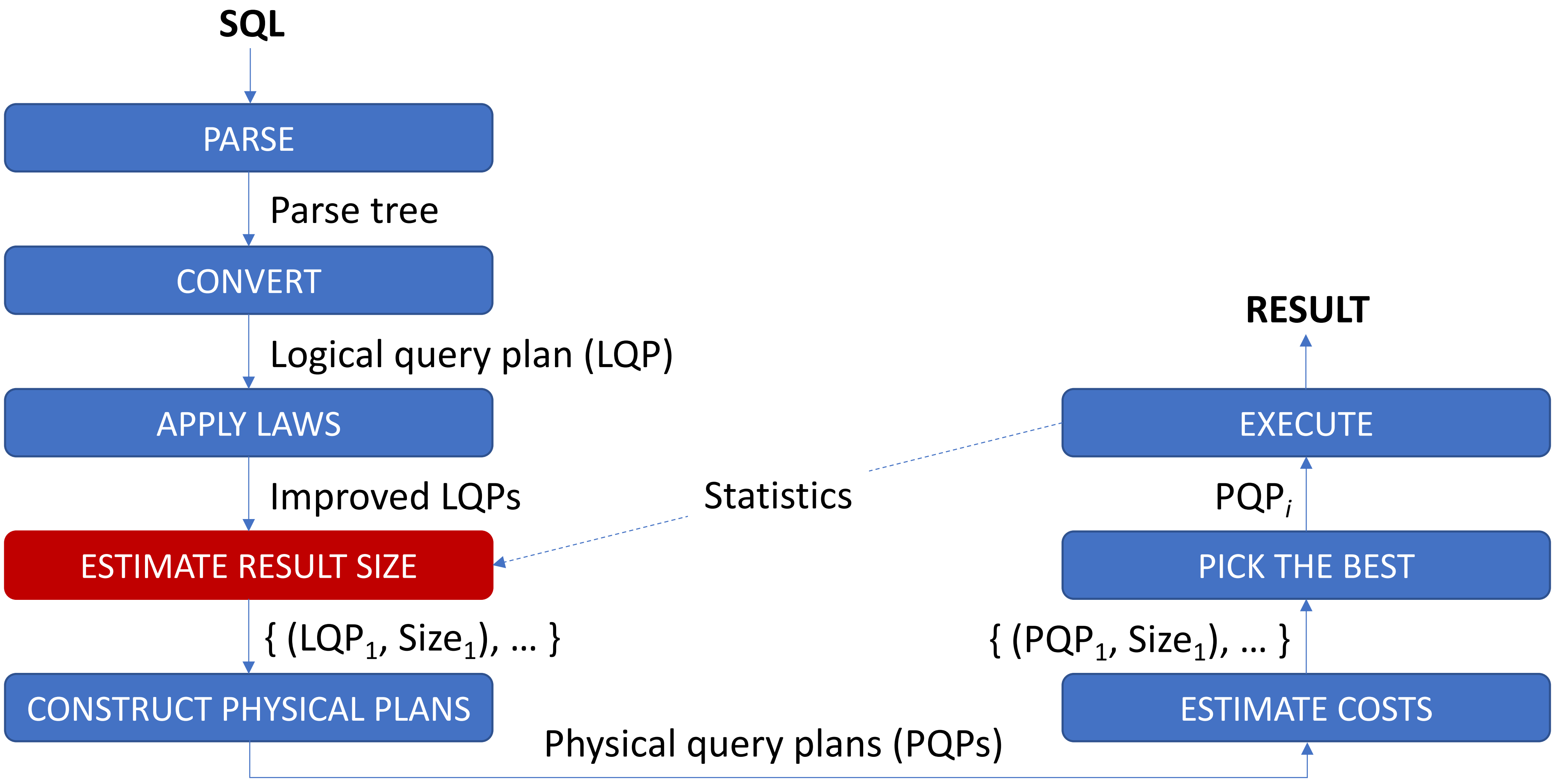
- Disk IO: Cost of reading
  - A given page (random read)
  - A sequence of pages (sequential read)
- Sequence is usually cheaper
- Relevant to choose from
  - Table scan
  - Index scan

Costs: Number of tuples (and size in bytes)

- Number of tuples to process in one operation
- Relevant for everything, including disk operations
- We will look at how this is estimated

- Size of each tuple in relation in bytes and size of the overall relation are together more fine-grained

Theme now: Estimating result size



SQL

PARSE

Parse tree

CONVERT

Logical query plan (LQP)

APPLY LAWS

Improved LQPs

ESTIMATE RESULT SIZE

{ (LQP$_1$, Size$_1$), … }

CONSTRUCT PHYSICAL PLANS

Physical query plans (PQPs)

ESTIMATE COSTS

{ (PQP$_1$, Size$_1$), … }

PICK THE BEST

PQP$_i$

EXECUTE

RESULT

Statistics

Estimating size

- Ideally, we want rules that are
    - Accurate: a small error can result in the selection of an inappropriate algorithm in the physical query plan
    - Easy to calculate: minimal extra cost to make the choice
    - Logically consistent: not dependent on a specific algorithm for the operator
- No universal algorithm exists
- Fortunately, approximate estimates also help choose a good physical query plan

Notation

- Tup_R (or Tup(R)) is the number of tuples in R
  - this is an estimate, so may be not integer
- TSize_R is the size of a tuple in R in bytes
- Size_R is the size of R in bytes: $\mathrm{Size}_R = \mathrm{Tup}_R * \mathrm{TSize}_R$

- $\mathrm{Val}_R(A)$ is the number of different values for attribute A in R
- Average number of tuples with equal A values: $\mathrm{Tup}_R / \mathrm{Val}_R(A)$

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of a projection

- The size of a projection $\pi_L(R)$ can be calculated accurately:
  - One result tuple for each argument tuple
    $Tup(\pi_{A,\ B,\ \ldots}(R)) = Tup_R$
  - Changes only the size of each tuple:
    $Size(\pi_{A,\ B,\ \ldots}(R)) = Tup_R * (TSize_{R.A} + TSize_{R.B} + \ldots)$
  - $Val(\pi_{A,\ B,\ \ldots}(R), A) = Val(R, A)$

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of projection: Example

R

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

A: 4 byte integer

B: 20 byte text string

C: 4 byte date (year)

D: 30 byte text string

$Val_R(A) = 5$

$Val_R(B) = 2$

$Val_R(C) = 4$

$Val_R(D) = 3$

$Tup_R = 5$

$TSize_R = 58$

$Size(\pi_{A, B, \ldots}(R)) = Tup_R * (TSize_A + TSize_B + \ldots)$

$Size_R = \ldots$

$Size(\pi_A(R)) = \ldots$

$Size(\pi_{A, B, C, D, (A+10) \rightarrow E}(R)) = \ldots$

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of a selection

- A selection $\sigma_{Cond}(R)$ reduces the number of tuples, but the size of each tuple remains unchanged
  - $Size(\sigma_{Cond}(R)) = Tup(\sigma_{Cond}(R)) * TSize_R$
- Estimating the number of tuples depends upon
  - the selection condition Cond
  - distribution of values for the relevant attributes:
    - we assume a uniform distribution where we use $Val_R(A)$ to estimate the number of tuples in the result (naive)
    - DBMSs use more advanced statistics (e.g., histograms)
- Estimation of the number of values Val can be done similarly

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of a selection (cont.)

For attribute A and constant *c*:

- Similarity, $\sigma_{A = c}(R)$:
  Use the selectivity factor $1/Val_R(A)$
  $Tup(\sigma_{A = c}(R)) = Tup_R / Val_R(A)$
- Inequality, $\sigma_{A \neq c}(R)$:
  Use the selectivity factor $1 - 1/Val_R(A)$
  $Tup(\sigma_{A \neq c}(R)) = Tup_R * (1 - 1/Val_R(A))$
- Interval, $\sigma_{A < c}(R)$: ...
- Equality of two attributes, $\sigma_{A = B}(R)$: ...

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of a selection: Example

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

A: 4 byte integer	$Val_R(A) = 5$
B: 20 byte text string	$Val_R(B) = 2$
C: 4 byte date (year)	$Val_R(C) = 4$
D: 30 byte text string	$Val_R(D) = 3$

$Tup_R = 5$

$Size_R = 58$

$Tup(\sigma_{A = c}(R)) = Tup_R \,/\, Val_R(A)$

$Tup(\sigma_{A = 3}(R)) = \ldots$

$Tup(\sigma_{B = \text{'cat'}}(R)) = \ldots$

$Tup(\sigma_{A \neq c}(R)) = Tup_R * (1 - 1/Val_R(A))$

$Tup(\sigma_{B \neq \text{'cat'}}(R)) = \ldots$

Size of a selection with AND and NOT

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

- Multi-condition selection with AND, $\sigma_{\text{Cond1 AND Cond2 AND …}}(R)$:
  - estimate the size using one selectivity factor for each condition:
    - $1 – 1 / Val_R (A)$ for ≠ on attribute A
    - $1 / Val_R (A)$ for = on attribute A
    - …
  - $Tup (\sigma_{\text{Cond1 AND Cond2 AND …}} (R)) = Tup_R * factor_{Cond1} * factor_{Cond2} * …$

- Selection with NOT, such as $\sigma_{\text{NOT Cond}} (R)$:
  - Use the selectivity factor $1 – Tup(\sigma_{Cond} (R)) / Tup_R$
  - $Tup (\sigma_{\text{NOT Cond}} (R)) = Tup_R - Tup (\sigma_{\text{NOT Cond}} (R))$

Selection with AND and NOT: Example

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

A: 4 byte integer
B: 20 byte text string
C: 4 byte date (year)
D: 30 byte text string

$Tup_R = 5$
$Size_R = 58$

$Val_R(A) = 5$
$Val_R(B) = 2$
$Val_R(C) = 4$
$Val_R(D) = 3$

$Tup(\sigma_{\text{Cond1 AND Cond2 AND ...}}(R)) = Tup_R * factor_{\text{Cond1}} * factor_{\text{Cond2}} * \ldots$

$Tup(\sigma_{C = 1999 \text{ AND } B \neq ´cat´}(R)) = \ldots$

$Tup(\sigma_{\text{NOT Cond}}(R)) = Tup_R - Tup(\sigma_{\text{Cond}}(R))$

$Tup(\sigma_{\text{NOT } A = 3}(R)) = \ldots$

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of a selection with OR

- Multiple condition selection with OR, $\sigma_{\text{Cond1 OR Cond2 OR …}}(R)$
  - Option 1:
    $Tup(\sigma_{\text{Cond1 OR Cond2 OR …}}(R)) = Tup(\sigma_{\text{Cond1}}(R)) + Tup(\sigma_{\text{Cond2}}(R)) + \ldots$
  - Option 2:
    $Tup(\sigma_{\text{Cond1 OR Cond2 OR …}}(R)) = \min(Tup_R, (Tup(\sigma_{\text{Cond1}}(R)) + Tup(\sigma_{\text{Cond2}}(R)) + \ldots))$
  - Option 3:
    - Assume that m_1 tuples satisfy the first condition, m_2 satisfy the second condition, …
    - $1 - m_i / Tup_R$ is the proportion of tuples that do not satisfy the i-th condition
    - $Tup(\sigma_{\text{A OR B OR …}}(R)) = Tup_R * [1 – (1 – m_1 / Tup_R) * (1 – m_2 / Tup_R) * \ldots]$

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of a PRODUCT

- The size of a Cartesian product R × S can be calculated accurately:
  - One tuple for each possible combination of the tuples in relations R and S: $Tup(R \times S) = Tup_R * Tup_S$
  - The size of each new tuple is the sum of the size of each of the original tuples: $TSize(R \times S) = TSize_R + TSize_S$
  - $Size(R \times S) = Tup(R \times S) * TSize(R \times S) = Tup_R * Tup_S * (TSize_R + TSize_S)$

Size of a NATURAL JOIN

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

- The size of natural join $R (X, A) \bowtie S (A, Y)$ depends on how the values of the join attribute A is distributed between relations R (X, A) and S (A, Y):
- Disjoint set of A values - empty result: $Tup (R \bowtie S) = 0$
- A is a foreign key from R to S -- each tuple in R matches one tuple in S: $Tup (R \bowtie S) = Tup_R$
- Almost all the R and S tuples have the same A value -- combine all the tuples in each relation:
  $Tup (R \bowtie S) = Tup_R * Tup_S$

Size of a NATURAL JOIN (cont.)

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Assumptions:

- **Inclusion of values:** If $Val_R(A) \leq Val_S(A)$, then assume that $\delta(\pi_A(R)) \subseteq \delta(\pi_A(S))$, i.e., each A value in R has a match in S
- **Value conservation:** Assume that the value of non-join attributes is the same before and after join, i.e., $Val(R \bowtie S, B) = Val_R(B)$, where B is an attribute in R, but not in S

Size of a NATURAL JOIN (cont.)

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

The number of tuples in R (X, A) ⋈ S (A, Y) can now be estimated as follows:

- If $Val_R(A) \leq Val_S(A)$, each tuple in R will match approximately $Tup_S/Val_S(A)$ tuples in S: $Tup(R \bowtie S) = Tup_R * Tup_S/Val_S(A)$
- Similarly, if $Val_S(A) \leq Val_R(A)$: $Tup(R \bowtie S) = Tup_R * Tup_S/Val_R(A)$
- General: $Tup(R \bowtie S) = Tup_R * Tup_S / \max(Val_R(A), Val_S(A))$
- $Val(R \bowtie S, B) = Val_R(B)$ for B an attribute in X
- $Val(R \bowtie S, C) = Val_S(C)$ for C an attribute in Y
- $Val(R \bowtie S, A) = \min(Val_R(A), Val_S(A))$ for the join-attribute A

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of a NATURAL JOIN: Example

A(a, b)	**B(b, c)**	**C(c, d)**
T_A = 10.000	T_B = 2.000	T_C = 5.000
$V_A(a)$ = 5.000	$V_B(b)$ = 100	$V_C(c)$ = 100
$V_A(b)$ = 1.000	$V_B(c)$ = 1.000	$V_C(d)$ = 100

$Tup(A \bowtie B) = \ldots$

$Tup(A \bowtie B \bowtie C) = \ldots$

Size of a NATURAL JOIN (cont.)

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

- If there is more than one join attribute, $R\ (X, A_1, A_2, \ldots) \bowtie S\ (A_1, A_2, \ldots, Y)$, we get:

  $$Tup\ (R \bowtie S) = \frac{(Tup_R * Tup_S)}{(\max\ (Val_R(A_1), Val_S(A_1)) * \max\ (Val_R\ (A_2), Val_S(A_2)) * \ldots}$$

  for each A_i attribute common to R and S
- For natural join between multiple relationships $R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n$
  - Start with maximum number of tuples $Tup(R_1) * Tup\ (R_2) * Tup\ (R_3) * \ldots * Tup\ (R_n)$
  - for each attribute A that occurs in more than one relationship, divide by all except the smallest $Val(R_i, A)$

Size of a EQUI-JOIN and THETA-JOIN

- The size of the equi-join is calculated as a natural join
- Calculate the size of theta-join $R \bowtie_{\text{Cond}} S$ by calculating the size of $\sigma_{\text{Cond}}(R \bowtie S)$

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of a UNION

Depends on whether we use the set or bag version:

- **Bag:**
  The result is exactly equal to the sum of tuples in the arguments:
  $Tup(R \cup_b S) = Tup_R + Tup_S$
- **Set:**
  - As for bags if relationships are disjointed
  - Number of tuples in the largest relation if the smallest is a subset of it
  - Usually somewhere between these. We can use the average, for example: $Tup(R \cup_s S) = (Tup_R + Tup_S) / 2$, where S is the smallest relationship

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of INTERSECTION

- Number of tuples in an intersection R∩S is
  - 0 if the relationships are disjointed
  - $\min(Tup_R, Tup_S)$ if one relationship is a subset of the other
  - Usually somewhere in between, for example, can use the average:
    $\min(Tup_R, Tup_S) / 2$

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of DIFFERENCE

- Number of tuples in a difference R – S is
  - Tup_R if the relationships are disjointed
  - Tup_R -Tup_S if all tuples in S is also in in R
  - Usually somewhere in between. We can use the average, for example: (Tup_R - Tup_S) / 2

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of DUPLICATE ELIMINATION

- The number of distinct tuples as a result of a duplicate elimination δ(R) is
  - 1 if all the tuples are the same
  - Tup_R if all the tuples are different
- An approach:
- Given $Val_R(A_i)$ for all *n* attributes, the maximum number of different tuples will be $Val_R(A_1) * Val_R(A_2) * \ldots * Val_R(A_n)$
- Let the estimated number of tuples be the least of this number and $Tup_R/2$, i.e., $\min (Val_R (A_1) * Val_R (A_2) * \ldots * Val_R (A_n), Tup_R/2)$

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

Size of GROUPING

- Grouping is similar to duplicate elimination:

  Looks only at the grouping attributes $A_1, A_2, \ldots A_m$

Comparing Logical Query Plans

- We compare different query plans for a given query using the **size of temporary relations**
  - Estimate the result of each operator in the questionnaire
  - Add costs to the tree
  - The cost of the plan is equal to the sum of all the costs in the tree, except for
    - the root – which is for the end result
    - the leaf nodes – data stored on disk

Comparing Logical Query Plans: Example

- Tup_R is the number of tuples in R
- $TSize_R$ is the size of a tuple in R in bytes
- $Val_R(A)$ is the number of different values for attribute A in R
- $Size_R$ is the size of R in bytes

```
StarsIn(title, year, starName)
MovieStar(name, address, gender, birthDate)
```

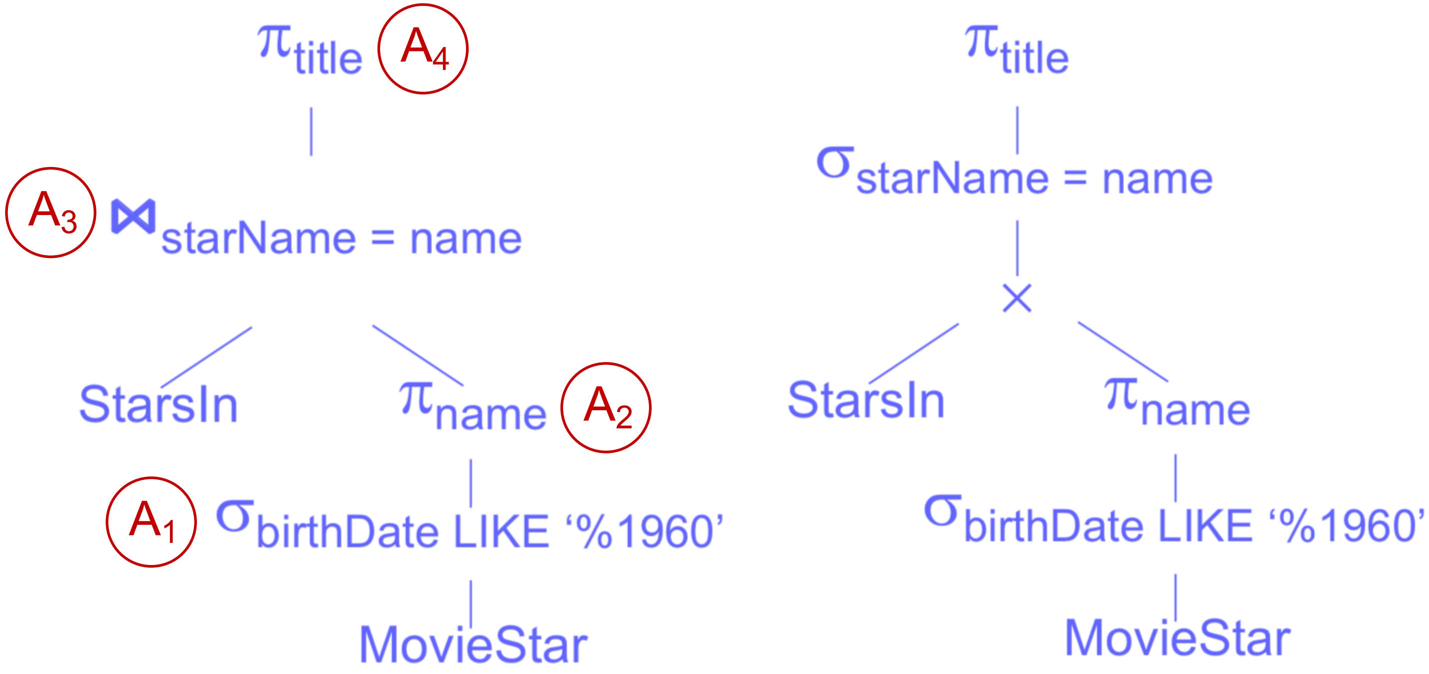
```
SELECT title FROM StarsIn
WHERE starName IN (
  SELECT name FROM MovieStar
  WHERE birthDate LIKE '%1960');
```

Statistics:

Tup(StarsIn) = 10,000

Val(StarsIn, starName) = 500
TSize(StarsIn) = 80

Tup(MovieStar) = 1,000
Val(MovieStar, name) = 1,000
Val(MovieStar, birthDate) = 50
TSize(MovieStar) = 100



π_{title} ④ A_4

A_3 $\bowtie_{starName = name}$

StarsIn

π_{name} A_2

A_1 $\sigma_{birthDate\ LIKE\ '\%1960'}$

MovieStar

π_{title}

$\sigma_{starName = name}$

$\times$

StarsIn

π_{name}

$\sigma_{birthDate\ LIKE\ '\%1960'}$

MovieStar

Comparing Logical Query Plans - Example

A_1 = $\sigma_{\text{birthDate LIKE '\%1960'}}$ (MS):
- $\text{Tup}(A_1) = \text{Tup}(\sigma(\text{MS})) = \text{Tup}(\text{MS}) / \text{Val}(\text{MS, birthDate}) = 1000 / 50 = 20$
- $\text{Size}(A_1) = 20 * 100 = 2000$

A_2 = $\pi_{\text{name}}(A_1)$:
- $\text{Tup}(A_2) = \text{Tup}(\pi(A_1)) = \text{Tup}(A_1) = 20$
- Assume that name is 20 bytes
- $\text{TSize}(A_2) = 20$
- $\text{Size}(A_2) = 20 * 20 = 400$

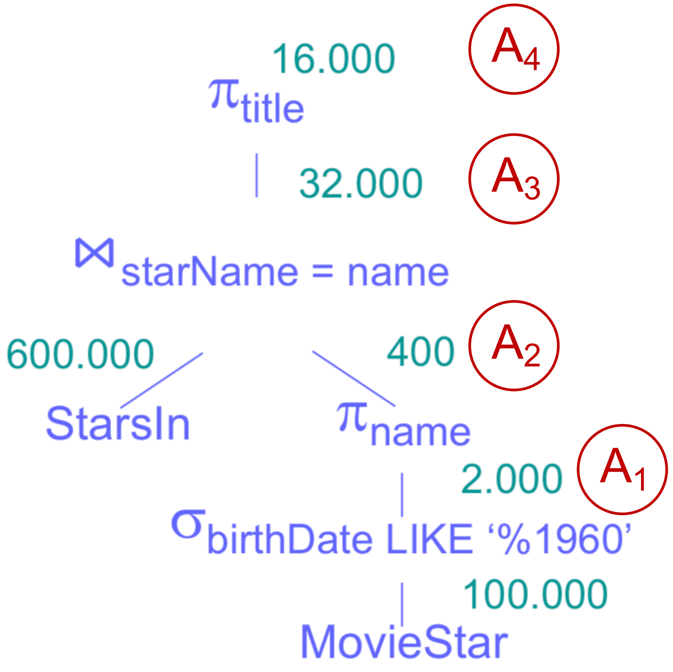
A_3 = SI $\bowtie$ A_2:
- $\text{Tup}(A_3) = \text{Tup}(\text{SI} \bowtie A_2) =$
- $\text{Tup}(\text{SI})*\text{Tup}(A_2) / \max[\text{Val}(\text{SI, starName}), \text{Val}(A_2\text{, name})] =$ $10{,}000 * 20 / \max(500, \underline{20}) = 400$
- $\text{Size}(A_3) = 400 * (80 + 20 - 20) = 32{,}000$

A_4 = $\pi_{title}(A_3)$:
- $\text{Tup}(A_4) = \text{Tup}(\pi(A_3)) = \text{Tup}(A_3) = 400$
- Assume that title is 40 bytes
- $\text{Size}(A_4) = 400 * 40 = 16{,}000$

Statistics:
Tup(StarsIn) = 10,000

Val(StarsIn, starName) = 500
TSize(StarsIn) = 80

Tup(MovieStar) = 1,000
Val(MovieStar, name) = 1,000
Val(MovieStar, birthDate) = 50
TSize(MovieStar) = 100



Comparing Logical Query Plans - Example

Statistics:
Tup(StarsIn) = 10,000

Val(StarsIn, starName) = 500
TSize(StarsIn) = 80

Tup(MovieStar) = 1,000
Val(MovieStar, name) = 1,000
Val(MovieStar, birthDate) = 50
TSize(MovieStar) = 100

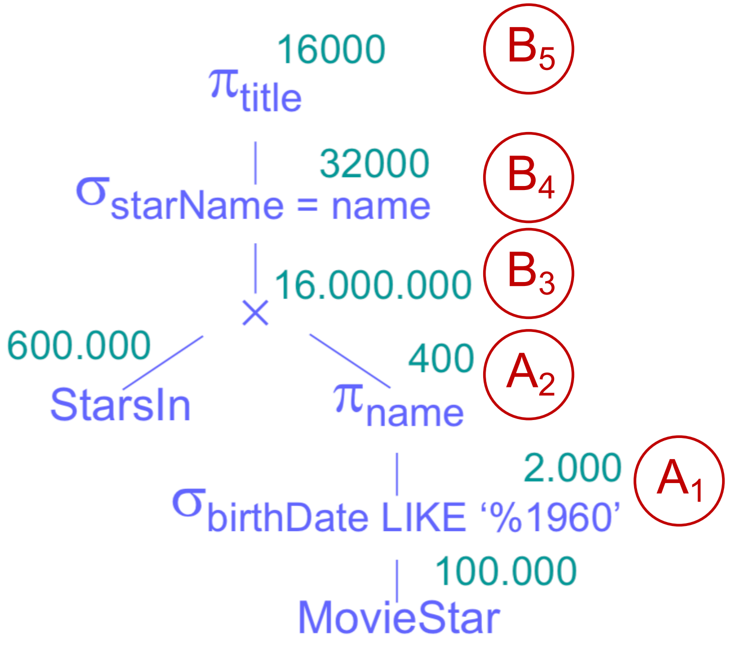
A_1 = $\sigma_{\text{birthDate LIKE '\%1960'}}(\text{MS})$ → as before: 2000, $\text{Tup}(\sigma(\text{MS}))=20$

A_2 = $\pi_{\text{name}}(A_1)$ → as before: 400, $\text{Tup}(A_2) = \text{Tup}(A_1) = 20$

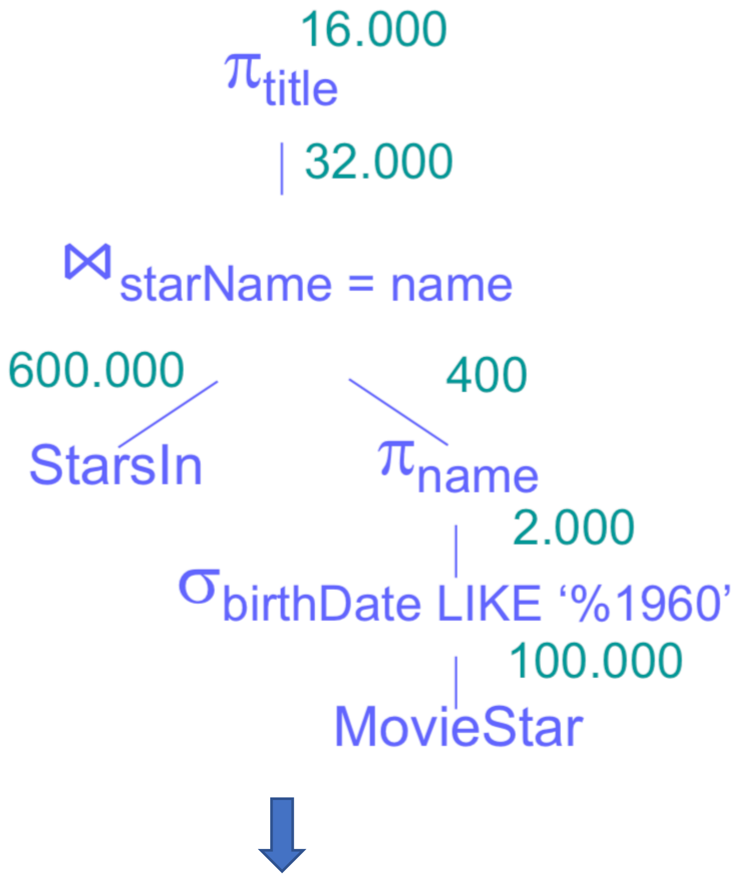
B_3 = $\text{SI} \times A_2$:
- $\text{Tup}(B_3) = \text{Tup}(\text{SI} \times A_2) = \text{Tup}(\text{SI}) * \text{Tup}(A_2) = 10000 * 20 = 200{,}000$
- $\text{TSize}(A_2) = 20$
- $\text{Size}(B_3) = 200{,}000 * (80 + 20) = 20{,}000{,}000$

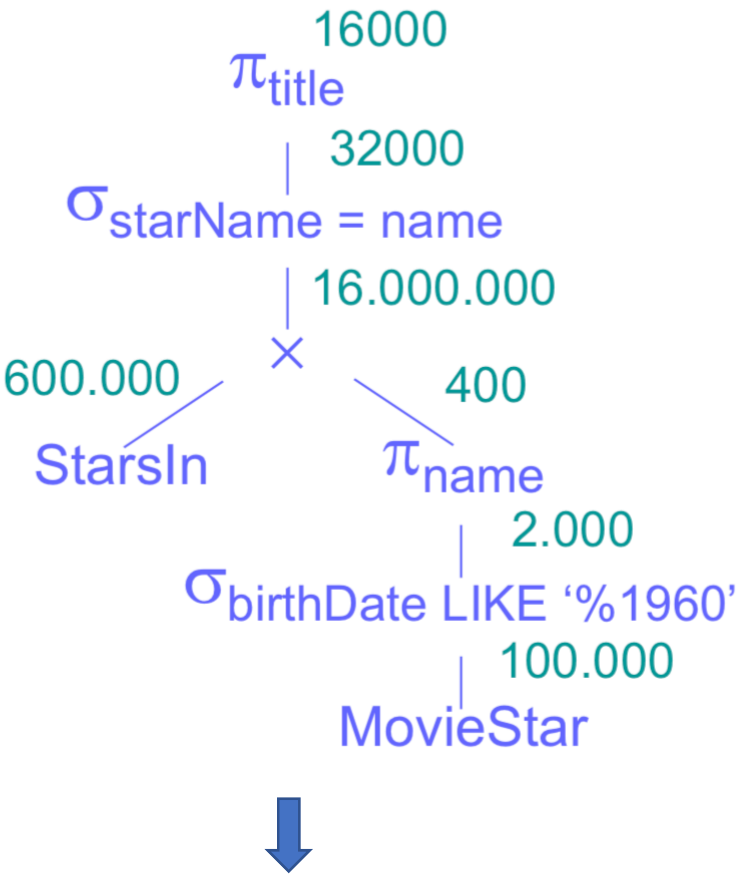
B_4 = $\sigma_{\text{starName = name}}(B_3)$:
- $\text{Tup}(B_4) = \text{Tup}(\sigma(B_3)) =$
  $\text{Tup}(B_3) / \max(\text{Val}(B_3, \text{name}), \text{Val}(\text{SI}, \text{starName})) =$
  $200{,}000 / \max(20, 500) = 400$
- $\text{TSize}(B_4) = \text{TSize}(\text{SI}) + \text{TSize}(B_3) = 80 + 20 = 100$
- $\text{Size}(B_4) = 400 * 100 = 40{,}000$

B_5 = $\pi_{\text{title}}(B_4)$ → as A_4: $400 * 40 = 16{,}000$



Comparing Logical Query Plans - Example





Total: 2000 + 400 + 32,000 = 34,400

Total: 2000 + 400 + 20,000,000 + 40,000 = 20,042,400