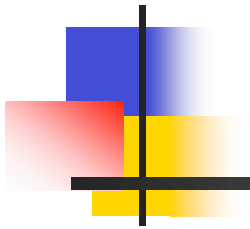
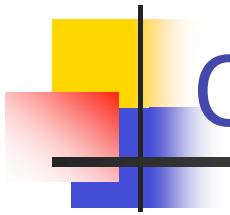


Query Compilation

Contains slides by
Hector Garcia-Molina



Overview

- ✓ Query processors
- ✓ Parsing
- ✓ Converting to logical query plans in relational algebra
- ✓ Query rewrite
- ✓ Estimate size of a intermediate relation
- ✓ Consider physical query plans

Example – I

- 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

Answer

B	C	Y
r	8	k

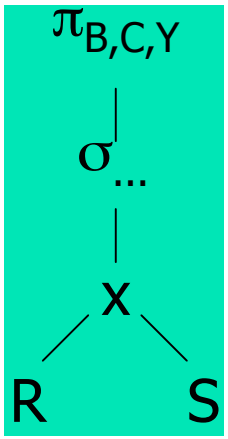
But, how is the query executed?

Example – II

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

✓ Example: **idea 1** – cartesian product, select tuples, project attributes

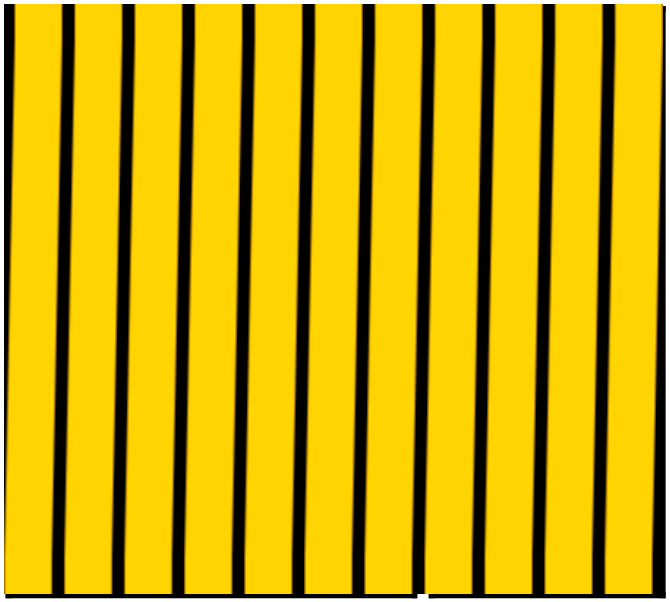
⇨ $\pi_{B,C,Y}(\sigma_{W=X \wedge A=3 \wedge Z="a"}(R \times S))$



NOTE:
#attributes = #R-attributes + #S-attributes
#tuples = #R-tuples * #S-tuples

Relation R

Relation 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	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
...	...	...	...	...	...	...	...

Answer

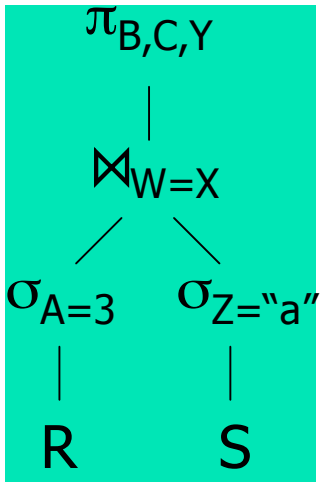
B	C	Y
r	8	k

Example – III

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

✓ Example: **idea 2** –select tuples, equijoin, project attributes

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



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

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

B	C	Y
r	8	k

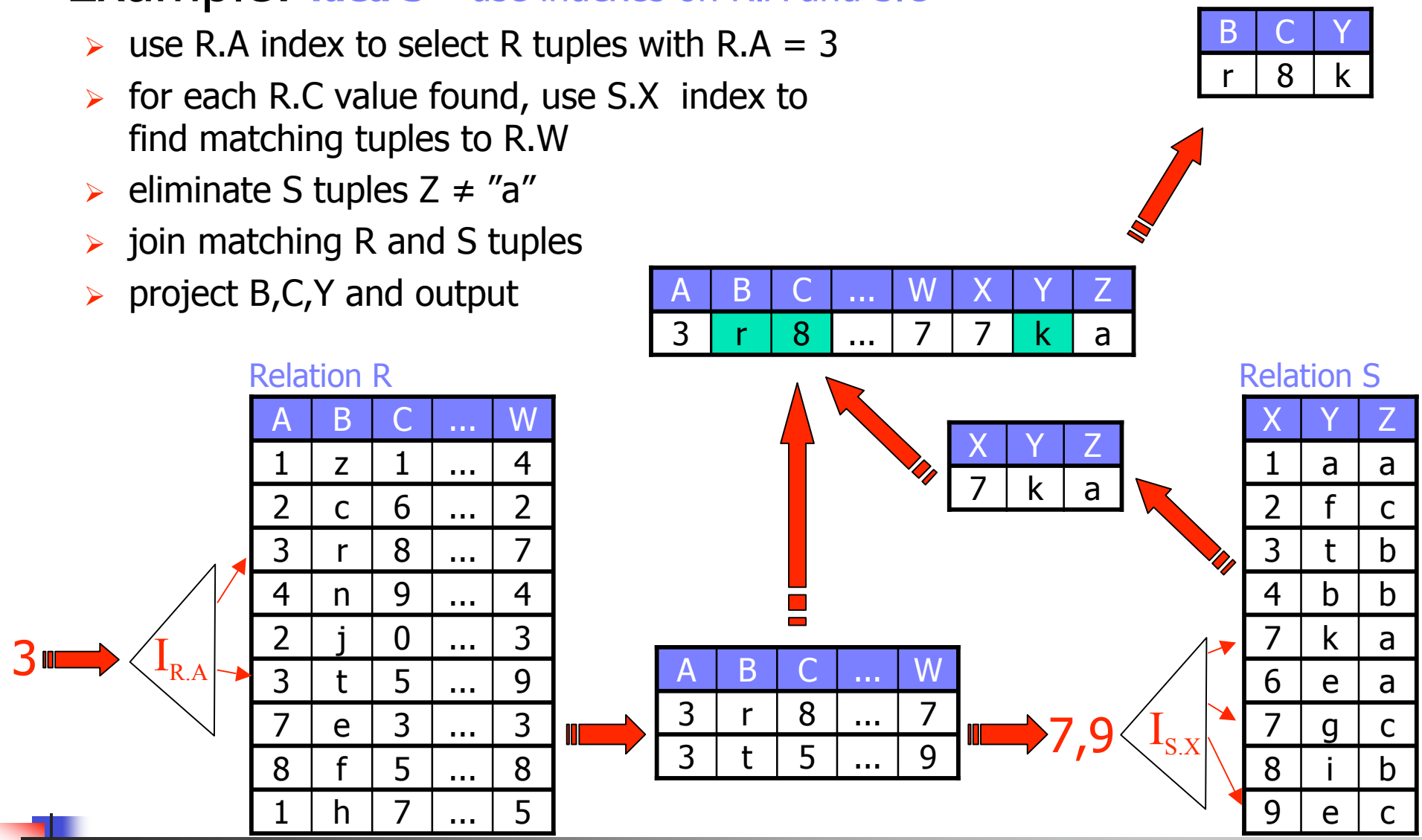
Relation S

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

Example – IV

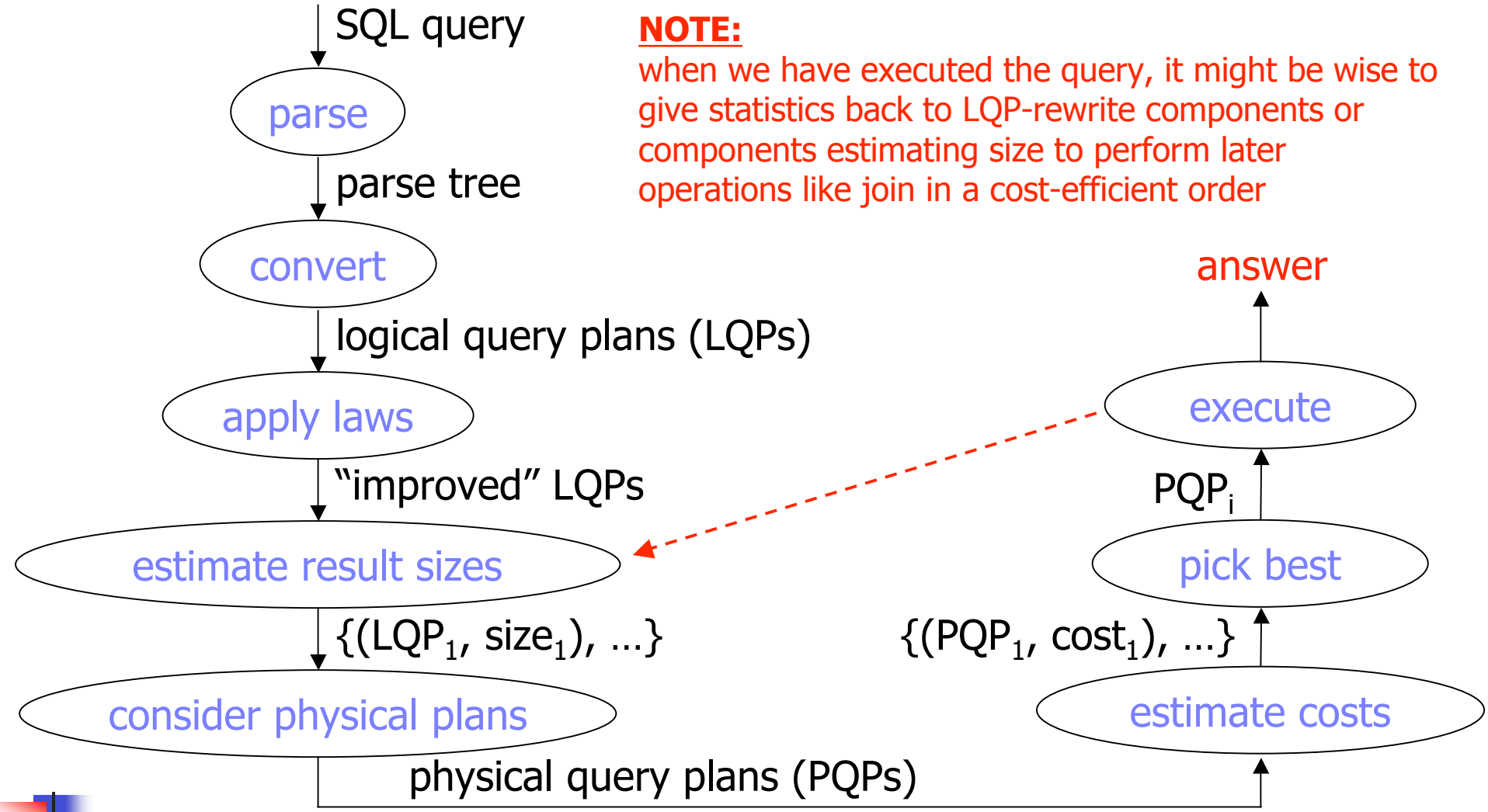
```
SELECT B,C,Y
FROM R,S
WHERE W=X AND A=3 AND Z="a"
```

- ✓ Example: **idea 3** – use indexes on R.A and S.C
  - use R.A index to select R tuples with R.A = 3
  - for each R.C value found, use S.X index to find matching tuples to R.W
  - eliminate S tuples Z ≠ ”a”
  - join matching R and S tuples
  - project B,C,Y and output



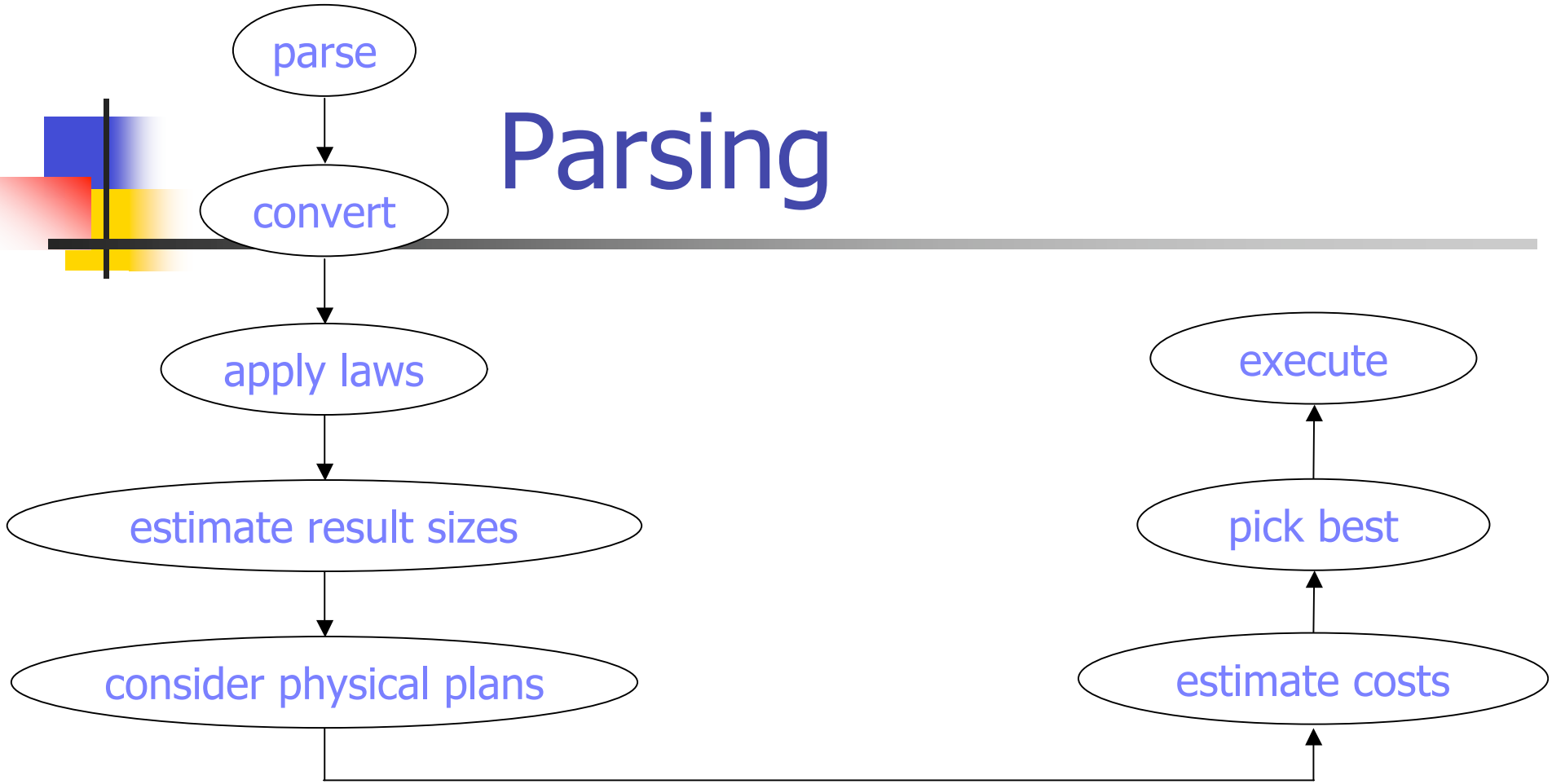
Query Processors

- A query processor must find a plan how to execute the query



NOTE:
when we have executed the query, it might be wise to give statistics back to LQP-rewrite components or components estimating size to perform later operations like join in a cost-efficient order

Parsing


parse
convert
apply laws
estimate result sizes
consider physical plans
execute
pick best
estimate costs

Parsing

- The job of the parser is to take a query written in a language like SQL and convert it to a parse tree

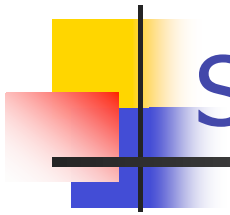
- In a parse tree, each node is either
  - *atoms* – lexical elements such as keywords, names, constants, parentheses, and operators (cannot have children)
  - *syntactic* categories – names of query sub-parts (represented by triangular brackets around descriptor)

Simple Grammar – I

- Queries:
  - <Query> ::= <SFW>
  - <Query> ::= **(** <Query> **)**
  - a complete grammar will also consist operations such as UNION, JOIN, …
  - the second rule is typically used in sub-queries
- Select-From-Where:
  - <SFW> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>
  - a complete grammar must include GROUP BY, HAVING, ORDER BY, …
- Select list:
  - <SelList> ::= <Attribute>
  - <SelList> ::= <Attribute>, <SelList>
  - a complete grammar must include expressions and aggregate functions
- From list:
  - <FromList> ::= <Relation>
  - <FromList> ::= <Relation>, <FromList>
  - a complete grammar must include aliasing and expressions like R JOIN S

Simple Grammar – II

- Conditions:
  - <Condition> ::= <Condition> AND <Condition>
  - <Condition> ::= <Tuple> IN <Query>
  - <Condition> ::= <Attribute> = <Attribute>
  - <Condition> ::= <Attribute> LIKE <Pattern>
  - a complete grammar must include operators like OR, NOT, etc. and all other comparison operators
- Tuple:
  - <Tuple> ::= <Attribute>
  - a complete grammar must include tuples of several attributes, …
- Basic syntactic categories like <Relation>, <Attribute>, <Pattern>, etc. does not have a rule, but are replaced by a name or a quoted string



Simple Grammar: Example

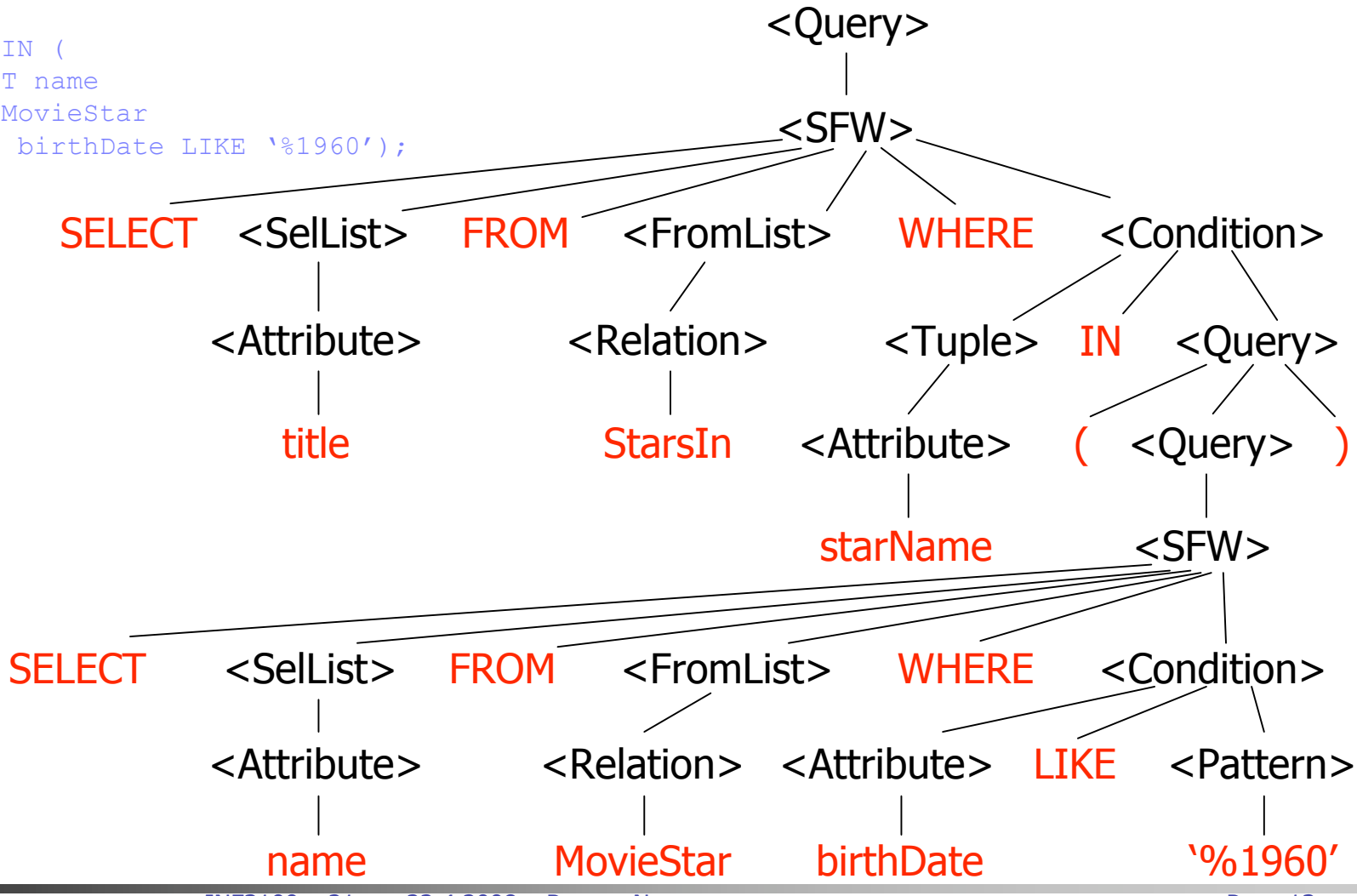
Example: Find the movies with stars born in 1960

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

Simple Grammar: Example

Example: Find the movies with stars born in 1960

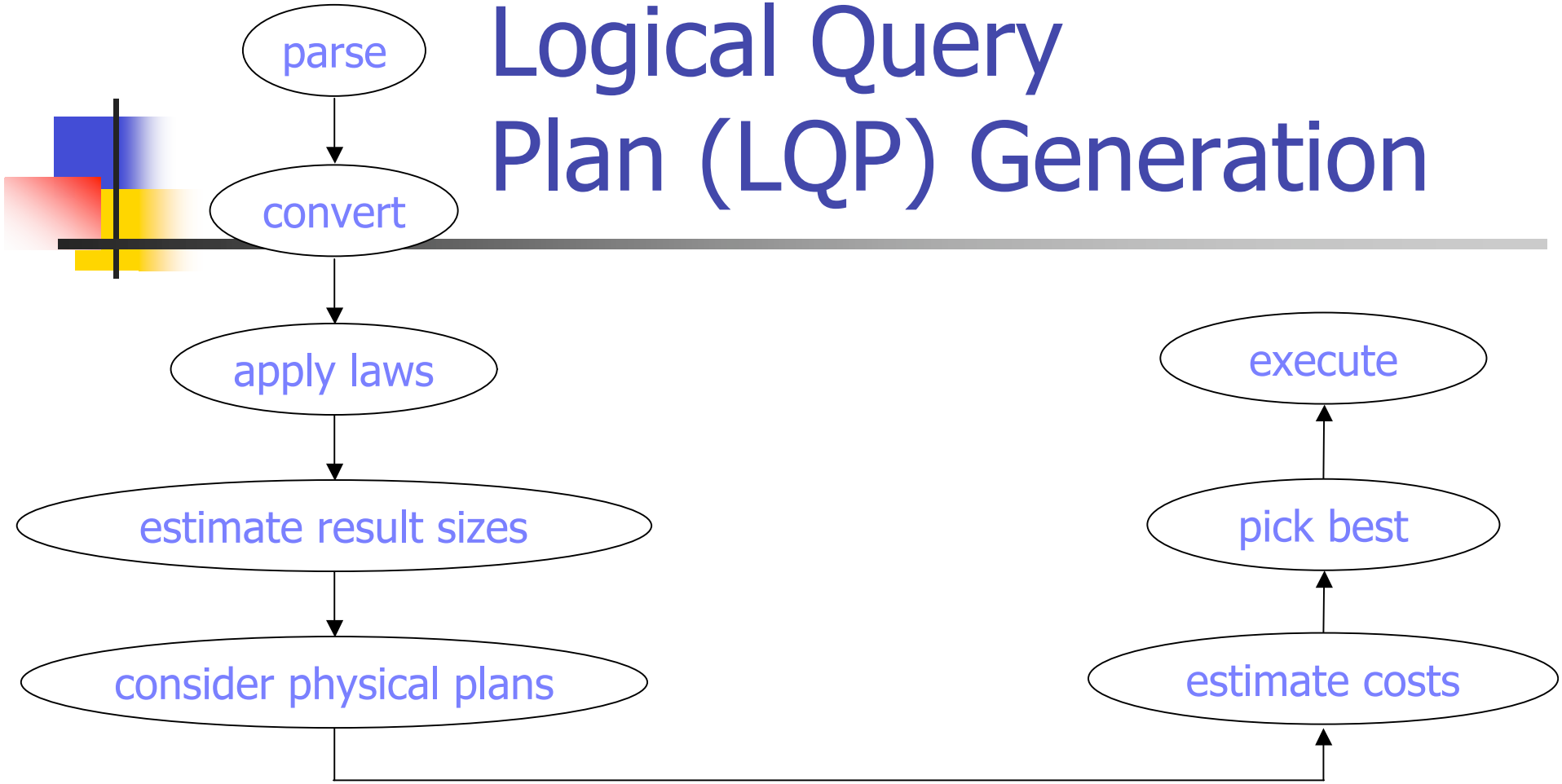
```
SELECT title
FROM StarsIn
WHERE starName IN (
          SELECT name
          FROM MovieStar
          WHERE birthDate LIKE '%1960');
```



Preprocessor

- ✓ The preprocessor checks whether the query is semantically correct, e.g.:
  - *relations* – every relation in FROM must be a relation or view in the schema on which the query is executed. If it is a view it must again be replaced by a new (sub-)parse tree.
  - *attributes* – every attribute must be part of one of the relations in the current scope of the query
  - *types* – all attributes must be of a type appropriate to their uses
- ✓ If the query (parse tree) passes the tests from the preprocessor, is is said to be *valid*
- ⇨ send to logical query plan (LQP) generator

Logical Query Plan (LQP) Generation



Conversion into Relational Algebra – I

- When the query is expressed as a valid parse tree, we can generate a LQP expressed by relational algebra operators

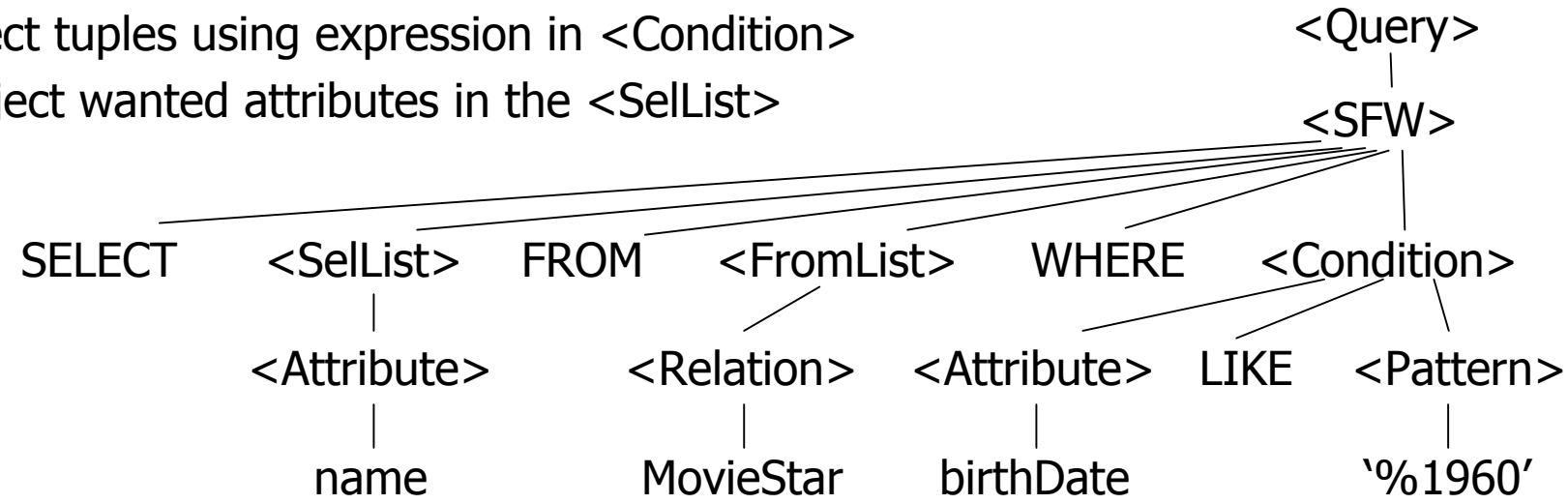
- SFW without sub-queries:
  - replace the relations in the <FromList> by the product, x, of all relations
  - this product is the argument of a selection, σ_C, where C is the <Condition> expression being replaced
  - this selection is in turn the argument of a projection, π_L, where L is the list of attributes in the <SelList>

Conversion into Relational Algebra – II

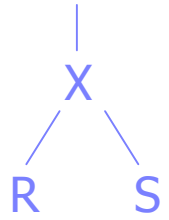
- Example:

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

  - product of relations in <FromList>
  - select tuples using expression in <Condition>
  - project wanted attributes in the <SelList>

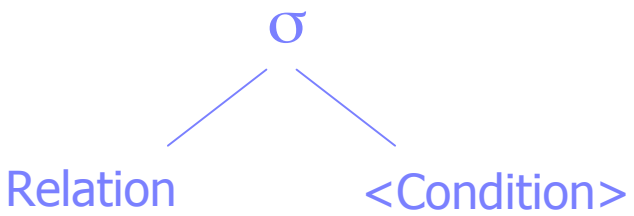


NOTE:
we have only one relation. If we would have two, the lower part of the tree would look something like:



Conversion into Relational Algebra – III

- If we have sub-queries, we must remove them by using an intermediate operator – two argument select σ :



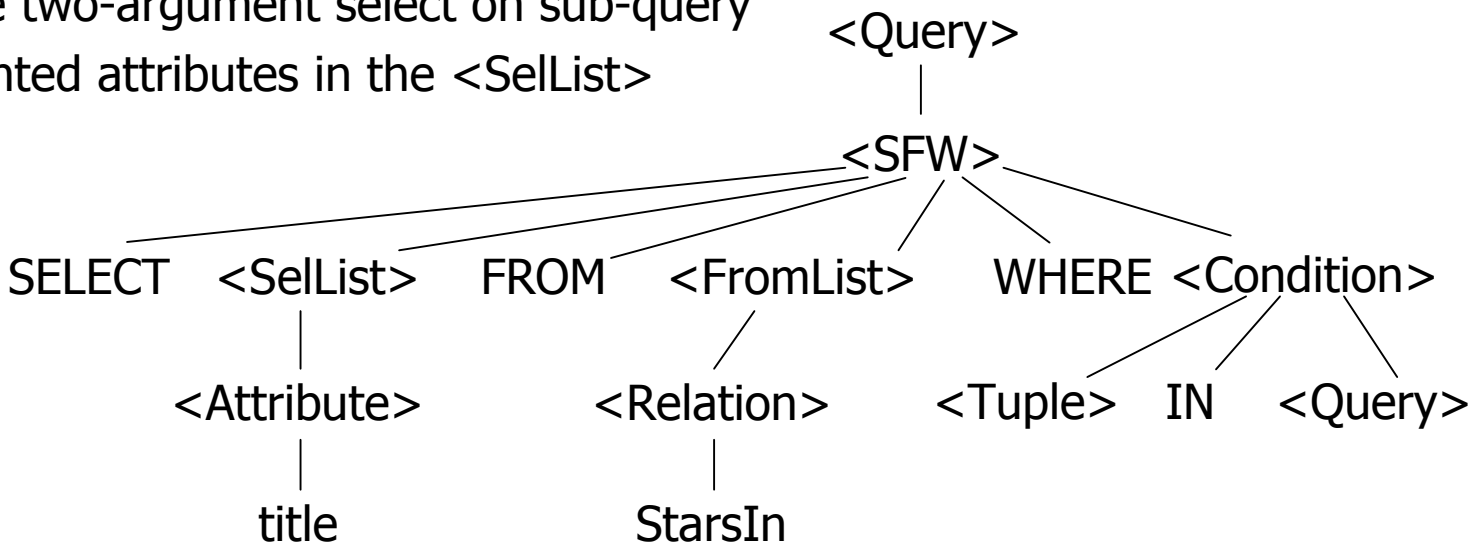
  - left child represent relation upon which the selection is performed
  - right child is an expression for the condition applied to each tuple of the relation

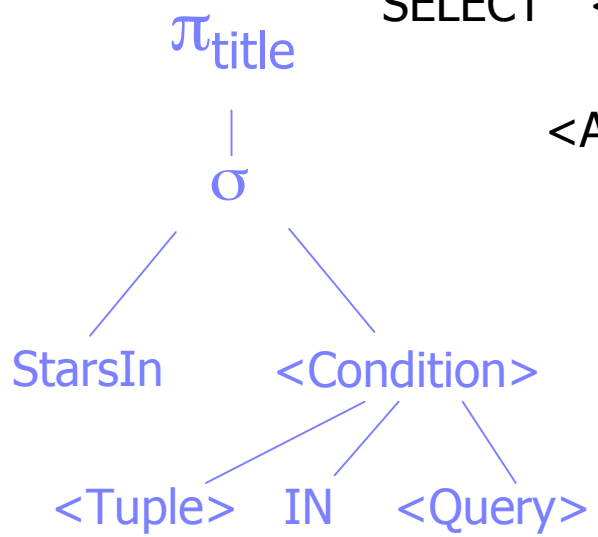
Conversion into Relational Algebra – IV

- Example:

```
SELECT title FROM StarsIn WHERE starName IN (<Query>)
```

  - product of relations in <FromList>
  - select tuples using expression in <Condition>, but use the two-argument select on sub-query
  - project wanted attributes in the <SelList>





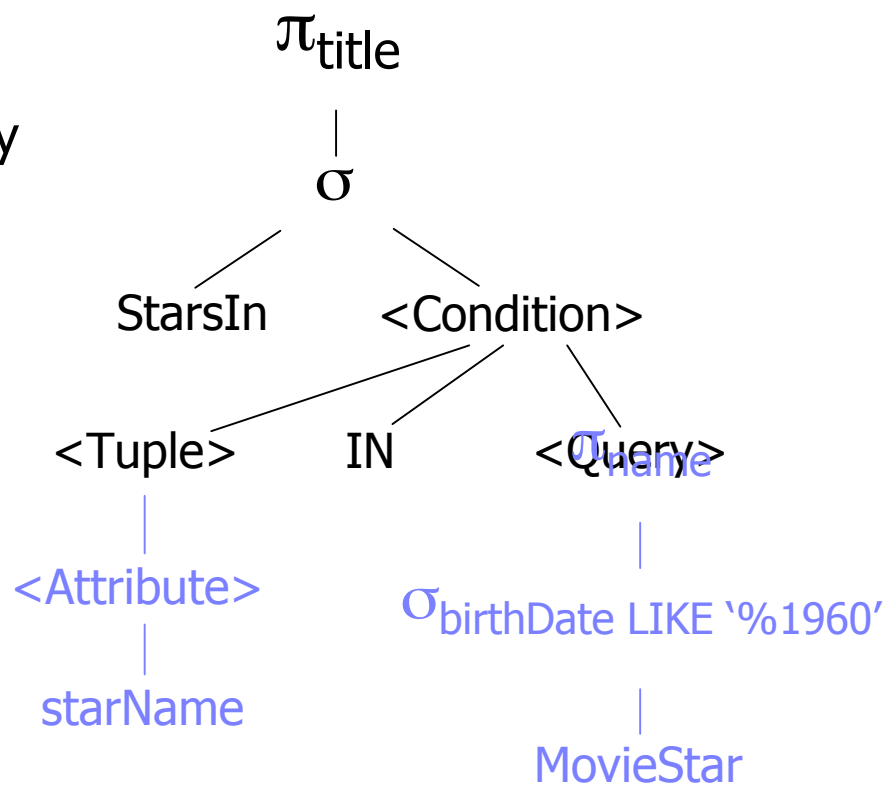
Conversion into Relational Algebra – IV

- Example (cont.):

```
SELECT title FROM StarsIn WHERE starName IN (<Query>)
```

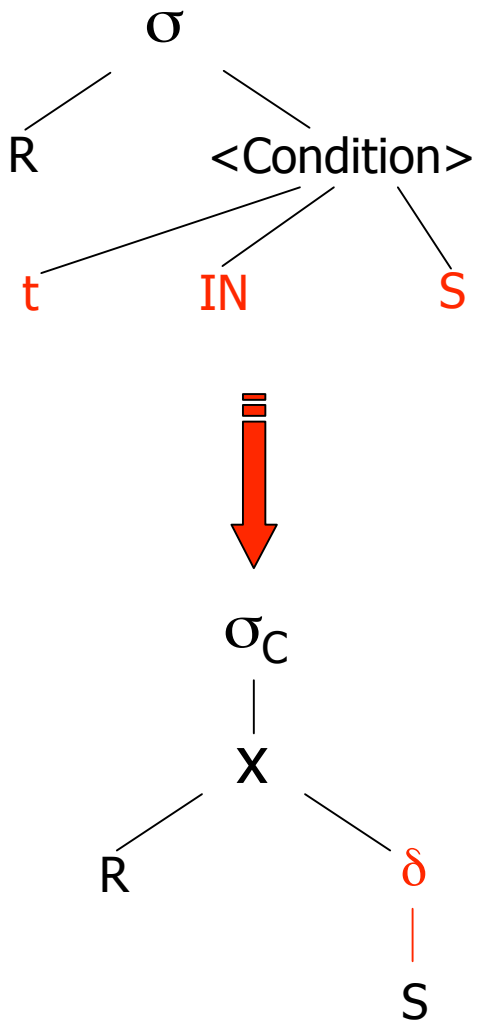
  - <Tuple> is represented by <Attribute> -- starName
  - the sub-query <Query> is the query we converted earlier

  - ⇨ This tree needs further transformation



Conversion into Relational Algebra – V

- Replace two-argument select:
  - different conditions require different rules
  - we will look at `t IN S`:
  - replace <Condition> with the tree representing S. If S may have duplicates we must include a δ–operator at the top
  - replace the two-argument selection by a one-argument selection σ_C, where C is the condition that equates each component of tuple t to the corresponding attribute in S
  - give σ_C an argument that is the product of R and S

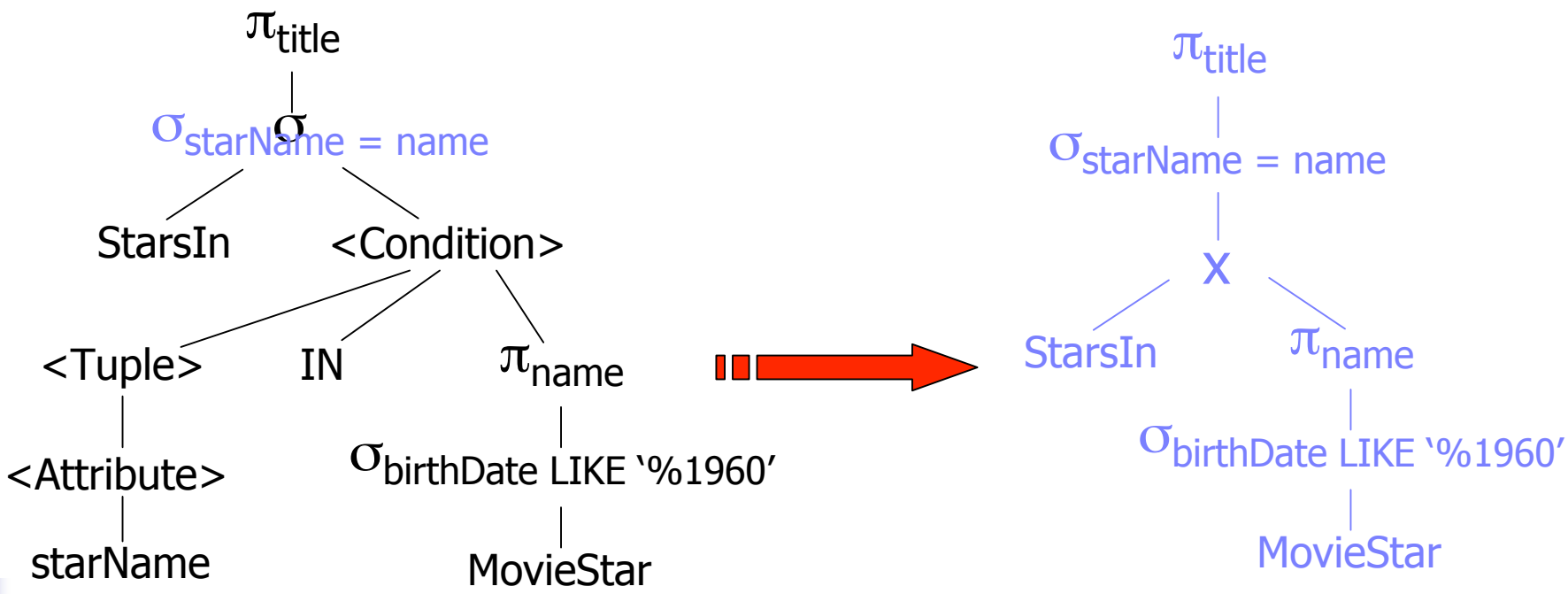


Conversion into Relational Algebra – VI

- Example (cont.):

`SELECT title FROM StarsIn WHERE starName IN (...)`

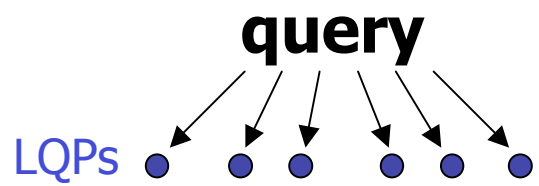
  - replace <Condition> with the tree representing the sub-query
  - replace the two-argument selection by a one-argument selection σ_C, where C is `starName = name`
  - give σ_C an argument that is the product of `StarsIn` and `MovieStar`



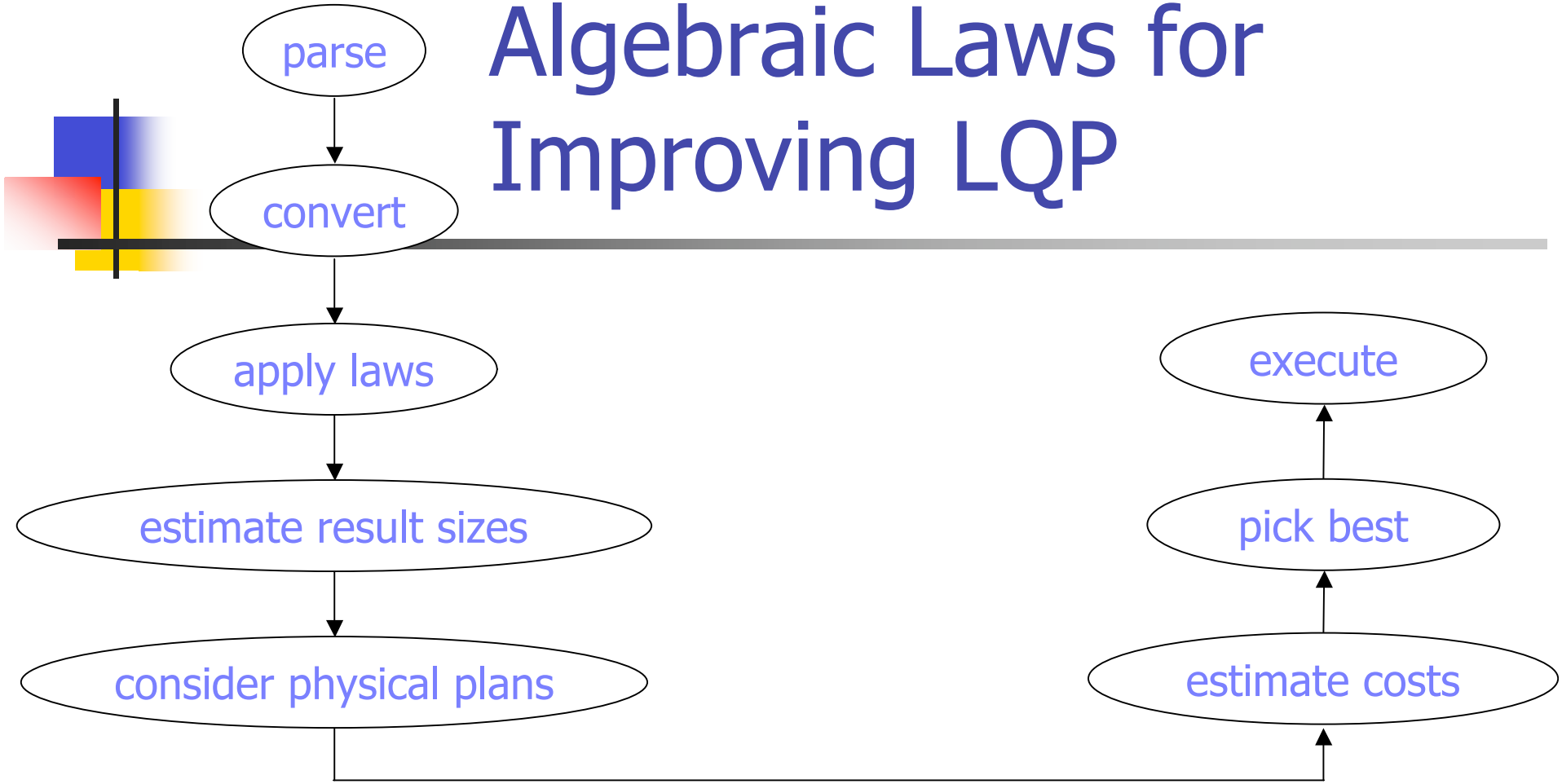
Conversion into Relational Algebra – VII

- Translating sub-queries can be more complex if the sub-query is correlated to values defined outside its scope
  - we must produce a relation with some extra attributes for comparison with external attributes
  - the extra attributes are later removed using projections
  - any duplicate tuples must be removed

- Translating the parse tree into expressions in algebra may give several equivalent LQP using different operators or just changing the order of the operators



Algebraic Laws for Improving LQP



Query Rewrite

- When we have translated the parse tree to a relational algebra expression, the next step is to optimize the expression:
  - possibly giving smaller temporary relations
  - possibly reducing the number of disk I/Os

⇨ The query is rewritten *applying algebraic laws* turning the expression into an equivalent expression that will have a more efficient physical query plan

Algebraic Laws

- The most common laws used for simplifying expressions are:
  - the commutative law allowing operators to be performed in any sequence, e.g.,
    $x \omega y = y \omega x$
    (where ω is an operator)
  - the associate law allowing operators to be grouped either from left or right, e.g.,
    $x \omega (y \omega z) = (x \omega y) \omega z$

Algebraic Laws: Joins and Products – I

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

Algebraic Laws: Joins and Products – II

- ✓ Does it matter in which order join or product are performed with respect to performance, e.g., R x S x T x …?

⇨ YES, it *may* be very important

- if only one of the relations fits in memory, we should perform the operation using this relation first – *one-pass* operation reducing the number of disk I/Os
- if joining or taking product of two of the relations in a large expression give a temporary relation which fits in memory, one should join these first to save both memory and disk I/Os
- one should try to make the temporary result as small as possible to save memory, result from final join or product may be final result going out to user
- if we can estimate (using statistics) the amount of tuples being joined, we can save a lot of operations by joining the two relations giving fewest tuples first (does not apply to products)

⇨ BUT, the final result will be the same

Algebraic Laws: Union and Intersect

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

- Note that laws for **sets** and **bags** can differ, e.g., (distributive law of intersection over union)

  $R \cap_S (S \cup_S T) = (R \cap_S S) \cup_S (R \cap_S T)$, but
  $R \cap_B (S \cup_B T) \neq (R \cap_B S) \cup_B (R \cap_B T)$,

Algebraic Laws: Select – I

- Select is a very important operator in terms of query optimization
  - reduces the number of tuples (size of relation)
  - an important general rule in optimization is to *push selects as far down the tree as possible*

- "Splitting" (AND and OR) laws:
  - $\sigma_{a \text{ AND } b}(R) = \sigma_a(\sigma_b(R))$
  - $\sigma_{a \text{ OR } b}(R) = (\sigma_a(R)) \cup_s (\sigma_b(R))$

    (works only for R a set, a bag-version will include a tuple twice in the last expression if both conditions are fulfilled)

- "Flexibility" (ordering) law:
  - $\sigma_a(\sigma_b(R)) = \sigma_b(\sigma_a(R))$

Algebraic Laws: Select – II

- Laws for pushing select – if pushing select, select …
  - … must be pushed through both arguments
    - union: $\sigma_a(R \cup S) = \sigma_a(R) \cup \sigma_a(S)$
    - cartesian product: $\sigma_a(R \times S) = \sigma_a(R) \times \sigma_a(S)$
  - … must be pushed through first arguments, optionally second
    - difference: $\sigma_a(R - S) = \sigma_a(R) - S = \sigma_a(R) - \sigma_a(S)$
  - … may be pushed through either one or both arguments
    - 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: $\sigma_a(R \bowtie_b S) = \sigma_a(R) \bowtie_b \sigma_a(S) = R \bowtie_b \sigma_a(S) = \sigma_a(R) \bowtie_b S$

NOTE:
for products and join it may not make sense to push select through both arguments, and even if it does, it may not improve the plan

Algebraic Laws: Select – III

- Example: each attribute is 1 byte
  - $\sigma_{A=2}(R \bowtie S)$
    - perform join:
      combine 4 * 4 elements = 16 operations
      store relation $R \bowtie S$ = 52 bytes
    - perform select:
      checks tuple-by-tuple: 2 operations
  - $\sigma_{A=2}(R) \bowtie S$
    - perform select:
      checks tuple-by-tuple: 4 operations
      store relation $\sigma_{A=2}(R)$ = 24 bytes
    - perform join:
      combine 1 * 4 elements = 4 operations
  - $R \bowtie \sigma_{A=2}(S)$
    does not make sense, a is not an attribute of S

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

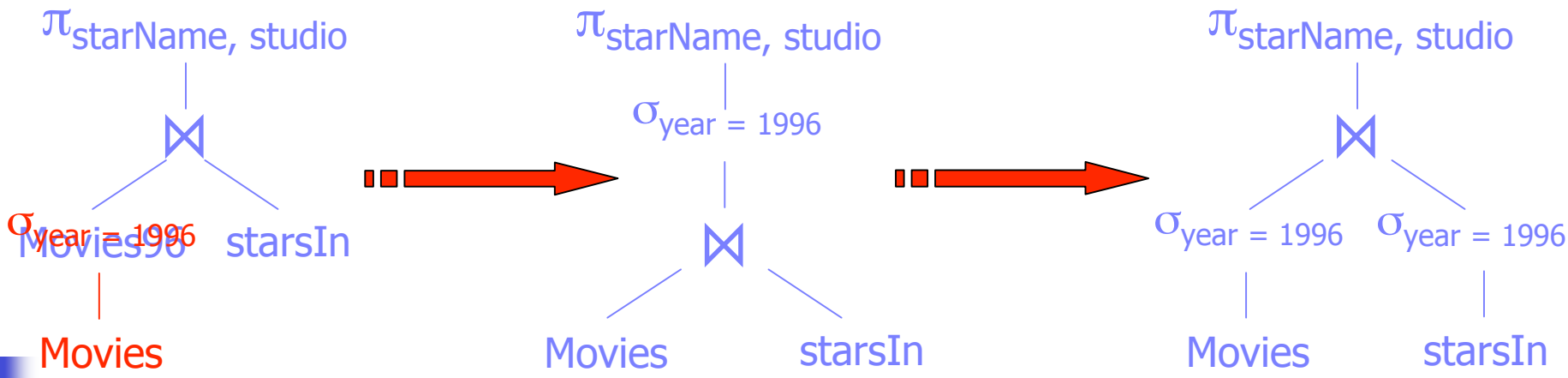
A	B	C	...	X
2	c	6	...	2

Algebraic Laws: Select – IV

- Sometimes it is useful to push selection the other way, i.e., up 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;`
  - Relational algebra tree:



Algebraic Laws: Project – I

- ✓ Projections can be pushed down through many operators:
  - a projection may be introduced anywhere as long as it does not remove any attributes used above in the tree
  - the projection operator is thus often not moved, we introduce a new
  - examples:
    - $\pi_L(R \bowtie S) = \pi_L(\pi_M(R) \bowtie \pi_N(S))$, if
      - M = join attribute or part of L in R
      - N = join attribute or part of L in S
    - $\pi_L(R \bowtie_C S) = \pi_L(\pi_M(R) \bowtie_C \pi_N(S))$, if
      - M = join attribute (part of C) or part of L in R
      - N = join attribute (part of C) or part of L in S
    - $\pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S))$, if
      - M = part of L in R
      - N = part of L in S

Algebraic Laws: Project – II

- Additionally, projections …
  - … can be pushed through a **bag**-union, but not **set**-union
  - … *cannot* be pushed through intersect or difference
  - … may be pushed through selections
    - $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$, if M is all attributes in L or part of condition C
- We usually wish to push projections as far down as possible as it reduces size of each tuple, but there are examples where this may cost time and resources, e.g.,
  - move before a select and we have an index on the stored relation
  - …

Algebraic Laws: Join, Product, Select, Project - I

- There are two laws that are important with respect to performance following 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
    - condition C equates each pair of tuples from R and S with the same name
    - L is a list of attributes including all distinct attributes from R and S
- If one has the opportunity to apply these rules, it generally will increase performance, because the algorithms for computing a join are much faster than computing the product followed by a selection on a very large relation

Algebraic Laws: Join, Product, Select, Project - II

- Example: $\pi_L(\sigma_{R.a = S.a}(R \times S))$ 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 attribute is 1 byte, a-attribute is key in both R and S
  - result: 100 tuples from S concatenated with tuples in R with matching a-attribute (assuming all tuples in S find a match)
  - $\pi_L(\sigma_C(R \times S))$:
    - perform product:
      combine 10.000 * 100 elements = 1.000.000 operations
      store relation R **x** S = 1.000.000 * (11 + 16) = 27.000.000 bytes
    - perform select:
      checks tuple-by-tuple: 1.000.000 operations
      store relation $\sigma_{R.a = S.a}(R \times S)$ = 100 * 27 = 2700 bytes
    - perform project:
      checks tuple-by-tuple: 100 operations
  - $R \bowtie S$:
    - perform join:
      check 10.000 * 100 elements = 1.000.000 operations

Algebraic Laws: Duplicate Elimination

- Duplicate elimination can reduce size of intermediate relations when pushed through
  - cartesian product: $\delta(R \times S) = \delta(R) \times \delta(S)$
  - join: $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$
  - theta-join: $\delta(R \bowtie_C S) = \delta(R) \bowtie_C \delta(S)$
  - select: $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$
  - **bag**-intersection: $\delta(R \cap_B S) = \delta(R) \cap_B \delta(S) = \delta(R) \cap_B S = R \cap_B \delta(S)$
- However, duplicate elimination cannot be pushed through
  - **set**-operations (make no sense)
  - **bag**-union and difference
  - projects

Algebraic Laws: Grouping and Aggregation

- Whether or not the grouping operator can be pushed depends on details of the aggregate operator used
  - cannot state general rules
  - MAX and MIN are not dependent on duplicates
    - $\gamma(R) = \gamma(\delta(R))$
  - SUM, COUNT, and AVG is dependent on duplicates
    - cannot push

Improving LQPs – I

- The described relational algebraic laws are used to improve – or rewrite – the LQPs generated from the parse tree to improve performance

- The *most commonly used* in query optimizers are:
  - push selects as far down as possible
    If the select condition consists of several parts, we often split the operation in several selects and push each select as far down as possible in tree
  - push projects as far down as possible
    Projects can be added anywhere as long as attributes used above in the tree is included
  - duplicate eliminations can sometimes be removed (e.g., if on key)
  - if possible, combine select with cartesian products to form a type of join

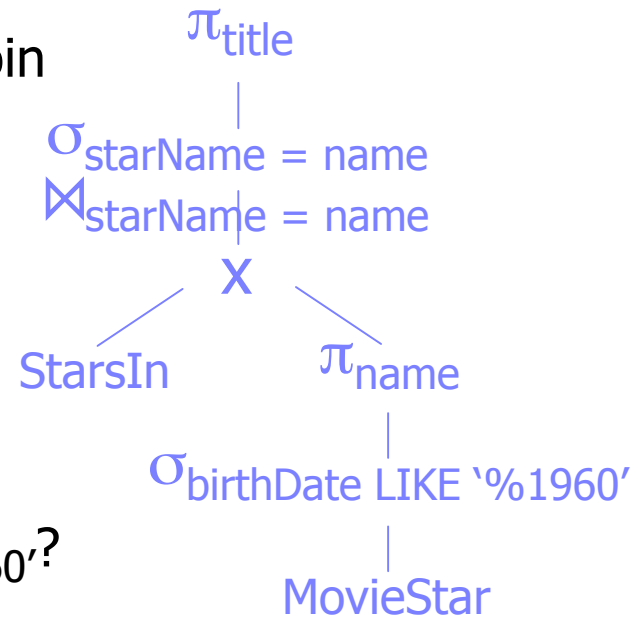
- But, no transformation is **always** good

Improving LQPs – II

- Example:

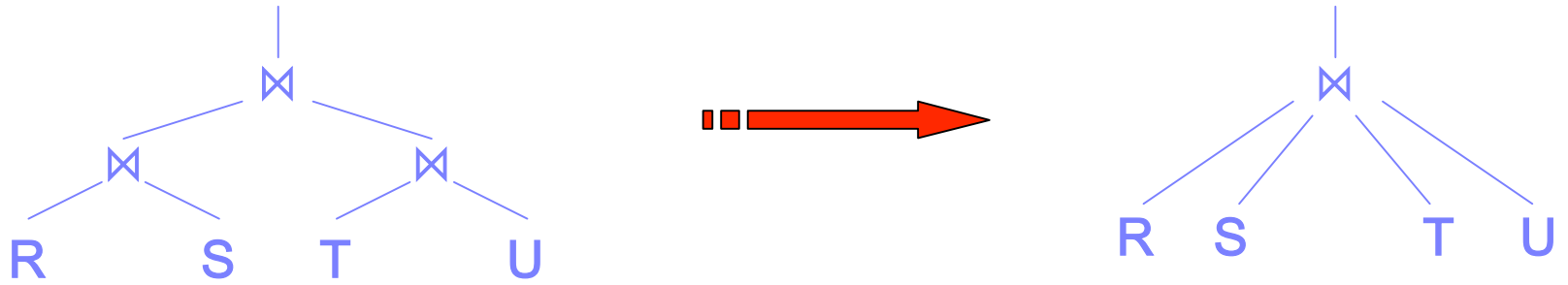
```
StarsIn(title, year, starName)
MovieStar(name, address, gender, birthDate);

SELECT title FROM StarsIn WHERE starName IN (...)
```

  - combine select and cartesian product into a join
  - Question:
    can we push π_{title} to StarsIn?
  - Question:
    can we push π_{name} before $\sigma_{birthDate\ LIKE\ '\%1960'}$?

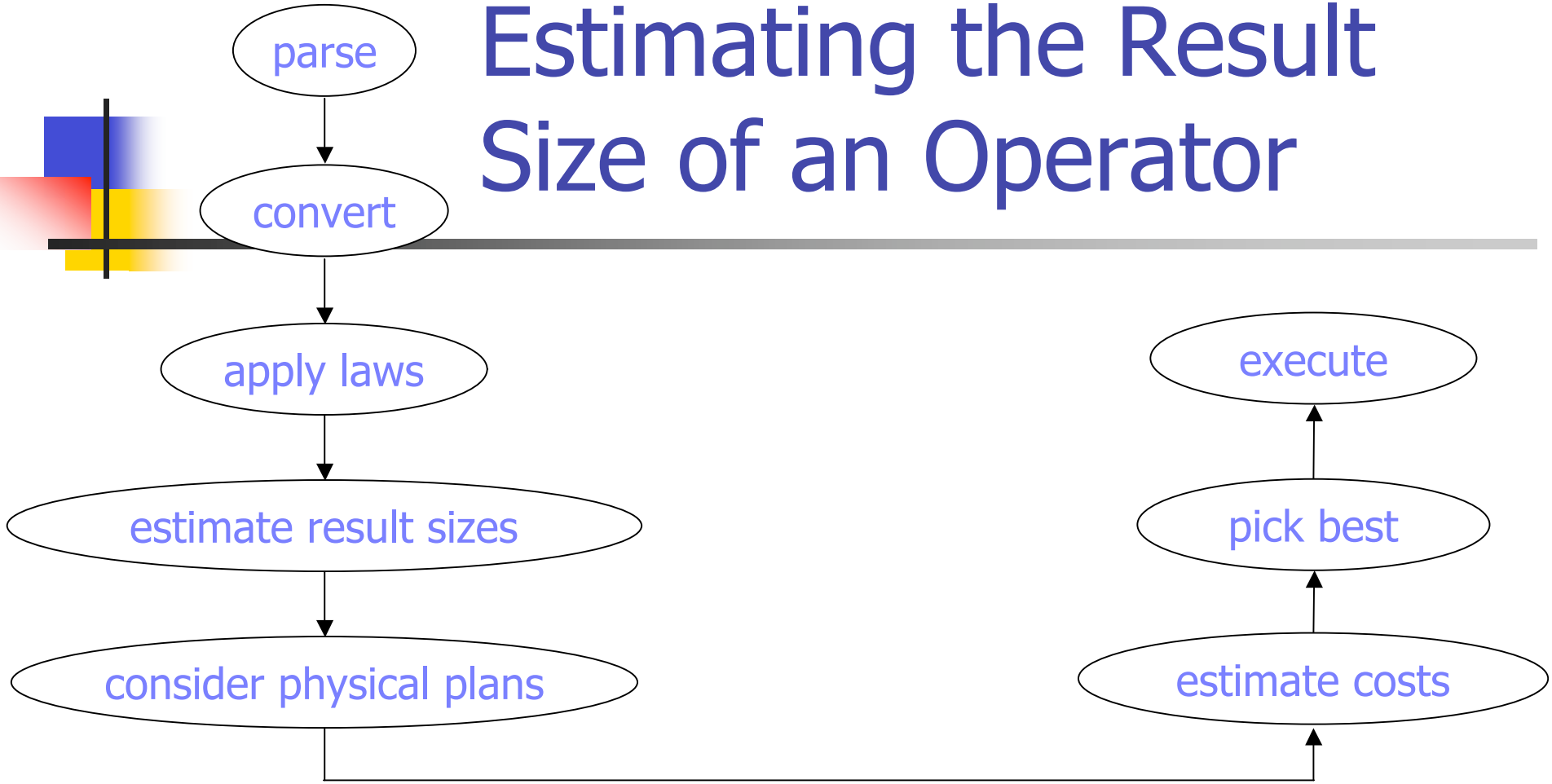


Grouping Operators

- To allow the query optimizer to reorder the operands in for a operator that is *both associative and commutative*, we may group nodes that have the same operator into one node with many children:



Estimating the Result Size of an Operator



Estimating Sizes – I

- ✓ The PQP is selected to minimize the estimated cost of the query
- ✓ The size of intermediate relations will have a large influence on costs as the choice of algorithm used for executing the operator is dependent on the amount of data and the amount of available memory
- ✓ Size estimation can be difficult, and ideally, the rules used should be:
  - accurate – a small error may result in choosing an inappropriate algorithm in the PQP
  - easy to compute – the overhead choosing a PQP should be minimal
  - logically consistent – not dependent how a operator is executed
- ⇨ BUT, no universally algorithms exists for computing sizes
- ✓ Fortunately, even inaccurate estimates helps picking a PQP

Estimating Sizes – II

- Notation reminder:
  - for a relation R
    - B(R) denotes the number of blocks to store all tuples
    - T(R) denotes the number of tuples in R
    - V(R, a) denotes the number of distinct values for attribute a (average identical a-value tuples is then T(R)/V(R,a))
  - additionally, we now add
    - S(R) denoting the size of a tuple in R
- For now, we will not count record headers, but when storing tuples on blocks, the size of these must be added to the size of each tuple

Size of a Projection – I

- The size of a projection (π) is computable
  - produces one tuple for each argument tuple
  - change the size of the tuple only, removing attributes (or adding new components that are combinations of other)
  - $\text{sizeof}[\pi_{A, B, \ldots}(R)] = T(R) * [\text{sizeof}(R.A) + \text{sizeof}(R.B) + \ldots]$

Size of a Projection – II

Relation 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

T(R) = 5

S(R) = 58

V(R,A) = 5
V(R,B) = 2
V(R,C) = 4
V(R,D) = 3

- ✓ Example: sizeof[$\pi_{A,B,\ldots}(R)$] = T(R) * [sizeof(R.A) + sizeof(R.B) + …]
  - sizeof(R) = T(R) * S(R) = 5 * 58 = 290 byte
  - sizeof[$\pi_{A}(R)$] = 5 * 4 = 20 byte
  - sizeof[$\pi_{B,C}(R)$] = 5 * (20 + 4) = 120 byte
  - sizeof[$\pi_{A,B,C,D,(A+10)\rightarrow E}(R)$] = 5 * (4 + 20 + 4 + 30 + 4) = 310 byte

Size of a Select – I

- A select (σ) reduces the number of tuples, but the size of each tuple is the same:
  - sizeof[$\sigma_X(R)$] = T($\sigma_X(R)$) * S(R), where X is the condition selecting tuples
  - how to estimate the number of tuples depends on
    - *value distribution* of attribute Y – we assume a uniform distribution where we use V(R,Y) to estimate the number of tuples returned by the selection
    - *condition* upon which the tuples are selected
- Equality selection, $\sigma_{A = c}(R)$, for attribute A and constant c:
  - T($\sigma_{A=c}(R)$) = T(R) / V(R, A)
- Inequality selection, $\sigma_{A < c}(R)$, for attribute A and constant c:
  - estimate the fraction of R having tuples satisfying the condition
  - usually the fraction is small – one third of all tuples frequently used
  - T($\sigma_{A<c}(R)$) = T(R) / 3

Size of a Select – II

- Not-equal selection, $\sigma_{A \neq c}(R)$, for attribute A and constant c:
  - rarely used
  - can usually use $T(\sigma_{A \neq c}(R)) = T(R)$ for simplicity
  - more accurately, subtract a fraction 1 / V(R,A)
  - $T(\sigma_{A \neq c}(R)) = T(R) * [(V(R,A) - 1) / V(R,A)]$

- Selection using several conditions with **AND**, $\sigma_{A \textbf{ AND } B \textbf{ AND} \ldots}(R)$
  - treat selection as a cascade of several selections
  - estimated size is original size multiplied by the *selectivity factor*, often
    - 1/3 for in-equality (<, >, ≤, ≥)
    - 1 for non-equality (≠)
    - 1 / V(R,A) for equality (=) on attribute A
  - $T(\sigma_{A \textbf{ AND } B \textbf{ AND} \ldots}(R)) = T(R) * \text{selectivity factor}_A * \text{selectivity factor}_B * \ldots$

Size of a Select – III

- Selection using several conditions with **OR**, $\sigma_{A \textbf{ OR } B \textbf{ OR} \ldots}(R)$
  - assume no tuple satisfies more than one condition
  - 1. approach: $T(\sigma_{A \textbf{ OR } B \textbf{ OR} \ldots}(R)) = T(\sigma_A(R)) + T(\sigma_B(R)) + \ldots$
  - 2. approach: $T(\sigma_{A \textbf{ OR } B \textbf{ OR} \ldots}(R)) = \min(T(R), (T(\sigma_A(R)) + T(\sigma_B(R)) + \ldots))$
  - 3. approach:
    - assume m_1 tuples satisfy first condition, m_2 satisfy second condition, ...
    - $1 - m_x/T(R)$ then is the fraction of tuples not satisfied by x'th condition
    - $T(\sigma_{A \textbf{ OR } B \textbf{ OR} \ldots}(R)) = T(R) * [1 - (1 - m_1/T(R)) * (1 - m_2/T(R))]$
- Selection conditions with **NOT**, $\sigma_{\textbf{NOT } A}(R)$
  - $T(\sigma_{\textbf{NOT } A}(R)) = T(R) - T(\sigma_A(R))$

Size of a Select – IV

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

T(R) = 5

S(R) = 58

V(R,A) = 5
V(R,B) = 2
V(R,C) = 4
V(R,D) = 3

✓ Example: *number of tuples*

- $T(\sigma_{A = 3}(R))$ = **T(R) / V(R, A)** = 5 / 5 = 1
- $T(\sigma_{B = \text{'cat'}}(R))$ = **T(R) / V(R, B)** = 5 / 2 = 2,5 ≈ 3
- $T(\sigma_{A > 2}(R))$ = **T(R) / 3** = 5 / 3 = 1,67 ≈ 2
- $T(\sigma_{B \neq \text{'cat'}}(R))$ = **T(R)** = 5

= **T(R) * [(V(R,B) – 1) / V(R,B)]** = 5 * ((2-1)/2) ≈ 3

NOTE:
we have estimated the number of tuples only. The size is given by the number of tuples multiplied with the size of the tuples – S(R) * T(σ(R))

Size of a Select – V

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

T(R) = 5

S(R) = 58

V(R,A) = 5
V(R,B) = 2
V(R,C) = 4
V(R,D) = 3

✓ Example: *number of tuples*

- $T(\sigma_{C = 1999 \textbf{ AND } A < 4}(R))$ = **T(R) * 1/V(R,C) * 1/3** = 5 * 1/4 * 1/3 ≈ 1
- $T(\sigma_{\text{NOT } A = 3}(R))$ = **T(R) - T($\sigma_{A = 3}$(R))** = 5 – 1 = 4
- $T(\sigma_{\text{NOT } C = 1999 \text{ AND } A < 4}(R))$ = **T(R) * (1 - 1/V(R,C)) * 1/3**

  = 5 * (1 – 1/4) * 1/3) = 1.25 ≈ 2

Size of a Select – VI

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

T(R) = 5

S(R) = 58

V(R,A) = 5
V(R,B) = 2
V(R,C) = 4
V(R,D) = 3

✓ Example: *number of tuples*

- $T(\sigma_{C = 1999 \textbf{ OR } A < 4}(R)) = \mathbf{T(\sigma_{C = 1999}(R)) + T(\sigma_{A < 4}(R))} =$

$= T(R)/V(R,C) + T(R)/3 = 5/4 + 5/3 \approx 2 + 2 = 4$

$= \mathbf{\min[T(R), T(\sigma_{C = 1999}(R)) + T(\sigma_{A < 4}(R))]} = 4$

$= \mathbf{T(R) * [1 - (1 - m_1/T(R)) * (1 - m_2/T(R))]}$

$= 5 * [1 - (1- 5/4 / 5)(1-5/3 / 5)] =$

$= 5 * [1 - 0,75*0,67] \approx 2,5 \approx 3$

Size of a Product

- As with projections, we can exactly compute the size of a cartesian product (x)
  - produces one tuple for each possible combination of each tuple in relation R and S:
    T(RxS) = T(R) * T(S)
  - the size of each new tuple is the sum of the size of each original tuple:
    S(RxS) = S(R) + S(S)
  - sizeof(R x S) = T(RxS) * S(RxS) = T(R) * T(S) * (S(R) + S(S))

Size of a Join – I

- In our size estimations for join, we will look at natural join (⋈), but other joins is managed similarly
  - equi-join as natural join
  - theta-joins as a cartesian product followed by a selection

- Estimating the size of a join of R(x,y) and S(y,z) is a challenge, because *we do not know how the join attribute y relates* in the relations R and S, e.g.:
  - disjoint sets of y-values – empty join:
    $T(R \bowtie S) = 0$
  - y is key in S, and a foreign key to R – each tuple in R joins with one tuple in S:
    $T(R \bowtie S) = T(R)$
  - Almost all tuples of R and S have the same y-value A – combine all tuples of each relation:
    $T(R \bowtie S) = T(R) * T(S)$

Size of a Join – II

- For our calculations, we will make two assumptions:
  - containment of value sets:
    - if attribute y appears in several relations, the values are chosen from the front of a given list of values
    - thus, if $V(R, y) \leq V(S, y)$, then every y-value in R will match a y-value in S
    - may certainly be violated, but holds in many cases, e.g., y is key in S, and a foreign key to R
  - preservation of value sets:
    - non-join attributes will not lose any values from its set of possible values
    - thus, $V(R \bowtie S, y) = V(R, y)$
    - is violated if there are "dangling tuples" in R

Size of a Join – III

- The size of $R(x, y) \bowtie S(y, z)$ in number of tuples can now be estimated as follows:
  - assume $V(R, y) \leq V(S, y)$, i.e.,
    every tuple *t* in R have a chance of $1/V(S, y)$ of joining with a given tuple in S
  - S has $T(S)$ tuples, i.e.,
    the expected number of tuples the tuple *t* from R joins with is $T(S)/V(S, y)$ - number of tuples with same y-value
  - $T(R \bowtie S) = T(R) * T(S) / V(S, y)$
  - if $V(S, y) \leq V(R, y)$ → $T(R \bowtie S) = T(S) * T(R) / V(R, y)$
  - in general, $T(R \bowtie S) = T(S) * T(R) / \max[V(R, y), V(S, y)]$

Size of a Join – IV

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

- Example:
  find T(A ⋈ B ⋈ C)
  - **T((A ⋈ B) ⋈ C):**
    - T(A ⋈ B) = T(A) * T(B) / max[V(A, b), V(B, b)]
      = 10000 * 2000 / max(1000, 100) = 20000
    - V(A ⋈ B, c) = V (B, c) = 1000 (preservation of value sets)
    - T((A ⋈ B) ⋈ C) = T(A ⋈ B) * T(C) / max[V(A ⋈ B, c), V(C, c)]
      = 20000 * 5000 / max(1000, 100) = 100.000
  - **T(A ⋈ (B ⋈ C)):**
    - T(B ⋈ C) = T(B) * T(C) / max[V(B, c), V(C, c)]
      = 2000 * 5000 / max(1000, 100) = 10000
    - V(B ⋈ C, b) = V (B, b) = 100 (preservation of value sets)
    - T(A ⋈ (B ⋈ C)) = T(B ⋈ C) * T(A) / max[V(B ⋈ C, b), V(A, b)]
      = 10000 * 10000 / max(100, 1000) = 100.000

Size of a Join – V

- If there are *more than one join attribute*, $R(x, y_1, y_2,\ldots) \bowtie S(y_1, y_2,\ldots, z)$, we must consider the probability that all join tuples find a match in the other relation:
  - for all $V(R, y_x) \leq V(S, y_x)$, the probability for tuple *t* in R can be joined with a certain tuple on the y_x attribute in S is $1/V(S, y_x)$
  - likewise, for all $V(S, y_x) \leq V(R, y_x)$, the probability for tuple *s* in S can be joined with a certain tuple on the y_x attribute in R is $1/V(R, y_x)$
  - $$T(R \bowtie S) = \frac{T(S) * T(R)}{\max[V(R, y_1), V(S, y_1)] * \max[V(R, y_2), V(S, y_2)] * \ldots}$$

    for each y_x attribute that is common in R and S

Size of a Join – VI

- Example:
  find $T(A \bowtie B)$

A(a, b, c)	B(b, c, d)
T(A) = 10.000	T(B) = 2.000
V(A, a) = 5.000	V(B, b) = 100
V(A, b) = 1.000	V(B, c) = 1.000
V(A, c) = 200	V(B, d) = 2.000

  - join on b and c:

$$T(A \bowtie B) = \frac{T(A) * T(B)}{\max[V(A, b), V(B, b)] * \max[V(A, c), V(B, c)]}$$

$$= \frac{10.000 * 2000}{\max[1000, 100] * \max[200, 1000]} = 20$$

Size of a Join – VII

- The general case of a natural join, $R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n$:
  - an attribute A appear in *k* of the *n* relations
  - the probability for that all these k relations agreeing in attribute A is then
    - $1 / v_2 * v_3 * \ldots * v_k$
    - $v_1 = \min(V(R_1, A), V(R_2, A), V(R_3, A), \ldots, V(R_k, A))$
  - general formula for finding size of any join:
    - find the maximum number of tuples using the product of the number of tuples in all relations – $T(R_1) * T(R_2) * T(R_3) * \ldots * T(R_n)$
    - then, for each attribute A appearing in more than one relation, divide the above result by all, but the least V(R, A)

Size of a Join – VIII

- Example:
  find T(A ⋈ B ⋈ C)

A(a, b, c)	B(b, c, d)	C(b, d, e)
T(A) = 10.000	T(B) = 2.000	T(C) = 5.000
V(A, a) = 5.000	V(B, b) = 50	V(C, b) = 100
V(A, b) = 1.000	V(B, c) = 1.000	V(C, d) = 100
V(A, c) = 50	V(B, d) = 200	V(C, e) = 100

- maximum number of tuples:
  T(A) * T(B) * T(C) = 10000 * 2000 * 5000 = 100.000.000.000

- for each attribute X appearing in more than one relation
  - b appear in all relations, V(A, b) = 1000, V(B, b) = 50, V(C, b) = 100
    → divide by 1000 * 100
  - c appear in all A and B, V(A, c) = 50, V(B, c) = 1000
    → divide by 1000
  - d appear in all B and C, V(B, d) = 200, V(C, d) = 100
    → divide by 200

- $T(A \bowtie B \bowtie C) = \frac{100.000.000.000}{(1000 * 100) * (1000) * (200)} = 5$

Size of a Join – IX

- So far, we have only calculated the number of tuples, but the size of a join is given by

$$sizeof(A \bowtie B) = T(A \bowtie B) * S(A \bowtie B)$$

- However, the size of the tuples from a join is dependent on which kind of join we perform, e.g.,
  - in a natural join, the join attributes only appear once
  - in a theta-join, all attributes from all relations appear

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

Size of a Union

- The number of tuples of a union ($\cup$) is dependent of whether it is a **set**- or **bag**-version:
  - bag:
    the result is exactly the sum of the tuples of all the arguments - $T(A \cup_b B) = T(B) + T(B)$
  - set:
    - as bag-version if disjoint relations
    - usually somewhere between sum of both and the number of the larger relation:
    - may for example use: $T(A \cup_s B) = T(A) + T(B)/2$
      where B is the smaller relation

Size of an Intersection and a Difference

- The number of tuples of an intersection (∩) can be
  - 0 if disjoint relations
  - min(T(R), T(S)) if one relation contains only a subset of the other
  - usually somewhere in-between –
    may for example use average: min(T(R), T(S)) / 2

- The number of tuples of a difference (–), R – S, is
  - T(R) if disjoint relations
  - T(R) – T(S) if all tuples in S also is in R
  - usually somewhere in-between –
    may for example use: T(R) – T(S)/2

Size of a Duplicate Elimination

- The number of tuples of a duplicate elimination (δ) is the same as the number of distinct tuples
  - 1 if all tuples are the same
  - T(R) if all tuples are different
  - one approach:
    - given $V(R, a_i)$ for all *n* attributes, the maximun number of different tuples are $V(R, a_1) * V(R, a_2) * \ldots * V(R, a_n)$
    - let estimated number of tuples be the smaller of this number and the number of tuples in the relation

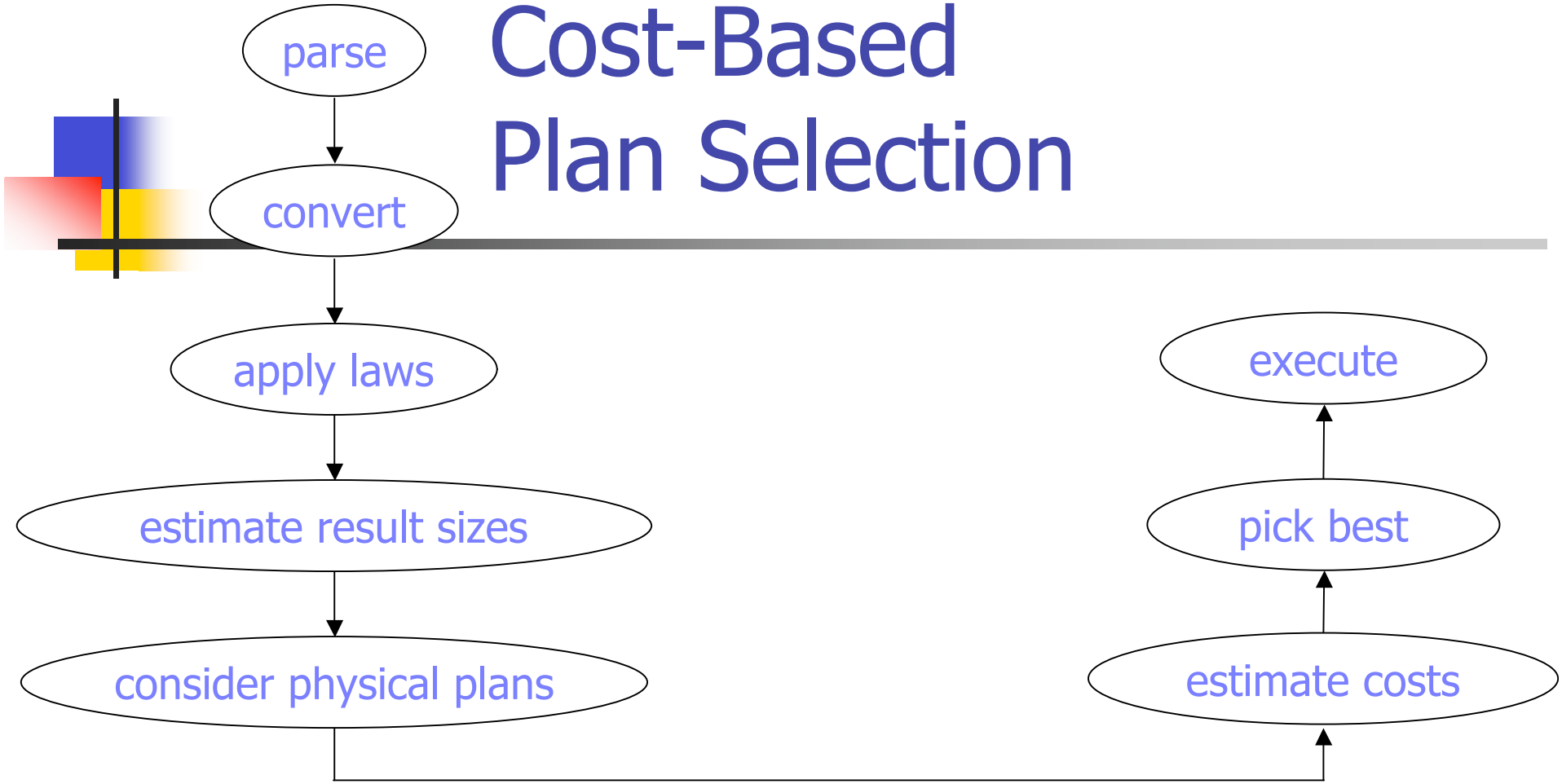
Size of a Grouping

- The number of tuples of a grouping (γ) is the same as the number of groups
  - 1 if all tuples are the same
  - T(R) if all tuples are different
  - one approach:
    - given $V(R, a_i)$ for all *n* attributes, the maximun number of different tuples are $V(R, a_1) * V(R, a_2) * \ldots * V(R, a_n)$
    - let estimated number of tuples be the smaller of this number and the number of tuples in the relation
  - Note that the size of each tuple can be different compared to the argument tuples

Obtaining Estimates for Size Parameters

- To estimate the size of the intermediate relations, we have used parameters like T(R) and V(R, a)
- The DBMS keeps statistics from previous operations to be able to provide such parameters
- However, computing statistics are *expensive* and should be recomputed periodically only:
  - statistics usually have few changes over a short time
  - even inaccurate statistics are useful
  - ⇨ statistics recomputation might be triggered after some period of time or after some number of updates

Cost-Based Plan Selection



Cost-Based Plan Selection

- The query optimizer estimates the costs of all generated plans
- As before, we will use disk I/Os, but this number is influenced by several factors:
  - which logical operators are chosen to implement the query
  - sizes of intermediate results
  - which physical operators are chosen to implement the logical operators
  - order of operations
  - method of passing arguments between physical operators

Comparing Intermediate Sizes for LQPs – I

- There may exist several LQPs for a given query, and we compare them by the size of intermediate relations
  - estimate the intermediate size of each operator in the LQP
  - add the cost into the LQP tree
  - the cost of the LQP is the sum of all costs in the tree, except the nodes not dependent on the LQP:
    - the root – the final result is given to the application
    - the leaves – data stored on disk

Comparing Intermediate Sizes for LQPs – II

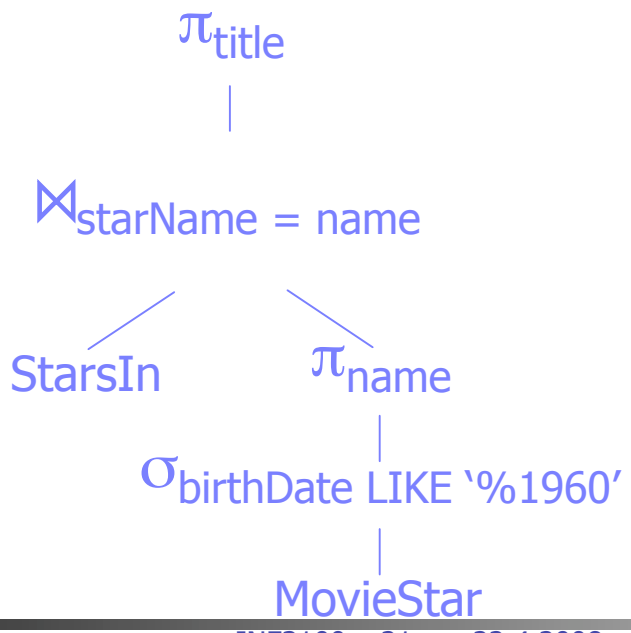
- Example:

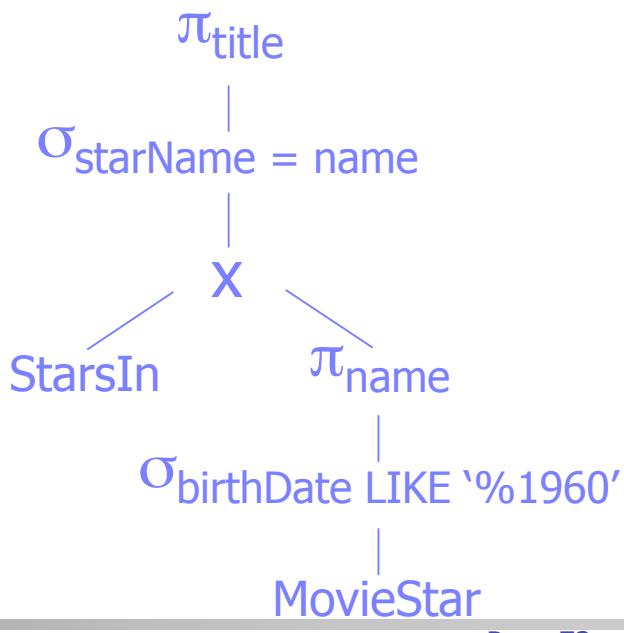
```
StarsIn(title, year, starName)
MovieStar(name, address, gender, birthDate)

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

Statistics:

T(StarsIn) = 10.000
V(StarsIn, starName) = 500
S(StarsIn) = 60

T(MovieStar) = 1.000
V(MovieStar, name) = 1.000
V(MovieStar, birthDate) = 50
S(MovieStar) = 100





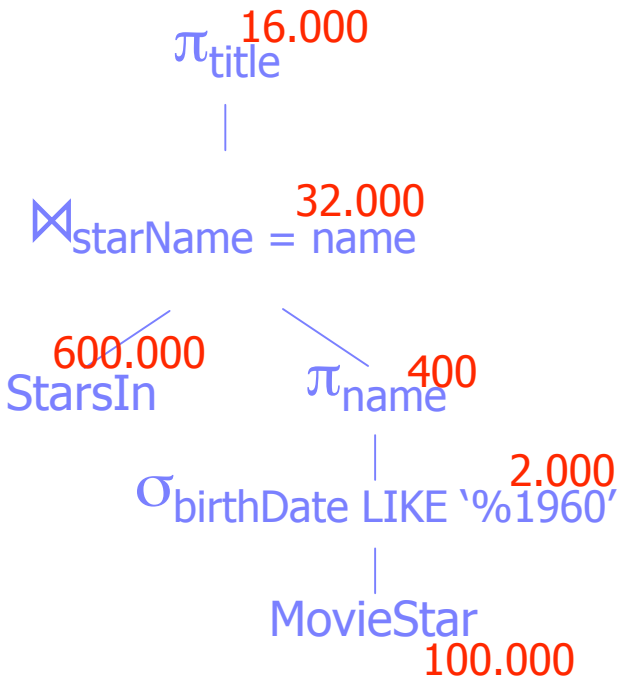
Comparing Intermediate Sizes for LQPs – III

- Example:
  - $A_1 = \sigma_{\text{birthDate LIKE '\%1960'}}(MS)$:
    - $T(\sigma(MS)) = T(MS) / V(MS, \text{birthDate}) = 1000 / 50 = 20$
    - $\text{sizeof}(A_1) = 20 * 100 = 2000$
  - $A_2 = \pi_{\text{name}}(A_1)$:
    - $T(\pi(A_1)) = T(A_1) = 20$
    - assume attribute name is 20 byte
    - $\text{sizeof}(A_2) = 20 * 20 = 400$
  - $A_3 = SI \bowtie A_2$:
    - $T(SI \bowtie A_2) =$ $T(SI)*T(A_2) / \max[V(SI, \text{starName}), V(A_2, \text{name})] =$ $10000 * 20 / \max(500, 20) = 400$
    - $S(A_2) = 20$
    - $\text{sizeof}(A_3) = 400 * (60 + 20) = 32000$
  - $A_4 = \pi_{\text{title}}(A_3)$:
    - $T(\pi(A_3)) = T(A_3) = 400$
    - assume title is 40 bytes
    - $\text{sizeof}(A_4) = 400 * 40 = 16000$

NOTE:
name is key in MS, and we have 20 tuples left

Statistics:
T(SI) = 10.000
V(SI, starName) = 500
S(SI) = 60

T(MS) = 1.000
V(MS, name) = 1.000
V(MS, birthDate) = 50
S(MS) = 100



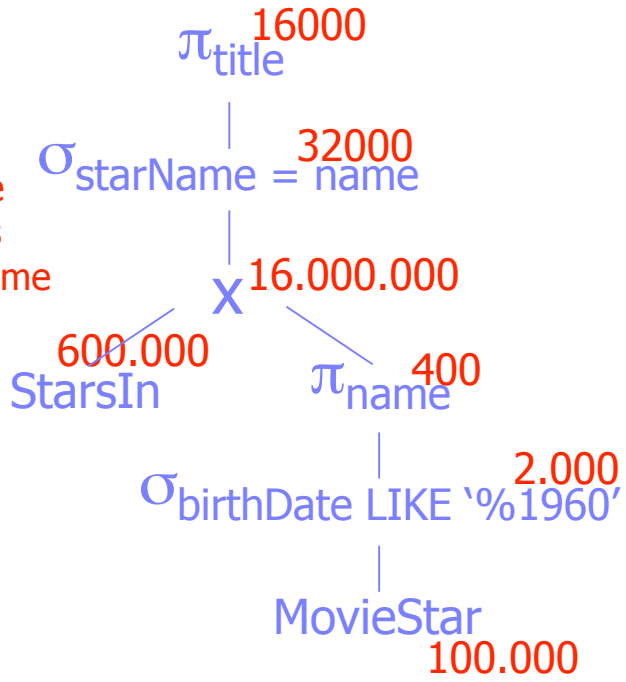
Comparing Intermediate Sizes for LQPs – IV

- Example:
  - $A_1 = \sigma_{birthDate\ LIKE\ '\%1960'}(MS)$ → as previous: 2000, $T(\sigma(MS))=20$
  - $A_2 = \pi_{name}(A_1)$ → as previous: 400, $T(A_2) = T(A_1) = 20$
  - $A_3 = SI \times A_2$:
    - $T(SI \times A_2) =$ $T(SI) * T(A_2) = 10000 * 20 = 200.000$
    - $S(A_2) = 20$
    - $sizeof(A_3) = 200.000 * (60 + 20) = 16.000.000$
  - $A_4 = \sigma_{starName = name}(A_3)$:
    - $T(\sigma(A_3)) =$ $T(A_3) / \max(V(A_3, name), V(SI, starName))$ $= 200.000 / \max(20, 500) = 400$
    - $S(A_4) = S(SI) + S(A_3) = 60 + 20 = 80$
    - $sizeof(A_4) = 400 * 80 = 32000$
  - $A_5 = \pi_{title}(A_4)$ → as previous: $400 * 40 = 16000$

NOTE:
does not match any of the rules we have seen so far for select, but it is equal to the join condition – use same

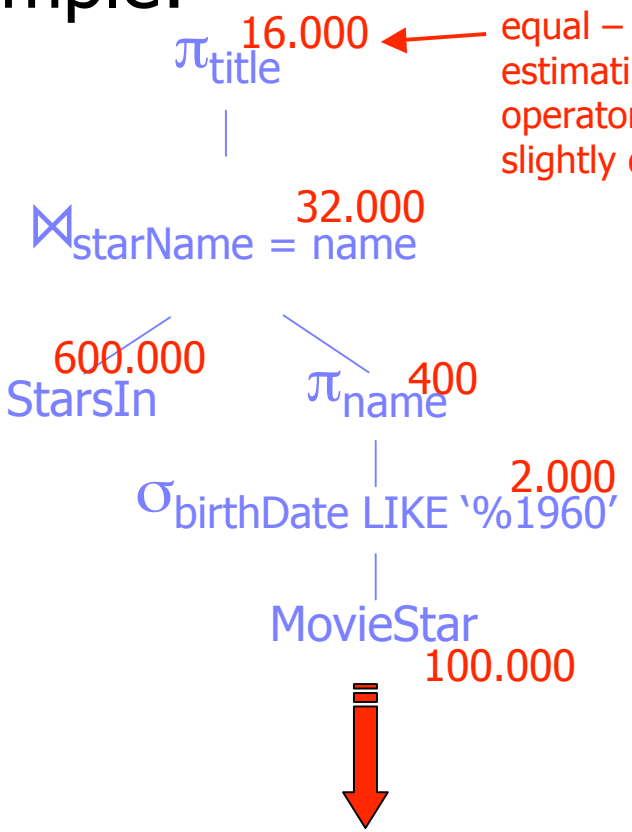
Statistics:
T(SI) = 10.000
V(SI, starName) = 500
S(SI) = 60

T(MS) = 1.000
V(MS, name) = 1.000
V(MS, birthDate) = 50
S(MS) = 100

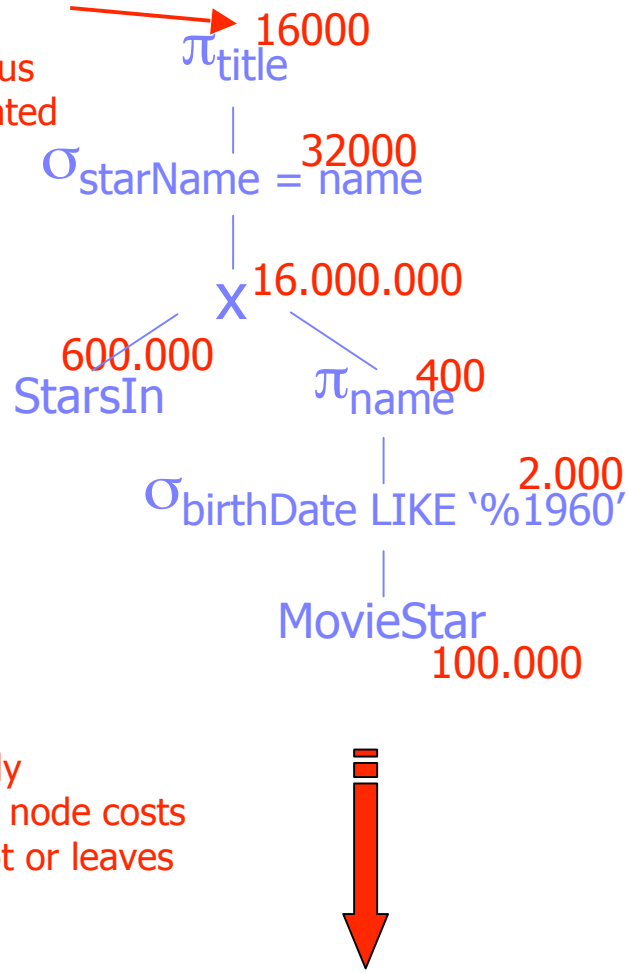


Comparing Intermediate Sizes for LQPs – V

- ✓ Example:



NOTE 2:
does not necessary have to be equal – remember we are estimating sizes and various operators might be estimated slightly different



NOTE 1:
we count only intermediate node costs only, not root or leaves

Total intermediate size:
2000 + 400 + 32000 = 34400

Total intermediate size:
2000 + 400 + 16000000 + 32000= 16034400

Conversion of a LQP to a PQP

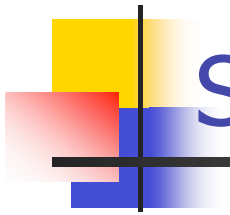
- When we shall convert a LQP to a PQP, there are a lot of different factors that must be considered
- Each different plan is given an estimated cost and the plan with smallest costs is selected
- There are many approaches to enumerate the cost estimates of a PQP, i.e., finding the "cheapest" plan
  - exhaustive
  - heuristic
  - branch-and-bound
  - hill climbing
  - dynamic programming
  - Selinger-style optimizations

Plan Enumeration – I

- Exhaustive:
  - consider all combinations of choices in a plan
  - estimate the cost of each plan
  - many plans, expensive
- Heuristic
  - choose a plan according to heuristic rules, i.e., on earlier experiences on efficient operators like
    - use index on operations like $\sigma_{A = 10}(R)$
    - use smallest relations first in a join of many relations
    - if arguments are sorted, use a sort-based operator
    - ...
  - fast, but based on general rules only

Plan Enumeration – II

- Branch-and-Bound:
  - find a plan using heuristic rules
  - then, consider small parts of this plan to see if it can be optimized
- Sellinger-style optimization:
  - keep for all sub-expressions the cost and expected kind of result
  - thus, a operator might have a higher individual cost, but if the result for example is sorted, later operators may use this
    - if considering intermediate sizes – no gain
    - if considering disk I/Os, one might save the first part of the sort-based operation saving disk I/Os and a lot of CPU operations



Selection of Algorithms – I

- After having determined the order of different operators, we must choose which algorithm that should implement an operator

- Such a choice is dependent of several factors
  - storage
  - existence of indexes
  - conditions of the operator
  - available memory
  - ...

Selection of Algorithms – II

- Example: selection method
  - R(x, y, z), T(R) = 5000, B(R) = 200, V(R, x) = 100, V(R, y) = 500
  - indexes on all attributes, index on z is clustered
  - $\sigma_{x=1 \text{ AND } y=2 \text{ AND } z<5}(R)$
  - table-scan – read block-by-block:
    - cost: B(R) = 200 disk I/Os since R is clustered
  - index-scan with x-index – find tuples x=1 using index, then check y and z:
    - x-index is not clustered, worst-case all tuples on different blocks
    - cost: T(R) / V(R, x) = 5000 / 100 = 50 disk I/Os
  - index-scan with y-index – find tuples y=2 using index, then check x and z:
    - y-index is not clustered, worst-case all tuples on different blocks
    - cost: T(R) / V(R, y) = 5000 / 500 = 10 disk I/Os
  - index-scan with z-index – find tuples z<5 using index, then check x and y:
    - we have estimated selections like this as 1/3 of the tuples, R is sorted on z
    - cost: B(R) / 3 = 67 disk I/Os

Selection of Algorithms – III

- Choosing join method if we are unaware of available resources
  - chose one-pass hoping we have enough memory
  - chose sort join if...
    - ... both arguments already is sorted
    - ... joining three or more relations on same attribute
  - chose index join if one relation is small and have index on other
  - chose hash-join otherwise as it requires less memory

Pipelining Versus Materialization - I

- The last major question is how to pass intermediate results between operators

- Two ways:
  - pipelining –
    pass result directly to new operator ,i.e., data remains in memory, enabling operations to be interleaved
    - possibly more efficient
    - requires more memory – possibly again requiring more disk accesses
  - materializations –
    store all intermediate results on disk until it is needed by another operator
    - must store all intermediate data – write to disk and retrieve again when needed
    - may allow easier algorithms as one operator may have more memory

Pipelining Versus Materialization - II

- *Unary operations*, selection and projection, should be pipelined as operations are performed on tuple-by-tuple

- *Binary operations* can be pipelined, but
  - the number of buffers needed for computation vary
  - the size of the result vary
  - ⇨ choice of whether to pipeline the result depends on memory

Note: Example 16.36, page 864 – 867 is wrong
The first two-pass hash-join makes 100 buckets of 50 blocks

Pipelining Versus Materialization - III

- Example: $[R(w,x) \bowtie S(x,y)] \bowtie T(y,z)$
  - B(R) = 5000, B(S) = 10.000, B(T) = 15.000, M = 151
  - use hash-join, one- or two-pass depending on memory
  - if $B(R \bowtie S) = k$, what is most useful for different values of k?

  - First, use two-pass hash-join on R and S as neither fits in memory
    - each bucket of the smaller relation must not exceed 150
      → assume partitioning R into 50 buckets give 100 blocks each
    - phase two – joining needs 101 blocks, 50 free for result
    - cost: 3B(R) + 3B(S)
      - read and write R to partition into buckets: 2 * 5000 = 10.000
      - read and write S to partition into buckets: 2 * 10000 = 20.000
      - read buckets-pairs and join – each block one time: 5000 + 10.000 = 15.000
      - total $R \bowtie S$ cost: 45.000 disk I/Os
        (assuming result in memory)

Pipelining Versus Materialization - IV

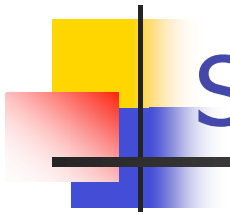
- Example: $[R(w,x) \bowtie S(x,y)] \bowtie T(y,z)$
  - B(R) = 5000, B(S) = 10.000, B(T) = 15.000, M = 151
  - if $B(R \bowtie S) = k \leq 50$
    - keep result in memory
    - reuse 101 available blocks to read T and join tuple by tuple (one-pass)
    - cost:
      - $R \bowtie S$: 45.000
      - read all blocks of T: 15.000
      - total $R \bowtie S \bowtie T$ cost: 60.000 disk I/Os
    - using materialization – write intermediate result back to disk and reread
      - total $R \bowtie S \bowtie T$ cost: 60.000 + 2k disk I/Os

Pipelining Versus Materialization - V

- Example: $[R(w,x) \bowtie S(x,y)] \bowtie T(y,z)$
  - B(R) = 5000, B(S) = 10.000, B(T) = 15.000, M = 151
  - if $50 < B(R \bowtie S) = k \leq 7500$
    - partition T into 50 buckets of 300 blocks
    - perform $R \bowtie S$, but use the 50 free blocks to make 50 buckets of the result – write to disk
    - join result from $R \bowtie S$ stored in 50 buckets with the 50 buckets from T (read bucket from $R \bowtie S$ result into 150 blocks, use 1 reminder for T-buckets)
    - cost:
      - partition T: 30.000
      - $R \bowtie S$: 45.000
      - write result $R \bowtie S$ to disk: k
      - join buckets from T and from result from $R \bowtie S$: 15.000 + k
      - total $R \bowtie S \bowtie T$ cost: 90.000 + 2k disk I/Os
    - using materialization – write intermediate result back to disk and read again
      - total $R \bowtie S \bowtie T$ cost: 90.000 + 2k disk I/Os if storing buckets from $R \bowtie S$ (if not, add another 2k for partitioning)

Pipelining Versus Materialization - V

- Example: $[R(w,x) \bowtie S(x,y)] \bowtie T(y,z)$
  - B(R) = 5000, B(S) = 10.000, B(T) = 15.000, M = 151
  - if $7500 < B(R \bowtie S) = k$
    - cannot perform join on T with result from $R \bowtie S$ in two passes, because each of the 50 buckets from $R \bowtie S$ will be larger than 150 blocks
      → can add another pass – add two accesses for each block 2 * (15.000 + k)
      → 120.000 + 4k disk I/Os using pipelining
    - try materialization
    - compute $R \bowtie S$ using two pass hash-join: 45.000
    - write result to disk: k
    - join T with result from $R \bowtie S$ using another two-pass
      (T can still be partitioned into 150 buckets regardless of k: 3 * (15.000 + k)
    - total $R \bowtie S \bowtie T$ cost: 90.000 + 4k disk I/Os using materialization



Summary

- ✓ Parsing
- ✓ Logical query plans (LQP) in relational algebra
- ✓ Optimize LQP using algebraic laws
- ✓ Estimate size of a intermediate relation
- ✓ Consider physical query plans