Data Management

Maurizio Lenzerini

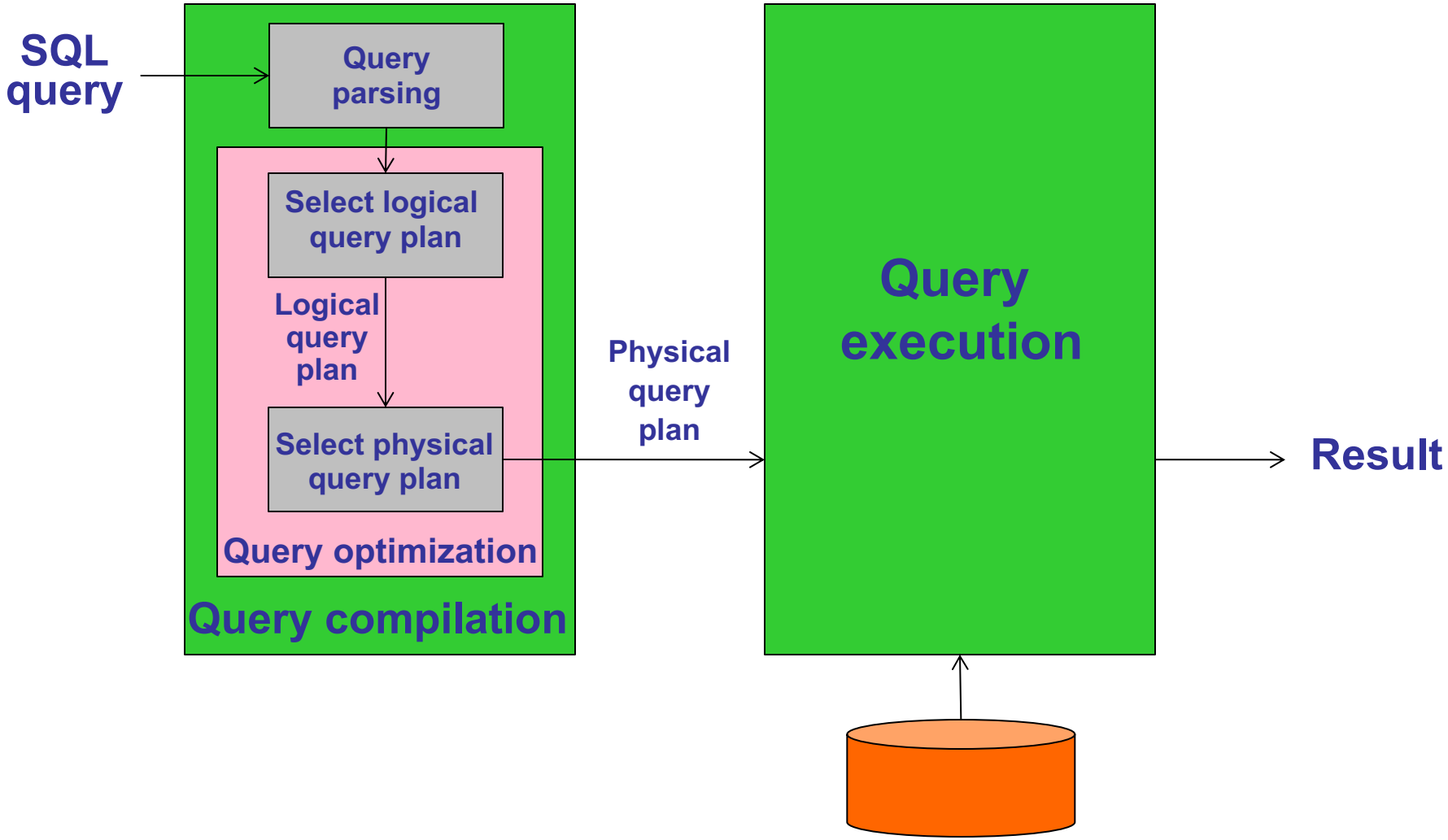
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Università di Roma "La Sapienza"

Academic Year 2018/2019

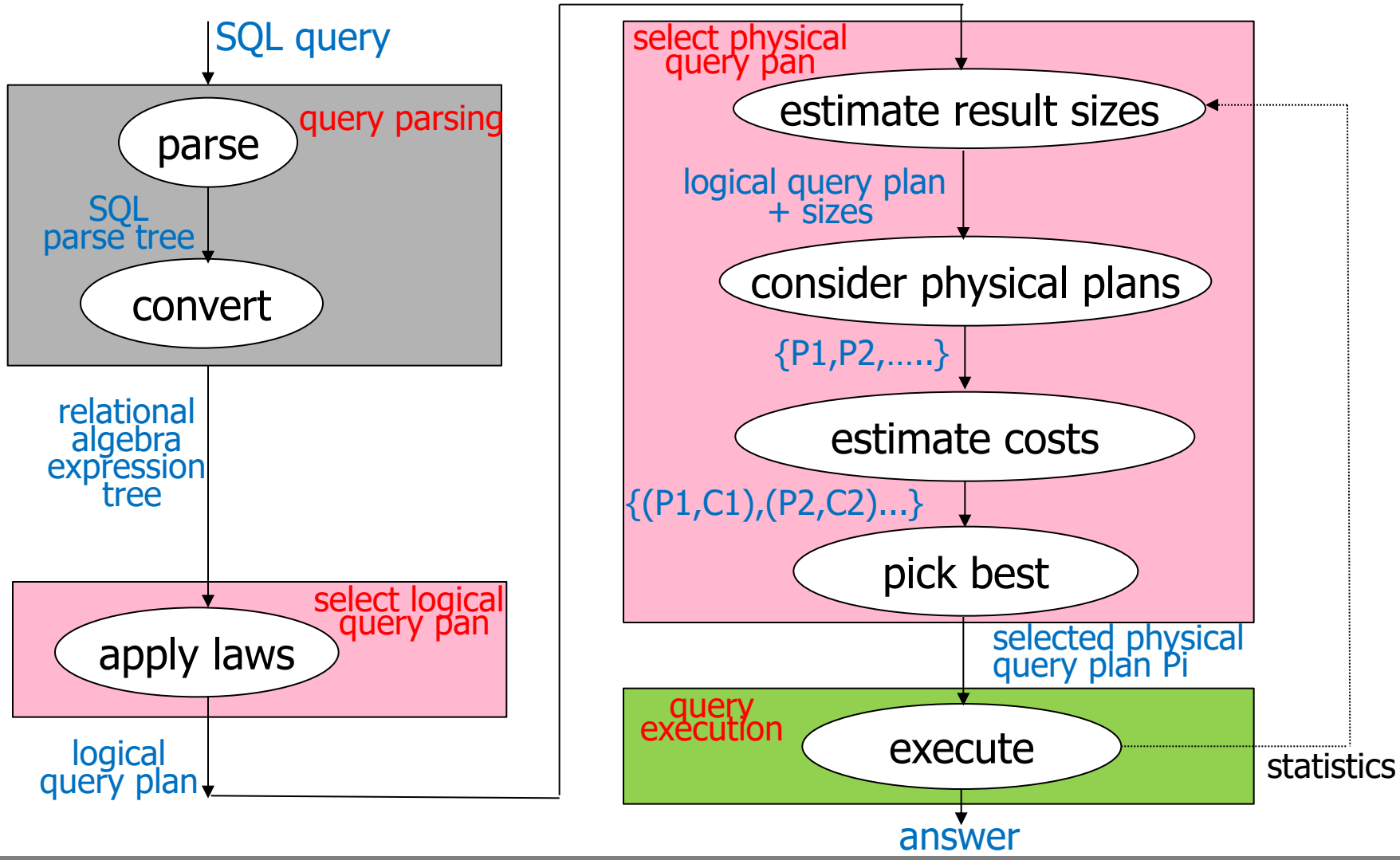
Part 7
Query processing – query compilation and optimization

http://www.dis.uniroma1.it/~lenzerin/index.html/?q=node/53

Query processing by the SQL engine



Query processing by the SQL engine



7. Query processing – query compilation and optimization

7.1 Query parsing
7.2 Selecting logical query plan
7.3 Selecting physical query plan

7. Query processing – query compilation and optimization

7.1 Query parsing
7.2 Selecting logical query plan
7.3 Selecting physical query plan

Query parsing

Query parsing is done through the following phases:

1. Parse
2. Convert

The "parse" phase is done through the following steps:

1.The SQL query is analyzed and represented as a parse tree

2.The parse tree is pre-processed with the goal of performing the following actions/cheking:

1. all views are substituted by their definition
2. every element of the query should be valid elements of the schema
3. resolving all ambiguities of attributes, if possible
4. type checking

If the parse tree is valid, then we turn to the "convert" phase, otherwise an error is issued.

Query parsing

The "convert" phase transforms an SQL parse tree into an extended relational algebra expression tree, using the following operators

- Union, intersection and difference

  on both sets and bags (where bags are like sets, but with duplicates); we use the subscript S to indicate that the operator that works on sets and produce a set. In the case where it works on sets or bags and produce bags, we use the subscript B
- Selection and projection (indicated as σ, π) – we assume that renaming is done with projection on bags
- Cartesian product and join (with all the variants of join – natural join indicated as $\bowtie$, theta-join indicated with $\bowtie_C$) on bags
- Duplicate elimination (turning a bag into a set, indicated as δ)
- Grouping corresponding to GROUP-BY and the aggregation operators (SUM, AVERAGE, etc.), indicated as γ
- Sorting (corresponding to ORDER-BY)

The "convert" phase

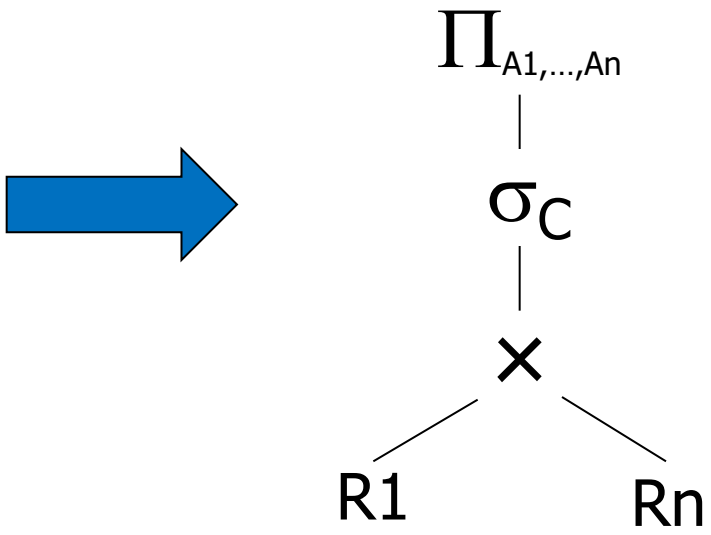
- An extended relational algebra expression tree is a tree where non-leaf nodes are labelled by operators of the extended relational algebra, and leaves are database relations

- A query of the form

  select A1,…, An
  from R1, ... , Rm
  where C

  is transformed into

$$\pi_{A1,\ldots,An}\left(\sigma_C\ (R1\ \ldots\ Rm)\right)$$



- A query having subqueries is transformed in a more complex way:

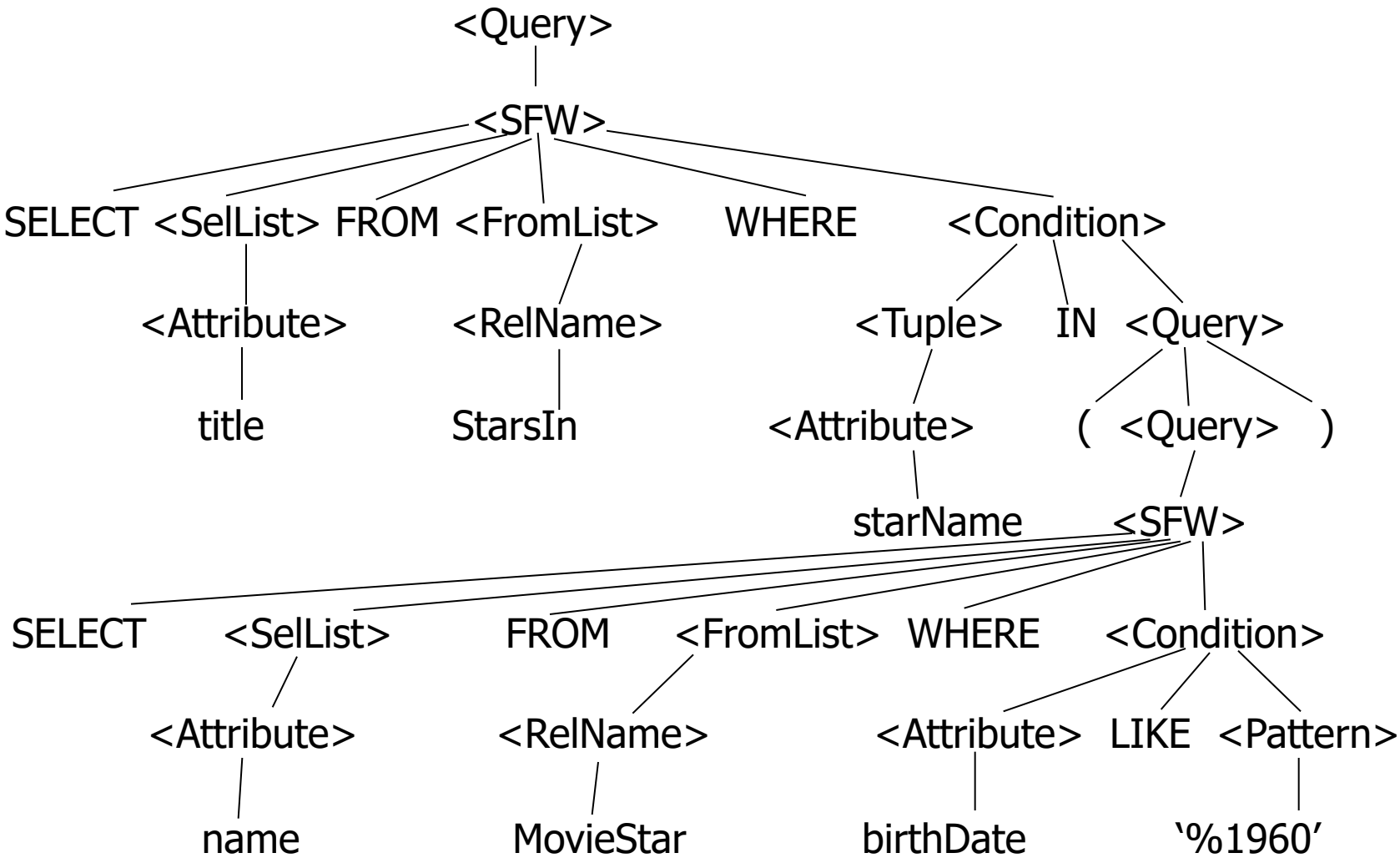
Query parsing: example

Consider the SQL query:

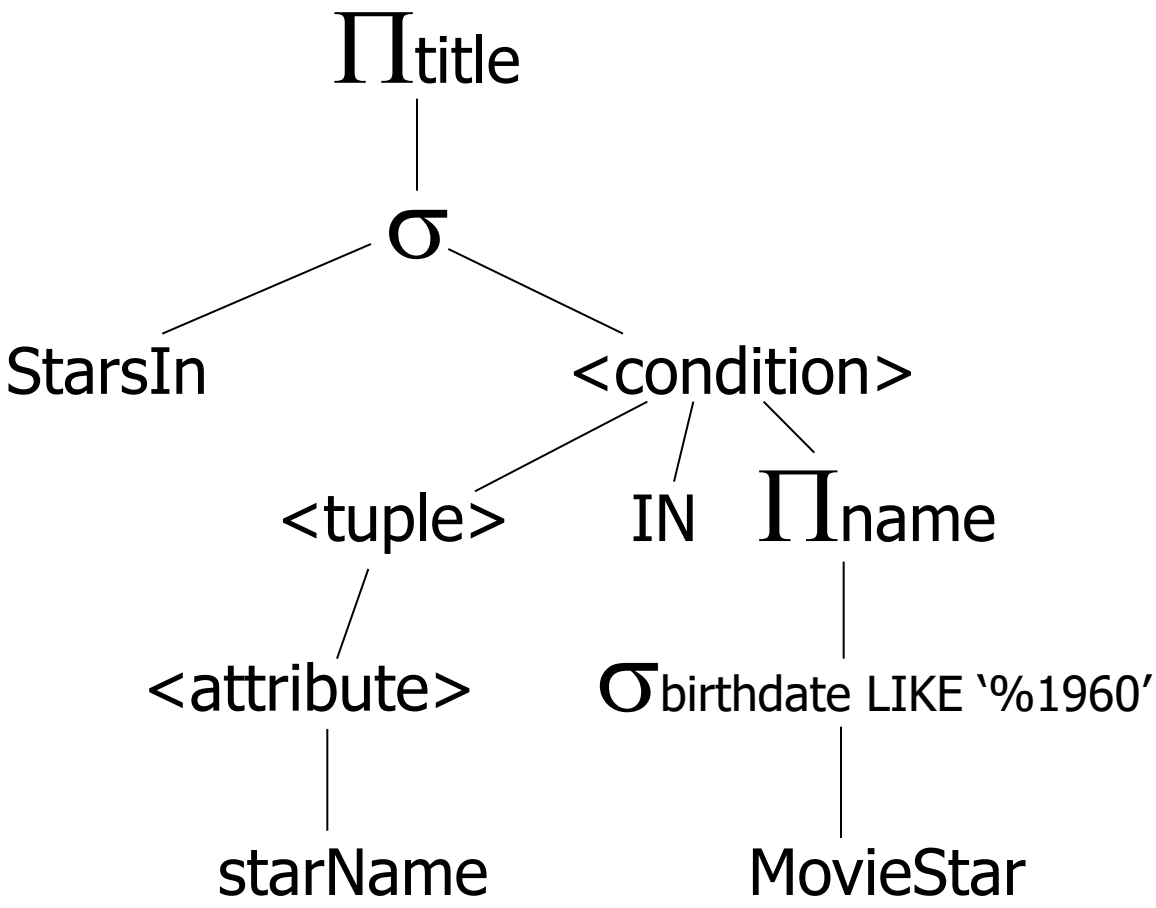
```
SELECT title
FROM StarsIn
WHERE starName IN (SELECT name
                   FROM MovieStar
                   WHERE birthdate LIKE '%1960')
```

(Find the movies with stars born in 1960)

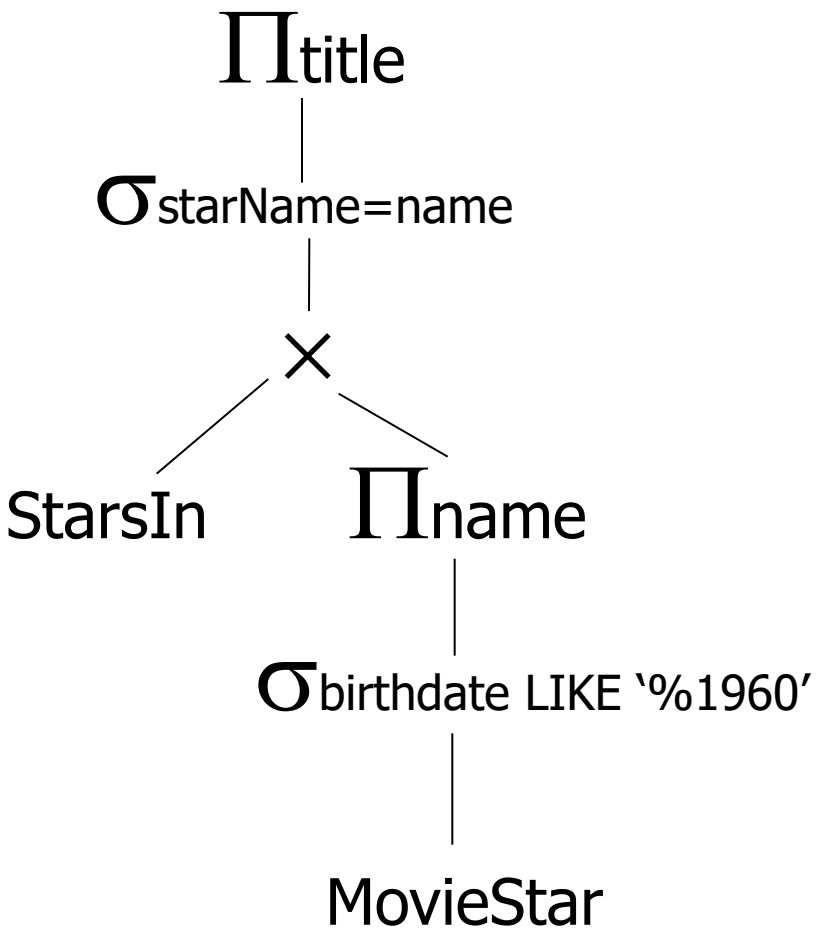
Parse Tree: example



Example: modified Relational Algebra



Example: Relational expression tree



7. Query processing – query compilation and optimization

7.1 Query parsing
7.2 Selecting logical query plan
7.3 Selecting physical query plan

Apply relational algebra rules

- Relational algebra rules are “equivalence-preserving” transformation rules

- We should understand which are the rules, but also what are the rules that are useful for transforming the query so as to make query evaluation more efficient

Natural joins, product, union, intersection

$R \bowtie S = S \bowtie R$

$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

$R \times S = S \times R$

$(R \times S) \times T = R \times (S \times T)$

$R \cup S = S \cup R$

$R \cup (S \cup T) = (R \cup S) \cup T$

$R \cap S = S \cap R$

$R \cap (S \cap T) = (R \cap S) \cap T$

commutative and associative laws for natural join, union, and intersection both for sets and bags

Be careful with bags

$R \cap (S \cup T) = (R \cap S) \cup (R \cap S)$

The above distributive law is valid for sets, but not for bags!

Proof: Consider R=S=T= {a}. Then

$R \cap_B (S \cup_B T) = \{a\}$, but

$(R \cap_B S) \cup_B (R \cap_B T) = \{a, a\}$

Selection

$$\sigma_{p1}\ [\ \sigma_{p2}\ (R)] = \sigma_{p2}\ [\ \sigma_{p1}\ (R)]$$

Note that one part of a complex condition, involving fewer attributes than the whole condition, may be moved to a convenient place that the entire condition cannot go. This means that sometimes it could be a good idea to break complex conditions into its parts.

Assume R is a set (not a bag):

$$\sigma_{p1 \wedge p2}(R) = \sigma_{p1}\ [\ \sigma_{p2}\ (R)] = \sigma_{p2}\ [\ \sigma_{p1}\ (R)]$$

$$\sigma_{p1\ v\ p2}(R) = [\ \sigma_{p1}\ (R)]\ \cup_s\ [\ \sigma_{p2}\ (R)]$$

Difference between sets and bags

Is the following rule correct, if R is a bag?

$$\sigma_{p1 \vee p2}(R) = [\sigma_{p1}(R)] \cup_B [\sigma_{p2}(R)]$$

Difference between sets and bags

Is the following rule correct, if R is a bag?

$$\sigma_{p1 \vee p2}(R) = [\sigma_{p1}(R)] \cup_B [\sigma_{p2}(R)]$$

Clearly NO! If a tuple satisfies both p1 and p2, it appears twice in the expression on the right, but only once in the expression on the left.

Projection

Let: X = set of attributes

Y = set of attributes

XY = X U Y

Is this rule correct?

$\pi_{xy}(R) = \pi_x[\pi_y(R)]$

Projection

Let: X = set of attributes

Y = set of attributes

XY = X U Y

Is this rule correct?

$\pi_{xy}(R) = \pi_x[\pi_y(R)]$

Clearly, NO, in particular if x is not a subset of y

Projection

Let: X = set of attributes

Y = set of attributes

X subset of Y

This rule is correct:

$$\pi_X [\pi_Y (R)] = \pi_X (R)$$

Projection

Let: X = the whole set of attributes of R

$\pi_X(R) = R$

This is obviously correct!

Rules: σ + $\bowtie$ combined

Let p = predicate with only R attributes

q = predicate with only S attributes

$$\sigma_p(R \bowtie S) = [\sigma_p(R)] \bowtie S$$

$$\sigma_q(R \bowtie S) = R \bowtie [\sigma_q(S)]$$

$$\sigma_q(R \bowtie S) = \sigma q(R) \bowtie \sigma q(S)$$

if q applies to both relations

Rules: π,σ combined

Let x = subset of R attributes

z = attributes in predicate P(subset of R attributes)

$$\pi_x[\sigma_{p\,(R)}] =$$

Rules: π,σ combined

Let x = subset of R attributes

z = attributes in predicate P (subset of R attributes)

$$\pi_x[\sigma_p(R)] = \{\sigma_p[\pi_x(R)]\} \text{ if z subset of x}$$

$$\pi_x[\sigma_p(R)] = \pi_x\{\sigma_p[\pi_{xz}(R)]\} \text{ otherwise}$$

Rules: π, $\bowtie$ combined

Let x = subset of R attributes

y = subset of S attributes

z = intersection of R,S attributes

$$\pi_{xy}(R \bowtie S) = \pi_{xy}\{[\pi_{xz}(R)] \bowtie [\pi_{yz}(S)]\}$$

Rules: π, σ, $\bowtie$ combined

Let x = subset of R attributes

y = subset of S attributes

z = intersection of R,S attributes

$$\pi_{xy}\{\sigma_{p\ (R \bowtie S)}\} =$$

$$\pi_{xy}\{\sigma_{p\ [\pi_{xz'}(R) \bowtie \pi_{yz'}(S)]}\}$$

where $z' = z \cup \{\text{attributes used in } p\}$

Rules for σ, π combined with $\times$

Let x = subset of R attributes
y = subset of S attributes
z = intersection of R,S attributes

similarly...

$$\pi_{xy}\{\sigma_p(R \times S)\} =$$

$$\pi_{xy}\{\sigma_p[\pi_x(R) \times \pi_y(S)]\}$$

Rules σ, ∪ combined

$$\sigma_p(R \cup S) = \sigma_p(R) \cup \sigma_p(S)$$

with union, if we push selection, then it must be pushed to both arguments

$$\sigma_p(R - S) = \sigma_p(R) - S = \sigma_p(R) - \sigma_p(S)$$

with difference, if we push selection, then it may be pushed either to the first argument or to both arguments

Rules combining joins and products

$$R \bowtie_C S = \sigma_C(R \times S)$$

$$R \bowtie S = \pi_L(\sigma_E(R \times S))$$

where E is the condition that equates each pair of attributes from R and S with the same name, and L is a list of attributes that contains the union of the attributes of R and the attributes of S.

Rules for duplicate elimination

δ denotes the duplicate elimination operator:

$$\delta(R \times S) = \delta(R) \times \delta(S)$$

$$\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$$

$$\delta(R \bowtie_C S) = \delta(R) \bowtie_C \delta(S)$$

$$\delta(\sigma_C(R)) = \sigma_C(\delta(R))$$

Rules for duplicate elimination

$\delta(R \cap_B S) = \delta(R) \cap_B \delta(S)$

$\delta(R \cap_B S) = \delta(R) \cap_B S$

$\delta(R \cap_B S) = R \cap_B \delta(S)$

However, δ cannot be moved across $\cup_B$, $-_B$, or π in general. Also, commuting δ with $\cup_S$, $-_S$, $\cap_S$ makes no sense.

Rules combining other operators

Let p = predicate with only R attributes

q = predicate with only S attributes

$$\sigma_{p \wedge q}(R \bowtie S) = \sigma_{p}(R) \bowtie \sigma_{q}(S)$$

This is why this rule is correct:

$$\sigma_{p \wedge q}(R \bowtie S) = \sigma_{p}\left[\sigma_{q}(R \bowtie S)\right] =$$

$$\sigma_{p}\left[R \bowtie \sigma_{q}(S)\right] = \sigma_{p}(R) \bowtie \sigma_{q}(S)$$

Rules combining other operators

Let p = predicate with only R attributes

q = predicate with only S attributes

m = predicate with only R,S attributes

$$\sigma_{p \wedge q \wedge m}(R \bowtie S) = \sigma_m \left[\sigma_p(R) \bowtie \sigma_q(S) \right]$$

$$\sigma_{p \vee q}(R \bowtie S) = \left[(\sigma_p(R) \bowtie S \right] \cup \left[R \bowtie \sigma_q(S) \right]$$

We should be careful

It seems that doing projection early is always beneficial. However, pushing projection can make an index useless!

Example: R(A,B,C,D,E)

x={E}

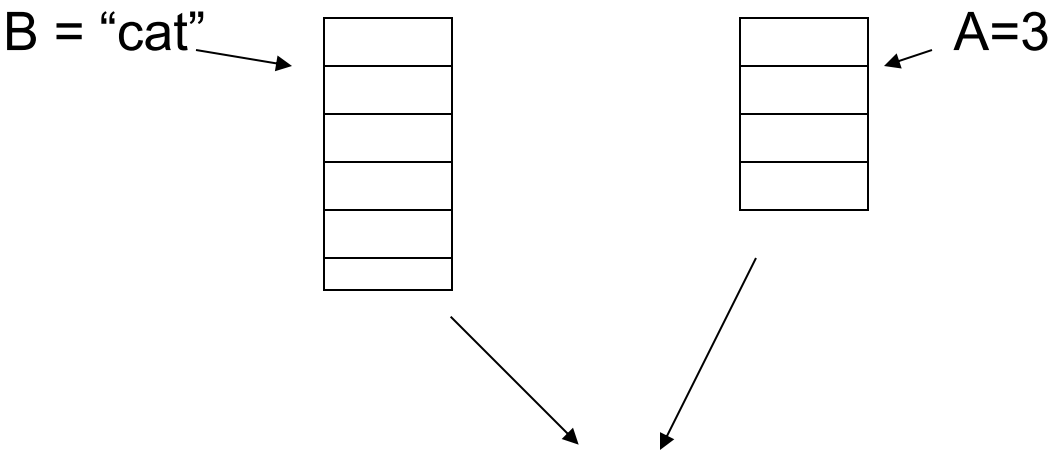
P: (A=3) $\wedge$ (B="cat")

$\pi_{x\ \{\sigma_{p(R)}\}}$ vs. $\pi_{E}\left\{\sigma_{p}\ \{\ \pi_{ABE(R)}\ \}\right\}$

It seems that the second expression is better.

We should be careful

But what if we have A,B indexes on R?



Intersect pointers to get pointers to matching tuples

In this case, the presence of the projection operator makes the index unusable.

Guidelines (heuristics)

1. Repeat if and until possible
   a) push selections over projections
   b) group the selections
   c) push selections over cartesian product
2. Eliminate useless projections by *(i)* $\pi_x(R) = R$, with x the attributes of R, or *(ii)* $\pi_x[\pi_y(R)] = \pi_x(R)$ with x subset of y
3. Push projections down the tree (in particular over cartesian products and over the selections that have not been already moved at step 1), or add new projections, with the proviso that care should be taken not to lose the possibility of using indexes
4. Try to remove duplicate eliminations (when applied to sets or grouping, for example), or move them to a more convenient position in the tree
5. Try to combine selections with product below so as to turn the two operations into an equijoin, which is generally more efficient to evaluate than are the two operations separately

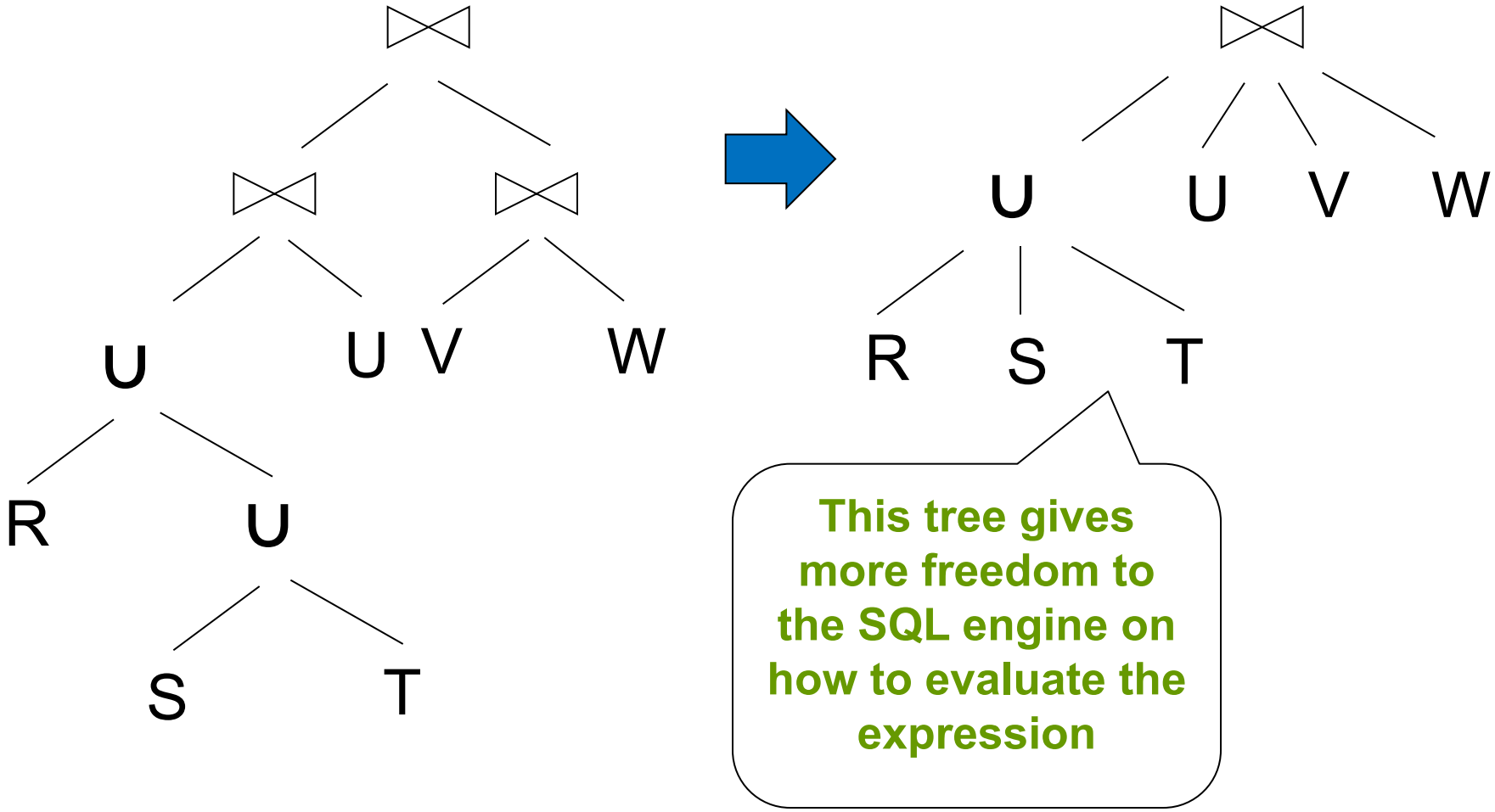
Guidelines for n-ary operators

Before producing the final logical query plan, we will perform a last step: for each portion of a subtree that consists of nodes with the same associative and commutative operator (i.e., natural join, union, and intersection), we group the nodes with these operators into a single node with many children. Such node will form a commutative and associative group, and the system can postpone the decision on the order to perform the various operations.

Natural joins, theta-joins and products can also be combined with each other under certain circumstances:

- if we replace natural join with equi-join (= on common attributes)
- if we add a projection to eliminate copies of attributes involved in a natural join that has become an equi-join
- The theta-join in question must be commutative (this is not always the case, e.g., $(R \underset{R.b>S.b}{\bowtie} S) \underset{a<d}{\bowtie} T$ not equivalent to $R \underset{R.b>S.b}{\bowtie} (S \underset{a<d}{\bowtie} T)$, where a is an attribute of R and d is an attribute of T.

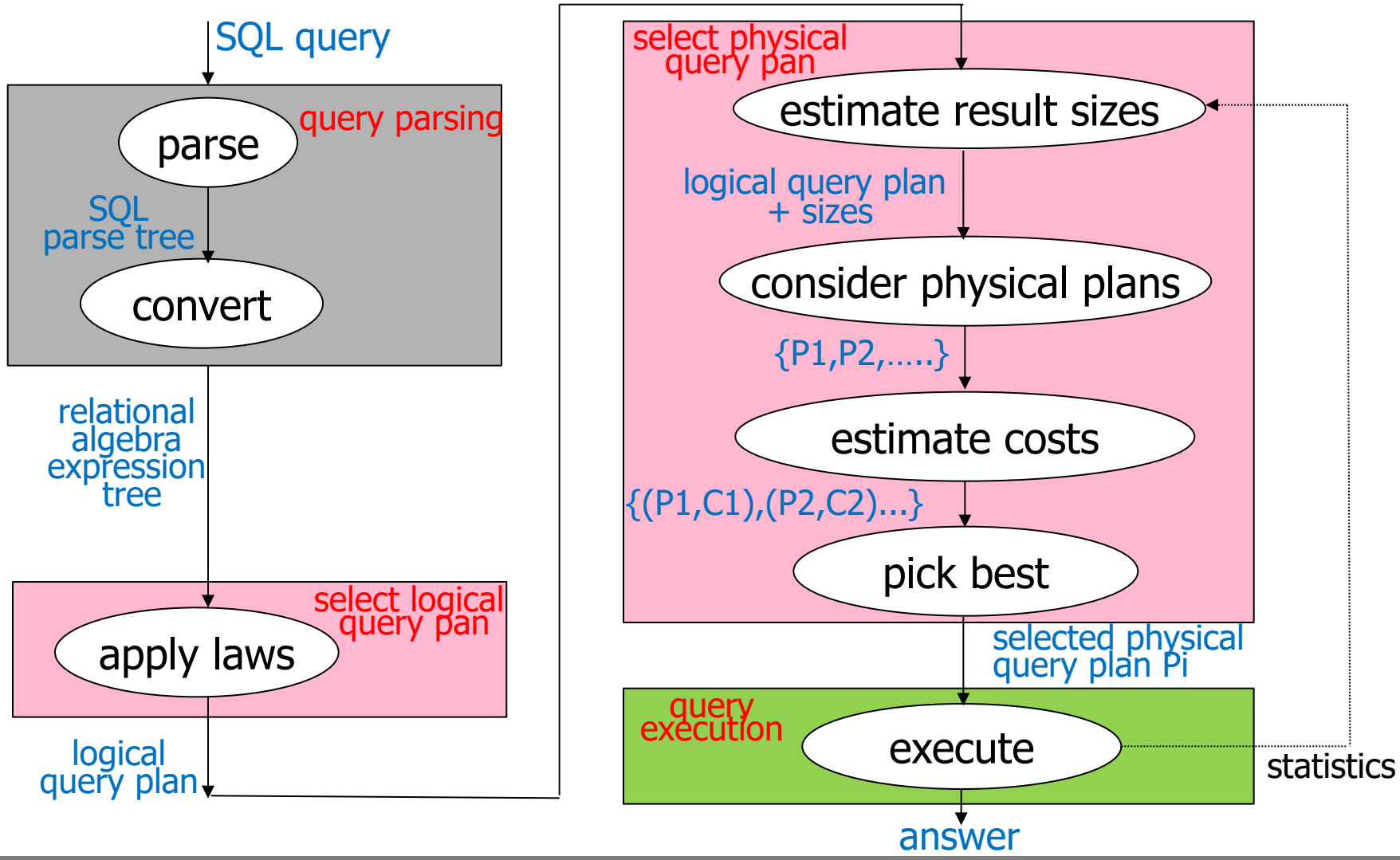
Grouping to n-ary operators



7. Query processing – query compilation and optimization

7.1 Query parsing
7.2 Selecting logical query plan
7.3 Selecting physical query plan

Query processing by the SQL engine



From logical to physical plans

When we work at the logical query plan level, we generally make transformations that are beneficial independently from the current statistics.

When we work at the physical query plan level, we aim at transformations that are effective with respect to the current volume of data we have.

That's why we need to have estimates of the size of the result of the execution of the operators.

Estimating cost of query plan

In order to estimate the size of the result of the execution of the operator, the DBMS keeps statistics for every relation R

- T(R) : # tuples in R
- S(R) : # of bytes in each R tuple
- B(R): # of pages to hold all R tuples
- V(R,A): # distinct values in R for attribute A

Estimating result size

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

A: 20 byte string
B: 4 byte integer
C: 8 byte date
D: 5 byte string

T(R) = 5

S(R) = 37

V(R,A) = 3

V(R,C) = 5

V(R,B) = 1

V(R,D) = 4

Size estimates for W = R1 x R2

T(W) = **T(R1) × T(R2)**

S(W) = **S(R1) + S(R2)**

Size estimates for projection

For the size estimate of a projection, we should remember that the number of tuples does not change with projection.

However, the size of each tuple may decrease, and therefore the number of pages needed to store the result of projection may decrease, and we should take this into account.

One way to take this into account is to measure the factor f by which the relation R shrinks with the projection, and consider the size of the projection to be $f \times T(R)$.

Size estimate for $W = \sigma_{A=a}(R)$

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

V(R,A)=3
V(R,B)=1
V(R,C)=5
V(R,D)=4

$W = \sigma_{z=val}(R)$ T(W) =

Size estimate for $W = \sigma_{A=a}(R)$

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

V(R,A)=3
V(R,B)=1
V(R,C)=5
V(R,D)=4

$W = \sigma_{z=val}(R)$ T(W) =

what is probability this tuple will be in the answer by assuming uniform distribution over V(R,A)?

Size estimate for $W = \sigma_{A=a}(R)$

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

V(R,A)=3
V(R,B)=1
V(R,C)=5
V(R,D)=4

$$W = \sigma_{z=val}(R) \quad T(W) = \frac{T(R)}{V(R,A)}$$

what is probability this tuple will be in the answer by assuming uniform distribution over V(R,A)?

What about $W=\sigma_{A \geq val}(R)$?

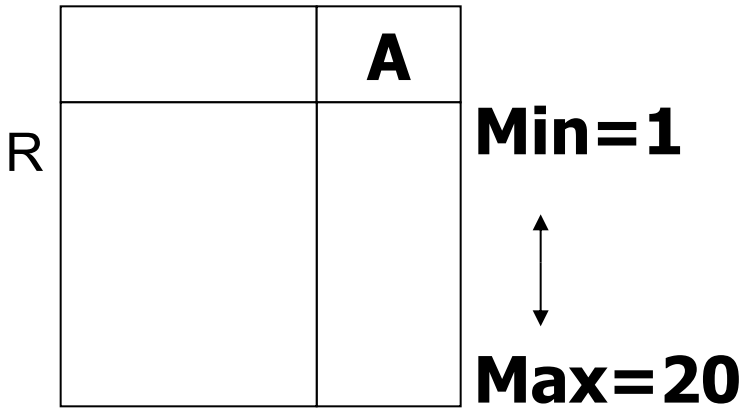
Inequality – approach # 1: one might think that in the average half of the tuples satisfies the condition. However, we follow the intuition that queries involving inequality tend to return a small fraction of the possible tuples

- **$W = \sigma_{A \geq val}(R)$** → $T(W) = T(R)/3$

What about W = $\sigma_{z \geq val}$ (R) ?

- **Inequality - approach # 2: Estimate values in range**

Example



V(R,A)=10

W= $\sigma_{A \geq 15}$ (R)

$$f = \frac{20-15+1}{20-1+1} = \frac{6}{20}$$ (fraction of range)

$$T(W) = f \times V(R,A) \times T(R)/V(R,A) = f \times T(R)$$

What about W = $\sigma_{A <> val}(R)$?

Disequality - approach # 1: we assume that satisfying an equality is rare!

- **W = $\sigma_{A <> val}$ (R)** → T(W) = T(R)

Disequality - approach # 2: T(R)/V(R,A) tuples fail to satisfy the condition

- **W = $\sigma_{A <> val}$ (R)** → $T(W) = T(R) - \frac{T(R)}{V(R,A)}$

Selection with complex conditions

When the selection condition C is the AND of several atomic conditions, we can treat the selection as a cascade of simple selections, each of which checks for one of the conditions.

The effect will be that the size estimate for the result is the size of the original relation multiplied by the selectivity factor for each condition: 1/3 for any inequality, 1 for <> (if we follow approach #1), and 1/V(R,a) (if we follow approach #1), for any attribute that is compared to a constant in the condition C

When the selection is an OR between two conditions, we can assume the sum of the number of tuples that satisfy each, but this is an overestimate. A more precise estimate is to take the smaller of the size of R and the sum of the number of the tuples that satisfy each.

Size estimate for W = R1 ⋈ R2

We will use two simplifying assumptions, that have been shown to be reasonable. Both assumptions are valid if the join is on a foreign key.

Assumption 1: Containment of value sets. If R and S are two relations with attribute Y in common, and V(R,Y) ≤ V(S,Y), then every Y-value of R will be a Y-value in S:

V(R1,A) ≤ V(R2,A) ⇒ Every A value in R1 is in R2

Assumption 2: Preservation of value sets. If we join a relation R with another relation, then an attribute A that is not a join attribute does not loose values from its set of possible values. That is: if A is an attribute in R but not in S, then V(R ⋈ S, A) = V(R,A).

Size estimate for W = R1 ⋈ R2

Let x = attributes of R1

y = attributes of R2

Case 1

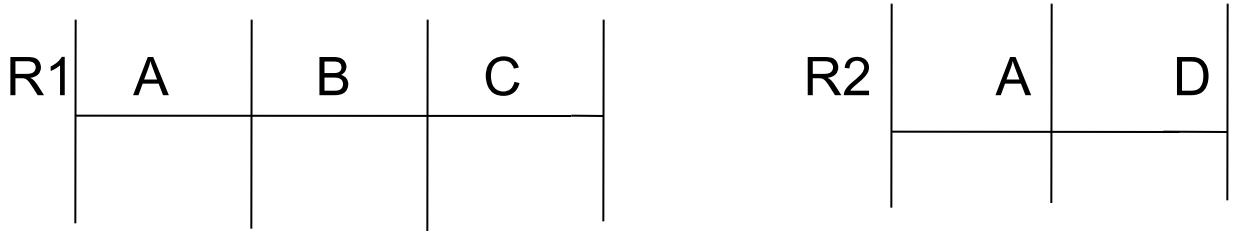
$$X \cap Y = \emptyset$$

Size estimate: the same as R1 x R2

Size estimate for W = R1 ⋈ R2

Case 2

W = R1 ⋈ R2 **X ∩ Y = A**



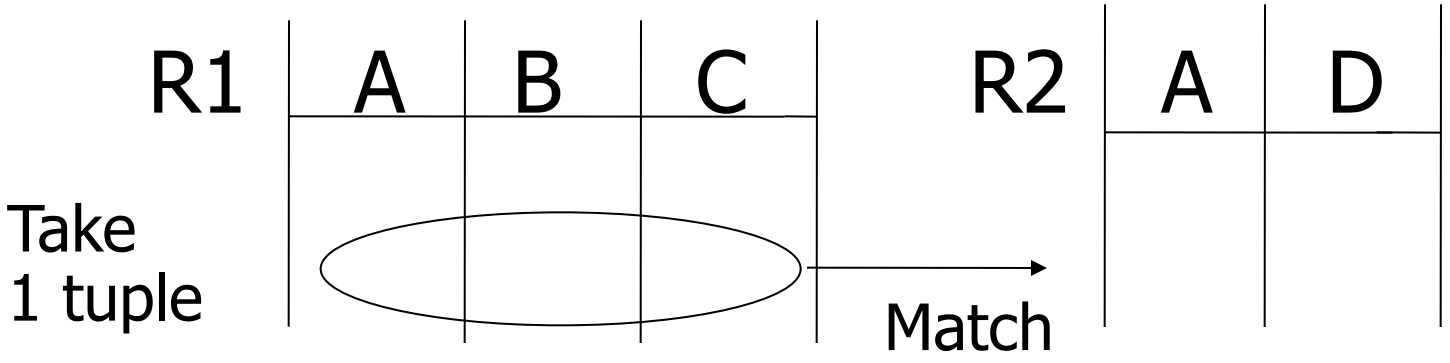
Remember assumption 1:

V(R1,A) ≤ V(R2,A) ⇒ Every A value in R1 is in R2

V(R2,A) ≤ V(R1,A) ⇒ Every A value in R2 is in R1

Size estimate for W = R1 ⋈ R2

Computing T(W) when V(R1,A) ≤ V(R2,A)



1 tuple matches with $\frac{T(R2)}{V(R2,A)}$ tuples...

so $T(W) = \frac{T(R2)}{V(R2,A)} \times T(R1)$

Size estimate for W = R1 ⋈ R2

- $V(R1,A) \leq V(R2,A)$ $\quad T(W) = \dfrac{T(R2) \times T(R1)}{V(R2,A)}$

- $V(R2,A) \leq V(R1,A)$ $\quad T(W) = \dfrac{T(R2) \times T(R1)}{V(R1,A)}$

A is the common attribute between R1 and R2

In general:

$$T(W) = \frac{T(R2) \times T(R1)}{\max\{ V(R1,A), V(R2,A) \}}$$

Size estimate for W = R1 ⋈ R2

W = R1 ⋈ R2, with X ∩ Y = A1,…,An but valid also for equijoin

Case 3

The estimate of the number of tuples of the join between R1 and R2 is computed by multiplying T(R1) and T(R2), and then dividing by the larger of V(R1,y) and V(R2,y), for each attribute y that is common to R1 and R2.

Example: Equijoin J between

R(a,b,c) and S(d,e,f) on b=d and c=e

T(R) = 1000, T(S) = 2000, V(R,b)=20, V(S,d)=50, V(R,c)=100, V(S,e)=60

larger value between b,d

T(J) = 1000 × 2000 / (50 × 100) = 400.

larger value between c,e

Size estimate for R1 ⋈ R2 ⋈ R3 ⋈ …

Start with the product of the number of tuples in each relation. Then, for each attribute A appearing at least twice, divide by all but the least of the V(R,A)'s.

We can also estimate the number of values that remain for attribute A after the join: by the preservation-of-value-sets assumption, it is the least of these V(R,A)'s.

Example: Consider the join W of R(a,b,c), S(b,c,d) and U(b,e) with

T(R)=1000, T(S)=2000, T(U)=5000

V(R,a)=100, V(R,b)=20, V(R,c)=160

V(S,b)=50, V(S,c)=100, V(S,d)=400

V(U,b)=200, V(U,e)=500

50 and 200 are the values of V(S,b),V(U,b), the two largest among the three values for b

T(W) = 1000 × 2000 × 5000 / (50 × 200 × 160) = 6250

160 is the largest among the two values for c

Size estimate for other types of join

- The equijoin can be treated as a natural join (as we saw before)

- Theta-joins can be treated as if they were a selection following a product, with the following additional observations:
  - An equality condition can be estimated using the method for natural join
  - An inequality comparison between two attributes (such as R.a < S.b) can be handled as for the inequality of the form R.a < constant. That is, we can assume a selectivity factor of 1/3. Similarly for disequality.

Size estimate for other operators

- For bag union, the size is simply the sum of the size of the two relations.
- For set union, a good estimate is the sum of the larger plus the half the smaller.
- For intersection, the result can have as few as 0 tuples or as many as the smaller of the two arguments. A good suggestion is to take the average of the extremes, which is half the smaller.
- For difference R - S, a good estimate is T(R) – ½ T(S).
- For duplicate elimination on relation R, a good estimate is the smaller between ½ T(R) and the product of all $V(R,a_i)$ (such product is the maximum number of distinct tuples that can exist in R)
- For grouping and aggregation on relation R, the number of tuples coincides with the number of groups. A reasonable estimate for such number is the smaller between ½ T(R) and the product of $V(R,a_i)$, for all grouping attributes $a_1,\ldots a_n$.

Summary

- Estimating size of results is an "art"
- Don't forget: statistics must be kept up to date…

Considering physical plans

In order to turn a logical query plan into a physical query plan, in general one can consider many different physical plans derived from the logical one and estimating the cost of each.

After this step of cost-based enumeration, one can then pick up the physical query plan with the least estimated cost.

Considering physical plans

How can we derive one of the many possible physical plans from a logical one? By selecting:

1. An order and grouping for associative-commutative operators, such as joins, unions, and intersections.
2. An algorithm for each operator in the logical plan (i.e., nested loop join, or the merge-sort join).
3. Additional operator (i.e., sorting, duplicate elimination) and/or transformations to be added to the logical plan
4. The way in which each argument is passed from one operator to the next, i.e., (1) storing the intermediate result in secondary storage, or (2) using iterators and passing an argument one main-memory buffer frame at a time.

Considering physical plans

Unfortunately, the number of physical query plans derivable from a single logical plan is enormous. Therefore, the DBMS applies **heuristics** for limiting the number of physical plans to be considered. We consider the following guidelines:

1. Use indexes for selection of the form A=c (or, A > c) on a stored relation, if available
2. If the selection involves one condition of the form A=c plus other conditions on a stored relation, then use the index, if available, and apply a further selection operator (new physical operator called **filter**)
3. If an argument R of the join has an index on the join attributes, then check whether it is advantageous to use an index-based join with R in the inner loop
4. If an argument R of the join is sorted on the join attribute, then prefer a join algorithm based on sorting rather than a hash-based one
5. When computing the union or the intersection of three or more relations, group the smaller relations first.
6. Apply specific algorithms for deciding join order (see later)

The order of joins

The item 6 above refers to an important aspect that has to be addressed explicitly: **deciding the order of joins**!

When we have two operands, we should remember that many of the join algorithms are asymmetric, in the sense that the roles played by the two argument relations are different, and the cost depends on which relation plays which role. For instance, the one-pass join reads one relation (called the **build relation**) in the buffer, and then it reads the other relation (called the **probe relation**) one page at a time. In this case it is obvious that we should choose the smaller relation as the build relation. More generally, when we have a join of two relations, we select the one whose estimated size is smaller as the left argument. This is a good choice in any case, in particular for one-pass, nested loop, and index-based join!

The order of joins

If the join is not binary, the problem is much more difficult. We consider a greedy algorithm for solving the problem, which makes one decision at a time about the order of join, and never backtracks or reconsiders decisions once made. We shall consider a greedy algorithm that selects a left-deep tree, where the greediness is based on the idea that we want to keep the intermediate relations as small as possible at each level of the tree. Note that the greedy algorithm does not guarantee to find the optimal solution!

Greedy algorithm (resulting in a left-deep join tree):

BASIS: Start with the pair of relations whose estimated join size is the smallest. The join of these relations becomes the **current tree**

INDUCTION: Find, among all relations not yet included in the current tree, the relation that, when joined with the current tree, yields the relation of smallest estimated size. The new current tree has the old current tree as its left argument, and the selected relation as its right argument.

The order of joins

Example

Consider the join of R(a,b), S(b,c), T(c,d), and U(d,a), each having 1000 tuples, and with

V(R,a)=100, V(R,b)=200

V(S,b)=100, V(S,c)=500

V(T,c)=20, V(T,d)=50

V(U,a)=50, V(U,d)=1000

In the basis step, we find out that the pair of relations with the smallest join is (T,U) with the size of $1000 \times 1000 / 1000 = 1000$. So, (T join U) is the current tree. Next, we compare the sizes of ((T join U) join R) and ((T join U) join S). The latter, with a size of $1000 \times 1000 / 500 = 2000$ is better than the former, with a size of $1000 \times 1000 / 100 = 10.000$. Thus, we choose (T join R) join S) as the new current tree. We then add R, and obtain the tuple-base cost of 1000+2000=3000 (measured as the sum of the sizes of the two intermediate relations).

Completing the physical query plan

There are still some steps for completing the physical query plan:

1. Selection of algorithms to implement the operations of the query plan that have not been decided yet.
2. Deciding when intermediate results will be materialized (stored in secondary storage) or pipelined (created in main memory and passed to other operations)
3. Building a comprehensive tree, using appropriate notation, to be passed to the query execution engine.

Item 1 has been already discussed extensively. We will deal with items 2 and 3 in the following.

Pipelining vs materialization

Materialization is an obvious approach: store the result of an operation in secondary storage for later usage.

Pipelining means that several operations are running at once, and the tuples produced by one operation are passed directly to the operation that uses it, without storing the intermediate tuples on secondary storage. Pipelining is implemented through a network of iterators, whose functions call each other at appropriate time.

Pipelining vs materialization

- **Unary operators** (selection and projection) are excellent candidates for pipelining. Consider the query

  select A from R where B=3

  corresponding to $\pi_A(\sigma_{B=3}(R))$. We will never store the result of the selection operator!

- **Binary operators** Pipelining means that we do not store the entire result in secondary storage (we may store some parts of it), and we store relevant data of the operation in the buffer. How many frames we use and how we produce the result depends on various factors.

Pipelining vs materialization

Consider the physical query plan for

(R(w,x) join S(x,y)) join U(y,z)

with

- 5.000 pages for R, 10.000 pages for S and U each
- k denoting the number of pages for the intermediate result of (R(w,x) join S(x,y))
- both joins implemented as hash-based join, either one pass (with a hash table in the buffer) or two-pass based on hashing, depending on k
- 102 buffer frames are available

We need a two-pass algorithm (note that $5.000 < 101 \times 100$) for (R join S), and in the first pass we can use the buffer so that each bucket for R has no more than 50 pages.

Pipelining vs materialization

CASE $k \leq 50$

We can pipeline the result of (R join S) into 50 frames, organize them as a hash table in main memory, and we have plenty of buffer frames left for reading U, thus executing the second join in one-pass. The total number of page accesses is:

- $3 \times (15.000) = 45.000$ for (R join S)
- 10.000 for reading U in the one-pass join

So, we spend 55.000 page accesses.

Pipelining vs materialization

CASE $k > 50$ and $k \leq 5000$

We can still pipeline the result of (R join S), i.e., avoid to store the entire result in secondary storage, but we need a different strategy.

- We first hash U (based on y) into 50 buckets of 200 pages each.
- We perform a two-pass hash-join of R and S as before, but as each tuple is generated, we place it in one of the 50 remaining frames that are used to form the 50 buckets for the subsequent join with U (the hash function in this case is based on y). These buckets are stored in secondary storage.
- Finally we join (R join S) and U bucket by bucket. Since $k \leq 5000$, each bucket of (R join S) is at most 100 pages, and therefore the join is feasible (each twin of buckets are joined in one pass).

The total number of page accesses is:

- 20.000 page accesses for building the buckets of U
- 45.000 for the two-pass join of R and S, and k for writing their buckets
- k + 10.000 for the final one-pass step.

The total cost is $75.000 + 2 \times k$

Pipelining vs materialization

CASE k > 5000

Now, after the first hash on U, we cannot use a two-pass join. In principle, we could use a three-pass join, but this would require an extra 2 accesses for each page of both arguments, which means 20.000 + 2×k more page accesses.

We can do better if we use the materialization approach.

1. We compute (R join S) in two-pass algorithm based on hashing and store the result in secondary storage.
2. We join (R join S) with U using also a two-pass algorithm based on hashing. We can do that by using U as the build relation, because $B(U) \leq 100^2$.

The total number of page accesses is:

- 45.000 for the two-pass join of R and S, and k for writing the corresponding buckets
- 3 × (10.000 + k) = 30.000 + 3×k for the second two-pass join
- The total cost is 75.000 + 4×k.

Notation for the physical plan tree

We use a tree, where

- we indicate the algorithm used for the operators in the nodes corresponding to the operators
- we indicate that a certain intermediate relation is materialized by a double line crossing the edge between the relation and its consumer. All other edges are pipelined
- in the leaves, we indicate how we access the relations of the database

Accessing the relations

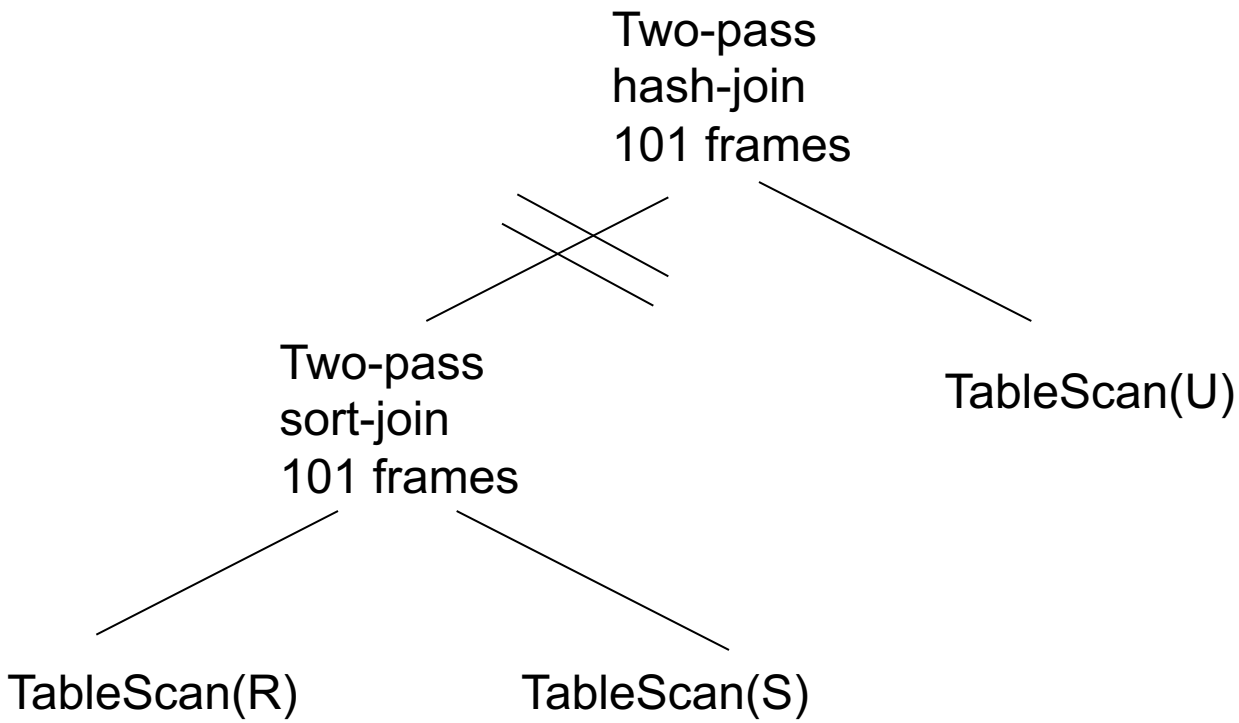
Each relation in the leaf of the logical query plan will be replaced by one of the following "scan" operator:

1. **TableScan(R)**: scan the relation R
2. **SortScan(R,L)**: R is scanned and sorted on the basis of attribute list L
3. **IndexScan(R,C)**: R is accessed through an index on attribute A, where A is the attribute mentioned in the condition C (the index must conform to the condition)
4. **IndexScan(R,A)**: the entire relation R is retrieved via an index on attributes A (index-only scan)

Physical operators

- For selection, we can use **Filter(C)**
- For sorting intermediate relations, we use **Sort(L)**, where L is a list of attributes
- For other operations, we indicate the algorithm used (for example, one-pass hash-join), and the number of buffer frames expected.

Example


Two-pass
hash-join
101 frames
Two-pass
sort-join
101 frames
TableScan(U)
TableScan(R)
TableScan(S)