Database Systems

Query Execution and Optimization

Katja Hose

Department of Computer Science
Aalborg University
khose@cs.aau.dk

Spring 2020

Learning goals

Learning goals

- Understand how selection statements are executed
- Understand the basic join algorithms
- Understand the basics of heuristic (logical) query optimization
- Understand the basics of physical query optimization

Motivation

- Understanding the basics of query processing and query optimization are fundamental to database tuning

Outline

Evaluation of an SQL statement

The clauses are specified in the following order.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

Evaluation of an SQL statement

The clauses are specified in the following order.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

But the query is evaluated in a different order

Evaluation of an SQL statement

The clauses are specified in the following order.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

But the query is evaluated in a different order

- Cartesian product of tables in the from clause

Evaluation of an SQL statement

The clauses are specified in the following order.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

But the query is evaluated in a different order

- Cartesian product of tables in the from clause
- Predicates in the where clause

Evaluation of an SQL statement

The clauses are specified in the following order.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

But the query is evaluated in a different order

- Cartesian product of tables in the from clause
- Predicates in the where clause
- Grouped according to the group by clause

Evaluation of an SQL statement

The clauses are specified in the following order.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

But the query is evaluated in a different order

- Cartesian product of tables in the from clause
- Predicates in the where clause
- Grouped according to the group by clause
- Predicate in the having clause applied to (eliminate) groups

Evaluation of an SQL statement

The clauses are specified in the following order.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

But the query is evaluated in a different order

- Cartesian product of tables in the from clause
- Predicates in the where clause
- Grouped according to the group by clause
- Predicate in the having clause applied to (eliminate) groups
- Compute aggregation functions for each remaining group

Evaluation of an SQL statement

The clauses are specified in the following order.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

But the query is evaluated in a different order

- Cartesian product of tables in the from clause
- Predicates in the where clause
- Grouped according to the group by clause
- Predicate in the having clause applied to (eliminate) groups
- Compute aggregation functions for each remaining group
- Projection on columns enumerated in the select clause

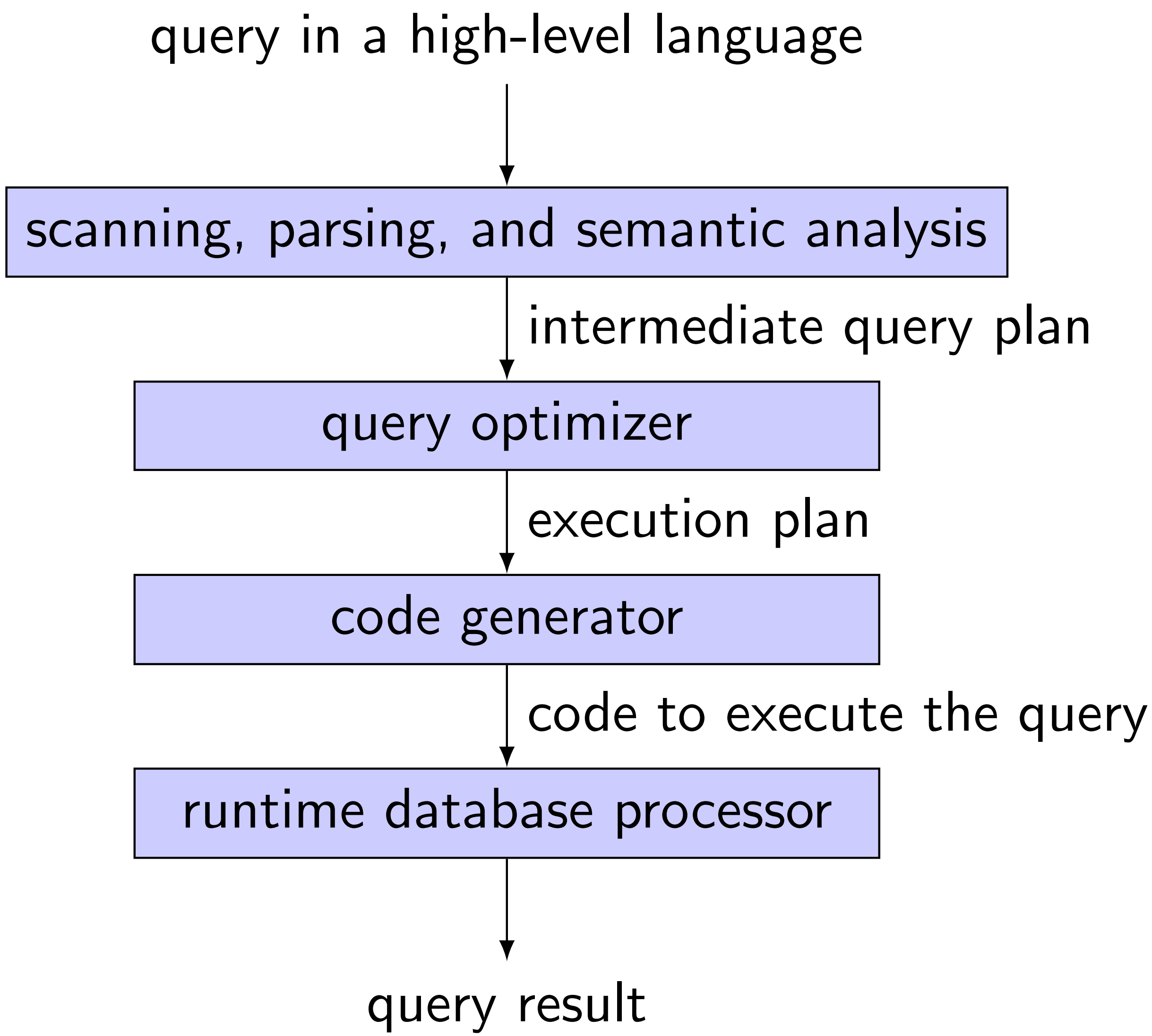
Evaluation of an SQL statement

The clauses are specified in the following order.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

SQL is declarative!

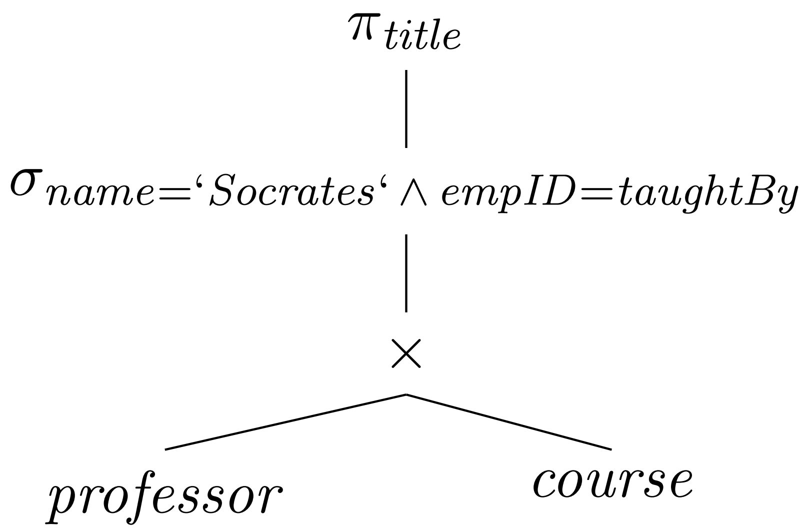
Steps of query processing



Parsing a query into an initial query plan

SELECT title
FROM professor, course
WHERE name=‘Socrates‘ AND
empID = taughtBy;

$\Rightarrow$

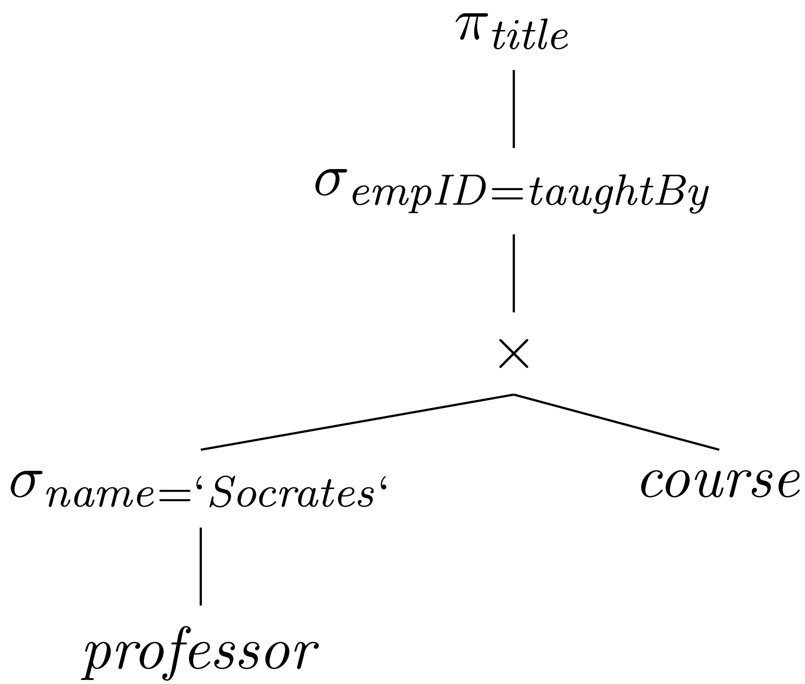


$$\pi_{title}(\sigma_{name=\text{‘}Socrates\text{‘} \wedge empID=taughtBy}(professor \times course))$$

Alternative query plan

SELECT title
FROM professor, course
WHERE name=‘Socrates‘ AND
empID = taughtBy;

$\Rightarrow$



$$\pi_{title}(\sigma_{empID=taughtBy}(\sigma_{name=`Socrates`} professor \times course))$$

Query optimization

Alternatives

- Equivalent query execution plans
- Algorithms to compute an algebra operation
- Methods to access relations (indexes)

Although the result is equivalent, execution costs might be different.

Query optimization

Alternatives

- Equivalent query execution plans
- Algorithms to compute an algebra operation
- Methods to access relations (indexes)

Although the result is equivalent, execution costs might be different.

Theory meets reality

It is not the task of the user to write queries “efficiently”, it is the task of the query optimizer to execute them efficiently!

Query optimization

Alternatives

- Equivalent query execution plans
- Algorithms to compute an algebra operation
- Methods to access relations (indexes)

Although the result is equivalent, execution costs might be different.

Theory meets reality

It is not the task of the user to write queries "efficiently", it is the task of the query optimizer to execute them efficiently!

But in reality... optimizers are not perfect.

Query costs

Measures

- Total elapsed time for answering a query (**response time**)
- Many factors contribute to response time
  - Disk access
  - CPU costs
  - network communication
  - query load
  - parallel processing
- Disk access most dominant
  - Block access time: seek time, rotation time
  - Transfer time

Query optimization

Logical query optimization

- Relational algebra
- Equivalence transformation
- Heuristic optimization

Physical query optimization

- Algorithms and implementations of operations
- Cost model

Outline

2. Heuristic (logical) query optimization
   - Equivalences in relational algebra
   - Phases of logical query optimization

Logical query optimization

Logical query optimization

- Foundation: algebraic equivalences
- Algebraic equivalences span the potential search space
- Given an initial algebraic expression:
  apply algebraic equivalences to derive new equivalent algebraic expressions

Logical query optimization

Logical query optimization

- Foundation: algebraic equivalences
- Algebraic equivalences span the potential search space
- Given an initial algebraic expression:
  apply algebraic equivalences to derive new equivalent algebraic expressions

What is a good plan?

- Difficult to compare plans without a cost function

$\Rightarrow$ logical query optimization relies on heuristics

Logical query optimization

Logical query optimization

- Foundation: algebraic equivalences
- Algebraic equivalences span the potential search space
- Given an initial algebraic expression:
  apply algebraic equivalences to derive new equivalent algebraic expressions

What is a good plan?

- Difficult to compare plans without a cost function

$\Rightarrow$ logical query optimization relies on heuristics

Main goal of logical query optimization

Reduce the size of intermediate results

Equivalences

Break up conjunctions in selection predicates

$$\sigma_{c_1 \wedge c_2 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$$

σ is commutative

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

π cascades

If $L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n$ then

$$\pi_{L_1}(\pi_{L_2}(\ldots(\pi_{L_n}(R))\ldots)) \equiv \pi_{L_1}(R)$$

Equivalences

Change the order of σ and π

If the selection involves only attributes $A_1, ..., A_n$ contained in the projection list, the order of σ and π can be changed

$$\pi_{A_1,...,A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1,...,A_n}(R))$$

$\cup, \cap$ and $\bowtie$ are commutative

$$R \bowtie_c S \equiv S \bowtie_c R$$

Equivalences

Change the order of σ and $\bowtie$

If the selection predicate c involves only attributes of relation R, the order of σ and $\bowtie$ can be changed

$$\sigma_c(R \bowtie_j S) \equiv \sigma_c(R) \bowtie_j S$$

If the selection predicate c is a conjunction of the form $c_1 \wedge c_2$ and c_1 involves only attributes in R and c_2 involves only attributes in S, then we need to split c

$$\sigma_c(R \bowtie_j S) \equiv \sigma_{c_1}(R) \bowtie_j \sigma_{c_2}(S)$$

Equivalences

Change the order of π and $\bowtie$

Given the projection list $L = \{A_1, ..., A_n, B_1, ..., B_m\}$ where A_i represents attributes in R and B_i attributes in S.

If the join predicate c only references attributes in L the following reformulation holds

$$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1,...,A_n}(R)) \bowtie_c (\pi_{B_1,...,B_m}(S))$$

Equivalences

$\bowtie, \cap, \cup$ (in separate) are all associative.
I.e., with Φ representing either of these operations, the following holds

$$(R \ \Phi \ S) \ \Phi \ T \ \equiv R \ \Phi \ (S \ \Phi \ T)$$

σ is distributive with $\cap, \cup, -$.
I.e., with Φ representing either of these operations, the following holds

$$\sigma_c(R \ \Phi \ S) \equiv (\sigma_c(R)) \ \Phi \ (\sigma_c(S))$$

π is distributive with $\cup$

$$\pi_c(R \cup S) \equiv (\pi_c(R)) \cup (\pi_c(S))$$

Equivalences

Join and/or selection predicates can be reformulated based on De Morgan's laws

$$\neg(c_1 \wedge c_2) \equiv (\neg c_1) \vee (\neg c_2)$$

$$\neg(c_1 \vee c_2) \equiv (\neg c_1) \wedge (\neg c_2)$$

Combination of Cartesian product and selection

A Cartesian product followed by a selection whose predicate involves predicates of both involved operands can be combined to a join

$$\sigma_\theta(R \times S) \equiv R \bowtie_\theta S$$

Equivalences

Join and/or selection predicates can be reformulated based on De Morgan's laws

$$\neg(c_1 \land c_2) \equiv (\neg c_1) \lor (\neg c_2)$$

$$\neg(c_1 \lor c_2) \equiv (\neg c_1) \land (\neg c_2)$$

Combination of Cartesian product and selection

A Cartesian product followed by a selection whose predicate involves predicates of both involved operands can be combined to a join

$$\sigma_\theta(R \times S) \equiv R \bowtie_\theta S$$

Remember the equivalent expressions for operators in relational algebra!

Outline

2. Heuristic (logical) query optimization
   - Equivalences in relational algebra
   - Phases of logical query optimization

Phases of logical query optimization

1. Break up conjunctive selection predicates
2. Push selections down
3. Introduce joins by combining selections and cross products
4. Determine join order
   Heuristic: execute joins with input from selections before executing other joins
5. Introduce and push down projections

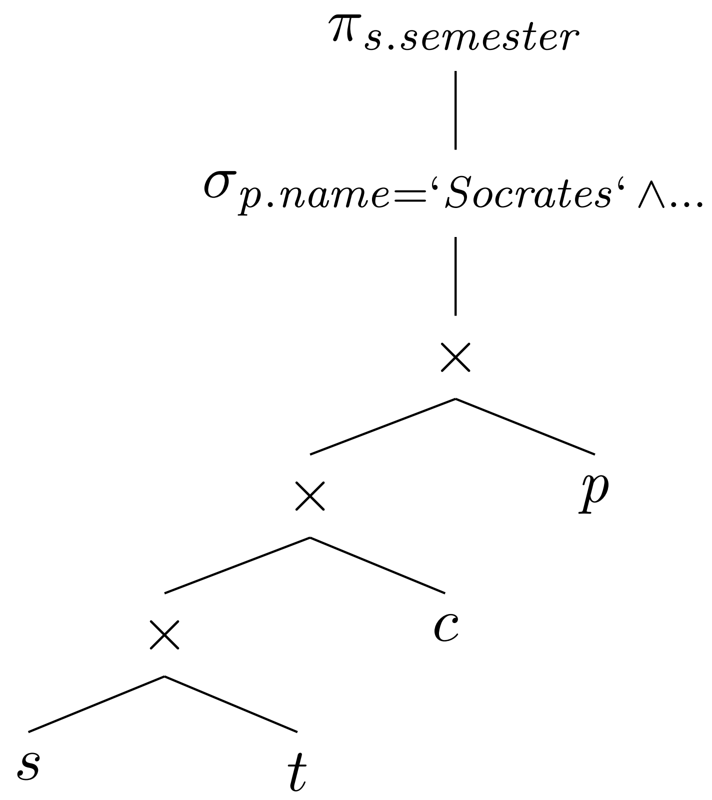
Phases of logical query optimization

1. Break up conjunctive selection predicates
2. Push selections down
3. Introduce joins by combining selections and cross products
4. Determine join order
   Heuristic: execute joins with input from selections before executing other joins
5. Introduce and push down projections
   Not always useful

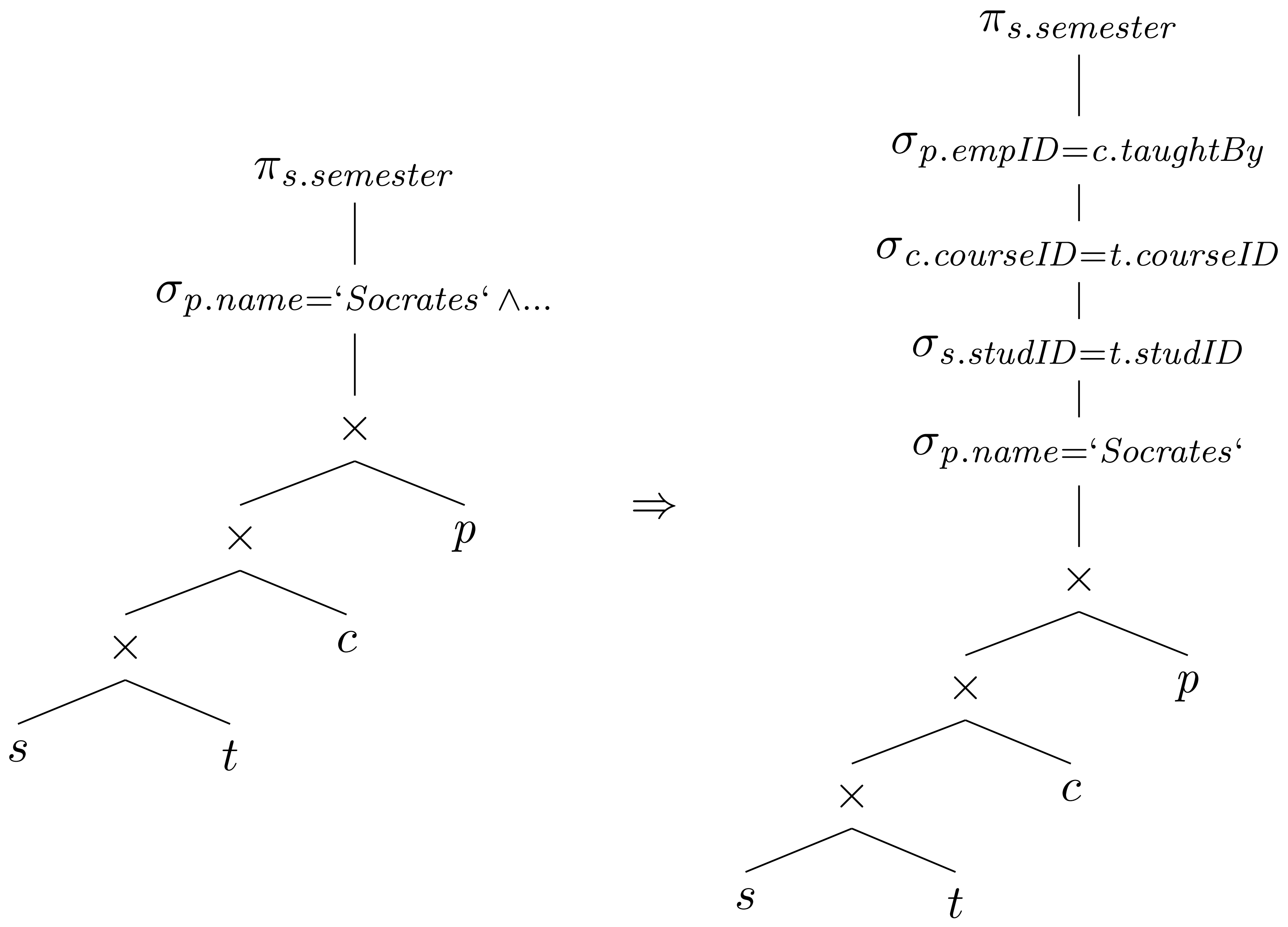
Example

SELECT DISTINCT s.semester
FROM student s, takes t,
course c, professor p
WHERE p.name=‘Socrates‘ AND
c.taughtBy = p.empID AND
c.courseID = t.courseID AND
t.studID = s.studID;

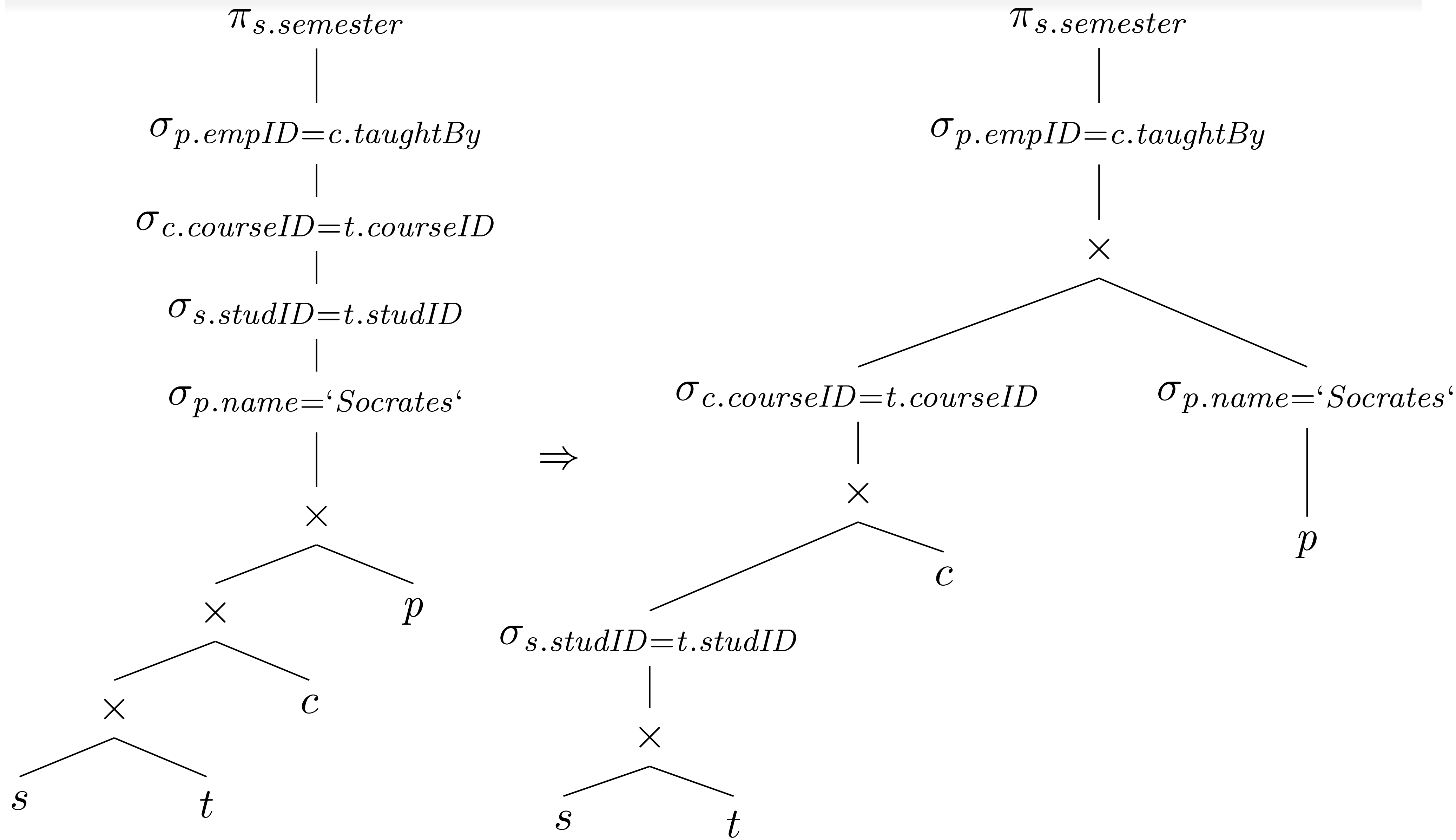
$\Rightarrow$



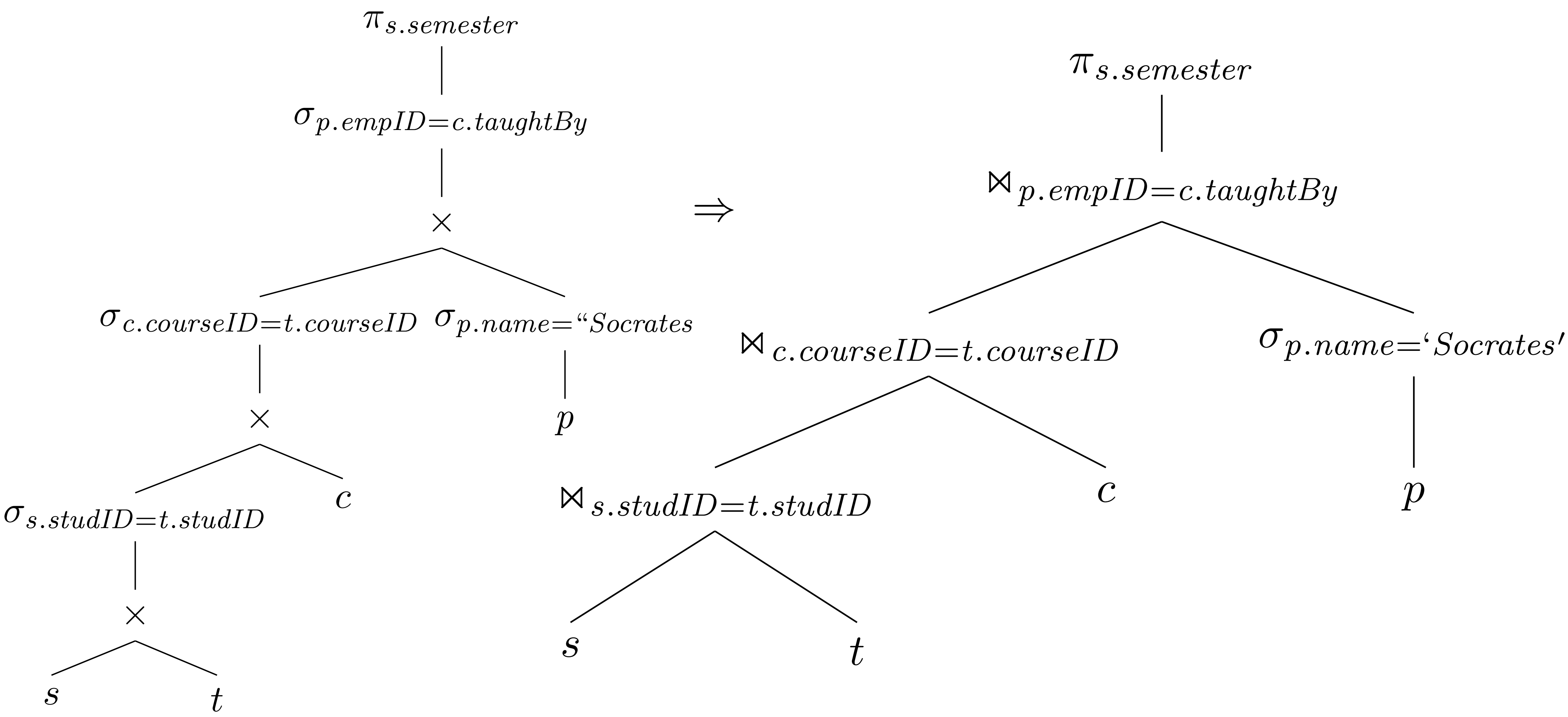
Break up selections



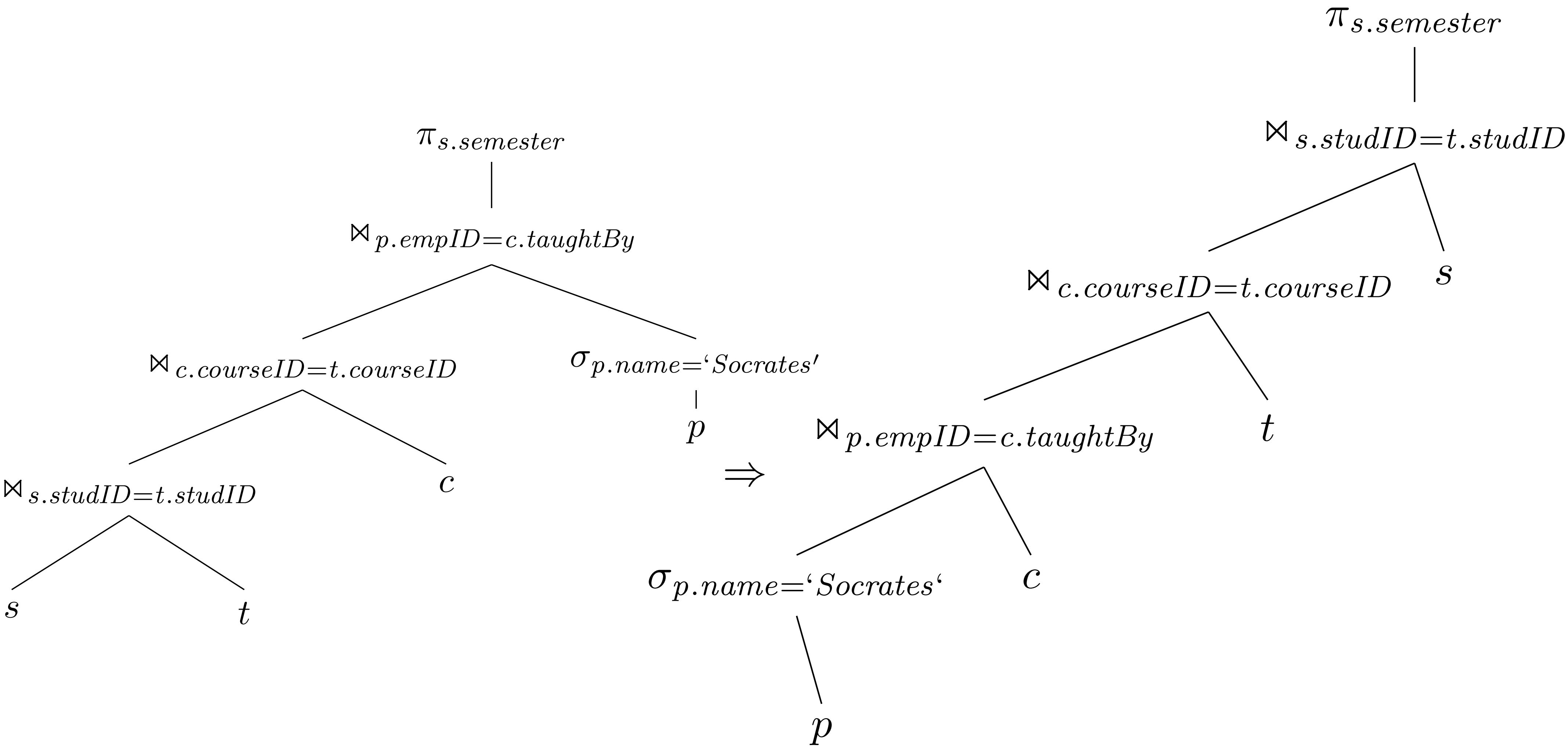
Push selections down



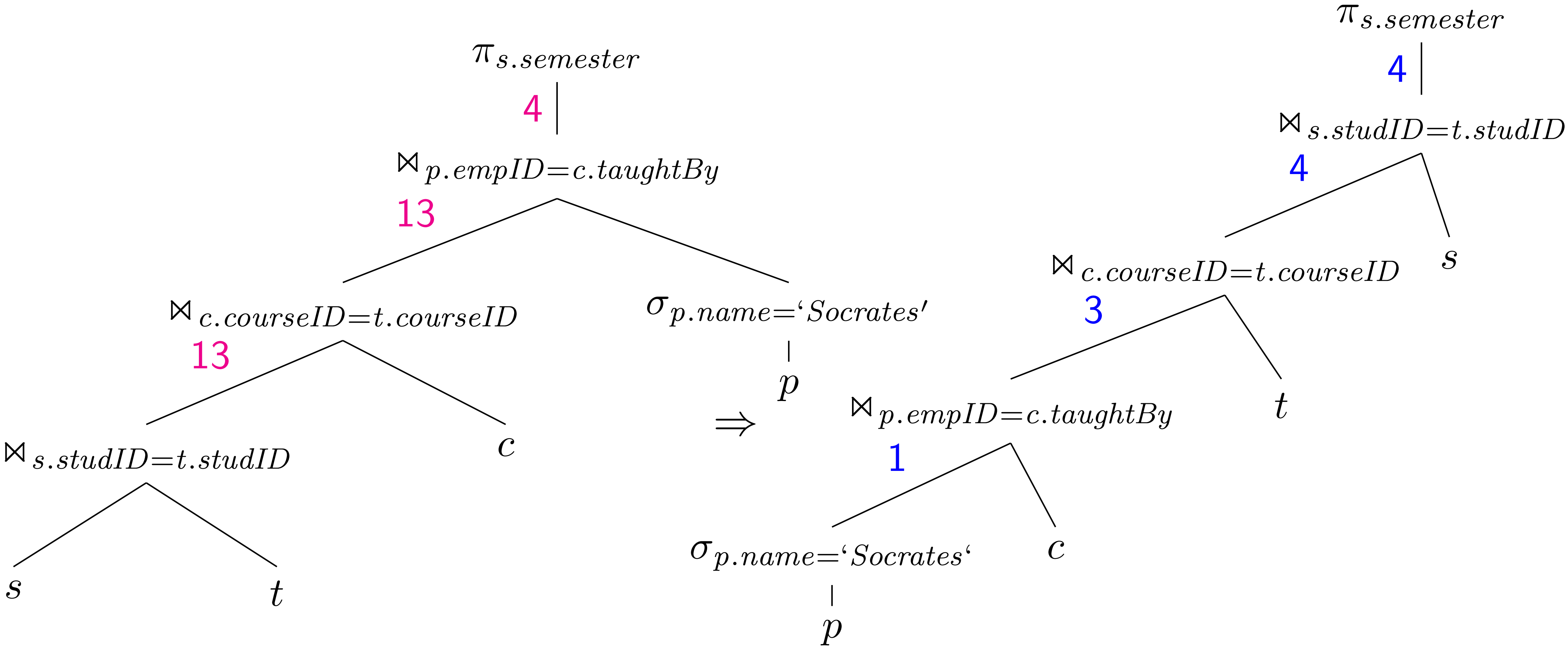
Introduce joins



Determine join order

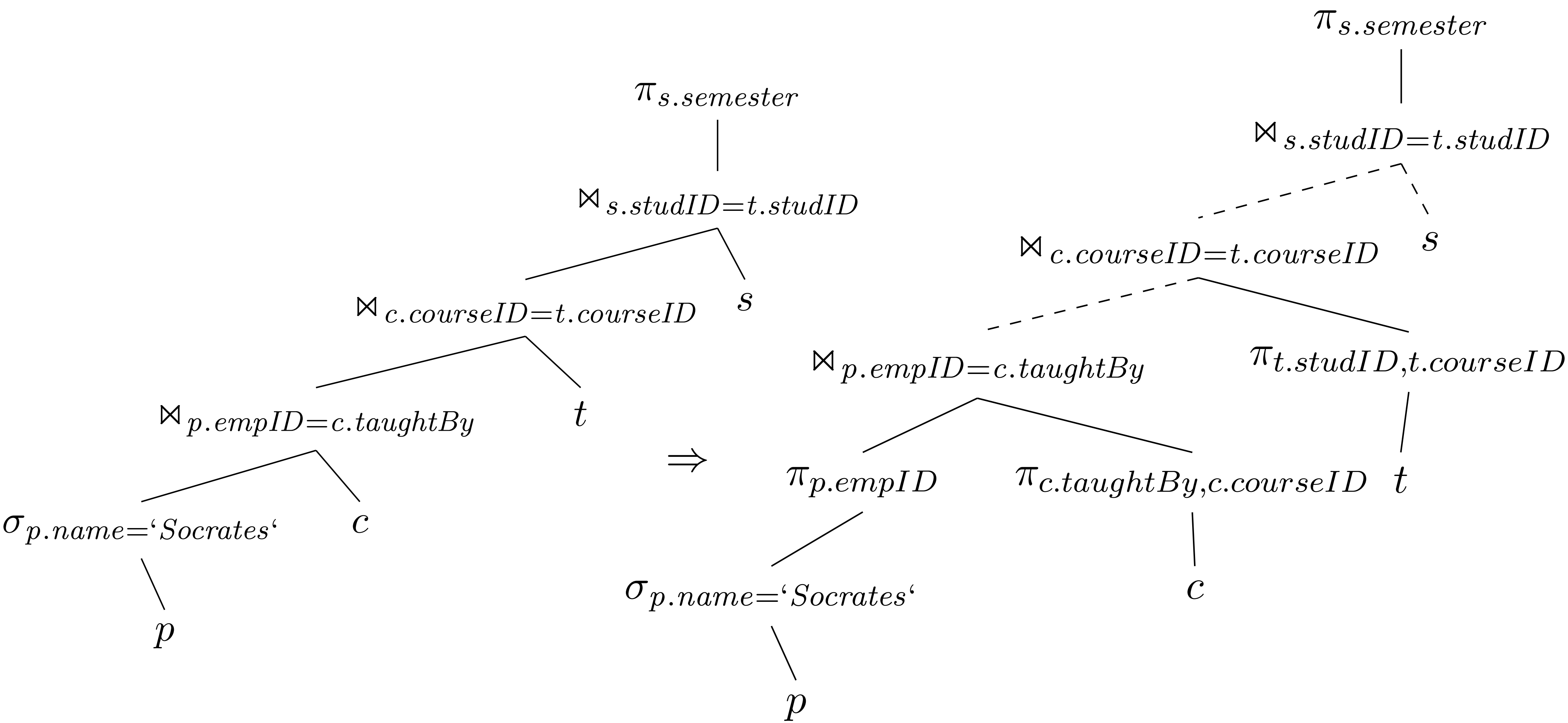


Effect: reducing the sizes of intermediate results



Sophisticated result size estimation only possible in the presence of statistics
$\rightarrow$ cost-based optimization

Introduce and push down projections



Be careful

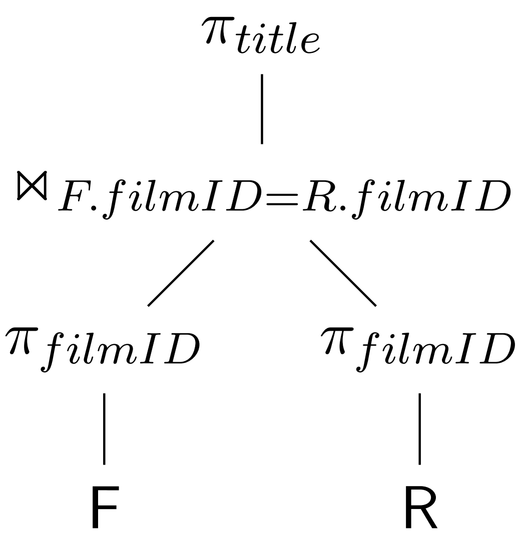
Find the titles of reserved films

SELECT DISTINCT title
FROM film F, reserved R
WHERE F.filmID = R.filmID

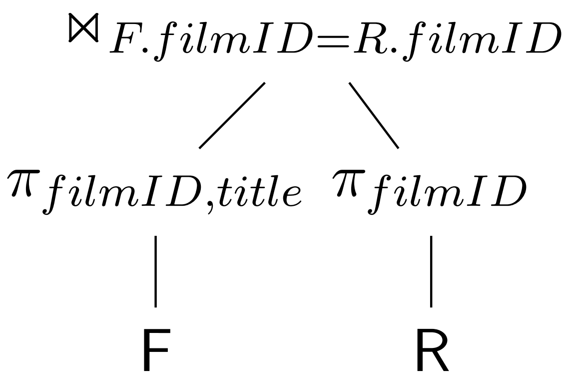
Be careful

Find the titles of reserved films

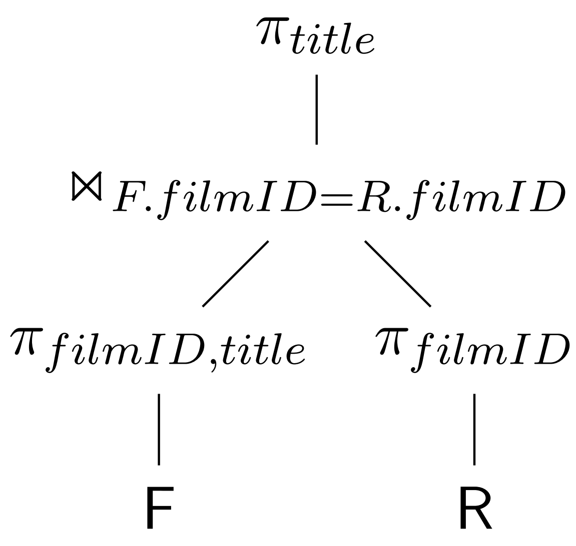
SELECT DISTINCT title
FROM film F, reserved R
WHERE F.filmID = R.filmID



Too much projection



Too little projection

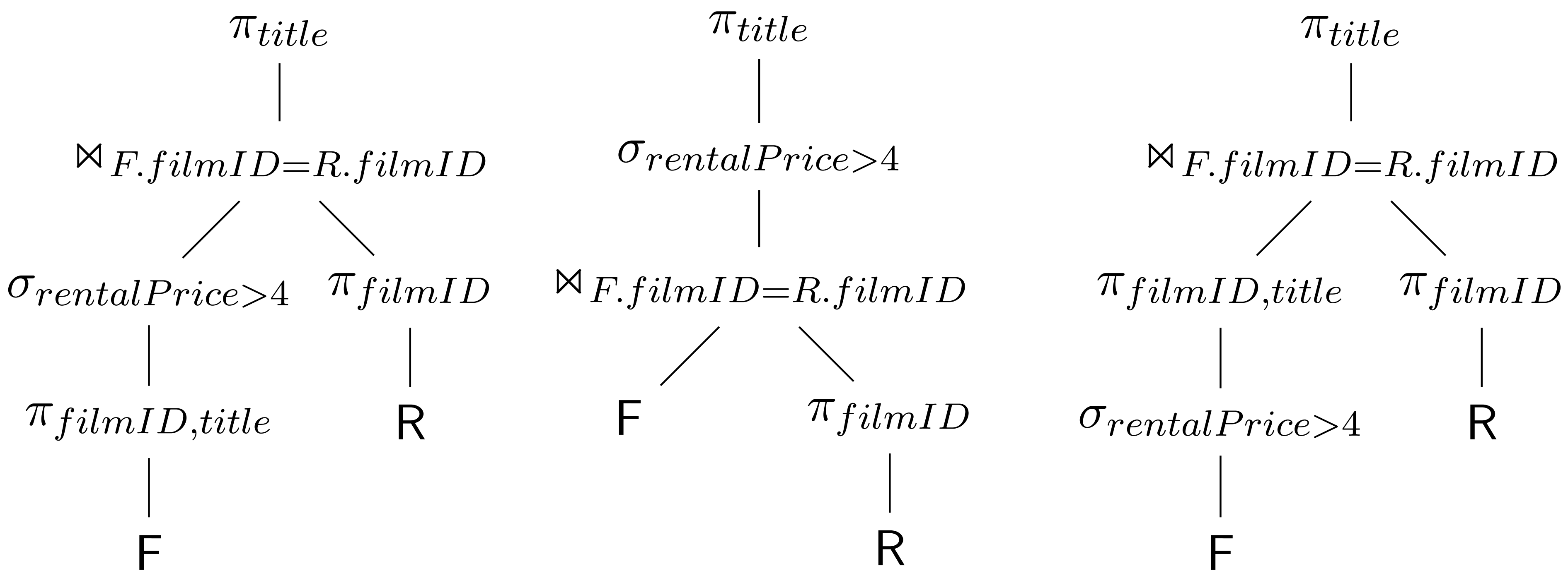


Correct

Be even more careful

Find the titles of expensive reserved films

SELECT DISTINCT title
FROM film F, reserved R
WHERE F.filmID $=$ R.filmID AND F.rentalPrice $>$ 4



Too much projection — Selection too late — Correct

Summary: heuristic query optimization

Rules of thumb

- Perform selections as early as possible
- Perform projections as early as possible

Summary: heuristic query optimization

Rules of thumb

- Perform selections as early as possible
- Perform projections as early as possible

The optimization process

- Generate initial query plan from SQL statement
- Transform query plan into more efficient query plan via a series of modifications, each of which hopefully reducing execution time

Summary: heuristic query optimization

Rules of thumb

- Perform selections as early as possible
- Perform projections as early as possible

The optimization process

- Generate initial query plan from SQL statement
- Transform query plan into more efficient query plan via a series of modifications, each of which hopefully reducing execution time

Note

- A single query plan provides all the results
- Sometimes also called rule-based query optimization

Outline

3. Operator implementations
    - Selection (access paths)
    - Join strategies

Sample database

- customer (customerID, name, street, city, state)
- reserved (customerID, filmID, resDate)
- film (filmID, title, kind, rentalPrice)

Selection taxonomy

- Primary key, point

  $$\sigma_{filmID=2}(film)$$

- Point

  $$\sigma_{title='Terminator'}(film)$$

- Range

  $$\sigma_{1<rentalPrice<4}(film)$$

- Conjunction (logical and)

  $$\sigma_{kind='F' \ \wedge \ rentalPrice=4}(film)$$

- Disjunction (logical or)

  $$\sigma_{rentalPrice<2 \ \vee \ kind='D'}(film)$$

Selection strategies

Main goal

Replace the leaf operators in the query plan with a specific access method

Selection strategies – point/range queries

Linear search

- Expensive, but always applicable

Selection strategies – point/range queries

Linear search

- Expensive, but always applicable

Binary search

- Applicable only when the file is appropriately ordered

Selection strategies – point/range queries

Linear search

- Expensive, but always applicable

Binary search

- Applicable only when the file is appropriately ordered

Primary hash index/table search

- Single record retrieval; does not work for range queries
- Retrieval of multiple records

Selection strategies – point/range queries

Linear search

- Expensive, but always applicable

Binary search

- Applicable only when the file is appropriately ordered

Primary hash index/table search

- Single record retrieval; does not work for range queries
- Retrieval of multiple records

Primary/clustering index search

- Multiple records for each index item
- Implemented with single pointer to block with first associated record

Selection strategies – point/range queries

Linear search

- Expensive, but always applicable

Binary search

- Applicable only when the file is appropriately ordered

Primary hash index/table search

- Single record retrieval; does not work for range queries
- Retrieval of multiple records

Primary/clustering index search

- Multiple records for each index item
- Implemented with single pointer to block with first associated record

Secondary index search

- Implemented with multiple pointers, each to a single record
- Might become expensive

Strategies for conjunctive queries

```
SELECT *
FROM customer
WHERE name = 'Jensen' AND street = 'Elm'
    AND state = 'Arizona'
```

- Can indexes on (name) and (street) be used?
- Can an index on (name, street, state) be used?
- Can an index on (name, street) be used?
- Can an index on (name, street, city) be used?
- Can an index on (city, name, street) be used?

Strategies for conjunctive queries

```
SELECT *
FROM customer
WHERE name = 'Jensen' AND street = 'Elm'
    AND state = 'Arizona'
```

- Can indexes on (name) and (street) be used? Yes
- Can an index on (name, street, state) be used? Yes
- Can an index on (name, street) be used? Yes
- Can an index on (name, street, city) be used? Yes
- Can an index on (city, name, street) be used? No

Strategies for conjunctive queries

```
SELECT *
FROM customer
WHERE name = 'Jensen' AND street = 'Elm'
    AND state = 'Arizona'
```

- Can indexes on (name) and (street) be used? Yes
- Can an index on (name, street, state) be used? Yes
- Can an index on (name, street) be used? Yes
- Can an index on (name, street, city) be used? Yes
- Can an index on (city, name, street) be used? No

Optimization of conjunctive queries

Indexing provides good opportunities for improving performance

Strategies for conjunctive queries

- Use available indexes
  - Ideal: composite index is applicable
  - If multiple are available
    $\rightarrow$ use the most selective index, then check remaining conditions

Strategies for conjunctive queries

- Use available indexes
  - Ideal: composite index is applicable
  - If multiple are available
    $\rightarrow$ use the most selective index, then check remaining conditions
- Use intersection of record pointers (if multiple indexes applicable)
  - Index lookups to fetch sets of record pointers
  - Intersect record pointers to perform conjunction
  - Retrieve (and check) the qualifying records

Strategies for conjunctive queries

- Use available indexes
  - Ideal: composite index is applicable
  - If multiple are available
    $\rightarrow$ use the most selective index, then check remaining conditions
- Use intersection of record pointers (if multiple indexes applicable)
  - Index lookups to fetch sets of record pointers
  - Intersect record pointers to perform conjunction
  - Retrieve (and check) the qualifying records

Disjunctive queries provide little opportunity for improving performance.

Strategies for conjunctive queries

- Use available indexes
  - Ideal: composite index is applicable
  - If multiple are available
    $\rightarrow$ use the most selective index, then check remaining conditions
- Use intersection of record pointers (if multiple indexes applicable)
  - Index lookups to fetch sets of record pointers
  - Intersect record pointers to perform conjunction
  - Retrieve (and check) the qualifying records

Disjunctive queries provide little opportunity for improving performance.

Database tuning and the creation of indexes is important!

Outline

3. Operator implementations
   - Selection (access paths)
   - Join strategies

Join algorithms

Join strategies

- Nested loop join
- Index-based join
- Sort-merge join
- Hash join

Join algorithms

Join strategies

- Nested loop join
- Index-based join
- Sort-merge join
- Hash join

Strategies work on a per block (not per record) basis

- Estimate I/Os (block retrievals)
- Use of main memory buffer

Join algorithms

Join strategies

- Nested loop join
- Index-based join
- Sort-merge join
- Hash join

Strategies work on a per block (not per record) basis

- Estimate I/Os (block retrievals)
- Use of main memory buffer

Table sizes and join selectivities influence join costs

- Query selectivity: $sel = \frac{\#\text{tuples in result}}{\#\text{candidates}}$
- For join, #candidates is the size of the Cartesian product

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

⋈

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110

Nested loop join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

⋈

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

=

ID	name	number
10	Jim	110

result

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

⋈

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

⋈

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

$=$

ID	name	number
10	Jim	110
13	Joe	150

result

Nested loop join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

$=$

ID	name	number
10	Jim	110
13	Joe	150

result

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

⋈

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170

Nested loop join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

$=$

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100

result

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100

Nested loop join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

$=$

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130

result

Nested loop join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

⋈

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

=

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

result

Nested loop join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

⋈

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

=

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

result

Nested loop join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

$=$

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

result

Nested loop join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

⋈

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

=

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

result

Nested loop join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

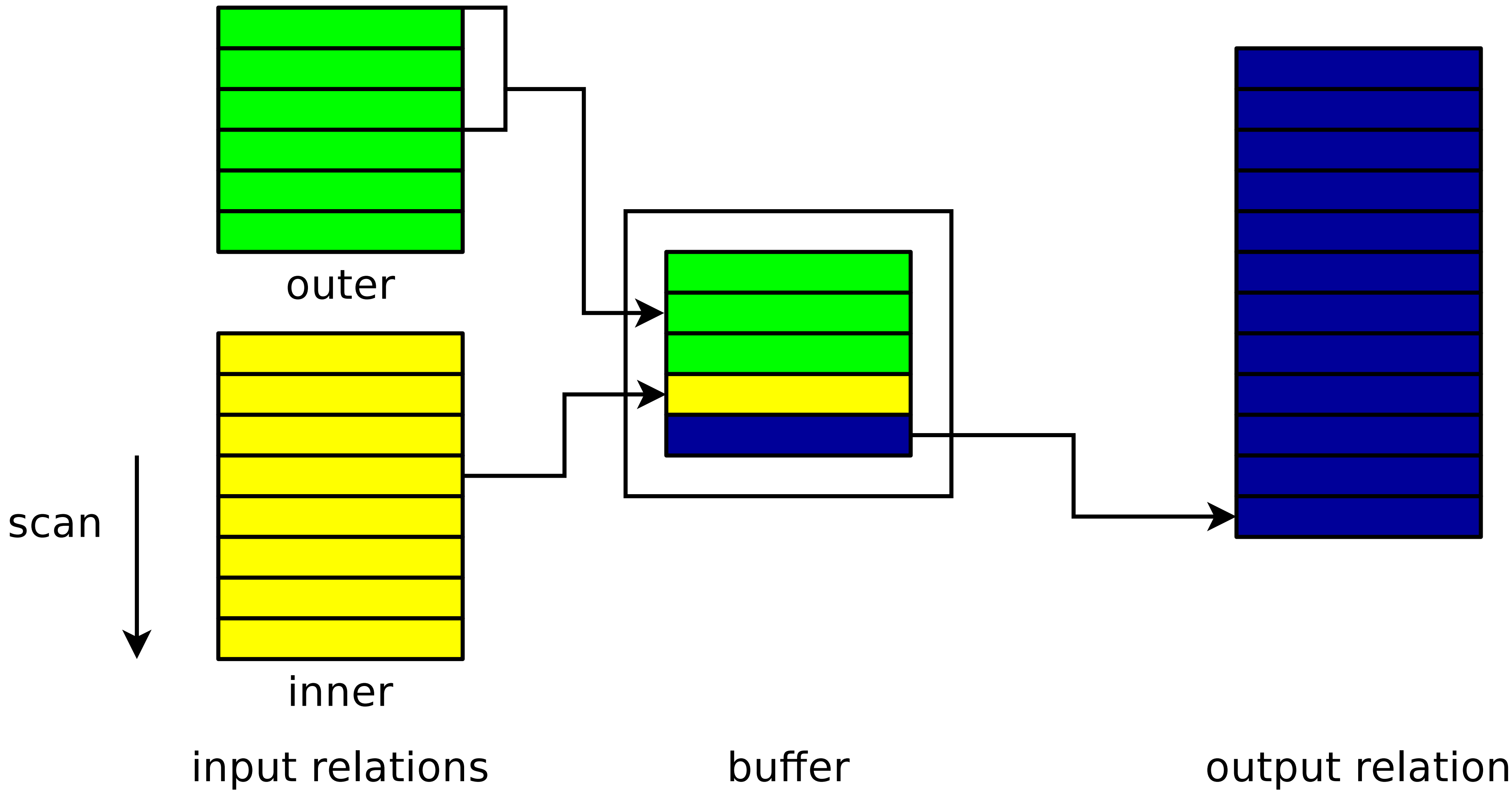
$=$

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

- Brute-force comparison, expensive exhaustive comparison
- No preprocessing of input relations needed
- No index required, all join conditions supported

Nested loop join



Block nested loop join

Not all blocks fit into main memory

repeat
  read $n_B - 2$ blocks from outer relation
  repeat
    read 1 block from inner relation
    compare tuples
  until complete inner relation read
until complete outer relation read

Block nested loop join

Not all blocks fit into main memory

repeat
  read $n_B - 2$ blocks from outer relation
  repeat
    read 1 block from inner relation
    compare tuples
  until complete inner relation read
until complete outer relation read

Parameters

- b_{inner}, b_{outer}: number of blocks
- n_B: size of main memory buffer

Cost estimation (block transfers)

$$b_{outer} + (\lceil b_{outer}/(n_B - 2)\rceil) \cdot b_{inner}$$

Block nested loop join

Not all blocks fit into main memory

repeat
  read $n_B - 2$ blocks from outer relation
  repeat
    read 1 block from inner relation
    compare tuples
  until complete inner relation read
until complete outer relation read

Parameters

- b_{inner}, b_{outer}: number of blocks
- n_B: size of main memory buffer

Cost estimation (block transfers)

$$b_{outer} + (\lceil b_{outer}/(n_B \ - 2)\rceil) \cdot b_{inner}$$

If we know more system parameters (block transfer, disk seeks, CPU speed,...) and the size of input relations, we can estimate the time to compute the join.

Block nested loop join

Example ($reserved \bowtie customer$)

- number of blocks
  $b_{reserved} = 2.000$, $b_{customer} = 10$
- size of main memory buffer
  $n_B = 6$
- Cost estimation (block transfers)
  $b_{outer} + (\lceil b_{outer}/(n_B - 2) \rceil) \cdot b_{inner}$

Block nested loop join

Example ($reserved \bowtie customer$)

- number of blocks
  $b_{reserved} = 2.000$, $b_{customer} = 10$
- size of main memory buffer
  $n_B = 6$
- Cost estimation (block transfers)
  $b_{outer} + (\lceil b_{outer}/(n_B - 2) \rceil) \cdot b_{inner}$

Costs

- reserved as outer
  $2.000 + \lceil (2.000/4) \rceil \cdot 10 = 7.000$
- customer as outer
  $10 + \lceil (10/4) \rceil \cdot 2.000 = 6.010$

Index-based block nested loop join

Same principle as standard nested loop join

- Outer relation
- Inner relation
- Index lookups can replace file scans on the inner relation

Merge join

Exploit sorted relations

R	
	A
...	0
...	7
...	7
...	8
...	8
...	10
...	...

$\leftarrow$ $\rightarrow$

S	
B	
5	...
6	...
7	...
8	...
8	...
11	...
...	...

Assumption:
Both input relations are sorted

Merge join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

Merge join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

=

result

ID	name	number

Merge join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

$=$

result

ID	name	number
10	Jim	110

Merge join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

=

result

ID	name	number
10	Jim	110
13	Joe	150

Merge join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

phone

=

ID	name	number
10	Jim	110
13	Joe	150

result

Merge join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

phone

$=$

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120

result

Merge join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160

Merge join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

phone

$=$

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170

result

Merge join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

⋈

phone

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100

Merge join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

⋈

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

phone

=

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130

result

Merge join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

=

result

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

Merge join – costs

Parameters

- b_1, b_2: number of blocks

Cost estimation (block transfers)

$$b_1 + b_2$$

Merge join – costs

Parameters

- b_1, b_2: number of blocks

Cost estimation (block transfers)

$$b_1 + b_2$$

Extensions

- Combination with sorting if input relations are not sorted
- Not enough main memory

Hash join

emp

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

phone

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

Apply hash functions to the join attributes
$\rightarrow$ partition tuples into buckets

Hash join

$emp_0 \bowtie phone_0 = result_0$

emp_0

ID	name
15	Pete
21	Dave

$phone_0$

number	ID
120	15
160	15
170	21

$result_0$

ID	name	number
15	Pete	120
15	Pete	160
21	Dave	170

$emp_1 \bowtie phone_1 = result_1$

emp_1

ID	name
10	Jim
13	Joe

$phone_1$

number	ID
110	10
150	13

$result_1$

ID	name	number
10	Jim	110
13	Joe	150

$emp_2 \bowtie phone_2 = result_2$

emp_2

ID	name
14	Sue
23	Anne

$phone_2$

number	ID
100	23
130	23
140	23

$result_2$

ID	name	number
23	Anne	100
23	Anne	130
23	Anne	140

Hash join

ID	name
15	Pete
21	Dave

emp_0

$\bowtie$

number	ID
120	15
160	15
170	21

$phone_0$

$=$

ID	name	number
15	Pete	120
15	Pete	160
21	Dave	170

$result_0$

ID	name
10	Jim
13	Joe

emp_1

$\bowtie$

number	ID
110	10
150	13

$phone_1$

$=$

ID	name	number
10	Jim	110
13	Joe	150

$result_1$

ID	name
14	Sue
23	Anne

emp_2

$\bowtie$

number	ID
100	23
130	23
140	23

$phone_2$

$=$

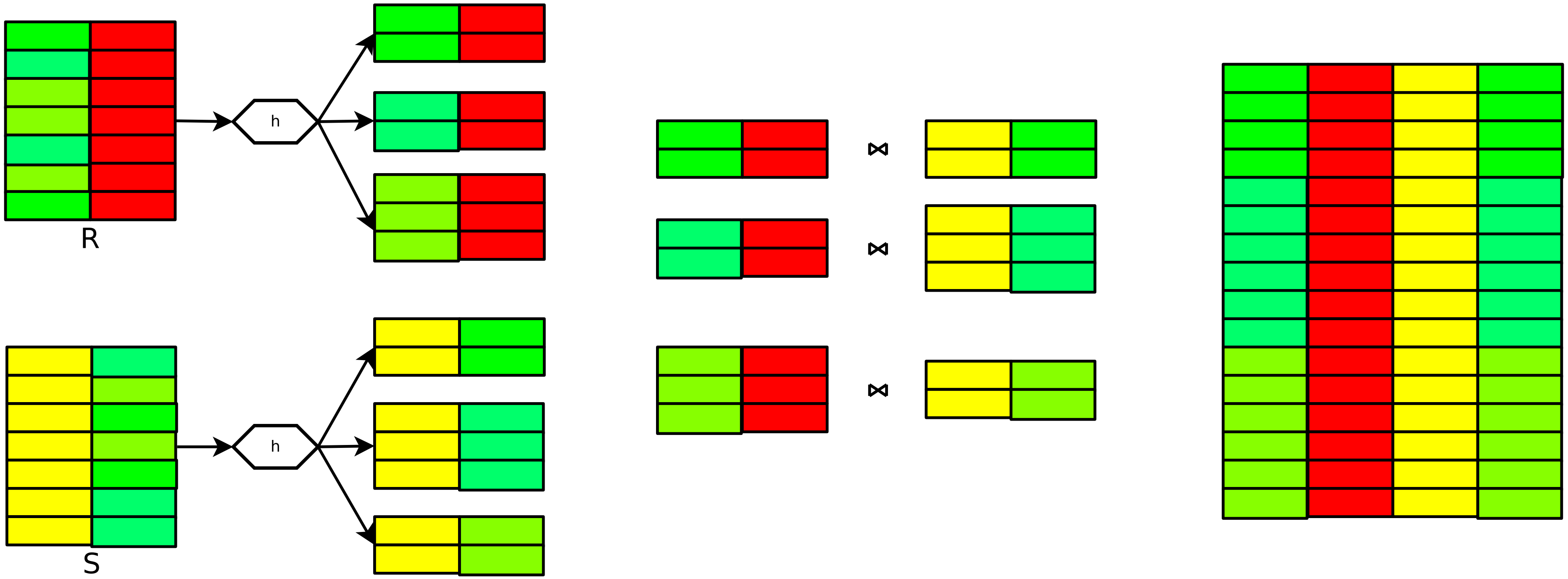
ID	name	number
23	Anne	100
23	Anne	130
23	Anne	140

$result_2$

$$result = result_0 \cup result_1 \cup result_2$$

Hash join

- Hash each relation on the join attributes
- Each bucket must be small enough to fit into memory
- Join corresponding buckets from each relation



Input Relations

Output Relation

Hash join

Parameters

- b_1, b_2: number of blocks for tables R_1 and R_2

Steps

- Partitioning table R_1 with h_1 into buckets r_{1_i} (read all / write all) $2 \times b_1$

Cost estimation (block transfers)

$$2 \times b_1$$

Hash join

Parameters

- b_1, b_2: number of blocks for tables R_1 and R_2

Steps

- Partitioning table R_1 with h_1 into buckets r_{1_i} (read all / write all) $2 \times b_1$
- Partitioning table R_2 with h_1 into buckets r_{2_i}(read all / write all) $2 \times b_2$

Cost estimation (block transfers)

$$2 \times b_1 + 2 \times b_2$$

Hash join

Parameters

- b_1, b_2: number of blocks for tables R_1 and R_2

Steps

- Partitioning table R_1 with h_1 into buckets r_{1_i} (read all / write all)
  $2 \times b_1$
- Partitioning table R_2 with h_1 into buckets r_{2_i}(read all / write all)
  $2 \times b_2$
- Build phase:
  use h_2 to create an in-memory hash index on bucket r_{1_i} (read all)
  b_1

Cost estimation (block transfers)

$$3 \times b_1 + 2 \times b_2$$

Hash join

Parameters

- b_1, b_2: number of blocks for tables R_1 and R_2

Steps

- Partitioning table R_1 with h_1 into buckets r_{1_i} (read all / write all)
  $2 \times b_1$
- Partitioning table R_2 with h_1 into buckets r_{2_i}(read all / write all)
  $2 \times b_2$
- Build phase:
  use h_2 to create an in-memory hash index on bucket r_{1_i} (read all)
  b_1
- Probe phase:
  for corresponding r_{2_i}, use h_2 to test in-memory index for matches (read all)
  b_2

Cost estimation (block transfers)

$$3 \times b_1 + 3 \times b_2$$

Hash join

Parameters

- b_1, b_2: number of blocks for tables R_1 and R_2

Steps

- Partitioning table R_1 with h_1 into buckets r_{1_i} (read all / write all)
  $2 \times b_1$
- Partitioning table R_2 with h_1 into buckets r_{2_i}(read all / write all)
  $2 \times b_2$
- Build phase:
  use h_2 to create an in-memory hash index on bucket r_{1_i} (read all)
  b_1
- Probe phase:
  for corresponding r_{2_i}, use h_2 to test in-memory index for matches (read all)
  b_2

Cost estimation (block transfers)

$$3 \times b_1 + 3 \times b_2 + \epsilon \quad \text{(partially filled blocks)}$$

Costs and applicability of join strategies

Nested loop join

- Can be used for all join types
- Can be quite expensive

Merge join

- Files need to be sorted on the join attributes
  Sorting can be done for the purpose of the join
- Can use indexes

Hash join

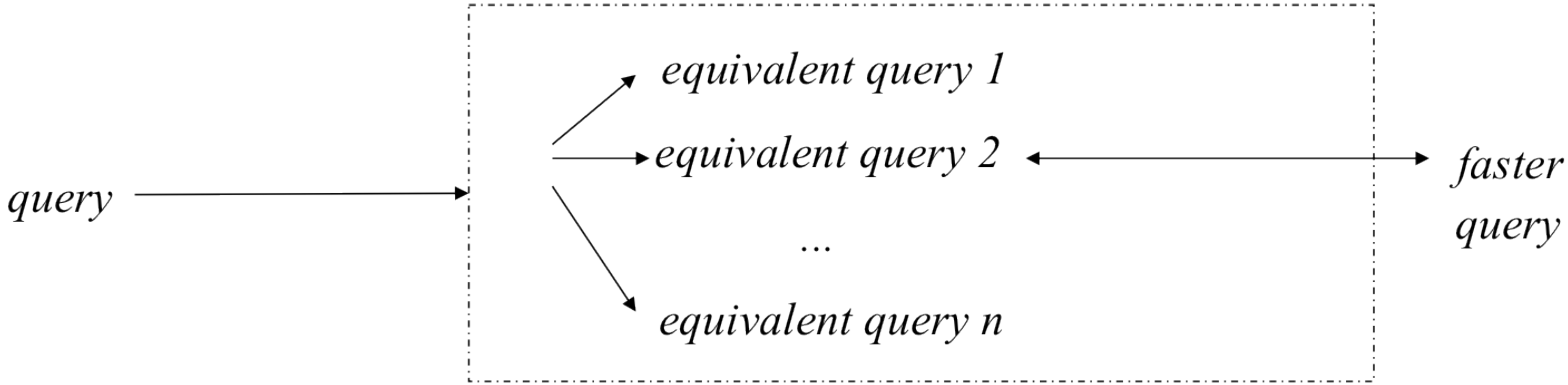
- Good hash functions are essential
- Performance best if smallest relation fits into main memory

Outline

4. Cost-based (physical) query optimization
   - Selectivity and cardinality
   - Cost estimation
   - PostgreSQL

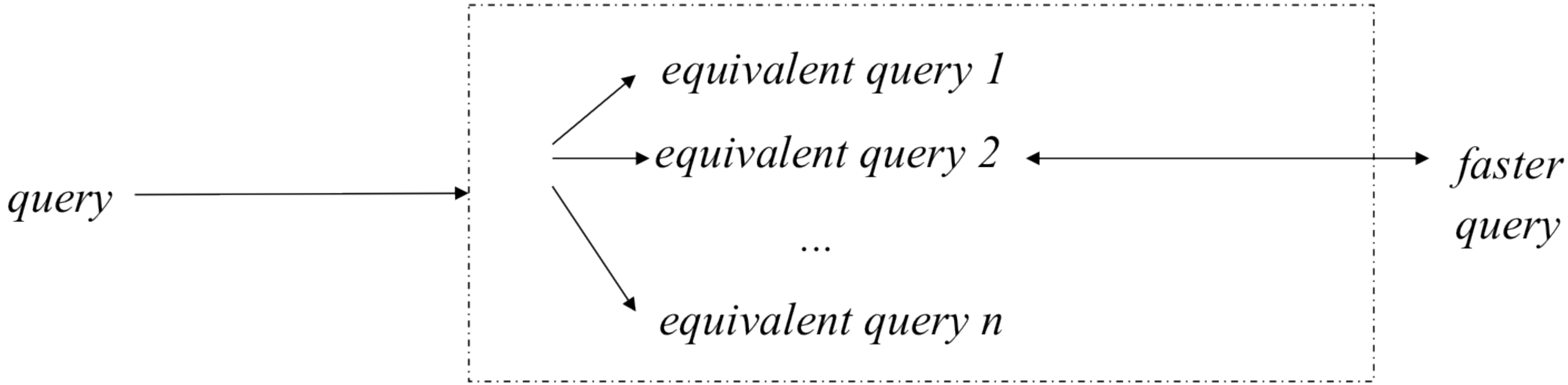
Objective

For a given query, find the most efficient query execution plan



Objective

For a given query, find the most efficient query execution plan



Optimization

- Heuristic (logical) optimization
  - Query tree (relational algebra) optimization
- Cost-based (physical) optimization

Physical query optimization

Physical query optimization

- Generate alternative query execution plans
- Choose algorithms and access paths
- Compute costs
- Choose cheapest query execution plan

Physical query optimization

Physical query optimization

- Generate alternative query execution plans
- Choose algorithms and access paths
- Compute costs
- Choose cheapest query execution plan

Prerequisite

- Cost model
- Statistics on the input to each operation
  - Statistics on leaf relations: stored in system catalog
  - Statistics on intermediate relations must be estimated (cardinalities)

Outline

4. Cost-based (physical) query optimization
   - Selectivity and cardinality
   - Cost estimation
   - PostgreSQL

Statistics per relation

For relation r

- Number of tuples (records): n_r
- Tuple size in relation r: l_r
- Load factor (fill factor), percentage of space used in each block
- Blocking factor (number of records per block)
- Relation size in blocks: b_r
- Relation organization
  Heap, hash, indexes, clustered
- Number of overflow blocks

Statistics per attribute

For attribute A in relation r

- Size and type
- Number of distinct values for attribute A: $V(A, r)$
  The same as the size of $\pi_A(r)$
- Selection cardinality $S(A, r)$
  The same as the size of $\sigma_{A=a}(r)$ for an arbitrary value a
- Probability distribution over the values
  Alternative: assume uniform distribution

Statistics per attribute

For attribute A in relation r

- Size and type
- Number of distinct values for attribute A: $V(A, r)$
  The same as the size of $\pi_A(r)$
- Selection cardinality $S(A, r)$
  The same as the size of $\sigma_{A=a}(r)$ for an arbitrary value a
- Probability distribution over the values
  Alternative: assume uniform distribution

Statistics need to be updated when the table is updated!

Statistics per index

- Base relation
- Indexed attribute(s)
- Organization, e.g., B^+-tree, hash
- Clustering index?
- On key attribute(s)?
- Sparse or dense?
- Number of levels (if appropriate)
- Number of leaf-level index blocks

Outline

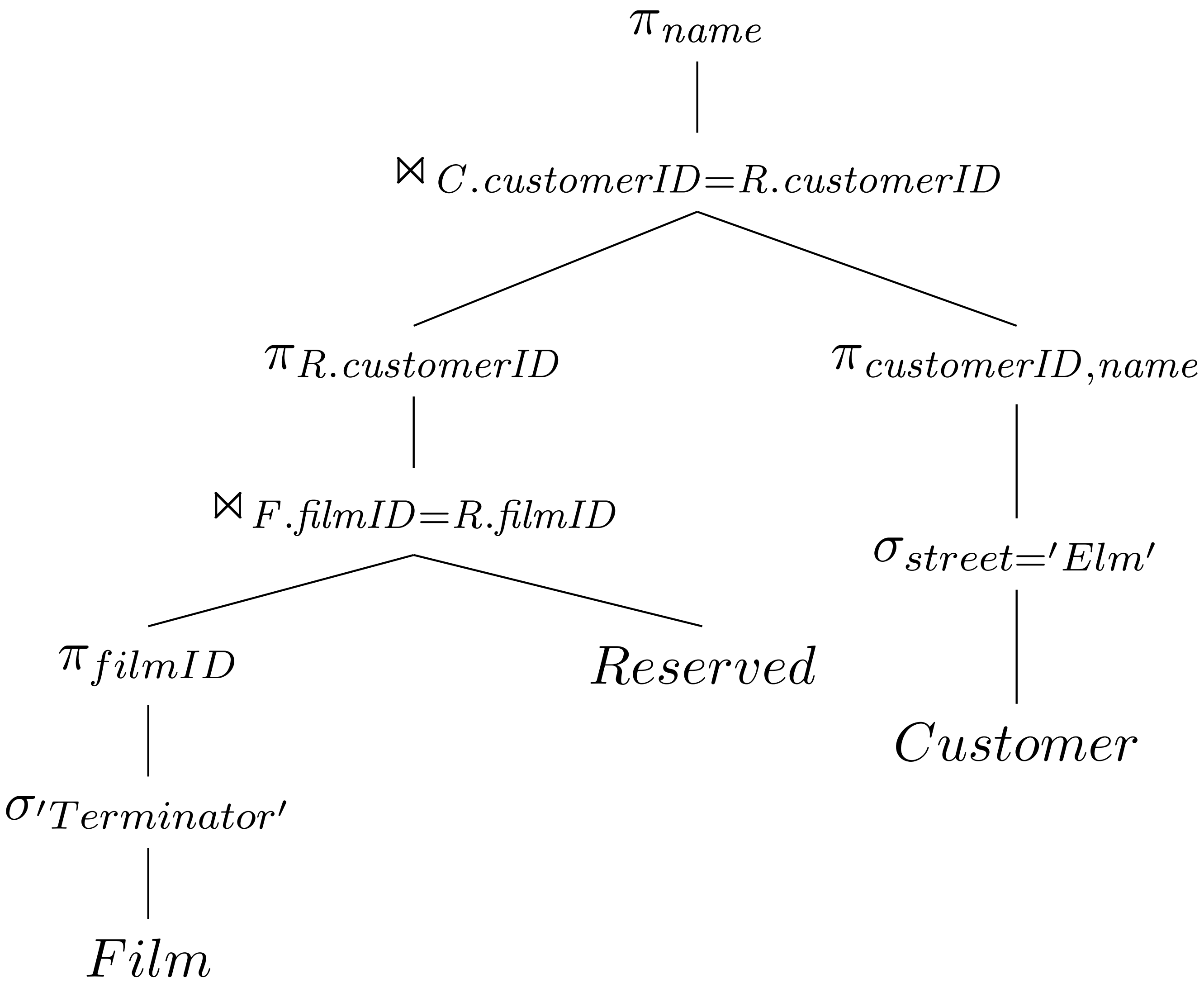
4. Cost-based (physical) query optimization
   - Selectivity and cardinality
   - Cost estimation
   - PostgreSQL

Cost estimation example

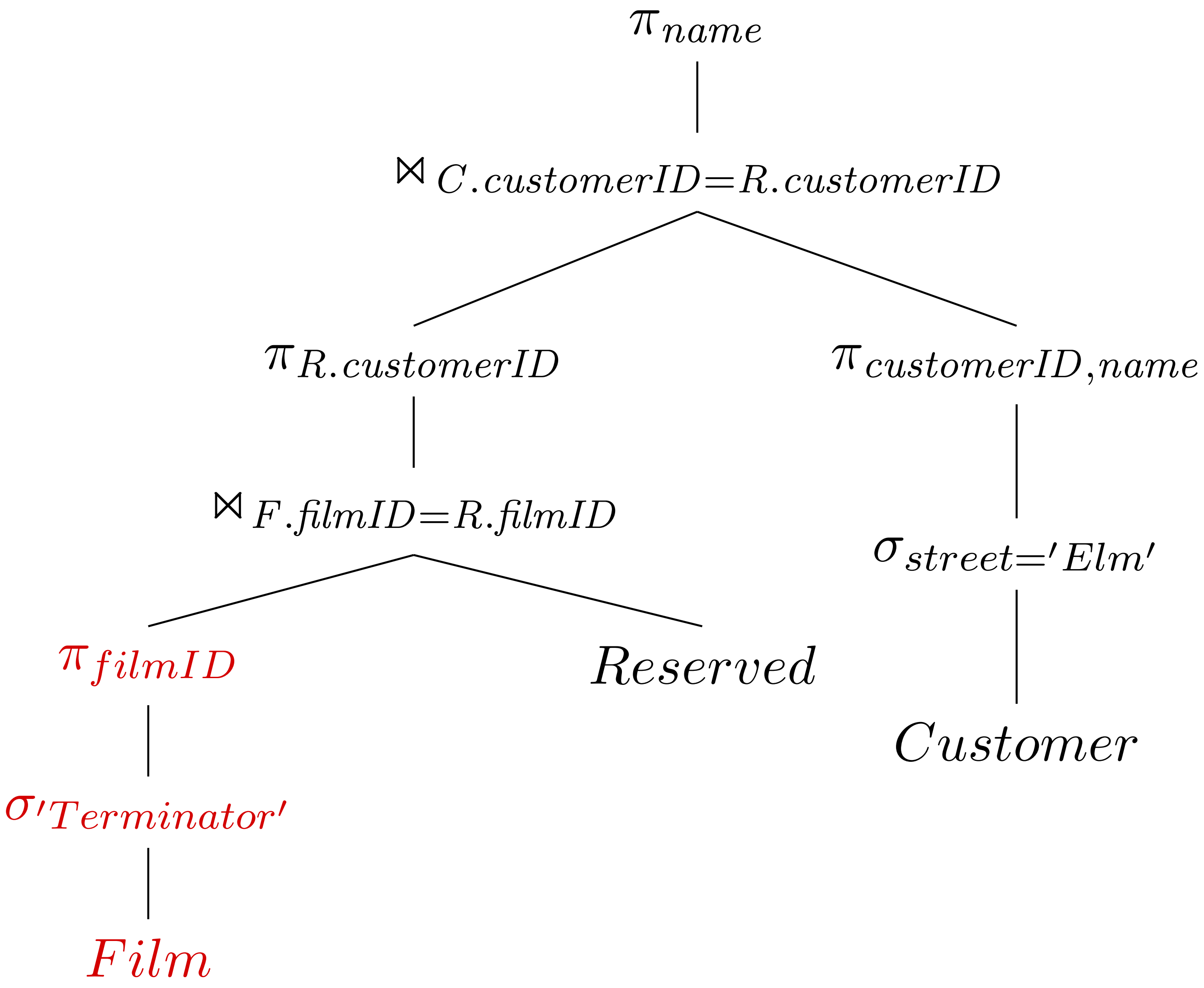
What are the names of customers living on Elm street who have reserved "Terminator"?

SELECT name
FROM customer C, reserved R, Film F
WHERE title = 'Terminator' AND F.filmID = R.filmID
AND C.customerID = R.customerID AND C.street = 'Elm';

Cost estimation example



Cost estimation example



Cost estimation example

$$\pi_{filmID}(\sigma_{title='Terminator'}(Film))$$

Statistics

- Relation statistics
  - number of tuples: $n_{Film} = 5000$
  - relation size in blocks: $b_{Film} = 50$
- Attribute statistics
  - Selection cardinality: $S(title, Film) = 1$
- Index statistics
  - Hash index on attribute “title”

Cost estimation example

$$\pi_{filmID}(\sigma_{title='Terminator'}(Film))$$

Statistics

- Relation statistics
  - number of tuples: $n_{Film} = 5000$
  - relation size in blocks: $b_{Film} = 50$
- Attribute statistics
  - Selection cardinality: $S(title, Film) = 1$
- Index statistics
  - Hash index on attribute “title”

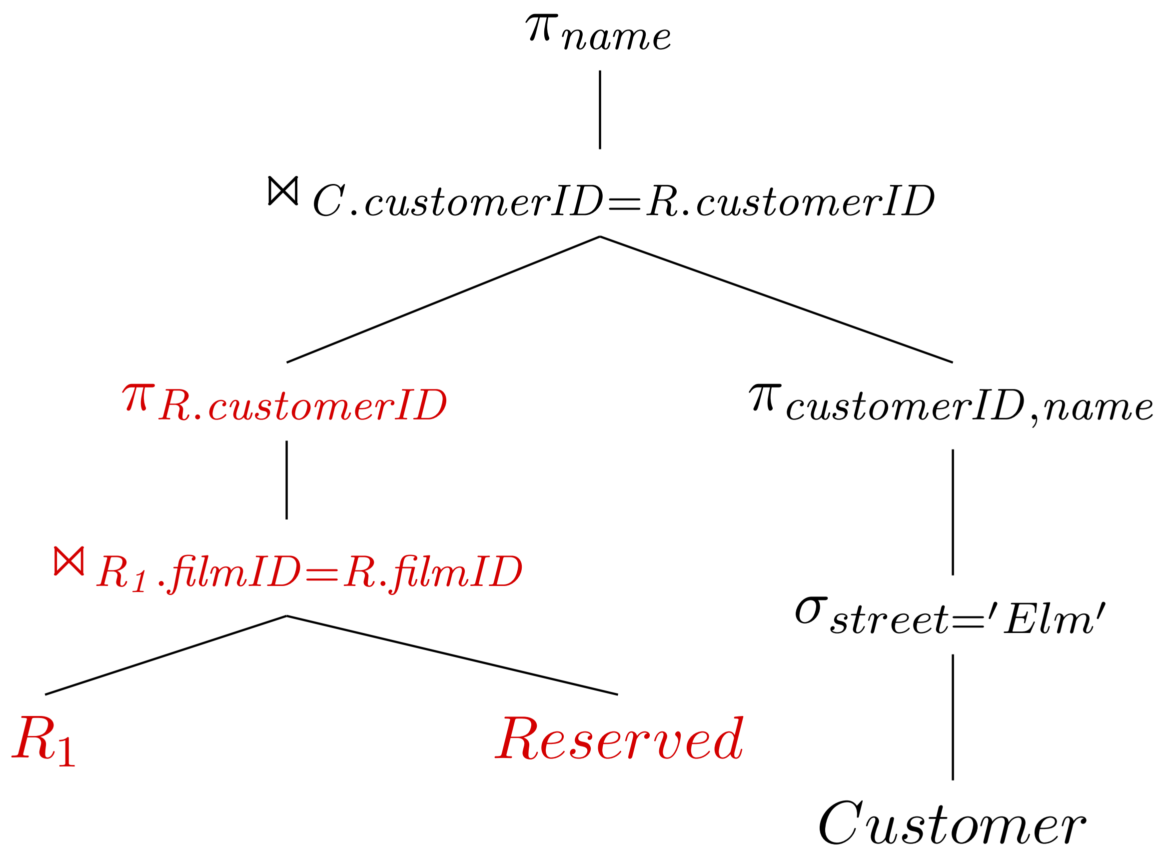
Execution

- Use index with ‘Terminator’
- Project on filmID
- Leave result in main memory (1 block)

$costs_{\text{disk access}} = 1$

Result size: 1 tuple

Cost estimation example



Cost estimation example

$$\pi_{R.customerID}(R_1 \bowtie_{R_1.filmID=R.filmID} Reserved)$$

Statistics

- Relation statistics
  - number of tuples: $n_{Reserved} = 40000$
  - relation size in blocks: $b_{Film} = 2000$
- Attribute statistics
  - Selection cardinality: $S(filmID, Reserved) = 8$
- Index statistics
  - Primary B$^+$-tree index for Reserved on filmID with 2 levels

Cost estimation example

$$\pi_{R.customerID}(R_1 \bowtie_{R_1.filmID=R.filmID} Reserved)$$

Statistics

- Relation statistics
  - number of tuples: $n_{Reserved} = 40000$
  - relation size in blocks: $b_{Film} = 2000$
- Attribute statistics
  - Selection cardinality: $S(filmID, Reserved) = 8$
- Index statistics
  - Primary B$^+$-tree index for Reserved on filmID with 2 levels

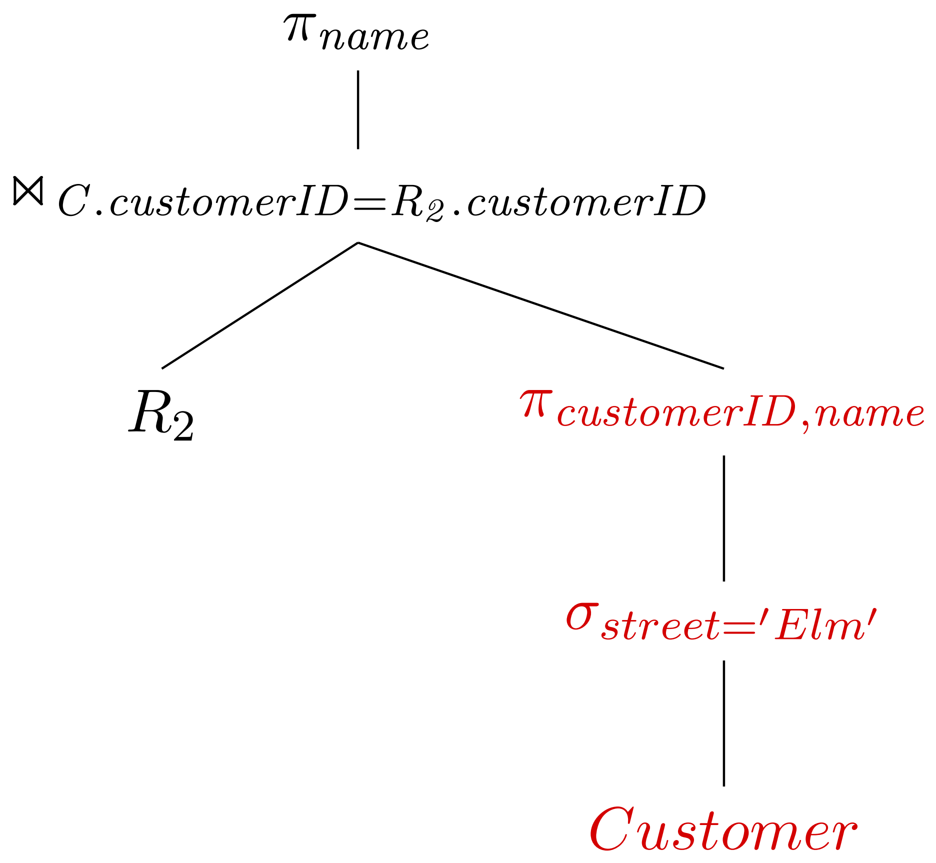
Execution

- Index join using B$^+$-tree
- Project on customerID
- Leave result in main memory (1 block)

$costs_{\text{disk access}} = 3$
(2 index levels, 1 record lookup)

Result size: 8 tuples

Cost estimation example



Cost estimation example

$$\pi_{customerID,name}(\sigma_{street='Elm'}(Customer))$$

Statistics

- Relation statistics
  - number of tuples: $n_{Customer} = 200$
  - relation size in blocks: $b_{Customer} = 10$
- Attribute statistics
  - Selection cardinality: $S(street, Customer) = 10$
- Index statistics
  - No index on “street”

Cost estimation example

$$\pi_{customerID,name}(\sigma_{street='Elm'}(Customer))$$

Statistics

- Relation statistics
  - number of tuples: $n_{Customer} = 200$
  - relation size in blocks: $b_{Customer} = 10$
- Attribute statistics
  - Selection cardinality: $S(street, Customer) = 10$
- Index statistics
  - No index on "street"

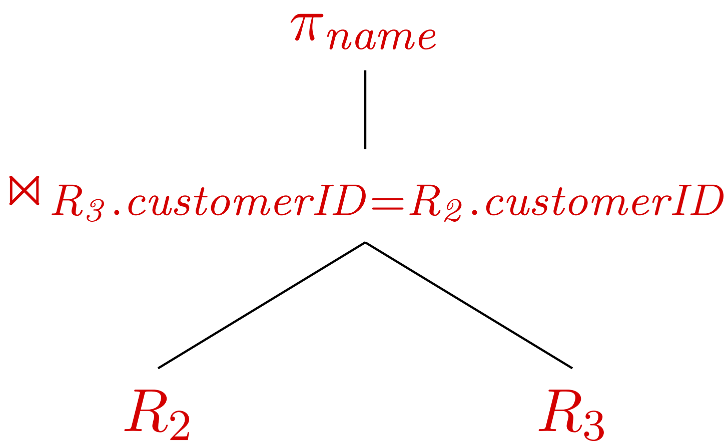
Execution

- Linear search of Customer
- Project on customerID, name
- Leave result in main memory (1 block)

$costs_{\text{disk access}} = 10$

Result size: 10 tuples

Cost estimation example



Cost estimation example

$$\pi_{name}(R_2 \bowtie_{R_3.customerID=R_2.customerID} R_3)$$

Execution

- Main memory join on relations

Total Costs

$costs_{\text{disk access}} = 1 + 3 + 10 + 0 = 14$

Cost model

Cost models consider more aspects than only disk access

- CPU time
- Communication time
- Main memory usage
- . . .

Cost model

Cost models consider more aspects than only disk access

- CPU time
- Communication time
- Main memory usage
- ...

For this purpose, we need to estimate input/output sizes of each operator

- Statistics on leaf relations: stored in system catalog
- Statistics on intermediate relations must be estimated (cardinalities)

Cost model

Cost models consider more aspects than only disk access

- CPU time
- Communication time
- Main memory usage
- ...

For this purpose, we need to estimate input/output sizes of each operator

- Statistics on leaf relations: stored in system catalog
- Statistics on intermediate relations must be estimated (cardinalities)

Additional aspects

- Spanning search space (dynamic programming, exhaustive search,...)
- Bushy vs. left-deep join trees (parallelism vs. pipelining)
- Multiquery optimization (shared scans,...)
- ...

Heuristic vs. cost-based query optimization

Heuristic

- Can always be used
- Sequences of query plans are generated
- Each plan is (presumably) more efficient than the previous
- Search is linear

Cost-based

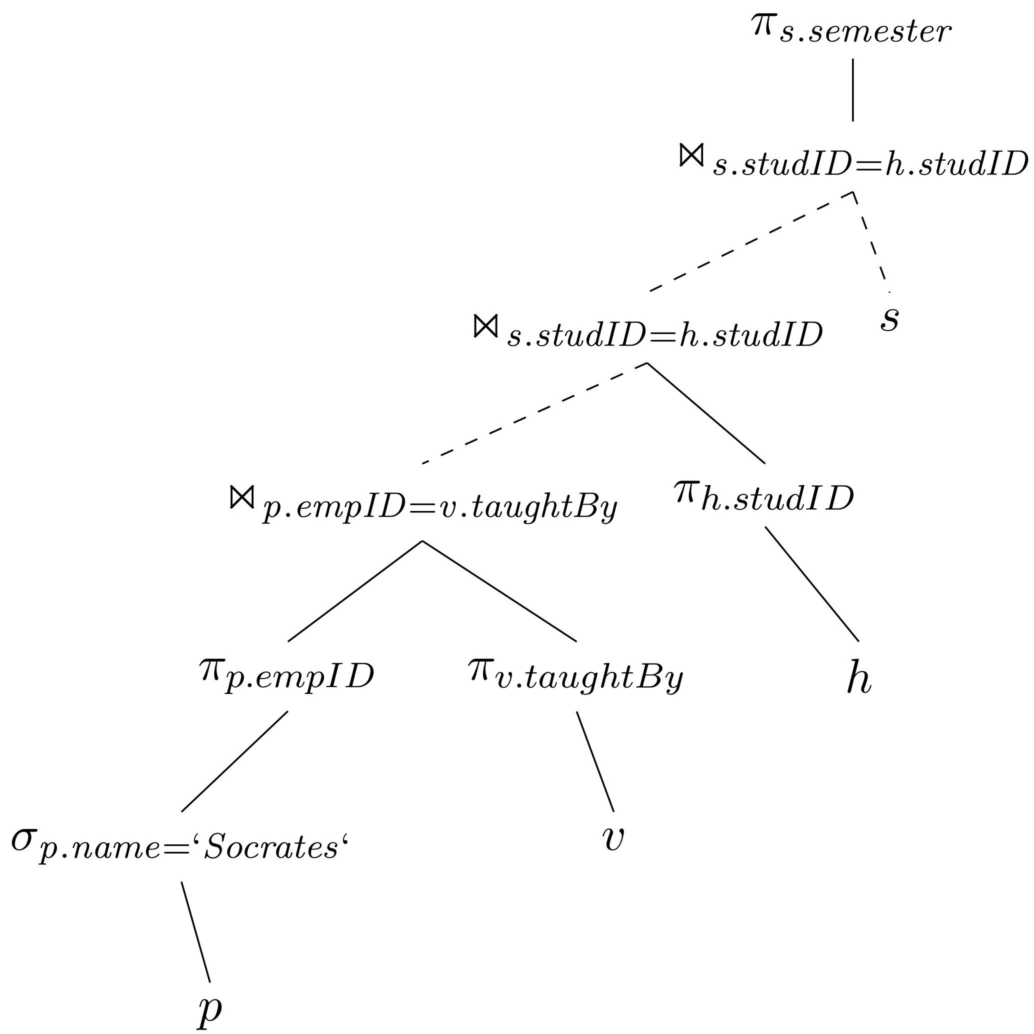
- Can only be used if statistics are kept and maintained
- Many query plans are generated
- The costs of each plan is estimated, and the most efficient one is chosen
- Search is multi-dimensional

Outline

4. Cost-based (physical) query optimization
   - Selectivity and cardinality
   - Cost estimation
   - PostgreSQL

PostgreSQL

SELECT DISTINCT s.semester
FROM student s, takes h,
course v, professor p
WHERE p.name=‘Socrates‘ AND
v.taughtBy = p.empID AND
v.courseID = h.courseID AND
h.studID = s.studID;



PostgreSQL EXPLAIN

EXPLAIN SELECT DISTINCT s.semester
FROM student s, takes h,
course v, professor p
WHERE p.name='Socrates' AND
v.taughtBy = p.empID AND
v.courseID = h.courseID AND
h.studID = s.studID;

EXPLAIN

Display the execution plan that the PostgreSQL planner generates for the supplied statement

PostgreSQL EXPLAIN

```
QUERY PLAN
text
Unique  (cost=4.61..4.62 rows=2 width=4)
  ->  Sort  (cost=4.61..4.62 rows=2 width=4)
        Sort Key: s.semester
        ->  Hash Join  (cost=3.47..4.60 rows=2 width=4)
              Hash Cond: (s.studid = h.studid)
              ->  Seq Scan on student s  (cost=0.00..1.08 rows=8 width=8)
              ->  Hash  (cost=3.45..3.45 rows=2 width=4)
                    ->  Hash Join  (cost=2.26..3.45 rows=2 width=4)
                          Hash Cond: (h.courseid = v.courseid)
                          ->  Seq Scan on takes h  (cost=0.00..1.13 rows=13 width=8)
                          ->  Hash  (cost=2.25..2.25 rows=1 width=4)
                                ->  Hash Join  (cost=1.10..2.25 rows=1 width=4)
                                      Hash Cond: (v.taughtby = p.empid)
                                      ->  Seq Scan on course v  (cost=0.00..1.10 rows=10 width=8)
                                      ->  Hash  (cost=1.09..1.09 rows=1 width=4)
                                            ->  Seq Scan on professor p  (cost=0.00..1.09 rows=1 width=4)
                                                  Filter: ((name)::text = 'Socrates'::text)
```

PostgreSQL EXPLAIN ANALYZE

EXPLAIN ANALYZE SELECT DISTINCT s.semester
FROM student s, takes h,
course v, professor p
WHERE p.name='Socrates' AND
v.taughtBy = p.empID AND
v.courseID = h.courseID AND
h.studID = s.studID;

EXPLAIN ANALYZE

The additional ANALYZE option causes the statement to be actually executed, not only planned.

ANALYZE

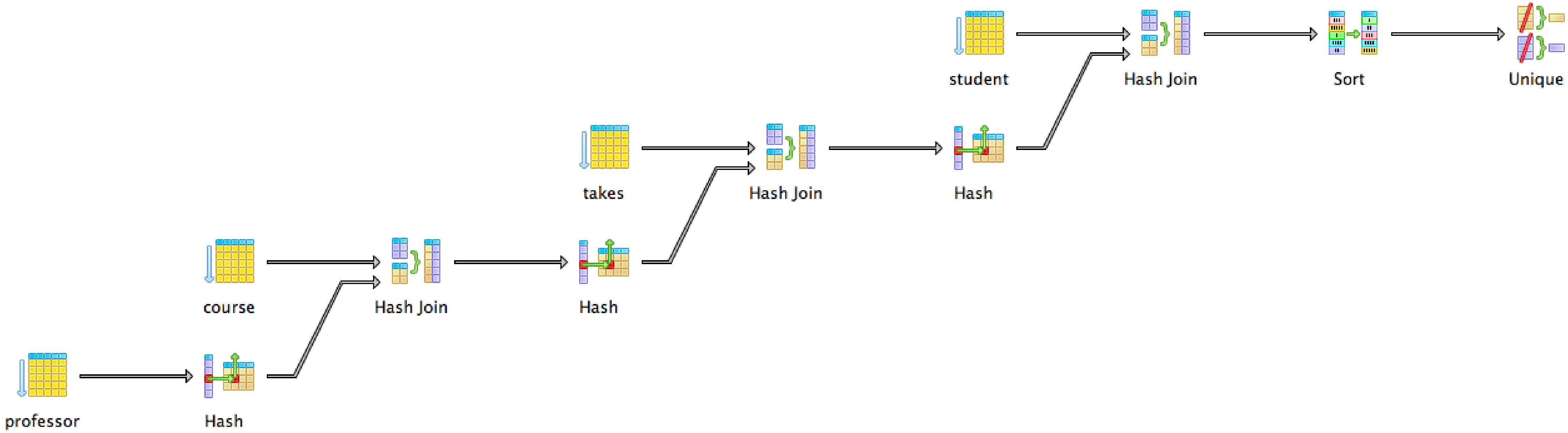
ANALYZE collects statistics about the contents of tables in the database.

PostgreSQL EXPLAIN ANALYZE

QUERY PLAN
text

```
Unique  (cost=4.61..4.62 rows=2 width=4) (actual time=0.087..0.091 rows=3 loops=1)
  ->  Sort  (cost=4.61..4.62 rows=2 width=4) (actual time=0.087..0.089 rows=4 loops=1)
        Sort Key: s.semester
        Sort Method: quicksort  Memory: 25kB
        ->  Hash Join  (cost=3.47..4.60 rows=2 width=4) (actual time=0.071..0.075 rows=4 loops=1)
              Hash Cond: (s.studid = h.studid)
              ->  Seq Scan on student s  (cost=0.00..1.08 rows=8 width=8) (actual time=0.004..0.005 rows=8 loops=1)
              ->  Hash  (cost=3.45..3.45 rows=2 width=4) (actual time=0.054..0.054 rows=4 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 1kB
                    ->  Hash Join  (cost=2.26..3.45 rows=2 width=4) (actual time=0.043..0.053 rows=4 loops=1)
                          Hash Cond: (h.courseid = v.courseid)
                          ->  Seq Scan on takes h  (cost=0.00..1.13 rows=13 width=8) (actual time=0.002..0.006 rows=13 loops=1)
                          ->  Hash  (cost=2.25..2.25 rows=1 width=4) (actual time=0.032..0.032 rows=3 loops=1)
                                Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                ->  Hash Join  (cost=1.10..2.25 rows=1 width=4) (actual time=0.022..0.029 rows=3 loops=1)
                                      Hash Cond: (v.taughtby = p.empid)
                                      ->  Seq Scan on course v  (cost=0.00..1.10 rows=10 width=8) (actual time=0.001..0.003 rows=10 loops=1)
                                      ->  Hash  (cost=1.09..1.09 rows=1 width=4) (actual time=0.012..0.012 rows=1 loops=1)
                                            Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                            ->  Seq Scan on professor p  (cost=0.00..1.09 rows=1 width=4) (actual time=0.006..0.010 rows=1 loops=1)
                                                  Filter: ((name)::text = 'Socrates'::text)
Total runtime: 0.185 ms
```

PostgreSQL EXPLAIN ANALYZE



Sequential scans vs. indexes

If an index is "useful" or not depends on

- How much data is relevant to the query
- Size of the relation
- Properties of the index (clustered, multiple columns,. . .)
- What algorithm needs the data as input
- . . .

Sequential scans vs. indexes

If an index is “useful” or not depends on

- How much data is relevant to the query
- Size of the relation
- Properties of the index (clustered, multiple columns,. . .)
- What algorithm needs the data as input
- . . .

Until query optimization is perfected, the main task of database administrators will remain query tuning (creating indexes, etc.).

Summary

- Query optimization is the heart of a relational DBMS
- Heuristic optimization can always be used but might potentially lead to bad plans
- Cost-based optimization relies on statistics gathered on the relations
- Database systems provide information on the “best” query execution plan (EXPLAIN)
- The database administrator needs to think of more improvements (e.g., indexes)