Electronics and
Computer Science

UNIVERSITY OF
Southampton

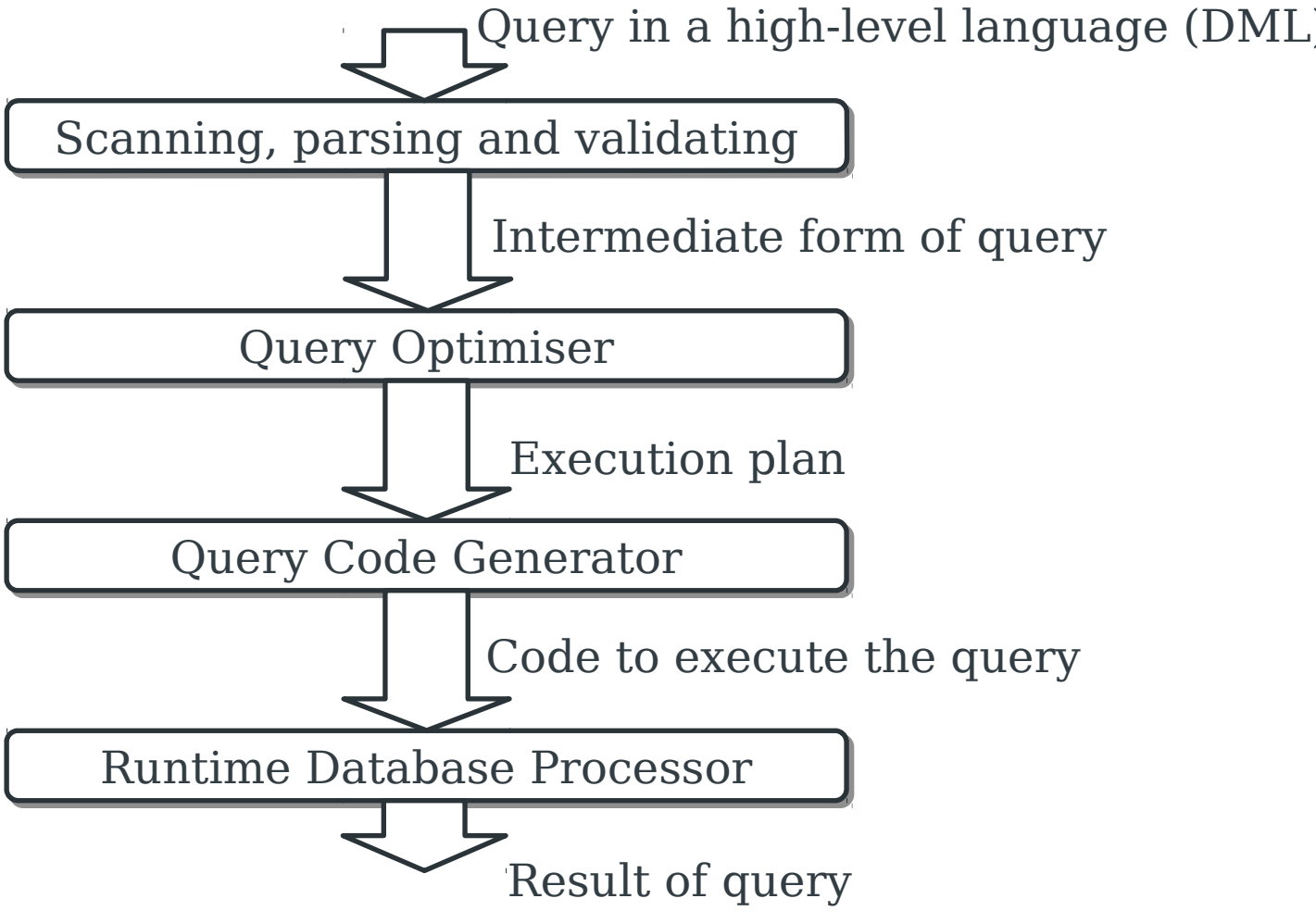
Query Processing

COMP3211 Advanced Databases

Nicholas Gibbins - nmg@ecs.soton.ac.uk
2016-2017

UNIVERSITY OF Southampton

Query Processing



UNIVERSITY OF Southampton

Query Plans

Logical Query Plan

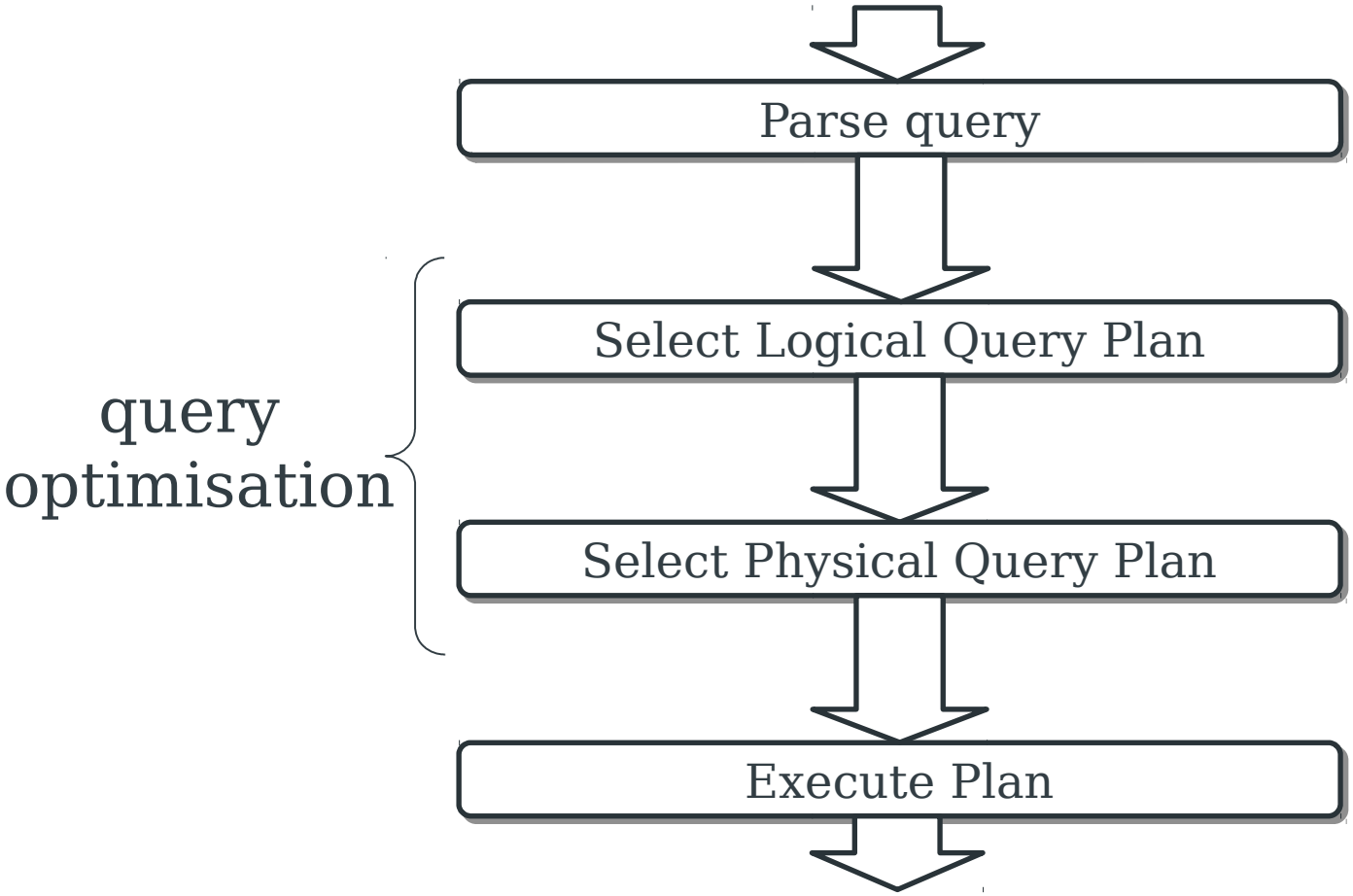
- algebraic representation of query
- operators taken from relational algebra
- abstract!

Physical Query Plan

- algorithms selected for each operator in plan
- execution order specified for operators

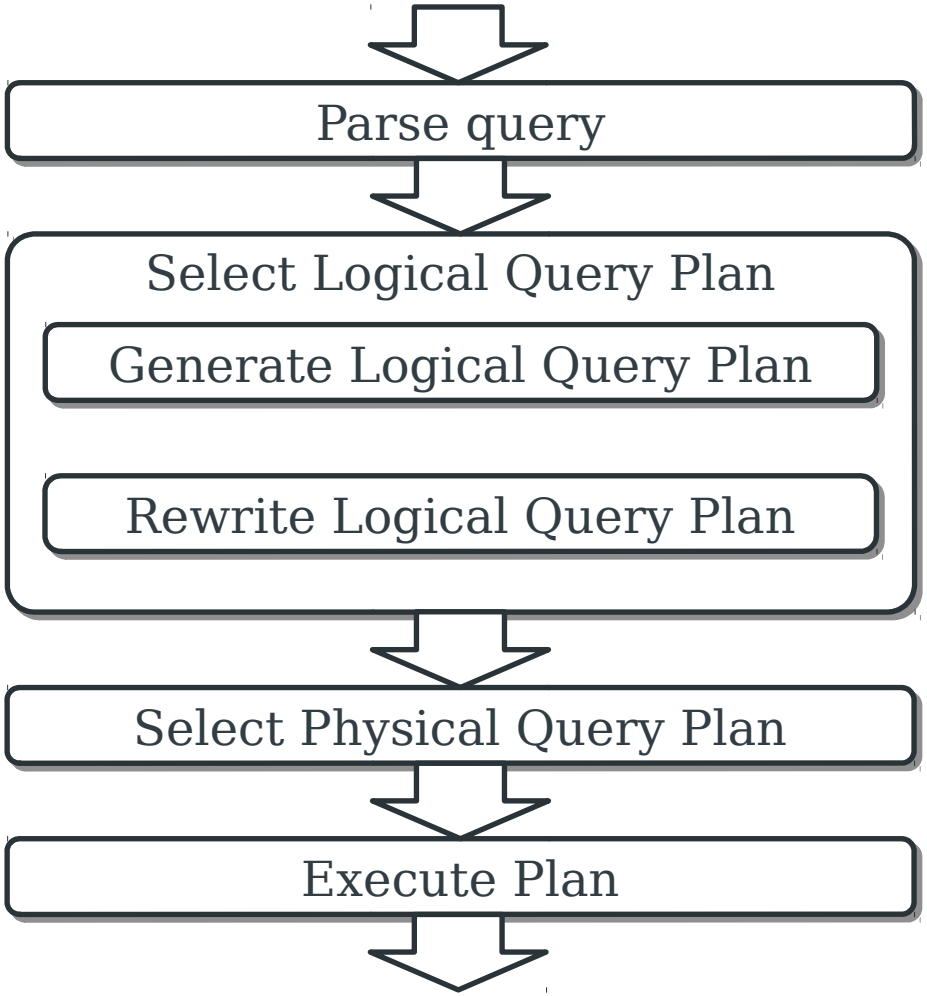
UNIVERSITY OF Southampton

Query Processing



UNIVERSITY OF Southampton

Query Processing



UNIVERSITY OF Southampton

This Week

Logical query plans

- Cost estimation
- Improving logical query plans
- Cost-based plan selection
- Join ordering

Physical query plans

- Physical query plan operators
- One-pass algorithms
- Nested-loop joins
- Two-pass algorithms
- Index-based algorithms

Optimisation

A challenge and an opportunity for relational systems

- Optimisation must be carried out to achieve performance
- Because queries are expressed at such a high semantic level, it is possible for the DBMS to work out the best way to do things

Need to start optimisation from a canonical form

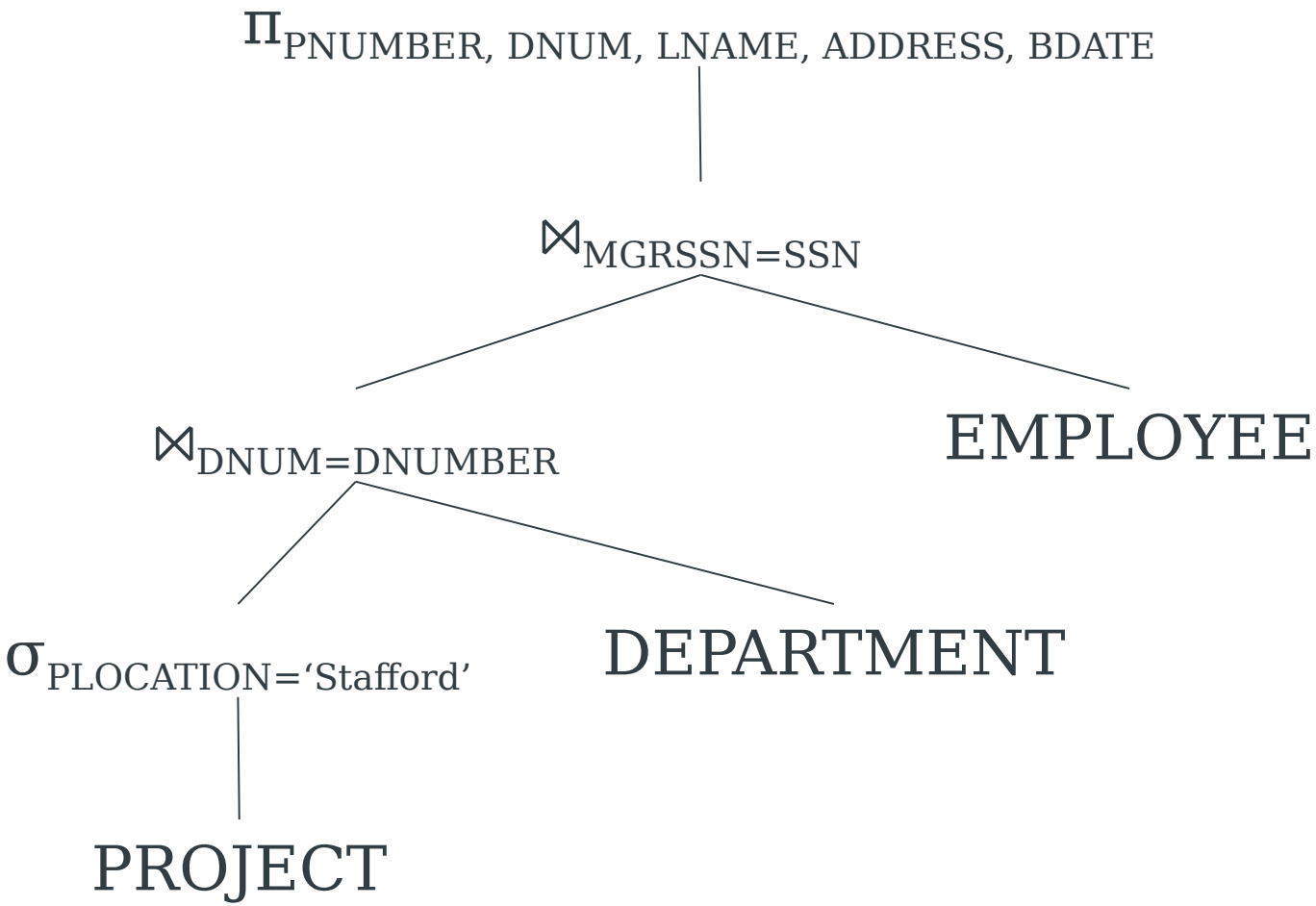
UNIVERSITY OF Southampton

Optimisation Example

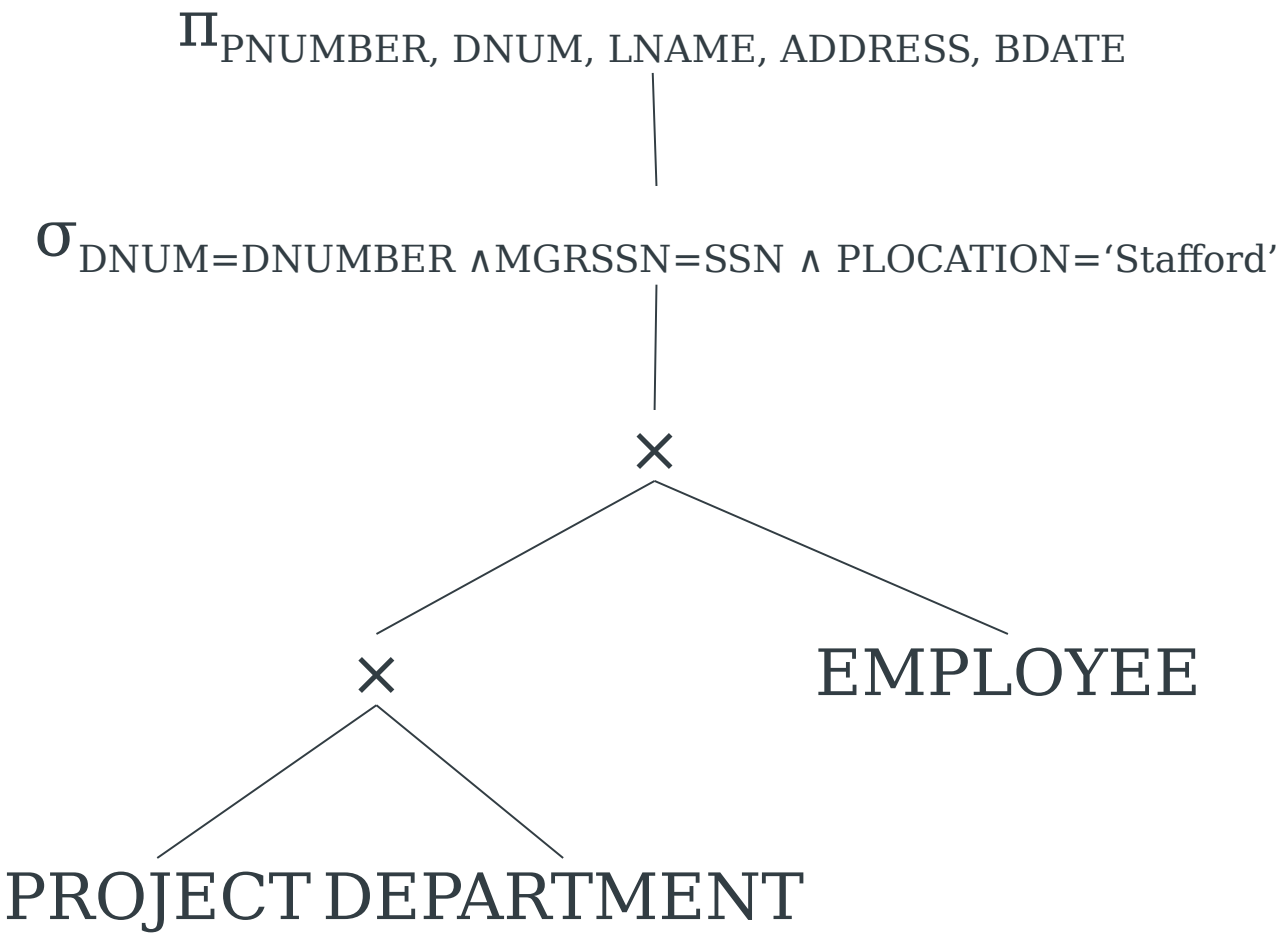
For every project located in Stafford, retrieve the project number, the controlling department number, and the department manager's last name, address and birth date

```
SELECT PNUMBER, DNUM, LNAME, ADDRESS,
DATE
FROM        PROJECT, DEPARTMENT, EMPLOYEE
WHERE DNUM=DNUMBER AND
        MGRSSN=SSN AND
        PLOCATION='Stafford'
```

Query Tree


πPNUMBER, DNUM, LNAME, ADDRESS, BDATE
⋈MGRSSN=SSN
⋈DNUM=DNUMBER
EMPLOYEE
σPLOCATION='Stafford'
DEPARTMENT
PROJECT

Canonical Form



UNIVERSITY OF Southampton

Cost Estimation

UNIVERSITY OF Southampton

Cost Estimation

At this stage, no commitment to a particular physical plan

- Estimate the “cost” of each operator in terms of the size relation(s) on which it operates
- Choose a logical query plan that minimises the size of the intermediate relations (= minimises the cost of the plan)

Assumption: system catalogue stores statistics about each relation

UNIVERSITY OF Southampton

Statistics

T(R): Number of tuples in relation R (cardinality of R)

V(R, A): Number of distinct values for attribute A in relation R

Note: for any relation R,
V(R, A) ≤ T(R) for all attributes A on R

Scan

Operation of reading all tuples of a relation

$T(\text{scan}(R)) = T(R)$

For all A in R, $V(\text{scan}(R), A) = V(R, A)$

Product

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

For all A in R, $V(R \times S, A) = V(R, A)$

For all B in S, $V(R \times S, B) = V(S, B)$

Projection

$T(\pi_A(R)) = T(R)$

For all A in R and $\pi_A(R)$, $V(\pi_A(R), A) = V(R, A)$

Assumption: projection does not remove duplicate tuples

(so value counts remain the same)

UNIVERSITY OF Southampton

Selection

Two forms to consider:

1. $\sigma_{attribute=value}(R)$
2. $\sigma_{attribute1=attribute2}(R)$

UNIVERSITY OF Southampton

Selection case 1: attr=val

$T(\sigma_{A=c}(R)) = T(R)/V(R, A)$

$V(\sigma_{A=c}(R), A) = 1$

Assumption: all values of A appear with equal frequency

Selection case 2: attr=attr

$T(\sigma_{A=B}(R)) = T(R)/\max(V(R, A),V(R, B))$

$V(\sigma_{A=B}(R), A) = V(\sigma_{A=B}(R), B) = \min(V(R, A), V(R, B))$

Assumption: all values of A appear with equal frequency

Assumption: all values of B appear with equal frequency

Note: for all other attributes X of R, $V(\sigma_{A=B}(R), X) = V(R, X)$

This may be reduced because $V(\sigma_{A=B}(R), X) \leq$

UNIVERSITY OF Southampton

Further Selection: Inequality

Selections involving inequalities and not equals require a more nuanced approach

Typical inequality written to match less half of a relation:

$T(\sigma_{A<c}(R)) = T(R)/3$ as a rule of thumb

$T(\sigma_{A\neq c}(R)) = T(R)$ as a first approximation

Alternatively,
$T(\sigma_{A\neq c}(R)) = T(R)(V(R, A) - 1)/V(R, A)$

Further Selection: Conjunction/Disjunction

$$T(\sigma_{A=c1 \wedge B=c2}(R)) = \frac{T(R)}{V(R,A)V(R,B)}$$

$$T(\sigma_{A=c1 \vee B=c2}(R)) = \frac{T(R)}{V(R,A)} + \frac{T(R)}{V(R,B)}$$

(overestimates number of tuples)

Alternatively,

$$T(\sigma_{A=c1 \vee B=c2}(R)) = T(R)\left(1 - \frac{1}{V(R,A)}\right)\left(1 - \frac{1}{V(R,B)}\right)$$

Join

$T(R \bowtie_{A=B} S) = T(R)T(S)/\max(V(R,A),V(S,B))$

$V(R \bowtie_{A=B} S, A) = V(R \bowtie_{A=B} S, B) = \min(V(R, A), V(S, B))$

Assumption: all values of A and B appear with equal frequency

Note: for all other attributes X of R and Y of S,
$V(R \bowtie_{A=B} S, X) = V(R,X)$ and $V(R \bowtie_{A=B} S, Y) = V(S, Y)$

This may be reduced because $V(R \bowtie_{A=B} S, X) \leq T(R \bowtie_{A=B} S)$
and $V(R \bowtie_{A=B} S, Y) \leq T(R \bowtie_{A=B} S)$

Further Join

If there are multiple pairs of join attributes:

$$T(R \bowtie_{R1=S1 \wedge R2=S2} S) = \frac{T(R)T(S)}{\max(V(R, R1),V(S, S1))\ \max(V(R, R2),V(S, S2))}$$

Further Statistics

Distinct values assumes that each attribute value appears with equal frequency

- Potentially unrealistic
- Gives inaccurate estimates for joins and selects

Other approaches based on histograms:

- Equal-width – divide the attribute domain into equal parts, give tuple counts for each
- Equal-height – sort tuples by attribute, divide into equal-sized sets of tuples and give maximum value for each set
- Most-frequent values – give tuple counts for top-n most frequent values

UNIVERSITY OF Southampton

Query Optimisation

Heuristic Approach

1. Start with canonical form
2. Move σ operators down the tree
3. Reorder subtrees to put most restrictive σ first
4. Combine × and σ to create ⋈
5. Move π operators down the tree

Optimising Query Trees

Find the last names of employees born after 1957 who work on a project named ‘Aquarius’

```
SELECT LNAME
FROM       EMPLOYEE, WORKS_ON, PROJECT
WHERE PNAME=‘Aquarius’ AND
        PNUMBER=PNO AND
        ESSN=SSN AND
        BDATE > ‘1957-12-31’
```

UNIVERSITY OF Southampton

Tree Structures and Canonical Form

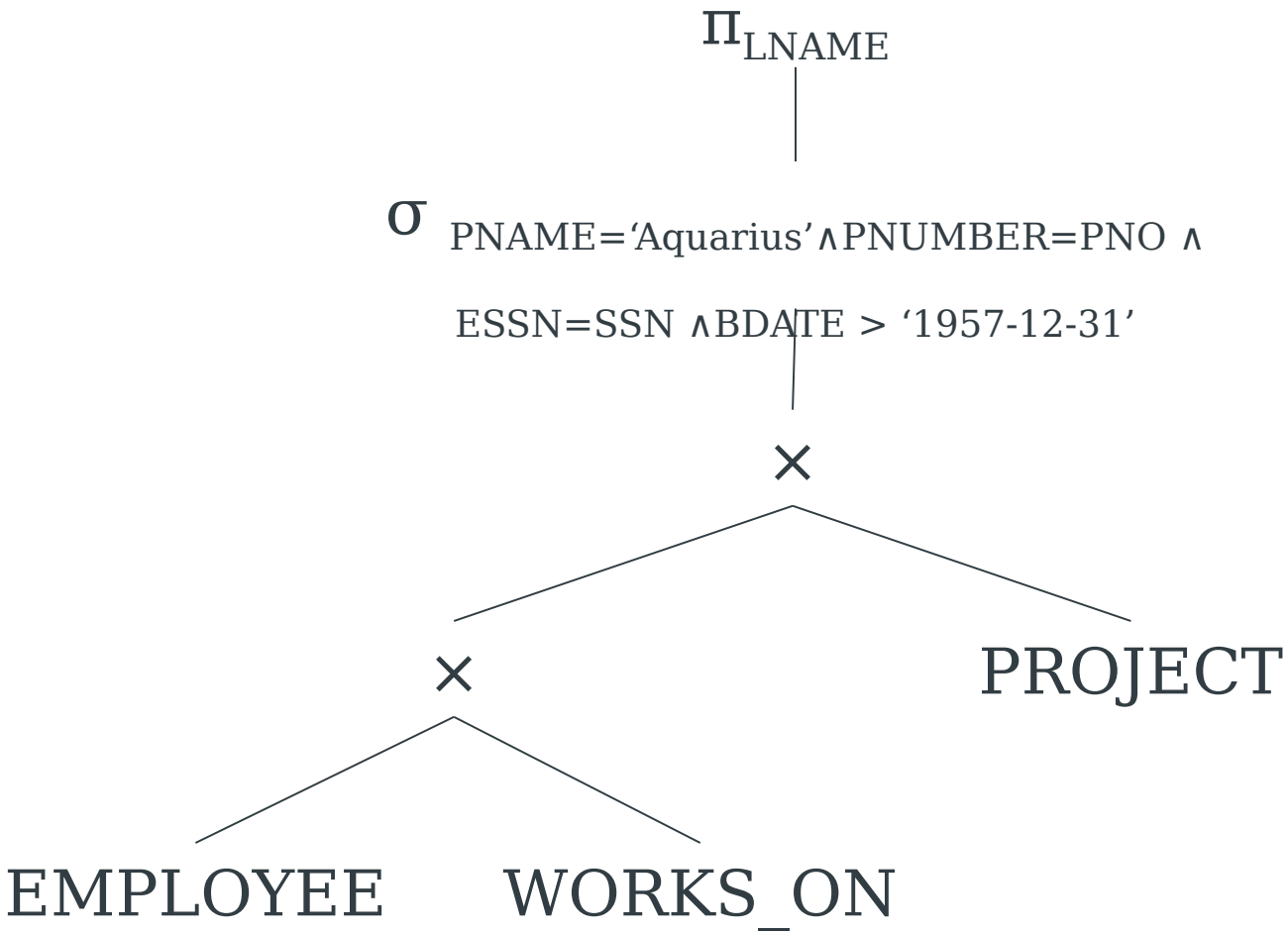
Useful to only consider left-deep trees

- Fewer possible left-deep trees than possible bushy trees - smaller search space when investigating join orderings
- Left deep trees work well with common join algorithms (nested-loop, index, one-pass – about which more later)

Canonical form should be:

1. a left-deep tree of products with
2. a conjunctive selection above the products and
3. a projection (of the output attributes) above the selection

Canonical Form



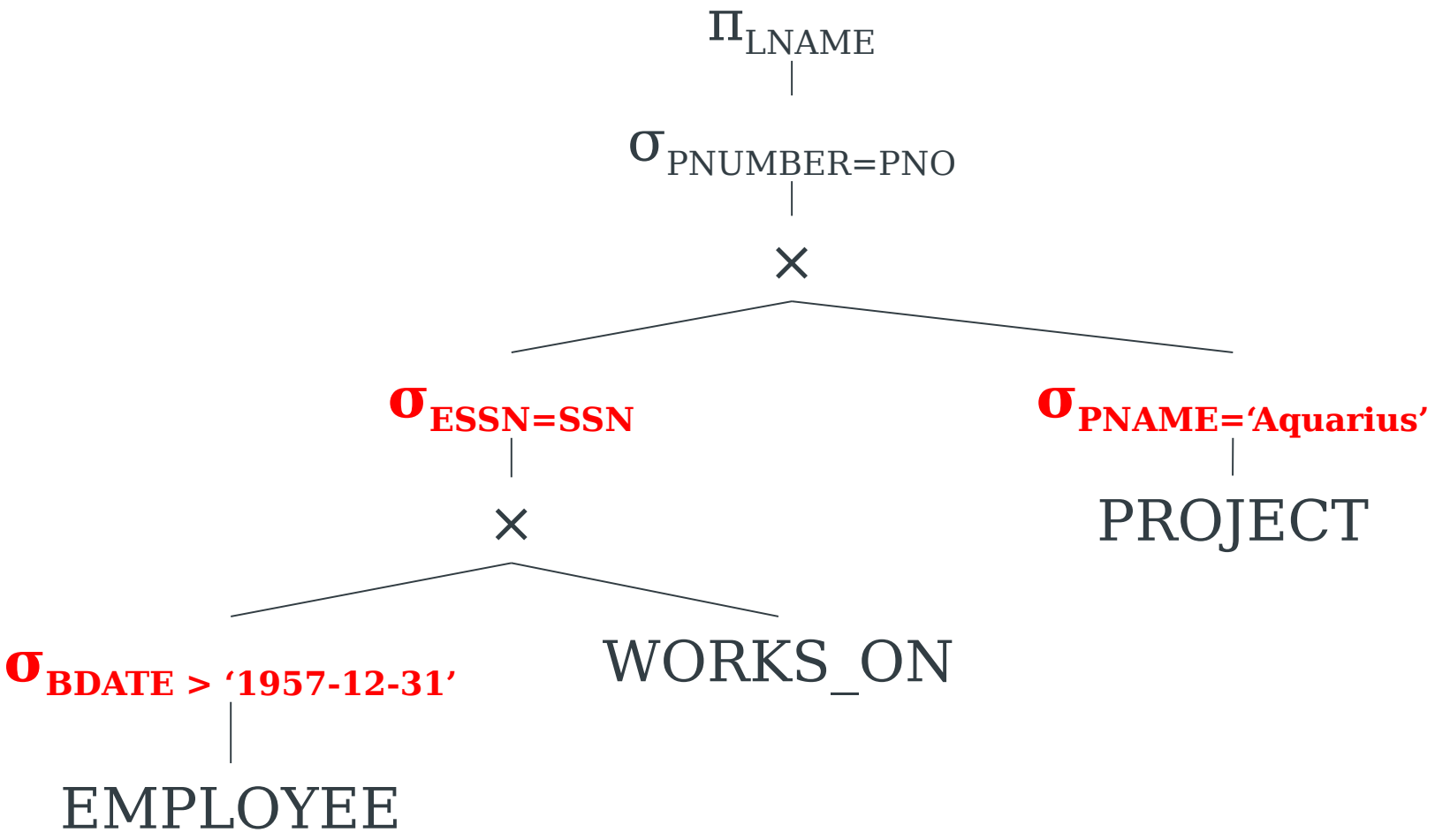
UNIVERSITY OF Southampton

Move σ down

A selection of the form $\sigma_{attr=val}$ can be pushed down to just above the relation that contains attr

A selection of the form $\sigma_{attr1=attr2}$ can be pushed down to the product above the subtree containing the relations that contain attr1 and attr2

Move σ down


π_{LNAME}
$\sigma_{PNUMBER=PNO}$
×
$\sigma_{ESSN=SSN}$
$\sigma_{PNAME='Aquarius'}$
×
PROJECT
$\sigma_{BDATE > '1957-12-31'}$
WORKS_ON
EMPLOYEE

Reorder Joins

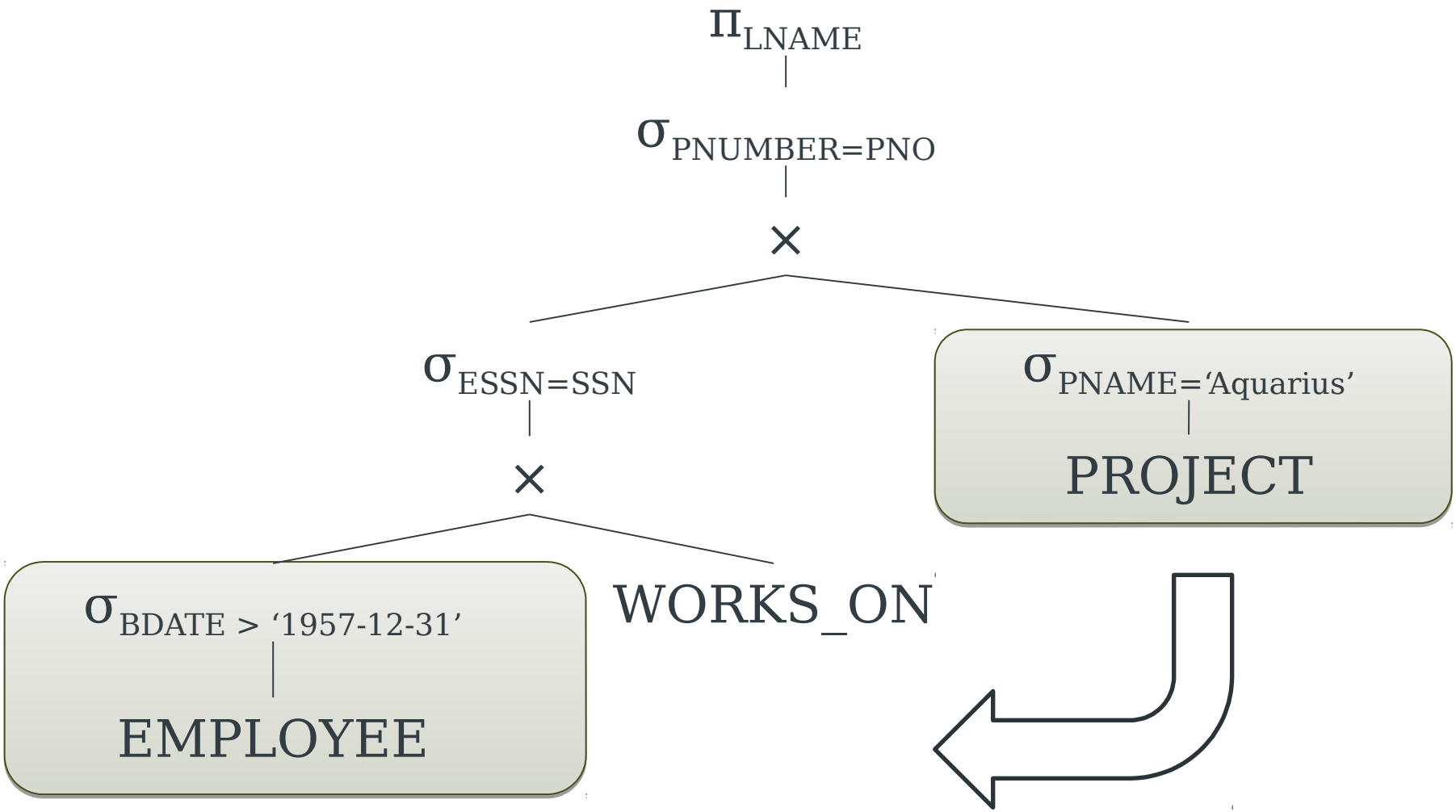
If a query joins N relations and we restrict ourselves only to left-deep trees, there are N! possible join orderings

- Far more possible orderings if don't restrict to left-deep

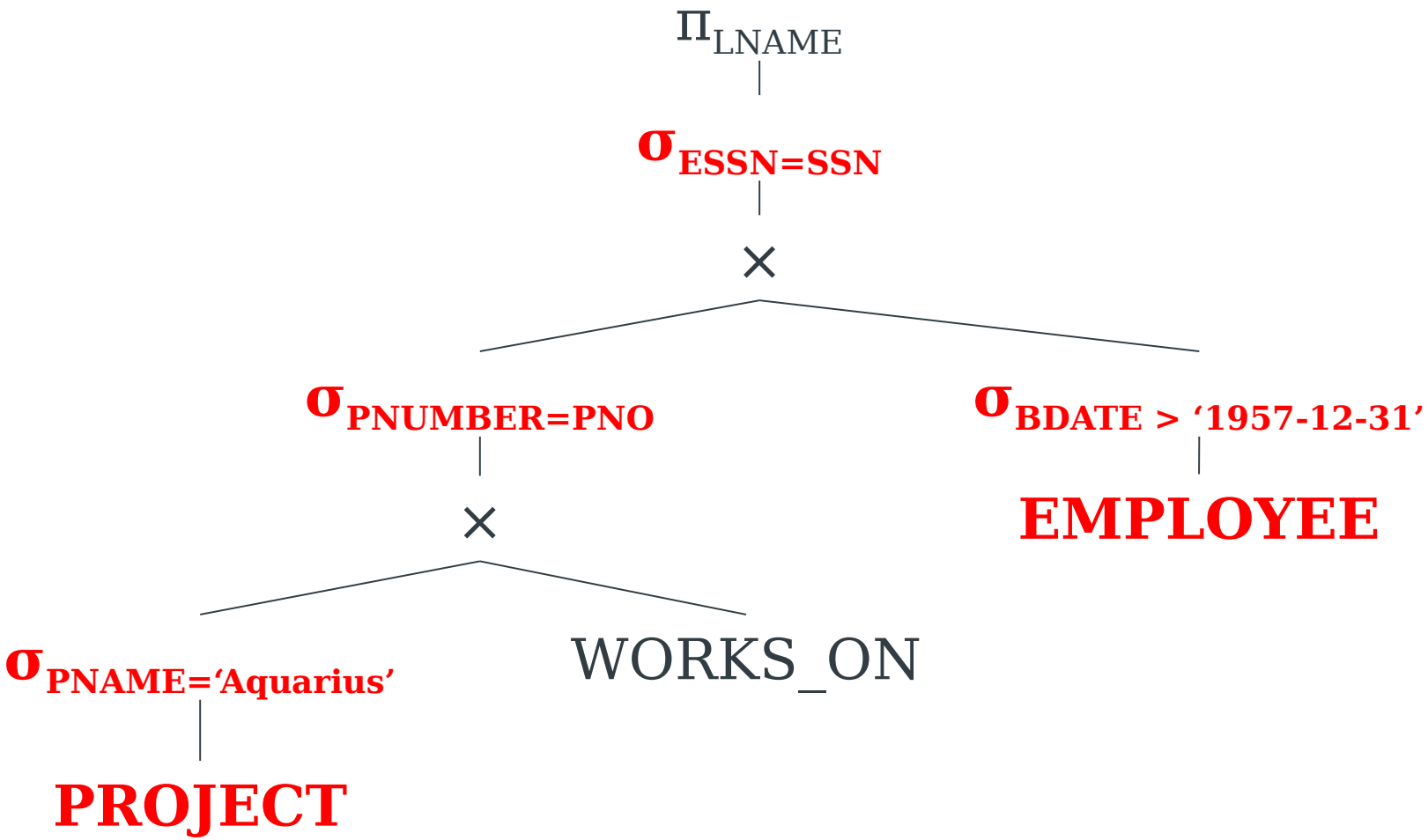
For simplicity of search, adopt a greedy approach:

Reorder subtrees to put the most restrictive relations (fewest tuples) first

Reorder Joins



Reorder Joins



UNIVERSITY OF Southampton

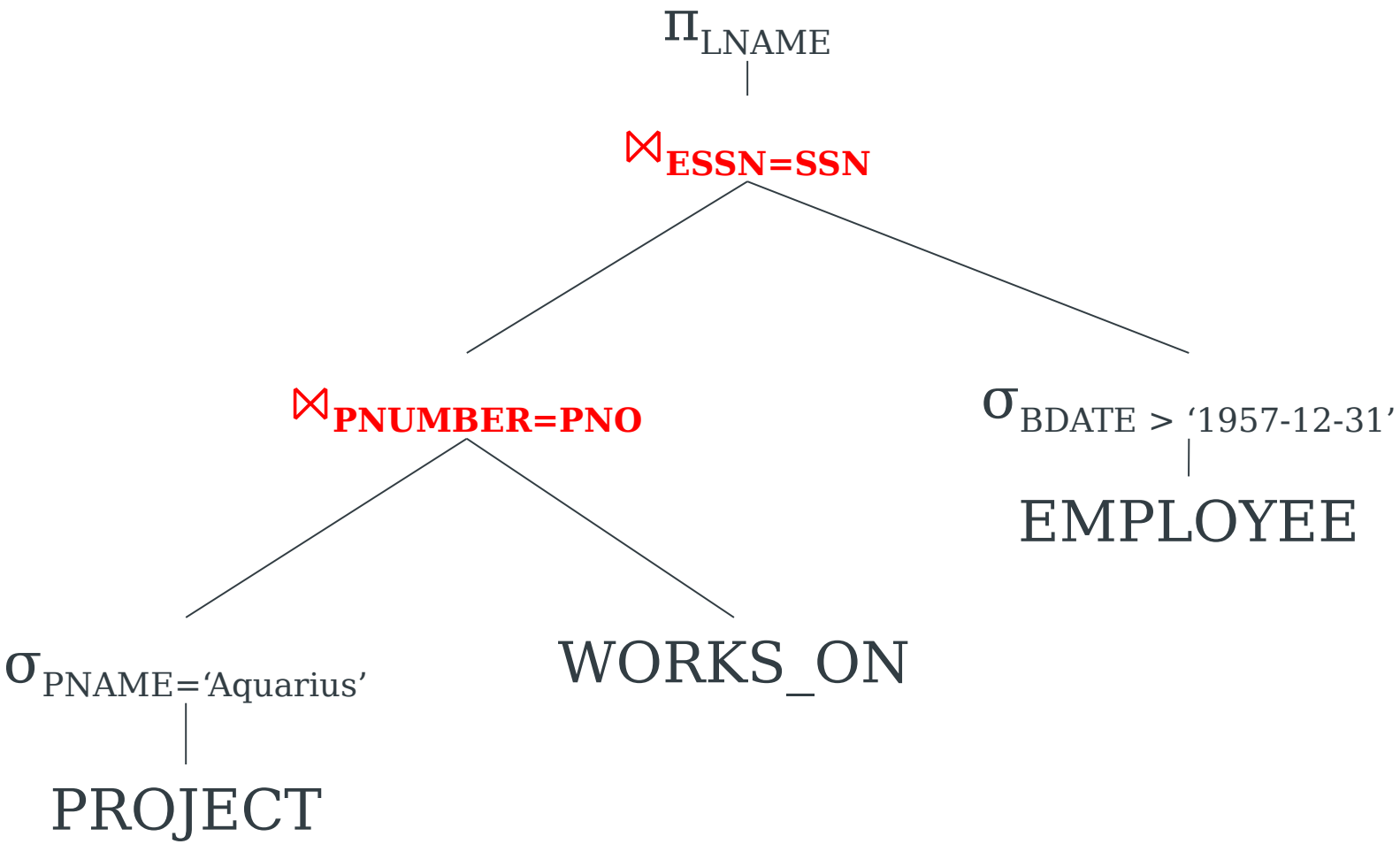
Create Joins

Combine $\times$ with a σ immediately above to form $\bowtie$

Uses the relational transformation $\sigma_a(R \times S) \equiv R \bowtie_a S$

Join more efficient than product followed by selection!

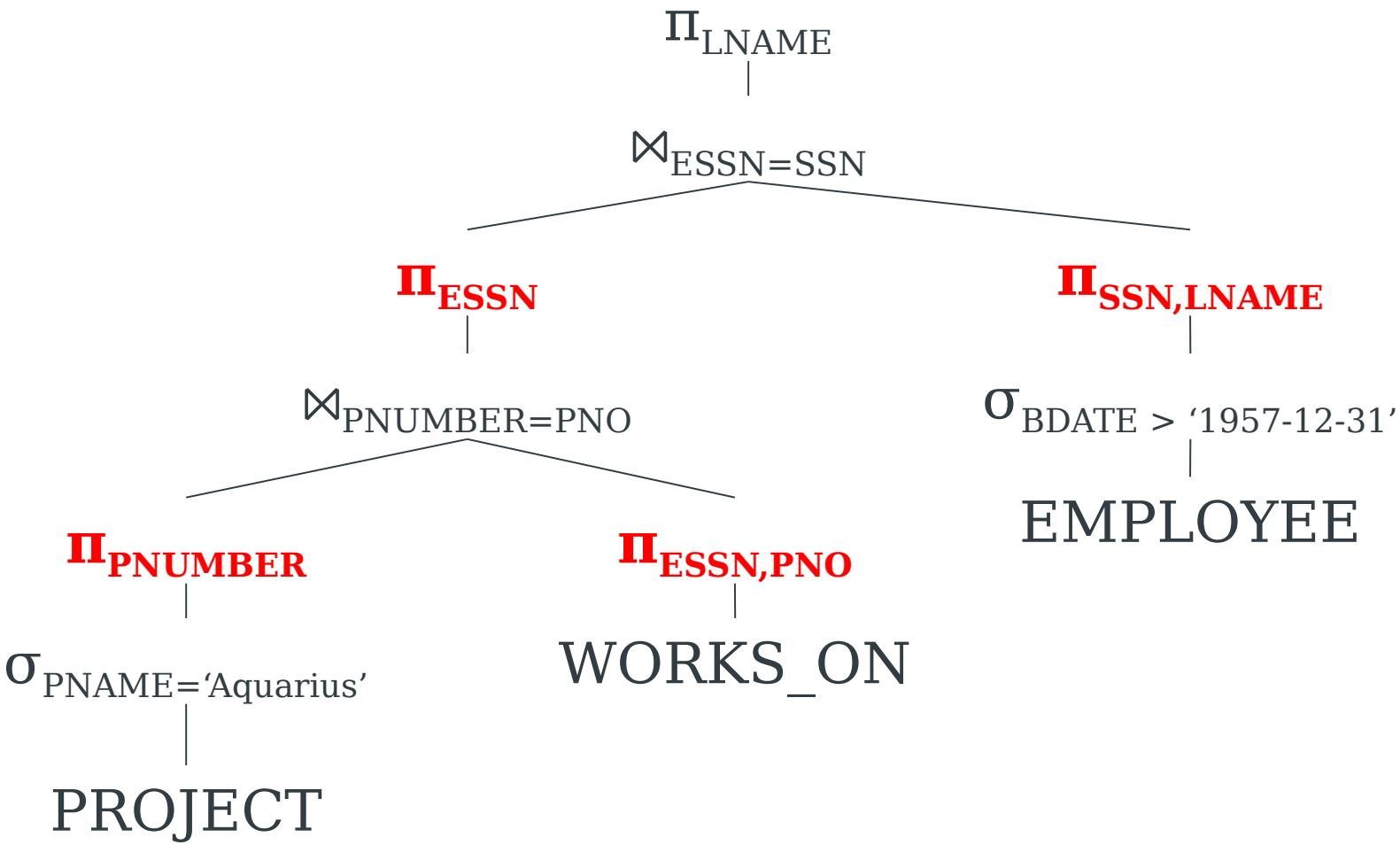
Create Joins



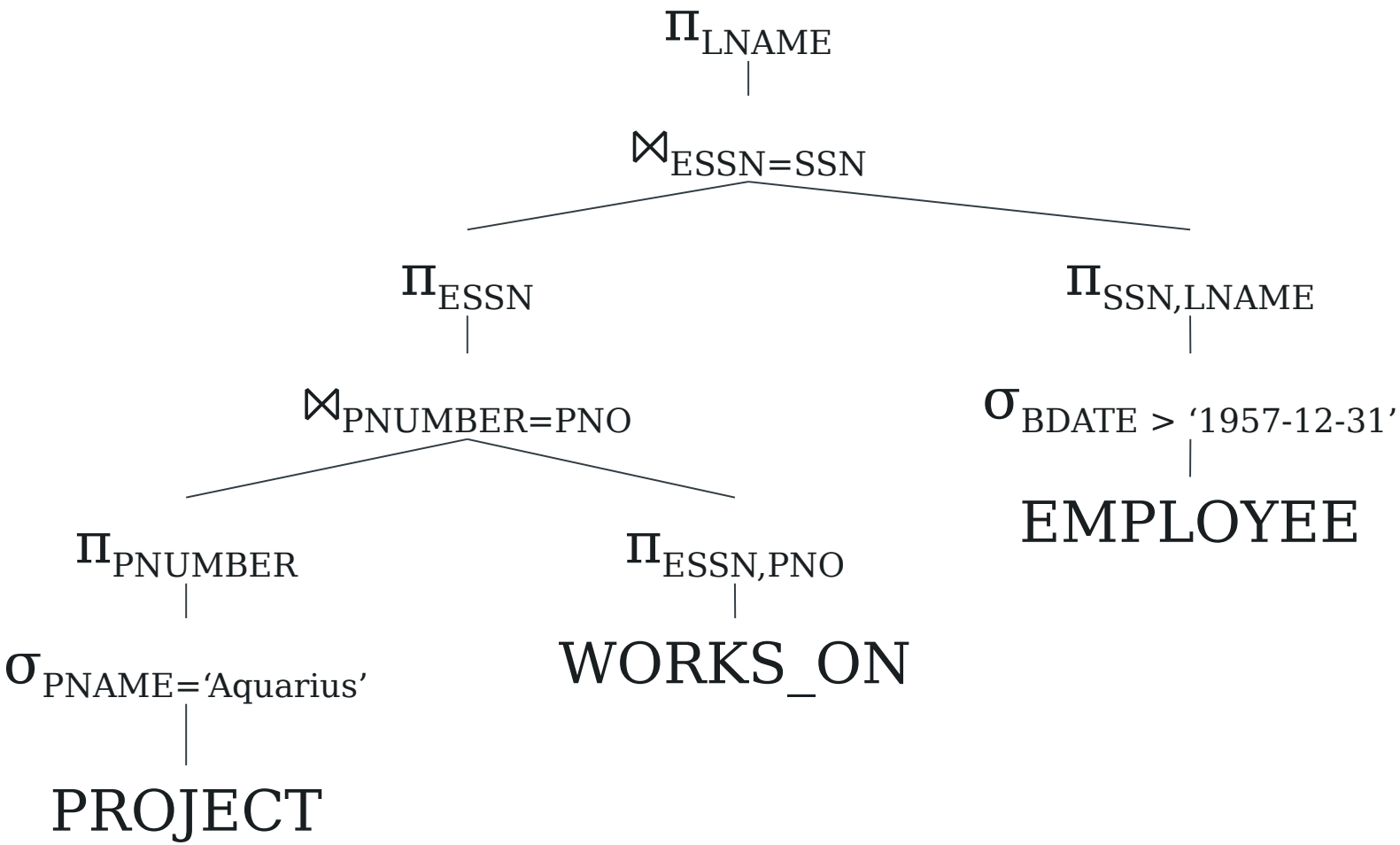
Move π down

If intermediate relations are to be kept in buffers, reducing the *degree* of those relations (= number of attributes) allows us to use fewer buffer frames

Move π down


π LNAME
⋈ ESSN=SSN
π ESSN
π SSN,LNAME
⋈ PNUMBER=PNO
σ BDATE > '1957-12-31'
EMPLOYEE
π PNUMBER
π ESSN,PNO
σ PNAME='Aquarius'
WORKS_ON
PROJECT

Optimised logical query plan



UNIVERSITY OF Southampton

Physical-Plan Operators

Physical-Plan Operators

Algorithm that implements one of the basic relational operations that are used in query plans

For example, relational algebra has join operator

How that join is carried out depends on:

- structure of relations
- size of relations
- presence of indexes and hashes
- ...

Computation Model

Need to choose good physical-plan operators

- Estimate the “cost” of each operator
- Key measure of cost is the number of disk accesses (far more costly than main memory accesses)

Assumption: arguments of operator are on disk, result is in main memory

Cost Parameters

M: Main memory available for buffers

S(R): Size of a tuple of relation R

B(R): Blocks used to store relation R

T(R): Number of tuples in relation R (cardinality of R)

V(R,a):Number of distinct values for attribute a in relation R

UNIVERSITY OF Southampton

Clustered File

Tuples from different relations that can be joined (on particular attribute values) stored in blocks together

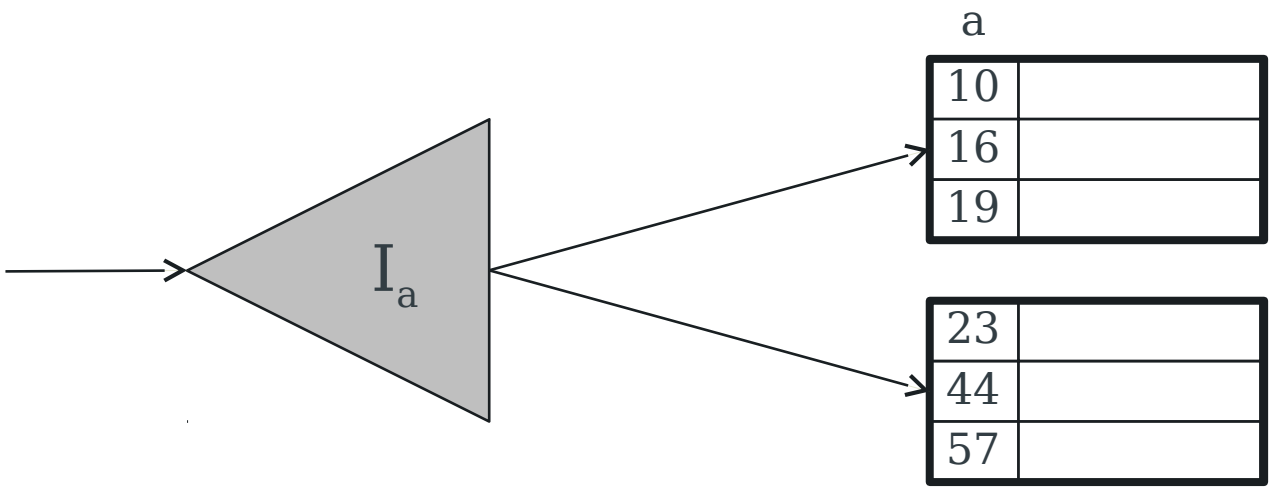
UNIVERSITY OF Southampton

Clustered Relation

Tuples from relation are stored together in blocks, but not necessarily sorted

Clustering Index

Index that allows tuples to be read in an order that corresponds to physical order



UNIVERSITY OF Southampton

Scanning

UNIVERSITY OF Southampton

Scan

- Read all of the tuples of a relation R
- Read only those tuples of a relation R that satisfy some predicate

Two variants:

- Table scan
- Index scan

UNIVERSITY OF Southampton

Table Scan

Tuples arranged in blocks

- All blocks known to the system
- Possible to get blocks one at a time

I/O Cost

- B(R) disk accesses, if R is clustered
- T(R) disk accesses, if R is not clustered

University of Southampton

Index Scan

An index exists on **some** attribute of R

- Use index to find all blocks holding R
- Retrieve blocks for R

I/O Cost

- B(R) + B(I_R) disk accesses if clustered
- B(R) >> B(I_R), so treat as only B(R)
- T(R) disk accesses if not clustered

UNIVERSITY OF Southampton

One-Pass Algorithms

UNIVERSITY OF Southampton

One-Pass Algorithms

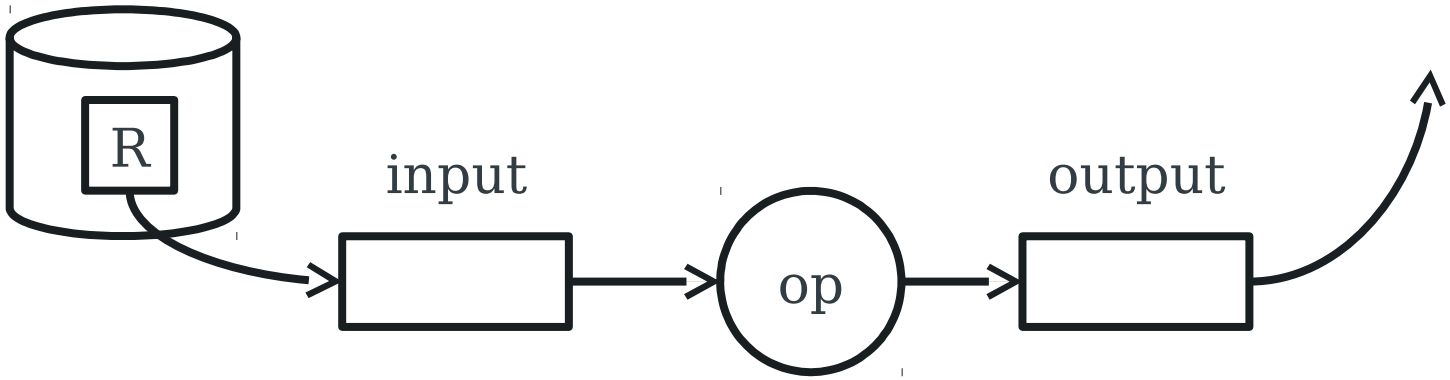
- Read data from disk only once
- Typically require that at least one argument fits in main memory

- Three broad categories:
  - Unary, tuple at a time (i.e. select, project)
  - Unary, full-relation (i.e. duplicate elimination, grouping)
  - Binary, full-relation

UNIVERSITY OF Southampton

Unary, tuple at a time

foreach block of R:
    copy block to input buffer
    perform operation (select, project) on each tuple in block
    move selected/projected tuples to output buffer



Unary, tuple at a time: Cost

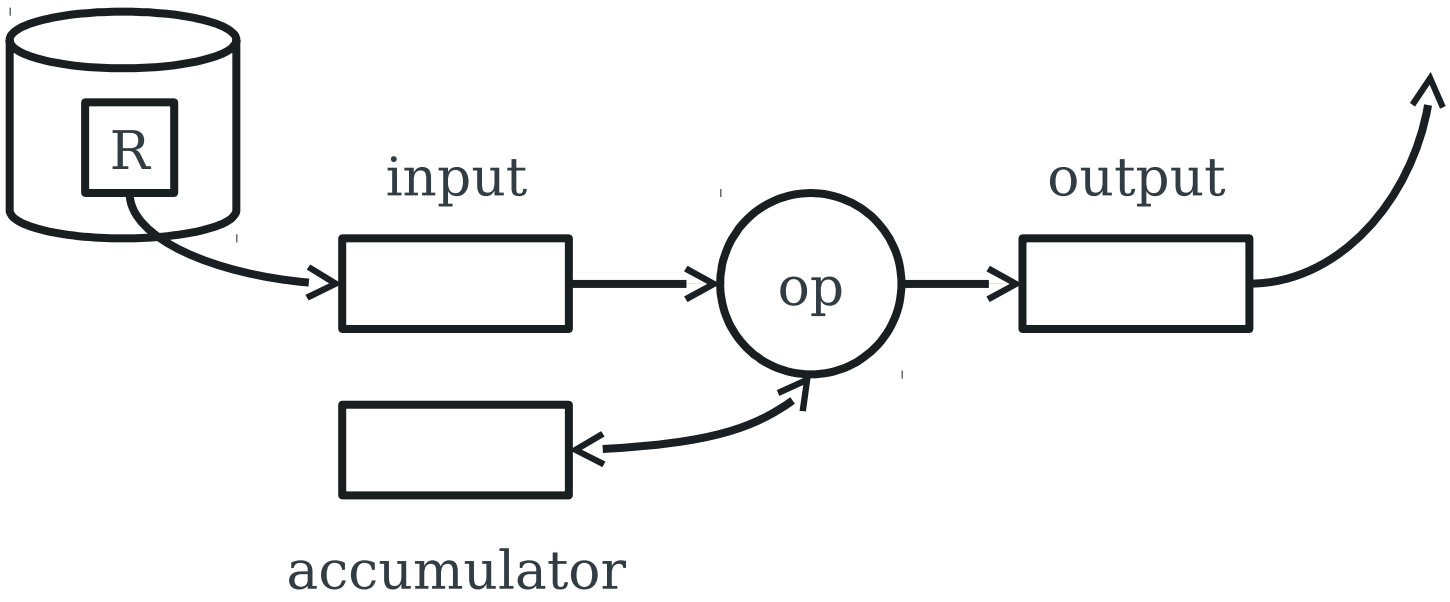
In general, B(R) or T(R) disk accesses depending on clustering

If operator is a select that compares an attribute to a constant and index exists for attributes used in select, <<B(R) disk accesses

Requires M>=1

Unary, full-relation

foreach block of R
    copy block to input buffer
    update accumulator
    move tuples to output buffer



Unary, full-relation: Duplicate elimination

```
foreach block of R:
    copy block to input buffer
    foreach tuple in block
        if tuple is not in accumulator
            copy to accumulator
            copy to output buffer
```

Unary, full-relation: Duplicate elimination

Requires M ≥ B(δ(R)) + 1 blocks of main memory

- 1 block for input buffer
- M-1 blocks for accumulator (records each tuple seen so far)

- Accumulator implemented as in-memory data structure (tree, hash)
- If fewer than B(δ(R)) blocks available, thrashing likely
- Cost is B(R) disk accesses

UNIVERSITY OF Southampton

Unary, full-relation: Grouping

Grouping operators: min, max, sum, count, avg

- Accumulator contains per-group values
- Output only when all blocks of R have been consumed
- Cost is B(R) disk accesses

Binary, full-relation

Union, intersection, difference, product, join

- We'll consider join in detail

In general, cost is B(R) + B(S)

- R, S are operand relations

Requirement for one pass operation: min(B(R), B(S)) ≤ M-1

UNIVERSITY OF Southampton

Binary, full-relation: Join

- Two relations, R(X,Y) and S(Y,Z), B(S)<B(R)
- Uses main memory search structure keyed on Y

```
foreach block of S:
    read block
    add tuples to search structure
foreach block of R
    copy block to input buffer
    foreach tuple in block
        find matching tuples in search structure
        construct new tuples and copy to output
```

UNIVERSITY OF Southampton

Nested-Loop Joins

University of Southampton

Nested-loop join

Also known as iteration join

Assuming that we're joining relations R,S on attribute C:

```
foreach tuple r in R
    foreach tuple s in S
        if r.C = s.C then output r,s pair
```

Factors that affect cost

- Tuples of relation stored physically together? (clustered)
- Relations sorted by join attribute?
- Indexes exist?

Example

Consider a join between relations R1, R2 on attribute C:

T(R1) = 10,000

T(R2) = 5,000

S(R1) = S(R2) =1/10 block

M = 101 blocks

Attempt #1: Tuple-based nested loop join

- Relations not contiguous
  - One disk access per tuple

Cost for each tuple in R1 = cost to read tuple + cost to read R2

Total Cost = T(R1) * (1 + T(R2))
= 10,000 * (1+5,000)
= 50,010,000 disk accesses

Can we do better?

Use all available main memory (M=101)

Read outer relation R1 in chunks of 100 blocks

Read all of inner relation R2 (using 1 block) + join

Attempt #2: block-based nested loop join

- Cost to read one 100-block chunk of R1 = 100 * 1/S(R1)
  = 1,000 disk accesses

Cost to process each chunk = 1000 + T(R2) = 6,000

Total cost = T(R1) / 1,000 * 6,000 = 60,000 disk accesses

Can we do better?

What happens if we reverse the join order?

- R1 becomes the inner relation
- R2 becomes the outer relation

Attempt #3: Join reordering

Cost to read one 100-block chunk of R2 = 100 * 1/S(R2)

= 1,000 disk accesses

Cost to process each chunk = 1000 + T(R1) = 11,000

Total cost = T(R1) / 1,000 * 11,000 = 55,000 disk accesses

UNIVERSITY OF Southampton

Can we do better?

What happens if the tuples in each relation are contiguous
(i.e. clustered)

Attempt #4: Contiguous relations

B(R1) = 1,000
B(R2) = 500

Cost to read one 100-block chunk of R2 = 100 disk accesses

Cost to process each chunk = 100 + B(R1) = 1,100

Total cost = B(R2) / 100 * 1,100 = 5,500 disk accesses

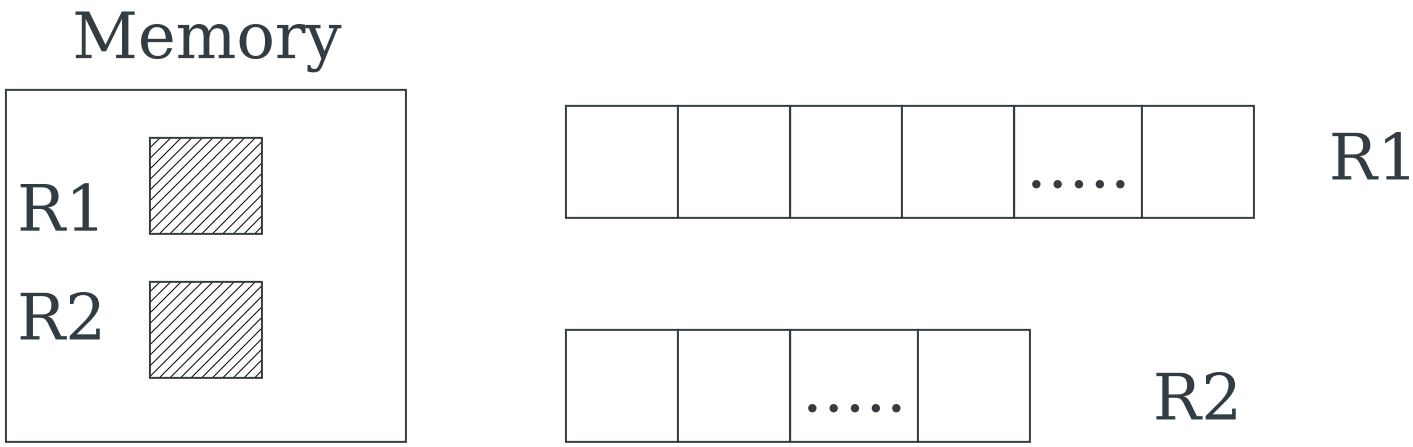
UNIVERSITY OF Southampton

Can we do better?

What happens if both relations are contiguous **and** sorted by C, the join attribute?

Attempt #5: Merge join

Total cost = B(R1) + B(R2)
= 1,000 + 500 = 1,500 disk accesses



UNIVERSITY OF Southampton

Two-Pass Algorithms

UNIVERSITY OF Southampton

Can we do better?

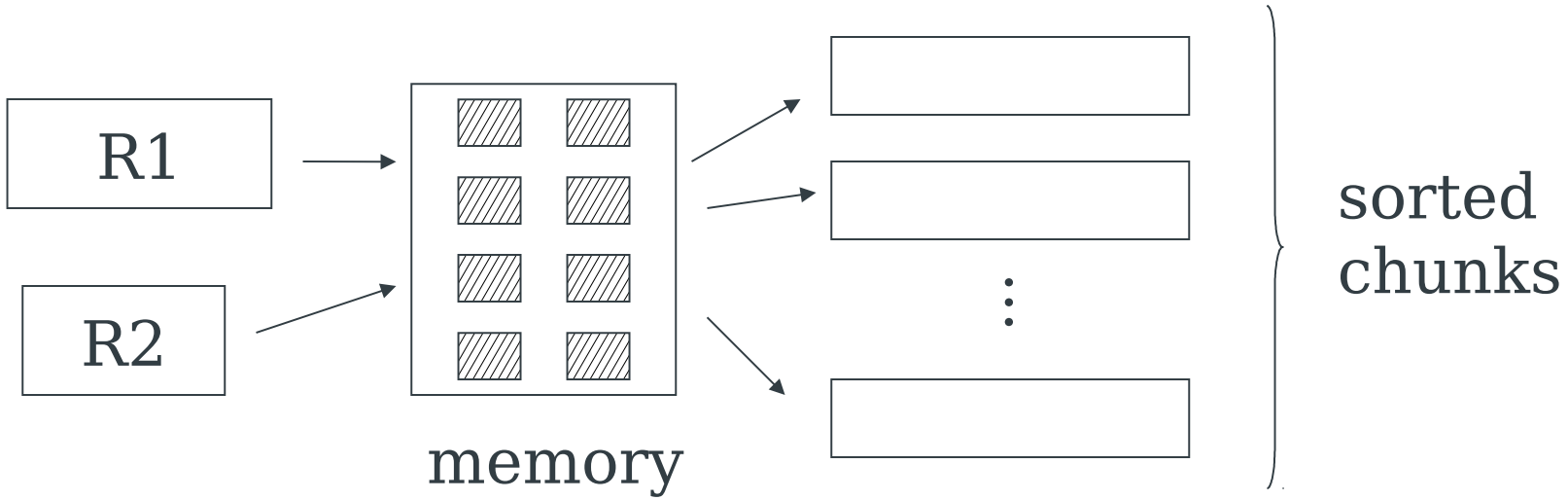
What if R1 and R2 *aren't* sorted by C?

...need to sort R1 and R2 first

Merge Sort

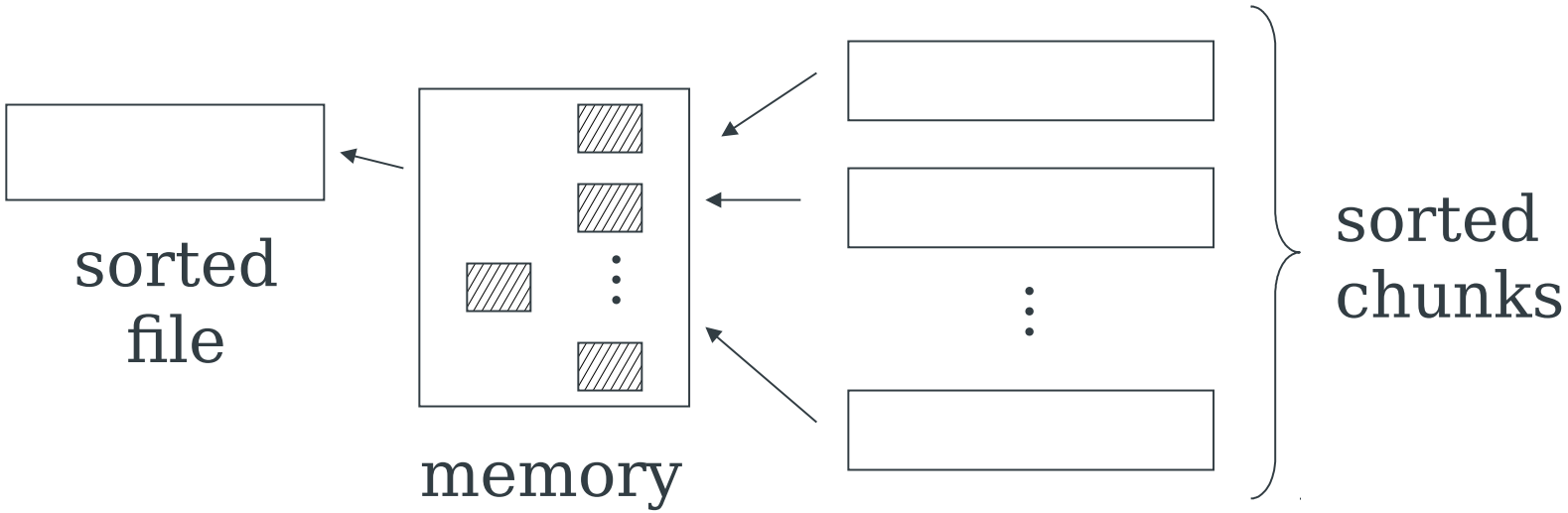
(i) For each 100 block chunk of R:

- Read chunk
- Sort in memory
- Write to disk



Merge Sort

(ii) Read all chunks + merge + write out



UNIVERSITY OF Southampton

Merge Sort: Cost

Each tuple is read, written, read, written

Sort cost R1: 4 x 1,000 = 4,000 disk accesses

Sort cost R2: 4 x 500 = 2,000 disk accesses

UNIVERSITY OF Southampton

Attempt #6: Merge join with sort

R1, R2 contiguous, but unordered

Total cost = sort cost + join cost
= 6,000 + 1,500
= 7,500 disk accesses

Nested loop cost = 5,500 disk accesses

- Merge join does not necessarily pay off

Attempt #6, part 2

If R1 = 10,000 blocks contiguous
R2 = 5,000 blocks not ordered

Nested loop cost = (5,000/100) * (100 + 10,000)
= 505,000 disk accesses

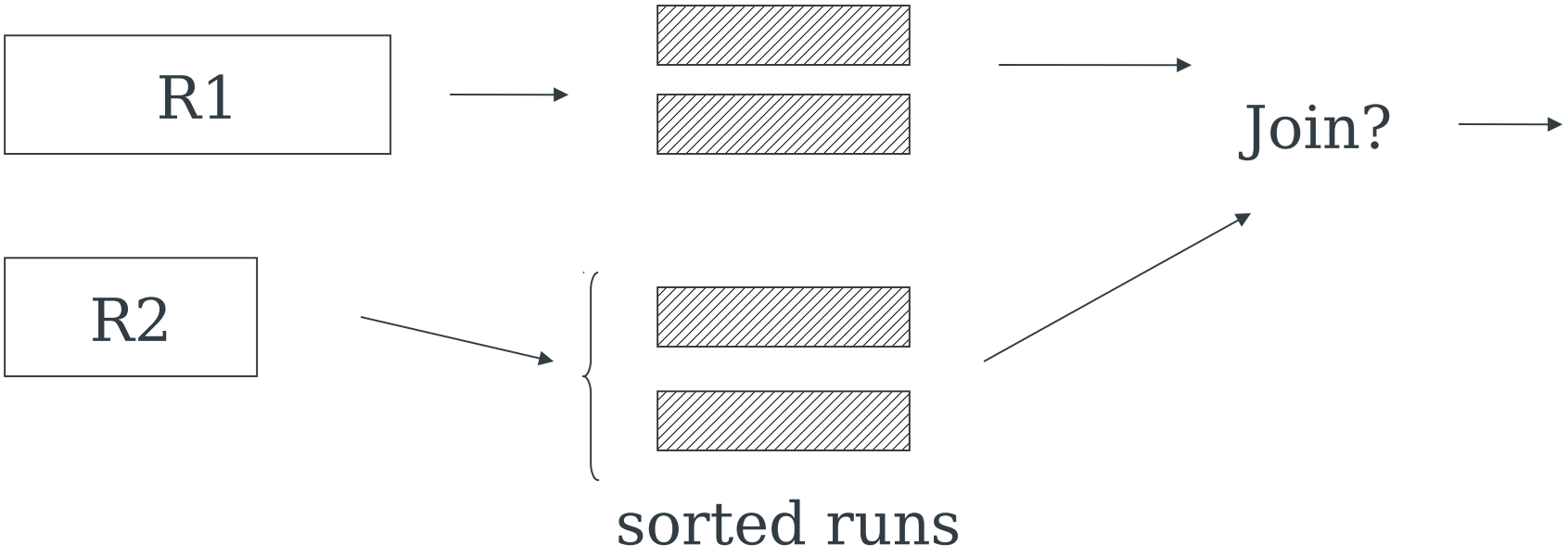
Merge join cost = 5 * (10,000+5,000)
= 75,000 disk accesses

In this case, merge join (with sort) is better

UNIVERSITY OF Southampton

Can we do better?

Do the entire files need to be sorted?



Attempt #7: Improved merge join

1. Read R1 + write R1 into runs
2. Read R2 + write R2 into runs
3. Merge join

Total cost = 2,000 + 1,000 + 1,500 = 4,500 disk accesses

Two-pass Algorithms using Hashing

Partition relation into M-1 buckets

In general:

- Read relation a tuple at a time
- Hash tuple to bucket
- When bucket is full, move to disk and reinitialise bucket

Hash-Join

The tuples in R1 and R2 are both hashed using the same hashing function on the join attributes

1. Read R1 and write into buckets
2. Read R2 and write into buckets
3. Join R1, R2

Total cost = 3 * (B(R1) + B(R2))
= 3 * (1,000 + 500)
= 4,500 disk accesses

UNIVERSITY OF Southampton

Index-based Algorithms

Can we do better?

What if we have an index on the join attribute?

- Assume R2.C index exists; 2 levels
- Assume R1 contiguous, unordered
- Assume R2.C index fits in memory

UNIVERSITY OF Southampton

Attempt #8: Index join

Cost: Reads: 500 disk accesses

foreach R1 tuple:

- probe index – free
- if match, read R2 tuple: 1 disk access

UNIVERSITY OF Southampton

How many matching tuples?

(a) If R2.C is key, R1.C is foreign key

expected number of matching tuples = 1

UNIVERSITY OF Southampton

How many matching tuples?

(b) If V(R2,C) = 5000, T(R2) = 10,000 and uniform assumption,

expected matching tuples = 10,000/5,000 = 2

UNIVERSITY OF Southampton

How many matching tuples?

(c) Assume DOM(R2, C)=1,000,000, T(R2) = 10,000 with alternate assumption

expected matching tuples = 10,000 /1,000,000 = 1/100

Attempt #8: Index join

(a) Cost = 500+5000 * 1 * 1 = 5,500 disk accesses

(b) Cost = 500+5000 * 2 * 1 = 10,500 disk accesses

(c) Cost = 500+5000 * 1/100 * 1 = 550 disk accesses

UNIVERSITY OF Southampton

Summary

Summary

use heuristics

choose cheapest