University of
Southampton

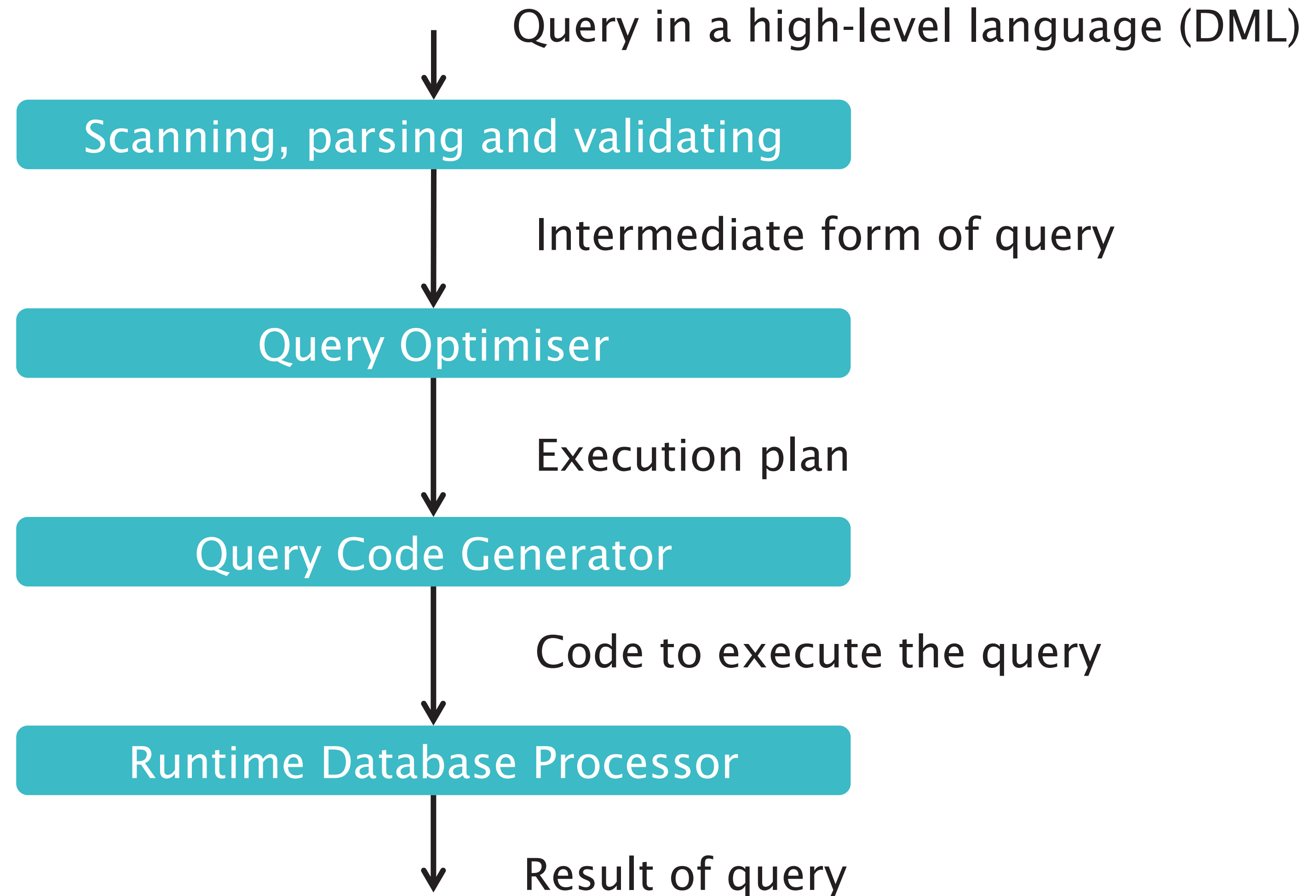
University of Southampton

# Query Processing

COMP3211 Advanced Databases

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

# Query Processing



University of Southampton

# Query Plans

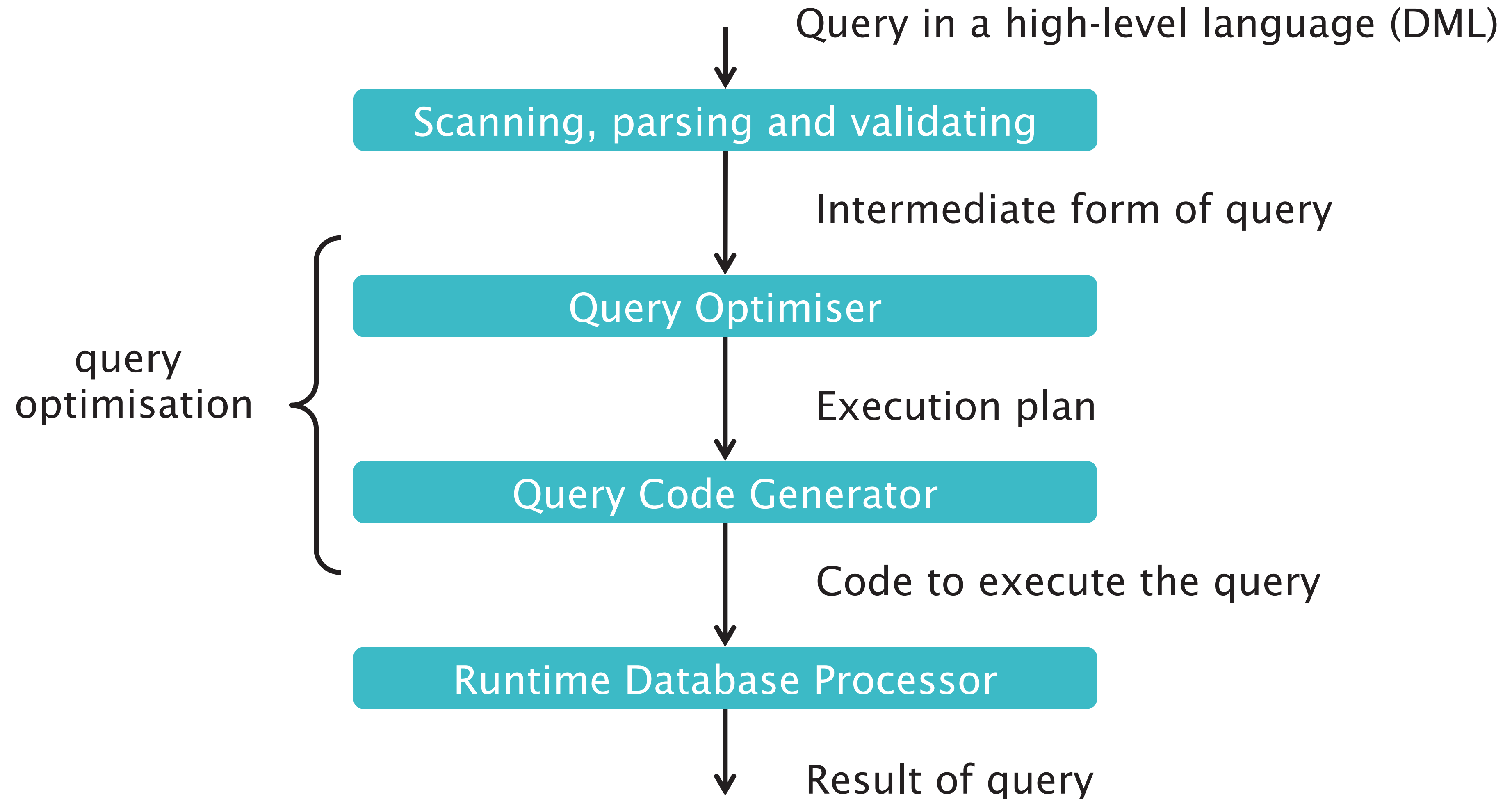
## Logical Query Plan

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

## Physical Query Plan

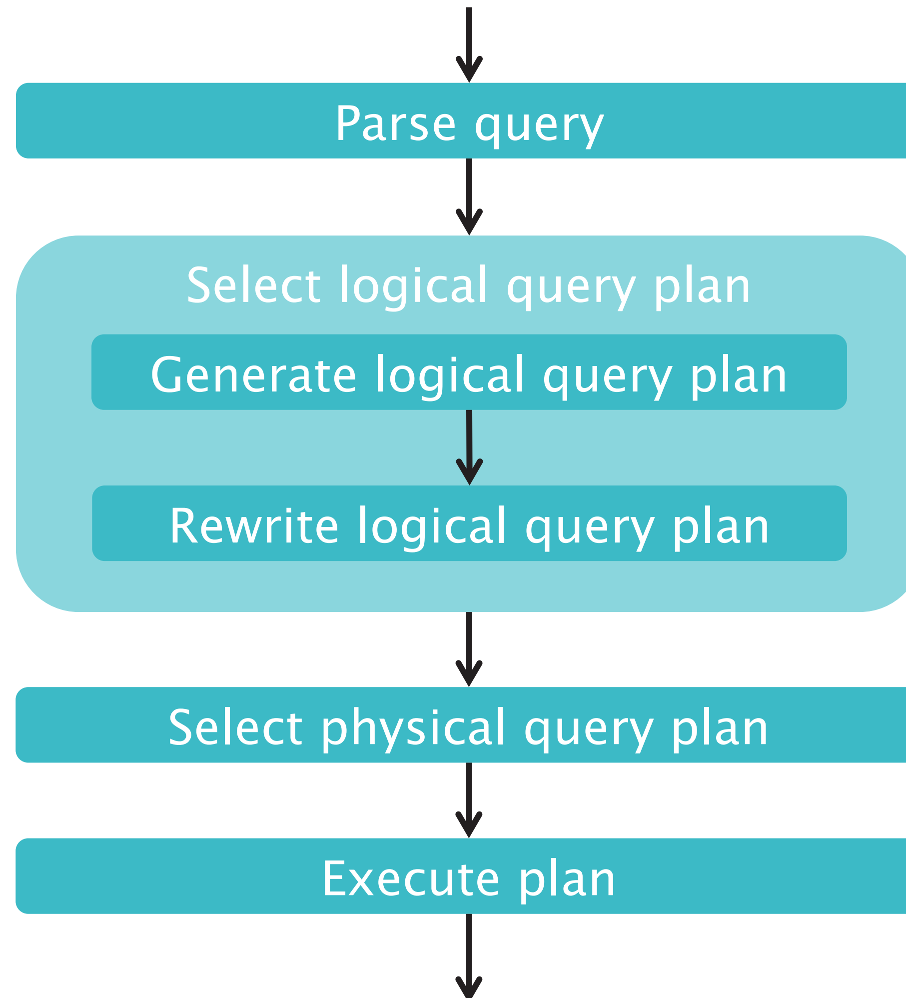
- algorithms selected for each operator in plan
- execution order specified for operators

# Query Processing



University of Southampton

# Query Processing



# Overview

## 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

University of Southampton

# 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

# Optimisation Example

| PROJECT | | |
|---|---|---|
| PNUMBER | PLOCATION | DNUM |

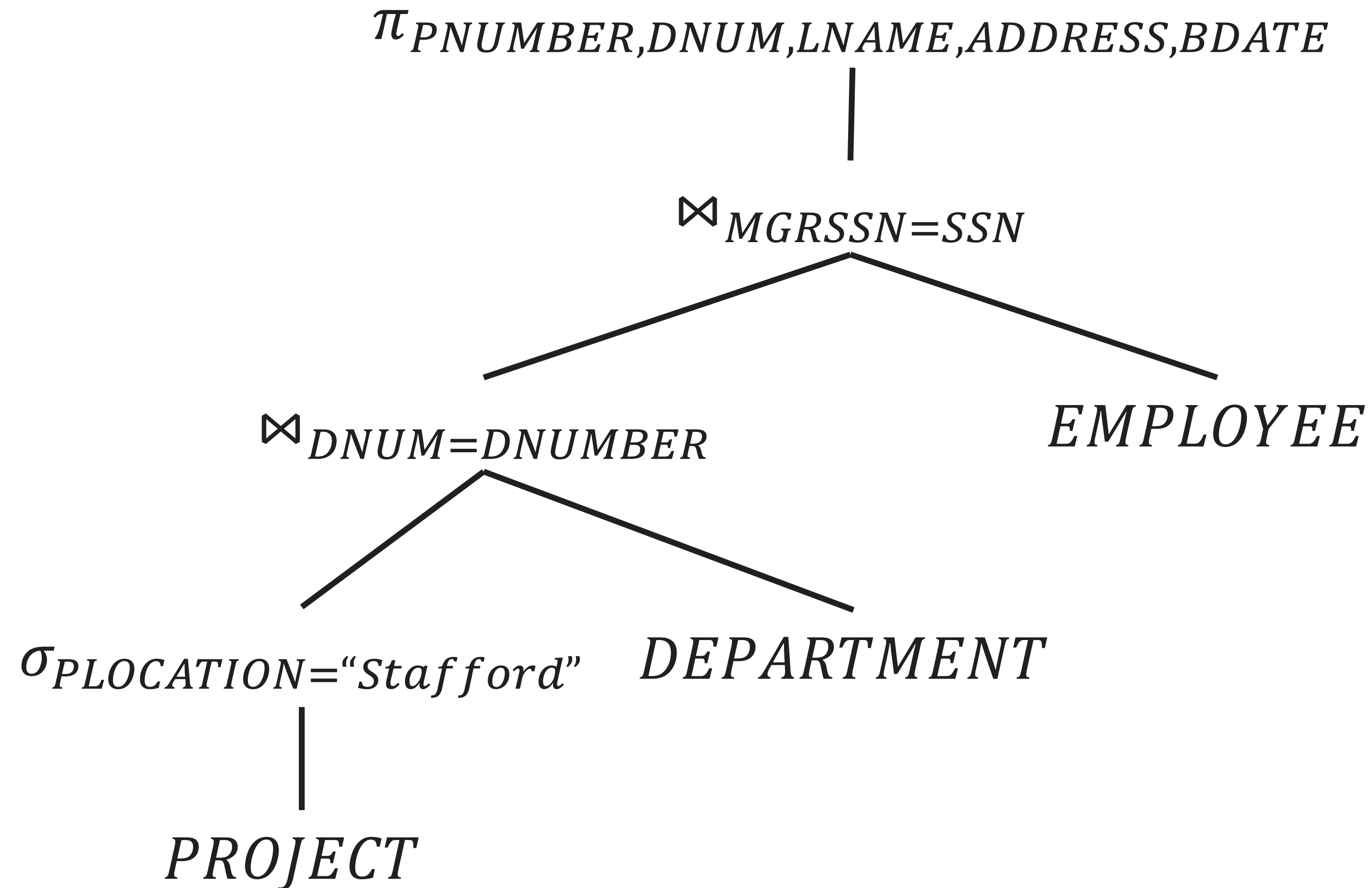
| DEPARTMENT | |
|---|---|
| DNUMBER | MGRSSN |

| EMPLOYEE | | | |
|---|---|---|---|
| SSN | LNAME | ADDRESS | DATE |

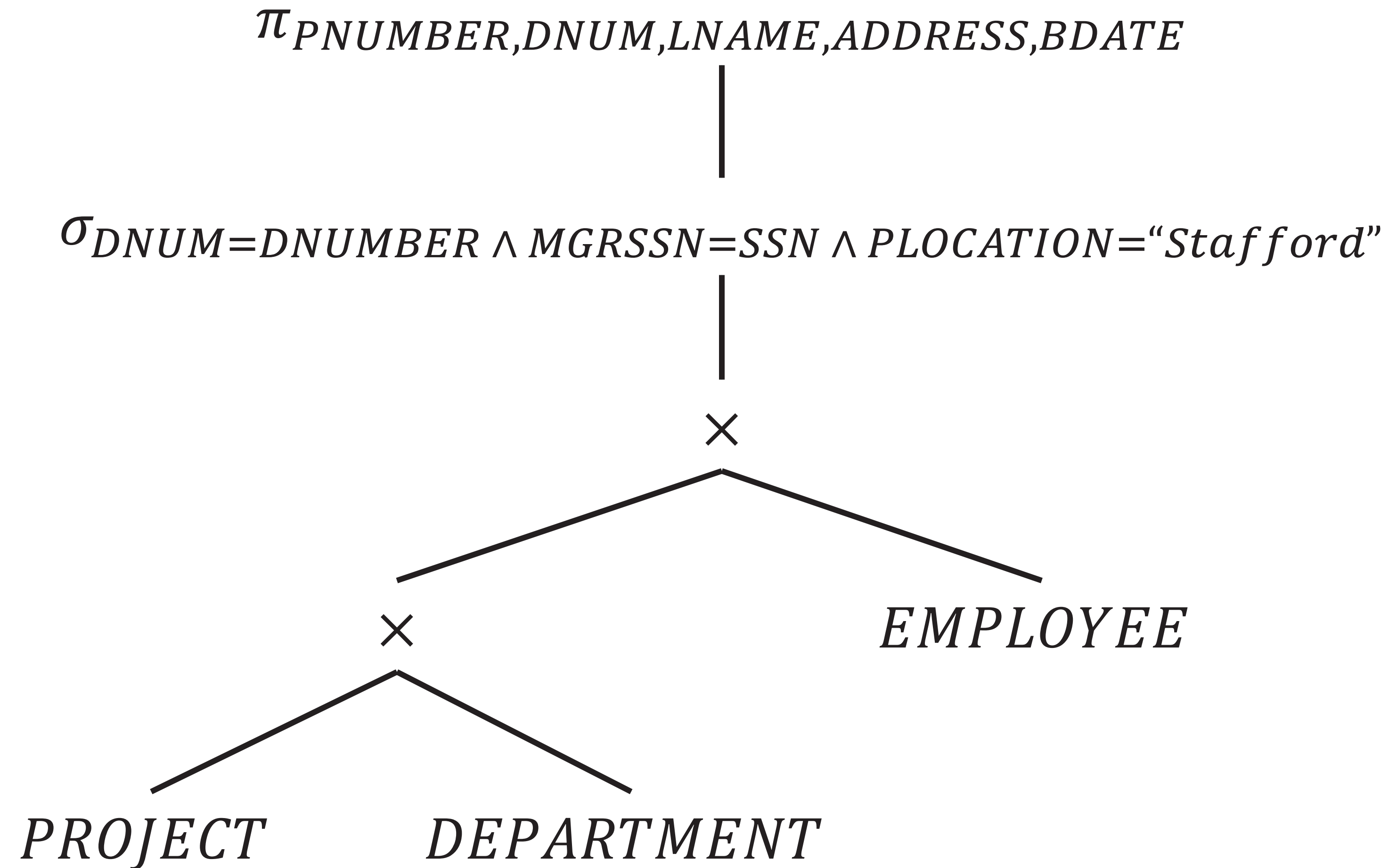
For every project located in Stafford, retrieve the project number, the controlling department number, and the department manager's last name, address and birth date

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

# Query Tree



# Canonical Form



# 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 $R$, $V(R, A) \leq T(R)$ for all attributes $A$ on $R$

University of Southampton

# Scan

Operation of reading all tuples of a relation

$T(scan(R)) = T(R)$

For all $A$ in $R$, $V(scan(R), A) = V(R, A)$

# 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 (value counts don’t change)

University of Southampton

# Selection

Two forms to consider:

- $\sigma_{attribute=value}(R)$
- $\sigma_{attribute1=attribute2}(R)$

# Selection case 1: attr=val

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

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

Assumption: all values of A appear with equal frequency

# Example: selection case 1: attr=val

$T(R) \ = \ 1000$
$V(R, A) \ = \ 10$
$V(R, B) \ = \ 1000$

$$T(\sigma_{A=c}(R)) = \frac{T(R)}{V(R, A)} = 100$$

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

$V(R, B) > T\left(\sigma_{A=c}(R)\right)$ so $\mathrm{V}(\sigma_{A=c}(R), B) = T\left(\sigma_{A=c}(R)\right) = 100$

# Selection case 2: attr=attr

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

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

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 $V(\sigma_{A=B}(R), X) \leq T(\sigma_{A=B}(R))$

# 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

What if we knew the range of values in $A$ and their distribution?

e.g., range [8, 57], uniformly distributed

University of Southampton

# Further Selection: Inequality

What about not equals?

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

Alternatively:

$$T(\sigma_{A \neq c}(R)) = T(R)\left(\frac{V(R,A) - 1}{V(R,A)}\right)$$

University of Southampton

# Further Selection: Conjunction

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

# Further Selection: Disjunction

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

This overestimates the number of tuples

Alternatively,

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

University of Southampton

# Join

Assume $R(X,Y) \bowtie S(Y,Z)$, i.e., natural join on attribute $Y$

Possible cases:

- $R$ and $S$ do not have any $Y$ value in common:
  - $T(R \bowtie S) = 0$
- $Y$ is the key of $S$ and a foreign key of $R$:
  - each tuple of $R$ joins with exactly one tuple of $S$
  - $T(R \bowtie S) = T(R)$
- All tuples of $R$ and $S$ have the same $Y$-value.
  - $T(R \bowtie S) = T(R)T(S)$

To capture the most common cases we need to make assumptions

University of Southampton

# Join

Assume $R(X, A) \bowtie_{A=B} S(B, Z)$

Assumptions:

- If $V(R, A) \leq V(S, B)$ then every $A$-value of $R$ will have a joining tuple $B$-value in $S$
- All values of $A$ and $B$ appear with equal frequency
- 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 $V(R \bowtie_{A=B} S, Y) \leq T(R \bowtie_{A=B} S)$

University of Southampton

# Join

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

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

# Further Join

If there are multiple pairs of join attributes:

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

University of Southampton

# 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

# Histograms

Let $R(A, B, C)$ be a relation with 10000 tuples.

Consider the following equal-width histogram on $A$:

| range | [1,10] | [11,20] | [21,30] | [31,40] | [41,50] |
| --- | --- | --- | --- | --- | --- |
| tuples | 50 | 2000 | 2000 | 3000 | 2950 |

$\mathrm{T}(\sigma_{A=10}(R)) = ?$

University of Southampton

# Histograms

Let $R(A, B, C)$ be a relation with 10000 tuples.

Consider the following equal-width histogram on $A$:

| range | [1,10] | [11,20] | [21,30] | [31,40] | [41,50] |
|---|---|---|---|---|---|
| tuples | 50 | 2000 | 2000 | 3000 | 2950 |

$$\mathrm{T}(\sigma_{A=10}(R)) = \frac{50}{10000} \times \frac{1}{10} \times T(R)$$

# Query Optimisation

University of Southampton

University of Southampton

# Cost estimation

The “cost” of an operator is the cardinality of its output relation

- Cost of processing or materialising its output

Overall query plan cost is the sum of cardinalities of intermediate relations

- Excluding the leaves (i.e. input relations)
- Excluding the end result (i.e. cost of the final operator)

Now that we have a way of judging whether one plan is better than another...

...all we need to do to find the optimal plan is to compare all the possible plans

So how many possible plans are there?

# How many query trees?

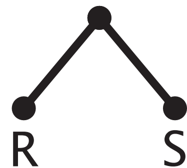
Considering plans with only $\times$ or $\bowtie$, and with $n$ relations:

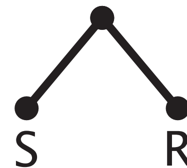
$n = 1$ R

# How many query trees?

Considering plans with only $\times$ or $\bowtie$, and with $n$ relations:

$n = 2$



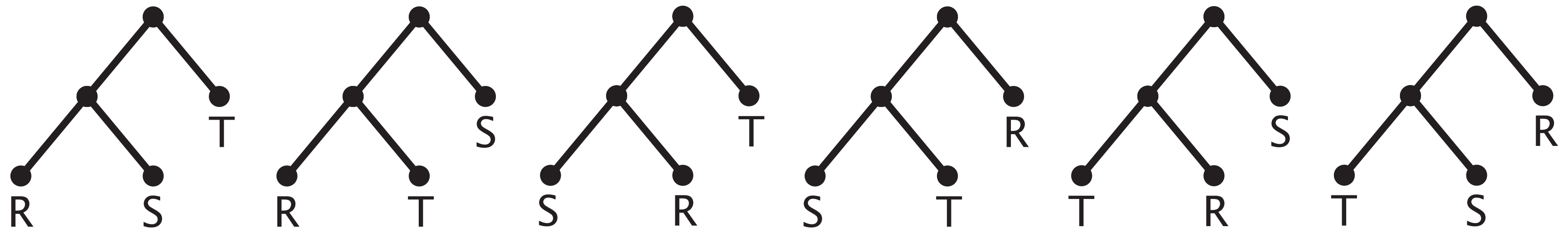

Note: while $\times$ and $\bowtie$ are symmetric, their corresponding physical operators aren't; the actual cost of $R \bowtie S$ may be different from that of $S \bowtie R$

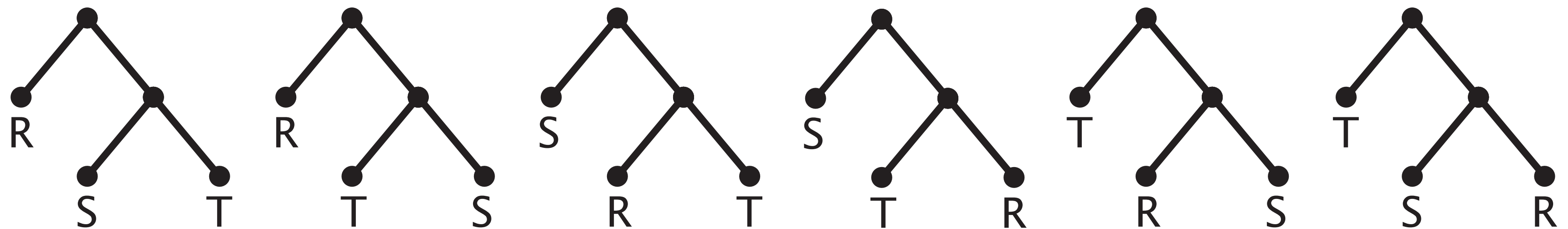
University of Southampton

# How many query trees?

Considering plans with only $\times$ or $\bowtie$, and with $n$ relations:

$n = 3$

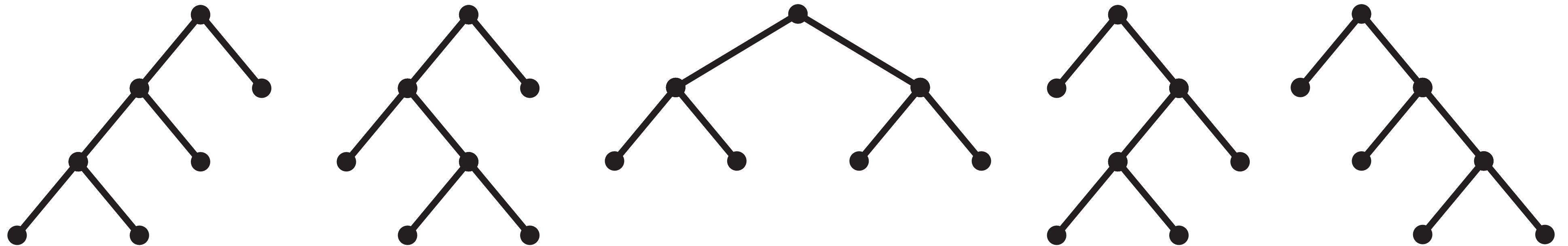


# How many query trees?

Considering plans with only $\times$ or $\bowtie$, and with $n$ relations:

$n = 4$



4! permutations of the relations for each of those shapes

120 different query trees

University of Southampton

# How many query trees?

When a query joins $n$ relations, how many possible query trees are there?

Number of possible binary trees with $n$ leaves is given by $C(n-1)$, where

$$C(n) = \frac{1}{n+1}\binom{2n}{2} = \frac{(2n)!}{(n+1)!\,n!}$$

(these are the Catalan numbers: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862...)

$n!$ permutations of the relations over the $n$ leaves of the binary trees

| #relns | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| #trees | 1 | 2 | 12 | 120 | 1680 | 30240 | 665280 | 17297280 |

University of Southampton

# Join ordering

Number of possible query plans for non-trivial queries precludes exhaustive search (and we haven’t even started considering choice of physical operators)

Join ordering is the main determiner of query cost

Need to guide search through space of possible join orderings

Prefer $\bowtie$ over $\times$ (cheaper – smaller output relation)

University of Southampton

# Query graphs

Consider conjunctive queries with simple predicates only
(i.e. predicates of the form $a_i = a_j$ or $a = const$)

Queries join base relations $R_1, R_2, \ldots, R_n$, possibly modified by selections

We can construct a **query graph** for queries of this type

- Undirected graph
- Vertices $R_1, R_2, \ldots, R_n$
- A predicate of the form $a_i = a_j$, where $a_i \in R_i$ and $a_j \in R_j$, gives an edge $\langle R_i, R_j \rangle$
- A predicate of the form $a = const$, where $a \in R_i$, gives an edge $\langle R_i, R_i \rangle$

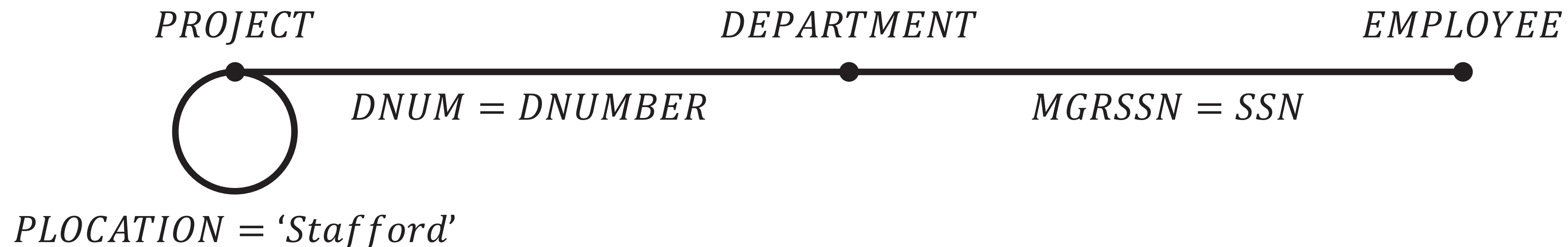
# Query graphs

| PROJECT | | |
|---|---|---|
| PNUMBER | PLOCATION | DNUM |

| DEPARTMENT | |
|---|---|
| DNUMBER | MGRSSN |

| EMPLOYEE | | | |
|---|---|---|---|
| SSN | LNAME | ADDRESS | DATE |

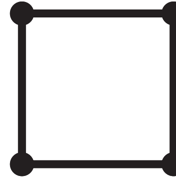
```
SELECT PNUMBER, DNUM, LNAME, ADDRESS, DATE
FROM   PROJECT, DEPARTMENT, EMPLOYEE
WHERE  DNUM=DNUMBER AND
       MGRSSN=SSN AND
       PLOCATION='Stafford'
```



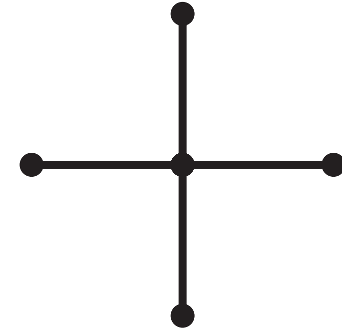
University of Southampton

# Query graph shapes

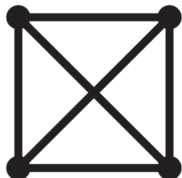
chain



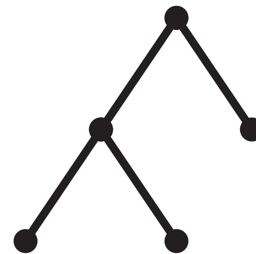
cycle



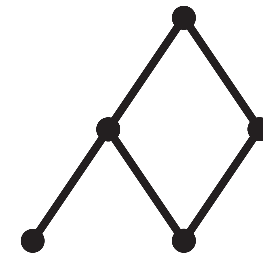
star



clique



tree



cyclic

University of Southampton

# Query graphs and join ordering

Some of these graph shapes are interesting because they can help us exclude join orderings that would lead to $\times$ (cross products)

- Chain (with cycle as a special case)
- Star
- Clique

General approach: repeatedly choose edges (i.e. joins) to add to the join tree that are adjacent to the edges already added

# Join trees

Choice of join tree shapes also constrains search space

Two main classes of join tree

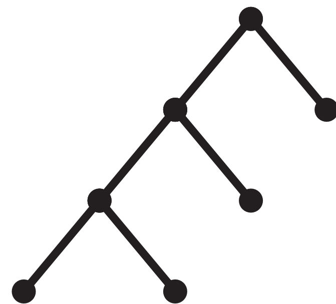
- Linear (left-deep, right-deep, zig-zag)
- Bushy

Choice depends on:

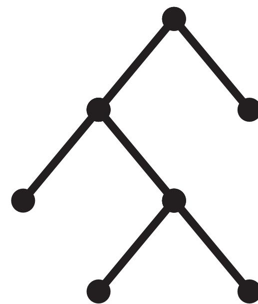
- Algorithms chosen (i.e. physical plan operators)
- Execution model

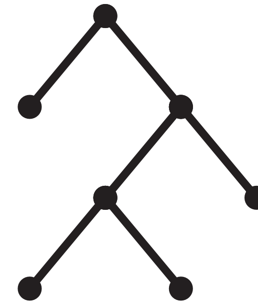
# Linear join trees

- Every join introduces at least one base relation
- Better for pipelining - avoids materialisation
- Possible left-deep trees: $n!$
- Possible right-deep trees: $n!$
- Possible zig-zag trees: $n!\,2^{n-2}$

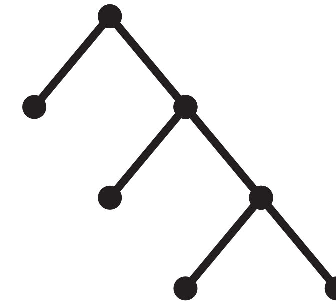


left-deep





zig-zag



right-deep

University of Southampton

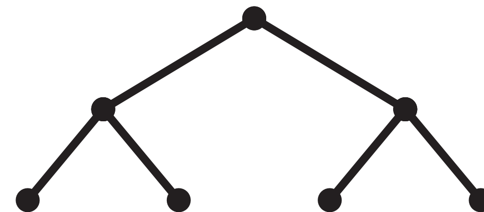
# Linear join trees

Combine with insights from query graph:

- Left/right-deep + chain: $2^{n-1}$ possible join trees without $\times$
- Left/right-deep + star: $2 * (n-1)!$ possible join trees without $\times$
- Left/right-deep + clique: $n!$ possible join trees without $\times$

# Bushy join trees

- Some joins may not join any base relations
- Need not be balanced
- Better for parallel processing
- Possible bushy trees: $n!\,C(n-1) = (2n)!/n!$



bushy

# Optimisation approaches

Wide variety of approaches – no single best approach

- Heuristic – transformation rules, keep transformed plan if cheaper
- Dynamic programming
- Randomised – avoid local minima by randomly jumping within big search spaces
- ...

(we could have a whole Part IV module on just this topic!)

University of Southampton

# Heuristic approach

1. Start with canonical form
2. Push $\sigma$ operators down the tree
3. Introduce joins (combine $\times$ and $\sigma$ to create $\bowtie$)
4. Determine join order
5. Push $\pi$ operators down the tree

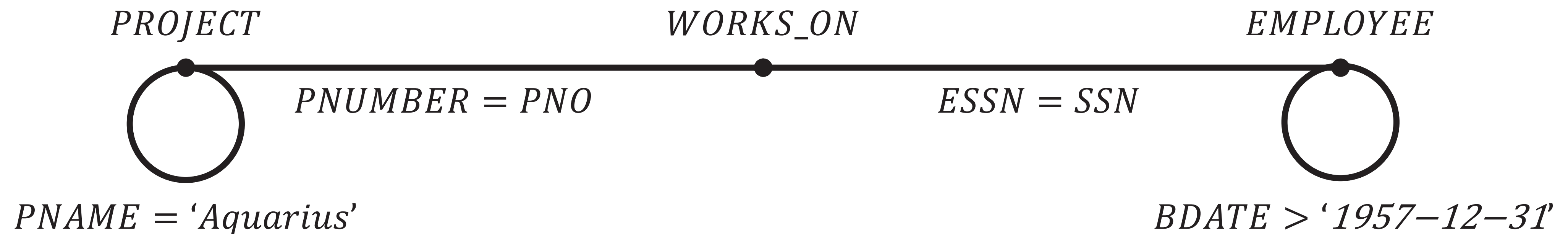
# Optimising query trees

| PROJECT | |
|---|---|
| PNUMBER | PNAME |

| WORKS_ON | |
|---|---|
| SSN | PNO |

| EMPLOYEE | | |
|---|---|---|
| ESSN | LNAME | BDATE |

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



University of Southampton

# Query trees and canonical form

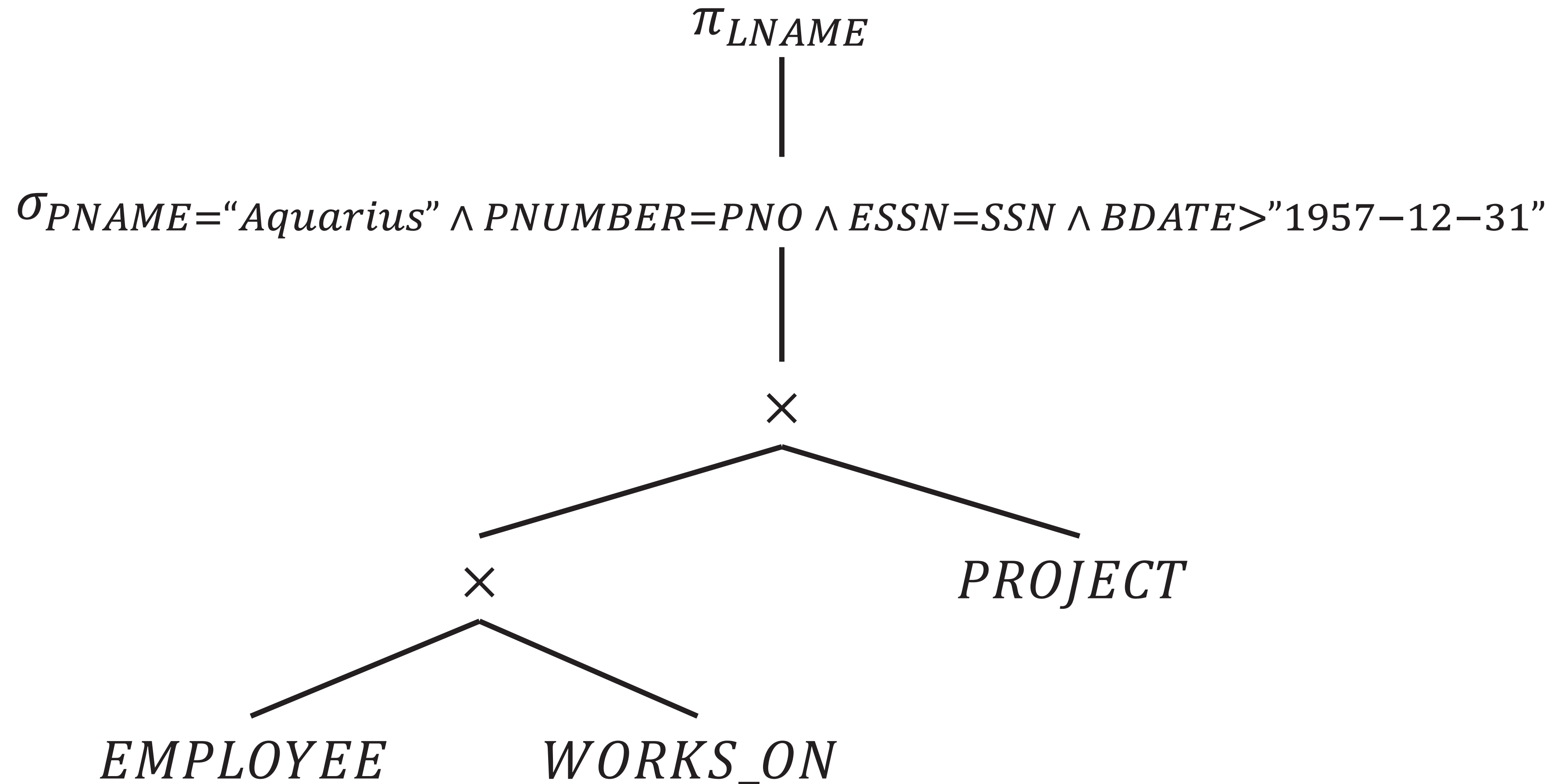
Useful to only consider left-deep trees

- 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 $\sigma$ down

Decompose selections containing conjunctive predicates:

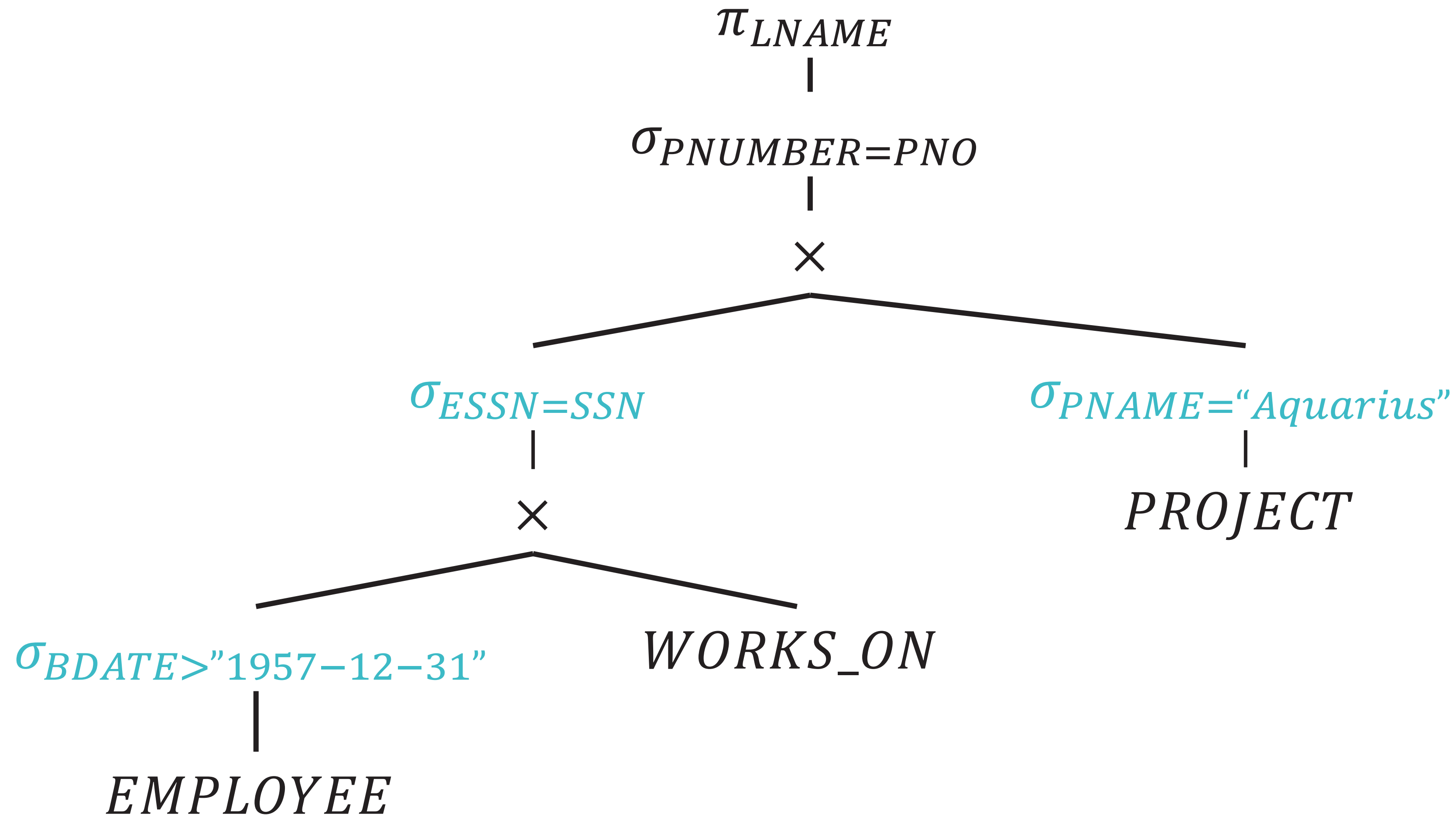
$$\sigma_{p1 \wedge p2 \wedge \cdots \wedge pn} R \equiv \sigma_{p1}(\sigma_{p2} \ldots (\sigma_{pn}(R)))$$

$$\sigma_{p1}(\sigma_{p2}(R) \equiv \sigma_{p2}(\sigma_{p1}(R)$$

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 $\sigma$ down



University of Southampton

# Reorder Joins

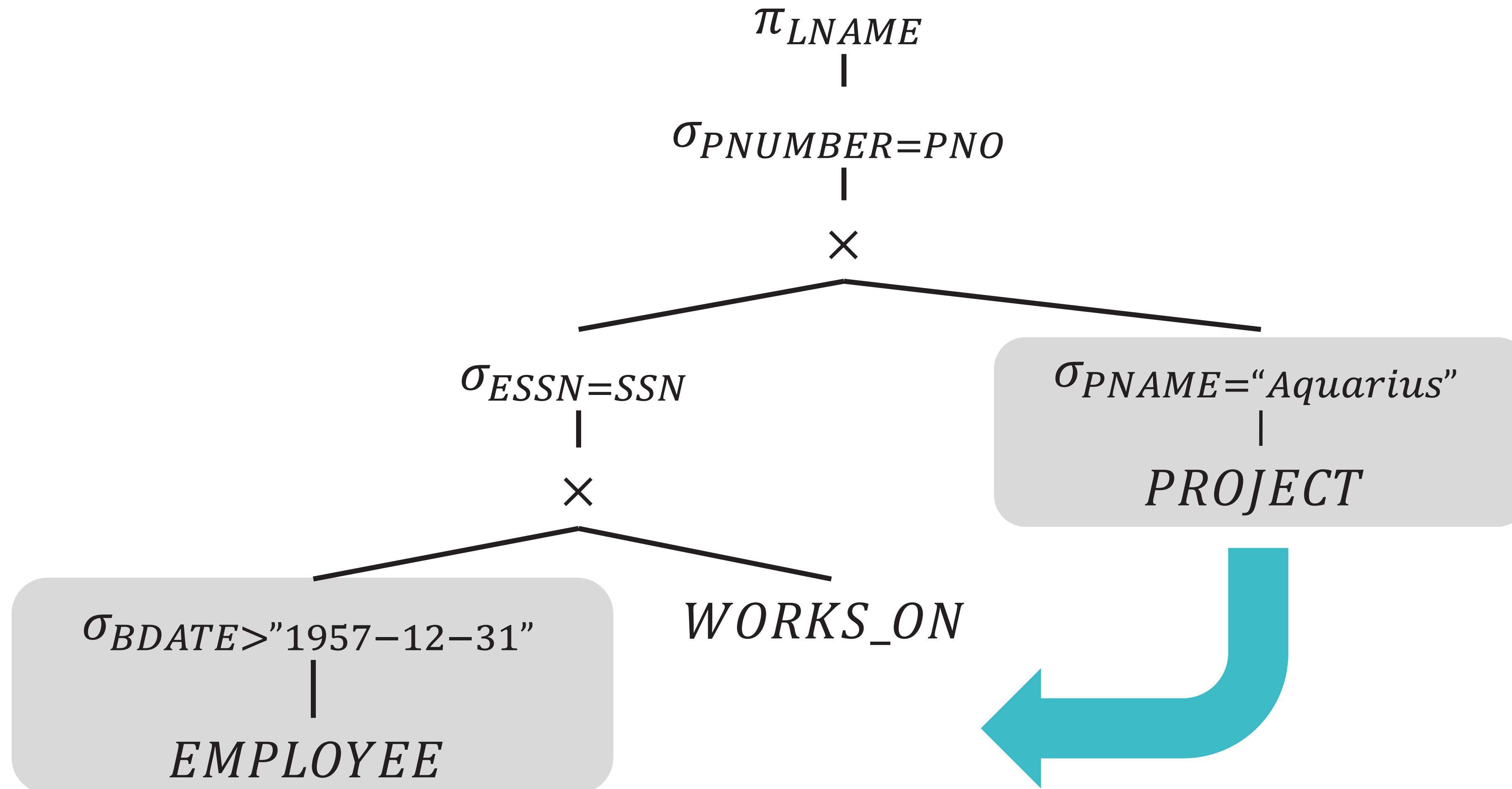
If a query joins $n$ relations and we restrict ourselves only to left-deep trees, there are $n!$ possible join orderings

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

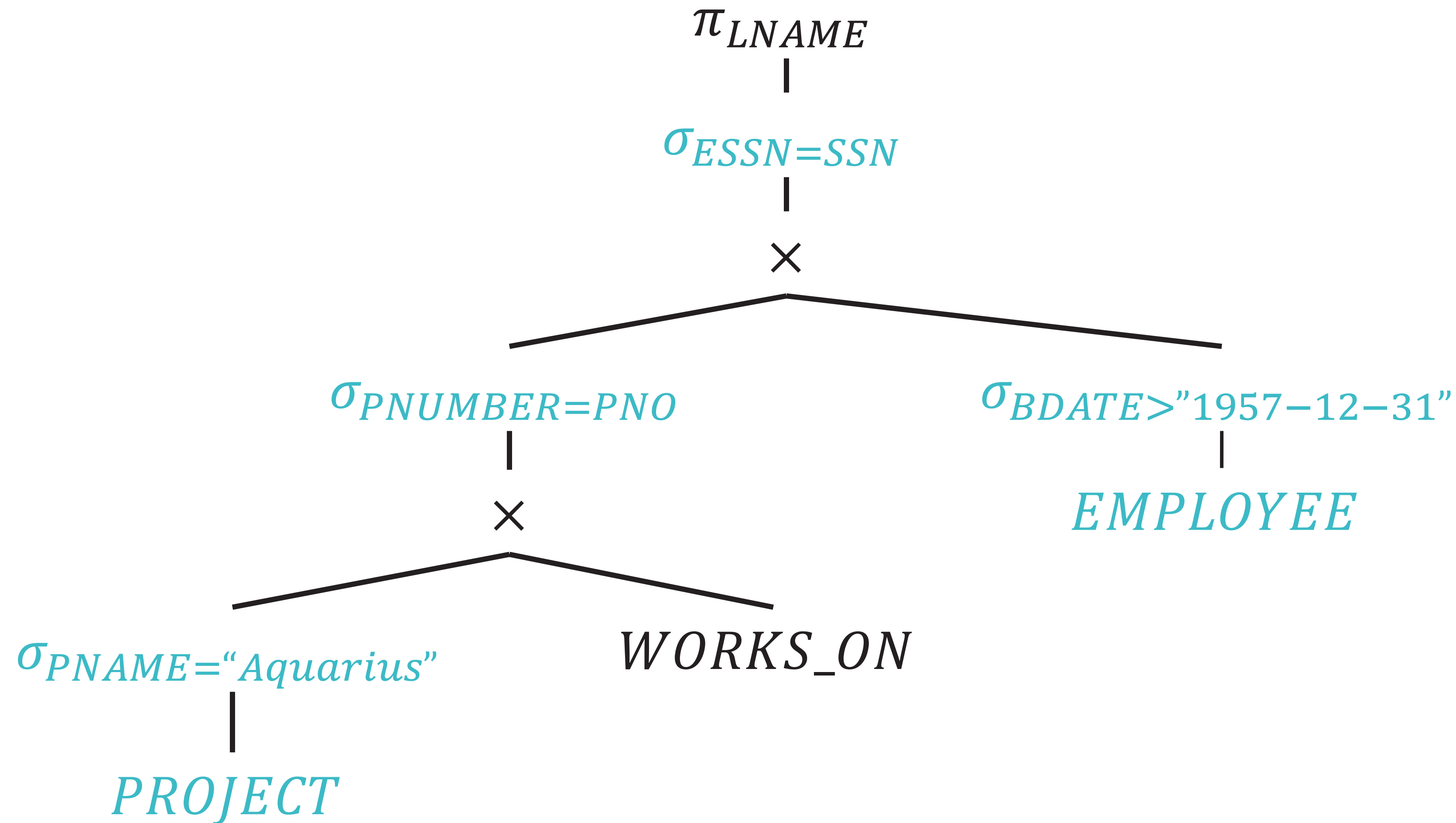
For simplicity of search, adopt a greedy approach:

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

# Reorder joins



# Reorder Joins



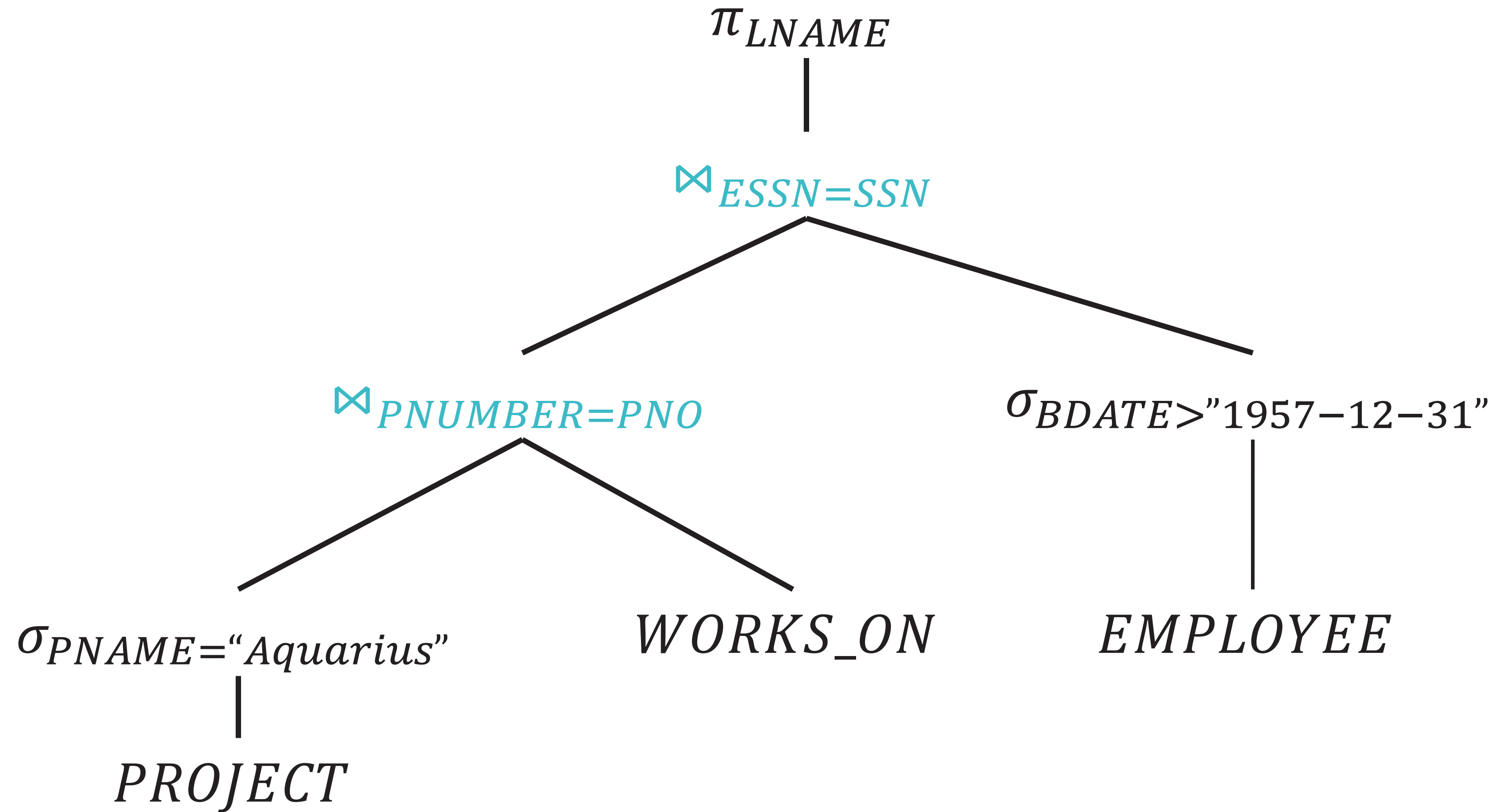
# Create joins

Combine $\times$ with adjacent $\sigma$ to form $\bowtie$

Uses the relational transformation $\sigma_p(R \times S) \equiv R \bowtie_p S$

Much cheaper than product followed by selection

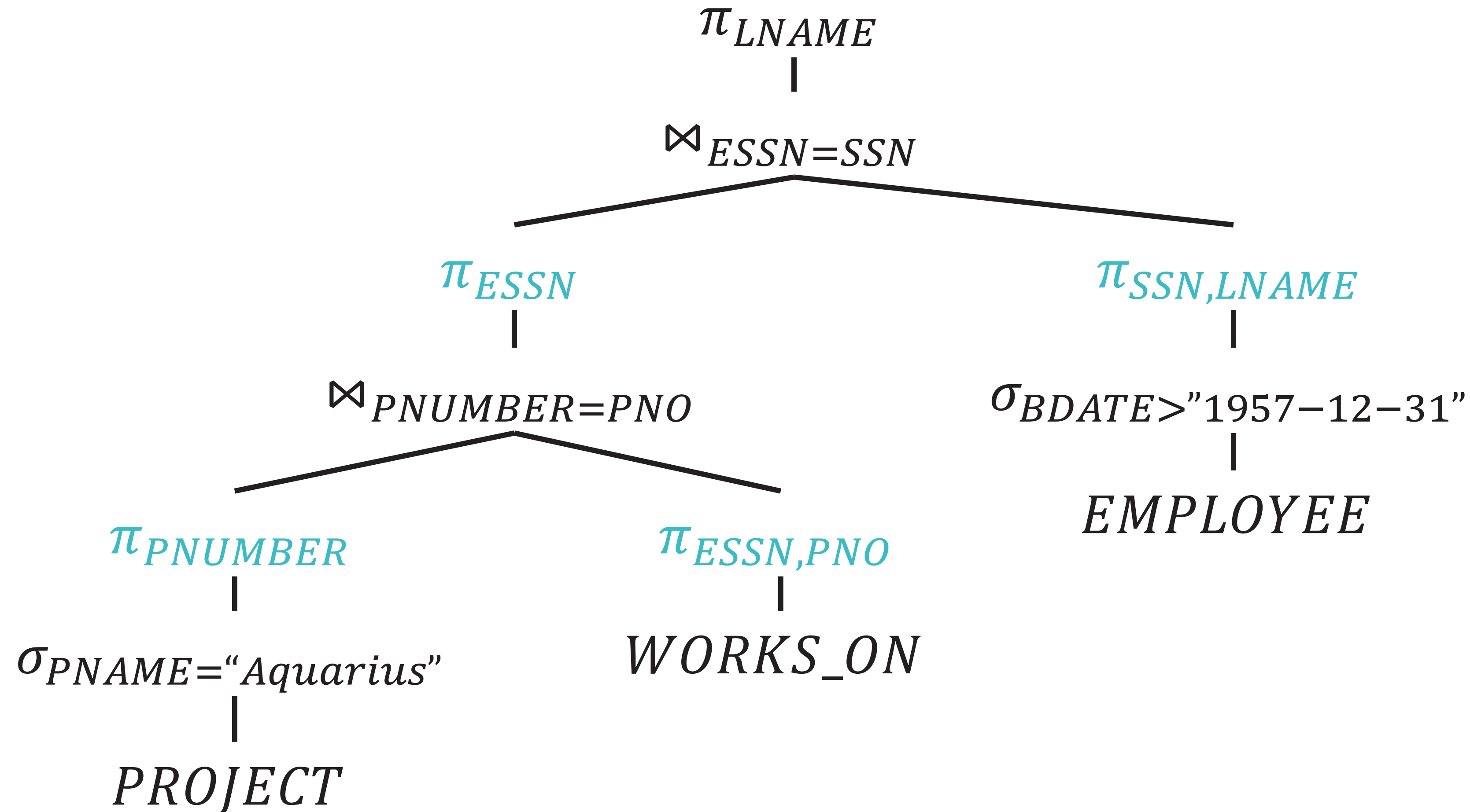
# Create joins



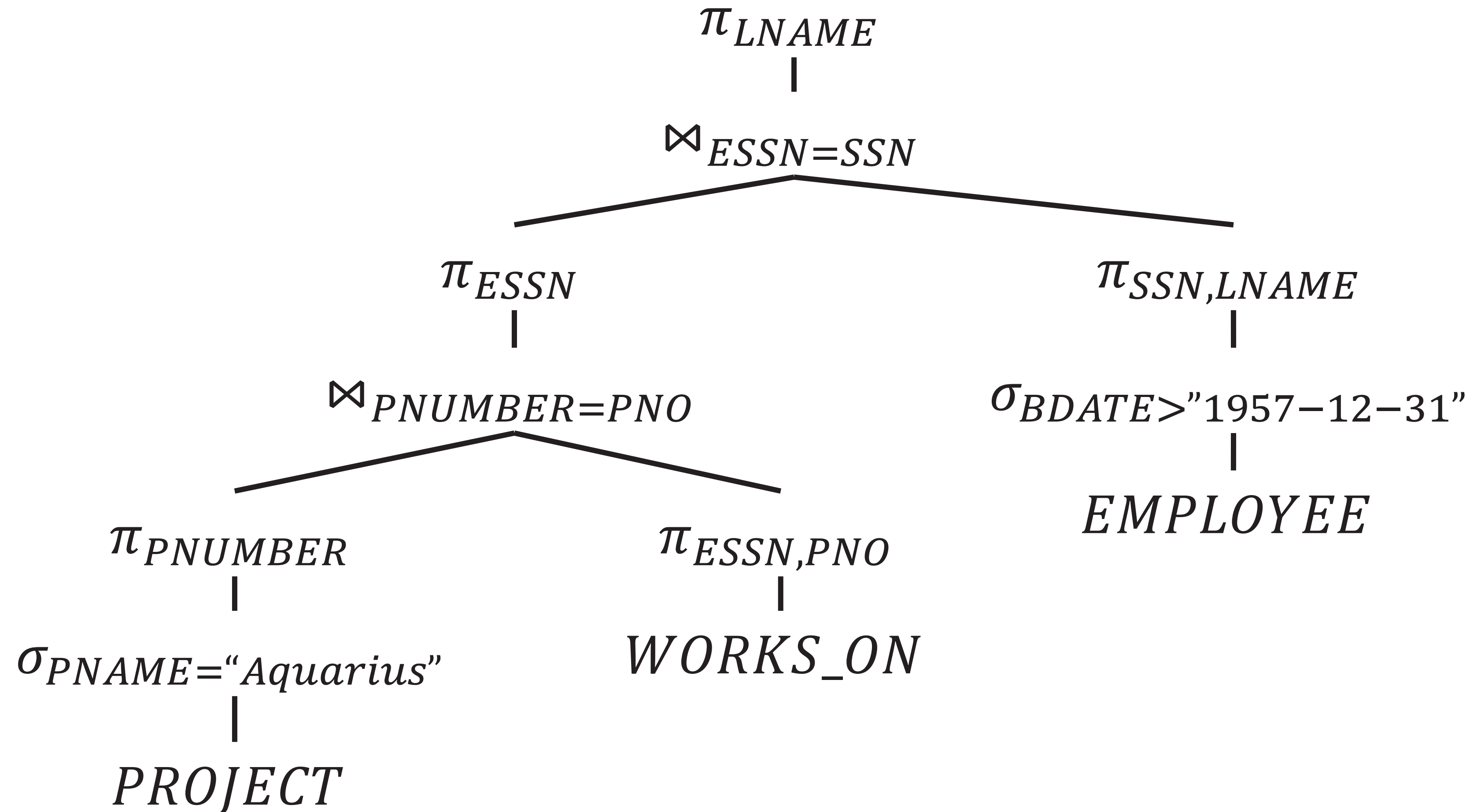
# Move $\pi$ down

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

# Move $\pi$ down



# Optimised logical query plan



# Execution Models

University of Southampton

# Execution models

A physical query plan is a tree of physical plan operators

The execution model defines:

- the interface that connects operators to each other
- how data is propagated between operators
- how operators are scheduled

Operator interface

- Relation(s) in, relation out
- Producer-consumer relationship

# Pipelining

**Pipelining** – read input, process, propagate output to next operator

Benefits of pipelining:

- No buffering (because no materialisation)
- Faster execution (no materialisation, so no disk I/Os)
- More in-memory operations

Not all operators can be pipelined

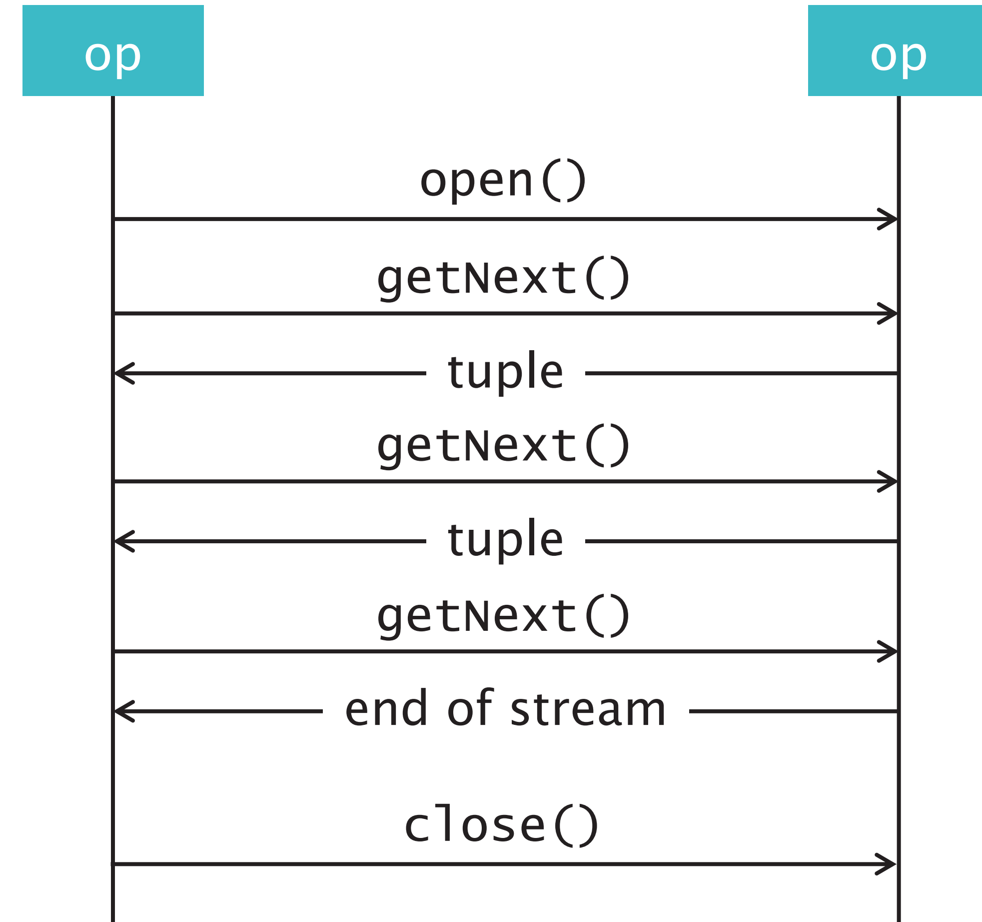
- Some require intermediate relations to be materialised
- Some operators will always block

# Iterators

Standard interface on each operator:

- `open()`
- `getNext()`
- `close()`

- Query engine calls the interface on the root operator
- Calls to interface are propagated down the tree




University of Southampton

University of Southampton

# Synchrony versus asynchrony

As presented, the operator interface is synchronous

- Operators don't generate tuples until `getNext()` is called
- In reality, different operators will have different evaluation times
- Some operators may block – causing the whole plan to block

Move to an asynchronous implementation by introducing buffering:

- Within the operator calling the interface (the push model)
- Within the operator being called (the pull model)
- In the connections between operators (the stream model)

Asynchronous implementations minimise time during which blocking occurs

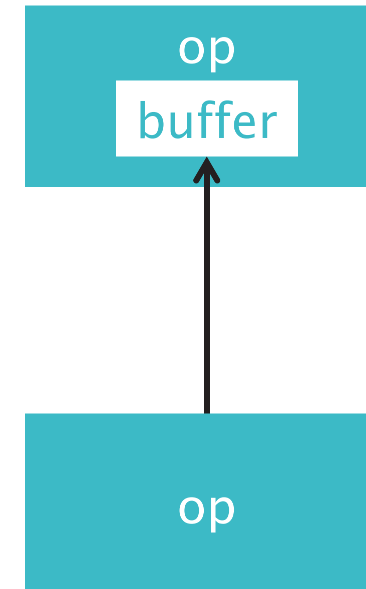
University of Southampton

# The push model

Propagate from the leaves upwards

- Producer propagates tuples as soon as they're available
- Producer propagates tuples regardless of whether consumer has yet called `getNext()`
- Consumer buffers incoming tuples until it calls `getNext()`

Minimises idle time, good for pipelining



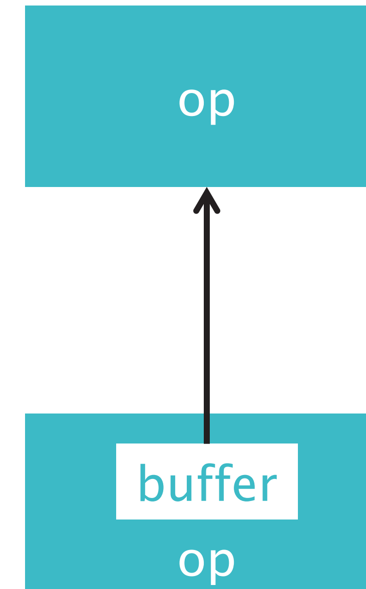
University of Southampton

# The pull model

Propagation driven from the root

- Producer buffers tuples until getNext() is called

On-demand, close to pure implementation



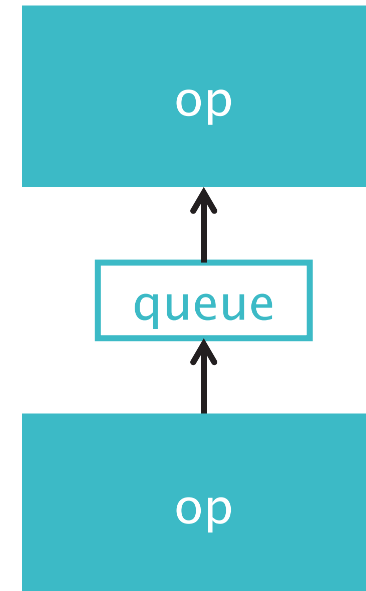
# The stream model

Connections as first-class objects:

- FIFO queues of tuples

- Producer propagates tuples to the queue as soon as they're available
- Consumer call to `getNext()` does not block if there's something in the queue

Asynchronous operators (but synchronous streams), good for parallelisation



# Physical Plan Operators

University of Southampton

# 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
- ...

University of Southampton

# 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

University of Southampton

# Cost Parameters

| | |
|---|---|
| $M$ | Main memory available for buffers |
| $S(R)$ | Size of a tuple of relation $R$ (in blocks) |
| $B(R)$ | 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

| R1 R2 S1 S2 | R3 R4 S3 S4 |
|---|---|

University of Southampton

# Clustered Relation

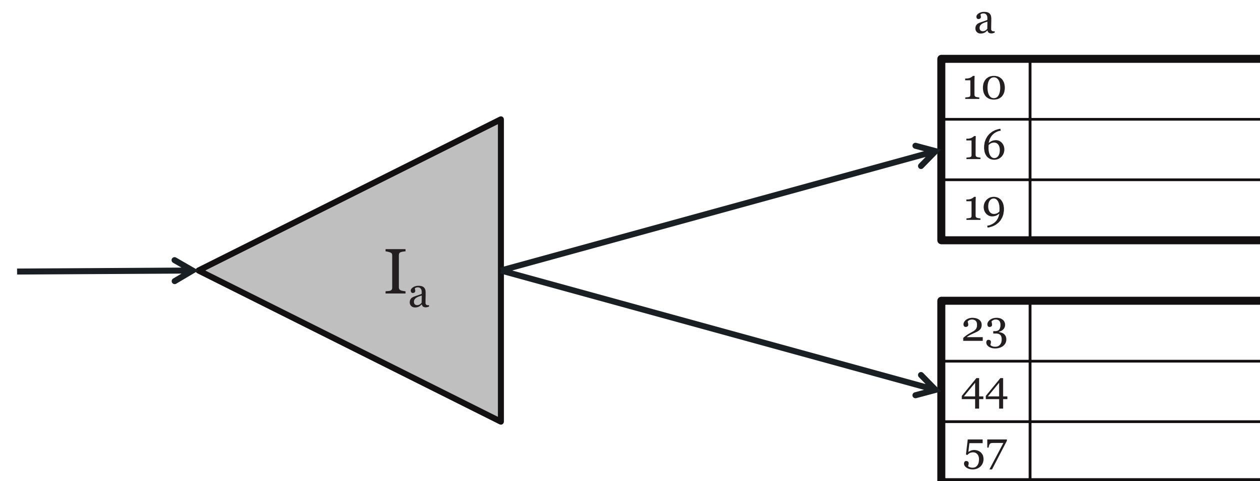
Tuples from relation are stored together in blocks, but not necessarily sorted

| R1 R2 R3 R4 |
|---|

| R5 R6 R7 R8 |
|---|

# Clustering Index

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



# 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

# 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
- $B(R) \gg B(I_R)$ so treat as only $B(R)$
- $T(R)$ disk accesses if not clustered

# One-Pass Algorithms

University of Southampton

# One-Pass Algorithms

Read data from disk only once

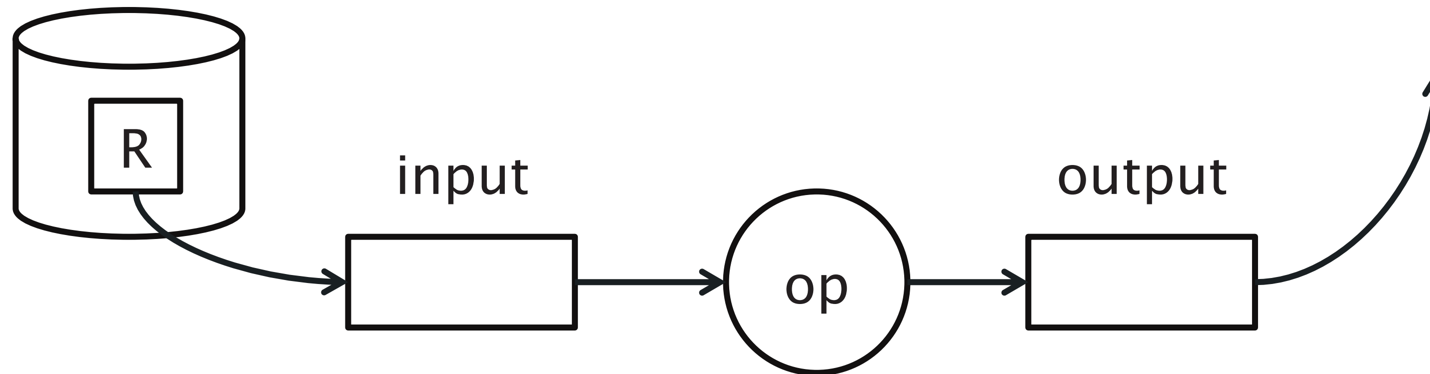
Typically require that at least one argument fits in main memory

Three broad categories:

- Unary, tuple at a time (i.e. select, project) – non-blocking
- Unary, full-relation (i.e. duplicate elimination, grouping) – may be blocking
- Binary, full-relation – typically blocking

# 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



University of Southampton

# Unary, tuple at a time: Cost

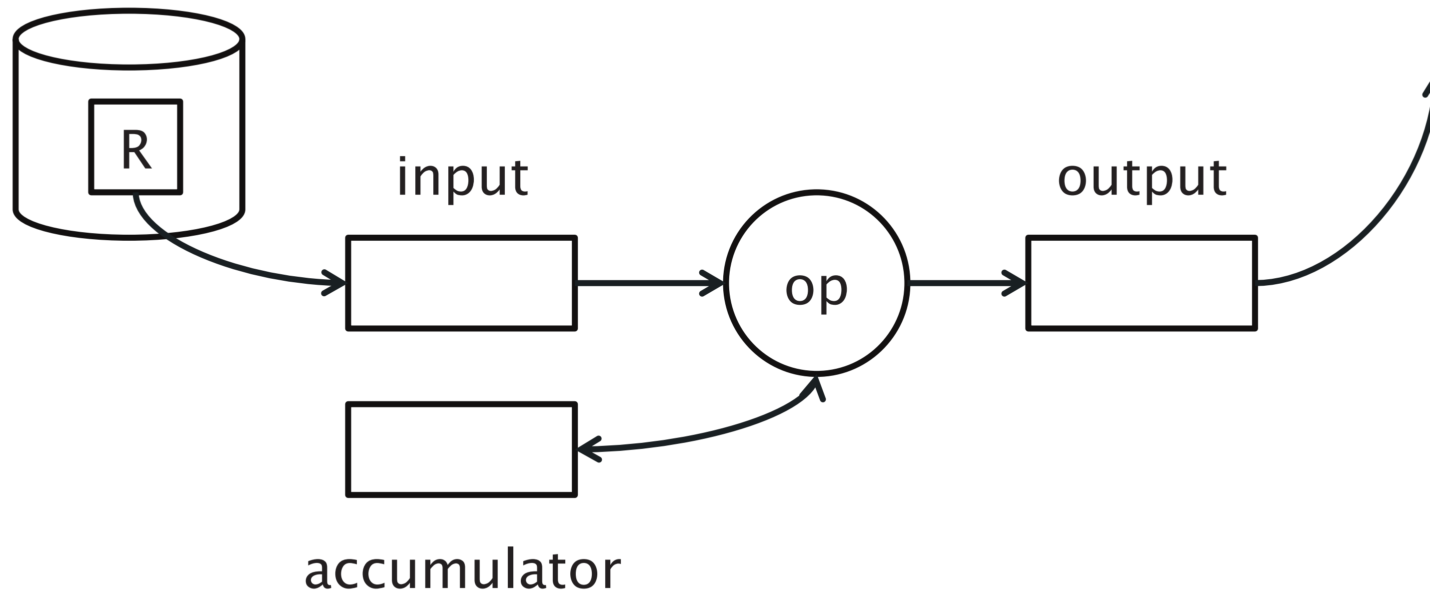
In general, $B(R)$ or $T(R)$ disk accesses depending on clustering

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

Requires $M \geq 1$

# Unary, full-relation

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



University of Southampton

# 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
```

University of Southampton

# Unary, full-relation: Duplicate elimination

Requires $M \geq B(\delta(R)) + 1$ blocks of main memory

- 1 block for input buffer
- $B(\delta(R))$ blocks for accumulator (records each tuple seen so far)

- Accumulator implemented as in-memory data structure (tree, hash)
- If fewer than $B(\delta(R))$ blocks of memory 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

University of Southampton

# 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: $\min(B(R), B(S)) \leq M - 1$

# 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 join

Also known as iteration join

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

foreach $r \in R$
        foreach $s \in S$
                if $r.a = s.a$ then output $\langle r, s \rangle$

University of Southampton

# Factors that affect cost

- Are the tuples of the relation stored physically together? (clustered)
- Are the relations sorted by the join attribute?
- Do indexes exist?

University of Southampton

# Example

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

$T(R1) = 10{,}000$
$T(R2) = 5{,}000$
$S(R1) = S(R2) = 0.1$
$M = 101$

# Attempt #1: Tuple-based nested loop join

Relations not contiguous - one disk access per tuple

R1 is outer relation

R2 is inner relation

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

$$
\begin{aligned}
\text{Cost} &= T(R1) * \left(1 + T(R2)\right) \\
&= 10{,}000 \ * (1 + 5{,}000) \\
&= 50{,}010{,}000
\end{aligned}
$$

# 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

Tuples of R1 stored in a 100-block chunk = 100 * 1/S(R1)
= 1,000 tuples

Number of 100-block chunks to store R1 = T(R1) / 1,000
= 10

Cost to read one 100-block chunk of R1 = 1,000 disk accesses

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

Total cost = 10 * 6,000 = 60,000 disk accesses

University of Southampton

# Can we do better?

What happens if we reverse the join order?

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

University of Southampton

# Attempt #3: Join reordering

| | |
|---|---|
| Tuples of R2 stored in a 100-block chunk | = 100 * 1/S(R2) |
| | = 1,000 tuples |
| Number of 100-block chunks to store R1 | = T(R2) / 1,000 |
| | = 5 |
| Cost to read one 100-block chunk of R2 | = 1,000 disk accesses |
| Cost to process each chunk = 1000 + T(R1) | = 11,000 |

Total cost = 5 * 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) = T(R1)/S(R1) = 1,000
B(R2) = T(R2)/S(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

# Can we do better?

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

# Attempt #5: Merge join

Read each block of R1 and R2 once only

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

# Two-Pass Algorithms

University of Southampton

# Can we do better?

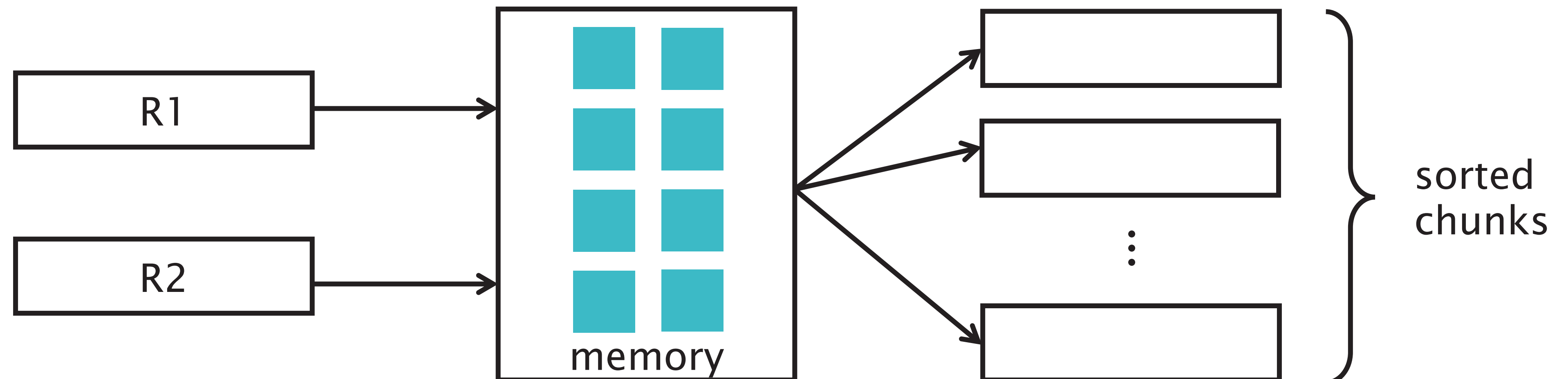
What if R1 and R2 *aren't* sorted by a?

...need to sort R1 and R2 first

# Merge Sort

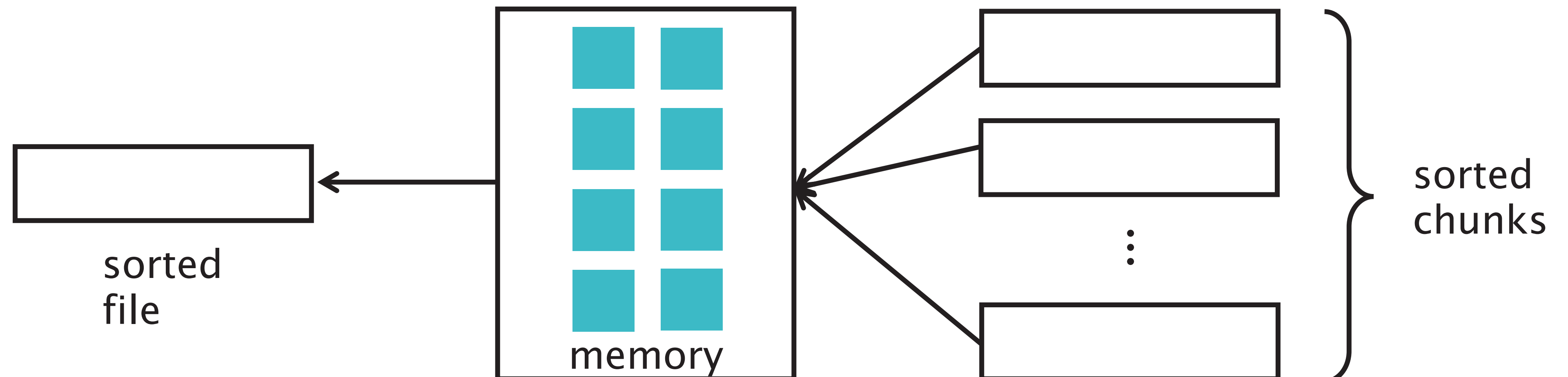
(i) For each 100 block chunk of R:

- Read chunk
- Sort in memory
- Write to disk



# Merge Sort

(ii) Read all chunks + merge + write out



# 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

# 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 with sort does not necessarily pay off

# Attempt #6, part 2

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

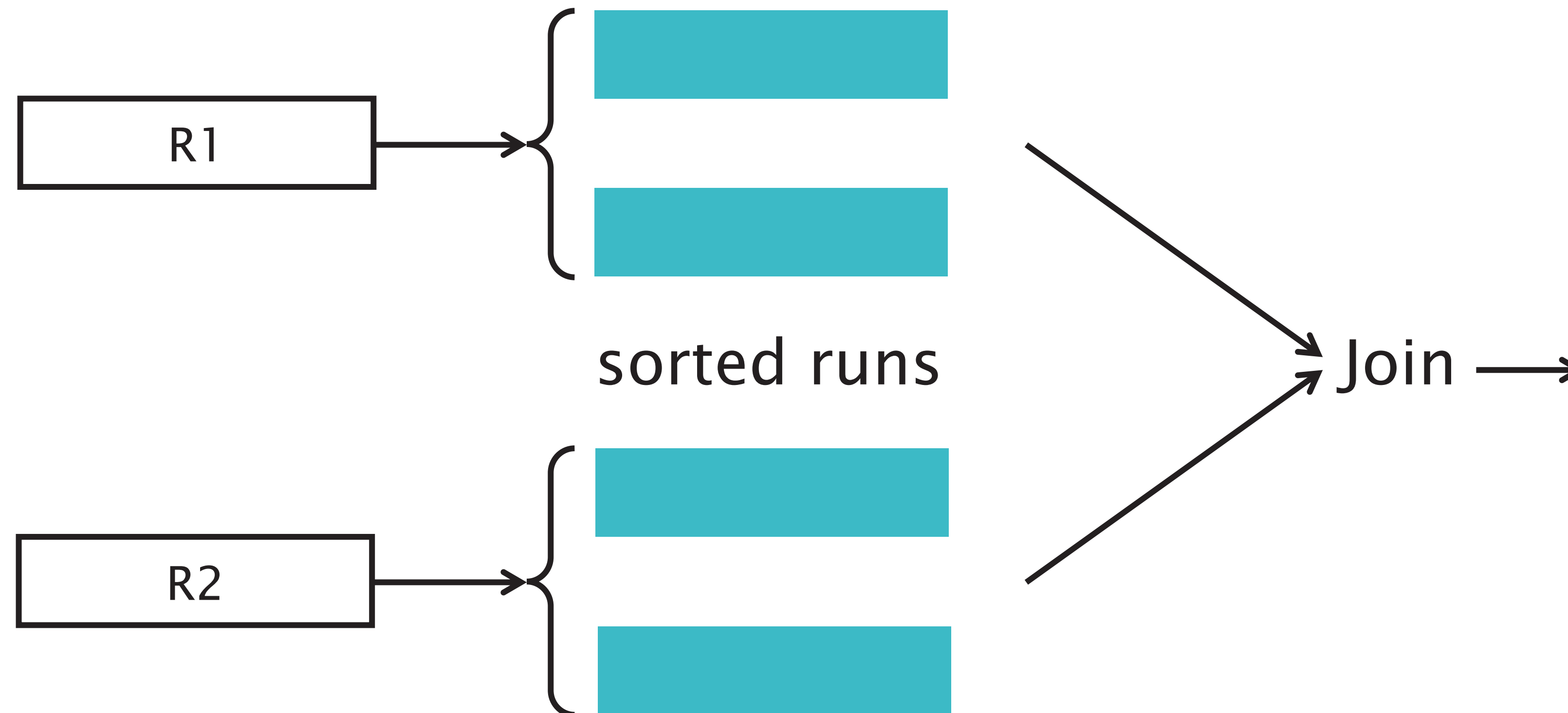
Nested loop cost = (5,000/100) * (100 + 10,000)
= 505,000 disk accesses

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

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

# 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

# Index-based Algorithms

University of Southampton

# Can we do better?

What if we have an index on the join attribute?

- Assume R2.a index exists and fits in memory
- Assume R1 contiguous, unordered

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

# How many matching tuples?

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

expected number of matching tuples = 1

# 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 domain(R2, C)=1,000,000, T(R2) = 10,000
with alternate assumption

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

University of Southampton

# Attempt #8: Index join

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

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

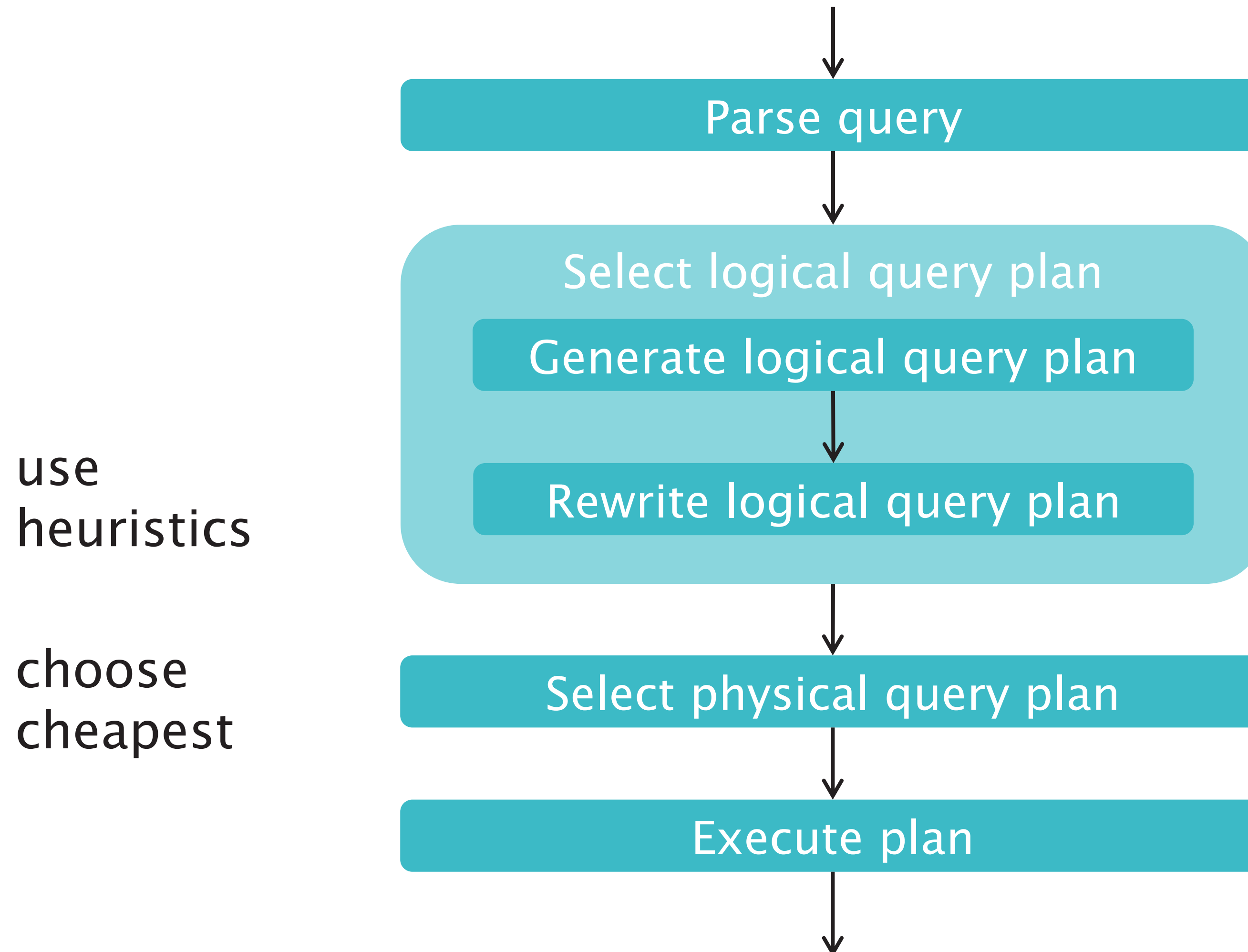
(c) Cost = 500+5000 * 1/100 * 1 = 550 disk accesses

University of Southampton

# Summary

University of Southampton

# Query Processing



University of Southampton

# Next Lecture: Transactions and Concurrency