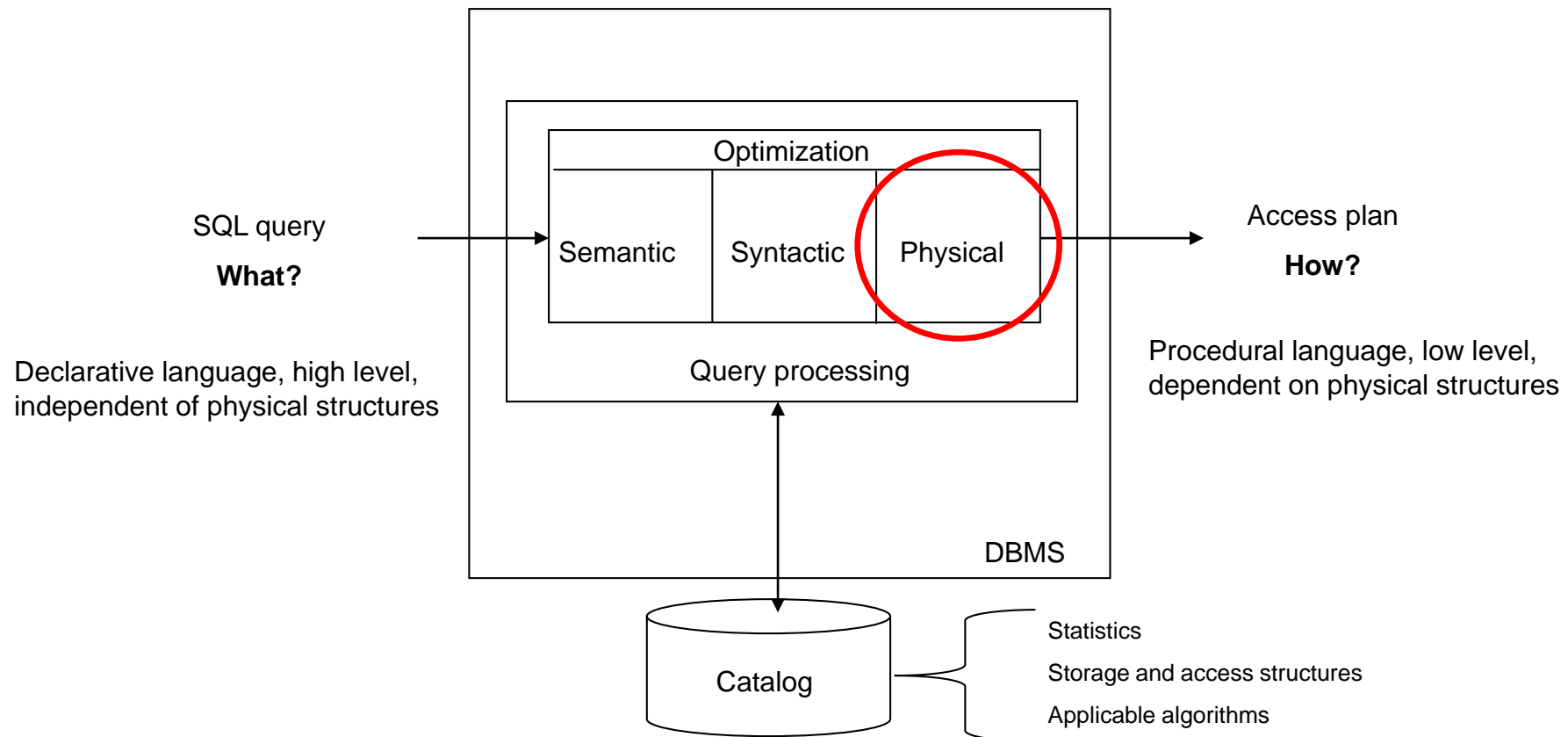


# Physical Optimization

Database Foundations

# Recall: Main Modules



# Definition

Consists of **generating** the **execution plan** of a query (from an optimized syntactic tree) by considering:

- The physical structures already created and their **access path** required to execute the query
  - An access path is an algorithmic scheme used to retrieve data. For example:
    - Access all tuples,
    - Access one specific tuple (e.g., based on its value on a specific attribute),
    - Access several tuples (e.g., based on a range of values on a specific attribute)
- Alternative execution algorithms for certain operations

The first step, is to transform the input syntactic tree (high level) into a process tree (physical operations implemented in the database)

# Physical Optimization: Overview

The physical optimization follows these steps:

[INPUT: A Process Tree]

- Generate execution alternatives
- Estimate the cost of each execution alternative
- Choose the cheapest one

[OUTPUT: Access Plan]

# Generating the Process Tree

# Optimized Syntactic Tree(s)

This tree is the output generated by the syntactic optimizer

- For example:

```
SELECT DISTINCT w.strength
```

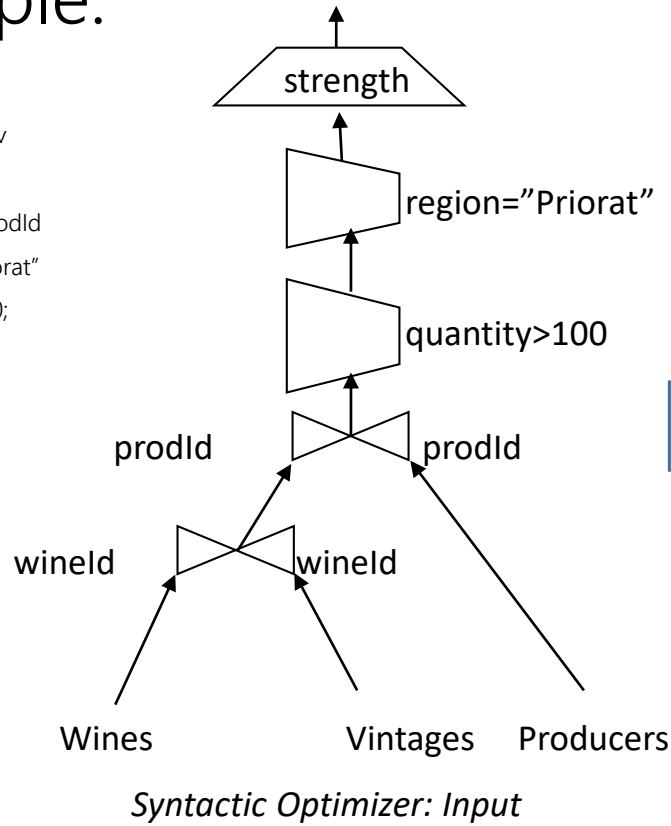
```
FROM wines w, producers p, vintages v
```

```
WHERE v.wineld=w.wineld
```

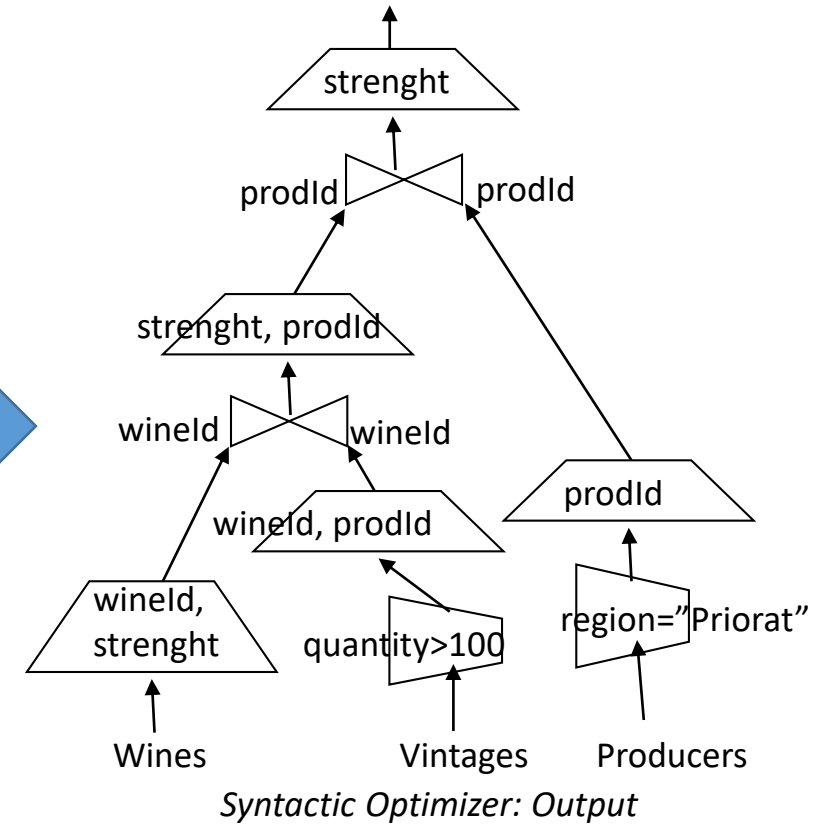
```
AND p.prold=v.prold
```

```
AND p.region="Priorat"
```

```
AND v.quantity>100;
```



Syntactic optimizer



# Process Tree

Consists of **transforming** the input syntactic tree into an equivalent **process tree** that models the **execution strategy**

- Therefore, it goes one level of abstraction down and deals with physical operations and specifies what operator consumes the output of what others
- The process tree looks as follows:
  - Nodes
    - Leaves: Tables (or Indexes)
    - Internal: Intermediate (temporal) tables generated by a physical operation
      - These are also known as **intermediate results**
    - Root: Result
  - Edges
    - Denote direct usage

# Internal Nodes of the Process Tree

- Operations related to relational algebra
  - Physical selection: Selection [+ projection]
  - Physical join: Join [+ projection]
  - Set operations:
    - Union [+ projection]
    - Difference [+ projection]

*Note: in general, the projection operator is meaningless as physical operator. Thus, it is always executed during the result writing phase of another operation (except for when it is the only operation in the tree)*

- Other operations:
  - Duplicate removal
  - Sorting
  - Grouping and aggregation



# Example of Process Tree Generation

SELECT DISTINCT w.strength

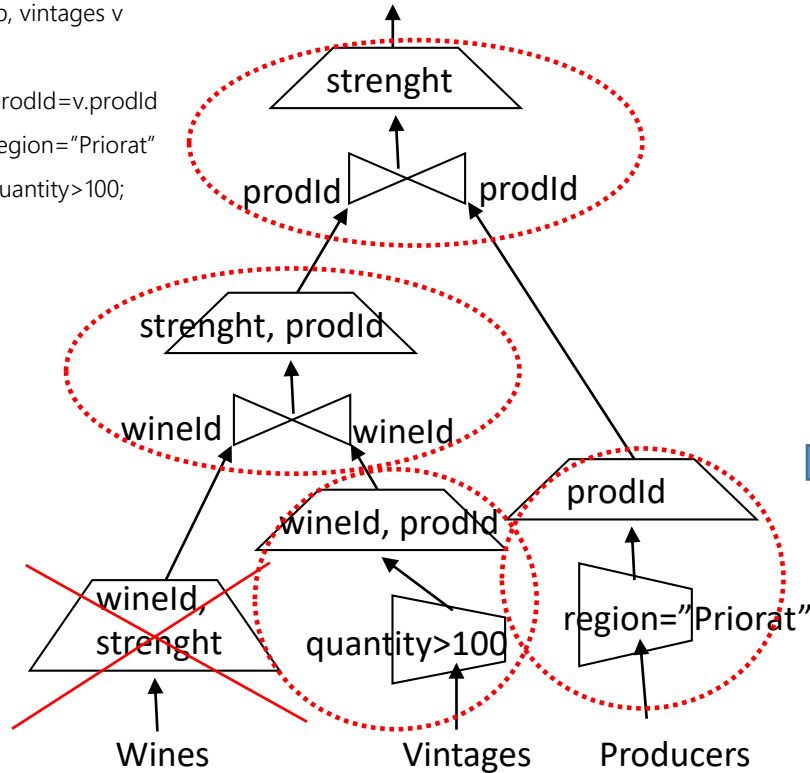
FROM wines w, producers p, vintages v

WHERE v.wineld=w.wineld

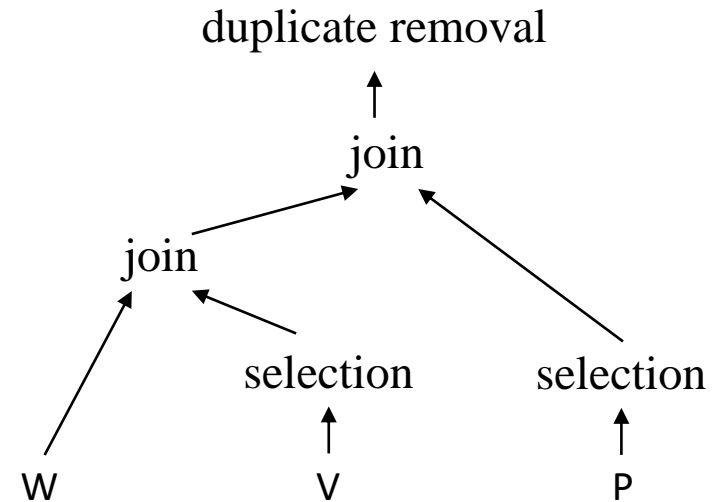
AND p.prold=v.prold

AND p.region="Priorat"

AND v.quantity>100;



*Input: Optimized Syntactic Tree*



*Note: W, V and P are used to denote the heap files implementing the wines, vintages and producers tables (respectively)*

# Phases of the Physical Optimization

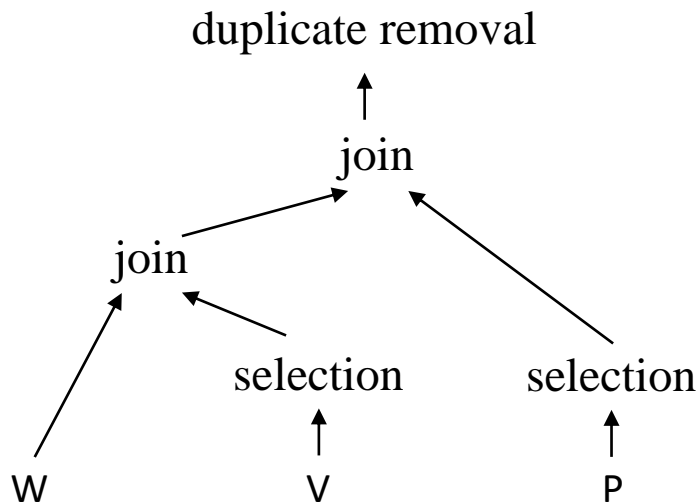
# Physical Optimizer: Phases

1. For each process tree node, **generate alternatives** in the search space
  - a. Alternative algorithms to execute an operation
    - a. Typical for joins (row nested loops, block nested loops, hash join, etc.)
  - b. Available physical structures (access path) for each operation
  - c. ~~Materialization or not of intermediate results~~
    - We will assume that they are always materialized
2. **Evaluate** those **alternatives** by estimating
  - a. The cardinality and size of intermediate results (required for b.)
  - b. Cost of each algorithm and access path based on the available data structures
3. Choose the *best* alternative

# Overall View

```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineld=w.wineld
```

```
AND p.prold=v.prold
AND p.region="Priorat"
AND v.quantity>100;
```



Consider now the following structures:

- Vintages has a B+(quantity)
- Producers has a hash(region)
- Producers has a B+(region)
- Wines has an index cluster on (wineld)

Consider the DBMS has 3 join algorithms:

- Row-nested loops (RNL)
- Block nested loops (BNL)
- Sort-match (SM)

## [PHASE 1] Generate alternatives:

- Selection (V) can be executed as:
  - Table scan (V) + selection
  - B+(quantity) + selection

*Let us call the intermediate result generated V'*
- Selection (P) can be executed as:
  - Table scan (P) + selection
  - B+(region) + selection
  - Hash(region) + selection

*Let us call the intermediate result generated P'*
- Join (W, V') can be executed as:
  - Table Scan (V') + Index Cluster(wineld) + join
    - The join can be executed either as RNL, BNL or SM (3 alternatives\*)
  - Table Scan (V') + Table Scan (W) + join
    - The join can be executed either as RNL, BNL or SM (3 alternatives\*)

*Let us call the intermediate result generated WV'*
- Join (WV', P') can be executed as:
  - Table Scan (WV') + Table Scan (P') + join
    - The join can be executed either as RNL, BNL or SM (3 alternatives\*)

\* SM is only an option if the inputs are sorted. RNL requires indexes on the join attribute

# Overall View

## [PHASE 2] Cost Estimation

- For each alternative, now, we must estimate their cost
- The cost of each operation is the sum of
  - Cost of solving it (i.e., execute the algorithm computing the operation)
  - Cost of writing its result as an intermediate result (to a temporary table in disk)
- Cost factors:
  - ~~CPU~~
  - ~~Memory access time~~
  - Disk access time

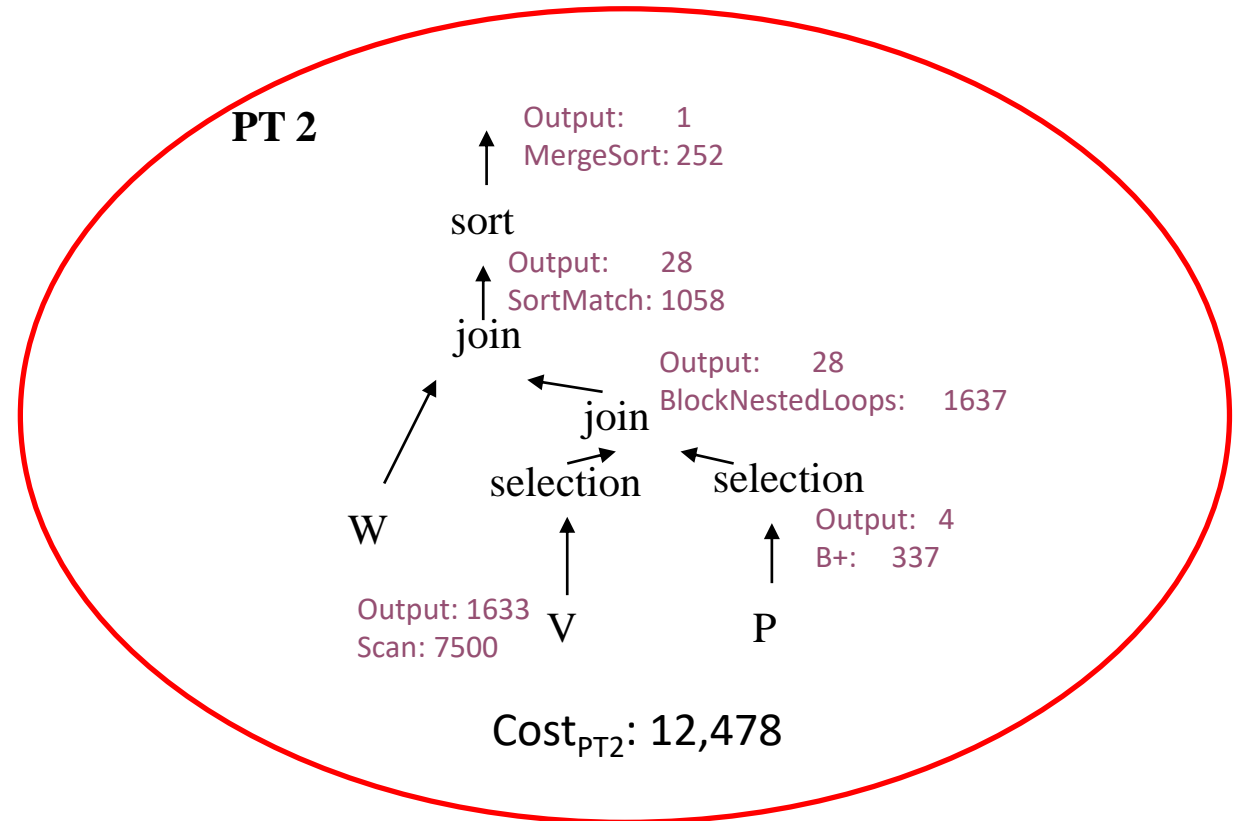
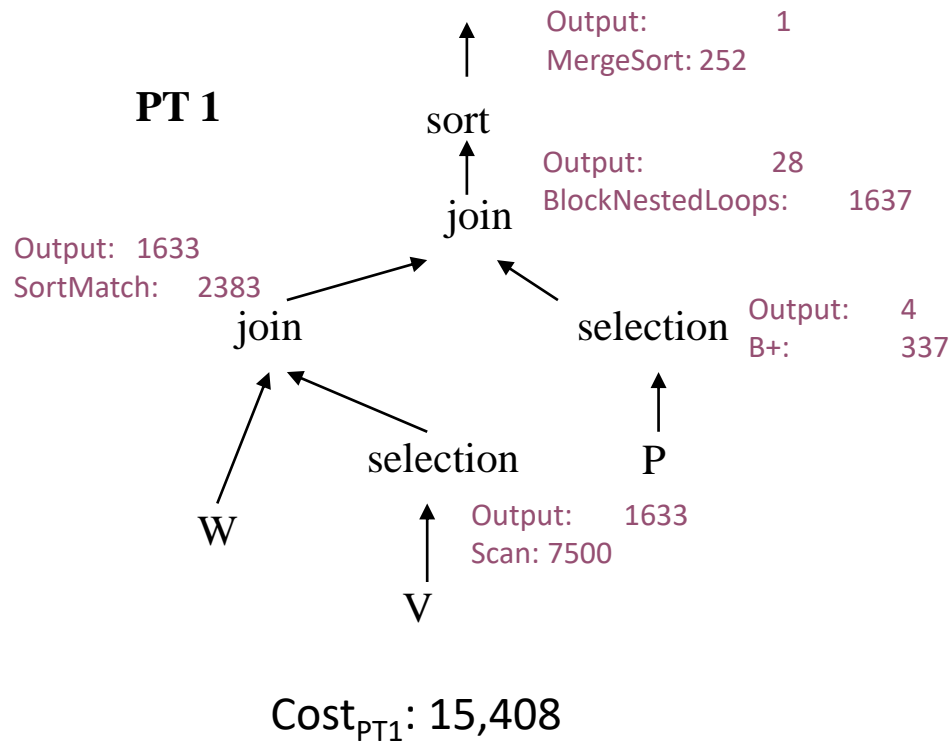
*Note: different systems may consider different variables when estimating the disk access time cost*

- The cost of a process tree is the sum of the costs of all physical operations participating in it

# Overall View

## [PHASE 3] Choose the Cheapest Alternative

```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineld=w.wineld
      AND p.prodId=v.prodId
      AND p.region="Priorat"
      AND v.quantity>100;
```



# Phase 1: Generating Alternatives

Details on how to generate alternatives from a process tree

# Alternatives in the Structures

- Whether indexed or not, the table can be:
  - Ordered
  - Unordered
- Two main indexing structures:
  - B-tree
  - Hash
- Indexes always keep entries composed by pairs value-information, where information can be:
  - The whole record
  - Physical address of the record
  - List of physical addresses of records
  - Bitmap
- Out of all possible combinations, in the next slides we will only consider:
  - No index (**table scan**)
  - Unordered and B-tree with addresses (**B+**)
  - Unordered and Hash with addresses (**Hash**)
  - Ordered and B-tree with addresses (**Index Cluster**)



# Alternatives in the Execution Algorithms

- Joins can typically be executed with alternative execution algorithms
  - However, it may happen that certain DBMS provide additional alternatives for other operations
- Usual join algorithms to consider:
  - Block Nested Loops (BNL), typically always available and it is considered the baseline algorithm (reads the smallest table and tries to fit it in memory, then, the largest table is read in chunks, block-wise): [https://en.wikipedia.org/wiki/Nested\\_loop\\_join](https://en.wikipedia.org/wiki/Nested_loop_join) (however, DBMS typically iterate per block, not per tuple as discussed in the Wikipedia page)
  - Row Nested Loops (RNL), which benefits from existing indexing structures (B+, hash, etc.) in one of the join attributes. It is an extension of the previous one (as before, we can optimize this code working block-wise):

```
For each r in R do //assume the join is via S.C and R.C
  X <- index-lookup(INDEX ON (S.C), r.C) // the inner loop is replaced by an index look-up
For each s in X do
  join (r,s)
```
  - Sort Match (SM), which requires the join inputs (both) to be sorted according to the join attributes (e.g., by means of index clusters or any other structure preserving the order): [https://en.wikipedia.org/wiki/Sort-merge\\_join](https://en.wikipedia.org/wiki/Sort-merge_join)
  - Hash Join (HJ), which, as BNL, is always available (a smarter version of BNL that hashes the input tables using the same hash function on the join attribute): [https://en.wikipedia.org/wiki/Hash\\_join](https://en.wikipedia.org/wiki/Hash_join)

*In this course we do not assume you know the internals of these algorithms, but you are encouraged to look for more details about them. Nevertheless, you are expected to know the idea, as in the links provided, and therefore about the data structures each require*

# Phase 2: Estimating the Cost of the Alternatives Generated

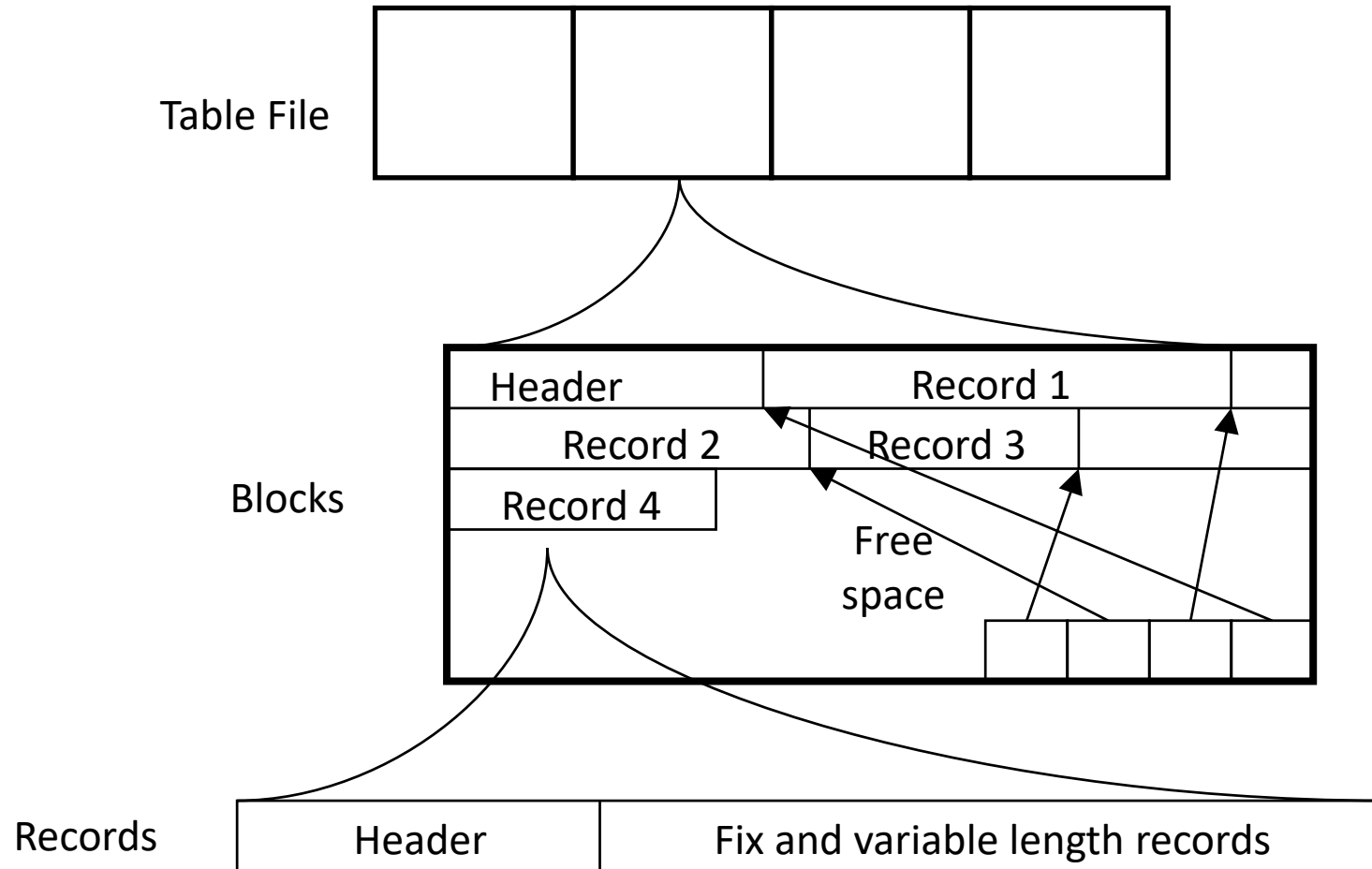
Details on how to estimate the cost of a given alternative process tree

# Cost-Based Estimation

- Most DBMS use cost-based formulas to estimate the cost of alternatives
- When based on cost formulas, the whole physical optimization process is call **cost-based optimization**
- Thus, internally, the DBMS have formulas that model the behaviour of the system when accessing data via the available data structures
  - Therefore, before moving on, be sure you master the database structures and you understand how they work
  - The formulas in this session model the behaviour of these data structures in front of different **access paths**

# Table Files

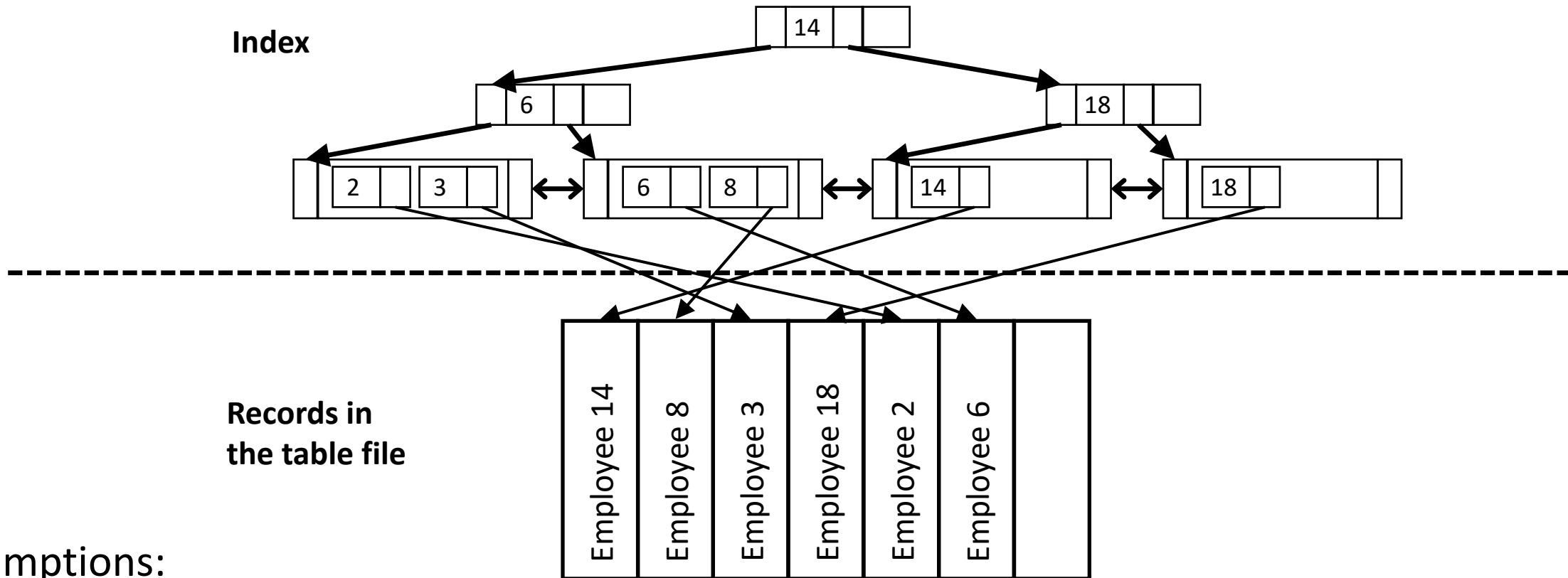
RECAP



**Note:** For the sake of simplicity, this slides ignore extensions and only show files and blocks

# B-Tree (B+)

RECAP

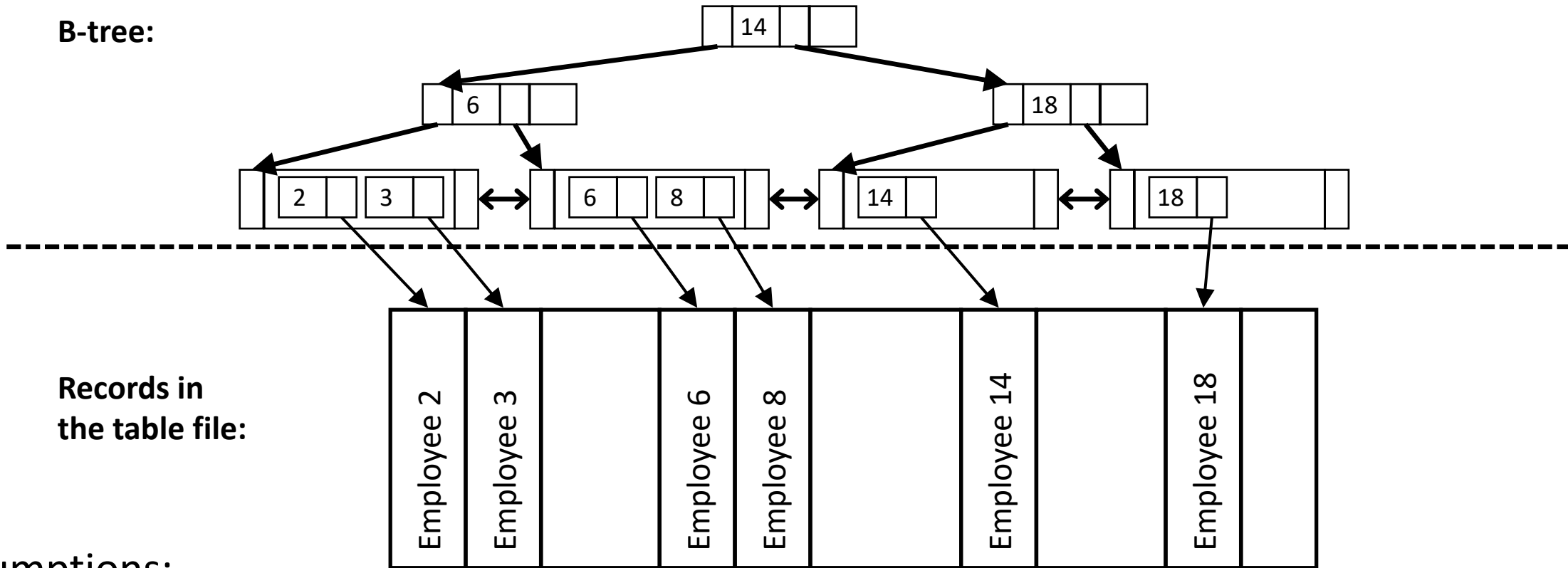


## Assumptions:

- In every tree node (a disk block)  $2d$  addresses fit;  $d$  is the tree order
- Usually, the tree load is 66% ( $2/3$  of its capacity)

# Clustered B-Tree (Index Cluster)

B-tree:



Records in  
the table file:

Employee 2	Employee 3		Employee 6	Employee 8		Employee 14		Employee 18	
------------	------------	--	------------	------------	--	-------------	--	-------------	--

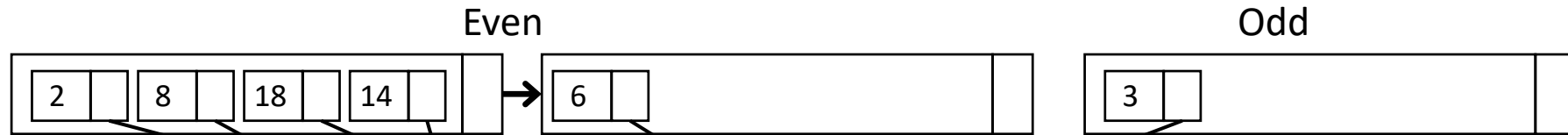
Assumptions:

Only 2/3 are used in every block (for index as well as for table blocks)

# Hash Index (Hash)

RECAP

Hash Index:



Records in  
the table file:

Employee 14	Employee 8	Employee 3	Employee 18	Employee 2	Employee 6	
-------------	------------	------------	-------------	------------	------------	--

Assumptions:

There are no blocks for excess

In a bucket block the same number of entries fit as in a tree block

Usually, bucket blocks are used at 80% (4/5 of their capacity)

# Cost Variables

- $|T|$ : Cardinality (i.e., number of tuples) of a table
- $B$ : Number of **full** blocks/pages needed to store all tuples
- $R$ : Number of records per block/page
- $D$ : Time to access (read or write) a disk block
  - Approximately 0'010 seconds
- $d$ : Tree order
  - Usually greater than 100
- $H$ : Cost of executing the hash function

We will **not** consider the improvement due to sequential scan nor cache!



# Estimating the Space Required per Structure

The next formulas estimate the space of each structure

- No index (regular heap table file)

- $B$  ← *Number of blocks to store all tuples*

- B+ over a table

- $\sum_{i=1}^{h+1} \lceil |T|/u^i \rceil + B$  ← *The first factor denotes the size of the tree; i.e., the number of intermediate nodes and leaves to index all tuples. The + B factor denotes the size of the table, which is created aside*

- Clustered B+

- $\sum_{i=1}^{h+1} \lceil |T|/u^i \rceil + \lceil 1.5B \rceil$  ← *Similar to the B+, but now, the table is not created aside but aligned with the clustered tree. Therefore, it must accommodate extra spaces to facilitate its management*

- Hash

- $1 + \lceil 1.25(|T|/2d) \rceil + B$  ← *The root structure is denoted by the first factor. The second factor is the number of buckets required to index all tuples. The third factor denotes the size of the table, which must be created aside*

$$u = \%load \cdot 2d = (2/3) \cdot 2d$$
$$h = \lceil \log_u |T| \rceil - 1$$

# Estimating the Cost of Each Possible Access Path

- A DBMS typically implements the following access paths to data:
  - Table Scan
  - Search one tuple
    - Equality of unique attribute
  - Search several tuples
    - Ranges
    - Equality of non-unique attribute
- Each of these access paths compute differently depending on the available physical structures (i.e., table file, B+, clustered B+ or hash)
- For each operator, the physical optimizer does as follows:
  1. Checks the operator and its predicate and **determines** which of the three **access paths** should execute
  2. Checks the **available structures** considering the attribute involved in the operator predicate
  3. **Applies cost formulas** to estimate the cost of that access path for each available structure

# Estimating the Cost of Each Possible Access Path

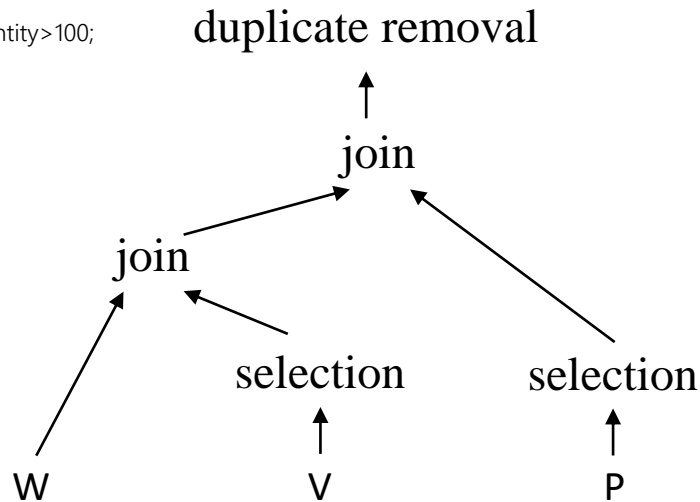
## Example:

```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineld=w.wineld
```

```
AND p.prodId=v.prodId
AND p.region="Priorat"
AND v.quantity>100;
```

Consider now the following structures:

- Vintages has a B+(quantity)
- Producers has a hash(region)
- Producers has a B+(region)
- Wines has an index cluster on (wineld)







## • Consider Selection (V):

1. The selection predicate ( $V.quantity > 100$ ) yields a **search several tuples** access path
2. Available structures: table file and B+(quantity)
3. Must execute two cost formulas:
  1. Search several tuples on table file
  2. Search several tuples via B+(quantity)

# Estimating a Table Scan Cost

Available structure:

- No index
  - $B \cdot D$   *Read the whole table*
- B+
  - $\lceil |T|/u \rceil \cdot D + |T| \cdot D$   *Number of leaves (first factor) and then we must access all tuples in the table*
- Table File of an Index Cluster
  - $\lceil 1.5B \rceil \cdot D$   *Read the whole table (but bear in mind the extra space harms the performance)*
- Hash
  - $\lceil 1.25(|T|/2d) \rceil \cdot D + |T| \cdot D$   *Read all buckets, considering the extra space and then read the tuples in the table*

$$u = \%load \cdot 2d = (2/3) \cdot 2d$$





**Only useful to sort**

**Always useless**

# Select One Tuple

Available structure:





$$u = \%load \cdot 2d = (2/3) \cdot 2d$$
$$h = \lceil \log_u |T| \rceil - 1$$

- No index
  - $0.5B \cdot D$   *In average, reading half of the table we will find it (thus, it is an estimation of the average case)*
- B+
  - $h \cdot D + D$   *Access via the tree, find the leaf with the interesting tuple and access the table*
- Clustered
  - $h \cdot D + D$   *Access via the tree, find the leaf with the interesting tuple and access the table*
- Hash
  - $H + D + D$   *Execute the hash function, access the identified bucket and access the table*

# Select Several Tuples

$u = \%load \cdot 2d = (2/3) \cdot 2d$   
 $h = \lceil \log_u |T| \rceil - 1$   
 $O$  = tuples in the output  
 $v$  = values in the range  
 $k$  = repetitions per value

## Available structure:

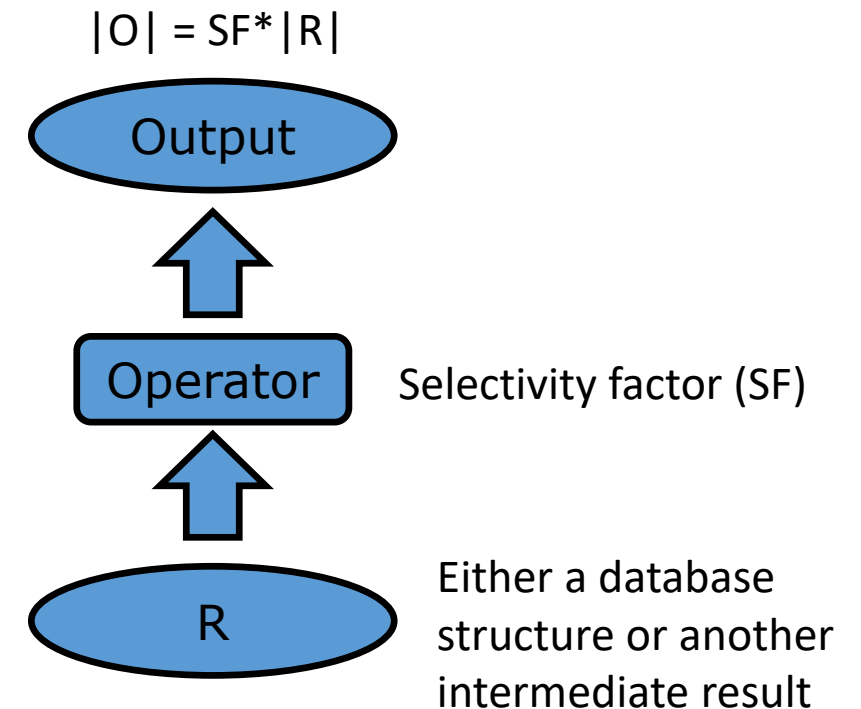
- No index
  - $B \cdot D$   *We must read the whole table*
- B+
  - $h \cdot D + ((|O|-1)/u) \cdot D + |O| \cdot D$   *Access via the tree, find the leaf with the first value in the range. From there, scan the following tree leaves until all tuples are identified. For each of them, access the table*
- Clustered
  - $h \cdot D + D + (1.5(|O|-1)/R) \cdot D$   *Access via the tree, find the leaf with the first value in the range and access its position in the table. From there, read the following records in the table belonging to the range of interest*
- Hash
  - $v=1: H + D + k \cdot D$
  - $v>1: v \cdot (H + D + k \cdot D)$   *If  $v=1$  we execute the hash function, find the only bucket containing all values in the range and access those  $k$  elements in the table.*
  - $v$  is unknown: Useless *If  $v>1$  the idea is the same but repeated  $v$  times (once per bucket accessed)*

# Cardinality and Size Estimation for the Intermediate Results

Required statistics to compute costs

# On the Need of Statistics on the Intermediate Results

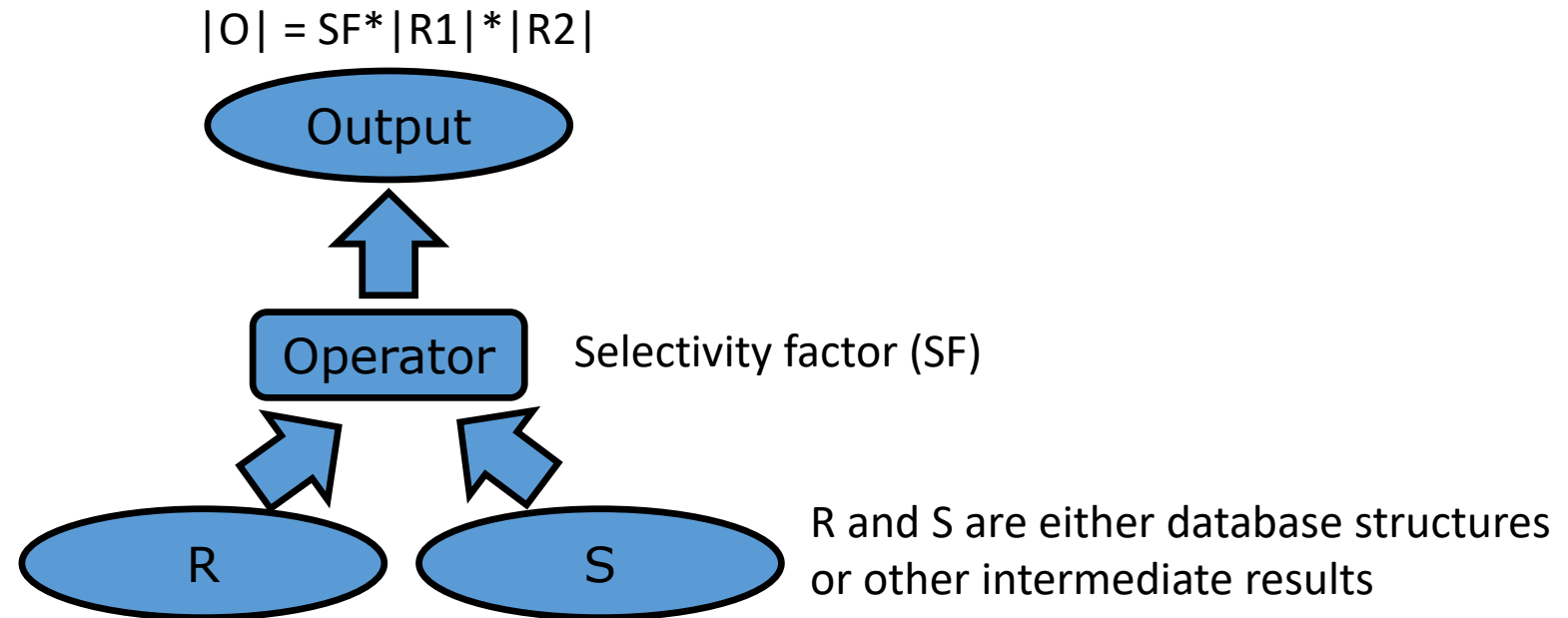
- Check with care the previous cost formulas we have discussed
  - $|T|$ ,  $B$ ,  $R$ ,  $d$  are variables that any DBMS compute beforehand for its structures
  - These values are stored in the so-called **database catalog**
- However, when an operation reads from an intermediate result these values are not available in the database catalog and must be estimated on the fly
  - Therefore, the DBMS must estimate these variables for intermediate results
  - In order to do so, it relies on the concept of **selectivity factor** ( $0 \leq SF \leq 1$ ) of an operation
    - Next to 0 means the operation is very selective (E.g.,: ID)
    - Next to 1 means it is not selective (E.g.,: Sex)





# On the Need of Statistics on the Intermediate Results

- We do similarly for binary operators, but the SF applies over the cartesian product of the two inputs



- The SFs of the operators in a process are always calculated from the tree leaves to the root
  - Therefore, it fully relies on the statistics stored in the database catalog

# Gathering Statistics: Oracle (Example)

a) `ANALYZE [TABLE | INDEX | CLUSTER] <name> [COMPUTE | ESTIMATE] STATISTICS;`

```
ANALYZE TABLE departments COMPUTE STATISTICS;  
ANALYZE TABLE employees COMPUTE STATISTICS;
```

b) `DBMS_STATS.GATHER_TABLE_STATS( <esquema>, <table> );`

```
DBMS_STATS.GATHER_TABLE_STATS("username","departments");  
DBMS_STATS.GATHER_TABLE_STATS("username","employees");
```

# Statistics

- The DBA is responsible for keeping the required statistics fresh
- Example of usual statistics gathered in the database catalog
  - Regarding relations:
    - Cardinality ( $|T|$ )
    - Number of blocks ( $B$ )
    - Average length of records ( $B/R$ )
  - Regarding attributes:
    - Length
    - Domain cardinality (maximum number of different values)
    - Number of existing different values
    - Maximum value
    - Minimum value
- When propagating these statistics, DBMS make two main hypotheses:
  - Uniform distribution of values for each attribute
  - Independence of attributes

# Selectivity factor of a Selection (I)

- Assuming equi-probability of values
  - $SF(A=c) = 1/ndist(A)$
- Assuming uniform distribution and  $A \in [min, max]$ 
  - $SF(A > c) = (max - c) / (max - min)$ 
    - $SF(A > c) = 0$  (if  $c \geq max$ )
    - $SF(A > c) = 1$  (if  $c < min$ )
  - $SF(A < c) = (c - min) / (max - min)$ 
    - $SF(A < c) = 1$  (if  $c > max$ )
    - $SF(A < c) = 0$  (if  $c \leq min$ )
- Assuming  $ndist(A)$  big enough
  - $SF(A \leq c) = SF(A < c)$
  - $SF(A \geq c) = SF(A > c)$

*\*In bold, statistics got from the DB catalog*

# Selectivity factor of a Selection (II)

Selections may have composite predicates (i.e., simple predicates composed by logic operators):

- Assuming P and Q are statistically **independent**
  - $SF(P \text{ AND } Q) = SF(P) \cdot SF(Q)$
  - $SF(P \text{ OR } Q) = SF(P) + SF(Q) - (SF(P) \cdot SF(Q))$
- $SF(\text{NOT } P) = 1 - SF(P)$
- $SF(A \text{ IN } (c_1, c_2, \dots, c_n)) = \min(1, n / \text{ndist}(A))$
- $SF(A \text{ BETWEEN } c_1 \text{ AND } c_2) = (\min(c_2, \text{max}) - \max(c_1, \text{min})) / (\text{max} - \text{min})$

# Selectivity factor of a Join

- It is difficult to approximate the general case  $R[A \theta B]S$ 
  - Usually, the required statistics are not available because it would be too expensive to maintain them. For these cases, the SF is roughly estimated:
    - $SF(R[A \times B]S) = 1$
    - $SF(R[A < > B]S) = 1$
    - $SF(R[A < B]S) = \frac{1}{2}$
    - $SF(R[A \leq B]S) = \frac{1}{2}$
- Only for equi-joins they apply smarter estimations:
  - $SF(R[A=B]S) = 1/\max(\text{ndist}(A), \text{ndist}(B))$ 
    - Assuming there is no FK
  - $SF(R[A=B]S) = 1/|R|$ 
    - Assuming S.B is not null,
    - S.B is FK to R.A and
    - R.A is PK

# Selectivity factor of a Join

The previous formulas do not consider that some previous operations may have removed\* some tuples from the original tables (remember the SF is always computed from leaves to the root). In this case, these SF formulas need to be tuned accordingly:

- For example:
  - Let  $R, S$  be two tables and  $x$  and  $y$  be their join attributes, where  $S.y$  is a FK to  $R.x$
  - Consider a selection prior to the join that filters out 60% of  $R$  tuples (i.e., the selection  $SF = 0.4$ )
    - Let  $R'$  be the result after the selection on  $R$
    - Considering that the join  $SF = 1/|R'|$  would be wrong because:
      - We are considering that  $|O| = |S|$ , where  $O$  is the join output
      - However, 60% of the  $R$  tuples were filtered out before the join and therefore some tuples from  $S$  will not join any tuple from  $R'$
    - Therefore, the SF must be fine-tuned assuming uniform data distribution:
      - Intuitively, assuming a uniform distribution, we should introduce a weighting factor in the SF to estimate the number of tuples in  $S$  that will not join any tuple of  $R'$  (proportional to the number of tuples filtered out)
      - Therefore,  $SF = (1/|R'|) * (ndist(x, R') / ndist(x, R))$ , where  $ndist(x, R)$  means the number of distinct values in  $R$  and  $R'$  (after the selection), respectively, is a better estimation

\*The same problem arises in the presence of NULL values

# Estimation of Intermediate Results

Once we know the number of records in the output (thanks to the selectivity factor), we can then compute the rest of relevant statistics:

- Record length
  - $\sum \text{attribute length}_i$  (+ control information)
- Number of records per block
  - $R_R = \lfloor \text{block size} / \text{record length} \rfloor$
- Number of blocks per table
  - $B_R = \lceil |R| / R_R \rceil$



# Summary

- Generating the process tree
- Physical optimization
  - Generation of alternatives
    - Data structures available
    - Alternative algorithms to execute an operation
  - Cost-based estimation per alternative
    - Cost formulas per access path and data structure available
      - Access path: all tuples, one tuple, several tuple
      - Data structures: table file, B+, clustered B+, hash index
    - The relevance of statistics in cost-based optimization
      - The database catalog
    - Estimation of statistics for the intermediate results
      - Selectivity factor (selections and joins)
      - Main hypotheses to propagate costs
        - Uniform distribution
        - Independence of attributes
  - Choose best alternative

# Bibliography

- Y. Ioannidis. *Query Optimization*. ACM Computing Surveys, vol. 28, num. 1, March 1996
- R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3<sup>rd</sup> Edition, 2003
- J. Lewis. *Cost-Based Oracle Fundamentals*. Apress, 2006
- S. Lightstone, T. Teorey and T. Nadeau. *Physical Database Design*. Morgan Kaufmann, 2007
- M. Golfarelli and S. Rizzi. *Data Warehouse Design*. McGraw-Hill, 2009
- C. S. Jensen, T. B. Pedersen, C. Thomsen. *Multidimensional Databases and Data Warehousing*. Morgan&Claypool Publishers, 2010
- Oracle Database. *Data Warehousing Guide*. 11g Release 2, September 2011