NIE-PDB: **Advanced Database Systems**

http://www.ksi.mff.cuni.cz/~svoboda/courses/231-NIE-PDB/

Lecture 12

Query Evaluation

Martin Svoboda

martin.svoboda@fit.cvut.cz

12. and 19. 12. 2023

Charles University, Faculty of Mathematics and Physics

Czech Technical University in Prague, Faculty of Information Technology

Lecture Outline

Algorithms

- **Access methods**
- **External sort**
- **Nested loops join**
- **Sort-merge join**
- **Hash join**

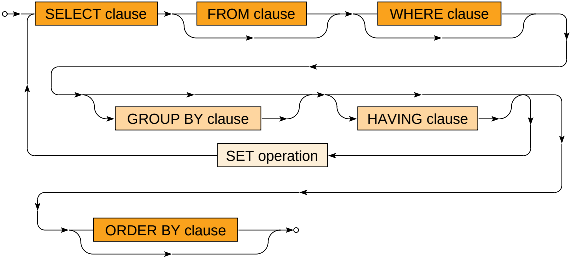
Evaluation

- **Query evaluation plans**
- **Optimization** techniques

Introduction

SQL queries

- `SELECT` **statements**



Introduction

Relational algebra

- Basic and inferred operations
  - Selection σ_φ, projection $\pi_{a_1,\ldots,a_n}$, renaming $\rho_{b_1/a_1,\ldots,b_n/a_n}$
  - Set operations: union $\cup$, intersection $\cap$, difference $\setminus$
  - Inner joins: cross join $\times$, natural join $\bowtie$, theta join $\bowtie_\varphi$
  - Left / right natural / theta semijoin $\ltimes$, $\rtimes$, $\ltimes_\varphi$, $\rtimes_\varphi$
  - Left / right natural / theta antijoin $\triangleright$, $\triangleleft$, $\triangleright_\varphi$, $\triangleleft_\varphi$
  - Division $\div$
- Extended operations
  - Left / right / full outer natural join ⟕, ⟖, ⟗
  - Left / right / full outer theta join ⟕$_\varphi$, ⟖$_\varphi$, ⟗$_\varphi$
  - Sorting, grouping and aggregation, distinct, …

Naïve Algorithms

Selection: $\sigma_{\varphi}(E)$

- Iteration over all tuples and removal of those filtered out

Projection: $\pi_{a_1,\ldots,a_n}(E)$

- Iteration over all tuples and removal of excluded attributes
    - But also removal of duplicates within the traditional model

Distinct

- Sorting of all tuples and removal of adjacent duplicates

Inner joins: $E_R \times E_S$, $E_R \bowtie E_S$, $E_R \bowtie_{\varphi} E_S$

- Iteration over all the possible combinations via nested loops

Sorting

- Quick sort, heap sort, bubble sort, insertion sort, …

Challenges

Blocks

- **Tuples** stored in data files are **not accessible directly**
  - Since we have **read / write operations** for **whole blocks** only
- That is true for all types of files…
  - And so not just **data files** for tables
  - But also files for **index structures** or **system catalog**

Latency

- Traditional **magnetic hard drives are extremely slow**
  - Efficient management of cached pages is hence essential

Memory

- Size of **available system memory** is always limited

$\Rightarrow$ **external algorithms** are needed

Objectives

Query evaluation plan

- Based on the **database** context and **available** memory…
  … suitable **evaluation** algorithms need to be selected…
  … so that the overall **evaluation** cost is **minimal**

Database context

- **Relational schema**: tables, columns, data types
- **Integrity constraints**: primary / unique / foreign keys, …
- **Data organization**: heap / sorted / hashed file
- **Index structures**: B^+ tree, bitmap index, hash index
- **Available statistics**: min / max values, histograms, …

Objectives

Available system memory

- **Number of pages** allocated for the **execution of a given query**
- There are two possible scenarios…
  - Having a particular **memory size**…
    - Propose its usage and estimate the evaluation cost
  - Having a particular **cost expectation**…
    - Determine the required memory and propose its usage

Evaluation algorithms

- **Access methods**
- **Sorting**: external sort approaches
- **Joining**: nested loops, merge join, and hash join approaches
- …

Objectives

Cost estimation

- Expressed in terms of **read / write disk operations**
  - Since hard drives are extremely slow, as already stated…
    - And so everything else can boldly be ignored
- We are interested in **estimates** only
  - Since it is unlikely we could provide accurate calculations
  - But still…
    - **The more accurate estimates, the better evaluation plans**
  - And there can really be huge differences in their efficiency…
    - Even up to **several orders of magnitude**!
- In other words…
  - **Query optimization is crucial for any database system**
  - As well as we also need to know what we are doing…

Available Statistics

Environment

- B: size of a block / page, usually $\approx 4\,kB$
- M: number of available **system memory** pages

Relation $\mathcal{R}$

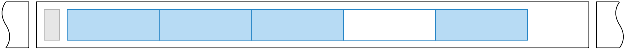
- n_R: **number of tuples**
- s_R: average / fixed tuple size
- $b_R \approx \lfloor B/s_R \rfloor$: **blocking factor**
  - Number of tuples that can be stored within one block
- $p_R \approx \lceil n_R/b_R \rceil$: **number of blocks**
- $V_{R.A}$: cardinality of the **active domain** of attribute A
  - Number of distinct values of A occurring in $\mathcal{R}$
- $min_{R.A}$ and $max_{R.A}$: minimal and maximal values for A

Access Methods

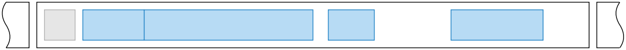
Data Files

Internal structure

- **Blocks** of data files for tables are **divided into slots**
  - Each slot is intended for storing exactly one tuple
    - By the way, they can easily be uniquely identified
    - Using a pair of **block and slot logical ordinal numbers**
- **Fixed-size slots**
  - Usage status of each slot just needs to be remembered



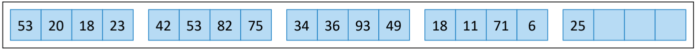
- **Variable-size slots**
  - When at least one variable-size attribute is involved
  - Slot **beginnings** and **lengths** need to be remembered



Heap File

Heap file

- Tuples are put into individual slots entirely **arbitrarily**
  - I.e., we do not have any specific knowledge of their position



Selection costs

- **Full scan** is inevitable in **almost all situations**
  - $c = p_R$
- **Equality test** with respect to a **unique attribute**
  - $c = \lceil p_R/2 \rceil$
    - Since we can stop at the moment a given tuple is found
    - However, uniform distribution of data and queries is assumed
    - And values outside of the active domain may also be queried

Sorted File

Sorted file

- Tuples are **ordered** with respect to a particular attribute

6	11	18	18	20	23	25	34	36	42	49	53	53	71	75	82	93			

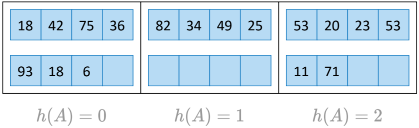
Selection costs

- Binary search (**half-interval search**) can be used in general
  - However, only when the same attribute is queried, of course
    - I.e., the same attribute as the one used for sorting
    - Otherwise, sequential read as in a heap file would be needed
- **Equality test**
  - $c = \lceil \log_2 p_R \rceil$ for a **unique** attribute
  - $c = \lceil \log_2 p_R \rceil + \lceil p_R / V_{R.A} \rceil$ for a **non-unique** attribute
- Various **range queries**

Hashed File

Hashed file

- Tuples are put into disjoint **buckets** (logical groups of blocks)
  - Based on a selected **hash function** over a particular attribute
    - E.g., $h(A) = A \bmod 3$



- **Hash function**
  - Its **domain** are values of a given attribute A
  - Its **range** provides H distinct values
    - There is exactly one bucket for each one of them
    - All tuples in a bucket always share the same hash value

Hashed File

File statistics

- H_R: **number of buckets**
- $C_R \approx \lceil p_R / H_R \rceil$: expected **bucket size**
  - Measured as a number of blocks in a bucket

Selection costs

- **Equality test** when the hashing attribute is queried
  - Only the corresponding bucket needs to be accessed
  - $c = C_R$ for a **non-unique** attribute
  - $c = \lceil C_R / 2 \rceil$ for a **unique** attribute
    - Similar assumptions as in the case of heap files
- Any other condition
  - $c = p_R$
    - I.e., **full scan** is needed

B^+ Tree Index

B^+ tree index structure = **self-balanced search tree**

- **Logarithmic height** is guaranteed (the same across all leaves)
- Moreover, very **high fan-out** is assumed
  - I.e., our trees will tend to be significantly **wider than taller**
    - $\Rightarrow$ **search times** will not only be **logarithmic**, but also really low

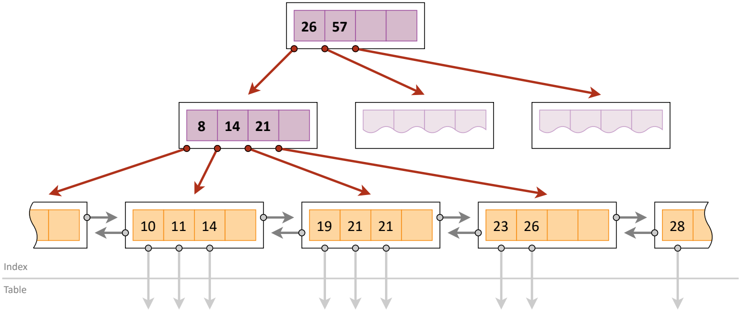
Logical structure

- **Internal node** (including a non-leaf root node)
  - Contains an ordered sequence of **dividing values** and **pointers to child nodes** representing the **sub-intervals** they determine
- **Leaf node**
  - Contains individual values and **pointers to tuples** in data file
  - Leaves are also interconnected by pointers in both directions

B$^+$ Tree Index

B$^+$ tree index structure (cont'd)

- Sample index for relation $\mathcal{R}$ and its attribute A



B^+ Tree Index

Physical structure

- **Each node** is physically represented by **one index file block**
  - And so they are treated the same way as data file blocks
    - I.e., loaded into the system memory one by one, etc.

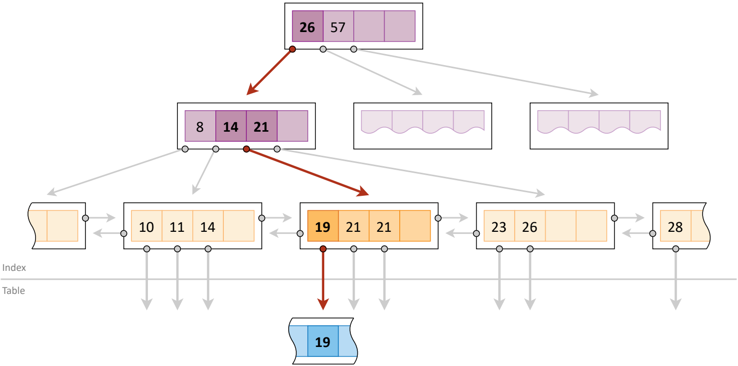
Index statistics

- $m_{R.A}$: maximal **number of children** (order of tree)
  - Usually up to small hundreds in practice
  - Actual number is guaranteed to be at least $\lceil m_{R.A}/2 \rceil$
    - Except for the root node
- $I_{R.A}$: **index height**
  - Usually just $\approx 2 - 3$ for typical real-world tables
- $p_{R.A}$: number of **leaf nodes**

B$^+$ Tree Index

Search algorithm

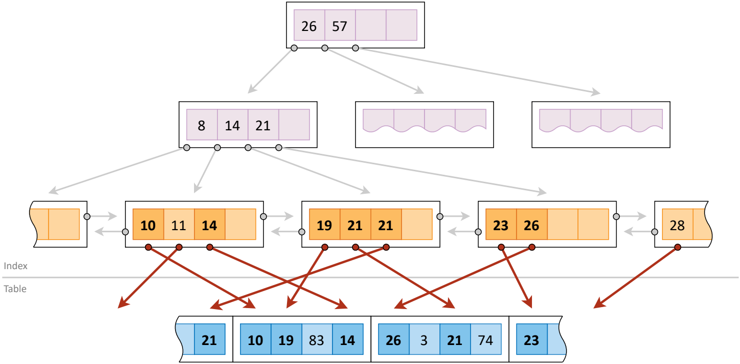
- Index is traversed from its root toward the corresponding leaf
  - Data tuple then needs to be fetched from the data file



Non-Clustered B$^+$ Tree Index

Non-clustered **index**

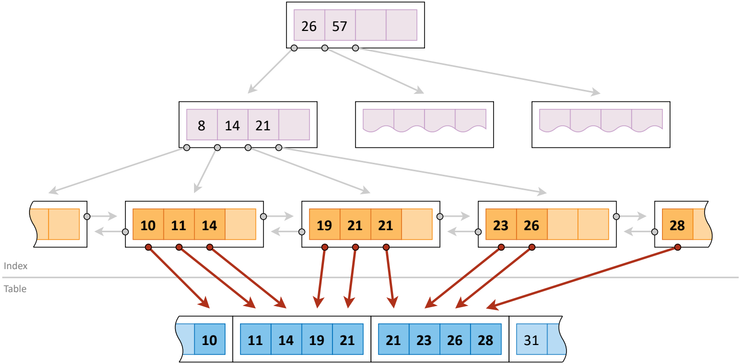
- Order of items within the leaves and data file is not the same
  - I.e., data file is organized as a **heap file** of **hashed file**



Clustered B$^+$ Tree Index

Clustered **index**

- On the contrary, order of items is (at least almost) the same
  - I.e., data file is a **sorted file** (with respect to the same attribute)



Selection costs

Non-clustered B$^+$ tree index

- **Equality test** for a **unique** / **non-unique** attribute
  - $c = I_{R.A} + 1$
  - $c = I_{R.A} + \lceil p_{R.A} / V_{R.A} \rceil + \min(p_R, \lceil n_R / V_{R.A} \rceil)$
- Various **range queries**
- …

Clustered B$^+$ tree index

- **Equality test** for a **unique** / **non-unique** attribute
  - $c = I_{R.A} + 1$
  - $c = I_{R.A} + \lceil p_R / V_{R.A} \rceil$
- Various **range queries**
- …

Examples

Sample scenario #1

- **Movie** (id, title, year, …)
  - Basic statistics
    - $n_M = 100\,000$ tuples, $b_M = 10$, $p_M = 10\,000$ blocks
    - $V_{M.id} = n_M = 100\,000$ values (since they are unique)
  - Heap file
  - Sorted file (using ids)
  - Hashed file
    - $h(M.id) = M.id \bmod 50$
    - $H_M = 50$ buckets, $C_M = 200$ blocks
  - B$^+$ tree index (using ids)
    - $m_{M.id} = 100$ followers
    - $I_{M.id} = 3$, $p_{M.id} = 1\,500$ blocks

Examples

Equality test: movie with a particular identifier

- Heap file
  - $c = \lceil p_M/2 \rceil = 5\,000$
- Sorted file
  - $c = \lceil \log_2 p_M \rceil = 14$
- Hashed file
  - $c = \lceil C_M/2 \rceil = 100$
- Non-clustered index (B^+ tree & heap file)
  - $c = I_{M.year} + 1 = 4$
- Clustered index (B^+ tree & sorted file)
  - $c = I_{M.year} + 1 = 4$

Examples

Sample scenario #2

- **Movie** (id, title, year, …)
  - Basic statistics
    - $n_M = 100\,000$ tuples, $b_M = 10$, $p_M = 10\,000$ blocks
    - $V_{M.year} = 50$ values
    - $min_{M.year} = 1943$, $max_{M.year} = 2022$ (i.e., 80 values)
  - Heap file
  - Sorted file (using years)
  - Hashed file
    - $h(M.year) = M.year \bmod 20$
    - $H_M = 20$ buckets, $C_M = 500$ blocks
  - B$^+$ tree index (using years)
    - $m_{M.year} = 100$ followers
    - $I_{M.year} = 3$, $p_{M.year} = 1\,500$ blocks

Examples

Equality test: movies filmed in a particular year

- Heap file
  - $c = p_M = 10\,000$
- Sorted file
  - $c = \lceil \log_2 p_M \rceil + \lceil p_M / V_{M.year} \rceil = 214$
- Hashed file
  - $c = C_M = 500$
- Non-clustered index (B$^+$ tree & heap file)
  - $c = I_{M.year} + \lceil p_{M.year} / V_{M.year} \rceil + \min(p_M, \lceil n_M / V_{M.year} \rceil)$ $= 2\,033$
- Clustered index (B$^+$ tree & sorted file)
  - $c = I_{M.year} + \lceil p_M / V_{M.year} \rceil = 203$

External Sort

External Sort

N-way external merge sort

- **Sort phase** (**pass 1**)
  - Groups of input blocks are loaded into the system memory
  - Tuples in these blocks are then sorted
    - Any **in-memory in-place sorting algorithm** can be used
    - E.g.: *quick sort*, *heap sort*, *bubble sort*, *insertion sort*, …
  - Created initial **runs** are written into a temporary file
- **Merge phase** (**passes 2 and higher**)
  - Groups of runs are loaded into the memory and merged
  - Newly created (longer) runs are written back on a hard drive
  - Merging is finished when exactly one run is obtained
    - And so the entire input table is sorted

Sort Phase

Pass 1

- Input data file
  - **Relational table $\mathcal{R}$**
    - E.g., $n_R = 18$ tuples, $b_R = 4$ tuples/block, $p_R = 5$ blocks

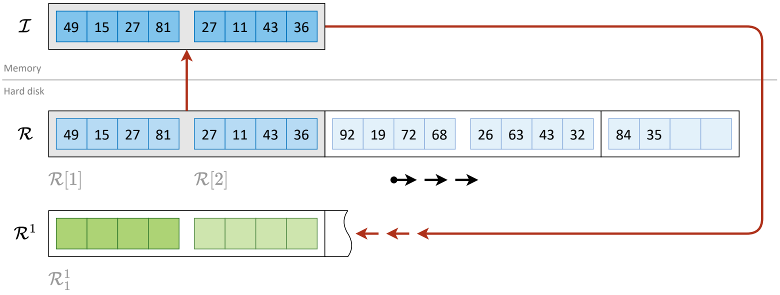
	$\mathcal{R}[1]$				$\mathcal{R}[2]$				$\mathcal{R}[3]$				$\mathcal{R}[4]$				$\mathcal{R}[5]$			
$\mathcal{R}$	49	15	27	81	27	11	43	36	92	19	72	68	26	63	43	32	84	35		

- System memory layout
  - **Input buffer $\mathcal{I}$**
    - E.g., size $M = 2$ pages

Sort Phase

Pass 1

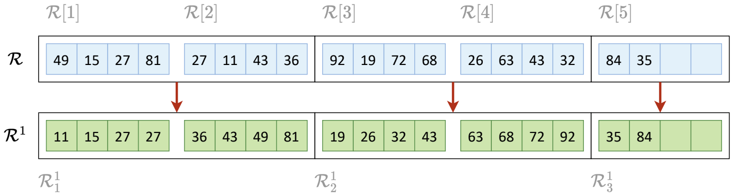
- **Groups of M blocks** are presorted and so **initial runs** created
  - Input blocks from $\mathcal{R}$ are first loaded to $\mathcal{I}$
    - Individual tuples in $\mathcal{I}$ are then sorted
    - Created runs are stored to a temporary file $\mathcal{R}^1$



Sort Phase

Pass 1

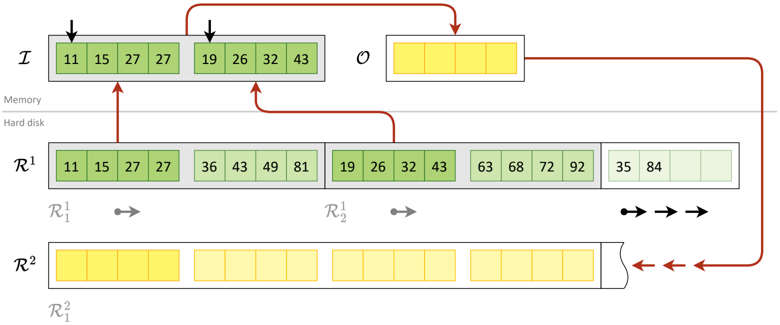
- **Resulting runs** in $\mathcal{R}^1$ within our sample scenario



Merge Phase

Pass 2

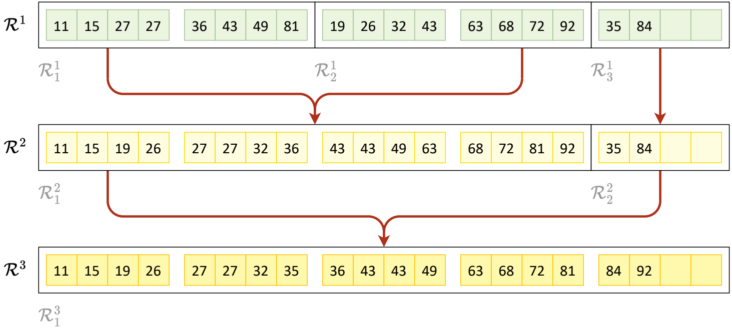
- **Groups of M runs** are iteratively **merged together**
  - Blocks from these input runs are gradually loaded into $\mathcal{I}$
    - Minimal items are then iteratively selected and moved to $\mathcal{O}$
    - Merged (longer) runs are written to a new temporary file $\mathcal{R}^2$



Merge Phase

Passes 2 and 3

- Merging continues until just a single run is acquired
  - And so the entire input table is sorted



Algorithm

Sort phase (pass 1)

```
p ← 1
foreach group of blocks B_1, ..., B_M (if any) from R do
    read these blocks to I
    sort all items in I
    write all blocks from I as a new run to R^p
```

Algorithm

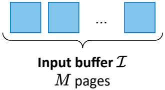
Merge phase (passes 2 and higher)

while $\mathcal{R}^p$ has more then just one run **do**
    $p \leftarrow p + 1$
    foreach group of runs $R_1, \ldots, R_M$ (if any) from $\mathcal{R}^{p-1}$ **do**
        start constructing a new run in $\mathcal{R}^p$
        read the first block from each run R_x to $\mathcal{I}[x]$
        while $\mathcal{I}$ contains at least one item **do**
            select the minimal item and move it to $\mathcal{O}$
            if the corresponding $\mathcal{I}[x]$ is empty **then**
                read the next block from R_x (if any) to $\mathcal{I}[x]$
            if $\mathcal{O}$ is full **then** write $\mathcal{O}$ to $\mathcal{R}^p$ and empty $\mathcal{O}$
        if $\mathcal{O}$ is not empty **then** write $\mathcal{O}$ to $\mathcal{R}^p$ and empty $\mathcal{O}$

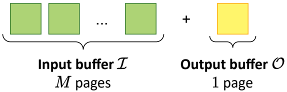
Summary

Memory layout

- Sort phase (**pass 1**): M
  - **Input buffer $\mathcal{I}$**: M pages



- Merge phase (**passes 2 and higher**): $M+1$
  - **Input buffer $\mathcal{I}$**: $M \geq 2$ pages
  - **Output buffer $\mathcal{O}$**: 1 page



Summary

Time complexity

- **Single pass** (regardless of the phase)
  - $c_{\texttt{read}} = c_{\texttt{write}} = p_R$
- **Number of passes**
  - $t = \lceil \log_M(p_R) \rceil$
- **Overall cost**
  - $c_{\texttt{ES}} = t \cdot (c_{\texttt{read}} + c_{\texttt{write}}) = \lceil \log_M(p_R) \rceil \cdot 2p_R$

Limitation of the overall **number of passes**

- In general…
  - $M = \lceil \sqrt[t]{p_R} \rceil$
- Specifically for $t = 2$ (i.e., **exactly 2 passes**)
  - $M = \lceil \sqrt{p_R} \rceil$

Nested Loops Join

Nested Loops

Binary nested loops

- **Universal approach** for all types of **inner joins**
  - Natural join, cross join, theta join
    - I.e., arbitrary joining condition can be involved
  - Support possible duplicates
  - Requires no index structures
- Not the best option in all situations, though
  - Suitable for tables with significantly different sizes

Basic idea

- **Outer loop**: iteration over the blocks of the **first** table
- **Inner loop**: iteration over the blocks of the **second** table

Nested Loops

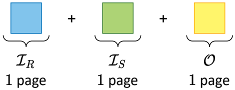
Sample input data

- Tables $\mathcal{R}$ and $\mathcal{S}$ to be joined using a **value equality** test

	Page 1				Page 2				Page 3				Page 4				Page 5			
$\mathcal{R}$	**21**	84	**56**	19	41	**72**	69	**35**	**56**	84										
$\mathcal{S}$	31	**56**	75	43	88	**21**	43	14	92	52	25	81	**72**	37	64	**35**	14	64		

Basic setup

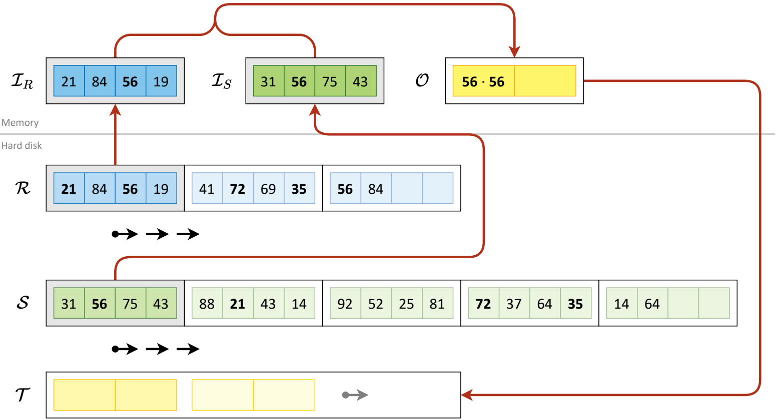
- Memory layout: $1 + 1 + 1$
  - **Input buffer** $\mathcal{I}_R$: 1 page
  - **Input buffer** $\mathcal{I}_S$: 1 page
  - **Output buffer** $\mathcal{O}$: 1 page



Nested Loops

Basic setup $(1 + 1 + 1)$

- Another pair of loops is used for joining tuples in the memory



Algorithm

Basic setup $(1+1+1)$

foreach block R from $\mathcal{R}$ **do**
    read R into $\mathcal{I}_R$
    foreach block S from $\mathcal{S}$ **do**
        read S into $\mathcal{I}_S$
        foreach item r in $\mathcal{I}_R$ **do**
            foreach item s in $\mathcal{I}_S$ **do**
                if r and s satisfy the join condition **then**
                    join r and s and put the result to $\mathcal{O}$
                    if $\mathcal{O}$ is full **then** write $\mathcal{O}$ to $\mathcal{T}$, empty $\mathcal{O}$
if $\mathcal{O}$ is not empty **then** write $\mathcal{O}$ to $\mathcal{T}$ and empty $\mathcal{O}$

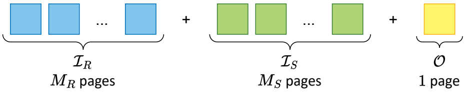
Observations

Time complexity

- Basic setup $(1 + 1 + 1)$
  - $c_{\texttt{NL}} = p_R + p_R \cdot p_S$
- $\Rightarrow$ **smaller table** should always be taken as the **outer** one

General setup

- Multiple pages are used for both the input buffers
- Memory layout: $M_R + M_S + 1$
  - **Input buffer** $\mathcal{I}_R$: M_R pages
  - **Input buffer** $\mathcal{I}_S$: M_S pages
  - **Output buffer** $\mathcal{O}$: 1 page



Algorithm

General setup ($M_R + M_S + 1$)

```
foreach group of blocks R_1, ..., R_{M_R} (if any) from 𝓡 do
    read these blocks into 𝓘_R
    foreach group of blocks S_1, ..., S_{M_S} (if any) from 𝓢 do
        read these blocks into 𝓘_S
        foreach item r in 𝓘_R do
            foreach item s in 𝓘_S do
                if r and s satisfy the join condition then
                    join r and s and put the result to 𝓞
                    if 𝓞 is full then  write 𝓞 to 𝓣, empty 𝓞
if 𝓞 is not empty then  write 𝓞 to 𝓣 and empty 𝓞
```

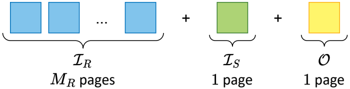
Observations

Time complexity

- General setup ($M_R + M_S + 1$)
  - $c_{\texttt{NL}} = p_R + \lceil p_R / M_R \rceil \cdot p_S$
- $\Rightarrow$ there is no reason of having $M_S \geq 2$

Standard setup

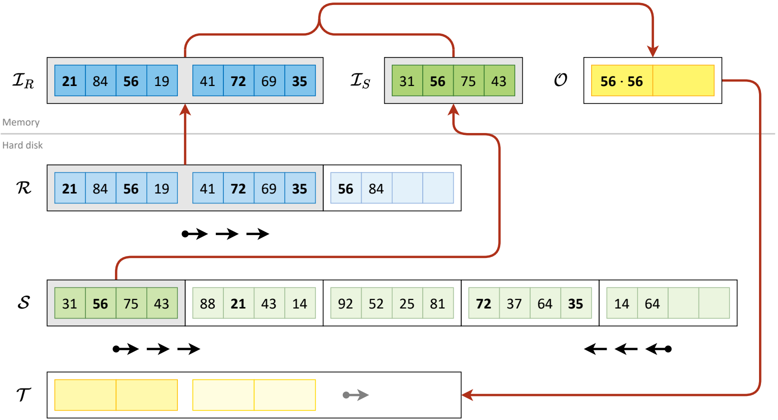
- Memory layout: $M_R + 1 + 1$
  - **Input buffer $\mathcal{I}_R$**: M_R pages
  - **Input buffer $\mathcal{I}_S$**: 1 page
  - **Output buffer $\mathcal{O}$**: 1 page



Standard Approach

Standard setup $(M_R + 1 + 1)$ with **zig-zag** optimization

- Multiple pages are used just for the outer table



Observations

Zig-zag optimization

- Reading of the inner table $\mathcal{S}$
  - **Odd iterations** normally
  - **Even iterations** in reverse order

Time complexity

- Standard setup ($M_R + 1 + 1$)
  - $c_{\texttt{NL}} = p_R + \lceil p_R / M_R \rceil \cdot p_S$ (without zig-zag)
  - $c_{\texttt{NL}} = p_R + \lceil p_R / M_R \rceil \cdot (p_S - 1) + 1$ (with zig-zag)

Special cases

- Smaller table fits entirely within the memory, i.e., $p_R \leq M_R$
  - $c_{\texttt{NL}} = p_R + p_S$
- **Non-brute-force** replacement for **inner loops**
  - When a suitable index exists on the inner table, …

Sort-Merge Join

Sort-Merge Join

Sort-merge join algorithm (or just **merge join**)

- Inner joins based on **value equality tests** only
  - Basic version **without duplicates**
    - Could be extended to support them, though
- Suitable for tables with relatively similar sizes
  - Especially when they are already sorted
  - Or when the final result is expected to be sorted

Phases

- **Sort phase**
  - Both tables are externally sorted, one by one (if not yet)
- **Join phase**
  - Items are joined while simulating the merge of the two tables

Basic Approach

Sample input data

- **Input tables $\mathcal{R}$ and $\mathcal{S}$**

$\mathcal{R}$	65 **19** 35 **92**	**49** 31			
$\mathcal{S}$	52 94 38 71	**92** 41 63 **19**	75 54 46 68	15 27 22 43	11 50 **49**

Sort phase

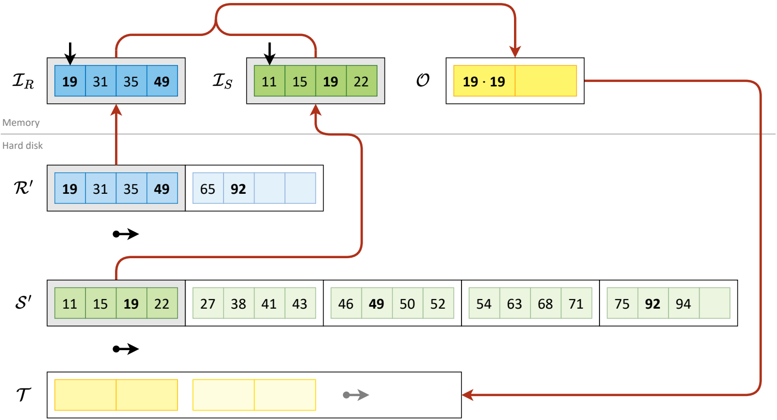
- Resulting sorted tables

$\mathcal{R}'$	**19** 31 35 **49**	65 **92**			
$\mathcal{S}'$	11 15 **19** 22	27 38 41 43	46 **49** 50 52	54 63 68 71	75 **92** 94

Basic Approach

Join phase

- Blocks from the sorted tables are processed one by one



Algorithm

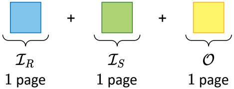
Join phase

read block $\mathcal{R}'[1]$ to $\mathcal{I}_R$ and block $\mathcal{S}'[1]$ to $\mathcal{I}_S$
while both $\mathcal{I}_R$ and $\mathcal{I}_S$ contain at least one item **do**
    let r be the minimal item in $\mathcal{I}_R$ and s minimal item in $\mathcal{I}_S$
    if r and s can be joined **then**
        join r and s and put the result to $\mathcal{O}$
        if $\mathcal{O}$ is full **then** write $\mathcal{O}$ to $\mathcal{T}$ and empty $\mathcal{O}$
        remove both r from $\mathcal{I}_R$ and s from $\mathcal{I}_S$
    else remove the lower one of r from $\mathcal{I}_R$ or s from $\mathcal{I}_S$
    if $\mathcal{I}_R$ is empty **then** read the next block from $\mathcal{R}'$ (if any)
    if $\mathcal{I}_S$ is empty **then** read the next block from $\mathcal{S}'$ (if any)
if $\mathcal{O}$ is not empty **then** write $\mathcal{O}$ to $\mathcal{T}$ and empty $\mathcal{O}$

Observations

Join phase

- Memory layout: $1 + 1 + 1$
  - **Input buffer** $\mathcal{I}_R$: 1 page
  - **Input buffer** $\mathcal{I}_S$: 1 page
  - **Output buffer** $\mathcal{O}$: 1 page



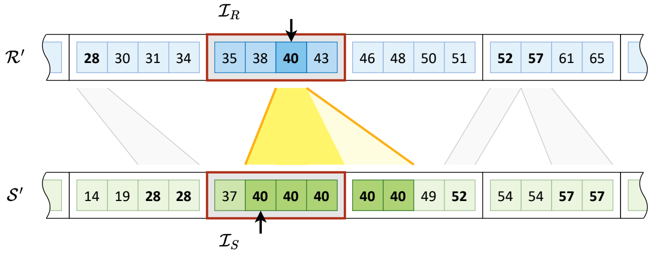
Time complexity

- Sort phase
- **Join phase**
  - $c_{\mathtt{MJ}} = p_R + p_S$

Extended Version

Duplicate items

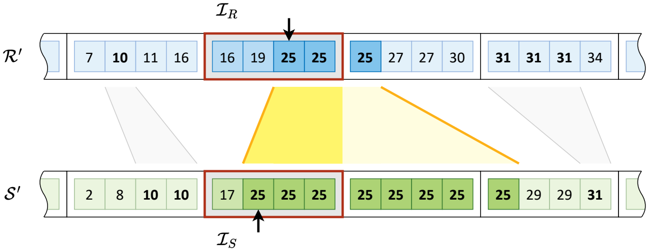
- Possible duplicates in **one table** only
  - Let it be $\mathcal{S}$ (without loss of generality)
  - Algorithm modification is straightforward…
    - Having successfully joined r and s, we just remove s from $\mathcal{I}_S$ and not r from $\mathcal{I}_R$ (line 7)



Extended Version

Duplicate items

- Possible duplicates in **both tables**
  - All matching pairs of r and s just need to be joined…
  - Unfortunately, **size of input buffers** might not be sufficient
    - Since we may span block boundaries, even repeatedly



Hash Join

Hash Join

Hash join approaches

- Basic principle
  - Items of the first table are hashed into the system memory
  - Items of the second table are then attempted to be joined
- Limitations
  - Inner joins based on **value equality tests** only
    - Including possible duplicates
  - Not suitable for small **active domains**
- Particular approaches
  - Classic hash join, **Simple** hash join, Partition hash join, **Grace** hash join, and **Hybrid** hash join

Classic Hashing

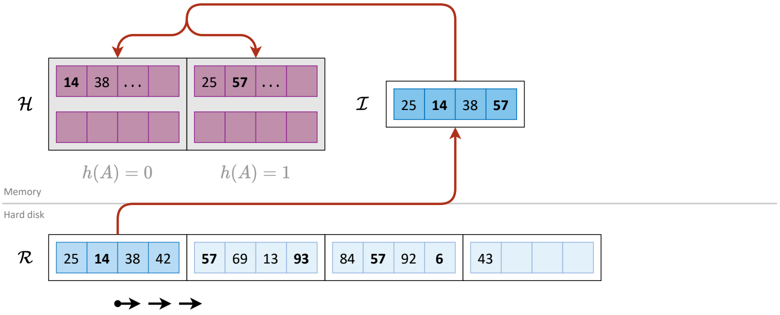
Classic hash join

- **Build phase**
  - **Smaller table** (let it be $\mathcal{R}$) is **hashed into the system memory**
    - I.e., it is sequentially loaded into the memory, block by block
    - All its tuples are then emplaced into the **hash container**
- **Hash function** h is assumed for this purpose
  - Its **domain** are values of the joining attribute A
  - Its **range** provides H distinct values
- **Hash container** internally contains H **buckets**
  - Its **overall size** will inevitably be somewhat larger than p_R
    - Let us say $M = \lceil F \cdot p_R \rceil$ pages for some small factor F
- **Probe phase**
  - Items from the **larger table** $\mathcal{S}$ are **attempted to be joined**

Build Phase

Build phase

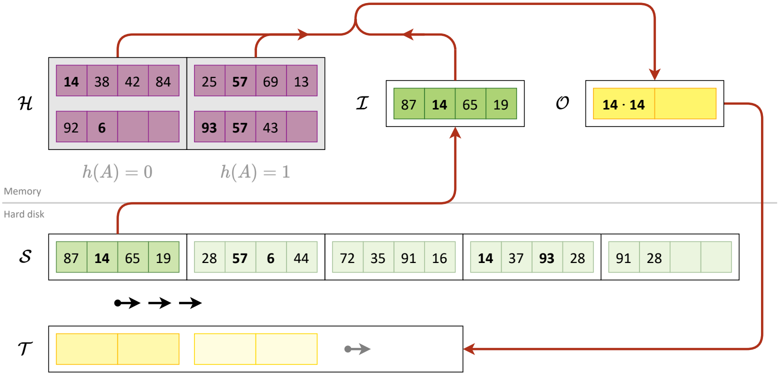
- Tuples from the smaller table are **hashed** into the memory
  - E.g., hash function $h(A) = A \bmod 2$ is assumed



Probe Phase

Probe phase

- Tuples from the larger table are attempted to be **joined**



Algorithm

Build phase

```
foreach block R from 𝓡 do
    read R into 𝓘
    foreach item r in 𝓘 do
        calculate hash value h ← h(r.A)
        add r into bucket h in 𝓗
```

Algorithm

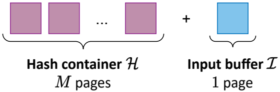
Probe phase

```
foreach block S from 𝒮 do
    read S into ℐ
    foreach item s in ℐ do
        calculate hash value h ← h(s.A)
        foreach item r in bucket h in ℋ do
            if r and s can be joined then
                join r and s and put the result to 𝒪
                if 𝒪 is full then  write 𝒪 to 𝒯 and empty 𝒪
if 𝒪 is not empty then  write 𝒪 to 𝒯 and empty 𝒪
```

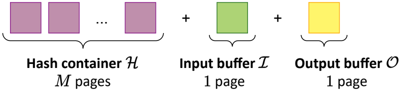
Observations

Memory layout

- Build phase: $M + 1$
  - **Hash container $\mathcal{H}$**: $M = \lceil F \cdot p_R \rceil$ pages
  - **Input buffer $\mathcal{I}$**: 1 page



- Probe phase: $M + 1 + 1$
  - **Hash container $\mathcal{H}$**: M pages (preserved from the build phase)
  - **Input buffer $\mathcal{I}$**: 1 page
  - **Output buffer $\mathcal{O}$**: 1 page



Observations

Time complexity

- Build and probe phases
  - $c_{\texttt{build}} = p_R$
  - $c_{\texttt{probe}} = p_S$
- **Overall cost**
  - $c_{\texttt{CH}} = c_{\texttt{build}} + c_{\texttt{probe}} = p_R + p_S$

Summary

- Interesting approach as for its efficiency
  - However, usable only when the smaller table can entirely be hashed into the system memory…

Partition Hashing

Partition hash join

- Basic principle
  - **Both tables are first partitioned**
    - Using partition function p
  - **Pairs of the corresponding partitions** are then **joined** together
    - Using the classic hash join approach
    - Or actually even nested loops if desired

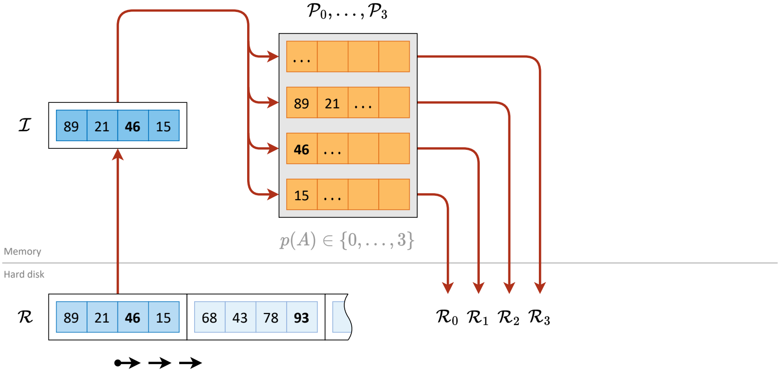
Overall procedure

split $\mathcal{R}$ and create partitions $\mathcal{R}_0, \dots, \mathcal{R}_P$
split $\mathcal{S}$ and create partitions $\mathcal{S}_0, \dots, \mathcal{S}_P$
foreach partition $p \in \{0, \dots, P-1\}$ **do**
    join partitions $\mathcal{R}_p$ and $\mathcal{S}_p$

Partition Phase

Partition phase (for table $\mathcal{R}$)

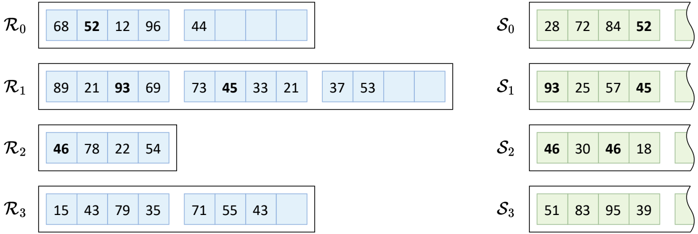
- Tuples of a given table are split to **disjoint partitions**



Join Phase

Partition phase

- **Resulting partitions** for our sample scenario



Join phase

- Pairs of the corresponding partitions are then joined together
  - $\mathcal{R}_0$ and $\mathcal{S}_0$, $\mathcal{R}_1$ and $\mathcal{S}_1$, …

Algorithm

Partition phase

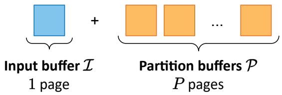
- Table $\mathcal{R}$ is assumed, partitioning of $\mathcal{S}$ is analogous

```
foreach block R from 𝓡 do
    read R into 𝓘
    foreach item r in 𝓘 do
        calculate partition value p ← p(r.A)
        add r into partition buffer 𝓟_p
        if 𝓟_p is full then  write 𝓟_p to 𝓡_p and empty 𝓟_p
foreach partition p ∈ {0, ..., P − 1} do
    if 𝓟_p is not empty then  write 𝓟_p to 𝓡_p and empty 𝓟_p
```

Observations

Memory layout

- Partition phase: $1 + P$
  - **Input buffer** $\mathcal{I}$: 1 page
  - **Partition buffers** $\mathcal{P}$: P pages



Time complexity

- **Partitioning phase**
  - $c_{\texttt{split}} \approx 2 \cdot p_R + 2 \cdot p_S$
- **Overall cost** (with classic hash join involved)
  - $c_{\texttt{PH}} = c_{\texttt{split}} + P \cdot c_{\texttt{CH}} \approx c_{\texttt{split}} + P\left[\frac{p_R}{P} + \frac{p_S}{P}\right] \approx 3 \cdot (p_R + p_S)$

Query Evaluation

Sample Query

Database schema

- **Movie** (<u>id</u>, title, year, …)
- **Actor** (<u>movie, actor</u>, character, …)
  - FK: Actor[movie] $\subseteq$ Movie[id]

Sample query

- **Actors and characters they played in movies filmed in 2000**
  - SQL expression

```
SELECT title, actor, character
FROM Movie JOIN Actor
WHERE (year = 2000) AND (id = movie)
```

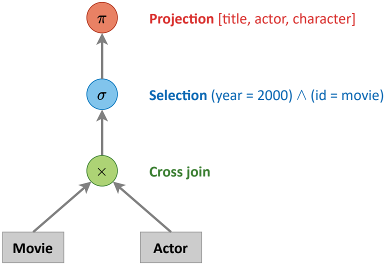
  - RA expression

$$\pi_{\text{title,actor,character}}\left(\varphi_{(\text{year}=2000)\wedge(\text{id}=\text{movie})}\left(\text{Movie} \times \text{Actor}\right)\right)$$

Sample Query

Sample query (cont'd)

- **Actors and characters they played in movies filmed in 2000**
  - $\pi_{\text{title,actor,character}}\left(\varphi_{(\text{year}=2000)\wedge(\text{id}=\text{movie})}\left(\text{Movie} \times \text{Actor}\right)\right)$



Query Evaluation

Basic idea

- SQL query $\rightarrow$ RA query $\rightarrow$ evaluation plan $\rightarrow$ query result

Evaluation process

- (1) **Scanning** [scanner]
  - **Lexical analysis** is performed over the input SQL expression
    - Lexemes are recognized and then tokens generated
- (2) **Parsing** [parser]
  - **Syntactic analysis** is performed
    - Derivation tree is constructed according to the SQL grammar
- (3) **Translation**
  - **Query tree** with **relational algebra operations** is constructed

Query Evaluation

Evaluation process (cont'd)

- (4) **Validation** [validator]
  - **Semantic validity** is checked
    - Compliance of relation schemas with intended operations
- (5) **Optimization** [optimizer]
  - **Alternative evaluation plans** are devised and compared
    - In order to find the most efficient plan
    - Based on their evaluation **cost estimates**
- (6) **Code generation** [generator]
  - Execution code is generated for the chosen plan
- (7) **Execution** [processor]
  - Intended query is finally evaluated
    - And the yielded result provided to the user

Query Evaluation

Query tree

- Internal tree structure
  - **Leaf nodes** = input **tables**
  - **Inner nodes** = individual RA **operations** (σ, π, $\times$, $\bowtie$, …)
- Root node represents the entire query
  - Nodes are evaluated from leaves toward the root

Query evaluation plan

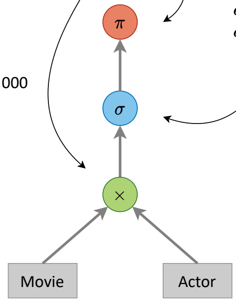
- **Query tree**
- For each inner node…
  - Calculated **statistics** (number of tuples, blocking factor, …)
  - Selected **algorithm** (limited by **context** and available **memory**)
  - Estimated **cost**
- **Overall cost**

Sample Plan #1

Projection [title, actor, character]
$n_3 = n_2 = 20\,000$
$b_3 \leftarrow 50$
$p_3 = n_3/b_3 = 400$
$c_3^{\mathtt{r}} = p_2 = 2\,500$
$c_3^{\mathtt{w}} = p_3 = 400$

Cross join
$n_1 = n_M \cdot n_A = 100\,000\,000\,000$
$b_1 = (b_M \cdot b_A)/(b_M + b_A) = 8$
$p_1 = n_1/b_1 = 12\,500\,000\,000$
Nested loops
$M_1 = 25 + 1 + 1 = 27$
$c_1^{\mathtt{r}} = p_M + (p_M/25) \cdot p_A = 10\,010\,000$
$c_1^{\mathtt{w}} = p_1 = 12\,500\,000\,000$

Selection (year = 2000) ∧ (id = movie)
$n_2 = n_1 \cdot (1/V_{M.year}) \cdot (1/n_M) = 20\,000$
$b_2 = b_1 = 8$
$p_2 = n_2/b_2 = 2\,500$
$c_2^{\mathtt{r}} = p_1 = 12\,500\,000\,000$
$c_2^{\mathtt{w}} = p_2 = 2\,500$



Sorted file (year)
$n_M = 100\,000$
$b_M = 10$
$p_M = 10\,000$
$V_{M.year} = 50$
B$^+$ tree index (year)
$m_{M.year} = 100$
$I_{M.year} = 3$

Heap file
$n_A = 1\,000\,000$
$b_A = 40$
$p_A = 25\,000$

Evaluation Plan Cost

Overall **evaluation cost**

- Let us first assume that all **intermediate results are always written to temporary files** and so each involved operation…
  - Reads its inputs from / writes its output to a hard drive
- **Overall cost** then equals to the **sum of all the partial costs**

Cost of **Plan #1**

- $M = 25 + 1 + 1$ memory pages
- $c = [c_1^{\mathtt{r}} + c_1^{\mathtt{w}}] + [c_2^{\mathtt{r}} + c_2^{\mathtt{w}}] + [c_3^{\mathtt{r}}]$
- $c = [p_M + (p_M/25) \cdot p_A + p_1] + [p_1 + p_2] + [p_2]$
- $c = [10\,010\,000 + 12\,500\,000\,000] + [12\,500\,000\,000 + 2\,500] + [2\,500]$
- $c = 25\,010\,015\,000$

Sample Query

Intuitive optimization

- **Actors and characters they played in movies filmed in 2000**
  - SQL expression

```
SELECT title, actor, character
FROM Movie JOIN Actor ON (id = movie)
WHERE (year = 2000)
```

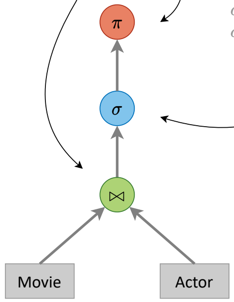
  - RA expression

$$\pi_{\text{title,actor,character}}\left(\varphi_{(\text{year}=2000)}\left(\text{Movie} \bowtie_{(\text{id}=\text{movie})} \text{Actor}\right)\right)$$

Sample Plan #2

Projection [title, actor, character]
$n_3 = n_2 = 20\,000$
$b_3 \leftarrow 50$
$p_3 = n_3 / b_3 = 400$
$c_3^{\mathtt{r}} = p_2 = 2\,500$
$c_3^{\mathtt{w}} = p_3 = 400$

Theta join [id = movie]
$n_1 = n_A = 1\,000\,000$
$b_1 = (b_M \cdot b_A)/(b_M + b_A) = 8$
$p_1 = n_1 / b_1 = 125\,000$
Nested loops
$M_1 = 25 + 1 + 1 = 27$
$c_1^{\mathtt{r}} = p_M + (p_M/25) \cdot p_A = 10\,010\,000$
$c_1^{\mathtt{w}} = p_1 = 125\,000$



Selection (year = 2000)
$n_2 = n_1 \cdot (1/V_{M.year}) = 20\,000$
$b_2 = b_1 = 8$
$p_2 = n_2 / b_2 = 2\,500$
$c_2^{\mathtt{r}} = p_1 = 125\,000$
$c_2^{\mathtt{w}} = p_2 = 2\,500$

Sorted file (year)
$n_M = 100\,000$
$b_M = 10$
$p_M = 10\,000$
$V_{M.year} = 50$
B$^+$ tree index (year)
$m_{M.year} = 100$
$I_{M.year} = 3$

Heap file
$n_A = 1\,000\,000$
$b_A = 40$
$p_A = 25\,000$

Sample Plan #2

Cost of **Plan #2**

- Again $M = 25 + 1 + 1$ memory pages
- $c = [c_1^{\mathtt{r}} + c_1^{\mathtt{w}}] + [c_2^{\mathtt{r}} + c_2^{\mathtt{w}}] + [c_3^{\mathtt{r}}]$
- $c = [p_M + (p_M/25) \cdot p_A + p_1] + [p_1 + p_2] + [p_2]$
- $c = [10\,010\,000 + 125\,000] + [125\,000 + 2\,500] + [2\,500]$
- $c = 10\,265\,000$
  - That is approximately $2\,400$ times better than the first plan

Pipelining

Pipelining mechanism

- **Intermediate results are passed** between the operations **directly without** the usage of **temporary files** on a disk
  - And so just within the system memory
    - It may even be possible to do it **in-place** without extra pages
- Unfortunately, such an approach is not always possible…

Cost of **Plan #2 with pipelining**

- Still $M = 25 + 1 + 1$ memory pages
- $c = [c_1^{\mathtt{r}} + \cancel{c_1^{\mathtt{w}}}] + [\cancel{c_2^{\mathtt{r}}} + \cancel{c_2^{\mathtt{w}}}] + [\cancel{c_3^{\mathtt{r}}}]$
  - Joined tuples are filtered and projected immediately in-place
- $c = 10\,010\,000$

Query Optimization

Objective = finding the most **optimal query evaluation plan**

- It is not possible to consider all plans, though
  - Simply because there are far too many of them
  - And so **pruning and heuristics** need to be incorporated

Optimization strategies

- **Algebraic**
  - Proposal of **alternative plans** using query tree transformations
- **Statistical**
  - Estimation of **costs and result sizes** based on available statistics
- **Syntactic**
  - Manual modification of query expressions by users themselves
    - In order to involve plans that would otherwise be unreachable
    - Breaches the principle of **declarative querying**, though

Statistical Optimization

Objective

- Capability of **calculating necessary result characteristics**…
  - Of both the final result as well as all intermediate ones
    - I.e., all individual nodes within a given evaluation plan tree
- … so that the **overall cost** can be estimated
  - And thus alternative plans mutually compared

Basic statistics

- **Data file** for table $\mathcal{R}$
  - n_R number of tuples, s_R tuple size, b_R blocking factor
  - p_R number of pages
  - Hashed file: H_R number of buckets, C_R bucket size
- **Index file** for attribute A from table $\mathcal{R}$
  - B$^+$ tree: $I_{R.A}$ tree height, $p_{R.A}$ number of leaf nodes

Statistical Optimization

Additional statistics

- Provide deeper insight into the **active domain**
  - May even be implicitly **derivable** from index structures
  - Unfortunately, they may also be **missing or unavailable**
    - Especially as for intermediate results
- $V_{R.A}$ number of distinct values
- $min_{R.A}$ and $max_{R.A}$ minimal and maximal values
- **Histograms**
  - Provide even more accurate understanding of the domain
    - And so better estimates
  - Especially useful for **non-uniform distributions**

Size Estimates: Selection

Selection: $T = \sigma_\varphi(E)$

Tuple size

- $s_T = s_E$
  - Tuples are just filtered out and so their size remains untouched

Blocking factor

- $b_T = b_E$

Number of tuples

- Basic idea: $n_T = \lceil n_E \cdot r_\varphi \rceil$
- $r_\varphi \in [0, 1]$ is an estimated **reduction factor**
  - Describes how much the original tuples will be reduced
    - Depends on a particular condition φ
    - As well as particular available statistics…

Size Estimates: Projection

Projection: $T = \pi_{a_1,\dots,a_n}(E)$

Tuple size

- s_T is simply calculated using sizes of all preserved attributes

Blocking factor

- $b_T = \lfloor B/s_T \rfloor$

Number of tuples

- Default SQL projection without the `DISTINCT` modifier
  - I.e., removal of potential duplicates is not performed
  - $n_T = n_E$
- With duplicates removal enabled
  - $n_T = n_E$ if at least one key of E is preserved
  - …

Size Estimates: Joins

Inner joins: $T = E_R \times E_S$ or $E_R \bowtie E_S$ or $E_R \bowtie_\varphi E_S$

Tuple size

- $s_T \approx s_R + s_S$
  - Less for natural join since shared attributes are not repeated

Blocking factor

- $b_T \approx \left\lfloor \frac{B}{s_T} \right\rfloor \approx \left\lfloor \frac{B}{s_R + s_S} \right\rfloor \approx \left\lfloor \frac{B}{B/b_R + B/b_S} \right\rfloor \approx \left\lfloor \frac{b_R \cdot b_S}{b_R + b_S} \right\rfloor$
  - Can be calculated exactly from the actual resulting tuple size
  - As well as **estimated just using the original blocking factors**

Number of tuples

- $n_T = \lceil n_R \cdot n_S \cdot r_\varphi \rceil$ with $r_\varphi \in [0, 1]$ for **joining condition** φ
  - Similar approach with **reduction factors** as in selections

Algebraic Optimization

Objective

- Capability of **finding alternative query evaluation plans**
  - Based on various **equivalence rules**
    - E.g.: commutativity of selection, associativity of inner joins, …
- Ultimate challenge
  - **Space of all possible plans** may be enormous
  - And so **significant pruning** must be involved

Basic strategy for **SPJ queries** = **select-project-join** queries

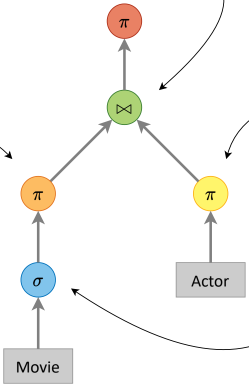
- They allow to be approached at two separate levels…
  - Single-relation plans = best **access method** for each table
  - Multi-relation plans = best **join plan** for all the tables
- But still an **NP-complete problem**

Examples

Sample transformations

- $\pi_{\text{title,actor,character}}\left(\boxed{\varphi_{(\text{year}=2000)\wedge(\text{id}=\text{movie})}}(\text{Movie} \times \text{Actor})\right)$ // #1
- $\pi_{\text{title,actor,character}}\left(\boxed{\varphi_{(\text{id}=\text{movie})}}\left(\boxed{\varphi_{(\text{year}=2000)}}(\text{Movie} \times \text{Actor})\right)\right)$
- $\pi_{\text{title,actor,character}}\left(\varphi_{(\text{year}=2000)}\left(\boxed{\varphi_{(\text{id}=\text{movie})}}(\text{Movie} \boxed{\times} \text{Actor})\right)\right)$
- $\pi_{\text{title,actor,character}}\left(\boxed{\varphi_{(\text{year}=2000)}}\left(\text{Movie} \bowtie_{(\text{id}=\text{movie})} \text{Actor}\right)\right)$ // #2
- $\boxed{\pi_{\text{title,actor,character}}}\left(\varphi_{(\text{year}=2000)}(\text{Movie}) \bowtie_{(\text{id}=\text{movie})} \text{Actor}\right)$
- $\pi_{\text{title,actor,character}}\left(\pi_{\text{id,title}}\left(\varphi_{(\text{year}=2000)}(\text{Movie})\right) \bowtie_{(\text{id}=\text{movie})} \pi_{\text{movie,actor,character}}(\text{Actor})\right)$ // #3

Sample Plan #3



Projection [title, actor, character]
$n_5 = n_4 = 20\,000$
$b_5 \leftarrow 50$
$p_5 = n_5/b_5 = 400$
$c_5^{\mathtt{w}} = p_5 = 400$

Theta join [id = movie]
$n_4 = n_3 \cdot (1/V_{M.year}) = 20\,000$
$b_4 = (b_2 \cdot b_3)/(b_2 + b_3) \approx 35$
Nested loops
$M_4 = p_2 + 1 + 1 = 27$

Projection [id, title]
$n_2 = n_1 = 2\,000$
$b_2 \leftarrow 80$
$p_2 = n_2/b_2 = 25$

Projection [movie, actor, character]
$n_3 = n_A = 1\,000\,000$
$b_3 \leftarrow 65$
$c_3^{\mathtt{r}} = p_A = 25\,000$

Heap file
$n_A = 1\,000\,000$
$b_A = 40$
$p_A = 25\,000$

Sorted file (year)
$n_M = 100\,000$
$b_M = 10$
$p_M = 10\,000$
$V_{M.year} = 50$
B$^+$ tree index (year)
$m_{M.year} = 100$
$I_{M.year} = 3$

Selection (year = 2000)
$n_1 = n_M \cdot (1/V_{M.year}) = 2\,000$
$b_1 = b_M = 10$
$p_1 = n_1/b_1 = 200$
$c_1^{\mathtt{r}} = I_{M.year} + p_M \cdot (1/V_{M.year}) = 203$

Sample Plan #3

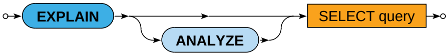
Cost of **Plan #3 with pipelining**

- $M = 25 + 1 + 1$ memory pages for buffers $\mathcal{I}_1$, $\mathcal{I}_2$ and $\mathcal{O}$
  - I.e., still the same amount of system memory pages used
- $c = [c_1^{\mathtt{r}} + \cancel{c_1^{\mathtt{w}}}] + [\cancel{c_2^{\mathtt{r}}} + \cancel{c_2^{\mathtt{w}}}] + [c_3^{\mathtt{r}} + \cancel{c_3^{\mathtt{w}}}] + [\cancel{c_4^{\mathtt{r}}} + \cancel{c_4^{\mathtt{w}}}] + [\cancel{c_5^{\mathtt{r}}}]$
  - $\mathcal{I}_2$ is used for index traversal and then reading of movies
  - All filtered and projected movies are put into $\mathcal{I}_1$
  - Actors are read into $\mathcal{I}_2$, their projection is postponed
  - Joined tuples are put into $\mathcal{O}$ and projected
- $c = [I_{M.year} + p_M \cdot (1/V_{M.year})] + [p_A]$
- $c = [203] + [25\,000]$
- $c = 25\,203$
  - That is approximately 400 times better than the second plan
    - And so almost 1 million times better than the first plan

Explain Statements

`EXPLAIN` **statement**

- Allows to **retrieve the evaluation plan** for a given query
  - When `ANALYZE` modifier is provided…
    - Query is also executed and the actual run times are returned



Example

- ```
  EXPLAIN
    SELECT title, actor, character
    FROM Movie JOIN Actor
    WHERE (year = 2000) AND (id = movie)
  ```
  ```

Observations

False assumptions and simplifications

- **Variable size** of tuples
- **Unused slots** and **inner fragmentation** within blocks
- **Overflow areas** in sorted / hashed files
- **Outer fragmentation** of files on a hard drive
- Impact of the **caching manager**
- Extent of **available statistics** and their **lazy maintenance**
- **Non-uniform distribution** of data / queries
- **Independence of conditions** in reduction factors
- …

Conclusion

Evaluation **algorithms**

- Access methods
- Sorting
  - **External merge sort** algorithm
- Joining
  - Binary **nested loops join** with / without zig-zag
  - **Sort-merge join**
  - Classic / partition **hash join**

Query **evaluation** and **optimization**

- Evaluation **plans**
  - Cost estimates, **pipelining**
- Statistical / algebraic optimization