B0B36DBS: **Database Systems**

http://www.ksi.mff.cuni.cz/~svoboda/courses/202-B0B36DBS/

Lecture 9

Query Evaluation

Martin Svoboda
martin.svoboda@fel.cvut.cz

13. 4. 2021

Czech Technical University in Prague, Faculty of Electrical Engineering

Lecture Outline

Evaluation algorithms

- **External sort**
- **Nested loops join**
- **Sort-merge join**

Query evaluation

- **Query evaluation plans**
- Optimization techniques

Introduction

Evaluation of **SQL queries**

- `SELECT` statements
  - `SELECT … FROM … WHERE … ORDER BY …`

Transformation to (extended) **relational algebra**

- Selection, projection, attribute renaming
- Set operations: union, intersection, difference
- Inner joins: Cartesian product, natural join, theta join
- Left / right natural / theta semijoin and antijoin
- Left / right / full outer natural / theta join
- Division
- Sorting, grouping and aggregation, distinct, …

Naive Algorithms

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

- Iteration over all items and removal of those not satisfying a given filtering condition

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

- Iteration over all items and removal of excluded attributes
- Removal of duplicates within the traditional relational model

Distinct

- Sorting of all items 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

- **Atomic read / write operations** at the level of blocks only
- I.e., individual items cannot be accessed directly
  - Primary files, index structures, system catalogs

Latency

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

Memory

- Size of **available system memory** will always be limited

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

Objective

Query evaluation plan

- Based on the **context** and available system **memory**, suitable evaluation **algorithm** needs to be selected for every involved **operation** so that the overall **cost** is minimal

Knowledge of context

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

Available Statistics

Environment

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

Relation R and its attribute A

- n_R: **number of tuples** in R
- s_R: (average) 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 A
  - Number of distinct values of A occurring in R

Available Statistics

B$^+$-tree index for relation R and its attribute A

- $f_{R.A}$: (average) **number of followers** in an inner node
  - Usually ≈ 100
- $I_{R.A}$: **index depth**
  - Usually $\approx 2 - 3$
- $p_{R.A}$: number of **leaf nodes**

External Sort

External Sort

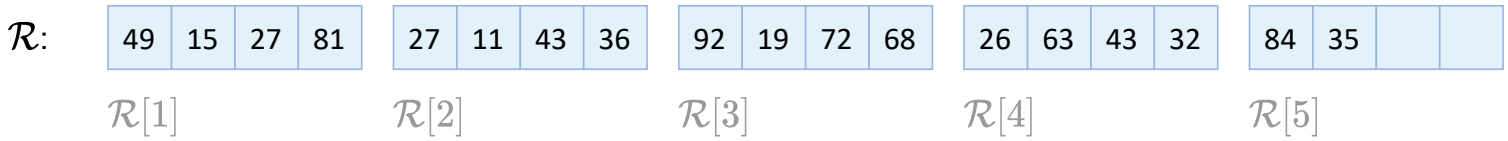
N-way external merge sort algorithm

- **Sorting phase** (**pass 1**)
  - Groups of input blocks are loaded into the system memory
  - Items in these blocks are sorted
    - Any **in-memory in-place sorting algorithm** can be used
    - E.g.: *quick sort*, *heap sort*, *bubble sort*, *insertion sort*, …
  - Created **runs** are written into a temporary file
- **Merging 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 just one run is obtained

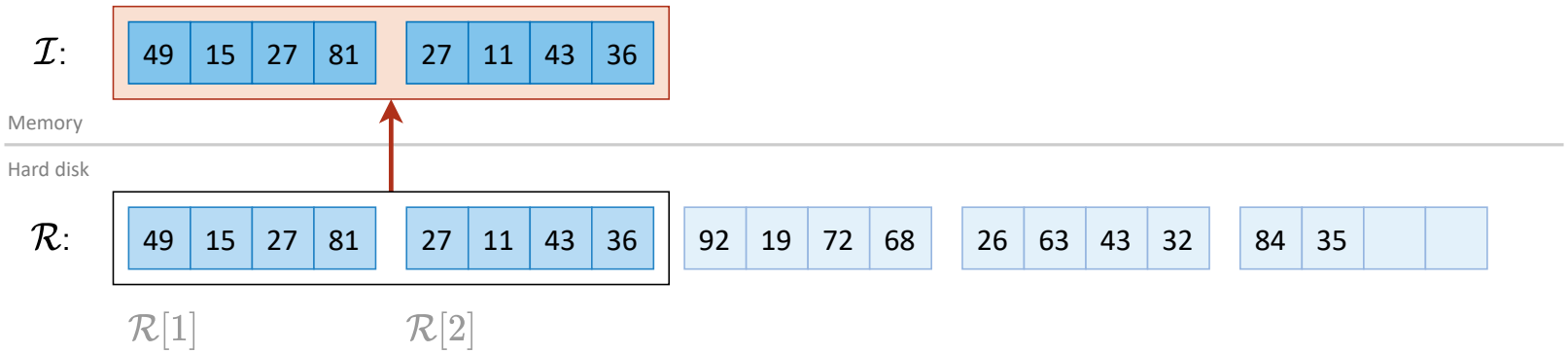
Sorting Phase

Pass 1

- Input file



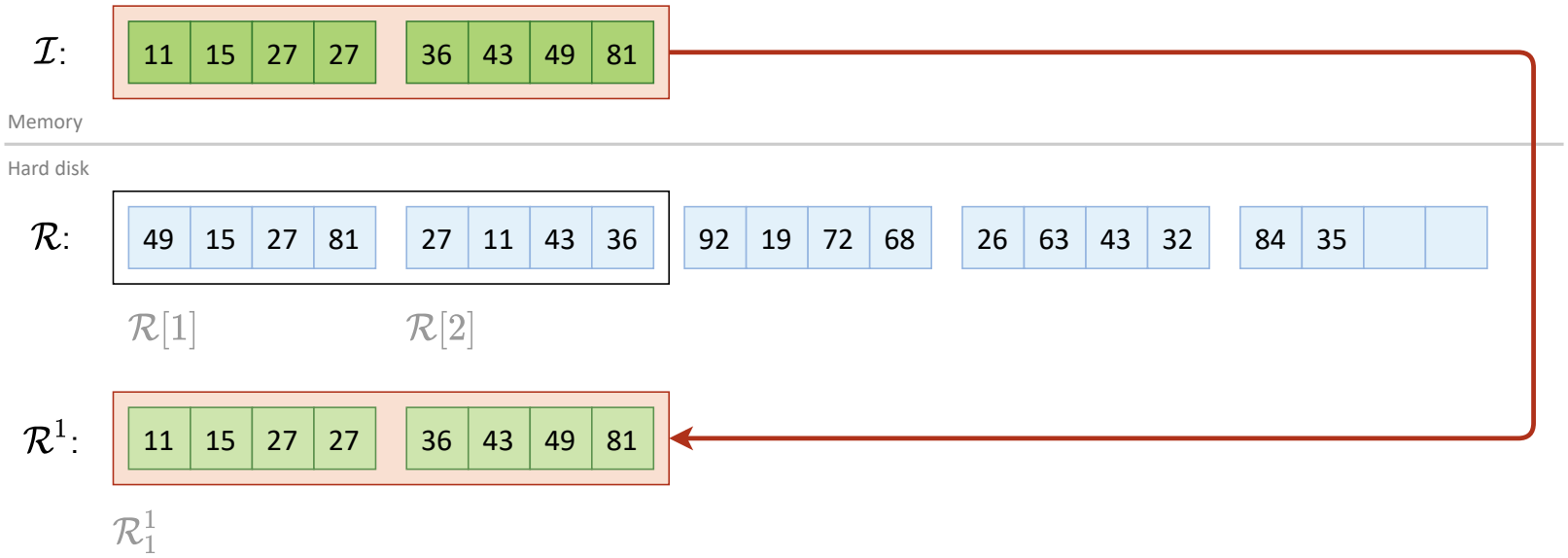
- Input buffer $\mathcal{I}$ in the system memory
  - Size, e.g., $M = 2$ blocks
- The first M blocks are loaded from $\mathcal{R}$ into $\mathcal{I}$



Sorting Phase

Pass 1 (cont'd)

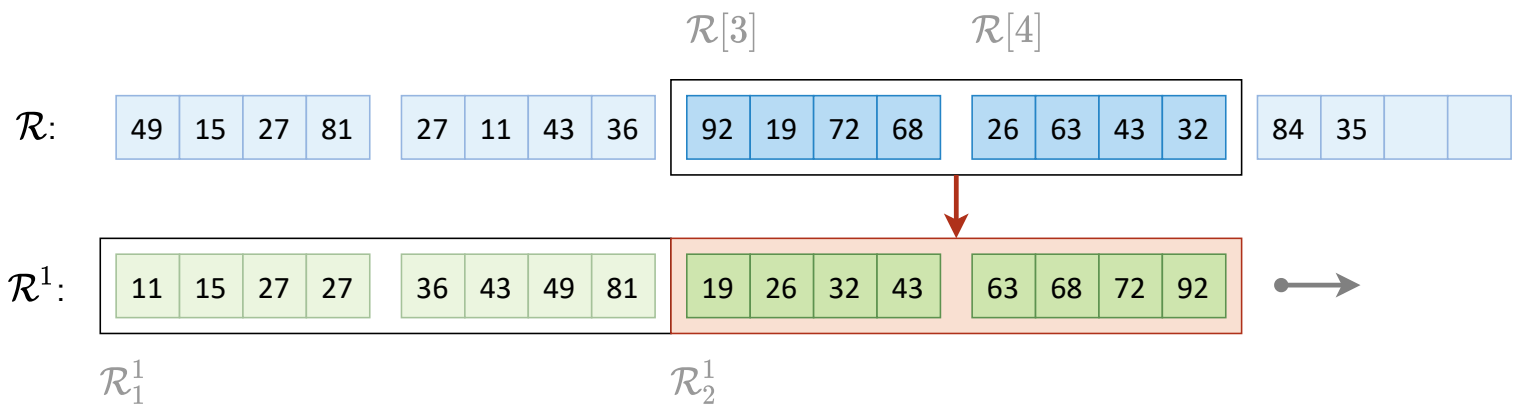
- Items in $\mathcal{I}$ are sorted and a yielded run is written into $\mathcal{R}^1$



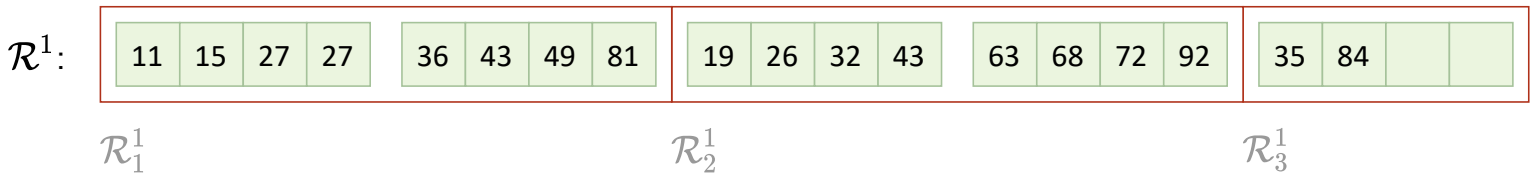
Sorting Phase

Pass 1 (cont'd)

- Next M blocks are processed and another run is produced



- All the remaining input blocks from $\mathcal{R}$ are processed as well

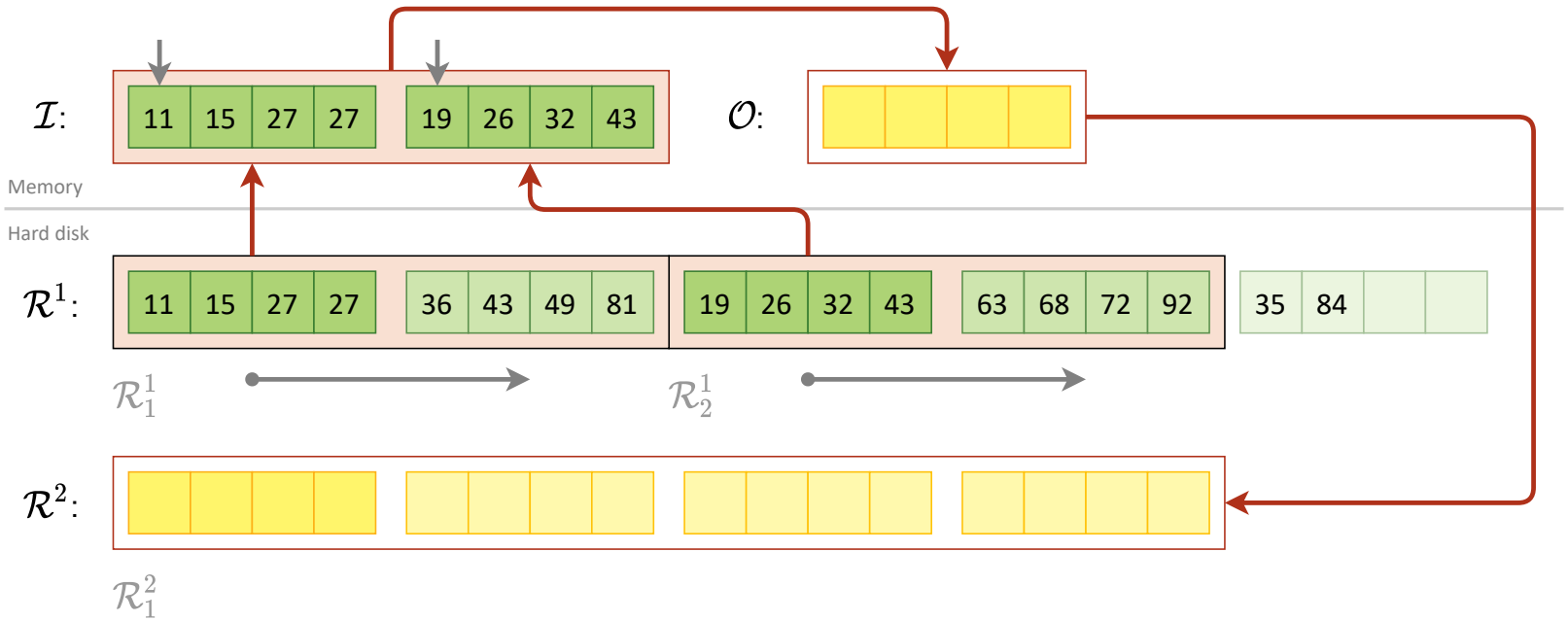


- Sorting phase is completed and $\mathcal{R}^1$ finalized

Merging Phase

Pass 2

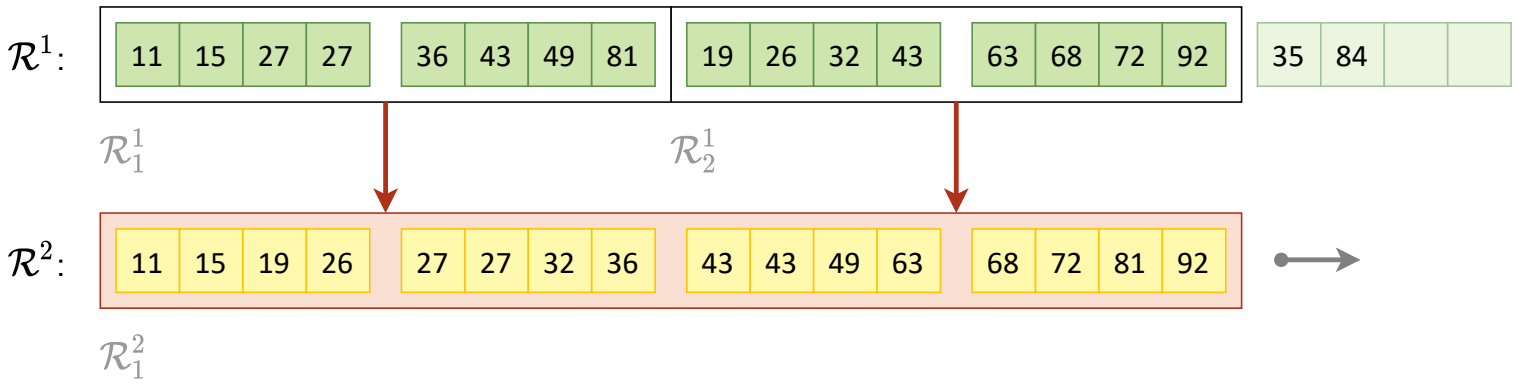
- The first M **runs are merged together** to a new longer run
  - Blocks from these input runs are gradually loaded into $\mathcal{I}$
  - Minimal items are iteratively searched for and moved to $\mathcal{O}$
  - Output buffer $\mathcal{O}$ is written into $\mathcal{R}^2$ whenever full



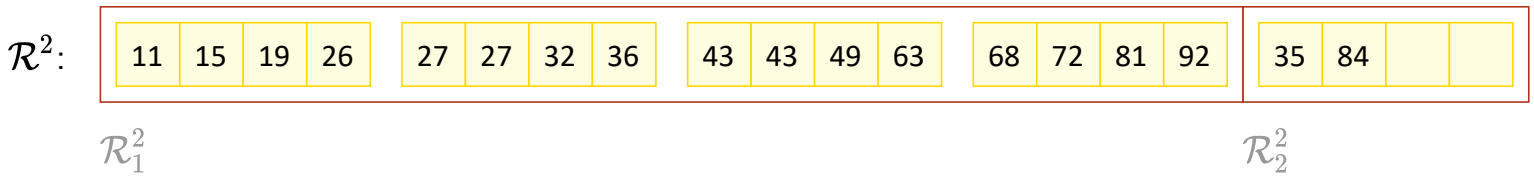
Merging Phase

Pass 2 (cont'd)

- The first M **runs are merged together** to a new longer run



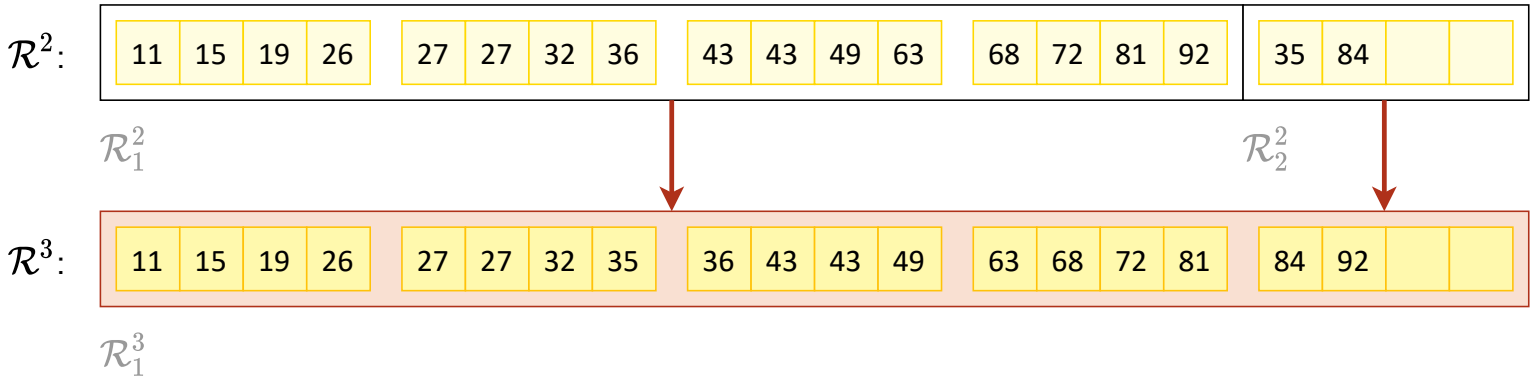
- All the remaining groups of M runs are merged as well



Merging Phase

Pass 3

- Merging continues until just a single run is acquired



Algorithm

Sorting phase (pass 1)

1 $p \leftarrow 1$
2 **foreach** group of blocks $B_1, \ldots, B_M$ (if any) from $\mathcal{R}$ **do**
3 read these blocks to $\mathcal{I}$
4 sort all items in $\mathcal{I}$
5 write all blocks from $\mathcal{I}$ as a new run to a temp file $\mathcal{R}^p$

Algorithm

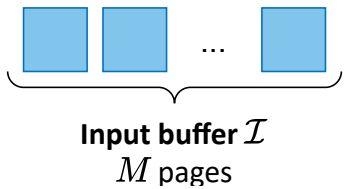
Merging phase (passes 2 and higher)

```
6  while R^p has more then just one run do
7      p ← p + 1
8      foreach group of runs R_1, ..., R_M (if any) from R^{p-1} do
9          start constructing a new run in R^p
10         read the first block from each run R_x to I[x]
11         while I contains at least one item do
12             find the minimal item and move it to O
13             if the corresponding I[x] became empty then
14                 read the next block from R_x (if any) to I[x]
15             if O is full then write O to R^p
16         if O is not empty then write O to R^p
```

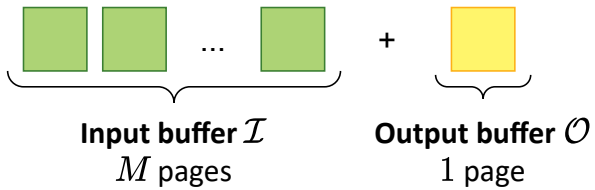
Summary

System memory usage

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



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



Summary

Time complexity

- **Single pass** (both during the sorting and merging phases)
  - $c_{\texttt{read}} = c_{\texttt{write}} = p_R$
- **Number of passes**
  - $t = \lceil \log_M(p_R) \rceil$
- **Overall cost**
  - $c = t \cdot (c_{\texttt{read}} + c_{\texttt{write}}) = \lceil \log_M(p_R) \rceil \cdot 2p_R$

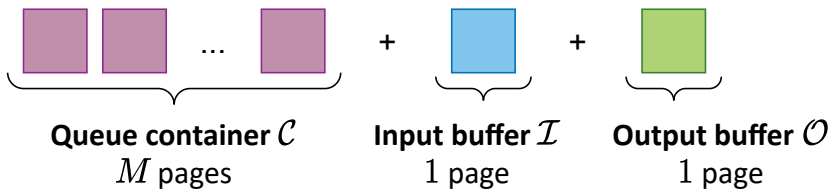
Limitation of the overall **number of passes**

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

Improvements

Priority queue

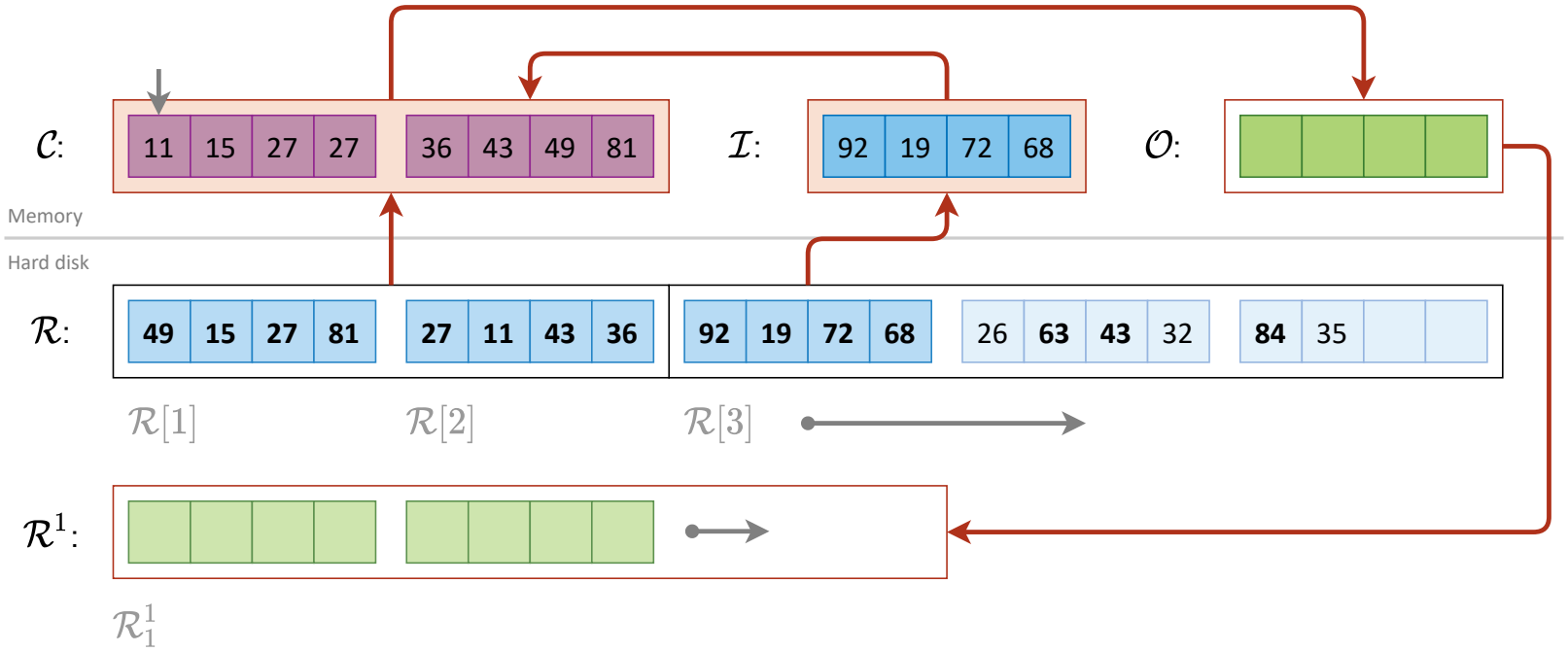
- **Sorting phase** is modified
  - Priority queue is involved
    - In particular, **min-heap** **data structure** is used
  - Minimal items are iteratively found and put to the current run
  - New input items are in turn added to the queue
- System memory usage: $M + 1 + 1$ pages
  - **Queue container** $\mathcal{C}$: $M \geq 1$ pages
  - **Input buffer** $\mathcal{I}$: 1 page
  - **Output buffer** $\mathcal{O}$: 1 page



Improved Sorting Phase

Pass 1

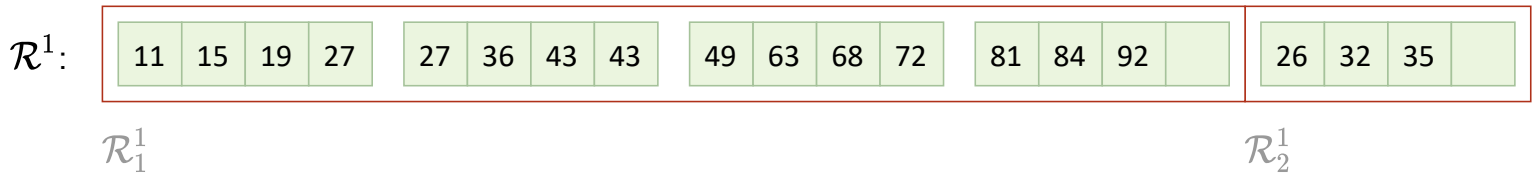
- Priority queue and input buffer are initialized
- Runs are generated on the fly
  - **Minimal item greater than or equal to the last value** is always extracted and replaced with a new item from the input file



Improved Sorting Phase

Pass 1 (cont'd)

- Two runs are obtained in this case



Impact

- Created **runs tend to be longer**
  - $2M$ blocks on average (instead of just M)
    - p_R in the best case
    - M in the worst case
- $\Rightarrow$ **number of runs will tend to be lower**

Algorithm

Improved **sorting phase** (pass 1)

```
1  read blocks R[1], ..., R[M] (if any) from R to C
2  read block R[M+1] (if any) from R to I
3  while C contains at least one item do
4      start constructing a new run in R^1, put v ← −∞
5      while C contains at least one item i ≥ v do
6          let i be the minimal one, move it to O, put v ← i
7          if O is full then write O to R^1
8          move the next item from I (if any) to C
9          if I became empty then
10             read the next block from R (if any) to I
11     if O is not empty then write O to R^1
```

Nested Loops Join

Nested Loops

Binary (block) **nested loops** algorithm

- Universal approach for all kinds of **inner joins**
  - Natural join, cross join, theta join
  - Allows duplicate items
  - Requires no indexes
- Not the best option in all situations
  - Suitable for tables with significantly different sizes

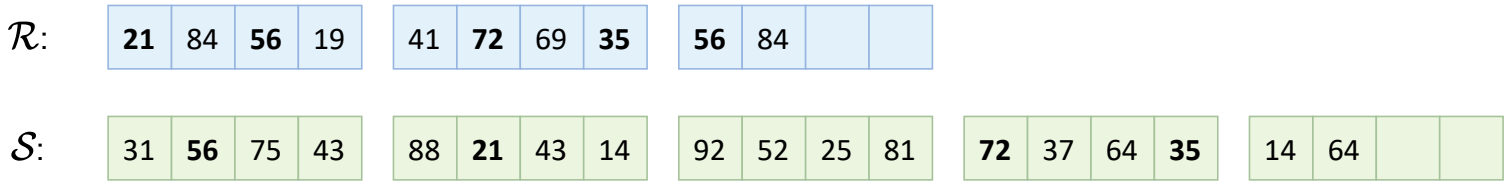
Idea

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

Nested Loops

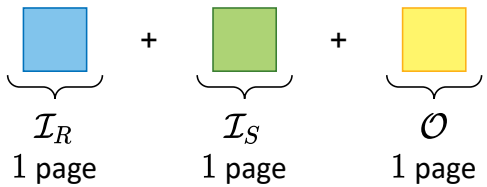
Sample input data

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



Basic configuration

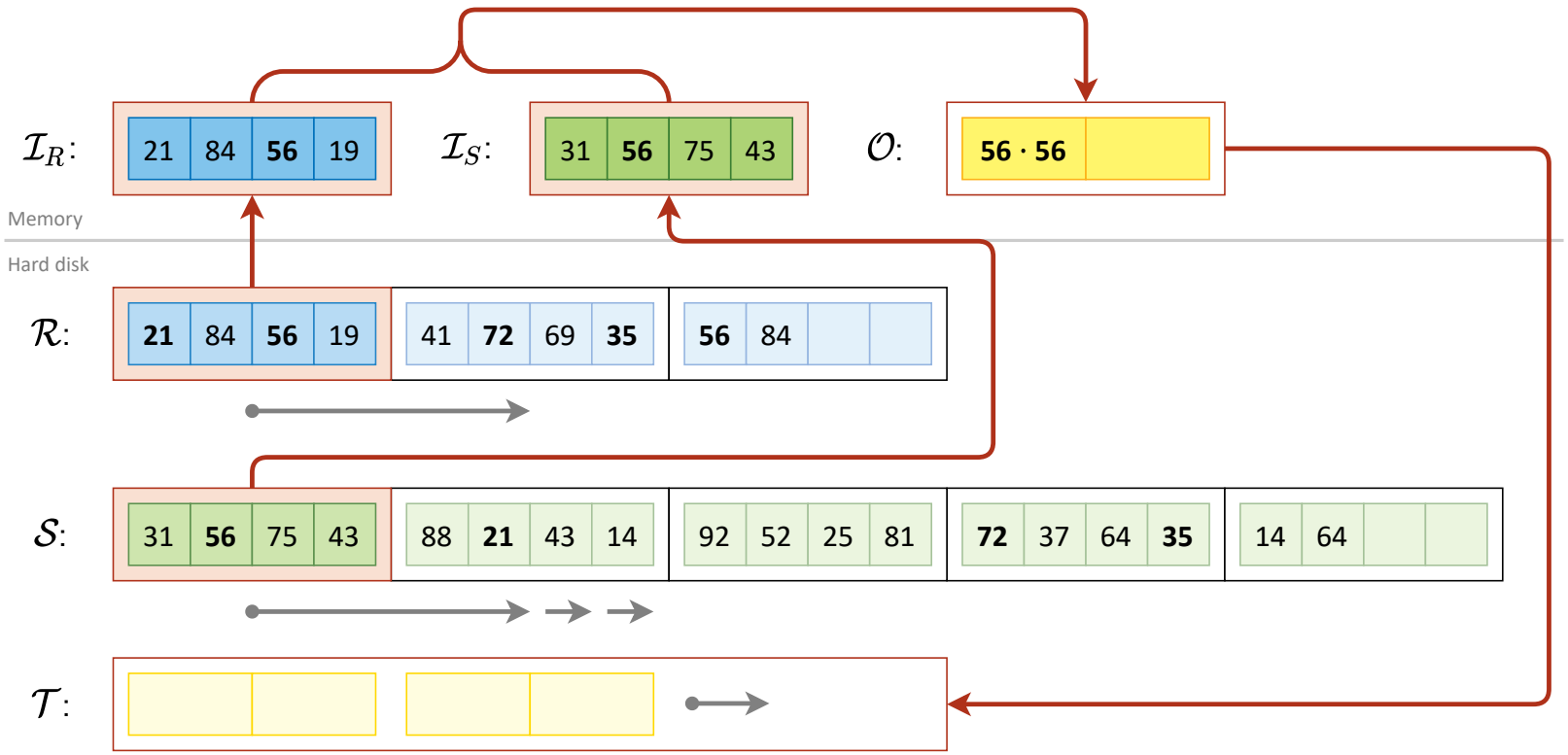
- System memory usage: $1 + 1 + 1$ pages
  - **Input buffer** $\mathcal{I}_R$: 1 page
  - **Input buffer** $\mathcal{I}_S$: 1 page
  - **Output buffer** $\mathcal{O}$: 1 page



Nested Loops

Basic configuration $(1 + 1 + 1)$

- Items already loaded into the system memory are joined using a naive algorithm, joined pairs are put into the output buffer



Algorithm

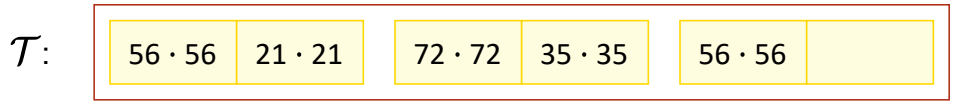
Basic configuration $(1 + 1 + 1)$

```
1  foreach block R from ℛ do
2      read R into ℐ_R
3      foreach block S from 𝒮 do
4          read S into ℐ_S
5          foreach item r in ℐ_R do
6              foreach item s in ℐ_S do
7                  if r and s satisfy the join condition then
8                      join r and s and put the result to 𝒪
9                      if 𝒪 is full then write 𝒪 to 𝒯
10 if 𝒪 is not empty then write 𝒪 to 𝒯
```

Observations

Resulting table

- Table $\mathcal{T}$ with pairs of joined items



Time complexity

- Basic configuration $(1 + 1 + 1)$
  - $c_{\texttt{read}} = p_R + p_R \cdot p_S$
  - $c_{\texttt{write}} = \lceil n_T / b_T \rceil$
    - n_T is the number of joined items
    - b_T is the resulting blocking factor
- $\Rightarrow$ **smaller table** should always be taken as the **outer** one

Improvements

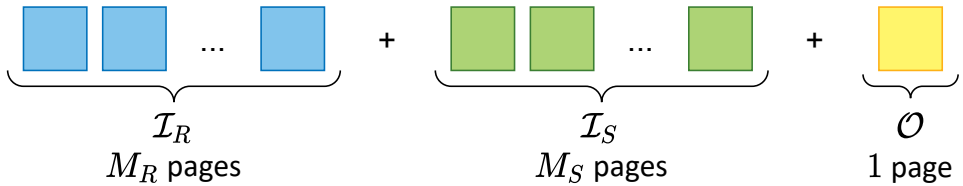
Zig-zag reading of the inner table

- Principle
  - Odd iterations normally
  - **Even iterations in the reverse order**
- Time complexity
  - $c_{\texttt{read}} = p_R + p_R \cdot (p_S - 1) + 1$

Improvements

Generic configuration

- *What if multiple pages were used for both the input buffers?*
- $M_R + M_S + 1$ pages
  - **Input buffer** $\mathcal{I}_R$: M_R pages
  - **Input buffer** $\mathcal{I}_S$: M_S pages
  - **Output buffer** $\mathcal{O}$: 1 page

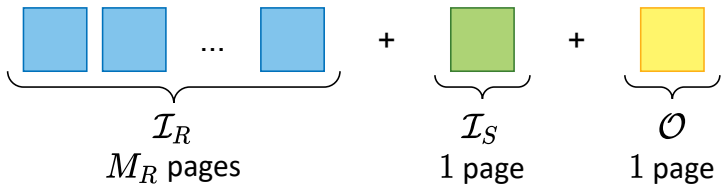


- **Time complexity**
  - $c_{\texttt{read}} = p_R + \lceil p_R / M_R \rceil \cdot p_S$ (without zig-zag)
  - $c_{\texttt{read}} = p_R + \lceil p_R / M_R \rceil \cdot (p_S - M_S) + M_S$ (with zig-zag)
- $\Rightarrow$ there is no practical reason of having $M_S \geq 2$

Improvements

Optimized configuration

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

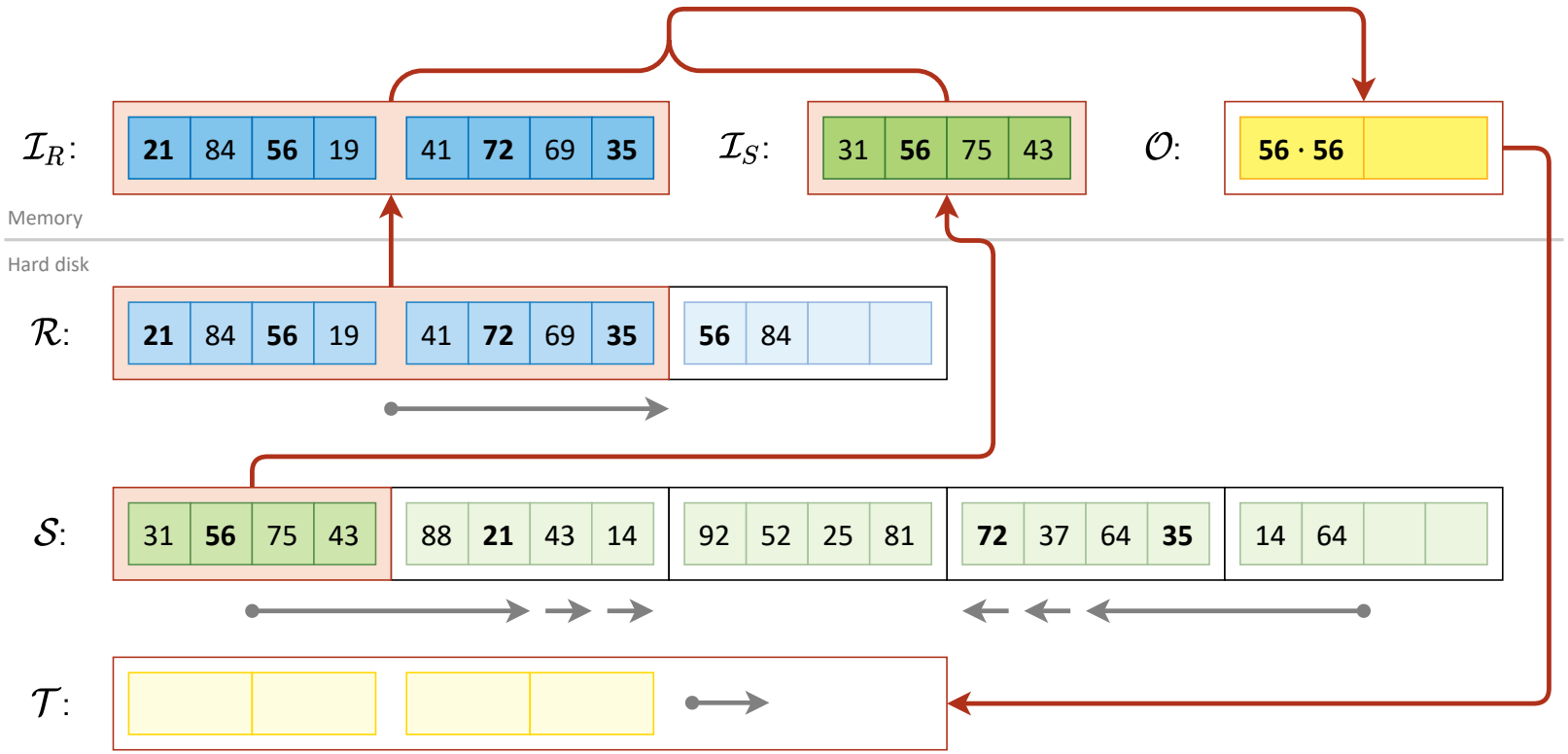


- **Time complexity**
  - $c_{\texttt{read}} = p_R + \lceil p_R / M_R \rceil \cdot p_S$ (without zig-zag)
  - $c_{\texttt{read}} = p_R + \lceil p_R / M_R \rceil \cdot (p_S - 1) + 1$ (with zig-zag)

Improvements

Optimized configuration ($M_R + 1 + 1$)

- Multiple pages are used just for the outer table



Algorithm

Optimized configuration ($M_R + 1 + 1$) with **zig-zag**

```
1 foreach blocks R_1, ..., R_{M_R} from R do
2     read these blocks into I_R
3     foreach block S from S do
4         read S into I_S
5         // In-memory join
6         ...
7 ...
```

Specific Cases

Smaller table fits entirely within the memory

- I.e., $p_R \leq M_R$
  - $c_{\mathtt{read}} = p_R + p_S$

B$^+$-tree index exists in S on the attribute A that is unique in S

- In case **R is organized as a heap**
  - $c_{\mathtt{read}} = p_R + n_R \cdot (I_{S.A} + 1)$
- In case **R is sorted** with respect to A
  - $c_{\mathtt{read}} = p_R + I_{S.A} + p_{S.A} + V_{R.A}$

S is a hashed file over the joining attribute A that is unique in S

- In case **R is sorted** with respect to A
  - $c_{\mathtt{read}} = p_R + V_{R.A} \cdot C_S$

...

Sort-Merge Join

Sort-Merge Join

Sort-merge join algorithm (or just merge join)

- Inner joins based on **value equality tests** only
  - **Without duplicates**
    - Could be extended to support duplicates in one of the relations
- Suitable for tables with relatively similar sizes
  - Especially when they are already sorted
  - Or when the final result is expected to be sorted

Idea

- **Sorting phase**
  - Both tables are externally sorted, one by one (if not yet)
- **Joining phase**
  - Pairs of items are joined directly during the merging process

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**	

Sorting phase

- Resulting sorted tables

$\mathcal{R}'$:

19	31	35	**49**	65	**92**		

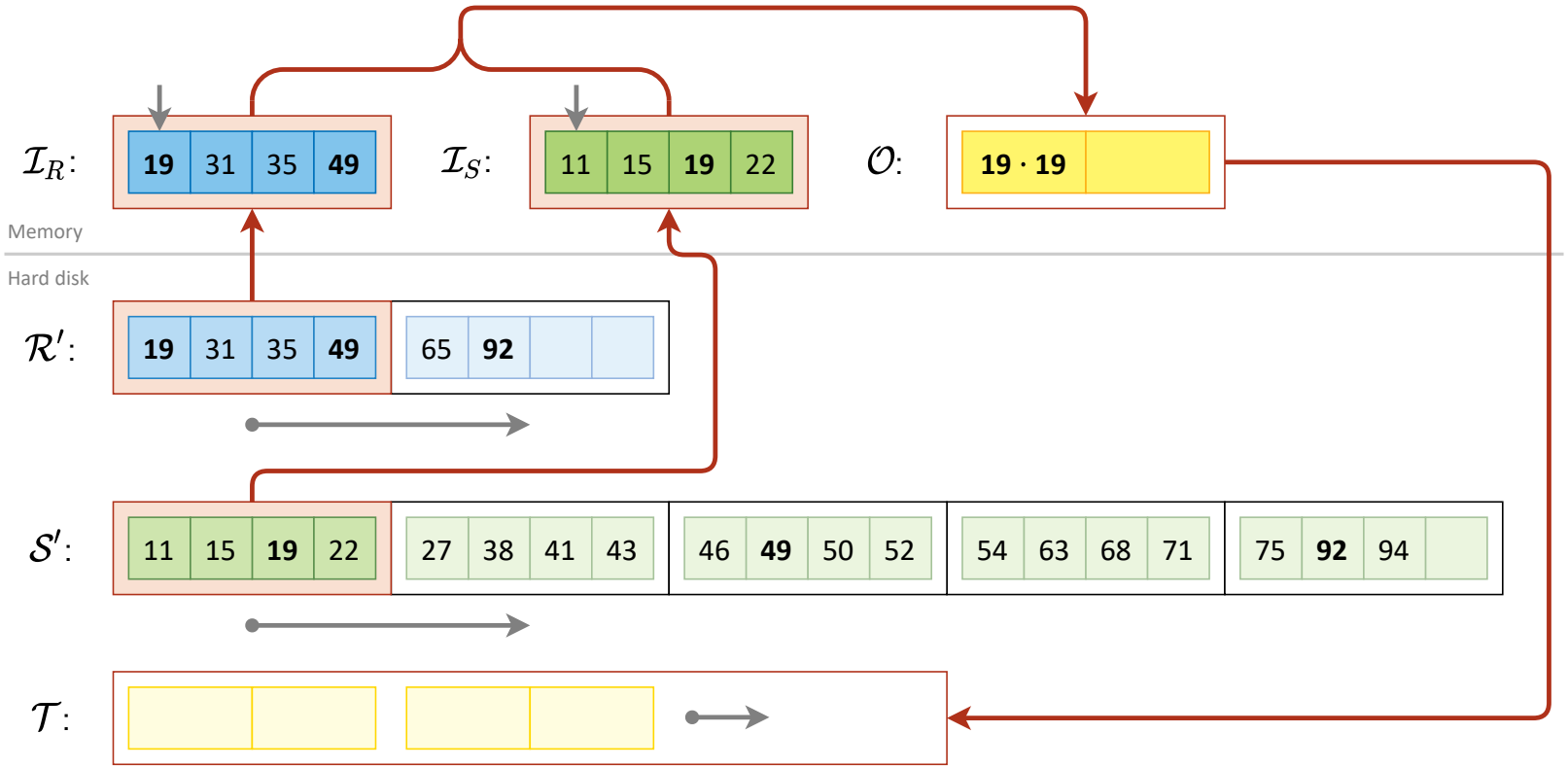
$\mathcal{S}'$:

11	15	**19**	22	27	38	41	43	46	**49**	50	52	54	63	68	71	75	**92**	94	

Basic Approach

Joining phase

- Blocks from the sorted tables are processed one by one
  - Internal pointers are used to iterate over the individual items



Algorithm

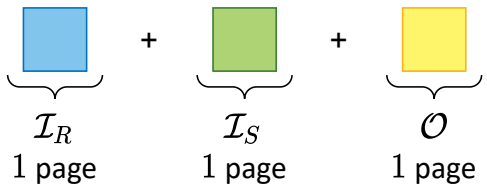
Joining phase

```
1  read blocks R'[1] to I_R and S'[1] to I_S
2  while both I_R and I_S contain at least one item do
3      let r be the minimal item in I_R and s minimal in I_S
4      if value(r) = value(s) then
5          join r and s and put the result to O
6          if O is full then write O to T
7          remove r from I_R and s from I_S
8      else remove the lower one from r or s
9      if I_R and / or I_S are empty then
10         read the next block from R' and / or S' (if any)
11 if O is not empty then write O to T
```

Observations

Joining phase

- System memory usage: $1 + 1 + 1$ pages
  - **Input buffer** $\mathcal{I}_R$: 1 page
  - **Input buffer** $\mathcal{I}_S$: 1 page
  - **Output buffer** $\mathcal{O}$: 1 page



Time complexity

- Sorting phase
- **Joining phase**
  - $c_{\texttt{read}} = p_R + p_S$

Improved Approach

2-pass integrated sort-merge join with **priority queue**

- **Sorting phase** (pass 1)
  - Tables are processed one by one
  - **Priority queue** is used to generate runs
  - **Required memory**: $M + 1 + 1$ pages
  - $M \approx \sqrt{p}$, where $p = \max(p_R, p_S)$
- Intermediate runs
  - Expected **length of runs** is $2M \approx 2\sqrt{p}$
  - Expected **number of all runs** is $p_S/2M + p_R/2M \approx \sqrt{p} \approx M$
- **Joining phase** (pass 2)
  - All runs from both the presorted tables are merged at once
  - Joined items are yielded directly without finishing the sorting
  - **Required memory**: $M + 1$ pages

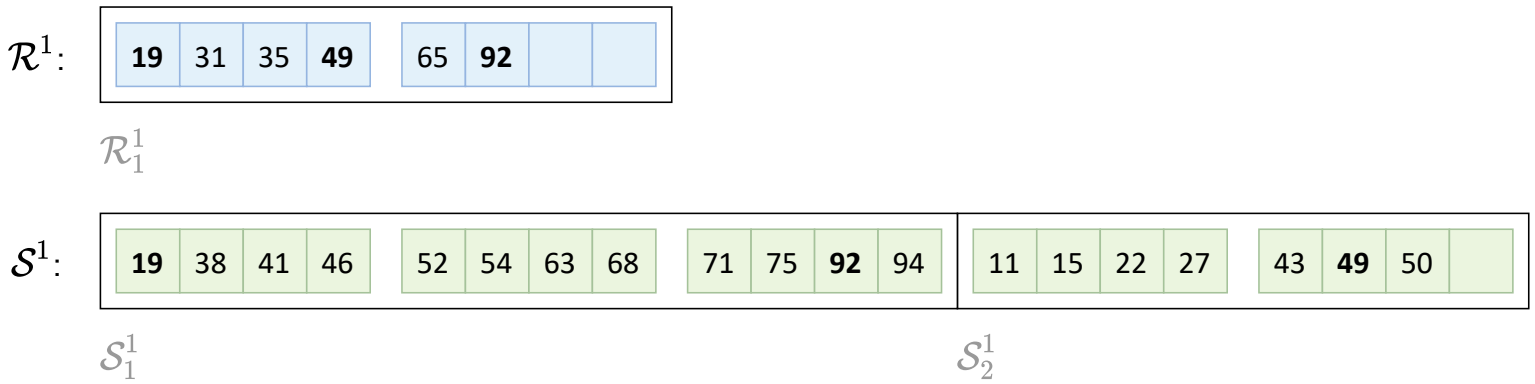
Improved Approach

Input tables

- Assumption
  - $M \approx \sqrt{5} \approx 3$

Sorting phase (pass 1)

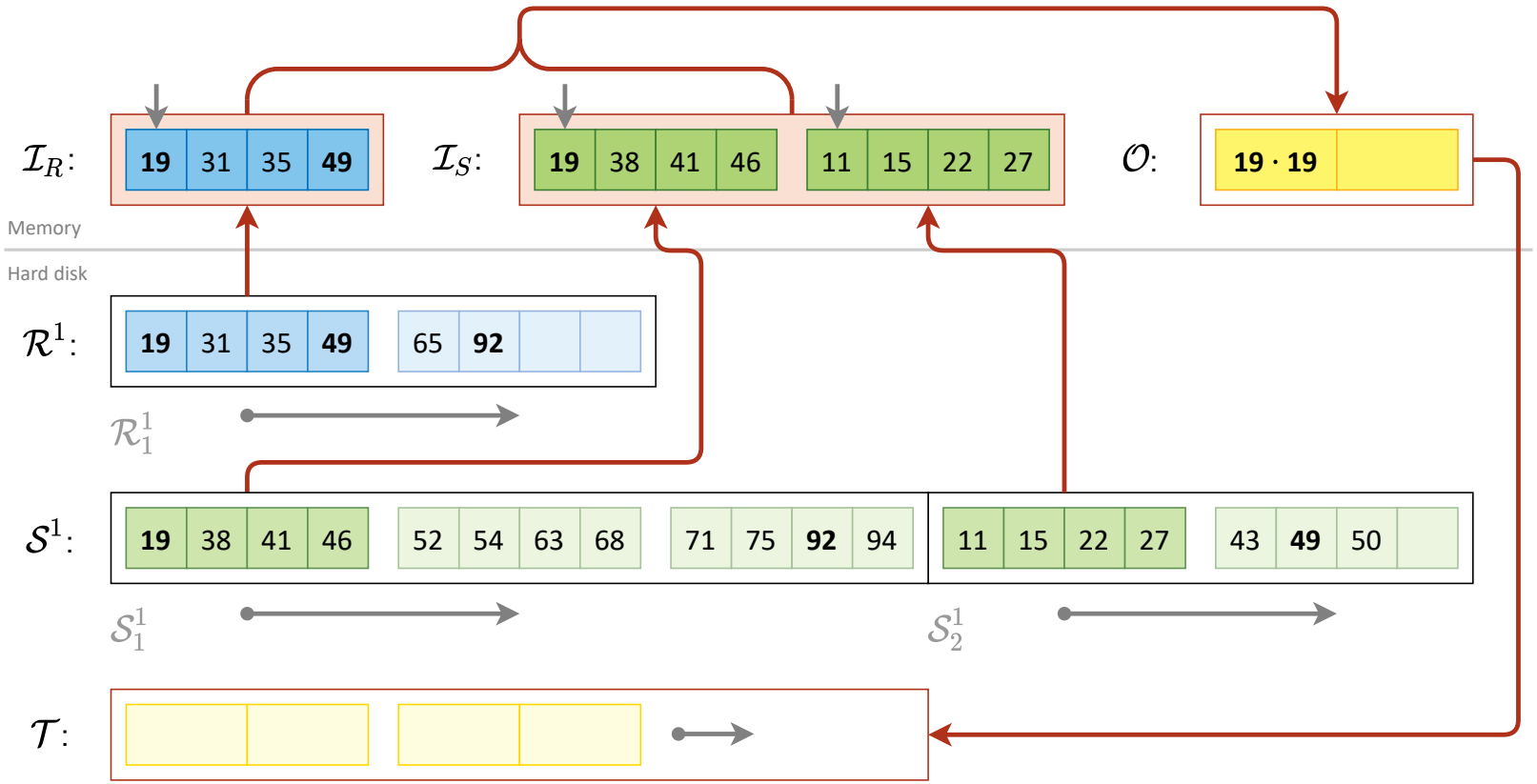
- Generated tables $\mathcal{R}^1$ and $\mathcal{S}^1$ with presorted runs



Improved Approach

Joining phase (pass 2)

- All runs from both the tables $\mathcal{R}^1$ and $\mathcal{S}^1$ are merged at once



Algorithm

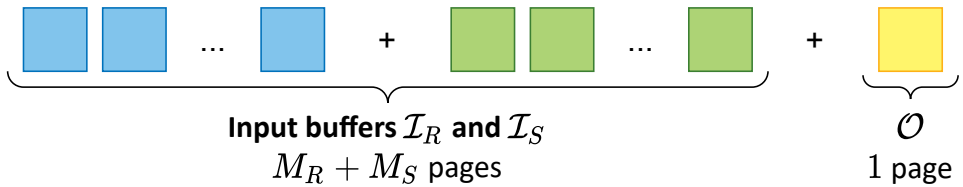
Improved approach: **joining phase**

1 read $\mathcal{R}^1_x[1]$ from each run x in $\mathcal{R}^1$ to $\mathcal{I}_R[x]$, the same for $\mathcal{S}^1$
2 **while** both $\mathcal{I}_R$ and $\mathcal{I}_S$ contain at least one item **do**
3 let r be the minimal item in $\mathcal{I}_R$ and s minimal in $\mathcal{I}_S$
4 **if** $value(r) = value(s)$ **then**
5 join r and s and put the result to $\mathcal{O}$
6 **if** $\mathcal{O}$ is full **then** write $\mathcal{O}$ to $\mathcal{T}$
7 remove r from $\mathcal{I}_R$ and s from $\mathcal{I}_S$
8 **else** remove the lower one from r or s
9 **if** the corresponding $\mathcal{I}_R[x]$ or $\mathcal{I}_S[y]$ are empty **then**
10 read the next block from $\mathcal{R}^1_x$ or $\mathcal{S}^1_y$ (if any)
11 **if** $\mathcal{O}$ is not empty **then** write $\mathcal{O}$ to $\mathcal{T}$

Observations

Improved approach

- System memory usage (**joining phase**): $M_R + M_S + 1$ pages
  - **Input buffer** $\mathcal{I}_R$: M_R pages $=$ number of runs in $\mathcal{R}^1$
  - **Input buffer** $\mathcal{I}_S$: M_S pages $=$ number of runs in $\mathcal{S}^1$
  - **Output buffer** $\mathcal{O}$: 1 page



- $M_R + M_S \approx M \approx \sqrt{p}$
  - The actual number of required pages may differ from M

Overall **time complexity**

- $c = c_{\texttt{sort}} + c_{\texttt{join}} = 2(p_R + p_S) + (p_R + p_S) = 3(p_R + p_S)$

Query Evaluation

Query Evaluation

Evaluation process

- SQL `SELECT` statement is parsed and transformed
- Query tree with (extended) **RA operations** is constructed

Query tree

- **Leaf nodes** correspond to the input **tables**
- **Inner nodes** correspond to the individual **operations**
  - Selection σ, projection π, …
- Root node represents the entire query
- Nodes are evaluated from leaves towards the root

Query Evaluation

Query evaluation plan = **query tree + algorithms**

- Suitable **algorithm** needs to be chosen for each operation
  - Based on the **context** and available system **memory**
- Overall **evaluation cost** is calculated
  - Measured in a number of **read / written blocks**
    - Basic **statistics** for all the nodes need to be calculated, too

Sample Query

Database schema

- **Movie** (id, title, year, …)
- **Actor** (movie, actor, character, …)
  - FK: (movie) $\subseteq$ Movie (id)

Sample query

- **Actors and characters they played in movies created in 2020**
  - SQL:

```
SELECT title, actor, character
FROM Movie JOIN Actor ON (id = movie)
WHERE year = 2020
```

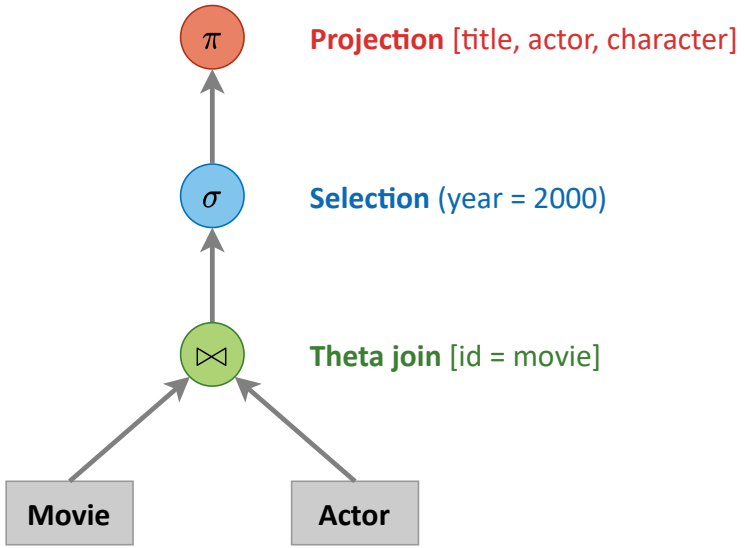
  - RA:

```
(Movie[id = movie]Actor)(year = 2000)
  [title, actor, character]
```

Sample Query

Sample query (cont'd)

- **Actors and characters they played in movies created in 2020**
  - `(Movie[id = movie]Actor)(year = 2000)`
    `[title, actor, character]`



Result Size Estimation

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

- **Tuple size**
  - $s_T = s_E$
- **Blocking factor**
  - $b_T = b_E$
- **Number of tuples**
  - $n_T = 1$ for **equality test** over a **unique** attribute
  - $n_T = n_E \cdot (1/V_{E.A})$ the same over a **non-unique** attribute A
  - $n_T = n_E \cdot (k_2 - k_1)/(max_{E.A} - min_{E.A})$ for a **range query** over an attribute A between boundaries k_1 and k_2
  - $n_T = n_E \cdot (\prod_{i=1}^{k} R_i)$ for a **conjunction** of independent atomic conditions with **reduction factors** $R_1, \ldots, R_k$
  - …

Result Size Estimation

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

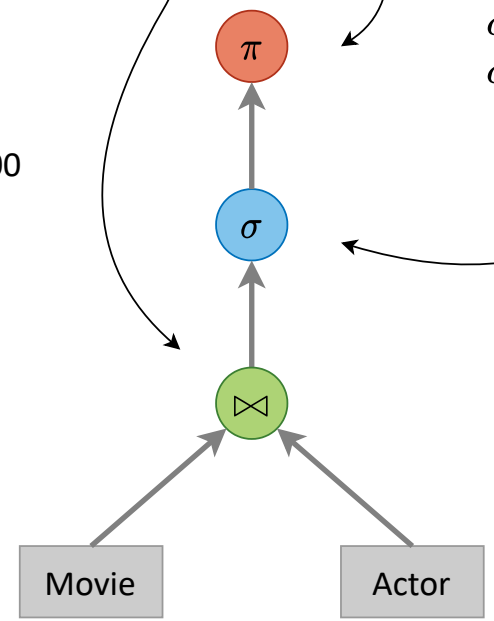
- **Tuple size**
  - s_T is derived from sizes of attributes $a_1, \ldots, a_n$
- **Blocking factor**
  - $b_T = \lfloor B / s_T \rfloor$
- **Number of tuples** for SQL without the `DISTINCT` modifier
  - $n_T = n_E$
- **Number of tuples** otherwise (i.e., without duplicates)
  - $n_T = n_E$ if attributes $a_1, \ldots, a_n$ contain at least one key of E
  - …

Result Size Estimation

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

- **Tuple size**
  - $s_T \approx s_R + s_S$
- **Blocking factor**
  - $b_T \approx \left\lfloor \frac{B}{s_T} \right\rfloor \approx \left\lfloor \frac{B}{B/b_R + B/b_S} \right\rfloor \approx \left\lfloor \frac{b_R \cdot b_S}{b_R + b_S} \right\rfloor$
- **Number of tuples**
  - $n_T = n_R \cdot (n_S / V_{S.A})$ for **equality test** over an attribute A in S
  - $n_T = n_R$ the same for a **unique attribute** A in S
  - $n_T = n_R \cdot n_S$ for **cross join**
  - …

Sample Query: Plan #1



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

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

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)
$f_{M.year} = 200$
$I_{M.year} = 3$

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

Evaluation Plan Cost

Overall **evaluation cost**

- Naive approach
  - Each operation **reads all its inputs** from hard disk
  - Each operation **write its output** to hard disk

Sample **Plan #1**

- $M = 25 + 1 + 1$ memory pages
- $c = [c_1^{\mathtt{r}} + c_1^{\mathtt{w}}] + [c_2^{\mathtt{r}} + c_2^{\mathtt{w}}] + [c_3^{\mathtt{r}}]$
- $c = [p_M + (p_M/25) \cdot p_A + p_1] + [p_1 + p_2] + [p_2]$
- $c = [10\,010\,000 + 125\,000] + [125\,000 + 2\,500] + [2\,500]$
- $c = 10\,265\,000$

Evaluation Plan Cost

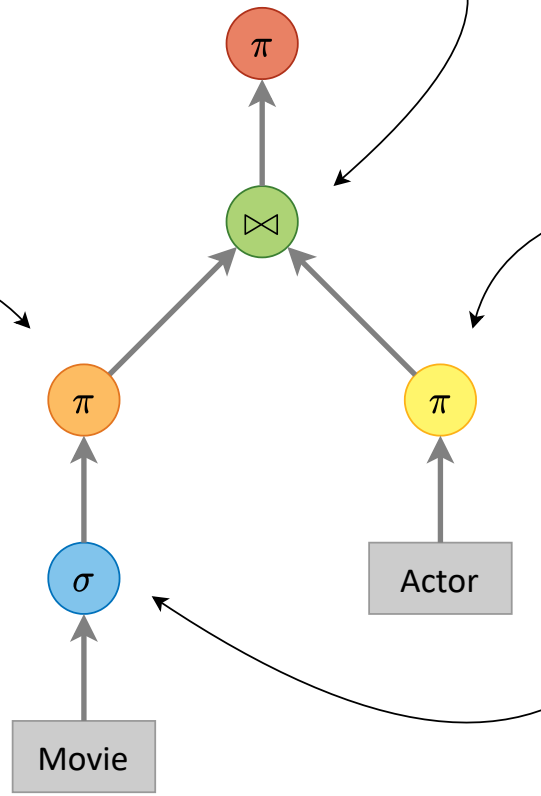
Overall **evaluation cost** (cont'd)

- **Pipelining**
  - **Output of one operation si directly forwarded to the next one** whenever possible
    - I.e., without hard disk usage
    - Unfortunately, not always possible…

Sample **Plan #1 with pipelining**

- $M = 25 + 1 + 1$ memory pages
- $c = [c_1^{\mathtt{r}} + \cancel{c_1^{\mathtt{w}}}] + [\cancel{c_2^{\mathtt{r}}} + \cancel{c_2^{\mathtt{w}}}] + [\cancel{c_3^{\mathtt{r}}}]$
- $c = 10\,010\,000$

Sample Query: Plan #2



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

Theta join [id = movie]
$n_4 = n_2 \cdot (n_A \cdot (1/V_{A.movie})) = 20\,000$
$b_4 = (b_2 \cdot b_3)/(b_2 + b_3) \approx 35$
Nested loops
$M_4 = p_2 + 1 + 1 = 27$

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

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

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)
$f_{M.year} = 200$
$I_{M.year} = 3$

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

Evaluation Plan Cost

Sample **Plan #2 with pipelining**

- $M = 25 + 1 + 1$ memory pages
- $c = [c_1^{\mathtt{r}} + \cancel{c_1^{\mathtt{w}}}] + [\cancel{c_2^{\mathtt{r}}} + \cancel{c_2^{\mathtt{w}}}] + [c_3^{\mathtt{r}} + \cancel{c_3^{\mathtt{w}}}] + [\cancel{c_4^{\mathtt{r}}} + \cancel{c_4^{\mathtt{w}}}] + [\cancel{c_5^{\mathtt{r}}}]$
- $c = [I_{M.year} + p_M \cdot (1 / V_{M.year})] + [p_A]$
- $c = [203] + [25\,000]$
- $c = 25\,203$

Query Optimization

Objective

- Finding the most **optimal query evaluation plan**
  - It is not possible to consider all the possible plans, though
    - Because there are far too many of them
  - Only suitable plans are selected based on **heuristics**

Strategies

- **Algebraic optimizations**
  - Equivalent transformations of query evaluation trees enabled by commutativity, associativity or other features of operations
    - E.g.: selections and projections as soon as possible
- **Statistical optimizations**
  - Estimation of result sizes based on histograms, …
- **Syntactical optimizations**

Observations

False assumptions and simplifications

- **Variable sizes** of records
- Inner **fragmentation** within blocks
- Overflow areas within primary files
- Outer fragmentation of files
- Extent of available information
- Lazy maintenance of **statistics**
- **Uneven distribution** of data as well as queries
- Independence of conditions in reduction factors
- Impact of **caching** manager

Conclusion

Evaluation **algorithms**

- Sort
  - N-way **external merge sort** algorithm
    - Priority queue
- Join
  - Binary (block) **nested loops** algorithm
    - Zig-zag
  - **Sort-merge join** algorithm
    - 2-pass integrated sort-merge join with priority queue

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

- Query evaluation **plans**
  - Overall cost, pipelining