

NDBI049: Query Languages

<http://www.ksi.mff.cuni.cz/~svoboda/courses/NDBI049/>

Lectures 2 – 5

Query Evaluation

Martin Svoboda

martin.svoboda@matfyz.cuni.cz

7. 10. – 4. 11. 2025

Charles University, Faculty of Mathematics and Physics

Lecture Outline

Algorithms

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

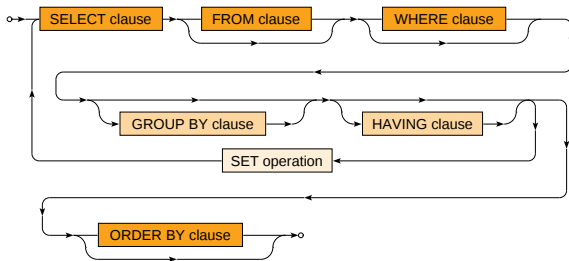
Evaluation

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

Introduction

SQL queries

- **SELECT statements**



Introduction

Relational algebra

- Basic and inferred operations
 - Selection σ_φ , projection $\pi_{a_1, \dots, a_n}$, renaming $\rho_{b_1/a_1, \dots, b_n/a_n}$
 - 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 $\lhd, \rhd, \lhd_\varphi, \rhd_\varphi$
 - Division $\div$
- Extended operations
 - Left / right / full outer natural join $\Join, \Joinr, \Joinl$
 - Left / right / full outer theta join $\Join_\varphi, \Joinr_\varphi, \Joinl_\varphi$
 - Sorting, grouping and aggregation, distinct, ...

Naïve Algorithms

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

- Iteration over all tuples and removal of those filtered out

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

- 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

⇒ **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 $\approx 4 \text{ kB}$
- M : number of available **system memory** pages

Relation $\mathcal{R}$

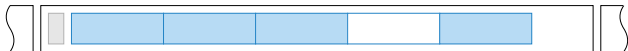
- n_R : **number of tuples**
- s_R : average / fixed tuple size
- $b_R \approx \lfloor B/s_R \rfloor$: **blocking factor**
 - Number of tuples that can be stored within one block
- $p_R \approx \lceil n_R/b_R \rceil$: **number of blocks**
- $V_{R.A}$: cardinality of the **active domain** of attribute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



Access Methods

Access method

- Particular approach for **finding the intended tuples**
 - I.e., reading blocks with such tuples into the system memory
 - Directly from **data files** for tables
 - But also indirectly using **index structures**
- **Full scan (sequential read)** is possible under all circumstances
 - However, we can do better in certain cases based on...
 - Involved **selection conditions**
 - Particular **data file organization**
 - Available **index structures** (if any)
 - I.e., number of blocks to be read can significantly be reduced
 - And so the evaluation cost
 - Since only relevant blocks are considered instead all of them

Access Methods

Data file organization

- **Heap file, sorted file, hashed file**

Index structures

- **B⁺ tree, ...**

Selection conditions

- **Equality tests** with respect to **unique / non-unique attributes**
 - $A = v$, where v is a particular value (not another attribute)
- **Range queries** for **one-sided / two-sided intervals**
 - $v_1 \leq A$, $A \leq v_2$, and $(v_1 \leq A) \wedge (A \leq v_2)$
 - Analogously for other comparison operators ($\geq$, $<$, $>$)
 - As well as their mutual combinations in two-sided intervals
 - However, only fixed boundary values are assumed again
- ...

Heap File

Heap file

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

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

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			

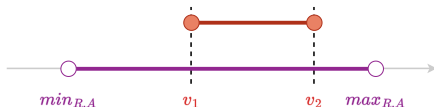
Selection costs

- 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

Sorted File

Selection costs (cont'd)

- **Range query** for two-sided intervals $[v_1, v_2]$ and other

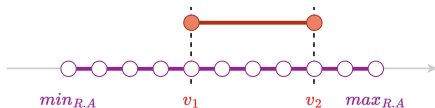


- For "**continuous**" domains...
 - Number of values between any two of them is not limited
 - At least potentially
 - In practical terms, there can simply be far too many of them
 - E.g.: FLOAT, VARCHAR, ...
 - $c = \lceil \log_2 p_R \rceil + \lceil p_R \cdot (v_2 - v_1) / (max_{R.A} - min_{R.A}) \rceil$
 - Boundary types (inclusive / exclusive) are unimportant

Sorted File

Selection costs (cont'd)

- **Range query** for two-sided intervals $[v_1, v_2]$ and other



- For "**discrete**" domains...
 - Number of values between any two of them is finite
 - E.g.: INTEGER, CHAR, DATE, ...
 - $$c = \lceil \log_2 p_R \rceil + \lceil p_R \cdot (v_2 - v_1 + \varepsilon) / (max_{R.A} - min_{R.A} + 1) \rceil$$
 - ε is 1 for closed intervals, -1 for open (unless $v_1 = v_2$), and 0 otherwise, i.e., half-open and zero-sized open

Sorted File

Selection costs (cont'd)

- **Range query** for one-sided intervals $(-\infty, v_2]$ and $(-\infty, v_2)$



- $c = \lceil p_R \cdot (v_2 - \min_{R.A}) / (\max_{R.A} - \min_{R.A}) \rceil$
- $c = \lceil p_R \cdot (v_2 - \min_{R.A} + \varepsilon) / (\max_{R.A} - \min_{R.A} + 1) \rceil$
- **Range query** for one-sided intervals $[v_1, \infty)$ and $[v_1, \infty)$
 - Analogously...
 - $c = \lceil p_R \cdot (\max_{R.A} - v_1) / (\max_{R.A} - \min_{R.A}) \rceil$
 - $c = \lceil p_R \cdot (\max_{R.A} - v_1 + \varepsilon) / (\max_{R.A} - \min_{R.A} + 1) \rceil$

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., $h(A) = A \bmod 3$

18	42	75	36
93	18	6	
$h(A) = 0$			
82	34	49	25
$h(A) = 1$			
53	20	23	53
11	71		
$h(A) = 2$			

- **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

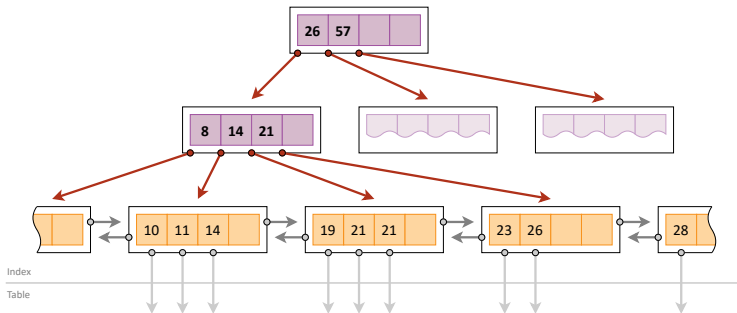
Logical structure

- **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.

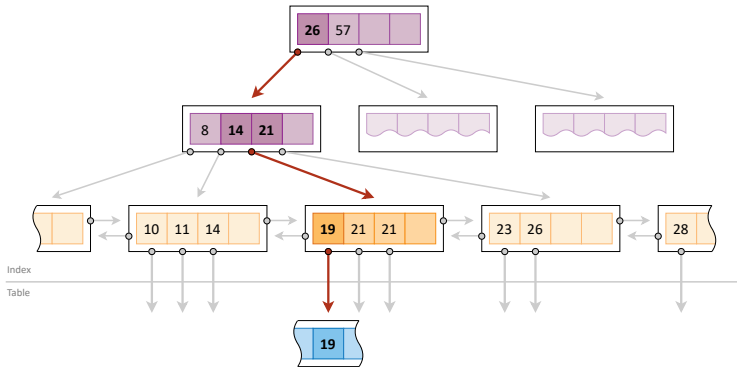
Index statistics

- $m_{R.A}$: 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

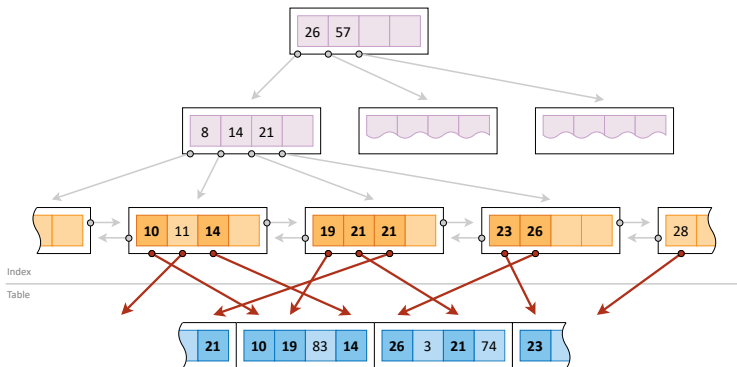
- Index is traversed from its root toward the corresponding leaf
 - Data tuple then needs to be fetched from the data file



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**



Non-Clustered B⁺ Tree Index

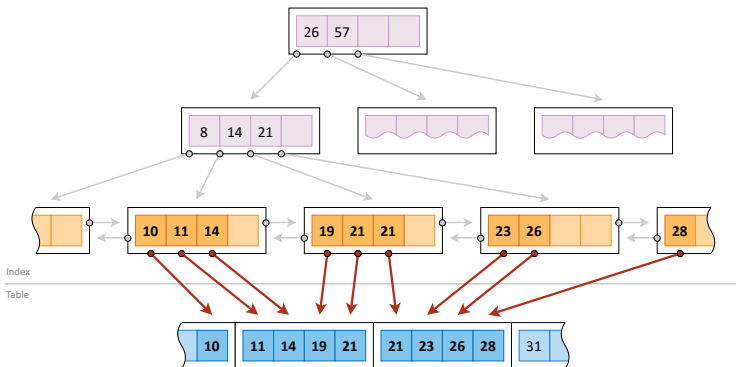
Selection costs

- **Equality test** for a **unique / non-unique** attribute
 - $c = I_{R.A} + 1$
 - $c = I_{R.A} + \lceil p_{R.A} / V_{R.A} \rceil + \min(p_R, \lceil n_R / V_{R.A} \rceil)$
- **Range query** for two-sided intervals $[v_1, v_2]$ and other
 - $c = I_{R.A} + \lceil p_{R.A} \cdot (v_2 - v_1) / (max_{R.A} - min_{R.A}) \rceil + \min(p_R, \lceil n_R \cdot (v_2 - v_1) / (max_{R.A} - min_{R.A}) \rceil)$
 - Analogously for discrete domains
- However, for small domains $V_{R.A}$ or large intervals...
 - Full scan of the data file is better
 - I.e., index is not utilized at all
- Conditions not involving the indexed attribute
 - Full scan again, of course

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)



Clustered B⁺ Tree Index

Selection costs

- **Equality tests**
 - $c = I_{R.A} + 1$ for a **unique** attribute
 - $c = I_{R.A} + \lceil p_R / V_{R.A} \rceil$ for a **non-unique** attribute
- **Range query** for two-sided intervals $[v_1, v_2]$ and other
 - $c = I_{R.A} + \lceil p_R \cdot (v_2 - v_1) / (max_{R.A} - min_{R.A}) \rceil$
 - Analogously for discrete domains
- **Range query** for one-sided intervals
 - Data file is read directly as an ordinary sorted file
- Conditions not involving the indexed attribute
 - Full scan again, of course

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)
 - $c = I_{M.id} + 1 = 3 + 1 = 4$
- Clustered index (B⁺ tree & sorted file)
 - $c = I_{M.id} + 1 = 3 + 1 = 4$

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
 - $c = \lceil \log_2 p_M \rceil + \lceil p_M / V_{M.year} \rceil = 14 + 200 = 214$
- Hashed file
 - $c = C_M = 500$
- Non-clustered index (B^+ tree & heap file)
 - $c = I_{M.year} + \lceil p_{M.year} / V_{M.year} \rceil + \min(p_M, \lceil n_M / V_{M.year} \rceil)$
 $= 3 + 30 + 2\,000 = 2\,033$
- Clustered index (B^+ tree & sorted file)
 - $c = I_{M.year} + \lceil p_M / V_{M.year} \rceil = 3 + 200 = 203$

Examples

Range query: movies filmed during years $[y_1 = 2016, y_2 = 2020]$

- Heap file
 - $c = p_M = 10\,000$
- Sorted file
 - Let $r \leftarrow (y_2 - y_1 + 1) / (\max_{M.\text{year}} - \min_{M.\text{year}} + 1) = 5/80$
 - $c = \lceil \log_2 p_M \rceil + \lceil p_M \cdot r \rceil = 14 + 625 = 639$
- Hashed file
 - $c = p_M = 10\,000$
- Non-clustered index (B^+ tree & heap file)
 - $c = I_{M.\text{year}} + \lceil p_{M.\text{year}} \cdot r \rceil + \min(p_M, \lceil n_M \cdot r \rceil)$
 $= 3 + 94 + 6\,250 = 6\,347$
- Clustered index (B^+ tree & sorted file)
 - $c = I_{M.\text{year}} + \lceil p_M \cdot r \rceil = 3 + 625 = 628$

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

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

- System memory layout

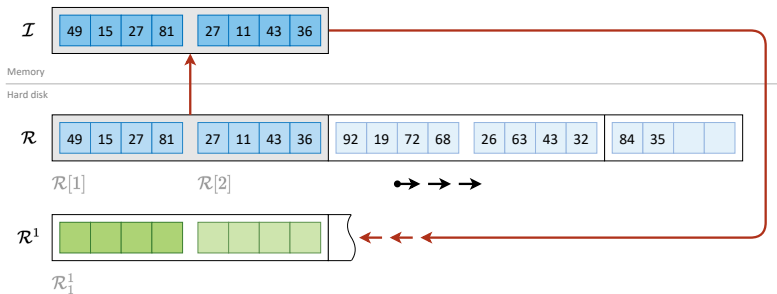
- **Input buffer $\mathcal{I}$**

- E.g., size $M = 2$ pages

Sort Phase

Pass 1

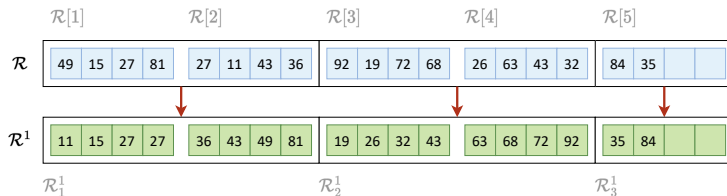
- **Groups of M blocks** are presorted and so **initial runs** created
 - Input blocks from $\mathcal{R}$ are first loaded to $\mathcal{I}$
 - Individual tuples in $\mathcal{I}$ are then sorted
 - Created runs are stored to a temporary file $\mathcal{R}^1$



Sort Phase

Pass 1

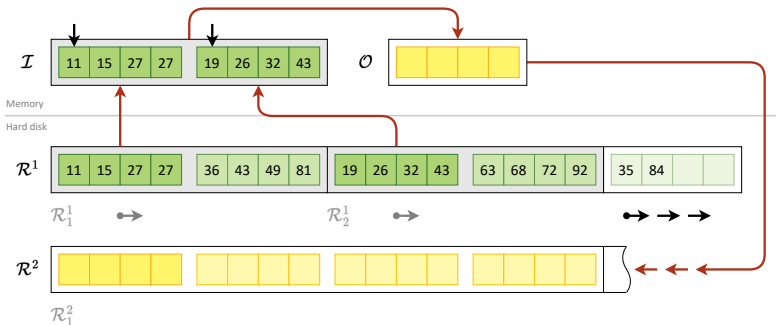
- Resulting runs in $\mathcal{R}^1$ within our sample scenario



Merge Phase

Pass 2

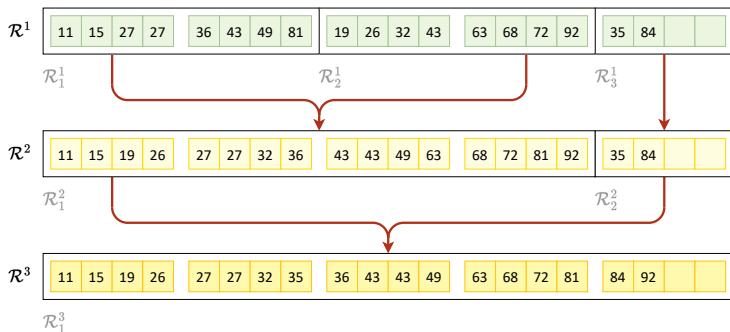
- **Groups of M runs are iteratively merged together**
 - Blocks from these input runs are gradually loaded into $\mathcal{I}$
 - Minimal items are then iteratively selected and moved to $\mathcal{O}$
 - Merged (longer) runs are written to a new temporary file $\mathcal{R}^2$



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)

```
1  $p \leftarrow 1$ 
2 foreach group of blocks  $B_1, \dots, B_M$  (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  $\mathcal{R}^p$ 
```

Algorithm

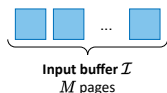
Merge phase (passes 2 and higher)

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

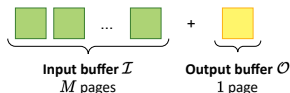
Summary

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)
 - $c_{\text{read}} = c_{\text{write}} = p_R$
- **Number of passes**
 - $t = \lceil \log_M(p_R) \rceil$
- **Overall cost**
 - $c_{\text{ES}} = t \cdot (c_{\text{read}} + c_{\text{write}}) = \lceil \log_M(p_R) \rceil \cdot 2p_R$

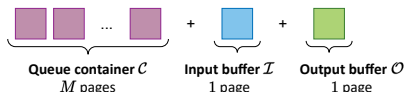
Limitation of the overall number of passes

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

Improved Approach

N-way external merge sort with priority queue

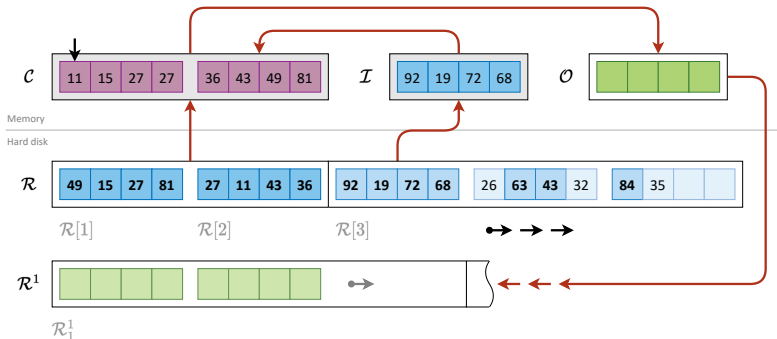
- **Sort phase** is modified
 - Instead of fixed-size initial runs...
 - ... we generate them using a **priority queue**
 - In particular, **min-heap data structure** is used
 - The aim is to **make the initial runs longer**
- **Memory layout:** $M + 1 + 1$
 - **Queue container $\mathcal{C}$:** $M \geq 1$ pages
 - **Input buffer $\mathcal{I}$:** 1 page
 - **Output buffer $\mathcal{O}$:** 1 page



Sort Phase

Pass 1

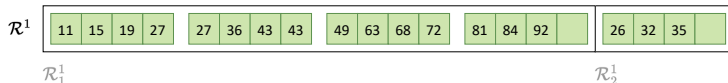
- Once the queue is initialized, **runs are generated on the fly**
 - Minimal item greater than or equal to the last value is always extracted and replaced with another item from the input file



Sort Phase

Pass 1 (cont'd)

- Two runs are obtained in our scenario



Impact summary

- Created initial **runs will 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 the runs will tend to be lower**

Algorithm

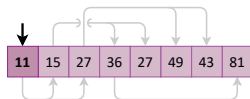
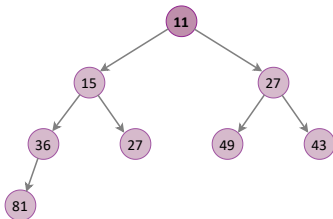
Improved **sort phase** (pass 1)

```
1 read blocks  $\mathcal{R}[1], \dots, \mathcal{R}[M]$  (if any) from  $\mathcal{R}$  to  $\mathcal{C}$ 
2 read block  $\mathcal{R}[M+1]$  (if any) from  $\mathcal{R}$  to  $\mathcal{I}$ 
3 while  $\mathcal{C}$  contains at least one item do
4     start constructing a new run in  $\mathcal{R}^1$ , put  $v \leftarrow -\infty$ 
5     while  $\mathcal{C}$  contains at least one item  $i \geq v$  do
6         let  $i$  be the minimal one, move  $i$  to  $\mathcal{O}$ , put  $v \leftarrow i$ 
7         move the next item from  $\mathcal{I}$  (if any) to  $\mathcal{C}$ 
8         if  $\mathcal{I}$  is empty then
9             read the next block from  $\mathcal{R}$  (if any) to  $\mathcal{I}$ 
10        if  $\mathcal{O}$  is full then write  $\mathcal{O}$  to  $\mathcal{R}^1$  and empty  $\mathcal{O}$ 
11    if  $\mathcal{O}$  is not empty then write  $\mathcal{O}$  to  $\mathcal{R}^1$  and empty  $\mathcal{O}$ 
```

Priority Queue

Min-heap data structure

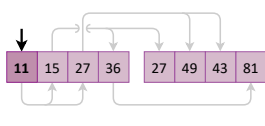
- Complete binary tree
 - Key associated with each node must be less than or equal to keys of all its child nodes
 - I.e., the **root node** contains the **minimal item** among them all
- **Array representation** is possible
 - Using a straightforward index arithmetic



Queue Container

Queue container $\mathcal{C}$

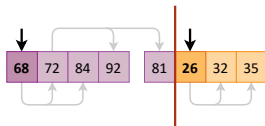
- **Two separate min-heap structures** are in fact used
 - **Active heap** with items greater than or equal to the last value
 - And so values that can still be (actually all really will be) used in the currently constructed run
 - **Inactive heap** with items not satisfying the condition
- Both are **represented as arrays**
 - Directly inside the container blocks
- **Container initialization** (line 1)
 - Active heap is built from the input items, inactive heap is empty



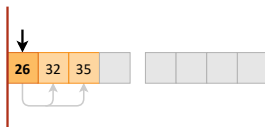
Queue Container

Queue container $\mathcal{C}$ (cont'd)

- Whenever an **item is added to the container** (line 7)
 - It is added to the active / inactive heap based on the condition



- Whenever the **active heap is fully depleted** (line 5)
 - I.e., the current run terminated, both the heaps are swapped



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

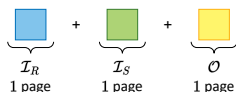
Sample input data

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

$\mathcal{R}$	<table><tr><td>21</td><td>84</td><td>56</td><td>19</td></tr></table>	21	84	56	19	<table><tr><td>41</td><td>72</td><td>69</td><td>35</td></tr></table>	41	72	69	35	<table><tr><td>56</td><td>84</td><td></td><td></td></tr></table>	56	84												
21	84	56	19																						
41	72	69	35																						
56	84																								
$\mathcal{S}$	<table><tr><td>31</td><td>56</td><td>75</td><td>43</td></tr></table>	31	56	75	43	<table><tr><td>88</td><td>21</td><td>43</td><td>14</td></tr></table>	88	21	43	14	<table><tr><td>92</td><td>52</td><td>25</td><td>81</td></tr></table>	92	52	25	81	<table><tr><td>72</td><td>37</td><td>64</td><td>35</td></tr></table>	72	37	64	35	<table><tr><td>14</td><td>64</td><td></td><td></td></tr></table>	14	64		
31	56	75	43																						
88	21	43	14																						
92	52	25	81																						
72	37	64	35																						
14	64																								

Basic setup

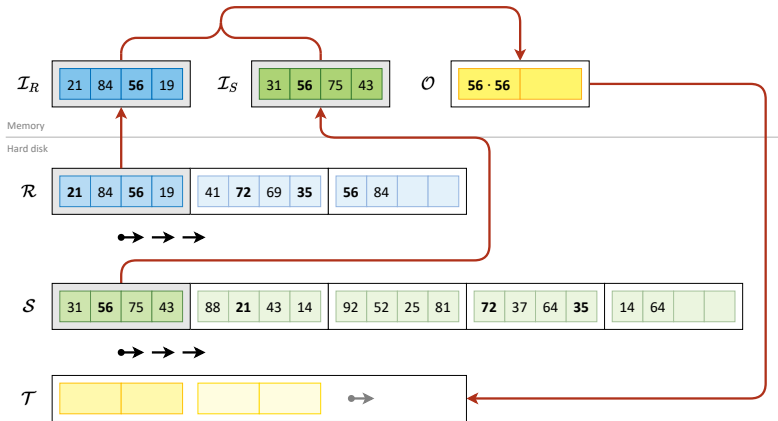
- Memory layout: $1 + 1 + 1$
 - Input buffer $\mathcal{I}_R$: 1 page
 - Input buffer $\mathcal{I}_S$: 1 page
 - Output buffer $\mathcal{O}$: 1 page



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)

```
1  foreach block  $R$  from  $\mathcal{R}$  do
2      read  $R$  into  $\mathcal{I}_R$ 
3      foreach block  $S$  from  $\mathcal{S}$  do
4          read  $S$  into  $\mathcal{I}_S$ 
5          foreach item  $r$  in  $\mathcal{I}_R$  do
6              foreach item  $s$  in  $\mathcal{I}_S$  do
7                  if  $r$  and  $s$  satisfy the join condition then
8                      join  $r$  and  $s$  and put the result to  $\mathcal{O}$ 
9                      if  $\mathcal{O}$  is full then write  $\mathcal{O}$  to  $\mathcal{T}$ , empty  $\mathcal{O}$ 
10 if  $\mathcal{O}$  is not empty then write  $\mathcal{O}$  to  $\mathcal{T}$  and empty  $\mathcal{O}$ 
```

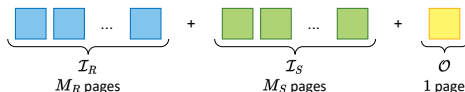
Observations

Time complexity

- Basic setup ($1 + 1 + 1$)
 - $c_{NL} = p_R + p_R \cdot p_S$
- $\Rightarrow$ **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 ($M_R + M_S + 1$)

```
1  foreach group of blocks  $R_1, \dots, R_{M_R}$  (if any) from  $\mathcal{R}$  do
2      read these blocks into  $\mathcal{I}_R$ 
3      foreach group of blocks  $S_1, \dots, S_{M_S}$  (if any) from  $\mathcal{S}$  do
4          read these blocks into  $\mathcal{I}_S$ 
5          foreach item  $r$  in  $\mathcal{I}_R$  do
6              foreach item  $s$  in  $\mathcal{I}_S$  do
7                  if  $r$  and  $s$  satisfy the join condition then
8                      join  $r$  and  $s$  and put the result to  $\mathcal{O}$ 
9                      if  $\mathcal{O}$  is full then write  $\mathcal{O}$  to  $\mathcal{T}$ , empty  $\mathcal{O}$ 
10 if  $\mathcal{O}$  is not empty then write  $\mathcal{O}$  to  $\mathcal{T}$  and empty  $\mathcal{O}$ 
```

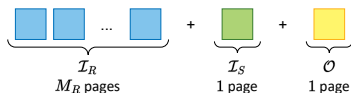
Observations

Time complexity

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

Standard setup

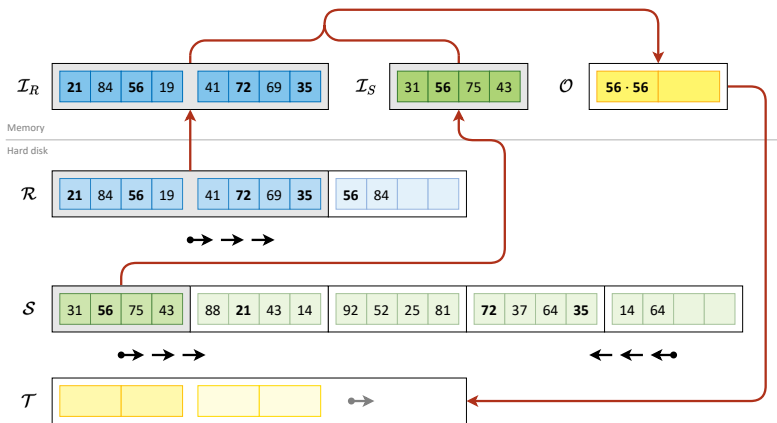
- Memory layout: $M_R + 1 + 1$
 - **Input buffer** $\mathcal{I}_R$: M_R pages
 - **Input buffer** $\mathcal{I}_S$: 1 page
 - **Output buffer** $\mathcal{O}$: 1 page



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

Special Cases

Very small tables

- Smaller table fits entirely within the memory, i.e., $p_R \leq M_R$
 - $c_{NL} = p_R + p_S$

Non-brute-force replacement for **inner loops**

- **B⁺ tree index exists** in S on attribute A that is unique in S
 - $c_{NL} = p_R + n_R \cdot (I_{S.A} + 1)$
 - If **R is organized as a heap**
 - $c_{NL} = p_R + I_{S.A} + p_{S.A} + V_{R.A}$
 - If **R is sorted** with respect to A
- **S is a hashed file** over attribute A that is unique in S
 - $c_{NL} = p_R + V_{R.A} \cdot C_S$
 - If **R is sorted** with respect to A
- ...

Non-Binary Nested Loops

Non-binary nested loops

- Nested loops algorithm for **multiple tables at once**
 - In particular, let us have tables $\mathcal{R}_1, \dots, \mathcal{R}_n$ for $n \geq 2, n \in \mathbb{N}$
 - Let their sizes be $p_1, \dots, p_n$
- Solution
 - We just need to embed more loops into each other
- **Memory layout:** $M_1 + \dots + M_n + 1$
 - **Input buffers** $\mathcal{I}_i$: M_i pages for each table $\mathcal{R}_i$
 - **Output buffer** $\mathcal{O}$: 1 page
- **Overall cost with zig-zag optimization**
 - $$c_{\text{NL}} = \binom{p_1}{1} + \left(\lceil p_1/M_1 \rceil \cdot (p_2 - M_2) + M_2 \right) + \dots + \left(\lceil p_1/M_1 \rceil \dots \lceil p_{n-1}/M_{n-1} \rceil \cdot (p_n - M_n) + M_n \right)$$

Memory Setup

Memory layout: $M_1 + \dots + M_n + 1$

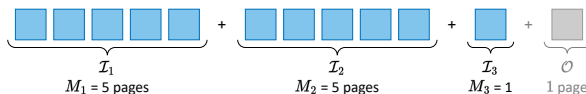
- Optimization problem
 - Finding integer M_i **minimizing the overall cost** c_{NL}
- **Heuristics**
 - Let $M \geq n$ be all the available pages (for input buffers)
 - Let $p_1 \leq \dots \leq p_n$ (without loss of generality)
 - Allocate **one page** for the **innermost table**, i.e., $M_n = 1$
 - Allocate the **remaining pages uniformly** to $\mathcal{R}_1, \dots, \mathcal{R}_{n-1}$
 - i.e., let $m = \lfloor (M-1)/(n-1) \rfloor$
 - Then put $M_i = m$ for each $i \in \{1, \dots, n-1\}$
 - It may happen that some pages will still be unallocated
 - There will be exactly $u = (M-1) - (n-1) \cdot m$ of them
 - Assign these **remaining pages** (if any) between **smaller tables**
 - i.e., $M_i += 1$ for each $i \in \{1, \dots, u\}$

Memory Setup

Memory layout (cont'd)

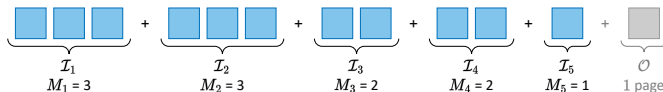
- Example #1

- $n = 3$ tables, $M = 11$ pages (for input buffers)
- Allocation: $\langle 5, 5, 1 \rangle$



- Example #2

- $n = 5$ tables, $M = 11$ pages
- Allocation: $\langle 3, 3, 2, 2, 1 \rangle$



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

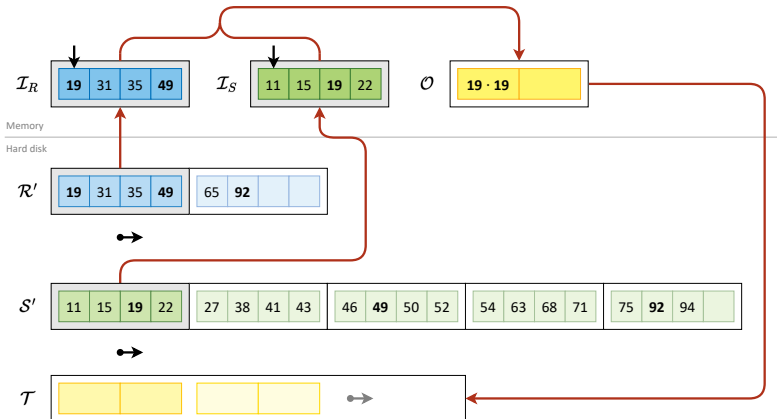
$\mathcal{R}'$	19	31	35	49	65	92		
----------------	----	----	----	----	----	----	--	--

$\mathcal{S}'$	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

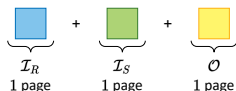
Join phase

```
1 read block  $\mathcal{R}'[1]$  to  $\mathcal{I}_R$  and block  $\mathcal{S}'[1]$  to  $\mathcal{I}_S$ 
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 item in  $\mathcal{I}_S$ 
4   if  $r$  and  $s$  can be joined then
5     join  $r$  and  $s$  and put the result to  $\mathcal{O}$ 
6     if  $\mathcal{O}$  is full then write  $\mathcal{O}$  to  $\mathcal{T}$  and empty  $\mathcal{O}$ 
7     remove both  $r$  from  $\mathcal{I}_R$  and  $s$  from  $\mathcal{I}_S$ 
8   else remove the lower one of  $r$  from  $\mathcal{I}_R$  or  $s$  from  $\mathcal{I}_S$ 
9   if  $\mathcal{I}_R$  is empty then read the next block from  $\mathcal{R}'$  (if any)
10  if  $\mathcal{I}_S$  is empty then read the next block from  $\mathcal{S}'$  (if any)
11 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



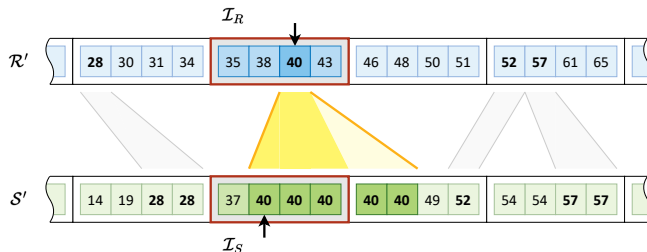
Time complexity

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

Extended Version

Duplicate items

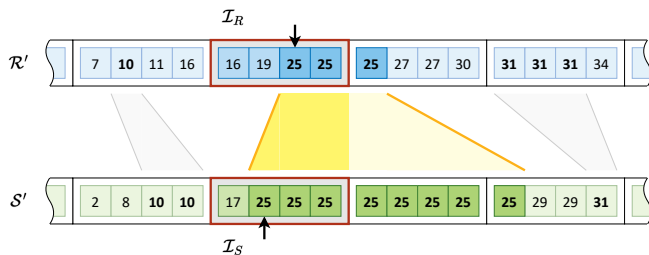
- Possible duplicates in **one table** only
 - Let it be $\mathcal{S}$ (without loss of generality)
 - Algorithm modification is straightforward...
 - Having successfully joined r and s , we just remove s from $\mathcal{I}_S$ and not r from $\mathcal{I}_R$ (line 7)



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



Integrated Approach

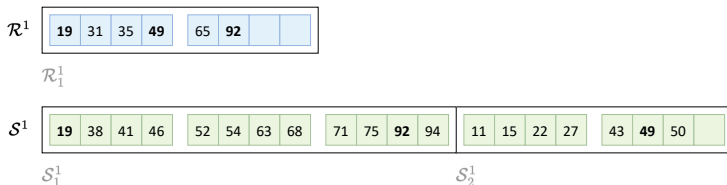
2-pass integrated sort-merge join with priority queue

- **Sort phase (pass 1)**
 - Tables are processed one by one
 - They are not sorted entirely, though
 - **Only initial runs are constructed**
 - Using just the **sort phase** (pass 1) of the **external sort** algorithm
 - **Priority queue** is involved to make these runs longer
 - And so their overall number lower
- **Join phase (pass 2)**
 - The same idea as in the basic sort-merge approach
 - We only have more runs within each presorted table

Integrated Approach

Sort phase (pass 1)

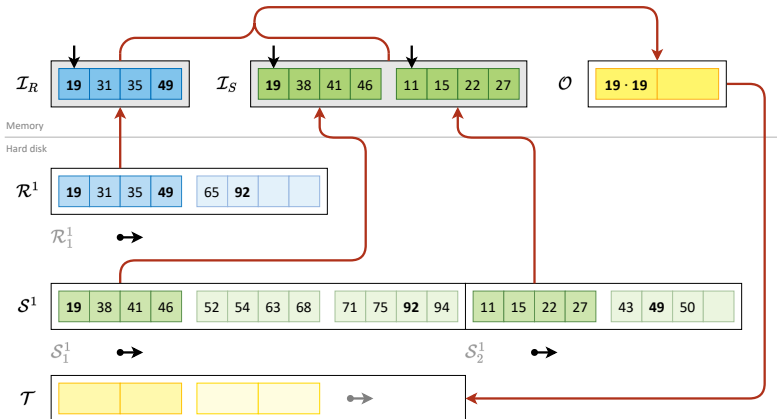
- Resulting initial runs within tables $\mathcal{R}^1$ and $\mathcal{S}^1$



Integrated Approach

Join phase (pass 2)

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



Algorithm

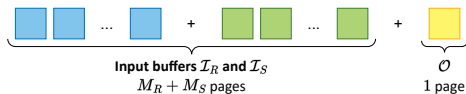
Join phase (pass 2)

```
1 read  $\mathcal{R}_x^1[1]$  from each run 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 item in  $\mathcal{I}_S$ 
4   if  $r$  and  $s$  can be joined then
5     join  $r$  and  $s$  and put the result to  $\mathcal{O}$ 
6     if  $\mathcal{O}$  is full then write  $\mathcal{O}$  to  $\mathcal{T}$  and empty  $\mathcal{O}$ 
7     remove both  $r$  from  $\mathcal{I}_R$  and  $s$  from  $\mathcal{I}_S$ 
8   else remove the lower one of  $r$  from  $\mathcal{I}_R$  or  $s$  from  $\mathcal{I}_S$ 
9   if the given  $\mathcal{I}_R[x]$  is empty then refill it from  $\mathcal{R}_x^1$ 
10  if the given  $\mathcal{I}_S[x]$  is empty then refill it from  $\mathcal{S}_x^1$ 
11 if  $\mathcal{O}$  is not empty then write  $\mathcal{O}$  to  $\mathcal{T}$  and empty  $\mathcal{O}$ 
```

Observations

Join phase (pass 2)

- Memory layout: $M_R + M_S + 1$
 - **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



Time complexity

- Sort phase: $c_{\text{sort}} = 2p_R + 2p_S$
- Join phase: $c_{\text{join}} = p_R + p_S$
- **Overall cost:** $c_{\text{MJ}} = c_{\text{sort}} + c_{\text{join}} = 3(p_R + p_S)$

Observations

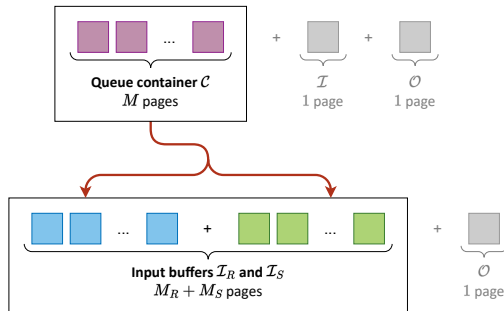
Optimized setup

- Motivation
 - Balanced memory usage across both phases
- **Sort phase (pass 1)**
 - Required memory: $M + 1 + 1$ pages
 - Let $M = \lceil \sqrt{p} \rceil$, where $p = \max(p_R, p_S)$
 - As if we wanted 2 passes for the external sort
 - If M **pages** are used for the **priority queue container**...
 - Expected **length of initial runs** should be $2M$
 - And so the expected **number of all runs** $p_S/2M + p_R/2M \leq p/2M + p/2M \approx 2p/2M = p/M \approx p/\sqrt{p} \approx \sqrt{p} \approx M$
- **Join phase (pass 2)**
 - Required memory: $M_R + M_S + 1$ pages
 - $\Rightarrow M_R + M_S \approx M$

Observations

Optimized setup (cont'd)

- In other words...
 - **The same number of M pages** should be sufficient for both...
 - Queue container $\mathcal{C}$ during **pass 1**, and
 - Input buffers $\mathcal{I}_R$ and $\mathcal{I}_S$ during **pass 2**



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

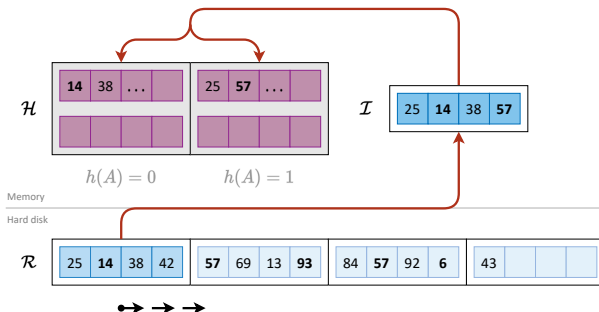
Classic hash join

- **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

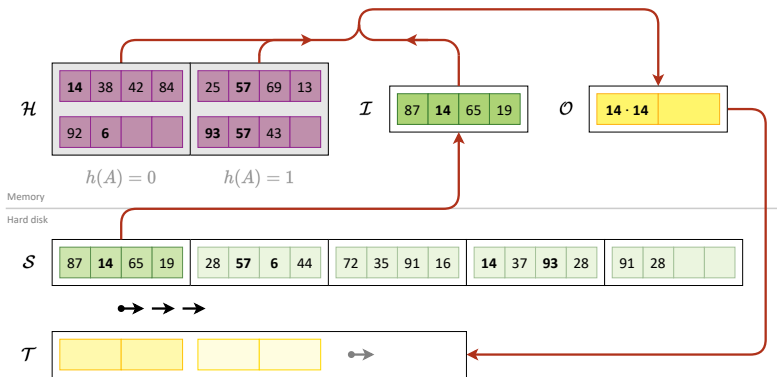
- Tuples from the smaller table are **hashed** into the memory
 - E.g., hash function $h(A) = A \bmod 2$ is assumed



Probe Phase

Probe phase

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



Algorithm

Build phase

```
1 foreach block  $R$  from  $\mathcal{R}$  do  
2   read  $R$  into  $\mathcal{I}$   
3   foreach item  $r$  in  $\mathcal{I}$  do  
4     calculate hash value  $h \leftarrow h(r.A)$   
5     add  $r$  into bucket  $h$  in  $\mathcal{H}$ 
```

Algorithm

Probe phase

```
1 foreach block  $S$  from  $\mathcal{S}$  do
2   read  $S$  into  $\mathcal{I}$ 
3   foreach item  $s$  in  $\mathcal{I}$  do
4     calculate hash value  $h \leftarrow h(s.A)$ 
5     foreach item  $r$  in bucket  $h$  in  $\mathcal{H}$  do
6       if  $r$  and  $s$  can be joined then
7         join  $r$  and  $s$  and put the result to  $\mathcal{O}$ 
8         if  $\mathcal{O}$  is full then write  $\mathcal{O}$  to  $\mathcal{T}$  and empty  $\mathcal{O}$ 
9 if  $\mathcal{O}$  is not empty then write  $\mathcal{O}$  to  $\mathcal{T}$  and empty  $\mathcal{O}$ 
```

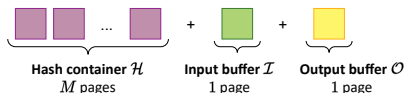
Observations

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
 - $c_{\text{build}} = p_R$
 - $c_{\text{probe}} = p_S$
- **Overall cost**
 - $c_{\text{CH}} = c_{\text{build}} + c_{\text{probe}} = p_R + p_S$

Summary

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

Simple Hashing

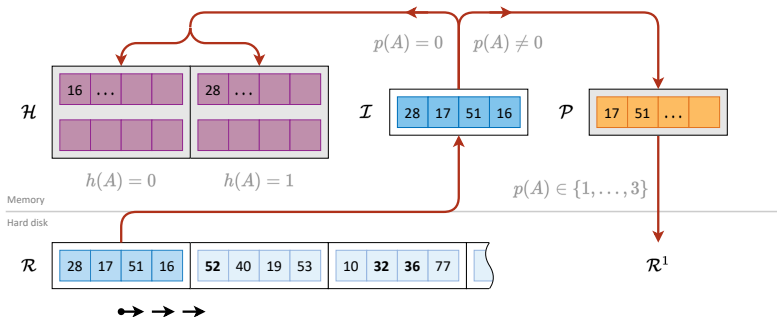
Simple hash join

- Basic idea
 - During each pass, **just a subset of all tuples is considered**
 - These are processed via analogous **build** and **probe** routines
 - The **remaining tuples are postponed** for the following passes
- **Partition function** p is assumed for this separation
 - Its **domain** are again values of the joining attribute A
 - Its **range** provides P distinct values
- Obvious requirement
 - **Both functions** p and h need to be **mutually orthogonal**
 - E.g.: $p(A) = A \bmod 4$ and $h(A) = A \bmod 2$ will not work
 - Because all items in a partition would either be even or odd

Build Phase

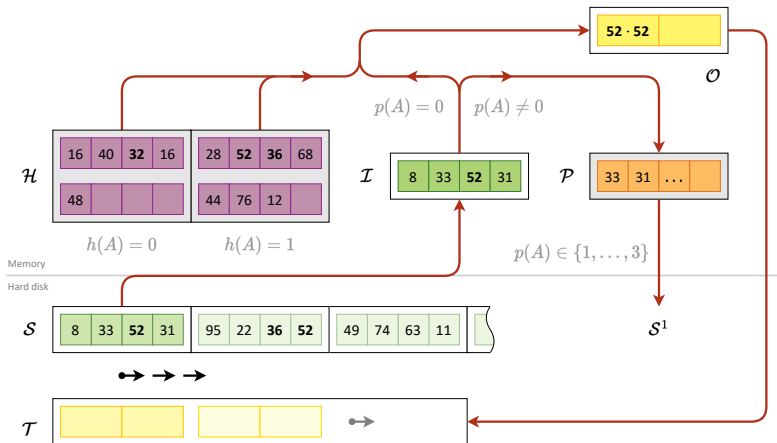
Build phase (partition 0)

- Items from the smaller table are either **hashed** or **postponed**
 - E.g., partition function $p(A) = A \bmod 4$ and hash function $h(A) = (A/4) \bmod 2$ are assumed



Probe Phase

Probe phase (partition 0)



Algorithm

Overall procedure

```
1 put  $\mathcal{R}^0 \leftarrow \mathcal{R}$ 
2 put  $\mathcal{S}^0 \leftarrow \mathcal{S}$ 
3 foreach partition  $p \in \{0, \dots, P-1\}$  do
4   execute build phase for partition  $p$  over  $\mathcal{R}^p$  and create
   postponed  $\mathcal{R}^{p+1}$ 
5   execute probe phase for partition  $p$  over  $\mathcal{S}^p$  and create
   postponed  $\mathcal{S}^{p+1}$ 
6   empty hash container  $\mathcal{H}$ 
```

Algorithm

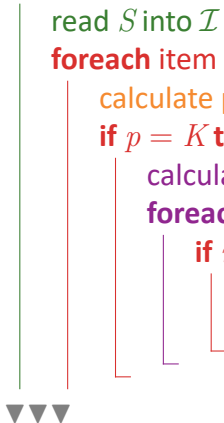
Build phase (for partition K)

```
1  foreach block  $R$  from  $\mathcal{R}^K$  do
2    read  $R$  into  $\mathcal{I}$ 
3    foreach item  $r$  in  $\mathcal{I}$  do
4      calculate partition value  $p \leftarrow p(r.A)$ 
5      if  $p = K$  then
6        calculate hash value  $h \leftarrow h(r.A)$ 
7        add  $r$  into bucket  $h$  in  $\mathcal{H}$ 
8      else
9        add  $r$  into partition buffer  $\mathcal{P}$ 
10       if  $\mathcal{P}$  is full then write  $\mathcal{P}$  to  $\mathcal{R}^{K+1}$  and empty  $\mathcal{P}$ 
11 if  $\mathcal{P}$  is not empty then write  $\mathcal{P}$  to  $\mathcal{R}^{K+1}$  and empty  $\mathcal{P}$ 
```

Algorithm

Probe phase (for partition K)

```
1  foreach block  $S$  from  $\mathcal{S}^K$  do
2      read  $S$  into  $\mathcal{I}$ 
3      foreach item  $s$  in  $\mathcal{I}$  do
4          calculate partition value  $p \leftarrow p(s.A)$ 
5          if  $p = K$  then
6              calculate hash value  $h \leftarrow h(s.A)$ 
7              foreach item  $r$  in bucket  $h$  in  $\mathcal{H}$  do
8                  if  $r$  and  $s$  can be joined then
9                      join  $r$  and  $s$  and put the result to  $\mathcal{O}$ 
10                     if  $\mathcal{O}$  is full then write  $\mathcal{O}$  to  $\mathcal{T}$ , empty  $\mathcal{O}$ 
```



The diagram illustrates the nested loop structure of the algorithm. It features four vertical lines of different colors: green, red, purple, and yellow. The green line starts at line 1 and ends at line 10, with a downward arrow at the bottom. The red line starts at line 3 and ends at line 10, with a downward arrow at the bottom. The purple line starts at line 5 and ends at line 10, with a downward arrow at the bottom. The yellow line starts at line 8 and ends at line 10, with a downward arrow at the bottom. These lines represent the execution flow of the nested loops.

Algorithm

Probe phase (for partition K) (cont'd)

```

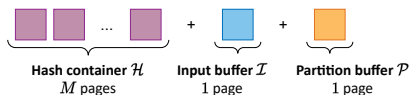
11   else
12       add  $s$  into partition buffer  $\mathcal{P}$ 
13       if  $\mathcal{P}$  is full then write  $\mathcal{P}$  to  $\mathcal{S}^{K+1}$  and empty  $\mathcal{P}$ 
14   if  $\mathcal{O}$  is not empty then write  $\mathcal{O}$  to  $\mathcal{T}$  and empty  $\mathcal{O}$ 
15   if  $\mathcal{P}$  is not empty then write  $\mathcal{P}$  to  $\mathcal{S}^{K+1}$  and empty  $\mathcal{P}$ 

```

Observations

Memory layout

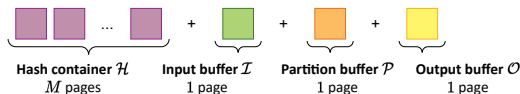
- Build phase: $M + 1 + 1$
 - **Hash container $\mathcal{H}$** : $M = \lceil F \cdot (p_R/P) \rceil$ pages
 - **Input buffer $\mathcal{I}$** : 1 page
 - **Partition buffer $\mathcal{P}$** : 1 page



Observations

Memory layout

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



Observations

Time complexity

- Build and probe phases

$$\begin{aligned}c_{\text{build}} &\approx \left(p_R + \frac{P-1}{P}p_R\right) + \left(\frac{P-1}{P}p_R + \frac{P-2}{P}p_R\right) + \cdots + \left(\frac{1}{P}p_R\right) \\&= p_R + 2\frac{1}{P}\left[(P-1) + (P-2) + \cdots + (1)\right]p_R \\&= p_R + 2\frac{1}{P}\left[\frac{(P-1)+(1)}{2} \cdot (P-1)\right]p_R = p_R + (P-1)p_R \\&= P \cdot p_R\end{aligned}$$

- Analogously $c_{\text{probe}} = P \cdot p_S$

- Overall cost

- $c_{\text{SH}} = c_{\text{build}} + c_{\text{probe}} = P \cdot (p_R + p_S)$

Summary

- We are now able to deal even with larger tables
 - However, overall cost is still not efficient enough...

Partition Hashing

Partition hash join

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

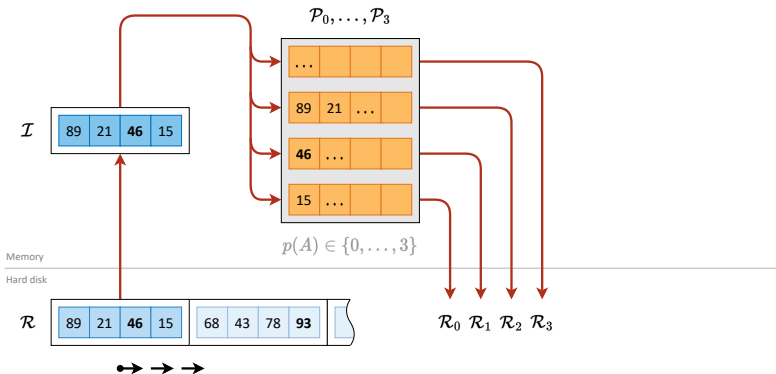
Overall procedure

-
- 1 split $\mathcal{R}$ and create partitions $\mathcal{R}_0, \dots, \mathcal{R}_{P-1}$
 - 2 split $\mathcal{S}$ and create partitions $\mathcal{S}_0, \dots, \mathcal{S}_{P-1}$
 - 3 **foreach** partition $p \in \{0, \dots, P-1\}$ **do**
 - 4 $\lfloor$ join partitions $\mathcal{R}_p$ and $\mathcal{S}_p$
-

Partition Phase

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

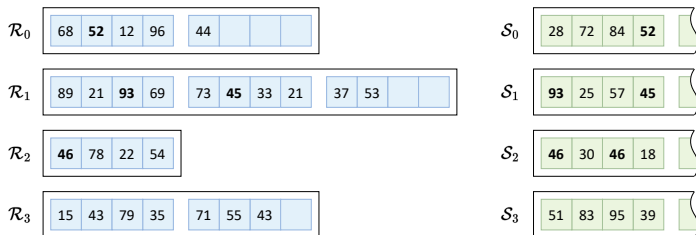
- Tuples of a given table are split to **disjoint partitions**



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

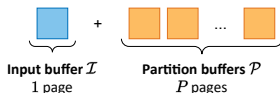
- Table $\mathcal{R}$ is assumed, partitioning of $\mathcal{S}$ is analogous

```
1 foreach block  $R$  from  $\mathcal{R}$  do
2   read  $R$  into  $\mathcal{I}$ 
3   foreach item  $r$  in  $\mathcal{I}$  do
4     calculate partition value  $p \leftarrow p(r.A)$ 
5     add  $r$  into partition buffer  $\mathcal{P}_p$ 
6     if  $\mathcal{P}_p$  is full then write  $\mathcal{P}_p$  to  $\mathcal{R}_p$  and empty  $\mathcal{P}_p$ 
7 foreach partition  $p \in \{0, \dots, P-1\}$  do
8   if  $\mathcal{P}_p$  is not empty then write  $\mathcal{P}_p$  to  $\mathcal{R}_p$  and empty  $\mathcal{P}_p$ 
```

Observations

Memory layout

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



Time complexity

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

Grace Hashing

Grace hash join

- Just ordinary **partition hash join**
 - ... with **balanced memory** requirements across all the phases

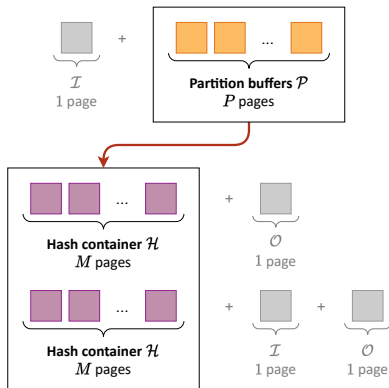
Memory setup

- Let $m \approx \sqrt{F \cdot p_R}$
 - I.e., square root of the size of an in-memory container that would roughly be needed for hashing of the smaller table $\mathcal{R}$
- **Partition function** p is chosen to ensure that $P = m$
 - $\Rightarrow m$ partitions will be created (for $\mathcal{R}$ as well as $\mathcal{S}$)
 - $\Rightarrow$ **expected size of each partition** of $\mathcal{R}$ should be...
 - $s = p_R / P = p_R / m = p_R / \sqrt{F \cdot p_R} \approx \sqrt{p_R / F}$ pages
 - $\Rightarrow$ **space needed for hashing** each of these partitions...
 - $F \cdot s = F \cdot \sqrt{p_R / F} \approx \sqrt{F \cdot p_R} \approx m$ pages

Grace Hashing

Memory setup (cont'd)

- I.e., size P of **partition buffers** $\mathcal{P}$ (partition phase) and size M of **hash container** $\mathcal{H}$ (build and probe phases) are equal to m



Hybrid Hashing

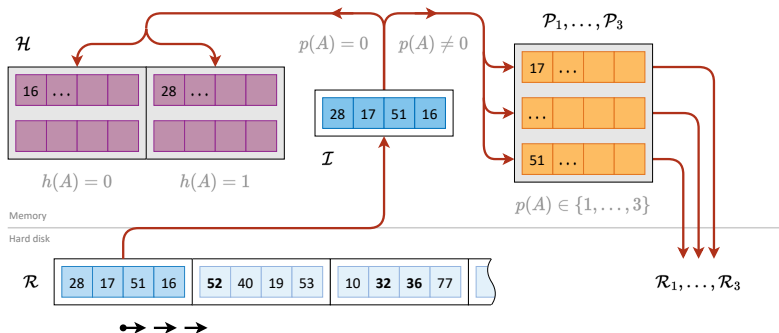
Hybrid hash join

- Basically an improvement of the **simple hash join** approach
 - Instead of using just one buffer for all items to be postponed...
 - ... we directly split them to **separate partitions**
 - I.e., as in the **partition hash join** approach
- In other words...
 - **Partitions** 0 are joined directly during the first pass
 - Using the altered build and probe phases
 - All the **remaining partitions** are pairwise joined subsequently
 - Using the **classic hash join** approach

Build Phase

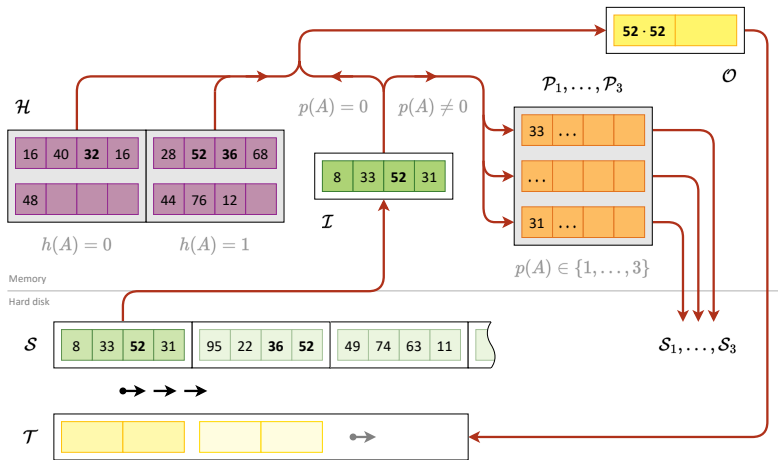
Build phase

- Items from the smaller table are either **hashed** or **postponed**
 - However, when they are to be postponed, they are branched to individual separated partitions



Probe Phase

Probe phase



Algorithm

Overall procedure

-
- 1 execute **build phase** over $\mathcal{R}$, hash items from partition 0
and create postponed partitions $\mathcal{R}_1, \dots, \mathcal{R}_{P-1}$
 - 2 execute **probe phase** over $\mathcal{S}$, join items from partition 0
and create postponed partitions $\mathcal{S}_1, \dots, \mathcal{S}_{P-1}$
 - 3 **foreach** partition $p \in \{1, \dots, P-1\}$ **do**
 - 4 └ join partitions $\mathcal{R}_p$ and $\mathcal{S}_p$
-

Algorithm

Build phase

```
1 foreach block  $R$  from  $\mathcal{R}$  do
2   read  $R$  into  $\mathcal{I}$ 
3   foreach item  $r$  in  $\mathcal{I}$  do
4     calculate partition value  $p \leftarrow p(r.A)$ 
5     if  $p = 0$  then
6       calculate hash value  $h \leftarrow h(r.A)$ 
7       add  $r$  into bucket  $h$  in  $\mathcal{H}$ 
8     else
9       add  $r$  into partition buffer  $\mathcal{P}_p$ 
10      if  $\mathcal{P}_p$  is full then write  $\mathcal{P}_p$  to  $\mathcal{R}_p$  and empty  $\mathcal{P}_p$ 
```



Algorithm

Build phase (cont'd)



- 11 **foreach** partition $p \in \{1, \dots, P - 1\}$ **do**
 - 12 **if** $\mathcal{P}_p$ is not empty **then** write $\mathcal{P}_p$ to $\mathcal{R}_p$ and empty $\mathcal{P}_p$
-

Algorithm

Probe phase

```
1 foreach block  $S$  from  $\mathcal{S}$  do
2   read  $S$  into  $\mathcal{I}$ 
3   foreach item  $s$  in  $\mathcal{I}$  do
4     calculate partition value  $p \leftarrow p(s.A)$ 
5     if  $p = 0$  then
6       calculate hash value  $h \leftarrow h(s.A)$ 
7       foreach item  $r$  in bucket  $h$  in  $\mathcal{H}$  do
8         if  $r$  and  $s$  can be joined then
9           join  $r$  and  $s$  and put the result to  $\mathcal{O}$ 
10          if  $\mathcal{O}$  is full then write  $\mathcal{O}$  to  $\mathcal{T}$ , empty  $\mathcal{O}$ 
```

▼▼▼

Algorithm

Probe phase (cont'd)

▲▲▲

```
11 |   | else
12 |   |   add  $s$  into partition buffer  $\mathcal{P}_p$ 
13 |   |   if  $\mathcal{P}_p$  is full then write  $\mathcal{P}_p$  to  $\mathcal{S}_p$  and empty  $\mathcal{P}_p$ 
    |   |
14 |   | if  $\mathcal{O}$  is not empty then write  $\mathcal{O}$  to  $\mathcal{T}$  and empty  $\mathcal{O}$ 
15 |   | foreach partition  $p \in \{1, \dots, P-1\}$  do
16 |   |   if  $\mathcal{P}_p$  is not empty then write  $\mathcal{P}_p$  to  $\mathcal{S}_p$  and empty  $\mathcal{P}_p$ 
```

Observations

Memory layout

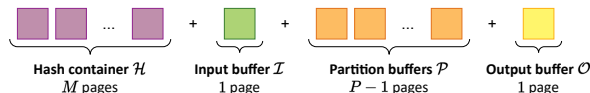
- Build phase: $M + 1 + (P - 1)$
 - **Hash container $\mathcal{H}$** : $M = \lceil F \cdot (p_R/P) \rceil$ pages
 - **Input buffer $\mathcal{I}$** : 1 page
 - **Partition buffers $\mathcal{P}$** : $P - 1$ pages



Observations

Memory layout

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



Observations

Time complexity

- Build and probe phases for partition 0
 - $c_{\text{build}} \approx p_R + p_R \cdot \frac{P-1}{P} = p_R \cdot (1 + \frac{P-1}{P}) = p_R \cdot (2 - \frac{1}{P})$
 - Analogously $c_{\text{probe}} \approx p_S \cdot (2 - \frac{1}{P})$
- **Overall cost** (with classic hash join involved)
 - $$\begin{aligned} c_{\text{HH}} &= c_{\text{build}} + c_{\text{probe}} + (P-1) \cdot c_{\text{CH}} \\ &\approx p_R \cdot (2 - \frac{1}{P}) + p_S \cdot (2 - \frac{1}{P}) + (P-1) \left[\frac{p_R}{P} + \frac{p_S}{P} \right] \\ &\approx (3 - \frac{2}{P}) \cdot (p_R + p_S) \end{aligned}$$

Query Evaluation

Sample Query

Database schema

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

Sample query

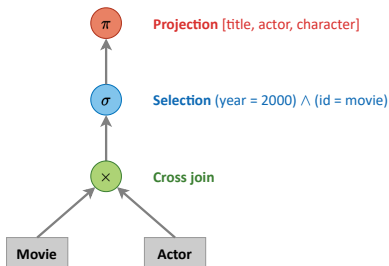
- **Actors and characters they played in movies filmed in 2000**
 - `SELECT title, actor, character`
`FROM Movie JOIN Actor`
`WHERE (year = 2000) AND (id = movie)`
 - `(Movie  $\times$  Actor)((year = 2000)  $\wedge$  (id = movie))`
`[title, actor, character]`
 - $\pi_{\text{title, actor, character}} \left(\sigma_{(\text{year}=2000) \wedge (\text{id}=\text{movie})} (\text{Movie} \times \text{Actor}) \right)$

Sample Query

Sample query (cont'd)

- Actors and characters they played in movies filmed in 2000

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



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

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

$$M_1 = 25 + 1 + 1 = 27$$

$$c_1^r = p_M + (p_M / 25) \cdot p_A = 10\,010\,000$$

$$c_1^w = p_1 = 12\,500\,000\,000$$

Sorted file (year)

$$n_M = 100\,000$$

$$b_M = 10$$

$$p_M = 10\,000$$

$$V_{M,year} = 50$$

B⁺ tree index (year)

$$m_{M,year} = 100$$

$$I_{M,year} = 3$$

Projection [title, actor, character]

$$n_3 = n_2 = 20\,000$$

$$b_3 \leftarrow 50$$

$$p_3 = n_3 / b_3 = 400$$

$$c_3^r = p_2 = 2\,500$$

$$c_3^w = p_3 = 400$$

Selection (year = 2000) ∧ (id = movie)

$$n_2 = n_1 \cdot (1 / V_{M,year}) \cdot (1 / n_M) = 20\,000$$

$$b_2 = b_1 = 8$$

$$p_2 = n_2 / b_2 = 2\,500$$

$$c_2^r = p_1 = 12\,500\,000\,000$$

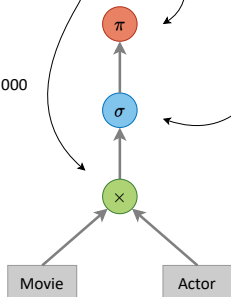
$$c_2^w = p_2 = 2\,500$$

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
- $c = [c_1^r + c_1^w] + [c_2^r + c_2^w] + [c_3^r]$
- $c = [p_M + (p_M/25) \cdot p_A + p_1] + [p_1 + p_2] + [p_2]$
- $c = [10\ 010\ 000 + 12\ 500\ 000\ 000] + [12\ 500\ 000\ 000 + 2\ 500] + [2\ 500]$
- $c = 25\ 010\ 015\ 000$

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

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

Sample Plan #2

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

$$M_1 = 25 + 1 + 1 = 27$$

$$c_1^r = p_M + (p_M / 25) \cdot p_A = 10\,010\,000$$

$$c_1^w = p_1 = 125\,000$$

Sorted file (year)

$$n_M = 100\,000$$

$$b_M = 10$$

$$p_M = 10\,000$$

$$V_{M.year} = 50$$

B⁺ tree index (year)

$$m_{M.year} = 100$$

$$I_{M.year} = 3$$

Projection [title, actor, character]

$$n_3 = n_2 = 20\,000$$

$$b_3 \leftarrow 50$$

$$p_3 = n_3 / b_3 = 400$$

$$c_3^r = p_2 = 2\,500$$

$$c_3^w = p_3 = 400$$

Selection (year = 2000)

$$n_2 = n_1 \cdot (1 / V_{M.year}) = 20\,000$$

$$b_2 = b_1 = 8$$

$$p_2 = n_2 / b_2 = 2\,500$$

$$c_2^r = p_1 = 125\,000$$

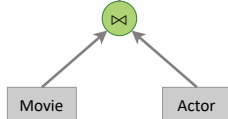
$$c_2^w = p_2 = 2\,500$$

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

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**

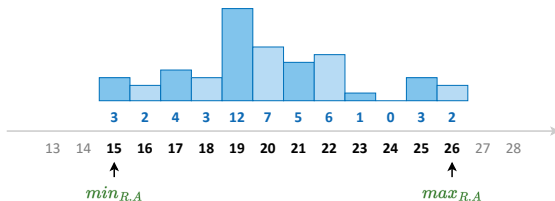
Histograms

Histogram = approximate representation of data distribution

- Active domain is split into **sub-intervals** called **buckets**
 - Usually consecutive and non-overlapping
- **Frequency of values** is determined for each one of them
 - I.e., count of values that fall into that bucket

Sample data

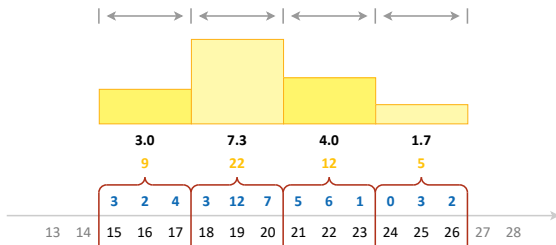
- Integer values from interval $[15, 26]$ and their frequencies



Histograms

Equi-width histogram

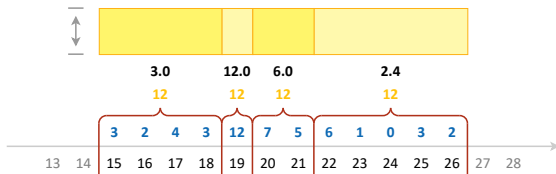
- Buckets have **equal widths** (count of distinct values)
- **Discrete** domains: average **frequencies** are stored
 - So that frequency $f_{E.A}(v)$ can be retrieved for any value v
- **Continuous** domains: **probabilities** are stored instead
 - So that probability $t_{E.A}(b)$ can be retrieved for any bucket b



Histograms

Equi-depth histogram

- **Buckets** are designed so that they have **equal depths**
 - I.e., absolute frequencies are the same
 - Or at least almost the same
 - Since real-world data will likely not be nice enough
- We also need to explicitly store **bucket placement** information
 - Since it is not derivable automatically



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: Selection

Reduction factors

- **Equality test** with respect to a **unique attribute**
 - $r_\varphi = 1/n_E$ (and so $n_T = 1$)
- **Equality test** with respect to a **non-unique attribute**
 - $r_\varphi = 1/V_{E.A}$
 - $r_\varphi = f_{E.A}(v)/n_E$ if histogram for discrete domains is available
 - As a consequence, $n_T = f_{E.A}(v)$
 - $r_\varphi = t_{E.A}(\text{bucket}(v))$ analogously for continuous domains
 - $r_\varphi = 1/10$ when no information is available at all
- **Estimates using constants** in general
 - May work well, not bad, as well as totally wrong...
 - But when nothing better is available, it must simply suffice
 - Of course, particular constant is just a matter of discussion

Size Estimates: Selection

Reduction factors (cont'd)

- **Range query** for two-sided intervals $I = [v_1, v_2]$ and other
 - $r_\varphi = (v_2 - v_1 + \varepsilon) / (\max_{E.A} - \min_{E.A} + 1)$
 - $r_\varphi = (\sum_{v \in I} f_{E.A}(v)) / n_E$
 - $r_\varphi = (v_2 - v_1) / (\max_{E.A} - \min_{E.A})$
 - $r_\varphi = \sum_{b \in \text{buckets}(I)} t_{E.A}(b)$
 - $r_\varphi = 1/4$
- **Range query** for one-sided intervals $(-\infty, v_2]$ and $(-\infty, v_2)$
 - Works analogously...
 - $r_\varphi = 1/2$
 - Unfortunately, there are certain **undesired consequences**...
 - E.g., reduction factors of $A \leq 1$ and $A \leq 1000$ are the same
- **Range query** for one-sided intervals $[v_1, \infty)$ and (v_1, ∞)
 - Works analogously again...

Size Estimates: Selection

Reduction factors (cont'd)

- **Conjunction:** $\varphi_1 \wedge \varphi_2$
 - $r_\varphi = r_{\varphi_1} \cdot r_{\varphi_2}$
 - **Independence** of both conditions is assumed
- **Disjunction:** $\varphi_1 \vee \varphi_2$
 - $r_\varphi = r_{\varphi_1} + r_{\varphi_2} - r_{\varphi_1} \cdot r_{\varphi_2}$
- **Negation:** $\neg \varphi_1$
 - $r_\varphi = 1 - r_{\varphi_1}$
- ...

Improved estimates might also be useful for **access methods**

- Since it is also about selection
 - However, technical possibilities of data files must be respected

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

Size Estimates: Joins

Reduction factors

- **Cross join**
 - $r_\varphi = 1$ (hence $n_T = n_R \cdot n_S$)
- **Foreign key lookup**
 - Let us assume that φ traverses a foreign key from $\mathcal{R}$ to $\mathcal{S}$
 - Then for each tuple $r \in \mathcal{R}$ there must exist exactly one $s \in \mathcal{S}$
 - And so $r_\varphi = 1/n_S$ (hence $n_T = n_R$)
- **Equality test over an attribute A in $\mathcal{S}$**
 - $r_\varphi = 1/V_{S.A}$
 - $r_\varphi = 1/n_S$ specifically for a **unique attribute** (again $n_T = n_R$)
- ...

Algebraic Optimization

Equivalence Rules: Selection

Commutativity of selection

- $\sigma_{\varphi_2}(\sigma_{\varphi_1}(E)) \equiv \sigma_{\varphi_1}(\sigma_{\varphi_2}(E))$
- **Mutual order of selections** can be changed
 - Condition with higher **selectivity** can be applied first
 - I.e., condition which yields a fewer number of tuples

Cascade of selections

- $\sigma_{\varphi_2}(\sigma_{\varphi_1}(E)) \equiv \sigma_{\varphi_1 \wedge \varphi_2}(E)$
- Direction $\rightarrow$
 - **Selections can be merged together** into just one
 - Via a conjunction over the original conditions
- Direction $\leftarrow$
 - **Conjunctive selection can be split** into separate selections

Equivalence Rules: Projection

Cascade of projections

- $\pi_{A_2}(\pi_{A_1}(E)) \equiv \pi_{A_2}(E)$
- $\rightarrow$: only the **outermost projection** actually matters
 - And so the inner one can entirely be omitted as meaningless

Commutativity of selection and projection

- $\pi_A(\sigma_\varphi(E)) \equiv \sigma_\varphi(\pi_A(E))$
- **Selection and projection can be mutually swapped**
 - $\leftarrow$: without any limitation
 - $\rightarrow$: only when all attributes in φ are still available
 - When this assumption is not satisfied...
- $\pi_A(\sigma_\varphi(E)) \equiv \pi_A(\sigma_\varphi(\pi_{A \cup S}(E)))$
 - Attributes S from E are those that are needed for the selection

Equivalence Rules: Joins

Commutativity of joins

- Cross join: $E_1 \times E_2 \equiv E_2 \times E_1$
- Natural join: $E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$
- Theta join: $E_1 \bowtie_{\varphi} E_2 \equiv E_2 \bowtie_{\varphi} E_1$
- **Operands of inner joins** can be mutually swapped
 - Such a thing is not possible for outer joins

Associativity of joins

- Inner joins are also associative (again, not outer)
- $(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$
- $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$
- $(E_1 \bowtie_{\varphi_{12}} E_2) \bowtie_{\varphi_{13} \wedge \varphi_{23}} E_3 \equiv E_1 \bowtie_{\varphi_{12} \wedge \varphi_{13}} (E_2 \bowtie_{\varphi_{23}} E_3)$
 - Assuming that each φ_{ij} only involves attributes from E_i and E_j

Equivalence Rules: Joins

Integration of selection into joins

- **Any inner join can be rewritten using theta join...**
- ... and then combined with selection
 - Intended for **conditions of joining nature**
 - I.e., conditions that involve attributes from both the operands
- $\sigma_{\varphi_S}(E_1 \times E_2) \equiv E_1 \bowtie_{\varphi_S} E_2$
- $\sigma_{\varphi_S}(E_1 \bowtie_{\varphi_J} E_2) \equiv E_1 \bowtie_{\varphi_J \wedge \varphi_S} E_2$
- $\sigma_{\varphi_S}(E_1 \bowtie E_2) \equiv E_1 \bowtie_{\varphi_N \wedge \varphi_S} E_2$
 - φ_N involves pairwise equality tests for all the shared attributes
 - I.e., attributes occurring in both the operands

Equivalence Rules: Joins

Distribution of selection over joins

- Let us have an inner join wrapped by a selection...
 - ... and this selection contains a **condition of filtering nature**
 - I.e., condition with attributes from just one join operand
- It can then be **executed before the join** over just that operand
 - And so the join evaluation cost can be decreased
- $\sigma_{\varphi_S}(E_1 \times E_2) \equiv \sigma_{\varphi_S}(E_1) \times E_2$
 - Assuming that, in particular, φ_S involves attributes from E_1 only
- $\sigma_{\varphi_S}(E_1 \bowtie E_2) \equiv \sigma_{\varphi_S}(E_1) \bowtie E_2$
- $\sigma_{\varphi_S}(E_1 \bowtie_{\varphi_J} E_2) \equiv \sigma_{\varphi_S}(E_1) \bowtie_{\varphi_J} E_2$

Equivalence Rules: Joins

Distribution of projection over joins

- Let us assume that attributes A_1 are from E_1 and A_2 from E_2
- $\pi_{A_1 \cup A_2}(E_1 \times E_2) \equiv \pi_{A_1}(E_1) \times \pi_{A_2}(E_2)$
- $\pi_{A_1 \cup A_2}(E_1 \bowtie E_2) \equiv \pi_{A_1}(E_1) \bowtie \pi_{A_2}(E_2)$
 - $\rightarrow$: only works when all joining attributes are still available
- $\pi_{A_1 \cup A_2}(E_1 \bowtie E_2) \equiv \pi_{A_1 \cup A_2}(\pi_{A_1 \cup N}(E_1) \bowtie \pi_{A_2 \cup N}(E_2))$
 - Attributes N are those that are needed for the natural join
 - Despite looking strange, the impact may be significant
 - Since unnecessary attributes are removed earlier
- $\pi_{A_1 \cup A_2}(E_1 \bowtie_{\varphi} E_2) \equiv \pi_{A_1}(E_1) \bowtie_{\varphi} \pi_{A_2}(E_2)$
 - $\rightarrow$: analogous assumption again
- $\pi_{A_1 \cup A_2}(E_1 \bowtie_{\varphi} E_2) \equiv \pi_{A_1 \cup A_2}(\pi_{A_1 \cup J_1}(E_1) \bowtie_{\varphi} \pi_{A_2 \cup J_2}(E_2))$
 - Attributes J_i from E_i are those needed for the theta join

Equivalence Rules: Set Operations

Commutativity of set operations

- $E_1 \cup E_2 \equiv E_2 \cup E_1$
- $E_1 \cap E_2 \equiv E_2 \cap E_1$
- Set difference is not commutative

Associativity of set operations

- $(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$
- $(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$
- Set difference is also not associative

Equivalence Rules: Set Operations

Distribution of selection over set operations

- $\sigma_{\varphi}(E_1 \cup E_2) \equiv \sigma_{\varphi}(E_1) \cup \sigma_{\varphi}(E_2)$
- $\sigma_{\varphi}(E_1 \cap E_2) \equiv \sigma_{\varphi}(E_1) \cap \sigma_{\varphi}(E_2)$
- $\sigma_{\varphi}(E_1 \setminus E_2) \equiv \sigma_{\varphi}(E_1) \setminus \sigma_{\varphi}(E_2)$

Distribution of projection over set operations

- $\pi_A(E_1 \cup E_2) \equiv \pi_A(E_1) \cup \pi_A(E_2)$
- Such a thing is not possible for intersection and difference

Recommendations

Basic heuristics

- **Push filtering selections** as close as possible to leaves
 - To throw away **not needed tuples** as soon as possible
- **Push projections** toward leaves the same way
 - So that **size of intermediate results** is decreased
- **Integrate joining selections** into joins
 - I.e, **rewrite other types of joins to theta joins**
- **Simplify cascades** of projections or selections
- **Transform sub-queries to joins** whenever possible
 - Since optimization only works for simple SELECT blocks
- **Exploit commutativity and associativity** of operations
 - Especially **joins** but also set operations

Examples

Sample transformations

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

Algebraic Optimization

Objective

- Capability of **finding alternative query evaluation plans**
 - Based on the so far introduced **equivalence rules**
 - As well as other not covered rules and heuristics
- 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 / **multi-relation** plans
- But still an **NP-complete problem**

Alternative Plans

Single-relation plans

- Finding the **best access method** for each **individual table**
 - Including optional filtering **selections** and **projections**

Multi-relation plans

- Finding the **best join plan** for a given set of tables
 - Only **binary joins** are usually assumed
 - And so we just need to take into account **all possible orderings**
 - Since inner joins are commutative and associative

Observation

- **Optimal plan** may not consist of **optimal sub-plans**
 - And so it may happen that the truly best plan will not be found

Algorithm

Basic top-down approach

- Finding the best plan for a set of relations S
 - Using a **dynamic programming** method

```
1  if the best plan for  $S$  is already calculated then
2      |    $\mathcal{P} \leftarrow$  fetch the best plan for  $S$ 
3      |   return  $\mathcal{P}$ 
4  else
5      |   if  $S$  contains just a single relation  $\mathcal{R}$  then
6      |       |    $\mathcal{P} \leftarrow$  find the best access method for  $\mathcal{R}$ 
7      |       |   store  $\mathcal{P}$  as the best plan for  $S$ 
8      |       |   return  $\mathcal{P}$ 
```



Algorithm

Basic top-down approach (cont'd)

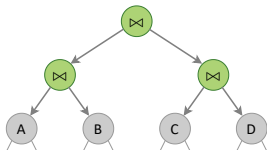


```
9  else
10      foreach  $S_L \subseteq S$  such that  $S_L \neq \emptyset \wedge S_L \neq S$  do
11           $\mathcal{P}_L \leftarrow$  recursively find the best plan for  $S_L$ 
12           $\mathcal{P}_R \leftarrow$  recursively find the best plan for  $S \setminus S_L$ 
13           $\mathcal{P} \leftarrow$  find the best join plan over  $\mathcal{P}_L$  and  $\mathcal{P}_R$ 
14          if  $\mathcal{P}$  is so far the best plan for  $S$  (if any) then
15              store  $\mathcal{P}$  as the best plan for  $S$ 
16           $\mathcal{P} \leftarrow$  fetch the best plan for  $S$ 
17      return  $\mathcal{P}$ 
```

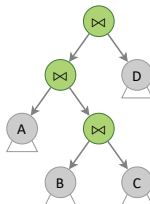
Left-Deep Linear Trees

Only left-deep linear trees are usually taken into account...

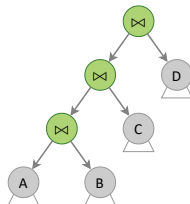
- **Linear tree**
 - Each non-leaf node must have at least one child with relation
- **Left-deep linear tree**
 - Moreover, that child must be the right-hand one
 - Since that also increases the chance of attainable **pipelining**



$((A \bowtie B) \bowtie (C \bowtie D))$



$((A \bowtie (B \bowtie C)) \bowtie D)$



$(((A \bowtie B) \bowtie C) \bowtie D)$

Algorithm

Restricted top-down approach

- For left-deep linear trees only
 - This means there will be just $O(n \cdot 2^n)$ instead of $O(3^n)$ plans

```
1 if the best plan for  $S$  is already calculated then
2   |  $\mathcal{P} \leftarrow$  fetch the best plan for  $S$ 
3   | return  $\mathcal{P}$ 
4 else
5   | if  $S$  contains just a single relation  $\mathcal{R}$  then
6   |   |  $\mathcal{P} \leftarrow$  find the best access method for  $\mathcal{R}$ 
7   |   | store  $\mathcal{P}$  as the best plan for  $S$ 
8   |   | return  $\mathcal{P}$ 
```



Algorithm

Restricted top-down approach (cont'd)



```
9  else
10    foreach single relation  $\mathcal{R} \in S$  do
11       $\mathcal{P}_L \leftarrow$  recursively find the best plan for  $S \setminus \{\mathcal{R}\}$ 
12       $\mathcal{P}_R \leftarrow$  recursively find the best plan for  $\{\mathcal{R}\}$ 
13       $\mathcal{P} \leftarrow$  find the best join plan over  $\mathcal{P}_L$  and  $\mathcal{P}_R$ 
14      if  $\mathcal{P}$  is so far the best plan for  $S$  (if any) then
15        store  $\mathcal{P}$  as the best plan for  $S$ 
16       $\mathcal{P} \leftarrow$  fetch the best plan for  $S$ 
17    return  $\mathcal{P}$ 
```

Algorithm

Restricted bottom-up approach

- We proceed by induction on the number of relations
 - All **single-relation plans** are found first
 - Then gradually all **multi-relation plans**
 - The best plan for n relations is found by considering all possible means of joining any of its $n - 1$ relations with the 1 remaining

```
1 foreach single relation  $\mathcal{R} \in S$  do
2    $\mathcal{P} \leftarrow$  find the best access method for  $\mathcal{R}$ 
3   store  $\mathcal{P}$  as the best plan for  $\{\mathcal{R}\}$ 
```



Algorithm

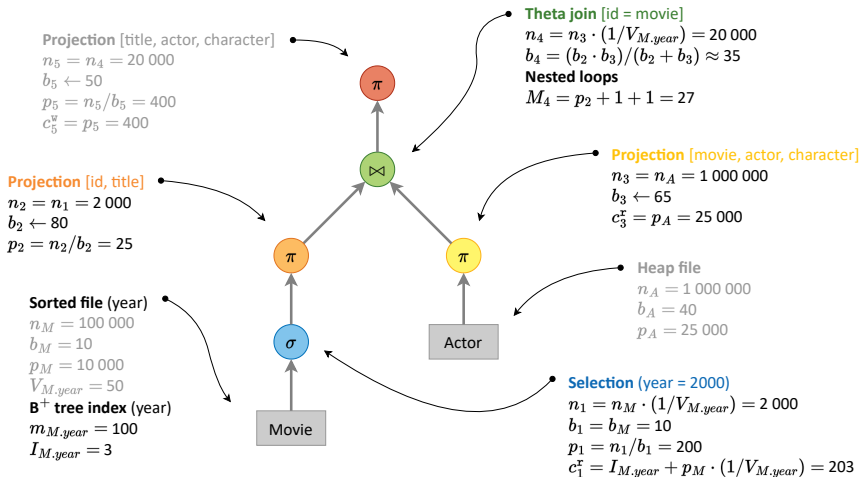
Restricted bottom-up approach (cont'd)



```
4  foreach pass  $p \in \{2, \dots, |S|\}$  do
5      foreach  $T \subseteq S$  such that  $|T| = p$  do
6          foreach single relation  $\mathcal{R} \in T$  do
7               $\mathcal{P}_L \leftarrow$  fetch the best plan for  $T \setminus \{\mathcal{R}\}$ 
8               $\mathcal{P}_R \leftarrow$  fetch the best plan for  $\{\mathcal{R}\}$ 
9               $\mathcal{P} \leftarrow$  find the best join plan over  $\mathcal{P}_L$  and  $\mathcal{P}_R$ 
10             if  $\mathcal{P}$  is so far the best plan for  $T$  (if any) then
11                 store  $\mathcal{P}$  as the best plan for  $T$ 
12  $\mathcal{P} \leftarrow$  fetch the best plan for  $S$ 
13 return  $\mathcal{P}$ 
```

Query Evaluation

Sample Plan #3



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
- $c = [\cancel{c_1^f} + \cancel{c_1^f}] + [\cancel{c_2^f} + \cancel{c_2^f}] + [\cancel{c_3^f} + \cancel{c_3^f}] + [\cancel{c_4^f} + \cancel{c_4^f}] + [\cancel{c_5^f}]$
 - $\mathcal{I}_2$ 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

- **Size of tuples**
 - Real-world **tuples usually have variable size**
 - Because data types such as VARCHAR are often used
 - That complicates internal block structure and cost estimates
- **Unused slots**
 - **Not all slots** within data file blocks **may really be used**
 - I.e., there can be gaps because of, e.g., deleted tuples
 - And so the actual file size may be greater than assumed
- **Inner fragmentation**
 - It may not be possible to utilize **inner block space** entirely
 - I.e., there can be **unused space** after the last slot
 - Or even around the slots in case of variable-size tuples

Observations

False assumptions and simplifications (cont'd)

- **Overflow areas** in **sorted files**
 - New tuples are usually not inserted to their correct positions
 - Instead, special dedicated area is used for that purpose
 - So that **time-complicated insertion** (up to linear) is avoided
 - Only time to time the whole file is reorganized (resorted)
- **Overflow areas** in **hashed files**
 - Allocated size of buckets may not be sufficient
- **Outer fragmentation**
 - **Layout of file blocks on a hard drive** may not be continuous
 - That may significantly increase time costs
 - Because of repeated seeks and rotational delays

Observations

False assumptions and simplifications (cont'd)

- **Impact of caching manager**
 - **Blocks** we require **may already be loaded into the memory**
 - And so the actual cost may be lower
- **Extent of available statistics**
 - **Not all statistics we worked with may be available**
 - Or derivable in case of inner nodes
 - And so less accurate estimates can then be made
- **Lazy maintenance of statistics**
 - **Statistics** we do have **may already be obsolete**
 - Simply because some of them are updated only occasionally

Observations

False assumptions and simplifications (cont'd)

- **Non-uniform distribution**
 - Assumption of **uniform distribution is often not realistic**
 - And it is not just about the **data**
 - But also **queries**
- **Independence of conditions**
 - When **reduction factors for conditions** are estimated...
 - Their independence is assumed
 - But this may not be realistic again
- **Cost estimation** in general
 - Our formulae provide only estimates, not precise calculations
 - Moreover, there was a lot of simplification
 - And the statistics we relied on may really be unavailable
 - And so despite the effort, they may not always work well

Conclusion

Evaluation **algorithms**

- Access methods
- Sorting
 - **External merge sort** with / without **priority queue**
- Joining
 - Binary / non-binary **nested loops join** with / without zig-zag
 - Basic / integrated **sort-merge join**
 - Classic / simple / partition / grace / hybrid **hash join**

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

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