



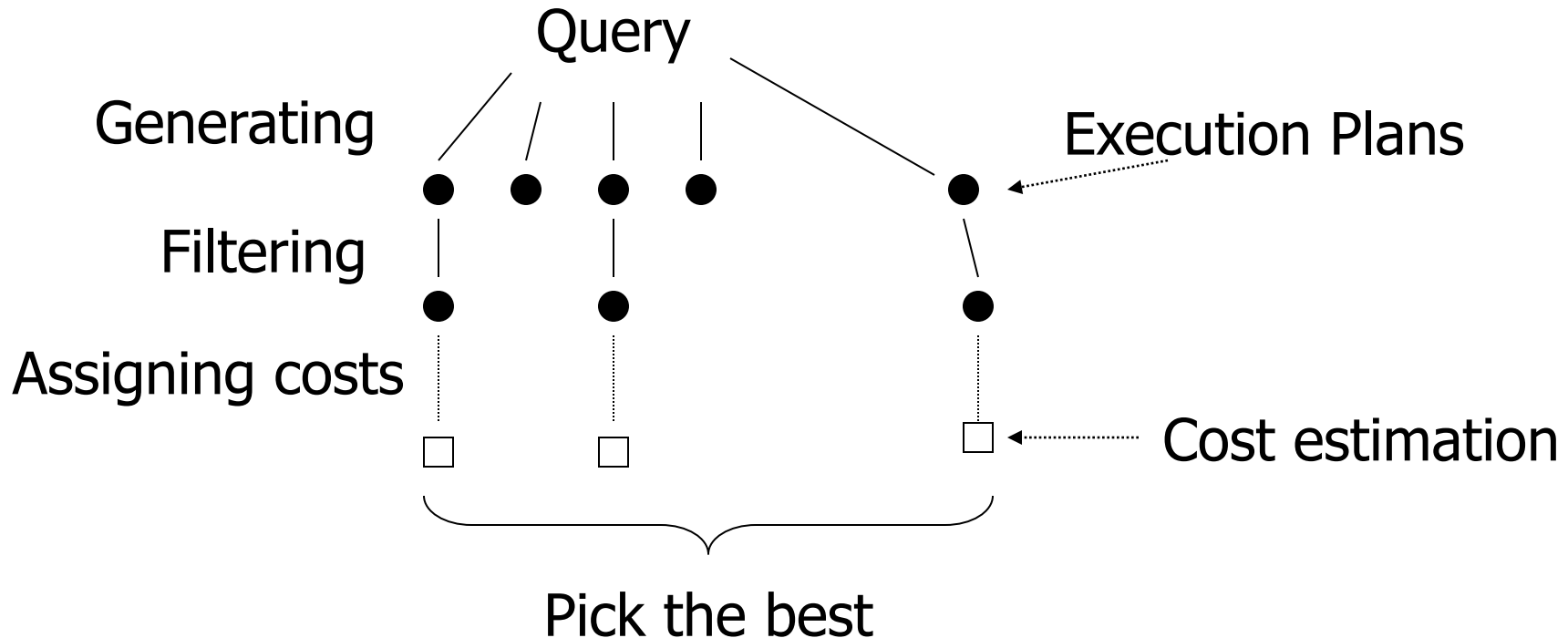
PA152: Efficient Use of DB

8. Query Optimization

Vlastislav Dohnal

Query Optimization

- Generating and comparing query execution plans



Generating Execution Plans

- Consider using:
 - Rel. algebra transformation rules
 - Implementations of rel. alg. operations
 - Use of existing indexes
 - Building indexes and sorting on the fly

Plan Cost Estimation

- Depends on costs of each operation
 - i.e., its implementation
- Assumptions for operation costs:
 - Input is read from and disk
 - Output is kept in memory
 - Costs on CPU
 - Processing on CPU is faster than reading from disk
 - Can be neglected but often simplified
 - Network communication costs
 - Issue in distributed databases
 - Ignoring contents of mem buffers/caches between queries
- Estimated costs of operation
 - = number of read and write accesses to disk

Operation Cost Estimation

■ Example: settings in PostgreSQL

<https://www.postgresql.org/docs/15/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS>

<https://www.postgresql.org/docs/15/static/runtime-config-resource.html>

- seq_page_cost (1.0)
- random_page_cost (4.0)
- cpu_tuple_cost (0.01)
- cpu_index_tuple_cost (0.005)
- cpu_operator_cost (0.0025)

- shared_buffers (32MB) – $\frac{1}{4}$ RAM
- effective_cache_size (4GB) – $\frac{1}{2}$ RAM
- work_mem (8MB)
 - Memory available to an operation

Operation Cost Estimation

■ Parameters

- $B(R)$ – size of relation R in blocks
- $f(R)$ – max. record count to store in a block
- M – max. RAM buffers available (in blocks)

- $HT(i)$ – depth of index i (in levels)
- $LB(i)$ – sum of all leaf nodes of index i

Operation Implementation

■ Based on concept of **iterator**

- *Open* – initialization

 - preparations before returning any record of result

- *GetNext* – return next record of result

- *Close* – finalization

 - release temp buffers, ...

■ Advantages

- Result may not be returned *at once*

 - Does not occupy main memory; may not be materialized on a disk

- Pipelining can be used

Accessing Relation: **table scan**

- Relation is not interlaced



- Reading costs: $B(R)$
- TwoPhase-MergeSort = $3B(R)$ reading/writing
 - Final writing is ignored

- Relation is interlaced



- Reading costs are up to $T(R)$ blocks!
- TwoPhase-MergeSort
 - $T(R) + 2B(R)$ reads and writes

Accessing Relation: **index scan**

■ Read relation using an index

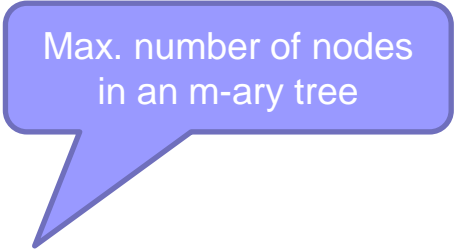
- Scanning index → reading records

- Read index blocks ($\ll B(R)$)
- Read records of relation

- Applicable to any attribute

- Max. costs:

- (max. $B(R)$ and $T(R)$ reads) + (up to $m^{HT+1} - 1$)
 - Where m is an index arity ($LB = m^{HT}$)



Max. number of nodes
in an m-ary tree

■ Advantages

- Can limit to a subset of records (interval)

- Min. costs: 0 read blocks of relation + 1.. HT blocks of index
 - For a covering index

One-Pass Algorithms

■ Implementation:

- Read relation → Processing → Output buffers
- Processing records one by one

■ Operations

- Projection, Selection, Duplicate elimination (DISTINCT)
 - costs: $B(R)$
- Aggregation functions (GROUP BY)
 - costs: $B(R)$
- Set operations, cross product
 - costs: $B(R) + B(S)$

Duplicate Elimination – distinct

■ Procedure

- Test whether the record is in output
- If not, output the record

■ Test for existence in output

- Store already seen records in memory
 - Can use $M-2$ blocks
- No data structure: n^2 complexity (comparisons)
- Use hashing

■ Limitation: $B(R) < M-1$

■ Can be implemented using iterators?

Distinct – example

■ Relation company(company_key,company_name)

```
# explain analyze SELECT DISTINCT company_name FROM provider.company;  
HashAggregate (cost=438.68..554.67 rows=11600 width=20) (actual time=9.347..12.133 rows=11615 loops=1)  
  Group Key: company_name  
    -> Seq Scan on company (cost=0.00..407.94 rows=12294 width=20)  
        (actual time=0.019..5.007 rows=12295 loops=1)  
  
Planning time: 0.063 ms  
Execution time: 12.799 ms
```

```
# explain analyze SELECT DISTINCT company_key FROM provider.company;  
Unique (cost=0.29..359.43 rows=12294 width=8) (actual time=0.041..8.857 rows=12295 loops=1)  
  -> Index Only Scan using company_pkey on company (cost=0.29..328.69 rows=12294 width=8)  
      (actual time=0.039..5.686 rows=12295 loops=1)  
  
    Heap Fetches: 4726  
Planning time: 0.063 ms  
Execution time: 9.645 ms
```

```
# explain analyze SELECT DISTINCT company_name FROM provider.company ORDER BY company_name;  
Unique (cost=1243.05..1304.52 rows=11600 width=20) (actual time=53.468..59.072 rows=11615 loops=1)  
  -> Sort (cost=1243.05..1273.79 rows=12294 width=20) (actual time=53.467..55.482 rows=12295 loops=1)  
    Sort Key: company_name  
    Sort Method: quicksort Memory: 1214kB  
      -> Seq Scan on company (cost=0.00..407.94 rows=12294 width=20)  
          (actual time=0.018..5.338 rows=12295 loops=1)
```

Aggregations / Grouping

■ Procedure

- Create groups for group-by attributes
- Store accumulated values of aggregation functions

■ Internal structure

- Organize values of grouping attributes, e.g., hashing
- Accumulated value of aggregations
 - MIN, MAX, COUNT, SUM – one value (number)
 - AVG – two numbers (SUM and COUNT)
- Accumulated values are small: $M-1$ blocks are enough

■ Iterators:

Output block is not reserved.

- All prepared in *Open*
- Advantage of pipelining is inapplicable

Set Operations

- Requirement: $\min(B(R), B(S)) \leq M-2$
 - Smaller relation read in memory
 - Larger relation is read gradually
 - Set union (possibly also Set difference):
 - Memory requirements: $B(R)+B(S) \leq M-2$
- Assumption
 - R is larger relation, i.e., S is in memory
- Implementation
 - Create a temp search structure
 - E.g., hashing

Set union

- Notice: Not *multiset union*

i.e., without ALL in SQL

- Read S; construct search structure
 - Eliminate duplicates
 - Output unique records immediately
- Read R and check existence of the record in S
 - If present, skip it.
 - If not seen, output it and add to structure
- Limitations
 - $B(R)+B(S) \leq M-2$

Set intersection

- Notice: Not *multiset intersection*

i.e., without ALL in SQL

- Read S; construct search structure
 - Eliminate duplicates
- Read R and check existence of the record in S
 - If present, output the record and delete it from structure.
 - If not seen, skip it.
- Limitations
 - $\min(B(R), B(S)) \leq M-2$

Set Difference

■ R–S

- Read S; construct search structure
 - Eliminate duplicates
- Read R and check existence of the record in S
 - If not present, output it
 - Also insert into internal structure
- $B(S) + B(R) \leq M-2$ (worse case, but with pipelining)
 - Or $\max(B(R), B(S)) \leq M-2$, when preprocessing R (no pipelining)

■ S–R

- Read S; construct search structure
 - Eliminate duplicates
- Read R and check existence of the record in S
 - If present, delete it from internal structure
- Output all remaining recs. in S (no pipel.)
- $B(S) \leq M-1$

Multiset (Bag) Operations

- Bag union $R \cup_B S$
 - Easy exercise...
- Bag intersection $R \cap_B S$
 - Read S ; construct search structure
 - Eliminate duplicates by storing their count
 - Read R and check existence of the record in S
 - If record is present, output it
 - and decrement record count!
 - If counter is zero, delete it from internal structure
 - If record is not found, skip it
 - $\min(B(R), B(S)) \leq M-2$

Multiset (Bag) Operations

■ Bag difference $S -_B R$

- Same idea
- If record of R is present in S , decrement its counter
- Output internal structure (recs. of S)
 - with positive count
- $B(S) \leq M-1$

■ Bag difference $R -_B S$

- By analogy...
- If record of R is not present in $S \rightarrow$ output
- If found,
 - \rightarrow if counter is zero, output it
 - \rightarrow decrement the counter and skip it

□ $B(S) \leq M-2$

Join Operation – one pass version

■ Cross product

- Easy exercise...

■ Natural join

- Assume relations $R(X,Y)$, $S(Y,Z)$

- X – unique attributes in R , Z – unique attrs. in S
- Y – common attributes in R and S

- Read S ; construct search structure on Y

- For each record of R , find all matching recs. of S

- Output concatenation of all combinations (eliminate repeating attributes Y)

■ Outer join ?

One-Pass Algorithms

■ Summary

- Unary operation: $op(R)$

- $B(R) \leq M-1$, 1 block for output; some need 1 for input

- Binary operation: $R \ op \ S$

- $B(S) \leq M-2$, 1 block for R, 1 block for output

- Some ops require: $B(R)+B(S) \leq M-2$ or $\max(B(R),B(S)) < M-1$

- Cost = $B(R) + B(S)$

■ Based on size of memory buffers M

- Known \rightarrow ok

- Not known \rightarrow estimate it

- Wrong size \rightarrow swapping, use two-pass algo instead of one-pass algorithm

Join Algorithms

- Relations do not fit in memory
 - So called “one and a half”-pass algorithms
- Basic variant: *Nested-loop join*
 - **for** each s in S **do**
 - **for** each r in R **do**
 - **if** r and s match in Y **then** output concatenation of r and s .

■ Example

□ $T(R) = 10\ 000$ $T(S) = 5\ 000$ $M=2$

□ $\text{Costs} = 5\ 000 \cdot (1 + 10\ 000) = 50\ 005\ 000$ IOs

reading a record of S

Reading whole R

Join Algorithms

- Relations accessed by blocks
- *Block-based nested-loop join*
 - R – inner relation, S – outer relation
- Example:

$$\square B(R) = 1000 \quad B(S) = 500 \quad M=3$$

$$\square \text{Costs} = 500 \cdot (1 + 1000) = 500\,500 \text{ IOs}$$

Join Algorithms

- Exploit all buffer blocks (M blocks)
 - Cached Block-based Nested-loop Join
 - Read M-2 blocks of relation S at once
 - Read relation R block by block
 - Join records
 - Costs: $B(S)/(M-2) \cdot (M-2 + B(R))$ IOs
- Example $R \bowtie S$:
 - M=102
 - Costs: $5 \cdot (100 + 1000) = 5\,500$ IOs
 - Swapping relations
 - Costs: $10 \cdot (100 + 500) = 6\,000$ IOs

Join Algorithms – Summary

- Nested-loops join
 - Use always blocked variant
 - Read the smaller relation into memory (if $M \gg 3$)
- Storage of relation
 - Important for final costs
 - Interlaced → each record needs one I/O
 - Non-interlaced → each record needs $B(R)/T(R)$ I/Os only
- Applicable to any join condition
 - theta joins

Two-Pass Algorithms

■ Procedure:

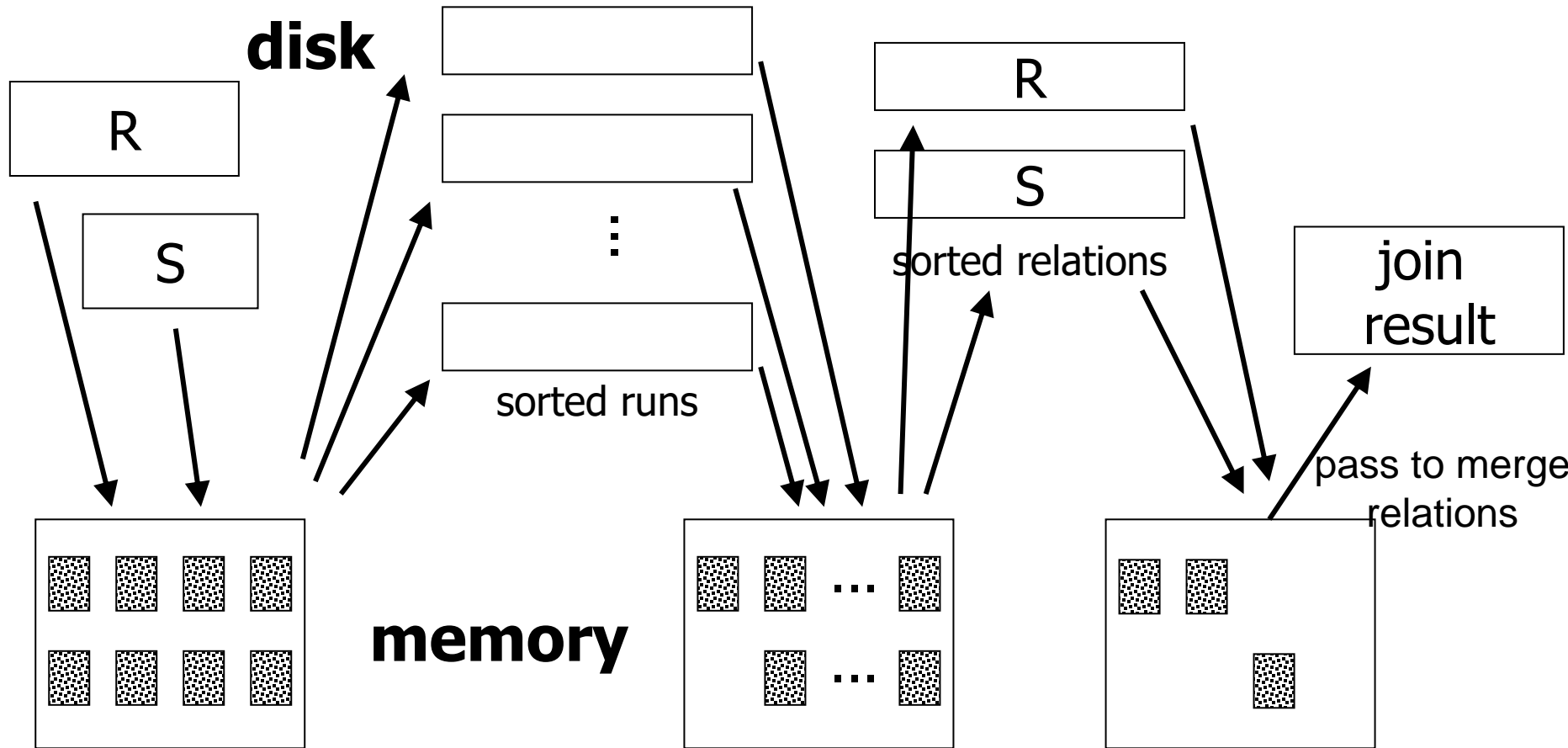
- Preprocess input relation → store it
 - Sorting (Multi-way MergeSort)
 - Hashing
- Processing

■ Operations:

- Joins
- Duplicate elimination (DISTINCT)
- Aggregations (GROUP BY)
- Set operations

Join Algorithms – MergeJoin

■ $R \bowtie S$ $R(X,Y), S(Y,Z)$



Join Algorithms – MergeJoin

- $R \bowtie S$ $R(X,Y), S(Y,Z)$
- Algorithm:
 - Sort R and S
 - $i = 1; j = 1;$
 - **while** $(i \leq T(R)) \wedge (j \leq T(S))$ **do**
 - **if** $R[i].Y = S[j].Y$ **then** doJoin()
 - **else if** $R[i].Y > S[j].Y$ **then** $j = j+1$
 - **else if** $R[i].Y < S[j].Y$ **then** $i = i+1$

Join Algorithms – MergeJoin

■ Function doJoin():

- Proceed nested-loop join for records of same Y

- **while** $(R[i].Y = S[j].Y) \wedge (i \leq T(R))$ **do**

- $j_2 = j$

- **while** $(R[i].Y = S[j_2].Y) \wedge (j_2 \leq T(S))$ **do**

- Output joined $R[i]$ and $S[j_2]$

- $j_2 = j_2 + 1$

- $i = i + 1$

- $j = j_2$

Join Algorithms – MergeJoin

i	R[i].Y	S[j].Y	j
1	10	5	1
2	20	20	2
3	20	20	3
4	30	30	4
5	40	30	5
		50	6
		52	7

Join Algorithms – MergeJoin

■ Costs

- MergeSort of R and S $\rightarrow 4 \cdot (B(R) + B(S))$
- MergeJoin $\rightarrow B(R) + B(S)$

■ Example (M=102)

□ MergeJoin

- Sorting: $4 \cdot (1000 + 500) = 6000$ read/write IOs
- Joining: $1000 + 500 = 1500$ read IOs
- Total: 7500 read/write IOs

□ Original cached block-based nested-loop join

- 5500 read IOs

Join Algorithms – MergeJoin

■ Another example

10x larger relations!!!

□ $B(R) = 10\ 000$

$B(S) = 5\ 000$

□ $M = 102$ blocks

□ Cached Block-based Nested-loop Join

■ $(5\ 000/100) \cdot (100 + 10\ 000) = 505\ 000$ read IOs

□ MergeJoin

■ $5 \cdot (10\ 000 + 5\ 000) = 75\ 000$ read/write IOs

Join Algorithms – MergeJoin

■ MergeJoin

- Preprocessing is expensive

- If relations are sorted by Y, can be omitted.

■ Analysis of IO costs

- MergeJoin

- linear complexity

- Cached Block-based Nested-loop Join

- quadratic complexity

- → from a certain size of relations,
MergeJoin is better

Join Algorithms – MergeJoin

■ Memory requirements

□ Limitation to $\max(B(R), B(S)) < M^2$

■ Optimal memory size

□ Using MergeSort on relation R

■ Number of runs = $B(R)/M$, Run length = M

■ Limitation: number of runs $\leq M - 1$

■ $B(R)/M < M \rightarrow B(R) < M^2 \rightarrow M > \sqrt{B(R)}$

■ Example

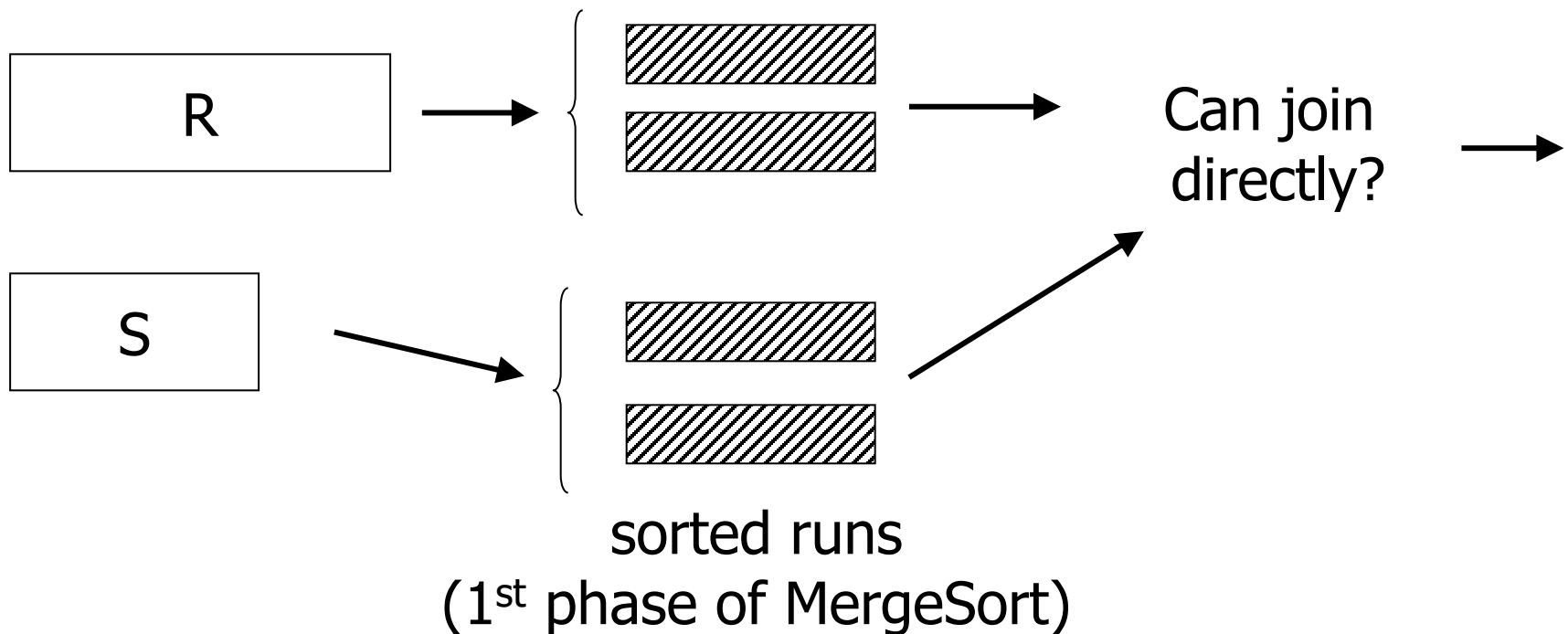
□ $B(R) = 1000 \rightarrow M > 31.62$

□ $B(S) = 500 \rightarrow M > 22.36$

Join Algorithms – MergeJoin → SortJoin

■ Improvement:

- Not necessary to have the relations sorted completely



Join Algorithms – SortJoin

■ Improvement

- Prepare sorted runs of R and S
- Read 1st block of all runs (R and S)
- Get min value in Y
 - Find corresponding records in other runs
 - Join them
- In case too many records with the same Y
 - Apply block-nested-loop join in the remaining memory

Join Algorithms – SortJoin

■ Costs

- Sorted runs: $2 \cdot (B(R) + B(S))$
- Joining: $B(R) + B(S)$

■ Limitations

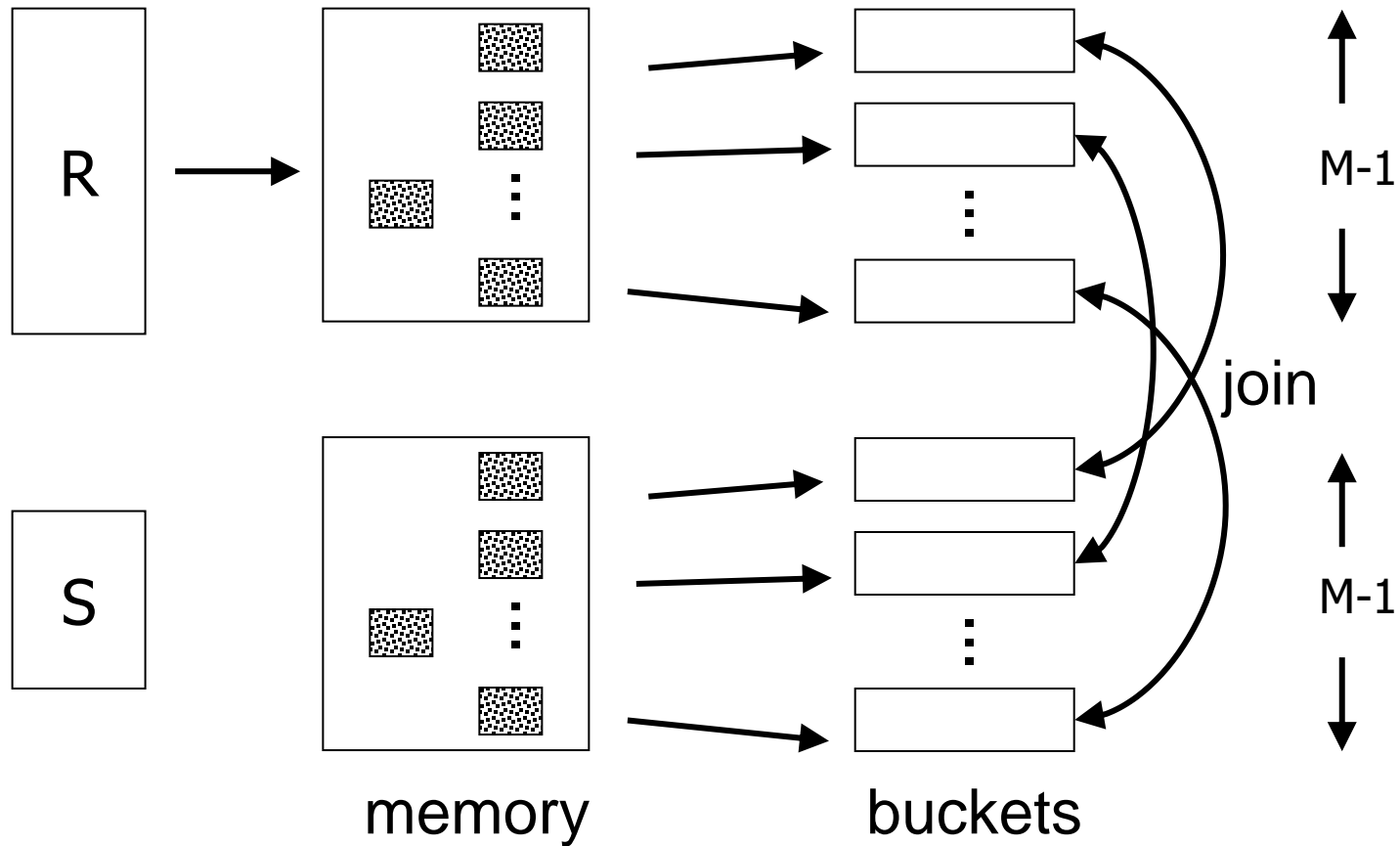
- Run length = M , number of runs $< M$
- $\rightarrow B(R) + B(S) < M \cdot (M-1)$

■ Example ($M=102$)

- Sorting: $2 \cdot (1000 + 500)$ Joining: $1000 + 500$
- Total: 4 500 read/write IOs
 - \rightarrow better than cached block-based nested-loop join

Join Algorithms – HashJoin

■ $R \bowtie S$ $R(X,Y), S(Y,Z)$



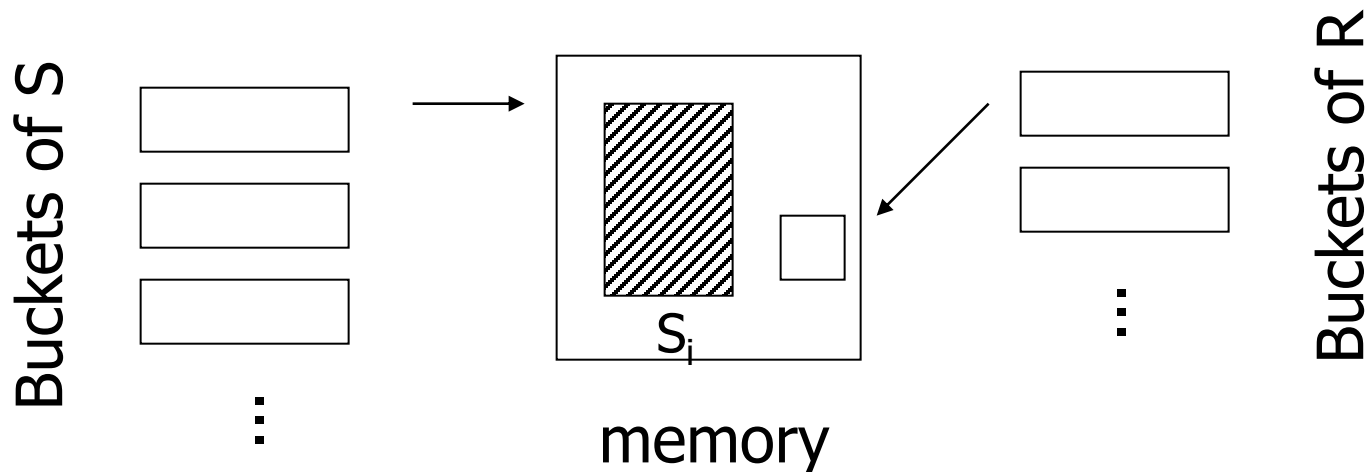
Join Algorithms – HashJoin

- $R \bowtie S$ $R(X,Y), S(Y,Z)$
 - Define a hash function for attributes Y
 - Create hashed index of R and S
 - Address space is M-1 buckets
 - For each $i \in [0, M-2]$
 - Read bucket i of R and S
 - Find matching records and join them

Join Algorithms – HashJoin

■ Joining buckets

- Read whole bucket of S ($\leq M-2$)
 - May create an internal structure to speed up
- Read bucket of R block by block



Join Algorithms – HashJoin

■ Costs:

- Create hashed index: $2 \cdot (B(R) + B(S))$
- Bucket joining: $B(R) + B(S)$

■ Limitations:

- Size of each bucket of $S \leq M - 2$
 - Estimate: $\min(B(R), B(S)) < (M - 1) \cdot (M - 2)$

■ Example:

- Hashing: $2 \cdot (1000 + 500)$
- Joining: $1000 + 500$
- Total: 4 500 read/write IOs

Join Algorithms – HashJoin

- Minimum memory requirements
 - Hashing S ; optimal bucket occupation
 - Memory buffer: M blocks
 - Bucket size = $B(S) / (M-1)$
 - This must be smaller than M (due to joining)
 - $\rightarrow [B(S)/(M - 1)] \leq M - 2$
 - $\approx M - 1 > \left\lceil \sqrt{B(S)} \right\rceil$

Join Algorithms – HashJoin

■ Optimization

- keep some buckets in memory
- Hybrid HashJoin

■ Bucketing of S – Optimal size

- $B(S)=500$

- $\sqrt{B(S)} \approx 23$

- i.e., each bucket is of 22 blocks

- $M=102$

- → keep 3 buckets in memory (66 blocks)

- → 36 blocks of memory to spare

Join Algorithm – Hybrid HashJoin

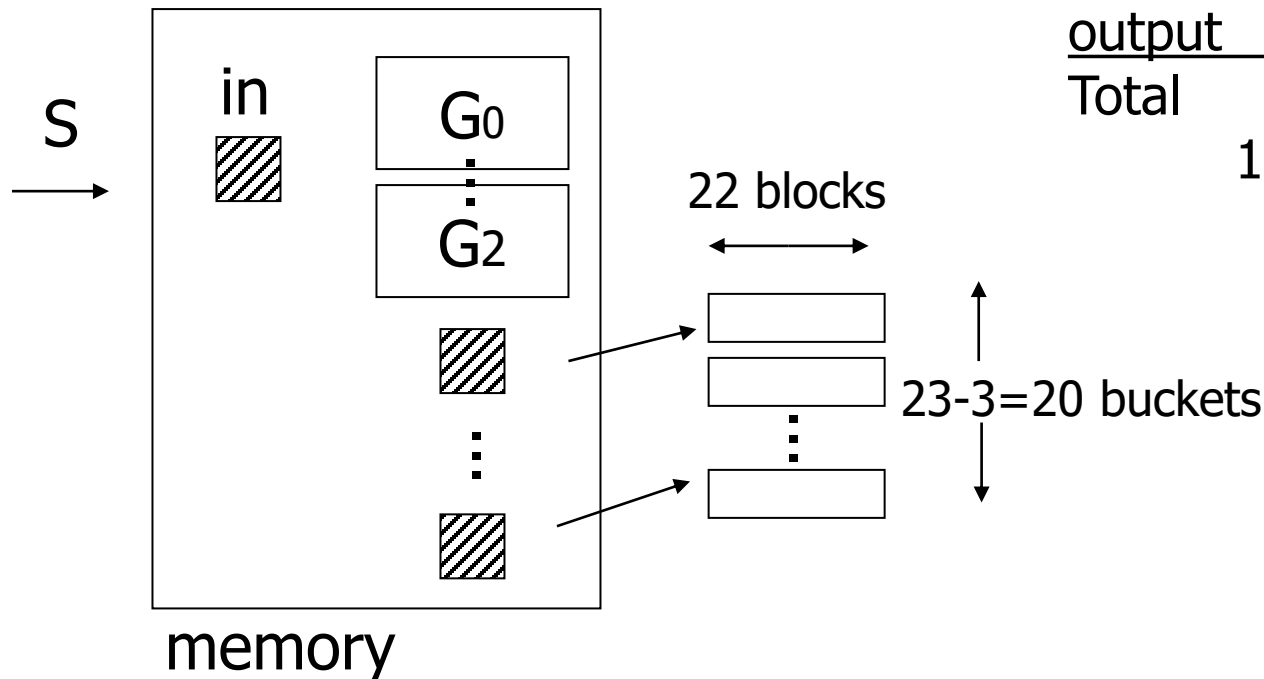
■ Preprocessing S

- Contents of memory buffer

Memory usage (M=102):

G0-2	3*22 blocks
Other buckets	23-3 blocks
Reading S	1 block
output	1 block
<hr/>	<hr/>
Total	88 blocks

14 blocks are available!



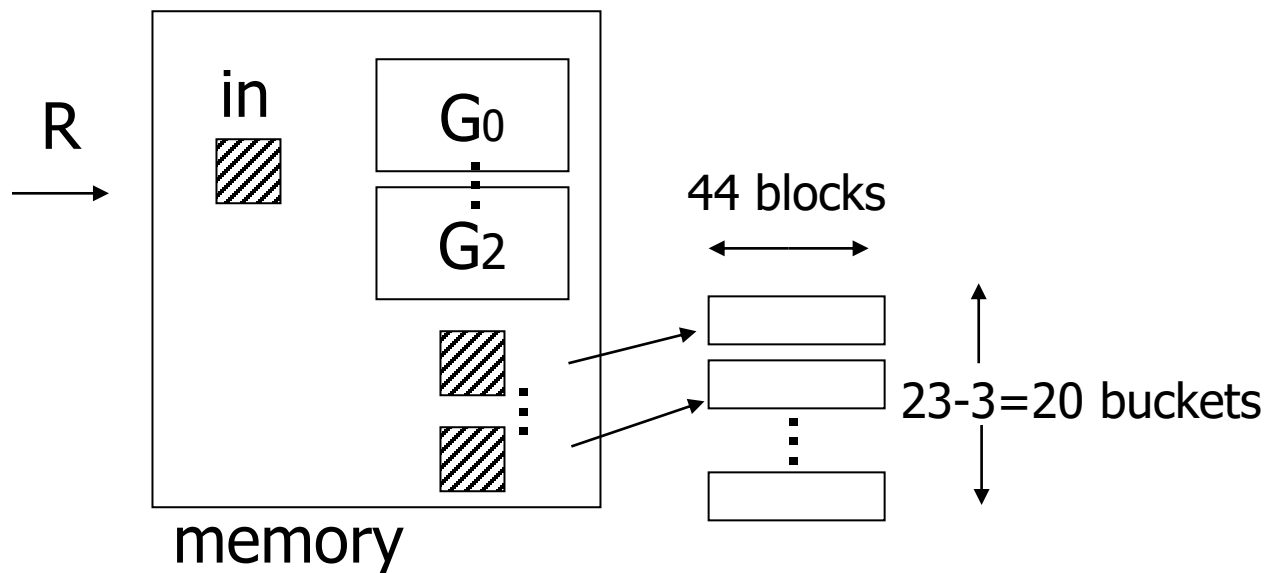
Join Algorithm – Hybrid HashJoin

- Structure of memory to hash R

- $1000/23 = 44$ blocks per bucket

- Records hashed to bucket 0-2

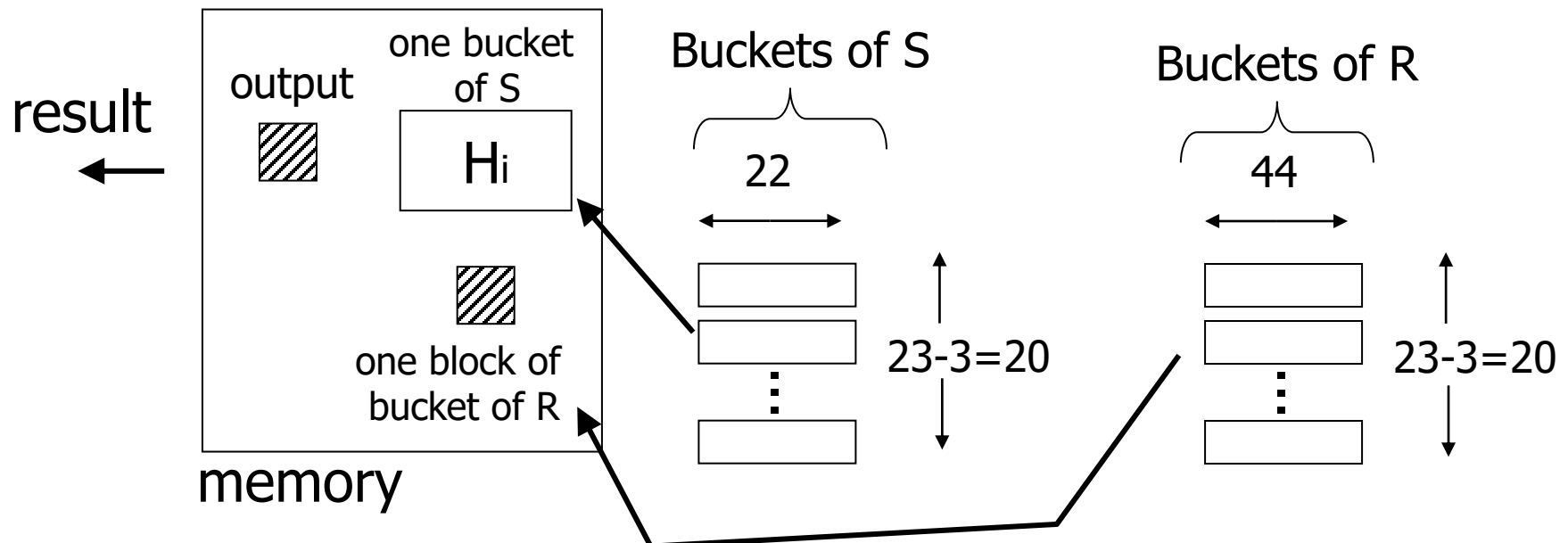
- Join immediately with S_{0-2} buckets (in memory) → output



Join Algorithm – Hybrid HashJoin

■ Joining buckets

- Do for buckets with id 3-22
- Read one whole bucket in memory; read the other bucket block by block



Join Algorithm – Hybrid HashJoin

■ Costs:

- Bucketize S: $500 + 20 \cdot 22 = 940$ read/write IOs
- Bucketize R: $1000 + 20 \cdot 44 = 1880$ read/write IOs
 - Only 20 buckets to write!
- Joining: $20 \cdot 44 + 20 \cdot 22 = 1320$ read IOs
 - Three buckets are already done (during bucketizing R)
- In total: 4140 read/write IOs

Join Algorithms

■ Hybrid HashJoin

- How many buckets to keep in memory?

 - Empirically: 1 bucket

■ Hashing record pointers

- Organize pointers to records instead of records themselves

 - Store pairs [key value, rec. pointer] in buckets

- Joining

 - If match, we must read the records

Join Algorithm – Hashing Pointers

■ Example

- 100 key-pointer pairs fit in one block
- Estimate results size: 100 recs
- Costs:
 - Bucketize S in memory (500 IOs)
 - 5000 records \rightarrow $5000/100$ blocks = 50 blocks in memory
 - Joining – read R gradually and join
 - If match, read full records of S \rightarrow 100 read IOs
 - Total: $500 + 1000 + 100 = 1600$ read IOs

Join Algorithms – IndexJoin

- $R \bowtie S$ $R(X,Y), S(Y,Z)$
- Assume:
 - Index on attributes Y of R
- Procedure:
 - For each record $s \in S$
 - Look up matches in index \rightarrow records A
 - For each record $r \in A$
 - Output concatenation of r and s

Join Algorithms – IndexJoin

■ Example

□ Assume

- Index on Y of R: HT=2, LB=200

■ Scenario 1

□ Index fits in memory

□ Costs:

- Pass of S: 500 read IOs ($B(S)=500, T(S)=5000$)
- Searching in index: for free
 - If match, read record of R → 1 read IO

Join Algorithms – IndexJoin

■ Costs

□ Depends on the number of matches

□ Variants:

- A) Y in R is primary key; Y in S is foreign key
→ 1 record

Costs: $500 + 5000 \cdot 1 \cdot 1 = 5500$ read IOs

- B) $V(R, Y) = 5000$ $T(R) = 10\ 000$
uniform distribution → 2 records

Costs: $500 + 5000 \cdot 2 \cdot 1 = 10500$ read IOs

- C) $DOM(R, Y) = 1\ 000\ 000$ $T(R) = 10\ 000$
→ $10k/1m = 1/100$ of record

Costs: $500 + 5000 \cdot (1/100) \cdot 1 = 550$ read IOs

Join Algorithms – IndexJoin

■ Scenario 2

- Index does not fit in memory

- Index on Y of R is of 201 blocks

 - Keep root-node block and 99 leaf-node blocks in memory $M=102$

- Costs for searching

 - $0 \cdot (99/200) + 1 \cdot (101/200) = 0.505$ read IOs per search (query)

Join Algorithms – IndexJoin

■ Scenario 2

□ Costs

- $B(S) + T(S) \cdot (\text{searching index} + \text{reading records})$

□ Variants:

- A) \rightarrow 1 record

Costs: $500 + 5000 \cdot (0.5 + 1) = 8000$ read IOs

- B) \rightarrow 2 records

Costs: $500 + 5000 \cdot (0.5 + 2) = 13000$ read IOs

- C) \rightarrow 1/100 of record

Costs: $500 + 5000 \cdot (0.5 + 1/100)$

= 3050 read IOs

Join Algorithms – Summary

$R \bowtie S$
 $B(R) = 1000$
 $B(S) = 500$

Algorithm	Costs
Cached Block-based Nested-loop Join	5500
Merge Join (w/o sorting)	1500
Merge Join (with sorting)	7500
Sort Join	4500
Index Join (R.Y index)	8000 → 550
Hash Join	4500
Hybrid	4140
Pointers	1600

Join Algorithms – Summary

$R \bowtie S$

Assume $B(S) < B(R)$, Y are common attributes

Algorithm	Costs	Limits
Block-based Nested-loop	$B(S) \cdot (1+B(R))$	$M=3$
Cached version	$B(S)/(M-2) \cdot (M-2 + B(R))$	$M \geq 3$
Merge Join (w/o sorting)	$B(R) + B(S)$	$M=3$
Merge Join (with sorting)	$5 \cdot (B(R) + B(S))$	$M = \sqrt{B(R)}$
Sort Join	$3 \cdot (B(R) + B(S))$	$M = \sqrt{B(R)} + \sqrt{B(S)} + 1$
Index Join (R.Y index) (max costs)	$B(S) + T(S) \cdot (HT + \theta)$ e.g. $\theta = T(R)/V(R,Y)$	min. $M=4$
Hash Join	$3 \cdot (B(R) + B(S))$	$M = 2 + \sqrt{B(S)}$ max. $M-1$ buckets
Hybrid	$3(B(R) + B(S)) - \frac{2(B(R) + B(S))}{\lceil \sqrt{B(R)} \rceil}$	$M = \frac{B(R)}{\lceil \sqrt{B(R)} \rceil} + (\lceil \sqrt{B(R)} \rceil) + 1$
Pointers	$B(S)+B(R)+T(R) \cdot \theta$ e.g. $\theta = T(S)/V(S,Y)$	$M=B(\text{hash index on } S)+3$

Join Algorithms – Recommendation

- **Cached Block-based Nested-loop Join**
 - Good for small relations (relative to memory size)
- **HashJoin**
 - For equi-joins (equality on attributes only)
 - Relations are not sorted or no indexes
- **SortJoin**
 - Good for *non-equi-joins*
 - E.g., $R.Y > S.Y$
- **MergeJoin**
 - If relations are already sorted
- **IndexJoin**
 - If index exists, it could be useful
 - Depends on expected result size

Two-Pass Algorithms

- Using sorting
 - Duplicate Elimination
 - Aggregations (GROUP BY)
 - Set operations

Duplicate Elimination

■ Procedure

- Do 1st phase of MergeSort
 - → sorted runs on disk
- Read all runs block by block
 - Find smallest record and output it
 - Skip all duplicate records

■ Properties

- Costs: $3B(R)$
- Limitations: $B(R) \leq M^*(M-1)$
 - Optimal $M \geq \sqrt{B(R)} + 1$

Aggregations

- Procedure (analogous to previous)
 - Sort runs of R (by group-by attributes)
 - Read all runs block by block
 - Find smallest value \rightarrow new group
 - Compute all aggregates over all records of this group
 - No more record in this group \rightarrow output it
- Properties
 - Costs: $3B(R)$
 - Limitations: $B(R) \leq M^*(M-1)$
 - Optimal $M \geq \sqrt{B(R)} + 1$

Set union

- Notice: No two-pass algo for bag union

- Set union

- Do 1st phase of MergeSort on R and S

- \rightarrow sorted runs on disk

- Read all runs (both R and S) gradually

- Find the first remaining record and output it

- Skip all duplicates of this record (in R and S)

- Properties

- Costs: $3(B(R) + B(S))$

- Limitations: $\sqrt{B(R) + B(S)} \leq M$

- Need one block per all runs (of R and also S)

Set/bag intersection and difference

- $R \cap S$, $R - S$, $R \cap_B S$, $R -_B S$
- Procedure
 - Do 1st phase of MergeSort on R and S
 - Read all runs (both R and S) gradually
 - Find the first remaining record t
 - Count t 's occurrences in R and S (separately)
 - $\#_R$, $\#_S$
 - Copy to output (respecting specific operation)

Set/bag intersection and difference

■ On *copy to output*.

□ $R \cap S$: output t ,

■ if $\#_R > 0 \wedge \#_S > 0$

□ $R \cap_B S$: output t $\min(\#_R, \#_S)$ -times

□ $R - S$: output t ,

■ if $\#_R > 0 \wedge \#_S = 0$

□ $R -_B S$: output t $\max(\#_R - \#_S, 0)$ -times

■ Properties

□ Costs: $3(B(R) + B(S))$

□ Limitations: $\sqrt{B(R) + B(S)} \leq M$

■ Need one block per all runs (of R and also S)

Two-Pass Algorithms

- Using hashing
 - Duplicate Elimination
 - Aggregations (GROUP BY)
 - Set operations

Duplicate Elimination

■ Procedure

- Bucketize R into $M-1$ buckets
 - → store buckets on disk
- For each bucket
 - Read it in memory and remove duplicates; output remaining records
 - bucket size is max. $M-1$ blocks

■ Properties

- Costs: $3B(R)$
- Limitations: $B(R) \leq (M-1)^2$

Aggregations

■ Procedure (analogous to previous)

- Bucketize R into $M-1$ buckets by group-by attrs.

- → store buckets on disk

- For each bucket

- Read block by block in memory and

- Create groups for new values and compute aggregates

- Limit on bucket size is not defined. But groups and partial aggregates must fit in max. $M-1$ blocks.

- Output results

■ Properties

- Costs: $3B(R)$

- Limitations: $B(R) \leq (M-1)^2$



can be relaxed

Set union, intersection, difference

■ Procedure

- Bucketize R and S (the same hash function)
 - into $M-1$ buckets
- Process the pair of buckets R_i and S_i
 - Read one in memory (depends on operation)
 - bucket size: max. $M-2$
 - Read the other gradually

■ Properties

- Costs: $3(B(R) + B(S))$
- Limitations on M depends on the operation

Set intersection, difference

- Intersection (smaller relation is S)
 - Load the bucket of S in mem
 - Restrictions: $\min(B(R), B(S)) \leq (M-2)*(M-1)$
- Difference R-S:
 - To eliminate duplicates in R, read bucket of R into mem
 - Restrictions: $B(R) \leq (M-2)*(M-1)$
- Difference S-R:
 - Load the bucket of S in mem
 - Restrictions: $B(S) \leq (M-2)*(M-1)$

Set Union

- Must eliminate duplicates in R and S
- for each i in hash addresses:
 - read Bkt^S_i , build in-mem hash table & eliminate dups
 - also gradually output the records
 - read Bkt^R_i gradually:
 - for each r in Bkt^R_i :
 - if r not in in-mem hash table
 - output r and add to in-mem hash table
- Restrictions: $\sqrt{B(R)} + \sqrt{B(S)} < M$
 - Need to load both the buckets (at worst) into M

Summary

■ Operations

- distinct, group by, set operations, joins

■ Algorithm type

- one-pass, one-and-a-half pass, two-pass

■ Implementation

- Sorting
- Hashing
- Exploiting indexes

■ Costs

- blocks to read/write
- memory footprint

Lecture Takeaways

- Influence of algorithm implementation on costs
- Estimated costs influence the choice of implementation
- If more mem is needed (estimation was wrong)
 - It is allocated and the operation is *not* terminated.
- Also, tiny code changes count!