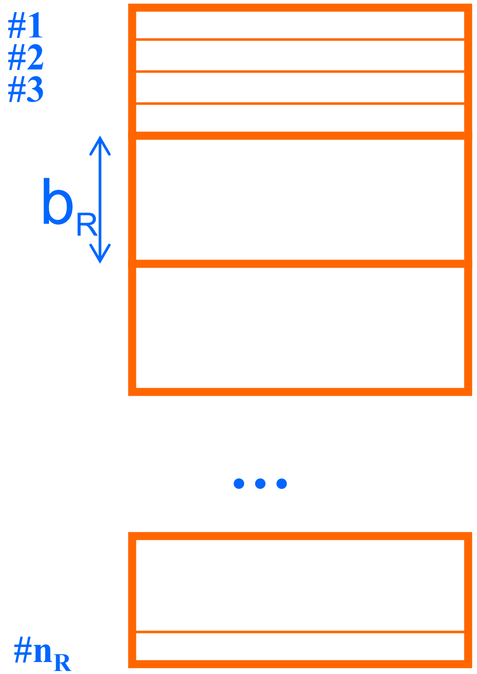
Query languages 1 (NDBI001)
Query evaluation

Jaroslav Pokorný
MFF UK, Praha
pokorny@ksi.mff.cuni.cz

Statistics

Statistics for each relation:

n_R	# of tuples in relation R
V(A,R)	# of elements in R[A]
p_R	# of pages to store R
b_R	blocking factor
M	# of pages of free space in RAM
I(A,R)	# of levels of index file for A in R



Notation:

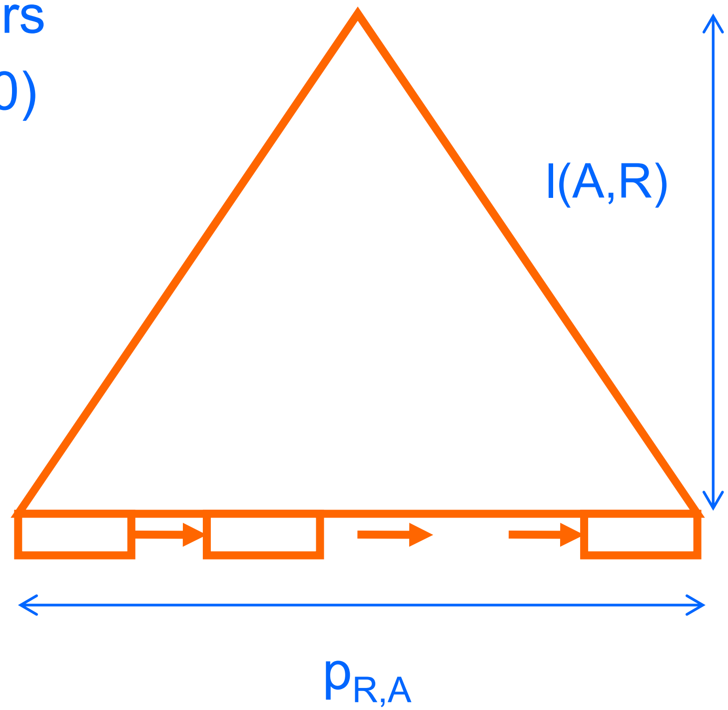
$buffer_R$	compact space of pages for R in RAM (we do not consider caching)

Indexing by B^+-trees:

Appropriate for: if there is an ordering on dom(A).

Consider attribute A of relation R:

- $f_{A,R}$: average number of successors in non-leaf node (~50-100)
- I(A,R): # index levels (~2-3)
  - ~ $\log(V(A,R))/\log(f_{A,R})$
- $p_{R,A}$: # leaf pages



Time and space complexity

- Measures for query cost:
  - CPU (cost of an operation is small; it is decreasing, difficult to estimate)
  - Disk (the main cost component - # of I/O operations)
- How many tuples is necessary to transfer?
- Which statistics should be maintained?

Notation: A instead of R.A

Methods for selection

SELECT *

FROM R

WHERE A = ‘a’

Cases: A is a primary key,

A is a secondary (alternative) key

there is an index on A - unclustered or of type CLUSTER

A is a hash key

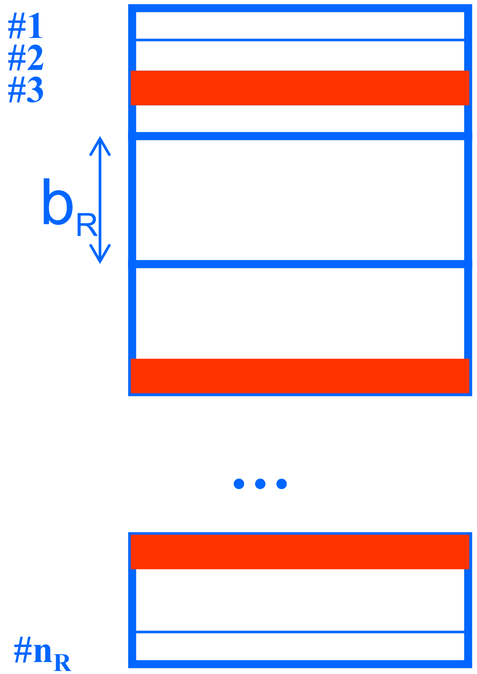
Assumption: uniform distribution of A values in R[A]

$\Rightarrow$ $n_{R(A=a)} = n_R / V(A,R)$

Methods for selection

Sequential scanning

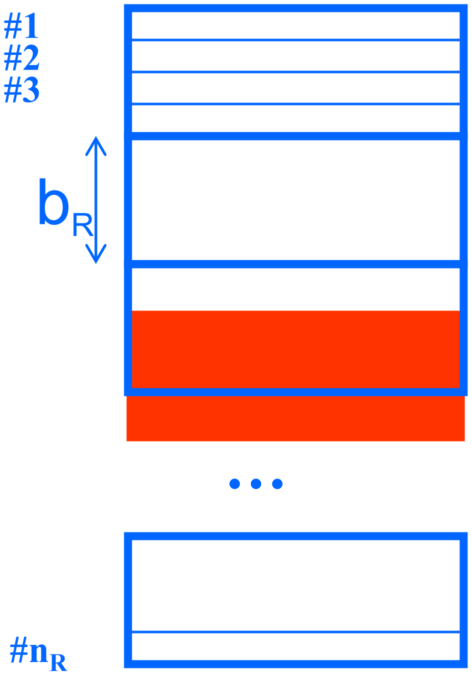
- p_R /*worst case*/
- $p_R/2$ /* average, if A is a primary key*/



Methods for selection

Binary search, if R is ordered on A

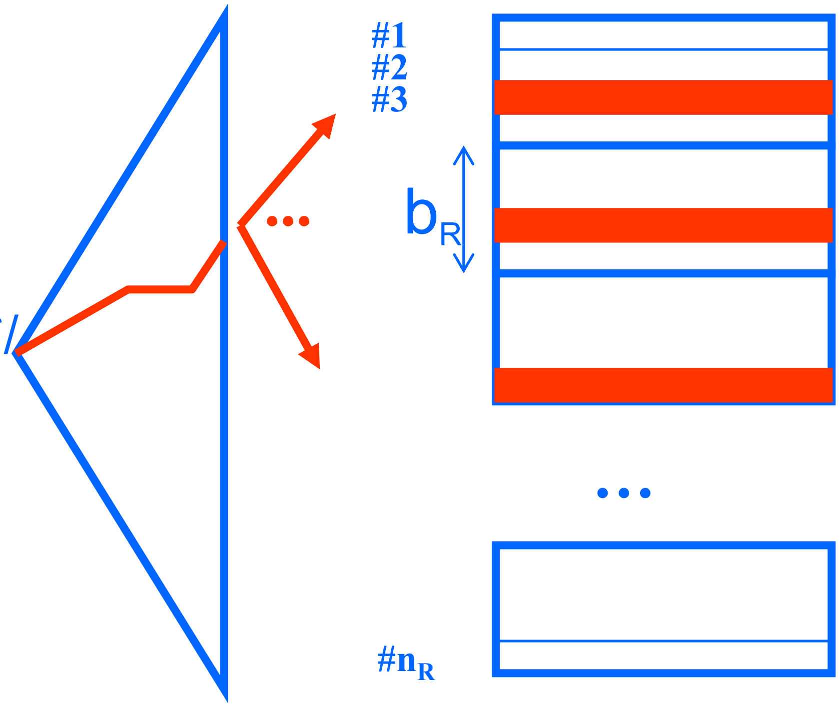
- $\log_2(p_{\mathbf{R}})$ /*if A is primary key*/
- $\log_2(p_{\mathbf{R}}) + n_{R(A=a)} / b_{\mathbf{R}}$ /*if A is arbitrary attribute*/



Methods for selection

Scanning, if there is an index for A

- $I(A) + 1$ /*if A is primary key*/
- $I(A) + n_{R(A=a)}/b_R$ /*if index for A is of type CLUSTER*/
- $I(A) + n_{R(A=a)}$ /*if index for A is not of type CLUSTER*/



Scanning, if A is a hash key

- ≈1 access

Methods for selection

SELECT *
FROM R
WHERE A < 'a'

Sequential scanning

- p_R /* the worst case*/
- $p_R(a - min_A)/(max_A - min_A)$ /*if R is sorted on A*/

Scanning, when there is an index

- $I(A) + p_R/2$ /*if R is sorted on A*/
- $I(A) + p_{R,A}/2 + n_R/2$ /* if there is an index for A, A is a secondary key*/

Example

Booking(passenger_n, flight_n, date, remark)

$n_{Booking}$ = 10 000

$b_{Booking}$ = 20

V(passenger_n, Booking) = 500

V(flight_n, Booking) = 50

$f_{flight\_n,Booking}$ = 20

Query: Find passengers with flight number = ‘77’

Example

Sequential scan:

$\Rightarrow$ query cost: 500 I/O operations

Clustered index for flight_n:

query cost = $I(flight\_n) + n_{Booking(flight\_n=70)}/b_{\mathbf{R}}$

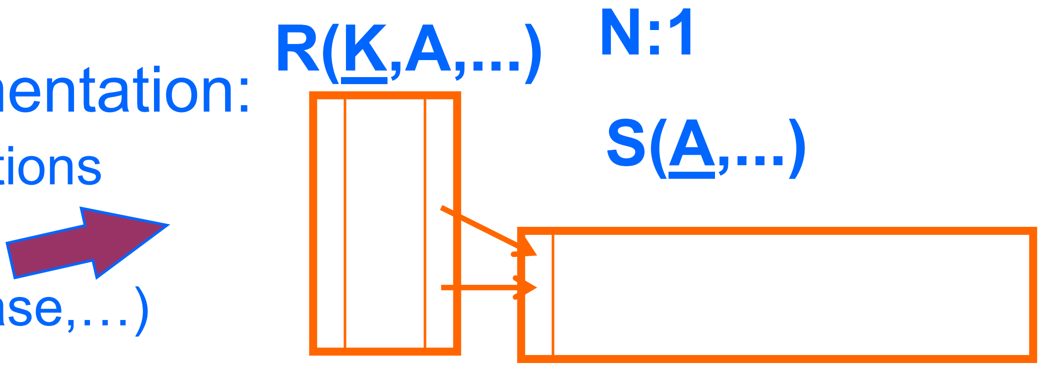
- $I(A)$: 50 values $f_A = 20 \Rightarrow I(A)=2$
  Justification: $(\log(50)/\log(20) \approx 2$
- $n_{R(A=a)} = n_R/V(A,r) =$ 10,000/50 = 200 tuples
  $n_{R(A=a)}/b_{\mathbf{R}}$ = 200/20 = 10 pages

$\Rightarrow$ query cost = 2+10= 12

Join operator calculation

Two types of implementation:

- independent relations
- with pointers
  (Starbrust, Winbase,…)



Basic methods:

- nested loops (variants with indexing, scanning)
- sort-merge
- hash join

Assumptions: join attribute A, $p_R \leq p_S$,
for the variant with pointers $R \rightarrow S$

Remark: special case - Cartesian product

Nested loops - binary

- naive algorithm

  for each r $\in$ R

  for each s $\in$ S

  if Θ(r,s) then begin u:= r [Θ] s; WRITE(u) end

  …

- by pages

  smaller relation as outer one!

  M=3 $\Rightarrow$ $p_R + p_R p_S$ reads

  $(n_R\, n_S / V(A,S))/b_{RS}$ writes (justify!)

Improvement: - inner relation is read ⇄

it saves 1 read at start (end)

Nested loops - binary

Variants:

- M big, then the partition: M-2, 1, 1

  outer inner result

  $\Rightarrow p_R + p_S p_R/(M-2)$ reads
- R in main memory

  $\Rightarrow p_R + p_S$ reads
- with pointers, M=3

  $\Rightarrow p_R + n_R$ reads

Nested loops - binary

- index on S.A (B^+-tree)

Assumptions: R sorted on R.A, S.A is primary

$\Rightarrow p_R + I(A,S) + p_{S,A} + V(A,R)$ reads

- S hashed on S.A

Assumptions: R sorted on R.A, S.A is primary

$\Rightarrow p_R + V(A,R)$ reads

- with selection (by scanning),

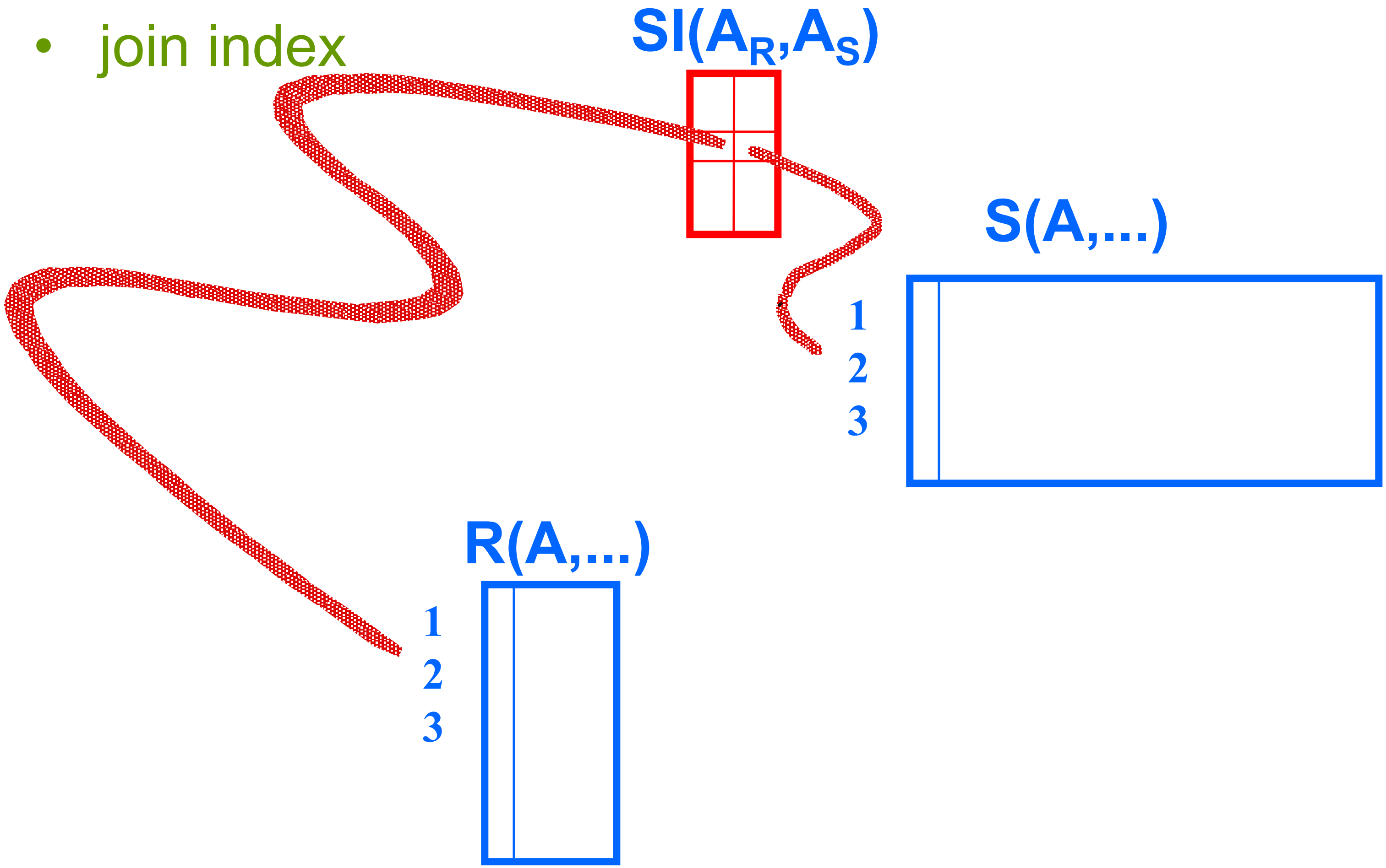
Ex.: SELECT * FROM R,S
WHERE R.A=S.A AND R.B=12

Assumptions: R.B primary key (indexed), S.A secondary key (clust. index, tuples with S.A=a in one page)

$\Rightarrow I(A,S) + I(B,R) + 2$ reads

Nested loops - binary

- join index



Nested loops - more relations

$M' = M_1 + M_2 + \ldots + M_n < M$

R_i are partitioned into l_i subrelations of length M_i, i.e.,
$l_i = \lceil p_i/M_i \rceil$, $(1 \leq i \leq n)$

Cost function [Kim84]:

$C = p_1 + [M_2+(p_2-M_2)\lceil p_1/M_1 \rceil]+\ldots+[M_n+(p_n-M_n)\lceil p_1/M_1 \ldots \lceil p_{n-1}/M_{n-1} \rceil]$

$\Rightarrow$ the problem of finding integer M_i, to obtain C minimal

Heuristics:

(1) List n relations into the algorithm proportionally by their size, that $p_1 \leq p_2 \leq \ldots \leq p_n$;

(2) For R_n allocate 1 page, M' - 1 divide equally;

(3) (M' - 1)/(n-1) is not integer then assign bigger M_i to smaller relations;

Nested loops - more relations

Structure of the basic algorithm (here for three relations):

```
for j:=1 to L1 do
   begin read R1j into bufferM1;
    for k:=1 to L2 into
         begin   read R2k into bufferM2;
                 for s:=1 to L3 into
                     begin read R3s into bufferM3;
                          create join of bufferMi, 1≤i ≤3;
                          write result
                     end
         end
   end
```

Nested loops - more relations

Ex.:

a) $p_1 = 7$, $p_2 = 14$, $p_3 = 21$, $M' = 11$

$\Rightarrow$ dividing M' = <5, 5, 1>

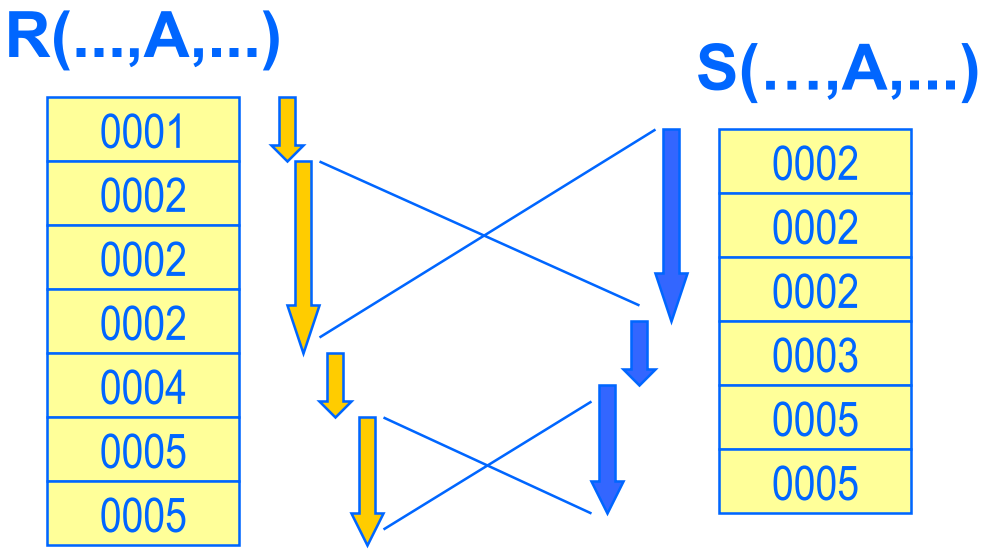
b) $p_1 \leq \ldots \leq p_5$, $M' = 16$

$\Rightarrow$ dividing M' = <4, 4, 4, 3, 1>

Sort-merge join

Idea: sorting, merging (two-pass algorithm)

Appropriate: if R and S are sorted



- min. M = 3, 2. phase requires $p_R + p_S$ reads
- Requires auxiliary space for sorting
- Result is sorted

Sort-merge

- M = 3 (with use of external sorting)

  $\Rightarrow \sim 2p_R \log(p_R) + 2p_S \log(p_S) + p_R + p_S$

  without writing the result

- M $\geq \sqrt{p_S}$ (two-pass algorithm)

(1) Sorted runs of length 2M pages are created (with a priority queue) and are written to disk;

$\Rightarrow$ length of run is $\geq 2\sqrt{p_S}$

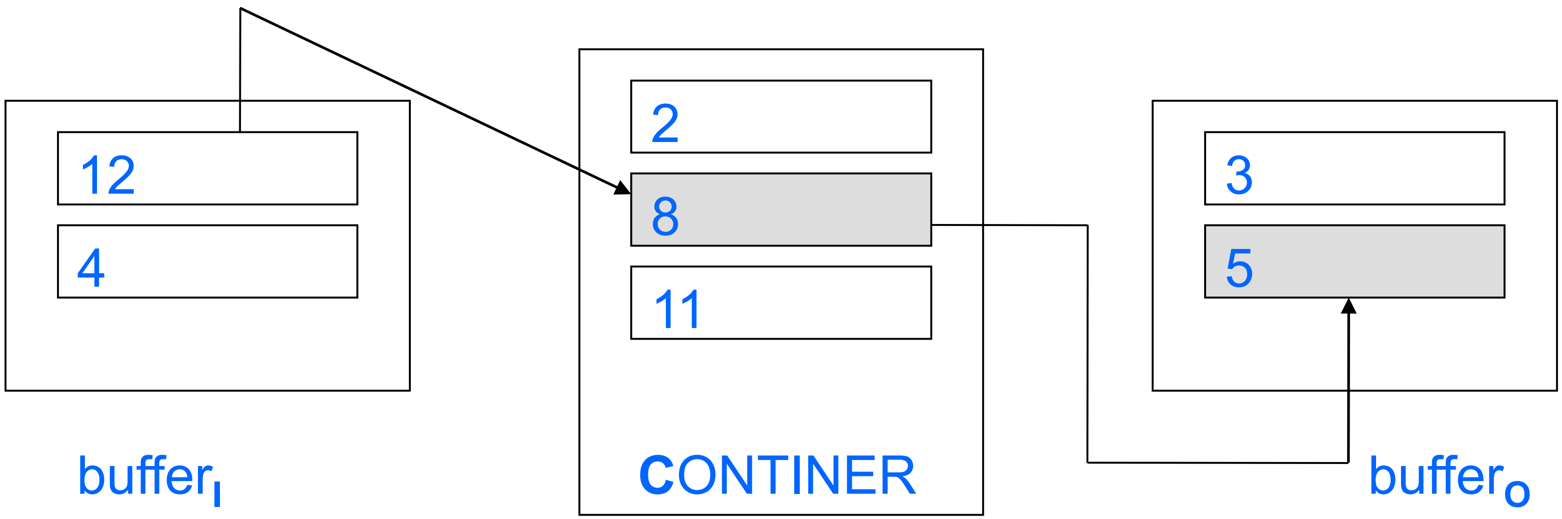
$\Rightarrow$ for S there is at most $p_S/2\sqrt{p_S}$ runs, for R also not more than $p_S/2\sqrt{p_S}$

$\Rightarrow$ totally at most $\sqrt{p_S}$

(2) For each run, a page is allocated in memory and these pages are concurrently merged;

$\Rightarrow 3(p_R + p_S)$ without writing the result

Principle of a priority queue



1. C and buffer$_I$ are filled up by tuples from R.
2. From C tuples *u* are selected such that ***u.A*** $\geq$ ***v.A***, $\forall v$ in buffer$_O$ (1) a rank in ascending order by values *A*.
3. Free place in K is filled up by a new tuple from buffer$_I$. If buffer$_I$ = $\emptyset$, then a new page R is read. If buffer$_O$ is full, then the given run on the disk is erlarged. If no tuple from the container fulfills (1), then the current state of buffer$_O$ is the last page of the run.

In this way, it is possible to create runs of length in average 2*M* pages.

Sort-merge

- variant with pointers

R is sorted by pointers

S is read only once, it has not to be sorted

$\Rightarrow 3p_R + p_S$ without writing the result

Comparison:

- $| p_R - p_S |$ is big $\Rightarrow$ nested loops is better
- $| p_R - p_S |$ is small $\Rightarrow$ sort-merge is better
- restricting selections $\Rightarrow$ better with scanning

Hash joins

Appropriate:

- indexes for R.A and S.A are not available
- result does not need to be sorted on A
  - classical hashing
  - GRACE algorithm
  - simple hashing
  - recursive partition of relations
  - hybrid hashing

Classical hash join

Assumption: R fits in M pages

$M = p_R$ *F + 1 + 1, where F is coefficient greater than 1

(1) Hash R into main memory;

(2) Read S sequentially;

Hash s.A and direct access read $r \in R$;

(3) if s.A = r.A then begin u:= r * s; WRITE(u) end

$\Rightarrow p_R + p_S$ reads

Partitioned hash join

Assumption: R does not fit in M pages

Idea: R and S are partitioned into disjunctive subsets in such way, that R tuples in partition *i* will only match S tuples in partition *i*.

Two pass algorithm:

(1) Partition R and S on disk;

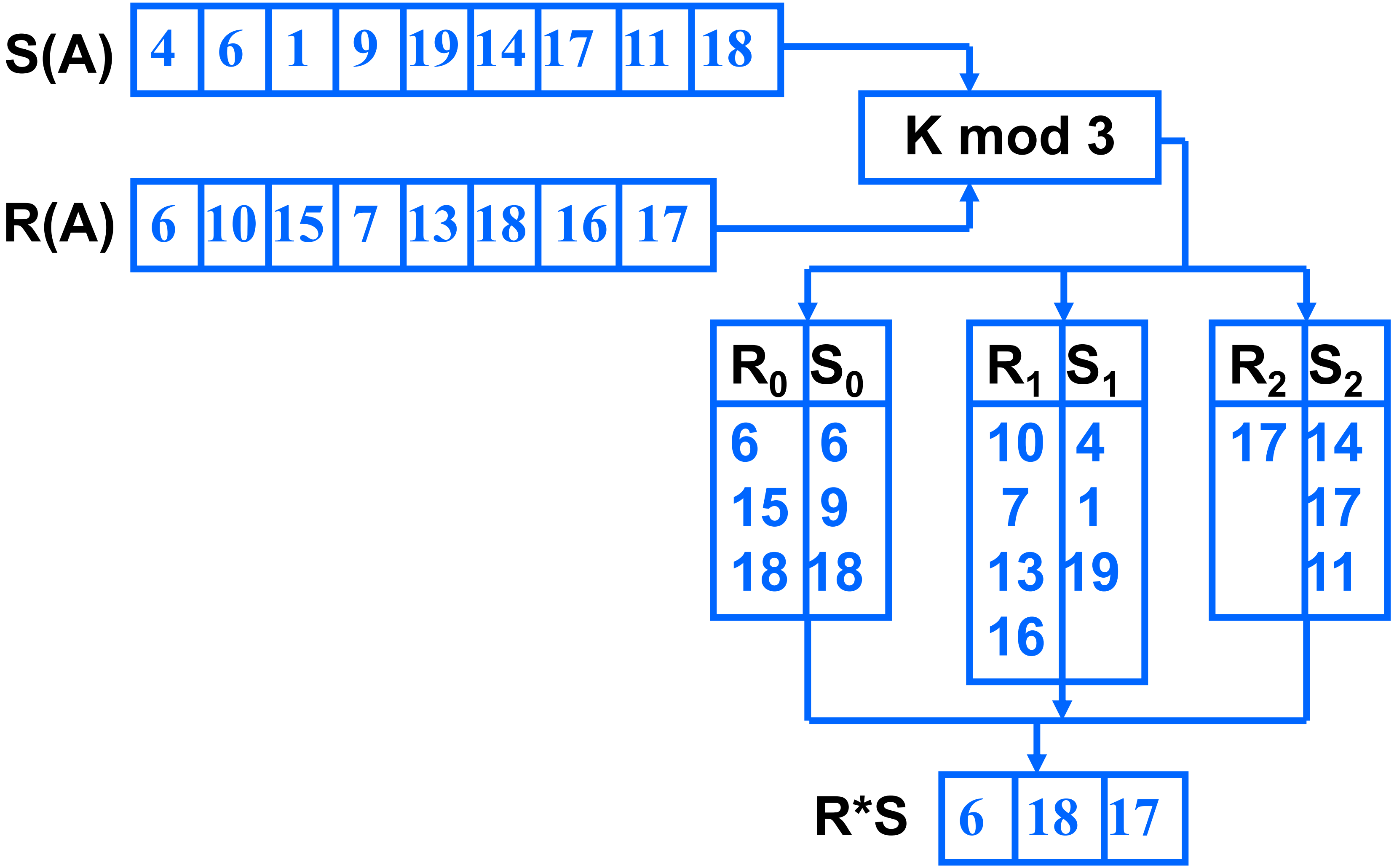
(2) Hash R part (R parts) into M-2 pages;

Read the related S part;

Hash s.A and by direct access search for matches in R;

Generate the result;

Example:


S(A)
4 6 1 9 19 14 17 11 18
R(A)
6 10 15 7 13 18 16 17
K mod 3
R0 S0
6 15 18
6 9 18
R1 S1
10 7 13 16
4 1 19
R2 S2
17
14 17 11
R*S
6 18 17

GRACE algorithm

- „school“ version

Data structures: tuples R a S, buckets of pointers HR_i, HS_i, $i \in \{0,1,\ldots,m-1\}$

Hash function h: dom(A) $\rightarrow$ <0,m-1>

Algorithm:

for k:=1 to n_R do begin i :=h(R[k].A); HR_i := $HR_i \cup \{k\}$ end

for k:=1 to n_S do begin i :=h(S[k].A); HS_i := $HS_i \cup \{k\}$ end

for i:=0 to m-1 do

begin $POM_R := \emptyset$; $POM_S := \emptyset$;

foreach $j \in HR_i$ do begin r:=R[j]; POM_R:=$POM_R \cup \{r\}$ end;

foreach $j \in HS_i$ do begin s:=S[j]; POM_S:=$POM_S \cup \{s\}$ end;

GRACE algorithm

```
foreach s ∈ POM_S do                          /* in RAM */
    begin foreach r ∈ POM_R do
        begin if s.A = r.A then begin u:= r * s;
                                      WRITE(u)
                                end
        end
    end
end
```

$\Rightarrow p_R + p_S + n_R + n_S$ read

appropriate: when $p_R/m + p_S/m < M$

GRACE with storing partitioned relations

- $M \geq \sqrt{(p_R * F)}$

(1) Choose h such that R can be partitioned into $m = \sqrt{(p_R * F)}$ partition;

(2) Read R a hash into (output) $buffer_i$, $(0 \leq i \leq m-1)$;

if $buffer_i$ is full then WRITE($buffer_i$);

(3) Do (2) for S;

(4) for i :=0 to m-1 do begin

(4.1) Read R_i and hash it into space of size $\sqrt{(p_R * F)}$;

(4.2) Read $s \in S_i$ a hash s.A.

If there is $r \in R_i$ a s.A = r.A, then generate the result.

end

GRACE with storing partitioned relations

Justification of 4.1: Assumption - $R_i \approx$ of the same size

$p_R/m = p_R/\sqrt{(p_R*F)} = \sqrt{(p_R/F)}$

R_i requires space $F\sqrt{(p_R/F)} = \sqrt{(p_R*F)}$

$\Rightarrow 3(p_R + p_S)$ I/O operations

Appropriate: when $p_R/m + p_S/m < M$

Remarks:

- S_i can be arbitrary. They require 1 page of memory;
- A problem, when V(A,R) is small;
- appropriate in situations, when R(<u>K</u>,…), S(<u>K</u>$_1$,K,…);
- If R_i resp. S_i does not fit in M-2 pages $\Rightarrow$ recursion
  i.e. R_i is partitioned into $R_{i0}, R_{i1},\ldots,R_{i(k-1)}$ sets of pages;

Simple hashing

Assumption: p_R*F > M-2, A is UNIQUE

Idea: special case of GRACE, when R $\rightarrow$ R_1, R_2

Algorithm:

repeat begin choose h; read R and hash r.A; M-2 buffers
    create R_1, other tuples into R_2 on disk;
        read S and hash s.A;
        if h(s.A) falls into space R_1
        then begin if s.A = r.A then generate result end
        else store s into S_2 on disk;
        R:=R_2 ; S:= S_2 end
until $R_2 \neq \emptyset$;

Hybrid hashing

Idea: combination of GRACE and simple hashing,

R is partitioned into parts R_1, R_2 ,…, R_k, R_0 such that R_0 fits into RAM.

Partition of M-2 pages: $|buffer_i| =1$ ($1 \leq i \leq k$), $|buffer_0| =p_{R0}$

Algorithm:

(1) Choose h;

(2) Read R and hash r.A; create R_i ($0 \leq i \leq k$);

/*R_0 is in $buffer_0$*/

(3) Read S and hash s.A; create S_i ($1 \leq i \leq k$);

if h(s.A) falls into space S_0 then create join;

(4) for i:=1 to k do create join by GRACE;

Comparing algorithms

Assumptions:

$M > \sqrt{p_S}$ for sort-merge

$M > \sqrt{p_R}$ for hashing

Notation: alg1 >> alg2 ⇔ alg1 is better than alg2

	Sort-merge	GRACE	Simple hashing	Hybrid hashing
GRACE	>>		>> (for smaller M)	
Simple hashing	>>	>> (for greater M)		
Hybrid hashing	>>	>>	>>	

Division

Df.: R and S with schemes Ω_1 and $\Omega_2 \subset \Omega_1$, respectively.

$T = R \div S = R[\Omega_1 - \Omega_2] - ((R[\Omega_1 - \Omega_2] \times S) - R)[\Omega_1 - \Omega_2]$

Ex.:

R	A	B
	3	6
	3	2
	8	2
	1	2
	1	3
	3	4
	3	3

S	B
	2
	4
	3

after sorting

R	A	B
	1	2
	1	3
	3	4
	3	3
	3	6
	3	2
	8	2

T	A
	3

Division by hashing

Idea: Buckets HS_i for values from S.B are created. The values from R.A are stored into them. Values from $\cap HS_i$ contribute to the result.

Algorithm: (elements of the hash table are, e.g., of type array or set, they represent buckets)

(1) Read S, calculate h(s.B) and denote the space (bucket) $HS_{s.B}$, foreach s.B do $HS_{s.B} := \emptyset$;

(2) for j:=1 to n_R do begin r:=R[j];
if there is a bucket for h(r.B)
then $HS_{r.B} := HS_{r.B} \cup \{r.A\}$ end

(3) foreach $HS_{s.B}$ do sort($HS_{s.B}$); /*is not necessary*/

(4) Create $\cap HS_i$ and generate T;

Other operations

GROUP BY

- index on A - over index we obtain groups
- sorting by R.A
- by hashing (as in division)

  foreach $a \in R[A]$ do create a bucket + variable for aggregation function calculation;

DISTINCT

also via hashing