Join Algorithms

Comparing Join Algorithms

Options:

Transformations: R1 $\bowtie$ R2, R2 $\bowtie$ R1

- Join algorithms:
  - Iteration (nested loops join)
  - Merge join
  - Join with index
  - Hash join

Factors that affect performance

(1) Tuples of relation stored physically together?

(2) Relations sorted by join attribute?

(3) Indexes exist?

Running Example

Example: R1 $\bowtie$ R2 over common attribute C

T(R1) = 10,000

T(R2) = 5,000

S(R1) = S(R2) = 1/10 block

Memory available = 101 blocks

$\rightarrow$ Metric: # of IOs (ignoring writing of result)

Iteration Join (Nested Loops Join)

for each r $\in$ R1 do
  for each s $\in$ R2 do
    if r.C = s.C then output r,s pair

Example: Iteration Join R1 $\bowtie$ R2

Relations not contiguous

Recall $\left\{ \begin{array}{l} T(R1) = 10{,}000 \quad T(R2) = 5{,}000 \\ S(R1) = S(R2) = 1/10 \text{ block} \\ MEM=101 \text{ blocks} \end{array} \right.$

Cost: for each R1 tuple:

[Read tuple + Read R2]

Total =10,000 [1+5000] = 50,010,000 IOs

Can we do better?

Use our memory

(1) Read tuples from R1 into 100 main memory blocks

(2) Read all of R2 (using 1 block) + join

(3) Repeat until done

Example: Improved Iteration Join

Cost: for each R1 chunk:

Read chunk:	1000 IOs
Read R2:	5000 IOs
	6000 IOs

$$\text{Total} = \frac{10{,}000}{1{,}000} \times 6000 = 60{,}000 \text{ IOs}$$

Can we do better?

☛ Reverse join order: R2 $\bowtie$ R1

Total = $\frac{5000}{1000}$ x (1000 + 10,000) =

5 x 11,000 = 55,000 IOs

Modified Example:

Relations contiguous

Cost

For each R2 chunk:

Read chunk: 100 IOs

Read R1: 1000 IOs

1,100

Total= 5 chunks x 1,100 = 5,500 IOs

Merge Join

Merge join (conceptually)

(1) if R1 and R2 not sorted, sort them

(2) i $\leftarrow$ 1; j $\leftarrow$ 1;

While (i $\leq$ T(R1)) $\wedge$ (j $\leq$ T(R2)) do

if R1{ i }.C = R2{ j }.C then outputTuples

else if R1{ i }.C > R2{ j }.C then j $\leftarrow$ j+1

else if R1{ i }.C < R2{ j }.C then i $\leftarrow$ i+1

Merge Join (cont.)

Procedure Output-Tuples

While (R1{ i }.C = R2{ j }.C) $\wedge$ (i $\leq$ T(R1)) do

[jj $\leftarrow$ j;

while (R1{ i }.C = R2{ jj }.C) $\wedge$ (jj $\leq$ T(R2)) do

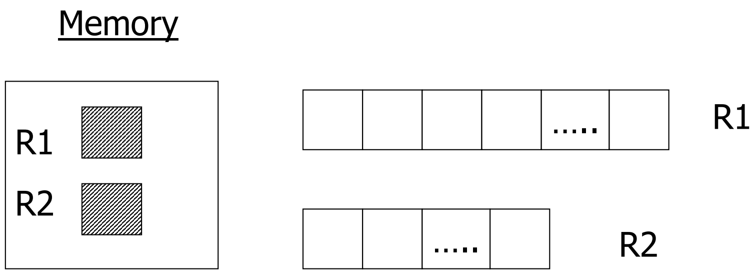
[output pair R1{ i }, R2{ jj };

jj $\leftarrow$ jj+1]

i $\leftarrow$ i+1]

Example: Merge Join

Assumption: Both R1, R2 ordered by C; relations contiguous



Total cost: Read R1 cost + read R2 cost

= 1000 + 500 = 1,500 IOs

Modified Example: Merge Join

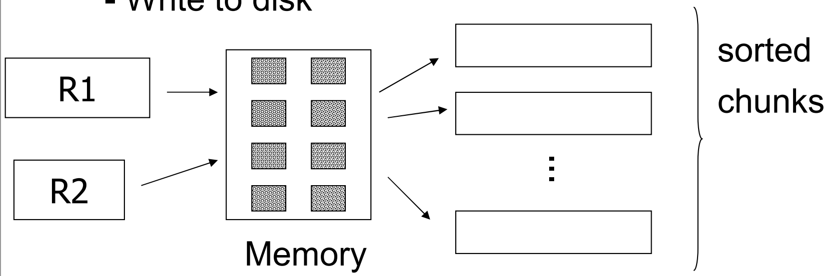
R1, R2 not ordered, but contiguous

--> Need to sort R1, R2 first…. HOW?

Recall: 2PMMS

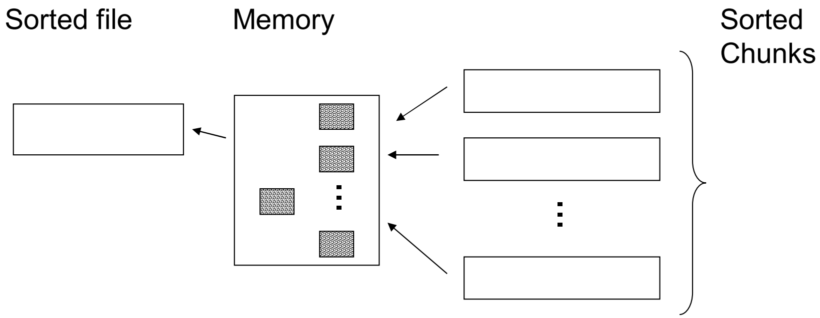
(i) For each 100 block chunk of R:

- Read chunk
- Sort in memory
- Write to disk



2PMMS (cont.)

(ii) Read all chunks + merge + write out



Cost 2PMMS

Each tuple is read,written, read, written

Thus,

Sort cost R1: 4 x 1,000 = 4,000

Sort cost R2: 4 x 500 = 2,000

Modified Example: Merge Join (cont.)

R1,R2 contiguous, but unordered

Total cost = sort cost + join cost

= 6,000 + 1,500 = 7,500 IOs

But: Iteration join cost = 5,500
so merge joint does not pay off!

Merge Join vs. Iteration Join

Iteration join is essetially quadratic whereas merge join is linear

Thus, the decision depends on the size of the relations:

Example: R1 = 10,000 blocks both contiguous
R2 = 5,000 blocks and not ordered

Iterate: $\frac{5000}{100}$ x (100+10,000) = 50 x 10,100
= 505,000 IOs

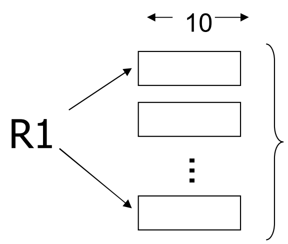
Merge join: 5 (10,000+5,000) = 75,000 IOs

Merge Join (with sort) Wins!

How much memory do we need for merge sort?

Example:

- contiguous relation with 1000 Blocks
- 10 memory blocks



100 chunks ⇒ to merge, need
100 input blocks!

In general:

Say M blocks in memory

B blocks for the relation to be sorted

chunks = $\lceil (B/M) \rceil$ size of chunk = M

chunks < buffers available for merge

Thus $\lceil (B/M) \rceil < M$

or approximately $M^2 > B$ or $M > \lceil \sqrt{B} \rceil$

In our example

R1 is 1000 blocks, $M > \lceil 31.62 \rceil$

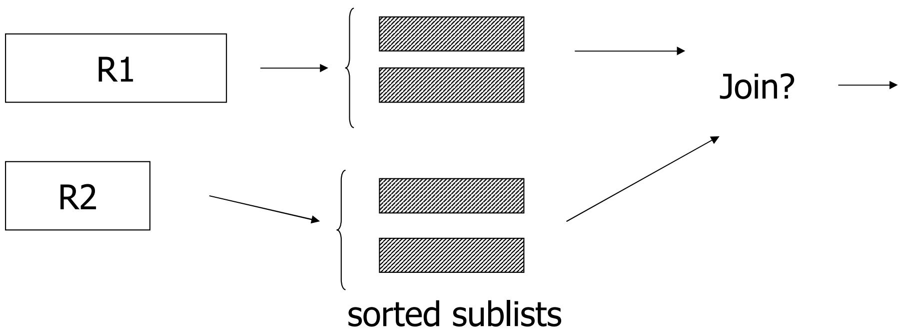
R2 is 500 blocks, $M > \lceil 22.36 \rceil$

Therefore, we need at least 33 buffer blocks in main memory

Can we improve on merge join?

Idea: do we really need the fully sorted files?

No! With enough main memory we can combine the last phase of the sorting algorithm with the actual join



Cost of improved merge join:

C = Read R1 + write R1 into sorted sublists

+ read R2 + write R2 into sorted sublists

+ join

= 2000 + 1000 + 1500 = 4500

Limitations: more memory required during join:

- In general we require $M^2 > B(R1) + B(R2)$
- For our example R1 and R2 we sort R1 (using 101 buffers) into 10 sorted sublists and R2 into 5 sublists.
- In the merge phase, we need at least 15 buffers, one per sublist, for input.
- That leaves additional 86 buffer blocks for records that share a C-value.

Hash Join

Hash join (conceptual)

- Hash function h, range 0 → k
- Buckets for R1: G0, G1, ... Gk
- Buckets for R2: H0, H1, ... Hk

Algorithm

(1) Hash R1 tuples into G buckets

(2) Hash R2 tuples into H buckets

(3) For i = 0 to k do

match tuples in Gi, Hi buckets

Implementation of DBMS

Simple example hash: even/odd

R1	R2
2	5
4	4
3	12
5	3
8	13
9	8
	11
	14

Buckets

	R1	R2
Even	2 4 8	4 12 8 14
Odd:	3 5 9	5 3 13 11

WS 24/25
Frankfurt UAS
Prof. Dr. Justus Klingemann

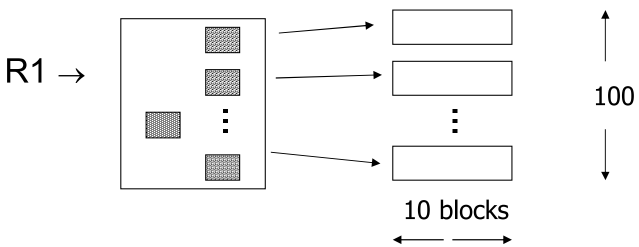
Implementation of DBMS

Example Hash Join

R1, R2 contiguous (un-ordered)

→ Use 100 buckets

→ Read R1, hash, + write buckets



WS 24/25
Frankfurt UAS
Prof. Dr. Justus Klingemann

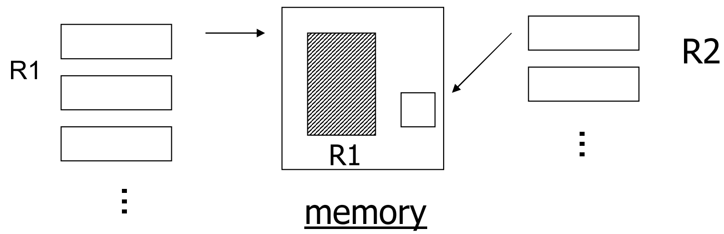
Implementation of DBMS

Example Hash Join

-> Same for R2

-> Read one R1 bucket; build memory hash table

-> Read corresponding R2 bucket + join



Then repeat for all buckets

WS 24/25
Frankfurt UAS
Prof. Dr. Justus Klingemann

Cost

"Bucketize:" Read R1 + write

Read R2 + write

Join: Read R1, R2

Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Minimum memory requirements:

Bucketizing: The number of buckets can be at most M-1

Assuming that all buckets have roughly the same size, the size of the bucket is = B / (M-1)

M = number of memory buffers

B = number of R blocks

Joining of individual buckets: The bucket of one relation has to be completely in main memory (in detail: fit in M-1 blocks)
Therefore: B/(M-1) <= M-1

or approximately $M > \lceil \sqrt{B} \rceil$

Note: In contrast to Merge join it is sufficient if this relationship holds for the smaller relation

Sort-based vs Hash-Techniques

The Hash-based algorithm has a size requirement that depends only on the smaller of the two relations rather than the sum of the argument sizes as for the optimized sort-based algorithm

The Sort-based algorithm allows us to produce a result in a sorted order and take advantage of that later

- The result might be used in another sort-based algorithm later, or
- It could be the answer to a query that is required to be produced in sorted order

The Hash-based algorithm depends on the buckets being of equal size. However, there is generally a variation in size. Result:

- It is not possible to use buckets that on average need the whole available main-memory. We must limit them to a smaller figure
- This affect is especially prominent, if the number of different hash keys is small

Different Number of Passes (1)

The execution of some algorithms relies on the availability of a certain number of memory buffers

- sort-based algorithms
- hash-based algorithms

Remind:

with

M = number of memory buffers

B = number of blocks for the relation(s)

we have to have approximately $M^2 > B$

The algorithms use two passes

- one pass to prepare the data (sorted sublists, buckets)
- a second pass to perform the desired action (e.g., join)

Different Number of Passes (2)

If we have relations of a larger size we can add one or more additional pass

- sort-based algorithms: the subsequences are merged to produce a smaller number of larger subsequences
- hash-based algorithms: each bucked is further divided into smaller buckets using a second hash function

Performance

- each additional pass requires reading and writing the relation

Size of the relation

- each additional pass increases the allowed size of the relation by a factor M
- using n passes, we have $M^n > B$

Different Number of Passes (3)

The other extreme: we have enough memory to accommodate one relation completely in main memory, i.e.

M > B

In this case we can omit one pass and get a trivial (one-pass) join-algorithm:

- load one relation completely in main-memory
- read the second relation one block at a time and join the tuples with the first relation in main-memory

Index Join:

Join with index (Conceptually)

Assume R1.C index

For each r ∈ R2 do
    [X ← index (R1, C, r.C)
        for each s ∈ X do
            output r,s pair]

Note: X ← index(rel, attr, value)
then X = set of rel tuples with attr = value

Example Index Join

Assumtions:

- R1.C index exists; 2 levels
- R2 contiguous, unordered
- R1.C index fits in memory

Cost: Reading R2: 500 IOs
for each R2 tuple (5000):
- probe index on R1: free
- if one matching tuple, read R1 tuple: 1 IO

Could be the best or worst method for this example

What is expected # of matching tuples?

(a) Case: R1.C is key, R2.C is foreign key

then expect = 1

(b) Case: V(R1,C) = 5000, T(R1) = 10,000

assumption: C-value we search is uniformly distributed over V(R1,C)

expect = 10,000/5,000 = 2

(c) Case: DOM(R1, C)=1,000,000

T(R1) = 10,000

with alternate assumption

$$\text{Expect} = \frac{10,000}{1,000,000} = \frac{1}{100}$$

Total cost with index join

(a) Total cost = 500 + 5000 * 1 = 5,500

(b) Total cost = 500 + 5000 * 2 = 10,500

(c) Total cost = 500 + 5000 * (1/100) = 550

What if index does not fit in memory?

Example: say R1.C index is 201 blocks

Keep root + 99 leaf nodes in memory

Expected cost of each probe is

$$E = (0) * \frac{99}{200} + (1) * \frac{101}{200} \approx 0.5$$

Total Cost (including Probes)

Total cost for case (b)

= 500+5000 [Probe + get records]

= 500+5000 [0.5+2]

= 500+12,500 = 13,000

Total cost for case (c):

= 500+5000 [0.5 + (1/100)]

= 500+2500+50 = 3050 IOs

Summary Join Algortihms

Iteration ok for "small" relations (relative to memory size)

For equi-join, where relations are not sorted and no indexes exist, hash join is usually best

Sort + merge join good for non-equi-join (e.g., R1.C > R2.C)

If relations already sorted, use merge join

If index exists, it could be useful

- depends on expected result size