Datenbanksysteme II: Implementation of Database Systems

Implementing Joins

Material von
Prof. Johann Christoph Freytag
Prof. Kai-Uwe Sattler
Prof. Alfons Kemper, Dr. Eickler
Prof. Hector Garcia-Molina

Content of this Lecture

- Nested loop and blocked nested loop
- Sort-merge join
- Hash-based join strategies
- Index join
- (Star – Join later)

Join Operator

- JOIN: Most important relational operator
  - Potentially very expensive
  - Required in all practical queries and applications
  - Often appears in groups of joins
  - Many variations with different characteristics, suited for different situations
- Example: Relations R (A, B) and S (B, C)

  SELECT * FROM R, S WHERE R.B = S.B

R

A	B
A1	0
A2	1
A3	2
A4	1

S

B	C
1	C1
2	C2
1	C3
3	C4
1	C5

$R \bowtie S$

A	B	C
A2	1	C1
A2	1	C3
A2	1	C5
A3	2	C2
A4	1	C1
A4	1	C3
A4	1	C5

Unary versus Binary Operations

- Relational operators working on one table
  - Selection, projection
- On two tables
  - Product, Join, Intersection, Union, Minus
- Binary operators are usually more expensive
  - Unary: Look at table (scanning, index, hash, ...)
  - Binary: Look at each tuple of first table for each tuple of second table
  - “Potentially” quadratic complexity

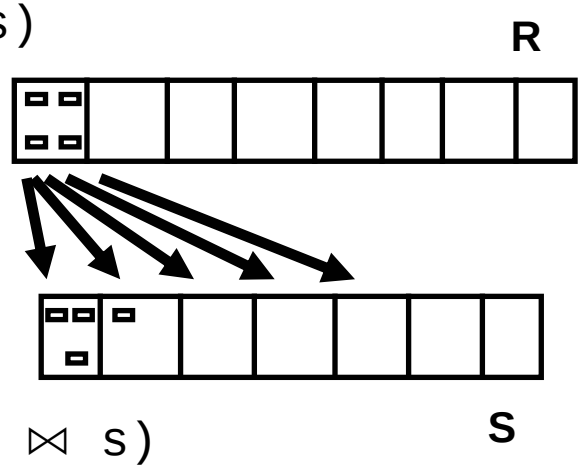
Nested-loop Join

- Super-naïve

```
FOR EACH r IN R DO
    FOR EACH s IN S DO
        IF ( r.B=s.B) THEN OUTPUT (r ⋈ s)
```

- Slight improvement

```
FOR EACH block x IN R DO
    FOR EACH block y IN S DO
      FOR EACH r in x DO
         FOR EACH s in y DO
           IF ( r.B=s.B) THEN OUTPUT (r ⋈ s)
```



- Cost estimations
  - b(R), b(S) number of blocks in R and in S
  - Each block of outer relation is read once
  - Inner relation is read once for each block of outer relation
  - Inner two loops are free (only main memory operations)
  - Altogether: b(R)+b(R)*b(S) IO

Example

- Assume b(R)=10.000, b(S)=2.000
- R as outer relation
  - IO = 10.000 + 10.000*2.000 = 20.010.000
- S as outer relation
  - IO = 2.000 + 2.000*10.000 = 20.002.000
- Use smaller relation as outer relation
  - For large relation, choice doesn't really matter

- Can't we do better??

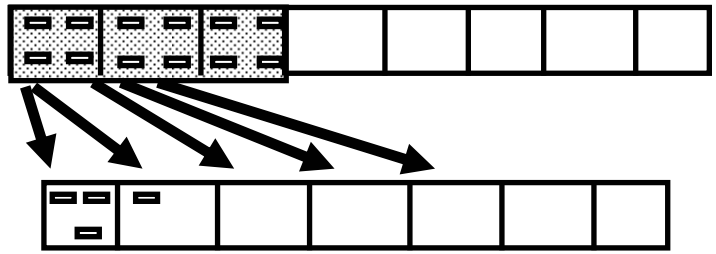
...

- There is no “m” in the formula
  - m: Size of main memory in blocks
- This should make you suspicious
- We are not using our available main memory

Blocked nested-loop join

- Rule of thumb: Use all memory you can get
  - Use all memory the buffer manager allocates to your process
  - This might be a difficult decision even for a single query – which operations get how much memory?
- Blocked-nested-loop

```
FOR i=1 TO b(R)/(m-1) DO
   READ NEXT m-1 blocks of R into M
   FOR EACH block y IN S DO
      FOR EACH r in R-chunk DO
       FOR EACH s in y do
         IF ( r.B=s.B) THEN OUTPUT (r ⋈ s)
```



- Cost estimation
  - Outer relation is read once
  - Inner relation is read once for every chunk of R
  - There are ~b(R)/m chunks
  - IO = b(R) + b(R)*b(S)/m
  - Further advantage: Outer relation is read in chunks – sequential IO

Example

- Example
  - Assume b(R)=10.000, b(S)=2.000, m=500
  - R as outer relation
    - IO = 10.000 + 10.000*2.000/500 = 50.000
  - S as outer relation
    - IO = 2.000 + 2.000*10.000/500 = 42.000
  - Compare to one-block NL: 20.002.000 IO
- Use smaller relation as outer relation
  - Again, difference irrelevant as tables get larger
- But sizes of relations do matter
  - If one relation fits into memory (b<m)
  - Total cost: b(R) + b(S)
  - One pass blocked-nested-loop

- We can do a little better with blocked-nested loop??

Zig-Zag Join

- When finishing a chunk of outer relation, hold last block of inner relation in memory
- Load next chunk of outer relation and compare with the still available last block of inner relation
- For each chunk, we need to read one block less
- Thus: Saves b(R)/m IO
  - If R is outer relation

Sort-Merge Join

- How does it work??
- What does it cost??
- Does it matter which is outer/inner relation??
- When is it better then blocked-nested loop??

Sort-Merge Join

- How does it work?
  - Sort both relations on join attribute(s)
  - Merge both sorted relations
- Caution if duplicates exist
  - The result size still is |R|*|S| in worst case
  - If there are r/s tuples with value x in the join attribute in R / S, we need to output r*s tuples for x
  - So what is the worst case??
- Example

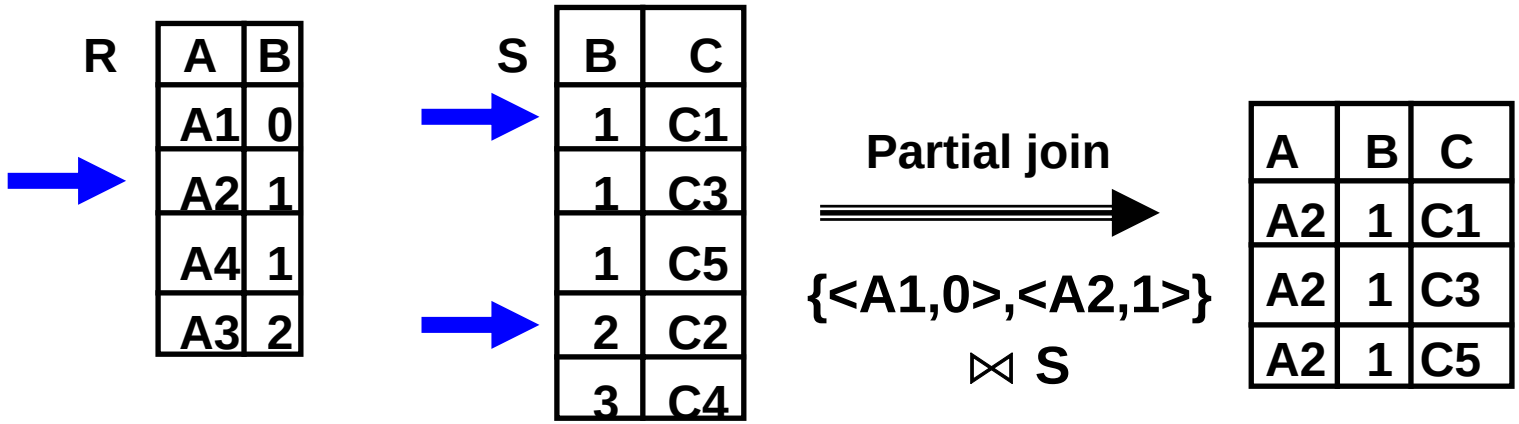
R

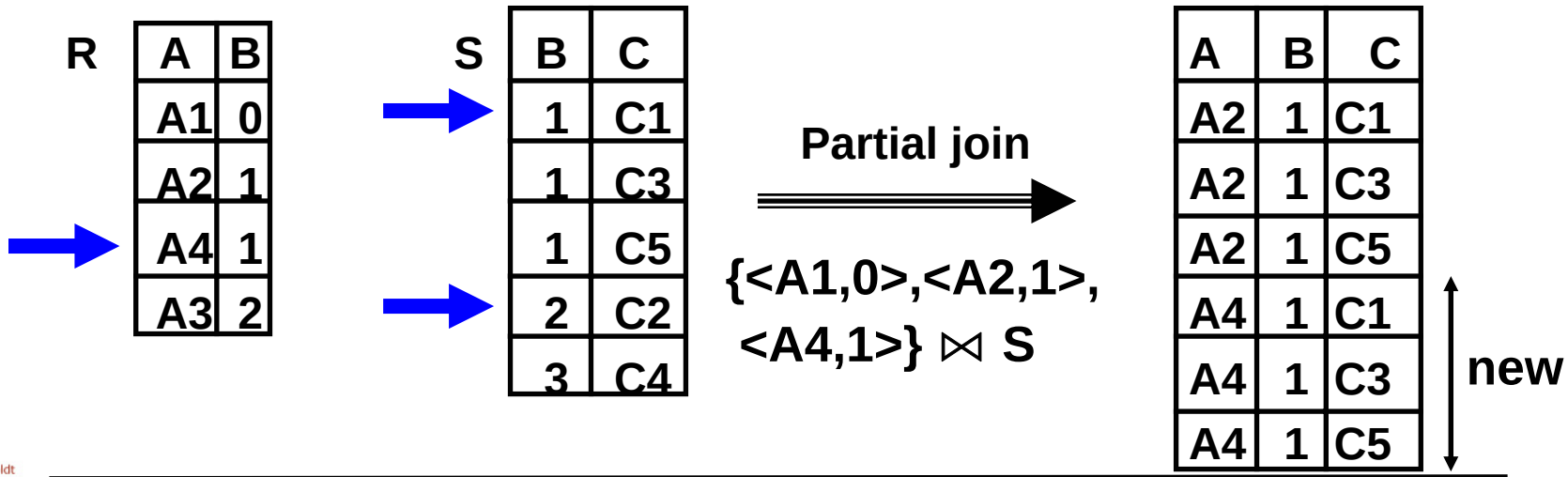
A	B
A1	0
A2	1
A4	1
A3	2

S

B	C
1	C1
1	C3
1	C5
2	C2
3	C4

Example Continued





Merge Phase

```
r := first (R);  s := first (S);
WHILE NOT EOR (R) and NOT EOR (S) DO
   IF r[B] < s[B] THEN r := next (R)
   ELSEIF r[B] > s[B] THEN s := next (S)
   ELSE                                  /* r[B] = s[B]*/
        b := r[B];  B := ∅;
        WHILE NOT EOR(S) and s[B] = b DO
             B := B ∪ {s};
             s = next (S);
        END DO;
        WHILE NOT EOR(R) and r[B] = b DO
             FOR EACH e in B DO
                   OUTPUT (r,e);
             r := next (R);
        END DO;
END DO;
```

- Can we improve pipeline behavior??

Cost estimation

- Sorting R costs 2*b(R) * ceil($\log_m$(b(R)))
- Sorting S costs 2*b(S) * ceil($\log_m$(b(S)))
- Merge phase reads each relation once
- Total IO
  - b(R) + b(S) + 2*b(R)*ceil($\log_m$(b(R))) + 2*b(S)*ceil($\log_m$(b(S)))
- Improvement
  - While sorting, do not perform last read/write phase
  - Open all sorted runs in parallel for merging
  - Saves 2*b(R) + 2*b(S) IO
- Sometimes, we are able to save the sorting phase
  - If sort is performed somewhere down in the tree
  - Needs to be a sort on the same attribute(s)
- Merge-sort join: No inner/outer relation

Better than Blocked-Nested-Loop?

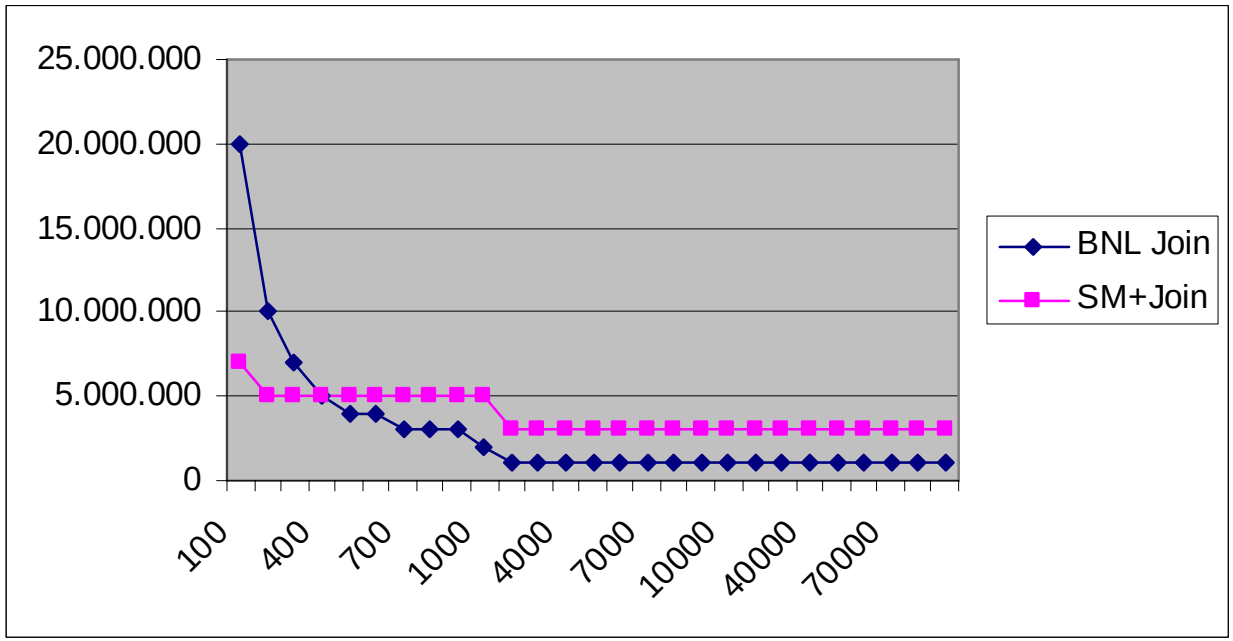
- Assume b(R)=10.000, b(S)=2.000, m=500
  - BNL costs 42.000
    - With S as outer relation
  - SM: 10.000+2.000+4*10.000+4*2.000 = 60.000
  - Improved SM: 36.000
- Assume b(R)=1.000.000, b(S)=1.000, m=500
  - BNL costs 1000 + 1.000.000*1000/500 = 2.001.000
  - SM: 1.000.000+1.000+6*1.000.000+4*1.000 = 7.005.000
  - Improved SM: 5.003.000
- When is SM better than BNL??
  - Consider improved version with
    - $2*b(R)*\text{ceil}(\log_m(b(R))) + 2*b(S)*\text{ceil}(\log_m(b(S))) - b(R) - b(S) \sim$
    - $2*b(R)*(\log_m(b)+1) + 2*b(S)*(\log_m(S)+1) - b(R) - b(S) \sim$
    - $2*b(R)*\log_m(b) + 2*b(S)*\log_m(S) - b(R) - b(S) \sim$
    - $b(R)*(2*\log_m(b)-1) + b(S)*(2*\log_m(S)-1)$
  - In most cases, this means 3*(b(S)+b(R))

Comparison

- Assume relations of equal size b
- SM: $2*b*(2*\log_m(b)-1)$
- BNL: $b+b^2/m$
- BNL > SM
  - $b+b^2/m > 2*b*(2*\log_m(b)-1)$
  - $1+b/m > 4*\log_m(b) - 2$
  - $b > 4m*\log_m(b) – 3m$
- Example
  - b=10.000, m=100 (10.000 > 500)
    - BNL: 10.000 + 1.000.000, SM: 6*10.000 = 60.000
  - b=10.000, m=5000 (10.000 < 25.000)
    - BNL: 10.000 + 20.000, SM: 6*10.000 = 60.000

Comparison 2

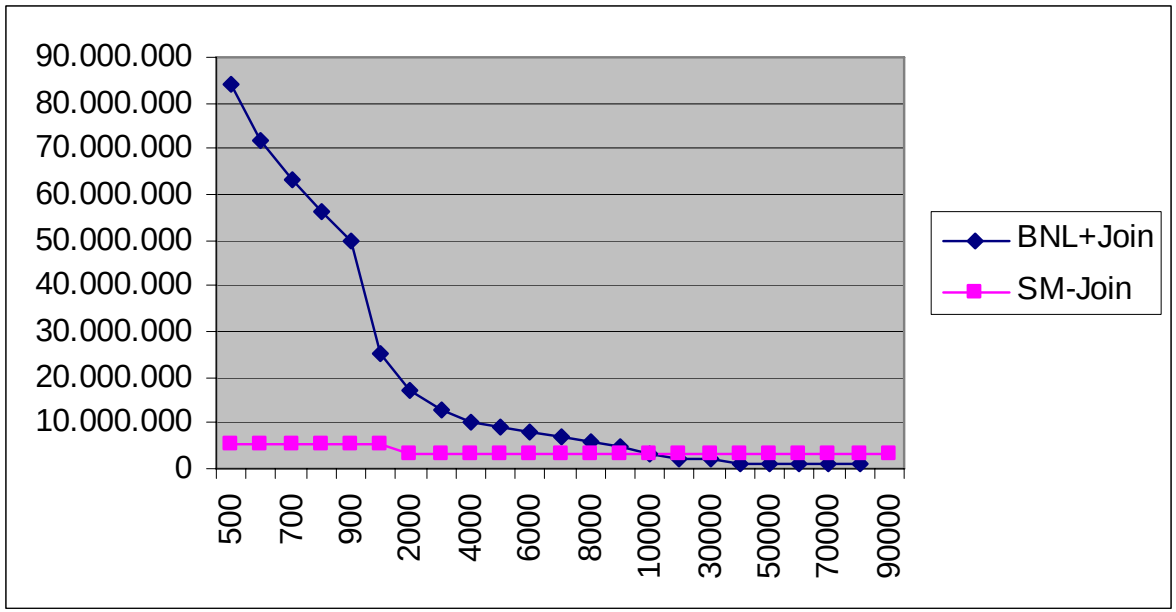
- b(R)=1.000.000, b(S)=2.000, m between 100 and 90.000



- BNL very good is one relation is much smaller than other and sufficient memory available
- SM can better cope with limited memory
- SM profit from more memory has jumps (= number of runs)

Comparison 3

- b(R)=1.000.000, b(S)=50.000, m between 500 and 90.000



- BNL very sensible to small memory sizes

Merge-Join and Main Memory

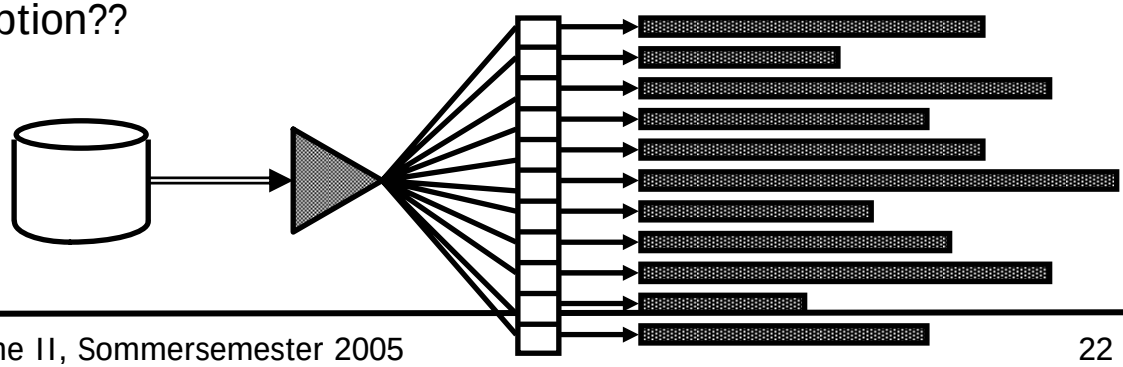
- We have no „m“ in the formula of the merge phase
  - Implicitly, it is in the number of runs required
- More memory doesn’t decrease number of blocks to read, but can be used for sequential reads
  - Always fill memory with m/2 blocks from R and m/2 blocks from S
  - Use asynchronous IO
    1. Schedule request for m/4 blocks from R and m/4 blocks from S
    2. Wait until loaded
    3. Schedule request for next m/4 blocks from R and next m/4 blocks from S
    4. Do not wait – perform merge on first 2 chunks of m/4 blocks
    5. Wait until previous request finished
       1. We used this waiting time very well
    6. Jump to 3, using m/4 chunks of M in turn

Hash Join

- As always, we may save sorting if good hash function available
- Assume a very good hash function
  - Distributes hash values almost uniformly over hash table
  - If we have good histograms (later), a simple interval-based hash function will usually work
- How can we apply hashing to joins??

Hash Join Idea

- Use join attributes as hash keys in both R and S
- Choose hash function for hash table of size m
  - Each bucket has size b(R)/m, b(S)/m
- Hash phase
  - Scan R, compute hash table, writing full blocks to disk immediately
  - Scan S, compute hash table, writing full blocks to disk immediately
  - Notice: Probably better to use some n<b(R)/m to allow for sequential writes
- Merge phase
  - Iteratively, load same bucket of R and of S in memory
  - Compute join
- Total cost
  - Hash phase costs 2*b(R)+2*b(S)
  - Merge phase costs b(R) + b(S)
  - Total: 3*(b(R)+b(S))
  - Under what assumption??



Hash Join with Large Tables

- Merge phase assumes that 2 buckets can be hold in memory
  - Thus, we roughly assume that 2*b(R)/m<m (if b(R)~b(S))
  - Note: Merge phase of sorting only requires 2 blocks (or more for more runs), hashing requires 2 buckets to be loaded
- What if $b(R)>m^2/2$??
  - We need to create smaller buckets
  - Partition R/S such that each partition is smaller than $m^2/2$
  - Compute buckets for all partitions in both relations
  - Merge in cross-product manner
    - $P_{R,1}$ with $P_{S,1}$, $P_{S,2}$, ..., $P_{S,n}$
    - $P_{R,2}$ with $P_{S,1}$, $P_{S,2}$, ..., $P_{S,n}$
    - ...
    - $P_{R,m}$ with $P_{S,1}$, $P_{S,2}$, ..., $P_{S,n}$
- Actually, it suffices if either b(R) or b(S) is small enough
  - Chose the smaller relation as driver (outer relation)
  - Load one bucket into main memory
  - Load same bucket in other relation block by block and filter tuples

Hybrid Hash Join

- Assume that min(b(R),b(S)) < $m^2/2$
- Notice: During merge phase, we used only (b(R)+b(S))/m (size of two buckets) memory blocks
  - Although there are much more
- Improvement
  - Chose smaller relation (assume S)
  - Chose a number k of buckets to build (with k<m)
    - Again, assuming perfect hash functions, each bucket has size b(S)/k
  - When hashing S, keep first x buckets completely in memory, but only one block for each of the (k-x) other buckets
    - These x buckets are never written to disk
  - When hashing R
    - If hash value maps into buckets 1..x, perform join immediately
    - Otherwise, map to the k-x other buckets/blocks and write buckets to disk
  - After first round, we have performed the join on x buckets and have k-x buckets of both relations on disk
  - Perform “normal” merge phase on k-x buckets

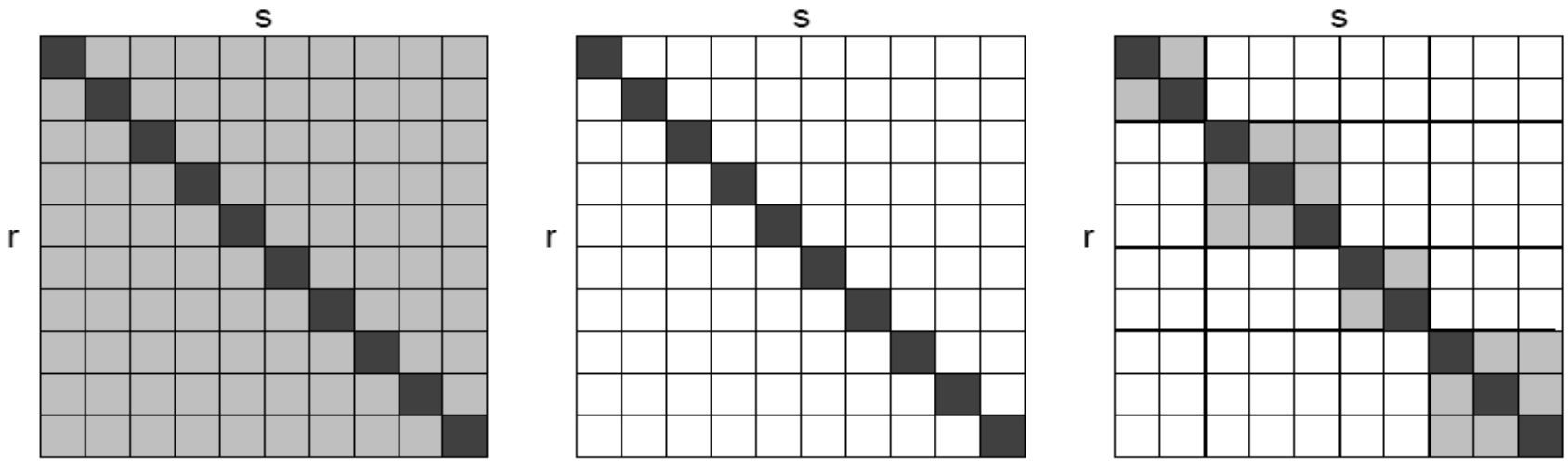
Hybrid Hash Join - Complexity

- Total saving (compared to normal hash join)
  - We save 2 IO (writing) for every block that is never written to disk
  - We keep x buckets in memory, each having b(S)/k and b(R)/k blocks, respectively
  - Together, we save 2*x/k*(b(S)+b(R)) IO operations
- Question: How should we choose k and x?
- Optimal solution
  - x=1 and k as small as possible
  - Build as large partitions as possible, such that still one entire partition and one block for all other partitions fits into memory
  - Thus, we use as much memory as possible for savings
  - Optimum reached at approximately k=b(S)/m
    - k must be smaller, so that M can accommodate 1 block for each other bucket
- Together, we save 2m/b(S)*(b(S)+b(R))
- Total cost: (3-2m/b(S))*(b(S)+b(R))

Comparing Hash Join and Sort-Merge Join

- If enough memory provided, both require approximately the same number of IOs
  - 3*(b(R)+b(S))
  - Hybrid-hash join improves slightly
- SM generates sorted results – sort phase of other joins in query plan can be dropped
  - Advantage propagates up the tree
- HJ does not need to perform O(n*log(n)) sorting in main memory
- HJ requires that only one relation is “small enough”, SM needs two small relations
  - Because both are sorted independently
- HJ depends on roughly equally sized buckets
  - Otherwise, performance might degrade due to unexpected paging
  - To prevent, estimate k more conservative and do not fill m completely
  - Some memory remains unused
- Both can be tuned to generate mostly sequential IO

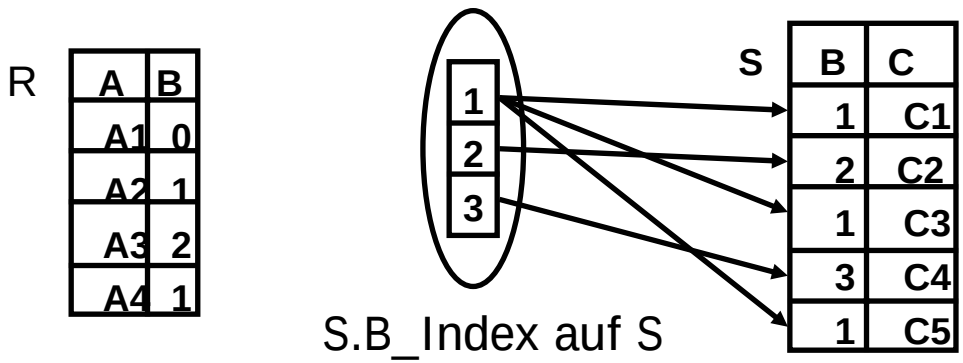
Comparing Join Methods



Index Join

- Assume we have an index “B_Index” on one join attribute
- Choose indexed relation as inner relation
- Index join

```
FOR EACH r IN R DO
     X = { SEARCH (S.B_Index, <r.B>) }
     FOR EACH TID i in X DO
         s = READ (S, i) ; output (r ⋈ s).
```



- Actually, this is a one block-nested loop with index access
  - Using BNL possible (and better)

Index Join Cost

- Assumptions
  - R.B is foreign key referencing S.B
  - Every tuple from R has one or more join tuples in S
- Let v(X,B) be the number of different values of attribute B in X
  - Each value in S.B appears v~b(S)/v(S,B) times
- For each $r \in R$, we read all tuples with given value in S
- Total cost: $b(R) + |R|*(\log_k(S/b) + v/k + v)$
  - Outer relation read once
  - Find value in B*, read all matching TIDs, access S for each TID
- Other way round: Assume that S.B is foreign key for R.B
  - Some tuples of R will have no join partner in S
  - Assume only r R tuples have partner
- Total cost: $b(R) + r*(\log_k(S/b) + v/k + v)$
  - No real change

Index Join Cost

- Comparison to sort-merge join
  - Neglect $\log_k(S/b) + v/k$
    - First term is only ~2, second ~1 in many cases
  - SM > IJ roughly requires
    - Assuming that 2 passes suffice for sorting
    - 3*(b(R)+b(S)) > b(R)+|R|*b(S)/v(S,B)
- Example
  - b(R)=10.000, b(S)=2.000, m=500, v(S,B)=10, k=50
  - SM: 36.000
  - IJ: 10.000 + 10.000*50*2.000/10 ~ 1.000.000.000
- When is an index join a good idea??

Index Join: Advantageous Situations

- When r is very small
  - If join is combined with selection on R
  - Most tuples are filtered, only very few accesses to S
  - Index pays off
- When R is very small, R.B is foreign key, S.B is primary key
  - Similar to previous case
  - If S is primary key, then v(S,B)=|S|, and hence v=1
  - R can be read fast and “probes” into S
  - We get total cost of ~b(R)+|R| (plus index access etc.)

Index Join with Sorting

- Problem with last approach: Blocks of S are read many times
  - Caching will reduce the overhead – difficult to predict
- Alternative
  - First compute all necessary TID's from S
  - Sort and read in sorted order
  - Advantage: blocks of S will mostly be in cache when accessed
  - Requires enough memory for keeping TID list and tuples of R
- Further improvement
  - Perform Sort-Merge join on B values only
  - B values from S are already sorted (if B*-index) is used
  - Hence: Sort R on B, merge lists, probe into S for required values
  - Can be combined with previous improvement

Index Join with 2 Indexes

- Assume we have an index on both join attributes
- What are we doing??

Index Join with 2 Indexes

- TID list join
  - Read both indexes sequentially (order does not matter)
  - Join (value,TID) lists
  - Probe into R and S
  - Large advantage, if intersection is small
  - Otherwise, we need sorted tables (index-organized)
    - But then sort-merge is probably faster

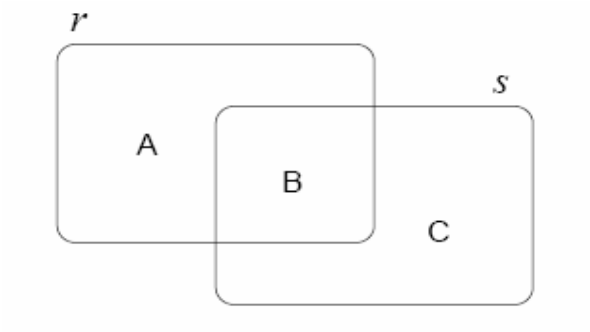
Semi Join

- Consider queries such as
  - `SELECT DISTINCT R.* FROM S,R WHERE R.B=S.B`
  - `SELECT R.* FROM R WHERE R.B IN (SELECT S.B FROM S)`
  - `SELECT R.* FROM R WHERE R.B IN ( …)`
- What's special?
  - No values from S are requested in result
  - S (or inner query) acts as filter on R
- Semi-Join $R \ltimes S$
  - Very important for distributed databases
    - Accessing data becomes even more expensive
    - Idea: First ship only join attribute values, compute Semi-Join, then retrieve rest of matching tuples
    - Technique also can be used for very large tuples and small result sizes
      - First project on attribute values
      - Intersect lists, probe into tables and load data
      - Question: How do we know the sizes of intermediate result?

Implementing Semi-Join

- Using blocked-nested-loop join
  - Chose filter relation as outer relation
  - Perform BNL
  - Whenever partner for R.B is found, exit inner loop
- Using sort-merge join
  - Sort R
  - Sort join attribute values from S, remove duplicates on the way
  - Perform merge phase as usual
  - Very effective if v(S,B) is small

Union, Difference, Intersection



Ergebnis-extensionen	Übereinstimmung auf allen Attributen	Übereinstimmung auf einigen Attributen
A	Differenz $r - s$	Anti-Semi-Verbund
B	Schnitt $r \cap s$	Verbund, Semi-Verbund
C	Differenz $s - r$	Anti-Semi-Verbund
$A \cup B$		Left Outer Join
$A \cup C$	symmetrische Differenz $(r - s) \cup (s - r)$	Anti-Verbund
$B \cup C$		Right Outer Join
$A \cup B \cup C$	Vereinigung $r \cup s$	Full Outer Join

- Other binary operations use methods similar as those for joins
  - Sorting, hashing, index, ...
- See Garcia-Molina et al. for detailed discussion