Datenbanksysteme II: Implementing Joins

Ulf Leser

Content of this Lecture

- Nested loop and blocked nested loop
- Sort-merge join
- Hash-based join strategies
- Index join

Join Operator

- Join: Highly time-critical operator
  - Required in virtually all queries and in all applications
  - Often appears in groups (multi-way joins – much theory)
  - Problem: May create very large results
    - Estimating result size is difficult, especially in multi-way settings
  - Relational operator with non-linear WC runtime: O(n*m)
  - Many variations, suited for different situations

A	B
A1	0
A2	1
A3	2
A4	1

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

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

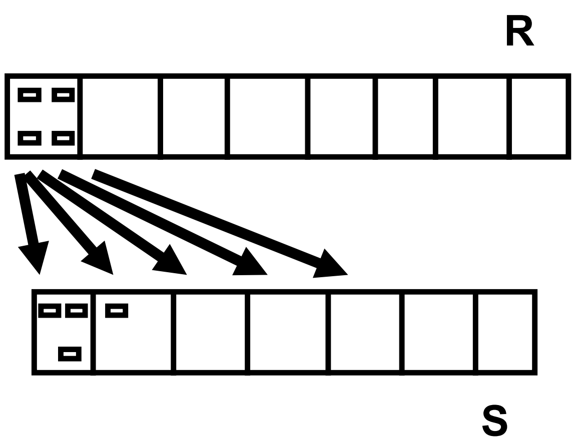
$R \bowtie S$

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

Implementation 1: Nested-loop Join

- Super-naïve

```
FOR EACH r IN R DO
  FOR EACH s IN S DO
     LOAD block(r) into M;
     LOAD block(s) into M;
     IF (r.B=s.B) THEN OUTPUT (r ⋈ s)
```



- Obvious improvement

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

Cost Estimation

- Let b(R), b(S) be 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 ops)
- Altogether IO: b(R)+b(R)*b(S)

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
- But choice doesn't really matter here …
- Can't we do better?

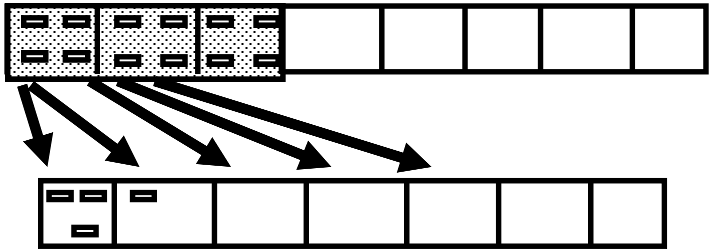
Observation

- There is no “m” in the formula
  - m: Size of main memory in blocks
- We are not using our available main memory
  - Only two blocks for reading and one for writing
- Rule of thumb: Use all memory you can get
  - Use all memory the buffer manager allocates to your process

Implementation 2: Blocked Nested-Loop Join

- 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
    READ BLOCK y into M
    FOR EACH r in R-chunk DO
      FOR EACH s in y do
        IF (r.B=s.B) THEN OUTPUT (r ⋈ s)
```



Cost

- Outer relation is read once – in chunks
- Inner relation is read once for every chunk of R
- There are ~b(R)/m chunks
- Total IO: b(R) + b(R)*b(S)/m
- Further advantage: Chunks of outer relation are read sequentially

Example

- Assume b(R)=10.000, b(S)=2.000, m=500
- R as outer relation: 10.000 + 10.000*2.000/500 = 50.000
- S as outer relation: 2.000 + 2.000*10.000/500 = 42.000
- Again: Use smaller relation as outer relation
- 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 the 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

Content of this Lecture

- Nested loop and blocked nested loop
- Sort-merge join
- Hash-based join strategies
- Index join

Sort-Merge Join

- Sort both relations on join attribute(s)
- Merge both sorted relations
- Caution if join values appear multiple times
  - The result size is |R|*|S| in worst case
  - If there are r and s tuples with value x in the join attribute in R and S, respectively, we need to output r*s tuples for x

Example

A	B
A1	0
A2	1
A3	2
A4	1

A	B
A1	0
A2	1
A4	1
A3	2

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

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

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

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;
```

Code ignores other than join attributes

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: $b(R) + b(S) + 2*b(R)*ceil(\log_m(b(R))) + 2*b(S)*ceil(\log_m(b(S)))$
- 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
- If sort was performed already somewhere down in the tree, sort phase can be skipped

Better than Blocked-Nested-Loop?

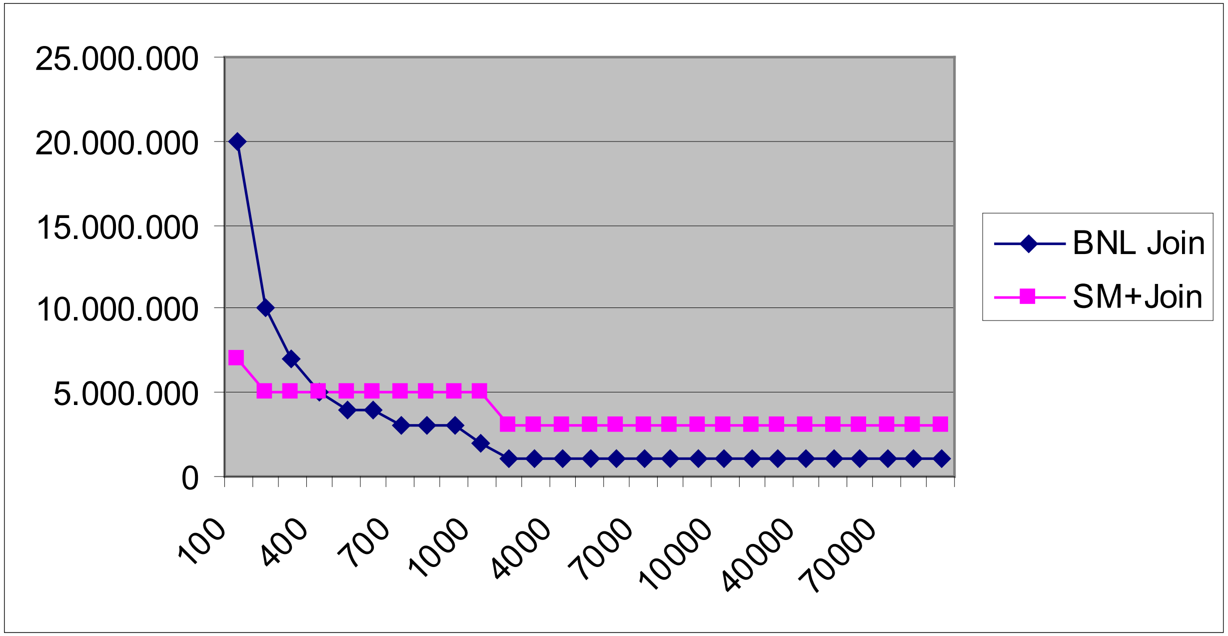
- Assume b(R)=10.000, b(S)=2.000, m=500
  - BNL costs 42.000 (with S as outer relation)
  - SM: 10.000+2.000+4*10.000+4*2.000 = 60.000
  - Improved SM: 36.000
- Assume b(R)=1.000.000, b(S)=1.000, m=500
  - BNL costs 1000 + 1.000.000*1000/500 = 2.001.000
  - SM: 1.000.000+1.000+6*1.000.000+4*1.000 = 7.005.000
- When is SM better than BNL?
  - Consider improved version with
    - $2*b(R)*ceil(\log_m(b(R))) + 2*b(S)*ceil(\log_m(b(S))) – b(R) - b(S) \sim$
    - $2*b(R)*(\log_m(b(R))+1) + 2*b(S)*(\log_m(S)+1) – b(R) – b(S) =$
    - $2*b(R)*\log_m(b(R)) + 2*b(S)*\log_m(S) + b(R) + b(S) \sim$
    - $b(R)*(2*\log_m(b(R))+1) + b(S)*(2*\log_m(S)+1)$
  - Compare to BNL: b(R) + b(R)*b(S)/m

Comparison

- Assume two relations of equal size b
- SM: $2*b*(2*\log_m(b)+1)$
- BNL: $b+b^2/m$
- BNL > SM iff
  - $b+b^2/m > 2*b*(2*\log_m(b)+1)$
  - $1+b/m > 4*\log_m(b) + 2$
  - $b > 4m*\log_m(b) + m$
- Example
  - b=10.000, m=100
    - BNL: 10.000 + 1.000.000, SM: 6*10.000 = 60.000
  - b=10.000, m=5.000
    - BNL: 10.000 + 20.000, SM: 6*10.000 = 60.000

Comparison 2

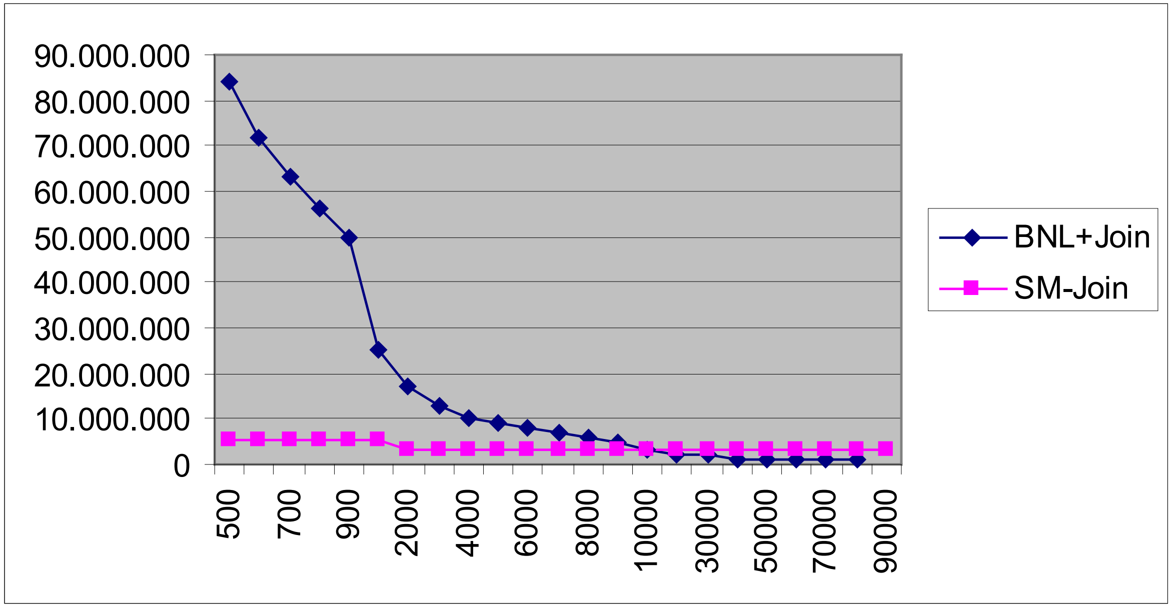
- b(R)=1.000.000, b(S)=2.000, m between 100 and 90.000



  - BNL very good if one relation is much smaller than other and sufficient memory available (~1 pass suffices)
  - SM can better cope with limited memory

Comparison 3

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



- BNL very sensible to small memory sizes

Content of this Lecture

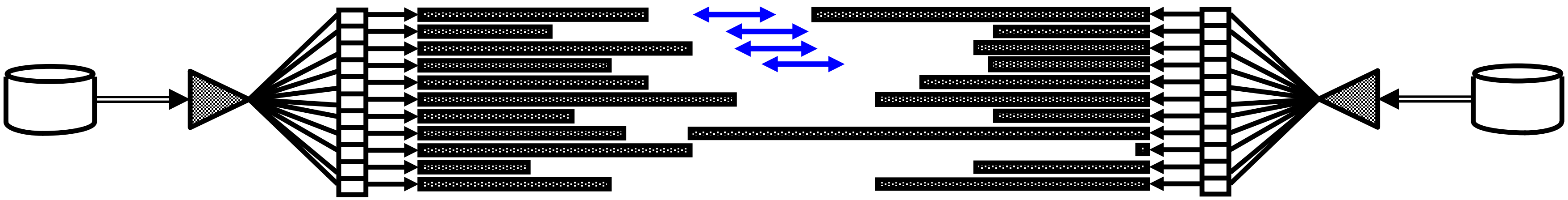
- Nested loop and blocked nested loop
- Sort-merge join
- Hash-based join strategies
- Index join

Hash Join

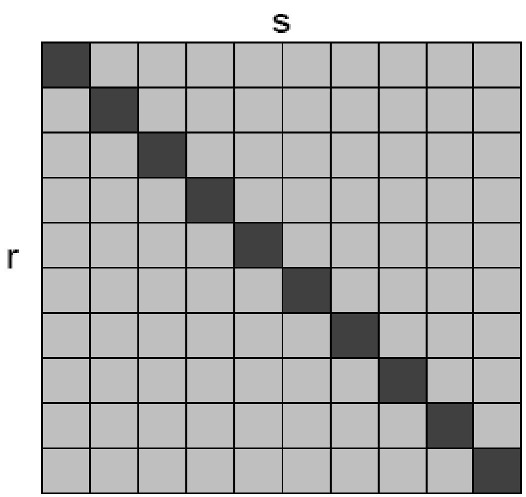
- As often, we can replace sorting with a good hash function
- Assume a very good hash function
  - Distributes hash values uniformly over hash table
  - If we have good histograms (later), a simple interval-based hash function can do the trick
- How can we apply hashing to joins?

Idea

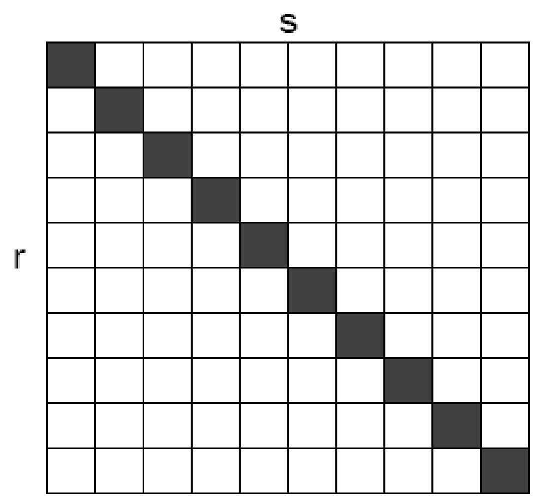
- Use join attribute(s) as hash keys in both R and S
  - Assume hash table of size m (use all memory)
  - Each bucket will have size approx. b(R)/m or b(S)/m
- Hash phase
  - Scan R, add to bucket, writing full blocks to disk immediately
  - Scan S, add to bucket, writing full blocks to disk immediately
  - [Better to use some n<b(R)/m to allow for sequential writes]
- Join phase
  - Iteratively, load same buckets of R and of S (assume we can)
  - Compute join in memory



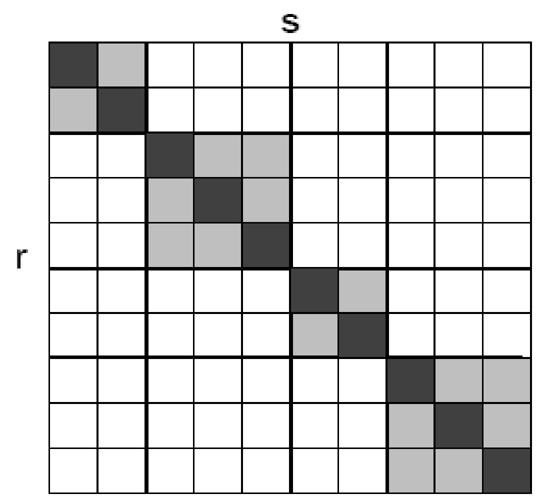
Comparing Join Methods



Nested-Loops-Join



Merge-Join



Hash-Join

Cost

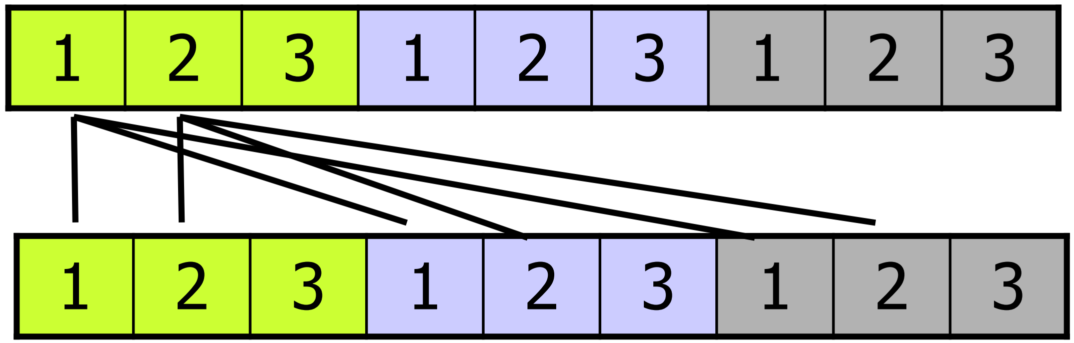
- Assume we can always load both buckets into main memory during the join phase
- Hash phase: 2*b(R)+2*b(S)
- Join phase: b(R) + b(S)
- Total: 3*(b(R)+b(S))

- What happens if hash function creates skew?
  - Some buckets will be very large, others very small
  - We cannot any more assume to load both join buckets into memory
  - Note: Merge phase of sorting requires |runs| blocks (where runs have equal and fixed size), hashing requires 2 buckets to be loaded (where buckets need not have equal and restricted size)

Partitioned Hash Join

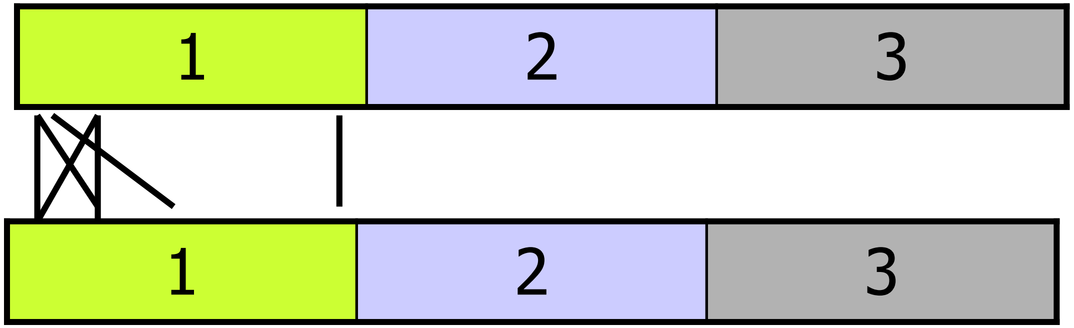
- Two phase hash join: First partition R and S such that each partition most likely has buckets that are small enough
- Compute buckets for all partitions in both relations
- Merge in cross-product manner
  - P_{ABC}: Relation A, partition B, hashkey C
  - $P_{R,1,1}$ with $P_{S,1,1}$, $P_{S,2,1}$, …, $P_{S,n,1}$
  - $P_{R,2,1}$ with $P_{S,1,1}$, $P_{S,2,1}$, …, $P_{S,n,1}$
  - …
  - $P_{R,m,k}$ with $P_{S,1,k}$, $P_{S,2,k}$, …, $P_{S,n,k}$

Cost (with Partioning)



- Assume $b(R)=b(S)=b$
- How many partitions (p) do we need (if buckets are of equal size)?
  - Goal: For each partition P, $b(P)/m<m/2$, or $b(P)<m^2/2$
  - Hence: $b/p \sim m^2/2$, or $p \sim 2*b/m^2$
- For each partition, there will be m buckets of size $\sim m/2$
- Hash/partition phase: 2b+2b (partitions are not materialized)
- Merge phase: $b + p*m * p*m/2 = b+ p^2*m^2/2 = b + 2b^2/m^2$
  - There are p*m buckets in outer relation
  - For each bucket of outer relation, we have to read p buckets of inner relation, each of size m/2

Alternative



- Accept overly large buckets
- Perform blocked-nested loop for each pair of buckets
- There are m buckets, each of size n=b/m (>m/2)
- Hash phase: 2b+2b
- BNL phase: $m * (n + n*n/m) = m*(b/m+b^2/m^3) = b+b^2/m^2$
  - There are m bucket pairs
  - For each, we perform blocked nested loop over two buckets of size n
- But: More sensitive to hash function; worst case much worse
  - n can approach b with only on1 non-empty bucket pair; this results in $1(b+b^2/m)$

Improvement

- Actually, it suffices if either b(R) or b(S) is small enough
- Load buckets of smaller relation into main memory
  - And sort for faster look-up
- Load same bucket in other relation block by block and filter tuples

Hybrid Hash Join

- Assume that $\min(b(R),b(S))<m^2/2$
- Note: During merge phase, we used only $(b(R)+b(S))/m$ memory blocks (size of two buckets)
- This usually does not fill the entire memory
- Improvement
  - Choose smaller relation (assume S)
  - Choose a number k of buckets (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 first x buckets are never written to disk

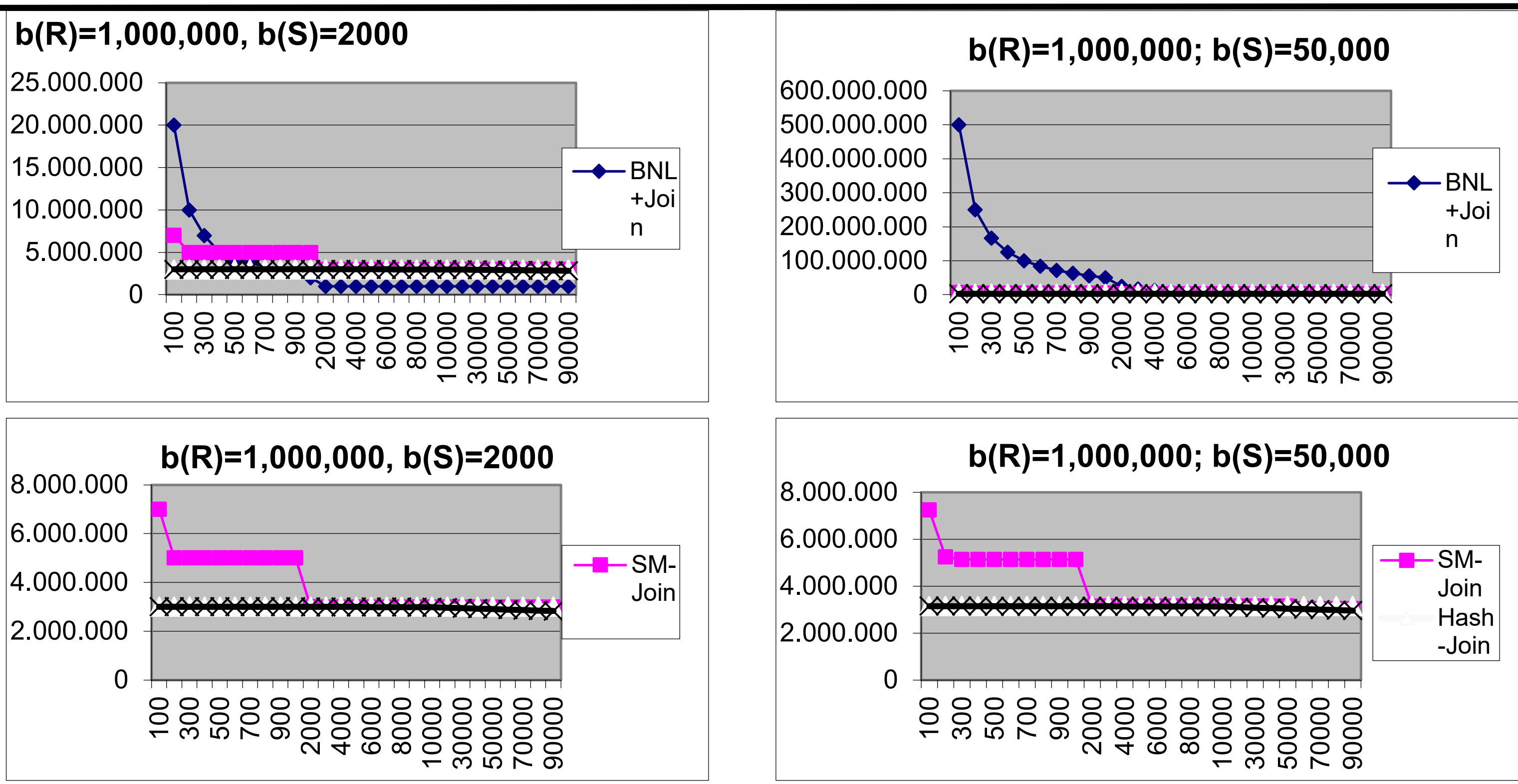
Continued

- …
- When hashing R
  - If hash value maps into buckets 1..x, perform join immediately
  - Otherwise, map to the k-x other buckets and write to disk
- After first round, we have already computed the join on x buckets and have k-x buckets of both relations on disk
- Perform “normal” merge phase on k-x buckets

Cost

- Total saving (compared to normal hash join)
  - We save 2 IO for every block in either relation that is never written
  - We keep x buckets in memory, having ~ b(S)/k and ~b(R)/k blocks
  - Together, we save 2*x*(b(S)+b(R))/k IO operations
- How should we choose k and x?
- Best solution: x=1 and k as small as possible
  - Build buckets as large as possible, such that still one entire bucket and one block for all other buckets fits into memory
  - Optimum reached at k ~ b(S)/m
    - Note: k must be a little smaller: One block for each other bucket
- Together, we save 2*(b(S)+b(R))*m/b(S)
- Total cost: (3-2m/b(S))*(b(S)+b(R)) = 6b-4m
  - With b=b(R)=b(S)

Quantitative Comparison



- BNLJ sensitive to memory and size differences
- HJ (under certain assumptions) with robust performance
  - Sometimes better, sometimes worse than SMJ
  - Insensitive to changing memory or size differences

Comparing Hash Join and Sort-Merge Join

- With enough memory, both require approximately the same number of IO
  - Hybrid-hash join improves slightly
- SM generates sorted results – sort phase of other joins in query plan can be dropped, entire queries get faster
- HJ: No need to perform sorting of runs in main memory
- HJ only requires that one relation is “small enough”
- HJ only performs well if we have equally sized buckets
  - Otherwise, performance might degrade due to unexpected paging
  - To prevent, estimate k conservative and do not fill m completely
- Both can be tuned to generate more sequential IO

Content of this Lecture

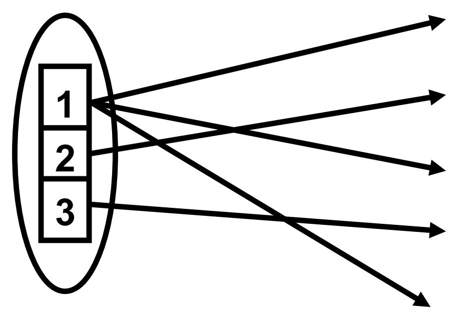
- Nested loop and blocked nested loop
- Sort-merge join
- Hash-based join strategies
- Index join

Index Join

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

```
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).
```

A	B
A1	0
A2	1
A3	2
A4	1



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

- Nested loop with index access

Cost

- Typical situation: R.B is primary key, S.B is foreign key
  - Every tuple from R has zero, one or more join tuples in S
- Let v(X,B) be number of unique values of B in relation X
  - Each value in S.B appears v~|S|/v(S,B) times
- For each $r \in R$, we need all tuples with given value in S
- Assume every r has at least one join partner (k~|block|): $b(R) + |R| * (\log_k(|S|) + v/k + v)$
  - Outer relation read once
  - Find value in B*-tree index, read all matching TIDs (with block size k), access S for each TID (assume they are all in different blocks)
- Assume only l tuples of R have partner: $b(R) + |R| * \log_k(|S|) + l(v/k + v)$

Comparison

- Compare to sort-merge join
  - Neglect $\log_k(|S|) + v/k$
    - First term is mostly ~2, second mostly ~1
  - SM > IJ roughly requires
    - Assume 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 really small
  - The join is highly selective – few tuples find a partner
  - For instance, if join is combined with selection on R
  - Most tuples are filtered, only very few require access to S
- 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

Index Join with Sorting

- Note: 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 tuples from S in sorted order
    - Sort by TID and hope that tuples didn't move too often and TIDs are created in sequential order
  - Advantage: Blocks of S more often will be in cache when accessed
  - Requires enough memory for keeping TID list and join tuples of R
  - Pipeline breaker

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
- Join (value,TID) lists on value
- Probe into R and S only if necessary
- Large advantage if intersection is small
  - Because indexes are much more compact than data blocks and data blocks are almost never accessed
- Otherwise, we need sorted tables (index-organized)
  - But then sort-merge is probably faster