Algorithms of Relational Algebra
Join Algorithms

Unary versus Binary Operations

- Relational operators working on one table
  - Selection (scanning, indexes)
  - projection (duplicate elimination -> use sorting or hashing)
  - O(1), O(log(N)), O(N), O(Nlog(N))
- On two tables
  - Product, Join, Semi-join, Intersection, Union, Difference
- Binary operators are usually more expensive
  - Binary: Look at each tuple of first table for each tuple of second table (naïve algorithm)
  - "Potentially" quadratic complexity in time and space O(N^2)

Unary relational operations

- Relation R; b(R) denotes number of blocks to hold records in R
- Selection

  case (1) A = ‘a’ index lookup or scanning

  case (2) A <> | <= | < | >= | > ‘a’ B+-tree or scanning

- Projection

  sorting followed by duplicate elimination: O(b(R)log(b(R))

  hashing followed by duplicate elimination: O(b(R))

Observation: output size is never larger than O(b(R))

Binary relational algebra operations

- Relation R; b(R) denotes numbers of blocks to hold records in R
  Relation S; b(S) denotes number of blocks to hold records in S

- Union, intersection, difference, join, semi-join, anti semi-join

- Naïve algorithm: Double nested loop algorithm

```
for r in R
    for s in S  {…}
```

- Time complexity O(b(R)b(S))
- Space complexity:
  O(b(R) + b(S)) for union, intersection, difference, semi-join, anti semijoin
  possibly O(b(R)b(S)) for join

Join Operator

- JOIN: Most important relational operator
  - Potentially very expensive
  - Required in all practical queries and applications
  - Often appears in groups of joins

- Example: Relations R (A, B) and S (B, C)

  SELECT *
  FROM R JOIN S ON (R.B θ S.B)
  with θ: $=, \neq, <, \leq, >, \geq$

Overview of Join Algorithms

- Nested-loop and block nested-loop join $R \bowtie_\theta S$
- Sort-merge join $R \bowtie S$
- Hash-based join strategies $R \bowtie S$
- Index join $R \bowtie S$

Nested-loop Join

Nested-loop join

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

Some improvement (block-based)

```
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 θ s.B) THEN OUTPUT (r ⋈θ s)
```

- Cost estimations
  - b(R), b(S) number of blocks in R and in S, respectively
  - 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)

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 relations, choice doesn’t really matter

- Can we do better?

…

- There is no “M” in the formula
  - M is the size of main memory (buffer) in blocks
- We should use our available main memory (buffer)
- We will sometimes need a somewhat different parameter than M to discuss buffer size.

Block nested-loop join

- Rule of thumb: Use all memory you can get
  - Use all memory the buffer manager allocates to the process

- Blocked-nested-loop

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

- 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
- But sizes of relations do matter:
  - If one relation fits into memory (min(b(R),b(S)) < M)
  - Total cost: b(R) + b(S)

Sort-Merge Join

- How does it work?
- Only works for Natural Join $R \bowtie S$
- What does it cost?
- Does it matter which is outer/inner relation?
- When is it better then block-nested loop?
- Be concerned about skew on join attribute

Sort-Merge Join

- How does it work?
  - Sort both relations on join attribute(s)
  - Merge both sorted relations
- Caution if duplicates (skew) exists
  - 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
  - More importantly, all these r/s must simultaneously fit in main memory (not always true if there is skew)

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

Partial join

{<A1,0>,<A2,1>} $\bowtie$ S

A	B	C
A2	1	C1
A2	1	C3
A2	1	C5

R

A	B
A1	0
A2	1
A4	1
A3	2

S

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

Partial join

{<A1,0>,<A2,1>, <A4,1>} $\bowtie$ S

A	B	C
A2	1	C1
A2	1	C3
A2	1	C5
A4	1	C1
A4	1	C3
A4	1	C5

new

Cost estimation (without skew)

- 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)))

This is only the case when, for each join-value b, the R and S blocks with that value b fit together in the buffer.
If this is not the case for value b, then the sort-merge algorithm needs to do a local block-nested join loop on the R and S blocks with value b.

Better than Block-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
    since $\text{ceil}(\log_{500}(10,000)) = \text{ceil}(\log_{500}(2,000)) = 2$

- Assume b(R)=1,000,000, b(S)=1,000, M=500
  - BNL costs 1,000 + 1,000,000*1000/500 = 2,001,000
  - SM: 1,000,000+1,000+6*1,000,000+4*1,000 = 7,005,000

Comparison

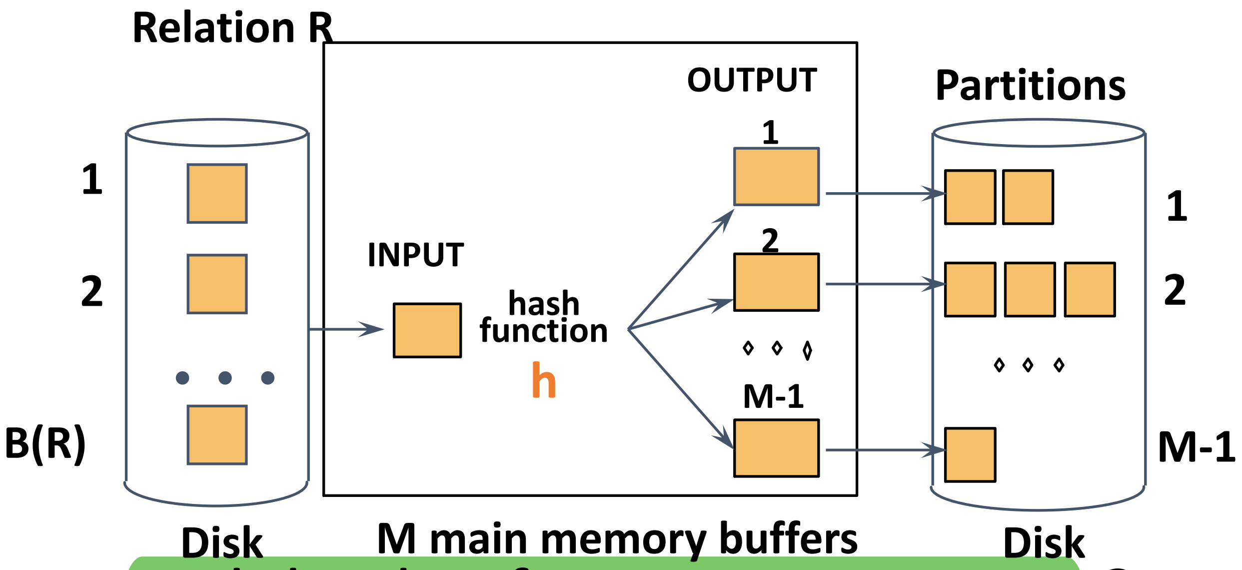
- Assume relations of equal size: B(R) = B(S) = B blocks
- SM: $2*B + 4*B*\log_M(B)$
- BNL: $B+B^2/M$
- BNL > SM
  – $B+B^2/M > 2*B + 4*B*\log_M(B)$
  – $B/M > 1+ 4*\log_M(B)$ (division by B)
  – $B > M + 4*M*\log_M(B)$
  - $M^k >$ M + 4*M*k = (1+4k) M (when B = M^k)

Hash Join

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

Hashing a file on join attribute(s)

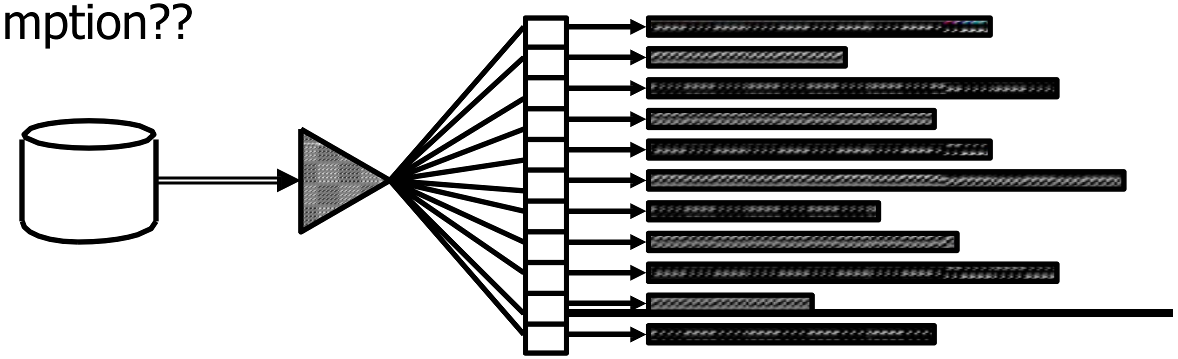
- Idea: partition a relation R into buckets, on disk
- Each bucket has size approx. B(R)/M



- Does each bucket fit in main memory ?
  - Yes if $B(R)/M \le M$, i.e. $B(R) \le M^2$

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
- 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??



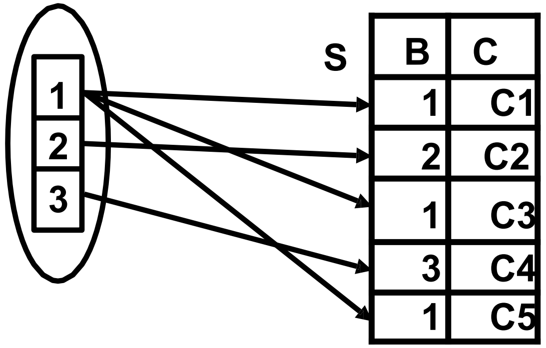
Index Join

- 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).
```

R

A	B
A1	0
A2	1
A3	2
A4	1



S

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

S.B_Index auf S

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

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)
- 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

Implementing Semi-Join

- Using blocked-nested-loop join
  - Choose relation R 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
- Using hash join
  - Hash R
  - Hash join values from S, remove duplicates on the way
  - Perform hash phase as usual

Implementing Intersection, Union, Difference

- Analogous with semi-join
- Using sort-merge join
  - Sort R, remove duplicates on the way
  - Sort S, remove duplicates on the way
  - Perform merge phase as usual (checking for and, or, not)
- Using hash join
  - Hash R, remove duplicates on the way
  - Hash R, remove duplicates on the way
  - Perform hash phase as usual (checking for and, or, not)

Observe that the performance is never O(R*S)