Query Evaluation
-- Join operation

References:

- [SKS-6ed] Chapter 12.5
- [RG-3ed] Chapter 14.4

Relational Operations

- We will consider how to implement:
  - Selection (σ): Selects a subset of rows from relation.
  - Projection (π): Deletes unwanted columns from relation.
  - Join ($\bowtie$): Allows us to combine two relations.
  - Set-difference (-): Tuples in relation 1, but not in relation 2.
  - Union (∪): Tuples in relation 1 and in relation 2.
  - Aggregation (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be composed! After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Similar to old schema; *rname* added for variations.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.

Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid = S1.sid
```

- In algebra: R ⋈ S. Common! Must be carefully optimized.
- R×S is large. So, R × S followed by a selection is inefficient.
- We will consider more complex join conditions later.
- Cost metric: # of I/Os. We will ignore output costs.

Join Operation

- Several different algorithms to implement joins
  - Simple nested-loop join: iteration
  - Block nested-loop join: iteration
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Our examples use the following information
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of pages of *student*: 100 *takes*: 400

Simple nested-Loop Join

- To compute the theta join $r \bowtie_\theta s = \sigma_\theta(r \times s)$

```
for each tuple $t_r$ in $r$ do begin
    for each tuple $t_s$ in $s$ do begin
        test pair $(t_r,t_s)$ to see if they satisfy the join condition $\theta$
        if they do, add $t_r \cdot t_s$ to the result.
    end
end
```

- *r* is called the **outer relation** and *s* the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Simple nested-Loop Join (Cont.)

- Given
  - n_r , b_r : number of tuples and pages in r
  - n_s , b_s : number of tuples and pages in s
- Case 1: worst case, memory hold one page of each relation
  - $b_r + n_r * b_s$

Simple nested-Loop Join (Example)

- Number of records of *student*: 5,000 *takes*: 10,000
- Number of pages of *student*: 100 *takes*: 400

- Assuming worst case memory availability cost estimate is
  - with *student* as outer relation:
    - 100 + 5000 * 400 = 2,000,100 block transfers,
  - with *takes* as the outer relation
    - 400 + 10000 * 100 = 1,000,400 block transfers

Simple nested-loop Join (Cont.)

- Case 2 (best case): enough space for both relations
    - Cost for block transfer: $b_r + b_s$
- If smaller relation fits entirely in memory, use that as the inner relation.
    - Reduces cost to $b_r + b_s$ block transfers
- If smaller relation (student) fits entirely in memory, the cost estimate will be 500 block transfers.

Simple nested-loop Join – analysis

- b_r pages in *r*, p_r tuples per page
- b_s pages in *s*, p_s tuples per page
- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: $b_r + (p_r * b_r) * b_s$
- Example
  - Reserves: each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
  - Sailors: each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - Cost: 1000 + (100*1000)*500 I/Os.

Simple nested-loop join (page-oriented)

- Page-oriented Nested Loops join:

```
For each page of r,
    For each page of s,
        Write out matching pairs of tuples <$t_r$, $t_s$>,
            where $t_r$ is in r-page and $t_s$ is in s-page.
```

- Cost: $b_r + b_r * b_s = 1000 + 1000*500$
- If smaller relation (S) is outer, cost = 500 + 500*1000

Simple nested-loop join (page-oriented)

- Worst case: each page in the inner relation *s* is read once for each *page* in the outer relation
  - $b_r + b_r * b_s$ block transfers
- Best case:
  - $b_r + b_s$ block transfers

- Example 1: 400 pages of takes, 100 pages of students
  - Outer relation is student: 100+100*400=400,100 transfer
    - Improves 2,000,100 (simple nested-loop)
- Example 2: Reserves 1000 pages, Sailor 500 pages
  - Outer relation is Reserves: 1000 + 1000*500
  - Outer relation is Sailor: 500 + 500*1000

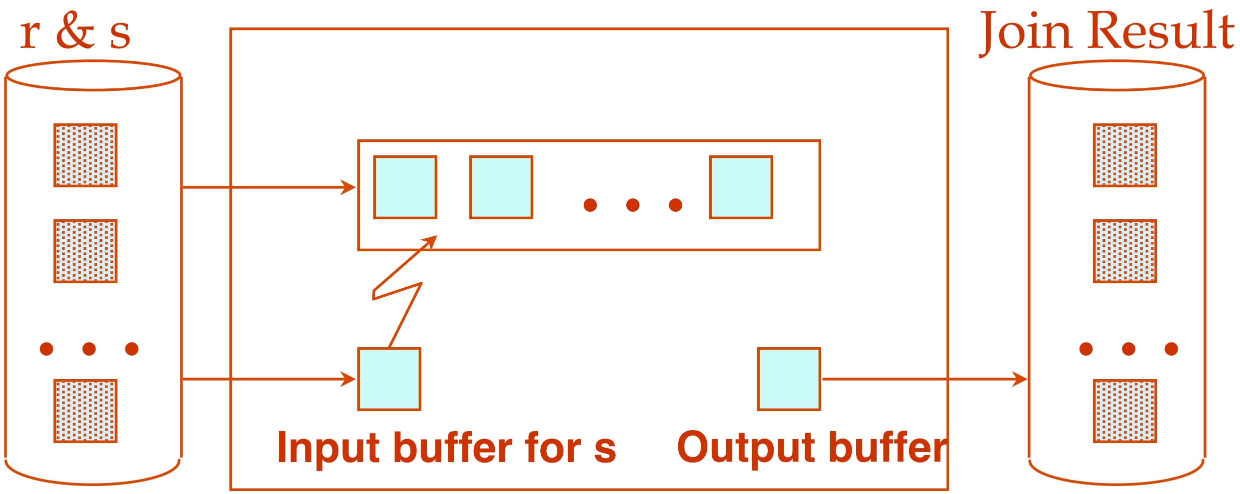
Simple nested-loop join (page-oriented)

- Improvements
  - If equi-join attribute forms a key on inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternatively, to make use of the blocks remaining in buffer (with LRU replacement)
  - Block nested-loop join
  - Indexed nested-loop

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner s, one page as the output buffer, and use all remaining pages to hold ``block’’ of outer r.

```
For each block of M-2 pages of r do
    For each page of s do
        For all matching in-memory tuples $t_r$ in r-block, $t_s$ in s-page, add <$t_r$, $t_s$> to result.
```



Analysis of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = ⌈# of pages of outer relation/block size⌉
  - M = memory size in blocks;
  - Cost
    - $b_r + \lceil b_r/(M-2) \rceil * b_s$ block transfers

Examples of Block Nested Loops

- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.

- Example 1: With Reserves as outer, and 100-page block of Reserves:
  - Block transfer cost: $1000 + \lceil 1000/100 \rceil * 500 = 6000$
  - 90-page block for Reserve, cost?
    - $1000 + \lceil 1000/90 \rceil * 500 = 1000 + 12 * 500 = 7000$
    - What is the minimum number of block pages to have this cost?
    - $\lceil 1000/(M-2) \rceil = 12$, $\lceil 1000/12 \rceil <= M <= \text{floor}(1000/11)$

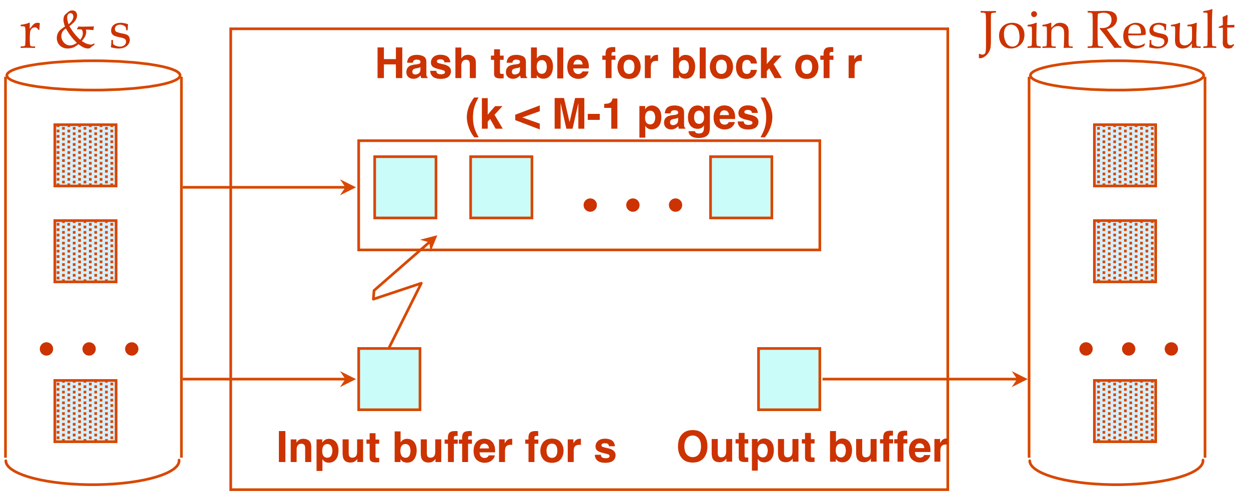
Examples of Block Nested Loops

- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.

- Example 2: 100-page block, Sailors as outer:
  - Block transfer cost: 500+(500/100)*1000 = 5500
  - 90-page block?
    - 500+ $\lceil 500/90 \rceil$ *1000 = 500+6*1000 = 6500
    - What is the minimum number of pages to have this cost?

Block Nested Loops Join -- improvement

- Hash table for outer relation r
  - The I/O cost does not change
  - The CPU cost is much lower



Index Nested Loops Join

```
For each tuple $t_r$ in $r$ do
    For each tuple $t_s$ in $s$ where $t_r == t_s$ do
        add $<t_r, t_s>$ to result
```

- Indexed relation as the inner relation
- Does not enumerate the cross-product of *r* and *s*

Indexed Nested-Loop Join

- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost (in I/Os): $b_r + ((b_r * p_r) *$ cost of finding matching *s* tuples)

- For each *r* tuple, cost of probing *s* index is
  - about 1.2 for hash index,
  - 2-4 for B+ tree
- Cost of finding *s* tuples depends on clustering.
  - Clustered index: 1 I/O (typical),
  - Un-clustered: up to 1 I/O per matching *s* tuple.

Examples of Index Nested Loops

- Example 1: Hash-index on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os
  - Reserves tuples: 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple.
    - Find sailor entry from index: 1.2 * (100 *1000) = 120,000 I/Os.
    - Find matching sailor tuple: 1*(100*1000) = 100,000
    - Total: 220,000
  - Total: 1000 + 220,000=221,000 I/Os

Examples of Index Nested Loops

- Example 2: Hash-index on *sid* of Reserves (as inner):
  - Scan Sailors: **500** page I/Os,
  - # of Sailors tuples: 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples.
    - Find Reserves entry from index: 1.2 * (80 * 500) = **48,000** I/Os.
  - Cost of retrieving matching Reserves tuples:
    - 100,000 reservations for 40,000 sailors
      - Assuming uniform distribution, 2.5 reservations per sailor (100,000/40,000).
      - Cost of retrieving reserves is 2.5 I/Os per sailor tuple.
        - Cost: 2.5 * (80*500) = **100,000** (un-clustered)
  - Total: 500+48,000+100,000 = 148,500 I/O

Example of Nested-Loop Join Costs

- Compute *student* ⋈ *takes,* with *student* as the outer relation.
- Let *takes* have a primary B+-tree index on the attribute *ID,* which contains 20 entries in each index node.
- students: 100 pages, 5000 tuples
- Takes: 400 pages, 10,000 tuples

- Cost of simple nested loops join (page-oriented)
  - 100 +100*400 = 40,100 block transfers
- Cost of indexed nested loops join
  - Since *takes* has 10,000 tuples, the approximate height of the tree is 4, and one/? more access is needed to find the actual data
  - 100 + 5000 * 5 = 25,100 block transfers and seeks.
- If indices are available on join attributes of both *r* and *s,*

  use the relation with fewer tuples as the outer relation.

Exercise

- Compute *student* $\bowtie$ *takes,*.
- Let the *student* relation have a primary B^+-tree index on the attribute *ID,* which contains 20 entries in each index node.

- students: 100 pages, 5000 tuples
- takes: 400 pages, 10,000 tuples

- with “takes” as the outer relation?

Sort-Merge Join $(r \bowtie_{i=j} s)$

- Sort r and s on the join column (external sort)
- Merging step: and output result tuples.
  - Advance scan of *r* until current r-tuple >= current s-tuple
    - Current r-tuple (Tr)
  - Then advance scan of *s* until current s-tuple >= current r-tuple; do this until current r-tuple = current s-tuple.
    - Current s-tuple (Gs)
  - At this point, all r-tuples with same value in r*i* (*current r partition*) and all *S* tuples with same value in *Sj* (*current s partition*) match;
    - For each Tr, loop using another pointer (Ts) all the s-tuples with the same value as the tuple pointed by Gs
    - Output $<t_r, t_s>$ for all pairs of such tuples.
    - After matching one Tr with all tuples in the s partition, advance Tr
  - Then resume scanning r and s.
- r is scanned once; each s group is scanned once per matching r tuple. (Multiple scans of an s group are likely to find needed pages in buffer.)

Example of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Example of Sort-Merge Join

- Cost (in I/Os): (sorting cost)+ (cost of merging)
  - The cost of merging, b_r+b_s, could be b_r*b_s (very unlikely!)
- With 101 buffer pages, both Reserves (1000 pages) and Sailors(500 pages) can be sorted in 2 passes;
  - M=101, with final result write:
    - Sort Reserves: 2*2*1000= 4000
    - Sort Sailors: 2*2*500 = 2000
    - Merge cost: 1000 + 500 = 1500
    - Total join cost: 7500.
- How about M=35? M=300?
- How about BNL cost?
  - 2500 to 15000 I/Os

Refinement of Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
  - With M > $\sqrt{L}$, where *L* is the size of the larger relation,
  - # of runs of each relation is < $\sqrt{L}$
  - Merging: buffer size $2\sqrt{L}$
    - Allocate 1 page per run of each relation, and “merge” while checking the join condition.
  - Cost: read and write each relation in Pass 0 + read each relation in (only) merging pass [+ writing of result tuples].
  - In example, cost goes down from 7500 to 4500 I/Os.

Sort-Merge Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:
  - $3*(b_r + b_s)$ (best)

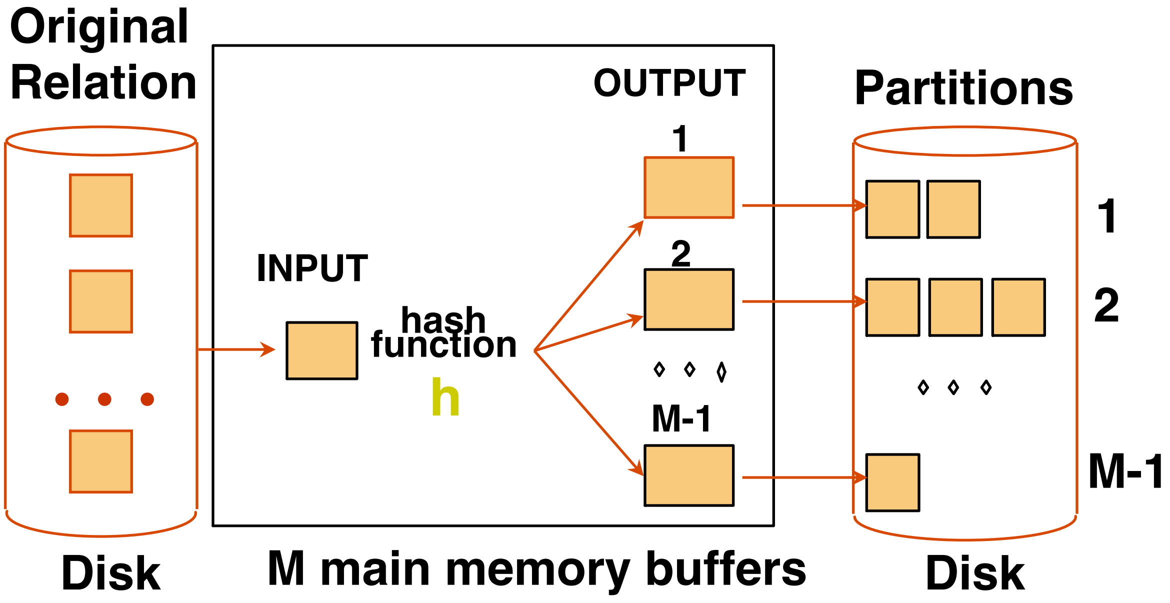
Example

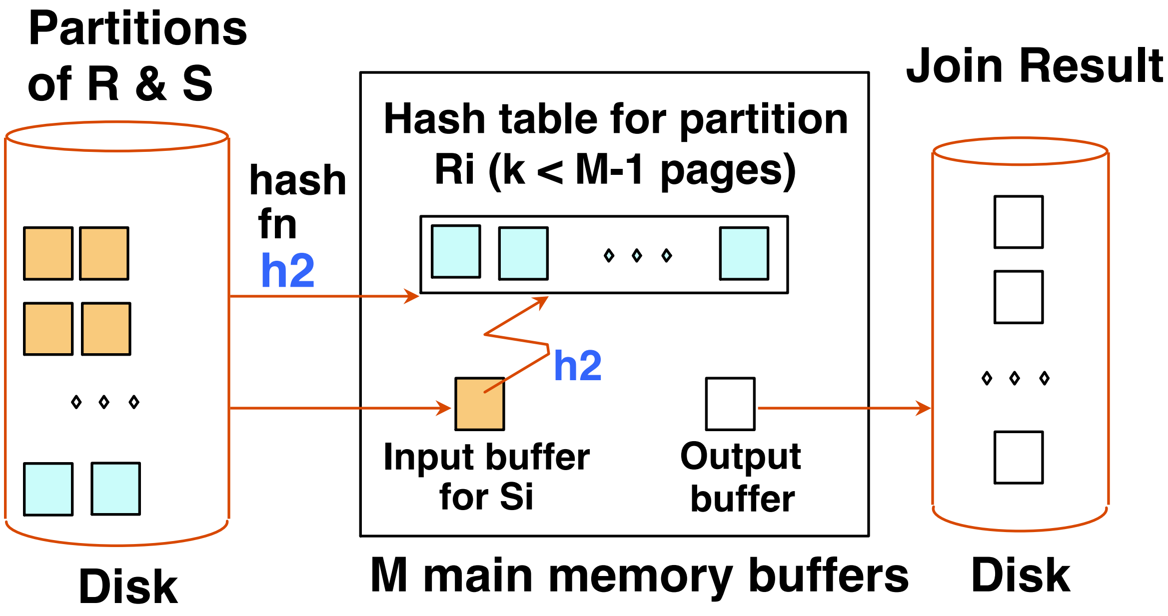
- Compute *student* $\bowtie$ *takes*
- student: 100 pages, 5000 tuples
- Takes: 400 pages, 10,000 tuples

- Already sorted on join attribute ID.
  - Merge cost = 400+100 = 500 block transfers
- Not sorted, M = 3
  - Sorting (write final output)
    - Takes: $\lceil \log_2 \lceil 400/3 \rceil \rceil$ = 8 merge passes; 2*400*(8+1) = 7200 block transfers; 2* $\lceil 400/3 \rceil$ + 8*(400/1)*2 =6668
    - Students: ???
  - Merging
    - 400+100 = 500 block transfers

Hash-Join

- Partition both relations using hash function **h**: r-tuples in partition i will only match s-tuples in partition i.
- Read in a partition of r, hash it using **h2 (<> h!)**. Scan matching partition of s, search for matches.
- Relation *r* is called the **build input** and *s* is called the **probe input.**





Observations on Hash-Join

- #partitions k <= M-1 (one input buffer), M-1 output buffers
- M-2 > size of largest partition to be held in memory.
- One partition fits in the memory, good.
- Assuming uniformly sized partitions, and maximizing k, we get:
  - k= M-1 (maximum)
  - If $b_r/(M-1) < M-2$, then $M > \sqrt{b_r}$
  - If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed. Typically *k* is chosen as $\lceil b_{build}/M \rceil * f$ where f is a "**fudge factor**", typically around 1.2
    - $M > \sqrt{f \bullet b_r}$
    - More specifically, $f*b_r/(M-1) < M-2$
  - The probe relation partitions need not fit in memory
- If the hash function does not partition uniformly, one or more r partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this r-partition with corresponding s-partition.

Cost of Hash-Join

- In partitioning phase, R+W both relations ; $2(b_r + b_s)$.
- In matching phase, read both relations; $b_r + b_s$ I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of $3(b_r + b_s)$ I/Os.
  - Hash Join is superior if relation sizes differ greatly.
  - Hash Join has shown to be highly parallelizable.
  - Sort-Merge is less sensitive to data skew; result is sorted.

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is $3(b_r + b_s)$ block transfers
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash-Join

- Compute *student* ⋈ *takes*
- student: 100 pages, 5000 tuples
- takes: 400 pages, 10,000 tuples

- Given M = 22 pages
- *student* is to be used as build input. Partition it into 5 partitions, each of size 20 pages (=M-2). This partitioning can be done in one pass.
- Similarly, partition *takes* into 5 partitions, each of size 80. This is also done in one pass.
- Total cost:
  - 3(100 + 400) = 1500 block transfers
- Always ignore cost of writing partially filled blocks
- Problem???

Complex Joins

- Join with a conjunctive condition:

  $$r \bowtie_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta i} s$
    - final result comprises those tuples in the intermediate results that satisfy the remaining conditions

  $$\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$

- Join with a disjunctive condition

  $$r \bowtie_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta i} s$:

    $$(r \bowtie_{\theta 1} s) \cup (r \bowtie_{\theta 2} s) \cup \ldots \cup (r \bowtie_{\theta n} s)$$

General Join Conditions

- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - For Index NL, build index on <R.*sid*, *R.sname*> (if R is inner); or use existing indexes on *sid* or *sname*.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner;
    - The # of matches is likely to be much higher than that for equality joins.
  - Hash Join, Sort-Merge Join is not applicable.
  - Block NL is quite likely to be the best join method here.

Summary

- No one join algorithm is uniformly superior to the others.
- The choice of a good algorithm
  - Sizes of the relations being joined
  - Available access methods
  - Size of the buffer pool