

# Comp115: Databases

## Query Processing with Relational Operations

Instructor: Manos Athanassoulis

# Query Processing

## Overview

Readings: Chapter 12

Selections

Projections

Nested loop joins

Sort-merge and hash joins

General joins and aggregates

# Query processing

- Some database operations are **EXPENSIVE**
- Can greatly improve performance by being 'smart'
  - e.g., can speed up 1,000,000x over naïve approach
- Main weapons are:
  1. clever implementation techniques for operators
  2. exploiting 'equivalencies' of relational operators
  3. using statistics and cost models to choose among these

# A Really Bad Query Optimizer

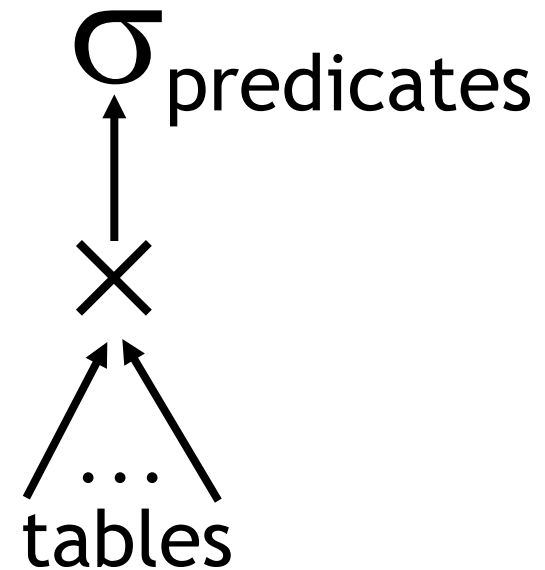
- For each Select-From-Where query block

- Create a plan that:

- Forms the cartesian product of the FROM clause
- Applies the WHERE clause
- Incredibly inefficient
  - Huge intermediate results!

- Then, as needed:

- Apply the GROUP BY clause
- Apply the HAVING clause
- Apply any projections and output expressions
- Apply duplicate elimination and/or ORDER BY



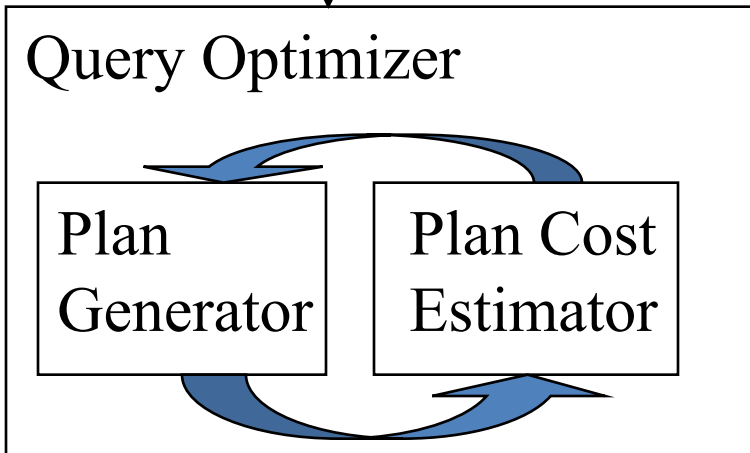
# Query execution

Query

```
Select *  
From Blah B  
Where B.blah = "foo"
```

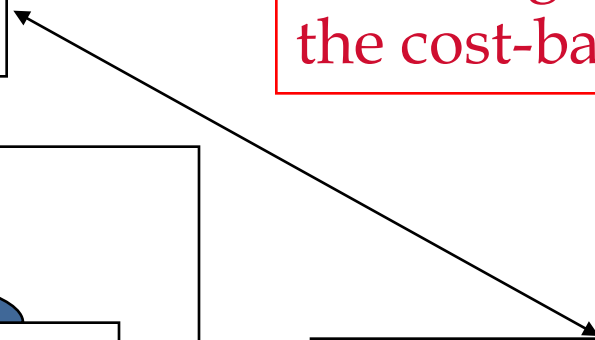


Query Parser

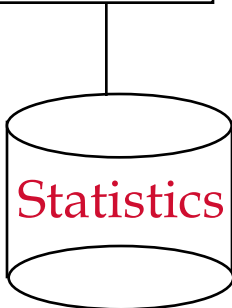
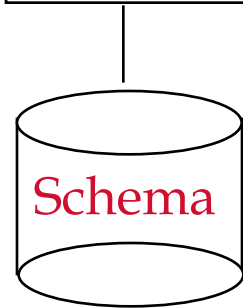
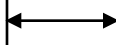


Query Plan Evaluator

Usually there is a heuristics-based rewriting step before the cost-based steps.



Catalog Manager



# The Query Optimization Game

- ‘Optimizer’ is a bit of a misnomer
- Goal: pick a ‘good’ (i.e., low expected cost) plan
  - Involves choosing access methods, physical operators, operator orders, ...
  - Notion of cost is based on an abstract ‘cost model’
- Roadmap for this topic:
  - First: basic operators
  - Then: joins
  - After that: optimizing multiple operators

# Relational Operations

- We will consider how to implement:
  - Selection ( $\sigma$ ) Selects a subset of rows from relation
  - Projection ( $\pi$ ) 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 ( $\cup$ ) Tuples in relation 1 and in relation 2
  - Aggregation (SUM, MIN, etc.) and GROUP BY
- Operators can be *composed* !
- Next: *optimizing* queries 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
  - $N=500$ ,  $p_S=80$
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages
  - $M=1000$ ,  $p_R=100$



# Query Processing

Overview

**Selections**

Readings: Chapters 14.1-14.2

Projections

Nested loop joins

Sort-merge and hash joins

General joins and aggregates

# Simple Selections

```
SELECT *
FROM Reserves R
WHERE R.rname < 'C%'
```

- Of the form  $\sigma_{R.attr \text{ op } value} (R)$
- Question: how best to perform? Depends on:
  - available indexes/access paths
  - expected size of the result (# of tuples and/or # of pages)
- **Size of result** approximated as
  - size of R \* reduction factor*
  - “reduction factor” is usually called *selectivity*
  - estimate of selectivity is based on statistics

# Alternatives for Simple Selections

- **With no index, unsorted:**
  - Must essentially scan the whole relation
  - **cost is  $M$**  (#pages in  $R$ ); for “reserves” = 1000 I/Os
- **With no index, sorted:**
  - cost of binary search + number of pages containing results.
  - For reserves = 10 I/Os +  $\lceil \text{selectivity} * \# \text{pages} \rceil$
- **With an index on selection attribute:**
  1. Use index to find qualifying data entries,
  2. then retrieve corresponding data records
    - Note: Hash index useful only for equality selections

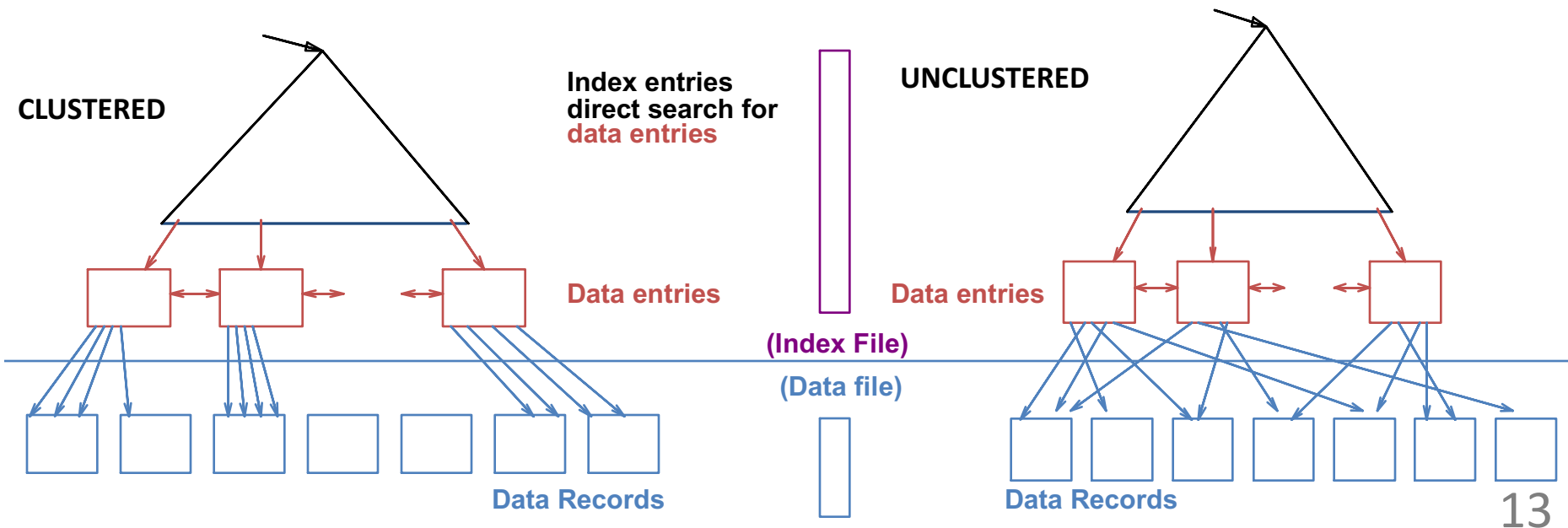
# Using an Index for Selections

- Cost  $\sim$  #qualifying tuples, clustering
  - Cost factors:
    - find qualifying data entries (typically small)
    - retrieve records (could be large w/o clustering)
  - Our example, “reserves” relation:  
if 10% of tuples qualify (100 pages, 10000 tuples)
    - *clustered* index  $\rightarrow$  a bit more than **100** I/Os
    - *unclustered*  $\rightarrow$  could be up to **10000** I/Os!  
unless...

# Selections using Index

- Important refinement for unclustered indexes:*

1. Find qualifying data entries
2. Sort the rid's of the data records to be retrieved
3. Fetch rids in order
  - Ensuring that each data page is looked at just once



# General Selection Conditions

☛ *(day < 8/9/94 AND rname = 'Paul') OR bid = 5 OR sid = 3*

- First converted to conjunctive normal form (CNF)
  - *(day < 8/9/94 OR bid = 5 OR sid = 3) AND (rname = 'Paul' OR bid = 5 OR sid = 3)*
- We assume no ORs (conjunction of *<attr op value>*)
- A **B-tree** index matches (a conjunction of) terms that involve only attributes in a *prefix* of the search key
  - Index on *<a, b, c>* matches *a = 5 AND b = 3*, but not *b = 3*
- **Hash** indexes must have all attributes in search key

# Selections – 1<sup>st</sup> approach

1. Find the *cheapest access path*
2. Retrieve tuples using it
3. Apply the terms that don't **match** the index (if any):
  - *Cheapest access path*  
An index or file scan with the fewest estimated page I/Os
  - **Terms that match** this index reduce the # of tuples *retrieved*
  - **Other terms** are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched

# Cheapest Access Path - Example

- Consider *day < 8/9/94 AND bid=5 AND sid=3*
- A **B+ tree index** on day can be used;
  - then, *bid=5* and *sid=3* must be checked for each retrieved tuple
- Similarly, a hash index on  $\langle bid, sid \rangle$  could be used;
  - Then, *day < 8/9/94* must be checked
- *How about a B+tree on  $\langle rname, day \rangle$ ?*
- *How about a B+tree on  $\langle day, rname \rangle$ ?*
- *How about a Hash index on  $\langle day, rname \rangle$ ?*



## Selections – 2<sup>nd</sup> approach: Intersecting RIDs

- If we have 2 or more matching indexes (w/Alternatives (2) or (3) for data entries):
  1. Get **sets of rids** of data records using **each** matching index
  2. Then *intersect* these **sets of rids**
  3. Retrieve the records and apply any remaining terms

**EXAMPLE:** Consider *day<8/9/94 AND bid=5 AND sid=3*

- With (i) a **B+ tree index on day** and (ii) an **index on sid**:
  1. a) Retrieve rids of records satisfying *day<8/9/94* using the first  
b) Retrieve rids of recs satisfying *sid=3* using the second
  2. **Intersect**
  3. Retrieve records and check *bid=5*

# Selections: summary

- Simple selections
  - On sorted or unsorted data, with or without index
- General selections
  - Expressed in conjunctive normal form
  - Retrieve tuples and then filter them through other conditions
  - Intersect RIDs of matching tuples for non-clustered indexes
- Choices depend on selectivities



# Break: The Halloween Problem

- Story from the early days of System R.
- While testing the optimizer on 10/31/76(?), the following update was run:

```
UPDATE payroll
SET salary = salary*1.1
WHERE salary < 25K;
```



- AND IT STOPPED WHEN ALL HAD salary  $\geq$  25K!
- Can you guess why? (hint: it was an optimizer bug...)

# Query Processing

Overview

Selections

**Projections**

Readings: Chapter 14.3

Nested loop joins

Sort-merge and hash joins

General joins and aggregates

# The Projection Operation

```
SELECT  DISTINCT
        R.sid, R.bid
FROM    Reserves R
```

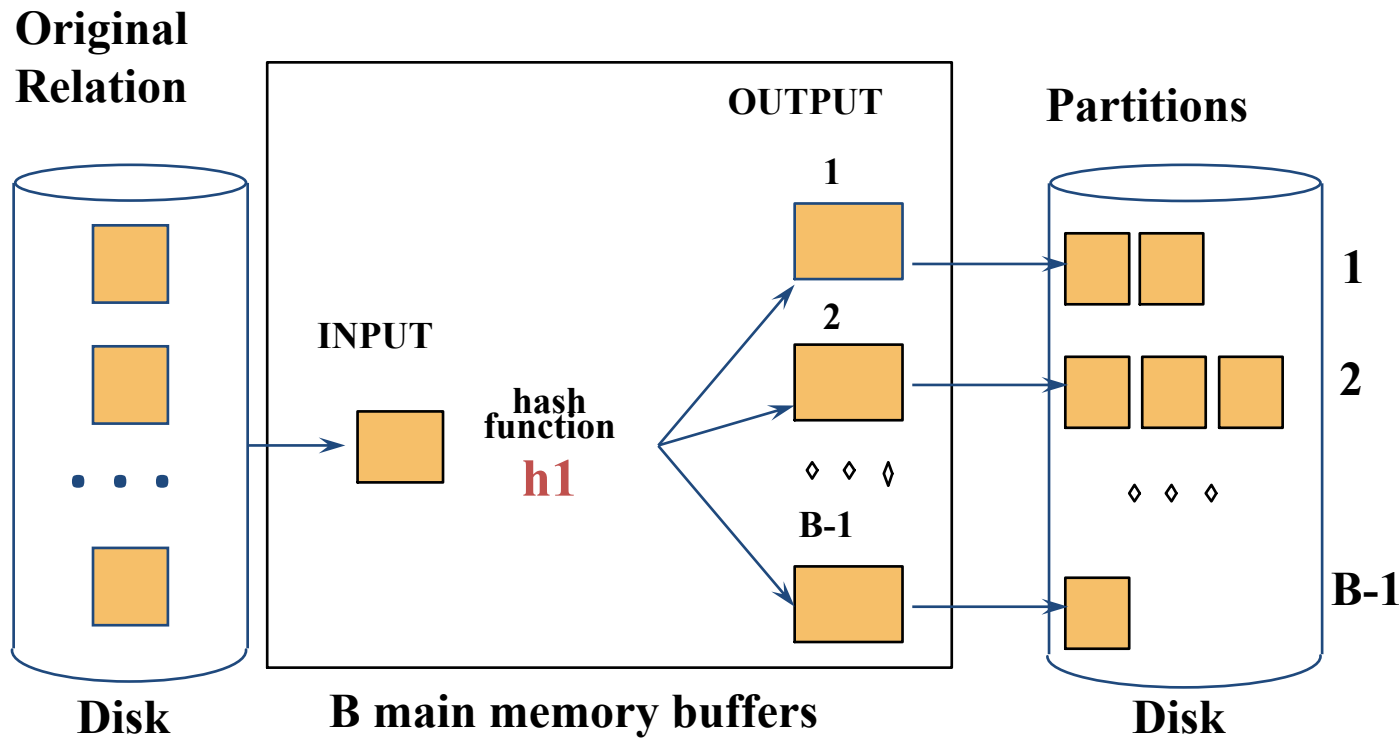
- Issue is removing **duplicates**
- Basic approach is to use sorting
  - 1. Scan R, extract only the needed attributes (why do this first?)
  - 2. Sort the resulting set
  - 3. Remove adjacent duplicates
  - Cost: Reserves with size ratio 0.25 = 250 pages  
With 20 buffer pages can sort in 2 passes, so:  
 $1000 + 250 + 2 * 2 * 250 + 250 = 2500$  I/Os

# Projection: Can do better!

```
SELECT  DISTINCT
        R.sid, R.bid
FROM    Reserves R
```

- Modify external sort algorithm (see chapter 13):
  - Modify Pass 0 of external sort to eliminate unwanted fields
  - Modify merging passes to eliminate duplicates
  - Cost for above case:  
read 1000 pages, write out 250 in runs of 40 pages,  
merge runs =  $1000 + 250 + 250 = 1500$

# Projection Based on *Hashing*



# Projection Based on *Hashing* (explained)

- *Partitioning phase:*
  - Read R using one input buffer
  - For each tuple:
    - Discard unwanted fields
    - Apply hash function  $h1$  to choose one of B-1 output buffers
  - Result is B-1 partitions (of tuples with no unwanted fields)
    - 2 tuples from different partitions guaranteed to be distinct



# Projection Based on *Hashing (explained)*

- *Duplicate elimination phase:*
  - For each partition
    - Read it and build an in-memory hash table
      - using hash function  $h2$  ( $\neq h1$ ) on all fields
    - while discarding duplicates
  - If partition does not fit in memory
    - Apply hash-based projection algorithm recursively to this partition

# Projection Based on *Hashing* (explained)

- Cost ???

- Assuming partitions fit in memory (i.e. #bufs  $\geq$  sqrt(#of pages) )
- Read 1000 pages
- Write partitions of projected tuples (250 I/Os)
- Do dup elim on each partition (total 250 I/Os)
- Total : 1500 I/Os

# Discussion of Projection (1/2)

- Sort-based approach is standard
  - Better handling of **skew**, and result is **sorted**
- If there are enough buffers, both have same I/O cost:

$$M + 2T$$

where:

- M is #pgs in R,
  - T is #pgs of R with unneeded attributes removed
- Although many systems don't use the specialized sort

# Discussion of Projection (2/2)

- If all wanted attributes are indexed
  - *index-only* scan
  - Apply projection techniques to data entries (much smaller!)
- If all wanted attributes are indexed as prefix of the search key → even better:
  - Retrieve data entries in order (index-only scan)
  - Discard unwanted fields
  - Compare adjacent tuples to check for duplicates

# Projections: summary

- Projection based on sorting
- Projection based on hashing
- Can use indexes if they cover relevant attributes

# Query Processing

Overview

Selections

Projections

**Nested loop joins**

Readings: Chapters 14.4-14.4.1

Sort-merge and hash joins

General joins and aggregates

# Joins...

- ...are very common.
- ...can be very expensive (cross product in the worst case).

➔ Many approaches to reduce join cost!

## Join techniques we will cover:

1. Nested-loops join
2. Index-nested loops join
3. Sort-merge join
4. Hash join

# Equality Joins With One Join Column

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

- In algebra:  $R \bowtie S$ . Common! Must be carefully optimized.  $R \times S$  is large; so,  $R \times S$  followed by a selection is inefficient
- Remember, join is associative and commutative
- Assume:
  - $M$  pages in  $R$ ,  $p_R$  tuples per page
  - $N$  pages in  $S$ ,  $p_S$  tuples per page
  - In our examples,  $R$  is Reserves and  $S$  is Sailors
- We will consider more complex join conditions later
- *Cost metric* : # of I/Os
- We will ignore output costs



# Simple Nested Loops Join

```
foreach tuple r in R do
  foreach tuple s in S do
    if  $r_i == s_j$  then add  $\langle r, s \rangle$  to result
```

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S
- How much does this Cost?
- $(p_R * M) * N + M = 100 * 1000 * 500 + 1000$  I/Os
  - At 10ms/IO, Total: ???
- What if smaller relation (S) was outer?
- What assumptions are being made here?

**Q: What is cost if one relation can fit entirely in memory?**

# Page-Oriented Nested Loops Join

```

foreach page  $b_R$  in R do
  foreach page  $b_S$  in S do
    foreach tuple  $r$  in  $b_R$  do
      foreach tuple  $s$  in  $b_S$  do
        if  $r_i == s_j$  then add  $\langle r, s \rangle$  to result
  
```

- For each **page** of R
  - get each **page** of S
  - write out matching pairs of tuples  $\langle r, s \rangle$ , where  $r$  is in R-page and  $S$  is in S-page
- What is the cost of this approach?
- $M * N + M = 1000 * 500 + 1000$ 
  - If smaller relation (S) is outer, cost =  $500 * 1000 + 500$

# Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where  $r_i == s_j$  do
        add <r, s> to result
```

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index
  - Cost:  $M + (M * p_R) * \text{cost of finding matching S tuples}$
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering
- Clustered index: 1 I/O per page of matching S tuples
- Unclustered: up to 1 I/O per matching S tuple

# Examples of Index Nested Loops (1/2)

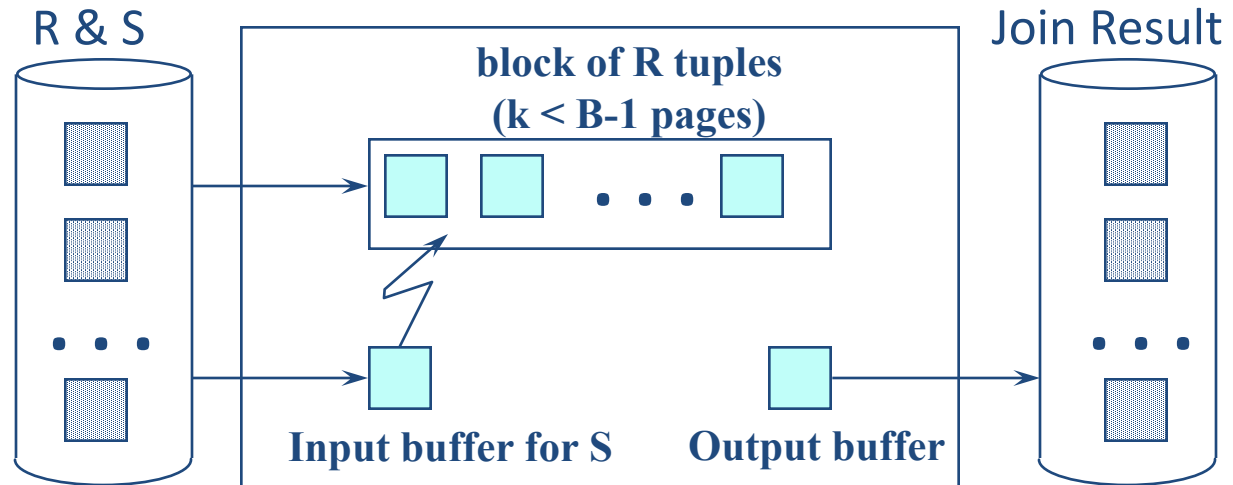
- Hash-index (Alt. 2) on *sid* of Sailors (inner):
  - Scan Reserves: 1000 page I/Os,  $100 * 1000$  tuples
  - For each Reserves tuple:
    - 1.2 I/Os to get data entry in index,
    - plus 1 I/O to get (the exactly one) matching Sailors tuple

## Examples of Index Nested Loops (2/2)

- Hash-index (Alt. 2) on *sid* of Reserves (inner):
  - Scan Sailors: 500 page I/Os,  $80 \times 500$  tuples
  - For each Sailors tuple:
    - 1.2 I/Os to find index page with data entries,
    - plus cost of retrieving matching Reserves tuples
    - **Assuming uniform distribution**, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered

# Block Nested Loops Join

- Page-oriented NL doesn't exploit extra buffers
- **Alternative approach:** 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 matching tuple  $r$  in R-block,  $s$  in S-page, add  $\langle r, s \rangle$  to result. Then read next R-block, scan S, etc



# Examples of Block Nested Loops

- **Cost: Scan of outer + #outer blocks \* scan of inner**
  - #outer blocks =  $\lceil \# \text{ of pages of outer} / \text{blocksize} \rceil$
- **With Reserves (R) as outer, and 100 pages of R:**
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*
  - Per block of R, we scan Sailors (S); 10\*500 I/Os
- **With 100-page block of Sailors as outer:**
  - Cost of scanning S is 500 I/Os; a total of 5 blocks
  - Per block of S, we scan Reserves; 5\*1000 I/Os
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S

# Nested loop joins: summary

- Simple nested loops
  - Optimized by page-oriented access
- Index nested loops
  - Costs depend on the type of index
- Block nested loops
  - Optimization of page nested loops which uses memory buffers



# Query Processing

Overview

Selections

Projections

Nested loop joins

**Sort-merge and hash joins**

Readings: Chapters 14.4.2-14.4.3

General joins and aggregates

# Sort-Merge Join $(R \bowtie_{i=j} S)$

- Sort R and S on the join column, then scan them to do a ‘merge’ (on join column), and output result tuples
- Useful if
  - one or both inputs are already sorted on join attribute(s)
  - output is required to be sorted on join attributes(s)
- ‘Merge’ phase can require some back tracking if duplicate values appear in join column
- R is scanned once; each S group is scanned once per matching R tuple. Note: Multiple scans of an S group will probably find needed pages in buffer

# Example of Sort-Merge Join

<u>sid</u>	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

<u>sid</u>	<u>bid</u>	<u>day</u>	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

- **Cost: Sort R + Sort S + (M+N)**
  - The cost of scanning, M+N, could be M\*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500

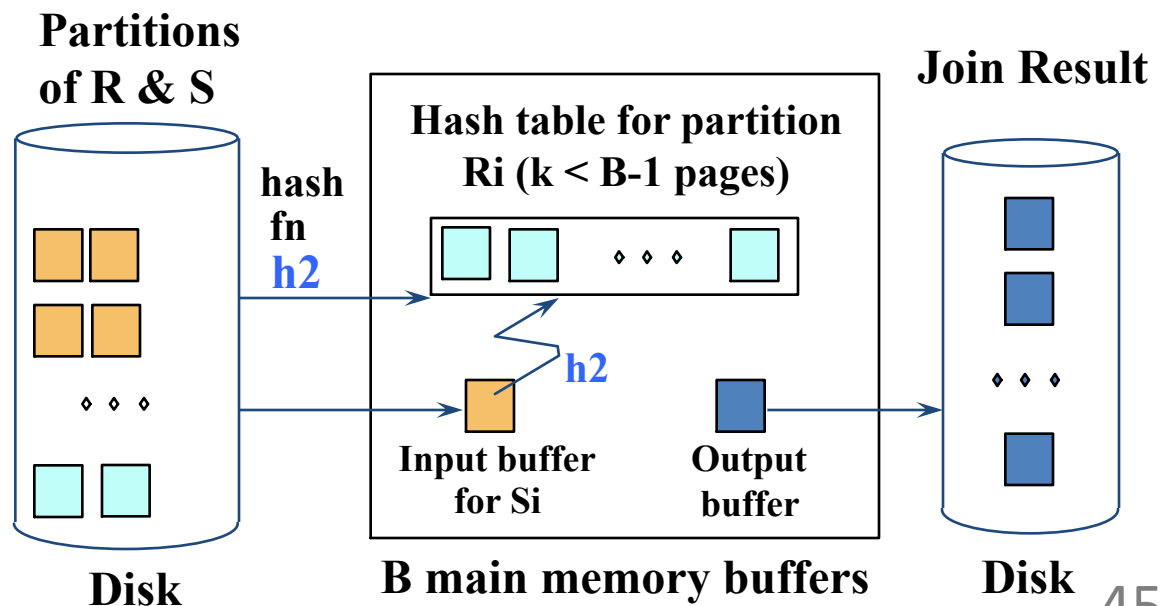
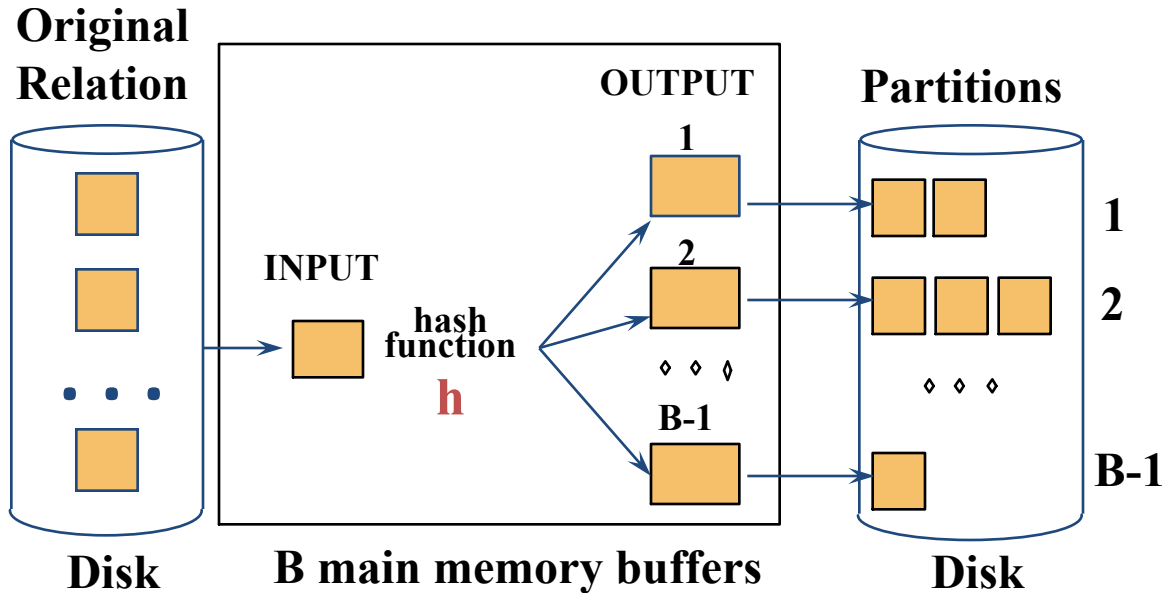
(BNL cost: 2500 to 15000 I/Os) 43

# 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
  - Allocate 1 page per run of each relation, and ‘merge’ while checking the join condition
  - With  $B > \sqrt{L}$ , where  $L$  is the size of the larger relation, using the sorting refinement that produces runs of length  $2B$  in Pass 0, #runs of each relation is  $< B/2$
  - **Cost:** read+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

# 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  $h_2$  ( $\ll h!$ ). Scan matching partition of  $S$ , probe hash table for matches



# Observations on Hash-Join

- First pass creates  $B-1$  partitions, each of size  $S_i = N/(B-1)$
- Need each  $S_i \leq B-2$  in order to fit in memory for 2<sup>nd</sup> pass
  - Need  $N/(B-1) \leq B-2$ 
    - ... or, roughly:  $B > \sqrt{N}$ 
      - where  $N$  is size of smaller relation

# More Observations on Hash-Join

- Since we build an in-memory hash table to speed up the matching of tuples in the second phase, a little more memory is needed
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. We 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, read+write both relns;  $2(M+N)$
- In matching phase, read both relns;  $M+N$  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  $3(M+N)$  I/Os
- Hash Join Pros:
  - Superior if relation sizes differ greatly
  - Shown to be highly parallelizable (*beyond scope of class*)
- Sort-Merge Join Pros:
  - Less sensitive to data skew
  - Result is sorted (may help “upstream” operators)
  - Goes faster if one or both inputs already sorted

# Hash-Join

Let  $B = 5$

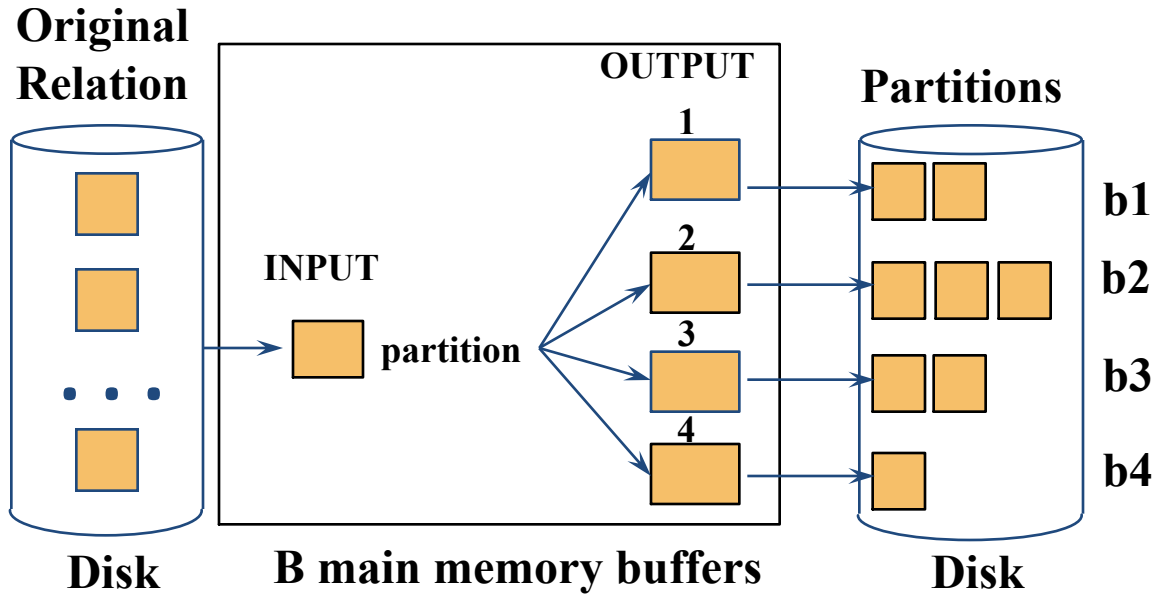
Buckets:

$b1: h \in [1,25]$

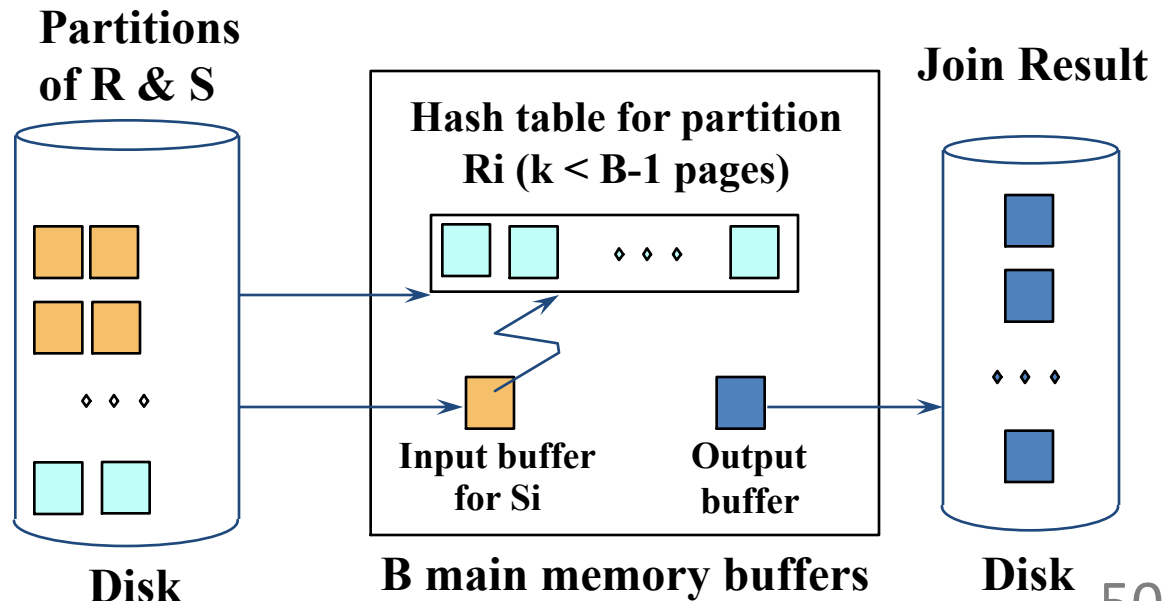
$b2: h \in [26,50]$

$b3: h \in [51,75]$

$b4: h \in [76,100]$



If  $|F| \leq |M|$ , in second phase build in-memory hash table on  $F$  partitions, and stream  $M$  partitions through memory



# Summary

- Sort merge join
  - Relies on the sorted order of join attributes
  - Produces sorted output
- Hash join
  - Uses little memory
  - Great when one relations is much smaller than the other
  - Has problems with data skew

# Query Processing

Overview

Selections

Projections

Nested loop joins

Sort-merge and hash joins

**General joins and aggregates**

Readings: Chapters 14.4.5-14.7

# General Join Conditions

- Equalities over several attributes (e.g., *R.sid=S.sid AND R.rname=S.sname*):
  - For Index NL, build index on *<sid, sname>* (if S 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; # matches likely to be much higher than for equality joins
  - Hash Join, Sort Merge Join not applicable!
  - Block NL quite likely to be the best join method here

# Set Operations

- Intersection and cross-product special cases of join
- Union (Distinct) and Except similar; we'll do **union**:
- Sorting based approach to union:
  - Sort both relations (on combination of all attributes)
  - Scan sorted relations and merge them
  - *Alternative*: Merge runs from Pass 0 for *both* relations
- Hash based approach to union:
  - Partition R and S using hash function  $h$
  - For each S-partition, build in-memory hash table (using  $h_2$ ), scan corresponding R-partition and add tuples to table while discarding duplicates

# Aggregate Operations (AVG, MIN, etc.)

- Without grouping:
  - In general, requires scanning the relation
  - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan

# Aggregate Operations (AVG, MIN, etc.)

- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. Note: we can improve upon this by combining sorting and aggregate computation
  - Similar approach based on hashing on group-by attributes
  - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, we can do index-only scan
  - If group-by attributes form prefix of the search key, we can retrieve data entries/tuples in group-by order



# Impact of Buffering

- If several operations are executing concurrently, estimating the number of available buffer pages is guesswork
- Repeated access patterns interact with buffer replacement policy
  - e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (*sequential flooding*)
  - Does replacement policy matter for Block Nested Loops?
  - What about Index Nested Loops?

# Summary

- A virtue of relational DBMSs:
  - queries are composed of a few basic operators
  - Implementation of operators can be **carefully tuned**
  - Important to do this!
- Many alternative implementations for each operator
  - No universally superior technique for most operators
- Must consider alternatives for each operation in a query and choose best one based on system statistics...
  - Part of the broader task of optimizing a query composed of several operations