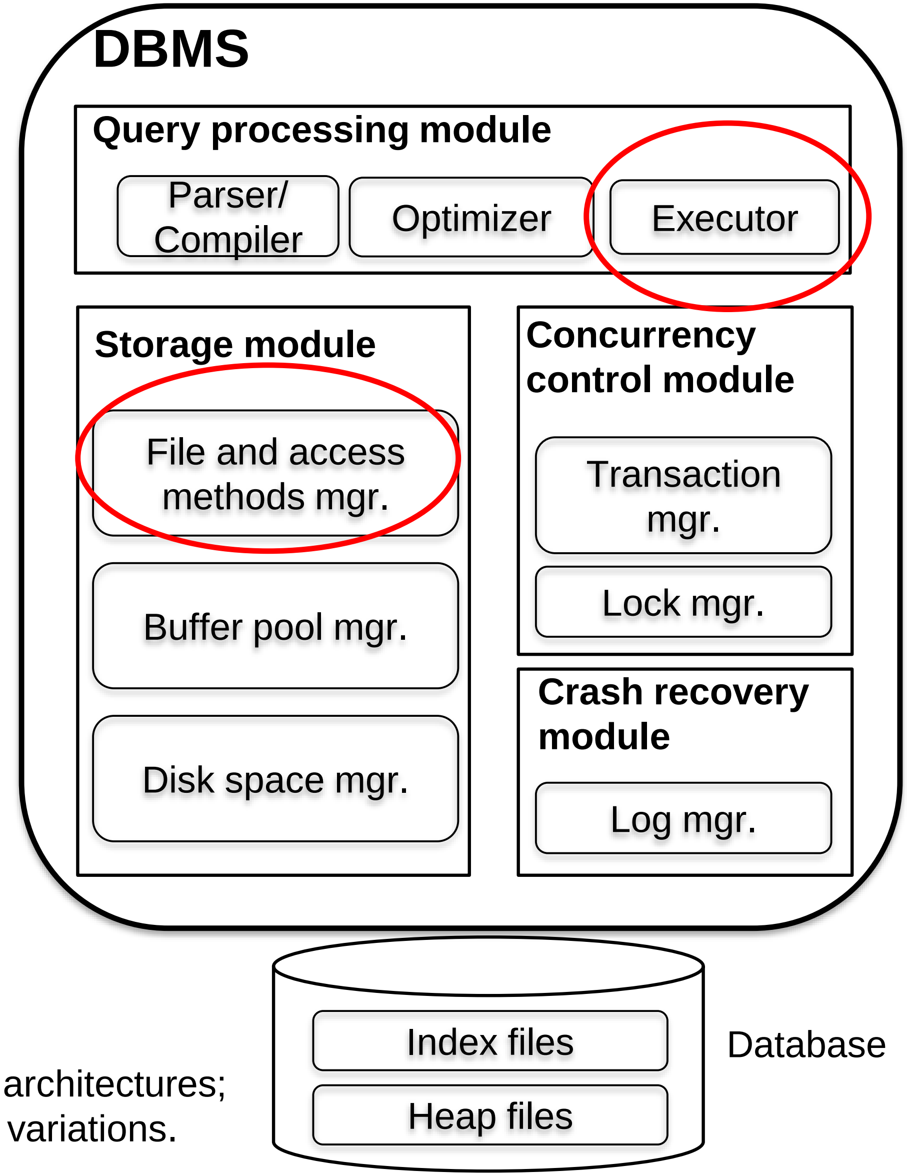
INFO20003 Database Systems

Dr Renata Borovica-Gajic

Lecture 11
Query Processing Part I

Remember this? Components of a DBMS



TODAY & Next time

Will briefly touch upon …

This is one of several possible architectures; each system has its own slight variations.

Coverage

- Query Processing Overview
- Selections
- Projections

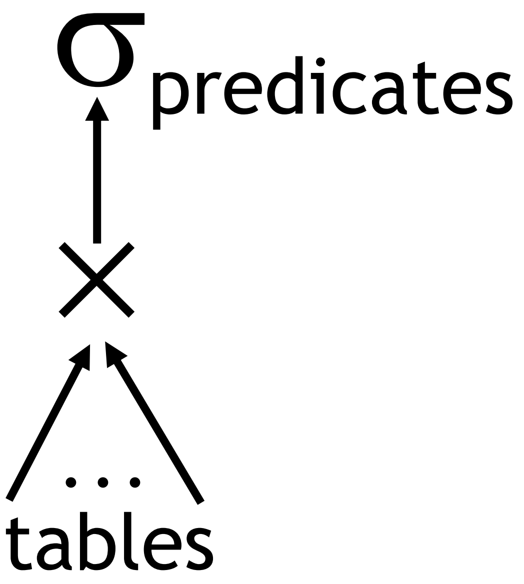
Readings: Chapter 12 and 14, Ramakrishnan & Gehrke, Database Systems

Query processing

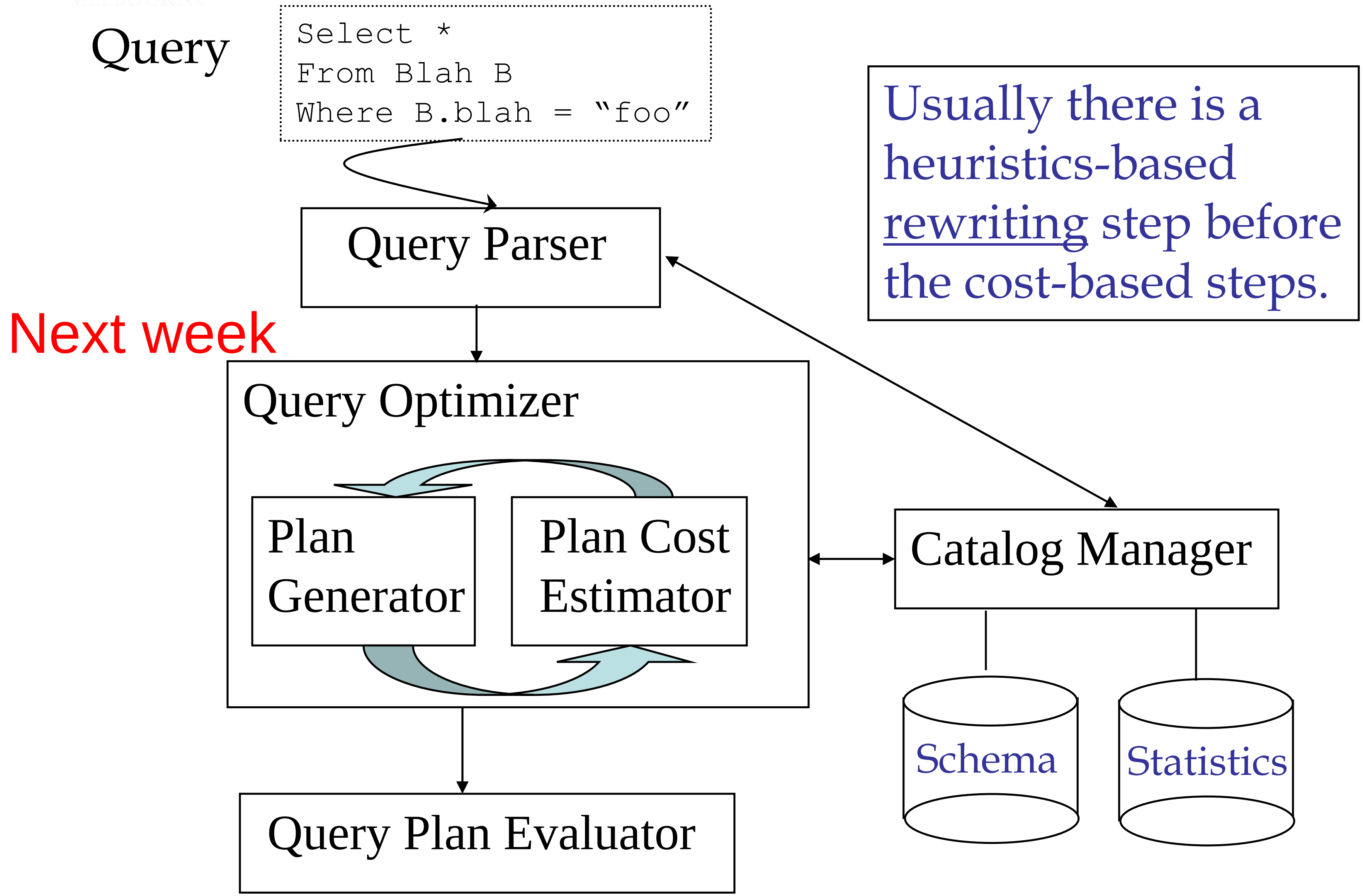
- Some database operations are EXPENSIVE
- Can greatly improve performance by being ‘smart’
  - e.g., can speed up 1,000,000x over naïve approach
- Main weapons are:
  1. clever implementation techniques for operators
  2. exploiting ‘equivalencies’ of relational operators
  3. using statistics and cost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

- ‘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* (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* ($\cup$) 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

- Query Processing Overview
- Selections
- Projections

Readings: Chapter 14, Ramakrishnan & Gehrke, Database Systems

Simple Selections

- Of the form $\sigma_{R.attr\ op\ value}(R)$

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

- 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

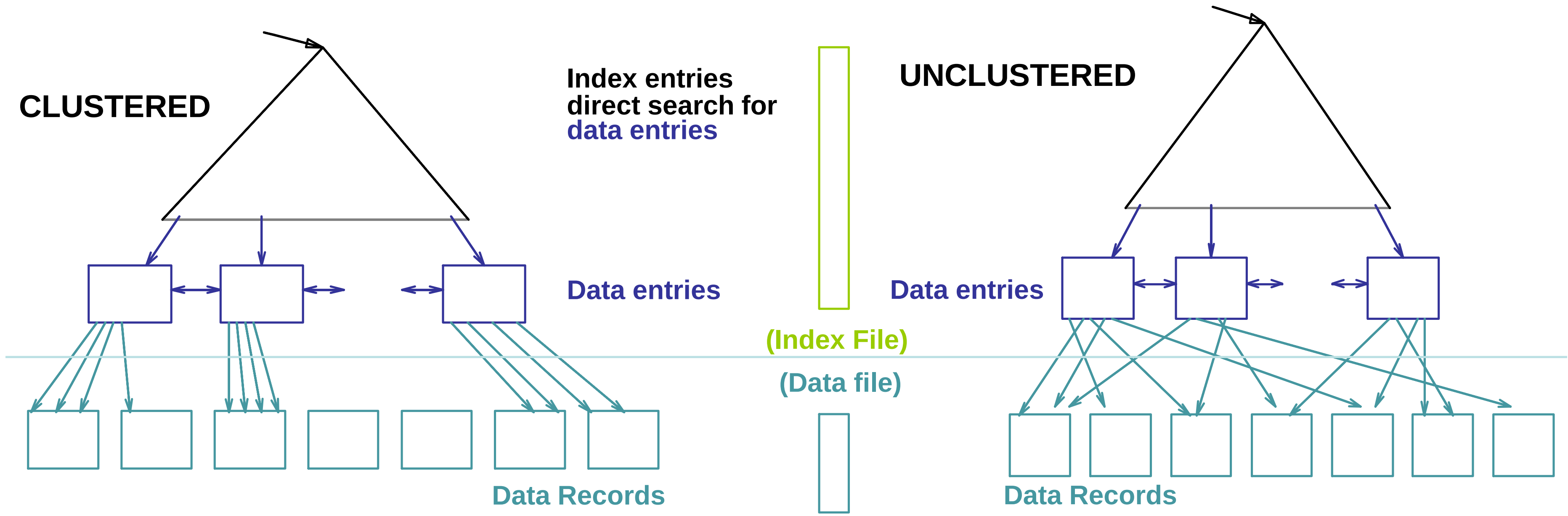
- With no index, unsorted:
  - Must essentially scan the whole relation
  - cost is M (#pages in R); for "reserves" = 1000 I/Os
- With no index, sorted:
  - cost of binary search + number of pages containing results.
  - For reserves = 10 I/Os + ⌈selectivity*#pages⌉
- With an index on selection attribute: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

Using an Index for Selections

- Cost ~ #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 → a bit more than 100 I/Os
    - *unclustered* → could be up to 10000 I/Os!

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

Example:

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

- First converted to <u>*conjunctive normal form* (CNF)</u>
  - (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 <u>*matches*</u> (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 – 1st 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 <*bid, sid*> could be used;
  - *Then, day<8/9/94* must be checked
- *How about a B+tree on <rname,day>?*
- *How about a B+tree on <day, rname>?*
- *How about a Hash index on <day, rname>?*

- 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 them filter them through other conditions
  - Intersect RIDs of matching tuples for non-clustered indexes
- Choices depend on selectivities

The Halloween Problem

- Story from the early days of System R.
- While testing the optimizer on 10/31/75(?), the following update was run:

```
UPDATE payroll
SET salary = salary*1.1
WHERE salary > 20K;
```

- AND IT NEVER STOPPED!

- Can you guess why?

Query Processing

- Overview
- Selections
- Projections

Readings: Chapter 14, Ramakrishnan & Gehrke, Database Systems

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

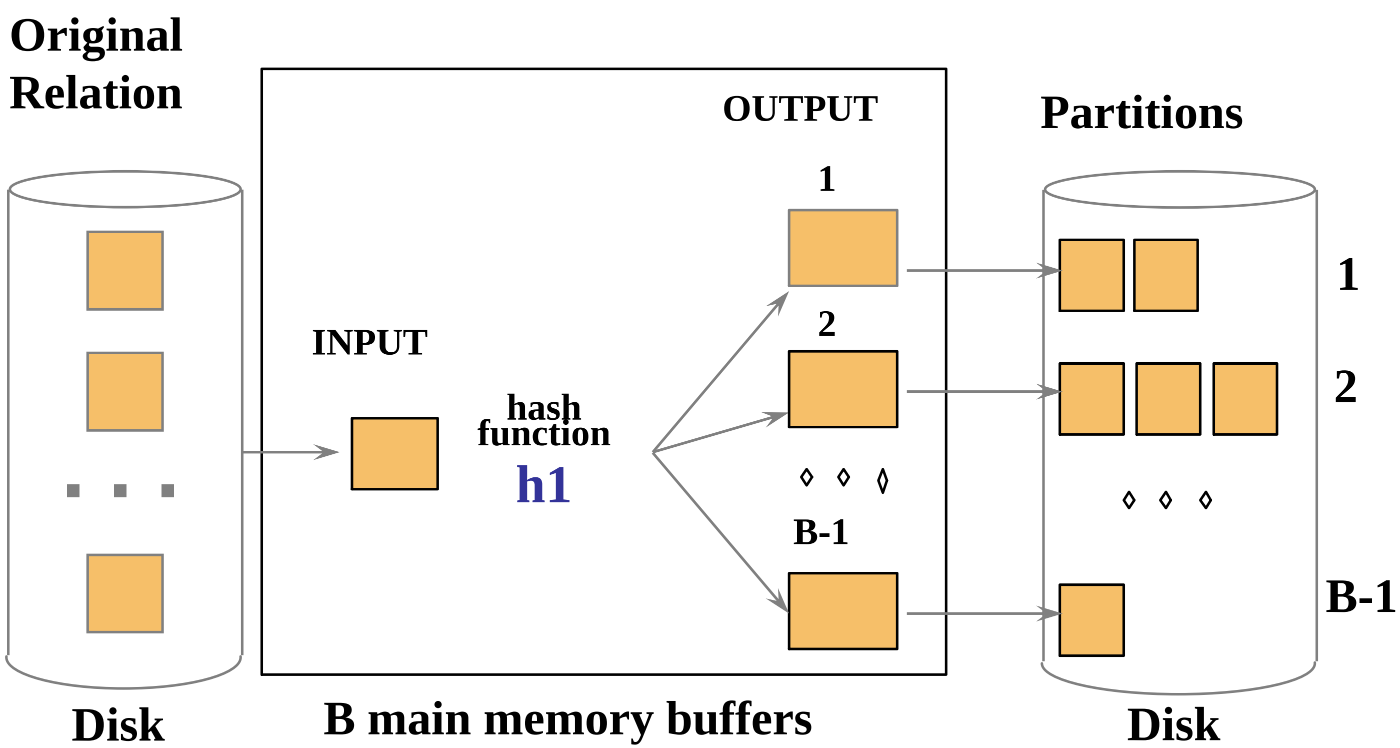
```
SELECT  DISTINCT  R.sid, R.bid
  FROM    Reserves R
```

- Modify external sort algorithm (see chapter 13):

  1. Modify Pass 0 of external sort to eliminate unwanted fields
  2. 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*

1. *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*

2. *Duplicate elimination phase:*
   - For each partition
     - Read it and build an in-memory hash table
       - using hash function *h2* (<> *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*

- Cost ?
- Assuming partitions fit in memory (i.e. #bufs >= 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

THE UNIVERSITY OF MELBOURNE

Summary

- Query processing understanding is crucial for active DBMS usage
- Performance difference between good and bad query processing strategies can differ by orders of magnitude
- Each relational query operator has several implementation alternatives: learn them

What's examinable

- Understand the logic behind relational operators
- Learn alternatives for selections and projections (for now)
  - Be able to calculate the cost of alternatives
- Important for Assignment 3 as well

Next Lecture

- Query Processing Part II
  - Join alternatives