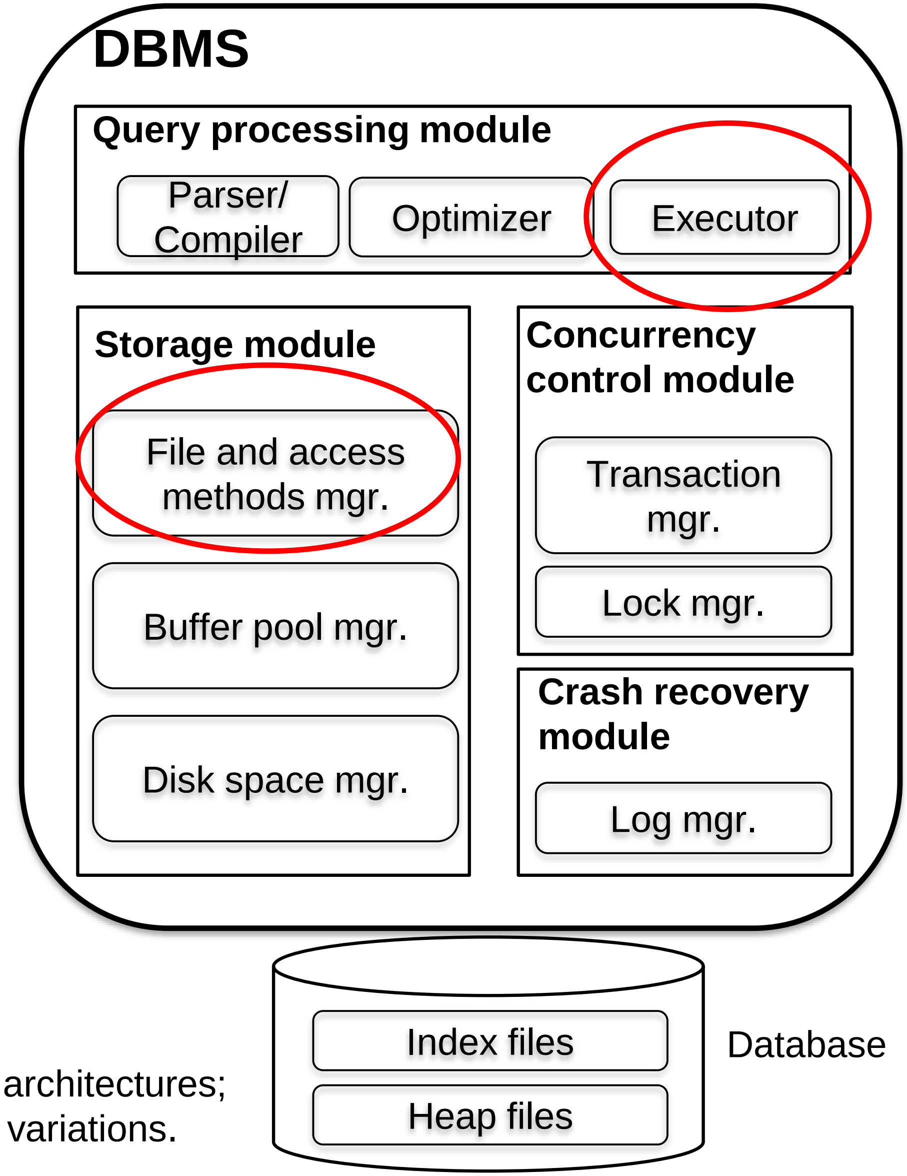
THE UNIVERSITY OF MELBOURNE

INFO20003 Database Systems

Dr Renata Borovica-Gajic

Lecture 12
Query Processing Part II

Remember this? Components of a DBMS



TODAY Joins

Will briefly touch upon …

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

Coverage

- Nested loop joins
- Sort-merge and hash joins
- General joins and aggregates

Readings: Chapter 14, Ramakrishnan & Gehrke, Database Systems

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 X S is large; so, R X 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 <r, s> 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 <r, s> to result
```

- For each *page* of R
  - get each *page* of S
  - write out matching pairs of tuples <r, s>, 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

THE UNIVERSITY OF MELBOURNE

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: $M + ((M*p_R) *$ 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

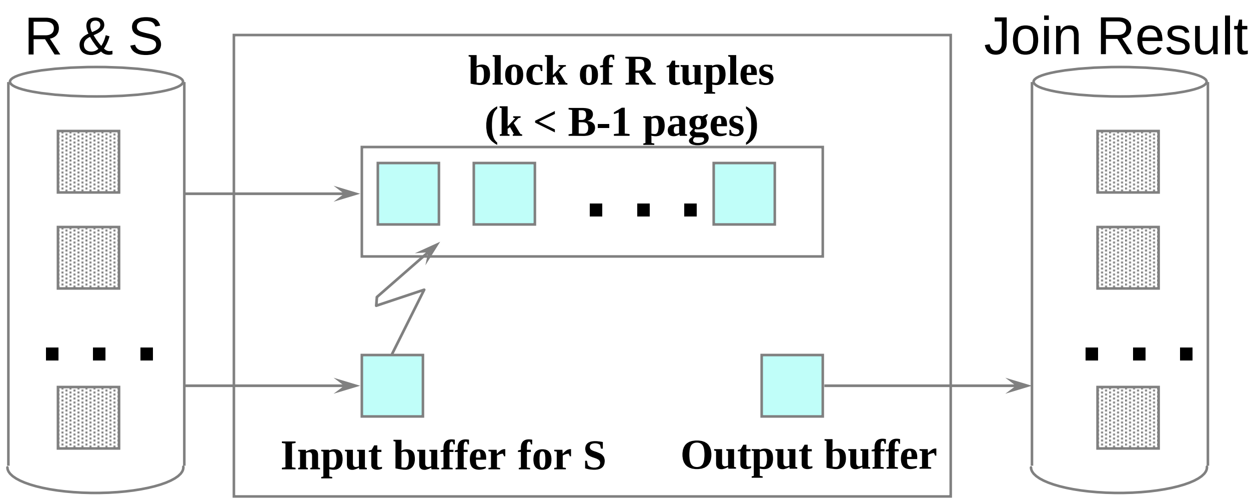
THE UNIVERSITY OF MELBOURNE

Examples of Index Nested Loops (2/2)

- Hash-index (Alt. 2) on *sid* of Reserves (inner):
  - –Scan Sailors: 500 page I/Os, 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
    - •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 <r, s> 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 = $\lceil \#\ of\ \ pages\ of\ \ outer\ /\ blocksize \rceil$
- 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: Joins

- Nested loop joins
- Sort-merge and hash joins
- General joins and aggregates

Readings: Chapter 14, Ramakrishnan & Gehrke, Database Systems

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

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

- 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

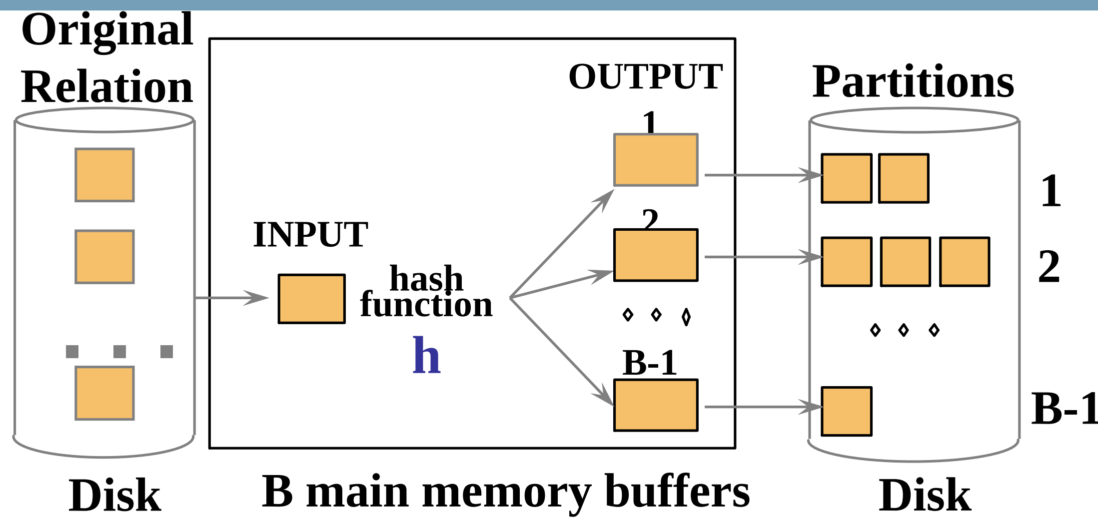
(*BNL cost: 2500 to 15000 I/Os*)

Refinement of Sort-Merge Join

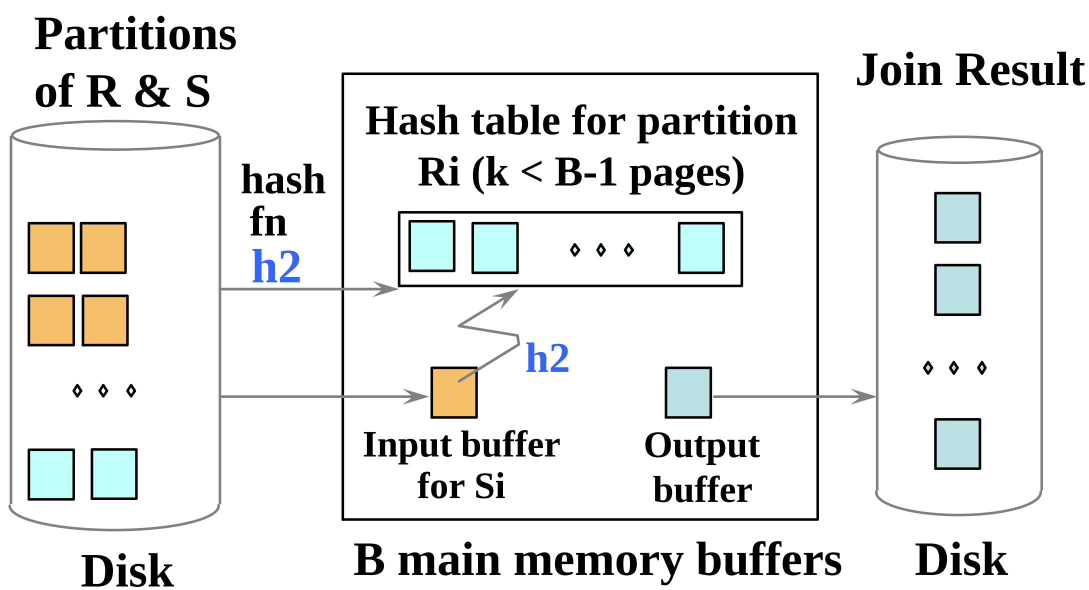
- We can combine the merging phases in the *sorting* of R and S with the merging required for the join
  - –Allocate 1 page per run of each relation, and 'merge' while checking the join condition
  - –With B > $\sqrt{L}$, where *L* is the size of the larger relation, using the sorting refinement that 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 funtion 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, 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 2nd pass

  → Need N/(B-1) ≤ B-2
  … or, roughly: B > $\sqrt{N}$
  where N is size of <u>smaller</u> 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 relations:
  2(M+N)
- In matching phase, read both relations:
  M+N I/Os

- In our running example, this is a total of 4500 I/Os

Sort-Merge Join vs. Hash Join

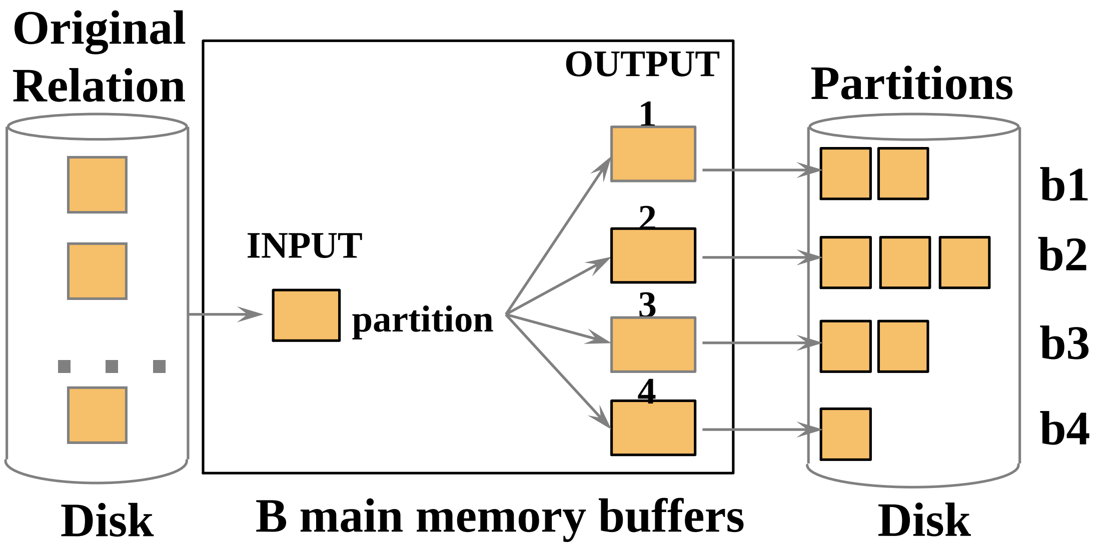
- Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os
- <u>Hash Join Pros:</u>
  - –Superior if relation sizes differ greatly
  - –Shown to be highly parallelizable *(beyond scope of class)*
- <u>Sort-Merge Join Pros:</u>
  - –Less sensitive to data skew
  - –Result is sorted (may help “upstream” operators)
  - –Goes faster if one or both inputs already sorted

Hash-Join

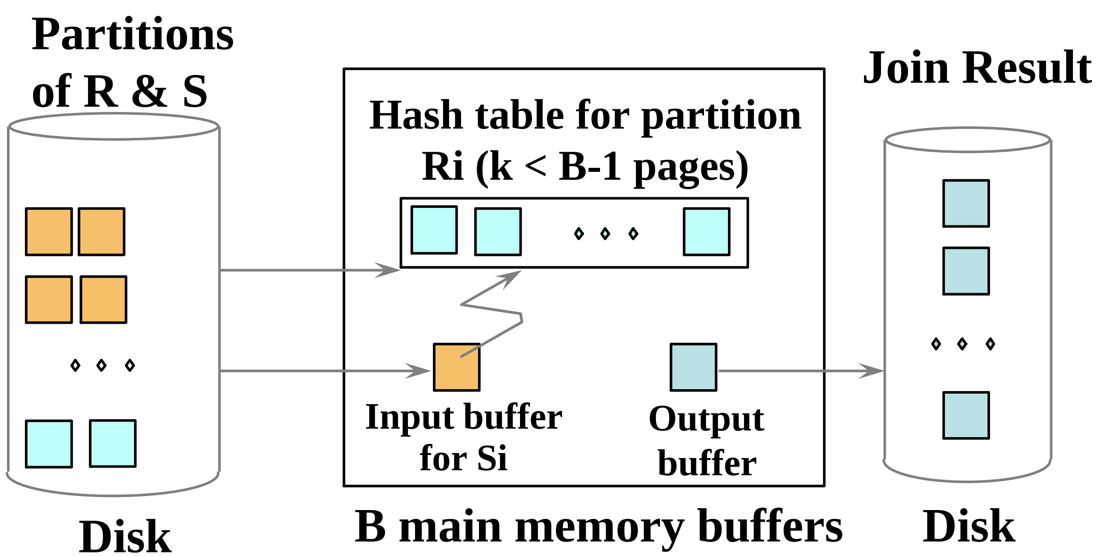
Let B = 5

Buckets:

- b1: h ∈ [1,25]
- b2: h ∈ [26,50]
- b3: h ∈ [51,75]
- b4: h ∈ [76,100]



If |F| ≤ |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

Now you try…

Query Processing: Joins

- Nested loop joins
- Sort-merge and hash joins
- General joins and aggregates

Readings: Chapter 14, Ramakrishnan & Gehrke, Database Systems

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 *h2*), 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, wecan 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

What's examinable

- Understand the logic behind relational operators
- Understand alternatives for join operator implementations
  - Be able to calculate the cost of alternatives
- Important for Assignment 3 as well

Next Lecture

- Query optimization
  - How does a DBMS pick a good query plan?