We always read smaller table into M

To compute $R - S$, $R -_B S$.

Read S
for every tuple t in R
    if t not in S
        output
    (for $-$ also keep track of those output)

To compute $S - R$, $S -_B R$

Read S
for $-$ remove all duplicates at the same time.
For every tuple t in R
    if t in S
        remove from S
        for $-_B$ remove one duplicate only
Output tuples left in S

Query Execution

query $\rightarrow$ Query Compilation $\xrightarrow{\text{query eval plan}}$ Query Execution $\rightarrow$ Result.

Query Compilation $\leftrightarrow$ DB $\leftrightarrow$ Query Execution

Query Compilation

a) Parsing. A parse tree is constructed
   • Create an algebraic expression.

b) Query Rewrite:
   • Several equivalent query expression

c) Physical plan generation
   • Each expression is converted to an evaluation plan by indicating the alg. to use.

b) and c) are the query optimizer

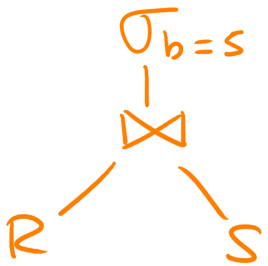
$\Rightarrow$ find best query plan:

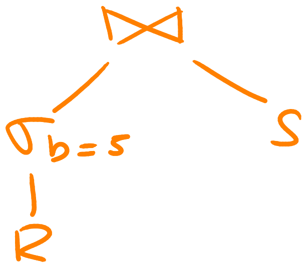
1) Which algebraic expression is the one leading to the most efficient alg.
2) For each operation in the expression which alg. will be used to answer it.
3) How should each operation pass data to the next operation.
4) How are the relations going to be accessed.

Ex: R(a, b)   S(a, c)

SELECT * from R natural Join S
WHERE b = 5

Equivalent Expressions

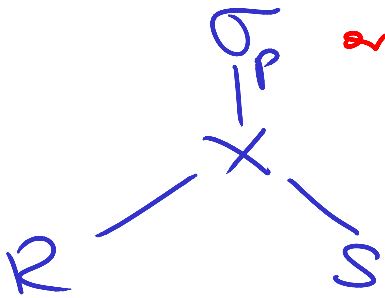




$\bowtie$

$$R \bowtie S = \sigma_p (R \times S)$$

Since we can do σ_p on the fly.



on the fly (does not need memory).

But join is common, so DBMS optimize it:

Read S
for every tuple t in R
  for every tuple s in S
    if t and s satisfy p
    output join(t, s)

— B —

Like ∩, ∪, etc. we load smaller table into memory.
But algorithm is different depending on which table is smaller:

$\cap, \cap_B, \times, \bowtie, -, -_B$.

- All commutative operations.
- Keep smaller table in memory (plus data structures for fast access).
- Plus at most one block for other table:

One pass if, approximately:

$$\min(B(R), B(S)) \le M.$$

Specifically for each of these operations:

Because they are commutative, assume

$$B(R) \ge B(S)$$

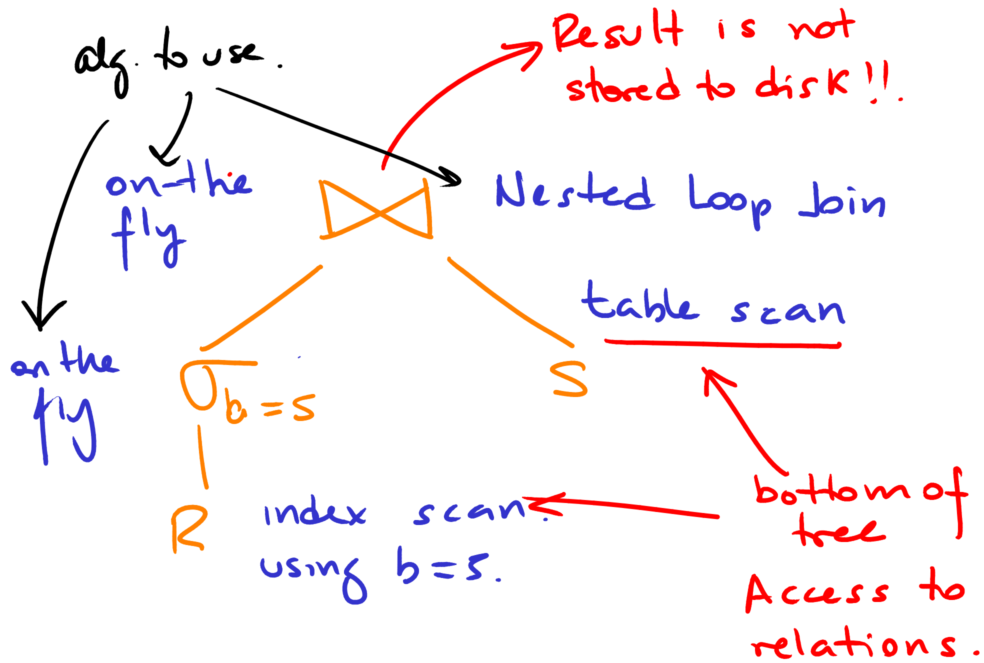
$\cap, \cap_B$

```
Read S, organize in data structure.
for every tuple t in R
    if t in S
        if bag op => output t if needed
        otherwise output t first time only.
```

$\times$

```
Read S
for every tuple t in R
    for every tuple s in S
        compute cross product, output.
```


Annotate tree with algorithms and access methods.



Estimate cost.

$\Rightarrow$ choose fastest!

Access to tuples:

- Sequential scan of heap of Rel.

or

- Using an index to scan a subset of tuples of R (index scan)

Result of query:

- Kept in memory.

Iterators:

- Many operations access only one tuple at a time.
  - read tuple
  - inspect
  - dispose
  - read next tuple...

Open() — initiates the process

GetNext() — return next tuple

close() — ends process

Example:

$$\pi_a \, \sigma_{b=3} \, R$$

π_a on the fly

|

$\sigma_{b=3}$ on the fly.

| seq scan of R

R

π and σ can be implemented as iterators

σ inspects one tuple at a time, sends one tuple at a time to π

No need to store any tuple in memory

U_B

- Similar to π:
  - We only need to inspect one tuple at a time.

    $M = 1$. regardless of size of input.

U

- Removes duplicates:
- Equivalent to. $\delta(R \cup_B S)$

The book is wrong. It states we only need to read S in M-1 and do one-tuple-at-a time for R (page 716)

We can do in one pass if

$$\delta(R \cup_B S) \le M$$

We can approximate to:

$$\delta(B(R)) + \delta(B(S)) \le M$$

We can remove duplicates as we read tuples:

if tuple already read, ignore

otherwise { output, add to read tuples.

One Pass alg. for binary operations:

$\cup, \cap, -, \times, \bowtie$

In practice set operations of two types:

- True sets: No duplicates (default).
- Bags: duplicates.

$\left.\begin{array}{l}\text{UNION}\\ \text{INTERSECT}\\ \text{EXCEPT}\end{array}\right\}$ ALL

$\Rightarrow$ Represented $\cup_B, \cap_B, -_B$

TABLE R UNION ALL TABLE S

Result contains all tuples in R plus all tuples in S.

TABLE R INTERSECT ALL TABLE S

if a tuple in has m duplicates in R and n duplicates in S result contains min(m,n) duplicates of tuple.

TABLE R EXCEPT ALL TABLE S

if a tuple in has m duplicates in R and n duplicates in S result contains min(m−n, ∅)

Parameters to measure cost

M. Amount of memory available in number of blocks

B(R) # of blocks used by heap of R

|R| # of tuples of R (book uses T(R))

V(R, a) # of different values of att a in R

In general:

$$V(R, [a_1, a_2 \ldots a_n]) = |\gamma_{a_1, a_2 \ldots a_n} R|$$

$\Rightarrow$ # of different values for tuple $a_1 \ldots a_n$

Cost Model

- We assume that the major component of cost is I/O
- Cost of read equal to cost of write
- Cost of random access of pages equal to cost of seq access.

Algorithms to answer queries.

2 main classifications.

a) based on type of algorithm:

1) Sorting based
2) Hash based
3) Index based

b) based on difficulty.

1) One-pass: Relations are read only once.
2) Two passes.
   - Read data (1st pass)
   - Process.
   - Write data.
   - Read data again. (2nd pass).

2nd pass might read diff number of blocks than 1st pass.

3) Three or more passes.
   (needed for very large relations).
   - Generalization of Two passes.

We can do it in one pass if we have enough memory to

- hold all different groups
- data structures for quick access to groups.
- any data required to compute grouping function.

In general size of tuple of result much smaller than original tuple.

So we simplify

We can do group-by in one pass if

$$B(R) \leq M$$

- min (x), max (x) } Keep current min/max
- sum (x) · Keep current sum
- count (x) Keep current count
- avg (x) Keep both current count and sum.

We cannot output tuples until we have read all input tuples.

- We must also create access structures in memory (hash tables, b+ trees) to efficiently find group tuple belongs to.

- In general
  - The amount of memory required per group is small.
  - Proportional to the number of different groups.

$$|\gamma_{a_1 \ldots a_i \, \langle \text{explist} \rangle} R| \propto V(R, a_1 \ldots a_i)$$

One Pass Alg.

1) Tuple-at-a time π, σ
   - We can read one block at a time.
     $\Rightarrow$ use <u>one</u> memory buffer.

π_a
|
R

- Read one block at a time,
- inspect each tuple, output result
- Repeat.

or

if we received tuples from another operation, one tuple at a time with no need for buffering.
(on the fly $\parallel$ no memory needed)

π_a
|
$\bowtie$

on the fly.
- Receive tuple from $\bowtie$ via iterator.
- Output result
- Repeat.

No block in memory needed.

But assume 1 block for simplicity's sake.

Other one pass unary operators.

Duplicate elimination (δ)

- Read each tuple.
  - If we have seen it, ignore
  - Otherwise output and keep track of it.

We need to keep a copy of each distinct tuple.

input tuples (iterator or from R heap)

at most $M-1$ available for distinct.

We do not need block for output.

$\Rightarrow$ tuples in result output immediately.

We can do δR in one pass as long as:

$$B(\delta(R)) \leq M-1.$$

Book uses.

$$B(\delta(R)) \leq M$$

because $M \gg 1$

So use latter for consistency.

But, how do we know $B(\delta(R))$ without calculating $\delta(R)$ first?

$\Rightarrow$ Stats.

$R(a_1, a_2 \ldots a_n)$

then.

We can use $V(R, a_1 \ldots a_n)$ and the size of the tuple in R to calculate $\delta(R)$.

Group By:

Generalization of $\delta(R)$

Remember

$$\delta(R) = \gamma^{a_1 \ldots a_n} R$$

For $\gamma^{\text{<att list>}}_{\text{<exp list>}} R$.

We need to keep track of:

- Each different value of <att list>
- Info needed to compute <exp list>.