Query Introduction 2

(This part is pretty hard to understand and record, so please have a look at the original pdf file)

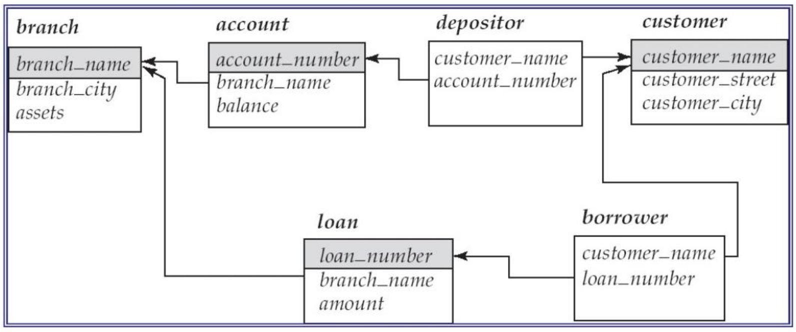
Introduction:

- n_r: number of tuples in a relation r.
- b_r: number of blocks containing tuples of r.
- l_r: size of a tuple of r.
- f_r: blocking factor of r; i.e., the number of tuples of r that fit into one block.
- $V(A, r)$: number of distinct values that appear in r for attribute A; same as the size of $\Pi_A(r)$.
- If tuples of r are stored together physically in a file, then: $b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$

Estimation of the Size of Joins

- The Cartesian product $r \times s$ contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
- If $R \cap S = \emptyset$, then $r \bowtie s$ is the same as $r \times s$.
- If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s.
- If $R \cap S$ is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
  - The case for $R \cap S$ being a foreign key referencing S is symmetric.

Banking example of estimate the size of joins:



- Number of records of *customer*: 10,000
- Number of blocks of *customer*: 400
- Number of records of *depositor*: 5,000
- Number of blocks of *depositor*: 100
- In the example query "*depositor* $\bowtie$ *customer*", *customer_name* in *depositor* is a foreign key (of *customer*), hence, the result has exactly $n_{depositor}$ tuples, which is 5000.

Also, we can calculate it as: $\frac{n_r * n_s}{V(A,s)}$

- $n_{customer}$ = 10,000.
- $f_{customer}$ = 25, which implies that $b_{customer}$ =10,000/25 = 400.
- $n_{depositor}$ = 5000.
- $f_{depositor}$ = 50, which implies that $b_{depositor}$= 5,000/50 = 100.
- *V(customer_name, depositor)* = 2,500, which implies that, on average, each customer has two accounts.

V(customer_name, customer) = 10,000 (primary key)

There have two estimates which are (10000*5000)/2500=20000 and (10000*5000)/10000=5000, choose the min of them.

🔺Some other specific method of V(A, r):

$$\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r) = \sigma_{\theta_1 \vee \theta_2}(r)$$

estimate size on different relations:

$$\begin{cases} r \cup s \Rightarrow r + s \\ r \cap s \Rightarrow \min(r, s) \\ r - s \Rightarrow r \end{cases}$$

θ forces A to take a specified value: $V(A, \sigma_\theta(r))$ = this value

If the selection condition θ is A option v, then $V(A, \sigma_\theta(r)) = V(A, r) * s$

option ⇑ (<, >, ...)

for othe options, use $option(V(A, r), n_{\sigma_\theta(r)})$

option ⇑ (min, max, ...)

If all attributes in A are from r, then $V(A, r \bowtie s) = \min(V(A, r), n_{r \bowtie s})$

If $A \begin{cases} A_1 \text{ from } r \\ A_2 \text{ from } s \end{cases}$ then

$$V(A, r \bowtie s) = \min(V(A_1, r) \cdot V(A_2 - A_1, s), V(A_1 - A_2, r) \cdot V(A_2, s), n_{r \bowtie s})$$

🔺For how to evaluate the algorithm:

对于每一个步骤，选择最简单的方法不一定会是最优解。

查询优化器：搜索所有的计划，并以成本为基础的方式选择最好的计划，并使用启发式方法来选择一个计划。

使用动态规划储存已经计算过的结果以减少不必要的计算，找到目前的最佳方案作为备选对比，如果出现更优的算法则替代。

🔺The logical of optimization algorithm:

```
// initialise bestplan[S].cost to ∞
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
         set bestplan[S].plan and bestplan[S].cost based on the best way
         of accessing S  /* Using selections on S, e.g. indices on S */
    else for each non-empty subset S1 of S such that S1 ≠ S
        P1= findbestplan(S1)
        P2= findbestplan(S - S1)
        A = best algorithm for joining results of P1 and P2
        cost = P1.cost + P2.cost + cost of A
        if cost < bestplan[S].cost
                bestplan[S].cost = cost
                bestplan[S].plan = "execute P1.plan; execute P2.plan;
                                    join results of P1 and P2 using A"
    return bestplan[S]
```

🔺Cost-Based Optimization with Equivalence Rules [基于等价规则的成本优化]

大都使用 heuristic（启发式）来处理 join 之外的步骤，将 Cost-Based Optimization 放在 join 和 selection。

Cost-Based Optimization 很容易使用新规则扩展优化器来处理不同的查询构造，但是枚举所有等价表达式的过程是非常昂贵的。

Cost-Based Optimization 的工作方法是避免重复的表达式和等效的计划，基于记忆使用动态规划，节省空间损耗。

🔺Heuristic Optimization [启发式优化]

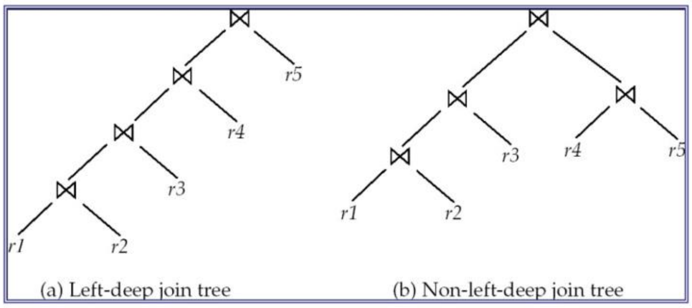
一些系统只用启发式，有些是启发式与基于成本的优化合并。

使用启发式能减少选择的次数，基于成本的优化既是使用动态规划消耗也很大。

启发式尽早执行选择(减少元组的数量)、尽早执行投影(减少属性的数量)、在执行其他类似操作之前执行限制最大的选择和连接操作(即结果大小最小的操作)。

其它的启发式：

In **left-deep join trees**, the right-hand-side input for each join is a relation, not the result of an intermediate join.



(a) Left-deep join tree (b) Non-left-deep join tree

Cost of using heuristic:

If only left-deep trees are considered, time complexity of finding best join order is O(n!), with dynamic programming this can be reduced to O(n*2^n), Space complexity remains at O(2^n)

Cost-based optimization 很昂贵，但是当查询一个非常大的数据集时这是值得的

一般优化器会对便宜的查询使用简单的启发式，对昂贵的查询使用详尽的枚举。