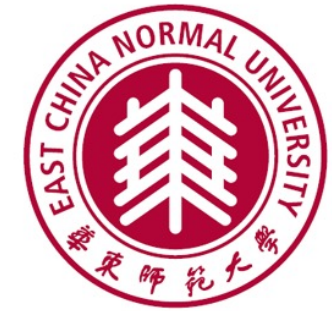
关系数据库的查询优化器



胡卉芪
华东师范大学
数据科学与工程学院
hqhu@dase.ecnu.edu.cn

DaSE
Data Science
& Engineering

Revisit数据库中算子实现

- SPJA查询中聚集算子如何实现？
  - 哈希聚集
  - 排序聚集

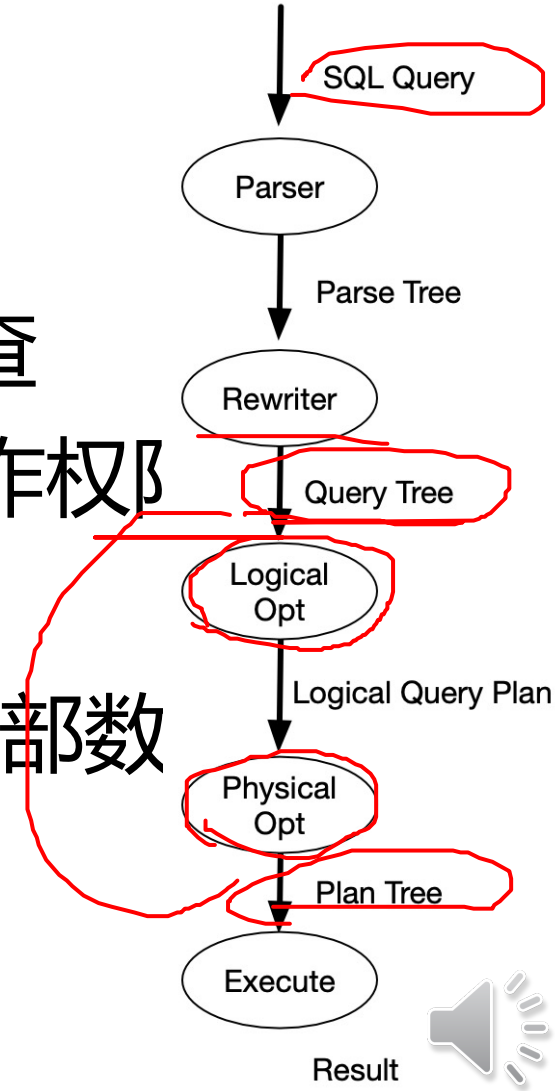
	id	dept_name	salary	name
1	10211	biology	66000.00	smith
2	10212	biology	66000.00	handsome
3	10213	history	12000.00	google
4	10214	history	12000.00	smith
5	10215	*null*	14400.00	wu

```
# 找出每个系的教师平均工资
SELECT dept_name, avg(salary) AS avg_salary
FROM instructor
GROUP BY dept_name;
```

	dept_name	avg_salary
1	*null*	14400.000000
2	biology	66000.000000
3	history	12000.000000

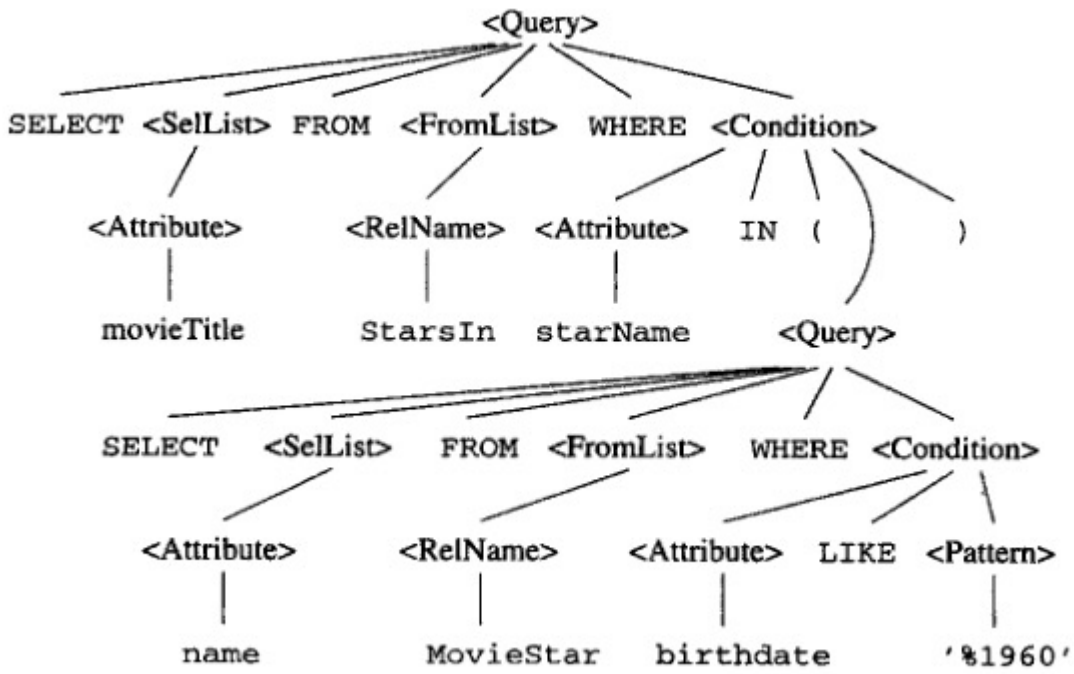
Revisit关系数据库查询执行流程

- SQL解析、重写
  - –对SQL进行词法分析和语法分析
  - –根据Metadata对SQL进行语义检查
  - –根据Metadata中用户权限检查操作权限
  - –视图重写
  - –把数据库对象的外部名称转换为内部数据结构称之为Query Tree



语法树数据结构

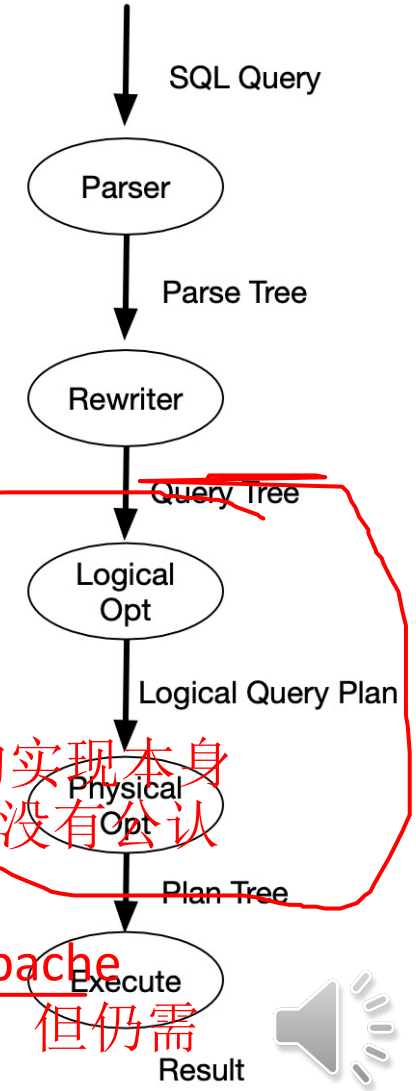
- 将SQL中元素识别出来构成数据结构




```
SELECT movieTitle
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);
```

查询优化器

- 目标：选择一个高效的查询执行计划
  - –查询树结构→树形的物理计划
  - –集中式数据库：磁盘存取块数(I/O代价)，处理机时间(CPU代价)和查询的内存开销
  - –分布式数据库：总代价=I/O代价+CPU代价+通信代价
  - –基于代价查询优化分类 ：
  - – 逻辑（代数）优化
    - 关系代数表达式的等价变化
  - – 物理优化：
    - 存取路径：索引、基本表
    - 底层操作算子的选择：连接算子的选择、聚合算子选择等
    - 多表连接顺序选择



注：查询优化器的实现本身是非常ad-hoc的，没有公认的代码框架。

有些解耦合的Apache Calcite，HAWQ等，但仍需集成系统时优化

经典案例-访问路径选择

路径选择

- **一、选择操作的实现**
- [例] Select * from student where <条件表达式> ；
- 考虑<条件表达式>的几种情况：
  - C1：无条件；
  - C2：Sno = '200215121'；
  - C3：Sage>20；
  - C4：Sdept = 'CS' AND Sage>20；

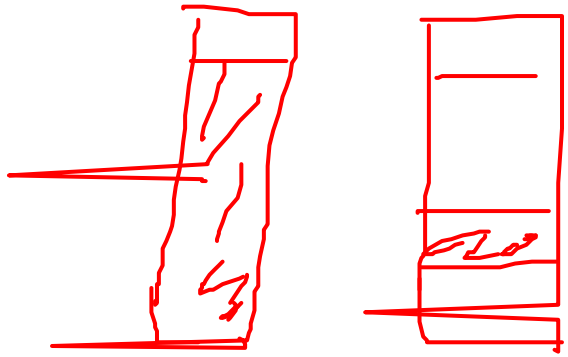
- **选择操作典型实现方法：**
- **1、简单的全表扫描方法**
  - –对查询的基本表顺序扫描，逐一检查每个元组是否满足选择条件，把满足条件的元组作为结果输出。
  - –适合小表，不适合大表。
- **2、索引(或散列)扫描方法**
  - –适合选择条件中的属性上有索引(例如B+树索引或Hash索引)。通过索引先找到满足条件的元组主码（主键，primary key）或元组指针，再通过元组指针直接在查询的基本表中找到元组。

[例]以C2为例，Sno＝ ‘200215121’，并且Sno上有索引(或Sno是散列码)

使用索引(或散列)得到Sno为‘200215121’元组的指针，通过元组指针在student表中检索到该学生

[例]以C3为例，Sage>20，并且Sage上有B+树索引

使用B+树索引找到Sage＝20的索引项，以此为入口点在B+树的顺序集上得到Sage>20的所有元组指针。通过这些元组指针到student表中检索到所有年龄大于20的学生。



更复杂的问题-多路路径选择

[例-C4]以C4为例，Sdept＝ ‘CS’ AND Sage>20，如果Sdept和Sage上都有索引：

算法一：分别用上面两种方法分别找到Sdept＝ ‘CS’的一组元组指针和Sage>20的另一组元组指针

- 求这2组指针的交集
- 到student表中检索
- 得到计算机系年龄大于20的学生

算法二：找到Sdept＝ ‘CS’的一组元组指针，

- 通过这些元组指针到student表中检索
- 对得到的元组检查另一些选择条件(如Sage>20)是否满足
- 把满足条件的元组作为结果输出。

逻辑优化：一些基本关系表达式的转换

关系等价变换

- 目标：通过对关系代数表达式的等价变换来提高查询效率
- 如果两个关系代数表达式在每个合法数据库实例上生成相同的元组集，则称这两个关系代数表达式是等价(**equivalent，** 表示为≡)的
  - 注：元组的顺序无关
  - 可以用第二个形式替换第一个形式的表达式，反之亦然

典型案例

- [例] 求选修了课程C2的学生姓名。用SQL语言表达：
  - SELECT S.SN
  - FROM S,SC
  - WHERE S.S#=SC.S# AND SC.C#= 'C2' ;
- 假定学生 - 课程数据库中有1000个学生记录，10000个选课记录，其中选修C2课程的选课记录为50个。
- 系统可用多种等价的关系代数表达式来完成这一查询
  - Q1= $\pi_{SN}(\sigma_{S.S\#=SC.S\#\wedge SC.C\#='C2'}(S\times SC))$
  - Q2= $\pi_{SN}(\sigma_{SC.C\#='C2'}(S\infty SC))$
  - Q3= $\pi_{SN}(S\infty\sigma_{SC.C\#='C2'}(SC))$
- 查询的评估计划之间的成本差异可能很大
  - 执行时间可能是秒级与天级的差异

•常用的等价变换规则：

•1. 连接、笛卡尔积交换律

• 设E_1和E_2是关系代数表达式，F是连接运算的条件，则有

$E_1 \times E_2 \equiv E_2 \times E_1$

$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$ 这有什么用？

$E_1 \underset{F}{\bowtie} E_2 \equiv E_2 \underset{F}{\bowtie} E_1$

•2. 连接、笛卡尔积的结合律

• 设E_1，E_2，E_3是关系代数表达式，F_1和F_2是连接运算的条件，则有

• $(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$

• $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$ 决定连接顺序

• $(E_1 \underset{F_1}{\bowtie} E_2) \underset{F_2}{\bowtie} E_3 \equiv E_1 \underset{F_1}{\bowtie} (E_2 \underset{F_2}{\bowtie} E_3)$

•4. 选择与笛卡尔积的交换律

• 如果F中涉及的属性都是E_1中的属性，则

$$\sigma_F(E_1 \times E_2) \equiv \sigma_F(E_1) \times E_2$$

• 如果$F=F_1 \wedge F_2$，并且F_1只涉及E_1中的属性，F_2只涉及E_2中的属性，则由上面的等价变换规则可推出：

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$$

• 若F_1只涉及E_1中的属性，F_2涉及E_1和E_2两者的属性，则仍有

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1) \times E_2)$$

它使部分选择在笛卡尔积前先做。

实现逻辑优化：基于规则优化

基于规则的优化

- 在逻辑优化阶段，我们对查询树依次执行设置的启发式规则，如果满足，则执行规则
  - 启发式规则
    - 我们认为这些规则通常情况下会使得查询计划更优
  - 如何实现？
    - 通常可以调整查询树的数据结构
    - 有些规则需要独特编码方式实现

启发式规则

- 选择运算应尽可能先做（最重要、最基本）
  - 选择下推
  - 如何实现？
    - 编码时将选择操作绑定到扫描算子上。

典型案例

- [例] 求选修了课程C2的学生姓名。用SQL语言表达：
  - SELECT S.SN
  - FROM S,SC
  - WHERE S.S#=SC.S# AND SC.C#= 'C2' ;
- 假定学生 - 课程数据库中有1000个学生记录，10000个选课记录，其中选修C2课程的选课记录为50个。
- 系统可用多种等价的关系代数表达式来完成这一查询
  - $Q1 = \pi_{SN}(\sigma_{S.S\#=SC.S\# \wedge SC.C\#='C2'}(S \times SC))$
  - $Q2 = \pi_{SN}(\sigma_{SC.C\#='C2'}(S \infty SC))$
  - $Q3 = \pi_{SN}(S \infty \sigma_{SC.C\#='C2'}(SC))$
- 查询的评估计划之间的成本差异可能很大
  - 执行时间可能是秒级与天级的差异

其他启发式规则

- 一些比较好用的规则
  - 把投影运算和其他运算结合
  - 条件化简
  - 子查询展开
  - 外连接消除

条件化简

- 利用等式和不等式的性质，可以将WHERE、HAVING和ON条件化简
- 例：
  - 常量传递：使得条件分离后有效实施“条件下推”
    - Col_1=Col_2 AND Col_2=3 化简为 Col_1=3 AND Col_2=3
  - 消除死码：化简条件，去除不必要的条件
    - WHERE（0>1 AND s1=5）条件恒假，则不必执行该SQL
  - 表达式计算：加快计算效率
    - WHERE Col_1 = 1+2 变换为 WHERE Col_1 = 3
  - 不等式变换：化简条件，去除不必要的条件
    - a>10 AND b = 6 AND a>2 化简为 b=6 AND a>10
  - 谓词传递闭包：加速计算，有效实施“条件下推”
    - a>b AND b>2 推导出 a>b AND b>2 AND a>2

子查询展开

- 又称子查询上拉，实质是把某些子查询重写为等价的多表连接操作
- 优势：连接方法和连接顺序选择更加灵活
- 例：
  - SELECT * FROM
    t1,(SELECT * FROM t2 WHERE t2.a2>10) v_t2
    WHERE t1.a1<10 AND v_t2.a2<20;
  - 可优化为：
  - SELECT * FROM
    t1,t2
    WHERE t1.a1<10 AND t2.a2<20 AND t2.a2>10

外连接消除

- 右表存在非Null条件

Select * From T1 Left Join T2 on T1.c1 = T2.c1 Where T2.c2 > 0 ;

注意这里SQL执行顺序是先做Join,再做选择

->Select * From T1,T2 Where T1.c1 = T2.c1 and R.c2 > 0 ;

- 满足传递性链式非Null条件

Select * From T1 Left Join T2 on T1.c1 = T2.c1 Left Join T3 on T2.c2 = T3.c2 Where T3.c3 > 0 ;

->Select * From T1,T2,T3 Where T1.c1 = T2.c1 and T2.c2 = T3.c2 and T3.c3 > 0 ;

物理优化

物理优化

- 逻辑优化改变查询语句中操作的次序和组合，不涉及底层的存取路径
- 对于一个查询语句的算子有很多实现方案，它们的执行效率不同
- 物理优化就是要选择高效合理的操作算子、数据访问路径和查询树结构，求得相对最优的查询计划
  - 基于规则的优化
  - 基于代价的优化

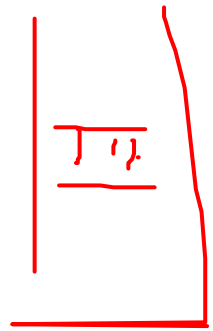
基于启发式规则的优化

--选择操作的启发式规则

--连接操作的启发式规则

路径选择操作的启发式规则

- 对于小关系(<1000行):
- 1. 使用全表顺序扫描，即使选择列上有索引
- 对于大关系：
- 2. 对于选择条件是主码 = 值的查询
  - 查询结果最多是一个元组，可以选择主码索引
- 3. 对于选择条件是非主属性 = 值的查询，并且选择列上有索引
  - 估算查询结果的元组数目:比例较小(<10%)可以使用索引扫描方法, 否则全表顺序扫描
- 4. 对于选择条件是属性上的非等值查询或者范围查询，并且选择列上有索引
  - 估算查询结果的元组数目:比例较小(<10%)可以使用索引扫描方法, 否则全表顺序扫描



1000行，10%等数字称为 Magic Numbers

选择操作的启发式规则

- 5. 对于用AND连接的多路选择条件
  - 如果有涉及这些属性的组合索引
    - 优先采用组合索引扫描方法
  - 如果某些属性上有一般的索引
    - 选择选择率最低的一个索引
- 6. 对于用OR连接的析取选择条件，一般使用全表顺序扫描 ???

连接操作的启发式规则

- 1. 如果2个表都已经按照连接属性排序
  - 选用排序-合并方法
- 2. 如果前表较小，后表在连接属性上有索引
  - 选用Nest Loop索引连接方法
- 3. 如果上面2个规则都不适用，其中一个表较小
  - 选用Hash join方法或内存Nest Loop Join
- 4. 否则采用Block Nest Loop Join或Grace Hash Join

基于代价的优化

基于代价模型

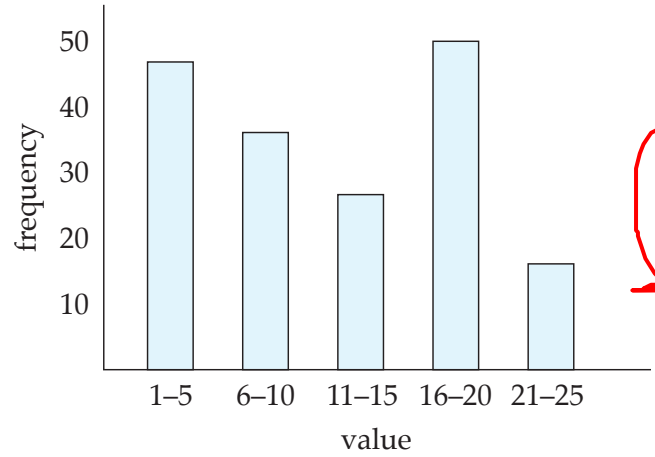
- 必要条件
  - 实现数据统计信息，用于帮助计算代价
- 核心代价模型
  - 访问路径
  - 多表连接顺序
  - 执行算子(很多情况下使用规则代替)
    - 连接、聚集

统计信息

1. 对每个基本表
   - 该表的元组总数(N)
   - 元组长度(l)
   - 占用的块数(B)
   - 占用的溢出块数(BO)

2. 对基表的每个列
   - 该列不同值的个数(m)
   - 选择率(f)
     - ➢如果不同值的分布是均匀的，f＝1/m
     - ➢如果不同值的分布不均匀，则每个值的选择率＝具有该值的元组数/N
   - 该列最大值
   - 该列最小值
   - 该列上是否已经建立了索引
   - 索引类型(B+树索引、Hash索引)

3. 对索引(如B+树索引)
   - 索引的层数(L)
   - 不同索引值的个数
   - 索引的选择基数S(有S个元组具有某个索引值)
   - 索引的叶结点数(Y)



等宽/等深直方图

两个关键的统计信息

- 直方图->条件选择率
  - 决定路径访问顺序
  - 决定连接算法
- 不同值个数
  - 决定连接后表的行数

访问路径代价模型

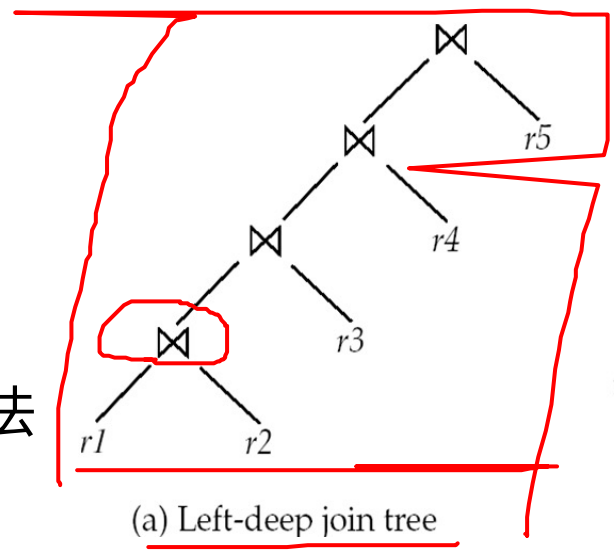
- 1. 全表扫描算法的代价估算公式
  - 如果基本表大小为B块，全表扫描算法的代价 cost = B
  - 如果选择条件是码 = 值，那么平均搜索代价 cost = B/2
- 2. 索引扫描算法的代价估算公式
  - 如果选择条件是码 = 值
    - 如 [例-C2] ，则采用该表的主索引
    - 若为B+树，层数为L，需要存取B+树中从根结点到叶结点L块，再加上基本表中该元组所在的那一块，所以cost=L+1
  - 如果选择条件涉及非码属性
    - 如 [例-C3] ，若为B+树索引，选择条件是相等比较，S是索引的选择基数(有S个元组满足条件，选择率)
    - 最坏的情况下，满足条件的元组可能会保存在不同的块上，此时，cost=L+S
  - 如果比较条件是 > ，> = ，< ，< = 操作
    - 假设有一半的元组满足条件就要存取一半的叶结点
    - 通过索引访问一半的表存储块cost=L+B/2
    - 如果可以获得更准确的选择基数，可以进一步修正B/2

连接代价模型

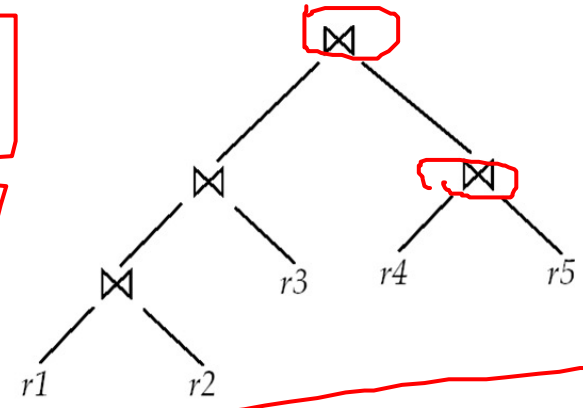
- 事实上已不太用
  - 嵌套
    - Nr*Ns
  - 哈希
    - 3Nr+3Ns
  - 排序
    - 两边有序
      - Nr+Ns
      - 3Nr+Ns

多表连接顺序搜索策略

- 连接顺序（树形结构+位置）
- 策略
  - 启发式
  - 启发式+基于代价的搜索
    - 枚举
    - DP算法
  - 随机算法(n>10)：遗传算法



(a) Left-deep join tree

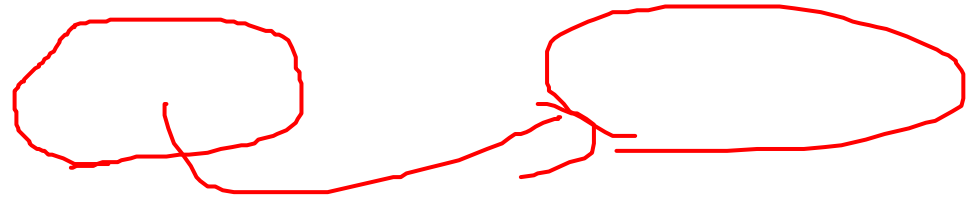


(b) Non-left-deep join tree

- 左深树
  - 每个连接算子的右侧输入是一个关系（基本表），而不是连接后的中间结果
  - 基于成本的优化是昂贵的，但对于大型数据集的查询是有价值的（典型的查询具有较小的n，通常小于10）

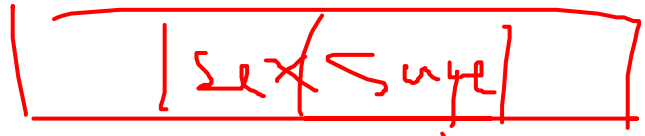
总结：如何设计一个简单查询优化器

- 逻辑优化
  - 定义一些启发式规则
- 物理优化
  - 先决定访问路径
  - 启发式规则决定连接顺序与连接方式



查询优化器设计的几个难题

- 统计信息的选择自动收集与更新
- 选择条件下数据的条件分布，通常导致估算不准确。
- 计划不准导致的查询超时无法从理论上避免



- Thanks