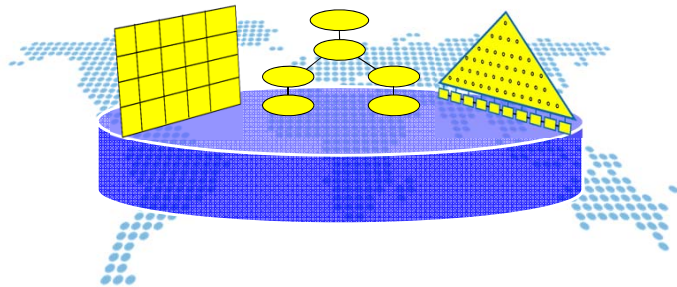


数据库系统 查询处理 (1)

陈世敏
(中科院计算所)



Outline

- 查询处理概述
 - 系统目录 (System Catalog)
 - 查询执行方式
 - 关系操作实现的常见方式
- 排序和外排序

系统目录 (System Catalog)

- 又称为目录表 (Catalog Table), 或数据字典 (Data Dictionary), 或目录 (Catalog)
- 存储数据的元信息
 - 表的信息
 - 索引的信息
 - 视图的信息
 - 其它: grant, check constraints, trigger等

对于每个Table

- 表名
- 表的存储方式
- 每个属性名和类型
- 每个索引名
- 表上的完整性约束
- 统计信息
 - 记录数 (Cardinality)
 - 表大小: 页数
 - 属性值的分布特征: 可能包括属性取值的大致个数, 统计直方图

对于每个索引

- 索引名称
- 索引结构类型
- 索引key的组成属性
- 统计信息
 - key的个数
 - 索引大小: 页数
 - 树结构索引高度
 - 索引范围: 最小和最大的键值

对于每个视图

- 视图的名称
- 视图的定义

目录的存储

- 有趣的是: 目录也可以存储在Relational Table中
- 例如:

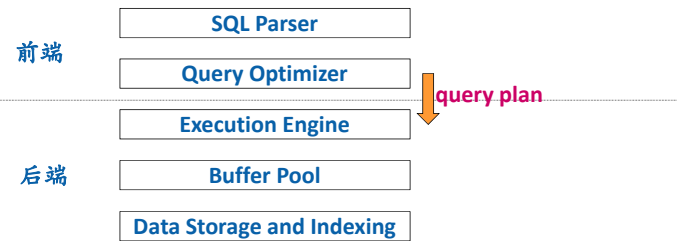
```
Attribute_catalog(  
    attr_name string,  
    rel_name string,  
    type enum)
```

Outline

- 查询处理概述
 - 系统目录 (System Catalog)
 - 查询执行方式
 - 关系操作实现的常见方式
- 排序和外排序

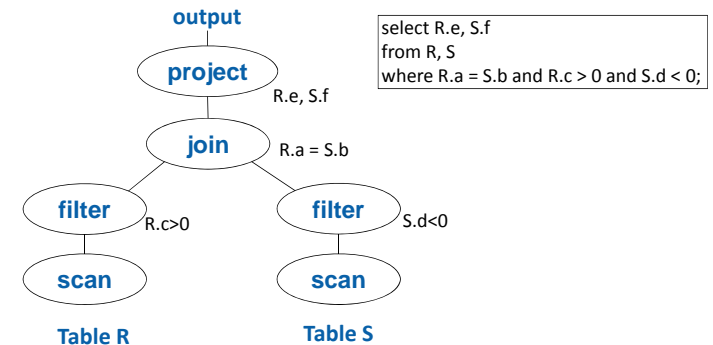
查询计划 (Query Plan)

- Query optimizer 产生
- Execution engine 按照 query plan 执行



Operator Tree

- Query plan 最终将表现为一棵Operator Tree



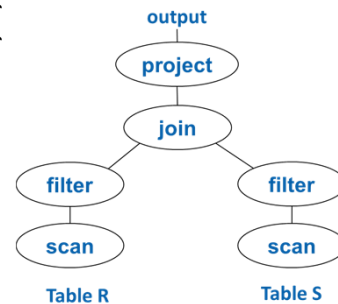
Operator Tree

- Query plan 最终将表现为一棵Operator Tree

- 每个节点代表一个运算
- 运算的输入来自孩子节点
- 运算的输出送往父亲节点

实现方法

- Operator-at-a-time
 - 完整执行每个Operator
- Tuple-at-a-time
 - 迭代求解
- 多线程流水线求解



方法1: Operator-at-a-time 完整执行每个Operator

- Scan table R + filter

- 完成整个操作
- 生成中间结果表A

- Scan table S + filter

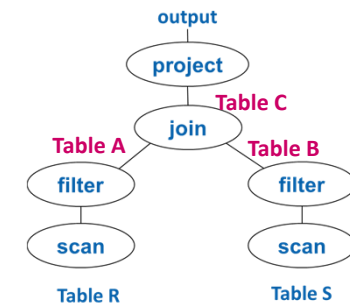
- 完成整个操作
- 生成中间结果表B

- Join A和B

- 完成整个Join
- 生成中间结果表C

- 最后在表C上计算投影

- 得到最终结果表



方法1的问题?

- 中间结果表的代价

- 外存代价: 如果不能放入内存, 那么就会有读写的I/O
- CPU Cache miss: 即使可以放入内存, 但是不能放入CPU Cache, 就会产生大量的Cache miss

- 我们希望减少中间结果的大小, 降低中间结果的代价

方法2: Tuple-at-a-time 迭代求解

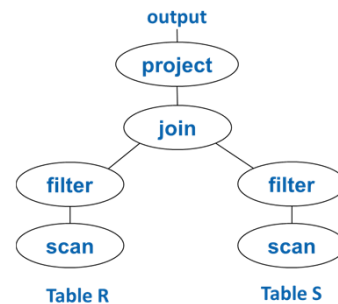
- 又称为Volcano方式

- 每个Operator设计成为具有统一的函数接口

- Open(): 初始化, 分配资源, 建立数据结构等
 - 进一步调用子Operator的open()
- GetNext(): 获得下一条Operator的处理结果
 - 调用子Operator的GetNext(), 在此基础上, 进行本Operator所设定的关系运算, 得到一条结果
- Close(): 结束, 释放资源
 - 调用子Operator的close()

方法2: Tuple-at-a-time 迭代求解

- 执行方式是Pull方式
- 建立了Operator tree之后
- 调用根的Open()初始化
- 循环调用根的GetNext()
 - 父结点将调用子结点的GetNext()
 - 生成一条条的结果
- 最后, Close()

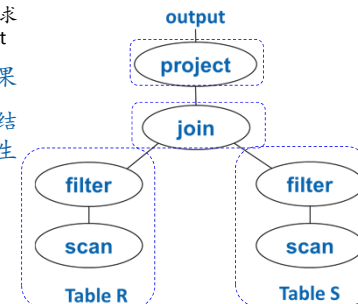


方法3: 多线程流水线

- 把Operator tree分到多个线程中

- 例如, 图中把每组filter+scan在一个线程求解, 一个线程求解join, 一个线程求解project

- 线程之间通信传递中间结果
- 一个线程等待孩子结点的结果来到后, 进行计算, 产生结果, 发送给父亲结点
- 这是一种push方式



Outline

- 查询处理概述
 - 系统目录 (System Catalog)
 - 查询执行方式
 - 关系操作实现的常见方式
- 排序和外排序

常用高效方式

- 哈希
 - 可以找到相同的属性的记录
- 排序
 - 如果记录有序, 那么很多操作就可以容易实现
 - 例如去重
- 索引
 - 可以快速地查询
 - 树结构索引提供了顺序

Outline

- 查询处理概述
- 排序和外排序
 - 排序的应用场景
 - 内存排序回顾
 - 外存排序
 - 使用B⁺-Tree获得排序数据

什么时候需要排序?

- Order by子句
 - 当SQL SELECT语句中使用了Order by子句
 - 用户明确要求按照给定的顺序产生查询结果
 - 那么, 就需要进行排序
- B⁺-tree的bulkloading
 - 在已知的数据上新建索引
 - 先按照索引键对数据进行排序, 然后从叶子到根, 一层层地建立B⁺-Tree
- 使用排序来实现算法
 - 例如: 去重distinct, 连接join, 分组group by

排序的迭代接口

- Open
 - 排序操作通常需要做大量的准备工作
 - 需要扫描全部的输入数据，调用子结点的GetNext
 - 产生中间数据结构
- GetNext
 - 生成下一条有序的结果记录
- Close
 - 释放资源

Outline

- 查询处理概述
- 排序和外排序
 - 排序的应用场景
 - 内存排序回顾
 - 外存排序
 - 使用B⁺-Tree获得排序数据

有哪些排序算法？

- $O(N^2)$
 - 冒泡排序，选择排序，插入排序等
- $O(N\log N)$
 - 快速排序，归并排序，堆排序
- $O(N)$
 - 基数排序

快速排序？

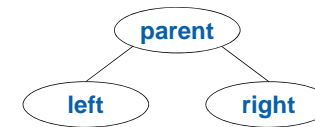
```
quicksort(A[], lo, hi)
    if lo < hi then
        choose pivot;
        p = partition(A, lo, hi, pivot);
        quicksort(A, lo, p - 1);
        quicksort(A, p + 1, hi);
```

归并排序

- 基本操作：归并两个有序的序列
 - 怎么实现？
- 开始，所有的单个元素都是长度为1的有序序列
- 归并长度为1的有序序列 → 长度为2的有序序列
- 归并长度为2的有序序列 → 长度为4的有序序列
- ...
- 最终得到整个有序序列

堆排序

- 这里的堆 (Heap) 是指二叉树，满足
 - $\text{left} \geq \text{parent}$
 - $\text{right} \geq \text{parent}$
- 建立堆之后，可以从堆顶取出最小的元素
- 更新堆的代价等于树的高度 $O(\log N)$
- 这个堆与 malloc/free heap 不是一个概念



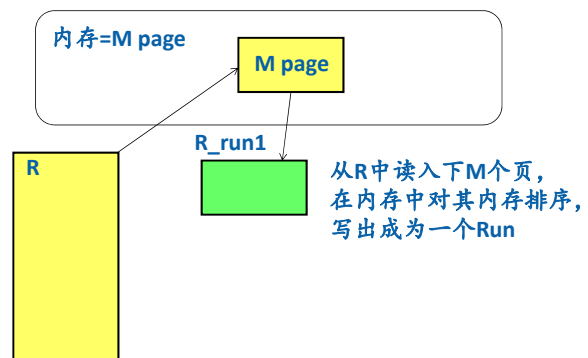
Outline

- 查询处理概述
- 排序和外排序
 - 排序的应用场景
 - 内存排序回顾
 - 外存排序
 - 使用 B⁺-Tree 获得排序数据

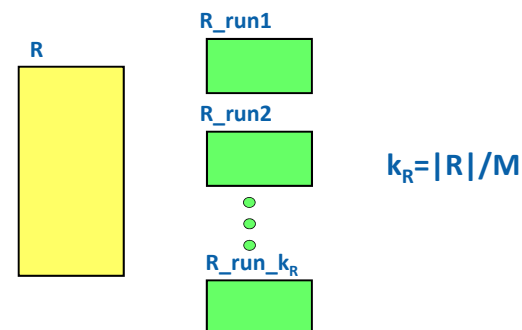
外存排序

- 输入表 R，大于可用内存大小 M
- 步骤
 - Run Generation，生成有序的数据段
 - Merge Runs，归并数据段

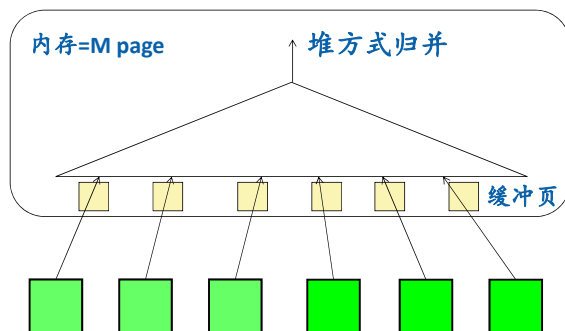
Run Generation



Run Generation



Merge



外存排序的I/O开销

- 假设一次Merge就可以完成
- I/O开销是多少?
 - 不考虑输出
 - Run generation: 读 $|R|$ 页, 写 $|R|$ 页
 - Merge run: 读 $|R|$ 页
 - 所以共读写 $3|R|$ 页

单层Merge

- 可以最多把M-1个Run归并为1个Run
 - M-1个输入缓冲页
 - 1个输出缓冲页
 - 每次扫描全部内存记录，找到最小的输出

多层Merge

- 如果数据量超过了M-1个Run
- 那么就需要多层Merge，需要的层数为
 - $\log_{M-1} \frac{|R|}{M}$

Replacement Selection内存排序算法

- 目的：产生更长的Run
- 思路（从小到大排序）
 - 读入数据占用所有内存；
 - 找到最小的记录输出，last_key=输出记录的键；
 - while(1){
 - 当前内存有一个空位，所以可以读入一条新记录；
 - 在内存所有记录中找到 $\geq \text{last_key}$ 的最小键；
 - if(没找到){
 - 当前Run结束；break;
 - 输出相应的记录，last_key=输出记录的键；

Replacement Selection

- 优点
 - 可以证明：算法产生的Run的平均长度为2M
 - 所以，可以减少Run的数量
 - 有可能减少Merge层数
- 缺点
 - 内存排序算法效率低

考虑实际情况

- 假设计算机的可用内存为10GB
- 为了提高I/O效率, 使用1MB大小的读写缓冲
 - 每次进行1MB的近似于顺序读写的操作
- 那么, 内存单次Merge可以归并
 - $10\text{GB}/1\text{MB} - 1 \approx 10^4$
 - 那么单次Merge可以支持 $10^4 * 10\text{GB} = 10^5\text{GB} = 100\text{TB}$
- 实际上, 在大部分情况下, 已经不需要Replacement Selection算法了

可以更快吗?

- 如果实际需要排序的数据比100TB小很多, 那么在Merge阶段, Run的个数就少很多
- 于是, Merge阶段的内存就不需要完全占用
- 那么, 我们可以利用这些多余的内存空间缓冲最后一个Run的部分数据, 从而进一步减少I/O

Outline

- 查询处理概述
- 排序和外排序
 - 排序的应用场景
 - 内存排序回顾
 - 外存排序
 - 使用B⁺-Tree获得排序数据

B⁺-Tree实际上对于键是有顺序的

- B⁺-Tree的叶子结点给出了顺序
- 如果是聚簇索引
 - 这个顺序也是表中记录的存储顺序
 - 可以直接访问, 非常高效
- 如果是二级索引
 - 这个顺序给出了RecordID
 - 通过RecordID访问Record需要随机I/O
 - 具体的代价需要考虑访问的数据量

二级索引访问举例

- 假设可用内存1GB
- 数据表有10GB大，每个记录100B
- 硬盘
 - 顺序访问速度：100MB/s
 - 随机访问速度：10ms/次
- 那么排序的时间
 - $3 \times 10\text{GB} / 100\text{MB} = 300\text{ s}$
- 这相当于随机访问 $300\text{s} / 10\text{ms} = 30000$ 个记录的时间
- 换言之，当访问少于30000个记录时，采用二级索引是更快的，否则直接排序更快

小结

- 查询处理概述
 - 系统目录 (System Catalog)
 - 查询执行方式
 - 关系操作实现的常见方式
- 排序和外排序
 - 排序的应用场景
 - 内存排序回顾
 - 外存排序
 - 使用B⁺-Tree获得排序数据