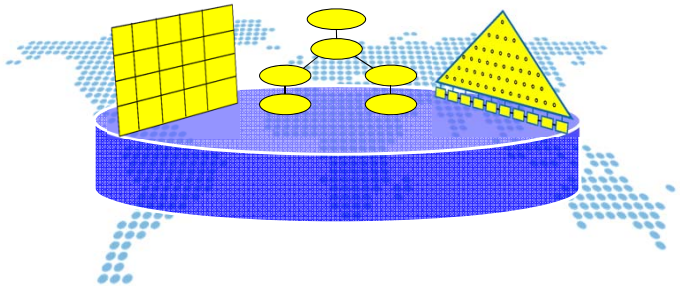
数据库系统
查询处理（2）

陈世敏
（中科院计算所）



上节相关内容

- 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

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作
- 内存数据库

Selection (选择)

$\sigma_{Major='计算机'}(Student)$

```
select *
from Student
where Major = '计算机';
```

选择所有计算机系学生记录

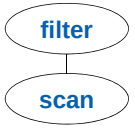
ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2014	95
…	…	…	…	…	…	
…	…	…	…	…	…	
…	…	…	…	…	…	

单个选择条件

- 情况1：无索引、未排序的数据
- 情况2：无索引、排序的数据
- 情况3：B+-Tree索引
- 情况4：哈希索引、等值比较

情况1：无索引、未排序的数据 $\sigma_{R.attr\ op\ value}(R)$

- 场景
  - R.attr上没有索引
  - 关系表的数据对于R.attr是**无序**的
- 访问路径（Access Path）：文件扫描
  - 顺序访问R每个页，在页中依次访问各个记录，对于每个记录求解选择条件



- I/O代价
  - 假设R共有M_R个数据页
  - 那么代价为M_R

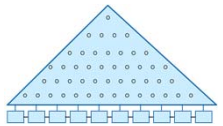
情况2：无索引、排序的数据 $\sigma_{R.attr\ op\ value}(R)$

- 场景
  - R.attr上没有索引
  - 关系表的数据对于R.attr是**有序**的
- 访问路径（Access Path）
  - 二分查找定位满足查询条件的第一个记录
  - 从这个记录开始，读取满足条件的记录
    - 例如，升序排列，R.attr > value，那么就向表末尾方向扫描

- I/O代价：优于情况1
  - 假设R共有M_R个数据页，满足条件的记录占据D个数据页
  - 总代价= $\log_2 M_R + D$

情况3：B+-Tree索引 $\sigma_{R.attr\ op\ value}(R)$

- 场景
  - R.attr上有B+-Tree索引



- 访问路径1：B+-Tree索引
  - 聚簇索引：叶子结点就是数据页，最高效
  - 非聚簇索引：叶子结点存RecordID，随机读取记录

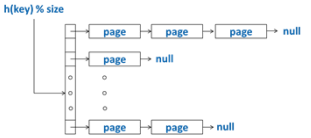
- 访问路径2：文件扫描
  - 同情况1，当非聚簇索引+很多结果时，顺序扫描代价可能优于大量随机I/O读取记录的代价

非聚簇索引的代价和访问优化

- 非聚簇索引代价
  - 假设B+-Tree的查找代价为H次I/O
  - 假设符合条件索引项有m个占据L个叶子结点页，指向m个记录
  - 那么总代价：H+L+m
- 访问优化：先排序后读取
  - 先对所有符合条件的RecordID进行排序
  - 再按照RecordID的排序的顺序访问数据页
- 为什么？
  - 空间局部性：若一个数据页中有多条符合条件的记录，只需读一次
  - 减少磁头移动距离：使磁头只向一个方向移动，节省seek时间

情况4：哈希索引、等值比较

$\sigma_{R.attr\ op\ value}(R)$



- 场景
  - R.attr上有哈希索引
  - 比较操作op是等值比较
- 访问路径1：哈希索引
  - 查找哈希索引找到满足条件的RecordID
  - 随机访问读取记录

Hash Table search

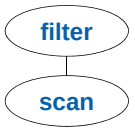
- 访问路径2：文件扫描
  - 同情况1，当有很多结果时，顺序扫描代价可能优于大量随机I/O读取记录的代价
- 访问路径1的I/O代价
  - 假设哈希表的平均查找代价为H次(与链长有关)I/O
  - 假设符合条件的记录有m个，需m次随机I/O
  - 那么总代价：H+m

多个选择条件

- 多个选择条件由AND, OR, NOT等连接在一起
- 如何求解？
  - 方法1：文件扫描
  - 方法2：先求解一个合取条件
  - 方法3：使用位图索引
  - 方法4：利用多个索引
- 注意
  - 大部分RDBMS可以有效地求解不含OR, NOT的条件
  - 而对于包含OR,NOT的条件，求解的代价较大

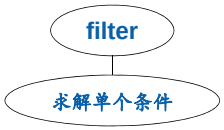
方法1：文件扫描

- 文件扫描是最通用的方法
- 在扫描的基础上，可以对于每个记录求解任意复杂的选择条件，判断选择条件是否成立



方法2：先求解一个合取条件

- 转换为合取范式（数理逻辑）
  - 最外层是AND
  - 内层可以有OR
    例如： (a>0) AND (b<0 OR C>100)
- 先求解一个单个条件的合取项
  - 用前面单个条件的4种方法之一
- 再进一步求解其它条件



方法3：使用位图索引

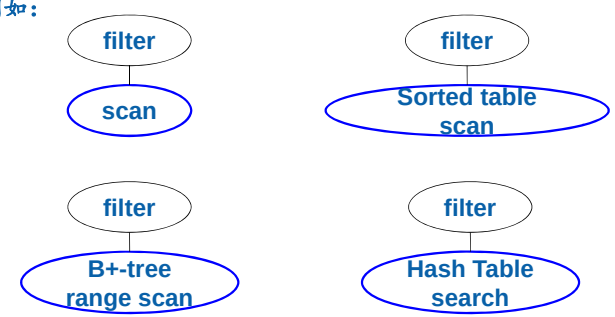
- 场景
  - 选择条件涉及的每个列都有位图索引
- 方法
  - 获得每个选择条件的位图
  - 计算选择条件的总位图
    - AND➔按位与
    - OR ➔按位或
    - NOT ➔按位取反

方法4：利用多个索引

- 场景
  - 多个选择的属性都有索引
- 方法：与方法3思路很像
  - 得到符合每个子条件的RecordID集合
  - 选择条件➔RecordID集合操作
    - AND ➔集合交集
    - OR ➔集合并集
    - NOT ➔全集 - 集合

选择Operator

例如：



ScanOperator

- Open( )：初始化，分配资源，建立数据结构等
  - 初始化文件扫描
- GetNext( )：获得下一条Operator的处理结果
  - 如果当前页访问完毕，读下一页
  - 返回下一条记录
- Close( )：结束，释放资源

Sorted Table Scan

- Open( )：初始化，分配资源，建立数据结构等
  - 进行有序文件二分查找，找到起始位置
- GetNext( )：获得下一条Operator的处理结果
  - 如果当前页访问完毕，读下一页
  - 返回下一条满足条件的记录
  - 如果遇到不满足条件的记录，那么完成
- Close( )：结束，释放资源

B^+-Tree Range Scan

- Open( )：初始化，分配资源，建立数据结构等
  - 进行B^+-Tree搜索，获得满足条件的RecordID
  - 对RecordID进行排序
- GetNext( )：获得下一条Operator的处理结果
  - 读取下一个RecordID所对应的记录并返回
- Close( )：结束，释放资源

Hash Table Search

- Open( )：初始化，分配资源，建立数据结构等
  - 进行Hash table搜索，找到满足条件RecordID
  - 对RecordID进行排序
- GetNext( )：获得下一条Operator的处理结果
  - 读取下一个RecordID所对应的记录并返回
- Close( )：结束，释放资源

FilterOperator

- Open( )：初始化，分配资源，建立数据结构等
  - 记录选择条件
  - 记录下层子Operator
- GetNext( )：获得下一条Operator的处理结果
  - 调用子Operator的GetNext( )
  - 进行具体的选择计算
  - 迭代找到下一个满足所有条件的记录返回
- Close( )：结束，释放资源

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作
- 内存数据库

Projection (投影)

$\pi_{Name, GPA}(Student)$

```
select Name, GPA
from Student;
```

提取学生姓名和平均分

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2014	95
…	…	…	…	…	…	
…	…	…	…	…	…	

Projection (投影)

- 投影：提取指定的列
- 行式数据库
  - 在选择的基础上，
  - 对于选中的记录，提取指定的列
  - 生成中间结果记录
- 列式数据库
  - 不同的列存在不同的文件中
  - 投影本身的主要操作是选择不同的文件
  - 选择过滤要在列文件上实现
  - 需要把同一记录的多个列从多个文件中读取、组装在一起
    - 根据具体的实现，关系型的运算有时可以一直保持列的形式
    - 但至少在结果记录生成时需要组装

Projection Operator

- Open( )：初始化，分配资源，建立数据结构等
  - 记录下层子Operator
- GetNext( )：获得下一条Operator的处理结果
  - 调用子Operator的GetNext( )
  - 完成投影操作
- Close( )：结束，释放资源

- 注意：行式
  投影可以在一个选择Operator中实现

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作
- 内存数据库

Join（连接）

$Student \bowtie_{ID = StudentID} TakeCourse$

Student

ID	Name				
131234	张飞				
145678	貂蝉				
129012	孙权				

TakeCourse

Couse ID	Student ID			
7001	131234			
7005	129012			
7012	145678			

```
select Student.Name, TakeCourse.Grade
from Student, TakeCourse
where TakeCourse.StudentID = Student.ID;
```


Equi-Join的实现

- 等值连接是最常见的连接
  - 参见TPCH

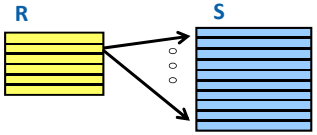
- 三种思路
  - Nested loop，嵌套循环
  - Sorting，排序
  - Hashing，哈希

Nested Loop Join（嵌套循环连接算法）

$R\bowtie_{R.a = S.b}S$

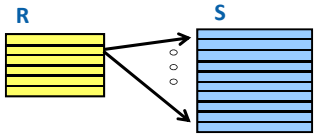
R称为Outer relation
S称为Inner relation

```
foreach tuple r ∈ R {
    foreach tuple s ∈ S {
        if (r.a == s.b) output(r,s);
    }
}
```



Nested Loop Join性能分析

$R\bowtie_{R.a = S.b}S$



R有 M_R个Page
S有 M_S个Page
每个Page有B个记录

- 外循环读R
  - 读了一遍R
- 内循环读S
  - 读了BM_R遍S
- 总共读的page数： $M_R + BM_RM_S$

I/O成本是二次的！

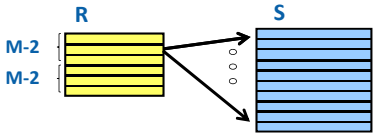
Block Nested Loop Join (块嵌套循环连接算法)

$R\bowtie_{R.a = S.b}S$

- 思想：利用大内存
- 假设内存可以容纳M页
  - 如果$M >= M_R+2$
    - 那么就可以把整个R读入内存
    - 利用1个内存页扫描S
    - 利用1个内存页收取输出结果
- 我们考虑不能把R读入内存的情况

Block Nested Loop Join

$R\bowtie_{R.a = S.b}S$



内存大小为M
外循环每次读入M-2页的R
每次内循环读一遍S

- 外循环读R
  - 读了一遍R，共M_R
- 内循环读S
  - 总共读了$M_R/(M-2)$遍S，共$M_RM_S/(M-2)$
- 总共读的page数： $M_R + M_RM_S/(M-2)$

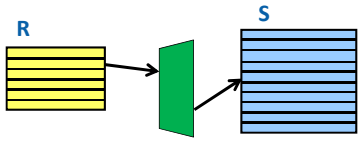
有进步，但仍然是二次的！

Index Nested Loop Join

$R \bowtie_{R.a = S.b} S$

（索引嵌套循环连接算法）

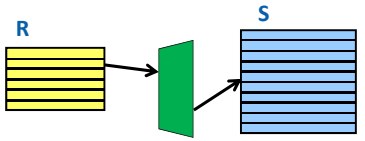
```
foreach tuple r ∈ R {
    lookup index for matching s in S
    if (found) output(r,s);
}
```



经常被使用，尤其是当很少有匹配时，效率很高

Index Nested Loop Join

$R \bowtie_{R.a = S.b} S$



- R有 M_R 个Page
- S有 M_S 个Page
- 每个Page有B个记录
- 索引一次访问需要H次I/O
- 每个R记录有J个匹配记录
- 非聚簇索引

- 外循环读R
  - 读了一遍R，共M_R
- 内循环读索引
  - 总共读了M_RB次索引，共M_RBH
- 内循环访问S
  - 总共有M_RBJ个匹配的S记录，共需M_RBJ
- 总共读的page数： $M_R + M_R BH + M_R BJ$

线性的，但是需要索引，而且后两项可能比较大

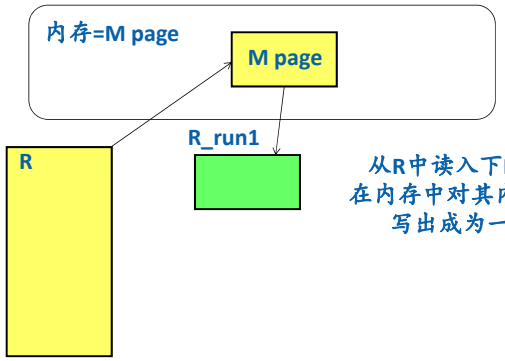
第二种思路：Sorting

$R \bowtie_{R.a = S.b} S$

- 排序
- Sort Merge Join(排序归并连接)思路
  - 把R按照R.a的顺序排序
  - 把S按照S.b的顺序排序
  - Merge(归并)两个有序的序列，找出所有的匹配

Run Generation

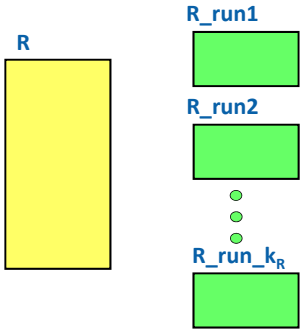
$R \bowtie_{R.a = S.b} S$



从R中读入下M个页，在内存中对其内存排序，写出成为一个Run

Run Generation

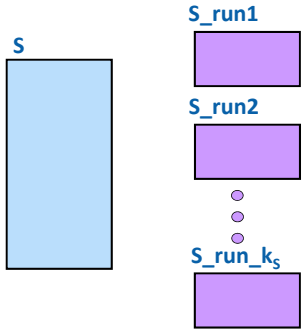
$R \bowtie_{R.a = S.b} S$



$k_R = M_R / M$

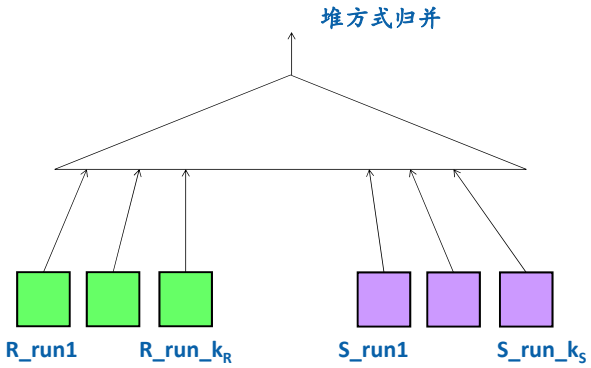
Run Generation

$R \bowtie_{R.a = S.b} S$



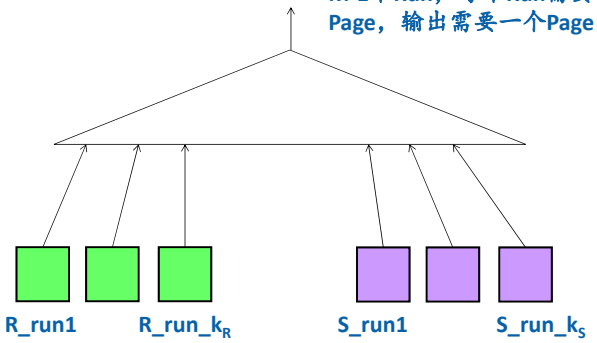
$k_S = M_S / M$

Merge



Merge

内存M page，最多同时归并M-1个Run，每个Run需要一个Page，输出需要一个Page



需要多少层归并？

- 共有$\frac{M_R}{M}+\frac{M_S}{M}$个Run
- 所以需要$\log_{M-1}(\frac{M_R}{M}+\frac{M_S}{M})$层才能完成全部归并

- 另一个角度：
  - 如果希望只使用一次归并
  - 那么：$M_R+M_S<=M(M-1)\approx M^2$

Sort Merge Join代价

R有 M_R个Page
S有 M_S个Page
每个Page有B个记录

- Run generation
  - R：读一遍R，写一遍R的run，共$2M_R$
  - S：读一遍S，写一遍S的run，共$2M_S$
- Merge
  - 假设一遍归并：读R的run和S的run，共M_R+M_S
- 总共： $3(M_R+M_S)$

I/O成本
是线性的！

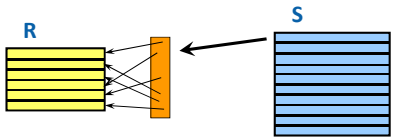
第三种思路：Hashing

- 等值连接
  - 找相同的值
  - 所以可以利用哈希来解决问题
- 思路
  - 哈希后匹配的记录在同一个桶里

Hash Join

$R\bowtie_{R.a = S.b}S$

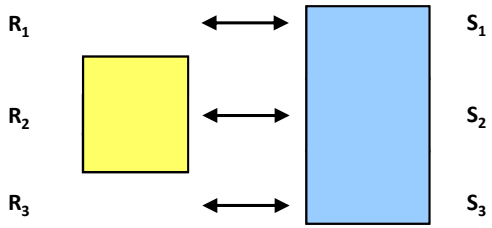
- Simple hash join



build： 读R建立hash table;
probe：读S访问hash table找到所有的匹配;

☞R比内存大怎么办？

I/O Partitioning



- 思路：把R和S划分成小块
  - PartitionID = hash(join key) % PartitionNumber
- R_j中记录的匹配只存在于相应的S_j中
  - 为什么？ 匹配的记录hash (join key)必然相同

GRACE Hash Join

$R \bowtie_{R.a = S.b} S$

- 对R进行I/O partitioning
- 对S进行I/O partitioning
- for (j=0; j< ParitionNumber; j++) {
  simple hash join计算$R_j \bowtie S_j$;
  }

GRACE Hash Join性能分析

内存有M个Page
R有 M_R个Page
S有 M_S个Page
每个Page有B个记录

- 对R进行I/O partitioning
  - 读M_R个Page，写M_R个Page
- 对S进行I/O partitioning
  - 读M_S个Page，写M_S个Page
- Simple hash join计算所有的$R_j \bowtie S_j$
  - 读$M_R + M_S$个Page
- 总代价（不考虑输出）
  - $3(M_R + M_S)$

I/O成本是线性的！

Partition Number?

- 内存有M个Page
- R有 M_R个Page
- S有 M_S个Page
- 每个Page有B个记录
- 每个记录在哈希表中需要f大小的空间（换算成Page为单位）

- Simple Hash Join阶段的要求
  - R_j在内存中
  - Hash table在内存中
  - S_j需要1页
- 设Partition Number = P，
  那么要求$(M_R/P + f\,BM_R/P + 1) <= M$
  - R_j大小为M_R/P
  - Hash table大小为$f\,BM_R/P$
  - S_j需要为1
- $P >= (M_R + f\,BM_R)/(M-1)$

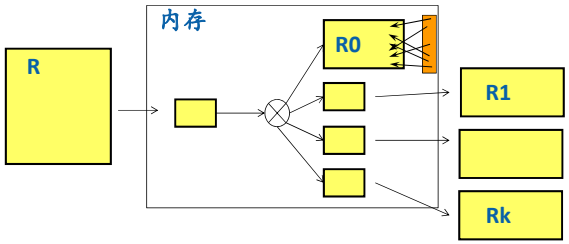
I/O Partitioning阶段需要的内存

- 1个输入页，P个输出页，所以至少需要
  - $\frac{(1+fB)M_R}{M-1}+1$页
- 一遍I/O partitioning可以处理，要求上述小于总内存
  - $\frac{(1+fB)M_R}{M-1}+1 \le M$
  - 所以，$M \ge \sqrt{(1+fB)M_R}+1$
- 比较Sort Merge Join一遍的要求$M_R+M_S<=M(M-1)$
  - Hash Join可以支持很大的M_S

利用额外的内存提高性能

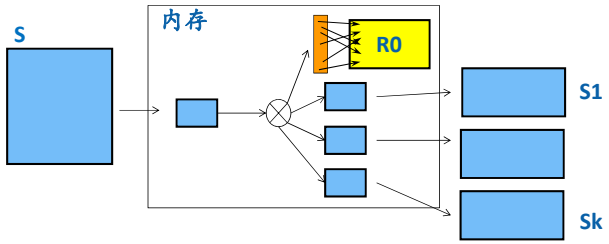
- 假设$M \gg \sqrt{(1+fB)M_R}+1$
  - 也就是，在I/O Partitioning阶段，P+1<<M
  - 有大量的内存没有被使用
- 如何利用这些额外的内存呢？
  - 把I/O partitioning和Join部分混合

利用额外的内存：混合哈希连接



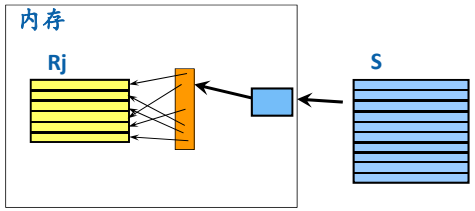
- 对R进行I/O Partitioning
  - 把一个桶的内容保持在内存中
  - 并在其上建立内存哈希表
  - 其它桶仍然输出

利用额外的内存：混合哈希连接



- 对S进行I/O Partitioning
  - 把S0桶的内容直接访问内存的哈希表，完成Join匹配
  - 其它桶仍然输出

利用额外的内存：混合哈希连接



- Join阶段，对于R1, R2, …
  - 与GRACE相同

Equi-Join算法比较

- Nested loop join
  - Block nested loop join
  - Index nested loop join：当有索引且预期匹配极少时
- Sort merge join
  - 代价比Hash Join要高：内存排序是O(NlogN)
  - 内存占用可能比Hash Join要大
  - 当其中一个表已经有序时，很常用
- Hash join
  - 在没有索引的情况下，很常用

一般连接条件

$R \bowtie_{连接条件} S$

- 多个条件，包含一个等值连接条件
  - 等值连接 + 其他条件过滤
- 非等值连接
  - 最通用的算法是(Block) nested loop
    - 可以考虑了任意两个记录的组合，求解任意连接条件
  - Sort merge join：可支持非等值比较连接
    - 把两个输入表分别排好序
    - 在排序的序列上，很容易求解R.a > S.b形式的条件
      - 对于每个R.a，在S上就是一个范围查找
  - Hash join：完全不可能支持
    - Hash table只支持点查询！

Join Operator

- Open( )：初始化，分配资源，建立数据结构等
  - 记录两个子Operator
  - 完成准备工作
    - Grace: 包括I/O partitioning+设置第一个Simple Hash Join,
    - Sort Merge Join: Run generation+初始化Run Merge等
- GetNext( )：获得下一条Operator的处理结果
  - 进行simple hash join的一步，或者Run merge的一步
  - 对于Hash join，一对Rj和Sj处理完后，初始化下一对的处理
- Close( )：结束，释放资源

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作
- 内存数据库

去重δ

- SQL的Distinct
- 如果A是一个包，那么δ(A)就是一个集合
- 例如
  - A={1,1,2,2,2,3}
  - δ(A)={1,2,3}

如何实现？

- 思路
  - 排序Sorting
  - 哈希Hashing

基于排序的去重

- 情况1：当输入可以放入内存时
  - 对去重运算的输入记录进行内存排序
  - 顺序扫描排序结果，重复记录必然相邻，只输出非重复记录
- 情况2：当输入比内存大时
  - 进行外存排序
  - Run generation：这时可以进行Run内部去重
  - Run merge: 在merge的过程中，就可以完成去重

基于哈希的去重

- 情况1：当输入可以放入内存时
  - 对输入记录建立哈希表，同时去重
    - 哈希表中每个值只存一次
    - 对于新插入的值，检查是否已经出现，如果没有出现，才插入；如果已经出现，那么丢弃
  - 最后，对哈希表进行输出
- 情况2：当输入比内存大时
  - I/O partitioning
  - 然后对每个partition，分别进行去重运算

Distinct Operator

- Open( )：初始化，分配资源，建立数据结构等
  - 记录孩子Operator
  - 完成准备工作
    - 内存排序，或 Run generation+初始化Run Merge等
    - 内存哈希，或I/O partitioning及初始化读第一个partition建哈希表
- GetNext( )：获得下一条Operator的处理结果
  - 扫描排序结果，或者Run merge的一步
  - 输出哈希表内容
- Close( )：结束，释放资源

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作
- 内存数据库

Group by：分组，然后统计

Student

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2013	95

统计各系的学生人数

select *Major*, count(*) as *Number*
from *Student*
group by *Major*;

输出结果

Major	Number
法律	1
经管	1
计算机	3

Aggregation的实现

- 每种Aggregation都可以拆分为下述的步骤
  - ❑init
    - 对中间结果进行初始化
  - ❑accumulate(x)
    - 把x累计到中间结果上
  - ❑merge(y)
    - y是另一个中间结果
  - ❑finalize
    - 计算最终结果

举例

	中间结果	init	accumulate(x)	finalize
SUM	部分和s	s=0	s += x	return s
COUNT	计数c	c=0	c++	return c
AVG	s, c	s=0; c=0;	s+=x; c++;	return s/c
MIN	当前最小m	m=+∞	m=min(m, x)	return m
MAX	当前最大m	m=-∞	m=max(m, x)	return m

Group by + Aggregation

- 中间结果记录包含
  - ❑group by分组的列
  - ❑每个Aggregation的中间结果部分

基于排序的分组聚集

- 情况1：当输入可以放入内存时
  - ❑对输入记录进行内存排序，排序的键是分组的列
  - ❑顺序扫描排序结果，同一组的记录必然相邻，对同一组的记录计算Aggregation，输出分组的列和Aggregation的结果

基于排序的分组聚集

- 情况2：当输入大于内存时
  - 类似外存排序
  - Run generation
    - 每个run，进行内存排序，然后计算输出分组聚集的中间结果
  - Run merge
    - 同一组的记录累计中间结果，调用merge()
    - 当出现不同组的记录时，说明当前组结束，调用finalize，输出最后结果
  - 提前计算有助于减少Run的大小，提高效率

基于哈希的分组聚集

- 情况1：当所有的组的中间结果可以放入内存时
  - 建立内存哈希表，key=group by key
  - 哈希表中存储分组聚集的中间结果
  - 每次哈希一个记录
    - 如果组不存在，那么就向哈希表中插入一条新记录，包含组和初始化的聚集的中间结果
    - 如果组存在，那么就累计聚集结果
  - 最后扫描哈希表输出每个组和聚集的最终结果
- 情况2：当所有的组的中间结果比内存大时
  - I/O partitioning，在组上hash得到partition
  - 然后对每个partition，分别进行分组聚集运算

Group by + Aggregation

- Open( )：初始化，分配资源，建立数据结构等
  - ?
- GetNext( )：获得下一条Operator的处理结果
  - ?
- Close( )：结束，释放资源
  - ?

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作
- 内存数据库

集合操作

- 并
- 交
- 差
- 笛卡尔积

SQL中的集合操作

- 并：**保留字Union**
  - $A \cup B$
- 交：**保留字Intersect**
  - $A \cap B$
- 差：**保留字Except**
  - $A - B$

```
(select ...
 from ...
 where ...)
```

集合操作保留字

```
(select ...
 from ...
 where ...)
```

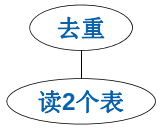

笛卡尔积

```
select ...
from R, S;
```

没有连接条件，在from语句中写多于1个表，
就是笛卡尔积

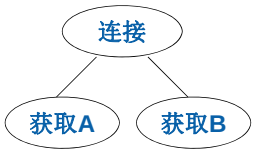
并

- $A \cup B$
- 要求：A的Schema与B的Schema完全一致
  - 列数相同
  - 对应的每个列的类型相同
  - 对应的每个列的名字也相同
- 并所需要的操作
  - 把两个表的记录放在一起
  - 去重
- 所以，并的实现
  - 把两个表作为去重操作的输入
  - 那么去重操作的输出就是并的结果



交

- $A \cap B$
- 对于A和B也要求具有完全一样的Schema
  - 与并对输入的要求一致
- 交是一种特殊的连接
  - 多属性等值连接
  - 所有的相应属性都相同
- 所以，交的实现
  - 用一个属性为基础进行等值连接
  - 对于匹配检查是否其它属性也相同
  - 去重



差

- $A-B$
- 对于A和B也要求具有完全一样的Schema
  - 与并对输入的要求一致
- 差可以修改连接操作来实现
  - 例如，采用sort merge join
  - 在最后的归并步骤，检查并输出A-B的结果，即在A中出现但是在B中没有匹配的记录

笛卡尔积

- Block nested loop即可
- 代价很大

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作
- 内存数据库

内存处理

- 随着内存容量的指数级增加
  - 越来越大的数据集可以完全存放在内存中
  - 或者完全存放在一个机群的总和的内存中
  - 例如，每台服务器64GB，一组刀片16台，就是1TB
  - 1TB对于很多重要的热点的数据可能已经足够了
- 内存处理的优点
  - 去除了硬盘读写的开销
  - 于是提高了处理速度

关系型内存数据库

- Main memory database system
- Memory-resident database system
- 上述两个名词有区别
  - Memory-resident：可能是在buffer pool中
  - MMDB: 可能彻底不用buffer pool，改变了系统内部设计

关系型内存数据库

- 最早的提法出现在1980末，1990初
- 第一代MMDB出现于1990初
  - 没有高速缓存的概念
  - 例如：TimesTen
- 第二代MMDB出现于1990末，2000初
  - 对于新的硬件进行优化
  - 主要是学术领域提出的
  - 例如：MonetDB
  - 这一时期，产业界也开始重视MMDB，但主要是用来作前端的关系型cache
- 近年来，主流数据库公司纷纷投入研发MMDB
  - IBM Blink, Microsoft Hekaton & Apollo, SAP HANA, IBM BLU, 等

内存数据库系统

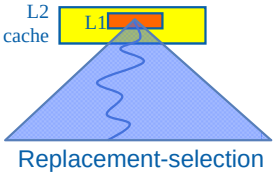
- 商业产品
  - SAP HANA
  - Microsoft
    - Hekaton：OLTP
    - Apollo: OLAP
  - IBM
    - Blink：IBM的数据仓库加速系统
    - BLU：列式数据库engine, 有硬盘存储，使用了许多内存处理技术
  - Oracle
    - In-memory data analytics caching
- 研究系统
  - MonetDB：荷兰CWI
  - HyperDB：德国TUM

主要挑战：内存墙问题

- 内存访问需要100~1000 cycles
- 思路1：减少cache miss
  - 调整数据结构或算法
- 思路2：降低cache miss对性能的影响
  - Software prefetch 预取指令
  - 并行的K个内存访问时，总时间<< k*单个内存访问时间

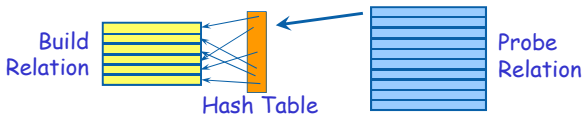
Sorting

- In-memory sorting / generating runs
- AlphaSort [NBC94]



- 使用 quick sort 而不是replacement selection
  - 顺序访问 vs. 随机访问
  - 当快排缩小到cache size以下时，就没有cache miss

Hash Join



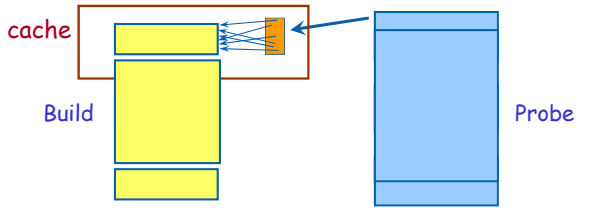
- 随机访问hash table
- 非常差的cache performance
  - ≥ 73% of user time is CPU cache stalls [CAG04]

☞ 解决方案

- Cache partitioning
- Prefetching

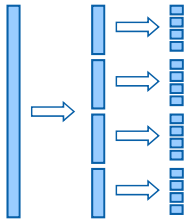
Cache Partitioning [SKN94]

- 类似于I/O partitioning
  - 划分为cache大小的 partitions
  - 从而避免了很多cache miss



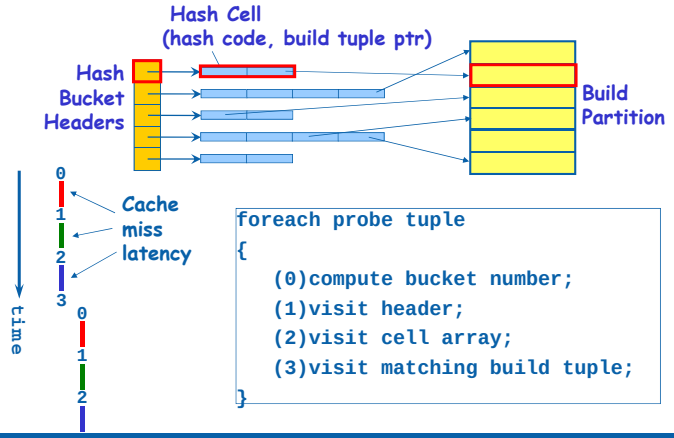
Radix Join: Reducing Partition Cost

- TLB thrashing if # partitions > # TLB entries
  - Cache thrashing if # partitions > # cache lines in cache
- 解决方案: multiple passes
  - # partitions per pass < TLB size
  - Radix-cluster [BMK99,MBK00]



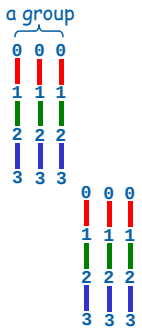
2-pass partition

Simplified Probing Algorithm




```
foreach probe tuple
{
   (0)compute bucket number;
   (1)visit header;
   (2)visit cell array;
   (3)visit matching build tuple;
}
```


Group Prefetching [CGM04]




```
foreach group of probe tuples {
   foreach tuple in group {
     (0)compute bucket number;
        prefetch header;
   }
   foreach tuple in group {
     (1)visit header;
        prefetch cell array;
   }
   foreach tuple in group {
     (2)visit cell array;
        prefetch build tuple;
   }
   foreach tuple in group {
     (3)visit matching build tuple;
   }
}
```


小结

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作
- 内存数据库