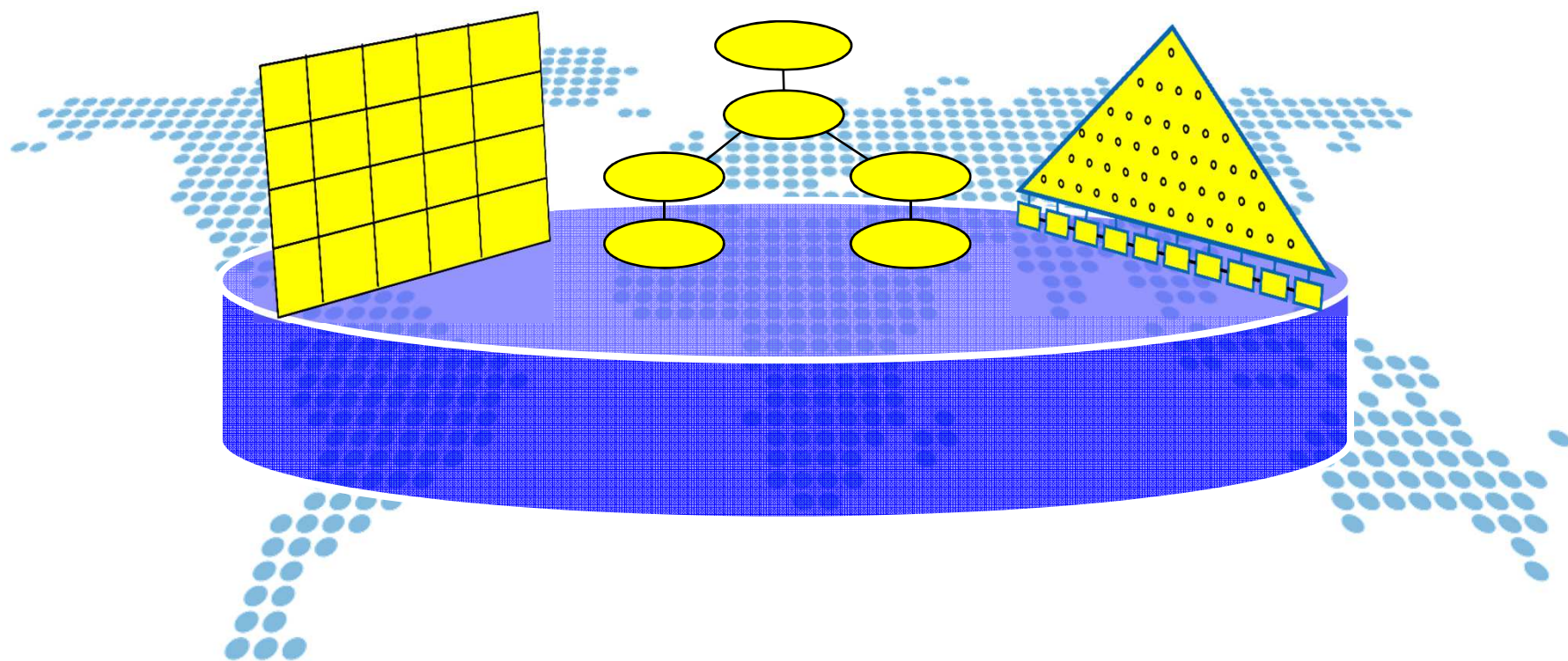


数据库系统

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

Selection (选择)

$\sigma_{\text{Major}=\text{'计算机'}}$ (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
...
...
...

情况1：无索引、未排序的数据

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

- 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：依次访问R的各个数据页，对于每页依次访问各个记录，对于每个记录求解选择条件
- I/O代价
 - 如果R的数据页共有 M_R 个，那么代价为 M_R

情况2：无索引、排序的数据

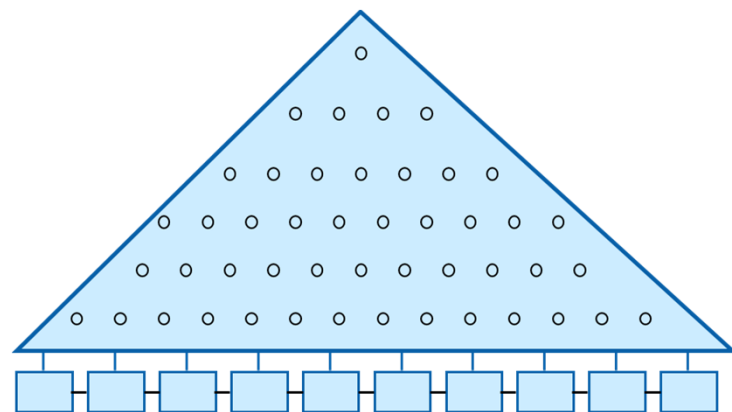
$\sigma_{R.attr \text{ op } value}(R)$

- R.attr上没有索引
- 但是关系表的数据对于R.attr是有序的
- 访问路径 (Access Path)
 - 使用二分查找定位满足查询条件的第一个记录
 - 从这个元组开始，扫描关系R，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
 - 如果R的数据页共有 M_R 个，那么二分查找 $\log_2 M_R$
 - 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

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

- R.attr上有B⁺-Tree索引



- 可选的访问路径（Access Path）

- B⁺-Tree索引：进行范围查询
- 聚簇索引：叶子结点就是数据页，包含记录
- 非聚簇索引：叶子结点包含RecordID，需要随机访问读取记录

- 非聚簇索引代价

- 假设B⁺-Tree的查找代价为H次(例如：2~3次)I/O
- 假设符合条件的记录有m个，占据L个叶子结点页
- 那么总代价：H+L+m

非聚簇索引范围查找的优化

- 当返回的RecordID非常多

- 对所有RecordID进行排序
- 按照RecordID的排序的顺序访问数据页

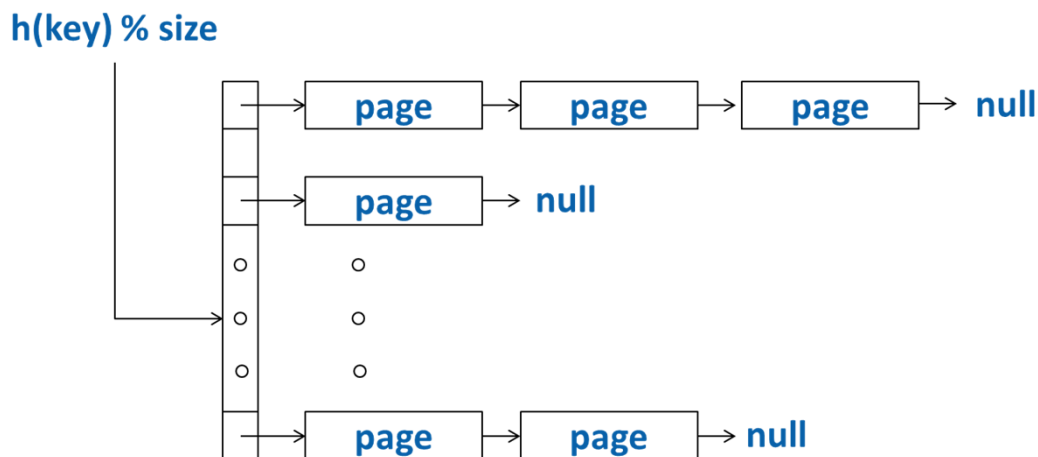
- 优势：

- 如果同一个数据页中有多个记录符合条件，只需要读一次
- 虽然不是顺序访问，但是可以使磁头只向一个方向移动，而不是来回移动，从而一定程度上节省seek时间
- 如果有多个操作共享硬盘，那么这种节省效果就比较有限

情况4：哈希索引、等值比较

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

- R.attr上有哈希索引
- 比较操作是等值比较



- 可选的访问路径（Access Path）
 - 哈希索引
 - 使用哈希索引找到满足查询条件的RecordID
 - 需要随机访问读取记录
- 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
 - 那么总代价：H+m

一般的选择条件

- 前述是单个比较的选择条件
- 一般地，选择条件可能包括
 - 多个比较
 - AND, OR, NOT等连接在一起
- 大部分系统对没有OR, NOT的条件可以比较有效地求解
- 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：先对于一个合取条件选择

- 选择的形式写成合取范式
 - 最外层是AND
 - 内层可以有OR
 - 例如： $(a > 0) \text{ AND } (b < 0 \text{ OR } C > 100)$
- 如果存在合取项中不包含任何OR和NOT，那么就可以先用这个合取项进行选择
 - 利用前述对单个条件的选择方法
- 在得到的结果记录上进行其它条件的求解

方法3：使用位图索引

- 对于一个选择条件，如果各个列都有位图索引
- 那么可以分别获得位图
- 计算选择条件的总位图
 - 把AND转化为按位与，OR转化为按位或

方法4：利用多个索引

- 与方法3类似，如果一个选择的所有属性和单个条件都可以通过索引求解
- 那么可以得到每个子条件的RecordID集合
- 那么可以把选择的求解转化为RecordID集合操作
 - AND：集合交集
 - OR：集合并集
 - NOT：全集 - 集合

迭代求解实现Selection Operator

- **Open()**: 初始化, 分配资源, 建立数据结构等
 - 初始化文件扫描
 - 或者进行有序文件二分查找, 找到起始位置
 - 或者进行B⁺-Tree搜索, 获得满足条件的起始RecordID
 - 或者进行Hash table搜索, 找到第一个满足条件RecordID
- **GetNext()**: 获得下一条Operator的处理结果
 - 调用子Operator的GetNext()
 - 对于文件扫描, 需要进行具体的选择计算
 - 对于其它方式, 如果还有额外的条件, 对额外条件进行计算
 - 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
 - 调用子Operator的close()

Outline


- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作

Projection (投影)

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

```
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，
- 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

迭代求解实现 Projection Operator

- Open(): 初始化, 分配资源, 建立数据结构等
 - 初始化孩子
- 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
 - 调用子Operator的close()
- 选择和投影可以在一个Operator中实现

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作

Join (连接)

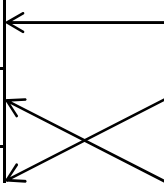
Student ⋈_{ID = StudentID} TakeCourse

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;
```

考虑：先求笛卡尔积，再选择投影？

- 笛卡尔积会产生大量的中间结果
- 但是，连接本身的结果可能只占一小部分
- 所以，这种方法效率比较低

Equi-Join的实现

- 三种思路

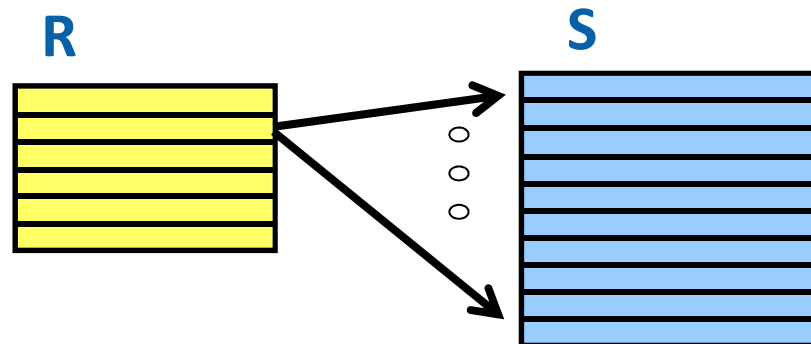
- Nested loop, 嵌套循环
- Sorting, 排序
- Hashing, 哈希

Nested Loop Join (嵌套循环连接算法)

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

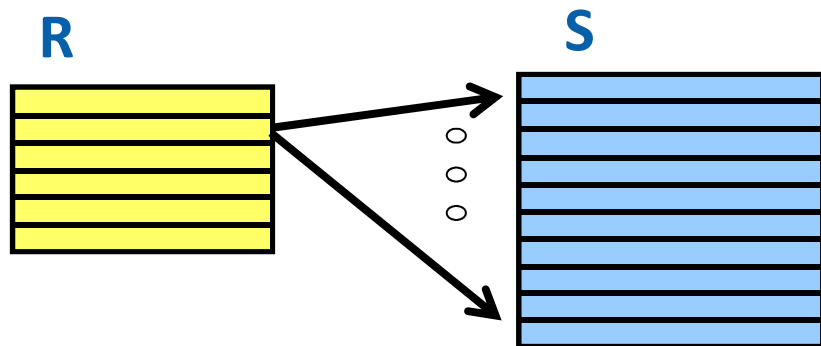
```
foreach tuple  $r \in R$  {  
    foreach tuple  $s \in S$  {  
        if ( $r.a == s.b$ ) output( $r,s$ );  
    }  
}
```

R称为Outer relation
S称为Inner relation



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
 - 对于R的每一个记录读所有的S
 - 总共读了 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

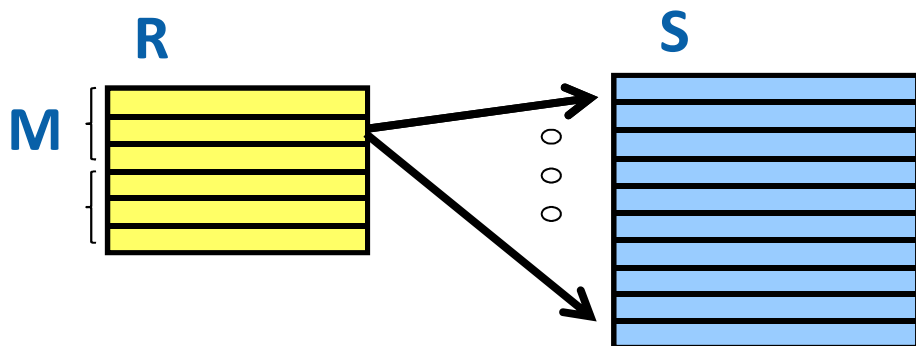
$$R \bowtie_{R.a = S.b} S$$

(块嵌套循环连接算法)

- 前面简单的嵌套循环算法没有重复利用可用的内存
- 假设内存可以容纳M页
- 如果 $M \geq M_R + 2$
 - 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
- 如果不成立，那么可以每次读入M-2页
 - S的扫描次数可以大大降低

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 $M_R M_S/(M-2)$

- 总共读的page数： $M_R + M_R M_S/(M-2)$

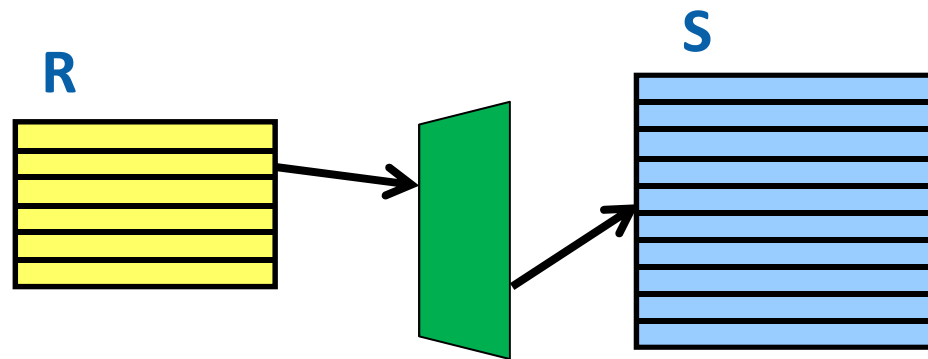
有进步，
但仍然是
二次的！

Index Nested Loop Join

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

(索引嵌套循环连接算法)

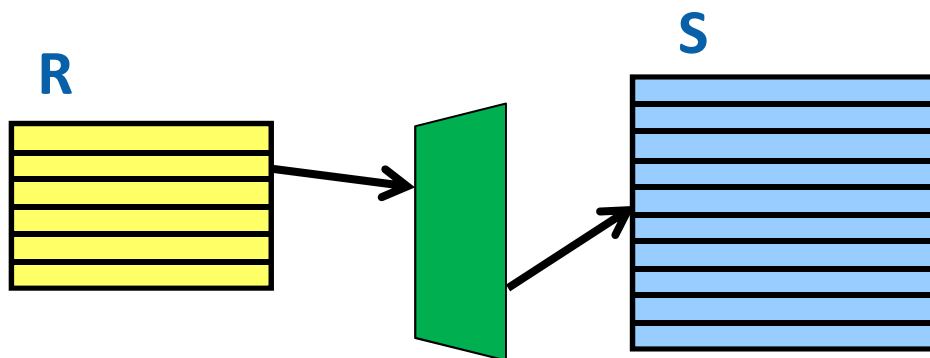
```
foreach tuple  $r \in R$  {  
    lookup index for matching  $s$  in  $S$   
    if (found) output( $r, s$ );  
}
```



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 $M_R B$ 次索引，共 $M_R B H$

- 内循环访问S

- 总共有 $M_R B J$ 个匹配的S记录，共需 $M_R B J$

- 总共读的page数: $M_R + M_R B H + M_R B J$

线性的，但是需要索引，而且后两项可能比较大

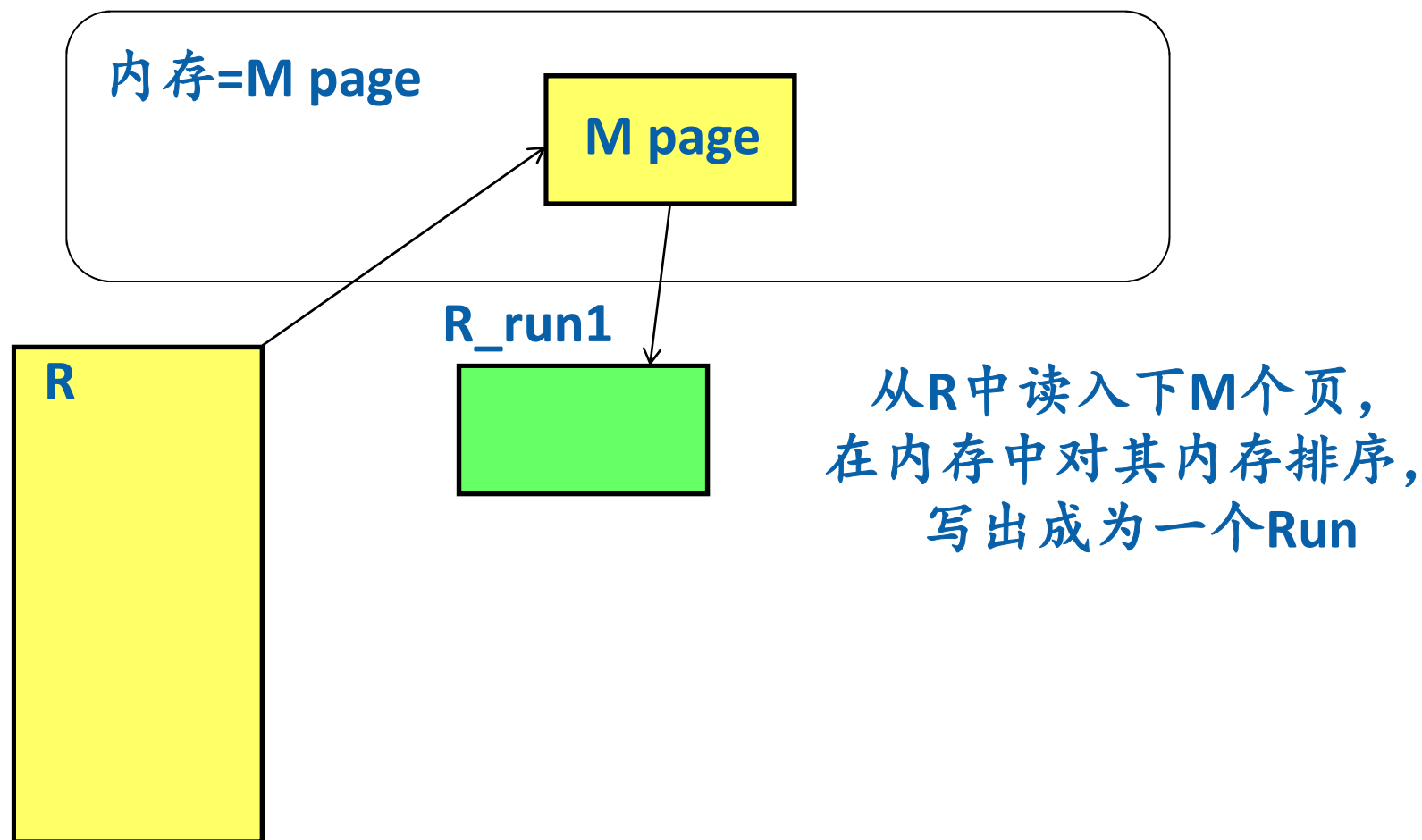
第二种思路：Sorting

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

- 排序
- Sort Merge Join(排序归并连接)思路
 - 如果把R按照R.a的顺序排序
 - 如果把S按照S.b的顺序排序
 - 那么可以Merge(归并)找出所有的匹配

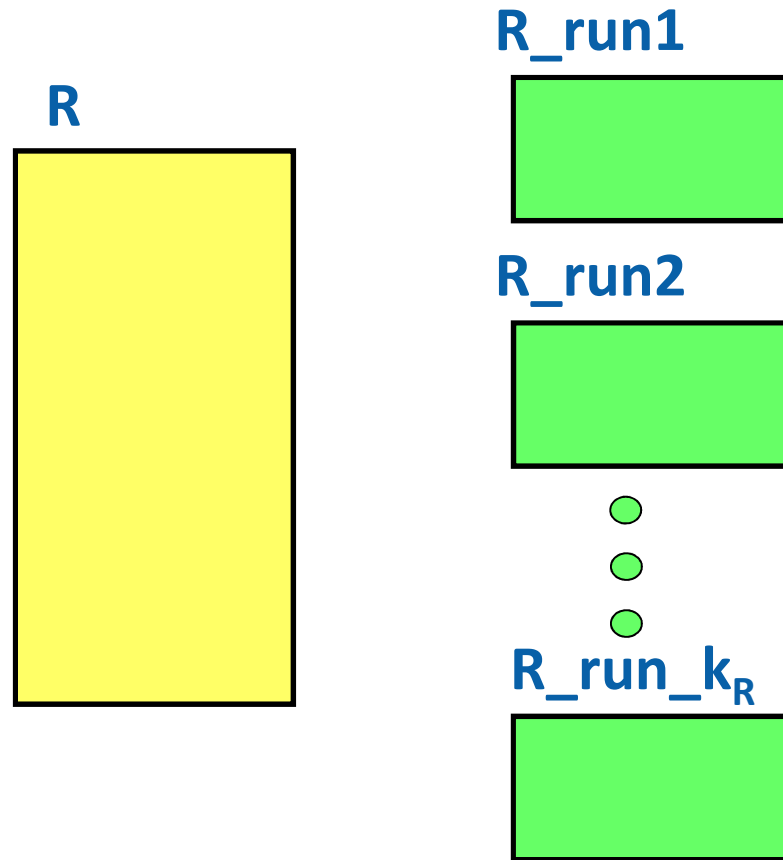
Run Generation

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



Run Generation

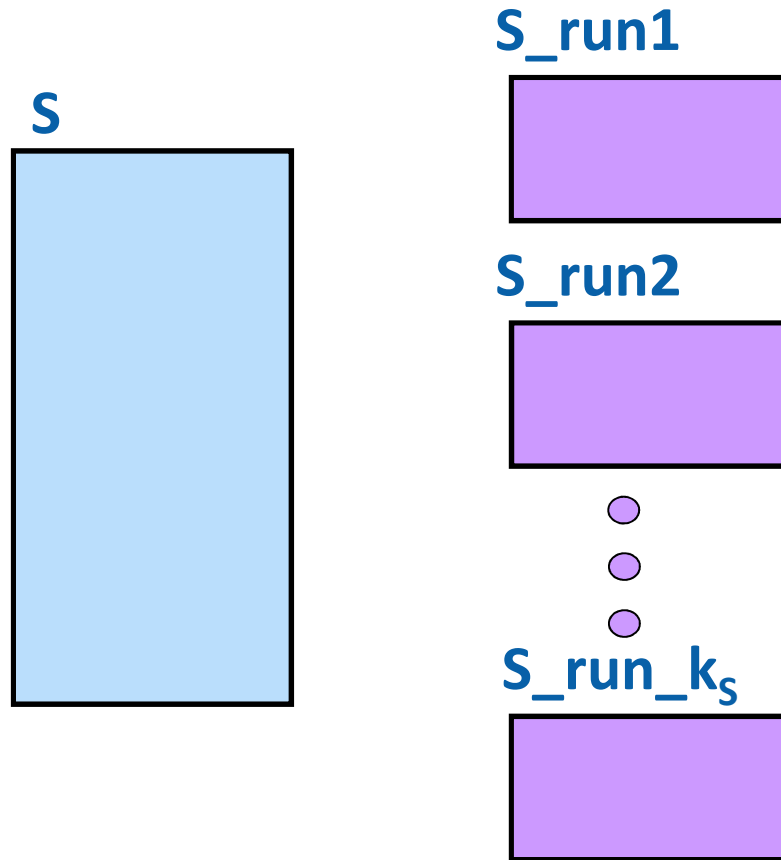
$$R \bowtie_{R.a = S.b} S$$



$$k_R = M_R / M$$

Run Generation

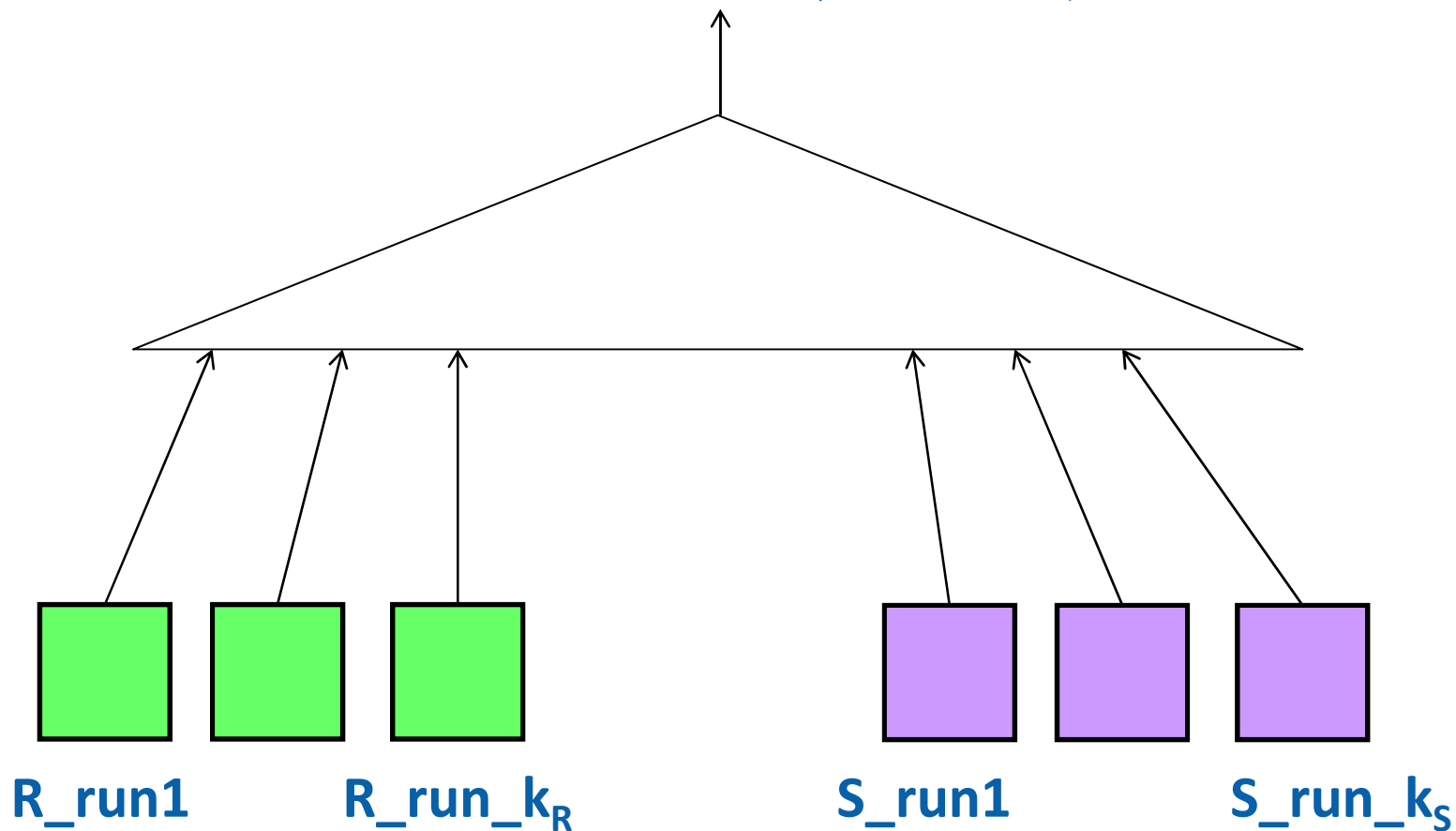
$$R \bowtie_{R.a = S.b} S$$



$$k_S = M_S / M$$

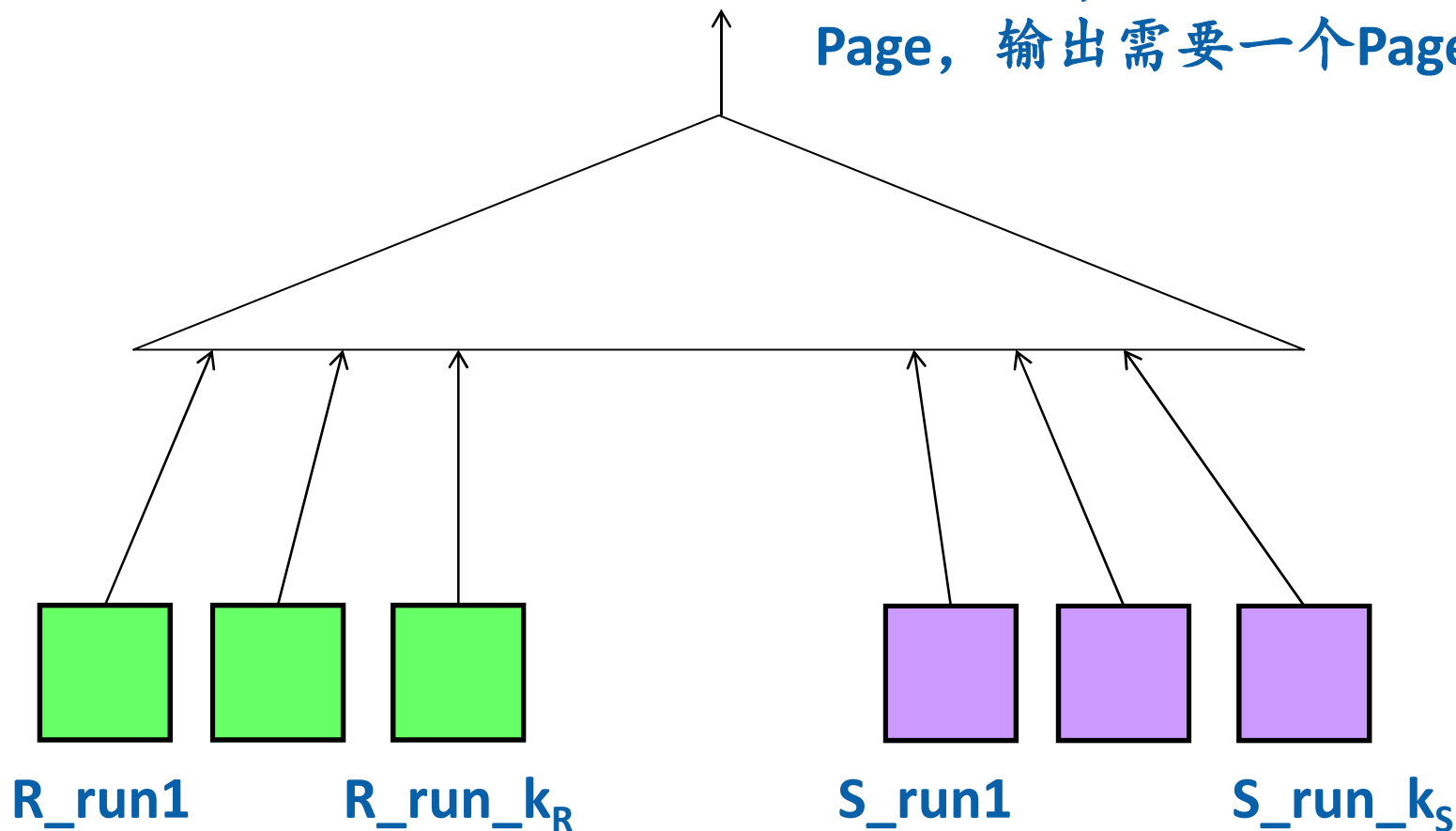
Merge

堆方式归并



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： $M_R + M_S \leq M(M-1) \approx M^2$

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

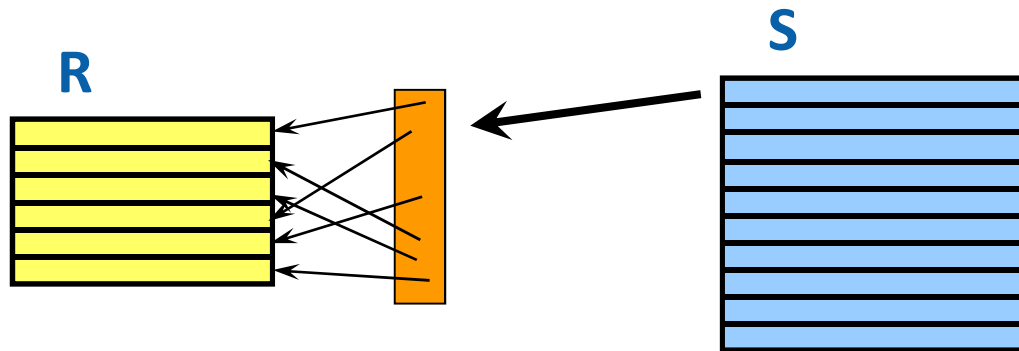
- 思路

- 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

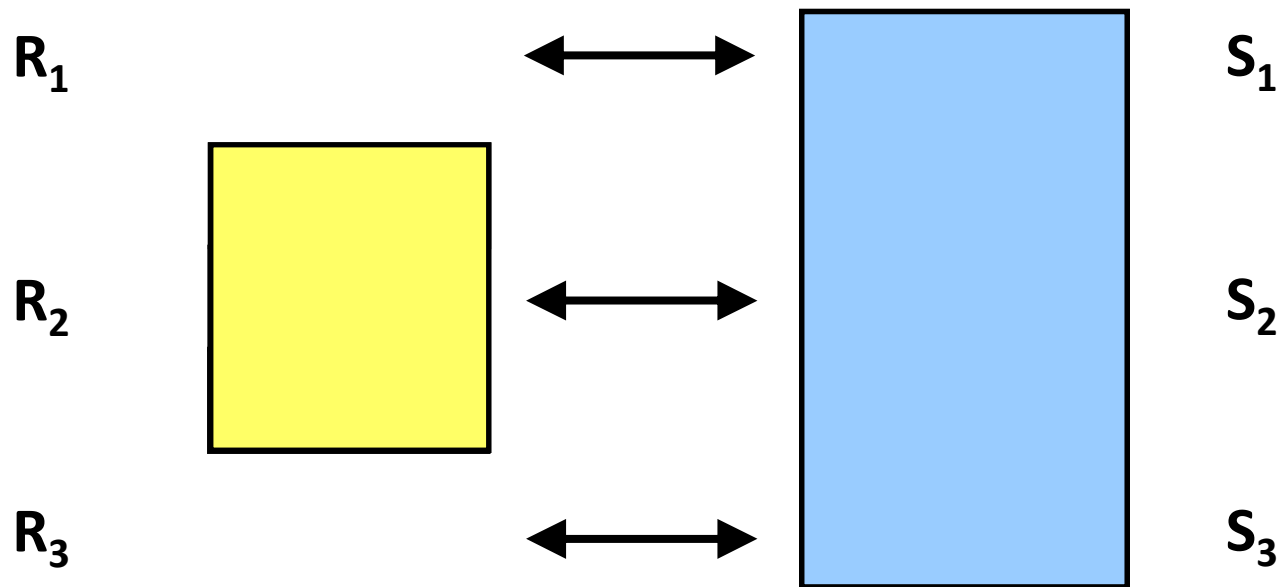


build: 读R建立hash table;

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
 - $\text{PartitionID} = \text{hash}(\text{join key}) \% \text{PartitionNumber}$
- 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 $\text{hash}(\text{join key})$ 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 ($j=0$; $j < \text{PartitionNumber}$; $j++$) {
 simple hash join 计算 $R_j \bowtie S_j$;
}

GRACE Hash Join性能分析

- 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)
 - $3(M_R + 3M_S)$

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

**I/O成本
是线性的!**

Partition Number?

- 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

- R_j 大小为 M_R / P
- Hash table 大小为 $f B M_R / P$
- S_j 需要为1

- 所以要求 $(M_R / P + f B M_R / P + 1) \leq M$

- $P \geq (M_R + f B M_R) / (M - 1)$

- 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)

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

$$\square \frac{(1+fB)M_R}{M-1} + 1 \text{ 页}$$

- 一遍I/O partitioning可以处理，要求上述小于总内存

$$\square \frac{(1+fB)M_R}{M-1} + 1 \leq M$$

$$\square \text{ 所以, } (1+fB)M_R \leq (M-1)^2$$

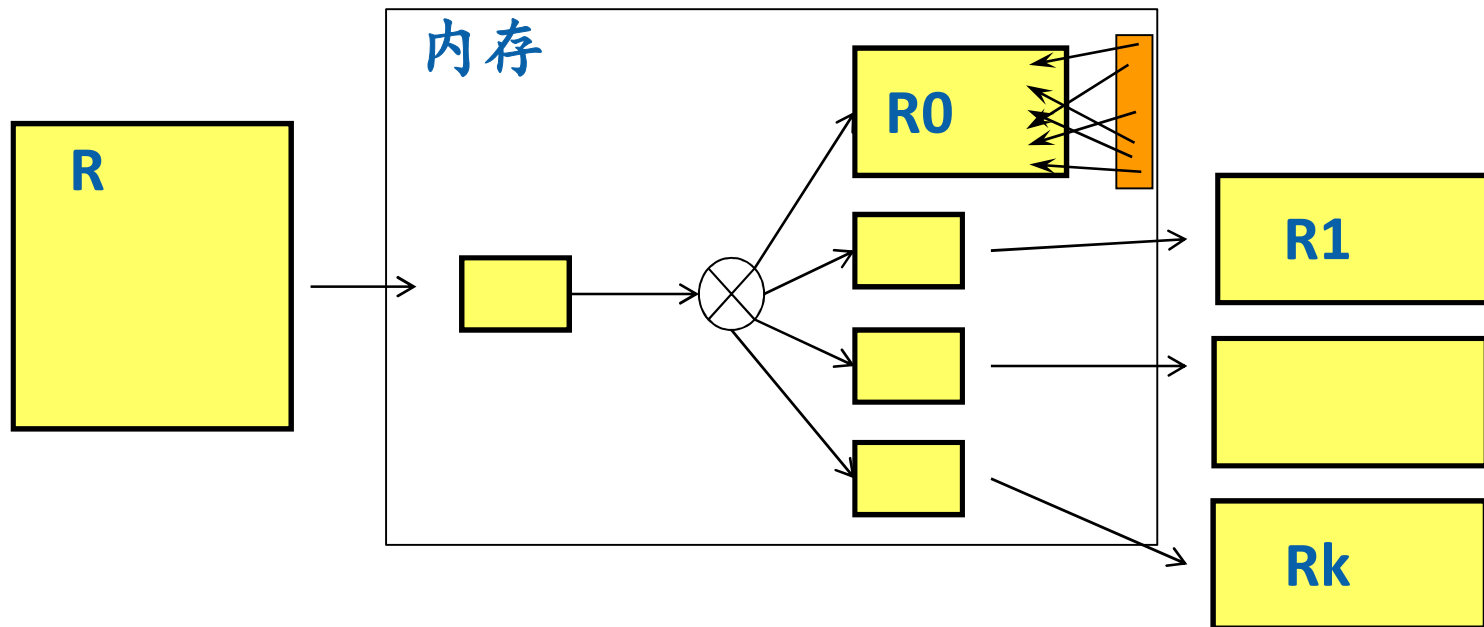
$$\square \text{ 所以, } M \geq \sqrt{(1+fB)M_R} + 1$$

- \square 比较Sort Merge Join一遍的要求 $M_R + M_S \leq M(M-1)$
 - Hash Join可以支持很大的 M_S

额外的内存

- 假设 $M \gg \sqrt{(1 + fB)M_R} + 1$
 - 也就是，在I/O Partitioning阶段， $P+1 \ll M$
 - 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

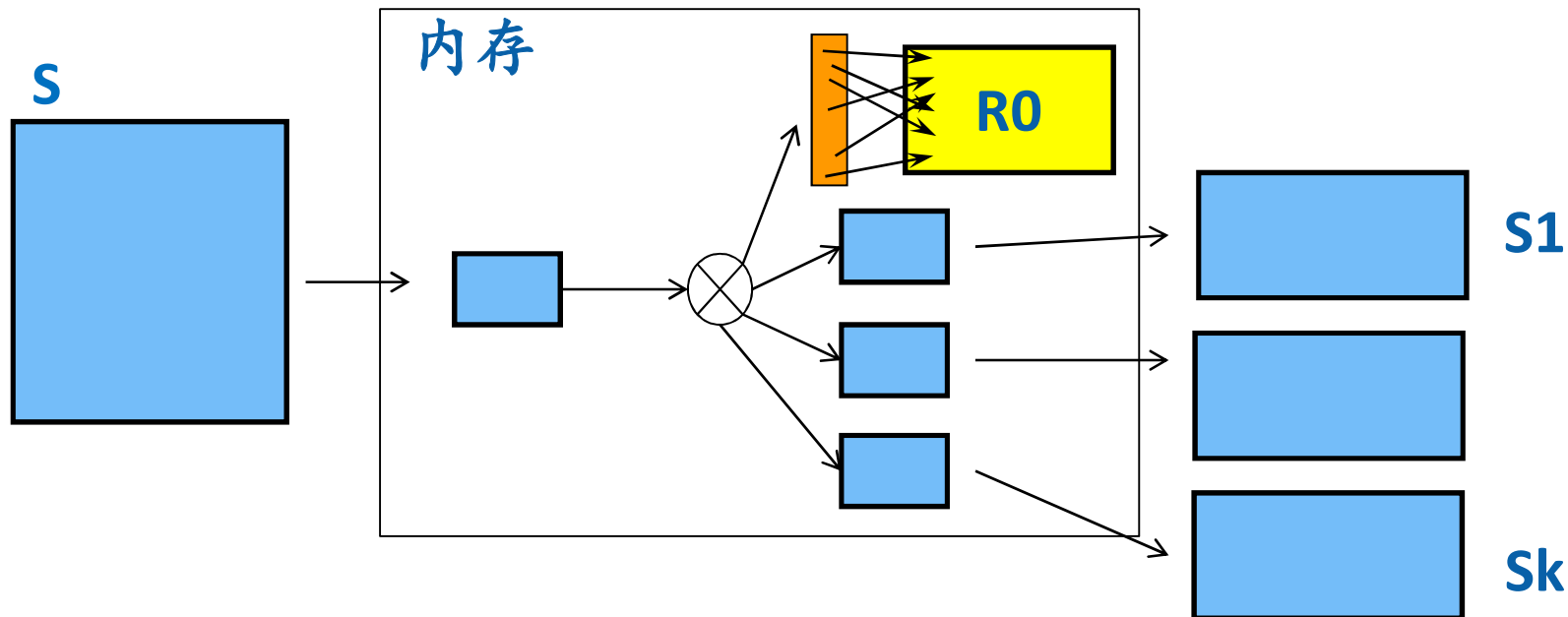
利用额外的内存：混合哈希连接



• 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

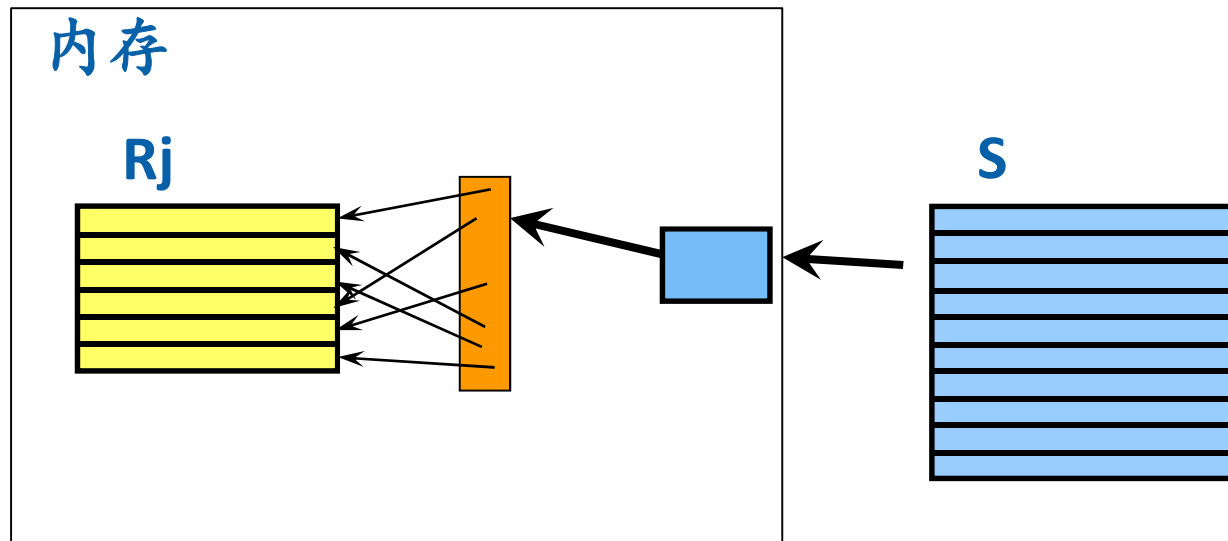
利用额外的内存：混合哈希连接



- 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

- 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 $O(N\log N)$
- ☐ 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

- 多个属性的等值连接
- 非等值连接

多属性连接

- 如果其中包含一个等值连接条件
- 那么可以先按照这个等值连接条件计算等值连接
- 在结果上再进行选择过滤操作

非等值连接操作

- 对于非等值连接，最通用的算法是
 - (Block) nested loop
 - 实际上考虑了任何两个记录的组合
 - 所以最通用
- Sort merge join可以修改来支持非等值连接
 - 把两个输入表分别排好序
 - 在排序的序列上，很容易求解 $R.a > S.b$ 的记录
 - 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！

迭代求解实现 Join Operator

- **Open()**: 初始化, 分配资源, 建立数据结构等
 - 初始化孩子
 - 完成准备工作: 可能包括I/O partitioning+设置第一个Simple Hash Join, Run generation+初始化Run Merge等
- **GetNext()**: 获得下一条Operator的处理结果
 - 调用子Operator的GetNext()
 - 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
 - 调用子Operator的close()

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作

去重 δ

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

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

基于排序的去重

- 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

- 对去重运算的输入记录建立哈希表
- 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
- 这样在哈希的过程中，就完成了去重
- 最后，对哈希表进行输出

基于哈希的去重

- 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

迭代求解实现 Distinct Operator

- **Open()**: 初始化, 分配资源, 建立数据结构等
 - 初始化孩子
 - 完成准备工作: 内存排序, 或 Run generation+初始化Run Merge等
- **GetNext()**: 获得下一条Operator的处理结果
 - 调用子Operator的GetNext()
 - 扫描排序结果, 或者Run merge的一步
- **Close()**: 结束, 释放资源
 - 调用子Operator的close()
- 基于哈希的去重思路类似Join

Outline

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作

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

- 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

基于哈希的分组聚集

- 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

迭代求解实现 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

集合操作

- 并
- 交
- 差
- 笛卡尔积

SQL中的集合操作

- 并：保留字Union

$$\square A \cup B$$

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

- 交：保留字Intersect

$$\square A \cap B$$

集合操作保留字

- 差：保留字Except

$$\square A - B$$

(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

小节

- 选择
- 投影
- 连接
- 去重
- 分组+聚集
- 集合操作