第9章：查询执行

Query Execution

李东博

哈尔滨工业大学
计算学部
物联网与泛在智能研究中心
电子邮件：ldb@hit.edu.cn

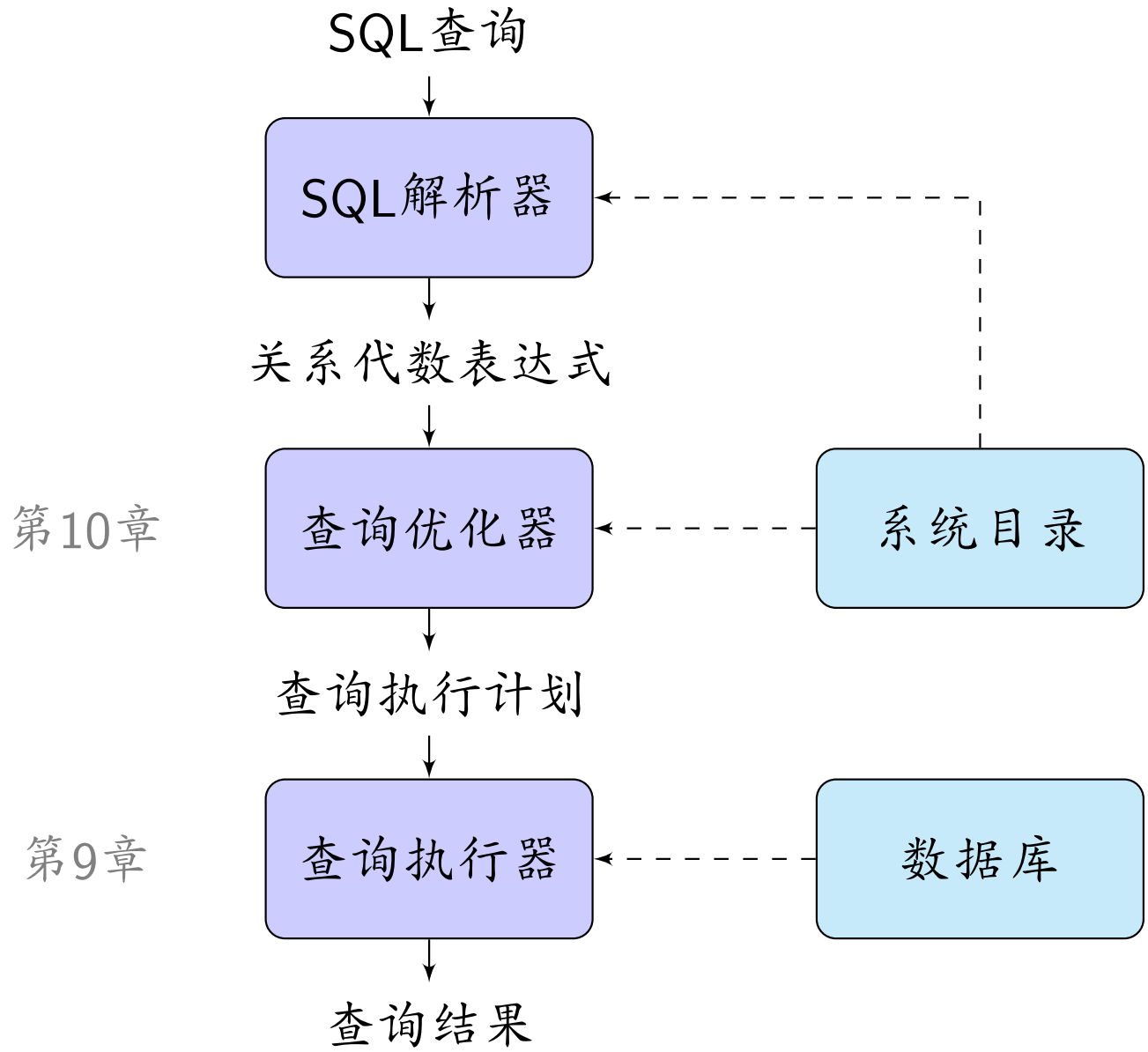
2023年春

教学内容[1]

[1]课件更新于2023年2月16日

Overview

查询处理(Query Processing)的基本过程



SQL解析器(SQL Parser)

SQL解析器将一个SQL查询转换成关系代数表达式[2]

Example (查询解析)

- 关系: Student(Sno, Sname, Ssex, Sage, Sdept)
- SQL查询:

```
SELECT Sno, Sage FROM Student WHERE Sage > 18;
```

- 关系代数表达式:

$$\Pi_{\text{Sno,Sage}}(\sigma_{\text{Sage}>18}(\text{Student}))$$

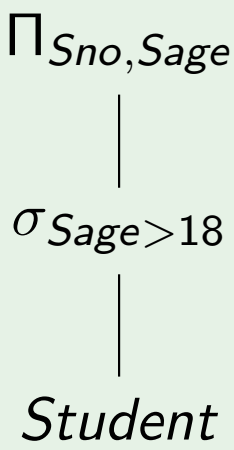
[2]含扩展的关系代数操作，如去重(deduplication)、排序(sorting)等

查询优化器(Query Optimizer)

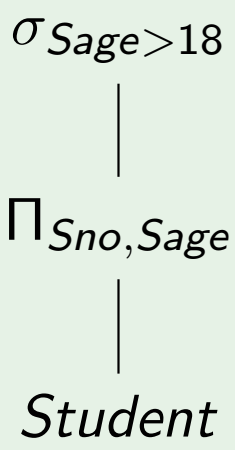
查询优化器将一个关系代数表达式转换为一个执行效率最高的等价关系代数表达式，并最终转换为一个查询执行计划(query execution plan)

Example (查询优化)

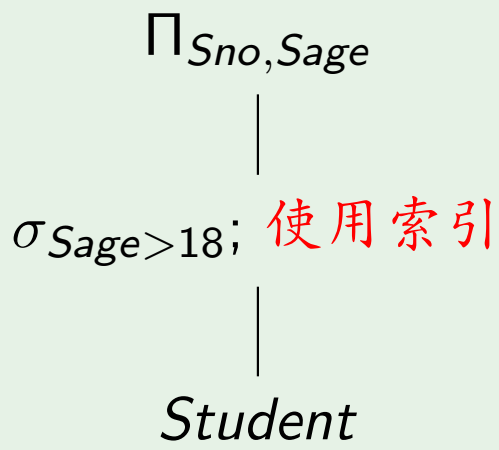
初始关系代数表达式



等价关系代数表达式



查询执行计划



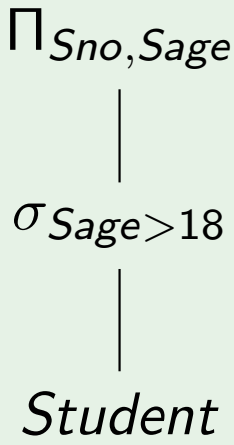
查询执行计划(Query Execution Plan)

原语操作(primitive operation)是带有“如何执行”注释的关系代数操作

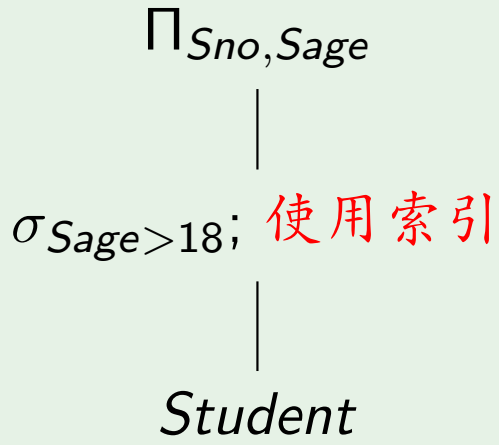
查询执行计划是用于执行一个查询的原语操作序列

Example (查询执行计划)

关系代数表达式



查询执行计划



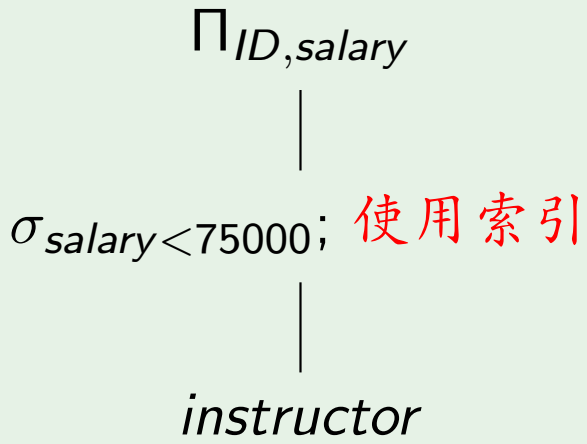
查询执行器(Query Executor)

查询执行器按照查询优化器给出的查询执行计划执行查询

- 关系代数操作的执行算法(第2–3节讲)
- 关系代数操作间的数据传递方法(第4节讲)

Example (查询执行)

查询执行计划



External Sort

排序(Sorting)

按照排序键(sort key)对元组进行排序是DBMS中非常重要的操作

- 用户使用ORDER BY对查询结果进行排序
- 批量加载(bulk loading) B+树的第一步是对索引项(index entry)进行排序
- 排序是关系代数操作执行过程的重要步骤

外排序(External Sorting)

当数据规模大到无法全部载入内存时，需要使用外排序算法

最小化外排序算法的磁盘访问(disk access)开销

- CPU计算时间在外排序算法的执行时间中只占很少一部分
- 外排序算法的执行时间主要用于磁盘访问
- 磁盘访问开销用磁盘I/O数量来近似衡量

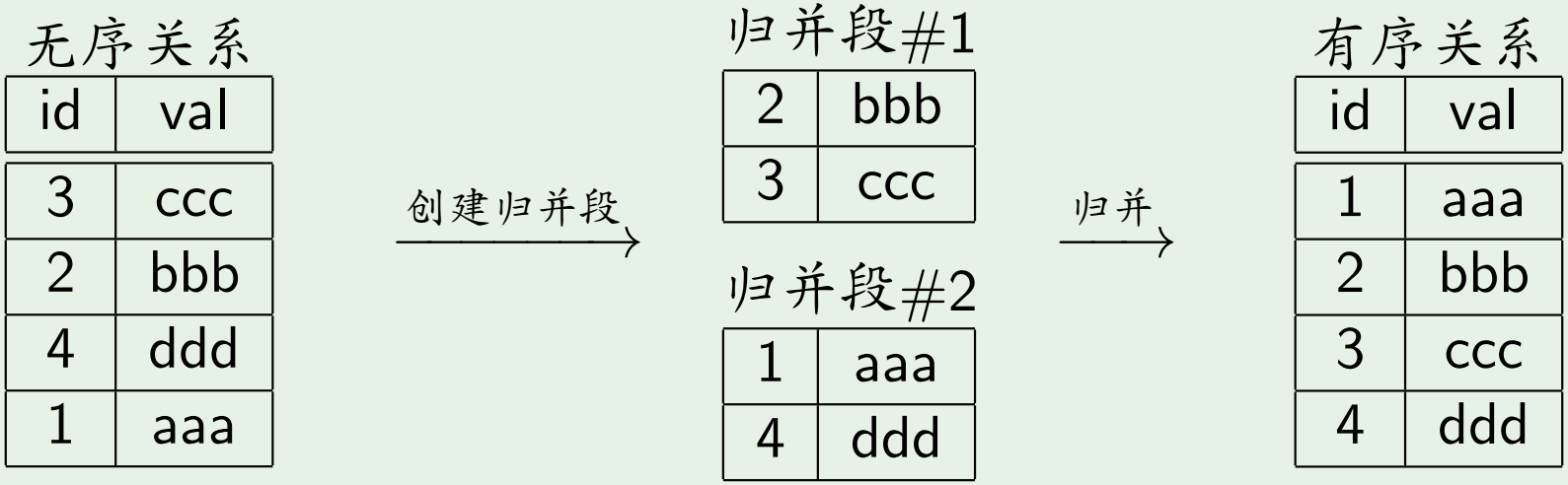
External Sort
External Merge Sort

两趟多路外存归并排序(Two-Pass Multi-Way External Merge Sort)

- 第1阶段: 创建归并段(run)
- 第2阶段: 归并(merge)

Example (两趟多路外存归并排序)

将关系中的元组按id属性排序



记法

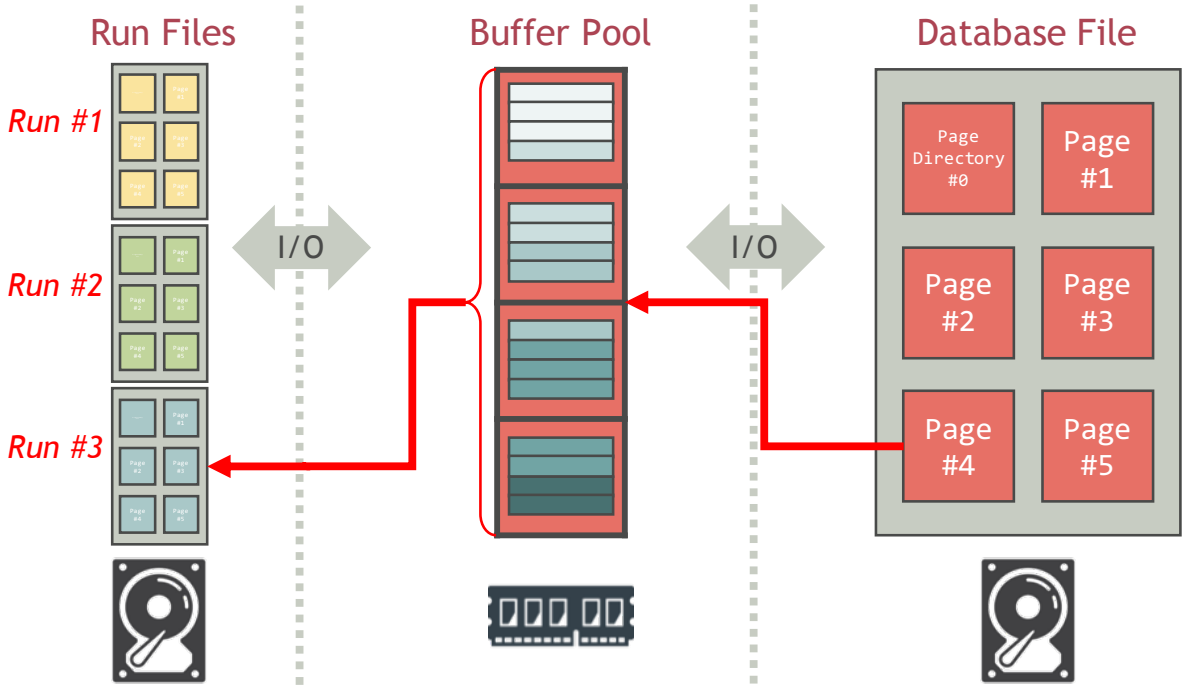
- N: R的元组数
- M: 缓冲池中可用内存页数
- B: 每块最多存B个元组
- $B(R)$: R的块数$\lceil N/B \rceil$

创建归并段

将关系R划分为$\lceil B(R)/M \rceil$个归并段(run)

过程: 创建归并段

1: **for** R的每M块 **do**
2: 将这M块读入缓冲池中M页
3: 对这M页中的元组按排序键进行排序，形成归并段
4: 将该归并段写入文件



多路外存归并排序算法运行示例

Example (外存多路归并排序)

- $R =$ | 2 5 | 2 1 | 2 2 | 4 5 | 4 3 | 4 2 | 1 5 | 2 1 | 3 | (每个数代表一个元组)
- $N = 17$
- $M = 3$
- $B = 2$
- $B(R) = 9$

创建归并段

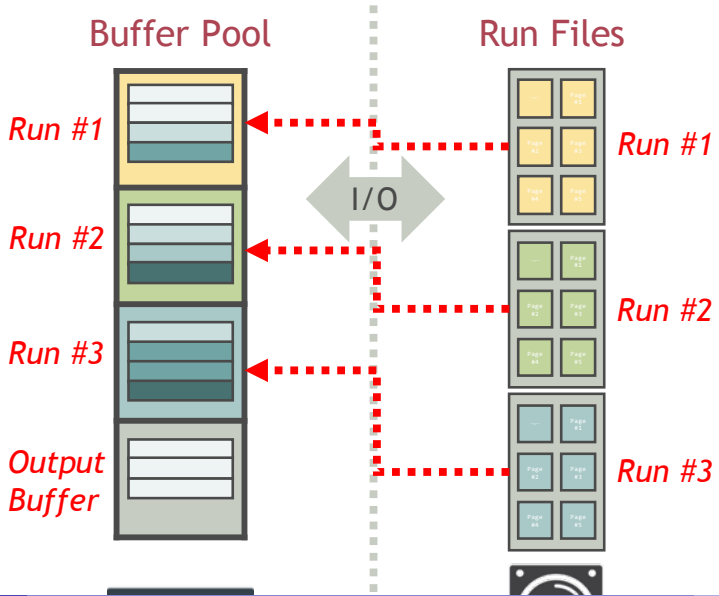
- $R_1 =$ | 1 2 | 2 2 | 2 5 |
- $R_2 =$ | 2 3 | 4 4 | 4 5 |
- $R_3 =$ | 1 1 | 2 3 | 5 |

多路归并(Multi-Way Merge)

将$\lceil B(R)/M \rceil$个归并段中的元组进行归并(merge)

过程: 多路归并

1: 将每个归并段的第一块读入缓冲池的一页
2: **repeat**
3: 找出所有输入缓冲页中最小的排序键值v
4: 将所有排序键值等于v的元组写入输出缓冲区
5: 任意输入缓冲页中的元组若归并完毕，则读入其归并段的下一页
6: **until** 所有归并段都已归并完毕



多路外存归并排序算法运行示例

多路归并

最小排序键=1

Run	In memory	Waiting on disk
R_1	~~1~~ 2	2 2 | 2 5
R_2	2 3	4 4 | 4 5
R_3	~~1~~ ~~1~~	2 3 | 5

输出: 1 1 | 1 (红色的数代表刚被输出的元组)

最小排序键=2

Run	In memory	Waiting on disk
R_1	~~2~~	2 2 | 2 5
R_2	~~2~~ 3	4 4 | 4 5
R_3	~~2~~ 3	5

输出: 1 1 | 1 2 | 2 2

多路外存归并排序算法运行示例(续)

最小排序键=2

Run	In memory	Waiting on disk
R_1	~~2~~ ~~2~~	2 5
R_2	3	4 4 | 4 5
R_3	3	5

输出: 1 1 | 1 2 | 2 2 | 2 2

最小排序键=2

Run	In memory	Waiting on disk
R_1	~~2~~ 5	
R_2	3	4 4 | 4 5
R_3	3	5

输出: 1 1 | 1 2 | 2 2 | 2 2 | 2

多路外存归并排序算法运行示例(续)

最小排序键=3

Run	In memory	Waiting on disk
R_1	5	
R_2	~~3~~	4 4 | 4 5
R_3	~~3~~	5

输出: 1 1 | 1 2 | 2 2 | 2 2 | 2 3 | 3

最小排序键=4

Run	In memory	Waiting on disk
R_1	5	
R_2	~~4 4~~	4 5
R_3	5	

输出: 1 1 | 1 2 | 2 2 | 2 2 | 2 3 | 3 4 | 4

多路外存归并排序算法运行示例(续)

最小排序键=4

Run	In memory	Waiting on disk
R_1	5	
R_2	~~4~~ 5	
R_3	5	

输出:

1 1	1 2	2 2	2 2	2 3	3 4	4 4	

最小排序键=5

Run	In memory	Waiting on disk
R_1	~~5~~	
R_2	~~5~~	
R_3	~~5~~	

输出:

1 1	1 2	2 2	2 2	2 3	3 4	4 4	5 5	5

算法结束

算法分析

分析算法时不考虑结果输出操作

输出结果时产生的I/O不计入算法的I/O代价

- 输出结果可能直接作为后续操作的输入，无需写入文件

输出缓冲区不计入可用内存页数

- 如果输出结果直接作为后续操作的输入，那么输出缓冲区将计入后续操作的可用内存页数

算法分析

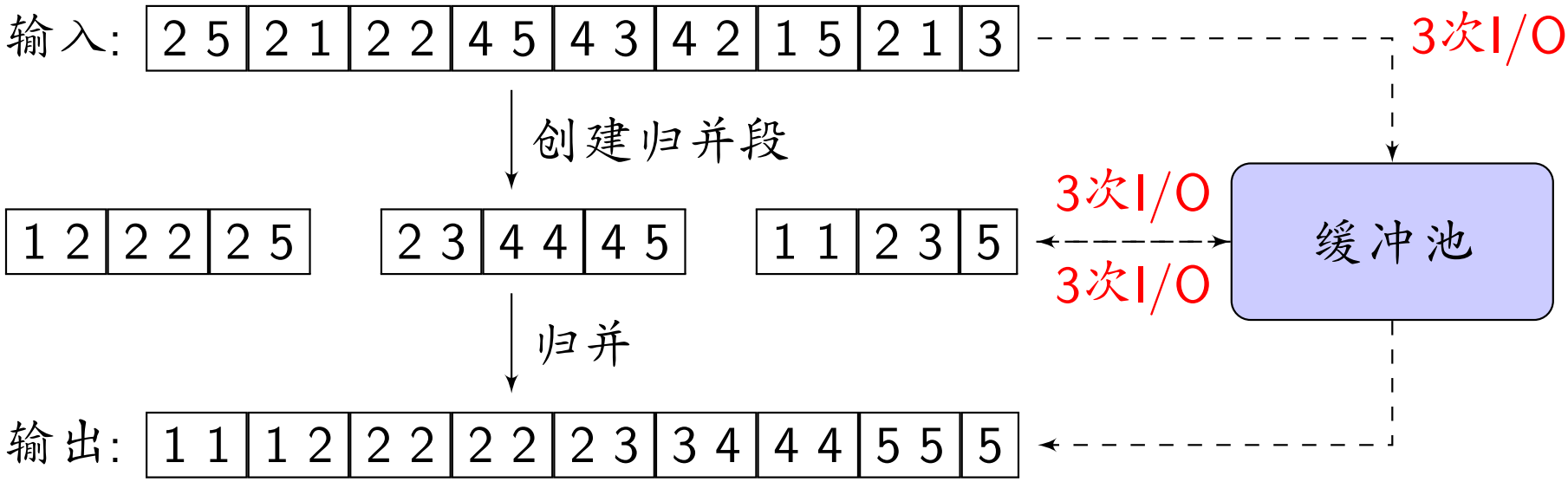
I/O代价: $3B(R)$

- 在创建归并段时，R的每块读1次，合计$B(R)$次I/O
- 将每个归并段写入文件，合计$B(R)$次I/O
- 在归并阶段，每个归并段扫描1次，合计$B(R)$次I/O

可用内存页数要求: $B(R) \leq M^2$

- 每个归并段不超过M页
- 最多M个归并段

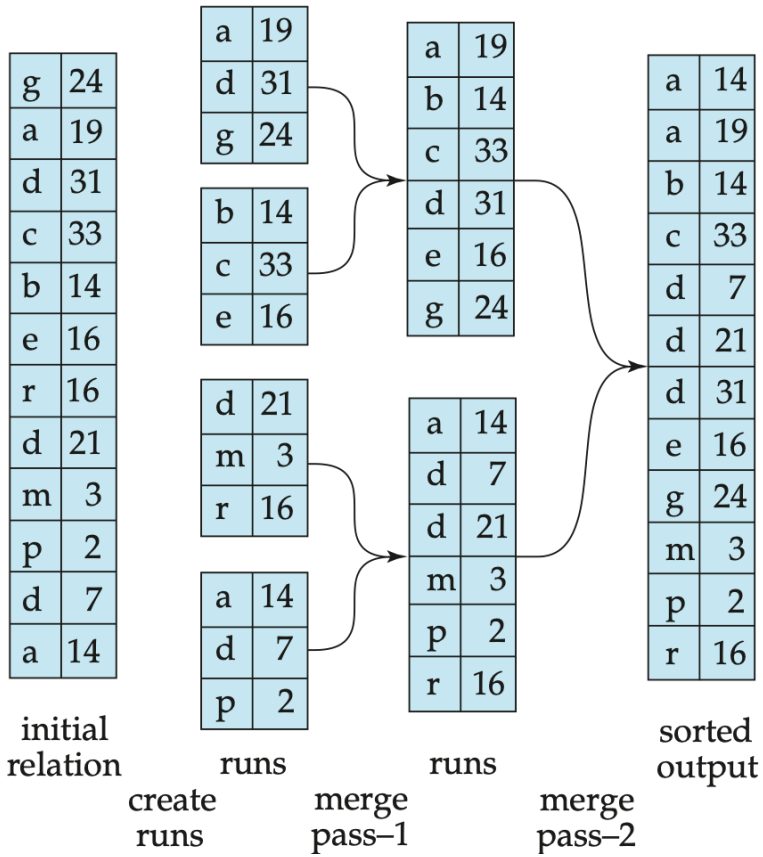
算法分析(续)


输入:
2 5 2 1 2 2 4 5 4 3 4 2 1 5 2 1 3
3次I/O
创建归并段
1 2 2 2 2 5
2 3 4 4 4 5
1 1 2 3 5
3次I/O
3次I/O
缓冲池
归并
输出:
1 1 1 2 2 2 2 2 2 3 3 4 4 4 5 5 5

多趟多路外存归并排序(Multi-Pass Multi-Way External Merge Sort)

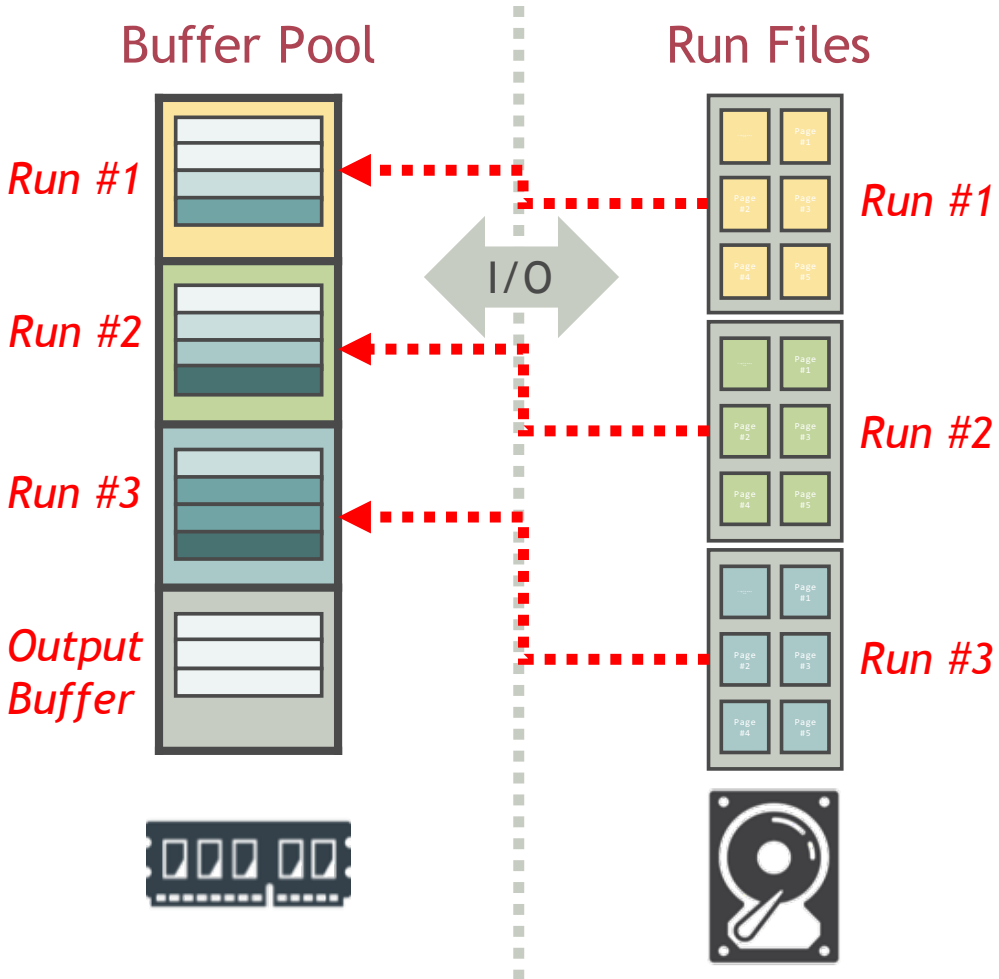
若$B(R) > M^2$，则需要执行多趟多路外存归并排序

- I/O代价是$(2m-1)B(R)$，其中m算法执行的趟数



多路归并排序的优化

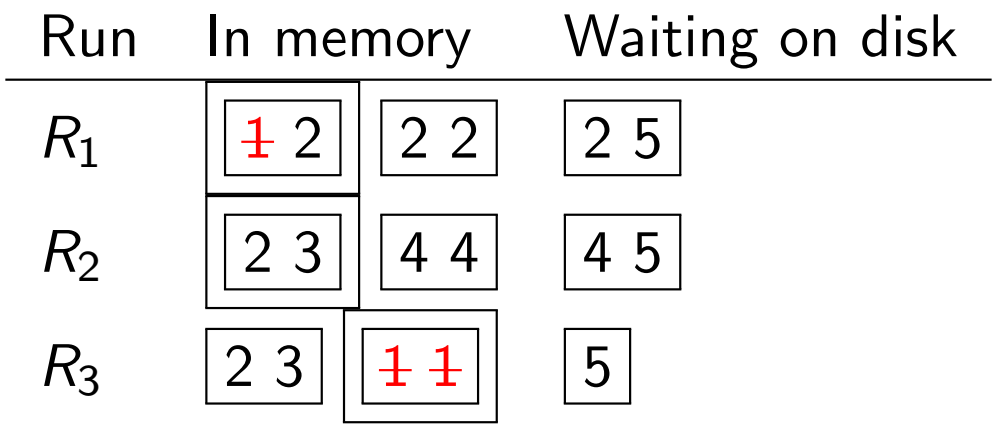
当某缓冲页中所有元组都被归并完毕，DBMS需读入其归并段的下一块
此时，归并进程被挂起(suspend)，直至I/O完成



双缓冲(Double Buffering)

为每个归并段分配多个内存页作为输入缓冲区，并组成环形(circular)

- 在当前缓冲页中所有元组都已归并完毕时，DBMS直接开始归并下一缓冲页中的元组
- 与此同时，DBMS将归并段文件的下一块读入空闲的缓冲页



Execution of Relational Algebraic Operations

Execution of Relational Algebraic Operations
Execution of Selection Operations

选择操作的执行

- 方法1: 基于扫描的选择算法(Scanning-based Selection)
- 方法2: 基于哈希的选择算法(Hash-based Selection)
- 方法3: 基于索引的选择算法(Index-based Selection)

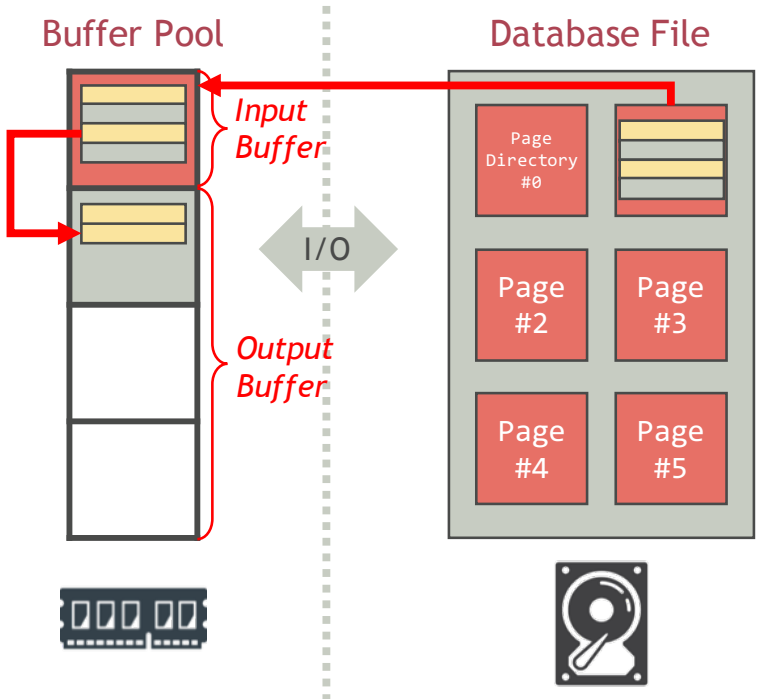
记法

- $T(R)$: 关系R的元组数
- $B(R)$: 关系R的块数
- M: 缓冲池可用内存页数
- $V(R, A)$: 关系R的属性集A的不同值的个数

基于扫描的选择算法(Scanning-based Selection)

算法

1: **for** R的每一块P **do**
2: 　将P读入缓冲池
3: 　**for** P中每条元组t **do**
4: 　　**if** t满足选择条件 **then**
5: 　　　将t写入输出缓冲区



算法分析

分析算法时不考虑结果输出操作

输出结果时产生的I/O不计入算法的I/O代价

- 输出结果可能直接作为后续操作的输入，无需写入文件

输出缓冲区不计入可用内存页数

- 如果输出结果直接作为后续操作的输入，那么输出缓冲区将计入后续操作的可用内存页数

算法分析

I/O代价: $B(R)$

- R的元组连续存储于文件中
- R的每块只读1次

可用内存页数要求: $M \geq 1$

- 至少需要1页作为缓冲区，用于读R的每一块

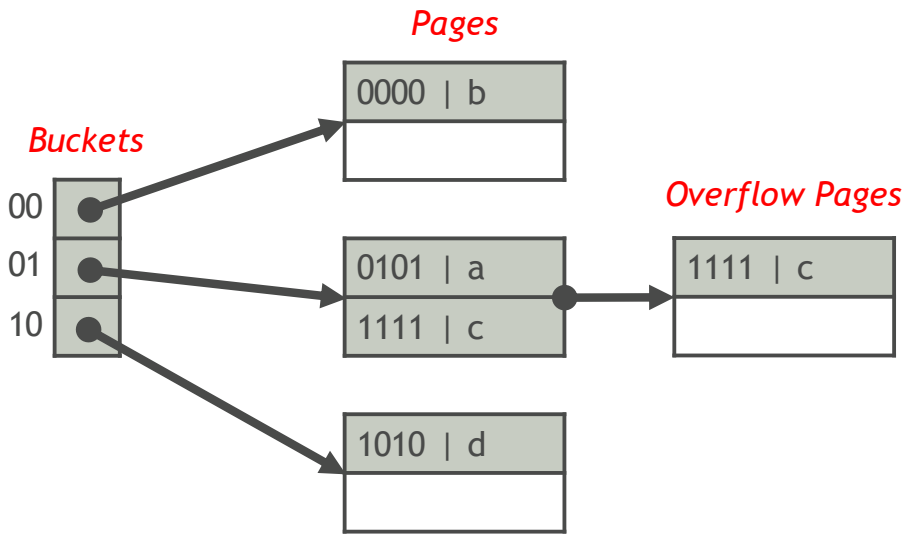
基于哈希的选择算法(Hash-based Selection)

使用该算法的前提条件

- 选择条件的形式是$K = v$
- 关系R采用哈希文件组织形式(hash-based file organization)
- 属性K是R的哈希键(hash key)

算法

1: 根据$hash(v)$确定结果元组所在的桶
2: 在桶页面中搜索键值等于v的元组，并将元组写入输出缓冲区



算法分析

I/O代价 $\approx \lceil B(R)/V(R,K) \rceil$

- 属性K有$V(R,K)$个不同的值
- 每个桶平均有$\lceil B(R)/V(R,K) \rceil$个页(很不准确的估计)

可用内存页数要求: $M \geq 1$

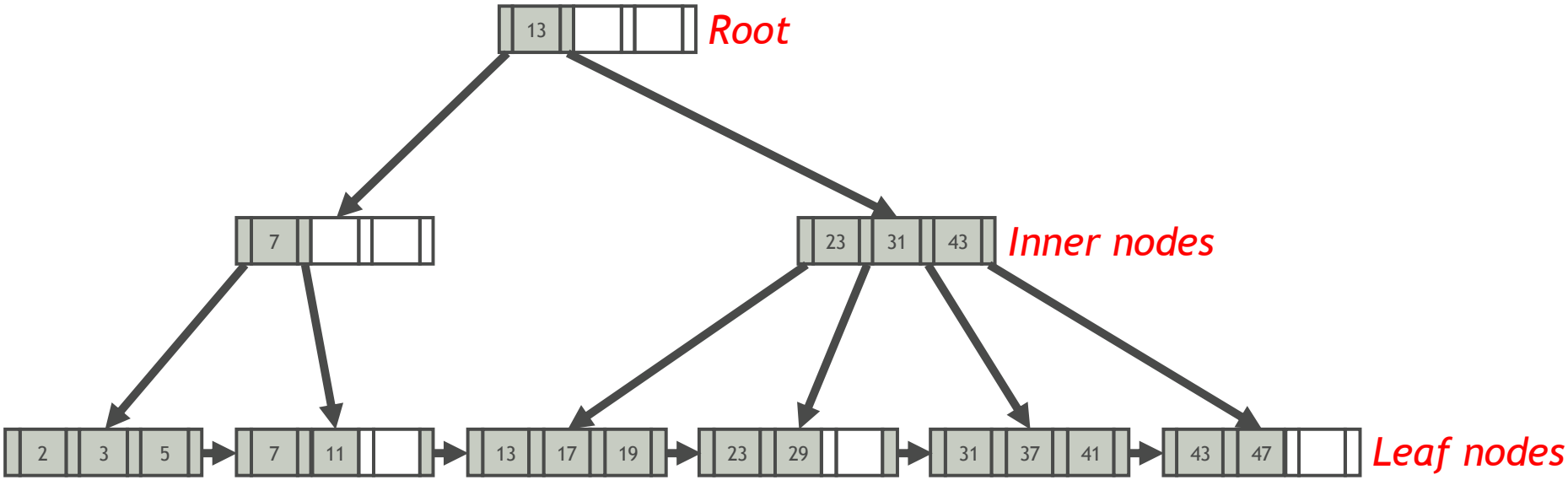
- 至少需要1页作为缓冲区，用于读桶的每一页

基于索引的选择算法(Index-based Selection)

使用该算法的前提条件

- 选择条件的形式是$K = v$或$l \leq K \leq u$
- 关系R上建有属性K的索引

在索引上搜索满足选择条件的元组，并将元组写入输出缓冲区



算法分析

I/O代价 $\approx \lceil B(R)/V(R,K) \rceil$ (如果索引是聚簇索引)

- 结果元组连续存储于文件中
- 属性K有$V(R,K)$个不同的值
- 结果元组约占$\lceil B(R)/V(R,K) \rceil$个页(很不准确的估计)

I/O代价 $\approx \lceil T(R)/V(R,K) \rceil$ (如果索引是非聚簇索引)

- 约有$\lceil T(R)/V(R,K) \rceil$个结果元组(很不准确的估计)
- 结果元组不一定连续存储于文件中
- 最坏情况下，所有结果元组均在不同页上

可用内存页数要求: $M \geq 1$

- 至少需要1页作为缓冲区，用于读B+树的节点

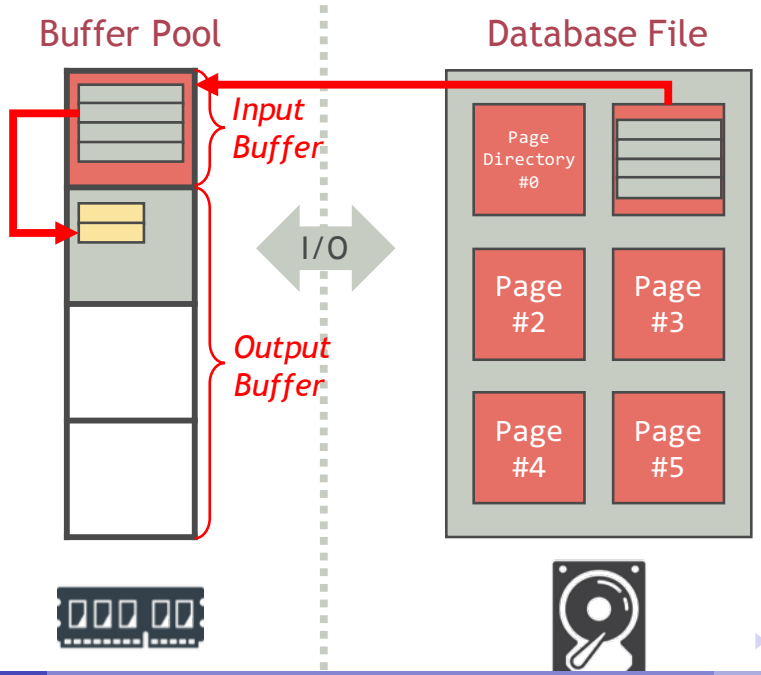
Execution of Relational Algebraic Operations
Execution of Projection Operations

不带去重的投影算法

算法

1: **for** R的每一块P **do**
2: 　将P读入缓冲池
3: 　**for** P中每条元组t **do**
4: 　　将t向投影属性集做投影
5: 　　将投影元组写入输出缓冲区

该算法在数据访问模式(access pattern)上与基于扫描的选择算法相同



算法分析

I/O代价: $B(R)$

- R的元组连续存储于文件中
- R的每块只读1次

可用内存页数要求: $M \geq 1$

- 至少需要1页作为缓冲区，用于读R的每一块

Execution of Relational Algebraic Operations
Execution of Duplicate Elimination Operations

去重(Duplicate Elimination)操作

关系R上的去重操作$\delta(R)$返回R中互不相同的元组

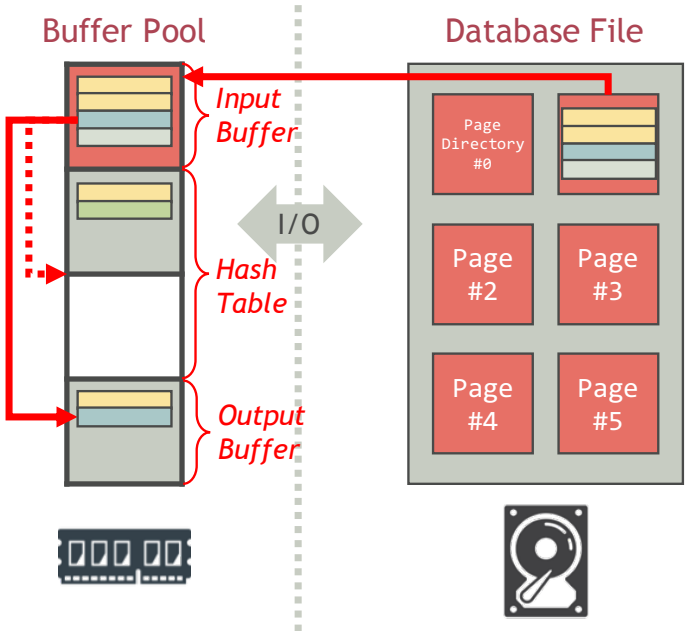
- 方法1: 一趟去重算法(One-Pass Duplicate Elimination)
- 方法2: 基于排序的去重算法(Sort-based Duplicate Elimination)
- 方法3: 基于哈希的去重算法(Hash-based Duplicate Elimination)

一趟去重算法(One-Pass Duplicate Elimination)

算法

1: **for** R的每一块P **do**
2: 　将P读入缓冲池
3: 　**for** P中每条元组t **do**
4: 　　**if** 未见过t **then**
5: 　　　将t写入输出缓冲区

在可用内存页中用哈希表记录见过的元组，哈希键为整个元组



一趟去重算法算法运行示例

Example (一趟去重算法)

- $R =$ | 2 5 | 2 1 | 2 2 | 4 5 | 4 3 | 4 2 | 1 5 | 2 1 | 3 |
- $M = 4$
- $B = 2$

输入缓冲区(1块)	哈希表($< M$块)	输出缓冲区
2 5	2 5	2 5
2 1	2 5 | 1	2 5 | 1
2 2	2 5 | 1	2 5 | 1
4 5	2 5 | 1 4	2 5 | 1 4
4 3	2 5 | 1 4 | 3	2 5 | 1 4 | 3
4 2	2 5 | 1 4 | 3	2 5 | 1 4 | 3
1 5	2 5 | 1 4 | 3	2 5 | 1 4 | 3
2 1	2 5 | 1 4 | 3	2 5 | 1 4 | 3
3	2 5 | 1 4 | 3	2 5 | 1 4 | 3

算法分析

该算法在数据访问模式上与基于扫描的选择算法相同

I/O代价: $B(R)$

- R的元组连续存储于文件中
- R的每块只读1次

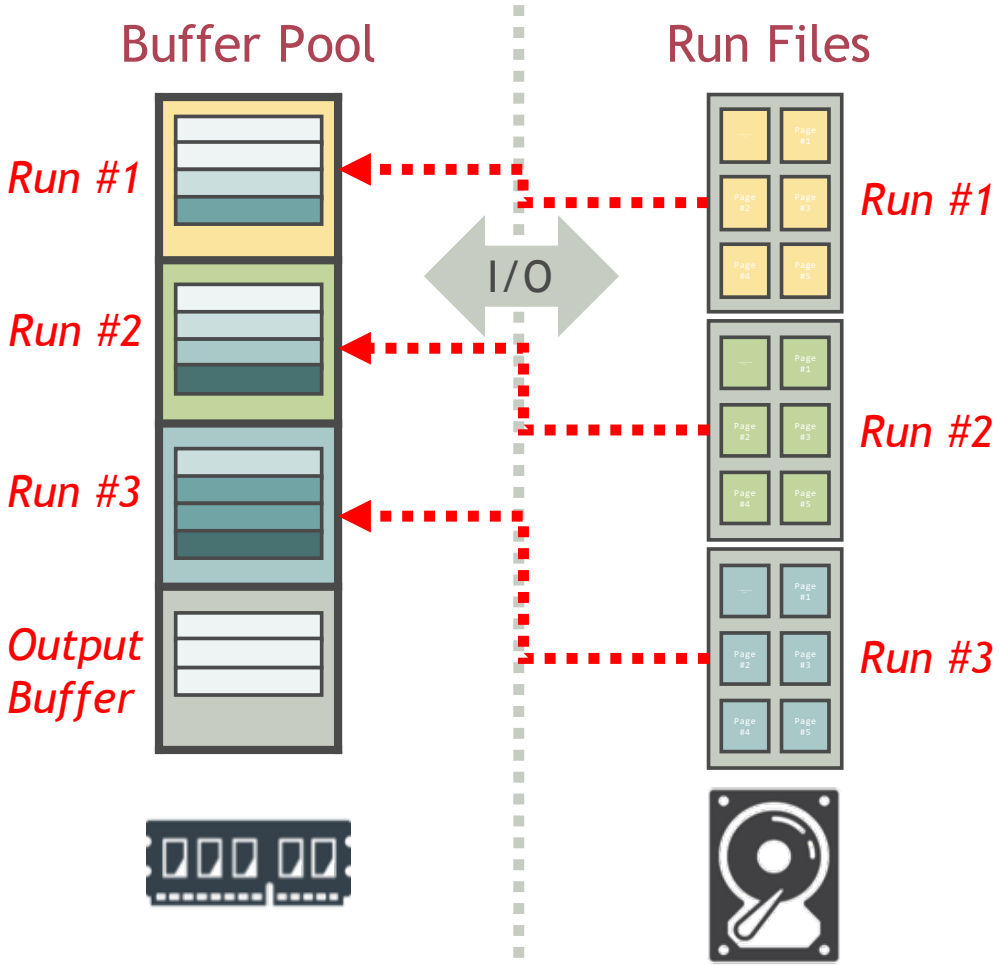
可用内存页数要求: $B(\delta(R)) \leq M - 1$

- R中互不相同的元组$\delta(R)$必须能在$M-1$页中存得下

基于排序的去重算法(Sort-based Duplicate Elimination)

基于排序的去重算法与多路归并排序(multi-way merge sort)算法本质上一样，两点区别如下:

- 在创建归并段(run)时，按整个元组进行排序
- 在归并阶段，相同元组只输出1个，其他全部丢弃



基于排序的去重算法运行示例

Example (基于排序的去重算法)

- $R =$ | 2 5 | 2 1 | 2 2 | 4 5 | 4 3 | 4 2 | 1 5 | 2 1 | 3 |
- $M = 3$
- $B = 2$

创建归并段

- $R_1 =$ | 1 2 | 2 2 | 2 5 |
- $R_2 =$ | 2 3 | 4 4 | 4 5 |
- $R_3 =$ | 1 1 | 2 3 | 5 |

基于排序的去重算法运行示例(续)

多路归并

最小元组=1

Run	In memory	Waiting on disk
R_1	~~1~~ 2	2 2 | 2 5
R_2	2 3	4 4 | 4 5
R_3	~~1~~ ~~1~~	2 3 | 5

输出: 1 (红色的数代表刚输出的去重元组)

最小元组=2

Run	In memory	Waiting on disk
R_1	~~2~~	2 2 | 2 5
R_2	~~2~~ 3	4 4 | 4 5
R_3	~~2~~ 3	5

输出: 1 2

基于排序的去重算法运行示例(续)

最小元组=2

Run	In memory	Waiting on disk
R_1	~~2~~ ~~2~~	2 5
R_2	3	4 4 | 4 5
R_3	3	5

输出: 1 2

最小元组=2

Run	In memory	Waiting on disk
R_1	~~2~~ 5	
R_2	3	4 4 | 4 5
R_3	3	5

输出: 1 2

基于排序的去重算法运行示例(续)

最小元组=3

Run	In memory	Waiting on disk
R_1	5	
R_2	~~3~~	4 4 | 4 5
R_3	~~3~~	5

输出: 1 2 | 3

最小元组=4

Run	In memory	Waiting on disk
R_1	5	
R_2	~~4 4~~	4 5
R_3	5	

输出: 1 2 | 3 4

基于排序的去重算法运行示例(续)

最小元组=4

Run	In memory	Waiting on disk
R_1	5	
R_2	~~4~~ 5	
R_3	5	

输出: 1 2 | 3 4

最小元组=5

Run	In memory	Waiting on disk
R_1	~~5~~	
R_2	~~5~~	
R_3	~~5~~	

输出: 1 2 | 3 4 | 5 算法结束

算法分析

I/O代价: $3B(R)$

- 在创建归并段时，R的每块读1次，合计$B(R)$次I/O
- 将每个归并段写入文件，合计$B(R)$次I/O
- 在归并阶段，每个归并段扫描1次，合计$B(R)$次I/O

可用内存页数要求: $B(R) \leq M^2$

- 每个归并段不超过M页
- 最多M个归并段

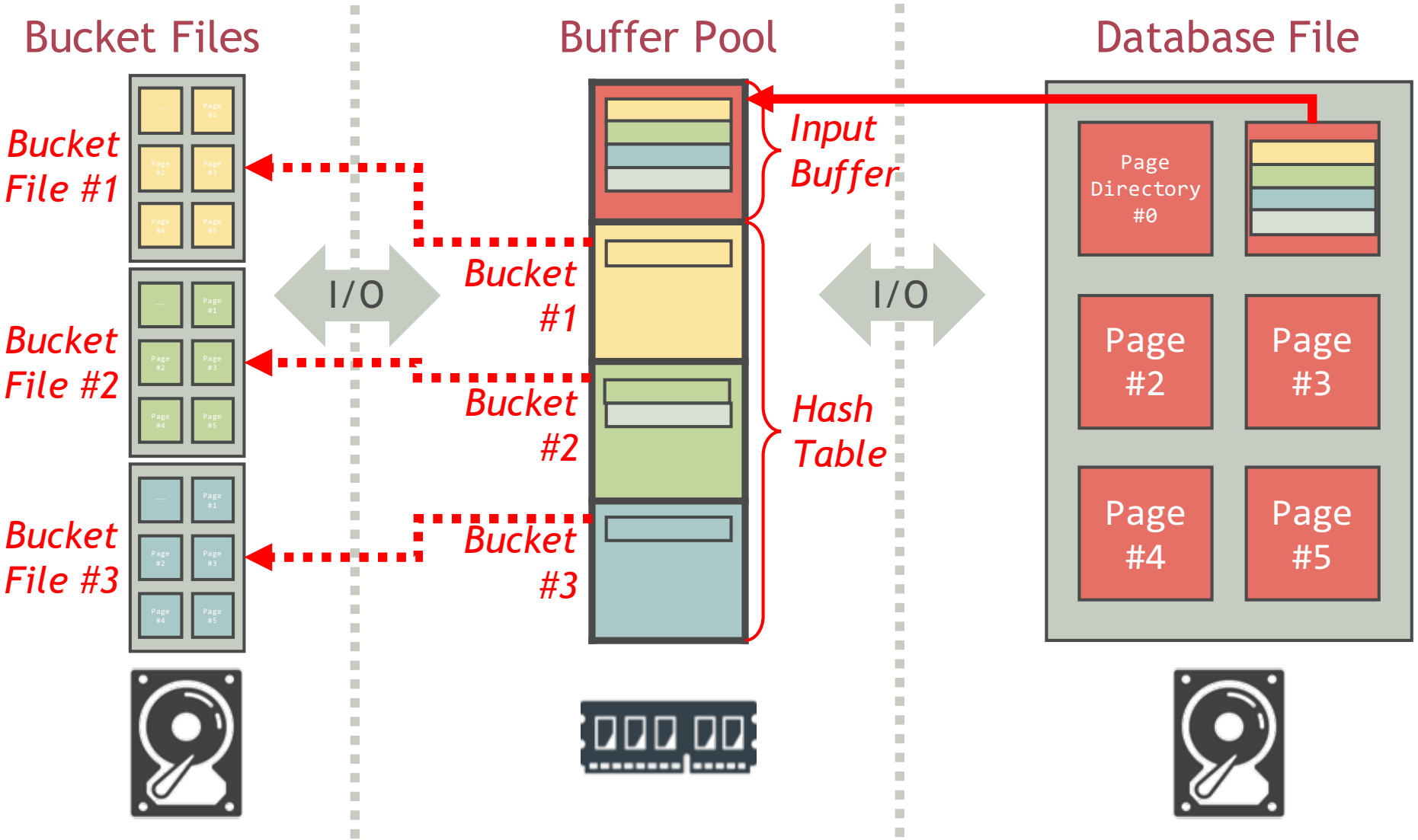
基于哈希的去重算法(Hash-based Duplicate Elimination)

算法

1: // 哈希分桶
2: **for** R的每一块P **do**
3: 将P读入缓冲池
4: 将P中元组哈希到$M-1$个桶$R_1, R_2, \ldots, R_{M-1}$中(哈希键为整个元组)
5: // 逐桶去重
6: **for** $i = 1, 2, \ldots, M-1$ **do**
7: 在桶R_i上执行一趟去重(one-pass duplicate elimination)算法，并将结果写到输出缓冲区

哈希分桶

性质：重复元组必落入相同桶中



按桶去重

每个桶R_i的去重结果放在一起就得到了R的去重结果

- $\delta(R) = \bigcup_{i=1}^{M-1} \delta(R_i)$
- $\delta(R_i) \cap \delta(R_j) = \emptyset$ for $i \neq j$

基于哈希的去重算法运行示例

Example (基于哈希的去重算法)

- $R =$ | 2 5 | 2 1 | 2 2 | 4 5 | 4 3 | 4 2 | 1 5 | 2 1 | 3 |
- $M = 4$
- $B = 2$
- $h(K) = K \mod 3$

哈希分桶

- $R_0 =$ | 3 3 |
- $R_1 =$ | 1 4 | 4 4 | 1 1 |
- $R_2 =$ | 2 5 | 2 2 | 2 5 | 2 5 | 2 |

逐桶去重

- $\delta(R_0) =$ | 3 |
- $\delta(R_1) =$ | 1 4 |
- $\delta(R_2) =$ | 2 5 |

算法分析

I/O代价: $3B(R)$

- 在哈希分桶时，R的每块读1次，合计$B(R)$次I/O
- 将每个桶写入文件，合计$\sum_{i=1}^{M-1} B(R_i) \approx B(R)$次I/O
- 在每个桶R_i上执行一趟去重算法的I/O代价是$B(R_i)$

可用内存页数要求: $B(R) \leq (M-1)^2$

- 共$M-1$个桶
- 每个桶不超过$M-1$块，因此在每个桶上执行一趟去重算法时，可用内存页数满足要求

Execution of Relational Algebraic Operations
Execution of Aggregation Operations

聚集操作(Aggregation Operations)的执行

聚集操作和去重操作的执行在本质上一样

- 方法1: 一趟聚集算法(One-Pass Aggregation)
- 方法2: 基于排序的聚集算法(Sort-based Aggregation)
- 方法3: 基于哈希的聚集算法(Hash-based Aggregation)

聚集算法的设计和分析留作课后练习

Execution of Relational Algebraic Operations
Execution of Set Operations

集合差操作的执行

- 方法1: 一趟集合差算法(One-Pass Set Difference)
- 方法2: 基于排序的集合差算法(Sort-based Set Difference)
- 方法3: 基于哈希的集合差算法(Hash-based Set Difference)

一趟集合差算法(One-Pass Set Difference)

算法

1: // 构建(build)阶段
2: 在$M-1$个可用内存页中建立一个内存查找结构(哈希表或平衡二叉树)，查找键是整个元组
3: **for** S的每一块P **do**
4: 将P读入缓冲池
5: 将P中元组插入内存查找结构
6: // 探测(probe)阶段
7: **for** R的每一块P **do**
8: 将P读入缓冲池
9: **for** P中每条元组t **do**
10: **if** t不在于内存查找结构中 **then**
11: 将t写入输出缓冲区

一趟集合差算法运行示例

Example (一趟集合差算法)

- $R =$ | 1 5 | 8 2 | 3 10 | 4 7 | 6 9 |
- $S =$ | 4 11 | 9 5 | 7 3 | 6 12 | 8 10 |
- $M = 6$
- $B = 2$

缓冲区前$M-1$块: | 4 11 | 9 5 | 7 3 | 6 12 | 8 10 | (内存哈希表)

序号	缓冲区第M块	输出
1	1 5	1
2	8 2	1 2
3	3 10	1 2
4	4 7	1 2
5	6 9	1 2

算法分析

I/O代价: $B(R) + B(S)$

- 在构建(build)阶段，S的每块只读1次，合计$B(S)$次I/O
- 在探测(probe)阶段，R的每块只读1次，合计$B(R)$次I/O

可用内存页数要求: $B(S) \leq M - 1$

- 内存查找结构约占$B(S)$页

基于哈希的集合差算法(Hash-based Set Difference)

算法

1: // 哈希分桶(与基于哈希的去重算法的分桶方法相同)
2: 将R的元组哈希到$M-1$个桶$R_1, R_2, \ldots, R_{M-1}$中(哈希键为整个元组)
3: 将S的元组哈希到$M-1$个桶$S_1, S_2, \ldots, S_{M-1}$中(哈希键为整个元组)
4: // 逐桶计算集合差
5: **for** $i = 1, 2, \ldots, M-1$ **do**
6: 使用一趟集合差(one-pass set difference)算法计算$R_i - S_i$，并将结果写入输出缓冲区

- R和S中相同的元组一定分别落入同号桶R_i和S_i中
- $R - S = \bigcup_{i=1}^{M-1}(R_i - S_i)$
- $(R_i - S_i) \cap (R_j - S_j) = \emptyset$ for $i \neq j$

基于哈希的集合差算法运行示例

Example (基于哈希的集合差算法)

- $R =$ | 1 5 | 8 2 | 3 10 | 4 7 | 6 9 |
- $S =$ | 4 11 | 9 5 | 7 3 | 6 12 | 8 10 |
- $M = 4$
- $B = 2$
- $h(K) = K \mod 3$

R的桶

- $R_0 =$ | 3 6 | 9 |
- $R_1 =$ | 1 10 | 4 7 |
- $R_2 =$ | 5 2 | 8 |

S的桶

- $S_0 =$ | 9 3 | 6 12 |
- $S_1 =$ | 4 7 | 10 |
- $S_2 =$ | 11 5 | 8 |

集合差

- $R_0 - S_0 = \emptyset$
- $R_1 - S_1 =$ | 1 |
- $R_2 - S_2 =$ | 2 |

算法分析

I/O代价: $3B(R)+3B(S)$

- 在对R进行哈希分桶时，R的每块读1次，合计$B(R)$次I/O
- 将R的桶全部写入文件，需$\sum_{i=1}^{M-1} B(R_i) \approx B(R)$次I/O
- 在对S进行哈希分桶时，S的每块读1次，合计$B(S)$次I/O
- 将S的桶全部写入文件，需$\sum_{i=1}^{M-1} B(S_i) \approx B(S)$次I/O
- 使用一趟集合差算法计算$R_i - S_i$的I/O代价是$B(R_i)+B(S_i)$

可用内存页数要求: $B(S) \leq (M-1)^2$

- S共有$M-1$个桶
- S的每个桶不超过$M-1$块

基于排序的集合差算法(Sort-based Set Difference)

算法

1: // 创建归并段
2: 将R划分为$\lceil B(R)/M \rceil$个归并段(每个归并段按整个元组进行排序)
3: 将S划分为$\lceil B(S)/M \rceil$个归并段(每个归并段按整个元组进行排序)
4: // 归并
5: 读入R和S的每个归并段的第1页
6: **repeat**
7: 找出输入缓冲区中最小的元组t
8: **if** $t \in R$且$t \notin S$ **then**
9: 将t写入输出缓冲区
10: 从输入缓冲区中删除t的所有副本
11: 任意输入缓冲页中的元组若归并完毕，则读入其归并段的下一页
12: **until** R的所有归并段都已归并完毕

算法分析

I/O代价: $3B(R) + 3B(S)$

- 在对R创建归并段时，R的每块只读1次，合计$B(R)$次I/O
- 将R的归并段全部写入文件，需$B(R)$次I/O
- 在对S创建归并段时，S的每块只读1次，合计$B(S)$次I/O
- 将S的归并段全部写入文件，需$B(S)$次I/O
- 在归并阶段，对R和S的每个归并段各扫描1次，合计$B(R) + B(S)$次I/O

可用内存页数要求: $B(R) + B(S) \leq M^2$

- R和S的每个归并段均不超过M块
- R和S共有不超过M个归并段

集合并操作的执行

集合并操作和集合差操作的执行在本质上一样

- 方法1: 一趟集合并算法(One-Pass Set Union)
- 方法2: 基于排序的集合并算法(Sort-based Set Union)
- 方法3: 基于哈希的集合并算法(Hash-based Set Union)

集合并算法的设计和分析留作课后练习

集合交操作的执行

集合交操作和集合差操作的执行在本质上一样

- 方法1: 一趟集合交算法(One-Pass Set Intersection)
- 方法2: 基于排序的集合交算法(Sort-based Set Intersection)
- 方法3: 基于哈希的集合交算法(Hash-based Set Intersection)

集合交算法的设计和分析留作课后练习

Execution of Relational Algebraic Operations
Execution of Join Operations

连接(Join)操作的执行

下面以$R(X,Y) \bowtie S(Y,Z)$为例，介绍连接操作的执行算法

- 方法1: 一趟连接算法(One-Pass Join)
- 方法2: 嵌套循环连接算法(Nested-Loop Join)
- 方法3: 排序归并连接算法(Sort-Merge Join)
- 方法4: 哈希连接算法(Hash Join)
- 方法5: 索引连接算法(Index Join)

一趟连接算法(One-Pass Join)

假设: $B(S) \leq B(R)$

算法

```
1: // 构建(build)阶段
2: 在M − 1个可用内存页中建立一个内存查找结构(哈希表或平衡二叉
   树)，查找键是S.Y
3: for S的每一块P do
4:     将P读入缓冲池
5:     将P中元组插入内存查找结构
6: // 探测(probe)阶段
7: for R的每一块P do
8:     将P读入缓冲池
9:     for P中每条元组r do
10:        for 内存查找结构中每条键值等于r.Y的元组s do
11:            连接r和s，并将结果写入输出缓冲区
```

一趟连接算法运行示例

Example (一趟连接算法)

- $R(X, Y) =$ | (1, 1), (5, 5) | (3, 2), (3, 1) | (2, 1), (4, 2) |
- $S(Y, Z) =$ | (2, 6), (1, 7) | (1, 8), (2, 5) | (2, 7) |
- $M = 4$
- $B = 2$

缓冲区前$M-1$块: | (2, 6), (1, 7) | (1, 8), (2, 5) | (2, 7) | (内存哈希表)

序号	缓冲区第M块	输出
1	(1, 1), (5, 5)	$(1,1,7),(1,1,8)$
2	(3, 2), (3, 1)	$\ldots,(3,2,6),(3,2,5),(3,2,7),(3,1,7),(3,1,8)$
3	(2, 1), (4, 2)	$\ldots,(2,1,7),(2,1,8),(4,2,6),(4,2,5),(4,2,7)$

算法分析

I/O代价: $B(R) + B(S)$

- 在构建(build)阶段，S的每块只读1次，合计$B(S)$次I/O
- 在探测(probe)阶段，R的每块只读1次，合计$B(R)$次I/O

可用内存页数要求: $B(S) \leq M - 1$

- 内存查找结构约占$B(S)$页

基于元组的嵌套循环连接(Tuple-based Nested-Loop Join)

算法

```
1: for S的每个元组s do
2:     for R的每个元组r do
3:         if r和s满足连接条件 then
4:             连接r和s，并将结果写入输出缓冲区
```

- S称为外关系(outer relation)
- R称为内关系(inner relation)

算法分析

I/O代价: $T(S)(T(R)+1)$

- 外关系S的每个元组只读1次，每次产生1个I/O，合计$T(S)$次I/O
- 内关系R的每个元组读$T(S)$次，每次产生1个I/O，合计$T(S)T(R)$次I/O

可用内存页数要求: $M \geq 2$

- 1页作为读S的缓冲区
- 1页作为读R的缓冲区

基于块的嵌套循环连接(Block-based Nested-Loop Join)

假设: $B(S) \leq B(R)$

算法

1: **for** 外关系S的每$M-1$块 **do**
2: 将这$M-1$块读入缓冲池
3: 用一个内存查找结构来组织这$M-1$块中的元组
4: **for** 内关系R的每一块P **do**
5: 将P读入缓冲池
6: **for** P中每条元组r **do**
7: **for** 内存查找结构中能与r进行连接的元组s **do**
8: 连接r和s，并将结果写入输出缓冲区

基于块的嵌套循环连接算法运行示例

Example (基于块的嵌套循环连接算法)

- $R(X, Y) =$ | (1, 1), (5, 5) | (3, 2), (3, 1) | (2, 1), (4, 2) |
- $S(Y, Z) =$ | (2, 6), (1, 7) | (1, 8), (2, 5) | (2, 7) |
- $M = 3$
- $B = 2$

缓冲区前$M-1$块		缓冲区第M块	输出
(2, 6), (1, 7)	(1, 8), (2, 5)	(1, 1), (5, 5)	(1, 1, 7), (1, 1, 8)
(2, 6), (1, 7)	(1, 8), (2, 5)	(3, 2), (3, 1)	(3, 2, 6), (3, 2, 5), (3, 1, 7), (3, 1, 8)
(2, 6), (1, 7)	(1, 8), (2, 5)	(2, 1), (4, 2)	(2, 1, 7), (2, 1, 8), (4, 2, 6), (4, 2, 5)
(2, 7)		(1, 1), (5, 5)	
(2, 7)		(3, 2), (3, 1)	(3, 2, 7)
(2, 7)		(2, 1), (4, 2)	(4, 2, 7)

算法分析

I/O代价: $B(S) + \frac{B(R)B(S)}{M-1}$

- 外关系S的每一块只读1次，合计$B(S)$次I/O
- 内关系R扫描$B(S)/(M-1)$次，合计$\frac{B(R)B(S)}{M-1}$次I/O

可用内存页数要求: $M \geq 2$

- 至少1页作为读S的缓冲区
- 1页作为读R的缓冲区

排序归并连接(Sort-Merge Join)

算法

1: // 创建归并段
2: 将R划分为$\lceil B(R)/M \rceil$个归并段(每个归并段按$R.Y$进行排序)
3: 将S划分为$\lceil B(S)/M \rceil$个归并段(每个归并段按$S.Y$进行排序)
4: // 归并
5: 读入R和S的每个归并段的第1页
6: **repeat**
7: 找出输入缓冲区中元组Y属性的最小值y
8: **for** R中满足$R.Y = y$的元组r **do**
9: **for** S中满足$S.Y = y$的元组s **do**
10: 连接r和s，并将结果写入输出缓冲区
11: 任意输入缓冲页中的元组若归并完毕，则读入其归并段的下一页
12: **until** R或S的所有归并段都已归并完毕

排序归并连接算法运行示例

Example (排序归并连接算法)

- $R(X, Y) =$

(1, 1), (5, 4)	(3, 2), (3, 1)	(6, 3), (2, 1)	(4, 2), (8, 5)	(4, 1), (3, 4)

- $S(Y, Z) =$

(2, 6), (1, 7)	(1, 8), (5, 9)	(5, 3), (2, 5)	(3, 1), (2, 7)	(3, 7), (4, 9)

- $M = 4$
- $B = 2$

创建归并段

- $R_1 =$

(1, 1), (2, 1)	(3, 1), (3, 2)	(6, 3), (5, 4)

- $R_2 =$

(4, 1), (4, 2)	(3, 4), (8, 5)

- $S_1 =$

(1, 7), (1, 8)	(2, 5), (2, 6)	(5, 3), (5, 9)

- $S_2 =$

(2, 7), (3, 1)	(3, 7), (4, 9)

排序归并连接算法运行示例(续)

多路归并

最小连接键=1

Run	In memory	Waiting on disk	
R_1	(1, 1), (2, 1)	(3, 1), (3, 2)	(6, 3), (5, 4)
R_2	(4, 1), (4, 2)	(3, 4), (8, 5)	
S_1	(1, 7), (1, 8)	(2, 5), (2, 6)	(5, 3), (5, 9)
S_2	(2, 7), (3, 1)	(3, 7), (4, 9)	

最小连接键=1

Run	In memory	Waiting on disk	
R_1	(1, 1), (2, 1)	(6, 3), (5, 4)	
	(3, 1), (3, 2)		
R_2	(4, 1), (4, 2)	(3, 4), (8, 5)	
S_1	(1, 7), (1, 8)	(2, 5), (2, 6)	(5, 3), (5, 9)
S_2	(2, 7), (3, 1)	(3, 7), (4, 9)	

输出: $(1,1,7),(1,1,8),(2,1,7),(2,1,8),(3,1,7),(3,1,8),(4,1,7),(4,1,8)$

排序归并连接算法运行示例(续)

最小连接键=2

Run	In memory	Waiting on disk
R_1	(3, 2)	(6, 3), (5, 4)
R_2	(4, 2)	(3, 4), (8, 5)
S_1	(2, 5), (2, 6)	(5, 3), (5, 9)
S_2	(2, 7), (3, 1)	(3, 7), (4, 9)

最小连接键=2

Run	In memory	Waiting on disk
R_1	(3, 2)	
	(6, 3), (5, 4)	
R_2	(4, 2)	
	(3, 4), (8, 5)	
S_1	(2, 5), (2, 6)	(5, 3), (5, 9)
S_2	(2, 7), (3, 1)	(3, 7), (4, 9)

输出: ..., (3, 2, 5), (3, 2, 6), (3, 2, 7), (4, 2, 5), (4, 2, 6), (4, 2, 7)

排序归并连接算法运行示例(续)

最小连接键=3

Run	In memory	Waiting on disk
R_1	(6, 3), (5, 4)	
R_2	(3, 4), (8, 5)	
S_1	(5, 3), (5, 9)	
S_2	(3, 1)	(3, 7), (4, 9)

输出: ..., (6, 3, 1)

最小连接键=3

Run	In memory	Waiting on disk
R_1	(6, 3), (5, 4)	
R_2	(3, 4), (8, 5)	
S_1	(5, 3), (5, 9)	
S_2	(3, 7), (4, 9)	

输出: ..., (6, 3, 7)

排序归并连接算法运行示例(续)

最小连接键=4

Run	In memory	Waiting on disk
R_1	(5, 4)	
R_2	(3, 4), (8, 5)	
S_1	(5, 3), (5, 9)	
S_2	(4, 9)	

输出: ..., (5, 4, 9), (3, 4, 9)

最小连接键=5

Run	In memory	Waiting on disk
R_1		
R_2	(8, 5)	
S_1	(5, 3), (5, 9)	
S_2		

输出: ..., (8, 5, 3), (8, 5, 9) 算法结束

算法分析

I/O代价: $3B(R) + 3B(S)$

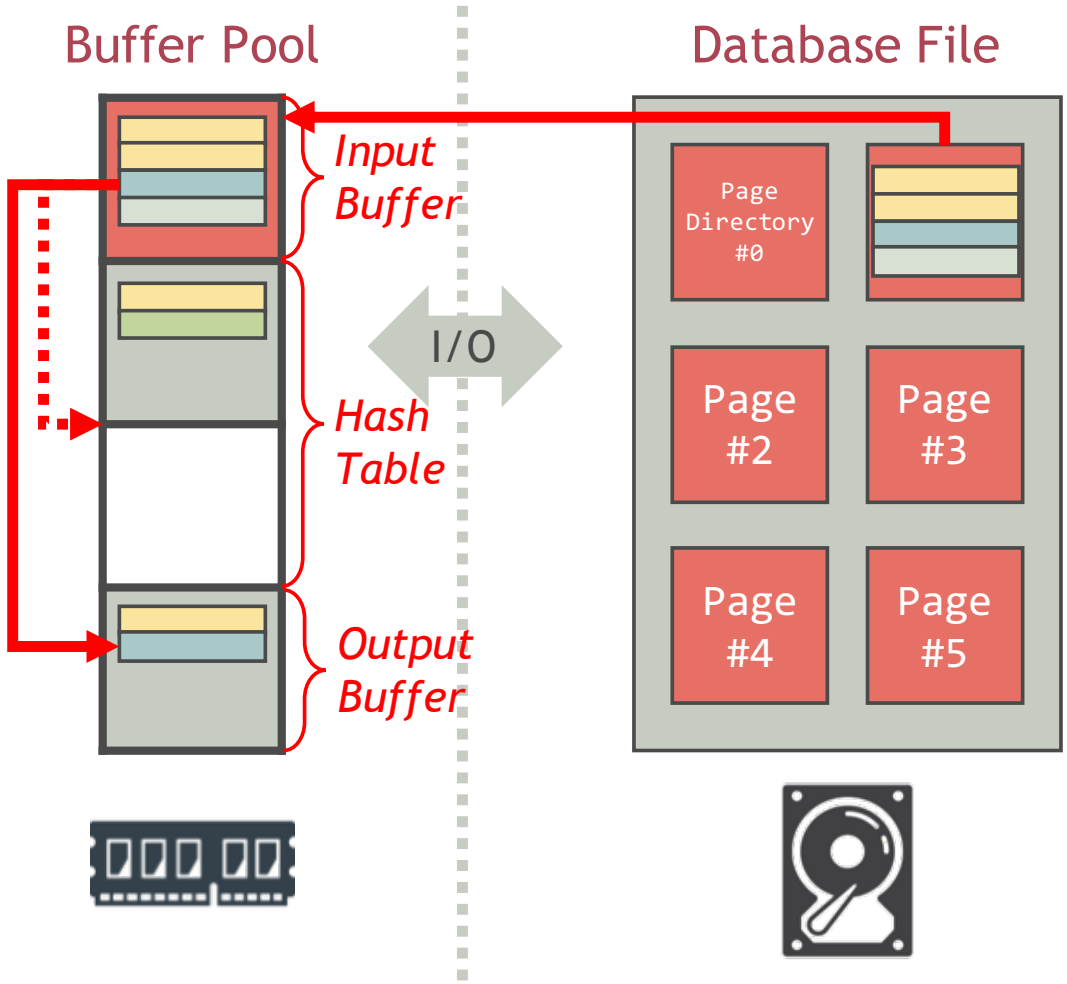
- 在对R创建归并段时，R的每块只读1次，合计$B(R)$次I/O
- 将R的归并段全部写入文件，需$B(R)$次I/O
- 在对S创建归并段时，S的每块只读1次，合计$B(S)$次I/O
- 将S的归并段全部写入文件，需$B(S)$次I/O
- 在归并阶段，对R和S的每个归并段各扫描1次，合计$B(R) + B(S)$次I/O

可用内存页数要求: $B(R) + B(S) \leq M^2$

- R和S的每个归并段均不超过M块
- R和S共有不超过M个归并段

经典哈希连接(Classic Hash Join)

如果一趟连接算法使用的内存查找结构是哈希表，则该算法称为经典哈希连接算法



Grace哈希连接(Grace Hash Join)

算法

1: // 哈希分桶
2: 将R的元组哈希到$M-1$个桶$R_1, R_2, \ldots, R_{M-1}$中(哈希键为$R.Y$)
3: 将S的元组哈希到$M-1$个桶$S_1, S_2, \ldots, S_{M-1}$中(哈希键为$S.Y$)
4: // 逐桶连接
5: **for** $i = 1, 2, \ldots, M-1$ **do**
6: 使用一趟连接(one-pass join)算法计算$R_i \bowtie S_i$，并将结果写入输出缓冲区

- R和S中相同的元组一定分别落入同号桶R_i和S_i中
- $R \bowtie S = \bigcup_{i=1}^{M-1}(R_i \bowtie S_i)$
- $(R_i \bowtie S_i) \cap (R_j \bowtie S_j) = \emptyset$ for $i \neq j$

算法分析

I/O代价: $3B(R) + 3B(S)$

- 在对R进行哈希分桶时，R的每块读1次，合计$B(R)$次I/O
- 将R的桶全部写入文件，需$\sum_{i=1}^{M-1} B(R_i) \approx B(R)$次I/O
- 在对S进行哈希分桶时，S的每块读1次，合计$B(S)$次I/O
- 将S的桶全部写入文件，需$\sum_{i=1}^{M-1} B(S_i) \approx B(S)$次I/O
- 使用一趟连接算法计算$R_i \bowtie S_i$的I/O代价是$B(R_i) + B(S_i)$

可用内存页数要求: $B(S) \leq (M-1)^2$

- S共有$M-1$个桶
- S的每个桶不超过$M-1$块

索引连接(Index Join)

假设：关系S上建有属性Y的索引

算法

1: **for** R的每一块P **do**
2: 　将P读入缓冲池
3: 　**for** P中每条元组r **do**
4: 　　在索引上查找键值等于$r.Y$的S的元组集合T
5: 　　**for** $s \in T$ **do**
6: 　　连接r和s，并将结果写入输出缓冲区

索引连接算法运行示例

Example (索引连接)

- $R(X, Y) =$ | (1, 1), (5, 5) | (3, 2), (3, 1) | (2, 1), (4, 2) |
- $S(Y, Z) =$ | (2, 6), (1, 7) | (1, 8), (2, 5) | (2, 7) | ($S.Y$上有索引)
- $M = 2$
- $B = 2$

序号	缓冲区	输出
1	(1, 1), (5, 5)	$(1,1,7),(1,1,8)$
2	(3, 2), (3, 1)	$\ldots,(3,2,6),(3,2,5),(3,2,7),(3,1,7),(3,1,8)$
3	(2, 1), (4, 2)	$\ldots,(2,1,7),(2,1,8),(4,2,6),(4,2,5),(4,2,7)$

算法分析

I/O代价: $B(R) + \frac{T(R)T(S)}{V(S,Y)}$ (若索引是非聚簇索引)

- R的每块只读1次，合计$B(R)$次I/O
- 对于R的每个元组r，S中平均约有$T(S)/V(S,Y)$个元组能与r连接
- 因为索引是非聚簇索引，这些元组在文件中不一定连续存储。最坏情况下，读每个元组产生1次I/O，合计$\frac{T(R)T(S)}{V(S,Y)}$次I/O

I/O代价: $B(R) + T(R)\lceil\frac{B(S)}{V(S,Y)}\rceil$ (若索引是聚簇索引)

- 因为索引是非聚簇索引，所以对于R的每个元组r，S中能与r连接的元组一定连续存储于S的文件中，约占$\lceil\frac{B(S)}{V(S,Y)}\rceil$个块

可用内存页数要求: $M \geq 2$

- 1页作为读R缓冲区
- 1页作为读索引节点缓冲区

Execution of Expressions

查询计划的执行方法

如何执行由多个操作构成的查询计划?

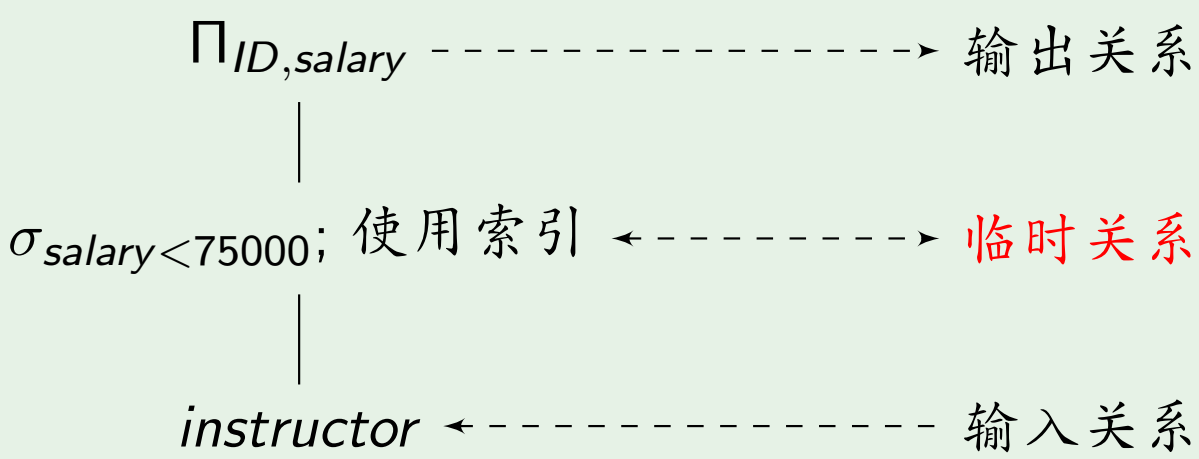
- 方法1: 物化执行(materialization)
- 方法2: 流水线执行(piplining)/火山模型(the volcano model)

Execution of Expressions
Materialization

物化执行(Materialization)

- 自底向上执行查询计划中的操作
- 每个中间操作的执行结果写入临时关系文件，作为后续操作的输入

Example (物化执行)



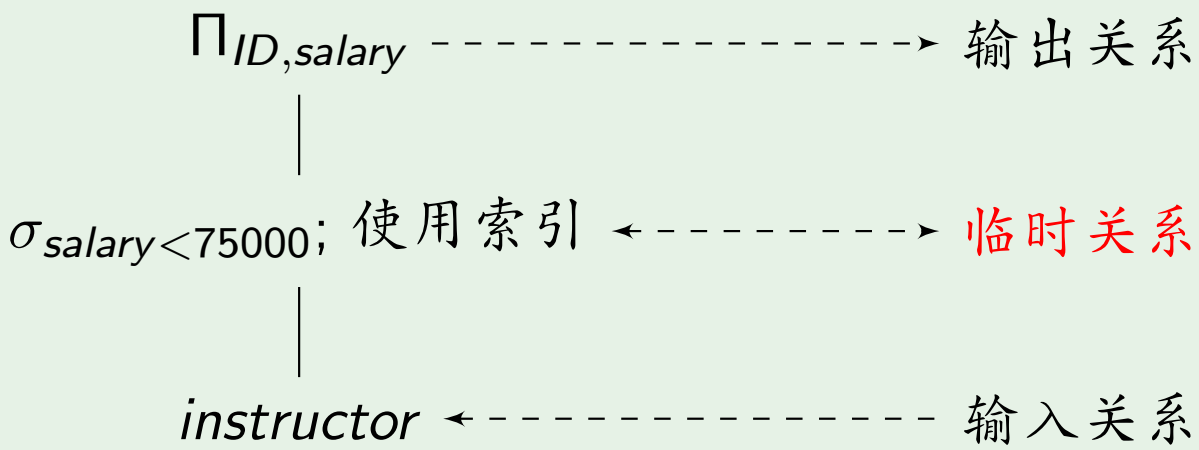
物化执行的缺点

缺点1: 物化(materialize)临时关系增加了查询执行的代价

- 执行完一个操作后，临时关系必须写入文件(除非临时关系非常小)
- 执行后续操作时，临时关系文件再被读入缓冲区

缺点2: 用户获得查询结果的时间延迟大

Example (物化执行)



Execution of Expressions
Piplining

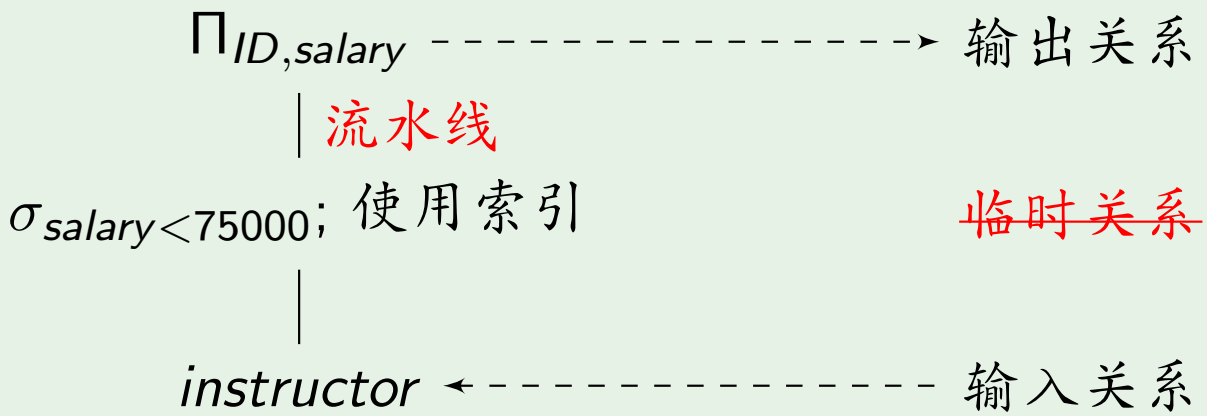
流水线执行(Piplining)/火山模型(The Volcano Model)

火山模型将查询计划中若干操作组成流水线(pipline)，一个操作的结果直接传给流水线中下一个操作

- 避免产生一些临时关系，避免了读写这些临时关系文件的I/O开销
- 用户能够尽早得到查询结果

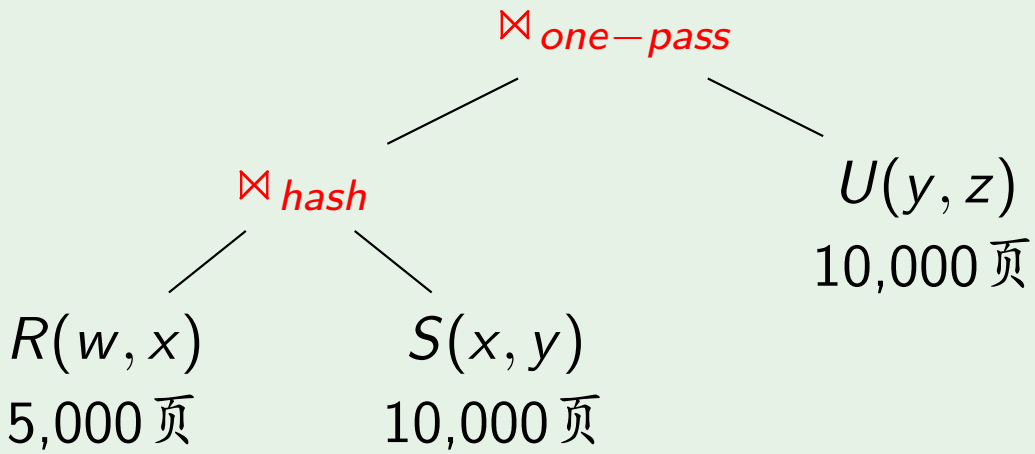
几乎所有DBMS都使用流水线执行查询计划

Example (流水线执行)

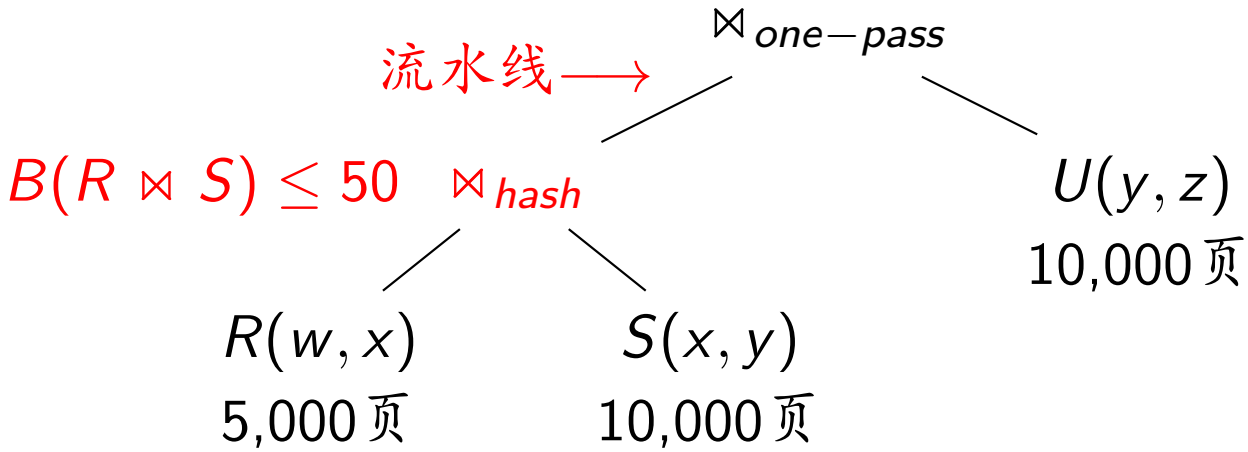


查询执行的示例

Example (查询执行)

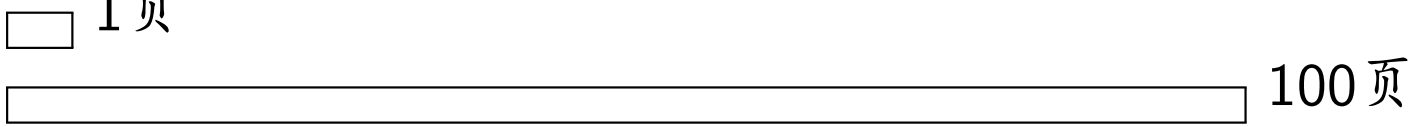


- 缓冲池中有$M = 101$个可用页
- R, S, U上均无索引且未按连接属性排序
- 设$B(R \bowtie S) \leq 50$

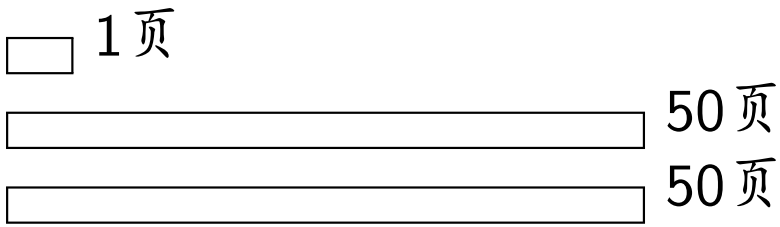


使用哈希连接(hash join)执行$R \bowtie S$

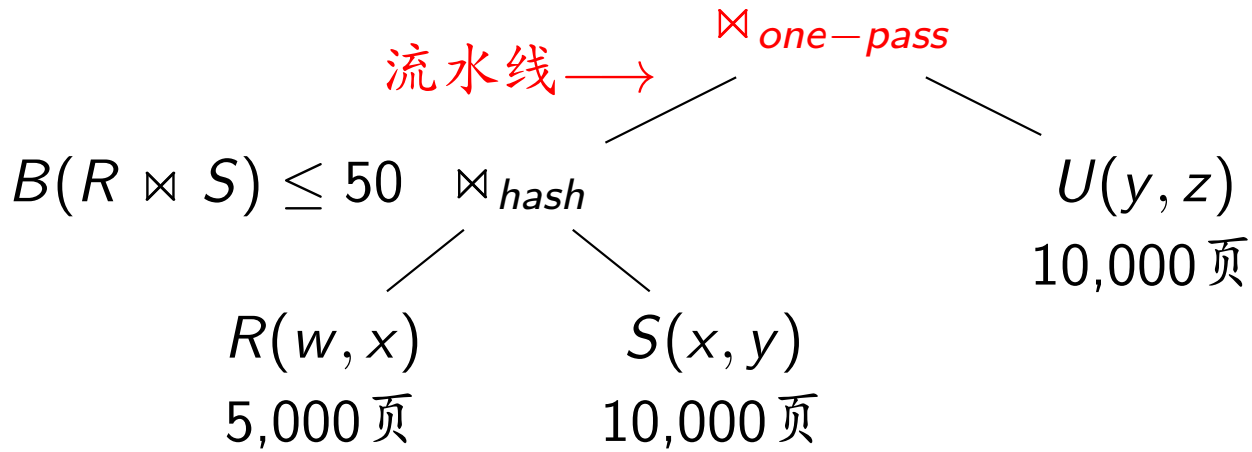
- 哈希分桶阶段使用101页内存



- 逐桶连接阶段使用51页内存(不计输出缓冲)

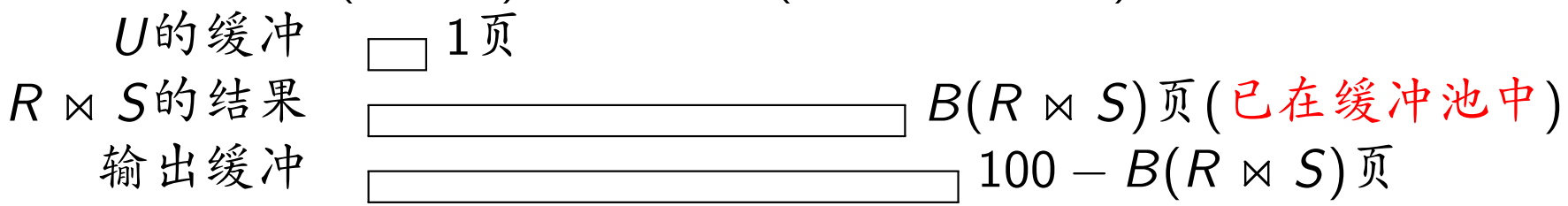


- I/O代价: $3B(R) + 3B(S) = 45000$
- 因为$B(R \bowtie S) \leq 50$，所以$R \bowtie S$的结果可以保留在输出缓冲区中，以流水线的形式输入给下一个操作$\bowtie_{one-pass}$



使用一趟连接(one-pass join)执行$R \bowtie S$的结果与U的连接

- 一趟连接使用$B(R \bowtie S)+1$页内存(不计输出缓冲)



- I/O代价: $B(U) = 10000$ ($R \bowtie S$的结果已在内存中，无需I/O)

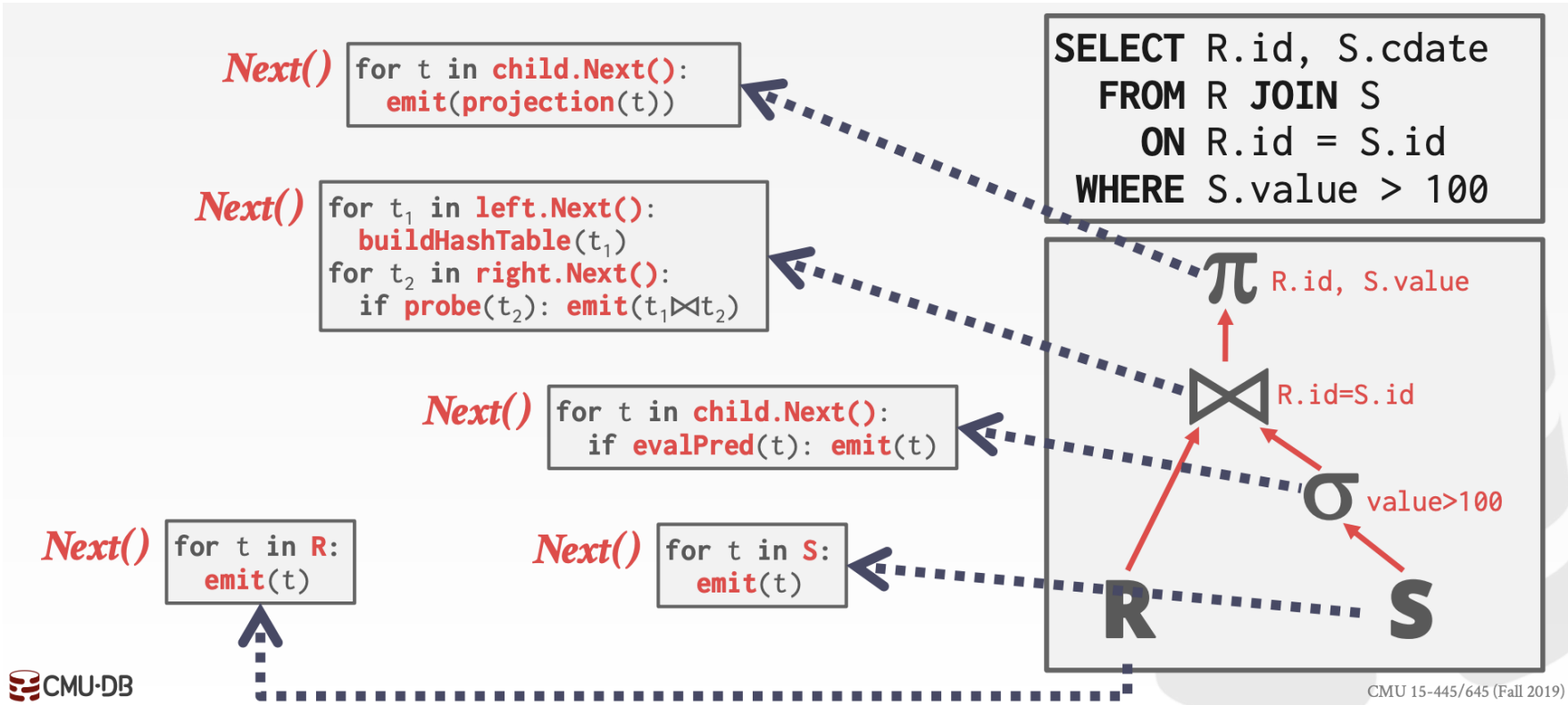
迭代器模型(Iterator Model)

使用迭代器(iterator)实现流水线中每个操作

- `open()`: 启动迭代器
- `next()`: 返回该操作的下一条结果元组
- `close()`: 关闭迭代器
- 迭代器需要维护其自身执行状态

迭代器模型(续)

- DBMS不断调用查询计划中最顶层操作的next()函数
- 如果一个操作的输入是通过流水线获得的，那么在执行该操作的next()函数时将调用其输入操作的next()函数



[3] 来源: Andy Pevlo, CMU 15-445/645

迭代器模型的不足

- 数据以元组为单位进行处理，不利于高速缓存发挥作用
- 每处理一条元组需要多次调用`next()`函数，而`next()`为虚函数，调用开销大
- 实验发现，`open()`、`next()`和`close()`的运算逻辑(即真正的查询执行过程)所花费的时间只占查询执行总时间的10%

Optimization of Query Execution

查询执行优化

查询执行优化: 利用计算机系统的特性来提高查询计划的执行效率

- 方法1: 编译执行(Compiled Execution)
- 方法2: 向量化执行(Vectorized Execution)

查询执行的优化(第9章)≠查询优化(第10章)

*openGauss*采用了编译执行和向量化执行

当代CPU的特性

当代CPU的特性是数据库查询执行优化的背后驱动力

- 超标量流水线与乱序执行
- 分支预测
- 多级存储与数据预取
- 单指令多数据流(SIMD)

超标量流水线与乱序执行

- CPU指令的执行可以分为多个阶段，如取址、译码、取数、运算等
- 流水线: 一套控制单元可以同时执行多条指令，但每条指令所处的执行阶段不同，例如上一条指令处理到了取数阶段，下一条指令处理到了译码阶段
- 超标量: 一个CPU核中有多套控制单元，可以同时并发执行多个流水线
- 乱序执行: CPU维护了一个可乱序执行的指令窗口，窗口中无数据依赖的指令可以被取来并发执行

分支预测

跳转指令

- 程序分支越少，流水线效率越高，但程序分支不可避免
- 当执行一个跳转指令时，在得到跳转的目的地址之前，不知道该从哪里取下一条指令，流水线只能空闲等待

分支预测

- CPU引入了一组寄存器，专门用来记录最近几次某个地址的跳转指令的目的地址
- 当再次执行到这个跳转指令时，就直接从上一次保存的目的地址出取指令，放入流水线
- 待获取到真正的目的地址时，如果预测的指令取错了，则抛弃当前流水线中的指令，取真正的指令进行执行

单指令多数据流(SIMD)

- 计算密集型程序通常会对大量不同的数据执行相同的运算
- SIMD引入了一组大容量寄存器，每个寄存器包含8*32位，可同时存储8个32位数据
- CPU新增了处理这种8*32位寄存器的SIMD指令，可以在一个指令周期内完成对寄存器内8个数据的相同运算

编译执行(Compiled Execution)

拉取式(pull-based)执行模型

- 火山模型采用了拉取式执行模型
- 大量虚函数调用导致性能损失

推送式(push-based)执行模型

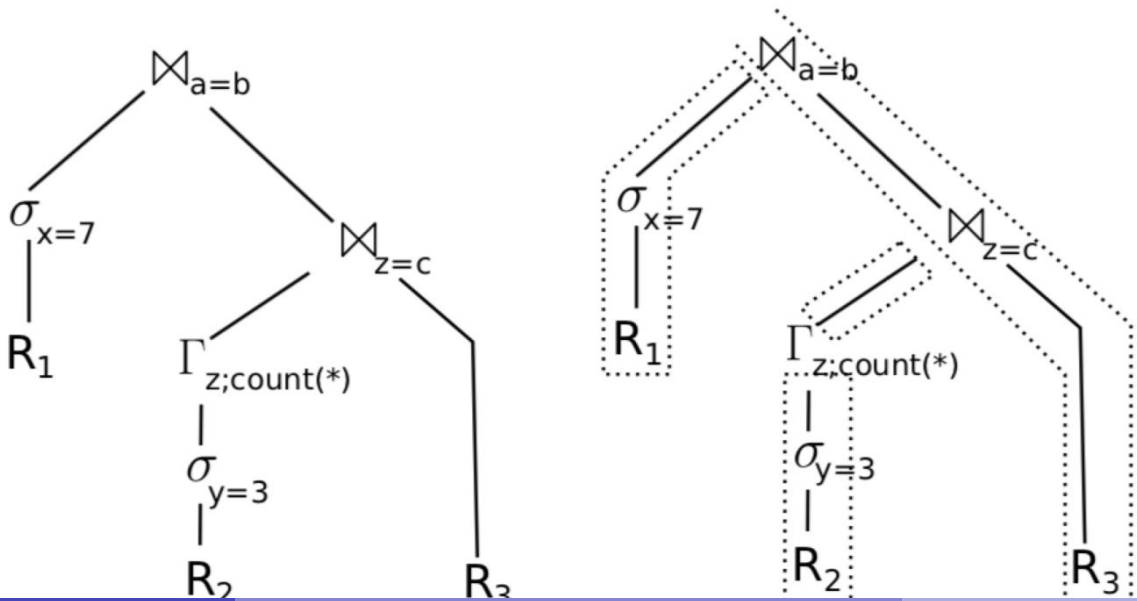
- 编译执行采用了推送式执行模型
- 与拉取式模型相反，推送式模型自低向上执行，执行逻辑由最底层operator开始，处理完一条元组之后，将元组传给上层operator继续进行处理

编译执行

- 编译执行方法将查询计划划分为多个流水线
- 在每个流水线内，数据可以一直留在寄存器中

Example (编译执行的流水线)

```
SELECT * FROM R1, R3,
  (SELECT R2.z, COUNT(*) FROM R2
   WHERE R2.y = 3
   GROUP BY R2.z) R2
WHERE R1.x = 7 AND R1.a = R3.b AND R2.z = R3.c;
```



编译执行

前面的查询计划对应的编译执行的伪代码[4]

- 每个流水线对应一个for循环
- 一次for循环处理一条元组，元组在一次for循环内不离开寄存器
- 编译执行的伪代码以数据为中心，消除了火山模型中大量的虚函数调用

initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z
for each tuple t in R_1
  if $t.x = 7$
    materialize t in hash table of $\bowtie_{a=b}$
for each tuple t in R_2
  if $t.y = 3$
    aggregate t in hash table of Γ_z
for each tuple t in Γ_z
  materialize t in hash table of $\bowtie_{z=c}$
for each tuple t_3 in R_3
  for each match t_2 in $\bowtie_{z=c}[t_3.c]$
    for each match t_1 in $\bowtie_{a=b}[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$

知乎 @群演

[4] https://zhuanlan.zhihu.com/p/63996040

向量化执行(Vectorized Execution)

- 向量化执行仍然采用拉取式执行模型
- 区别在于每个迭代器的next()函数每次返回一批数据(如1000行)，而不是一条元组

向量化执行的优点

- 减少了火山模型中虚函数调用的数量
- 以块为单位处理数据，提高了cache命中率
- 多行并发处理，符合当代CPU乱序执行与并发执行的特性
- 同时处理多行数据，使SIMD有了用武之地

总结

1. Overview
2. External Sort
   - External Merge Sort
3. Execution of Relational Algebraic Operations
   - Execution of Selection Operations
   - Execution of Projection Operations
   - Execution of Duplicate Elimination Operations
   - Execution of Aggregation Operations
   - Execution of Set Operations
   - Execution of Join Operations
4. Execution of Expressions
   - Materialization
   - Piplining
5. Optimization of Query Execution

习题

1. Describe the *one-pass aggregation algorithm* and analyze its I/O cost and memory requirement
2. Describe the *hash-based aggregation algorithm* and analyze its I/O cost and memory requirement
3. Describe the *sort-based aggregation algorithm* and analyze its I/O cost and memory requirement
4. Write pseudocode for an iterator that implements a join algorithm (*one-pass join*, *block-based nested loop join*, *sort-merge join*, *hash join*, *index-based join*)