

第七章 查询处理

章成源

湖南大学-信息科学与工程学院-计算机科学系

办公室：院楼403

Email: cyzhangcse@hnu.edu.cn

第七章 查询处理

7.1 查询处理概述

7.2 查询编译

7.3 物理操作符

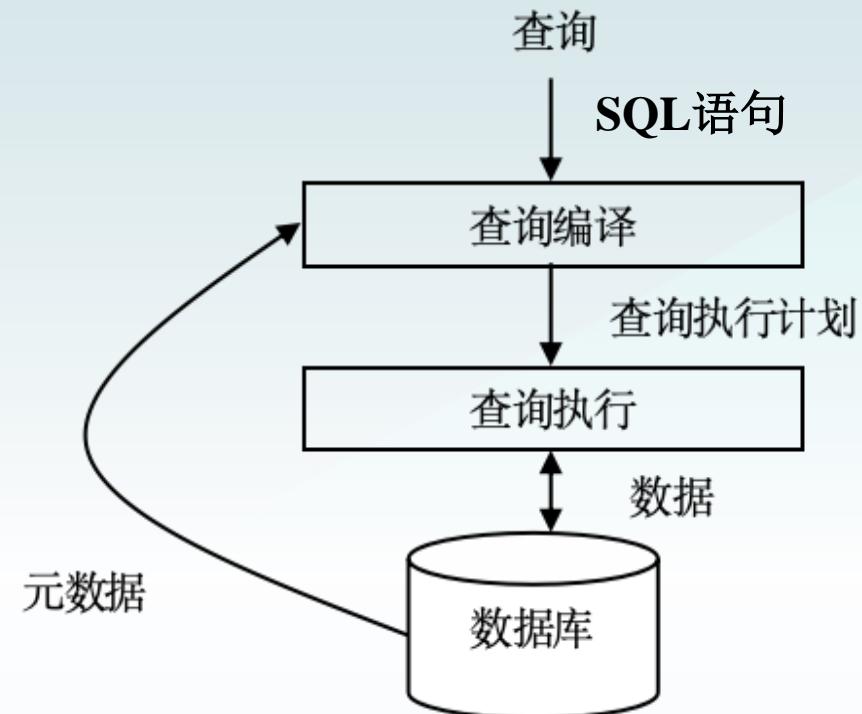
7.4 小结

7.1 查询处理概述

- **查询处理器**
 - DBMS中的一个部件集合，它能够将用户的SQL命令转变成数据库上的操作序列，并且执行这些操作，通常也被称为SQL引擎。
- **SQL引擎功能**
 - 将SQL语言表示的查询语句翻译成能在文件系统的物理层上使用的表达式；
 - 为优化查询进行各种转换；
 - 查询的实际执行。

7.1 查询处理概述

- 查询处理器的主要部分：
 - 查询编译
 - 查询执行
- 查询编译阶段
 - 输入：SQL语句的字符串
 - 输出：查询执行计划树
→交给查询执行器执行。



7.1 查询处理概述

- SQL语句的分类：
 - DCL/DDL语句
 - 定义或修改数据库的型或状态
 - DML语句
 - 修改数据库的值
 - Insert语句
 - Update和Delete语句
 - DQL语句（本章重点讲解）
 - 非过程化
 - Select语句

第七章 查询处理

7.1 查询处理概述

7.2 查询编译

7.3 物理操作符

7.4 小结

第七章 查询处理

7.2.1 查询编译概述

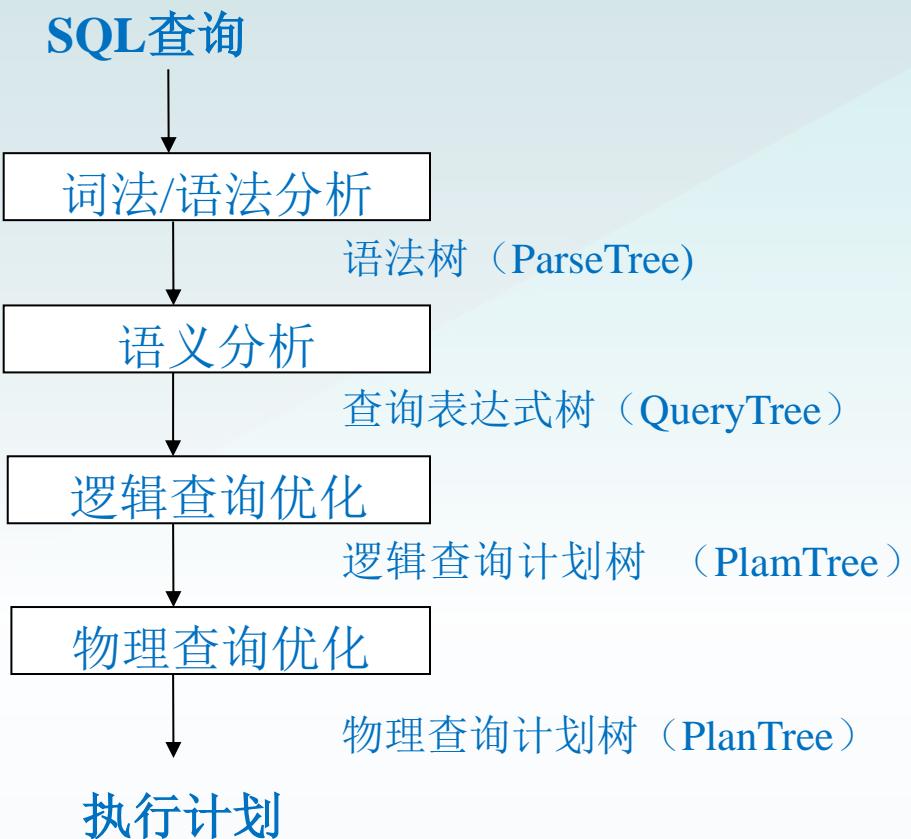
7.2.2 词法与语法分析

7.2.3 语义分析

7.2.4 查询优化

7.2.1 查询编译概述

- **查询编译器：**
 - 输入：SQL语句（字符串）
 - 输出：查询执行计划（查询计划树/执行树）
- **查询编译步骤：**
 - 1) **查询解析**
 - 词法/语法分析
 - 语义分析
 - 2) **查询优化**
 - 逻辑查询优化
 - 物理查询优化



7.2.2 词法与语法分析

- 词法分析

根据预定义的模式对SQL字符序列进行模式匹配，从查询语句中识别出正确的语言符号。

- 语法分析

进行语法检查，从给定模式序列输入中寻找某一特定语法结构，并按照给定的规则生成语法树

7.2.2 词法与语法分析

- 语法分析示例

simple_select:

```
SELECT target_list  
FROM from_list  
WHERE bool_expr  
GROUP BY group_by_list  
HAVING bool_expr
```

注：大写单词为关键字，小写单词为语法结构

7.2.2 词法与语法分析

- SELECT语句中的目标列是由一个或多个由逗号分隔的列名组成:
 - target_list:
 - col_name
 - | target_list ',' col_name
- FROM子句由一个或多个由逗号分隔的表名组成:
 - from_list:
 - table_name
 - | from_list ',' table_name
- GROUP BY子句由一个或多个由逗号分隔的列名组成:
 - group_by_list:
 - col_name
 - | group_by_list ',' col_name

7.2.2 词法与语法分析

- WHERE子句和HAVING子句由布尔表达式组成:

bool_expr:

bool_expr AND bool_expr
| bool_expr OR bool_expr
| col_name op const
| col_name IN '(' simple_select ')'

op: '>' | '<' | '='

const: ICONST | FCONST | SCONST

col_name: IDENTIFIER

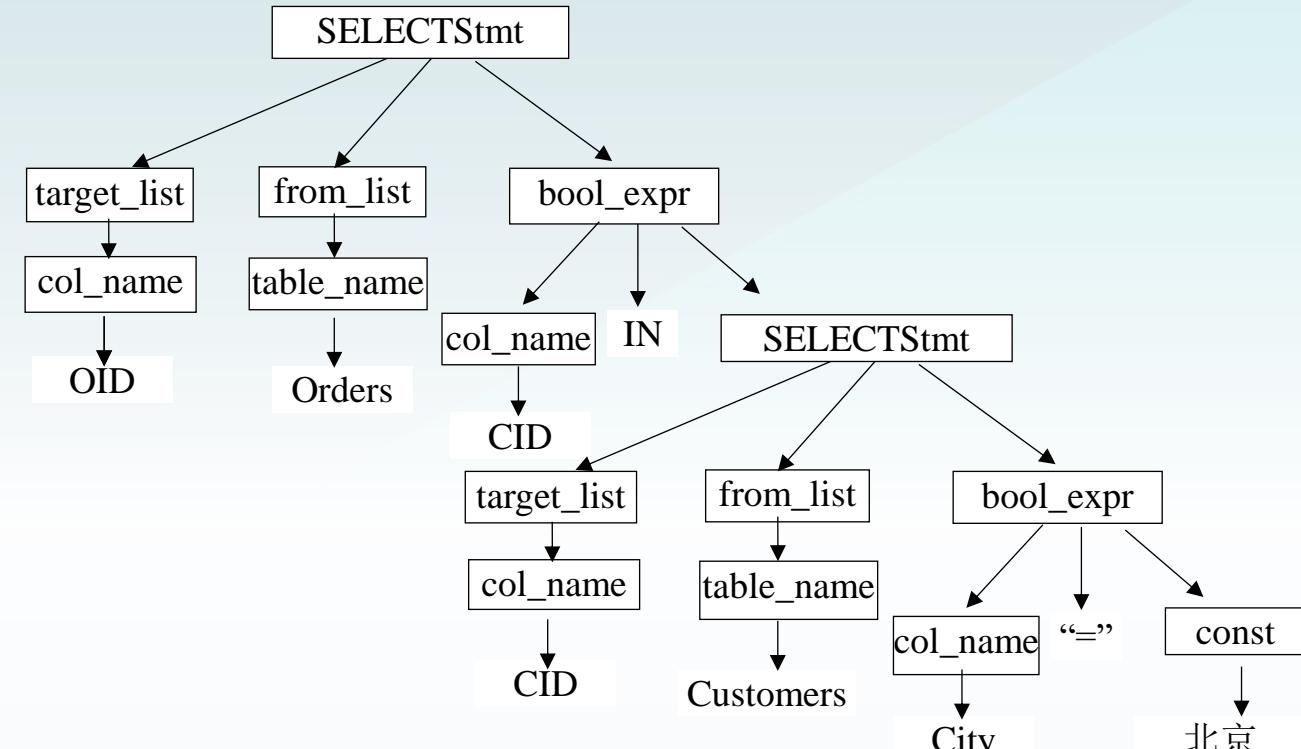
table_name: IDENTIFIER

7.2.2 词法与语法分析

- 例：

查询北京顾客的订单号

```
SELECT OID  
FROM Orders  
WHERE CID IN (  
    SELECT CID  
    FROM Customers  
    WHERE City = '北京');
```



语法树

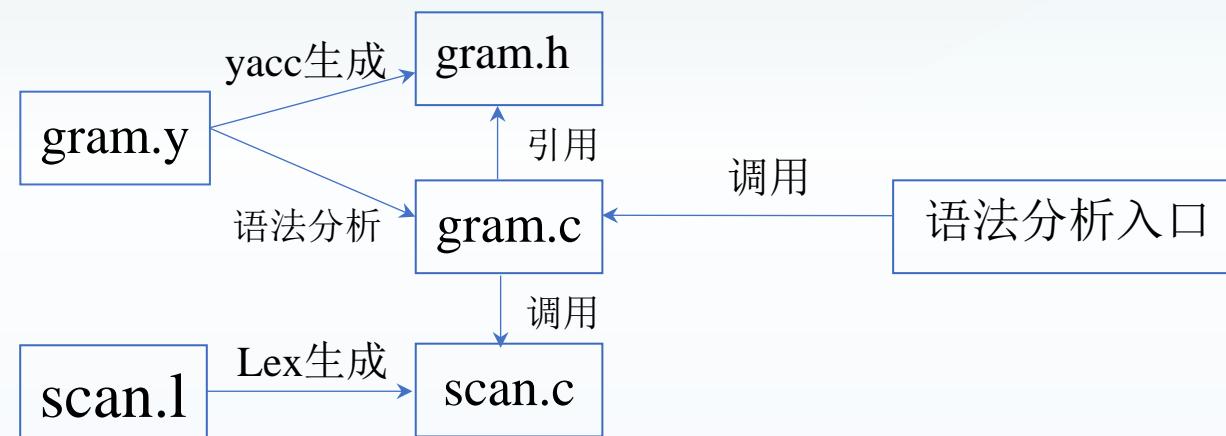
7.2.2 词法与语法分析

- LEX: 词法分析器的生成工具

正则表达式以及相关的动作代码通常写在一个后缀为“.l”的文件中，称为Lex文件

- YACC: 语法分析程序的自动生成器

SQL语法规则和一些必要的代码通常写在一个后缀为“.y”的文件中，称为YACC文件



词法语法文件关系图

7.2.3 语义分析

- 语义分析主要工作：根据数据字典中的内容检查SQL语句的有效性
 - 审核用户存取权限
 - 数据完整性检查
 - 语义的正确性检查
- 语义分析进行有效性语义绑定：
 - 根据语法树的内容，使用关系代数表达式来构造查询表达式树，体现查询的语义和结构。

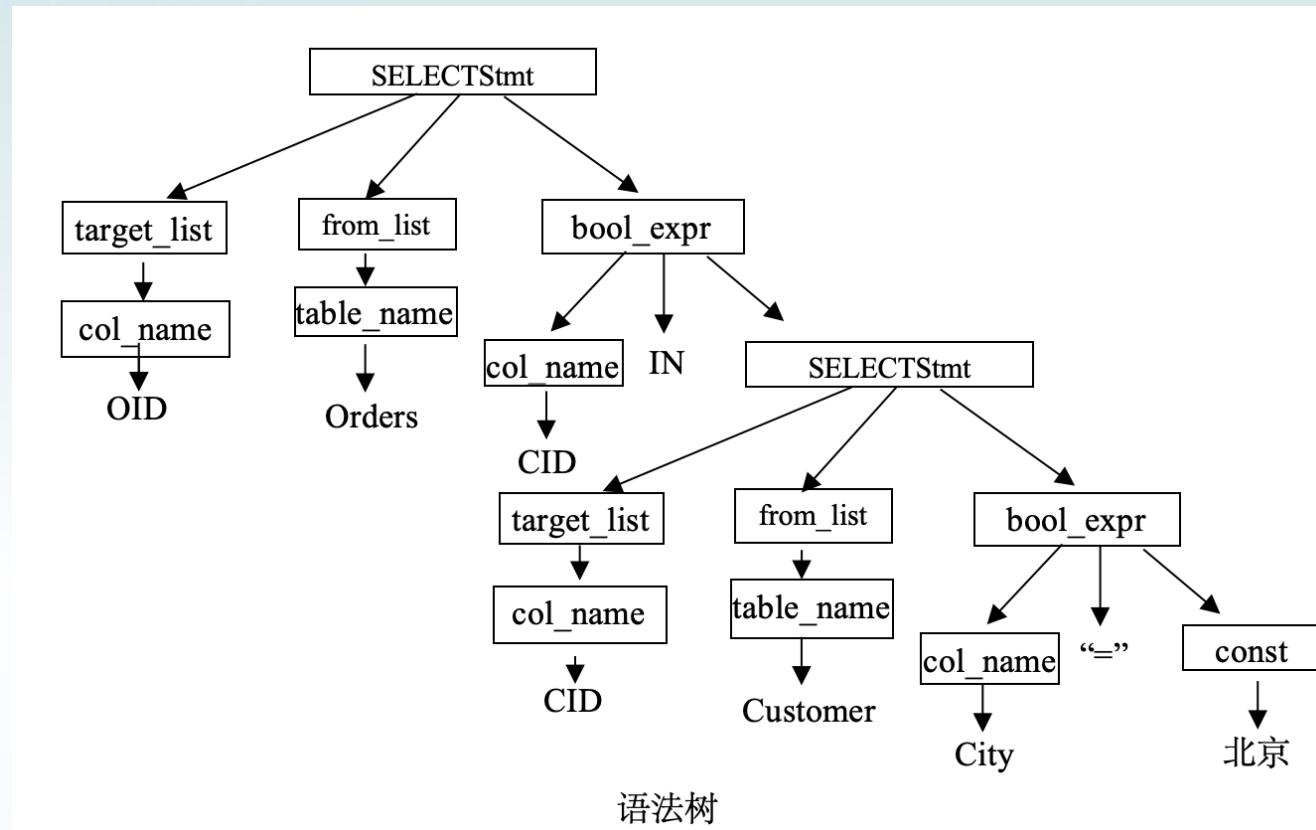
7.2.3 语义分析

- 查询表达式树的基本动作通常由关系代数操作符进行表达

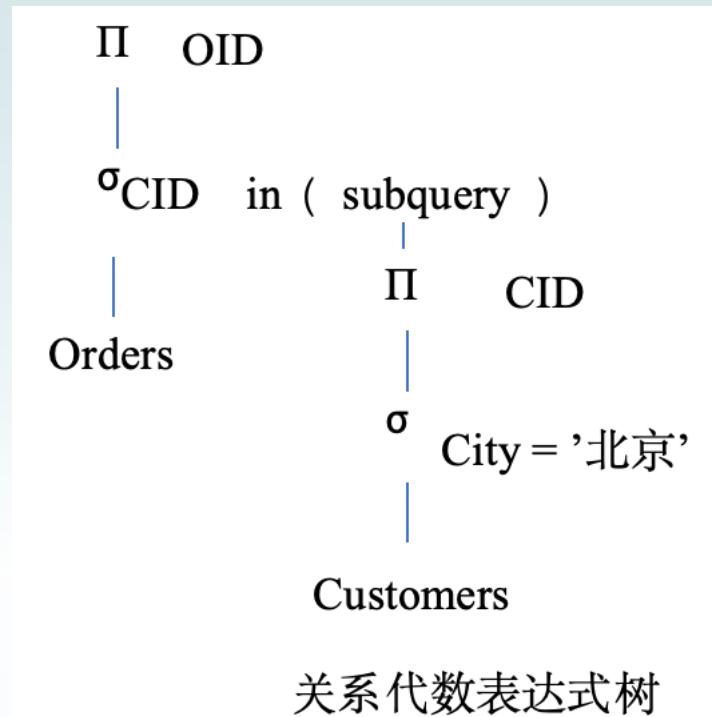
| SQL 查询子句 | 关系代数操作符 |
|----------------------------|-----------------------------------|
| SELECT | 投影 (Π) |
| FROM | 卡氏积 (\times) |
| WHERE | 选择 (σ) |
| JOIN、NATURAL JOIN | 连接 (\bowtie) |
| FULL/LEFT/RIGHT OUTER JOIN | 外连接 ($\bowtie \bowtie \bowtie$) |
| UNION | 并 (\cup) |
| INTERSECT | 交 (\cap) |
| EXCEPT | 差 (-) |
| DISTINCT | 去重 (δ) |
| GROUP BY | 分组聚集 (g) |
| ORDER BY | 排序 (τ) |

7.2.3 语义分析

- 示例：语法树经语义分析生成关系代数表达式树



语义分析
→



7.2.3 语义分析

- **查询重写**

查询使用视图时，依据数据库的规则系统对查询树进行转换，根据视图定义将其替换成对基表的操作。

- 例：系统有视图cust_orderinfo，可以查询每个顾客的订单数量：
- CREATE VIEW cust_orderinfo(v_CID, ordercount) AS
- SELECT CID, count(OID) AS ordercount
- FROM Orders
- GROUP BY CID;、
- 现查询每个顾客的姓名、城市和他的订单个数

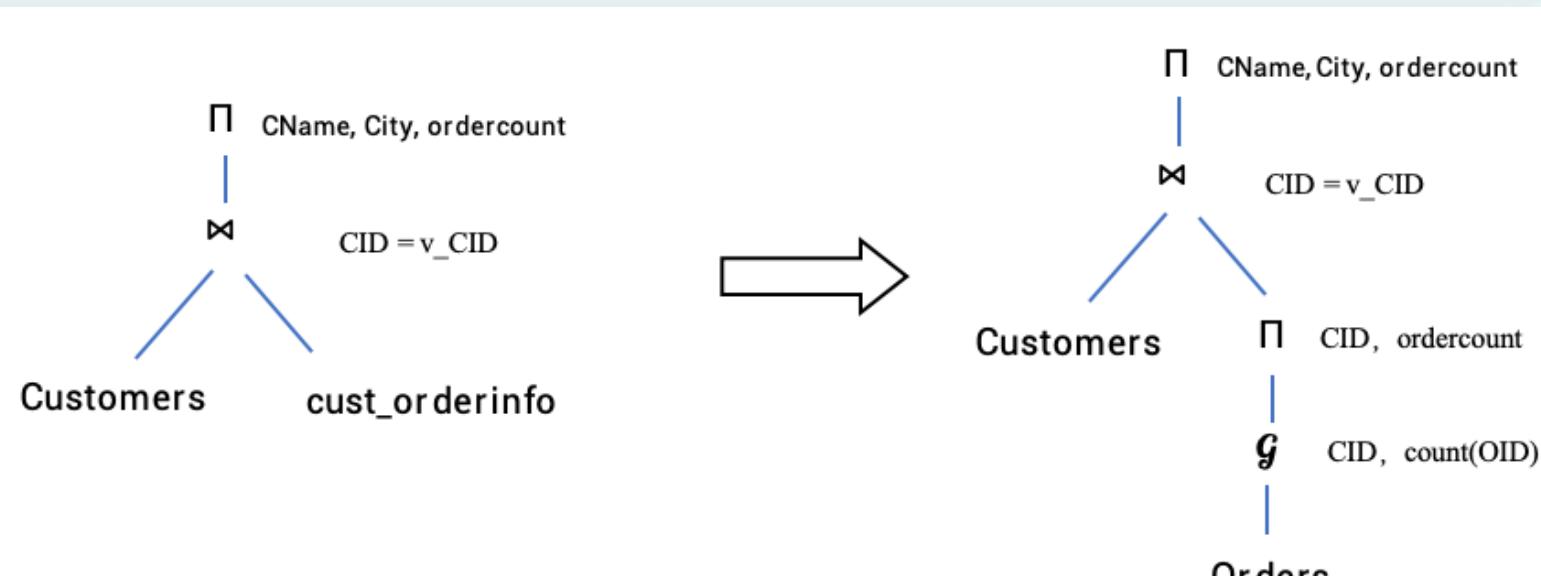
```
SELECT CName, City, ordercount  
      FROM Customers JOIN cust_orderinfo V ON customer.CID= V.v_CID
```

7.2.3 语义分析

- 查询重写（续）
- 查询每个顾客的姓名、城市和他的订单个数

SELECT CName, City, ordercount

FROM Customers JOIN cust_orderinfo V ON customer.CID= V.v_CID



视图的查询重写

7.2.4 查询优化

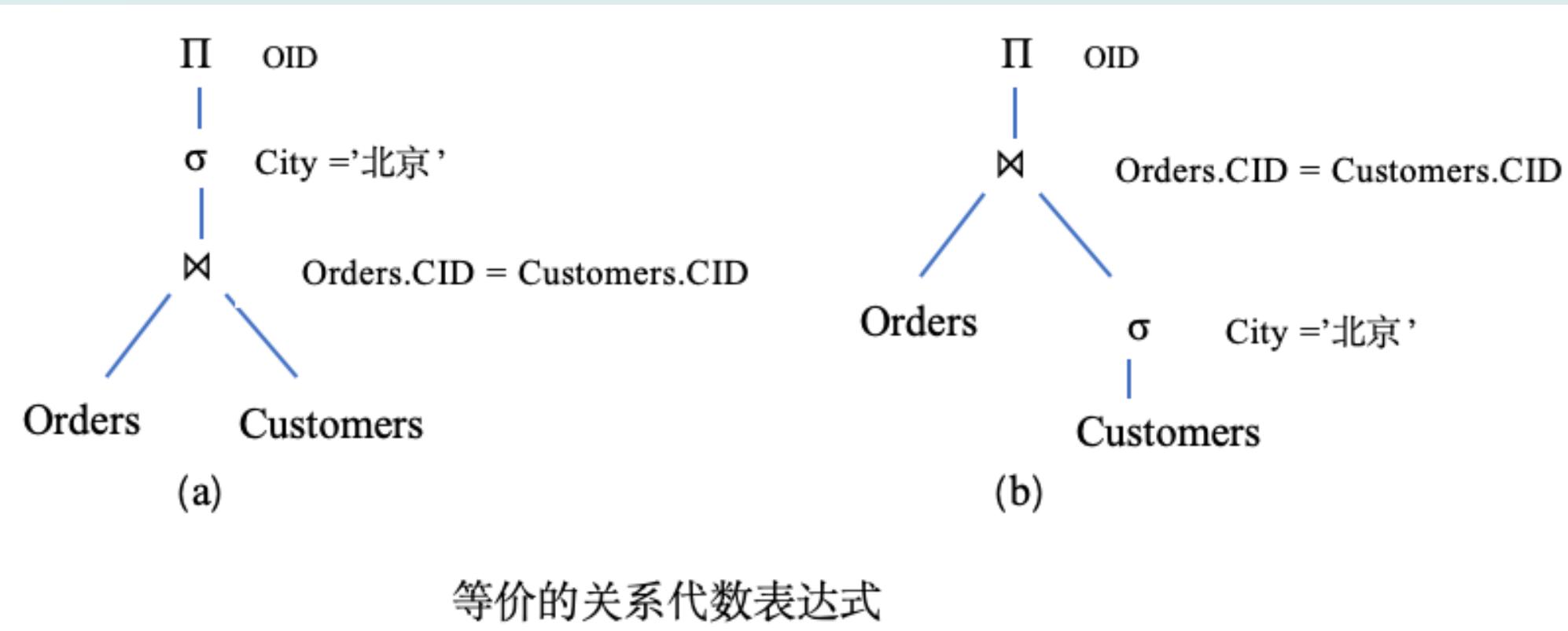
- 查询优化：为查询选择更高效（或代价更小）的查询执行计划的过程
 - 输入：查询表达式树
 - 输出：查询执行计划

7.2.4 查询优化

- 查询优化分类
 - 代数优化/逻辑优化：指关系代数表达式的优化
 - 物理优化：为关系运算符选择高效合理的存取路径或运算方法
 - 存取路径
 - 多表连接顺序、表连接算法
 - 操作结果是否排序
 - 数据传递方式

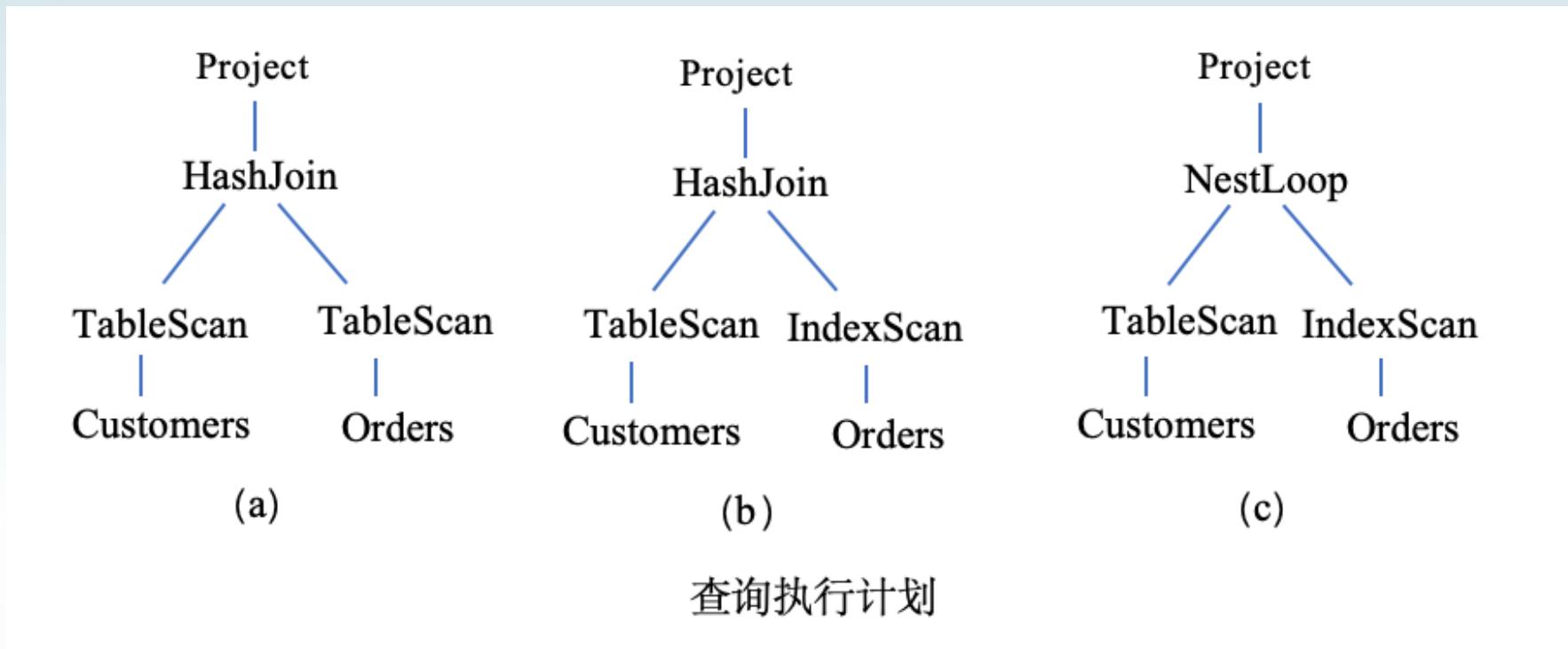
7.2.4 查询优化

- 逻辑查询优化：转化为预期执行时间较少的等价的查询分析树
 - 例：选择操作下推



7.2.4 查询优化

- 根据不同的表连接操作算法和扫描方式生成多个物理执行计划



- 查询优化器根据代价模型计算每种物理执行计划的代价；
- 选取代价最小的执行计划作为最终的查询执行计划交给查询执行器。

第七章 查询处理

7.1 查询处理概述

7.2 查询编译

7.3 物理操作符

7.4 小结

第七章 查询处理

- 7.3.1 物理操作符的代价模型
- 7.3.2 扫描操作
- 7.3.3 排序操作
- 7.3.4 连接操作
- 7.3.5 去除重复值
- 7.3.6 分组聚集
- 7.3.7 集合操作

7.3 物理操作符

- 一元操作符
 - 扫描、聚集、去重、排序
- 二元操作符
 - 连接、集合

7.3.1 物理操作符的代价模型

- 物理操作符的执行代价以不同资源的形式进行度量：
 - 磁盘存取
 - 执行一个查询所用的CPU时间，
 - 通信代价（并行或分布式数据库系统）

7.3.1 物理操作符的代价模型

- 代价模型
 - 模型中所有操作的代价都被量化，并且各种代价都能统一成相同的单位，使得执行计划的代价是可比较的。
 - 本章重点讲述物理操作符的算法，为简化起见不包括CPU代价，使用磁盘I/O的数量作为衡量每个操作代价的标准。

7.3.1 物理操作符的代价模型

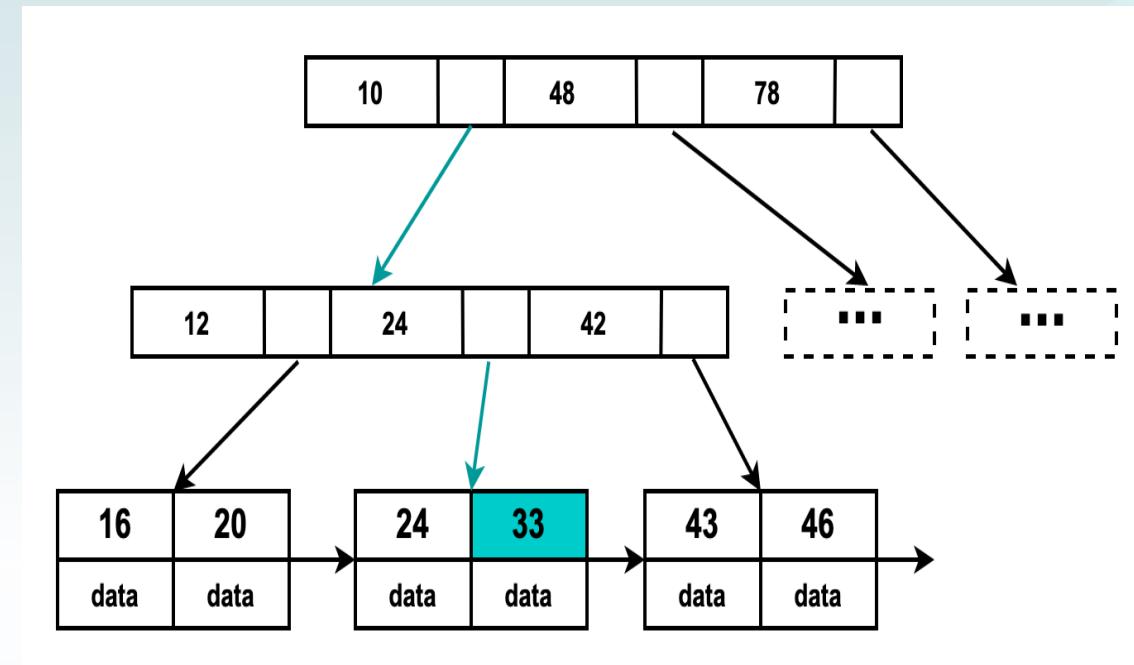
- 计算操作符代价的参数
 - 参数1：可以使用的内存缓冲块个数 M 。
 - 参数2：关系 R 的元组占用磁盘块数 $B(R)$ 。
 - 参数3：关系 R 中的元组数目 $T(R)$ 。
 - 参数4：关系 R 中的元组长度 $S(R)$ 。
 - 参数5：关系 R 中某属性 a 不同值的个数 $V(R,a)$ 。
- 关系中数据的多少及其分布常被定期计算并存储于数据字典中，用于计算操作符的代价

7.3.2 扫描操作

- **全表扫描:** 读取表中所有数据块，访问表中的所有行，每一行都要经WHERE子句判断是否满足检索条件；关系R上全表扫描的代价= $B(R)$ 。
- 全表扫描使用场景：
 - 没有可用的索引；
 - 查询优化器认为查询将会访问表中绝大多数的数据块；
 - 表的数据块只需要一次I/O 就能扫完。

7.3.2 扫描操作

- 索引扫描（IndexScan）：先在索引上找出满足条件的元组的物理位置，再根据其物理位置直接找到表中相应的元组。
 - 有序的索引，例如B+树索引，索引扫描后的元组是有序的
 - 可用于实现范围查询；
 - 索引扫描的代价：扫描索引读取的页面数+扫描表读取的页面数。



B+树索引

7.3.2 扫描操作

- 索引扫描（续）
 - 特别地，以下情况不需要到表中获取满足条件的元组：
 - 聚集索引：表中数据直接在索引的叶子节点上
 - 索引覆盖查询（cover index query）：索引的列都是索引的码

7.3.2 扫描操作

- 索引选择的基本原则：使用选择率较低的索引

例：假设Customers表中有10000条元组，分别在表的Age和City字段上有索引，则对于下面的查询，应该使用哪个索引呢？

```
SELECT * FROM Customers WHERE Age < 30 AND City = '北京';
```

- 考虑下面两个场景：

场景一：表中90%的顾客都在30岁以下，北京的顾客20%。

场景二：表中10%的顾客在30岁以下，北京的顾客50%。

显然，对于场景一应该使用City字段上的索引，场景二则使用Age字段上的索引。

7.3.2 扫描操作

- 位图扫描（BitmapScan）
 - 扫描索引，找出所有满足条件的元组位置。
 - 对元组位置进行排序，然后统一到表中取元组

7.3.2 扫描操作

例： **SELECT * FROM Customers WHERE Age = 30 AND City = '北京';**

- (1) 先使用Age列上的索引，找出满足Age = 30条件的元组地址集合A。
- (2) 再使用City列上的索引，找出满足City = '北京'条件的元组地址集合B。
- (3) 求集合A和集合B的交集C。
- (4) 对集合C中的元组地址排序，然后去表中访问元组，输出满足条件的元组。

7.3.3 排序操作

- 内存排序算法

适用于需要排序的数据集小于可利用的内存

- 快速排序算法

- 外部排序算法

适用于数据集无法全部放入内存时

- 两阶段多路归并排序(Two-Phase, Multiway Merge-Sort, TPMMS)算法

7.3.3 排序操作

- 假设有M个缓冲块。TPMMS算法步骤如下：
 - 阶段1：操作对象的数据被读入内存，以某种方式排序，并再次写回磁盘形成N个长度为M块的有序子表；
 - 阶段2：重新读取磁盘上所有子表并通过某种方式“归并”（如多路归并算法）以完成排序操作。

7.3.3 排序操作

- 例：初始关系有20个元组，假设有内存可以使用6个缓存块，每个缓存块上有一个数据项，则阶段1把初始关系分成4个有序子表

初始关系

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 1 | j | p | b | o | a | i | e | q | d | t | g | s | k | c | r | m | f | h | 0 |

阶段 1: 形成 4 个有序的子表

| | |
|---|---|
| 6 | a |
| 4 | b |
| 2 | j |
| 1 | l |
| 5 | o |
| 3 | p |

| | |
|----|---|
| 10 | d |
| 8 | e |
| 12 | g |
| 7 | i |
| 9 | q |
| 11 | t |

| | |
|----|---|
| 15 | c |
| 18 | f |
| 14 | k |
| 17 | m |
| 16 | r |
| 13 | s |

| | |
|----|---|
| 19 | h |
| 20 | n |
| | |
| | |
| | |

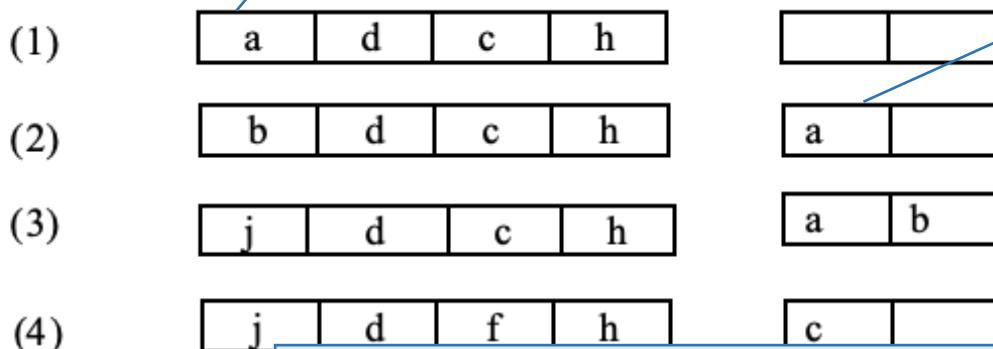
归并外排序阶段 1

7.3.3 排序操作

在阶段2，对4个有序子表进行分别读取4个有序子表中的第一个元组到输入缓冲块，2个缓冲块用作输出

分别读取4个有序子表中的第一个元组到输入缓冲块

阶段 2：归并排序，其中 4 块为输入缓冲块，2 块为输出缓冲块



每一次取其中的最小值放到输出缓冲块；

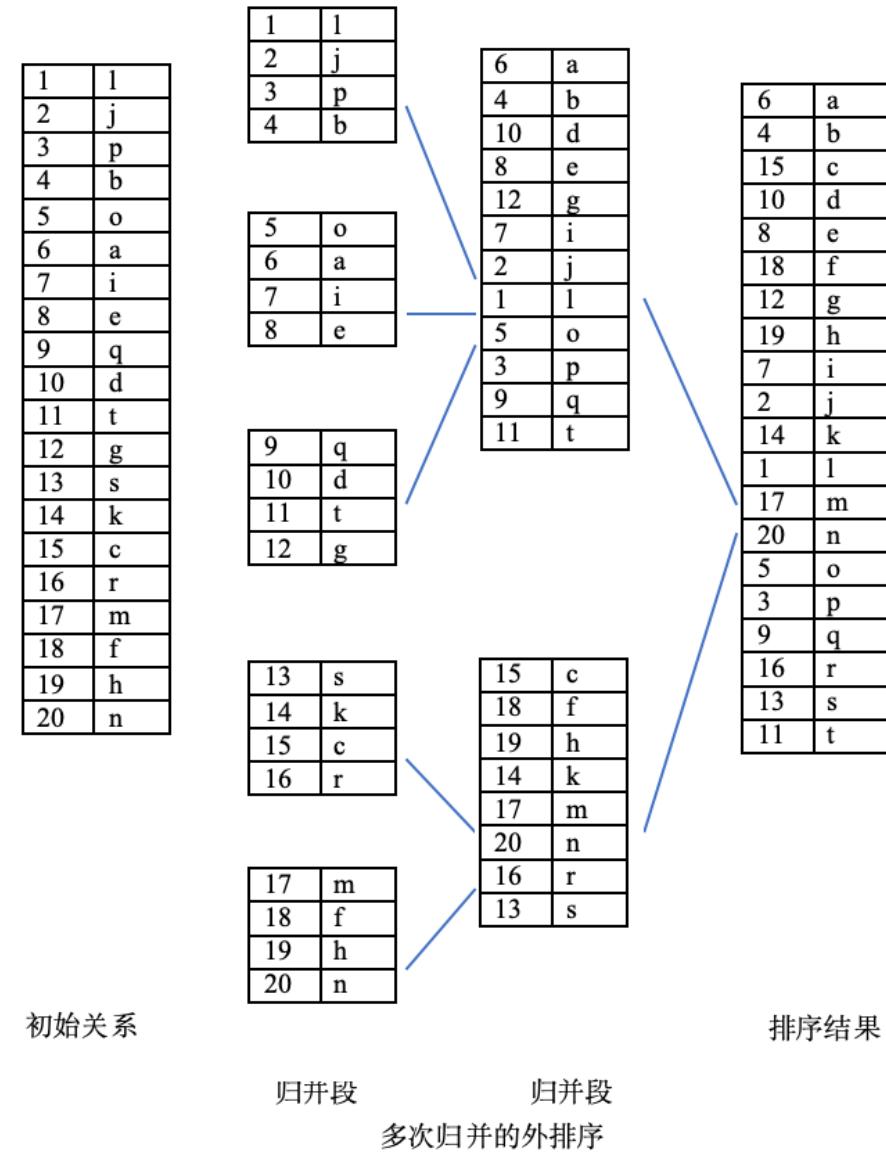
输出关系

若输出缓冲块满，则将其写入到输出关系中。

归并外排序阶段 2

7.3.3 排序操作

- 若关系非常大，阶段1可能产生多于M-1个有序子表，则归并操作就需分多趟才能完成。
- 例：如图，初始关系有20个元组，假设有内存可以使用4个缓存块，每个缓存块上有一个元素，则阶段1分成5个有序子表，在阶段2每次只能归并3个子表，则分两次归并。



7.3.3 排序操作

- 阶段1生成归并段的方法除快排以外，还可以采用替换算法。
 - (1) 将数据项读入内存，并在内存中组织成一个数据结构，可以有效支持插入和删除最小值的操作。
 - (2) 从该数据结构中找出最小值，移出到一个归并段文件，再从磁盘中读入一个数据项取代以前项空出的位置。

7.3.3 排序操作

- (3) 从该数据结构中再次找出最小值，如果该最小值比前一个移出的数据项大，则写入同一个归并段文件；如果新的最小值小，则开始一个新的段文件。
- (4) 重复(3)操作，直到处理完所有的数据项。

7.3.4 连接操作

- 嵌套循环连接（Nested Loop Join, NLJ）
 - 最直接和基础的连接算法，任何连接条件都可以使用该连接方式。
 - 给定两个关系R和S， $R \bowtie_{\theta} S$ 表示关系R和S的θ连接，其嵌套循环连接算法形式化表示如下：

```
FOR R中每个元组r DO      /*外表循环*/  
    FOR S中每个元组s DO /*对于外表的每个元组查找内表中与其匹配的所有元组*/  
        IF s能与r生成连接元组t THEN  
            输出满足θ连接条件的元组t;  
        END;  
    END;  
END;
```

7.3.4 连接操作

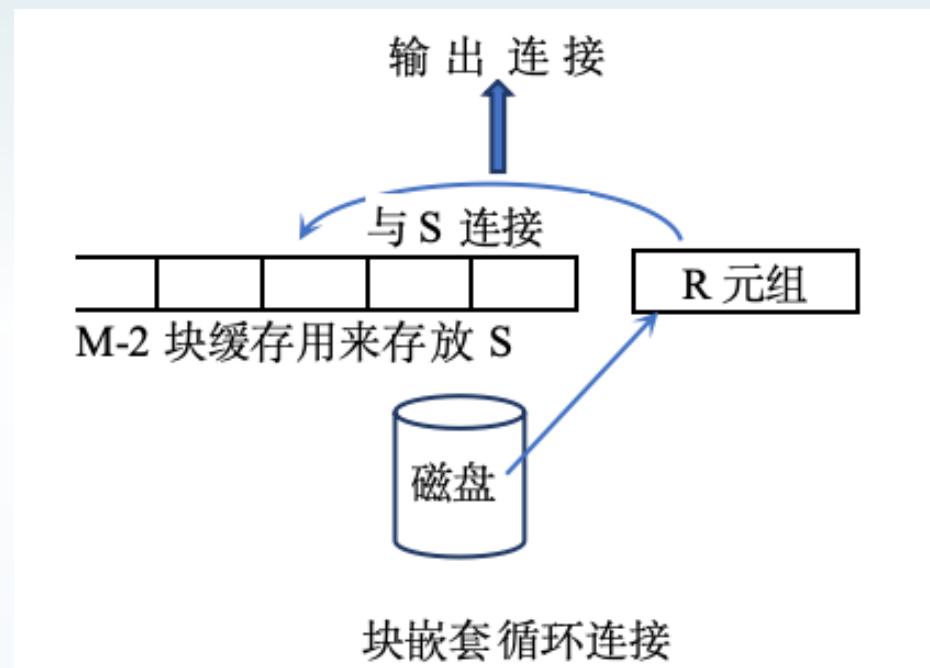
- 嵌套循环连接代价分析
 - 最差的情况: $B(R) + T(R)*B(S)$
 - 如果有一个关系可以放到内存中: $B(R) + B(S)$
 - 当被连接的数据子集较小时, 嵌套循环连接是个较好的选择, 且小表做内表

7.3.4 连接操作

- 块嵌套循环连接 (Block Nested Loop Join, BNLJ)
 - 假设系统可使用的缓冲区个数是M，且 $B(R) \geq B(S) \geq M$
 - 块嵌套循环连接算法的基本思想:选择较小的表S做为外表，把S拆分为能放入内存的多个子表（每次可以读入内存），让每个子表与R进行连接。

7.3.4 连接操作

- 块嵌套循环连接（Block Nested Loop Join, BNLJ）
 - 通常使用M-2块缓冲区存放外表S，使用最后一块缓冲区来存放R的数据块
 - 将R的该块中所有元组与S的内存中所有元组进行连接，输出满足连接条件的元组。



7.3.4 连接操作

```
/*外层循环：对S进行*/  
FOR S的M-2个数据块 DO BEGIN  
    将这些块读入缓冲区中；  
    将其元组组织成查找结构，查找关键字是连接属性；  
    FOR R的每个数据块r DO BEGIN /*内层：一次一块处理R*/  
        将r读入缓冲区；  
        FOR r中的每个元组t DO BEGIN /*处理当前块所有元组*/  
            找出S在内存中能与t进行连接的元组；  
            输出满足连接条件的元组；  
        END;  
    END;  
END;
```

7.3.4 连接操作

- 块嵌套循环连接代价分析
 - 磁盘I/O次数: $B(S) + B(S)/(M-2) * B(R)$
 - 若 $B(S) \leq M-1$: $B(S) + B(R)$
 - 思考: 若R或者S上建立索引呢?
 - 若内表的连接属性上有索引, 则对内表不需要全表扫描, 可以借助索引来查找S中与元组t满足连接条件的元组。
→索引嵌套循环连接

7.3.4 连接操作

- 索引嵌套循环连接 (Indexed Nested Loop Join, INLJ)
 - 内表的连接属性上有索引，用索引查找来替代文件扫描；
 - 对于外存关系R中的每个元组t,利用索引来查找S中与元组t满足连接条件的元组。
- 代价估算
 - 对于外表R中的每个元组，需要在S的索引上执行一次查找，并检索相关的元组
 - 假设一次索引扫描的代价为C, INLJ的总的代价： $B(R) + T(R)*C$

7.3.4 连接操作

- 索引嵌套循环连接算法（续）
 - 只需要在外表上循环一次，连接操作的执行高效。
 - 在做SQL语句优化时，不仅在选择条件上创建索引，在连接条件上更需要创建索引。
 - 如果两个关系R和S在连接属性上均有索引可用，通常使用小表作为外表，减少索引查找的次数。

7.3.4 连接操作

- 归并连接 (Merge Join , MJ)
 - 常用于自然连接或等值连接
 - 要求参与归并的关系数据在连接属性上是有序的。

7.3.4 连接操作

- 给定两个关系R和S, $R \text{ } R_{Ra=Sgb} \bowtie S$ 表示关系R和S 的等值连接
 - 阶段1（排序阶段）：若R和S没有顺序，则对R,S进行排序
 - 阶段2（归并阶段）：与排序算法中的归并非常相似，比较队列头上元组连接属性值。属性值小的元组出队，直到找到满足连接条件的元组，组合成结果元组。

7.3.4 连接操作

- 归并连接算法

pr := R的第一个元组的地址;

ps := S的第一个元组的地址;

```
WHILE (pr ≠ NULL and ps ≠ NULL) DO /*从两表的第一个元组开始查找*/
    IF pr(R).a > ps(S).b THEN ps++;
    ELSEIF pr(R).a < ps(S).b THEN pr++;
    ELSE /*比较队列头的元组连接属性值*/
        WHILE ((pr(R).a = ps(S).b) AND pr ≠ NULL) DO /*属性值小的元组出队,再使用下一个元组进行比较*/
            mark = ps;
            markvalue = pr(R).a;
            WHILE ((pr(R).a = ps(S).b) AND ps ≠ NULL) DO /*组合成结果元组并输出*/
                output pair pr(R), ps(S);
                ps++;
            END WHILE
            pr++;
            if(pr(R).a = markvalue) ps = mark;
        END WHILE
    END WHILE
END WHILE
```

7.3.4 连接操作

- 归并连接代价分析
 - 若S和R均已排序，则连接属性上具有相同值的元组逻辑上连续
 - 磁盘I/O: $B(S) + B(R)$
 - 注:如果R表在连接属性上有重复值，就可能存在S指针回退的情况。
因此，当外表具有相同连接属性值的时候，要求内表是可重复读的。

7.3.4 连接操作

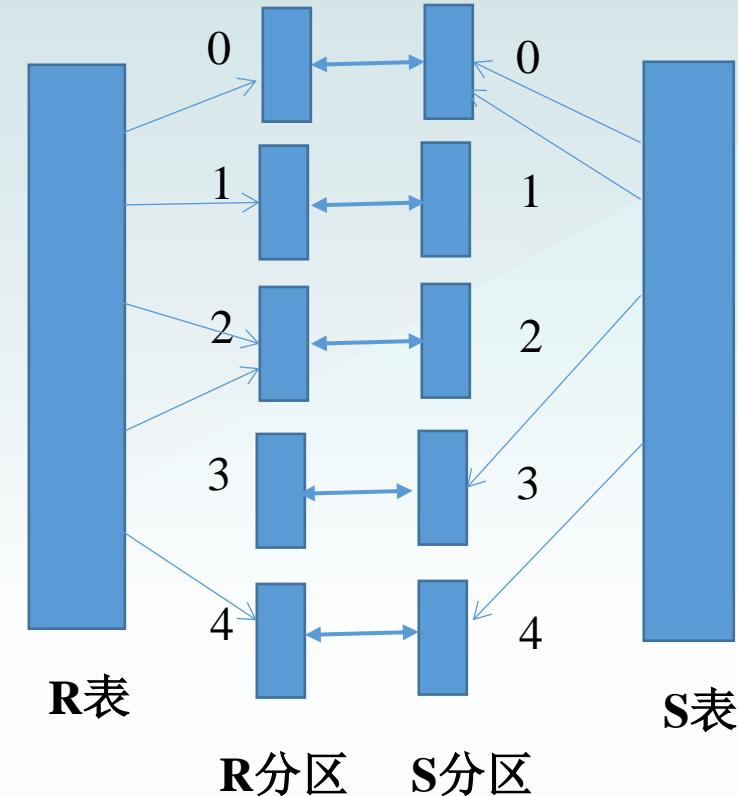
- 哈希连接（Hash Join , HJ）
 - 常用于自然连接或等值连接
- 算法基本思想
 - Hash函数可以根据连接属性把关系中的元组划分到不同的桶中
 - 满足连接条件的两个关系的元组必定在hash值相同的桶中
 - 不同哈希值的桶不需要参与连接匹配

7.3.4 连接操作

- 哈希连接（续）

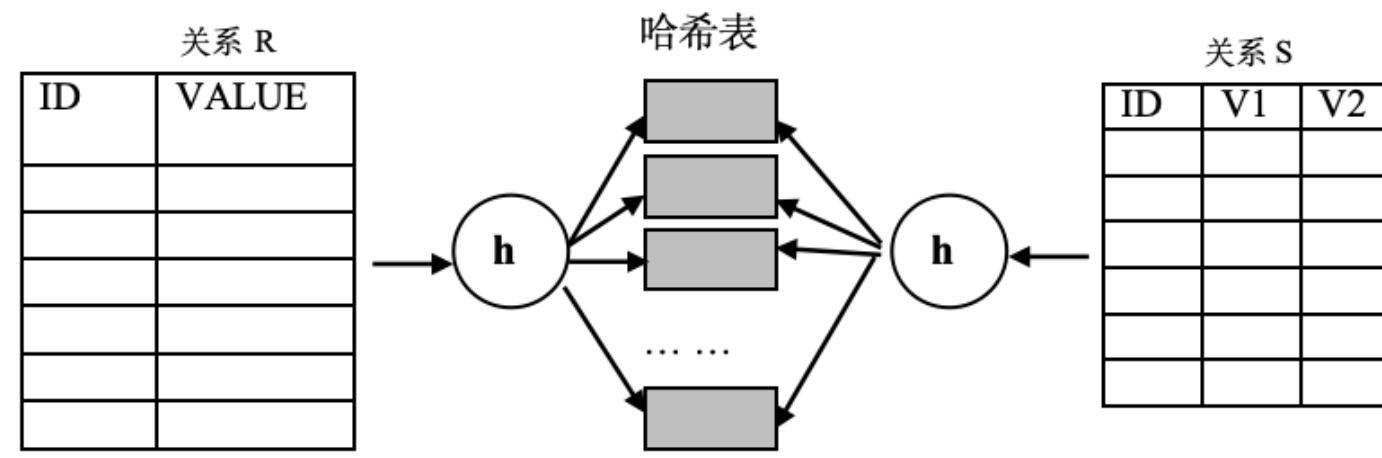
如果 r是Customers表上的元组， s是Orders表中的元组， h是元组属性CID上的哈希函数；

- 只有在 $h(r.CID)= h(s.CID)$ 时才需要比较r和s；
- 如果 $h(r. CID)\neq h(s. CID)$ ， 则r和s对于在SNO上的取值必不相等；
- 但如果 $h(r. CID)= h(s. CID)$ ， 还必须检查r和s在连接属性上的值是否相同， 因为不同的值可能会有相同的哈希值。



7.3.4 连接操作

- 哈希连接算法两阶段
 - 构造 (build) 阶段：对其中的一个表R的连接属性做为哈希键创建哈希表，称之为构建表；
 - 探测 (probe) 阶段：对另一个表（称为探测表）S的每个元组使用相同的哈希函数来检索和探测R中的元组是否在连接属性上匹配。



7.3.4 连接操作

- 哈希连接代价分析

- 若 $B(R) < M$ 或 $B(S) < M$, 其磁盘I/O的次数为 $B(R) + B(S)$
- 否则, 采取“分而治之”方式:

假设 h_1 是可以将连接属性的值映射到 K 个值 $\{a_1, a_2, \dots, a_k\}$ 的哈希函数, 使用该哈希函数将 R 表划分成 K 个分区 r_1, r_2, \dots, r_k , 将 S 表划分成 K 个分区 s_1, s_2, \dots, s_k , 则下面的表达式成立:

$$\begin{aligned} R_{R.a=S.b} S \\ = (R_1 \underset{R1.a=S1.b}{\bowtie} S_1) \cup (R_2 \underset{R2.a=S2.b}{\bowtie} S_2) \cup \dots \cup (R_k \underset{Rk.a=Sk.b}{\bowtie} S_k) \end{aligned}$$

7.3.4 连接操作

- Grace Hash Join (GHJ)
 - 阶段一：使用哈希函数分别把R和S划分成K个桶；
 - 阶段二：对于每对 r_i 和 s_i ，进行连接操作，然后合并结果集
- 代价分析
 - 阶段一，关系R和S需要一次完整的读入以及写回： $2*(B(S)+B(R))$
 - 阶段二，每个分区需要再次读入： $B(S)+B(R)$
 - 因此，总的磁盘I/O次数： $3*(B(S)+B(R))$

7.3.4 连接操作

- Grace Hash Join (续)

思考：使用哈希函数分别把R和S划分成K个桶，那么K值取多大合适？

假设 $B(R) > B(S)$, S作为内表构建哈希表：

- 若K足够大，则每个 s_i 都可以在内存中构建；
- 若 $K >$ 内存可用的缓冲块，则对R或S的分区不能一次扫描完成，如何解决？

→递归分区（recursive partitioning）

7.3.4 连接操作

- 递归分区（recursive partitioning）
 - K的值大于等于内存块数，完成关系的分区需要多趟，每趟能划分的最多分区数是可用于输出的缓存块的个数。
 - 每趟生成的分区在下一趟中作为输入使用不同的哈希函数继续进行分区，直到用于构造哈希表的分区可以放在内存中。

7.3.4 连接操作

- 数据偏斜 (data skew)
 - 当关系S中有多个元组在连接属性上取相同值或哈希函数不符合随机性和均匀性时，则元组在分区间分布就会不均匀，使得某些分区的元组数会远多于平均数。
 - 数据偏斜的分区在构建哈希表时会导致溢出现象。

7.3.4 连接操作

- 规避哈希表溢出问题的方法：
 - 适当增加分区的个数，可以处理少量的偏斜，增加的数量称为避让因子（fudge factor），通常取20%。
 - 溢出分解（overflow resolution）：对于发生偏斜分区，使用不同的哈希函数将其进一步划分成更小的分区。同理，对应的另一张表也要做相同的划分。

7.3.4 连接操作

- 规避哈希表溢出问题的方法：
 - 溢出避免 (overflow avoidance)：采用保守的策略，规避哈希表溢出，例如，首先将构造关系S划分成更多小的分区，然后把某些分区合并但是保证每个合并后的分区都能放到内存中。

若关系S中有多个元组在连接属性上取相同值，则上述方法可能失效，此时可选择采用其他连接算法。

7.3.4 连接操作

- 混合哈希连接（Hybrid Hash Join, HHJ）
 - 当内存规模相对较大但还不足存放整个构造关系时可以采用。
 - 基本思想：

对S表进行hash分区时，把S1留在内存中，则当系统对R表进行分区时，对于R1中的元组直接可以进行连接操作输出结果

7.3.5 去除重复值

- 需要去除重复值的情况：
 - distinct: SQL中的distinct关键字
 - union: 集合的并操作

7.3.5 去除重复值

- 去重方法
 - 排序：排序后相同元组相互邻近，删除重复元组只留下一个即可。对于外部归并排序，在创建归并段时即可发现部分重复元组并在写回磁盘前删除，其余可以在归并过程中去除。
 - 哈希：使用哈希函数对关系进行分区，对每个分区创建内存哈希表，在创建哈希表时，只有不在哈希表的元组才插入，否则，该元组丢弃。

7.3.5 去除重复值

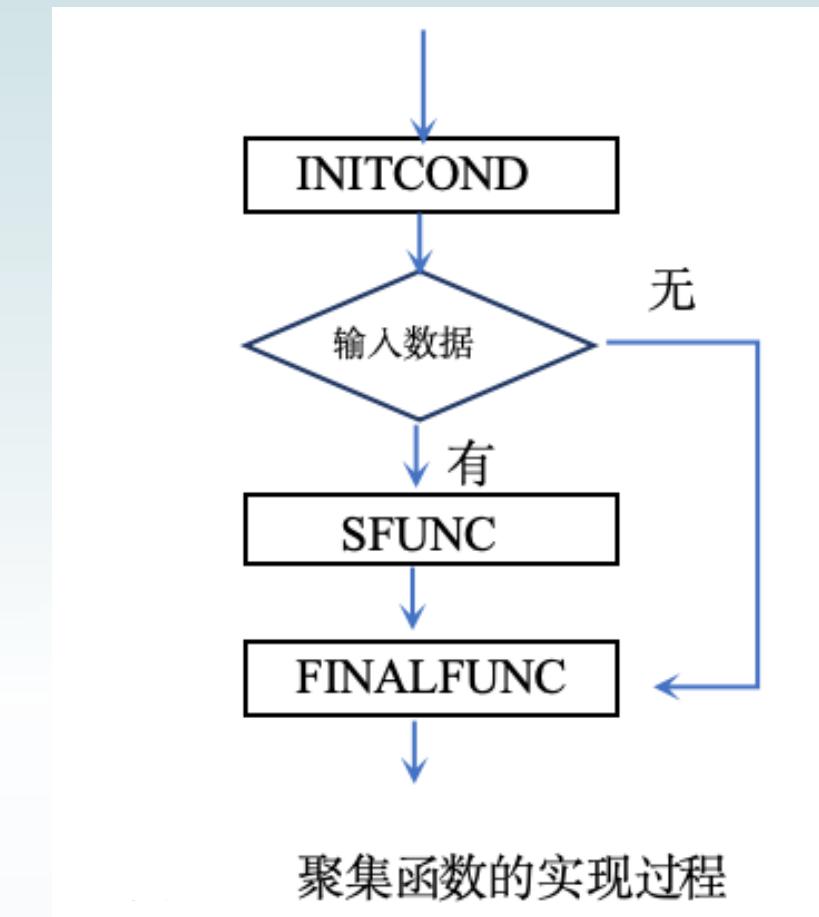
- 去重代价分析
 - 如果R可以放入内存，则磁盘I/O的次数是 $B(R)$;
 - 如果不能放入内存，则磁盘I/O的次数是 $3*B(R)$ 。

7.3.6 分组聚集

- 分组聚集步骤
 - 对给定的输入序列按照某一特定规则进行分组，例如使用排序或哈希表；
 - 对于已经分好的组进行SUM、MAX、MIN、AVG、COUNT等函数的聚集运算；通常在分组过程中就可以进行，不必等收集完一个组的所有元组后再进行计算。

7.3.6 分组聚集

- 聚集函数一般由3个部分协同完成：
 - 初始状态值INITCOND：初始状态的具体值。
 - 状态转移函数SFUNC：根据当前输入值、状态决定下一次状态的值。
 - 结束收尾函数FINALFUNC：处理最后一步的状态转换。



7.3.6 分组聚集

- 分组聚集代价
 - 若R可以放入内存，则磁盘I/O： $B(R)$
 - 若不能，则磁盘I/O： $3*B(R)$

7.3.7 集合操作

- 对于集合的并、交、差操作运算
 - 基于排序的方法：首先对参与集合运算的关系R和S进行排序，
 - 计算 $R \cup S$ 时，如果发现在两个关系中存在相同的元组，则只保留一个；
 - 计算 $R \cap S$ 时，将只包含在两个关系中都出现的元组；
 - 计算 $R - S$ 时，只保留R中那些不出现在S中的元组。

7.3.7 集合操作

- 对于集合的并、交、差操作运算
 - 基于哈希的方法：首先使用相同的哈希函数对两个关系进行分区，分别是 r_1, r_2, \dots, r_k 和 s_1, s_2, \dots, s_k ，
 - $R \cup S$ ：对 r_i 构建内存哈希表；对于 s_i 中的每个元组，检索 r_i 的哈希表，如果其中不存在相同的元组，则加入哈希表；最后将哈希表的元组输入到结果集中。

7.3.7 集合操作

- $R \cap S$: 对 r_i 构建内存哈希表；对于 s_i 中的每个元组，检索 r_i 的哈希表，如果其中存在相同的元组，则将该元组输入到结果集中。
- $R - S$: 对 r_i 构建内存哈希表；对于 s_i 中的每个元组，检索 r_i 的哈希表，如果其中存在相同的元组，则将该元组从哈希表中删除；将哈希表的元组输入到结果集中。

第七章 查询处理

7.1 查询处理概述

7.2 查询编译

7.3 物理操作符

7.4 小结

小结

- 本章对查询处理过程做了整体介绍，它包括查询编译和查询执行两个阶段。
 - 查询编译阶段：SQL词法分析器和语法分析器使用平台工具开发，采用的分别是正则表达式和LR文法。翻译转换后得到的查询表达式树需要进行优化才能得到高效的查询执行计划树。
 - 查询执行阶段：查询执行计划树中的操作算子节点在执行器的调度下做完查询任务。
- 本章对基本的查询操作算子的实现进行了具体说明和简要的代价分析：
 - 通过查询编译和基本关系操作的调度执行，SQL语句被构造为执行计划并获得查询结果。

