西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY 计算机学院 SCHOOL OF COMPUTER SCIENCE

西北工业大学 | Northwestern Polytechnical University

数据库系统原理 | Database System

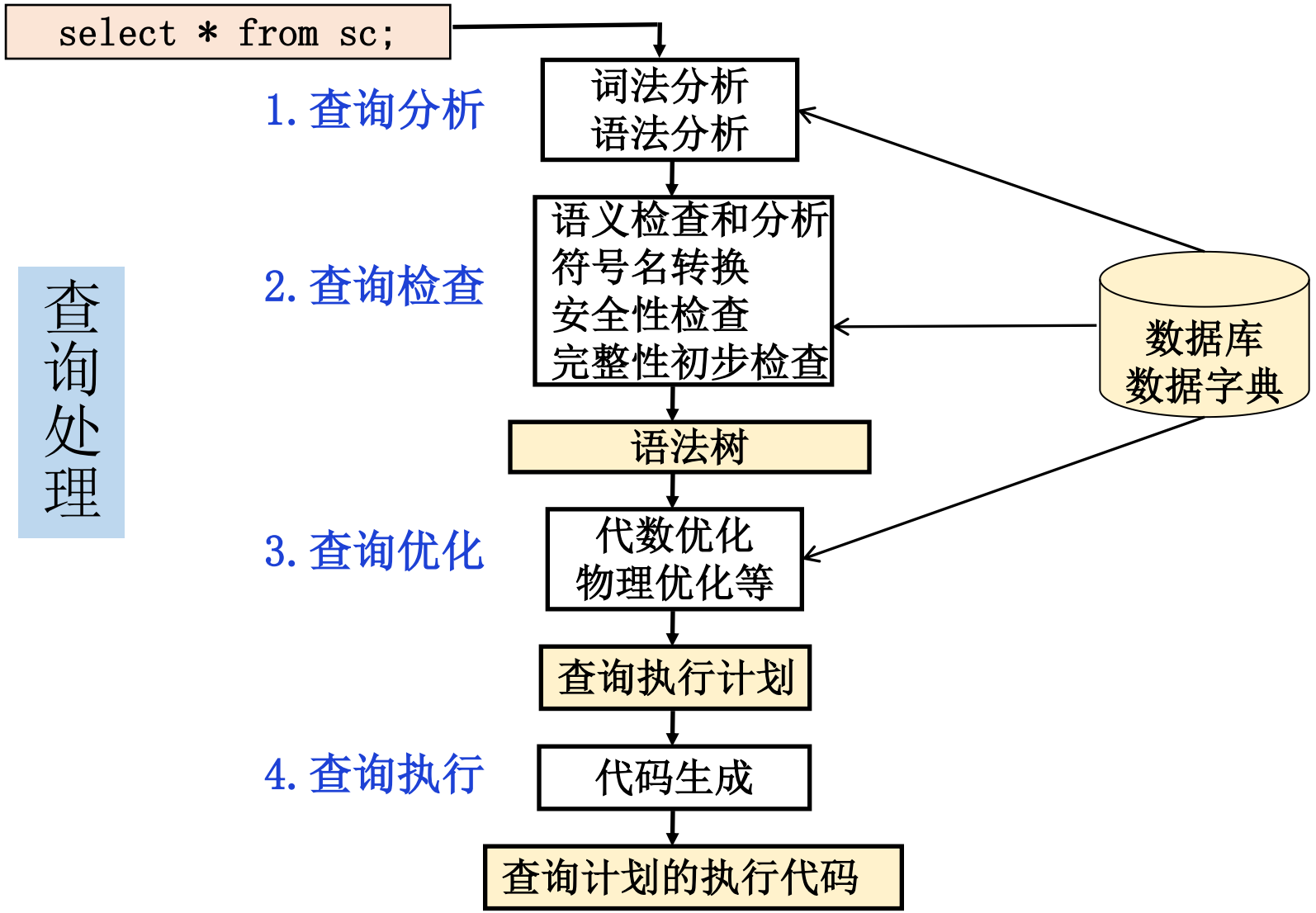
第十章 关系查询处理与查询优化

赵晓南

2024.10

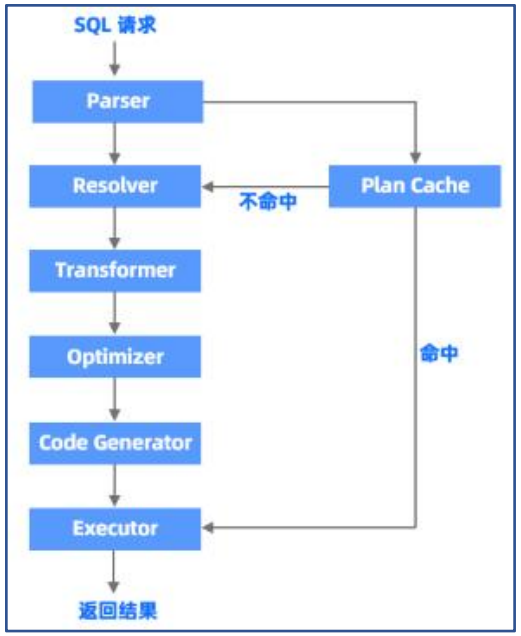
本章目录

10.1.1 查询处理步骤

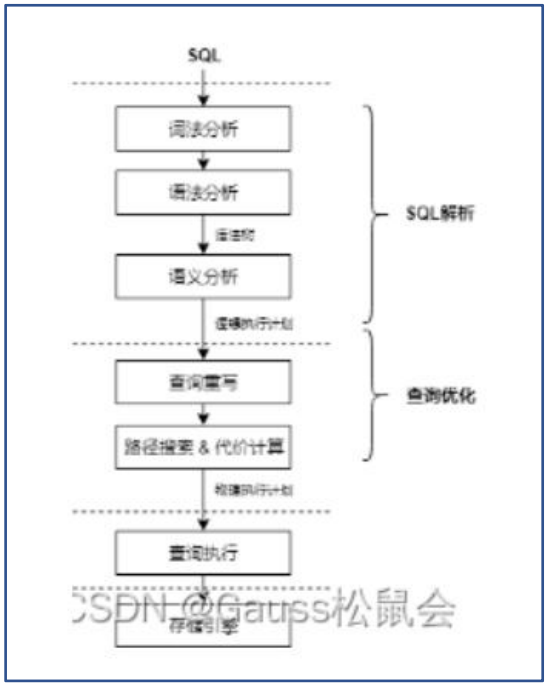

select * from sc;
查询处理
1. 查询分析
词法分析
语法分析
2. 查询检查
语义检查和分析
符号名转换
安全性检查
完整性初步检查
数据库
数据字典
语法树
3. 查询优化
代数优化
物理优化等
查询执行计划
4. 查询执行
代码生成
查询计划的执行代码

10.1.1 关系数据库系统的查询处理

OceanBase的查询处理



openGauss的查询处理



Parser：词法解析 => Parser Tree

Resolver：语义解析 => Statement Tree

Transformer:逻辑改写模块（基于规则的优化）

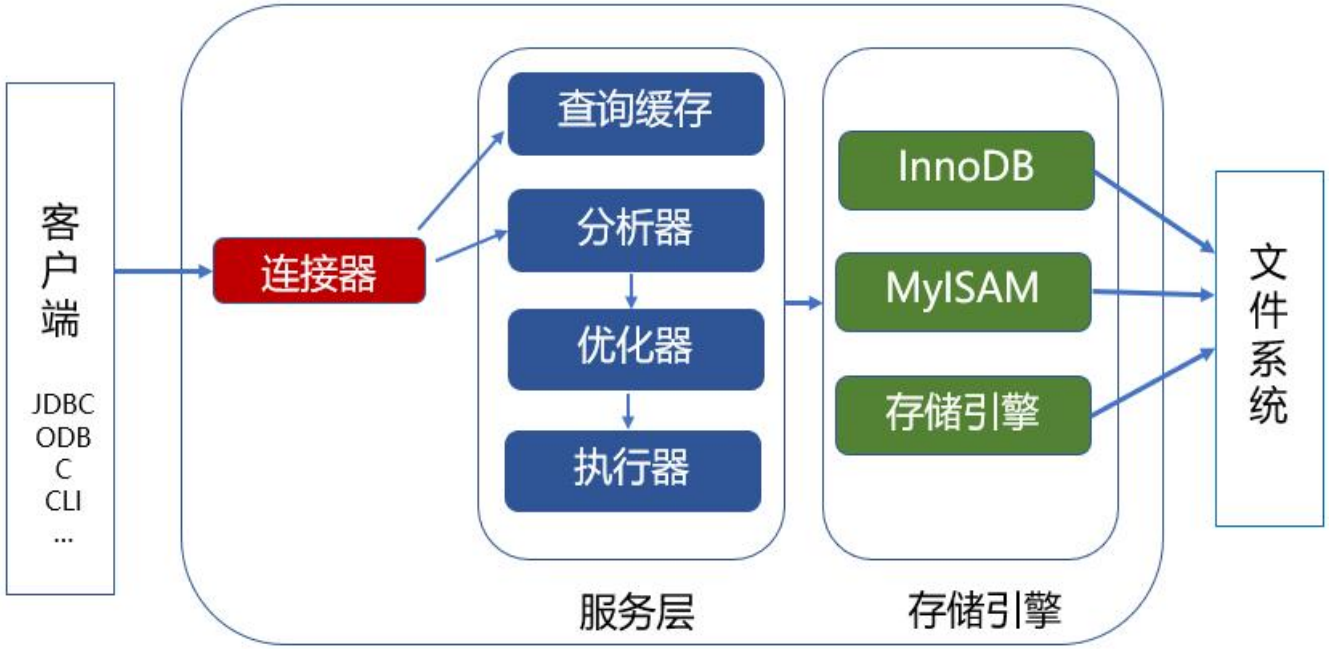
Optimizer: 查询优化（基于代价的优化）

Code Generator：代码生成器[查询计划]

Executor：查询执行器

10.1.1 关系数据库系统的查询处理

- MySQL内核架构



查询分析、查询检查、查询优化、查询执行

10.1.1 查询处理过程

西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

■ 查询处理示例 —— 步骤一：查询分析

1. 词法分析:将用户输入的SQL语句拆解成单词(Token)序列，并识别出关键字、标识、常量等

```
select sno, sname from s where sno=2001
```

保留字： select[SELECT]，from[FROM]， where[WHERE]
变量： sno[IDENT]， sname[IDENT]， s[IDENT]
常量： 2001
运算符： EQ

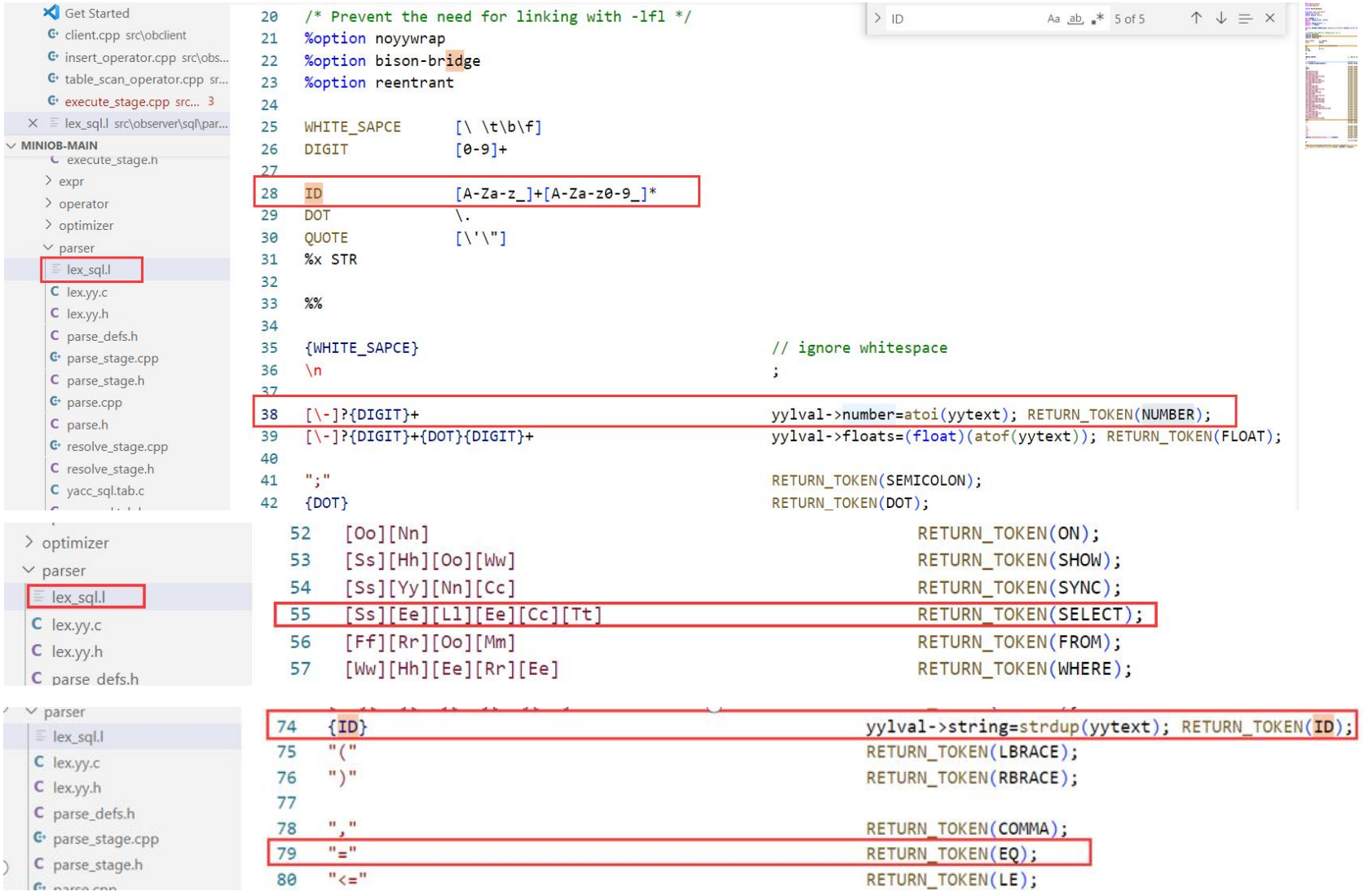
返回token序列：

SELECT， ID， ID， FROM， ID， WHERE， ID， EQ， NUMBER

flex工具 => .l 文件

能发现拼写、命名、非法符号等错误

10.1.1 查询处理过程

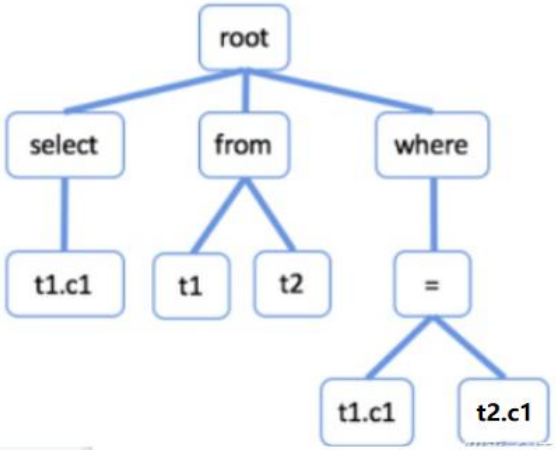

Get Started
client.cpp src\obclient
insert_operator.cpp src\obs...
table_scan_operator.cpp sr...
execute_stage.cpp src... 3
lex_sql.l src\observer\sql\par...
MINIOB-MAIN
execute_stage.h
expr
operator
optimizer
parser
lex_sql.l
lex.yy.c
lex.yy.h
parse_defs.h
parse_stage.cpp
parse_stage.h
parse.cpp
parse.h
resolve_stage.cpp
resolve_stage.h
yacc_sql.tab.c
ID
Aa ab .* 5 of 5
20 /* Prevent the need for linking with -lfl */
21 %option noyywrap
22 %option bison-bridge
23 %option reentrant
24
25 WHITE_SAPCE [\ \t\b\f]
26 DIGIT [0-9]+
27
28 ID [A-Za-z_]+[A-Za-z0-9_]*
29 DOT \.
30 QUOTE [\'\"]
31 %x STR
32
33 %%
34
35 {WHITE_SAPCE} // ignore whitespace
36 \n ;
37
38 [\-]?{DIGIT}+ yylval->number=atoi(yytext); RETURN_TOKEN(NUMBER);
39 [\-]?{DIGIT}+{DOT}{DIGIT}+ yylval->floats=(float)(atof(yytext)); RETURN_TOKEN(FLOAT);
40
41 ";" RETURN_TOKEN(SEMICOLON);
42 {DOT} RETURN_TOKEN(DOT);
optimizer
parser
lex_sql.l
lex.yy.c
lex.yy.h
parse_defs.h
52 [Oo][Nn] RETURN_TOKEN(ON);
53 [Ss][Hh][Oo][Ww] RETURN_TOKEN(SHOW);
54 [Ss][Yy][Nn][Cc] RETURN_TOKEN(SYNC);
55 [Ss][Ee][Ll][Ee][Cc][Tt] RETURN_TOKEN(SELECT);
56 [Ff][Rr][Oo][Mm] RETURN_TOKEN(FROM);
57 [Ww][Hh][Ee][Rr][Ee] RETURN_TOKEN(WHERE);
parser
lex_sql.l
lex.yy.c
lex.yy.h
parse_defs.h
parse_stage.cpp
parse_stage.h
74 {ID} yylval->string=strdup(yytext); RETURN_TOKEN(ID);
75 "(" RETURN_TOKEN(LBRACE);
76 ")" RETURN_TOKEN(RBRACE);
77
78 "," RETURN_TOKEN(COMMA);
79 "=" RETURN_TOKEN(EQ);
80 "<=" RETURN_TOKEN(LE);

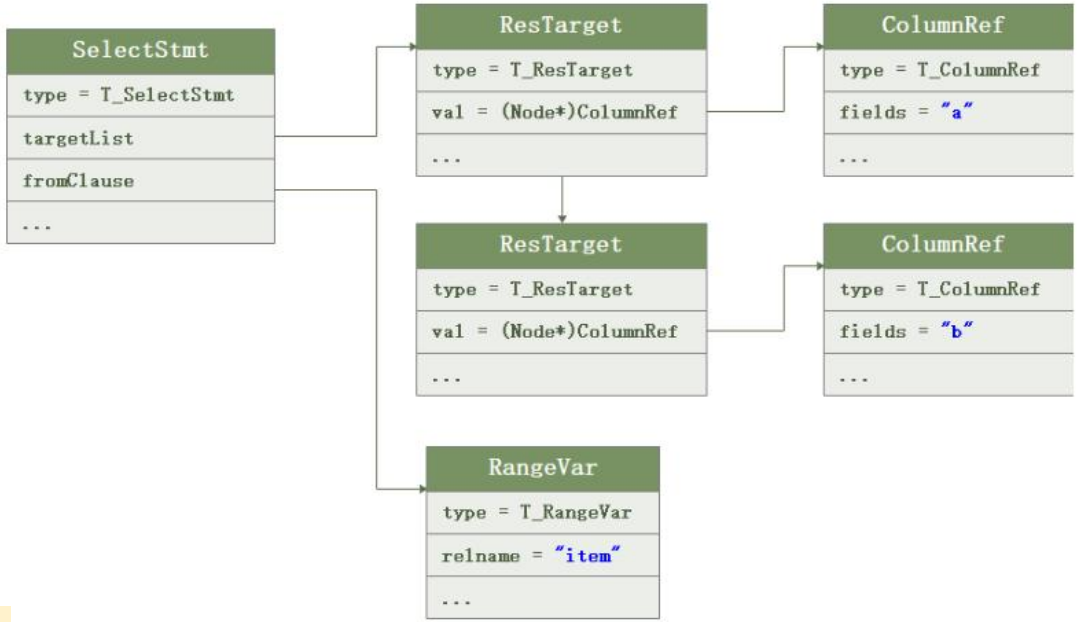
10.1.1 查询处理过程

西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

■ 查询处理示例 —— 步骤一：查询分析

2. 语法分析：将SQL语句生成语法分析树（Parse Tree）。





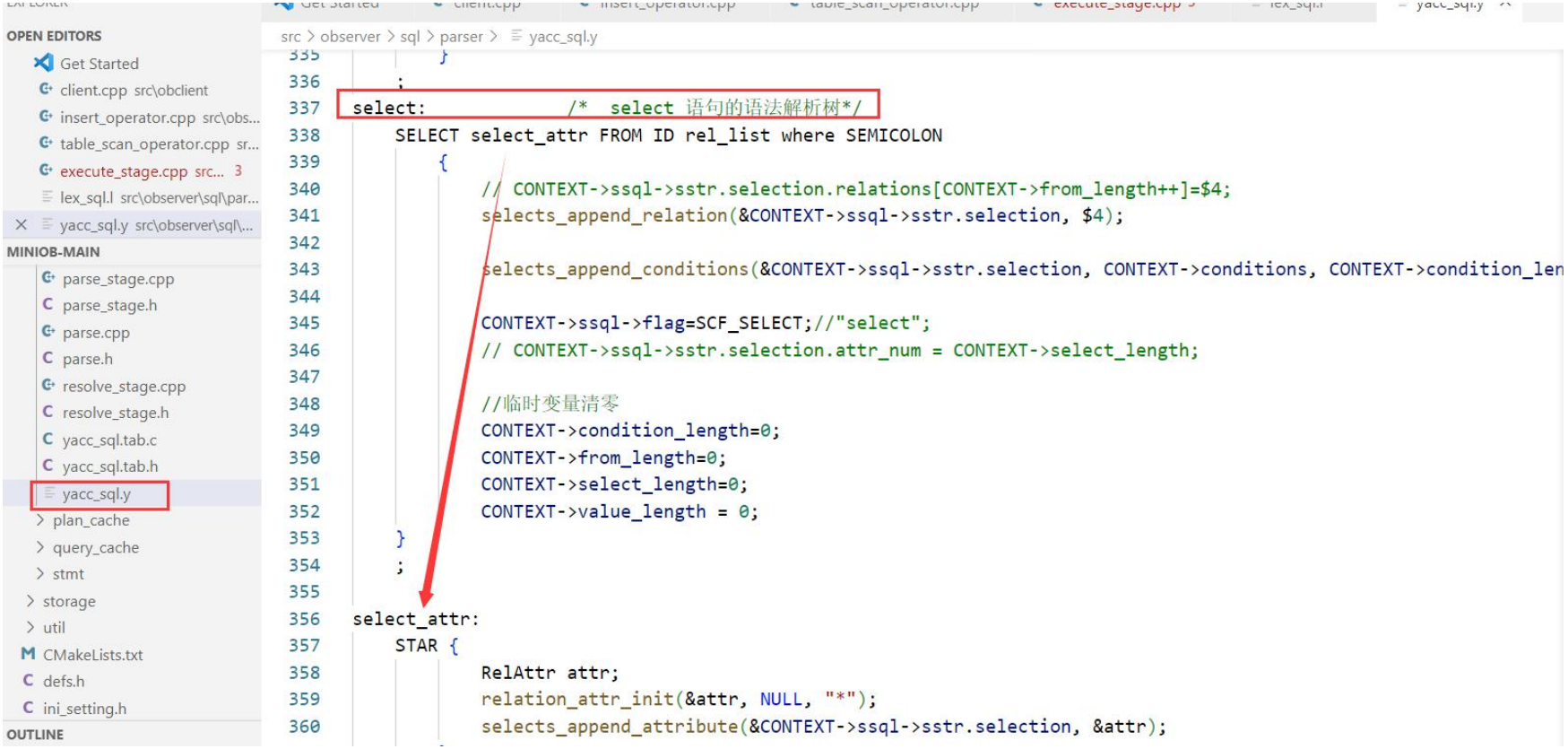
yacc工具 => .y 文件

能发现语法错误
select * from s
where sno=2001

openGauss示例：select a, b from item

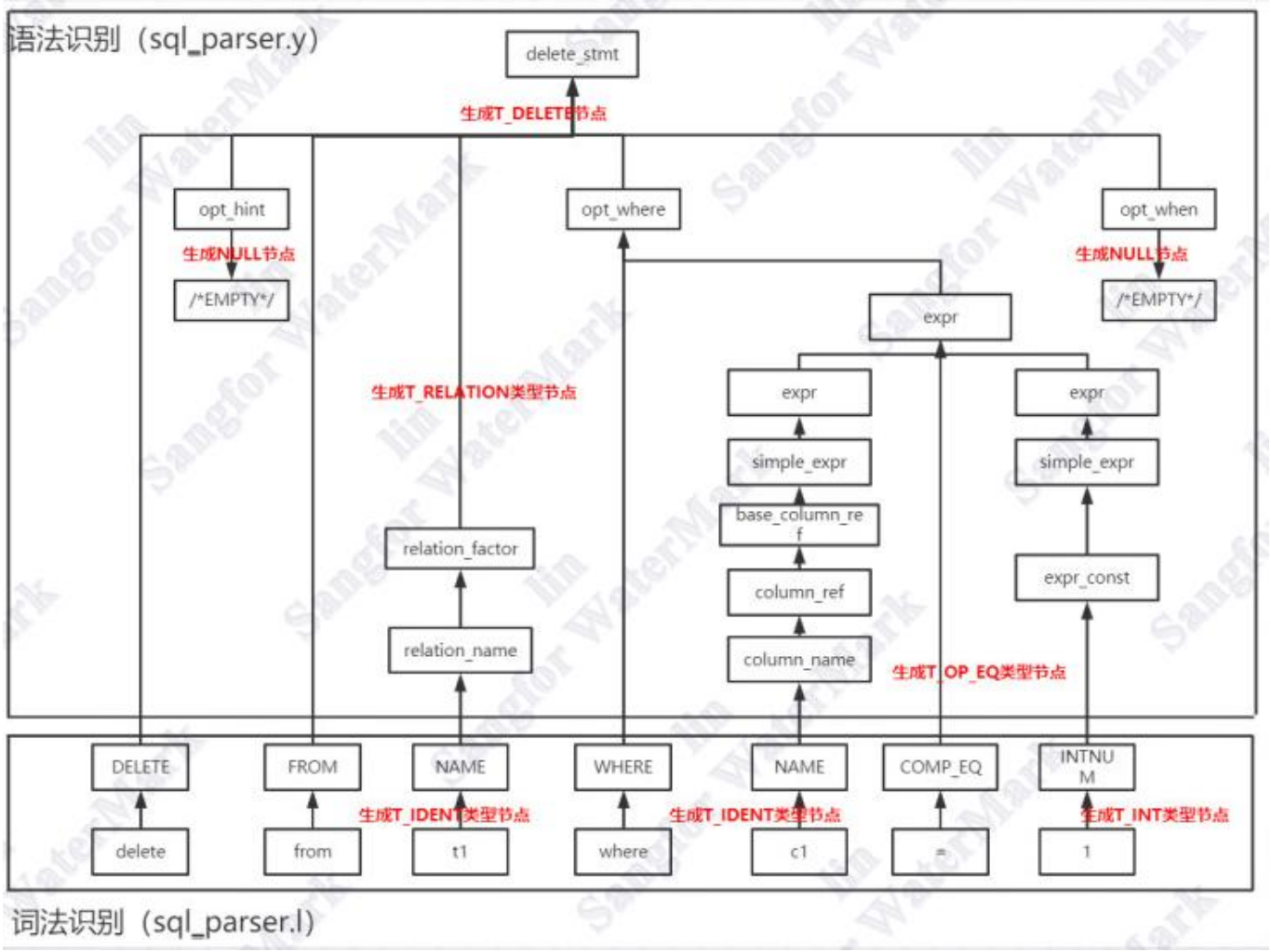
参考：https://zhuanlan.zhihu.com/p/534781312

10.1.1 查询处理过程


OPEN EDITORS
Get Started
client.cpp src\obclient
insert_operator.cpp src\obs...
table_scan_operator.cpp sr...
execute_stage.cpp src... 3
lex_sql.l src\observer\sql\par...
yacc_sql.y src\observer\sql\...
MINIOB-MAIN
parse_stage.cpp
parse_stage.h
parse.cpp
parse.h
resolve_stage.cpp
resolve_stage.h
yacc_sql.tab.c
yacc_sql.tab.h
yacc_sql.y
plan_cache
query_cache
stmt
storage
util
CMakeLists.txt
defs.h
ini_setting.h
OUTLINE
src > observer > sql > parser > yacc_sql.y
}
;
select: /* select 语句的语法解析树*/
SELECT select_attr FROM ID rel_list where SEMICOLON
{
// CONTEXT->ssql->sstr.selection.relations[CONTEXT->from_length++]=$4;
selects_append_relation(&CONTEXT->ssql->sstr.selection, $4);
selects_append_conditions(&CONTEXT->ssql->sstr.selection, CONTEXT->conditions, CONTEXT->condition_len
CONTEXT->ssql->flag=SCF_SELECT;//"select";
// CONTEXT->ssql->sstr.selection.attr_num = CONTEXT->select_length;
//临时变量清零
CONTEXT->condition_length=0;
CONTEXT->from_length=0;
CONTEXT->select_length=0;
CONTEXT->value_length = 0;
}
;
select_attr:
STAR {
RelAttr attr;
relation_attr_init(&attr, NULL, "*");
selects_append_attribute(&CONTEXT->ssql->sstr.selection, &attr);

10.1.1 查询处理过程

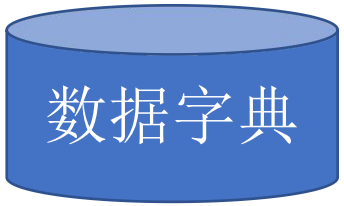
delete from t1 where c1=1

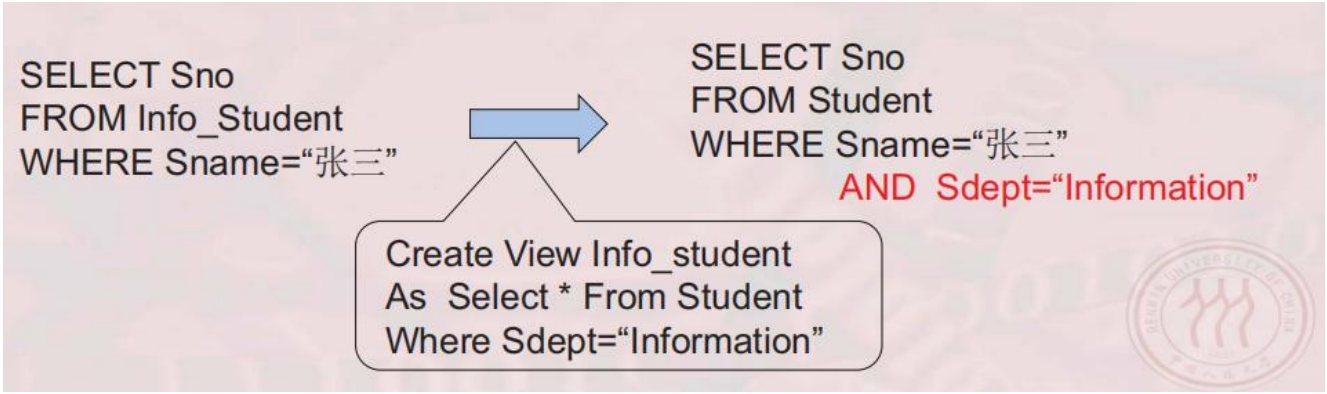


10.1.1 查询处理过程

■ 查询处理示例 —— 步骤二：查询检查

- 有效性：对象是否存在，如数据库名、表明、列名等
- 视图转换：视图消解
- 安全性：是否拥有相关权限
- 完整性：取值类型、范围等是否满足各种约束





10.1.1 关系数据库系统的查询处理

西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

■ 查询处理示例 —— 步骤三：查询优化

目标：选择一个高效执行的查询处理策略

- 查询改写
- 查询优化
  - 逻辑优化（代数优化）
  - 物理优化

- 查询优化选择策略
  - 基于规则
  - 基于代价
  - 基于语义

- 时间代价差异巨大

$Q_1=\pi_{Sname}(\sigma_{Student.Sno=SC.Sno \wedge Sc.Cno='2'}(Student \times SC))$ 10^5s

$Q_2=\pi_{Sname}(\sigma_{Sc.Cno='2'}(Student \bowtie SC))$ 205s

$Q_3=\pi_{Sname}(Student \bowtie \sigma_{Sc.Cno='2'}(SC))$ 10s

10.1.1 关系数据库系统的查询处理

- 查询处理示例 —— 步骤四：查询执行

1. 各个查询算子（投影、选择、连接等）的实现
2. 采用一定的执行模型具体执行（火山模型等）

10.1.2 查询处理 — 查询算子实现

■ 选择算子（SELECT）

全表扫描Table Scan 与 索引扫描Index Scan

[例10.1] SELECT *
FROM Student
WHERE <条件表达式>

考虑<条件表达式>的几种情况：

C1：无条件；

C2：Sno＝'20180003';

C3：Sbirthdate>='2000-1-1';

C4：Smajor='计算机科学与技术'
AND Sbirthdate>='2000-1-1';

- C1:全表扫描Table Scan

（假设可使用内存为M块）

① 按照物理存储顺序，依次读入S表的M块；

② 读取的每个元组t，检查是否满足条件（如无条件，则无需检查）

③ 如果S表还有未处理数据，循环前两步

10.1.2 查询处理 — 查询算子实现

■ 选择算子（SELECT）

C2: select * from s where sno='2001'

- 如sno列有索引，采用Index scan

C3: select * from s where Sbirthdate>='2000-1-1'

- 如Sbirthdate有B+树辅助索引，采用该B+树索引查询，得到所有符合条件数据的主码sno，再回表到s查询其他属性

问题：如果99%的学生都是是2000年后出生，使用索引效率高还是不使用高？

10.1.2 查询处理 — 查询算子实现

- 选择算子　需要根据查询条件的复杂程度决定策略

C4: select * from s where sdept='cs' and Sbirthdate>='2000-1-1'

假设Sbirthdate和sdept都有索引，如何实现查询？

- 方法1：同时使用两个索引
  - ① 分别使用Sbirthdate索引和sdept索引查询
  - ② 取交集输出

使用哪个更好？
Sbirthdate?sdept?

- 方法2：仅使用一个索引
  - ① 使用Sbirthdate索引或者sdept索引查询
  - ② 在上步查询结果中对另外一个条件过滤
  - ③ 输出结果

选择算子

- 选择率较低（命中数据占所有数据的比例）时，基于索引的选择算法要优于全表扫描。
- 某些情况下，如选择率较高、或者要查找的元组均匀分散在表中，由于还需要考虑扫描索引带来的额外开销，此时索引扫描法的性能可能还不如全表扫描法。

10.1.2 查询处理 —查询算子实现

- 连接算子

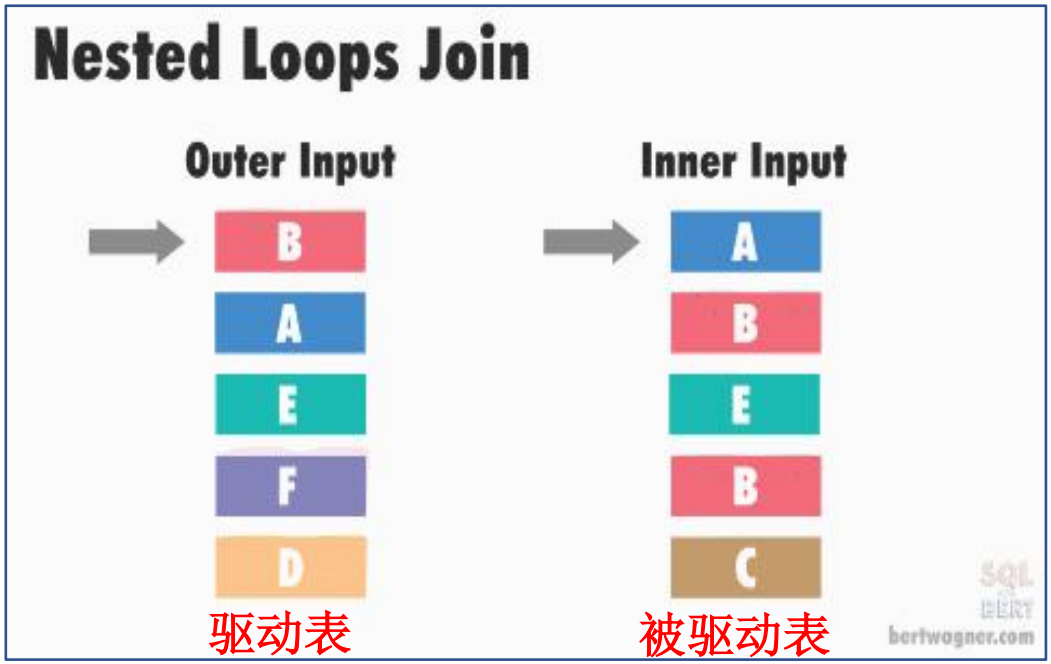
```
select *
from s, sc
where s.sno = sc.sno
```

- ✓ 嵌套循环算法（Nested Loop Join, NLJ）
- ✓ 归并排序算法（Merged Join）
- ✓ 索引连接算法（Index Join）
- ✓ 散列连接算法(Hash join)

10.1.2 查询处理 — 连接算子实现

西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

■ 基本嵌套循环算法(NLJ)



动画: 引自bertwager.com

NLJ的优化算法:
块嵌套查询Block Nested-Loop Join
索引嵌套查询Index Nested-Loop Join

```
select *
from t1, t2
where t1.id = t2.id
```

1. 实际物理实现中，磁盘IO以块为单位，实际每次读取1块而不是1个元组。
2. 推荐：内外循环表的连接属性上有索引

问题：t1和t2哪个作为外循环表（驱动表），效率更高？

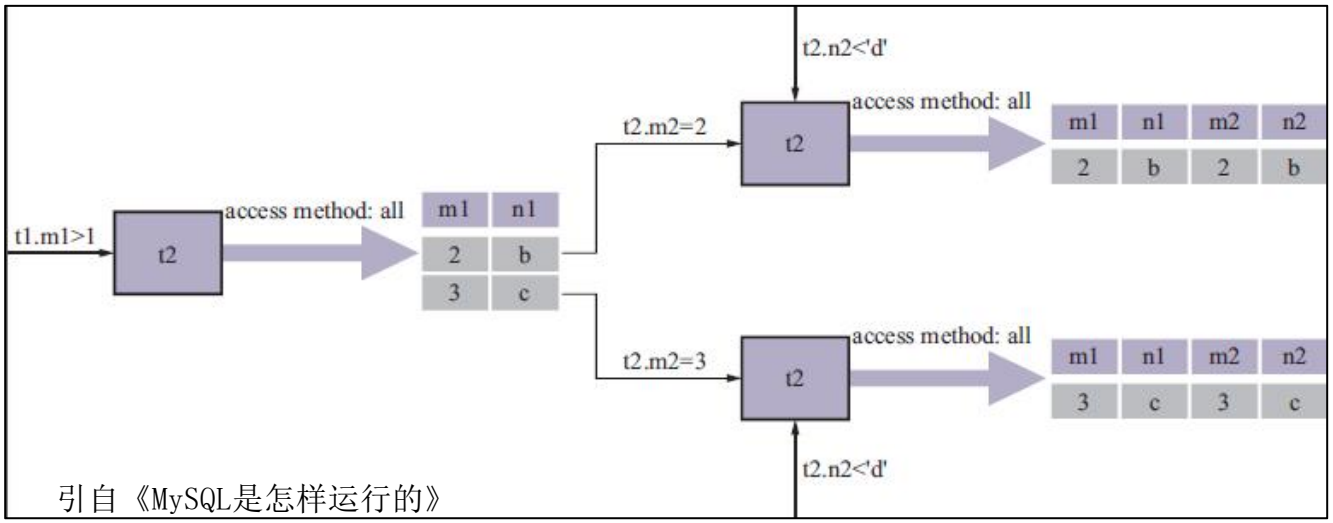
10.1.2 查询处理 — 连接算子实现

t1		t2	
m1	n1	m2	n2
1	a	2	b
2	b	3	c
3	c	4	d

■ 基本嵌套循环算法(NLJ)

select * from t1, t2 where t1.m1>1 and t1.m1=t2.m2 and t2.n2<’d’

驱动表：t1
被驱动表:t2



查询驱动表t1的t1.m1>1，此时结果集2条记录

① t1.m1=2时，查询一遍t2表：select * from t2 where t2.m2=2 and t2.n2 <’d’

② t1.m1=3时，查询一遍t2表：select * from t2 where t2.m2=3 and t2.n2 <’d’

提高效率的关键：1）访问表的次数少； 2）访问表的代价小；

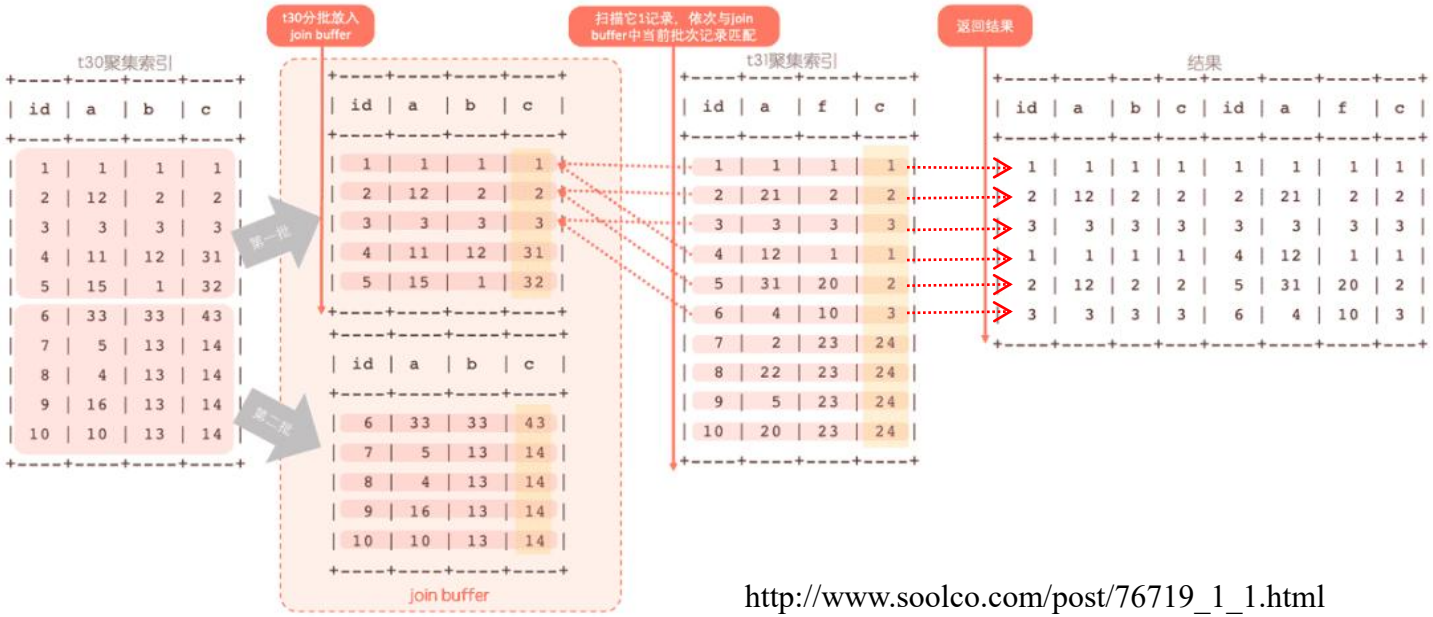
10.1.2 查询处理 — 连接算子实现

西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

■ 嵌套循环算法(NLJ)

块嵌套查询Block Nested-Loop Join（BNLJ）

select * from t30 join t31 on t30.c=t31.c



http://www.soolco.com/post/76719_1_1.html

块嵌套：

1. 磁盘IO需要，物理上以块为单位读写
2. 为了尽可能利用较大的内存，内存缓冲区可存放多个物理块

1. 将t30(驱动表)的数据一次或者多次读入到join_buffer中(默认内存大小为256k, 如果数据量多, 会进行分段存放, 然后进行比较)
2. 把表t31的每一行数据, 跟join_buffer中的数据批量进行对比。
3. 循环上两个步骤, 直到无法满足条件, 将结果集返回给客户端。

10.1.2 查询处理 — 连接算子实现

西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

■ 嵌套循环算法(NLJ)

索引嵌套查询Index Nested-Loop Join（INLJ）

```
select * from t30 join t31
on t30.a=t31.a
```

如t31.a上有辅助索引

1. 从外表(驱动表)t30中读取1行数据R;
2. 用数据R中a对应的值(如12),在t31的索引表中查找相关主键id记录;
3. 回表操作：根据t31中访问对应id的记录;
4. 组合t30表和t31表组成1行，加入结果集;
5. 重复执行步骤1到3，直到表t1循环结束。

http://www.soolco.com/post/76719_1_1.html

10.1.2 查询处理 — 连接算子实现

西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)

■ 嵌套循环算法(NLJ)

关系	元组数	元组长度(字节)	存储元组/每块	所有元组所需的存储块数
	n	l	tb	b
student(外r)	5000	100	5000/100=50	5000/50=100
takes(内s)	10000	200	5000/200=25	10000/25=400

假设每个block存储5000字节

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add t_r · t_s to the result;
    end
end
```

Figure 15.5 Nested-loop join.

查询代价 => 扫描代价尽可能低

1）s表扫描代价=单次代价*次数

2）t表扫描代价=单次代价*次数

3）满足连接条件元组的比较代价

本例待比较的总pair:

$n_r * n_s = 5000 * 10000$

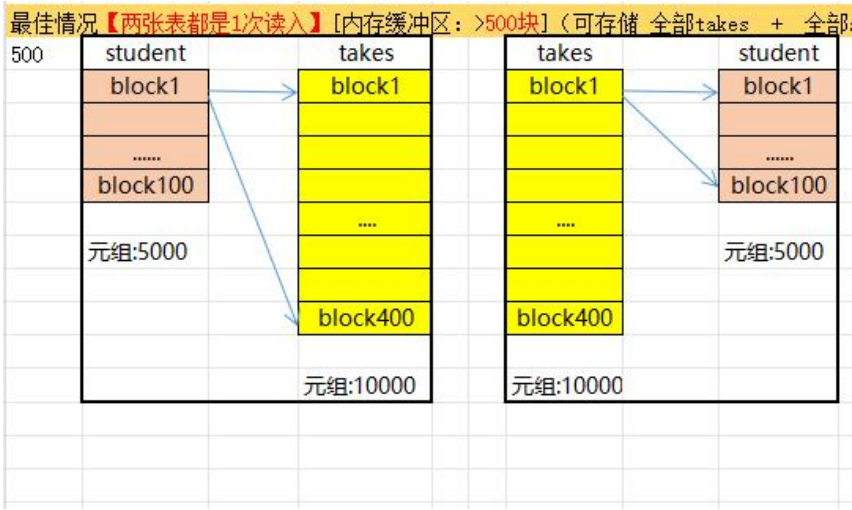
4）结果输出代价

10.1.2 查询处理 — 连接算子实现

关系	元组数	元组长度（字节）	存储元组/每块	所有元组所需的存储块数
	n	l	tb	b
student(r)	5000	100	5000/100=50	5000/50=100
takes(s)	10000	200	5000/200=25	10000/25=400

NLJ性能分析：

1. 最佳情况：两张表都可以1次全部读入内存中



基本嵌套查询

驱动表（左表）	读次数【左表次数+右表次数】	块传输次数【左表次数+右表次数】
student	2=1+1	500=100+400
takes	2=1+1	500=400+100

2=1+1：左表student寻址1次，右表takes寻址1次

2=1+1：左表takes寻址1次，右表student寻址1次

传输次数 = br + bs

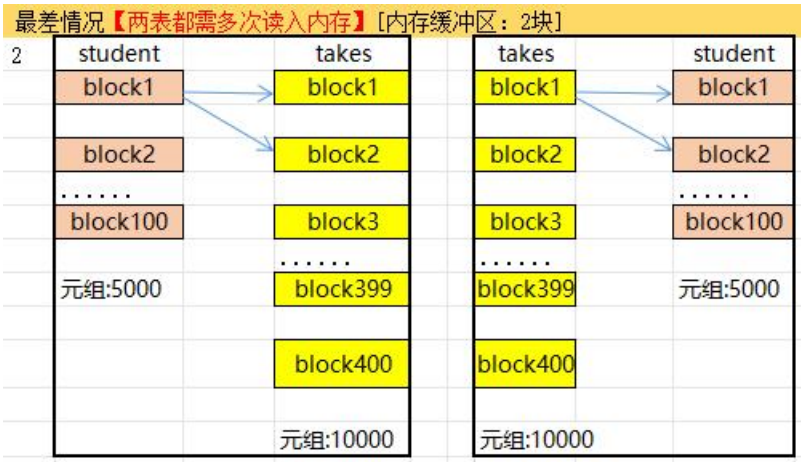
块嵌套查询

驱动表（左表）	读次数【左表次数+右表次数】	块传输次数【左表次数+右表次数】
student	2=1+1	500=100+400
takes	2=1+1	500=400+100

10.1.2 查询处理 — 连接算子实现

NLJ性能分析：

2. 最差情况：两张表都不能完全读入内存，1次只能读1块



基本嵌套查询

驱动表（左表）	读次数（磁盘寻道）【左表次数+右表次数】	块传输次数【左表次数+右表次数】
student	5100【100+5000】	2000100【100+(5000*400)】
takes	10400【400+10000】	1000400【400+(10000*100)】

100 + 5000：左表student寻址100次，右表takes寻址5000次（每条student记录需要重新寻址1次takes）

400 + 10000：左表takes寻址400次，右表student寻址10000次（每条takes记录需要重新寻址1次student）

寻道次数 = br + nr
传输次数 = br + nr*bs

块嵌套查询

驱动表（左表）	读次数（磁盘寻道）【左表次数+右表次数】	块传输次数【左表次数+右表次数】
student	200【100+100】	40100【100+(100*400)】
takes	800【400+400】	40400【400+(400*100)】

100 + 100：左表student寻址100次，右表takes寻址100次（每个student块寻址1次takes）

400 + 400：左表takes寻址400次，右表student寻址400次（每个takes块寻址1次student）

寻道次数 = br + br=2br
传输次数 = br+br*bs

基本嵌套查询算法：大表作为驱动表效率更高

块嵌套查询算法：　小表作为驱动表效率更（实际中应用）

10.1.2 查询处理 — 连接算子实现

西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

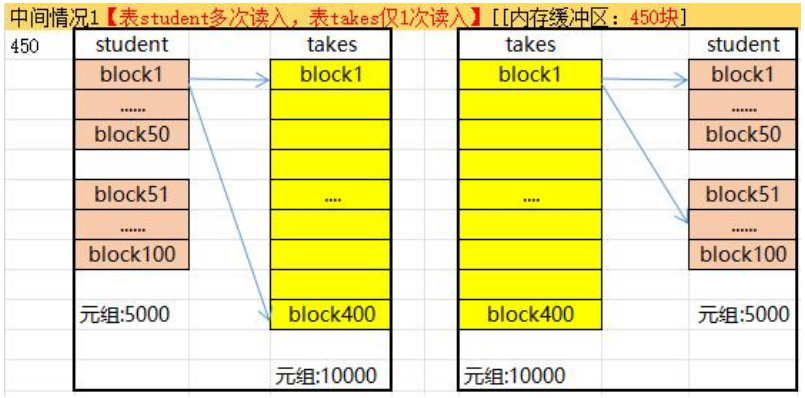
NLJ性能分析：3. 中间情况：仅一张表可以完全放入内存

将可以完全放入内存的表作为内表性能更高，且内表越大越好。

takes比student表大，此时takes做内表更好。

10.1.2 查询处理 — 连接算子实现

中间情况1的详细过程参考



基本嵌套查询

驱动表（左表）	读次数（磁盘寻道+旋转）【左表次数+右表次数】	块传输次数【左表次数+右表次数】
student	2=1+1	500【100+400】
takes	100001=1+10000	1000400【400+(10000*100)】

块嵌套查询

驱动表（左表）	读次数（磁盘寻道+旋转）【左表次数+右表次数】	块传输次数【左表次数+右表次数】
student	2=1+1	500【100+400】
takes	401=1+400	40400【400+(400*100)】

基本嵌套查询过程（student为驱动表）：共计2次寻址，500块传输	基本嵌套查询过程（takes为驱动表）：共计1+10000次寻址，400+10000*100个块传输
1.从磁盘读入student表的第一批1-50块数据到内存（1次寻址，50块传输）	1.从磁盘读入takes表的所有400块进内存，一直暂存在内存中（1次寻址，400块传输）
2.从磁盘读入takes表的所有400块进内存，一直暂存在内存中（1次寻址，400块传输）	2.从磁盘读入student表的第一批1-50块数据（1次寻址，50块传输）
3.student表的第1条记录，依次和takes的所有记录逐条对比连接	3.takes表的第1条记录，依次和student表1-50块内的记录逐条对比连接
4.student表的第i条记录，依次和takes的所有记录逐条对比连接	4.从磁盘读入student表的第二批51-100块数据（两个块顺序存储，不需寻址，50块传输）
5.student表的第50条记录，依次和takes的所有记录逐条对比连接	5.takes表的第1条记录，依次和student表51-100块内的记录逐条对比连接
6.从磁盘读入student表的第二批数据51-100块到内存（假设两个块顺序存储，不需寻址，50块传输）	6.takes表的第2条记录，重复以上2-5的动作，即takes表的每条记录需要1次寻址，100块传输
7.student表的第51条记录，依次和takes的所有记录逐条对比连接	7.takes表的10000条记录，依次处理，共计需要 10000次寻址，10000*100个块传输
8.student表的第n条记录，依次和takes的所有记录逐条对比连接	
9.student表的第100条记录，依次和takes的所有记录逐条对比连接	
块嵌套查询过程（student为驱动表）【与基本嵌套查询相同】：共计2次寻址，500块传输	块嵌套查询过程（takes为驱动表）：共计1+400次寻址，400+400*100个块传输
1.从磁盘读入student表的第一批1-50块数据到内存（1次寻址，50块传输）	1.从磁盘读入takes表的所有400块进内存，一直暂存在内存中（1次寻址，400块传输）
2.从磁盘读入takes表的所有400块进内存，一直暂存在内存中（1次寻址，400块传输）	2.从磁盘读入student表的第一批1-50块数据（1次寻址，50块传输）
3.student表的第1条记录，依次和takes的所有记录逐条对比连接	3.takes表的第1块中所有记录，依次和student表1-50块内的记录逐条对比连接
4.student表的第i条记录，依次和takes的所有记录逐条对比连接	4.从磁盘读入student表的第二批51-100块数据（两个块顺序存储，不需寻址，50块传输）
5.student表的第50条记录，依次和takes的所有记录逐条对比连接	5.takes表的第1块中所有记录，依次和student表51-100块内的记录逐条对比连接
6.从磁盘读入student表的第二批数据51-100块到内存（假设两个块顺序存储，不需寻址，50块传输）	6.takes表的第2块中所有记录，重复以上2-5的动作，即takes表的每块数据需要1次寻址，100块传输
7.student表的第51条记录，依次和takes的所有记录逐条对比连接	7.takes表的400块数据，依次处理，共计需要 400次寻址，400*100个块传输
8.student表的第n条记录，依次和takes的所有记录逐条对比连接	
9.student表的第100条记录，依次和takes的所有记录逐条对比连接	

10.1.2 查询处理 — 连接算子实现

关系	元组数	元组长度（字节）	存储元组/每块	所有元组所需的存储块数
	n	l	tb	b
student(r)	5000	100	5000/100=50	5000/50=100
takes(s)	10000	200	5000/200=25	10000/25=400

嵌套查询性能分析结论（需要存储块数多称为大表，块数少称为小表）：

1）最好情况下(两表都能放入内存)：任何一个表做驱动表效率相同。

2）中间情况下(仅1张表能放入内存)：将可以完全放入内存的表作为内表性能更高，且内表越大越好。如takes比student表大，则takes做内表更好。

3）最差情况下(两表都不能放入内存)：基本嵌套查询算法时，大表作为驱动表效率更高；块嵌套查询算法时（由于磁盘IO基本单位是Block，实际DBMS系统中使用该算法），则小表驱动更好。

10.1.2 查询处理 — 连接算子实现

- 嵌套循环算法小结(NLJ)

```
select *  from t1, t2  where t1.a = t2.a
```

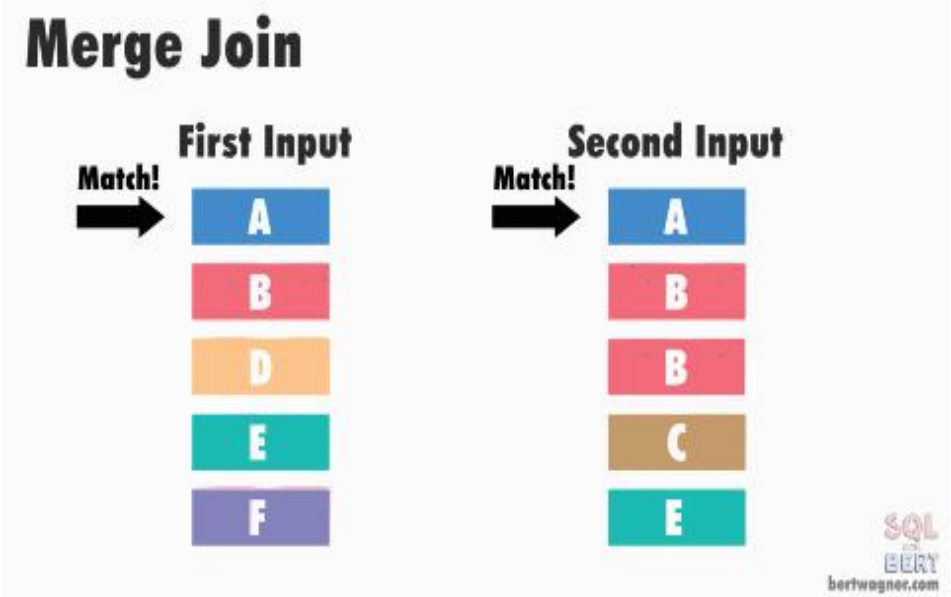
优化目标：减少嵌套的循环次数

- Block Nested-Loop Join： 是通过一次缓存多条数据批量匹配的方式来减少外层表的循环次数。
- Index Nested-Loop Join：通过索引的机制减少内层表的循环次数。
- 对于简单NLJ：大关系做驱动的外表，小关系做被驱动的内表，性能较好；且一个关系能完全放入内存，则放入内表。

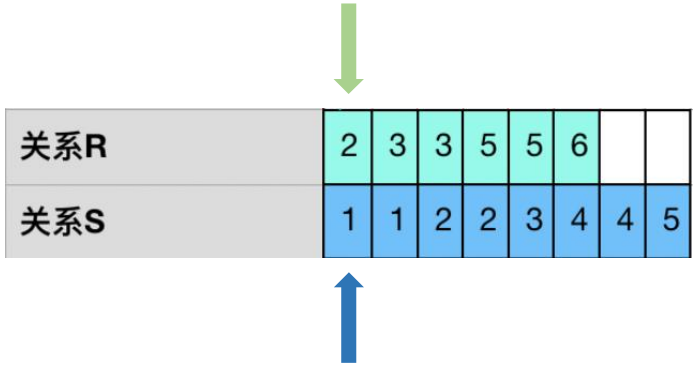
10.1.2 查询处理 — 连接算子实现

西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

- 归并连接算法（Merged Join）




```
select *
from t1, t2
where t1.a = t2.a
```



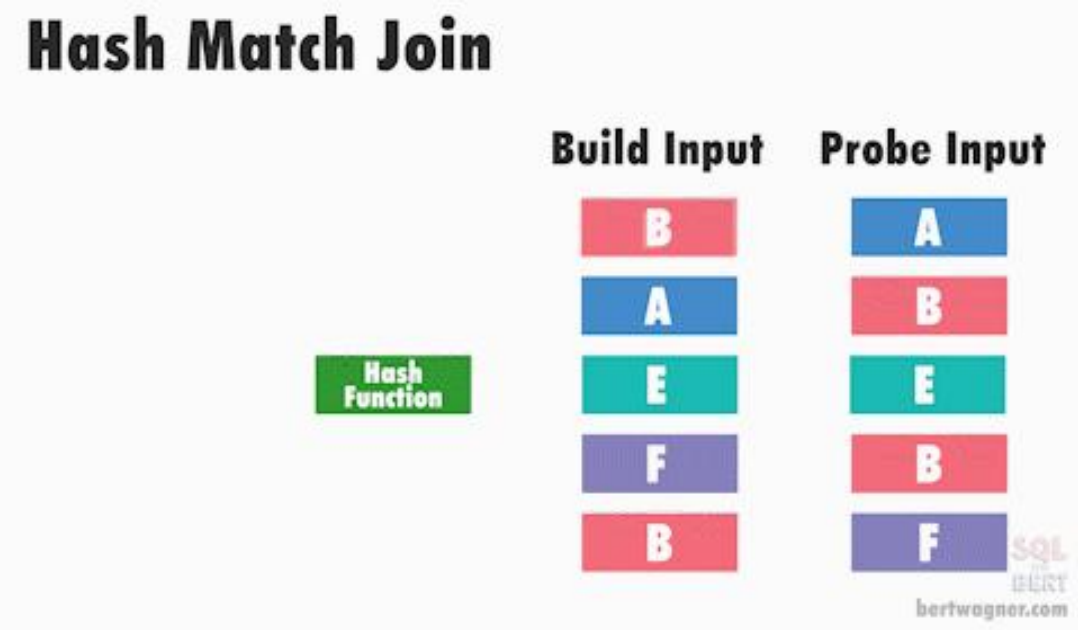
1. 两个表分别排序；
2. 两个指针分别下移，如果相等返回相等行的连接；
3. 假如两个值不匹配，则较小值的输入集合往后移动1次
4. 重复以上2和3，到两个表都遍历结束

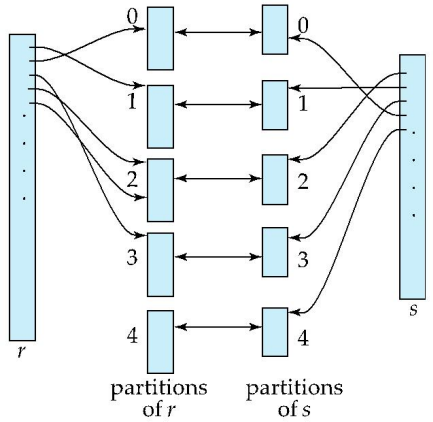
https://blog.csdn.net/enmotech/article/details/86581190

10.1.2 查询处理 — 连接算子实现

- 散列连接算法（Hash Join）

```
select *
from t1, t2
where t1.a = t2.a
```





查询步骤：

1. Build Input：驱动表t1（小表）的连接字段a上创建哈希桶；
2. Probe Input：被驱动表t2，按照连接字段a使用同样的hash函数进行哈希匹配（相同值的hash函数值也必然相同），若落在相同的hash桶，且满足a的连接条件，则进行连接。

10.1.2 查询处理 — 连接算子实现

- 连接算子对比

比较	Nested Loop	Hash Join	Merge Join
使用条件	任何条件	等值连接（=）	等值或非等值连接(>，<，=，>=，<=，‘<>’除外)
相关资源	CPU、磁盘I/O	内存、临时空间	内存、临时空间
特点优点	当有高选择性索引或进行限制性搜索时效率比较高，能够快速返回第一次的搜索结果。	当缺乏索引或者索引条件模糊时，Hash Join比Nested Loop有效。通常比Merge Join快。特别是表记录多时，优势明显。	若本身就是有序数据，更有优势。 非等值连接时，Merge Join比Hash Join更有效。
缺点	当索引丢失或者查询条件限制不够时，效率很低；当表的纪录数多时，效率低。	为建立哈希表，需要大量内存。第一次的结果返回较慢。	所有的表都需要排序。它为最优化的吞吐量而设计，并且在结果没有全部找到前不返回数据。

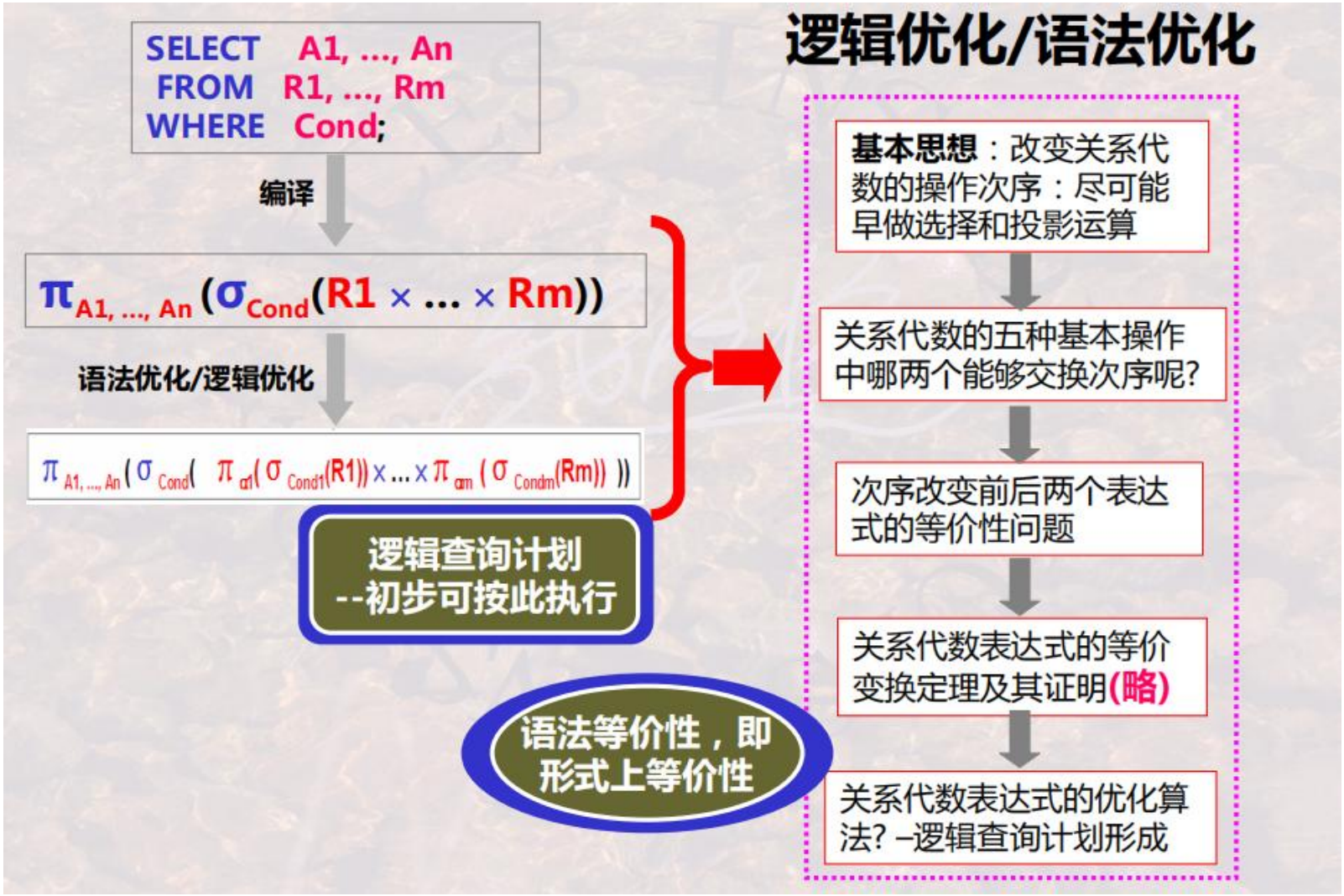
本章目录

目标：如何使数据库查询的执行时间最短？

引自：哈工大 战德臣教授 中国大学MOOC 数据库原理

本章目录

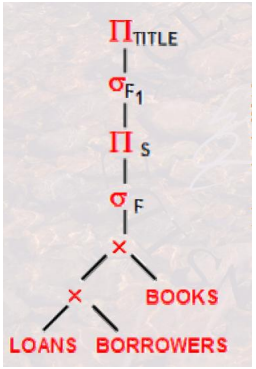
10.3 代数优化


SELECT A1, ..., An
FROM R1, ..., Rm
WHERE Cond;
编译
$\pi_{A1,...,An}(\sigma_{Cond}(R1 \times ... \times Rm))$
语法优化/逻辑优化
$\pi_{A1,...,An}(\sigma_{Cond}(\pi_{am}(\sigma_{Condm}(Rm)) \times ... \times \pi_{a1}(\sigma_{Cond1}(R1))))$
逻辑查询计划
--初步可按此执行
语法等价性，即
形式上等价性
逻辑优化/语法优化
基本思想：改变关系代数的操作次序：尽可能早做选择和投影运算
关系代数的五种基本操作中哪两个能够交换次序呢?
次序改变前后两个表达式的等价性问题
关系代数表达式的等价变换定理及其证明(略)
关系代数表达式的优化算法? –逻辑查询计划形成

关系代数表达式的等价变化规则

1. 连接、笛卡尔积的交换律 $E_1 \times E_2 \equiv E_2 \times E_1$，$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$
2. 连接、笛卡尔积的结合律 $(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$，$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$
3. 投影的串接定律 $\pi_{A_1,A_2,\cdots A_n}(\pi_{B_1,B_2,\cdots B_m}(E)) \equiv \pi_{A_1,A_2,\cdots A_n}(E)$
4. 选择的串接定律 $\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$
5. 选择与投影的交换律 $\sigma_F(\pi_{A_1,A_2,\cdots A_n}(E)) \equiv \pi_{A_1,A_2,\cdots A_n}(\sigma_F(E))$
6. 选择与笛卡尔积的交换律 $\sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$
7. 选择与并的分配律 $\sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2)$
8. 选择与差的分配律 $\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2)$
9. 选择对自然连接的分配律 $\sigma_F(E_1 \bowtie E_2) \equiv \sigma_F(E_1) \bowtie \sigma_F(E_2)$
10. 投影对笛卡尔积的分配律 $\pi_{A_1,A_2,\cdots A_n,B_1,B_2,\cdots B_m}(E_1 \times E_2) \equiv \pi_{A_1,A_2,\cdots A_n}(E_1) \times \pi_{B_1,B_2,\cdots B_m}(E_2)$
11. 投影与并的分配律 $\pi_{A_1,A_2,\cdots A_n}(E_1 \cup E_2) \equiv \pi_{A_1,A_2,\cdots A_n}(E_1) \cup \pi_{A_1,A_2,\cdots A_n}(E_2)$

查询树启发式优化规则(一)



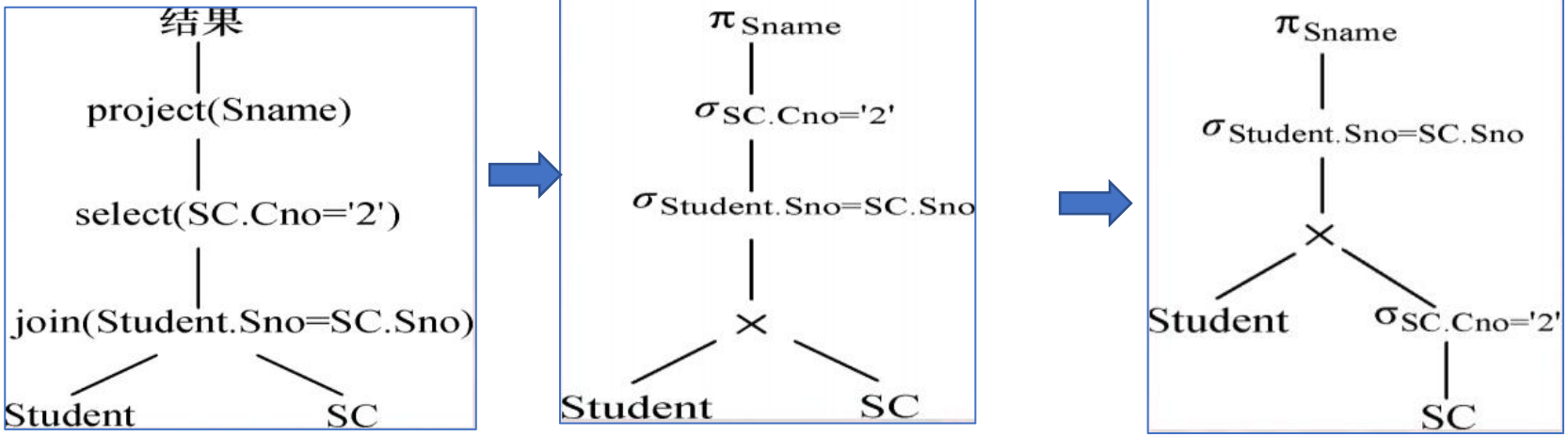
1. 尽可能地早做选择和投影：可使中间结果变小，节省执行时间。
2. 选择与投影同时进行：避免对整个关系的多次重复扫描。
3. 把投影与其前或后的二元运算结合起来：在第一次用关系时去掉一些无关属性，可以避免多次扫描整个关系。
4. 把某些选择与其前的笛卡尔积合并成一个连接：连接运算（特别是等值连接）比笛卡尔积快很多。
5. 执行连接运算前对关系做适当预处理：文件排序、建立临时索引等，可使两关系公共值高效联接。
6. 找出表达式里的公共子表达式：若公共子表达式结果不大，则预先计算，特别是视图情况下有用。

查询树启发式优化规则（二）

1. 利用等价变化规则4（选择的串接）变换。
2. 对每一个选择运算，利用规则4-9尽可能移动到树的叶端。
3. 对每一个投影运算，利用规则3, 5, 10-11尽可能移动到树的叶端。
4. 利用规则3-5，尽可能使得多个投影和选择同时进行，只做一次扫描。
5. 对得到的语法树，进行语法树内的节点分组。

例：对如下查询进行代数优化：查询选修2号课程的学生姓名。

```
select sname
from student as s join sc on s.sno = sc.sno
where sc.cno='2'
```



SQL查询树　　关系代数树　　规则2, 4：选择移到叶端

10.3 代数优化

求选修了2号课程的学生姓名，Q1,Q2,Q3谁的效率高?

- 时间代价差异巨大

$Q_1=\pi_{Sname}(\sigma_{Student.Sno=SC.Sno \wedge Sc.Cno='2'}(Student \times SC))$ 10^5s

$Q_2=\pi_{Sname}(\sigma_{Sc.Cno='2'}(Student \bowtie SC))$ 205s

$Q_3=\pi_{Sname}(Student \bowtie \sigma_{Sc.Cno='2'}(SC))$ 10s

效率最高方法是效率最低方法的多少倍?

A. 100倍， B：1000倍， C:10000倍

本章目录

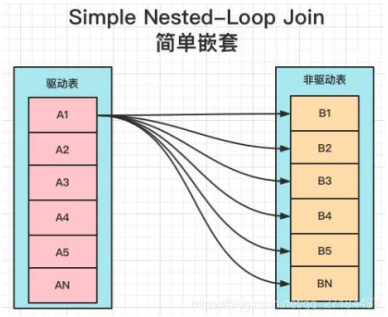
基于启发式规则的存取路径选择优化

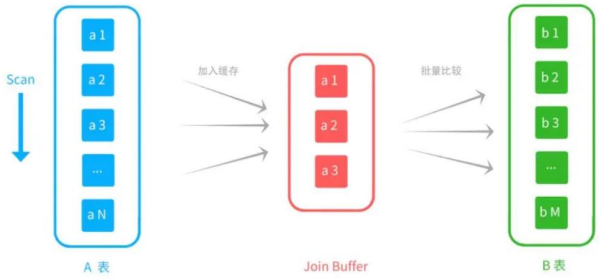
- 选择操作
  1. 小关系：全表扫描
  2. 大关系：
     ① 查询条件：主码==某个值 => 主码索引
     ② 查询条件：非主属性==某个值（该列有索引）
        => 选择率较小(<10%) 使用索引扫描,否则全表扫
     ③ 查询条件：非等值或范围（该列有索引）
        => 选择率较小(<10%)使用索引扫描 ,否则全表扫
     ④ AND连接的多个条件：
        => 复合属性：组合索引；索引分别检索求交集
        => 其他情况：全表扫
     ⑤ OR：全表扫

基于启发式规则的存取路径选择优化

- **连接操作**

1. 两个表均已排序 => Merge Join
2. 一个表在连接属性有索引 => 索引连接算法
3. 以上都不适用，且一个表较小 => Hash jion
4. 嵌套循环查询：小表（占用Block少）为外表（驱动表）





10.4. 2 物理优化 —— 基于代价的优化

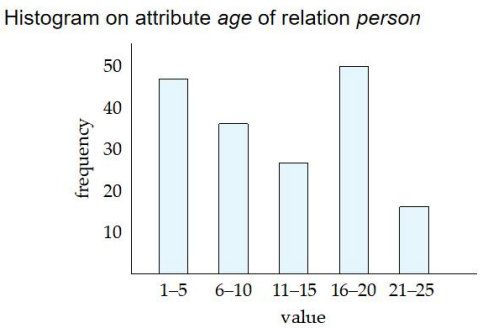
西北工业大学 NORTHWESTERN POLYTECHNICAL UNIVERSITY

- 统计信息
- 代价估算
- 优化方法

统计信息 => 查询计划选择依据

- 基本表：元组数、元组长度、占用块数、占用溢出块数等
- 基本表每列：不同值个数、max/min、索引信息、选择率[分布情况]
- 索引：不同索引值个数、索引的叶节点数。

```
mysql> show variables like 'innodb_stats%';
+---------------------------------------+-------------+
| Variable_name                         | Value       |
+---------------------------------------+-------------+
| innodb_stats_auto_recalc              | ON          |
| innodb_stats_include_delete_marked    | OFF         |
| innodb_stats_method                   | nulls_equal |
| innodb_stats_on_metadata              | OFF         |
| innodb_stats_persistent               | ON          |
| innodb_stats_persistent_sample_pages  | 20          |
| innodb_stats_transient_sample_pages   | 8           |
+---------------------------------------+-------------+
```

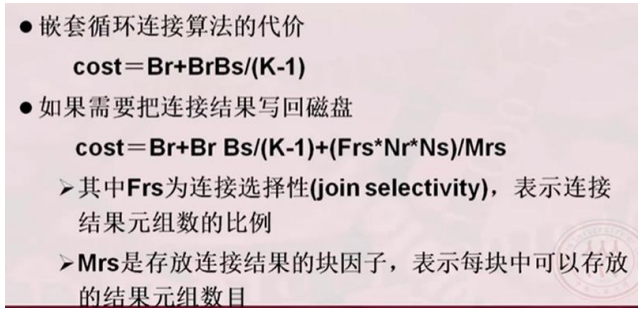


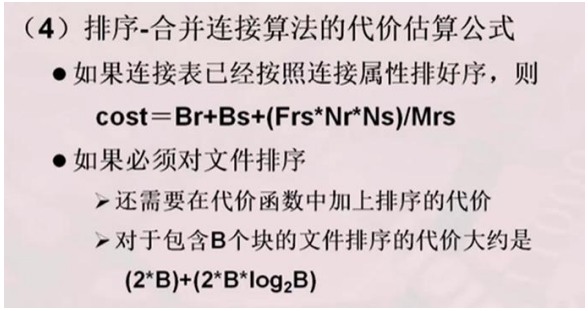
https://www.bilibili.com/video/BV1PJ411F78b?p=227

收集统计信息：ANALYZE语句

- 代价估计示例（具体估算参考课本或视频）

1. 全表扫描代价（如总块数为B，则cost=B/2[二分查找]）
2. 索引扫描代价
   ① 主属性索引扫描：若索引是L层的B+树，则cost=L+1；
   ② 非主属性索引扫描：cost=L+S（有S个元组满足条件）
   ③ 非等值条件：Cost=L + Y/2 + B/2
3. 嵌套循环连接
4. 归并排序





https://www.bilibili.com/video/BV1PJ411F78b?p=227

本章目录

➢ 查询计划的执行方式

- 迭代模型/火山模型：每次调用next接口返回一条数据
- 物化模型：每个operator一次处理所有输入，结果一次输出
- 向量化/批处理模型：以上两者的折中，每次调用next会返回一批元组

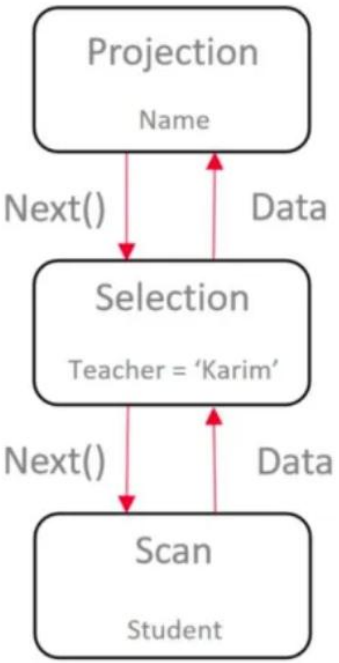
火山模型：将关系代数中每一种操作抽象为一个算子Operator，将整个SQL构建成一个Operator树，查询树自顶向下的调用next()接口，数据则自底向上的被拉取处理。

大多数关系型数据库都是使用该模型，如 SQLite、MongoDB、SQLServer、PostgreSQL、Oracle、MySQL等。

10.5. 查询计划执行

```
select name
from student
where teacher = 'Karim'
```

对应的火山模型

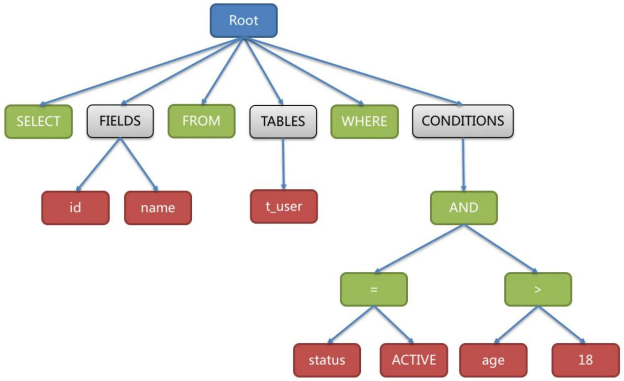



```
// Projection Next()
func (p Projection) Next(){
    row := p.children.Next()
    return row.name
}

// Selection Next()
func (s Selection) Next(){
    row := s.children.Next()
    if row != nil && row. teacher == 'Karim'{
        return row
    }
    return nil
}

// Scan Next()
func (s Scan) Next(){
    return student.ReadRow()
}
```

https://zhuanlan.zhihu.com/p/506202734

本章小结

- 查询处理的步骤
  - 查询分析（词法、语法、语义）
  - 查询处理（选择算子、连接算子）
  - 查询优化（代数优化、物理优化）
  - 查询计划执行（火山模型）



- ✓ 嵌套循环算法（Nested Loop Join, NLJ）
- ✓ 归并排序算法（Merged Join）
- ✓ 索引连接算法（Index Join）
- ✓ 散列连接算法(Hash join)

本章小结

➢ 本章作业

- MiniOB训练营第1、13题（选做）
  - 第1题：环境搭建与代码调试运行
  - 第13题：熟悉基本Select语句的解析与执行
- 阅读开源DBMS中的NLJ相关的实现代码（选做）

https://www.oceanbase.com/video/9000660

https://github.com/oceanbase/miniob