# Query Optimization

汇报人：朱道冰

2022-09-19

DSC
分布式存储与计算实验室

# 目录

# 01 ／ 背景

# 01 背景

## 1.1 为什么要有查询优化

### 1．声明式SQL语句

用户只需说明他们想做什么，具体的执行方式由数据库系统决定。

### 2．SQL语句分类

#### 1．数据操作语言(DML：Data Manipulation Language)

由数据库管理系统(DBMS) 提供，用于让用户或程序员使用，实现对数据库中数据的操作。 主要包含 **SELECT**、 INSERT、 UPDATE、 DELETE、 MERGE、 CALL、 EXPLAIN PLAN、 LOCK TABLE等语句。

#### 2．数据定义语言(DDL：Data Definition Language)

用于定义SQL模式、基本表、视图和索引的创建和撤消操作。 主要包含 CREATE、 ALTER、 DROP、 TRUNCATE、 COMMENT、 REPLACE(RENAME)等语句，一般不需要commit等事务操作。

#### 3．数据控制语言(DCL：Data Control Language)

用于数据库授权、角色控制等管理工作。 主要包含 GRANT、 REVOKE等语句。

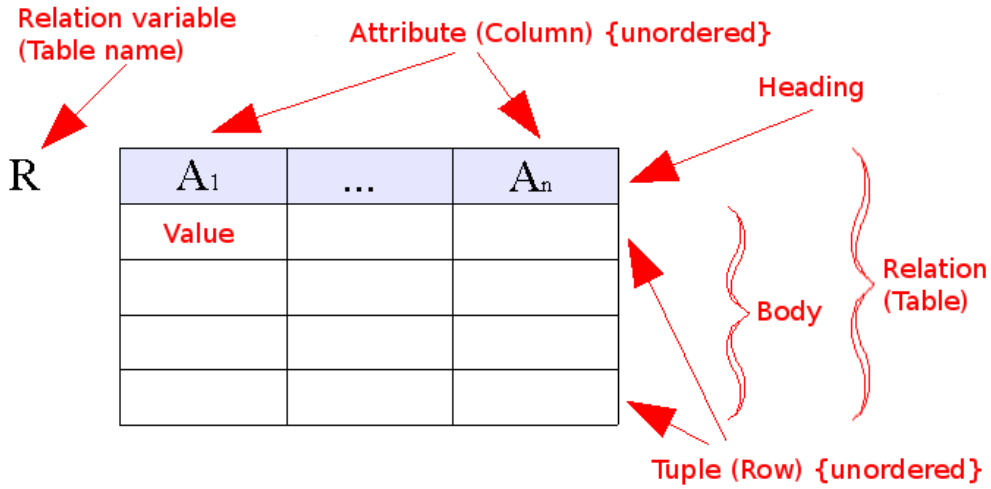
#### 4．事务控制语言(TCL：Transaction Control Language)

用于数据库的事务管理。 主要包含 SAVEPOINT、 ROLLBACK、 COMMIT、 SET TRANSACTION等语句。

# 01 背景

## 1.2 关系模型

- 存储模型
  - 关系型：行存、列存
  - 非关系型：键值对、图、文档、时序




| Customer ID | Tax ID | Name | Address | [More fields...] |
|---|---|---|---|---|
| 1234567890 | 555-5512222 | Ramesh | 323 Southern Avenue | ... |
| 2223344556 | 555-5523232 | Adam | 1200 Main Street | ... |
| 3334445563 | 555-5533323 | Shweta | 871 Rani Jhansi Road | ... |
| 4232342432 | 555-5325523 | Sarfaraz | 123 Maulana Azad Sarani | ... |

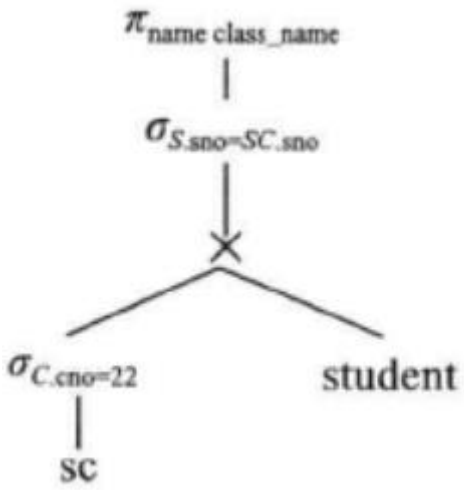
# 01 背景

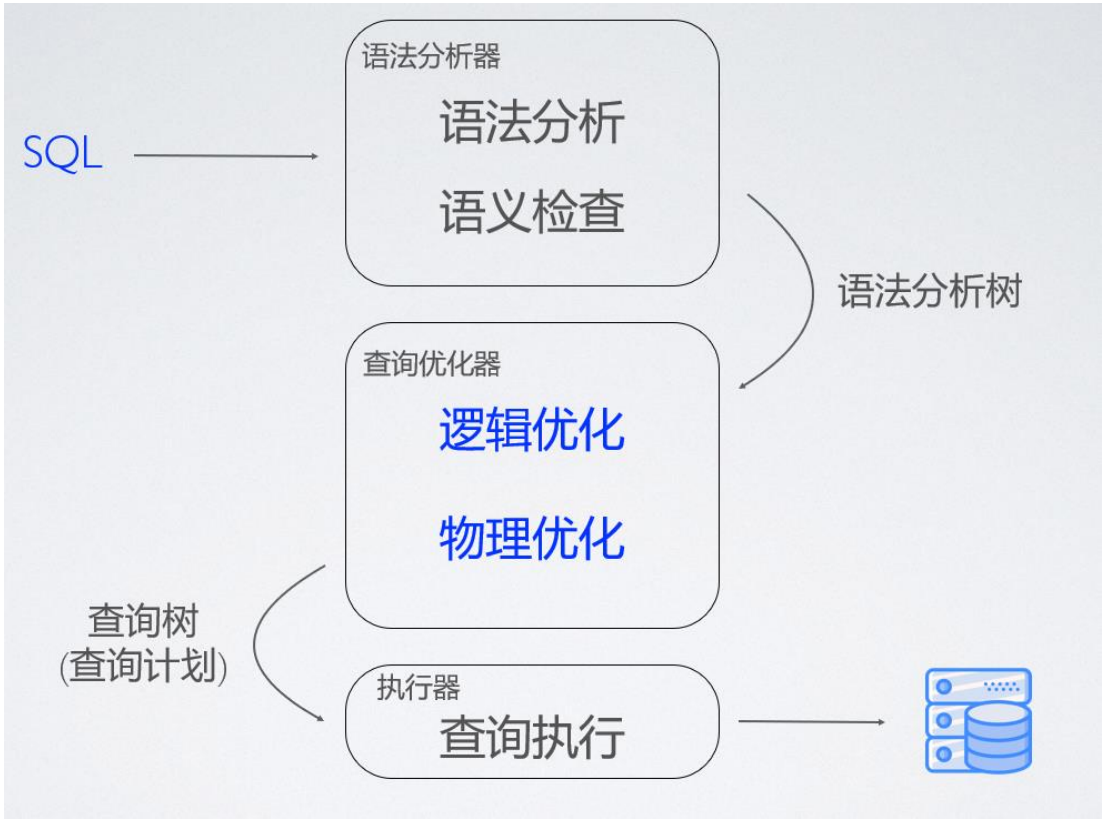
## 1.3 查询优化流程

SQL输入

```
SELECT name, class_name
FROM student S, course C, sc SC
WHERE S.sno=SC.sno AND C.cno=SC.cno AND C.cno=22;
```

优化后的物理执行计划



# 01 背景

## 1.3 查询优化流程

1．SQL输入

数据库接受用户输入的SQL语句（字符串）

```
SELECT name, class_name
FROM student S, course C, sc SC
WHERE S.sno=SC.sno AND C.cno=SC.cno AND C.cno=22;
```

2．语法分析

对输入的SQL语句进行词法分析，语法分析，得到语法分析树

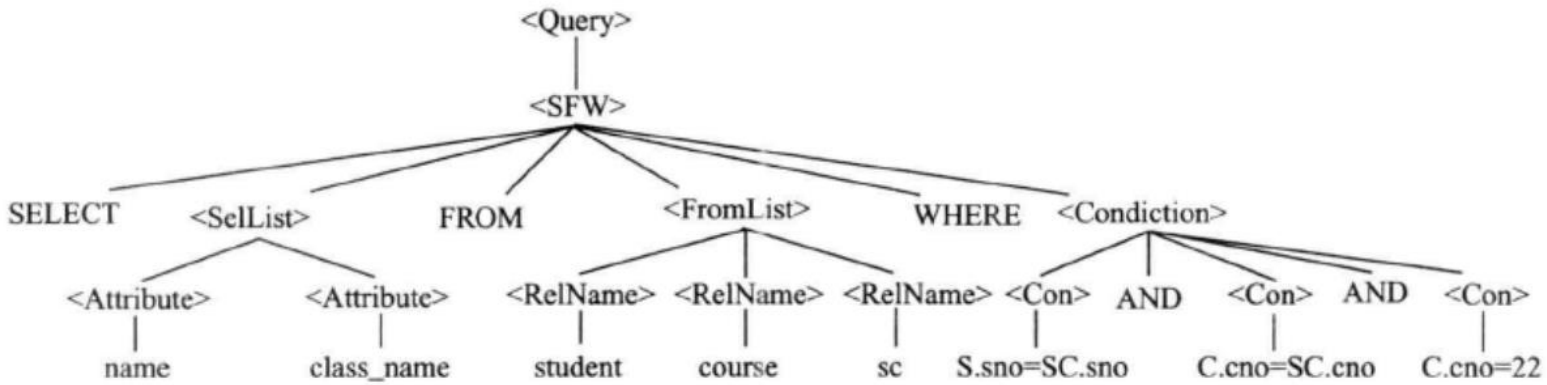



图 4-1　语法分析树

# 01 背景

## 1.3 查询优化流程

### 3．语义检查

根据系统元信息，对语法树进行语义检查。

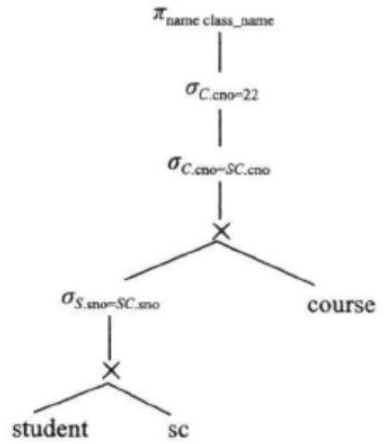



图 4-2　关系代数语法树

### 4．逻辑优化

把词法分析树变为关系代数语法树，并应用逻辑优化规则生成逻辑执行计划。

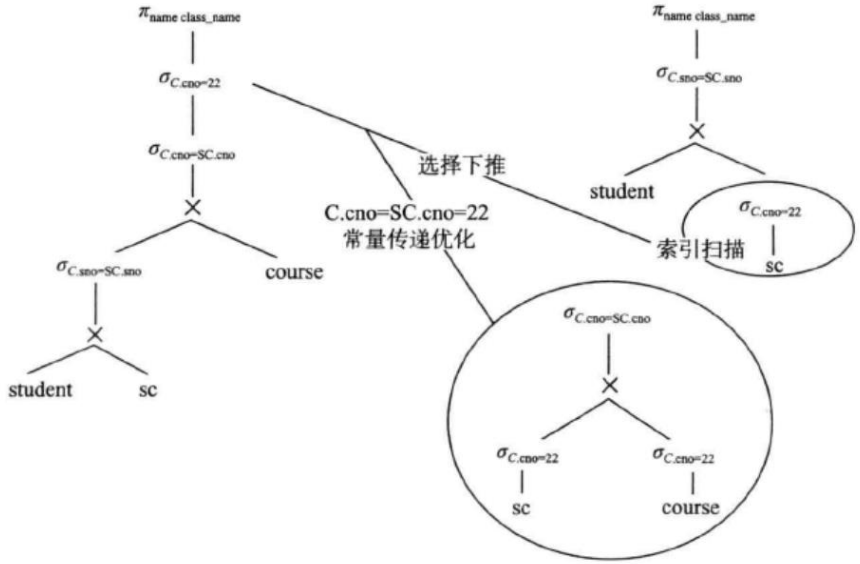



图 4-3　逻辑查询优化过程样式

# 01 背景

## 1.3 查询优化流程

### 5．物理优化

对逻辑执行计划进行改造：

调整连接顺序，选择具体的物理算子

生成物理执行计划。

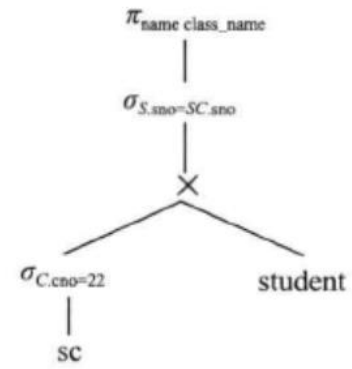
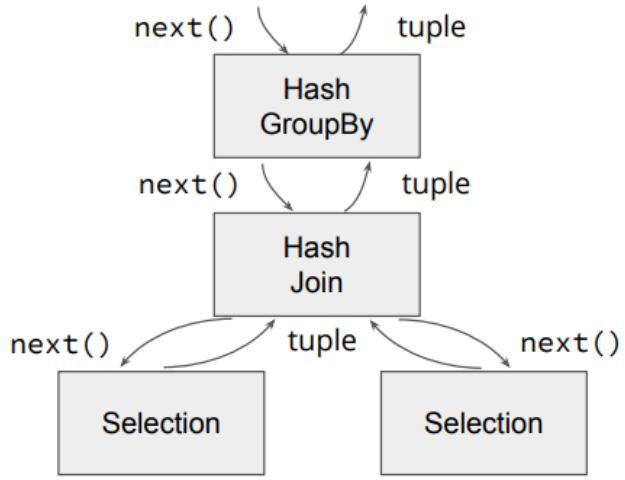



图 4-4　查询树（查询执行计划）

### 6．　查询执行

依据物理执行计划执行查询。



# 02 语法分析&语义检查

# 02 语法分析&语义检查

## 2.1 编译原理

1. 词法分析（lexical analysis）

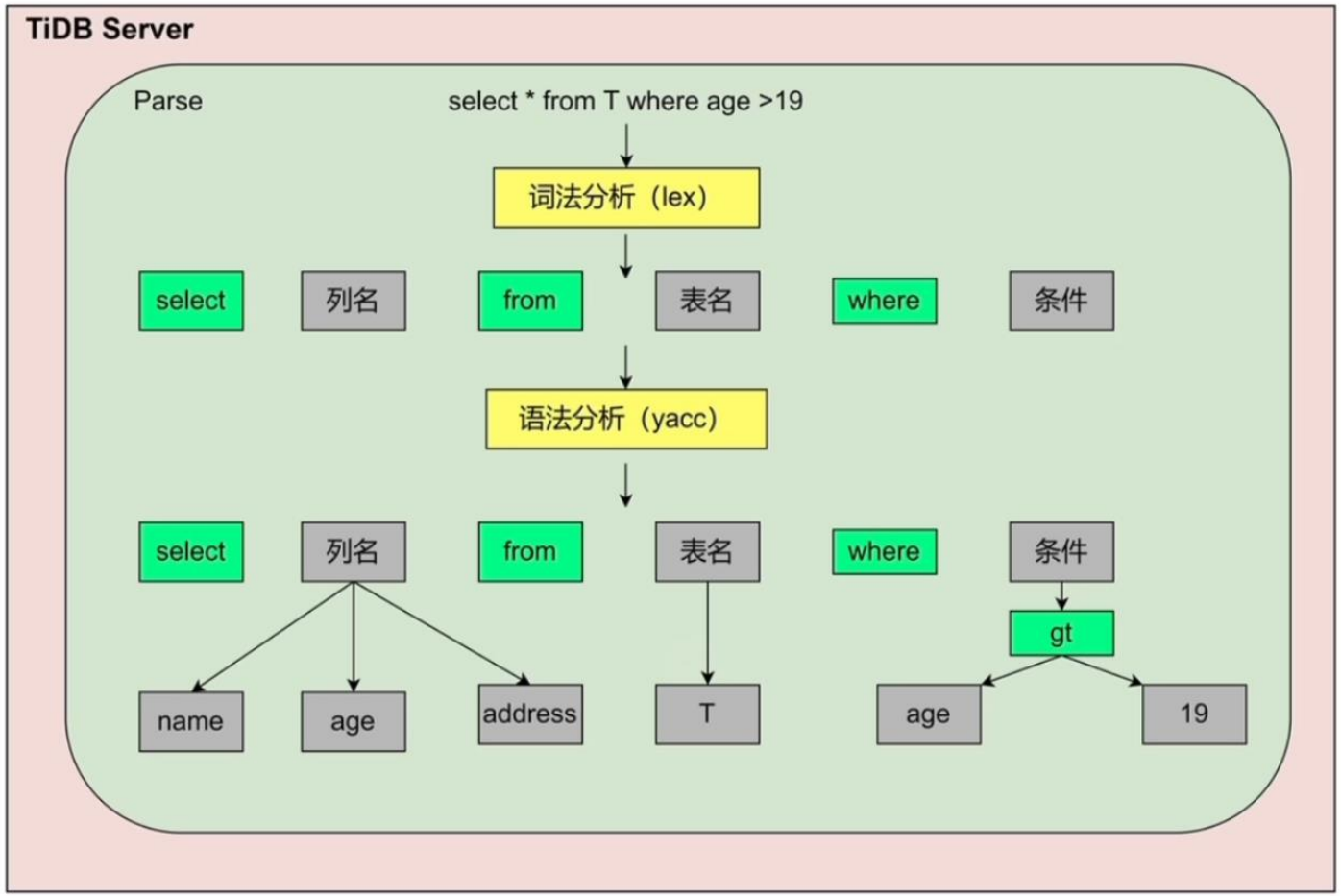
提取编程语言占用的各种保留字、操作符等等语言的元素。

2.语法分析（syntactic analysis）

将词法单元转换为语法分析树。

BNF范式

```
select:
      SELECT [ hintComment ] [ STREAM ] [ ALL | DISTINCT ]
          { * | projectItem [, projectItem ]* }
      FROM tableExpression
      [ WHERE booleanExpression ]
      [ GROUP BY [ ALL | DISTINCT ] { groupItem [, groupItem ]* } ]
      [ HAVING booleanExpression ]
      [ WINDOW windowName AS windowSpec [, windowName AS windowSpec ]* ]
```



# 02 语法分析&语义检查

## 2.1 编译原理

1. 词法分析

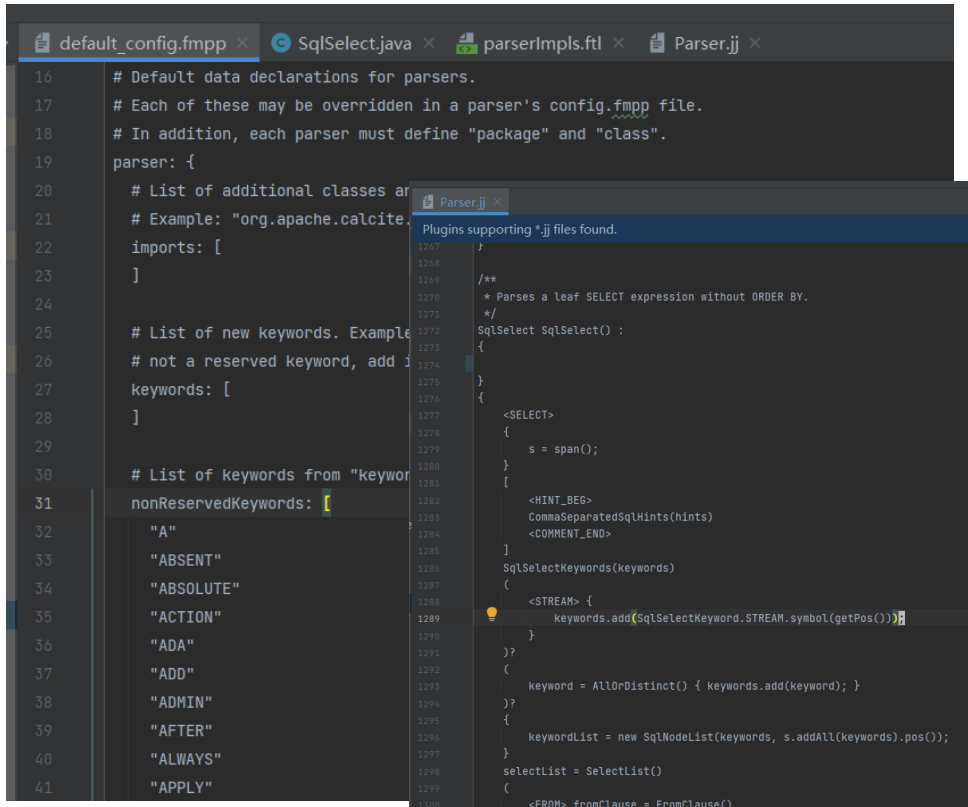
提取编程语言占用的各种保留字、操作符等等语言的元素。

2.语法分析

BNF范式

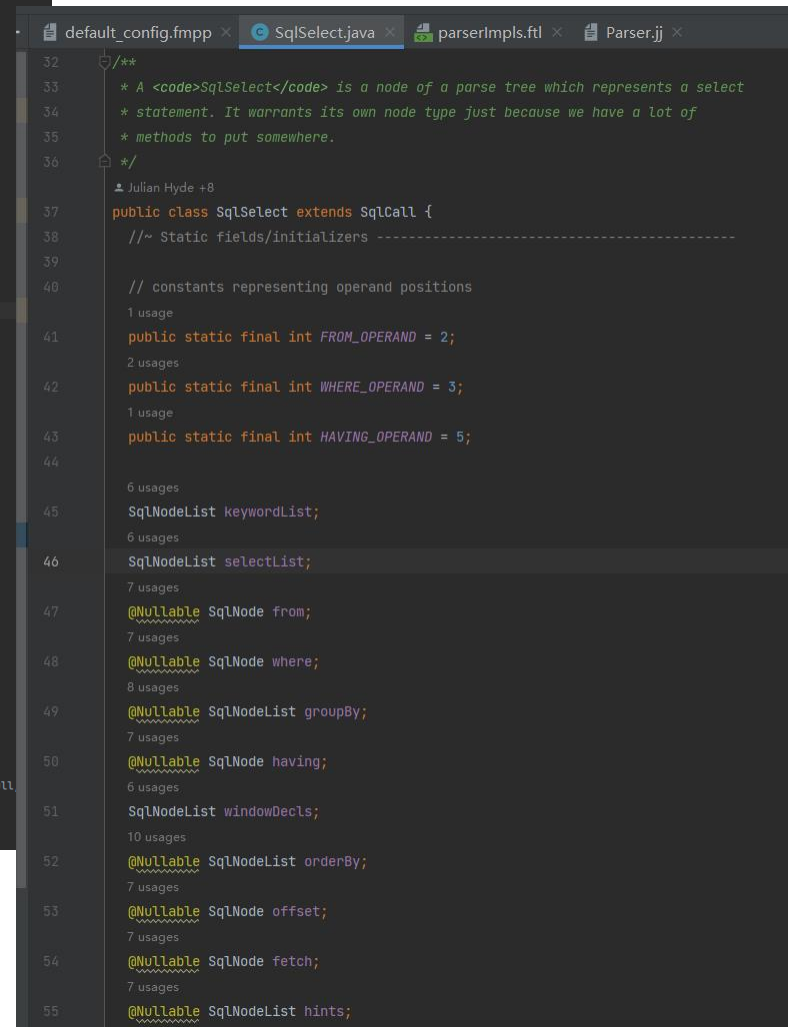
```
select:
      SELECT [ hintComment ] [ STREAM ] [ ALL | DISTINCT ]
          { * | projectItem [, projectItem ]* }
      FROM tableExpression
      [ WHERE booleanExpression ]
      [ GROUP BY [ ALL | DISTINCT ] { groupItem [, groupItem ]* } ]
      [ HAVING booleanExpression ]
      [ WINDOW windowName AS windowSpec [, windowName AS windowSpec ]* ]
```

https://calcite.apache.org/docs/reference.html









# 语法分析&语义检查

## 2.2 常用框架

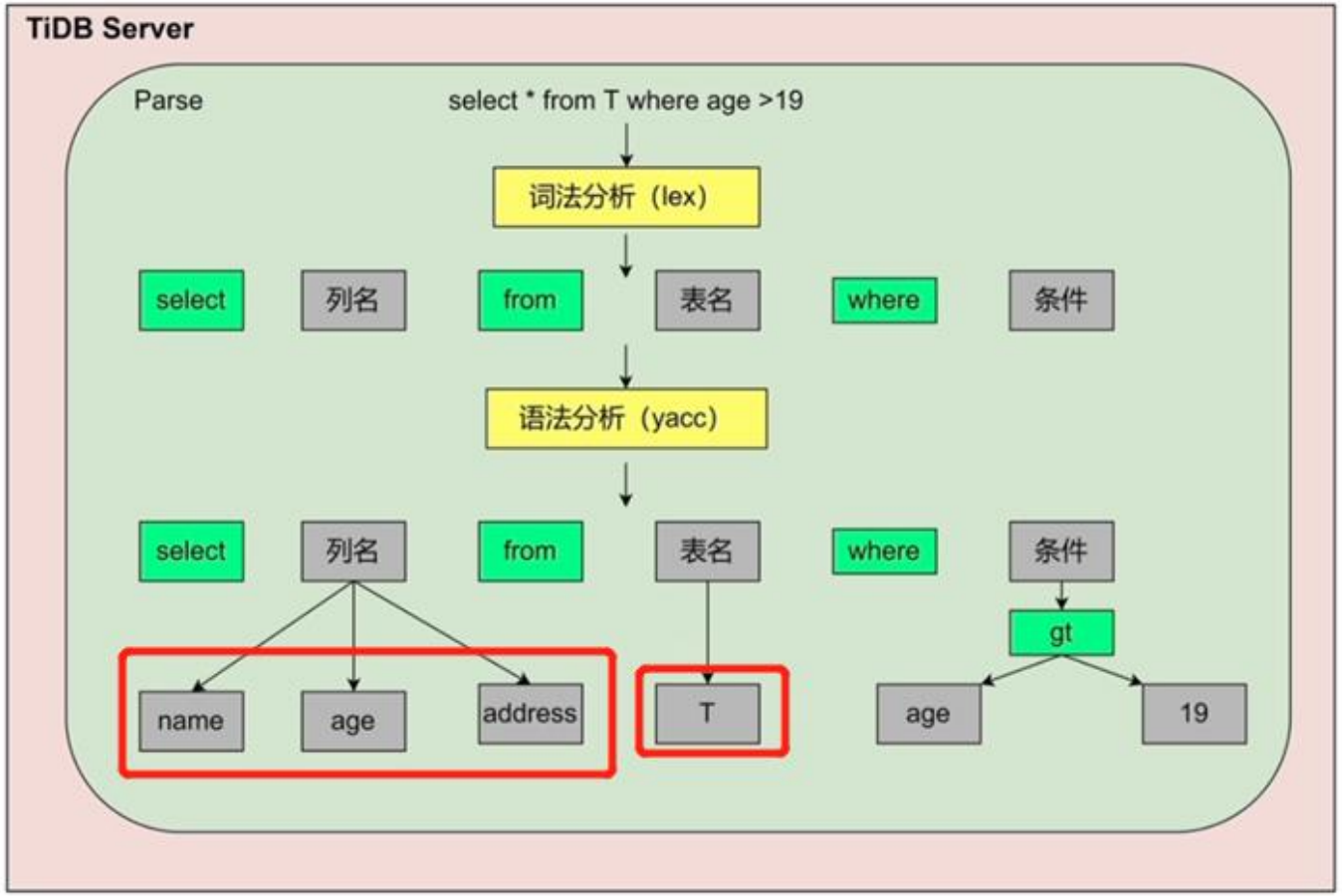
| 类型 | 工具 | 特点 |
| --- | --- | --- |
| 词法分析 | Lex(LEXical compiler) | |
| | Flex | Lex的开源版本 |
| 语法分析 | Yacc(Yet Another Compiler-Compiler) | 自底向上 |
| | Bison | GNU Bison 是 Yacc 的 GNU 自由软件版本 |
| 词法分析+语法分析 | JavaCC | 自顶向下，java届的 Yacc + Lex 或 Bison + Flex |
| | Antlr (ANother Tool for Language Recognition) | |

# 语法分析&语义检查

## 2.3 语义检查

1．语义检查

查语句中的数据库对象，如关系名、属性名是否存在和有效。



# 03 逻辑优化

# 03 逻辑优化

## 3.1 关系代数等价变换规则

逻辑优化 → 关系代数等价变换规则 → 查询重写

| 传统运算符 | 专有运算符 |
| --- | --- |
| 并 | 选择 |
| 交 | 投影 |
| 差 | 连接 |
| 积 | 除 |

# 03 逻辑优化

## 3.1 关系代数等价变换规则

并

```
Select * from R union Select * from S
```

交

```
Select * from R where kr not in (
Select kr from R where kr not in (
Select ks from S))
```

差

```
Select * from R where kr not in (Select ks from S)
```

积

```
Select R.* , S.* from R , S
```

# 03 逻辑优化

## 3.1 关系代数等价变换规则

选择

```
Select * from R where condition
```

投影

```
Select col_1,col_2+2 from R
```

连接

```
Select r.col_1,s.col_2 from R,S where condition
```

除

```
Select Distinct r1.x from R,r1 where not exists (
Select S.y from S Where not exists (
Select * from R r2 where r2.x=r1.x and r2.y=S.y))
```

# 03 逻辑优化

## 3.1 关系代数等价变换规则

| 规则名称 | 公式 | 优化意义 |
| --- | --- | --- |
| 连接，笛卡尔积的结合 | (E1xE2)xE3 ≡ E1x(E2xE3)<br>(E1⋈E2)⋈E3 ≡ E1⋈(E2⋈E3) | 减小中间关系的大小 |
| 选择的串联 | σf1(σf2(E)) ≡ σf1∧f2(E)) | 合并选择条件是得可以一次就检查全部条件，不必多次过滤元组 |
| 选择与并的分配 | σf(E1∪E2) ≡<br>σf(E1)∪σf(E2)<br>E1E2有相同的列名 | 条件下推到相关的关系上，可以减小中间结果的大小 |
| 选择与笛卡尔积的分配 | σf(E1xE2) ≡ σf(E1)xE2<br>f中涉及的属性都是E1中的属性 | |
| 投影与并的分配 | ∏A1,A2,…,An(E1∪E2) ≡<br>∏A1,A2,…,An(E1)∪∏A1,A2,…,An(E2)<br>E1E2有相同的列名 | 先投影后并，可以减少做并前每个元组的长度 |

# 03 逻辑优化

## 3.2 常见的逻辑优化规则

### 1.常量传递

“select * from t1,t2 where t1.a = 5 AND t2.b > t1.a;”

→ “select * from t1,t2 where t1.a = 5 AND t2.b > **5**;”

### 2. 等值传递

“select * from t1,t2,t3 where t1.a = 5 AND t2.b = t1.a AND t3.c = t2.b;”

→ “select * from t1,t2 where t1.a = 5 AND t2.b = **5** AND t3.c = **5** ;”

### 3.提取公共谓词

“select * from t where (t.c2>18 or t.c1=‘f’) and (t.c2>18 or t.c2>15);”

→ “select * from t where (t.c1=‘f’ and t.c2>15) or t.c2>18;”

# 03 逻辑优化

## 3.1 常见的逻辑优化规则

### 4. 用inner join 替换 outer join

“select * from t1 left join t2 on t1.a=t2.a where t2.a is not null;”

→ “select * from t1,t2 where t1.a=t2.a and t2.a is not null;”

```
select * from a_table a left join b_table b on a.a_id = b.b_id;
```

a_table(+)

select * from a_table a left join b_table b on a.a_id = b.b_id | Enter a SQL expression to filter re

| | a_id | a_name | a_part | b_id | b_name | b_part |
|---|---|---|---|---|---|---|
| 1 | 2 | 老王 | 秘书部 | 2 | 老王 | 秘书部 |
| 2 | 3 | 老张 | 设计部 | 3 | 老张 | 设计部 |
| 3 | 1 | 老潘 | 总裁部 | [NULL] | [NULL] | [NULL] |
| 4 | 4 | 老李 | 运营部 | [NULL] | [NULL] | [NULL] |



### 5.子查询表优化

“select * from t1 where a IN (select a from t2 where a =1 );”

→ “select * from t1 inner join t2 where t1.a=t2.a and t2.a =1;”

“t.id < all(select s.id from s);”

-> t.id < min(s.id) and if(sum(s.id is null) != 0, null, true);”

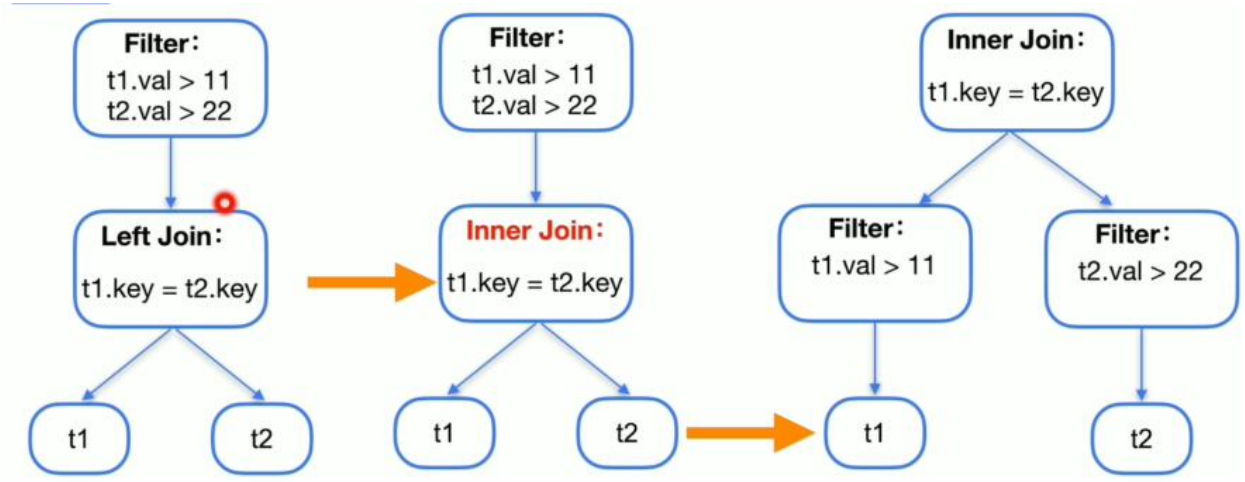
# 03 逻辑优化

## 3.1 常见的逻辑优化规则

**6．谓词下推**

**减少需要计算的数据量**

**减少需要传输的数据量**




```
create table t(id int primary key, a int);
explain select * from t where a < 1;
+-------------------------+----------+-----------+---------------+--------------------------------+
| id                      | estRows  | task      | access object | operator info                  |
+-------------------------+----------+-----------+---------------+--------------------------------+
| TableReader_7           | 3323.33  | root      |               | data:Selection_6               |
| └─Selection_6           | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:pseudo |
+-------------------------+----------+-----------+---------------+--------------------------------+
3 rows in set (0.00 sec)
```

**7．列裁剪**

**8．TopN和limit下推**

**9．分区裁剪**

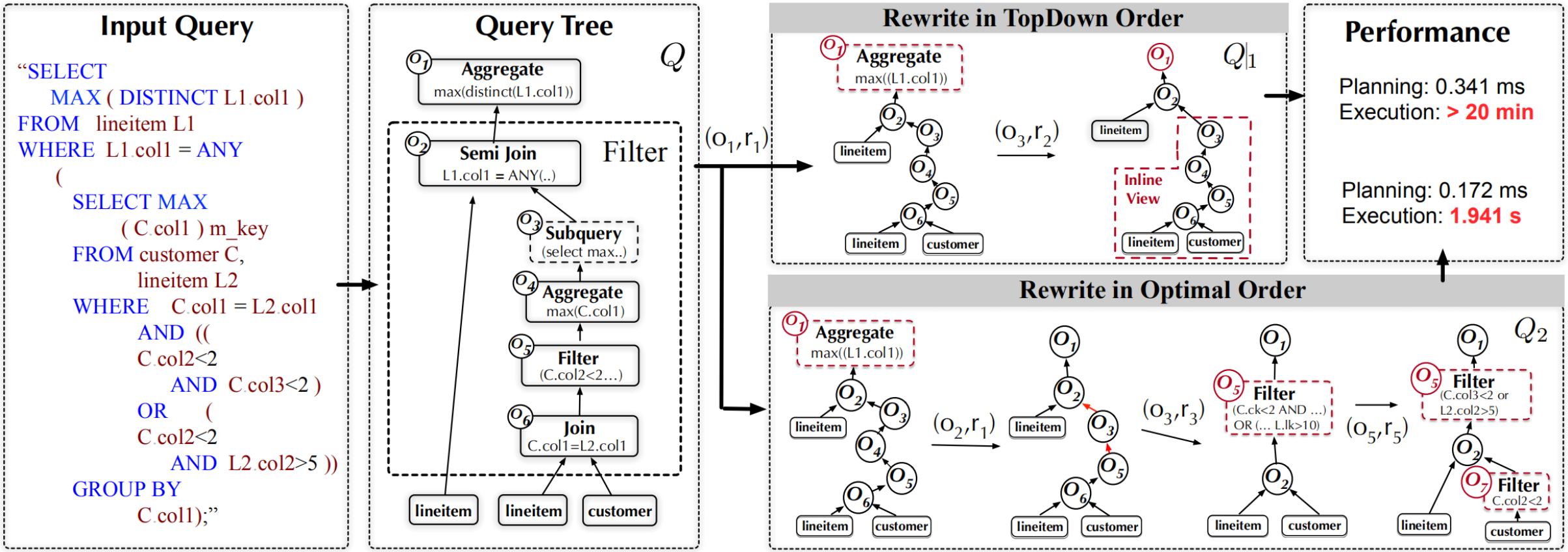
**…**

**https://docs.pingcap.com/zh/tidb/stable/sql-logical-optimization**

# 03 逻辑优化

## 3.2 规则应用顺序

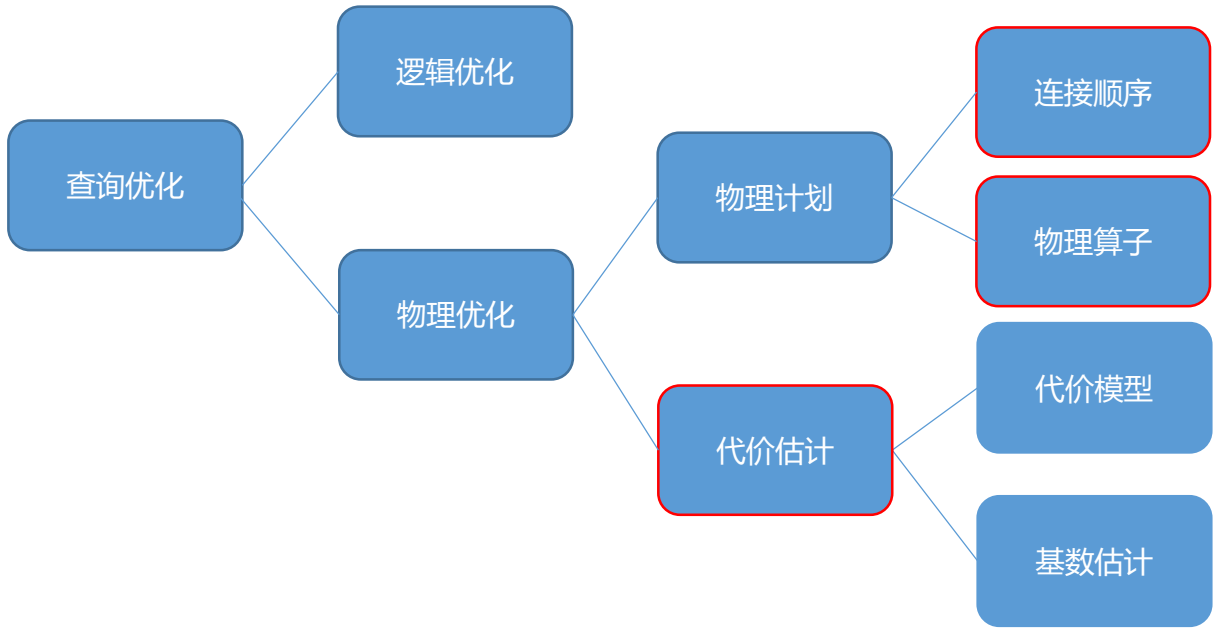
### 1.最终执行计划的好坏与规则应用顺序有关



# 04 物理优化

# 04 物理优化

## 4.1 物理优化流程




物理优化 → 代价模型 → 连接顺序<br>物理算子

# 04 物理优化

## 4.1 物理优化流程

物理优化 → 代价模型 → 连接顺序<br>物理算子

代价估计 = 基数估计 + 代价模型

总代价 = IO代价 + CPU代价

cost = P * a_page_cpu_time + T * W

P：预计访问页面数　a_page_cpu_time：每个页面读取时间　T：访问元组数　W：权重因子

# 04 物理优化

## 4.3 join reorder

1. 多表连接顺序 O(N!) NP问题

2. 启发式规则限制搜索空间(SystemR)
   a)只考虑左深树
   b)在选择下一个join table时优先选择有join条件的从而将笛卡尔积放到后面计算。

**基于动态规划的计划搜索**
优点：穷举类型的算法。适合查询中包含较少关系的搜索，可得到全局最优解。

缺点：搜索空间随关系个数增长呈指数增长。

| 层级 | 说明 | 产生的结果 |
| --- | --- | --- |
| 4 | 第四层通过第三层与第一层关联和第二层与第二层关联得到 | {A, B, C, D}, {A, B, D, C} {B, A, C, D}, {B, A, D, C} {A, C, B, D}, {A, C, D, B} {C, A, B, D}, {C, A, D, B} {A, D, B, C}, {A, D, C, B} {D, A, B, C}, {D, A, C, B} {B, C, A, D}, {B, C, D, A} {C, B, A, D}, {C, B, D, A} {B, D, A, C}, {B, D, C, A} {D, B, A, C}, {D, B, C, A} {C, D, B, A}, {C, D, A, B} {D, C, B, A}, {D, C, A, B}, {A, B} \| {C, D}, {A, B} \| {D, C} {B, A} \| {C, D}, {B, A} \| {D, C} {C, D} \| {A, B}, {C, D} \| {B, A} {D, C} \| {A, B}, {D, C} \| {B, A} |
| 3 | 第三层通过第二层与第一层关联得到 | {A, B, C}, {A, B, D} {B, A, C}, {B, A, D} {A, C, B}, {A, C, D} {C, A, B}, {C, A, D} {A, D, B}, {A, D, C} {D, A, B}, {D, A, C} {B, C, A}, {B, C, D} {C, B, A}, {C, B, D} {B, D, A}, {B, D, C} {D, B, A}, {D, B, C} {C, D, B}, {C, D, A} {D, C, B}, {D, C, A} |
| 2 | 第二层通过第一层关联得到 | {A, B}, {B, A}, {A, C}, {C, A}, {A, D}, {D, A}, {B, C}, {C, B}, {B, D}, {D, B}, {C, D}, {D, C} |
| 1 | 树叶，初始层 | {A}, {B}, {C}, {D} |

https://icode.best/i/68785034949832

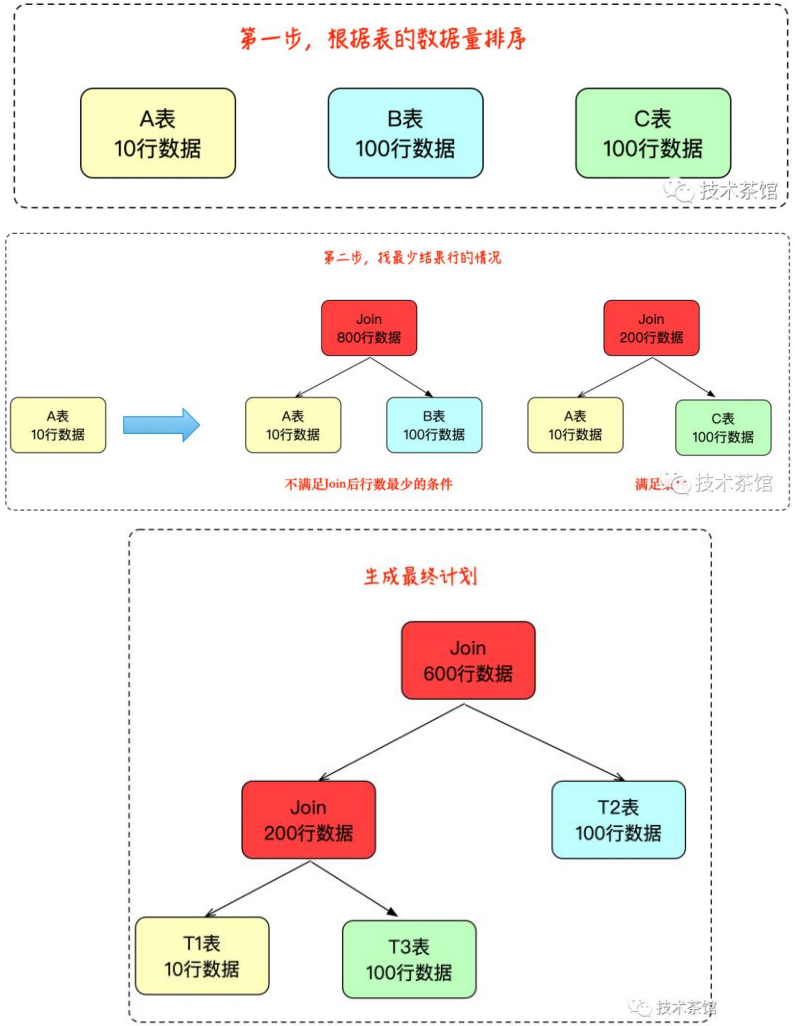
# 04 物理优化

## 4.3 join reorder

**贪心**
每次选代价最小的两个表/中间结果

优点：非穷举类型的算法。适合解决较多关系的搜索。

缺点：得到局部最优解。




https://icode.best/i/68785034949832

# 04 物理优化

## 4.3 join reorder

| 算法名称 | 特点与适用范围 | 缺点 |
| --- | --- | --- |
| 启发式算法 | 适用于任何范围，与其他算法结合，能有效提高整 体效率 | 不知道得到的解是否最优 |
| 贪婪算法 | 非穷举类型的算法。适合解决较多关系的搜索 | 得到局部最优解 |
| 爬山法 | 适合查询中包含较多关系的搜索，基于贪婪算法 | 随机性强，得到局部最优解 |
| 遗传算法 | 非穷举类型的算法。适合解决较多关系的搜索 | 得到局部最优解 |
| 动态规划算法 | 穷举类型的算法。适合查询中包含较少关系的搜索， 可得到全局最优解 | 搜索空间随关系个数增长呈指数增长 |
| System R优化 | 基于自底向上的动态规划算法，为上层提供更多可 能的备选路径，可得到全局最优解 | 搜索空间可能比动态规划算法更大一些 |

https://icode.best/i/68785034949832

| Join Reorder算法 | 默认使用的数据库 | INFO |
| --- | --- | --- |
| 单序列贪心启发式Join Reorder算法 | TiDB | Left Deep Tree |
| 多序列贪心启发式Join Reorder算法 | Flink, Drill, PolarDB-X | Left Deep Tree，可以比较N个Join序列，选出代价较低的 |
| 遗传算法 | PostgreSQL | 只有表数目大于12张时候才会开启，每次Join顺序不稳定 |
| 深度优先枚举Join Reorder算法 | MySQL | Left Deep Tree，表数目<=7张算法复杂度为N!，没有利用动态规划 |
| Bottom-Up枚举的Join Reorder算法 | PostgreSQL, OceanBase | Bottom Up动态规划枚举Left Deep Tree或Bushy Join Tree搜索空间 |
| 基于规则变换Top-Down枚举的Join Reorder算法 | PolarDB-X, SQLServer, CockroachDB | TopDown动态规划枚举Left Deep Tree或Bushy Join Tree，可以做空间剪枝，与其他优化规则一起混合做全局优化 |

https://zhuanlan.zhihu.com/p/470139328

# 04 物理优化

## 4.4 物理算子

### 1. 读表算子

**读表算子**

| 读表算子 | 触发条件 | 适用场景 | 说明 |
| --- | --- | --- | --- |
| PointGet/BatchPointGet | 读表的范围是一个或多个单点范围 | 任何场景 | 如果能被触发，通常被认为是最快的算子，因为其直接调用 kvget 的接口进行计算，不走 coprocessor |
| TableReader | 无 | 任何场景 | 从 TiKV 端直接扫描表数据，一般被认为是效率最低的算子，除非在 `_tidb_rowid` 这一列上存在范围查询，或者无其他可以选择的读表算子时，才会选择这个算子 |
| TableReader | 表在 TiFlash 节点上存在副本 | 需要读取的列比较少，但是需要计算的行很多 | TiFlash 是列式存储，如果需要对少量的列和大量的行进行计算，一般会选择这个算子 |
| IndexReader | 表有一个或多个索引，且计算所需的列被包含在索引里 | 存在较小的索引上的范围查询，或者对索引列有顺序需求的时候 | 当存在多个索引的时候，会根据估算代价选择合理的索引 |
| IndexLookupReader | 表有一个或多个索引，且计算所需的列**不完全**被包含在索引里 | 同 IndexReader | 因为计算列不完全被包含在索引里，所以读完索引后需要回表，这里会比 IndexReader 多一些开销 |

https://docs.pingcap.com/zh/tidb/stable/choose-index#%E8%AF%BB%E8%A1%A8%E7%AE%97%E5%AD%90

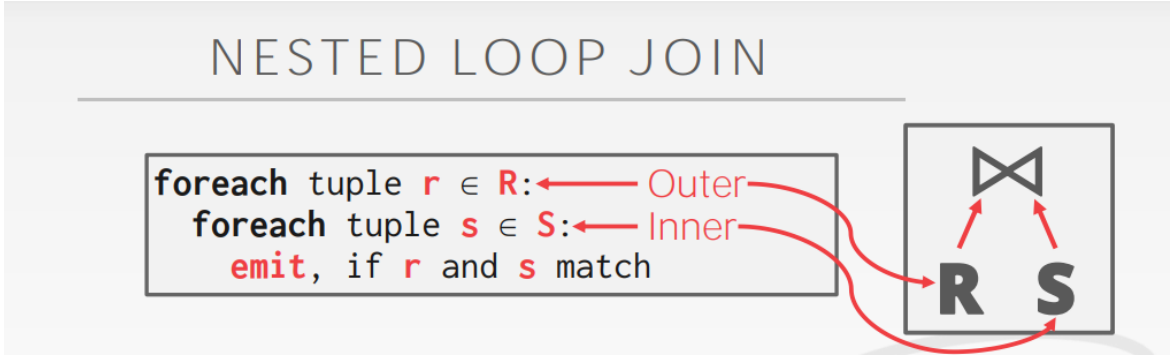
# 04 物理优化

4.4 物理算子

2. Join 算子

Nested loop join

思考：如何优化?




Why is this algorithm stupid?

→ For every tuple in **R**, it scans **S** once

**Cost: M + (m · N)**

R(id,name)

| id | name |
|---|---|
| 600 | MethodMan |
| 200 | GZA |
| 100 | Andy |
| 300 | ODB |
| 500 | RZA |
| 700 | Ghostface |
| 400 | Raekwon |

*M* pages
*m* tuples

S(id,value,cdate)

| id | value | cdate |
|---|---|---|
| 100 | 2222 | 10/7/2020 |
| 500 | 7777 | 10/7/2020 |
| 400 | 6666 | 10/7/2020 |
| 100 | 9999 | 10/7/2020 |
| 200 | 8888 | 10/7/2020 |

*N* pages
*n* tuples

20)

图源cmu 15-445

# 04 物理优化

## 4.4 物理算子

### 2. Join 算子

SORT MERGE JOIN

思考：如果内存放不下怎么处理?

## SORT-MERGE JOIN

Sort Cost (R): $2M \cdot (1 + \lceil \log_{B-1} \lceil M / B \rceil \rceil)$

Sort Cost (S): $2N \cdot (1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil)$

Merge Cost: $(M + N)$

**Total Cost: Sort + Merge**

## SORT-MERGE JOIN

```
sort R,S on join keys
cursor_R ← R_sorted, cursor_S ← S_sorted
while cursor_R and cursor_S:
  if cursor_R > cursor_S:
    increment cursor_S
  if cursor_R < cursor_S:
    increment cursor_R
  elif cursor_R and cursor_S match:
    emit
    increment cursor_S
```



## SORT-MERGE JOIN

R(id,name)

| id | name |
|---|---|
| 100 | Andy |
| 200 | GZA |
| 200 | GZA |
| 300 | ODB |
| 400 | Raekwon |
| 500 | RZA |
| 600 | MethodMan |
| 700 | Ghostface |

S(id,value,cdate)

| id | value | cdate |
|---|---|---|
| 100 | 2222 | 10/7/2020 |
| 100 | 9999 | 10/7/2020 |
| 200 | 8888 | 10/7/2020 |
| 400 | 6666 | 10/7/2020 |
| 500 | 7777 | 10/7/2020 |

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

这边是从200到200，发现值一样，那么这个200需要倒着检查回去直到比它小的值100；

如果还有一个200，那么400到100这个过程还是要检查一次的

Output Buffer

| R.id | R.name | S.id | S.value | S.cdate |
|---|---|---|---|---|
| 100 | Andy | 100 | 2222 | 10/7/2020 |
| 100 | Andy | 100 | 9999 | 10/7/2020 |
| 200 | GZA | 200 | 8888 | 10/7/2020 |



图源cmu 15-445

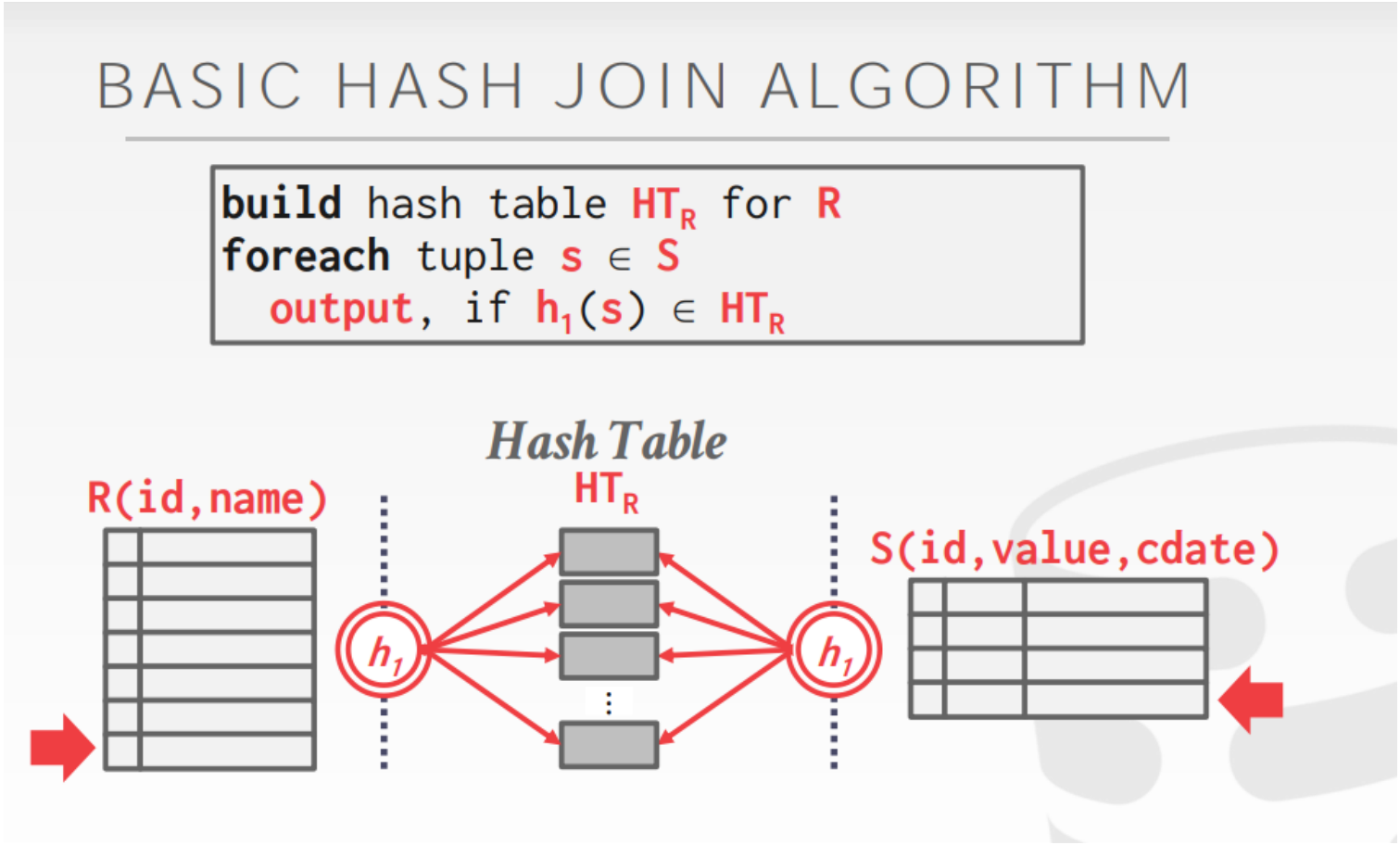
# 04 物理优化

## 4.4 物理算子

### 2. Join 算子

HASH JOIN

思考:
a)如果内存放不下小表怎么处理?
b)小表和大表哪个作为build表?
c)开销如何计算?




图源cmu 15-445

# 04 物理优化

## 4.5 统计信息

### 1. 直方图

误差来源于均匀分布（Uniform assumption）

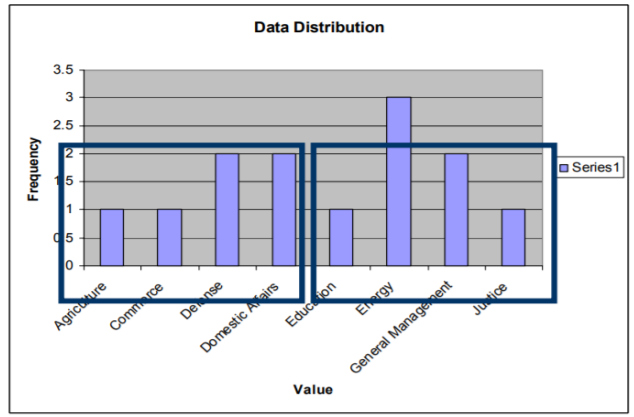
```
SELECT *
FROM employee as e
WHERE e.Department=Agriculure;
```

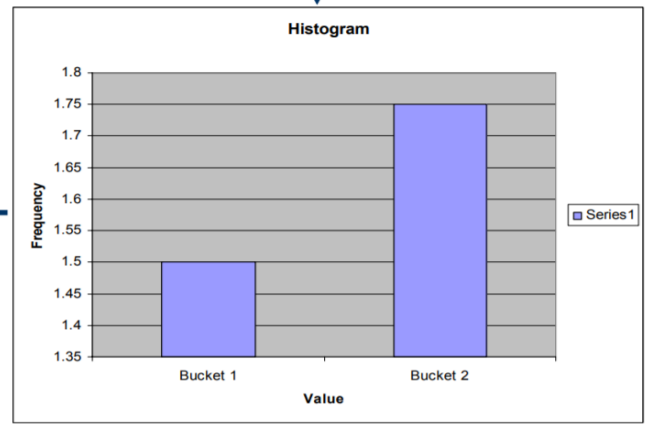
Selectivity = 1.5/6 = ¼

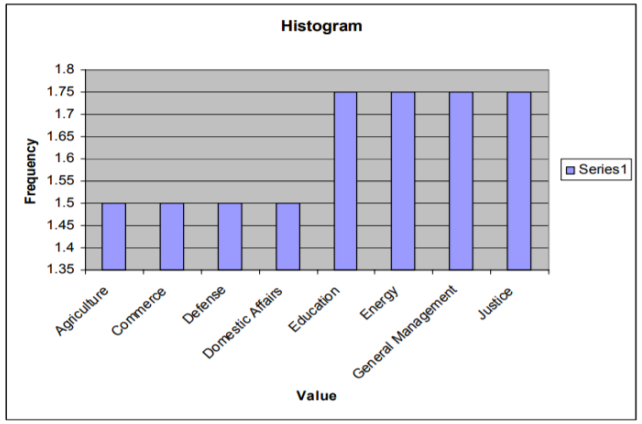
Rows = C(e) * selectivity

比实际多

| Department | Histogram H1 Frequency in Bucket | Histogram H1 Approximate Frequency |
|---|---|---|
| Agriculture | 1 | 1.5 |
| Commerce | 1 | 1.5 |
| Defense | 2 | 1.5 |
| Domestic Affairs | 2 | 1.5 |
| Education | 1 | 1.75 |
| Energy | 3 | 1.75 |
| General Management | 2 | 1.75 |
| Justice | 1 | 1.75 |



# 04 物理优化

## 4.5 统计信息

### 2. Most common values

右图以PG为例

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  | {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

```
selectivity = mcf[3]
            = 0.003

rows = 10000 * 0.003
     = 30
```

https://www.postgresql.org/docs/13/row-estimation-examples.html

# 04 物理优化

## 4.5 统计信息

### 3. 更新

统计信息生成、更新时机

- 手动（VACUUM，ANALYZE）
- 触发(a few DDL cmds:CREATE INDEX)/定时

统计信息生成、更新方式

- 随机采样生成统计信息（PostgreSQL）
- 全表统计


通过 SHOW STATS_HEALTHY 可以查看表的统计信息健康度，并粗略估计表上统计信息的准确度。

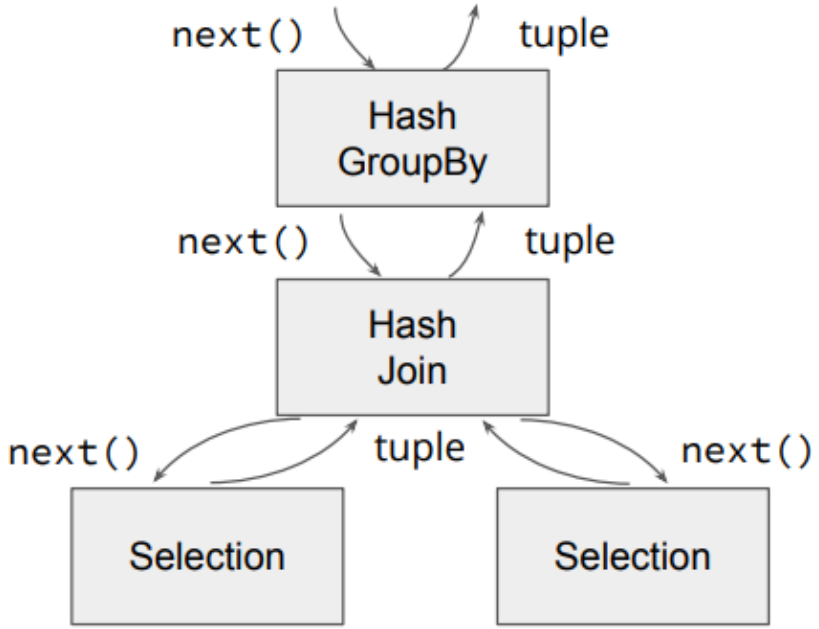
当 modify_count >= row_count 时，
健康度为 0；
当 modify_count < row_count 时，
健康度为 (1 - modify_count/row_count) * 100。

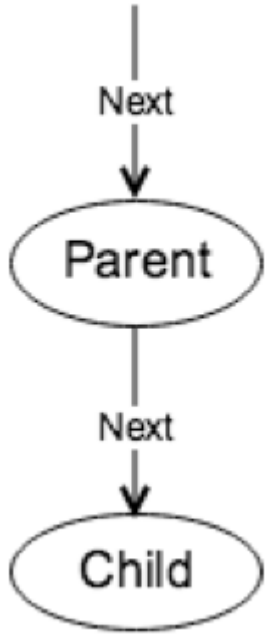
# 05 查询执行

# 05 查询执行

## 5.1 不同的执行引擎

### 1. 火山执行引擎（Volcano）

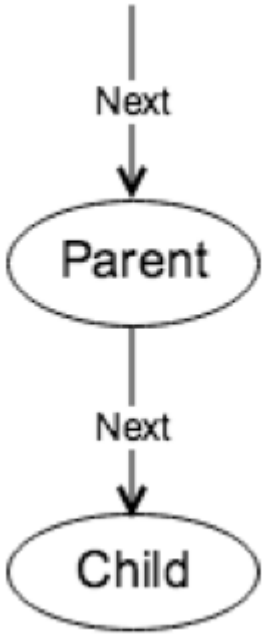


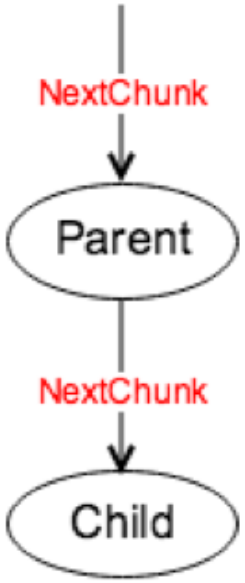



# 05 查询执行

## 5.1 不同的执行引擎

### 1. 向量化执行引擎（Vectorization）




```
189  // Executor executes a query.
190  type Executor interface {
191      Next(goctx.Context) (Row, error)
192      Close() error
193      Open(goctx.Context) error
194      Schema() *expression.Schema
195      retTypes() []*types.FieldType
196      supportChunk() bool
197      newChunk() *chunk.Chunk
198      NextChunk(goCtx goctx.Context, chk *chunk.Chunk) error
199  }
```

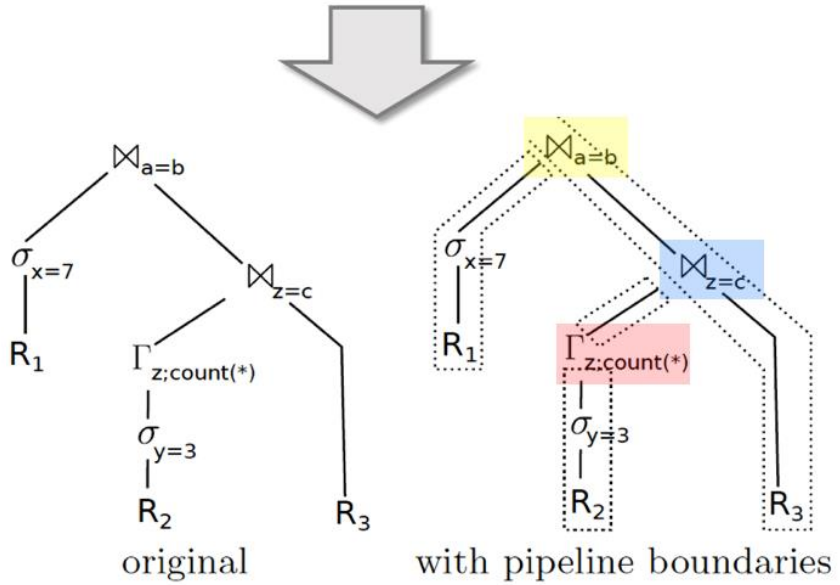
https://github.com/pingcap/tidb/blob/source-code/executor/executor.go#L198

# 05 查询执行

## 5.1 不同的执行引擎

### 1. 编译执行（Vectorization）

```
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
```




initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and $\Gamma_z$
**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 $\Gamma_z$
**for each** tuple $t$ in $\Gamma_z$
  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 $t_1 \circ t_2 \circ t_3$

# 05 查询执行

## 5.2 不同的执行引擎对比

| 执行方式 | 特征 | 优缺点 |
| --- | --- | --- |
| Volcano | Open-Next-Close<br>Pull-based<br>Tuple-at-a-time | Pipeline不友好 |
| | | 函数调用开销大 |
| | | cache命中率低 |
| Vectorization | Open-Next-Close<br>Pull-based<br>Vector-at-a-time | 一次处理一批，针对memory-bound的操作能有效形成pipeline，能够使用simd指令 |
| | | 均摊了函数调用和内存访问开销 |
| | | 代码不够紧凑，向父亲节点返回结果时需要经过内存，不适合computaiton为主的quey |
| Compilation | Push-based<br>Data-centric<br>Tuple-at-a-time | 代码紧凑，pipeline内操作一起完成，指令数少 |
| | | Cache利用率高 |
| | | Tuple-at-a-time不好形成pipeline，不适合memory-bound的query |

# 06 AIforDB

# 06 AIforDB

## 6.1 可以结合的点

构建一个高效（读写）、高可靠（崩溃次数少）、高可用（崩溃时间短）、自适应强（各种应用场景）的数据库，需要哪些技术?

# 06 AIforDB

## 6.1 可以结合的点

DB

运维调参

基数估计

计划选择

索引

物化视图

**Table 1: Machine Learning Techniques for Databases**

| | Database Problem | Method | Performance | Overhead | Training Data | Adaptivity |
|---|---|---|---|---|---|---|
| **Offline NP Problem** | knob space exploration | gradient-based [1, 18, 47] | High | High | High | – |
| | | dense network [37] | Medium | High/Medium | High | – / instance |
| | | DDPG [23, 46] | High | High | Low/Medium | query |
| | index selection | q-learning [19] | – | High | Low | – |
| | view selection | q-learning [43] | Medium | High | Low | – |
| | | DDQN [9] | High | High | Low | query |
| | partition-key selection | q-learning [11] | – | High | Low | – |
| **Online NP Problem** | join order selection | q-learning [27] | High | High | Low | – |
| | | DQN [26, 42] | High | High | Low | query |
| | | MCTS [38] | Medium | Low | Low | instance |
| | query rewrite | MCTS [21, 49] | – | Low | Low | query |
| **Regression Problem** | cost estimation | tree-LSTM [35] | High | High | High | query |
| | cardinality estimation | tree-ensemble [7] | Medium | Medium | High | query |
| | | autoregressive [41] | High | High/Medium | Low | data |
| | | dense network [16] | High | High | High | query |
| | | sum-product [12] | Medium | High | Low | data |
| | index benefit estimation | dense network [5] | – | High | High | query |
| | view benefit estimation | dense network [9] | – | High | High | query |
| | latency prediction | dense network [28] | Medium | High | High | query |
| | | graph embedding [50] | High | High | High | instance |
| | learned index | dense network [3] | – | High | High | query |
| **Prediction Problem** | trend prediction | clustering-based [24] | – | Medium | Medium | instance |
| | transaction scheduling | q-learning [44] | – | High | Low | query |

http://dbgroup.cs.tsinghua.edu.cn/ligl/papers/icde22-tutorial-paper.pdf

# 07 总结

# 07 总结

**01**. 背景

**02**. 语法分析&语义检查

**03**. 逻辑优化

**04**. 物理优化

**05**. 查询执行

**06**. AIforDB

资料推荐：

1. 优化器经典论文解析专栏：
https://www.zhihu.com/column/c_1364661018229141504
* Access Path Selection in a Relational Database Management System. SystemR的动态规划方法
* The Volcano Optimizer Generator : Extensibility and Efficient Search. Volcano 框架
* The Cascades Framework for Query Optimization. Casscase框架， Columbia optimiser

2. polarDB 和OB对他们的优化器的分享：
https://www.zhihu.com/collection/673214926?page=1

3. TiDB对他们优化器的分享：
https://docs.pingcap.com/zh/tidb/stable/sql-tuning-overview
https://cn.pingcap.com/blog/?tag=TiDB%20%E6%80%A7%E8%83%BD%E8%B0%83%E4%BC%98

4. 经典开源查询优化框架：
calcite(Volcano)，polardb-x用的这个框架 https://github.com/apache/calcite
orca（Cascades），https://zhuanlan.zhihu.com/p/365496273

5. 源码
Noisepage https://github.com/cmu-db/noisepage
Columbia https://github.com/yongwen/columbia
TiDB https://github.com/pingcap/tidb

6. 参考书籍：
数据库查询优化器的艺术 https://book.douban.com/subject/25815707/
PostgreSQL技术内幕：查询优化深度探索 https://book.douban.com/subject/30256561/

7. AIforDB 综述
http://dbgroup.cs.tsinghua.edu.cn/ligl/papers/icde22-tutorial-paper.pdf

鉴于个人涉猎有限，很多好的文章和代码没列出来，欢迎大家share~

分享结束，谢谢~