Security Level:

数据库SQL引擎

华为 李帅团 2021.04.10

www.huawei.com

HUAWEI TECHNOLOGIES CO., LTD.

HUAWEI

目录

SQL引擎架构

数据库核心主要由两大引擎组成：SQL引擎和存储引擎，关系数据库出现到现在，架构基本保持不变

SQL解析器的作用就是把一个文本的SQL语句通过词法和语法解析成一个符合内部C语言处理的结构体（一般叫解析树），典型的词法分析器采用lexer（flex），语法分析器采用yacc（bision）。

SQL逻辑优化是利用一些固定的关系变换规则，对SQL解析树进行重写，生成更加高效、利于下一步做优化的查询树。

SQL物理优化是数据库中算法要求最高的模块，优化器分两类：基于规则优化器(RBO)和基于代价优化器（CBO ），CBO是主流技术，其基础技术有两个：代价评估和计划空间搜寻算法。

执行引擎由操作算子和其相关的执行环境组成。执行环境主要由执行框架和资源管理组成。数据库包含多种操作算子，index scan、hash join、aggregate、sort、parallel等。

存储引擎主要由3大模块组成访问层、事务层和存储层：

- 访问层：提供访问存储引擎数据的相关接口，实现索引和堆访问等方式。
- 事务层：事务层是实现数据库事务的核心，主要通过事务管理、日志管理、锁管理、mvcc来实现。
- 存储层：存储层是数据库数据的物理存储，其中缓冲区管理是性能的关键。

SQL语句的处理流程

输入
- 我和小丽去学校

词法分析
- 对象：我、小丽
- 动作：移动
- 目的地：学校

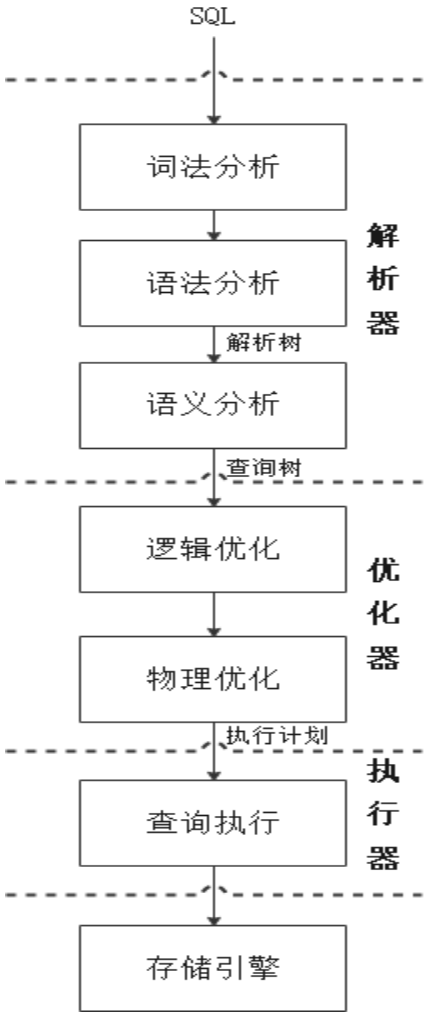
语法分析
- 我、小丽 - 去 - 学校（复合主谓宾语法）

语义分析
- 9527/9528 – goto() -- 长安区东祥路1号

逻辑优化
- 规则重写：我和小丽结对（转换规则：结对出行）

物理优化
- **RBO**：打车（优化规则：只要和小丽结伴，打车优先）
- **CBO**：
  - ~~✓ 打车(80元)~~
  - ~~✓ 步行(200元, 30km累坏了)~~
  - ✓ 公共交通：地铁六号线+旅游1号线(4元)

执行
- RBO：打车
- CBO：坐地铁 -> 导公交



目录

解析器

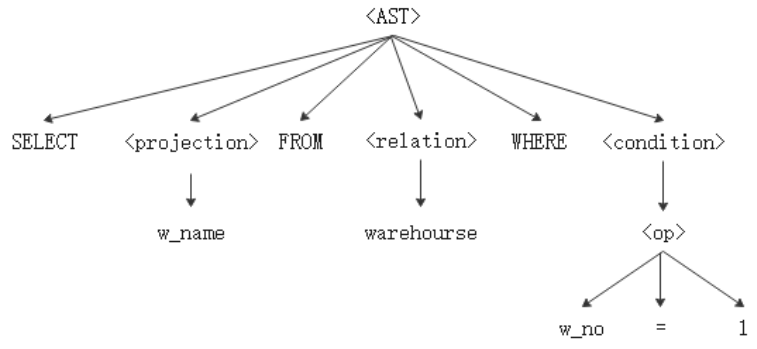
SQL解析器负责将SQL文本，编译成一个由关系算子组成的逻辑执行计划（关系代数表达式）。SQL编译过程符合编译器实现的常规过程，包含词法分析、语法分析和语义分析三个阶段。

```
SELECT w_name FROM warehouse
WHERE w_no = 1;
```

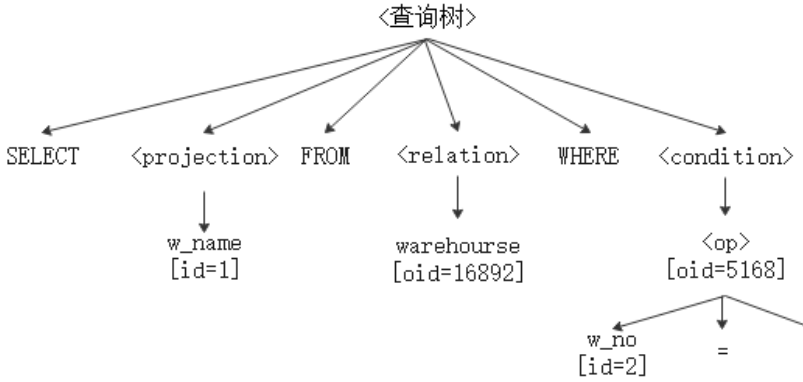
词法分析

关键字	SELECT、FROM、WHERE
标识符	w_name、warehouse、w_no
操作符	=
常量	1

语法分析



语义分析



- 词法分析：从查询语句中识别出系统支持的关键字、标识符、操作符、终结符等，每个词确定自己固有的词性。
- 语法分析：根据SQL语言的标准定义语法规则，使用词法分析中产生的词去匹配语法规则，如果一个SQL语句能够匹配一个语法规则，则生成对应的抽象语法树（Abstract Syntax Tree，AST）。
- 语义分析：对语法树（AST）进行有效性检查，检查语法树中对应的表、列、函数、表达式是否有对应的元数据，将抽象语法树转换为逻辑执行计划（查询树）。

目录

优化器-逻辑优化

逻辑优化就是规则重写，即根据规则将SQL语句重写为更高效的等价SQL，关系代数是等价变换的基础，商业数据库转换规则有成百上千条。规则重写遵循两个基本原则：

- 等价性：原语句和转换后的语句，输出结果相同。
- 高效性：转换后的语句，比原语句在执行时间和资源使用上更高效。

等价变换	内容
交换律	$A \times B == B \times A$ $A \bowtie B == B \bowtie A$ $A \bowtie_F B == B \bowtie_F A$ -- F是连接条件 $\Pi_p(\sigma_F(B)) == \sigma_F(\Pi_p(B))$ -- $F \in p$
结合律	$(A \times B) \times C == A \times (B \times C)$ $(A \bowtie B) \bowtie C == A \bowtie (B \bowtie C)$ $(A \bowtie_{F1} B) \bowtie_{F2} C == A \bowtie_{F1} (B \bowtie_{F2} C)$ -- F1和F2是连接条件
分配律	$\sigma_F(A \times B) == \sigma_F(A) \times B$ -- $F \in A$ $\sigma_F(A \times B) == \sigma_{F1}(A) \times \sigma_{F2}(B)$ -- $F = F1 \cup F2$，$F1 \in A, F2 \in B$ $\sigma_F(A \times B) == \sigma_{FX}(\sigma_{F1}(A) \times \sigma_{F2}(B))$ -- $F = F1 \cup F2 \cup FX$，$F1 \in A, F2 \in B$ $\Pi_{p,q}(A \times B) == \Pi_p(A) \times \Pi_q(B)$ -- $p \in A$，$q \in B$ $\sigma_F(A \times B) == \sigma_{F1}(A) \times \sigma_{F2}(B)$ -- $F = F1 \cup F2$，$F1 \in A, F2 \in B$ $\sigma_F(A \times B) == \sigma_{Fx}(\sigma_{F1}(A) \times \sigma_{F2}(B))$ -- 其中$F = F1 \cup F2 \cup Fx$，$F1 \in A, F2 \in B$
串接律	$\Pi_{P=p1,p2,...pn}(\Pi_{Q=q1,q2,...qn}(A)) == \Pi_{P=p1,p2,...pn}(A)$ -- $P \subseteq Q$ $\sigma_{F1}(\sigma_{F2}(A)) == \sigma_{F1 \wedge F2}(A)$

常见的重写规则-常量表达式化简

常量表达式化简：常量表达式即用户输入SQL语句中包含运算结果为常量的表达式，分为算数表达式、逻辑运算表达式、函数表达式，查询重写可以对常量表达式预先计算以提升效率。

查询重写技术	内容
常量表达式化简	**示例1**：该语句为典型的算数表达式查询重写，经过重写之后，避免了在执行时每条数据都需要进行1+1运算。 SELECT * FROM t1 WHERE c1=1+1; → SELECT * FROM t1 WHERE c1=2; **示例2**：该语句为典型的逻辑运算表达式，经过重写之后，条件永远为false，可以直接返回0行结果，避免了整个语句的实际执行。 SELECT * FROM t1 WHERE 1=0 AND a=1; →SELECT * FROM t1 WHERE false; **示例3**：该语句包含函数表达式，由于函数的入参为常量，经过重写之后，直接把函数运算结果在优化阶段计算出来，避免了在执行过程中逐条数据的函数调用开销。 SELECT * FROM t1 WHERE c1 = ADD(1,1); →SELECT * FROM t1 WHERE c1=2;

常见的重写规则–子查询优化

子查询优化是规则重写的重要内容，高效的子查询优化，对查询效率的提升非常关键。常见的子查询优化方法包括：子查询提升、子查询合并、子查询消除等。对子查询优化可避免子查询多次执行、提升为连接后可减少连接层次，为优化器提供更多表连接计划的机会。

规则重写技术	内容
子查询提升	**优化：**将子查询合并到父查询中。避免多次执行，并参与父查询连接顺序、连接算法、谓词下推优化。 **示例1：**该语句为典型的子查询提升重写，重写之后利用Hash Join可以提升查询性能。常见的IN/ANY/SOME/ALL/EXISTS都可采用该方式转换为半连接（SEMI JOIN）。 SELECT * FROM t1 WHERE t1.c1 IN (SELECT t2.c1 FROM t2); → SELECT * FROM t1 Semi Join t2 ON t1.c1 = t2.c1; **示例2：**子查询v_t2上拉合并到父查询中，降低查询层次，且t1,t2表可选择多种连接方式、连接顺序。 SELECT * FROM t1, (SELECT * FROM t2 WHERE t2.c2 > 10) v_t2 WHERE t1.c1 < 10 AND v_t2.c2 < 20; →SELECT * FROM t1, t2 WHERE t1.c1 < 10 AND t2.c2 < 20 AND t2.c2 > 10;

常见的重写规则–子查询优化

查询重写技术	内容
子查询合并	**优化：**在某些条件下（语义等价: 两个查询块产生相同结果集），多个子查询能合并为一个子查询。这样可以把多次表扫描、多次连接减少为单次表扫描和单次连接。 **示例1：**两个子查询等价合并为一个子查询。 SELECT * FROM t1 WHERE a1 < 10 AND (EXISTS (SELECT a2 FROM t2 WHERE t2.a2 < 5 AND t2.b2 = 1) OR EXISTS (SELECT a2 FROM t2 WHERE t2.a2 < 5 AND t2.b2 = 2)); → SELECT * FROM t1 WHERE a1 < 10 AND (EXISTS (SELECT a2 FROM t2 WHERE t2.a2 < 5 AND (t2.b2 = 1 OR t2.b2 = 2))); **优化：**聚集函数上推，将子查询转变为一个新的不包含聚集函数的子查询，并与父查询的部分或者全部表做连接。 **示例1：**两个子查询等价合并为一个子查询。（有没有问题这个变换） SELECT * FROM t1 WHERE t1.a1 > (SELECT avg(t2.a2) FROM t2); → SELECT t1.* FROM t1, (SELECT avg(t2.a2) FROM t2) v_t2(a2) WHERE t1.a1 > v_t2.a2; **示例2：** SELECT * FROM (SELECT max(a) FROM t) ORDER BY 1; →SELECT max(a) FROM t;
聚集子查询消除	

常见的重写规则-外连接消除

外连接和内连接的主要区别，是对于不能产生连接结果的元组需要补NULL值，如果SQL语句中有过滤条件符合空值拒绝的条件（即会将补充的NULL值再过滤掉），则可以直接消除外连接。满足“空值拒绝”时，可以将外连接转换为内连接，便于优化器灵活选择表连接顺序。

类型	PG 代码表示方式	R <op> S 的结果 = A + B + C			说明
		A	B	C	
θ- 连接	JOIN_INNER	pairs$^{\ominus}$	—	—	LHS 表示左部分的关系 R，RHS 表示右部分的关系 S
左外连接	JOIN_LEFT	pairs	unmatched LHS tuples	—	
全外连接	JOIN_FULL	pairs	unmatched LHS	unmatched RHS tuples	
右外连接	JOIN_RIGHT	pairs	—	unmatched RHS tuples	

重写规则	规则描述
外连接消除	**示例1**：外连接转成内连接之后，便于优化器应用更多的优化规则，提高执行效率。 SELECT * FROM t1 FULL JOIN t2 ON t1.c1 = t2.c1 WHERE t1.c2 > 5 AND t2.c3 < 10; → SELECT * FROM t1 INNER JOIN t2 ON t1.c1 = t2.c2 WHERE t1.c2 > 5 AND t2.c3 < 10; **示例2**：条件t2.c2可以排除掉LHS元组，满足空值拒绝条件，可转换为内连接。 SELECT * FROM t1 LEFT JOIN t2 ON (t1.c1 = t2.c1) WHERE t2.c2 > 10; → SELECT * FROM t1, t2 WHERE t1.c1 = t2.c1 AND t2.c2 > 10;

常见的重写规则-其他规则

重写规则	规则描述
视图展开	**优化**：视图从逻辑上可以简化书写SQL的难度，提高查询的易用性，而视图本身是虚拟的，因此在查询重写的过程中，需要对视图展开。 **示例1**：可以将视图查询重写成子查询的形式，然后再对子查询做简化。 CREATE VIEW v1 AS (SELECT * FROM t1,t2 WHERE t1.c1=t2.c2); SELECT * FROM v1; → SELECT * FROM (SELECT * FROM t1,t2 WHERE t1.c1=t2.c2) as v1; → SELECT * FROM t1,t2 WHERE t1.c1=t2.c2;
条件下推	**优化**：将条件下推到表扫描时，尽早过滤，减少数据传递。 **示例1**：将t1.c2下推到t1扫描时，降低JOIN计算量。 SELECT * FROM t1 JOIN t2 ON (c1) AND t1.c2 = 5; → SELECT * FROM (SELECT * FROM t1 WHERE t1.c2 = 5) JOIN t2 ON (c1);
IN谓词展开	**优化**：将IN操作符改写成等值的过滤条件，便于借助索引减少计算量。 **示例1**：谓词展开后可利用c1列索引，减少扫描量。 SELECT * FROM t1 WHERE c1 IN (10,20,30); → SELECT * FROM t1 WHERE c1=10 or c1=20 OR c1=30;
DISTINCT消除	**优化**：DISTINCT列上如果有主键约束，则此列不可能为空，且无重复值，因此不需要DISTINCT操作，减少计算量。 **示例1**：c1列上有的主键属性决定了无需做DISTINCT操作。 CREATE TABLE t1(c1 INT PRIMARY KEY, c2 INT); SELECT DISTINCT(c1) FROM t1; → SELECT c1 FROM t1;

目录

物理优化

物理优化通过枚举不同的候选执行路径，按照一定的规则，最终获得一个最优的执行路径（也称为执行计划）。主要的涉及到问题：

- 单表扫描路径选择：全表扫描 or 索引扫描
- 多表连接路径选择：NestLoop or Merge JOIN or Hash JOIN，如何从指数搜索空间，高效搜索到最优(近似最优)路径。

四表联接顺序：

- ✓abcd=[a]⋈[bcd];[b]⋈[acd];[c]⋈[abd];[d]⋈[abc];[ab]⋈[cd];[ac]⋈[bd];[ad]⋈[bc];[bcd]⋈[a];[acd]⋈[b];[abd]⋈[c];[abc]⋈[d];[cd]⋈[ab];[bd]⋈[ac];[bc]⋈[ad].
- ✓abc=[a]⋈[bc]; [b]⋈[ac]; [c]⋈[ab];[bc]⋈[a]; [ac]⋈[b]; [ab]⋈[c];
- ✓abd=[a]⋈[bd]; [b]⋈[ad]; [d]⋈[ab];[bd]⋈[a]; [ad]⋈[b]; [ab]⋈[d].
- ✓acd =[a]⋈[cd]; [c]⋈[ad]; [d]⋈[ac];[cd]⋈[a]; [ad]⋈[c]; [ac]⋈[d].
- ✓bcd =[b]⋈[cd]; [c]⋈[bd]; [d]⋈[bc];[cd]⋈[b]; [bd]⋈[c]; [bc]⋈[d].
- ✓ab= [a]⋈[b]; [b]⋈[a];
- ✓ac= [a]⋈[c]; [c]⋈[a];
- ✓ad= [a]⋈[d]; [d]⋈[a];
- ✓bc= [b]⋈[c]; [c]⋈[b];
- ✓bd= [b]⋈[d]; [d]⋈[b];
- ✓cd= [c]⋈[d]; [d]⋈[c];

如果是100表联接？

如果再加上全表扫描 or 索引扫描、NestLoop or Merge JOIN or Hash JOIN，路径搜索空间？

RBO和CBO

有两种物理优化方式：

- **基于规则的查询优化（Rule Based Optimization，RBO）**：根据预定义的启发式规则对SQL语句进行优化。
- **基于代价的查询优化（Cost Based Optimization，CBO）**：对SQL语句对应的待选执行路径进行代价估算，从待选路径中，选择代价最低的执行路径，作为最终的执行计划。

RBO Path 1: Single Row by Rowid

RBO Path 2: Single Row by Cluster Join

RBO Path 3: Single Row by Hash Cluster Key with Unique or Primary Key

RBO Path 4: Single Row by Unique or Primary Key

RBO Path 5: Clustered Join

RBO Path 7: Indexed Cluster Key

RBO Path 8: Composite Index

RBO Path 9: Single-Column Indexes

RBO Path 10: Bounded Range Search on Indexed Columns

RBO Path 11: Unbounded Range Search on Indexed Columns

RBO Path 12: Sort Merge Join

RBO Path 13: MAX or MIN of Indexed Column

RBO Path 14: ORDER BY on Indexed Column

RBO Path 15: Full Table Scan

代价模型

CBO根据代价模型估算每个算子的执行代价，以便选择代价最小的路径。执行代价主要考虑CPU代价和磁盘存取代价两个方面。磁盘代价以顺序存取一个页面代价为单位，其他代价计算相对于该元单位计算的。

算子执行代价估算模型：

Cost = P + W * T

以上公式中：

- P：表示计划执行时访问的页面数，反映了IO开销。
- T：表示计划执行时处理的元组数，反映了CPU开销。
- W: 表示IO开销和CPU开销的权重因子。

用于估算代价的参数（openGauss为例）：

- seq_page_cost: 顺序存取页面的代价，值为1.0。
- random_page_cost: 非顺序存取页面的代价，值为4.0。
- cpu_tuple_cost：典型的CPU处理一个元祖的代价，值为0.01。
- cpu_operator_cost：CPU处理一个典型的函数操作的代价，值为0.0025。

典型算子代价估算

我们称执行器的一个执行单元为算子，如扫描算子、连接算子、物化算子(Sort\Group)等。每个算子都有一套复杂的代价估算公式。

扫描方式	P	T
顺序扫描	NumPages	NumTuples
索引扫描	NumPages * F	NumTuples * F

扫描算子执行代价估算模型：

- NumPages：表的页面数。
- NumTuples：表的元组数。
- F：多个约束条件组合后的选择度。

连接方式	代价计算公式
嵌套循环连接	Couter + Nouter * Cinner
归并连接	Couter + Csortouter + Cinner + Csortinner
哈希连接	Couter + Ccreatehash + Nouter * Chash

连接算子执行代价估算模型：

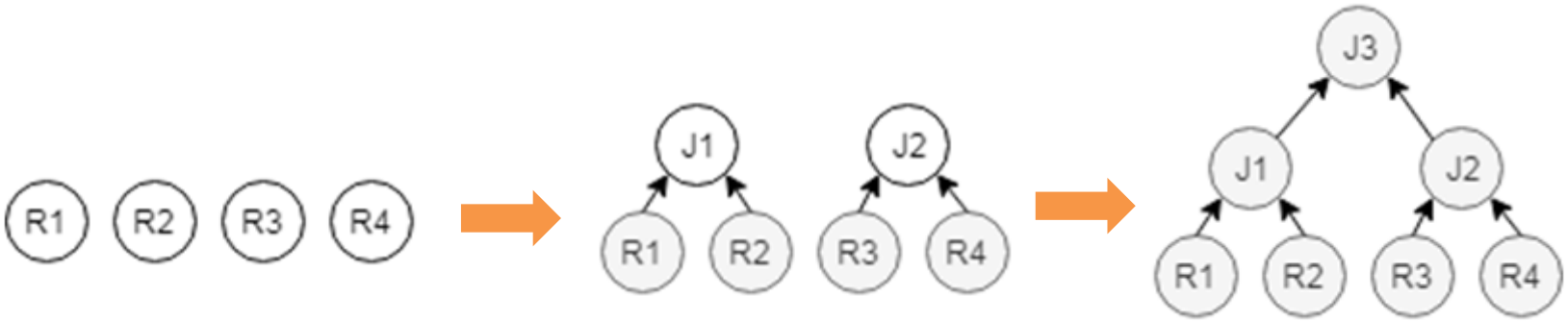
- C_{outer}：扫描外表的代价。
- C_{inner}：扫描内表的代价。
- $C_{sortouter}$：外表排序的代价。
- $C_{sortinner}$：内表排序的代价。
- $C_{createhash}$：内表创建Hash表的代价。
- C_{hash}：执行单独Hash操作的代价。
- N_{outer}：外表的大小。

可以看出，算子代价的估算依赖一些统计数据：

- 表/索引规模信息：页面数、行数
- 各列的数据统计：列的distinct值、MVC(most common value)、直方图。

路径搜索

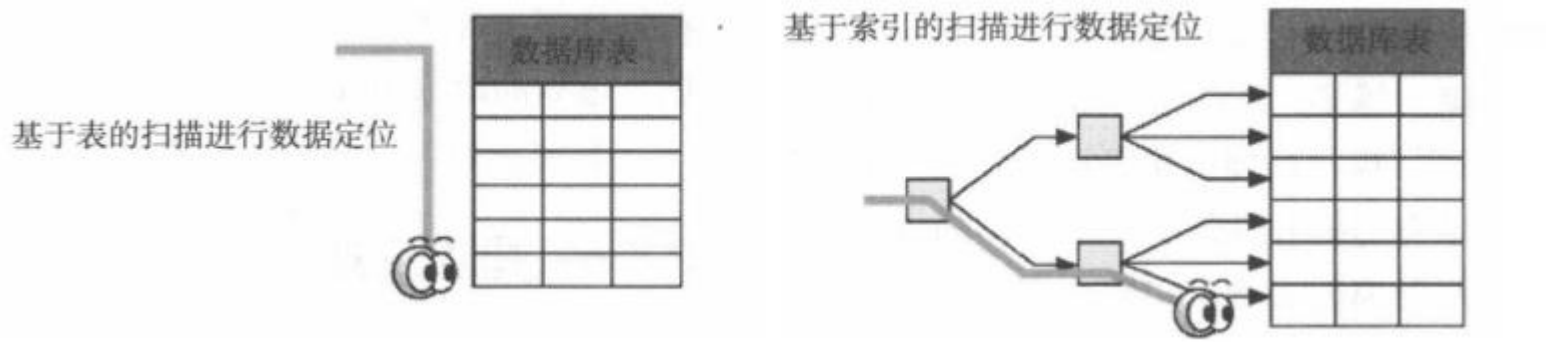
数据库实现路径搜索时，通常采用自底向上的搜索，先建立对表的扫描算子，然后由扫描算子构成连接算子，最终堆成一个物理执行计划。



在这个过程中，由于物理扫描算子和物理连接算子有多种可能，因此会生成多个物理执行路径，优化器会根据各个执行路径的估算代价，选择出代价最低的执行计划，然后转交由执行器负责执行。

单表扫描路径

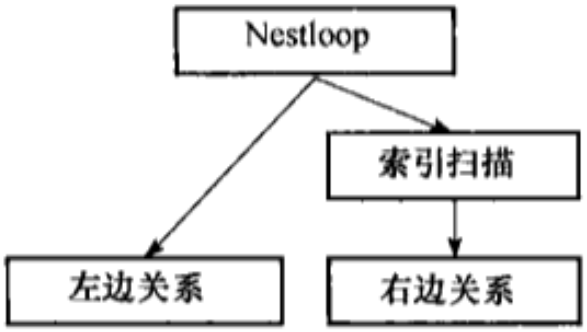
通常有2种单表扫描路径：SeqScan、IndexScan。CBO优化器选择代价最小的扫描路径。

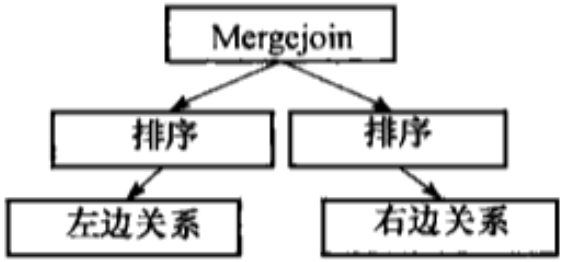


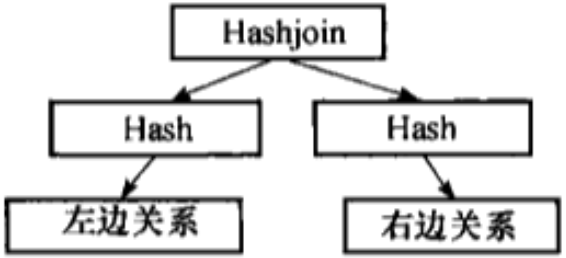
扫描算子	执行方式	限制	优势	劣势	适用场景
SeqScan	顺序扫描页面，一次匹配元组并返回	无（兜底方案）	当返回元组数较多时，所需扫描的页面较多且连续时适用（预取技术）	表页面、元组较多时，执行比较慢	当过滤度较低时（复合条件的元组较多）
IndexScan	利用索引结构快速检索符合条件的页面	过滤条件满足索引搜索条件	通过 过滤条件访问较少页面即可获得结果	当返回元组较多时，过滤效果不明显，且产生大量随机IO，不如顺序扫描页面	当过滤度较高时（复合条件的元组较少）

两表连接路径

通常有3种两表连接路径：NestLoop、MergeJoin、HashJoin，CBO优化器选择代价最小的连接路径。



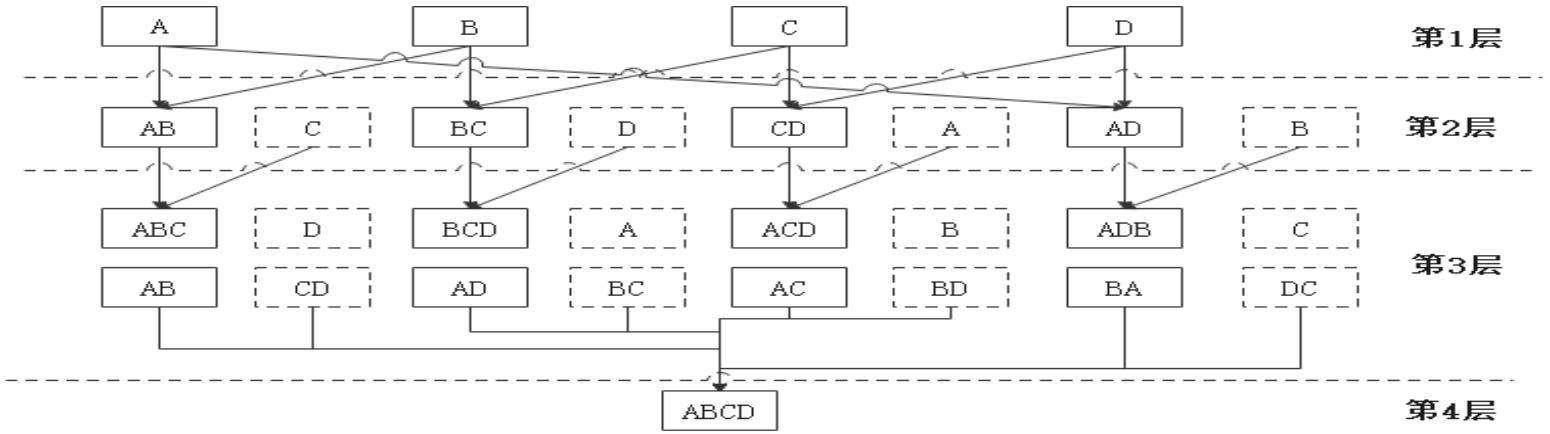




JOIN算子	执行方式	限制	优势	劣势	适用场景
NestLoop	对于外表每一行，嵌套扫描内表返回结果	无（兜底方案）	当内表使用索引时，可以快速定位连接元组（RBO规则）	每个外表元组都需要重新执行内节点操作	外表结果集小
MergeJoin	内外表均排序后，进行归并连接操作	等值连接、内外表有序（否则需要排序）	通过归并连接，一次定位连接元组	内外表需要有序，因此必须承受内外表索引扫描或排序的代价	内外表已经有序，不需要重新排序
HashJoin	内表根据JOIN列建立Hash表，外表元组进行Hash匹配。	等值连接	通过Hash散列，一次定位连接元组	内表过大导致Hash表外存、列重复值较多导致冲突链过长、数据倾斜导致分桶	内表较小可以在内存中放下，列重复值较少、数据倾斜较少

多表连接路径-动态规划

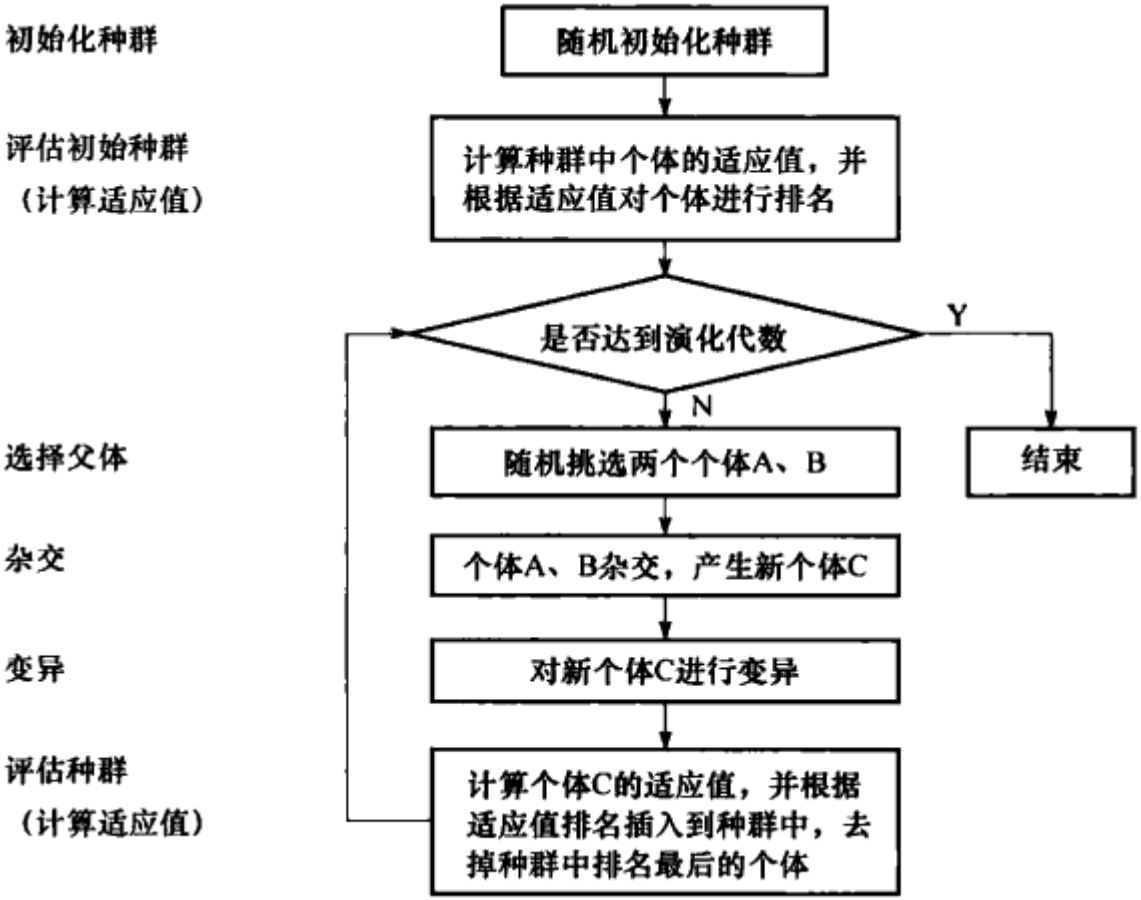
多表路径生成的难点，主要在于如何枚举所有的表连接顺序和连接算法。即如何从指数搜索空间，高效搜索到最优(近似最优)路径。常用的连接算法：动态规划、遗传算法。动态规划本质上是全路径枚举方式。



层数	说　明	可能得到的连接的中间结果
4	第四层可以通过第一、三层或第二层与第二层，进行每层间的每个表两两连接得到	{A,B,C,D}
3	第三层可以通过第一、二层两两连接得到。第三层得到的最终中间结果不多，但同一个中间结果可能由多种连接组合得到，如{A,B,C}可以由“{A,B}和{A,C}”或“{A,B}和{B,C}”连接得到，也可以由“{A,C}和{C,B}”或“{A,C}和{B,C}”等得到，方式多，但选取哪个，需要经过代价估算选取花费最小的	{A,B,C},{A,B,D}, {A,C,D},{B,C,D}
2	第二层可以通过树叶层中的表两两连接得到{A,B},{B,A}可能的连接代价是不同的，所以算作两个候选（有的书籍忽略了{A,B}和{B,A}的差别，似乎连接向来是从左至右的，所以如果树叶是“{A},{B},{C},{D}”次序，则结果只有{A,B}没有{B,A}），这对于没有特别限定表的顺序的连接是有遗漏的	{A,B},{A,C},{A,D},{B,A}, {C,A},{D,A}, {B,C},{C,B},{B,D},{D,B}, {C,D},{D,C}
1	树叶，初始层。每个单表都计算单表扫描代价	{A},{B},{C},{D}

多表连接路径-遗传算法

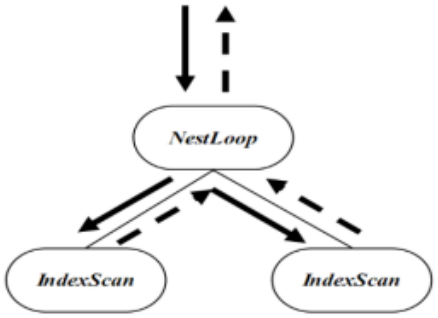
遗传算法是基于自然群体遗传一花机制的高效探索算法，该算法模拟自然生物进化过程，采用人工进化的方式对目标空间进行随机化搜索，让优化器以非穷举搜索，减少搜索空间，获取接近最优的执行计划。



目录

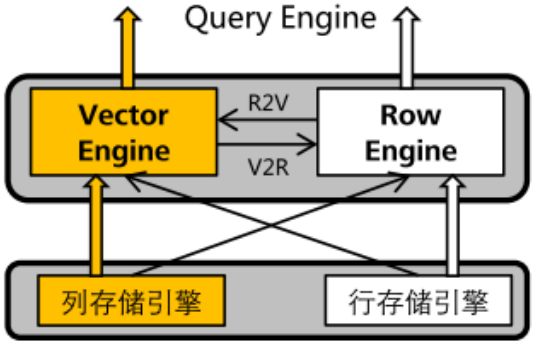
执行器

执行器以执行计划作为输入，执行的基本单位是算子，完成计划执行后，将执行结果返回给用户。



执行器从不同维度，有如下不同的执行模式：

- 数据处理流向：
  - ✓ Pipeline模式：控制流向下，数据流向上，上层驱动下层
  - ✓ Spool模式：每个算子单独执行，利用spool缓存结果
- 数据处理粒度：
  - ✓ 行执行引擎：一次处理一行数据
  - ✓ 向量化执行引擎：一次批量处理一批数据，效率较高



目录

数据表设计 - 范式规则

表的设计要尽量满足第二范式（2NF），基于提升性能的考虑可以适当增加冗余而不必满足第三范式（3NF）。

范式	范式描述	范式举例
第一范式 (1NF)	**描述：**列不可再分。 **建议：**第一范式是对关系模式的设计基本要求。	**示例1：**如下键值数据库设计违反第一范式，tel列可以再分。 { "employee":{ "id":12345, "tel":{ "mobiphone":"13612345678", “fixedphone":"029-9876543" } **示例2：**如下关系数据表满足第一范式。 CREATE TABLE employee (id INT, mobiphone TEXT, fixedphone TEXT);
第二范式 (2NF)	**描述：**主键依赖。在满足1NF的基础上，要求有主键，且非主键列必须完全依赖于主键，不能依赖部分主键(针对联合主键而言)。 **建议：**尽量满足第二范式。	**示例3：**姓名和年龄均不能唯一标识学生，违反第二范式。 CREATE TABLE student (stuname TEXT, stuage INT); **示例4：**student表课程信息、成绩信息不依赖/部分依赖stuid，违反第二范式。 CREATE TABLE student (stuid INT, name TEXT, curid INT, curname TEXT, score FLOAT, PRIMARY KEY(stuid)); **示例5：**stuname 部分依赖主键stuid，违反第二范式。 CREATE TABLE student_course(stuid INT, curid INT, score FLOAT, stuname TEXT, PRIMARY KEY(stuid,curid)); **示例6：**明确的主键依赖关系，满足第二范式。 CREATE TABLE student (stuid INT, stuname TEXT, PRIMARY KEY(stuid)); CREATE TABLE course (curid INT, curname TEXT, PRIMARY KEY(curid)); CREATE TABLE student_course(stuid INT, curid INT, score FLOAT, PRIMARY KEY(stuid,curid));

数据表设计 - 范式规则

范式	范式描述	范式举例
第三范式(3NF)	**描述：**在满足1NF和2NF的基础上，所有非主键列对任何主键列都不存在传递依赖(不依赖非主键)。 **建议：**基于提升性能的考虑可以适当增加冗余而不必满足第三范式（3NF）。	**示例7：**总额amount可以由单价和数量计算出来，依赖非主键price和ncount，违反第三范式。 CREATE TABLE product (id INT, price INT, ncount INT, amount INT, PRIMARY KEY(id));
反范式设计	**描述：**基于性能考虑，可以适当违反第二范式和第三范式：减少表的关联、更好进行索引优化。	**示例8：**示例7中，可以在冗余字段amount上添加索引，以快速检索到总额超过100的产品： SELECT * FROM product WHERE amount > 100; **示例9：**示例7中，对于“统计学生平均成绩最高的TOP 2课程名称（授课优秀/考题放水）”，违反范式的示例4要比遵循范式的示例6设计更有效： SELECT curname, avg(score) FROM student GROUP BY curid, curname ORDER BY 2 DESC LIMIT 2; SELECT course.curname, avg(student_course.score) FROM course INNER JOIN student_course ON (course.curid = student_course.curid) GROUP BY course.curid, course.curname ORDER BY 1 DESC LIMIT 2;

数据表设计 - 其他规则

在考虑三个范式的同时，还应考虑常见的设计建议。

规则	规则描述
规则1	任何表的设计都要考虑到数据的删除策略，表中的数据不能无止境的增长而不删除。
规则2	索引数据和表数据要分开存储，放在不同的表空间中。
规则3	超大表考虑使用分区表，并合理设计分区规则，以及分区索引（本地索引、全局索引）。
规则4	要区分近期记录和历史记录，不能把所有记录放都放到一个表中，要有历史表，要有定期删除历史表记录的功能。
规则5	不建议表中存储过多的null值，要考虑使用not null约束。或者考虑字符串使用NA，数值型用0作为缺省值。
规则6	给明确不存在NULL值的列加上NOT NULL约束，优化器会在特定场景下对其进行自动优化。
规则7	对于关联两个表的列，一般应该分别建立主键、外键。
规则8	通过列约束实现数据完整性校验，不允许在应用中完成对数据完整性校验。
规则9	不允许用字符类型存放时间、日期、数字类数据。
规则10	尽量避免使用大字段或者超长字段（varchar2 > 1000）。
规则11	系统中不应有过多的触发器，过多的触发器会增加维护的难度。
规则12	尽量不要使用复杂视图（即数据来自多个表，或有分组，有函数）。

索引设计

使用索引避免全表扫描，可提升数据检索效率，但在增删改时，也带来额外维护开销。在创建索引时，因遵循一定规则，创建高效的索引；在使用索引时，应避免编写无法使用索引的SQL。

规则	规则描述
规则1	应选择经常查询且过滤度高的的列创建索引。如sex字段不宜创建索引，id字段过滤度高适合创建索引。
规则2	索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引。
规则3	当需要对大数据量排序或分组时，可以通过创建索引来避免排序。
规则4	对于组合索引，要把高选择度的列放在前面；避免大范围的使用复合索引，复合字段不宜过多。
规则5	可以创建函数索引来完成特殊的优化。例如：SELECT * FROM t1 WHERE f(a) > 100; 可考虑建立函数索引：CREATE INDEX i1 ON t1 (f(a));
规则6	分区表上尽量使用本地索引，否则在分区维护的时候必须重建索引。
规则7	避免在索引字段上做计算。如SELECT * FROM DEPT WHERE SAL * 12 > 25000;
规则8	避免在索引字段上使用NOT：SELECT …FROM DEPT WHERE DEPT_CODE NOT = 0;
规则9	避免在索引列上使用IS NULL和IS NOT NULL，部分数据库实现时，索引中不包含NULL值记录。
规则10	避免改变索引列的类型。例如索引列empno为INT类型，SELECT … FROM EMP WHERE EMPNO = '123‘;
规则11	总是使用索引的第一个列。如复合索引(a,b,c)，SELECT * FROM t1 WHERE b > 10将不会使用索引。某些数据库实现支持跳跃索引。
规则12	LIKE子句尽量前端匹配。如LIKE ‘key%’可以使用索引，使用‘%key’无法使用索引。

SQL编写建议

使用索引避免全表扫描，可提升数据检索效率，但在增删改时，也带来额外维护开销。在创建索引时，因遵循一定规则，创建高效的索引；在使用索引时，应避免编写无法使用索引的SQL。

规则	规则描述
规则1	让SQL语句合理地用上索引，避免索引失效。
规则2	对于大表频繁增删操作的，建议用更新替代，并对该表定期重建索引。
规则3	考虑使用TRUNCATE替代大表DELETE。
规则4	避免频繁对表进行count操作，对大数据量表进行count操作非常耗时。
规则5	禁止多表无条件自然连接，避免产生笛卡尔积。如SELECT * FROM t1, t2;
规则6	复杂的关联查询语句中，建议增加固定的执行计划提示。
规则7	尽量不要使用负向查询，避免全表扫描。使用负向查询是指使用负向运算符，如：NOT、!=、<>、NOT EXISTS、NOT IN以及NOT LIKE等。
规则8	判断是否为“空”只能用is null或is not null，严禁使用比较运算符进行判断。
规则9	查询条件相同时，同一表的数据要一次提取完毕，不允许分多条语句提取。 SELECT c1 FROM t1 WHERE id = 1; SELECT c2 FROM t1 WHERE id = 2;
规则10	查询分区表时，在where条件中要尽可能用到“分区字段”。
规则11	尽量去掉"OR"避免全表扫描。

目录

openGauss社区：共建生态，共议发展方向

2019.9.19	2020.06.30	2020.07	2020.08	2020.09	2021~
openGauss宣布开源	openGauss源代码正式开放	社区关键组织开始运作	openGauss社区首届Meetup	HC大会伙伴案例发布	openGauss社区认证 openGauss社区用户组

openGauss社区组织架构

委员会

SIG

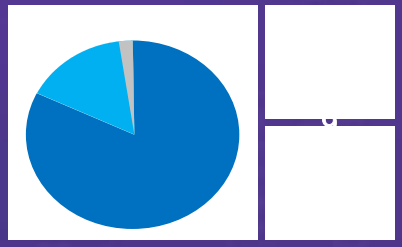
- SQLEngine
- StorageEngine
- Connectors
- Docs
- Infra
- Security

openGauss社区角色

- TC 决策社区技术发展方向
- Maintainers 看护项目代码
- Committers 看护代码质量和合入
- Contributors 需求、Bug反馈和开发实现

开源协议类型：Mulan PSL v2

openGauss社区运营进展



- 官方网站访问量：548459；官方网站访客人数：50675；官方网站安装包下载量：68274
- 微信公众号关注人数：1630；微信社群人数：718
- 社区用户D0（安装包下载IP数）：8024
- 社区参与者D1（Star、Fork、Watch）：442
- 社区贡献者D2（提交Issue和PR）：239

openGauss

https://opengauss.org

openGauss 定位

把企业级数据库能力带给用户和伙伴

价值

openGauss提供面向多核的极致性能、全链路的业务和数据安全、基于AI的调优和高效运维的能力，全面友好开放，携手伙伴共同打造全球领先的企业级开源关系型数据库；

关键特性

高性能

- **两路鲲鹏性能150万tpmC**
- ① 面向多核架构的并发控制技术；
- NUMA-Aware数据结构；
- SQL-Bypass智能选路执行技术；
- ④ 面向实时高性能场景的内存引擎；

高可用 & 高安全

- ② 业务无忧，故障切换时间RTO<10s；
- 精细安全管理：细粒度访问控制、多维度审计；
- ⑤ 全方位数据保护：存储&传输&导出加密、动态脱敏、全密态计算；

易运维

- ③ 基于AI的智能参数调优，提供AI参数自动推荐；
- 慢SQL诊断，多维性能自监控视图，实时掌控系统性能表现；
- 提供在线自学习的SQL时间预测、快速定位，急速调优；

全开放

- 采用木兰宽松许可证协议，允许对代码自由修改，使用、引用；
- 数据库内核能力完全开放；
- 开放运维监控、开发和迁移工具；
- 开放伙伴认证、培训体系及高校课程

openGauss

https://opengauss.org

Thank you

www.huawei.com