

OB SQL 引擎高级技术



目录

第一章/ OB 分布式架构高级技术

第二章 / OB 存储引擎高级技术

第三章 / OB SQL 引擎高级技术

第四章/ OB SQL调优

第五章 / OB 分布式事务高级技术

第六章/ OBProxy 路由与使用运维

第七章 / OB 备份与恢复

第八章 / OB 运维、 监控与异常处理

目录

第三章 / OB SQL引擎高级技术

3.1 SQL 请求执行流程

3.2 DML 语言处理

3.3 DDL 语言处理

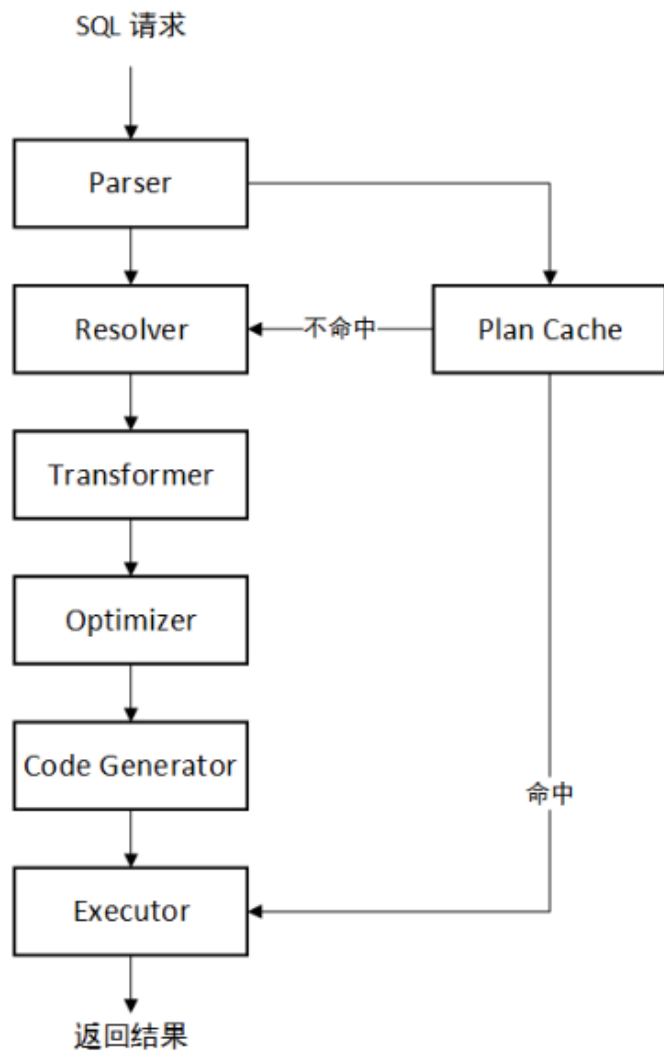
3.4 查询改写

3.5 执行计划

3.6 执行计划缓存

3.1 SQL 请求执行流程

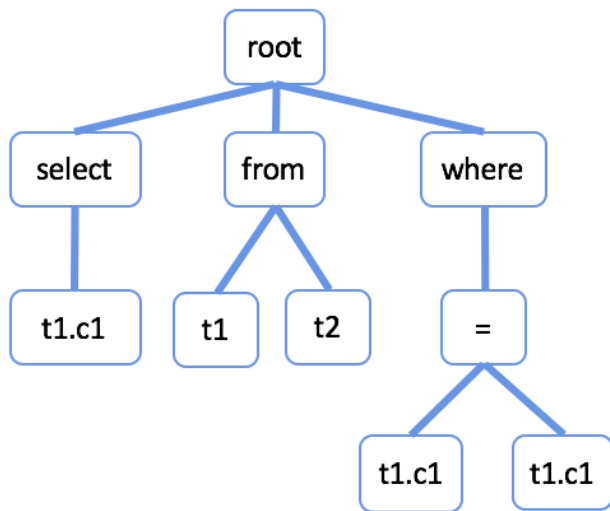
3.1 SQL请求执行流程



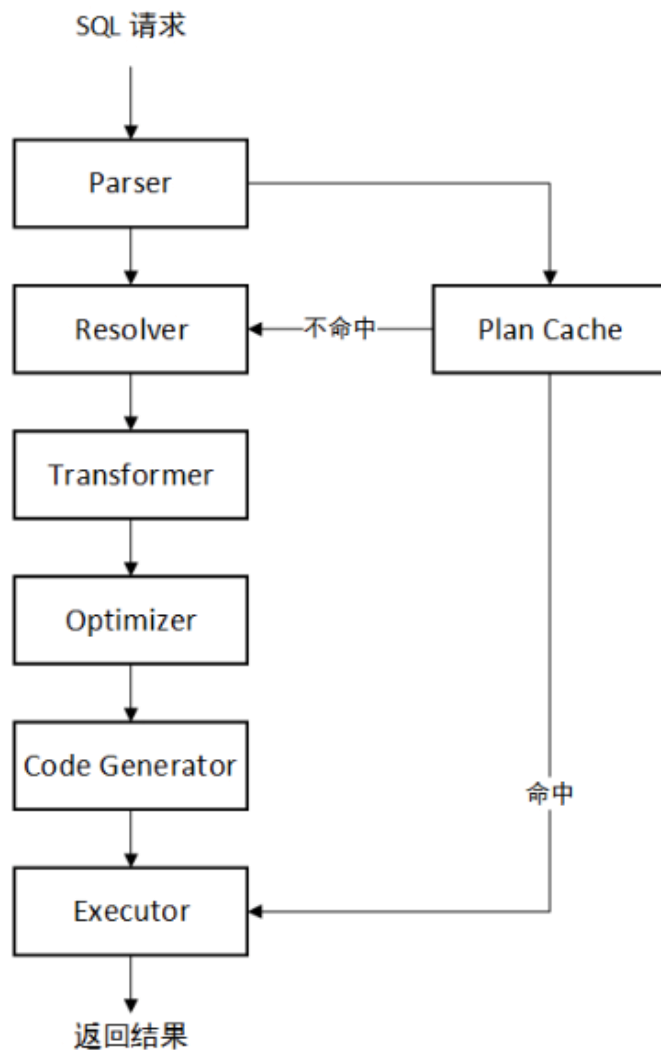
Parser (词法/语法解析模块) :

在收到用户发送的SQL请求串后，Parser会将字符串分成一个个的“单词”，并根据预先设定好的语法规则解析整个请求，将SQL请求字符串转换成带有语法结构信息的内存数据结构，我们称为“语法树” (Syntax Tree) 。

为了加速SQL请求的处理速度，OceanBase对SQL请求采用了特有的“快速参数化”，以加速查找plan cache的速度。



3.1 SQL请求执行流程

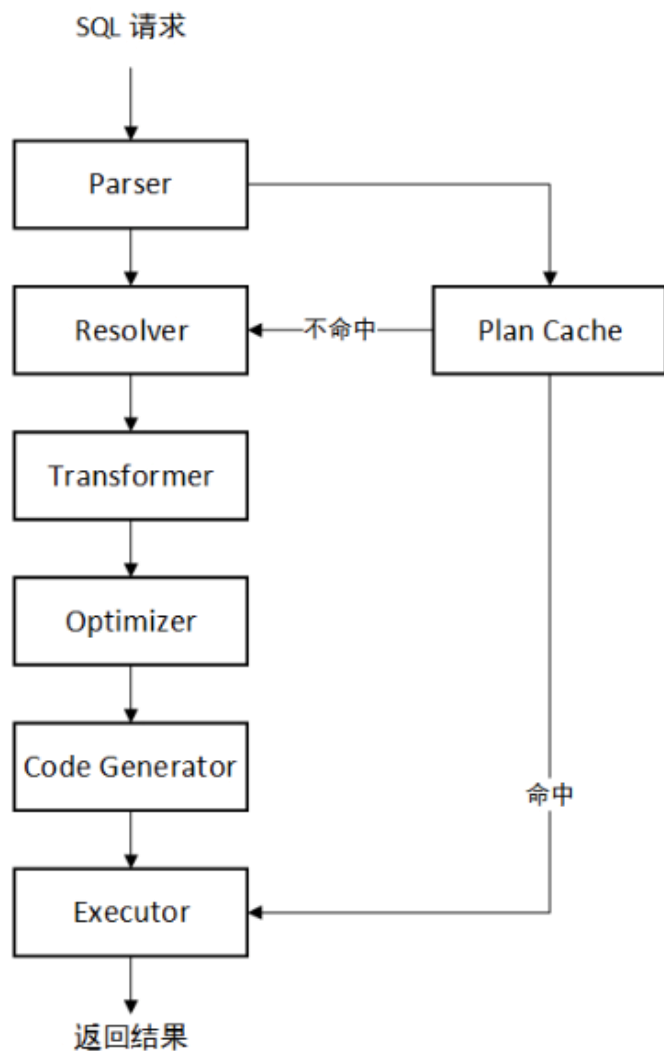


Resolver（语义解析模块）：

当SQL请求字符串经过语法、词法解析，生成“语法树”之后，resolver会进一步将该语法树转换为带有数据库语义信息的内部数据结构。

在这一过程中，resolver将根据数据库元信息将SQL请求中的token翻译成对应的对象（例如库、表、列、索引等），生成的数据结构叫做Statement Tree。

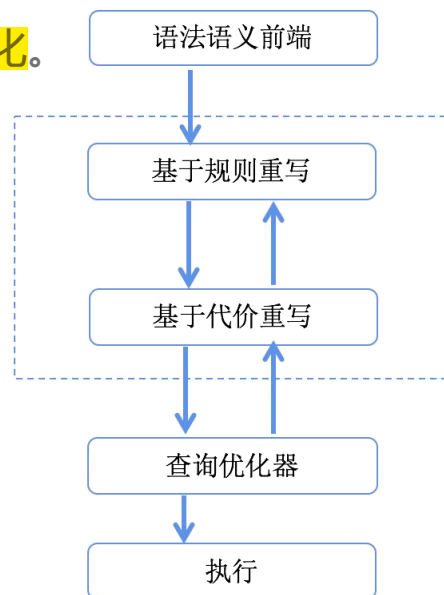
3.1 SQL请求执行流程



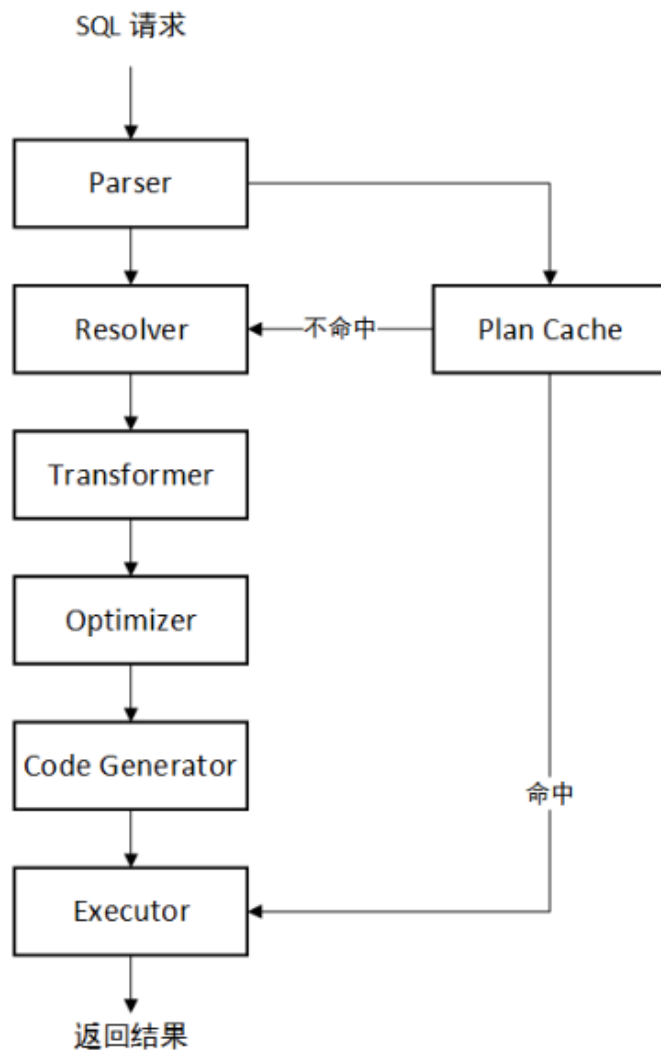
Transformer（逻辑改写模块）：

在查询优化中，经常利用等价改写的方式，将用户SQL转换为与之等价的另一条SQL，以便于优化器为之生成最佳的执行计划，我们称这一过程为“查询改写”。

Transformer在resolver之后，分析用户SQL的语义，并根据内部的规则或代价模型，将用户SQL“改写”为与之等价的其他形式，并将其提供给后续的优化器做进一步的优化。



3.1 SQL请求执行流程



Optimizer (优化器) :

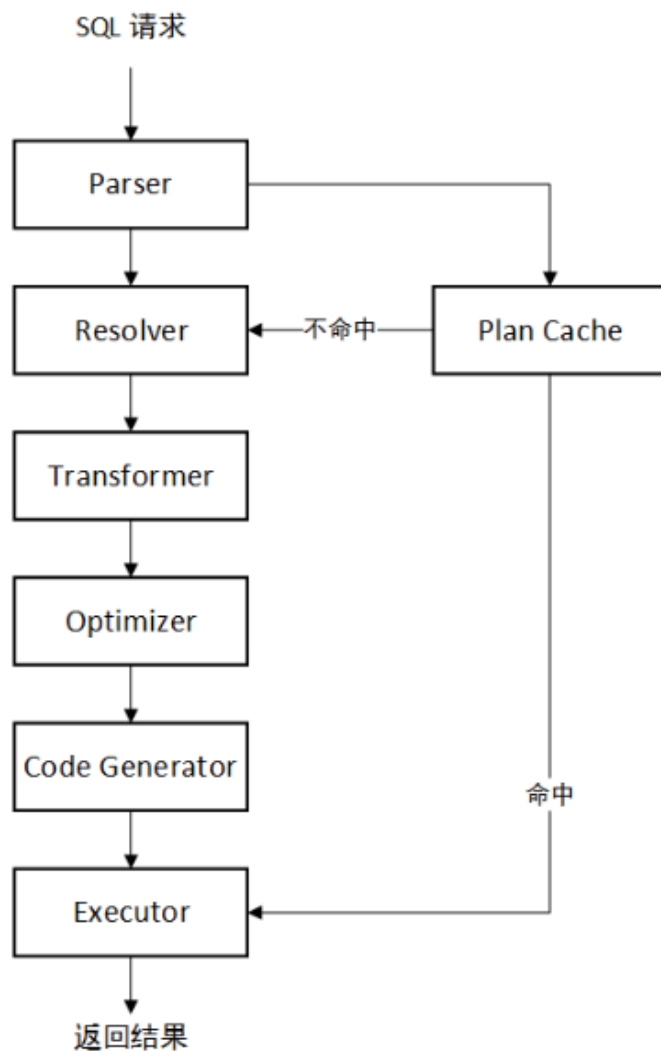
优化器是整个SQL请求优化的核心，其作用是为SQL请求生成最佳的执行计划。

在优化过程中，优化器需要综合考虑SQL请求的语义、对象数据特征、对象物理分布等多方面因素，解决访问路径选择、连接顺序选择、连接算法选择、分布式计划生成等多个核心问题，最终选择一个对应该SQL的最佳执行计划。

为了充分利用OceanBase的分布式架构和多核计算资源的优势，

OceanBase的查询优化器会对执行计划做并行优化：根据计划树上各个节点的数据分布，对串行执行计划进行自底向上的分析，把串行的逻辑执行计划改造成一个可以并行执行的逻辑计划。

3.1 SQL请求执行流程

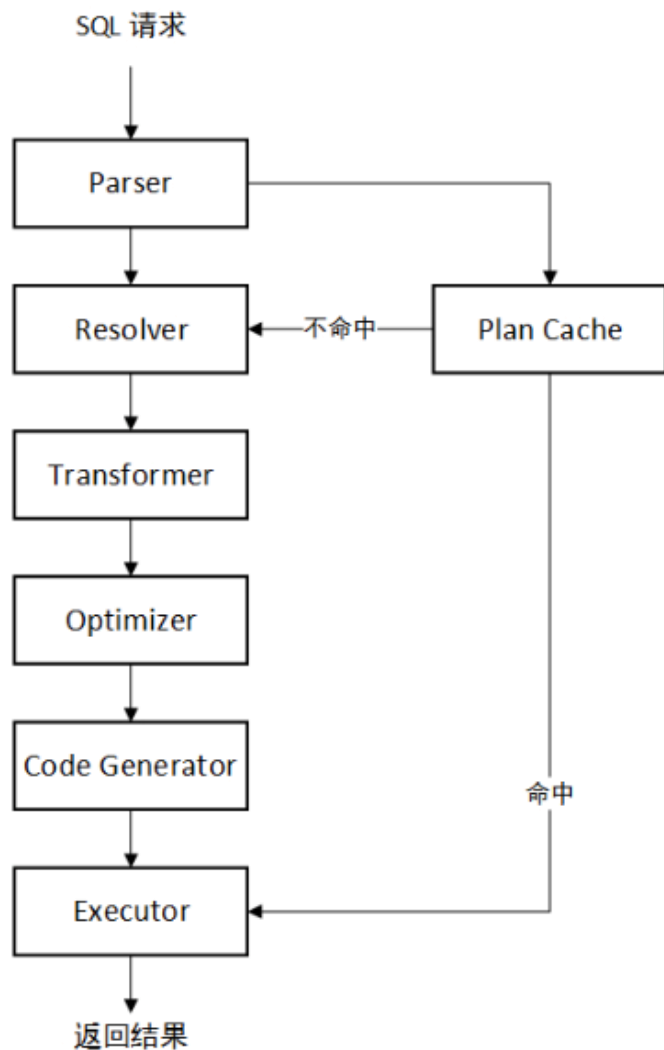


Code Generator（代码生成器）：

优化器负责生成最佳的执行计划，但其输出的结果并不能立即执行，还需要通过代码生成器将其转换为可执行的代码，这个过程由Code Generator负责。

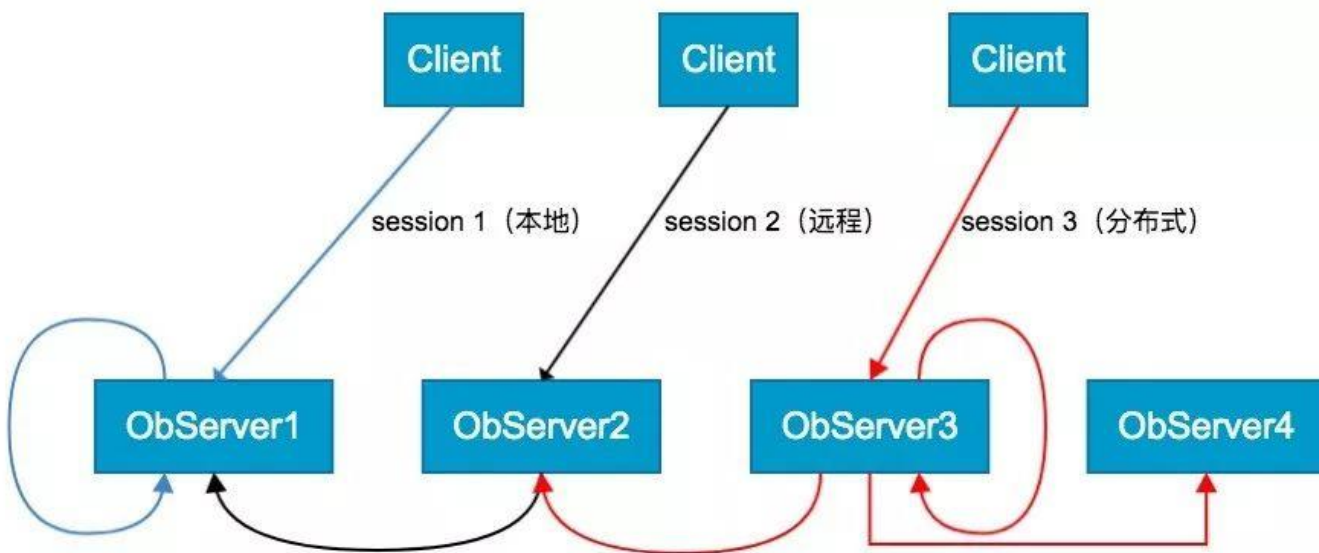
Code Generator执行的过程只是忠实地将优化器的生成结果翻译成可执行代码，并不做任何优化选择。

3.1 SQL请求执行流程

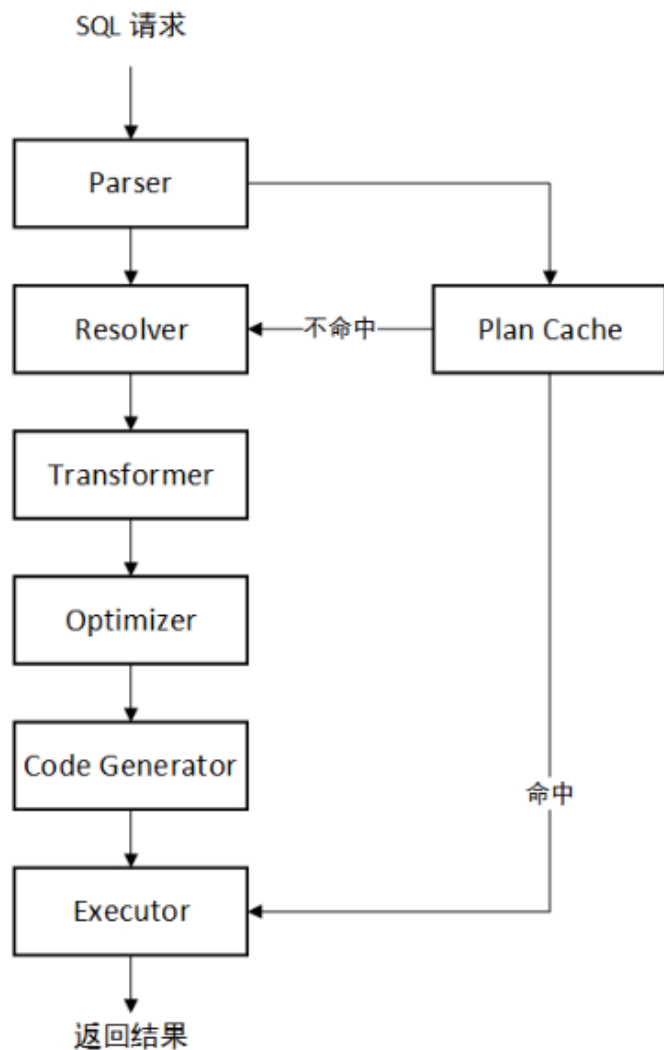


Executor (执行器) :

- 对于本地执行作业, Executor会简单的从执行计划的顶端的算子开始调用, 由算子自身的逻辑完成整个执行的过程, 并返回执行结果。
- 对于远程或分布式作业, Executor需要根据预选的划分, 将执行树分成多个可以调度的Job, 并通过RPC将其发送给相关的节点执行。



3.1 SQL请求执行流程



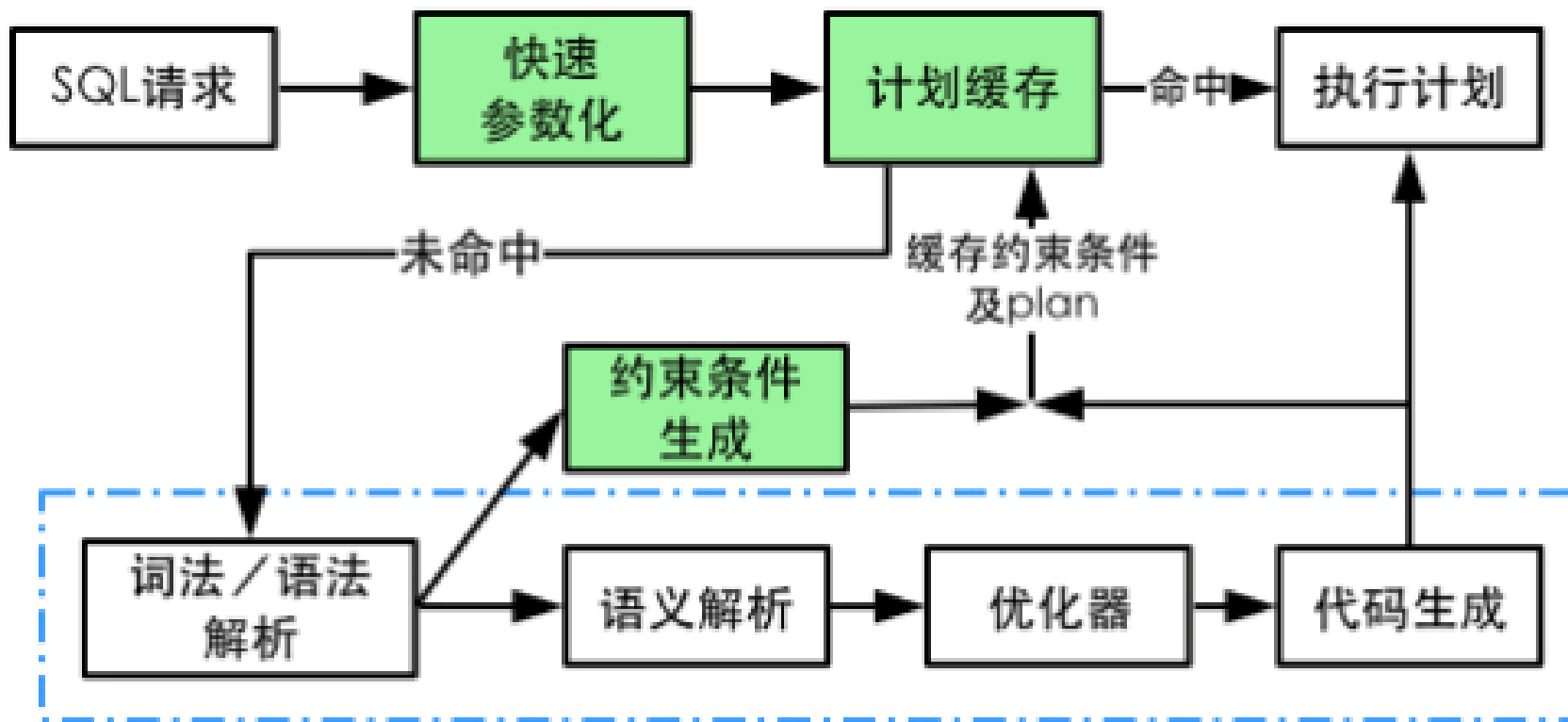
Plan Cache（执行计划缓存模块）：

执行计划的生成是一个比较复杂的过程，耗时比较长，尤其是在OLTP场景中，这个耗时往往不可忽略。

为了加速SQL请求的处理过程，SQL执行引擎会将SQL第一次生成的执行计划缓存在内存中，后续对该SQL的重复执行可以复用这个计划，避免了重复查询优化的过程。

3.1 执行计划快速参数化

- 将SQL进行参数化(即将SQL中的常量转换为参数), 然后使用参数化的SQL文本作为键值在Plan Cache中获取执行计划, 从而达到仅参数不同的SQL能够共用相同的计划目的。
- 参数化过程是指把SQL查询中的常量变成变量的过程:



3.1 执行计划快速参数化 - 例子

```
select * from t1 where c1 = 5 and c2 = "oceanbase";
```



```
select * from t1 where c1 = @1 and c2 = @2;
```

3.1 执行计划快速参数化—常量不能参数化的场景

1. 所有 ORDER BY 后常量 (例如"ORDER BY 1,2;")
2. 所有 GROUP BY 后常量 (例如"GROUP BY 1,2;")
3. LIMIT 后常量 (例如"LIMIT 5;")
4. 作为格式串的字符串常量 (例如"SELECT DATE_FORMAT('2006-06-00', '%d'); "里面的"%d")
5. 函数输入参数中, 影响函数结果并最终影响执行计划的常量 (例如"CAST(999.88 as NUMBER(2,1))"中的"NUMBER(2,1)", 或者"SUBSTR('abcd', 1, 2)"中的"1, 2")
6. 函数输入参数中, 带有隐含信息并最终影响执行计划的常量 (例如"SELECT UNIX_TIMESTAMP('2015-11-13 10:20:19.012');" 里面的"2015-11-13 10:20:19.012", 指定输入时间戳的同时, 隐含指定了函数处理的精度值为毫秒)

3.1 不能做快速参数化的常量 - 例子

- 表t1中含c1, c2列, 其中c1为主键列, 比较:

```
select c1, c2 from t1 order by 1;
```



```
OceanBase (root@oceanbase)> explain select c1, c2 from t1 order by 1;
```

```
|=====|
|ID|OPERATOR  |NAME|EST. ROWS|COST|
|-----|
|0 |TABLE SCAN|t1  |1000      |1381|
|=====|
```

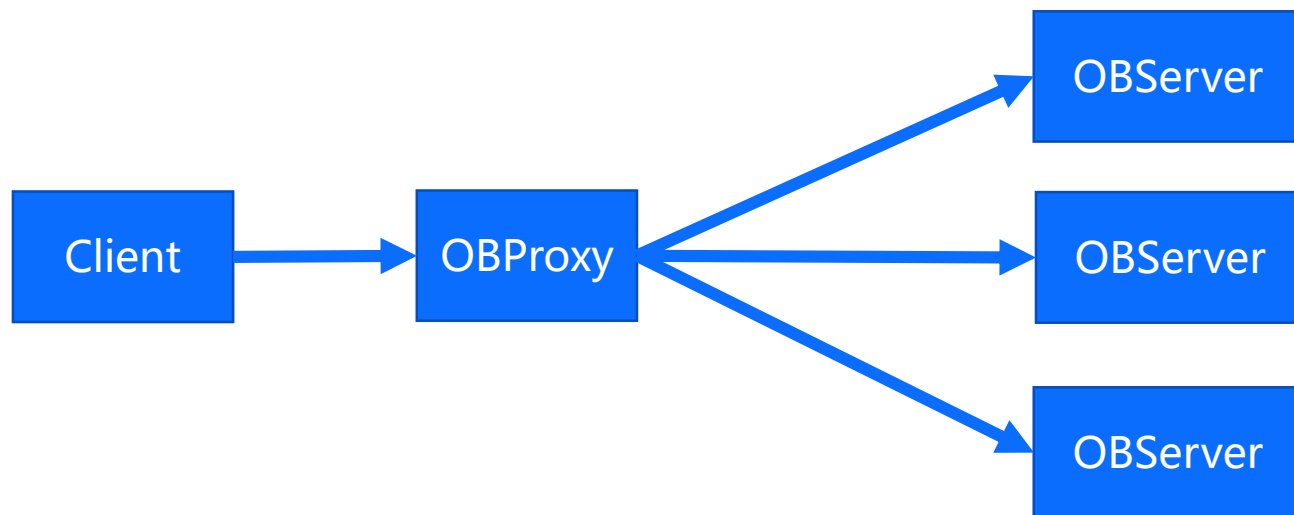
```
select c1, c2 from t1 order by 2;
```



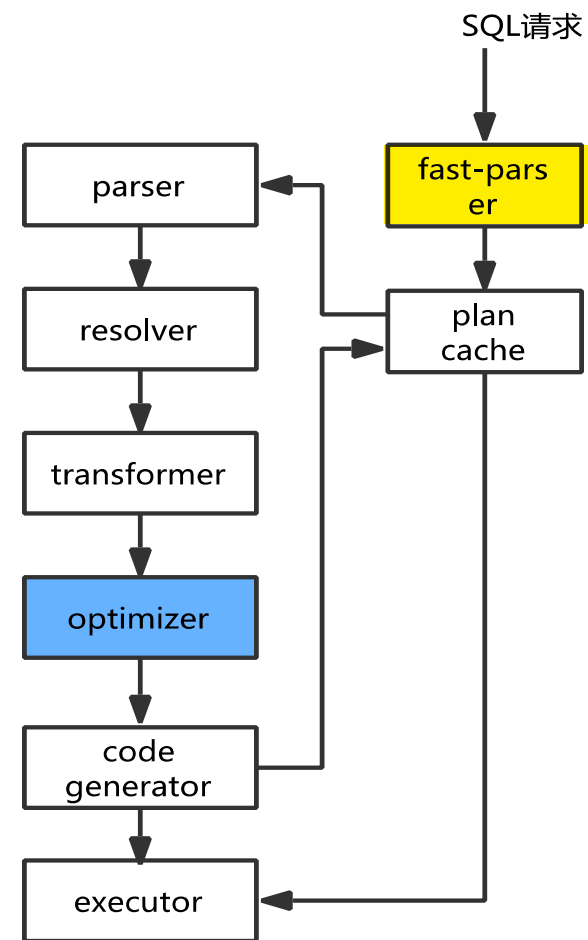
```
OceanBase (root@oceanbase)> explain select c1, c2 from t1 order by 2;
```

```
|=====|
|ID|OPERATOR  |NAME|EST. ROWS|COST|
|-----|
|0 |SORT      |    |1000      |1886|
|1 |TABLE SCAN|t1  |1000      |1381|
|=====|
```

3.1 从网络包到执行



从网络到任务队列



从解析到执行

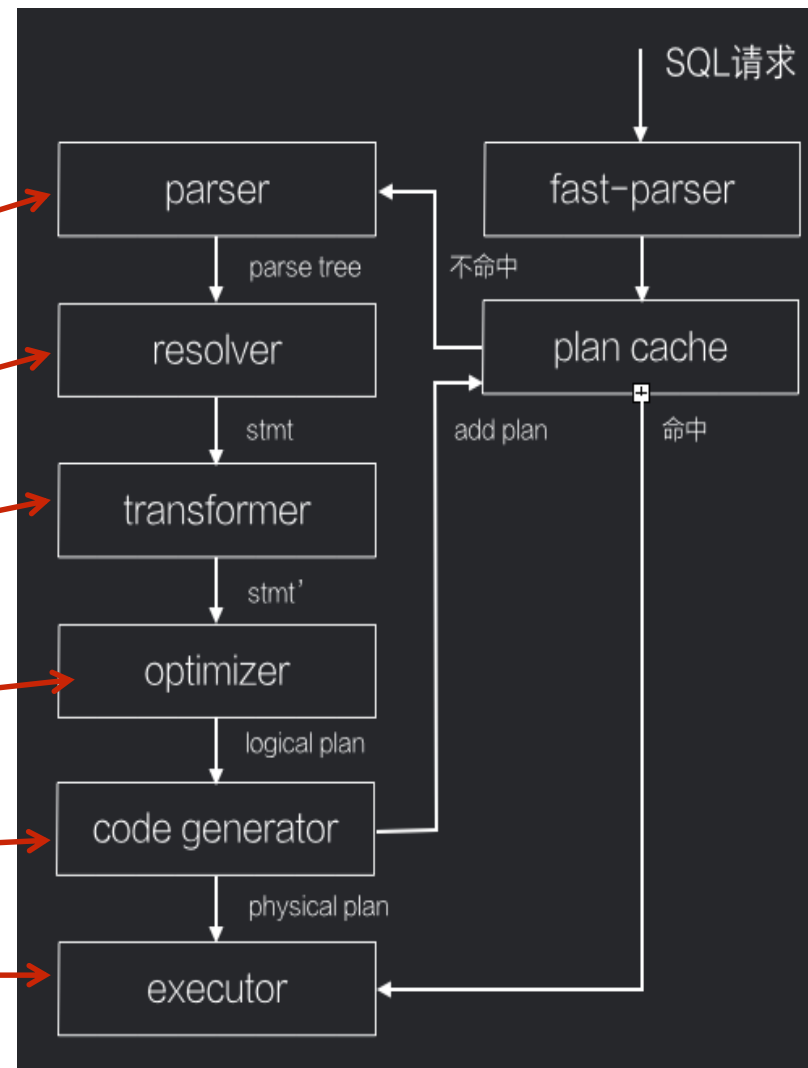
3.1 查看各阶段耗时

```
oceanbase> set ob_enable_trace_log = 1
Query OK, 0 rows affected (0.01 sec)
OceanBase (root@oceanbase)> select count(*) from __all_table;
+-----+
| count(*) |
+-----+
|      168 |
+-----+
1 row in set (0.03 sec)
```

```
OceanBase (root@oceanbase)> show trace;
```

Title	KeyValue	Time
query start	trace_id: "[Y3B6C6451982C-3E9627]";	0
parse start	stmt: "select count(*) from __all_table";	99
pc get plan start		16
pc get plan end		50
resolve start		62
resolve end		355
transform start		105
transform end		107
optimizer start		3
optimizer end		623
CG start		1
CG end		156
execution start		87
execution end		28364
query end		166

```
15 rows in set (0.01 sec)
```



3.2 DML 语言处理

3.2 DML 语句处理

- 数据操纵语言（Data Manipulation Language, DML）是SQL语言中，负责对数据库对象运行数据访问工作的指令集，以INSERT、UPDATE、DELETE三种指令为核心。
- DML的主要功能即是访问数据，因此其语法都是以读取与写入数据库为主，除了INSERT以外，其他指令都可能需搭配WHERE指令来过滤数据范围，或是不加WHERE指令来访问全部的数据。

3.2 DML 语句处理—INSERT执行计划

对于INSERT/REPLACE语句而言，由于其不用读取表中的已有数据，因此，INSERT语句的执行计划相对简单，其执行计划为简单的EXPR VALUES+INSERT OP算子构成：

```
create table t1(a int primary key, b int, index idx1(b));
explain insert into t1 values(1, 1), (2, 2);
```

```
| =====
| ID|OPERATOR      |NAME|EST. ROWS|COST|
|-----|
| 0 |INSERT          |    |0         |0    |
| 1 | EXPRESSION     |    |0         |0    |
|=====
```

Outputs & filters:

```
-----
0 - output([column_conv(INT,PS:(11,0),NOT NULL,__values.a)],
[column_conv(INT,PS:(11,0),NULL,__values.b)]), filter(nil),
   columns([t1.a], [t1.b]), partitions(p0)
1 - output([__values.a], [__values.b]), filter(nil)
   values({1, 1}, {2, 2})
|
```

3.2 DML 语句处理—UPDATE执行计划

对于UPDATE或者DELETE语句而言，优化器会通过代价模型对WHERE条件进行访问路径的选择，或者ORDER BY数据顺序的选择：

```
create table t1(a int primary key, b int, index idx1(b));
explain update t1 set b=10 where b=1;
```

```
| =====
| ID|OPERATOR      |NAME      |EST. ROWS|COST|
|-----|
| 0 |UPDATE          |          |1         |37  |
| 1 | TABLE SCAN    |t1(idx)   |1         |36  |
|=====
```

Outputs & filters:

```
-----
0 - output(nil), filter(nil), params([{t1: (t1.a, t1.b)}]), update([t1.b=?])
1 - output([t1.a], [t1.b], [?]), filter(nil),
    access([t1.b], [t1.a]), partitions(p0)
|
```

3.2 DML 语句处理—DELETE执行计划

对于UPDATE或者DELETE语句而言，优化器会通过代价模型对WHERE条件进行访问路径的选择，或者ORDER BY数据顺序的选择：

```
create table t1(a int primary key, b int, index idx1(b));
explain delete from t1 where b=1;
```

```
| =====
| ID|OPERATOR      |NAME          |EST. ROWS|COST|
|-----|
| 0 |DELETE          |              |2         |39  |
| 1 |TABLE SCAN|t1(idx1)|2         |37  |
|=====
```

Outputs & filters:

```
-----
0 - output(nil), filter(nil), params([{t1: (t1.a, t1.b)}])
```

```
1 - output([t1.a], [t1.b]), filter(nil),
      access([t1.a], [t1.b]), partitions(p0)
```

```
|
```

3.2 DML 语句处理 — 一致性校验

- DML操作的表对象每一列都有相关的约束性定义，例如列的NOT NULL约束， UNIQUE KEY约束等，写入数据前进行：
 - 约束检查
 - 类型转换
- 约束性检查失败，需要回滚该DML语句写入的脏数据。

3.2 DML 语句处理—锁管理

- 加锁时机 -通过MVCC和锁结合的机制
 - 只有行锁，没有表锁；在线DDL，不中断DML。
 - 尽量避免大量DML语句对同一行进行频繁的并发读写
 - 热点行：可以使用SELECT...FOR UPDATE先对该行加锁，然后再执行DML操作
- 加锁顺序
 - DML会级联的同步更新数据表中的数据和索引表中的数据
 - local index 和 单表 global index，绑定
 - 分区表global index，完全独立

3.3 DDL 语言处理

3.3 DDL 语句处理

1. OceanBase支持传统数据库的DDL语句，自动完成全局统一的schema变更，无需用户在多节点间做schema一致性检查。
2. DDL任务由OceanBase的RootServer统一调度执行，保证全局范围内的schema一致性。
3. DDL不会产生表锁；DML根据schema信息的变更自动记录格式，对业务零影响。

```
OceanBase(root@test)>begin;
Query OK, 0 rows affected (0.00 sec)

OceanBase(root@test)>update t1 set c2=2 where c1=1;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1 Changed: 1 Warnings: 0

OceanBase(root@test)>select * from t1;
+-----+-----+
| c1 | c2 | c3 |
+-----+-----+
| 1 | 2 | 1 |
| 2 | 2 | 2 |
+-----+-----+
2 rows in set (0.01 sec)

OceanBase(root@test)>select * from t1;
+-----+-----+
| c1 | c2 | c3 | c4 |
+-----+-----+
| 1 | 2 | 1 | NULL |
| 2 | 2 | 2 | NULL |
+-----+-----+
2 rows in set (0.00 sec)
```

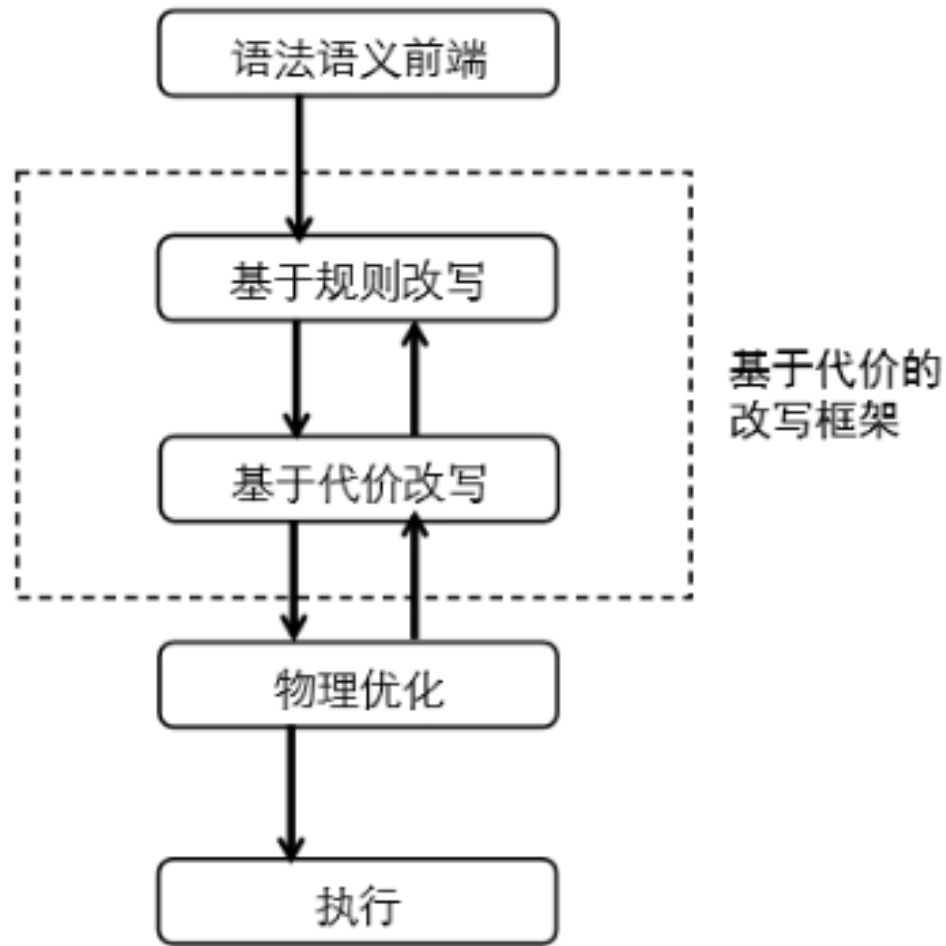
OceanBase(root@oceanbase)>use test;
Reading table information for completion of table
mes
You can turn off this feature to get a quicker sta
Database changed
OceanBase(root@test)>alter table t1 drop c4 ;
Query OK, 0 rows affected (0.07 sec)
OceanBase(root@test)>alter table t1 add c4 int;
Query OK, 0 rows affected (0.01 sec)

3.4 查询改写

3.4 查询改写概念

数据库中的查询改写(query rewrite)把一个 SQL 改写成另外一个更加容易优化的 SQL。

- 基于规则的查询改写总是会把 SQL 往“好”的方向进行改写，从而增加该SQL的优化空间。
- 基于规则的查询改写并不能总是把 SQL 往“好”的方向进行改写，所以需要代价模型来判断。
- 基于代价的改写之后可能又会重新触发基于规则的改写，所以整体上采用迭代式的方式进行改写。



1.0基于代价的逻辑优化框架

3.4 基于规则的查询改写

优化器对于子查询一般使用嵌套执行的方式，这种方式需要多次执行子查询，执行效率低。对于子查询的优化，一般会改写为连接操作，提高执行效率，主要优点如下：

- 可避免子查询多次执行；
- 优化器可根据统计信息选择更优的连接顺序和连接方法；
- 子查询的连接条件、过滤条件改写为父查询的条件后，
优化器可以进行进一步优化，比如条件下压等。

下面会列举几例介绍，更多内容请参照 OB 产品文档。

改写类型
视图合并
子查询展开
any/all 使用 MAX/MIN 改写
外连接消除
简化条件改写
等价关系推导
恒真/假消除
非 SPJ 改写
limit 下压
distinct 消除
MIN/MAX 改写

3.4 视图合并

视图合并是指将代表一个视图的子查询合并到包含该视图的查询中，视图合并后，有助于优化器增加连接顺序的选择、访问路径的选择以及进一步做其他改写操作，从而选择更优的执行计划。

```
create table t1 (c1 int, c2 int);
create table t2 (c1 int primary key, c2 int);
create table t3 (c1 int primary key, c2 int);
```

```
SQL_A: select t1.c1, v.c1
        from t1, (select t2.c1, t3.c2
                  from t2, t3
                  where t2.c1 = t3.c1) v
        where t1.c2 = v.c2;
```

<==>

```
SQL_B: select t1.c1, t2.c1
        from t1, t2, t3
        where t2.c1 = t3.c1 and t1.c2 = t3.c2;
```

SQL_A 不进行改写， 可选连接顺序有：

- t1, v(t2,t3)
- t1, v(t3,t2)
- v(t2,t3), t1
- v(t3,t2), t1

视图合并改写后， 可选连接顺序有：

- t1, t2, t3
- t1, t3, t2
- t2, t1, t3
- t2, t3, t1
- t3, t1, t2
- t3, t2, t1

3.4 子查询展开：子查询展开为semi-join/anti-join

子查询展开是指将 where 条件中子查询提升到父查询中，并作为连接条件与父查询并列进行展开。一般涉及子查询表达式有 not in、in、not exist、exist、any、all

```
create table t1 (c1 int, c2 int);
create table t2 (c1 int primary key, c2 int);

explain select * from t1 where t1.c1 in (select t2.c2 from t2)\G;
***** 1. row *****
Query Plan: =====
|ID|OPERATOR          |NAME|EST. ROWS|COST|
-----
|0 |HASH SEMI JOIN|    |495      |3931|
|1 | TABLE SCAN  |t1  |1000     |499 |
|2 | TABLE SCAN  |t2  |1000     |433 |
=====
```

Outputs & filters:

```
-----
0 - output([t1.c1], [t1.c2]), filter(nil),
   equal_conds([t1.c1 = t2.c2]), other_conds(nil)
1 - output([t1.c1], [t1.c2]), filter(nil),
   access([t1.c1], [t1.c2]), partitions(p0)
2 - output([t2.c2]), filter(nil),
   access([t2.c2]), partitions(p0)
```

改写条件

- 当生成的连接语句能返回与原始语句相同的行。
- 展开为半连接(SEMI JOIN / ANTI JOIN)

如左例所示，t2.c2 不具有唯一性，改为 semi join，该语句改写后执行计划如图所示。

3.4 子查询展开：子查询展开为内连接

子查询展开是指将 where 条件中子查询提升到父查询中，并作为连接条件与父查询并列进行展开。一般涉及子查询表达式有 not in、in、not exist、exist、any、all

上面示例的 SQL_A 中如果将 t2.c2 改为 t2.c1，由于 t2.c1 为主键，子查询输出具有唯一性，此时可以直接转换为内连接：

SQL_A: `select * from t1 where t1.c1 in (select t2.c1 from t2);`

⇕

SQL_B: `select t1.* from t1, t2 where t1.c1 = t2.c1;`

```
explain select * from t1 where t1.c1 in (select t2.c1 from t2)\G;
***** 1. row *****
```

Query Plan: =====

ID	OPERATOR	NAME	EST. ROWS	COST
0	HASH JOIN		1980	3725
1	TABLE SCAN	t2	1000	411
2	TABLE SCAN	t1	1000	499

=====

Outputs & filters:

```
0 - output([t1.c1], [t1.c2]), filter(nil),
    equal_conds([t1.c1 = t2.c1]), other_conds(nil)
1 - output([t2.c1]), filter(nil),
    access([t2.c1]), partitions(p0)
2 - output([t1.c1], [t1.c2]), filter(nil),
    access([t1.c1], [t1.c2]), partitions(p0)
```


3.4 外连接消除

外连接消除是指将外连接转换成内连接，从而可以提供更多可选择的连接路径，供优化器考虑。外连接消除需要存在“空值拒绝条件”，即 where 条件中，存在当内表生成的值为 null 时，使得输出为 false 的条件。

示例：

```
select t1.c1, t2.c2 from t1 left join t2 on t1.c2 = t2.c2
```

这是一个外连接，在其输出行中 t2.c2 可能为 null。如果加上一个条件 t2.c2 > 5，则通过该条件过滤后，t2.c2 输出不可能为 NULL，从而可以将外连接转换为内连接：

```
select t1.c1, t2.c2 from t1 left join t2 on t1.c2 = t2.c2 where t2.c2 > 5
```



```
select t1.c1, t2.c2 from t1 inner join t2 on t1.c2 = t2.c2 where t2.c2 > 5
```

3.4 基于代价的查询改写

OceanBase 目前只支持基于代价的查询改写——或展开（Or-Expansion）。

或展开（Or-Expansion）： 把一个查询改写成若干个用 union all 组成的子查询，这个改写可能会给每个子查询提供更优的优化空间，但是也会导致多个子查询的执行，所以这个改写需要基于代价去判断。

通常来说，Or-Expansion 的改写主要有如下三个作用：

1. 允许每个分支使用不同的索引来加速查询。
2. 允许每个分支使用不同的连接算法来加速查询，避免使用笛卡尔连接。
3. 允许每个分支分别消除排序，更加快速的获取top-k结果。

3.4 基于代价的查询改写

- 允许每个分支使用不同的索引来加速查询

示例：Q1 会被改写成 Q2 的形式，其中 Q2 中的谓词 `lnnvl(t1.a = 1)` 保证了这两个子查询不会生成重复的结果。

如果不进行改写，Q1 一般来说会选择主表作为访问路径，对于 Q2 来说，如果 t1 上存在索引 (a) 和索引 (b)，那么该改写可能会让 Q2 中的每一个子查询选择索引作为访问路径

```
Q1: select * from t1 where t1.a = 1 or t1.b = 1;
```

```
Q2: select * from t1 where t1.a = 1 union all select * from t1 where t1.b = 1 and lnnvl(t1.a = 1);
```

3.4 基于代价的查询改写

- 允许每个分支使用不同的索引来加速查询

--- 如果不进行or-expansion的改写，该查询只能使用主表访问路径

```
OceanBase (root@test)> explain select /*+NO_REWRITE()*/ * from t1 where
```

```
+-----+
| Query Plan
+-----+
| =====
| ID|OPERATOR  |NAME|EST. ROWS|COST|
+-----+
| 0 |TABLE SCAN|t1  |4        |649 |
+-----+
```

Outputs & filters:

```
-----
0 - output([t1.a], [t1.b], [t1.c], [t1.d], [t1.e]), filter([t1.a = 1
      access([t1.a], [t1.b], [t1.c], [t1.d], [t1.e]), partitions(p0)
```

3.4 基于代价的查询改写

- 允许每个分支使用不同的索引来加速查询

---改写之后，每个子查询能使用不同的索引访问路径

```
OceanBase (root@test)> explain select * from t1 where t1.a = 1 or t1.b
```

```
+-----+
| Query Plan
+-----+
| =====
| ID|OPERATOR  |NAME        |EST. ROWS|COST|
+-----+
| 0 |UNION ALL  |            |3         |190 |
| 1 | TABLE SCAN|t1(idx_a)|2         |94  |
| 2 | TABLE SCAN|t1(idx_b)|1         |95  |
+-----+
```

Outputs & filters:

```
-----
0 - output([UNION(t1.a, t1.a)], [UNION(t1.b, t1.b)], [UNION(t1.c, t1
1 - output([t1.a], [t1.b], [t1.c], [t1.d], [t1.e]), filter(nil),
    access([t1.a], [t1.b], [t1.c], [t1.d], [t1.e]), partitions(p0)
2 - output([t1.a], [t1.b], [t1.c], [t1.d], [t1.e]), filter([lnnvl(t1
    access([t1.a], [t1.b], [t1.c], [t1.d], [t1.e]), partitions(p02
```

3.4 基于代价的查询改写

- 允许每个分支使用不同的连接算法来加速查询，避免使用笛卡尔连接

示例：Q1 会被改写成 Q2 的形式。

对于 Q1 来说，它的连接方式只能是 nested loop join (笛卡尔乘积)，但是被改写之后，每个子查询都可以选择 nested loop join, hash join 或者 merge join，这样会有更多的优化空间。

```
Q1: select * from t1, t2 where t1.a = t2.a or t1.b = t2.b;  
Q2: select * from t1, t2 where t1.a = t2.a  
    union all  
    select * from t1, t2 where t1.b = t2.b and lnnvl(t1.a = t2.a)
```

3.4 基于代价的查询改写

- 允许每个分支使用不同的连接算法来加速查询，避免使用笛卡尔连接

---如果不进行改写，只能使用nested loop join

```
OceanBase (root@test)> explain select /*+NO_REWRITE()*/ * from t1, t2
```

=====				
ID	OPERATOR	NAME	EST. ROWS	COST

0	NESTED-LOOP JOIN		3957	585457
1	TABLE SCAN	t1	1000	499
2	TABLE SCAN	t2	4	583
=====				

Outputs & filters:

```
-----  
0 - output([t1.a], [t1.b], [t2.a], [t2.b]), filter(nil),  
    conds(nil), nl_params_([t1.a], [t1.b])  
1 - output([t1.a], [t1.b]), filter(nil),  
    access([t1.a], [t1.b]), partitions(p0)  
2 - output([t2.a], [t2.b]), filter([? = t2.a OR ? = t2.b]),  
    access([t2.a], [t2.b]), partitions(p0)
```

3.4 基于代价的查询改写

- 允许每个分支使用不同的连接算法来加速查询，避免使用笛卡尔连接
---被改写之后，每个子查询都使用了hash join

```
OceanBase (root@test)> explain select * from t1, t2 where t1.a = t2.a
```

ID	OPERATOR	NAME	EST. ROWS	COST
0	UNION ALL		2970	9105
1	HASH JOIN		1980	3997
2	TABLE SCAN	t1	1000	499
3	TABLE SCAN	t2	1000	499
4	HASH JOIN		990	3659
5	TABLE SCAN	t1	1000	499
6	TABLE SCAN	t2	1000	499

Outputs & filters:

```
0 - output([UNION(t1.a, t1.a)], [UNION(t1.b, t1.b)], [UNION(t2.a, t2
1 - output([t1.a], [t1.b], [t2.a], [t2.b]), filter(nil),
   equal_conds([t1.a = t2.a]), other_conds(nil)
2 - output([t1.a], [t1.b]), filter(nil),
   access([t1.a], [t1.b]), partitions(p0)
3 - output([t2.a], [t2.b]), filter(nil),
   access([t2.a], [t2.b]), partitions(p0)
4 - output([t1.a], [t1.b], [t2.a], [t2.b]), filter(nil),
   equal_conds([t1.b = t2.b]), other_conds([lnnvl(t1.a = t2.a)])
5 - output([t1.a], [t1.b]), filter(nil),
   access([t1.a], [t1.b]), partitions(p0)
6 - output([t2.a], [t2.b]), filter(nil),
   access([t2.a], [t2.b]), partitions(p0)
```


3.4 基于代价的查询改写

- 允许每个分支分别消除排序，更加快速的获取top-k结果

示例：Q1 会被改写成 Q2。

对于 Q1 来说，执行方式是只能把满足条件的行数找出来，然后进行排序，最终取top-10 结果。对于 Q2 来说，如果存在索引(a,b), 那么 Q2 中的两个子查询都可以使用索引把排序消除，每个子查询取 top-10 结果，然后最终对这20行数据排序一下取出最终的 top-10 行。

```
Q1: select * from t1 where t1.a = 1 or t1.a = 2 order by b limit 10;
```

```
Q2: select * from
      (select * from t1 where t1.a = 1 order by b limit 10
       union all
       select * from t1 where t1.a = 2 order by b limit 10)
as temp
order by temp.b limit 10;
```

3.4 基于代价的查询改写

- 允许每个分支分别消除排序，更加快速的获取top-k结果

---不改写的话，需要排序最终获取top-k结果

```
OceanBase (root@test)> explain select /*+NO_REWRITE()*/ * from t1 where
| =====
| ID | OPERATOR          | NAME          | EST. ROWS | COST |
|-----|-----|-----|-----|-----|
| 0  | LIMIT              |               | 4          | 77   |
| 1  | TOP-N SORT         |               | 4          | 76   |
| 2  | TABLE SCAN|t1(idx_a)| 4          | 73   |
|=====|
```

Outputs & filters:

```
-----
0 - output([t1.a], [t1.b]), filter(nil), limit(10), offset(nil)
1 - output([t1.a], [t1.b]), filter(nil), sort_keys([t1.b, ASC]), top
2 - output([t1.a], [t1.b]), filter(nil),
    access([t1.a], [t1.b]), partitions(p0)
```

3.4 基于代价的查询改写

- 允许每个分支分别消除排序，更加快速的获取top-k结果
---进行改写的话，排序算子可以被消除，最终获取top-k结果

```
OceanBase (root@test)> explain select * from t1 where t1.a = 1 or t1.a
```

```
| =====  
| ID | OPERATOR          | NAME          | EST. ROWS | COST |  
|----|-----|-----|-----|  
| 0 | LIMIT              |               | 3         | 76   |  
| 1 | TOP-N SORT         |               | 3         | 76   |  
| 2 | UNION ALL          |               | 3         | 74   |  
| 3 | TABLE SCAN        | t1(idx_a)     | 2         | 37   |  
| 4 | TABLE SCAN        | t1(idx_a)     | 1         | 37   |  
|=====
```

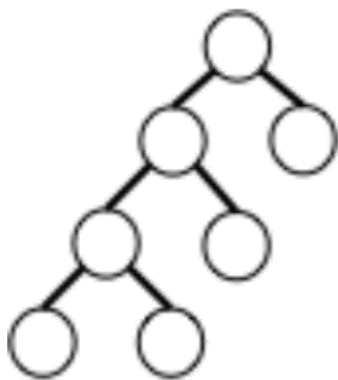
```
Outputs & filters:
```

```
-----  
0 - output([UNION(t1.a, t1.a)], [UNION(t1.b, t1.b)]), filter(nil), 1  
1 - output([UNION(t1.a, t1.a)], [UNION(t1.b, t1.b)]), filter(nil), s  
2 - output([UNION(t1.a, t1.a)], [UNION(t1.b, t1.b)]), filter(nil)  
3 - output([t1.a], [t1.b]), filter(nil),  
    access([t1.a], [t1.b]), partitions(p0),  
    limit(10), offset(nil)  
4 - output([t1.a], [t1.b]), filter([lnnvl(t1.a = 1)]),  
    access([t1.a], [t1.b]), partitions(p0),  
    limit(10), offset(nil)
```

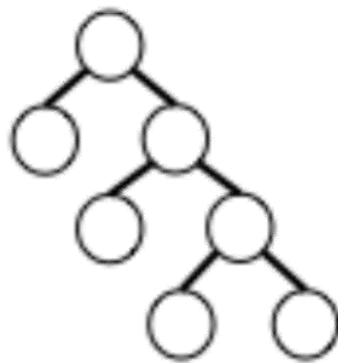
3.5 执行计划

3.5 执行计划

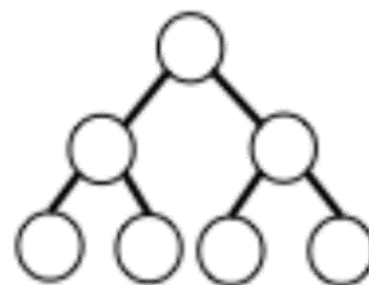
- SQL 是一种“描述型”语言。与“过程型”语言不同，用户在使用 SQL 时，只描述了“要做什么”，而不是“怎么做”。
- 数据库在接收到 SQL 查询时，必须为其生成一个“执行计划”。OceanBase 的执行计划与本质上是由物理操作符构成的一棵执行树。
- 执行树从形状上可以分为“左深树”、“右深树”和“多枝树”三种（参见下图）。OceanBase 的优化器在生成连接顺序时主要考虑左深树的连接形式。



左深树



右深树



多枝树

3.5 执行计划展示 (EXPLAIN)

- 通过Explain命令查看优化器针对给定SQL生成的逻辑执行计划。
- Explain不会真正执行给定的SQL，可以放心使用该功能而不用担心在性能调试中可能给系统性能带来影响。
- Explain 命令格式如下例所示，展示的格式包括 BASIC、EXTENDED、PARTITIONS 等等，内容的详细程度有所区别：

```
EXPLAIN [BASIC | EXTENDED | PARTITIONS | FORMAT = format_name] explainable_stmt
```

```
format_name: { TRADITIONAL | JSON }
```

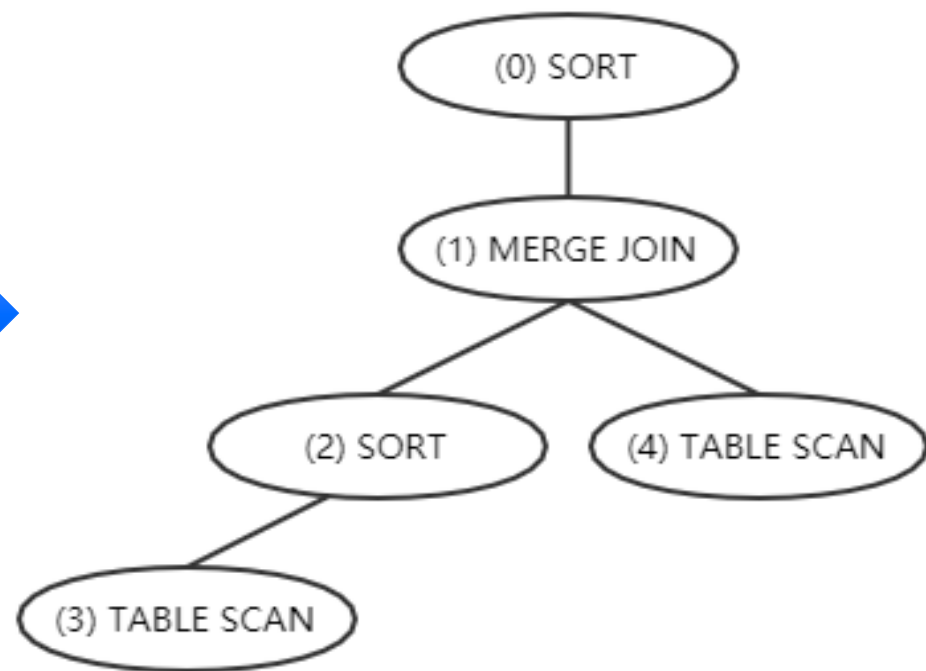
```
explainable_stmt: { SELECT statement  
                  | DELETE statement  
                  | INSERT statement  
                  | REPLACE statement  
                  | UPDATE statement }
```

3.5 执行计划展示 (EXPLAIN)—计划形状与算子信息

- Explain输出的第一部分是执行计划的树形结构展示。其中每一个操作在树中的层次通过其在OPERATOR中的缩进予以展示：

=====				
ID	OPERATOR	NAME	EST. ROWS	COST

0	SORT		1	2763
1	MERGE INNER JOIN		1	2735
2	SORT		1000	1686
3	TABLE SCAN	t2	1000	1180
4	TABLE SCAN	t1	1	815
=====				



3.5 执行计划展示 (EXPLAIN)—计划形状与算子信息

各列的含义

列名	含义
ID	执行树按照前序遍历的方式得到的编号 (从0开始)
<u>OPERATOR</u>	<u>操作算子的名称</u>
NAME	对应表操作的表名 (索引名)
<u>EST. ROWS</u>	<u>估算的该操作算子的输出行数</u>
<u>COST</u>	<u>该操作算子的执行代价 (微秒)</u>

常见的算子

类型	算子
表访问	table scan, table get
连接	NESTED-LOOP, BLK-NESTED-LOOP, Merge、hash
排序	sort, top-n sort
聚合	merge group-by, hash group-by, window function
分布式	exchange in/out remote/distribute
集合	union, except, intersect, minus
其他	limit, material, subplan, expression, count

3.5 执行计划展示 (EXPLAIN)—操作算子详细输出

- Explain输出的第二部分是各操作算子的详细信息，包括输出表达式、过滤条件、分区信息以及各算子的独有信息，包括排序键、连接键、下压条件等等：

Outputs & filters:

```
0 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil), sort_keys([t1.c1, ASC],
[t1.c2, ASC]), prefix_pos(1)
1 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil),
   equal_conds([t1.c1 = t2.c2]), other_conds(nil)
2 - output([t2.c1], [t2.c2]), filter(nil), sort_keys([t2.c2, ASC])
3 - output([t2.c2], [t2.c1]), filter(nil),
   access([t2.c2], [t2.c1]), partitions(p0)
4 - output([t1.c1], [t1.c2]), filter(nil),
   access([t1.c1], [t1.c2]), partitions(p0)
```

3.5 执行计划展示 (EXPLAIN)—举例

```
OceanBase (root@oceanbase)> explain extended
select /*+ index(t1 t1_c2) */ from t1 where c3 = 5 and c1 = 6 order by c2, c3;
=====
|ID|OPERATOR  |NAME          |EST. ROWS|COST|
-----
|0 |TABLE SCAN|t1 (t1_c2)    |1         |1255|
=====
Outputs & filters:
-----
 0 - output([t1.c1(0x7f1d520a0a98)], [t1.c2(0x7f1d520a0d98)], [t1.c3(0x7f1d5209fbe0)]),
filter([t1.c3(0x7f1d5209fbe0) = 5(0x7f1d5209f5d8)], [t1.c1(0x7f1d520a0a98) = 6(0x7f1d520a0490)]),
      access([t1.c3(0x7f1d5209fbe0)], [t1.c1(0x7f1d520a0a98)], [t1.c2(0x7f1d520a0d98)]),
partitions(p0),
      is_index_back=true, filter_before_indexback[false,true],
      range_key([t1.c2(0x7f1d520a0d98)], [t1.c1(0x7f1d520a0a98)]), range(MIN,MIN ; MAX,MAX) always
true
```

可以看出要访问的表为t1_c2这张索引表，表的主键为(c2, c1)，扫描的范围是全表扫描。

3.5 实时执行计划展示

Step1: 首先通过(g)v\$plan_cache_plan_stat虚拟表查询到plan_id。

Step2: 通过查询(g)v\$plan_cache_plan_explain这张虚拟表来展示某条SQL在计划缓存中的执行计划。

```
OceanBase(root@oceanbase)>select *
                                from v$plan_cache_plan_stat
                                where tenant_id= 1001
                                   and statement like 'insert into t1 values%'\G
*****1. row *****
      tenant_id: 1001
        svr_ip:100.81.152.44
        svr_port:15212
         plan_id: 7
         sql_id:0
           type: 1
    statement: insert into t1 values(1)
      plan_hash:1
last_active_time:2016-05-28 19:08:57.416670
      avg_exe_usec:0
slowest_exe_time:1970-01-01 08:00:00.000000
slowest_exe_usec:0
       slow_count:0
        hit_count:0
         mem_used:8192
1 row in set (0.01 sec)
```

3.5 实时执行计划展示

- `v$plan_cache_plan_explain`
 - 查询时必须指定以下值:
 - `tenant_id`和`plan_id`
- `gv$plan_cache_plan_explain`
 - 查询时必须指定以下这四列的值:
 - `ip`、`port`、`tenant_id`、`plan_id`

字段名称	类型	描述
TENANT_ID	bigint(20)	租户id
IP	varchar(32)	ip地址
PORT	bigint(20)	端口号
PLAN_ID	bigint(20)	plan的id
OPERATOR	varchar(128)	operator的名称
NAME	varchar(128)	表的名称
ROWS	bigint(20)	预估的结果行数
COST	bigint(20)	预估的代价
PROPERTY	varchar(256)	对应operator的信息

3.5 实时执行计划展示

- EXPLAIN命令的输出:
- v\$plan_cache_plan_explain中的数据:

ID	OPERATOR	NAME	EST. ROWS	COST
0	SCALAR GROUP BY		1	353261534
1	PX COORDINATOR		1	298249962
2	EXCHANGE OUT DISTR	:EX10000	1	298249962
3	MERGE GROUP BY		1	298249962
4	PX PARTITION ITERATOR		288000000	243238389
5	HASH JOIN		288000000	243238389
6	TABLE SCAN	t1	24000	27265
7	TABLE SCAN	t2	24000	27265

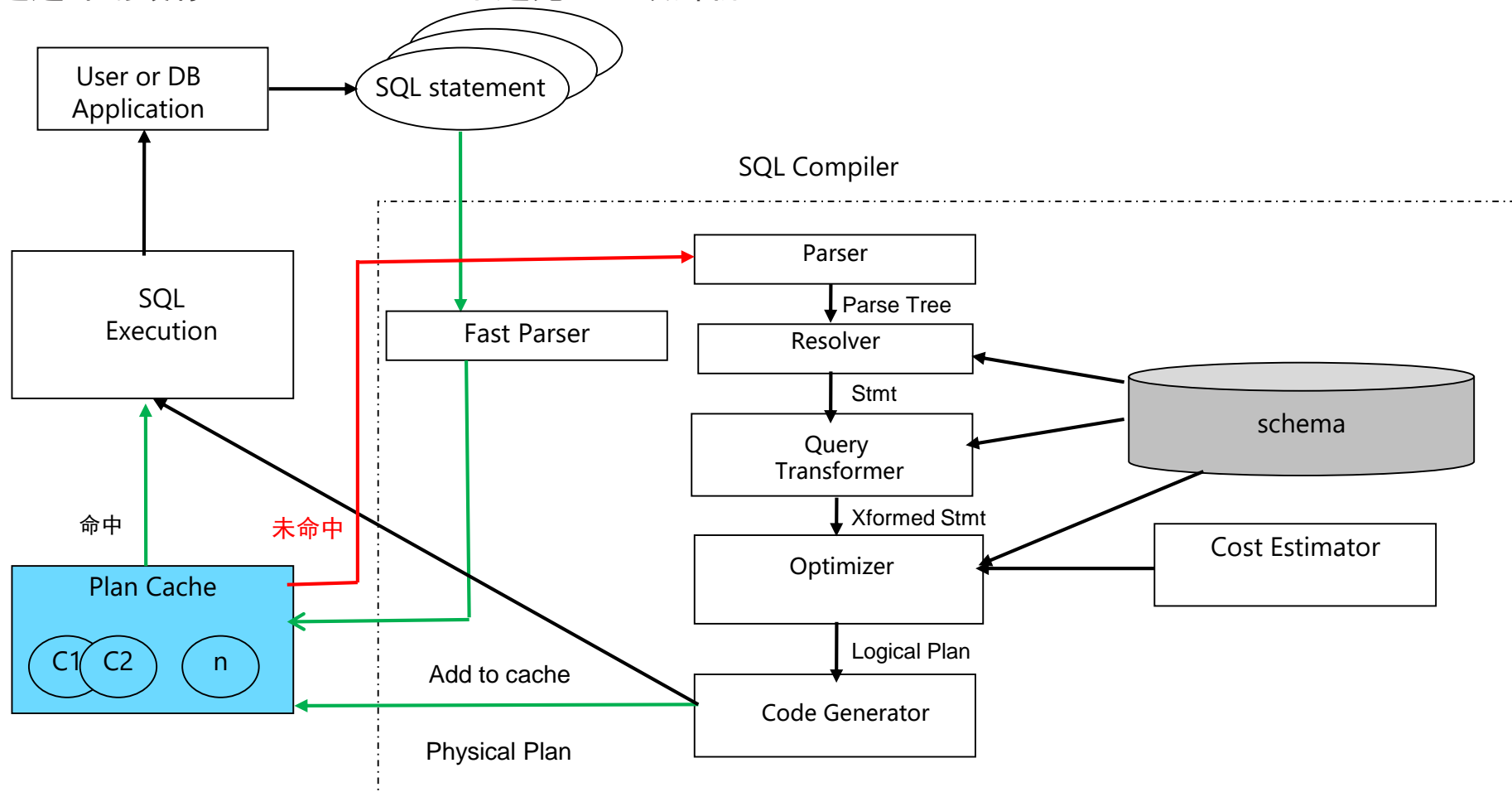
OPERATOR	NAME	ROWS	COST
PHY_SCALAR_AGGREGATE	NULL	1	353261533
PHY_PX_FIFO_COORD	NULL	1	298249961
PHY_PX_REDUCE_TRANSMIT	NULL	1	298249961
PHY_MERGE_GROUP_BY	NULL	1	298249961
PHY_GRANULE_ITERATOR	NULL	288000000	243238388
PHY_HASH_JOIN	NULL	288000000	243238388
PHY_TABLE_SCAN	t1	24000	27264
PHY_TABLE_SCAN	t2	24000	27264



3.6 执行计划缓存

3.6 执行计划缓存

- 一次完整的语法解析、语义分析、查询改写、查询优化、代码生成的SQL编译流程称为一次“硬解析”，硬解析生成执行计划的过程比较耗时（一般为毫秒级），这对于OLTP语句来说是很难接受的。
- OceanBase通过计划缓存（Plan Cache）来避免SQL硬解析



3.6 计划缓存例子

Remote计划

参数化SQL

第一次优化的查询

环境参数

```
mysql> select * from v$sqlplan_cache_plan_stat limit 1 \G
***** 1. row *****
      tenant_id: 1
        svr_ip: 10.101.163.82
        svr_port: 43225
        plan_id: 391
        sql_id: 555FFA5F489B7ACCB66F9C830F03B09
          type: 2
    statement: SELECT data_version,row_count,data_size FROM
__all_root_table WHERE TENANT_ID = ? AND TABLE_ID=? AND PARTITION_ID=? ORDER
BY data_version DESC
    query_sql: SELECT data_version,row_count,data_size FROM
__all_root_table WHERE TENANT_ID = 1 AND TABLE_ID=1 AND PARTITION_ID=1 ORDER
BY data_version DESC
    special_params:
      sys_vars: 45,4194304,2,4,1,0,0,32,1,0,1,1,0,10485760,1
      plan_hash: 17638921308295333636
    first_load_time: 2017-05-19 14:52:24.579007
    schema_version: 1495176592902432
    merged_version: 2
    last_active_time: 2017-05-19 16:07:12.971074
      avg_exe_usec: 1304
    slowest_exe_time: 1970-01-01 08:00:00.000000
    slowest_exe_usec: 0
      slow_count: 0
      hit_count: 39
      plan_size: 65536
    executions: 40
      disk_reads: 0
    direct_writes: 0
      buffer_gets: 0
    application_wait_time: 0
    concurrency_wait_time: 0
      user_io_wait_time: 0
    rows_processed: 0
      elapsed_time: 52169
      cpu_time: 39823
      large_querys: 0
    delayed_large_querys: 0
      outline_version: 0
      outline_id: -1
      table_scan: 1
1 row in set (0.04 sec)
```


3.6 执行计划缓存的淘汰

➤ 自动淘汰

➤ 手动淘汰

- `alter system flush plan cache;` 命令

有些计划缓存暂不支持的场景：

(1) 内存大小超过 20 M 执行计划不加入计划缓存。

(2) 如果该计划为分布式计划且涉及多个表，则不加入计划缓存。

3.6 执行计划缓存—自动淘汰

- 当计划缓存占用的内存达到了需要淘汰计划的内存上限(即淘汰计划的高水位线)时，对计划缓存中的执行计划自动进行淘汰。
- 优先淘汰最久没被使用的执行计划，影响淘汰策略的参数和变量如下：

- `plan_cache_evict_interval` (parameter)：检查执行计划是否需要淘汰的间隔时间。
- `ob_plan_cache_percentage` (variable)：计划缓存可使用内存占租户内存的百分比（最多可使用内存为：租户内存上限 * `ob_plan_cache_percentage`/100）。
- `ob_plan_cache_evict_high_percentage` (variable)：计划缓存使用率（百分比）达到多少时，触发计划缓存的淘汰。
- `ob_plan_cache_evict_low_percentage` (variable)：计划缓存使用率（百分比）达到多少时，停止计划缓存的淘汰。

3.6 执行计划缓存—自动淘汰举例

➤ 如果租户内存大小为10G，并且变量设置如下：

- `ob_plan_cache_percentage = 10;`
- `ob_plan_cache_evict_high_percentage = 90;`
- `ob_plan_cache_evict_low_percentage = 50;`

则：

- 计划缓存内存上限绝对值 = $10\text{G} * 10 / 100 = 1\text{G}$;
- 淘汰计划的高水位线 = $1\text{G} * 90 / 100 = 0.9\text{G}$;
- 淘汰计划的低水位线 = $1\text{G} * 50 / 100 = 0.5\text{G}$;

➤ 当该租户在某个server上计划缓存使用超过0.9G时，会触发淘汰，优先淘汰最久没执行的计划；

➤ 当淘汰到使用内存只有0.5G时，则停止淘汰。

➤ 如果淘汰速度没有新计划生成速度快，计划缓存使用内存达到内存上限绝对值1G时，将不再往计划缓存中添加新计划，直到淘汰后使用的内存小于1G才会添加新计划到计划缓存中。

3.6 执行计划缓存—手动淘汰

- 手动删除计划缓存中的计划，忽略当前参数/变量的设置。
- 支持按租户、server删除计划缓存，或者全部删除缓存：

```
alter system flush plan cache [tenant_list] [global];
```

3.6 执行计划缓存的刷新

计划缓存中的执行计划因各种原因失效时，会将计划缓存中失效的计划 依据流量驱动地刷新（可能会导致新的计划）：

1. SQL中涉及的表的SCHEMA进行变更时（比如添加索引，删除或增加列等），该SQL在计划缓存中对应的执行计划将被刷新；
2. SQL中涉及的表的统计信息被更新时，该SQL对应的执行计划会被刷新，由于OceanBase在合并时统一进行统计信息的收集，因此每次合并之后，计划缓存中所有的计划将被刷新；
3. SQL进行outline计划绑定变更时，该SQL对应的执行计划会被刷新，更新为按绑定的outline生成的执行计划。

3.6 执行计划缓存的使用控制

➤ 系统变量控制

- ob_enable_plan_cache 设置为ture时表示SQL请求可以使用计划缓存，设置为false时表示SQL请求不使用计划缓存。可进行session级和global级设置，默认设置为true。

➤ Hint控制

- /*+use_plan_cache(none)*/，该hint表示请求不使用计划缓存
- /*+use_plan_cache(default)*/，该hint表示使用计划缓存

感谢学习