Postgres 学习手册： 查询优化

jask

09/24/2024

配置文件

主要包含三个文件：

postgresql.conf： 该文件包含一些通用设置，比如内存分配、新建 database 的默认存储位置、PostgreSQL 服务器的 IP 地址、日志的位置以及许多其他设置。

pg_hba.conf： 该文件用于控制 PostgreSQL 服务器的访问权限，具体包括：允许哪些用户连接到哪个数据库，允许哪些 IP 地址连接到本服务器，以及指定连接时使用的身份验证模式

pg_ident.conf： 如果该文件存在，则系统会基于文件内容将当前登录的操作系统用户映射为一个 PostgreSQL 数据库内部用户的身份来登录

postgresql.conf 中的配置

通过查询 pg_settings 视图即可查看所有配置项内容，无须打开配置文件

连接管理

强行中断语句执行或者终止连接是一种很不“优雅”的行为，应当尽量避免。在客户端应用程序中，首先应该采取措施防止或者想办法解决 SQL 语句出现失控（耗时长或者占资源多）的情况，然后再考虑通过在后台直接杀掉的方式处理。

要想终止正在执行的语句并杀掉连接，请使用以下步骤。

1. 查出活动连接列表及其进程 ID。

```
SELECT * FROM pg_stat_activity;
```

pg_stat_activity 视图包含每个连接上最近一次执行的语句、使用的用户名（usename 字段）、所在的 database 名（datname 字段）以及语句开始执行的时间。通过查询该视图可以找到需要终止的会话所对应的进程 ID。

2. 取消连接（假设对应的进程号是 1234）上的活动查询。

```
SELECT  pg_cancel_backend(1234);
```

该操作不会终止连接本身。

3. 终止该连接。

```
SELECT  pg_terminate_backend(1234);
```

PostgreSQL 支持在 SELECT 查询语句中调用函数。因此，尽管 pg_terminate_backend 和 pg_cancel_backend 一次仅能处理一个连接，但你可以通过在 SELECT 语句中调用函数的方式实现一次处理多个连接。如下是一个终止某个用户的所有连接的例子：

```
select pg_terminate_backend(pid) from pg_stat_activity where username='some_role';
```

基于表空间机制进行存储管理

PostgreSQL 在安装阶段会自动生成两个表空间：一个是 pg_default ，用于存储所有的用户级数据；另一个是 pg_global ，用于存储所有的系统级数据。这两个表空间就位于默认的数据文件夹下。你可以不受限地创建表空间并将其物理存储位置设定到任何一块物理磁盘上。你也可以为 database 设定默认表空间，这样该 database 中创建的任何新对象都会存储到此表空间上。你也可以将现存的数据库对象迁移到新的表空间中。

数据类型

数值类型

serial 类型

serial 类型和它的兄弟类型 bigserial 是两种可以自动生成递增整数值的数据类型，一般如果表本身的字段不适合作为主键字段，会增加一个专门的字段并指定为 serial 类型以作为主键。

文本类型

PostgreSQL 有三种最基础的文本数据类型：character （也称为 char ）、character varying （也称为 varchar ）和 text 。

常见的字符串操作包括：填充（lpad 、rpad ）、修整空白（rtrim 、ltrim 、trim 、btrim ）、提取子字符串（substring ）以及连接（|| ）。

split_part 函数可以将指定位置的元素从用固定分隔符分隔的字符串中取出来

string_to_array 函数可以将基于固定分隔符的字符串拆分为一个数组。

PostgreSQL 对正则表达式的支持是极其强大的。你可以设定查询返回结果的格式为表或者数组，并且对其进行极其复杂的替换和更新操作。包括逆向引用（back reference）在内的一些高级搜索方法都是支持的。

时间类型

除了常见的日期和时间类型，PostgreSQL 还支持时区，并能够按照不同的时区对夏令时进行自动转换。此外 PostgreSQL 还支持一些特殊的数据类型，如 interval ，该类型可以用于对日期时间进行数学运算。

数组类型

最基本的构造数组的方法就是一个个元素手动录入：

```
SELECT ARRAY[2001,2002,2003] As yrs;
```

如果数组元素存在于一个查询返回的结果集中，那么可以使用这个略复杂一些的构造函数 array() 来生成数组：

```
SELECT array(
SELECT DISTINCT date_part('year', log_ts)
FROM logs
ORDER BY date_part('year', log_ts)
);
```

把一个直接以字符串格式书写的数组转换为一个真正的数组：

```
SELECT '{Alex,Sonia}'::text[] As name, '{46,43}'::smallint[] As age;
```

PostgreSQL 支持使用 start:end 语法对数组进行拆分。操作结果是原数组的一个子数组。

```
SELECT fact_subcats[2:4] FROM census.lu_fact_types;
```

连接在一起：

```
SELECT fact_subcats[1:2] || fact_subcats[3:4] FROM census.lu_fact_types;
```

一般来说，我们会通过数组下标来引用数组元素，请特别注意 PostgreSQL 的数组下标从 1 开始。

如果你试图越界访问一个数组，也就是说数组下标已经超过了数组元素的个数，那么不会返回错误，而是会得到一个空值 NULL 。

区间类型

区间数据类型可以表达一个带有起始值和结束值的值区间。PostgreSQL 为区间类型提供了很多配套的运算符和函数，例如判定区间是否重叠，判定某个值是否落在区间内，以及将相邻的若干区间合并为一个完整的区间等。

int4range、int8range

: 这是整数型离散区间，其定义符合前闭后开的规范化要求。

numrange

: 这是连续区间，可以用于描述小数、浮点数或者双精度数字的区间。

daterange

这是不带时区信息的日期离散区间。

tsrange、tstzrange

这是时间戳（日期加时间）类型的连续区间，秒值部分支持小数。tsrange 不带时区信息，tstzrange 带时区信息。

JSON 数据类型

```
CREATE TABLE persons (id serial PRIMARY KEY, person json);
```

就可以创建一个 json 类型的字段。

查询时，最简单的方法就是使用路径指向符，也可以是路径数组。

XML 数据类型

在往一个 xml 数据类型的列中插入数据时，PostgreSQL 会自动判定并确保只有格式合法的 XML 才会创建成功。

全文检索

FTS 的核心是一个被称为"FTS 配置库"的东西。这个库记录了词与词之间的语义匹配规则，其依据是一本或者多本词典。

TSVector 原始文本向量 原始文本必须先被向量化然后才能通过 FTS 对其进行全文检索，向量化以后的内容需存储在一个单独的向量字段中，该向量字段使用的数据类型是 tsvector 。

要从原始文本中生成 tsvector 向量字段，需要先指定使用哪个 FTS 配置库。

原始文本经过向量化处理以后会变成一个很精简的单词库，这个库中的每个词都被称为"词素"(lexeme，即不能再拆解的单词或词组，如果拆解将失去原来的含义)，同时这个库已经剔除了前面介绍过的停止词。

tsvector 字段中记录了每个词素在原始文本中出现的位置。一个词出现的次数越多，其权重值也就越大。

这样每个词素都会对应至少一个位置信息，如果出现多次则对应多个位置信息，看起来就像是一个可变长或变短的向量，这也是 tsvector 这个名字的由来。

TSQueries 检索条件向量 对 FTS 来说，原始文本和检索条件都必须先被向量化然后才能使用。前面已经介绍过了如何针对原始文本创建 tsvector 向量字段，接下来将介绍如何对检索条件进行向量化处理。

FTS 检索机制中，用 tsquery 类型来表示向量化以后的检索条件。PostgreSQL 提供了若干函数来实现检索条件的向量化处理，包括 to_tsquery 、plainto_tsquery 和 phraseto_tsquery 。

表，约束，索引

表

那么在什么情况下应该使用 IDENTITY 替代 serial 呢？IDENTITY 语法的主要优点在于一个 identity 总是与所属表绑定的，其值的递增或者重置都是与表本身一体化管理的，不会受其他对象干扰。serial 类型则不是这样，它会在后台自动创建一个序列号生成器，这个序列号生成器可以与别的表共享也可以本表独享，当不需要该序列号生成器时需要手动删除它。

继承表

PostgreSQL 是唯一提供表继承功能的数据库。如果创建一张表（子表）时指定为继承自另一张表（父表），则建好的子表除了含有自己的字段外还会含有父表的所有字段。PostgreSQL 会记录下这个继承关系，这样一旦父表的结构发生了变化，子表的结构也会自动跟着变化。这种父子继承结构的表可以完美地适用于需要数据分区的场景。当查询父表时，PostgreSQL 会自动把子表的记录也取出来。

无日志表

对于发生磁盘故障或者系统崩溃后可以被重建的临时数据来说，其操作速度比可靠性更重要。PostgreSQL 从 9.1 版开始支持 UNLOGGED 修饰符，使用该修饰符可以创建无日志的表。

系统不会为这种表记录任何事务日志（业界一般也称为 WAL 日志，即 write-ahead log)。

无日志表的一大优势是其写入记录的速度远远超过普通的有日志表。

无日志表的另一个特性是它无法被纳入 PostgreSQL 的复制机制，因为复制机制依赖事务日志。pg_dump 有一个选项可以允许你跳过备份无日志的表。

```
CREATE UNLOGGED TABLE web_sessions (
        session_id text PRIMARY KEY,
        add_ts timestamptz,
        upd_ts timestamptz,
        session_state xml);
```

创建语句

约束机制

用户可以在创建约束时定制其各方面的属性，包括约束的名称、如何处理现有数据、级联生效条件选项、如何执行匹配算法、使用哪些索引以及在何种情况下约束可以不生效等。

外键约束

```
SET search_path=census, public;
ALTER TABLE facts ADD CONSTRAINT fk_facts_1 FOREIGN KEY (fact_type_id)
REFERENCES lu_fact_types (fact_type_id) ON UPDATE CASCADE ON DELETE RESTRICT;

CREATE INDEX fki_facts_1 ON facts (fact_type_id);
```

语句

- 我们在 **facts** 表和 **lu_fact_types** 表之间定义了一个外键约束关系。有了这个约束以后，如果主表 **lu_fact_types** 中不存在某 **fact_type_id** 的记录，那么从表 **fact** 中就不能插入该 **fact_type_id** 的记录。
- 我们定义了一个级联规则，实现了以下功能：**(1)** 如果主表 **lu_fact_type** 的 **fact_type_id** 字段值发生了变化，那么从表 **fact** 中相应记录的 **fact_type_id** 字段值会自动进行相应修改，以维持外键引用关系不变；**(2)** 如果从表 **fact** 中还存在某 **fact_type_id** 字段值的记录，那么主表 **lu_fact_type** 中相同 **fact_type_id** 字段值的记录就不允许被删除。ON DELETE RESTRICT 是默认行为模式，也就是说这个子句不加也可以，但为了清晰起见最好加上。
- PostgreSQL 在建立主键约束和唯一性约束时，会自动为相应字段建立索引，但在建立外键约束时却不会，这一点需要注意。你需要为外键字段手动建立索引，以加快关联引用时的查询速度。

唯一性约束

主键字段的值是唯一的，但每张表只能定义一个主键，因此如果你需要保证别的字段值唯一，那么必须在该字段上建立唯一性约束或者唯一索引。建立唯一性约束时会自动在后台创建一个相应的唯一索引。

check 约束

check 约束能够给表的一个或者多个字段加上一个条件，表中每一行记录必须满足此条件。查询规划器也会利用 check 约束来优化执行速度。

排他性约束

传统的唯一性约束在比较算法中仅使用了“等于”运算符，即保证了指定字段的值在本表的任意两行记录中都不相等，而排他性约束机制拓展了唯一性比较算法机制，可以使用更多的运算符来进行比较运算。

排他性约束适用的另一个场景是处理数组类型的数据。

索引

B-树索引

主键约束和唯一性约束唯一支持的后台索引就是 B-树索引。

BRIN 索引

BRIN（block range index，块范围索引）是 PostgreSQL 9.4 中引入的一种索引类型，其设计目的是针对超大表做索引，在这种表上创建 B-树索引耗费的空间过大，以至于无法全部容纳在内存中，这会导致内存和磁盘间的索引数据块换入换出，从而严重影响查询速度。

GiST 索引

GiST（generalized search tree，通用搜索树）主要的适用场景包括全文检索以及空间数据、科学数据、非结构化数据和层次化数据的搜索。该类索引不能用于保障字段的唯一性，也就是说建立了该类型索引的字段上可插入重复值，但如果把该类索引用于排他性约束就可以实现唯一性保障。

GiST 是一种有损索引。

GIN 索引

GIN（generalized inverted index，通用逆序索引）主要适用于 PostgreSQL 内置的全文搜索引擎以及二进制 json 数据类型。

GIN 其实是从 GiST 派生出来的一种索引类型，但它是无损的，也就是说索引中会包含有被索引字段的值。

SP-GiST 索引

SP-GiST 是指基于空间分区树（space-partitioning trees）算法的 GiST 索引。该类型的索引与 GiST 索引的适用领域相同，但对于某些特定领域的数据算法，其效率会更高一些。

散列索引

散列索引在 GiST 和 GIN 索引出现前就已经得到了广泛使用。业界普遍认为 GiST 和 GIN 索引在性能和事务安全性方面要胜过散列索引。PostgreSQL 10 之前的版本中，事务日志中不会记录散列索引的变化，那么在流式复制环境中就不能使用散列索引，否则会导致修改无法被同步。

基于 B-树算法的 GiST 和 GIN 索引

这两类混合算法索引的优势在于，它们既能够支持 GiST 和 GIN 索引特有的运算符，又具有 B-树索引对于“等于”运算符的良好支持。当需要建立同时包含简单和复杂数据类型的多列复合索引时，你会发现这两类索引不可或缺。

多列索引

PostgreSQL 的规划器在语句执行过程中会自动使用一种被称为“位图索引扫描”的策略来同时使用多个索引。该策略可以使得多个单列索引同时发挥作用，达到的效果与使用单个复合索引相同。

假设你建了一个 B-树多列索引，其中包含 type 和 upper(name) 两个字段，那么完全没必要针对 type 字段再单独建立一个索引，因为规划器即使在遇到只有 type 单字段的查询条件时，也会自动使用该多列索引，这是规划器的一项基本能力。如果查询条件字段没有从多列索引中的第一个字段开始匹配，规划器其实也能用上索引，但请尽量避免这种情况，因为从索引原理上说，从索引的第一个字段开始匹配才是最高效的。

规划器支持一种仅依赖索引内数据的查询策略（index-only scan），也就是说如果查询的目标字段在索引内都有，那么直接扫描索引就可以得到查询结果，根本不需要访问表的本体了。

特色 SQL

CTE 表达式

公用表表达式（CTE）本质上来说就是在一个非常庞大的 SQL 语句中，允许用户通过一个子查询语句先定义出一个临时表，然后在这个庞大的 SQL 语句的不同地方都可以直接使用这个临时表。CTE 本质上就是当前语句执行期间内有效的临时表，一旦当前语句执行完毕，其内部的 CTE 表也随之失效。

```
WITH cte AS (
    SELECT
        tract_id, substring(tract_id,1, 5) As county_code,
        COUNT(*) OVER(PARTITION BY substring(tract_id,1, 5)) As cnt_tracts
    FROM census.lu_tracts
)
SELECT MAX(tract_id) As last_tract, county_code, cnt_tracts
FROM cte
WHERE cnt_tracts > 100
GROUP BY county_code, cnt_tracts;
```

with 后面的就是 CTE 表达式，本地是由一个 SELECT 语句定义出来的。

技术内幕

查询树

Node 结构

(src/include/backend/nodes/node.h:593) PostgreSQL 数据库中的结构体采用了统一的形式，它们都是基于 Node 结构体进行的"扩展"，Node 结构体中只包含一个 NodeTag 成员变量，NodeTag 是 enum（枚举）类型。

其他的结构体则利用 C 语言的特性对 Node 结构体进行扩展，所有结构体的第一个成员变量也是 NodeTag 枚举类型，例如在 List 结构体里，第一个成员变量是 NodeTag，它可能的值是 T_List、T_intList 或者 T_OidList，这样就能分别指代不同类型的 List。

List(pg_list.h):

```
typedef struct List
{
    NodeTag     type;           /* T_List, T_IntList, or T_OidList */
    int         length;
    ListCell   *head;
    ListCell   *tail;
} List;
```

而 Query 结构体也是以 NodeTag 枚举类型作为第一个变量，它的取值为 T_Query。

这样无论是 List 结构体的指针，还是 Query 结构体的指针，我们都能通过 Node 结构体的指针（Node*）来表示，而在使用对应的结构体时，则通过查看 Node 类型的指针中的 NodeTag 枚举类型就可以区分出该 Node 指针所代表的结构体的实际类型。

var 结构

Var 结构体表示查询中涉及的表的列属性，在 SQL 语句中，投影的列属性、约束条件中的列属性都是通过 Var 来表示的，在语法分析阶段会将列属性用 ColumnRef 结构体来表示，在语义分析阶段会将语法树中的 ColumnRef 替换成 Var 用来表示一个列属性。

Var(primenodes.h)

```
typedef struct Var
{
    Expr        xpr;
    Index       varno;          /* index of this var's relation in the range
                                 * table, or INNER_VAR/OUTER_VAR/INDEX_VAR */
    AttrNumber  varattno;       /* attribute number of this var, or zero for
                                 * all attrs ("whole-row Var") */
    Oid         vartype;        /* pg_type OID for the type of this var */
    int32       vartypmod;      /* pg_attribute typmod value */
    Oid         varcollid;      /* OID of collation, or InvalidOid if none */
    Index       varlevelsup;    /* for subquery variables referencing outer
                                 * relations; 0 in a normal var, >0 means N
                                 * levels up */
    Index       varnoold;       /* original value of varno, for debugging */
    AttrNumber  varoattno;      /* original value of varattno */
    int         location;       /* token location, or -1 if unknown */
} Var;
```

varno：用来确定列属性所在的表的"编号"，这个编号源自 Query（查询树）中的 rtable 成员变量，查询语句中涉及的每个表都会记录在 rtable 中，而其在 rtable 中的"编号"(也就是处于链表的第几个，从 1 开始计数，这个编号用 rtindex 表示）是唯一确定的值，在逻辑优化、物理优化的过程中可能 varno 都是 rindex，而在生成执行计划的阶段，它可能不再代表 rtable 中的编号。

varattno/vartype/vartypmod：varattno 确定了这个列属性是表中的第几列，vartype 和 vartypmod 则和列属性的类型有关。在创建表的时候，PostgreSQL 数据库会按照 SQL 语句中指定的列的顺序给列属性编号，并将编号记录在 PG_ATTRIBUTES 系统表中，同时会将 SQL 语句指定的列的类型也记录到 PG_ATTRIBUTES 中，因此可以说 varattno、vartype、vartypemod 都是取自 PG_ATTRIBUTES 系统表中的。

varlevelsup：确定了列属性对应的表所在的层次，这个层次值是一个相对值。

varnoold/varoattno：通常和 varno/varattno 相同，但是在等价变化的过程中，varno/varattno 的值可能发生变化，而 varnoold/varoattno 记录变化前的初值。

RangeTblEntry 结构体

RangeTblEntry（范围表，简称 RTE）描述了查询中出现的表，它通常出现在查询语句的 FROM 子句中，范围表中既有常规意义上的堆表，还有子查询、连接表等。

RangeTblEntry(parsenodes.h)

```
typedef enum RTEKind
{
    RTE_RELATION,               /* ordinary relation reference */
    RTE_SUBQUERY,               /* subquery in FROM */
    RTE_JOIN,                   /* join */
    RTE_FUNCTION,               /* function in FROM */
    RTE_TABLEFUNC,              /* TableFunc(.., column list) */
    RTE_VALUES,                 /* VALUES (<exprlist>), (<exprlist>), ... */
    RTE_CTE,                    /* common table expr (WITH list element) */
    RTE_NAMEDTUPLESTORE,        /* tuplestore, e.g. for AFTER triggers */

    RTE_VOID,                   /* PX: deleted RTE */
    RTE_TABLEFUNCTION,          /* PX: Functions over multiset input */
} RTEKind;

typedef struct RangeTblEntry
{
    NodeTag     type;

    RTEKind     rtekind;        /* see above */

    /*
     * XXX the fields applicable to only some rte kinds should be merged into
     * a union.  I didn't do this yet because the diffs would impact a lot of
     * code that is being actively worked on.  FIXME someday.
     */

    /*
     * Fields valid for a plain relation RTE (else zero):
     *
     * As a special case, RTE_NAMEDTUPLESTORE can also set relid to indicate
     * that the tuple format of the tuplestore is the same as the referenced
     * relation.  This allows plans referencing AFTER trigger transition
     * tables to be invalidated if the underlying table is altered.
     */
    Oid         relid;          /* OID of the relation */
    char        relkind;        /* relation kind (see pg_class.relkind) */
    struct TableSampleClause *tablesample;  /* sampling info, or NULL */

    /*
     * Fields valid for a subquery RTE (else NULL):
     */
    Query      *subquery;       /* the sub-query */
    bool        security_barrier;   /* is from security_barrier view? */

    /*
     * Fields valid for a join RTE (else NULL/zero):
     *
     * joinaliasvars is a list of (usually) Vars corresponding to the columns
     * of the join result.  An alias Var referencing column K of the join
     * result can be replaced by the K'th element of joinaliasvars --- but to
     * simplify the task of reverse-listing aliases correctly, we do not do
     * that until planning time.  In detail: an element of joinaliasvars can
     * be a Var of one of the join's input relations, or such a Var with an
     * implicit coercion to the join's output column type, or a COALESCE
     * expression containing the two input column Vars (possibly coerced).
```

```
     * Within a Query loaded from a stored rule, it is also possible for
     * joinaliasvars items to be null pointers, which are placeholders for
     * (necessarily unreferenced) columns dropped since the rule was made.
     * Also, once planning begins, joinaliasvars items can be almost anything,
     * as a result of subquery-flattening substitutions.
     */
    JoinType    jointype;       /* type of join */
    List       *joinaliasvars;  /* list of alias-var expansions */

    /*
     * Fields valid for a function RTE (else NIL/zero):
     *
     * When funcordinality is true, the eref->colnames list includes an alias
     * for the ordinality column.  The ordinality column is otherwise
     * implicit, and must be accounted for "by hand" in places such as
     * expandRTE().
     */
    List       *functions;      /* list of RangeTblFunction nodes */
    bool        funcordinality; /* is this called WITH ORDINALITY? */

    /*
     * Fields valid for a TableFunc RTE (else NULL):
     */
    TableFunc  *tablefunc;

    /*
     * Fields valid for a values RTE (else NIL):
     */
    List       *values_lists;   /* list of expression lists */

    /*
     * Fields valid for a CTE RTE (else NULL/zero):
     */
    char       *ctename;        /* name of the WITH list item */
    Index       ctelevelsup;    /* number of query levels up */
    bool        self_reference; /* is this a recursive self-reference? */

    /*
     * Fields valid for CTE, VALUES, ENR, and TableFunc RTEs (else NIL):
     *
     * We need these for CTE RTEs so that the types of self-referential
     * columns are well-defined.  For VALUES RTEs, storing these explicitly
     * saves having to re-determine the info by scanning the values_lists. For
     * ENRs, we store the types explicitly here (we could get the information
     * from the catalogs if 'relid' was supplied, but we'd still need these
     * for TupleDesc-based ENRs, so we might as well always store the type
     * info here).  For TableFuncs, these fields are redundant with data in
     * the TableFunc node, but keeping them here allows some code sharing with
     * the other cases.
     *
     * For ENRs only, we have to consider the possibility of dropped columns.
     * A dropped column is included in these lists, but it will have zeroes in
     * all three lists (as well as an empty-string entry in eref).  Testing
     * for zero coltype is the standard way to detect a dropped column.
     */
    List       *coltypes;       /* OID list of column type OIDs */
    List       *coltypmods;     /* integer list of column typmods */
    List       *colcollations;  /* OID list of column collation OIDs */

    /*
     * Fields valid for ENR RTEs (else NULL/zero):
```

```
     */
    char       *enrname;        /* name of ephemeral named relation */
    double      enrtuples;      /* estimated or actual from caller */

    /*
     * Fields valid in all RTEs:
     */
    Alias      *alias;          /* user-written alias clause, if any */
    Alias      *eref;           /* expanded reference names */
    bool        lateral;        /* subquery, function, or values is LATERAL? */
    bool        inh;            /* inheritance requested? */
    bool        inFromCl;       /* present in FROM clause? */
    AclMode     requiredPerms;  /* bitmask of required access permissions */
    Oid         checkAsUser;    /* if valid, check access as this role */
    Bitmapset  *selectedCols;   /* columns needing SELECT permission */
    Bitmapset  *insertedCols;   /* columns needing INSERT permission */
    Bitmapset  *updatedCols;    /* columns needing UPDATE permission */
    List       *securityQuals;  /* security barrier quals to apply, if any */

    /* POLAR px */
    List       *values_collations;      /* OID list of column collation OIDs */
    List       *ctecoltypes;            /* OID list of column type OIDs */
    List       *ctecoltypmods;          /* integer list of column typmods */
    List       *ctecolcollations;       /* OID list of column collation OIDs */
} RangeTblEntry;
```

RangeTblEntry 根据成员变量 rtekind 来确定自己的类型，类型不同 RangeTblEntry 成员变量的作用也不同，例如针对 RTE_RELATION 类型的 RangeTblEntry，成员变量 relid 和 relkind 就是有用的，而对于 RTE_SUBQUERY 类型的 RangeTblEntry，它是一个子查询衍生的 RangeTblEntry，本身没有 relid 和 relkind，因此 relid 和 relkind 这时就处于无用的状态，而这时有效的是 subquery 成员变量。

RangeTblRed 结构体

RangeTblEntry 只保留在查询树的 Query-> rtable 链表中，而链表是一个线性结构，它如何保存树状的关系代数表达式中的连接操作呢？答案是在 Query-> jointree 中保存各个范围表之间的连接关系。

如果在 Query-> jointree 中还保存同样的一份 RangeTblEntry，那么一方面会造成存储的冗余，另一方面也容易产生数据不一致的问题，因此在查询树的其他任何地方都不再存放新的 RangeTblEntry，每个范围表在 Query-> rtable 链表中有且只能有一个，在其他地方使用到范围表都使用 RangeTblRef 来代替。RangeTblRef 是对 RangeTblEntry 的引用，因为 RangeTblEntry 在 Query-> rtable 中的位置是确定的，因此可以用它在 Query-> rtable 链表中的位置 rtindex 来标识。

primenodes.h

```
typedef struct RangeTblRef
{
    NodeTag     type;
    int         rtindex;
} RangeTblRef;
```

JoinExpr 结构体

在查询语句中如果显式地指定两个表之间的连接关系，例如 A LEFT JOIN B ON Pab 这种形式，就需要一个 JoinExpr 结构体来表示它们

```
typedef struct JoinExpr
{
    NodeTag     type;
    JoinType    jointype;       /* type of join */
    bool        isNatural;      /* Natural join? Will need to shape table */
    Node       *larg;           /* left subtree */
    Node       *rarg;           /* right subtree */
    List       *usingClause;    /* USING clause, if any (list of String) */
    Node       *quals;          /* qualifiers on join, if any */
    Alias      *alias;          /* user-written alias clause, if any */
```

```
    int         rtindex;        /* RT index assigned for join, or 0 */
} JoinExpr;
```

例 1：SELECT * FROM STUDENT INNER JOIN SCORE ON STUDENT.sno = SCORE.sno; 这个示例语句明确指定了两个表进行内连接操作，因此它们需要用 JoinExpr 来表示，JoinExpr-> quals 中保存的是 STUDENT.sno = SCORE.sno。

例 2：SELECT * FROM STUDENT LEFT JOIN SCORE ON STUDENT.sno = SCORE.sno; 这个示例语句中明确指定了两个表进行左外连接操作，因此它也必须使用 JoinExpr 来表示，JoinExpr-> quals 中保存的是 STUDENT.sno = SCORE.sno。

例 3：SELECT * FROM STUDENT LEFT JOIN SCORE ON TRUE LEFT JOIN COURSE ON SCORE.cno = COURSE.cno; 这个示例语句中明确指定了 3 个表之间的连接关系，它需要有两个 JoinExpr 来表示，如图 2-2 所示。

FromExpr 结构体

FromExpr 和 JoinExpr 是用来表示表之间的连接关系的结构体，通常来说，FromExpr 中的各个表之间的连接关系是 Inner Join，这样就可以在 FromExpr-> fromlist 中保存任意多个表，默认它们之间是内连接的关系，我们先来看一下它的结构体的定义。

```
typedef struct FromExpr
{
    NodeTag     type;
    List       *fromlist;       /* List of join subtrees */
    Node       *quals;          /* qualifiers on join, if any */
} FromExpr;
```

例 1：SELECT * FROM STUDENT, SCORE, COURSE WHERE STUDENT.sno = SCORE.sno；这个示例中的 3 个表没有明确地指定连接关系，默认它们是 InnerJoin，因此可以通过 FromExpr 来表示，FromExpr-> fromlist 中保存了语句中的 3 个表，FromExpr-> quals 中保存了 STUDENT.sno= SCORE.sno 这个约束条件。

Query 结构体

parsenodes.h

```
//PolarDB 的定义
typedef struct Query
{
    NodeTag     type;

    CmdType     commandType;    /* select|insert|update|delete|utility */

    QuerySource querySource;    /* where did I come from? */

    uint64      queryId;        /* query identifier (can be set by plugins) */

    bool        canSetTag;      /* do I set the command result tag? */

    Node       *utilityStmt;    /* non-null if commandType == CMD_UTILITY */

    int         resultRelation; /* rtable index of target relation for
                                 * INSERT/UPDATE/DELETE; 0 for SELECT */

    bool        hasAggs;        /* has aggregates in tlist or havingQual */
    bool        hasWindowFuncs; /* has window functions in tlist */
    bool        hasTargetSRFs;  /* has set-returning functions in tlist */
    bool        hasSubLinks;    /* has subquery SubLink */
    bool        hasDistinctOn;  /* distinctClause is from DISTINCT ON */
    bool        hasRecursive;   /* WITH RECURSIVE was specified */
    bool        hasModifyingCTE;     /* has INSERT/UPDATE/DELETE in WITH */
    bool        hasForUpdate;   /* FOR [KEY] UPDATE/SHARE was specified */
    bool        hasRowSecurity; /* rewriter has applied some RLS policy */

    List       *cteList;        /* WITH list (of CommonTableExpr's) */

    List       *rtable;         /* list of range table entries */
    FromExpr   *jointree;       /* table join tree (FROM and WHERE clauses) */
```

```
    List       *targetList;     /* target list (of TargetEntry) */

    OverridingKind override;    /* OVERRIDING clause */

    OnConflictExpr *onConflict; /* ON CONFLICT DO [NOTHING | UPDATE] */

    List       *returningList;  /* return-values list (of TargetEntry) */

    List       *groupClause;    /* a list of SortGroupClause's */

    List       *groupingSets;   /* a list of GroupingSet's if present */

    Node       *havingQual;     /* qualifications applied to groups */

    List       *windowClause;   /* a list of WindowClause's */

    List       *distinctClause; /* a list of SortGroupClause's */

    List       *sortClause;     /* a list of SortGroupClause's */

    Node       *limitOffset;    /* # of result tuples to skip (int8 expr) */
    Node       *limitCount;     /* # of result tuples to return (int8 expr) */

    List       *rowMarks;       /* a list of RowMarkClause's */

    Node       *setOperations;  /* set-operation tree if this is top level of
                                 * a UNION/INTERSECT/EXCEPT query */

    List       *constraintDeps; /* a list of pg_constraint OIDs that the query
                                 * depends on to be semantically valid */

    List       *withCheckOptions;   /* a list of WithCheckOption's, which are
                                     * only added during rewrite and therefore
                                     * are not written out as part of Query. */

    /*
     * The following two fields identify the portion of the source text string
     * containing this query.  They are typically only populated in top-level
     * Queries, not in sub-queries.  When not set, they might both be zero, or
     * both be -1 meaning "unknown".
     */
    int         stmt_location;  /* start location, or -1 if unknown */
    int         stmt_len;       /* length in bytes; 0 means "rest of string" */

    /*
     *  POLAR px
     * Used to indicate this query is part of CTAS or COPY so that its plan
     * would always be dispatched in parallel.
     */
    ParentStmtType  parentStmtType;
    /* POLAR end */
} Query;
//Postgres 的定义
typedef struct Query
{
    NodeTag     type;

    CmdType     commandType;    /* select|insert|update|delete|merge|utility */

    /* where did I come from? */
```

```
QuerySource querySource pg_node_attr(query_jumble_ignore);

/*
 * query identifier (can be set by plugins); ignored for equal, as it
 * might not be set; also not stored.  This is the result of the query
 * jumble, hence ignored.
 */
uint64      queryId pg_node_attr(equal_ignore, query_jumble_ignore, read_write_ignore, read_as(0));

/* do I set the command result tag? */
bool        canSetTag pg_node_attr(query_jumble_ignore);

Node       *utilityStmt;    /* non-null if commandType == CMD_UTILITY */

/*
 * rtable index of target relation for INSERT/UPDATE/DELETE/MERGE; 0 for
 * SELECT.  This is ignored in the query jumble as unrelated to the
 * compilation of the query ID.
 */
int         resultRelation pg_node_attr(query_jumble_ignore);

/* has aggregates in tlist or havingQual */
bool        hasAggs pg_node_attr(query_jumble_ignore);
/* has window functions in tlist */
bool        hasWindowFuncs pg_node_attr(query_jumble_ignore);
/* has set-returning functions in tlist */
bool        hasTargetSRFs pg_node_attr(query_jumble_ignore);
/* has subquery SubLink */
bool        hasSubLinks pg_node_attr(query_jumble_ignore);
/* distinctClause is from DISTINCT ON */
bool        hasDistinctOn pg_node_attr(query_jumble_ignore);
/* WITH RECURSIVE was specified */
bool        hasRecursive pg_node_attr(query_jumble_ignore);
/* has INSERT/UPDATE/DELETE/MERGE in WITH */
bool        hasModifyingCTE pg_node_attr(query_jumble_ignore);
/* FOR [KEY] UPDATE/SHARE was specified */
bool        hasForUpdate pg_node_attr(query_jumble_ignore);
/* rewriter has applied some RLS policy */
bool        hasRowSecurity pg_node_attr(query_jumble_ignore);
/* is a RETURN statement */
bool        isReturn pg_node_attr(query_jumble_ignore);

List       *cteList;        /* WITH list (of CommonTableExpr's) */

List       *rtable;         /* list of range table entries */

/*
 * list of RTEPermissionInfo nodes for the rtable entries having
 * perminfoindex > 0
 */
List       *rteperminfos pg_node_attr(query_jumble_ignore);
FromExpr   *jointree;       /* table join tree (FROM and WHERE clauses);
                             * also USING clause for MERGE */

List       *mergeActionList;    /* list of actions for MERGE (only) */

/*
 * rtable index of target relation for MERGE to pull data. Initially, this
 * is the same as resultRelation, but after query rewriting, if the target
 * relation is a trigger-updatable view, this is the index of the expanded
 * view subquery, whereas resultRelation is the index of the target view.
```

```
     */
    int         mergeTargetRelation pg_node_attr(query_jumble_ignore);

    /* join condition between source and target for MERGE */
    Node       *mergeJoinCondition;

    List       *targetList;     /* target list (of TargetEntry) */

    /* OVERRIDING clause */
    OverridingKind override pg_node_attr(query_jumble_ignore);

    OnConflictExpr *onConflict; /* ON CONFLICT DO [NOTHING | UPDATE] */

    List       *returningList;  /* return-values list (of TargetEntry) */

    List       *groupClause;    /* a list of SortGroupClause's */
    bool        groupDistinct;  /* is the group by clause distinct? */

    List       *groupingSets;   /* a list of GroupingSet's if present */

    Node       *havingQual;     /* qualifications applied to groups */

    List       *windowClause;   /* a list of WindowClause's */

    List       *distinctClause; /* a list of SortGroupClause's */

    List       *sortClause;     /* a list of SortGroupClause's */

    Node       *limitOffset;    /* # of result tuples to skip (int8 expr) */
    Node       *limitCount;     /* # of result tuples to return (int8 expr) */
    LimitOption limitOption;    /* limit type */

    List       *rowMarks;       /* a list of RowMarkClause's */

    Node       *setOperations;  /* set-operation tree if this is top level of
                                 * a UNION/INTERSECT/EXCEPT query */

    /*
     * A list of pg_constraint OIDs that the query depends on to be
     * semantically valid
     */
    List       *constraintDeps pg_node_attr(query_jumble_ignore);

    /* a list of WithCheckOption's (added during rewrite) */
    List       *withCheckOptions pg_node_attr(query_jumble_ignore);

    /*
     * The following two fields identify the portion of the source text string
     * containing this query.  They are typically only populated in top-level
     * Queries, not in sub-queries.  When not set, they might both be zero, or
     * both be -1 meaning "unknown".
     */
    /* start location, or -1 if unknown */
    ParseLoc    stmt_location;
    /* length in bytes; 0 means "rest of string" */
    ParseLoc    stmt_len pg_node_attr(query_jumble_ignore);
} Query;
```

Query 结构体是查询优化模块的输入参数，其源自于语法分析模块，一个 SQL 语句在执行过程中，经过词法分析、语法分析和语义分析之后，会生成一棵查询树，PostgreSQL 用 Query 结构体来表示查询树。

查询优化模块在获取到查询树之后，开始对查询树进行逻辑优化，也就是对查询树进行等价变换，将其重写成一棵新的查询树，这个

新的查询树又作为物理优化的输入参数，进行物理优化。

rtable：在查询中 FROM 子句后面会指出需要进行查询的范围表，可能是对单个范围表进行查询，也可能是对几个范围表做连接操作，rtable 中则记录了这些范围表。rtable 是一个 List 指针，所有要查询的范围表就记录在这个 List 中，每个表以 RangeTblEntry 结构体来表示，因此 rtable 是一个以 RangeTblEntry 为节点的 List 链表。

jointree：rtable 中列出了查询语句中的表，但没有明确指出各个表之间的连接关系，这个连接的关系则通过 jointree 来标明，jointree 是一个 FromExpr 类型的结构体，它有 3 种类型的节点：FromExpr、JoinExpr 和 RangeTblRef。

targetlist：targetlist 中包含了需要投影（Project）的列，也就是 SFW 查询中的投影列。

查询树展示

PostgreSQL 数据库提供了参数让我们查看查询树和执行计划树：debug_pretty_print, debug_print_parse, debug_print_rewritten, debug_print_plan。

查询树的遍历

在对查询树进行重写的过程中，需要经常性地对查询树（Query）进行遍历，从中查找、替换、编辑某个结构体的值，以实现重写的功能。例如在子查询提升的过程中，由于在父查询和子查询中的 Var 具有不同的层次关系，因此它们的 varlevelsup 是不同的，但是在子查询提升之后，这种层次关系就消失了，因此需要调整一些 Var 的 varlevelsup 的值，这时候就需要从查询树中找出这些 Var 并编辑或替换这些 Var。

可以使用 explain 来查看优化。

逻辑重写优化

通用表达式

WITH 语句，作用于子查询类似，但是数据库不会对通用表达式做提升优化。

子查询提升

应用程序通过 SQL 语句来操作数据库时会使用大量的子查询，这种写法比直接对两个表做连接操作结构上要更清晰，尤其是在实现一些比较复杂的查询语句时，子查询由于具有一定的独立性，会使写出的 SQL 脚本更容易理解。

相关子查询：指在子查询语句中引用了外层表的列属性，这就导致外层表每获得一个元组，子查询就需要重新执行一次；

非相关子查询：指在子查询语句是独立的，和外层的表没有直接的关联，子查询可以单独执行一次，外层表可以重复利用子查询的执行结果。

由此对于 ANY 类型的子连接我们也可以得到一个感性的原则：ANY 类型的非相关子连接可以提升（仍然需要是“简单”的子连接），并且可以通过物化的方式进行优化，而 ANY 类型的相关子连接目前还不能提升。

PostgreSQL 数据库为子连接定义了单独的结构体—SubLink 结构体，其中主要描述了子连接的类型、子连接的操作数及操作符等信息。

pull_up_sublink 函数

子连接提升的主要过程是在 pull_up_sublinks 函数中实现的，该函数的主要参数是 Query-> jointree。

pull_up_sublinks_qual_recurse 函数

pull_up_sublinks_qual_recurse 函数主要是对 Query-> jointree 进行递归分析，Query-> jointree 节点分为三种类型：RangeTblRef、FromExpr 或 JoinExpr，针对这 3 种类型的节点 pull_up_sublinks_qual_recurse 函数做了不同的处理。

1）RangeTblRef：RangeTblRef 一定是 Query-> jointree 上的叶子节点，因此是递归的结束条件，保存当前表的 relid 返回给上层，执行到这个分支通常有两种情况。

a）情况一：查询语句只有单表，没有连接操作，这种情况递归处理结束，另外去查看子连接是否满足提升的其他条件。

b）情况二：查询语句有连接关系，在对 FromExpr-> fromlist、JoinExpr-> larg 或者 JoinExpr-> rarg 递归的过程中，遍历到了叶子节点 RangeTblRef，这时候需要将这个 RangeTblRef 节点的 relids 返回给上一层，主要用于判断该子查询能否被提升

2）FromExpr：

a) 对 FromExpr-> fromlist 中的节点做递归遍历，对每个节点递归调用 pull_up_sublinks_jointree_recurse 函数，一直处理到叶子节点 RangeTblRef 才返回。b) 调用 pull_up_sublinks_qual_recurse 函数处理 FromExpr-> qual，对其中可能出现的 ANY_SUBLINK 或 EXISTS_SUBLINK 做处理。

3）JoinExpr：

a) 分别调用 pull_up_sublinks_jointree_recurse 函数递归处理 JoinExpr-> larg 和 JoinExpr-> rarg，一直处理到叶子节点 RangeTblRef 才返回。另外还需要根据连接操作的类型区分子连接是否能够被提升，以左连接为例，如果子连接的左操作数是 LHS 的列属性，则该子连接不能被提升，因为提升后两个查询树不等价。例如对于如下 SQL 语句，子连接是不能提升的。

提升子查询

子查询和子连接不同，它出现在 RangeTableEntry 中，它存储的是一个子查询树，如果这个子查询树不被提升，则经过查询优化之后形成一个子执行计划，上层执行计划和子查询计划做嵌套循环得到最终结果，在这个过程中，查询优化模块对这个子查询所能做的优化选择较少。如果这个子查询被提升，转换成与上层的连接（Join），由于查询优化模块对连接操作的优化做了很多工作，因此可能获得更好的执行计划。

RangeTblRef 结构体又细分成 RTE_SUBQUERY(simple)、RTE_SUBQUERY(union)、RTE_VALUES

对于 FromExpr，因为 FromExpr-> fromlist 中的范围表默认是内连接，所以只需要遍历 FromExpr-> fromlist 中的节点，这里主要判断下层的节点是否存在可删除的情况。

RTE_SUBQUERY(SIMPLE) 对于 RTE_SUBQUERY（simple）类型的子查询，我们先看一下它的提升条件，首先要求子查询的类型是“简单（simple）”的，所谓的简单，需要满足如下条件：

a）做一个确认，必须是一个子查询树，而且是 SELECT 查询语句。

b）子查询树的 subQuery-> setOperations 必须是 NULL，如果不是 NULL，应该先交给 pull_up_simple_union_all 函数去处理。

c）不能包含聚集操作、窗口函数、GROUP 操作等，在子连接提升的时候我们已经见过类似的条件。

d）如果没有范围表，那么在无约束条件或者满足删除条件或者不是外连接的情况下才能提升，这是因为在物理优化阶段所有的表都会建立一个 RelOptInfo，如果空范围表的子查询不提升，那么可以用 RelOptInfo 来表示子查询，然后可以将这个 RelOptInfo 优化成一个 Result 计划节点，如果将空范围表提升上来，那么无法用 RelOptInfo 表示它

e）Lateral 语义支持在子查询中引用上一层的表，但是如果引用的是更上层的表，可能会出现问题。

f）如果有易失性函数也不能提升。

RTE_SUBQUERY(UNION) 对于 RTE_SUBQUERY(union) 类型的子查询，它的主要问题是在子查询中如果有 UNION ALL 操作，就会出现“子子查询”

子查询的提升需要满足一些条件才能进行，这些条件的判断在 is_simple_union_all 函数中实现，分别如下：

a）必须是 SELECT 查询语句。

b）subquery-> setOperations 必须有值，且集合操作的操作符必须是 UNION ALL。

c）子查询中不能有 ORDER BY、LIMIT/OFFSET 等。

d）SetOperationStmt 中的子语句必须包含相同类型的投影列（is_simple_union_all_recurse 函数）。

RTE_VALUES RTE_VALUES 类型的子查询的 RangeTblEntry 中不止有 RTE_VALUES，而且包含 RTE_SUBQUERY，它的层次关系是：RTE_SUBQUERY 封装 RTE_VALUES，也就是说 RTE_VALUES 是保存在 RTE_SUBQUERY 的查询树 subquery-> rtable 中的，因此它的提升的过程中可能经历两次提升，一次是子查询 RTE_SUBQUERY(SIMPLE) 的提升，另一次是 RTE_VALUES 的提升。

UNION ALL 优化

在子查询的提升过程中，对“简单”的 UNION ALL 集合操作进行了提升，但是并没有处理顶层的 UNION ALL 集合操作，这里通过 flatten_simple_union_all 函数对顶层的 UNION ALL 进行处理，目的是将 UNION ALL 这种集合操作的形式转换为 AppendRelInfo 的形式

展开继承表

在子查询提升的过程中，对 UNION ALL 类型的操作符进行优化时已经涉及了继承表，因为 UNION ALL 和继承表有一定的相似性，最终都采用 AppendRelInfo 结构体来实现，UNION ALL 使用 AppendRelInfo 来封装子查询，提升子查询的层次，而继承表则用 AppendRelInfo 来记录子表。

继承表的使用对于用户是"透明"的，也就是说用户操作继承表的父表时，不需要了解继承表的细节，例如它有几个子表，每个子表的结构是什么样的，用户都无须知道，这些都由 PostgreSQL 数据库自动完成查询。

GROUP BY 键值消除

Group By 子句的实现需要借助排序或者 Hash 来实现，如果能减少 Group By 后面的字段，就能降低排序或者 Hash 带来的损耗。

对于一个有主键（Primary Key）的表，如果 Group By 的字段包含了主键的所有键值，实际上这个主键已经能够表示当前的结果就是符合分组的，因此可以将主键之外的字段去掉。例如对于 SQL 语句 SELECT * FROM STUDENT GROUP BY sno, sname,ssex，因为 sno 属性上有一个主键索引，所以 Group By 子句中的 sname 和 ssex 可以被去除掉。

逻辑分解优化

创建 RelOptInfo

在查询树中，将基表信息以 RangeTblEntry 的形式存放在 Query->rtable 链表中，在查询优化的后期，尤其是物理优化的阶段，RangeTblEntry 保存的信息已经无法满足代价计算的需要，因为针对每个基表都需要生成扫描路径（Scan），多个基表之间还会产生连接（Join）路径，并且需要计算这些路径的代价，因此需要一个新的结构体 RelOptInfo 来替换原来的 RangeTblEntry。

RelOptInfo 结构体

node/relation.h

```
typedef struct RelOptInfo
{
    NodeTag     type;

    RelOptKind  reloptkind;

    /* all relations included in this RelOptInfo */
    Relids      relids;         /* set of base relids (rangetable indexes) */

    /* size estimates generated by planner */
    double      rows;           /* estimated number of result tuples */

    /* per-relation planner control flags */
    bool        consider_startup;   /* keep cheap-startup-cost paths? */
    bool        consider_param_startup; /* ditto, for parameterized paths? */
    bool        consider_parallel;  /* consider parallel paths? */

    /* default result targetlist for Paths scanning this relation */
    struct PathTarget *reltarget;   /* list of Vars/Exprs, cost, width */

    /* materialization information */
    List       *pathlist;       /* Path structures */
    List       *ppilist;        /* ParamPathInfos used in pathlist */
    List       *partial_pathlist;   /* partial Paths */
    struct Path *cheapest_startup_path;
    struct Path *cheapest_total_path;
    struct Path *cheapest_unique_path;
    List       *cheapest_parameterized_paths;

    /* parameterization information needed for both base rels and join rels */
    /* (see also lateral_vars and lateral_referencers) */
    Relids      direct_lateral_relids;  /* rels directly laterally referenced */
    Relids      lateral_relids; /* minimum parameterization of rel */
```

```
    /* information about a base rel (not set for join rels!) */
    Index       relid;
    Oid         reltablespace;  /* containing tablespace */
    RTEKind     rtekind;        /* RELATION, SUBQUERY, FUNCTION, etc */
    AttrNumber  min_attr;       /* smallest attrno of rel (often <0) */
    AttrNumber  max_attr;       /* largest attrno of rel */
    Relids     *attr_needed;    /* array indexed [min_attr .. max_attr] */
    int32      *attr_widths;    /* array indexed [min_attr .. max_attr] */
    List       *lateral_vars;   /* LATERAL Vars and PHVs referenced by rel */
    Relids      lateral_referencers;    /* rels that reference me laterally */
    List       *indexlist;      /* list of IndexOptInfo */
    List       *statlist;       /* list of StatisticExtInfo */
    BlockNumber pages;          /* size estimates derived from pg_class */
    double      tuples;
    double      allvisfrac;
    PlannerInfo *subroot;       /* if subquery */
    List       *subplan_params; /* if subquery */
    int         rel_parallel_workers;   /* wanted number of parallel workers */

    /* Information about foreign tables and foreign joins */
    Oid         serverid;       /* identifies server for the table or join */
    Oid         userid;         /* identifies user to check access as */
    bool        useridiscurrent;    /* join is only valid for current user */
    /* use "struct FdwRoutine" to avoid including fdwapi.h here */
    struct FdwRoutine *fdwroutine;
    void       *fdw_private;

    /* cache space for remembering if we have proven this relation unique */
    List       *unique_for_rels;    /* known unique for these other relid
                                     * set(s) */
    List       *non_unique_for_rels;    /* known not unique for these set(s) */

    /* used by various scans and joins: */
    List       *baserestrictinfo;   /* RestrictInfo structures (if base rel) */
    QualCost    baserestrictcost;   /* cost of evaluating the above */
    Index       baserestrict_min_security;  /* min security_level found in
                                             * baserestrictinfo */
    List       *joininfo;       /* RestrictInfo structures for join clauses
                                 * involving this rel */
    bool        has_eclass_joins;   /* T means joininfo is incomplete */

    /* used by partitionwise joins: */
    bool        consider_partitionwise_join;    /* consider partitionwise
                                                 * join paths? (if
                                                 * partitioned rel) */
    Relids      top_parent_relids;  /* Relids of topmost parents (if "other"
                                     * rel) */

    /* used for partitioned relations */
    PartitionScheme part_scheme;    /* Partitioning scheme. */
    int         nparts;         /* number of partitions */
    struct PartitionBoundInfoData *boundinfo;   /* Partition bounds */
    List       *partition_qual; /* partition constraint */
    struct RelOptInfo **part_rels;  /* Array of RelOptInfos of partitions,
                                     * stored in the same order of bounds */
    List      **partexprs;      /* Non-nullable partition key expressions. */
    List      **nullable_partexprs; /* Nullable partition key expressions. */
    List       *partitioned_child_rels; /* List of RT indexes. */
} RelOptInfo;
```

在查询优化的过程中，我们首先面对的是 FROM 子句中的表，通常称之为范围表（RangeTable），它可能是一个常规意义上的表，也

可能是一个子查询，或者还可能是一个查询结果的组织为表状的函数（例如 TableFunction），这些表处于查询执行计划树的叶子节点，是产生最终查询结果的基础，我们称之为基表，这些基表可以用 RelOptInfo 结构体表示，它的 RelOptInfo-> reloptkind 是 RELOPT_BASEREL。基表之间可以进行连接操作，连接操作产生的"中间"结果也可以用 RelOptInfo 结构体来表示，它对应的 RelOptInfo-> reloptkind 是 RELOPT_JOINREL。另外还有 RELOPT_OTHER_MEMBER_REL 类型的 RelOptInfo 用来表示继承表的子表或者 UNION 操作的子查询等。

等价类

EquivalenceClass

```
typedef struct EquivalenceClass
{
    NodeTag     type;

    List       *ec_opfamilies;  /* btree operator family OIDs */
    Oid         ec_collation;   /* collation, if datatypes are collatable */
    List       *ec_members;     /* list of EquivalenceMembers */
    List       *ec_sources;     /* list of generating RestrictInfos */
    List       *ec_derives;     /* list of derived RestrictInfos */
    Relids      ec_relids;      /* all relids appearing in ec_members, except
                                 * for child members (see below) */
    bool        ec_has_const;   /* any pseudoconstants in ec_members? */
    bool        ec_has_volatile;    /* the (sole) member is a volatile expr */
    bool        ec_below_outer_join;    /* equivalence applies below an OJ */
    bool        ec_broken;      /* failed to generate needed clauses? */
    Index       ec_sortref;     /* originating sortclause label, or 0 */
    Index       ec_min_security;    /* minimum security_level in ec_sources */
    Index       ec_max_security;    /* maximum security_level in ec_sources */
    struct EquivalenceClass *ec_merged; /* set if merged into another EC */
} EquivalenceClass;
```

等价类又是由等价类成员结构体构成的，等价类成员结构体的定义如下：

```
typedef struct EquivalenceMember
{
    NodeTag     type;

    Expr       *em_expr;        /* the expression represented */
    Relids      em_relids;      /* all relids appearing in em_expr */
    Relids      em_nullable_relids; /* nullable by lower outer joins */
    bool        em_is_const;    /* expression is pseudoconstant? */
    bool        em_is_child;    /* derived version for a child relation? */
    Oid         em_datatype;    /* the "nominal type" used by the opfamily */
} EquivalenceMember;
```

例如对于 SQL 语句 SELECT * FROM STUDENT LEFT JOIN (SCORE INNER JOIN COURSE ON SCORE.cno = COURSE.cno AND COURSE.cno = 5) ON STUDENT.sno = SCORE.SNO WHERE STUDENT.sno = 1，最终会产生 3 个等价类，这 3 个等价类会保存在 PlannerInfo-> eq_classes 中

谓词下推

RelOptInfo 结构体生成之后是保存在数组中的，也就是说这些基表已经从树状结构（Query-> jointree）被拉平成线性结构（PlannerInfo-> simple_rel_array），随之而来的问题是基表之间的逻辑关系无法在 PlannerInfo-> simple_rel_array 数组上得到体现，例如在查询树中记录了有哪些基表做了什么类型的连接（记录在 FromExpr、JoinExpr 中），各个基表之间有哪些约束条件（包括连接条件和过滤条件），这些使用树状结构比较容易表现，而对于数组这种"扁平化"的结构则不容易表现，因此，逻辑分解优化要做的工作一方面是将约束条件下推到 RelOptInfo，另一方面就是要将基表之间的连接类型记录起来。

连接条件的下推 我们将约束条件细分成了连接条件和过滤条件，它们的作用略有不同，连接条件强调的是连接操作的计算过程，由于外连接会对没有连接上的元组补 NULL 值，所以对于 A=B 这样的连接条件产生出来的连接结果不一定满足 A=B，有可能产生出来的是 A 和 NULL。而过滤条件强调的是对查询结果的过滤作用，主要是对信息的筛选。

对连接条件而言，如果只有内连接，连接条件下推的过程就变得简单了，连接条件可以直接下推到自己所涉及的基表上。

结论 1：连接条件下推之后会变成过滤条件，过滤条件下推之后仍然是过滤条件。

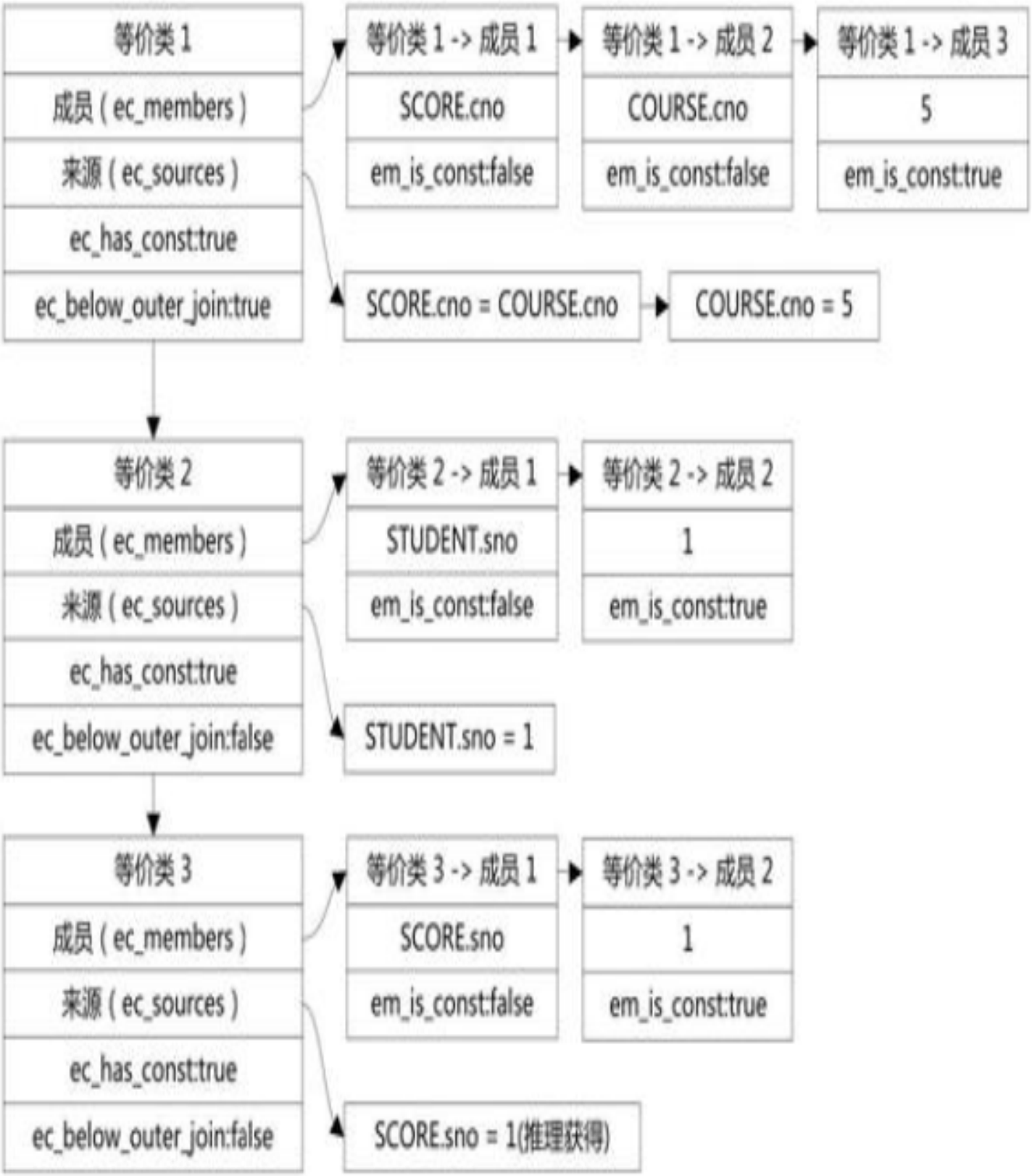
等价类 1
成员 (ec_members)
来源 (ec_sources)
ec_has_const:true
ec_below_outer_join:true
等价类 1 -> 成员 1
SCORE.cno
em_is_const:false
等价类 1 -> 成员 2
COURSE.cno
em_is_const:false
等价类 1 -> 成员 3
5
em_is_const:true
SCORE.cno = COURSE.cno
COURSE.cno = 5
等价类 2
成员 (ec_members)
来源 (ec_sources)
ec_has_const:true
ec_below_outer_join:false
等价类 2 -> 成员 1
STUDENT.sno
em_is_const:false
等价类 2 -> 成员 2
1
em_is_const:true
STUDENT.sno = 1
等价类 3
成员 (ec_members)
来源 (ec_sources)
ec_has_const:true
ec_below_outer_join:false
等价类 3 -> 成员 1
SCORE.sno
em_is_const:false
等价类 3 -> 成员 2
1
em_is_const:true
SCORE.sno = 1(推理获得)

Figure 1: 如图

但是在引入了外连接和（反）半连接之后，情况就变得复杂起来，不同的连接类型给连接操作符两端的表赋予了不同的属性：Nonnullable-side 和 Nullable-side。通常来说在外连接中 Nonnullable-side 的表没匹配上连接条件的元组也会显示出来，并且在表 Nullable-side 补 NULL 值。

由于外连接的 Nullable-side 一端可能要补 NULL 值，连接条件的下推就会受到阻碍，如果连接条件引用了 Nullable-side 的表，连接条件是可以下推变成过滤条件的，如果连接条件引用了 Nonnullable-side 的表，那么这个连接条件无法下推，仍然是一个连接条件。

结论 2：如果连接条件引用了 Nonnullable-side 的表，那么连接条件不能下推，如果连接条件只引用了 Nullable-side 的表，那么连接条件可以下推。

过滤条件的下推 如果过滤条件只引用了 Nonnullable-side 的表，那么这个过滤条件能够下推到 Nonnullable-side 的表上，如果过滤条件引用了 Nullable-side 的表且过滤条件是严格的，那么会导致外连接消除变成内连接。在内连接的情况下，过滤条件显然也能下推到对应的表上。

结论 3：如果过滤条件只引用了 Nonnullable-side 的表，那么这个过滤条件能够下推到表上，如果过滤条件引用了 Nullable-side 的表且过滤条件是严格的，那么会导致外连接消除，外连接消除之后变成内连接，过滤条件也是能下推的。

Lateral 语法

在 query_planner 函数中分别调用了 find_lateral_references 函数和 create_lateral_join_info 函数两个函数来处理 Lateral 语法中涉及的变量（Var 或者 PlaceHolderVar），在介绍这两个函数之前，先对 Lateral 语法做一个介绍。

Lateral 语义分析

PostgreSQL 数据库在 Lateral 实现之前是基于这样一个假设：所有的子查询（不包括子连接）都独立存在，不能互相引用属性。

消除无用连接项

什么样的连接项才能被消除掉呢?

·必须是左连接，且内表是基表（注意这时候已经没有右连接了，在消除外连接的时候把所有的右连接已经转换成了左连接）。

·除了当前连接中，其他位置不能出现内表的任何列。

·连接条件中内表的列具有唯一性。

Semi Join 消除

Semi Join 的本质是对于外表也就是 LHS 的每一条元组，如果在内表也就是 RHS 的表中找到一条符合连接条件的元组，就表示连接成功，即使内表中有多个符合连接条件的元组，也只匹配一条。

那么如果内表能保证唯一性，Semi Join 的连接结果就可以和 Inner Join 相同了，因此在这种情况下，可以将 Semi Join 消除，转换为内连接。例如对于具有 DISTINCT 属性的内表，就可以将 Semi Join 转换为 Inner Join。

扫描路径

代价

基于页面的 IO 基准代价

PostgreSQL 数据库采用顺序读写一个页面的 IO 代价作为单位 1，用 DEFAULT_SEQ_PAGE_COST 来表示。

需要注意的是，“顺序 IO”和“随机 IO”是相对应的，从基准代价的定义可以看到这二者之间相差 4 倍：

1. 目前的存储介质大部分仍然是机械硬盘，机械硬盘的磁头在获得数据库的时候需要付出寻道时间，如果要读写的是一串在磁盘上连续的数据，就可以节省寻道时间，提高 IO 性能，而如果随机读写磁盘上任意扇区的数据，那么会有大量的时间浪费在寻道上。

2. 大部分磁盘本身带有缓存，这就形成了主存 → 磁盘缓存 → 磁盘的三级结构，在将磁盘的内容加载到内存的时候，考虑到磁盘的 IO 性能，磁盘会进行数据的预读，把预读到的数据保存在磁盘的缓存中，也就是说如果用户只打算从磁盘读取 100 字节的数据，那么磁盘可能会连续读取磁盘中的 512 字节（不同的磁盘预读的数量可能不同），并将其保存到磁盘缓存，如果下一次是顺序读取 100 字节之后的内容，那么预读的 512 字节的数据就会发挥作用，性能会大大地增加。而如果读取的内容超出了 512 字节的范围，那么预读的数据就没有发挥作用，磁盘的 IO 性能就会下降。

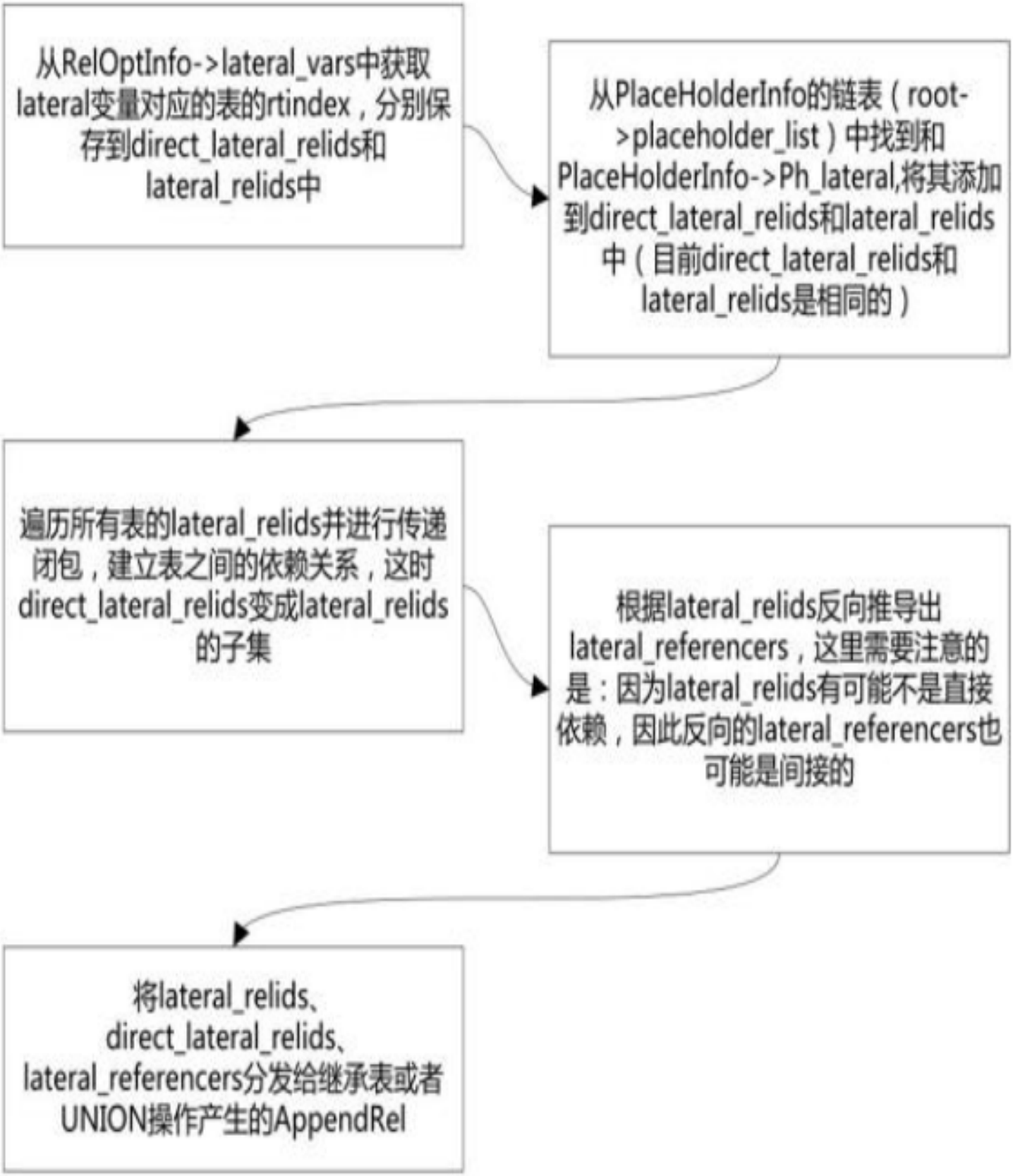
从RelOptInfo->lateral_vars中获取lateral变量对应的表的rtindex，分别保存到direct_lateral_relids和lateral_relids中
从PlaceHolderInfo的链表（root->placeholder_list）中找到和PlaceHolderInfo->Ph_lateral,将其添加到direct_lateral_relids和lateral_relids中（目前direct_lateral_relids和lateral_relids是相同的）
遍历所有表的lateral_relids并进行传递闭包，建立表之间的依赖关系，这时direct_lateral_relids变成lateral_relids的子集
根据lateral_relids反向推导出lateral_referencers，这里需要注意的是：因为lateral_relids有可能不是直接依赖，因此反向的lateral_referencers也可能是间接的
将lateral_relids、direct_lateral_relids、lateral_referencers分发给继承表或者UNION操作产生的AppendRel

Figure 2: 流程

基于元组的 CPU 基准代价

读取页面通常并不是查询的终极目标，查询的终极目标是将元组以要求的形式展示出来，因此就产生了从页面读取元组以及对元组处理的代价，这部分代价不同于读取页面的 IO 代价，这时页面已经在主存中了，从主存中的页面获取元组不会产生磁盘 IO，它的代价主要产生在 CPU 的计算上。

基于表达式的 CPU 基准代价

在执行计划的过程中，不止处理元组需要消耗 CPU 资源，在投影、约束条件中包含大量的表达式，对这些表达式求值同样需要消耗 CPU 资源，因此 PostgreSQL 数据库把表达式的求值代价单独剥离出来。

并行查询产生的基准代价

目前，PostgreSQL 数据库部分支持了并行查询，因此通常在分布式数据库系统中才考虑的通信代价目前 PostgreSQL 数据库也需要考虑了，因为 Gather 进程和 Worker 进程在并行查询的过程中需要进行通信，因此需要考虑进程间通信（IPC）所需的初始化代价（DEFAULT_PARALLEL_SETUP_COST），以及 Worker 进程向 Gather 进程投递元组的代价（DEFAULT_PARALLEL_TUPLE_COST）。

缓存对代价的影响

数据库本身有缓存系统，磁盘上也有磁盘缓存，当读取一个缓存中的数据页面时是不会产生磁盘 IO 的，因此，如果对每个页面都计算磁盘 IO 的代价，代价的计算结果就会失真，所以我们还需要对缓存中的页面数量有一个估计，目前 PostgreSQL 数据库用 effective_cache_size 参数来表示，实际上这个值一定是不准确的，这是 PostgreSQL 数据库需要改进的地方。

路径

物理优化对查询树的改造与逻辑优化不同，逻辑优化主要是基于 SQL 语句中指定的逻辑运算符进行等价的变换，而物理优化则在逻辑优化的基础之上建立一棵路径树，路径树中的每一个路径都是一个物理算子，这些物理算子是为查询执行器准备的，查询执行器通过运行这些物理算子来实现查询中的逻辑运算，这些物理算子又大体可以分为两类，它们是扫描路径和连接路径。

Path 结构体

```
typedef struct Path
{
    NodeTag     type;

    NodeTag     pathtype;       /* tag identifying scan/join method */

    RelOptInfo *parent;         /* the relation this path can build */
    PathTarget *pathtarget;     /* list of Vars/Exprs, cost, width */

    ParamPathInfo *param_info;  /* parameterization info, or NULL if none */

    bool        parallel_aware; /* engage parallel-aware logic? */
    bool        parallel_safe;  /* OK to use as part of parallel plan? */
    int         parallel_workers;   /* desired # of workers; 0 = not parallel */

    /* estimated size/costs for path (see costsize.c for more info) */
    double      rows;           /* estimated number of result tuples */
    Cost        startup_cost;   /* cost expended before fetching any tuples */
    Cost        total_cost;     /* total cost (assuming all tuples fetched) */

    List       *pathkeys;       /* sort ordering of path's output */
    /* pathkeys is a List of PathKey nodes; see above */
} Path;
```

路径使用 Path 结构体来表示，Path 结构体“派生”自 Node 结构体，Path 结构体同时也是一个“基”结构体，类似于 C++ 中的基类，每个具体路径都从 Path 结构体中“派生”，例如索引扫描路径使用的 IndexPath 结构体就是从 Path 结构体中“派生”的。

参数化路径

在谓词下推的过程中，对于…FROM A JOIN B ON A.a = B.b 这种类型的约束条件肯定是不能下推的，因为 A.a = B.b 这样的约束条件既引用了 LHS 表的列属性，又引用了 RHS 表的列属性，必须在获得两个表的元组之后才能应用这样的约束条件。现在假如在 B.b 属性上有一个索引 B_b_index，但如果没有能够匹配索引的约束条件，索引扫描路径或者不会被采纳，或者采纳为对整个索引进行扫描（例如 Fast Index Scan，这时候将索引视为一个表，实际上和 SeqScan 类似），发挥不了索引对数据筛选的作用。

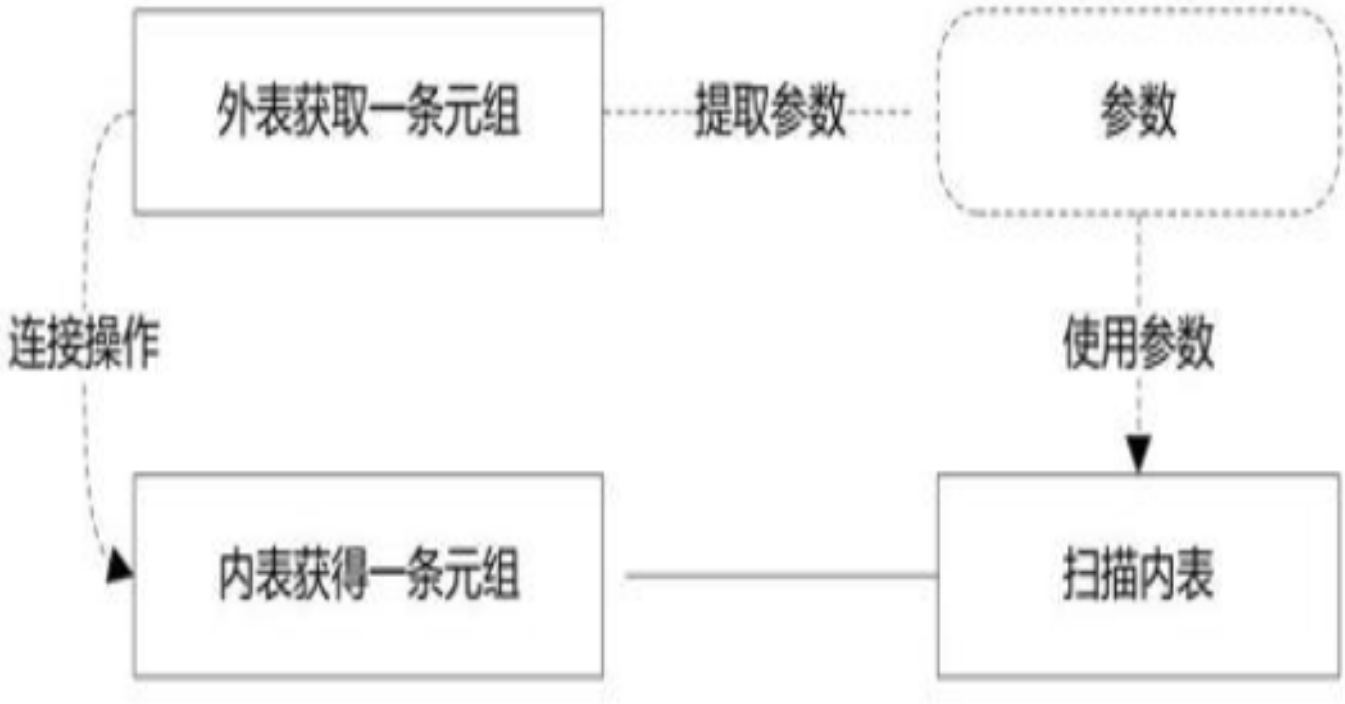

Figure 3: 参数化路径说明

PostgreSQL 数据库已经定义了 Param 结构体，它就是负责传递参数的，这时候我们是否能利用这个结构体为我们传递参数呢？通常，Param 用来从执行计划外向执行计划内传递参数，比如 PBE（Prepare、Bind、Execute）中使用了 Param，它是从执行计划外向执行计划内 Bind 参数，再比如父执行计划和子执行计划之间通过 Param 进行"通信"，可以是从父执行计划传递给子执行计划，也可以从子执行计划传递给父执行计划，这种情况参数值也来自于执行计划外，而目前要用的参数在执行计划内传递，从一个路径向另一个路径传递参数值，或者说从外表向内表传递参数，和上面使用 Param 的方式有些区别，因此 PostgreSQL 数据库在这里定义了 NestLoopParam 结构体来封装 Param，保证查询执行器在执行 Nested Loop Join 执行算子的时候，能够根据 NestLoopParam 找到对应的 Param 进行参数传递，以实现这种执行计划内传递参数的功能。

make_one_rel 函数

make_one_rel 函数就是"生成一个关系"，实际上就是创建一个 RelOptInfo 结构体，这个结构体可以用来表示基表，也可以用来表示连接操作，make_one_rel 的过程就是向 RelOptInfo 中增加物理路径的过程。目前还不考虑上层的 Non-SPJ 操作，例如 GROUP BY、ORDER BY 等（也不是完全不考虑，例如启动代价就是为上层的 LIMIT 操作设计的，但主要考虑的是基表之间的连接关系）。

RelOptInfo 通过层层叠加最终形成一个"路径树"，这个路径树的叶子节点一定是表示基表的 RelOptInfo，它里面保存的是扫描路径，上层节点是表示连接操作的 RelOptInfo（还可能有物化、排序等节点），它里面保存的是连接路径。make_one_rel 通过动态规划的方法或者遗传算法对扫描路径进行规划，并且根据 SpecialJoinInfo 中记录的逻辑连接关系将基表组合到一起，最终形成最上层的"一个 RelOptInfo"。

make_one_rel 函数主要分成两个阶段：生成扫描路径的阶段和生成连接路径的阶段

普通表的扫描路径

在所有基表中最重要的、也是最常见的就是普通的堆表，它的扫描路径通过 set_plain_rel_pathlist 函数来设置，该函数对堆表尝试生成顺序扫描路径、并行顺序扫描路径、索引扫描路径和 TID 扫描路径，这些扫描路径需要通过 add_path 函数加入基表的 RelOptInfo-> partial_pathlist

扫描顺序

顺序扫描（SeqScan）又称作全表扫描，是最基本的扫描方式，它的算法复杂度是 O(N)，其中 N 代表的是整个表中元组的数量，也就是说一个表中所有的元组都会被读取出来，读取出来的元组交付给约束条件进行过滤，把符合约束条件的元组交给投影函数进行投影，最终以查询语句指定的方式输出。

顺序扫描路径的代价主要计算了 3 部分：启动代价(startup_cost)、CPU 代价(cpu_run_cost)、磁盘的 IO 代价(disk_run_cost)。

索引扫描

PostgreSQL 数据库的表是堆组织表，数据在堆页面没有顺序排列的要求，向堆组织表插入一条数据时，可以将其存放在新开辟的页面中，也可以存放在之前删除元组时空出来的空白空间上，写入的位置没有特定的要求。这种形式带来的好处是堆组织表的结构比较简单，带来的问题是无法使用一些常用的查找算法来查询数据，通常只能采用顺序扫描（SeqScan）的方法，也就是对整个堆进行扫描，即使 SQL 语句中包含像 a=1 这样的约束条件，顺序扫描也无法应用任何优化措施。

为了提高数据库的查询性能，PostgreSQL 数据库提供了多种索引类型提供给各种不同的查询需求。例如 B 树索引，它是一种基于磁盘的树状数据组织结构，能够将像 a=1 这样的约束条件数据查找的复杂度减低成为 O(Log(N))。再例如 Hash 索引，我们知道 Hash 方法经常会用于数据查找，类似 a=1 这样的约束条件进行 Hash 查找的复杂度近似为 O(1)，因此索引是效率提升的有力武器。

如果查询中的约束条件的选择率比较大，那么查询优化器会倾向于选择顺序扫描；如果约束条件的选择率非常小，那么查询优化器会倾向于选择索引扫描；如果约束条件的选择率介于二者之间（中等），那么很可能就会选择位图扫描。

索引扫描路径用 indexPath 结构体来表示，它在 Path"基类"的基础上又增加了一些信息，这些信息主要与索引匹配的约束条件相关。需要注意的是，在 Path 基类中已经保存了 total_cost，而在索引路径结构体上又定义了一个 indextotalcost 变量，这是因为在索引路径的生成过程中，一个索引路径既可能直接被用于索引扫描，还可能会用于位图扫描（位图扫描中也会对索引进行扫描，只不过它是将索引扫描出来的结果保存成为图，避免随机读的问题，BitmapHeapPath 结构体会复用 IndexPath 结构体），这两种扫描的代价计算使用了不同的模型，为了不混淆这两种扫描的代价，使用 IndexPath-> indextotalcost 来保存位图扫描的代价，而用 Path-> tatal_cost 来保存普通索引扫描和快速索引扫描的代价，IndexPath-> indexselectivity 同理。

位图扫描

位图扫描是针对索引扫描的一个优化，它通过建立位图的方式将原来的随机堆表访问转换成了顺序堆表访问，索引必须支持建立位图（IndexAmRoutine-> amgetbitmap）才能用于建立位图扫描路径。

在索引扫描路径创建的过程中，一直同时在为位图扫描路径生成待选路径（保存在 bitindexpaths 中），但是只有这些待选路径还不够，因为索引和约束条件匹配的要求比较严格，只支持 OpExpr、ScalarArrayOpExpr、NullTest、BooleanTest、RowCompareExpr 等几种简单的表达式，除了这些表达式，位图扫描还能支持一些更为复杂的情况。例如约束条件中的 OR 相关的子约束条件，对于 A > 1 OR A > 2 这样的约束语句，即使 A 是索引的键，在创建索引扫描路径时也无法和索引匹配成功，但是位图扫描不同，位图扫描虽然也借助了索引，但它通过索引来生成位图，而位图本身具有交集和并集的操作，因此对于 A > 1 OR A > 2 这样的约束条件可以通过位图的并集来实现

动态规划与遗传算法

动态规划

如果使用动态规划方法来生成执行路径，那么每个路径节点都应该产生一个最优子问题，例如在给 TEST_A 表建立扫描路径的时候，会建立顺序扫描路径和索引扫描路径，但是在这两个路径中哪个是子问题的最优解呢？在介绍索引扫描路径的时候我们有一个感性的认识：如果约束条件的选择率高，通常会选择顺序扫描路径，如果约束条件的选择率低，通常会选择索引扫描路径，然后我们做 3 个假设。

假设 1：约束条件的选择率高，TEST_A 的扫描路径中顺序扫描是最优解。

假设 2：TEST_A 和 TEST_B 的连接操作使用 MergeJoin 方法，如果 TEST_A 采用顺序扫描路径，则还需要显式地对其结果进行排序。

假设 3：TEST_A 的索引扫描路径产生的结果恰好符合 MergeJoin 的 Sort 要求。那么这时候就可能出现如下情况：

顺序扫描代价＜索引扫描代价

顺序扫描代价 + 排序代价＞索引扫描代价

这也就导致了对 TEST_A 表的扫描的最优子问题无法产生整个查询计划的最优解的情况。

动态规划方法首先考虑两个表的连接，其中优先考虑有连接关系（SpecialJoinInfo）的表进行连接，两个表的连接可以建立一个新的 RelOptInfo 标识，生成的连接路径也记录到这个 RelOptInfo 中，这时的 RelOptInfo 是 RELOPT_JOINREL 类型的，表示它是连接操作产生的，它里面存储的也是连接路径

遗传算法

在 PostgreSQL 数据库中，遗传算法是动态规划方法的有益补充，只有在 enable_geqo 打开并且待连接的 RelOptInfo 的数量超过 geqo_threshold（默认为 12 个）的情况下，才会使用遗传算法。

PostgreSQL 数据库通过 geqo_eval 函数来生成计算适应度，它首先根据染色体的基因编码生成一棵连接树，然后计算这棵连接树的代价。

遗传算法使用 gimme_tree 函数来生成连接树，而 gimme_tree 则递归调用了 merge_clump，merge_clump 的作用是将能够进行连接的表尽量连接并且生成连接子树，并记录每个连接子树中节点的个数，然后将连接子树记录到 clumps 链表，而且按照节点个数从高到低的顺序记录到 clumps 链表。