目录

图表目录

PostgreSQL 源码分析报告

——查询处理部分

摘　要: PostgreSQL 是一个优秀的开放源码数据库管理软件，阅读并分析其实现代码，对于学习《数据库系统实现》课程和提高自己实践能力都有很大的帮助。本文着力于分析 PostgreSQL 的查询处理部分的流程与实现，为在 PostgreSQL 基础上实现查询优化的扩展功能奠定基础。

关键词:: PostgreSQL; 查询处理; 查询优化; 数据库系统实现; 课程实习

第一章 引言

1.1　课程实习的目的与意义

PostgreSQL 是一个优秀的开放源码数据库软件。其良好的性能、优秀的代码及风格都值得我们学习研究。对其进行代码分析和修改，同时结合《数据库实现》课程，对于提高我们把理论应用于实践，在实践中把握理论的能力有很大的帮助。

1.2　本报告组织形式及本组任务分配

本文主要分为两个大的部分，查询前台处理与后台执行。其中查询的后台执行是主要部分，它又分为四个大的子部分：Parser,Rewriter,Planner,Executor。每部分都有体系结构、流程与具体实现、重要的数据结构的分析。本文共为五章：第一章是对课程实习和本文做了个概述性说明；第二章对 PostgreSQL 做了一个概要叙述、源码文件结构说明和查询体系结构的总体流程概述。第三章是查询前台的分析。第四章为查询后台执行的详细分析。第五章是结合实习的主题——查询处理的优化，来重点分析了几个基本的查询运行。主体是第四章，在第四章中，第一节主要集中分析各组件的流程与架构，第二节分析各组件的具体实现与重要的数据结构；

本小组组长谢少峰。组员为：谢少峰，欧阳锦林。任务分配情况为：前台部分、后台的词法、语法分析、重写与计划树的生成部分的源码分析主要由谢少峰完成。后台的总体结构、计划树的组织与执行部分的源码分析和文档的组织由欧阳锦林完成。

第二章　PostgreSQL 简介

2.1　PostgreSQL概要及特点

PostgreSQL 是一个典型的开放源码的对象-关系型数据库管理系统，它的前身是加利福尼亚大学伯克利分校的数据库管理系统 Postgres。通过十几年的发展，PostgreSQL 已经成为世界上可以获得的最先进的开放源码的数据库系统。目前它已被成功的应用在包括医疗、电子商务、财务数据分析、小行星数据跟踪和地理信息等在内的广阔领域中。PostgreSQL 符合 SQL92／99 标准，具有以下一些鲜明特色：

1、支持复杂对象数据类型许多领域的数据

比如：工程、地理等，比起通常的商业数据，更加复杂、多变。它们通常都是多维数据的集合体，如果用通常的 RDBMS 来管理，则每维的数据都必须有一个表来模拟；这样一来，为了查询一个对象的有关信息，必然涉及到访问若干个表从而影响到效率。一个直接的解决办法是将描述一类对象所有维的数据构造成一个对象类型，用这个对象作为属性的数据类型存储在表中，这就是数据库管理系统的对象支持技术。

2、支持用户自定义数据类型、操作符以及访问方法

一般来说，这几个方面是紧密相连的：支持用户自定义数据类型要求提供相应操作符以便运算，同时也要求提供相应类型的访问方法。这里有几个难点：(1)如何做到用户定义的数据类型与原有的操作符方法运算原则相一致?(2)如何做到用户加入的操作符方法在满足自定义的数据类型的同时也能运算现有的数据类型且不冲突?PostgreSQL 的实现做到用户增加数据类型时，不需要有专业的知识和技能。

3、支持“活跃”数据库和规则

一些与数据的性质联系强的应用使用触发器和报警器就可以满足其要求。然而，一些专家系统的知识比较适合用规则来描述，它的工作完全依赖于规则库的规则，这些规则有以下一些特征：规则之间可能彼此冲突，导入一个规则可能导致规则库的缩小而不是增加，因为有些冲突的规则必须被删除。这单靠触发器和报警器是解决不了问题的。所以 Postgres 设计时就考虑到要支持规则库和“活跃数据库”。

4、减少系统崩溃后的恢复代码和代价

PostgresSQL 使用了“多版本”控制技术，即删除元组的操作并不实际删除它而只是作无效标志，更新元组的操作取代为先删除后插入；数据库管理员可以用命令 VACUUM 定期地清理数据库中无效元组来紧缩存储空间。

PostgreSQL 还拥有其他一些现代数据库特征，甚至某些特色是其他商业系统所没有的。

2.2　PostgreSQL部分源文件介绍

对于 PostgreSQL 我们分析过的较为重要的文件给出一个列表：

pgsql/src/bin/psql/

startup.c　建立交互平台，初始化数据结构，建立和后台的连接

mainloop.c　主循环，把从命令行读出的命令送到后台执行

tab-complete.c　代码自动完成部份，interactive terminal 把已输入的 sql 查询语句发送到后台进行智能补全操作。限定的返回行数是 1000，可以补全的内容有编目的名字，将被查询的编目，或者符合选择的对象。

fe-exec.c 传递查询数据到后台的相关函数。

src/backend/postmaster/

postmaster.c 后台的主启动文件。

syslogger.c　这是在 Postgres 8.0 以后增加的，系统日志程序

src/backend/tcop/

postgres.c　PostgresSQL backend 的主模块，也是 Traffic cop 的主模块。

src/include/nodes/

nodes.h Node 结点所有子类的定义

execnodes.h 执行结点的定义

parsenodes.h 树各个结点的定义，各个语句的定义

primnodes.h 各原语结点的定义

pg_list.h PostgreSQL 通用链表包的接口表

src/backend/executor/

execAmi.c：一些不便归类的函数

execGrouping.c：用于 grouping, hashing, aggregation 的工具类函数

execJunk.c：与 Junk 相关的函数

execMain.c：执行器提供给其他模块的接口及相关函数（Executor 层和 Plan 层）

execProcnode.c：执行器内部执行函数（Node 层入口）

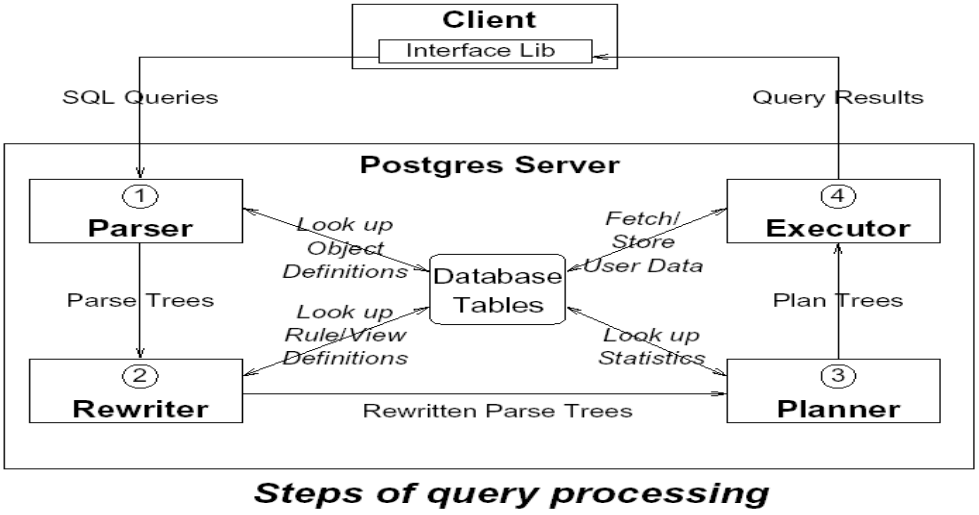
execQual.c：计算表达式的值和进行条件检验的函数

execScan.c：扫描一个关系、及其相关的函数

execTuples.c：执行与执行器内使用的 Tuple 和 TupleTable 相关的函数 execUtils.c：执行器用到的各种工具类辅助函数

2.3 PostgreSQL查询体系结构

下图是一个简短的描述，描述一个查询从开始到得到结果要经过的阶段。



图表 1:PostgreSQL 查询处理流程

从上图可以看出：PostgreSQL 从用户输入查询到最后结果的输出可以分为两个大的部分：前台处理和后台执行。其中后台执行又有四个部分：Parser,Rewriter,Planner,Executor。下面就以这五个部分为序。去分析各个部分的体系结构、流程与实现。

第三章　查询前台处理

我们对查询前台的分析是从 PSQL 命令行（the PostgreSQL interactive terminal）发出查询命令开始，GUI 交互前台的分析也是类似的。首先从 startup.c 的 main()启动，根据用户的指令建立与相应后台的连接，初始化基本数据结构，在用户输入正确的用户名和密码之后，显示欢迎信息。

由 PQExpBuffer 的设计可以知道其是一种可扩充的 buffer，为了提高性能，并不要求装满。最初的大小是 256，通过 createPQExpBuffer()或 void initPQExpBuffer(PQExpBuffer str)两个函数来分配空间。实质上它和 backend 中的 StringInfo 数据类型是相同的，但是它是为前台服务的。通过 enlargePQExpBuffer(PQExpBuffer str, size_t needed)函数为 PQExpBuffer 分配更大的空间。该函数当发现空间内部不够大的时候，给当前的 buffer 多分配原来大小一倍的空间，然后用 realloc()函数移动原来的数据，并释放原来占有的空间。

```
typedef struct PQExpBufferData
{
    char    *data;
    size_t    len;
    size_t    maxlen;
} PQExpBufferData;
typedef PQExpBufferData *PQExpBuffer;
```

图表 2：PQExpBufferData 结构体

最后跳入 mainloop.c 中的 MainLoop 的函数，MainLoop 维护 PQExpBuffer 类型的 query_buf，previous_buf，history_buf 三个 buffer。其中 history_buf 保存的是以前的历史操作。previous_buf 保存的当前操作，由于 psql 中每个命令可以有多行（通过"\"+"Enter"进行分割），所以 previous_buf 会一行一行的添加进 char* line 中的输入，当一个命令发射时，再把 previous_buf 中的数据送到 query_buf 中去。

MainLoop 通过 psql_scan（）函数返回的状态集 PsqlSettings pset 各个属性值来控制交互平台的流程。下表是 PsqlSettings 结构体，用来放前台与词法器生成程序交互的信息。

```
typedef struct _psqlSettings
{
    PGconn      *db;                //指向后台结构联系的指针
    int           encoding;      //客户端的编码
    FILE    *queryFout;          //结果送去的地方。
    bool      queryFoutPipe;     //queryFont来自于popen() 命令吗?
    printQueryOpt popt;
    char    *gfname;             /* one-shot file output argument for \g */
    bool      notty;             /* stdin or stdout is not a tty (as determined
                                     * on startup) */
    bool      getPassword;  /* prompt the user for a username and password */
    FILE    *cur_cmd_source; /* describe the status of the current main
                                     * loop */
    bool      cur_cmd_interactive;
    int           sversion;      //后台服务器版本
    const char *progname;        /* in case you renamed psql */
```

```
    char    *inputfile;         /* for error reporting */
    char    *dirname;       /* current directory for \s display */
    uint64        lineno;           /* also for error reporting */
    bool      timing;           /* enable timing of all queries */
    FILE    *logfile;       //log文件的名称
    VariableSpace vars;             /* "shell variable" repository */
}
```

图表 3：psqlSettings 结构体

对于 PsqlSettings 结构体 PGconn *db，下面我们给出更详细的分析。从下面经过节选的 pg_conn 的结构体我们可以看清楚 PSQL 与后台所需要交换的数据。另外结构体里面还存放了 sql 指令运行后返回的数据。

```
struct pg_conn  {
    char    *pghost;            //服务器主机的名字
    char    *pghostaddr;        //服务器主机的IP v4地址
    char    *pgport;        //与服务器通讯的端口
    char    *connect_timeout;   //超时时间
    char    *pgoptions;         //启动后台的选项
    char    *dbName;            //数据库名字
    char    *pguser;            //后台用户名
    char    *pgpass;           //后台密码
    //状态标识
    ConnStatusType status;   //连接状态
    PGAsyncStatusType asyncStatus;
    PGQueryClass queryclass;
    char    *last_query;        上一条执行的sql语句
        bool      options_valid;     /* true if OK to attempt connection */
    bool      nonblocking;  //这个连接是否使用非阻塞方式传送信息/
    char      copy_is_binary; /* 1 = copy binary, 0 = copy text */
    int           copy_already_done;
    //消息队列的管理
    PGnotify   *notifyHead;     /* oldest unreported Notify msg */
    PGnotify   *notifyTail;     /* newest unreported Notify msg */

    //连接网络信息
    int           sock;             /* Unix FD for socket, -1 if not connected */
    SockAddr  laddr;            //本地地址
    SockAddr  raddr;            //远程地址
    ProtocolVersion pversion;       /* FE/BE protocol version in use */
    int       sversion;      //服务器的版本
    //在建立连接时暂时的状态
    struct addrinfo *addrlist;  /* list of possible backend addresses */
    struct addrinfo *addr_cur;  /* the one currently being tried */
```

```
    int  addrlist_family;   /* needed to know how to free addrlist */
    PGSetenvStatusType setenv_state; /* for 2.0 protocol only */
    const PQEnvironmentOption *next_eo;
    //从后台返回的并且还没被处理的数据
    char    *inBuffer;      /* currently allocated buffer */
    int             inBufSize;          /* allocated size of buffer */
    int             inStart;        /* offset to first unconsumed data in buffer */
    int             inCursor;       /* next byte to tentatively consume */
    int             inEnd;              /* offset to first position after avail data */
    //维护没有送到后台服务器的信息
    char   *outBuffer;      /* currently allocated buffer */
    int  outBufSize;        /* allocated size of buffer */
    int  outCount;      /* number of chars waiting in buffer */
    PGresult   *result;             /* result being constructed */
    PGresAttValue *curTuple;    //当前读的元组
    PQExpBufferData errorMessage;         //错误信息缓存
    PQExpBufferData workBuffer;         //工作缓存
};
```

图表 4：pg_conn 结构体

执行一句 SQL 语句的流程：MainLoop 调用 SendQuery(const char *query)函数执行命令，该函数处理没有连接数据库、单步执行、log文件维、事务处理等具体细节，再调用 results = PQexec(pset.db, query)，其中 result 就是数据库后台返回的结果。在命令执行以后，使用 ProcessCopyResult(results)把运行结果显示在屏幕上。

下面简要介绍一下前台词法分析的过程，通过 void psql_scan_setup(PsqlScanState state,const char *line, int line_len) 函数启动对指定行的词法分析。然后调用 PsqlScanResult psql_scan(PsqlScanState state, PQExpBuffer query_buf,promptStatus_t *prompt) 返回的状态有下面几种，MainLoop 函数就是根据返回值控制 Buffer，当一个命令输入完毕以后发送到后台去执行。

状　　态	含　　义
PSCAN_SEMICOLON	以分号结束的命令
PSCAN_BACKSLASH	以反斜杆结束的命令
PSCAN_INCOMPLETE	到达行尾并没有完成的命令
PSCAN_EOL	遇到了 EOL 结束符

图表 5：PsqlScanResult 枚举状态种类

如果后台完成查询任务，会告诉前台它已经空闲了，这时前台可以发送新的查询命令。下面给出了 backend 返回给前台的数据结构，前台按照该结构显示结果。值得说一句的是，虽然对于 psql 交互平台显示出的结果可以完全看作是一串字符串,并不需要区分出表中结果每一个域。但 psql 和 backend 的通信协议是所有前台(包括基于 GUI 界面)和后台的通信协议。只不过 psql 显示时把它转换成字符串的表现形式。分析 psql 最重要是其与 backend 联系的方式和交换的数据结构。

```
struct pg_result
{
    int             ntups;
```

```
    int           numAttributes;
    PGresAttDesc *attDescs;
    PGresAttValue **tuples;
    int           tupArrSize;        //元组的占用空间
    int           numParameters;
    PGresParamDesc *paramDescs;
    ExecStatusType resultStatus;
    char      cmdStatus[CMDSTATUS_LEN];        //执行中返回的query状态/
    //以下信息复制自PGconn，使得在PGconn上操作不需要再引用PGconn了。
    PGNoticeHooks noticeHooks;
    int       client_encoding;   /* encoding id */
    //错误信息，如果没有错误的话，就全为NULL值，
    char    *errMsg;             /* error message, or NULL if no error */
    PGMessageField *errFields;  /* message broken into fields */
    //空间管理信息：
    PGresult_data *curBlock;    /* most recently allocated block */
    int  curOffset;         /* start offset of free space in block */
    int  spaceLeft;         /* number of free bytes remaining in block */
};
```

图表 6：pg_result 结构体

第四章　查询后台执行体系与流程

4.1　查询后台处理流程与体系结构

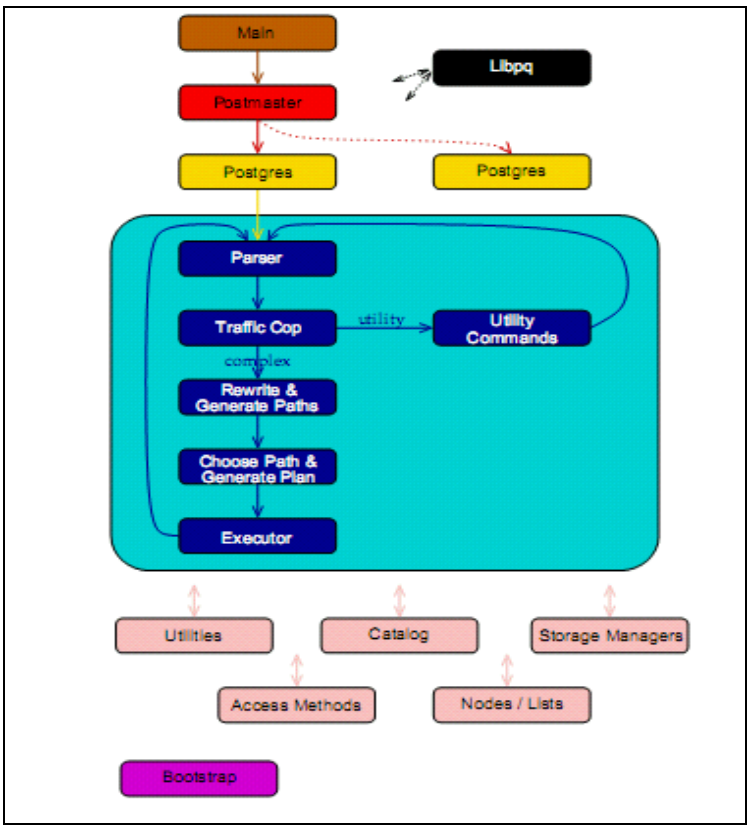
4.1.1 查询后台处理流程触发与总体体系

PostgreSQL 是采用的是“每用户对应一个进程”的 client/server 模型实现的。 在这种模式里一个客户端进程只是与一个服务器进程连接。 因为我们不知道具体要建立多少个连接， 所以我们利用一个叫做 postmaster 的主进程在每次联接请求时派生出一个新的服务器进程来。postmaster 监听着一个特定的 TCP/IP 端口等待进来的连接。 每当检测到一个连接请求时，postmaster 进程派生出一个新的叫 postgres 的服务器进程。 服务器任务（postgres 进程）相互之间使用信号灯和共享内存进行通讯， 以确保在并行的数据访问过程中的数据完整性。下面讲述的是它的具体工作流程：

前台程序发出一个启动命令后到 Postmaster 后，Postmaster 根据其提供的信息建立一个子进程，也就是后台进程，专门为前台服务。Postmaster 负责维护后台进程的生命周期，但与后台进程相独立。这样在后台进程后可以重启动后台进程而不会和后台进程起崩溃。

从 src/backend/main/main.c 开始执行，调用 int PostmasterMain(int argc, char *argv[])进入，首先在启动前根据命令行参数，设定相应的环境值。初始化监听端口，检查其维护的数据库文件是否存在，设置 signal handlers 从操作系统上监听其感兴趣的消息来维护 Backends。Postmaster 调用 StartupDataBase()函数来使自身进入工作状态。然后 ServerLoop()。

ServerLoop()是一个死循环，当有一个新的建立连接消息到来的时候，查找自身维护的端口列表，看是否有空闭的端口，如果有调用 static int BackendStartup(Port *port)来 fork 一个后台进程。然后，ServerLoop()判断下列几个后台支持进程的状态：system logger，autovacuum process，background writer process，the archiver 和 the stats collector。当发现一个或多个进程发现崩溃后，重新启动它们，确保数据库整体的正常运行。Postmaster 在循环中就这样一直不停的监听联系请求和维护后台支持进程。



图表 7：查询处理体系结构

对于在 BackendStartup(Port *port)来 fork 一个后台进程，Backend 的初始化由其中的 BackendInitialize(port)完成，主要包括读入 postgre 中的配置文件，根据配置文件进行端口的绑定和对客户进行验证，保证前台和数据库连接符合配置文件的要求。接着调用 BackendRun()，它会为 backend 设置好启动参数，以字符串方式传递给 PostgresMain(ac, av, port->user_name)，ac 为 int 型，代表参数个数；char **av 则是一个二维字符串数组；从这里开始 backend 正式开始工作。

PostgresMain 函数来自 pgsql/src/backend/tcop/postgres.c，首先处理 BackendRun()传递进来的参数。用字符串来传递配置这种方式，可以增加 PostgresMain 的灵活性，使其与其它模块以最小接口的方式传递数据，更方便与模块之间整合。然后按 GUC（Global User Configuration）初始化运行环境。

在 POSTGRES 的 main（）过程中，如果遇到异常的话，就会放弃当前的事务而开始一个新的事务。在初始化后就开始接受从前台发送过来的 SQL 查询命令。根据前台和后台发送信息的协议，第一个字母决定任务的种类，由 ReadCommand(&input_message)决定，下表描述了在 backend 主循环中所接受命令：

First Char	作用	执行函数/操作
Q	普通的查询	exec_simple_query()
P	对于 Statement 支持	exec_parse_message()
B	prepared statement 建立路口	exec_bind_message
E	处理 "Execute" message 入口	exec_execute_message()
F	fastpath function call(允许应用直接调用底层函数)	start_xact_command();
C	关闭当前正处理完的任务	根据 close_type 不同决定关闭时操作
D	Process a "Describe" message for a prepared statement	exec_describe_statement_message()
H	flush	输出所有缓存中的内容
S	synchronization	finish_xact_command();

图表 8：backend 主循环中所接受命令

对于 SQL 语句在 Backend 中的执行过程详见后面章节的分析。

4.1.2　Parser 流程

Postgre 命令的词法分析和语法分析是由 Unix 工具 yacc 和 lex 制作的。它们依赖的文件定义在 src\backend\parser 下的 scan.l 和 gram.y. 词法器在文件 scan.l 里定义，负责识别标识符，SQL 关键字等。 对于发现的每个关键字或者标识符都会生成一个记号并且传递给分析器。

分析器在文件 gram.y 里定义并且包含一套语法规则和触发规则时执行的动作。

核对语法并创建一棵查询树（由 ParseNode 构成）。在分析阶段如果发现语法错误，如输入的命令中有 SQL 中不存在的关键字，或者不符合已定义的语法规则。就会返回客户端错误信息并不再执行之后的流程。注意在这一阶段是没有事务保护处理。因为这是中间过程，并不会对数据本身产生不良影响。

在分析器完成之后，由 parse_analyze 函数进行进一步处理，又称为转换处理，该阶段接受分析器传过来的分析树然后做进一步处理，解析那些理解查询中引用了哪个表，哪个函数以及哪个操作符的语意。所生成的表示这个信息的数据结构叫做查询树。有关字段和表达式结果的具体数据类型也添加到查询树中。这一阶段可以解决语义错误。

4.1.3　Rewriter 流程

Postgre 命令的词法分析和语法分析是由 Unix 工具 yacc 和 lex 制作的。它们依赖的文件定义在

src\backend\parser 下的 scan.l 和 gram.y. 词法器在文件 scan.l 里定义，负责识别标识符，SQL 关键字等。 对于发现的每个关键字或者标识符都会生成一个记号并且传递给分析器。

分析器在文件 gram.y 里定义并且包含一套语法规则和触发规则时执行的动作。

核对语法并创建一棵查询树（由 ParseNode 构成）。在分析阶段如果发现语法错误，如输入的命令中有 SQL 中不存在的关键字，或者不符合已定义的语法规则。就会返回客户端错误信息并不再执行之后的流程。注意在这一阶段是没有事务保护处理。因为这是中间过程，并不会对数据本身产生不良影响。

在分析器完成之后，由 parse_analyze 函数进行进一步处理，又称为转换处理，该阶段接受分析器传过来的分析树然后做进一步处理，解析那些理解查询中引用了哪个表，哪个函数以及哪个操作符的语意。所生成的表示这个信息的数据结构叫做查询树。有关字段和表达式结果的具体数据类型也添加到查询树中。这一阶段可以解决语义错误。

4.1.4 Planner 流程

规划器/优化器（optimizer）的任务是创建一个优化了执行规划。 一个特定的 SQL 查询（因此也就是一个查询树）实际上可以以多种不同的方式执行， 每种都生成相同的结果集。如果可能，查询优化器将检查每个可能的执行规划，最终选择认为运行最快的执行计划，然后就制作一个完整的规划树传递给执行器。

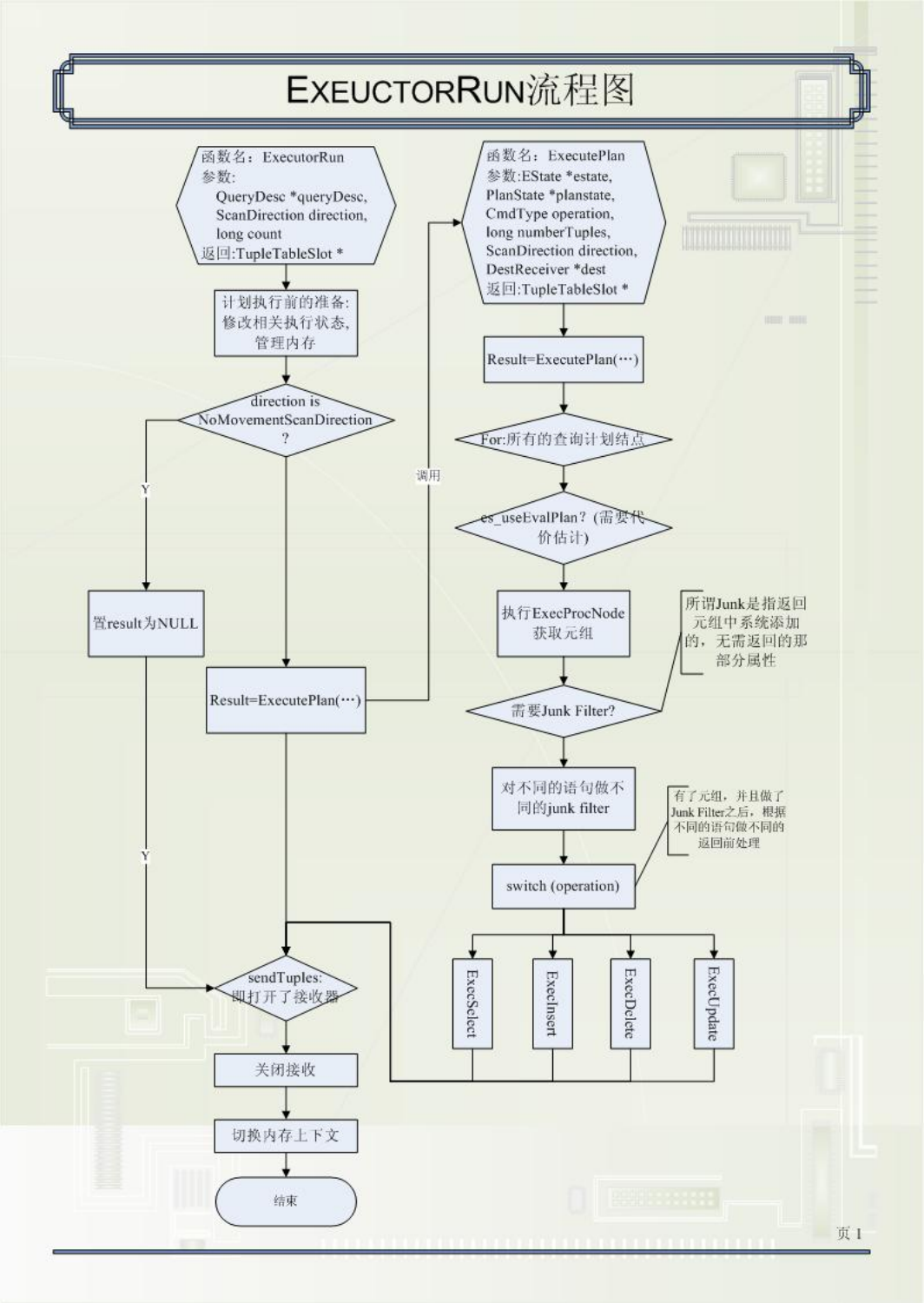
规划器的搜索过程实际上是与叫做 paths 的数据结构一起结合运转的， 规划器生成可能的规划。采用动态规划算法或者遗传算法优化（GEQO），根据查询树生成的所有可能的查询路径，然后评估每个路径的代价，从中选择一个查询代价最小的，即最终的查询规划树（由 Plan 结点构成），作为规划器的输出。优化由 pg_plan_queries 函数完成。

4.1.5 Executor 流程

执行器从根结点开始递归地以先左后右的后续遍历的方式扫描每个节点，直至所有结点都已经处理完毕，完成一个查询语句得到所有的查询的元组，最后由根结点返回给执行器，终止查询并返回最终的查询结果。

执行器接受规划器/优化器传过来地查询规划然后递归地处理它， 抽取所需要地行集合。它实际上是一个需求－拉动地流水线机制。 每次调用一个规划节点地时候，它都必须给出更多的一个行，或者汇报它已经完成行的传递了。

下图给出 PostgreSQL 执行查询的关键函数 ExeuctorRun 的详细流程图：



图表 9：ExeuctorRun 流程图

4.2　查询后台执行实现与数据结构

4.2.1　PostgreSQL 中类的继承与实现

PostgreSQL 用过程语言 C 实现的，但其中的设计思想却借用了很多面向对象的思想。主要体现在伪继承与虚拟函数等类的设计与实现上。

伪继承机制得以运作的原因：ISO 强制定义了 C 语言中结构体内存分配与排列模型，即结构体中各个

变量如何分配空间和在内存中顺序如何都已经被标准强制规定下来；C 语言中强制类型转换机制。不同的符合 ISO 标准的 C 语言编译器实现编译产生的二进制文件中结构体的内存分布一致，因此可以如下在 C 语言中实现继承：子结构 B 的前 n 个变量正好是父结构 A 的全部变量，n 等于父结构中变量个数，且 B 中前 n 个变量与 A 中排列顺序一致；所有继承结构内的结构体定义，第一个变量是一个可以标识结构体类型的变量。

下表便是所有结点类的基类 Node 类的定义：

```
typedef struct Node
{
    NodeTag        type;
} Node;
```

图表 10:所有结点类的基类：Node

```
//这个类是所有结点类的基类,里面含有所有结点
类的枚举.是所有类的第一个域。对本实习将涉及的
类添加一些简要注释
typedef enum NodeTag
{
    T_Invalid = 0,

    /*
    * 查询执行相关类
    */
    T_IndexInfo = 10,
    T_ExprContext,
    T_ProjectionInfo,
    T_JunkFilter,
    T_ResultRelInfo,
    T_EState,
    T_TupleTableSlot,

    /*
    * 查询计划相关类
    */
    T_Plan = 100,//plan是所有下面的类的基类
    T_Result,
    T_Scan,
    T_SeqScan,
    ……
    T_NestLoop,
    T_MergeJoin,
    T_HashJoin,
    T_Material,
    T_Sort,
    T_Group,
    ……
    /*
    * 表达式执行状态类
    */
    T_ExprState = 400,
    T_GenericExprState,
    T_AggrefExprState,
    T_ArrayRefExprState,
    T_FuncExprState,
    T_ScalarArrayOpExprState,
    T_BoolExprState,
    T_SubPlanState,
    T_FieldSelectState,
    ……

    /*
    * TAGS FOR PLANNER NODES (relation.h)
    */
    T_PlannerInfo = 500,
    T_RelOptInfo,
    T_IndexOptInfo,
    T_Path,
    T_IndexPath,
    ……
    /*
    * 内存管理类
    */
    T_MemoryContext = 600,
    T_AllocSetContext,

    /*
    * TAGS FOR VALUE NODES (value.h)
```

```
    T_Agg,
    T_Unique,
    T_Hash,
    T_SetOp,
    T_Limit,

    /*
    * 执行查询计划状态类
    */
    T_PlanState = 200,//PlanState类是所有
state类的基类
    T_ResultState,
    T_AppendState,
    T_BitmapAndState,
    T_BitmapOrState,
    T_ScanState,
    ……
    T_SortState,
    T_GroupState,
    T_AggState,
    T_UniqueState,
    T_HashState,
    T_SetOpState,
    T_LimitState,
    */
    T_Value = 650,
    T_Integer,
    T_Float,
    T_String,
    T_BitString,
    T_Null,

    /*
    * List相关类
    */
    T_List,
    T_IntList,
    T_OidList,

    /*
    * TAGS FOR PARSE TREE NODES (parsenodes.h)
    */
    T_Query = 700,//所有语法树的基类
    T_InsertStmt,
    T_DeleteStmt,
    T_UpdateStmt,
    T_SelectStmt,
    ……
} NodeTag;
```

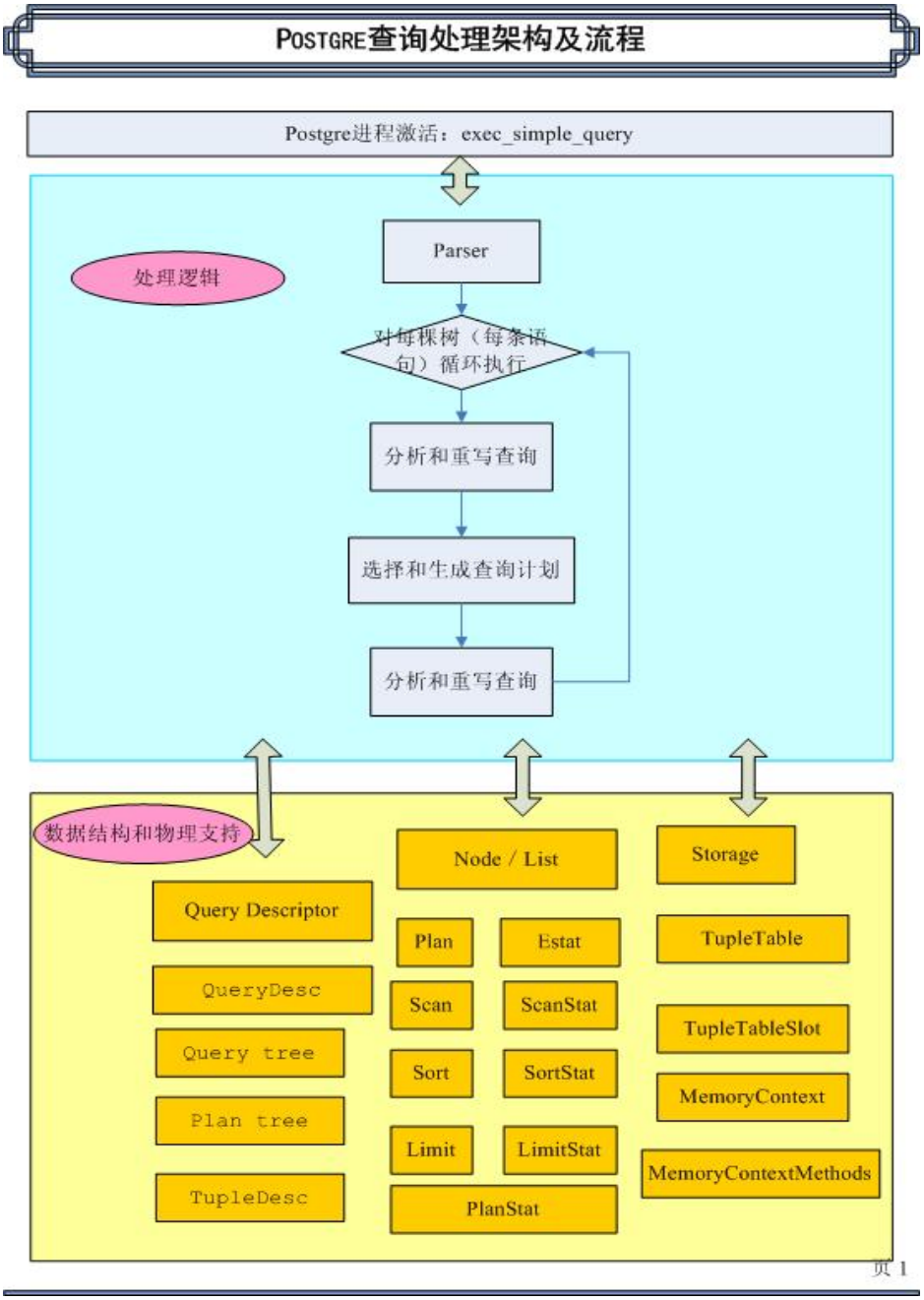
图表 11：NodeTag 枚举简表及注释

下面简要描述 PostgreSQL 中在查询计划的描述、查询各个结点的执行、查询状态的保存与传递，元组的存储与表示、内存管理相关数据结构用其层次。

4.2.2　查询后台处理实现与数据结构浏览

当后台处理进程激活后，进入 exec_simple_query 处理查询。这也是我们实习相关的重点分析入口。在 exec_simple_query 中各个模块的层次非常明显。共分为接口——逻辑——底层数据结构与库的物理支持。

具体模块层次与重要的数据结构如下图：下面就跟寻着这个层次结构由底向上的，详细分析 Parser,Rewriter,Planner,Executor 这四个部分的重要的数据结构和相关函数的具体实现。



图表 12：PostgreSQL 查询处理模块层次与数据结构

4.2.3　Parser 具体实现与相关数据结构

我们能把 SELECT 语句大体分解为以下的几个部分（这也是直观理解上 select 子树中的第一层元素）

SELECT opt_distinct target_list into_clause from_clause where_clause group_clause having_clause

下面我们以一个简单的“select * from student where age>20” 流程来加深理解 Postgre 处理查询的全过程。上述的 SQL 语句经过 Parser 阶段，进行词法和语法分析后，生成以 SelectStmt 为根的查询树。我们先看 SelectStmt 结构体中的内容：

```
typedef struct SelectStmt
{
    NodeTag  type;
```

```
    //以下的属性域只作为SelectStmts中的叶子结点
    List    *distinctClause; /* NULL, list of DISTINCT ON exprs, or
                              * lcons(NIL,NIL) for all (SELECT DISTINCT) */
    RangeVar   *into;   /* target table (for select into table) */
    List    *intoColNames; /* column names for into table */
    List    *intoOptions; /* options from WITH clause */
    OnCommitAction intoOnCommit; /* what do we do at COMMIT? */
    char    *intoTableSpaceName;  /* table space to use, or NULL */
    List    *targetList;  /* the target list (of ResTarget) */
    List    *fromClause;  /* the FROM clause */
    Node    *whereClause; /* WHERE qualification */
    List    *groupClause; /* GROUP BY clauses */
    Node    *havingClause; /* HAVING conditional-expression */

    //在一个代表值域的叶子结点中，以上的属性域都为空，但是下面的属性域被设置。
    //注意sublists中的元素只是表达式，并不能用ResTarget修饰。
    List    *valuesLists; /* untransformed list of expression lists */

    //这些属性域在SelectStmts上层和叶子中都有效。
    List    *sortClause;  /* sort clause (a list of SortBy's) */
    Node    *limitOffset; /* # of result tuples to skip */
    Node    *limitCount;  /* # of result tuples to return */
    List    *lockingClause; /* FOR UPDATE (list of LockingClause's) */

    //这些属性域在上层SelectStmts有效。
    SetOperation op;   /* type of set op */
    bool  all;   /* ALL specified? */
    struct SelectStmt *larg; /* left child */
    struct SelectStmt *rarg; /* right child */
    /* Eventually add fields for CORRESPONDING spec here */
} SelectStmt;
```

图表 13:SelectStmt 结构体

按照上面所给的数据结构，SelectStmt 远比我们想像的复杂，但是各基本成分还是有相应的表述，此外包含不少其它辅助执行的信息，以下是打开 postgre.conf 中相应开关 debug_print_parse、debug_print_rewritten、debug_print_plan、 debug_pretty_print 在日志输出的中间结果。表中红色部分是较为重要的子属性。可以看出使用树的后序遍历法可以得出执行步骤。

```
DETAIL:     {QUERY
                :utilityStmt <>
                :resultRelation 0
                :into <>
                :intoOptions <>
                :intoOnCommit 0
                :intoTableSpaceName <>
```

```
    :hasAggs false
    :hasSubLinks false
    :rtable (
{RTE
//Range Table Entry，以下是其中元素，以链表的方式存储。
// range table entry（RTE）代表的是在from之后的一个普通的关系，
//一个子查询或者是一个JOIN连接语句。
:alias <>    //别名
:eref    //eref对象存的是表的引用名和列的引用名
{ALIAS
:aliasname student
:colnames ("id" "name" "age")
}
//以下的信息是对所有的RTE有效。
:rtekind 0
    :relid 16386
    :inh true
    :inFromCl true
    :requiredPerms 2
    :checkAsUser 0
}
)
    :jointree
{FROMEXPR
:fromlist (
{RANGETBLREF
:rtindex 1
}
)
    :quals
{OPEXPR
:opno 521
:opfuncid 0
:opresulttype 16
:opretset false
:args (
{VAR //变量，见src/include/nodes/primnodes.h中的定义
:varno 1
:varattno 3
:vartype 23
:vartypmod -1
:varlevelsup 0
:varnoold 1
:varoattno 3
```

```
}
{CONST    //常数，见src/include/nodes/primnodes.h中的定义
    :consttype 23
    :constlen 4
    :constbyval true
    :constisnull false
    :constvalue 4 [ 20 0 0 0 ]
}
)
}
}
:targetList (
{TARGETENTRY    //TargetEntry的入口，以下是其参数
:expr    //展开部分略
:resno 1
:resname id
:ressortgroupref 0
:resorigtbl 16386
:resorigcol 1
:resjunk false
}
{TARGETENTRY
    :expr  //展开部分略
    :resno 2
    :resname name
    :ressortgroupref 0
    :resorigtbl 16386
    :resorigcol 2
    :resjunk false
}
{TARGETENTRY  //TargetList Entry
    :expr
    :resno 3
    :resname age
    :ressortgroupref 0
    :resorigtbl 16386
    :resorigcol 3
    :resjunk false
}
)
    :returningList <>
    :groupClause <>
    :havingQual <>
    :distinctClause <>
```

```
            :sortClause <>
            :limitOffset <>
            :limitCount <>
            :rowMarks <>
            :setOperations <>
            :resultRelations <>
            :returningLists <>
        }
```

图表 14:从 postgre.conf 中不同开关日志中看 Parse tree 在内存中的结构

4.2.4 Rewriter 具体实现与相关数据结构

第一步完成后，只是做了很简单、很粗糙的事情，然后，要进一步进行 rewrite，在交给优化器进行优化和路径选择之前，需要把 SelectStmt 转化成 Query，下表中列出了 Query 的成员变量。

```
typedef struct Query
{
    NodeTag  type;
    CmdType  commandType;   /* 类型：select、insert、update、delete、utility */
    /*    注意：如果commandType为: utility，优化器不会对该SQL进行进一步优化，因为这个SQL
    就是一些建表或者其他命令操作，无法进行路径选择和优化，这时候，executor直接    执行
utilityStmt这个Node对
    应的Struct.  对于select|insert|update|delete这些SQL，优化器都需要进行评估和优化。    */
    QuerySource querySource; /* where did I come from? */
    bool  canSetTag;  /* do I set the command result tag? */
    Node    *utilityStmt; /* non-null if this is a non-optimizable statement/
                            int   resultRelation; /* rtable index of target relation for
INSERT/UPDATE/DELETE; 0 for SELECT */
    RangeVar   *into;   /* target relation for SELECT INTO */
    List    *intoOptions; /* options from WITH clause */
    OnCommitAction intoOnCommit; /* what do we do at COMMIT? */
    char    *intoTableSpaceName;  /* table space to use, or NULL */
    bool  hasAggs;  /* has aggregates in tlist or havingQual */
    bool  hasSubLinks; /* has subquery SubLink */
    List    *rtable;   /* list of range table entries */
    FromExpr   *jointree;  /* table join tree (FROM and WHERE clauses) */
    List    *targetList;  /* target list (of TargetEntry) */
    List    *returningList; /* return-values list (of TargetEntry) */
    List    *groupClause; /* a list of GroupClause' s */
    Node    *havingQual;  /* qualifications applied to groups */
    List    *distinctClause; /* a list of SortClause' s */
    List    *sortClause;  /* a list of SortClause' 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    *resultRelations; /* integer list of RT indexes, or NIL */
    List    *returningLists; /* list of lists of TargetEntry, or NIL */
} Query;
```

图表 15:Query 数据结构

PostgreSQL rewriter 阶段返回的内存数据结构：仔细对比 Query 和 SelectStmt 的数据结构，它们是很相似的。其实，Select、Insert、Delete、 Update 最终都要转换成 Query。所以尽管前台命令表现出来的差距很大，但最后都可以归为相同的数据结构，这样也为下一阶段的统一处理作好了准备。

```
DETAIL:  (
         {QUERY
         :commandType 1
         :querySource 0
         :canSetTag true
         :utilityStmt <>
         :resultRelation 0
         :into <>
         :intoOptions <>
         :intoOnCommit 0
         :intoTableSpaceName <>
         :hasAggs false
         :hasSubLinks false
         :rtable (
         {RTE
         :alias <>
         :eref
         {ALIAS
         :aliasname student
         :colnames ("id" "name" "age")
         }
         :rtekind 0
              :relid 16386
              :inh true
              :inFromCl true
              :requiredPerms 2
              :checkAsUser 0
         }
         )
              :jointree
         {FROMEXPR
         :fromlist (
         {RANGETBLREF
         :rtindex 1
```

```
}
)
    :quals
{OPEXPR
:opno 521
:opfuncid 0
:opresulttype 16
:opretset false
:args (
{VAR
:varno 1
:varattno 3
:vartype 23
:vartypmod -1
:varlevelsup 0
:varnoold 1
:varoattno 3
}
{CONST
    :……
    :constvalue 4 [ 20 0 0 0 ]
}
//以下内容同parse tree中内容中一致，省略
}
}
)
```

图表 16:Rewritten parse tree

4.2.5　Planner 具体实现与相关数据结构

4.2.5.1　Plan 树的表示

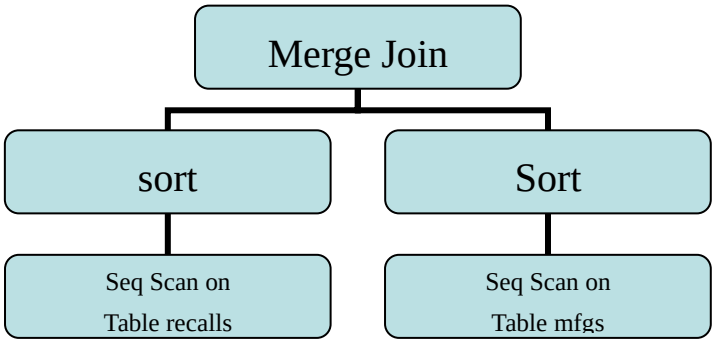
通过 PostgreSQL 提供命令 EXPLAIN，可以显示规划器生成的执行计划，我们可以研究对不同 SQL 查询语句生成执行计划。我们先看一句的执行过程。

EXPLAIN SELECT * FROM recalls, mfgs WHERE recalls.mfgname = mfgs.mfgname;

输出结果如下(Psql 命令台中)：

```
NOTICE:  QUERY PLAN:
Merge Join
  -> Sort
      -> Seq Scan on recalls
  ->  Sort
      -> Seq Scan on mfgs
```

从上面的过程可以看出，规划器/优化器生成执行计划如下图， 是一个树型的执行结构，正规执行方式是后序遍历方式：首先顺序检索 recalls 表，把结果装入内存，然后进行排序。然后对 mfgs 表进行顺序检索，装入内存后进行排序。最后再使用融合排序连接把两个结果进行连接，输出结果。

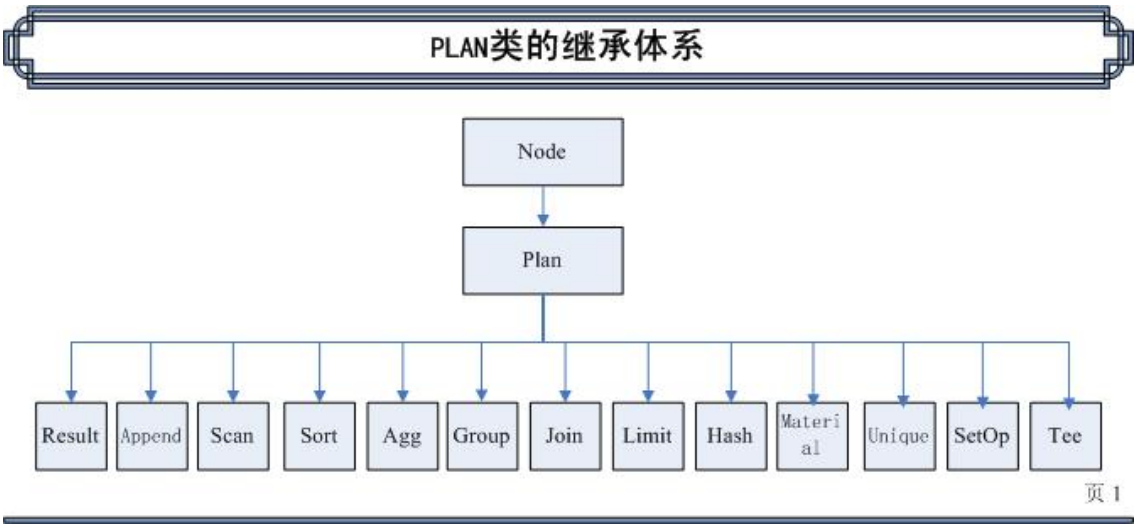


图表 17:Select 语句在内存中的执行树

4.2.5.2　Plan 相关类的继承体系

查询计划部分的数据结构设计为树型的形式。同时，所有的类都继承自抽象类plan类，使得在程序静态时看上去都是plan树，实际上在运行状态时，程序会根据NodeTag来判断是不同的计划类型，从而实现了抽象、统一的比较灵活的处理方式。

Plan 类的继承体系如下图所示：



图表 18:Plan 类的继承体系

4.5.2.3　Plan 结构及简要注释

```
//Plan是所有计划类型结点的基类
typedef struct Plan
{
    NodeTag       type;
    /*
    *估计代价所用
    */
    Cost      startup_cost; /* cost expended before fetching any tuples */
    Cost      total_cost;       /* total cost (assuming all tuples fetched) */
```

```
        double          plan_rows;          /* number of rows plan is expected to emit */
        int             plan_width;         /* average row width in bytes */

        /*
        * 所有plan类型的一般结构
        */
        List    *targetlist;        /* target list to be computed at this node */
        List    *qual;          /* implicitly-ANDed qual conditions */
        struct Plan *lefttree;      /* input plan tree(s) */
        struct Plan *righttree;
        List    *initPlan;

        Bitmapset  *extParam;
        Bitmapset  *allParam;

        int             nParamExec;         /* Number of them in entire query. This is to
                                            * get Executor know about how many PARAM_EXEC
                                            * there are in query plan. */
    } Plan;
```

图表 19：Plan 结构体

4.5.2.4　查询优化实习涉及的相关 plan 类型

```
//排序plan结点
typedef struct Sort
{
    Plan     plan;
    int           numCols;      /* number of sort-key columns */
    AttrNumber *sortColIdx;     /* their indexes in the target list */
    Oid         *sortOperators; /* OIDs of operators to sort them by */
} Sort;
```

图表 20：Sort plan 结构体

```
//Limit plan
typedef struct Limit
{
    Plan     plan;
    Node    *limitOffset;   /* OFFSET parameter, or NULL if none */
    Node    *limitCount;        /* COUNT parameter, or NULL if none */
} Limit;
```

图表 21:Limit plan 结构体

4.2.6　Executor 具体实现与相关数据结构

查询执行器根据优化器生成的最佳执行方案开始执行。首先要会要求在最顶的操作返一个结果集。每一个操作都会把自己的输入转换成输出，而它们的输入有可能是来自树中更底层的操作的结果。当最顶层

的操作完成它的计算后，结果就被返回应用程序。

更直白就是说，外部 query 数据在对于执行树中不同结点的访问是异步的。比方说，执行树向其左结点传递一个元组，而左结点其下是一个 sort 结点，当递归下归执行到该结点时，虽然上层结点只访问了一个数据，而 sort 结点却要访问完所有的数据。返回上层结点。执行树继续下一个元组的输入，无论它是否在 sort 中已经被访问过了。

下面就让我们从查询描述开始，对查询的描述数据结构、元组数据结构和执行状态数据结构做一个逐一的探查：

先看看要执行的一个查询的描述数据结构：

```
//Query Descriptor 类
typedef struct QueryDesc
{
    CmdType Operation;//结构体的标识码，Executor 中所有结构体均有
    Query *parsetree;//SQL 语句的语法解析树
    Plan *plantree;//生成的Query Plan 的根节点
    CommandDest dest;//标识结果送往的目的地
    const char *portalName;//用于支持PostgreSQL 的portal 功能
    TupleDesc tupDesc;//元组的说明，即为模式定义
} QueryDesc;
```

图表 22：Query Descriptor 结构体

再看看元组的表示与存储的相关数据结构：

```
//元组表
typedef struct TupleTableData
{
    int           size;             /* size of the table (number of slots) */
    int           next;             /* next available slot number */
    TupleTableSlot array[1];    /* VARIABLE LENGTH ARRAY - must be last */
} TupleTableData;               /* VARIABLE LENGTH STRUCT */

typedef TupleTableData *TupleTable;
//元组项结构
typedef struct TupleTableSlot
{
    NodeTag       type;             /* 表示节点的类型*/
    bool      tts_isempty;  /* true = slot is empty */
    bool      tts_shouldFree; /* 不用后是否应该pfree() */
    bool      tts_slow;     /* saved state for slot_deform_tuple */
    HeapTuple tts_tuple;        /* 元组在堆中的值，如果没有则为空*/
    TupleDesc tts_tupleDescriptor;   /* 元组对应的模式信息*/
    MemoryContext tts_mcxt;     /* slot itself is in this context */
    Buffer        tts_buffer;       /* 元组指向的磁盘缓冲位置*/
    int           tts_nvalid;       /* # of valid values in tts_values */
```

```
    Datum       *tts_values;        /* current per-attribute values */
    bool    *tts_isnull;        /* current per-attribute isnull flags */
    MinimalTuple tts_mintuple;  /* set if it's a minimal tuple, else NULL */
    HeapTupleData tts_minhdr;   /* workspace if it's a minimal tuple */
    long     tts_off;      /* saved state for slot_deform_tuple */
} TupleTableSlot;
```

图表 23：元组的表示与存储

最后，我们来看看查询执行时用来存储和保存状态的数据结构。

Executor State 是查询执行时状态转换和记录的重要数据结构。所有的结构都从 EState 继承。

下表就是 Estate 类的描述：

```
// Estate：执行状态类
typedef struct EState
{
    NodeTag       type;//结构体的类型

    /* 所有查询类型的基本对象*/
    ScanDirection es_direction; /* 表扫描的方向*/
    Snapshot  es_snapshot;  /* 执行时的系统事务快照，用于支持事务处理，需要初始化*/
    Snapshot  es_crosscheck_snapshot; /* crosscheck time qual for RI */
    List    *es_range_table; /* 查询涉及到的表*/

    /* Info about target table for insert/update/delete queries: */
    ResultRelInfo *es_result_relations; /* 结果表数组*/
    int           es_num_result_relations;        /* 数组长度*/
    ResultRelInfo *es_result_relation_info;       /* currently active array elt */
    JunkFilter *es_junkFilter;  /* 当前激活的JunkFilter 结构，在执行时变化*/

    TupleTableSlot *es_trig_tuple_slot; /* for trigger output tuples */

    Relation es_into_relation_descriptor;     /*用于SELETCT INTO 语句情况，记录SELETCT
INTO 的关系模式*/
    bool      es_into_relation_use_wal;
    /* 参数相关信息*/
    ParamListInfo es_param_list_info;     /* 用于处理带参数的批量查询的数据结构，其中记
录了参数的信息*/
    ParamExecData *es_param_exec_vals;    /* 为子查询提供的指针，记录参数信息*/

    /* 其他工作状态*/
    MemoryContext es_query_cxt; /* 查询时的存储上下文*/

    TupleTable    es_tupleTable;     /* 结果表*/
```

```
    uint32        es_processed; /* 已经处理的元组个数*/
    Oid           es_lastoid;        /* INSERT 语句情况下，上次处理操作的oid */
    List    *es_rowMarks;   /* 语句影响过的行的列表*/

    bool      es_is_subquery;    /* true if subquery (es_query_cxt not mine) */

    bool      es_instrument;     /* true requests runtime instrumentation */
    bool      es_select_into; /* true if doing SELECT INTO */
    bool      es_into_oids; /* true to generate OIDs in SELECT INTO */

    List    *es_exprcontexts;    /* List of ExprContexts within EState */

    /*
         以下属性用于在处理SQL 语句中的函数、谓词和约束等，做求值操作
    */
    ExprContext *es_per_tuple_exprcontext;

    Plan    *es_topPlan;         /* link to top of plan tree */
    struct evalPlanQual *es_evalPlanQual;     /* 指向求值函数*/
    bool    *es_evTupleNull; /* 元组是否为空*/
    HeapTuple  *es_evTuple;      /* shared array of EPQ substitute tuples */
    bool      es_useEvalPlan; /* 是否使用求值*/
} EState;
```

图表 24：Estate：执行状态类

还有其他的和本实习相关的状态类：

```
/* ----------------
 *   SortState information
 * ----------------
 */
typedef struct SortState
{
    ScanState ss;                /* its first field is NodeTag */
    bool      randomAccess; /* need random access to sort output? */
    bool      sort_Done;         /* sort completed yet? */
    void    *tuplesortstate; /* private state of tuplesort.c */
} SortState;
/* ----------------
 *   LimitState information
 * ----------------
 */
typedef enum
{
```

```
    LIMIT_INITIAL,                  /* initial state for LIMIT node */
    LIMIT_EMPTY,                /* there are no returnable rows */
    LIMIT_INWINDOW,                 /* have returned a row in the window */
    LIMIT_SUBPLANEOF,           /* at EOF of subplan (within window) */
    LIMIT_WINDOWEND,            /* stepped off end of window */
    LIMIT_WINDOWSTART           /* stepped off beginning of window */
} LimitStateCond;

typedef struct LimitState
{
    PlanStateps;                /* its first field is NodeTag */
    ExprState  *limitOffset;    /* OFFSET parameter, or NULL if none */
    ExprState  *limitCount;     /* COUNT parameter, or NULL if none */
    int64         offset;           /* current OFFSET value */
    int64         count;            /* current COUNT, if any */
    bool      noCount;      /* if true, ignore count */
    LimitStateCond lstate;      /* state machine status, as above */
    int64         position;     /* 1-based index of last tuple returned */
    TupleTableSlot *subSlot;    /* tuple last obtained from subplan */
} LimitState;
//执行计划的状态结点
typedef struct PlanState
{
    NodeTag       type;
    Plan   *plan;           /* associated Plan node */
    EState   *state;            /* at execution time, state's of individual

    struct Instrumentation *instrument; /* Optional runtime stats for this*/
    List    *targetlist;        /* target list to be computed at this node */
    List    *qual;          /* implicitly-ANDed qual conditions */
    struct PlanState *lefttree; /* input plan tree(s) */
    struct PlanState *righttree;
    List    *initPlan;      /* Init SubPlanState nodes (un-correlated expr
                                 * subselects) */
    List    *subPlan;       /* SubPlanState nodes in my expressions */
    Bitmapset  *chgParam;       /* set of IDs of changed Params */

    TupleTableSlot *ps_OuterTupleSlot;    /* slot for current "outer" tuple */
    TupleTableSlot *ps_ResultTupleSlot; /* slot for my result tuples */
    ExprContext *ps_ExprContext;     /* node's expression-evaluation context */
    ProjectionInfo *ps_ProjInfo;     /* info for doing tuple projection */
    bool      ps_TupFromTlist;/* state flag for processing set-valued
                                 * functions in targetlist */
} PlanState;
```

图表 25：sort、limit、planstat 状态结构体

第五章 查询优化实习相关的基本查询运算分析

5.1 分析代价计算和来源

当所有生成可执行的计划后，优化器寻找执行代价最小的计划。每一个计划有了一个估计的执行代价。代价的估计是基于磁盘 I/O 的读取次数。一个运算符从磁盘读取一个大小为 8192 字节的块花费的代价为 1 个单位。对于 CPU 时间的估计也是基于磁盘的读取次数，不过只是其的小一部份。一般而言，CPU 处理一个元组所需要时间设为从磁盘中读取一个块时间的百分之一。另外每一个查询运算也有不同的花销估计。比方说，一次顺序检索一个表的代价是按这个表所占 8k 块的数量，加上少量的 CPU 花销。

每一个花销的估计需要根据一些基本的统计数据。关于表的基本统计数据储存在 PostgreSQL 一个数据库中：pg_class 和 pg_statistic，我们的数据库每一个表在 pg_class 表中都有一个表项。当然其中还包括了有关视图、索引等的信息。对任意一个给定的表，pg_class.relpages 中表项中包含一个该表所占用 8KB 大小页面数的估计值。而 pg_class.reltuple 表项中包含对一个表中元组个数的估计值。

当你建立一个新表时，relpages 被设成 10 个页面，而 reltuples 被设置成 1000 个元组。当你在表格中增删数据时，PostgreSQL 并不维护 pg_class 中的估计值。这时你就需要手动去维护 pg_class：主要命令有 VACUUM, ANALYZE, CREATE INDEX.

5.2 顺序查询运算（Seq Scan）

顺序查询运算是最基本的查询运算。任何一个单表格的查询可以通过顺序查询运算实现。顺序查询运算从表格的开始处顺序访问直到表格的结束处。在查询的过程中，其可以实现查询的约束条件（如 where 子句的约束）。如果该元组满足约束条件，其就会被添加到结果项中。

尽管顺序查询不会把那些无效的行放入到结果集中，不过它会读出那些无效行（由于被删除的原因）。所以对一个经常更新的表格来说，这样的运算代价是很大的。

规划器/优化器在没有可以满足查询条件可以选择的情况下使用顺序查询运算。当然也有可能是在规划器/优化器发现直接使用顺序查询运算会比先排序去满足某一个约束代价更低时采用。

5.3 索引查询运算（Index Scan）

索引查询运算是通过启遍历索引结构来完成运算的。如果你对一个有索引的项指定了一个起始值（比方说 where record_id >=1000），索引查询会在合适的值处开始运算。如果你指定了一个结束值(好像 WHERE record_id < 2000)，它就会在找到第一个比结束值大的值时结束。

索引查询运算比起顺序查询运算有两个优点。首先，对于一个顺序查询运算，它必须读出表中的每一个元组——它只能在 where 子句评估完每一行之后才能移除相应的结果。而索引查询运算在你提供起始值和结束值的情况下不需要读取每一行。第二，顺序查询运算只能在返回元组在表中的顺序，而索引查询运算可以按索引的顺序返回。

规划器/优化器在通过遍历索引结构可以降低结果集的大小时或者在当使用索引可以避免一次显式排序运算时（排序的结果和索引序相同），使用索引查询运算，

5.4 排序运算（Sort）

排序运算的目的是使结果集变得有序。PostgreSQL 运用了两种不同的排序策略：内排序和外排序。通过调整运行时 sort_mem 参数的值来改变 PostgreSQL 的行为模式。如果结果集的大小超过了 sort_mem 时，排序运算将把输入集转到一系列小的已排好序的文件中，然后再回来作处理。如果结果集的大小小于 sort_mem*1024 字节，那么直接在内存中采用 QSort 算法。

不像顺序查询运算和索引查询运算，一开始就能返回结果。查询运算必须等到运算处理完毕后才能返回结果。

显然，排序运算可以用于多种目的。最明显的是排序运算可以满足 ORDER BY 子句。有些查询运算要求它们的输入结果必须有序。比方说，UNIQUE 运算是通过排序来发现和消除重复元组。在其它的运算中，如组运算和连接运算都会用到排序运算。

5.5 连接运算（Join）

连接运算主要是为了连接两个表，它需要两个不同的输入集：一个内表和一个外表，并且两个被连接

表必须按照连接项排好序。以下有三种可能的连接策略：

嵌套循环连接(Nested Loop)：对左边的关系里面找到的每条元组都对右边关系进行一次扫描。 这个策略是最容易实现，但是可能会很耗费时间。

融合排序连接(Merge join)：在连接开始之前，每个关系都对连接字段进行排序。 然后对两个关系并行扫描，匹配的行就组合起来形成连接行。

散列连接(hash join) ：首先扫描右边的关系，并用连接的字段作为散列键字装载进入一个散列表，然后扫描左边的关系，并将找到的每行用作散列键字来以定位表里匹配的行。

具体使用哪一个策略也是由计算的结果决定的，规划器/优化器会计算出所有的连接关系的代价，选择运行代价最小方式生成下一阶段的执行指令。

参考文献:

[1] PostgreSQL 中国. PostgreSQL 8.1 中文文档.

[2] Bruce Momjian: PostgreSQL: Introduction and Concepts, http://www.postgresql.org/docs/awbook.html

[3] Diomidis.Spinellis. 代码阅读方法与实践. 北京:清华大学出版社,2004

[4] W.Richard Stevens. UNIX 环境高级编程. 北京:机械出版社,2006