标题：数据库安全

来源：

编辑日期：2020.06.08

目录：

正文：

# 第一章 数据库整体架构与安全需求

### 1.1 问题模型

1. 用户
   * 最终用户：通过应用程序与数据库交互
   * 应用程序员：开发数据库和应用程序
   * 数据库管理员：数据定义、安全、完整性约束、监控、日志和文件管理、故障故障恢复、优化
2. 数据模型（概念模型）
3. 服务提供者：表示层
4. 数据库体系结构
   * 单用户（单机）
   * 主从式
   * 分布式
   * 客户/服务器模式
5. 网络层
6. 设备层
7. 服务提供者：逻辑层
8. 数据库产品
   * Orecle
   * Microsoft SQL Server
   * Microsoft Access
   * PostgreSQL
9. 角色
10. 身份认证
11. 数据库接口
    * 数据定义语言DDL：create、alter、drop
    * 数据操作语言DML：Insert、update、delete
    * 数据控制语言：DCL：grant、revoke
    * 数据查询语言DQL：select
    * 事务控制语言TCL：commit、rollbakc、savepoint
12. 数据库层
    * master：系统级信息
    * msdb：代理、警报
    * tempdb：
13. 数据层：通过文件组管理文件
    * 主要数据文件.mdf：启动、用户管理
    * 次要数据文件.ndf：用户数据
    * 事务日志文件.ldf：日志

### 1.2 数据库架构

（1）数据库设计过程

1. 需求分析：信息整理、分析
   * 分析用户活动，生成业务流程图
   * 分析人工处理与计算机处理范围，生成人机界面
   * 分析用户数据，生成数据流图。包括数据流、数据加工或处理、数据存储、外部实体
   * 分析系统数据，生成数据字典
   * 生成软件结构图或模块图
   * 生成反应系统全貌的数据流图与数据
2. 结构设计：概念结构、逻辑结构、物理结构
   * 概念模型：ER模型，包括实体、属性、联系
   * 逻辑模型：数据库支持的结构，分为层次模型、网状模型、关系模型、面向对象模型。关系模式、视图
   * 物理结构：索引，分为索引、聚簇、哈希、存储结构、系统配置
3. 行为设计：功能设计、事务设计、程序设计
   * 功能需求：操作、结构和操作约束、事务、操作频率、响应时间
   * 功能设计
   * 事务设计：输入设计、输出设计
4. 数据库实施：数据库、应用程序
5. 数据库运行和维护
   * 备份和恢复、安全性和完整性控制、监视、分析、性能、重组

（2）数据库安全

数据库安全包括：

（1）系统运行安全

物理安全、运行环境安全等。

（2）系统信息安全

数据安全性、独立性、完整性、并发控制、故障恢复等。

# 第二章 PostgreSQL架构

### 2.1，术语和概念

1. 数据库集簇：一组数据库的集合，由一个PostgreSQL服务器管理。本质上为文件目录，可理解为根目录，执行initdb后会创建基础目录
2. 数据库：一组数据库对象的结合。每个数据库可理解为一个子目录，目录名为数据库oid
3. 数据库对象：堆表、索引、序列、视图、函数等。小于1GB的表或索引都存储为单个文件refilenode(=oid)，以及两个管理文件refilnode\_fsm,refilnode\_vm，分别表示空闲空间映射和可见性映射文件。当大于1GB时存储为relfilenode.1以及.2等多个文件，便于适配文件系统以及复制等。可通过--with-segsize改变表和索引的最大文件大小。
4. 对象标识符（object identifier, oid）:4字节整型，每个数据库对象都有唯一的标识符，对应关系存储于系统目录中

### 2.2 数据库文件以及存储结构

（1）底层数据结构

1. 堆表文件结构：划分为页（区块），默认为8kb，从0开始编号，内部构造取决于文件类型

* 头部：24b（src/include/storage/bufpage.h）
  + typedef struct PageHeaderDate{

uint16 pd\_lsn 本页面最后变更对应的xlog的标识

uint16 pd\_checksum 本页面校验和

LocationIndex pd\_lower 行指针末尾，即空闲空间起始位置

LocationIndex pd\_upper 新堆元组起始位置，即空闲空间终止位置

LocationIndex pd\_special 特殊用途空间的其实位置，在堆表页中指向页尾，在索引页中指向数据结构的root节点

uint16 pd\_pagesize\_version 页大小即页面布局版本号

TransactionId pd\_prune\_xid 本页中可剪枝的最老元组中的xid

ItemIdData pd\_linp[FLEXIBLE\_ARRY\_MEMBER] 行指针的数组

}

* 堆元组：数据记录
  + 每个元组都有元组标识符（tuple identifier, TID），即元组索引，由元组所属页面的区块号和指向元组的行指针的偏移号
  + 当元组大小超过2kb时，采用TOAST超大属性存储技术
* 行指针：4b，指向对元组，即项目指针。行指针存储为数组，作为堆元组的索引
  + typedef struct ItemIdData{

unsigned lp\_off:15, 相对于页面起始位置的元组偏移量

lp\_flags:2, 行指针状态，参加#define

lp\_len:15, 元组长度（字节）

}

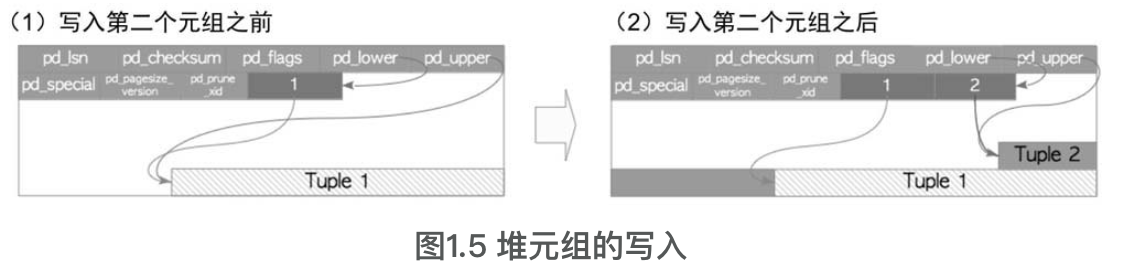
#define LP\_UNUSED 0 未使用

#define LP\_NORMAL 1 使用

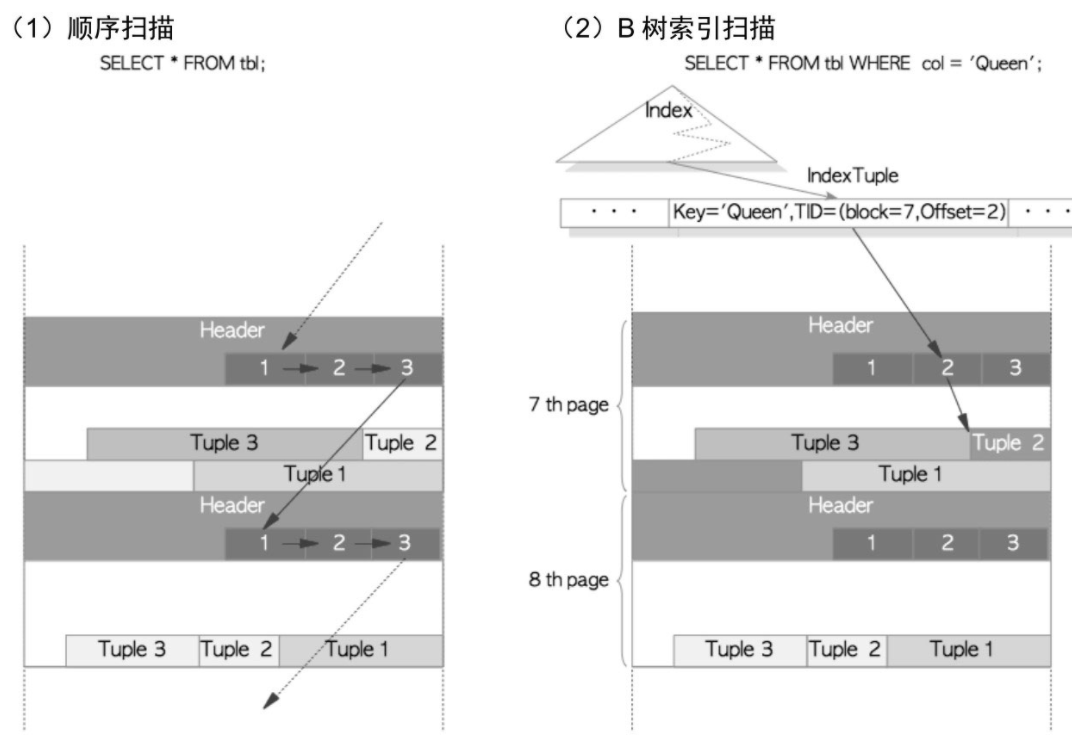
#define LP\_REDIRECT 2 重定向

#define LP\_DEAD 3 死元组

1. 写入堆元组



1. 读取堆元组
   * 顺序扫描
   * B树索引扫描
   * TID扫描
   * 位图扫描
   * 索引扫描



（2）数据库文件



（3）系统表

1. pg\_database：数据库datname oid
2. pg\_class：堆表对象relname oid

（4）进程架构

PostgreSQL采用单机体系，C/S模式，多进程架构

1. Postgre服务进程postgre server process：所有数据库集簇管理进程的父进程
   * + 启动postgre时在内存中分配共享内存，监听一个端口，默认为5432。然后fork各种后台进程，以及复制相关进程和后天工作进程，等待客户端请求。每当接收到一个客户端请求时，启动一个后端进程
2. 后端进程backend process：处理客户端的请求
   * 通过单条TCP连接与客户端通信，并在客户端断开连接时终止。每个连接只能操作一个数据库，连接时需要指定数据库。客户端连接数默认为100。
   * PostgreSQL没有原生的连接池，如果客户端频繁断开连接，会导致创建进程的开销，可使用池化中间件
3. 后台进程background process：数据库管理任务，例如清理过程与存档过程
   * background writer：将共享缓冲池中的脏页逐渐刷入持久化存储中
   * checkpointer：处理检查点
   * autovacum launcher：周期性地启动自动清理工作进程，即向postgre服务器请求创建进程
   * WAL writer：周期性将WAL缓冲区中的WAL数据刷入持久性存储中
   * statistics collector：收集统计信息，用于系统视图
   * logging collector：将错误消息写入日志文件
   * archiver：将日志归档
4. 复制相关进程replication assocaated preocess：流复制
5. 后台工作进程background worker process：

（5）内存架构

1. 本地内存：每个后端进程都有一块本地内存
   * work\_mem：当执行order by和distinct时使用该区域对元组做排序
   * maintenace\_work\_mem：
   * temp\_buffers：执行器使用该区域存储零时表
2. 共享内存区域：固定大小的几个子区域
   * shared buffer pool：将表和索引的页面从持久化存储加载至此，可直接操作他们
   * WAL buffer：WAL机制确保服务故障不会导致任何数据丢失，也才xlog，即事务日志。是WAL写入持久化存储之前的缓冲区
   * commit log：提交日志，clog，为并发控制机制保存所有事务状态，包括进行中，已提交，已终止等
3. 系统内存
   * 访问控制的子区域，如信号量、轻量级锁、共享和排他锁等

## 2.3 系统工作流程

（1）后端进程的子系统

后端进程的工作模块：解析器、分析器、重写器、计划器、执行器

1. 解析器：根据SQL语句生成一棵语法解析树
   * 输入：select id,data from tbl\_a where id < 300 order by data;
   * 功能：检查语法，但不会检查输入的语义，如不存在的表
   * 语法解析树根节点

typedef struct SelectStmt{

NodeTag type;

List \*distinctClause;

IntoClause \*intoCalause; //select into的目标

List \*targetList; //结果目标列表 -- struct targetlist \*

List \*fromClause; //From子句

Node \*whereClause; //Where限定条件

List \*groupClause; //Group by子句

Node \*havingCaluse; //Having条件表达

List \*windowClause; //Window name as

List \*valuesLists; // 为转换的表达式列表

List \*sortClause; //排序子句

Node \*limitOffset; //需要跳过的元组数目

Node \*limitCount; //需要返回的元组数目

List \*lockingClause; //锁子句列表

Withclause \*withClause; //With子句

Setoperation op; //set操作的类型

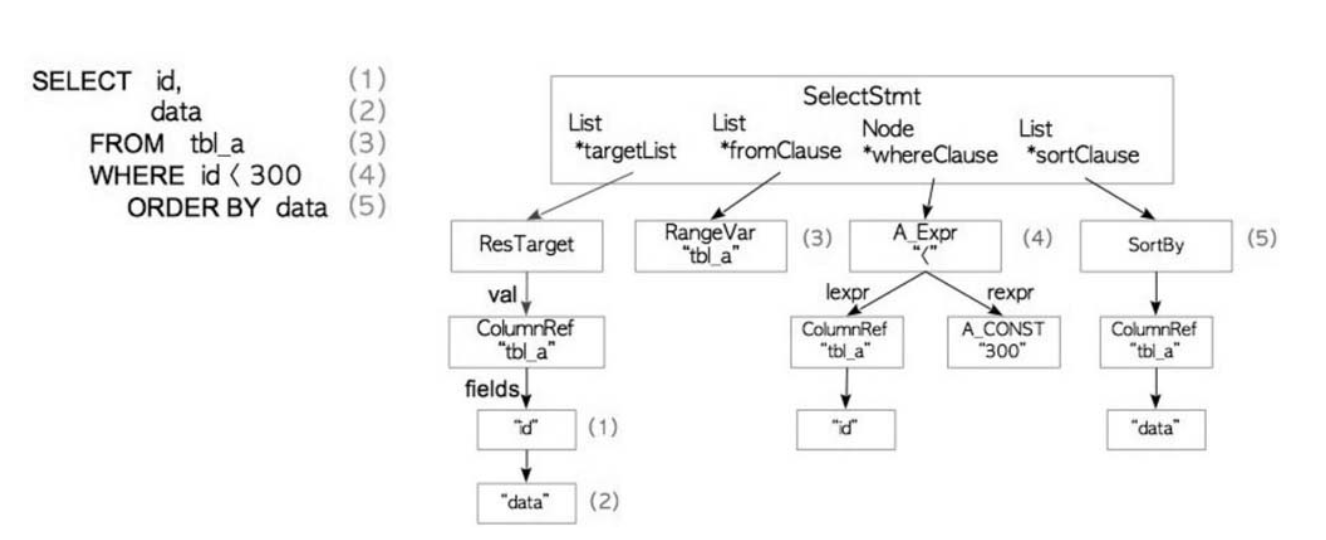
bool all; //是否指明了ALL选项

struct SelectStmt \*larg; //左子节点

struct SelectStmt \*rarg; //右子节点

}

* 输出：语法解析树



1. 分析器：对语法解析树进行语义分析，生成一棵查询树
   * 输入：语法解析树
   * 功能：对语法解析树进行语义分析，产生一棵查询树
   * typedef struct Query{ //parsenode.h

NodeTag type;

CmdType command Type;

uint32 queryId //查询标识符，可由插件配置

bool canSetTag; //

Node \*utilityStmt; //Declare cursor或不可优化的语句

int resultRelation; //增删改语句的目标关系索引

bool hasAggs; //是否在目标列表或habing表达式中知否指定聚合函数

bool hasWindowFuncs; //tlist是否包含窗口函数

bool hasSubLinks; //是否包含子查询 sublink

bool hasDinstionctOn; //是否包含来自Distinct on的distinct子句

bool hasRecursive; //是否制作了with recursive

bool hasModifyingCTE; //是否在With子句中包含了Insert/Uptate/Delete

bool hasForUpdate; //是否指定了For key Update/Share

bool hasRowSecrity; //是否使用了行安全策略

List \*cteList; //cte列表

List \*rtable; //查询所用到关系的列表

FromExpr \*jointree; //表连接树，存储From 与 Where子句

List \*targetList; //查询的列的列表

List \*withCheckOptions;

Onconflict \*onConflict;

List \*returningList; //返回值列表

List \*groupClause;

List \*groupSets; //如果查询带group语句

Node \*havingQual; //分组的having条件列表

List \*windowClause; //窗口子句列表

List \*distinctClause;

List \*sortClause;

Node \*limitOffset; //跳过的元组数目

Node \*limitCount; //Limit返回的元组数目

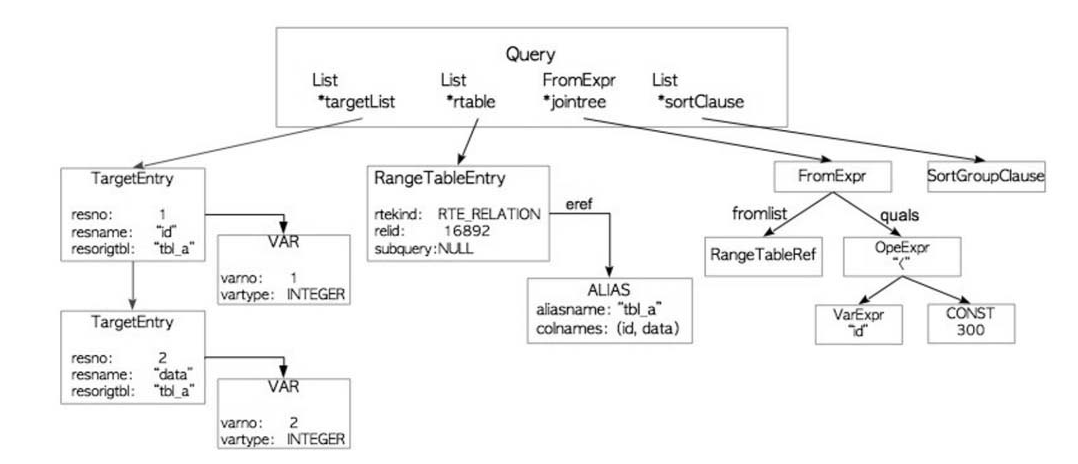
List \*rowMrks;

Node \*setOperations;

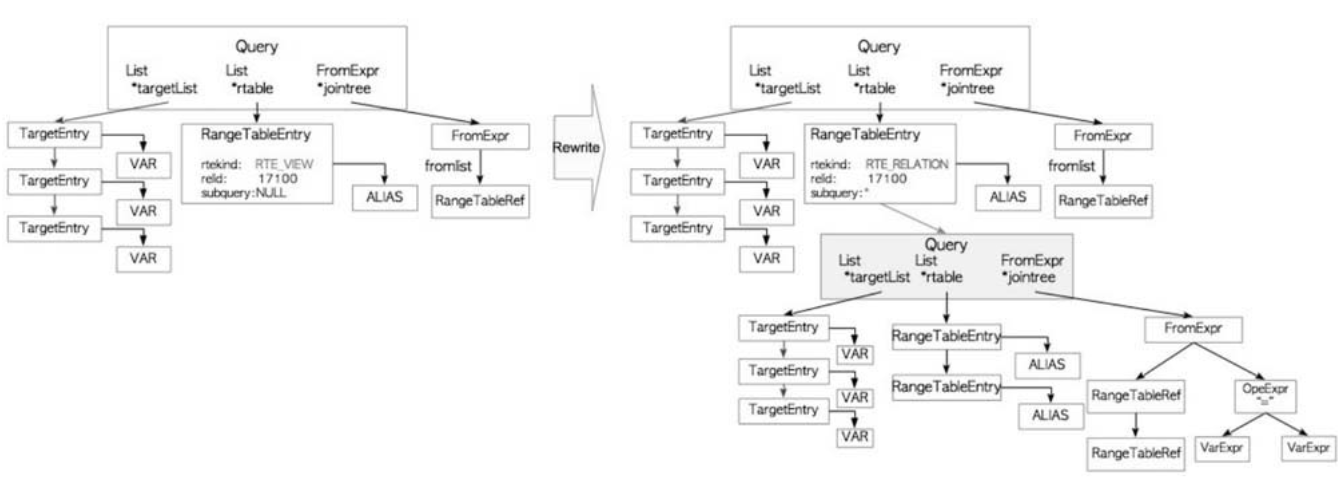
List \*constraintDeps;

}

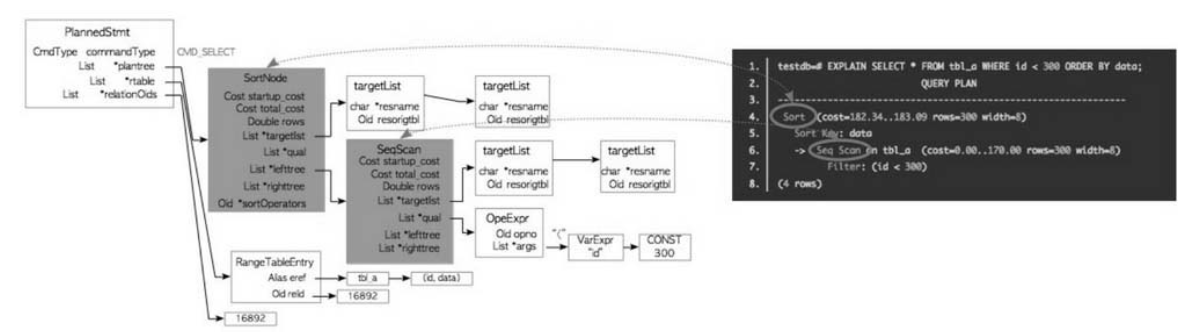
* + 输出：查询树



1. 重写器：按照规则系统中存在的规则对查询树进行改写
   * 输入：查询树
   * 功能：根据存储在pg\_rules中的视图规则对查询树进行转换
   * 输出：



1. 计划器：基于查询树生成一棵执行效率最高的计划树
   * 输入：从重写器输入一棵查询树
   * 功能：基于查询树生成一个能被执行器高效执行的查询计划树



* 输入：包含多个计划节点的计划树

1. 执行器：按照计划树中的顺序访问表和索引，执行相应的查询
   * 功能：预先分配与分配的内存空间，如temp\_buffers和work\_mem，以及临时文件等

### 2.4 结果与分析

（1）代价

PostgreSQL中有启动代价、运行代价和总代价，总代价=启动代价+运行代价

* 启动代价：当读取到第一条元组前所花费的代价，如索引扫描时读取到一个元组的代价
* 获取全部元组的代价

（2） 单表查询的代价估计

PostgreSQL的查询优化是基于代价的。costsize.c中的函数用于估计各种操作的代价。如执行器的操作代价函数为顺序扫描代价cost\_seqscan()，索引扫描代价cost\_index()，排序代价cost\_sort

1. 顺序扫描代价

* 启动代价 = 0
* 运行代价: run\_cost = cpu\_run\_cost + disk\_run\_cost

=(cpu\_tuple\_cost+cpu\_operator\_cost)×N-tuple+ seq\_page\_cost × N-page

其中，seg\_page\_cost = 1.0, cpu\_tupple = 0.01, cpu\_operator\_cost = 0.0025；（postgresql.conf），N-tuple = 元组总数，N-page = 页面总数

1. 索引扫描代价：PostgreSQL支持B树，GiST, GIN和BRIN等索引.
   * 启动代价：start-up\_cost = {ceil(log2(N-(index, tuple)) + (H-index+1)x50)} x cpu\_operator\_cost

其中，H-index是索引树的高度，cpu\_operator\_cost=0.0025

* 运行代价： run\_cost = (index\_cpu\_cost + table\_cpu\_cost) + (index\_io\_cost + table\_io\_cost)

其中：index\_cpu\_cost = Selectivity x N-(index, tuple) x (cpu\_index\_tuple\_cost + qual\_op\_cost)

table\_cpu\_cost = Selectivity x N-tuple x cpu\_tuble\_cost

index\_io\_cost = ceil(Selectivity x N-(index, page) x random\_page\_cost)

cpu\_index\_tuple\_cost 0.005, random\_page\_cost = 4.0, qual\_op\_cost = 0.0025（索引求值的代价），选择率 = { 高频值most\_common\_vals, 高频值的频率most\_common\_freqs} （未完）

1. 排序代价：包括Order by，归并连接的预处理操作，其他函数。如果能在工作内存中放下所有元组，使用快速排序算法，否则创建临时文件，使用文件归并排序算法。
   * 启动代价：start-up\_cost = C + comparison\_cost x N-sort x log2(N-sort)

其中，C = 上一次扫描的总代价