MySQL 源代码分析

余上

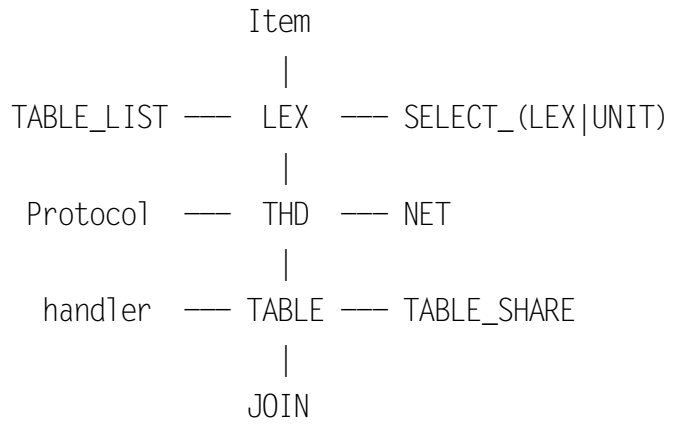
2007-11-21

内容目录

第一章 MySQL 主要结构

第一节 MySQL 主要对象关系

如下图:



- LEX,Item,SELECT_(LEX|UNIT),TABLE_LIST,SQL 语法分析以及 SQL 语句的抽象表示,Item 用于实现表达式,如查询条目,函数,where,order,group,on 子句等等.SELECT_(LEX|UNIT)则用来表达 SELECT ()和 UNION 操作.TABLE_LIST 则用来表达 JOIN 操作.
- Protocol 和 NET,前者实现数据库服务器和客户端的通信协议,例如,构建协议包等,后者提供网络支持,如原始数据的读写.
- TABLE_SHARE 和 TABLE,前者代表表的元信息,例如字段定义,索引定义等等,后这代表一个打开的表实例.
- handler,存储引擎接口.
- JOIN,SQL 的执行引擎,MySQL 的中查询的执行要经历准备,优化,执行几个阶段.

第二节 表定义及表

一,TABLE_SHARE 和 frm 文件

TABLE_SHARE 用来代表数据库表的元数据.frm 文件是 TABLE_SHARE 的永续存储,对象的实例从 frm 文件构建.重要的成员如下:

名称	类型	说明
field	Field**	字段定义,组成如下: (a)(fields+1)个 Field* + (b)interval_count 个 TYPELIB + (c)(fields + interval_parts + keys + 3)个 my_string + (d)(n_length + int_length + com_length)字节 **注**:n_length 等于(int)frm+268,而 int_length 等于(int)frm+274,com_length 等于(int)frm+284 **注**:(d)开始到 n_length+int_length 是字段的名字字符串池,n_length+int_length 开始处是注释字符串池. **注**:没有成员变量显式的指向(c)(d)
intervals	TYPELIB*	指向(b)处
key_info	KEY*	索引定义.这个指针指向一个 KEY 类型的数组,数组的大小由 keys 决定,在 KEY 类型数组的后面是一个 KEY_PART_INFO 类型的数组,数组大小由 key_parts 决定
keys	uint	索引个数
key_parts	uint	总的索引”分量”个数
db_type	handlerton*	存储引擎
ref_count	uint	引用计数,get_table_share 和 release_table_share 函数维护之
open_count	uint	打开的 TABLE 的个数
reclength	ulong	记录长度
rec_buff_length	uint	记录缓冲区长度,这是由下面公式计算出来的 ALIGN_SIZE(share->reclength + 1 + extra_rec_buf_length),reclength 如上所示,extra_rec_buf_length 是 frm 文件+59 处的值
default_values	byte*	默认值.rec_buff_length 字节

名称	类型	说明
system	bool	是否系统表
crypted	bool	frm 文件是否加密
is_view	bool	是否视图

主要的接口如下:

```
int open_table_def(THD *thd, TABLE_SHARE *share, uint db_flags)
```

这个函数打开指定的 frm 文件(由 share->normalized_path.str 指定).读取表定义并填冲 share 结构.

```
TABLE_SHARE *get_table_share(THD *thd, TABLE_LIST *table_list, char *key,
                             uint key_length, uint db_flags, int *error)
```

这个函数还额外维护一个 TABLE_SHARE 的 hash 表.这个函数会先查找 hash 表,如果找不到定义,再调用 open_table_def 读入表定义.

```
void release_table_share(TABLE_SHARE *share, enum release_type type)
```

释放一个 TABLE_SHARE 引用.

```
static TABLE_SHARE
*get_table_share_with_create(THD *thd, TABLE_LIST *table_list,
                             char *key, uint key_length,
                             uint db_flags, int *error)
```

这个函数先调用 get_table_share,如果失败,会调用 ha_create_table_from_engine 创建之.

二,TABLE

TABLE 实例代表一个打开的 TABLE_SHARE,主要成员如下:

名称	类型	说明
s	TABLE_SHARE*	表的定义
file	handler*	存储引擎
field	Field**	同 TABLE_SHARE

名称	类型	说明
used_next,used_prev	TABLE*,TABLE**	这两个成员用于实现 used 关系
open_next,open_prev	TABLE*,TABLE**	这两个成员用于实现 opened 关系
prev,next	TABLE*	THD 中的 open_tables 关系
in_use	THD*	使用这个表的”线程”
pos_in_table_list	TABLE_LIST*	
位图		
quick_keys	key_map	
merge_keys	key_map	
covering_keys	key_map	
keys_in_use_for_query	key_map	
keys_in_use_for_group_by	key_map	
keys_in_use_for_order_by	key_map	
map	table_map	表的位图值(形如 1,2,4,8 等等),在 setup_table_map 中创建.所有表的位图值加到一起形成表位图,很多地方都用到这个位图.

第三节 SQL 语法解析及执行

一,LEX

LEX 是语法分析的主要对象,主要成员如下:

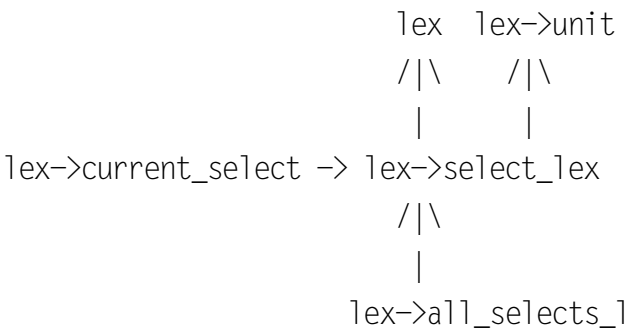
名称	类型	说明
unit	SELECT_LEX_UNIT	最顶层的 SELECT_UNIT,注意这不是一个指针哈.
select_lex	SELECT_LEX	最顶层的 SELECT_LEX

名称	类型	说明
current_select	SELECT_LEX*	当前解析的...
key_list	List<Key>	索引
col_list	List<key_part_spec>	索引分量
alter_tablespace_info	st_alter_tablespace	tablespace 操作相关的信息
sql_command	enum_sql_command	SQL 命令类型
alter_info	ALTER_INFO	解析 ALTER...命令
name	LEX_STRING	对象的名称,在不同的命令下有不同的用途
spname	sp_name*	存储过程/函数名称
server_options	LEX_SERVER_OPTIONS	server 操作相关的信息
part_info	partition_info*	partition 操作相关的信息
query_tables	TABLE_LIST*	TABLE_LIST 的链表(通过 next_global 和 prev_global).
query_tables_last	TABLE_LIST**	指向上面链表的尾巴,初始化为 query_tables 地址.参见 Query_tables_list::reset_query_tables_list 和 add_to_query_tables 函数
proc_list	SQL_LIST	当 select 列表中有 procedure 调用时,参见 mySQL 帮助文件 26.4.1 节以及函数 add_proc_to_list
create_list	List<create_field>	需要创建的表字段定义,参见 add_field_to_list
all_selects_list	SELECT_LEX*	全部的 SELECT_LEX 结点(这些结点通过 link_prev,link_next 连接).参见 SELECT_NODE 定义,mysql_new_select 函数以及 SELECT_LEX::include_global 函数.

LEX 的初始化,当需要解析一个 SQL 语句时,MySQL 调用下面函数设置 LEX 的初始装态

```
void lex_start(THD *thd, const char *buf, uint length)
```

这个函数初始化 thd->lex 结构,建立了如下的对象关系(lex 指 thd->lex):



即:lex->select_lex.parent_lex 指向 lex,lex->select_lex.master 指向 lex->unit,lex->current_select 指向 lex->select_lex,lex->select_lex 在 all_selects_list 链表中.

lex->select_number = 1

lex->next_state=MY_LEX_START

二,SELECT_NODE,SELECT_LEX 和 SELECT_UNIT

SELECT_LEX 和 SELECT_UNIT 都是从 SELECT_NODE 继承过来.SELECT_LEX 用来表示 SELECT 操作符,SELECT_UNIT 用来代表嵌套查询. SELECT_NODE 则提供了基本的关系操作所需要的设施,这 3 个类的重要成员如下:

名称	类型	说明
SELECT_NODE 主要成员		
master,slave	SELECT_LEX_NODE*	嵌套查询(子查询)
prev	SELECT_LEX_NODE**	
next	SELECT_LEX_NODE*	UNION 关系的查询
link_prev	SELECT_LEX_NODE**	
link_next	SELECT_LEX_NODE*	全局关系,参见 LEX::all_selects_list
SELECT_LEX 主要成员		
SELECT_LEX:解析结果相关的成员,这些成员组成 SQL 最终的抽象语法		
table_list	SQL_LIST	TABLE_LIST 的链表(通过 next_local),参见

名称	类型	说明
		SELECT_LEX::add_table_to_list 函数
item_list	List<Item>	所有的字段和表达式,参见 SELECT_LEX::add_item_to_list 函数
with_wild	uint	带有*的字段个数,例如对于: select t1.*,t2* from t1,t2 with_wild 等于 2,对于: select * from t1 等于 1.参见 sql_yacc.yy::6276
top_join_list	List<TABLE_LIST>	JOIN 操作解析树的根结点
where	Item*	where 子句(解析为一个表达式),参见 sql_yacc.yy::7784
having	Item*	having 子句,同上
order_list	SQL_LIST	order 子句,这应该是一个 Item 类型的链表.参见 SELECT_LEX::add_order_to_list 函数
group_list	SQL_LIST	group 子句,这应该是一个 Item 类型的链表.参见 SELECT_LEX::add_group_to_list 函数
select_limit,offset_limit	Item*	limit 子句
explicit_limit	bool	是否使用了显式的 limit 子句
context	Name_resolution_context	
parent_lex	LEX*	父 LEX,在 lex_start 中初始化
select_n_where_fields	uint	所有在 select 和 where 后用到的字段的个数,包括子查询
select_n_having_items	uint	所有在 select 和 having 后用到的字段的个数,包括子查询
SELECT_LEX:解析时成员,下面这些成员是辅助于查询语句解析过程的中间变量		
parsing_place	enum_parsing_plac	用于指示解析上下文,包括:

名称	类型	说明
	e	NO_MATTER,无关紧要 IN_HAVING,having 子句中 SELECT_LIST,字段列表中 IN_WHERE,where 子句中 IN_ON,on 中
join_list	List<TABLE_LIST>*	用于处理 select 操作符表连接操作,参见 TABLE_LIST 对象的相关说明.初始化为指向 top_join_list,参见 SELECT_LEX::init_query.
expr_list	List<List_item>	用于表达式的解析.存放 expr1, expr2, ..., exprn 格式的语法.参见 sql_yacc.yy::6321 等处
embedding	TABLE_LIST*	(语法解析过程中)指向当前的嵌套结点.参见 SELECT_LEX::init_query,在这个函数中初始化为 0; SELECT_LEX::init_nested_join,在这个函数中更新为新创建的 TABLE_IST
index_hints	List<index_hint>	暂存当前表的“索引提示”列表,在一个表解析完成后,这个列表会移动到 TABLE_LIST::index_hints 中.参见 SELECT_LEX::set_index_hint_type, SELECT_LEX::add_index_hint, SELECT_LEX::pop_index_hints, SELECT_LEX::add_table_to_list 等函数
SELECT_LEX:执行时成员,这些成员辅助于查询的执行		
join	JOIN*	对应的 JOIN,参见 JOIN::prepare 函数
ref_pointer_array	Item**	一个 Item*类型的数组.参见 SELECT_LEX::setup_ref_array 函数
SELECT_UNIT 主要成员		
table	TABLE*	用于保存 UNION 操作结果的临时表

名称	类型	说明
item_list	List<Item>	临时表字段
result	select_result*	结果集,用这个成员来返回结果
union_result	select_union*	连接结果集
table	TABLE*	临时表,用于存放连接结果集定义.参见 SELECT_UNIT::prepare
result_table_list	TABLE_LIST	和 table 临时表对应的 TABLE_LIST 结构
item	Item_subselect*	

SELECT 和 UNION 操作的解析:

UNION 表达式 := (SELECT 表达式| UNION 表达式) UNION (SELECT 表达式| UNION 表达式)

SELECT 表达式 := SELECT (SELECT 表达式*)

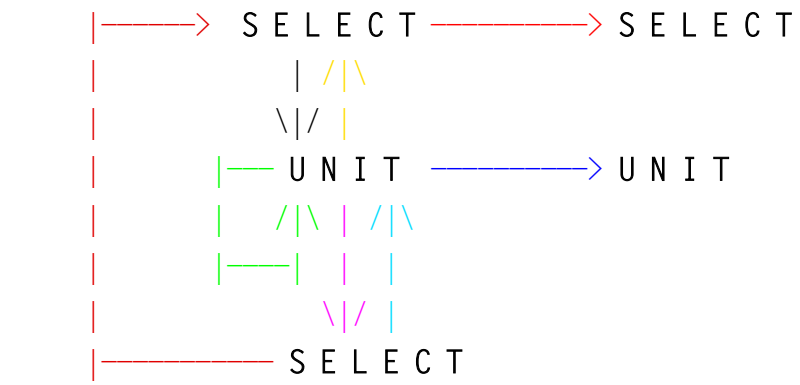
1)MySQL 用 SELECT 结点来代表一个 SELECT 查询,而用一个 UNIT 结点来代表一个 UNION 操作符或者 SELECT 下的子查询

2)LEX::unit 为根结点,LEX::select_lex 指向 SQL 中的第一个 SELECT 语句

当解析器遇到一个 select 操作符时,会调用下面函数:

```
bool mysql_new_select(LEX *lex, bool move_down)
{
      1,select_lex = new SELECT_LEX();//创建一个 SELECT 类型的结点
      2,if(move_down)
        {
            unit = new SELECT_LEX_UNIT();
            lex->subqueries= TRUE;//表明有子查询
            unit->include_down(lex->current_select);
            unit->return_to= lex->current_select;
            select_lex->include_down(unit);
        }
      else
        {   //UNION 操作数
            select_lex->include_neighbour(lex->current_select);
            unit= select_lex->master_unit();
            if(!unit->fake_select_lex)
                  unit->add_fake_select_lex(lex->thd);
```

```
        }
      3,select_lex->include_global(...);//加入到 lex->all_selects_list 全局链表中
      4,lex->current_select= select_lex; //更新当前 SELECT_LEX
}
```

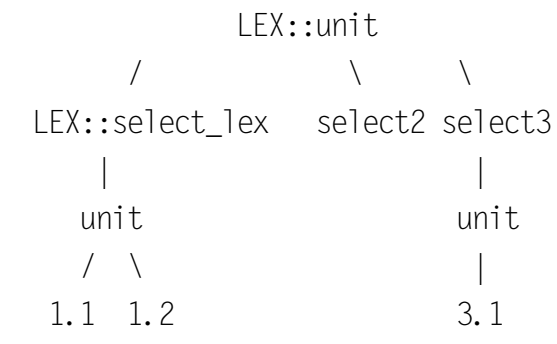


图例:

几个例子:

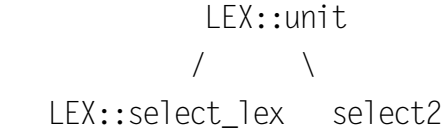
select1 (select1.1 union select1.2) union select2 union select3 (select3.1)

解析为:



select1 (select1.1 () union select1.2 ()) union select2 (select2.1 () select2.2 ())

解析为:



```
      |           |
    unit1        unit2
    /  \         /  \
  1.1  1.2     2.1  2.2
```

从上面的解析树可以看出不同层之间的查询是通过unit来隔离的,unit结点的作用就是隔离不同层次的查询.

三,TABLE_LIST

名称	类型	说明
table	TABLE*	打开的TABLE
next_leaf	TABLE_LIST*	所有的”叶子”TABLE_LIST.
next_global	TABLE_LIST*	
prev_global	TABLE_LIST**	
next_local	TABLE*	参见SELECT_LEX::table_list
index_hints	List<index_hint>	表的索引提示
options	ulonglong	参见JOIN::select_options
下面是和join操作有关的成员		
natural_join	TABLE_LIST*	自然连接表.例如:a NATURAL JOIN b则b->natural_join=a.参见add_join_natural函数.
is_natural_join	bool	是否是自然连接.在nest_last_join结点上设置.
on_expr	Item*	连接表达式,在内表结点上设置.例如:a JOINT b ON (a.col1 = b.col2 and a.col2 = b.col3),则b->on_expr指向ON表达式.
straight	bool	当使用STRAIGHT_JOIN操作符时
outer_join	uint	OUTER JOIN的类型,在内表结点上设置:JOIN_TYPE_LEFT=1,JOIN_TYPE_RIGHT=2.为0时表示是INNER JOIN.例如,

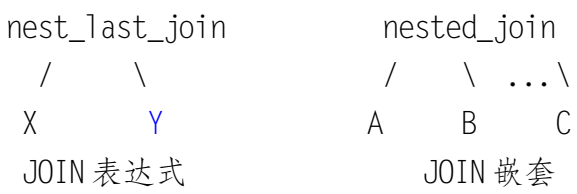
名称	类型	说明
		T1 left join T2 on (...) 这是 T2->outer_join 会置为 JOIN_TYPE_LEFT
join_using_fields	List<String>	USING 子句. 同 on_expr.
nested_join	NESTED_JOIN*	指向嵌套结点.参见稍后的嵌套结点定义
embedding	TABLE_LIST*	所属的嵌套结点.参见稍后的嵌套结点定义
join_list	List<struct st_table_list>	TABLE_LIST 所属的 join list,参见后面说明
视图相关		
view	LEX*	对应视图的语法树
merge_underlying_list	TABLE_LIST*	List (based on next_local) of underlying tables of this view. I.e. it does not include the tables of subqueries used in the view. Is set only for merged views.
referencing_view	TABLE_LIST*	The view directly referencing this table(non-zero only for merged underlying tables of a view).
view_tables	List<st_table_list>*	- 0 for base tables - in case of the view it is the list of all (not only underlying tables but also used in subquery ones) tables of the view.
belong_to_view	TABLE_LIST*	most upper view this table belongs to
where	Item*	VIEW WHERE clause condition
check_option	Item*	WITH CHECK OPTION condition
query	LEX_STRING	text of (CRETE/SELECT) statement

名称	类型	说明
md5	LEX_STRING	md5 of query text
scource	LEX_STRING	source of CREATE VIEW
view_db	LEX_STRING	saved view database
view_name	LEX_STRING	saved view name
timestamp	LEX_STRING	GMT time stamp of last operation
timestamp_buffer[20]	char	
definer	st_lex_user	definer of view
file_version	ulonglong	version of file's field set
updatable_view	ulonglong	VIEW can be update
revision	ulonglong	revision control number
algorithm	ulonglong	0 any, 1 tmp tables , 2 merging
view_suid	ulonglong	view is suid (TRUE dy default)
with_check	ulonglong	WITH CHECK OPTION
位图		
dep_tables	table_map	
on_expr_dep_tables	table_map	

JOIN操作符具有下面的形式

JOIN表达式 := (JOIN表达式|JOIN嵌套) JOIN (JOIN表达式|JOIN嵌套)
JOIN嵌套 := '(' (表引用|JOIN表达式) (,表引用|,JOIN表达式)* ')' | 表引用

即,所有的JOIN表达式都可以用JOIN表达式和JOIN嵌套这两个基本结点来代表:

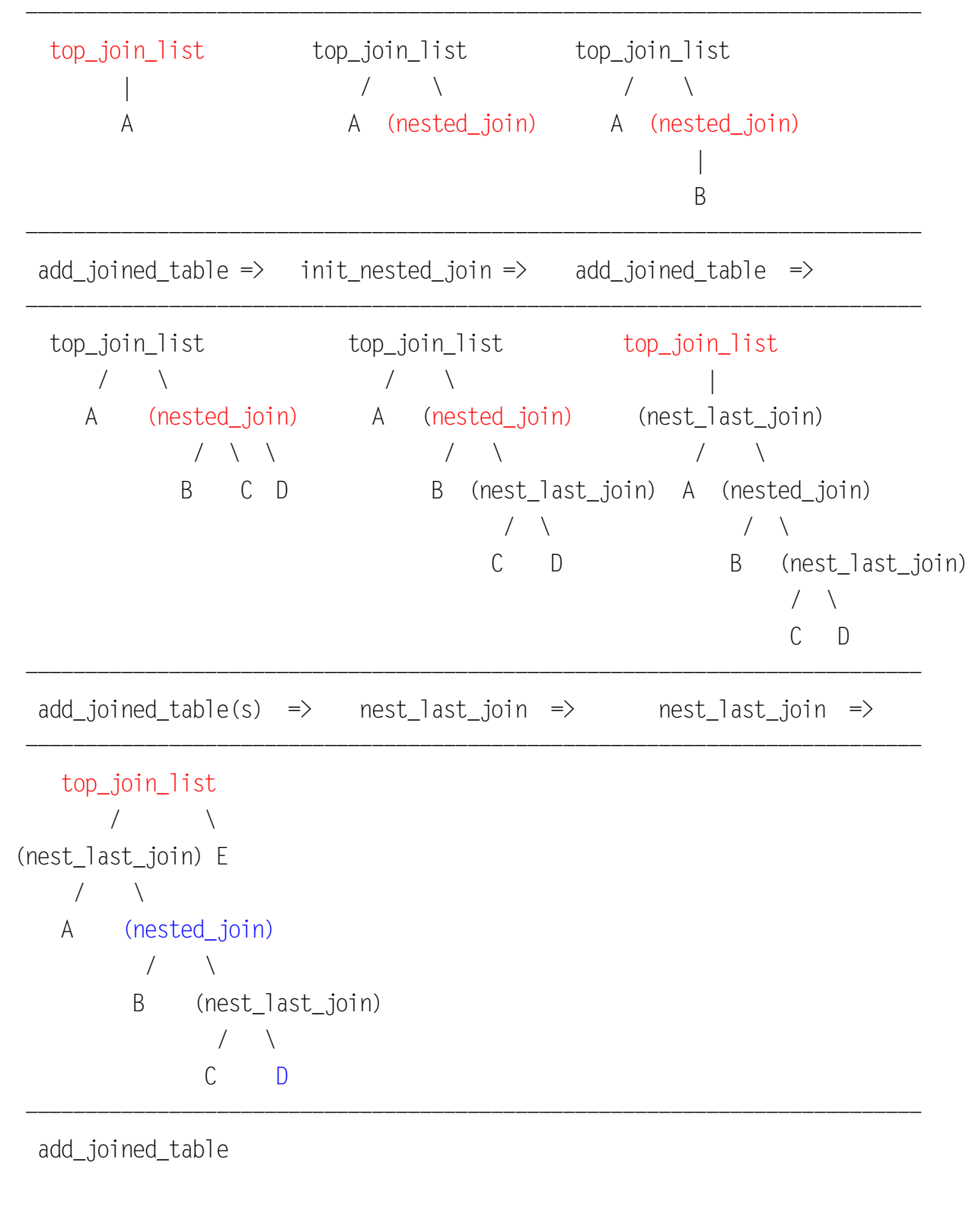


X,Y 结点可以是:JOIN 表达式,JOIN 嵌套或表引用之一,而 ON (...)表达式一定在 Y 结点上.同时当 Y 为表引用时,也称之为内表(inner table).
nest_last_join 和 nested_join 都称之为”**嵌套结点**”

下面是一些具体的例子说明解析树的形成过程,可以看出最终结果都具有上面所述的形式(图中红色结点代表 SELECT_LEX::join_list 所指):
例一,A JOIN B on (...) JOIN C on (...), D

注 1:on 表达式在 B 和 C 结点上
注 2:如果 A,B 之间是”自然”连接,即,使用 NATURAL 修饰符或者用 USING 子句,则 B->natural_join 指向 A,并且 nest_last_join 的 is_natural_join 成员为真

例二,A JOIN (B,C JOIN D) on (...) , E



主要接口

```
TABLE_LIST *st_select_lex::add_table_to_list(THD *thd,
                                             Table_ident *table,
                                             LEX_STRING *alias,
                                             ulong table_options,
                                             thr_lock_type lock_type,
                                             List<index_hint> *index_hints_arg,
                                                    LEX_STRING *option)
```

这个函数新建一个 TABLE_LIST 实例,并添加到两个链表:query_tables_last 和 table_list

四,NESTED_JOIN

名称	类型	说明
join_list	List<TABLE_LIST>	嵌套结点
used_tables	table_map	嵌套用到的表的位图.参见前图,假设 A-E 表的编号依次是 1,2,4,8,16,则,(nested_join)->nested_join->used_tables = 14,(nest_last_join)->nested_join->used_tables=15.参见 simplify_joins 函数
not_null_tables	table_map	
nj_map	nested_join_map	嵌套结点的编号.参见前图:(nested_join),(nest_last_join)就属于嵌套结点.参见 build_bitmap_for_nested_joins 函数

四,Item/COND

Item 是 MySQL 表达式的核心.主要成员如下:

名称	类型	说明
Type	enum Type	Item 类型,例如,FIELD_ITEM,STRING_ITEM,

名称	类型	说明
		INT_ITEM 等
cond_result	enum cond_result	
traverse_order	enum traverse_order	
name	my_string	名称
next	Item*	

Item 提供了一个重要的函数 fix_fields,这个函数”编译”一个 Item.不同的 Item 的子类需要实现自己的”编译”方法.最终都直接或间接的转化为对数据库字段(Field)的引用,这是在 Item_field 类中实现的,参见下面函数:

```
bool Item_field::fix_fields(THD *thd, Item **reference)
```

举个例子,MySQL 中所有的函数都是用 Item_func 类来表示的,这个类使用 Item **args 和 uint arg_count 来代表函数参数,例如 ifnull(a.col , '#NULL')中,a.col 和'#NULL'就是 Item_func 的两个参数.Item_func 的 fix_fields 函数就实现为调用每个参数的 fix_fields 函数:

```
for (arg=args, arg_end=args+arg_count; arg != arg_end ; arg++)
{
      if ((!(*arg)->fixed && (*arg)->fix_fields(thd, arg)))
}
```

在这个例子中,a.col 应该是一个 Item_field 实例,'#NULL'应该是一个 Item_string 实例.前者最终”编译”为对 Field 的引用.

五,JOIN

名称	类型	说明
::prepare 阶段涉及的字段		
tables_list	TABLE_LIST*	查询的涉及的表
conds	COND*	where 子句
order	ORDER*	order 子句
group_list	ORDER*	group 子句

名称	类型	说明
having	Item*	having 子句
select_lex	SELECT_LEX*	对应的 SELECT_LEX
proc_param	ORDER*	procedure 的参数
join_list	List<TABLE_LIST>*	初始化为指向 SELECT_LEX::top_join_list
JOIN 在 prepare 后和其它对象的关系: select_result <— JOIN —> THD |——————> SELECT_LEX		
::optimize 阶段涉及的字段		
cond_value	Item::cond_result	调用 optimize_cond 对 conds 进行处理后的值.如果这个值为 Item::COND_FALSE,查询肯定不会有结果集返回,MySQL 会设置 zero_result_cause 为”Impossible WHERE”.
having_value	Item::cond_result	同上,对 having 进行处理后的值. 如果这个值为 Item::COND_FALSE,查询肯定不会有结果集返回,MySQL 会设置 zero_result_cause 为”Impossible HAVING”
zero_result_cause	const char*	查询返回 0 行的原因.参见 JOIN::optimize 函数.
JOIN::JOIN		
select_options	ulonglong	在构造 JOIN 时初始化.在二次执行时可以修改.这个值会控制查询的很多行为.这个值主要来自: SELECT_NODE::options 和 THD::options 参见 mysql_select 函数.以及 mysql_priv.h::298 处,这里有所有可用的控制选项的列表.
select_distinct	bool	当 select_options 中有 SELECT_DISTINCT

名称	类型	说明
		时
result	select_result	结果集
fields_list	List<Item>&	存放进行 setup_wild 处理前,后的字段
all_fields	List<Item>	
JOIN 在初始化后和其它对象的关系: select_result <— JOIN —> THD		
其它		
conds_history	Item*	当 LEX::describe 字段含有 DESCRIBE_EXTENDED 时,这两个字段分别用来暂存 conds 和 having.参见 mysql_select 函数.
having_history	Item*	
tables	uint	”叶子”结点的个数,参见 JOIN::prepare
join_tab	JOIN_TAB*	JOIN_TAB 数组,为 tables 大小,参见 make_join_statistics 函数
位图		
const_table_map	table_map	
found_const_table_map	table_map	
outer_join	table_map	所有外连接的表,参见 make_join_statistics

六,JOIN_TAB

名称	类型	说明
dependent	table_map	
key_dependent	table_map	
embedding_map	nested_join_map	连接表所属的嵌套结点.由 NESTED_JOIN::nj_map 合成.参见

名称	类型	说明
		make_join_statistics 函数.
on_expr_ref	Item**	JOIN 操作符的 ON 子句

七,THD

名称	类型	说明
temporary_tables		
options	ulonglong	参见 JOIN::select_options

第二章 SQL 的主要执行流程

第一节 连接操作

一个 SQL 语句经过解析后,最终形成的就是一个需要查询的基表的队列,这个队列的顺序就是连接的顺序.连接操作就是依次查询这些基表,并对符合连接条件的行进行连接,即:

```
nest_loop( tab )
{
      for rowi in tab
            for rowj in tab+1
                  if( rowi match rowj )
                  {
                        result = rowi join rowj;
                        if( tab + 1 is the most inner table )
                              return result;
                        else
                              nest_loop ( tab + 1 );
                  }
}
```

一,主入口

```
        (1)     |-> mysql_union -> SELECT_UNIT::prepare -> SELECT_UNIT::exec
                |                       |                       |
handle_select -> |                      |              |------|
                |                     \|/            \|/
        (2)     |-> mysql_select -> JOIN::preapre -> JOIN::optimize ->JOIN::exec
```

SQL 的执行有两条主要的调用路径,一个是带 UNION 操作情况下的执行路径,大致如上面(1)所示,一个是单个 SELECT 的执行路径,大致如上面(2)所示.

```
bool mysql_select(THD *thd,
          Item ***rref_pointer_array,
```

```
        TABLE_LIST *tables, //查询涉及的表,SELECT_LEX::table_list
        uint wild_num, // * 字段的个数,SELECT_LEX::with_wild
        List<Item> &fields,//SELECT_LEX::item_list
        COND *conds, //where 子句,SELECT_LEX::where
        uint og_num,  //SELECT_LEX::(order_list.elements+group_list.elements)
        ORDER *order, //order 子句,SELECT_LEX::order_list
        ORDER *group,//group 子句,SELECT_LEX::group_list
        Item *having, //having 子句,SELECT_LEX::having
        ORDER *proc_param, //procedure 操作符参数,SELECT_LEX::proc_list
        ulonglong select_options,
        select_result *result, //结果集
        SELECT_LEX_UNIT *unit,
        SELECT_LEX *select_lex)//和 JOIN 建立关联
{
    //这个函数的主要逻辑如下
    JOIN *join;
    //准备
    if (select_lex->join != 0)
    {
        //二次执行
        join= select_lex->join;
        join->change_result(result)
        //或者
        join->prepare(rref_pointer_array, tables, wild_num,
                      conds, og_num, order, group, having, proc_param,
                      select_lex, unit)
    }else
    {
        join= new JOIN(thd, fields, select_options, result);
        join->prepare(rref_pointer_array, tables, wild_num,
                    conds, og_num, order, group, having, proc_param,
                    select_lex, unit)
    }
    ...
```

```
        //优化
        join->optimize();
        //执行
        join->exec();
        ...
}
```

二,准备

```
int
JOIN::prepare(Item ***rref_pointer_array,
              TABLE_LIST *tables_init,//查询的涉及的表
              uint wild_num,// * 字段个数
              COND *conds_init,//where 子句
              uint og_num,//SELECT_LEX::(order_list.elements+group_list.elements)
              ORDER *order_init, //order 子句
              ORDER *group_init, //group 子句
              Item *having_init, //having 子句
              ORDER *proc_param_init, //procedure 参数
              SELECT_LEX *select_lex_arg,//和 JOIN 建立关联
              SELECT_LEX_UNIT *unit_arg)
{
        //主要逻辑
        if(!(select_options & OPTION_SETUP_TABLES_DONE))
                setup_tables_and_check_access(...);
        setup_wild(...);//展开*字段
        select_lex->setup_ref_array(...);//分配 ref_pointer_array 数组
        setup_fields(thd, (*rref_pointer_array), fields_list, MARK_COLUMNS_READ,
                 &all_fields, 1);//这个函数初始化 ref_pointer_array 数组:数组的前
                                 //fields_list.elements 个元素
        setup_without_group(...);
                setup_conds(...);//
                setup_order(...);//这个函数查找 order 子句中引用的字段,如果必要(即,在
                                 //查询的字段列表中不存在)加入到查询字段列表中
                setup_group(...);
        if (having) {...};
```

```
        if (order) {...};
        setup_ftfuncs(...);
        setup_procedure(...);
        result->prepare(...);
        rollup_init();
        alloc_func_list();
}
bool st_select_lex_unit::prepare(THD *thd_arg, select_result *sel_result,
                                 ulong additional_options)
{
        ...
        SELECT_LEX *first_sl= first_select();
        ...
        sl =  first_sl;
        ...
        if (is_union)
        {       //创建连接结果集
                union_result= new select_union;
        }
        //为一个 UNIT 内的查询准备 JOIN 对象
        for (;sl; sl= sl->next_select())
        {
                ...
                JOIN *join= new JOIN(thd_arg, sl->item_list,
                        sl->options | thd_arg->options | additional_options,
                        tmp_result);
                ...
                saved_error= join->prepare(...);
        }
        if (is_union)
        {
                //union_result 创建中间表
                union_result->create_result_table(thd, &types, test(union_distinct),
                                                  create_options, "")
```

```
            result_table_list.db= (char*) "";
            result_table_list.table_name= result_table_list.alias=
                        (char*) "union";
            result_table_list.table= table= union_result->table;
            ...
            fake_select_lex->join= new JOIN(thd, item_list, thd->options,
                                    result);
            fake_select_lex->join->prepare(...);
      }
}
```

三,优化

这个阶段包括以下任务:

- ✔ 简化连接

  可能的外连接到内连接的转化

  内连接的平坦化:将结点的 on_expr 合并到 where 条件中并将 on_expr 置空(这时内连接的解析树变得和冗余嵌套的解析树一样).

  冗余嵌套的消解
- ✔ where 和 having 条件优化
- ✔ 收集统计数据

```
int JOIN::optimize()
{
      ...
      if (sel->first_cond_optimization)//第一次进行条件优化
      {
            ...
            sel->first_cond_optimization= 0;
            ...
            conds= simplify_joins(this, join_list, conds, TRUE);
            //为每一个嵌套结点编号
            build_bitmap_for_nested_joins(join_list, 0);
            ...
      }
      //优化 where 条件
```

```
        conds= optimize_cond(this, conds, join_list, &cond_value);
        //优化 having 条件
        having= optimize_cond(this, having, join_list, &having_value);
        //如果现在就能判断出 where 或者 having 恒假(例如 where 1=2),则这个查询根本就
不用执行了
        if (cond_value == Item::COND_FALSE || having_value == Item::COND_FALSE ||
          (!unit->select_limit_cnt && !(select_options & OPTION_FOUND_ROWS)))
        {
              return;
        }
        ...
        make_join_statistics(this, select_lex->leaf_tables, conds, &keyuse);
        ...
        select= make_select(*table, const_table_map,
                        const_table_map, conds, 1, &error);
        reset_nj_counters(join_list);
        make_outerjoin_info(this);
        make_join_select(this, select, conds);
}
```

1)简化连接

假设需要处理的连接为:A JOIN (B , (C,D,E)) on (...),则对应的解析树为:

```
        top_join_list
              |
        (nest_last_join)
           /     \
          A     (nested_join1)
                   /   \
                  B   (nested_join2)
                         /  \  \
                        C    D  E
```

```
static COND *
simplify_joins(JOIN *join, List<TABLE_LIST> *join_list, COND *conds, bool top)
{
```

```
...
//从 top_join_list 开始历遍连接树:
//1,计算每个嵌套结点的 used_tables 值
//2,计算每个结点的 dep_tables
while ((table= li++))//li 为 join_list 的历遍器
{
      if ((nested_join= table->nested_join))//嵌套结点
      {
            if (table->on_expr) //例如(nested_join1)
            {
                  expr= simplify_joins(join, &nested_join->join_list,
                        expr, FALSE);
            }
            conds= simplify_joins(join, &nested_join->join_list,
                                  conds, top);
      }else//叶子结点
      {
            if (!table->prep_on_expr)
                  table->prep_on_expr= table->on_expr;
            used_tables= table->table->map;
            if (conds)
                  not_null_tables= conds->not_null_tables();
      }
      //将满足条件的外连接转化为内连接
      //例如:select a.col, b.col from a left join b on (a.col1 = b.col1 )
      //     where b.col < 10
      //这时这个外连接就可以转化,因为假设 a 中出现了不匹配 b 的行,在外连接下
      //b.col 将为空,而 b.col<10 这个条件将过滤掉这个连接行,实际查询出来的结
      //果和内连接下是一样的,因此,这样的外连接就可以转化为内连接.如果把
      //where 条件换成诸如: b.col is null 或 nvl( b.col, 'DD') = ...之类不能
      //使条件恒为假的条件表达式时,这个外连接就不会转化.
      //对内连接进行 on_expr 表达式合并
      if (!table->outer_join || (used_tables & not_null_tables))
      {
```

```
        table->outer_join= 0;
        if (table->on_expr)
        {
              //将 on_expr 并入 conds
              if (conds)
              {
                    conds= and_conds(conds, table->on_expr);
                    ...
              }else
                    conds= table->on_expr;
              //有可能在后面将 table 平坦化了
              table->prep_on_expr= table->on_expr= 0;
        }
  }
  if (!top)continue;
  //Only inner tables of non-convertible outer joins
  //remain with on_expr.
  if (table->on_expr)
  {
        //dep_tables 就是 on_expr 中用到的表
        //例如: a join b on ( a.col = b.col )
        //则 b->dep_tables 将包含 a
        table->dep_tables|= table->on_expr->used_tables();
        if (table->embedding)//这是一个连接嵌套,例如,(A,B)
        {
              //同一个嵌套结点下的表(含自己)不相互依赖
              table->dep_tables&=
                  ~table->embedding->nested_join->used_tables;
              //累计到嵌套结点中
              table->embedding->on_expr_dep_tables|=
                  table->on_expr->used_tables();
        }else
              //自己不算
              table->dep_tables&= ~table->table->map;
```

```
        }
        if (prev_table)
        {
        //The order of tables is reverse: prev_table follows table
        //意思应该是 a join b 中,b 是 prev_table
              if (prev_table->straight)//STRAIGHT_JOIN
                    prev_table->dep_tables|= used_tables;
              if (prev_table->on_expr)
              {
                    prev_table->dep_tables|= table->on_expr_dep_tables;
                    table_map prev_used_tables= prev_table->nested_join ?
                              prev_table->nested_join->used_tables :
                              prev_table->table->map;
                    if (!(prev_table->on_expr->used_tables() &
                          ~prev_used_tables))
                          prev_table->dep_tables|= used_tables;
              }
        }
        prev_table= table;
  }
  结果如下:
  top_join_list
        |
  (nest_last_join) -> (A,B,C,D,E)
     /     \
    A     (nested_join1) -> (B,C,D,E)
            /   \
           B   (nested_join2) -> (C,D,E)
                  /  \  \
                 C    D  E
  蓝色箭头代表 used_tables
  //平坦化一些不必要的嵌套连接
  //情形一,假设(nested_join1)和 A 之间是不可转化的外连接
  //上面解析树中(nested_join2)结点就属于可以平坦化的,这个结点和(nested_join1)
```

的区别在于前者的 on_expr 成员为空,平坦化后的结果如下:

```
      top_join_list
            |
      (nest_last_join)
         /     \
        A      (nested_join1)
                 /  \  \  \
                B    C  D  E
```

等价于 A JOIN (B,C,D,E) on (...)

//情形二,假设(nested_join1)和 A 之间是内连接,则(nested_join1)->on_expr 在前面已经被置空,因此,(nested_join1)也是可以平坦化的,结果如下:

```
       top_join_list
            |
      (nest_last_join)
         /   \  \  \  \
        A     B  C  D  E
```

等价于 A,B,C,D,E where (...)

}

2)条件优化

```
static COND *
optimize_cond(JOIN *join, COND *conds, List<TABLE_LIST> *join_list,
              Item::cond_result *cond_value)
{
      if (!conds)
            *cond_value= Item::COND_TRUE;//没条件?恒真
      else
      {
            ...
      }
}
```

3)收集统计数据

```
      top_join_list
         /        \
 (nest_last_join)  F
      /    \
     A    (nested_join)
            /  \     \
           B    C  (nest_last_join)
                     /   \
                    D     E
A JOIN ( B , C , D JOIN E ) , F
```

假设下面的make_join_statistics以上面这棵解析树为操作对象

```
static bool
make_join_statistics(JOIN *join, TABLE_LIST *tables, COND *conds,
                     DYNAMIC_ARRAY *keyuse_array)
{
      ...
      //连接表个数
      table_count=join->tables;
      //连接操作所需数据
      stat=(JOIN_TAB*) join->thd->calloc(sizeof(JOIN_TAB)*table_count);
      stat_ref=(JOIN_TAB**) join->thd->alloc(sizeof(JOIN_TAB*)*MAX_TABLES);
      table_vector=(TABLE**) join->thd->alloc(sizeof(TABLE*)*(table_count*2));
      ...
      for (s= stat, i= 0; tables; s++, tables= tables->next_leaf, i++)
      {
            TABLE_LIST *embedding= tables->embedding;
            ...
            s->dependent= tables->dep_tables;//外连接涉及的表
            s->key_dependent= 0;
            ...
            s->on_expr_ref= &tables->on_expr;
            if (*s->on_expr_ref)//如E
            {
```

```
            if ((!table->file->stats.records||table->no_partitions_used) &&
                 !embedding)//!embedding && *s->on_expr_ref 的结点?可能吗
            {
                  s->dependent= 0;
                  set_position(join,const_count++,s,(KEYUSE*) 0);
                  continue;
            }
            ...
            outer_join|= table->map;//table_map
            //计算表所属的嵌套结点:所有的,包括直接和间接所属
            //参见 NESTED_JOIN::embedding_map 说明
            continue;
      }
      if (embedding)//如 A,B,C,D
      {
            //计算表所属的嵌套结点:所有的,包括直接和间接所属
            //参见 NESTED_JOIN::embedding_map 说明
            //计算 outer_join
            continue;
      }
      if ((table->s->system ||
                  table->file->stats.records <= 1 ||no_partitions_used) &&
           !s->dependent &&
           (table->file->ha_table_flags() & HA_STATS_RECORDS_IS_EXACT) &&
           !table->fulltext_searched)
      {
            set_position(join,const_count++,s,(KEYUSE*) 0);
      }
}
stat_vector[i]=0;
join->outer_join=outer_join;//所有的外连接表
if (join->outer_join)
{
      //
```

```
        }
        ...
        join->join_tab=stat;
        join->map2table=stat_ref;
        join->table= join->all_tables=table_vector;
        join->const_tables=const_count;
        join->found_const_table_map=found_const_table_map;
        ...
}
```

四,执行

1)单个 SELECT 的执行

```
void
JOIN::exec()
{
        //没有表的查询
        if (!tables_list && (tables || !select_lex->with_sum_func))
        {
                ...
                return;
        }
        JOIN *curr_join = this;
        List<Item> *curr_all_fields= &all_fields;
        List<Item> *curr_fields_list= &fields_list;
        TABLE *curr_tmp_table = 0;
        //
        if (select_options & SELECT_DESCRIBE)
        {
                ...
                return;
        }
        if (need_tmp)//注 1
        {
```

```
            if (tmp_join) { curr_join = tmp_join; }
            curr_tmp_table= exec_tmp_table1;
            //Copying to tmp table
      }
      ...
      if (curr_join->group_list || curr_join->order)
      {
            //Sorting result
      }
      if (is_top_level_join() && thd->cursor && tables != const_tables)
      {
      }
      else
      {
            //Sending data
            do_select(curr_join, curr_fields_list, NULL, procedure);
      }
}
```

2) 多个 SELECT(UNION)的执行

指一个 UNIT 结点下的 SELECT 结点的执行,参见下图,查询总是从最顶层的 UNIT 开始,第二及以后层的 UNIT 结点代表的一定是子查询,子查询在子查询表达式求解的过程中执行,参见后面子查询的说明.

```
start from here  -> SELECT_LEX::unit
                      /    \     \
                   sel1   sel2   sel3
                                   |
                                  unit    -> Item_subselect
                                  /   \
                              sel3.1 sel3.2
bool st_select_lex_unit::exec()
{
      //执行一个 UNIT,例如 sel1,sel2,sel3 或者 sel3.1,sel3.2
```

```
        for (SELECT_LEX *sl= select_cursor; sl; sl= sl->next_select())
        {
              sl->join->optimize();
              sl->join->exec();
        }
        //执行 fake_select
        init_prepare_fake_select_lex(thd);
        JOIN *join= fake_select_lex->join;
        if (!join)
        {
              fake_select_lex->join= new JOIN(thd, item_list,
                                    fake_select_lex->options, result);
        }
        mysql_select(thd, &fake_select_lex->ref_pointer_array,
                              &result_table_list,
                              0, item_list, NULL,
                              global_parameters->order_list.elements,
                              (ORDER*)global_parameters->order_list.first,
                              (ORDER*) NULL, NULL, (ORDER*) NULL,
                              fake_select_lex->options | SELECT_NO_UNLOCK,
                              result, this, fake_select_lex);

}
```

注 1:

need_map 在 JOIN::optimize 中估值,逻辑如下:

```
need_tmp= (const_tables != tables &&
           ((select_distinct || !simple_order || !simple_group) ||
            (group_list && order) ||
            test(select_options & OPTION_BUFFER_RESULT)));
```

五,子查询

MySQL 中的子查询实现为函数,这些函数都是从 Item 继承过来,参见下面的关系图:

类	说明
Item	
|__Item_result_field	
|__Item_subselect	
|__Item_singlerow_subselect	如 where a=(select ...)
| |__Item_maxmin_subselect	
|__Item_exists_subselect	如 exists (select ...)
|__Item_in_subselect	如 in (select ...)
|__Item_allany_subselect	
subselect_engine 负责驱动查询的执行	
Sql_alloc	
|__subselect_engine	
|__subselect_single_select_engine	单个的子查询
|__subselect_union_engine	UNION 的子查询
|__subselect_uniquesubquery_engine	
|__subselect_indexsubquery_engine	

```
void Item_subselect::init(st_select_lex *select_lex,
                          select_subselect *result)
{
      //这个函数建立起 Item 和其驱动的子查询(UNIT)之间的关连
      ...
      unit= select_lex->master_unit();
      ...
      if (unit->item)
      {
            unit->item= this;
      }else
      {
```

```
            SELECT_LEX *outer_select= unit->outer_select();
            ...
            //下面的 engine 构造函数中会建立 UNIT 和 Item 关连
            if (select_lex->next_select())
                  engine= new subselect_union_engine(unit, result, this);
            else
                  engine= new subselect_single_select_engine(select_lex,
                        result, this);
      }
}
```

这个函数初始化一个子查询表达式,表达式持有到子查询 select_lex 的引用,因此,在表达式的求解过程中,该子查询就自动的执行了,即:

编译 -> 准备

::fix_fields -> engine->prepare -> JOIN::prepare or SELECT_UNIT::prepare

求值 -> 执行

Item_subselect::val_* -> Item_subselect::exec -> engine->exec -> JOIN::exec or SELECT_UNIT::exec

第二节 结果集