rewrite_main.cc----Rewriter::dispatchOnLex

SQL语句首先被拆分为LEX, query_parse(a.getdatabasename(),query) 生成query_parse指针，存储lex对象

```
std::unique_ptr<query_parse> p;
LEX *const lex = p->lex();
LOG(cdb_v) << "pre-analyze " << *lex;
```

分dml，ddl和不重写三种类型，Dispatcher根据lex的结构，为SQL语句分配handler，不同类型的SQL语句有不同的handler，用来处理sql语句的加密。加密以后的结果则放置在executor中.

```
if (noRewrite(*lex)) {}
else if (dml_dispatcher->canDo(lex)) {}
else if (ddl_dispatcher->canDo(lex)){}
else{}


 switch (lex.sql_command) {
    case SQLCOM_SHOW_DATABASES:
    case SQLCOM_SET_OPTION:
    case SQLCOM_BEGIN:
    case SQLCOM_ROLLBACK:
    case SQLCOM_COMMIT:
    case SQLCOM_SHOW_TABLES:
    case SQLCOM_SHOW_VARIABLES:
    case SQLCOM_UNLOCK_TABLES:


else if (ddl_dispatcher->canDo(lex)) {
        const SQLHandler &handler = ddl_dispatcher->dispatch(lex);
        LEX *const out_lex = handler.transformLex(a, lex, ps);
        ...


LEX *DDLHandler::transformLex(Analysis &a, LEX *lex,const ProxyState &ps) const
{return this->rewriteAndUpdate(a, lex, ps);}
```

ddl系列类型的典型处理流程包含了两部分，一是对SQL语句进行加密，二是以delta类来基于

数据库的变化，这个delta在next阶段会写入到本地的embedded数据库中。例如CREATE TABLE语句，delta需要记录添加的表的名字， 表有多少列，每列分别采用什么样的加密算法。这些功能全都实现在rewriteAndUpdate函数中了。

ddl_handler.cc----rewriteAndUpdate

不同的handlerrewriteandupdate还不一样。

首先是获取数据库和表的名称，之后对参数lex进行浅复制得到new_lex

```
const std::string db_name =lex->select_lex.table_list.first->db;
TEST_DatabaseDiscrepancy(db_name, a.getDatabaseName());

const std::string table =lex->select_lex.table_list.first->table_name;

LEX *const new_lex = copyWithTHD(lex);
```

如果表不存在，就新建一个TableMeta，之后对表名和列名进行重写

```
// Create the table regardless of 'IF NOT EXISTS' if the table
// doesn't exist.
if (false == a.tableMetaExists(db_name, table)) {
    // TODO: Use appropriate values for has_sensitive and has_salt.
    std::unique_ptr<TableMeta> tm(new TableMeta(true, true));
    assert(1 == new_lex->select_lex.table_list.elements);

    TABLE_LIST *const tbl =
        rewrite_table_list(new_lex->select_lex.table_list.first,tm-
>getAnonTableName());
    new_lex->select_lex.table_list = *oneElemListWithTHD<TABLE_LIST>(tbl);

    auto it = List_iterator<Create_field>(lex->alter_info.create_list);
    new_lex->alter_info.create_list =accumList<Create_field>
        (it,[&a, &ps, &tm] (List<Create_field> out_list, Create_field *const
cf)
          {
          return createAndRewriteField(a, ps, cf, tm.get(), true, out_list);
          }
        );
```

最后这一块就有点复杂了，accumList函数和createAndRewriteField函数

lex_util.hh-----accumList

```
template <typename Type> List<Type>
accumList(List_iterator<Type> it,
          std::function<List<Type>(List<Type>, Type *)> op)
{
    List<Type> accum;

    for (Type *element = it++; element ; element = it++) {
        accum = op(accum, element);
    }

    return accum;
}
```

这个函数看上去是一个迭代的过程，不段重复。结合函数调用，应该就是对表中的每列调用createAndRewriteField。有个counter，每次加一，记录每一列的顺序，传递到新建的FieldMeta结构中，这样一个TableMeta下的FieldMeta就可以根据这个counter的值进行排序了。

rewrite_util.cc-------createAndRewriteField

到这里就从TableMeta进展到了FieldMeta，这个函数都是对具体的列进行重写

```
List<Create_field>
createAndRewriteField(Analysis &a, const ProxyState &ps,
                      Create_field * const cf,
                      TableMeta *const tm, bool new_table,
                      List<Create_field> &rewritten_cfield_list)
{
    const std::string name = std::string(cf->field_name);
    auto buildFieldMeta =
        [] (const std::string name, Create_field * const cf,
            const ProxyState &ps, TableMeta *const tm)
    {
        return new FieldMeta(name, cf, ps.getMasterKey().get(),
                             ps.defaultSecurityRating(),
                             tm->leaseIncUniq());
    };
    std::unique_ptr<FieldMeta> fm(buildFieldMeta(name, cf, ps, tm));
```

```
    // ----------------------------
    //         Rewrite FIELD
    // ----------------------------
    const auto new_fields = rewrite_create_field(fm.get(), cf, a);
    rewritten_cfield_list.concat(vectorToListWithTHD(new_fields));

    // ----------------------------
    //         Update FIELD
    // ----------------------------

    // Here we store the key name for the first time. It will be applied
    // after the Delta is read out of the database.
    if (true == new_table) {
        tm->addChild(IdentityMetaKey(name), std::move(fm));
    } else {
        a.deltas.push_back(std::unique_ptr<Delta>(
                               new CreateDelta(std::move(fm), *tm,
                                               IdentityMetaKey(name))));
        a.deltas.push_back(std::unique_ptr<Delta>(
               new ReplaceDelta(*tm,
                               a.getDatabaseMeta(a.getDatabaseName()))));
    }

    return rewritten_cfield_list;
}
```

修改思路：针对create，insert，alter等SQL语句

lex结构中增加一个变量（flag），用来标识是否需要加密；

在accumList函数调用op前，检查flag位的取值，如过是1，则进行op，也就是重写，否则明文