2021.3.27 11:39 PM更新

1. 我找到了位于rewrite\_ds.hh的EncSet类，它能分析出一个元素支持的所有洋葱加密方式，比如支持det洋葱等，我们可以加入两个泛化的EncSet，让它只支持frequncy hiding det洋葱 / 明文洋葱 和 OPE的洋葱。因此可以在securitylevel和对应的onion枚举类中加入自定义的fh变量。

2. Onion头文件中还定义了onion layout，初步观察分析如下：

对于每个不同的数据类型而言，**相同名称的洋葱可能实现方法也不同。**

例如，对于数字类型的DET的洋葱，它可以支持DETJOIN，DET和RND，但是对于str类型的DET的洋葱却不支持DETJOIN。该layout类实现了对于**不同类型数据的各大类洋葱的实现方式【也就是同名洋葱在不同数据类型下的具体实现】。**

3. SECLEVEL是security level，它是对洋葱层的抽象，每个洋葱层代表的加密层级由这个枚举类定义。**这个东西被包装在EncLayer类中**。

相应地便可以在EncSet中添加我们自定义的FH的encset和onionlayout类了。

而每个列的洋葱的洋葱分布会有一个叫做**FieldMeta::determinOnionLayout(见schema.hh)的函数确定。**

static onionlayout FH\_NUM\_ONION\_LAYOUT = {

{oDET, std::vector<SECLEVEL>({SECLEVEL::FH\_DET, SECLEVEL::RND)}, <- 对频率泛化的数字类型的DET洋葱层，应该是由泛化的DET加密和随机加密组成的。

{oOPE, std::vector<SECLEVEL>({SECLEVEL::FH\_OPE, SECLEVEL::RND})}, 同理

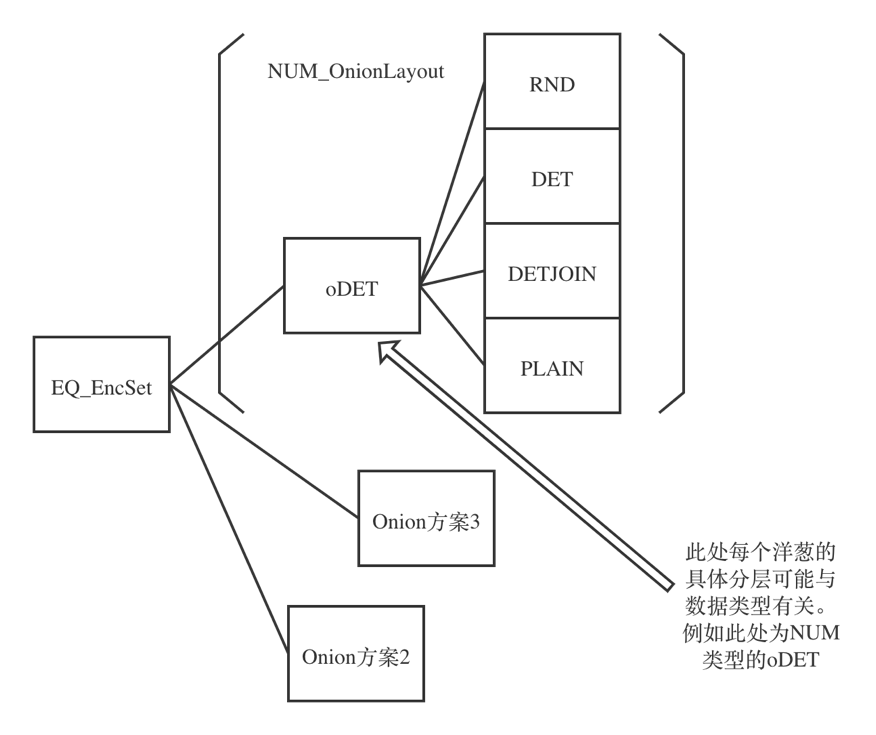
};

static onionlayout FH\_STR\_ONION\_LAYOUT = {

{oDET, std::vector<SECLEVEL>({SECLEVEL::FH\_DET, SECLEVEL::RND})},

{oOPE, std::vector<SECLEVEL>({SECLEVEL::FH\_OPE, SECLEVEL::RND})},

};



**4. 为实现Frequency Smoothing方法，如何在列生成的时候指定其为FH类型的列是一个重点。我的解决方案是：**

在列名前加上前缀“FH\_"

然后onionlayout就可以选择了。

逻辑：

1. 发现是FH类型的；

2. 到选取相应的onionlayout；

3. 在重写的时候回选择一个encset（fh的），作为洋葱；

4. 而我们又对每个FH洋葱进行了定义，例如我们认为DET\_OnionLayout是oFHDET，它的各个SECLEVEL为FH\_DET和RND的。

2021.3.28 9:34 PM 更新

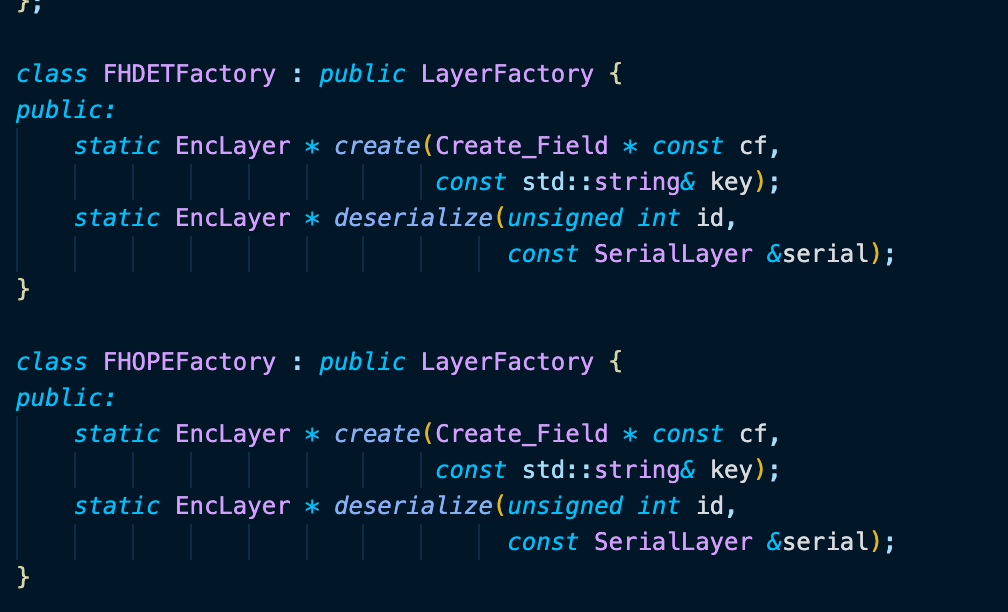
1. EncLayer作为洋葱层的抽象类，有一个专门生产的工厂类叫做EncLayerFactory，而针对不同的加密方式，EncLayerFactory也有不同的类型。例如：

* DETFactory生产oDET；
* OPEFactory生产oOPE。

**\*我们自己建立不用建立factory，只需要在其对应的DET类里，定义好FH\_DET\_INT, DEC, MEDIUMINT...，然后更改一下create函数就行。**

**\*问题在于，create接受的参数，不包括是否smooth。所以可能需要修改一下create的参数。而且，我了不影响原有的加密逻辑的实现方式，我认为把所有DET和OPE的create函数加入第三个默认参数type。**

2. 选择了一个Onion之后，对其内部所有的EncLayer，需要根据前面所说的SECLEVEL枚举类，创建一个对应的EncLayer出来。





应该在此定义两个新的工厂类和新的创建case。

而为了让工厂能够生产出我们所需的enclayer，我们要定义FH\_OPE\_str / int和FH\_DETstr / int的enclayer。

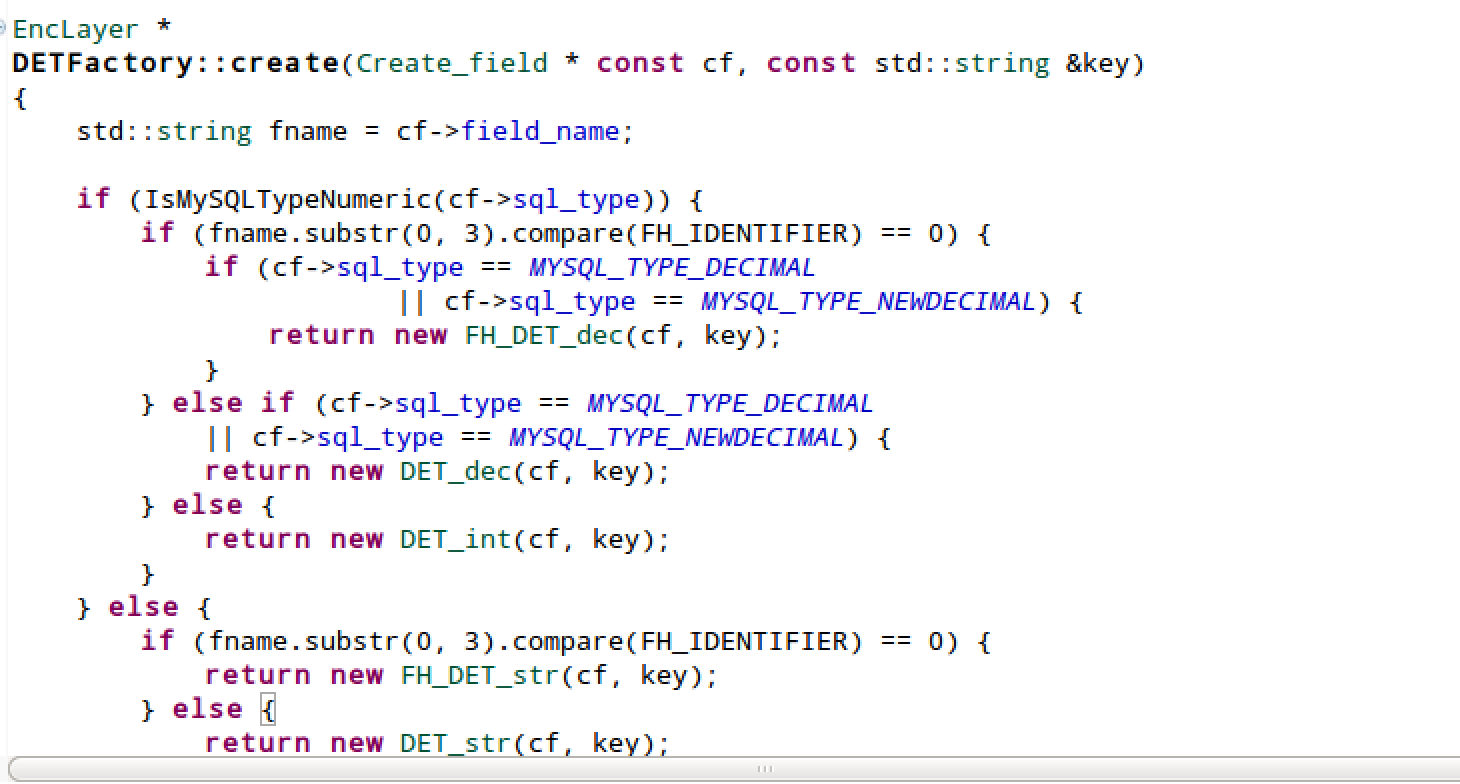
特别需要注意的是，如果deserializer没有实现，会报错，所以工厂的deserialize方法和serialize、create等都需要自行添加泛化类。

3. 泛化表可以存到文件里，而不是embedded database（还是为了方便起见。。）

**修改CryptoHandler文件！！！！**

2021.3.29 11:54 PM 更新

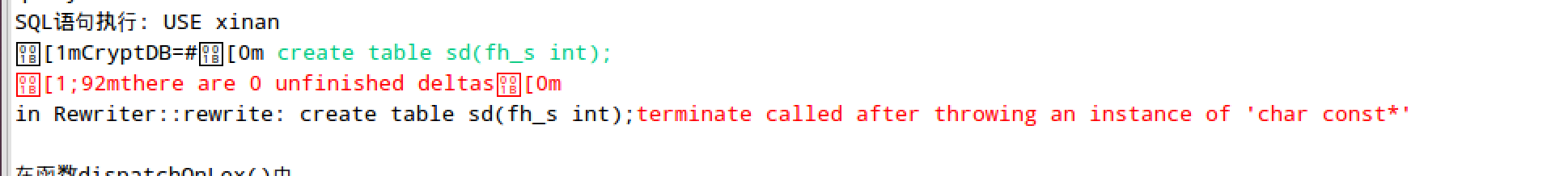
1. 在DET类型下新定义FH\_DET\_str, int和dec类型，并修改DETFactory::create(Create\_field \* const cf, const std::string& key)函数，在这个函数里面加入分支判断，获取cf->field\_name之后截取前三个字符，比对是否为“fh\_”。



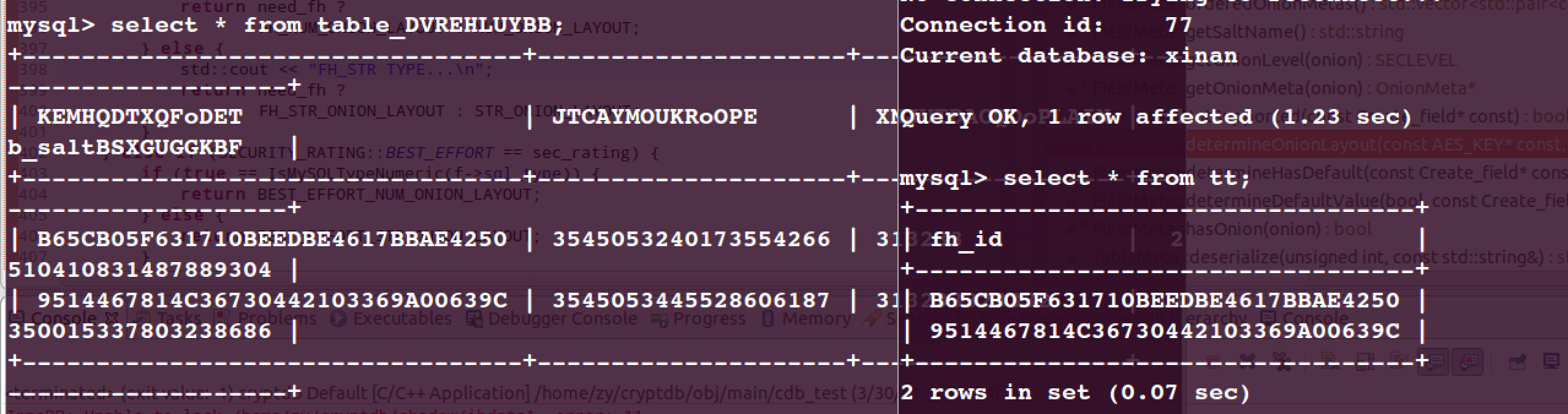
2. 在EncSet::chooseOne()函数中应该需要修改选择加密方式的函数逻辑。

2021.3.30 15:31 P.M.更新

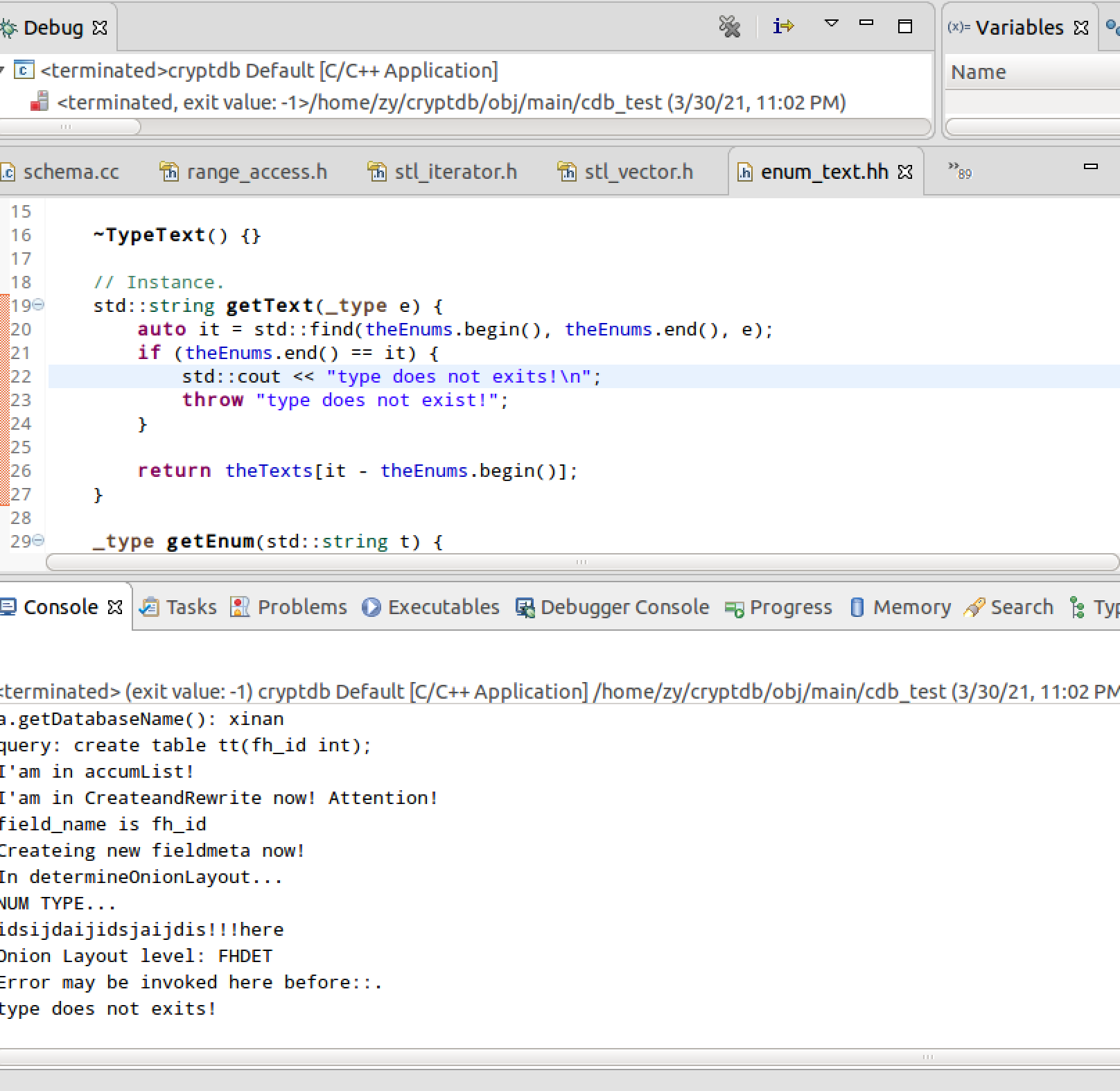
1. 如果仅仅加入自定义的洋葱层、洋葱类以及EncSet，但是没有把所有的判断函数和逻辑进行重写的话，会报各种异常。我遇到了特别多。

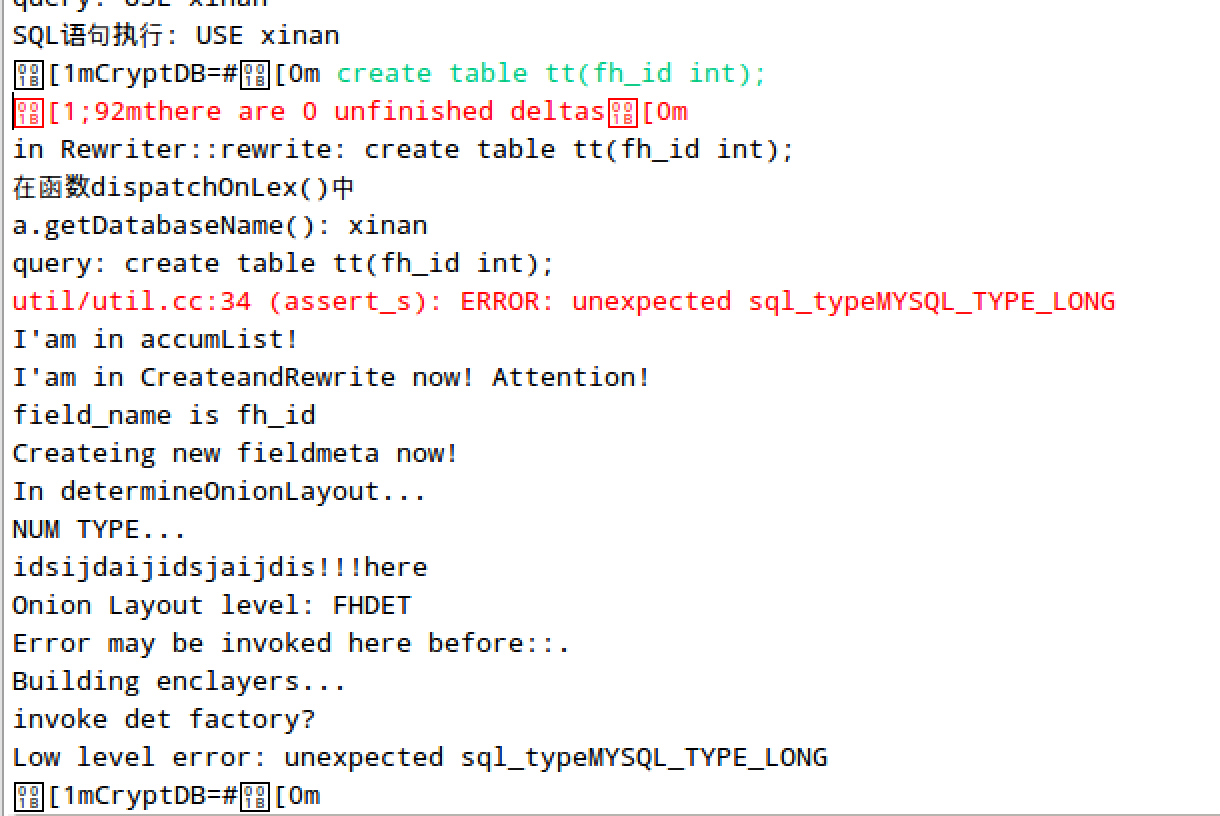
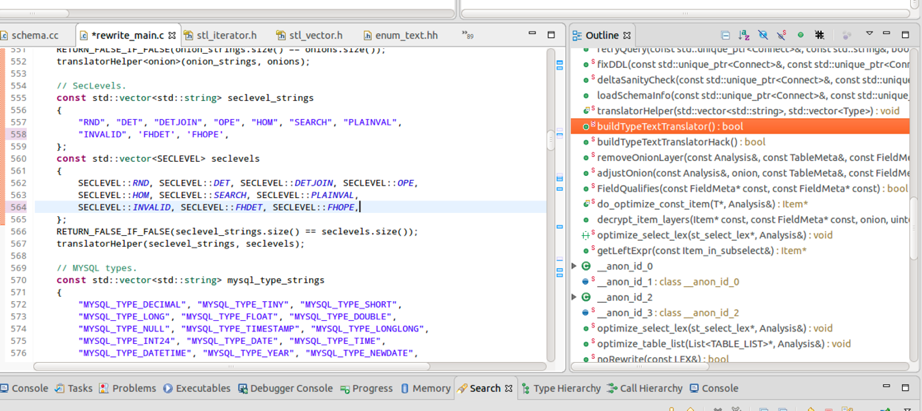


特别无语的是这个东西仅支持在console里面打印错误，但是在eclipse中没法显示具体的错误内容是什么。好在经过我们的一系列修改，我们自定义的泛化洋葱已经可以被CryptDB部分运行支持了。~~问题在于解密函数没写，所以它直接把DET拿了回来。~~Emm其实是decryptResult函数判断列名的bool值写反了orz。



！！！还需要实现string化，不然报错...





总之一番魔改之后它能认出来了:)。

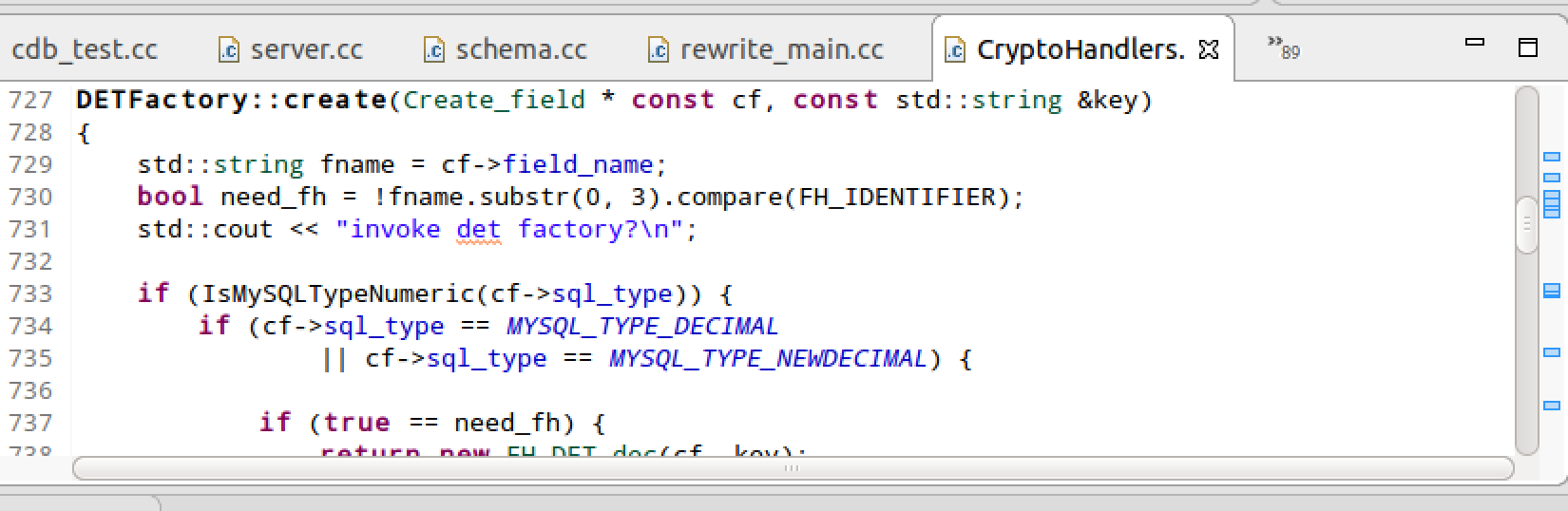
每个类型的string化是通过util下的stringify.hh实现的，它内部数据的初始化由rewrite\_main完成，其中buildTypeText便定义了这些字符串和对应的类。

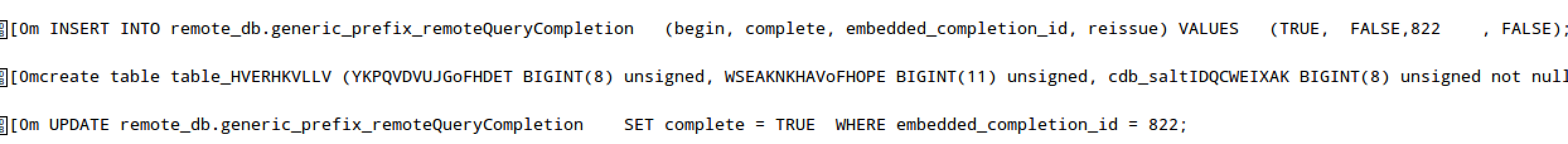
2. 泛化仅支持一个洋葱层。我想这样：

* 指定列名为“fh\_det\_...”；
* 指定列名为“fh\_ope...”。这个洋葱可以支持DET层，也可以支持泛化的OPE层。

2021.3.31 7:34 P.M. 更新

1. 通过修改DETFactory::create()函数，我们已经能够支持泛化洋葱类和相关的洋葱层级了。如下图所示：

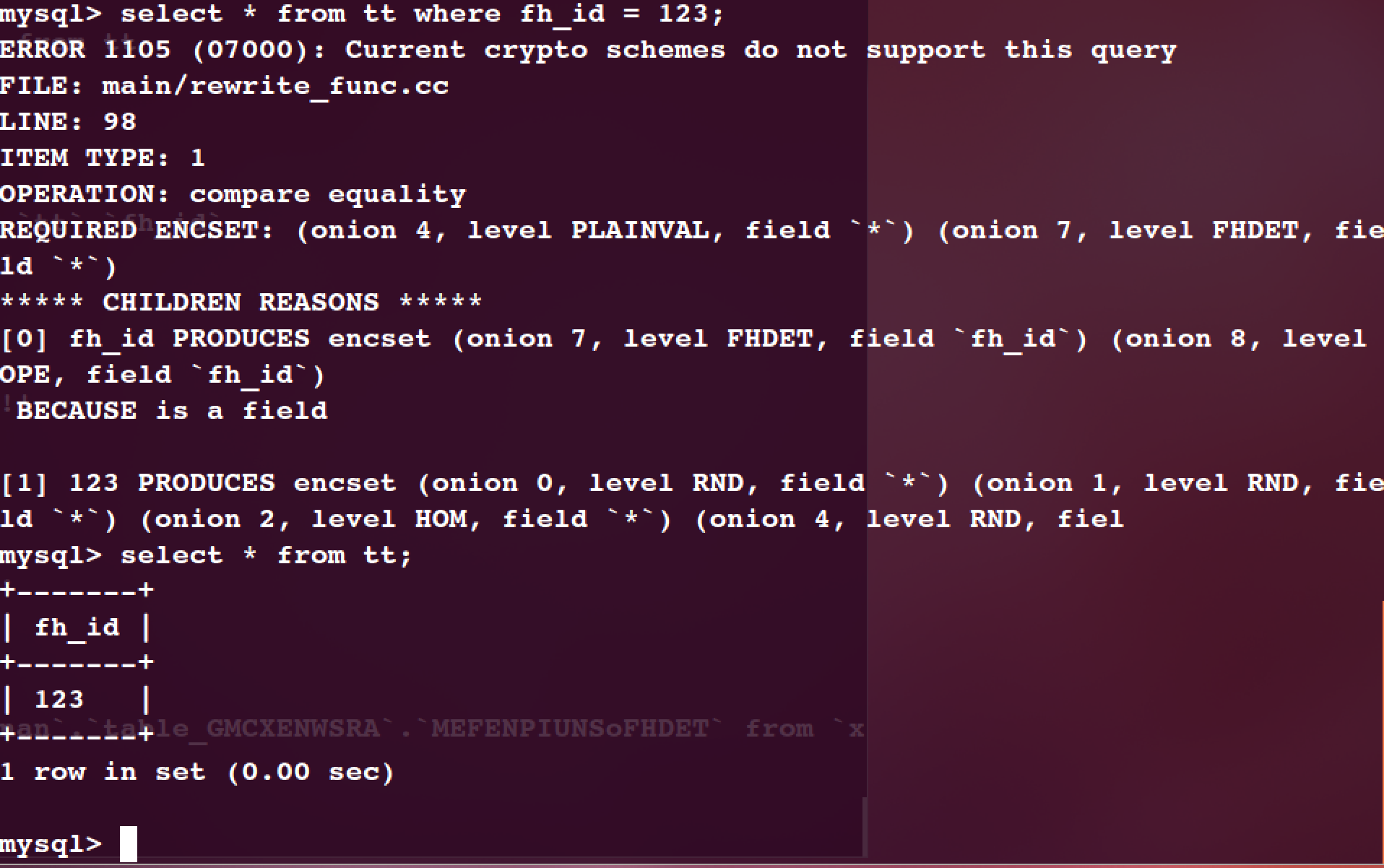




当我们指定某个列为 fh\_xxx的时候，可以发现cryptdb系统把这个语句改写成了仅支持DET和OPE的加密算法（salt可以给他扔了，也可以不扔，反正没有用到）。**至此，我们插入自定义的洋葱便实现了**。

2. 想要改写加解密函数也不会太困难，因为泛化的加密层FH\_DET\_str和FH\_OPE\_int /dec 都是有自己的encrypt和decrypt函数的。

3. 现在的主要问题是改写SQL语句了，增删改查都需要重写，不然就会这样：

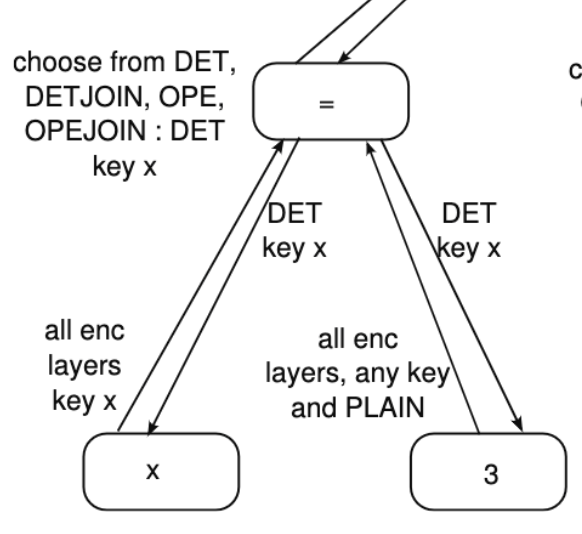


我认为gather函数也需要进行些许调整，毕竟where会调用gather进行重写，如果 equality comparison的右边这个常量的加密方法和左边的不一样那么结果就是找不到任何一个可以重写的方法。

cryptdb里面有个encset的交叉方法，它能够分析两个变量共同支持的加密方法，这样在做一些比较、相等的操作的时候，两者就能够被同时改写成所需的密文。不然就是报错了。

2021.4.1 6：24 P.M. 更新

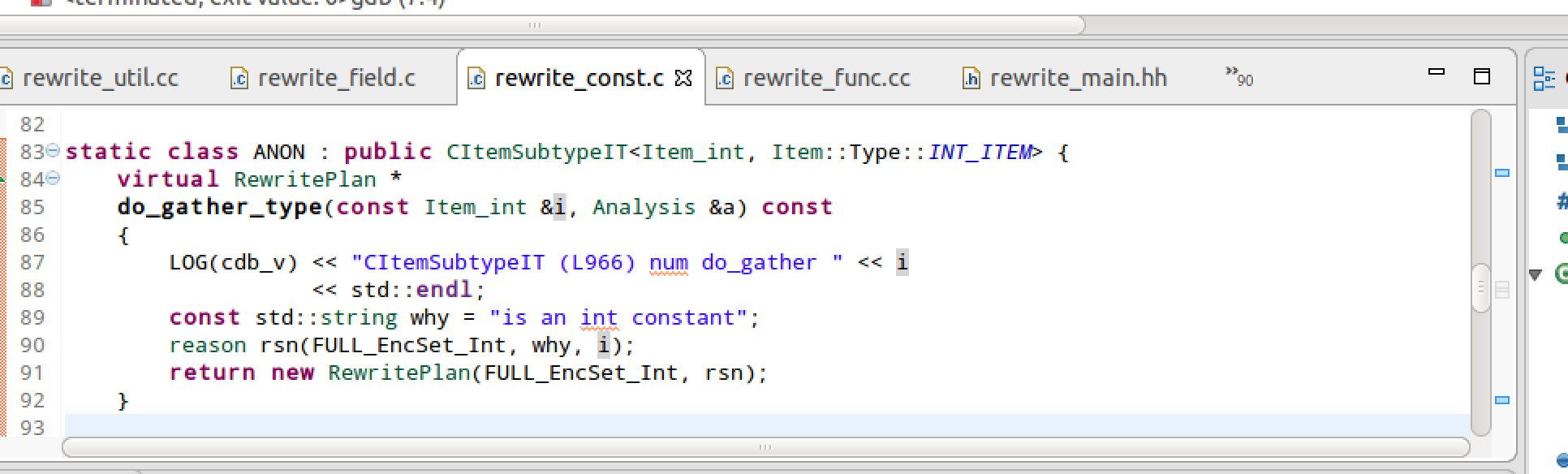
1. 如果使用FH\_DET\_ONION\_Layout的话会让一个元素仅有FH\_DET的洋葱层。但是cryptdb对一些where谓词类型的洋葱层没有设置FH\_DET和FH\_OPE等操作，所以在做EncSet的intersect的时候，会找不到共同支持的加密操作而产生一个异常。

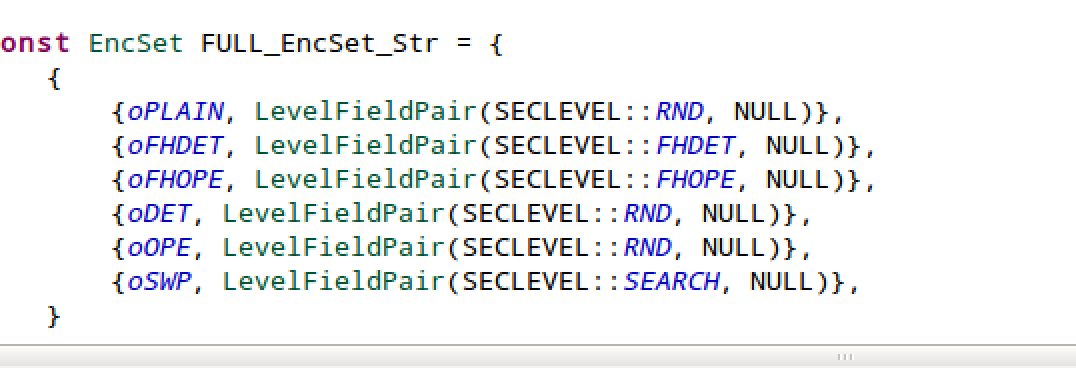


例如在做**SELECT** \* **FROM** table\_123456 **WHERE** X(如果是泛化的列) **=** 3的时候，lex树会生成一个=的操作，而cryptdb会在这个层级分析出DET/DETJOIN/PLAIN/OPE/OPE JOIN的洋葱层，但是对于这里的X它仅支持FH\_DET / FH\_OPE，所以在=的时候会找不到任何加密方法。~~为了解决这个问题，可能需要在CItemCompare这个类里面重写chooseEncSet函数。【别的类型同理】~~ （2021.4.1 edited 不需要，这是通过intersect方法来找的）

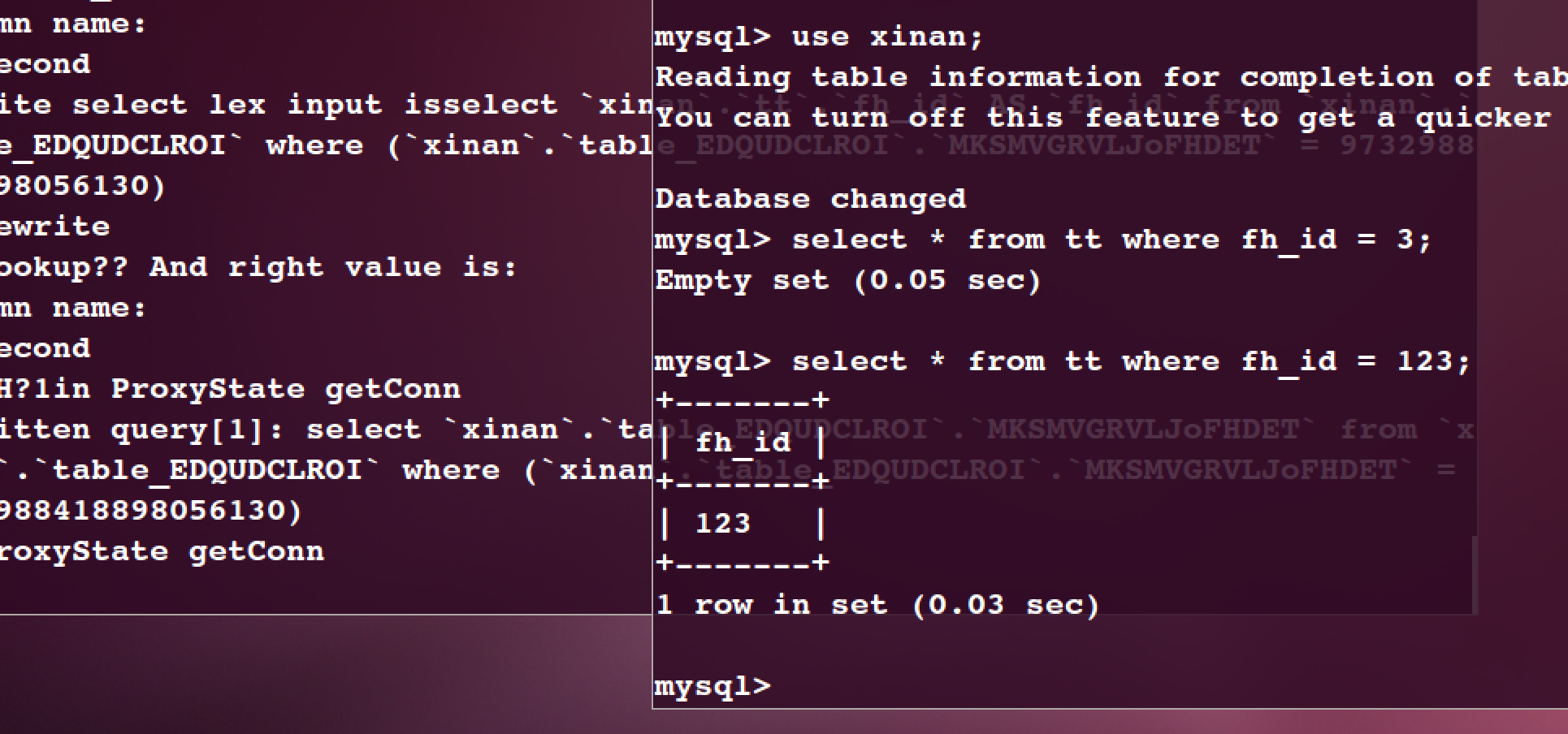
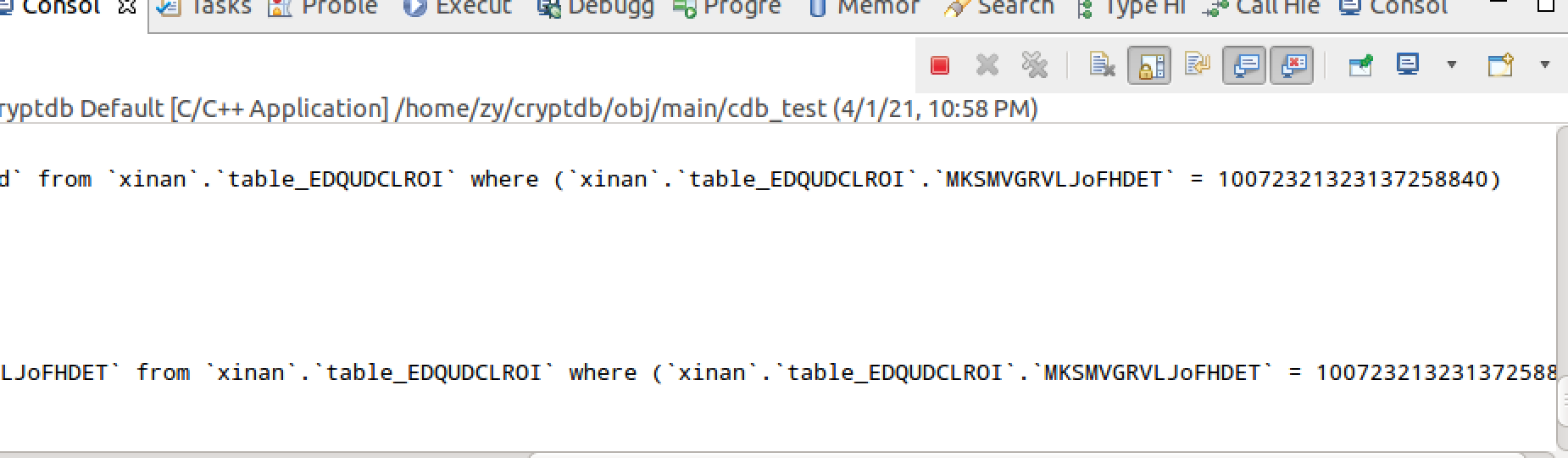


经过debugger定位后分析，常量会产生一个全集的加密集，而我们忘记改了，所以导致报错。把FH洋葱定义到FULLENCSET之后尝试WHERE ID = 3这种语句。

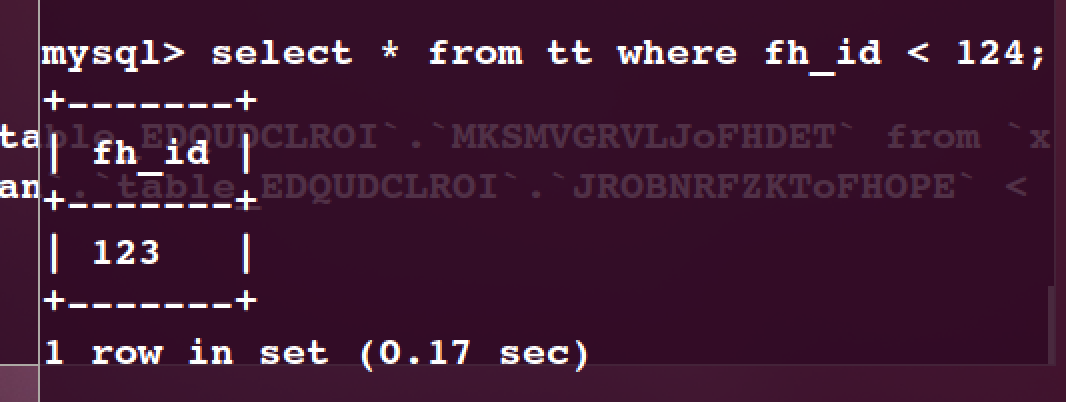




插入我们的FHDET洋葱。

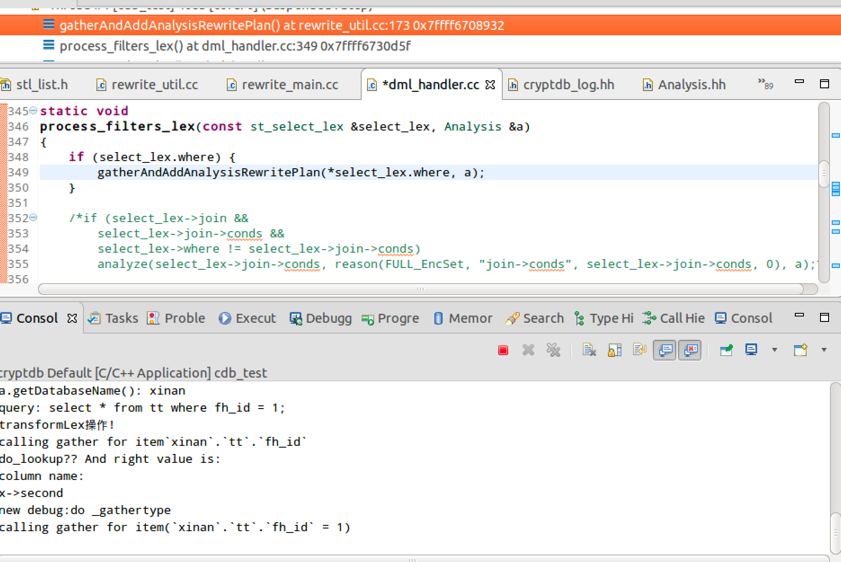


WHERE FH\_ID = 123已经成功被支持了。



P.S. 其实OPE和DET一样实现，就都可以成功查询了:)

2. where谓词通过下面的函数进行gather，但尚且不确定OR如何进行插入，才能把所有的泛化密文都给找出来，可能需要修改一下gather的逻辑，不过需要破坏lex结构了（要插入OR这个树节点）。



3. 还有个问题，泛化的salt查询表存储到哪个结构呢？对全局的Analysis类，FieldMeta中的私有成员？还是TableMeta？还是DataBaseMeta？反正读写都直接从文件中读写。

4.最后如果一切OK，记得把一堆调试的cout语句给删了，实在是太乱了orz

2021.4.2 21：22 P.M. 更新

1. 每个WHERE的判断谓词都会被分析成最小的type类型进行重写。如何将WHERE谓词进行修改，我认为可以有这样的思路：

① 在gather之前的时候就截取fieldmeta中的泛化信息（获取该重复明文使用的不同的盐值数量），然后对当前的lex树的where中不断插入和之前一样的filter，两者之间再用OR连接。加密重写的时候可以设置一个计数器叫做cnt，标记这次加密是应该用到哪个盐值了；

② 在rewrite的时候，对lex树进行添加OR的操作。加密逻辑同①。

③ 在DMLHandler进行transform的时候，对where进行添加。

我们可以尝试在encrypt\_item\_layers的时候进行判断并计数。为了不改动函数参数，我觉得可以把变量放进类里。。

最大的问题在于，到底在什么时候对where进行修改，如何对where进行修改。Lex树长什么样子？如果添加节点？？

* 可能需要学习sql\_lex结构。

2. 盐值可以在调用encrypt和decrypt函数之前给定，这样就不需要太多修改加解密逻辑了。只需要更改encrypt\_item\_layers函数的逻辑，以及对于FH类的encrypt函数的参数增加盐值。

2021.4.4 8:08 P.M. 更新

1. 【已解决】根据官方文档，加上我的源代码分析，我已经知道where xxx and xxxx or xxx这种语句是通过哪个子类进行重写的了。它属于conditional statement，所以会分发给ItemType下的CitemCond类中的gather和rewrite方法进行聚集和重写。

这个Item\_Cond东西的好处就是，可以把所有的Item\_Func和逻辑词包装成一个，就算很多个都可以塞下。



我们想要做到，如何在where谓词上插东西进去。

似乎有个函数叫做transform，能把不同类型的Item转换？。。。

2. 【已解决】已知rewrite\_filter函数会调用rewrite函数，这个rewrite函数的实现是通过查表，先判断where包含的item是什么属性，再调用对应的子函数重写。也就是说，想要把单个的fh\_id = 123改成OR属性，就需要在调用子函数之前截获，提升到它的父类CItemCond类型，再进行重写。

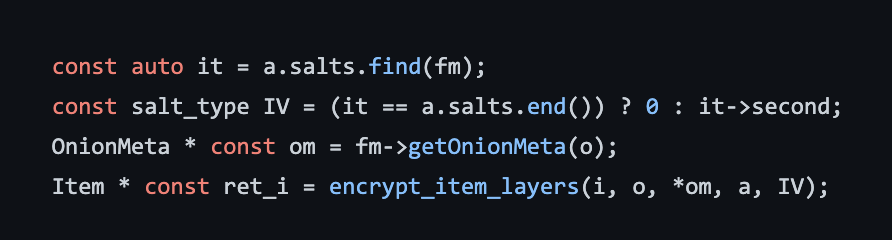
3. **直接在dispatchOnLex截获**，强制将where转换成一个Item\_cond\_or类型的对，这样可以保证多个or，只是可能需要获取本地数据。**这部分可以写一些函数。**

这样就可以保证我们的工作大部分是在where的增加以及encrypt / decrypt逻辑的修改了。

Where中可能会有多个and or嵌套，这部分需要分析lex结构，并且将属于conditional statement语句进行递归分析。不断地加入到结果的item\_cond\_or这个类中去，最后加入上层的对应conditional statement中。**我们甚至还可以在这些函数内部，做一下查表，然后把盐值插入到item里面去？。**

2021.4.6 8:26 A.M. 更新

1. 多次加密，可以在位于rewrite\_const的encrypt\_item中的实现。根据出现的次数对其进行加密。

****

↑这个函数可以对salt\_type IV的获取进行调整。

这里有大量难点：

① 如何将该表进行存储？ -> 存到~~fieldmeta~~；Analysis

② 如何遍历该表，且能够判断当前应该获取哪个盐值呢？

一个元素对应一个vector<Salt>结构，每个Salt结构内含使用次数。

已知encrypt\_item函数中可以获取到item对应的fieldmeta，也就是说，①中的表可以从此处获取。

③ 如何解密？盐值定长，比如16个字符；固定去除后16位。

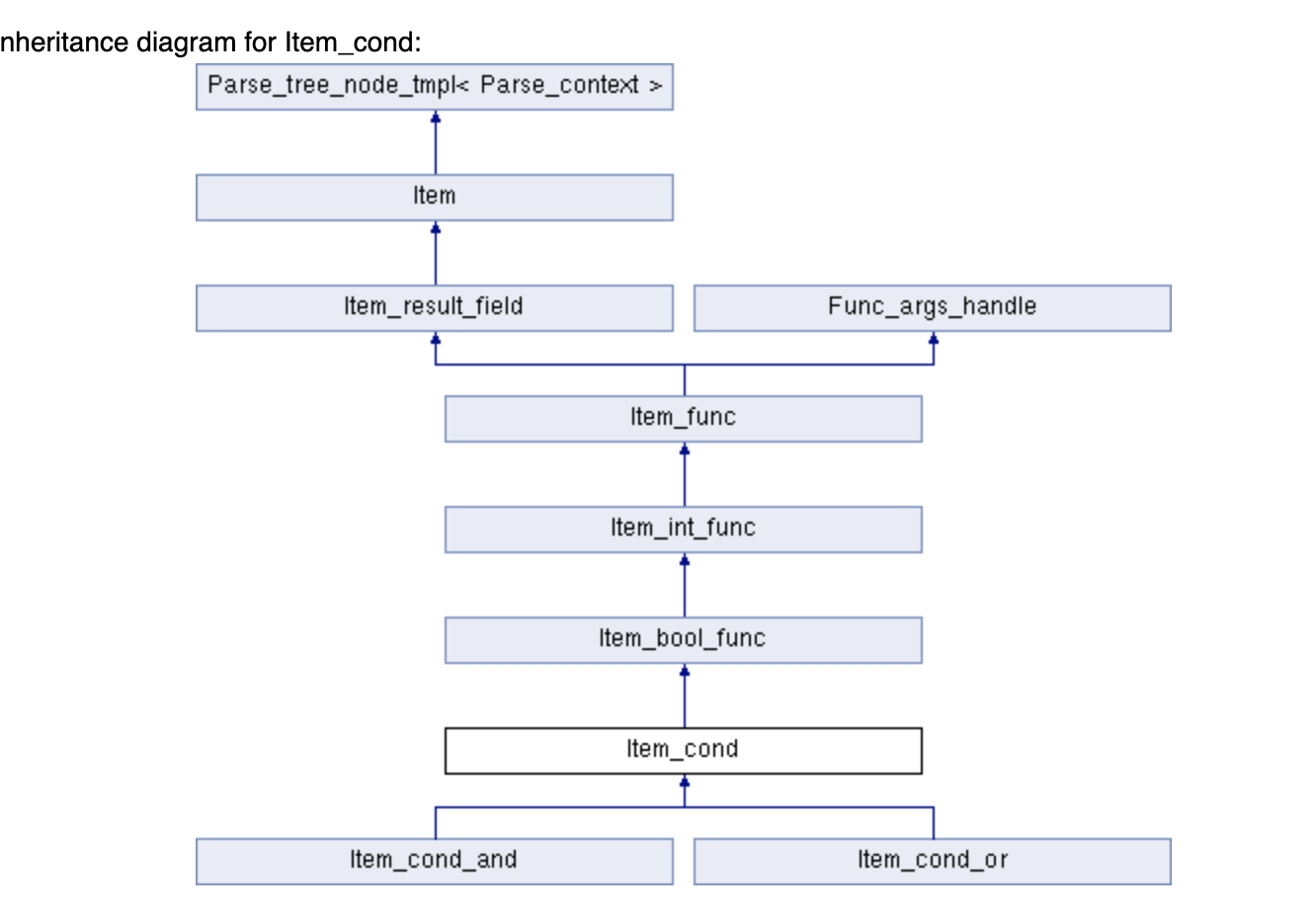
④（抬杠）UNIQUE的FH\_列？

这么来看，问题只剩下如何实现，盐值的选取。

~~2. 已经将salt table放入了fieldmeta，所以FieldMeta的deserializor和serializor需要增加存储和获取salt\_table的方法。~~

2021.4.7 6:26 P.M. 更新

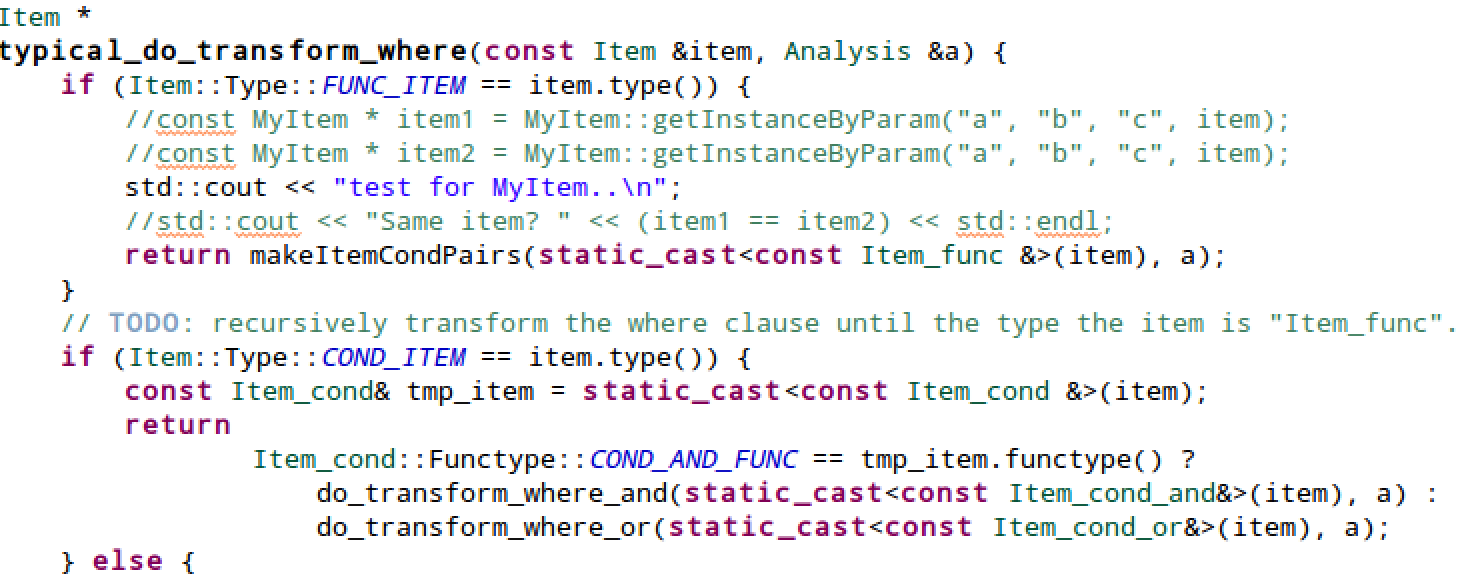
1.介绍一下我实现where重写的原理。



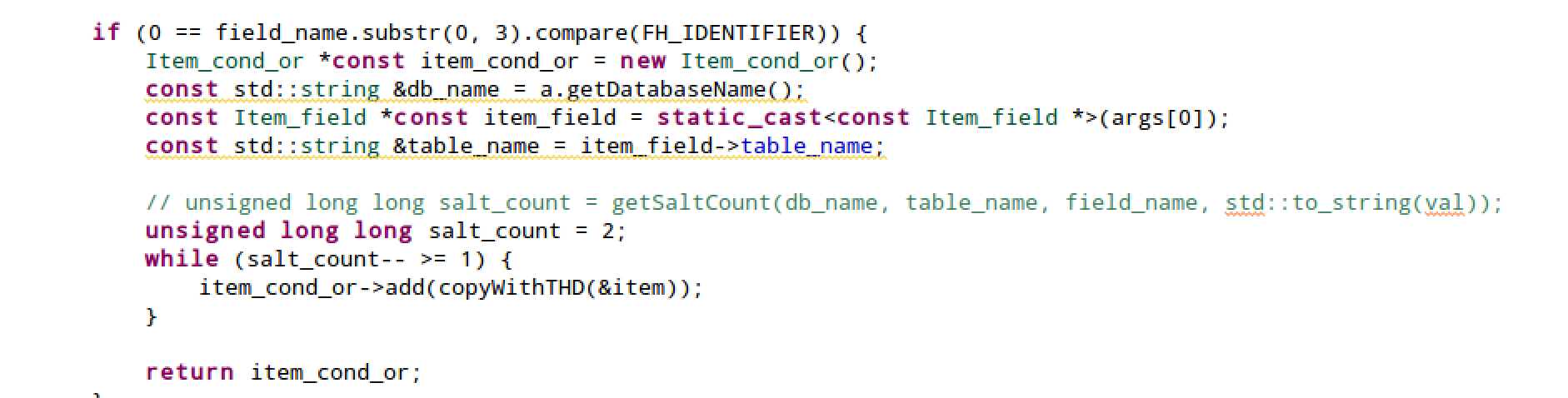
这是Item\_cond的图，它在LEX语法树中是属于XXX OR XXX的节点，它有一个公有方法叫做add，把某个Item插入到它自身之中。

那么，我们可以在最外层截获LEX树中的where语句，它返回一个Item类型的未知类型，所以我们要判断这个Item是不是OR / AND还是FUNC类型的。恰好Item也提供了一个公有接口叫做type，我们可以根据这个type来判断类型。

如下图所示。

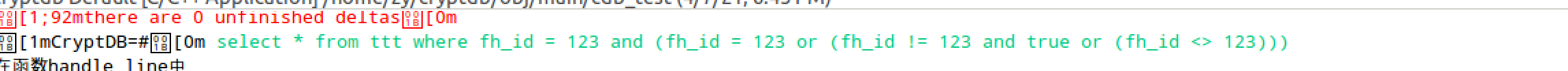


将where扔进这个函数之后我们会判断这是条件语句还是单个的判断语句，如果是单个的判断语句，我们就分发给转换func to cond的函数makeItemCondPairs，它获取盐的数量，并扩写func类型成为or。



如果在typical\_do\_transform\_where函数中发现是cond类型，就需要保留外层的AND或者OR，不然逻辑会乱。因此需要判断是and或者or，再调用对应的类型进行重写。对于cond的重写，就是遍历它的所有子节点（不一定是func类型的），然后递归地调用原来的typical\_do\_transform\_where函数进行再一次重写。

本质上是个递归操作，不过语句深度过大可能会导致栈溢出，这里可以修改成迭代类型。



被改写成了

SELECT \* FROM ttt AS ttt WHERE

((`xinan`.`ttt`.`fh\_id` = 123) or (`xinan`.`ttt`.`fh\_id` = 123)) and (((`xinan`.`ttt`.`fh\_id` = 123) or (`xinan`.`ttt`.`fh\_id` = 123)) or (((`xinan`.`ttt`.`fh\_id` <> 123) or (`xinan`.`ttt`.`fh\_id` <> 123)) and 60725032) or ((`xinan`.`ttt`.`fh\_id` <> 123) or (`xinan`.`ttt`.`fh\_id` <> 123)))

即使嵌套很多也能够正确改写。



被改写成了

SELECT \* FROM ttt AS ttt WHERE

(((`xinan`.`ttt`.`fh\_id` = 123) or (`xinan`.`ttt`.`fh\_id` = 123)) and ((`xinan`.`ttt`.`enc\_id` <> 123) or (((`xinan`.`ttt`.`fh\_id` = 123) or (`xinan`.`ttt`.`fh\_id` = 123)) and (`xinan`.`ttt`.`enc\_id` = 1234))))

非泛化列不会受到影响。

P.S. 因为没有实现盐表的读写，所以这里我统一扩展成2个了。

2. 实现自定义MyItem类。为了能够为每个单独的重复元素记录一个盐值表，以及方便管理，将Item包装成一个自定义的类，这样能够在Analysis类中加入一个map结构，用来查询每个元素对应的盐值表了；实际上为了能够解决加密次数查询问题，还需要在它内部加入一个计数表。由于用来区分每个item的标识符是item的值、列名、表名和数据库名，所以私有成员会包含这些变量。

3. 盐值变动，可以借助Delta类进行实现，只不过我们不需要写入embedded database，直接读写文件即可。

4. 文件中盐表的设计准则。

~~① 使用csv作为盐值文件进行保存。~~

~~② 读写采用github上的CSV parser库。（rapidcsv）~~

[~~https://github.com/d99kris/rapidcsv~~](https://github.com/d99kris/rapidcsv)

GCC版本过低，仅支持部分C++11语法，因此弃用这部分高级csv库。考虑使用JSON作为元数据文件存储。这里可以用腾讯在github上提供的rapidjson库。经测试，这个库works fine。

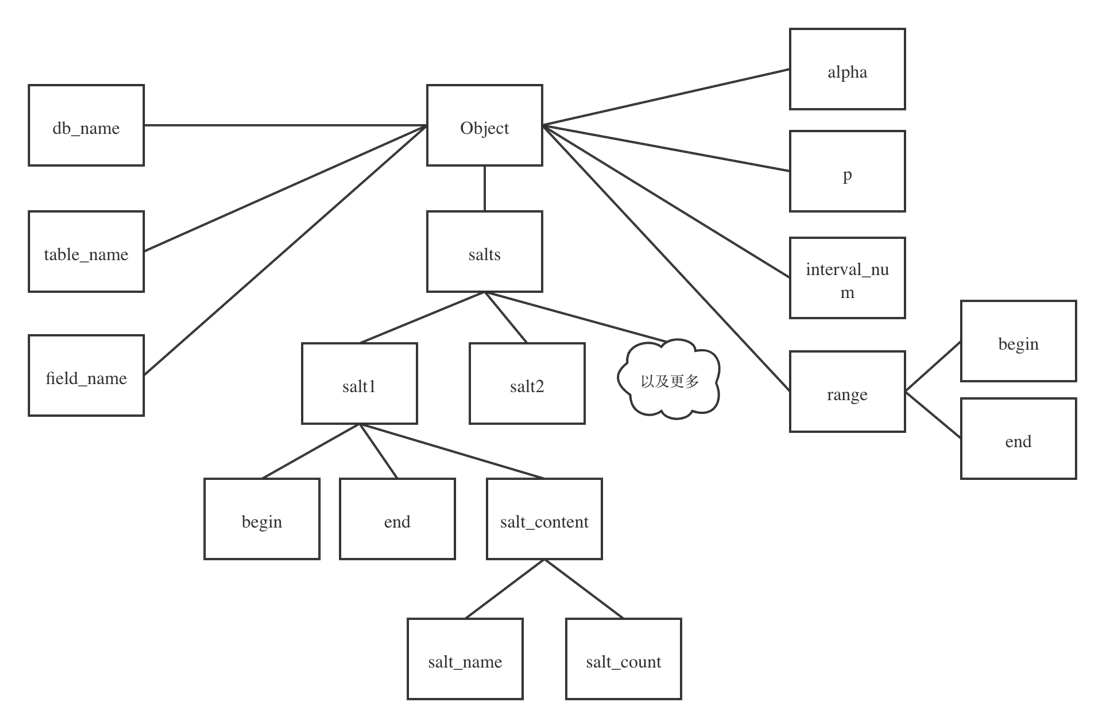
<http://rapidjson.org/zh-cn/index.html>

2021.4.10 00:38 A.M. 更新

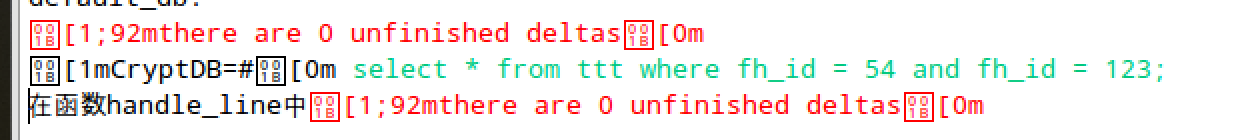
1. 完成了对subselect的where从句的改写。（不过默认的cryptDB会报错，因为subselec只支持明文改写？）

2021.4.11 3:15 P.M. 更新

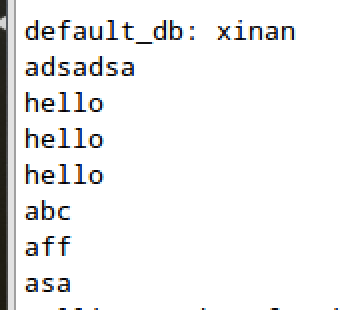
1.目前完成了从JSON文件中读取盐值，和一些关于列的必要信息了。其中json文件格式如下：



通过读取这个JSON文件，我们可以获取泛化必须的变量值，以及各自对应的盐值，盐的个数等，这样就可以很方便地加载到Analysis类中去了。利用rapidjson库中的数组和对象以及各自获取值的方法，我们就可以轻松地读取，也可以将值修改后写入等。



↑原始语句

← 获取到的盐值

select `xinan`.`table\_WNYJFADJKQ`.`BGRGSGOTEAoDET`,`xinan`.`table\_WNYJFADJKQ`.`YRNNGAJAPMoFHDET` from `xinan`.`table\_WNYJFADJKQ`

where

(((`xinan`.`table\_WNYJFADJKQ`.`YRNNGAJAPMoFHDET` = 1751847389435159482) or (`xinan`.`table\_WNYJFADJKQ`.`YRNNGAJAPMoFHDET` = 1751847389435159482) or (`xinan`.`table\_WNYJFADJKQ`.`YRNNGAJAPMoFHDET` = 1751847389435159482) or (`xinan`.`table\_WNYJFADJKQ`.`YRNNGAJAPMoFHDET` = 1751847389435159482))

↑54区间所在的盐值的数量 = 4。

and

((`xinan`.`table\_WNYJFADJKQ`.`YRNNGAJAPMoFHDET` = 15859447718984017919) or (`xinan`.`table\_WNYJFADJKQ`.`YRNNGAJAPMoFHDET` = 15859447718984017919) or (`xinan`.`table\_WNYJFADJKQ`.`YRNNGAJAPMoFHDET` = 15859447718984017919)))

↑123所在区间的盐值的数量 = 3。