##### 资源访问控制KV存储设计

###### 背景

为解决关系型数据库存储带来的部署、运维困难以及解决关系型数据库并发读写（网络传输、网络数据序列化、SQL解析、执行计划开销）效率较低，以及更新数据库连接数需要调优一堆参数的麻烦，为此设计一个本地文件数据库，以精简部署、降低运维工作量，且并发读写基本优于关系型数据库的本地索引数据库；

KV不足，限制多进程写访问，没有SQL在查询、存储方面的灵活性；

###### KV开源选型

* 目前后端程序多数以go为基础开发，以降低部署困难以及为后续轻量化部署，可选开源KV数据库有Leveldb、rocksdb、boltdb，badgerdb；
* LevelDB是采用LSM-Tree做索引键，索引键排序后放置key排序表SST文件（有共同前缀），并通过bloom过滤器用来加速查找，数据写入log，异步刷新键值到SST文件，所以写入速度很高，但是查询效率一般，对范围查询较多的需求影响较大；同时不支持事务，没有rollback操作，有go实现的goLevelDB版本；
* Rocksdb是对leveldb的各种优化，读写性能应该是最好的KV数据库，内核使用C++编写，依赖C库较多，在程序或者docker容器加载部署会遇到动态库匹配、缺失问题；同时goRocksdb对C接口支持并不友好，代码编译、部署问题较多；
* boltdb纯go编写，写性能稍微比Rocksdb弱，读性能与Rocksdb相当，支持事务，B+Tree检索Key值，BoltDB使用一个单独的内存映射的文件，实现一个写入时拷贝的B+树，这能让读取更快支持key值快速查找以及range范围扫描、支持bucket为对象的B+tree创建；
* badgerdb纯go编写，整个B+tree建立的索引存储内存，因为IO次数较低，所以查询较快，但是随着写入数据增多，内存吃紧，对硬件要求较高；
* 考虑以上因素，我们可选kv定格在boltdb（螺栓数据库）上做基础开发；

###### 总体思路

替换目前mysql数据库，首要考虑的问题是，关系型数据库大部分存储的是结构化的数据，灵活的支持数据检索以及查找；

KV作为非结构化的存储，KV根据业务可以存储任意结构数据，并且根据Key检索数据；所有没有关系数据库可以友好的复杂检索数据功能;

所以KV存储结构化的数据，首要解决的问题是:

1.建立结构化数据;

2.数据索引建立以及支持范围查询等问题；

对于关系型数据库，我们建立数据表一般有index索引，where查询也是根据索引查询的；

一般数据表都含有聚集索引作为主键，通过主键检索数据value，通过非聚集索引检索数据也是先通过非聚集索引找到主键索引，在通过主键索引找到value；

根据以上信息结合boltdb桶以及B+Tree原理，在将结构化数据表转为kv存储时，首先生成value的主键，通过boltdb自带id生成器去生成唯一主键或者根据纳秒随机数生成唯一主键；将唯一主键id作为key，存储的数据作为value；

主键key值将作为B+Tree的非叶子节点的数据域，方便快速的查找；value数值部分进行序列化存储，支持可选压缩方式存储value；

将其主键+value数据域放置一个桶中，降低B+tree的高度，降低IO次数；

同时将非唯一索引和主键索引放到另一个bucket桶中，其中对于非唯一索引，采用非主键索引+主键索引作为key值，value可取空值；

对于唯一索引，新建bucket桶，唯一索引直接作为key值，主键索引作为value值；

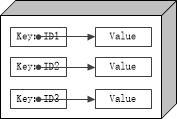
###### 基本存储

1. 基本存储结构

* 生成主键ID并和value存储在一个桶中，根据主键ID生成B+Tree；

保证主键ID索引的唯一性，如果非唯一主键ID会造成数据丢失

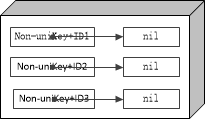
桶1：



* 非唯一索引建立复合键keys，value可为主键ID或为空；

检索时根据非唯一索引检索，获取复合键keys字段中的主键ID，再根据主键ID检索桶1中的 value数据；

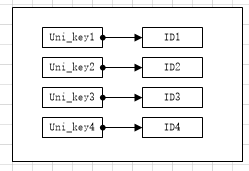
桶2：



* 唯一索引为key值，主键ID为value值

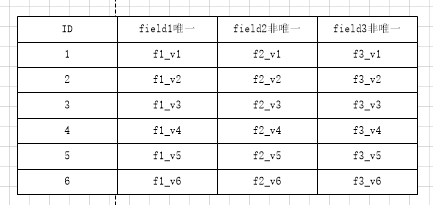
检索时根据唯一索引，获取value数值，将value转主键ID，再根据主键ID获取桶1中的value数值；

桶3：



2. 结构化数据表转化

例如一个mysql数据表:



主键ID为唯一索引;

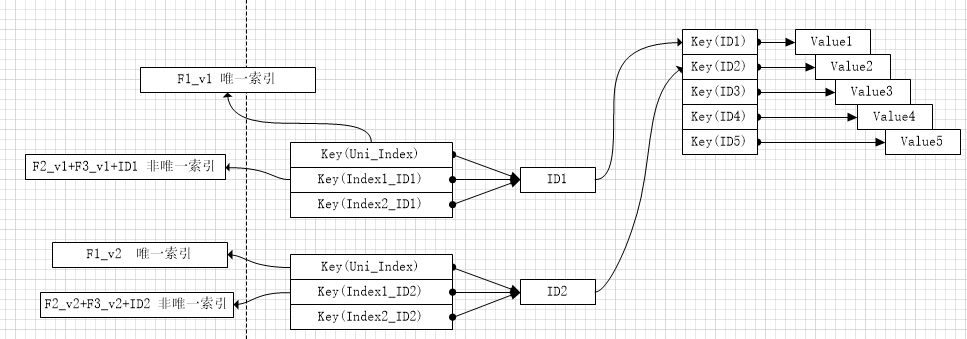
field1列需要建立唯一索引;

field2\_field3 两列建立非唯一索引;

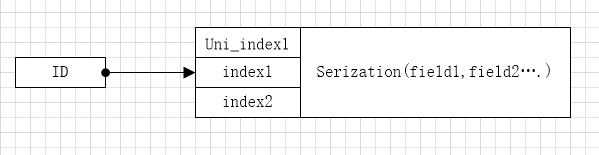
存储模型如下:

key(ID1)为唯一索引--->简单方式序列化Serialization(f1\_v1,f2\_v1,f3\_v1)

key(f2\_v1,f3\_v1,ID1) ---> Serialization(ID1)



3.怎么解决kv删除问题



在主键ID作为key，value存储序列化后多个字段；

删除时，根据主键ID检索出value，获取头部的索引字段：

如果是唯一索引则直接delete(唯一索引)，这样就对唯一索引这个桶进行了清楚，释放存储空间；

如果是非唯一索引,执行delete(非唯一索引+主键ID)进行一条一条删除；

4.更灵活的获取value方式：

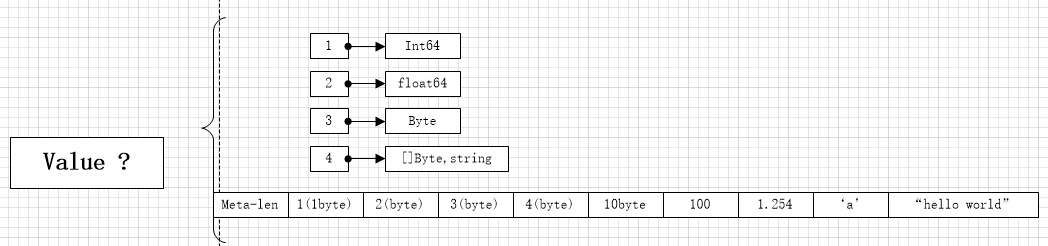
在每个value头部定义meta元数据,定义每个数据类型占据长度,根据偏移获取每隔数据长度;

meta-len=1len(int64)+2len(float64)+3len(byte)+4len(string)

(value+meta-len)[0:len(int64)]为第一个int64字段的数据;

如果每次get(key,field1[offset:len],field2[offset:len].....) 获取value中的局部字段；

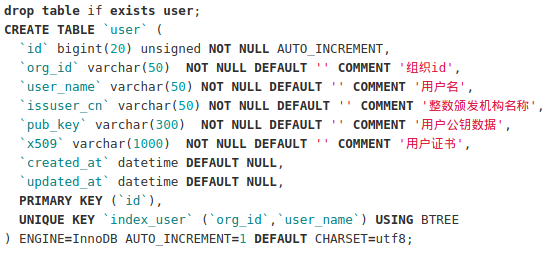
需要bolt深度开发，支持value局部字段获取；



基于where条件的范围查找，boltdb提供seek定位B+Tree节点，以及next、pre等函 数迭代查询,基本满足范围查询要求；

###### 权限访问控制建立KV存储结构设计

* 机构user表：



* 主键索引建立：

主键ID（boltdb-uniID1），采用boltdb自身id生成唯一主键key；

value：是{org\_id,user\_name,issuser\_cn,pub\_key,x509,created\_at,update\_at} 序列化后的数据；

最终的主表存储结构是：

key(boltdb-uniID1）->value1(bytes{org\_id,user\_name,issuser\_cn,pub\_key,x509,created\_at,update\_at})

key(boltdb-uniID2）->value2(bytes{org\_id,user\_name,issuser\_cn,pub\_key,x509,created\_at,update\_at})

key(boltdb-uniID3）->value3(bytes{org\_id,user\_name,issuser\_cn,pub\_key,x509,created\_at,update\_at})

.................

* 唯一索引建立：

其中访问控制sql语句：

select org\_id,issuser\_cn,user\_name,pub\_key,x509,created\_at,updated\_at from user

     where org\_id=? and user\_name=?

是根据组织org\_id和用户名 user\_name精确查询一条记录,查询是根据unique-key 唯一检索查询的，对KV数据库怎么解决唯一索引问题，如下：

将唯一索引unique-key的字段org\_id,user\_name作为key值，value值为主键ID（boltdb-uniID)；

检索时根据org\_id,user\_name条件，检索出主键id，在根据主键id检索出实际value；

最终存储结构就是：

unique-key1(org\_id,user\_name) -->value(主键boltdb-uniID1)

unique-key2(org\_id,user\_name) -->value(主键boltdb-uniID2)

unique-key3(org\_id,user\_name) -->value(主键boltdb-uniID3)

* 资源权限表



此表索引较多，需要在kv设计更多的索引建立，以达到数据检索要求；

* 主键索引建立

主键ID（boltdb-uniID1），采用boltdb自身id生成唯一主键key；

value：是{res\_type,res\_id,auth,acc\_org\_id,acc\_user\_name,own\_org\_id,own\_user\_name,created\_at,updated\_at} 序列化后的数据；

最终的主表存储结构是：

key(boltdb-uniID1）->value1(bytes{res\_type,res\_id,auth,acc\_org\_id,acc\_user\_name,own\_org\_id,own\_user\_name,created\_at,updated\_at})

key(boltdb-uniID2）->value2(bytes{res\_type,res\_id,auth,acc\_org\_id,acc\_user\_name,own\_org\_id,own\_user\_name,created\_at,updated\_at})

key(boltdb-uniID3）->value3(bytes{res\_type,res\_id,auth,acc\_org\_id,acc\_user\_name,own\_org\_id,own\_user\_name,created\_at,updated\_at})

.................

* 唯一索引uniq建立

表唯一索引是：

UNIQUE KEY `uniq` (`res\_type`,`res\_id`,`acc\_org\_id`,`acc\_user\_name`,`own\_org\_id`,`own\_user\_name`) USING BTREE

sql语句是：

`update res\_auth set auth=:auth,updated\_at=:updated\_at  where res\_type=:res\_type and res\_id=:res\_id and acc\_org\_id=:acc\_org\_id and acc\_user\_name=:acc\_user\_name and own\_org\_id=:own\_org\_id and own\_user\_name=:own\_user\_name`

根据资源类型res\_type、资源res\_id、机构own\_org\_id、机构用户名own\_user\_name和被授权机构acc\_org\_id、授权机构用户名acc\_user\_name,修改其auth权限；

将唯一索引unique-key的字段res\_type,res\_id,acc\_org\_id,acc\_user\_name,own\_org\_id,own\_user\_name作为key值，value值为主键ID（boltdb-uniID)；

检索时根据res\_type,res\_id,acc\_org\_id,acc\_user\_name,own\_org\_id,own\_user\_name条件，检索出主键id，在根据主键id检索出实际value；

最终存储结构就是：

unique-key1(es\_type,res\_id,acc\_org\_id,acc\_user\_name,own\_org\_id,own\_user\_name) -->value(主键boltdb-uniID1)

unique-key2(es\_type,res\_id,acc\_org\_id,acc\_user\_name,own\_org\_id,own\_user\_name) -->value(主键boltdb-uniID2)

unique-key3(es\_type,res\_id,acc\_org\_id,acc\_user\_name,own\_org\_id,own\_user\_name) -->value(主键boltdb-uniID3)

* 非唯一索引建立

例如对索引：

KEY `idx\_acc` (`res\_type`,`acc\_org\_id`,`acc\_user\_name`) USING BTREE,

KV建立如下结构：

Key(res\_type,acc\_org\_id,acc\_user\_name,主键boltdb-uniID1) ---> value(nil)

检索时根据res\_type,acc\_org\_id,acc\_user\_name条件，检索出一堆key值，在分离出key的主键boltdb-uniID1，在根据主键boltdb-uniID1检索出主表value值；

注：

同时对需要进行merge操作的复杂sql语句，例如where in(??) and cond1 and cond2...

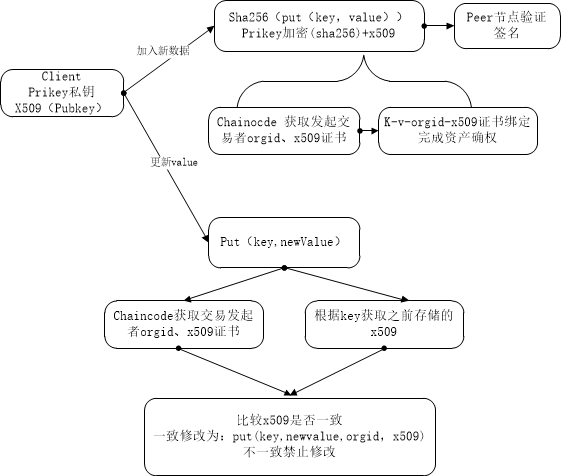
可以先建立：

cond1，cond2作为key的非唯一主键，value存储主键；

检索时，根据条件检索出结果集，同时与gets对等的where in(??)结果集进行过滤合并操作,这些需要调用接口层根据数据特性做更多的合并操作，底层提供灵活的接口；

###### 资源访问控制

方案1:



方案2:

