DECIMAL 功能开发文档

修订历史

版本	日期	修改描述	作者	备 注
Cedar 0.3	2017-9-28	DECIMAL 功能开发文档	徐石磊	无

1 需求分析

Cedar是华东师范大学计算机科学与软件工程学院基于OceanBase数据库研发的可扩展的关系数据库，实现了巨大数据量上的跨行跨表事务。在cedar 0.2版本中缺乏对DECIMAL数据类型的支持，为满足业务功能上的需求，在cedar 0.3版本中增加DECIMAL数据类型，使得数据库能够实现存储，读取decimal类型的数据，并能进行相关的数学运算。

2 模块设计

2.1 表Schema设计

2.1.1 需求

为了使Cedar支持decimal数据类型，首先需要在建表的时候支持包含有decimal数据类型的表的创建，同时需要存储用户在建表时指定的相关列上decimal数据类型的参数，包括有效数字和精度。比如用户定义了某一列为decimal(10,5)。其中precision为10，表示有10位有效数字，scale为5，表示有5位小数。
对于Cedar来说，每个server（包括rs，ups，cs，ms）都有自己的ObSchemaManagerV2。ObSchemaManagerV2里存储并管理所有表的schema，包括每张表有多少列，每一列是什么数据类型，并且保持所有server上ObSchemaManagerV2所管理的表schema是一致的。下面是ObSchemaManagerV2的主要数据结构。

```
    class  ObSchemaManagerV2{
      private:
        ObTableSchema    table_infos_[OB_MAX_TABLE_NUMBER];
        ObColumnSchemaV2 *columns_;
    };
    class  ObColumnSchemaV2
    {
     private:
        bool maintained_;
        bool is_nullable_;
        uint64_t table_id_;
        uint64_t column_group_id_;
        uint64_t column_id_;
        int64_t size_;  //only used when type is char or varchar
        ColumnType type_;
        char name_[OB_MAX_COLUMN_NAME_LENGTH];
        ObObj default_value_;
    //join info
        ObJoinInfo join_info_;
    //in mem
        ObColumnSchemaV2* column_group_next_;
};
```

每个server运行的时候都会维护自己的ObSchemaManagerV2。这个schema是怎么获得的呢？对于RS来说，第一次启动时从磁盘中读取，每次执行建表语句时对其进行修改。对于其他server来说：1. RS通过心跳通知。2. 建表语句执行的时候，RS会向其他server发送最新版本的schema。

2.1.2 定义

由于Cedar 0.2不支持decimal，在实现ObSchemaManagerV2的时候，没有设计数据结构来存储用户定义的精度和有效数字。所以，为了实现decimal，必须对ObSchemaManagerV2进行修改。我们的修改方案是：在ObColumnSchemaV2里面增加了一个数据结构：

```
struct ObDecimalHelper {
    uint32_t dec_precision_ :7;
    uint32_t dec_scale_ :6;
};
```

相应的，为了使该数据结构能够起作用，需要补上对它进行设置、修改、获取的接口，以及修改ObSchemaManagerV2中schema序列化和反序列化函数。因为ObSchemaManagerV2是各个server在内存中管理各个表shema的结构，无法直接在不同server之间进行传递。
注：在修改了ObSchemaManagerV2之后，要对整个集群进行boot_strap（删除集群原有数据）之后才能重启，否则会报错。

2.1.3 设计

建表语句执行的时候，MS会把建表语句进行解析，把用户输入的一些schema信息封装到一个TableSchema数据结构中。这个TableSchema中OB设计了对用户输入的精度和有效数字的存储空间，故无需要修改。仅需修改MS封装TableSchema的函数，将用户输入精度和有效数字存到TableSchema里面即可。具体的代码为：

```
Sql/ob_transformer.cpp
gen_physical_create_table()
{
col.data_precision_ = col_def.precision_;
col.data_scale_ = col_def.scale_;
}
```

TableSchema封装好之后，MS会把TableSchema发送到RS。RS接收到TableSchema之后，会调用add_new_table_schema()函数根据TableSchema里面的值，对自己的ObSchemaManagerV2进行修改。我们只需要在该函数里，把TableSchema里面的用户输入的精度和有效数字位数存到TableSchema里的ObDecimalHelper即可。具体代码如下：

```
Common/ob_schema.cpp
add_new_table_schema()
{
ObColumnSchemaV2 old_tcolumn;
const ColumnSchema &tcolumn = tschema->columns_.at(i);
old_tcolumn.set_table_id(tschema->table_id_);
if (tcolumn.data_type_ == ObDecimalType)
{
old_tcolumn.set_precision(static_cast<uint32_t>(tcolumn.data_precis
ion_));
old_tcolumn.set_scale(static_cast<uint32_t(tcolumn.data_scale_));
}
}
```

由于在ObSchemaManagerV2里面增加了一个数据结构，而ObSchemaManagerV2又是需要在各个server之间进行传输的，所以我们要继续修改ObSchemaManagerV2的序列化与反序列化函数。具体代码如下：

```
Common/ob_schema.cpp
DEFINE_SERIALIZE(ObColumnSchemaV2)
{
...
  if(OB_SUCCESS == ret&&ObDecimalType==type_)
  {
      ret = serialization::encode_i8(buf, buf_len, tmp_pos,ob_decim
al_helper_.dec_precision_);
  }
  if(OB_SUCCESS == ret&&ObDecimalType==type_)
  {
      ret = serialization::encode_i8(buf, buf_len, tmp_pos,ob_decim
al_helper_.dec_scale_);
  }
...
}

int ObColumnSchemaV2::deserialize_v4()
{
...
if(ObDecimalType==type_){
          int8_t p= 0;
          int8_t s = 0;
          if (OB_SUCCESS == ret) {
              ret = serialization::decode_i8(buf, data_len,tmp_po
s,&p);ob_decimal_helper_.dec_precision_=static_cast<uint8_t>(p) & M
ETA_PREC_MASK;;
          }
          if (OB_SUCCESS == ret) {
              ret = serialization::decode_i8(buf, data_len,tmp_po
s,&s);
              ob_decimal_helper_.dec_scale_=static_cast<uint8_t>(s)
& META_SCALE_MASK;
          }
}
...
}
```

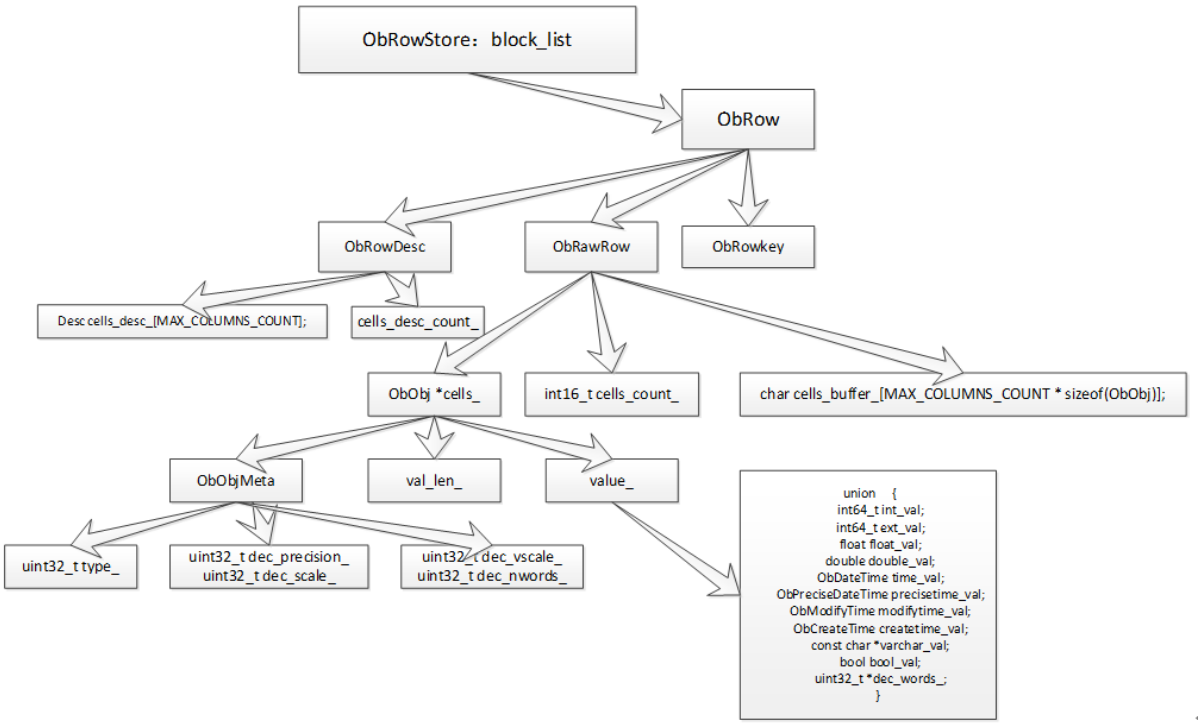
至此，我们完成了对ObSchemaManagerV2的修改

2.2 ObDecimal基本类设计

在完成了decimal类型的schema的存储以及sql输入decimal数据的识别之后。为了封装整个decimal的操作（如内存结构，序列化，反序列化，四则运算等），我们增加了一个新的ObDecimal类。

2.2.1 Decimal类型实现：

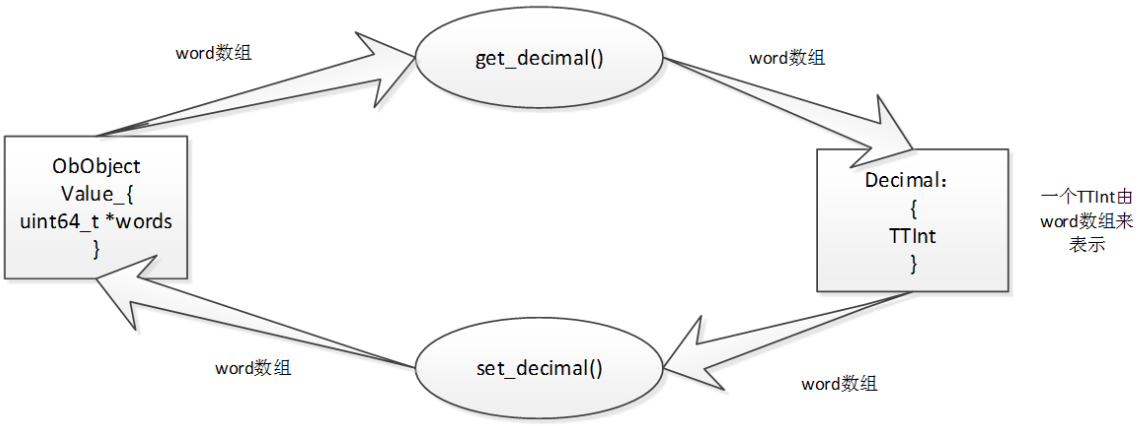
Cedar 通过ObObj封装了最小数据处理单元，进而实现上层操作对不同数据类型的泛型化操作。但是ObObj未能实现对变长和定长数据的统一泛型化，进而导致在现有OB的基础上增加对decimal类型支持的难度（极大的扩大了代码修改范围）。
下面我们首先认识ObObj的结构：



ObObj是Cedar最基本的存储结构。Cedar把用户输入的值封装成ObObj。写到磁盘的时候调用ObObj的序列化。ObObj序列化的时候会根据自己的type，调用不同类型的序列化。同样在读取数据的时候也会根据不同的数据类型进行反序列化与解析。
从上图可以看出，以一个联合体union来表示对于不同种数据类型的值，

```
union           // value实际内容
{
    int64_t int_val;
    int64_t ext_val;
    float float_val;
    double double_val;
    ObDateTime time_val;
       ……
}value_
```

考虑到Decimal类型的数据的存储空间较大（目前使用N*64bit存储），同时不增加ObObj中value_的存储开销（64bit），我们将具体的Decimal使用外部空间来存储（如果将Decimal的数值放在ObObj中将大幅降低修改代码的范围）。
为了不增加内存开销，我们的实现手段为，在value_结构体内增加一个uint64_t类型的指针*tt，这个指针指向一个uint64_t类型的数组（我们设计Decimal类型的时候借用了开源的ttmath的Int类，将一个Decimal数转换成大数来处理，运算时也使用这个大数来作运算，TTmath通过这个数组计算转换成一个大数。）ObObj中decimal值与ObDecimal中decimal值相互赋值过程如下：



在ObObj类当中，有get_decimal和set_decimal两种方法，get_decimal将value_中指向的uint64_t数组取出，赋值给Decimal类型的TTInt成员，反之，通过set_decimal将一个封装的Decimal类中TTInt的uint64_t数据赋值给value_。
因为要尽量对得上原先OceanBase的一些接口，所以Decimal类的数据结构设计成与ObNumber类似。

```
Class ObDecimal
{
 Private:
    uint32_t vscale_;
    uint32_t precision_;//存储用户定义的精度
    uint32_t scale_;     //存储用户定义小数位数
    TTInt word[NWORDS]; //用一个ttmath的Int类型ttmath::int来存储实际值
}
```

其中，NWORDS=1，即使用一个TTInt来存储Decimal值。

```
typedef ttmath::Int<2> TTInt;
typedef ttmath::Int<4> TTCInt;
typedef ttmath::Int<8> TTLInt;
```

对于64位机器，在ttmath运算接口当中，ttmath::Int是由一组uint64_t数组来实现大整数，也就是ttmath:Int,size的值可以自定义设置，比如当size=2的时候，用两个uint64_t实现大整数。这时候一个TTInt可以表示的数值范围是-2^(64*2-1)to2^(64*2-1)，转换成十进制就是38位整数。我们选取的t_size为2，即这个大整数word可以表示的数值范围是38位整数。其中，约定小数部分最多为37位。
将一个数转换成TTInt通过如下计算实现：

```
TTInt=输入数*kMaxScaleFactor
const static TTInt kMaxScaleFactor=10…0   (30个0)
例如：
输入数为3.14，
则用转化成TTInt表示为
3140000000000000000000000000000                    （28个0）
又如输入数为3.014
转换成TTInt表示为
3014000000000000000000000000000                     （27个0）
```

2.3 运算设计

2.3.1四则运算

四则运算，包括加，减，乘，除。计算的时候，直接取出TTInt类型的word进行四则运算，因为总长度是对齐的，所以只需要在最后的结果根据两个数的scale，precision进行结果处理即可。而对于乘法和除法，直接做运算的时候会造成结果溢出，所以用typedef ttmath::Int<8> TTLInt；用于乘除计算。因为这个类型长度四倍于TTInt，所以不会造成溢出，计算时，先将两个乘数转化成TTLInt，再进行乘除计算，结果与MaxDecimalValue/MinDecimalValue比较，做溢出处理。Cedar数据库在存储Decimal时进行数值检测，运算是在存入的数值上进行，因此运算时不需要再进行数值检测，只需要检测是否溢出。
在实现运算时，为了保持和Cedar中ObNumber类的运算接口一致，将ObDecimal类运算定义如下：

```
int add(const ObDecimal &other,ObDecimal &res) const;
int sub(const ObDecimal &other,ObDecimal &res) const;
int mul(const ObDecimal &other,ObDecimal &res) const;
int div(const ObDecimal &other,ObDecimal &res) const;
```

其中，other表示的是加数、减数、乘数和除数；res表示的是结果。
Add和Sub运算中，将两个Decimal的数值存放在TTInt中，然后调用TTMATH的add和sub进行运算，将运算结果存在res.word[0]中。计算res.word[0]的长度，并与38进行比较来判断是否溢出，最后设置相应的precision和scale。
Mul运算中，将两个Decimal的数值存放在TTLInt中，然后调用TTMATH的mul进行运算，计算res.word[0]的长度，并与38进行比较来判断是否溢出，最后设置相应的precision和scale。
Div运算中，将两个Decimal的数值存放在TTLInt中并将被除数乘以10^37（num1 = num1 * kMaxScaleFactor;），然后调用TTMATH的div进行运算，将运算结构存放在res.word[0]中。计算res.word[0]的长度，并与38进行比较来判断是否溢出，最后设置相应的precision和scale。

3.3.2 Decimal逻辑运算

ObDecimal类的逻辑运算的实现类似OCEANBASE数据库中ObNumber类的逻辑运算——重载<、<=、>、>=、==、!=6种逻辑运算符。每种逻辑运算调用ObDecimal类中的sub来实现比较，并返回bool值。

4 总体设计

一个ObDecimal的对象代表一个decimal类型的数值。对于decimal的精度和有效数字位数，我们也同样把他存到ObDecimal类里面。而什么时候把精度和有效数字位数存到ObDecimal里面呢？当然是在插入语句执行的时候。Cedar在执行插入语句的时候，会把用户输入的值转成相应类型的ObObj存起来。如果某一列的类型是decimal，我们会先把用户输入的值转换成一个ObDecimal的对象，然后把该对象存到一个类型为ObDecimalType的ObObj里面。具体的实现函数是int ObExprValues::eval()。在该函数里，我们会获得ObSchemaManagerV2里面存储的该列的数据类型以及用户输入的精度和有效数字位数，在生成相应的ObDecimal对象的时候，把用户输入的精度和有效数字位数存到该对象中去即可。

4.1 词法、语法解析

Cedar 0.2在处理insert语句的时候，首先会对sql语句进行词法解析。对于带有小数点的数值，比如3.14，CEDAR会在词法解析的时候把它解析成T_DOUBLE类型。然后在生成逻辑计划的时候将该值存成C++中的double类型，最后存到ObObj中去。但是如果表的某一列为decimal类型，在插入该列数据的SQL语句解析过程中，将原本用户输入的包含有小数的数值被转成double类型，将会造成精度上的损失。所以我们的实现是在词法解析里面增加了T_DECIMAL类型，对于带有小数点的数值，全被解析成T_DECIMAL类型，为了保留用户输入数据的全部精度，在SQL解析阶段，我们在T_DECIMAL类型中先存储用户输入的数据。
注：科学计数法的数值依然被解析成T_DOUBLE类型。

4.2 生成逻辑计划

MS生成逻辑计划的时候，会将语法树中存的数值转换成中缀表达式，存到逻辑计划树中。对于T_DECIMAL类型的语法树节点，我们首先把它存到ObObj中，然后存到中缀表达式中。

4.3. 生成物理计划

MS生成物理计划的时候，会将中缀表达式转换成后缀表达式，存到物理操作符中。在ob_postfix_expression.cpp中，对于类型为ObDecimalType的中缀表达式，我们把它转换成后缀表达式的代码如下：

```
  case T_DECIMAL:
      item_type.set_int(CONST_OBJ);
      obj.set_decimal_v2(item.dec,item.len);    //add xsl ECNU_DECIM
AL
      if (OB_SUCCESS != (ret = str_buf_.write_obj(obj, &obj2)))
      {
        TBSYS_LOG(WARN, "fail to write object to string buffer. er
r=%d", ret);
      }
      else if (OB_SUCCESS != (ret = expr_.push_back(item_type)))
      {
          TBSYS_LOG(WARN, "fail to push item.type. err=%d", ret);
      }
      else if (OB_SUCCESS != (ret = expr_.push_back(obj2)))
      {
          TBSYS_LOG(WARN, "fail to push object. err=%d", ret);
      }
      break;
```

注：因为Cedar 的内存分配机制比较特殊，此处尚未解决内存分配问题（在Obj析构的时候释放内存依然存在问题）。主要的原因在于解析后缀表达式的时候，需要将后缀表达式中的每个数据解析成obj并准出到expr_中，然后执行相关的后缀表达式。Cedar中对于变长的类型（如varchar）分配了额外的内存空间，因此需要修正这个OBDecimal的问题，需要采用类似的机制。

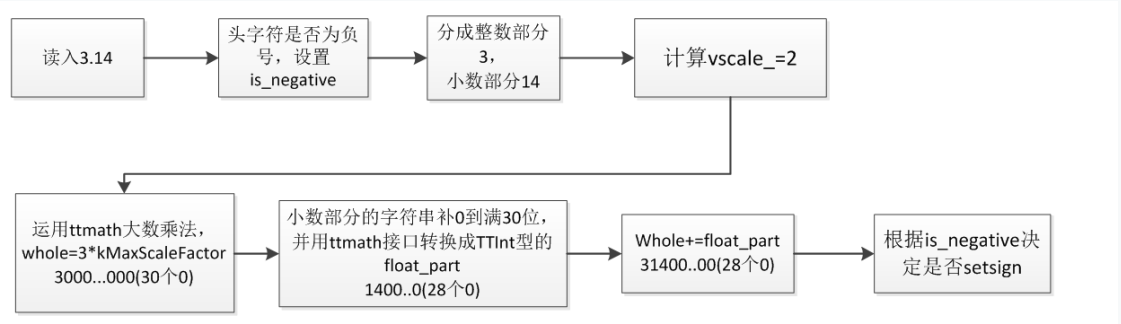
4.4 Decimal类型的转换

对Obj_cast这个函数，需要修改很多地方。下面是具体分析该函数：

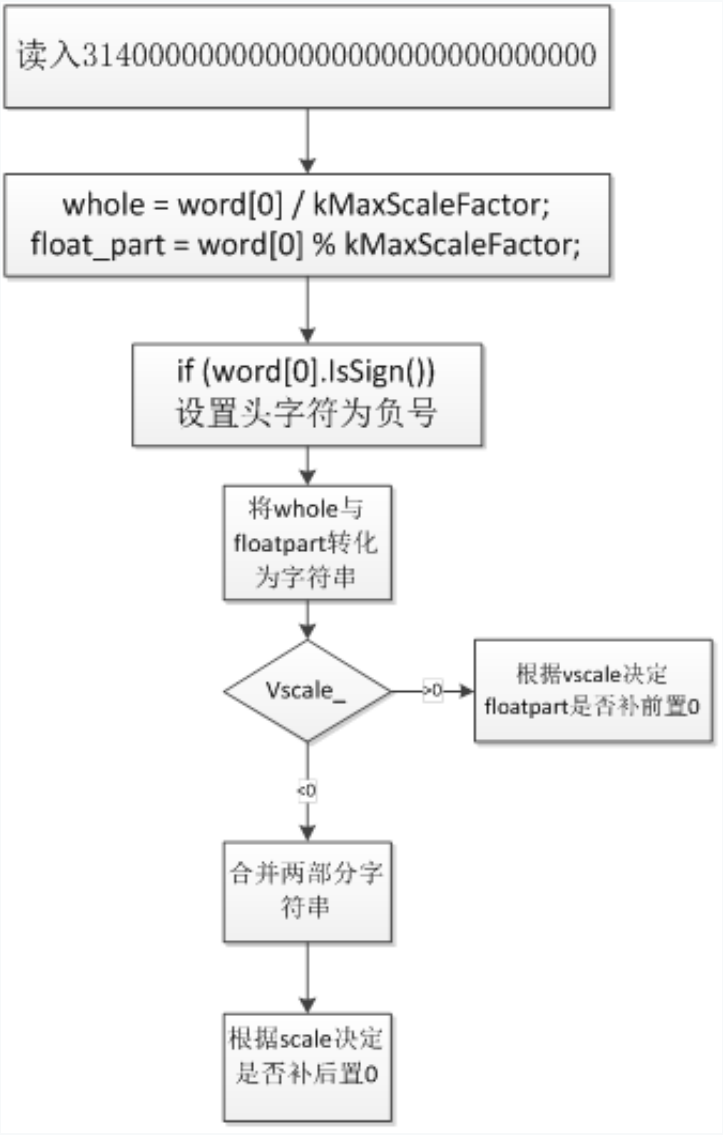
```
int obj_cast(const ObObj &orig_cell, const ObObj &expected_type,ObO
bj &casted_cell, const ObObj *&res_cell)
```

该函数有四个参数，orig_cell表示原来的数据，即后缀表达式中的数据，expected_type表示表定义时的列的类型，casted_cell表示作类型转换的中间量，res_cell表示转换后的结果。Cedar对该函数的实现是：先判断orig_cell和expected_type的类型是否一致，如果一致，则不作转换，如果不一致，将调用相应的转换函数进行转换，最后的结果作为res_cell返回。由于我们对Cedar的词法解析做了修改，所以必须要在obj_cast函数里面做些相应的修改。对于expected_type为double，float,int的，我们要写相应的double_decimal, float_decimal,int_decimal函数来供调用。如果expect_type为ObDecimalType，我们会在Obj_cast函数里根据用户输入的精度和有效数字位数对res_cell进行修改。因为如果用户要插入的数值的小数位数大于用户输入的精度，我们的实现是将用户要插入的数值进行一次截取，将超过精度的小数部分直接截掉，保证存到数据库中的数是截取后的数。

示例：
Varchar 与 Decimal之间的相互转化：从char*到Decimal之间的类型转化函数为from()与tostring()；其中from()接口为from(const char* str, int64_t buf_len)。由char*转化成decimal时，以Decimal(20,7)为例，输入3.14的处理流程如下：



其中to_string()接口为to_string(char* buf, const int64_t buf_len)。由decimal转化成char*时，处理流程如下：



注：float、double与 Decimal之间的相互转化：在做float/double等类型的小数与Decimal之间的转化时，首先通过snprintf函数将小数转化成字符串varchar buf，然后再调用varchar与double之间的转化函数from或tostring进行转化。
类型转换之后，即可以ObObj为单位进行传输、操作，必要的是在进行ObObj的拷贝对Decimal的拷贝需要参照varchar进行额外处理。

4.5 序列化以及存储

Cedar 将数值封装在一个ObObj当中，序列化的时候，首先判断Object的Type，调用对应的序列化函数：

```
DEFINE_SERIALIZE(ObObj)
{
…
ObObjType type = get_type();
 switch (type)
    {
…
 case ObDecimalType:
    ret=serialization::encode_comm_decimal(buf,buf_len,tmp_pos,ob
j_op_flag == ADD,meta_.dec_precision_,  meta_.dec_scale_, meta_.de
c_vscale_,value_.ii,(int32_t)sizeof(uint64_t)*get_nwords());
…
    }
…
}
```

序列化类型为Decimal的Object的时候，我们将Object的Meta中的内容，与TTInt的内容一同序列化。

> 参数解释
> meta_.dec_precision_ ：ObObjMeta中的精度
> meta_.dec_scale_ ：ObObjMeta中的小数长度
> meta_.dec_vscale_：ObObjMeta中的实际小数长度
> value_.ii：指向Decimal数据的指针
> get_nwords()：Decimal占据uint64_t的空间长度个数

因为是用的变长存储，在反序列化的时候依次从buf当中读出char。因此，对于每个Decimal类型的数据会占用掉8*N byte的空间大小（N = get_nwords( )）。