Project 4 Report

1. Overview

In this project, we have implemented iterator based pipeline query execution engine. What's more, we also coordinate indices with tuple inserting and deletion.

2. Design & Implementation

- **ValueStream**

As we know, in this project, we often face with the situation where we have to read/write three different data type into/from byte stream. Different data types should convert into bytes differently. To make things more clearly and universally, I define two class: `AttrValue` and `ValueStream`:

```
class ValueStream{
public:
   ValueStream(void *data = NULL): _data((char *)data), _offset(0){}
   ~ValueStream();
   ValueStream& operator<<(int v);
   ValueStream& operator<<(float v);
   ValueStream& operator<<(string s);
   ValueStream& operator>>(int &v);
   ValueStream& operator>>(float &v);
   ValueStream& operator>>(string &s);
   AttrValue read(const Attribute &attr);
   ValueStream& write(const AttrValue av);
   bool search(const vector<Attribute> &attrs, const string name, AttrValue &av);
};
struct Attribute {
   string   name;     // attribute name
   AttrType type;     // attribute type
   AttrLength length; // attribute length
   Attribute(){}
   Attribute(string _name, AttrType _type, AttrLength _len):
   name(_name), type(_type), length(_len){}
};
```

`ValueStream` is associated with a buffer pointer, all the <</>> operator makes bytes input/output into that bytes buffer. So we can use it like this:

```
ValueStream(data)<<id<<column_name<<column_type<<column_len;
```

This greatly shorts the code length and make the things more clear.

- ***Index Coordination***

In this part of project, we have to coordinate the index operation with RelationManager's operation, namely, when we do tuple inserting/deletion, we have to update the existing associated indices record as well.

The way we maintain the index meta-info is to create another database catalog table, named "index_table". This has following schema:
`{"tableName", "attrName", "indexName"};`

Just as what we did before, we maintain this info and load all index table into in-memory map data structure during RelationManager constructor.

- ***Query Engine***

Each query operator is implemented as an iterator subclass, providing two interface:

```
RC getNextTuple(void *data) = 0;
void getAttributes(vector<Attribute> &attrs) const = 0;
```

`getAttributes()` return attribute descriptor, except that the name of attribute becomes "***`relation.attribute`***". But this does not need extra attention, as all the iterator receive another iterator as input, so the format of attribute name is consistent.

The implementation itself is quite straightforward. Only for join operator we have to remember that, the iterator needs to remember the last retrieved Tuple, because inner relation needs to search for remaining possible join. It's like this:

```
while _last_left != NULL && _right->getNextTuple() != QE_EOF
       ValueStream(rbuf).search(rAttrs, _cond.rhsAttr, rv);
       if isOK(_last_left, rv, OP)
           return output()
while (_left->getNextTuple(lbuf) != QE_EOF)
   _right->setIterator();
   while (_right->getNextTuple() != QE_EOF)
       ValueStream(rbuf).search(rAttrs, _cond.rhsAttr, rv);
       if (AttrValue::compareValue(lv, rv, _cond.op))
           return output();
```

- ***Aggregate Operator***

To earn extra credits, I also implement aggregate operator. The way to achieve that is like this: before calling ***`getNextTuple(),`*** we first scan all the data, and do aggregation computation. So the whole job leaving for ***`getNextTuple()`*** is just returning the computed results, one by one.

If the `groupby` attribute is specified, we have to store aggregated value into different map-based slot, the key of may is `groupby` attribute value.

3. Acknowledgement

Finally we finished all four projects, and after getting my hands dirty, now we could proudly say, database is no longer a black-box for me!
Thanks all of you for these wonderful class, it’s the best class I’ve ever took! :)