Project 4 Report / Group 16

In this project, we implemented filter, project and joins. We also implemented the aggregate interface. Next we'll describe how we implemented these classes.

Filter Class

For this class, when getNextTuple is called, we use the given iterator to get a tuple. If the tuple matches the filter condition, then we'll output the tuple. If it doesn't match the condition, we get one more tuple until we find one that matches the filter condition. When the given iterator returns a QE_EOF, that means there are no more tuples in the iterator, so we'll return a QE_EOF.

Project Class

Project class selects certain attributes from a tuple. It's core design is that when processing a tuple getting from input iterator, it scans each attribute of this tuple. If the attribute belongs to the required attributes group, copy this attribute into output buffer. If not, discard it. In such a way, each tuple will get checked to make sure that only required tuples will be sending out.

Nested-Loop Join Class

Nested-Loop Join takes two input, leftIn and rightIn. The three member functions will be like this:

void getAttributes(vector<Attribute> &attrs) const ---- the attributes of NLJoin will be the attributes concatenation of the left iterator and right iterator.

NLJoin::NLJoin(Iterator *leftIn, TableScan *rightIn, const Condition &condition, const unsigned numPages) ---- The constructor will record information like the leftIn iterator attributes, the rightIn iterator attributes, the attribute of condition, etc. We will also create variables to keep track of the current tuple location.

RC getNextTuple(void *data) ---- We use the simple way to get next tuple. Firstly, get tuple A from leftIn. Then compare each tuple from rightIn with A. If it satisfies the comparing condition, this tuple will be one of the output.

Index Nested-Loop Join Class

Index Nested-Loop Join is quite similar with Nested-Loop Join. The differences include the following:

1.The rightIn iterator in INLJoin is a IndexScan. Which means we can find the corresponding tuple in constant time. Which greately reduces the times of comparision.

2. The Index Nested-Loop does not suppport string type.

The design of RC getNextTuple(void *data) is almost the same with NLJoin. When we have a tuple from leftIn, we compare it with tuples from rightIn that satisfy a certain condition – this time we will not compare one by one.

Thus we can get a remarkable improvement from the index.

HashJoin Class

We used the give number of memory pages minus one as our number of partitions, and we used it to design our own hash function. After we build the partitions, we can use them to do getNextTuple.

Probing Phase:

We use probingPhase() function to implement getNextTuple(void *data).After partitioning, we have pageNum partitions. And within each partition pair(left partition and right partition), we firstly build a hash table (using unordered_mutilpmap) based on the left partition in memory. Then we read one tuple from right partition, find its corresponding tuple, which satisfy certain given condition, in this hash table. This can be done in constant time.

To facilitate our probing, we create a class called ReadPartition. It helps us read tuples from a partition one by one. And it keeps records of the current position and some other useful information.

Aggregate – not include “group by”

When doing a aggregate SQL query , we adapt such a design --- Firstly create five float type member variables(sum, count, min, max, avg). Secondly scan all the input iterator, and during this process, we keep modifying these five member variables. For example, after scanning one tuple, the count will add one. In this way, we can get the final results after just one scan.

Based on what kind of aggregate operation, we output the corresponding member variable.

Aggregate – include “group by”

Group by is very tricky because we need to divide tuples into groups according to gAttr(group Attribute). Within each group, we need to keep records of the aggregate information(sum, count, min, max, avg) according to another attribute(aggAttribute, aggregation attribute).

This is our design. Firstly, we create a class called AggrCon (Aggregation Content) to keep information for each group. It has five member variables, sum, count, min, max, avg.

Then we use a hash table to help us divide tuples into groups. We use the unordered_map for this implementation. Because the gAttr may have different attrType. The pair of this hash table may be <int, AggrCon>, <float, AggrCon> and <string, AggrCon>.

With the help of a hash table, every time we find a new tuple that belong to a group, instead of searching the groups one by one we can find this group in constant time.

Thus the implementation of group by is, firstly we scan the input iterator. Based on the aggregation attribute, we build hash table for the input. If there has not been a bucket for a tuple with a certain attribute value, we create a new bucket for this tuple with the certain attribute value. And if we find a new tuple that belonging to the same bucket with some former tuples, we update the AggrCon inforamtion of this bucket.

In this way, after one time of scanning, we can divide the tuples into groups based on the aggrAttribute and at the same time calculate the aggregation values for each group.

With all the results we've got, it is very easy to format the output. Since there is not a “order by” command required, the output can be unordered.