Database final project report

- **Team ID:** group 15
- **The IDs and names of team members**
- **The basic information of the paper**
    1. Title: Optimized Stratified Sampling for Approximate Query Processing
    2. Conference ACM Transactions on Database System
    3. published years: June 2007
    4. authors: SURAJIT CHAUDHURI , GAUTAM DAS , GAUTAM DAS
- **The main idea of the paper**

  Traditional uniform sampling methods are not good enough doe to the following two disadvantages:
    1. First, avoiding large errors on an arbitrary query, especially for queries with relatively low selectivity, is virtually impossible.
    2. Second, uniform random sampling ignores variance in the data distribution of the aggregated column(s).

  to overcome these two disadvantages, the paper proposes a stratified sampling technique.

  This technique consists of two parts:
    1. The offline part using a defined workload to divide records into regions and generate samples.
    2. The online part that handles incoming querries.

  The first part consists of five main steps:
    1. Tagging each record according to whether or not they've been used to answer each query, the purpose of this is to identify fundamental regions which can be done by sorting the records by their tags.
    2. Compute alpha $_{j,Q}$ and alpha$_j$ using

       $$Let \quad \alpha_{j,Q} = \frac{\sum_{R_j \subseteq R_Q} n_j^2 \delta(1-\delta) + \sum_{R_j \subseteq R \setminus R_Q} n_j^2 \gamma(1-\gamma)}{(\sum_{R_j \subseteq R_Q} \delta n_j + \sum_{R_j \subseteq R \setminus R_Q} \gamma n_j)^2}$$

       $$\alpha_j = \sum_{i=1}^{q} w_i \alpha_{j,Q_i}.$$

       which will be used in the next step, parameters gamma and delta can be set by the administrator, they are used to describe the similarity between the workload and incoming queries.

Boundary Settings	Interpretation
$\delta = 1$ and $\gamma = 0$	incoming queries are identical to workload queries
$\delta = 1$ and $\gamma > 0$	incoming queries are supersets of workload queries
$\delta < 1$ and $\gamma = 0$	incoming queries are subsets of workload queries
$0 \leq \gamma = \delta \leq 1$	incoming queries (with expected selectivity δ $(= \gamma)$) are uncorrelated with the workload queries

    3. Calculate the number of records that should be sampled in each region k which can be calculated using the following equation

       LEMMA 7. *$\sum_{1 \leq j \leq r} \alpha_j / k_j$ is minimized under the constraint $\sum_{1 \leq j \leq r} k_j = k$ by setting $k_j = k \left(\sqrt{\alpha_j} / \sum_{1 \leq i \leq r} \sqrt{\alpha_i}\right)$.*

       It has been proven in the paper that by setting k using this equation the errors for SUM and Count aggregation can be minimized

       Approximate error for count

$$ApproxMSE(p_{(Q)}) = \frac{\sum_{R_j \subseteq R_Q} \frac{n_j^2}{k_j}\delta(1-\delta) + \sum_{R_j \subseteq R \setminus R_Q} \frac{n_j^2}{k_j}\gamma(1-\gamma)}{\left(\sum_{R_j \subseteq R_Q} \delta n_j + \sum_{R_j \subseteq R \setminus R_Q} \gamma n_j\right)^2}.$$

Approximate error for sum

$$ApproxMSE(p_{(Q)}) = \frac{\sum_{R_j \subseteq R_Q} \frac{n_j^2}{k_j}y_j^2\delta(1-\delta) + \sum_{R_j \subseteq R \setminus R_Q} \frac{n_j^2}{k_j}y_j^2\gamma(1-\gamma)}{\left(\sum_{R_j \subseteq R_Q} \delta n_j y_j + \sum_{R_j \subseteq R \setminus R_Q} \gamma n_j y_j\right)^2}.$$

4. Perform stratified sampling to pick k_j record from region R_j and generate a sample of R. Then use this samples and aggregation functions to compute the estimated answer

- **A brief summarization of what you implement in VanillaDB**

  In the item table, a new field *record_tag* is added to keep track of the queries that used the record.

  Another table *sample_item.tbl* is also constructed for aggregation functions to retrieve samples from them quickly.
  Finally, a table *region.tbl* is also maintained to store the *r_id*(record id) *ni*(number of records),*ki*(number of records to be sampled).

  A class *Workload* is implemented as an instance of the workload used to divide the records into regions. It contains a set of 18 queries as the workload and 18 queries as incoming queries, and they have 9 same query, which have 50% similarity.

```
public static final String[] workload = {
        "SELECT COUNT(i_im_id) FROM item WHERE i_im_id < 4967 and i_im_id > 1611",
        "SELECT COUNT(i_id) FROM item WHERE i_id < 4036 and i_id > 3626",
        "SELECT SUM(i_im_id) FROM item WHERE i_im_id < 7233 and i_im_id > 6239",
        "SELECT SUM(i_price) FROM item WHERE i_price < 32 and i_price > 27",
        "SELECT COUNT(i_price) FROM item WHERE i_price < 16 and i_price > 8",
        "SELECT COUNT(i_price) FROM item WHERE i_price < 99 and i_price > 76",
        "SELECT SUM(i_im_id) FROM item WHERE i_im_id < 506 and i_im_id > 72",
public static final String[] query = {
        "SELECT COUNT(i_im_id) FROM item WHERE i_im_id < 4967 and i_im_id > 1611",
        "SELECT COUNT(i_id) FROM item WHERE i_id < 4036 and i_id > 3626",
        "SELECT SUM(i_im_id) FROM item WHERE i_im_id < 7233 and i_im_id > 6239",
        "SELECT SUM(i_price) FROM item WHERE i_price < 32 and i_price > 27",
        "SELECT COUNT(i_price) FROM item WHERE i_price < 16 and i_price > 8",
        "SELECT COUNT(i_price) FROM item WHERE i_price < 99 and i_price > 76",
        "SELECT SUM(i_im_id) FROM item WHERE i_im_id < 506 and i_im_id > 72",
```

  In *Planner.java* two methods are implemented to handle sample queries

```
public int executeRecordTag(String qry,Transaction tx,int i) {
    Parser parser = new Parser(qry);
    QueryData data = parser.queryCommand();
    Verifier.verifyQueryData(data, tx);
    BasicSamplePlanner bsp = (BasicSamplePlanner)sPlanner;
    return bsp.tag_record(data, tx,i);
}
```

A new planner class *BasicSamplePlan* that implements *QueryPlanner* is implemented and contains method tag_*record* which tags the record field of that record, setting the i'th bit to one.

```
public int tag_record(QueryData data, Transaction tx,int i) {
    //i mean sql is i-th element in workload Set
    List<Plan> plans = new ArrayList<Plan>();

    boolean hasCf = false;
    boolean hasSf = false;
    Set<String> proField = new HashSet<String>();
    proField.add("RECORD_TAG");
    data.projFields = proField;
    for(AggregationFn af:data.aggregationFn()) {
        if(af.getClass().equals(CountFn.class))
            hasCf = true;
        if(af.getClass().equals(SumFn.class))
            hasSf = true;
    }

    for (String tblname : data.tables()) {
        String viewdef = VanillaDb.catalogMgr().getViewDef(tblname, tx);
        if (viewdef != null)
            plans.add(VanillaDb.newPlanner().createQueryPlan(viewdef, tx));
        else
            plans.add(new TablePlan(tblname, tx));
    }
    // Step 2: Create the product of all table plans
    Plan p = plans.remove(0);
    for (Plan nextplan : plans)
        p = new ProductPlan(p, nextplan);
    // Step 3: Add a selection plan for the predicate
    p = new SelectPlan(p, data.pred());

    UpdateScan us = (UpdateScan) p.open();
```

```
us.beforeFirst();
int count = 0;
while (us.next()) {
    if(hasCf || hasSf) {
        String record_tag = (String)us.getVal("record_tag").asJavaVal();
        Boolean [] b_map = str2BooleanArray(record_tag);
        b_map[i] = true;
        record_tag = booleanArray2Str(b_map);
        Constant c = Constant.newInstance(Type.VARCHAR(200),record_tag.getBytes());
        us.setVal("record_tag",c);
        //TODO:set bit map to 1
    }
    count++;
}
//ps.close();
us.close();
//VanillaDb.statMgr().countRecordUpdates(data.tableName(), count);
return count;
```

A class *Region* is implemented to store the information of each region including *r_id ,alphaJ,alphaJQ,ki*

```java
public class Region{
    private Vector<Integer> recordIds;
    public double alphaJ;
    public int r_id;
    public double[] alphaJQ;
    private int ki;
    //private Strisng record_tag;
    public Region(int r_id) {
        this.r_id = r_id;
        this.recordIds  = new Vector();
        this.alphaJQ = new double[workload_size];
        for(int i=0;i<workload_size;++i)
            this.alphaJQ[i] = 0;
    }
```

I also modified the method *executeSql*(),calling methods tag_*record*() and *sample*()

```java
//TODO:tag_record
logger.info("Start tag record.");
queryCount = tag_record();
logger.info("Start sampling.");
//TODO:sample
sample();
```

Method tag_*recod* retrieves *queryCount, which indicated how many record the workload query select.*

```java
private int[] tag_record() {
    int[] counts = new int[workload_size];
    for(int i=0;i<workload_size;++i) {
        String sql = wl.workload[i];
        counts[i] = tag_record(sql,i);
    }
    return counts;
```

Method sample is the process of creating regions, calculating *alphaJ, alphaJQ,* and k_i which represents the number of records that should be sampled in each region.

```java
private void sample() {
    Scan recordTagScan = sample("SELECT RECORD_TAG, i_id FROM item");
    recordTagScan.beforeFirst();
    int r_id = 1;
    while(recordTagScan.next()) {
        String record_tag = (String) recordTagScan.getVal("record_tag").asJavaVal();
        Integer i_id = (Integer) recordTagScan.getVal("i_id").asJavaVal();
        if(Regions.containsKey(record_tag)) {
            Regions.get(record_tag).addRecord(i_id);
        }else {
            Region temp = new Region(r_id);
            temp.addRecord(i_id);
            Regions.put(record_tag,temp);
            r_id +=1;
        }
    }
    recordTagScan.close();
    region_size = Regions.size();
    System.out.println("[MicroTestbedLoad]"+region_size);
    //TODO:calculate the alpha(J,Q)
    calAlphaJQ();
    //TODO:calculate alphaJ
    calAlphaJ();
    //TODO:calculate ki
    calKI();
    //TODO:sample ki records and store in table
    logger.info("Start creating sample table.");
    createSampleTbl();
    logger.info("Finish creating sample table.");
    //TODO:create region table
    createRegionTbl();
```

Method *createRegionTable* constructs a table for regions, containing fields *r_id* (record id), ni(number of records inside a region), k_i(the number of samples that should be taken from that region)
Method createSampleTable constructs a table for samples, selecting k_i samples from each region.

```
private void createRegionTbl() {
    executeUpdate("CREATE TABLE region (r_id INT, ni INT, ki INT)");
    for(Object k: Regions.keySet()) {
        String key = (String)k;
        int r_id = Regions.get(key).r_id;
        int ni = Regions.get(key).getNi();
        int ki = Regions.get(key).getKi();
        executeUpdate("INSERT INTO region(r_id, ni, ki) VALUES (" +r_id + ", " + ni + ","
                + ki + ")");
    }
}
private void createSampleTbl() {
    executeUpdate("CREATE TABLE sample_item (i_id INT, i_im_id INT, i_name VARCHAR(24), "
            + "i_price DOUBLE, i_data VARCHAR(50),r_id INT)");
    for(Object k:Regions.keySet()) {
        String key = (String)k;
        int ki = Regions.get(key).getKi();
        String sql = "SELECT i_id,i_im_id,i_name,i_price,i_data FROM item WHERE record_tag = '"+key+"'";
        Scan s = executeQuery(sql);
        s.beforeFirst();
        int count =0;
        while(s.next()) {
            if(count>=ki)
                break;
            int iid = (int)s.getVal("i_id").asJavaVal();
            int iimid = (int)s.getVal("i_im_id").asJavaVal();
            String iname = (String)s.getVal("i_name").asJavaVal();
            double iprice =(double)s.getVal("i_price").asJavaVal();
            String idata = (String)s.getVal("i_data").asJavaVal();
            executeUpdate("INSERT INTO sample_item(i_id, i_im_id, i_name, i_price, i_data,r_id) VALUES (" + iid + ", " + iimid + ", '"
                    + iname + "', " + iprice + ", '" + idata + "', "+Regions.get(key).r_id+")");
            count++;
        }
        s.close();
```

Aggregation functions for Count and Sum are implemented in *CountFn.java* and *SumFn.java*. We can know whether we select record from sample table by the isSample flag. According the isSample flag, we can use precomputed values *r_id ,ki* and *ni, to compute the result of aggregation functions.*

```
public void processFirst(Record rec) {
    if(isSample) {
        int r_id = (int)rec.getVal("r_id").asJavaVal();
        double ki = (double)VanillaDb.getRegions().getKibyId(r_id);
        double ni = (double)VanillaDb.getRegions().getNibyId(r_id);
        count = ni/ki;
        System.out.println("[CountFn] "+r_id+" "+ki+" "+ni);
    }else
        count = 1;
}

@Override
public void processNext(Record rec) {
    if(isSample) {
        int r_id = (int)rec.getVal("r_id").asJavaVal();
        double ki = (double)VanillaDb.getRegions().getKibyId(r_id);
        double ni = (double)VanillaDb.getRegions().getNibyId(r_id);
        count +=ni/ki;
    }else
        count++;
}
```

```java
public void processFirst(Record rec) {
    if(isSample) {
        int r_id = (int)rec.getVal("r_id").asJavaVal();
        int ki = VanillaDb.getRegions().getKibyId(r_id);
        int ni = VanillaDb.getRegions().getNibyId(r_id);
        DoubleConstant dKi = new DoubleConstant((double)ki);
        DoubleConstant dNi = new DoubleConstant((double)ni);
        Constant c = rec.getVal(fldName);
        this.val = c.castTo(DOUBLE).mul(dNi).div(dKi);
    }else {
        Constant c = rec.getVal(fldName);
        this.val = c.castTo(DOUBLE);
    }
}

@Override
public void processNext(Record rec) {
    if(isSample) {
        int r_id = (int)rec.getVal("r_id").asJavaVal();
        int ki = VanillaDb.getRegions().getKibyId(r_id);
        int ni = VanillaDb.getRegions().getNibyId(r_id);
        DoubleConstant dKi = new DoubleConstant((double)ki);
        DoubleConstant dNi = new DoubleConstant((double)ni);
        Constant newval = rec.getVal(fldName).castTo(DOUBLE).mul(dNi).div(dKi);
        val = val.add(newval);
    }else {
        Constant newval = rec.getVal(fldName);
        val = val.add(newval);
```

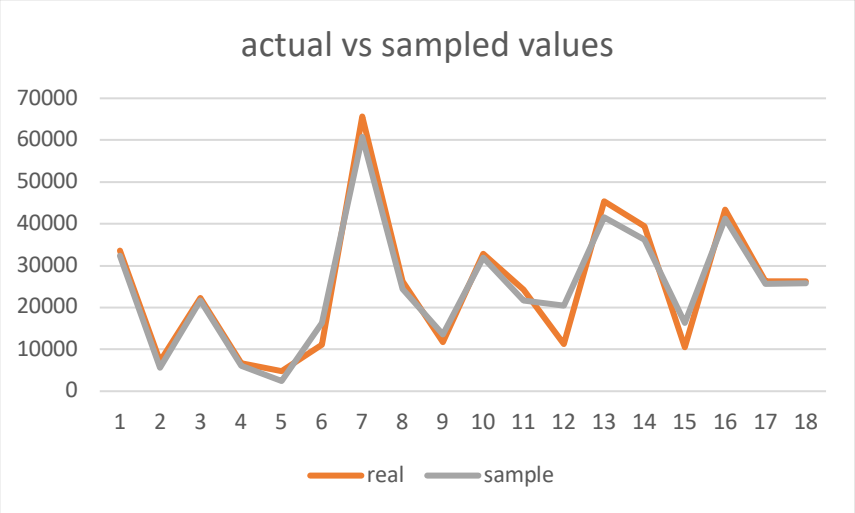
When a user wants to get an aggregated value using sampling instead of the actual value, simply add “sample” in the beginning of the query

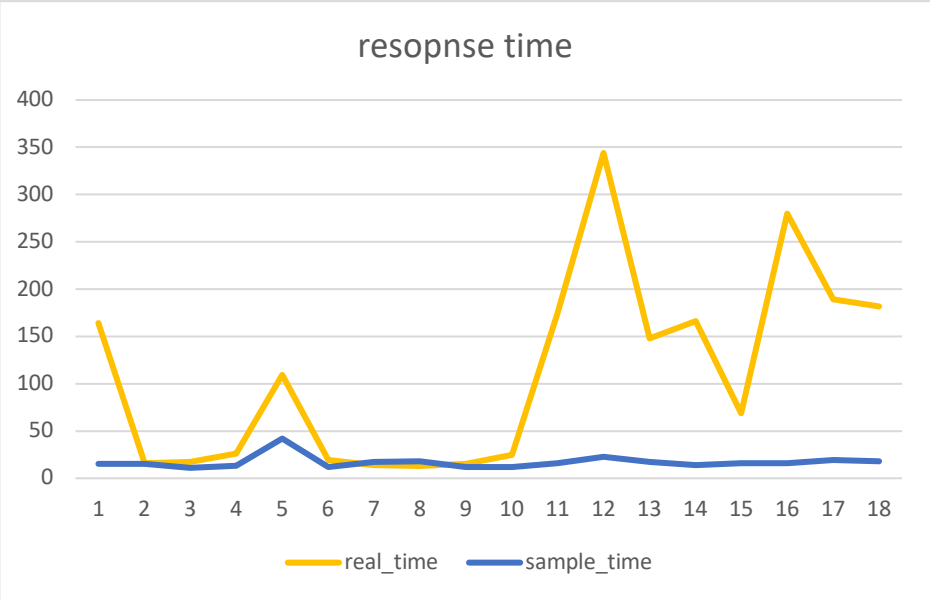
```
SQL> sample SELECT COUNT(i_im_id) FROM item WHERE i_im_id < 4967 and i_im_id > 1611
```

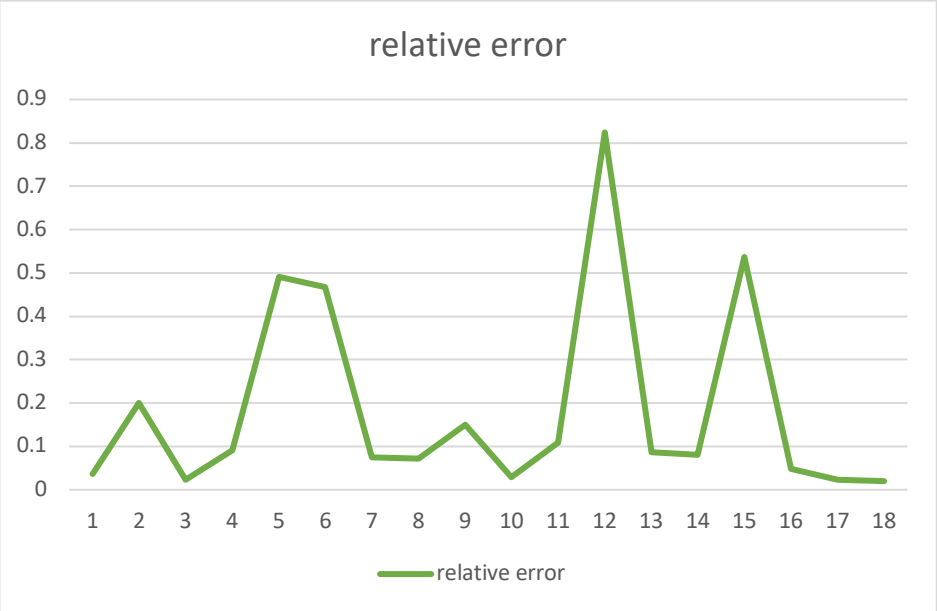
This part is done the same way as in HW3.

- **Your evaluation and experiments**
    1. Comparison of the performance with and without sampling
    2. The correctness of queries in and out the workload
    3. Setting different alpha and delta parameters
- **An analysis for your experiments**

  Using tagging and sorting the records, we are capable of dividing the records into about 340 regions, with each region containing approximately 40 records.
  We sample approximately 25000 records.

1. 





As shown in the above results, the results using sample have curves almost identical to the actual values, and we are able to keep the relative error bellow 0.5. However, using this sampling Technique allows us significantly reduce the response time in some cases.
Proving the usefulness of this technique.

2. In the above results, the first 9 queries are the same as that of the workload, and the remaining 9 aren't in the workload, contrary to our expectations the latter 9 queries don't exhibit larger errors.
3. We also tested on different gamma and delta settings, however the results are identical for all settings, only the response time slightly differ which is most due to the running environment, not the parameters. This may be caused by a lack of a specific pattern in our workload and incoming queries.

- **The problems you occur during implementation and how you solve them**
    1. Reconstructing the regions and computing the values of n_i k_i takes too much time, so a method *init_RegionTable()* is called to retrieve the value for the regions. However, this results in needing to manually comment out this call the first time, since no region table is retrievable in the beginning.
    2. In ours incoming query, we randomly choose a query from Workload.java file. We unable to use store procedures due to lack of metadata, so we don't use store procedures.
       We use SQL Console to test our error and response time.

- **conclusion**
  In addition to significantly reducing response time, using stratified can provide a more accurate estimate due to avoiding large errors and taking the variance in the data distribution of the aggregated column into account. Both are unable to be accomplished using traditional unified sampling.
  This technique is most likely more suitable for databases that have a specific pattern in incoming queries, allowing them to use previous queries as workload to divide the records into regions to perform stratified sampling.
  Though in is not apparent from our experiment, the parameters alpha and gamma represent the similarity between the incoming queries and the workload, since these two parameters influence the sampling, the administrator should recognize the patterns of incoming queries of the database to set the two parameters.