Database Systems

Course instructor: Yanif Ahmad.
Course staff: P.C. Shyamshankar (Shyam) (TA).
Schedule: MW 12-1.15pm, Hackerman 320.

Office hours:
Yanif: T 5:00-7pm, Malone 328.
Shyam: T 5:00-7pm, Malone 328.

Assignment 3: Query Optimization

Due date: 11:59pm Friday April 24th, 2015

Description | Logistics | Exercises | Relevant lectures

Description

In this third assigment, we ask you to implement a basic query optimizer that picks an efficient plan for our query processing engine. In particular, we will implement a System-R style query optimizer that uses dynamic programming, with sampled query execution for cost estimation. This assignment will build on the operator algorithms and query plans used in Assignment 2. Recall that our query plans are tree data structures, comprised of operator nodes. Our query optimizer will consider alternate tree plans that are guaranteed to produce the same query results, picking the lowest cost plan to execute and answer the query. These alternate plans are derived by considering algebraic equivalences for the relational algebra. Specifically in this assignment, you will consider plan rewrites that reorder joins, and push down selection and project operations.

Our support code for this assignment provides the definition and API for the query optimizer class, as well as a working implementation of the query processor you saw in Assignment 2. Furthermore, we have extended this query processor to support **sampling** (completed for you), and **cost computation** (API provided for you to fill in). You will continue to construct query plans using our `PlanBuilder` class. This plan must be optimized with the query optimizer prior to query execution. Given a query plan as its input, the query optimizer will construct a new plan for actual use during execution.

By the end of this assignment, you will have a **declarative** database engine that introspects on the queries to improve system efficiency and scalability. We will use this engine in the final homework for our parallel and distributed DBMS implementation.

As with Assignment 1 and 2, we will use the lab sessions to discuss the design of the codebase, and the concepts behind our query engine implementation. **We encourage you to work in pairs on this exercise, and get started early on this assignment (we estimate you will be writing approximately 500-1000 lines of code).** We strongly recommend you think through the design and sketch out pseudocode rather than diving straight into the exercises. You are welcome to ask design questions on Piazza. Try to have Exercise 1 and 2 completed by the end of the first week.

Logistics

Support code The Python support code for this assignment can be found at **`~cs416/handouts/hw3/dbsys-hw3.tar.gz`** on the ugrad network. In addition to a sample solution to the query processing exercise, the new support code includes an **Optimizer** class, as well as extensions to the **Plan** and **Operator** that we detail below:

```
dbsys-hw3/              ## Source code directory
 |-- Database.py        ## Top-level database API
 |-- Catalog/           ## Catalog as in HW1 and 2
 |-- Storage/           ## Storage engine as in HW1 and 2
 |-- Query/
 |---- Plan.py          ## Query plans, NEW: sampling API
 |---- Operator.py      ## Operator base class, NEW: costing API
 |---- Optimizer.py     ## NEW: optimizer class skeleton
 |---- Operators/       ## Operator implementations directory
 |------ TableScan.py   ## Scan operator, using file-based iterators
 |------ *.py           ## Select, Project, Union, Join, and Group-By operator implementations.
```

Creating and modifiying query plans: Recall from Assignment 2 on our use of the `Query.Plan.PlanBuilder` class to construct query plans as `Query.Plan.Plan` objects. We can write the following code to create a select-from-where query with a database instance `db`:

```
>>> query1 = db.query().fromTable('employee').where("age < 30").finalize()
```

You may want to revisit the Assignment 2 handout to refresh on the parameters and APIs for plans and operators. In this exercise, in order to implement optimizer rewrite rules, you will be directly creating `Query.Operator.Operator` objects from existing ones.

Query optimization The `Query.Optimizer.Optimizer` class provides the following API:

- **`optimizeQuery(self, plan)`**

  This is the entry point for the optimizer. Given a `Plan` object as its input, this method should return an optimized plan. This method should call the two rewrite algorithms below to first perform operator pushdown, before picking the join order.

- **`pushdownOperator(self, plan) -> plan`**

  This method is one of the primary components of your query optimizer, and is responsible for pushing down unary operators, such as selection and projection.

- **`pickJoinOrder(self, plan) -> plan`**

  This method is the second primary component of your query optimizer, and is responsible for picking the best join ordering possible for the given plan.

- **`getPlanCost(self, plan)`**

  In order to get reuse of optimization computation across multiple queries, you should cache the estimated costs of the plans you evaluate. This method should retrieve the stored estimated cost of a plan from the cache, with appropriate handling if the plan isn't in the cache.

- **`addPlanCost(self, plan)`**

  A counterpart to the above method, this method should update the estimated cost of a plan in the optimizer's cache, adding it if it isn't already present.

Rewrite Rules and Expression API : In order to determine which rewrite rules can be applied, we require the ability to inspect the Python expressions that parameterize our Operators. For example, consider the following query over two relations `R(r int, x1 int)`, and `S(s int, x2 int)`:

```
SELECT * FROM R JOIN S ON x1 == x2 WHERE r > 10 and s == 0 .
```

A naive query plan may first join R and S, and then filter out results using the expression `'r > 10 and s == 0'`. A better plan would involve decomposing the expression into two parts: `'r > 10'` and `'s == 0'`, and applying the predicates to each relation separately, before joining the results. We provide new functionality for analyzing and decomposing expressions in this assignment.

The `Utils.ExpressionInfo` class provides the following API:

- **`ExpressionInfo(expr)`**

  The constructor for the ExpressionInfo class takes a single string, *expr*, as an argument. *expr* is a Python expression (string) that will be evaluated by one of the Operators in the database (via a call to `eval()`).

- **`getAttributes(self)`**

  This method returns a list of attributes mentioned in the expression that was used to construct the `ExpressionInfo` object. For example, `ExpressionInfo('x == 10 and y == 100').getAttributes()` will return `['x', 'y']`.

- **`isAttribute(self)`**

  This method determines whether the expression simply refers to an attribute of a relation, without any modification. For example `ExpressionInfo('x').isAttribute()` will return `True`, while `ExpressionInfo('2*x').isAttribute()` and `ExpressionInfo('x > 5').isAttribute()` will return `False`.

- **`decomposeCNF(self)`**

  All of the expressions used to construct `Project` and `Select` Operators are expected to be in Conjunctive Normal Form (CNF). This method decomposes the original expression into its components, returning a list of sub-expressions. The original expression can be reconstructed by joining all sub-expressions with logical 'and's. For example, `ExpressionInfo('x > 10 and y == 5 and (z > 2 or t == 100)').decomposeCNF()` will return `['x > 10', 'y == 5', '(z > 2 or t == 100)']`.

Sampling We have extended the `Plan` `Operator` and `TableScan` classes to support query sampling. Our `Plan` class now provides a **`sample(self, sampleFactor)`**. The **sampleFactor** parameter indicates the divisor to achieve desired sample size. That is:

```
sample factor = relation size / desired sample size
```

So, for a 10% dataset sample, you should use a sampleFactor of 10.0, and 4.0 for a 25% sample, and so forth. Each operator now tracks whether it is asked to execute in *normal* mode or *sampling* mode via the `Operator.sampled` boolean (also note the presence of `Operator.sampleFactor`). We also provide a `Operator.useSampling(self, sampled, sampleFactor)` method to toggle between normal and sampling mode with the given sample factor.

To run a query plan in sample mode, see the `Plan.sample(self, sampleFactor)` method. This first calls `Operator.useSampling` recursively throughout the tree to enable sampling mode, iterates through all query results produced by the tree tracking the number of results, and then resets the plan back to normal mode. The `Plan.sample` method returns the estimated query result cardinality.

For simplicity, we have kept our query sampling algorithm as a naive per-table sampling method. This independently samples each table in the query plan. Here, we include a table page as a scan output page by applying per-page sampling in the scan operator implementation. This can be seen in the `TableScan` operator as:

```
# Next method implementation of the scan operator's iterator.
def __next__(self):
  if self.sampled:
    return self.sampledOutput() # Sample an output page.
  else:
    return self.nextOutput() # Return the page in normal processing mode.

# Page-based sampling at a scan operator. (Simplifed for the handout, see the full source)
def sampledOutput(self):
  if self.pagesToSample > 0:
    pageId, page = None, None
    while pageId is None and page is None:
      pr = ...    # Compute inclusion probability for this page based on self.sampleFactor
      pageId, page = self.nextOutput()
      if pr >= random.random():
        pageId, page = None, None
    self.pagesToSample -= 1
    return (pageId, page)
  else:
    raise StopIteration
```

Query statistics (cardinality, selectivity, cost) Both query processing modes, sampling and normal, use operator-level counters to keep track of query statistics, in the `Operator.estimatedCardinality` and `Operator.actualCardinality` fields. These cardinalities represent the **output** cardinality at the operator. All other query metrics are derived from these cardinalities. Additionally, we have defined `Operator.tupleCost` as the constant-valued per-tuple processing cost. Each operator can customize this as necessary for its cost model. Each `Operator` now has the ability to compute three query metrics via the following functions. For each of these methods, the `estimated` parameter indicates whether to ask for an estimated statistic, or the profiled statistic. The estimated statistic is only available after the query has been run in sampling mode, and the profiled only after it has been run in normal mode.

- **`cardinality(self, estimated)`**: this method returns the output cardinality of the current operator. For the estimated cardinality, this is scaled by the scale factor used during the last sampled run.
- **`selectivity(self, estimated)`**: this method returns the selectivity as the ratio of this oeprator's outputs, relative to the total input cardinality.
- **`cost(self, estimated)`**: this method returns the cumulative cost of the subplan rooted by the operator. This is calculated by summing up all descendants' costs, as well as the local cost of the operator. The default local cost is a product of the number of inputs the operator has processed, and the per-tuple processing cost (given by the `tupleCost` field).

While we have provided default implementations of these methods in the base `Operator` class, each operator implementation (e.g., `Join`, `GroupBy`) is free to override these methods with a more detailed model. We return to this in the exercises below.

Exercises

CS 316 students should implement exercises **1-3 and 5**. You can receive extra credit for exercises **4 and 6**.
CS 416 students should implement exercises **1-5**. You can receive extra credit for completing exercise **6**.
Point totals for this homework are 100 points for CS 316, and 125 points for CS 416 (excluding extra credit).

1. Push down optimization (30 points)

For this exercise, you'll be implementing the `Optimizer.pushdownOperator` method. This method should rearrange the unary operators (selection, projection) so that they occur as close to their respective base relations as possible. However, you'll also need to ensure that the attributes referred to in join predicates but are not explicitly requested by the query are not projected out before the join actually occurs.
Please use the following assumptions when implementing push down optimization:

- Attribute names are unique, and appear in only one relation in a plan.
- All 'Select' expressions are in Conjunctive Normal Form
- Operators do not need to be pushed down 'through' a GroupBy Expression

2. Join order optimization (30 points)

The main join-selection algorithm to be implemented for this assignment is a System-R-style dynamic programming algorithm. You only need to consider left-deep plans for this particular exercise, a subsequent exercise will ask you to additionally consider bushy plans. To recap, the System-R optimization algorithm builds up a table of the best join orderings and join methods by cost, successively for each sized subset of relations in a query. That is, it first determines the best access methods for each relation, the best join ordering and algorithm for each pair of relations using the access methods previously computed, and so on.

This algorithm consists of two steps during each pass -- an enumeration of viable candidate plans for the given subsets of relations, and an evaluation of the best plan in each subset. For a left-deep-only optimizer, you only need to consider plans where the right-hand-side operand to the join is a base relation, with potential unary operators attached.

The determination of the best plan for each subset of relations should be done by evaluating the plan over a sampled portion of the dataset, and estimating the cost of that plan. The runtime cost of the optimizer can be mitigated by caching cost estimations across queries over the same tables.

The complexity of your optimizer will depend in large part on the assumptions you can make about the plan given to the optimizer. For the purposes of this assignment, you may make the following assumptions:

1. Joins are only ever between two relations, and join expressions only refer to attributes from the two relations being joined.

If you wish to make further assumptions, feel free to run them by us.

3. Cost model design (15 points)

The default cost model for an operator uses the following formula:

```
cost = total inputs from children * per-tuple cost
```

This is not a very accurate cost formula for join operations, and does not distinguish the various join algorithms (nested loops, block nested loops or hash join) present in the codebase. This is similarly true for the group-by operator. In this exercise, we ask you to overload the `Operator.cost` method for the join and group-by operations, to provide a more suitable cost model. Your cost model can use existing statistics such as local and child selectivities and cardinalities. For example, for the hash-join algorithm, you may want to incorporate properties such as the hash table build phase amongst other algorithm phases. (This is an additive component of the cost rather than a multiplicative component.) You are free to collect other statistics during sampling or query execution and use them as part of your cost model. You should not need to consider more than the statistics presented during class (e.g., number of distinct values, min/max values, etc.).

4. Bushy plans and optimizer scalability (25 points. CS416 only)

In this exercise, we ask you to extend the join ordering optimization you implemented in Exercise 2 in two ways:

1. In a `BushyOptimizer` class that inherits from `Optimizer`, implement a variant of the dynamic programming optimizer that considers **bushy** plans as well as left-deep plans.
2. In a `GreedyOptimizer` class that inherits from `Optimizer`, implement a **greedy** optimizer variant that greedily constructs plans using the cheapest join available over the subplans as described in our lecture slides.

With these two variants, you should experimentally evalute the optimizer's scalablity reporting the number of plans considered (kept as a counter in your algorithm implementation), and the optimizer's running time. You should use apply these to join plans that are of size 2, 4, 6, 8, 10, 12. For this experiment, you can use a synthetic database schema, consisting of relations with triples of integers (e.g., R(a: int, b: int, c: int), S(d: int, e: int, f: int) and so forth). Note we are asking for an evaluation of the optimization overhead alone. You do not need to run the chosen query plans.

5. Experiments (25 points)

The goal of this exercise is to validate whether your optimizer is indeed selecting more efficient plans. We ask you to check this by running both your optimizer and your chosen plan on the queries provided below over the TPCH dataset provided. This includes a mix of select-project-join queries as well as select-project-join-aggregate queries. As with Assignment 2, we will supply the binary files used by our storage engine for the TPC-H dataset. This way you will not need to ingest the dataset for every run.

Dataset Available on the CS grad/ugrad network at: `~cs416/datasets/tpch-hw3/`.

Plots/figures For each query, graph the performance (running time) of an unoptimized query, and the optimized query as produced by your optimizer. You can generate a single bar chart with a pair of bars for each query.

1. **TPC-H Query 6: a 4-chain filter and aggregate query**

```
select
        sum(l_extendedprice * l_discount) as revenue
from
        lineitem
where
        l_shipdate >= 19940101
        and l_shipdate < 19950101
        and l_discount between 0.06 - 0.01 and 0.06 + 0.01
        and l_quantity < 24
```

2. **TPC-H Query 14: a 2-way join and aggregate query**

```
select
        sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
        lineitem,
        part
where
        l_partkey = p_partkey
        and l_shipdate >= 19950901
        and l_shipdate < 19951001
```

3. **TPC-H Query 3: a 3-way join and aggregate query**

```
select
        l_orderkey,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        o_orderdate,
        o_shippriority
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = 'BUILDING'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < 19950315
        and l_shipdate > 19950315
group by
        l_orderkey,
        o_orderdate,
        o_shippriority
```

4. **TPC-H Query 10: a 4-way join and group-by aggregate query**

```
select
        c_custkey,
        c_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        c_acctbal,
        n_name,
        c_address,
        c_phone,
        c_comment
from
        customer,
        orders,
        lineitem,
        nation
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate >= 19931001
        and o_orderdate < 19940101
        and l_returnflag = 'R'
        and c_nationkey = n_nationkey
group by
        c_custkey,
        c_name,
        c_acctbal,
        c_phone,
        n_name,
        c_address,
        c_comment
```

5. **TPC-H Query 5: a 6-way join and aggregate query**

```
select
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        customer,
        orders,
        lineitem,
        supplier,
        nation,
        region
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'ASIA'
        and o_orderdate >= 19940101
        and o_orderdate < 19950101
group by
        n_name
```

6. Histograms (40 points. Extra credit)

For the extra-credit exercise in this assignment, we ask you to implement an alternative selectivity estimation algorithm using 1-D histograms. The sampling-based estimator requires us to evalute each candidate query plan that we consider during optimization on a small fraction of the database. While this technique is elegant due to its simplicity, it does not provide that ability to ask "what-if" questions on either query plans or database designs by simulating the behavior of query execution. Histograms are the basic technique used in DBMS to provide this functionality.

In this exercise, you should implement a general-purpose histogram class that uses the **equi-width** representation for values. You should then design a `StatisticsManager` class that keeps a dictionary of histograms for each attribute in the database. For cardinality estimation on a given query, you should determine the attributes used by the query and retrieve their histograms. You must then establish and evalute an **estimation plan**. This estimation plan will define how to combine cardinalities from expressions estimated on each 1-D histogram, where you can rely on uniformity and independence assumptions. Consider the following query:

```
select * from R where R.a > 20 and R.b > 40
```

We can use the following estimation plan to determine query statistics:

```
Multiply cardinalities based on independence
|-- Compute cardinality of R.a > 20 on histogram over R.a
\-- Compute cardinality of R.b > 40 on histogram over R.b
```

Note that this also provides cardinality estimates for each subplan in the query. We can then derive selectivities and costs for each subplan in the same fashion as the default implementations of the `Operator.selectivity` and `Operator.cost` methods. We will award a maximum of 25 points for cardinality estimation over single-variable predicates (i.e. *variable < constant*), and 40 points for cardinality estimation over binary-variable predicates (i.e., *variable < variable*, as used in joins).

Summary of grading

Exercise	**Points**
1: Push down optimization	30
2: Join order optimization	30
3: Cost model design	15
4: Bushy plans and optimizer scalability (416 only)	25
5: Experiments	25
6: Histograms (extra credit)	40

Handin

Please submit your solutions by email to `cs416@cs.jhu.edu`. The submission must be a zip/tar.gz archive with the same structure as the handout, including both the handout code and your own. Name your submission with the names of your group members, and include a README with your names and email addresses.

IMPORTANT do not include the supplied data files in your repository, these are large and should not be duplicated for handin.

Relevant lectures

Lecture 8-9: query optimization