# Project 2: Join Algorithms and Query Optimization

## Logistics

**Due date: Wednesday 11/16/16, 11:59:59 PM**

### Grading

This project is worth **15%** of your overall course grade. Your project grade will be out of 100 points:

- 90 points for passing all of the tests provided for you. All tests are in `src/test/java/edu/berkeley/cs186/database`, so you can run them as you write code and inspect the tests to debug. Our testing provides extensive unit testing and some integration (end-to-end) testing.
- 10 points for writing your own, valid tests (10 tests total, 1 point each). The tests **must** pass both your implementation and the staff solution to be considered valid tests.

### Extra Credit

You can earn up to 10 points in extra credit:

- 5 points for submitting **one week** before the due date (Wednesday 11/09/16, 11:59:59 PM).
- Up to 5 points for how effective your tests are. The way we will determine the effectiveness of your tests is based on how many edge cases you identify that the rest of the class did not identify. You **will not** be graded on how many other students' tests you pass.

## Background

This project builds on top of the code and functionality that you should be familiar with from Project 1. In this project, you will be implementing several of the join algorithms covered in class as well as a cost-based query optimizer. We have released the staff solutions for Project 1 to a private repository (link can be found on Piazza), and you are free to integrate those into your codebase. Though if you feel confident about your own solutions to Project 1, you may stick with those as well. Regardless, a correct working implementation of Project 1 is **necessary** in order to complete this project.

This project uses the same function-oriented query interface we implemented in Project 1, but with a few differences. The original implementation was inefficient because at every stage of the query processing pipeline, we would fully materialize every record in memory. This means the query operators execute in sequential batches, and furthermore queries will fail if data does not fit in memory. To address this, we have implemented an `Iterator` interface directly into each query operator, and you'll be adding some more iterators for the join operators in this project!

Additionally, the original query plan would always execute a naive plan without any optimizations built in. We've included a separate execution method that will leverage a query optimizer that you will be filling in to generate better plans according to the System R cost-based approach we've covered in class.

## Getting Started

As usual, you can get the assignment from the Github course repository (assuming you've set things up as we asked you to in homework 0) by running `git pull course master`. This will get you all of the starter code we've provided, this project spec, and all of the tests that we've written for you.

**We've also generated all of the API documentation of the code as webpages viewable here**

### Setup

All of the Java, Maven, and IntelliJ setup from Project 1 is still applicable for this project. Assuming that you were able to complete Project 1, you should not need to do any additional setup for this project. Refer to the Project 1 spec for setup details.

## Starter Code

### Package Overview

There's a bunch of code that we've provided for you, both from Project 1 and some new additions in this project. We strongly suggest that you spend some time looking through it before starting to write your own code. The Project 1 spec contains descriptions for each package of the codebase, but the relevant parts for this project are all in the `query` package. Briefly, it consists of a `QueryPlan` class that holds the query optimizer, and several classes that all inherit from `QueryOperator` with each class representing a different operator in the plan tree.

All of the starter code descriptions from Project 1 are still valid. We have modified some classes and added additional functionality, but for the most part the code is the same except where we have specifically noted differences here in this document.

### Query Generation

The `QueryPlan` interface allows you to generate SQL-like queries without having to parse actual SQL queries. We use the same interface from Project 1 to create queries, but whereas before you would call the naive `QueryPlan#execute` method to execute your query, now there exists a `QueryPlan#executeOptimal` which will generate an optimal plan, and then execute your query.

If you would like to run queries on the database, you can create a new `QueryPlan` by calling `Transaction#query` and passing the name of the base table for the query. You can then call the `QueryPlan#where`, `QueryPlan#join`, etc. methods in order to generate as simple or as complex a query as you would like. Finally, call `QueryPlan#executeOptimal` to run the query optimizer and execute the query and get a response of the form `Iterator<Record>`. You can also use the `Transaction#queryAs` methods to alias tables.

As a quick example, a simple query might look something like this:

```
// create a new transaction
Database.Transaction transaction = this.database.beginTransaction();

// alias both the Students and Enrollments tables
transaction.queryAs("Students", "S");
transaction.queryAs("Enrollments", "E");

// add a join and a where to the QueryPlan
QueryPlan query = transaction.query("S");
query.join("E", "S.sid", "E.sid");
query.where("E.cid", PredicateOperator.EQUALS, "CS 186");

// execute the query and get the output
Iterator<Record> queryOutput = query.executeOptimal();
```

You can also examine the query plan tree generated by the query optimizer by printing the final operator of the query plan:

```
// assuming query.executeOptimal() has already been called as above
QueryOperator finalOperator = query.getFinalOperator();
System.out.println(finalOperator.toString());
```

which results in a print-out of the query plan:

```
type: BNLJ
leftColumn: S.sid
rightColumn: E.sid
```

```
(left)
type: WHERE
column: E.cid
predicate: EQUALS
value: CS 186
    type: SEQSCAN
    table: E

(right)
type: SEQSCAN
table: S
```

You can find more examples in the `OptimalQueryPlanTest` and `OptimalQueryPlanJoinsTest` files.

## Your Assignment

Alright, now we can write some code! **NOTE**: Throughout this project, you're more than welcome to add any and all helper methods you'd like to write. However, it is very important that you **do not change any of the interfaces that we've given you**. It's also a good idea to always check the course repo for updates on the project.

### Part 1: Join Algorithms

The first part of the project involves completing the implementations of `PNLJOperator`, `BNLJOperator`, and `GraceHashOperator`, which correspond to page nested loop join, block nested loop join, and grace hash join, respectively. Simple nested loop join has already been implemented for you in `SNLJOperator`. We will not be implementing sort-merge join in this project. It is recommended that you look through the code for `SNLJOperator`, and use this as an example of how to implement a join algorithm before starting the others.

As a brief overview of how the join algorithms in this project work, each join operator first reads in the records from the left input operator and the right input operator and stores those records in a left and right temporary table. The `JoinOperator` abstract class, which all the join algorithms extend, provides several methods for accessing record or page iterators from the left and right temporary tables along with other useful information. The `SNLJIterator` inside `SNLJOperator` reads records from the left and right operators in a nested loop fashion and joins the appropriate records.

#### 1.1 Page nested loop join

For the first part of this project you'll need to implement the `PNLJIterator` inside `PNLJOperator`. Similar to the `SNLJIterator`, this iterator should join records from the left and right source operators using a nested loop method. As discussed in class, the difference in this join algorithm (compared to simple nested loop join) is that you iterate over the right table once for every page in the left table instead of every record in the left table. Don't forget that the first page of a page iterator may be a header page!

We **strongly** recommend you write helper methods that retrieve the next left record and the next right record when you need to get them. Remember that a record is valid and should be retrieved only if the corresponding bit in the page header is set to 1. The `PNLJIterator` class has a lot of member variables already given, which serve as a guide for what you might need. You may also add any other member variables as you see fit. The order in which records are yielded is part of the tests, so make sure you are following the algorithms from lecture correctly.

It would also be helpful to familiarize yourself with the implementation of `SNLJIterator` and `TableIterator` before starting to write any code for this section.

#### 1.2 Block nested loop join

Once you have implemented page nested loop join, you should be able to extend this logic to implement block nested loop join. You'll need to implement the `BNLJIterator` inside `BNLJOperator` specifically. The algorithm should iterate over the right table once for every block of `B-2` pages in the left table instead of every page in the left table. `B` represents the total number of memory pages available to the query, and is accessible from the `numBuffers` field of `BNLJOperator`.

#### 1.3 Grace hash join

For the last chunk of this section you'll need to implement grace hash join in the `GraceHashIterator` inside `GraceHashOperator`. For grace hash join, you can make the simplifying assumption that we don't need to do recursive partitioning (i.e. you will only partition the inputs once and not multiple times). On the first phase of the algorithm, each record is hashed into its corresponding partition by being added to a temporary table that represents that partition. For an example of this, you can look inside the `GroupByOperator` class. However unlike the `GroupByOperator`, you should use a `DataType`'s `hashCode()` method and the modulo operator to hash a record to a particular partition. On the second phase, you can build an in-memory hash table using Java's built-in `HashMap` class. You should build an in-memory hash table using the records from the left input partitions, and probe the hash table using the records from the right input partitions.

#### 1.4 Testing

Once you've implemented all of these methods, you should be passing all of the tests in `JoinOperatorTest`. We strongly recommend you start writing tests once you've wrapped your head around the code to try to catch some of the edge cases that you might have missed. It's generally a good idea to write your own tests as you go along due to the fact that the given tests don't cover all edge cases.

## Part 2: Query Optimization

The second part of the project is focused on the query optimizer. We have provided some structure to help you get started. The `QueryPlan#executeOptimal` method runs the query optimizer and has been implemented already. This method calls other methods which you'll need to implement in order to make sure the optimizer works correctly.

As an overview of what this method does, it first searches for the lowest cost ways to access each single table in the query. Then using the dynamic programming search algorithm covered in lecture, it will try to compute a join between a set of tables and a new table if there exists a join condition between them. The lowest cost join of all the tables is found, and then a group by and select operator is applied on top if those are specified in the query. The method returns an iterator over the final operator created. The search should only consider left-deep join trees and avoid Cartesian products. Note that we are not expecting you to consider "interesting orders" for the purposes of this project.

For an example of the naive query plan generation, look at the code inside `QueryPlan#execute()`. Note that the query optimizer code will look quite different from the naive code, but the naive code still serves as a good example of how to compose a query plan tree.

#### 2.1 Cost estimation

The first part of building the query optimizer is ensuring that each query operator has the appropriate IO cost estimates. In order to estimate IO costs for each query operator, you will need the table statistics for any input operators. This information is accessible from the `QueryOperator#getStats` method. The `TableStats` object returned represents estimated statistics of the operator's output, including information such as number of tuples and number of pages in the output among others. These statistics are generated whenever a `QueryOperator` is constructed.

All of the logic for estimating statistics has been fully implemented except for the calculation of the reduction factor of a `WHERE` predicate. You must implement the `IntHistogram#computeReductionFactor` method which will return a reduction factor based on the type of predicate given. The reduction factor calculations should be the same as those that were taught in class.

After implementing this method, you should be passing all of the tests in `IntHistogramTest`.

Each type of `QueryOperator` has a different `estimateIOCost` method which handles IO cost estimation. You will be implementing this method in a few of the operators. This method should estimate the IO cost of executing a query plan rooted at that query operator. It is executed whenever a `QueryOperator` is constructed, and afterwards the cost of an operator can be accessed from the `QueryOperator#getIOCost` method.

Several operators already have their `estimateIOCost` methods implemented. In this project, you are only responsible for implementing this method in `IndexScanOperator`, `SNLJOperator`, `PNLJOperator`, `BNLJOperator`, and `GraceHashOperator`. For the index scan cost, assume an unclustered index is used. The methods in `Transaction` and `TableStats#getReductionFactor` will be useful for implementing the index scan cost. For the grace hash join cost, assume there is only one phase of partitioning.

After implementing the methods in this section, you should be passing all of the tests in `QueryPlanCostsTest`. And you should now have everything you need to start building the search algorithm.

#### 2.2 Single table access selection (Pass 1)

The first part of the search algorithm involves finding the lowest cost plans for accessing each single table in the query. You will be implementing this functionality in `QueryPlan#minCostSingleAccess`. This method takes in a single table name and should first calculate the estimated IO cost of performing a sequential scan. Then if there are any eligible indices that can be used to scan the table, it should calculate the estimated IO cost of performing such an index scan. The `QueryPlan#getEligibleIndexColumns` method can be used to determine whether there are any existing indices that can be used for this query. Out of all of these operators, keep track of the lowest cost operator.

Then as part of a heuristic-based optimization we covered in class, you should push down any selections that correspond to the table. You should be implementing the push down select functionality in `QueryPlan#pushDownWheres` which will be called by the `QueryPlan#minCostSingleAccess` method.

The end result of this method should be a query operator that starts with either a `SequentialScanOperator` or `IndexScanOperator` followed by zero or more `WhereOperator`'s.

After implementing all the methods up to this point, you should be passing all of the tests in `OptimalQueryPlanTest`. These tests do not involve any joins.

#### 2.3 Join selection (Pass i > 1)

The next part of the search algorithm involves finding the lowest cost join between each set of tables in the previous pass and a separate single table. You will be implementing this functionality in `QueryPlan#minCostJoins`. Remember to only consider left-deep plans and to avoid creating any Cartesian products. Use the list of explicit join conditions added through the `QueryPlan#join` method to identify potential joins. Once you've identified a potential join between a left set of tables and a right table, you should be considering each type of join implementation in `QueryPlan#minCostJoinType` which will be called by the `QueryPlan#minCostJoins` method.

The end result of this method should be a mapping from a set of tables to a join query operator that corresponds to the lowest cost join estimated.

#### 2.4 Testing

If you've completed all the sections up to this point, you should now be passing **all** of the tests that we've given you. Again, we strongly encourage you to write tests as you go to try to catch any relevant bugs in your implementation.

## Part 3: Testing

We can't emphasize enough how important it is to test your code! Like we said earlier, writing valid tests that **test actual code** (i.e., don't write `assertEquals(true, true);`, or we'll be mad at you) is worth 10% of your project grade.

CS 186 is a design course, and validating the reasonability and the functionality of the code you've designed and implemented is very important. We suggest you try to find the trickiest edge cases you can and expose them with your tests. Testing that your code does exactly what you expect in the simplest case is good for sanity's sake, but it's often not going to be where the bugs are.

To encourage you to try to find interesting edge cases, we're going to give you some extra credit. If you find edge cases that most other students didn't find, you can get up to 5 points of extra credit. Keep in mind that if some other people find the same edge cases as you, that doesn't mean you won't get extra credit -- so don't treat this as a competition!

#### Writing Tests

In the `src/test` directory you'll notice we've included several tests for you already. You should take a look at these to get a sense of how to write tests. You should write your tests in one of the existing files according to the functionality you're trying to test.

All test methods you write should have both the `@Test` and `@Category(StudentTestP2.class)` annotations. Note that this is different from the `@Category(StudentTest.class)` annotation you used in Project 1! This is important in making sure that your Project 2 tests are not mixed up with your Project 1 tests. We have included an example test in the `OptimalQueryPlanTest` class:

```
@Test
@Category(StudentTestP2.class)
public void testSample() {
  assertEquals(true, true); // Do not actually write a test like this!
}
```

Then whenever you run `mvn test`, your test will be run as well. To run only the tests that you wrote for this project, you may run `mvn test -Dtest=StudentTestSuiteP2`. You now also have the ability to run only the tests in a specific package. For example, you may run `mvn test -Dtest="edu.berkeley.cs186.database.query.*"` to run all of the tests in the `query` package. This may be helpful for debugging to save you time from running the whole test suite.

## Part 4: Feedback

We've been working really hard to give you guys the best experience possible with these new projects. We'd love to improve on them and make sure we're giving reasonable assignments that are helping you learn. In that vein, please fill out this Google Form to help us understand how we're doing!