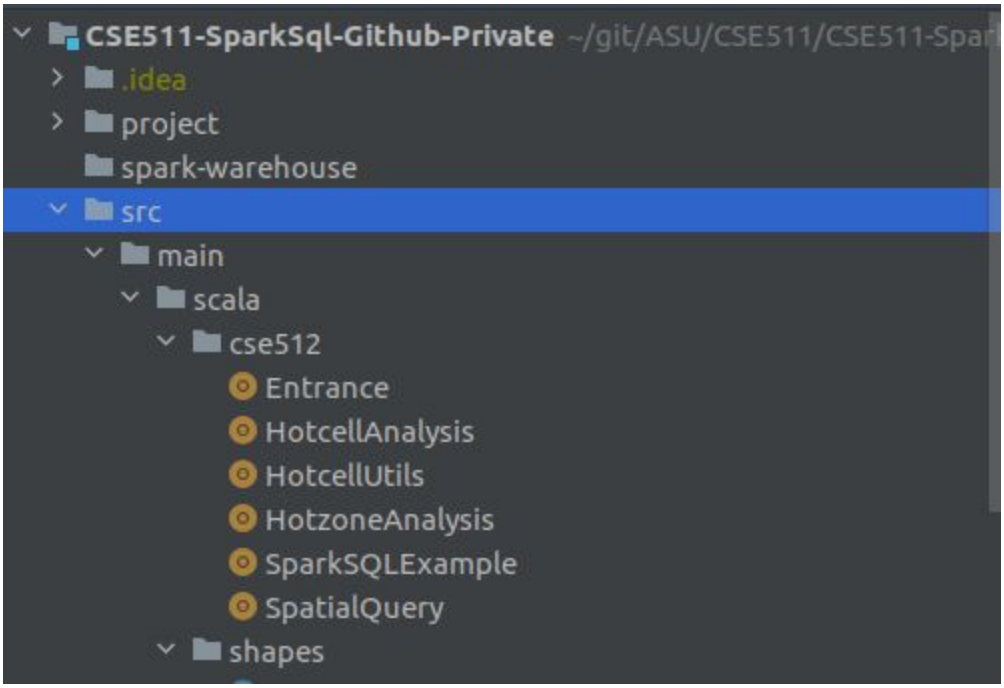
Requirements

- Hot Zone Analysis: This task will need to perform a range join operation on a rectangle datasets and a point dataset. For each rectangle, the number of points located within the rectangle will be obtained. The hotter rectangle means that it includes more points. So this task is to calculate the hotness of all the rectangles.
- Hot Cell Analysis:
    - Input: A collection of New York City Yellow Cab taxi trip records spanning January 2009 to June 2015. The source data may be clipped to an envelope encompassing the five New York City boroughs in order to remove some of the noisy error data (e.g., latitude 40.5N – 40.9N, longitude 73.7W – 74.25W).
    - Output: A list of the fifty most significant hot spot cells in time and space as identified using the Getis-Ord G_i^* statistic.
    - More details

Design

Reuse Existing Code

- Since this is Scala (or any JVM language), there's no harm for use to reuse the existing Git repository as this assignment will use a different entry point (Entrance.scala instead of SparkSQLExample.scala).
- To make this easier, we can copy the new sample source files from the zip file and place them into the src/main/scala/cse512 folder. The result should look like this:



Hot Zone Analysis

- Since we've already solved range join (ST_Contains) in the previous phase of the assignment, we can reuse the Point and Rectangle classes that we created. We can start by defining a UDF for ST_Contains and create a simple SQL query that returns the rectangle coordinates and the number of points that lie inside
- Things to be careful of:
    - Order the results by the rectangle value
    - Call coalesce(1) on the resulting DataSet before returning it
        - This ensures that we get down to one partition before submitting which forces the output CSV as a single file
- Testing:
    - Unit Tests: We will not need to write any additional unit tests since this is functionality that we have already tested
    - Functional Tests: We should include a functional test that compares the sample input and output from the provided zip file

```
test("run HotZoneAnalysis on test data") {
  val actualAnswer = HotzoneAnalysis.runHotZoneAnalysis(
    HotzoneAnalysisTest.spark,
    "src/resources/point_hotzone.csv",
    "src/resources/zone-hotzone.csv")

  val schema = new StructType()
    .add("rectangle", StringType, false)
    .add("count", IntegerType, false)

  val expectedAnswer = HotzoneAnalysisTest.spark.read
    .format("com.databricks.spark.csv")
    .option("delimiter", ",")
    .option("header", "false")
    .schema(schema)
    .load("testcase/hotzone/hotzone-example-answer.csv")
    .orderBy("rectangle")

  actualAnswer.collect() should contain theSameElementsAs expectedAnswer.collect()
  expectedAnswer.collect() should contain theSameElementsAs actualAnswer.collect()
}
```

Refactoring for Hot Cell Analysis

- For Hot Cell Analysis, we need to consider points and rectangles in three-dimensional space. However, we still need the classes that map points and rectangles in two-dimensional space. To avoid confusion, we can rename Point and Rectangle classes to Point2D and Rectangle2D. IntelliJ does this really well by allowing you to rename a class which percolates all changes down to all uses.

- For sanity, we should look to make this its own pull request to avoid accidentally breaking out code. This is one of the biggest advantages of unit tests as we can check that things work as expected pretty quickly (seconds vs waiting on the auto-grader).

Implementing Point3D and Rectangle3D

- Using the two-dimensional versions as a template, we need to create three-dimensional versions of Point and Rectangle classes (Point3D and Rectangle3D). These classes should implement the following:
  - Point3D
    - `within(otherPoint: Point3D, range: Double): Boolean`
      - Determines whether the current point and another point are within the given range using Euclidean distance
  - Rectangle3D
    - `contains(point: Point3D): Boolean`
      - Determines whether a Point3D is contained within the Rectangle3D
- Testing:
  - Unit Tests: Similar to those created in Point2D and Rectangle2D will suffice

Implementing Hot Cell Analysis

The template gets us to the point that we have pickupInfo, which is a table that is a projection of all x, y, z coordinates of all pickups. From this we need to do the following:

- Filter rides to only those contained within the search space
  - This is just creating another St_Contains UDF that checks to see if a Point3D is contained within a Rectangle3D
  - NOTE: You can avoid constructing Rectangle3D multiple times by making it extend Serializable. This allows us to use objects created outside the UDF as part of the UDF function.

```
*/
@SerialVersionUID(100L)
class Rectangle3D private(val vertex1: Point3D, val vertex2: Point3D) extends Serializable {
```

- Count the number of appearances of the each cell and store this as a table (cellToScore)
  - Grouping by x, y, z
- Write a query that calculates the total number of rides and the sum of squares of all rides in cells using cellToScore
  - Total number of rides is used to calculate $\bar{X}$
  - Sum of squares of all rides is used to calculate S (standard deviation)
- Write another query to count the number of rides in the cell and its neighbors
  - This is a cross join operation that uses the cellToScore twice
  - Neighbor cells are cells that are within 1 unit in each the x, y, and z direction

    - To ease calculation, we can use the `Point3D#within` function to determine if the cell is a neighbour (ex. If the distance is <= sqrt(3), the cell / point is a neighbor)
- Write another query to get the total number of neighbors the cell (this can be added to the previous query to avoid recalculation)
    - Unfortunately, not all cells contain data, thus we may get some cells with the incorrect number of neighbors if we chose to just do a count on the cross join. We also need to consider points that lie on any boundary of the Rectangle3D, such that if it is on a boundary, it will have fewer neighbors.
    - To combat this problem, we can do the following:
        - Create a method in Rectangle3D that calculates the number of boundaries a Point3D is touching:

```
def getBoundariesTouching(point: Point3D): Int
```

            - In the function, set a counter to 0.
            - Checks if the Point3D's x coordinate is equal to one of the Rectangle3D's x vertices. If so, increment a counter by 1.
            - Repeat for y and z
            - Return the counter value

        We then create a UDF (`ST_GetCellNeighborCount`) that returns the number of neighbor cells:

```
neighbor cell count = 2 ^ (search space boundaries) * 3 ^
(3 - search space boundaries)
```

- After generating the $\bar{X}$, S, cell scores, and cell neighbor counts, we have all the components we need to generate the G_i^* score as a UDF and store the results for each cell in another table
    - In the equation given, weight (w) is 1 for cells that are neighbours, and 0 if otherwise. So we can shortcut the equation as we've done a lot of the work already:
        - Sum of W = count of neighbor cells
        - (Sum of W) ^ 2 = above to pow 2
        - Sum (X * W) = cell score
        - Number of cells = total cells in the search space
    - We should look to extract this function into HotcellUtils to allow for easy verification of the algorithm using a unit test. I wrote the following which is a very simple calculation where we restrict the cell space to a 5 x 5 x 3 space and only have two points that have a score of 1, rest at zero

```
test("correctly calculates gi score for 5x5x3 using (2,2,1) with score 1, and a non-neighbor has 1") {
  val numCells = (5 * 5 * 3).toDouble
  val sumOfValues = 2
  val sumOfSquares = 2
  val partialGScore = HotcellUtils.partialGscore(sumOfValues, sumOfSquares, numCells)

  val cellByWeightSum = 1
  val adjacentNeighbors = 27

  assert(math.round(partialGScore(cellByWeightSum, adjacentNeighbors) * 10000) / 10000.0 == 0.4153)
}
```

- Once we have calculated the G_i^* score for each cell, we sort, coalesce, and project (we only need the x, y, and z coordinates)
    - NOTE: Order is important to make the auto-grader happy. The correct order is Gscore DESC, x DESC, y ASC, z DESC. Honestly, I can't make this up. Your submission score can vary from 9 to 15 points based on the order you use.
- Functional Testing:
    - I created a simple functional test that uses the entire search space but is enough to capture the following elements:
        - Cell with score 1 touching one boundary
        - 4 cells with score 1 touching no boundaries
            - This is to verify ordering so have 3 with the same x, 2 with the same y, and different z's
    - NOTE: The test samples given in the zip DO NOT WORK. They are totally wrong (z-score is incorrect, values are incorrect, ordering is not correct). What passes the autograder does not conform, so only use for reference into understanding what the output structure looks like.