Project 4: Joins

Tests Due: Friday 12/05/22 Before 11:59 PM

Advising (e.g., Piazza, Office Hours) may not be available over the weekend after 4:00PM on the Friday preceding the deadline. Plan accordingly

GitHub Classroom Invitation: https://classroom.github.com/a/QnW10Nzd

Total Points = 30

Objectives

In this assignment, you will:

- Use maps to perform join two datasets together
- Use maps to compute statistics over a dataset
- Learn about the limitations of simple data anonymization techniques
- Learn about privacy issues resulting from data release

Useful Resources

Review the lecture notes and provided example code for some insight into the Scala syntax. You will also want to read the Scala references provided below:

- The Scala API
    - Scala Collections
    - scala.io.Source
    - scala.collection.mutable.Map
    - scala.collection.mutable.Set
    - scala.collection.mutable.HashMap
- Scala Tour
- Scala Resources
- ScalaTest: Writing your first test

Late Policy

The policy for late submissions on assignments is as follows. Your project grade is the grade assigned to the latest (most recent) submission you make to autolab (or 0 if no submissions are made).

If your submission is made...

- ... prior to the deadline, your submission is assigned a grade of 100% of the points it earns.
- ... up to 24 hours after the deadline, your submission is assigned a grade of 75% of the points it earns.
- ... more than 24 hours after the deadline, but within 48 hours of the deadline, your submission is assigned a grade of 50% of the points it earns.
- ... more than 48 hours after the deadline, it will not be accepted.

Starting with this assignment there are no bonus points will be awarded for early submissions. (Early submissions are still encouraged, as Autolab gets congested close to deadlines)

You will have the ability to use three grace days throughout the semester, and at most two per assignment (since submissions are not accepted after two days). Using a grace day will negate the 25% penalty per day, but will not allow you to submit more than two days late. Please plan accordingly. You will not be able to recover a grace day if you decide to work late and your score was not sufficiently higher. **Grace days are automatically applied** to the first instances of late submissions, **and are non-refundable**. For example, if an assignment is due on a Friday and you make a submission on Saturday, you will automatically use a grace day, regardless of whether you perform better or not. **Be sure to test your code before submitting**, especially with late submissions **in order to avoid wasting grace days**.

Keep track of the time if you are working up until the deadline. Submissions become late after the set deadline. Keep in mind that **submissions will close 48 hours after hte original deadline** and you will not be able to submit your code after that time.

AI Policy Overview

As a gentle reminder, please re-read the academic integrity policy of the course. I will continue to remind you throughout the semester and hope to avoid any incidents.

What constitutes a violation of academic integrity?

These bullets should be obvious things not to do (but commonly occur): * Turning in your friend's code/write-up (obvious). * Turning in solutions you found on Google with all the variable names changed (should be obvious). This is a copyright violation, in addition to an AI violation. * Turning in solutions you found on Google with all the variable names changed and 2 lines added (should be obvious). This is also a copyright violation. * Use of Github Autopilot (should be obvious). This is still in murky legal water, and may be a copyright violation, in addition to being an AI violation. * Paying someone to do your work. You may as well not submit the work, since you will fail the exams and the course. * Posting to forums asking someone to solve assignment problems *(even if you do not receive the solution)* * Accessing solutions to assignment problems.

Note: Aggregating every { stack overflow answer, result from google, other source } because you "understand it" will likely result in full credit on assignments (if you are not caught), and then failure on every exam. Exams don't test if you know how to use Google, but rather test your understanding (i.e., do you understand the problem and material well enough to arrive at a solution on your own). Also, other students are likely doing the same thing, and then you will be wondering why 10 people that you don't know have your exact solution.

What collaboration is allowed?

There is a grey area when it comes to discussing the problems with your peers, and I do encourage you to work with one another to discuss course *concepts* related to an assignment. That is the best way to learn and to overcome obstacles. At the same time, you need to be sure you do not overstep and not plagiarize. Discussions pointing to relevant course materials are OK. For example, the following is acceptable advice:

It would be helpful to review the usage of the stack in the recitation slides from week XX.

When working with your peers, we ask that you include attribution; In the header comment of the Main function of your submission, please list all peers who you have discussed the project with.

Explaining every step in detail and/or giving pseudocode that solves the problem is **not ok**. For example, the following is **not acceptable** advice:

I copied the algorithm from the week XX notes into my code at the start of the function, created a function that went through the given data and put it into a list, called that function, and then sorted the results.

The first example is OK. The second example, however, is a summary of your code and is not acceptable. Remember that you should never show any of your code to other students prior to any deadlines. Regardless of where you are working, you must always follow this rule: **Never come away from discussions with your peers with any written work, either typed or photographed, and especially do not share or allow viewing of your written code**.

If you have concerns that you may have overstepped or worked too closely with someone, please address this with me prior to deadlines for the assignment. **Even if you have submitted an assignment that may have violated the course academic integrity policy, if you approach me *prior to detection* you will not face academic integrity proceedings**. We will address options at that point.

What resources are allowed?

With all of this said, please feel free to use any { files | examples | tutorials } that course staff provides, directly in your code. Feel free to directly use any materials from lecture or recitations. You will never be penalized for doing so, but must always provide attribution/citation for where you retrieved code from. Just remember, if you are citing an algorithm that is not provided by us, then you are probably overstepping.

More explicitly, you may use any of the following resources (with proper citation/annotation in your code: * Any example files posted on the course webpage or Piazza (from lecture or recitation) * Any code that the instructor provides * Any code that the TAs provide * Any code from the Tour of Scala * Any code from Scala Collections * Any code from Scala API * Additional references may be provided as the semester progresses, but only those provided publicly by course staff are allowed for use. These will be listed on Piazza under Resources

Omitting citation/attribution will result in an AI violation (and lawsuits later in life at your job). This is true, **even if you are using provided resources**.

Again, **if you think you are going to violate/have violated this policy, please come talk to a member of the course staff ASAP so we can figure out how to get you on track to succeed in the course**. If you have a question about the validity of a resource, please ask a TA or your instructor prior to using it. If you have already used it, please discuss with the

instructor to determine a workaround and, at the very least, avoid an academic integrity infraction. For example, you might send an email such as the following to the course instructor:

Clarus T Example
UBIT: ctexamp
Person #: 123456789

Dear Dr. Kennedy/Mikida,

I believe that I may have submitted work that is *{ not fully my own | not properly attributed }*. I wish to retract my submission to preserve academic integrity in the course.

Signed,
Clarus T Example

This policy on assignments is here so that you learn the material and how to think for yourself. There is no cognitive benefit achieved by submitting solutions someone else has written (which likely already exist in some form).

Collaboration Policy

The policy for collaboration on assignments is as follows:

- All work for this course must be original individual work.
- You must follow the limits on collaboration as defined in the AI policy (i.e., no shared code/etc...)
- You must identify any collaborators (first and last name) on every assignment. This can be in a comment at the top of your code submissions or on the first page at the top of your written work, beside your name.

All references must be cited using a comment containing a direct link to the resource, as well as a brief description of what was used. For example, if you reference the textbook, a page number and description is sufficient. If you copy example code from the Scala Language API, then include the link to the class page within the API as well as where the example code resides.

Relational Database Joins

Relational (SQL) databases like MySQL, Postgresql, SQLite, and others are used everywhere for data storage and processing. One of the main computational tasks relational databases handle on a regular basis is called a **join**. Although the computation you will be asked to implement in this assignment is not exactly a join, you may find the following both informative and helpful when implementing your assignment.

Relational Tables

A relational database stores data in **Tables** (you may have heard these called "Dataframes" or "Tensors" in other settings). A table is a collection of **Records**. All records have a common set of **Fields** (sometimes called Attributes). For example, the following is a table that we'll call **Customers**.

#	First Name	Last Name	Birthday	Zip Code
0	SIMON	DURAN	11/17/1978	14261
1	EMELIA	STEWART	9/23/1996	14201
2	NIA	GONZALEZ	7/12/1970	14210
3	VIOLET	HARMON	3/16/1986	14216
4	EDWIN	SUTTON	6/21/1986	14201
5	LILY	BAKER	8/31/1988	14213
6	MILO	ORTIZ	11/15/1973	14201
7	KARA	OLIVER	6/10/1968	14214
8	BRANDON	GARDNER	2/20/1957	14223
9	DEWEY	WILSON	3/14/1987	14212

There are 10 records (numbered 0-9) in this table. Each record has four fields: **First Name**, **Last Name**, **Birthday**, and **Zip Code**.

Here's another example that we'll call **Pizza By Zip**.

#	Zip Code	Closest Pizza
0	14201	La Nova Wing Incorporated
1	14216	Jet's Pizza Delivery
2	14223	Pie-O-Mine Greens
3	14213	Sports City Pizza Pub
4	14261	Imperial Pizza
5	14212	Pizza Express
6	14214	Just Pizza

Joins

A **relational join** (sometimes called an inner join, or just a join) links the records of two tables together on some field, called the **join key**. For each record in one table, we're going to find the corresponding record in the other table and produce a new "combined" record.

(To be pedantic, what we're talking about here is a specific kind of join called an equi-join. However, equi-joins are sufficiently common that it's common to refer to equi-joins as simply joins)

For example, let's join together the **Customer** and **Pizza By Zip** tables on their mutual **Zip Code** attribute (e.g., to find the reccomended pizza place for each customer). It is customary to represent the join operation with a bowtie (⋈), so this join would be written:

$$\textbf{Customer} \bowtie \textbf{Pizza by Zip}$$

For each record in **Customer**, we're going to find the record in **Pizza By Zip** with an identical **Zip Code**. The result should look like the following:

#	First Name	Last Name	Birthday	Zip Code	Zip Code	Closest Pizza
0	SIMON	DURAN	11/17/1978	14261	14261	Imperial Pizza
1	EMELIA	STEWART	9/23/1996	14201	14201	La Nova Wing Incorporated
2	VIOLET	HARMON	3/16/1986	14216	14216	Jet's Pizza Delivery
3	EDWIN	SUTTON	6/21/1986	14201	14201	La Nova Wing Incorporated
4	LILY	BAKER	8/31/1988	14213	14213	Sports City Pizza Pub
5	MILO	ORTIZ	11/15/1973	14201	14201	La Nova Wing Incorporated
6	KARA	OLIVER	6/10/1968	14214	14214	Just Pizza
7	BRANDON	GARDNER	2/20/1957	14223	14223	Pie-O-Mine Greens
8	DEWEY	WILSON	3/14/1987	14212	14212	Pizza Express

Note a few things about the above:

1. The output of the join operation is also a table.
2. The records in the output table have all of the fields of **both** input tables (in fact, the **Zip Code** attribute is repeated for this reason).
3. A single record in one table may join with multiple records in the other table (e.g., Record 0 of **Pizza By Zip** joins with Records 1, 4, and 6 of **Customers**).
4. If a record in one table does not match with any records in the other table, it is omitted from the output (e.g., Record 2 of **Customers** does not have a matching **Zip Code** in **Pizza By Zip**).

Nested Loop Join

The Join operation is usually defined by its simplest implementation, called the Nested-Loop Join. The algorithm, in lightly simplified form, appears as follows:

```
def nestedLoopJoin[R1, R2, K](
  tableA: Iterable[R1],
  tableB: Iterable[R2],
  getTableAKey: R1 => K,
  getTableBKey: R2 => K
): Seq[(R1, R2)] =
{
  val result = new ArrayBuffer[(R1, R2)]()
  for( a <- tableA ){
    for( b <- tableB ){
      if( getTableAKey(a) == getTableBKey(b) ){
        result.append( (a, b) )
      }
    }
  }
  return result.toSeq
}
```

The body of the function is a nested for loop. For every record in the **A** table, it iterates over every record in the **B** table to find the B record(s) that match. If it finds a matching record, it adds it to the result list.
After it's done with every **A** record, it returns all of the matches it's made.

Runtime

The runtime of the nested-loop join algorithm is $O(|\texttt{tableA}| \cdot |\texttt{tableB}|)$. From one perspective, this growth is only linear in each *individual* table's size. However, if we allow both tables to grow together (e.g., if we set $|\texttt{tableA}| = |\texttt{tableB}| = n$), then the runtime becomes quadratic.

Example

Let's assume that the records of our example tables are defined as:

```
case class Customer(
  firstName: String,
  lastName: String,
  birthday: String,
  zipCode: String,
)
case class PizzaByZip(
  zipCode: String,
  closestPizza: String
)
```

To join the two example tables:

```
val customer: Seq[Customer] = /* Load Customer Table */
val pizzaByZip: Seq[PizzaByZip] = /* Load Pizza By Zip Table */

val output = nestedLoopJoin(
  customer,
  pizzaByZip,
  { c => c.zipCode } /* function to get a Customer record's join key */
  { pz => pz.zipCode } /* function to get a Pizza By Zip record's join key */
)
```

Hash Join

The runtime of the nested-loop join algorithm is high. To get a better runtime we can observe that a large part of the cost is repeatedly running the inner loop: We can do a bit of preparation work before we start running the algorithm to make the inner loop much faster. The result is what people who use and build relational databases called the "hash join" algorithm.

(Again, some terminological pedantry: this is specifically the one-pass hash join. A closely related variant is called the two-pass, or "grace" hash join, and is used when there isn't enough space to hold all of `tableA` and `tableB` in memory, or when the join computation is big enough that it needs to be run across multiple computers --- e.g., "in the cloud")

Note: Although the runtime of accesses to a `HashTable` are worst-case linear, we will use *expected* runtimes in this assignment.

Consider what happens when we load `tableB` into a HashMap first. Specifically for each record `b` in `tableB`, insert `b` into the HashMap with a key of `getTableBKey(b)` (this is typically called the *build* phase of hash join).

```
val hashTable = mutable.HashMap[K, List[R2]]()
for(b <- tableB){
  val key = getTableBKey(b)
  if(hashTable.contains(key)){
    hashTable(key) = b +: hashTable(key)
  } else {
    hashTable(key) = List(b)
  }
}
```

Note that multiple records of `tableB` may have the same join key (e.g., records 1, 4, and 6 of the example **Customer** table). Since we can't rule this out in general, typically, the hash join will store a Sequence of records for each join key.

Note: `+:`, the *prepend* operation on an immutable `List` (i.e., a Singly-Linked List) only requires modifying the list head, and runs in $O(1)$ time.

As discussed in class, building a HashMap with n records at a pre-determined load factor α takes an expected $O(n)$ (i.e., $O(|\text{tableB}|)$) time, even accounting for any necessary resizes.

Next, when we loop over the records of `tableA`, we can recover all of the `tableB` records by probing the hash table instead of looping over `tableB`.

```
for(a <- tableA){
  val key = getTableAKey(a)
  if(hashTable.contains(key)){
    for(b <- hashTable(key)){
      result.append( (a, b) )
    }
  }
}
```

We're still looping over every element of `tableA`, but now instead of a full iteration of `tableB`, we do an $O(1)$ hash table lookup (technically two in the algorithm above: `contains(key)` and `hashTable(key)`), which has an *expected* runtime of O(1). Since there might be multiple matches for a given `a` record, we also need to iterate over all of these.

Although the worst-case runtime of this step (typically called the *probe* phase) can be $O(|\texttt{tableA}| \cdot |\texttt{tableB}|)$ (i.e., if every `b` record has the same key), it is typically much lower. As a result, it is customary to capture the runtime of the hash join by looking at the number of records it outputs. For example, if we know that every record in `tableB` joins with at most one record of `tableA` (as is the case for **Pizza by Zip**), then we can bound the number of records in the join output by $|tableB|$.

Observe that each iteration of the inner loop (over the records in `tableB`) appends one record to the result. Thus, the total runtime for this function is

$$O\left(\sum_{a \in \texttt{tableA}} \left(1 + \sum_{b \in \texttt{tableB} \,:\, a.key = b.key} 1\right)\right)$$

$$= O\left(\sum_{a \in \texttt{tableA}} 1 + \sum_{a \in \texttt{tableA}} \sum_{b \in \texttt{tableB} \,:\, a.key = b.key} 1\right)$$

$$= O(|\texttt{tableA}| + |\texttt{result}|)$$

Specifically, it's possible for $|result|$ to be as large as $|\texttt{tableA}| \cdot |\texttt{tableB}|$ (if there's exactly one join key value). but in practice, result is usually linear in the size of one (or both) tables.

Runtime

Combining the runtimes of the build and probe phases, we get an overall **expected** runtime for the hash join algorithm of:

$$O(|\texttt{tableB}| + |\texttt{tableA}| + |\texttt{result}|)$$

Regardless of how the tables scale (and under the common assumption that the size of the output scales linearly with the size of the input), this function always grows **linearly**.

Anonymization

Suppose you are working for Company X and your team is tasked with producing a health record data set to be released to the participants in an upcoming hackathon. Your colleague removed the columns for names and says that the data is ready to go but you aren't convinced. After reviewing the data set, you are concerned that if someone were to combine your "anonymous" data with other publicly available sources there could be trouble. Unfortunately your colleague doesn't share in your concerns and requires proof.

Is this so far fetched? In 2007, Netflix held a competition to improve their recommendation algorithms. This required releasing a large amount of anonymous data from their customers' movie ratings and viewing histories. By combining the Netflix data set with the Internet movie database (IMDb) site data, researchers found a way to link the identity of a Netflix user to a user's IMDb profile based on the select reviews that they published.

With more data being generated and released (and redacted) every day, this is not limited to an old and irrelevant data set. Consider the APIs that expose locations of bikeshare/scooter information (e.g., NABSA). Making these locations publicly available through an API allows for this data to be incorporated into ecosystems that extend beyond the provider's control. As a result you can get useful features like Google maps providing walking directions to the closest scooter/bike within a city, regardless of the service provider. But what is the cost of this? After all, these locations are public knowledge since we can physically see these bikes and scooters on the street. So the exact location of a scooter at a given time may not expose much. What about the pairs of start and end locations along with the date and time of usage? Now we can track usage habits at a particular location and possibly an address. Combining this data and auxiliary information we may be able to identify a set of candidates that are using the

scooters. Further analysis may expose things such as daily habits -- particularly when someone normally leaves their house and returns, exposing further sensitive information. Fortunately there are techniques to thwart this.

To address this we have two main problems to tackle: (i) maintain the statistical relevance of the sensitive data and (ii) protect the people represented by the data. Fortunately, there is a well-studied body of tools and techniques under the umbrella of differential privacy for exactly this purpose. By training a model to understand an originally sensitive, but anonymized, data set we can generate synthetic data that looks statistically similar but further protects the individuals contained within. To see more about this, you can read the blog post by Alexander Watson and follow along with the example notebook they provide to understand further.

Instructions

Answer the following questions by:

1. Accept The PA4 Assignment in GitHub Classroom.
2. Implement the `???` placeholders in `DataTools.scala`
3. Commit **and push** your answers to GitHub
4. Submit PA4 in Autolab (note: Submissions open by Nov 21). Repeat 2-4 as needed.
5. After the deadline, an additional round of tests will be conducted on your implementation for the final 5 points.

Make sure your submission is committed and pushed into your GitHub Classroom Git repository. Seriously, make sure it's committed. Yeah you... the person who clicked submit without checking.

Expect this project to take 10-12 hours of setting up your environment, reading through documentation, and planning, coding, and testing your solution.

Problem 1: Loading Data (5 points)

In order to work with the data, we need to import it into our program. Your task is: given a filename, load the rows of the health dataset file (respectively, the voter records dataset) specified by `filename` into sequence of `HealthRecord` objects (respectively, `VoterRecord` objects) so that you will be able to process them later.

Examples of both datasets of varying sizes are provided in the `src/test/resources` directory.

Hint: be sure to review the data to understand what the format of a line looks like. For this problem you should complete the definition of the `DataTools.loadHealthRecords` and `DataTools.loadVoterRecords` methods.

Both functions can make the following assumptions:

- The file referenced by `filename` exists.
- The first line of the CSV file is a header.
- Every subsequent line may be split into fields with `String`'s split method.

- Columns containing dates are in a format interpretable by `parseDate` (which is already defined in `DataTools`).
- Header fields for the data files are as follows:

Header fields for the health records are:

1. "Birthday"
2. "Zip Code"
3. "Wear Glasses?"
4. "Allergic to Dogs?"
5. "Brown Hair?"
6. "Blue Eyes?"

Header fields for the voter records are:

1. "First Name"
2. "Last Name"
3. "Birthday"
4. "Zip Code"

As in PA0, you are encouraged to use scala.io.Source to load the data.

Problem 2: De-Anonymizing Data (20 points)

Given the data that we loaded, it is time to illustrate the problem to your colleague. Using the lists of `VoterRecord` and `HealthRecord` objects, produce a Map from each voter's full name (given by the `fullName` method of the `VoterRecord` class) to the matched `HealthRecord`. Every record in the map should correspond to a pair of `VoterRecord` and `HealthRecord` that can be matched uniquely (re-identified).

For this problem you should complete the definition of the `DataTools.identifyPersons` method.

```
def identifyPersons(
  voterRecords: Seq[VoterRecord],
  healthRecords: Seq[HealthRecord]
): mutable.Map[String, HealthRecord] = ???
```

Runtime: This *expected* runtime of this function **must** be $O(|\texttt{voterRecords}| + |\texttt{healthRecords}|)$

Note: A unique match occurs when exactly one voter record can map to exactly one health record and the opposite is also true.

Hint: To gain an understanding of a process to solve this problem, you may want to work it out by hand on the small data sets provided.

Problem 3: Statistics (5 points)

Now that you have your colleague on board, you decide to hand off your data set to the data science team to produce a statistically similar, but secured, data set. In order to ensure that the datasets are still relevant, we want to be able to compute and compare stats about the distribution of certain sensitive values. You will write a function that takes as input a list of `HealthRecord` objects and an attribute of the record, and computes the distribution of the associated values for the given list of records. For this problem you should complete the definition of the `DataTools.computeHealthRecordDist` method:

```
def computeHealthRecordDist(
  records: Seq[HealthRecord],
  attribute: HealthRecordAttribute
): mutable.Map[String, Double] = ???
```

`attribute` can have exactly one of two values.

- Given the value of `attribute`, you should calculate stats for the list of `HealthRecord` objects as follows:
  - If `attribute == HealthRecordBirthday`, use `HealthRecord`'s `m_Birthday` field.
  - If `attribute == HealthRecordZipCode`, use `HealthRecord`'s `m_ZipCode` field.
- In the output, store a mapping from each unique value to the percentage of the records containing that value.
  - The values should be stored as a `String`, regardless of initial type.
  - The percentages should be a value in the range $(0, 1]$.

Runtime: The *expected* runtime of this function **must** be $O(|\texttt{healthRecords}|)$

As a follow-up question, which you don't need to answer: why do we not care about trying to anonymize the `VoterRecord` data set?

Allowed library/container usage

- You may use any container in the scala.collection or scala.collection.mutable packages.

Submission

For **Project 4** you may submit as many times as you want. Your final score will be the last (most recent) submission

Revision History

- Summer 2021 - Initial project draft developed as part of the Mozilla Responsible Computer Science Challenge by:
  - Joshua Caskie (jmcaskie@buffalo.edu)
  - Alexander Fernandez (adfernan@buffalo.edu)

    - Garegin Grigoryan (grigoryan@alfred.edu)
    - Andrew Hughes (ahughes6@buffalo.edu)
    - Macy McDonald (macymcdo@buffalo.edu)
- Fall 2021 - Adapted for CSE-250 by Oliver Kennedy (okennedy@buffalo.edu)
- Fall 2022 - Updated by Oliver Kennedy (okennedy@buffalo.edu), Eric Mikida (epmikida@buffalo.edu)