Common Patterns for Implementing Algorithms in Scala and Spark

CS-562
Design Patterns

- A "Design Pattern" is a common solution to a common problem that isn't concrete enough to be packaged up as a helper method, class or module.

- A design pattern is a high-level sketch of a solution, and you still have to write all the code yourself.

- If the "commonality" is concrete enough you can create a helper method, then it is no longer called a design pattern.

- The idea of "design patterns" will always have a bit of vagueness to it.
Scala Language Features

➢ Override Modifier
➢ Case Classes and Immutability
➢ Recursion and Tail Recursion
➢ Exception Handling
Override Modifier

- Always add override modifier for methods, both for overriding concrete methods and implementing abstract methods. The Scala compiler does not require override for implementing abstract methods.
- However, we should always add override to make the override obvious, and to avoid accidental non-overrides due to non-matching signatures.
trait Parent {
  def hello(data: Map[String, String]): Unit = {
    print(data)
  }
}

class Child extends Parent {
  import scala.collection.Map

  // The following method does NOT override Parent.hello,
  // because the two Maps have different types.
  // If we added "override" modifier, the compiler would've caught it.
  def hello(data: Map[String, String]): Unit = {
    print("This is supposed to override the parent method, but it is actually not!")
  }
}
Case Classes and Immutability

Case classes are regular classes but extended by the compiler to automatically support:

- Public getters for constructor parameters
- Copy constructor
- Pattern matching on constructor parameters
- Automatic toString/hash/equals implementation

Constructor parameters should NOT be mutable for case classes. Instead, use copy constructor. Having mutable case classes can be error prone, e.g. hash maps might place the object in the wrong bucket using the old hash code.
Case Classes and Immutability

// This is OK
case class Person(name: String, age: Int)

// This is NOT OK
case class Person(name: String, var age: Int)

// To change values, use the copy constructor to create a new instance
val p1 = Person("Peter", 15)
val p2 = p2.copy(age = 16)
Recursion and Tail Recursion

❖ Avoid using recursion, unless the problem can be naturally framed recursively (e.g. graph traversal, tree traversal).
❖ For methods that are meant to be tail recursive, apply @tailrec annotation to make sure the compiler can check it is tail recursive. (You will be surprised how often seemingly tail recursive code is actually not tail recursive due to the use of closures and functional transformations.)
❖ Most code is easier to reason about with a simple loop and explicit state machines. Expressing it with tail recursions (and accumulators) can make it more verbose and harder to understand.
Recursion and Tail Recursion

// Tail recursive version.
def max(data: Array[Int]): Int = {
  @tailrec
def max0(data: Array[Int], pos: Int, max: Int): Int = {
    if (pos == data.length) {
      max
    } else {
      max0(data, pos + 1, if (data(pos) > max) data(pos) else max)
    }
  }
  max0(data, 0, Int.MinValue)
}

// Explicit loop version

def max(data: Array[Int]): Int = {
  var max = Int.MinValue
  for (v <- data) {
    if (v > max) {
      max = v
    }
  }
  max
}
Exception Handling (Try vs try)

Do NOT catch Throwable or Exception. Use scala.util.control.NonFatal:

```scala
try {
  ...
} catch {
  case NonFatal(e) =>
    // handle exception; note that NonFatal does not match InterruptedException
  case e: InterruptedException =>
    // handle InterruptedException
```

Exception Handling (Try vs try)

Background information:

- Scala provides monadic error handling (through Try, Success, and Failure) that facilitates chaining of actions. However, we found from our experience that the use of it often leads to more levels of nesting that are harder to read.

- In addition, it is often unclear what the semantics are for expected errors vs exceptions because those are not encoded in Try.
Exception Handling (Try vs try)

As a contrived example:

```scala
class UserService {
  /** Look up a user's profile in the user database. */
  def get(userId: Int): Try[User]
}
```

is better written as:

```scala
class UserService {
  /**
   * Look up a user's profile in the user database.
   * @return None if the user is not found.
   * @throws DatabaseConnectionException when we have trouble connecting to the database/
   */
  @throws(DatabaseConnectionException)
  def get(userId: Int): Option[User]
}

→ The 2nd one makes it very obvious error cases the caller needs to handle.
Part 2. Statistics Computation in Spark
Challenges of numerical computation over Big Data

When applying any algorithm to big data watch for:

➢ Correctness
➢ Performance
➢ Trade-off between accuracy and performance
Practical Examples

1. Point Estimations (Variance)

2. Approximate estimation (Cardinality)

- We use these examples to demonstrate Spark internals, data flow, and challenges of implementing algorithms for Big Data.
Big Data Variance

The plain variance formula requires two passes over data:

\[
Var(X) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2
\]
Fast but inaccurate solution

\[ Var(X) = E[X^2] - E[X]^2 \]

\[ = \frac{\sum x^2}{N} - \left( \frac{\sum x}{N} \right)^2 \]

- Can be performed in a single pass, but subtracts two very close and large numbers
Accumulator Pattern in Scala

→ An object that incrementally tracks the variance

```scala
class RunningVar {
  var variance: Double = 0.0

  // Compute initial variance for numbers
  def this(numbers: Iterator[Double]) {
    numbers.foreach(this.add(_))
  }

  // Update variance for a single value
  def add(value: Double) {
    ...
  }
}
```
Parallelize for performance

- Distribute adding values in map phase
- Merge partial results in reduce phase

```java
Class RunningVar {
    ...
    // Merge another RunningVar object
    // and update variance
    def merge(other: RunningVar) = {
        ...
    }
}
```
Computing Variance in Spark

→ Use the RunningVar in Spark

```scala
val doubleRDD = doubleRDD.mapPartitions(v => Iterator(new RunningVar(v))).reduce((a, b) => a.merge(b))
```

- Or simply use the Spark API

```scala
doubleRDD.variance()
```
Approximate Estimations

Often an approximate estimate is good enough especially if it can be computed faster or cheaper.

1. Trade accuracy with memory
2. Trade accuracy with running time

We really like the cases where there is a bound on error that can be controlled.
Cardinality Problem

Example:
Count number of unique words in Shakespeare’s work.

- Using a HashSet requires ~10GB of memory
- This can be much worse in many real world applications involving large strings, such as counting web visitors.
Linear Probabilistic Counting

1. Allocate a bitmap of size m and initialize to zero.
   
   A. Hash each value to a position in the bitmap
   
   B. Set corresponding bit to 1

2. Count number of empty bit entries: $v$

   $$\text{count} \approx -m \ln \frac{v}{m}$$
**The Spark API**

Use the LogLinearCounter in Spark

```scala
rdd
  .mapPartitions(v => Iterator(new LPCounter(v)))
  .reduce((a, b) => a.merge(b)).getCardinality
```

- Or simply use the Spark API

```scala
myRDD.countApproxDistinct(0.01)
```
References


2. https://databricks.com/spark/about
