W raporcie omówie krótko kilka decyzji projektowych.

Notebook implementuje algorytmy mapReduce, terasort, prefixSum, ranking, perfectBalanceSort i slidingWindowAggregation dla jednej lub wielu pól.

Algorytm jest implementowany zgodnie z artykułem “Minimal MapReduce Algorithms” co oznacza że funkcja agregująca musi być:

Distributive Aggregate Function: An aggregate function is *distributive* if it can be computed in a distributed manner as follows. Suppose the data are partitioned into *n* sets. We apply the function to each partition, resulting in *n* aggregate values. If the result derived by applying the function to the *n* aggregate values is the same as that derived by applying the function to the entire data set (without partitioning), the function can be computed in a distributed manner.

Więc sama średnia z dystansu i ilości osób jest liczona jako sum() a potem dzielona przez długość okna ponieważ funkcja average() nie jest dystrybutywna bo:

```
scala> val l = (1 to 10).toList
val l: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> l.sum.toDouble/l.size
val res3: Double = 5.5

scala> (l.slice(0,3).sum/3 + l.slice(3,10).sum.toDouble/7) / 2
val res5: Double = 4.5
```

Również ostatni Reduce algorytmu sliding window nie jest liczony możliwie “sprawnie” ale zgodnie z algorytmem z artykułu. Dla kazdego wiersza obliczenie okna polega na agregacji 3 elementowego zbioru { agregacji elementów z pierwszej maszyny, agregacji lokalnych agregacji maszyn pośrednich, agregacji elementów na maszynie liczącej okno } i tak jest zaimplementowany. Nie jestem pewien ale być może nie każdą funkcję dystrybutywną agregującą można by napisać sprawniej niż tak.

Dla uproszczenia dzielę zawsze przez rozmiar 1000 nawet jeśli okno jest krótsze bo tych okien na ponad milion istniejących jest tylko 999 i jest to szczególnie związany z implementacją minimalnych algorytmów

Wybrane fragmenty i decyzje:

trait MapReduce posiada wszystkie 4 możliwe metody mapReduce w zależności czy funkcje map bierze pary lub listy par a reduce zwraca element typu finałowego RDD czy liste tych elementów. Trzeba było użyć DummyImplicit bo type erasure. MapReducy są w pełni generyczne.

Więcej sensu ma wmieszanie Settings w object MinimalALgorithms niż jego niejawnie dziedziczenie więc tutaj jest sygnalizowany jako selftype

Używany jest trywialny partitioner by precyzyjnie wysyłać dane na inną maszynę lub nie wysyłać ich jeśli maszyna jest właściwa ponieważ zgodnie z mapReduce wszystkie dane zamieniane są na (klucz, wartość) również te które w reduce mają zostać. Przy czym najpierw tworzę abstrakcyjną klasę genericPartitioner[K] i następnie tylko jeden singleton object go dziedziczący dla K=Int jako implicit. To sprawi że jeśli osoba używająca metod mapReduce nie poda własnego partitionera ten zostanie przekazany tylko gdy klucz będzie typu Int i będzie można bezpiecznie scastować go z typu Any który jest w metodzie getPartition(key: Any) do Int.

```
trait MapReduce extends java.io.Serializable {
    this: Settings =>
    import org.apache.spark.Partitioner

    // force user to use partitioner for appriopriate type of key with static typing
    abstract class genericPartitioner[K] extends Partitioner {
        override def getPartition(key: Any) = convert(key)
        override def numPartitions = numberOfMachines

        def convert(key: Any): Int
    }


    implicit object minimalAlgorithmsPartitioner extends genericPartitioner[Int] {
        def convert(key: Any): Int = {
            val machineId = key.asInstanceOf[Int]
            assert(0 <= machineId && machineId <= lastMachineId, s"bad machineId $machineId")
            machineId
        }
    }


    def mapReduce[K : ClassTag, T1, T2 : ClassTag, R : ClassTag](dataset: RDD[T1],
                                mapFun: (T1) => (K, T2),
                                reduce: ((K, Iterable[T2])) => R)
                                (implicit partitioner: genericPartitioner[K]): RDD[R] = {

        return dataset.map(mapFun).groupByKey(partitioner).map(reduce)
    }


    def mapReduce[K : ClassTag, T1, T2 : ClassTag, R : ClassTag](dataset: RDD[T1],
                                mapFun: T1 => List[(K, T2)],
                                reduce: ((K, Iterable[T2])) => R)
                                (implicit partitioner: genericPartitioner[K],
                                d: DummyImplicit): RDD[R] = {

        return dataset.flatMap(mapFun).groupByKey(partitioner).map(reduce)
    }


    def mapReduce[K : ClassTag, T1, T2 : ClassTag, R : ClassTag](dataset: RDD[T1],
                                mapFun: T1 => (K, T2),
                                reduce: ((K, Iterable[T2])) => List[R])
                                (implicit partitioner: genericPartitioner[K],
                                d1: DummyImplicit,
                                d2: DummyImplicit): RDD[R] = {

        return dataset.map(mapFun).groupByKey(partitioner).flatMap(reduce)
    }


    def mapReduce[K : ClassTag, T1, T2 : ClassTag, R : ClassTag](dataset: RDD[T1],
                                mapFun: T1 => List[(K, T2)],
                                reduce: ((K, Iterable[T2])) => List[R])
                                (implicit partitioner: genericPartitioner[K],
                                d1: DummyImplicit,
                                d2: DummyImplicit,
                                d3: DummyImplicit): RDD[R] = {

        return dataset.flatMap(mapFun).groupByKey(partitioner).flatMap(reduce)
    }
}
```

Metody algorytmów minimalnych są w singleton object bo wystarczy jedna ich instancja tak samo jak implementując list wystarczy nam jeden obiekt Nil pustej listy typu List[Nothing]

Każda metoda implementująca minimalny algorytm po prostu implementuje metody map i reduce i wykonuje rundę. Przykładowo perfectBalanceSort:

```
def perfectBalanceSort[E : ClassTag, T : ClassTag](dataset: RDD[E],
                                                   pickToSort: E => T,
                                                   rddSize: Option[Int] = None)
                                                   (implicit ev: T => Ordered[T]): RDD[(E, Rank)] = {

    assert(numberOfMachines > 0, "number of machines must be non-zero!")

    val n = if (rddSize.isDefined) rddSize.get else dataset.count()
    if (n == 0) return dataset.map( e => (e, 0))

    val t = numberOfMachines
    val m = (n.toDouble / t).ceil.toInt

    def roundMap(rankingPair: (E, Rank)): (MachineId, (E, Rank)) = {
        val (elem, rank) = rankingPair
        val receiverPartitionId = (rank-1) / m
        (receiverPartitionId, (elem, rank))
    }

    def roundReduce(keyIterable: (MachineId, Iterable[(E, Rank)])): List[(E, Rank)] = {
        keyIterable._2.toList.sortBy{ case (elem, rank) => rank }
    }

    val rddRanking = ranking[E, T](dataset, pickToSort, rddSize)
    val prefixSumResult = mapReduce(rddRanking, roundMap(_), roundReduce(_))

    prefixSumResult
}
```

Do shufflowania danych między map i reduce używane są proste typy algebraiczne Scali przez co trudniej zamienić coś źle miejscami w kodzie bo kompilator się zorientuje.

```
sealed trait ShuffleData extends java.io.Serializable
final case class MachineAggregation(val aggregatedFrom: Int, val partialResult: Array[W]) extends ShuffleData
final case class RelevantToMachine(val dataWithRank: Set[(E, Int)]) extends ShuffleData
final case class MachineData(val data: Array[(E, Int)]) extends ShuffleData
```

Dostępne są dwie metody sliding window aggregation, jedna dla jednej wagi i druga dla wielu wag typu W. Jeteśmy wstanie liczyć zbiór wag różnych typówi jeśli za W = Any i musimy wtedy poprawnie castować w liczących wagę i agregacje funkcjach oraz wynik.

```
def slidingAggregationMany[E : ClassTag, T : ClassTag, W : ClassTag](dataset: RDD[E],
                                pickToSort: E => T,
                                windowSize: Int,
                                weights: List[E => W],
                                aggregates: List[List[W] => W],
                                rddSize: Option[Int] = None)
                                (implicit ev: T => Ordered[T]): RDD[(E, Array[W])] = {
```

```
def slidingAggregation[E : ClassTag, T : ClassTag, W : ClassTag](dataset: RDD[E],
                                pickToSort: E => T,
                                windowSize: Int,
                                weight: E => W,
                                aggregate: List[W] => W,
                                rddSize: Option[Int] = None)
                                (implicit ev: T => Ordered[T]): RDD[(E, W)] = {
```

Ilość elementów RDD[E] n jeśli nie zostanie przekazana zostanie obliczona ale jeśli algorytm wywoła kolejne minimalne algorytmy będzie już przekazana i nie liczona drugi raz. Pomocny był implicit conversion Int -> Option[Int] i default argument n = None.

Chciałem spróbować uogólnić rundę do traitu mniej więcej w ten sposób:

```
trait Round[] {
    def roundMap()

    def roundReduce()

    def roundResult(roundStart) = mapReduce(roundStart, roundMap(_), roundReduce(_))

    def algorithmResult(roundStart) = super.algorithmResult(roundResult(roundStart))
}
```

Tak by ktoś implementujący algorytm musiał zdefiniować tylko tyle map i reduce ile rund w singleton object oznaczającym algorytm a metoda apply będzie już gotowa i będzie używać stackable modyfications do przechodzenia między rundami w kolejności jak zostały wmieszane.

Natomiast nie wiadomo jakie typy brały i zwracałyby map i reduce oraz musiało by być kilka podobnych traitów gdy jest więcej niż jedna runda.

Staram się potrzebne rzeczy importować lokalnie