DATA ACQUISITION AND PREPARATION

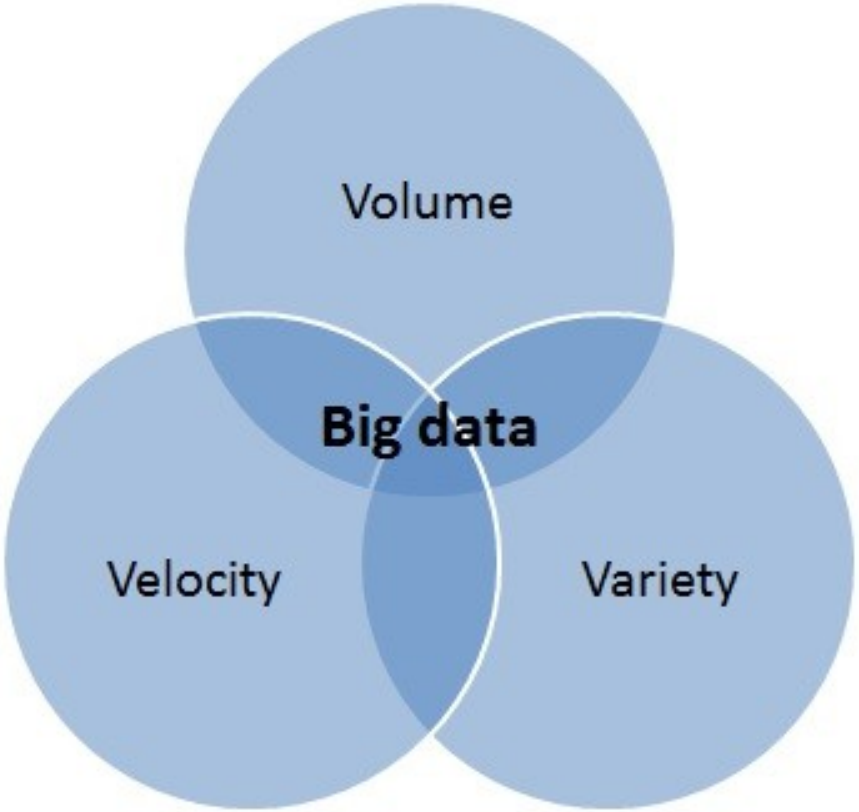
Caso ARPA

Argomenti della lezione

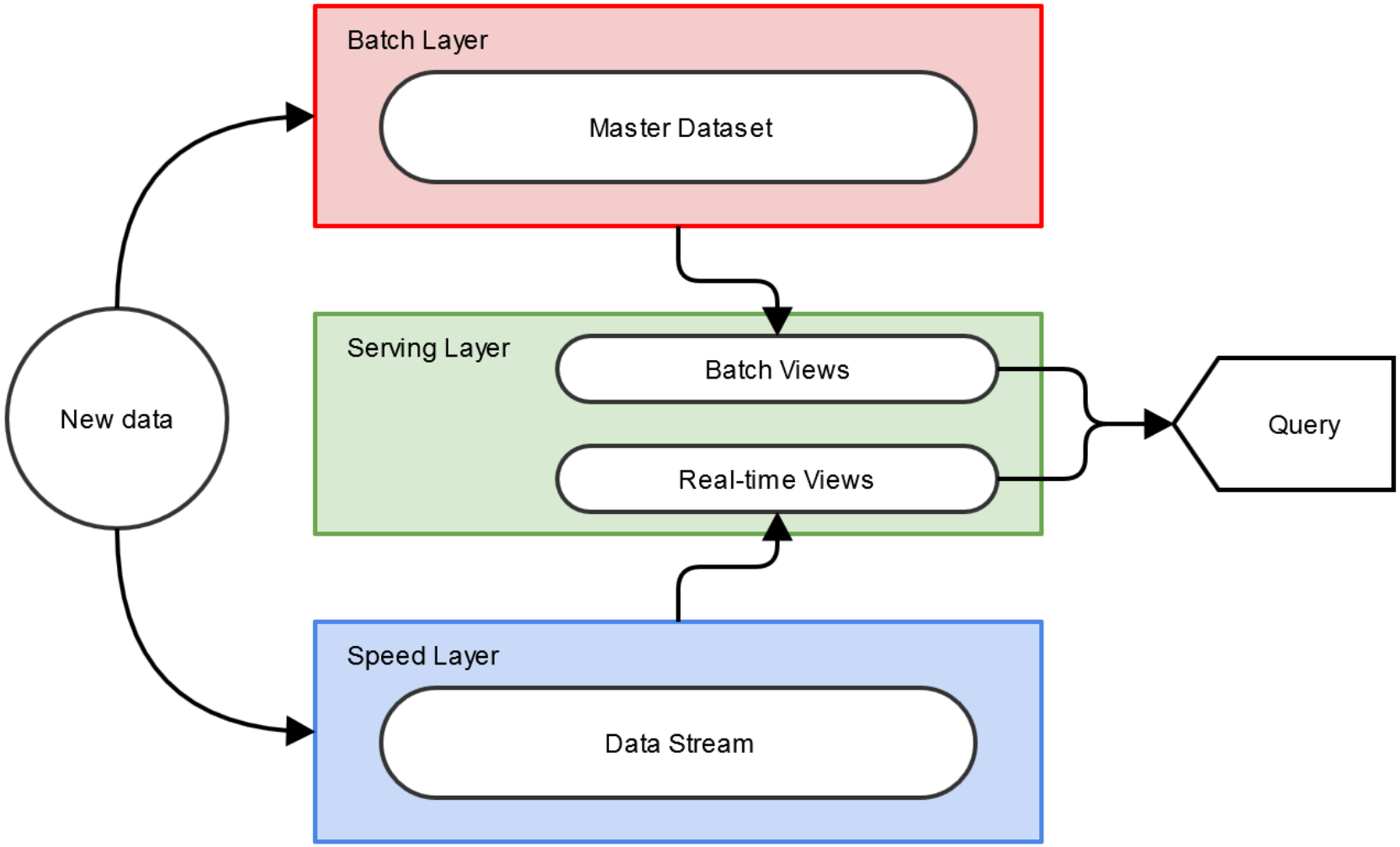
- I sistemi di gestione Big Data come mezzo per l'acquisizione e la preparazione dei dati
- La Lambda Architecture
- Il caso ARPA: definizione del problema e costruzione della soluzione.

Big Data

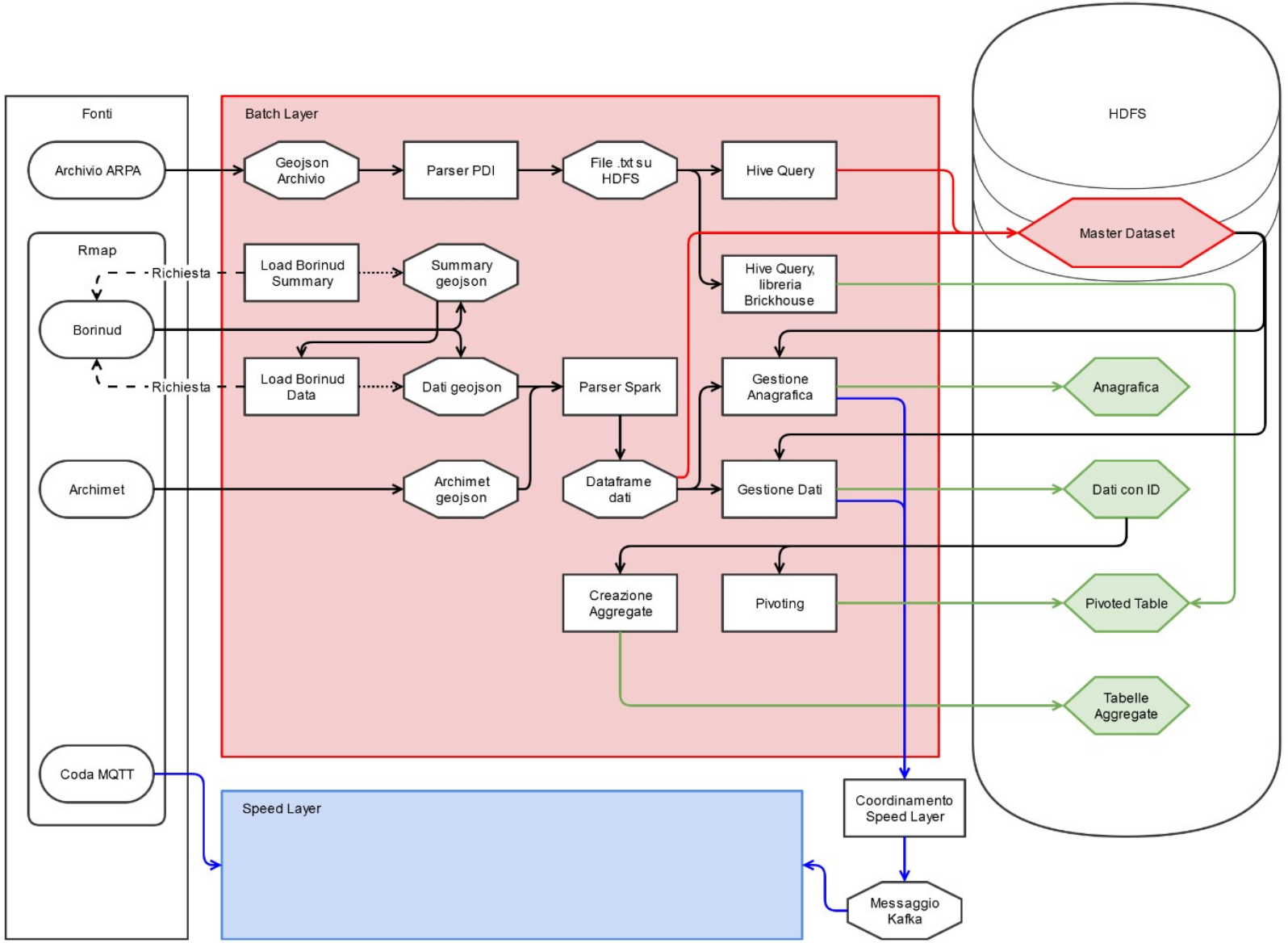
Big Data



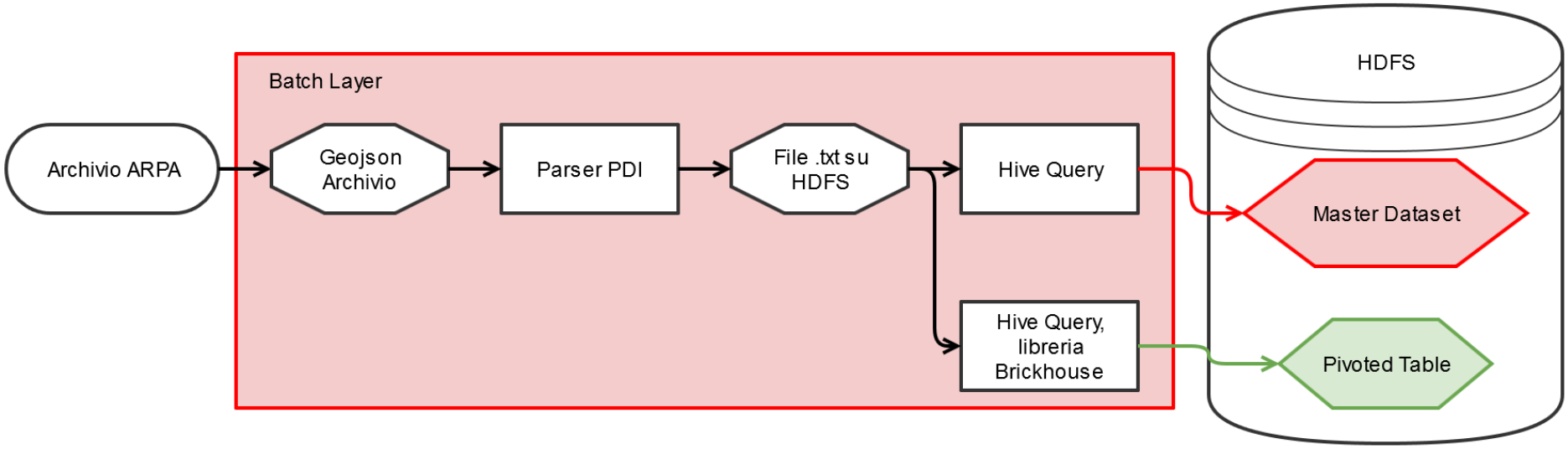
Lambda Architecture



Implementation



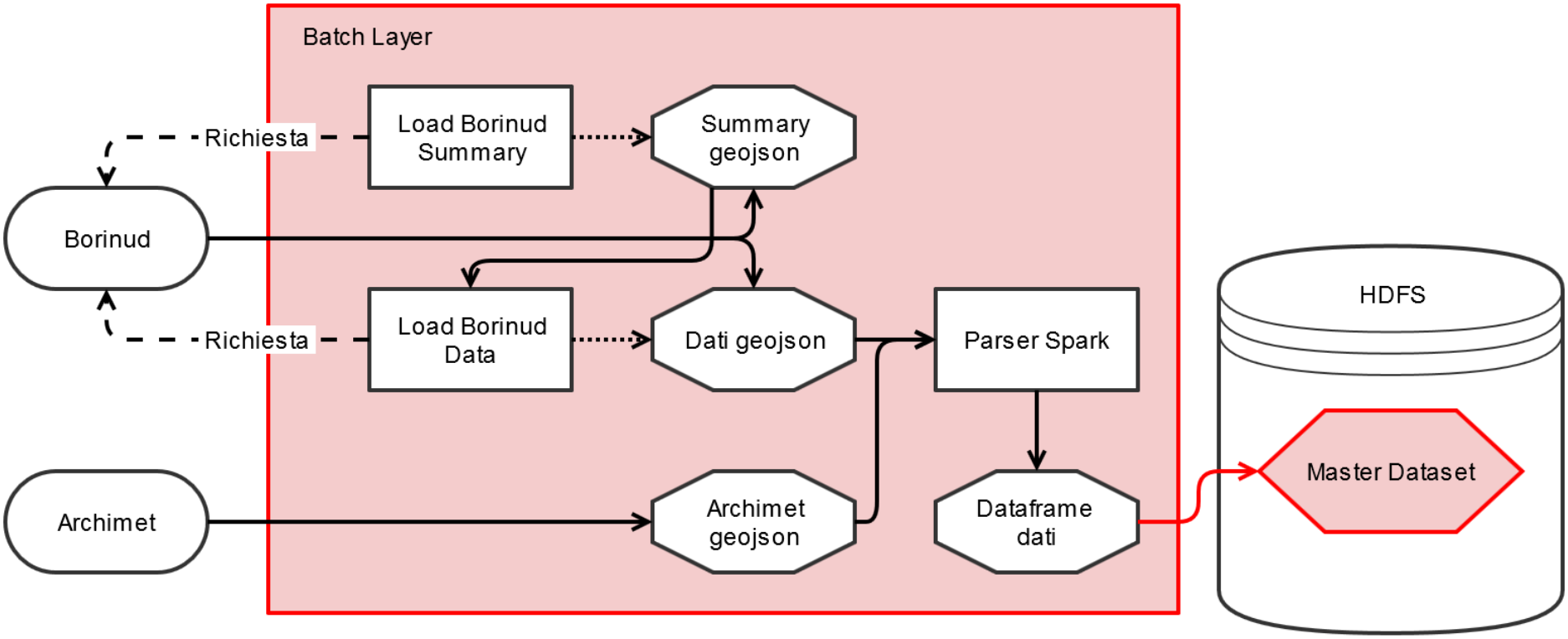
First steps


Batch Layer
Archivio ARPA
Geojson Archivio
Parser PDI
File .txt su HDFS
Hive Query
Hive Query, libreria Brickhouse
HDFS
Master Dataset
Pivoted Table

First steps (2)

- Creazione del master dataset
- Studio della struttura della tabella finale

Data acquisition



Data acquisition (2)

- Aggiornamento del master dataset

Data acquisition (3)

```
def loadBorinudDati(d: Int, m: Int, y: Int, bcode: String,
                    summa: org.apache.spark.sql.DataFrame
                   ): org.apache.spark.sql.DataFrame = {
    val targetAnno = y.toString
    val targetMese = if (m <= 9)
      "0" + m.toString
    else m.toString
    val targetGiorno = if (d <= 9)
      "0" + d.toString
    else d.toString
    var startCount = 0
    var stop = false
    var flag = false
//trasformazioni summary, estrazione codici utili
    val summary = summa.filter(summa("bcode") === bcode)
    val morphRichiesta = (x: Any) =>
      if (x == null) "-" else x.toString
```

Data acquisition (4)

```
//costruzione richieste
val arrayTemp = summary.collect()
val arrayReq = new Array[String](arrayTemp.length)
for (i <- 0 until arrayTemp.length) {
  val ident = morphRichiesta(arrayTemp(i)(0)) + "/"
  val lonlat = morphRichiesta(arrayTemp(i)(1)) + "," +
    morphRichiesta(arrayTemp(i)(2)) + "/"
  val network = morphRichiesta(arrayTemp(i)(3)) + "/"
  val trange = morphRichiesta(arrayTemp(i)(4)) + "," +
    morphRichiesta(arrayTemp(i)(5)) + "," +
    morphRichiesta(arrayTemp(i)(6)) + "/"
  val level = morphRichiesta(arrayTemp(i)(7)) + "," +
    morphRichiesta(arrayTemp(i)(8)) + "," +
    morphRichiesta(arrayTemp(i)(9)) + "," +
    morphRichiesta(arrayTemp(i)(10)) + "/"
  arrayReq(i) = rmapURL + ident + lonlat + network +
    trange + level + bcode + "/timeseries/" + targetAnno +
    "/" + targetMese + "/" + targetGiorno
}
```

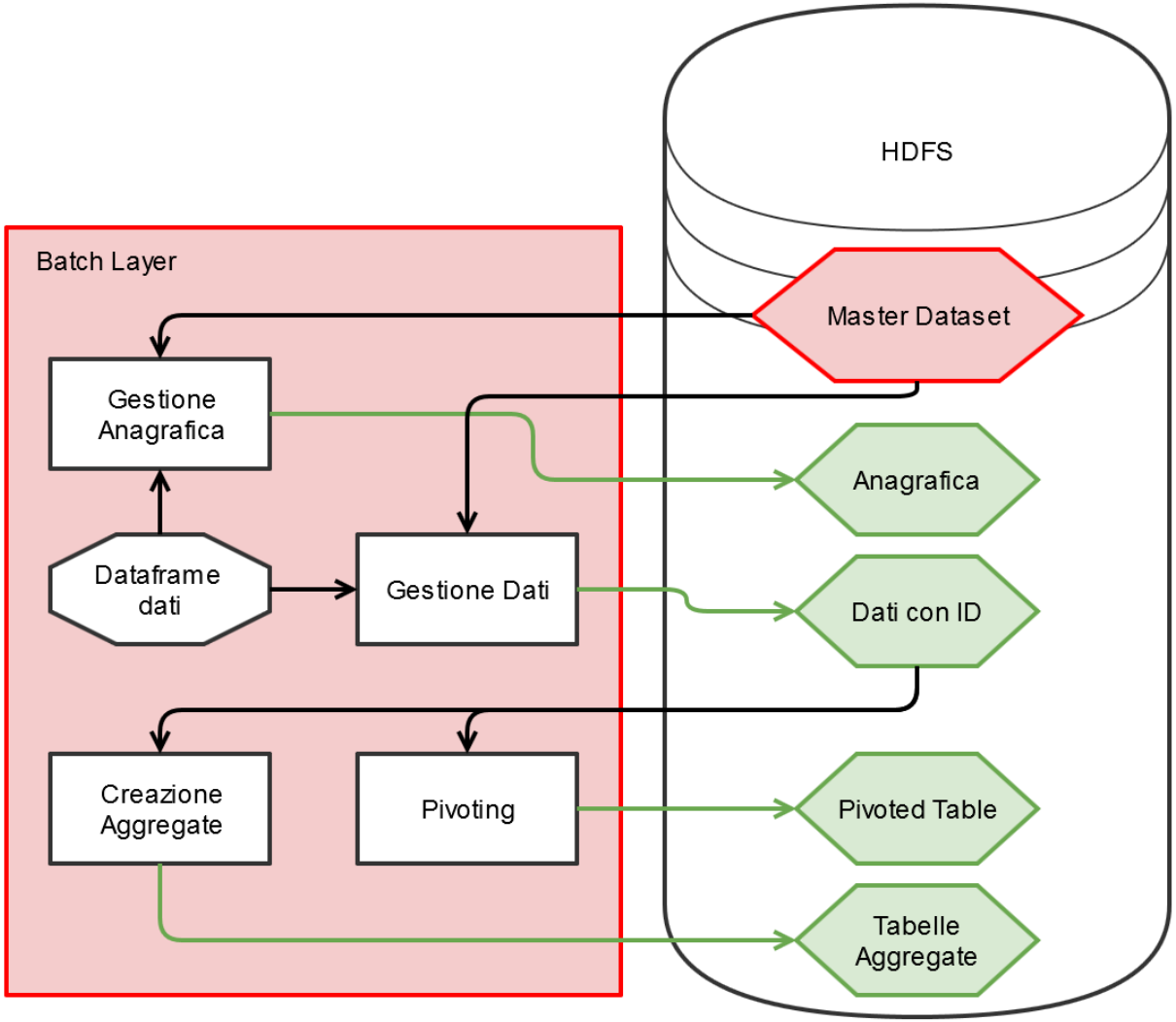
Data acquisition (5)

```
// Verifica la presenza dei dati.
val datiString = "vuoto"
while (stop == false || startCount != arrayReq.length) {
  val datiRaw = Source.fromURL(arrayReq(startCount))
  startCount = startCount + 1
  if (startCount == arrayReq.length) stop = true
  val datiString = datiRaw.mkString
  if (datiString !=
    """{"type": "FeatureCollection", "features": []}""") {
    println("Almeno una richiesta non vuota presente.")
    stop = true
    flag = true
  }
}
```

Data acquisition (6)

```
// Costruzione DF dati
if (flag == true) {
  val datiArray = Array(datiString)
  val datiRDD = sc.parallelize(datiArray)
  val datiJson = sqlContext.read.json(datiRDD)
  datiJson.registerTempTable("jsonTable")
  var dfResult = sql("SELECT f.properties.datetime, " +
    "f.properties.ident, f.properties.lon, " +
    "f.properties.lat, f.properties.network, " +
    "f.properties.bcode, f.properties.value " +
    "FROM jsonTable LATERAL VIEW explode(features) a AS f")
  for (i <- startCount until arrayReq.length) {
    val datiRaw = Source.fromURL(arrayReq(i))
    val datiString = datiRaw.mkString
    if (datiString !=
      """{"type": "FeatureCollection", "features": []}""") {
      val datiArray = Array(datiString)
      val datiRDD = sc.parallelize(datiArray)
      val datiJson = sqlContext.read.json(datiRDD)
      datiJson.registerTempTable("jsonTable")
      try {
        val dfTemp = sql("SELECT f.properties.datetime, " +
          "f.properties.ident, f.properties.lon, " +
          "f.properties.lat, f.properties.network, " +
          "f.properties.bcode, f.properties.value " +
          "FROM jsonTable LATERAL VIEW explode(features) a AS f")
        dfResult = dfResult.unionAll(dfTemp)
      } catch {
        case a: org.apache.spark.sql.AnalysisException => {
          println("JSON reading error")
        }}}}}
```

Data preparation



Data preparation (2)

- Costruzione delle batch view
- Creazione delle tabelle richieste per l'analisi

Data preparation (3)

```
def nonInAnagrafica(bcode: String,
                    anag: org.apache.spark.sql.DataFrame,
                    dati: org.apache.spark.sql.DataFrame
                   ): org.apache.spark.sql.DataFrame = {
  import org.apache.spark.sql.Row
  import org.apache.spark.sql.types._
  // Schema utilizzato dal DataFrame
  val schema = StructType(
    StructField("id", StringType, true) ::
      StructField("lat", DoubleType, true) ::
      StructField("lon", DoubleType, true) ::
      StructField("temp", IntegerType, true) ::
      StructField("prec", IntegerType, true) ::
      StructField("umid", IntegerType, true) :: Nil)
  // UDF per trasformare lat e lon da Double a String
  val latLonDaDouble = udf((lon: Double, lat: Double) =>
    (math round lon * 100000).toString +
      (math round lat * 100000).toString)
  var result = sqlContext
    .createDataFrame(sc.emptyRDD[Row], schema)
```

Data preparation (4)

```
  // Preparazione dati per il confronto
  val compattaAnag = anag
    .withColumn("lon+lat", latLonDaDouble(anag("lon"),
anag("lat")))
    .select("lon+lat", "lon", "lat")
  val arrayAnag = compattaAnag
    .select("lon+lat").rdd.map(r => r(0)
    .asInstanceOf[String]).collect()
  val listAnag = List.fromArray(arrayAnag)
  val compattaDati = dati
    .withColumn("lon+lat", latLonDaDouble(dati("lon"),
dati("lat")))
    .select("lon+lat", "lon", "lat")
    .distinct
  val arrayDati = compattaDati
    .select("lon+lat").rdd.map(r => r(0)
    .asInstanceOf[String])
    .collect()
  val listDati = List.fromArray(arrayDati)
```

Data preparation (5)

```
  // Confronto tra le nuove stazioni e l'anagrafica
  val filtro = listAnag.toSet
  val newAnagList = listDati.filterNot(filtro)
  if (newAnagList.isEmpty) {
    return result
  }
  val newDF = sc.parallelize(newAnagList).toDF
  val nuoviId = compattaDati
    .join(newDF, compattaDati("lon+lat") === newDF("_1"))
    .select("lon+lat", "lat", "lon")
  // Un txt per bcode con la combinazione di flag corrispondente
  val flags = sc.textFile("/Progetti/METEO/SISTEMA/" +
    bcode + ".txt")
    .map(_.split(", "))
    .map(f => Flag(f(0).toInt, f(1).toInt, f(2).toInt))
    .toDF()
  result = nuoviId.withColumnRenamed("lon+lat", "id")
    .join(flags)
  return result
}
```

Data preparation (6)

```
def batchPivot(datiDF: org.apache.spark.sql.DataFrame) = {
  val appoggio = datiDF.select("id").distinct.collect()
  // Creazione lista con i valori usati come pivot
  val idArray = new Array[String](appoggio.length)
  for (i <- 0 until appoggio.length) {
    idArray(i) = appoggio(i).getString(0)
  }
  val idList = List.fromArray(idArray)
  // Inserisce il valore corrispondente all'id o 0 altrimenti
  def genCase(x: String) = {
    when($"id" <=> lit(x), $"valore").otherwise(0).alias(x)
  }
  def genAgg(f: org.apache.spark.sql.Column =>
    org.apache.spark.sql.Column)(x: String) = f(col(x)).alias(x)
  // Creazione DF con pivot
  val pivotDF = datiDF
    .select($"data" :: idList.map(genCase): _*)
    .groupBy($"data")
    .agg($"data".alias("tempCol"), idList.map(genAgg(sum)): _*)
    .drop("tempCol")
   pivotDF.write.mode(SaveMode.Overwrite).saveAsTable("PivotTable")
}
```