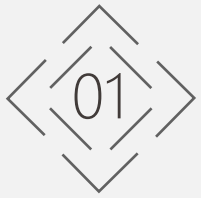
Movie Recommendation System

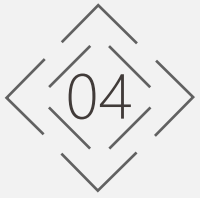
Yunlu Liaozheng
Lingfeng Zhou
Cheng Jin



Goal Review and Completion Report

System Structure

Demo Presentation and Test Result



Implementation Detail

Part One

Goal Review & Completion Report

Goal

Build a scalable movie recommendation System to answer the question “What movie should I watch tonight?”. It will return some related movies based on the given movie.

Acceptance criteria:
The system should return answers within 3 seconds and can handle maximum 10 requests per second.

Three modules of the system:

Controller: The frontend contains webserver, user interface, session management, authentication.

Model: The model contains main logic of generating response and manipulating data.

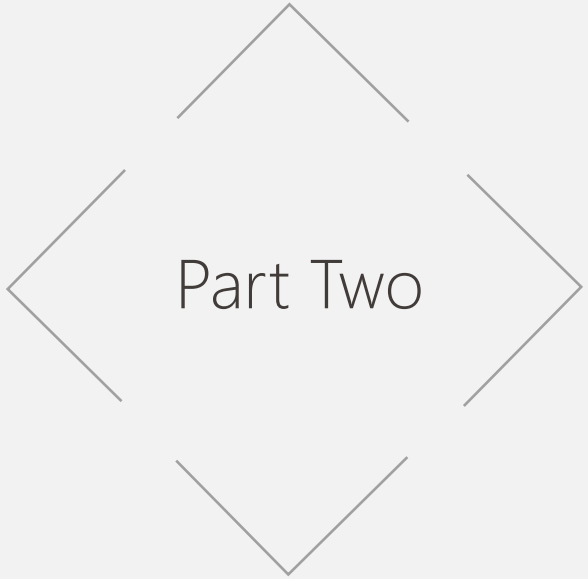
Datacenter: The data manager contains all calculations and low-level data management.

Completion

Controller: Controller was implemented as a stateless server based on Scala Play framework. As it is stateless, it is safe to deploy multiple servers, users can access each of them.

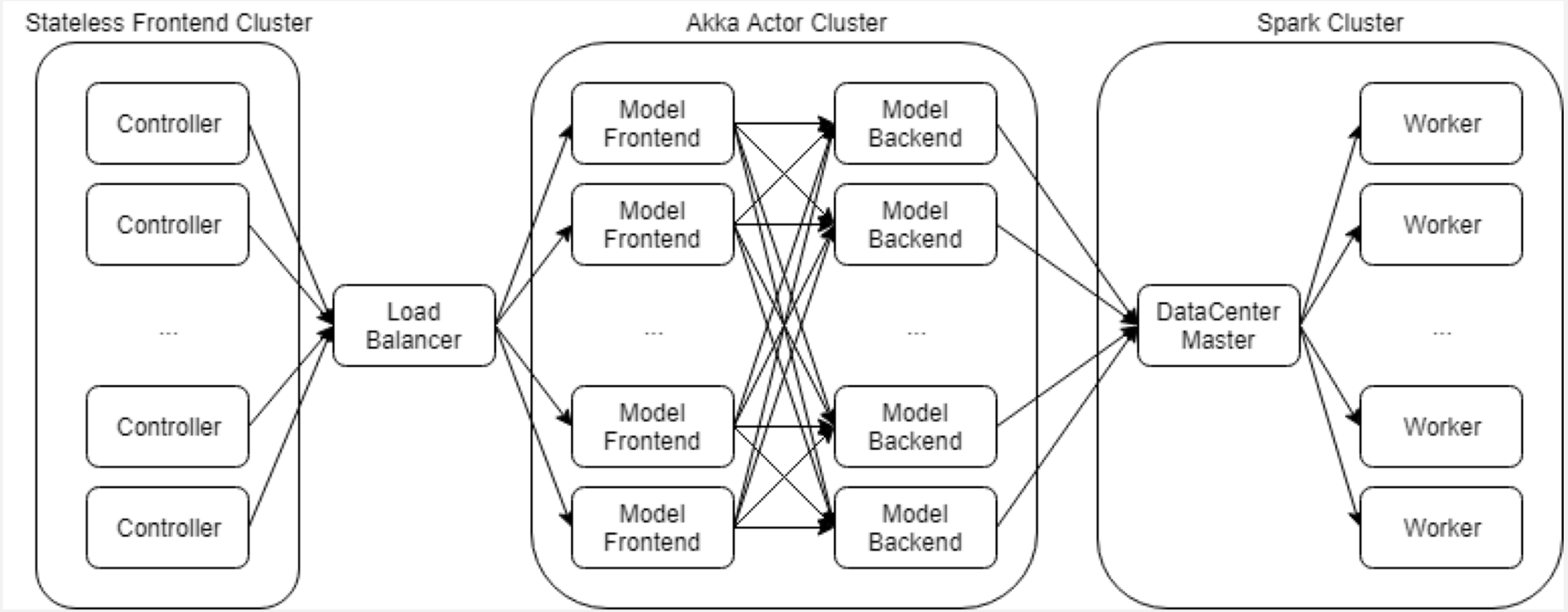
Model: Model part was implemented as a frontend/backend cluster based on Akka Actor Cluster. The frontend handled requests comes from Controller, and forward requests to backend. The backend is used to do real works.

Datacenter: Datacenter part was implemented as a Spark Cluster, which manages over 1,000,000 movies.



System Structure

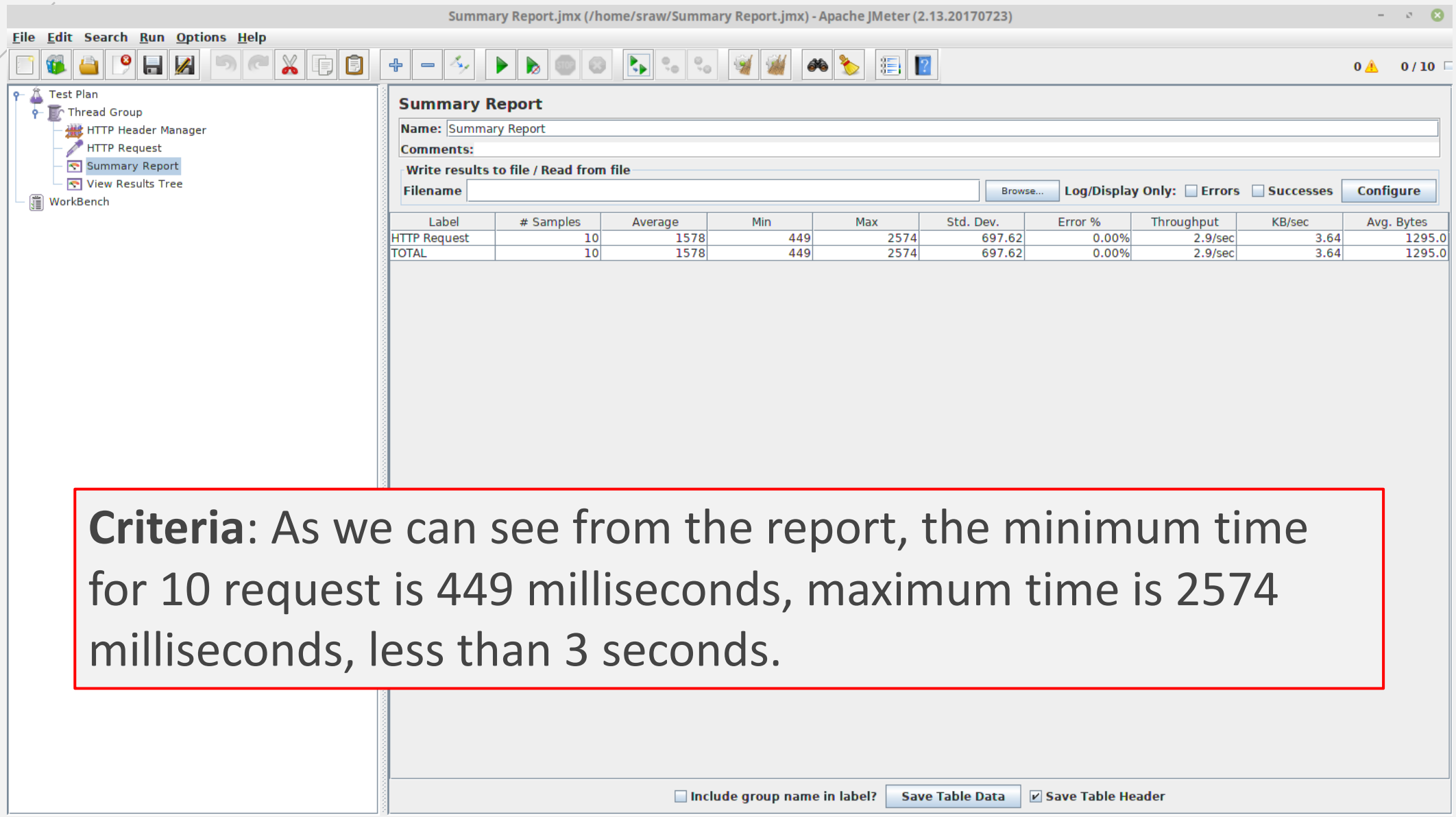
System Structure



Part Three

Demo Presentation & Test Result

Test Result



Criteria: As we can see from the report, the minimum time for 10 request is 449 milliseconds, maximum time is 2574 milliseconds, less than 3 seconds.

Part Four

Implementation Detail

Akka Http

```
class ModelFrontend(port: Int) extends HttpRoute {
  val config: com.typesafe.config.Config = Config.getFrontendConfig(port)

  implicit val system: ActorSystem = ActorSystem("ModelCluster", config)
  implicit val executor: ExecutionContextExecutor = system.dispatcher
  implicit val materializer: ActorMaterializer = ActorMaterializer()
  val logger = Logging(system, getClass)

  val postSender: ActorRef = system.actorOf(Props[PostSender], name = "frontend")

  private val mode = config.getString( path = "frontend.mode")

  private val interface: String = config.getString( path = s"frontend.$mode.interface")
  private val HttpServerPort: Int = config.getInt( path = s"frontend.$mode.port")

  def start : Unit  {
    start(HttpServerPort)
  }

  private def start(HttpServerPort: Int): Unit = {
    Http().bindAndHandle(route, interface, HttpServerPort) onComplete {
      case Success(_) => logger.info(s"Server listens on $interface:$HttpServerPort")
      case Failure(exception) => exception.getCause match {
        case _: java.net.BindException => start(HttpServerPort + 1)
        case causeException => causeException.printStackTrace()
      }
    }
  }
}
```

Akka actor cluster

```
class StaticModel(config: Config, dataSourceOption: Option[RequestEntity => HttpResponse]) extends Actor with ActorLogging with Js
  def this(config: Config) = this(config, None)

  import context.dispatcher
  val cluster = Cluster(context.system)

  override def preStart(): Unit = cluster.subscribe(self, classOf[MemberUp])
  override def postStop(): Unit = cluster.unsubscribe(self)

  final implicit val materializer: ActorMaterializer = ActorMaterializer(ActorMaterializerSettings(context.system))

  val http = Http(context.system)

  def register(member: Member): Unit = {
    if (member.hasRole( role = "frontend")) context.actorSelection(RootActorPath(member.address) / "user" / "frontend") ! BackendReg
  }

  def receive: PartialFunction[Any, Unit] = {
    case Movie(title) => handleRequest(Marshal(MovieRequest(title, 3)).to[RequestEntity])
    case HttpResponse(StatusCodes.OK, _, entity, _) =>
      log.debug(s"Receive movie info: $entity")
      handleMovieInfo(Unmarshal(entity.withContentType(ContentTypes.`application/json`)).to[MovieResponse])
    case resp@HttpResponse(code, _, _, _) =>
      log.info("Request failed, response code: " + code)
      resp.discardEntityBytes()
    case state: CurrentClusterState =>
      state.members.filter(_.status == MemberStatus.Up) foreach register
    case MemberUp(m) => register(m)
  }
```

Spark cluster

```scala
object DoKmeans {
    val spark = SparkSession.builder
      .master(Config.getSparkMaster())
      .appName(Config.getSparkName())
      .getOrCreate()
    spark.sparkContext.setLogLevel(Config.getLogLevel())
    import spark.implicits._

    def train(): DataFrame = {


        val file_basic = "DataCenter/src/main/resources/title.basics.tsv"
        val df_basic_raw = spark.read.format( source = "csv")
          .option("header", "true")
          .option("inferSchema", "true")
          .option("delimiter", "\t")
          .load(file_basic)
          .cache()
        df_basic_raw.printSchema()
        val df_basic = df_basic_raw.filter( conditionExpr = "titleType == 'movie'")
          .filter( conditionExpr = "runtimeMinutes != '\\\\N' AND genres != '\\\\N'")
          .drop( colNames = "endYear", "isAdult")
        //    val genre = df_basic.select("genres").map(attributes => attributes.toString().split(",").head)


        val file_names = "DataCenter/src/main/resources/name.basics.tsv"
        val df_names_raw = spark.read.format( source = "csv")
```

Optimization

```scala
object GetResults {
  val spark = SparkSession.builder
    .master(Config.getSparkMaster())
    .appName(Config.getSparkName())
    .getOrCreate()
  spark.sparkContext.setLogLevel(Config.getLogLevel())
  import spark.implicits._
  val df = spark.read.parquet( path = "DataCenter/src/main/resources/result.parquet").cache()
  def getmovies(title: String, limit: Int): String = {

    //read result  select required rows

    val movie_df = df.select( col = "primaryTitle",  cols = "genresSeq", "director","1actor","rms",
      "averageRating", "numVotes","syear","prediction","distanceFromCenter")

    val newNames = Seq("title", "genres", "director", "actor", "length", "score", "voteNumber",
      "year","prediction","distanceFromCenter")

    val dfRenamed = movie_df.toDF(newNames: _*)
    //get cluster
    val cluster = dfRenamed.select( col = "title", cols = "prediction").filter( condition = $"title"===title).limit(1)("predict
    //sort with distance
    val related_df: Dataset[Row] = dfRenamed.filter( condition = $"prediction" === cluster).sort(asc( columnName = "distanceF

    val all = related_df.filter( condition = $"prediction" === cluster)
      .drop( colNames = "prediction","distanceFromCenter")
    val related = all.filter( condition = $"title"=!=title).limit(limit)
    val origin_df = all.filter( condition = $"title"===title)
    val res = origin_df.union(related).na.fill( value = "").toJSON.collect()
    val origin_json = res(0)
    val related_json = res.drop(1).mkString("\"related\": [", "," , "]" )

    "{ \"origin\": ".concat(origin_json).concat( str = ",").concat(related_json).concat( str = " }")
```

Only one action used

Unit Test

```
class StaticModelSpec() extends TestKit(ActorSystem("StaticModelSpec")) with ImplicitSender
  with WordSpecLike with Matchers with BeforeAndAfterAll {

  def GetResult(path: String): MovieResponse = {
    val getDataSource: String => RequestEntity => HttpResponse =
      entity => { _ => HttpResponse(entity = entity, headers = List(headers.`Content-Type`(ContentTypes.`application/jso
    val fileStream = getClass.getResourceAsStream(path)
    val entity = Source.fromInputStream(fileStream).mkString
    val staticModel = TestActorRef(Props(classOf[StaticModel], ConfigFactory.empty(), Some(getDataSource(entity))))
    implicit val timeout: Timeout = 3.seconds
    val future = staticModel ? Movie("asdfasd")
    val Success(result) = future.mapTo[MovieResponse].value.get
    result
  }

  override def afterAll: Unit = {
    TestKit.shutdownActorSystem(system)
  }

  "StaticModel actor" must {

    "work with response1.json" in {
      val result = GetResult( path = "response1.json")
      result shouldBe MovieResponse(MovieInfo("Test", List(), "", "", 100, 8.9, 1000, 1980), List())
    }

    "work with response2.json" in {
      val result: MovieResponse = GetResult( path = "response2.json")
      result.related.length shouldBe 6
```

Thanks !