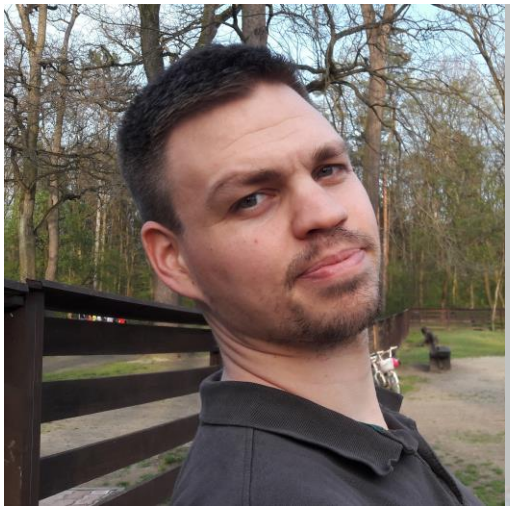



Dr. Hegedűs Péter, Dr. Ferenc Rudolf

Nagyméretű adatbázisok

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

Hadoop feletti végrehajtó motorok

Összefoglalás

Az olvasó ebben az olvasóleckében betekintést nyer az általános célú BigData feldolgozó, elosztott végrehajtó motorok világába. Bemutatjuk, mik a klasszikus MapReduce végrehajtó motor (a Hadoop alapértelmezett feladat végrehajtója) hiányosságai, amik sok korszerű BigData alkalmazás hatékony megvalósítását korlátozhatják. Áttekintést adunk arra vonatkozóan, hogy milyen újabb, általánosabb és bizonyos felhasználási esetekben hatékonyabb adat feldolgozó keretrendszerek léteznek, valamint hogy hogyan viszonyulnak ezek a MapReduce-hoz, illetve a Hadoop ökoszisztémához.
A lecke fejezetei:

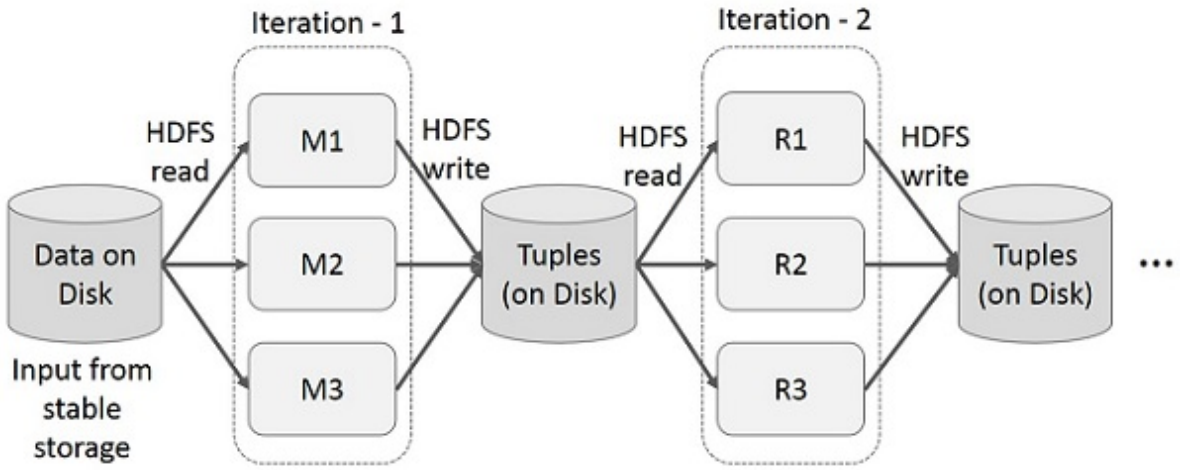
- 1. fejezet: **A MapReduce engine hiányosságai, általánosabb alternatíváinak bemutatása (olvasó)**
- 2. fejezet: **Az Apache Tez keretrendszer ismertetése (olvasó)**
- 3. fejezet: **Az Apache Spark keretrendszer ismertetése (olvasó)**

Téma típusa: **elméleti**
Olvasási idő: **40 perc**

📖 1. fejezet
A MapReduce koncepció hátrányai, modernebb és általánosabb alternatívái

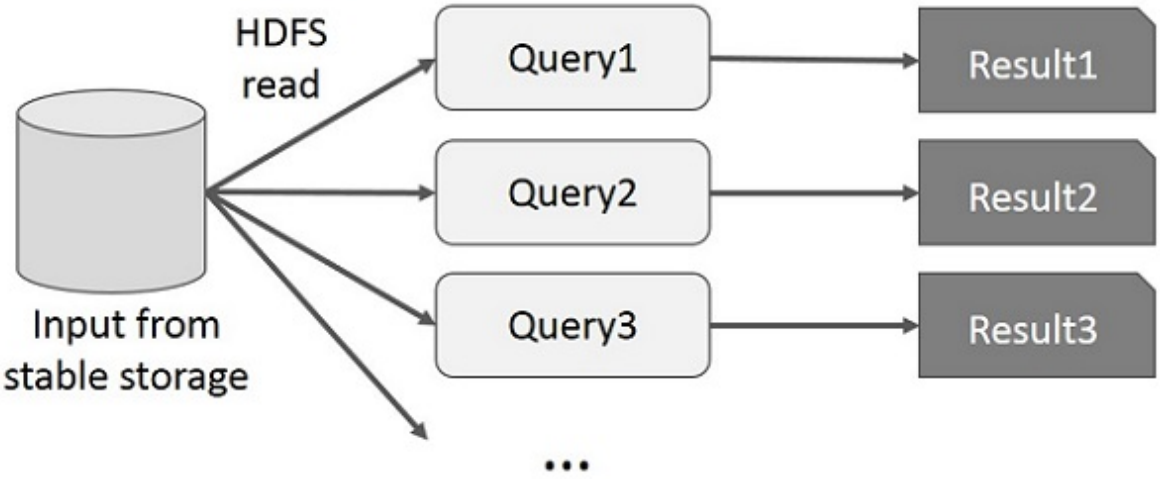
A Hadoop keretrendszer alapértelmezett számítás végrehajtó motorja a MapReduce [1]. Ez kiválóan alkalmas kötegelt feldolgozáshoz, a map és reduce műveletek klaszteren elosztott, párhuzamos végrehajtásához. A hatékonyságot leginkább az által éri el, hogy nem a nagy adatokat mozgatja a klaszter node-jai között, hanem az azokat feldolgozó programkódot másolja arra a csomópontra, ahol az adatok is vannak. Így a műveletek lokális adatokon, párhuzamosan futnak, és a végén az eredmények összegződnek. Vannak azonban bizonyos használati esetek, amikor ez a megközelítés nem alkalmazható. Ilyenek a következők:

- **Iteratív alkalmazások:** az ilyen alkalmazásoknak ugyan megfelelő lenne a kötegelt feldolgozás, viszont az eredményt nem tudják egy job segítségével előállítani, több job kombinálására van szükség, ahol bizonyos job-ok más job-ok eredményeit használják bemenetként. A MapReduce megvalósítások esetében az adatok megosztása két job között csak úgy valósulhat meg, hogy a köztes eredmények a fájlrendszerre (pl. HDFS) íródnak, ami lassú (lásd ábra).



> https://www.tutorialspoint.com/apache_spark/images/iterative_operations_on_mapreduce.jpg

- **Interaktív alkalmazások:** az ilyen alkalmazások lehetővé teszik ad-hoc lekérdezések végrehajtását, amelyekre a felhasználó közel valós idejű választ vár. Mivel minden lekérdezés közvetlenül a fájlrendszerről tölti be az inputot (lásd ábra), ami nagyságrendekkel hosszabb időt vehet igénybe, mint maga a számítási művelet, a megoldás túl lassú lesz ehhez a fajta felhasználáshoz.



> https://www.tutorialspoint.com/apache_spark/images/interactive_operations_on_mapreduce.jpg

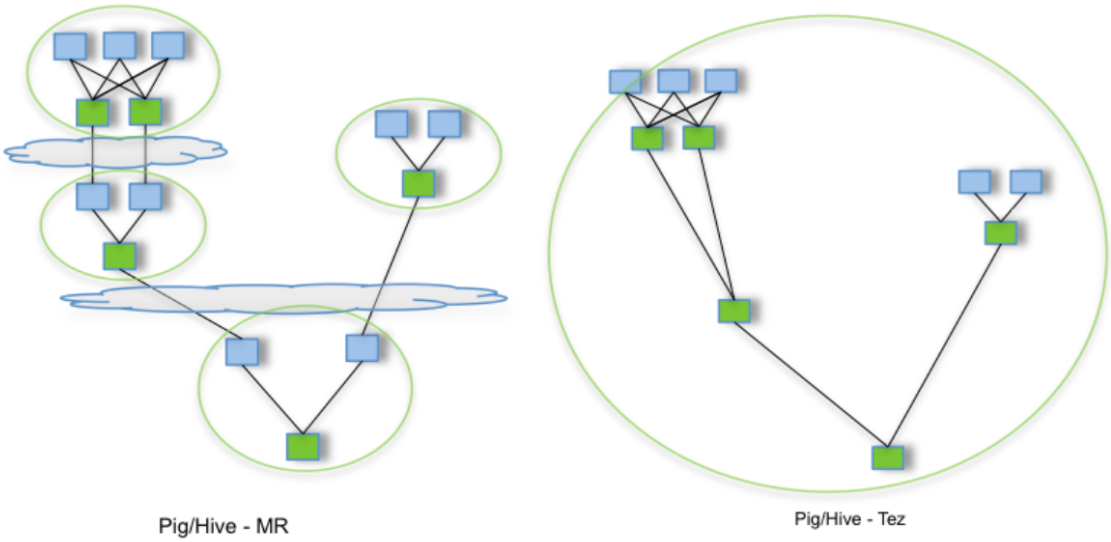
Jól látszik, hogy a fenti két tipikus BigData felhasználási mód nagyságrendekkel gyorsabb adatmegosztást követel a párhuzamosan futó job-ok között. Mivel mind az iteratív mind pedig az interaktív alkalmazások nagyon népszerűek, szükség volt a klasszikus MapReduce-tól eltérő megközelítések alkalmazására és újabb végrehajtó motorok fejlesztésére. A MapReduce két legelterjedtebb alternatívája/utódja/fejlesztése az alábbi:

- **Apache Tez** [2]: az Apache Hadoop YARN-ra [4] épülő adatfeldolgozó motor, amely képes komplex, ***körmentes irányított gráfba (DAG) szervezett feladatok*** hatékony és elosztott végrehajtására. Mivel a job-okat össze lehet fűzni, és egyszerre végrehajtani őket, ami korábban több MapReduce job futtatását igényelte, a Tez jelentős teljesítmény növekedést hoz a MapReduce-hoz képest.
- **Apache Spark** [3]: egy villámgyors klaszter számítási keretrendszer, amit nagyon gyors adatfeldolgozásra terveztek. A Hadoop MapReduce modellen alapul, de olyan módon általánosítja és terjeszti ki azt, ami lehetővé teszi a hatékony felhasználását interaktív lekérdezések készítéséhez vagy stream feldolgozáshoz is. Az alapvető technika, ami ezt a fajta sebesség növekedést lehetővé teszi, a ***memóriában tárolt klaszter számítási modell***.

📖 2. fejezet

Az Apache Tez keretrendszer

Az Apache Tez [2] az Apache Hadoop YARN-ra épülő adatfeldolgozó motor, amely képes komplex, körmentes irányított gráfba (DAG) szervezett feladatok hatékony és elosztott végrehajtására. Mivel a job-okat össze lehet fűzni, és egyszerre végrehajtani őket, ami korábban több MapReduce job futtatását igényelte, a Tez jelentős teljesítmény növekedést hoz a MapReduce-hoz képest.



> https://tez.apache.org/images/PigHiveQueryOnMR.png

A fenti ábrán jól látszik, hogy ha például egy Hive SQL lekérdezés a fenti módon több különböző forrást használ (pl. tábla `JOIN` vagy egyéb összetett lekérdezés miatt), akkor az több MapReduce job-ot eredményez, amiket külön-külön végre kell hajtani, és az egyes job-ok (köztes) eredményei HDFS-en tárolásra kerülnek. Ezzel szemben ha a Tez végrehajtó motort használjuk (Hive teljes körűen támogatja), a fenti irányított körmentes gráfba rendezett feladatokat kapjuk, amelyek egyben futtathatók a köztes eredmények kiírása nélkül. Ez nagy teljesítmény javulást hoz a MapReduce-hoz képest, emlékezzünk vissza a Hive használata során megjelenő figyelmeztető üzenetekre: MapReduce helyett használjuk a Tez vagy a Spark futtató motort, mert az hatékonyabb!

A Tez az alábbi két csoportba sorolható tervezési elveket követi :

- **Felhasználást megkönnyítő funkciók**:
  - Kifejező adatfolyam leíró API-k nyújtása
  - Rugalmas bemenet-feldolgozó-kimenet futás idejű modell
  - Adat típus függetlenség
  - Egyszerűsített fejlesztés támogatása
- **Végrehajtási teljesítmény**:
  - Teljesítmény növekedés MapReduce-hoz képest
  - Optimális erőforrás kezelés
  - Futás idejű végrehajtási terv újra konfigurálás
  - Dinamikus fizikai adat folyam döntések támogatása

Az alábbi Java programkód [5] egy két node-ból álló Tez DAG létrehozását és futtatását demonstrálja, amely a klasszikus word count problémát oldja meg:

```
package io.github.ouyi.tez;
```

```
import com.google.common.base.Preconditions;
import com.google.common.collect.Sets;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.yarn.api.records.ApplicationId;
import org.apache.tez.client.CallerContext;
import org.apache.tez.client.TezClient;
import org.apache.tez.dag.api.*;
import org.apache.tez.dag.api.client.DAGClient;
import org.apache.tez.dag.api.client.DAGStatus;
import org.apache.tez.dag.api.client.StatusGetOpts;
import org.apache.tez.mapreduce.input.MRInput;
import org.apache.tez.mapreduce.output.MROutput;
import org.apache.tez.mapreduce.processor.SimpleMRProcessor;
import org.apache.tez.runtime.api.ProcessorContext;
import org.apache.tez.runtime.library.api.KeyValueReader;
import org.apache.tez.runtime.library.api.KeyValueWriter;
import org.apache.tez.runtime.library.api.KeyValuesReader;
import org.apache.tez.runtime.library.api.TezRuntimeConfiguration;
import org.apache.tez.runtime.library.conf.OrderedPartitionedKVEdgeConfig;
import org.apache.tez.runtime.library.partitioner.HashPartitioner;
import org.apache.tez.runtime.library.processor.SimpleProcessor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.util.Arrays;
import java.util.Optional;
import java.util.Set;
import java.util.StringTokenizer;

public class HelloWorld extends Configured implements Tool {

    private static final String INPUT = "Input";
    private static final String TOKENIZER_VERTEX = "TokenizerVertex";
    private static final String SUMMATION_VERTEX = "SummationVertex";
    private static final String OUTPUT = "Output";
    private static final int PARALLELISM = 1;
    private static final boolean ENABLE_SPLIT_GROUPING = true;
    private static final boolean GENERATE_SPLIT_IN_AM = true;
    private static final Logger LOGGER =
LoggerFactory.getLogger(HelloWorld.class);

    public static class TokenProcessor extends SimpleProcessor {
        private final IntWritable one = new IntWritable(1);
        private final Text word = new Text();

        public TokenProcessor(ProcessorContext context) {
            super(context);
        }

        @Override
```

```
        public void run() throws Exception {
            Preconditions.checkArgument(getInputs().size() == 1);
            Preconditions.checkArgument(getOutputs().size() == 1);
            KeyValueReader kvReader = (KeyValueReader)
getInputs().get(INPUT).getReader();
            KeyValueWriter kvWriter = (KeyValueWriter)
getOutputs().get(SUMMATION_VERTEX).getWriter();
            while (kvReader.next()) {
                StringTokenizer itr = new
StringTokenizer(kvReader.getCurrentValue().toString());
                while (itr.hasMoreTokens()) {
                    word.set(itr.nextToken());
                    kvWriter.write(word, one);
                }
            }
        }
    }

    public static class SumProcessor extends SimpleMRProcessor {

        public SumProcessor(ProcessorContext context) {
            super(context);
        }

        @Override
        public void run() throws Exception {
            Preconditions.checkArgument(getInputs().size() == 1);
            Preconditions.checkArgument(getOutputs().size() == 1);
            KeyValueWriter kvWriter = (KeyValueWriter)
getOutputs().get(OUTPUT).getWriter();
            KeyValuesReader kvReader = (KeyValuesReader)
getInputs().get(TOKENIZER_VERTEX).getReader();
            while (kvReader.next()) {
                Text word = (Text) kvReader.getCurrentKey();
                int sum = 0;
                for (Object value : kvReader.getCurrentValues()) {
                    sum += ((IntWritable) value).get();
                }
                kvWriter.write(word, new IntWritable(sum));
            }
        }
    }

    @Override
    public int run(String[] args) throws Exception {
        // Parse args
        LOGGER.info(Arrays.toString(args));
        String inputPath = args[0];
        String outputPath = args[1];
        boolean localMode = args.length > 2 ? Boolean.parseBoolean(args[2]) :
false;

        // Setup configs
        Configuration conf = Optional.ofNullable(getConf()).orElse(new
Configuration());
        TezConfiguration tezConf = new TezConfiguration(conf);
        if (localMode) {
            tezConf.setBoolean(TezConfiguration.TEZ_LOCAL_MODE, localMode);
```

```
            conf.set("fs.default.name", "file:///");

 conf.setBoolean(TezRuntimeConfiguration.TEZ_RUNTIME_OPTIMIZE_LOCAL_FETCH,
true);
        }

        // Create and run DAG
        TezClient tezClient = TezClient.create(getClass().getSimpleName(),
tezConf);
        DAG dag = createDAG(inputPath, outputPath, tezConf);
        return runDAG(dag, tezClient, tezConf);
    }

    private DAG createDAG(String inputPath, String outputPath, TezConfiguration
tezConf) {
        // Create the tokenizer vertex with the input data source and
TextInputFormat
        DataSourceDescriptor dataSourceDescriptor =
MRInput.createConfigBuilder(new Configuration(tezConf), TextInputFormat.class,
inputPath)
                .groupSplits(ENABLE_SPLIT_GROUPING)
                .generateSplitsInAM(GENERATE_SPLIT_IN_AM)
                .build();
        Vertex tokenizerVertex = Vertex.create(TOKENIZER_VERTEX,
ProcessorDescriptor.create(TokenProcessor.class.getName()))
                .addDataSource(INPUT, dataSourceDescriptor);

        // Create the summation vertex with the output data source and
TextOutputFormat
        DataSinkDescriptor dataSinkDescriptor = MROutput.createConfigBuilder(new
Configuration(tezConf), TextOutputFormat.class, outputPath)
                .build();
        Vertex summationVertex = Vertex.create(SUMMATION_VERTEX,
ProcessorDescriptor.create(SumProcessor.class.getName()), PARALLELISM)
                .addDataSink(OUTPUT, dataSinkDescriptor);

        // Create a key-value edge with Text key type and IntWritable value type
        OrderedPartitionedKVEdgeConfig edgeConfig =
OrderedPartitionedKVEdgeConfig
                .newBuilder(Text.class.getName(), IntWritable.class.getName(),
HashPartitioner.class.getName())
                .setFromConfiguration(tezConf)
                .build();
        Edge edge = Edge.create(tokenizerVertex, summationVertex,
edgeConfig.createDefaultEdgeProperty());

        DAG dag = DAG.create("HelloWorld DAG")
                .addVertex(tokenizerVertex)
                .addVertex(summationVertex)
                .addEdge(edge);
        return dag;
    }

    public int runDAG(DAG dag, TezClient tezClient, TezConfiguration tezConf)
throws TezException,
            InterruptedException, IOException {
        try {
            tezClient.start();
```

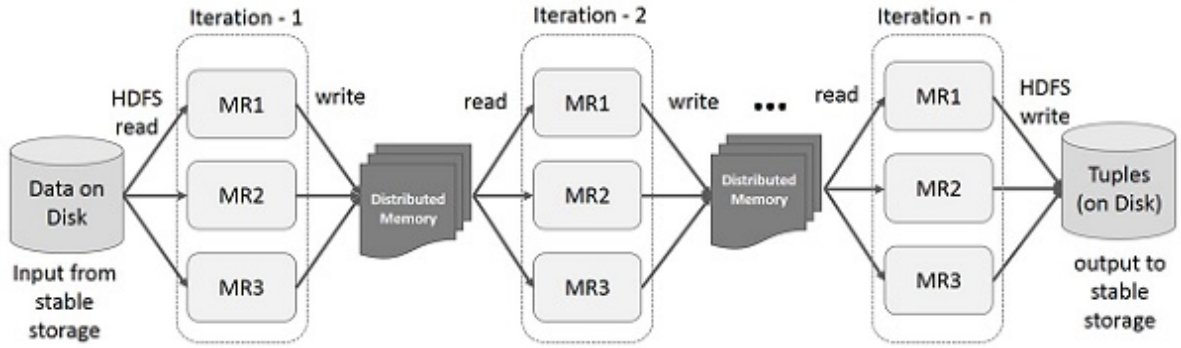
```
            tezClient.waitTillReady();

            // Set up caller context
            ApplicationId appId = tezClient.getAppMasterApplicationId();
            CallerContext callerContext =
CallerContext.create("HelloWorldContext", "Caller id: " + appId,
"HelloWorldType", "Tez HelloWorld DAG: " + dag.getName());
            dag.setCallerContext(callerContext);

            // Submit DAG and wait for completion
            DAGClient dagClient = tezClient.submitDAG(dag);
            Set<StatusGetOpts> statusGetOpts =
Sets.newHashSet(StatusGetOpts.GET_COUNTERS);
            DAGStatus dagStatus =
dagClient.waitForCompletionWithStatusUpdates(statusGetOpts);

            // Check status
            if (dagStatus.getState() == DAGStatus.State.SUCCEEDED) {
                return 0;
            } else {
                LOGGER.error("DAG diagnostics: " + dagStatus.getDiagnostics());
                return -1;
            }
        } finally {
            tezClient.stop();
        }
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(null, new HelloWorld(), args);
        System.exit(res);
    }
}
```

📖 3. fejezet

Az Apache Spark keretrendszer

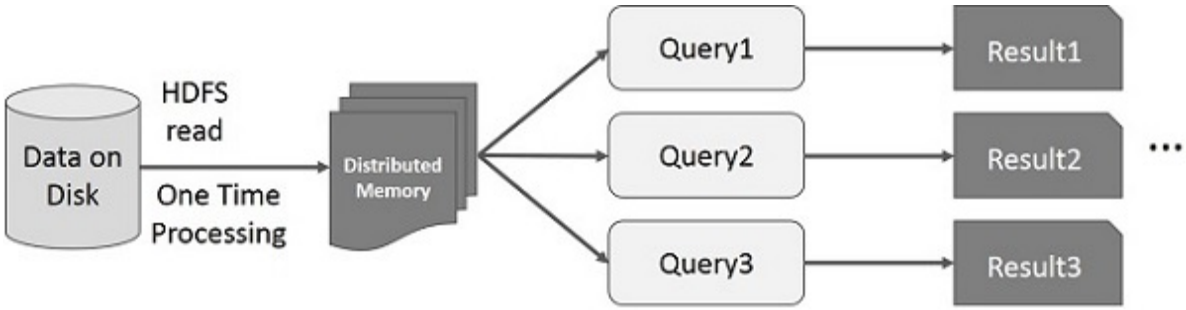
Az Apache Spark [3] egy villámgyors klaszter számítási keretrendszer, amit nagyon gyors adatfeldolgozásra terveztek. A Hadoop MapReduce modellen alapul (koncepcionálisan, nem kód szintjén), de olyan módon általánosítja és terjeszti ki azt, ami lehetővé teszi a hatékony felhasználását interaktív lekérdezések készítéséhez vagy stream feldolgozáshoz is. Az alapvető technika, ami ezt a fajta sebesség növekedést lehetővé teszi, a memóriában tárolt klaszter számítási modell. A Spark (ellentétben a gyakori tévhittel) nem a Hadoop módosított verziója, a Hadoop ökoszisztémát tárolásra (HDFS) használja, de ettől sem függ, mert saját klaszter menedzsmenttel is rendelkezik. A Spark-ot széleskörű használatra tervezték, támogatja többek közt a kötegelt feldolgozást, iteratív algoritmusokat, interaktív lekérdezéseket és a stream feldolgozást is, a fent vázolt MapReduce hiányosságokat az alábbi módon küszöböli ki.

Iteratív alkalmazások felgyorsítása a köztes eredmények memóriában való tárolása által.



> https://www.tutorialspoint.com/apache_spark/images/iterative_operations_on_spark_rdd.jpg

Interaktív alkalmazások felgyorsítása az adatok egyszeri elosztott memóriába történő beolvasásával.



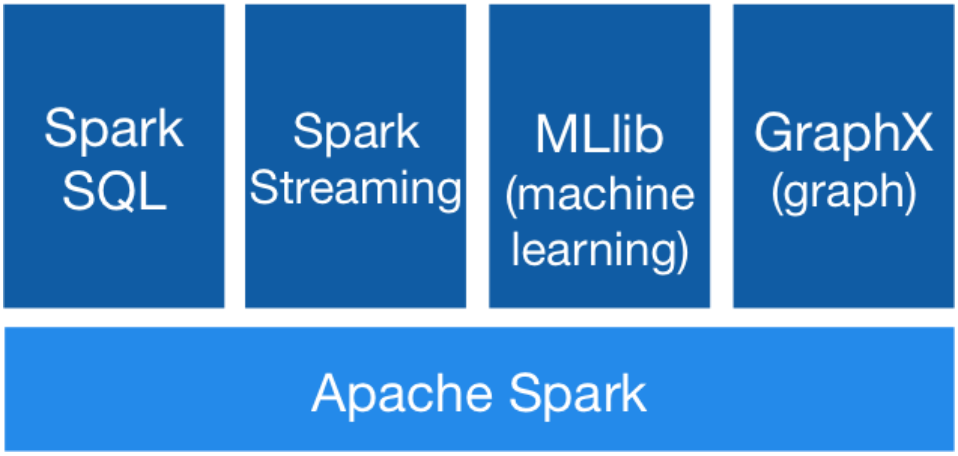
> https://www.tutorialspoint.com/apache_spark/images/interactive_operations_on_spark_rdd.jpg

A számos felhasználási mód támogatása mellett nagy előnye még az üzemeltetési költségek csökkentése azáltal, hogy a különböző feladatok megoldására szolgáló eszközöket mind tartalmazza, nem kell azokat külön-külön karbantartani. A Spark a Hadoop egy al-projektje, amit 2009-ben kezdett fejleszteni Matei Zaharia a UC Berkeley AMPLab-jában. 2010-ben BSD licensszel nyílt forrásúvá tették, és 2013-ban az Apache szoftver alapítványnak adományozták. Így a Spark 2014-től felső szintű Apache projektté vált.

Az Apache Spark főbb jellemzői

Az Apache Spark az alábbi főbb jellemzőkkel bír:

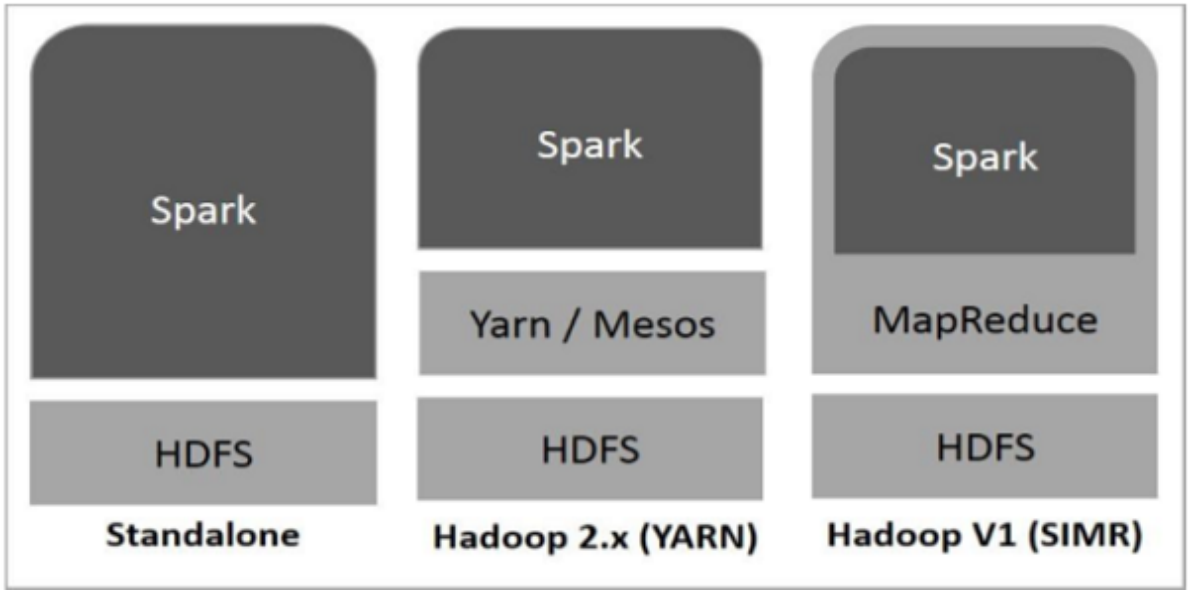
- **Sebesség** − a Spark segítségével kb. 100-szoros sebességgel futtathatunk alkalmazásokat Hadoop klaszter fölött, amennyiben az adatok memóriában tárolódnak, és kb. 10-szeres a sebesség növekedés, ha az adatok merevlemezen vannak. Ezt az I/O műveletek számának drasztikus csökkentésével és a köztes eredmények memóriában történő tárolásával éri el.
- **Többnyelvűség támogatása** − a Spark beépített API-val rendelkezik Java, Scala és Python nyelvekhez. Ennek köszönhetően a Spark alkalmazásokat akár különböző nyelveken is írhatjuk. A Spark 80 magas-szintű operátorral rendelkezik az interaktív lekérdezések támogatásához.
- **Fejlett adatelemzés** − a Spark nem csak a `map` és `reduce` műveleteket támogatja. Beépített SQL lekérdezési lehetőséggel, adat streameléssel, gépi tanulási modulokkal és gráf algoritmusokkal is rendelkezik (lásd lenti ábra).



> https://spark.apache.org/images/spark-stack.png

Spark működése Hadoop fölött

A Spark több módon is integrálható Hadoop-pal. Az alábbi ábra szemlélteti a három alapvető konfigurációt, amellyel a Spark és a Hadoop összekapcsolható.



> https://www.tutorialspoint.com/apache_spark/images/spark_built_on_hadoop.jpg

- **Standalone** − ebben az üzemmódban a Spark a HDFS fölött önállóan fut, a MapReduce keretrendszer mellett. Mind a SPark, mind pedig a MapReduce használható a klaszteren futó job-ok végrehajtásához.
- **Hadoop YARN** − ebben a konfigurációban a Spark nem közvetlenül a HDFS fölött fut, hanem a YARN-t használja az erőforrások eléréséhez. Ebben az esetben nincs szükség komponensek elő telepítésére vagy rendszergazdai jogosultságra. A YARN által sokkal egyszerűbben integrálható a Spark a Hadoop ökoszisztémába.
- **Spark in MapReduce (SIMR)** − korábbi Hadoop verzió esetén (YARN még nem támogatott) lehetőség van a Spark MapReduce engine-be ágyazására, így Spark job is futtatható MR motorral. A felhasználóknak lehetősége van a konzolos kliens használatára is adminisztrátori jogosultság nélkül.

Ellenőrző kérdések

1. Mik a klasszikus Hadoop MapReduce végrehajtó motor legfőbb hiányosságai (olyan jellemzői, ami bizonyos típusú alkalmazások számára lehetetlenné teszik a használatát)?
2. Mi jellemzi az iteratív és interaktív BigData alkalmazásokat? Milyen fontos szempontnak kell megfelelnie egy feladat végrehajtó motornak ahhoz, hogy megfelelően ki tudja szolgálni az ilyen alkalmazásokat?
3. Milyen általános, elosztott, BigData adatfeldolgozó feladat végrehajtó motor megvalósításokat ismersz?
4. Milyen tervezési koncepciónak köszönhető a Tez teljesítmény előnye a MapReduce-hoz képest?
5. Az alábbiak közül melyik Java objektum jelöl egy végrehajtási csomópontot egy Tez DAG job-ban: a) `Node` b) `Vertex` c) `Edge` d) `DAG`?
6. Mivel éri el az Apache Spark a 10-100x-os gyorsulást a MapReduce-hoz képest?
7. Milyen komponenseket foglal magában az Apache Spark? Mire szolgálnak ezek?
8. Milyen programnyelveken írhatunk programot Apache Spark fölé?
9. Hogyan oldja meg az Apache Spark az iteratív és interaktív alkalmazások hatékony kiszolgálását?
10. Milyen módokon integrálható az Apache Spark a Hadoop klaszterrel? Mik az egyes konfigurációk jellemzői?

Referenciák

[1] https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[2] https://tez.apache.org/

[3] https://spark.apache.org/

[4] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[5] https://github.com/ouyi/tez-demo/blob/master/src/main/java/io/github/ouyi/tez/HelloWorld.java

[6] https://www.xplenty.com/blog/apache-spark-vs-tez-comparison/