ABIS is a training & consulting company
located in Leuven (Belgium) & Woerden (The Netherlands);
main topics of interest include
databases, data analytics, mainframe, and programming languages.
Peter Vanroose is senior instructor & consultant at ABIS,
where he teaches (a.o.) Db2 and data analytics related courses.
This includes SQL for BI & Data Science, Spark, Hadoop,
and of course Db2 for z/OS: performance, DBA, and application design.

Agenda

- Understand how Db2 may integrate with Spark, R, and Hadoop
- Realize which high-volume building blocks of typical analytics algorithms are well suited to be delegated to Db2
- Learn some basic Spark syntax, sufficient to build useful data analysis tasks with Db2 data

Over the last few years, Spark has become the most popular open-source analytics engine for large-scale data processing. Its success is mainly due to its ease of use, its performance, and its flexible access to external data sources.

With "Big R", a BigInsights component, IBM already explored the possibilities of delegating data-intensive analytic workload from the open-source statistics platform "R" to Db2.

In this presentation, we explore similar possibilities with open-source Spark as a front-end.

By isolating some recurring data-intensive tasks in typical Spark workloads, and delegating those to Db2, we were able to considerably improve performance of some data analytics use cases.

Key to this is a well configured data warehouse in Db2, including e.g. indexes, MQTs, and stored procedures. This is only possible if Db2 architects have a good understanding of how Spark (and underlying Hadoop MapReduce) works.

Agenda – details:

- **I.** Context: The data warehouse
  - The “classic” data warehouse
  - The “NoSQL” data warehouse
- **II.** Spark
  - Background: cluster computing; Hadoop; MapReduce
  - Setup: ease of use; connectors (e.g. Db2); development environment
- **III.** Db2 with Spark
  - Cloud solutions; BigR ideas
  - Spark “data souce” integration
  - Use cases for data-intensive tasks

This presentation consists of three main parts:
– In part I, the “use case” context for Spark is sketched
– In part II, Spark is briefly discussed, together with some background necessary to understand its structure & design, viz. Hadoop & MapReduce
– Part III contains the main message of this presentation:
What can Spark do for a Db2 user,
and how can Db2 provide direct benefit to a Spark user?

Disclaimers

- This presentation is unavoidably **limited** in scope;
  Please feel free to *ask questions* during the presentation!
- All explanations are correct, but will be somewhat **simplified**
  (i.e.: might miss some detail and/or nuance)
  Again: please feel free to ask for more details during the presentation
- See the **notes** (in the PDF) for pointers to additional information

I. Context: The data warehouse

- The classic data warehouse:
  - **RDBMS** like Db2: aggregation, OLAP, indexes, MQTs, …
  - **Statistical** software (e.g. SPSS, R)
  - “**Machine Learning**”
  - pattern recognition, clustering, classification, regression, ...
- The NoSQL case:
  - Less guarantees, but more **volume** & **velocity**, less structure
  - Implementations: Hadoop, Spark, ...

Summary of part I

The classic data warehouse

- RDBMS:
  - On-Line Analycs (OLAP) => **aggregation** (SUM, COUNT, AVG) + grouping sets
  - answer **BI** questions, like:
    revenue: overview per year, month, region, product
    TOP-10 (best customers, most promising new markets, least profitable products)
    => needs "total sorting" (= n log n); indexing not allways possible …
  - typical setup: **data warehouses**: ETL; heavy pre-sorting & aggregation
- Statistical software (e.g. SPSS, **R**):
  - graphical possibilities (better than Excel): scatter plot, histogram, time series, …
  - statistical **modeling** (e.g. lin. regression) & trend analysis => **decision** support
- Machine learning (**ML**)
  - "classic" examples: spam filters, virus scanners, OCR, search engines

Part I – the "classic" RDBMS data warehouse:
– a typical ETL (extract / transform / load) database environment
– read-only for most of the time; re-loaded with production data from time to time
– use cases: analytics-related ("BI": business intelligence)

The NoSQL data warehouse

Enormous amounts of data, less structured, no time (or need) for **ETL**:

- The **3 V's**: volume, velocity, variety (& variability)
- "Big Data" => Hadoop (HDFS), Amazon S3, …
  - assumes a cluster of commodity hardware (**sharding** — scale out)
  - fail-safe because of **redundance**
- but ... less data consistency guarantees
  - because of the **CAP theorem** (Brewer, 2000):
    can only have 2 out of 3: ***c****onsistency,* ***a****vailablility,* ***p****artitioned*
  - BASE instead of ACID
- analytical frame work: **MapReduce** => "access path" framework

Part I – the "not so classic" data warehouse
– the so-called NoSQL context
– similar analytics use cases, but completely different setup
– perfectly suited for
  * larger volume data
  * ad-hoc querying

II. Spark

- became the most popular "distributed data" analytics engine
- Background:
  - cluster computing
  - Hadoop
  - MapReduce
  - "function to data"
- The Spark setup
  - ease of use; performance; connectors; development environment

Summary of part II

Spark: background

- cluster computing
- Hadoop
  - is the *first generation* analytics engine
  - built after Google's prototype framework
- MapReduce
  - Hadoop's "computational" building block
  - "distributed computing" framework
  - Consists of: Mapper, Shuffler, Reducer
- "Function-to-data", instead of data-to-function

Part II: summary of Spark background

Cluster computing

- Like a supercomputer
- But “shared-nothing” setup:

  Every node has
  - its own disk
  - its own RAM
  - its own processor(s)

  “commodity hardware”

  Jobs ideally run
  - in parallel
  - on partitions of the data

-

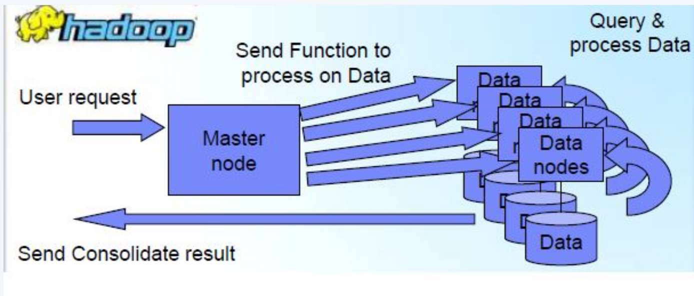
Hadoop

- An *Apache* project (`https://hadoop.apache.org/`)
- Implemented in Java => runs in JVM
- 3-in-1:
  - Storage: **HDFS**, the Hadoop Distributed File system
    Partitioned data resides on different cluster nodes; *replication* (3-fold)
    No support for *updates*! => only read, append, drop; auto-partitioning
  - Computation engine & framework: **MapReduce**
    Parallelized algorithm runs on all processing nodes (ideally: = data node)
  - Job scheduler / resource negotiator: **Yarn**
    Submit job steps on selected nodes: CPU / RAM / disk

-

Function-to-data

- I/O efficiency : avoid (unnecessary) data transfer

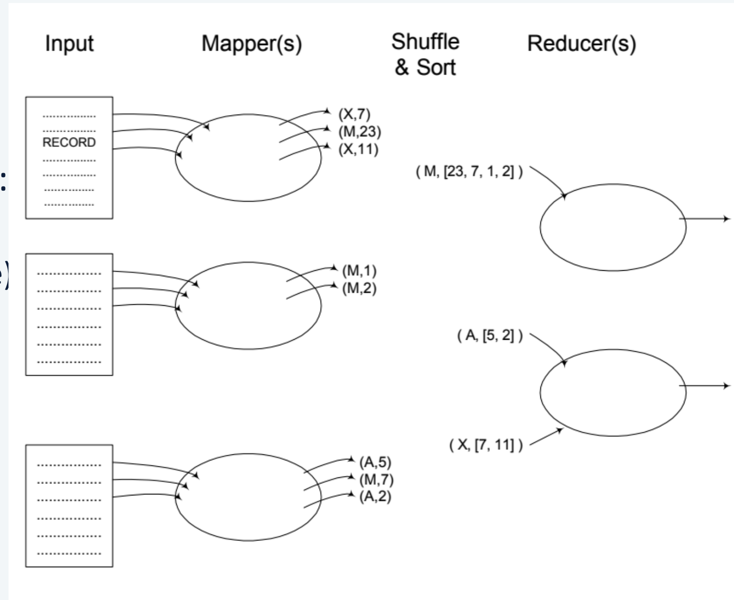


- Especially important with clustered, high-volume data !
- No **ACID** guarantees, though … (data copies; data modifications; ...)

-

MapReduce

- Hadoop's (and Spark's) **computational** framework:
  - user provides **mapper**:
    specify: record =>(key,value)
  - user provides **reducer**:
    (key, value-list) => record
  - framework provides **shuffler**:
    guarantee:
    equal keys from mappers
    go to same reducer



-

Hadoop + Hive

- Hive (`https://hive.apache.org/`): layer on top of Hadoop MapReduce
- A "de facto" SQL optimizer: translates SQL into MapReduce job(s)
- Example:

```
CREATE TABLE weblog (ip STRING, ..., webpage STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ' '
LOCATION  'hdfs://host/dir/weblogs.log' ;

SELECT webpage,COUNT(*)   FROM weblog   WHERE ip LIKE '10.%'
GROUP BY webpage   ORDER BY 2 DESC   LIMIT 30;
```

-

Spark: design principles

- Ease of use
- Performance
- Easy integration with other components
  - Data **connectors** (storage; input & output)
  - Cluster **scheduler**
  - Extension **libraries**
  - Interactive & programmatoric **user interfaces**
- Development environment
  - Single “node”; REPL; integrates with **R**, Python, Java, **Scala**

Part II: summary of Spark setup & design principles

Spark: an overview

- Kind of redesign of Hadoop + Hive

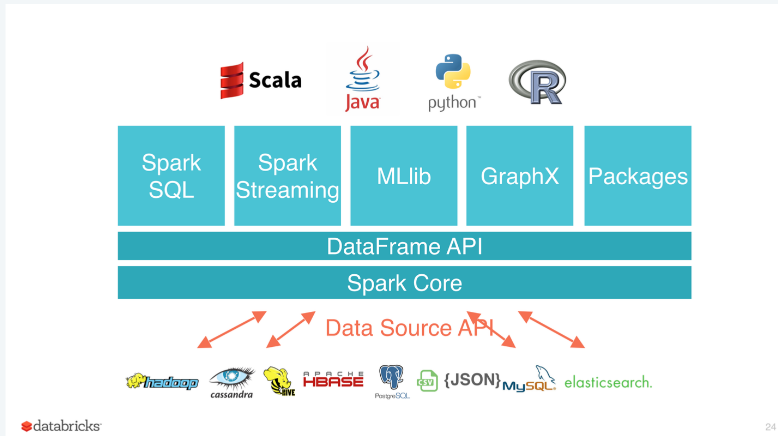
  & with ideas from R, (I)Python, Jupyter, Zeppelin, Mahout, Storm, Avro, …
- *combines the best elements of all its predecessors*
- top-down approach:
  - good, simple user interface, prevents making “stupid mistakes”:
    - **fast prototyping**: command interface (interactive) => **deploys** easily
    - provide for a data flow pipeline via **immutable** objects & methods
  - simple integration with existing frameworks
- better than its predecessors: e.g. in-memory where possible

-

Spark: data connectors

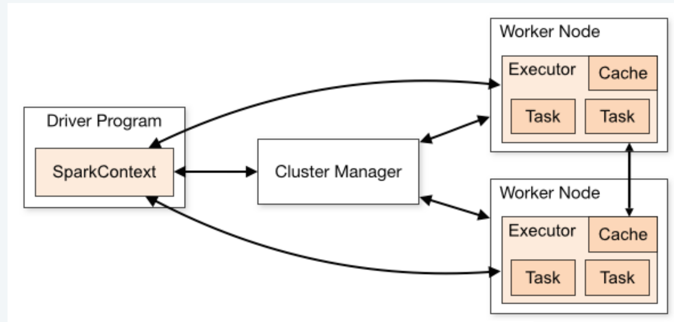
Flexible interface to data sources (& data sinks):

- Through **URI** notation:
  - file:///dir/filename
  - hdfs://dirname/filename
  - s3a://bucket/filename
  - s3n://bucket/filename
  - jdbc:db2://host/ssid
- Through context methods:
  - val x = sc.cassandraTable(...)
  - val y = read.format("jdbc"). ...
  - y.write.format("csv"). ...



-

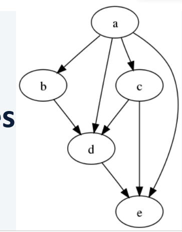
Spark: cluster manager (job scheduler)



Flexible setup:

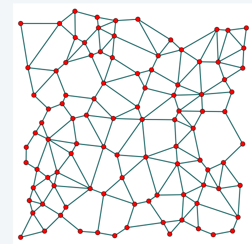
- Spark can work with any cluster manager:
  – Yarn (<== Hadoop)
  – Mesos
  – Spark-provided
- “local mode” (no cluster)

- Cluster manager is a **resource negotiator**: RAM / (disk) / CPU
- *SparkContext* is **programmer’s abstraction** of resources:
  – variables => RDDs (resilient distributed data) & **DataFrames**
  – dependencies: “directed acyclic graph” (DAG):

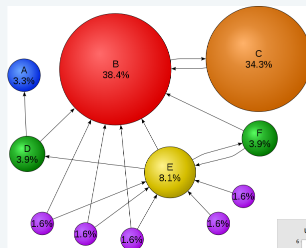


-

Spark: extension libraries

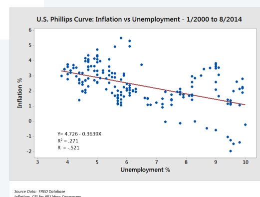


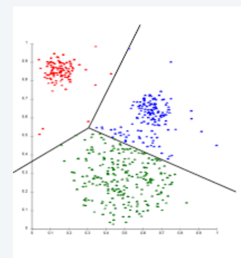
- GraphX: “interconnected” data representation
  - Parallelized implementations of well-known algorithms
  - e.g. Page Rank algo (Google)



- Streaming

- MLlib: machine learning
  - Parallelized implementations of well-known algorithms e.g. regression, classification, clustering, ...





-

Spark: development environment

- REPL:
  Read – evaluate – print loop
  - Kind of “shell” environment
  - Similar to R, to IPython
- Straightforward deployment
  - No need to change a single command
  - Interactive “script” becomes production “batch program”
- Integrates with:
  - R, Python, Java (no REPL), **Scala**

-

Using Spark

- Installation:
  - Download & install latest version from spark.apache.org on a Linux system
    or: download a preconfigured virtual image, e.g. CDH from www.cloudera.com
  - or: use a preconfigured cloud solution:
    AWS (**Amazon** Web Services) EMR (Elastic MapReduce), EC2
    Google Cloud Platform(https://cloud.google.com/hadoop/)
    IBM Cloud: https://www.ibm.com/cloud/spark (Watson, BigInsights)
- Logon to the (stand-alone) server (Linux command line)
- Start the REPL shell

-

Spark: an example (1|3)

```
[Linux]$ spark-shell --jars $DB2HOME/sqllib/java/db2jcc4.jar
Spark context Web UI available at http://spark.abis.be:4040
Spark context available as 'sc' (master = local[*], app id = local-123456).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.3
      /_/
Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 1.8.0_191)
Type in expressions to have them evaluated.
scala>
```

-

Spark: an example (2|3)

```
scala> 1 + 2
res1: Int = 3
scala> 333 - 1000.0 / 3
res2: Double = -0.3333333333333144
scala> sc
res3: org.apache.spark.SparkContext = org.apache.spark.SparkContext@abcdef01
scala> val textFile = sc.textFile("file:/home/peter/mytext.txt")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1]
scala> val words = textFile.flatMap( line => "\\w+".r.findAllIn(line) )
scala> val words_as_key_val = words.map( word => (word, 1) )
scala> val words_with_counts = words_as_key_val.reduceByKey( (v1,v2)=>v1+v2 )
       # all unique words from the text, with their occurrence count
scala> words_with_counts.saveAsTextFile("file:/home/peter/count.out")
scala> :quit
[Linux]$
```

-

Spark: an example (3|3)

```
scala> val mytbl = spark.read.format("csv").option("delimiter","\t").
                        option("inferSchema","true").option("header","true").
                        load("hdfs://localhost/user/peter/mytbl.csv")
mytbl: org.apache.spark.sql.DataFrame = [nbr: int, name: string, tel: string]
scala> mytbl.registerTempTable("persons")
scala> val cnt = spark.sql("SELECT count(*) AS total FROM persons")
cnt: org.apache.spark.sql.DataFrame = [total: bigint]
scala> val total = cnt.head(1)(0)
total: org.apache.spark.sql.Row = [4162051]
scala> val telnrs = mytbl.where("tel IS NOT NULL").select("name","tel")
telnrs: org.apache.spark.sql.DataFrame = [name: string, tel: string]
scala> telnrs.write.format("csv").option("header","true").save("hdfs:telnr")
scala> :sh hadoop fs -ls -R
drwxr-xr-x   - peter supergroup      0 2019-10-24 09:37 telnr
-rw-r--r--   3 peter supergroup 643731 2019-10-24 09:37 telnr/part-00000-abcdef.csv
```

-

Spark essentials in a nutshell

1st example: **RDD** = resilient distributed dataset; *immutable* object

- Is a "*virtual object*"; lazy evaluation: dependency graph (DAG) is stored
- Is a (long) list of "similar records" (very flexible)
- "**transformation**": in-cluster (virtual) conversion from RDD to RDD
  e.g. map(f), flatMap(f), filter(f), reduceByKey(f), sortBy(f)
- "**action**": cluster-to-client (real) conversion
  e.g. collect(), take(10), max()

2nd example: **DataFrame** = "table-like" cluster object

- Similar to RDD
- Can only store rows with *identical* column structure & data types
- 2nd generation: more efficient than RDD; new (different) API; supports **SQL**

-

III. Db2 with Spark

- The Spark JDBC data connector
  - How to use it with Db2
  - Possibilities & limitations
- “Nice to have” mutual benefits (cf. idea’s in IBM’s BigR)
  - Avoid huge amounts of data traffic
  - Use Db2’s strengths: indexes & optimal access paths
- Other connection possibilities
- Use cases for Db2 / Spark cooperation
  - Spark functionality (e.g. Machine Learning algos) on Db2 data
  - Db2 data (e.g. pre-aggregated data warehouse columns) for BI use
  - Spark & Db2 “in the cloud” => IBM/AWS/... cloud solutions

Summary of part III

IDUG Db2 Tech Conference
Rotterdam, Netherlands | October 20-24, 2019

The Spark JDBC connector with Db2

- “Read from JDBC” returns a DataFrame:

```
val url = "jdbc:db2://hostname:50000/ssid?user=peter&password=pwd"
val act = spark.read.format("jdbc").option("url",url).option("driver",
          "com.ibm.db2.jcc.DB2Driver").option("dbtable","db2.act").load()
act: org.apache.spark.sql.DataFrame = [actno: smallint, actkwd: string, actdesc: decimal(9,2)]
```

- No query is run yet: DataFrame is a “just-in-time” object! (=> DAG)
- Saving a DataFrame as a Db2 table:

```
telnrs.write.format("jdbc").option("url",url).option("dbtable","peter.tel").
  option("createTableColumnTypes","name VARCHAR(64),tel VARCHAR(32)").save()
```

- Interrogating a Db2 table:

```
val nrcourses = spark.read.format("jdbc").option("url",url).
      option("query","SELECT count(*) FROM db2.act").load().head(1)(0)
```

For an example from IBM (for their cloud usage of Spark), see e.g.
https://cloud.ibm.com/docs/services/AnalyticsEngine?topic=AnalyticsEngine-working-with-sql

For an example of more specifically the Spark Db2 connector: see e.g.
https://cloud.ibm.com/docs/services/AnalyticsEngine?topic=AnalyticsEngine-spark-connectors

Those examples use the Python language interface instead of Scala which is used in this slide; the example in this slide would read as follows with Python:

```
url = "jdbc:db2://hostname:50000/ssid?user=peter&password=pwd"
courses = spark.read.format("jdbc").option("url",url).option("dbtable","tu.courses").load()
telnrs.write.format("jdbc").option("url",url).option("dbtable","peter.tel")
  .option("createTableColumnTypes","name VARCHAR(64),tel VARCHAR(32)").save()
nrcourses = spark.read.jdbc(url).option(query,"
    SELECT count(*) FROM tu.courses").load().head(1)[0]
```

The Spark JDBC connector – possibilities & limitations

- "Read from JDBC" always returns a DataFrame
  - is a distributed virtual object, **but** no way to make the Db2 connection "parallel"
  - full dataframe has to "materialize" in the Spark cluster
- User should delegate any filtering / mapping / aggregation to Db2
  - i.e.: write those actions inside the SQL statement, **not** in DataFrame terms
  - no automatic delegation (yet)
- Db2 data presents itself as DataFrame
  - flexible & straightforward to combine it with DF from other sources (e.g. join)
  - careful with (automatic) datatype conversions!

    More specifically: DECIMAL(n,p) <==> double; VARCHAR length; TIMESTAMP

-

The Spark JDBC connector – an example

- Don’t write

```
val courses = spark.read.format("jdbc").option("url",url).
                  option("dbtable","tu.courses").load()
val c=courses.filter(col("ctitle").like("%Db2%")).select("cdate","cprice")
val summary = c.map((x,y)=>(year(x),y)).groupBy("cyear").sum("cprice")
```

- But instead write

```
val summary = spark.read.format("jdbc").option("url",url).option("query","
         SELECT year(cdate), SUM(cprice)
         FROM   tu.courses
         WHERE  ctitle LIKE '%Db2%'
         GROUP BY year(cdate)
     ").collect()
```

-

Db2 & Spark – "Nice to have" mutual benefits

- cf. ideas found in IBM's BigR
- Avoid unnecessary data transfers:
  - delegate table filtering to Db2 (*indexable*!)
  - delegate aggregation to Db2? (sum/count/min/max/avg)
  - efficient table joining (*indexable*!)
- **Challenges**:
  - join Db2 table with Spark DataFrame
  - access path selection (optimizer) – is whose responsibility?
  - map Db2 partitioning to Spark partitioning (parallelism)

-

Db2 & Spark – ideas from BigR

IBM's BigR: integrates **R** within IBM InfoSphere BigInsights

- Using a standard R interface, user can access BigInsights cluster data
- Data is presented as an (R) **DataFrame** => standard R functions can be used
- R functions are ***pushed down*** to the data ("function-to-data")
- R syntax (esp. Operator overloading) easily allows this implementation

Could these ideas be mapped onto

- Scala (instead of R)
- Db2 (instead of BigInsights MapReduce cluster)

?

See e.g.
https://www.ibm.com/support/knowledgecenter/en/SSPT3X_3.0.0/
com.ibm.swg.im.infosphere.biginsights.bigr.doc/doc/intro.html

Db2 & Spark – how to optimize data transfers?

- Set up parallel threads from Spark “worker nodes” to Db2
- Exchange meta-data knowledge
  - esp. Db2 partitioning
  - maybe also catalog statistics
  - or just filter factor estimates?
- Push down data reduction transformations to Db2:
  - e.g. WHERE (indexable)
  - JOIN (indexable & filtering; star join) → even with non-Db2 data!
  - GROUP BY => SUM, COUNT, MIN, MAX

-

Db2 & Spark – the importance of Db2 range partitioning

- Parallelism (for Spark–Db2 communication) can only work
  - when Db2 data is **partitioned**
  - when this partitioning can easily be described => need **range** partitioning (**PBR**)
    => rule-of-thumb: at most 128 MB per partition
    => Db2 LUW: use **pureScale**
  - when the Db2 partitions are easily accessed **independently** & in parallel
    => I/O parallelism (different **disks** / volumes)
    => CPU parallelism (multi-processor; Db2 z/OS: use **data sharing**)
- Need for transparent partition-level interface in Spark
    => to be implemented ...

-

Db2 & Spark – the importance of design patterns

- Transparent & efficient implementation (i.e.: translation to Db2) of Mappers & Reducers (esp. SUM / AVG / MIN MAX / FETCH FIRST) can only work well
  - when **design patterns** for parallelism are well understood by implementors
  - these have been studied intensively by implementors of e.g. *Hive*
  - a very readable & practically useful book on the subject:
    "MapReduce Design Patterns", Building Effective Algorithms and Analytics for Hadoop, Donald Miner & Adam Shook, O'Reilly, December 2012, ISBN 978-1-449-32717-0
- the most important MapReduce design patterns are summarized on next slides

-

Db2 & Spark – MapReduce design patterns (1|2)

- Filtering patterns:
  - *Simple filters* (based on content) => leave to Db2 (indexing!); careful: **sargeable**
  - *Top-N filters* => **avoid total sort**: delegate to partitions, then recombine!
  - *Bloom filters* => for sparse data; not supported directly by Db2 …
  - *Distinct filters* (removing duplicates) => **avoid total sort**; e.g. only through index
- Summarization patterns:
  - *counting* & *summing* (= adding numerical values) => easily parallelizable
  - *min* & *max* => often efficiently indexable
  - *avg*, *std dev*, *correlation* => less evident! (only indirectly parallelizable)
  - *median*, *quantiles*, … => require **total sort** ...

-

Db2 & Spark – MapReduce design patterns (2|2)

- Summarization patterns (cont.):
  - *inverted index* (cf. keyword index at end of book)

    => a "GROUP BY" without loss of detail !

    => Db2 has **LISTAGG** function (z/OS: as of version 12 FL 501)
- Total order sorting
- Joining: Db2 table(s) with non-Db2 data
  - *replicated join* (when all but 1 of the sets fits in memory) => cf. Db2 star join

    => *make sure the in-memory lookup tables have their FKs as (hash) keys*
  - *reduce-side join* (generic) => use the Db2 idea of a **hybrid join** ("list prefetch")
  - most inner/outer join variants are easily covered

-

Db2 / Spark cooperation – use cases

Apply Machine Learning algorithms to Db2 data:

- *Example 1*: train a linear regression model from some training data:

```
import org.apache.spark.ml._
val lr = new regression.LinearRegression
val trainingData = spark.read.format("jdbc").option("url",url).option("query","
         SELECT avg(time) AS deliv,avg(satisfac) AS label FROM orders GROUP BY cust_id")
val assembler = new feature.VectorAssembler().setInputCols(Array("deliv")).setOutputCol("features")
val lrModel = lr.fit(assembler.transform(trainingData))
println(s"Coefficient: ${lrModel.coefficients(0)} Intercept: ${lrModel.intercept}")
println(s"RMSE: ${lrModel.summary.rootMeanSquaredError}")
// Next apply the model to new data, as to predict satisfaction from delivery time:
val predicted_satisfaction = lrModel.predict(linalg.Vectors.dense(2.5))
```

-

Db2 / Spark cooperation – use cases

- *Example 2*: cluster my customers into 3 "typical groups"

```
import org.apache.spark.ml._
val kmeans = new clustering.KMeans;  kmeans.setK(3).setSeed(1L)
val trainingData = spark.read.format("jdbc").option("url",url).option("query","
         SELECT cust_id,avg(price) AS price,count(*) AS cnt FROM orders GROUP BY cust_id")
val a = new feature.VectorAssembler().setInputCols(Array("price","cnt")).setOutputCol("features")
val t = a.transform(trainingData);     val model = kmeans.fit(t)
println(s"Cluster centers: ${model.clusterCenters} Sizes: ${model.summary.clusterSizes}")
println(s"within-set sum of squared errors: ${model.computeCost(t)}")
```

// Show all "similar" customers, e.g. those that fall into cluster nr. 2 :

```
model.summary.predictions.filter("prediction=2").select("cust_id").collect
```

-


Please fill out your session evaluation before leaving!
Peter Vanroose
ABIS Training & Consulting
pvanroose@abis.be
Spark your Db2 data warehouse
Session code: F16
IDUG
Leading the Db2 User Community since 1988
abis
TRAINING & CONSULTING

-