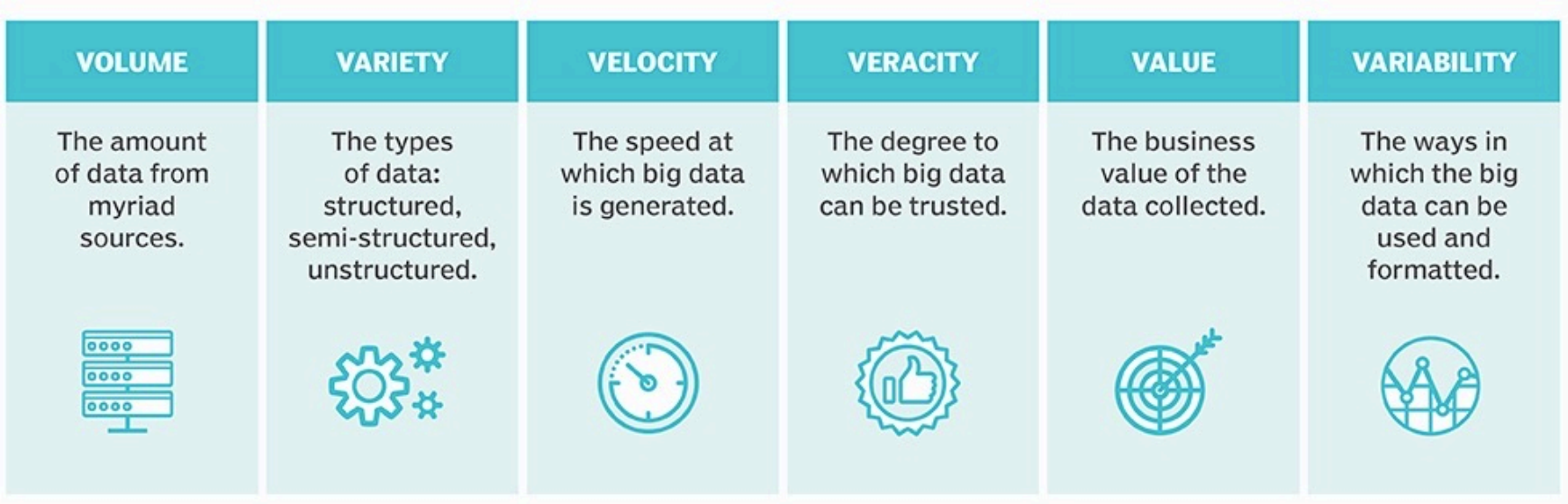
Building effective data pipelines

Master DAC – BDLE
Mohamed-Amine Baazizi
mohamed-amine.baazizi@lip6.fr
2024

Context and goals



Main challenge: how to build an effective data pipeline transforming raw data into insights?

Characteristics of (big) data

- Hosted in different storage systems
  - files, online access
- Represented in various formats
  - JSON, CSV, XML, raw text, images
- Schema-less and heterogenous
- Subject to change over time
  - updates, deletions
- Produced and consumed continuously
  - stream processing and analytics

Data Lakes

- A centralized repository of (un)-structured data at any scale
- Various types of analytics
  - dashboards and visualizations
  - big data processing, real-time analytics
  - machine learning
- Data-format agnostic:
  - store raw data
- Future-proof
  - even if no specific existing need, data should be available

Data lake vs Data warehouse

characteristics	Data warehouse	Data lake
data	structured	both structured and unstructured
schema	schema-on-write (designed a priori)	schema-on-read (time of analysis)
processing	SQL + UDF	more flexible using a programming language
quality	highly curated	raw data
analysis	BI reporting, visualizations	ML, mining, streaming, …

Challenges of Data Lakes

- Data Ingestion
  - connect with external sources
  - Ensure high throughput, low latency
  - data keeping only, indexing, versioning, basic data sketches
- Data Extraction
  - transforming raw data to a pre-determined data model
    - e.g. table extraction from web pages (scrapping)
- Data Cleaning
  - enforce quality constraints and fix errors

Nargesian et al. Data lake management: challenges and opportunities. PVLDB, vol 12

Challenges (cont'd)

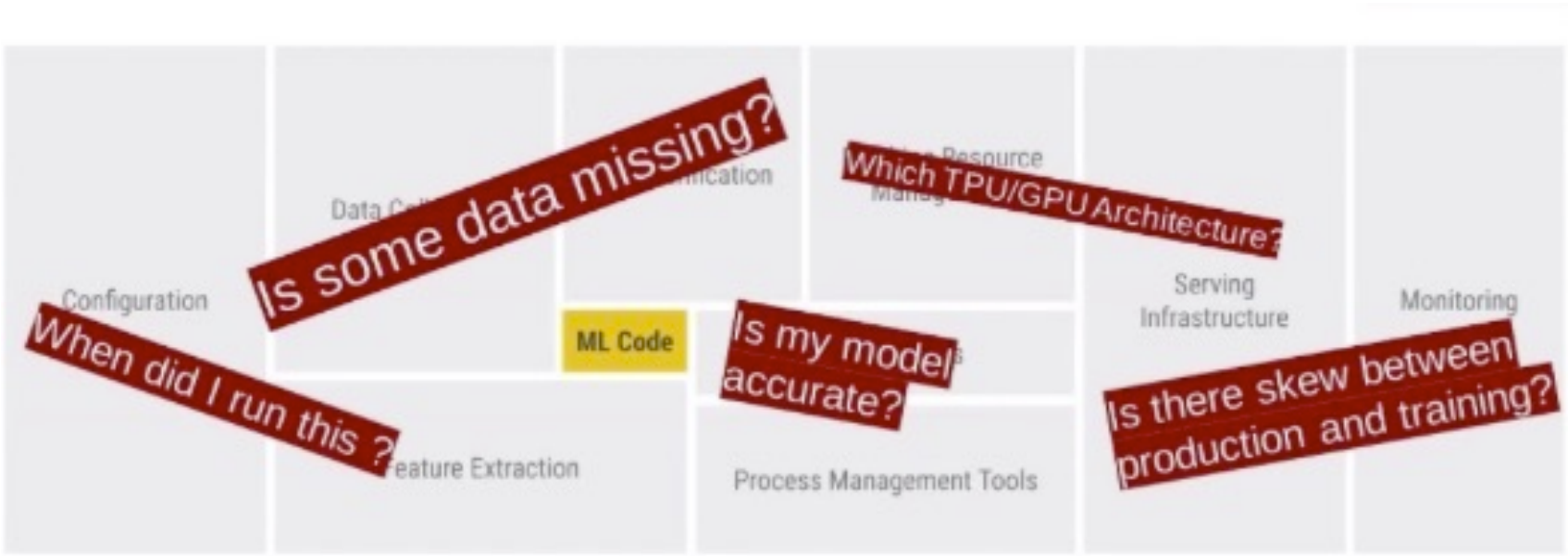
- Dataset Discovery
  - schema-driven or query-driven
  - search for joinable tables
- Metadata Management
  - generate and maintain catalogs
- Data Integration
  - find relevant data, schema-mapping
- Dataset Versioning
  - deal with dynamicity, many users
  - solutions: DataHub - issues: cost effectiveness

Nargesian et al. Data lake management: challenges and opportunities. PVLDB, vol 12

Data quality is an important issue

- Data integrated from different sources
  - different schemas/representations/formats
- ... produced w/o schema nor quality guarantees
  - Missing values, data drift, outliers, different units
- Impact
  - Business intelligence: wrong decisions
  - Machine learning: model with poor performance
  - Pipelines: glitches due to unexpected situations

Many other considerations



Need for effective DataOps and MLOps

Sculley et al. Hidden. Technical Debt in Machine Learning Systems. NIPS 2015

Outline

- So far: data preparation and analytics in SQL
- Next steps
  - Managing semi-structured data
    - not all data is CSV, other formats: XML, JSON
    - advantage: nested data, schema-less, variable structure
  - Datalakes management: the Delta approach
    - Bringing ACID to data lakes
    - schema management, constraints enforcement, data updates and schema evolution
  - Data quality verification
  - Use case: end-to-end data pipeline
    - from raw data to analytics/ML model

Managing semi-structured data

Context

- NoSQL trend, represent denormalized data
  - Web crawled data, LLMs output, APIs, open data, …
- May formats
  - Standard: XML, JSON and associated dialects
  - proprietary: Protobuff, Avro
- JSON is the most popular
  - Serialization format for Java Script objects
  - Human-readable, self-describing
  - Simple, yet expressive
  - Supported by several DBMS and prog. languages

JSON syntax

```
{
    "event_id": 1024,
    "event_type": "post",
    "author": {
        "id": 123,
        "platform": 2
    },
    "tags": ["...", "..."]
},
{
    "event_id": 4096,
    "event_type": "share",
    "author": {
        "id": 456,
        "platform": 3,
        "shared_post_author_id": 123
    },
    "shared_post_id": 1024
}
```

Complex values

Records { “prop”: val, …}
Arrays [val, …]

Base values

”text”, 123, True/False, null

Characteristics of JSON data

- no schema required
  - self-describing
- varying structure
  - freedom to mix different structures
- arbitrary nesting level
  - up to 7-8 in real-world col.
- denormalized data
  - many reptition

```
{
    "event_id": 1024,
    "event_type": "post",
    "author": {
        "id": 123,
        "platform": 2
    },
    "tags": ["...", "..."]
},
{
    "event_id": 4096,
    "event_type": "share",
    "author": {
        "id": 456,
        "platform": 3,
        "shared_post_author_id": 123
    },
    "shared_post_id": 1024
}
```

Managing JSON data

- Storage
  - Native using binary objects (e.g. Mongodb)
  - Enabled systems: using nested tables
- Querying
  - no standard query language
  - dedicated languages (e.g. Mongodb)
  - SQL, using built-in functions
- Schema management
  - possibility to validate against a schema
  - ensures robust execution of data pipelines
  - A standard language, JSON Schema, widely adopted

Native JSON systems

- Two major players: Mongodb and Couchbase
  - binary encoding (e.g. BSON for Mongo)
  - support query and update, proprietary language
  - schema validation (Mongodb only)
  - schema inference capabilities: native in Couchbase, via external tools in Mongodb
  - built-in distributed query processing, and physical optimization layer (indexing)

JSON-enabled systems

- Major DBMS (Oracle, Postgres, …)
- and Big Data platforms (Spark, Flink…)
- Data loaded into nested relations or kept as Binary Objects
- Queries using SQL (+ built-in functions)
- schema automatically inferred (Spark) or manually set (DBMS)
- *next, we adopt Spark, Dataframe* algebra

Spark Dataframe model recall

- Primivite types
  - boolean, numeric family (integer, decimal, …), String, null, timestamp
- Complex types
  - Arrays : (type of element, containsNull) → *homogenous*
  - Structure :
    - [StructField, …, StructField]
      - StructField(name, type, nullable) → name is a string and is unique
  - Maps:
    - (keyType, valueType, valueContainsNull) → key of any type and is unique
- User-defined types (UDT)
  - Application-specific data
  - Object-oriented databases style

containsNull, nullable and valueContainsNull indicate the presence of null, but are always set to true

Tabular data

```
movies = spark.read.format("csv"). …
```

```
movies.show(truncate=False)
```

```
+-------+---------------------------------+-----------------------------------------+
|movieId|title                            |genres                                   |
+-------+---------------------------------+-----------------------------------------+
|1      |Toy Story (1995)                 |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                   |Adventure|Children|Fantasy               |
|3      |Grumpier Old Men (1995)          |Comedy|Romance                           |
|4      |Waiting to Exhale (1995)         |Comedy|Drama|Romance                     |
|5      |Father of the Bride Part II (1995)|Comedy                                  |
|6      |Heat (1995)                      |Action|Crime|Thriller                    |
|7      |Sabrina (1995)                   |Comedy|Romance                           |
+-------+---------------------------------+-----------------------------------------+
```

```
movies.printSchema()
```

```
root
 |-- movieId: long (nullable = true)
 |-- title: string (nullable = true)
 |-- genres: string (nullable = true)
```

Spark Dataframe operators

- *SQL written differently, inspired by Python Dataframe*

- Relational operators
  - select, where, join
  - intersect, subtract, union
- Sorting
- Aggregation
- Grouping and aggregation
  - groupBy(col*)
    - groups on a set of columns
    - Produces a *GroupedDataset*

- Column operators
  - drop: removing
  - withColumn: definition
  - withColumnRenamed: renaming

single or multiple aggregations

- min(col), max(col), sum(col)
- agg(min(col), max(col), …)

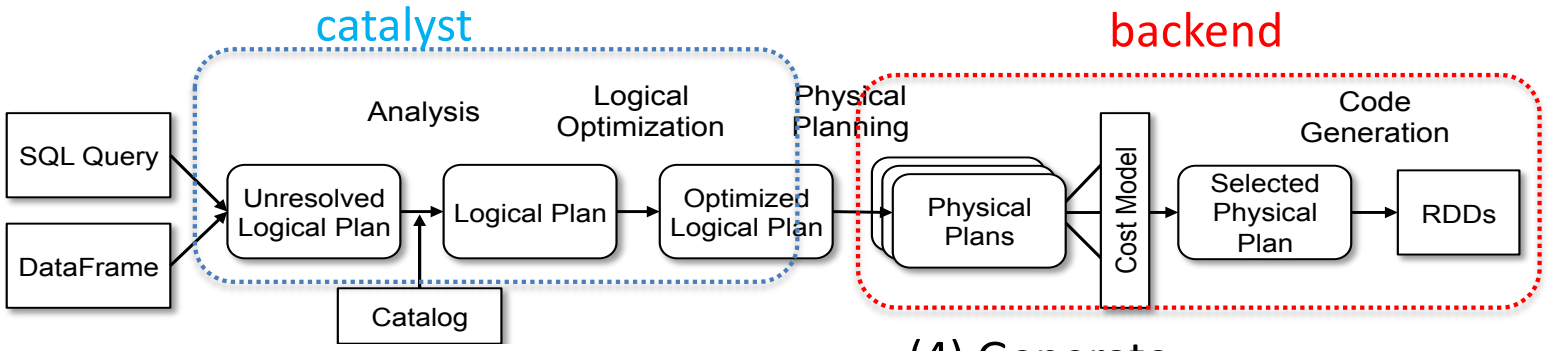
Pivoting

- column values -> column names

Query evaluation

(1) Check attribute types, …

(3) Choose optimal plan



(2) Transform query

(4) Generate optimized code

Rewrite query into logical plan

Catalyst rule-based optimizer, extensible

Standard optimization rules

- Sub-query elimination
- Column Pruning, Projection Collapsing
- Predicate Push Down

Build optimal physical plan

Cost-based adaptive optimization

Statistics about the data

Partitioning of the data

Illustration:
Building and flattening lists

```
+-------+---------------------------------+-----------------------------------------+
|movieId|title                            |genres                                   |
+-------+---------------------------------+-----------------------------------------+
|1      |Toy Story (1995)                 |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                   |Adventure|Children|Fantasy                 |
|3      |Grumpier Old Men (1995)          |Comedy|Romance                           |
|4      |Waiting to Exhale (1995)         |Comedy|Drama|Romance                     |
|5      |Father of the Bride Part II (1995)|Comedy                                  |
|6      |Heat (1995)                      |Action|Crime|Thriller                    |
|7      |Sabrina (1995)                   |Comedy|Romance                           |
+-------+---------------------------------+-----------------------------------------+
```

```
root
 |-- movieId: long
 |-- title: string
 |-- genres: string
```

```
from pyspark.sql.functions import col, split, explode
genres_list = movies.select(split(movies.genres,'\|').alias('genre_list'))
genres = genres_list.select(explode(col('genre_list')).alias('genre')).distinct()
genres.count()
```

Size → 7

10

```
+---------+
|    genre|
+---------+
|Adventure|
|   Comedy|
|  Romance|
| Children|
|  Fantasy|
```

Illustration:
Pivoting rows to columns

```
+-------+---------------------------------+-----------------------------------------+
|movieId|title                            |genres                                   |
+-------+---------------------------------+-----------------------------------------+
|1      |Toy Story (1995)                 |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                   |Adventure|Children|Fantasy               |
|3      |Grumpier Old Men (1995)          |Comedy|Romance                           |
|4      |Waiting to Exhale (1995)         |Comedy|Drama|Romance                     |
|5      |Father of the Bride Part II (1995)|Comedy                                  |
|6      |Heat (1995)                      |Action|Crime|Thriller                    |
|7      |Sabrina (1995)                   |Comedy|Romance                           |
+-------+---------------------------------+-----------------------------------------+
```

```
root
 |-- movieId: long
 |-- title: string
 |-- genres: string
```

```
movies_genre = movies.select('movieId',explode(split(movies.genres,'\|')).alias('genre'))\
.groupBy('movieId').pivot('genre').count().orderBy('movieId')
```

```
+-------+------+---------+---------+--------+------+-----+-----+-------+-------+--------+
|movieId|Action|Adventure|Animation|Children|Comedy|Crime|Drama|Fantasy|Romance|Thriller|
+-------+------+---------+---------+--------+------+-----+-----+-------+-------+--------+
|1      |null  |1        |1        |1       |1     |null |null |1      |null   |null    |
|2      |null  |1        |null     |1       |null  |null |null |1      |null   |null    |
|3      |null  |null     |null     |null    |1     |null |null |null   |1      |null    |
```

…

Built-in functions

- Array/maps manipulation
  - Arrays: containment, distinct, intersect, except, max/min, search, size, sort,..
  - Maps: concatenation, filtering, size,…
- General-purpose
  - Math: descriptive stats, trigonometry, skewness
  - Summary: *countDistinct, sumDistinct, …*
  - Elements to Collections and vice-versa
    - Nesting: *collect_list*, *collect_set*
    - Flattening: *explode, explode_outer*
  - Temporal:
    - current date, date arithmetic, day/month/year extraction, conversion
  - Data transformation
    - Json/csv parsing
    - Timestamp extraction/encoding
  - Strings: length, distance, trim, regexp extraction , splitting, case conversion, …
- Fully-documented online

Illustration:
applying built-in functions

```
+------+-------+------+----------+
|userId|movieId|rating| timestamp|
+------+-------+------+----------+
|     1|      1|   2.5|1260759144|
|     1|      2|   3.0|1260759179|
|     2|      1|   3.0|1260759182|
|     4|      3|   2.0|1260759185|
```

```
userId: long (nullable = true)
movieId: long (nullable = true)
rating: double (nullable = true)
timestamp: long (nullable = true)
```

```
from pyspark.sql.functions import from_unixtime, year, month, dayofmonth

ratings_date = ratings.select(from_unixtime('timestamp').alias('datetime'))\
.select('datetime', dayofweek('datetime').alias('day'))
```

```
+-------------------+---+
|           datetime|day|
+-------------------+---+
|2009-12-14 02:52:24|  2|
|2009-12-14 02:52:59|  2|
|2009-12-14 02:53:02|  2|
|2009-12-14 02:53:05|  2|
```

User-defined Functions

- Express *non-relational* operations
  - Ex. map rating to categories, compute user similarity
- Supported in most database systems
- Executed out of the SQL context
  - No automatic optimisation but
  - Possibility of adding a new optimisation rules (for experts)
  - If not carefully designed, performance degradation
- Only use when no available built-in function can do the work
- Two possibilities in Spark
  - Standard: invoke on each row → costly
  - Vectorized: invoke on a vector (batch of rows) → more efficient

  Benefit from memory columnar storage (Arrow format)

Illustration: applying a standard UDF

```
+-------+---------------------------------+-----------------------------------------+
|movieId|title                            |genres                                   |
+-------+---------------------------------+-----------------------------------------+
|1      |Toy Story (1995)                 |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                   |Adventure|Children|Fantasy               |
|3      |Grumpier Old Men (1995)          |Comedy|Romance                           |
|4      |Waiting to Exhale (1995)         |Comedy|Drama|Romance                     |
|5      |Father of the Bride Part II (1995)|Comedy                                  |
|6      |Heat (1995)                      |Action|Crime|Thriller                    |
|7      |Sabrina (1995)                   |Comedy|Romance                           |
+-------+---------------------------------+-----------------------------------------+
```

```
root
 |-- movieId: long
 |-- title: string
 |-- genres: string
```

```
from pyspark.sql.functions import udf

@udf('string')
def nb_genres(l):
return str(len(l))+' genres(s)'

select('movieId', 'title', nb_genres('genre_list').alias('nb_genres'))
```

```
+-------+---------------------------------+-----------+
|movieId|title                            |nb_genres  |
+-------+---------------------------------+-----------+
|1      |Toy Story (1995)                 |5 genres(s)|
|2      |Jumanji (1995)                   |3 genres(s)|
|3      |Grumpier Old Men (1995)          |2 genres(s)|
```

JSON management in Spark

- Data ingestion
  - schema inferred from a sample of the data
  - data mapped to *Dataframe*, guided by the schema
  - Shredding rules
    - JSON Object → Dataframe Struct
    - JSON Arrays → Dataframe Array
    - JSON Base types → Dataframe primitive types
    - Missing fields indicated with null
    - Conflict resolution using type promotion rules

Mapping JSON to Dataframes

Input: 2 documents

```
{
  "person" : {
    "firstname" : "Melena",
    "lastname" : "RYZIK",
    "role" : "reported",
    "rank" : 1,
    "organization" : ”abc"
  }
}
```

```
{
  "person" : {
    "firstname" : "other",
    "lastname" : "ABCD",
    "rank" : 1,
    "organization" : "OO"
  }
}
```

spark.read.json()

```
root
 |-- person: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |    |-- organization: string (nullable = true)
 |    |-- rank: long (nullable = true)
 |    |-- role: string (nullable = true)
```

schema

person

firstname	lastname	organization	rank	role
Melena	RYZIK	abc	1	reported
other	ABCD	OO	1	

data

Missing role

Mapping JSON to Dataframes

Input: 3 documents

```
{
  "first" : "al",
  "coord" : [],
  "last" : "jr"
}
```

```
{
  "first" : "al",
  "coord" : null,
  "last" : "jr"
}
```

```
{
  "email" : "abc@ef",
  "first" : "li",
  "coord" : {
    "lat" : 45,
    "long" : 12
  },
  "last" : null
}
```

spark.read.json()

```
root
 |-- coord: string (nullable = true)
 |-- email: string (nullable = true)
 |-- first: string (nullable = true)
 |-- last: string (nullable = true)
```

schema

coord	email	first	last
[ ]	null	al	jr
null	null	al	jr
{"long":12,"lat":45}	abc@ef	li	null

Object stored as a string!!

data

Mongodb schema extraction

Couchbase schema extraction

```
{first:"al",
 last:"jr",
 coord: null,
 email:".."
}

{first:"li",
 last:"ban",
 coord:{lat:45,
 long:12}

{first:"jo",
 last:"do",
 coord:[45,12]
}
```

```
[
 [
  {#docs:3,
   properties:
   {
    first: {#docs:3, %docs:100, type:"string"},
    coord: {#docs:1, %docs:33.33, type:"object",
           properties:
            {lat: {#docs:1, %docs:100, type:"number"},
            long: {#docs:1, %docs:100, type:"number"}
           }
    email: {#docs:1, %docs:33.33, type:"string"},
    last: {#docs:3, %docs:100, type:"string"}
   },
   type: "object"
 ]
]
```

Querying nested data

- Navigating the hierarchy of objects
  - Dot-notation
    - select (level1.level2. …)
    - Traversing arrays is not allowed using the dot-notation!!
- Accessing the content of arrays
  - Flatten then navigate
    - explode(), explode_outer()
  - Use existing (built-in) array functions
    - E.g. array_exist(), array_contains()
  - Performance issues:
    - Flattening allows for standard logical optimization to be applied
    - Many research effort on how to rewrite queries to take advantage of logical optimization

Navigating structures

```
{
  "person" : {
    "firstname" : "Melena",
    "lastname" : "RYZIK",
    "role" : "reported",
    "rank" : 1,
    "organization" : ""
  }
}
```

```
{
  "person" : {
    "firstname" : "other",
    "lastname" : "ABCD",
    "rank" : 1,
    "organization" : "OO"
  }
}
```

```
root
 |-- person: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |    |-- organization: string (nullable = true)
 |    |-- rank: long (nullable = true)
 |    |-- role: string (nullable = true)
```

select("person.*")

firstname	lastname	organization	rank	role
Melena	RYZIK	abc	1	reported
other	ABCD	OO	1	

Accessing Arrays

```
{
  "persons" : [
    {
      "firstname" : "Melena",
      "lastname" : "RYZIK",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    },
    {
      "firstname" : "derba",
      "lastname" : "OKYZ",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    }
  ]
}
```

```
{
  "persons" : [
    {
      "firstname" : "other",
      "lastname" : "ABCD",
      "rank" : 1,
      "organization" : "OO"
    }
  ]
}
```

select(**explode**(col("persons")))

```
root
 |-- persons: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- firstname: string (nullable = true)
 |    |    |-- lastname: string (nullable = true)
 |    |    |-- organization: string (nullable = true)
 |    |    |-- rank: long (nullable = true)
 |    |    |-- role: string (nullable = true)
```

```
root
 |-- person: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |    |-- organization: string (nullable = true)
 |    |-- rank: long (nullable = true)
 |    |-- role: string (nullable = true)
```

Accessing Arrays

```
root
 |-- persons: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- firstname: string (nullable = true)
 |    |    |-- lastname: string (nullable = true)
 |    |    |-- organization: string (nullable = true)
 |    |    |-- rank: long (nullable = true)
 |    |    |-- role: string (nullable = true)
```

persons

firstname	lastname	organization	rank	role
other	ABCD	OO	1	

firstname	lastname	organization	rank	role
derba	OKYZ		1	

...

firstname	lastname	organization	rank	role
other	ABCD	OO	1	

```
root
 |-- person: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |    |-- organization: string (nullable = true)
 |    |-- rank: long (nullable = true)
 |    |-- role: string (nullable = true)
```

person

firstname	lastname	organization	rank	role
other	ABCD	OO	1	

firstname	lastname	organization	rank	role
derba	OKYZ		1	

firstname	lastname	organization	rank	role
other	ABCD	OO	1	

Lab session about wrangling JSON data

JSON Schema

- Community-driven evolving standard
- Many drafts, latest 12-2020, supported by several implementations
- Main goal: describe collections of documents
- Objects:
  - names for properties, default schema, required or optional, associated schemas, number of properties
- Arrays:
  - length, schemas at each index, value, existence, uniqueness
- Primitive types:
  - strings (length, pattern)
  - numbers (bounds, multiplicity)
- Expressive language:
  - logical connectives and references
- JSON syntax

JSON Schema illustrated

```
{ "type" : "object",
  "properties"  :  { "name" : { "type":"string"} },
  "patternProperties" : {
                "a(b|c)a" : { "type" : "number", "multipleOf" : 2} },
  "additionalProperties" : {
                "type": "number",
                "minimum" : 1,
                "maximum" : 1 }
            }
}
```

- names for properties, default schema
- required or optional
- associated schemas
- size of the object

```
 {
  "type" : "array",
  "items"  : [ { "type" : "string" }, { "type" : "string" } ],
  "additionalItems"  : { "type" : "number" }, "uniqueItems" : true
 }
```

- schemas at each index, default schema
- distinct values
- length of the array

Expressive language, logical operators

Mongodb built-in validation

- Support for JSON Schema draft4
- Flexible management:
  - possibility to enforce on existing collection
  - schema checked upon new inserts and updates
  - 2 options in case of failure: reject or simply log invalidity
  - pre-existing documents unchecked, until updated
- pros
  - agile development of application w/o restrictions
- cons
  - querying both valid and invalid documents

Illustration

```
validator: { $jsonSchema: {
     bsonType: "object",
     required: [ "phone" ],
     properties: {
        phone: {
           bsonType: "string",
           description: "must be a string and is required"
        },
        email: {
           bsonType : "string",
           pattern : "@su\\.fr$",
           description: "must be a string and end with
'@su.fr'"
        }
     }
  } }
```

schema

{ name: "Amanda",
email: "amanda@xyz.com" }

objects to insert

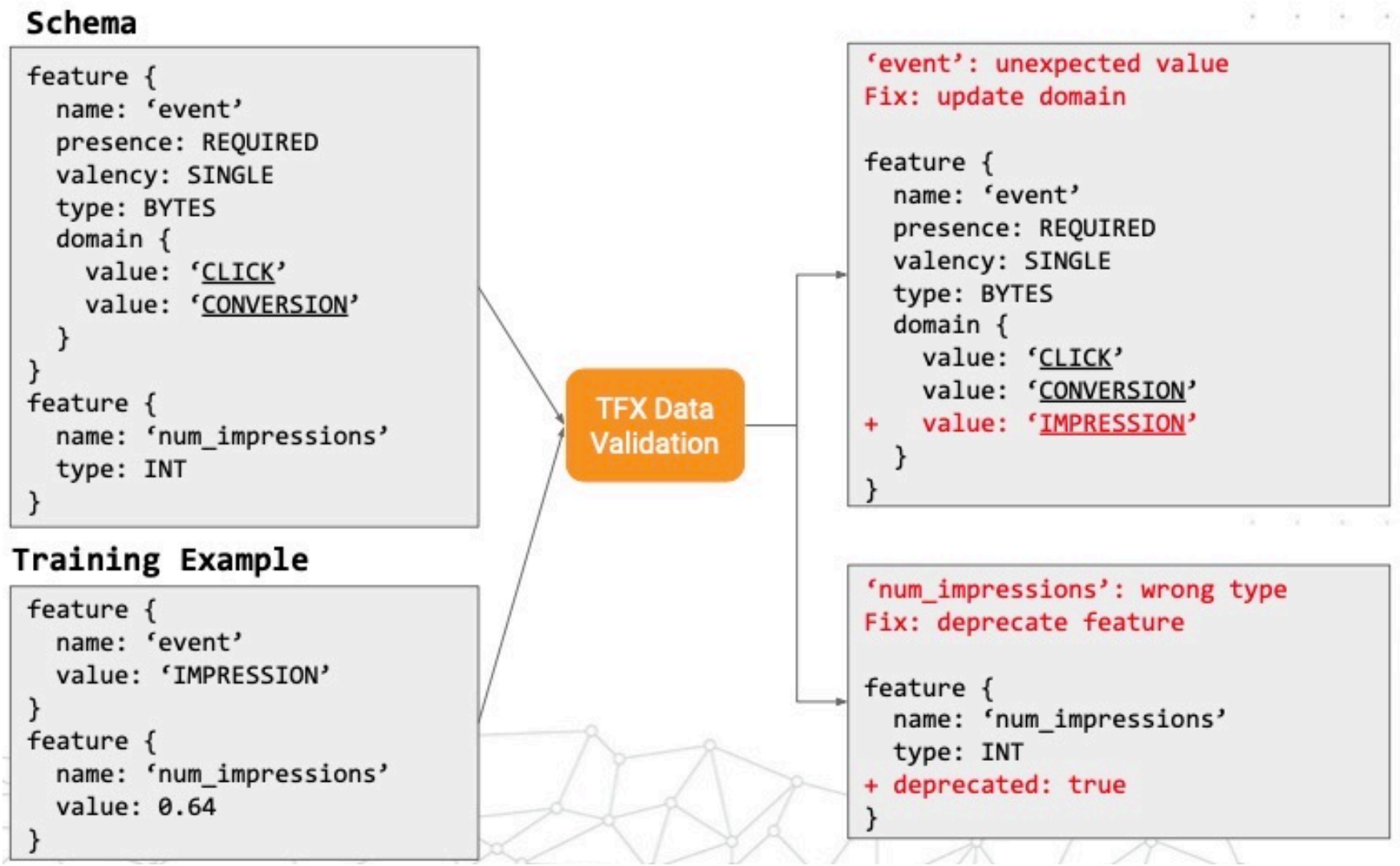
outcome:
reject due to email
not conforming to the pattern

TFX validation for ML pipelines

- Context
  - ML in production is hard, mostly due to data quality issues
  - Need to analyze, validate, track data quality
- Goal
  - Add validation/testing capabilities to TensorFlow
  - Validation kinds:
    - schema: infer from original data, proposes evolution, curation by users
    - skew detection: distribution change over time
    - model validation: assumption in training logic, generate test data from schema

Breck et al. Data Validation for Machine Learning. *SysML 2019*

TFX schema validation

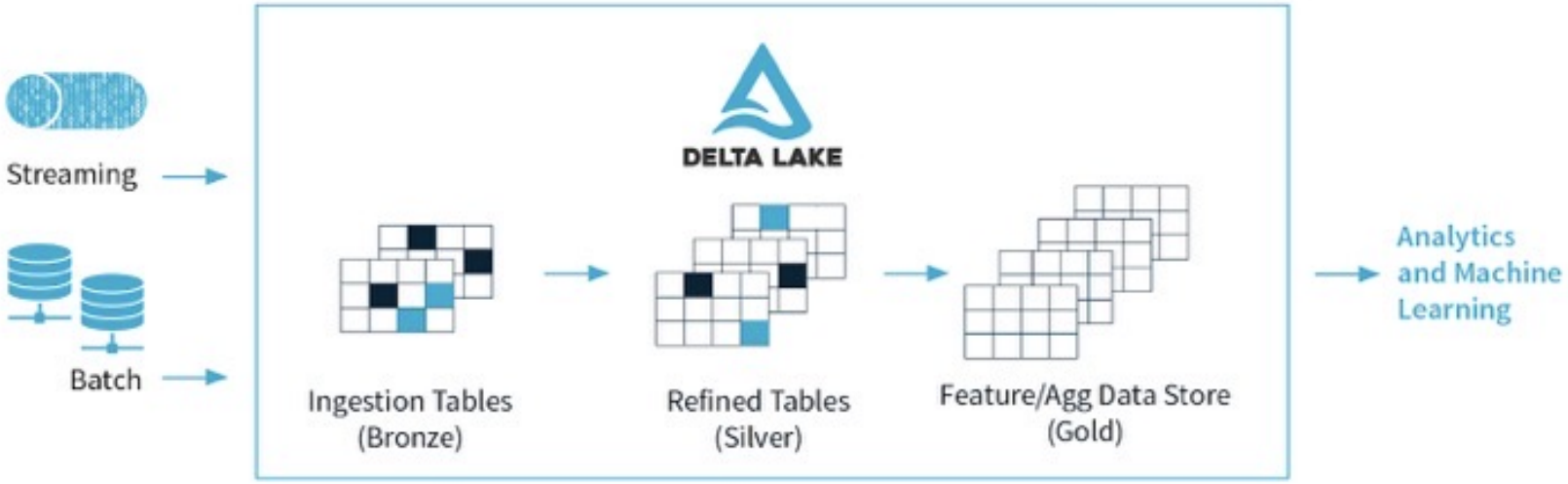


Datalakes management: the apache delta approach

Apache delta

- Addresses a sub-set of the challenges
- Open-source solution for:
  - Support for DDL and DML operations
  - ACID transactions - optimistic concurrency control
  - Data versioning
  - Schema enforcement and evolution
  - Scalable metadata management
- Principles
  - decouple storage from compute (cloud requirement)
  - platform-agnostic (Spark, Flink,…)
  - unified format (Parquet)

A typical scenario: analytics



- Multi-stage architecture (called Medallion):
  - ingestion (Bronze), feature engineering (silver), training/prediction (gold)
- Create feature store
- Reproducible experiments with ML Flow
  - recreate state of data at specific point-in time
  - rollback unwanted changes, based on model quality

Data definition in Delta

- Support for SQL DDL (partially)
- Table creation and constraints definition
  - not-null, default values
  - column assertion
  - generated columns using functions
  - No referential constraints
- Partitioning
  - one or more columns → speed up queries
- Storage in parquet

Table creation

```
CREATE TABLE default.persons (
    serial INT NOT NULL,
    name STRING,
    birthDate TIMESTAMP,
    city STRING
  ) USING DELTA
PARTITIONED BY (city);
```

```
ALTER TABLE default.persons
ADD CONSTRAINT allowedAge
        CHECK (birthDate > '2000-01-01');
```

serial	name	birthDate	city
1234	Anna	2002-2-12	Paris
4563	John	**1999-12-31**	London

Generated columns

```
DeltaTable.createOrReplace(spark) \
  .tableName("default.sales") \
  .addColumn("saleid", "STRING") \
  .addColumn("saledate", "TIMESTAMP") \
  .addColumn("quantity", "INT") \
  .addColumn("year", "INT", generatedAlwaysAs="YEAR(saledate)") \
  .addColumn("month", "INT", generatedAlwaysAs="MONTH(saledate)") \
  .addColumn("day", "INT", generatedAlwaysAs="DAYOFMONTH(saledate)") \
  .partitionedBy("year", "month") \
  .execute()
```

saleid	saledate	quantity	year	month	day
S124	2023-02-26	2	2023	02	26
S184	2023-03-16	2	2023	03	14

Data modification

- Support change data capture (CDC) operations
  - Standard SQL DML
    - update, delete, insert, upsert (merge)
- Optimized evaluation
  - clustering, z-ordering data skipping, deletion vectors
- Automatic versioning
  - old data is automatically kept
  - Vacuuming on demand or retention threshold
- Systematic schema validation
- Support for schema evolution

Update

- Modify the column value for a set of rows, possibly selected using a condition
- SQL compatible Syntax

  update <Table>
  set column = <expression>
  [where conditon]

```
UPDATE delta.`/tmp/deltaSales`
SET unitprice = unitprice * 1.05
WHERE saledate >= '2023-01-01' and category='Furniture'
```

Useful metric to monitor: num_affected_rows

Delete

- Remove a set of rows, possibly selected using a condition
- SQL compatible Syntax

  delete from <Table>
  [where conditon]

```
DELETE FROM delta.`/tmp/deltaSales`
WHERE saledate < '2023-01-01'
```

Useful metric to monitor: num_affected_rows

Upsert

- Combined operator: update, delete and insert
- Operates on two tables with joinable column(s)
  - Source: provides new values/rows to set/insert
  - Target: where update, insert or delete takes place
- Specify a matching condition and associated actions
- *When Matched*
  - Delete or update matching rows from target using values from source
  - Possibility to specify a filtering condition
- *When Not Matched*
  - Insert rows from source into target

Use case1: insert rows

Persons

serial	name	age	address
12345	Alice	25	123 Main St
67890	Bob	30	456 Oak Ave
24680	Charlie	35	789 Elm St

New Persons

serial	name	age	address
78120	Dan	42	432 Holly Rd
12345	Alice	25	123 Main St
97362	Lorry	..	…

```
MERGE INTO delta.`/tmp/persons` AS oldData
USING newPersons
ON oldData.serial = newPersons.serial
WHEN NOT MATCHED
  THEN INSERT *;
```

```
+-----------------+----------------+----------------+----------------+
|num_affected_rows|num_updated_rows|num_deleted_rows|num_inserted_rows|
+-----------------+----------------+----------------+----------------+
|                2|               0|               0|               2|
+-----------------+----------------+----------------+----------------+
```

Use case 2: update rows

Salaries

Serial	Salary
24680	36000
12345	45000
…	…

newSalaries

Serial	Salary
24680	46000
12345	47000
…	…

```
MERGE INTO delta.`/tmp/salaries` AS oldData
USING newSalaries
ON oldData.serial = newSalaries.serial
WHEN MATCHED AND oldData.salary<50000
  THEN UPDATE SET oldData.salary=newSalaries.salary;
```

update the salary of the employees who earn less than 50,000

Use case 3: combine information

sales

product_id	quantity	totalprice
BED_4	1	300
SHO_15	2	60
CHA_2	2	60

New sales

product_id	quantity	totalprice
SHO_15	3	90
CHA_2	1	30
BED_6	1	200

```
MERGE INTO delta.`/tmp/sales` AS oldData
USING newSales
ON oldData.product_id = newSales.product_id
WHEN MATCHED
  THEN UPDATE SET oldData.quantity = oldData.quantity + newSales.quantity,
                  oldData.totalprice = oldData.totalprice + newSales.totalprice
WHEN NOT MATCHED
  THEN INSERT *
```

```
+-----------------+----------------+----------------+-----------------+
|num_affected_rows|num_updated_rows|num_deleted_rows|num_inserted_rows|
+-----------------+----------------+----------------+-----------------+
|                3|               2|               0|                1|
+-----------------+----------------+----------------+-----------------+
```

Use case 4: Merge schemas

sales

product_id	quantity	totalprice
BED_4	1	300
SHO_15	2	60
CHA_2	2	60
BED_6	1	200

products

product_id	category	color
CHA_2	Furniture	blue
SHO_15	Cloth	black
BED_4	Furniture	brown

```
MERGE INTO delta.`/tmp/sales` oldData
USING products
ON oldData.product_id = products.product_
WHEN MATCHED
  THEN UPDATE SET *
WHEN NOT MATCHED
  THEN INSERT *
```

```
+-----------------+-----------------+--
|num_affected_rows|num_updated_rows|nu
+-----------------+-----------------+--
|                3|                3|
+-----------------+-----------------+--
```

product_id	quantity	totalp	Category	color

Remarks

- Source and target have different schemas
  - Default: fail
  - schema evolution activated: outer join effect
- multiple matches from source
  - non authorized because ambiguous, failure
- Multiple When(Not)Matched statements
  - Evaluation follows order of appearance
- *When Not Matched By source*
  - Delete or update target rows

Schema enforcement

- Reject writes not conforming to the schema
- Conformance logic
  - source data (sd), target table (tt)
  - accept only if: if schema(sd) is a subset of/equal to schema(tt) and all columns are type-compatible
  - in case of missing values, set to null

Schema evolution

- Alternative to strict enforcement
- Useful in many realistic scenarios
  - introducing new (computed) columns
  - changing the original schema (e.g column types)
- Possible options
  - mergeSchema: add missing sf columns into tt
  - overwriteSchema: alter tt schema according to sf

Use case 5: combine information

sales

product_id	quantity	totalprice	status
BED_4	1	300	Available
SHO_15	2	60	Available
CHA_2	2	60	Available

New sales

product_id	quantity	totalprice
SHO_15	3	90
CHA_2	1	30
BED_6	1	200

```
MERGE INTO delta.`/tmp/sales` AS oldData
USING newSales
ON oldData.product_id = newSales.product_id
WHEN MATCHED
  THEN UPDATE SET …
WHEN NOT MATCHED BY SOURCE
  THEN UPDATE SET oldData.status = 'unavailable'
```

```
+-----------------+---------------+---------------+-----------------+
|num_affected_rows|num_updated_rows|num_deleted_rows|num_inserted_rows|
+-----------------+---------------+---------------+-----------------+
|                3|              3|              0|                0|
+-----------------+---------------+---------------+-----------------+
```

Data versioning

- Useful in many contexts
  - reproducing experiments (eg. change parameters)
  - rolling back (address bugs, quality issues)
  - auditing data (legal issues)
- Retrieving or restoring past versions
  - By snapshot timestamp (ts)
  - by version number (ver#)
- history analysis
  - detailed log of operations associated ver# and ts

Change data feed

- Row-level change information
  - Recorded upon request
  - delete, insert, update information
  - update: pre-image and post-image
  - Useful for fine-grained auditing

Illustration

Sales table with CDF enabled

saleid	Saledate	….	unitprice	…

```
+-------+
|nb_rows|
+-------+
|   4916|
+-------+
```

```
UPDATE delta.`/tmp/deltaSalesCDF`
SET unitprice = unitprice * 1.05
WHERE saledate >= '2023-02-01' and category='Cloth'
```

```
+-----------------+
|num_affected_rows|
+-----------------+
|              765|
+-----------------+
```

```
SELECT _change_type, _commit_version, _commit_timestamp, count(*)
FROM table_changes_by_path('/tmp/deltaSalesCDF', 0)
GROUP BY _change_type, _commit_version, _commit_timestamp
```

```
+----------------+----------------+--------------------+--------+
|    _change_type|_commit_version|   _commit_timestamp|count(1)|
+----------------+----------------+--------------------+--------+
| update_preimage|               1|2024-10-10 19:40:...|     765|
|update_postimage|               1|2024-10-10 19:40:...|     765|
|          insert|               0|2024-10-10 19:34:...|    4916|
+----------------+----------------+--------------------+--------+
```

Illustration (cont'd)

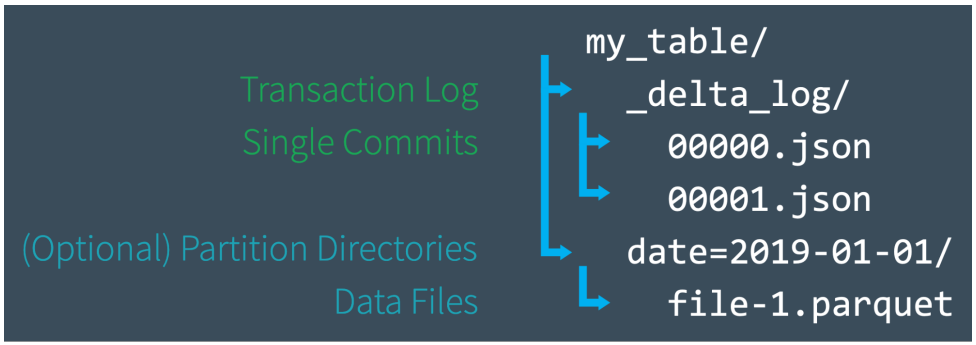
```
SELECT saleid, _change_type, unitprice
FROM table_changes_by_path('/tmp/deltaSalesCDF', 0)
WHERE saledate >= '2023-02-01' and category='Cloth' and _commit_version = 1
CLUSTER BY saleid
```

saleid	_change_type	unitprice
S000000374	update_preimage	20.0
S000000374	update_postimage	21.0
S000000609	update_preimage	60.0
S000000609	update_postimage	63.0
S000001229	update_preimage	22.0
S000001229	update_postimage	23.1
S000004225	update_preimage	60.0
S000004225	update_postimage	63.0
S000002542	update_preimage	60.0
S000002542	update_postimage	63.0

Update …
SET unitprice =
unitprice * 1.05
….

Delta internals: Logging

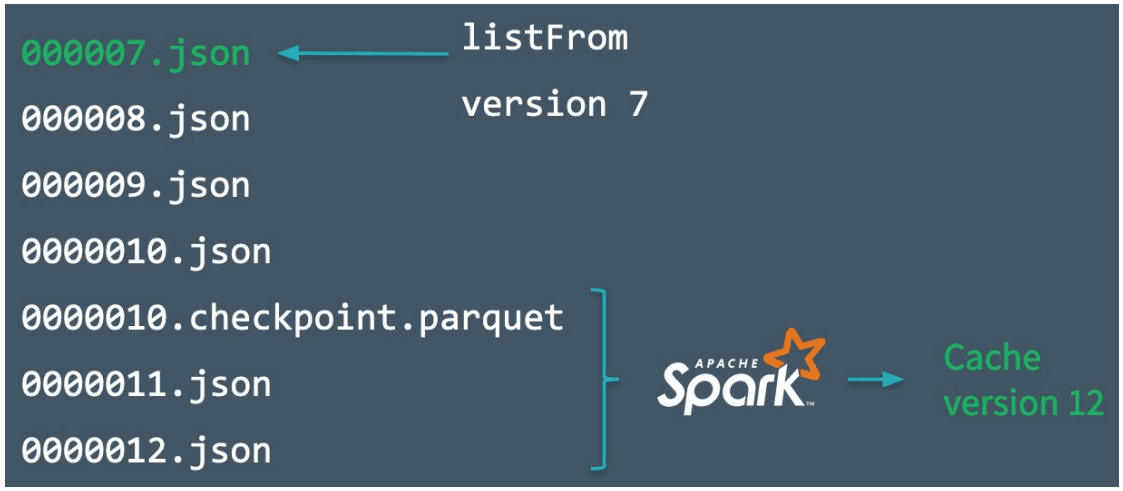
- Goal: Keep a Single Source of Truth
- Transaction log
  - Central repository tracks all changes to table
  - checked by the system before applying changes
  - initialized upon creation of a table
    - JSON files named with increasing version numbers
  - based on a set of atomic actions
    - add/remove file, updated metadata (table name, schema part), commit info + 2 other operations



Delta internals: Checkpointing

- Keep intermediate state of a table
  - alternative to playing the deltas in the JSON files
- Automatically generated, every 10 transactions, to maintain good performance

to obtain version 12,
apply delta11 and 12 on checkpoint
Gain: skip delta7-> 10



Delta Live Tables

- Proprietary delta solution for Databricks
- Add-ons
  - Constraints enforcement
  - Expectations: assertions on tables using simple conditions, evaluated on streaming, non-blocking operations only
  - associate discard/keep policy
- Optimization at large scale, avoid overhead

Armbrust et al. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. CIDR 2021

Data quality management at scale

Data quality: overview

- Context
  - Data integrated from different sources
  - often produced w/o schema nor quality guarantees
- Improving data quality impacts all applications
  - Business intelligence: avoid wrong decision
  - Machine learning: improve model performance
  - Pipelines: prevent glitches (null values, type mis match…)
- Manually checking DQ may be complex
  - data volume and dynamicity
  - user-defined code: cumbersome and error prone
- Automation for quality check

Data quality dimensions

- Standard classification, extensive literature
- Dimensions
  - Completeness
    - degree to which an entity reflects the real-world
    - general: column with missing (null) values
    - contextualized: absence of value means not-applicable
  - Consistency
    - adherence to semantic rules: categorical column, referential constraints (foreign key)
  - Accuracy
    - syntactic: column value adheres to its type
    - semantic: domain-specific
- Several associated metrics

The Deequ approach

- Declarative specification and verification of DQ metrics
- Draws inspiration from software engineering: unit tests, cont. testing
- Features
  - A rich set of useful metrics
  - Suggested based on confidence score or verified upon user request
  - Use: enforce constraints upon ingestion, non valid data is quarantined
  - Low overhead: automatic optimization of the metric computation
  - Incremental maintenance: cope with data dynamicity
  - Deployed on Apache Spark and operational on AWS suite
  - Open source implementation (py)Deequ

Schelter et al. Automating large-scale data quality verification. VLDB 2018

Supported metrics

constraint	arguments	semantic
dimension *completeness*		
`isComplete`	column	check that there are no missing values in a column
`hasCompleteness`	column, udf	custom validation of the fraction of missing values in a column
dimension *consistency*		
`isUnique`	column	check that there are no duplicates in a column
`hasUniqueness`	column, udf	custom validation of the unique value ratio in a column
`hasDistinctness`	column, udf	custom validation of the unique row ratio in a column
`isInRange`	column, value range	validation of the fraction of values that are in a valid range
`hasConsistentType`	column	validation of the largest fraction of values that have the same type
`isNonNegative`	column	validation whether all values in a numeric column are non-negative
`isLessThan`	column pair	validation whether values in the 1s column are always less than in the 2nd column
`satisfies`	predicate	validation whether all rows match predicate
`satisfiesIf`	predicate pair	validation whether all rows matching 1st predicate also match 2nd predicate
`hasPredictability`	column, column(s), udf	user-defined validation of the predictability of a column

Supported metrics

constraint	arguments	semantic
statistics (can be used to verify dimension *consistency*)		
`hasSize`	udf	custom validation of the number of records
`hasTypeConsistency`	column, udf	custom validation of the maximum fraction of values of the same data type
`hasCountDistinct`	column	custom validation of the number of distinct non-null values in a column
`hasApproxCountDistinct`	column, udf	custom validation of the approx. number of distinct non-null values
`hasMin`	column, udf	custom validation of a column's minimum value
`hasMax`	column, udf	custom validation of a column's maximum value
`hasMean`	column, udf	custom validation of a column's mean value
`hasStandardDeviation`	column, udf	custom validation of a column's standard deviation
`hasApproxQuantile`	column, quantile, udf	custom validation of a particular quantile of a column (approx.)
`hasEntropy`	column, udf	custom validation of a column's entropy
`hasMutualInformation`	column pair, udf	custom validation of a column pair's mutual information
`hasHistogramValues`	column, udf	custom validation of column histogram
`hasCorrelation`	column pair, udf	custom validation of a column pair's correlation
time		
`hasNoAnomalies`	metric, detector	validation of anomalies in time series of metric values

Metrics computation

Completeness $|\{d \in D \mid d(col) \neq \texttt{null}\}| / N$

Compliance $|\{d \in D \mid p(d)\}| / N$ *ratio of records sat. predicate p*

Uniqueness $|\{v \in V \mid c_v = 1\}| / |V|$ *ratio of unique values, c_v card of v*

Distinctness $|V| / N$

Entropy $-\sum_v \frac{c_v}{N} \log \frac{c_v}{N}$

Mutual Information $\sum_{v_1} \sum_{v_2} \frac{c_{v_1 v_2}}{N} \log \frac{c_{v_1 v_2}}{c_{v_1} c_{v_2}}$

v1 and v2 refer to different columns

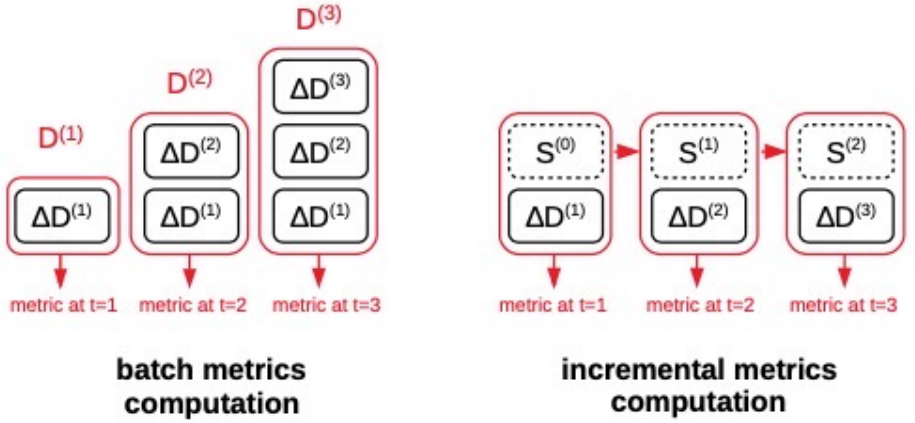
ApproxQuantile and ApproxCountDistinct: SOTA algorithms (refer to the paper)

Predictability: predict column from other columns - max. a posteriori decision rule

Incremental computation

Goal: avoid batch computation which may be costly!

Assumption: append-only updates



notation ΔD. newly added records ΔV denote all unique values

general formulae $S^{(t)} = f(\Delta D^{(t)}, S^{(t-1)})$

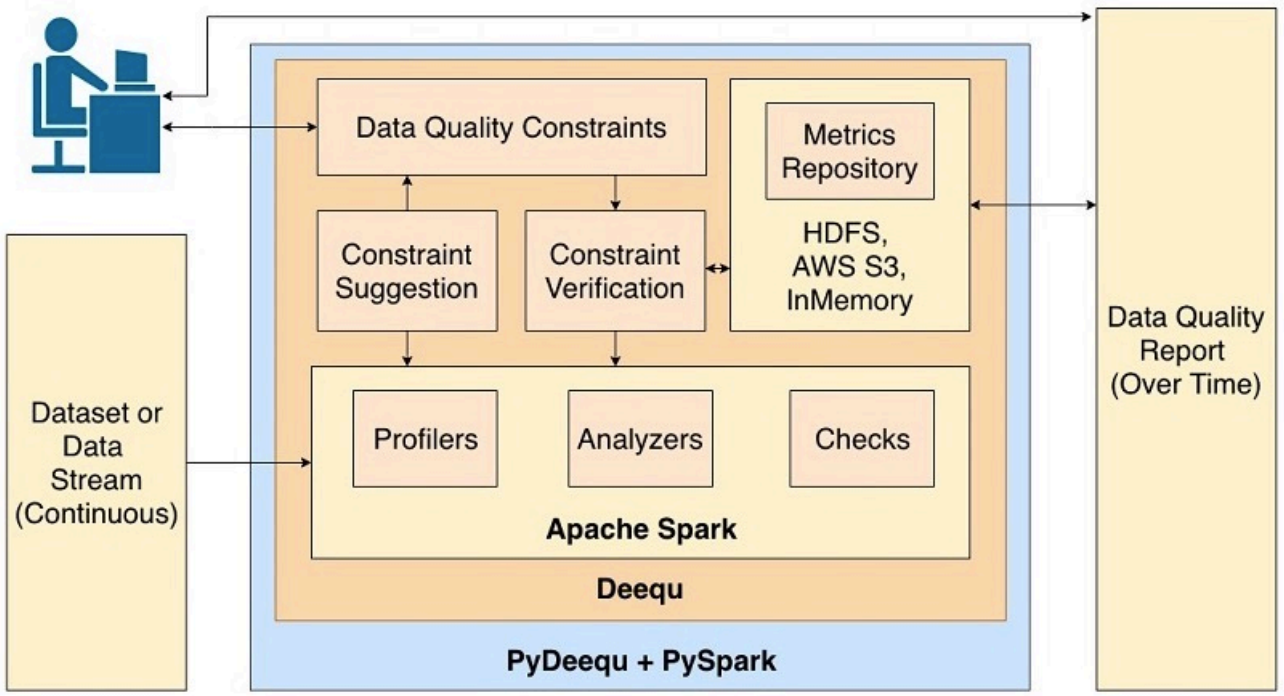
Compliance $$\frac{|\{d \in D \mid p(d)\}| + |\{d \in \Delta D \mid p(d)\}|}{N + \Delta N}$$

Uniqueness $$\frac{|\{v \in V \cup \Delta V \mid c_v + \Delta c_v = 1\}|}{|V \cup \Delta V|}$$

other formulas in the paper

Architecture and typical scenario

User
- runs analyses,
- requests profiles or suggestions
- enforces constraints



Analyzer

- Features
  - request the computation of a specific metric
  - optimizes the resulting computations: minimize data scans
  - Store the internal states of the computation to enable incremental computations.
- Usage
  - onData(df:DF): set the data source
  - addAnalyzer(metric): set the metric to be computed. Invoke many times
  - run(): launch the computations
  - Possible to show the report as a dataframe (see demo notebook)

Analyzer illustrated

```
analysisResult = AnalysisRunner(spark) \
                .onData(_data) \
                .addAnalyzer(Size()) \

.addAnalyzer(Completeness(_colName)) \

.addAnalyzer(CountDistinct(_colName)) \

.addAnalyzer(Distinctness(_colName))\
                .run()
```

```
+-------+----------+-------------+--------------------+
|entity |instance  |name         |value               |
+-------+----------+-------------+--------------------+
|Dataset|*         |Size         |37000.0             |
|Column |vehicleType|Completeness |0.8987567567567567  |
|Column |vehicleType|CountDistinct|8.0                 |
|Column |vehicleType|Distinctness |2.4057256269922414E-4|
+-------+----------+-------------+--------------------+
```

```
analysisResult = AnalysisRunner(spark) \
                .onData(_data) \
                .addAnalyzer(Entropy(..))\
                .addAnalyzer(Correlation( , ))\
                .run()
```

```
+----------+------------+----------+-----------
|entity    |instance    |name      |value
+----------+------------+----------+-----------
|Mutlicolumn|powerPS,price|Correlation|0.0055233714
|Column    |vehicleType |Entropy   |1.7715625847
+----------+------------+----------+-----------
```

Profiler

- Analyzes each column w.r.t. common metrics
- General case: completeness, approx. distinct values
- Numerical columns: statistical information (Mean, max, std…)

```
result = ColumnProfilerRunner(spark) \
   .onData(_data.select('name','powerPs')) \
   .run()
```

```
StandardProfiles for column: name: {
    "completeness": 1.0,
    "approximateNumDistinctValues": 26262,
    "dataType": "String",
    "isDataTypeInferred": false,
    "typeCounts": {

        ….
```

```
NumericProfiles for column: powerPs: {
    "completeness": 1.0,
    "approximateNumDistinctValues": 446,
    "dataType": "Integral",
    "isDataTypeInferred": false,
    "typeCounts": {},
    "histogram": null,
    "kll": "None",
    "mean": 116.16367567567568,

        ….
```

Constraint suggestions

- Hint users with relevant constraints
- Based on heuristics, single column profiling
- Three passes:
  - 1st pass: type detection, size, approx. # distinct values, completeness
  - 2nd pass -numeric types only-: desc. Stats (min, max, std), approx. quartiles
  - 3rd pass: frequency distribution and apply heuristics
- Some heuristics
  - For complete columns, suggest isComplete(), otherwise, hasCompleteness() with an estimated value
  - For *numeric* columns, suggest compliance() with an interval-based predicate
  - For columns s.t # distinct values < T suggest isInRange (consider categorical)

Constraint suggestions illustrated

```
suggestionResult = ConstraintSuggestionRunner(spark) \
        .onData(_data.select('fuelType','powerPs')) \
        .addConstraintRule(DEFAULT()) \
        .run()
```

```
{'code_for_constraint': '.isContainedIn("fuelType", '
                        '["benzin", "diesel", '
                        '"lpg", "cng", "hybrid", '
                        '"andere", "elektro"])',
 'column_name': 'fuelType',
 'constraint_name': "ComplianceConstraint(Compliance('fuelType'
                    "has value range 'benzin', "
                    "'diesel', 'lpg', 'cng', "
                    "'hybrid', 'andere', "
                    "'elektro',`fuelType` IN "
                    "('benzin', 'diesel', 'lpg', "
                    "'cng', 'hybrid', 'andere', "
                    "'elektro'),None))",
 'current_value': 'Compliance: 1',
 'description': "'fuelType' has value range "
                "'benzin', 'diesel', 'lpg', 'cng', "
                "'hybrid', 'andere', 'elektro'",
 'rule_description': 'If we see a categorical '
                     'range for a column, we '
                     'suggest an IS IN (...) '
                     'constraint',
 'suggesting_rule': 'CategoricalRangeRule()'},
```

```
{'code_for_constraint': '.isNonNegative("powerPs
 'column_name': 'powerPs',
 'constraint_name': "ComplianceConstraint(Compli
                    'has no negative '
                    'values,powerPs >= 0,None))
 'current_value': 'Minimum: 0.0',
 'description': "'powerPs' has no negative value
 'rule_description': 'If we see only non-negati
                     'numbers in a column, we '
                     'suggest a corresponding '
                     'constraint',
 'suggesting_rule': 'NonNegativeNumbersRule()'}
```

More on constraints suggestion

- Evaluated on held-out data
- Exploit column names
  - id_xx may suggest unique, is_xx_yy Boolean
  - Use open source projects
- Anomaly detection
- Related techniques
  - TensorFlow validation: based on schema for nested data
    - **Baylor et al**. Tfx: A tensorflow-based production-scale machine learning platform. KDD ’17
  - Semantic type detection: richer types (categorical, ...)
    - **Shah et al.** Towards benchmarking feature type inference for automl platforms. VLDB 2021

Constraint verification

- Specify the predicates in a declarative manner
- Submit to system, report with status (failure/success) and message

```
checkResult = VerificationSuite(spark) \
   .onData(_data) \
   .addCheck(       check  .isComplete('price')      .isNonNegative('price')
.isUnique('seller'))\
   .run()
```

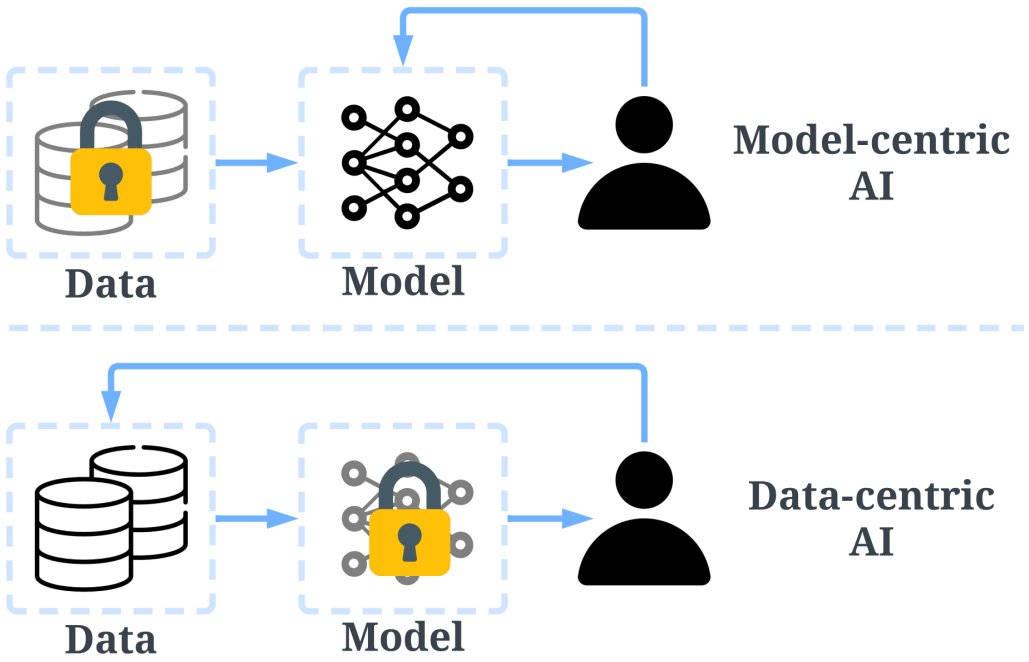
```
+------------+-----------+------------+--------------------+-----------------+--------------------+
|       check|check_level|check_status|          constraint|constraint_status|  constraint_message|
+------------+-----------+------------+--------------------+-----------------+--------------------+
|Review Check|    Warning|     Warning|CompletenessConst...|          Success|                    |
|Review Check|    Warning|     Warning|ComplianceConstra...|          Success|                    |
|Review Check|    Warning|     Warning|UniquenessConstra...|          Failure|Value: 0.0 does n...|
+------------+-----------+------------+--------------------+-----------------+--------------------+
```

Efficient computation of metrics

- Aggregation queries
- Group statistics on a single query
  - Avoid repeated processing of the data
  - Strategy: multi-query optimization to enable scan-sharing
- Stateful computation
  - Deal with updates (insertions)
  - Compute on delta and propagate
- Algebraic representation
  - Schelter et al. Differential Data Quality Verification on Partitioned Data. ICDE'2019
  - Commutative monoids

Building effective ML pipelines

Data-centric AI vs. Model-centric AI



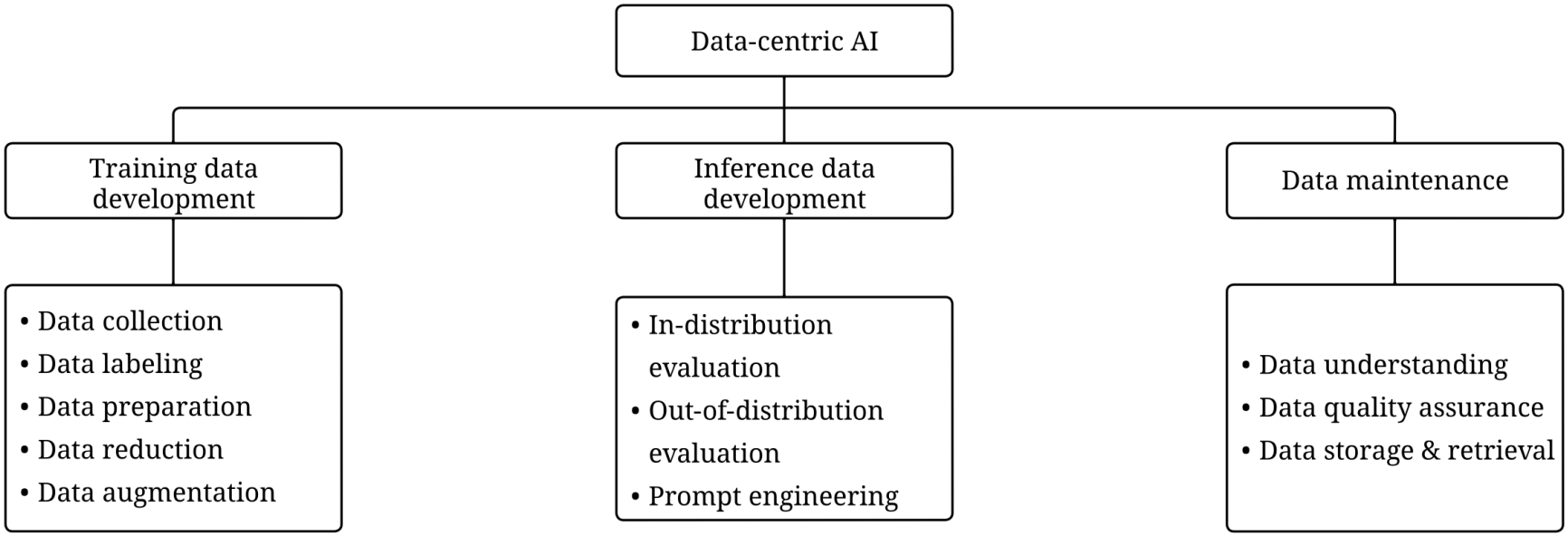
focus on model
data unchanged

focus on data
systematic engineering
of data to build AI

Zha, Daochen, et al. "Data-centric Artificial Intelligence: A Survey."

https://github.com/daochenzha/data-centric-AI

Data-centric AI Framework



Zha, Daochen, et al. "Data-centric Artificial Intelligence: A Survey."

https://github.com/daochenzha/data-centric-AI

Which solutions

- Data profiling, quality assessment
  - Deequ (Amazon)[2], Delta Lake (Databricks), …
- Continuous data validation
  - Delta Lake, TFX validation [1]
- Manual approach
  - Error prone, costly, not always scalable
- Semantic type detection [3]
  - Use ML to classify column types
  - Extract richer information about types (date, location, name, … vs basic types like text/numbers/…)

[1] Breck et al. Data validation for machine learning. In Proceedings of SysML, 2019.
[2]Schelter et al. Automating large-scale data quality verification. VLDB 2018
[3] Shah et al. Towards benchmarking feature type inference for automl platforms. SIGMOD 202

Focus: feature engineering

- Real data uses a rich set of types
  - text, number, Booleans, timestamps, ...
- ML algorithms expect numeric data
- dealing with real data may be challenging
  - Fix/clean dirty data, deal with missing values, outliers
  - Enrich (collect additional data)
  - Decide whether feature is categorical or continuous?
- Model inference (and prediction) precision impacted by the data quality
  - Recall the garbage-in garbage-out principle

Spark Machine Learning Library

- Effectiveness
  - Streamlined integration with data-prep pipeline
  - Large set of supervised and unsupervised models
  - Model selection and tuning, grid search, cross validation
- Efficiency
  - Distributed processing: large datasets, parallel training large set of parameters
  - Main-memory and caching capabilities
- Native Stream processing
  - Prediction in continuous for unseen data
- Usage
  - High-level API backed with an optimized lower API

Spark ML main ingredients

- Largely inspired by SickitLearn
- Transformer
  - Create features or perform prediction (using a trained model)
  - Invoke transform()
  - Ex. feature transformation:
    - Input : Dataframe with n columns of numbers -> a dataframe with one column of n-dimensional vectors
  - Ex. prediction
    - Input : Dataframe with a features vector -> the input dataframe augmented with predictions column
- Estimator
  - trains an ML model on the data (ex. logistic regression)
  - Invoke fit()

Spark ML main ingredients

- Parameter
  - A uniform class for describing parameters passed to an estimator or extracted from a transformer
  - Ex. for decision tree inference: the number of nodes, the selection criterion (info gain or Gini index), ..
- Pipeline
  - Sequence of stages performing a specific ML algorithm
  - A stage = either an estimator or a transformer
  - Usually Linear, DAG are also possible (specified using a topological order)
- Evaluator
  - Several metrics (MAE, RMSE, …)

Spark ML Data model

- Builds on the Dataset
  - Basic types: boolean, numeric (integer, decimal, …), String, null, timestamp
  - Complex types: arrays, structures, maps
  - User-defined types
- Support for the Vector type
  - Part of the org.apache.spark.ml.linalg package
  - Seen as a UDT
  - An n-dimensional structure of *Doubles*
  - Possibility to use the **dense** or the **sparse** variant
  - And to convert dense to spare or vice versa

Dense vs Sparse Vectors

- Dense
  - Sequence of values [v1, v2, ….]
  - E.g [0,1,3,0]
- Sparse
  - Optimized storage by storing non-0 values only!
  - Only interesting when the ratio of 0-values is very high
  - Tuple (s, I, V) indicating
    - s = the vector size
    - I = a sequence indicating the indices of non-0 values as per a dense vector
    - V = the sequence of non-0 values
  - E.g (4, [1,2], [1,3]) encodes [0,1,3,0]

Dense vs Sparse Vectors

```
from pyspark.ml.linalg import Vectors

vec1 = Vectors.dense(1.0, 1.0, 18.0)
vec2 = Vectors.dense(0.0, 2.0, 20.0)
vec3 = Vectors.sparse(3,[0.0,2.0],[1.0,18.0])
vec4 = Vectors.sparse(3,[0.0,1.2,2.0],[2.0,3.0,11.0])
vectors =
spark.sparkContext.parallelize([vec1,vec2,vec3,vec4
])
vectors.collect()
```

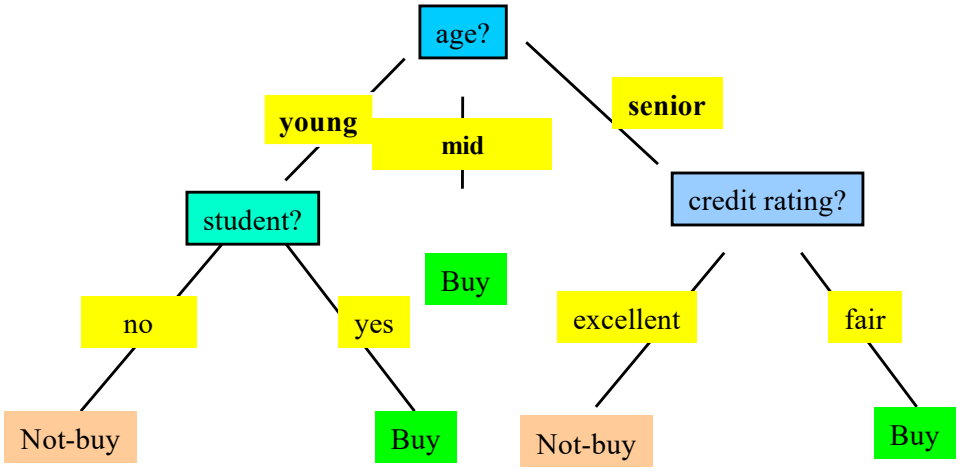
```
[DenseVector([1.0, 1.0, 18.0]), DenseVector([0.0, 2.0,
20.0]), SparseVector(3, {0: 1.0, 2: 18.0}),
SparseVector(3, {0: 2.0, 1: 3.0, 2: 11.0})]
```

Case study: decision tree inference

Original data

age	income	student	credit_rating	buys_computer
young	high	no	fair	no
young	high	no	excellent	no
middle	high	no	fair	yes
senior	medium	no	fair	yes
senior	low	yes	fair	yes
senior	low	yes	excellent	no
middle	low	yes	excellent	yes
young	medium	no	fair	no
young	low	yes	fair	yes
senior	medium	yes	fair	yes
young	medium	yes	excellent	yes
middle	medium	no	excellent	yes
middle	high	yes	fair	yes
senior	medium	no	excellent	no

Training data set: Who buys computer?



Adapted from
Data Mining: concepts and techniques
by *J.Han, M. Kamber et J. Pei*

Case study: decision tree inference

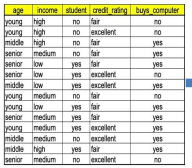
Original data

age	income	student	credit_rating	buys_computer
young	high	no	fair	no
young	high	no	excellent	no
middle	high	no	fair	yes
senior	medium	no	fair	yes
senior	low	yes	fair	yes
senior	low	yes	excellent	no
middle	low	yes	excellent	yes
young	medium	no	fair	no
young	low	yes	fair	yes
senior	medium	yes	fair	yes
young	medium	yes	excellent	yes
middle	medium	no	excellent	yes
middle	high	yes	fair	yes
senior	medium	no	excellent	no

Encoded features (what Spark ML expects)

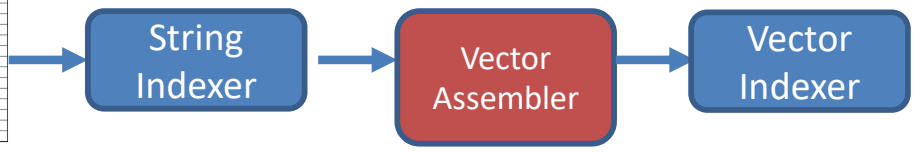
```
 |-- features: vector (nullable = true)
 |-- indexed_label: double (nullable = false)

+-----------------+-------------+
|         features|indexed_label|
+-----------------+-------------+
|[1.0,1.0,0.0,0.0]|          1.0|
|[1.0,1.0,0.0,1.0]|          1.0|
|[2.0,1.0,0.0,0.0]|          0.0|
|        (4,[],[])|          0.0|
|[0.0,2.0,1.0,0.0]|          0.0|
|[0.0,2.0,1.0,1.0]|          1.0|
|[2.0,2.0,1.0,1.0]|          0.0|
|    (4,[0],[1.0])|          1.0|
|[1.0,2.0,1.0,0.0]|          0.0|
|    (4,[2],[1.0])|          0.0|
|[1.0,0.0,1.0,1.0]|          0.0|
|[2.0,0.0,0.0,1.0]|          0.0|
|[2.0,1.0,1.0,0.0]|          0.0|
|    (4,[3],[1.0])|          1.0|
+-----------------+-------------+
```

Case study: decision tree inference



age	income	student	credit_rating	buys_computer
young	high	no	fair	no
young	high	no	excellent	no
middle	high	no	fair	yes
senior	medium	no	fair	yes
senior	low	yes	fair	yes
senior	low	yes	excellent	no
middle	low	yes	excellent	yes
young	medium	no	fair	no
young	low	yes	fair	yes
senior	medium	yes	fair	yes
young	medium	yes	excellent	yes
middle	medium	no	excellent	yes
middle	high	yes	fair	yes
senior	medium	no	excellent	no

data.csv

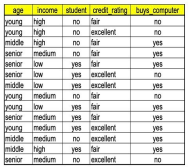



```
 |-- features: vector (nullable = true)
 |-- indexed_label: double (nullable = false)

+-----------------+-------------+
|         features|indexed_label|
+-----------------+-------------+
|[1.0,1.0,0.0,0.0]|          1.0|
|[1.0,1.0,0.0,1.0]|          1.0|
|[2.0,1.0,0.0,0.0]|          0.0|
|        (4,[],[])|          0.0|
|[0.0,2.0,1.0,0.0]|          0.0|
|[0.0,2.0,1.0,1.0]|          1.0|
|[2.0,2.0,1.0,1.0]|          0.0|
|    (4,[0],[1.0])|          1.0|
|[1.0,2.0,1.0,0.0]|          0.0|
|    (4,[2],[1.0])|          0.0|
|[1.0,0.0,1.0,1.0]|          0.0|
|[2.0,0.0,0.0,1.0]|          0.0|
|[2.0,1.0,1.0,0.0]|          0.0|
|    (4,[3],[1.0])|          1.0|
+-----------------+-------------+
```

String Indexer

- Maps a column of strings to a column of longs corresponding to indices from [0, numLabels[
- 4 ordering options:
  - Descending or ascending combined with frequency or alphabetical
- 3 possible outcomes for unseen labels:
  - Raise exception (default)
  - Skip row
  - Keep row with label = numLabels
- Behavior with missing values
  - to *setHandleInvalid( )*

String Indexer illustrated



age	income	student	credit_rating	buys_computer
young	high	no	fair	no
young	high	no	excellent	no
middle	high	no	fair	yes
senior	medium	no	fair	yes
senior	low	yes	fair	yes
senior	low	yes	excellent	no
middle	low	yes	excellent	yes
young	medium	no	fair	no
young	low	yes	fair	yes
senior	medium	yes	fair	yes
young	medium	yes	excellent	yes
middle	medium	no	excellent	yes
middle	high	yes	fair	yes
senior	medium	no	excellent	no

data.csv

```
from pyspark.ml.feature import StringIndexer

field = 'age'
age_indexer = StringIndexer(inputCol=field,\
outputCol='indexed_'+field)

df_age_idx =
age_indexer.fit(data).transform(data)
```

```
+------+------+-------+-------------+-----+-----------+
|   age|income|student|credit_rating|label|indexed_age|
+------+------+-------+-------------+-----+-----------+
| young|  high|     no|         fair|   no|        1.0|
| young|  high|     no|    excellent|   no|        1.0|
|middle|  high|     no|         fair|  yes|        2.0|
|senior|medium|     no|         fair|  yes|        0.0|
|senior|   low|    yes|         fair|  yes|        0.0|
|senior|   low|    yes|    excellent|   no|        0.0|
|middle|   low|    yes|    excellent|  yes|        2.0|
| young|medium|     no|         fair|   no|        1.0|
| young|   low|    yes|         fair|  yes|        1.0|
|senior|medium|    yes|         fair|  yes|        0.0|
| young|medium|    yes|    excellent|  yes|        1.0|
|middle|medium|     no|    excellent|  yes|        2.0|
|middle|  high|    yes|         fair|  yes|        2.0|
|senior|medium|     no|    excellent|   no|        0.0|
+------+------+-------+-------------+-----+-----------+
```

```
age: string
income: string
student: string
credit_rating: string
label: string
```

schema

train an estimator based on the frequencies

age	**Count(*)**	***Label***
Senior	5	0.0
Young	5	1.0
Middle	4	2.0

```
age: string
income: string
student: string
credit_rating: string
label: string
indexed_age: double
```

schema

IndexToString

- Retrieves the original labels from a string indexed column
- Helps in explaining the inferred models

```
from pyspark.ml.feature import IndexToString

age_rev_indexer = IndexToString(inputCol=age_indexer.getOutputCol(),
outputCol='original_age')

df_orig_age =age_rev_indexer.transform(df_age_idx)
```

No training, simply back-transformation

age	l	indexed_age	originalAge
young	ɔ	1.0	young
young	ɔ	1.0	young
middle	s	2.0	middle
senior	s	0.0	senior
senior	s	0.0	senior
senior	ɔ	0.0	senior
middle	s	2.0	middle
young	ɔ	1.0	young
young	s	1.0	young
senior	s	0.0	senior
young	s	1.0	young
middle	s	2.0	middle
middle	s	2.0	middle
senior	ɔ	0.0	senior

Vector assembler/slicer

- Assembler
  - Combines a list of columns C1,…, Cn into a single column of vectors obtained by concatenating values/vectors in C_i
- Slicer
  - Restricts to a set of columns, indicated by their coordinates

Vector assembler

indexed_age	indexed_income
1.0	1.0
1.0	1.0
2.0	1.0
0.0	0.0
0.0	2.0
0.0	2.0
2.0	2.0
1.0	0.0
1.0	2.0
0.0	0.0
1.0	0.0
2.0	0.0
2.0	1.0
0.0	0.0

```
from pyspark.ml.feature import VectorAssembler

cols = ['indexed_age','indexed_income']
vec_assembler = VectorAssembler(inputCols= cols, \
outputCol= 'ageIncomeVec')

df_age_income_vec = vec_assembler.\
        transform(df_age_income_idx)
```

ageIncomeVec
[1.0,1.0]
[1.0,1.0]
[2.0,1.0]
(2,[],[])
[0.0,2.0]
[0.0,2.0]
[2.0,2.0]
[1.0,0.0]
[1.0,2.0]
(2,[],[])
[1.0,0.0]
[2.0,0.0]
[2.0,1.0]
(2,[],[])

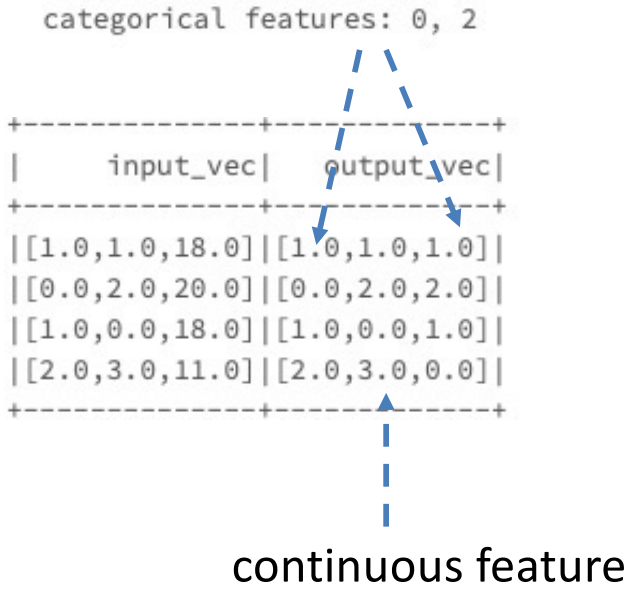
Vector Indexer

- Discriminate categorical from continuous features in a vector
- Index categorical features using 0-based indexes
- Input: col: Vector, maxCategories: int
- Set the maxCategories parameter
- If # d-values( ) <= maxCategories
  - then the feature is categorical
  - Otherwise, the feature is continuous

Vector Indexer Illustrated

```
+--------------+
|     input_vec|
+--------------+
|[1.0,1.0,18.0]|
|[0.0,2.0,20.0]|
|[1.0,0.0,18.0]|
|[2.0,3.0,11.0]|
+--------------+
```

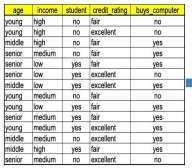
```
from pyspark.ml.feature import VectorIndexer

vecIndexer = VectorIndexer(inputCol='ageIncomeVec',\
        outputCol='indexed_ageIncomeVec',\
        maxCategories=3)

df_age_income_vec_idx =
        vecIndexer.fit(df_age_income_vec).\
transform(df_age_income_vec)
```



Pipelines

- Inspired by SickitLearn pipeline
- Used for combining several algorithms into one workflow
  - setStages(Array[<: PipelineStage])
- Each algorithm is either a transformer or an estimator
- P = op1, op2, …, op*n*
- Invoking fit() for P
  - Sequential processing of op*i s*
  - if op*i* is an estimator then invoke fit() for op*i*
  - Else // op*i* is a transformer
  - invoke transform()

Pipelines illustrated



data.csv

String Indexer → Vector Assembler → Vector Indexer

Pipeline

fit()

String Indexer Model | Vector Assembler | Vector Indexer Model

Pipeline Model

transform()

Transformer | Estimator

Legend

```
 |-- features: vector (nullable = true)
 |-- indexed_label: double (nullable = false)

+-----------------+-------------+
|         features|indexed_label|
+-----------------+-------------+
|[1.0,1.0,0.0,0.0]|          1.0|
|[1.0,1.0,0.0,1.0]|          1.0|
|[2.0,1.0,0.0,0.0]|          0.0|
|        (4,[],[])|          0.0|
|[0.0,2.0,1.0,0.0]|          0.0|
|[0.0,2.0,1.0,1.0]|          1.0|
|[2.0,2.0,1.0,1.0]|          0.0|
|    (4,[0],[1.0])|          1.0|
|[1.0,2.0,1.0,0.0]|          0.0|
|    (4,[2],[1.0])|          0.0|
|[1.0,0.0,1.0,1.0]|          0.0|
|[2.0,0.0,0.0,1.0]|          0.0|
|[2.0,1.0,1.0,0.0]|          0.0|
|    (4,[3],[1.0])|          1.0|
+-----------------+-------------+
```

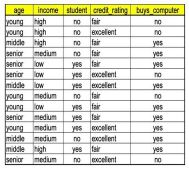
Pipelines illustrated

```
label = 'label'
features_col = data.columns
features_col.remove(label)

prefix = 'indexed_'


label_string_indexer = StringIndexer(inputCol=label, outputCol=prefix+label)

features_str_col = list(map(lambda c:prefix+c, features_col))
features_string_indexer = StringIndexer(inputCols=features_col,outputCols=features_str_col)
```

```
vec_assembler = VectorAssembler(inputCols= features_string_indexer.getOutputCols(),\
        outputCol= 'vector')

vec_indexer = VectorIndexer(inputCol='vector',outputCol='features', maxCategories=3)
```

Pipelines illustrated

```
stages = [label_string_indexer,features_string_indexer,vec_assembler,vec_indexer]

from pyspark.ml import Pipeline

pipeline = Pipeline(stages = stages)
train_data = pipeline.fit(data).transform(data)
train_data.select("features","indexed_label").show()
```



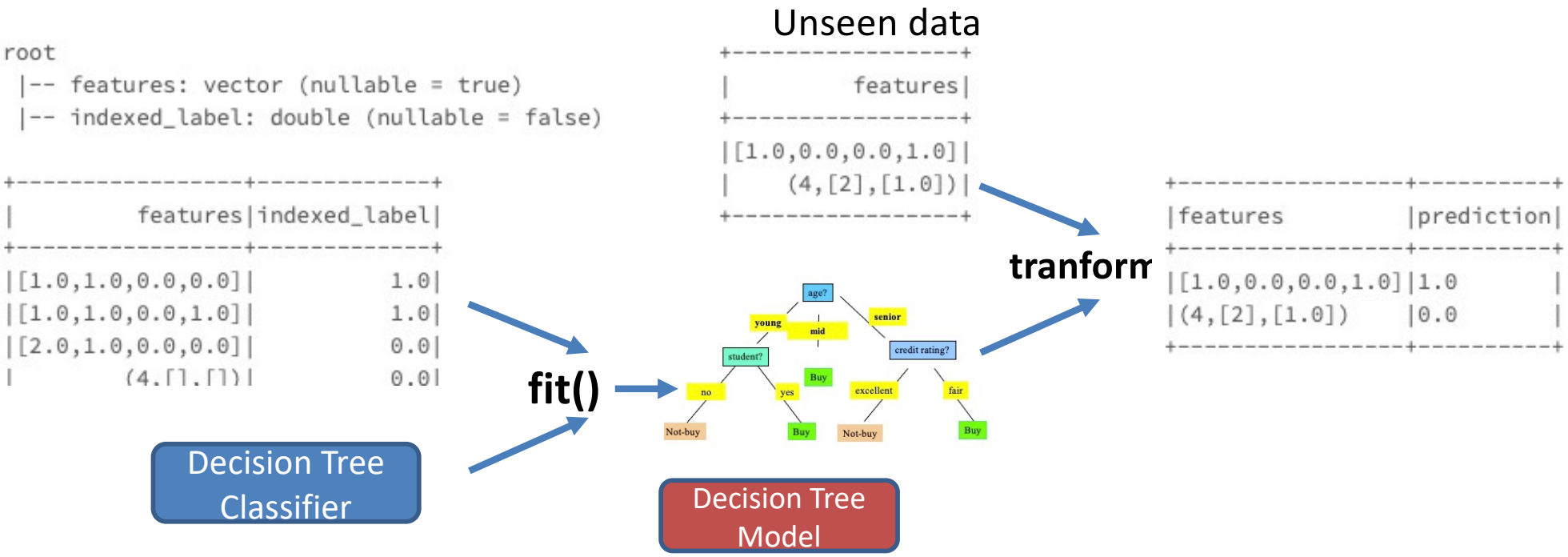
age	income	student	credit_rating	buys_computer
young	high	no	fair	no
young	high	no	excellent	no
middle	high	no	fair	yes
senior	medium	no	fair	yes
senior	low	yes	fair	yes
senior	low	yes	excellent	no
middle	low	yes	excellent	yes
young	medium	no	fair	no
young	low	yes	fair	yes
senior	medium	yes	fair	yes
young	medium	yes	excellent	yes
middle	medium	no	excellent	yes
middle	high	yes	fair	yes
senior	medium	no	excellent	no

```
root
 |-- features: vector (nullable = true)
 |-- indexed_label: double (nullable = false)

+-----------------+-------------+
|         features|indexed_label|
+-----------------+-------------+
|[1.0,1.0,0.0,0.0]|          1.0|
|[1.0,1.0,0.0,1.0]|          1.0|
|[2.0,1.0,0.0,0.0]|          0.0|
|        (4,[],[])|          0.0|
```

Decision Tree inference

- Expects a DF with
  - label column (target variable)
  - Features column (vector of indexed values)
- Exploits existing metadata :
  - maxCategories of the indexed vector to decide how to deal with features
  - Two kinds of conditions
    - Categorical features -> value equality
    - Continuous features -> interval comparison
- Multi-class/multi-label
- The inferred tree is binary, used for prediction

Decision Tree inference illustrated



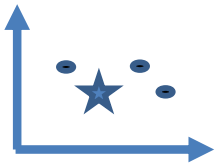
Decision Tree inference illustrated

```
from pyspark.ml.classification import DecisionTreeClassificationModel,
DecisionTreeClassifier

dt = DecisionTreeClassifier(featuresCol="features", labelCol= "indexed_label")
dtModel = dt.fit(train_data)
```

```
If (feature 0 in {2.0})
 Predict: 0.0
Else (feature 0 not in {2.0})
 If (feature 2 in {1.0})
  If (feature 3 in {0.0})
   Predict: 0.0
  Else (feature 3 not in {0.0})
   If (feature 0 in {1.0})
    Predict: 0.0
   Else (feature 0 not in {1.0})
    Predict: 1.0
 Else (feature 2 not in {1.0})
  If (feature 0 in {0.0})
   If (feature 3 in {0.0})
    Predict: 0.0
   Else (feature 3 not in {0.0})
    Predict: 1.0
  Else (feature 0 not in {0.0})
   Predict: 1.0
```

```
DecisionTreeClassificationModel:
uid=DecisionTreeClassifier_5c99afcc
20f4, depth=4, numNodes=13,
numClasses=2, numFeatures=4
```

Model Selection and Tuning



- To derive the best model:
  - experiment several hyper-parameters
  - split data in several manners
- Grid Search class
  - trying different combinations of pre-set parameters
- CrossValidator class
  - Build different (train, test) candidates
- Use default evaluation metrics (e.g. areaUnderROC for classif)
- Extract the best model w.r.t. the defined metrics

Model Selection and Tuning

A DT classifier

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

dt_paramGrid = ParamGridBuilder()\
        .addGrid(dt.maxBins, [40,42])\
                .addGrid(dt.minInstancesPerNode, [10,100]) \
                .build()
```

A parameter of the model

Values to experiment

The Grid contains 2 X 2 = 4 configuration to run

Model Selection and Tuning

```
#create k folds with k=5.
cv = CrossValidator(estimator=dt, \
estimatorParamMaps=dt_paramGrid, \
evaluator=evaluatorPR, \Size of the split
numFolds=5, \
parallelism=2)
```

test Train

The Grid contains 2 X 2 = 4 configuration to run
There are 5 folds

```
cvModel = cv.fit(train_data)
```

20 DT are inferred

```
bestModel = cvModel.bestModel
```

Closing remarks

- Pros
  - Efficiency thanks to the distributed evaluation
  - Static typing facilitates examining and reusing the pipeline
  - Metadata collection
- Cons
  - No fine-grained control on how to define categorical features
  - Impute of missing values limited to number (not possible for textual values)
- Possible extensions
  - Impute text values by using advanced NLP techniques (word2vec,…)
  - Parallel exploration of the search space to identify sub-set of relevant features
  - AutoML: automatic feature extraction, model selection and hyper-parameter search

Use-case of the lab session

- Goal: write a Spark ML pipeline for inferring a regression tree
- Data: any relevant table, suggest using the car price table
- Requirement:
  - Engineer/encode the features from input data
  - Deal with null values, outliers
- Compare two strategies:
  - Use every column, choice of categorical based on default values
  - Select a sub-set of columns:
    - make an informed decision about categorical features
    - Discard columns with a high ratio of null values OR impute missing values
- Model Selection:
  - 3-fold cross validation, output importance vector

Map Reduce on Spark

Big Data Timeline

Big Data Phase 1 – Structured Content	Big Data Phase 2 – Web Based Unstructured Content	Big Data Phase 3 – Mobile and Sensor Based Content
Period: 1970-2000	Period: 2000 – 2010	Period: 2010 - Present
• RDBMS & data warehousing • Extract Transfer Load • Online Analytical Processing • Dashboards & scorecards • Data mining & statistical analysis	• Information retrieval and extraction • Opinion mining • Question answering • Web analytics and web intelligence • Social media analytics • Social network analysis • Spatial-temporal analysis	• Location-aware analysis • Person-centred analysis • Context-relevant analysis • Mobile visualization • Human-Computer-Interaction

Address the volume challenge

- Hardware
  - Cluster of *commodity* machines
  - Fault tolerance guarantee
  - Resource management
- Software
  - Shared nothing massively parallel computing
  - Several models: Map-reduce, BSP
  - Various systems
    - Compute: Hadoop MR, Spark, Flink, Dask
    - Storage: HDFS, Hbase

Technology Stack

Source: Flink

Applications

Hive

Cascading

Mahout

Pig

Crunch

Data processing engines

MapReduce 

Flink 

Spark 

Storm 

Tez 

App and resource management

Yarn

Mesos

Storage, streams

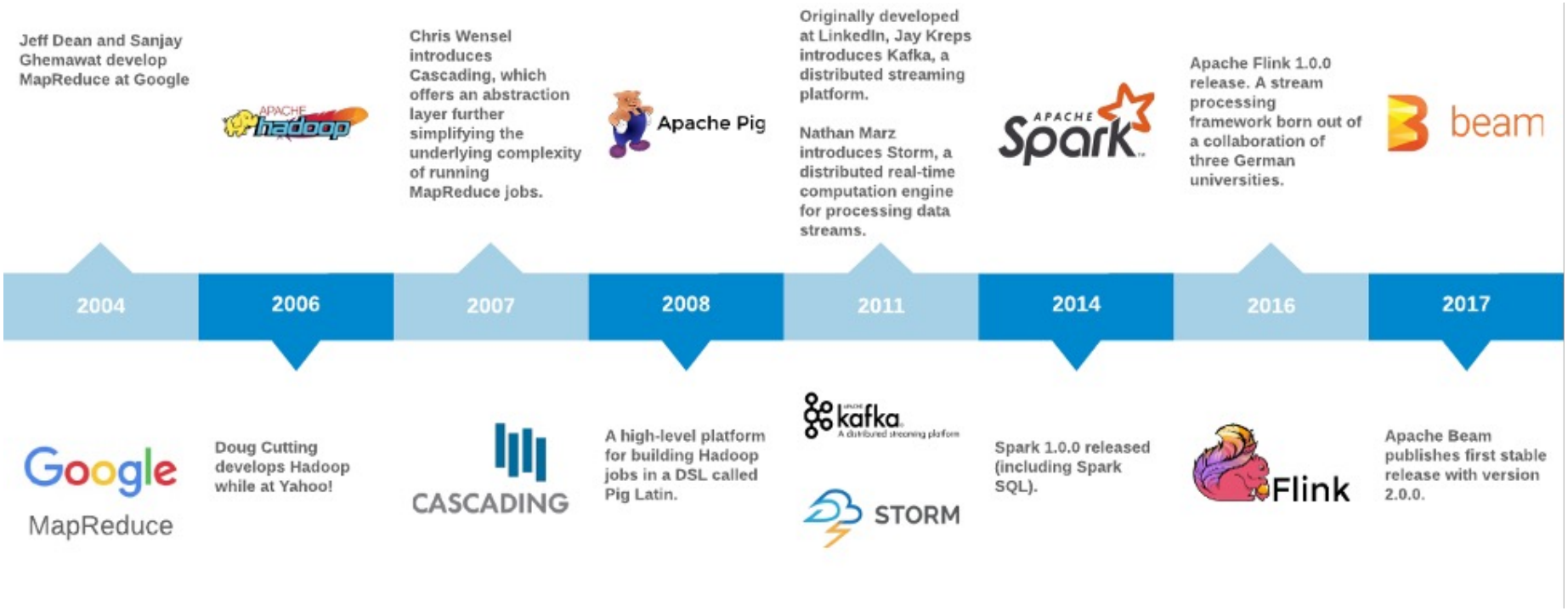
HDFS

HBase

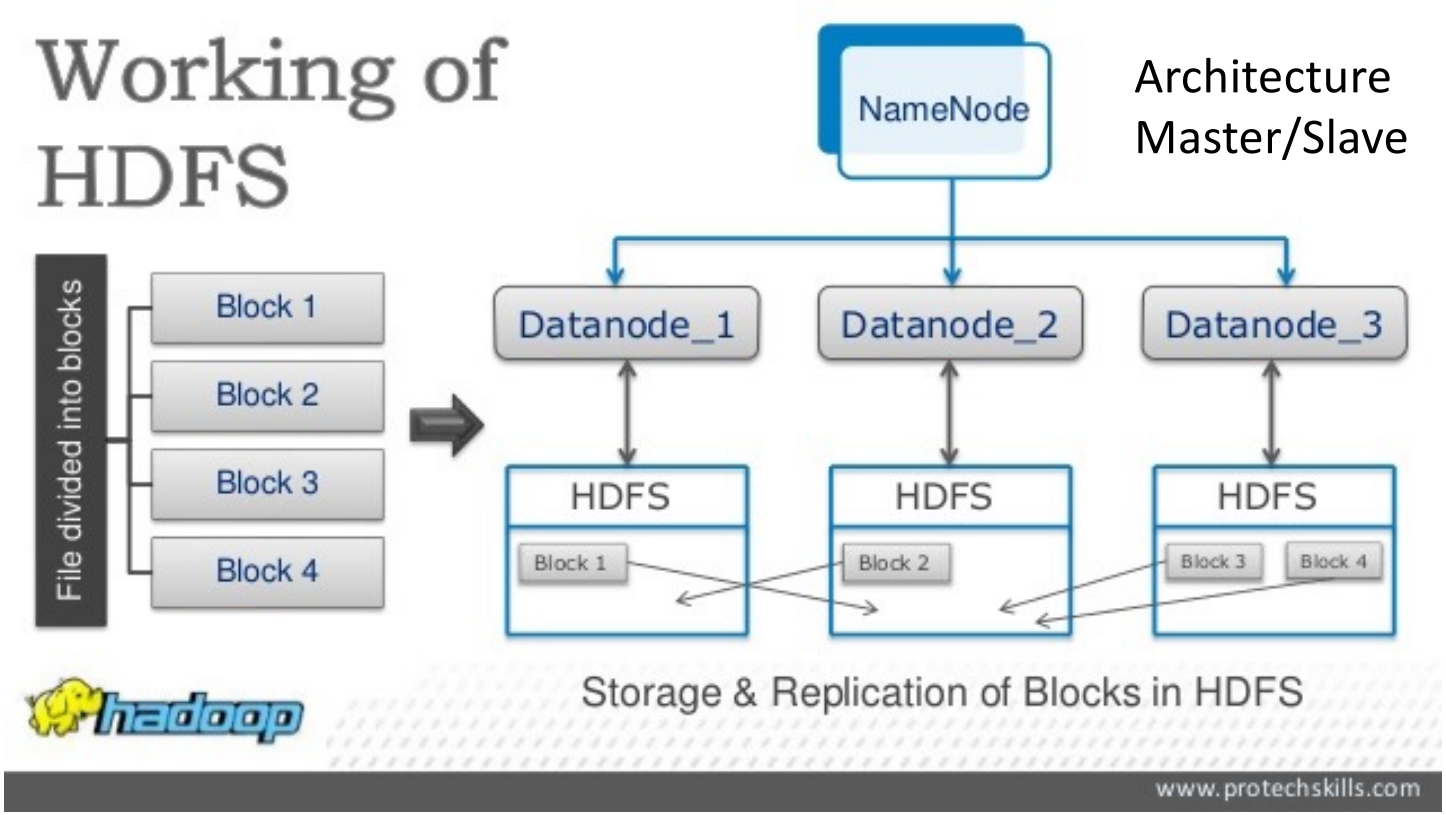
Kafka

...

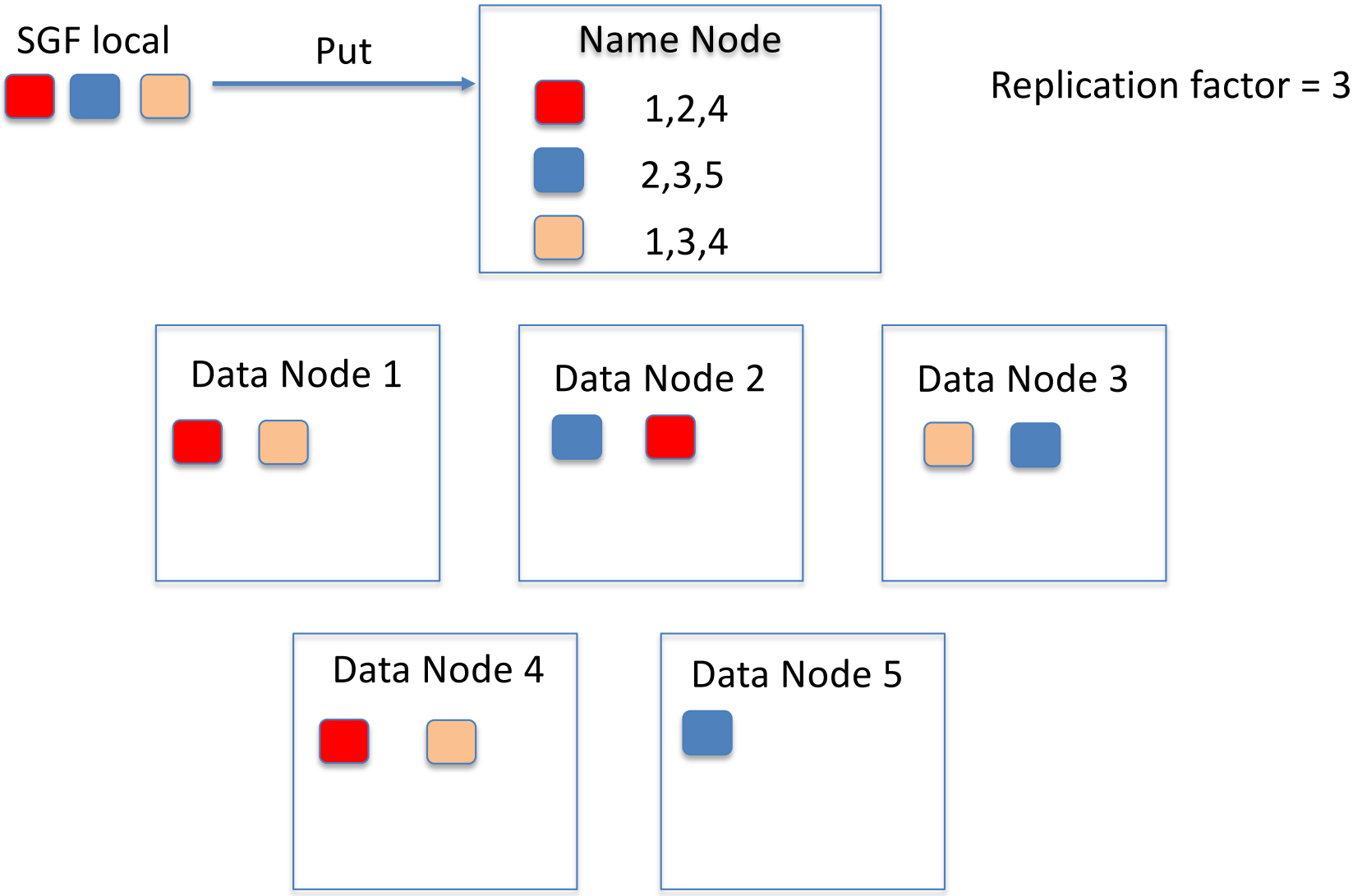
Technology timeline



A typical storage system: HDFS



Replication in HDFS

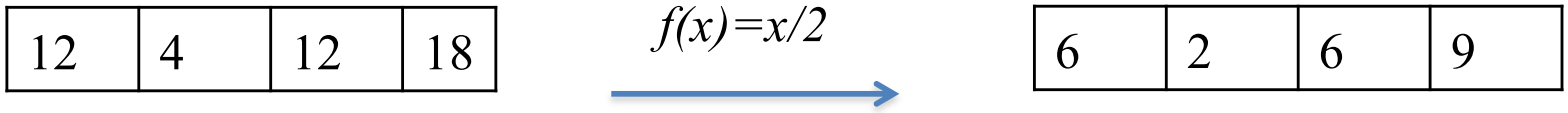


A typical computation model: Map-Reduce

- calcul massivement parallèle, mode *shared nothing*
- Fonctions d'ordre supérieur (prog. Fonctionnelle)

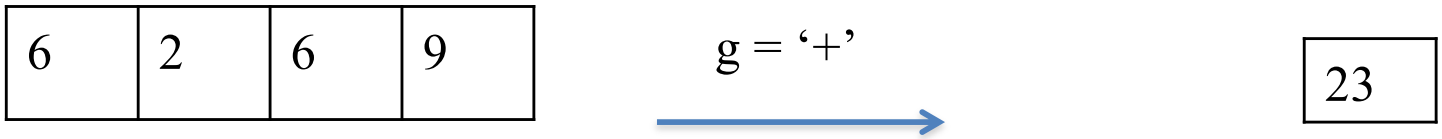
– *C collection d'éléments*

– *Map (f:* T=>U), *f* unaire : appliquer *f* à chaque élément de *C*



la dimension de C est preservée le type en entrée peut changer

– *Reduce (g*: (T,T)=>T), *g* binaire



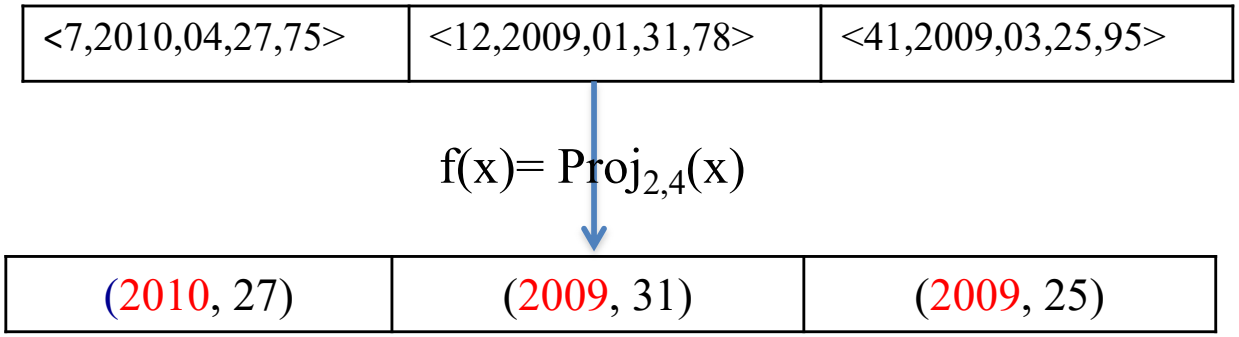
réduit la dimension de *n* à 1 le type en sortie identique à celui en entrée

Adaptation pour le big data

- Type de données
  - logs de connections, transactions, interactions utilisateurs, texte, images
  - structure homogène (schéma implicite)
- Type de traitements
  - Aggrégations (count, min, max, avg)
  - Autres traitements (indexation, parcours graphes, Machine Learning)

Map Reduce pour le big data

- Les données en entrée sont des **nuplets** : identifier attribut de groupement (appelé **clé**)
- *Map (f:* T=>(*k*,U)), *f* unaire
  - produire une paire (clé, val) pour chaque val de *C*



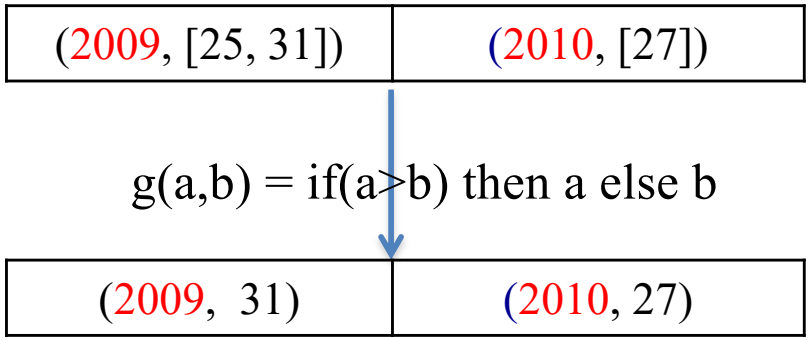
  - Regrouper les paires ayant la même clé pour obtenir (clé, [list-val])

(2009, [25, 31])	(2010, [27])

Map Reduce pour le big data

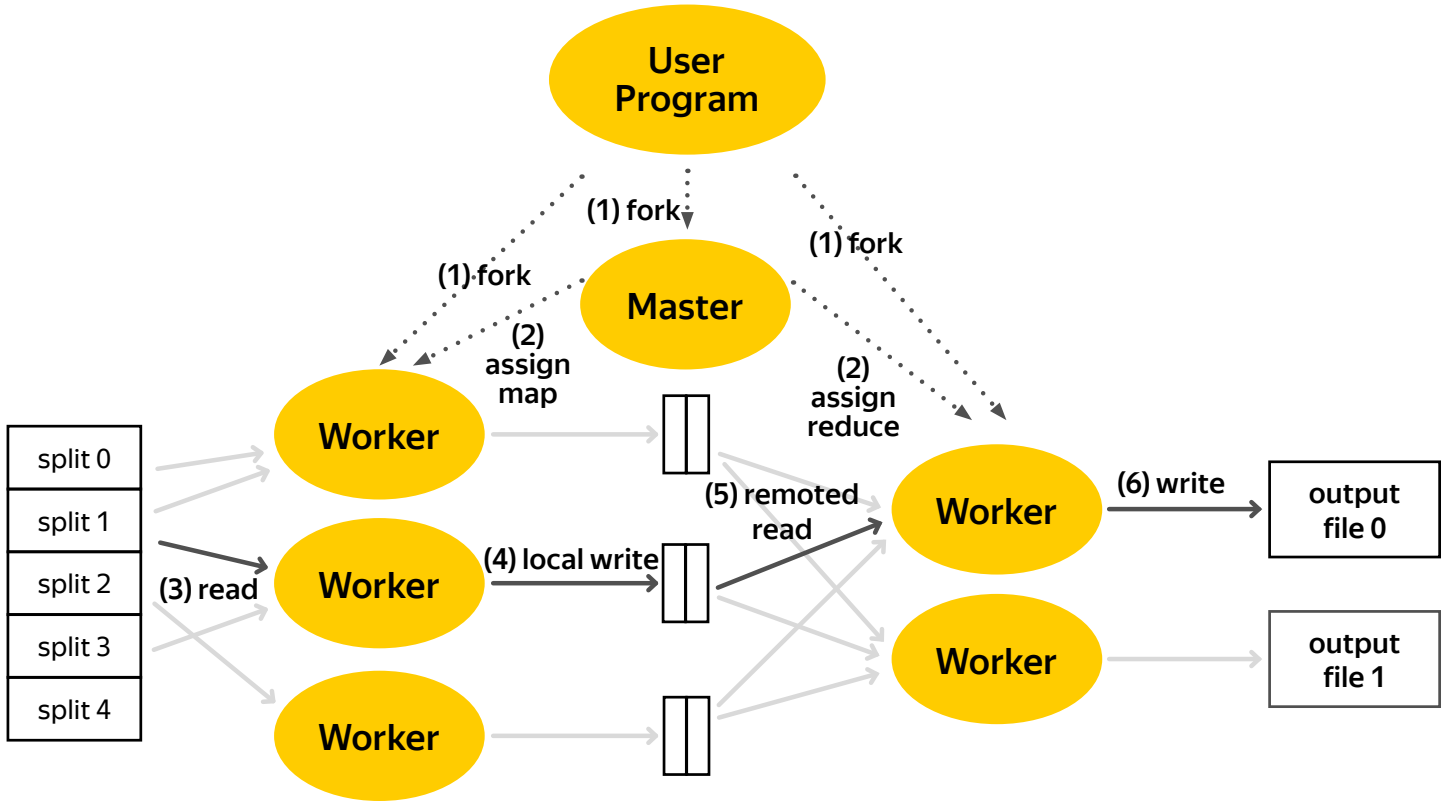
– *Reduce (g*: (T,T)=>T), *g* binaire

- pour chaque (clé, [list-val]) produit (clé, val) où val = g([list-val])

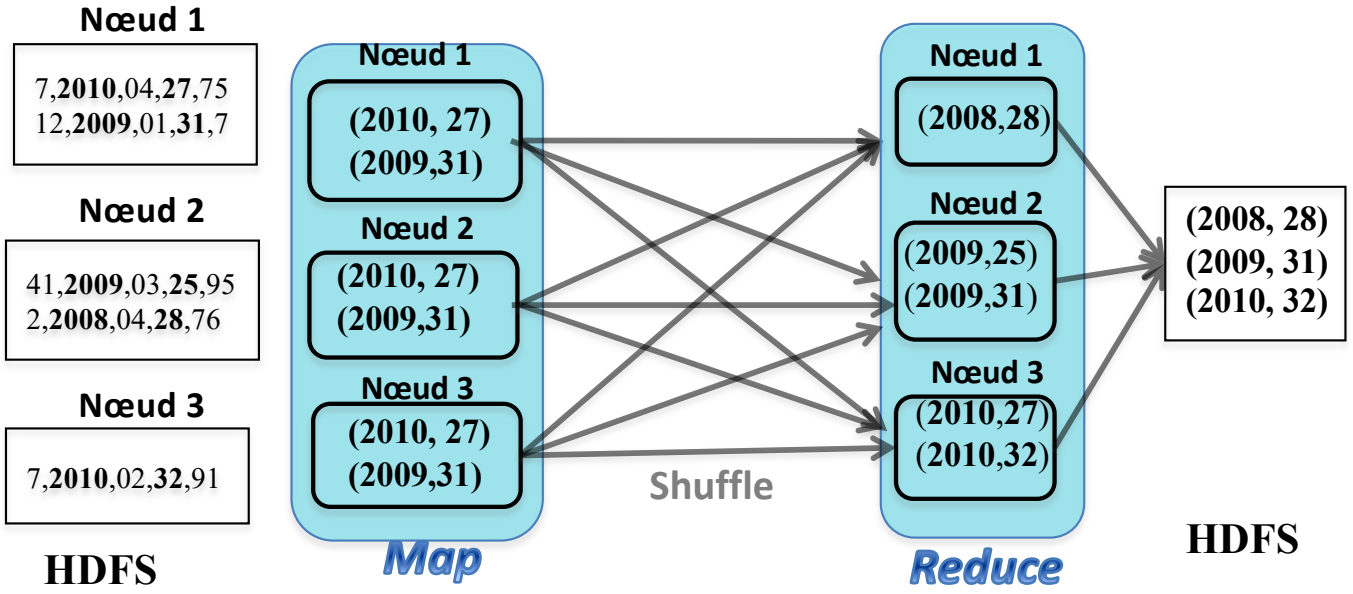


- **Important** : dans certains systemes, g doit être **associatif** car ordre de traitment des élément de *C* non présrcit

Exécution Map-Reduce

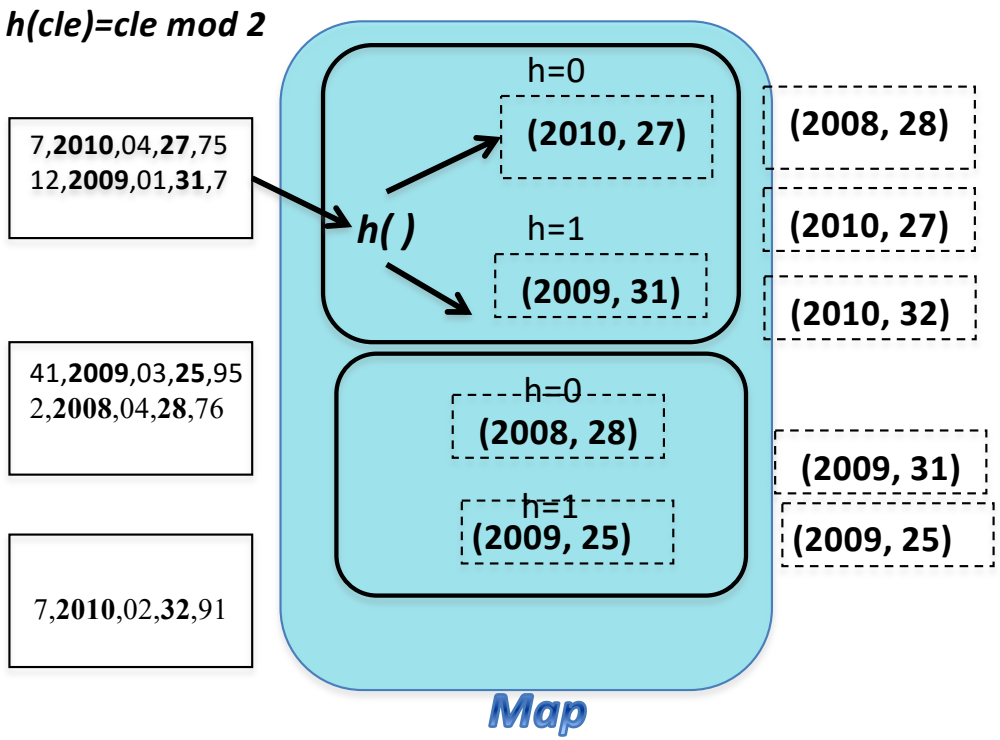


Exécution Map Reduce

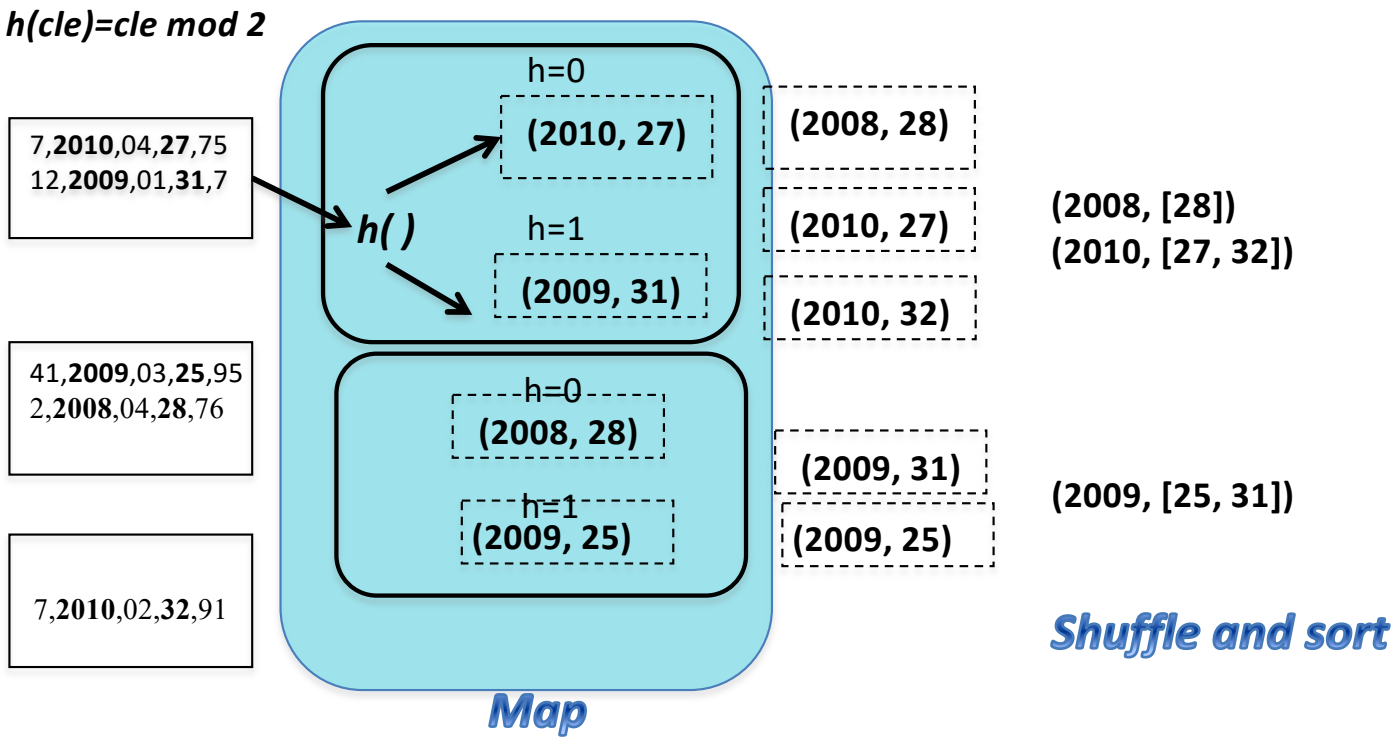


Entrée : n-uplets (station, **annee**, mois, **temp**, dept)
Résultat : select annee, Max(temp) group by annee

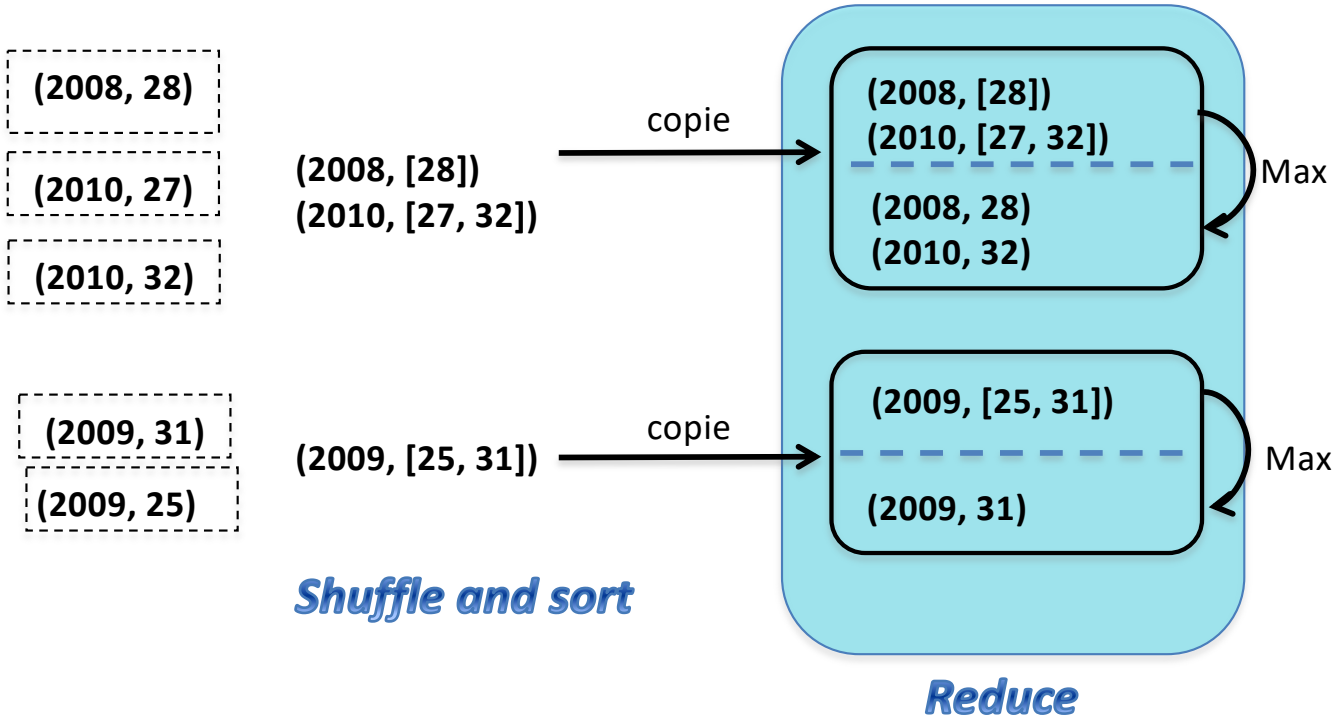
Phase Map


h(cle)=cle mod 2
7,2010,04,27,75
12,2009,01,31,7
41,2009,03,25,95
2,2008,04,28,76
7,2010,02,32,91
h( )
h=0
(2010, 27)
h=1
(2009, 31)
h=0
(2008, 28)
h=1
(2009, 25)
(2008, 28)
(2010, 27)
(2010, 32)
(2009, 31)
(2009, 25)
Map

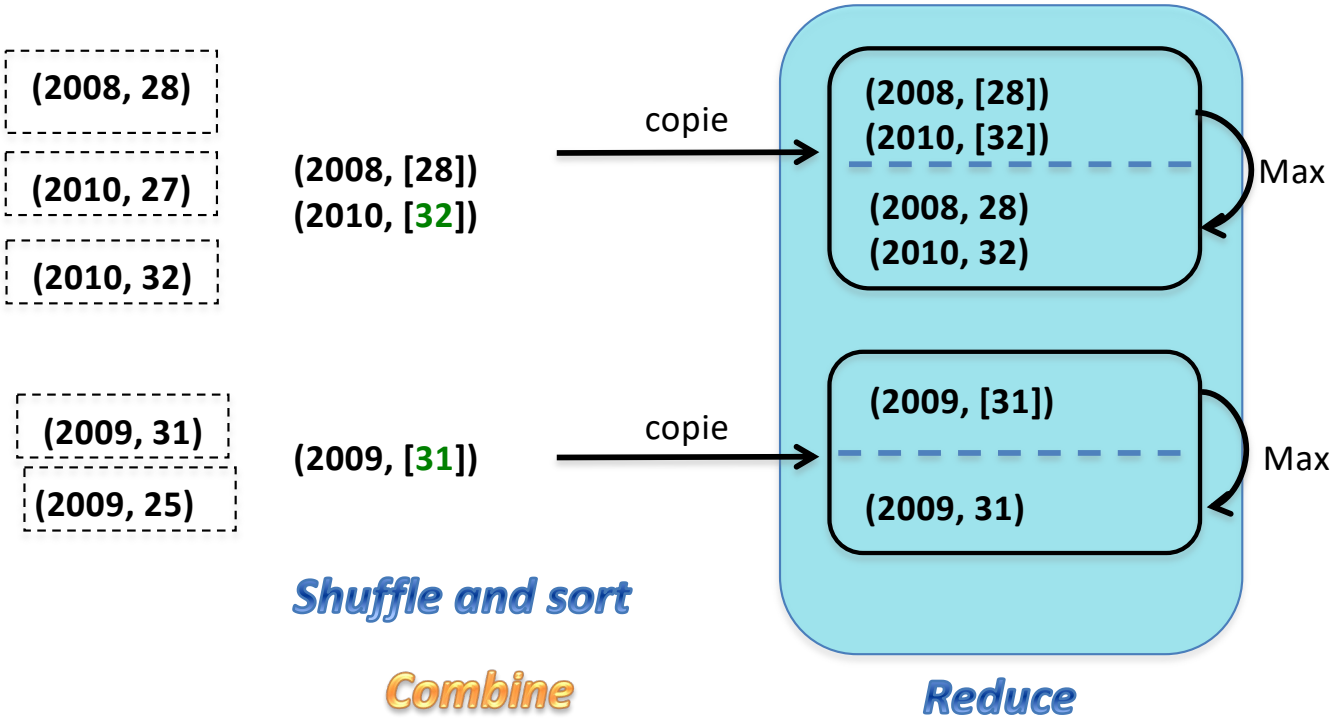
Phase Shuffle & Sort



Phase Reduce



Combine


(2008, 28)
(2010, 27)
(2010, 32)
(2009, 31)
(2009, 25)
(2008, [28])
(2010, [32])
(2009, [31])
copie
copie
(2008, [28])
(2010, [32])
(2008, 28)
(2010, 32)
(2009, [31])
(2009, 31)
Max
Max
Shuffle and sort
Combine
Reduce

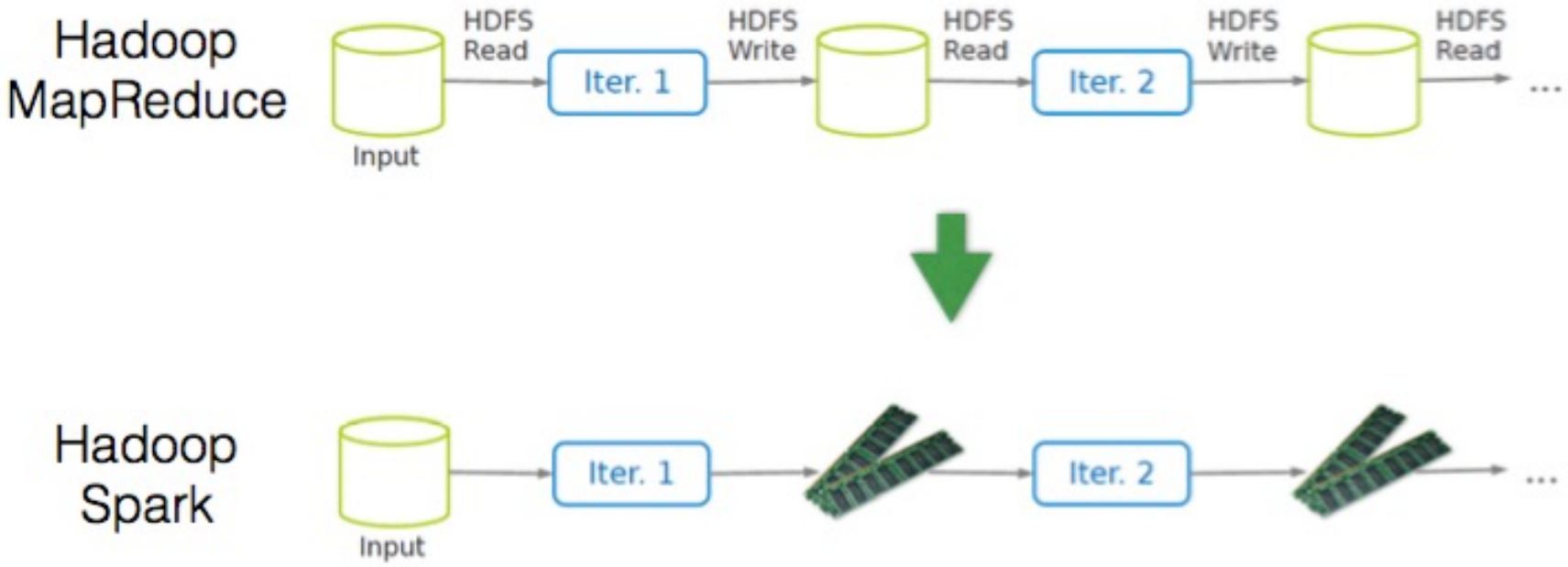
Limites de Hadoop Map Reduce

- Performances dégradées
  - Traitement complexe en plusieurs étapes
  - Raison : matérialisation résultat de chaque étape
    - avantages : reprise sur panne performante
    - inconvénients : accès fréquent au disque
  - Optimisation possible : pipelining et partage
- Inadapté aux traitements itératifs ML et analyse graphes
- Pas d'interaction avec l'utilisateur
- Alternative : Apache Spark

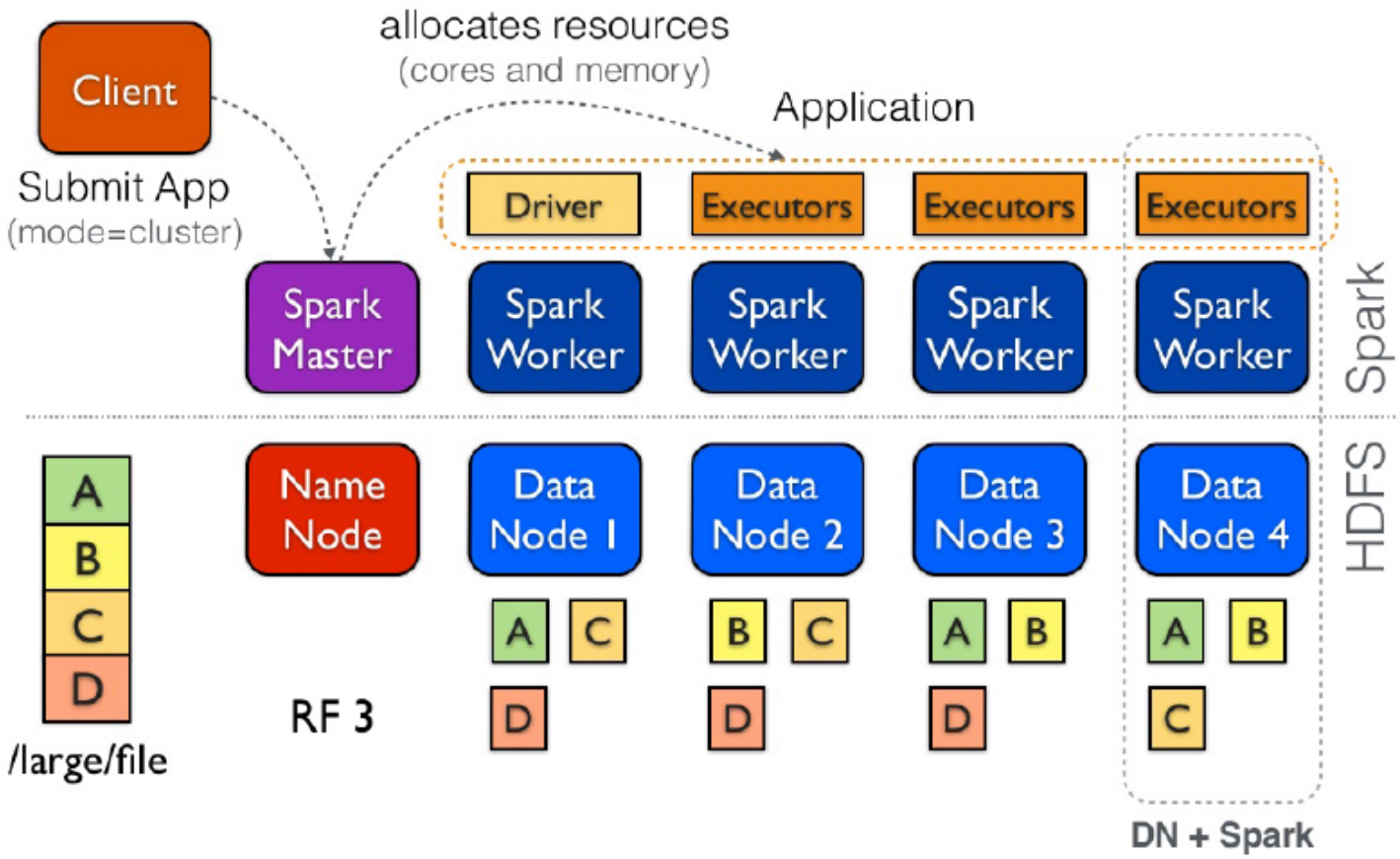
The 2022 ACM SIGMOD Systems Award goes to "Apache Spark", an innovative, widely-used, open-source, unified data processing system encompassing relational, streaming, and machine-learning workloads.

Spark vs Hadoop Map Reduce

- Résoudre les limitations de Map Reduce
  - *Persistance* en mémoire centrale
  - *Lazy evaluation*



Architecture type Spark



Attrait de Spark

- Framework assez complet, préparation et l'analyse des données massives
  - API de base, SQL pour CSV et JSON
  - ML pipeline, exécution distribuée
  - Graphes : modèle BSP
- Système interactif, utilisé en production
- Plusieurs langages hôtes
  - Scala (natif), Java, Python et R
- Plusieurs connecteurs
  - NoSQL systems (Couchbase, MongoDB), BI tools (Tableau, Superset), …

Spark main ingredients

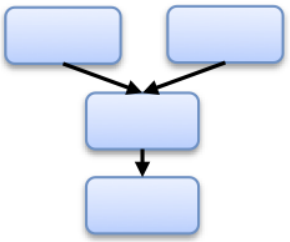
- Resilient Distributed Dataset (RDD)
  - collection logique de données distribuées sur plusieurs nœuds
  - traitement gros granule (pas de modification possible)
  - matérialisation à la demande (*lazy evaluation*)
  - tolérance aux pannes par réexécution d'une chaine de traitement
- Avantages:
  - pas de problème de cohérence (lecture seule)
  - gestion de pannes simplifiée : rejouer le *lineage*
  - gestion de charge simplifiée par copies
- Inconvénients:
  - inadapté aux transactions et aux flux de données

Deux types d'opérations

- Transformations :
  - opérations qui **définissent** une collection à partir d'une ou plusieurs autres collections
  - **unaires** : map, reduceByKey, groupByKey, ...
  - **binaires** : union, subtract, join, ...
- Actions:
  - opérations qui **évaluent** des compositions de transformations (collections) et **génèrent des données dans le modèle du langage hôte** (Java, Scala, Python)

Modèle d'exécution

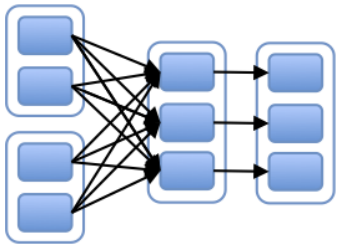
RDD Objects



```
rdd1.join(rdd2)
.groupBy(...)
.filter(...)
```

Build the operator DAG

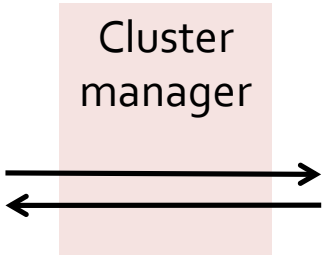
DAG Scheduler



Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

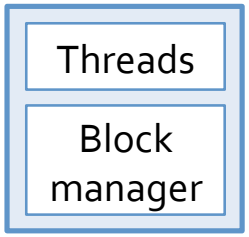
Task Scheduler



Launch tasks via Master

Retry failed and strag-gler tasks

Worker

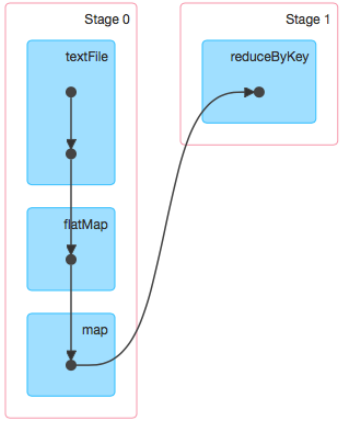


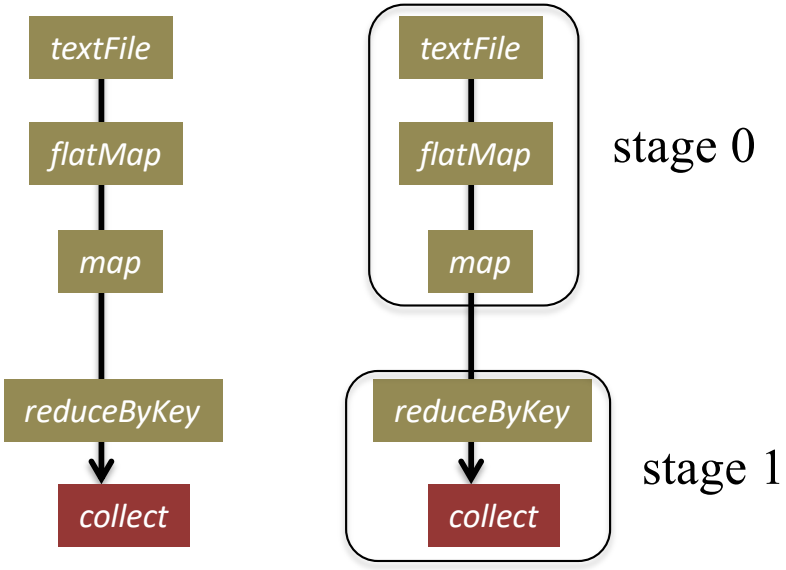
Execute tasks

Store and serve blocks

Wordcount en RDD

```
val lines = sc.textFile(filename)
val words = lines.flatMap(x=>x.split(" "))
val pairs = words.map(x=>(x,1))
val counts = pairs.reduceByKey(_+_)
val results = counts.collect
```





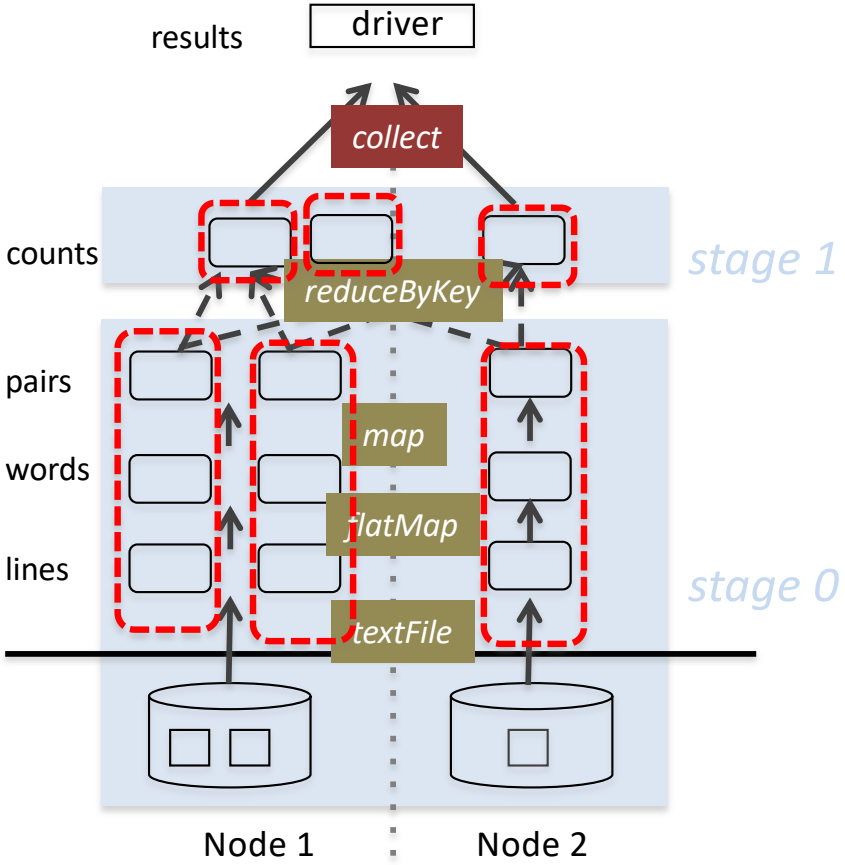
DAG d'opérateurs DAG de Stages

action transformation

Wordcount en RDD

Attention au sens de lecture

```
lines = sc.textFile(filename)
words = lines.flatMap(lambda x:
x.split(" "))
pairs = words.map(lambda x: (x,1))
counts = pairs.reduceByKey(add)
results = counts.collect()
```



task

partition d'une RDD

action

transformation

Quelques opérateurs RDD

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

map

Map (*f:*T⇒U) : RDD[T] => RDD[U]

7,2010,04,27,75
12,2009,01,31,78
…
..
..

RDD[String]

map(lambda x: x.split(","))

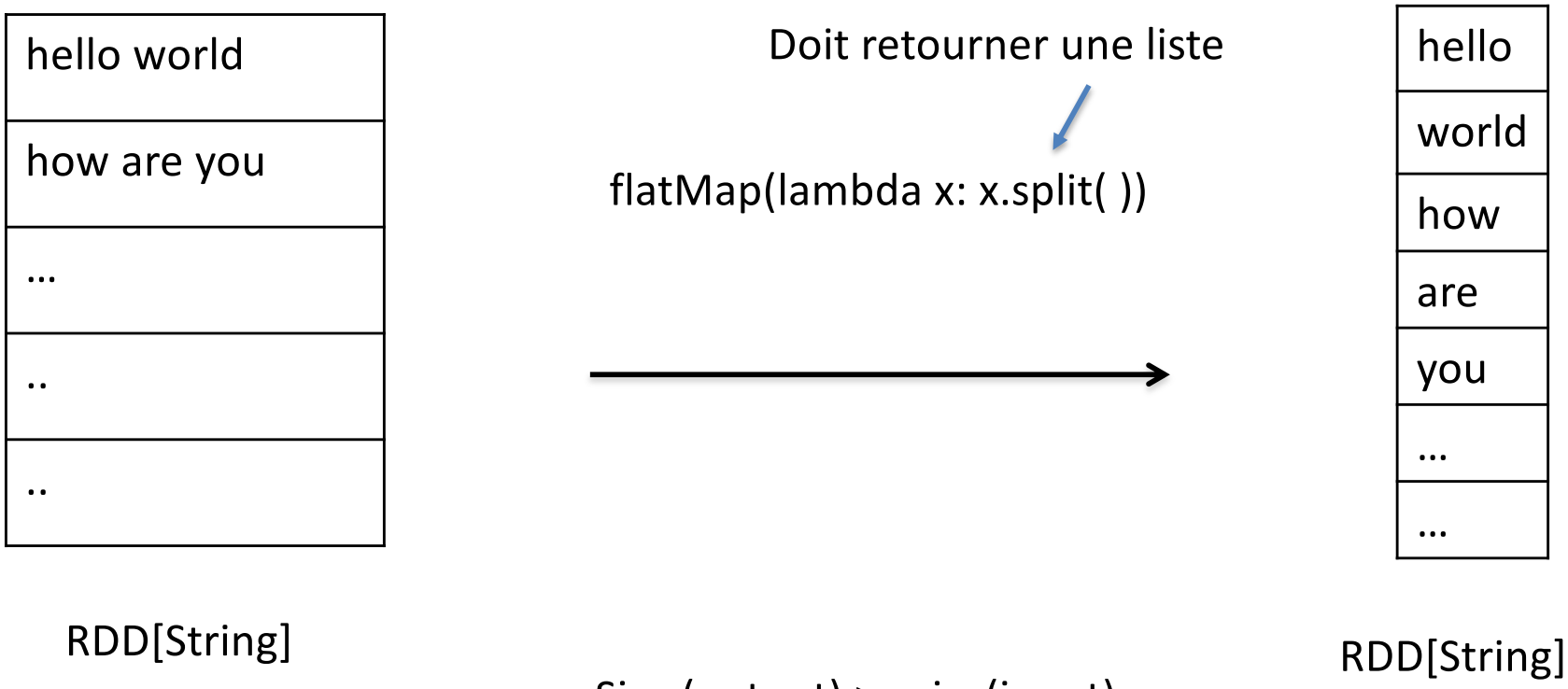
7	2012	04	27	75
12	2009	01	31	78
…				
…				
…				

RDD[Array[String]]

Size (output) = size(input)

flatMap

flatMap (*f:*T⇒Seq[U]) : RDD[T] => RDD[U]



filter

filter(*f:* T⇒Bool) : RDD[T] => RDD[T]

(2010,27)
(2009,31)
(2008,28)
(2010,32)
(2009,25)

RDD[(Int, Double)]

filter(lambda p : p[1]>30)

(2009,31)
(2010,32)

RDD[(Int, Double)]

Size (output) <= Size (input)

reduceByKey

reduceByKey(*f:* (V, V)⇒V) : RDD[(K,V)] => RDD[(K,V)]

(2010,27)
(2009,31)
(2008,28)
(2010,32)
(2009,25)

RDD[(Int, Double)]

```
def f(a,b):
 if a>b :
  return a
 else
    return b

reduceByKey(lambda a,b : f(a,b))
```

(2010,32)
(2009,31)
(2008,28)

RDD[(Int, Double)]

f commutative et associative

Size (output) = |distinct keys|

groupByKey

groupByKey() : RDD[(K,V)] => RDD[(K,Seq[V])]

(2010,27)
(2009,31)
(2008,28)
(2010,32)
(2009,25)

RDD[(Int, Double)]

groupByKey()

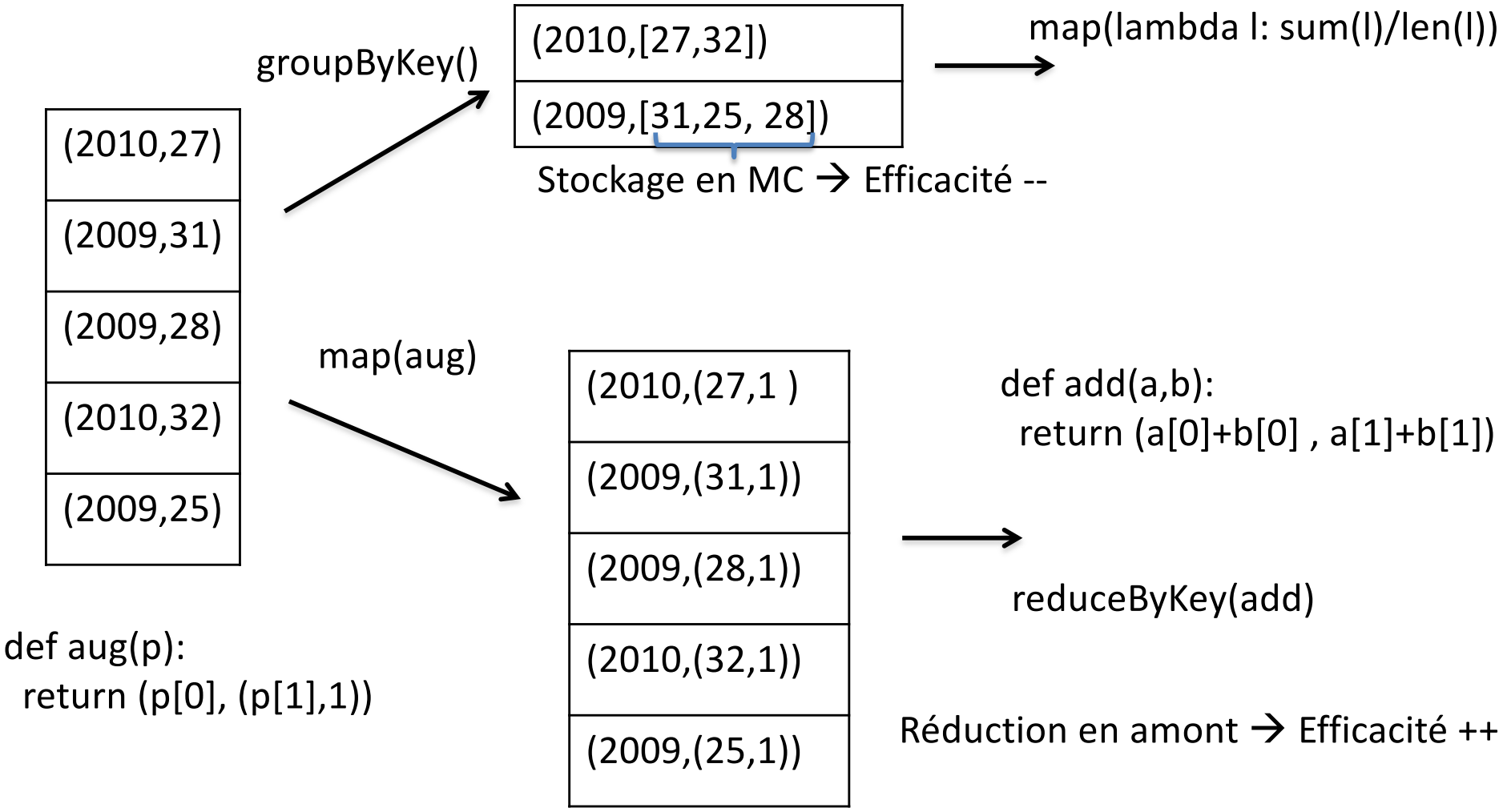
(2010,[27,32])
(2009,[31,25])
(2008,[28])

RDD[(Int, Seq[Double])]

Possibilité d'appliquer
g algébrique (pas forcément associative)

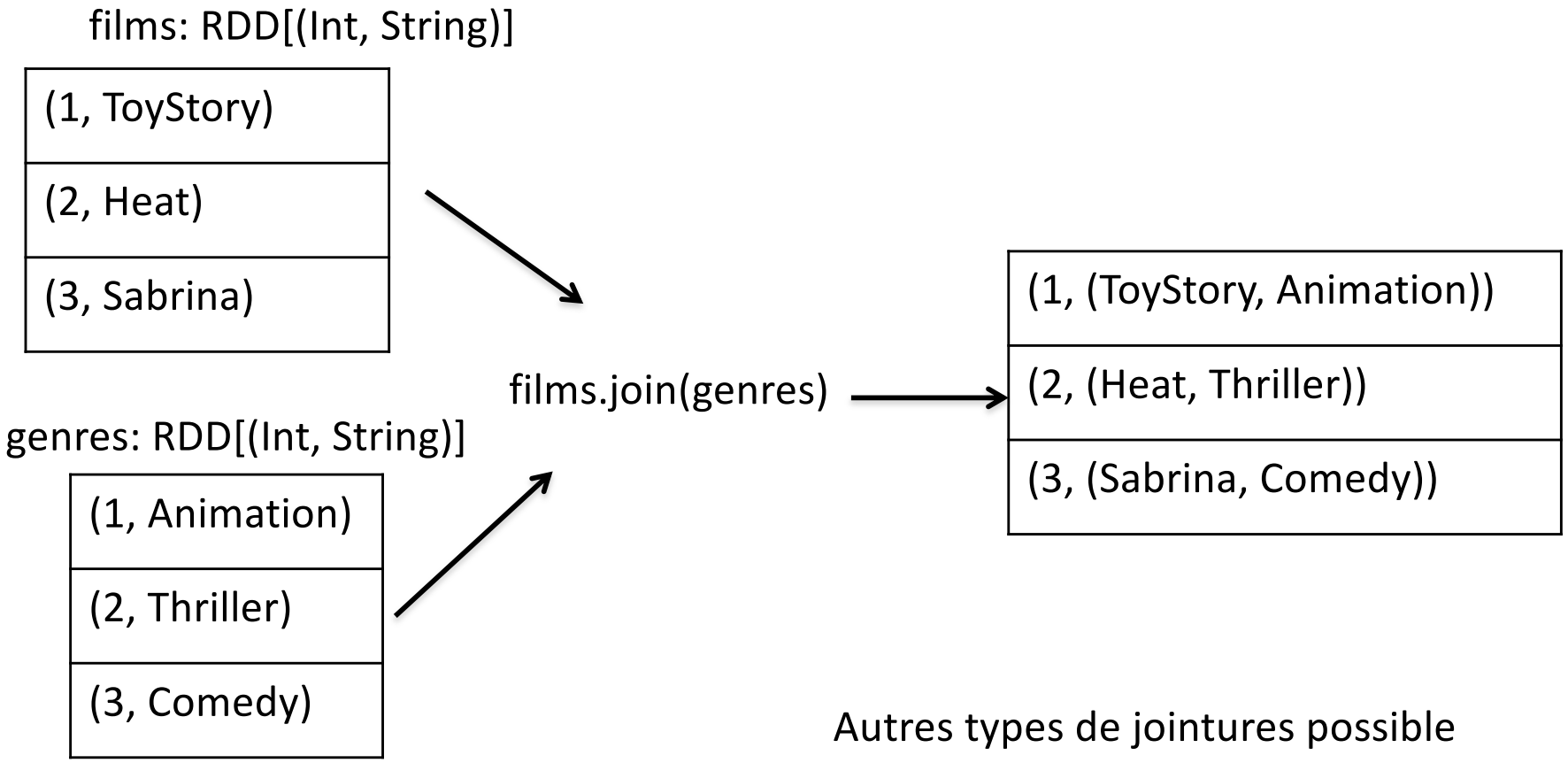
Size (output) = |distinct keys|

Calcul de la moyenne



join

join(): (RDD[(K,V)], RDD[(K,W)]) => RDD[(K,(V,W))]



Autres types de jointures possible

zipWithIndex

zipWithIndex() : rdd[T] => rdd[(T,Long)]

hello
world
how
are
you
…
…

RDD[String]

Augment une RDD avec un entier correspondant à l'indice de chaque élément

zipWithIndex()

(hello, 0)
(world, 1)
(how, 2)
(are, 3)
(you, 4)
…
…

RDD[(String,Long)]

Size (output) = Size (input)

RDD : Actions

Transformation RDD vers un objet/une valeur

Python :

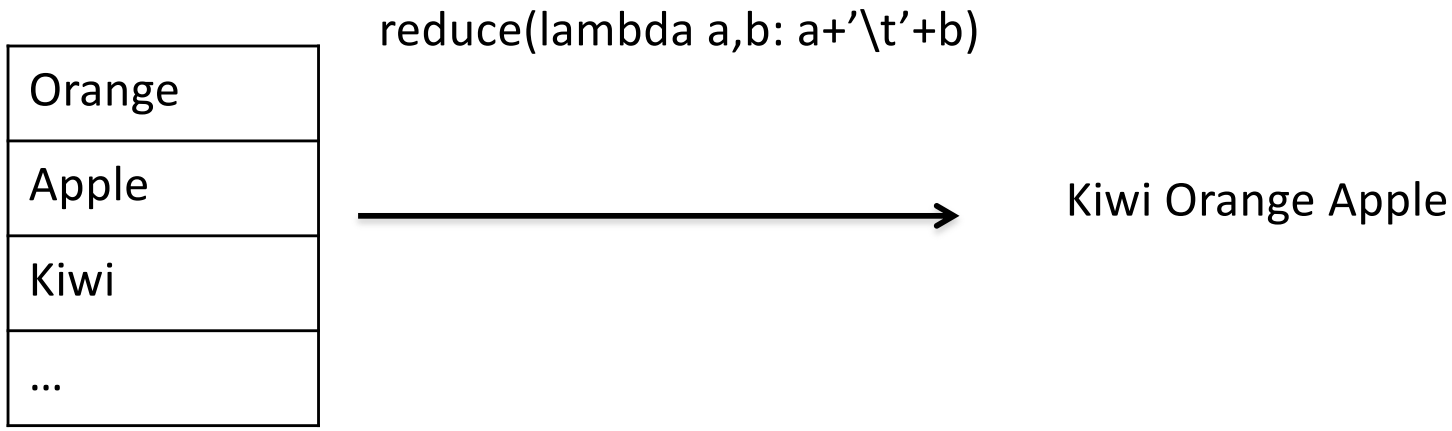
- reduce() : rdd[T] -> T
- count() : rdd[T] → Long
- countByKey() : rdd[(K,V)] → dict(K:Long)
- collect() : rdd[T] → [T]
- take(n) : rdd[T] → [T]

reduce

reduce(f : (T,T)=>T): RDD[(T,T)] => T

- Réduction de la dimension en utilisant une *User Defined Function (UDF)*
- Traitement distribué, aucun ordre prescris

  →dans Spark *f* doit être commutative et associative car aucune garantie sur l'ordre d'application



Exercice : simplified TD-IDF

d1 one fish two fish

d2 red fish blue

d3 one red bird

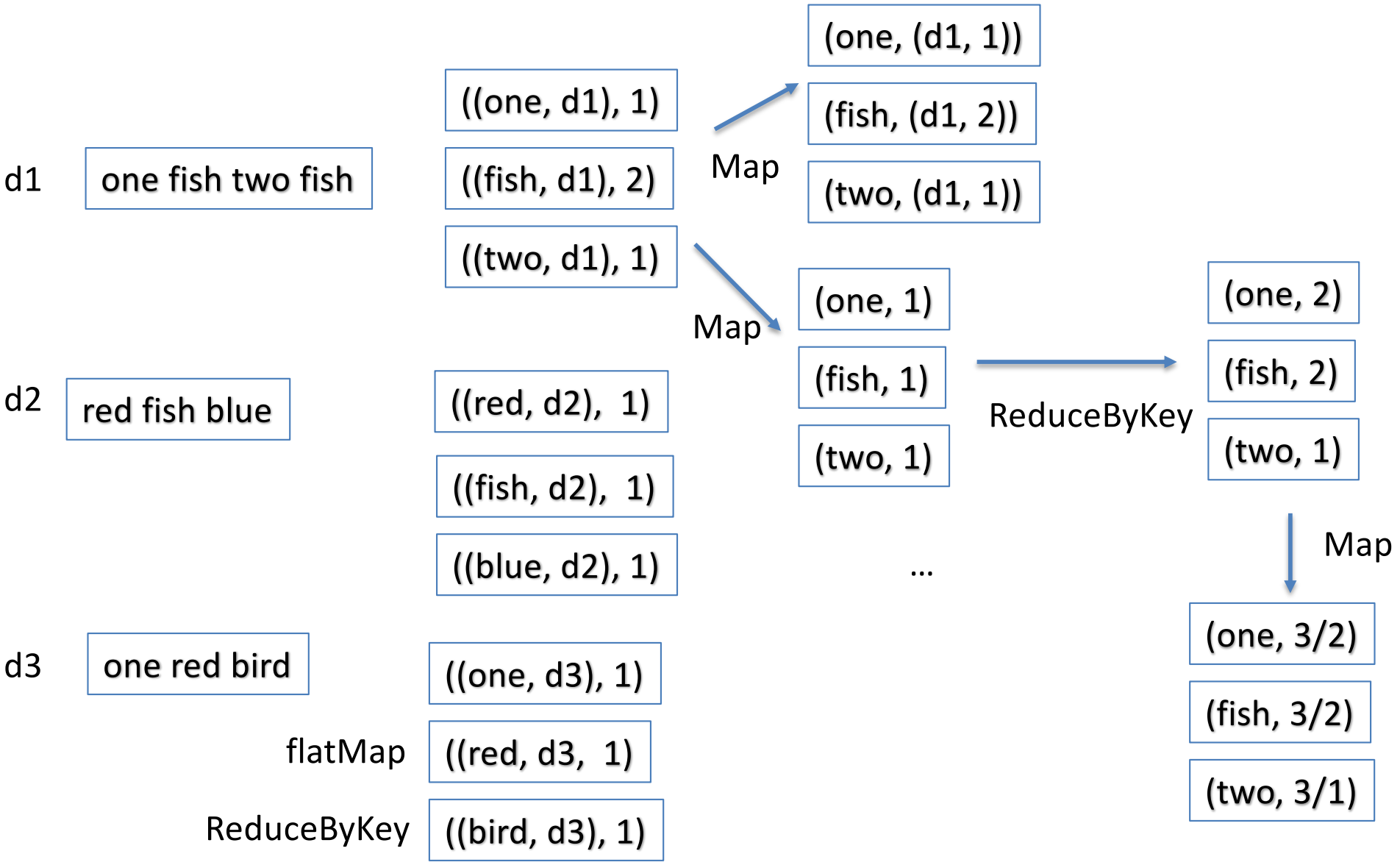
Tf*ifd(d_i,w_j)
i = 1,3
w_j in {one, fish, …}

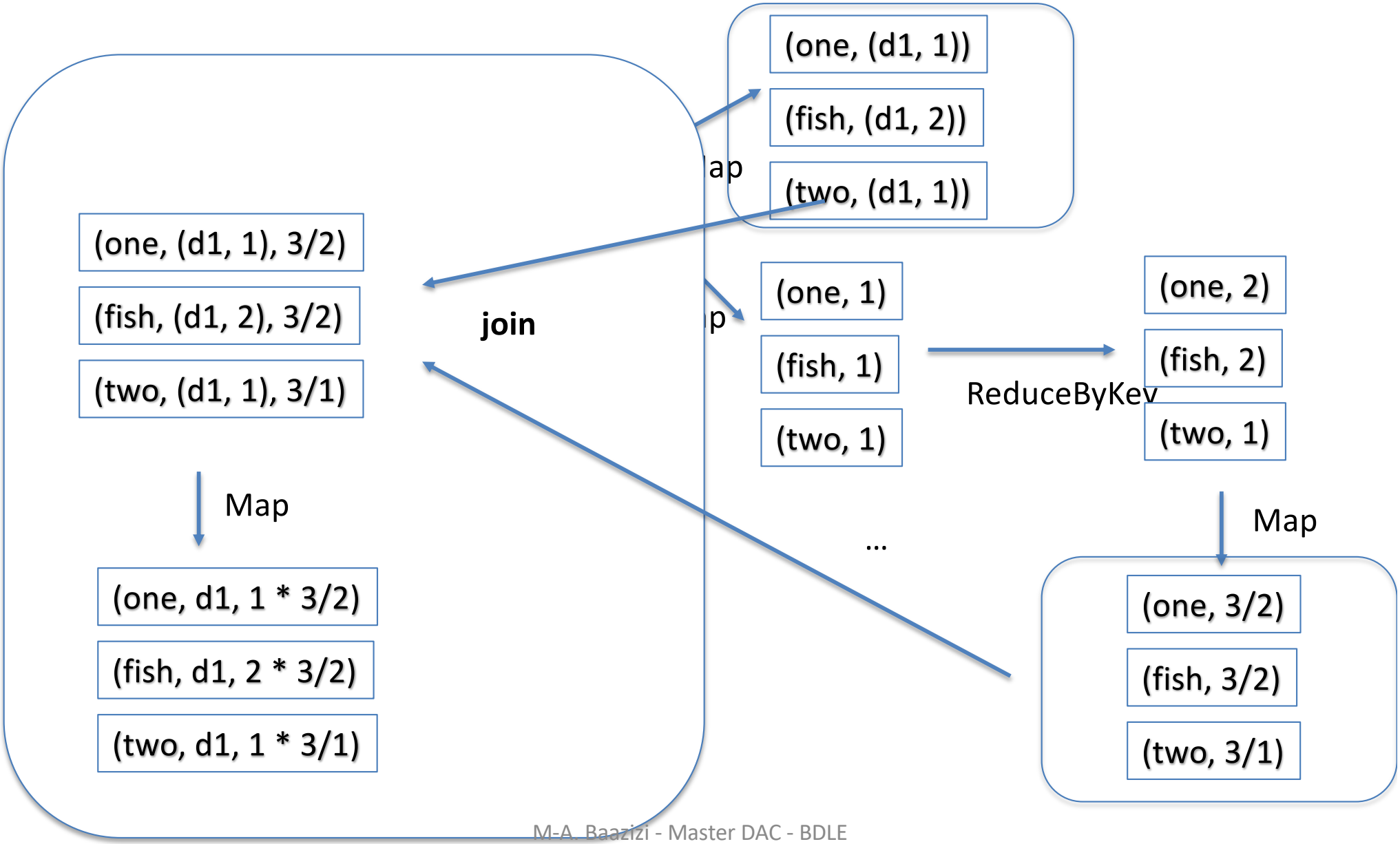
TF(d,w) = |w in d|

IDF(w) = |collection|/ |d containing w|

input

output


d1
one fish two fish
((one, d1), 1)
((fish, d1), 2)
((two, d1), 1)
Map
(one, (d1, 1))
(fish, (d1, 2))
(two, (d1, 1))
Map
(one, 1)
(fish, 1)
(two, 1)
d2
red fish blue
((red, d2), 1)
((fish, d2), 1)
((blue, d2), 1)
...
ReduceByKey
(one, 2)
(fish, 2)
(two, 1)
Map
(one, 3/2)
(fish, 3/2)
(two, 3/1)
d3
one red bird
((one, d3), 1)
flatMap
((red, d3, 1)
ReduceByKey
((bird, d3), 1)


(one, (d1, 1))
(fish, (d1, 2))
(two, (d1, 1))
ap
(one, (d1, 1), 3/2)
(fish, (d1, 2), 3/2)
(two, (d1, 1), 3/1)
join
p
(one, 1)
(fish, 1)
(two, 1)
ReduceByKey
(one, 2)
(fish, 2)
(two, 1)
Map
Map
...
(one, d1, 1 * 3/2)
(fish, d1, 2 * 3/2)
(two, d1, 1 * 3/1)
(one, 3/2)
(fish, 3/2)
(two, 3/1)

Bilan Spark RDD

- Limitations
  - Pas de schéma: code peu lisible, prog. fastidieuse
  - Possibilité de définir type T, perf dégradée à cause de la sérialisation et GC
  - Optimisation logique (ex. sélection avant jointure) quasi impossible
- API de base supportant les API de plus haut niveau
- utiliser si traitement pas exprimable API haut niveau

Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. 20xx
Zaharia et al . Apache Spark: A Unified Engine For Big Data Processing. 2016