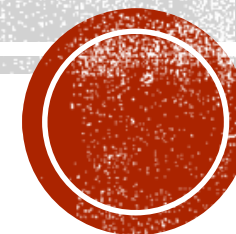


Doc. RNDr. Irena Holubová, Ph.D. & PROFINIT

DATA SCIENCE

NDBI048

Big Data and Data Science



<https://www.ksi.mff.cuni.cz/~holubova/NDBI048/>

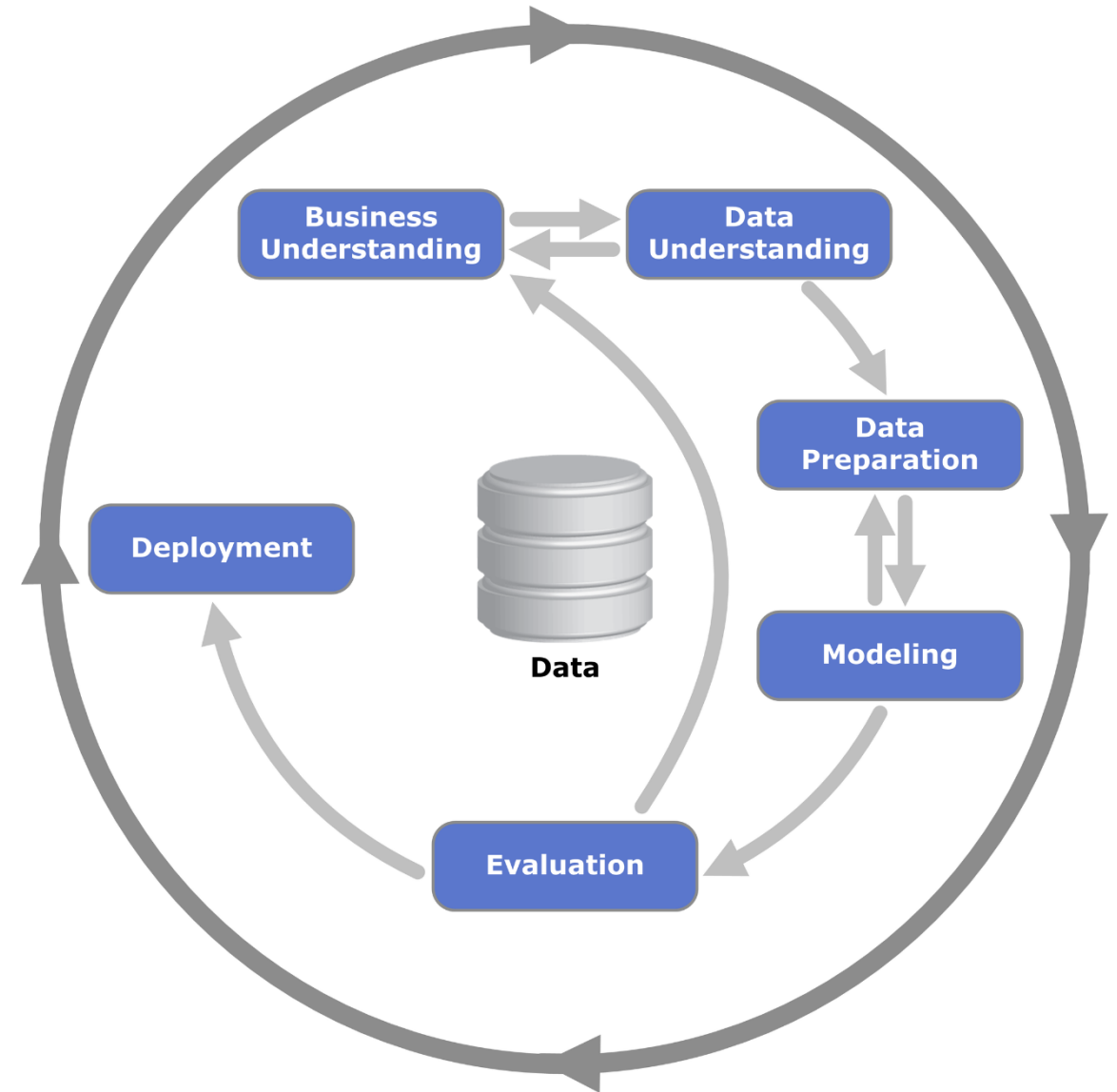
OUTLINE

- Big Data
- MapReduce
- Apache Spark
- Modern database systems



CRISP-DM PHASES

- I. Business Understanding
- II. Data Understanding
- III. Data Preparation
- IV. Modeling
- V. Evaluation
- VI. Deployment

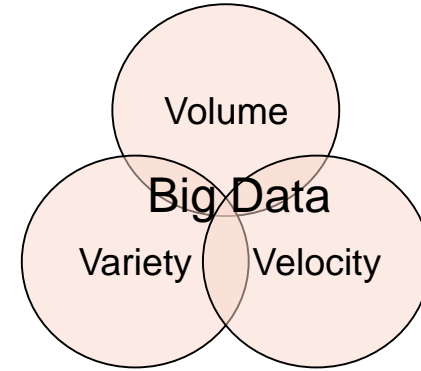


BIG DATA

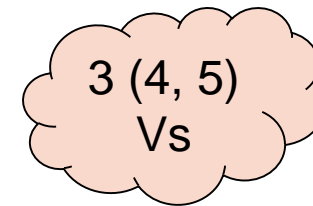


WHAT IS BIG DATA?

- No standard definition
- First occurrence of the term: **High Performance Computing (HPC)**



Gartner: “**Big Data**” is high **volume**, high **velocity**, and/or high **variety** information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.



WHAT IS BIG DATA?



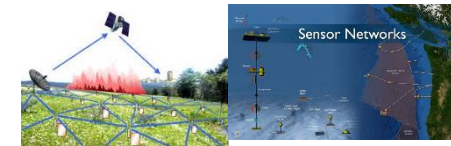
Social media and networks
(all of us are generating data)



Scientific instruments
(collecting all sorts of data)



Mobile devices
(tracking all objects all the time)



Sensor technology and networks
(measuring all kinds of data)

IBM: Depending on the industry and organization, **Big Data** encompasses information from internal and external sources such as transactions, social media, enterprise content, sensors, and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.



FACEBOOK BY THE NUMBERS: STATS, DEMOGRAPHICS & FUN FACTS (LAST UPDATE: APRIL 2020)



- 2.5 billion monthly active users
- 5 billion comments are left on Facebook pages monthly
- 55 million status updates are made every day
- Every 60 seconds
 - 317,000 status updates
 - 147,000 photos uploaded
 - 54,000 shared links

<https://www.omnicoreagency.com/facebook-statistics/>



MAIN PROBLEM: SCALABILITY

Vertical Scaling (scaling up)

- Traditional choice has been in favour of strong consistency
 - System architects have in the past gone in favour of scaling up (**vertical scaling**)
 - Involves larger and more powerful machines
- Works in many cases but...
- **Vendor lock-in**
 - Not everyone makes large and powerful machines
 - Who do, often use proprietary formats
 - Makes a customer dependent on a vendor for products and services
 - Unable to use another vendor

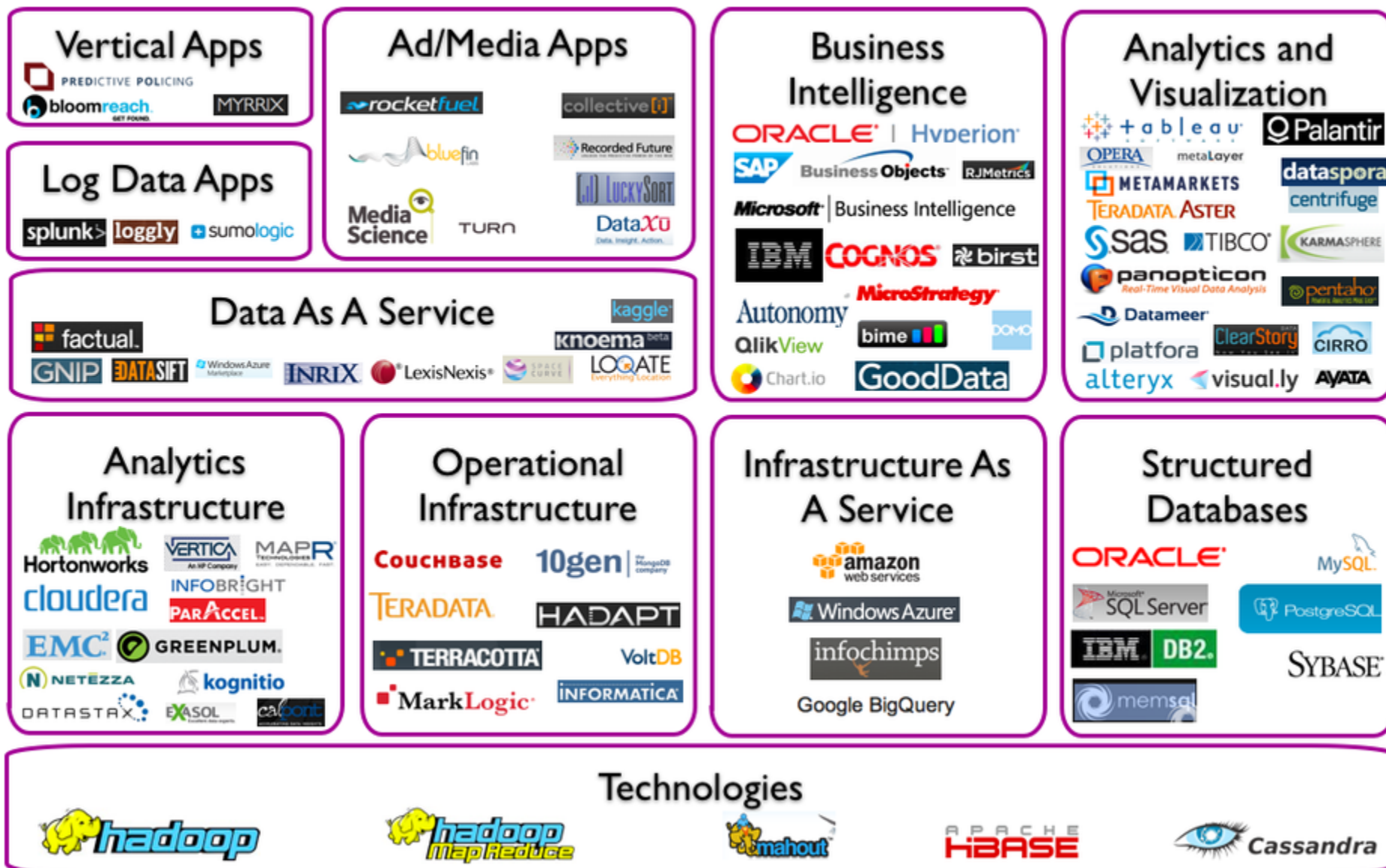
Horizontal Scaling (scaling out)

- Systems are distributed across multiple machines/nodes (**horizontal scaling**)
 - Commodity machines (cost effective)
 - Often surpasses scalability of vertical approach
- But...
- Fallacies of distributed computing:
 - The network is reliable
 - Latency is zero
 - Bandwidth is infinite
 - The network is secure
 - Topology does not change
 - There is one administrator
 - Transport cost is zero
 - The network is homogeneous



2012

Big Data Landscape



2021

Big Data Landscape



MAPREDUCE



MAPREDUCE FRAMEWORK



- A programming model + implementation
- Developed by Google in 2008
 - To replace old, centralized index structure
- Distributed, parallel computing on large data

Example: Von
Neumann's
model =
sequence of
instructions

Google: “A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.”

- **Programming model** in general:
 - Mental model a programmer has about execution of application
 - Purpose: improve programmer's productivity
 - Evaluation: expressiveness, simplicity, performance



MAPREDUCE FRAMEWORK

- Divide-and-conquer paradigm
 - **Map** breaks down a problem into sub-problems
 - Processes a key/value pair to generate a set of intermediate key/value pairs
 - **Reduce** receives and combines the sub-solutions to solve the problem
 - Processes intermediate values associated with the same intermediate key
- Many real-world tasks can be expressed this way
 - Programmer focuses on map/reduce code
 - Framework cares about data partitioning, scheduling execution across machines, handling machine failures, managing inter-machine communication, ...



MAPREDUCE

A BIT MORE FORMALLY

■ Map

- Input: a key/value pair
- Output: a set of intermediate key/value pairs
 - Usually different domain
- $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$

■ Reduce

- Input: an intermediate key and a set of all values for that key
- Output: a possibly smaller set of values
 - The same domain
- $(k_2, \text{list}(v_2)) \rightarrow (k_2, \text{possibly smaller list}(v_2))$



MAPREDUCE

EXAMPLE: WORD FREQUENCY

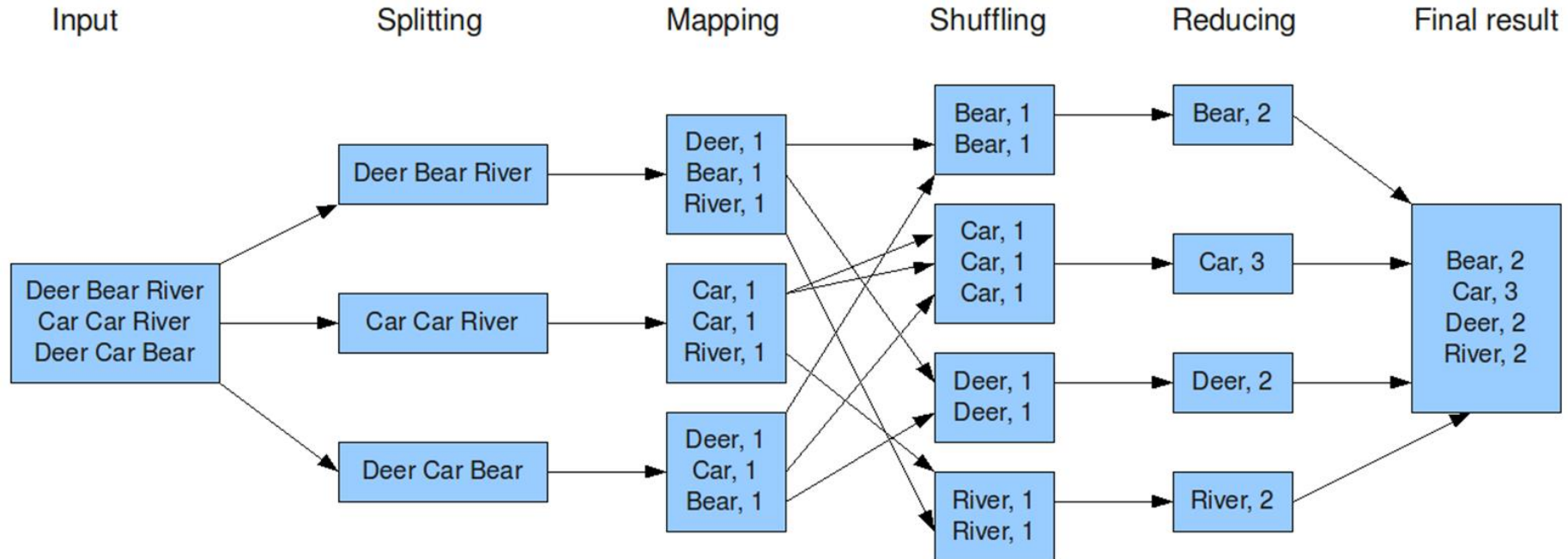
```
map(String key, String value):  
    // key: document name  
    // value: document contents  
    for each word w in value:  
        EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(key, AsString(result));
```



MAPREDUCE

EXAMPLE: WORD FREQUENCY



MAPREDUCE

MORE EXAMPLES

- **distributed grep**
 - Map: emits `<word, line number>` if it matches a supplied pattern
 - Reduce: identity
- **URL access frequency**
 - Map: processes web logs, emits `<URL, 1>`
 - Reduce: sums values and emits `<URL, sum>`
- **reverse web-link graph**
 - Map: `<target, source>` for each link to a target URL found in a page named source
 - Reduce: concatenates the list of all source URLs associated with a given target URL `<target, list(source)>`



MAPREDUCE CRITICISM

DAVID DEWITT AND MICHAEL STONEBRAKER — 2008

1. MapReduce is a step backwards in database access based on
 - Schema describing data structure
 - Separating schema from the application
 - Advanced query languages
2. MapReduce is a poor implementation
 - Instead of indices it uses brute force
3. MapReduce is not novel (ideas more than 20 years old and overcome)
4. MapReduce is missing features common in DBMSs
 - Indices, transactions, integrity constraints, views, ...
5. MapReduce is incompatible with applications implemented over DBMSs
 - Data mining, business intelligence, ...



NOTE: WHO IS MICHAEL STONEBRAKER?

- *1943
- Computer scientist – database researcher
- Academic prototypes form the core of various databases
 - Ingres, Postgres, C-store (Vertica), H-store (VoltDB), SciDB, ...
- 2015 – Turing award (ACM)
 - “Nobel Prize of computing”
 - For concepts and practices underlying modern database systems
- 2016 – Tim Berners Lee
 - For inventing the WWW



HADOOP MAPREDUCE

- MapReduce requires:
 - Distributed file system
 - HDFS = Hadoop distributed file system
 - Engine that can distribute, coordinate, monitor and gather the results
- Hadoop: HDFS + JobTracker + TaskTracker
 - **JobTracker** (master) = scheduler
 - **TaskTracker** (slave per node) – is assigned a Map or Reduce (or other operations)
 - Map or Reduce run on a node → so does the TaskTracker
 - Each task is run on its own JVM



MAPREDUCE

JOBTRACKER (MASTER)

- Like a scheduler:
 1. A client application is sent to the JobTracker
 2. It “talks” to the NameNode (= HDFS master) and locates the TaskTracker (Hadoop client) near the data
 3. It moves the work to the chosen TaskTracker node

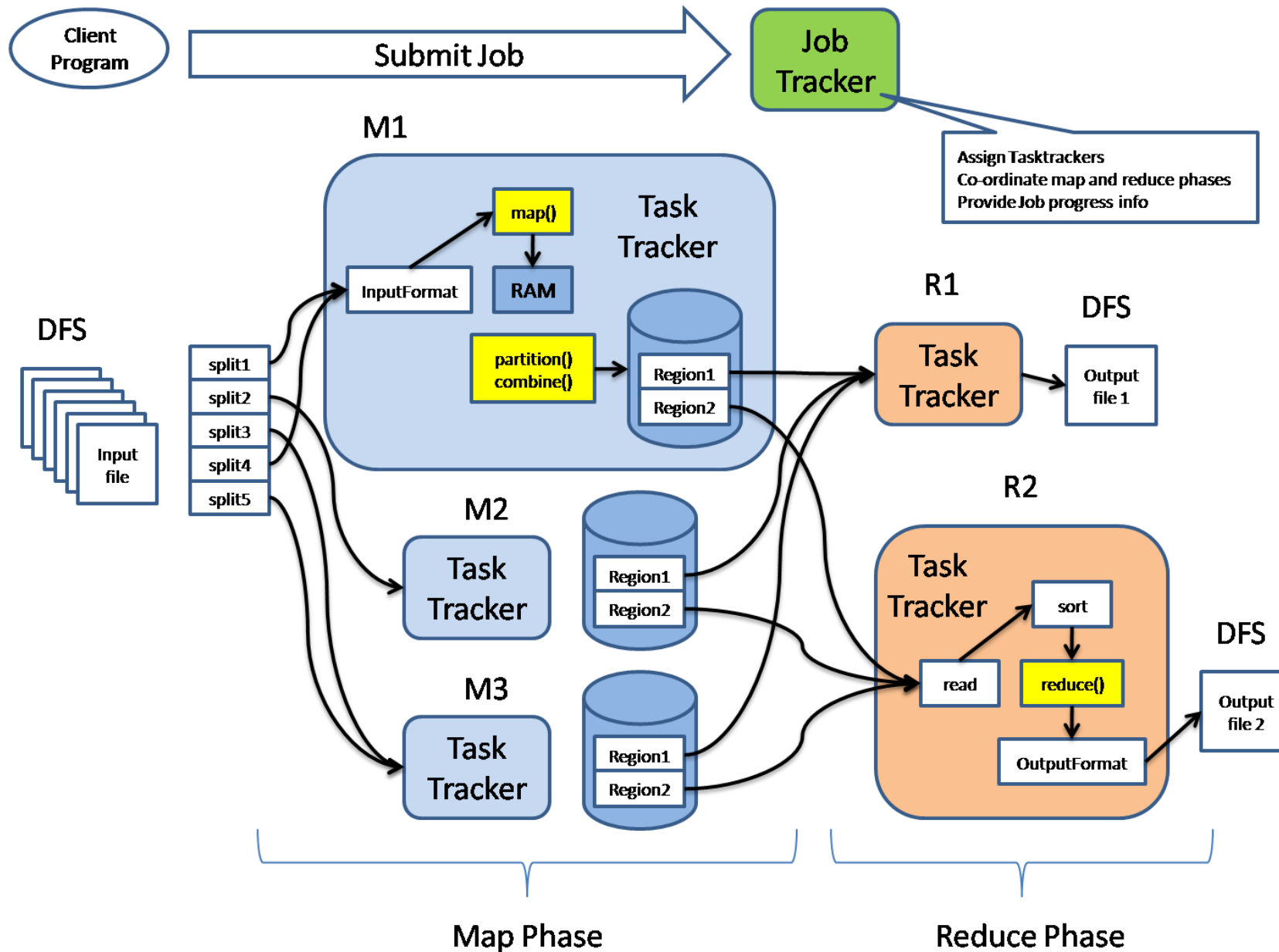


MAPREDUCE

TASKTRACKER (CLIENT)

- Accepts tasks from JobTracker
 - Map, Reduce, Combine, ...
 - Input, output paths
- Has a number of slots for the tasks
 - Execution slots available on the machine (or machines on the same rack)
- Spawns a separate JVM for execution of a task
- Indicates the number of available slots through the **heartbeat** message to the JobTracker
 - A failed task is re-executed by the JobTracker





JOB LAUNCHING

JOB CONFIGURATION

- For launching program:
 1. Create a **Job** to define a job
 - Using class Configuration
 2. Submit Job to the cluster and wait for completion
- **Job** involves:
 - Classes implementing Mapper and Reducer interfaces
 - `Job.setMapperClass()`
 - `Job.setReducerClass()`
 - Job outputs
 - `Job.setOutputKeyClass()`
 - `Job.setOutputValueClass()`
 - Other options:
 - `Job.setNumReduceTasks()`
 - ...



JOB LAUNCHING

JOB

- `waitForCompletion()` – waits (blocks) until the job finishes
- `submit()` – **does not block**
- `monitorAndPrintJob()` – monitor a job and print status in real-time as progress is made and tasks fail



```
public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map (Object key, Text value, Context context)
        throws IOException, InterruptedException {
        StringTokenizer itr
            = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}
```



```
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce (Text key,
                        Iterable<IntWritable> values,
                        Context context
                        )
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```



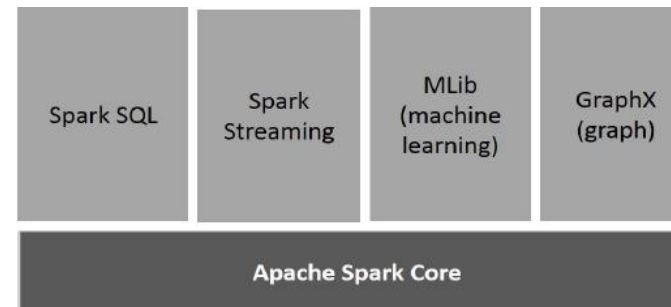
APACHE SPARK



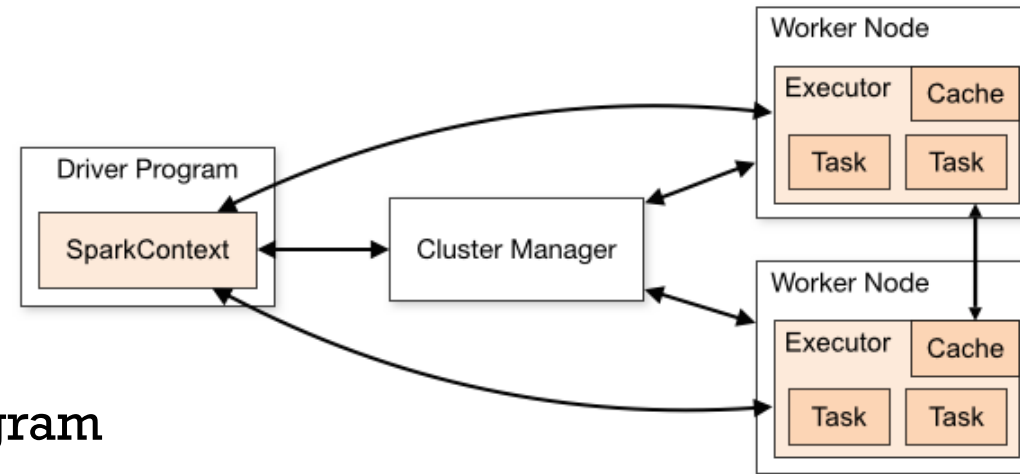
APACHE SPARK



- Initial release : 2014
- Unified analytics engine for large-scale data processing
 - Runs on a cluster of nodes
- Contains:
 - High-level APIs in Java, Scala, Python and R
 - Optimized engine that supports general execution graphs (DAGs)
 - MapReduce has only 2 levels
 - Higher-level tools
 - **Spark SQL** (SQL and structured data processing)
 - MLlib (machine learning)
 - GraphX (graph processing)
 - Spark Streaming



SPARK APPLICATION



- Spark application = driver program
 - Runs the user's main function
 - Executes parallel operations on a cluster
 - Independent set of processes
 - Coordinated by **SparkContext** object in the driver program
- SparkContext can connect to several types of cluster managers
 - They allocate resources across applications
- When connected:
 1. Spark acquires executors on nodes in the cluster
 - Processes that run computations and store data for the application
 2. Sends the application code to the executors
 - Defined by JAR or Python files passed to SparkContext
 3. Sends tasks to the executors to run



RESILIENT DISTRIBUTED DATASET (RDD)

- Immutable collection of elements partitioned across the nodes of the cluster
 - Can be operated on in parallel
 - Can be persisted in memory
 - Automatically recover from node failures
- Ways to create RDDs:
 1. Parallelizing an existing collection in a driver program
 2. Referencing a dataset in an external storage system
 - e.g., HDFS, HBase, ...
 - In general: any offering a Hadoop InputFormat



RESILIENT DISTRIBUTED DATASET (RDD)

PARALLELIZED COLLECTIONS

- Parallelized collections are created by calling SparkContext's **parallelize** method
 - Elements of the collection are copied to form a distributed dataset
 - The distributed dataset (**distData**) can be operated on in parallel
 - See later

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);  
JavaRDD<Integer> distData = sc.parallelize(data);
```



RESILIENT DISTRIBUTED DATASET (RDD)

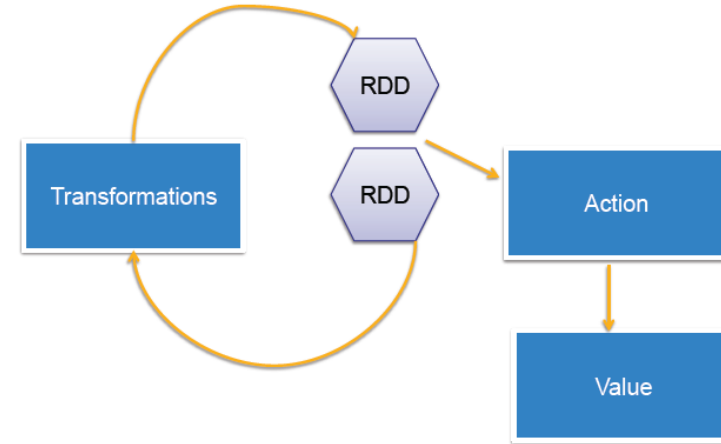
EXTERNAL DATASETS

- Spark can create distributed datasets from any storage source supported by Hadoop
 - Local file system, HDFS, Cassandra, HBase, ...
- Supports text files, SequenceFiles, and any other Hadoop InputFormat
- Example:
 - Text file RDDs can be created using SparkContext's `textFile` method
 - Takes an URI for the file (local, HDFS, ...)
 - Reads it as a collection of lines
 - Optional argument: number of partitions of the file
 - Default: one partition for each block of the file (128MB by default in HDFS)
 - Once created, `distFile` can be acted on by dataset operations

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```



RDD OPERATIONS



1. **Transformations** = create (lazily) a new dataset from an existing one
 - e.g., map = passes each dataset element through a function and returns a new RDD representing the results
2. **Actions** = return a value to the driver program after running a computation on the dataset
 - e.g., reduce = aggregates all the elements of the RDD using some function and returns the final result to the driver program
- By default: each transformed RDD may be recomputed each time we run an action on it
 - We may also persist an RDD in memory using the **persist** (or **cache**) method
 - Much faster access the next time we query it
 - There is also support for persisting RDDs on disk or replicated across multiple nodes



TRANSFORMATIONS

- **map**(func) Returns a new distributed dataset formed by passing each element of the source through a function func.
- **union**(otherDataset) Returns a new dataset that contains the union of the elements in the source dataset and the argument.
 - **intersection, distinct**
- **filter**(func) Returns a new dataset formed by selecting those elements of the source on which func returns true.
- **reduceByKey**(func, [numPartitions]) When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type $(V, V) \Rightarrow V$. The number of reduce tasks is configurable through an optional second argument.
- **sortByKey**([ascending], [numPartitions]) When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.
- ...



ACTIONS

- **reduce(func)** Aggregates the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- **count()** Returns the number of elements in the dataset.
- **first()** Returns the first element of the dataset.
- **take(n)** Returns an array with the first n elements of the dataset.
- **takeOrdered(n, [ordering])** Returns the first n elements of the RDD using either their natural order or a custom comparator.
- ...



SIMPLE SPARK EXAMPLE

```
JavaRDD<String> lines = sc.textFile("data.txt");  
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());  
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

1. Defines a base RDD from an external file
 - Not loaded in memory or otherwise acted on, due to laziness
 - **lines** is merely a pointer to the file
2. Defines **lineLengths** as the result of a map transformation
 - Not immediately computed, due to laziness
3. Runs reduce = action
 - Spark breaks the computation into tasks to run on separate machines
 - Each machine runs both its part of the map and a local reduction, returning its answer to the driver program



SPARK SQL



- Spark module for structured data processing
- More information about the structure of both the data and the computation being performed
 - Internally, Spark SQL uses this extra information to perform extra optimizations
- Interact with Spark SQL: SQL, Dataset API, ...



RDD VS. DATAFRAME VS. DATASET

- **RDD** = primary API in Spark since its inception
 - Since Spark 1.0
 - Internally each final computation is still done on RDDs
- **DataFrame** = data organized into named columns
 - Since Spark 1.3
 - Distributed collection of data, which is organized into named columns
 - Designed to make data processing easier
 - Higher level of abstraction
 - Similar to a table in a relational database or a data frame in R/Python
 - Can be constructed from: structured data files, tables in Hive, external databases, existing RDDs , ...
 - API: Scala, Java, Python, R



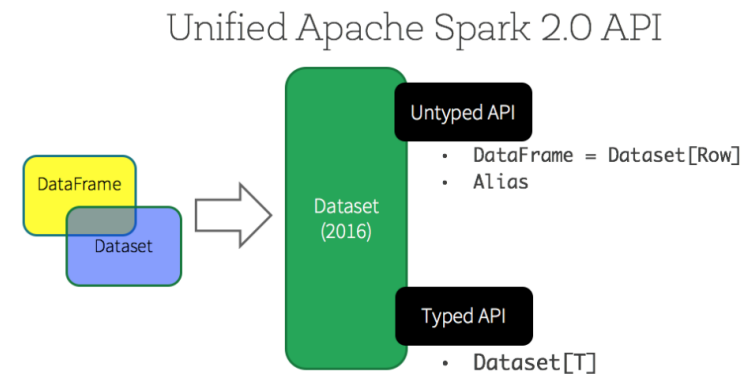
RDD VS. DATAFRAME VS. DATASET

- **Dataset** = a distributed collection of data
 - Since Spark 1.6
 - Provides the benefits of
 - RDDs - strong typing, ability to use powerful lambda functions
 - Spark SQL - optimized execution engine
 - i.e. DataFrame processing
 - Can be constructed from: JVM objects
 - API: Scala, Java



RDD VS. DATAFRAME VS. DATASET

- Since Spark 2.0: unification of DataFrame and Dataset
- Two distinct APIs:
 - Untyped API
 - Conceptually: DataFrame ~ collection of generic objects **Dataset<Row>**, where a Row is a generic untyped JVM object
 - Strongly-typed API
 - Conceptually: Dataset ~ collection **Dataset<T>** of strongly-typed JVM objects, dictated by a case **class T**
 - Defined in Scala or a class in Java



BASIC EXAMPLES


```
SparkSession spark = SparkSession.builder().  
  appName("Java Spark SQL basic example").  
  config("spark.some.config.option", "some-value").getOrCreate();
```

```
Dataset<Row> df =  
  spark.read().json("examples/src/main/resources/people.json");
```

DataFrame - untyped

```
df.show();  
// +----+-----+  
// | age|   name|  
// +----+-----+  
// |null|Michael|  
// |  30|   Andy|  
// |  19|  Justin|  
// +----+-----+
```

```
df.printSchema();  
// root  
// |-- age: long (nullable = true)  
// |-- name: string (nullable = true)
```



4 lines (3 sloc) 73 Bytes	
1	{"name": "Michael"}
2	{"name": "Andy", "age": 30}
3	{"name": "Justin", "age": 19}



```
// Select only the "name" column
```

```
df.select("name").show();
```

```
// +-----+
```

```
// |   name|
```

```
// +-----+
```

```
// |Michael|
```

```
// |   Andy|
```

```
// | Justin|
```

```
// +-----+
```

```
// Select everybody, but increment the age by 1
```

```
df.select(col("name"), col("age").plus(1)).show();
```

```
// +-----+-----+
```

```
// |   name|(age + 1)|
```

```
// +-----+-----+
```

```
// |Michael|      null|
```

```
// |   Andy|       31|
```

```
// | Justin|       20|
```

```
// +-----+-----+
```

```
// Select people older than 21
```

```
df.filter(col("age").gt(21)).show();
```

```
// +---+---+
```

```
// |age|name|
```

```
// +---+---+
```

```
// | 30|Andy|
```

```
// +---+---+
```



```
// Count people by age
df.groupBy("age").count().show();
// +-----+-----+
// | age|count|
// +-----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +-----+-----+
```

```
// Register the DataFrame as an SQL temporary view
df.createOrReplaceTempView("people");
```

```
Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
// +-----+-----+
// | age|  name|
// +-----+-----+
// |null|Michael|
// |  30|  Andy|
// |  19| Justin|
// +-----+-----+
```

- **Temporary views are session-scoped**
 - Disappear if the session that creates it terminates
- **Global temporary view = a temporary view shared among all sessions**
 - Keeps alive until the Spark application terminates
 - Tied to a system preserved database `global_temp`
 - `df.createGlobalTempView("people");`
 - Must use the qualified name to refer it
 - e.g. `SELECT * FROM global_temp.people`



MODERN DATABASE SYSTEMS



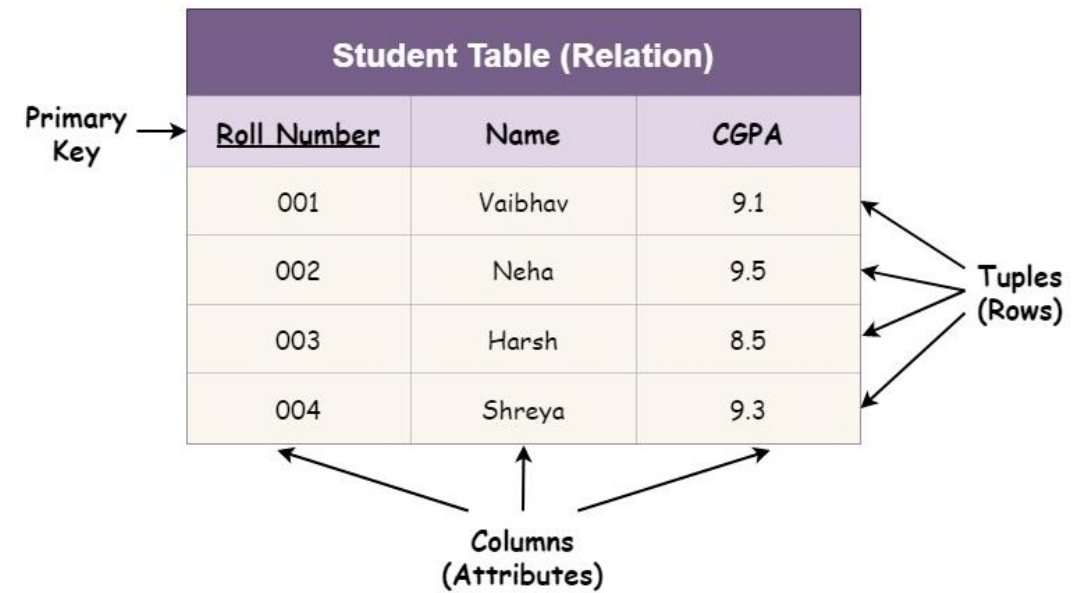
DATABASE = RELATIONAL DATABASE?

- A common assumption for many years
- Relational databases are able to store and process various data structures
- Advantages:
 - Simplicity
 - of the model
 - of the respective query language
 - After so many years mature and verified database management systems (DBMSs)
 - Strong mathematical background
 - ...



RELATIONAL MODEL

- Proposed by E.F. Codd in 1970
 - Paper: “A relational model of data for large shared data banks”
 - IBM Research Labs
- Basic idea:
 - Storing of object and their mutual associations in **tables** (relations)
 - A **relation** R from X to Y is a subset of the Cartesian product $X \times Y$.
 - **Row** in a table (member of relation) = object/association
 - **Column** (attribute) = attribute of an object/association
 - **Table** (relational) **schema** = name of the schema + list of attributes and their types
 - **Schema of a relational database** = set of relational schemas
 - Integrity constraints



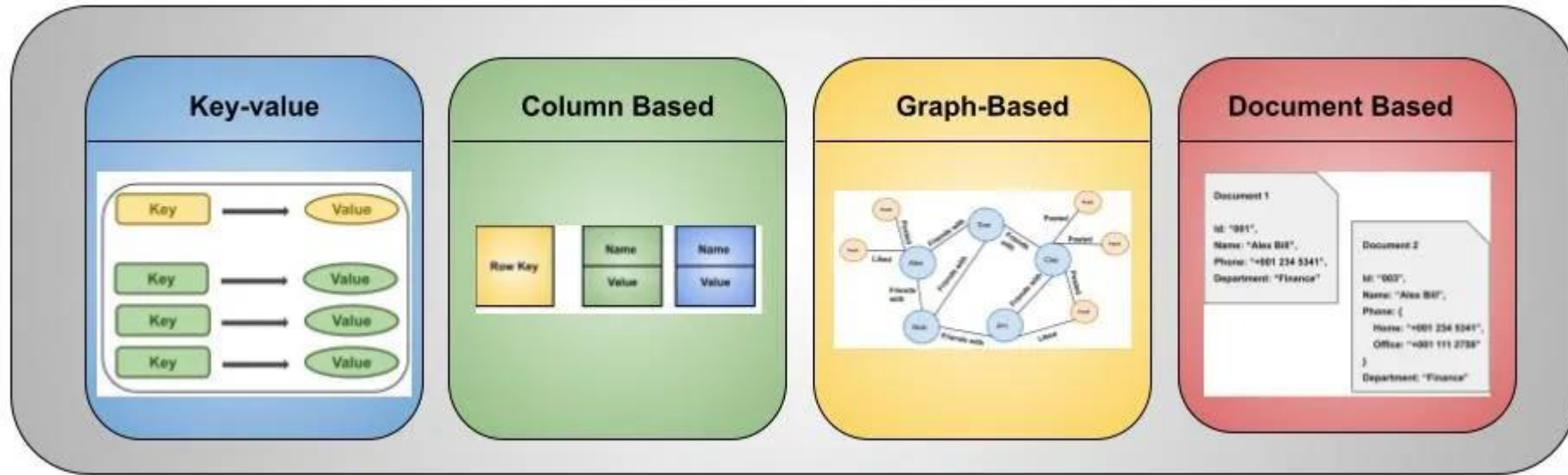
BUT THE RELATIONAL MODEL WAS NOT THE FIRST ONE...

- First generation: navigational
 - Hierarchical model
 - Network model
- Second generation: relational
- Third generation: post-relational
 - Extensions of relational model
 - Object-relational
 - New models reacting to popular technologies
 - Object
 - XML
 - NoSQL (key/value, column, document, graph, ...) - Big Data
 - Array databases
 - Multi-model systems
 - ...
 - NewSQL
 - Back to the relations

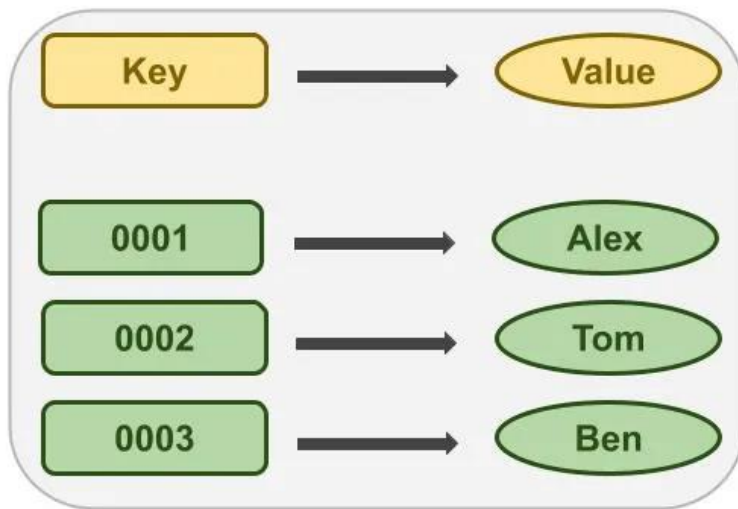
time



NOSQL DATABASES



KEY-VALUE STORE



- The simplest NoSQL data stores
- A simple hash table (map), primarily used when all access to the database is via primary key
- A table in RDBMS with **two columns**, such as ID and NAME
 - **ID** column being the key
 - **NAME** column storing the value
 - A BLOB that the data store just stores
- Basic operations:
 - Get the value for the key
 - Put a value for a key
 - Delete a key from the data store



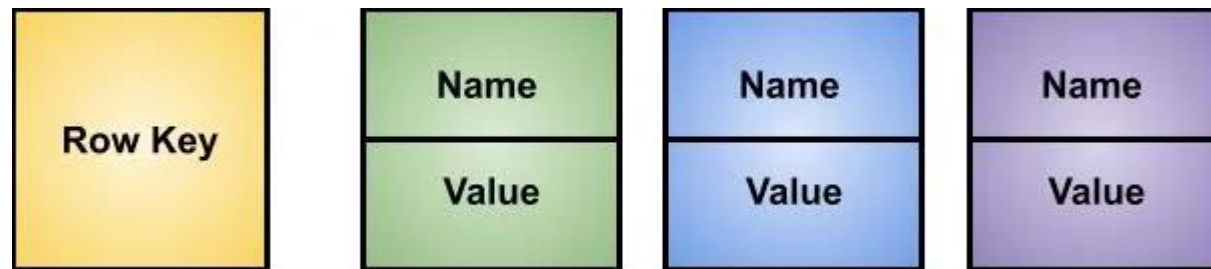
KEY-VALUE STORE

- We can query by the **key**
- To query using some attribute of the value column is (typically) not possible
 - We need to read the value to figure out if the attribute meets the conditions
- What if we do not know the key?
 - Some systems enable to retrieve the list of all keys
 - Expensive
 - Some support searching inside the value
 - Using, e.g., a kind of full-text index
- Often many extensions
 - MapReduce



COLUMN-FAMILY STORES

- Also “columnar” or “column-oriented”
- **Column families** = rows that have many columns associated with a **row key**
- Column families are groups of related data that is often accessed together
 - e.g., for a customer we access all profile information at the same time, but not orders



blog relational database

users table

user_id	username	state
1	jbellis	TX
2	dhutch	CA
3	egilmore	NULL

blog table

blog_id	user_id	blog_entry	categoryid
101	1	Today I ...	3
102	2	I am ...	2
103	1	This is ...	3

subscriber table

subscriber	blogger	row_id
1	2	1
2	1	2
1	3	3

category table

category	categoryid
sports	1
fashion	2
technology	3

blog keyspace

users

	name	state
jbellis	jonathan	TX
dhutch	daria	CA
egilmore	eric	

blog entries

	body	user	category
92dbeb5	Today I ...	jbellis	tech
d418a66	I am ...	dhutch	fashion
6a0b483	This is ...	egilmore	sports

* = secondary indexes

subscribes_to

jbellis	dhutch	egilmore
dhutch	jbellis	
egilmore	jbellis	dhutch

subscribers_of

jbellis	dhutch	egilmore
dhutch	egilmore	dhutch
egilmore	jbellis	

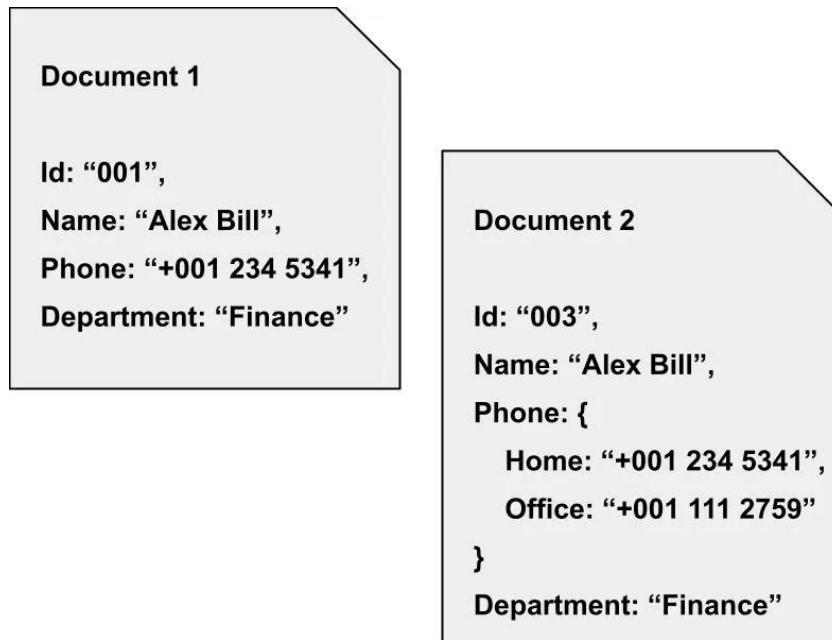
time_ordered_blogs_by_user

jbellis	1289847840615
	92dbeb5
dhutch	1289847840615
	d418a66
egilmore	1289847844275
	6a0b483

Other column families / secondary indexes for special queries



DOCUMENTS STORES

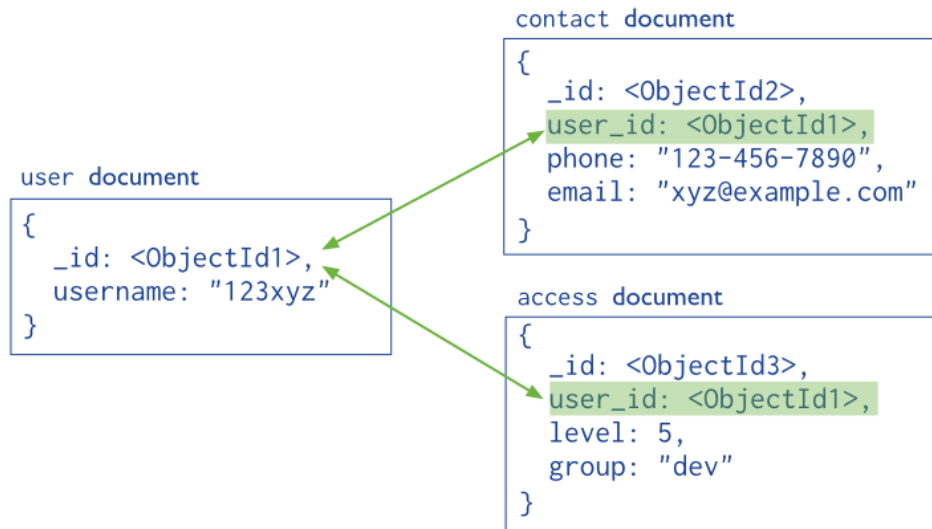


- Documents are the main concept
 - Stored and retrieved
 - XML, JSON, ...
- Documents are
 - Self-describing
 - Hierarchical tree data structures
 - Can consist of maps, collections (lists, sets, ...), scalar values, nested documents, ...
- Documents in a collection are expected to be **similar**
 - Their schema can differ
- Document databases store documents in the value part of the key-value store
 - Key-value stores where the value is **examinable**

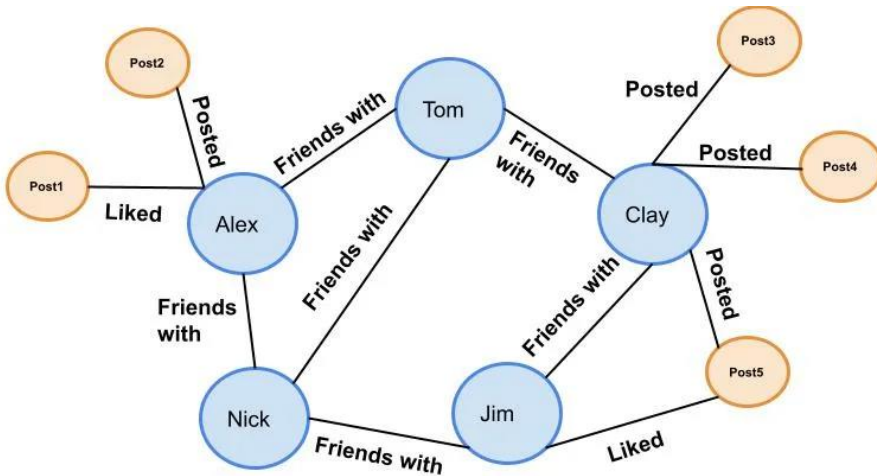


DOCUMENTS STORES

References vs. embedding



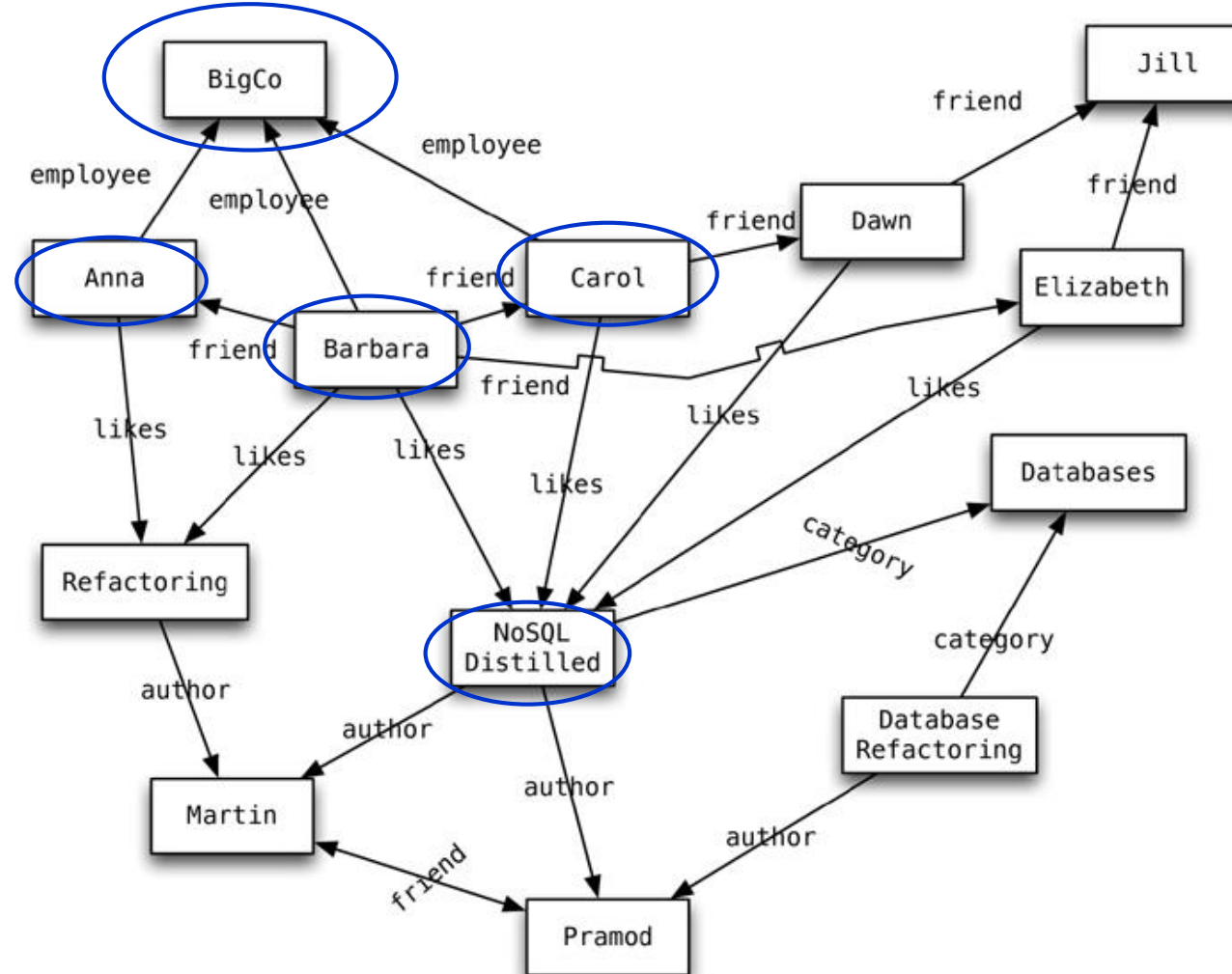
GRAPH STORES



- To store entities and relationships between these entities
 - Node is an instance of an object
 - Nodes have properties
 - e.g., name
 - Edges have directional significance
 - Edges have types
 - e.g., likes, friend, ...



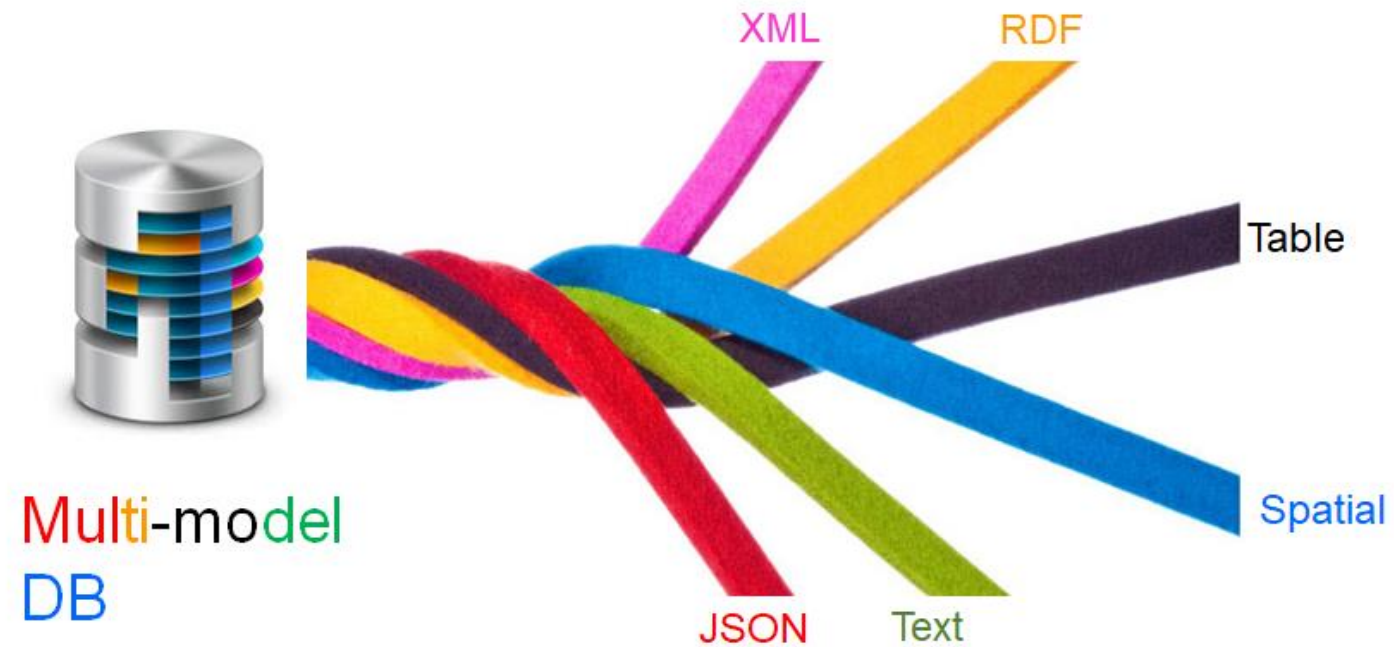
EXAMPLE:



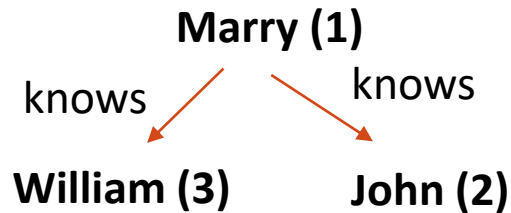
“Get all people (= nodes in the graph) employed by Big Co that like (book called) NoSQL Distilled”



MULTI-MODEL DATABASES



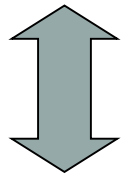
AN EXAMPLE OF MULTI-MODEL DATA AND QUERY



Graph-key/value join

"1" --> "34e5e759"

"2"--> "0c6df508"



Relation-graph join

Customer_ID	Name	Credit_limits
1	Mary	5,000
2	John	3,000
3	William	2,000

Key/value-JSON join

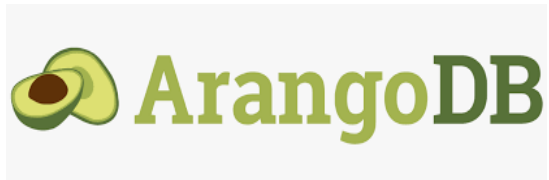


```
{ "Order_no": "0c6df508",  
  "Orderlines": [  
    { "Product_no": "2724f",  
      "Product_Name": "Toy",  
      "Price": 66 },  
    { "Product_no": "3424g",  
      "Product_Name": "Book",  
      "Price": 40 } ] }
```

Recommendation query:

Return all product_no-s which are ordered by a friend of a customer whose credit_limit>3000





```
LET CustomerIDs = (  
  FOR Customer IN Customers  
  FILTER Customer.CreditLimit > 3000  
  RETURN Customer.id)  
LET FriendIDs = (  
  FOR CustomerID IN CustomerIDs  
    FOR Friend IN 1..1 OUTBOUND CustomerID Knows  
  RETURN Friend.id)  
FOR Friend in FriendIDs  
FOR Order in 1..1 OUTBOUND Friend Customer2Order  
RETURN Order.orderlines[*].Product_no
```

```
SELECT  
  expand( out("Knows").Orders.  
    orderlines.Product_no )  
FROM Customers  
WHERE CreditLimit > 3000
```



REFERENCES

- Jeffrey Dean and Sanjay Ghemawat: **MapReduce: Simplified Data Processing on Large Clusters**, Google, Inc.
 - <http://labs.google.com/papers/mapreduce.html>
- Google Code: **Introduction to Parallel Programming and MapReduce**
 - code.google.com/edu/parallel/mapreduce-tutorial.html
- **Apache Hadoop:** <http://hadoop.apache.org/>
- **Hadoop Map/Reduce Tutorial**
 - http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html
- **Open Source MapReduce**
 - <http://lucene.apache.org/hadoop/>
- **Hadoop: The Definitive Guide**, by Tom White, 2nd edition, O'Reilly's, 2010
- David DeWitt and Michael Stonebraker: **Relational Database Experts Jump The MapReduce Shark**



REFERENCES

- Spark Overview <https://spark.apache.org/docs/latest/index.html>
- Apache Spark Examples <https://spark.apache.org/examples.html>
- Mastering Apache Spark 2.3.2 <https://jaceklaskowski.gitbooks.io/mastering-apache-spark/>
- A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets <https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html>



REFERENCES

- <http://nosql-database.org/>
- Pramod J. Sadalage – Martin Fowler: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence
- Eric Redmond – Jim R. Wilson: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement
- Sherif Sakr – Eric Pardede: Graph Data Management: Techniques and Applications
- Shashank Tiwari: Professional NoSQL
- Neither Fish Nor Fowl: the Rise of Multi-model Databases. The 451 Group, 2013.
- D. Feinberg, M. Adrian, N. Heudecker, A. M. Ronthal, and T. Palanca. Gartner Magic Quadrant for Operational Database Management Systems, 12 October 2015.
- J. Lu, Z. H. Liu, P. Xu, and C. Zhang. UDBMS: road to unification for multi-model data management. CoRR, abs/1612.08050, 2016
- J. Lu: Towards Benchmarking Multi-model Databases. CIDR 2017
- S. Abiteboul et al: Research Directions for Principles of Data Management, Dagstuhl Perspectives Workshop 16151 (2017)

