CHAPTER 1

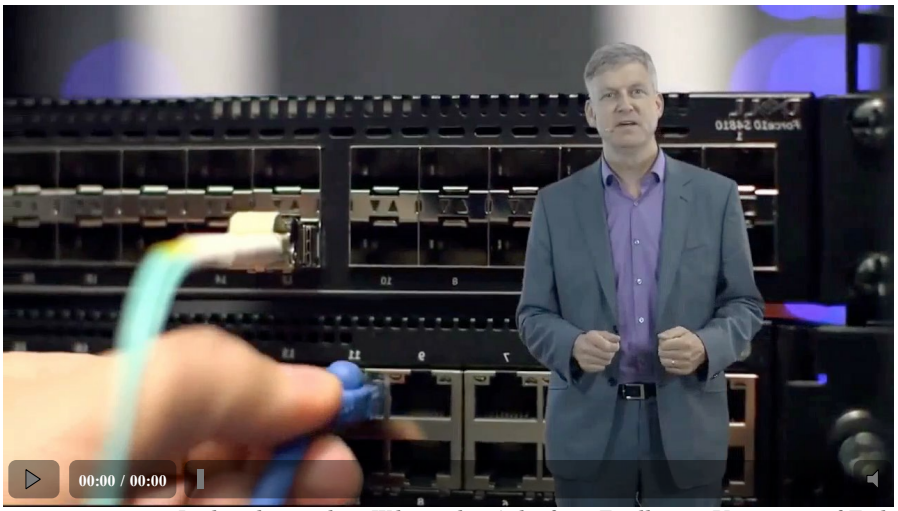
What Is Data Science?

This chapter introduces the concept of Data Science and presents the European Data Science Academy.

Data Scientist: The Sexiest Job Of The 21st Century

Information technology is so established in the European economy that it has driven, directly or indirectly (for instance, in software creation and in the upheaval of modern financial trading), much of the continent's economic growth during the last decades. The increasing ubiquity of computing devices, and high bandwidth consumer Internet, recording every aspect of life, from private households to multinational corporations and public administration, is driving a new information revolution based on the availability of ever increasing amounts of data. Handling digital data in all its forms (textual, geographical, financial, photographic/video, etc.) is now a core competence for many businesses, and also increasingly important for governmental agencies. This surfeit of information, supported by rapidly evolving technologies, makes it possible to do things that could not be done before: spot business trends, optimize supply chains, prevent diseases, detect earthquakes, combat crime, and so on. This data-centric phenomenon has been referred to as "data science", and has "gone mainstream" after being discussed in a number of prominent journalistic sources.

MOVIE 1.1 Introduction to Data Science



In this short video, Wil van der Aalst from Eindhoven University of Technology talks about Data Science and Process Mining.

Declared by Harvard Business Review as the "sexiest job of the 21st century", data science skills are becoming a key asset in any organization confronted with the daunting challenge of making sense of information that comes in varieties and volumes never encountered before. The title has been around for a while now, after being first introduced in 2008 to refer to the leads of data analytics efforts at two prominent IT companies in Silicon Valley, and is typically linked to a number of core areas of expertise, from the ability to operate high-performance computing clusters and cloud-based infrastructures, to the know-how that is required to devise and apply sophisticated Big Data analytics techniques, and the creativity involved in designing powerful visualizations. Moving further away from the purely technical, organizations are more and more looking into novel ways to capitalize on the data they own, and to generate added value from an increasing number of data sources openly available on the Web, a trend which has been coined as "open data". To do so they need their employees to understand the legal and economic aspects of data-driven business development, as a prerequisite for the creation of product and services that turn open and corporate data assets into decision making insight and commercial value.

Surviving in the data economy depends on hiring data professionals who master both the technical and non-technical facets of data science, from Big Data technology and data-driven storytelling, to new data monetization and innovation models. The challenge for managers is thus to identify and prioritize their knowledge gaps in this rapidly evolving, and to some extent interdisciplinary field, secure new data science talent, and train their existing staff into becoming proficient data practitioners and entrepreneurs.

AUDIO 1.1 Distilling value and driving productivity from mountains of data.

In this podcast, Michael Chui discusses how the scale and scope of companies' access to data is changing the way they do business.

Data scientists are, however, still a rare breed. Beyond the occasional data-centric startup and the data analytics department of large corporations, the skills scarcity is already becoming a threat for many European companies and public sector organizations as they struggle to seize Big Data opportunities in a globalized world. A well-known McKinsey study estimated already in 2011 that the United States will soon require 60 percent more graduates able to handle large amounts of data as part of their daily jobs. With an economy of comparable size (by GDP) and growth prospects, Europe will most likely be confronted with a similar talent shortage of hundreds of thousands of qualified data scientists, and an even greater need of executives and support staff with basic data literacy. The number of job descriptions and an increasing demand in higher-education programs and professional training confirm this trend, with some EU countries forecasting an increase of almost 100 percent in the demand for data science positions in less than a decade.

The European Data Science Academy

The 'Age of Data' continues to thrive, with data being produced from all industries at a phenomenal rate that introduces numerous challenges regarding the collection, storage and analysis of this data. To address this problem, the European Data Science Academy (EDSA) will establish a virtuous learning production cycle for Data Science, and will:

Analyse the sector specific skillsets for data analysts across Europe's main industrial sectors;

Develop modular and adaptable curricula to meet these Data Science needs; and

Deliver training supported by multiplatform and multilingual learning resources based on these curricula.

Throughout the project, the curricula and learning resources will be guided and evaluated by experts in both Data Science and pedagogy to ensure they meet the needs of the Data Science community. The following sections outline some of the activities that are being carried out to help meet these goals.

Demand Analysis

EDSA is monitoring trends across the EU to assess the demands for particular Data Science skills and expertise. We are leveraging a vast network of European data providers, consumers and intermediaries to "track the pulse" of the European data landscape. This allows us to align our criteria with the latest demands of the community.

Using interviews with Data Science practitioners, an industry advisory board representing a mix of sectors and automated tools for extracting data about job posts and news articles, we are building dashboards to present the current state of the European Data Science landscape, with the data feeding into our curricula development.

Curricula Development

EDSA is developing a core Data Science curriculum based on topics extracted from the demand analysis. The first version of this curriculum is available here. We are producing high-quality, multilingual and multimodal training materials to cover these topics, utilising existing resources available in the public domain and the internal expertise of the EDSA consortium. As one of the most innovative distance education institutions in the world, The Open University's outstanding knowledge in delivering eLearning will be leveraged extensively.

All resources produced by the project will be licensed as Creative Common 4.0 (CC BY 4.0). The curricula design will arrange these resources into pathways that can be customised to suit the different needs of individuals, and the requirements of self-study and tutor-led training formats.

Training Delivery and Learning Analytics

Key parts of our curricula will be delivered through eBooks, MOOCs, webinars, video lectures and face-to-face training. Several members of the EDSA consortium are already established as high-quality training providers in core Data Science topics. As part of EDSA, these initiatives will be structured into integrated learning pathways, translated into European languages, and expanded to meet the requirements for specific sectors as indicated by our demand analysis.

We are primarily using VideoLectures.NET and FutureLearn – the largest European MOOC platform, founded by The Open University – to maximise outreach and uptake of our materials. Engagement with learners and Big Data stakeholders is key to our training, and therefore monitoring and analysis tools will be utilised to assess learner progress – rather than by simply listening to academics or technology evangelists.

CHAPTER 2

Foundations Of Big Data

This chapter introduces the foundations of Big Data by presenting the tools that operate with Big Data and Big Data applications, such as the QMiner tool for Data Analytics.

Introduction To Big Data

This section describes the notion of Big Data, paying a special attention to interesting facts about Big Data. This section also provides a view on Big Data in numbers.

What is Big Data? In numbers, Big Data is a growling torrent:

$600 to buy a disk drive that can store all of the world's music

- 5 billion mobile phones in use in 2010

30 billion pieces of content shared of Facebook every month

- 40% projected growth in global data generated per year vs 5% growth in global IT spending

235 terabytes data collected by the US Library of Congress by April 2011

- 15 out of 17 sectors in the United States have more data stored per company than the US Library of Congress

$5 million vs. $400 Price if the fastest supercomputer in 1975 and an iPhone 4 with equal performance.

Figure 1 demonstrates the intensity of new data arrival.

Figure 1: In 60 seconds

Global online population comes to around 30% of world's population. Global time spent online every month is equivalent to 3,995,444 years. Average time spent by global use per month is 16 hours.

Figure 2 shows how people spend their time online. Figure 3 demonstrates a set of interesting facts about WWW users.

Figure 2: How people spend their time online

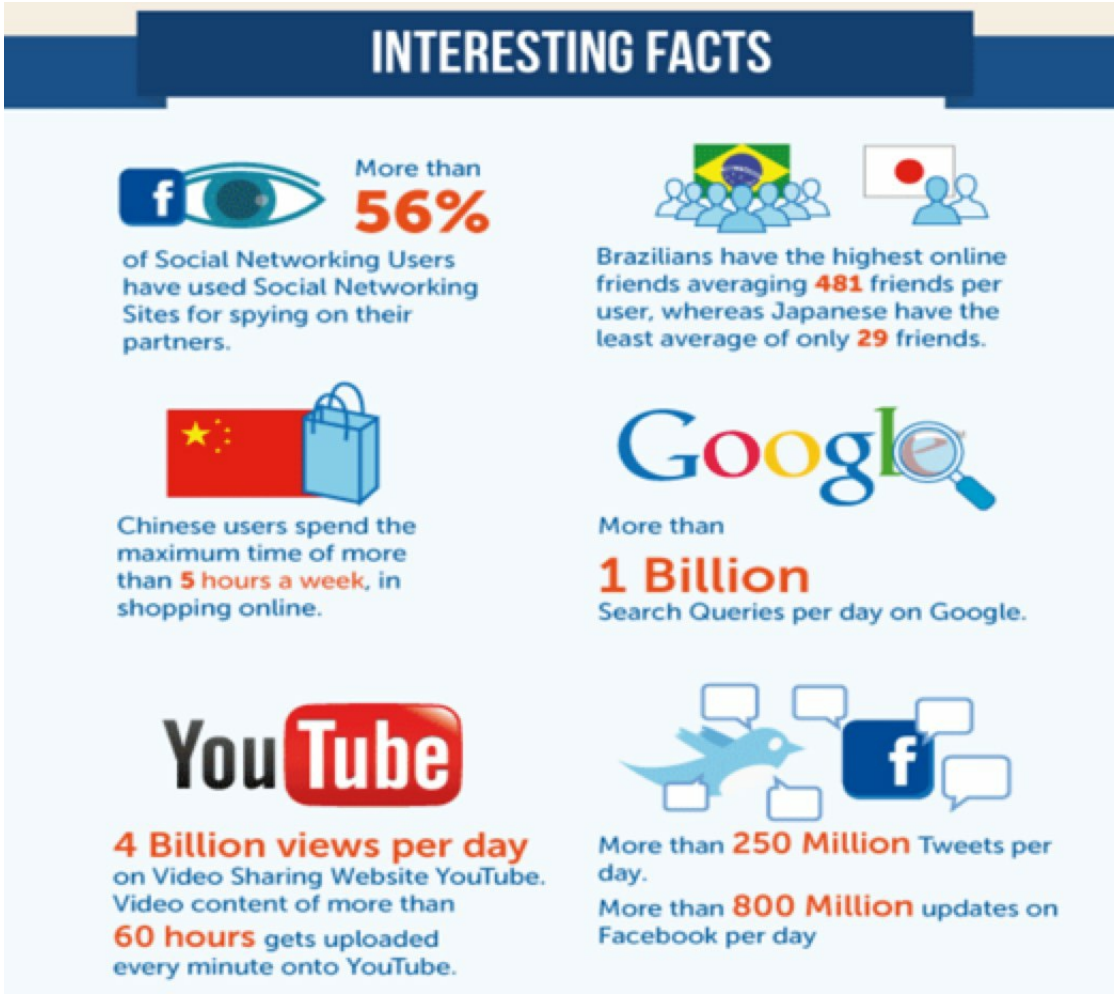


Figure 3: Interesting facts about WWW users

Big Data Definitions, Motivation and State of Market

This section shows the definitions of Big Data, the motivation behind operating with Big Data. The characterization of Big Data by volume, velocity, variety (V3) is provided as well as Big Data popularity on the Web, Big Data hype cycles, Big Data value chain and view on the Big Data market.

Big Data is a term for large and complex data sets that are difficult to process with standard data processing scenarios. Having data "bigger" requires different approaches: techniques, tools, architectures. Big Data is processed with an aim to solve new problems or to improve the solutions for the old problems.

Big Data is characterized by *Volume*, *Velocity* and *Variety* (V3):

Volume can be described as a characteristic for a challenging to load and process information (how to index, retrieve data).

Variety is a characteristic for different data types and degree of structure (how to query semi-structured data)

Velocity is a characteristic for real-time processing influenced by rate of data arrival.

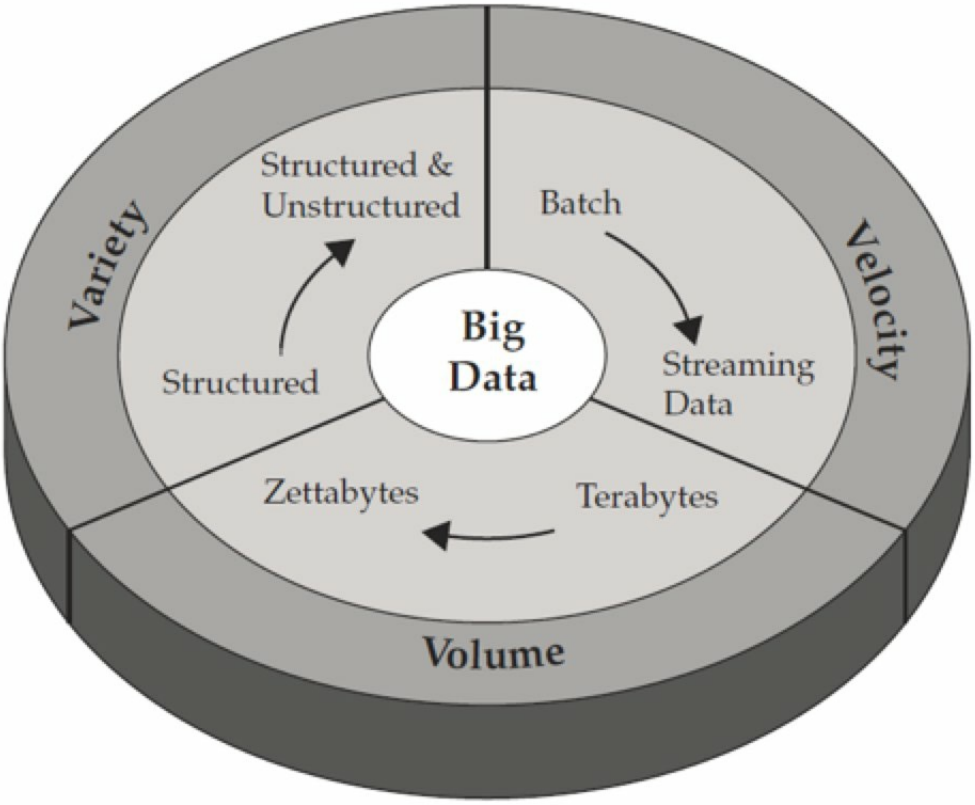


Figure 4: From “Understanding Big Data” by IBM

The extended 3+n Vs of Big Data include:

Volume (lots of data = “Tonnabytes”)

Variety (complexity, curse if dimensionality)

Velocity (rate of data and information flow)

Veracity (verifying inference-based models from comprehensive data collections)

Variability

Venue (location)

Vocabulary (semantics)

Historically speaking, the popularity of Big Data increases from year 2006.

Figure 5 shows the Big Data popularity on the Web (via Google Trends). The popularity of “Big Data” (blue line) is growing since 2006, in comparison to the popularity of “Machine learning” (red line).

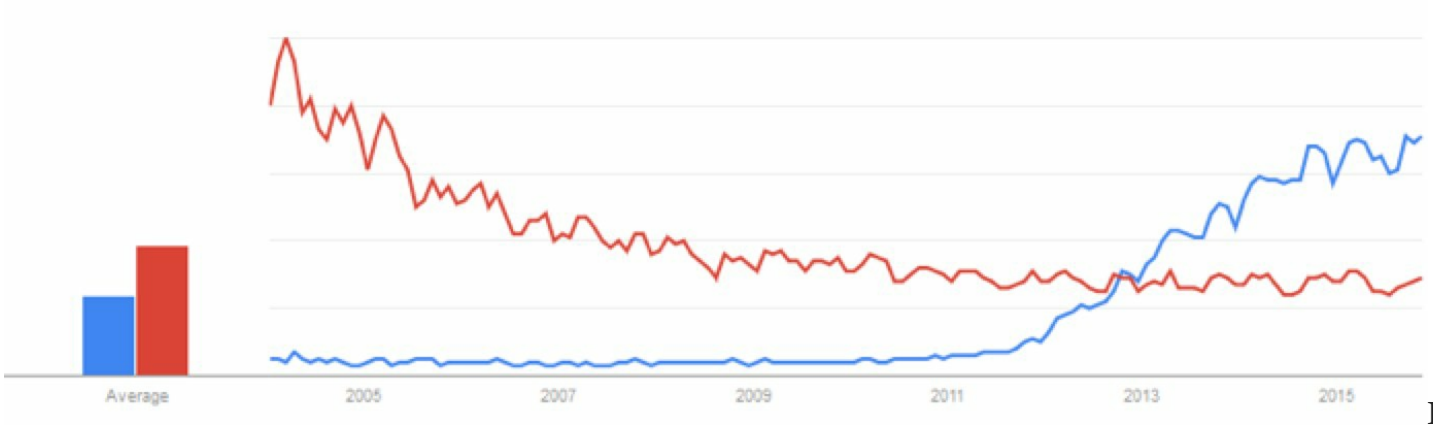


Figure 5: Big Data popularity on the Web

Figure 6 and Figure 7 demonstrate the emerging technologies hype cycles for 2013 and 2014. On the graphs it is possible to see how “Big Data” point moves inside the peak of inflated expectations.

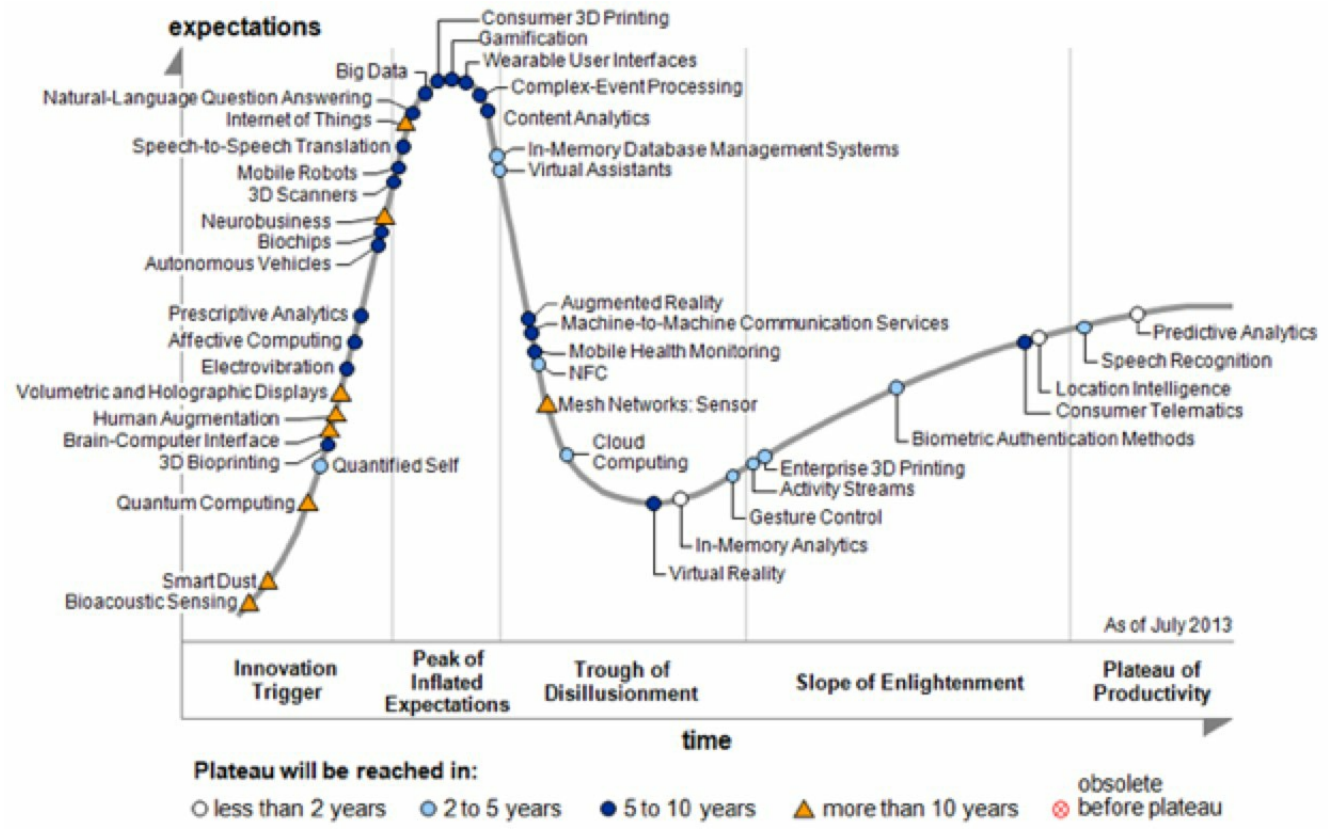


Figure 6: Emerging technologies hype cycle 2013

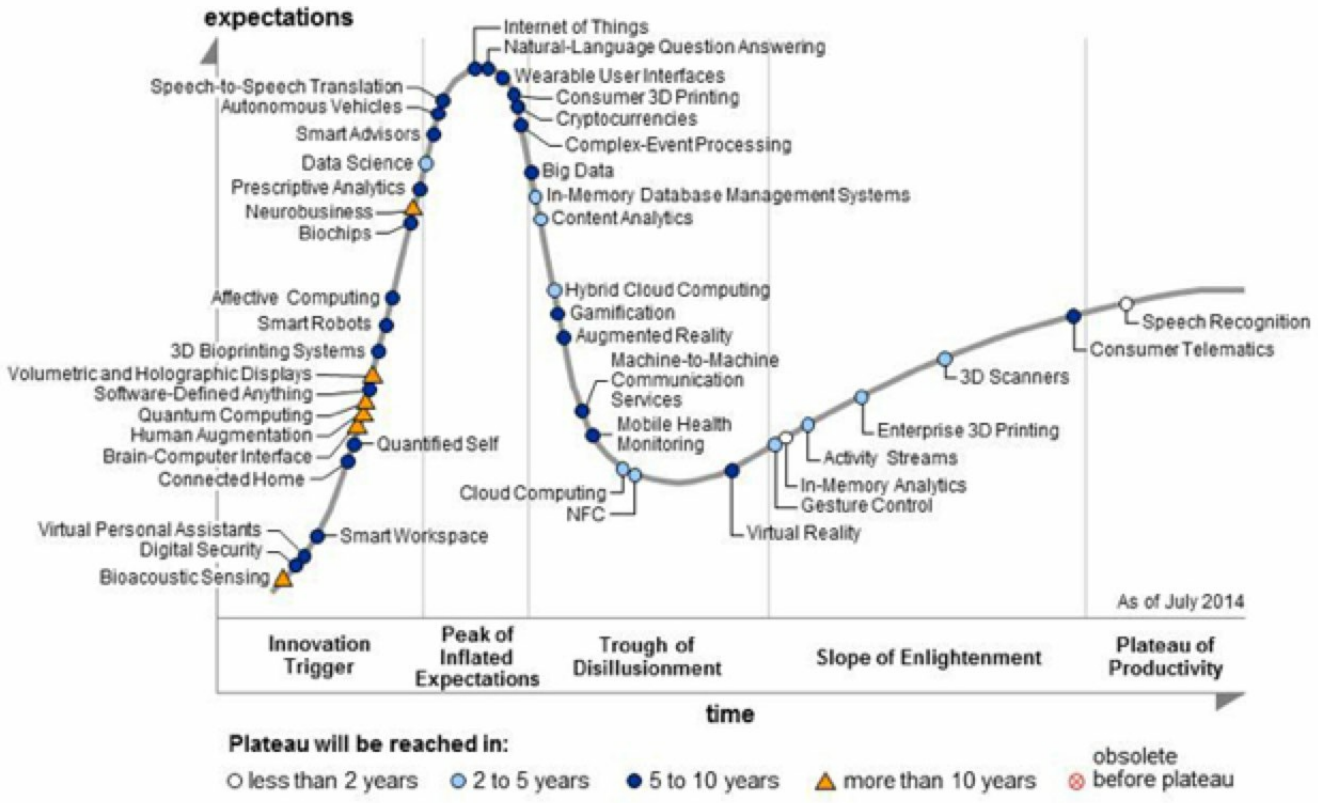


Figure 7: Emerging technologies hype cycle 2014

At the same time, Gartner hype cycle for Big Data (Figure 8) provides a more detailed view on the areas development inside of Big Data topic.

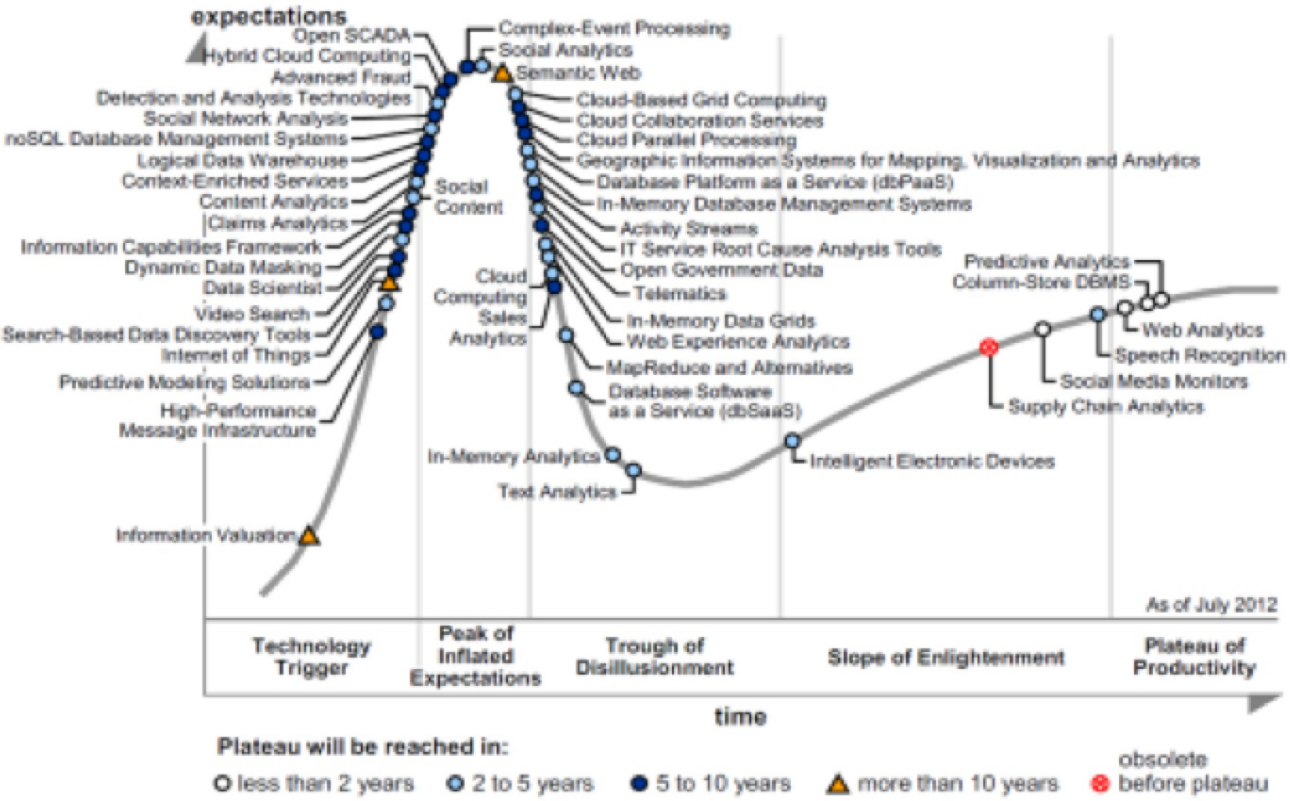


Figure 8: Gartner hype cycle for Big Data 2012

The motivation behind usage of Big Data lies in the increased possibilities to handle and analyse the available data. Key enablers for the appearance and growth of Big Data are:

Increase of storage capacities

Increase of processing power

Availability of data.

Data storage has grown significantly, shifting markedly from analog to digital after 2000 as shown in Figure 9. Computational capacity has also risen as shown in Figure 10.

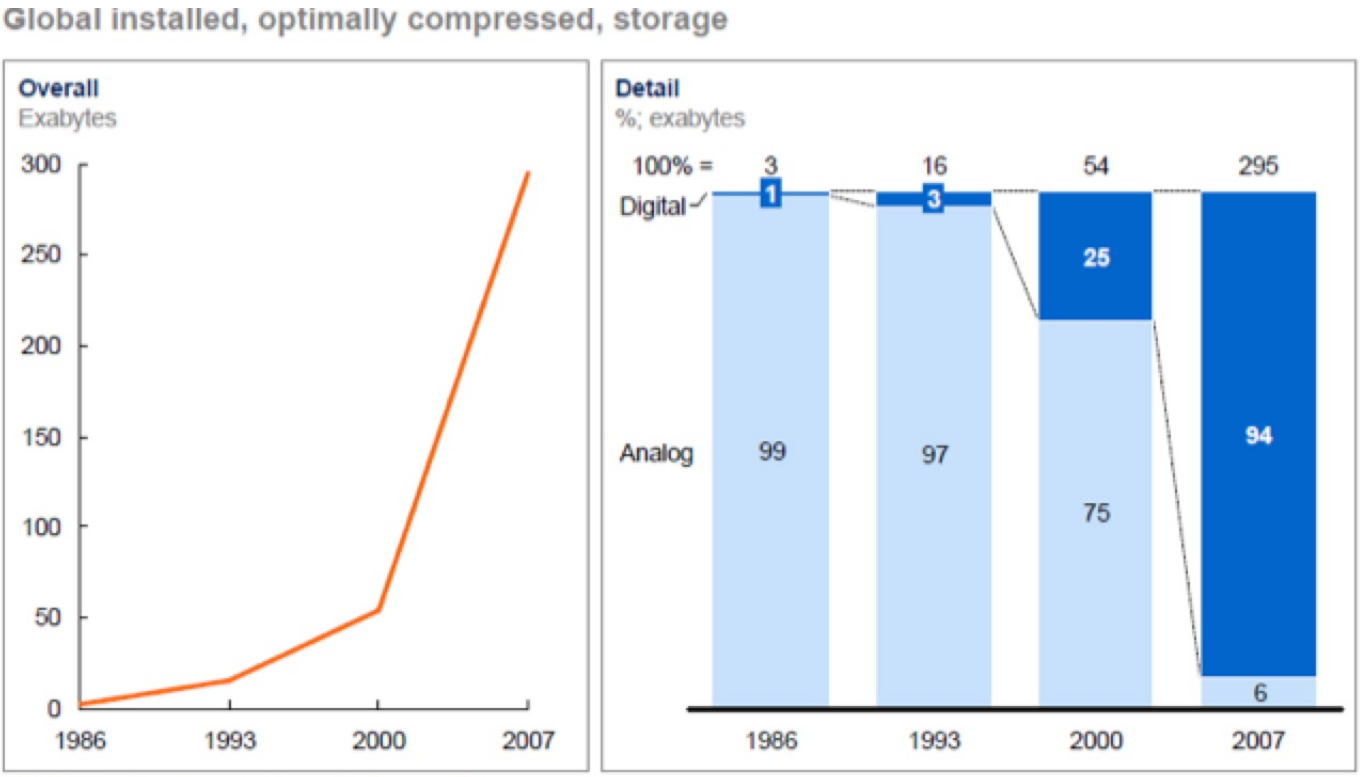


Figure 9: Enabled data storage

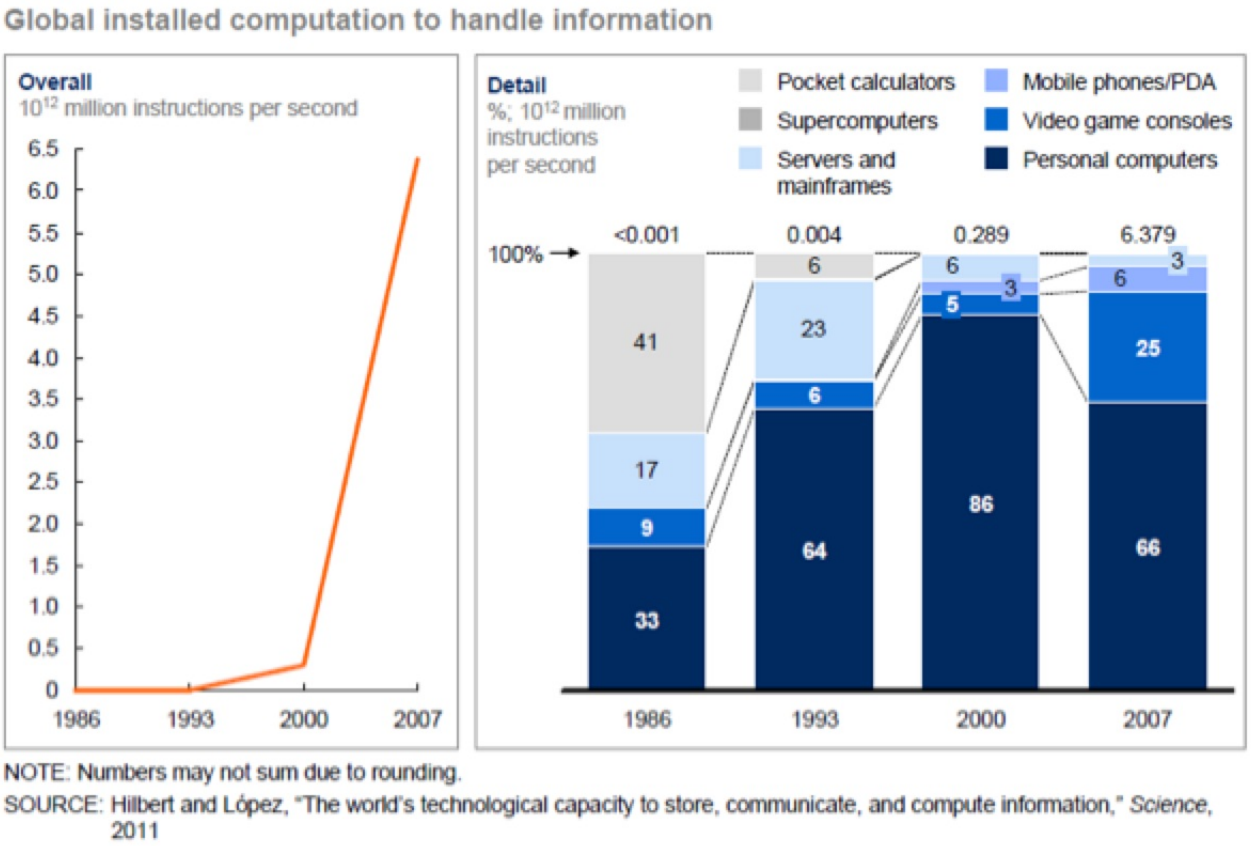
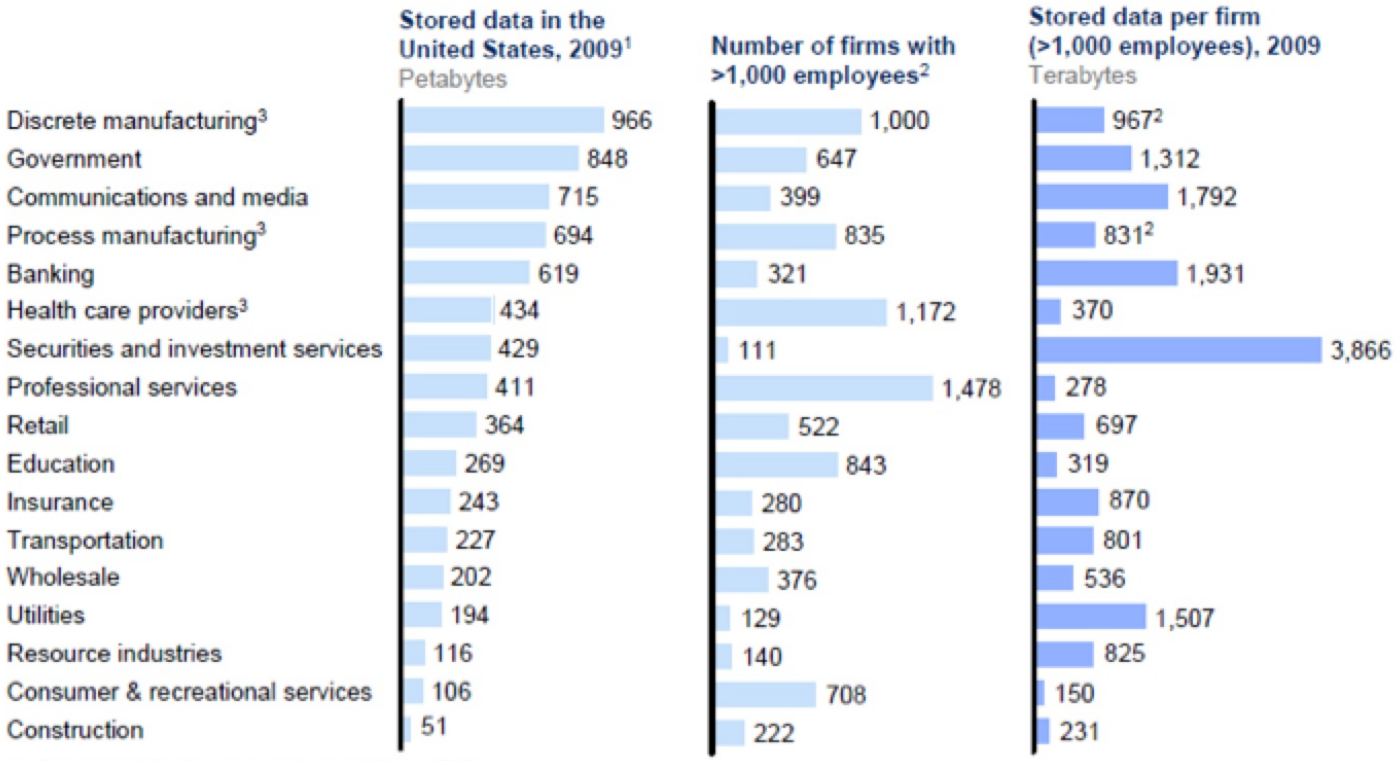


Figure 10: Enabled computation capacity

Companies in all sectors have at least 100 terabytes of stored data in the United States; many have more than 1 petabyte, as shown in Figure 11.



1 Storage data by sector derived from IDC.
2 Firm data split into sectors, when needed, using employment
3 The particularly large number of firms in manufacturing and health care provider sectors make the available storage per company much smaller.

SOURCE: IDC; US Bureau of Labor Statistics; McKinsey Global Institute analysis

Figure 11:

Enabled data availability

Different types of data (video, image, audio, text/numbers) are generated and stored by the variety of sectors: Banking, Insurance, Retail, Health care, Transportation etc.

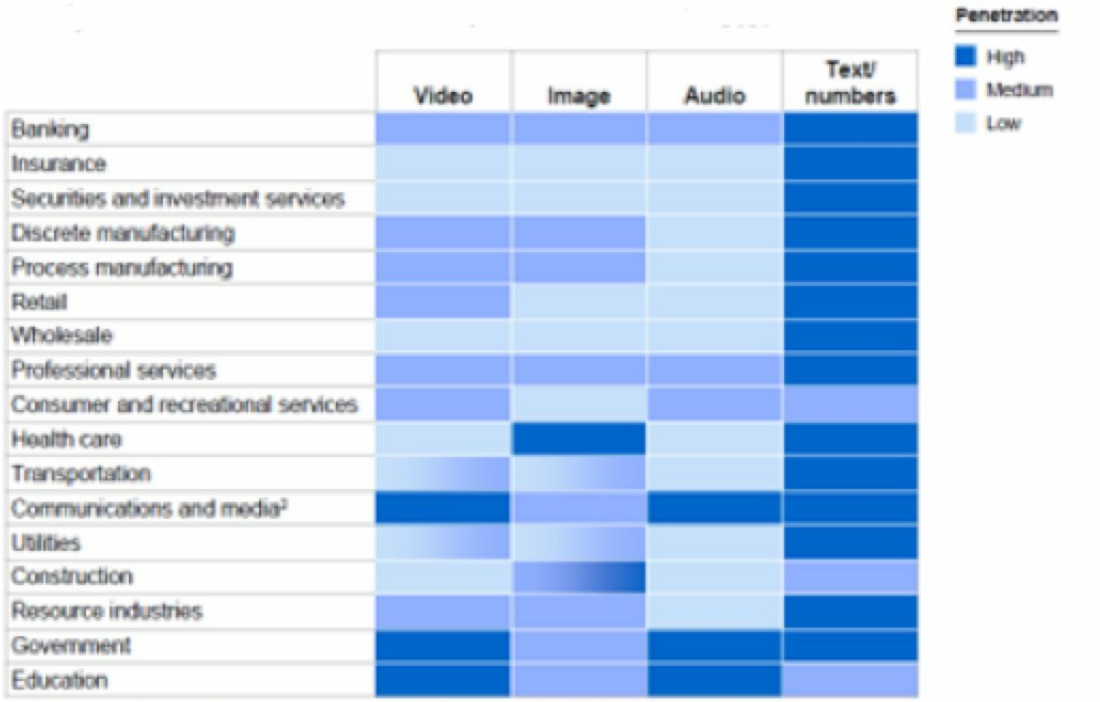


Figure 12: Types of available data

The usage of social networks is increasing – social networks are more frequently used via mobile devices. The number of frequent users is increasing as well.

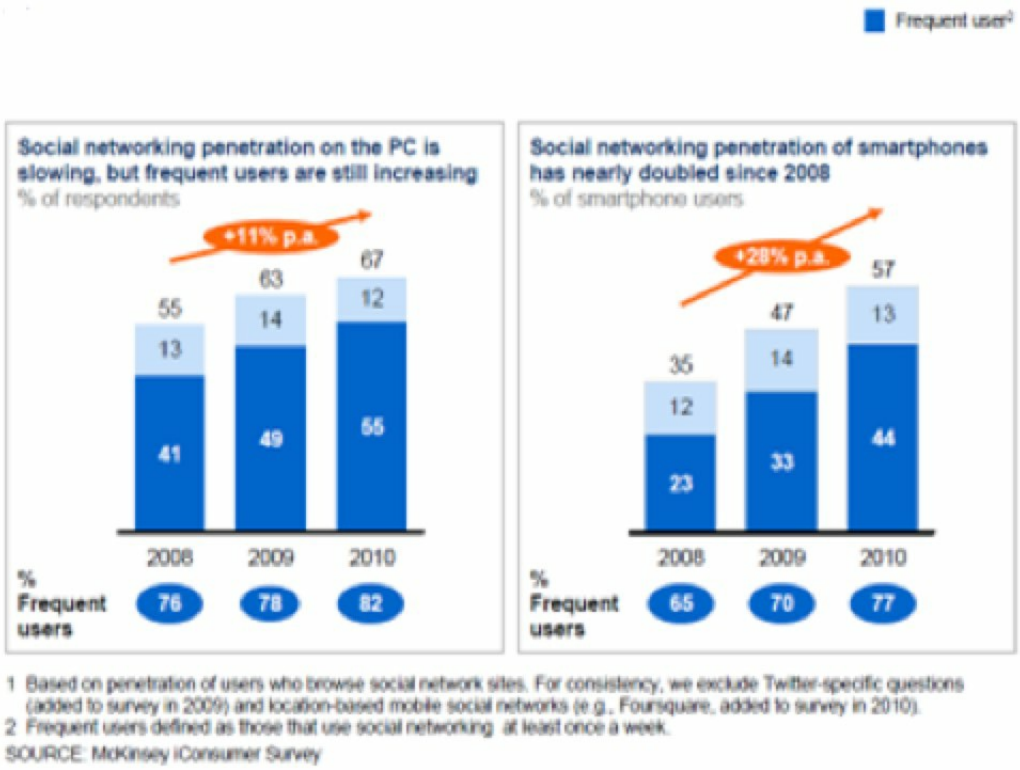


Figure 13: Data available from social networks and mobile devices

Data generated from the Internet of Things will grow exponentially as the number of connected nodes increases, as shown in Figure 14.

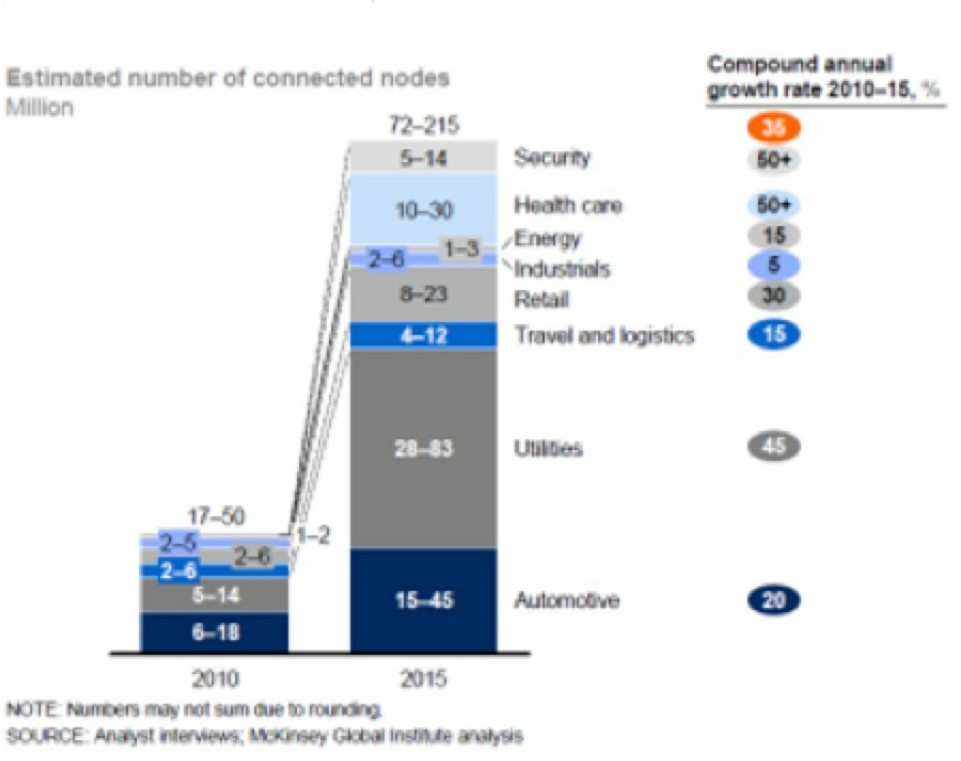


Figure 14: Data available from “Internet of Things”

Figure 15 presents Big Data activity/value chain, while Figure 16 provides gains from Big Data per sector.

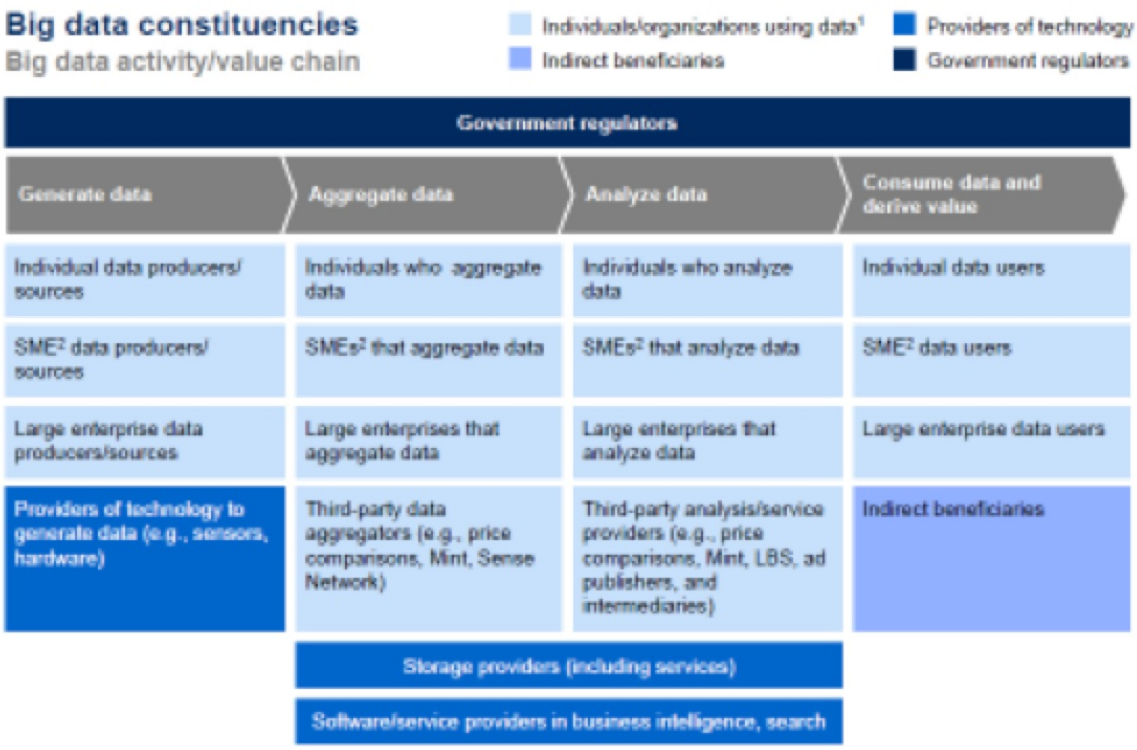


Figure 15: Big Data value chain

Figure 16: Gains from Big Data per sector

As predicted and visualized on Figure 17, the demand for Big Data specialists will increased in the next couple of years.

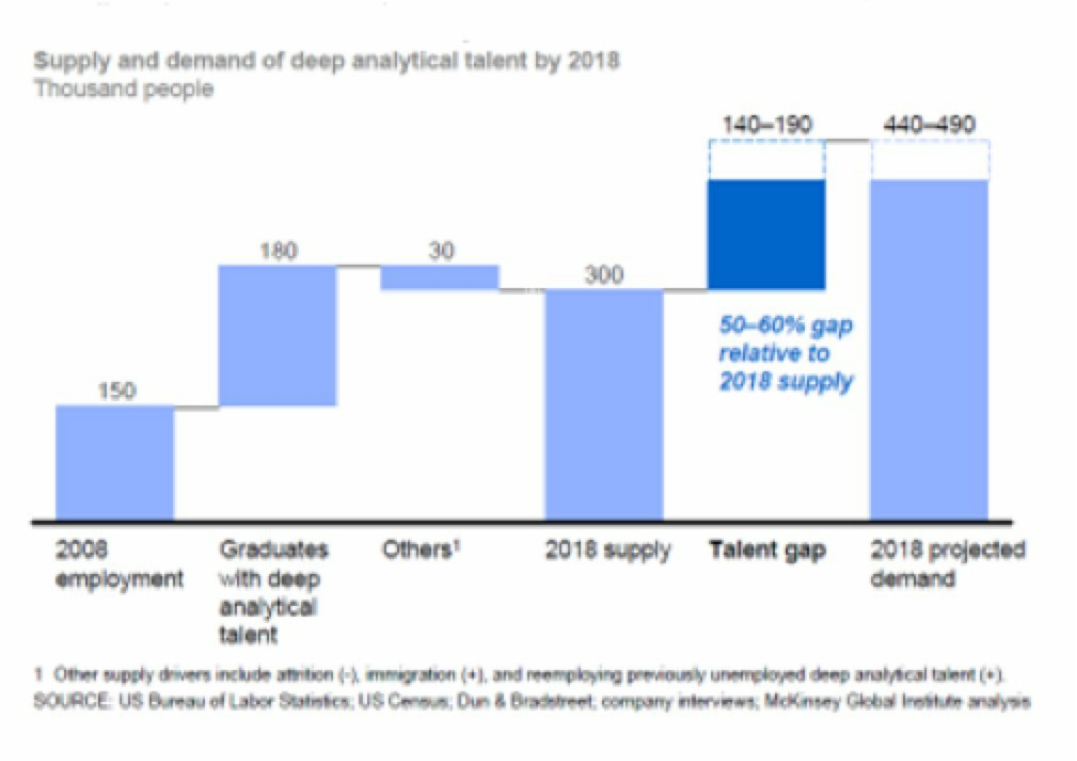


Figure 17: Predicted lack of talent for Big Data related technologies

Looking into the Big Data market, it is possible to notice that Big Data revenues already constitute substantial percentage for some of the world largest IT companies.

2012 Worldwide Big Data Revenue by Vendor ($US millions)

Vendor	Big Data Revenue	Total Revenue	Big Data Revenue as % of Total Revenue	% Big Data Hardware Revenue	% Big Data Software Revenue	% Big Data Services Revenue
IBM	$1,352	$103,930	1%	22%	33%	44%
HP	$664	$119,895	1%	34%	29%	38%
Teradata	$435	$2,665	16%	31%	28%	41%
Dell	$425	$59,878	1%	83%	0%	17%
Oracle	$415	$39,463	1%	25%	34%	41%
SAP	$368	$21,707	2%	0%	67%	33%
EMC	$336	$23,570	1%	24%	36%	39%
Cisco Systems	$214	$47,983	0%	80%	0%	20%
Microsoft	$196	$$71,474	0%	0%	67%	33%
Accenture	$194	$29,770	1%	0%	0%	100%
Fusion-io	$190	$439	43%	71%	0%	29%
PwC	$189	$31,500	1%	0%	0%	100%
SAS Institute	$187	$2,954	6%	0%	59%	41%

Source: WikiBon report on "**Big Data Vendor Revenue and Market Forecast 2012-2017**", 2013

Figure 18: Worldwide Big Data revenue by vendor ($US millions, 2012)

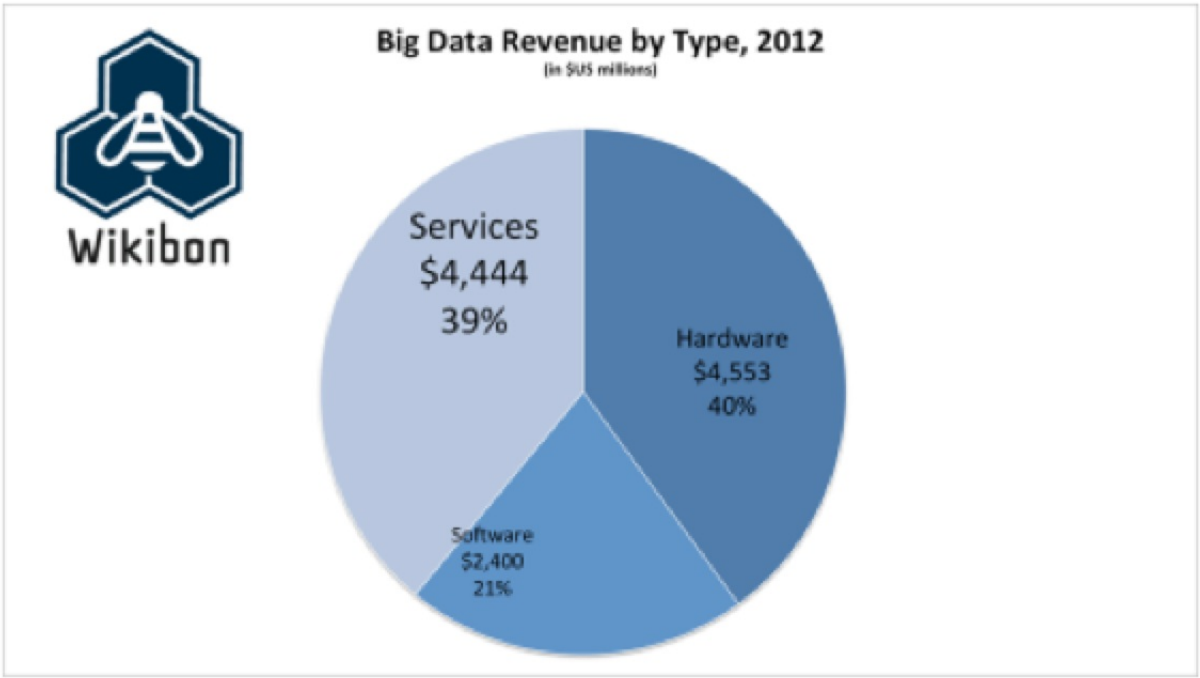
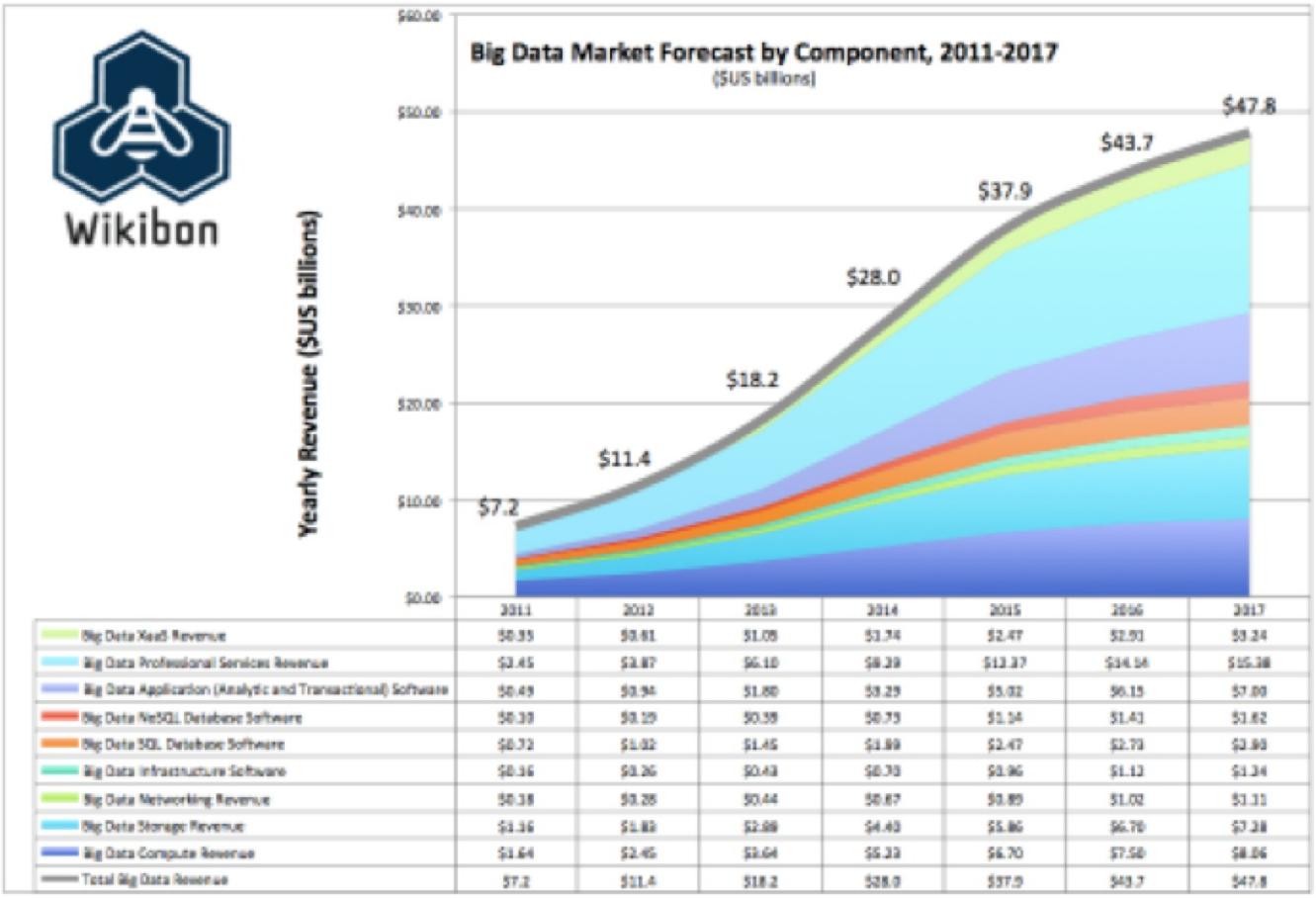


Figure 19: Big Data revenue by type, 2012

Figure 20 shows a forecast for Big Data market for 2011-2017. According to this forecast, the revenues in many

areas connected to Big Data topic are going to increase substantially.



	2011	2012	2013	2014	2015	2016	2017
Big Data Hadoop Revenue	$0.35	$0.61	$1.05	$1.74	$2.47	$2.91	$3.24
Big Data Professional Services Revenue	$3.45	$3.87	$6.10	$8.29	$12.37	$14.14	$15.38
Big Data Application (Analytic and Transactional) Software	$0.49	$0.94	$1.80	$3.29	$5.02	$6.15	$7.00
Big Data NoSQL Database Software	$0.33	$0.19	$0.38	$0.73	$1.14	$1.41	$1.62
Big Data SQL Database Software	$0.72	$1.02	$1.45	$1.89	$2.47	$2.73	$2.93
Big Data Infrastructure Software	$0.16	$0.26	$0.43	$0.73	$0.96	$1.12	$1.24
Big Data Networking Revenue	$0.18	$0.28	$0.44	$0.67	$0.89	$1.02	$1.11
Big Data Storage Revenue	$1.16	$1.83	$2.88	$4.43	$5.86	$6.78	$7.28
Big Data Compute Revenue	$1.64	$2.45	$3.64	$5.23	$6.70	$7.50	$8.06
Total Big Data Revenue	$7.2	$11.4	$18.2	$28.0	$37.9	$43.7	$47.8

Figure 20: Big Data market forecast, 2011-2017

Techniques & Tools Overview

This section is dedicated to the analytic techniques that are used for operation with Big Data. The specific analytical operators for Big Data are discussed.

When Big Data is really a hard problem? The Big Data becomes problematic when the when the operations on data are complex (e.g. simple counting is not a complex problem).

Modeling and reasoning with data of different kinds can get extremely complex. Often, because of vast amount of data, modeling techniques can get simpler (e.g. smart counting can replace complex model-based analytics) as long as we deal with the scale.

What matters when dealing with data? A number of operators (collect, prepare, represent, model, reason, visualize), a number of data modalities (ontologies, structure, networks, text, multimedia, signals) and a number of additional issues (usage, quality, context, streaming, scalability) can be considered, as shown in Figure 21.

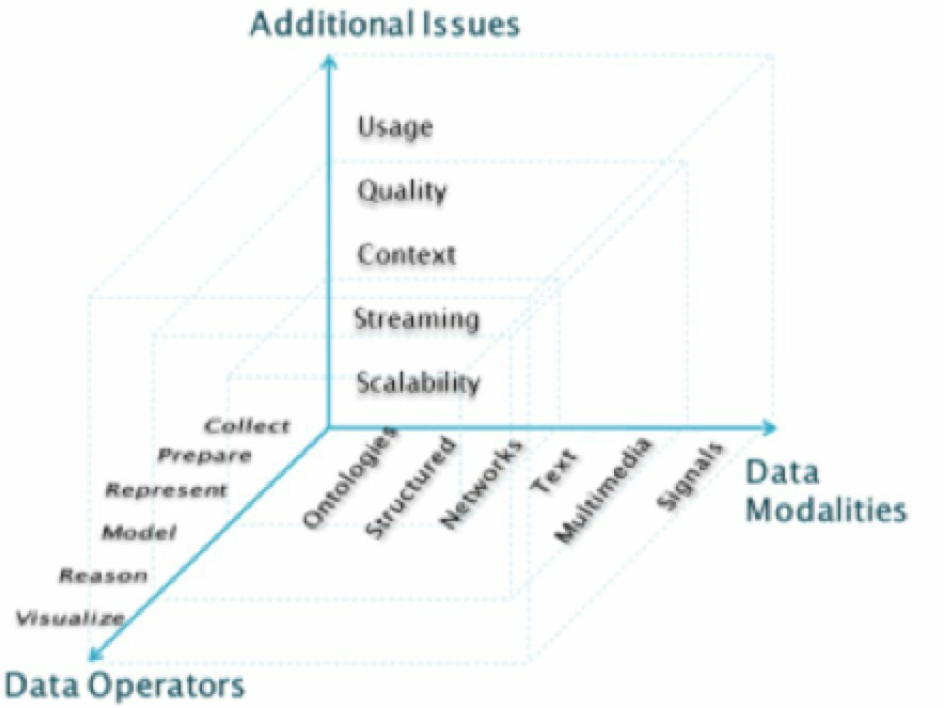


Figure 21: Data operators, Data modalities, Additional issues

At the same time, there are a number of risks to take into account. A risk with “Big Data mining” is that an analyst can “discover” patterns that are meaningless. Statisticians call it Bonferroni’s principle. Roughly, if you look in more places for interesting patterns than your amount of data will support, you are bound to find crap.

For example, we want to find (unrelated) people who at least twice have stayed at the same hotel on the same day

- 10^9 people being tracked.
- 1000 days.
- Each person stays in a hotel 1% of the time (1 day out of 100)

- Hotels hold 100 people (so 10^5 hotels).
- If everyone behaves randomly (i.e., no terrorists) will the data mining detect anything suspicious?

Expected number of “suspicious” pairs of people:

- 250,000
- … too many combinations to check – we need to have some additional evidence to find “suspicious” pairs of people in some more efficient way

Example taken from: Rajaraman, Ullman: Mining of Massive Datasets

There are specific operators used in Big Data applications:

Smart sampling of data
- …reducing the original data while not losing the statistical properties of data

Finding similar items
- …efficient multidimensional indexing

Incremental updating of the models
- (vs. building models from scratch)
- …crucial for streaming data

Distributed linear algebra
- …dealing with large sparse matrices

Analytical operators on Big Data are:

On the top of the previous ops we perform usual data mining/machine learning/statistics operators:
- Supervised learning (classification, regression, …)
- Non-supervised learning (clustering, different types of decompositions, …)

We are just more careful which algorithms we choose
- typically linear or sub-linear versions of the algorithms

Tools Overview

This section provides a view on types of tools that are used for Big Data. Distributed infrastructure and Distributed processing are discussed. A particular attention is given to MapReduce, NoSQL databases. Open source Big Data tools are listed.

Types of tools typically used in Big Data scenarios:

Where processing is hosted?
- Distributed Servers / Cloud (e.g. Amazon EC2)

Where data is stored?
- Distributed Storage (e.g. Amazon S3)

What is the programming model?
- Distributed Processing (e.g. MapReduce)

How data is stored & indexed?

- High-performance schema-free databases (e.g. MongoDB)

What operations are performed on data?

- Analytic / Semantic Processing

Figure 22 presents a landscape of “Big Data” related tools.

Figure 22: Big Data tools

Distributed infrastructure is often used for Big Data analytics:

Computing and storage are typically hosted transparently on cloud infrastructures

- ...providing scale, flexibility and high fail-safety

Distributed Servers

- Amazon-EC2, Google App Engine, Elastic, Beanstalk, Heroku

Distributed Storage

- Amazon-S3, Hadoop Distributed File System

MapReduce is an approach for processing and generating large datasets. The key ideas of the MapReduce approach are the following:

A target problem needs to be parallelizable

First, the problem gets split into a set of smaller problems (Map step)

Next, smaller problems are solved in a parallel way

Finally, a set of solutions to the smaller problems get synthesized into a solution of the original problem (Reduce step)

Figure 23 demonstrates the MapReduce approach.

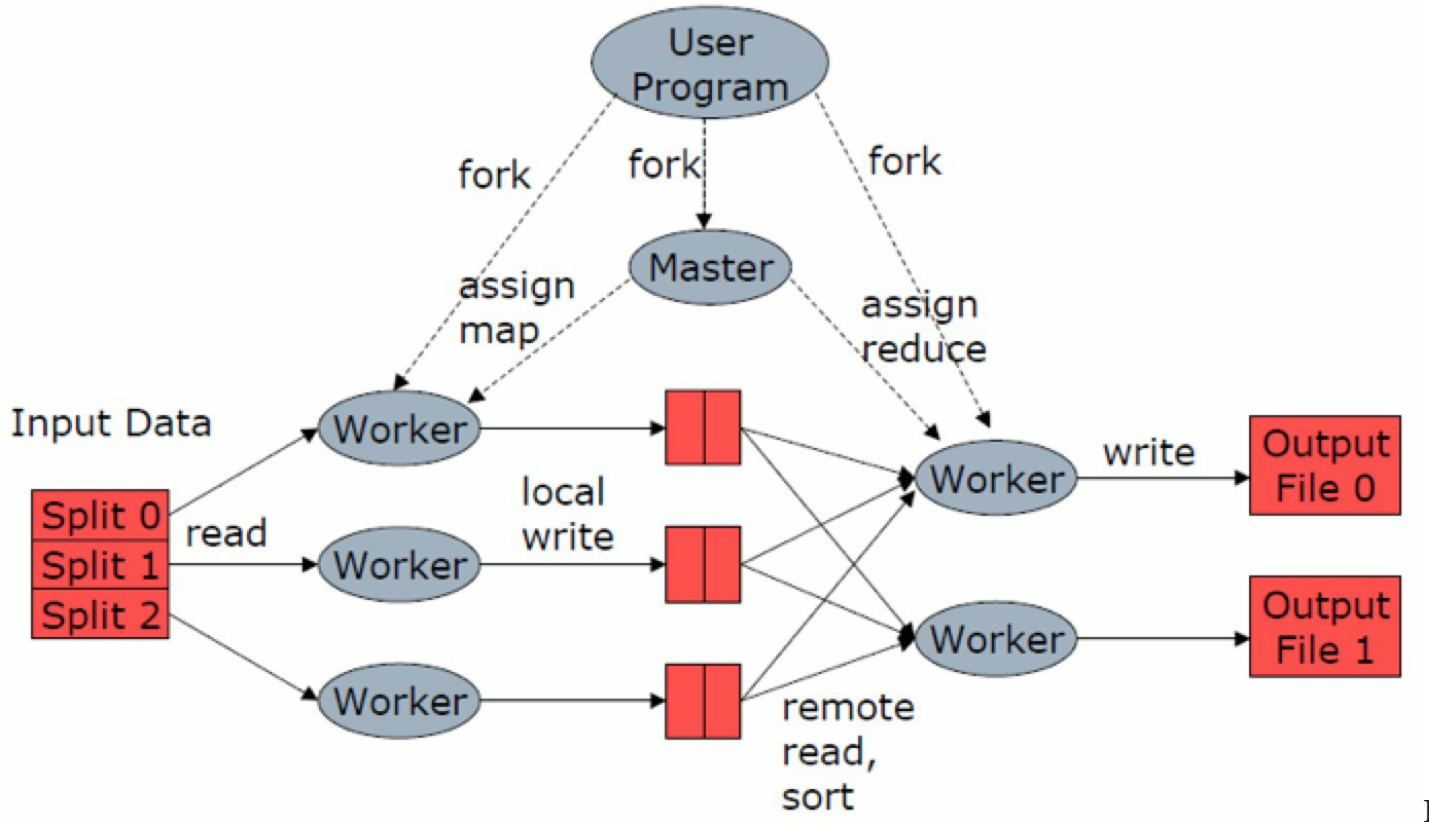


Figure 23: MapReduce approach

Figures 24-28 provide a number of MapReduce examples.

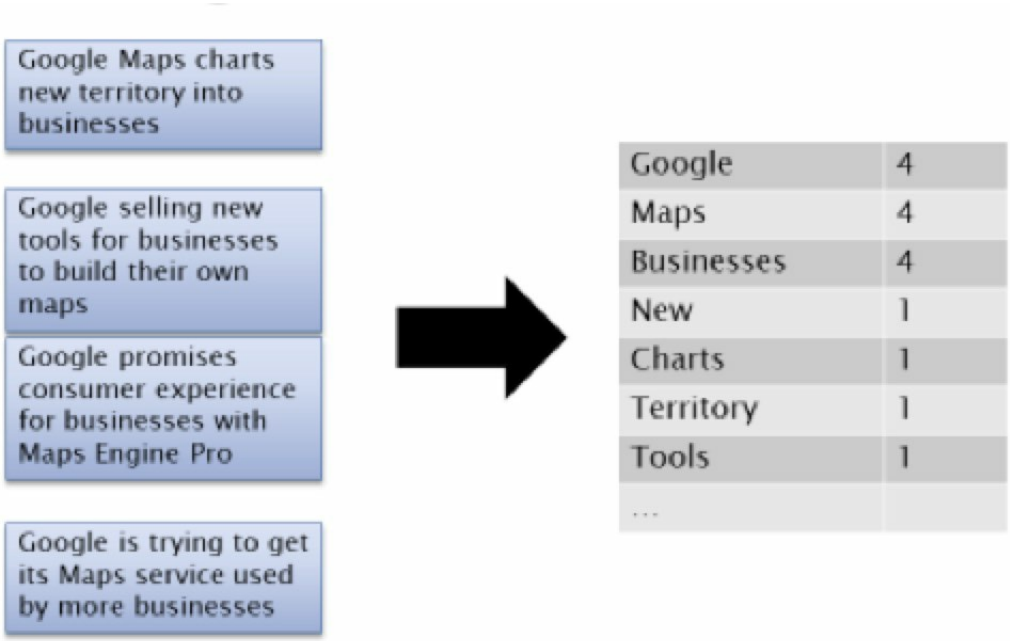


Figure 24: MapReduce example: counting words in documents

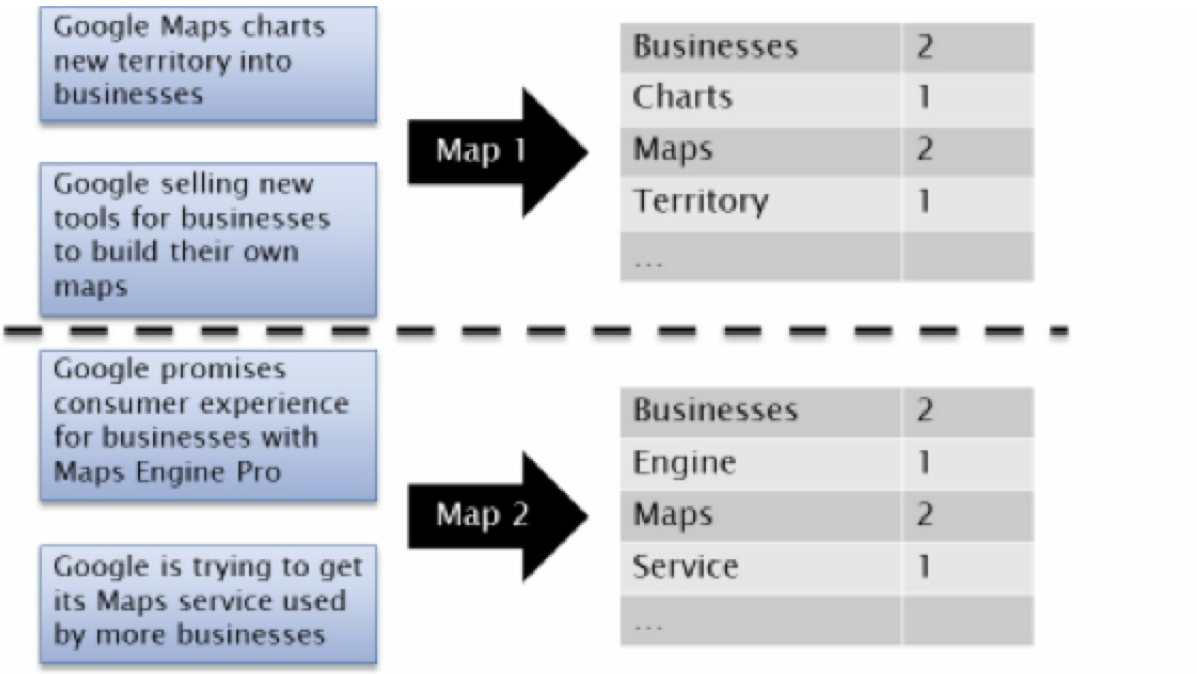


Figure 25: MapReduce example: Map task

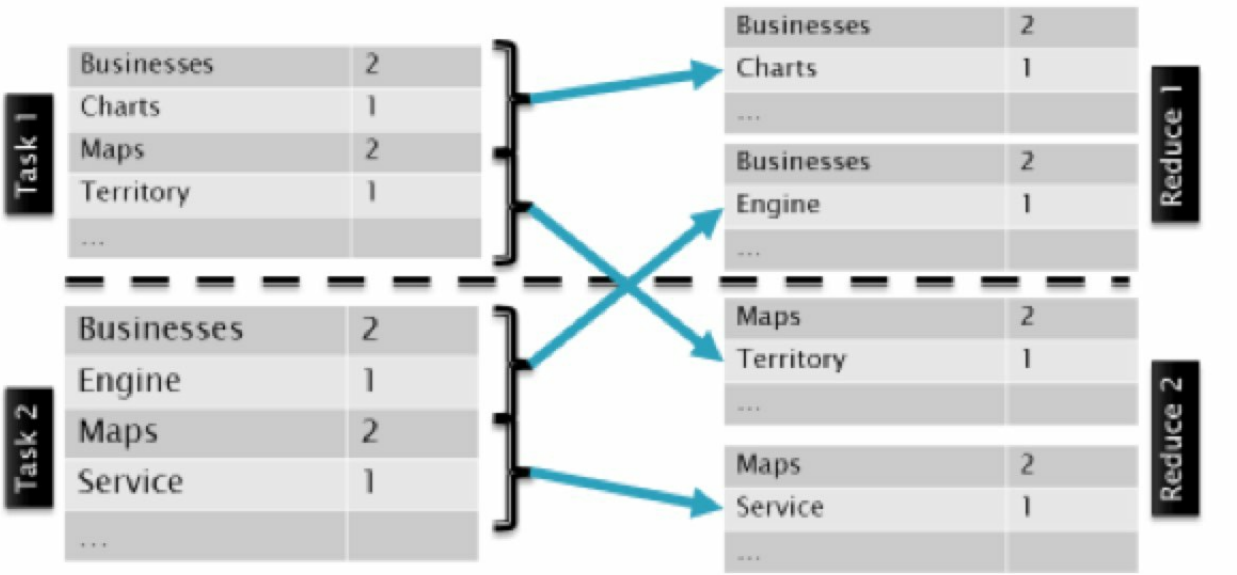


Figure 26: MapReduce example: Group and aggregate

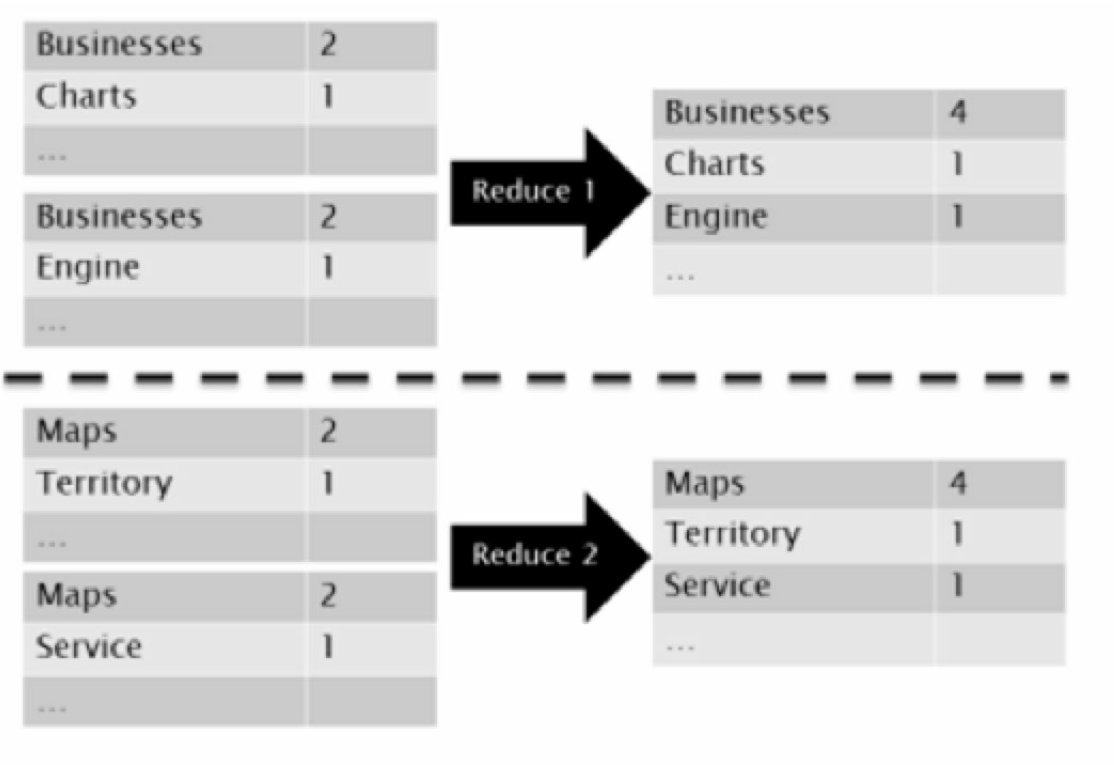


Figure 27: MapReduce example: Reduce task

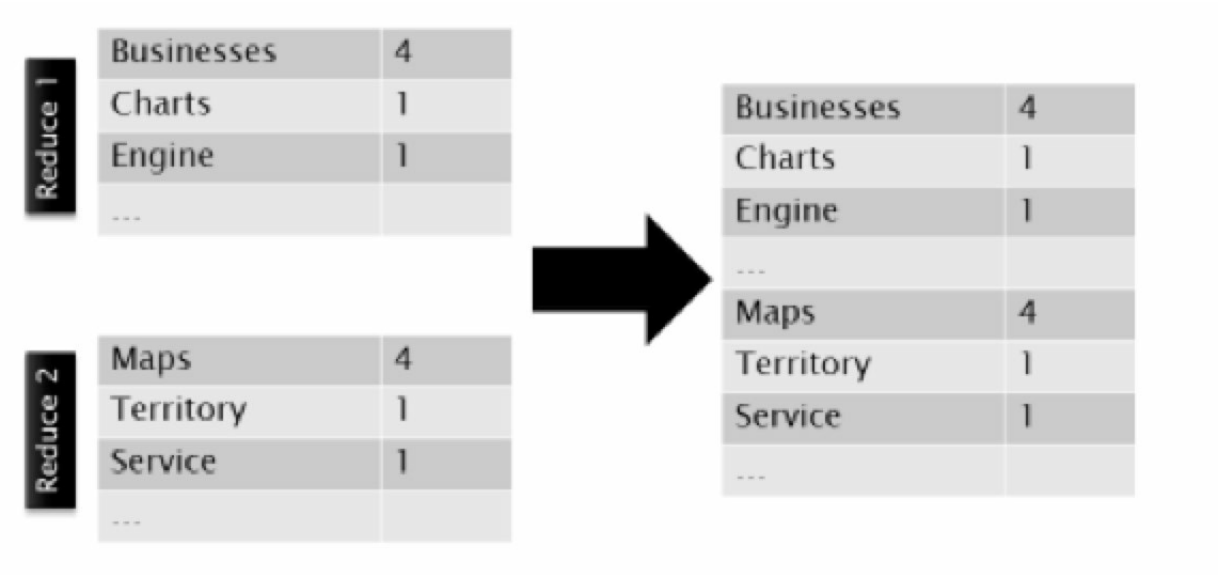


Figure 28: MapReduce example: Combine

There are a number of *MapReduce* tools available:

Apache Hadoop

- Open-source MapReduce implementation

Tools using Hadoop:

- Hive: data warehouse infrastructure that provides data summarization and ad hoc querying (HiveQL)

- Pig: high-level data-flow language and execution framework for parallel computation (Pig Latin)
- Mahout: Scalable machine learning and data mining library
- Flume: Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data
- Many more: Cascading, Cascalog, mrjob, MapR, Azkaban, Oozie, ...

NoSQL databases are used for "[...] need to solve a problem that relational databases are a bad fit for", Eric Evans

Motives behind NoSQL databases are the following:

- Avoidance of Unneeded Complexity – many use-case require only subset of functionality from RDBMSs (e.g ACID properties)
- High Throughput - some NoSQL databases offer significantly higher throughput then RDBMSs
- Horizontal Scalability, Running on commodity hardware
- Avoidance of Expensive Object-Relational Mapping – most NoSQL store simple data structures
- Compromising Reliability for Better Performance

[Based on "NoSQL Databases", Christof Strauch]

Basic concepts – *Consistency*:

BASE approach
- Availability, graceful degradation, performance
- Stands for "Basically available, soft state, eventual consistency"

Continuum of tradeoffs:
- Strict – All reads must return data from latest completed writes
- Eventual – System eventually return the last written value
- Read Your Own Writes – see your updates immediately
- Session – RYOW only within same session
- Monotonic – only more recent data in future requests

Basic concepts – *Partitioning*:

Consistent hashing
- Use same function for hashing objects and nodes
- Assign objects to nearest nodes on the circle
- Reassign object when nodes added or removed

- Replicate nodes to r nearest nodes

[White, Tom: Consistent Hashing. November 2007. – Blog post of 2007-11-27]

Other basic concepts:

Storage Layout
- Row-based
- Columnar
- Columnar with Locality Groups

Query Models
- Lookup in key-value stores

Distributed Data Processing via MapReduce

[Lipcon, Todd: Design Patterns for Distributed Non-Relational Databases. June 2009. – Presentation of 2009-06-11]

Key-Value Stores:

Map or dictionary allowing to add and retrieve values per keys

Favor scalability over consistency
- Run on clusters of commodity hardware
- Component failure is “standard mode of operation”

Examples:
- Amazon Dynamo
- Project Voldemort (developed by LinkedIn)
- Redis
- Memcached (not persistent)

Document Databases:

Combine several key-value pairs into documents

Documents represented as JSON

Examples:
- Apache CouchDB
- MongoDB

Column-Oriented:

Using columnar storage layout with locality groups (column families)

Examples:

- Google Bigtable
- Hypertable, HBase
  - open source implementation of Google Bigtable
- Cassandra
  - combination of Google Bigtable and Amazon Dynamo
  - Designed for high write throughput

A number of Big Data open source tools are the following:

Infrastructure:

Kafka
- A high-throughput distributed messaging system

Hadoop
- Open-source map-reduce implementation

Storm
- Real-time distributed computation system

Cassandra
- Hybrid between Key-Value and Row-Oriented DB
- Distributed, decentralized, no single point of failure
- Optimized for fast writes

Open source Big Data tools for Machine Learning

Mahout

Machine learning library working on top of Hadoop

MOA

Mining data streams with concept drift

Integrated with Weka

Mahout currently has:

Collaborative Filtering

User and Item based recommenders

K-Means, Fuzzy K-Means clustering

Mean Shift clustering

Dirichlet process clustering

Latent Dirichlet Allocation

Singular value decomposition

Parallel Frequent Pattern mining

Complementary Naive Bayes classifier

Random forest decision tree based classifier

Applications

This section discusses a number of Big Data applications, such as

Recommendation

Social Networks

Media Monitoring

Recommendation is one type of applications that benefit from the Big Data usage. Possible data in recommendation applications are:

User visit logs
- Track each visit using embedded JavaScript

Content
- The content and metadata of visited pages

Demographics
- Metadata about (registered) users

Visit log example:

```
User ID cookie: 1234567890
IP: 95.87.154.251 (Ljubljana, Slovenia)
Requested URL: http://www.bloomberg.com/news/2012-
07-19/americans-hold-dimmest-view-on-economic-
outlook-since-january.html
Referring URL: http://www.bloomberg.com/
Date and time: 2009-08-25 08:12:34
Device: Chrome, Windows, PC
```

Content example:

News-source:
- www.bloomberg.com

Article URL:
- http://www.bloomberg.com/news/2011-01-17/video-gamers-prolonged-play-raises-risk-of-depression-anxiety-phobias.html

Author:
- Elizabeth Lopatto

Produced at:
- New York

Editor:
- Reg Gale

Publish Date:
- Jan 17, 2011 6:00 AM

Topics:
- U.S., Health Care, Media, Technology, Science

Extracted topics/keywords/entities:

Topics (e.g. DMoz):
- Health/Mental Health/…/Depression
- Health/Mental Health/Disorders/Mood
- Games/Game Studies

Keywords (e.g. DMoz):
- Health, Mental Health, Disorders, Mood, Games, Video Games, Depression, Recreation, Browser Based, Game Studies, Anxiety, Women, Society, Recreation and Sports

Locations:
- Singapore (sws.geonames.org/1880252/)
- Ames (sws.geonames.org/3037869/)

People:
- Duglas A. Gentile

Organizations:
- Iowa State University (dbpediapa.org/resource/ Iowa_State_University)

Pediatrics (journal)

The Demographics example shown in Figure 29 illustrates how to obtain news recommendation based on user context model:

Provided only for registered users
- Only some % of unique users typically register
- Each registered users described with:
- Gender
- Year of birth
- Household income

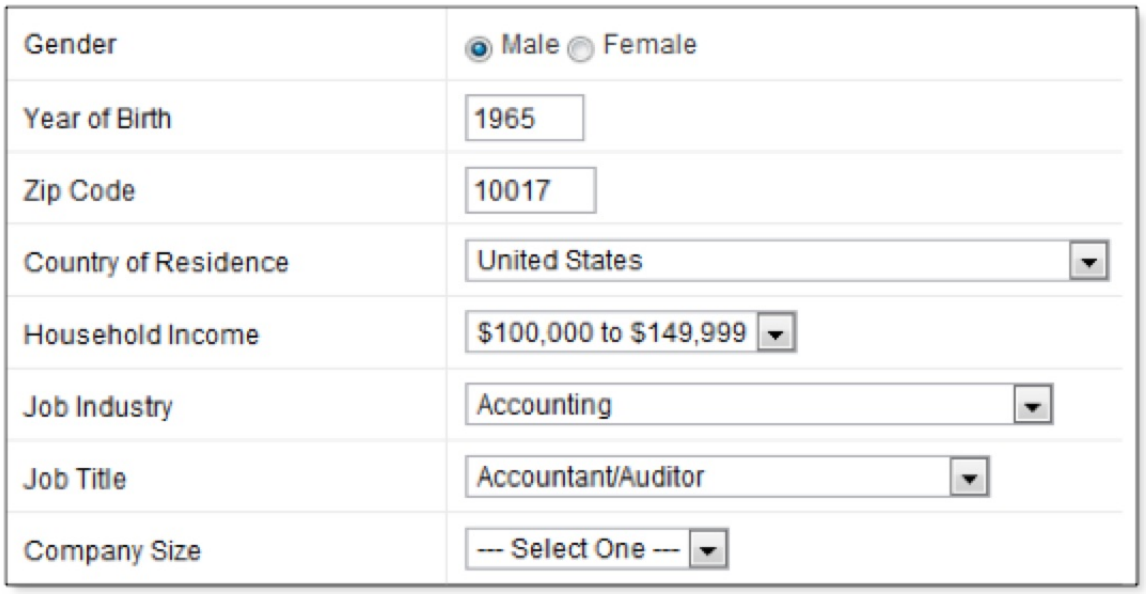


Figure 29: Demographic example

News recommendation - list of articles is based on:

- Current article
- User's history
- Other Visits

In general, a combination of text stream (news articles) with click stream (website access logs). The key is a rich context model used to describe the user.

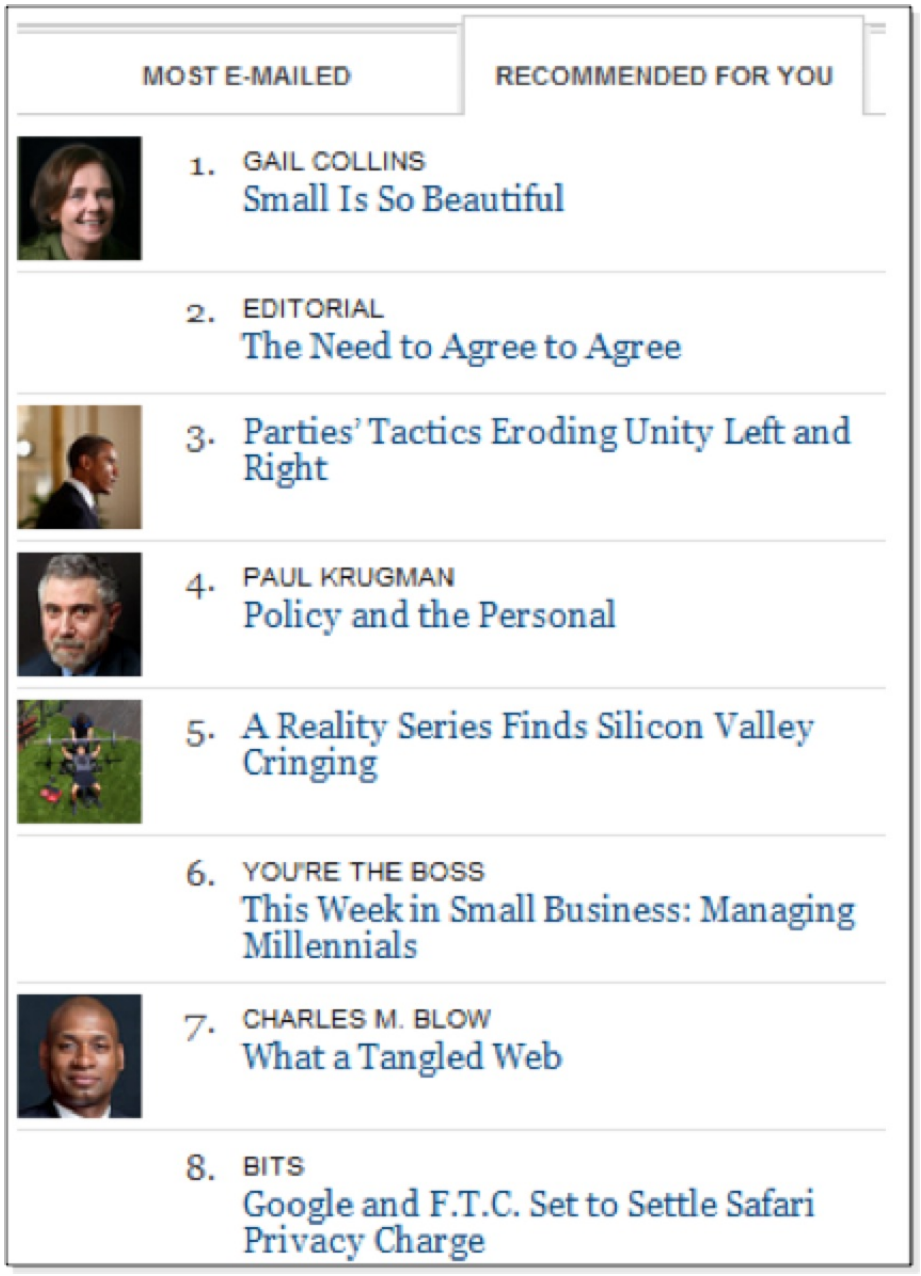


Figure 30: Recommendations

Developing efficient news recommendation services are beneficial in several ways. First of all, they allow for an "increase in engagement". Good recommendations can make a difference when keeping a user on a web site. As well, they improve "user experience" - users return to the site.

However, there are a number of challenges in developing a news recommendation service:

Cold start

- Recent news articles have little usage history
- More sever for articles that did not hit homepage or section front, but are still relevant for particular user segment

Recommendation model must be able to generalize well to new articles.

Example: Bloomberg.com

Access logs analysis shows, that half of the articles read are less then ~8 hours old

Weekends are exception

Figure 31 demonstrates the articles read in a weekend.

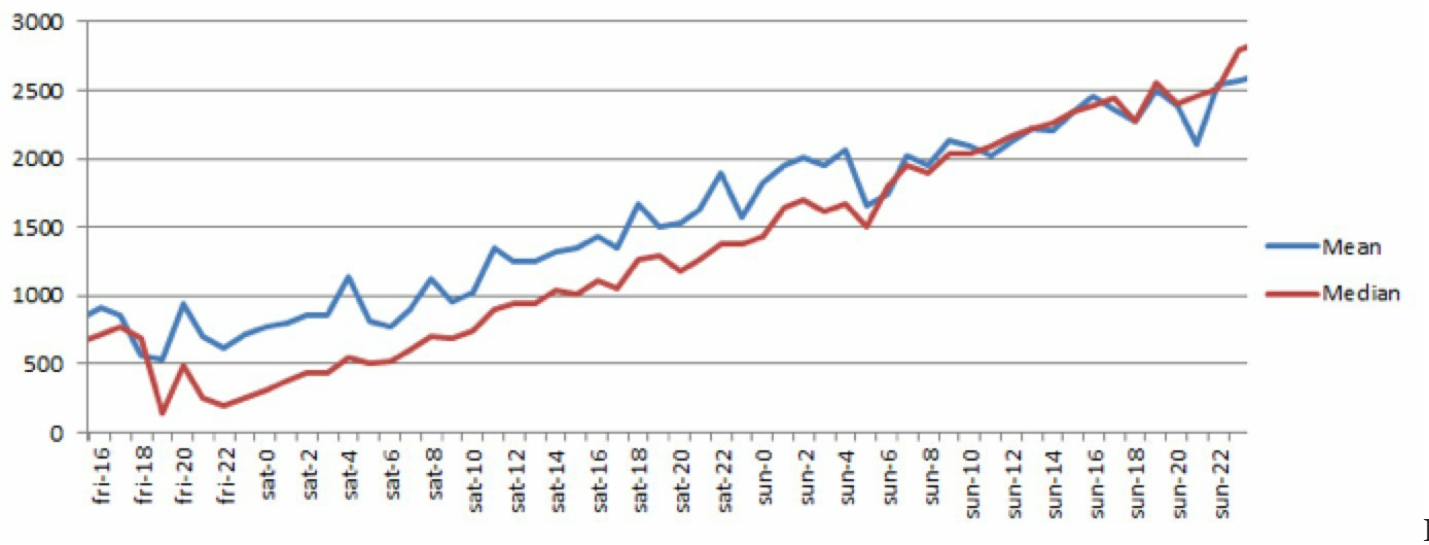


Figure 31: Articles read

The user profile can include history and current requests:

History

- Time
- Article

Current request:

- Location
- Requested page
- Referring page
- Local Time

In addition, each article from the time window is described with the following features:

- Popularity (user independent)
- Content
- Meta-data
- Co-visits
- Users

Features computed by comparing article's and user's feature vectors. Features computed on-the-fly when preparing recommendations.

In an experimental setting, we shall take a real-world dataset from a major news publishing website (5 million

daily users, 1 million registered). We shall test prediction of three demographic dimensions (gender, age, income). Three user groups based on the number of visits: (≥2, ≥10, ≥50)

Evaluation:

- Break Even Point (BEP)
- 10-fold cross validation

Figures 32-34 present the results of a conducted experiment.

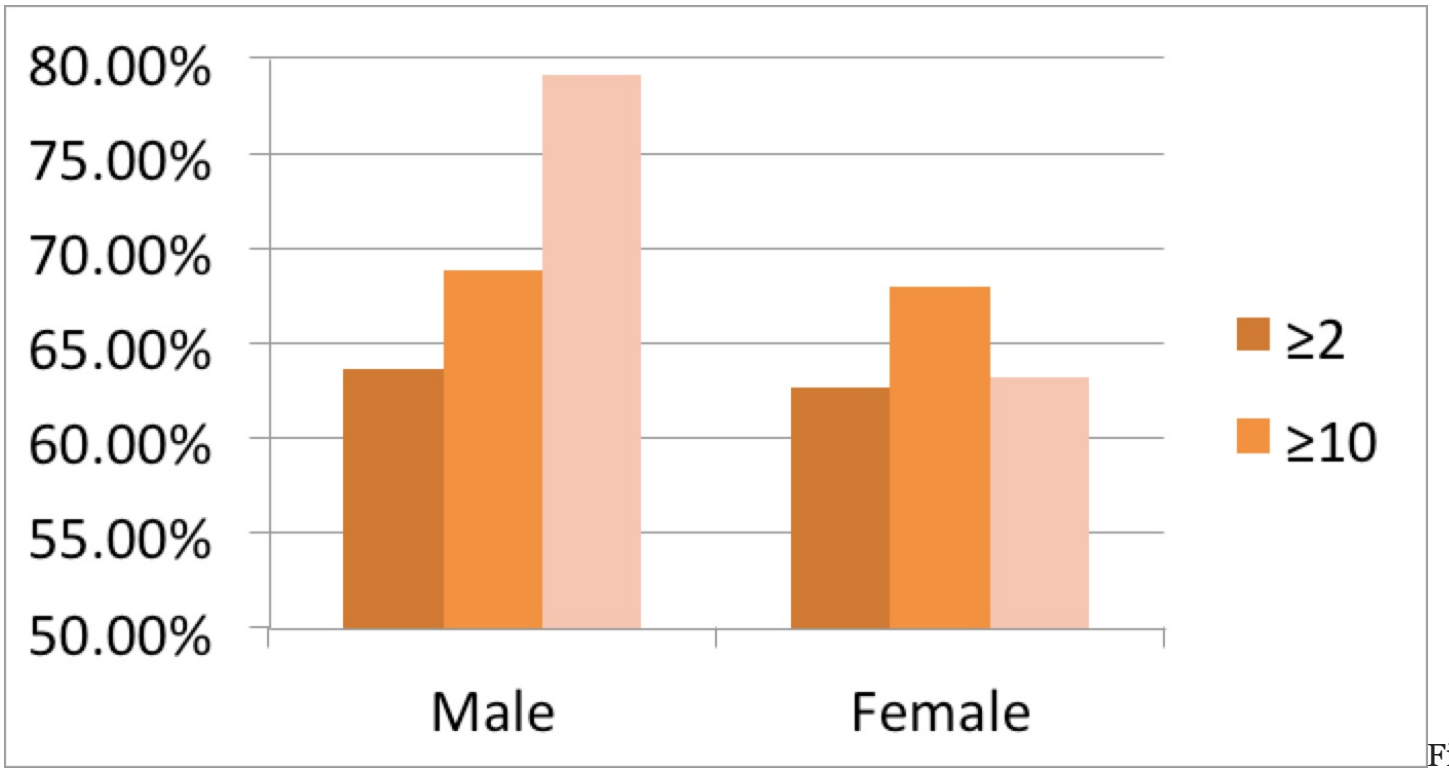


Figure 32: Demographic dimension: Gender

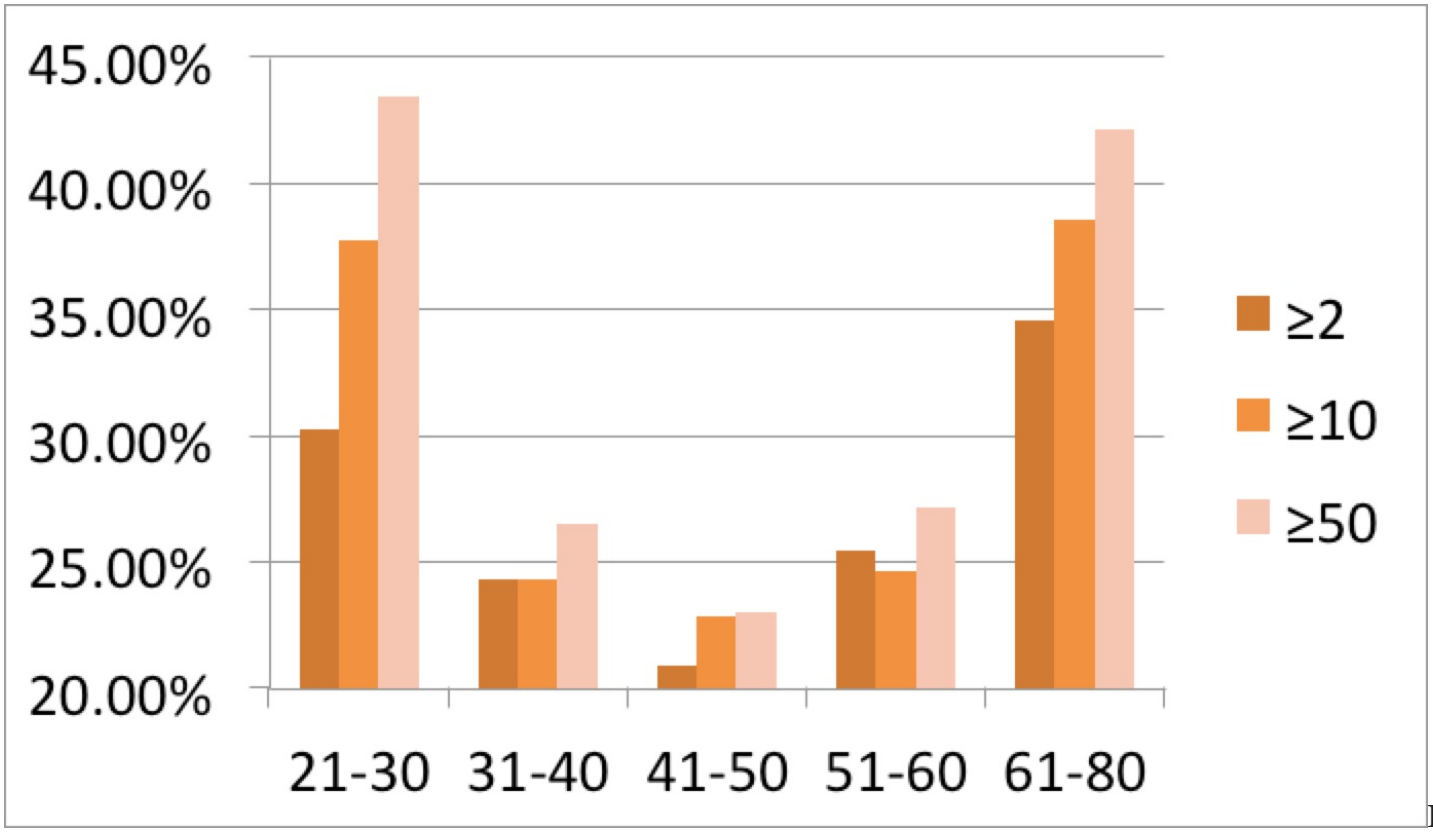


Figure 33: Demographic dimension: Age

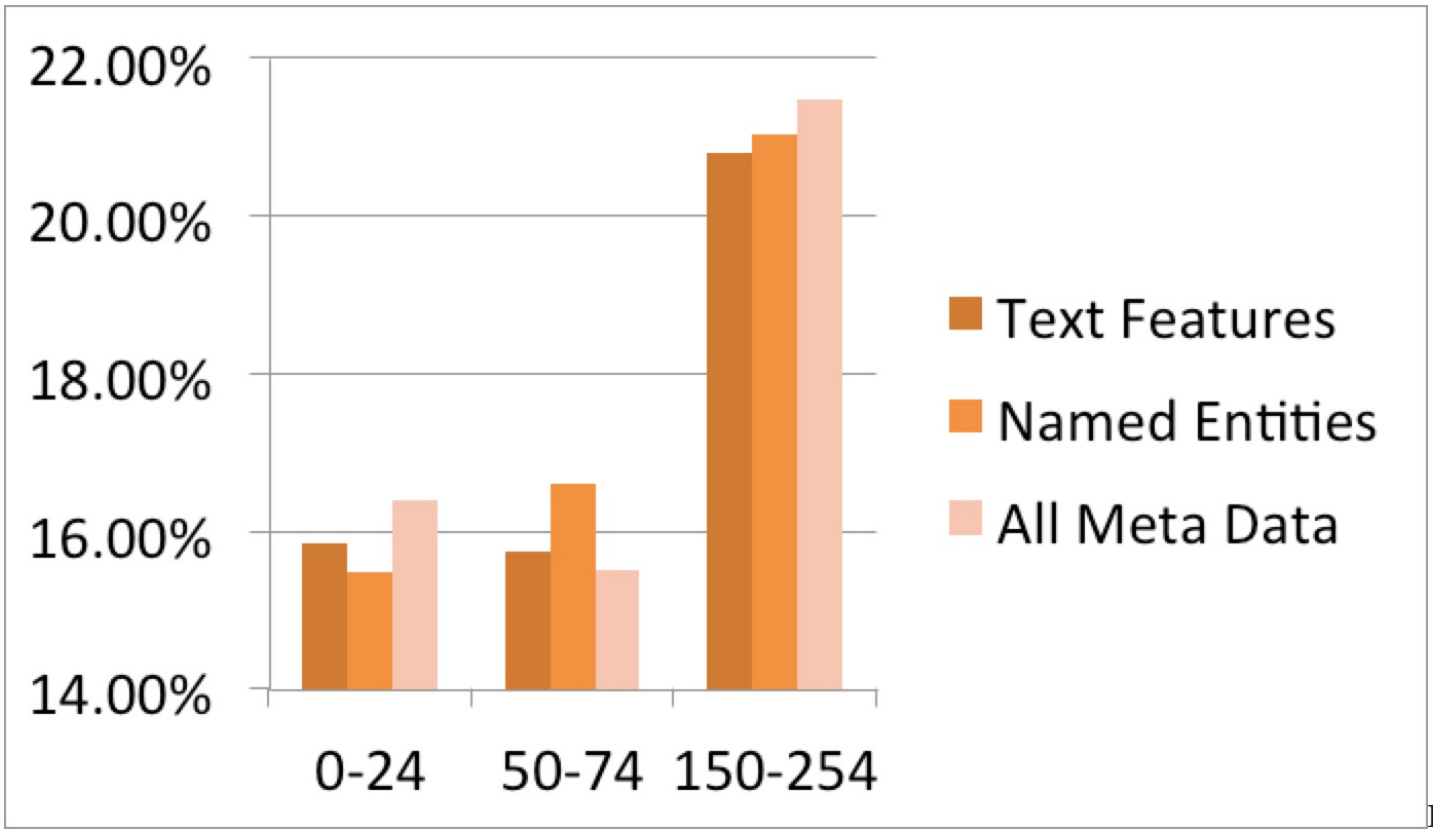


Figure 34: Demographic dimension: Income (≥10)

Social Network Analysis

One of the social network analysis application is the analysis of MSN Messager social network:

Observe social and communication phenomena at a planetary scale

Largest social network analyzed till 2010

Research questions for this task are the following:

How does communication change with user demographics (age, sex, language, country)?

How does geography affect communication?

What is the structure of the communication network?

[Soure: Planetary-Scale Views on a Large Instant-Messaging Network" Leskovec & Horvitz WWW2008]

For this task we collected the data for June 2006. The log size consisted 150Gb/day (compressed) and total: 1 month of communication data was 4.5Tb of compressed data.

Activity over June 2006 (30 days):

- 245 million users logged in
- 180 million users engaged in conversations
- 17,5 million new accounts activated
- More than 30 billion conversations
- More than 255 billion exchanged messages

Figures 35-36 demonstrate who talks to whom in social network (number of conversations and conversation duration).

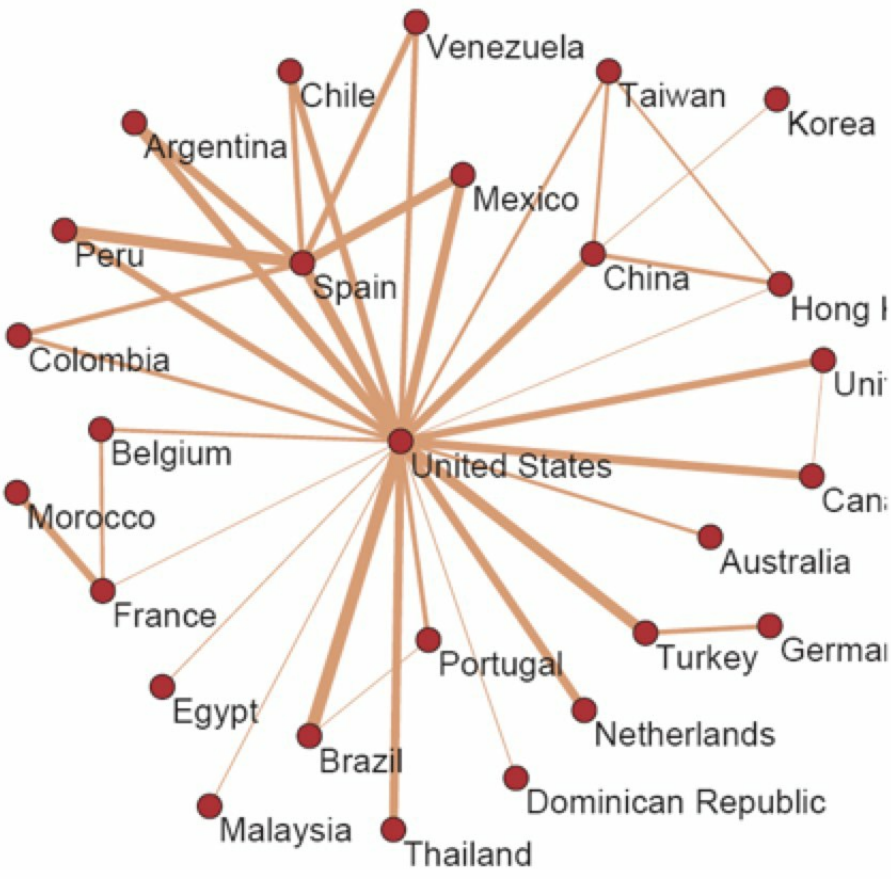


Figure 35: Who talks to whom: Number of conversations

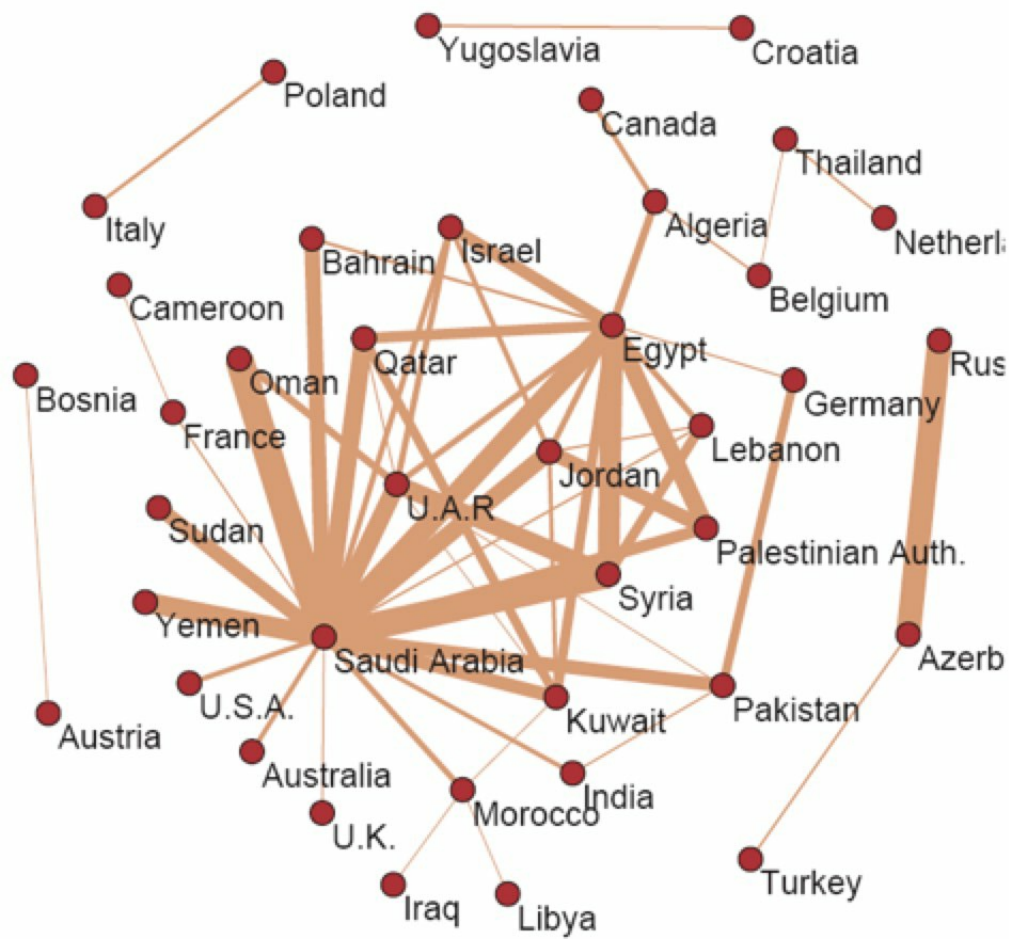


Figure 36: Who talks to whom: Conversation duration

Another interesting question is the geography and communication:

The number users are logged in from particular location on the earth (Figure 37)

How Europe is talking (Figure 38)

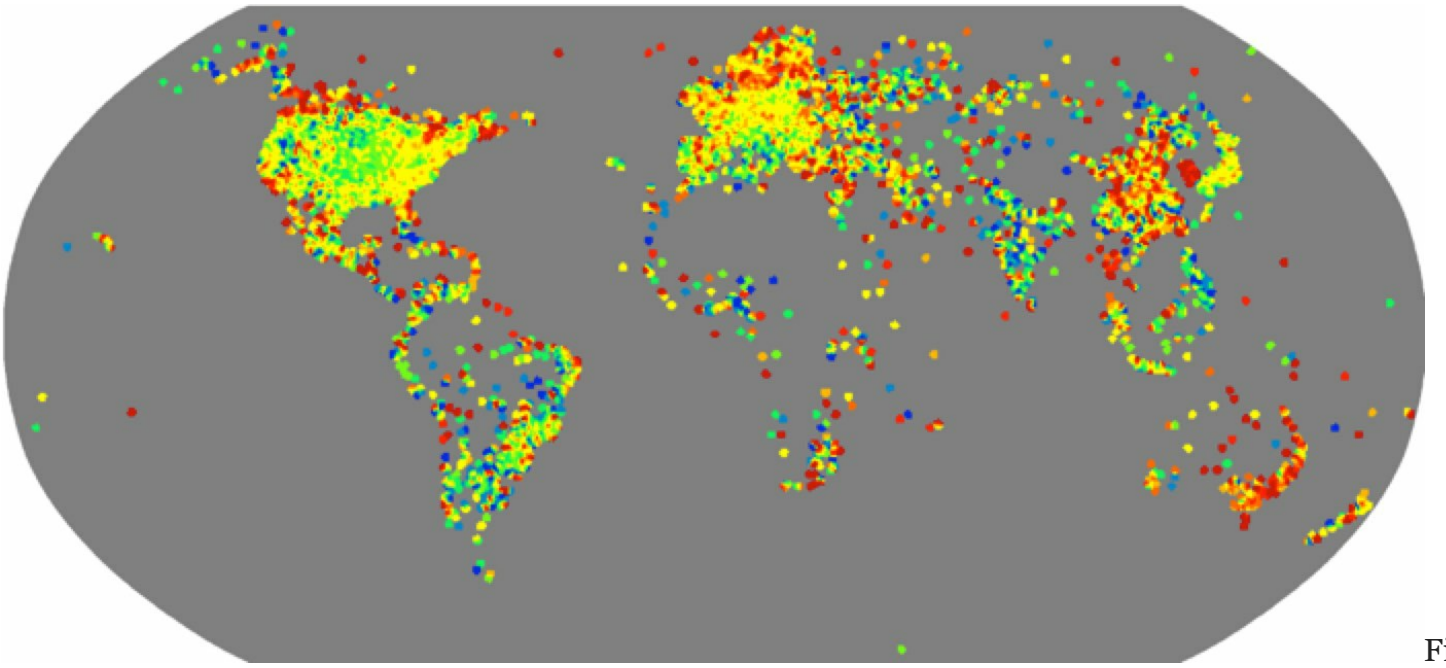


Figure 37: Users logging from particular location

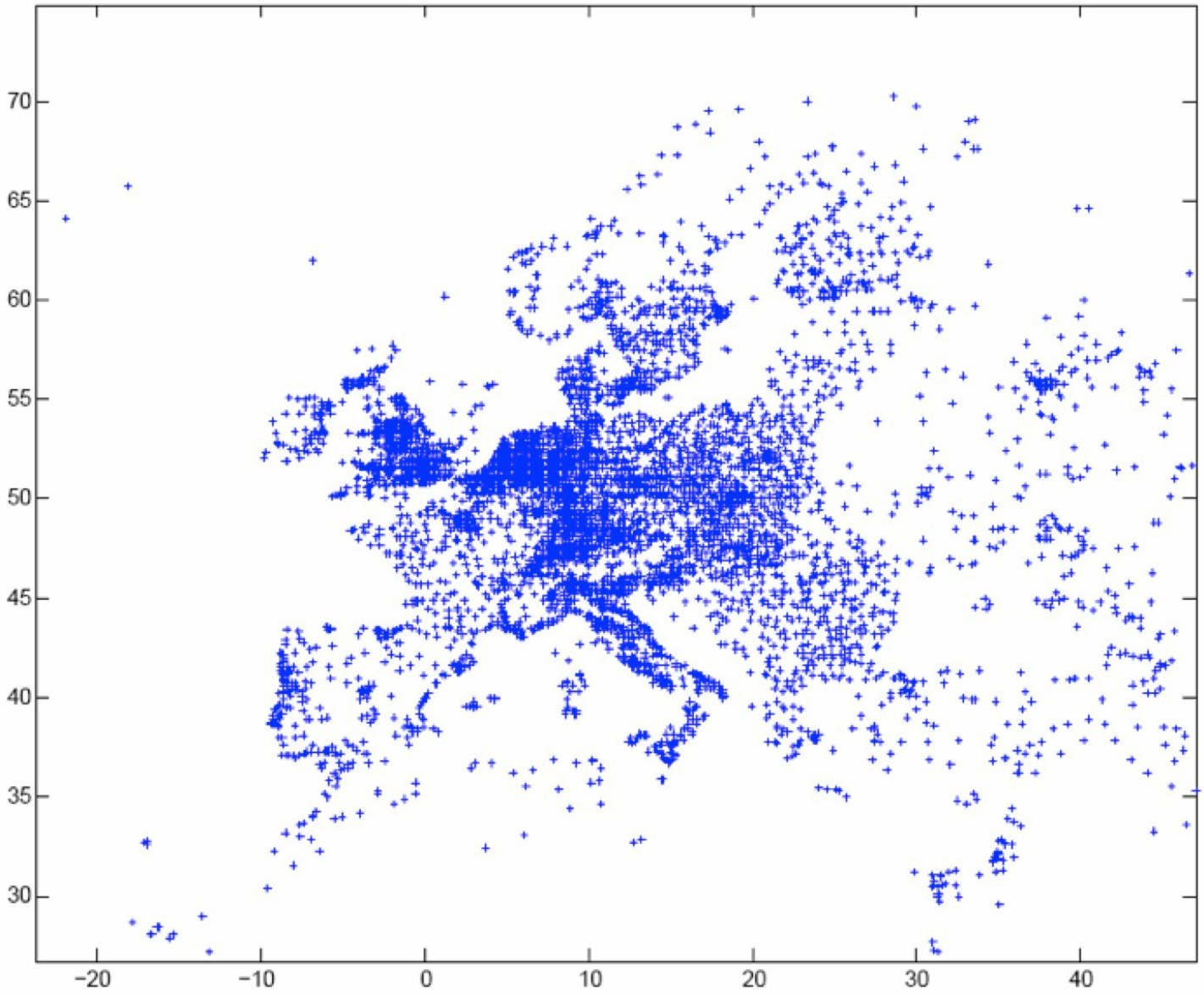


Figure 38: How Europe is talking

The Small-world problem is illustrated in Figure 39 & Table 1:

6 degrees of separation [Milgram ’60s]

Average distance between two random users is 6.6

90% of nodes can be reached in < 8 hops

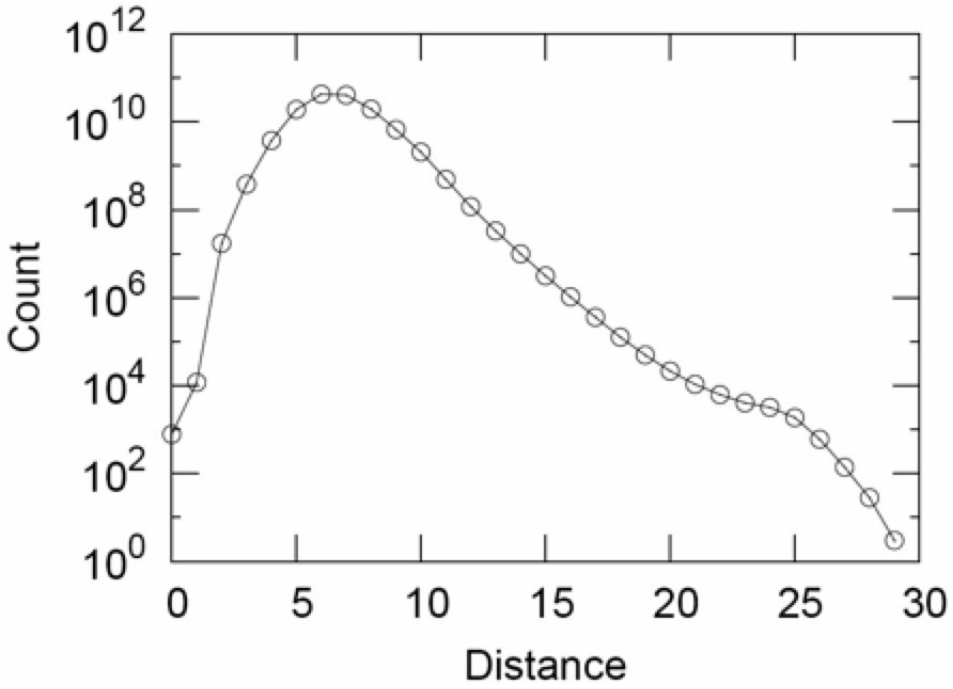


Figure 39: Network: Small-world

Table 1: Hopes/Nodes

Hops	Nodes	Hops	Nodes
1	10	14	13740
2	78	15	4476
3	396	16	1542
4	8648	17	536
5	3299252	18	167
6	283958491	9	71
7	79059497	20	29
8	52995778	21	16
9	10321008	22	10
10	1955007	23	3
11	518410	24	2
12	149945	25	3
13	44616		

Global Media Monitoring

The aim of the Global Media Monitoring project is to collect and analyze global main-stream and social media:

...documents are crawled from 100 thousands of sources

...each crawled document gets cleaned, linguistically and semantically enriched

...we connect documents across languages (cross-lingual technology)

...we identify and connect events

Collecting global media is performed in near-real-time via NewsFeed. The NewsFeed.ijs.si system collects:

- 40.000 main-stream news sources
- 250.000 blog sources
- Twitter stream

...resulting in ~500.000 documents + #N of twits per day. Each document gets cleaned, linguistically and semantically annotated.

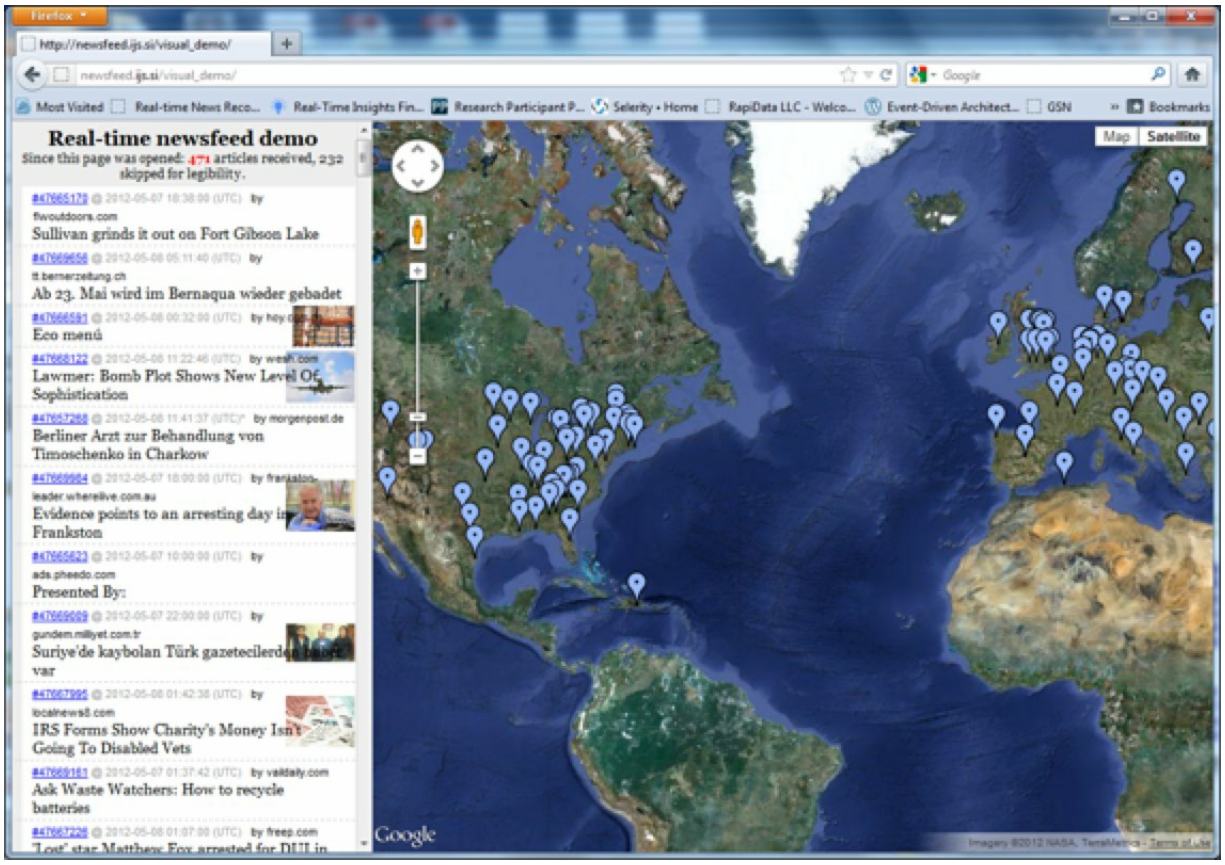


Figure 40: Newsfeed demo

Semantic text enrichment

Semantic text enrichment can be performed by the Enrycher system. Enrycher is available as a web-service generating Semantic Graph, LOD links, Entities, Keywords, Categories, Text Summarization.

DiversiNews

DiversiNews is a system for exploring news diversity. Reporting has bias – same information is being reported in different ways. DiversiNews system allows exploring news diversity along:

- Topicality
- Geography
- Sentiment

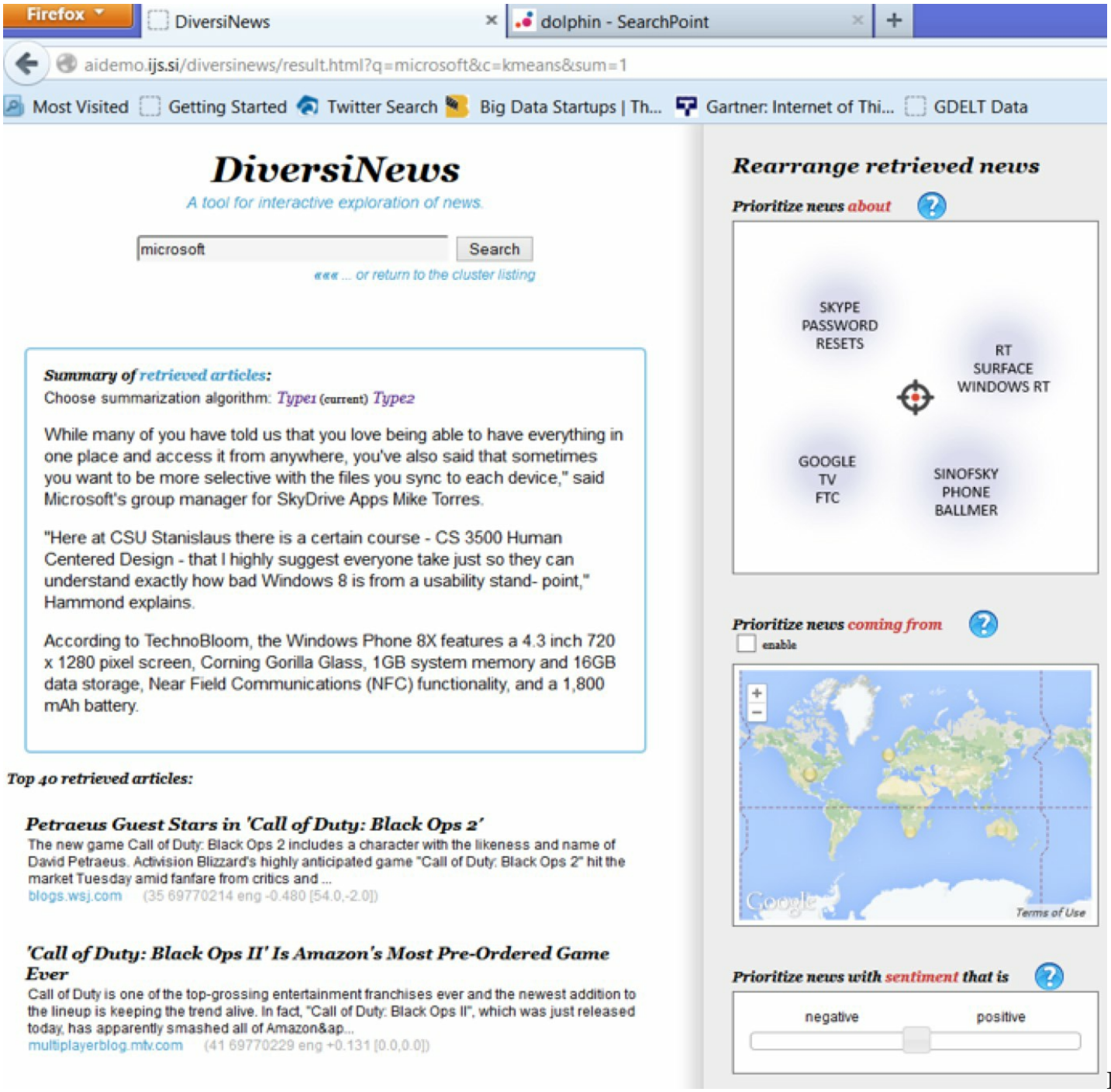


Figure 41: DiversiNews demo

Event Registry

EventRegistry is a system for event identification and tracking. Having stream of news & social media, the task is to structure documents into events. EventRegistry allows for:

- Identification of events from documents
- Connecting documents across many languages

- Tracking events and constructing story-lines
- Describing events in a (semi)structured way
- UI for exploration through Search & Visualization
- Export into RDF (Storyline ontology)

Data Analytics With QMiner

This section provides some practical insights on data analytics using QMiner. QMiner is a data analytics platform for processing large-scale real-time streams containing structured and unstructured data.

QMiner implements a comprehensive set of techniques for supervised, unsupervised and active learning on streams of data. It enables easy extraction of rich feature vectors from data streams using the data importing, normalization, re-sampling, merging and enrichment functionality.

QMiner provides support for unstructured data, such as text and social networks across the entire processing pipeline, from feature engineering and indexing to aggregation and machine learning. It also provides out-of-the-box support for indexing, querying and aggregating structured, unstructured and geospatial data using a simple query language.

QMiner applications are implemented in JavaScript, making it easy to get started. Using the Javascript API it is easy to compose complete data processing pipelines and integrate with other systems via RESTful web services.

QMiner is implemented in C++ and can be included as a library into custom C++ projects, thus providing them with stream processing and data analytics capabilities.

Examples

Twitter

This example shows how to perform text mining (feature spaces, active learning, classification) and record set filtering. It also applies a record set aggregate that builds communication graphs based on collections of Twitter messages (Twitter specific qminer aggregate). You can download the code from GitHub.

Twitter.def:

```
[
   {
     "name": "Tweets",
     "fields": [
       { "name": "ID", "type": "string", "primary": true },
       { "name": "Date", "type": "datetime" } ,
       { "name": "Text", "type": "string", "store": "cache" },
       { "name": "Geo", "type": "float_pair", "null": true },
       { "name": "Sentiment", "type": "int", "null": true }
     ],
     "joins": [
       { "name": "author", "type": "field", "store": "Users", "inverse" : "author" },
       { "name": "hasUser", "type": "index", "store": "Users", "inverse" : "hasUser" },
       { "name": "hasHashtag", "type": "index", "store": "Hashtags", "inverse" : "hasHashtag" },
       { "name": "hasPages", "type": "index", "store": "Pages", "inverse" : "hasPages" },
```

```
    { "name": "hasMedia", "type": "index", "store": "Pages", "inverse" : "hasMedia" }
  ],
  "keys": [
    { "field": "Text", "type": "text" },
    { "field": "Geo", "type": "location" }
  ]
},
{
  "name": "Users",
  "fields": [
    { "name": "Username", "type": "string", "primary": true },
    { "name": "Name", "type": "string", "null": true },
       { "name": "Location", "type": "string", "null": true },
    { "name": "Followers", "type": "int", "null": true },
    { "name": "Friends", "type": "int", "null": true }
  ],
  "joins": [
    { "name": "author", "type": "index", "store": "Tweets", "inverse" : "author" },
    { "name": "hasUser", "type": "index", "store": "Tweets", "inverse" : "hasUser" }
  ],
  "keys": [
    { "field": "Location", "type": "value" }
  ]
},
{
  "name": "Hashtags",
  "fields": [
    { "name": "Name", "type": "string", "primary": true }
  ],
  "joins": [
    { "name": "hasHashtag", "type": "index", "store": "Tweets", "inverse" : "hasHashtag" }
  ]
},
{
  "name": "Pages",
  "fields": [
```

```
      { "name": "URL", "type": "string", "primary": true }
    ],
    "joins": [
      { "name": "hasPages", "type": "index", "store": "Tweets", "inverse" : "hasPages" },
      { "name": "hasMedia", "type": "index", "store": "Tweets", "inverse" : "hasMedia" }
    ]
  }
]
```

Twitter.js

```
// This example demonstrates text mining (feature vectors, active learning and classification)
// as well as record set filtering (based on time and classification results). It also builds
// communication graphs based on sets of twitter messages (twitter specific)

// Import libraries
var qm = require('qminer');
var analytics = qm.analytics;
var fs = qm.fs;

var base = new qm.Base({ mode: "createClean", schemaPath: "twitter.def" });

// Load tweets from a file (toy example)
// Set the filename
var tweetsFile = "./sandbox/twitter/toytweets.txt";
// Get the store
var Tweets = base.store("Tweets");
// Load tweets (each line is a json)
qm.load.jsonFile(Tweets, tweetsFile);
// Print number of records

console.log("number of records: " + Tweets.length);
// Select all tweets
var recSet = Tweets.allRecords;

// Active learning settings: start svm when 2 positive and 2 negative examples are provided
var nPos = 2; var nNeg = 2; //active learning query mode
// Initial query for "relevant" documents
```

```
var relevantQuery = "nice bad";

// Initial query for positive sentiment

var sentimentQuery = "nice";



// Compute the feature space (if buildFtrSpace is false loads it from disk)

var buildFtrSpace = true;

// Learn a model that filters "relevant" documents (if learnSvmFilter is false, then the model is loaded from disk)

var learnSvmFilter = true;

// Learn a sentiment model (if learnSvmSentiment is false, then the model is loaded from disk)

var learnSvmSentiment = true;



// Load everything?

var justLoad = false;

if (justLoad) {

    buildFtrSpace = false;

    learnSvmFilter = false;

    learnSvmSentiment = false;

}

// The feature space provides the mapping from documents (tweets) to sparse vectors (provided by linear algebra module)

// Create or load feature space

var ftrSpace = new qm.FeatureSpace(base, [

       { type: "text", source: "Tweets", field: "Text" },

]);

if (buildFtrSpace) {

    // Builds a new feature space

    ftrSpace.updateRecords(recSet);

    // Saves the feature space

    var fout = fs.openWrite("./sandbox/twitter/fs.dat");

    ftrSpace.save(fout);

    fout.close();

} else {

    // Load the feature space

    var fin = fs.openRead("./sandbox/twitter/fs.dat");

    ftrSpace = new qm.FeatureSpace(base, fin);

}
```

```
// Learn a model of relevant tweets
if (learnSvmFilter) {
    // Constructs the active learner
    var al = new analytics.ActiveLearner(relevantQuery, recSet, undefined, ftrSpace,
        {nPos: nPos, nNeg: nNeg, textField: "Text"}
    );
    // Starts the active learner (use the keyword stop to quit)
    al.startLoop();
    // Save the model
    var fout = fs.openWrite('./sandbox/twitter/svmFilter.bin');
    al.saveSvmModel(fout);
    fout.close();
}

// Load the model from disk
var fin = fs.openRead("./sandbox/twitter/svmFilter.bin");
var svmFilter = new analytics.SVC(fin);
// Filter relevant records: records are dropped if svmFilter predicts a v negative value (anonymous function)
recSet.filter(function (rec) { return svmFilter.predict(ftrSpace.extractSparseVector(rec)) > 0; });

// Learn a sentiment model
if (learnSvmSentiment) {
    // Constructs the active learner
    var al = new analytics.ActiveLearner(sentimentQuery, recSet, undefined, ftrSpace,
        {nPos: nPos, nNeg: nNeg, textField: "Text"}
    );
    // Starts the active learner
    al.startLoop();
    // Saves the sentiment model
    var fout = fs.openWrite('./sandbox/twitter/svmSentiment.bin');
    al.saveSvmModel(fout);
    fout.close();
}
// Loads the sentiment model
var fin = fs.openRead('./sandbox/twitter/svmSentiment.bin');
```

```
var svmSentiment = new analytics.SVC(fin);


// Classify the sentiment of the "relevant" tweets

for (var recN = 0; recN < recSet.length; recN++) {

    recSet[recN].Sentiment = svmSentiment.predict(ftrSpace.extractSparseVector(recSet[recN])) > 0 ? 1 : -1;

}


// Filter the record set of by time

// Clone the rec set two times

var recSet1 = recSet.clone();

var recSet2 = recSet.clone();


// Set the cutoff date

var tm = new Date("2011-08-01T00:05:06");

// Get a record set with tweets older than tm

recSet1.filter(function (rec) { return rec.Date.getTime() < tm.getTime() })

// Get a record set with tweets newer than tm

recSet2.filter(function (rec) { return rec.Date.getTime() > tm.getTime() })


// Print the record set length

console.log("recSet1.length: " + recSet1.length + ", recSet2.length: " + recSet2.length);
```

Movies

This example shows how to extract features from the movie dataset and use them to build classification and regression models to predict movie genre and rating. The code is available on GitHub.

Linalg

This example shows how to use linear algebra functionality. It covers building vectors and matrices, solving linear systems and performing matrix singular value decomposition. The code is available on GitHub.

Timeseries

This example shows how resampling, enrichment and prediction work on time series data. The code is available on GitHub.

Applications of Qminer

Event Registry: The Event Registry is a system that can automatically identify events happening across the world that have been reported in news articles. For each event it can extract, from the available articles, the main information about the event (who, when, where, ...).

Web Audience Segmentation: The Web analytics product provides insights into the audience of a website, how

segments differ between each other. I also provides predictive analytics for targeting specific user segments.

Real-Time Recommendation: QMiner powers a state-of-the-art recommendation system, which provides accurate and up-to-date recommendations for highly dynamic domains. Examples of such domains include news portals, IPTV and social media.

StreamSense: StreamSense is a sensor stream processing system based on tightly integrated and scalable custom software modules. StreamSense provides interfaces and means of information collection from a set of Smart Objects and generic APIs for data feeds.

Elycite: Elycite is semi-automatic, collaborative, data exploration and organization tool. It integrates machine learning and text mining algorithms into a simple user interface and a Client/Server architecture.

Big Data Architecture

Big data components, as used by internet giants such as Google, Facebook, or Amazon, are provided, and often enriched, by open source communities. The technological basis needed to exploit big data opportunities is available both as free software and as part of the portfolios of large systems providers. However, there are comparatively strong barriers in the development of big data applications: big data components usually offer a smaller range of functionality than conventional operating systems, classical relational database systems, or business intelligence systems.

This chapter provides insights into the essential technological offerings and the resulting value of big data components. In a concrete example, we show how big data components can be combined based on the concept of the lambda architecture.

This chapter is an extended version of the following article:

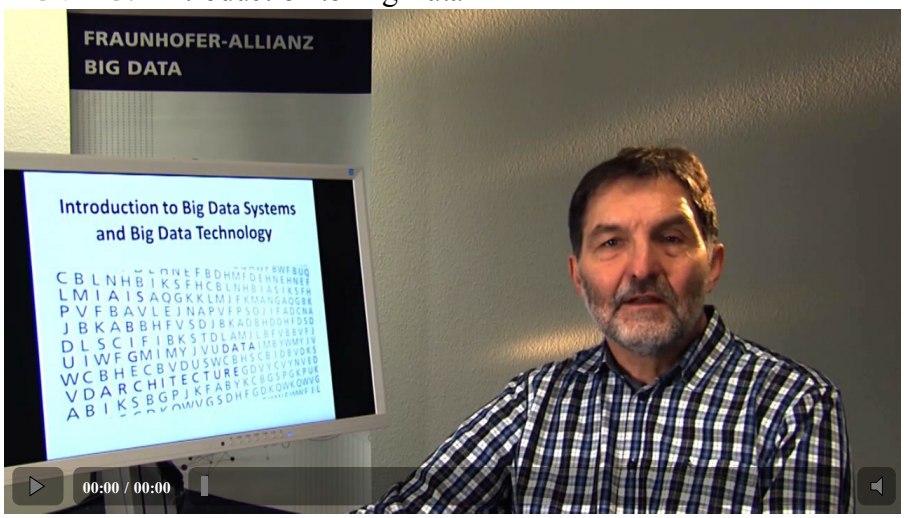
OBJEKTspektrum Online-Themenspecial: IT-Trends: (2014) BigData/Hadoop und Internet der Dinge M. Mock, K.H. Sylla, D. Hecker: “Skalierbarkeit und Architektur von Big-Data-Anwendungen”. It is also based on the contents of this training course on Big Data Architecture.

Introduction

Let us start with two examples of players who have adopted big data architectures early on: last.fm and Amazon.

Last.fm is both, an internet radio and a music community website. It provides music streams and downloads but also many user specific services such as music and event recommendation, personalised chart lists, and presentation according to personal listening habits, taste in music, similar music, and stylistic similarities. In particular, it provides these services for many concurrent users. In 2012, last.fm had more than 25 million users per month.

MOVIE 3.1 Introduction to Big Data



Amazon have to handle many separate applications working on big data sets such as

Seller lists,

Shopping carts,

Customer preferences,

Session management,

Sales rank,

Product catalogue.

Websites are rendered by using data, aggregations, and analysis results of the different applications. This leads to the following strong requirements for their infrastructure:

High availability – for read-write operations,

High performance – short response times,

Service level agreement between applications – short, guaranteed response times even under high load.

Two models for big data systems

Historically, last.fm and Amazon provide examples of two different aspects of big data systems. First (see Model 1 below), there is the aspect of redundant distributed storage of large amounts of data. Second (see Model 2 below), there is the question of databases whose size exceeds the capabilities of traditional relation database systems.

Model 1: Hadoop

Since 2006 last.fm uses Apache Hadoop as an infrastructure to stem the heavy computational load created by its broad user base. Here are a few reasons why Hadoop is a good solution in such a case:

Hadoop provides a distributed filesystem with redundant backups,

It is scalable with "commodity hardware",

It is available for free,

It is an open source system which is extensible with your own features, and

It is flexible and easy to learn.

In 2010 last.fm was managing about 100 TB of data using 50 nodes with a total of 300 computer cores.

Facebook, as another example, is running one of the largest Hadoop clusters in the world to provide their users with

Daily/hourly reports

- For product analysis and product planning
- Performance of marketing campaigns
- Reports on usage data

ad hoc evaluation of historical data

Long-term archiving of log data

Already some years ago, this cluster provided 9 TB of main memory.

You can find more famous Hadoop users on Hadoop's powered by website.

Model 2: Amazon Dynamo

Amazon have to deal with extremely large amounts of data. Even though these are structured in a way to suggest the use of traditional relational database systems, the sheer amount of data makes their use prohibitive. For this and other reasons, they have developed their own big data database system, Dynamo, along the following design principles:

Incremental scalability: Dynamo should be able to scale out one storage host ("node") at a time

Symmetry: Every node in Dynamo should have the same set of responsibilities as its peers

Decentralisation: Favour decentralised peer-to-peer techniques over centralised control

Heterogeneity: The system needs to be able to exploit heterogeneity in the infrastructure it runs on. This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.

This, together with Google's BigTable marked the beginning of a new generation of database systems specifically designed to handle big data problems.

Relational databases and big data

Relational databases are a traditional and well-developed tool for the backend of internet shops and other online applications. They are used in many solutions for shopping carts, product catalogues, and related data products. So why would these not fit the needs of a giant like Amazon?

Here are the main reasons why traditional relational database management systems (RDBMS) do not fit well into the big data scenario:

RDBMS are not good in data distribution and partitioning

RDBMS produce an overhead that can be avoided when

- Many read or write operations occur as key-value-pairs
- Only large data blobs have to be stored

Problems like these have led to a new generation of database systems: the NoSQL family. NoSQL stands for not only SQL, a collection of paradigms which lead to the following approaches:

Flexible schemes

Queries without JOINs

Horizontally scalable

By now, various different types of databases with different data models are available and can be roughly categorised into the following groups:

- Key Value Stores
- Column Oriented
- Document Oriented
- Graph Oriented

This leads to more possibilities of developing new solutions or solving problems than with classical RDBMS. However, it is also a step away from multiple purpose solutions towards a plethora of different solutions for different data types and processing needs.

Technically, the adoption of NoSQL systems means to replace the well-known ACID principle of atomicity, consistency, isolation, and durability by a new set of influencing factors for new data management technologies known as SPRAIN:

Scalability – efficiency of hardware

Performance – limits of SQL databases, "impedance mismatch"

Relaxed consistency – distribution, CAP Theorem

Agility – variety of data and persistence, diverse languages and systems

Intricacy – complex requirements (big data, total data)

Necessity –need for technology, use of open source products

These factors lead to three current trends in regarding database systems for big data:

Trend 1: NoSQL databases are designed to meet the scalability requirements of distributed architectures and/or schemaless data management requirements.

Trend 2: NewSQL databases are designed to meet the requirements of distributed architectures or to improve performance such that horizontal scalability is no longer needed.

Trend 3: Data grid/cache products are designed to store data in memory to increase application and database performance.

Complex event processing

A third aspect of big data comes into the play in scenarios like complex event processing where close to real-time delivery of results based on parallel data streams from many sources or sensors are needed. In comparison to other systems, streaming and real-time systems can only work with constant memory. This means that the amount of data to be handled is much larger than what can be buffered on a single computer.

An illustrative example of complex event processing is given by fraud detection in credit card transactions. In this case, credit card transactions come from many sources including automatic teller machines around the world. It is only in the combination of all these data streams that the patterns of credit card fraud become visible and finding them is meaningless if this does not happen close to real-time.

There are several systems, both open source and commercial, available which are suitable for complex event processing. Examples comprise Apache Storm (see the chapter about the speed layer), S4, StreamSQL, SQLstream, and IBM InfoSphere Streams.

Big data systems

Big data systems are inherently distributed and big data algorithms and applications must be parallelisable. Similar to the case of operating systems, a separation into the actual application and a big data system in the background should be offered which hides data distribution and its handling from the application programmer and provides a restricted set of operations on data which are automatically parallelisable and hence lead to applications making transparent use of data and algorithm parallelisation.

So what is a big data system or rather when is a system capable of handling big data? There are two central requirements which define such a system:

Horizontal scalability

Availability and fault tolerance

Current big data systems are based on computing nodes each of which has its own main memory and secondary storage. These nodes are connected by network only (shared nothing architecture).

A Big Data System should provide horizontal scalability in the sense that adding more computing nodes provides more storage and processing capacity to the application without the need to modify its implementation. This is ensured by the system.

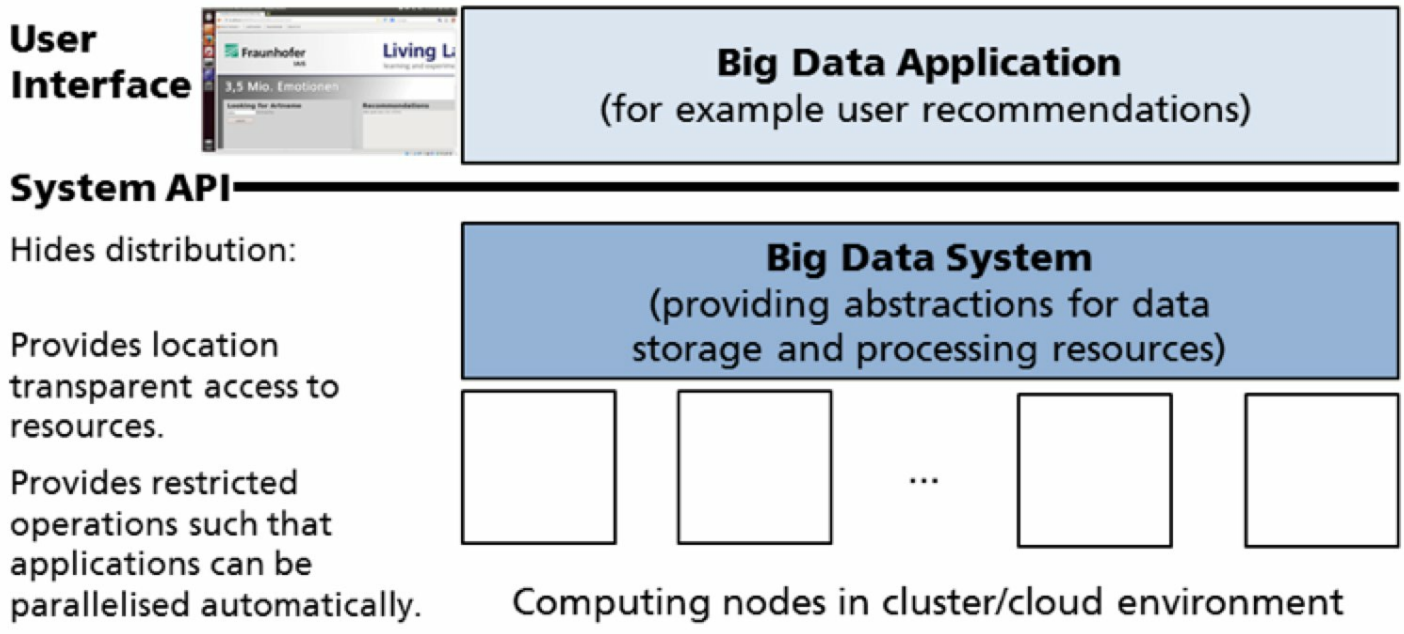


Figure 1: Separation of Big Data Application and Big Data System by an API.

Horizontal scalability is contrasted by vertical scalability: this means increasing computing resources and/or storage resources in a single node. This approach to scalability reaches natural limits and is therefore not suited for big data approaches.

When using large numbers of nodes, any computing node may fail at any time (stop working – fail stop model) and needs to be repaired after some time. The availability of a computing node, in its classical definition is based on two measures:

MMTF: Mean Time to Failure, MMTR: Mean Time to Repair

Based on these, the availability of a system is given by:

Availability = MTTF / (MTTF + MTTR)

The availability is often interpreted as a probability pNode that the node is working at a given point of time. Using this definition of availability for big data system leads to the following

Assumption (*): The complete system is working only if all N computing nodes are working and the probability for computing nodes to fail is independent.

Then the probability that the whole system works is given by pSystem = (pNode) x N. This means, for example, that in a system with 1000 nodes in which each single node has a downtime of 1 day per year pSystem = (359/360) x 1000 = 0.06. Such a system would hardly ever work.

A system working only under assumption (*) is therefore not horizontally scalable. A big data system must be fault-tolerant. It must continue working even when individual nodes fail because this is hardly an exceptional case.

The CAP Theorem

A fundamental limitation for distributed systems is given by the CAP theorem. It states that a distributed system can fulfil at most two of the following qualities at the same time:

Consistency (C): all replicates of data are equal and all nodes see the same state at the same time.

Availability (A): every request is handled by the system.

Partition tolerance (P): The system continues to work even at loss of messages, network nodes or a network partition.

This gives rise to a certain design space by selecting any pair of qualities and sacrificing the third. The following list gives some examples for systems based on the different choices:

Consistency and availability: This is what classical relational database management systems provide.

Consistency and partition tolerance is a typical choice for banking applications.

Availability and partition tolerance is well-known from the domain name system (DNS) and from cloud computing.

More than a decade after the advent of the CAP theorem, its perception has changed so that it is better adjusted to the context of cluster and distributed computing. On the one hand, partitions are rare and there is therefore little incentive to sacrifice consistency or availability. On the other hand, the qualities in the theorem are understood less as binary choices than as continuous ones which moreover can be locally reconsidered repeatedly during the operation of a system.

In real-world applications, it is often sufficient to reach eventual consistency. A system may be inconsistent between different partitions for some time. This situation is often acceptable if consistency is (quickly) regained

again and again: consistency is recovered whenever no writing is performed and communication between the partitions works.

Big data challenges

Now that we have an idea what a big data system is, let us have a look at those challenges which define big data. There are often summarised by the three Vs:

Volume: the massive amounts of data to handled today lead to strong challenges in data management, data analysis, and machine learning approaches.

Variety: the many different kinds of new data types and new information sources pose challenges in data integration.

Velocity: the massive data flow due to highly increased data production, asks for new approaches in data processing.

These aspects of big data are addressed by different technologies. We will see in this course how Hadoop and big data databases are used to handle volume, how different types of big data databases tailored for certain aspects of unstructured data handle variety, and how stream processing technologies like Storm are employed to handle velocity.

Time and again, the three Vs are seen as too narrow a picture for the big data world. To finish our overview, let us mention two additional candidates for central aspects of big data:

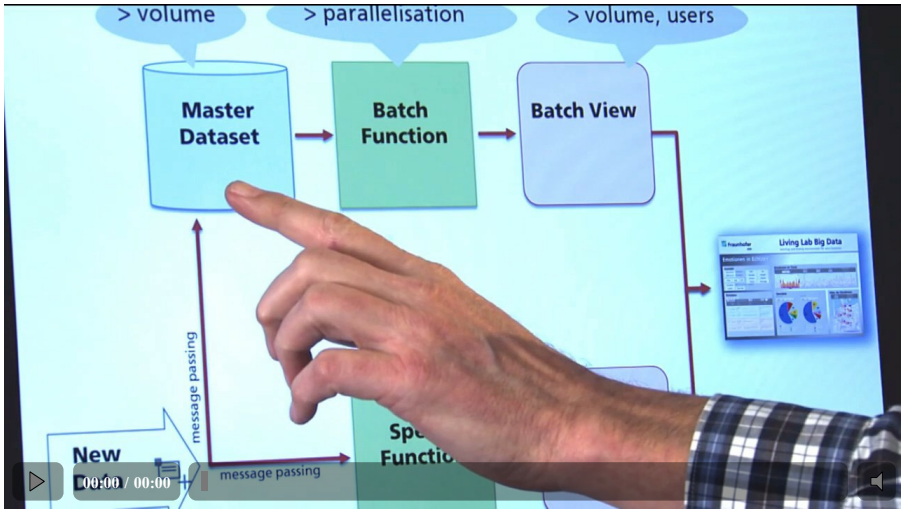
Veracity and value: The large volume of data often makes it hard to verify the value and correctness of data. Poor data quality is often unavoidable when working with sources generating huge amounts of data in an uncontrolled way. Therefore handling the veracity problem is of the essence for successful big data applications. “Value” denotes the aspect that the data holds information that is relevant and useful to the business context.

Analytics: To make sense of the huge amounts of data which are processed in big data systems, it is necessary to scale up traditional analytics tools from statics, machine learning, and pattern recognition. This is the topic of the field Big Data Analytics.

The Lambda Architecture

A constructive template for the conception and design of big data applications is the lambda architecture approach published by Nathan Marz with James Warren (see Nathan Marz, James Warren; Big Data - Principles and Best Practices of Scalable Real-Time Data Systems; Manning Publications; 2015 for further information). They suggest an architecture whose modularisation reflects typical requirements to big data applications and combines them in a systematic picture. This architecture approach helps to uncover and judge technical and non-functional requirements in new applications independent of the form and extent in which certain modules are realised as technical components of the application. The template makes it possible to identify the necessary services and allows justifying the selection of necessary components.

MOVIE 3.2 The Lambda Architecture



Applications are usually based on data streams such as user activities or sensor data from technical systems. These data are logged and lead to an increasing volume of data. In the lambda architecture, original data is tagged by a time-stamp and recorded without loss of information. In the data storage, the complete and growing set of original data remains available. In this way, all recorded information can be used in new, corrected, or extended functionality.

The lambda architecture is structured into two layers:

The batch layer: In this layer, all the collected original data is processed using a *batch function*. It is prepared for the presentation of computation results in a *batch view*. Batch processed are repeated either cyclically or on demand. This is very similar to ETL processes filling data warehouses with data prepared and optimized for online analyses (OLAP).

The speed layer: In this layer, incoming data is processed immediately and as quickly as possible and then prepared for presentation or visualisation in the application based on a *real-time view*. The speed layers bridges the much longer processing times of the batch layer and reduces the delay between the arrival of data and the time when results become available in the batch view.

Figure 2 shows these in components as an architecture diagram.

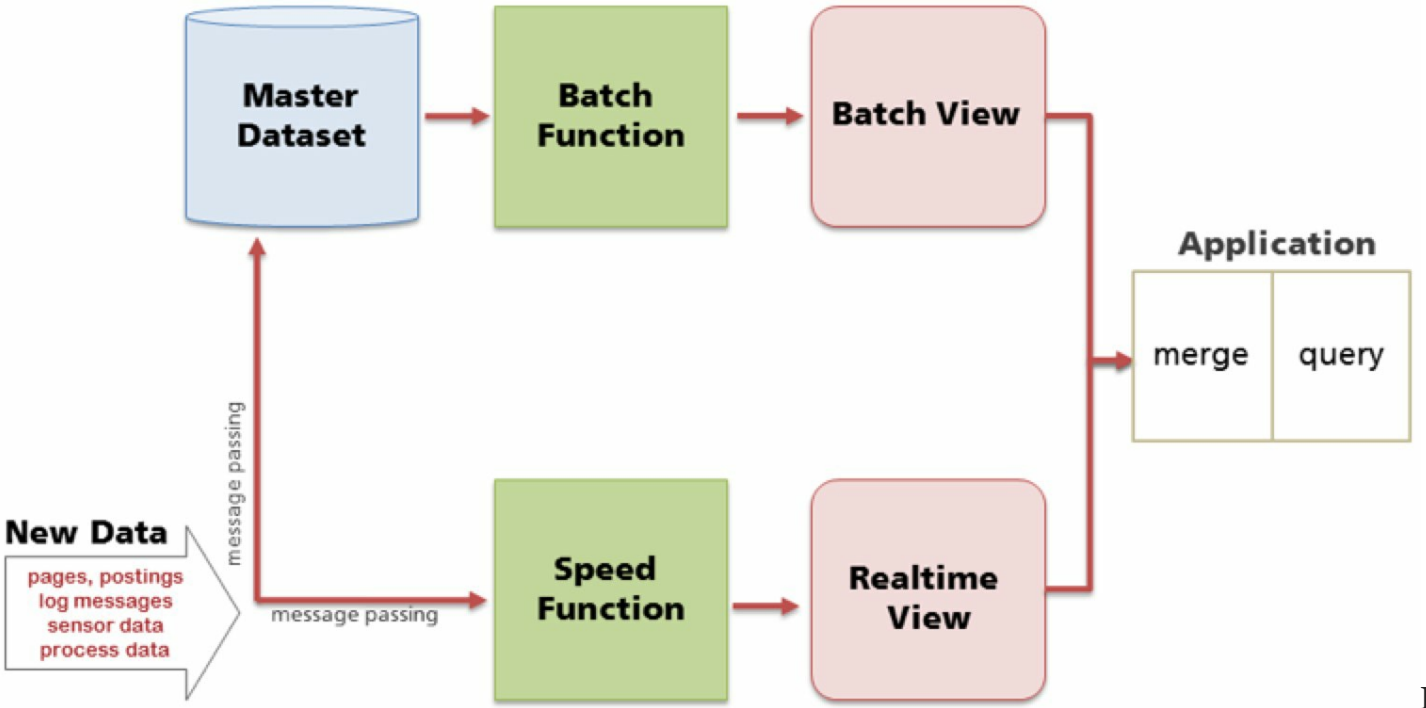


Figure 2: Components of a lambda architecture.

The name lambda architecture is derived from a functional point of view regarding data processing: all data processing is understood as the application of a function to all data.

The ability of the lambda architecture template to help design systems that are highly scalable and capable of short response times on huge amounts of data rests on the concepts of immutability of data, data denormalisation, and precomputed views. These, we will explain next.

Immutability of Data

An important concept regarding data handling in big data systems is the immutability of data. To avoid data loss and data corruption in such systems, data is handled in such a way that records can never be altered or deleted.

Immutable data is fundamentally simpler than mutable data. The concept allows only to create and to read records (CR) as opposed to additionally being able to update and delete records (CRUD) like in relational databases. Thus write operations only add new data units.

This approach makes data handling highly scalable because it is very easy to distribute and replicate data. The data system itself becomes a kind of a logging system that, on appearance of new data, adds a timestamp and a unique id to that data record which is then recorded in the data store.

As a drawback, even more data is generated and answering queries becomes more difficult. For example, to find the current location of a person, the location entry for that person with the latest timestamp has to be found.

The following example shows how to go from operations on mutable data to the concept immutable data. Let's assume that we are storing information about car brands and their owners in a database of mutable data. Figure 3 shows what happens, when a car brand changes ownership: the original data in database is replaced by the new information.

Brand	Owner
Smart	Daimler
Mini	Rover

Brand	Owner
Smart	Daimler
Mini	BMW

Figure 3: Capturing of changes with a mutable data model.

Here, it is no longer possible to find out that the brand Mini was once owned by Rover. This is different when using immutable data which is achieved by adding a timestamp to each data record.

Brand	Owner
Smart	Daimler
Mini	Rover

Brand	Owner	Time
Smart	Daimler	1193903402
Mini	Rover	1325404202
Mini	BMW	1351756202

Figure 4: Capturing of changes with an immutable data model.

Now it is possible to get both bits of information: the fact that Mini is now owned by BMW (latest timestamp) and the fact the it was formerly owned by Rover. It is also much easier to recover from errors because the old information is not deleted.

Data Denormalisation

A fundamental reason for the limited ability of traditional database systems to scale their read performance to huge amounts of data lies in the normalized way in which data is stored. In order to make it easier to ensure consistency, schemas are designed in such a way that information is not duplicated. In this way, the problem that a single piece of information has to be updated in several places does not appear. However, in order to bring together the necessary information to answer complex queries costly join operations have to be computed most of the time. For the sake of performance, big data systems accept a denormalisation of the data schema such that data is stored in representation equivalent to that after performing joins on normalised tables. This is fundamentally simpler than using joins, no knowledge about the schema is necessary, and it is much faster because all related information is already in place. Even though the same Information may then be held in several places, even more data is generated than ever, and consistency is not ensured automatically upon partial updates, this is often acceptable, especially when denormalised representations are used as precomputed views.

Precomputed Views

In order to be able to give fast and consistent answers to queries on huge amounts of data, precomputed views are prepared both in the batch layer and in the speed layer. In the batch layer, these are constructed by applying a batch function to all of the data. This leads to a transformation of the data into a more compact form suitable for answering a pre-defined set of queries. This idea is essentially the same as what is done in data warehousing.

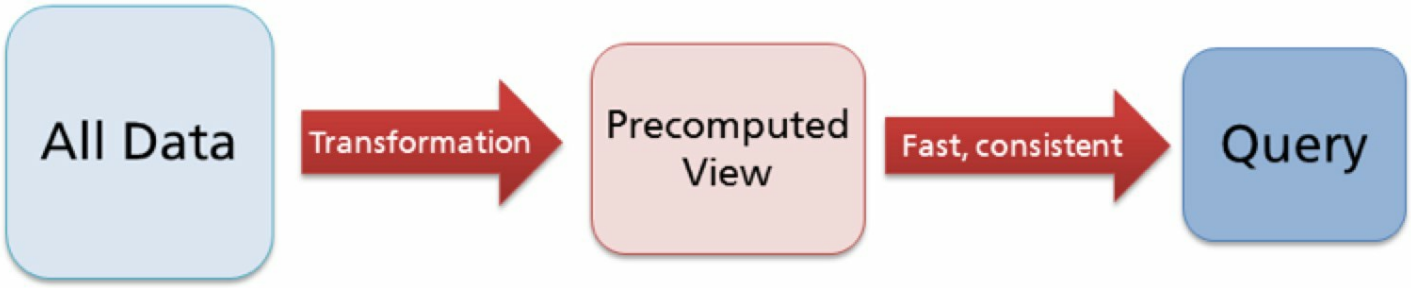


Figure 5: Fast and consistent answering of queries based on precomputed views.

The example in Figure 6 gives a first idea how this is applied. In order to count the number of postings in specific forum, the batch function aggregates information about individual postings over time spans of a suitable resolution (days in the example). The resulting view allows answering any queries about the number of postings with a resolution that is not larger than that used to prepare the view.

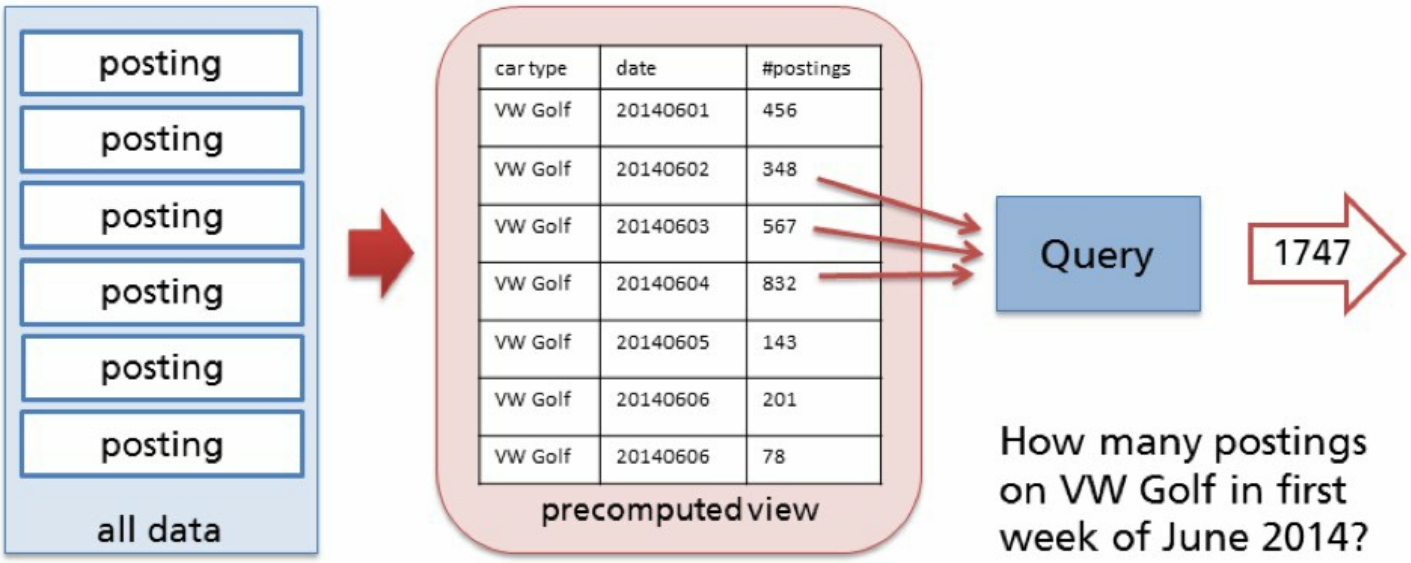


Figure 6: Example of a precomputed view: aggregation of postings over time ranges.

The batch view computed by the batch function is only updated after each new run of the batch process which may take a longer time. The gap to including the latest information is bridged by a second precomputed view, the speed view, which is prepared in the speed layer. Queries are then resolved by combining both precomputed views.

Relationship to Commercial Big Data Architectures

There are a number of commercial big data architectures available. In parts, these are comparable to the lambda architecture template. The template is an invaluable tool to help assess and combine different modules into a big data architecture.

A Running Example

In the following sections, we will discuss in turn processing in the batch layer and in the speed layer. In order to fill this discussion with life, we will employ a running example: the analysis of posts to the automobile enthusiast's forum MOTOR-TALK.

Figure 7 shows the user interface of our example system. It is based on forum posts about cars and their parts. The application analyses each post to prepare information about which part of which car the post is about and

what the predominant emotion in this context is, i.e., whether the author of the post seems to happy, angered, or neutral about the part of the car he is commenting on. The final application allows to analyse these data in different contexts like brand, location, and time span.

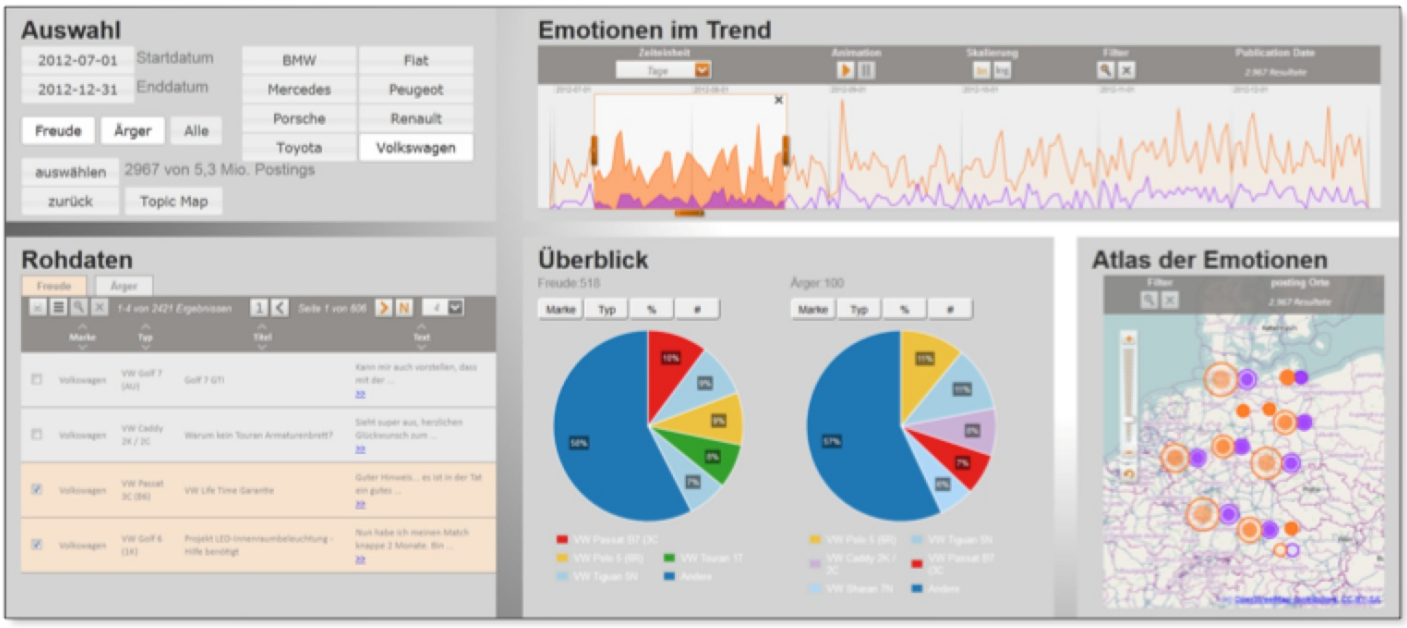


Figure 7: User Interface of the example system.

Figure 8 shows the choices of modules that we have made to fill in the functionality allotted by the lambda architecture template. Each of these components will be described in the subsequent sections.

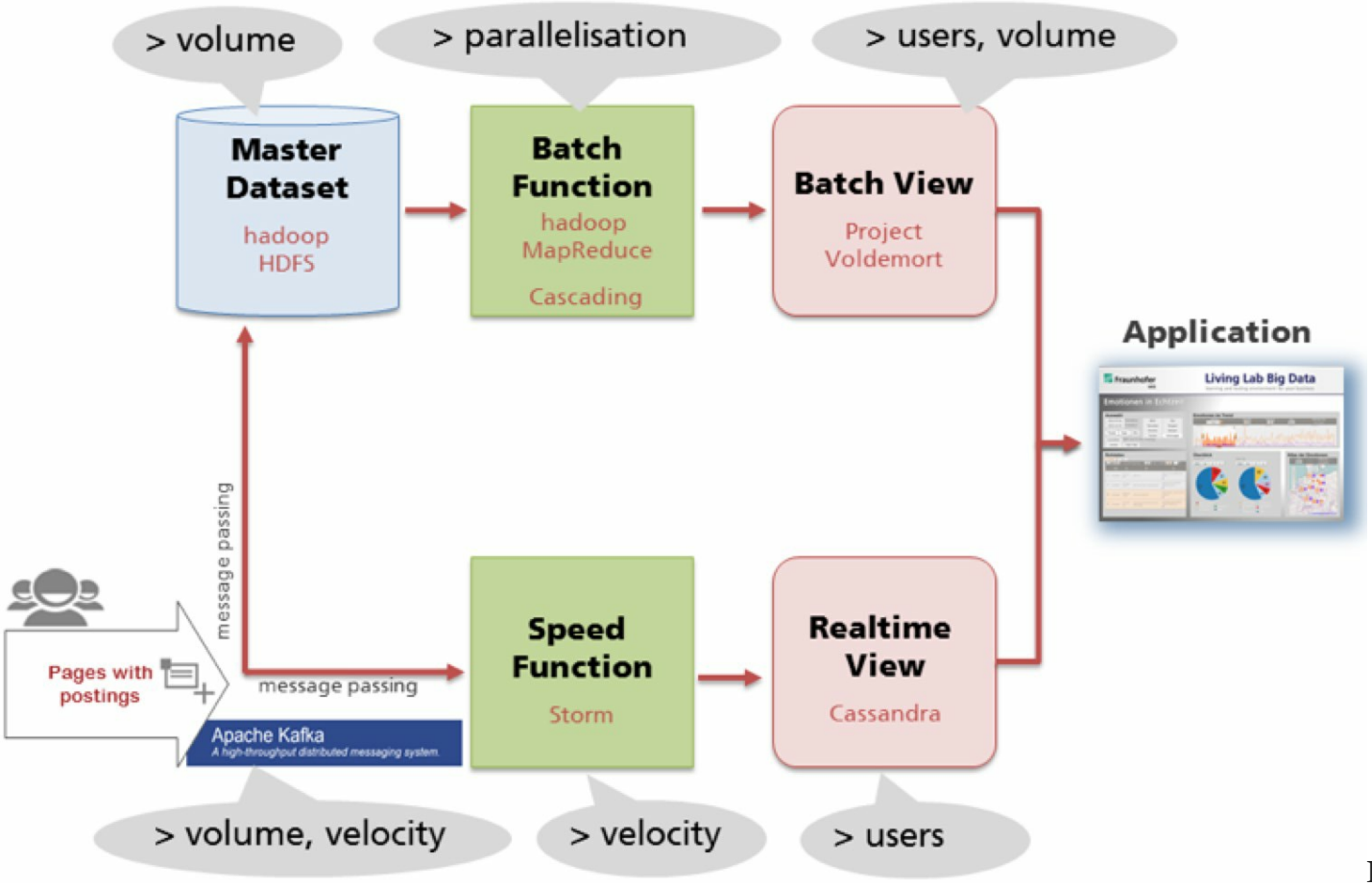


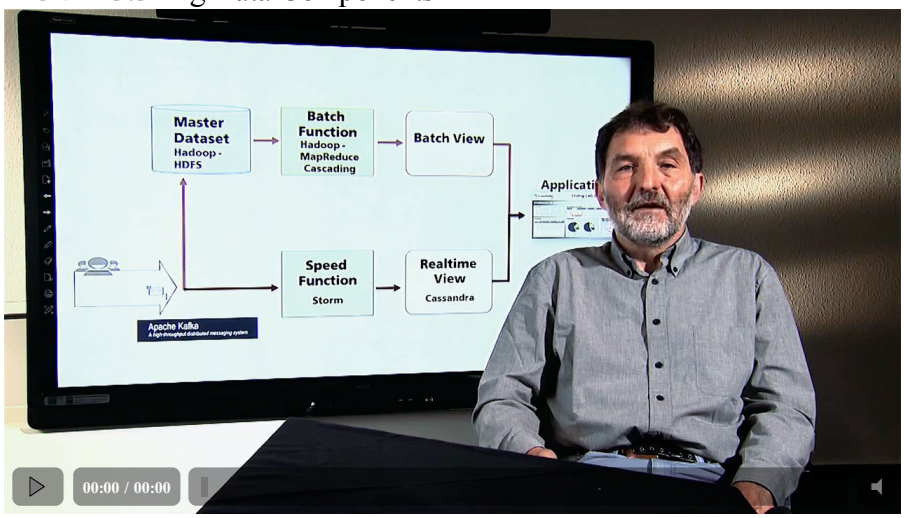
Figure 8:

Lambda architecture completed with module choices for the running example.

Batch Processing

We start our more detailed discussion of the elements of lambda architectures with the batch layer. Its job is to handle the distributed computing of large volumes of data and therefore needs to ensure horizontal scalability. The central aspect of parallel processing in this context is data parallel processing in which the processing of large amounts of data is broken down by distributing partitions of the data to different computers. A computing concept that makes this kind of parallelisation widely automatic is the MapReduce approach. This is available in open source from the Apache Hadoop project and together with Hadoop's distributed file system forms the core of many big data systems.

MOVIE 3.3 Big Data Components



MapReduce

The MapReduce concept offers a limited set of operations, namely map and reduce, which allow distributing data and data processing without extra effort of the programmer once a computation has been described as a sequence of map and reduce operations.

MapReduce is inspired by functional programming paradigms and provides a common pattern for processing data. Formally, it is defined as follows:

A *map procedure* projects (key, value) pairs onto an intermediate result which also is a list of (key, value) pairs:

$$Map\colon K\times V \to (L\times W)^*$$
$$(k, v) \mapsto [(\ell_1, x_1), \ldots, (\ell_{r_k}, x_{r_k})]$$

A *reduce procedure* projects a list of key-value pairs with the same key to a list of final result values:

$$Reduce\colon L\times W^* \to X^*$$
$$\left(\ell, [y_1, \ldots, y_{s_\ell}]\right) \mapsto [w_1, \ldots, w_{m_\ell}]$$

Accordingly processing is divided into two phases in the MapReduce framework:

Map Phase

1. Iterate over all input data items (pairs),
2. Construct a key-value pair for each item,
3. Group the intermediate resulting key-value pairs by key index.

Reduce Phase

1. Iterate over the groups (so-called partitions),
2. Reduce each of the groups to a final list of values

Implementations of MapReduce may extend this schema by split, combine, shuffle and partition functions.

Let us try to elucidate this concept by an example, the now classical word count example. Assume that the data is given as one huge text file and the task is to count which words occur and how often. A sequential approach to solve this task might look like Figure 9.

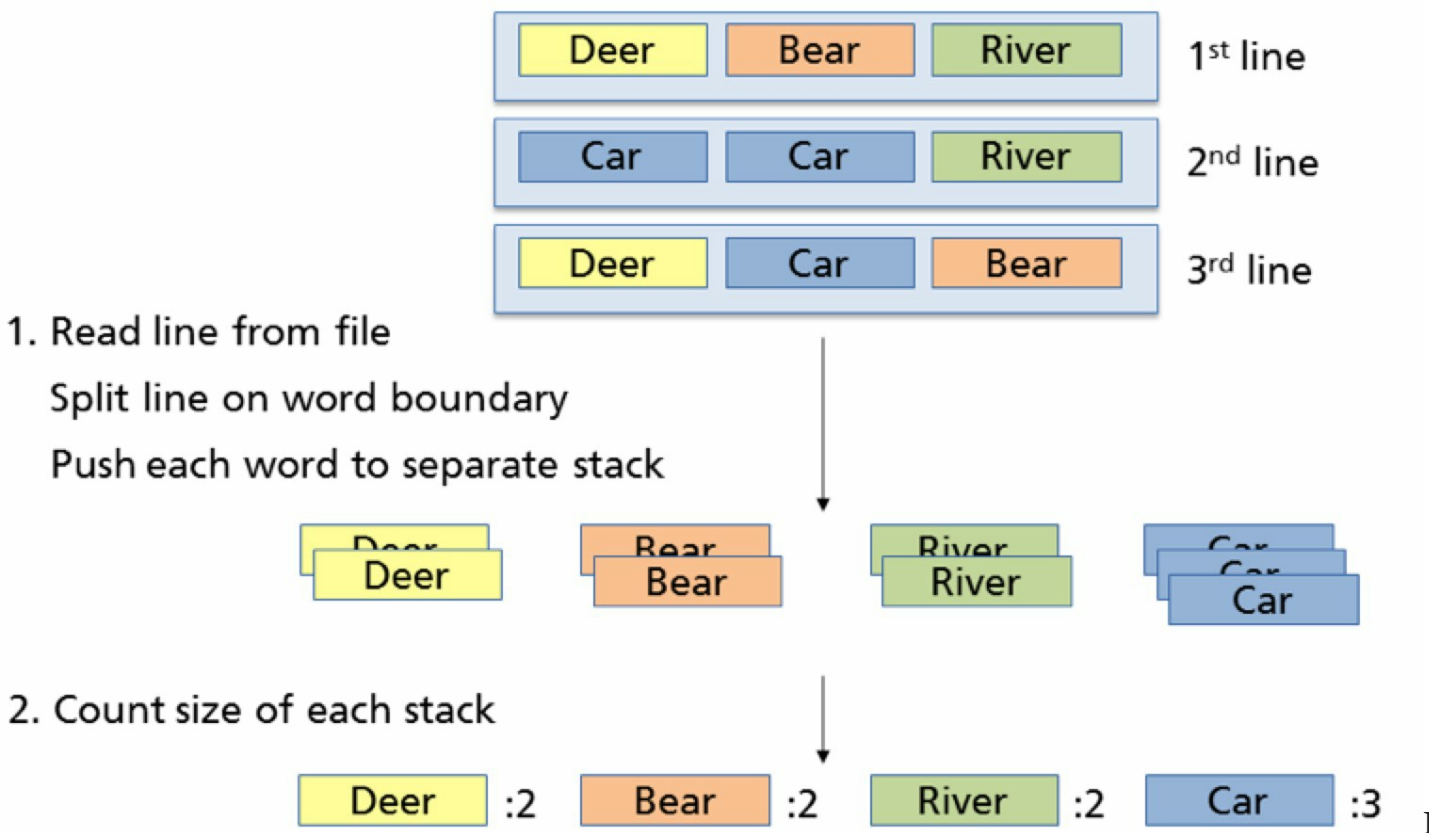


Figure 9: A sequential approach to counting words.

A MapReduce approach to the same task would break down the computation into map and reduce functions as follows:

Map phase: One map task is performed for each input line. This generates for each word in the line a key-value pair: (word, 1)

Reduce phase: One reduce task is performed for each distinct word. This sums as all the pairs (word, 1) for each specific word and thus produces the desired sum.

In this set-up, all mappers send data for the same word to the same reducer.

For our example data, this yields the following computation. First, the map jobs are executed as shown in Figure 10.

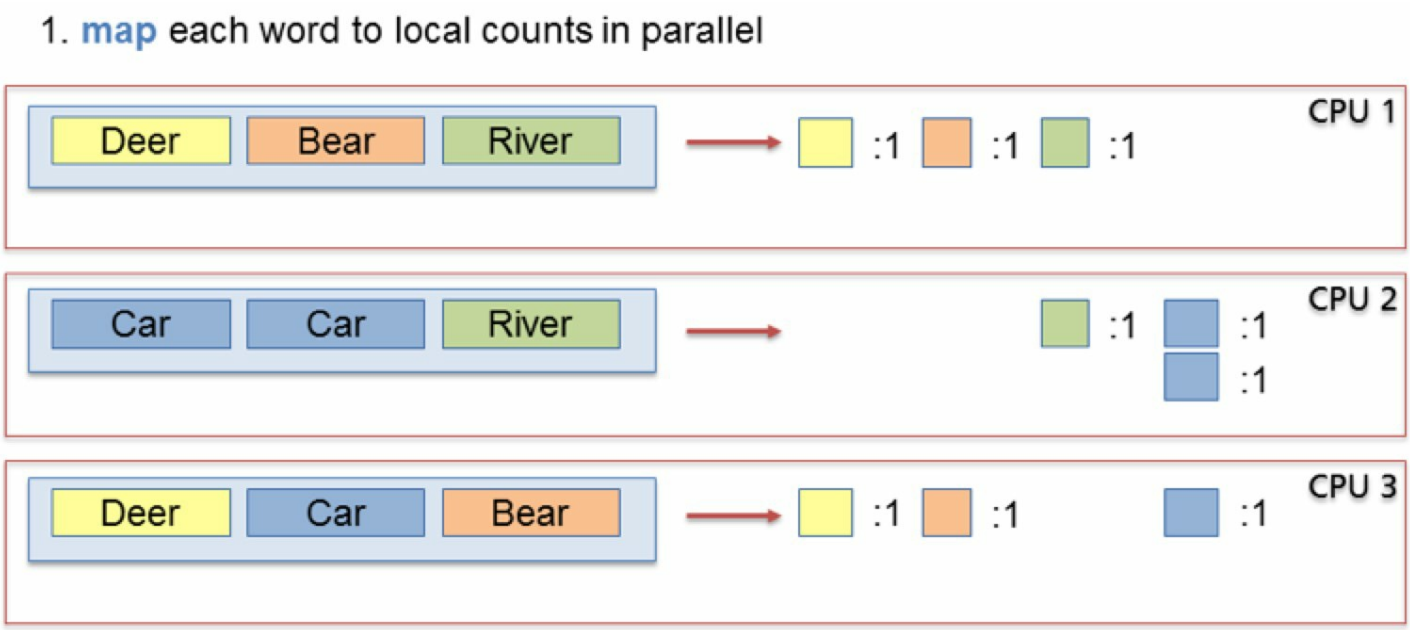


Figure 10: Execution of map jobs for the word count example.

Then the reducers compute the final result as shown in Figure 11.

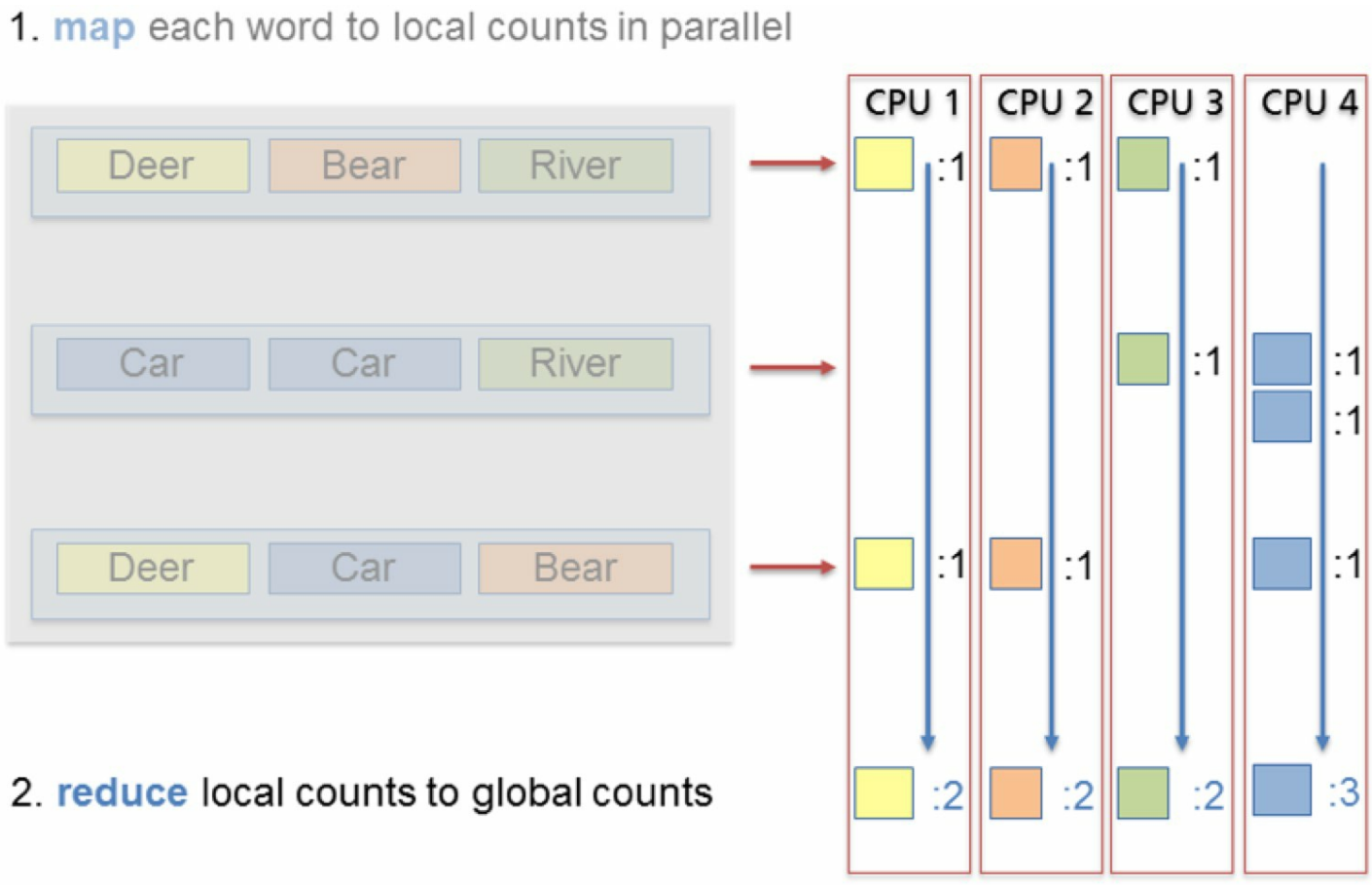


Figure 11: Execution of reduce jobs for the word count example.

Hadoop

Apache Hadoop is an implementation of Google's MapReduce framework. It is an open source project hosted by

the Apache Software Foundation. It is implemented in Java with bindings for many other languages. Commercial support and vendor specific additions are available. Hadoop simplifies the development of distributed data processing tasks and runs on commodity hardware.

Hadoop has two implementation layers: a distributed file system (HDFS) and an execution layer. The former covers process scheduling, coordination, monitoring, and restarting of distributed processes. The latter implements a distributed storage system that is fault tolerant and allows high data throughput.

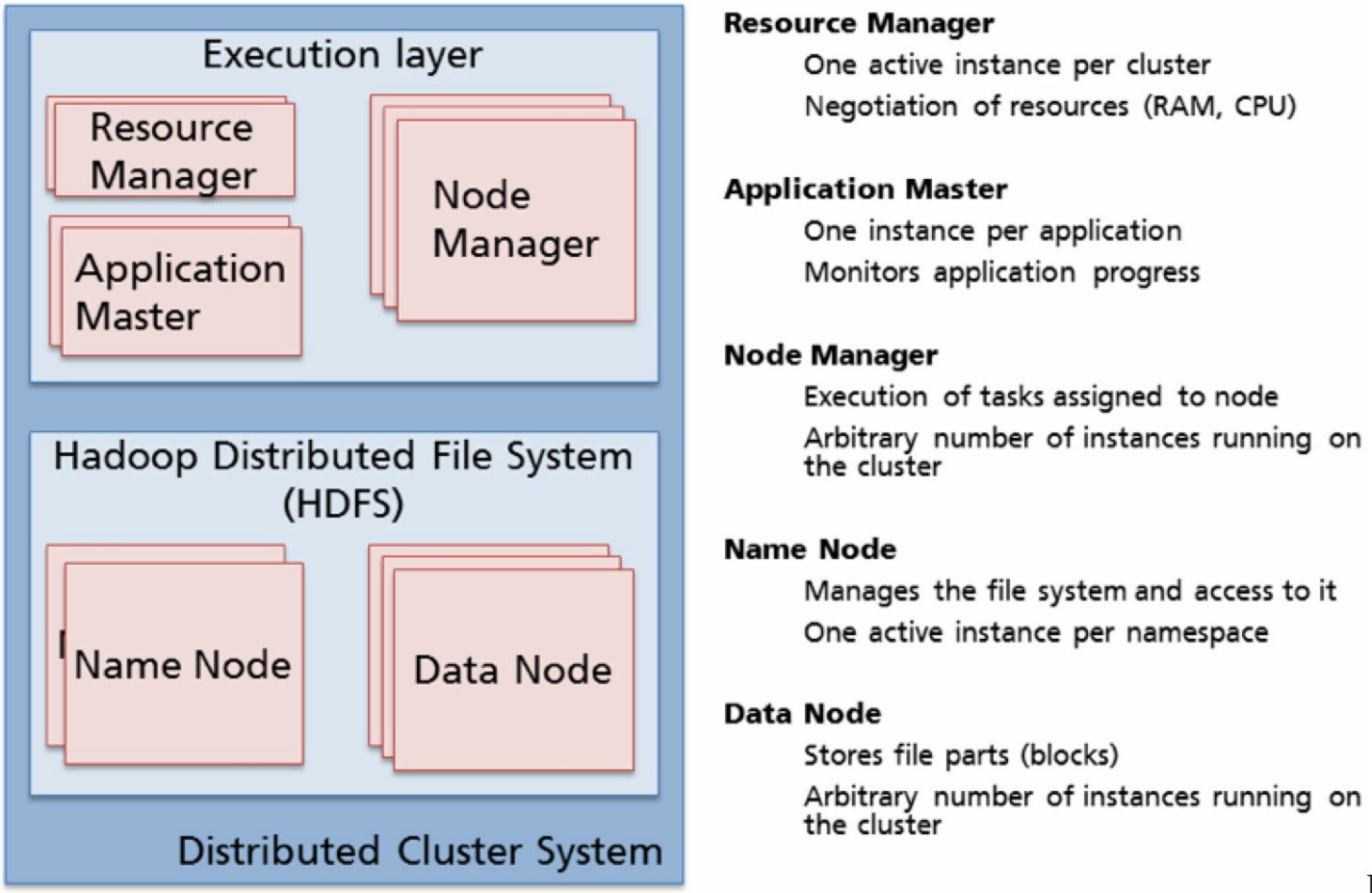


Figure 12: Overview of Hadoop components.

Hadoop Distributed File System (HDFS)

In the Hadoop file system, each file is split into blocks of size 128 MB (configurable). Such large blocks allow coherent space allocation on local disks and sequential read/write operations with high throughput. Data and metadata are separated as indicated below in Figure 13.

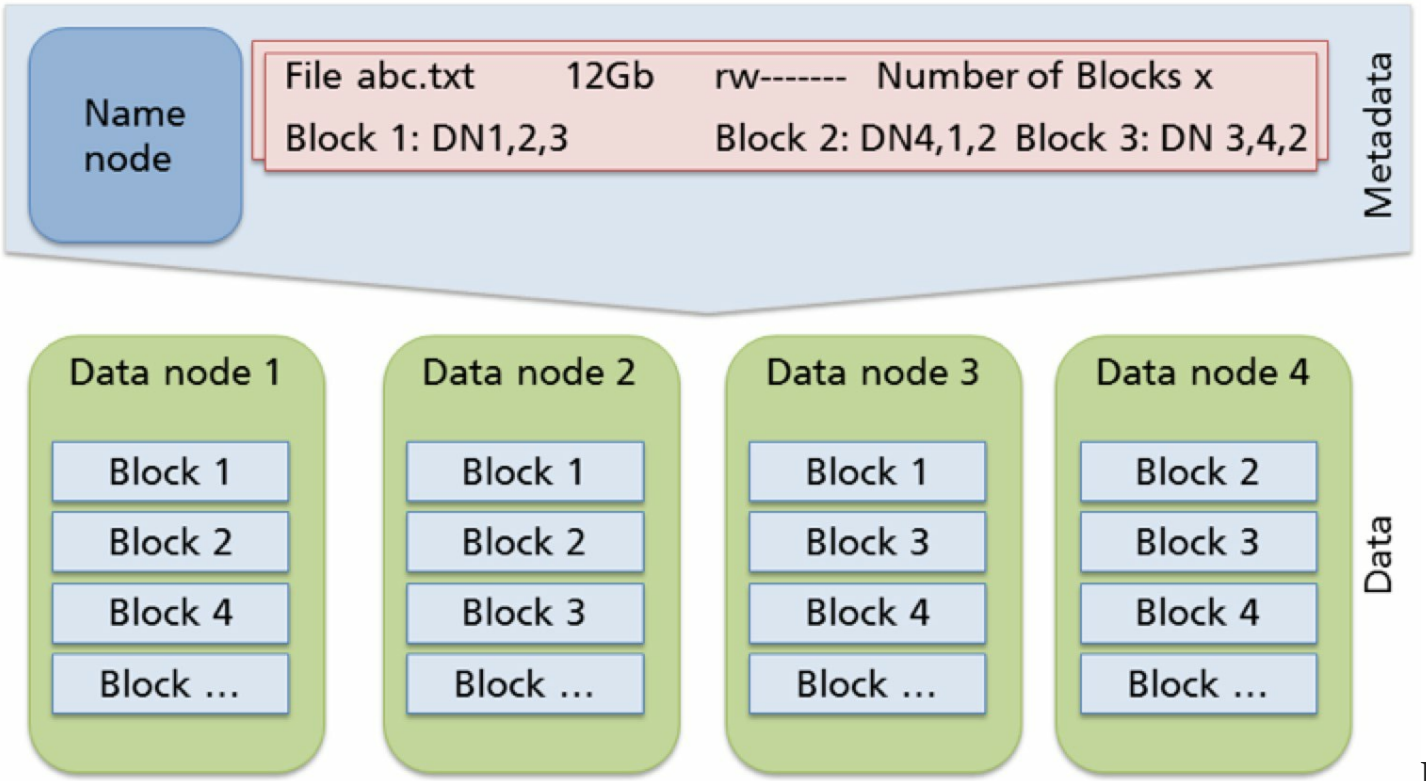


Figure 13: Data organisation in the Hadoop file system.

Each data node in the HDFS stores several file blocks and can use multiple persistent storage locations at the same time. It calculates checksums for each of the blocks it stores and sends heartbeat signals to the name node regularly. It may receive requests by the name node to replicate blocks to other data nodes. Each data block is stored redundantly on several data nodes. Therefore, the filesystem remains available in case of data node crashes.

The name node handles only file metadata (a comparably small amount of data). The complete file system image is kept in main memory for performance reasons. All edits, however, are written to persistent storage. The coordination of all distributed file operations is performed centralised by the currently active name node. Since Hadoop 2, horizontal scaling by partitioning the name space over several name nodes is possible.

A dedicated shell, the HDFS shell, allows access to the file system in a manner very similar to the shell commands used for file system commands in Linux.

MapReduce on Hadoop

The MapReduce scheme is implemented in Hadoop by providing Java methods for the map and the reduce function. The following code examples give a first feeling for how this is done. Hadoop's MapReduce Tutorial is a good starting point for more in depth information.

Let us start with a mapper for the word count example:

```
public static class Map extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, IntWritable> {

        //inputkeytype, inputvaluetype,outputkeytype, outputvaluetype

    private final static IntWritable one = new IntWritable(1);   // predefined classes for Text and Int.
    private Text word = new Text();



    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
Reporter reporter) throws IOException {
```

```
String line = value.toString();       // an input line

StringTokenizer tokenizer = new StringTokenizer(line);  // all words in the line

while (tokenizer.hasMoreTokens()) {   // for each word

                word.set(tokenizer.nextToken());

                        output.collect(word, one);

        }    // output key-value pair (word, 1)

    }

}
```

As in the example above, this mapper produces a stream of pairs (word, 1) for each word in the input data. Now, these are summed for each word by the reducer:

```
public static class Reduce extends MapReduceBase
        implements Reducer<Text, IntWritable, Text, IntWritable> {

 // inputkeytype, inputvaluetype, outputkeytype, outputvaluetype



    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter)   throws IOException

    {

int sum = 0;

        while (values.hasNext()) {      // iterate through list of input values

                        sum += values.next().get();

        }

        output.collect(key, new IntWritable(sum));

    }

}  // emit (word, sum)
```

Application Development on Hadoop

If the compute cluster is understood as the next generation of hardware then Hadoop is the operating system for that hardware. It is good at storing, processing, cleaning, and exporting data at an immense scale. Following this metaphor, MapReduce is the assembly language for the cluster hardware. In other domains, programming has long abstracted away from the assembly language level and the same is happening for big data applications.

One more abstract language for expressing jobs which can be parallelised using MapReduce is Cascading. This workflow engine on top of Hadoop is advertised as the "C" of the Hadoop ecosystem. It is sparse, elegantly composable, and low-level enough to solve large computing jobs but it also allows abstracting away many complications of MapReduce.

In Cascading, workflows are described in Java and the workflow steps can contain any Java code (e.g. data analysis). The individual steps can be composed by operators (e.g. groupBy, fork, join). These workflows are then compiled to MapReduce jobs in the form of Hadoop executable jar files.

In Cascading, inputs and outputs are abstracted as “taps”. In the simplest case, taps are mapped to HDFS files but can be any concept feeding data into or out of the system. Data itself is abstracted as “tuples”. Each tuple has a number of fields and each field has a name, its “fieldname”. Fields are quite flexible as they can contain a Java object of any type that is accessed via the field name. Each field may have an associated type declaration which is checked at configuration of the workflow.

Workflow steps are connected by “pipes” pretty much in the same way in which pipes work in Linux. For example, a pipe in Unix which counts txt files in a directory is:

```
ls | grep '\.txt$' | wc -l
```

The steps in a workflow can be composed by “operators”. We will describe a small subset relevant to our example application.

Basic operators include:

EACH: This operator requires a function working on a single tuple. It applies the function individually to each incoming tuple, possibly producing one or more output tuples. In MapReduce, this is translated into a mapper.

GROUPBY: Requires a field indicating the “key” to be used for grouping Repartitions and groups the data according to the indicated key Is translated to mappers outputting the data with “key”

EVERY: This operation can only be applied after a “GROUPBY” operation. It requires an “aggregator” working on all tuples of a group.

General aggregators can be implemented by providing a function start (called before any tuple comes in), a function aggregate (called on each tuple of a group), and a function complete (called after all tuples of the group have been processed). Aggregators can generate one or more output tuples.

There are also predefined functions and aggregators to combine data:

The *REGEX Generator function* emits a new tuple for every string (found in an input tuple) that matches a specified regular expression.

Example:
String regex = "(?<!\\pL)(?=\\pL)[^]*(?<=\\pL)(?!\\pL)";
Function function = new RegexGenerator(new Fields("word"), regex);

This splits a text line into words and emits each found word as a new tuple with a single field named “word”.

COUNT aggregator: This predefined aggregator operation counts how many tuples are in a group and emits a long-value when the processing of a group is completed.

On top of the language for defining workflows, there is the Cascading scheduler. It allows multiple flows to be chained. Such a cascade of flows is a directed acyclic graph. In it, independent flows will be executed in parallel. Dependent flows start as soon as the previous flow on which it depends is complete. If the source has not changed, a flow may be skipped.

The following flow chart shows are Cascading operations are combined in our example application to implement the workflow for extracting information from posting to the MOTOR-TALK forum as shown in Figure 14.

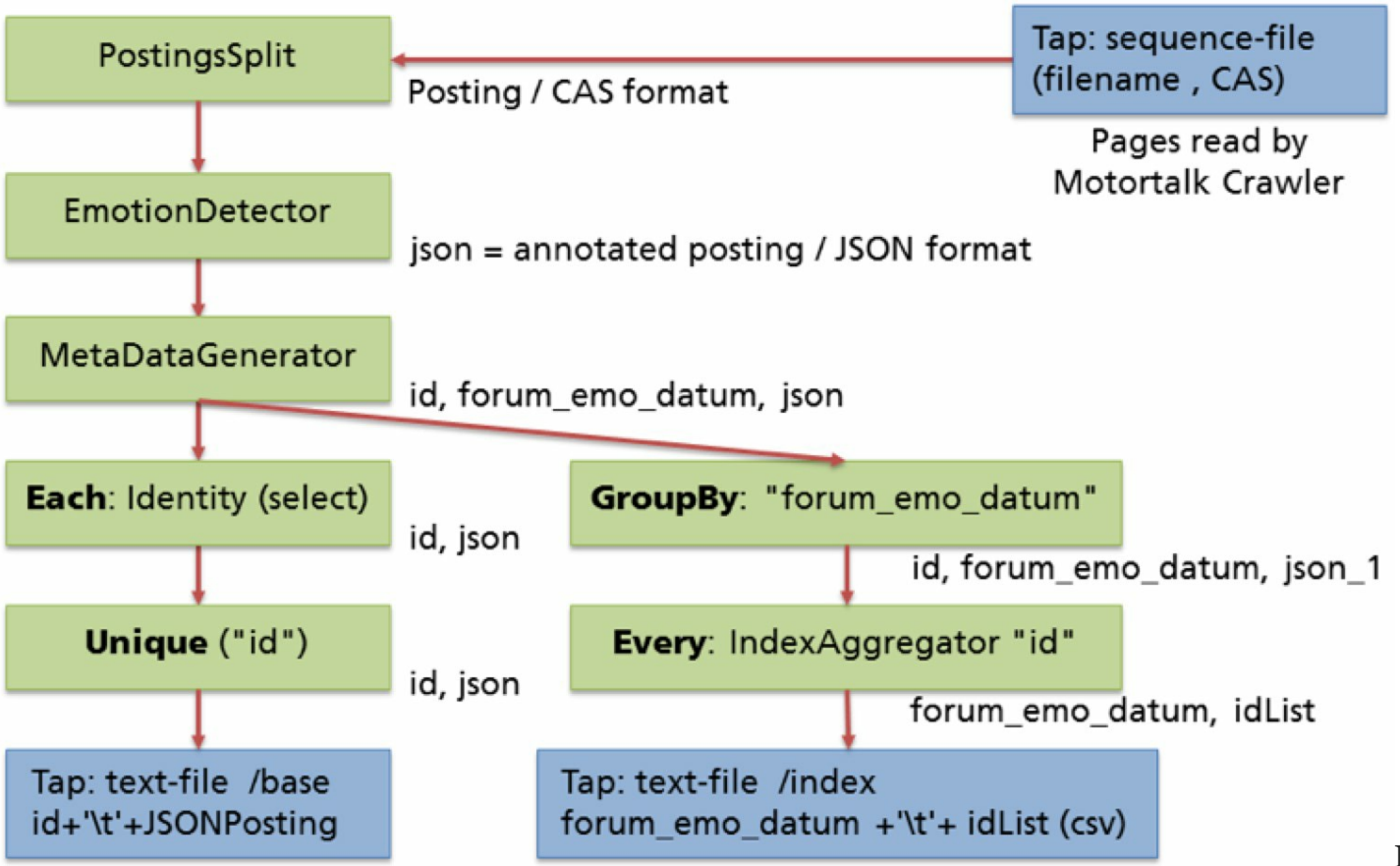


Figure 14: Workflow for our example application in Cascading.

Speed Processing

The next part to discuss in the lambda architecture is the speed layer. First, a messaging infrastructure suitable for big data systems is needed to pass data through this layer in a way that is scalable and fault-tolerant. Then, parallel stream processing is employed to handle real-time updates to the system. Finally, a database with high write performance is needed to handle frequent updates to the real-time view.

Big Data Messaging

As indicated in our architecture diagrams, a messaging system is needed to transport data to the different modules of the big data architecture. It should be capable of handling large amounts of data quickly and also provide measures to safeguard against data loss. The latter is often achieved by message buffering which also allows the playback of messages to components that need to be restarted or for debugging purposes.

In our example application, we have chosen Kafka as a messaging system. Kafka is similar to the Java Messaging System (JMS). However, it is faster, scales within a computing cluster, allows as many messages as desired, and messages are saved persistently on hard disk.

Messaging systems are often based on the publish/subscribe model. This is a communication model for loosely-coupled processes which are connected via named message queues or channels. At each channel there can be both message producer processes and message consumer processes. Producers generate messages and feed them into the channel. Consumers are informed about messages and receive their contents. Communication is anonymous as addressing works only by the queue name. Producers and consumers can register dynamically at a queue.

Parallel Stream Processing

A big data application usually makes use of several parallel data streams. In the speed layer, these have to be processed quickly to give the user immediate access. This requirement, however, means that buffering large parts of the input before running computations is not feasible in this part of the architecture. Speed layer processing is best modelled as stream processing in which each processor has only a constant amount of memory in terms of the size of the data stream.

Apache Storm is a real-time computation system suitable for using computer clusters to process unbounded streams of data. It is thus the speed layer equivalent of Hadoop for the batch layer.

The basic concept in Storm is the *topology*. A topology comprises the logic for a speed layer function. It is analogous to a MapReduce job. Its components are streams (data sequences), spouts (data sources), and bolts (processing units).

In Storm, data flows in so-called *streams* which are defined as possibly unbounded sequences of tuples. Similar as in the approaches above, a tuple contains a fixed number of objects. Streams can be named and connect spouts (the data sources in Storm) and bolts (the data processors).

Spouts feed data streams into the system and there are a number of predefined spouts in the Storm libraries. For example, there is a KafkaSpout feeding data from the Kafka messaging pipeline into storm topologies. Each spout receives information whether a tuple that it has fed into a topology has passed the topology (ACK) or failed (FAIL) within a predefined period of time. This allows handling data loss by resending tuples or by other concepts.

Bolts are programmable filters solving tasks like aggregation, analysis, and storage. They receive tuples from input streams, process them, and may emit new tuples to output streams. A bolt acknowledges (ACK) when a tuple has been received and processed.

Spouts and bolts are executed as many tasks across the cluster. The programmer specifies the degree of

parallelism per spout and per bolt. Storm then distributes and monitors the instances in the cluster. It is also possible to simulate a local cluster on a single machine to allow local development and debugging in Eclipse.

Similar to how data is distributed by Hadoop, Storm performs stream grouping to decide which tuple goes to which task. The topology determines the stream grouping for every consumer. There are several grouping options to select from:

If *shuffle grouping* is selected, tuples are distributed at random.

In *field grouping*, consistent hashing on a subset of tuple fields governs tuple distribution.

All grouping allows sending each tuple to all tasks.

Global grouping always picks the task with the smallest id for a tuple.

Figure 15 gives an overview of how to apply the concepts of Storm to create the speed function for our example application.

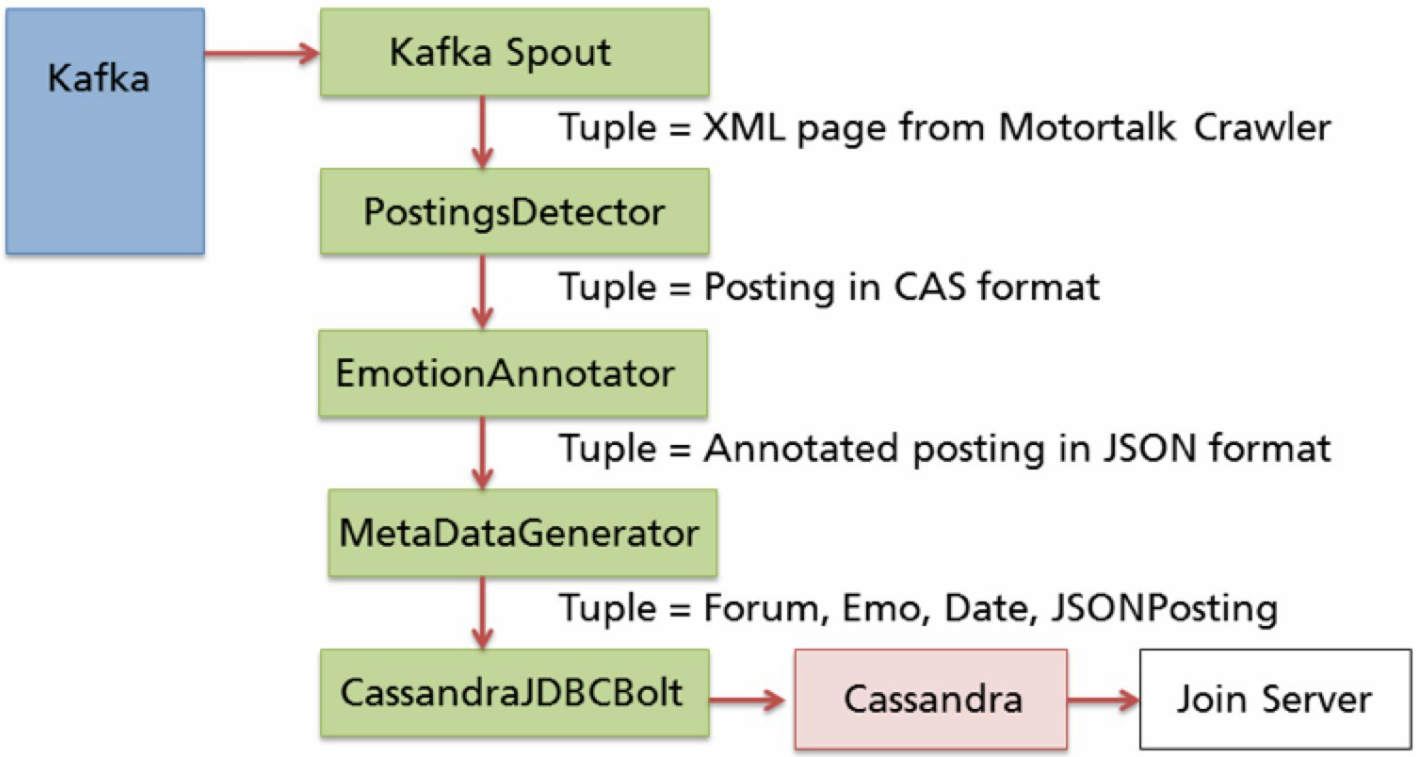


Figure 15: Workflow overview for the speed layer of our example application in Storm.

In this application, Storm receives a stream of XML pages and creates a stream of postings annotating them with metadata concerning joy or annoyance.

The first component of the workflow is a spout reading the pages from the Kafka messaging pipeline. This spout is connected to a chain of bolts realising the subsequent computations:

The PostingsDetectorBolt decomposes a page into a stream of postings.

The EmotionAnnotatorBolt analyses each posting with regard to emotions.

The MetaDataGeneratorBolt creates metadata to fill the database columns (Forum, emo, Date, ...).

The CassandraJDBCBolt writes each posting which is annotated by metadata into the Cassandra database.

The spouts and bolts are combined into a topology as shown in Figure 16.

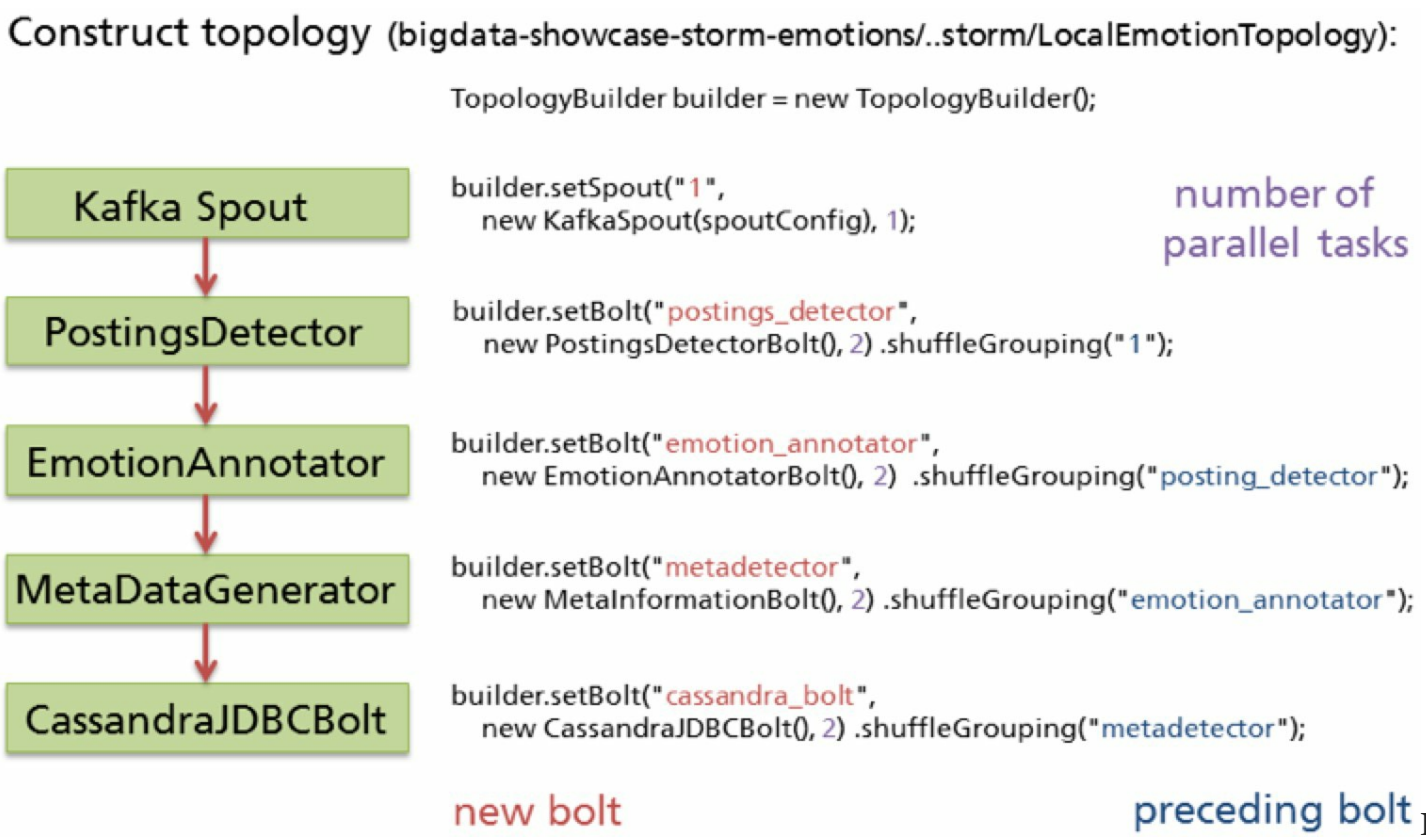


Figure 16: Constructing the Storm topology for our example.

Within our application, the bolts have no memory and bolts are connected with “shuffle” grouping. Let’s look at some source code for a bolt. Here is the EmotionAnnotatorBolt encapsulating the emotion detector.

Bolt

(bigdata-showcase-storm-emotions/..storm.bolts/EmotionAnnotaterBolt):

```
public class EmotionAnnotatorBolt implements IRichBolt {

        private OutputCollector collector;
        private EmotionDetector detector;


@Override       public void prepare (Map map,TopologyContext topologyContext,OutputCollector collector) {

// Is executed once

        this.collector = collector;
        this.detector = new EmotionDetector(); // Analysis class, same as in Hadoop/Cascading

}


@Override       public void execute (Tuple tuple) {

// Is executed for each arriving tuple, ...

}

@Override       public void declareOutputFields (OutputFieldsDeclarer declarer) {
// Defines output stream(s)
```

```
        declarer.declare(new Fields("json"));  // Tuple with a field named „json"
}
```

Now let us have a closer look at the execute function to see how incoming tuples are processed and new output tuples are created.

Bolt

(bigdata-showcase-storm-emotions/..stom.bolts/EmotionAnnotaterBolt):

```
@Override public void execute (Tuple tuple) {
        String text = tuple.getStringByField("cas");
        // posting is contained in tuple field „cas"

        DataTuple process = detector.process(new DataTuple(text));
        // posting is analysed, result is returned as JSON

        Values values = new Values();         // create a Storm tuple for output
        values.add(process.getString(0));  // with a field JSON text

        collector.emit(values);  // send output tuple to output stream
        collector.ack(tuple); // Done!
}
```

Similar to Cascading for Hadoop, there is a higher abstraction level on top of Storm called Trident. It provides features such as batching, states, and higher level workflow operators.

Real-time Views

In speed processing, high write performance of the database is of the essence lest the database become a bottleneck of the speed layer. Apache Cassandra is a big data database with high write performance. It exemplifies many of the aspects relevant for real-time views. It scales automatically, is flexibly extendable, and elastic. It is partition tolerant, has no single point of failure and the consistency levels are customisable by the user. Most importantly, it has high write performance.

Cassandra provides a distributed key-value store with basic operations to insert (PUT), retrieve (GET), and delete data. These operations work based on timestamps as described above.

A central issue in distributed databases is data replication. In Cassandra, replication is implemented as a hash ring. Computers are organised in a ring and each computer has and id. The main replica of a data item is stored on a machine identified based on the hash value of the data key. Additional replicas are stored clockwise in the ring. Cassandra provides two main parameters to influence the replication strategy:

Replication factor: This indicates how many replicas of each key-value pair are stored.

Strategy class: This determines whether replicas are stored successively along the ring or whether the network topology is taken into account, so that it can be ensured to have replicas in different data centres or in different racks of single data centre.

All data is combined with a timestamp. The GET operation is evaluated such that it provides that value to a key with the newest timestamp from the replicas that could be read. Timestamps are set by the client during the PUT operation.

Cassandra can be configured to use the quorum concept to ensure consistency. In this case, write operations are only considered successful if more than half of the replications are written successfully (acknowledged by the data nodes). Write operations are atomic on the row level, i.e., writing or updating the values in a single row is considered as a single write operation.

Applications must be designed to avoid concurrent updates. If present, concurrent updates must not cause problems within the application. In our application example, Storm only writes to the database and every line is written exactly once. The join server on the application backend only reads from database.

Even though Cassandra is a key-value store, it supports tables with columns and a custom structured query language called CQL with SQL as its role model. Initially, columns are just names attributed to the different elements of the data tuples which are then assumed to be of equal length. Since CQL3 there is limited support for queries with WHERE clauses as in SQL. However, such queries can only be answered in sufficiently fast if the WHERE clause respects data partition. Under certain conditions, it is easy to extract consecutive data from a column. In this context, the order of columns is important. In practice, this means that WHERE clauses can include comparisons on columns other than the first but only as long as all previous key-component columns have already been identified with strict equality comparisons. The last given key component column can then be any sort of comparison.

To close our discussion, let us have a look at how some of the queries used in our example application look like in CQL. First, we create a table for forum postings:

```
CREATE TABLE postings (

forum text,
emo text,
date text,
id text,
json text,

PRIMARY KEY (forum, emo, date, id)

)
```

Note how the primary key is given to support queries with WHERE clauses.

Now we can do queries on this table:

```
SELECT COUNT(*) FROM postings;

SELECT id,forum,emo,date FROM postings WHERE forum = 'Audi';

SELECT id,forum,emo,date FROM postings  WHERE
        forum = 'Audi' AND emo = 'pos' AND date > '200120101';
```

The last query shows how WHERE queries need to follow the order of columns in the primary key. The larger than statement for the date field is only possible because equality conditions are given for all the preceding columns of the primary key.

Big Data Analytics

Data Analytics is more and more seen as key driver in business excellence. This leads to a great demand in data scientists who are proficient within this field. However, the push towards big data also means that the traditional analytics tools that run on data in the memory of single computer are no longer sufficient. In this module, we provide an overview of approaches facilitating data analytics on huge datasets. We present different strategies including sampling to make classical analytics tools amenable for big datasets, but also analytics tools that can be applied in the batch or the speed layer of a lambda architecture, stream analytics, and commercial attempts to make big data manageable in massively distributed or in-memory databases. Based on this chapter, you should be able to realistically assess the application of big data analytics technologies for different usage scenarios and start with your own experiments.

This chapter is based on the contents of the following training course on Big Data Analytics: http://www.iais.fraunhofer.de/bigdataanalytics.html

Introduction

Big data analytics solutions ask for skills from at least two different fields. First, information technology skills are needed to provide horizontally scalable systems for storing and processing data. Second, data analysis skills are needed and, in particular, tools and methods for the mathematical modelling of data. In this module, we assume some basic knowledge from the first set of skills, in particular about the concept of lambda architecture. This is provided in the module Big Data Architecture. Based on this, we will focus on data analytics and how to incorporate it in big data systems.

MOVIE 4.1 Introduction to Big Data Analytics

We make use of a running example – collaborative filtering – which provides illustrations for many aspects of big data analytics. The use-case in our example is to provide recommendations for products based on user preferences. More concretely, we cover the task of music recommendation based on the last.fm dataset. We use the so-called 1K users dataset containing data from 1000 users over several years (available here). It provides user/artist pairings based on listening events and can be used to generate artist recommendations. This data can be represented by a so-called utility matrix shown in Figure 1.

← Artists → \ ← Users →				
	1	0	1	0
	1	0	0	1
	0	1	0	1
	0	1	0	1
	1	0	1	0
	0	1	0	1

Figure 1: Utility matrix associating artists by the users listening to them.

This matrix has one row for each artist and one column for each user. Whenever a user has listened to a given artist, the respective matrix entry associating this user with the given artist is set to one otherwise it is set to zero.

Now we need to find a way to measure the similarity of two artists in order to be able to recommend artists which are similar to the ones a user has already listened to. The basis for measuring artist similarity is to define two artists as similar if they have a similar set of listeners. In terms of our matrix, this means that the rows representing artists should be similar for similar artists.

The Jaccard similarity is one way to make this precise. Mathematically, the Jaccard similarity of two sets and (in our case two sets of users for two different artists) is given as the ratio of the size of the intersection of the two sets to the size of their union:

$$J(A_1, A_2) := \frac{|A_1 \cap A_2|}{|A_1 \cup A_2|}$$

Graphically, this can be understood by Figure 2, where it is defined as the ratio of joint users $|A_1 \cap A_2|$ to the total number of users $A_1 \cup A_2$.

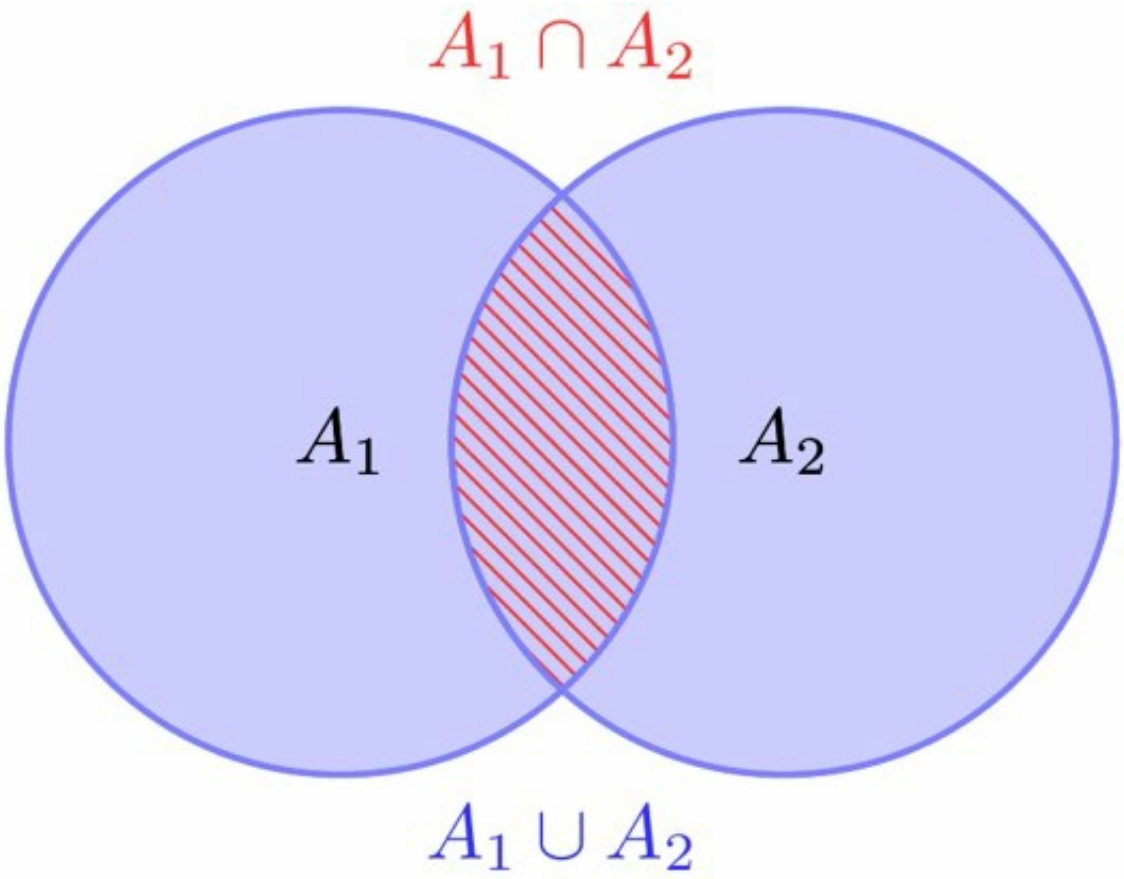


Figure 2: The Jaccard similarity.

The Jaccard similarity is a number between zero (very dissimilar) and one (very similar). Hence, it is easy to define a distance measure based on it, the Jaccard distance, which is simply defined as . This distance measure can be used in machine learning algorithms such as clustering.

Let us see how the Jaccard distance works out for example in Figure 3.

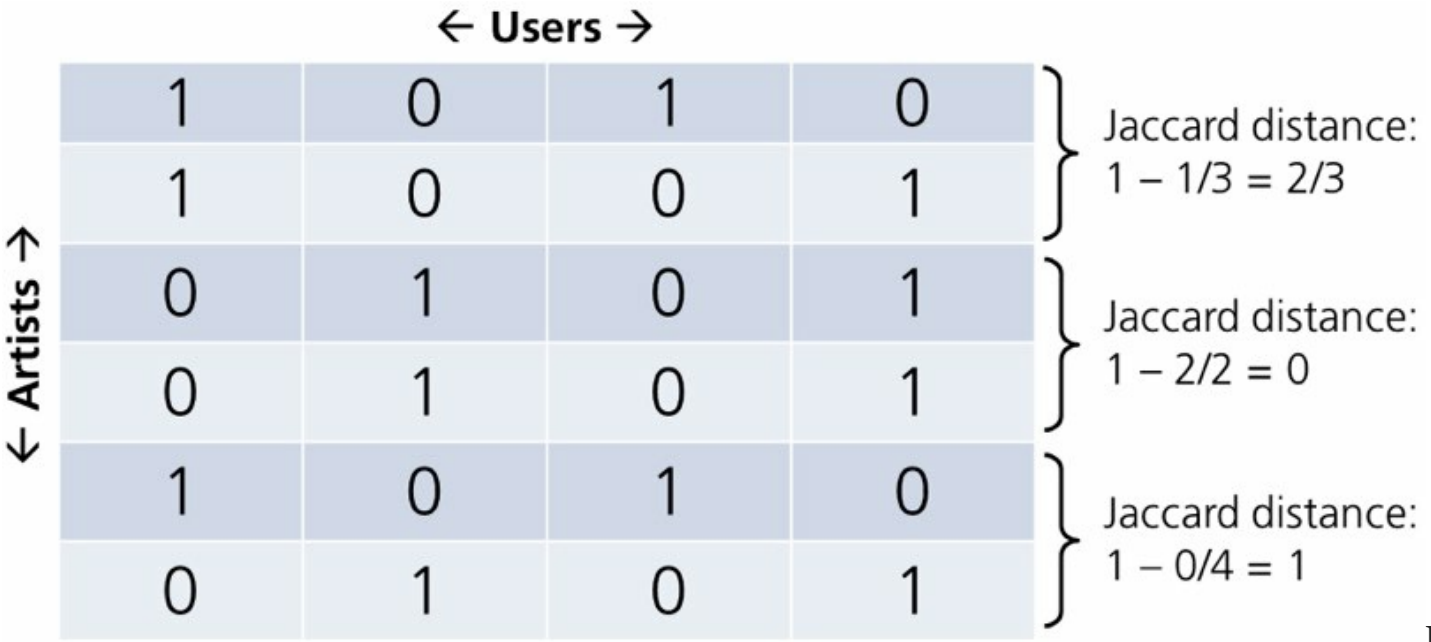


Figure 3: Some of the Jaccard distances in our example.

In the following, we will show two ways of applying these ideas in the big data context. On the one hand, sampling can be used to fit big data sets into classical analytics tools. On the other hand, big data analytics solutions provide solutions that can be applied to the whole data set. Both approaches can be integrated in a lambda architecture and we will show how to do this for the first case.

Collaborative Filtering In The Lambda Architecture

In this section, we will follow key steps in the Cross Industry Standard Process for Data Mining (CRISP, see Figure 4) to show how to go from our task – generating recommendations of artists based on the last.fm data set – to an implementation of a recommender system in a lambda architecture.

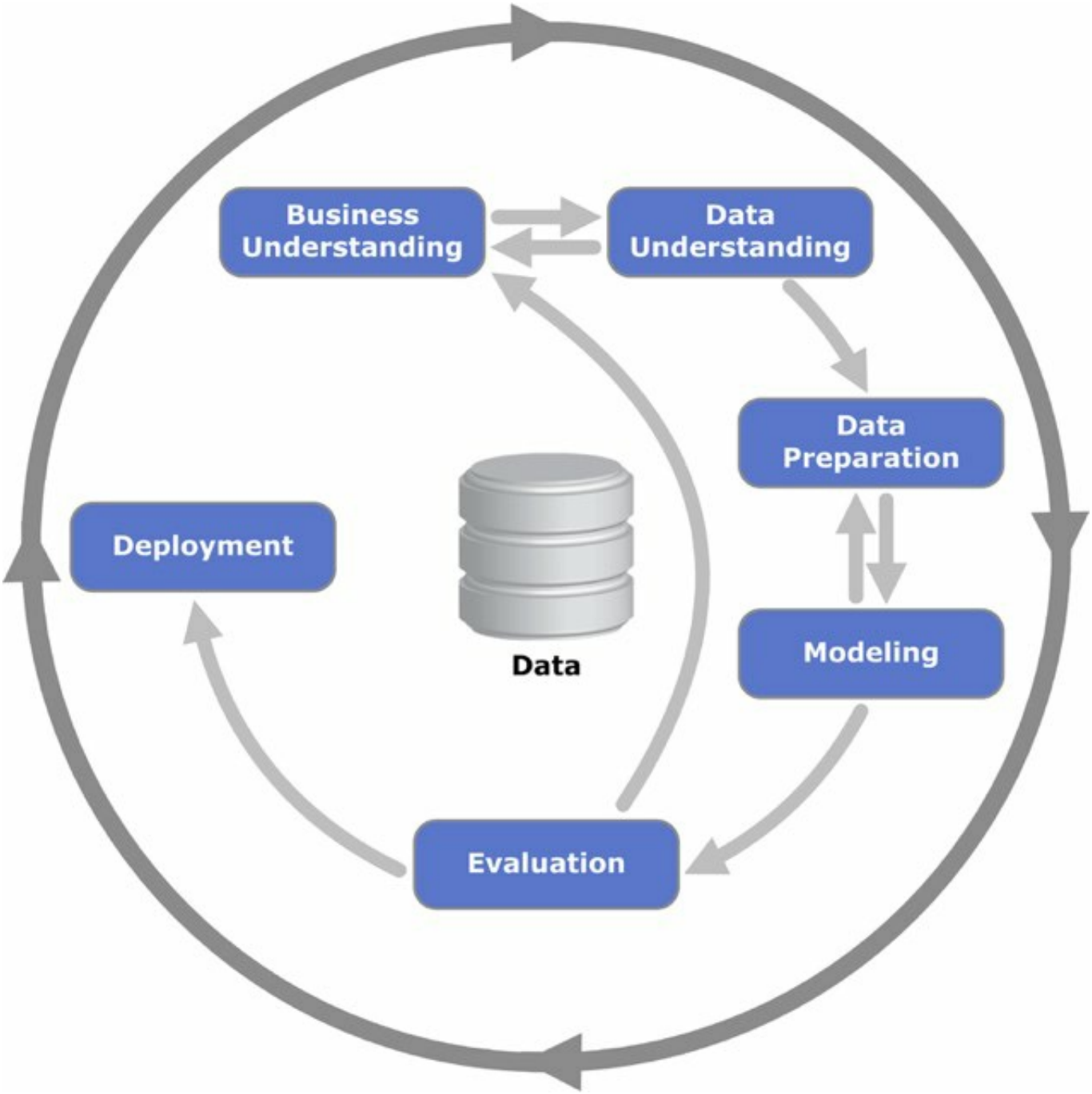


Figure 4: The Cross Industry Standard Process for Data Mining. Figure by Kenneth Jensen - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=24930610

Data understanding

The first step from CRISP that we will look at is data understanding. Often, crucial questions about the data can

be posed most easily as SQL statements. For example, we would like to know how many distinct users and artists are represented in our data set and how many distinct users are listening to a given artist. In SQL, this is readily found out using the following queries:

```
SELECT count(DISTINCT(artistname)) FROM datatable;

SELECT count(DISTINCT (userid)) FROM datatable;

SELECT artistname, count(DISTINCT (userid)) FROM datatable GROUP BY artistname;
```

However, our data is potentially much too large to fit inside a relational database. In our example, it is available as a file in the Hadoop File System. In this file, a listening event consists of the following entries given as tab separated strings:

Username (user_0001 to user_1000 in our data sample)

Timestamp (in ISO format)

Artiid (artist unique ID from the MusicBrainz database: http://musicbrainz.org)

Artistname

Track-id (track unique ID from the MusicBrainz database)

Trackname

For example, a line in our file might look like this:

```
user_000639 \t 2009-04-08T01:57:47Z \t MBID \t The Dogs D'Amour \t MBID \t Fall in Love Again?
```

Cascading Lingual is an extension to Cascading that provides an ANSI SQL interface for Apache Hadoop. It allows us to work with such text files as if they were tables in a database. Cascading itself is a workflow language on top of Hadoop that will be described in the next section. We will now apply Lingual for data understanding.

Lingual has a dedicated command line tool for DDL operations, called catalog. A new data source such as our file of last.fm listening events needs to be registered using this tool. This is done by defining the table structure (column names and types). This is called a stereotype in Lingual and can be imposed on a data file. Here is how this looks in our case:

```
 5 lingual catalog --schema LAST_FM --add
 6 lingual catalog --schema LAST_FM --stereotype listen_evts_s --add \
 7   --columns UID,DATETIME,ARTIST_MBID,ARTIST_NAME,TRACK_MBID,TRACK_NAME \
 8   --types string,string,string,string,string,string
 9
10 lingual catalog --schema LAST_FM --table LISTEN_EVENTS --stereotype listen_evts_s \
11   --add last.fm/clean.tsv
```

Once a schema has been defined, we can access the data. Lingual comes with a shell very similar to those provided together with relational databases. It allows performing ANSI SQL queries on the data. For example, we can find the number of listening events in the data as follows:

```
livlab@livlab-VirtualBox:~$ lingual shell


Concurrent, Inc - Lingual 1.1.0
only 10,000 rows will be displayed
sqlline version 1.1.6
0: jdbc:lingual:hadoop> SELECT count(*) FROM last_fm.listen_events;
{utcTimestamp=1403707456573, currentTimestamp=1403714656573, localTimestamp=1403714656573, timeZone=sun.util.calendar.ZoneInfo[id="Europe/Berlin"
,offset=3600000,dstSavings=3600000,useDaylight=true,transitions=143,lastRule=java.util.SimpleTimeZone[id=Europe/Berlin,offset=3600000,dstSavings=
3600000,useDaylight=true,startYear=0,startMode=2,startMonth=2,startDay=-1,startDayOfWeek=1,startTime=3600000,startTimeMode=2,endMode=2,endMonth=9
,endDay=-1,endDayOfWeek=1,endTime=3600000,endTimeMode=2]]}
+-----------+
|  EXPR$0   |
+-----------+
| 19150867  |
+-----------+
1 row selected (493.283 seconds)
0: jdbc:lingual:hadoop>
```

If we are to

answer our SQL queries from above, we can simply perform them in Lingual:

```
SELECT count(DISTINCT artist_name) FROM last_fm.listen_events;
-- result:
+---------+
| EXPR$0  |
+---------+
| 174052  |
+---------+
1 row selected (499.498 seconds)


SELECT count(DISTINCT uid) FROM last_fm.listen_events;
+---------+
| EXPR$0  |
+---------+
| 992     |
+---------+
1 row selected (449.482 seconds)

SELECT artist_mbid, count(DISTINCT uid) count_dist
                FROM last_fm.listen_events GROUP BY artist_mbid ORDER BY count_dist;

+--------------------------------------+-------------+
|              ARTIST_MBID             | COUNT_DIST  |
+--------------------------------------+-------------+
| 02b4a39e-04c0-4eb7-a0eb-c919f3d14154 | 70          |
| 01e60eba-52df-4694-8f09-39f43abe54e9 | 71          |
```

Behind the scenes, Lingual translates these SQL statements into code for Cascading which is in turn translated into MapReduce jobs in Hadoop.

Lingual provides a JDBC driver so that it can be accessed from any JDBC aware application, for instance from a desktop SQL client like Squirrel. In the context of data understanding, it is even more useful that it can be accessed from R via the RDBC package.

```
drv <- JDBC("cascading.lingual.jdbc.Driver", cp)
connection <- dbConnect(drv, "jdbc:lingual:hadoop;")

# query the repository

df <- dbGetQuery(connection, "SELECT artist_mbid, count(DISTINCT uid) count_dist
                FROM last_fm.listen_events GROUP BY artist_mbid ORDER BY count_dist")
head(df)
summary(df)

m <- ggplot(df, aes(x=COUNT_DIST))
m <- m + ggtitle("Distribution of distinct listeners on artists")
m + geom_histogram(binwidth=1, aes(y=..density.., fill=..count..)) + geom_density()
m + geom_histogram(binwidth=1, aes(y=..count..))
```

This gives us a plot of the distribution of the number of artist for a given number of distinct listeners, as shown in Figure 5.

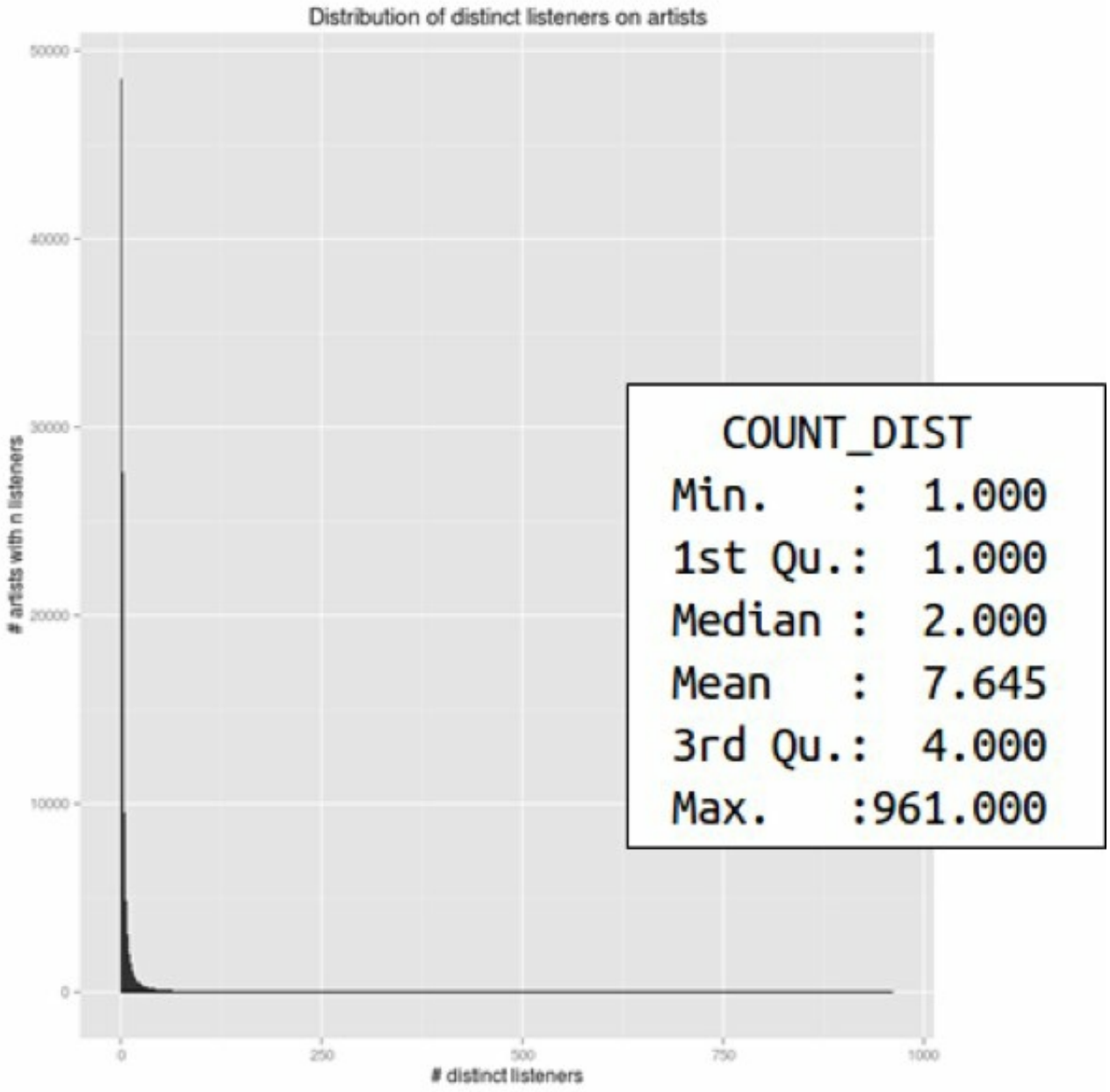


Figure 5: Number of artists (vertical axis) having a given number of distinct listeners (horizontal axis).

What we see in the plot is a typical long-tail distribution and it shows us that about 50% of all artists have less than two listeners. In a real business case, this should make us think hard about whether the given data can indeed give us the kind of information we are looking for. As we are using this data only as an example and for pedagogical reasons, we carry on, ignoring that the data distribution is not entirely optimal.

Before we carry on, there is one nice thing to observe about Lingual: queries can use multiple different sources and thus, for instance, combine RDBMs and HDFS text files. Thus, if there is data in different databases and/or the Hadoop File System, these can all be utilised within the same SQL query in Lingual.

Computing the utility matrix

Building the utility matrix for collaborative filtering is a typical batch processing task. This can be modelled on top of Hadoop using Cascading. Cascading provides a workflow abstraction on top of Hadoop that is easier to use than native Hadoop and more flexible than SQL. It is easiest to get a feeling for how Cascading is used by following how to compute the utility matrix with it.

As a result, we want to get a line-wise representation of the utility matrix such that each line is defined as

follows:

artistname, 0,1,0,0,1,1,0,0, ….,1

We need one line per artist in the input data and as many columns as there are users (1000 in our sample of the last.fm dataset) and each column has an entry that is either zero or one. The i-th column is one if and only if user i is in the user set of the artist with the given artist name.

In Cascading, data is abstracted as tuples where a tuple has a number of fields. Each field has a name ("fieldname") and contains a Java object (of any type) that is accessed via the field name. Workflow steps can contain any Java code working on tuples (e.g. data analysis). They are connected by "pipes" similar to those in the Unix bash. For example in Unix

```
ls | grep '*.txt' | wc -l
```

counts the txt files in a directory. However, in Cascading these pipes can be more complex "operators" like groupBy, fork, or join.

A Cascading workflow compiles to MapReduce jobs in Hadoop in the form of an executable jar file.

Data flows into a Cascading workflow by using connectors called taps. For example, those taps can be fed by a reader which reads from a text file in the Hadoop file system and translates this data into tuples with named fields based on a schema provided by the user:

```
Fields sourceFields = new Fields("uid", "datetime", "artist_mbid",
        "artist_name", "track_mbid", "track_name");
Scheme sourceScheme = new TextDelimited(sourceFields);
Tap source = new Hfs(sourceScheme, inputPath);
```

The pipes available in Cascading comprise Each, Merge, GroupBy, Join, CoGroup, Every, and SubAssembly. In our example, we will use the following three types of pipes:

1. *Each*: This pipe requires a function working on a single tuple. It applies the function individually to each incoming tuple, producing zero, one, or more output tuples. In Hadoop it is simply translated to a mapper.

2. *GroupBy*: The pipe requires a field indicating the key to be used for grouping. It repartitions and groups the data according to the indicated key. In Hadoop, it is translated to mappers outputting the data with the respective key.

3. *Every*: This can only be applied after a GroupBy pipe. It requires an aggregator working on all tuples of a group. Every can generate one or more output tuples. This aggregator implements the methods

Start, called before any tuple comes in,

Aggregate, called on each tuple of the group, and

Complete, called after all tuples of the group have been processed.

Now we will compute the utility matrix in two steps: computing the user sets for each artist and translating these into string representations of the matrix lines.

Step 1: Computing the user set per artist name is similar to the following SQL query:

```
SELECT artistname, userid FROM datatable GROUP BY artistname;
```

In Cascading it can be expressed using the operators "Group By" and "Every":

```
Pipe listenEvts = new Pipe("listenEvts");

listenEvts = new GroupBy(listenEvts, new Fields("artist_name"));
```

```
listenEvts = new Every(listenEvts, new
JaccardSetAggregator(new Fields("userset"), "uid"), new Fields("artist_name”,"userset"));
```

Step 2: Translating each user set into a string representation for the lines of the utility matrix is just a for-loop over all possible user names. It can be expressed using the operators Each and a function convertSetToLine which produces a string representation of the user set:

```
listenEvts = new Each(listenEvts, new Fields("userset"),  new convertSetToLineFunction(new
Fields("userArray")), new Fields("artist_name", "userArray"));
```

After this workflow is translated to Hadoop by Cascading and run on the data set, the resulting utility matrix will be available as a file in the Hadoop File System.

Sampling

Classical data analytics software cannot handle huge amounts of data. A possible solution is given by sampling a subset of the dataset which can then be analysed in classical tools using RHadoop. The actual cluster model for the batch view is computed in R whereas the application of this model is again performed using Cascading. The size of the final model is small enough such that validation is possible in R via Lingual.

The Jaccard distance is not directly suitable for the standard k-means clustering algorithm and density based clustering like DBSCAN is not particularly well suited to fit the MapReduce parallelisation (neither agglomeration nor decomposition). Therefore our approach for this module is to take a sample of the utility matrix and perform standard (local) clustering with the Jaccard distance in R.

For each cluster, we will compute the medoid as exemplar representing the cluster. The medoid of a cluster is defined as the data point with the smallest sum of squares distances to all other nodes in the same cluster.

Sampling directly speeds up model building and often more data does not generate better models (see Figure 6). Therefore, using the full dataset is often not required: additional instances yield diminishing returns. However, sampling possibly introduces a bias and it has the danger of under sampling relevant instances.

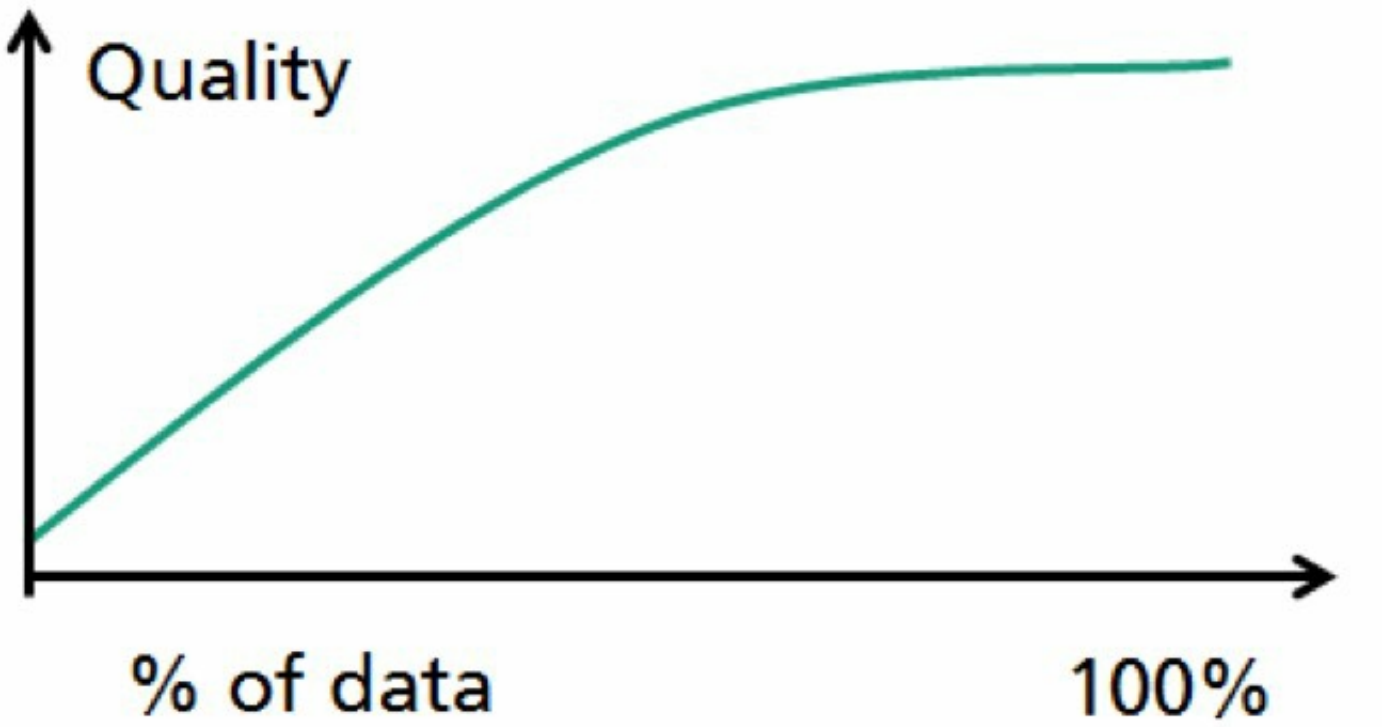


Figure 6: Typical relationship between the amount of data used to build a model and the quality of the model.

There are several strategies for sampling. In random sampling one selects sample data by drawing uniformly at random. It is hoped that if the sample is large enough, the statistical distribution of the whole data set is retained automatically. Of course, this can introduce a bias.

There are several strategies to overcome such bias. For example, in stratified sampling data is selected such that the sample reflects the distribution of the original data with respect to specific attributes (e.g. maintain the original ratio between female/male persons in the sample).

Oversampling may be used to represent specific cases in the sample more often than their occurrence in the data set suggests when it is clear that these cases are underrepresented for some reason. For example in fraud detection, it may be necessary to include several times more fraudulent transactions than in the original data.

Before implementing sampling for big data, there should be a further data understanding step, for example in Lingual. For random sampling, this should determine the relevance and occurrences of the attributes to be sampled. For stratified sampling it should determine the distribution of attribute values. In our example this would be the user counts per artist (see data understanding discussed earlier).

On the SQL level, random sampling is often performed like this:

```
SELECT * FROM datatable ORDER BY Rand() LIMIT 1000;
```

However, the Rand function is not supported by Lingual and therefore we need a lower level implementation. Here, Cascading would be a good option but we have already introduced this tool. To introduce another good big data tool, let us introduce RHadoop, an R package that allows writing MapReduce jobs directly in R.

After installation (a good starting point for installing RHadoop is this tutorial), it can be made available in R as follows:

```
## Run first time only
#install.packages('rJava')
#install.packages(c( "Rcpp", "RJSONIO", "bitops", "digest",
# "functional", "stringr", "plyr", "reshape2"))

## Install R packages via src, needs gcc & JDK
#R CMD INSTALL rmr2_2.3.0.tar.gz
#R CMD INSTALL rhdfs_1.0.7.tar.gz
```

The integration of RHadoop with a given Hadoop installation is done using the Hadoop streaming API. It is necessary to have R and the RHadoop package installed on every cluster machine.

Then, RHadoop is enabled in an R session by setting the relevant environment variables and including the relevant libraries:

```
# Set required env vars to determine hadoop to use
Sys.setenv("HADOOP_HOME"="~/software/hadoop-1.2.1")
Sys.setenv("HADOOP_CMD"="~/software/hadoop-1.2.1/bin/hadoop")

library(rmr2)
library(rhdfs)
```

Let us see how to write a mapper for randomly sampling 10% of the last.fm dataset:

```
sampleSize <- 0.10

map <- function(., lines) {
  N <- nrow(lines)
  n <- N*sampleSize
  results <-lines[sample(N, size = n),]
  keyval( NULL, results)
}

rnd_sample <- function (input, output=NULL) {
  mapreduce(input=input, output=output,
            input.format = make.input.format("csv", sep = "|"),
            map=map, combine=F)
}
```

In this example, each mapper – in the form of the map function – receives in parallel a subset of the whole dataset (variable lines) and uses the sample function from R to randomly select 10% of them. In this example we do not need a reducer and hence do not use one (combine=F).

To apply the MapReduce version rnd_sample of the sample function, we need to specify the data within the Hadoop File System and apply our function to it:

```
## read input file from folder last.fm
## save result in folder last.fm/sample/
## Submit job

hdfs.root <- 'last.fm'
hdfs.data <- file.path(hdfs.root, 'matrix.psv')
hdfs.out <- file.path(hdfs.root, 'sample')
out <- rnd_sample(hdfs.data, hdfs.out)


## Fetch results from HDFS
results <- from.dfs(out)
a <-head(results$val)
```

In this way, RHadoop provides an interface between a Hadoop Cluster and R and allows MapReduce functions to be written in R. We can therefore access and work on data stored in Hadoop.

Building a cluster model in R

Now that we have used RHadoop to draw a sample of the utility matrix, we have this sample available as an object in R. Hence, we can use the R package “flexclust” for clustering the sample utility matrix based on the Jaccard distance. This can be done in just a few lines of R code:

```
mycont <- list(iter=2, tol=0.001, verbose=1)
opts <- as(mycont, “flexclustcontrol”)

cl1 <- kcca(data, k=50, family=kccafamily(“jaccard”),
control=opts)
```

```
centers <- cl1@centers

write.csv(centers, "centers_1000_iter_2_50.csv")
```

For each cluster, compute the medoid as example representing the cluster (e.g. node with the smallest sum of square distances to all other nodes in the cluster).

One way to assess the quality of a clustering result is silhouette plot analysis. A silhouette plot, as shown in Figure 7, shows for each cluster how the well the data points in that cluster fit to the cluster as compared to the other clusters. This is quantified by a measure between 1 and -1 signifying whether the average distance of a data point to the other data points in the same cluster is larger than the smallest distance to a data point in any of the other clusters. If the measure is 1, the data point is well represented by its cluster, if it is 0, it lies between two clusters, and if it is -1, it would better be represented by a different cluster.

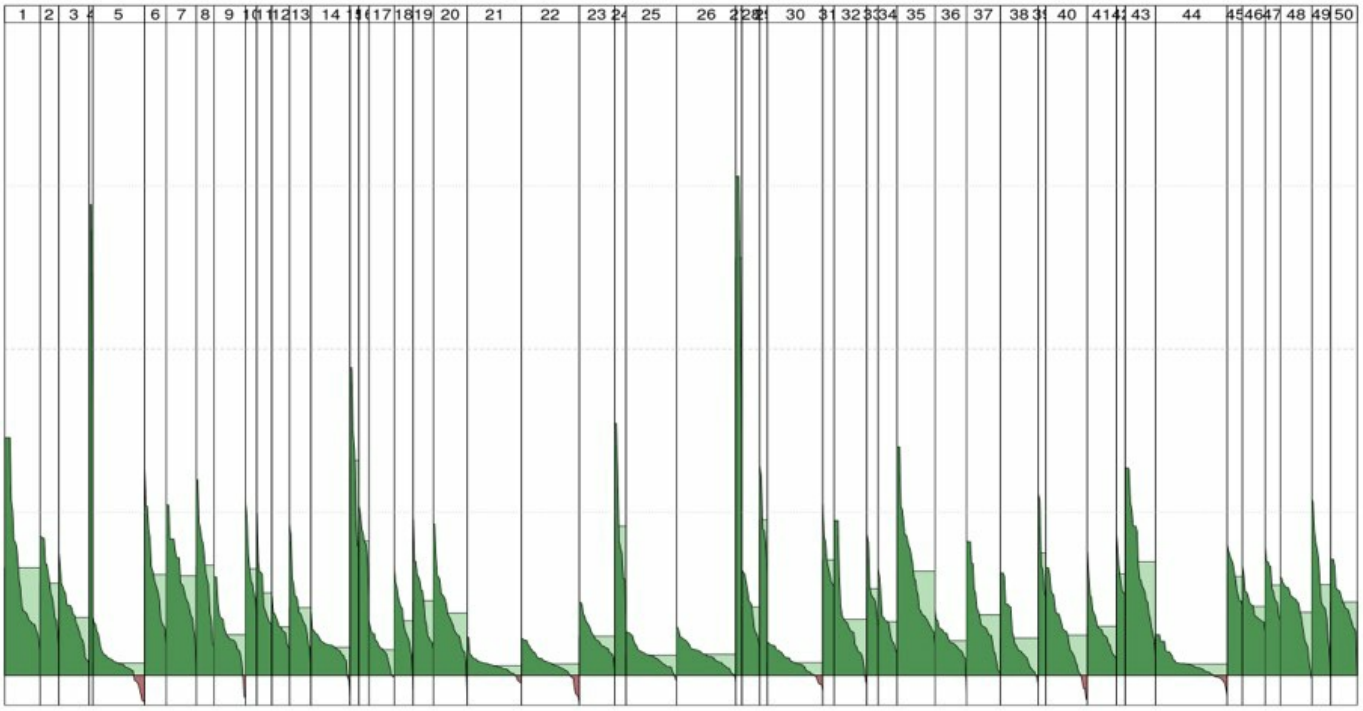


Figure 7: A silhouette plot for assessing the quality of a clustering result.

A histogram of cluster sizes – see Figure 8 – shows that each cluster seems to have a reasonable number of data points. If there were many clusters with very few data points, this would be an indication of either too many clusters or of the inability of the distance measure to reflect the differences in the data.

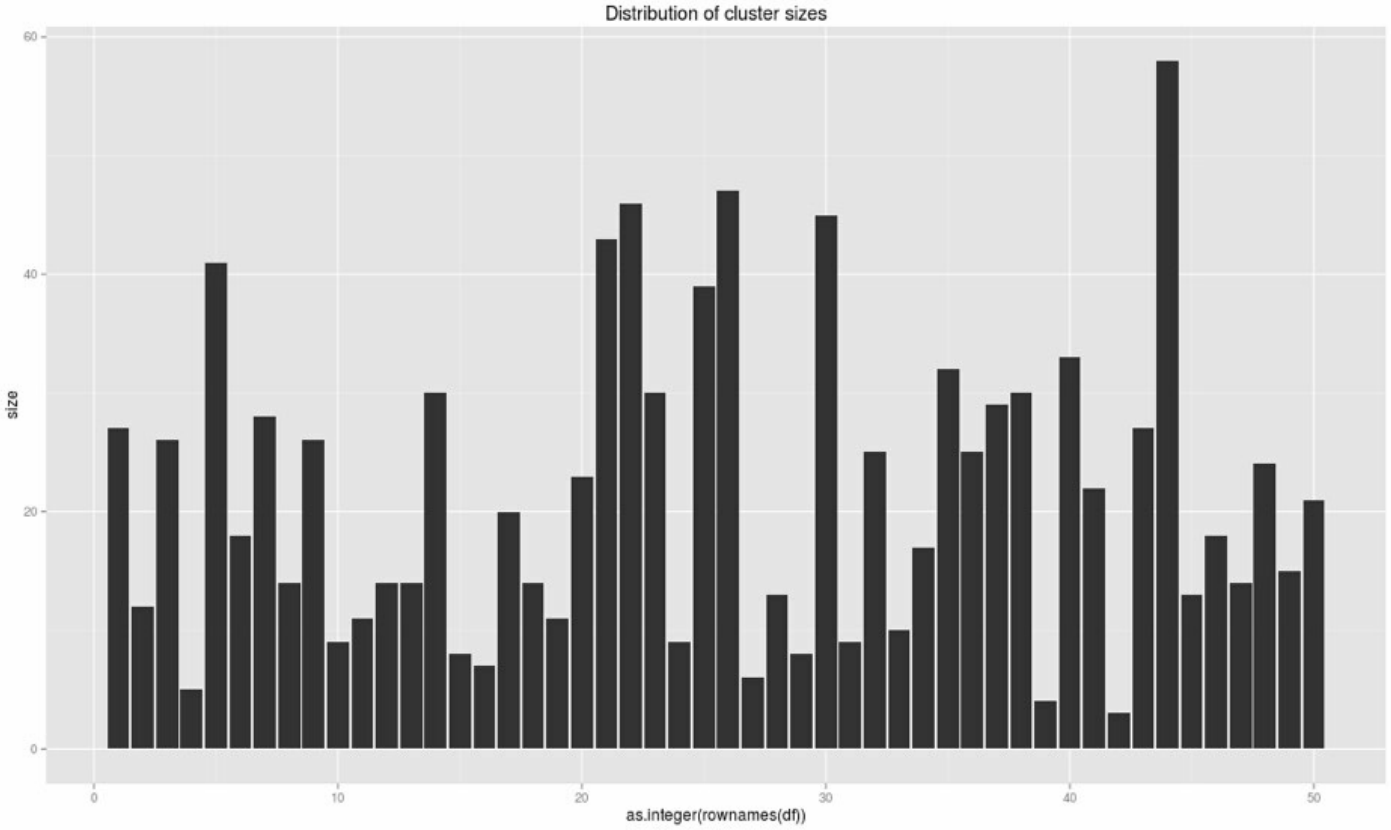


Figure 8: Distribution of cluster sizes.

In principle, machine learning models can be described and exchanged in the Predictive Model Markup Language (PMML). This is supported both in R and in Cascading/pattern. Even though this is a good approach, it is not yet fully implemented (in particular not for the Jaccard distance).

We therefore encapsulate the model in a Java class called ClusterModelFactory that creates a ClusterModel object and provides the following methods:

```
Set<String> getKeys();
```

This returns the set of cluster ids in the model.

```
String getValue(String clusterid);
```

This returns the string of zeros and ones representing the user set of the metroid of the cluster with the given cluster id.

Applying the cluster model

Now, each point in the cluster corresponds to an artist. The cluster model provides the medoids of the clusters where each medoid represents a user set of users listening to this artist. We already have a Cascading workflow that computes the user sets for all artists from the raw data. A second workflow now just determines for each artist, which of the cluster medoids is closest to it.

In order to apply the cluster model, we need a method to compute the Jaccard distance:

```
public class JaccardSet implements Serializable {

private final SortedSet<String> vals = new TreeSet<>();

public void add(String name) {this.vals.add(name);}

public double distanceTo(JaccardSet other) {
```

```
        TreeSet<String> n = new TreeSet<>(vals);
              n.retainAll(other.vals);

        TreeSet<String> d = new TreeSet<>(vals);
              d.addAll(other.vals);

        if (d.size() == 0)
              {return 0.0;}
        else
              {return 1 - n.size() * 1.0 / d.size();

    }
}
```

Based on this method, we can apply the cluster model to each artist in the data set in two steps:

Step 1: We apply a function findClosestCluster which finds the medoid with the smallest Jaccard distance to a given user set by applying an Each pipe to the user sets.

```
listenEvts = new Each(listenEvts, new Fields("userset"), new getClosestClusterFunction(new
Fields("clusterid")), new Fields("artist_name","clusterid"));
```

Step 2: Now a new pipe for the listening events is split to aggregate the artist names for the same cluster using the GroupBy and Every pipes.

```
Pipe clusters = new Pipe("cllist", listenEvts);

clusters = new GroupBy(clusters, new Fields("clusterid"));

clusters = new Every(clusters, new JaccardSetAggregator(new Fields("artistset"), "artist_name"),
new Fields("clusterid","artistset"));
```

The JaccardSetAggregator adds all the artist names it receives to a Java set object.

Putting this together gives us a Cascading workflow to find the cluster for each artist.

Validation

Now that we have applied the cluster model which has been built on a sample of the data set to the whole data set, we want to validate that the distribution of data to the clusters is similar for the sampled data set and the whole data set.

To do this, we compute cluster sizes for clusters built on the sample using R and we compute the cluster sizes for clusters resulting from the previous step – applying the cluster model to the whole data set – using on top of Lingual.

Here is the R code to produce the plots used to compare distributions:

```
58 drv <- JDBC("cascading.lingual.jdbc.Driver", cp)
59 connection <- dbConnect(drv, "jdbc:lingual:hadoop;")
60
61 # query the repository
62
63 postAssing <- dbGetQuery(connection, "SELECT cluster_id, count(DISTINCT artist_name) COUNT_DIST
64                 FROM last_fm.cluster_assignment GROUP BY cluster_id ORDER BY cluster_id")
65 head(postAssing)
66 a <- subset(postAssing, postAssing$CLUSTER_ID >0)
67
68 m <- ggplot(postAssing, aes(x=as.integer(postAssing$CLUSTER_ID), y=COUNT_DIST) )
69 m <- m + ggtitle("Distribution of cluster sizes")
70 m + geom_bar( stat="identity")
```

Figure 9 and Figure 10 show these distributions for the data sample and for the whole data set, respectively.

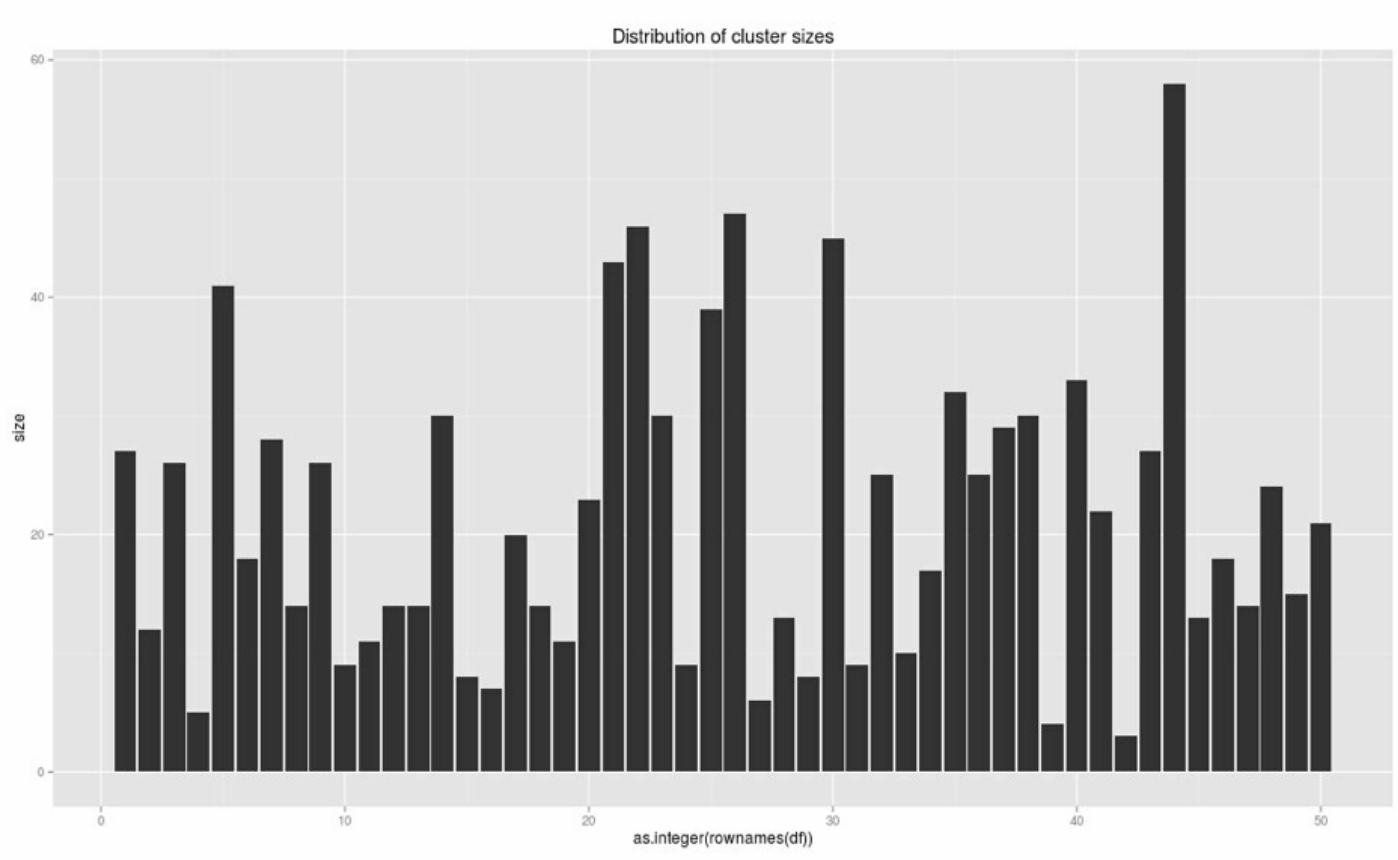


Figure 9: Cluster distribution of data sample.

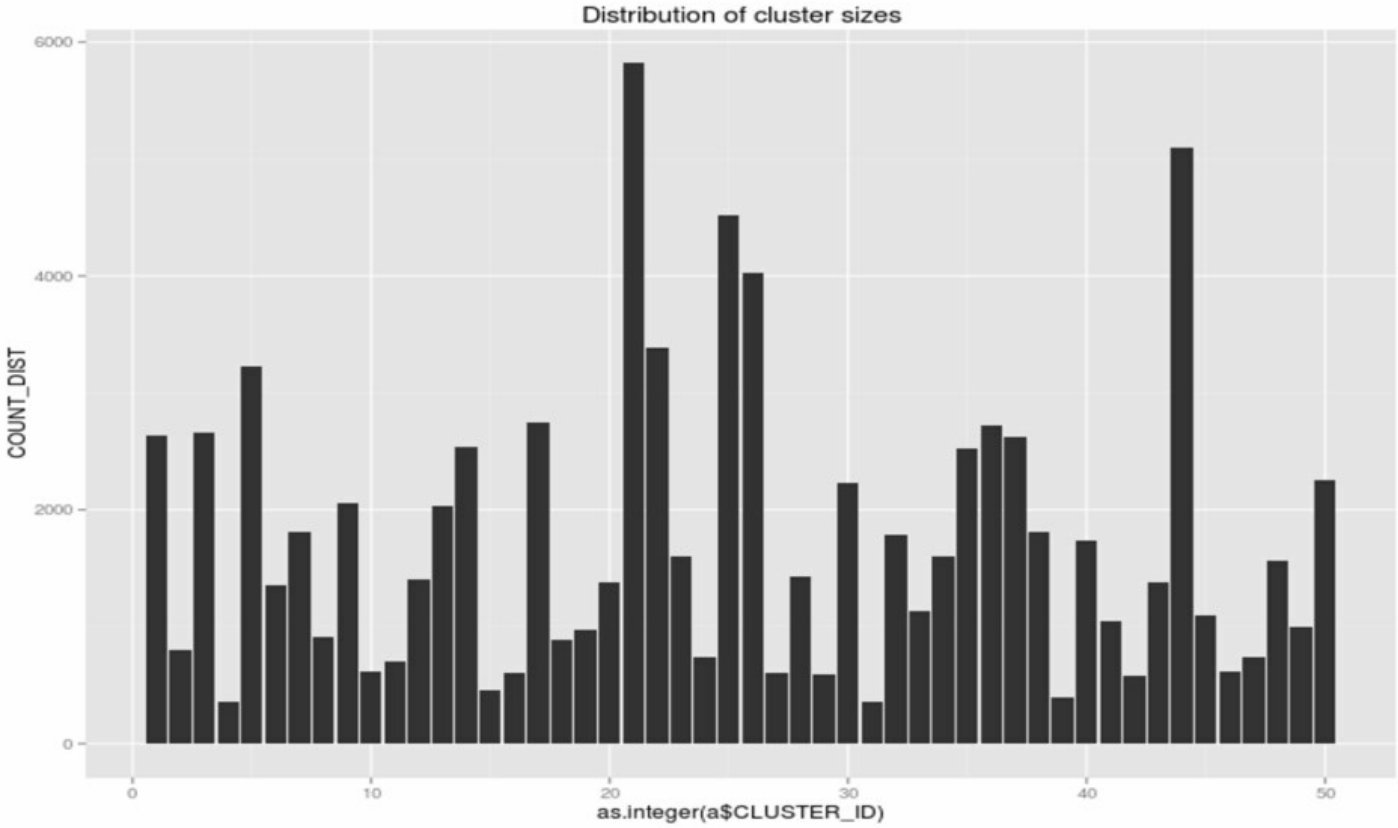


Figure 10: Cluster distribution of whole dataset.

Even though there are some differences, the two histograms look sufficiently similar to assume that the cluster model built on the sample generalises to the whole data set.

Finally, let us use R again to compute the correlation between these two distributions:

```
# compute correlation
pre <- df['size']
# ...
post <- transform(a, CLUSTER_ID = as.numeric(CLUSTER_ID))
post <- post[with(post, order(CLUSTER_ID)), ]
post <-subset(post, post$CLUSTER_ID >0)
cor(pre,post[,2])
```

For our sample dataset, we get a correlation coefficient of 0.8678174. For the small excerpt of the last.fm data that we are using this is quite reasonable and we can move on to thinking about adding a real-time layer to our lambda architecture which handles recommendations for new artists which have not yet been seen in the batch run.

Generating Real-Time Recommendations

Hadoop stores and processes large volumes of data in the Hadoop File System (HDFS). Consequently, Hadoop jobs transform HDFS input data into HDFS output data. This produces an overhead for distributing and monitoring jobs which only pays off for large input size. All data must be available in HDFS before a job starts. Loading data into HDFS requires a significant amount of time and Hadoop jobs can run for days. This batch style of processing is not suited for processing data as soon as it arrives.

In the last.fm recommendations example, Hadoop generates a clustering of artists based on the similarity of user sets. For new artists who are not present in the clustering model, no recommendation is possible. However, new artists are listened to by already existing users. Therefore, the user sets of new artists can be used to compute their distance to the clusters provided by the batch process. Thus, in order to generate real-time recommendations based on new artists, the task is to compute the related user sets in-stream.

In-stream computing comes with a number of challenges. In order to process massive amounts of distributed events at high speed, algorithms should have a small memory footprint (aggregations only in memory) and they should be very fast in the sense that they are only allowed to perform simple computations in constant time per event.

A very useful set of techniques to cope with these challenges are massive stream synopses. A good source for learning more about stream synopses is: Cormode, Garafolakis, Haas, Germaine: Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches, 2012. These comprise probabilistic data structures and techniques for stream aggregation. The basic idea is to aggregate streams in such a way as to provide answers to predefined queries. For example, the following techniques are available:

Frequency counts: AMS-Sketches, FM-Sketches, CountMin Sketches,

Count Distinct, Set Membership: Linear Counting, Bloom-Filters,

Aggregates: Wavelets, exponential Histograms

Linear Counting can be seen as a Bloom-Filter with just one hash-function. It provides a compact synopsis for the set of distinct entities in a stream. In the following, we will use this so-called CountDistinctSketch to provide fast estimates for the Jaccard distance.

CountDistinctSketch is a set synopsis that can be used to estimate the number of distinct elements of a set. Primitives on CountDistinctSketch provide estimates for the number of distinct elements in unions and intersections of sets as well (see Kamp, Kopp, Mock, Boley, May: Privacy-Preserving Mobility Monitoring using Sketches of Stationary Sensor Readings, ECML 2013).

The CountDistinctSketch works as follows. First, each element of a set is hashed into an array of size m. Each cell of the array is represented by a single bit which is set to one if at least one element was hashed to that cell and to zero otherwise. Now, the number n of distinct elements in the original set can be estimated as $\hat{n}$ based on the number U_n of zeros in the array as follows (see: Whang, Vander-Zanden, Taylor: A linear-time probabilistic counting algorithm for database applications, ACM TODS, 1990):

$$\hat{n} = -m \ln(U_n/m)$$

An example for estimating the number of unique users for a given artist based on listening events is given in Figure 11.

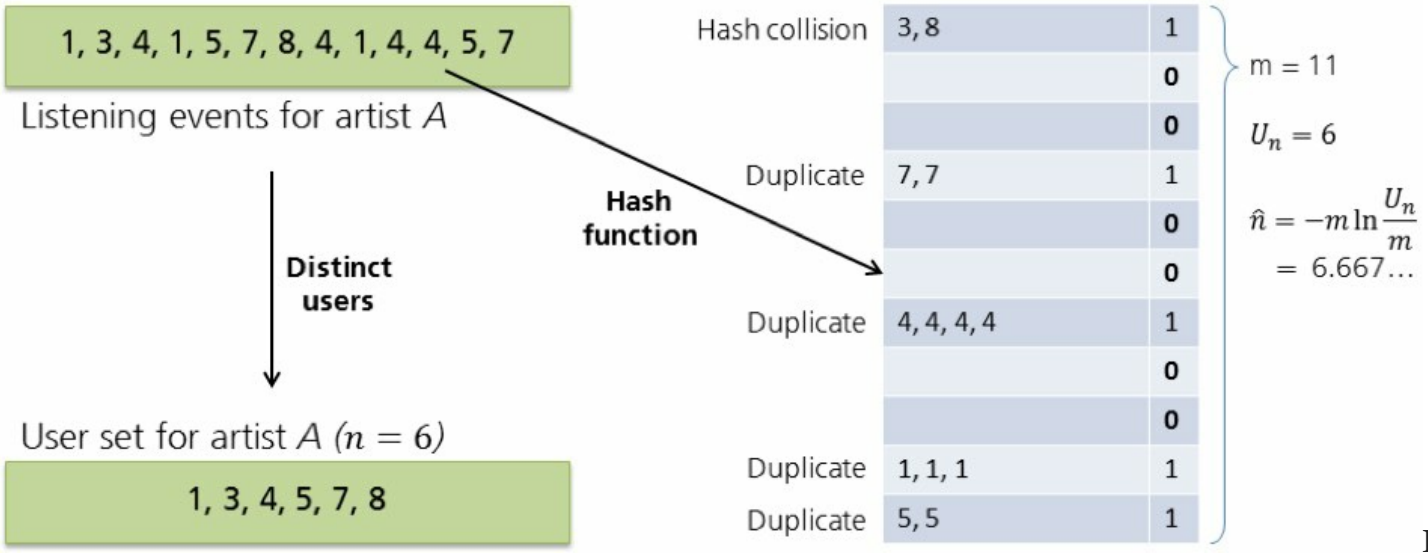


Figure 11: Example for the CountDistinctSketch.

It is possible to compute the expected value of $\hat{n}$ in terms of the ratio n/m and parameters controlling the expected load factor of the array, i.e., how many non-zero entries there are. Thus, a balance can be found between the expected error in estimating n and the size of the sketch.

If we represent two sets A_1 and A_2 by CountDistinctSketches, we can also get an estimate of the number of distinct elements in the union of these sets. To this end, we compute the bit-wise disjunction *CountDistinctSketch(A_1) V CountDistinctSketch(A_2)* of the two sketches and use the number of zeros in the result to estimate the number of elements as given above.

To estimate the Jaccard distance based on this type of sketch, define:

CountDistinctSketch(Artistname) = Sketch of user names with listening events for Artistname

| CountDistinctSketch(Artistname) | = Estimate of the size of the set represented by the sketch.

Recall that the Jaccard similarity J of two artists A_1 and A_2 is given by:

$$J(A_1, A_2) := \frac{|A_1 \cap A_2|}{|A_1 \cup A_2|}$$

We can deduce the size of the intersection of A_1 and A_2 as the sum of the sizes of A_1 and A_2 minus the size of the union of A_1 and A_2.

The Jaccard similarity can now be computed in stream by making use of sketches as follows:

$$J(A_1, A_2) = (|CountDistinctSketch(A_1)| + |CountDistinctSketch(A_2)| - |CountDistinctSketch(A_1) \vee CountDistinct\ Sketch(A_2)|) / |CountDistinctSketch(A_1) \vee CountDistinctSketch(A_2)|$$

Now that we can compute the Jaccard distance in stream and thus get the user sets for new artists, we can compare these to the medoids of the clusters we have got from batch processing. This will give us the cluster comprising recommendations related to a new artist.

This can now be put into the speed layer of a lambda architecture, see Figure 12. The in-stream processing can be implemented using Apache Storm as described in the module Big Data Architecture.

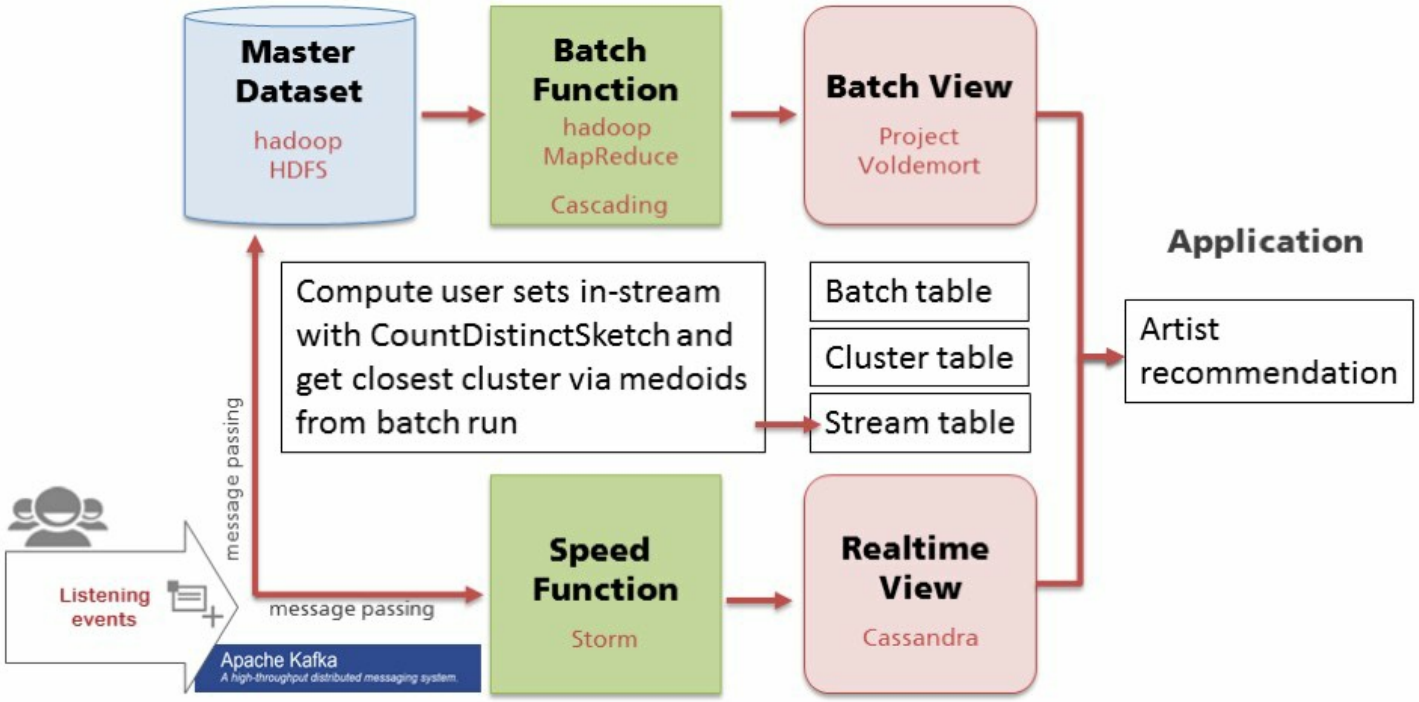


Figure 12: Putting in-stream processing into the lambda architecture.

Big Data Analytics In Spark

Apache Spark is one of the fastest developing platforms for big data analytics. It provides means to overcome the disc I/O overhead often seen in MapReduce based processing. Concepts like data frames and Spark SQL make it convenient to work with structured data inside Spark programs.

On the one hand, even though Hadoop is designed to handle large volumes of data and to be fault tolerant by fall-back to intermediate results, any iterative processing is slowed down by the resulting disk reads and writes. Therefore it is not well suited for machine learning.

On the other hand, stream processing provides fast processing of data, but no storage of intermediate results so that machine or network failures cause loss of data. This allows only for online learning algorithms and not for iterative algorithms which are a mainstay of machine learning.

The central concept in Spark is the resilient distributed dataset (RDD). An RDD manages a collection of elements which can be operated on in parallel. They are implemented in such a way as to make them fault tolerant. Spark can make use of several basic technologies to support this. For example, RDDs can be created by referencing datasets in the Hadoop File System or a big data database such as HBase.

RDDs offer a number of transformations that work on a given RDD in read-only mode (RDDs are immutable) and transform it into a new RDD. These transformations are not executed immediately but only represented in the data structure for lazy evaluation. These transformations comprise such functions as map and reduceByKey which are known from the MapReduce framework. There are several more transformations including filter, flatMap, sample, groupByKey, mapValues, sort, and partitionBy. The function of some of these can be deduced from their names. We will see a selection of important transformations in the following sections.

Evaluation of transformations takes place once a so-called action is performed on an RDD. These use an RDD as input and produce a value as an output. This value can of different types or even a sequence of values but it is no longer an RDD. Examples of actions comprise:

count: Return the number of rows in an RDD.

first: Return the first row of an RDD.

take(n): Return the first n rows of an RDD.

collect: Return all rows of an RDD as a sequence of rows.

reduce: Aggregation of elements in an RDD based on a user-provided aggregation function.

save: Store an RDD in a file.

Spark is written in Scala and can therefore be used comfortably in Scala. Additionally, it provides APIs in Java, Python, and R. In the following, we will show how to use Spark based on the Python API.

Machine learning in Spark

In addition to the basic functionality of RDDs, Spark offers a number of libraries built on top of this. Spark Streaming facilitates building streaming applications like the ones we have discussed in this module (generating real-time recommendations) and seen realised in Apache Storm in the module Big Data Architecture. In the previous section, we have already seen how Spark SQL provides data frames and SQL capabilities that make working with data quite elegant. Additionally, the GraphX library provides a way to model graph structures using RDDs and some basic functionality for working with graphs. However, at the time of writing this, the GraphX library did not yet offer a wide range of functionality.

In addition to these libraries, Spark provides the Spark MLlib, a library that contains many distributed machine

learning approaches suitable for big data analytics. The following list gives an overview of the contents of this library:

Basic statistics (correlations, stratified sampling, hypothesis testing, ...)

Classification and regression (linear regression, SVMs, decision trees, ...)

Collaborative filtering (alternating least squares)

Clustering (k-means, Gaussian mixture, power iteration clustering, ...)

Dimensionality reduction (singular value decomposition, PCA)

Feature extraction and transformation (TF-IDF, feature selection, ...)

Frequent pattern mining (FP-growth, PrefixSpan)

Evaluation metrics (precision, recall, F-measure, ROC, ...)

PMML model export

Optimization (stochastic gradient descent)

We will see the methods printed in bold face in action in the following sections.

Spark and Python

PySpark allows the powerful combination of Spark with Python and thus to combine distributed processing and the wide range of libraries for data analytics and visualisation in Python. It can either be used in an interactive shell coming with the Spark distribution or in stand-alone Python. The interactive shell can be started from the Spark directory as follows:

```
IPYTHON=1 ./bin/pyspark
```

In this way, it uses IPython as the interactive shell, without the IPYTHON variable, its uses a more basic shell.

When run from the PySpark shell, the API is already available as a so-called Spark context. If used in a Python script, this needs to be initialised first as follows:

```
from pyspark import SparkContext

sc = SparkContext("local", "App Name", pyFiles=['MyFile.py', 'lib.zip', 'app.egg'])
```

The optional pyFiles parameter gives a list of files which will be added to the Python path and also be shipped to all machines in the cluster running Spark.

As a first step, let us create an RDD from the last.fm dataset using the Spark context and split its tab separated lines into tuples:

```
words = sc.textFile("file:///home/livlab/data/last-fm.tsv")
words.take(5)

words2 = words.map(lambda line: line.split("\t"))
words2.take(5)
```

Now that we have written our first map transformation which takes each line in an RDD (words) and produces a new line in a new RDD (words2) by applying a function to each line. Using the functional programming style of lambda expressions makes it very easy to define the function to be applied.

DataFrames in Spark

DataFrames in Spark are an abstraction on top of RDDs. They interpret RDDs as sequences of rows and wrap

them using schemas, i.e., rows have columns with names and types. With DataFrames it is possible to execute SQL queries and transformations, read and write data in Hadoop and Hive formats (ORC, Parquet), and to connect to standard databases using JDBC.

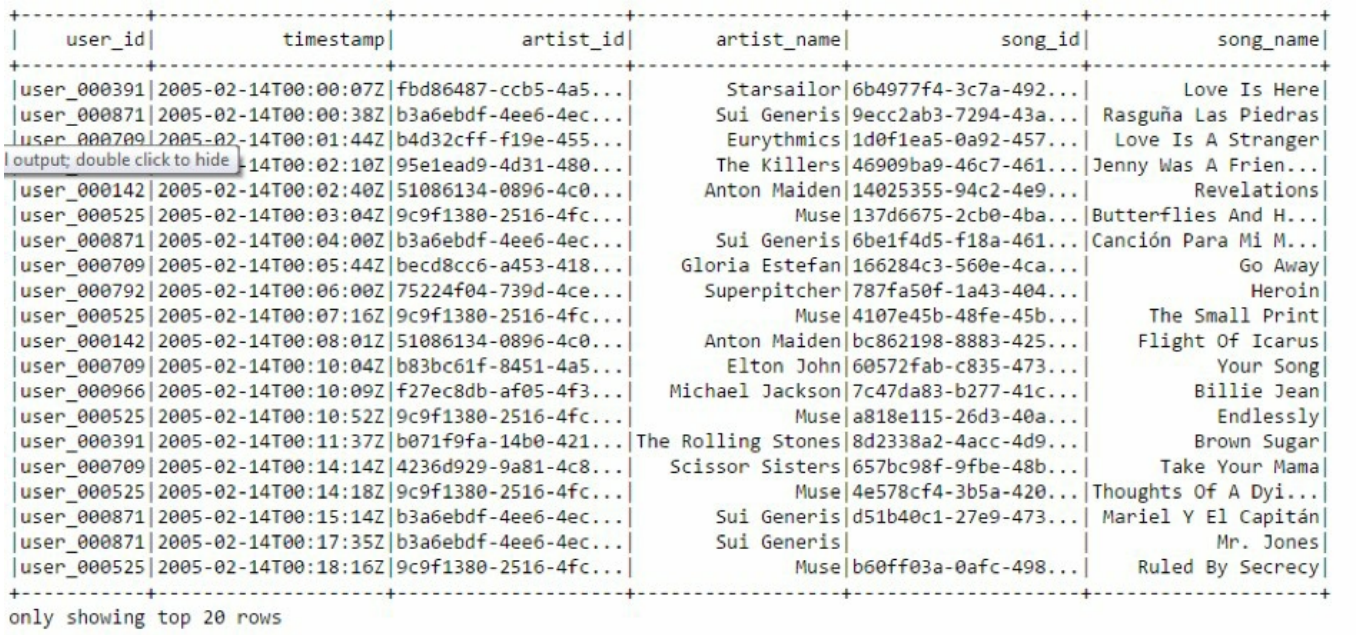
To get a basic impression of this, let us convert our second RDD above into a DataFrame:

```
words_df = words2.toDF(["user_id", "timestamp", "artist_id", "artist_name", "song_id", "song_name"])
```

The list given as parameter gives names to each of the columns in the DataFrame. Spark can give a concise overview of the contents of the DataFrame:

```
words_df.show()
```

This results in the following output:



```
+-----------+--------------------+--------------------+-----------------+--------------------+--------------------+
|    user_id|           timestamp|           artist_id|      artist_name|             song_id|           song_name|
+-----------+--------------------+--------------------+-----------------+--------------------+--------------------+
|user_000391|2005-02-14T00:00:07Z|fbd86487-ccb5-4a5...|        Starsailor|6b4977f4-3c7a-492...|        Love Is Here|
|user_000871|2005-02-14T00:00:38Z|b3a6ebdf-4ee6-4ec...|       Sui Generis|9ecc2ab3-7294-43a...| Rasguña Las Piedras|
|user_000709|2005-02-14T00:01:44Z|b4d32cff-f19e-455...|       Eurythmics|1d0f1ea5-0a92-457...|  Love Is A Stranger|
|user_000709|2005-02-14T00:02:10Z|95e1ead9-4d31-480...|       The Killers|46909ba9-46c7-461...|Jenny Was A Frien...|
|user_000142|2005-02-14T00:02:40Z|51086134-0896-4c0...|      Anton Maiden|14025355-94c2-4e9...|         Revelations|
|user_000525|2005-02-14T00:03:04Z|9c9f1380-2516-4fc...|             Muse|137d6675-2cb0-4ba...|Butterflies And H...|
|user_000871|2005-02-14T00:04:00Z|b3a6ebdf-4ee6-4ec...|       Sui Generis|6be1f4d5-f18a-461...|Canción Para Mi M...|
|user_000709|2005-02-14T00:05:44Z|becd8cc6-a453-418...|    Gloria Estefan|166284c3-560e-4ca...|             Go Away|
|user_000792|2005-02-14T00:06:00Z|75224f04-739d-4ce...|      Superpitcher|787fa50f-1a43-404...|              Heroin|
|user_000525|2005-02-14T00:07:16Z|9c9f1380-2516-4fc...|             Muse|4107e45b-48fe-45b...|     The Small Print|
|user_000142|2005-02-14T00:08:01Z|51086134-0896-4c0...|      Anton Maiden|bc862198-8883-425...|    Flight Of Icarus|
|user_000709|2005-02-14T00:10:04Z|b83bc61f-8451-4a5...|       Elton John|60572fab-c835-473...|           Your Song|
|user_000966|2005-02-14T00:10:09Z|f27ec8db-af05-4f3...|   Michael Jackson|7c47da83-b277-41c...|         Billie Jean|
|user_000525|2005-02-14T00:10:52Z|9c9f1380-2516-4fc...|             Muse|a818e115-26d3-40a...|           Endlessly|
|user_000391|2005-02-14T00:11:37Z|b071f9fa-14b0-421...|The Rolling Stones|8d2338a2-4acc-4d9...|         Brown Sugar|
|user_000709|2005-02-14T00:14:14Z|4236d929-9a81-4c8...|   Scissor Sisters|657bc98f-9fbe-48b...|      Take Your Mama|
|user_000525|2005-02-14T00:14:18Z|9c9f1380-2516-4fc...|             Muse|4e578cf4-3b5a-420...|Thoughts Of A Dyi...|
|user_000871|2005-02-14T00:15:14Z|b3a6ebdf-4ee6-4ec...|       Sui Generis|d51b40c1-27e9-473...| Mariel Y El Capitán|
|user_000871|2005-02-14T00:17:35Z|b3a6ebdf-4ee6-4ec...|       Sui Generis|                    |           Mr. Jones|
|user_000525|2005-02-14T00:18:16Z|9c9f1380-2516-4fc...|             Muse|b60ff03a-0afc-498...|    Ruled By Secrecy|
+-----------+--------------------+--------------------+-----------------+--------------------+--------------------+
only showing top 20 rows
```

To run SQL queries on the DataFrame, we need an analogue of the SparkContext, called the SQLContext. It is automatically included if the PySpark shell is used. Otherwise, it can be created as follows:

```
from pyspark import SQLContext

sqlContext = SQLContext(sc)
```

With the SQLContext at hand, here is an example how to run SQL queries on DataFrames in Spark:

```
words_df.registerTempTable("lastfm")

sqlContext.sql("select * from lastfm where artist_name like 'E%'").show()
```

The first line registers the DataFrame as a table together with a name that can be used as a table name in SQL queries. The second line runs an SQL query for all artists with a name starting with an E. The result looks like this:

```
+----------+--------------------+--------------------+--------------------+--------------------+--------------------+
|   user_id|           timestamp|           artist_id|         artist_name|             song_id|           song_name|
+----------+--------------------+--------------------+--------------------+--------------------+--------------------+
|user_000709|2005-02-14T00:01:44Z|b4d32cff-f19e-455...|          Eurythmics|1d0f1ea5-0a92-457...|  Love Is A Stranger|
|user_000709|2005-02-14T00:10:04Z|b83bc61f-8451-4a5...|          Elton John|60572fab-c835-473...|           Your Song|
|user_000709|2005-02-14T00:23:14Z|4967c0a1-b9f3-465...|                Enya|3ce14282-0188-470...|         China Roses|
|user_000709|2005-02-14T00:53:14Z|5bc3a5be-a7a7-4a8...|                Eko|de4f1a92-f7a4-4ca...|          Relativity|
|user_000391|2005-02-14T01:07:20Z|14387b0f-765c-485...|                Eels|453e9897-3507-40e...|  Somebody Loves You|
|user_000961|2005-02-14T01:50:07Z|430f3c84-b455-434...|Elvis Costello Wi...|022b6980-f34a-40e...|God Give Me Strength|
|user_000709|2005-02-14T01:52:56Z|b4d32cff-f19e-455...|          Eurythmics|dd729d6f-a0d0-423...|There Must Be An ...|
|user_000709|2005-02-14T02:01:56Z|b4d32cff-f19e-455...|          Eurythmics|59f24a74-aaa6-48b...|     DonT Ask Me Why|
|user_000709|2005-02-14T02:09:18Z|4967c0a1-b9f3-465...|                Enya|4da47913-7f98-45b...|       The Longships|
|user_000709|2005-02-14T03:09:38Z|b83bc61f-8451-4a5...|          Elton John|e3710ecc-2578-4a0...|DonT Let The Sun ...|
|user_000961|2005-02-14T03:14:23Z|430f3c84-b455-434...|Elvis Costello Wi...|6f5ad926-8851-4b8...|            My Thief|
|user_000709|2005-02-14T03:43:55Z|4967c0a1-b9f3-465...|                Enya|a090633b-2406-489...|    Hope Has A Place|
|user_000709|2005-02-14T03:57:55Z|8d223216-c738-484...|           EveS Plum|ce227b08-410a-405...|       Save A Prayer|
|user_000709|2005-02-14T04:39:02Z|5bc3a5be-a7a7-4a8...|                Eko|97af25c4-0938-4ef...|Morning In Martin...|
|user_000709|2005-02-14T04:59:53Z|4967c0a1-b9f3-465...|                Enya|b1e1f99d-be18-4e6...|       Shepherd Moons|
|user_000871|2005-02-14T07:15:51Z|f4a31f0a-51dd-4fa...|         Evanescence|16acf955-ae50-44c...|    Bring Me To Life|
|user_000709|2005-02-14T07:22:17Z|43b58c98-3779-4b0...|             Erasure|342da90a-8155-485...|     Save Me Darling|
|user_000871|2005-02-14T08:33:42Z|aad38e08-baeb-49f...|    Evan Rachel Wood|                    |    Evan Rachel Wood|
|user_000961|2005-02-14T08:34:44Z|85df390a-5784-494...|          Elie Karam|93d32440-8a58-414...|             Baadima|
|user_000709|2005-02-14T08:41:47Z|b83bc61f-8451-4a5...|          Elton John|64cc71ff-f7f8-418...|Something About T...|
+----------+--------------------+--------------------+--------------------+--------------------+--------------------+
only showing top 20 rows
```

DataFrames can also be saved in and imported from standard database formats such as Apache Parquet.

With this basic knowledge at hand, we can start into our data analytics examples using Spark.

Linear regression

Let us start our journey into analytics with Spark with a simple example: linear regression. The aim of linear regression is to explain a dependent variable y as linear function of an independent variable x:

$$y = mx + b$$

In our case, we will try to predict the total number of listening events on last.fm for a certain time. Therefore we choose time as our independent variable x and the number of clicks at a given time as the dependent variable y.

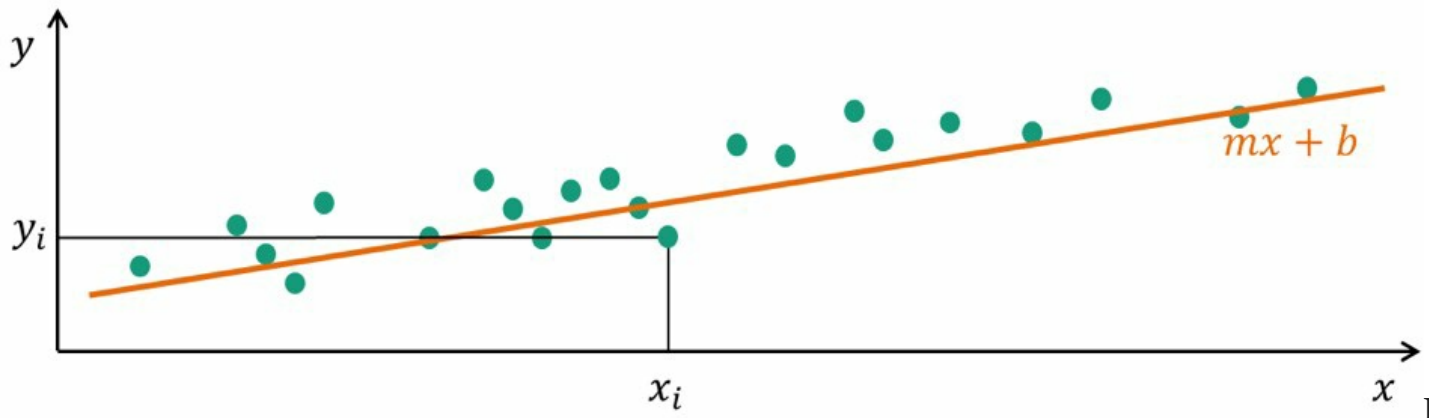


Figure 13: The setup of linear regression: modelling a linear relationship with slope m and intercept b between the variables x and y.

The regression line is defined as the minimizer in terms of m and b of a quadratic error function E defined on a set of data points (x_i, y_i) from which a linear model is to be inferred:

$$E := \sum_i \left(y_i - (mx_i + b)\right)^2$$

Finding m and b based on this setup is a standard tool in pretty much any analytics toolbox. The question for large data sets is how to parallelise this. A popular technique which is also implemented in Spark is stochastic gradient descent. This is a version of gradient descent, where the contribution of each data point to the gradient of the error function E is computed in parallel and each leads to a gradient descent step. These can either be applied independently or after aggregation.

For linear regression we first restrict the error function to single data point:

$$E_i = (y_i - (mx_i + b))^2$$

Then we can iteratively update slope and intercept of the regression model by the following algorithm.

Choose initial parameters m and b (and learning rate α).

Repeat until stopping criterion reached:

Compute $\frac{\partial E_i}{\partial m}$ and $\frac{\partial E_i}{\partial b}$ in parallel

Randomly shuffle data points (x_i, y_i)

For each data point (x_i, y_i)

$$m \leftarrow m - \alpha \cdot \frac{\partial E_i}{\partial m}(m, b)$$

$$b \leftarrow b - \alpha \cdot \frac{\partial E_i}{\partial b}(m, b)$$

Stochastic gradient descent makes use of a learning rate to regulate the size of each update step, i.e., to regulate how strongly the partial derivatives $\frac{\partial E_i}{\partial m}$ and $\frac{\partial E_i}{\partial b}$ influence the update of the parameters m and b.

Actually running this in Spark amounts to a single line of code. However, our data first needs to be set up in the format expected by the Spark function.

First, we will need to import the data type Spark is using for linear regression. This is called LabeledPoint. A LabeledPoint represents a pair of data whose first element is a value or label and corresponds to y-values in our example. The second element is a list of values corresponding to our x-values. In the example, we are using one-dimensional regression and therefore each x-value will be represented by a list of length one. In multidimensional regression, this would be a list with as many entries as dimensions. We will also import a parser from Python's dateutil library in order to work more conveniently with the time stamps of listening events which are given as strings in the data set.

```
from pyspark.mllib.regression import LabeledPoint

from dateutil import parser
```

In the following lines of code, we can see how to parse the dates given as strings in the last.fm dataset into more handy Python objects representing date and time. This is done by a map transformation. We will also see how to use the filter transformation to restrict the listening events to those from the year 2005. We use a very simple reduction of the date to a number which more or less corresponds to the number of the day within the year. It is normalised such that our data starts at day zero to make the results of linear regression easier to interpret. The reduceByKey transformation is used to count the number of listening events per day. It applies an aggregation function to the data for each key. Spark simply assumes the first element in each tuple to be the key. The transformation defines how to combine intermediate results based on the non-key elements of the tuples. Finally, we transform each pair (day, number of listening events) to Spark's LabeledPoint data type.

```
# 1. Create pairs (date, 1):
```

```
times_count = words2.map(lambda x: (parser.parse(x[1]), 1))

# 2. Select listening events from the year 2005:

times_count_2005 = times_count.filter(lambda x: x[0].year==2005)

# 3. Convert day and month to 31*month + day:

days_count = times_count_2005.map( lambda x: (x[0].month*31+x[0].day - 76, x[1]) )

# 4. Count events per day:

count_by_day = days_count.reduceByKey(lambda x,y: x+y)

# 5. Convert to LabeledPoint:

lrinput = count_by_day.map(lambda x: LabeledPoint(x[1], [x[0]]))
```

Now we can run linear regression on this data. Spark provides a function called LinearRegressionWithSGD which can be imported from its regression library:

```
from pyspark.mllib.regression import LinearRegressionWithSGD
```

If you have not guessed so already: SGD is short for stochastic gradient descent. The actual regression model is now trained like this:

```
model = LinearRegressionWithSGD.train(lrinput, 1000,
intercept = True)
```

The parameters are the input data (lrinput), the number of iterations (1000) and whether the model should allow a y-axis intercept different from zero. We would like to allow the latter in our example (intercept = True).

Now we use matplotlib to plot the data and regression line. First, we need to import that library:

```
import matplotlib.pyplot as plt
```

For visualisation purposes, we use a small enough excerpt of the dataset so that the input data for linear regression actually fits into the memory of a single computer. We can get the data into a Python list using the collect action:

```
points = lrinput.collect()
```

This data needs to be split into separate lists for the x and y values for the plotting function:

```
datax = []
datay = []

for p in range(len(points)):
    datax.append(points[p].features[0])
    datay.append(points[p].label)
```

Now everything is prepared for plotting the data and the regression line:

```
myplot=plt.scatter(datax, datay)
xlim = myplot.axes.get_xlim()
plt.plot([xlim[0], xlim[1]], [model.predict([xlim[0]]), model.predict([xlim[1]])], 'r')
```

To draw the regression line, we use the left-most (*xlim[0]*) and right-most (*xlim[1]*) point displayed on the x-axis of the plot and predict their y-values using the predict function of the model.

Figure 14 shows the result on a small sample of the data (using the first 10,000 listening events).

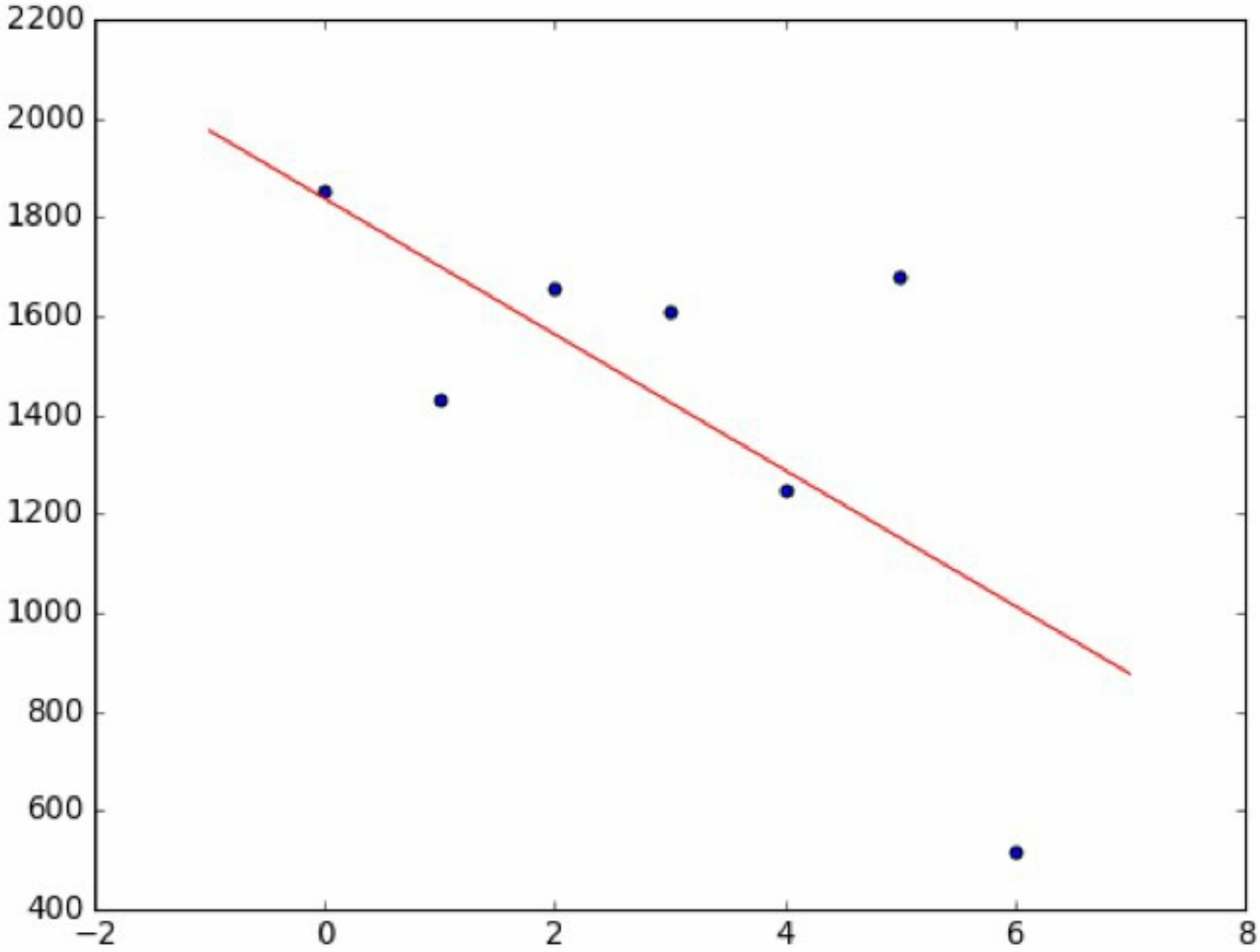


Figure 14: Regression line (red) for the number of listening events (y-axis) per day (x-axis).

Collaborative filtering with alternating least squares

We will now go back to the utility matrix used before and use an alternative collaborative filtering algorithm to produce artist recommendations using Spark. Spark's MLlib comes with a collaborative filtering algorithm based on alternating least squares (ALS). This is an algorithm that was very successful in the Netflix Prize. To employ it, we look at a slightly modified version of the utility matrix from Figure 1. Rather than allowing only zero and one as entries, we will not only incorporate the knowledge whether a user has listened to an artist but also how often. Thus, an entry in the new utility matrix will simply signify how many listening events there are for a given user-artist pair.

We assume that there is a given fixed number of unknown concepts which are the reason why users are listening to specific artists. In the algorithm, this number will be set in advance and is a design parameter. The concepts can be completely abstract and the algorithm does not provide an interpretation. However, in order to have a better idea of how the algorithm works, we will work with specific hidden context that a human can understand. Let us assume these concepts were genres of music and that a given user listens to a certain artist because he likes the genre of music that the artist is playing. However, this information is not given in the dataset. In order to deduce this information, we fix the number of genres we expect and try to find a factorisation of the utility matrix such as in the following example.

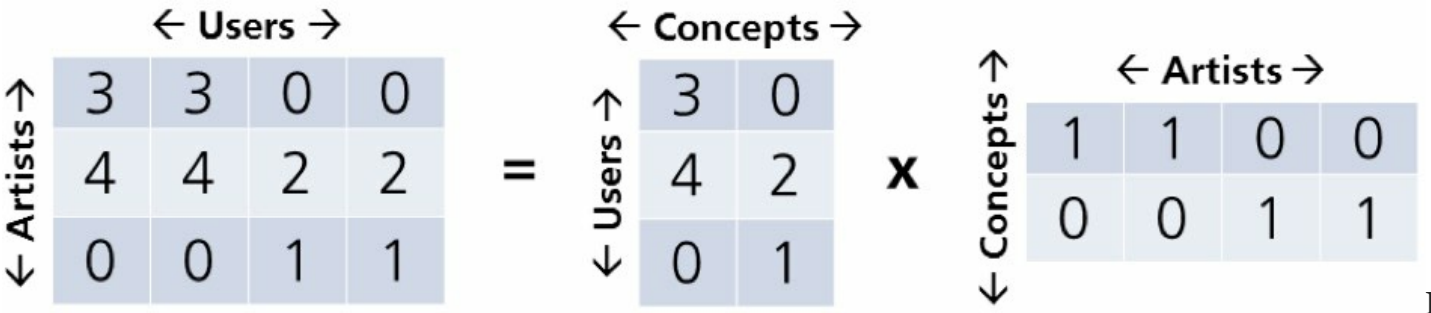


Figure 15: Example of a factorisation of the utility matrix.

In Figure 15, the utility matrix is factored into the product of two matrices: a matrix relating users and concepts and a matrix relating artists and concepts. If we think of the concepts as genres again, then, in this example, we assume that there are two genres and the factorisation shows that two artists belong to the first genre and two other artists to the second genre. It also shows the weights of preference of the users for certain genres. The first user listens only to artists from the first genre, the second user listens twice as often to artists from the first genre than from the second genre, and the third user listens only to artists from the second genre.

The general setup for the collaborative filtering algorithm is as follows. We fix the number of concepts and try to find a factorisation of the form given in Figure 16.

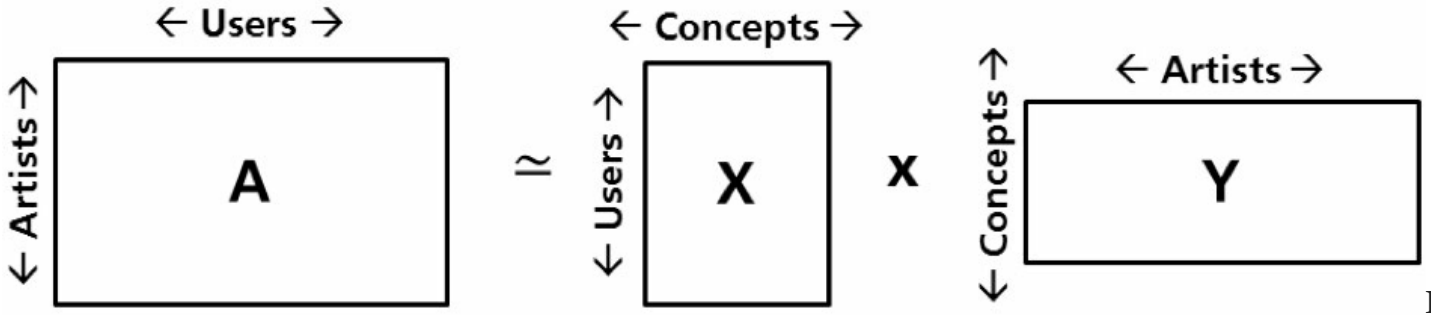


Figure 16: Factorisation of the utility matrix based on hidden concepts.

Among all possible factorisations we look for one making the product XY as similar to A as possible. This is measured in a matrix norm, i.e., we look for matrices X and Y minimising $|A-XY|$. In practice, one might wish to add a regularisation term controlling properties of X and Y such as their sparsity.

This optimisation is very hard to do in both X and Y at the same time. (Indeed this would solve an NP-hard problem). However, it becomes an easy least squares problem when one of the factors, either X or Y is known. Based on this, a solution can be approximated by the following popular iterative procedure:

Initialise X and Y with random values.

Alternately optimise for X and for Y.

This is called the alternating least squares (ALS) algorithm. It can be parallelised efficiently because the optimisation can be performed row-wise (resp. column-wise) when it is run for X (resp. Y). Thus, the degree of parallelisation is governed by the number of users (resp. artists) and the amount of memory necessary in a single processor is limited by the number of concepts.

This is already implemented in Spark and, as before, we need to bring our data into the format expected by Spark:

```
from pyspark.mllib.recommendation import ALS, Rating
import hashlib
```

```
# Prepare pairs: ((user, artist), number of listening events)
ratings = words2.map(lambda x: [(x[0], x[2]), 1])
ratings2 = ratings.reduceByKey(lambda x, y: x+y)

def myhash(inp):
        return int( int(hashlib.md5(inp).hexdigest(), 16) % (10**8) )

alsinput = ratings2.map( lambda x: Rating(  myhash( x[0][0]), myhash(x[0][1]), x[1]  )    )
```

The ALS algorithm in Spark only allows using a restricted number of distinct users and artists. Therefore, we use hashing to map the user and artist ids into a number in the range from 0 to 10^8-1. Similar to the LabeledPoint datatype used in the linear regression example, we need to use the Rating datatype to represent triples of the form (user, artist, number of listening events). Also similar to the linear regression example, we employ the reduceByKey transformation to aggregate the number of listening events per user-artist pair. Note how (user, artists) is grouped to jointly form the key.

Now again when the data is set up, training the model is very easy:

```
# Build the recommendation model using alternating least squares (ALS)

rank = 10               # number of concepts
numIterations = 10      # number of times X and Y are recomputed

model = ALS.train(alsinput, rank, numIterations)

# Get recommendations

myuser = alsinput.first().user
oneuser = alsinput.filter(lambda x: x.user == myuser)
input = oneuser.take(10)

model.predict(input[0].user, input[0].product)
```

The resulting model comes with a predict function which provides the rating for any user-artist pair.

Clustering

We have already seen previously, how clustering can be employed to learn about different associations in the listening events. Figure 17 illustrates two such tasks which can also be answered based on the factored utility matrix rather than the utility matrix itself. Coming back to the genre interpretation of the hidden concepts, it may make more sense to cluster artists or users based on the genres they are associated to than on user or artist sets.

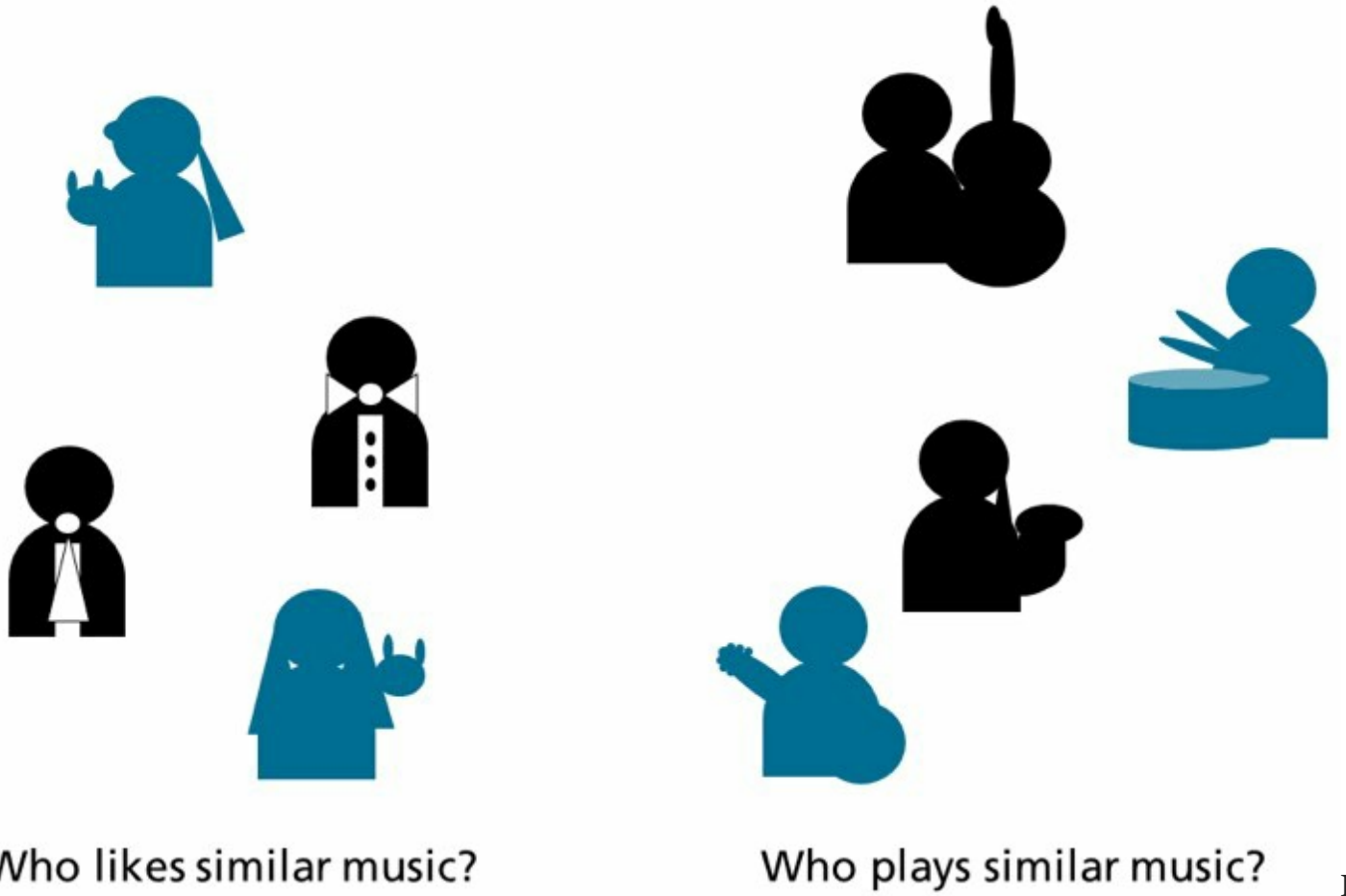


Figure 17: Clustering tasks based on the factors of the utility matrix.

We use this scenario as a motivation to look at a clustering algorithm implemented in Spark: K-means clustering. This is one of the most popular and well-known clustering algorithms. Figure 18 gives an overview of how this algorithm works.

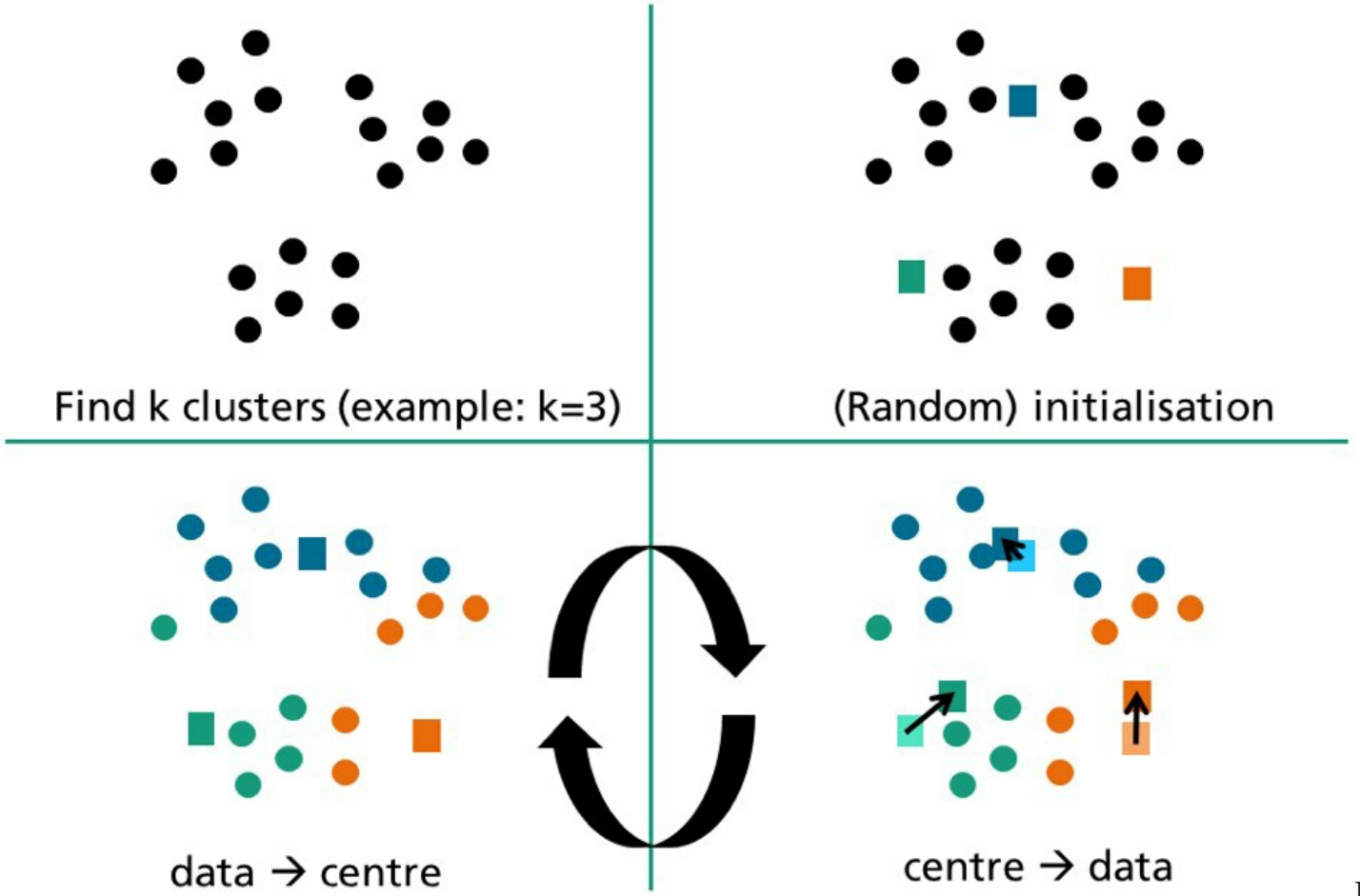


Figure 18: Overview of the k-means clustering algorithm.

In the upper left corner of the figure, you can see the raw data. This is meant to be clustered into a fixed number k of clusters and we will illustrate this with k=3 clusters. The upper right corner of the figure shows the initialisation. Three initial cluster centres are chosen. One way to do this (although not the best one) is to choose the positions of those centres randomly. Then an iterative process starts (lower half of the figure). In one step (lower left corner) each data point is assigned to the cluster represented by the cluster centre to which it has the smallest distance. In another step (lower right corner), the positions of the cluster centres are updated. This is done by computing the mean of the data points that were assigned to that cluster in the previous step. These two steps are repeated until a stopping criterion becomes true. For example, until no data point changes its cluster assignment.

This can be parallelised very well as follows (see Figure 19). First, there is an intelligent parallelised initialisation called k-means|| (k-mean parallel). This is a way to come up with a probability distribution over the data points which allows drawing good candidates for the cluster centres with high probability. The details are beyond the scope of this module. They can be found in the paper cited in Figure 19. After this initialisation, the alternating steps described above can each be parallelised efficiently. In order to assign data points to cluster centres, each processor does this for its share of the data points. This means that the cluster centres have to be distributed to the processors. The number of cluster centres is small so that this overhead is negligible. The cluster centres can first be updated locally on each processor using its share of the data points. Then these local cluster centres together with the information how many data points were used to compute them can be combined to compute the actual cluster centres in a reduce step.

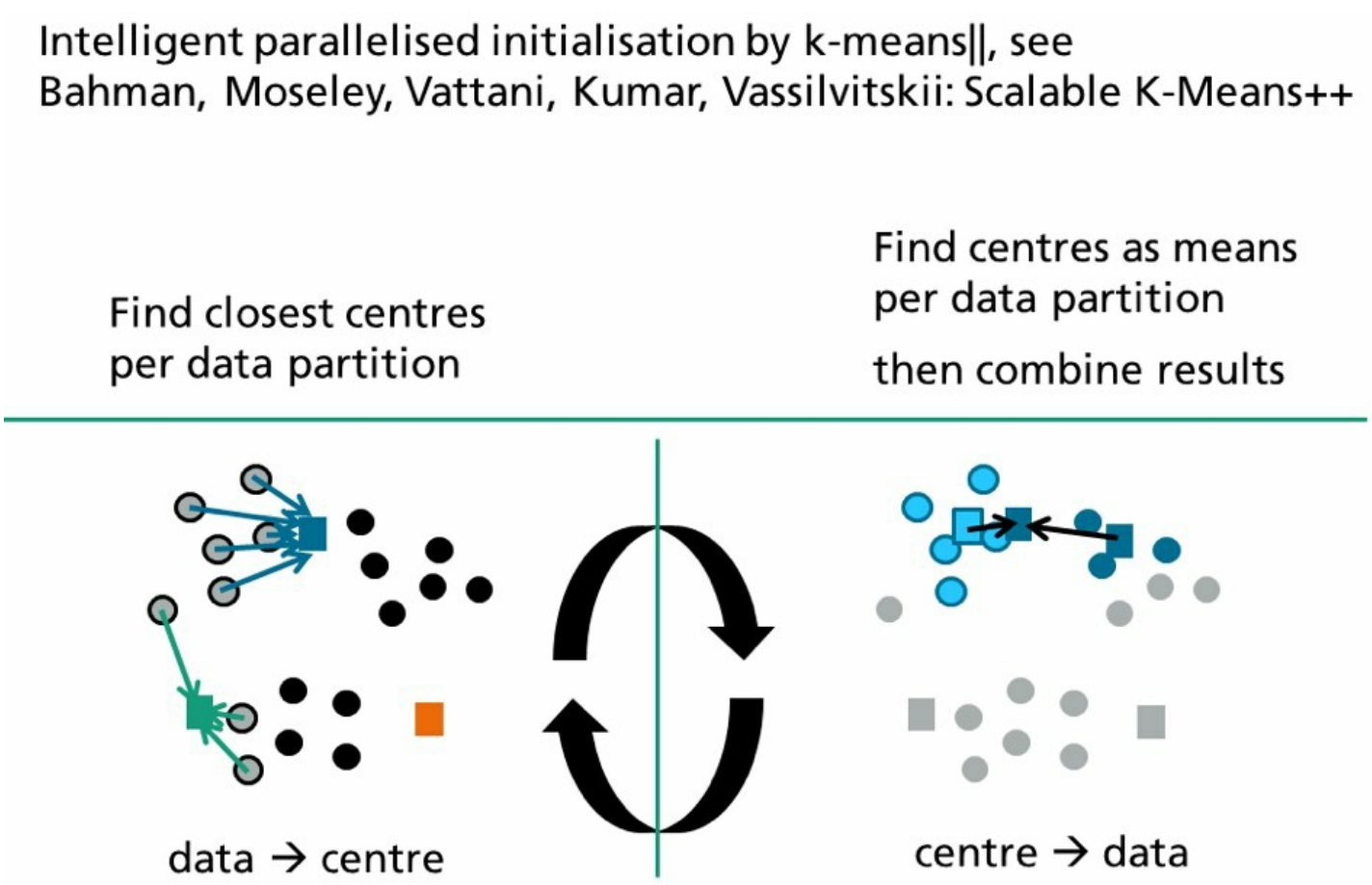


Figure 19: Parallelisation of the k-means algorithm.

In Spark this is in principle done by a single line of code again after preparing data. Fortunately, the ALS model already gives us well-prepared data for clustering users. It is a good exercise to adapt this example to clustering artists.

```
from pyspark.mllib.clustering import KMeans

# Prepare data from ALS
data = model.userFeatures().map(lambda x: x[1])

# Build the model (cluster the data), k=2
clusters = KMeans.train(data, 2, maxIterations=10, runs=10, initializationMode="random")
```

Note that the number of clusters is parameter to the algorithm. Also, the basic initialization of cluster centres can be selected.

Machine learning pipelines

Spark offers a framework for machine learning pipelines. These make it easier to combine multiple algorithms into a single workflow with a common interface to parameters. Pipelines are based on the following concepts:

DataFrame: Machine learning datasets are modelled as DataFrame (from Spark SQL). A DataFrame can have different columns storing text, feature vectors, ground truth labels, and predictions.

Transformer: This is an algorithm to transform a DataFrame into another DataFrame. A machine learning model can be interpreted as a Transformer which transforms a DataFrame containing features into a DataFrame containing predictions.

Estimator: This is an algorithm to transform a DataFrame into a Transformer. A learning algorithm can be understood as an Estimator which transforms a DataFrame containing the training set into a model (learning)

Pipeline: A pipeline chains multiple Transformers and Estimators together to specify a machine learning

workflow.

Parameter: All Transformers and Estimators share a common API for specifying parameters.

Once a pipeline is specified, it can be used to first learn models and then apply them to a dataset. In the first step Pipeline.fit is called to run the fit methods of all Estimators to create a model of the type PipelineModel. In the second step PipelineModel.transform is called to invoke the Transformers of the pipeline and thus apply the model to data.

In Spark this may look like this:

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer
from pyspark.sql import Row

# Prepare training documents from a list of (id, text, label) tuples.
LabeledDocument = Row("id", "text", "label")

training = sqlContext.createDataFrame([
(0L, "a b c d e spark", 1.0),
(1L, "b d", 0.0),
(2L, "spark f g h", 1.0),
(3L, "hadoop mapreduce", 0.0)], ["id", "text", "label"])
```

Here, we have prepared the training data: lines together with unique identifiers and classification labels have been prepared as a DataFrame.

```
# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and logistic regression.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)

pipeline = Pipeline( stages=[tokenizer, hashingTF, lr] )

# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

Now, we have defined a pipeline which can then be applied to the training data by calling its fit method. The pipeline consists of three stages: a tokenizer that converts input data to lower case characters and splits it according to white spaces, a feature extractor which maps a sequence of terms to their term frequencies based on hashing (hashingTF), and logistic regression as the classifier.

```
# Prepare test documents, which are unlabeled (id, text) tuples.
 test = sqlContext.createDataFrame([
(4L, "spark i j k"),
(5L, "l m n"),
(6L, "mapreduce spark"),
(7L, "apache hadoop")], ["id", "text"])

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
selected = prediction.select("id", "text", "prediction")

for row in selected.collect():
        print(row)
```

Finally, we prepare some more data as a DataFrame containing test data. This is then used with the model's transform function to produce predictions for classification labels.

Pipelines provide a nice way to experiment with alternative approaches to a machine learning problem and make it easier to keep all parameters together. Unfortunately, not all algorithms in the MLlib are already instances of Transformers or Estimators. For example the ALS algorithm is not yet of that form and is therefore harder to integrate into a pipeline.

Who's hot? Many linear regressions

Linear regression is available in both Spark and Python. It may be beneficial to use either the one or the other, depending on data size. Let us look at another example of using the last.fm dataset. In this case we will not compute a single large linear regression but rather compute many small linear regressions. The scenario (see Figure 20) is the following. We want to find out which artists are currently hot, i.e., are trending because of a rising number of listeners. We solve this by computing one linear regression per artist based on the number of listening events per day for that artist over the last few days.

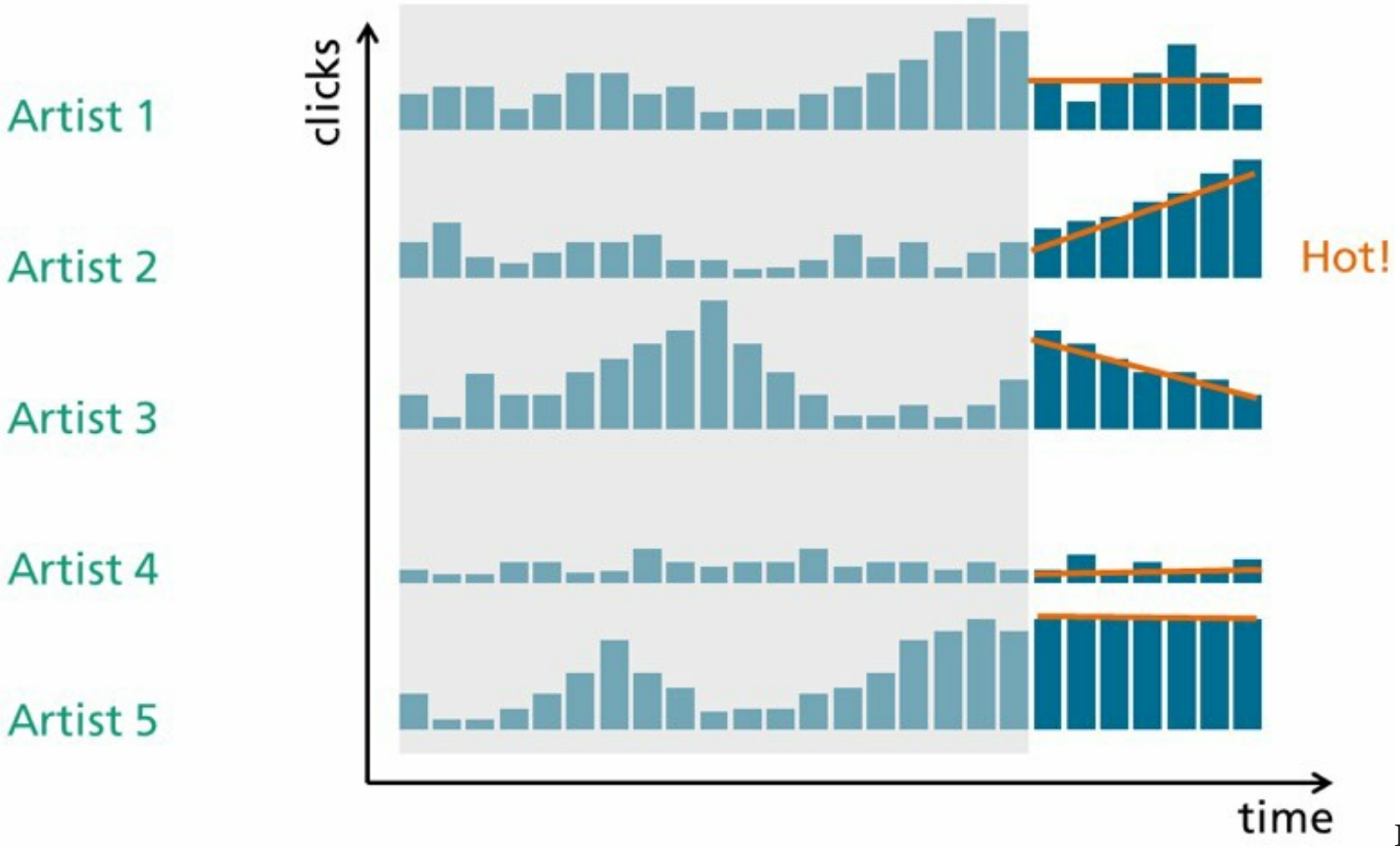


Figure 20: Using linear regression per artist to find which artist has an increasing number of listeners.

First, we will prepare the data again. We need listening event counts per day per artists.

```
# 1. Keep artist and parsed date/time
artist_times = words2.map(  lambda x: ( x[2], parser.parse(x[1]) )  )

# 2. Keep only February 2005
artist_times_filtered = artist_times.filter( lambda x: x[1].year==2005 and x[1].month==2  )

# 3. Produce entries of form ((artist, day), 1)
artist_times_days = artist_times_filtered.map( lambda x: ((x[0], x[1].day), 1) )

# 4. Group by (artist, day) pairs
artist_times_grouped = artist_times_days.reduceByKey(lambda x,y: x+y)

# 5. Produce entries of form (artist, (day, count))
artist_index = artist_times_grouped.map(  lambda x: (x[0][0], (x[0][1], x[1]))  )

# 6. Group by artist
artist_day_count = artist_index.groupByKey()
```

We now have, for each artist, a list of the form: (day1, count1), (day2, count2), ...

For the linear regression function in Python, we need two lists of the forms:

day1, day2, ... and count1, count2, ...

There is a nice Python method to solve this:

```
lrinput = artist_day_count.map(  lambda x: ( x[0], zip(*list(x[1])) )  )
```

Now the linear regressions can be computed in parallel using a map transformation and polyfit function from numpy.

```
import numpy as np

results = lrinput.map(lambda x: (x[0], np.polyfit(x[1][0], x[1][1], 1)))
```

polyfit can actually fit polynomials of any degree. In our case, we wish to use linear regression and hence linear polynomials, i.e., polynomials of degree 1.

Outlook

Complex event processing with Proton

The basic principle of complex event processing is to derive complex events on the basis of a possibly large number of simple events using an event processing logic. Proton on Storm allows running an open source complex event processing engine in a distributed manner on multiple machines using the Apache Storm infrastructure. Event processing networks provide a conceptual model describing the event processing flow execution. Such a network comprises a collection of event processing agents, event producers, and event consumers that are linked by channels.

You can learn more about Proton on Storm at:

http://github.com/ishkin/Proton/tree/master/IBM%20Proactive%20Technology%20Online%20on%20STORM

SQL operators for MapReduce with Teradata

Database management system providers seek to enhance their traditional databases and make them applicable to big data use-cases. A basic concept to achieve this is given by partitioning of tables, leading to massively parallel databases. Table operators allow making use of the partitioning for distributed algorithms using MapReduce. A selected commercial tool offering these approaches is the Teradata Aster solution.

You can learn more about this solution at:

http://www.teradata.de/products-and-services/analytics-from-aster-overview

In-memory processing

More and more main memory becomes available at a reasonable price. As access speed is reduced significantly once data outside of main memory is accessed, high performance applications focus on keeping as much data as possible in main memory. There is a wide variety of in-memory database systems available. Central performance and applicability measures to be kept in mind when choosing such a system comprise operating system compatibility, hardware requirements, license and support issues, runtime monitoring capabilities, memory utilisation, database interface standards, extensibility, portability, integration of open source big data technologies, local and distributed scaling and elasticity, available analytics functionality, persistence, availability, and security.

A first overview of available products can be won at:

http://en.wikipedia.org/wiki/List_of_in-memory_databases

A more in-depth study is available at:

http://s.fhg.de/in-memory-systems (German only, however).