Chapter 15:

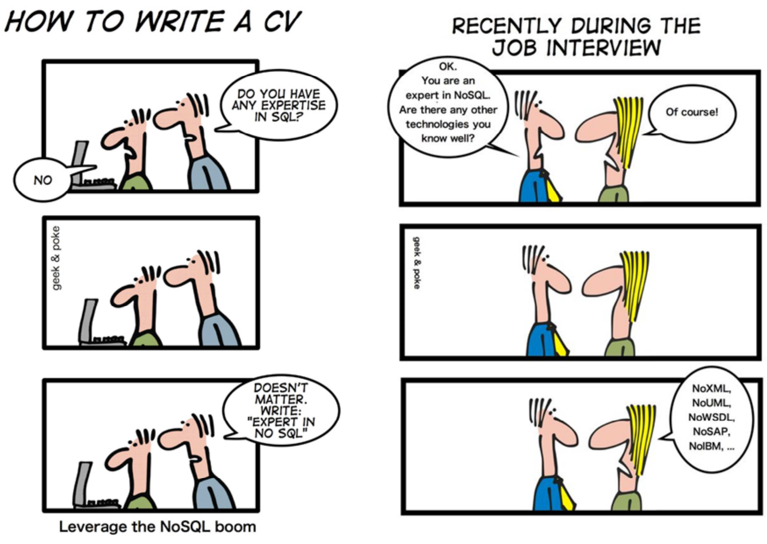
NoSQL Databases

2016-07-20
Johannes Schildgen
schildgen@cs.uni-kl.de

Recent Developments for Data Models 2016

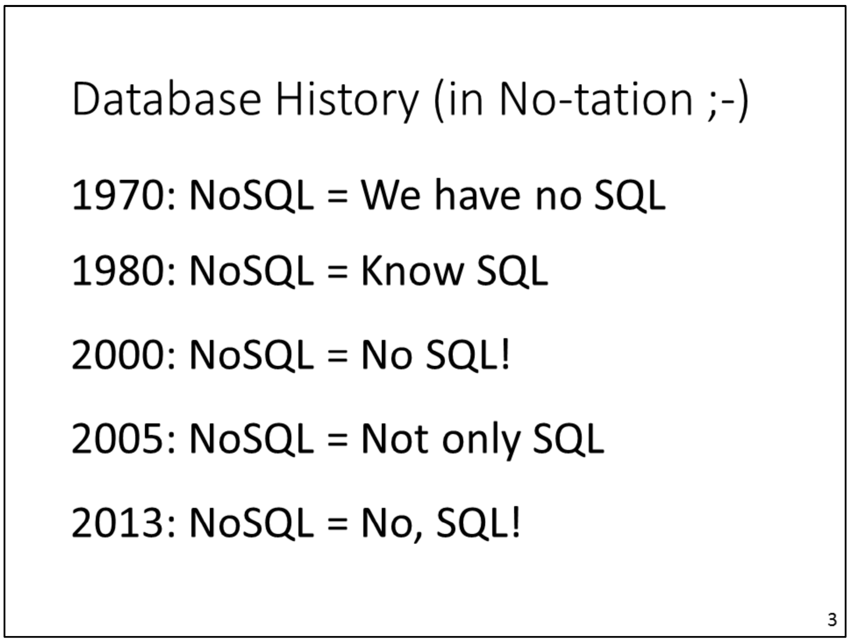
TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

1


HOW TO WRITE A CV
DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"
Leverage the NoSQL boom
RECENTLY DURING THE JOB INTERVIEW
OK. You are an expert in NoSQL. Are there any other technologies you know well?
Of course!
geek & poke
NoXML, NoUML, NoWSDL, NoSAP, NoIBM, ...

Source: http://geek-and-poke.com/

2



Database History (in No-tation ;-)

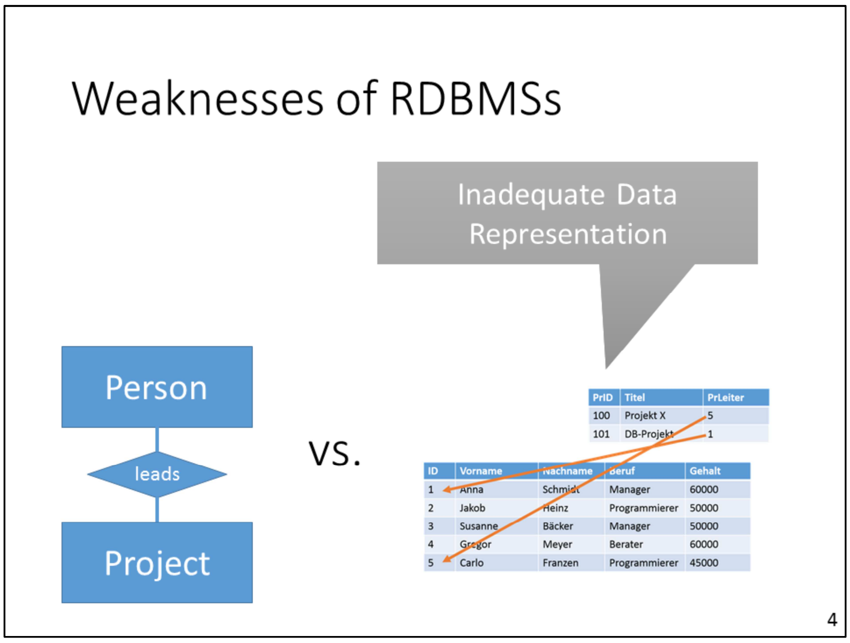
1970: NoSQL = We have no SQL

1980: NoSQL = Know SQL

2000: NoSQL = No SQL!

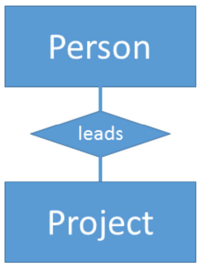
2005: NoSQL = Not only SQL

2013: NoSQL = No, SQL!



Weaknesses of RDBMSs

Inadequate Data Representation

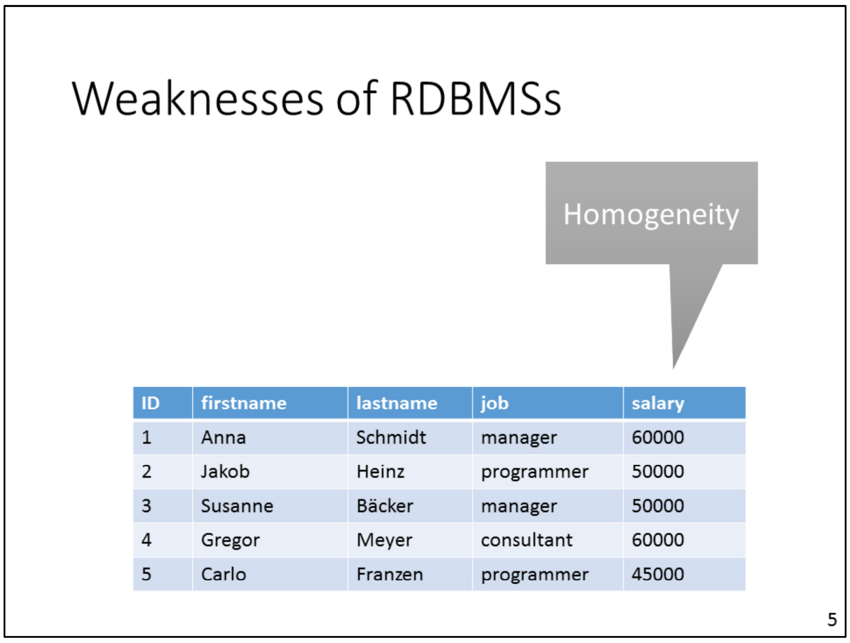


vs.

PrID	Titel	PrLeiter
100	Projekt X	5
101	DB-Projekt	1

ID	Vorname	Nachname	Beruf	Gehalt
1	Anna	Schmidt	Manager	60000
2	Jakob	Heinz	Programmierer	50000
3	Susanne	Bäcker	Manager	50000
4	Gregor	Meyer	Berater	60000
5	Carlo	Franzen	Programmierer	45000

(Source: WIESE, Lena. Advanced Data Management. 2015. Chapter 3)
The data is “squeezed” into the relational schema. Data that belongs together is split into multiple tables and connected via foreign-key relationships (normalization). In the application this data need to be joined again (denormalization).

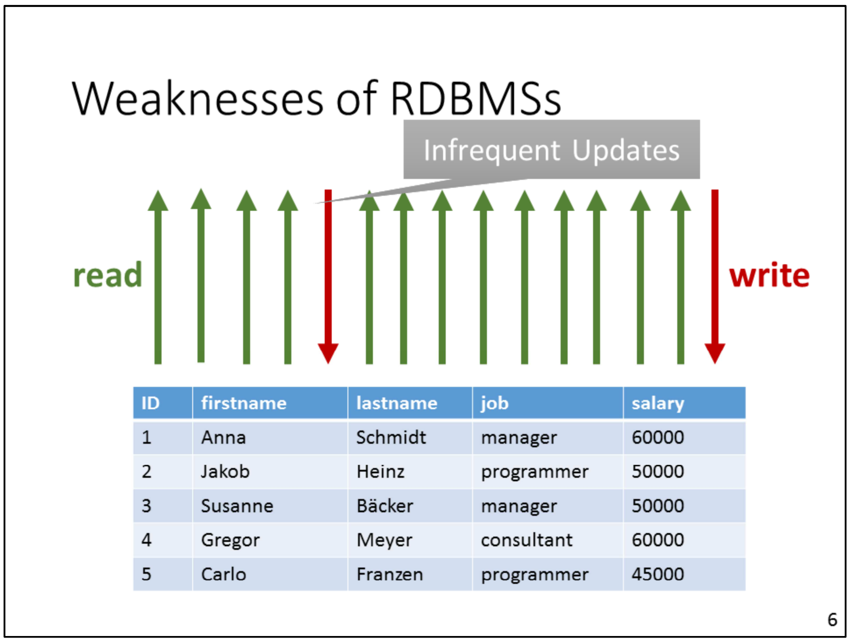


Weaknesses of RDBMSs

Homogeneity

ID	firstname	lastname	job	salary
1	Anna	Schmidt	manager	60000
2	Jakob	Heinz	programmer	50000
3	Susanne	Bäcker	manager	50000
4	Gregor	Meyer	consultant	60000
5	Carlo	Franzen	programmer	45000

Horizontal Homogeneity: Each row in the table has the same columns. When a new column is added, every row will have this new column (with many NULL values => sparse data).
Vertical Homogeneity: Within one column, all rows have the same data type.

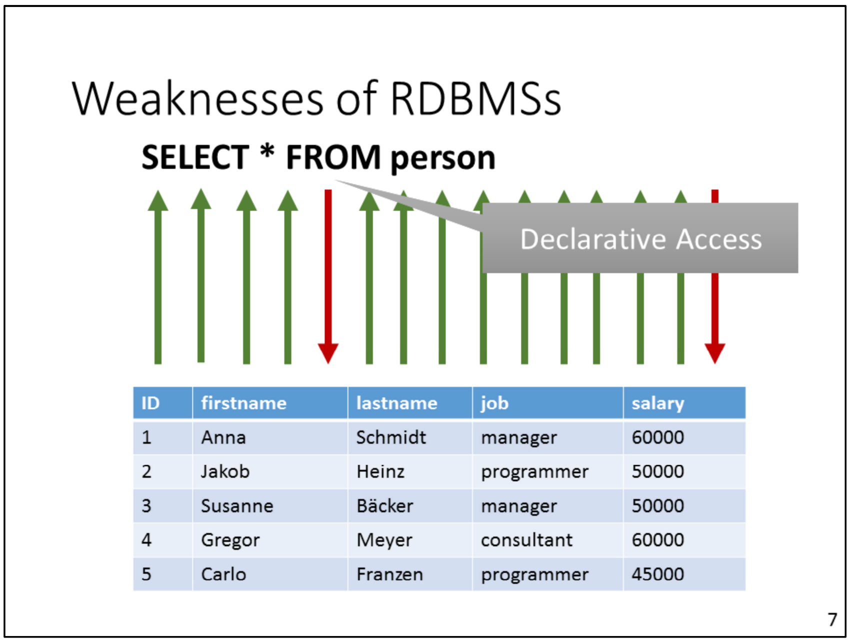


Weaknesses of RDBMSs

Infrequent Updates
read
write

ID	firstname	lastname	job	salary
1	Anna	Schmidt	manager	60000
2	Jakob	Heinz	programmer	50000
3	Susanne	Bäcker	manager	50000
4	Gregor	Meyer	consultant	60000
5	Carlo	Franzen	programmer	45000

Relational databases are made for applications that often read and less often write data.



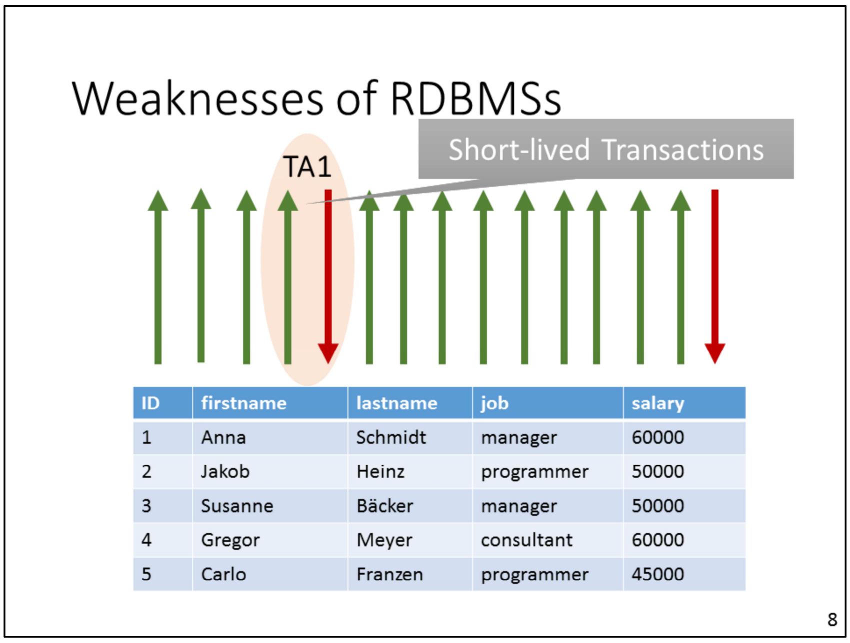
Weaknesses of RDBMSs

SELECT * FROM person

ID	firstname	lastname	job	salary
1	Anna	Schmidt	manager	60000
2	Jakob	Heinz	programmer	50000
3	Susanne	Bäcker	manager	50000
4	Gregor	Meyer	consultant	60000
5	Carlo	Franzen	programmer	45000

7

The declarative query language SQL is very powerful, but often a simple Put/Get API would be desirable. Furthermore, SQL is only well-suited for relational databases. For other databases, there are better ways, e.g. XQuery for XML.

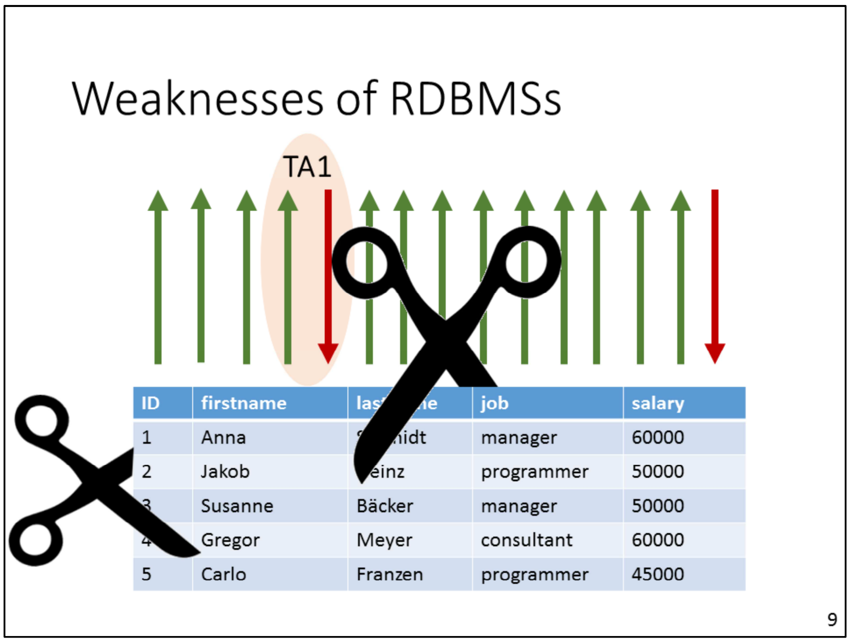


Weaknesses of RDBMSs

ID	firstname	lastname	job	salary
1	Anna	Schmidt	manager	60000
2	Jakob	Heinz	programmer	50000
3	Susanne	Bäcker	manager	50000
4	Gregor	Meyer	consultant	60000
5	Carlo	Franzen	programmer	45000

8

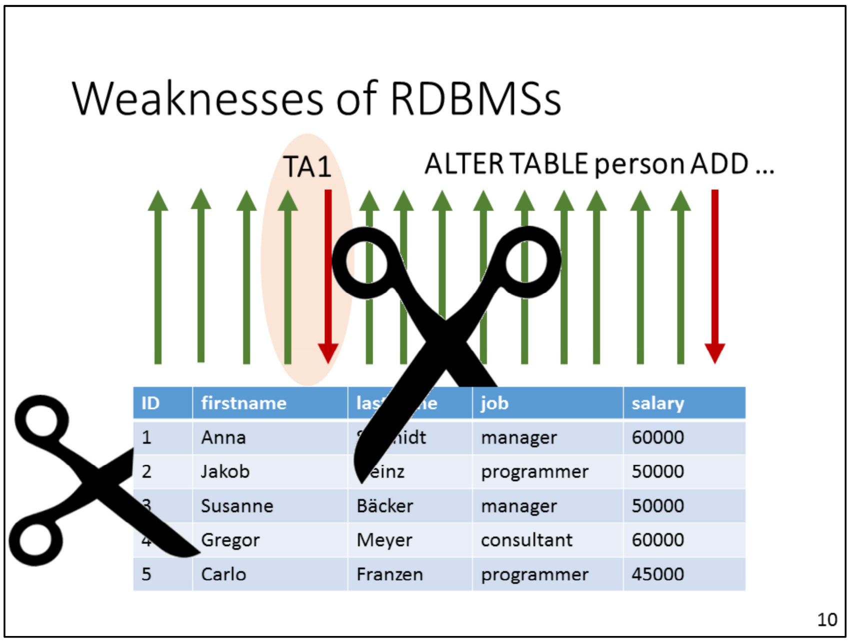
For supporting ACID transactions, lock mechanism are needed. To avoid long-duration locks, transactions should be as short as possible. For data-analytics or data-transformation tasks, long transactions would be desirable.



Weaknesses of RDBMSs

ID	firstname	las[illegible]e	job	salary
1	Anna	[illegible]idt	manager	60000
2	Jakob	[illegible]einz	programmer	50000
3	Susanne	Bäcker	manager	50000
4	Gregor	Meyer	consultant	60000
5	Carlo	Franzen	programmer	45000

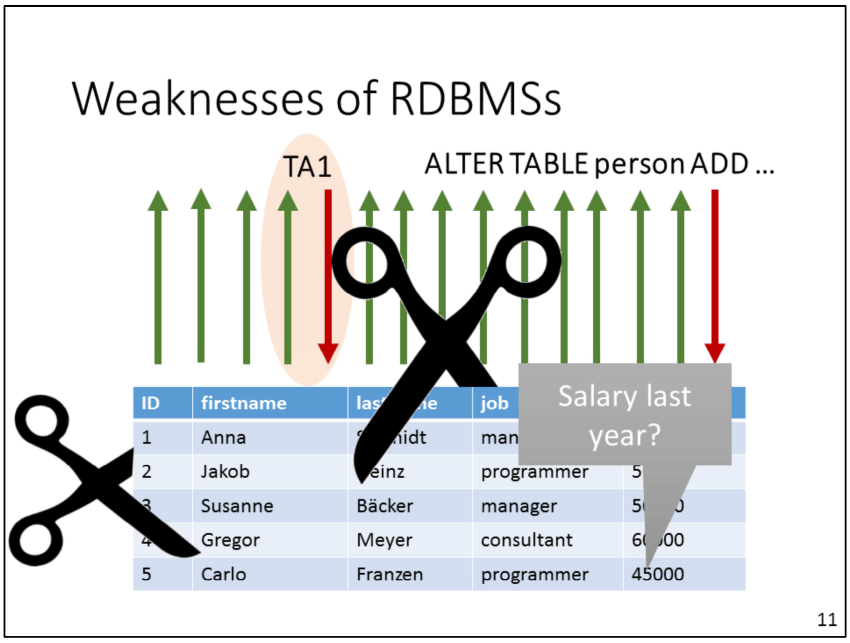
Tables in relational databases cannot be distributed across multiple nodes properly. That is why most RDBMS run on one machine. In distributed environments, joins are very costly, because a lot of data has to be shipped over the network.



Weaknesses of RDBMSs

ID	firstname	las[illegible]e	job	salary
1	Anna	[illegible]idt	manager	60000
2	Jakob	[illegible]einz	programmer	50000
3	Susanne	Bäcker	manager	50000
4	Gregor	Meyer	consultant	60000
5	Carlo	Franzen	programmer	45000

Schema evolution (adding columns, etc.) is very costly, because it often requires a reorganization of the data.

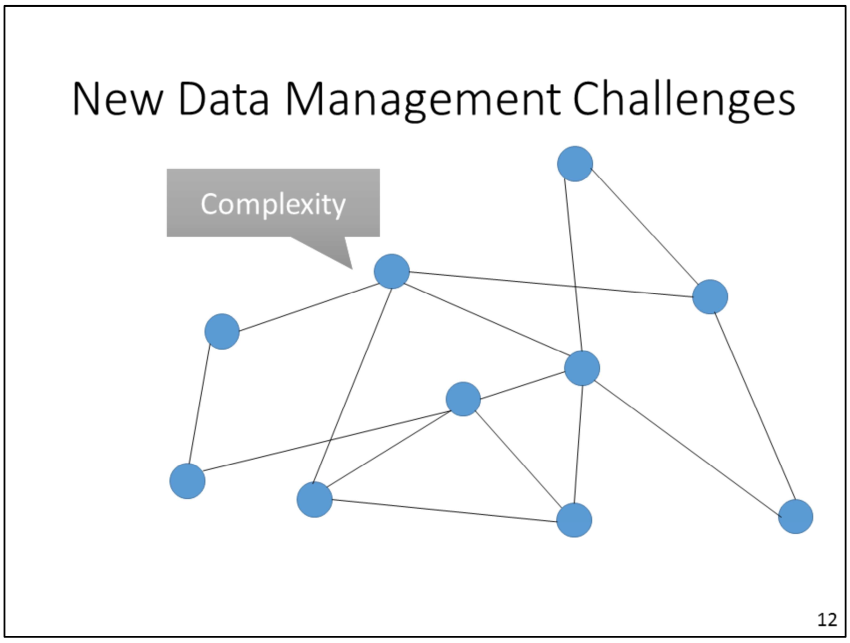


Weaknesses of RDBMSs

ID	firstname	las[illegible]e	job	[illegible]
1	Anna	[illegible]idt	man[illegible]	[illegible]
2	Jakob	[illegible]einz	programmer	5[illegible]
3	Susanne	Bäcker	manager	5[illegible]0
4	Gregor	Meyer	consultant	6[illegible]00
5	Carlo	Franzen	programmer	45000

11

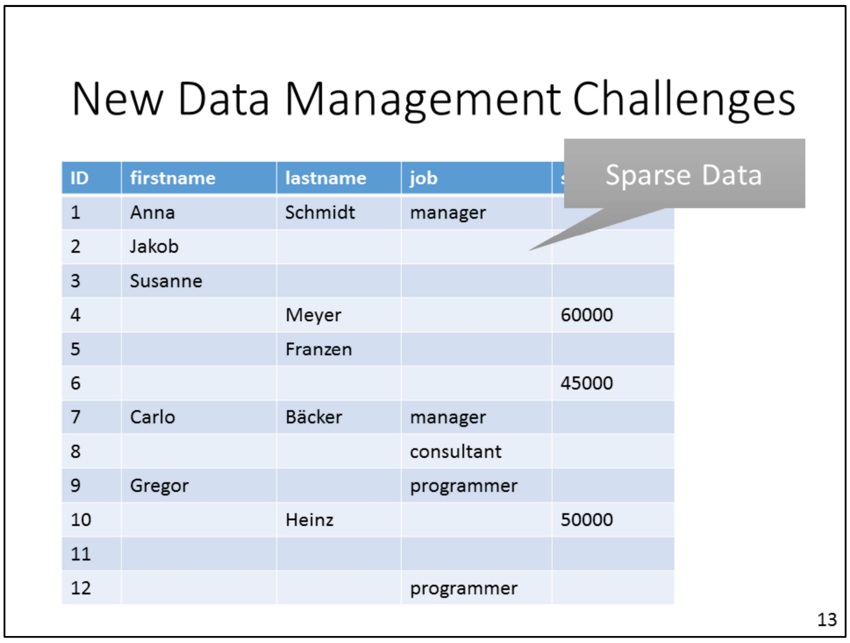
Originally, relational databases do not support data versioning. In some applications it can be useful to access historical data, i.e. former versions.



New Data Management Challenges

12

Not only structured data, but also semi-structured data, graph data, unstructured data, media data (more later), …



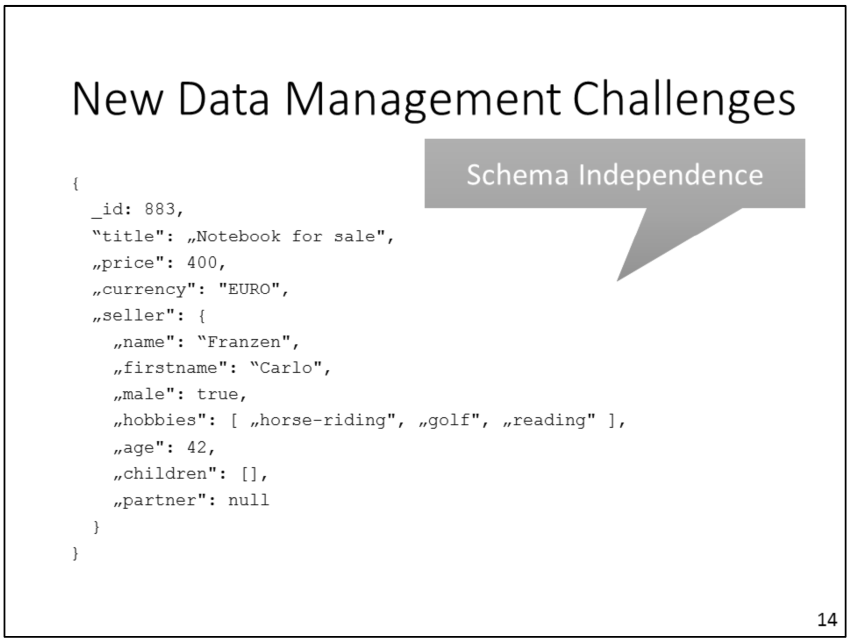
New Data Management Challenges

ID	firstname	lastname	job	s
1	Anna	Schmidt	manager	
2	Jakob			
3	Susanne			
4		Meyer		60000
5		Franzen		
6				45000
7	Carlo	Bäcker	manager	
8			consultant	
9	Gregor		programmer	
10		Heinz		50000
11				
12			programmer	

Sparse Data

13

When many attribute values are incomplete or unknown, a fixed schema would lead to wide tables with many columns and many NULL values. This needs a lot of storage and can make queries inefficient.



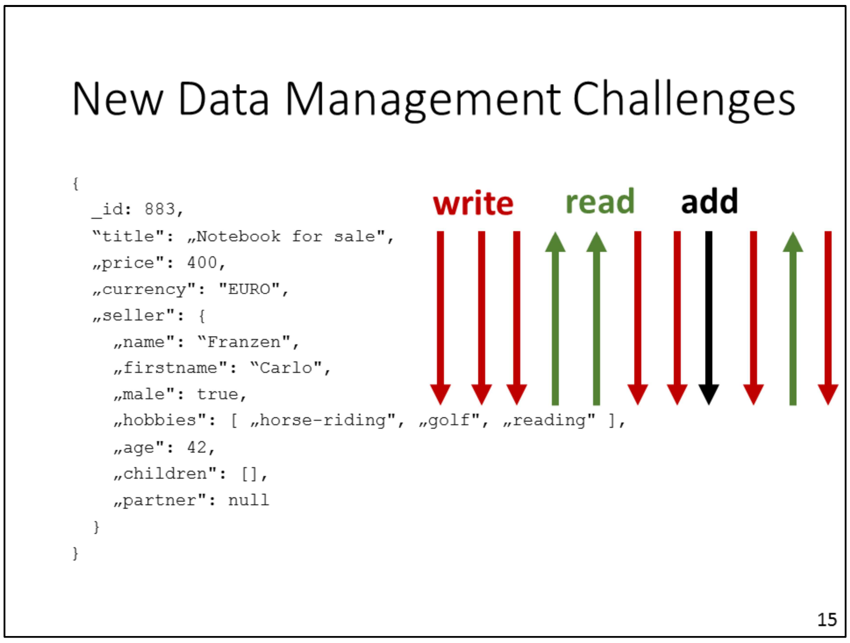
New Data Management Challenges

```
{
  _id: 883,
  "title": „Notebook for sale",
  „price": 400,
  „currency": "EURO",
  „seller": {
    „name": "Franzen",
    „firstname": "Carlo",
    „male": true,
    „hobbies": [ „horse-riding", „golf", „reading" ],
    „age": 42,
    „children": [],
    „partner": null
  }
}
```

Schema Independence

14

With a flexible schema, it is possible to save data with an arbitrary structure (attributes, data types, nesting). There is no need to predefine a schema. Within one data collection, different data items can look completely different. E.g. in another item, the hobbies of a seller can be a comma-separated string.



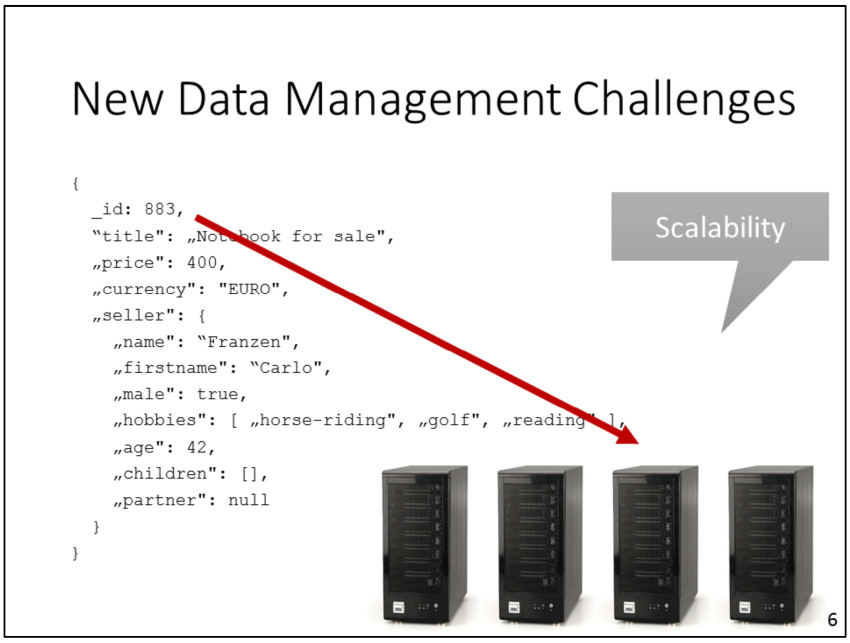
New Data Management Challenges

```
{
  _id: 883,
  “title": „Notebook for sale",
  „price": 400,
  „currency": "EURO",
  „seller": {
    „name": “Franzen",
    „firstname": “Carlo",
    „male": true,
    „hobbies": [ „horse-riding", „golf", „reading" ],
    „age": 42,
    „children": [],
    „partner": null
  }
}
```

write
read
add

15

Data is always changing. A database system must support fast changes, also schema changes.



New Data Management Challenges

```
{
  _id: 883,
  “title": „Notebook for sale",
  „price": 400,
  „currency": "EURO",
  „seller": {
    „name": “Franzen",
    „firstname": “Carlo",
    „male": true,
    „hobbies": [ „horse-riding", „golf", „reading" ],
    „age": 42,
    „children": [],
    „partner": null
  }
}
```

Scalability

6

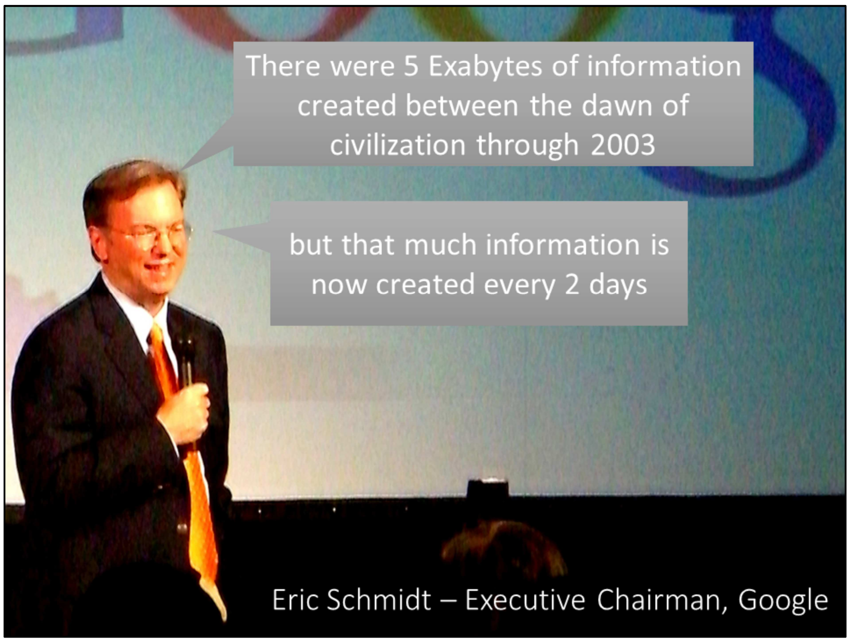
Data should be distributed across multiple machines. For an application it should look like there is only one location where all data lives. It should not need to know the exact storage location for each data item. Horizontal scalability means that a system can be extended by adding more machines. Adding and removing machines should be possible during runtime and the data should be distributed, partitioned and shipped automatically.

New Data Management Challenges

Big Data

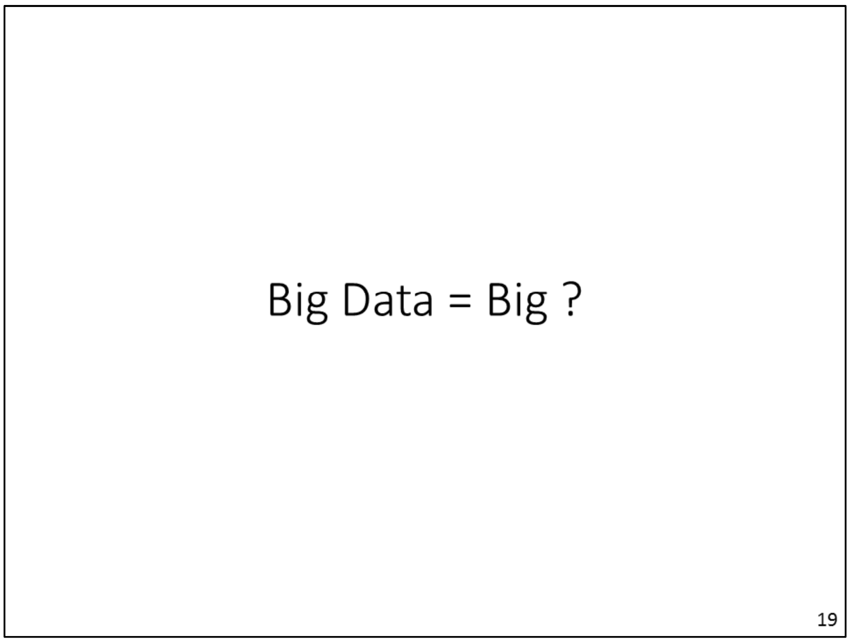
17

17


There were 5 Exabytes of information created between the dawn of civilization through 2003
but that much information is now created every 2 days
Eric Schmidt – Executive Chairman, Google

Eric Schmidt.jpg, Author: Gisela Giardino, Wikimedia Commons, 12 April 2007, 20:24:40

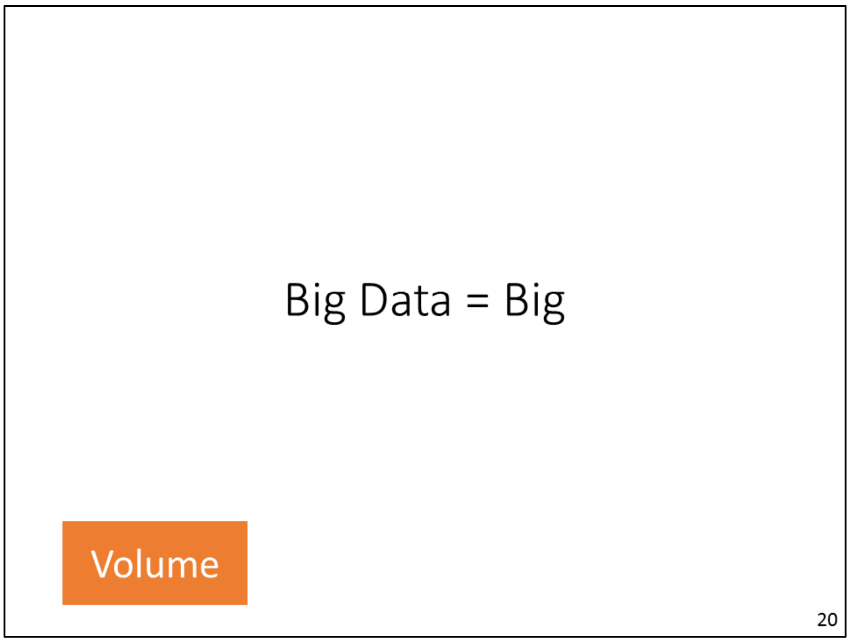
18



Big Data = Big ?

19

19



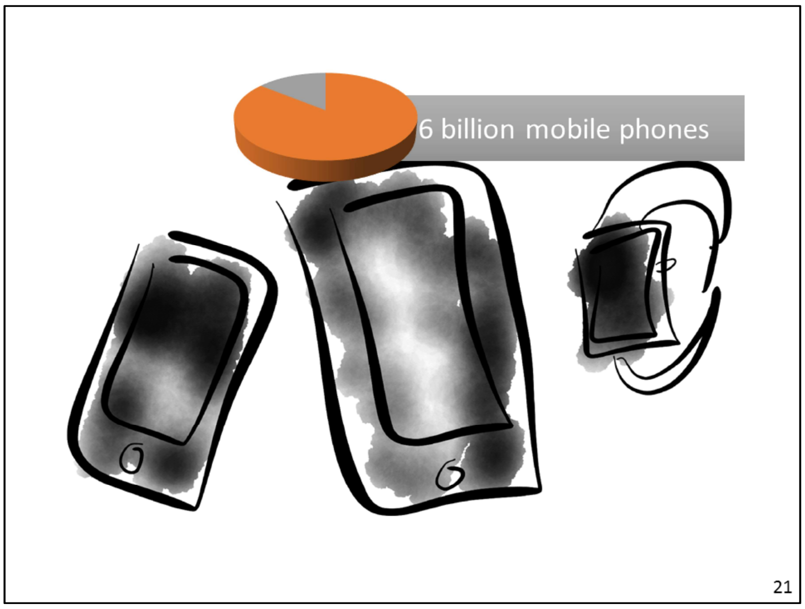
Big Data = Big

Volume

20

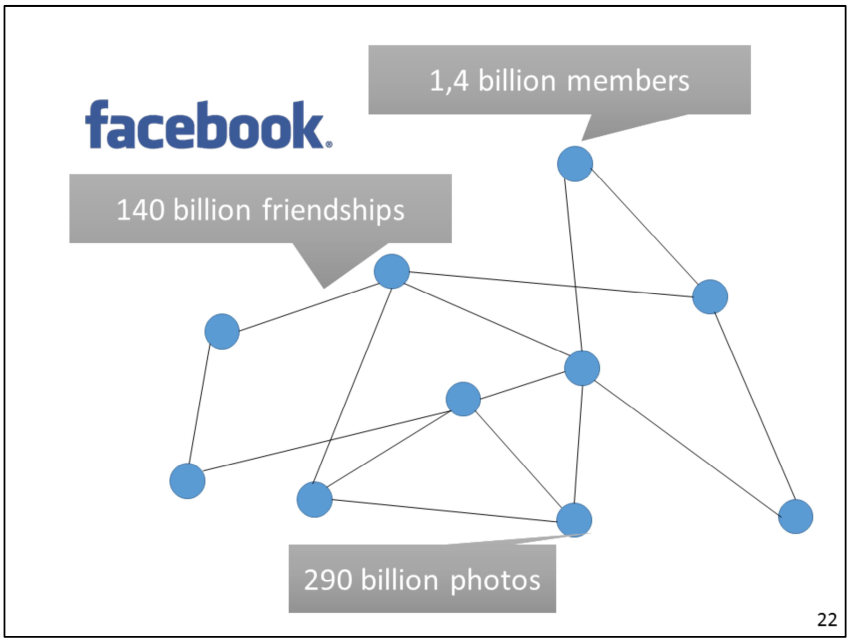
(At least) 3 Vs describe the characteristics of Big Data (see http://www.gartner.com/newsroom/id/1731916)

20



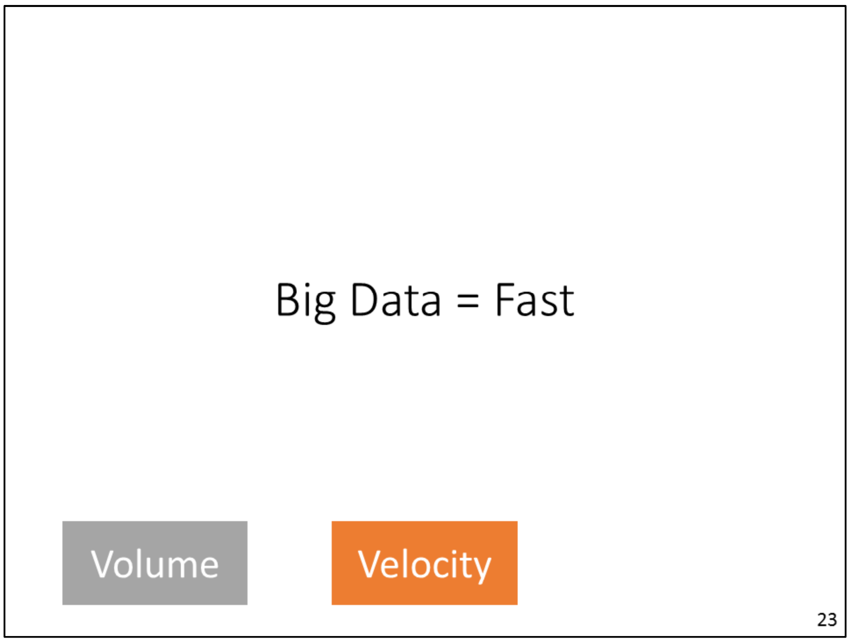
Huge amount of data is produced by mobile phones: locations, app-store data, cloud services, chat protocols, phone logs, and much more.

21



300 Petabytes (?) = 300 Million Gigabyte

22



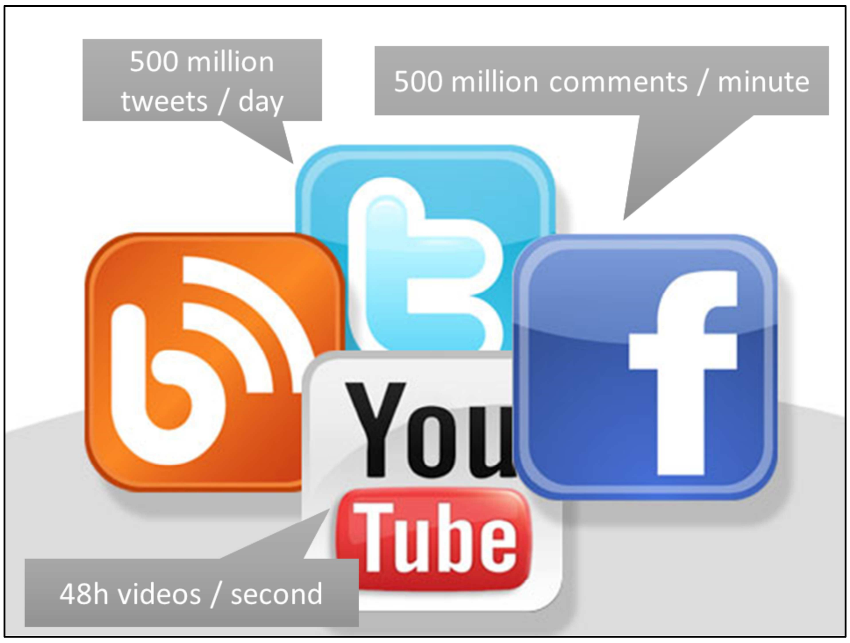
Big Data = Fast

Volume

Velocity

23

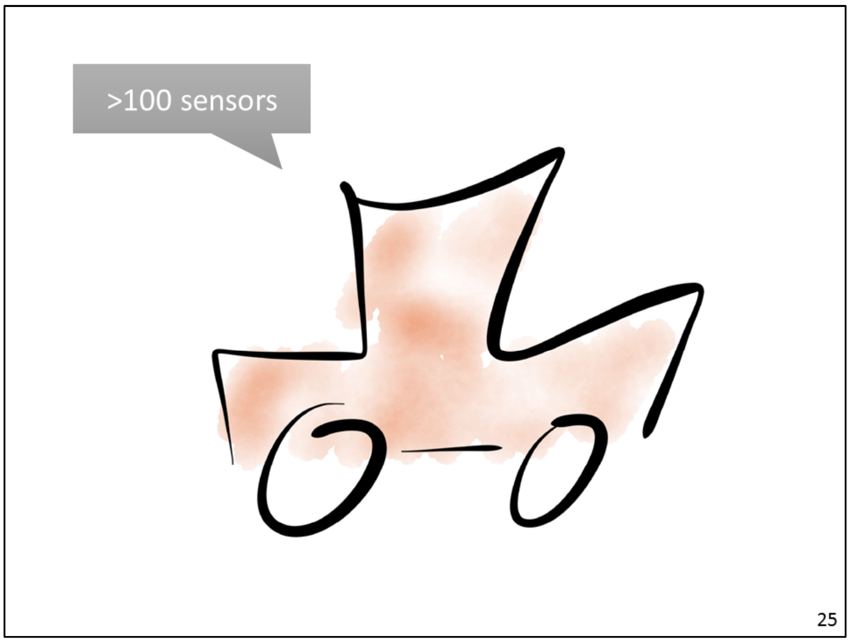
23



Furthermore, every visit and “like” of this contents produces data.

Socmed - Flickr - USDAgov.jpg, Author: U.S. Department of Agriculture, Wikimedia Commons, 19 December 2011, 15:38

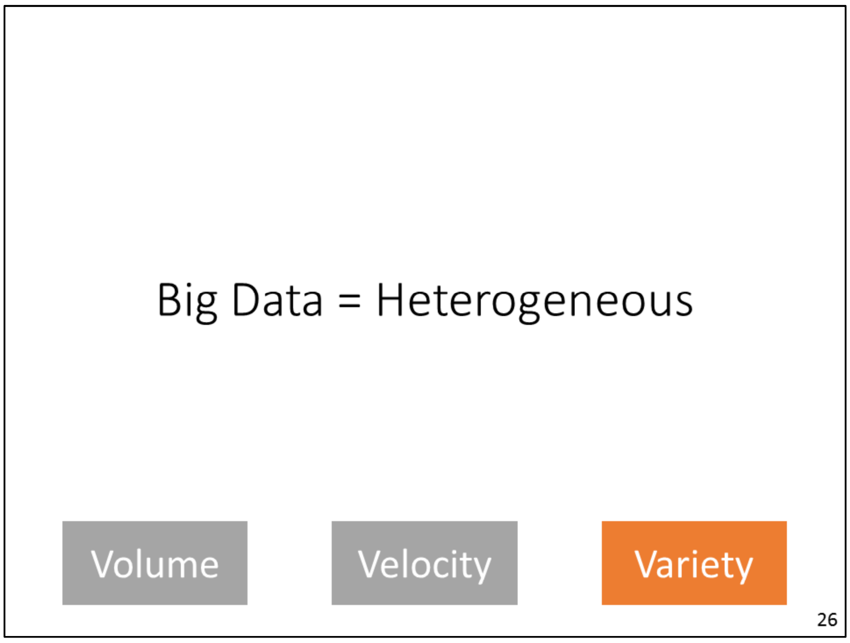
24



25

Sensors in cars, planes and colliders (e.g., the LHC at CERN) produce data so fast that it is not possible to process it in real-time.

25

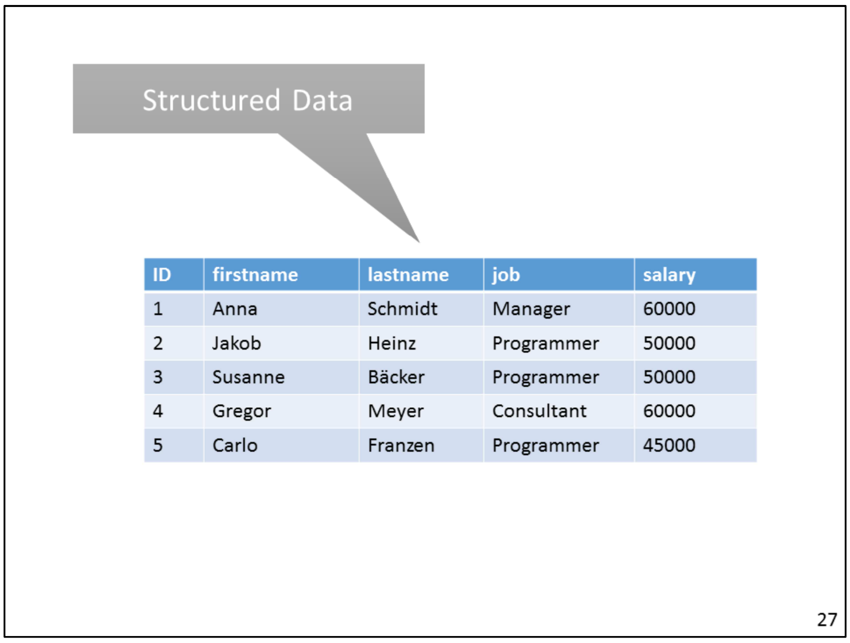


Big Data = Heterogeneous

Volume Velocity Variety

26

26



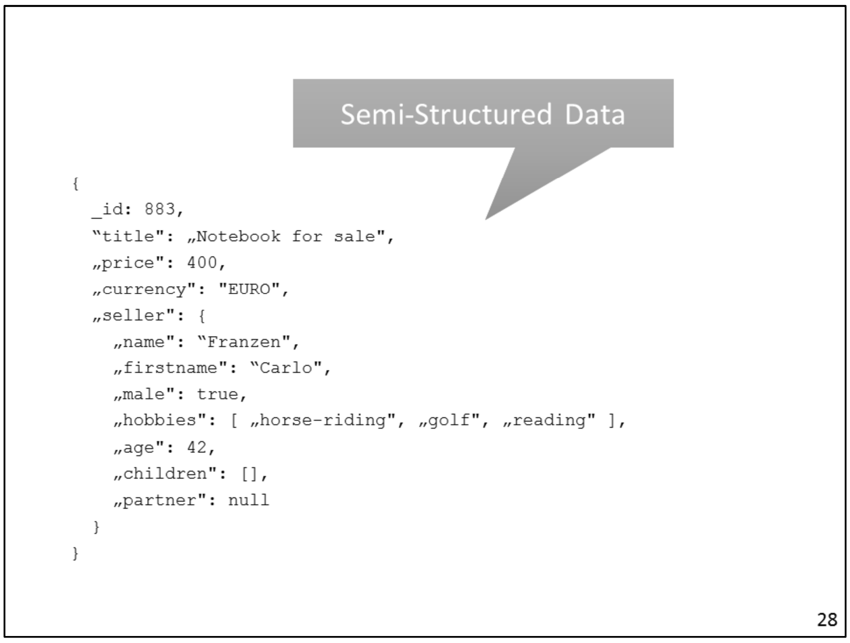
Structured Data

ID	firstname	lastname	job	salary
1	Anna	Schmidt	Manager	60000
2	Jakob	Heinz	Programmer	50000
3	Susanne	Bäcker	Programmer	50000
4	Gregor	Meyer	Consultant	60000
5	Carlo	Franzen	Programmer	45000

27

This is a relational table, e.g. in MySQL, SQLite, DB2, PostgreSQL.
You first create the schema (CREATE TABLE). After that, data can be inserted (INSERT). Changing the schema is supported (ALTER TABLE).

27



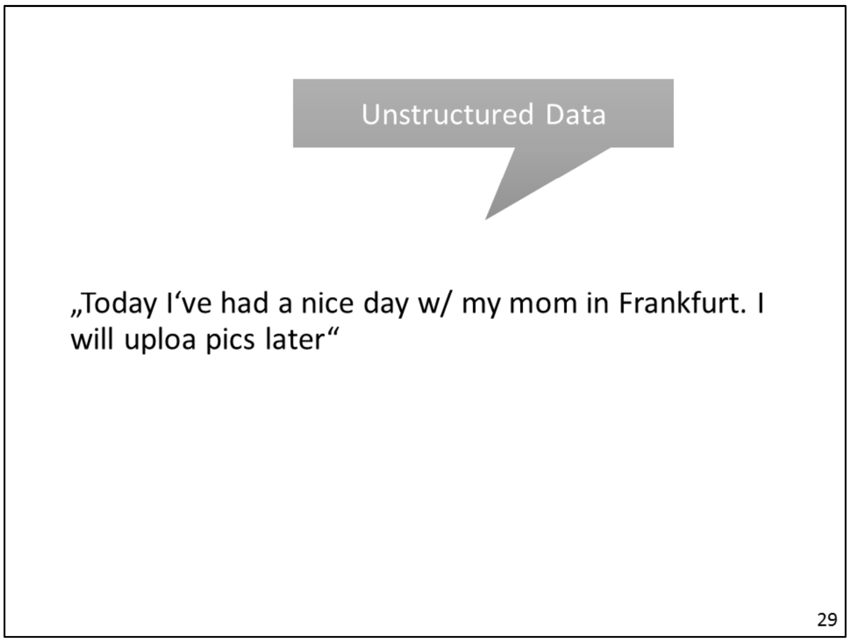
Semi-Structured Data

```
{
  _id: 883,
  “title": „Notebook for sale",
  „price": 400,
  „currency": "EURO",
  „seller": {
    „name": “Franzen",
    „firstname": “Carlo",
    „male": true,
    „hobbies": [ „horse-riding", „golf", „reading" ],
    „age": 42,
    „children": [],
    „partner": null
  }
}
```

28

This is a JSON document. Similar to XML, it allows to model a tree structure. A field value can be an atomic value, an array of values, or a sub-document. Different from relational databases, the schema in JSON does not have to be defined in advance. This schema-flexibility allows for using arbitrary attributes when documents are inserted or updated. Applications often expect a specific schema of the data (Schema-on-Read).

28

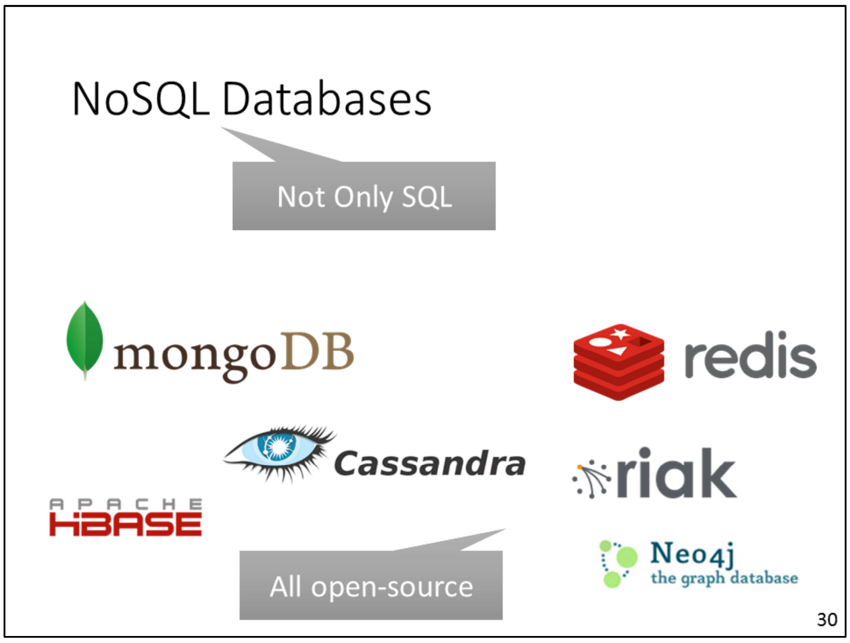

Unstructured Data

„Today I‘ve had a nice day w/ my mom in Frankfurt. I will uploa pics later“

29

Unstructured data do not follow any schema. They are very hard to analyze and process. Problems here are: slang („pics“), stop words („a“, „in“), word forms („had“), compound words, typos („uploa“), references (Pics of Frankfurt? Pics of the mother? Pics of a person called Will?), and more.

29


NoSQL Databases

Not Only SQL

mongoDB

redis

Cassandra

riak

APACHE HBASE

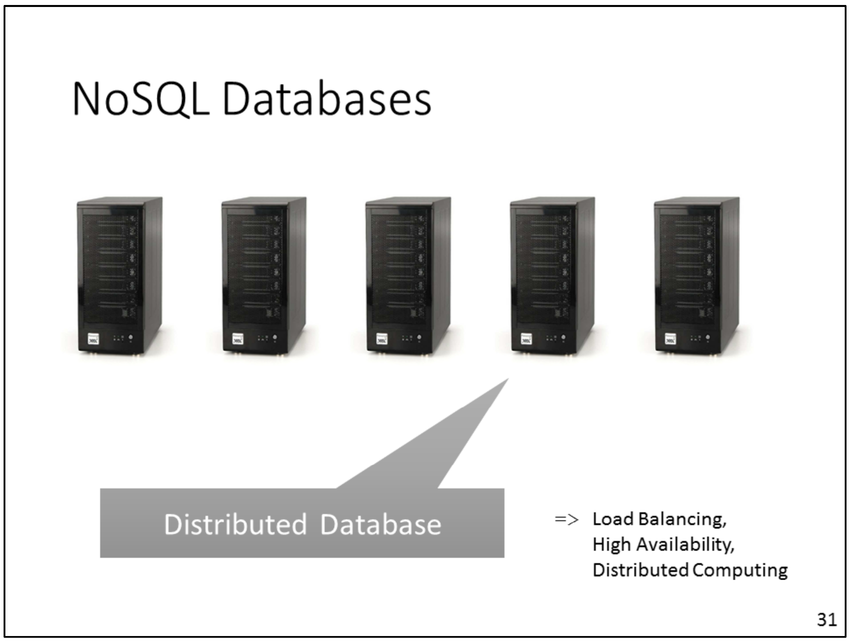
Neo4j the graph database

All open-source

30

NoSQL databases are non-relational.

30

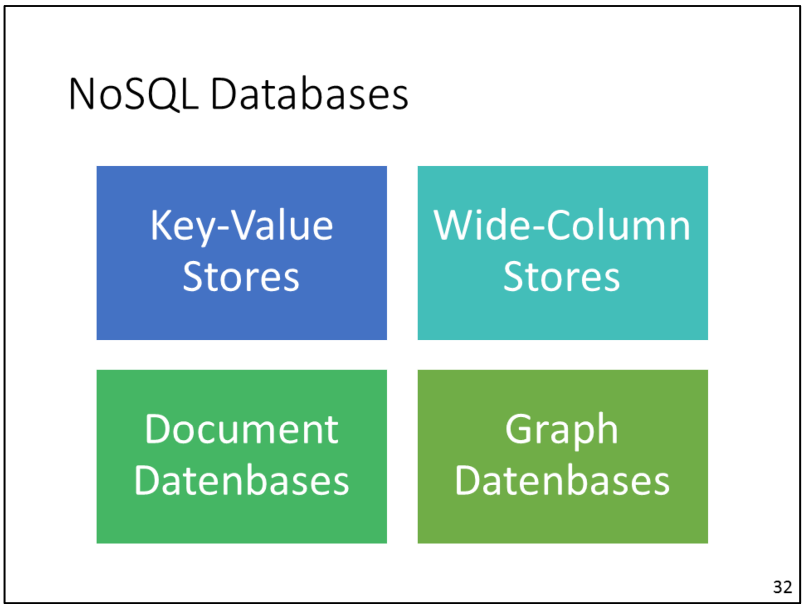


NoSQL Databases

=> Load Balancing,
High Availability,
Distributed Computing

31

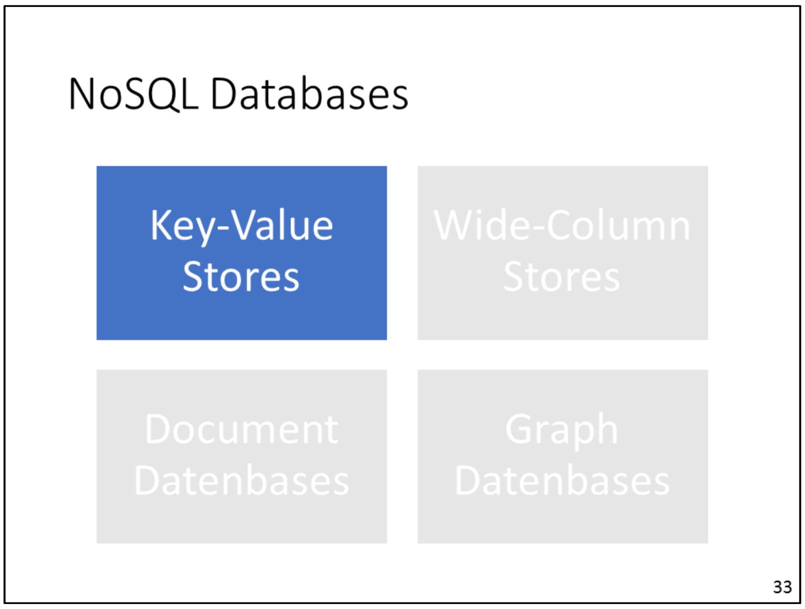
NoSQL databases are distributed. A scale-out (adding more machines) is easily possible. Relational databases often run on one single powerful machine because joins and other operations are hard to execute across multiple nodes. Often, the only possibility to increase the performance in an RDBMS, is a scale up (add more RAM or a better CPU on the machine). A scale up is expensive and limited.



NoSQL Databases

32

In the literature, NoSQL is categorized using these four classes. http://nosql-database.org/ shows a list of more than 200 NoSQL databases. Most of them belong to one of these four classes, but there are also multi-media databases, object databases, XML databases and more.



NoSQL Databases

Key-Value Stores

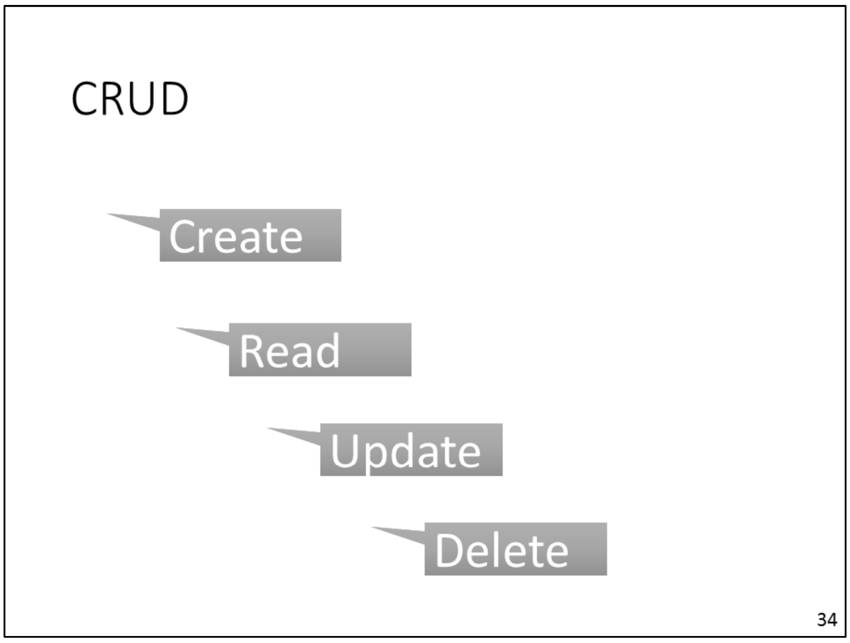
Wide-Column Stores

Document Datenbases

Graph Datenbases

33

33



CRUD

Create

Read

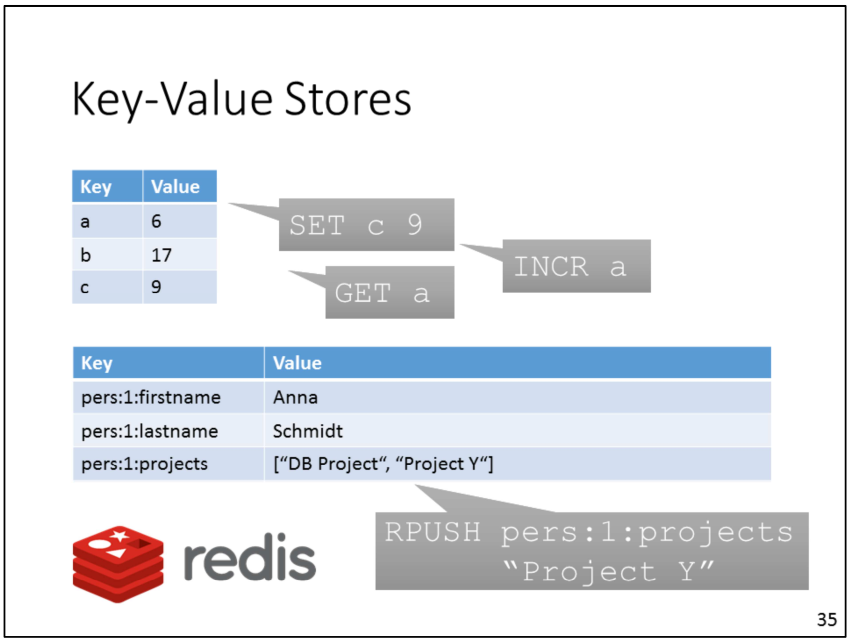
Update

Delete

34

Each database system has a so-called CRUD-API. These are methods to create, read, update and delete data items. In SQL, the equivalent language constructs are INSERT, SELECT, UPDATE, and DELETE. In NoSQL databases, the CRUD-API is often much simpler than SQL.

34



Key-Value Stores

Key	Value
a	6
b	17
c	9

```
SET c 9
GET a
INCR a
```

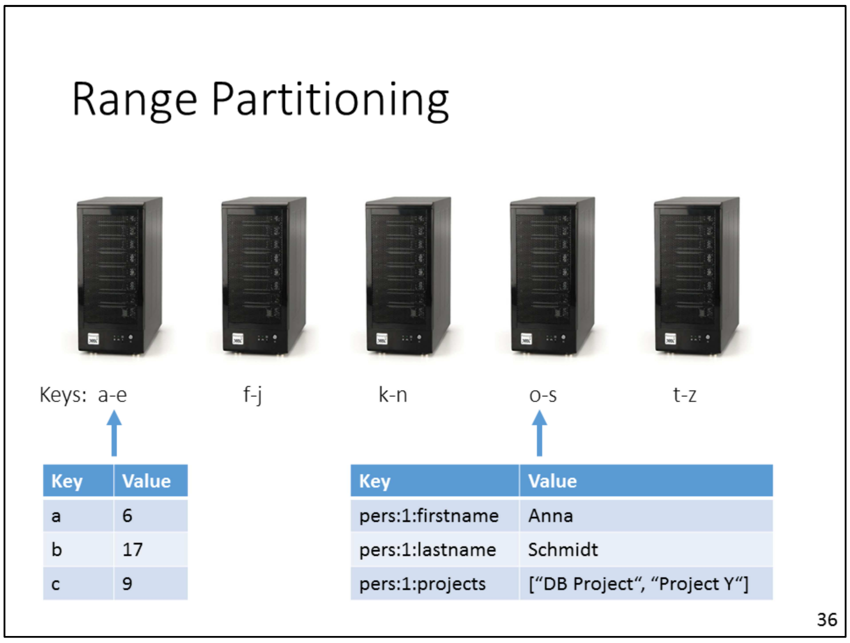
Key	Value
pers:1:firstname	Anna
pers:1:lastname	Schmidt
pers:1:projects	["DB Project", "Project Y"]

redis

```
RPUSH pers:1:projects
      "Project Y"
```

35

A key-value store saves key-value pairs (like a Hash Map). The values can be complex, e.g. a list or a map. The API is very simple: GET key, SET key value, etc. RPUSH adds an item to the right end of a list. RPOP reads and removes this item. LPUSH and LPOP do the same thing from the left of the list.



Range Partitioning

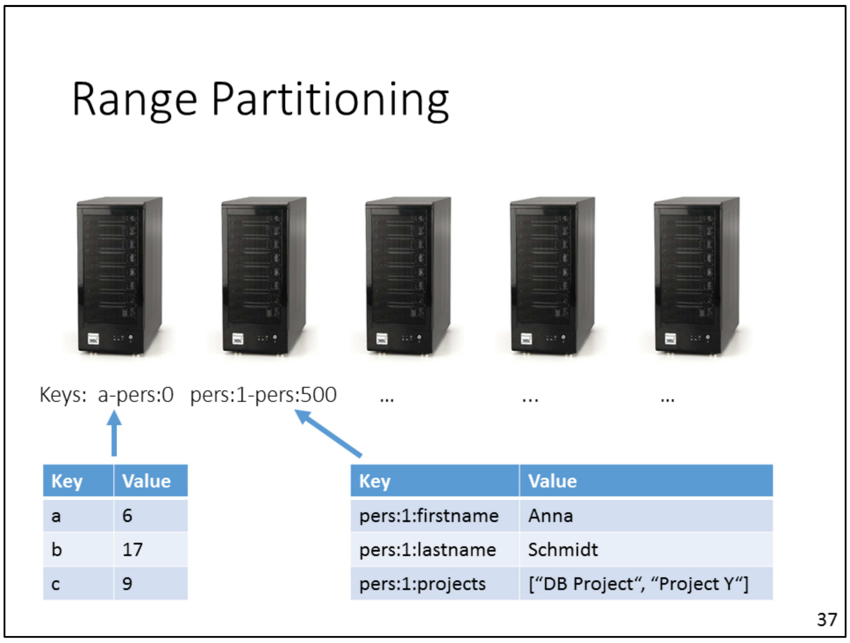
Keys: a-e f-j k-n o-s t-z

Key	Value
a	6
b	17
c	9

Key	Value
pers:1:firstname	Anna
pers:1:lastname	Schmidt
pers:1:projects	["DB Project", "Project Y"]

36

Key-value stores, wide-colum stores and document databases belong to the so-called **aggregate-oriented stores**. Here, aggregate means „the whole thing". So, everything that belongs together is stored together. An example: A blog entry is stored with all its comments and a view counter instead of splitting it into multiple tables and connecting the fragments via foreign keys. All aggregate-oriented stores have the concept of a unique object ID. Often, data partitioning across multiple machines is based on this ID. Partitioning is also called **sharding**.



Range Partitioning

Keys: a-pers:0 pers:1-pers:500 ... ...

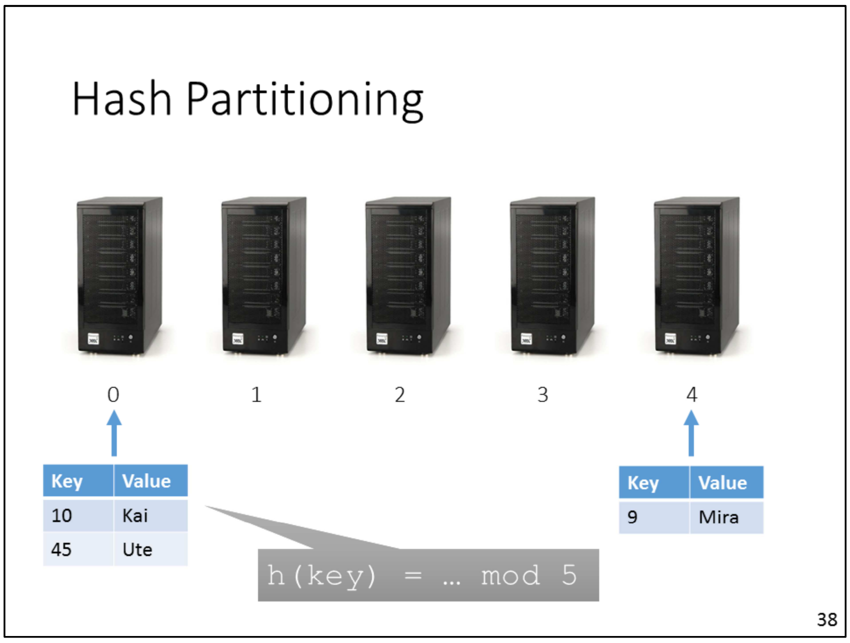
Key	Value
a	6
b	17
c	9

Key	Value
pers:1:firstname	Anna
pers:1:lastname	Schmidt
pers:1:projects	["DB Project", "Project Y"]

37

37

When range partitioning is used, every machine stores data items that have a specific range of keys. Consequently, there must exist an order on the keys. The ranges must be chosen wisely, so that data items are equally distributed over all machines. This can be reached by an automatic repartitioning performed periodically. In this task, the ranges are changed, which leads to moving data from one machine to another. Read and write operations that are based on the partitioning key always know which machine to access.



Hash Partitioning

Key	Value
10	Kai
45	Ute

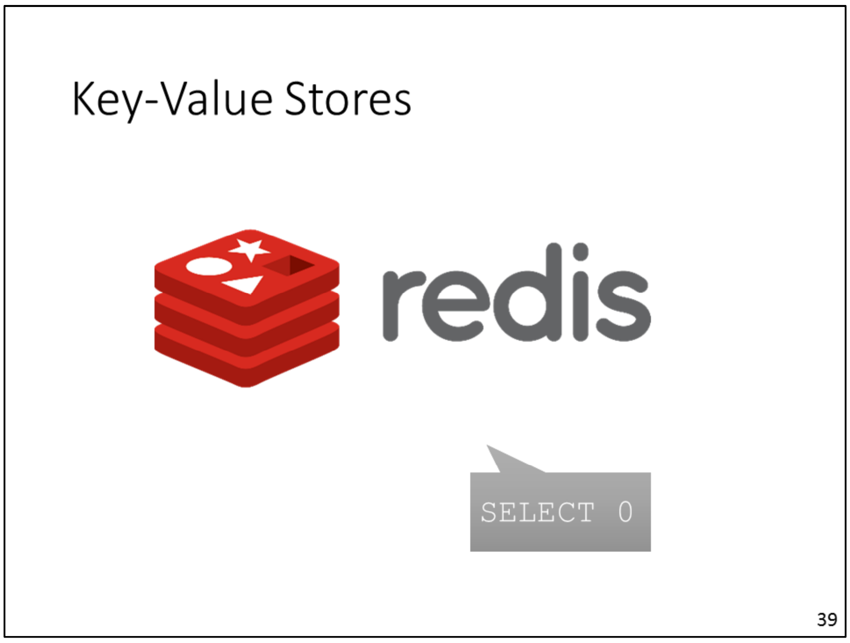
Key	Value
9	Mira

`h(key) = … mod 5`

38

38

One drawback of range partitioning: When the ID increases continuously, all the latest entries are placed on the same machine. As there usually are more accesses to recent data items than to historic data, the load on this one machine is much higher than on the others. Furthermore, repartitioning is necessary very often. As an alternative, hash partitioning can be used to store a data item on a machine based on its hash value. The hash function h(key) determines where to store the item with the given key. Example: h(key) = key % N where N=number of machines. The domain of the hash function is [0..N-1]. For a non-numeric key, it is necessary to convert it into a number first, e.g. by using the ASCII representation or using MD5 or Java‘s hashCode().



Key-Value Stores

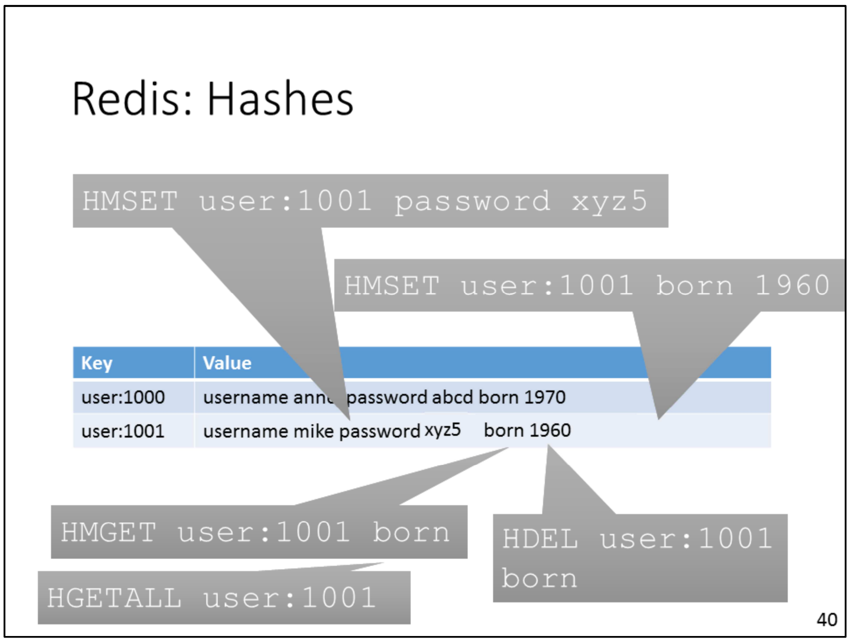
redis

```
SELECT 0
```

39

39

In Redis, SELECT is not a read command. It is used to switch to a specific database on which the following commands are executed. Databases are enumerated.



Redis: Hashes

```
HMSET user:1001 password xyz5
HMSET user:1001 born 1960
```

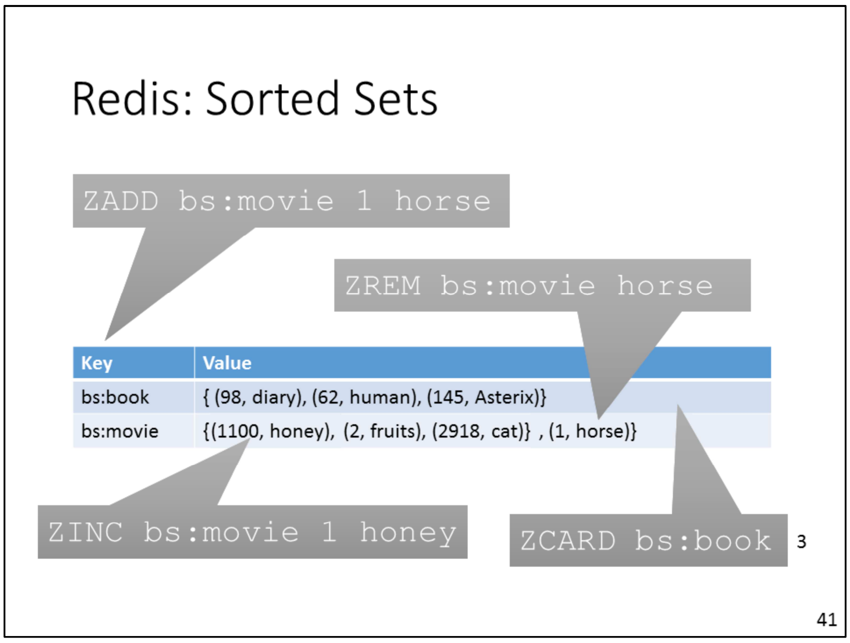
Key	Value
user:1000	username anne password abcd born 1970
user:1001	username mike password xyz5 born 1960

```
HMGET user:1001 born
HGETALL user:1001
HDEL user:1001 born
```

40

40

With hashes, the value of a key-value pair can be a list of key-value pairs. This leads to a data model similar to wide-column stores (more later). Like the SET command in Redis, HMSET has upsert behavior. This means, if a hash key is already present, its value will be overridden.



Redis: Sorted Sets

```
ZADD bs:movie 1 horse
ZREM bs:movie horse
```

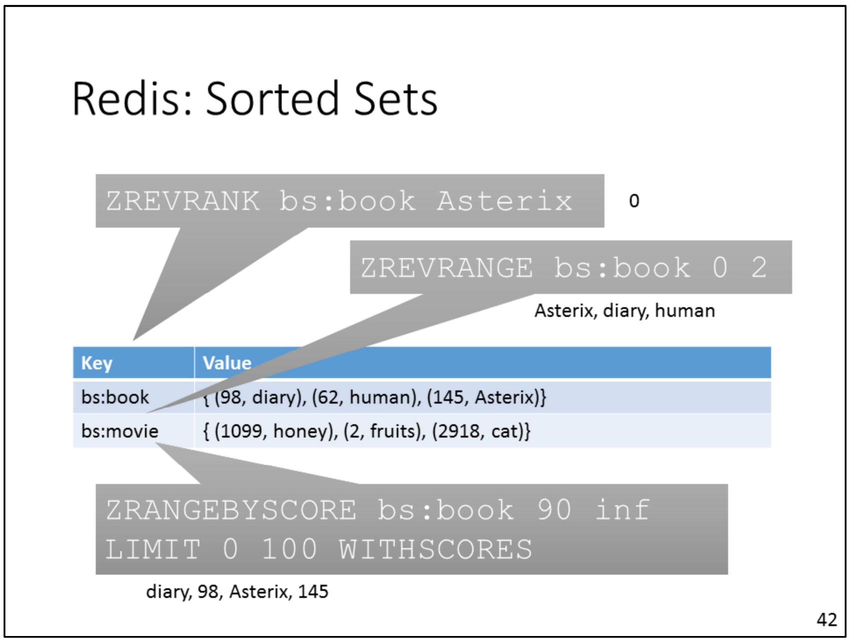
Key	Value
bs:book	{ (98, diary), (62, human), (145, Asterix)}
bs:movie	{(1100, honey), (2, fruits), (2918, cat)} , (1, horse)}

```
ZINC bs:movie 1 honey
ZCARD bs:book
```

3

41

Sorted sets are a data structure in Redis to store a set of values where each value has a score. In the table on this slide, there are categories. ZADD adds a new value into the set with the given score, ZREM removes a value, ZINC increases the value by the given number (negative values are allowed). Scores are floating point numbers.
As usual in sets, there are no duplicates and the items are unordered. The score makes sorted sets sortable. ZCARD delivers the cardinality of the set, i.e. the number of elements.



Redis: Sorted Sets

```
ZREVRANK bs:book Asterix
```

0

```
ZREVRANGE bs:book 0 2
```

Asterix, diary, human

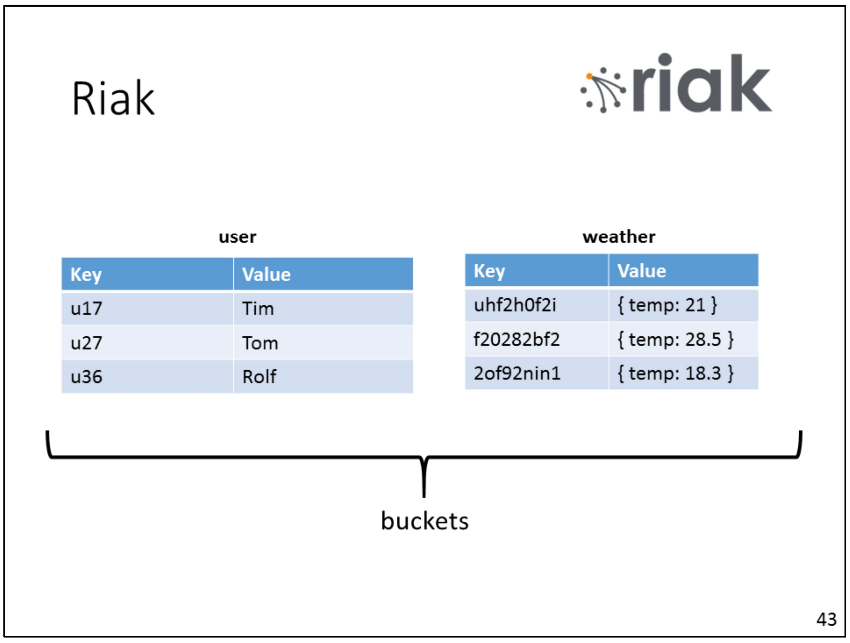
Key	Value
bs:book	{ (98, diary), (62, human), (145, Asterix)}
bs:movie	{ (1099, honey), (2, fruits), (2918, cat)}

```
ZRANGEBYSCORE bs:book 90 inf
LIMIT 0 100 WITHSCORES
```

diary, 98, Asterix, 145

42

ZREVRANK returns 0 in this example, because Asterix is the best-sold item in the book category. ZRANK delivers the rank, when the scores are sorted in ascending order, so 0 for the item with the lowest score.
The ZREVRANGE query delivers the top-3 of the books. ZRANGE would deliver the last 3.
ZRANGEBYSCORE delivers all books that have a score between 90 and infinity. In this example, the diary and Asterix (in this order).
With the limit clause, at most 100 entries are delivered, beginning from item 0 (the item with the lowest score greater or equal than 90). Remark: When ZREVRANGEBYSCORE is used, the ranges have to be swapped, e.g. inf 90 LIMIT 0 100. This finds Asterix first, then the diary.
With the option WITHSCORES the scores are part of the result.



Riak

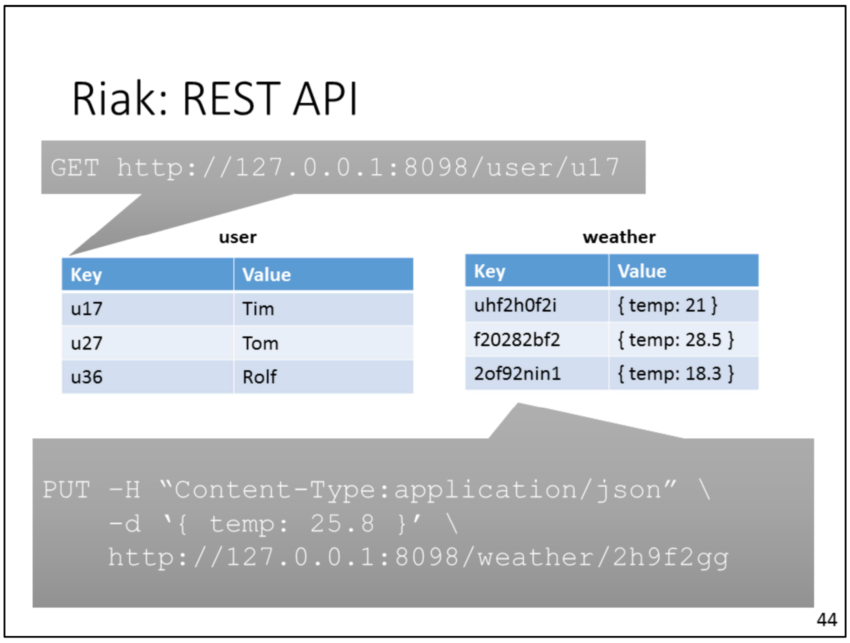
user

Key	Value
u17	Tim
u27	Tom
u36	Rolf

weather

Key	Value
uhf2h0f2i	{ temp: 21 }
f20282bf2	{ temp: 28.5 }
2of92nin1	{ temp: 18.3 }

Riak is another key-value score. It is inspired by Amazon's Dynamo (2007). Key-value pairs are saved in so-called buckets. Values can be of an arbitrary type, e.g. JSON (in the right example bucket). Different from document databases like MongoDB, there are basically no other operations possible than PUT and GET.



Riak: REST API

```
GET http://127.0.0.1:8098/user/u17
```

user

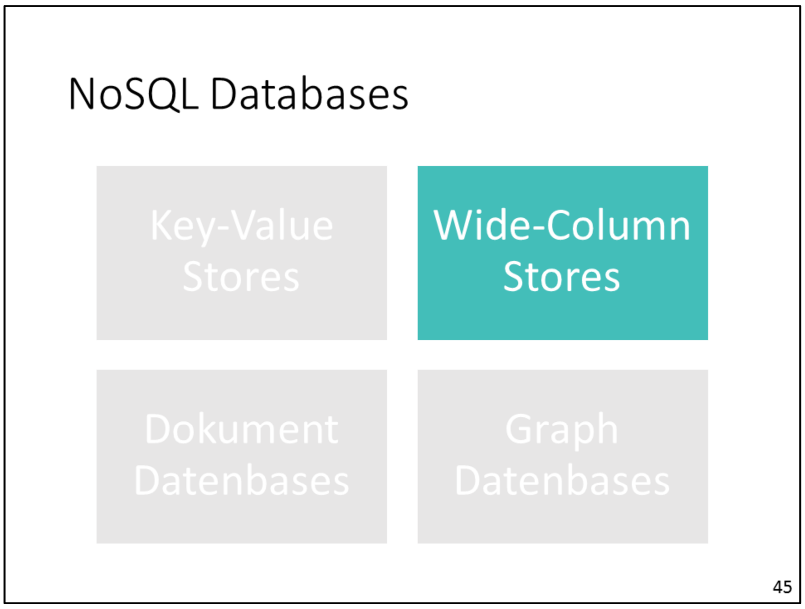
Key	Value
u17	Tim
u27	Tom
u36	Rolf

weather

Key	Value
uhf2h0f2i	{ temp: 21 }
f20282bf2	{ temp: 28.5 }
2of92nin1	{ temp: 18.3 }

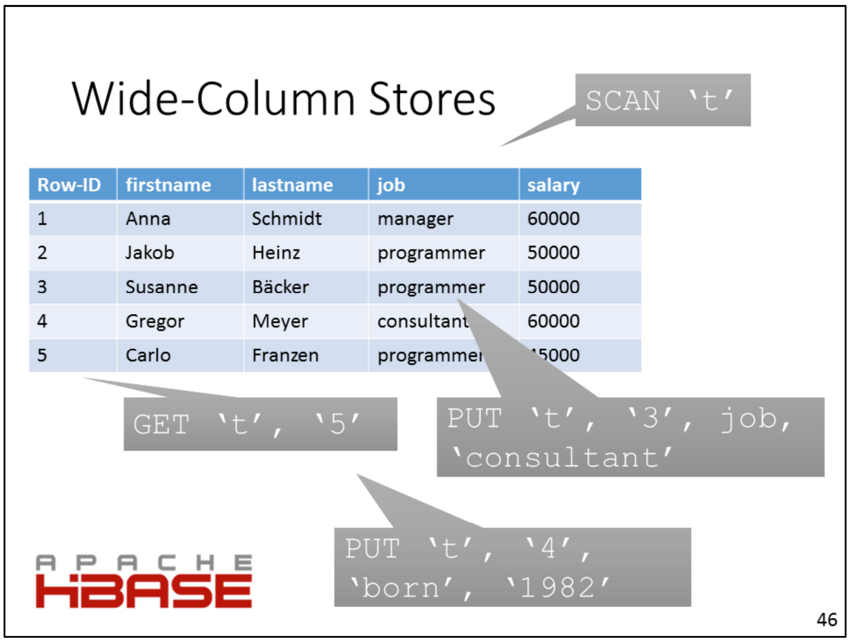
```
PUT -H "Content-Type:application/json" \
    -d '{ temp: 25.8 }' \
    http://127.0.0.1:8098/weather/2h9f2gg
```

Riak has a REST API over HTTP. This makes it possible to use Riak with any programming language without special client libraries. Furthermore, HTTP caches can be used to cache queries and their results.



NoSQL Databases

Key-Value Stores	Wide-Column Stores
Dokument Datenbases	Graph Datenbases



Wide-Column Stores

```
SCAN 't'
```

Row-ID	firstname	lastname	job	salary
1	Anna	Schmidt	manager	60000
2	Jakob	Heinz	programmer	50000
3	Susanne	Bäcker	programmer	50000
4	Gregor	Meyer	consultant	60000
5	Carlo	Franzen	programmer	45000

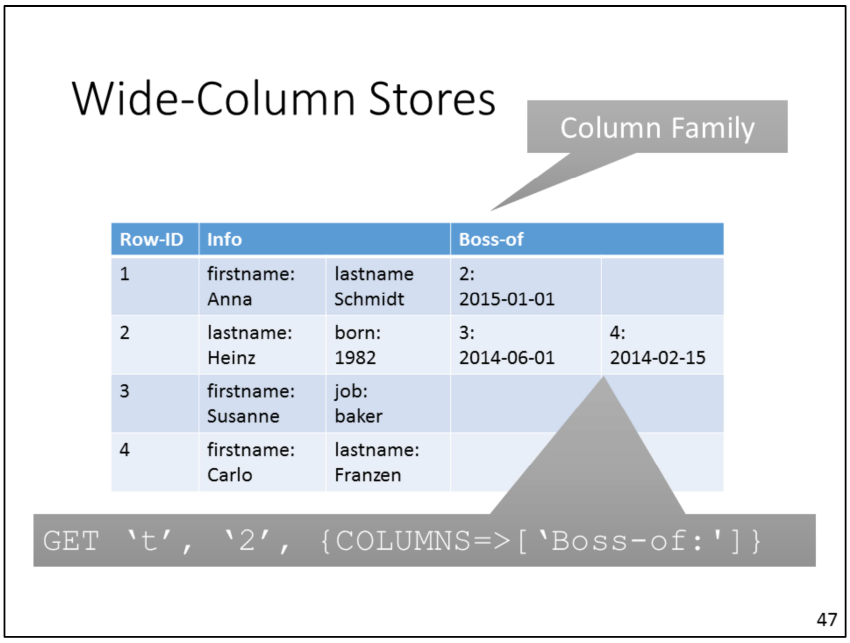
```
GET 't', '5'
```

```
PUT 't', '3', job,
'consultant'
```

```
PUT 't', '4',
'born', '1982'
```

APACHE HBASE

An HBase table consists of rows that have a flexible schema. There always is a row-id. With a PUT operation, arbitrary columns can be set. Different from relational databases, one does not need to predefine the column schema first. HBase uses range partitioning to distribute tables across multiple machines.



Wide-Column Stores

Column Family

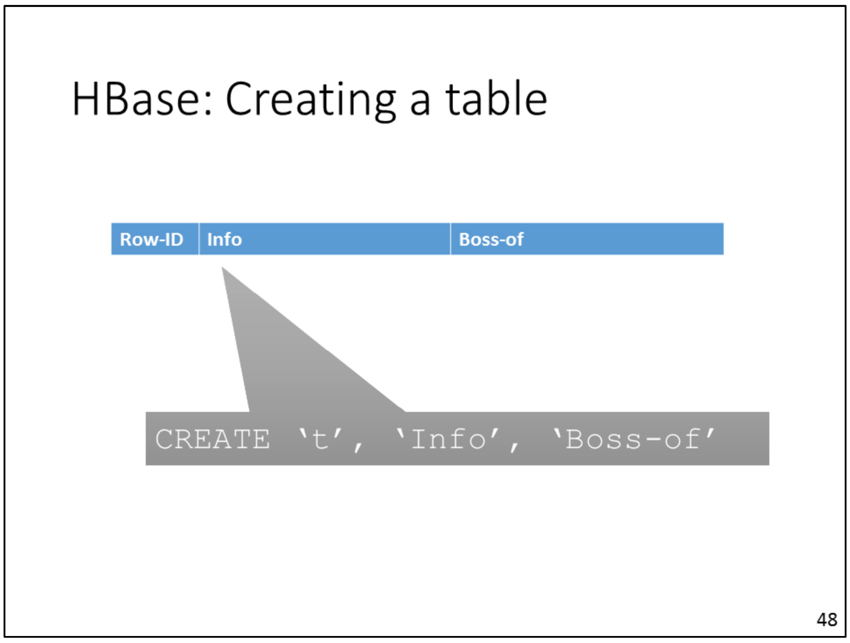
Row-ID	Info		Boss-of	
1	firstname: Anna	lastname Schmidt	2: 2015-01-01	
2	lastname: Heinz	born: 1982	3: 2014-06-01	4: 2014-02-15
3	firstname: Susanne	job: baker		
4	firstname: Carlo	lastname: Franzen		

```
GET 't', '2', {COLUMNS=>['Boss-of:']}
```

47

47

At table-creation time, column families must be set. These are used to group the columns.
The query below hides all column families except the Boss-of family.



HBase: Creating a table

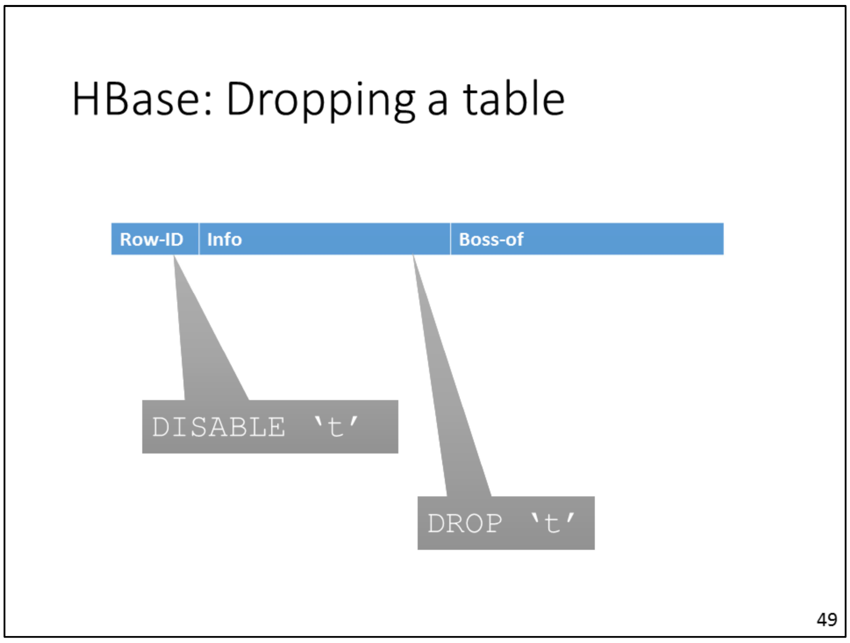
Row-ID	Info	Boss-of

```
CREATE 't', 'Info', 'Boss-of'
```

48

48

The first parameter of CREATE is the table name. The other parameters are used to define the column families.



HBase: Dropping a table

Row-ID	Info	Boss-of

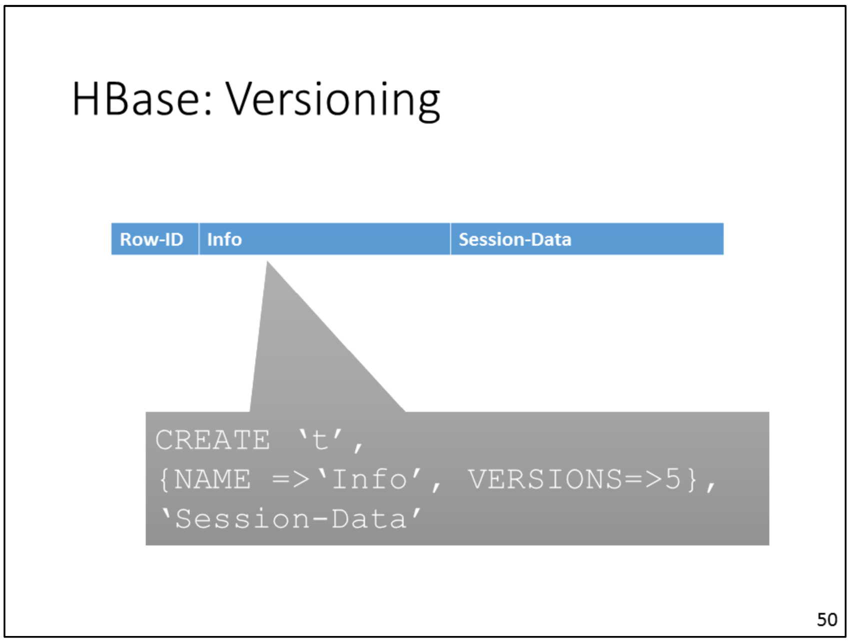
```
DISABLE 't'
```

```
DROP 't'
```

49

Before a table can be dropped, it must be disabled first. A useful command is TRUNCATE 't'. It executes DISABLE, DROP and CREATE, to drop and recreate the table with the same column families. This is the easiest way to empty a table.

49



HBase: Versioning

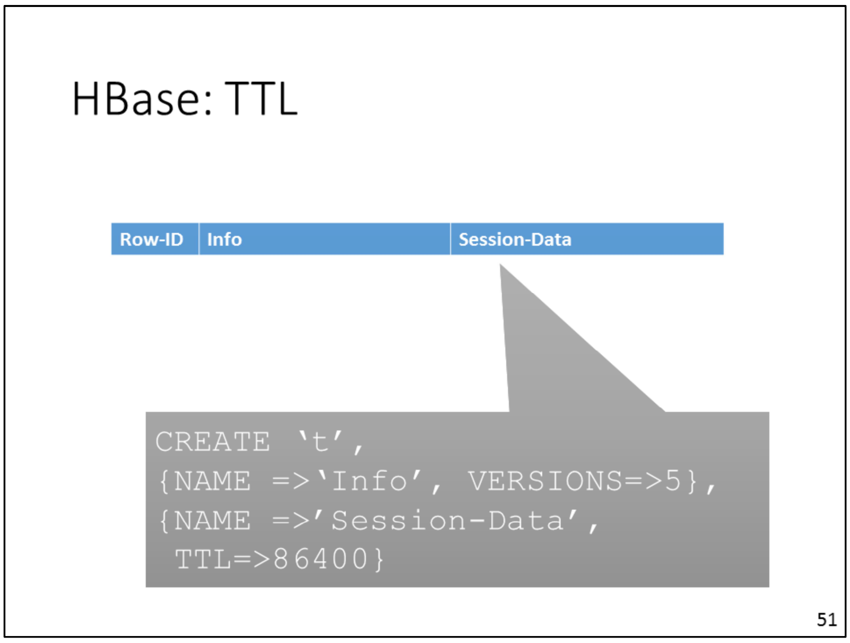
Row-ID	Info	Session-Data

```
CREATE 't',
{NAME =>'Info', VERSIONS=>5},
'Session-Data'
```

50

When column families are defined, on can set the number of value versions that HBase should keep in the columns of that family. The default value is 3. VERSION=>-1 would be an unlimited number of versions. Both in the HBase shell (console) and via the API, old versions can be accessed. By default, a read operation delivers the current version.

50



HBase: TTL

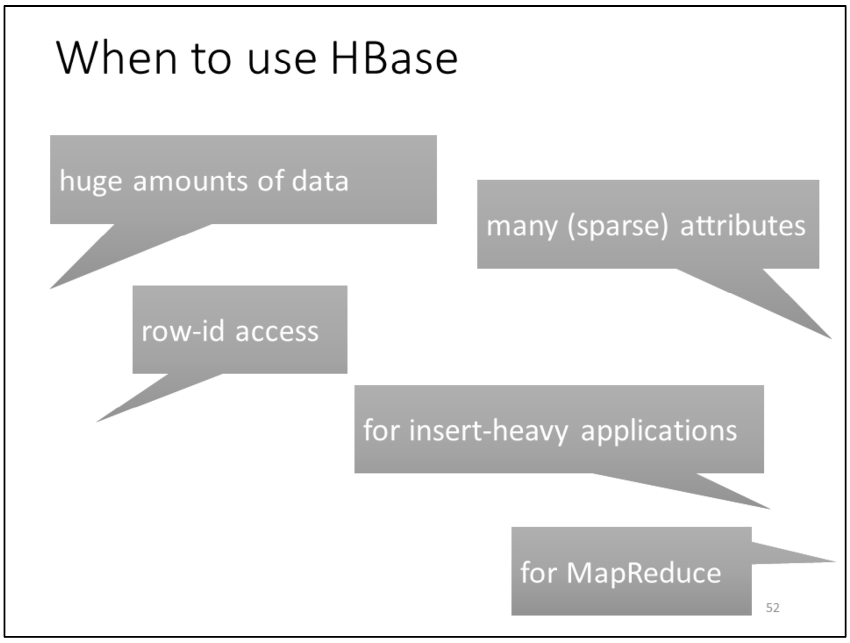
Row-ID	Info	Session-Data

```
CREATE 't',
{NAME =>'Info', VERSIONS=>5},
{NAME =>'Session-Data',
 TTL=>86400}
```

51

The column-family parameter time-to-live (TTL) defines the number of seconds after which a value in the columns in this family should be deleted automatically.

51



When to use HBase

huge amounts of data

many (sparse) attributes

row-id access

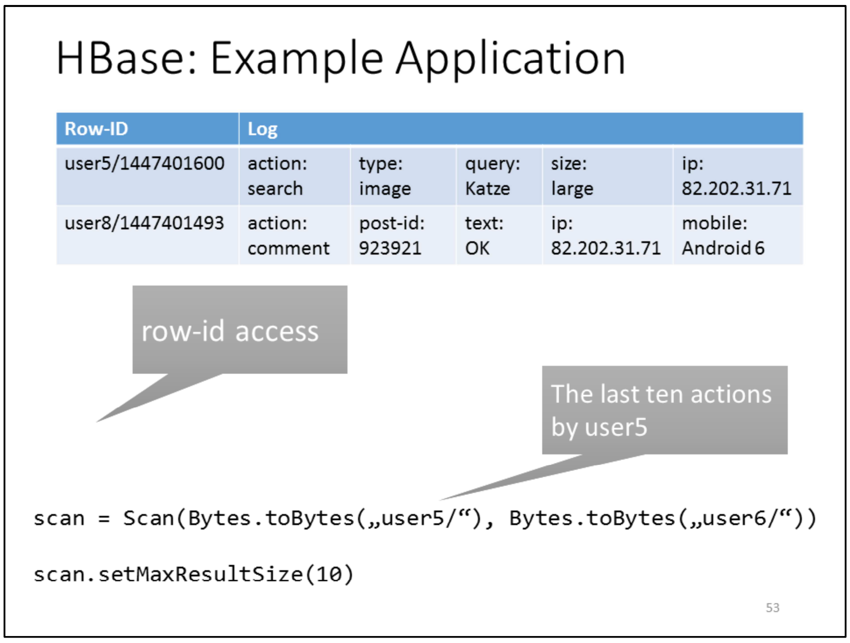
for insert-heavy applications

for MapReduce

52

HBase offers strong consistency. Atomicity is given for updates within a single row. When an application modifies multiple rows, these updates cannot be executed within one transaction. Hbase can read very fast. Also, writes are very fast, when new rows are written. Updates on existing rows are the slowest operation.

52



HBase: Example Application

Row-ID	Log				
user5/1447401600	action: search	type: image	query: Katze	size: large	ip: 82.202.31.71
user8/1447401493	action: comment	post-id: 923921	text: OK	ip: 82.202.31.71	mobile: Android 6

row-id access

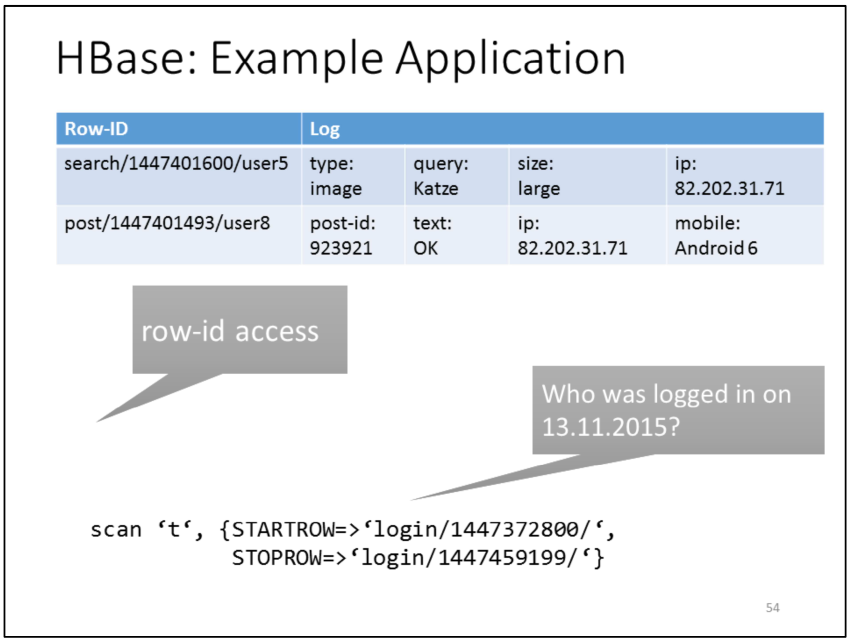
The last ten actions by user5

```
scan = Scan(Bytes.toBytes(„user5/“), Bytes.toBytes(„user6/“))

scan.setMaxResultSize(10)
```

53

53

In this example table, user actions are logged. The row-id consists of the username and a timestamp.
HBase does not have any data types except Byte arrays. That is why every value has to be converted to/from Byte[] before writing and after reading.



HBase: Example Application

Row-ID	Log			
search/1447401600/user5	type: image	query: Katze	size: large	ip: 82.202.31.71
post/1447401493/user8	post-id: 923921	text: OK	ip: 82.202.31.71	mobile: Android 6

row-id access

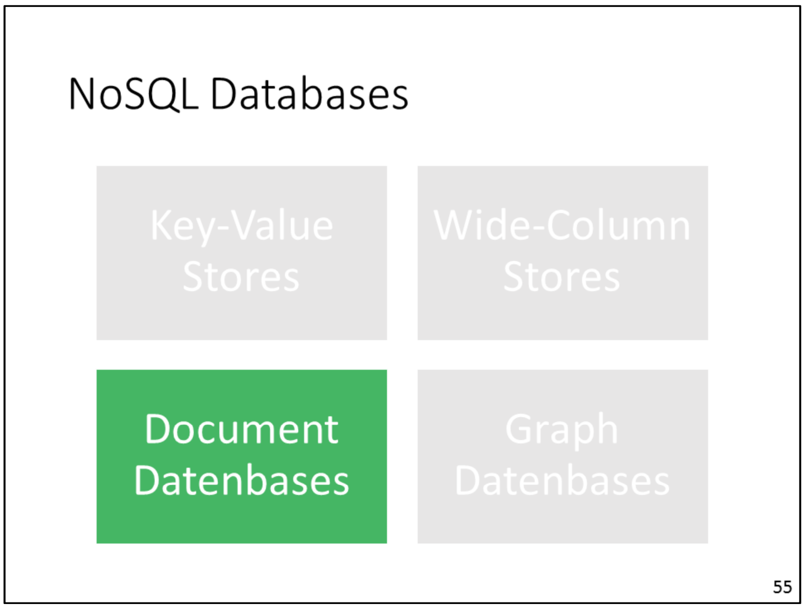
Who was logged in on 13.11.2015?

```
scan ‘t‘, {STARTROW=>‘login/1447372800/‘,
           STOPROW=>‘login/1447459199/‘}
```

54

54

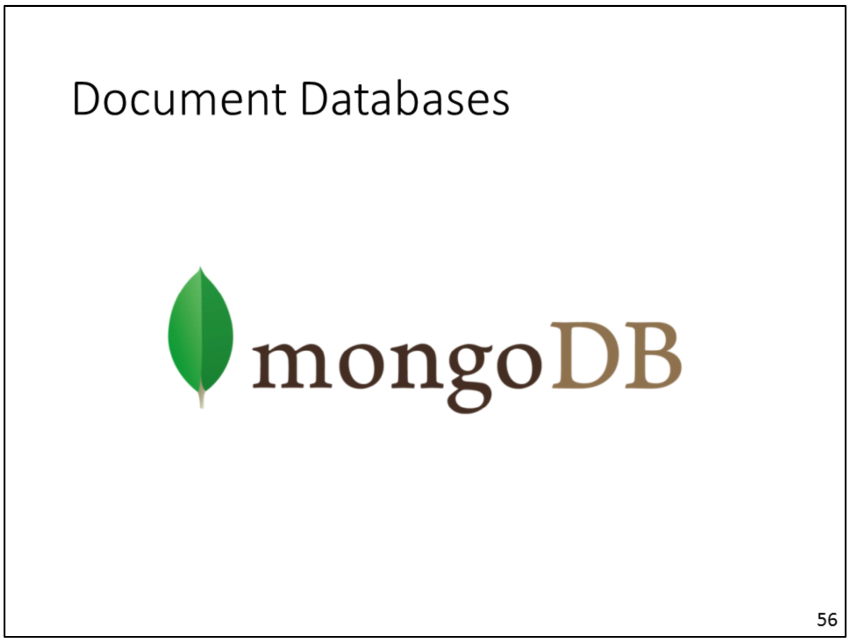
The row-id should be chosen close to the way how the data is accessed. In this table, a scan command can find all login activities within the given time range. Range queries over row-ids are very fast in HBase.



NoSQL Databases

Key-Value Stores	Wide-Column Stores
Document Datenbases	Graph Datenbases

55

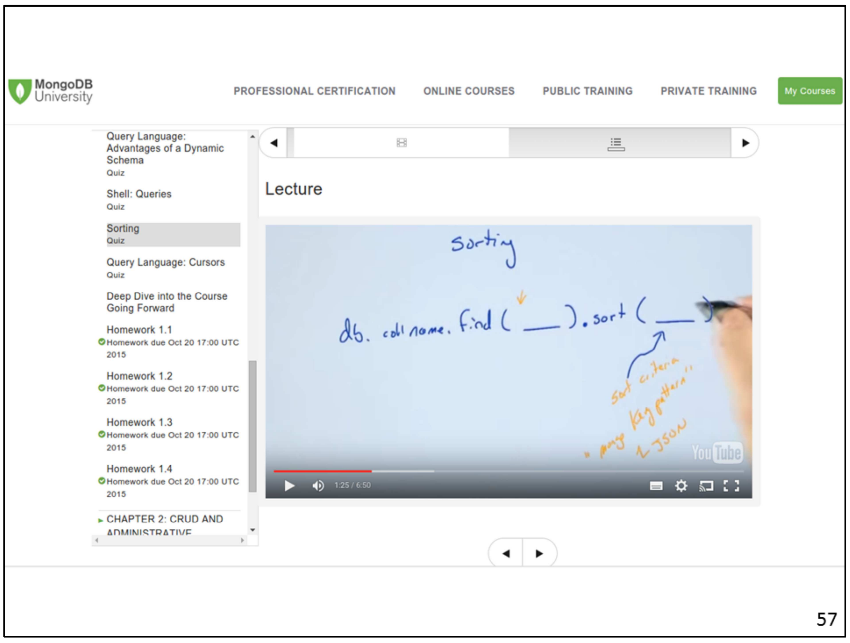


Document Databases

mongoDB

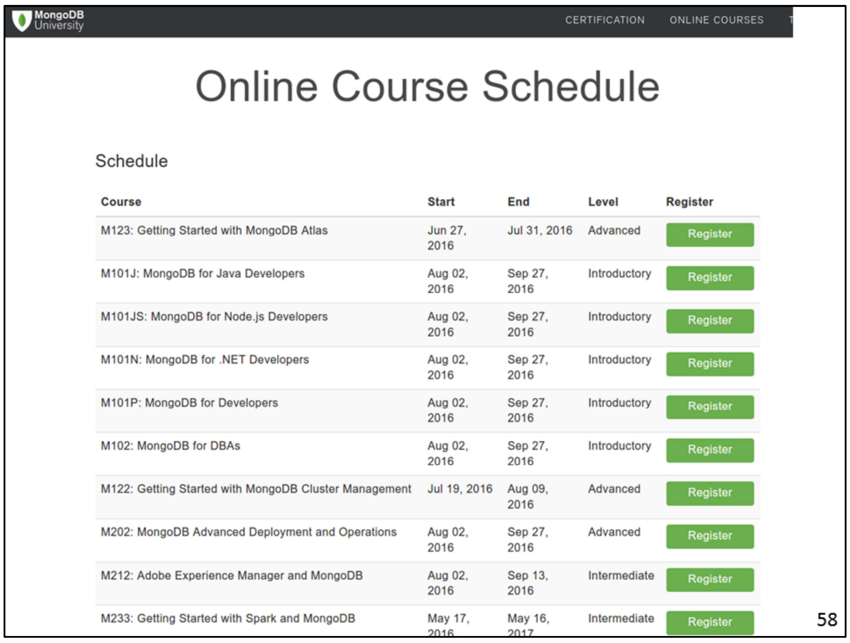
MongoDB (from „humongous“).
The most popular NoSQL database.

56


MongoDB University
PROFESSIONAL CERTIFICATION
ONLINE COURSES
PUBLIC TRAINING
PRIVATE TRAINING
My Courses
Query Language: Advantages of a Dynamic Schema
Quiz
Shell: Queries
Quiz
Sorting
Quiz
Query Language: Cursors
Quiz
Deep Dive into the Course Going Forward
Homework 1.1
Homework due Oct 20 17:00 UTC 2015
Homework 1.2
Homework due Oct 20 17:00 UTC 2015
Homework 1.3
Homework due Oct 20 17:00 UTC 2015
Homework 1.4
Homework due Oct 20 17:00 UTC 2015
CHAPTER 2: CRUD AND ADMINISTRATIVE
Lecture
Sorting
db. coll name. find (___). sort (___
sort criteria
array key pattern
JSON
1:25 / 6:50
YouTube
57

57

The MongoDB University offers free video courses to different topics. One course lasts seven weeks. In each week there are two hours of video material and simple homework which has to be submitted. After a successful participation and a final exam, you get a certificate.



Online Course Schedule

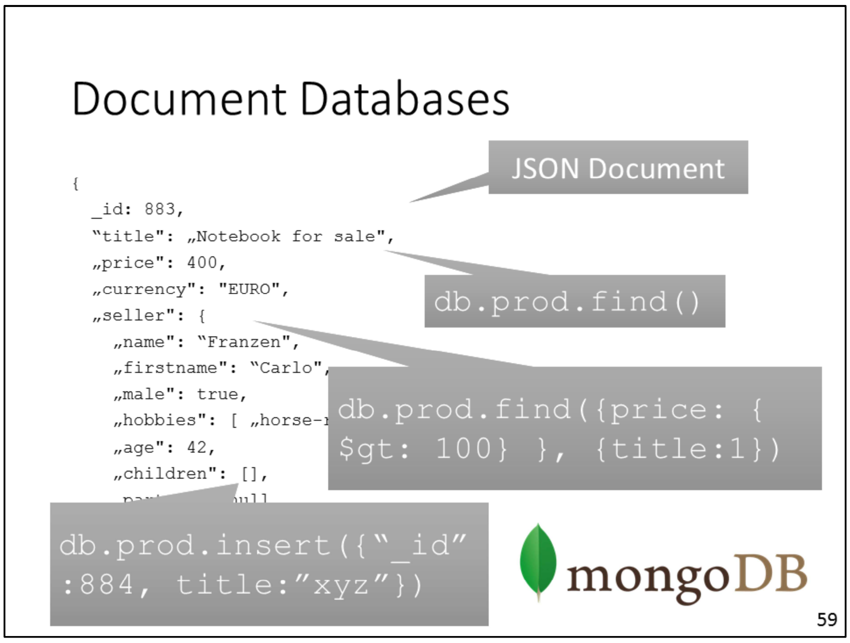
Schedule

Course	Start	End	Level	Register
M123: Getting Started with MongoDB Atlas	Jun 27, 2016	Jul 31, 2016	Advanced	Register
M101J: MongoDB for Java Developers	Aug 02, 2016	Sep 27, 2016	Introductory	Register
M101JS: MongoDB for Node.js Developers	Aug 02, 2016	Sep 27, 2016	Introductory	Register
M101N: MongoDB for .NET Developers	Aug 02, 2016	Sep 27, 2016	Introductory	Register
M101P: MongoDB for Developers	Aug 02, 2016	Sep 27, 2016	Introductory	Register
M102: MongoDB for DBAs	Aug 02, 2016	Sep 27, 2016	Introductory	Register
M122: Getting Started with MongoDB Cluster Management	Jul 19, 2016	Aug 09, 2016	Advanced	Register
M202: MongoDB Advanced Deployment and Operations	Aug 02, 2016	Sep 27, 2016	Advanced	Register
M212: Adobe Experience Manager and MongoDB	Aug 02, 2016	Sep 13, 2016	Intermediate	Register
M233: Getting Started with Spark and MongoDB	May 17, 2016	May 16, 2017	Intermediate	Register

58

58

The next courses star on 02.08.2016. Every Tuesday, new course material and new exercises are published. The course M102 is very good for the beginning. The courses M101* are also good, but there the focus is on coupling MongoDB with a programming language, here Python, Java, Node.js, .NET.



Document Databases

```
{
  _id: 883,
  "title": „Notebook for sale",
  „price": 400,
  „currency": "EURO",
  „seller": {
    „name": "Franzen",
    „firstname": "Carlo",
    „male": true,
    „hobbies": [ „horse-r
    „age": 42,
    „children": [],
```

JSON Document

```
db.prod.find()
```

```
db.prod.find({price: {
$gt: 100} }, {title:1})
```

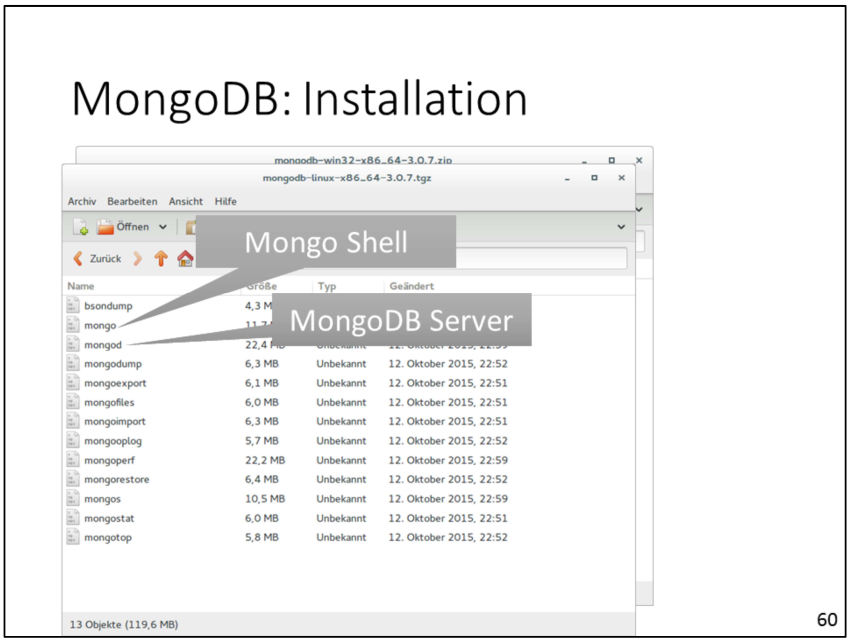
```
db.prod.insert({"_id"
:884, title:"xyz"})
```

mongoDB

59

In the document database MongoDB, the data is stored in JSON format. Every document has an unique _id.

59



MongoDB: Installation

mongodb-win32-x86_64-3.0.7.zip

mongodb-linux-x86_64-3.0.7.tgz

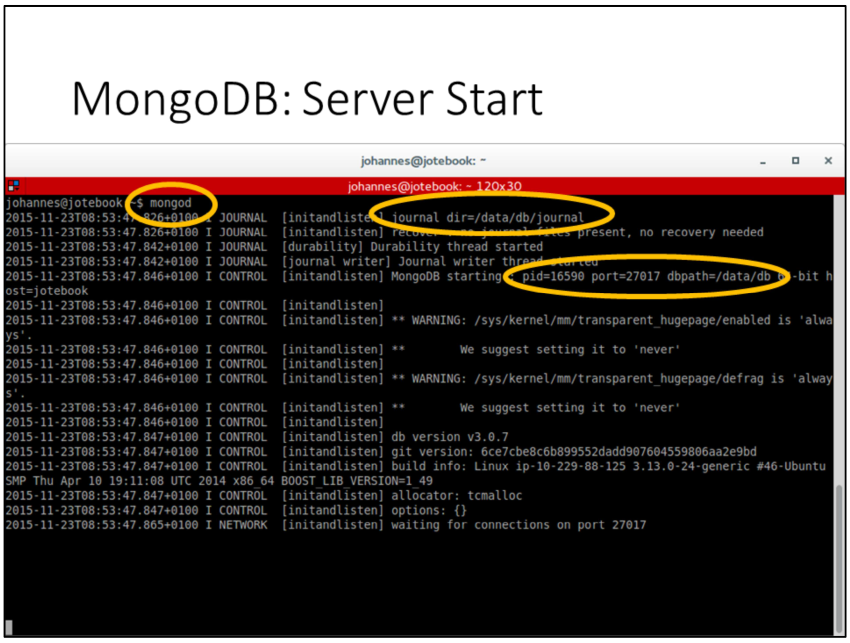
Name	Größe	Typ	Geändert
bsondump	4,3 M		
mongo	[illegible]		
mongod	22,4 MB	[illegible]	[illegible]
mongodump	6,3 MB	Unbekannt	12. Oktober 2015, 22:52
mongoexport	6,1 MB	Unbekannt	12. Oktober 2015, 22:51
mongofiles	6,0 MB	Unbekannt	12. Oktober 2015, 22:51
mongoimport	6,3 MB	Unbekannt	12. Oktober 2015, 22:51
mongooplog	5,7 MB	Unbekannt	12. Oktober 2015, 22:52
mongoperf	22,2 MB	Unbekannt	12. Oktober 2015, 22:59
mongorestore	6,4 MB	Unbekannt	12. Oktober 2015, 22:52
mongos	10,5 MB	Unbekannt	12. Oktober 2015, 22:59
mongostat	6,0 MB	Unbekannt	12. Oktober 2015, 22:51
mongotop	5,8 MB	Unbekannt	12. Oktober 2015, 22:52

13 Objekte (119,6 MB)

60

MongoDB runs on Windows, Linux and Mac. The easiest way to install is to download it as a zip file and unzip it. The program mongod starts the server process. When a mongod is running, the mongo shell (mongo) can be used to communicate with the database using JavaScript commands.

60

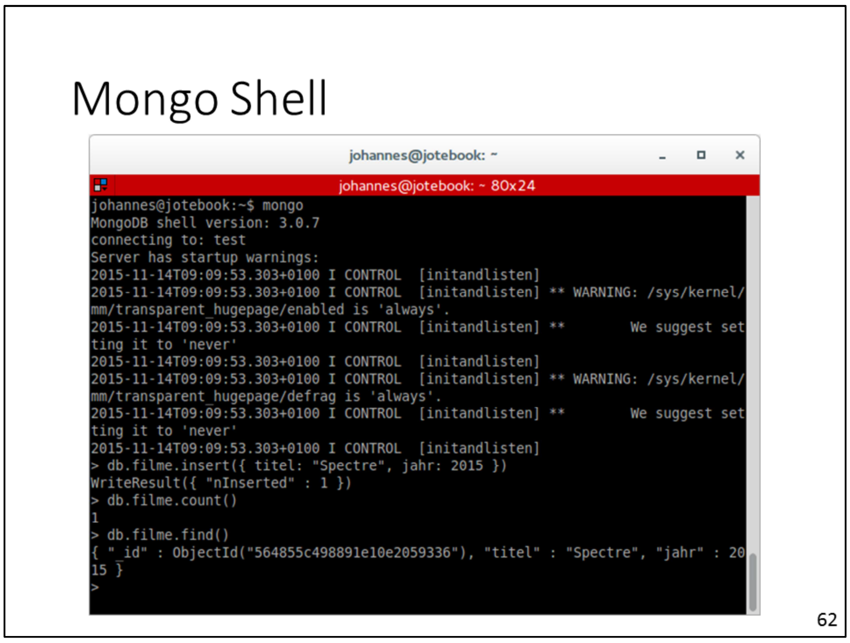


MongoDB: Server Start

```
johannes@jotebook:~$ mongod
2015-11-23T08:53:47.826+0100 I JOURNAL  [initandlisten] journal dir=/data/db/journal
2015-11-23T08:53:47.826+0100 I JOURNAL  [initandlisten] recover : no journal files present, no recovery needed
2015-11-23T08:53:47.842+0100 I JOURNAL  [durability] Durability thread started
2015-11-23T08:53:47.842+0100 I JOURNAL  [journal writer] Journal writer thread started
2015-11-23T08:53:47.846+0100 I CONTROL  [initandlisten] MongoDB starting : pid=16590 port=27017 dbpath=/data/db 64-bit h
ost=jotebook
2015-11-23T08:53:47.846+0100 I CONTROL  [initandlisten]
2015-11-23T08:53:47.846+0100 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is 'alwa
ys'.
2015-11-23T08:53:47.846+0100 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2015-11-23T08:53:47.846+0100 I CONTROL  [initandlisten]
2015-11-23T08:53:47.846+0100 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is 'alway
s'.
2015-11-23T08:53:47.846+0100 I CONTROL  [initandlisten] **        We suggest setting it to 'never'
2015-11-23T08:53:47.846+0100 I CONTROL  [initandlisten]
2015-11-23T08:53:47.847+0100 I CONTROL  [initandlisten] db version v3.0.7
2015-11-23T08:53:47.847+0100 I CONTROL  [initandlisten] git version: 6ce7cbe8c6b899552dadd907604559806aa2e9bd
2015-11-23T08:53:47.847+0100 I CONTROL  [initandlisten] build info: Linux ip-10-229-88-125 3.13.0-24-generic #46-Ubuntu
SMP Thu Apr 10 19:11:08 UTC 2014 x86_64 BOOST_LIB_VERSION=1_49
2015-11-23T08:53:47.847+0100 I CONTROL  [initandlisten] allocator: tcmalloc
2015-11-23T08:53:47.847+0100 I CONTROL  [initandlisten] options: {}
2015-11-23T08:53:47.865+0100 I NETWORK  [initandlisten] waiting for connections on port 27017
```

The MongoDB server is started using mongod. The console output lists useful information such as the server storing the data (/data/db) and the log destination (/data/db/journal). Moreover, the log output contains the process ID (in case you need to kill it) and the port number for MongoDB.

Parameters can be set manually when starting the sever: mongod --port 27017 --dbpath /data/db --logpath /data/db/journal
Using the --fork parameter will start the process in the background. Without this parameter, mongod will end when the terminal window is closed (e.g., with Ctrl+c).

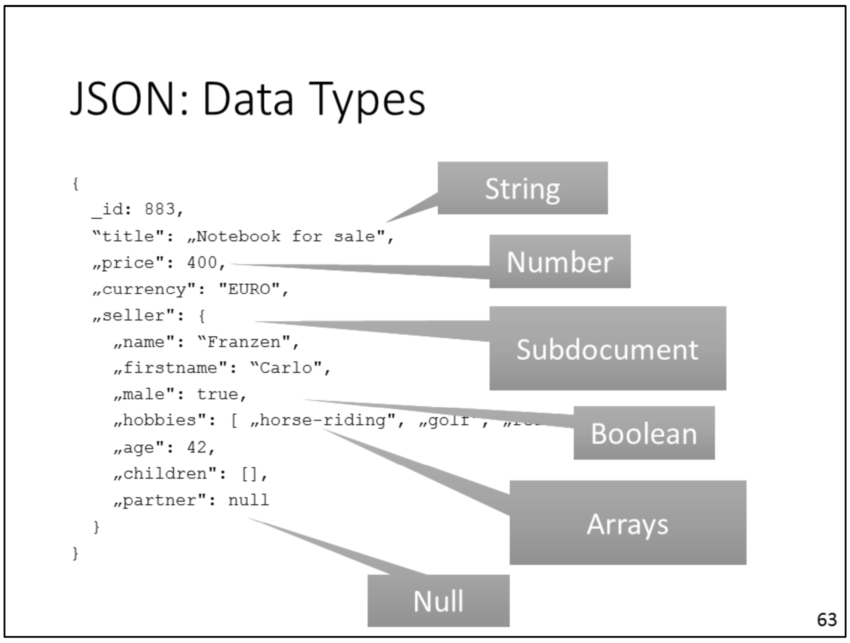


Mongo Shell

```
johannes@jotebook:~$ mongo
MongoDB shell version: 3.0.7
connecting to: test
Server has startup warnings:
2015-11-14T09:09:53.303+0100 I CONTROL  [initandlisten]
2015-11-14T09:09:53.303+0100 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
mm/transparent_hugepage/enabled is 'always'.
2015-11-14T09:09:53.303+0100 I CONTROL  [initandlisten] **        We suggest set
ting it to 'never'
2015-11-14T09:09:53.303+0100 I CONTROL  [initandlisten]
2015-11-14T09:09:53.303+0100 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
mm/transparent_hugepage/defrag is 'always'.
2015-11-14T09:09:53.303+0100 I CONTROL  [initandlisten] **        We suggest set
ting it to 'never'
2015-11-14T09:09:53.303+0100 I CONTROL  [initandlisten]
> db.filme.insert({ titel: "Spectre", jahr: 2015 })
WriteResult({ "nInserted" : 1 })
> db.filme.count()
1
> db.filme.find()
{ "_id" : ObjectId("564855c498891e10e2059336"), "titel" : "Spectre", "jahr" : 20
15 }
>
```

62

The shell can be started with the command mongo. The information “connecting to test” shows that the shell is connected with the database test. With “use differentdb”, one can switch to another database.



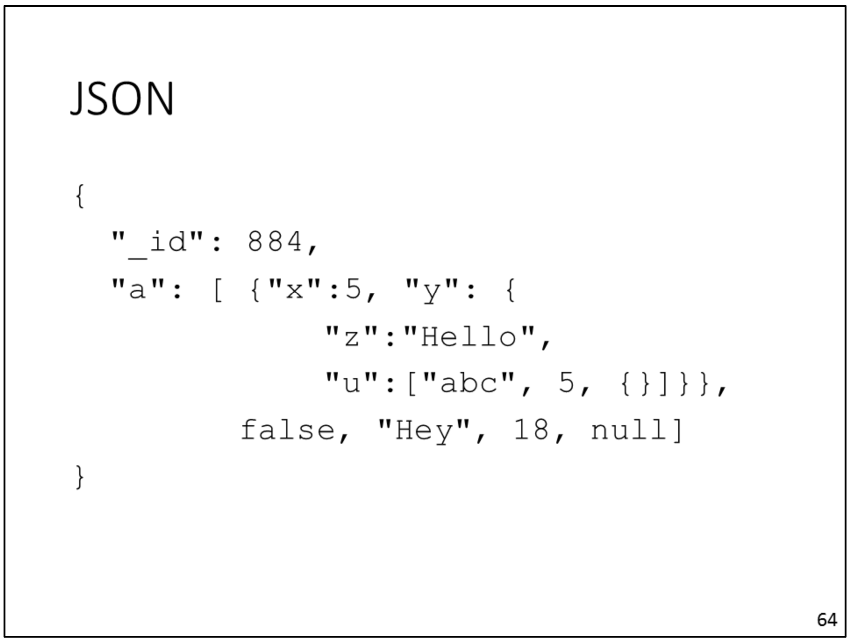
JSON: Data Types

```
{
  _id: 883,
  “title": „Notebook for sale",
  „price": 400,
  „currency": "EURO",
  „seller": {
    „name": “Franzen",
    „firstname": “Carlo",
    „male": true,
    „hobbies": [ „horse-riding", „golf", „
    „age": 42,
    „children": [],
    „partner": null
  }
}
```

String
Number
Subdocument
Boolean
Arrays
Null

Here are six JSON datatypes. A JSON field consists of a field name (e.g., “currency”) and a value (e.g., “EURO”). The field name (the part left of the colon) is always a string and is therefore written in quotation marks. When using JavaScript (e.g. in the mongo shell), these quotation marks are optional for names without spaces, dots and other special characters. The field value (right of the colon) can be a number (integer, float, …), a null value or a Boolean (true or false) without quotation marks. Everything in quotation marks is a string. With { }, sub-documents can be nested. Arrays are written in [“box”, “brackets”]. Field names are unique within one document, but reuse in sub-documents is allowed.

63

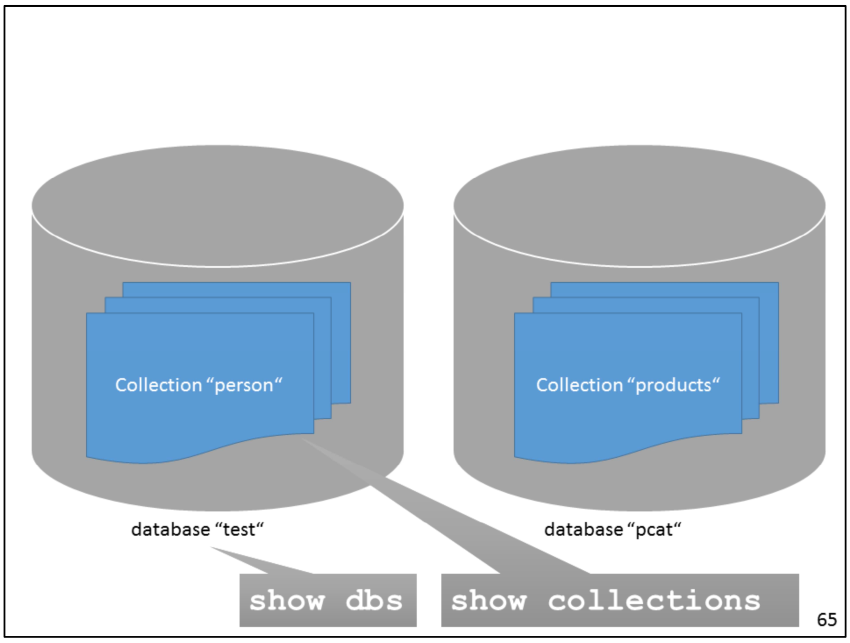


JSON

```
{
  "_id": 884,
  "a": [ {"x":5, "y": {
              "z":"Hello",
              "u":["abc", 5, {}]}},
        false, "Hey", 18, null]
}
```

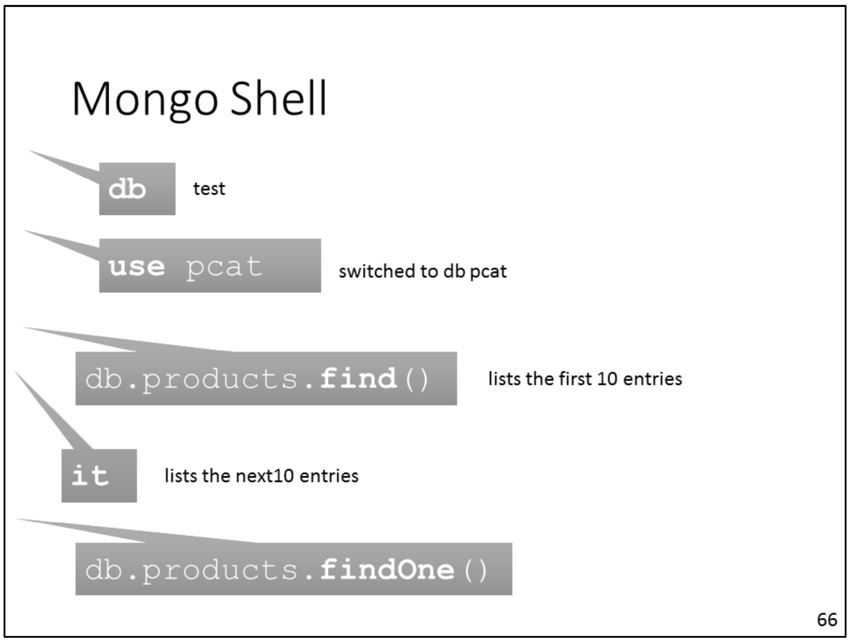
This is a valid JSON document. An arbitrary nesting and using heterogeneous types is allowed.

64



```
show dbs
show collections
```

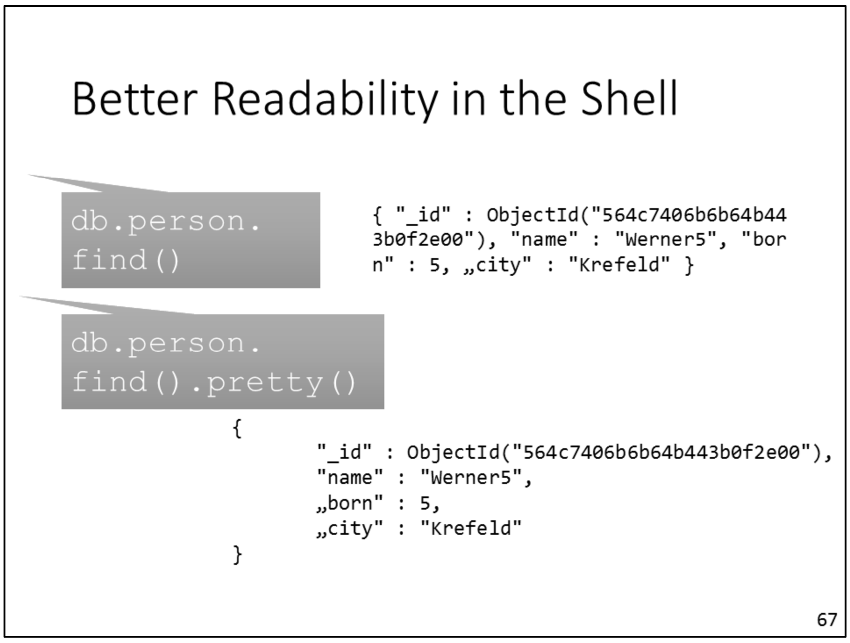
A MongoDB instance consists of multiple databases. “show dbs” shows the list of databases. Within one database, there are many collections. A list of collection names can be printed with “show collections”.



Mongo Shell

```
db                      test
use pcat                switched to db pcat
db.products.find()      lists the first 10 entries
it                      lists the next10 entries
db.products.findOne()
```

„db“ shows the name of the current database. With “use”, one can switch to another database. A find() command opens a cursor, which is used in an application by iterating over it with next() commands. In the shell, the first ten documents are printed and the cursor remains opened, so that one can fetch the next ten elements with the “it” (“iterate”) command.
“fineOne()” is a useful command to show how typical documents within one collection look like. It just delivers one document.



Better Readability in the Shell

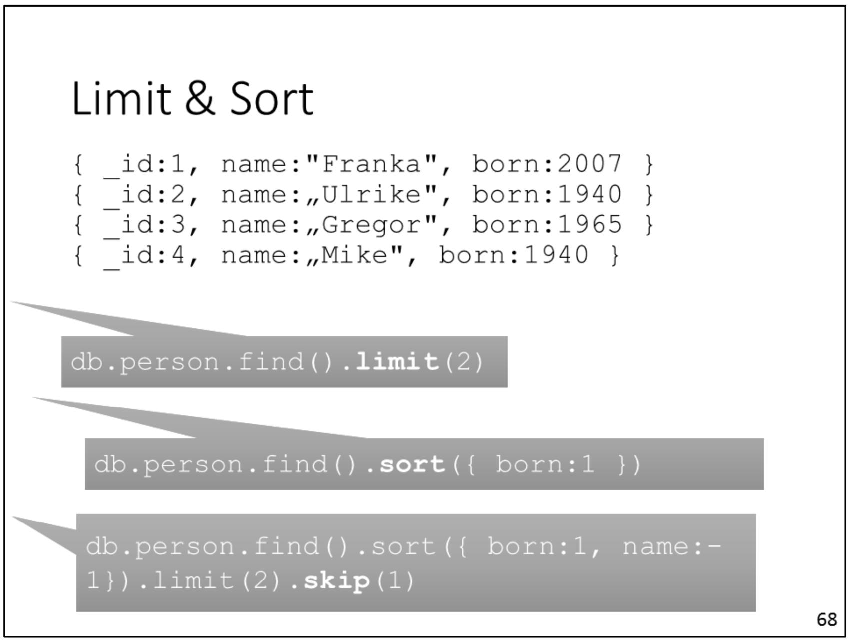
```
db.person.
find()
```

```
{ "_id" : ObjectId("564c7406b6b64b44
3b0f2e00"), "name" : "Werner5", "bor
n" : 5, „city" : "Krefeld" }
```

```
db.person.
find().pretty()
```

```
{
        "_id" : ObjectId("564c7406b6b64b443b0f2e00"),
        "name" : "Werner5",
        „born" : 5,
        „city" : "Krefeld"
}
```

67



Limit & Sort

```
{ _id:1, name:"Franka", born:2007 }
{ _id:2, name:„Ulrike", born:1940 }
{ _id:3, name:„Gregor", born:1965 }
{ _id:4, name:„Mike", born:1940 }
```

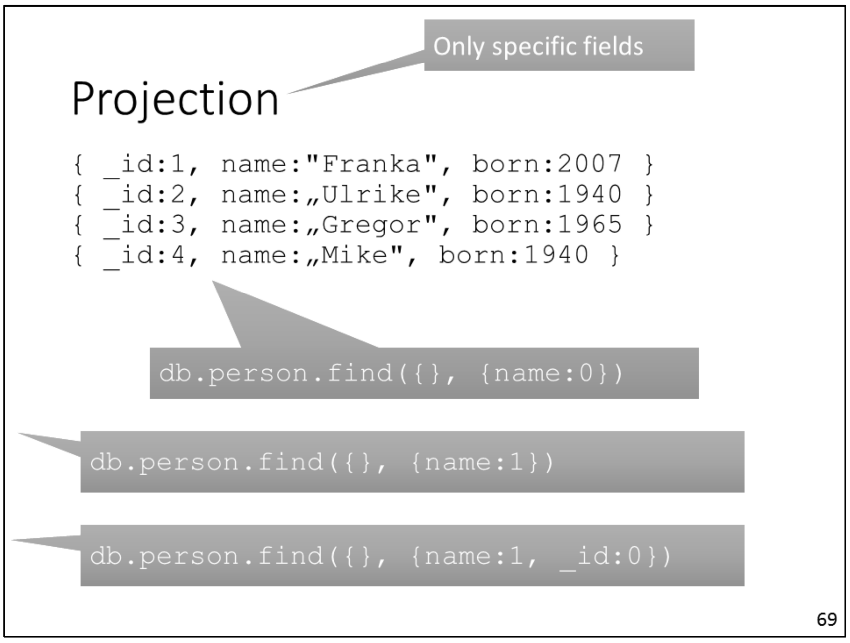
```
db.person.find().limit(2)
```

```
db.person.find().sort({ born:1 })
```

```
db.person.find().sort({ born:1, name:-
1}).limit(2).skip(1)
```

68

.limit limits the number of result documents. .sort sorts the result by one or more sort criteria. Except .limit and .skip, nearly every method in MongoDB takes a JSON document as a parameter. For sort, the key is the field name for the ordering and the value is the direction: 1 (ascending), -1 (descending).
When .skip is used together with limit, the given number of documents are skipped.
In the third query on this slide, the documents are ordered by born ascending and then - if two documents have the same born value – name descending. So Ulrike would be before Mike. Due to the skip, Ulrike is skipped and Mike and Gregor are in the result.



Projection

Only specific fields

```
{ _id:1, name:"Franka", born:2007 }
{ _id:2, name:„Ulrike", born:1940 }
{ _id:3, name:„Gregor", born:1965 }
{ _id:4, name:„Mike", born:1940 }
```

```
db.person.find({}, {name:0})
```

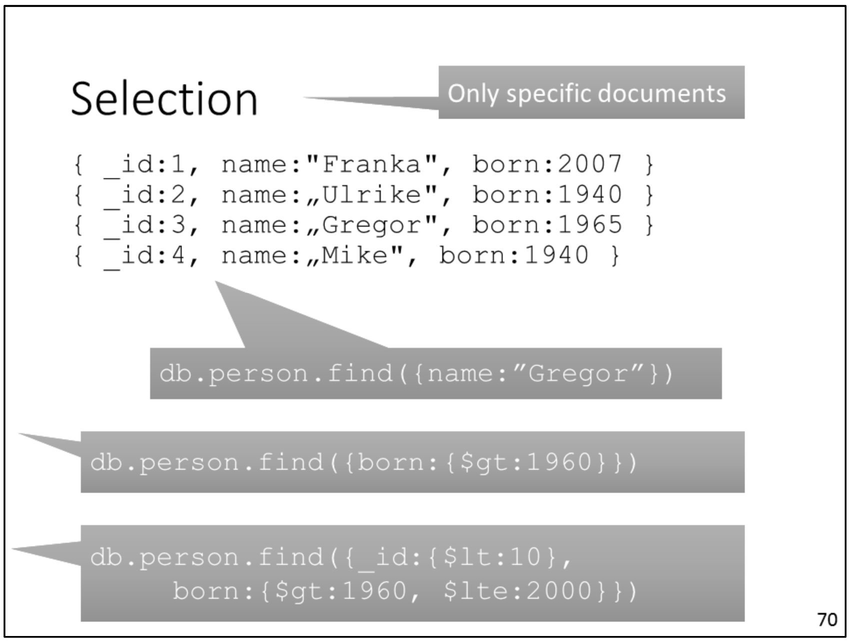
```
db.person.find({}, {name:1})
```

```
db.person.find({}, {name:1, _id:0})
```

69

The first parameter of find is a selection (see next slide). The second parameter can be used for a projection, i.e. defining which field should be in the result, and which not. The projectionargument is a JSON document. A 1 stands for including the field, 0 for excluding it. The first query gives all fields but name (here: _id and born). The _id is always part of the result, unless it is explicitly excluded. The second query shoes the name and _id, the third query only the name.

69



Selection

Only specific documents

```
{ _id:1, name:"Franka", born:2007 }
{ _id:2, name:„Ulrike", born:1940 }
{ _id:3, name:„Gregor", born:1965 }
{ _id:4, name:„Mike", born:1940 }
```

```
db.person.find({name:”Gregor”})
```

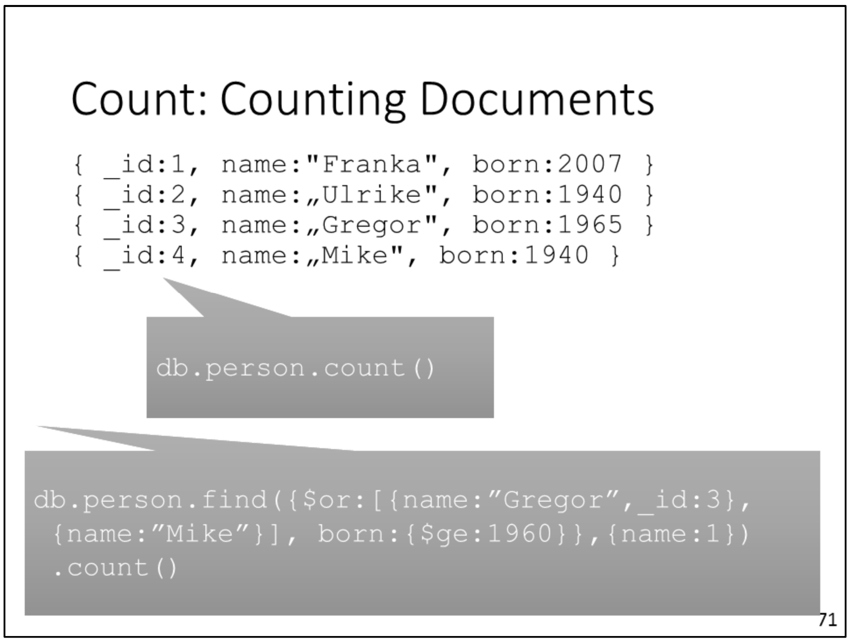
```
db.person.find({born:{$gt:1960}})
```

```
db.person.find({_id:{$lt:10},
    born:{$gt:1960, $lte:2000}})
```

70

All parameters of find are optional. The first one is for the selection. The queries have a query-by-example pattern. The given selection JSON document shows how the result document should look like. For comparisons different from equality, {$operator:value} must be used. The second query finds all people having a year of birth greater than 1960. Other comparison operators are $gte (greater or equal), $lt (less than), $lte, $ne (not equal). When multiple selections are given, they are and-connected. So, a document has to fulfil all of them. $or can be used with an array of criteria to make an or-connection (see next slide).

70



Count: Counting Documents

```
{ _id:1, name:"Franka", born:2007 }
{ _id:2, name:„Ulrike", born:1940 }
{ _id:3, name:„Gregor", born:1965 }
{ _id:4, name:„Mike", born:1940 }
```

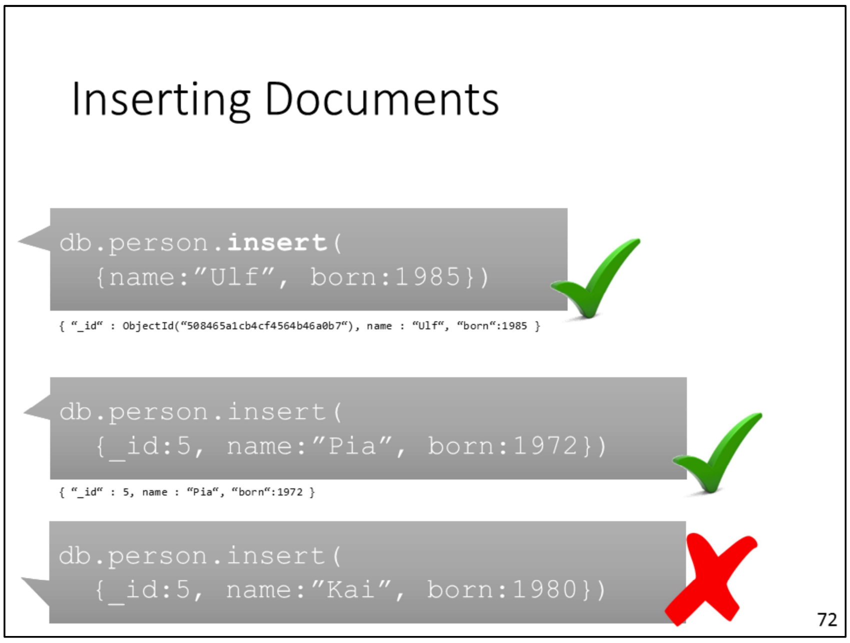
```
db.person.count()
```

```
db.person.find({$or:[{name:"Gregor",_id:3},
 {name:"Mike"}], born:{$ge:1960}},{name:1})
 .count()
```

71

count() can be called on a collection or a query (cursor). It shows the number of documents.

71



Inserting Documents

```
db.person.insert(
  {name:"Ulf", born:1985})
```

```
{ "_id" : ObjectId("508465a1cb4cf4564b46a0b7"), name : "Ulf", "born":1985 }
```

✓

```
db.person.insert(
  {_id:5, name:"Pia", born:1972})
```

```
{ "_id" : 5, name : "Pia", "born":1972 }
```

✓

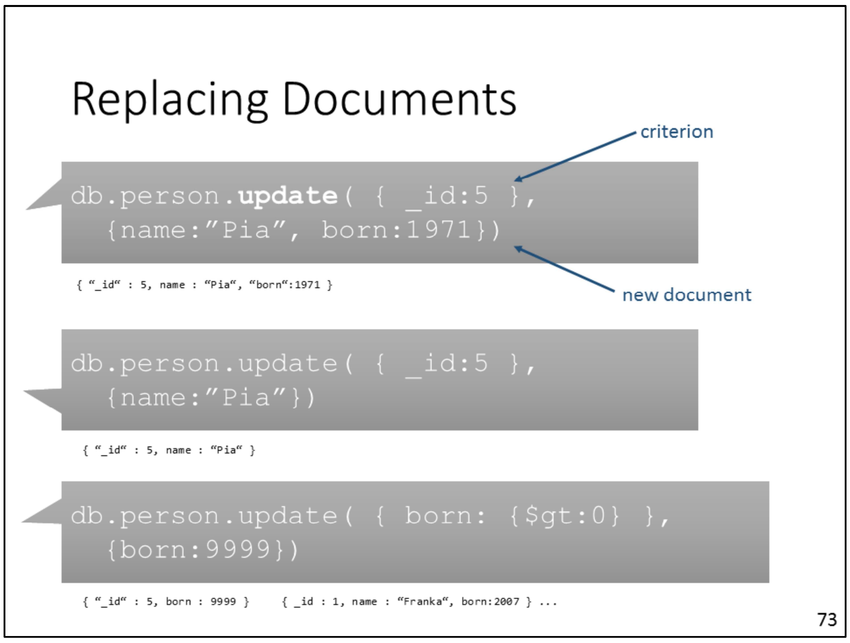
```
db.person.insert(
  {_id:5, name:"Kai", born:1980})
```

✗

72

When an insert does not specify the _id field, it is automatically set with an unique ObjectId. When the _id is set manually, it must be unique within the given collection, otherwise the insert will fail.

72



Replacing Documents

```
db.person.update( { _id:5 },
  {name:"Pia", born:1971})
```

criterion
new document

```
{ "_id" : 5, name : "Pia", "born":1971 }
```

```
db.person.update( { _id:5 },
  {name:"Pia"})
```

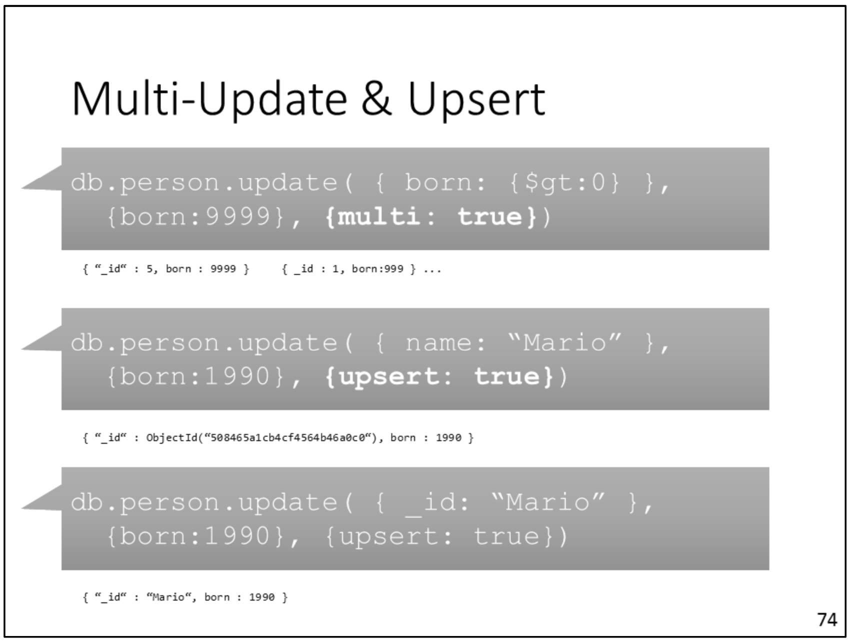
```
{ "_id" : 5, name : "Pia" }
```

```
db.person.update( { born: {$gt:0} },
  {born:9999})
```

```
{ "_id" : 5, born : 9999 }    { _id : 1, name : "Franka", born:2007 } ...
```

73

The update command takes two parameters: The first is a criteria which documents to update (like in find()). The second parameter is a new document which will completely replace the selected document. In the new document, the _id must not change. It is the only field that remains unchanged after the update.
Remark: The update command only updates at most one single document. When the criteria is fulfilled for multiple documents, only one document is changed.



Multi-Update & Upsert

```
db.person.update( { born: {$gt:0} },
  {born:9999}, {multi: true})
```

```
{ "_id" : 5, born : 9999 }    { _id : 1, born:999 } ...
```

```
db.person.update( { name: "Mario" },
  {born:1990}, {upsert: true})
```

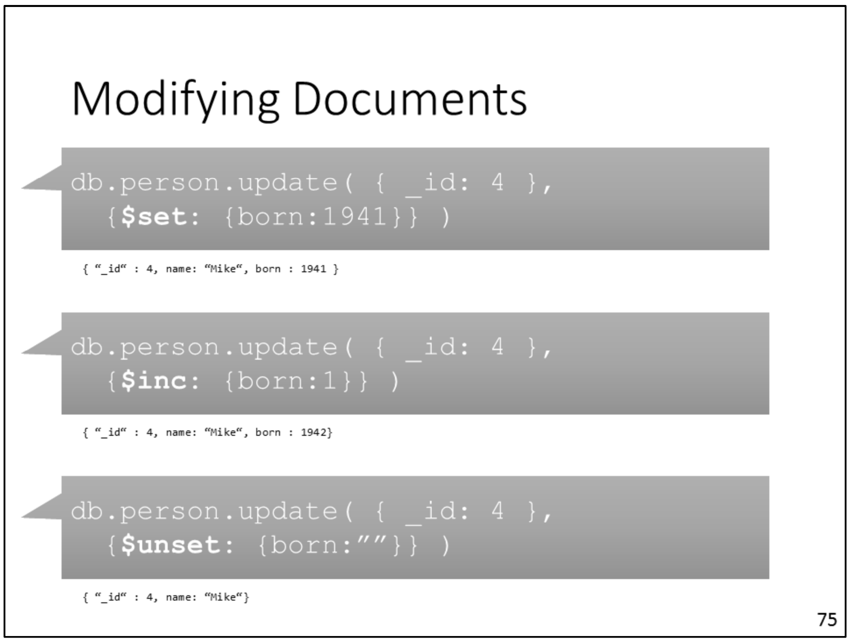
```
{ "_id" : ObjectId("508465a1cb4cf4564b46a0c0"), born : 1990 }
```

```
db.person.update( { _id: "Mario" },
  {born:1990}, {upsert: true})
```

```
{ "_id" : "Mario", born : 1990 }
```

74

The optional third parameter of update is for options. The option multi:true allows for updating multiple documents. The update is executed on every document fulfilling the predicate (here: born>0).
The upsert:true option is a combination of insert and update. If no document fulfills the update criterion, the given document will be inserted. Otherwise, the matching document will be updated. If the criterion contains an _id, the inserted document will have this _id. Other criteria do not have any influence on the new document.



Modifying Documents

```
db.person.update( { _id: 4 },
  {$set: {born:1941}} )
```

```
{ "_id" : 4, name: "Mike", born : 1941 }
```

```
db.person.update( { _id: 4 },
  {$inc: {born:1}} )
```

```
{ "_id" : 4, name: "Mike", born : 1942}
```

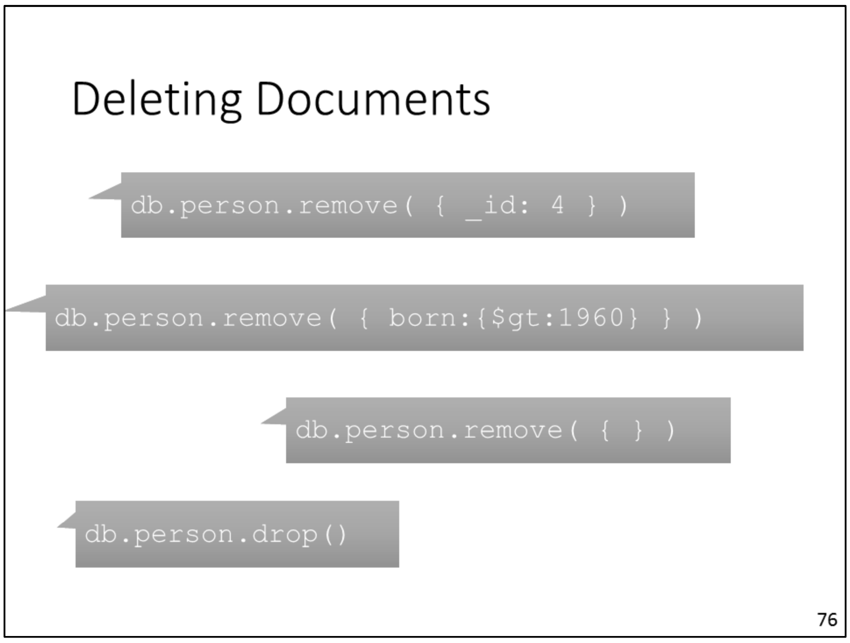
```
db.person.update( { _id: 4 },
  {$unset: {born:""}} )
```

```
{ "_id" : 4, name: "Mike"}
```

75

With $set : { field : value, field : value }, fields can be changed in an existing document. When the field is not yet existing, it will be added. Other fields are not influenced by this update. Options like multi and upserts are possible here, too.
$inc increases the value by the given number. If the field does not exist, it will be initialized with the increment value (here: 1). Negative increment values are possible. $uset removes the field from the document.
Other modifiers like $mul, $rename, $setOnInsert are possible. See the Mongo DB documentation.

75



Deleting Documents

```
db.person.remove( { _id: 4 } )
```

```
db.person.remove( { born:{$gt:1960} } )
```

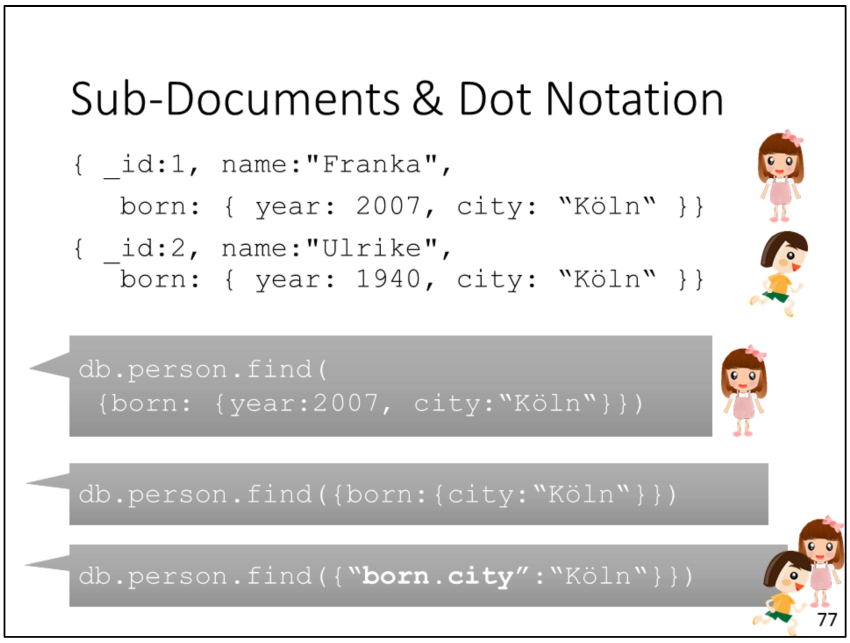
```
db.person.remove( { } )
```

```
db.person.drop()
```

76

The remove command deletes all documents matching the given criterion. The first query removes the person with _id 4, the second query deletes all people, which are born after 1960, the third query deletes all people. The difference between remove({}) and drop is the following: Remove deletes one document after the other. Drop is much faster.

76



Sub-Documents & Dot Notation

```
{ _id:1, name:"Franka",
   born: { year: 2007, city: "Köln" }}
{ _id:2, name:"Ulrike",
   born: { year: 1940, city: "Köln" }}
```

```
db.person.find(
 {born: {year:2007, city:"Köln"}})
```

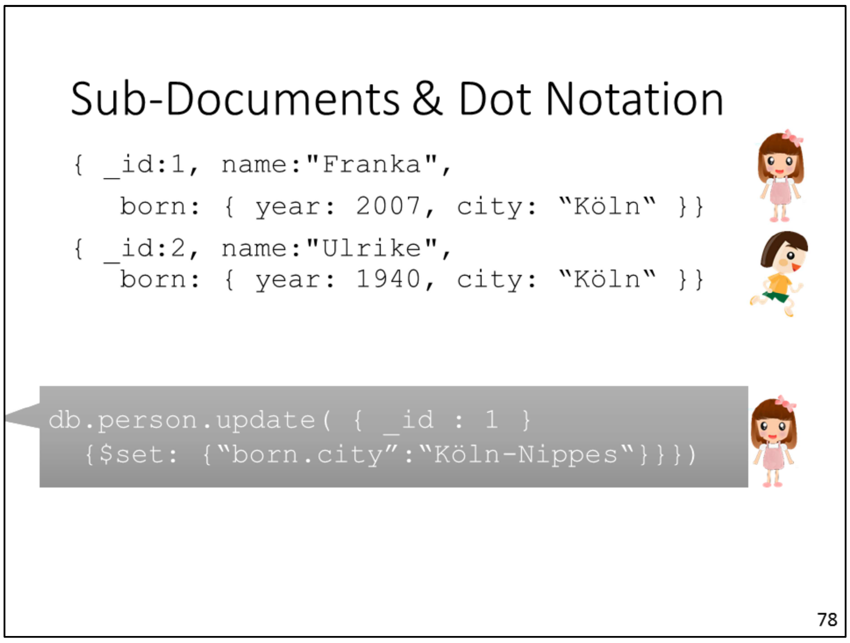
```
db.person.find({born:{city:"Köln"}})
```

```
db.person.find({"born.city":"Köln"}})
```

77

The second query on this slide does not make sense. It delivers no result. A document would be found, if the born field has exactly the given structure. Therefore, for checking sub-document fields, the dot-notation is used: field.subfield

77



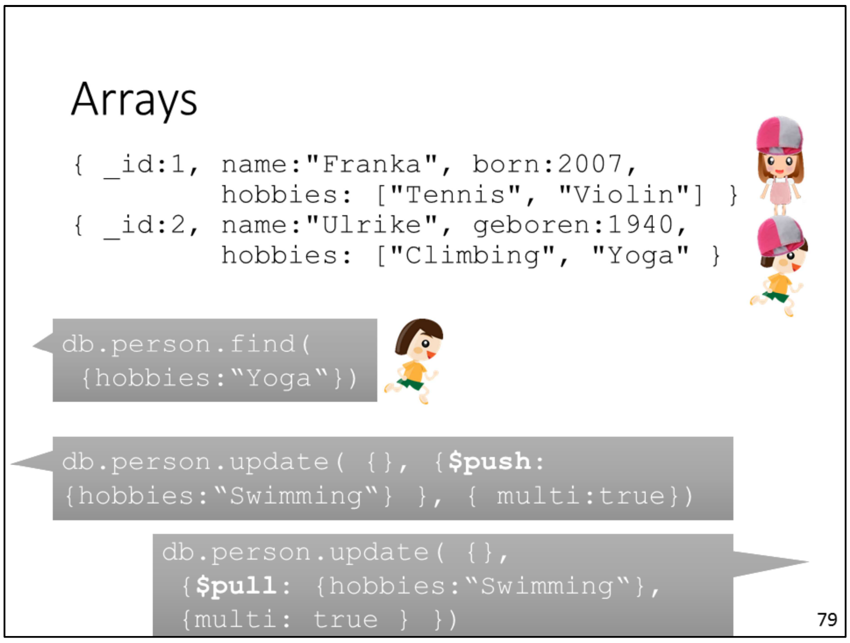
Sub-Documents & Dot Notation

```
{ _id:1, name:"Franka",
   born: { year: 2007, city: "Köln" }}
{ _id:2, name:"Ulrike",
   born: { year: 1940, city: "Köln" }}
```

```
db.person.update( { _id : 1 }
 {$set: {"born.city":"Köln-Nippes"}}})
```

78

78



Arrays

```
{ _id:1, name:"Franka", born:2007,
         hobbies: ["Tennis", "Violin"] }
{ _id:2, name:"Ulrike", geboren:1940,
         hobbies: ["Climbing", "Yoga" }
```

```
db.person.find(
 {hobbies:“Yoga“})
```

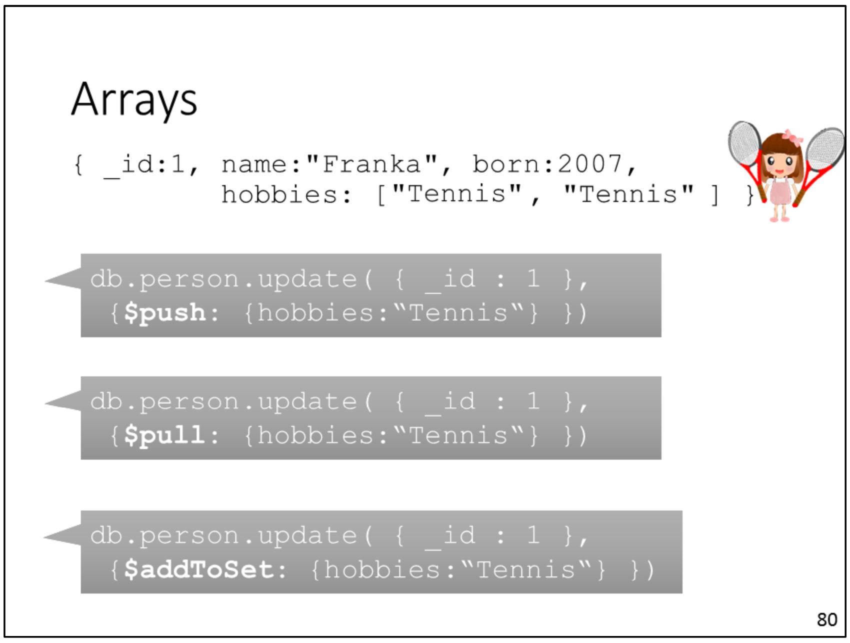
```
db.person.update( {}, {$push:
{hobbies:“Swimming“} }, { multi:true})
```

```
db.person.update( {},
 {$pull: {hobbies:“Swimming“},
 {multi: true } })
```

79

When an array field is compared to a value, it is evaluated as true when the array contains the value.
The second command on this slide uses the $push modifier to add the hobby swimming to all documents. If a document does not have the hobbies array, it will be initialized with an one-element array that contains “swimming”. With $pull, an element can be removed from an array.
Arrays are ordered. Similar to Redis, the last element in an array can be removed using $pop : { hobbies: 1 }. The first element can be removed using $pop : { hobbies: -1 }.

79



Arrays

```
{ _id:1, name:"Franka", born:2007,
         hobbies: ["Tennis", "Tennis" ] }
```

```
db.person.update( { _id : 1 },
 {$push: {hobbies:“Tennis“} })
```

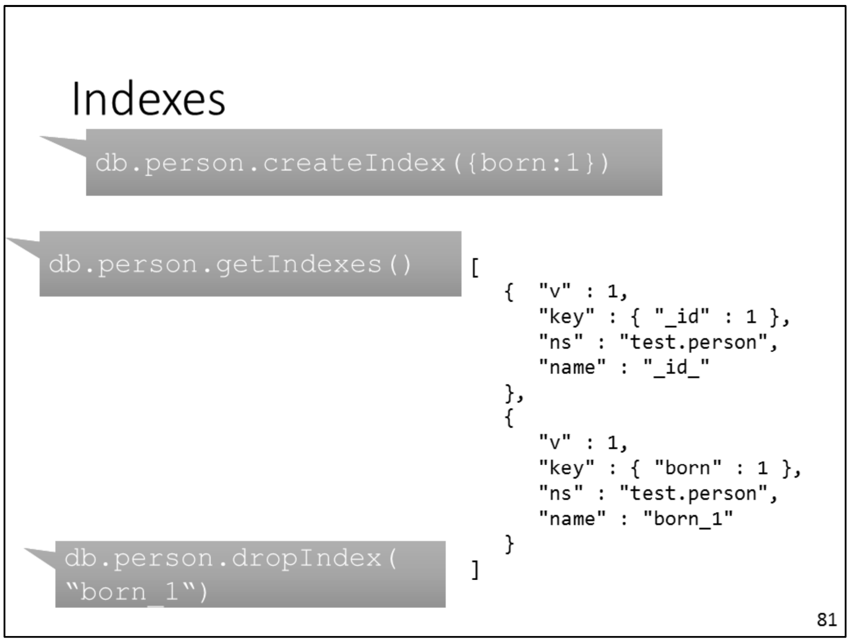
```
db.person.update( { _id : 1 },
 {$pull: {hobbies:“Tennis“} })
```

```
db.person.update( { _id : 1 },
 {$addToSet: {hobbies:“Tennis“} })
```

80

Arrays can contain duplicates. $pull removes all hits.
Instead of $push, $addToSet can be used to insert an element into an array. If it is already in there, it will not be inserted. This is how MongoDB supports sets.

80



Indexes

```
db.person.createIndex({born:1})
```

```
db.person.getIndexes()
```

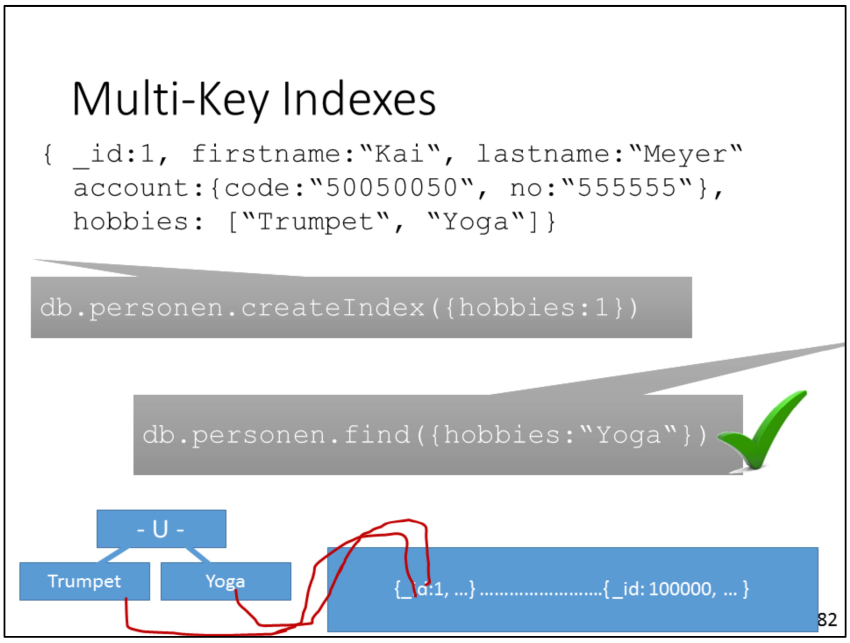
```
[
  {  "v" : 1,
     "key" : { "_id" : 1 },
     "ns" : "test.person",
     "name" : "_id_"
  },
  {
     "v" : 1,
     "key" : { "born" : 1 },
     "ns" : "test.person",
     "name" : "born_1"
  }
]
```

```
db.person.dropIndex(
"born_1")
```

81

createIndex creates an index on a given field. Similar to the sort method, 1 / -1 defines the sort order: ascending or descending. MongoDB uses a B+ tree as index structure. The index makes exact-match queries, range queries and sort queries on the given field very fast. There always is an index on the _id. That is why queries on the _id are very fast.

81



Multi-Key Indexes

```
{ _id:1, firstname:"Kai", lastname:"Meyer"
  account:{code:"50050050", no:"555555"},
  hobbies: ["Trumpet", "Yoga"]}
```

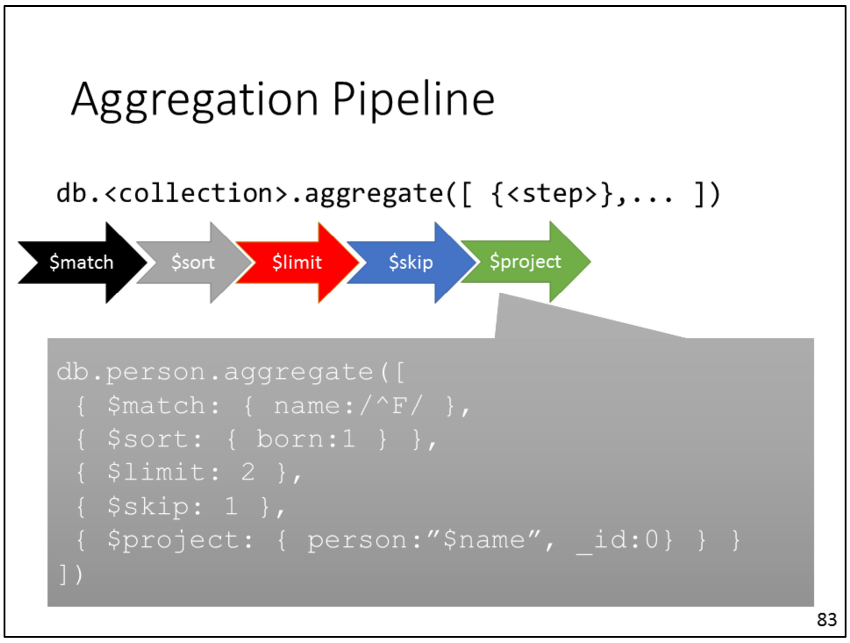
```
db.personen.createIndex({hobbies:1})
```

```
db.personen.find({hobbies:"Yoga"})
```

82

If an index is created on a field which is used for arrays, the index will automatically become a multi-key index. When a document is inserted, each value in the array is inserted into the index, together with pointers to the document. The multi-key index accelerates containment queries on arrays.

82



Aggregation Pipeline

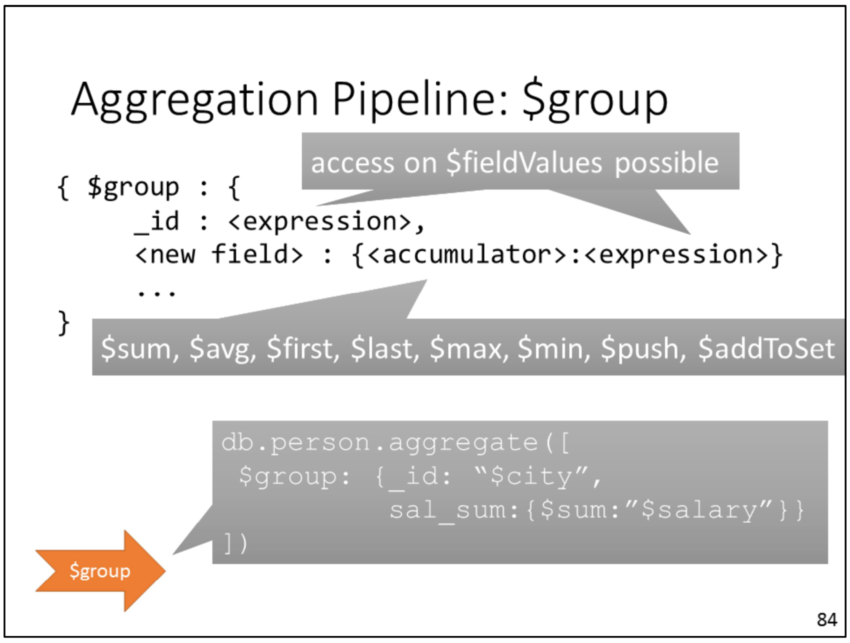
```
db.<collection>.aggregate([ {<step>},... ])
```

$match
$sort
$limit
$skip
$project

```
db.person.aggregate([
 { $match: { name:/^F/ },
 { $sort: { born:1 } },
 { $limit: 2 },
 { $skip: 1 },
 { $project: { person:"$name", _id:0} } }
])
```

83

The MongoDB aggregation pipeline makes it possible to write complex queries. The aggregate operation takes an array of steps which should be executed. The example on this slide uses five steps.



Aggregation Pipeline: $group

```
{ $group : {
     _id : <expression>,
     <new field> : {<accumulator>:<expression>}
     ...
}
```

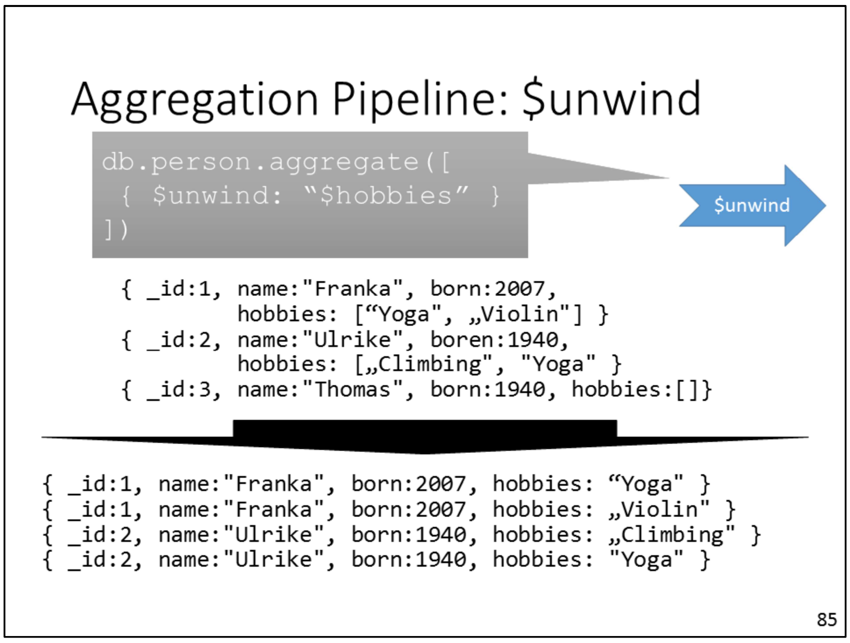
access on $fieldValues possible

$sum, $avg, $first, $last, $max, $min, $push, $addToSet

```
db.person.aggregate([
 $group: {_id: "$city",
          sal_sum:{$sum:"$salary"}}
])
```

84

The _id contains the field on which the data is grouped by. With a dollar sign, field values from the input document (or the previous step) can be accessed. Other fields must contain accumulators, i.e. aggregation functions.



Aggregation Pipeline: $unwind

```
db.person.aggregate([
  { $unwind: "$hobbies" }
])
```

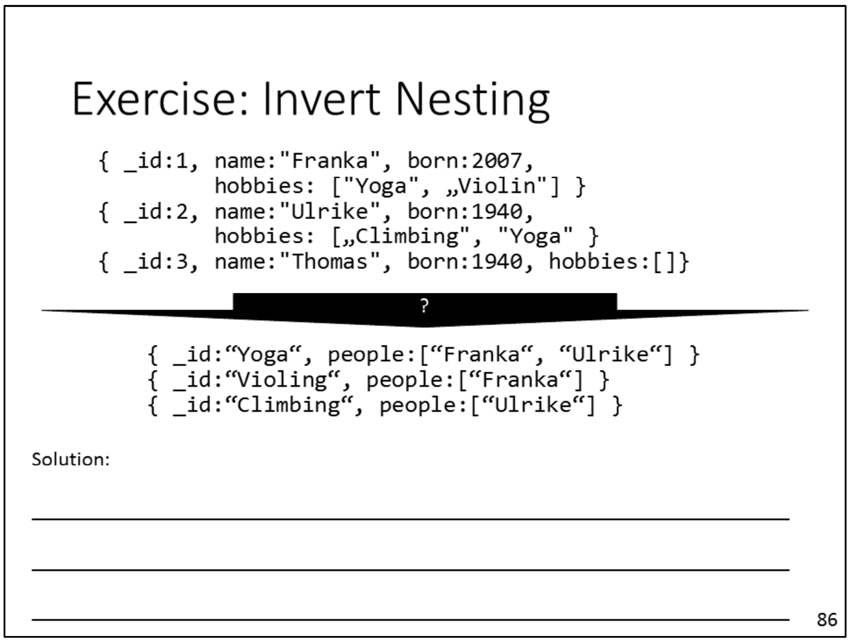
```
  { _id:1, name:"Franka", born:2007,
           hobbies: ["Yoga", „Violin"] }
  { _id:2, name:"Ulrike", boren:1940,
           hobbies: [„Climbing", "Yoga" }
  { _id:3, name:"Thomas", born:1940, hobbies:[]}
```

```
{ _id:1, name:"Franka", born:2007, hobbies: "Yoga" }
{ _id:1, name:"Franka", born:2007, hobbies: „Violin" }
{ _id:2, name:"Ulrike", born:1940, hobbies: „Climbing" }
{ _id:2, name:"Ulrike", born:1940, hobbies: "Yoga" }
```

85

85

In the $unwind step, an array is split into its elements. The result is that every document is there multiple times, as often as the array has elements.
When the array is empty or not present, the document is skipped.



Exercise: Invert Nesting

```
{ _id:1, name:"Franka", born:2007,
         hobbies: ["Yoga", „Violin"] }
{ _id:2, name:"Ulrike", born:1940,
         hobbies: [„Climbing", "Yoga" }
{ _id:3, name:"Thomas", born:1940, hobbies:[]}
```

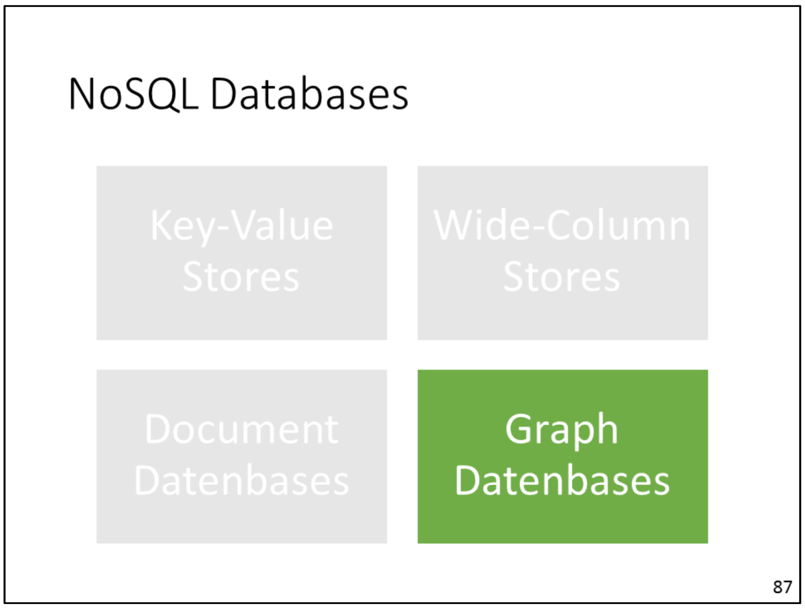
```
{ _id:"Yoga", people:["Franka", "Ulrike"] }
{ _id:"Violing", people:["Franka"] }
{ _id:"Climbing", people:["Ulrike"] }
```

Solution:

86

86

Use the aggregation pipeline to transform the person collection into the given form.



NoSQL Databases

Key-Value Stores

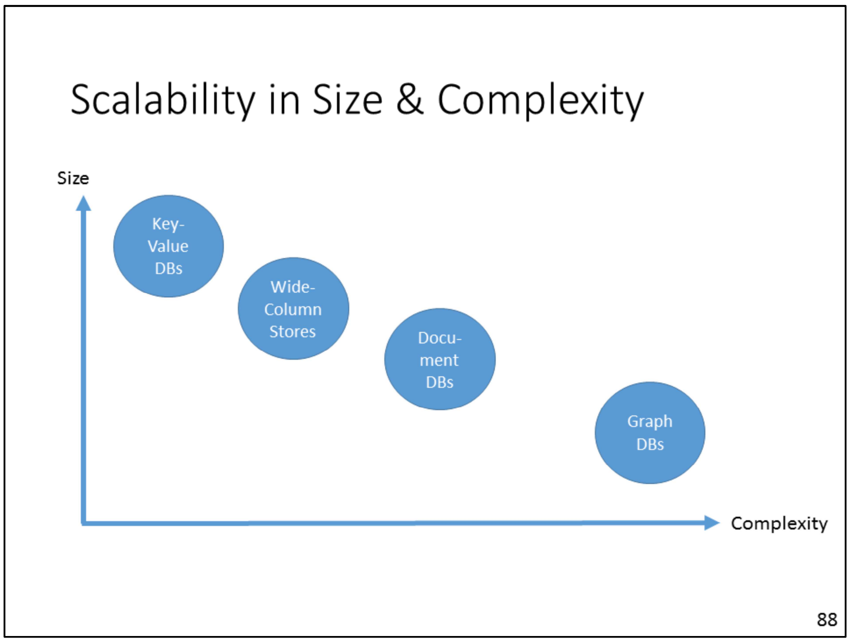
Wide-Column Stores

Document Datenbases

Graph Datenbases

87

87



Scalability in Size & Complexity

88

The database systems presented so far scale in the size. With the usage of partitioning, large data sets can be distributed across multiple machines. Key-value stores do not allow complex structure. Even if lists and hashes are used, a data item is still a key-value pair. Wide-column stores and document databases allow more complex structures, but the complexity is limited to one single data item. In graph databases, relationships between data items can be stored.

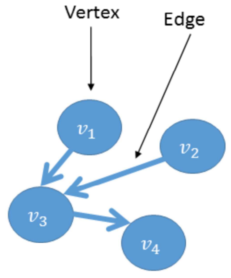
88

Mathematical Foundations

$$\overset{\text{Graph}}{G} = (\overset{\text{Vertices}}{V}, \overset{\text{Edges}}{E})$$

e.g.

$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{e_1, e_2, e_3\}$$
$$= \{(v_1, v_3), (v_2, v_3), (v_3, v_4)\}$$



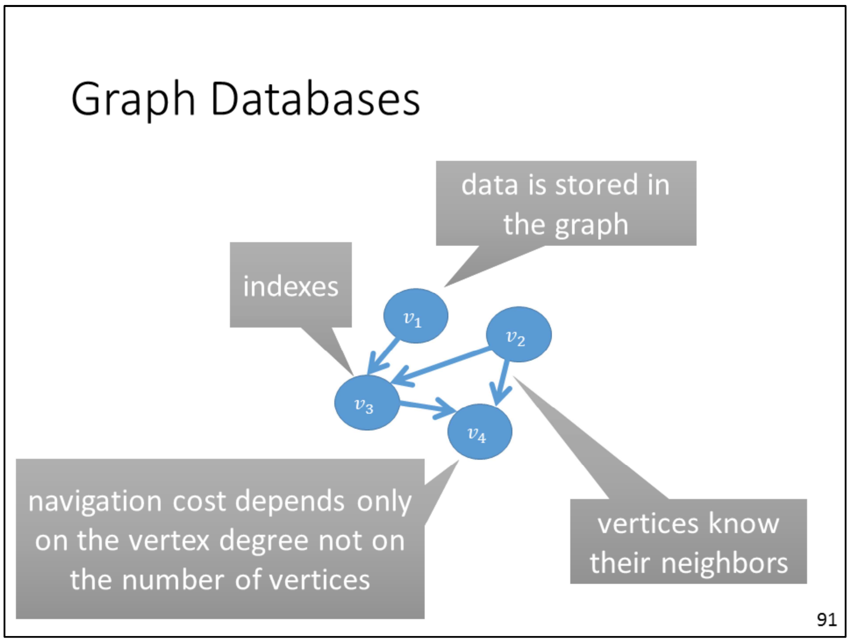
A graph is defined by a set of all its vertices and edges that connect the vertices.

RDF: Resourse Description Framework

subject	predicate	object
http://dbpedia.org/resource/ Krefeld_Hauptbahnhof	rdf:type	http://schema.org/Place
http://dbpedia.org/resource/ Krefeld_Hauptbahnhof	foaf:name	Krefeld Hauptbahnhof
http://dbpedia.org/resource/ Krefeld_Hauptbahnhof	georss:point	51.325833333333335 6.569444444444445
http://dbpedia.org/resource/ Krefeld_Hauptbahnhof	rdf:comment	Krefeld Hauptbahnhof is the largest train station in Krefeld. There …
http://dbpedia.org/resource/ Krefeld_Hauptbahnhof	country	http://dbpedia.org/resource/Germany
http://dbpedia.org/resource/ Germany	foaf:name	Germany

```
SELECT ?land
WHERE { ?x foaf:name "Krefeld Hauptbahnhof"; ?x country ?y .
        ?y foaf:name ?land }
```

RDF is very important for the sematic web. Every expression is a triplet: subject, predicate, object. The subject is a resource (identified by a URI). The object is either a literal (string, number, …) or the URI of another resource.
This example is from the RDF database DBPedia.org
The query on this slide is given in the language SPARQL. Expressions in SPARQL also have the subject-predicate-object form. The shown query finds the name of the county in which the main station of Krefeld is located.



Graph Databases

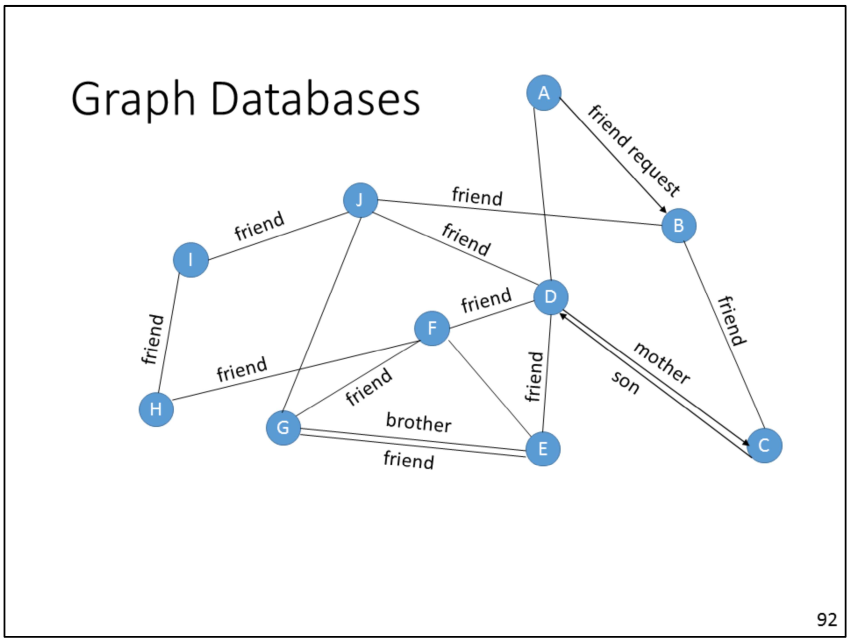
data is stored in the graph

indexes

navigation cost depends only on the vertex degree not on the number of vertices

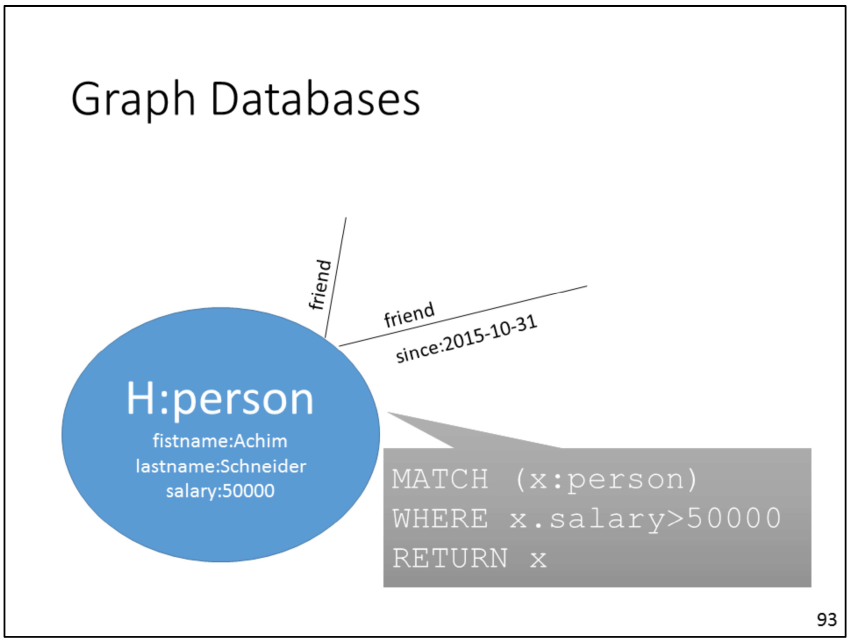
vertices know their neighbors

1) Data is stored in the graph (in vertices and edges), not in tables
2) Edges (information about neighbors) are stored directly within the node data item.
3) When the graph grows, the performance of queries stays the same. Costs for traversing the graph depend only on the in- and out-degree of the vertex, i.e. the number of incoming and outgoing edges.
4) Typical operations are traversing from vertex to vertex via edges. To quickly find vertexes and edges, there are index structures.



Graph Databases

Different from the other classes of NoSQL databases, graph databases have no simple put/get API but a query language or a more complex API. Vertices have an internal ID, a label and properties (see next slide). Edges connect to vertices and also have a label and properties. Edges can be directed or undirected (in most graph databases, like Neo4J, they can only be directed).

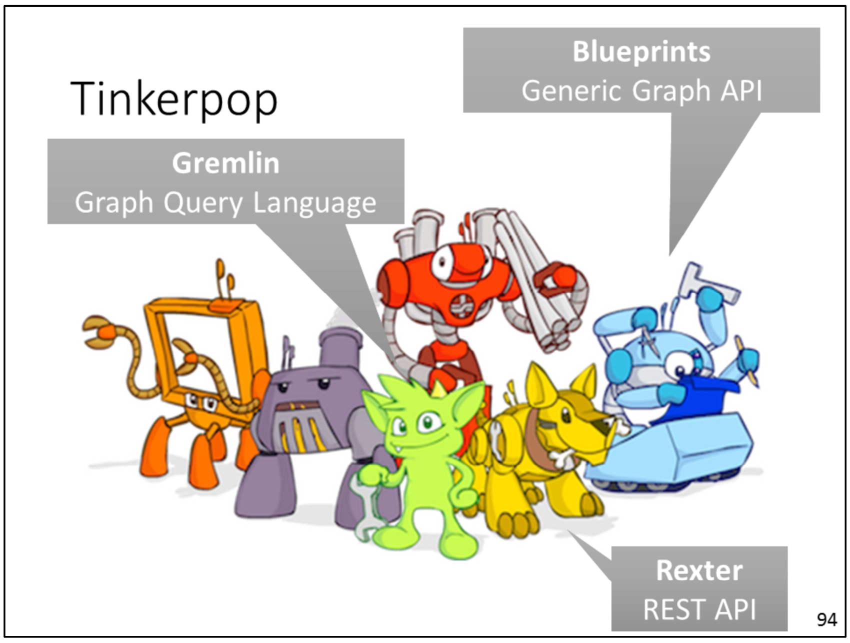


Graph Databases

```
MATCH (x:person)
WHERE x.salary>50000
RETURN x
```

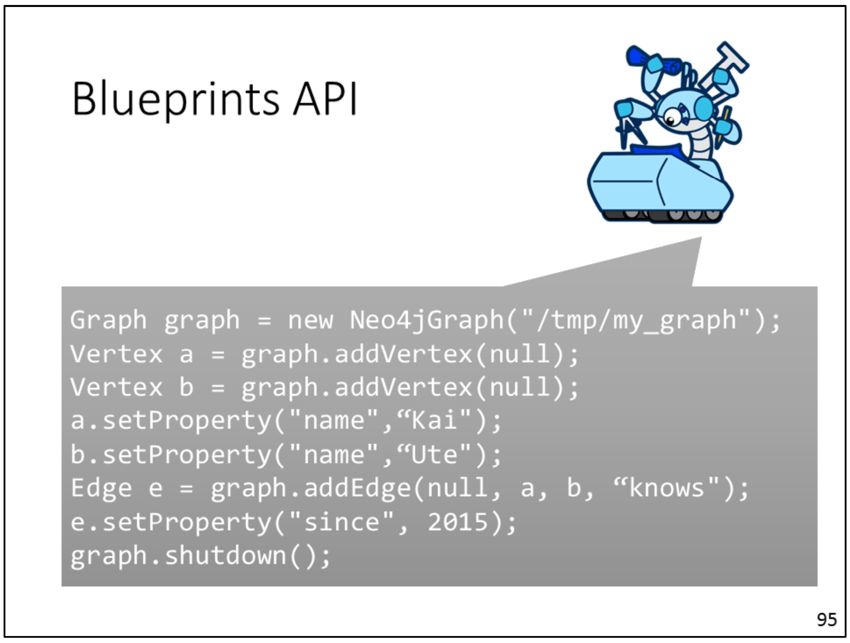
Both vertices and edges can have properties of the form key:value, like in JSON. Furthermore they have labels, here “person“ and “friend“.

The shown query is a Cypher query (more later).



Tinkerpop

Tinkerpop is an open framework that consists of multiple components. Tinkerpop is independent of an actual graphdatabase and it acts like a DB Gateway. That makes a migration from one graph database to another simple (like JDBC for relational databases). Many graph databases are supported: Neo4j, Titan, OrientDB, DEX, TinkerGraph, …



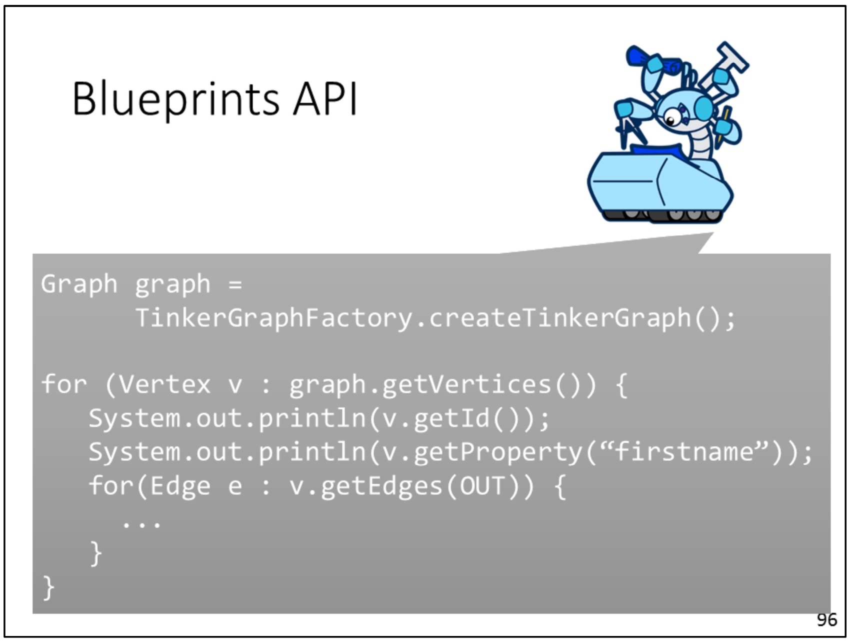
Blueprints API

```
Graph graph = new Neo4jGraph("/tmp/my_graph");
Vertex a = graph.addVertex(null);
Vertex b = graph.addVertex(null);
a.setProperty("name",“Kai");
b.setProperty("name",“Ute");
Edge e = graph.addEdge(null, a, b, “knows");
e.setProperty("since", 2015);
graph.shutdown();
```

95

Blueprints is the basis of the Tinkerpop stack. The API can be used to make read and write accesses to a graph in Java. The other Tinkerpop components (Gremlin etc.) use Blueprints.

95



Blueprints API

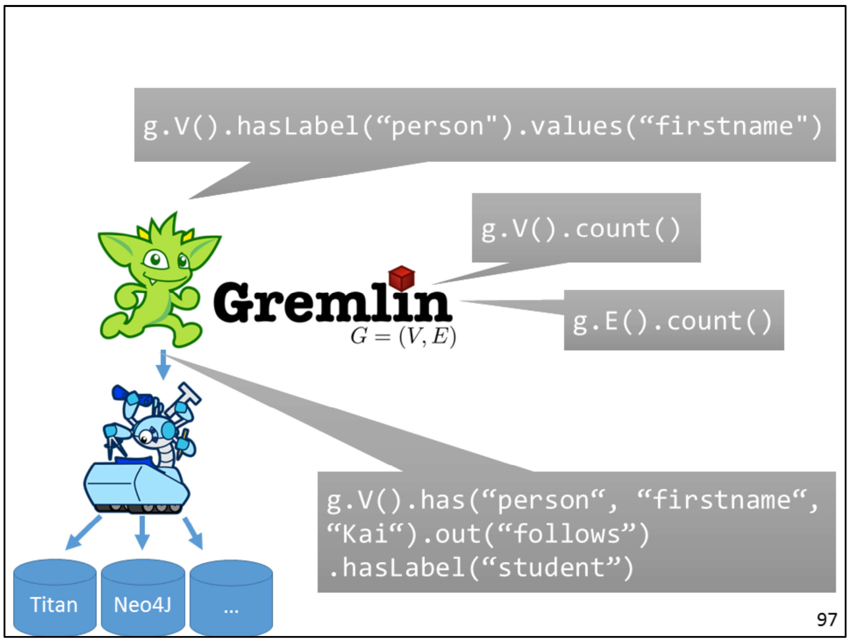
```
Graph graph =
      TinkerGraphFactory.createTinkerGraph();

for (Vertex v : graph.getVertices()) {
   System.out.println(v.getId());
   System.out.println(v.getProperty(“firstname”));
   for(Edge e : v.getEdges(OUT)) {
     ...
   }
}
```

96

Here, a Tinkergraph is opened (an in-memory graph database).

The Blueprints API has methods to iterate over vertices and edges of a graph, to find specific vertices and edges, e.g. graph.getVertex(“1”) to find the vertex with the ID 1. From a vertex, one can use methods to read its properties and access incoming and outgoing edges. This way, one can navigate to neighbor vertices.

96



```
g.V().hasLabel("person").values("firstname")
```

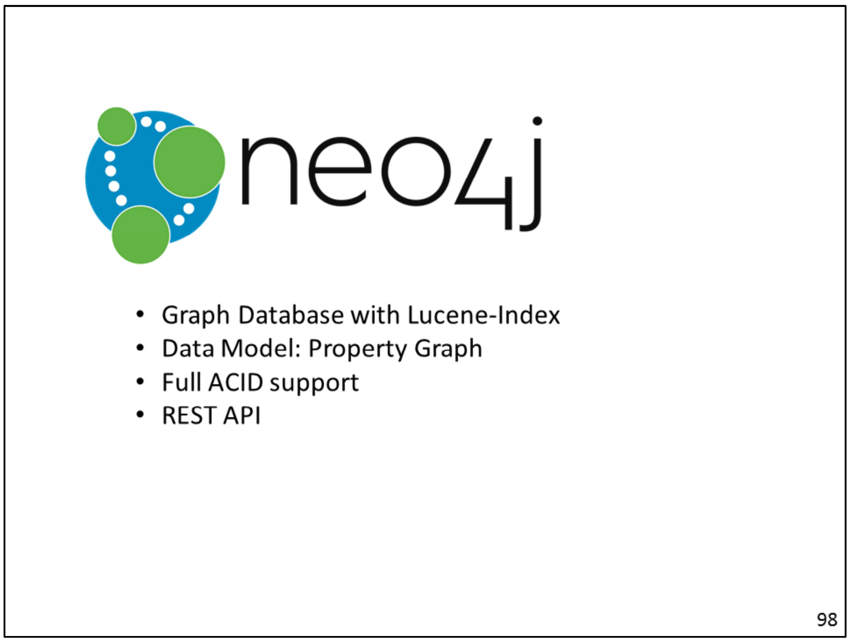
```
g.V().count()
```

```
g.E().count()
```

```
g.V().has("person", "firstname",
"Kai").out("follows")
.hasLabel("student")
```

These are some example queries in Gremlin. The first one finds the first names of all person vertices, the second one the number of vertices in the graph. The bottom query delivers all student vertices that are followed by the person with the first name Kai. This query shows how to traverse the graph: Start at the vertex with first name Kai, navigate to the neighbors using the edges with label "follows" and filter out all vertices which do not have the label student.

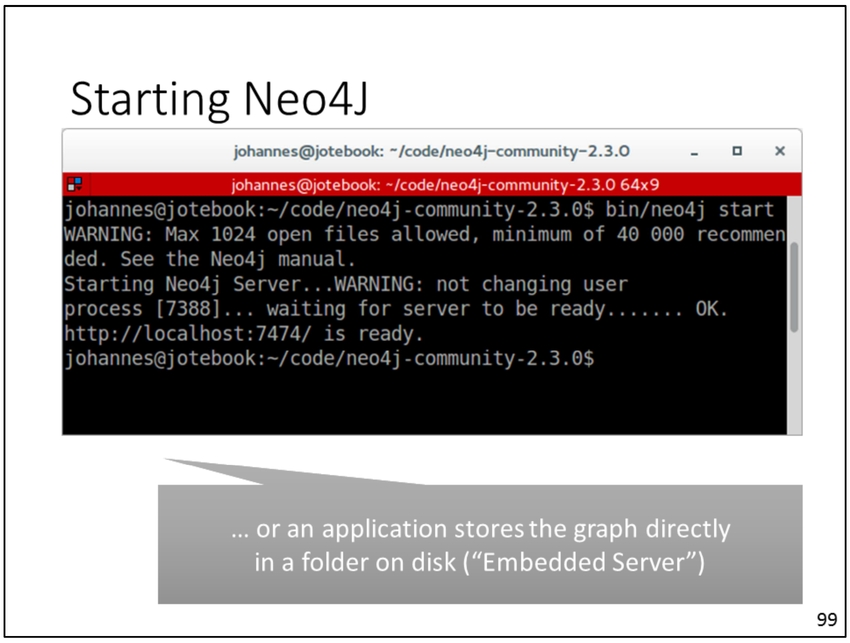
97



- Graph Database with Lucene-Index
- Data Model: Property Graph
- Full ACID support
- REST API

The Apache Lucene Index makes it possible to quickly find nodes and edges that fulfil specific criteria. Full-text search is supported.

98



Starting Neo4J

johannes@jotebook: ~/code/neo4j-community-2.3.0

johannes@jotebook: ~/code/neo4j-community-2.3.0 64x9

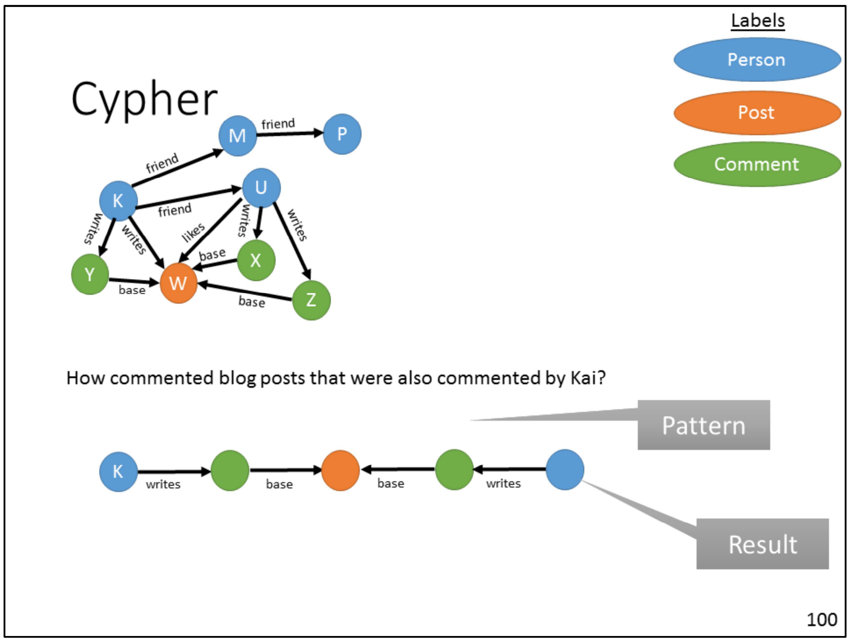
```
johannes@jotebook:~/code/neo4j-community-2.3.0$ bin/neo4j start
WARNING: Max 1024 open files allowed, minimum of 40 000 recommen
ded. See the Neo4j manual.
Starting Neo4j Server...WARNING: not changing user
process [7388]... waiting for server to be ready....... OK.
http://localhost:7474/ is ready.
johannes@jotebook:~/code/neo4j-community-2.3.0$
```

… or an application stores the graph directly in a folder on disk (“Embedded Server”)

99

After starting the Neo4J server process, the web interface can be used and queries can be sent using the REST API.

99



Cypher

Labels

Person

Post

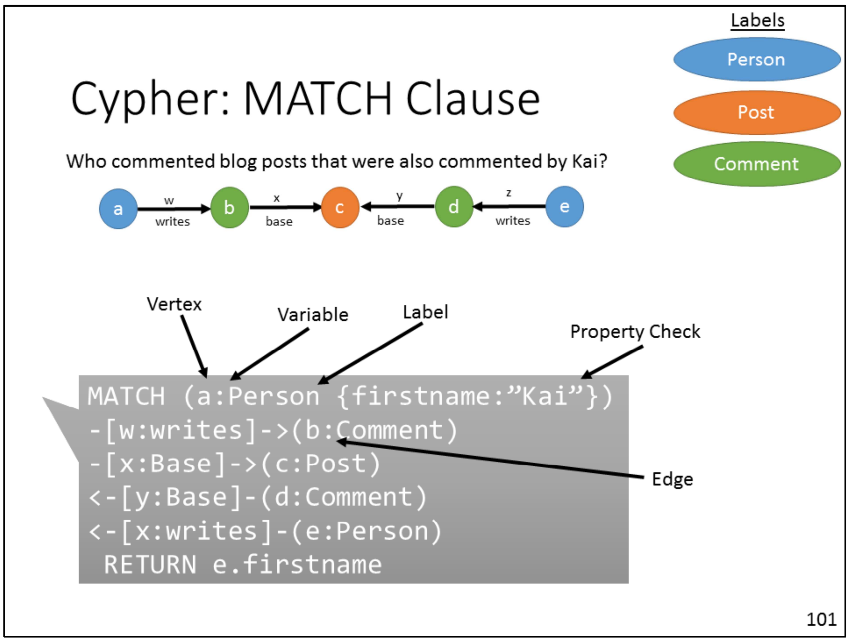
Comment

How commented blog posts that were also commented by Kai?

100

While Gremlin and the APIs of graph databases are used to navigate through the graph, Cypher is a language where you give a specific pattern that is searched in the graph. For every match, the rest of the query is evaluated. In the MATCH pattern, variables can be introduced. These variables can be accessed in other query parts (see next slide).

100



Cypher: MATCH Clause

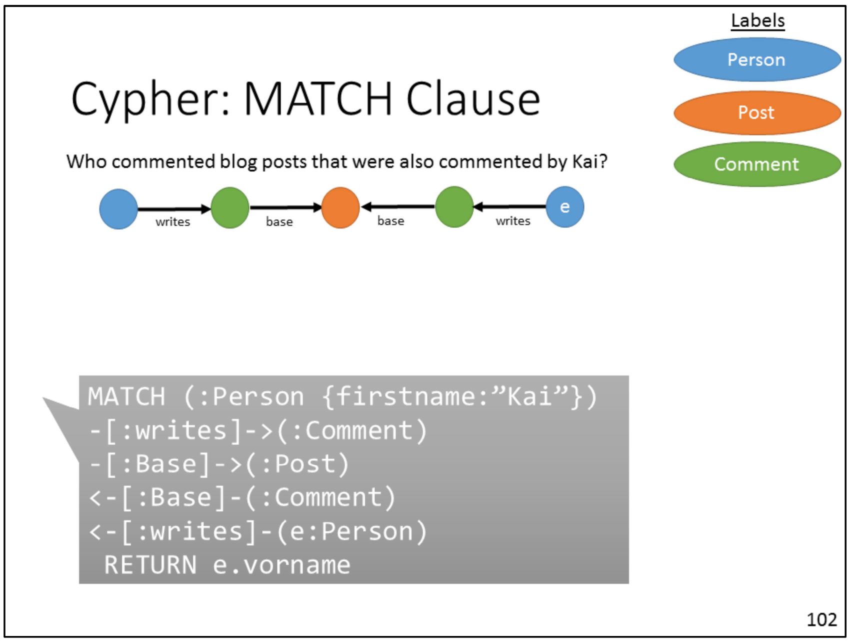
Who commented blog posts that were also commented by Kai?

Labels
Person
Post
Comment
a
w
writes
b
x
base
c
y
base
d
z
writes
e
Vertex
Variable
Label
Property Check
Edge

```
MATCH (a:Person {firstname:”Kai”})
-[w:writes]->(b:Comment)
-[x:Base]->(c:Post)
<-[y:Base]-(d:Comment)
<-[x:writes]-(e:Person)
 RETURN e.firstname
```

101

Paranthesis: (Node)
after the colon: Label
Brackets: [Edge]
The first item written in parenthesis or brackets: (variable) or [variable]
Edge directions: <-- or --> or -- (both directions), or -[edge]- etc.
Curly braces: {Property: „value“, Property: „value“}



Cypher: MATCH Clause

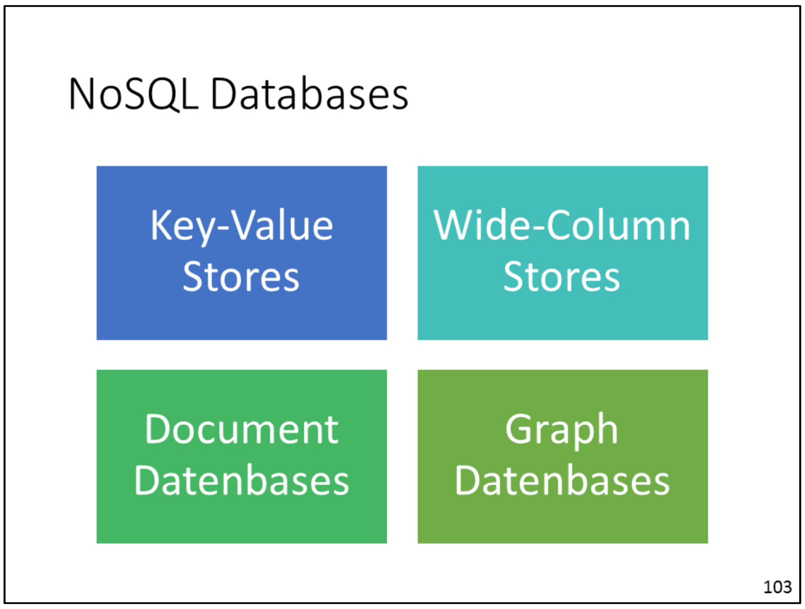
Who commented blog posts that were also commented by Kai?

Labels
Person
Post
Comment
writes
base
base
writes
e

```
MATCH (:Person {firstname:”Kai”})
-[:writes]->(:Comment)
-[:Base]->(:Post)
<-[:Base]-(:Comment)
<-[:writes]-(e:Person)
 RETURN e.vorname
```

102

Variables can be omitted if they are never needed.
Labels are also optional. MATCH ({fristname:”Kai”}) would also work. But maybe a cat with the name Kai is also found, not only a person ;-) Omitting edge labels is also possible, e.g. (:Comment)<--(e:Person) Then, the edge is traversed independent from its label.



NoSQL Databases

Key-Value Stores

Wide-Column Stores

Document Datenbases

Graph Datenbases