Distributed Data Management

Data Models and Query Languages

Thorsten Papenbrock
Felix Naumann
F-2.03/F-2.04, Campus II
Hasso Plattner Institut

Introduction

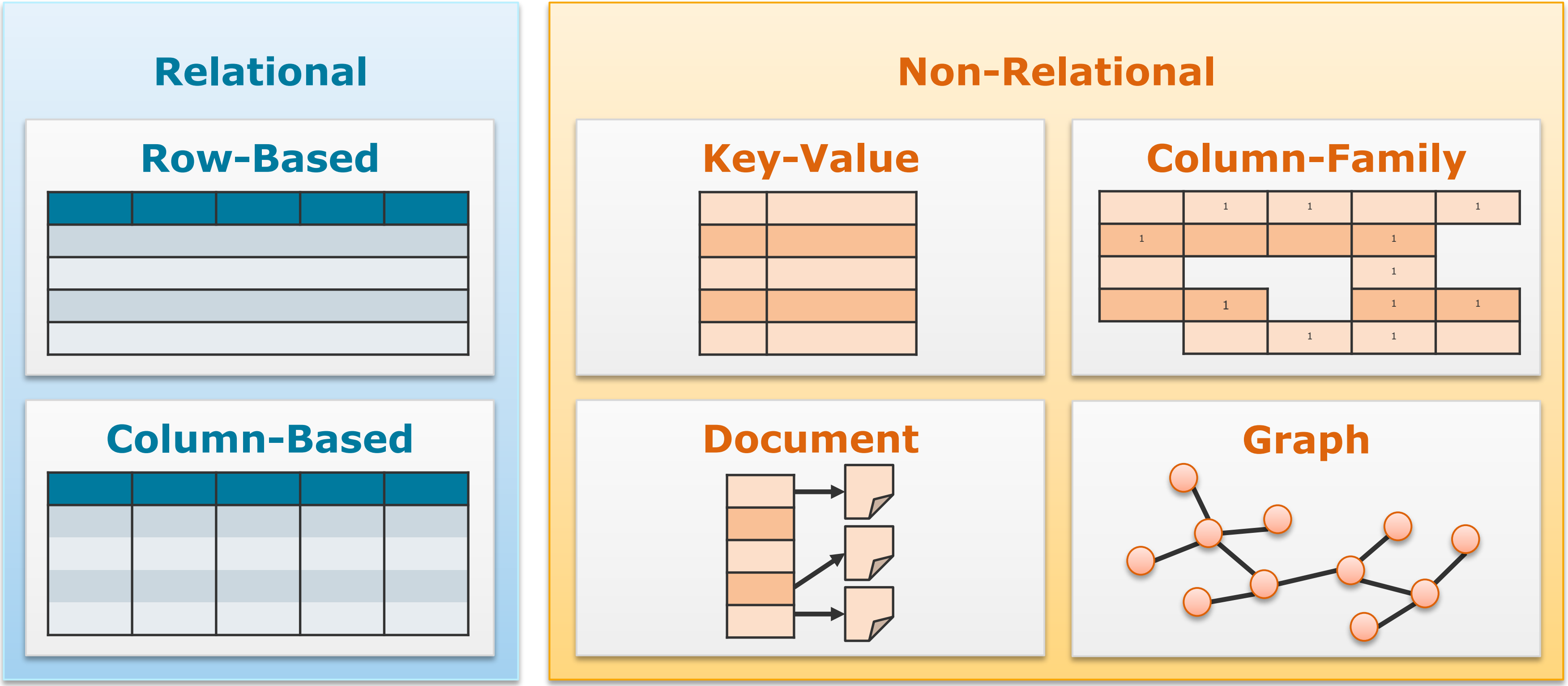
Layering Data Models

1. Conceptual layer
   - Data structures, objects, modules, …
     - ➢ Application code
2. Logical layer

   our focus now
   - Relational tables, JSON, XML, graphs, …
     - ➢ Database management system (DBMS) or storage engine
3. Representation layer
   - Bytes in memory, on disk, on network, …
     - ➢ Database management system (DBMS) or storage engine
4. Physical layer
   - Electrical currents, pulses of light, magnetic fields, …
     - ➢ Operating system and hardware drivers

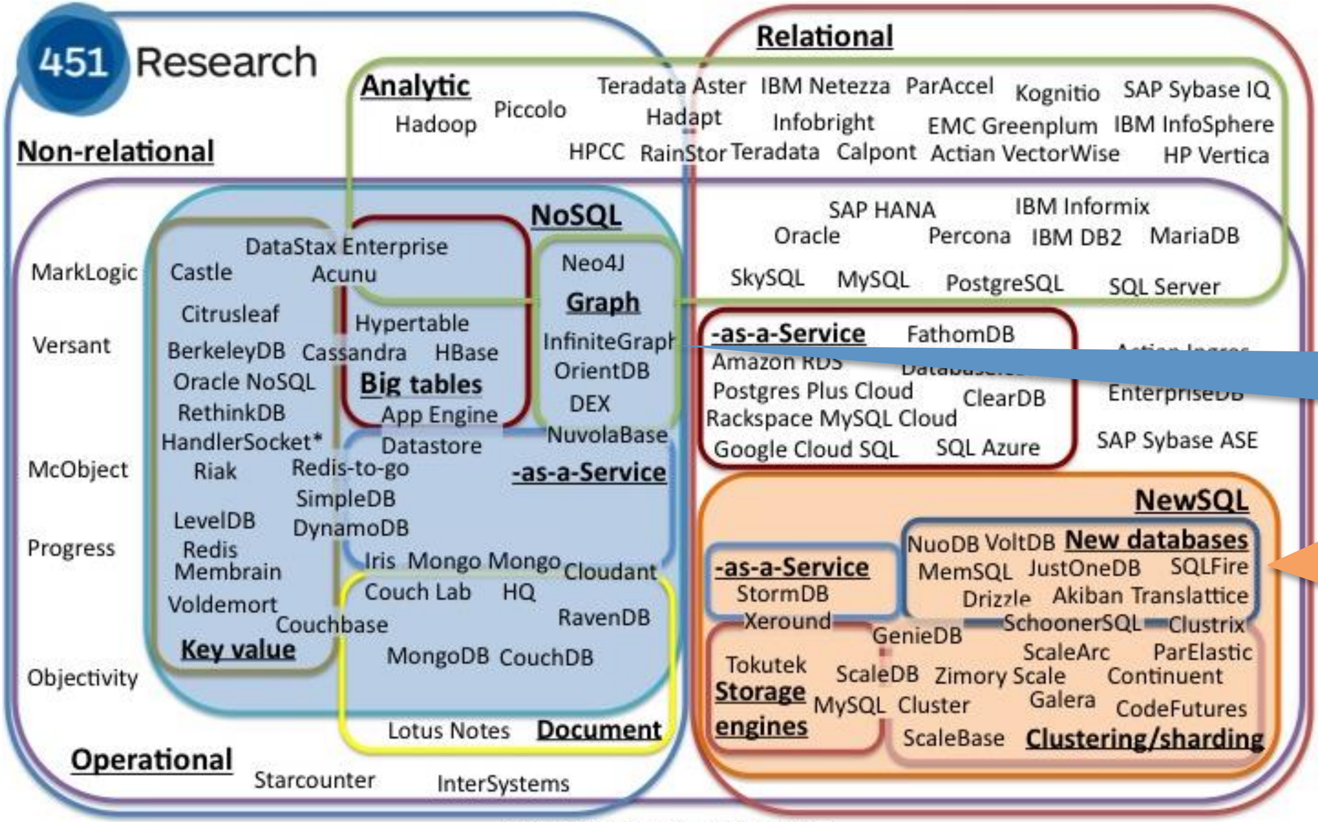
Relational and Non-Relational Data Models

HPI Hasso Plattner Institut



Overview

Relational and Non-Relational DBMSs



"No SQL" or rather "not only SQL" systems because most support some SQL dialect.

A class of relational DBMSs that seek to provide the same scalable performance of NoSQL systems for OLTP workloads while still maintaining all ACID guarantees.

Data Models

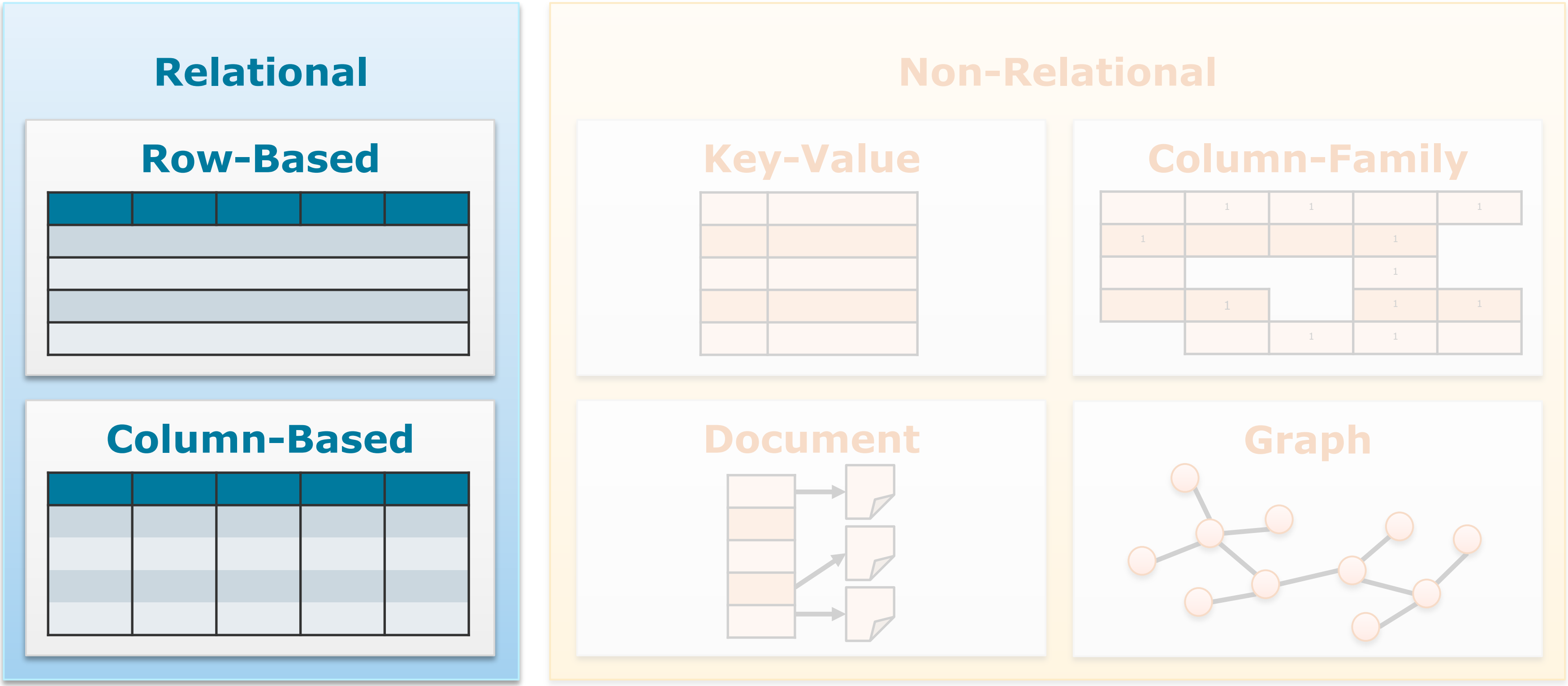
A data model consists of three parts:

1. Structure
   - physical and conceptual data layout
2. Constraints
   - inherent limitations and rules
3. Operations
   - possible query and modification methods

[Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 9780131873254]

Overview

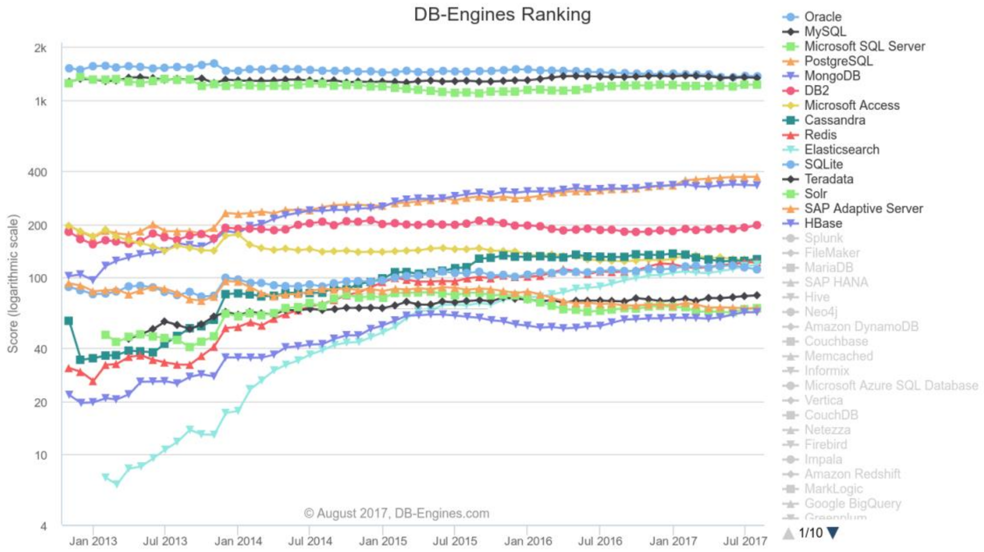
Relational and Non-Relational Data Models



The Relational Data Model

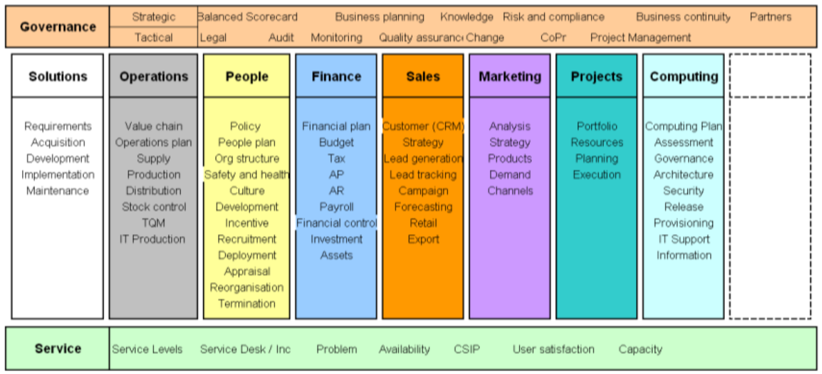
Natural Relational Data

Transactional Data



Statistical Data

Master Data



Business Data

Popular relational DBMS

Rank			DBMS	Database Model	Score		
Aug 2017	Jul 2017	Aug 2016			Aug 2017	Jul 2017	Aug 2016
1.	1.	1.	Oracle	Relational DBMS	1367.88	-7.00	-59.85
2.	2.	2.	MySQL	Relational DBMS	1340.30	-8.81	-16.73
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1225.47	-0.52	+20.43
4.	4.	4.	PostgreSQL	Relational DBMS	369.76	+0.32	+54.51
5.	5.	5.	DB2	Relational DBMS	197.47	+6.22	+11.58
6.	6.	6.	Microsoft Access	Relational DBMS	127.03	+0.90	+2.98
7.	7.	7.	SQLite	Relational DBMS	110.85	-3.02	+0.99
8.	8.	8.	Teradata	Relational DBMS	79.23	+0.86	+5.59
9.	9.	9.	SAP Adaptive Server	Relational DBMS	66.92	+0.00	-4.13
10.	10.	10.	FileMaker	Relational DBMS	59.65	+1.00	+4.64
11.	11.	↑ 13.	MariaDB	Relational DBMS	54.70	+0.33	+17.82
12.	12.	12.	SAP HANA	Relational DBMS	47.97	+0.03	+5.24
13.	13.	↓ 11.	Hive	Relational DBMS	47.30	+1.10	-0.51
14.	14.	14.	Informix	Relational DBMS	27.42	-0.25	-1.63
15.	15.	↑ 16.	Microsoft Azure SQL Database	Relational DBMS	21.91	-0.38	+2.39
16.	16.	↓ 15.	Vertica	Relational DBMS	21.81	+0.02	+1.33
17.	17.	17.	Netezza	Relational DBMS	19.58	-0.28	+0.10
18.	18.	18.	Firebird	Relational DBMS	18.07	-0.92	+2.17
19.	19.	↑ 23.	Impala	Relational DBMS	13.06	-0.25	+4.25

https://db-engines.com/en/ranking

Definition

1. Structure

- Schemata: named, non-empty, typed, and unordered sets of attributes
  - Example: `Person(ID,Surname,Name,Gender,Address)`
- Instances: sets of records, i.e., functions that assign values to attributes
  - Example: `(275437,`Miller´,`Frank´,`male´,`Millstr. 5´)`

2. Constraints

- Integrity constraints: data types, keys, foreign-keys, …

3. Operations

- Relational algebra (and relational calculus)
- Usually implemented as Structured Query Language (SQL)

[E. F. Codd. *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, 13(6):377–387, 1970]

Querying: SQL

SELECT <attribute list>
FROM <relation list>
WHERE <conditions>
GROUP BY <grouping attributes>
HAVING <grouping conditions>
ORDER BY <attribute list>**;**

Declarative query languages specify the result of a query and not how it should be obtained:

- Easier to understand
- Transparently optimizable
- Implementation independent

Further keywords:

DISTINCT, AS, JOIN

AND, OR

MIN, MAX, AVG, SUM, COUNT

NOT, IN, LIKE, ANY, ALL, EXISTS

UNION, EXCEPT, INTERSECT

DDL\DML:

CREATE TABLE

DROP TABLE

ALTER TABLE

INSERT INTO … VALUES

DELETE FROM … WHERE

UPDATE … SET … WHERE

The Relational Data Model
Querying: SQL

HPI Hasso Plattner Institut

Grundbausteine:

SELECT <Attributliste> (= Schema der Ergebnisrelation; * für alle Attribute)
FROM <Relationenliste> (Relationen aus denen die Tupel stammen)
WHERE <Bedingungen>; (Bedingungen an die Daten; Verknüpfung über Schlüsselwort **AND**)

SELECT:
- Entspricht Projektion π der relationalen Algebra
- Umbenennung: SELECT Titel AS Filmtitel
- Arithmetik: SELECT Länge * 3.1415 AS LängeMalPi
- Konstanten: SELECT ‚Herr' AS Titel
- Duplikateliminierung δ: SELECT DISTINCT Titel

FROM:
- Entspricht Kreuzprodukt × der relationalen Algebra (falls mehrere Relationen gewählt)

WHERE:
- Entspricht Selektion σ der relationalen Algebra
- WHERE-Teil der Anfrage ist optional
- Operatoren: =, <>, >, <, <=, >=, LIKE, NOT, ANY, ALL, EXISTS, IN, …
- Kann Kreuzprodukt des FROM-Teils zum Join machen:
  - WHERE Person.ID = Mensch.ID (Natürlicher Join $\bowtie$)
  - WHERE Person.Name = Mensch.Vorname (Theta Join $\bowtie_\theta$)

Komplexe Anfragen:

SELECT <Attributliste>
FROM <Relationenliste>
WHERE <Bedingungen>
GROUP BY <Gruppierungsattribute>
HAVING <Bedingungen auf Gruppierungsattribute>
ORDER BY <Attributliste>;

GROUP BY … HAVING:
- Entspricht Gruppierung γ der relationalen Algebra
- Aggregationsoperatoren für SELECT-Statement: AVG(<Attribut>), COUNT(<Attribut>), SUM(<Attribut>)
- HAVING entspricht einer Selektion nach der Gruppierung

ORDER BY:
- Entspricht Sortierung τ der relationalen Algebra
- Sortiert das Ergebnis der Anfrage entsprechend der Attributliste
- Aufsteigend: ORDER BY Vorname, Nachname ASC
- Absteigend: ORDER BY Vorname, Nachname DESC

Datendefinition: Data Definition Language (DDL)

CREATE TABLE <Tabellenname>(<Attributliste mit Datentypen>);
- Aufgabe: neue Tabelle erstellen
- Datentypen mit Länge n bzw. m: CHAR(n), VARCHAR(n), BIT(n), DECIMAL(n,m)
- Datentypen mit impliziter Länge: INT, FLOAT
- Datentypen für Objekte: CLOB, BLOB
- Datentypen für Zeiten: TIME, DATE, TIMESTAMP
- Bsp.: CREATE TABLE Schauspieler (
  Name CHAR(30),
  Adresse VARCHAR(255),
  Geschlecht CHAR(1),
  Geburtstag DATE);
- Nebenbedingungen:
  - Primärschlüssel: PRIMARY KEY
  - Eindeutigkeit: UNIQUE
  - Default-Werte: DEFAULT <Defaultwert>
  - Nicht-Null: NOT NULL
  - Fremdschlüssel: FOREIGN KEY (<Attributliste>) REFERENCES <Tabellenname>(<Attributliste>)
  - Weitere: CHECK, CREATE TRIGGER, CREATE ASSERTION, …

  Bsp.: CREATE TABLE Schauspieler (
  SchauspielerNummer INT PRIMARY KEY,
  Name CHAR(30) NOT NULL,
  Adresse VARCHAR(255) NOT NULL UNIQUE,
  Geschlecht CHAR(1),
  Geburtstag DATE DEFAULT DATE ‚0000-00-00',
  FOREIGN KEY (Adresse) REFERENCES Haus(Adresse));

DROP TABLE <Tabellenname>;
- Aufgabe: bestehende Tabelle löschen
- Bsp.: DROP TABLE Schauspieler;

ALTER TABLE <Tabellenname> <Aktion>;
- Aufgabe: bestehende Tabelle ändern
- ADD:
  - Attribut hinzufügen
  - Bsp.: ALTER TABLE Schauspieler ADD Telefon CHAR(16)
- DROP:
  - Attribut löschen
  - Bsp.: ALTER TABLE Schauspieler DROP Geburtstag

CREATE INDEX <Indexname> **ON** <Tabellenname>(<Attributliste des neuen Index>);
- Aufgabe: Index erstellen

DROP INDEX <Indexname>;
- Aufgabe: Index löschen

Datenbearbeitung: Data Modelling Language (DML)

INSERT INTO <Tabellenname>(<Attributliste>) **VALUES** (<Attributliste>);
- Aufgabe: Tupel einfügen
- Bsp.: INSERT INTO Studio(Name, Nummer) VALUES ('Pixa', 34);
- Ergebnis einer Anfrage für Einfügen nutzen:
  Bsp.: INSERT INTO Studios(Name)
  SELECT DISTINCT StudioName
  FROM Film
  WHERE StudioName NOT IN
  (SELECT Name
  FROM Studios);

DELETE FROM <Tabellenname> **WHERE** <Bedingung>;
- Aufgabe: Tupel löschen
- Bsp.: DELETE FROM Studio WHERE Name='Pixa';

UPDATE <Tabellenname> **SET** <Zuweisung> **WHERE** <Bedingung>;
- Aufgabe: Attributwerte ändern
- Bsp.: UPDATE Studio SET Name='Pixa' WHERE Name='Pi';

Bulk insert: IMPORT, LOAD, … (→ DBMS spezifisch)

Mengenoperationen

(<Anfrage>) **UNION** (<Anfrage>) (Liefert Vereinigung „∪" der beiden Ergebnismengen)

(<Anfrage>) **EXCEPT** (<Anfrage>) (Liefert Differenz „–" der beiden Ergebnismengen)

(<Anfrage>) **INTERSECT** (<Anfrage>) (Liefert Schnittmenge „∩" der beiden Ergebnismengen)

- UNION, EXCEPT und INTERSECT nutzen Mengensemantik (→ eliminieren Duplikate)
- UNION ALL, EXCEPT ALL und INTERSECT ALL nutzen Multimengensemantik (→ erhalten Duplikate)

Join-Varianten

1.) Kreuzprodukt mit Bedingung:
  SELECT *
  FROM <Join-Relation1>, <Join-Relation2>
  WHERE <Join-Attribut1> = <Join-Attribut2>;

2.) Schlüsselwort:
  <Tabellenname> **CROSS JOIN** <Tabellenname>
  <Tabellenname> **NATURAL JOIN** <Tabellenname>
  <Tabellenname> **NATURAL INNER JOIN** <Tabellenname>
  <Tabellenname> **NATURAL LEFT OUTER JOIN** <Tabellenname>
  <Tabellenname> **NATURAL RIGHT OUTER JOIN** <Tabellenname>
  <Tabellenname> **NATURAL FULL OUTER JOIN** <Tabellenname>

Sichten

CREATE VIEW <Sichtname> **AS** <Anfrage>;
- Aufgabe: Erstelle eine Sicht für die gegebene SQL-Anfrage

Querying: SQL – Examples

Schemata:

- `Product(maker, model, type)`
- `PC(model, speed, ram, hd, rd)`
- `Laptop(model, speed, ram, hd, screen)`

```
(SELECT DISTINCT maker
 FROM Product, Laptop
 WHERE Product.model = Laptop.model)
EXCEPT
(SELECT DISTINCT maker
 FROM Product, PC
 WHERE Product.model = PC.model);
```

"Find all makers that produce Laptops but no PCs."

```
SELECT *
FROM PC PC1, PC PC2
WHERE PC1.speed = PC2.speed
AND PC1.ram = PC2.ram
AND PC1.model < PC2.model;
```

"Find all pairs of PCs with same speed and ram sizes."

```
SELECT COUNT(hd)
FROM PC
GROUP BY hd
HAVING COUNT(model) > 2;
```

"How many hard disk sizes are built into more than two PCs?"

Strengths and Weaknesses

Strengths

- Strict schemata good for point queries, error prevention, compression, …
- Universal data model serving linked and unconnected data, all data types, …
- Consistency checking (ACID) with support for different consistency levels

Weaknesses

- Schemata need to be altered globally if certain records require additional attributes
- Impedance Mismatch:
  - Objects, structs, pointers vs. relations, records, attributes
  - Object-relational mapping (ORM) frameworks like ActiveRecord or Hibernate to the rescue
  - ➢ Complicates and slows data access; source for errors

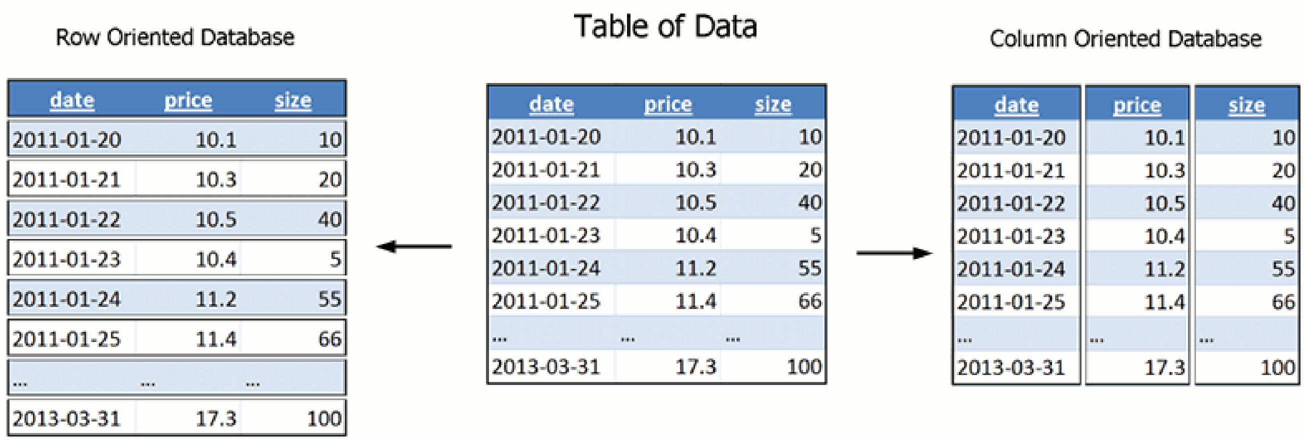
Storage Variations

Row-Based

- Store rows continuously
- See "Database Systems II" course

Column-Based

- Store columns continuously
- See "Trends and Concepts in Software Industry" course



date	price	size
2011-01-20	10.1	10
2011-01-21	10.3	20
2011-01-22	10.5	40
2011-01-23	10.4	5
2011-01-24	11.2	55
2011-01-25	11.4	66
...	...	...
2013-03-31	17.3	100

Storage Variations

Row-Based

- Store rows continuously

Column-Based

- Store columns continuously

Operation	Row-Based	Column-Based
Single column aggregation	Slow (full table scan)	Fast (single column scan)
Compression	Only NULL compression	Run length encoding
Column scans	Slow (skip irrelevant data)	Fast (one continuous read)
Insert/update of records	Fast (simply append)	Slow (many inserts; move data)
Single record point queries	Fast (one continuous read)	Slow (many seeks and reads)

Better **OLTP** performance

Better **OLAP** performance

Storage Variations

Row-Based

- Store rows continuously
- Examples by popularity:
  - Oracle
  - MySQL (open source)
  - Microsoft SQL Server
  - PostgreSQL (open source)
  - DB2
  - Microsoft Access
  - …

Many of these (e.g. Oracle and DB2) also support columnar data layouts.

Column-Based

- Store columns continuously
- Examples by popularity:
  - Teradata
  - SAP HANA
  - SAP Sybase IQ
  - Vertica
  - MonetDB (open source)
  - C-Store (open source)
  - …

CSV Files

Definition

- Relational structure but no constraints
  (no key-enforcement, data types, consistency checking, …)
- Operations: linear read and appending insert

Properties

- Encoding (ASCII, UTF-8, UTF-16, …)
- Value separator (usually semicolon ';', comma ',', or tab '	')
- Quote character (usually double-quotes '"')
- Escape character (usually slash '\')

Uses

- Data archiving and data exchange between heterogeneous systems
- File system storage engines (HDFS, NTFS, Ext3, …)
- Data dumping: sensor data, measurement data, scientific data, …

The Relational Data Model

CSV Files

Format Example

Name	Type	Equatorial diameter	Mass	Orbital radius	Orbital period	Rotation period	Confirmed moons	Rings	Atmosphere
Mercury	Terrestrial	0.382	0.06	0.47	0.24	58.64	0	no	minimal
Venus	Terrestrial	0.949	0.82	0.72	0.62	−243.02	0	no	CO_2, N_2
Earth	Terrestrial	1.000	1.00	1.00	1.00	1.00	1	no	N_2, O_2, Ar
Mars	Terrestrial	0.532	0.11	1.52	1.88	1.03	2	no	CO_2, N_2, Ar
Jupiter	Giant	11.209	317.8	5.20	11.86	0.41	67	yes	H_2, He
Saturn	Giant	9.449	95.2	9.54	29.46	0.43	62	yes	H_2, He
Uranus	Giant	4.007	14.6	19.22	84.01	−0.72	27	yes	H_2, He
Neptune	Giant	3.883	17.2	30.06	164.8	0.67	14	yes	H_2, He

represented as CSV File

WDC_planets.csv

```
"Name","Type","EquatorialDiameter","Mass","OrbitalRadius","OrbitalPeriod","RotationPeriod","ConfirmedMoons","Rings","Atmosphere"
"Mercury","Terrestrial","0.382","0.06","0.47","0.24","58.64","0","no","minimal"
"Venus","Terrestrial","0.949","0.82","0.72","0.62","-243.02","0","no","CO2 N2"
"Earth","Terrestrial","1.000","1.00","1.00","1.00","1.00","1","no","N2 O2 Ar"
"Mars","Terrestrial","0.532","0.11","1.52","1.88","1.03","2","no","CO2 N2 Ar"
"Jupiter","Giant","11.209","317.8","5.20","11.86","0.41","67","yes","H2 He"
"Saturn","Giant","9.449","95.2","9.54","29.46","0.43","62","yes","H2 He"
"Uranus","Giant","4.007","14.6","19.22","84.01","-0.72","27","yes","H2 He"
"Neptune","Giant","3.883","17.2","30.06","164.8","0.67","14","yes","H2 He"
```

CSV Files

Access Example

Java 1.7 using *au.com.bytecode.opencsv*

```
CSVWriter writer = null;
try {
  writer = new CSVWriter(
    new OutputStreamWriter(new FileOutputStream(dataFile, true), StandardCharsets.UTF_8), ',', '\"', '\\');

  for (String[] record : records) {
    writer.writeNext(record);
  }
  writer.close();
}
```

write

```
CSVReader reader = null;
try {
  reader = new CSVReader(
    new InputStreamReader(new FileInputStream(dataFile), StandardCharsets.UTF_8), ',', '\"');

  String[] record = null;
  while ((record = reader.readNext()) != null) {
    this.process(record);
  }
  reader.close();
}
```

read

CSV Files

Access Example

Java 1.8 using *au.com.bytecode.opencsv*

```
try (CSVWriter writer = new CSVWriter(
      new OutputStreamWriter(new FileOutputStream(dataFile, true), StandardCharsets.UTF_8), ',', '\"', '\\')) {
   Arrays.stream(records).forEach(record -> writer.writeNext(record));
}
```

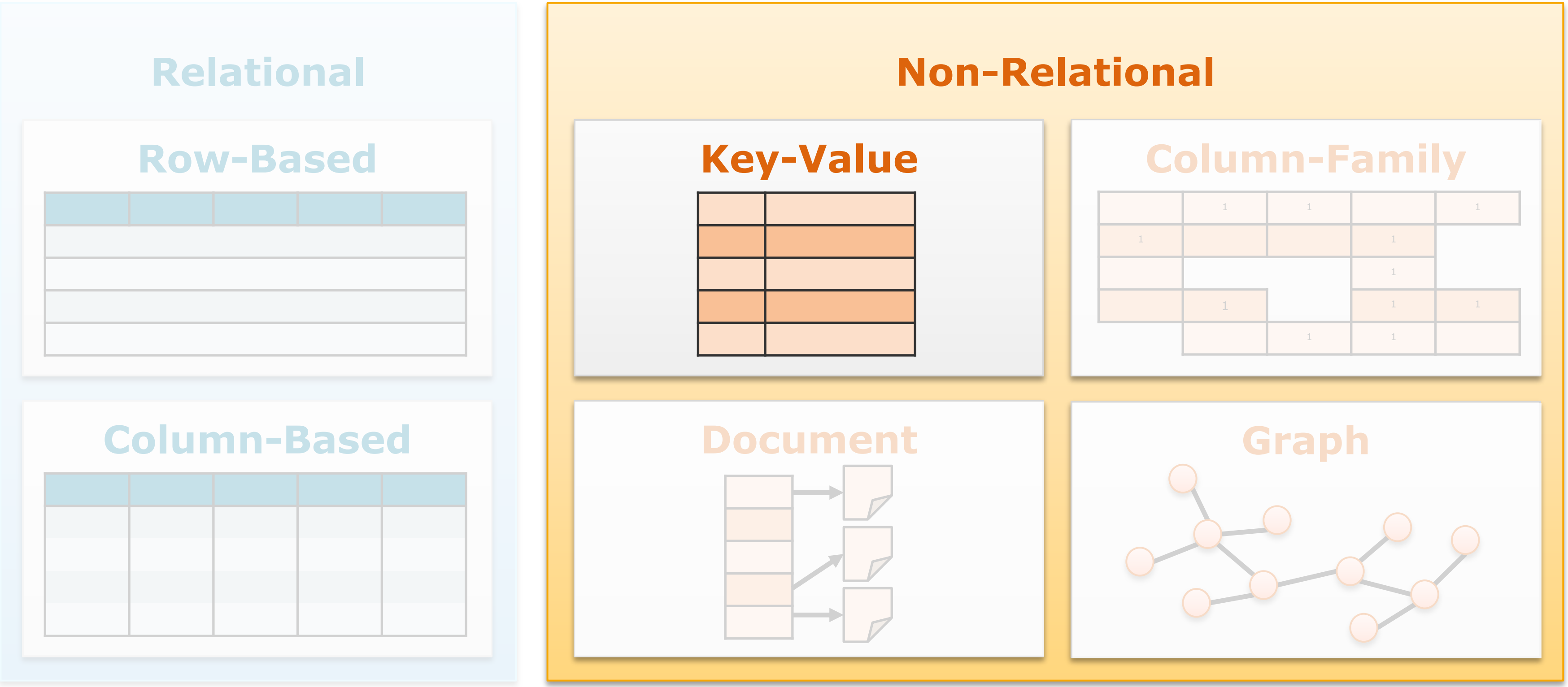
write

```
try (CSVReader reader = new CSVReader(
      new InputStreamReader(new FileInputStream(dataFile), StandardCharsets.UTF_8), ',', '\"')) {
   reader.forEach(record -> this.process(record));
}
```

read

Overview

Relational and Non-Relational Data Models



Popular Key-Value Stores

HPI Hasso Plattner Institut

https://db-engines.com/en/ranking

Rank Aug 2017	Rank Jul 2017	Rank Aug 2016	DBMS	Database Model	Score Aug 2017	Score Jul 2017	Score Aug 2016
1.	1.	1.	Redis	Key-value store	121.90	+0.38	+14.57
2.	2.	2.	Memcached	Key-value store	29.96	+1.43	+2.27
3.	↑ 4.	↑ 13.	Microsoft Azure Cosmos DB	Multi-model	9.42	+1.72	+6.87
4.	↓ 3.	4.	Hazelcast	Key-value store	8.73	-0.18	+1.41
5.	5.	↓ 3.	Riak KV	Key-value store	7.26	-0.10	-4.29
6.	6.	↓ 5.	Ehcache	Key-value store	6.93	-0.22	+0.41
7.	7.	↓ 6.	OrientDB	Multi-model	5.67	+0.10	-0.30
8.	8.	↓ 7.	Aerospike	Key-value store	4.36	+0.15	-0.17
9.	↑ 11.	↑ 15.	ArangoDB	Multi-model	2.92	-0.04	+0.99
10.	↑ 12.	↓ 9.	Oracle Berkeley DB	Multi-model	2.90	+0.03	-0.19
11.	↓ 9.	↑ 12.	Oracle NoSQL	Key-value store	2.88	-0.20	+0.22
12.	↓ 10.	↓ 11.	Oracle Coherence	Key-value store	2.78	-0.21	-0.01
13.	13.	↑ 14.	Caché	Multi-model	2.71	-0.01	+0.75
14.	↑ 15.		Ignite	Key-value store	2.60	+0.21	
15.	↓ 14.	↑ 16.	Infinispan	Key-value store	2.57	-0.05	+0.64
16.	↑ 17.	↑ 17.	LevelDB	Key-value store	2.19	+0.02	+0.50
17.	↓ 16.	↓ 10.	Amazon SimpleDB	Key-value store	2.18	-0.16	-0.66
18.	18.	18.	GridGain	Key-value store	1.44	+0.11	+0.60
19.	19.	19.	RocksDB	Key-value store	1.34	+0.12	+0.78

In Akka!

In Kafka!

Definition

Some implementations do support this and some don't.

1. Structure

- Index (e.g. hash map): (large, distributed) key-value data structure

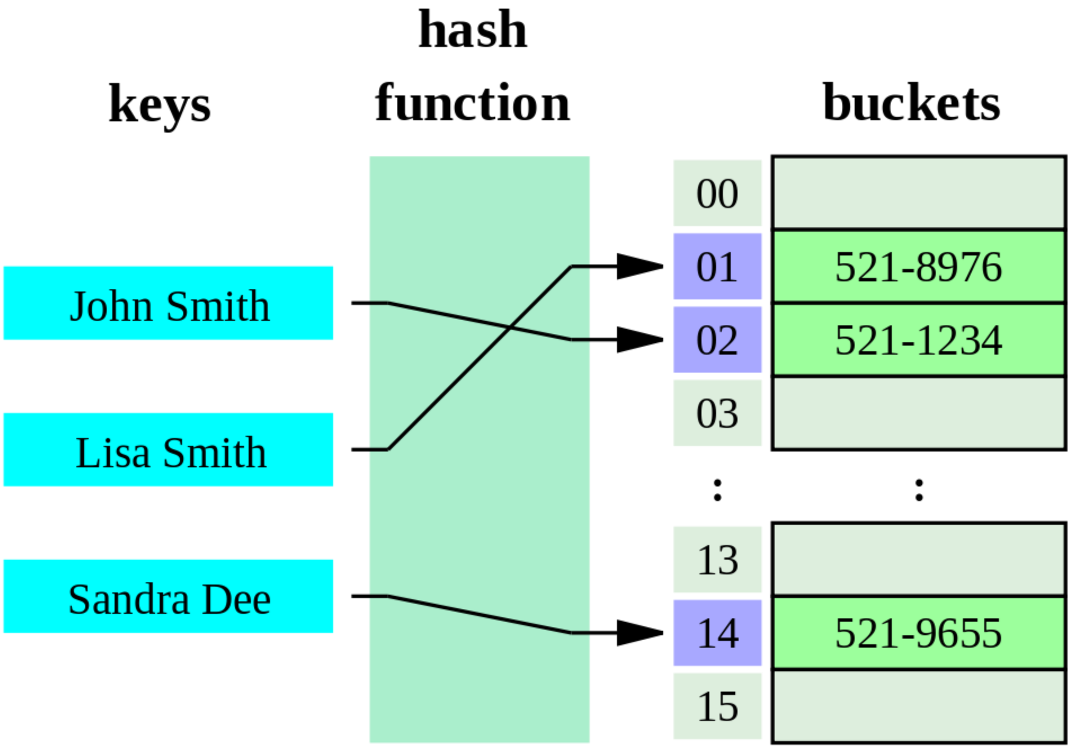
2. Constraints

- Each value is associated with a unique key.

3. Operations

- Store a key-value pair.
- Retrieve a value by key.
- Remove a key-value mapping.

Example


keys
hash function
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
13
14
15
521-8976
521-1234
521-9655

Querying: Redis API

Redis

- In-memory key-value store with file persistence on disk
- Supports five data structures for values:
  - Strings: byte arrays that may represent actual strings or integers, binary serialized objects, …
  - Hashes: dictionaries that map secondary keys to strings
  - Lists: sequences of strings that support insert, append, pop, push, trim, and many further operations
  - Sets: duplicate free collections of strings that support set operations such as diff, union, intersect, …
  - Ordered sets: duplicate free, sorted collections of strings that use explicitly defined scores for sorting and support range operations

Querying: Redis API – Examples

Redis API

- Strings:
  SET hello “hello world”
  GET hello
  → “hello world”
  SET users:goku {race: 'sayan', power: 9001}
  GET users:goku
  → {race: 'sayan', power: 9001}
- Hashes:
  HSET users:goku race 'sayan'
  HSET users:goku power 9001
  HGET users:goku power
  → 9001

“<group>:<entity>” is a naming convention.

- Lists:
  LPUSH mylist a // [a]
  LPUSH mylist b // [b,a]
  RPUSH mylist c // [b,a,c]
  LRANGE mylist 0 1
  → b, a
  RPOP mylist
  → c
- Sets:
  SADD friends:lisa paul
  SADD friends:lisa duncan
  SADD friends:paul duncan
  SADD friends:paul gurney
  SINTER friends:lisa friends:paul
  → duncan
- Ordered sets:
  ZADD lisa 8 paul
  ZADD lisa 7 duncan
  ZADD lisa 2 faradin
  ZRANGEBYSCORE lisa 5 8
  → duncan
  → paul

Strengths and Weaknesses

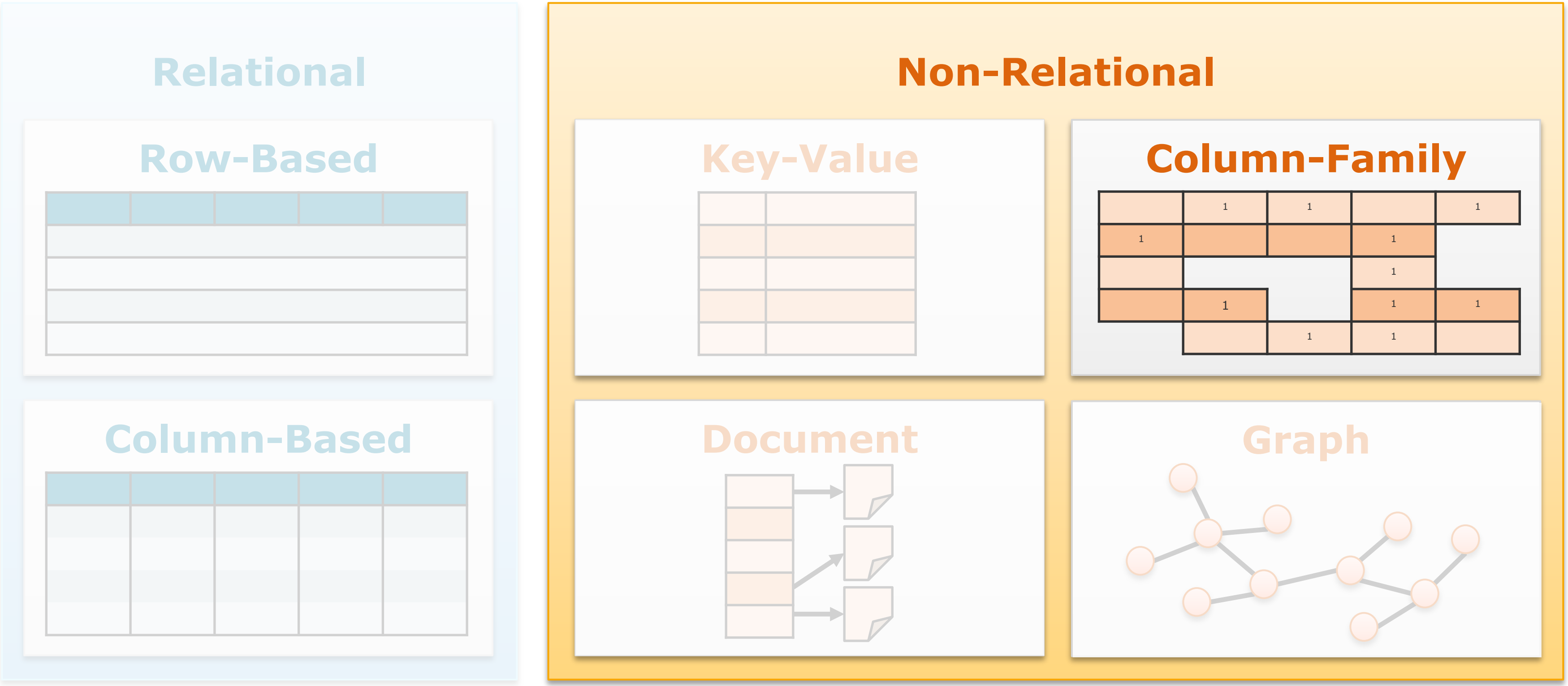
Strengths

- Efficient storage: fast inserts of key-value pairs
- Efficient retrieval: fast point queries, i.e., value look-ups
- Key-value pairs are easy to distribute across multiple machines
- Key-value pairs can be replicated for fault-tolerance and load balancing

Weaknesses

- No filtering, aggregation, or joining of values/entries
  - ➢ Must be done by the application (or cluster computing framework!)
- (Usually) no parsing of complex values; must be done by the application
  - ➢ Must be done by the application (or cluster computing framework!)

Relational and Non-Relational Data Models



Popular Column-Family Stores

Rank			DBMS	Database Model	Score		
Aug 2017	Jul 2017	Aug 2016			Aug 2017	Jul 2017	Aug 2016
1.	1.	1.	Cassandra	Wide column store	126.72	+2.60	-3.52
2.	2.	2.	HBase	Wide column store	63.52	-0.10	+8.01
3.	3.	↑ 4.	Microsoft Azure Cosmos DB	Multi-model	9.42	+1.72	+6.87
4.	4.	↓ 3.	Accumulo	Wide column store	3.66	+0.17	+0.31
5.	5.		Microsoft Azure Table Storage	Wide column store	2.96	-0.13	
6.	6.	6.	Google Cloud Bigtable	Wide column store	0.70	-0.11	+0.41
7.	↑ 8.		MapR-DB	Multi-model	0.51	-0.04	
8.	↑ 9.	↓ 7.	Sqrrl	Multi-model	0.50	+0.01	+0.24
9.	↑ 10.	↓ 8.	ScyllaDB	Wide column store	0.39	-0.00	+0.35
10.			Alibaba Cloud Table Store	Wide column store	0.01		

https://db-engines.com/en/ranking

Definition

1. Structure

- Multi-dimensional index (e.g. multi-dimensional hash map)
  - (large, distributed) key-value data structure that uses a hierarchy of up to three keys for one typed value
- Conceptually equivalent to sparse relational tables, i.e., each row supports arbitrary subsets of attributes.

> For this reason, they are also called "Wide Column Stores".

2. Constraints

- Each value is associated with a unique key.
- Hierarchy of keys is a tree.
- Integrity constraints: keys, foreign-keys, cluster-keys (for distribution), …

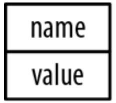
3. Operations

- At least: store key-value pair; retrieve value by key; remove key-value pair
- Usually: relational algebra support without joins (with own SQL dialect)

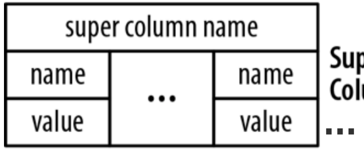
The Column-Family Data Model

Example

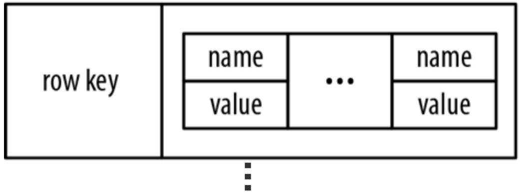
©https://neo4j.com/blog/aggregate-stores-tour/



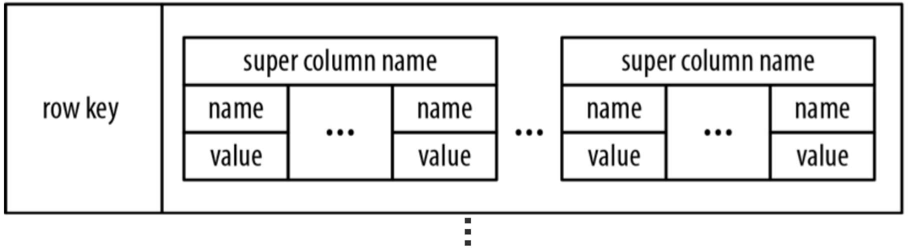
Column = key-value pair



Super Column = key-hashmap pair

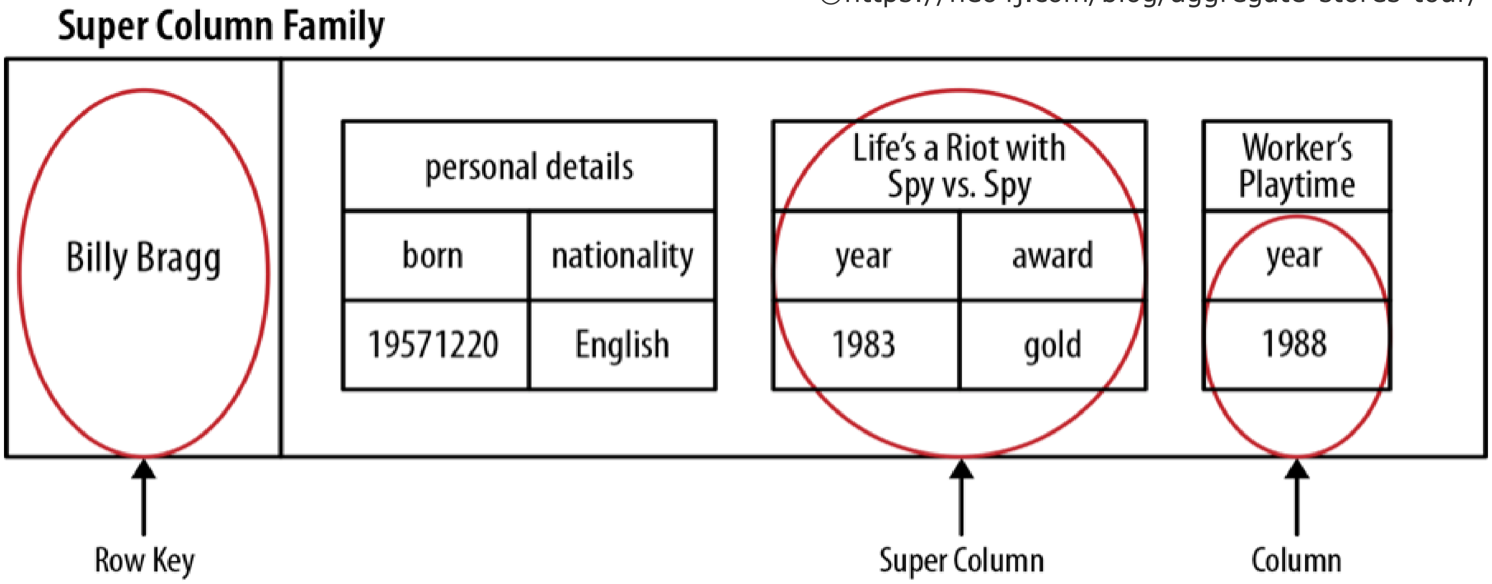


Column Family = Map<RowKey, SortedMap<ColumnKey, ColumnValue>>
≈ relational table



Super Column Family = Map<RowKey,
SortedMap<SuperColumnKey,
SortedMap<ColumnKey, ColumnValue>>>

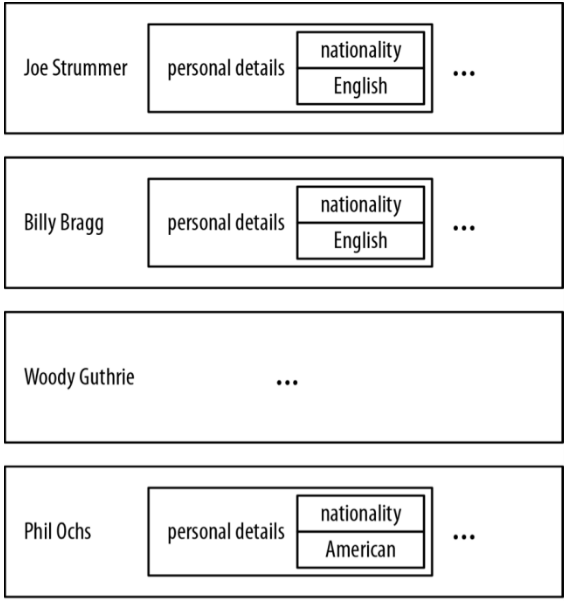
Example 1



Hierarchy of keys enables:

- Flexible schemata (column names model attributes and row keys records)
- Value groupings (by super column names and row keys)

The Column-Family Data Model
Example 2



Analogy:

Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

Hierarchy of keys enables:

- Flexible schemata (column names model attributes and row keys records)
- Value groupings (by super column names and row keys)

Querying: CQL

HPI Hasso Plattner Institut

Cassandra Query Language CQL …

- is an SQL dialect (same syntax).
- supports all DML and DDL functionalities.
- does not support:
  - joins, group by, triggers, cursors, transactions, or (stored) procedures
  - OR and NOT logical operators (only AND)
  - subqueries
- makes, inter alia, the following restrictions:
  - WHERE conditions *should* be applied only on columns with an index
  - timestamps are comparable only with the equal operator (not <,>,<>)
  - UPDATE statements only work with a primary key
    (they do not work based on other columns or as mass update)
  - INSERT overrides existing records, UPDATE creates non-existing ones

Querying: CQL – Examples

Schema:

first key attribute(s) = **partition key** (determines which node stores the data)

- **Playlists(id, song_order, album, artist, song_id, title)**

further key attribute(s) = **cluster key** (keys within a partition/node)

Query:

```
SELECT *
FROM Playlists
WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
ORDER BY song_order DESC
LIMIT 4;
```

= key attribute (id)

= key attribute (song_order)

Result:

id	song_order	album	artist	song_id	title
62c36092...	4	No One Rides for Free	Fu Manchu	7db1a490...	Ojo Rojo
62c36092...	3	Roll Away	Back Door Slam	2b09185b...	Outside Woman Blues
62c36092...	2	We Must Obey	Fu Manchu	8a172618...	Moving in Stereo
62c36092...	1	Tres Hombres	ZZ Top	a3e63f8f...	La Grange

Querying: CQL – Examples

SQL:

```
CREATE DATABASE myDatabase;
```

```
SELECT *
FROM myTable
WHERE myField > 5000
AND myField < 100000;
```

CQL:

```
CREATE KEYSPACE myDatabase
WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 1};
```

```
SELECT *
FROM myTable
WHERE myField > 5000
AND myField < 100000
ALLOW FILTERING;
```

Otherwise:
Bad Request: Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute it despite the performance unpredictability, use ALLOW FILTERING.

Strengths and Weaknesses

Strengths

- Efficient storage: fast inserts of data items
- Efficient retrieval: fast point queries, i.e., value look-ups
- Data structure is easy to distribute across multiple machines
- Data structure can be replicated for fault-tolerance and load balancing
- Flexible schemata

Weaknesses

- No join and limited filtering support (filtering might also be super slow)
  - Must be done by the application (or cluster computing framework!)
- Multi-key structure groups values to entities but general groupings and aggregations are not supported
- Non-point queries, i.e., those that read more than one mapping, are costly

Strengths and Weaknesses

"Writes are cheap. Write everything the way you want to read it."
If you have people and addresses and you need to read people and their addresses, then store people and addresses additionally(!) in one column family.
"Not just de-normalize, forget about normalization all together."

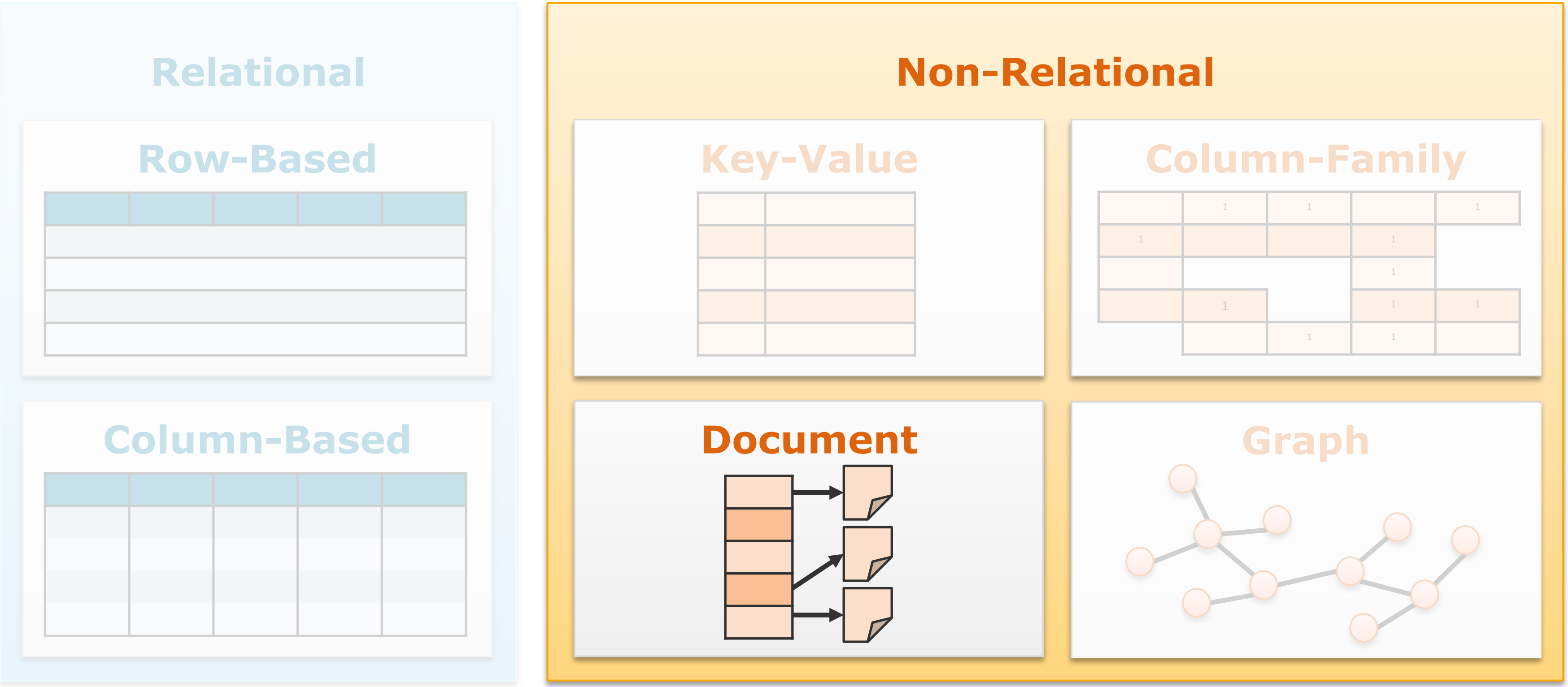
Alex Meng
https://medium.com/@alexbmeng/cassandra-query-language-cql-vs-sql-7f6ed7706b4c

Weaknesses

- No join and limited filtering support (filtering might also be super slow)
  - Must be done by the application (or cluster computing framework!)
- Multi-key structure groups values to entities but general groupings and aggregations are not supported
- Non-point queries, i.e., those that read more than one mapping, are costly

Overview

Relational and Non-Relational Data Models



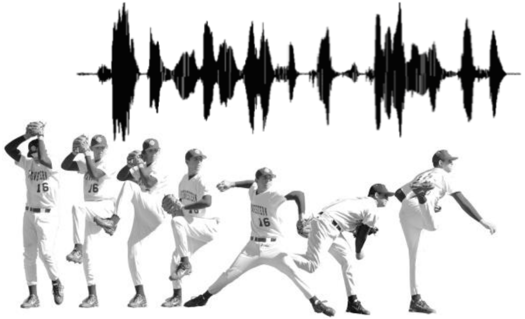
The Document Data Model

Natural Document Data

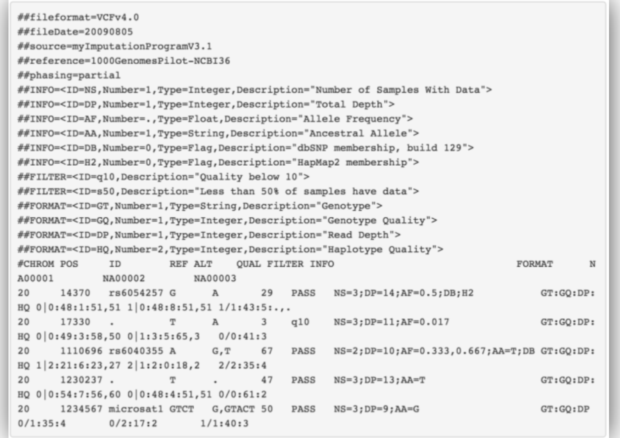
Digital Documents

```
<!DOCTYPE html>
<html>
  <head>
    <title>Canvas-Rotation</title>
    <meta charset="UTF-8" />
    <style>
      #square {
        border: 1px solid black;
        transform: scale(10) rotate(3deg) translateX(0px);
        -moz-transform: scale(10) rotate(3deg) translateX(0px);
      }

      .box {
        transition-duration: 2s;
        transition-property: transform;
        transition-timing-function: linear;
      }
    </style>
  </head>
  <body>
    <canvas id="square" width="200" height="200"></canvas>
    <script>
      var canvas = document.createElement('canvas');
      canvas.width = 200;
      canvas.height = 200;

      var image = new Image();
      image.src = 'images/card.png';
      image.width = 114;
      image.height = 158;
      image.onload = window.setInterval(function() {
        rotation();
      }, 1000/60);
    </script>
  </body>
</html>
```

Web Pages

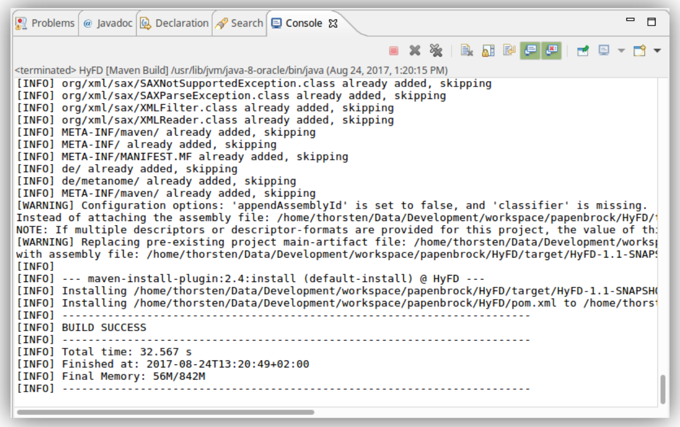


Structured Data



```
##fileformat=VCFv4.0
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=1000GenomesPilot-NCBI36
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=.,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
#CHROM POS     ID        REF ALT    QUAL FILTER INFO                              FORMAT      N
A00001         NA00002          NA00003
20     14370   rs6054257 G      A       29   PASS   NS=3;DP=14;AF=0.5;DB;H2           GT:GQ:DP:
HQ 0|0:48:1:51,51 1|0:48:8:51,51 1/1:43:5:.,.
20     17330   .         T      A       3    q10    NS=3;DP=11;AF=0.017               GT:GQ:DP:
HQ 0|0:49:3:58,50 0|1:3:5:65,3   0/0:41:3
20     1110696 rs6040355 A      G,T     67   PASS   NS=2;DP=10;AF=0.333,0.667;AA=T;DB GT:GQ:DP:
HQ 1|2:21:6:23,27 2|1:2:0:18,2   2/2:35:4
20     1230237 .         T      .       47   PASS   NS=3;DP=13;AA=T                   GT:GQ:DP:
HQ 0|0:54:7:56,60 0|0:48:4:51,51 0/0:61:2
20     1234567 microsat1 GTCT   G,GTACT 50   PASS   NS=3;DP=9;AA=G                    GT:GQ:DP
0/1:35:4       0/2:17:2         1/1:40:3
```

Scientific Data Formats




```
<terminated> HyFD [Maven Build] /usr/lib/jvm/java-8-oracle/bin/java (Aug 24, 2017, 1:20:15 PM)
[INFO] org/xml/sax/SAXNotSupportedException.class already added, skipping
[INFO] org/xml/sax/SAXParseException.class already added, skipping
[INFO] org/xml/sax/XMLFilter.class already added, skipping
[INFO] org/xml/sax/XMLReader.class already added, skipping
[INFO] META-INF/maven/ already added, skipping
[INFO] META-INF/ already added, skipping
[INFO] META-INF/MANIFEST.MF already added, skipping
[INFO] de/ already added, skipping
[INFO] de/metanome/ already added, skipping
[INFO] META-INF/maven/ already added, skipping
[WARNING] Configuration options: 'appendAssemblyId' is set to false, and 'classifier' is missing.
Instead of attaching the assembly file: /home/thorsten/Data/Development/workspace/papenbrock/HyFD/
NOTE: If multiple descriptors or descriptor-formats are provided for this project, the value of th
[WARNING] Replacing pre-existing project main-artifact file: /home/thorsten/Data/Development/works
with assembly file: /home/thorsten/Data/Development/workspace/papenbrock/HyFD/target/HyFD-1.1-SNAP
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ HyFD ---
[INFO] Installing /home/thorsten/Data/Development/workspace/papenbrock/HyFD/target/HyFD-1.1-SNAPSH
[INFO] Installing /home/thorsten/Data/Development/workspace/papenbrock/HyFD/pom.xml to /home/thors
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 32.567 s
[INFO] Finished at: 2017-08-24T13:20:49+02:00
[INFO] Final Memory: 56M/842M
[INFO] ------------------------------------------------------------------------
```

Log Data

Popular Document Stores

Rank Aug 2017	Rank Jul 2017	Rank Aug 2016	DBMS	Database Model	Score Aug 2017	Score Jul 2017	Score Aug 2016
1.	1.	1.	MongoDB	Document store	330.50	-2.27	+12.01
2.	2.	↑ 3.	Amazon DynamoDB	Document store	37.62	+1.16	+11.02
3.	3.	↓ 2.	Couchbase	Document store	32.97	-0.05	+5.57
4.	4.	4.	CouchDB	Document store	21.34	-0.81	+0.28
5.	5.	5.	MarkLogic	Multi-model	12.50	+0.07	+2.46
6.	6.	↑ 10.	Microsoft Azure Cosmos DB	Multi-model	9.42	+1.72	+6.87
7.	7.	↓ 6.	OrientDB	Multi-model	5.67	+0.10	-0.30
8.	8.	↓ 7.	RethinkDB	Document store	4.88	-0.07	+0.12
9.	↑ 10.		Firebase Realtime Database	Document store	4.51	+0.44	
10.	↓ 9.	↓ 8.	Cloudant	Document store	4.31	-0.33	-0.20
11.	11.	↑ 17.	Google Cloud Datastore	Document store	3.49	-0.40	+2.41
12.	12.	↓ 9.	RavenDB	Document store	3.27	-0.24	-1.17
13.	13.	↓ 12.	Apache Drill	Multi-model	3.21	-0.01	+0.83
14.	14.	↓ 13.	PouchDB	Document store	2.99	-0.07	+0.64
15.	15.	↓ 14.	ArangoDB	Multi-model	2.92	-0.04	+0.99
16.	16.	↓ 15.	CloudKit	Document store	2.20	-0.21	+0.37
17.	17.	↓ 11.	Virtuoso	Multi-model	1.98	-0.01	-0.42
18.	18.	↓ 16.	Datameer	Document store	1.73	+0.11	-0.01
19.	19.	↓ 18.	Mnesia	Document store	1.17	-0.21	+0.12

https://db-engines.com/en/ranking

Definition

1. Structure

- Index: (large, distributed) key-value data structure
- Documents: values are documents or collections of documents that (usually) contain hierarchical data.
  - XML, JSON, RDF, HTML, …

2. Constraints

- Each value/document is associated with a unique key.

3. Operations

- Store a key-value pair.
- Retrieve a value by key.
- Remove a key-value mapping.
- Update a value of a key.

> Document stores are often considered to be schemaless, but since the applications usually assume some kind of structure they are rather schema-on-read in contrast to schema-on-write.

Definition

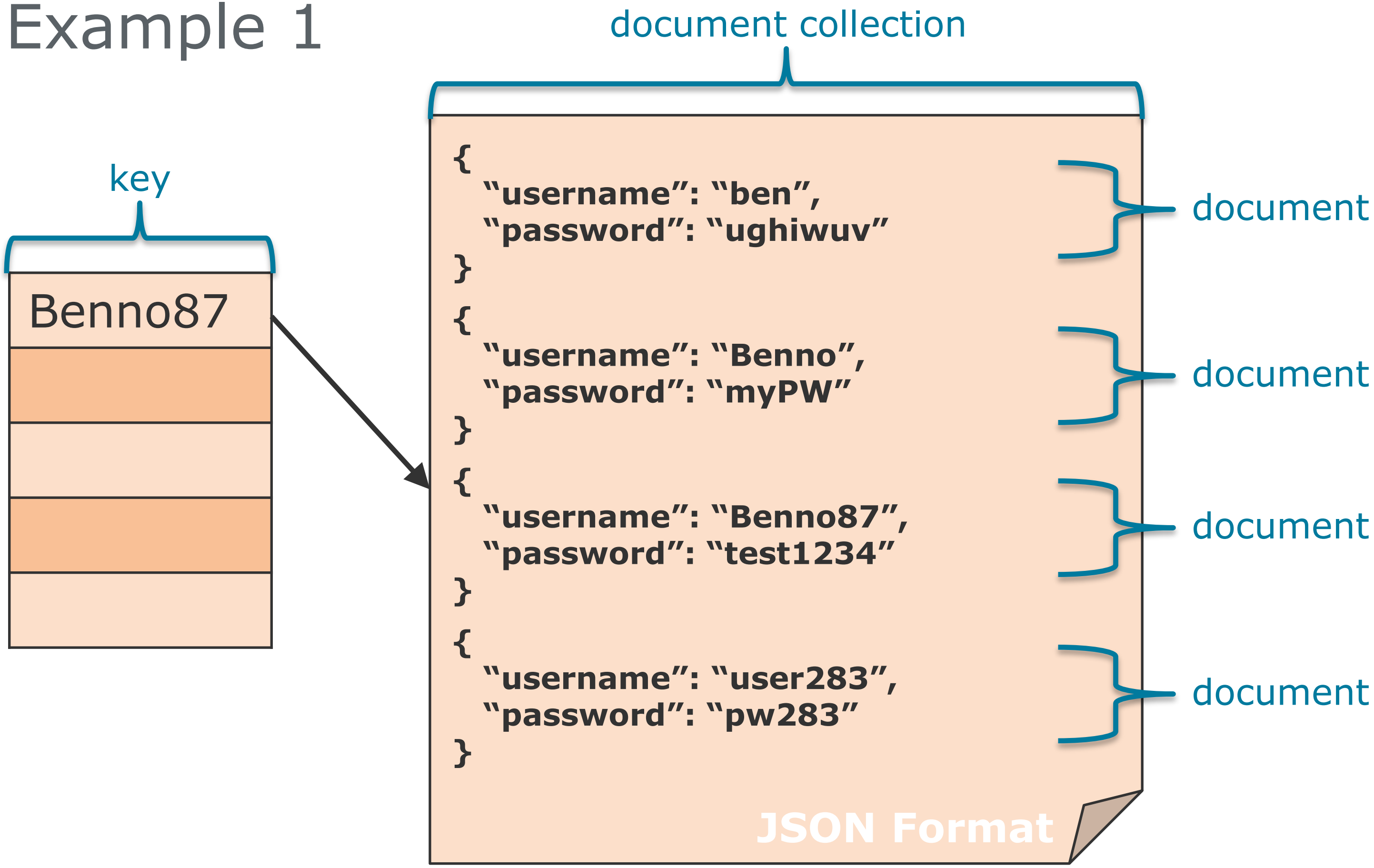
Relational data model

Highly-structured table organization with rigidly-defined data formats and record structure.

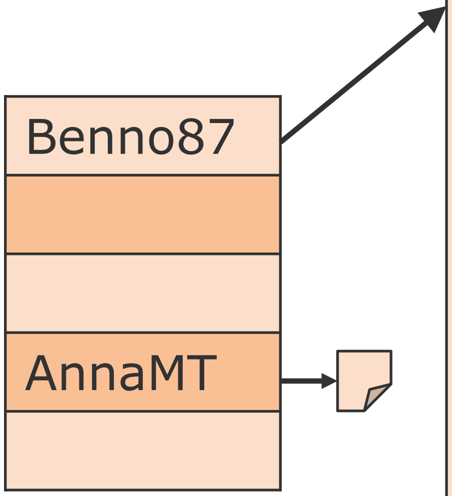
Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.

Example 1


document collection
key
Benno87
{
"username": "ben",
"password": "ughiwuv"
}
{
"username": "Benno",
"password": "myPW"
}
{
"username": "Benno87",
"password": "test1234"
}
{
"username": "user283",
"password": "pw283"
}
document
document
document
document
JSON Format

Example 2



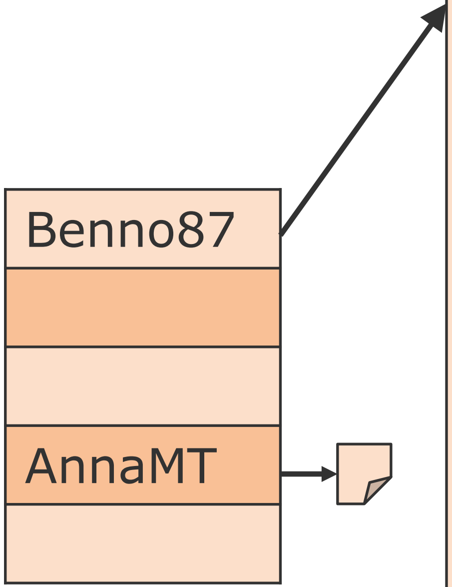

```
{
  "_id": 1,
  "username": "ben",
  "password": "ughiwuv",
  "contact": {
        "phone": 0331-1781471,
        "email": "ben87@gmx.de",
        "skype": "benno.miller"
     },
  "access": {
        "level": 3,
        "group": "user"
     },
  "supervisor": {
        "$ref": "AnnaMT",
        "$id": 2,
        "$db": "users"
     }
}
```

JSON Format

- "_id": 1 → document ID
- "contact" → embedded subdocument
- "access" → embedded subdocument
- "supervisor" → document reference

The Document Data Model

Example 3




```xml
<_id>1</_id>
<username>ben</username>
<password>ughiwuv</password>
<contact>
    <phone>0331-1781254</phone>
    <email>ben87@gmx.de</email>
    <skype>benno.miller</skype>
</contact>
<access>
    <level>3</level>
    <group>user</group>
</access>
<supervisor>
    <ref>AnnaMT</ref>
    <id>2</id>
    <db>users</db>
</supervisor>
```

XML Format

Note that relational databases also support hierarchical data types (e.g. XML and JSON) in their attributes.

Strengths and Weaknesses

Strengths

- Efficient storage: fast inserts of key-value pairs
- Efficient retrieval: fast point queries, i.e., document (collection) look-ups
- Document (collections) are easy to distribute across multiple machines
- Document (collections) can be replicated for fault-tolerance and load balancing
- Flexible document formats: self-describing documents that may use different formats

Weaknesses

- (Usually) developers need to explicitly/manually plan for distribution of data across instances (key-value and column-family stores do this automatically)
- Updates to documents are expensive if they alter encoding or size

Querying: MongoDB API

MongoDB …

- is a free and open-source document-oriented DBMS.
- uses JSON-like documents with schemata and integrity constraints (keys).

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row/record	document
column/attribute	field
index	index
table join	$lookup, embedded document
primary key (any column)	primary key (always the _id filed)
aggregation (group by)	aggregation pipeline

Querying: MongoDB API – Examples

Create/Drop

Document:

```
{
  _id: 1,
  user_id: "abc123",
  age: 55,
  status: 'A'
}
```

SQL	MongoDB
`CREATE TABLE people ( id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1), PRIMARY KEY (id) )`	`db.people.insertOne( { user_id: "abc123", age: 55, status: "A" } )`
`DROP TABLE people`	`db.people.drop()`

SQL:

```
CREATE TABLE people (
    id MEDIUMINT NOT NULL
        AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)
```

MongoDB:

```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
} )
```

```
DROP TABLE people
```

```
db.people.drop()
```

First insert automatically creates
the document collection "people" but no schema!

https://docs.mongodb.com/manual/

Querying: MongoDB API – Examples

Alter

Document:

```
{
  _id: 1,
  user_id: "abc123",
  age: 55,
  status: 'A'
}
```

SQL	MongoDB
`ALTER TABLE people ADD join_date DATETIME`	`db.people.updateMany( { }, { $set: { join_date: new Date() } } )`
`ALTER TABLE people DROP COLUMN join_date`	`db.people.updateMany( { }, { $unset: { "join_date": "" } } )`

SQL:

```
ALTER TABLE people
ADD join_date DATETIME
```

MongoDB:

```
db.people.updateMany(
    { },
    { $set: { join_date: new Date() } }
)
```

"$" introduce operators (= functions)

SQL:

```
ALTER TABLE people
DROP COLUMN join_date
```

MongoDB:

```
db.people.updateMany(
    { },
    { $unset: { "join_date": "" } }
)
```

Collections do not describe or enforce the structure of their documents, i.e., no structural alteration at collection level.

But: $set and $unset can be used for bulk updates.

https://docs.mongodb.com/manual/

Querying: MongoDB API – Examples

Insert, Update and Delete

Document:

```
{
  _id: 1,
  user_id: “abc123”,
  age: 55,
  status: 'A'
}
```

SQL	MongoDB
`INSERT INTO people(user_id, age, status) VALUES ("bcd001", 45, "A")`	`db.people.insertOne( { user_id: "bcd001", age: 45, status: "A" } )`
`UPDATE people SET status = "C" WHERE age > 25`	`db.people.updateMany( { age: { $gt: 25 } }, { $set: { status: "C" } } )`
`DELETE FROM people WHERE status = "D"`	`db.people.deleteMany( { status: "D" } )`

https://docs.mongodb.com/manual/

Querying: MongoDB API – Examples

Select

Document:

```
{
  _id: 1,
  user_id: “abc123”,
  age: 55,
  status: 'A'
}
```

SQL	MongoDB
SELECT * FROM people	db.people.find()
SELECT user_id, status FROM people WHERE status = "A"	db.people.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM people WHERE status = "A" OR age = 50	db.people.find({ $or: [{ status: "A" } , { age: 50 }] })
SELECT * FROM people WHERE age > 25 AND age <= 50	db.people.find({ age: { $gt: 25, $lte: 50 } })

Always selected if not deselected.

https://docs.mongodb.com/manual/

Querying: MongoDB API – Examples

Aggregate

Document:

```
{
  _id: 1,
  user_id: "abc123",
  age: 55,
  status: 'A'
}
```

SQL

```
SELECT COUNT(*)
FROM people
WHERE age > 30
```

MongoDB

```
db.people.count( { age: { $gt: 30 } } )
```

```
db.sales.aggregate(
  [ { $group : {
        _id : { month: { $month: "$date" },
                year: { $year: "$date" } },
        totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
        averageQuantity: { $avg: "$quantity" },
        count: { $sum: 1 } } } ] )
```

MongoDB's **aggregation pipeline**:
We can add additional operators like $match after the $group to further refine the result.

Group the documents by month and year and calculate the **total price,** the **average quantity,** and the **count of documents** per group.

https://docs.mongodb.com/manual/

The Document Data Model

Querying: MongoDB API –

Join

```
db.orders.aggregate(
   [ { $lookup: {
          from: "inventory",
          localField: "item",
          foreignField: "sku",
          as: "inventory_docs" } } ] )
```

orders

```
{ "_id" : 1, "item" : "abc", "price" : 12, "quantity" : 2 }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1 }
{ "_id" : 3 }
```

inventory

```
{ "_id" : 1, "sku" : "abc", description: "product 1", "instock" : 120 }
{ "_id" : 2, "sku" : "def", description: "product 2", "instock" : 80 }
{ "_id" : 3, "sku" : "ijk", description: "product 3", "instock" : 60 }
{ "_id" : 4, "sku" : "jkl", description: "product 4", "instock" : 70 }
{ "_id" : 5, "sku": null, description: "Incomplete" }
{ "_id" : 6 }
```

inventory_docs

```
{
  "_id" : 1,
   "item" : "abc",
  "price" : 12,
  "quantity" : 2,
  "inventory_docs" : [
    { "_id" : 1, "sku" : "abc", description: "product 1", "instock" : 120 }
  ]
}
{
  "_id" : 2,
  "item" : "jkl",
  "price" : 20,
  "quantity" : 1,
  "inventory_docs" : [
    { "_id" : 4, "sku" : "jkl", "description" : "product 4", "instock" : 70 }
  ]
}
{
  "_id" : 3,
  "inventory_docs" : [
    { "_id" : 5, "sku" : null, "description" : "Incomplete" },
    { "_id" : 6 }
  ]
}
```


https://docs.mongodb.com/manual/

The Document Data Model

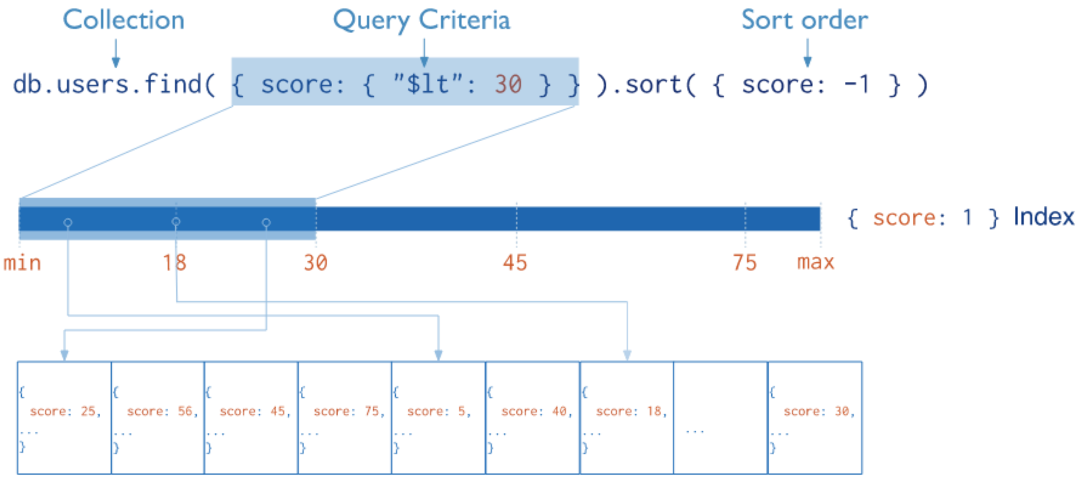
Querying: MongoDB API – Examples

Index

SQL	MongoDB
`CREATE INDEX idx_user_id_asc ON people(user_id)`	`db.people.createIndex( { user_id: 1 } )`

Document:

```
{
  _id: 1,
  user_id: "abc123",
  age: 55,
  status: 'A'
}
```



For Indexes, the DBMS maintains the document offsets in collections so that indexes work similar to indexes in relational databases.

https://docs.mongodb.com/manual/

Querying: MongoDB API – Examples

Very rich API

Document:

```
{
  _id: 1,
  user_id: "abc123",
  age: 55,
  status: 'A'
}
```

Visit the manual!

https://docs.mongodb.com/manual/

- db.collection.aggregate()
- db.collection.bulkWrite()
- db.collection.copyTo()
- db.collection.count()
- db.collection.createIndex()
- db.collection.dataSize()
- db.collection.deleteOne()
- db.collection.deleteMany()
- db.collection.distinct()
- db.collection.drop()
- db.collection.dropIndex()
- db.collection.dropIndexes()
- db.collection.ensureIndex()
- db.collection.explain()
- db.collection.find()
- db.collection.findAndModify()
- db.collection.findOne()
- db.collection.findOneAndDelete
- db.collection.findOneAndReplac
- db.collection.findOneAndUpdat

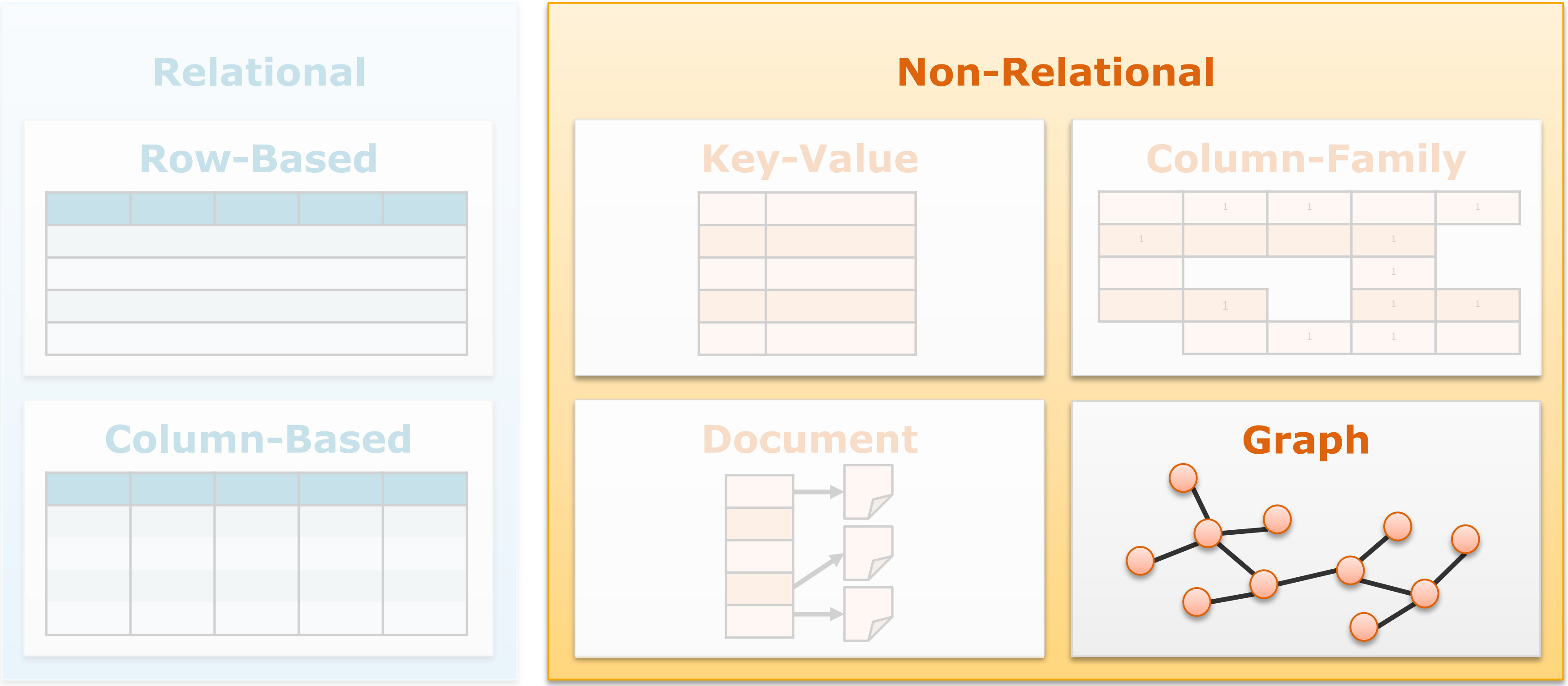
- cursor.batchSize()
- cursor.close()
- cursor.collation()
- cursor.comment()
- cursor.count()
- cursor.explain()
- cursor.forEach()
- cursor.hasNext()
- cursor.hint()
- cursor.itcount()
- cursor.limit()
- cursor.map()
- cursor.max()
- cursor.maxScan()
- cursor.maxTimeMS()
- cursor.min()
- cursor.next()
- cursor.noCursorTimeout()
- cursor.objsLeftInBatch()
- cursor.pretty()

- db.adminCommand()
- db.cloneCollection()
- db.cloneDatabase()
- db.commandHelp()
- db.copyDatabase()
- db.createCollection()
- db.createView()
- db.currentOp()
- db.dropDatabase()
- db.eval()
- db.fsyncLock()
- db.fsyncUnlock()
- db.getCollection()
- db.getCollectionInfos()
- db.getCollectionNames()
- db.getLastError()
- db.getLastErrorObj()
- db.getLogComponents()
- db.getMongo()
- db.getName()

- sh.addShard()
- sh.addShardTag()
- sh.addShardToZone()
- sh.addTagRange()
- sh.disableBalancing()
- sh.enableBalancing()
- sh.enableSharding()
- sh.getBalancerHost()
- sh.getBalancerState()
- sh.removeTagRange()
- sh.removeRangeFromZone()
- sh.help()
- sh.isBalancerRunning()
- sh.moveChunk()
- sh.removeShardTag()
- sh.removeShardFromZone()
- sh.setBalancerState()
- sh.shardCollection()
- sh.splitAt()
- sh.splitFind()

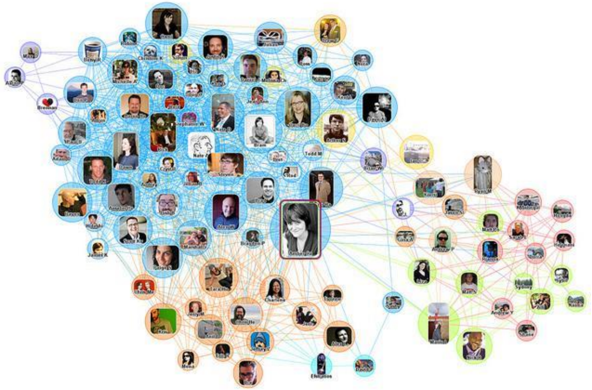
Overview

Relational and Non-Relational Data Models

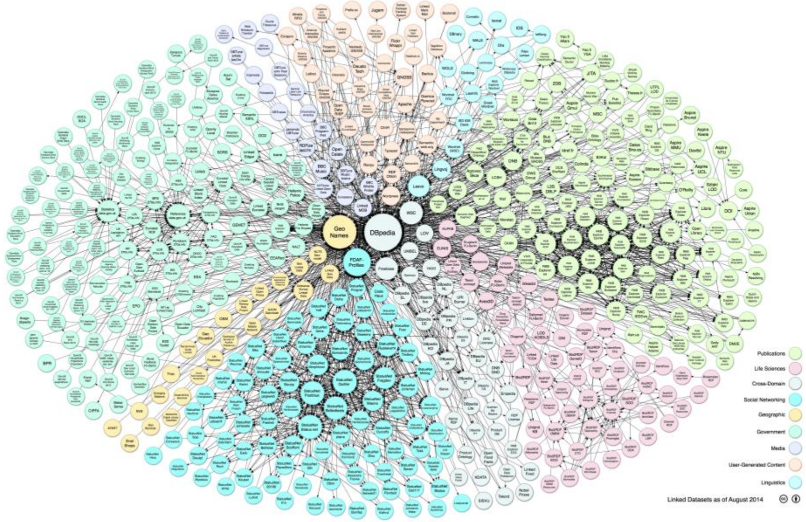


The Graph Data Model

Natural Graph Data

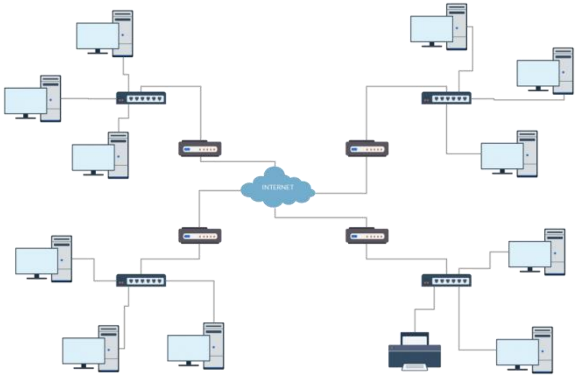


Social Graphs

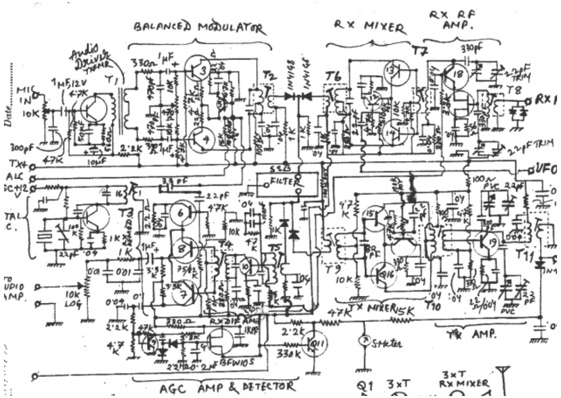


Linked Open Data

Road and Rail Maps



Network Topologies



Circuit Diagrams

The Graph Data Model

Popular Graph DBMS

Rank Aug 2017	Rank Jul 2017	Rank Aug 2016	DBMS	Database Model	Score Aug 2017	Score Jul 2017	Score Aug 2016
1.	1.	1.	Neo4j	Graph DBMS	38.00	-0.52	+2.43
2.	2.	↑ 4.	Microsoft Azure Cosmos DB	Multi-model	9.42	+1.72	+6.87
3.	3.	↓ 2.	OrientDB	Multi-model	5.67	+0.10	-0.30
4.	4.	↓ 3.	Titan	Graph DBMS	5.21	+0.29	+0.33
5.	5.	↑ 6.	ArangoDB	Multi-model	2.92	-0.04	+0.99
6.	6.	↓ 5.	Virtuoso	Multi-model	1.98	-0.01	-0.42
7.	7.	7.	Giraph	Graph DBMS	1.05	-0.01	+0.10
8.	8.	↑ 9.	AllegroGraph	Multi-model	0.63	+0.02	+0.17
9.	9.	↓ 8.	Stardog	Multi-model	0.57	+0.02	+0.03
10.	10.	↑ 12.	GraphDB	Multi-model	0.57	+0.04	+0.40
11.	11.	↓ 10.	Sqrrl	Multi-model	0.50	+0.01	+0.24
12.	12.		Graph Engine	Multi-model	0.33	-0.03	
13.	13.	↓ 11.	InfiniteGraph	Graph DBMS	0.30	+0.00	+0.11
14.	↑ 15.	14.	Dgraph	Graph DBMS	0.27	-0.01	+0.12
15.	↓ 14.	↑ 17.	Blazegraph	Multi-model	0.26	-0.03	+0.18
16.	16.		JanusGraph	Graph DBMS	0.23	-0.00	
17.	17.	↑ 19.	Sparksee	Graph DBMS	0.19	+0.03	+0.13
18.	18.	↓ 15.	FlockDB	Graph DBMS	0.17	+0.01	+0.05
19.	19.	↓ 18.	HyperGraphDB	Graph DBMS	0.15	-0.00	+0.08

https://db-engines.com/en/ranking

The Graph Data Model

Definition

Also called **property graph model**.

1. Structure

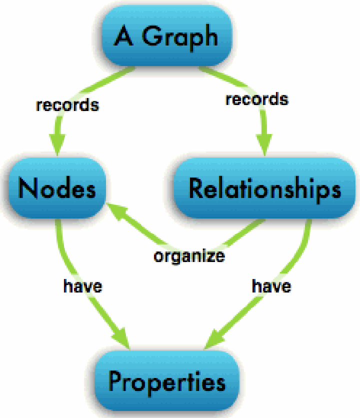
- Nodes: entities equivalent to records in the relational model
- Edges: (un)directed connections between nodes; represent relationships
- Properties: information relating to nodes (and edges); equivalent to attribute-value or key-value pairs

2. Constraints

- Nodes consist of a unique identifier, a set of outgoing edges, a set of incoming edges, and a collection of properties.
- Edges consist of a unique identifier, the end- and start-nodes, a label, and a collection of properties.

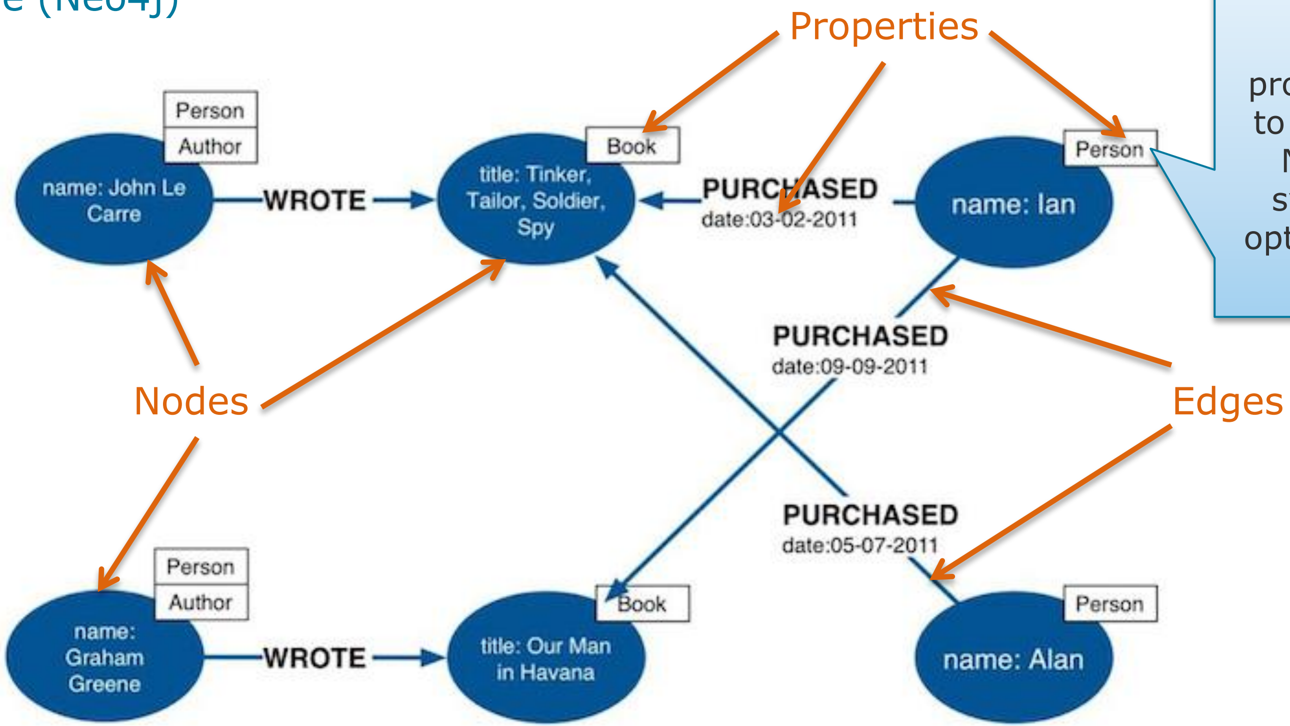
3. Operations

- Insert/query/update/delete notes, edges, and properties (CRUD)
- Traverse edges; aggregate queries (avg, min, max, count, sum, …)
- Most popular query language: Cypher (declarative; uses pattern matching)



Definition

Example (Neo4j)



= **Label**: dedicated property (label:"Person") to describe categories in Neo4j; allows special syntax in queries; still optional like any property

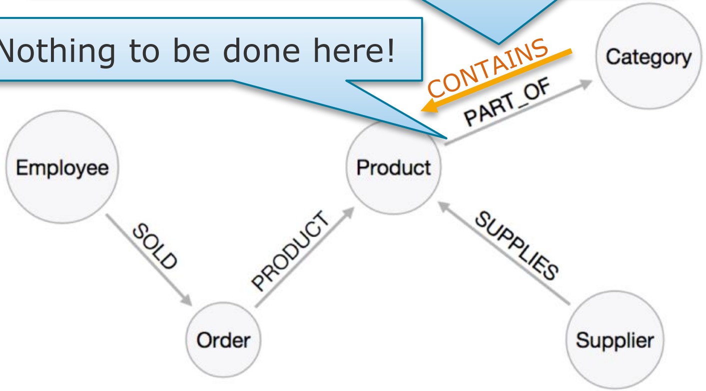
The Graph Data Model

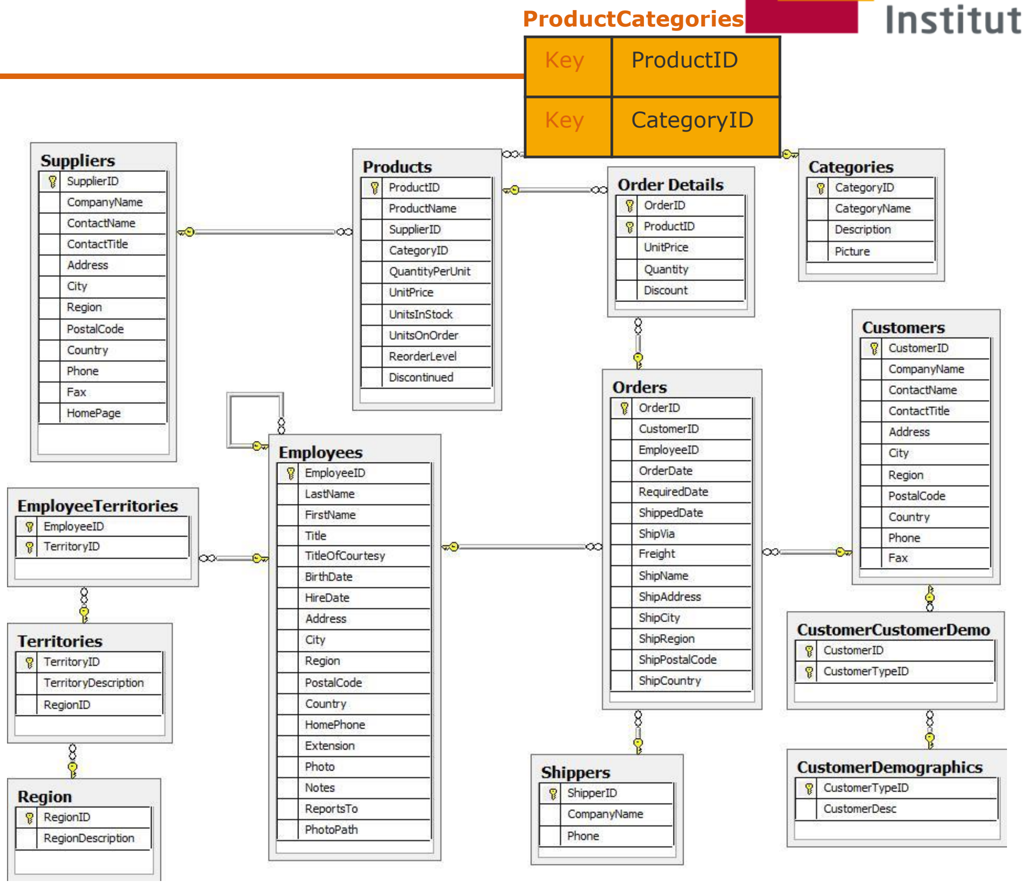
Graph vs. Relations

Model that **Products** can have multiple **Categories**!

We **could** (for semantic reasons) also add this inverse relation.

Nothing to be done here!

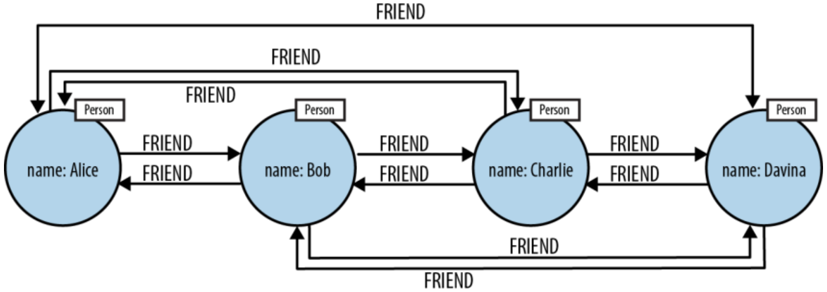




Storage Variations

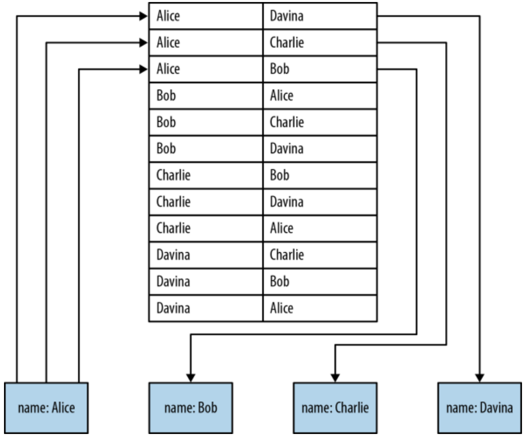
Native Graph Storage (e.g. Neo4j)

- Stores graph in a specialized graph format that points nodes directly to their adjacent nodes.
- Graph processing engines can traverse the graph by simply following links between nodes.



Non-Native Graph Storage (e.g. Titan)

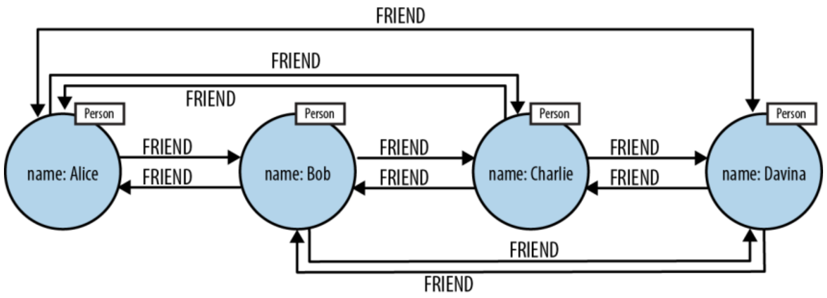
- Stores graph in relational or object-oriented format and uses indexes or join-tables to find adjacent nodes.
- Graph processing engine needs to look-up links in a global index or join records/entities.



Storage Variations

Native Graph Storage (e.g. Neo4j)

- Stores graph in a specialized graph format that points nodes directly to their adjacent nodes.
- Graph processing engines can traverse the graph by simply following links between nodes.



Non-Native Graph Storage (e.g. Titan)

- Example for relational model:

```
CREATE TABLE vertices (
  id            integer PRIMARY KEY,
  properties    json
);
CREATE TABLE edges (
  id            integer PRIMARY KEY,
  tail_vertex   integer REFERENCES vertices(id),
  head_vertex   integer REFERENCES vertices(id),
  label         text,
  properties    json
);
CREATE INDEX edges_tails ON edges (tail_vertex);
CREATE INDEX edges_heads ON edges (head_vertex);
```

Querying: Cypher

Cypher …

- is a declarative query language for graphs.
- formulates queries as **patterns** to match them against the graph.
- uses an ascii-art syntax:
  - Nodes: statements in parentheses, e.g. **(node)**
  - Relationships: statements in arrows, e.g. **-[connects]->**
  - Properties: statements in curly brackets, e.g. **{name:“Peter”}**
- is designed for Neo4j but intended as a standard (like SQL).
- is shortened CQL (Cypher Query Language), which is not to be confused with CQL (Cassandra Query Language)!

Querying: Cypher

General structure for patterns

Named variable to be referenced

We do not need to specify a variable for nodes/edges

":" is the short notation to filter by label, i.e., category

```
MATCH (node1:Label1)-[:Relationlabel]->(node2:Label2)
WHERE node1.propA = {value}
RETURN node2.propA, node2.propB
```

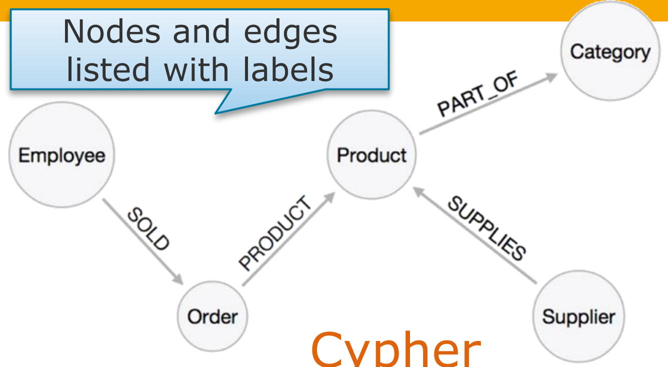
It's declarative!
→ The query planner can decide, for instance, to first find all *node1*s and then work the way to *node2*s or to do this vice versa.

Same as where clause above but as **property pattern** inside the node

Relation without label matches **any edge**

```
MATCH (node1:Label1 {node1.propA = {value}})-->(node2:Label2)
RETURN node2.propA, node2.propB
```

Querying: Cypher



Example: Basic graph queries

SQL	Cypher
`SELECT p.*` `FROM products as p;`	`MATCH (p:Product)` `RETURN p;`

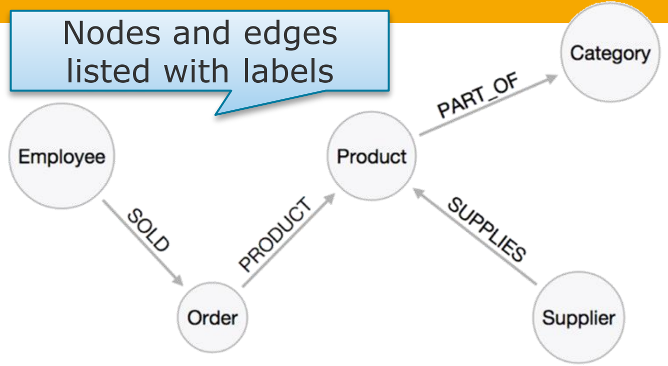
`SELECT p.ProductName, p.UnitPrice`
`FROM products as p`
`ORDER BY p.UnitPrice DESC`
`LIMIT 10;`
	`MATCH (p:Product)`
	`RETURN p.productName, p.unitPrice`
	`ORDER BY p.unitPrice DESC`
	`LIMIT 10;`

`SELECT p.ProductName, p.UnitPrice`
`FROM products AS p`
`WHERE p.ProductName = 'Chocolade';`
	`MATCH (p:Product)`
	`WHERE p.productName = "Chocolade"`
	`RETURN p.productName, p.unitPrice;`
	
	`MATCH (p:Product {productName:"Chocolade"})`
	`RETURN p.productName, p.unitPrice;`

The Graph Data Model

Querying: Cypher

Nodes and edges listed with labels



Example: Edge traversal queries

SQL

```
SELECT DISTINCT c.Name
FROM customers c, orders o, order_details od, products p
WHERE c.CustomerID = o.CustomerID
AND o.OrderID = od.OrderID
AND od.ProductID = p.ProductID
AND p.ProductName = 'Chocolade';
```

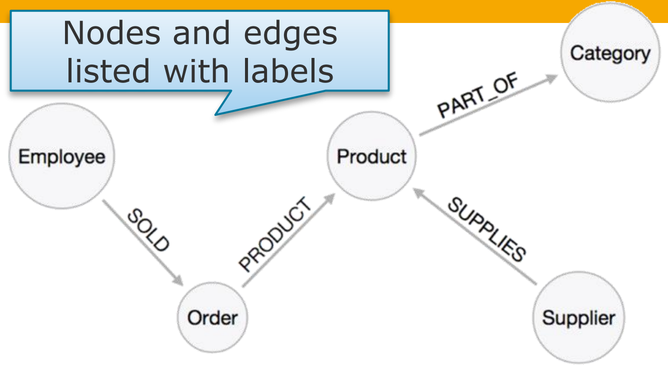
Cypher

```
MATCH (c:Customer)-[:PURCHASED]->(:Order)-[:PRODUCT]->(p:Product)
WHERE p.productName = "Chocolade"
RETURN distinct c.name;
```

Note that indexing is also possible on graphs:
CREATE INDEX ON :Product(productName);

Querying: Cypher

Nodes and edges listed with labels



Example: Aggregation queries

SQL

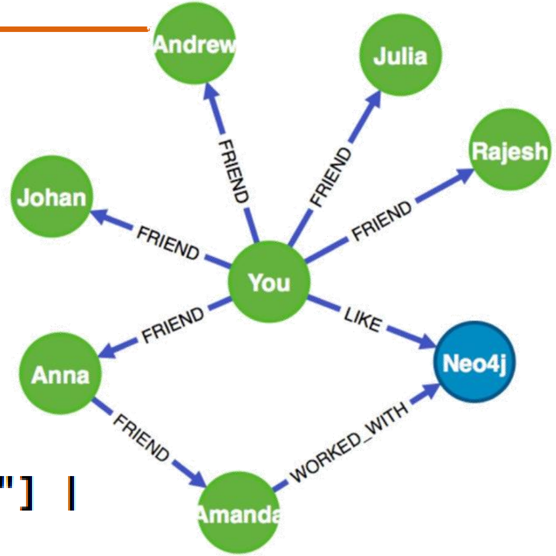
```
SELECT e.name, count(o.OrderID) AS Count
FROM Employee e JOIN Order o ON (o.EmployeeID = e.EmployeeID)
GROUP BY e.EmployeeID, e.name
ORDER BY Count DESC LIMIT 10;
```

Cypher

```
MATCH (:Order)<-[:SOLD]-(e:Employee)
RETURN e.name, count(o.id) AS Count
ORDER BY Count DESC LIMIT 10;
```

Grouping for aggregation is implicit: The first aggregation function causes all non-aggregated columns to automatically become grouping keys.
→ group by employee ID

Querying: Cypher



Example: Creating a graph

```
CREATE (you:Person {name:"You"})
RETURN you
```

```
MATCH  (you:Person {name:"You"})
CREATE (you)-[like:LIKE]->(neo:Database {name:"Neo4j" })
RETURN you,like,neo
```

```
MATCH (you:Person {name:"You"})
FOREACH (name in ["Johan","Rajesh","Anna","Julia","Andrew"] |
  CREATE (you)-[:FRIEND]->(:Person {name:name}))
```

```
MATCH (neo:Database {name:"Neo4j"})
MATCH (anna:Person {name:"Anna"})
CREATE (anna)-[:FRIEND]->(:Person:Expert {name:"Amanda"})-[:WORKED_WITH]->(neo)
```

Two labels!

https://neo4j.com/developer/cypher-query-language/

Querying: Cypher

"*" signals multiple levels; at least 1 and at most 3 "FRIEND" relations away (a clumsy SQL:1999 equivalent is **WITH RECURSION**)

Example: Where it gets interesting

```
MATCH (me:Person {name:"T. Papenbrock"})-[:FRIEND*1..3]->(friend:Person)
RETURN me, friend
```

My node

Direct, indirect, and in-indirect friends

Any relationship

Any node

Any relationship

```
MATCH (me:Person {name:"T. Papenbrock"})-->()-->(someone:Person)
RETURN someone.name
```

Multiple MATCH-statements in one query pattern if pattern cannot be expressed with one linear path expression.
→ in this way we can build star- or multidirectional-patterns

```
MATCH (me {name:"T. Papenbrock "})
MATCH (expert)-[:WORKED_WITH]->(db:Database {name:"Neo4j"})
MATCH path = shortestPath( (me)-[:FRIEND*..5]-(expert) )
RETURN db, expert, path
```

The shortest path of maximum length 5 from me to a person in my friends-network that can teach me Neo4j.

Ways to Model Properties/Relationships

Model "Nodes that have an address", which should be used for filtering.

a) Using a property and then filtering by property
(node {address: "address"})

b) Using a specific relationship type and then filtering by relationship type
(node)-[:HAS_ADDRESS]->(address)

c) Using a generic relationship type and then filtering by end node label
(node)-[:HAS]->(address:Address)

d) Using a generic relationship type and then filtering by relationship property
(node)-[:HAS {type: "address"}]->(address)

e) Using a generic relationship type and then filtering by end node property
(node)-[:HAS]->(address {type: "address"})

- ➢ Best way depends on query performance (for filtering probably b)), semantic fit (maybe c)), and extensibility (maybe a) or d))

https://neo4j.com/developer/cypher-query-language/ For further reading on Cypher

Triple-Stores

Definition

- Same graph definition as property graphs, but graph is stored in simple three-part, sentence-like statements of the form

  `(subject, predicate, object)`

  instead of nodes with collections of direct links.
- Subject: start node label
- Predicate: edge/property label
- Object: end node label or static value with primitive data type

then: triple = property

then: triple = edge

Examples

- **`(Jim, likes, Bananas)`**
- **`(Jim, age, 28)`**
- **`(Leon, is_a, Lion)`**
- **`(Leon, lives_in, Africa)`**
- **`(Africa, is_a, Continent)`**

Triple-Stores

Triple-Stores

- Examples:
  - Datomic
  - AllegroGraph
  - Virtuoso
- Query languages:
  - SPARQL
  - Datalog

Property Graph DBMSs

- Examples:
  - Neo4j
  - Titan
  - InfiniteGraph
- Query languages:
  - Cypher
  - Gremlin

Triple-Stores

Semantic Web

- Initiative of the World Wide Web Consortium (W3C) to extend the Web through standards for data formats and exchange protocols
- Most popular use case for triple stores
- Idea: Store entities/relations AND their semantic meaning in machine readable format!
- Approach: "Resource Description Framework" (RDF)
  - Subject, predicate and object in triples are represented as URIs
  - Example:

```
<http://www.hpi.de/#TPapenbrock>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.w3.org/2000/10/swap/pim/contact#Person> .
```

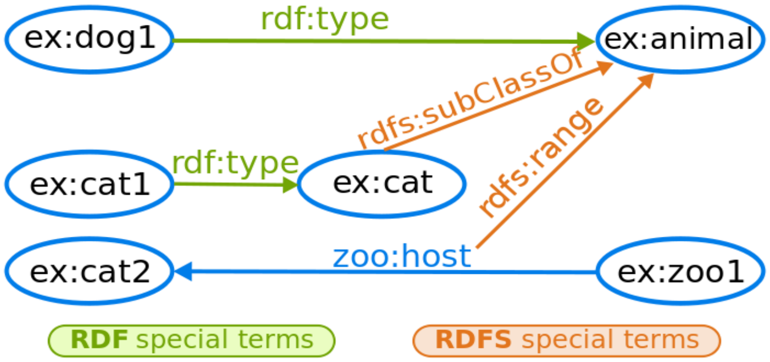
  - Ensures that datasets can be combined without semantic conflicts:

`<http://www.hpi.de/#HS1>` ≠ `<http://www.uni-potsdam.de/#HS1>`

Which tie their meaning to an ID

URIs don't need to resolve to web pages

Triple-Stores



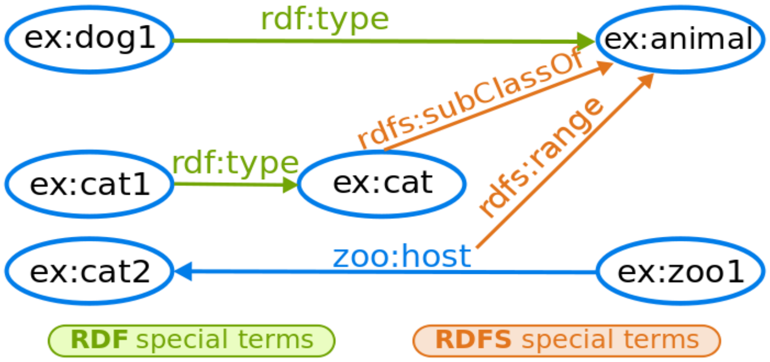
HPI Hasso Plattner Institut

Semantic Web

- Store semantic meaning with RDF:
  - "Resource Description Framework Schema" (RDFS)
    - A set of well defined RDF classes and properties to describe ontologies (=formal description of "real" entities in some domain)
    - Example for RDFS classes:
      rdfs:Class (declares a node as a class for other nodes)
      foaf:Person rdf:type rdfs:Class .
      → **ex:Lisa rdf:type foaf:Person .**
    - Example for RDFS properties:
      rdfs:domain (declares the subject type for a predicate)
      rdfs:range (declares the object type for a predicate)
      ex:student rdfs:domain foaf:Person .
      ex:student rdfs:range foaf:University .
      → **ex:Lisa ex:student ex:UniversityPotsdam .**

> If RDFS is insufficient to build your ontology, use its extension **OWL ("Web Ontology Language")**

Triple-Stores



HPI Hasso Plattner Institut

Semantic Web

- Store semantic meaning with RDF:

Turtle notation: a textual syntax for RDF that allows a graph to be written in compact and natural form (https://www.w3.org/TR/turtle/)
General syntax: <url>:subject <url>:predicate <url>:object .

rdfs:Class (declares a node as a class for other nodes)

```
foaf:Person rdf:type rdfs:Class .
→ ex:Lisa rdf:type foaf:Person .
```

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
ex:student rdfs:range foaf:University .
→ ex:Lisa ex:student ex:UniversityPotsdam .
```

For more details:

a) Web page: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

b) Lecture "Semantic Web" (Dr. Sack)

Querying: SPARQL

SPARQL …

- is a declarative query language for triple-store graphs in RDF format.
- formulates queries in RDF syntax.
- is an acronym for “SPARQL Protocol and RDF Query Language”.
- Example:

SPARQL

```
SELECT ?locationName
WHERE {
    ?hpi :name “HPI gGmbH” .
    ?hpi :location ?locationName .
}
```

Cypher

```
MATCH (hpi {name: “HPI gGmbH”})-[:location]->(loc)
RETURN loc.name
```

Querying: SPARQL

SPARQL …

- is a declarative query language for triple-store graphs in RDF format
- formulates queries in RDF syntax
- is an acronym for "SPARQL Protocol and RDF Query Language"
- Example:

SPARQL and Cipher are quite similar.

SPARQL

```
SELECT ?personName
WHERE {
    ?person :name ?personName .
    ?person :bornIn / :within* / :name "Europe" .
}
```

<url>:label

<variable>:label

Cypher

```
MATCH (person)-[:bornIn]->()-[:within*0..]->(location {name: "Europe"}})
RETURN person.name
```

Strengths and Weaknesses

Strengths

- Many-to-many relationships (other data models heavily prefer one-to-many)
- Efficient traversal of relationships between entities (relationship queries)
  - Traversal costs proportional to the average out-degree of nodes (and not proportional to the overall number of relationships)
  - Join performance scales naturally with the size of the data
- Natural support for graph queries: shortest path, community detection, …
- Flexible schemata due to flexible edge and property definitions
- Direct mapping of nodes/edges to data structures of object-oriented applications

Weaknesses

- OLTP and CRUD operations on many nodes are comparatively slow
- Data Distribution is hard, because workload is based on data locality
- Querying difficult due to unknown schema (flexibility leads to misuse)

Graph DBMSs and Distribution

Replication/Clustering

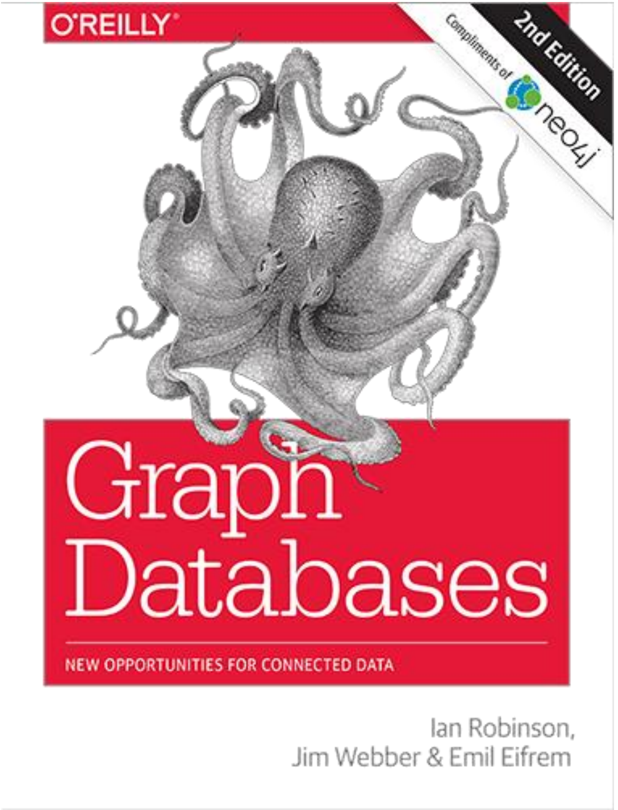
- Supported by most graph DBMSs
- Same techniques for consistency management as other DBMSs
- Queries can be routed to any replica and then be served from it

Partitioning/Sharding

- Performance-wise problematic, because graph queries have join character rather than point query character and often cross partition boundaries.
  - ➢ Most systems offer rudimentary partitioning support, but try to avoid it and go for replication (e.g. Neo4j).
- Challenge: Find a graph partitioning with …
  a) possibly few inter-partition links;
  b) possibly balanced partition sizes;
  c) a certain number of partitions that matches physical nodes.

} Subject to research!

Further Reading on Graph Databases



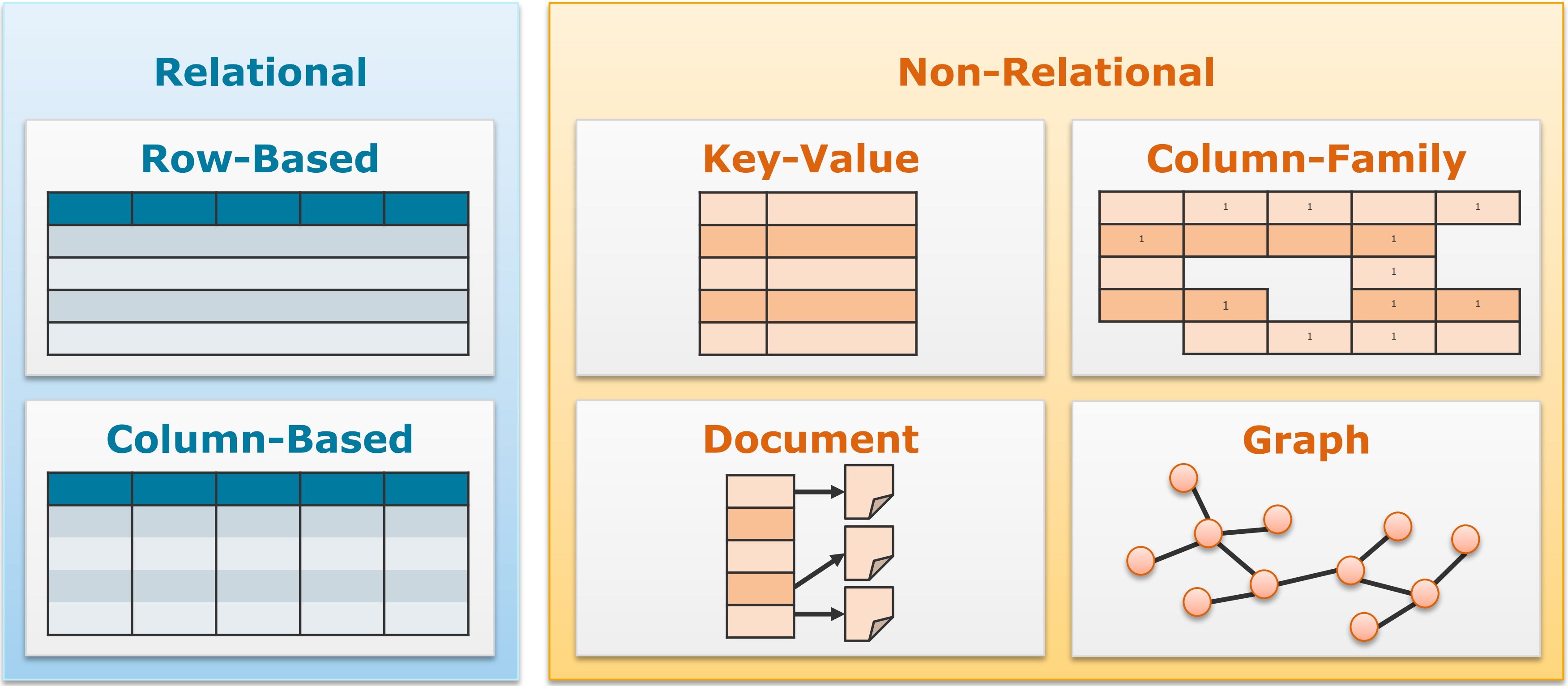
Graph Databases

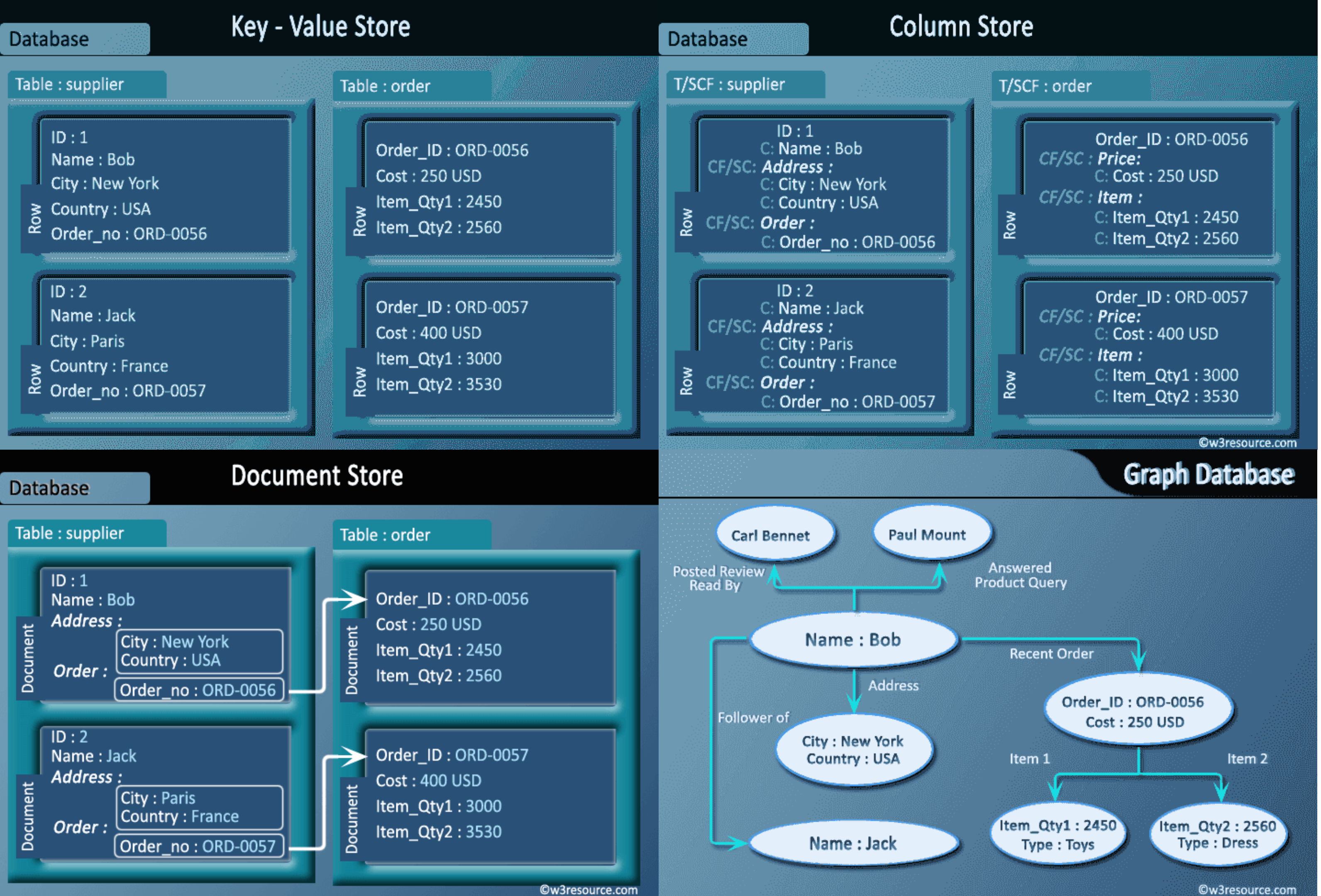
- Free to download as pdf at:
  - http://graphdatabases.com/

Overview

Relational and Non-Relational Data Models

HPI Hasso Plattner Institut





Hasso Plattner Institut

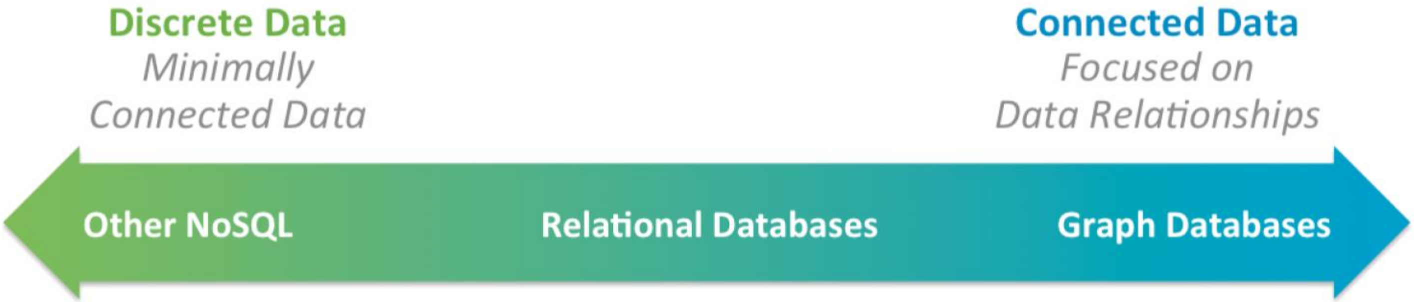
Distributed Data Management

Data Models and Query Languages

ThorstenPapenbrock

Slide **84**

Summary



Other NoSQL

- inhomogeneous data
- frequent schema changes
- fast growth
- little/no relationship support
- usually sacrifice ACID
- usually horizontal scaling
- data distribution
- throughput
- OLTP focus

Relational Databases

- homogeneous data
- relatively reliable schemata
- moderate growth
- full relationship support
- usually comply with ACID
- usually vertical scaling
- data compression
- transactions and security
- OLTP and OLAP

Graph Databases

- highly linked data
- frequent schema changes
- moderate growth
- specialized on relationships
- usually comply with ACID
- usually vertical scaling
- data optimization
- relationship traversal
- OLAP focus

Image: © https://neo4j.com/blog/aggregate-stores-tour/

Check yourself

- Train your query skills with the following exercises:
  - MongoDB
    - https://www.w3resource.com/mongodb-exercises/
    - (includes solutions)
  - Neo4j / Cypher
    - https://www.uio.no/studier/emner/matnat/ifi/INF3100/v17/undervisningsmateriale/graph-dbs---neo4j.pdf
- It helps if you really set up a database and try the queries yourself. If you face any problems in doing so, please do not hesitate to ask us or the mailing list for help.

Chapter 2. Data Models and Query Languages