Distributed Data Management
Data Models and Query Languages

Thorsten Papenbrock
F-2.04, Campus II
Hasso Plattner Institut
Introduction
Layering Data Models

1. Conceptual layer
   - Data structures, objects, modules, ...
     - Application code

2. Logical layer
   - Relational tables, JSON, XML, graphs, ...
     - Database management system (DBMS) or storage engine
   - our focus now

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
Overview
Relational and Non-Relational Data Models

Relational

Row-Based

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Column-Based

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Non-Relational

Key-Value

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Column-Family

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Document

Graph

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
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

Overview
Relational and Non-Relational Data Models

Relational

Row-Based

Column-Based

Non-Relational

Key-Value

Column-Family

Document

Graph
The Relational Data Model

Natural Relational Data

Transaction Data

Statistical Data

Master Data

Business Data
## The Relational Data Model

### Popular relational DBMS

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

The Relational Data Model

**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)

---

The Relational Data Model

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

Grundbausteine:
- SELECT <Attributliste>
  -属性列表
- FROM <Relationenliste>
  -关系列表
- WHERE <Bedingungen>
  -条件
- GROUP BY ...
  -按...
  -按...
  -按...
- HAVING <Funktionen auf Gruppierungseigenschaften>
  -函数...
  -函数...
  -函数...
- ORDER BY <Attribut>
  -按...
  -按...
  -按...

Komplexe Anfragen:
- SELECT <Attributliste>
  -属性列表
- FROM <Relationenliste>
  -关系列表
- WHERE <Bedingungen>
  -条件
- GROUP BY <Gruppierungseigenschaften>
  -按...
  -按...
  -按...
- HAVING <Funktionen auf Gruppierungseigenschaften>
  -函数...
  -函数...
  -函数...
- ORDER BY <Attribut>
  -按...
  -按...
  -按...

Datenelemente: Data Definition Language (DDL)
- CREATE TABLE <Tabellenname>(<Attributliste mit Datentypen>):
  - 创建表...
  - 创建表...
  - 创建表...
- DROP TABLE <Tabellenname>
  - 删除表...
  - 删除表...
  - 删除表...

Datenelemente: Data Modelling Language (DML)
- INSERT INTO <Tabellenname> [(<Attributliste>)] VALUES (<Attributliste>):
  - 插入...
  - 插入...
  - 插入...
- DELETE FROM <Tabellenname> WHERE <Bedingung>
  - 删除...
  - 删除...
  - 删除...
- UPDATE <Tabellenname> SET <Spalte> = Wert WHERE <Bedingung>
  - 更新...
  - 更新...
  - 更新...

Mengenoperationen
- INTERSECT <Anfrage1> <Anfrage2>
- UNION <Anfrage1> <Anfrage2>
- EXCEPT <Anfrage1> <Anfrage2>

Join-Varianten
1. Kreuzprodukt mit Bedingungen
   - SELECT *
     FROM <Tabelle1> JOIN <Tabelle2>
     WHERE <Bedingung>
2. Schlüsselwort
   - CROSS JOIN <Tabellenname> ON <Spalte1> = <Spalte2>

Sichten
- CREATE VIEW <Sichtname> AS <Anfrage>
  - 创建视图...
  - 创建视图...
  - 创建视图...

HPI Hasso Plattner Institut
The Relational Data Model

Querying: SQL – Examples

Schemata:

- **Product** \( \text{(maker, model, type)} \)
- **PC** \( \text{(model, speed, ram, hd, rd)} \)
- **Laptop** \( \text{(model, speed, ram, hd, screen)} \)

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

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

```sql
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.”

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

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

“Find all makers that produce Laptops but no PCs.”

Distributed Data Management

Data Models and Query Languages

Thorsten Papenbrock

Slide 12
The Relational Data Model

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
The Relational Data Model
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

Row Oriented Database

<table>
<thead>
<tr>
<th>date</th>
<th>price</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-01-20</td>
<td>10.1</td>
<td>10</td>
</tr>
<tr>
<td>2011-01-21</td>
<td>10.3</td>
<td>20</td>
</tr>
<tr>
<td>2011-01-22</td>
<td>10.5</td>
<td>40</td>
</tr>
<tr>
<td>2011-01-23</td>
<td>10.4</td>
<td>5</td>
</tr>
<tr>
<td>2011-01-24</td>
<td>11.2</td>
<td>55</td>
</tr>
<tr>
<td>2011-01-25</td>
<td>11.4</td>
<td>66</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2013-03-31</td>
<td>17.3</td>
<td>100</td>
</tr>
</tbody>
</table>

Table of Data

<table>
<thead>
<tr>
<th>date</th>
<th>price</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-01-20</td>
<td>10.1</td>
<td>10</td>
</tr>
<tr>
<td>2011-01-21</td>
<td>10.3</td>
<td>20</td>
</tr>
<tr>
<td>2011-01-22</td>
<td>10.5</td>
<td>40</td>
</tr>
<tr>
<td>2011-01-23</td>
<td>10.4</td>
<td>5</td>
</tr>
<tr>
<td>2011-01-24</td>
<td>11.2</td>
<td>55</td>
</tr>
<tr>
<td>2011-01-25</td>
<td>11.4</td>
<td>66</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2013-03-31</td>
<td>17.3</td>
<td>100</td>
</tr>
</tbody>
</table>

Column Oriented Database

<table>
<thead>
<tr>
<th>date</th>
<th>price</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-01-20</td>
<td>10.1</td>
<td>10</td>
</tr>
<tr>
<td>2011-01-21</td>
<td>10.3</td>
<td>20</td>
</tr>
<tr>
<td>2011-01-22</td>
<td>10.5</td>
<td>40</td>
</tr>
<tr>
<td>2011-01-23</td>
<td>10.4</td>
<td>5</td>
</tr>
<tr>
<td>2011-01-24</td>
<td>11.2</td>
<td>55</td>
</tr>
<tr>
<td>2011-01-25</td>
<td>11.4</td>
<td>66</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2013-03-31</td>
<td>17.3</td>
<td>100</td>
</tr>
</tbody>
</table>

## The Relational Data Model
### Storage Variations

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

Better **OLTP** performance

Better **OLAP** performance

**Distributed Data Management**

Data Models and Query Languages

Thorsten Papenbrock

Slide 15
The Relational Data Model

Storage Variations

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

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Equatorial diameter</th>
<th>Mass</th>
<th>Orbital radius</th>
<th>Orbital period</th>
<th>Rotation period</th>
<th>Confirmed moons</th>
<th>Rings</th>
<th>Atmosphere</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mercury</td>
<td>Terrestrial</td>
<td>0.382</td>
<td>0.06</td>
<td>0.47</td>
<td>0.24</td>
<td>58.64</td>
<td>0</td>
<td>no</td>
<td>minimal</td>
</tr>
<tr>
<td>Venus</td>
<td>Terrestrial</td>
<td>0.949</td>
<td>0.82</td>
<td>0.72</td>
<td>0.62</td>
<td>-243.02</td>
<td>0</td>
<td>no</td>
<td>CO₂, N₂</td>
</tr>
<tr>
<td>Earth</td>
<td>Terrestrial</td>
<td>1.000</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1</td>
<td>no</td>
<td>N₂, O₂, Ar</td>
</tr>
<tr>
<td>Mars</td>
<td>Terrestrial</td>
<td>0.532</td>
<td>0.11</td>
<td>1.52</td>
<td>1.88</td>
<td>1.03</td>
<td>2</td>
<td>no</td>
<td>CO₂, N₂, Ar</td>
</tr>
<tr>
<td>Jupiter</td>
<td>Giant</td>
<td>11.209</td>
<td>317.8</td>
<td>5.20</td>
<td>11.86</td>
<td>0.41</td>
<td>67</td>
<td>yes</td>
<td>H₂, He</td>
</tr>
<tr>
<td>Saturn</td>
<td>Giant</td>
<td>9.449</td>
<td>95.2</td>
<td>9.54</td>
<td>29.46</td>
<td>0.43</td>
<td>62</td>
<td>yes</td>
<td>H₂, He</td>
</tr>
<tr>
<td>Uranus</td>
<td>Giant</td>
<td>4.007</td>
<td>14.6</td>
<td>19.22</td>
<td>84.01</td>
<td>-0.72</td>
<td>27</td>
<td>yes</td>
<td>H₂, He</td>
</tr>
<tr>
<td>Neptune</td>
<td>Giant</td>
<td>3.883</td>
<td>17.2</td>
<td>30.06</td>
<td>164.8</td>
<td>0.67</td>
<td>14</td>
<td>yes</td>
<td>H₂, He</td>
</tr>
</tbody>
</table>

The data is represented as a CSV file:

```
"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"
```
Access Example

```java
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();
} finally {
    writer = null;
}
```

```java
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();
} finally {
    reader = null;
}
```

Java 1.7 using `au.com.bytecode.opencsv`
The Relational Data Model

CSV Files

Access Example

Java 1.8 using au.com.bytecode.opencsv

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

Relational
- Row-Based
- Column-Based

Non-Relational
- Key-Value
- Column-Family
- Document
- Graph
## The Key-Value Data Model

### Popular Key-Value Stores

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

**In Akka!**

**In Kafka!**


---

**Distributed Data Management**

Data Models and Query Languages

Thorsten Papenbrock

Slide 22
The Key-Value Data Model

Definition

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.

Some implementations do support this and some don’t.
The Key-Value Data Model

Example

keys

John Smith

Lisa Smith

Sandra Dee

hash function

buckets

00

01 521-8976

02 521-1234

03

13

14 521-9655

15

©Jorge Stolfi (https://commons.wikimedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg)

Distributed Data Management
Data Models and Query Languages
ThorstenPapenbrock
Slide 24
The Key-Value Data Model

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
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

- **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

"<group>:<entity>" is a naming convention.
The Key-Value Data Model
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!)
Overview
Relational and Non-Relational Data Models

Relational

Row-Based

Column-Based

Non-Relational

Key-Value

Column-Family

Document

Graph
# The Column-Family Data Model

## Popular Column-Family Stores

<table>
<thead>
<tr>
<th>Rank</th>
<th>Aug 2017</th>
<th>Jul 2017</th>
<th>Aug 2016</th>
<th>DBMS</th>
<th>Database Model</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Cassandra</td>
<td>Wide column store</td>
<td>126.72</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td>HBase</td>
<td>Wide column store</td>
<td>63.52</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
<td>Microsoft Azure Cosmos DB</td>
<td>Multi-model</td>
<td>9.42</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td></td>
<td>Accumulo</td>
<td>Wide column store</td>
<td>3.66</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td>Microsoft Azure Table Storage</td>
<td>Wide column store</td>
<td>2.96</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td>Google Cloud Bigtable</td>
<td>Wide column store</td>
<td>0.70</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td>MapR-DB</td>
<td>Multi-model</td>
<td>0.51</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>7</td>
<td></td>
<td>Sqrrl</td>
<td>Multi-model</td>
<td>0.50</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>8</td>
<td></td>
<td>ScyllaDB</td>
<td>Wide column store</td>
<td>0.39</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>Alibaba Cloud Table Store</td>
<td>Wide column store</td>
<td>0.01</td>
</tr>
</tbody>
</table>

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

---

**Distributed Data Management**

Data Models and Query Languages

Thorsten Papenbrock
Slide 29
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.

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)

For this reason, they are also called “Wide Column Stores”.
The Column-Family Data Model

Example

- **Column** = key-value pair
  - super column name = key-hashmap pair
    - name
    - value
  - name
  - value

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

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

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

Hierarchy of keys enables:

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

Analogy:

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Cassandra Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Keyspace</td>
</tr>
<tr>
<td>Table</td>
<td>Column Family (CF)</td>
</tr>
<tr>
<td>Primary key</td>
<td>Row key</td>
</tr>
<tr>
<td>Column name</td>
<td>Column name/key</td>
</tr>
<tr>
<td>Column value</td>
<td>Column value</td>
</tr>
</tbody>
</table>

Distributed Data Management

Data Models and Query Languages

Thorsten Papenbrock

Slide 33
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
The Column-Family Data Model
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;
```

Result:
```
id         | song_order | album               | artist         | song_id          | title
-----------------|------------|---------------------|----------------|------------------|------------------------
62c36092... | 4          | No One Rides for Free | Fu Manchu      | 7db10490...     | 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
```
The Column-Family Data Model
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};

CREATE DATABASE myDatabase;

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.
```
The Column-Family Data Model
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

**Strengths**

- **efficient storage:** fast inserts of data items
- **efficient retrieval:** fast point queries, i.e., value lookups
- **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

---

“**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
Overview
Relational and Non-Relational Data Models

Relational

Row-Based

Column-Based

Non-Relational

Key-Value

Column-Family

Document

Graph
The Document Data Model

Natural Document Data

Digital Documents

Web Pages

Structured Data

Structured Data

Log Data

Scientific Data Formats
# The Document Data Model

## Popular Document Stores

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1.</td>
<td>1.</td>
<td>1.</td>
<td>MongoDB</td>
<td>Document store</td>
<td>330.50</td>
<td>-2.27</td>
<td>+12.01</td>
</tr>
<tr>
<td>2.</td>
<td>2.</td>
<td>3.</td>
<td>3.</td>
<td>Amazon DynamoDB</td>
<td>Document store</td>
<td>37.62</td>
<td>+1.16</td>
<td>+11.02</td>
</tr>
<tr>
<td>3.</td>
<td>3.</td>
<td>2.</td>
<td>2.</td>
<td>Couchbase</td>
<td>Document store</td>
<td>32.97</td>
<td>-0.05</td>
<td>+5.57</td>
</tr>
<tr>
<td>4.</td>
<td>4.</td>
<td>4.</td>
<td>4.</td>
<td>CouchDB</td>
<td>Document store</td>
<td>21.34</td>
<td>-0.81</td>
<td>+0.28</td>
</tr>
<tr>
<td>5.</td>
<td>5.</td>
<td>5.</td>
<td>5.</td>
<td>MarkLogic</td>
<td>Multi-model</td>
<td>12.50</td>
<td>+0.07</td>
<td>+2.46</td>
</tr>
<tr>
<td>6.</td>
<td>6.</td>
<td>10.</td>
<td>6.</td>
<td>Microsoft Azure Cosmos DB</td>
<td>Multi-model</td>
<td>9.42</td>
<td>+1.72</td>
<td>+6.87</td>
</tr>
<tr>
<td>7.</td>
<td>7.</td>
<td>6.</td>
<td>6.</td>
<td>OrientDB</td>
<td>Multi-model</td>
<td>5.67</td>
<td>+0.10</td>
<td>-0.30</td>
</tr>
<tr>
<td>8.</td>
<td>8.</td>
<td>7.</td>
<td>8.</td>
<td>RethinkDB</td>
<td>Document store</td>
<td>4.88</td>
<td>-0.07</td>
<td>+0.12</td>
</tr>
<tr>
<td>9.</td>
<td>10.</td>
<td>9.</td>
<td>8.</td>
<td>Firebase Realtime Database</td>
<td>Document store</td>
<td>4.51</td>
<td>+0.44</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>9.</td>
<td>8.</td>
<td>8.</td>
<td>Cloudant</td>
<td>Document store</td>
<td>4.31</td>
<td>-0.33</td>
<td>-0.20</td>
</tr>
<tr>
<td>11.</td>
<td>11.</td>
<td>17.</td>
<td>11.</td>
<td>Google Cloud Datastore</td>
<td>Document store</td>
<td>3.49</td>
<td>-0.40</td>
<td>+2.41</td>
</tr>
<tr>
<td>12.</td>
<td>12.</td>
<td>9.</td>
<td>9.</td>
<td>RavenDB</td>
<td>Document store</td>
<td>3.27</td>
<td>-0.24</td>
<td>-1.17</td>
</tr>
<tr>
<td>13.</td>
<td>12.</td>
<td>12.</td>
<td>12.</td>
<td>Apache Drill</td>
<td>Multi-model</td>
<td>3.21</td>
<td>-0.01</td>
<td>+0.83</td>
</tr>
<tr>
<td>14.</td>
<td>13.</td>
<td>13.</td>
<td>13.</td>
<td>PouchDB</td>
<td>Document store</td>
<td>2.99</td>
<td>-0.07</td>
<td>+0.64</td>
</tr>
<tr>
<td>16.</td>
<td>15.</td>
<td>15.</td>
<td>15.</td>
<td>CloudKit</td>
<td>Document store</td>
<td>2.20</td>
<td>-0.21</td>
<td>+0.37</td>
</tr>
<tr>
<td>17.</td>
<td>17.</td>
<td>11.</td>
<td>17.</td>
<td>Virtuoso</td>
<td>Multi-model</td>
<td>1.98</td>
<td>-0.01</td>
<td>-0.42</td>
</tr>
<tr>
<td>18.</td>
<td>18.</td>
<td>16.</td>
<td>18.</td>
<td>Datameer</td>
<td>Document store</td>
<td>1.73</td>
<td>+0.11</td>
<td>-0.01</td>
</tr>
<tr>
<td>19.</td>
<td>19.</td>
<td>18.</td>
<td>19.</td>
<td>Mnnesia</td>
<td>Document store</td>
<td>1.17</td>
<td>-0.21</td>
<td>+0.12</td>
</tr>
</tbody>
</table>

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

---

**Distributed Data Management**

Data Models and Query Languages

Thorsten Papenbrock

Slide 41
The Document Data Model

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**.
The Document Data Model

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.

Distributed Data Management
Data Models and Query Languages

ThorstenPapenbrock
Slide 43
The Document Data Model

Example 1

Data Models and Query Languages

Distributed Data Management

Data Models and Query Languages

Thorsten Papenbrock

Slide 44
The Document Data Model

Example 2

Benno87

AnnaMT

Distributed Data Management

Data Models and Query Languages

Thorsten Papenbrock

Slide 45

JSON Format

{  
  "_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"
  }
}
The Document Data Model

Example 3

Benno87

AnnaMT

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

XML Format

```
<_id>1</_id>
<username>ben</username>
<password>ughiuwuv</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>
```
The Document Data Model

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
MongoDB ...

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

<table>
<thead>
<tr>
<th>SQL Terms/Concepts</th>
<th>MongoDB Terms/Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>database</td>
</tr>
<tr>
<td>table</td>
<td>collection</td>
</tr>
<tr>
<td>row/record</td>
<td>document</td>
</tr>
<tr>
<td>column/attribute</td>
<td>field</td>
</tr>
<tr>
<td>index</td>
<td>index</td>
</tr>
<tr>
<td>table join</td>
<td>$lookup, embedded document</td>
</tr>
<tr>
<td>primary key (any column)</td>
<td>primary key (always the _id filed)</td>
</tr>
<tr>
<td>aggregation (group by)</td>
<td>aggregation pipeline</td>
</tr>
</tbody>
</table>
The Document Data Model
Querying: MongoDB API – Examples

Create/Drop

Document:

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

SQL

```sql
CREATE TABLE people (  
id MEDIUMINT NOT NULL AUTO_INCREMENT,  
user_id VARCHAR(30),  
age NUMBER,  
status CHAR(1),  
PRIMARY KEY (id)
)

DROP TABLE people
```

MongoDB

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

db.people.drop()
```

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

https://docs.mongodb.com/manual/
The Document Data Model

Querying: MongoDB API – Examples

### Alter

**Document:**

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

**SQL:**

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

**MongoDB:**

```javascript
db.people.updateMany(

  { },

  { $set: { join_date: new Date() } }
)
```

**SQL:**

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

**MongoDB:**

```javascript
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/
The Document Data Model

Querying: MongoDB API – Examples

**Insert, Update and Delete**

**Document:**

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

**SQL**

```
INSERT INTO people(user_id, age, status) VALUES ("bcd001", 45, "A")
```

**UPDATE**

```
UPDATE people SET status = "C" WHERE age > 25
```

**DELETE**

```
DELETE FROM people WHERE status = "D"
```

**MongoDB**

```
db.people.insertOne(
    { user_id: "bcd001", age: 45, status: "A" }
)
```

```
db.people.updateMany(
    { age: { $gt: 25 } },
    { $set: { status: "C" } }
)
```

```
db.people.deleteMany( { status: "D" } )
```

https://docs.mongodb.com/manual/
# The Document Data Model

## Querying: MongoDB API – Examples

### Select

Document:

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

<table>
<thead>
<tr>
<th>SQL</th>
<th>MongoDB</th>
</tr>
</thead>
</table>
| ```
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 } }
)
``` |

[https://docs.mongodb.com/manual/](https://docs.mongodb.com/manual/)
The Document Data Model
Querying: MongoDB API – Examples

### Aggregate

**Document:**

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

**SQL**

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

**MongoDB**

```javascript
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/
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": "ijkl", description: "product 3", "instock": 60 }
  { "_id": 4, "sku": "jkl", description: "product 4", "instock": 70 }
  { "_id": 5, "sku": null, description: "Incomplete" }
  { "_id": 6 }
  ```
For Indexes, the DBMS maintains the document offsets in collections so that indexes work similar to indexes in relational databases.
The Document Data Model

Querying: MongoDB API – Examples

**Very rich API**

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

Visit the manual!

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

Relational and Non-Relational Data Models

Relational

Row-Based

Column-Based

Non-Relational

Key-Value

Column-Family

Document

Graph
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

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

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

### Distributed Data Management

Data Models and Query Languages

Thorsten Papenbrock

Slide 59
The Graph Data Model

Definition

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)

Also called **property graph model**.
Example (Neo4j)

**Nodes**
- name: John Le Carre
- name: Graham Greene
- title: Our Man in Havana
- title: Tinker, Tailor, Soldier, Spy
- name: lan
- name: Alan

**Edges**
- WROTE
- PURCHASED

**Properties**
- 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!
The Graph Data Model

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.

Distributed Data Management
Data Models and Query Languages

ThorstenPapenbrock
Slide 63
The Graph Data Model

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:

```sql
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);
```
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)!
The Graph Data Model
Querying: Cypher

General structure for patterns

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

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

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

It’s declarative!
→ The query planner can decide, for instance, to first find all node1s and then work the way to node2s or to do this vice versa.

Same as where clause above but as property pattern inside the node
Relation without label matches any edge

Distributed Data Management
Data Models and Query Languages
Thorsten Papenbrock
Slide 66
The Graph Data Model

Querying: Cypher

Example: Basic graph queries

**SQL**

```
SELECT p.*
FROM products as p;
```

```
SELECT p.ProductName, p.UnitPrice
FROM products as p
ORDER BY p.UnitPrice DESC
LIMIT 10;
```

```
SELECT p.ProductName, p.UnitPrice
FROM products AS p
WHERE p.ProductName = 'Chocolade';
```

**Cypher**

```
MATCH (p:Product)
RETURN p;
```

```
MATCH (p:Product)
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice DESC
LIMIT 10;
```

```
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

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]-(o: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);
```
The Graph Data Model

Querying: Cypher

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 (o: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
The Graph Data Model

Querying: Cypher

Example: Creating a graph

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

MATCH (you:Person {name:"You"})
CREATE (you)-[: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)

https://neo4j.com/developer/cypher-query-language/
The Graph Data Model
Querying: Cypher

Example: Where it gets interesting

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

“My node” signals multiple levels; at least 1 and at most 3 “FRIEND” relations away.
(a clumsy SQL:1999 equivalent is WITH RECURSION)

Direct, indirect, and in-indirect friends

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.
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
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

Examples

- \((Jim, likes, Bananas)\)
- \((Jim, age, 28)\)
- \((Leon, is_a, Lion)\)
- \((Leon, lives_in, Africa)\)
- \((Africa, is_a, Continent)\)
Triple-Stores

- Examples:
  - Datomic
  - AllegroGraph
  - Virtuoso

- Query languages:
  - SPARQL
  - Datalog

Property Graph DBMSs

- Examples:
  - Neo4j
  - Titan
  - InfiniteGraph

- Query languages:
  - Cypher
  - Gremlin
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:
    
    \[
    \text{<http://www.hpi.de/\#TPapenbrock>}
    \text{<http://www.w3.org/1999/02/22-rdf-syntax-ns\#type>}
    \text{<http://www.w3.org/2000/10/swap/pim/contact\#Person> .}
    \]
  - Ensures that datasets can be combined without semantic conflicts:
    
    \[
    \text{<http://www.hpi.de/\#HS1> \neq <http://www.uni-potsdam.de/\#HS1>}
    \]
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`.
    - 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:range foaf:University`.

If RDFS is insufficient to build your ontology, use its extension OWL ("Web Ontology Language")
The Graph Data Model

Triple-Stores

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/)

  - `rdfs:Class` (declares a node as a class for other nodes)
  - `foaf:Person` `rdf:type` `rdfs:Class` .

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

  - `ex:student` `rdfs:range` `foaf:University` .

For more details:
  a) Web page: https://www.w3.org/2014/REC-rdf11-concepts-20140225/
  b) Lecture "Semantic Web" (Dr. Sack)
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
```
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 ?personName 
WHERE {
  ?person :name ?personName .
  ?person :bornIn / :within* / :name "Europe".
}
```

SPARQL and Cipher are quite similar.

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

Distributed Data Management
Data Models and Query Languages
The Graph Data Model
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)
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!
The Graph Data Model
Further Reading on Graph Databases

Graph Databases

- Free to download as pdf at:
  - [http://graphdatabases.com/](http://graphdatabases.com/)
Overview
Relational and Non-Relational Data Models

Relational
- Row-Based
- Column-Based

Non-Relational
- Key-Value
- Column-Family
- Document
- Graph
Data Models and Query Languages

Distributed Data Management

Thorsten Papenbrock
### Data Models and Query Languages

#### Summary

<table>
<thead>
<tr>
<th>Discrete Data</th>
<th>Connected Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Minimally Connected Data</strong></td>
<td><strong>Focused on Data Relationships</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other NoSQL</th>
<th>Relational Databases</th>
<th>Graph Databases</th>
</tr>
</thead>
<tbody>
<tr>
<td>- inhomogeneous data</td>
<td>- homogeneous data</td>
<td>- highly linked data</td>
</tr>
<tr>
<td>- frequent schema changes</td>
<td>- relatively reliable schemata</td>
<td>- frequent schema changes</td>
</tr>
<tr>
<td>- fast growth</td>
<td>- moderate growth</td>
<td>- moderate growth</td>
</tr>
<tr>
<td>- little/no relationship support</td>
<td>- full relationship support</td>
<td>- specialized on relationships</td>
</tr>
<tr>
<td>- usually sacrifice ACID</td>
<td>- usually comply with ACID</td>
<td>- usually comply with ACID</td>
</tr>
<tr>
<td>- usually horizontal scaling</td>
<td>- usually vertical scaling</td>
<td>- usually vertical scaling</td>
</tr>
<tr>
<td>- data distribution</td>
<td>- data compression</td>
<td>- data optimization</td>
</tr>
<tr>
<td>- throughput</td>
<td>- transactions and security</td>
<td>- relationship traversal</td>
</tr>
<tr>
<td>- OLTP focus</td>
<td>- OLTP and OLAP</td>
<td>- OLAP focus</td>
</tr>
</tbody>
</table>

#### Distributed Data Management

- Data Models and Query Languages
- ThorstenPapenbrock
- Slide 85

Image: © https://neo4j.com/blog/aggregate-stores-tour/
Train your query skills with the following exercises:

- MongoDB
  - [https://www.w3resource.com/mongodb-exercises/](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](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