

Distributed Information Management

Daniel Kocher

Salzburg, Summer semester 2026

Department of Computer Science
University of Salzburg



Part I

Data Management

Literature, Sources, and Credits

Literature:

- Silberschatz et al. *Database System Concepts*. McGraw Hill, Sixth Edition, 2010.
- Wiese. *Advanced Data Management*. De Gruyter Oldenbourg, 2015.

Credits: These slides are partially based on slides of other lectures.

- Nikolaus Augsten, University of Salzburg, Austria (Credits also for valuable discussions and guidance).
- Andrew Pavlo, Carnegie Mellon University (CMU), USA.
- Pictograms from <https://flaticon.com>, users: freepik, imaginationlol
- Memes generated using <https://imgflip.com>

Introduction

The amount of **data is growing continuously and rapidly** in many different domains.

We do not collect data to simply store it, but we want to

- **access** it fast at any time and from any place,
- **search** it (for exact and similar patterns),
- **aggregate** it (partially),
- **join** it with other data,
- have **multiple users** working on the data concurrently,
- **make sense** out of it.

Motivation

“But I can organize my data myself, I do not need a dedicated system!”

Theoretically yes, but this implies:

- **Organizing your data** in multiple independent (plain) files.
- **Other users** may have trouble to understand your organization.
- Taking care of all the **requirements** yourself.
- You lose **flexibility and scalability**.

What could possibly go wrong?



Example Application

Managing the DAS faculty members.

Members:

Name	Employment	Department
Daniel Kocher	Senior Scientist	CS
Florian Graf	Postdoc. Researcher	AIHI
Bianca Löhnert	PhD Student	CS
Christoph Kirsch	Full Professor	CS
Michael Linortner	PhD Student	AIHI
Roland Kwitt	Full Professor	AIHI
Dirk Tiede	Assoc. Professor	Z_GIS
Simon Blatt	Assoc. Professor	Mathematics

Departments:

Name	Location
CS	Science City Salzburg
AIHI	Science City Salzburg
Z_GIS	Science City Salzburg
Mathematics	Faculty of Nat. Sciences (NAWI)

Example Application

Query: Find the names of all members that belong to the “CS” department.

```
1  Read "Members" table line by line
2  If value in column "Department" is equal to "CS" then
3    Report value in column "Name"
```

Listing 1: High-level pseudocode.

Redundancy and Inconsistency:

- Several copies of a datum may exist (possibly stored differently).
- **Redundancy:** Higher storage requirements.
How about accessing the data? $\Rightarrow$ Multiple file accesses (which is slow).
- **Inconsistency:** What happens if you have to update the data?
 $\Rightarrow$ We must not forget a single copy.
- **Goal:** Minimize redundancy while avoiding inconsistencies.

Example: Update “CS” to “Computer Science”.

Example Application

Query: Update “CS” to “Computer Science”.

Members:

Name	Employment	Department
Daniel Kocher	Senior Scientist	CS
Florian Graf	Postdoc. Researcher	AIHI
Bianca Löhnert	PhD Student	CS
Christoph Kirsch	Full Professor	CS
Michael Linortner	PhD Student	AIHI
Roland Kwitt	Full Professor	AIHI
Dirk Tiede	Assoc. Professor	Z_GIS
Simon Blatt	Assoc. Professor	Mathematics

Departments:

Name	Location
CS	Science City Salzburg
AIHI	Science City Salzburg
Z_GIS	Science City Salzburg
Mathematics	Faculty of Nat. Sciences (NAWI)

Data Access and Analysis:

- We want to analyze our data. How to link related data?
⇒ Each analysis requires a tailored program.
- **Goal:** Generic analysis and linkage of related data.

Example: List all members that belong to the “AIHI” department with their location.

Example Application

Query: List all members that belong to the “AIHI” department with their location.

Members:

Name	Employment	Department
Daniel Kocher	Senior Scientist	CS
Florian Graf	Postdoc. Researcher	AIHI
Bianca Löhnert	PhD Student	CS
Christoph Kirsch	Full Professor	CS
Michael Linortner	PhD Student	AIHI
Roland Kwitt	Full Professor	AIHI
Dirk Tiede	Assoc. Professor	Z_GIS
Simon Blatt	Assoc. Professor	Mathematics

Departments:

Name	Location
CS	Science City Salzburg
AIHI	Science City Salzburg
Z_GIS	Science City Salzburg
Mathematics	Faculty of Nat. Sciences (NAWI)

Data Integrity Issues:

- Updates may violate the integrity of your data.
- How do you ensure data integrity?
⇒ Each single application must respect all consistency constraints.
- **Goal:** Global definition and monitoring of consistency constraints.

Example: Insert “Thomas Blaschke” as member of the “CS” department.

Concurrency Issues:

- Multiple users should be able to access and update the data simultaneously. How do you ensure consistency over all applications that access the data?
- **Anomalies:** Inconsistencies, e.g., lost updates.
- **Efficiency:** If one user locks a file, then the other user must wait.
- **Goal:** Out-of-the-box multi-user operation without anomalies.

Example: User A inserts “Sebastian Forster” as member **while** user B inserts “Ana Sokolova” as member of the “CS” department simultaneously.

Example Application

Scenario: User A inserts “Sebastian Forster” as member **while** user B inserts “Ana Sokolova” as member of the “CS” department simultaneously.

Members:

Name	Employment	Department
Daniel Kocher	Senior Scientist	CS
Florian Graf	Postdoc. Researcher	AIHI
Bianca Lohnert	PhD Student	CS
Christoph Kirsch	Full Professor	CS
Michael Linortner	PhD Student	AIHI
Roland Kwitt	Full Professor	AIHI
Dirk Tiede	Assoc. Professor	Z_GIS
Simon Blatt	Assoc. Professor	Mathematics
Ana Sokolova	Full Professor	CS

Atomicity and Recovery:

- Data must neither be lost nor inconsistent when the system crashes.
- **Atomicity:** Data may be inconsistent if an operation is only applied partially
⇒ Execute an operation in an all-or-nothing manner.
- **Recovery:** Backup of data may not reflect the latest state.
- **Goal:** Avoid data loss and inconsistencies by design.

Example: Update all “CS” members to belong to the “Computer Science” department.
Your system **crashes in between** “Bianca Löhnert” and “Christoph Kirsch”.

Example Application

Scenario: Update all “CS” members to belong to the “Computer Science” department.
Your system **crashes in between** “Bianca Löhnert” and “Christoph Kirsch”.

Members:

Name	Employment	Department
Daniel Kocher	Senior Scientist	Computer Science
Florian Graf	Postdoc. Researcher	AIHI
Bianca Löhnert	PhD Student	Computer Science
Christoph Kirsch	Full Professor	CS
Michael Linortner	PhD Student	AIHI
Roland Kwitt	Full Professor	AIHI
Dirk Tiede	Assoc. Professor	Z_GIS
Simon Blatt	Assoc. Professor	Mathematics
Ana Sokolova	Assoc. Professor	CS

Other Issues:

- **Efficiency:** Efficient algorithms are required to analyze large amounts of data.
- **General-purpose:** The problems of application developers will partially overlap.
- **Security issues:** Flexible and fine-grained access rights for multiple users.

Database Management Systems

A **database management system** (DBMS) is

- (i) a collection of interrelated data, the **database**, and
- (ii) a **set of programs** to access and analyze the data.

In other words, you *do not* have to care about how to store the data (physically), how to analyze it (efficiently), how to (partially) update data, or how to deal with multiple users. A DBMS organizes all this for you.

DBMSs are **at the core of many applications**.

When Not to Use a DBMS

- The **data** are **too complex** to model it.
- **Specific requirements** like real-time queries or special operations.
- The **overhead** of a DBMS is **too high** or unnecessary.
- No or **low return on investment (ROI)**.

One Size Fits All?



General-Purpose DBMS

A DBMS that tries to fit as many application scenarios as possible with a single system. This implies higher complexity but also a large user base.

Examples:

- PostgreSQL (open source)
- MySQL (open source)
- MonetDB (open source)
- SQLite (open source)
- IBM DB2 (closed source)
- Oracle Database (closed source)
- Microsoft SQL Server (closed source)
- ...

“But this sounds good, no?”

Problems:

- **Overhead**, e.g., recovery or strong consistency
- **Native support** for application-specific operations
- Limited **performance**
- Limited **flexibility**



“One Size Fits All: An Idea Whose Time Has Come and Gone”

Michael Stonebraker¹ and Ugur Cetintemel (2005)

“A one size fits all database doesn’t fit anyone”

Werner Vogels² (2018)

¹Database Systems Researcher at the MIT. Won the Turing Award in 2014.

²Computer Scientist and CTO at Amazon.

Special-Purpose DBMS

A DBMS that is tailored to fit a specific purpose best, i.e., provide all the functionality that is required while also providing the best performance and flexibility (with respect to the specific application domain).

Synonyms: Specialized DBMS, purpose-built DBMS.

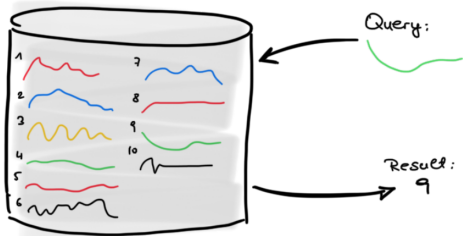
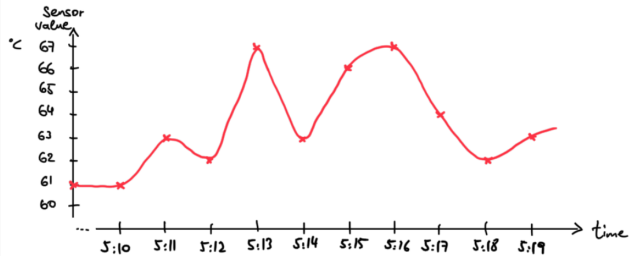
Special-Purpose DBMS – Temporal Data

A temporal DBMS is optimized to manage and analyze data that references time (i.e., they are timestamped). For example, a time series $x = \langle x_{t_1}, x_{t_2}, \dots, x_{t_n} \rangle$ is often a sequence of n data points that are spaced at strictly increasing times ($t_i < t_{i+1}$ with $i = 1, \dots, n - 1$).

Requirements:

- Exact/Approximate matching of (parts of) time series.
- Efficient compression mechanisms.
- Serve specific aspects like valid time or transaction time.
- ...

Special-Purpose DBMS – Temporal Data



Special-Purpose DBMS – Real-Time Data

A real-time DBMS manages data that is changed continuously. A DBMS that operates in real time answers the queries within a guaranteed time frame (the response time, i.e., it has a deadline).

Requirements:

- Answer every query in a given time frame.
- Query scheduling (or queuing).
- Consistency may not be that important.
- ...

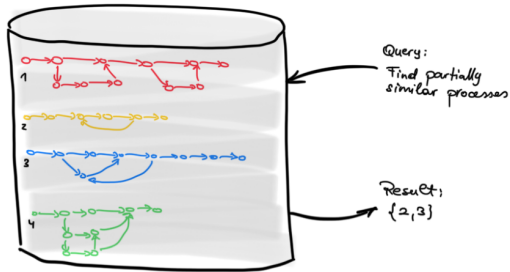
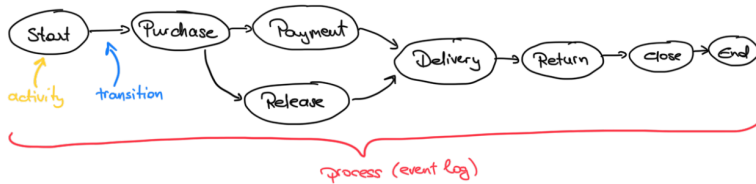


Process mining engines manage business event logs. An example event log is the sequence of activities if you place an order in some online shop. These systems are required to analyze large amounts of data in real time.

Requirements:

- Optimized, domain-specific language.
- Real-time performance for best user experience.
- ...

Special-Purpose DBMS – Process Mining Data



Multiple specific aspects may need to be combined to serve a novel application scenario. This may also result in a new special-purpose DBMS.

A modern application is not monolithic, i.e., different DBMSs may be used to implement different parts of an application.

“Towards a One Size Fits All Database Architecture”

Jens Dittrich³ and Alekh Jindal. 2011.

“One Size Fits all, Again! The Architecture of the Hybrid OLTP&OLAP Database Management System HyPer”

Alfons Kemper⁴ and Thomas Neumann⁵. 2011.

³Database Systems Researcher at Saarland University.

⁴Database Systems Researcher at the TU Munich co-author of the book *Datenbanksysteme*.

⁵Database Systems Researcher at the TU Munich.

Database Fundamentals



Basic Terminology

Data are facts that are to be stored.

Information is data combined with semantics (meaning).

Knowledge is information combined with an application.

Example Application

What are data, information, and knowledge in our example?

Members:

Name	Employment	Department
Daniel Kocher	Senior Scientist	CS
Florian Graf	Postdoc. Researcher	AIHI
Bianca Löhnert	PhD Student	CS
Christoph Kirsch	Full Professor	CS
Michael Linortner	PhD Student	AIHI
Roland Kwitt	Full Professor	AIHI
Dirk Tiede	Assoc. Professor	Z_GIS
Simon Blatt	Assoc. Professor	Mathematics

Departments:

Name	Location
CS	Science City Salzburg
AIHI	Science City Salzburg
Z_GIS	Science City Salzburg
Mathematics	Faculty of Nat. Sciences (NAWI)

A **database** (DB) is a collection of interrelated data.

Metadata provides us with information about the structure of a database. All the metadata are stored in a **catalog**.

A **database system** (DBS) is also referred to as the combination of a database, the corresponding metadata, and a DBMS (which in this case only provides the set of programs). The terms DBS and DBMS are often used interchangeably.

Example Application

Members:

Name	Employment	Department
Daniel Kocher	Senior Sc.	CS
Florian Graf	Postdoc.	AIHI
...	...	...

Departments:

Name	Location	Year
CS	Sci. City Salzburg	2022
AIHI	Sci. City Salzburg	2022
...	...	...

Catalog:

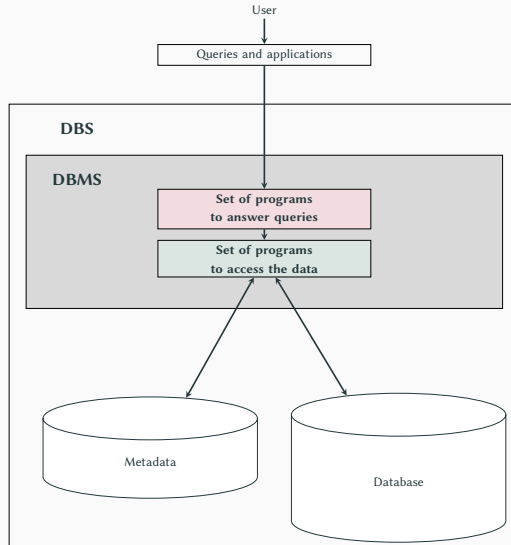
Tables Metadata:

Relation	RowCount
Members	8
Departments	4

Columns Metadata:

ColumnName	DataType	Relation
Name	TEXT	Departments
Year	INTEGER	Departments
Name	TEXT	Members
...	...	...

Database System



More Terminology

A **table** consists of multiple **tuples**, each of which is a sequence of **attributes**. Informally, a tuple can be imagined as a row of a table, whereas an attribute can be imagined as a column.

A **key** is subset of attributes. A **primary key** is a key of minimum length that uniquely identifies a tuple. A **foreign key** is a reference to a primary key.

More Terminology

A **schema** describes the overall structure of the data (often using tables) and is typically stable (i.e., rarely modified).

An **instance** is the information that is stored at a particular point in time (e.g., of a single tuple, a table, or an entire database). The instance may be frequently subject to changes.

Each level has its own schema with the logical schema being the most important one.

A **valid instance** satisfies all structural requirements and consistency constraints.

Data-Definition Language (DDL): Specify the structure of your data and the consistency constraints that are enforced.

- **Schema:** Describes the structure of your data and how the data are interrelated, e.g., a department has 3 columns: name, location, and year.
- **Consistency Constraints:** Describe integrity constraints that must be satisfied at any given point in time, e.g., the year of establishment is an integer ≥ 1622 .

Example Application

Members:

name	employment	department
Daniel Kocher	Senior Sc.	CS
Florian Graf	Postdoc.	AIHI
...	...	...

Departments:

name	location	year
CS	Sci. City Salzburg	2022
AIHI	Sci. City Salzburg	2022
...	...	...

Pseudo-DDL (simplified):

```
1 CREATE TABLE Members (  
2   name TEXT KEY,  
3   employment TEXT,  
4   department TEXT REFERENCES(Departments.name)  
5 )
```

```
CREATE TABLE Departments (  
  name TEXT KEY,  
  location TEXT,  
  year INTEGER (>= 1622)  
)
```

```
1  
2  
3  
4  
5
```

Data-Manipulation Language (DML): Query and manipulate your data.

- **Query Language:** Allows you to query the data without modifying it, e.g., get all members of the “CS” department or all departments in the “Science City Salzburg”.
- **Manipulation Language:** Allows you to modify your data, e.g., insert a new professor, delete an existing person, update the name of a particular department.

A **query** is a statement that requests some information. Informally, a query “asks” and the database system answers by returning the corresponding information.

Caveat: The term query language often refers to both DML parts.

Example Application

Members:

name	employment	department
Daniel Kocher	Senior Sc.	CS
Florian Graf	Postdoc.	AIHI
...	...	...

Departments:

name	location	year
CS	Sci. City Salzburg	2022
AIHI	Sci. City Salzburg	2022
...	...	...

Pseudo-DML (simplified):

```
1 SELECT department FROM Members
2 WHERE name = 'Daniel Kocher'
3
4 SELECT name FROM Members
5 WHERE employment = 'Senior Sc.'
```

```
INSERT INTO Members
VALUES ('Ana Sokolova', 'Full Prof.', 'CS')
3
UPDATE Departments
4 SET name = 'Computer Science'
5 WHERE name = 'CS'
6
```

Imperative Languages⁶: Describe a sequence of operations to retrieve the desired data.

Declarative Languages: Describe what data you are interested in (no specific steps).

Pure Languages: Form the (theoretical) foundation underneath the languages that are used in practice. Examples include relational algebra (imperative) or tuple calculus (declarative).

Our Focus: Declarative languages (and a bit of imperative languages).

⁶Imperative and procedural are often used synonymously.

Send someone to the supermarket to get milk.

The SQL Query Language

The **Structured Query Language (SQL)** was developed by IBM and is the de-facto standard language in database systems.

SQL is a declarative query language and includes DDL and DML elements.

The SQL standard (last revision: 2016) comprehensively summarizes all elements.

Inventing new approaches is like *“trying to swim up the Niagara Falls”*.

Michael Stonebraker in Information Age. 2010.

Example Queries in SQL

Data Definition:

```
1 CREATE TABLE Members (  
2   name VARCHAR(50) PRIMARY KEY,  
3   employment VARCHAR(50),  
4   department VARCHAR(100) REFERENCES Departments(name)  
5 );
```

```
1 CREATE TABLE Departments (  
2   name VARCHAR(100) PRIMARY KEY,  
3   location VARCHAR(75),  
4   year INTEGER NOT NULL CHECK(year >= 1622)  
5 );
```

- **VARCHAR(n), INTEGER:** Domain of a single column (data types).
- **NOT NULL, CHECK:** Constraints on a single column.
- **PRIMARY KEY, REFERENCES:** Constraints on an entire table.

Example Queries in SQL

Queries:

```
1 SELECT department FROM Members
2 WHERE name = 'Daniel Kocher';
3
4 SELECT name FROM Members
5 WHERE employment = 'Senior Sc.';
```

- **SELECT:** Specifies the column to retrieve.
- **FROM:** Specifies the tables to consider.
- **WHERE:** Specifies the condition(s) the result must satisfy.

Example Queries in SQL

Queries:

```
1 INSERT INTO Members(name, employment, department)
2 VALUES ('Ana Sokolova', 'Full Prof.', 'CS');
3
4 UPDATE Departments SET name = 'Computer Science'
5 WHERE year = 2022;
```

- **INSERT INTO ...VALUES:** Adds new tuple to a table based on the given values.
- **UPDATE ...SET:** (Partially) changes the values of a tuple.

Abstraction: Hide the complexity of the system (from people without deep computer science background) while providing all the functionality. Everyone should be able to use a database system.

Three Levels of Data Abstraction (bottom-up):⁷

1. **Physical: How** the data is stored (e.g., as sequence of bytes on hard disk, SSD, ...).
2. **Logical: What** data are stored and what relationships exist.
3. **View: Specific views** on the data (e.g., on a specific part of the entire database).

⁷Cf. also ANSI/SPARC architecture.

Three Levels of Data Abstraction

The **physical level** defines the **physical** data structures that are used to **store**, **organize**, and **access** the data **efficiently**. Examples include tables or access paths (e.g., using so-called indexes as shortcuts).

The **logical level** defines the **schemata** and **constraints** of the entire database. Physical data structures may be used underneath, but the user does not have to know them. $\Rightarrow$ **physical data independence**.

The **view level** reduces the complexity by providing only **information that is necessary for the respective user**. Irrelevant data are not shown and the focus is to simplify the interaction with the database system.

The Physical Level – Example

What is a **physical representation**⁸ of our **Members** table?

name	employment	department
Daniel Kocher	Senior Sc.	CS
Florian Graf	Postdoc.	AIHI
Bianca Löhnert	PhD Student	CS
Christoph Kirsch	Full Prof.	CS
Michael Linortner	PhD Student	AIHI
Roland Kwitt	Full Prof.	AIHI
Dirk Tiede	Assoc. Prof.	Z_GIS
Simon Blatt	Assoc. Prof.	Math.

```
Daniel Kocher;Senior Sc.;CS;
Florian Graf;Postdoc.;AIHI;
Bianca Löhnert;PhD Student;CS;
Christoph Kirsch;Full Prof.;CS;
Michael Linortner;PhD Student;AIHI;
Roland Kwitt;Full Prof.;AIHI;
Dirk Tiede;Assoc. Prof.;Z_GIS;
Simon Blatt;Assoc. Prof.;Mathematics;
```

1
2
3
4
5
6
7
8

A sequence of tuples with special characters as delimiters for attributes (;) and tuples (newline), respectively.

⁸This is an example – there are many possible physical representations.

The Logical Level – Example

What is a **logical representation**⁹ of our **Members** table?

name	employment	department
Daniel Kocher	Senior Sc.	CS
Florian Graf	Postdoc.	AIHI
Bianca Löhnert	PhD Student	CS
Christoph Kirsch	Full Prof.	CS
Michael Linortner	PhD Student	AIHI
Roland Kwitt	Full Prof.	AIHI
Dirk Tiede	Assoc. Prof.	Z_GIS
Simon Blatt	Assoc. Prof.	Math.

```
CREATE TABLE Members (  
  name VARCHAR(50) PRIMARY KEY,  
  employment VARCHAR(50),  
  department VARCHAR(100) REFERENCES Departments(name)  
);
```

1
2
3
4
5

Our table consists of three attributes with type definitions (**VARCHAR(n)**), constraints (**PRIMARY KEY**), and interrelationships (**REFERENCES** Departments(name)).

⁹Again, this is an example – there exist multiple different logical representations.

The View Level – Example

What is a **view** of our **Members** table?

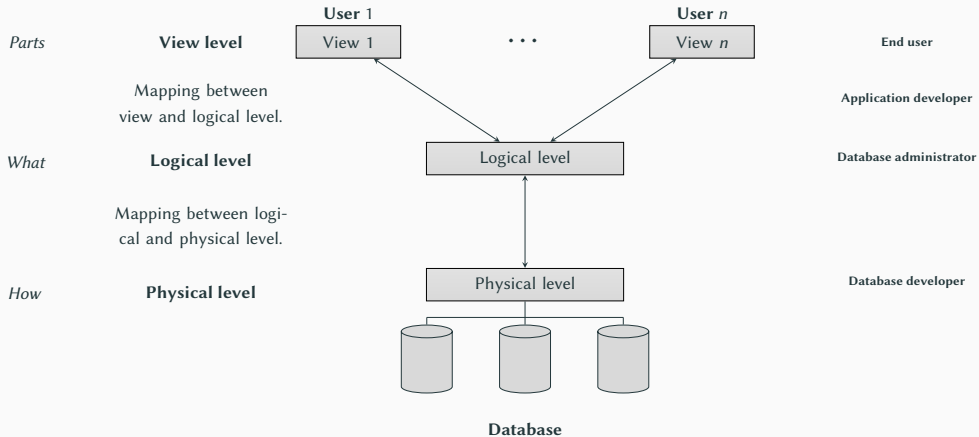
name	employment	department
Daniel Kocher	Senior Sc.	CS
Florian Graf	Postdoc.	AIHI
Bianca Löhnert	PhD Student	CS
Christoph Kirsch	Full Prof.	CS
Michael Linortner	PhD Student	AIHI
Roland Kwitt	Full Prof.	AIHI
Dirk Tiede	Assoc. Prof.	Z_GIS
Simon Blatt	Assoc. Prof.	Math.

name	employment	department
Bianca Löhnert	PhD Student	CS
Michael Linortner	PhD Student	AIHI

name	employment	department
Daniel Kocher	Senior Sc.	CS
Florian Graf	Postdoc.	AIHI
Bianca Löhnert	PhD Student	CS
Christoph Kirsch	Full Prof.	CS

Secretaries at the university may only have access to specific members depending on their area of responsibility (**PhD students**) or affiliation (**CS department**).

Three Levels of Data Abstraction



Mappings are used to link the abstraction levels.

Instance vs. Schema Revisited

A **schema** describes the overall structure of the data (often using tables) and is typically stable (i.e., rarely modified).

An **instance** is the information that is stored at a particular point in time (e.g., of a single tuple, a table, or an entire database). The instance may be frequently subject to changes.

Each level has its own schema with the logical schema being the most important one.

A **valid instance** satisfies all structural requirements and consistency constraints.

Instance vs. Schema Revisited

Schemata:

Members:

name	employment	department
------	------------	------------

Instances:

Members:

name	employment	department
Daniel Kocher	Senior Sc.	CS
Florian Graf	Postdoc.	AIHI
...	...	...

Departments:

name	location	year
------	----------	------

Departments:

name	location	year
CS	Sci. City Salzburg	2022
AIHI	Sci. City Salzburg	2022
...	...	...

Logical Data Independence: The ability to **update the logical schema transparently**, i.e., no change on the view level is required.

Physical Data Independence: The ability to **update the physical schema transparently**, i.e., no change on the logical level is required.

Benefits:

- Only the mapping between the levels need to be adapted.
- No change in the application required (it operates on the views).

(Declarative) Query Processing

Example SQL Query (+ Result):

```
1 SELECT name FROM Members  
2 WHERE employment = 'Senior Sc.';
```



name
Daniel Kocher

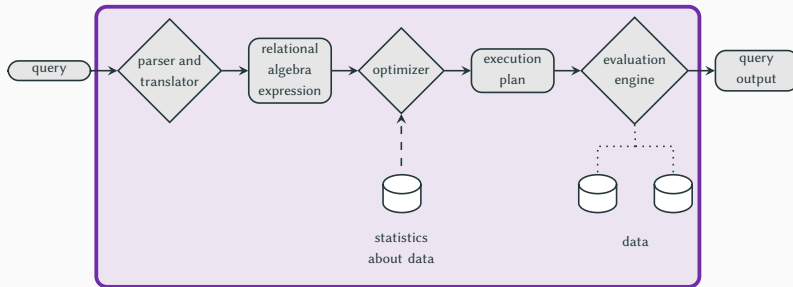
Query processing describes the process of extracting data from a database. In other words: What happens in a database when we issue a (declarative) query?

(Declarative) Query Processing

```
1 SELECT name FROM Members
2 WHERE employment = 'Senior Sc.';
```



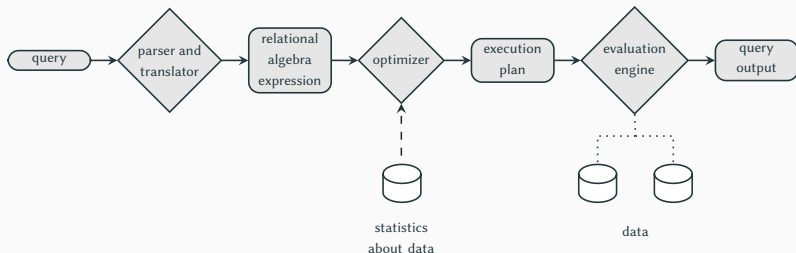
name
Daniel Kocher



(Declarative) Query Processing

On a high level, **three major components** are used to process a query:

1. **Parser:** Translates the query into an internal representation.
2. **Optimizer:** Chooses the most efficient way to evaluate the query.
3. **Evaluation Engine:** Executes the evaluation plan and returns the result.



(Declarative) Query Processing

An evaluation plan typically consists of multiple operation. Optimization is done based on the **estimated costs** of all involved operations.

For a given query, multiple valid **evaluation plans** may exist and must be **compared efficiently** (with respect to their estimated costs).

The estimated costs consider **many different factors** including access to hard disk, time to execute the query on the CPU, or network communication costs.

Data Models



Types of Data Models

3 types of data models that are somewhat related to the 3 levels of data abstraction.

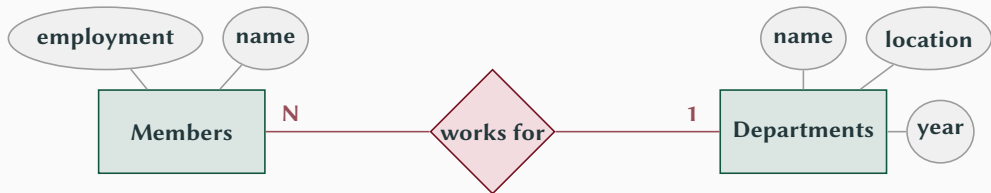
Conceptual data models: High level, i.e., only the schema is reflected but no instances. Related to the view level. Examples include Entity-Relationship (ER) and Unified Modeling Language (UML) models.

Logical data models: Depicts the instances and can be used to implement a database. Related to the logical level. Examples include the relational and the object-based models.

Physical data models: Low level, i.e., as close to the physical storage as possible. Related to the physical level and is typically system-specific.

Types of Data Models – Examples

Conceptual Model (ER):



Logical Model (Relational):

Members:

name	employment	department
Daniel Kocher	Senior Sc.	CS
Florian Graf	Postdoc.	AIHI
...	...	...

Departments:

name	location	year
CS	Sci. City Salzburg	2022
AIHI	Sci. City Salzburg	2022
...	...	...

Relational Data Model

Intuitive and **widely used** model. An example of a **record-based** model.

A **collection of relations (tables)** stores records of data as rows. A **tuple (row)** is an **entity** of the real world, an **attribute (column)** is a **property** of an entity. The **structure** of a record is **fixed**.

A **relation** has a **name** and a **set of (unique) attributes**. An **attribute** has a **name** and a **predefined domain**, i.e., values originate from a specific domain.

Tables are **filled row-wise** and a **row** represents the **state of an entity**.

Relational Data Model

The set of columns is called **relation schema**, and the set of relation schemata over all tables is called the **database schema**.

Intrarelatational constraints: Dependencies inside a single table, Σ_i

Interrelational constraints: Dependencies between different tables, Σ

Relation schema: $R_i = (\{A_{i1}, A_{i2}, \dots, A_{im}\}, \Sigma_i)^{10}$.

Database schema: $D = (\{R_1, R_2, \dots, R_n\}, \Sigma)$.

¹⁰ A_{ij} ... Name of the j -th attribute (column) of the i -th relation (table).

Relational Data Model – Example

Members:

name	employment	year	department
Daniel Kocher	Senior Sc.	2021	CS
Florian Graf	Postdoc.	2024	AIHI
...	...	...	...

Departments:

name	location	year
CS	Sci. City Salzburg	2022
AIHI	Sci. City Salzburg	2022
...	...	...

Relation schemata:

Members = ($\{\text{name, employment, year, department}\}, \{\text{name, year} \rightarrow \text{employment}\}$)

Departments = ($\{\text{name, location, year}\}, \{\text{name, year} \rightarrow \text{location}\}$)

Database schema:

DASFaculty = ($\{\text{Members, Departments}\}, \{\text{Members.department} \subseteq \text{Departments.name}\}$)

Normalization: Reduce anomalies (which lead to inconsistencies) by distributing attributes among tables and linking them using **foreign key constraints**.

Referential Integrity: Values of foreign keys exist as values in the referenced table, i.e., the referenced table contains at least one tuple that holds the value of the foreign key.

Relational Data Model – Example¹¹

Members:

id	name	employment	year	department
1	Daniel Kocher	Senior Sc.	2021	CS
2	Florian Graf	Postdoc.	2024	AIHI
3	Bianca Löhnert	PhD Student	2020	CS
4	Christoph Kirsch	Full Prof.	1995	CS
5	Michael Linortner	PhD Student	2022	AIHI
6	Roland Kwitt	Full Prof.	2019	AIHI
7	Dirk Tiede	Assoc. Prof.	2018	Z_GIS
8	Simon Blatt	Assoc. Prof.	2017	Math.

Departments:

id	name	location	year
201	CS	Sci. City Salzburg	2022
202	AIHI	Sci. City Salzburg	2022
203	Z_GIS	Sci. City Salzburg	2010
204	Math.	NAWI	1967

¹¹This is an example with synthetic data

Relational Data Model – Example¹¹

Members:

id	name	employment	year	department-id
1	Daniel Kocher	Senior Sc.	2021	201
2	Florian Graf	Postdoc.	2024	202
3	Bianca Löhnert	PhD Student	2020	201
4	Christoph Kirsch	Full Prof.	1995	201
5	Michael Linortner	PhD Student	2022	202
6	Roland Kwitt	Full Prof.	2019	202
7	Dirk Tiede	Assoc. Prof.	2018	203
8	Simon Blatt	Assoc. Prof.	2017	204

Departments:

id	name	location-id	year
201	CS	402	2022
202	AIHI	402	2022
203	Z_GIS	402	2010
204	Math.	403	1967

Locations:

id	location
402	Sci. City Salzburg
403	NAWI

Drawbacks:

- Relations may not be optimal to represent the data.
- Everything is a relation (semantic overloading).
- Homogeneous structure of data.
- Limited flexibility and data types.

Drawbacks and new challenges gave rise to **non-relational data models**.

Key-Value Data Model

Prototype of a **schemaless** model where each tuple is a pair of two strings (k, v) . k is a **unique key** that is associated with an arbitrary **value** v .

Data are accessed using the key and values are not interpreted by any means.

Example operations:

- `put(k, v)`: Inserts a new pair key-value pairs (k, v)
- `get(k)`: Retrieves the value associated with k .
- `delete(k)`: Deletes the tuple(s) associated with k .

Caveat: In programming often referred to as hash table or dictionary (Python).

Key-Value Data Model

Only keys can be searched, **values cannot be searched**. Combinations of values must be done by the application that accesses the data.

Easy to distribute $\Rightarrow$ Good for **data-intensive applications**.

Example Key-Value Store¹²:

Key	references	Value
User-ID		Shopping Cart
1002	→	Shoe, Jordans, red, 37 # Computer game “Diablo”, Blizzard # Headset, Razer, Kraken Kitty
1003	→	Wilson American Football, NFL, Replica # Hail Mary, Gloves Receiver, 2.0, Black & White
1007	→	Book “Database System Concepts”, 7th Edition, Silberschatz
...	→	...

¹²The “#” symbol is an example delimiter; you can basically choose the delimiter yourself.

Document-Based Data Model

A document stores data in a semi-structured and nested text format (e.g., XML or JSON).

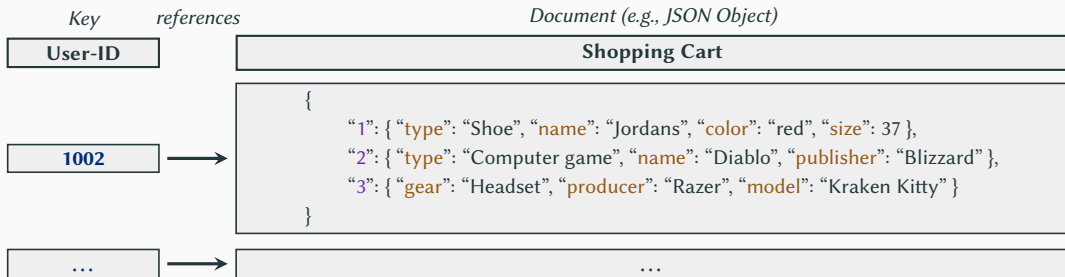
Each document has a unique identifier, but the value is a document structured in a specific format that is interpretable (as opposed to a key-value store).

JavaScript Object Notation (JSON): Human-readable text format for data structures. A JSON document consists of possibly nested key-value pairs.

Document-Based Data Model

A JSON document consists of a JSON object, which is enclosed by curly braces, { . . . }. Inside, keys and the corresponding values are separated by a colon, and a value can be a JSON object itself.

Example Document Store:



Graph-Based Data Model

Graphs: Informally, graphs are structures that represent data (as nodes) and their interrelation (as edges in between) by design. Both nodes and edges may carry information. Efficient graph operations are supported natively.

Data and their relationship are distinct naturally (cf. semantic overloading of the relational data model).

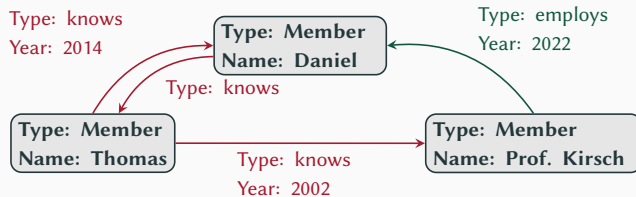
Example Graph:



Graph-Based Data Model

Property Graph Model: Multiple types for nodes/edges (multi-relational), each of which may contain multiple properties (attributes) as name-value pairs (similar to key-value pairs).

Example Property Graph:



Extensible Record Data Model

Also called **wide column data model**: A generalization of Google's BigTable¹³ system.

Table cells are represented as (2-dimensional) key-value pairs, with row (unique) and column (repeatedly) being the keys, i.e., a 2-dimensional key-value data model.

Example: Access table cell (4:employment).

id	name	employment	year	department
1	Daniel Kocher	Senior Sc.	2021	CS
2	Florian Graf	Postdoc.	2024	AIHI
3	Bianca Löhnert	PhD Student	2020	CS
4	Christoph Kirsch	Full Prof.	1995	CS
5	Michael Linortner	PhD Student	2022	AIHI
...	...	...	...	...

¹³Chang et al. *Bigtable: A Distributed Storage System for Structured Data*. OSDI, 2006.

Extensible Record Data Model

Consider the following table:

id	name	employment	year	department
1	Daniel Kocher	Senior Sc.	2021	CS
2	Florian Graf	Postdoc.	NULL	AIHI
3	Bianca Löhnert	PhD Student	2020	CS
4	Christoph Kirsch	Full Prof.	1995	CS
...	...	...	...	...

Represent it as a set of cells:

1

name	employment	year	department
Daniel Kocher	Senior Sc.	2021	CS

2

name	employment	department
Florian Graf	Postdoc	AIHI

Extensible Record Data Model

Column Families: Groups of related columns that are accessed together.

1

name	employment	year	department
Daniel Kocher	Senior Sc.	2021	CS

2

name	employment	department
Florian Graf	Postdoc	AIHI

PersonInfo:

1	<table><tr><th>name</th></tr><tr><td>Daniel Kocher</td></tr></table>	name	Daniel Kocher	<table><tr><th>department</th></tr><tr><td>CS</td></tr></table>	department	CS
name						
Daniel Kocher						
department						
CS						
2	<table><tr><th>name</th></tr><tr><td>Florian Graf</td></tr></table>	name	Florian Graf	<table><tr><th>department</th></tr><tr><td>AIHI</td></tr></table>	department	AIHI
name						
Florian Graf						
department						
AIHI						

EmploymentInfo:

1	<table><tr><th>employment</th></tr><tr><td>Senior Sc.</td></tr></table>	employment	Senior Sc.	<table><tr><th>year</th></tr><tr><td>2021</td></tr></table>	year	2021
employment						
Senior Sc.						
year						
2021						
2	<table><tr><th>employment</th></tr><tr><td>Postdoc</td></tr></table>	employment	Postdoc			
employment						
Postdoc						

Extensible Record Data Model

Encourages de-normalization (redundancy) for higher performance and provides **more flexibility**, i.e., every row may be composed of different columns.

Scales to enormous amounts of data, i.e., **petabyte-scale data**.

Favors column-oriented over row-oriented data storage (like the relational model).
⇒ Related columns are stored together on disk (rather than rows/tuples).

The query workload defines how the data is modeled ⇒ Know your workload.

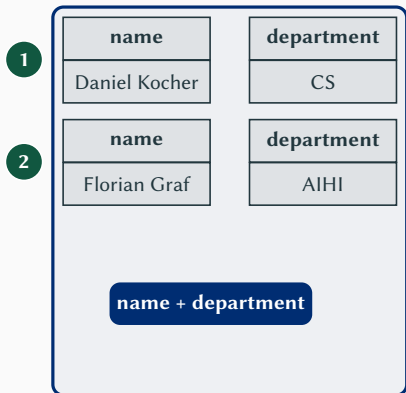
Full key access to a value: (<rowid>:<columnfamily>:<columnname>).

For example: (2:EmploymentInfo:employment) returns Postdoc.

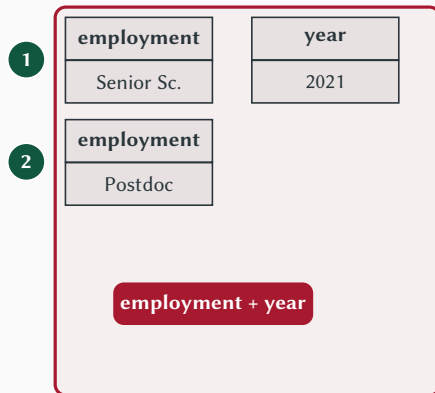
Extensible Record Data Model

Columns that belong to the **same column family** are stored together on disk.

PersonInfo:



EmploymentInfo:



Vector-Based Data Model

Embeddings: Objects are transformed into **fixed-size vectors** of d floating-point values, e.g., $[0.1, 0.2, 0.3, 0.75, 0.45, 0.6]$ is a 6-dimensional vector.

Resulting vectors $v_1, v_2, \dots$ are organized in a **d -dimensional vector space** with respect to some similarity function, e.g., Cosine similarity.

Goal: Similar objects “appear” (i.e., are located) near each other in this vector space.

Many different embedding techniques exist, e.g.,

- **word2vec** uses a neural network trained to recognize the context of a word
- **(S)BERT** uses a 12-layer deep neural network trained to understand language
- ...

Vector-Based Data Model

Strings

Cat

Banana

Dog

Apple

House

Wolf

Labrador

Embeddings/Vectors

1.5	-0.4	7.2	19.6	20.2
-----	------	-----	------	------

-4.9	3.6	0.9	7.8	3.6
------	-----	-----	-----	-----

1.7	-0.3	6.9	19.1	21.1
-----	------	-----	------	------

-5.2	3.1	0.2	8.1	3.5
------	-----	-----	-----	-----

60.1	-60.3	10.0	-12.3	9.2
------	-------	------	-------	-----

-0.2	-3.7	2.9	15.5	14.4
------	------	-----	------	------

1.6	-0.3	7.0	19.3	21.2
-----	------	-----	------	------

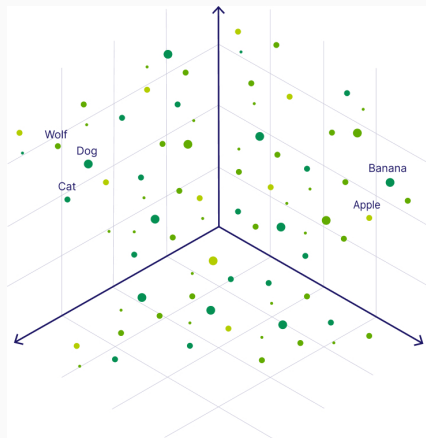


Image on right-hand side taken from <https://weaviate.io/blog/vector-embeddings-explained>.

Array-Based Data Model

Complex structures (like 2-dimensional satellite or n -dimensional sensor data) are organized along multiple dimensions.

An array cell contains a tuple of a specific length and a tuple element may contain a value or another array (allowing arbitrary nestings).

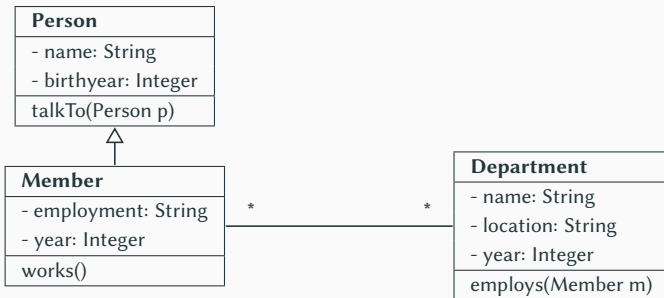
Specialized array functionality is natively supported (e.g., joins and aggregations).

Object-Based Data Model

Many programming languages are object-oriented, i.e., based on the concept of objects.

Objects contain data and provide functionality, and interact with each other (e.g., like human beings in the real world).

Example Objects:



Three Options:

1. **Object-relational data model:** Extends relational data model with object-oriented features.
2. **Object-relational mapping (ORM):** Maps objects to tuples in the relational model and handles the translation.
3. **Object-oriented data model:** Implements an object-based data model natively.

Workloads & Challenges

- New data management **challenges**.
- Basics of **distributed** database systems (also compared to parallel DBS).
- **Consistency guarantees**: ACID vs. BASE.
- The **CAP** Theorem.
- **Workloads**: OLTP vs. OLAP, batch vs. stream processing.

New Data Management Challenges

- Data may be organized in **complex structures** (e.g., graphs).
- Data may be **schemaless** (e.g., not comply to a fixed schema).
- Data may be **sparse** (e.g., values may be non-existent).
- Data or schema may be **constantly changing** (e.g., schema evolution).
- Data may be **distributed over multiple machines**.
 - Access must be **transparent** (i.e., user need not know where data resides).
 - Systems must **scale horizontally** (i.e., new machines come and go).
 - Systems must cope with **large data volumes**.

New Data Management Challenges

NoSQL (Not Only SQL):

- Class of non-relational DBSs.
- Weaker consistency guarantees (e.g., BASE¹⁴)
- Support for schema independence.
- Highly scalable.

NewSQL Movement:

- Class of relational DBSs that aim at
 - providing NoSQL-like scalability (for OLTP⁹ workloads), and
 - retaining strong consistency guarantees like RDBMS (e.g., ACID⁹).

¹⁴We will cover this term subsequently.

Parallel Database Systems

Parallel machines are ubiquitous, i.e., CPUs, memory¹⁵, and disks are affordable.

In database systems:

- **Parallel I/O:** Data is partitioned across multiple disks.
⇒ Reduces the time to retrieve data from disk
- **Parallel Execution:** Execute individual operations in parallel, e.g., sorting, joins, ...
⇒ Each CPU (core) may work independently on its own data partition.

Makes database systems **more complicated**:

- Queries that are executed in parallel **must not interfere with each other**.
⇒ Resolved by, e.g., locking mechanisms and/or concurrency control.

¹⁵*Caveat:* Currently, the prices for main memory are very high due to the AI hype.

Interquery Parallelism:

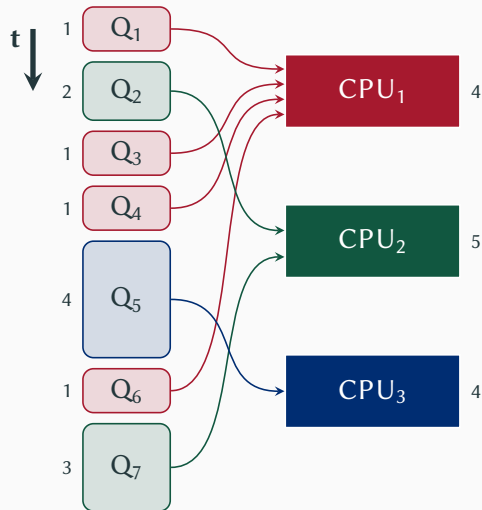
- Queries may execute in parallel **with one another**.
- Support for a larger overall number of queries.
- Easiest form of parallelism.

Intraquery Parallelism:

- A single query is executed in parallel on multiple CPUs (and disks).
- Parallelizes either **within** an operation, or **across** different operations.
- Important to speed up long-running queries.

Interquery Parallelism

Example: Seven queries Q_1-7 with different runtimes, and 3 CPU (cores) CPU_{1-3} .



Sequential execution on one CPU (core):
 $\Rightarrow 4 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 + 1 \cdot 4 = \mathbf{13 \text{ time units}}$

Parallel execution on three CPU (cores):

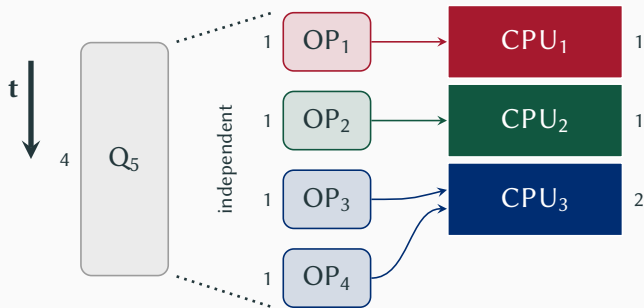
- Four queries ($Q_{1,3,4,6}$) on CPU_1
- Two queries ($Q_{2,7}$) on CPU_2
- One query (Q_5) on CPU_3

$\Rightarrow \max(4 \cdot 1, 1 \cdot 2 + 1 \cdot 3, 1 \cdot 4) = \mathbf{5 \text{ time units}}$

Intraquery Parallelism

Example: One query Q_5 with four operations OP_{1-4} , and 3 CPU (cores) CPU_{1-3} .

Assumption for simplicity: Operations are fully independent and run one time unit



Sequential execution:

$$\Rightarrow 4 \cdot 1 = 4 \text{ time units}$$

Parallel execution:

- OP_1 on CPU_1
- OP_2 on CPU_2
- $OP_{3,4}$ on CPU_3

$$\Rightarrow \max(1, 1, 2) = 2 \text{ time units}$$

Parallel database management systems (PDBMSs) have multiple processors and hard disks that are connected via a fast interconnection.

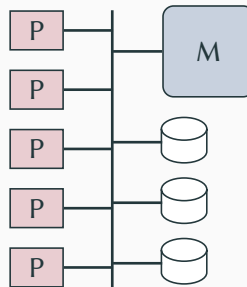
Performance characteristics:

- **Throughput:** Number of tasks (e.g., queries) that can be completed in a given time frame. Example: Queries per seconds.
- **Response time:** Time it takes to complete a single task (e.g., query).

Parallel Database System Architectures

Shared memory DBMSs have many processors and disks that share a common memory (typically via a bus).

- + Efficient communication between processors ($< 1\mu s$).
- Limited scalability (≤ 64 processors; interconnection to memory becomes the bottleneck).

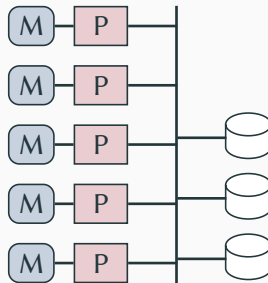


P ... processor, M ... memory, $\ominus$... disks.

Parallel Database System Architectures

Shared disk DBMSs have many processors (with isolated memory) that share all disks (typically via a bus).

- + Scale to a larger number of processors.
- Communication between processors is slower (ms; bottleneck now at interconnection to disks).

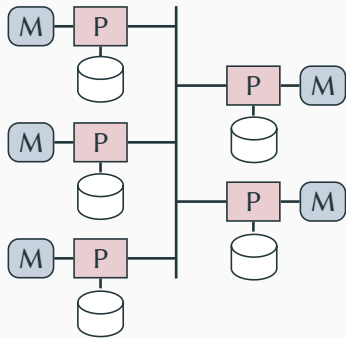


P ... processor, M ... memory,  ... disks.

Parallel Database System Architectures

Shared nothing DBMSs have many processors with isolated memory and disk(s). The combination of a processor with isolated memory and disk(s) is also referred to as **node**.

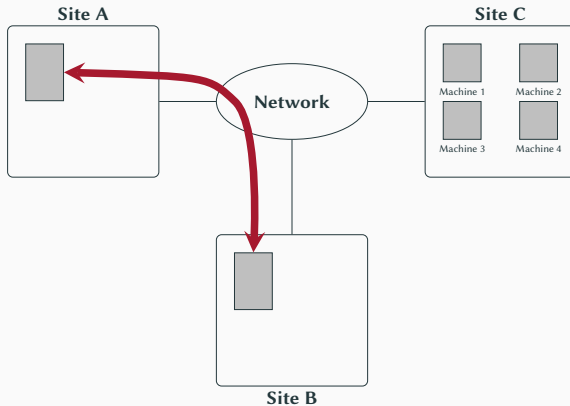
- + Scale to thousands of processors.
- Communication between processors is slow; access to non-local disk data is slow.



P ... processor, M ... memory,  ... disks.

Distributed Database Systems

Distributed database management systems (DDBMSs) are DBMSs that operate on multiple machines that are located at geographically separated **sites**¹⁶.



¹⁶A site can be imagined as a data center with multiple machines.

DDBMS vs. Shared-Nothing PDBMS

- Sites within a DDBMS are typically
 - **geographically separated** (i.e., not a single data center)
⇒ lower bandwidth (less throughput), higher latency (higher response time).
 - **separately administered** (i.e., retain some degree of autonomy).
- PDBMS can deal with node failures, whereas **DDBMS** can deal with **failures of entire sites** (e.g., due to natural disasters).
- DDBMS distinguish between **local** and **global transactions**.

Homogeneous vs. Heterogeneous DDBMS

A DDBMS is called **homogeneous** if the nodes share a common global database schema, perform the same tasks (e.g., run the same software), and actively cooperate in processing. Goal: View of a single database.

A DDBMS is called **heterogeneous** if the nodes have different schemata, perform different tasks (e.g., run different software), and may not be aware of other nodes. Goal: Integrate different databases.

Transactions

A **transaction** refers to a **sequence of operations** that accesses and (possibly) updates various data items. A transaction **transitions the database from one consistent state into another consistent state**.

Example: Transfer EUR 500 from bank account A to bank account B.

1. BEGIN
2. READ(A)
3. $A = A - 500$
4. WRITE(A)
5. READ(B)
6. $B = B + 500$
7. WRITE(B)
8. COMMIT

account	balance
A	800
B	2000



account	balance
A	300
B	2500

A DBMS must deal with **two major issues**:

- **System crash**: Software or hardware failures.
- **Concurrency**: Many different users may work at the same time (i.e., multiple transactions are executed).

A DBS implements particular consistency guarantees. Most relational DBMS manage transactions according to the so-called **ACID properties**: **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

The ACID properties are considered **strong consistency guarantees**.

Atomicity: Execute **all operations** of a transaction **or none** of them (*“all or nothing”*).

Example: Transfer EUR 500 from bank account A to bank account B.

1. BEGIN
2. READ(A)
3. $A = A - 500$
4. WRITE(A)
5. READ(B)
6. $B = B + 500$
7. WRITE(B)
8. COMMIT

account	balance
A	800
B	2000

What happens if the system crashes after step 4?

ACID Properties

Consistency: After a transaction, all values in the database are correct (i.e., consistency is preserved).

Example: Transfer EUR 500 from bank account A to bank account B.

1. BEGIN
2. READ(A)
3. $A = A - 500$
4. WRITE(A)
5. READ(B)
6. $B = B + 500$
7. WRITE(B)
8. COMMIT

account	balance
A	800
B	2000



account	balance
A	300
B	2500

What is a consistency constraint in the above example? $A + B$ is the same before and after the transaction.

ACID Properties

Isolation: A transaction must be unaware of other simultaneously executing transactions (otherwise an inconsistent state may be encountered)¹⁷.

Example: Transfer EUR 500 from bank account A to bank account B.

1. BEGIN
2. READ(A)
3. $A = A - 500$
4. WRITE(A)
5. READ(B)
6. $B = B + 500$
7. WRITE(B)
8. COMMIT

What does BEGIN; READ(A); READ(B); print(A + B); COMMIT in between steps 4 and 5 print?

¹⁷ For every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished.

ACID Properties

Durability: Successful transaction results persist in the database, even if the system crashes.

Example: Transfer EUR 500 from bank account A to bank account B.

1. BEGIN
2. READ(A)
3. $A = A - 500$
4. WRITE(A)
5. READ(B)
6. $B = B + 500$
7. WRITE(B)
8. COMMIT

If the system crashes and the EUR 500 are lost, nobody is happy.

Transactions in PostgreSQL

Documentation -- PostgreSQL 13

Supported Versions: [Current \(13\)](#) / [12](#) / [11](#) / [10](#) / [9.6](#)

Development Versions: [devel](#)

Unsupported versions: [9.5](#) / [9.4](#) / [9.3](#) / [9.2](#) / [9.1](#) / [9.0](#) / [8.4](#) / [8.3](#) / [8.2](#) / [8.1](#) / [8.0](#) / [7.4](#) / [7.3](#) / [7.2](#)

[Prev](#)[Up](#)

3.4. Transactions
Chapter 3. Advanced Features

[Home](#)[Next](#)

3.4. Transactions

Transactions are a fundamental concept of all database systems. The essential point of a transaction is that **it bundles multiple steps into a single, all-or-nothing operation**. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of \$100.00 from Alice's account to Bob's account. Simplifying outrageously, the SQL commands for this might look like:

```
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

The details of these commands are not important here; the important point is that there are several separate updates involved to accomplish this rather simple operation. Our bank's officers will want to be assured that either all these updates happen, or none of them happen. It would certainly not do for a system failure to result in Bob receiving \$100.00 that was not debited from Alice. Nor would Alice long remain a happy customer if she was debited without Bob being credited. We need a guarantee that if something goes wrong partway through the operation, none of the steps executed so far will take effect. **Grouping the updates into a transaction gives us this guarantee. A transaction is said to be atomic**; from the point of view of other transactions, it either happens completely or not at all.

We also want a guarantee that once a transaction is completed and acknowledged by the database system, **it has indeed been permanently recorded and won't be lost even if a crash ensues shortly thereafter**. For example, if we are recording a cash withdrawal by Bob, we do not want any chance that the debit to his account will disappear in a crash just after he walks out the bank door. A transactional database guarantees that all the updates made by a transaction are logged in permanent storage (i.e., on disk) before the transaction is reported complete.

Another important property of transactional databases is closely related to the notion of atomic updates: **when multiple transactions are running concurrently, each one should not be able to see the incomplete changes made by others**. For example, if one transaction is busy totalling all the branch balances, it would not do for it to include the debit from Alice's branch but not the credit to Bob's branch, nor vice versa. So transactions must be all-or-nothing not only in terms of their permanent effect on the database, but also in terms of their visibility as they happen. The updates made so far by an open transaction are invisible to other transactions until the transaction completes, whereupon all the updates become visible simultaneously.

In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with `BEGIN` and `COMMIT` commands. So our banking transaction would actually look like:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
-- etc etc
COMMIT;
```

Programmers must ensure to properly define the transactions to preserve consistency.

A DBS typically includes a so-called **transaction manager**, which ensures that the transactions comply to the consistency guarantees (e.g., the ACID properties).

A transaction has **committed** if it completes successfully. Otherwise, the transaction is **aborted**. Undoing the changes of an aborted transaction is referred to as **rollback**.

Transaction Management – Atomicity

Before the DBS changes the database, it writes some information (transaction identifier, old/new values) about the changes into a so-called **log file**.

Example: Transfer EUR 500 from bank account A (EUR 800) to account B (EUR 2,000).

Transaction T_1 :

1. BEGIN
2. READ(A)
3. $A = A - 500$
4. WRITE(A)
5. READ(B)
6. $B = B + 500$
- [CRASH]---
7. WRITE(B)
8. COMMIT

Pseudo log file:

T1: OLD(A=800), NEW(A=300)

Database:

UPDATE(A=300)

ROLLBACK T1 on restart

Transaction Management – Lock-Based Isolation

Example: Transfer EUR 500 from bank account A (EUR 800) to account B (EUR 2,000).

Transaction T_1 :

1. BEGIN
2. READ(A)
3. $A = A - 500$
4. WRITE(A)
5. READ(B)
6. $B = B + 500$
7. WRITE(B)
8. COMMIT

Transaction T_2 :

1. BEGIN
2. READ(A)
3. $A = A + 1,000$
4. WRITE(A)
5. COMMIT

Database:

- | | |
|------------------------------|----------|
| LOCK(A, T1) | OK |
| UPDATE(A=300) | |
| LOCK(A, T2) | CONFLICT |
| LOCK(B, T1) | OK |
| UNLOCK(A, T1), UNLOCK(B, T1) | OK |

Strict serial execution of concurrent transactions guarantees isolation, but severely limits the performance.

A **concurrency-control scheme** ensures that transactions can execute concurrently (i.e., their operations can be interleaved). Interleaving the operations of a transaction is called **schedule** and may result in a correct database state, or not!

A concurrent schedule is **serializable** if an equivalent serial schedule exists, i.e., the **result** produced by a concurrent schedule is **identical** to some serial schedule (of the same transactions).

Transaction Management – Schedules

Assume two transactions T_1 , T_2 that change the balance of bank accounts A , B .

Transaction T_1 :

1. READ(A)
2. $A = A + 100$
3. WRITE(A)
- 4.
5. READ(B)
- 6.
- 7.
- 8.
9. $B = B + 100$
10. WRITE(B)

Transaction T_2 :

- 1.
- 2.
- 3.
4. READ(A)
- 5.
6. $A = A * 2$
7. WRITE(A)
8. READ(B)
- 9.
- 10.
11. $B = B * 2$
12. WRITE(B)

Transaction Management – (Strict) Serial Schedules

Assume two transactions T_1 , T_2 that change the balance of bank accounts A , B .

Transaction T_1 : Transaction T_2 :

1. READ(A)
2. $A = A + 100$
3. WRITE(A)
4. READ(B)
5. $B = B + 100$
6. WRITE(B)

7. READ(A)
8. $A = A * 2$
9. WRITE(A)
10. READ(B)
11. $B = B * 2$
12. WRITE(B)

Transaction T_1 : Transaction T_2 :

1. READ(A)
2. $A = A * 2$
3. WRITE(A)
4. READ(B)
5. $B = B * 2$
6. WRITE(B)

7. READ(A)
8. $A = A + 100$
9. WRITE(A)
10. READ(B)
11. $B = B + 100$
12. WRITE(B)

Transaction Management – Non-Serializable Concurrent Schedule

Assume two transactions T_1 , T_2 that change the balance of bank accounts A , B .

Transaction T_1 :

1. READ(A)
2. $A = A + 100$
3. WRITE(A)

10. READ(B)
11. $B = B + 100$
12. WRITE(B)

Transaction T_2 :

4. READ(A)
5. $A = A * 2$
6. WRITE(A)
7. READ(B)
8. $B = B * 2$
9. WRITE(B)

Transaction Management – Serializable Concurrent Schedule

Assume two transactions T_1 , T_2 that change the balance of bank accounts A , B .

Transaction T_1 :

1. READ(A)
2. $A = A + 100$
3. WRITE(A)

7. READ(B)
8. $B = B + 100$
9. WRITE(B)

Transaction T_2 :

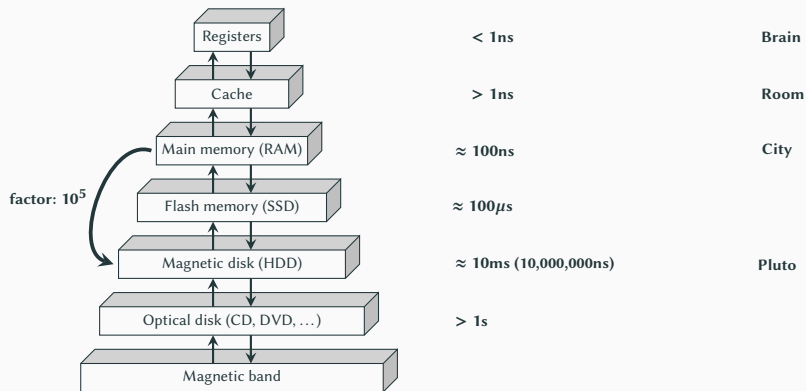
4. READ(A)
5. $A = A * 2$
6. WRITE(A)

10. READ(B)
11. $B = B * 2$
12. WRITE(B)

Updating the persistent data may be postponed to improve performance, i.e., the data is provisionally updated in main memory (RAM). Data in main memory is volatile, i.e., it is lost on system restart.

Excursion: Nowadays, data can be considered persistent if it is written to hard disk. DBMSs aim to be efficient/fast, thus they try to **avoid** or **postpone expensive/slow operations** like accesses to hard disk. Multiple data structures are maintained in main memory, which are commonly referred to as **buffers**. The content of the buffers is **only written to disk if inevitable**.

Memory Hierarchy Revisited



Transaction Management – Durability

Example: Transfer EUR 500 from bank account A (EUR 800) to account B (EUR 2,000).

Transaction T_1 :

1. BEGIN
2. READ(A)
3. $A = A - 500$
4. WRITE(A)
5. READ(B)
6. $B = B + 500$
7. WRITE(B)
8. COMMIT

Pseudo log file:

T1: OLD(A=800), NEW(A=300)

T1: OLD(B=2,000), NEW(B=2,500)

Database:

postponed UPDATE(A=300)

postponed UPDATE(B=2,500)

---[CRASH]---

A DDBMS must consider additional aspects:

- **Distributed data storage** (replication and fragmentation).
- **Distributed transactions** (local and global).

Distributed Data Storage

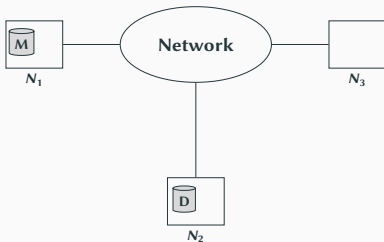
Assume the tables of our faculty database and the relational data model.

Members:

id	name	employment	year	department
1	Daniel Kocher	Senior Sc.	2021	CS
2	Florian Graf	Postdoc.	2024	AIHI
...	...	...	...	...

Departments:

id	name	location	year
201	CS	Sci. City Salzburg	2022
202	AIHI	Sci. City Salzburg	2022
...	...	...	...

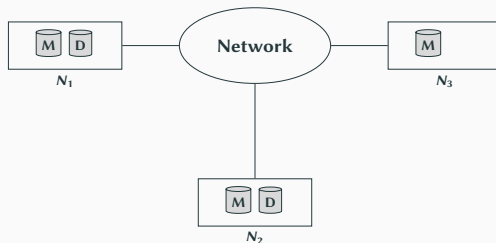


What if N_1 gets disconnected?

Distributed Data Storage – Replication

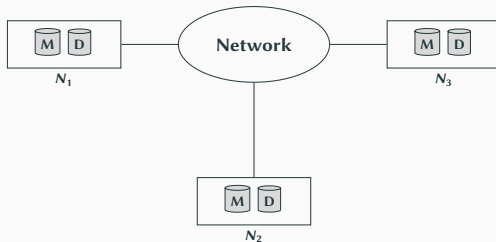
Replication: Data are **replicated** among multiple sites, i.e., a copy (replica) of the same data exists in different sites (intentional redundancy).

Example Replication:



Distributed Data Storage – Replication

Full replication: A copy of a relation is stored at **all sites**.



Fully replicated database: Every site contains a copy of the entire database.

Pros: Higher availability and performance, reduced data transfer.

Cons: Increased update costs and higher complexity of concurrency control.

Replication in MongoDB (Document-Based)

[SERVER](#)[DRIVERS](#)[CLOUD](#)[TOOLS](#)[GUIDES](#)[Get MongoDB](#)

Replication

A *replica set* in MongoDB is a group of [mongod](#) processes that maintain the same data set. Replica sets provide redundancy and [high availability](#), and are the basis for all production deployments. This section introduces replication in MongoDB as well as the components and architecture of replica sets. The section also provides tutorials for common tasks related to replica sets.

Redundancy and Data Availability

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.

In some cases, replication can provide increased read capacity as clients can send read operations to different servers. Maintaining copies of data in different data centers can increase data locality and availability for distributed applications. You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

Distributed Data Storage – Fragmentation

Fragmentation: Data are **partitioned into fragments** stored at distinct sites, i.e., a specific part of the data is stored at a site.

Horizontal fragmentation: Relation is split row- or tuple-wise, and each row/tuple resides at a separate site. Allows **parallel processing** on **fragments** of a relation.

Vertical fragmentation: Relation is split into smaller subschemata (based on the columns/attributes), and each subschema resides at a separate site. Allows **parallel processing** on a **relation**.

Example Horizontal Fragmentation

Members M1:

id	name	employment	year	department
1	Daniel Kocher	Senior Sc.	2021	CS
2	Florian Graf	Postdoc.	2024	AIHI

Members M2:

id	name	employment	year	department
3	Bianca Löhnert	PhD Student	2020	CS
4	Christoph Kirsch	Full Prof.	1995	CS

Members M3:

id	name	employment	year	department
5	Michael Linortner	PhD Student	2022	AIHI
6	Roland Kwitt	Full Prof.	2019	AIHI

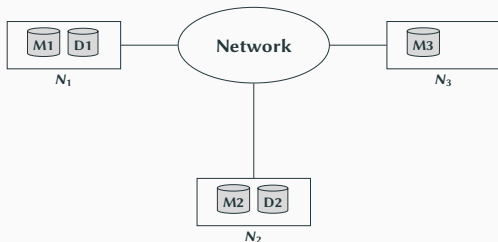
Departments D1:

id	name	location	year
201	CS	Sci. City Salzburg	2022
202	AIHI	Sci. City Salzburg	2022

Departments D2:

id	name	location	year
203	Z_GIS	Sci. City Salzburg	2010
204	Math.	NAWI	1967

Example Horizontal Fragmentation



Members M1:

id	name	employment	year	department
1	Daniel Kocher	Senior Sc.	2021	CS
2	Florian Graf	Postdoc.	2024	AIHI

Members M2:

id	name	employment	year	department
3	Bianca Löhnert	PhD Student	2020	CS
4	Christoph Kirsch	Full Prof.	1995	CS

Members M3:

id	name	employment	year	department
5	Michael Linortner	PhD Student	2022	AIHI
6	Roland Kwitt	Full Prof.	2019	AIHI

Distributed Data Storage – Fragmentation

Horizontal and vertical fragmentation can be **mixed**. In any case, the fragmented data must be **reconstructable**.

Pros: Higher performance (parallel processing), better locality (reduced data transfer).

Cons: Increased costs on site failure and if data is retrieved from different sites.

Distributed Data Storage – Replication, Fragmentation, and Transparency

Fragmentation and **Replication** can be **combined**:

- Relation is partitioned into several fragments.
- System maintains several identical replicas of each such fragment.

Data Transparency: User may be unaware of how and where data items are stored.

- Fragmentation transparency
- Replication transparency
- Location transparency

Local transactions access/update data at only one (local) site. A local transaction manager enforces the ACID properties.

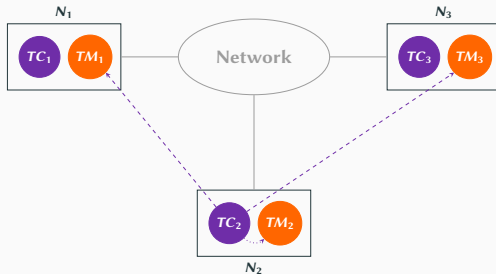
Global transactions access/update data at multiple (local) sites. Local subtransaction are executed at each site. Enforcing the ACID properties is much more complex.

Each site has a **local transaction manager** and a **transaction coordinator**.

Distributed Transactions

The **local transaction manager** ensures that the ACID properties hold for local transactions and maintains the local log files.

The **transaction coordinator** starts transactions that are initiated at one site, distributes the subtransactions to other sites, and ensures that a transaction either executes **at all sites or at none**.



Required because we must ensure **atomicity across all sites**, e.g., we are not allowed to commit a transaction at node N_1 but abort the same transaction at node N_2 .

List of Protocols:

- Two-Phase Commit (2PC; used in practice).
- Three-Phase Commit (3PC; solves issues of 2PC but impracticable).
- Persistent Messaging (PM).

Distributed Transactions – 2PC in a Nutshell

Transaction T is initiated at node N_i with transaction coordinator TC_i .

Phase 1: TC_i “asks” other participants to **prepare to commit** (and logs it beforehand). Each **transaction manager** determines if **it can commit**, logs it, and “reports” it to TC_i .

Phase 2: If TC_i receives a **single abort** message, then all participants are **informed to abort T** . Otherwise, all participants are **notified to commit T** . Prior to that, TC_i logs the decision locally. The involved transaction managers comply to the decision.

Different notions of consistency to specify desired properties in a DDBMS. Ideally, all updates appear immediately **at all sites** in the same order (illusion of a single data copy).

Strong consistency refers to this ideal scenario, but this is often expensive (or even impracticable). **Weak consistency** relaxes the consistency constraints to improve the performance or the availability of a DDBMS.

High Availability: A DDBMS with extremely low downtime (about 99.99% available).

In large systems, failures happen frequently (nodes may fail; the network may partition).

Trade (or relax) **consistency** to achieve **high availability**.

The BASE Properties

BASE is an **alternative consistency model** that is **weaker** than the ACID properties and favors **availability over consistency**.

Basically Available: The system appears to **work most of the time**, i.e., reads/writes should be allowed even if the network partitions, but without consistency guarantee.

Soft State: The **state** of the database **may not be precisely defined** all the time, i.e., replicas do not have to be mutually consistent (e.g., in case of writes).

Eventually Consistent: Once the network partitioning is resolved, the states of all replicas converge, i.e., all **replicas become consistent eventually** (i.e., over time).

Eventual Consistency

No new updates $\Rightarrow$ Writes are propagated to all replicas and all replicas converge towards a common state.

Inconsistent replicas must be identified because two replicas may be updated independently (e.g., version-vector scheme).

Inconsistent updates may need to be **merged** (e.g., in the worst case, human interaction is required – comparable to a merge conflict in git).

Brewer's CAP Theorem

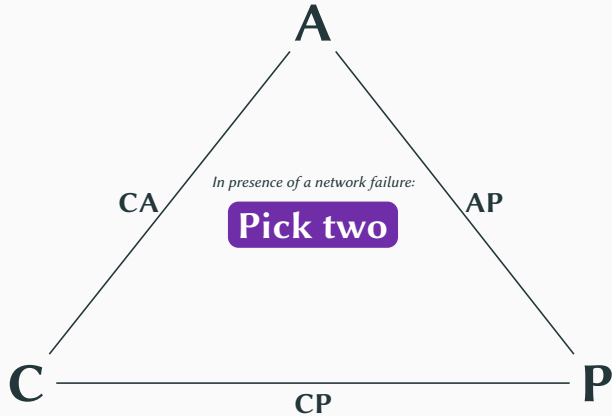
A distributed database system has **three main properties**:

- **Consistency**: All replicated copies are in the same state.
- **Availability**: System runs even in case of failures due to replication.
- **Partition-tolerance**: System runs even if the network is partitioned.

Network partition: Network decomposes into multiple parts/subsystems that cannot reach one another.

CAP Theorem: You can have **at most two** of the three **properties**. Precisely, in **case of an error**, one of the 3 properties **C**onsistency, **A**vailability, or **P**artition-tolerance is lost.

Brewer's CAP Theorem



Brewer's CAP Theorem

AP Systems: Sacrifice consistency in case of a network **P**artition, but stay **A**vailable. Inconsistencies must be resolved once the network partition is resolved.

CP Systems: Sacrifice availability in case of a network **P**artition, but maintain **C**onsistency.

CA Systems: “Ideal” **C**onsistency and **A**vailability. No availability and consistency guarantees in case of a network partition.

Reality: AP or CP

- CP and CA systems are similar since CA systems are unavailable if the network partitions.
- Network partitions are rare but unpredictable.
- **Question:** How does the DDBMS trade **C**onsistency and **A**vailability if network partitions?

Abadi's PACELC Theorem

PAC stands for CAP (rewritten) and covers the DDBMS properties in case of an error.
ELC stands for **E**lse (DDBMS is running without error): How does the DDBMS trade Latency and **C**onsistency?

PACELC ¹⁸¹⁹²⁰ **extends** CAP and considers the tradeoff between **L**atency and **C**onsistency that a DDBMS has to make when it **operates regularly**.

Even if a DDBMS operates without failure, we have a **tradeoff** between **L**atency and **C**onsistency. This is **due to data replication** that is a prerequisite for high availability.

¹⁸https://en.wikipedia.org/wiki/PACELC_theorem

¹⁹<http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>

²⁰<http://www.cs.umd.edu/~abadi/papers/abadi-pacelc.pdf>

With regard to database systems, a **workload** is a set of queries/updates that reflects a **typical usage pattern (load)**.

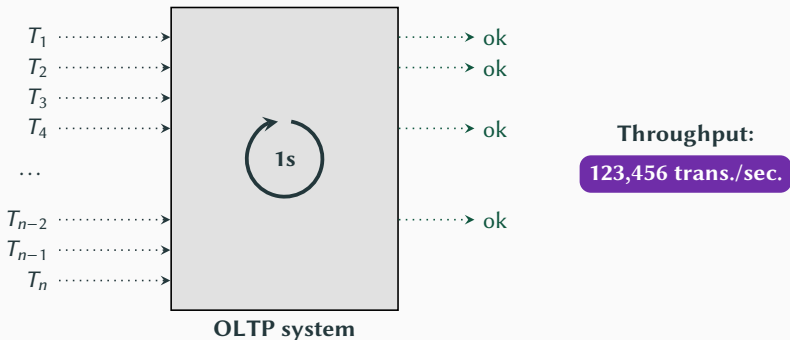
Different database systems perform better/worse on particular workloads.

Transaction Processing Performance Council (TPC): An independent consortium that releases standardized benchmarks for various workloads, the TPC benchmarks ²¹.

²¹<http://www.tpc.org/>

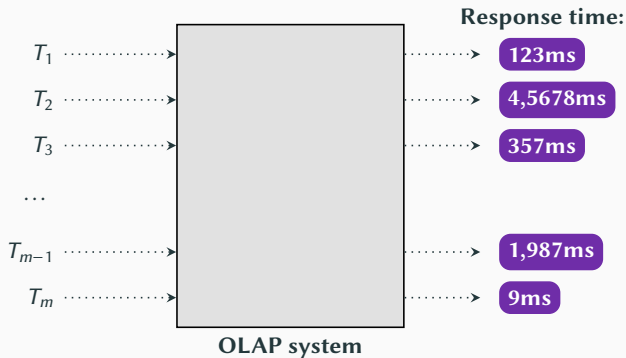
OLTP Workloads

Online Transaction Processing (OLTP): Short-lived read/write transactions with a small footprint (i.e., only a small portion of the data is touched). **Many transactions** must be processed **as fast as possible** (high throughput, low response time). The TPC-C benchmark provides typical OLTP workloads.



OLAP Workloads

Online Analytical Processing (OLAP): Long-running read-only queries that exploratory analyze a large portion of the overall data (to support decisions). This often involves complex join operations and the main focus is **low response time**. The TPC-DS benchmark provides typical OLAP workloads; “Data Warehouse”.



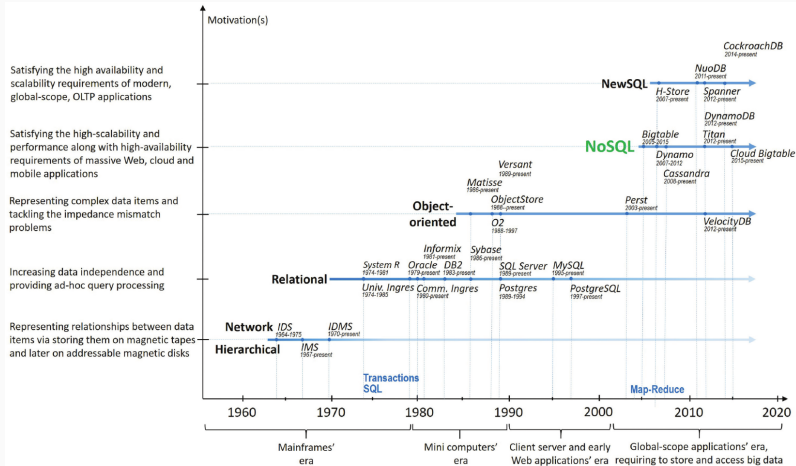
Workloads – Batches vs. Streams

Batch Workloads: A **batch** is a large but **bounded static dataset**. Before data can be processed, all data must be completely available (e.g., on hard disk).

Stream Workloads: A **stream** is an **unbounded evolving dataset**. Data items are processed as they stream into the system one after another, i.e., the data does not have to be completely available.

Systems Potpourri

Continuous Development



Picture taken from Davoudian et al. *A Survey on NoSQL Stores*. ACM Computing Surveys, 2018.

In-Memory Database Systems: Relies on **main memory (RAM)** for storing the **data** rather than hard disks (HDDs) or SSDs. Very **low response times (microseconds)** but must deal with durability (logs).

Database as a Service (DBaaS): A cloud-based platform (service) that provides computing infrastructure, data storage ,and **database functionality in the cloud**. The clients (you) do not have to set up their own database system on their own hardware, but just **access and use a database that runs in the cloud**.

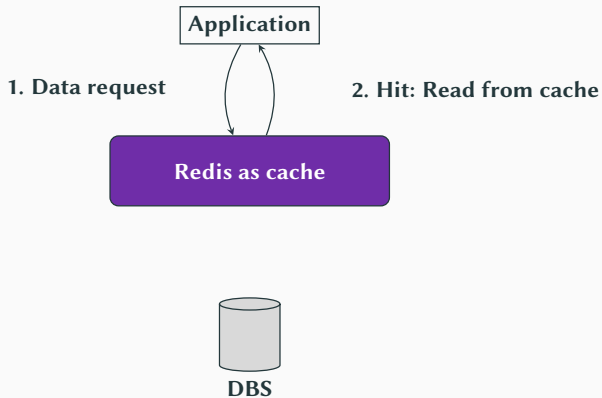
The **REmote DIctionary Server** ²² is an **in-memory** NoSQL database system based on the **key-value data model** that **supports durability** (i.e., data can be made persistent).

- Redis is a CP system.
- In-memory, i.e., very fast (avg. read/write performance: < 1ms).
- Collection of useful, high-performance data structures (lists, sets, bitmaps, ...).
- Replication and persistence support.
- Supports many programming languages incl. Java, Python, C/C++, and JavaScript.

²²<https://redis.io>

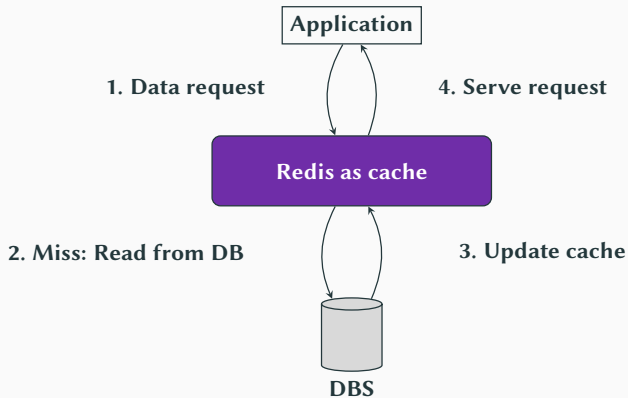
Key-Value Stores – Redis

Use Case: **Cache** between application and database system – **Cache Hit**.



Key-Value Stores – Redis

Use Case: Cache between application and database system – **Cache Miss.**



Key-Value Stores – Other Systems

Riak KV ²³ is a distributed **persistent** key-value store with support for **advanced data types** (e.g., JSON). It is inspired by Amazon's DynamoDB ²⁴ and provides **high availability** (i.e., is an AP system).

Memcached ²⁵ (“Mem-Cash-Dee”) is a distributed **in-memory** key-value store with a performance similar to Redis. However, Memcached supports only the **simple key-value data model** and has **no support for durability**. Not a database system!

²³<https://riak.com/products/riak-kv/>

²⁴<https://aws.amazon.com/dynamodb/>

²⁵<https://memcached.org/>

MongoDB ²⁶ is a distributed NoSQL database system based on the document-oriented data model with a focus on consistency, high read availability, and horizontal scalability.

- MongoDB is a CP system (single leader based system).
- Uses JSON as document format (specifically, a binary JSON format called BSON).
- Shared-nothing system architecture.
- JSON-based query language (limited support for joins as we know it).
- Replication and sharding (aka fragmentation) support through replica sets.

²⁶<https://www.mongodb.com/>

Apache CouchDB ²⁷ is a distributed document store with support for **master-master replication** ²⁸ and a conflict resolution protocol. It is an AP system that uses the plain JSON format.

Couchbase ²⁹ is a distributed document store with a built-in in-memory cache (memcached) and an SQL-like query language (N1QL). It uses CouchDB as back end.

²⁷<https://couchdb.apache.org/>

²⁸https://en.wikipedia.org/wiki/Multi-master_replication

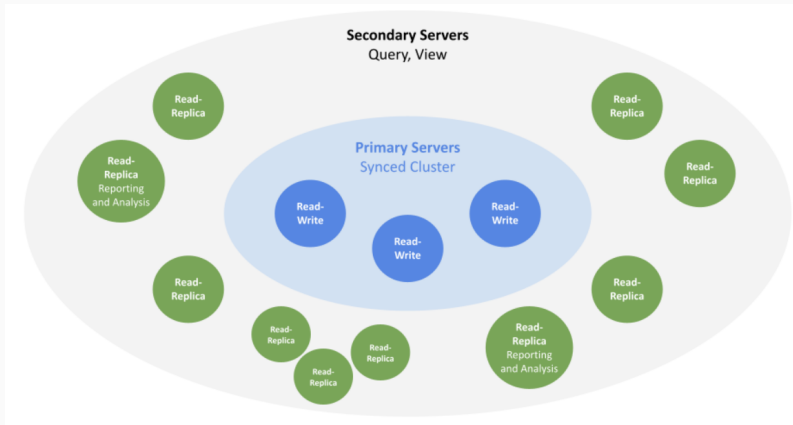
²⁹<https://www.couchbase.com/>

Neo4j³⁰ is a NoSQL database system based on the graph data model with ACID guarantees. Neo4j is a **native** graph store.

- Neo4j is a CP system (as of version 3.5).
- Shared-nothing system architecture.
- Set of core servers (serve write+read) and many read replicas.
- Emphasis on read availability (full replication).
- SQL-inspired query language (Cypher) to describe graph patterns (ASCII art).

³⁰<https://neo4j.com/>

Graph Stores – Neo4j



<https://neo4j.com/docs/operations-manual/current/clustering/introduction/>.

Tigergraph ³¹ is a distributed native graph store that supports parallel computation and advanced analytics. It has its own query language GSQL.

ArangoDB ³² is a multi-model database system that supports the graph-based model in JSON format.

OrientDB ³³ is another multi-model database system that supports the graph-based model. It uses an SQL-like syntax that is extended for graphs.

³¹<https://www.tigergraph.com/>

³²<https://www.arangodb.com/>

³³<https://orientdb.org/>

Apache Cassandra ³⁴ is a distributed NoSQL database system based on the extensible record (or wide column) data model. It is a decentralized and “master-less” DDBMS that supports eventual consistency.

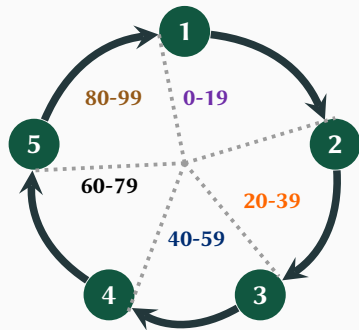
- Apache Cassandra is an AP system (but can be configured as CP system).
- Shared-nothing system architecture.
- Support for nested column families (so-called super column families).
- The Cassandra query language (CQL) resembles SQL.

³⁴<https://cassandra.apache.org/>

Extensible Record Store – Apache Cassandra

Master-less architecture $\Rightarrow$ **No single point of failure.**

Consistent Hashing (Token Ring)



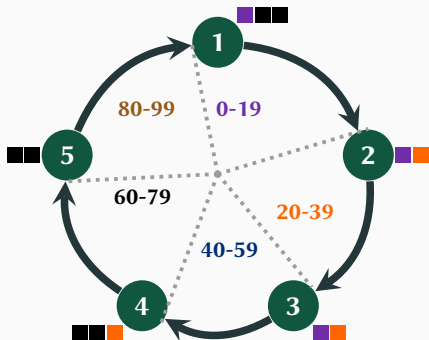
Hash function $h(.)$ determines node.

Data			$h(.)$	Node
Daniel	Kocher	Senior Sc.	18	1
Florian	Graf	Postdoc	21	2
Bianca	Löhnert	PhD Student	67	4
Christoph	Kirsch	Full Prof.	70	4

Extensible Record Store – Apache Cassandra

Master-less architecture $\Rightarrow$ **No single point of failure.**

Replication Factor (for example, 3)



Hash function $h(.)$ determines node.

Data			$h(.)$	Node
Daniel	Kocher	Senior Sc.	18	1
Florian	Graf	Postdoc	21	2
Bianca	Löhnert	PhD Student	67	4
Christoph	Kirsch	Full Prof.	70	4

Other Systems

- **RDBMS:** PostgreSQL ³⁵, SQLite ³⁶, Oracle ³⁷, Microsoft SQL Server ³⁸, ...
- **NewSQL Systems:** CockroachDB ³⁹, VoltDB ⁴⁰, ...
- **In-Memory OTLP & OLAP Systems:** HyPer ⁴¹, SAP Hana ⁴², ...
- **Cloud Services:** Amazon DynamoDB ⁴³, Snowflake Cloud Data Warehouse ⁴⁴, ...

³⁵<https://www.postgresql.org/>

³⁶<https://www.sqlite.org/>

³⁷<https://www.oracle.com/database/>

³⁸<https://www.microsoft.com/en-us/sql-server/sql-server-2019>

³⁹<https://www.cockroachlabs.com/>

⁴⁰<https://www.voltdb.com/>

⁴¹<https://hyper-db.de/>

⁴²<https://www.sap.com/products/hana.html>

⁴³<https://aws.amazon.com/dynamodb/>

⁴⁴<https://www.snowflake.com/>

Use the **database of databases** ⁴⁵ for a first impression and cross references.

A 37-page **survey on NoSQL database systems** ⁴⁶.

Books on NoSQL database systems ⁴⁷ ⁴⁸.

⁴⁵<https://dbdb.io/>

⁴⁶Davoudian et al. *A Survey on NoSQL Stores*. ACM Computing Surveys, 2018.

⁴⁷Sadalage and Fowler. *NoSQL Distilled – A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2013.

⁴⁸Redmond and Wilson. *7 Databases in 7 Weeks - A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf, 2012.