What is a database?

- → Storing large amounts of data in a structured way
- → Allows for dynamic queries for data

- → It used to be about storing:
  - → Corporate data
    - → Payrolls, inventory, sales, customers, accounting, documents, …
    - → Banking Systems
    - → Stock Exchanges
    - → Airline Systems
    - → Etc.

- → Today, databases are used in all fields:
  - → Web backends:
    - → Web search (google, bing, etc.)
    - → Social Networks (Facebook, Instagram, etc.)
    - → Blogs, Forums, etc.
  - → Mobile applications
  - → Data Warehouses
  - → Big Data
  - → AI
  - → Etc.

Why use a database?

→ Easy to use

→ Flexible search

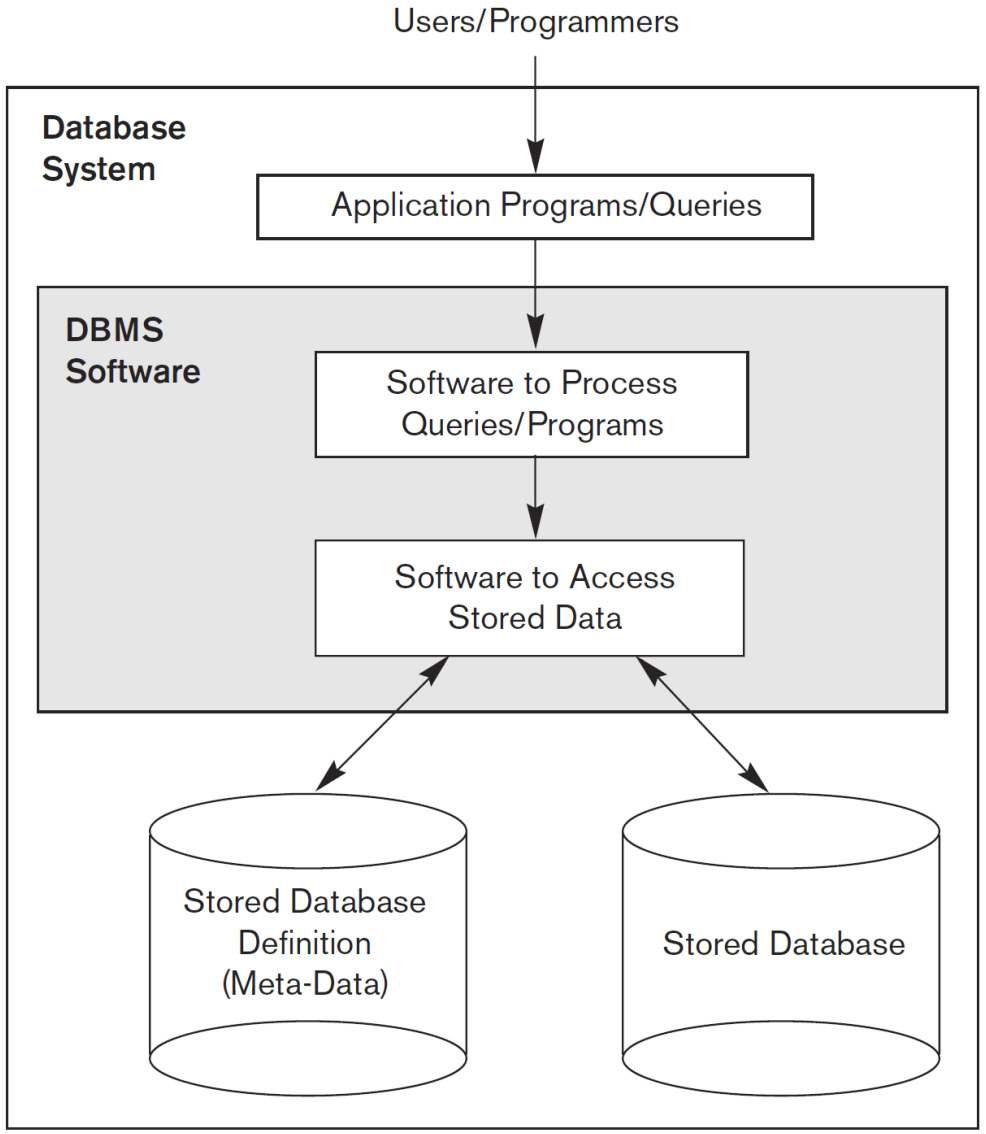
→ Efficient

→ Centralized storage

→ Multi-user access

→ Scalability

→ Security and consistency

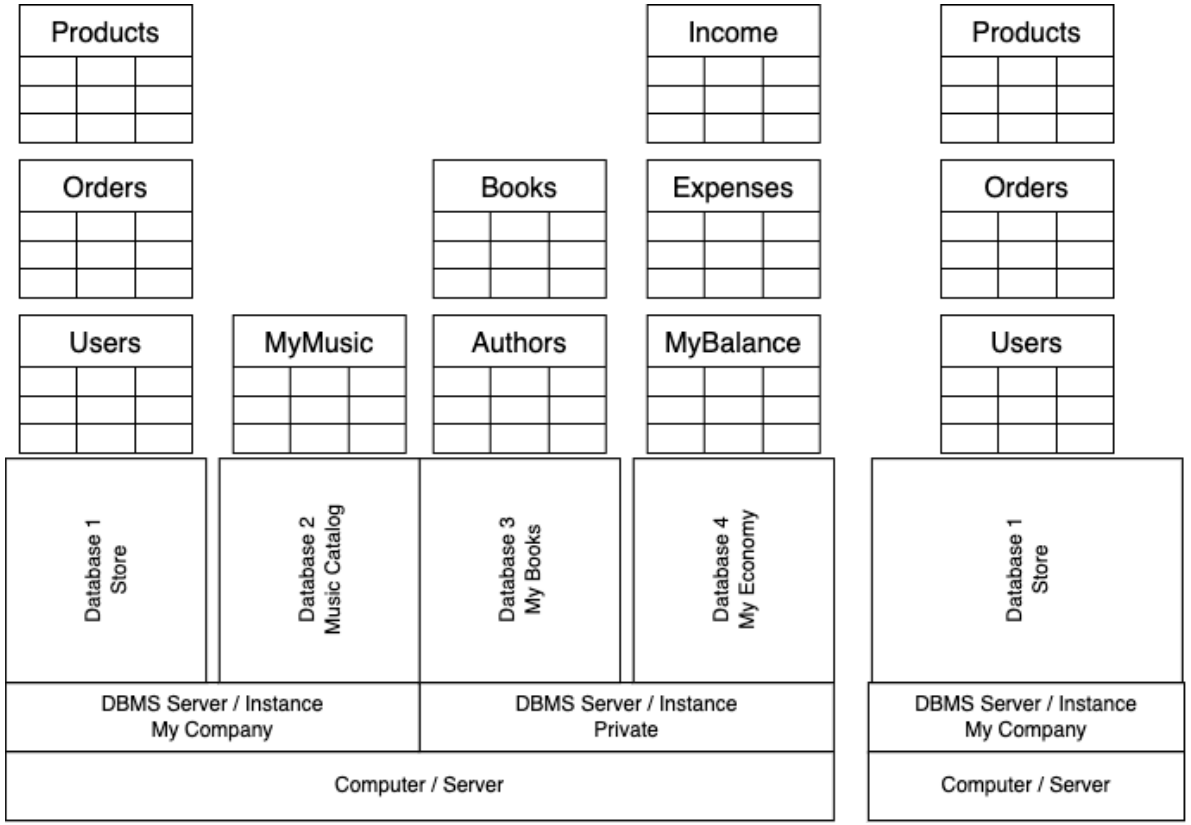


Definitions 1/4

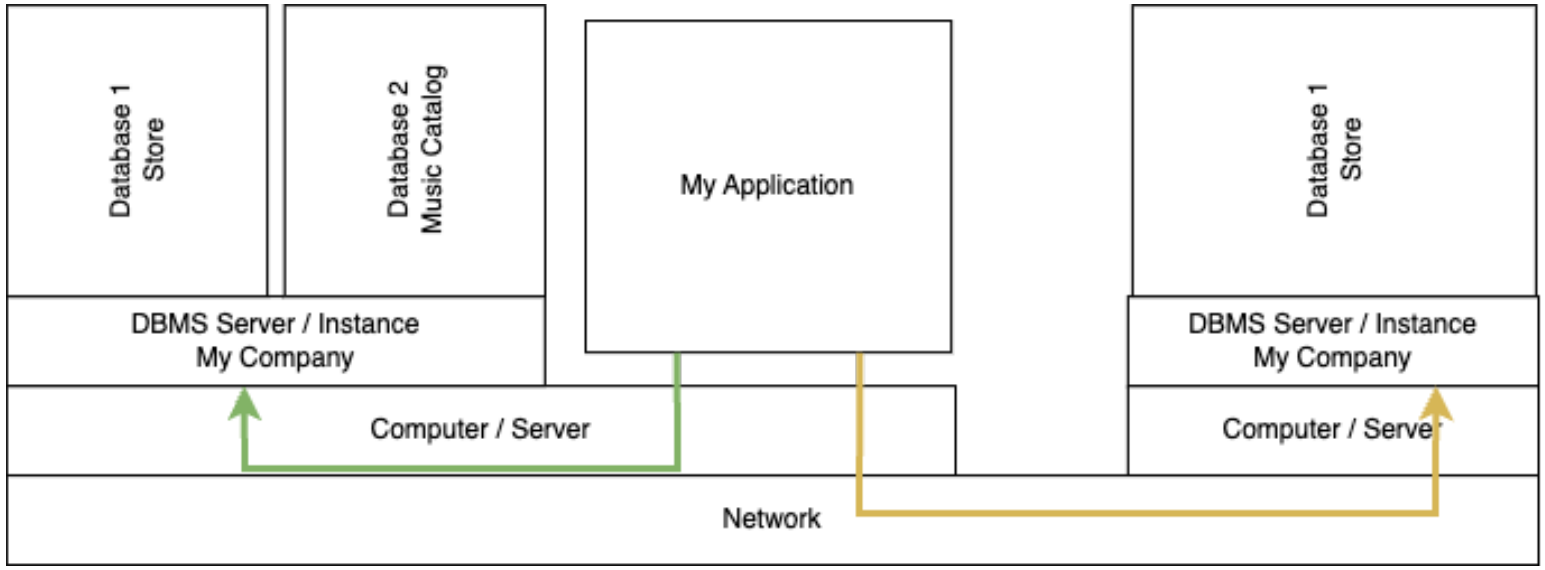
- → Database Management System (DBMS) – A system that enables users to create and maintain a database.
- → Database – A collection of related data. A collection of random data is not a database. Often the word database also, incorrectly, refers to a DBMS.
- → Data – Known facts that can be recorded that has implicit meaning.
- → Mini-World / Universe of Discourse – a database that represents some aspect of the real world. (Mostly here as the book refers to it)
- → Meta data – Data about data, or data that gives context to other data.
- → Schema – A definition of table structures and their relationships.
- → SQL – Structured Query Language – A language to extract data from a database.

Source: Fundamentals of DB Systems, by Elmasri and Navathe

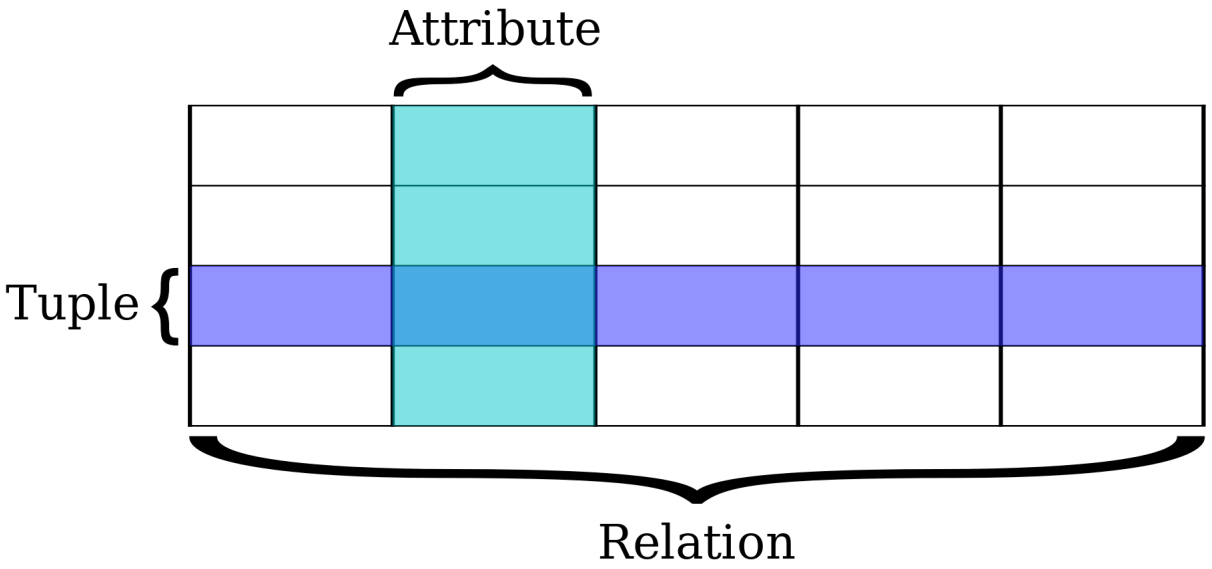
Definitions 2/4



Definitions 3/4



Definitions 3/4



- → Table – A collection of rows and columns that contains Cells.
- → Row, Tuple, or Record – A collection of cells that together form a set of data that has a concrete Relation to each other. Shown as the horizontal line to the left.
- → Column, Attribute, or Field – A collection of Cells that that are all the same type. They are a part of multiple Rows.
- → Cell – A specific Rows Column. Ergo the intersection.
- → Relation – The relationship between data in a Row.
- → Value – The information saved in the specific Cell.
- → View / Result Set – The table that is created in memory and sent to the client as a result of a query.

Relational Databases

- → Consists of multiple tables
- → Tables relate to each other
- → Uses primary and foreign keys to enforce the relationship constraints

Rank			DBMS	Database Model	Score		
Feb 2020	Jan 2020	Feb 2019			Feb 2020	Jan 2020	Feb 2019
1.	1.	1.	Oracle	Relational, Multi-model	1344.75	-1.93	+80.73
2.	2.	2.	MySQL	Relational, Multi-model	1267.65	-7.00	+100.36
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1093.75	-4.80	+53.69
4.	4.	4.	PostgreSQL	Relational, Multi-model	506.94	-0.25	+33.38
5.	5.	5.	IBM Db2	Relational, Multi-model	165.55	-3.15	-13.87
6.	6.	6.	Microsoft Access	Relational	128.06	-0.52	-15.96
7.	7.	7.	SQLite	Relational	123.36	+1.22	-2.81
8.	8.	8.	MariaDB	Relational, Multi-model	87.34	-0.11	+3.91
9.	9.	↑ 10.	Hive	Relational	83.53	-0.71	+11.25
10.	10.	↓ 9.	Teradata	Relational, Multi-model	76.81	-1.48	+0.84
11.	↑ 12.	↑ 12.	SAP HANA	Relational, Multi-model	54.97	+0.28	-1.58
12.	↓ 11.	↓ 11.	FileMaker	Relational	54.88	-0.23	-2.91
13.	13.	13.	SAP Adaptive Server	Relational	52.73	-1.86	-3.02
14.	14.	14.	Microsoft Azure SQL Database	Relational, Multi-model	31.41	+3.20	+4.28
15.	15.	↑ 20.	Google BigQuery	Relational	27.56	+0.81	+8.81

Data Model	Example Databases
Key-Value ("Key-Value Databases," p. 81)	BerkeleyDB
	LevelDB
	Memcached
	Project Voldemort
	Redis
	Riak
Document ("Document Databases," p. 89)	CouchDB
	MongoDB
	OrientDB
	RavenDB
	Terrastore
Column-Family ("Column-Family Stores," p. 99)	Amazon SimpleDB
	Cassandra
	HBase
	Hypertable
Graph ("Graph Databases," p. 111)	FlockDB
	HyperGraphDB
	Infinite Graph
	Neo4J
	OrientDB

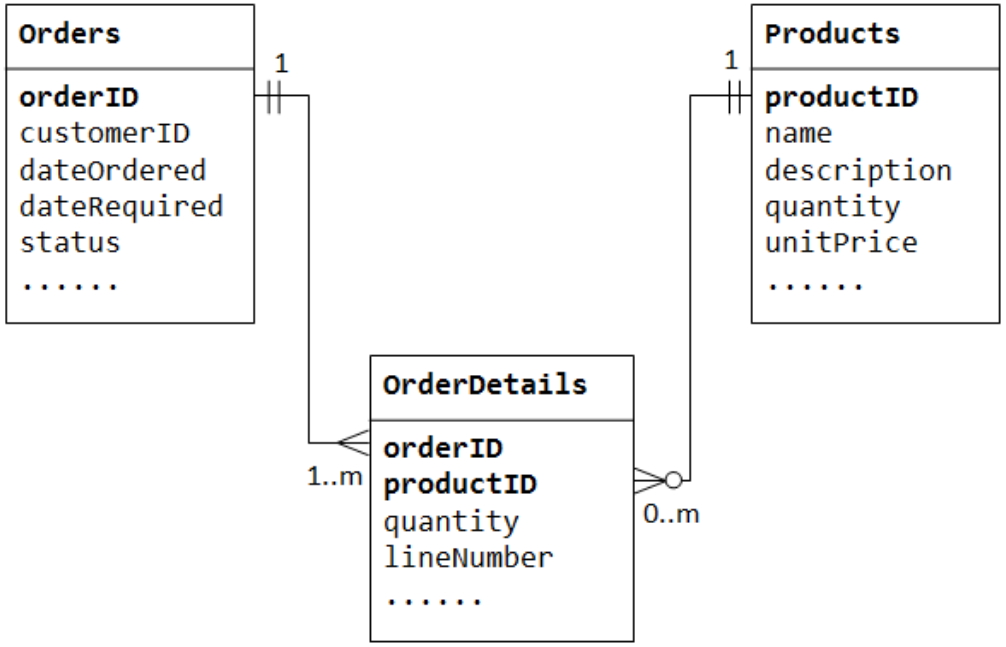
Source: NoSQL Distilled, by P. Sadalage and M. Fowler

NoSQL Databases

→ NoSQL means **N**ot **o**nly **SQL**

→ NoSQL covers multiple types of databases

What is a relational database?



- → Has a Schema that defines the Tables
- → Uses multiple tables to store data
- → Uses the notion of Primary Keys and Foreign keys to relate table content to each other.
- → Uses SQL which makes CRUD (Create, Read, Update, Delete) operations on Schemas, Tables, and Rows.

PostgreSQL

- → This course will use PostgreSQL (AKA Postgres)
- → However, this is **NOT** a PostgreSQL course!

- → PostgreSQL is:
  - → A Relational Database
  - → Cross-Platform (Windows, Mac OS, Linux, BSD, Solaris)
  - → Licensed under the PostgreSQL License, a liberal Open Source license, similar to the BSD or MIT licenses.
  - → Open Source (https://github.com/postgres/postgres)
- → Documentation and usage sites:
  - → https://www.postgresql.org/docs/12/index.html
  - → https://www.postgresqltutorial.com/

```sql
-- creating the initial accounts table
CREATE TABLE account(
   user_id serial PRIMARY KEY,
   username VARCHAR (50) UNIQUE NOT NULL,
   password VARCHAR (50) NOT NULL,
   email VARCHAR (355) UNIQUE NOT NULL,
   created_on TIMESTAMP NOT NULL,
   last_login TIMESTAMP
);

-- inserting two users
INSERT INTO account (username, password, email, created_on)
VALUES ('John', 'myPassW0rd', 'john@acme.com', NOW());

INSERT INTO account (username, password, email, created_on)
VALUES ('Anne', 'myPassW0rd', 'anne@acme.com', NOW());

-- querying for all rows in the account table
SELECT * FROM account;

-- updating the password of the user Anne
UPDATE account SET password = 'newPassW0rd' WHERE username = 'Anne';
```

What is SQL?

→ Acronym for: **S**tructured **Q**uery **L**anguage.

→ Pronounced either Sequel, or SQL.

→ Allows for retrieval of data from a database.

→ A query results in a result set, which is structured as a table with rows and columns.

Tables and relationships

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

Creating a Table

```
CREATE TABLE account(
    user_id serial PRIMARY KEY,
    username VARCHAR (50) UNIQUE NOT NULL,
    password VARCHAR (50) NOT NULL,
    email VARCHAR (355) UNIQUE NOT NULL,
    created_on TIMESTAMP NOT NULL,
    last_login TIMESTAMP
);
```

user_id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone

Inserts (CRUD)

```
INSERT INTO account (username, password, email, created_on)
VALUES ('John', 'myPassW0rd', 'john@acme.com', NOW());

INSERT INTO account (username, password, email, created_on)
VALUES ('Anne', 'myPassW0rd', 'anne@acme.com', NOW());
```

	user_id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone
1	4	John	myPassW0rd	john@acme.com	2020-02-06 11:43:44.158522	[null]
2	5	Anne	myPassW0rd	anne@acme.com	2020-02-06 11:43:44.158522	[null]

Selects (CRUD)

```sql
SELECT * FROM account;

SELECT username, created_on FROM account WHERE email = 'john@acme.com';

SELECT username, created_on FROM account WHERE email LIKE '%anne%';
```

	user_id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone
1	1	John	myPassW0rd	john@acme.com	2020-02-06 11:41:11.729203	[null]

Updates (CR**U**D)

```
UPDATE account SET password = 'newPassW0rd' WHERE username = 'Anne';
```

	user_id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone
1	1	John	myPassW0rd	john@acme.com	2020-02-06 11:41:11.729203	[null]
2	2	Anne	newPassW0rd	anne@acme.com	2020-02-06 11:42:13.27506	[null]

Delete (CRU**D**)

```
DELETE FROM account WHERE email = 'john@acme.com';
```

	user_id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone
1	5	Anne	myPassW0rd	anne@acme.com	2020-02-06 11:43:44.158522	[null]

BEWARE of doing this: `DELETE FROM account;`

Constraints and Data types in SQL

```sql
CREATE TABLE account(
   user_id serial PRIMARY KEY,
   username VARCHAR (50) UNIQUE NOT NULL,
   password VARCHAR (50) NOT NULL,
   email VARCHAR (355) UNIQUE NOT NULL,
   created_on TIMESTAMP NOT NULL,
   last_login TIMESTAMP
);
```

Constraints – Abbreviated, more later.

→ PRIMARY KEY – The unique identifier for the current row, which is always NOT NULL

→ FOREIGN KEY – A key that refers to a primary key in a different table, signifying their relationship to each other

→ UNIQUE – Two cells in this Column cannot be the same

→ NOT NULL – This attribute HAS to be specified

PostgreSQL Data Types 1/3

- → Null – Value Missing
- → Boolean – 1 bit
  - → Converts Boolean values e.g., 1, yes, y, t, true are converted to true, and 0, no, n false, f are converted to false.
- → Character types
  - → CHAR(n) – Fixed length text. Unused space is padded with space characters.
  - → VARCHAR(n) – Variable length string, as in “store up to”.
  - → TEXT – Large size text lengths such as book texts.

PostgreSQL Data Types 2/3

- → Numeric types
  - → Integer
    - → SMALLINT – 2 byte, ranges from -32.768 to 32.767 (2 byte = 2 x 8 bit = 256 x 256 = 65.536)
    - → INT – 4 byte integer, rage from -2,147,483,648 to 2,147,483,647
    - → SERIAL – Same as INT, but used for auto incrementing.
  - → Floating-point
    - → FLOAT(n) – floating point number with at the precision of n, and a maximum of 8 bytes. (n as in numbers after the “,”)
    - → REAL – A floating point number. 0.001, 0.00000001, etc.
- → Temporal types
  - → DATE – dates only
  - → TIME – time of day values
  - → TIMESTAMP – both date and time values
  - → INTERVAL – periods of time

PostgreSQL Data Types 3/3

→ UUID for storing Universally Unique Identifiers

→ Array for storing array strings, numbers, etc.

→ JSON stores JSON data

→ hstore stores key-value pair

→ Special types

  → box, line, point, lseg, polygon, inet, macaddr.

→ See more here: https://www.postgresql.org/docs/12/datatype.html

Contraint types

→ PRIMARY KEY – The unique identifier for the current row, which is always NOT NULL

→ REFERENCES (FOREIGN KEY) – A key that refers to a primary key in a different table, signifying their relationship to each other

→ UNIQUE – Two cells in this Column cannot be the same

→ NOT NULL – This attribute HAS to be specified

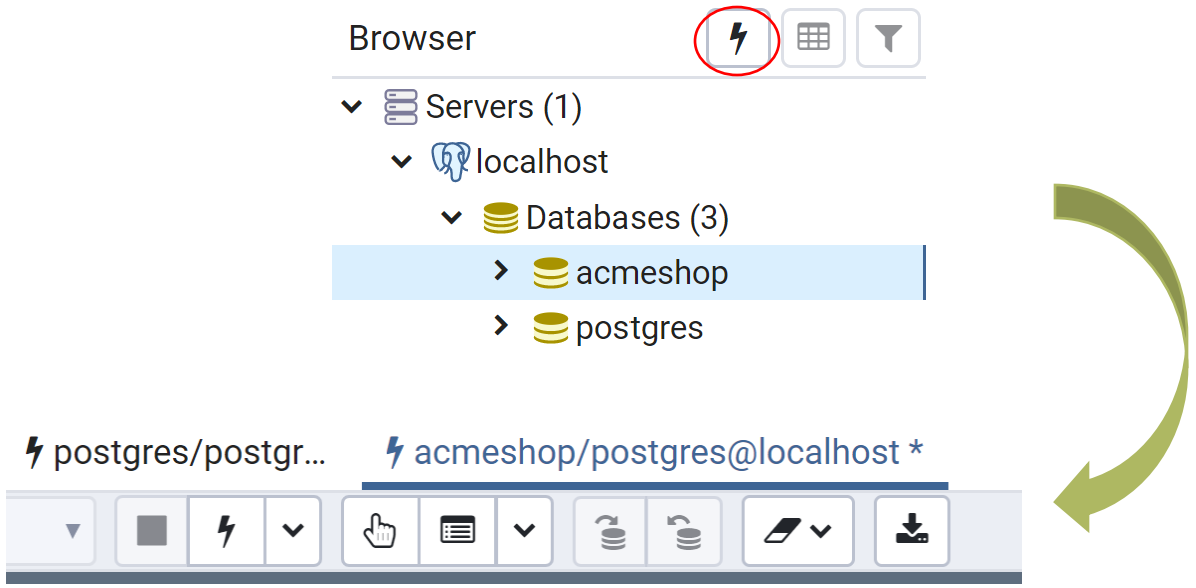
Constraints and Data types in SQL

```
CREATE TABLE account(
   user_id serial PRIMARY KEY,
   username VARCHAR (50) UNIQUE NOT NULL,
   password VARCHAR (50) NOT NULL,
   email VARCHAR (355) UNIQUE NOT NULL,
   created_on TIMESTAMP NOT NULL,
   last_login TIMESTAMP
);
```

Creating a Database

- → PostgreSQL only allows connections to one database at a time, and does not allow switching between them.
- → This is not normal, and usually you can “use database” to attach to another database.
- → Because of this restriction, run the command, pick the newly created database, and start a new SQL window.

```
-- creating a database
create database AcmeShop;
```



Deleting a Database

```
drop database AcmeShop;
```

Active connections to a database will block a deletion attempt!

Tables and relationships

Customers Table

	id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone
1	1	John	myPassW0rd	john@acme.com	2020-02-13 10:40:41.660572	[null]
2	2	Anne	SomePassword	anne@acme.com	2020-02-13 10:40:41.660572	[null]

Products Table

	id [PK] integer	name character varying (150)	price real
1	1	Samsung Galaxy S20	7799.95
2	2	Samsung Galaxy S20 - Leathe…	799.95
3	3	iPhone 11 Pro	8899
4	4	iPhone 11 Pro - Leather Cover	399.5
5	5	Huawai P30 Lite	1664.5
6	6	Huawai P30 - Leather Cover	1664.5

Orders Table

	id [PK] integer	order_number character (10)	customer_id integer
1	1	DA-0001234	1
2	2	DA-0001235	1
3	3	DE-0001236	2
4	4	DE-0001237	2

Order_Lines Table

	id [PK] integer	order_id integer	product_id integer	amount integer
1	1	1	1	2
2	2	1	2	2
3	3	1	5	1
4	4	3	3	2
5	5	3	4	1
6	6	4	1	1

Creating Tables with Relationships

```
CREATE TABLE account(
   id serial PRIMARY KEY,
   username VARCHAR (50) UNIQUE NOT NULL,
   password VARCHAR (50) NOT NULL,
   email VARCHAR (355) UNIQUE NOT NULL,
   created_on TIMESTAMP NOT NULL,
   last_login TIMESTAMP
);

CREATE TABLE blog_entries(
    id serial PRIMARY KEY,
    header varchar(255) NOT NULL,
    body TEXT NOT NULL,
    created_by integer NOT NULL REFERENCES account(id)
);
```

Constraint Enforcement

```sql
INSERT INTO account (username, password, email, created_on)
VALUES ('John', 'myPassW0rd', 'john@acme.com', NOW()); -- becomes user 1
```

```sql
INSERT INTO blog_entries (header, body, created_by)
VALUES ('My article', 'my body text', 1); -- works!
```

```sql
INSERT INTO blog_entries (header, body, created_by)
VALUES ('My article', 'my body text', 6); -- ERROR!
```

```
ERROR:  insert or update on table "blog_entries" violates foreign key constraint "blog_entries_created_by_fkey"
DETAIL:  Key (created_by)=(6) is not present in table "account".
SQL state: 23503
```

Deleting tables, and tables with constraints

```
CREATE TABLE account(
   id serial PRIMARY KEY,
   username VARCHAR (50) UNIQUE NOT NULL,
   password VARCHAR (50) NOT NULL,
   email VARCHAR (355) UNIQUE NOT NULL,
   created_on TIMESTAMP NOT NULL,
   last_login TIMESTAMP
);

CREATE TABLE blog_entries(
    id serial PRIMARY KEY,
    header varchar(255) NOT NULL,
    body TEXT NOT NULL,
    created_by integer NOT NULL REFERENCES account(id)
);

drop table account;
```

→ Delete in order, where non of a tables attributes has a constraint from another table

```
ERROR:  cannot drop table account because other objects depend on it
DETAIL:  constraint blog_entries_created_by_fkey on table blog_entries depends on table account
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
SQL state: 2BP01
```

Altering tables and deleting them

```
-- creating and altering tables
CREATE TABLE my_table(
   id serial PRIMARY KEY,
   my_attribute VARCHAR (50) UNIQUE NOT NULL
);

ALTER TABLE my_table ADD COLUMN my_new_coulmn TIMESTAMP;

ALTER TABLE my_table ALTER COLUMN my_new_coulmn TYPE varchar(50);

ALTER TABLE my_table DROP COLUMN my_new_coulmn;

DROP TABLE my_table;
```

Inserts (CRUD)

```
INSERT INTO account (username, password, email, created_on)
VALUES ('John', 'myPassW0rd', 'john@acme.com', NOW());

INSERT INTO account (username, password, email, created_on)
VALUES ('Anne', 'myPassW0rd', 'anne@acme.com', NOW());
```

	user_id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone
1	4	John	myPassW0rd	john@acme.com	2020-02-06 11:43:44.158522	[null]
2	5	Anne	myPassW0rd	anne@acme.com	2020-02-06 11:43:44.158522	[null]

Selects (CRUD)

```sql
SELECT * FROM account;

SELECT username, created_on FROM account WHERE email = 'john@acme.com';

SELECT username, created_on FROM account WHERE email LIKE '%anne%';
```

	user_id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone
1	1	John	myPassW0rd	john@acme.com	2020-02-06 11:41:11.729203	[null]

Updates (CRUD)

```
UPDATE account SET password = 'newPassW0rd' WHERE username = 'Anne';
```

	user_id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone
1	1	John	myPassW0rd	john@acme.com	2020-02-06 11:41:11.729203	[null]
2	2	Anne	newPassW0rd	anne@acme.com	2020-02-06 11:42:13.27506	[null]

Delete (CRU**D**)

```
DELETE FROM account WHERE email = 'john@acme.com';
```

	user_id [PK] integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp without time zone
1	5	Anne	myPassW0rd	anne@acme.com	2020-02-06 11:43:44.158522	[null]

BEWARE of doing this: `DELETE FROM account;`

Querying Your Result Set - Nested Queries

```
CREATE TABLE account(
   id serial PRIMARY KEY,
   username VARCHAR (50) UNIQUE NOT NULL,
   password VARCHAR (50) NOT NULL,
   email VARCHAR (355) UNIQUE NOT NULL,
   created_on TIMESTAMP NOT NULL,
   last_login TIMESTAMP
);

CREATE TABLE blog_entries(
    id serial PRIMARY KEY,
    header varchar(255) NOT NULL,
    body TEXT NOT NULL,
    created_by integer NOT NULL REFERENCES account(id)
);
```

```
select * from account, blog_entries
where blog_entries.created_by = account.id;
```

	id integer	username character varying (50)	password character varying (50)	email character varying (355)	created_on timestamp without time zone	last_login timestamp witho
1	1	John	myPassW0rd	john@acme.com	2020-02-13 11:55:37.125376	[null]
2	1	John	myPassW0rd	john@acme.com	2020-02-13 11:55:37.125376	[null]
3	1	John	myPassW0rd	john@acme.com	2020-02-13 11:55:37.125376	[null]
4	2	Anne	SomePassword	anne@acme.com	2020-02-13 11:55:42.35475	[null]
5	1	John	myPassW0rd	john@acme.com	2020-02-13 11:55:37.125376	[null]

```
select username, email, created_by from (
    select * from account, blog_entries
    where blog_entries.created_by = account.id
) as result_set where created_by = 2;
```

	username character varying (50)	email character varying (355)	created_by integer
1	Anne	anne@acme.com	2

Join types

- → INNER JOIN
  - → For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.
- → LEFT OUTER JOIN
  - → First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Thus, the joined table always has at least one row for each row in T1.
- → RIGHT OUTER JOIN
  - → First, an inner join is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is added with null values in columns of T1. This is the converse of a left join: the result table will always have a row for each row in T2.
- → FULL OUTER JOIN
  - → First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Also, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is added.
- → Source: https://www.postgresql.org/docs/9.2/queries-table-expressions.html

Inner Join

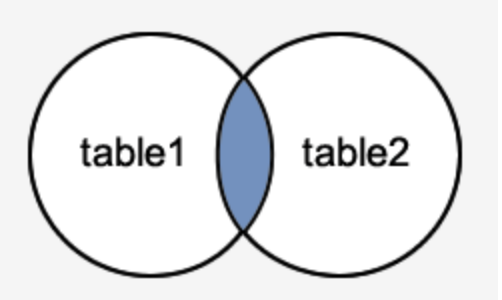
```
SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
```

```
SELECT * FROM t1,t2 WHERE t1.num = t2.num;
```

num integer	name character varying (10)	num integer	value character varying (10)
1	a	1	xxx
3	c	3	yyy

```
SELECT * FROM t1 INNER JOIN t2 USING (num);
```

num integer	name character varying (10)	value character varying (10)
1	a	xxx
3	c	yyy



T1 table

num integer	name character varying (10)
1	a
2	b
3	c

T2 table

num integer	value character varying (10)
1	xxx
3	yyy
5	zzz

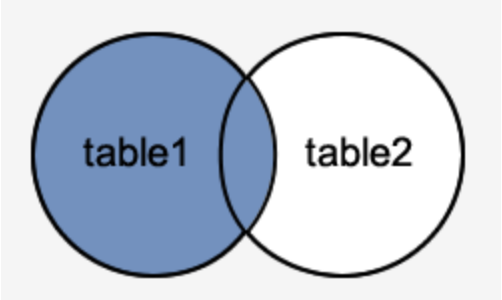
Left Outer Join

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
```

num integer	name character varying (10)	num integer	value character varying (10)
1	a	1	xxx
2	b	[null]	[null]
3	c	3	yyy

```
SELECT * FROM t1 LEFT JOIN t2 USING (num);
```

num integer	name character varying (10)	value character varying (10)
1	a	xxx
2	b	[null]
3	c	yyy



T1 table

num integer	name character varying (10)
1	a
2	b
3	c

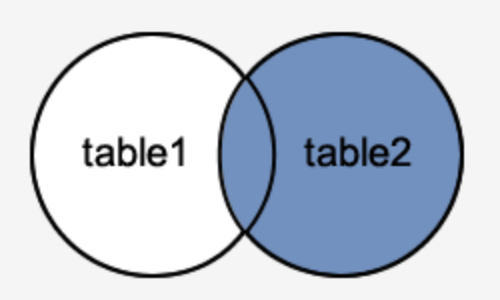
T2 table

num integer	value character varying (10)
1	xxx
3	yyy
5	zzz

Right Outer Join

```
SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
```

num integer	name character varying (10)	num integer	value character varying (10)
1	a	1	xxx
3	c	3	yyy
[null]	[null]	5	zzz



T1 table

num integer	name character varying (10)
1	a
2	b
3	c

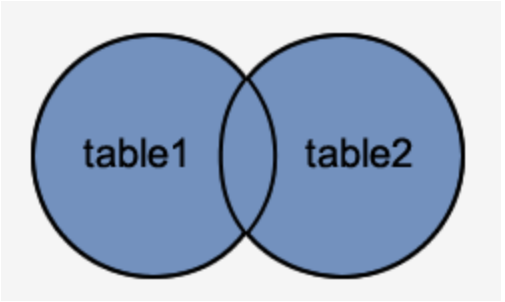
T2 table

num integer	value character varying (10)
1	xxx
3	yyy
5	zzz

Full Outer Join

```
SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
```

num integer	name character varying (10)	num integer	value character varying (10)
1	a	1	xxx
2	b	[null]	[null]
3	c	3	yyy
[null]	[null]	5	zzz



T1 table

num integer	name character varying (10)
1	a
2	b
3	c

T2 table

num integer	value character varying (10)
1	xxx
3	yyy
5	zzz

Views – Creating Virtual Tables

```
CREATE VIEW MyCustomView AS
    SELECT email, username FROM account, blog_entries
    WHERE blog_entries.created_by = account.id;

SELECT * FROM MyCustomView;
```

Query defining the view content

	email character varying (355)	username character varying (50)
1	john@acme.com	John
2	john@acme.com	John
3	john@acme.com	John
4	anne@acme.com	Anne
5	john@acme.com	John

- Can join and simplify multiple tables into a view.
- Result of the query in the view is dynamically updated, when new database content id added.
- Can hide complexity of data
- Takes very little space as only the query is stored

Normalization: What and why?

- → A systematic approach to:
  - → Restructure tables to eliminate data redundancy (duplicated data)
  - → Eliminate Update Anomalies
    - → Insertion
    - → Deletion
    - → Modification
- → Bottom up approach
  - → Starts from the data
  - → Ends with multiple related tables, with a minimum of duplicated data

Update Anomaly

Employees' Skills

Employee ID	Employee Address	Skill
426	87 Sycamore Grove	Typing
426	87 Sycamore Grove	Shorthand
519	94 Chestnut Street	Public Speaking
519	96 Walnut Avenue	Carpentry

→ Employee 519 is shown as having different addresses on different records.

Insertion Anomaly

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201
424	Dr. Newsome	29-Mar-2007	?

→ Until the new faculty member, Dr. Newsome, is assigned to teach at least one course, his or her details cannot be recorded.

Deletion Anomaly

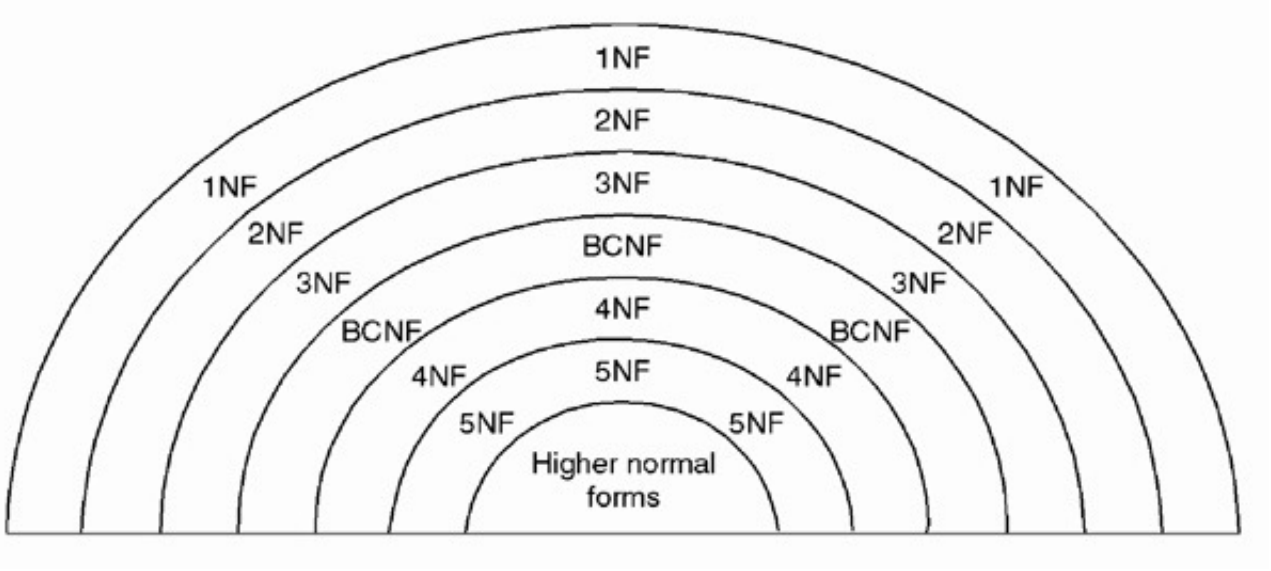
Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201

DELETE

→ All information about Dr. Giddens is lost if he or she temporarily ceases to be assigned to any courses.

Normal Forms



→ UNF: Unnormalized Form

→ NF1 – First Normal Form

→ NF2 – Second Normal Form

→ NF3 – Third Normal Form

→ BCNF – Boyce-Codd Normal Form

→ *NF4, NF5, and higher …*

→ Practical use of NF's go to BCNF, or at most NF4

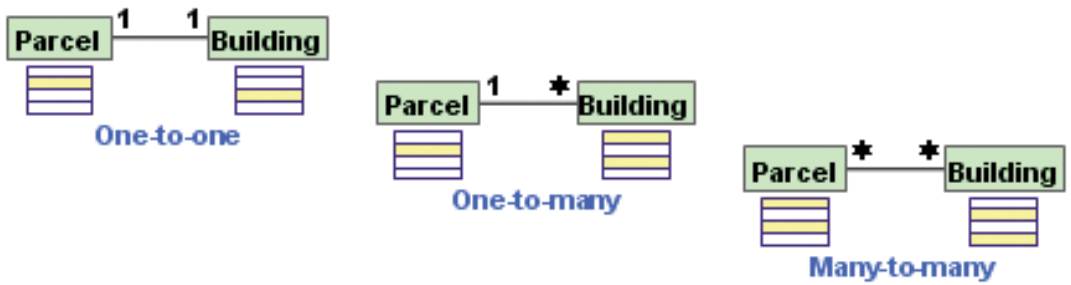
Guidelines

→ Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation

→ Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values

→ Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

Cardinalities



→ 1 - * , one to many

→ * - 1, many to one

→ * - *, many to many

→ 1 - 1, one to one

→ 0..1 - 1, zero/one to one

UNF: Unnormalized Form

→ Data idea on loan from YouTube

Customer Name	Item	Shipping Address	Newsletter	Supplier	Supplier Phone	Price
Jens Bergholm	XBOX One	Mølletoften 20, 7100 Vejle	Xbox News	Microsoft	12341324	1800
Dennis Jørgensen	Playstation 4	Vinkelvej 167, 7100 Vejle	Playstation News	Sony	23452345	1900
Hans Jensen	XBOX One, PS Vita	Kongensgade 45, 5000 Odense	Xbox News, Playstation News	Wholesale	00000000	3300
Anne Johansen	Playstation 4	Vinkelvej 167, 7100 Vejle	Playstation News	Sony	23452345	1900

NF1 – First Normal Form

→ Each column should contain atomic values – entries like “x, y” violate this rule
→ A column should contain values that are of the same type
→ Each row must be unique

Customer Name	Item	Shipping Address	Newsletter	Supplier	Supplier Phone	Price
Jens Bergholm	XBOX One	Mølletoften 20, 7100 Vejle	Xbox News	Microsoft	12341324	1800
Dennis Jørgensen	Playstation 4	Vinkelvej 167, 7100 Vejle	Playstation News	Sony	23452345	1900
Hans Jensen	XBOX One, PS Vita	Kongensgade 45, 5000 Odense	Xbox News, Playstation News	Wholesale	00000000	3300
Anne Johansen	Playstation 4	Vinkelvej 167, 7100 Vejle	Playstation News	Sony	23452345	1900

ID	Customer Name	Item	Shipping Address	Zip	City	Newsletter	Supplier	Supplier Phone	Price
1	Jens Bergholm	XBOX One	Mølletoften 20	7100	Vejle	Xbox News	Microsoft	12341324	1800
2	Dennis Jørgensen	Playstation 4	Vinkelvej 167	7100	Vejle	Playstation News	Sony	23452345	1900
3	Hans Jensen	XBOX One	Kongensgade 45	5000	Odense	Xbox News	Microsoft	12341324	1800
3	Hans Jensen	PS Vita	Kongensgade 45	5000	Odense	Playstation News	Sony	23452345	1500
4	Anne Johansen	Playstation 4	Vinkelvej 167	7100	Vejle	Playstation News	Sony	23452345	1900

Functional Dependencies

→ A strong connection between two attributes in a table

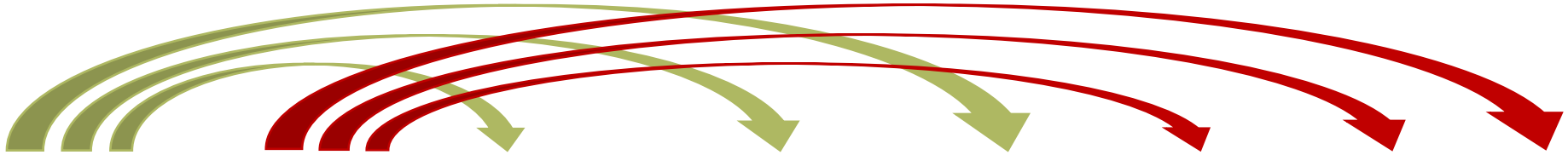
→ Denoted as A → B

→ A functionally determines B, or B is functionally dependent on A.

→ Customer Name and Item is the **Determinant**.

→ Customer Name → {Shipping Address, City, Newsletter}

→ Item → {Supplier, Supplier Phone, Price}



ID	Customer Name	Item	Shipping Address	Zip	City	Newsletter	Supplier	Supplier Phone	Price
1	Jens Bergholm	XBOX One	Mølletoften 20	7100	Vejle	Xbox News	Microsoft	12341324	1800
2	Dennis Jørgensen	Playstation 4	Vinkelvej 167	7100	Vejle	Playstation News	Sony	23452345	1900
3	Hans Jensen	XBOX One	Kongensgade 45	5000	Odense	Xbox News	Microsoft	12341324	1800
3	Hans Jensen	PS Vita	Kongensgade 45	5000	Odense	Playstation News	Sony	23452345	1500
4	Anne Johansen	Playstation 4	Vinkelvej 167	7100	Vejle	Playstation News	Sony	23452345	1900

Partial Dependencies

→ An Xbox one purchase does not require you to be “Jens Bergholm”

→ But there is a dependency, as “Jens Bergholm” purchased this item

ID	Customer Name	Item	Shipping Address	Zip	City	Newsletter	Supplier	Supplier Phone	Price
1	Jens Bergholm	XBOX One	Mølletoften 20	7100	Vejle	Xbox News	Microsoft	12341324	1800
2	Dennis Jørgensen	Playstation 4	Vinkelvej 167	7100	Vejle	Playstation News	Sony	23452345	1900
3	Hans Jensen	XBOX One	Kongensgade 45	5000	Odense	Xbox News	Microsoft	12341324	1800
3	Hans Jensen	PS Vita	Kongensgade 45	5000	Odense	Playstation News	Sony	23452345	1500
4	Anne Johansen	Playstation 4	Vinkelvej 167	7100	Vejle	Playstation News	Sony	23452345	1900

NF2 – Second Normal Form

→ No partial Dependencies

Customer Table

ID	Customer Name	Shipping Address	Zip	City	Newsletter
1	Jens Bergholm	Mølletoften 20	7100	Vejle	Xbox News
2	Dennis Jørgensen	Vinkelvej 167	7100	Vejle	Playstation News
3	Hans Jensen	Kongensgade 45	5000	Odense	Xbox News
3	Hans Jensen	Kongensgade 45	5000	Odense	Playstation News
4	Anne Johansen	Vinkelvej 167	7100	Vejle	Playstation News

Order Table

Customer_Id	Product_Id
1	1
2	2
3	1
3	3
4	2

Product Table

ID	Item	Supplier	Supplier Phone	Price
1	XBOX One	Microsoft	12341324	1800
2	Playstation 4	Sony	23452345	1900
3	PS Vita	Sony	23452345	1500

NF3 – Third Normal Form

→ A table is said to be in 3NF, if and only if:

→ That table is in the second normal form (2NF)

→ Every attribute in the table that do not belong to a candidate key should directly depend on every candidate key of that table.

Customer Table

ID	Customer Name	Shipping Address	Newsletter	Zip	City
1	Jens Bergholm	Mølletoften 20	Xbox News	7100	Vejle
2	Dennis Jørgensen	Vinkelvej 167	Playstation News	7100	Vejle
3	Hans Jensen	Kongensgade 45	Xbox News	5000	Odense
3	Hans Jensen	Kongensgade 45	Playstation News	5000	Odense
4	Anne Johansen	Vinkelvej 167	Playstation News	7100	Vejle

Order Table

Customer_Id	Product_Id
1	1
2	2
3	1
3	3
4	2

Product Table

ID	Item	Price	Supplier_ID
1	XBOX One	1800	1
2	Playstation 4	1900	2
3	PS Vita	1500	2

Supplier Table

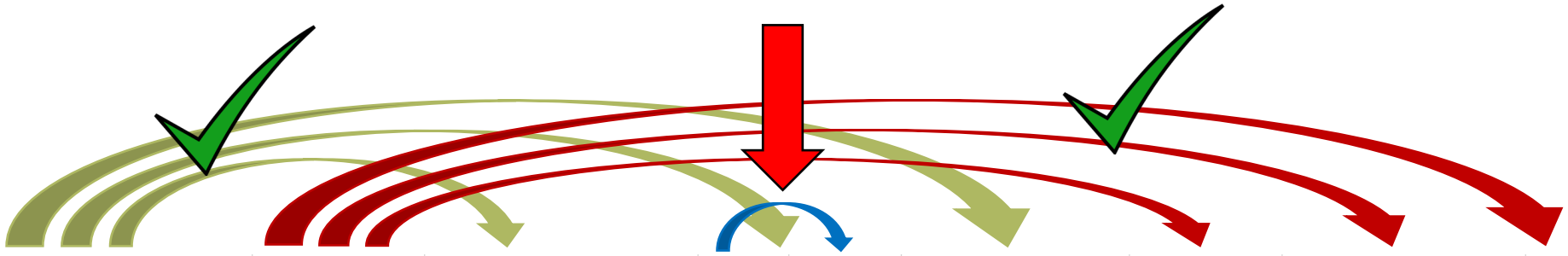
ID	Name	Phone
1	Microsoft	12341234
2	Sony	23452345

BCNF – Boyce–Codd Normal Form

→ AKA: NF3.5

→ If a relational schema is in BCNF then all redundancy based on functional dependency has been removed, although other types of redundancy may still exist.

→ Or said in another way: Even when a database is in 3rd Normal Form, still there would be anomalies resulted if it has more than one Candidate Key.



ID	Customer Name	Item	Shipping Address	Zip	City	Newsletter	Supplier	Supplier Phone	Price
1	Jens Bergholm	XBOX One	Mølletoften 20	7100	Vejle	Xbox News	Microsoft	12341324	1800
2	Dennis Jørgensen	Playstation 4	Vinkelvej 167	7100	Vejle	Playstation News	Sony	23452345	1900
3	Hans Jensen	XBOX One	Kongensgade 45	5000	Odense	Xbox News	Microsoft	12341324	1800
3	Hans Jensen	PS Vita	Kongensgade 45	5000	Odense	Playstation News	Sony	23452345	1500
4	Anne Johansen	Playstation 4	Vinkelvej 167	7100	Vejle	Playstation News	Sony	23452345	1900

BCNF – Boyce–Codd Normal Form

→ If a relational schema is in BCNF then all redundancy based on functional dependency has been removed, although other types of redundancy may still exist.

Customer Table

ID	Customer Name	Shipping Address	Newsletter	Zip_ID
1	Jens Bergholm	Mølletoften 20	Xbox News	7100
2	Dennis Jørgensen	Vinkelvej 167	Playstation News	7100
3	Hans Jensen	Kongensgade 45	Xbox News	5000
3	Hans Jensen	Kongensgade 45	Playstation News	5000
4	Anne Johansen	Vinkelvej 167	Playstation News	7100

Zip Table

Zip	Name
5000	Odense
7100	Vejle

Order Table

Customer_Id	Product_Id
1	1
2	2
3	1
3	3
4	2

Product Table

ID	Item	Price	Supplier_ID
1	XBOX One	1800	1
2	Playstation 4	1900	2
3	PS Vita	1500	2

Supplier Table

ID	Name	Phone
1	Microsoft	12341234
2	Sony	23452345

NF4 – Fourth Normal Form

→ All columns can be determined only by the key in the table and no other column

→ No Multi-valued Dependencies

→ Or said in another way: If no database table instance contains two or more, independent and multivalued data describing the relevant entity, then it is in 4th Normal Form.

Newsletters Tab.

ID	Newsletter
1	Xbox News
2	Playstation News

Subscriptions Table

Newsletter_ID	Customer_ID
1	1
2	2
1	3
2	3
2	4

Customer Table

ID	Customer Name	Shipping Address	Zip_ID
1	Jens Bergholm	Mølletoften 20	7100
2	Dennis Jørgensen	Vinkelvej 167	7100
3	Hans Jensen	Kongensgade 45	5000
4	Anne Johansen	Vinkelvej 167	7100

Zip Table

Zip	Name
5000	Odense
7100	Vejle

Order Table

Customer_Id	Product_Id
1	1
2	2
3	1
3	3
4	2

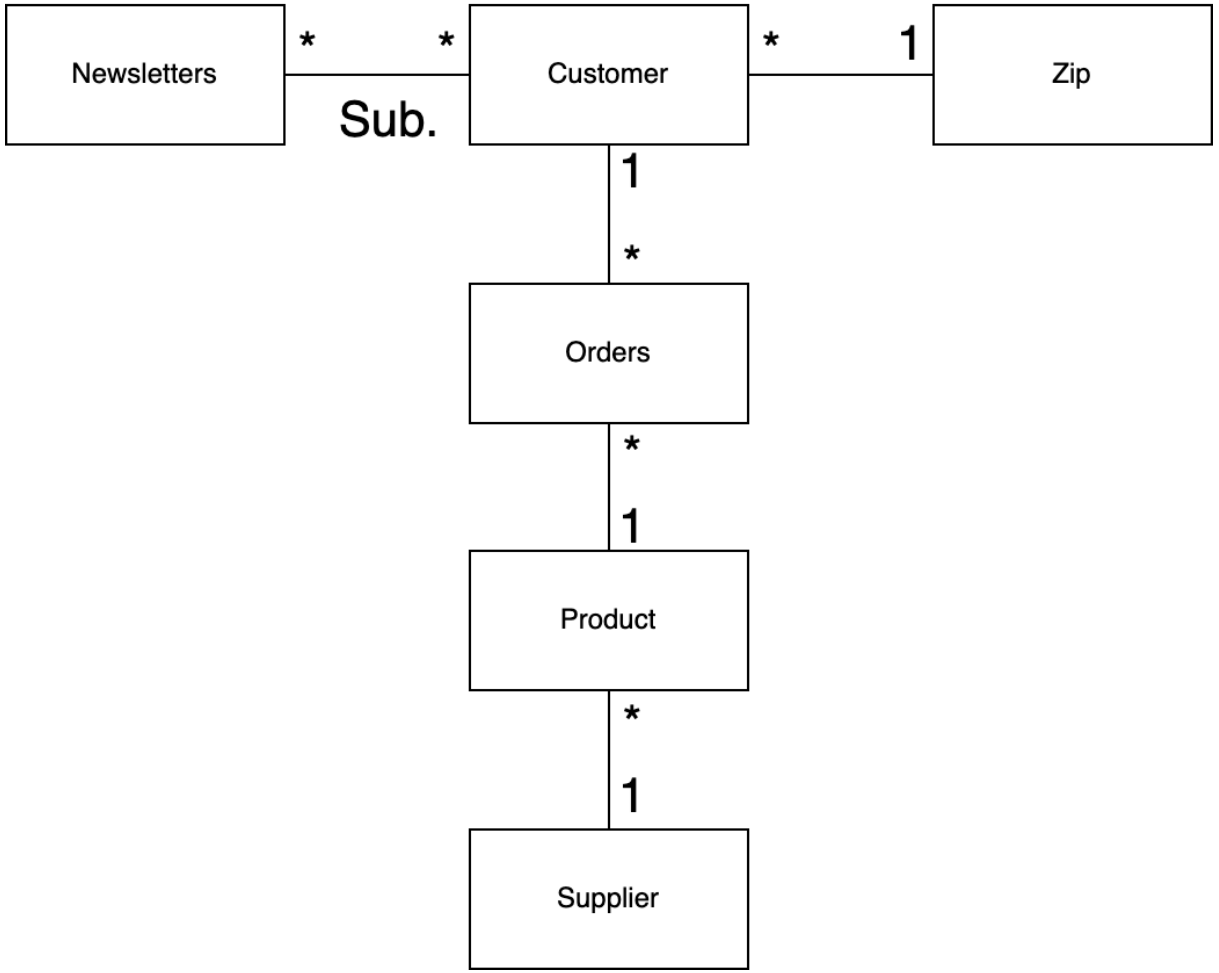
Product Table

ID	Item	Price	Supplier_ID
1	XBOX One	1800	1
2	Playstation 4	1900	2
3	PS Vita	1500	2

Supplier Table

ID	Name	Phone
1	Microsoft	12341234
2	Sony	23452345

Cardinalities



Recap with using simple rules

→ Rows identify single entities

→ A tables attributes should be directly related

  → Think object oriented, or about physical entities

→ Avoid Data Redundancy

→ Preserve Relationships

Working from NF1

Entity

Entity + Redundancy

Entity + Redundancy

Entity + Redundancy

Entity + Redundancy

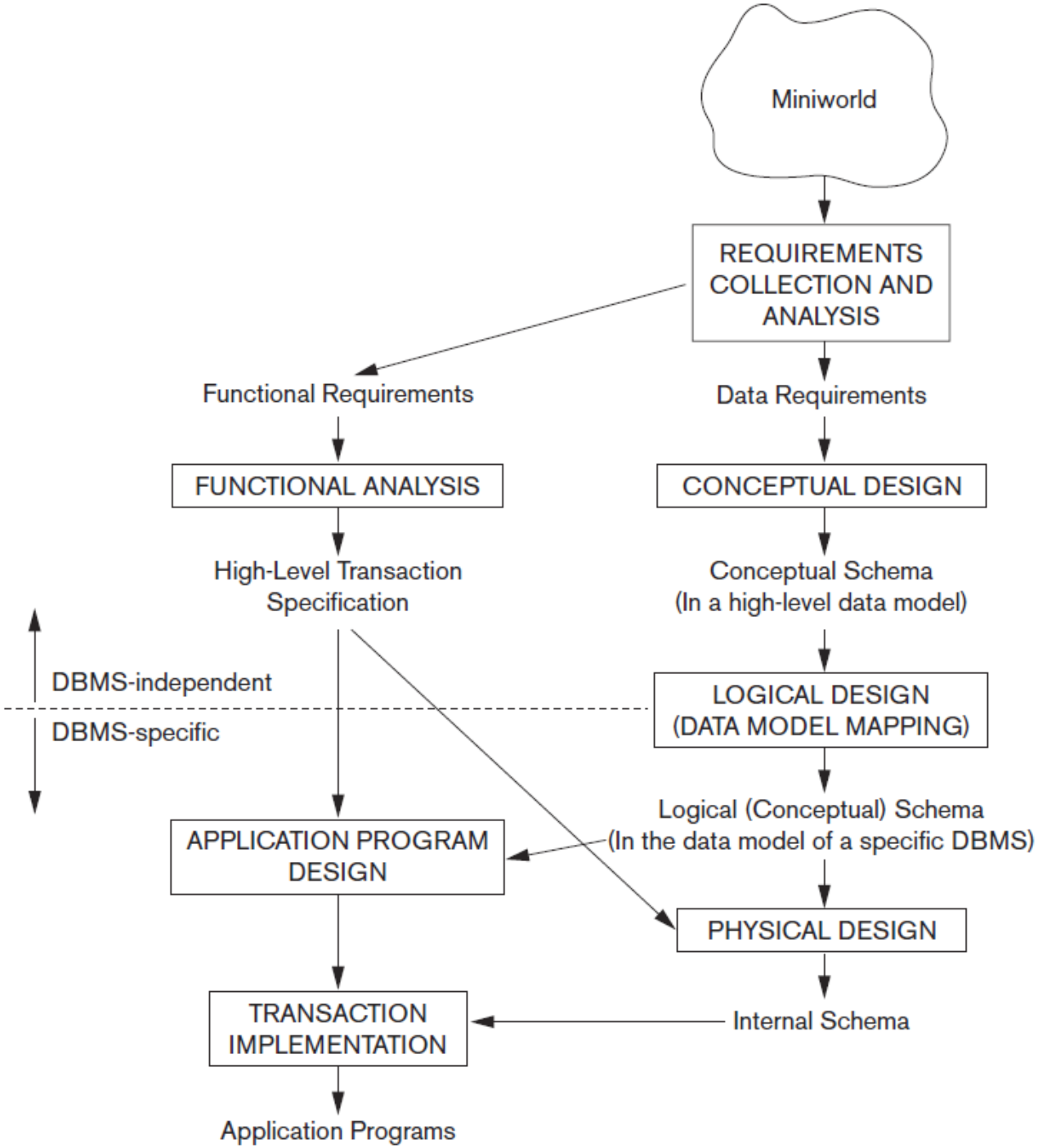
ID	Customer Name	Shipping Address	Zip	City	Newsletter	Item	Price	Supplier	Supplier Phone
1	Jens Bergholm	Mølletoften 20	7100	Vejle	Xbox News	XBOX One	1800	Microsoft	12341324
2	Dennis Jørgensen	Vinkelvej 167	7100	Vejle	Playstation News	Playstation 4	1900	Sony	23452345
3	Hans Jensen	Kongensgade 45	5000	Odense	Xbox News	XBOX One	1800	Microsoft	12341324
3	Hans Jensen	Kongensgade 45	5000	Odense	Playstation News	PS Vita	1500	Sony	23452345
4	Anne Johansen	Vinkelvej 167	7100	Vejle	Playstation News	Playstation 4	1900	Sony	23452345

Redundancy

Domain model vs. ER Diagram

→ ISO/IEC/IEEE 24765 : Systems and software engineering – defines the vocabulary as:
  → **domain model** "a product of domain analysis that provides a representation of the requirements of the domain."
  → **entity-relationship diagram** "a diagram that depicts a set of real-world entities and the logical relationships among them.“
→ But a domain model can evolve into an ER diagram.

The process



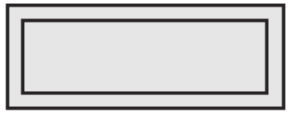
→ DBMS independent works with entities and concepts.
→ DBMS specific works with SQL and Tables.
→ The mapping in the middle is the process of working from the ER diagrams to the table mapping.

In relation to UP

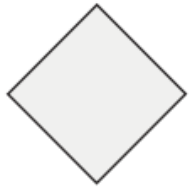
Phase	Artifact
Business Modelling	Domain Model (None from DM)
Requirements	(None from DM)
Analysis	ER, EER, UML (Entities only)
Design	UML (Tables)
Implementation	SQL Creation Script
Test	(None from DM)
Deployment	(None from DM)

ER Components 1/3

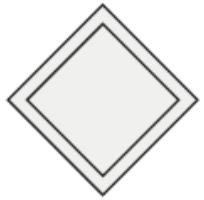
Entity



Weak Entity



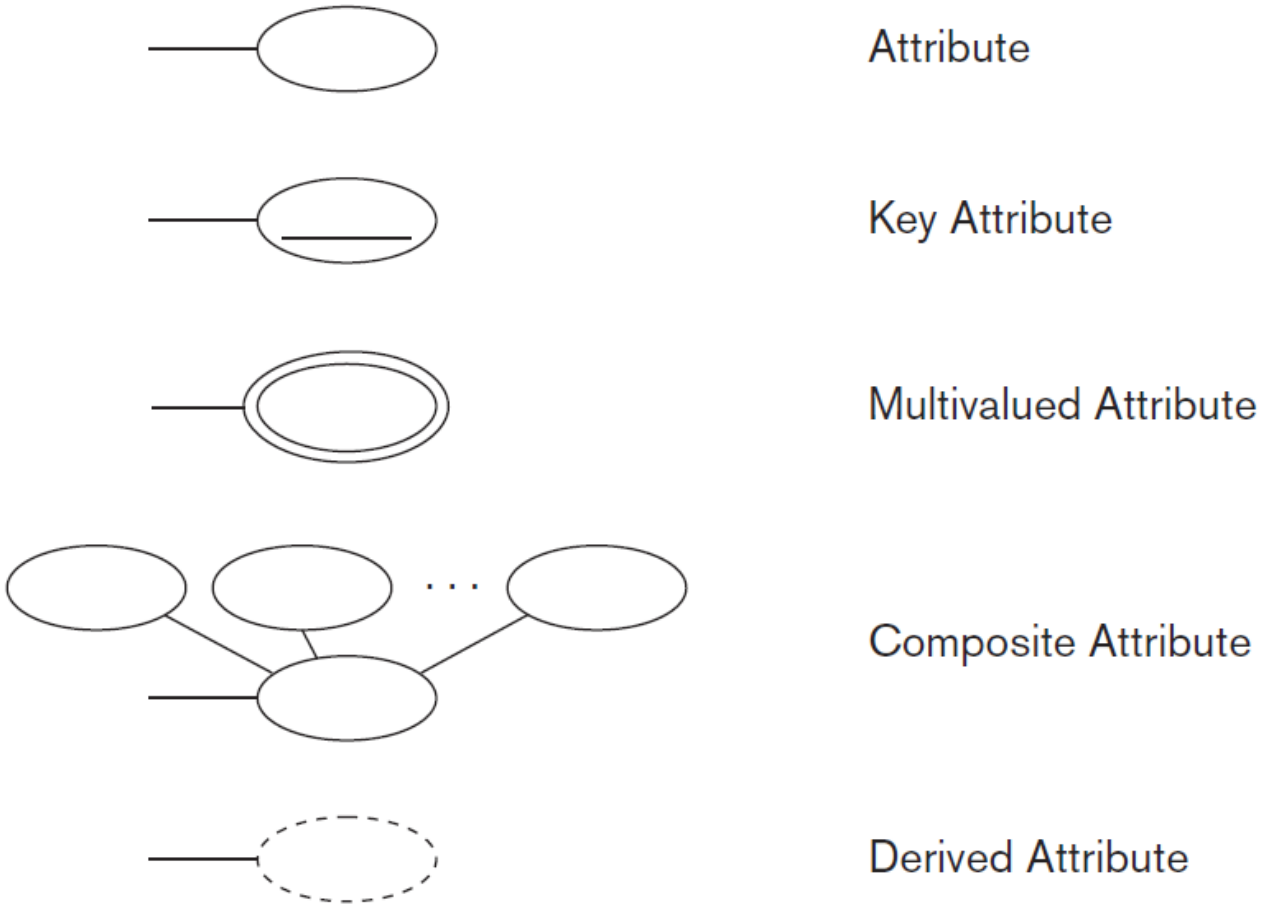
Relationship



Indentifying Relationship

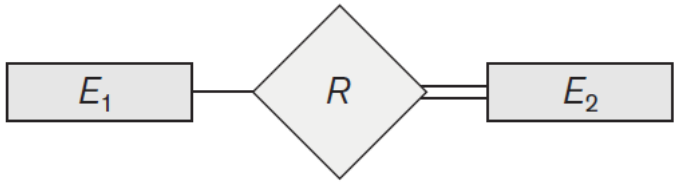
→ An entity type is strong if its existence does not depend on another entity type. Otherwise, the entity type is weak.

ER Components 2/3

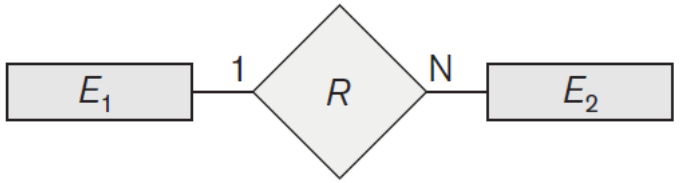


→ …

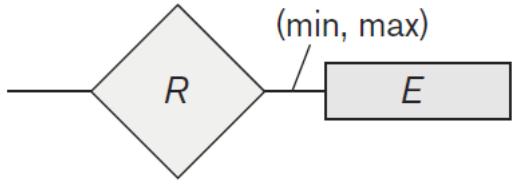
ER Components 3/3



Total Participation of E_2 in R



Cardinality Ratio 1: N for $E_1 : E_2$ in R

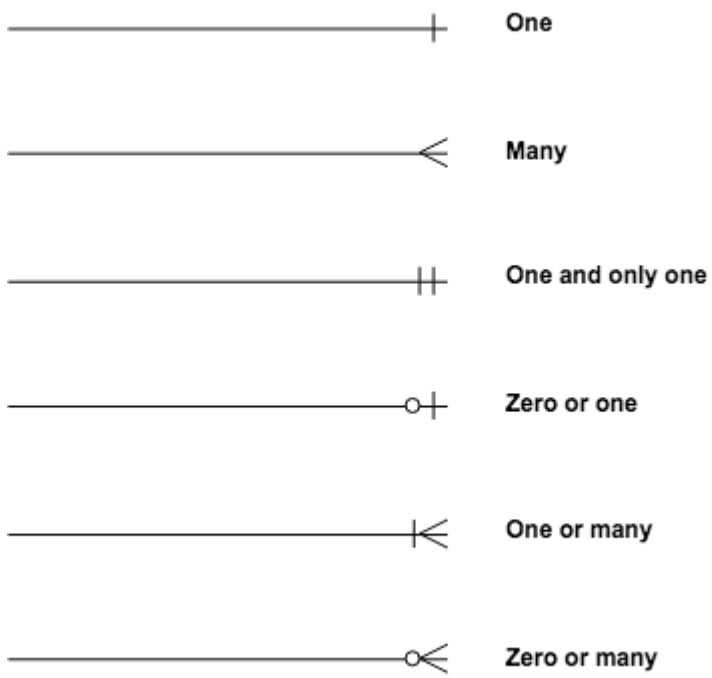


Structural Constraint (min, max) on Participation of E in R

→ Total participation can also be understood as E2 HAS to have one entry.

→ Example, a teacher MUST teach a class (or they are not a teacher).

Cardinalities



Crow’s foot notation

→ 1 - * , one to many, 1:N

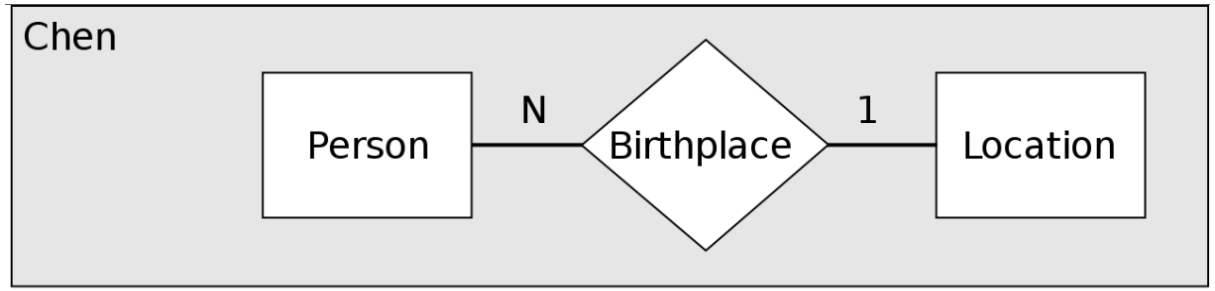
→ * - 1, many to one, N:1

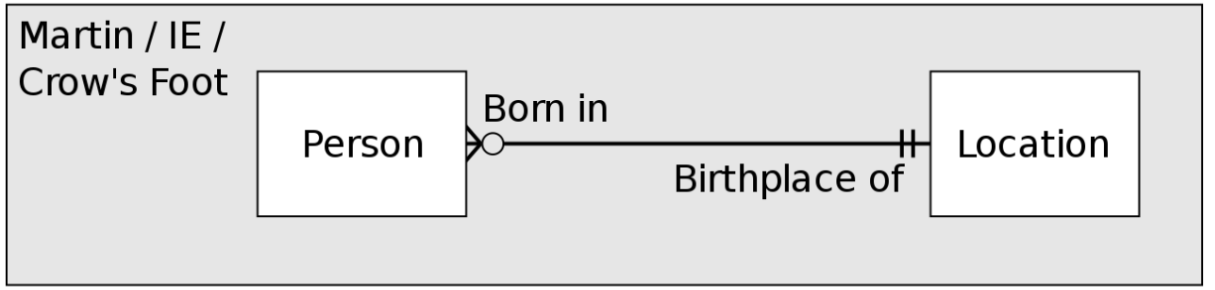
→ * - *, many to many M:N

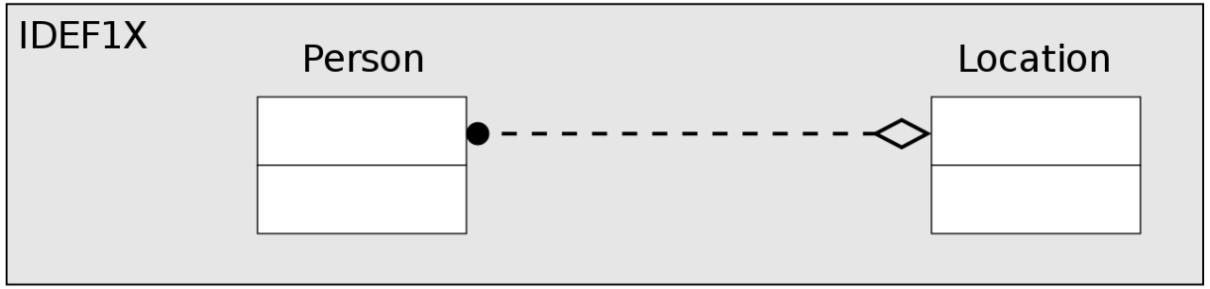
→ 1 - 1, one to one 1:1

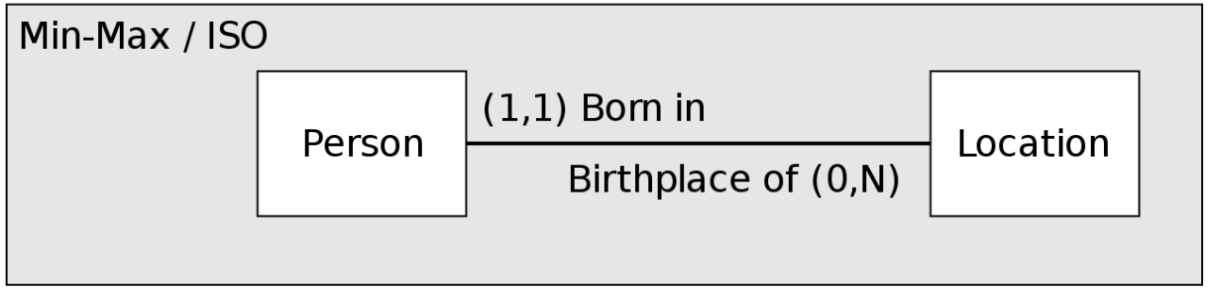
→ 0..1 - 1, zero/one to one, 0..1:1

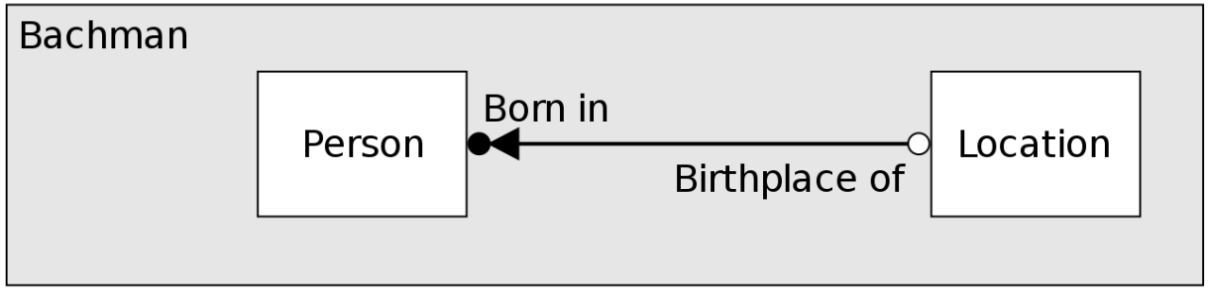
And there are many alternatives

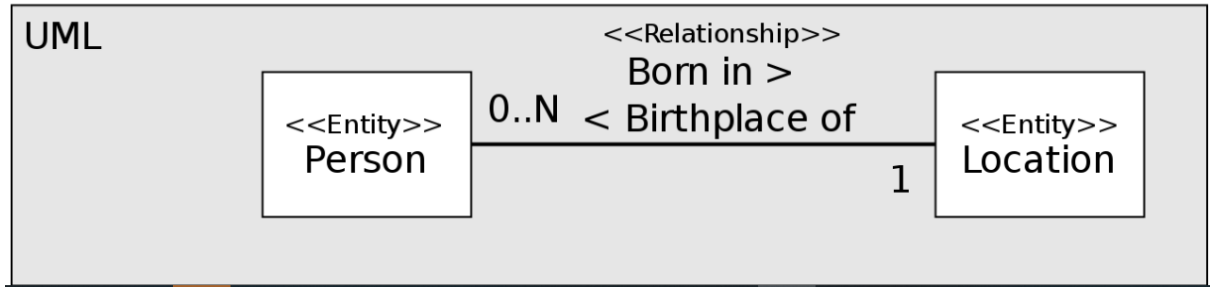




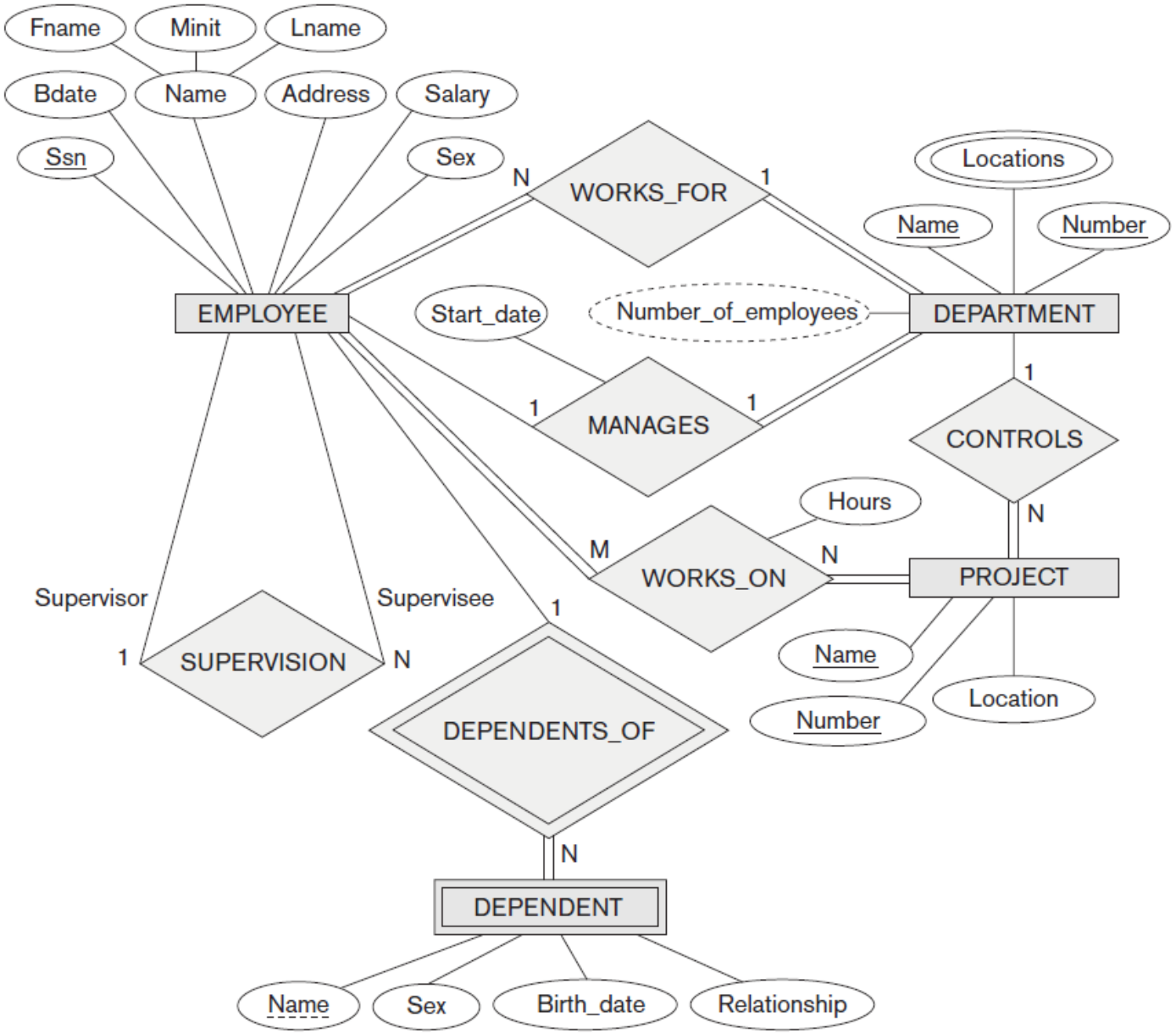






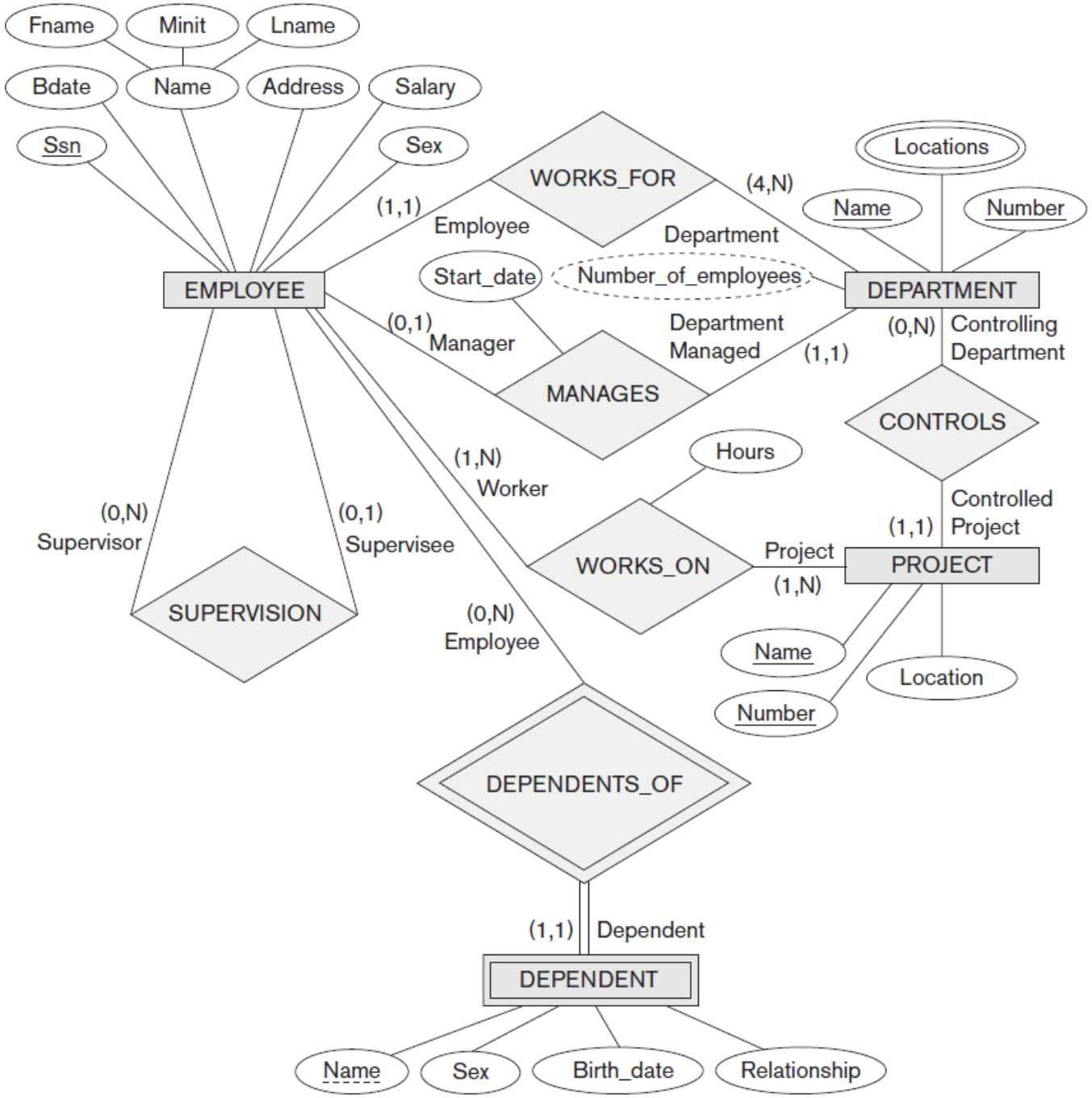


ER Diagram Example



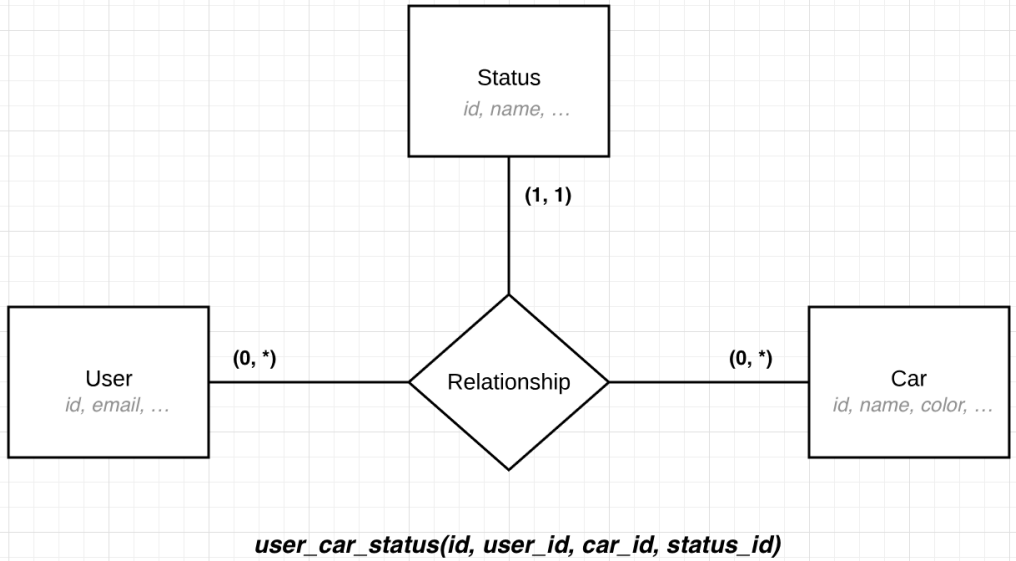
- → Entity types (strong/weak)
- → Relationship types (Identifying or not)
- → Recursive relationship
- → Attribute types (composite, key, derived, multiple values, etc.)
- → Cardinalities

Alternative structural constraints



→ Min, Max notation

N-ary



→ Multiple relationships in one

→ Results in table with multiple foreign keys

EER: Enhanced entity-relationship diagram

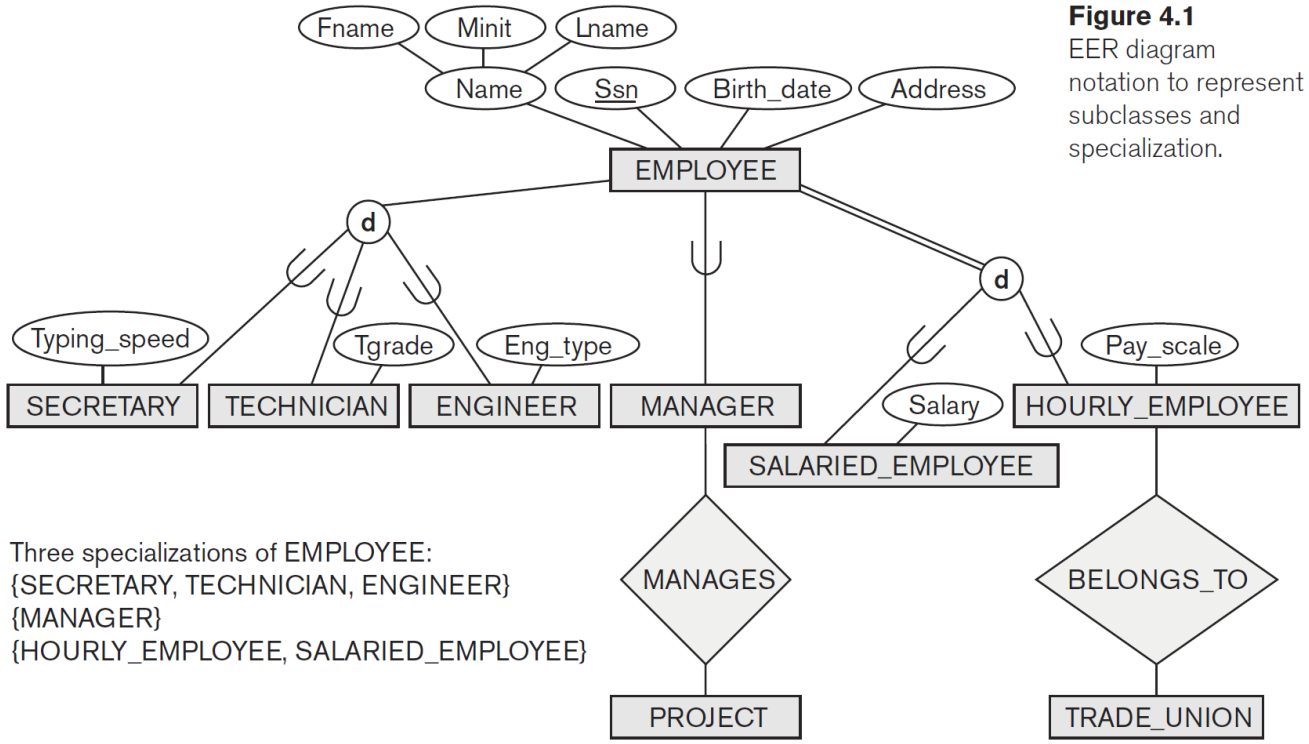


Figure 4.1
EER diagram notation to represent subclasses and specialization.

Three specializations of EMPLOYEE:
{SECRETARY, TECHNICIAN, ENGINEER}
{MANAGER}
{HOURLY_EMPLOYEE, SALARIED_EMPLOYEE}

→ Expands with the following components:

- → Attribute or relationship inheritances
- → Category or union types
- → Specialization and generalization
- → Subclasses and superclasses

Inheritance

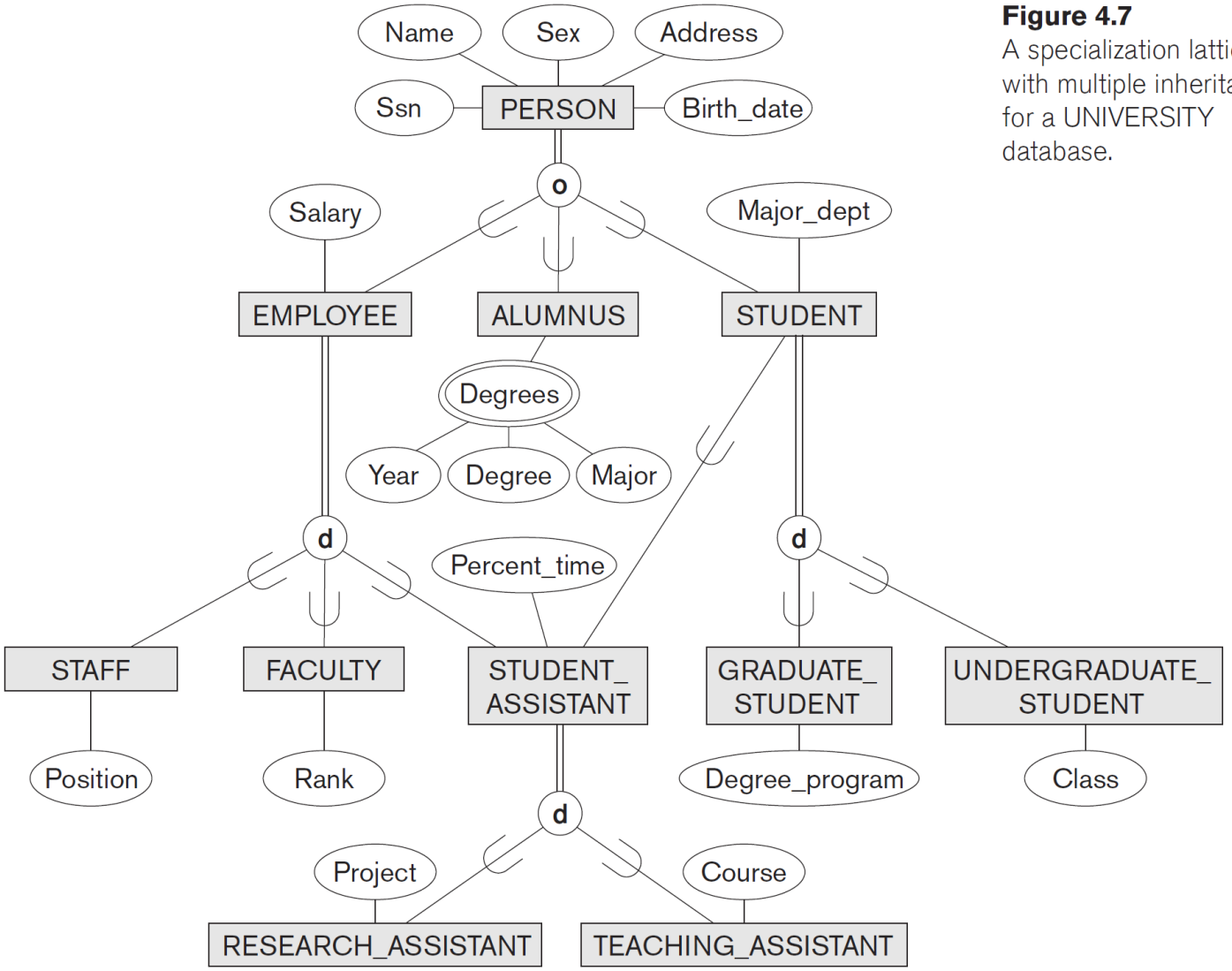
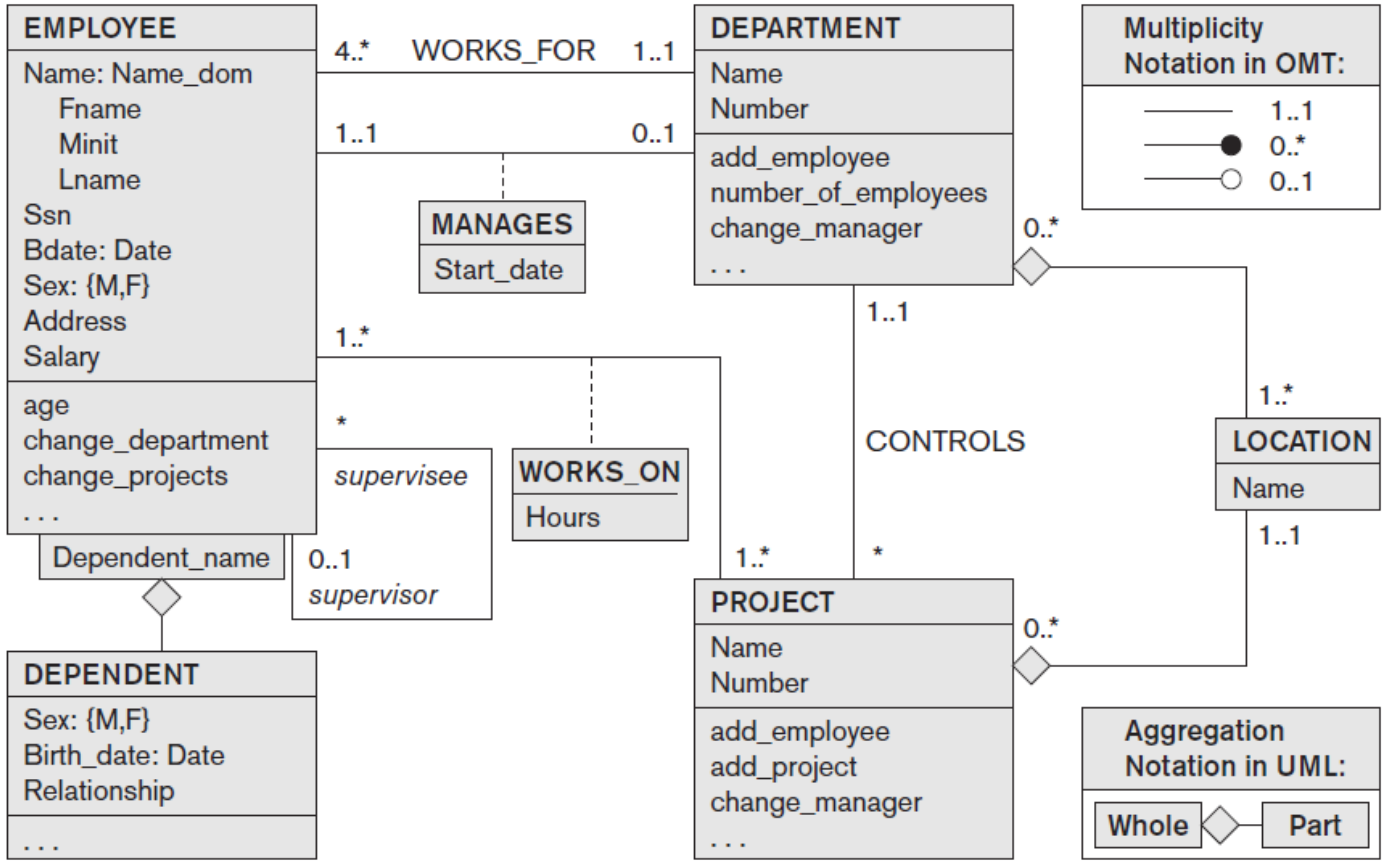


Figure 4.7
A specialization lattice with multiple inheritance for a UNIVERSITY database.

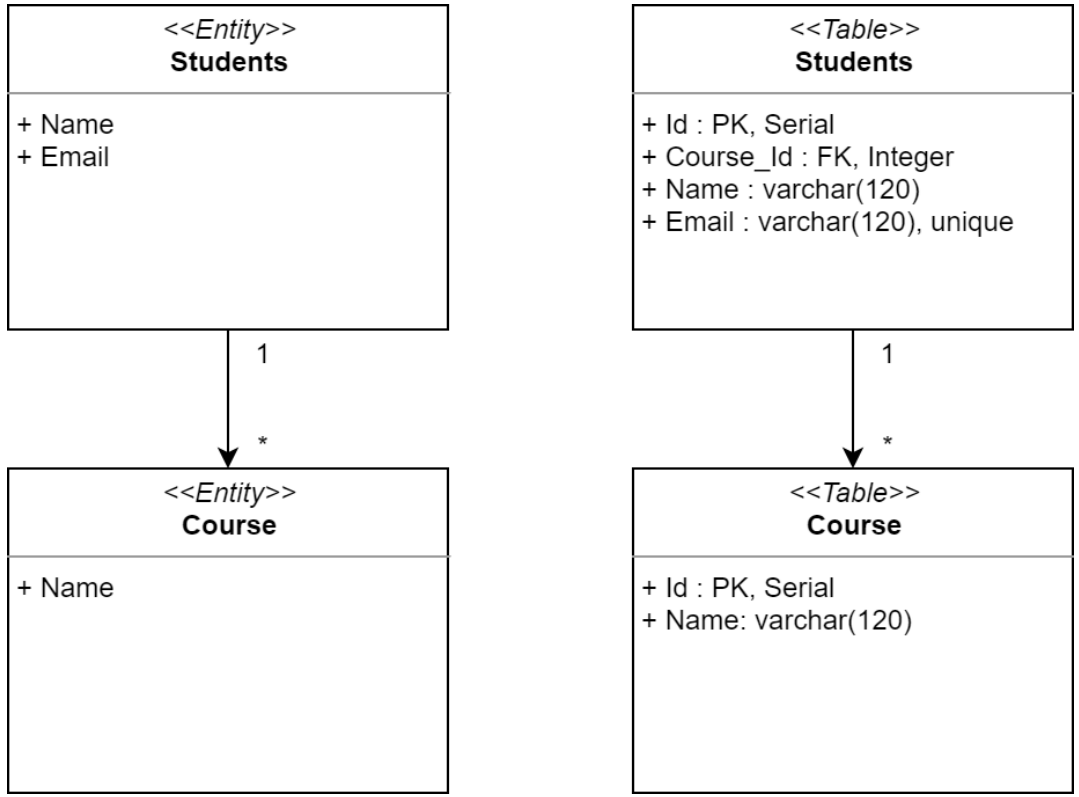
- → Disjoint
  - → Is one of
- → Overlapping
  - → Can be any of, and more than one.

UML Notation



- → Often preferred due to the common use of UML for class diagrams.
- → Notation differs.
- → Not the main notation for this class!
- → At the exam use the notation shown prior to UML in these slides.

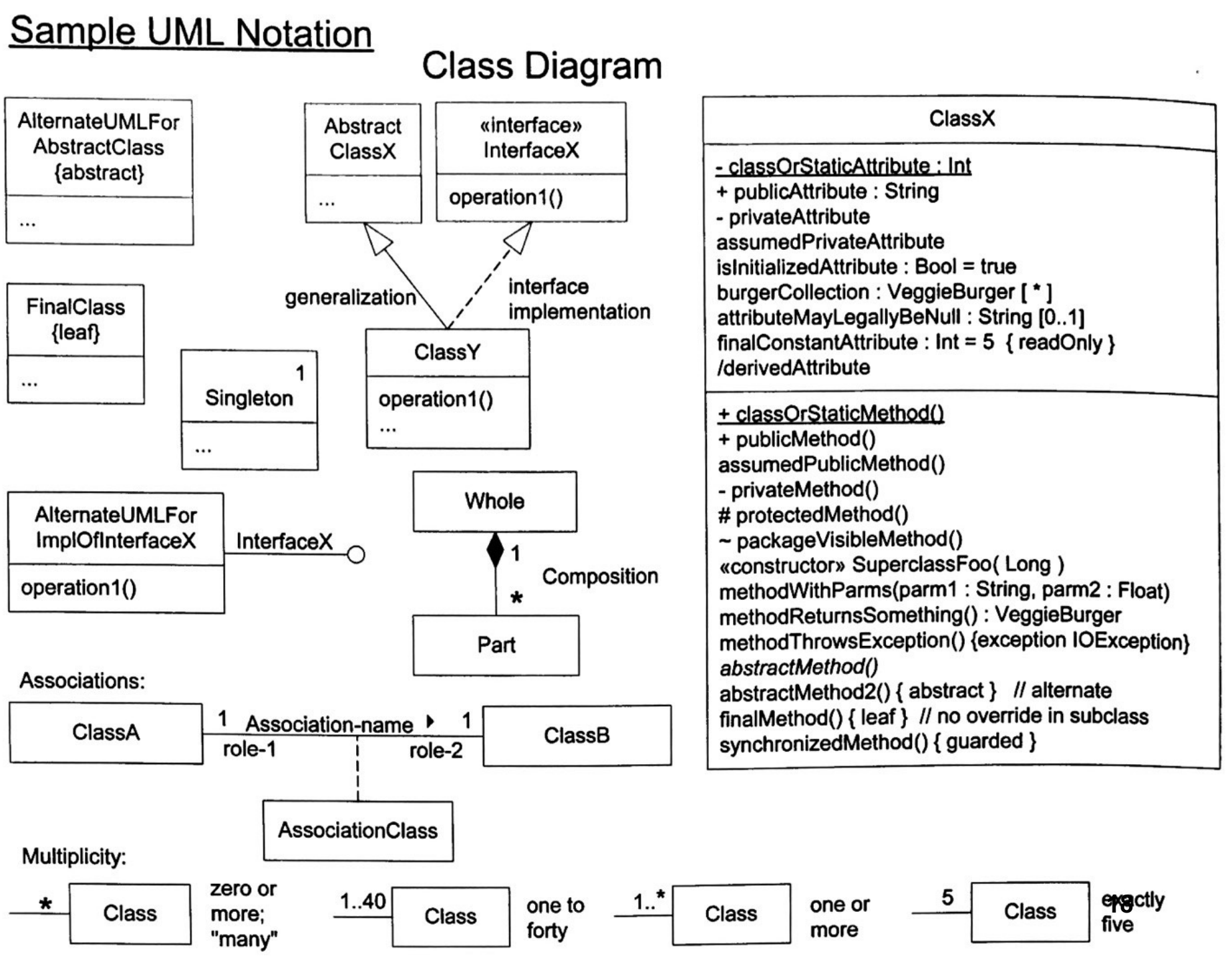
UML Entity vs Table



→ If no << >> is present, we are talking about an entity.

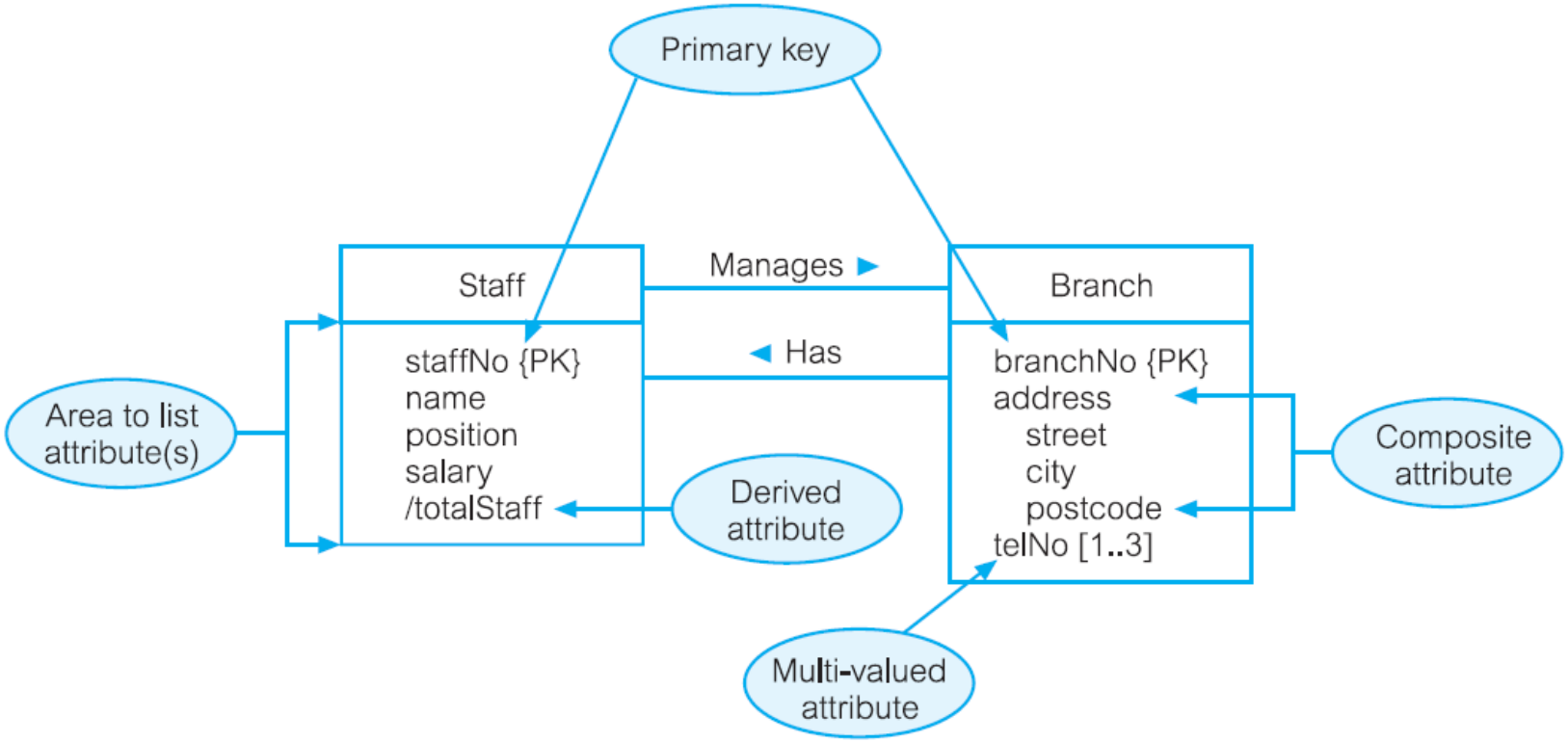
→ **Please be explicit in this course about what type it is!**

Standard UML Notation

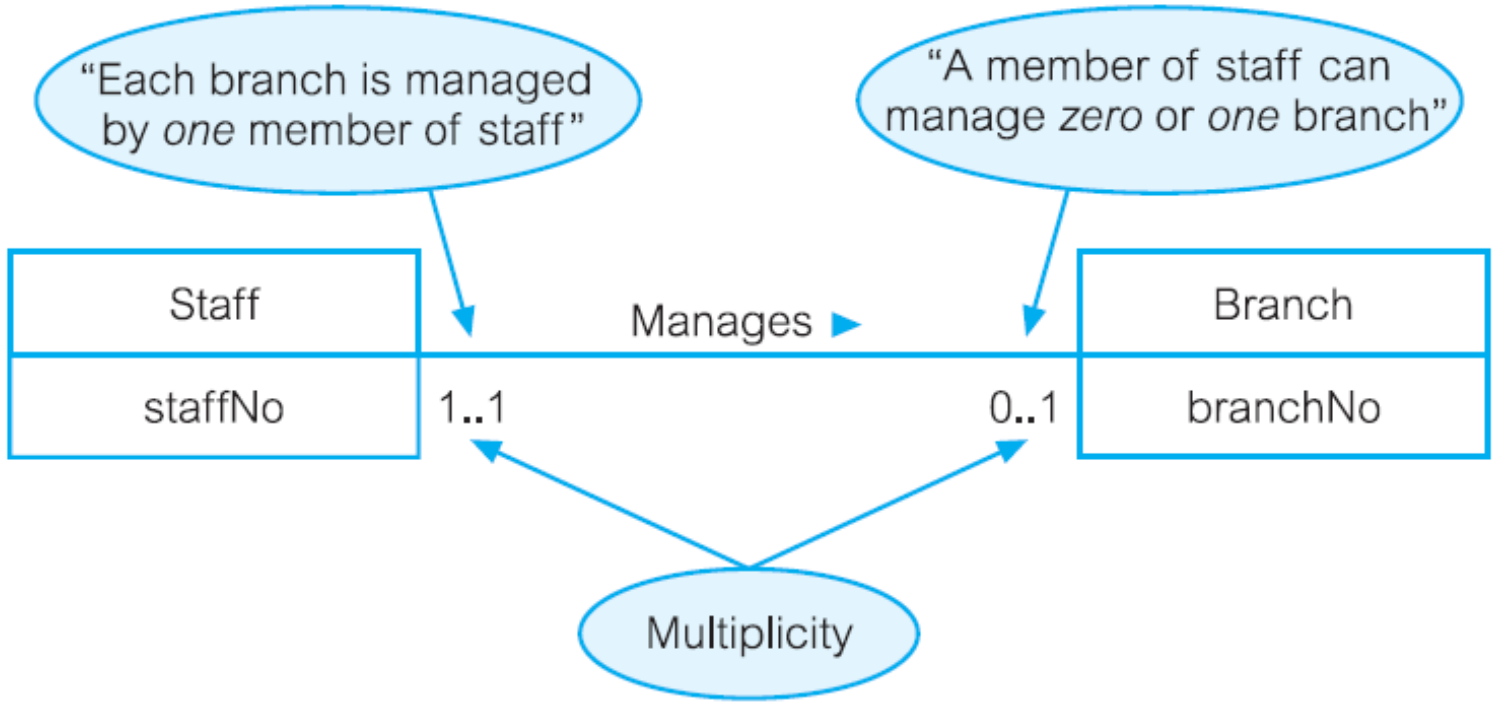


Source:
Applying UML and Patterns, Larman, 2009

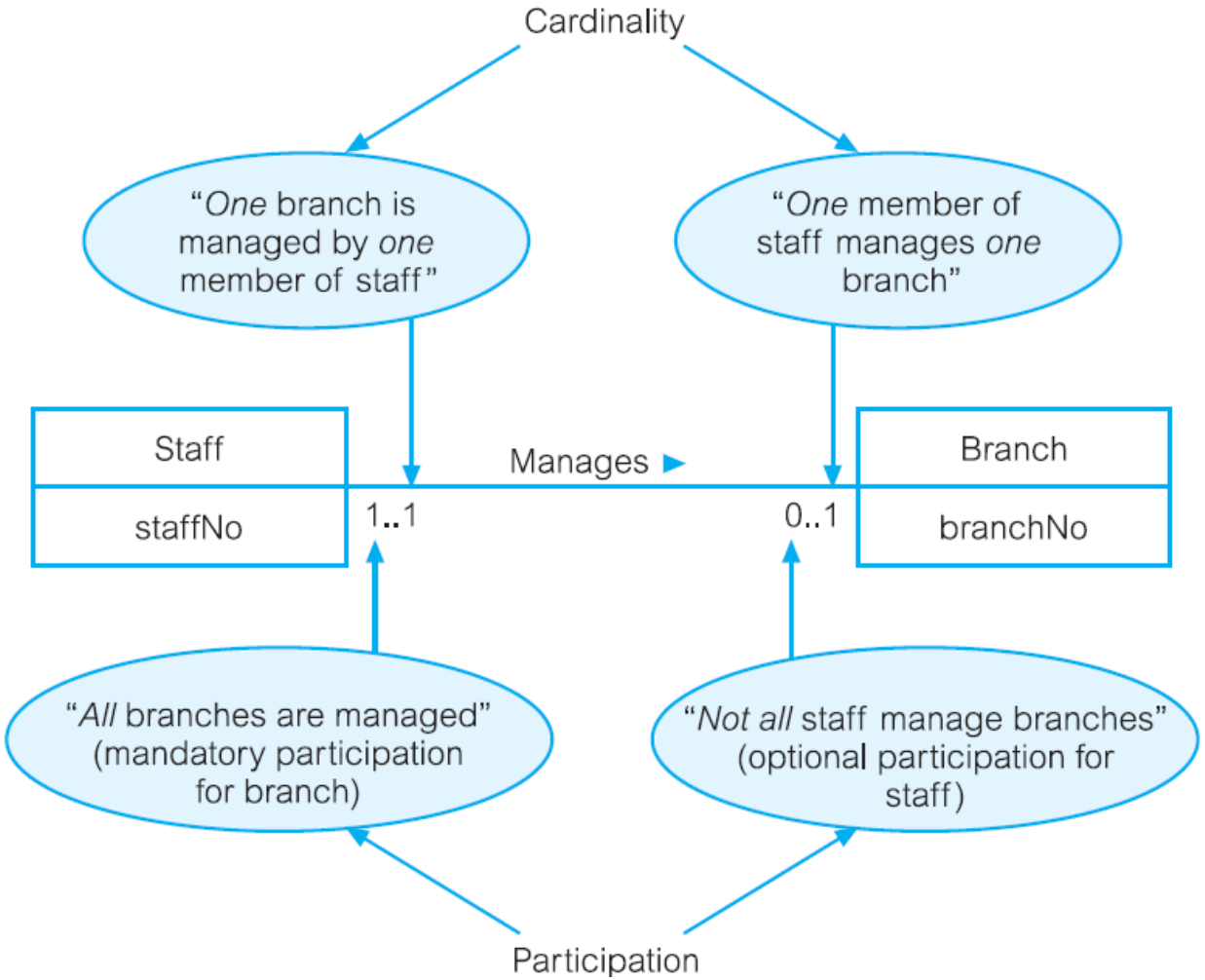
Attributes in UML



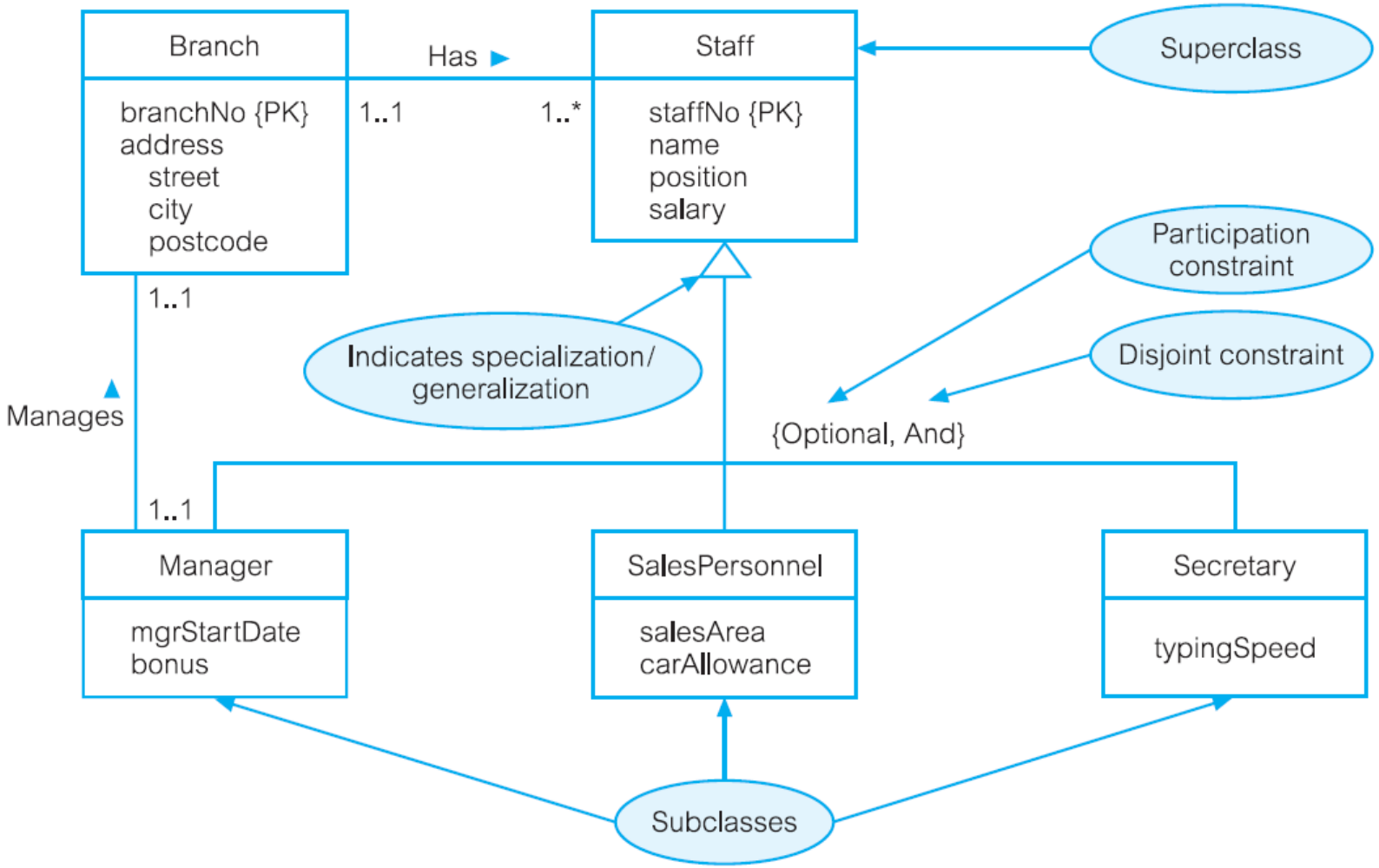
Cardinalities in UML - 1/2



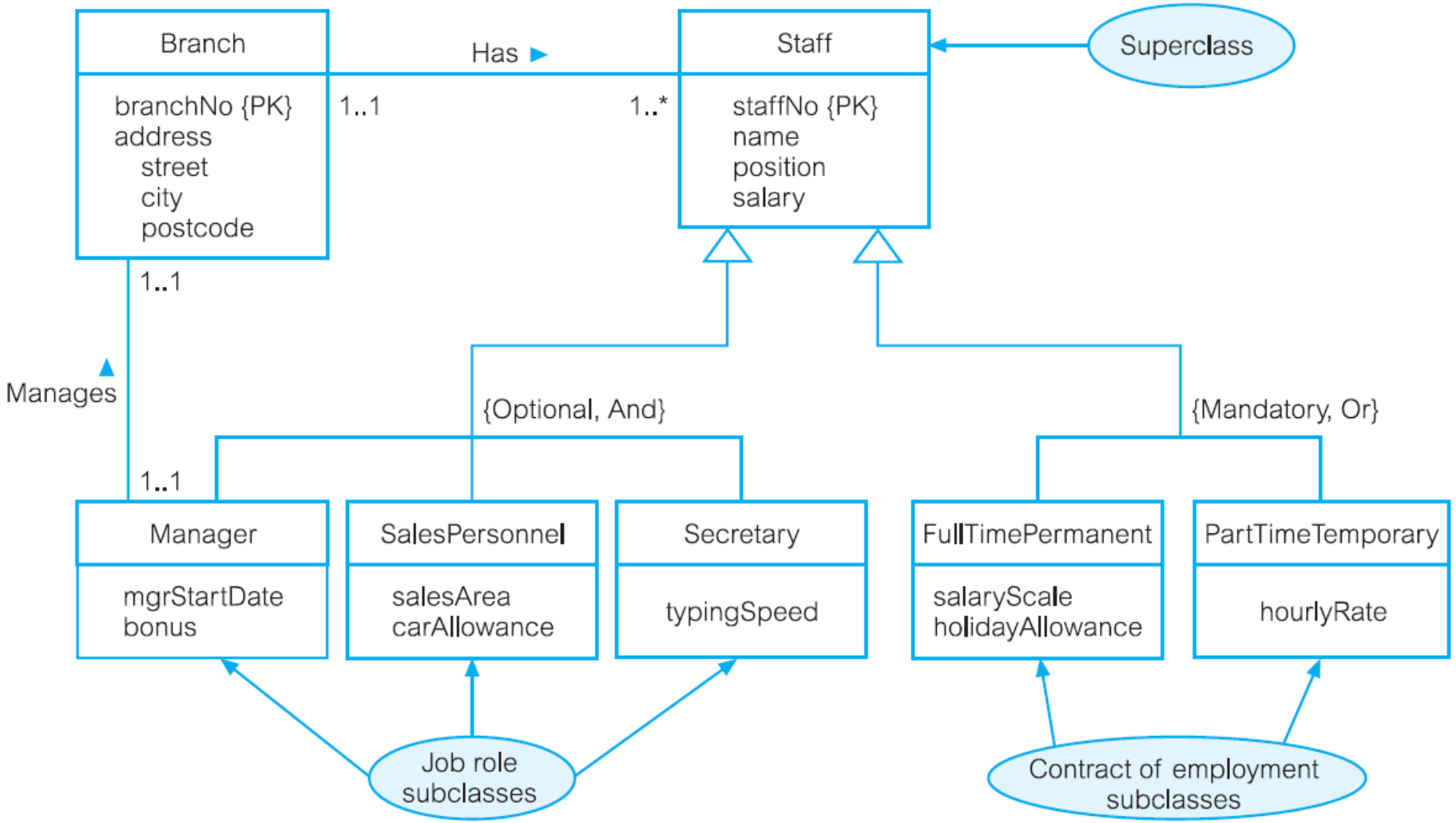
Cardinalities in UML – 2/2



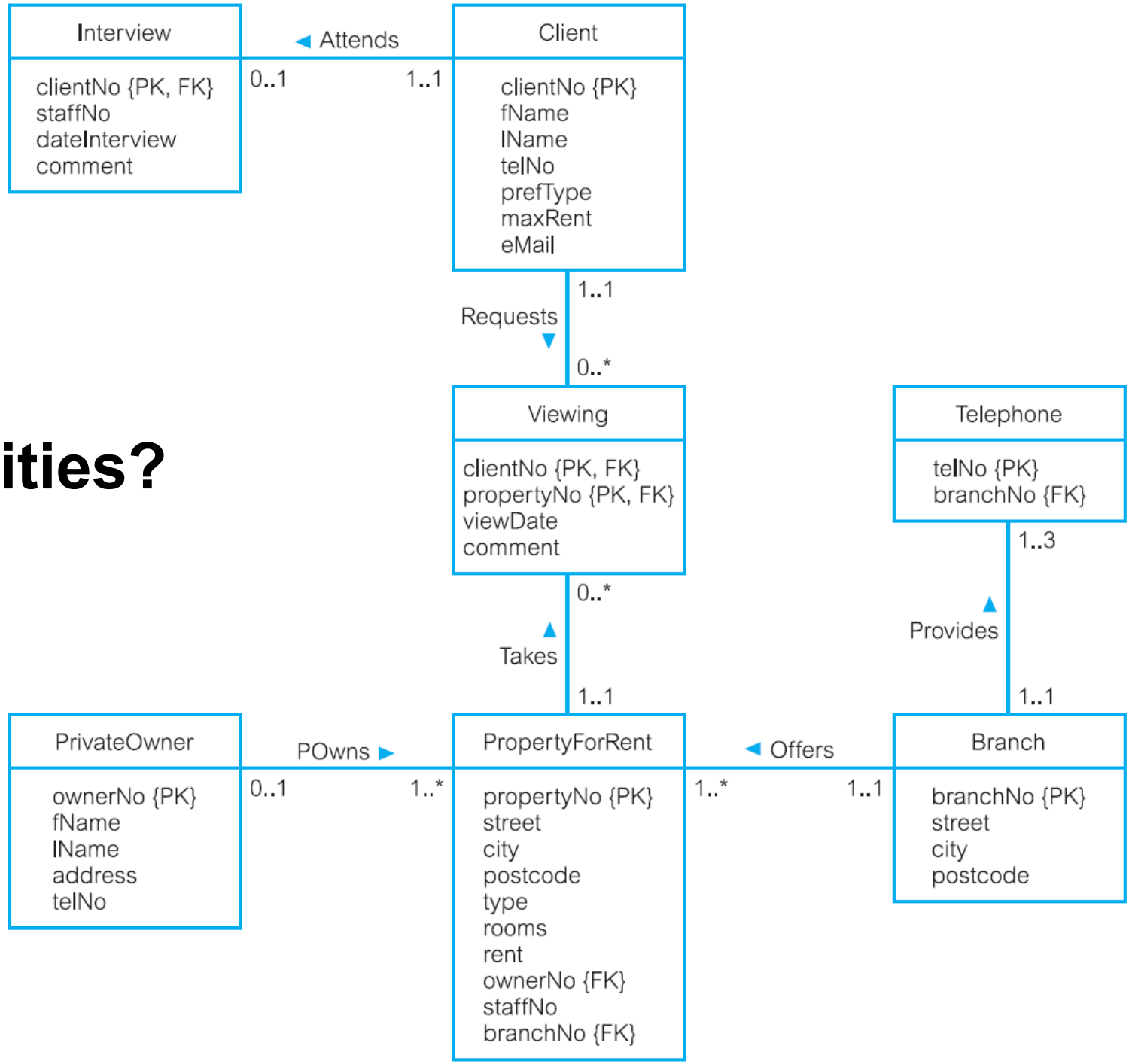
EER in UML – 1/2

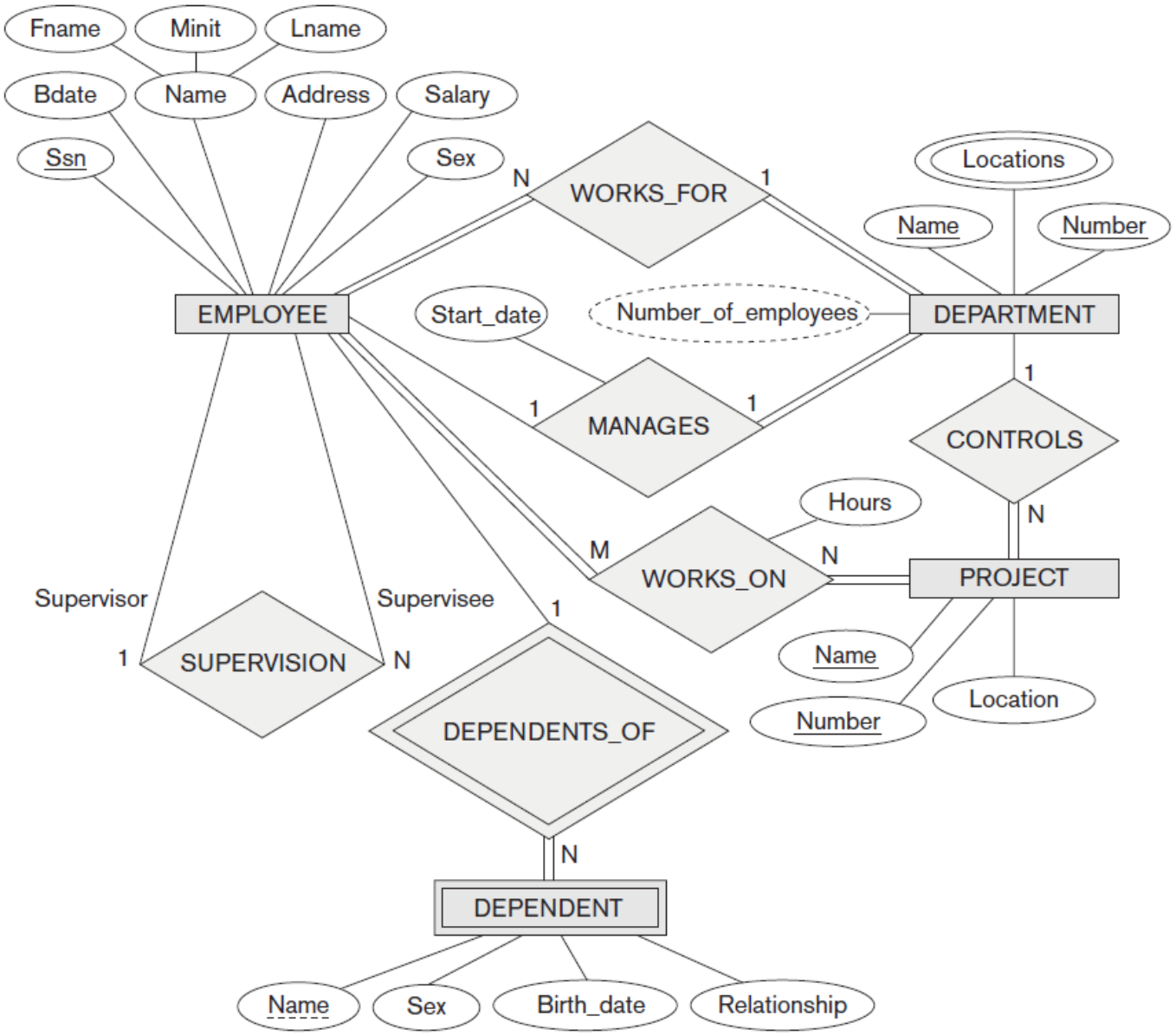

Branch
branchNo {PK}
address
street
city
postcode
Has
1..1
1..*
Staff
staffNo {PK}
name
position
salary
Superclass
Participation constraint
Disjoint constraint
Indicates specialization/ generalization
{Optional, And}
Manages
1..1
1..1
Manager
mgrStartDate
bonus
SalesPersonnel
salesArea
carAllowance
Secretary
typingSpeed
Subclasses

EER in UML – 2/2


Branch
branchNo {PK}
address
street
city
postcode
Has
1..1
1..*
Staff
staffNo {PK}
name
position
salary
Superclass
1..1
Manages
1..1
{Optional, And}
{Mandatory, Or}
Manager
mgrStartDate
bonus
SalesPersonnel
salesArea
carAllowance
Secretary
typingSpeed
FullTimePermanent
salaryScale
holidayAllowance
PartTimeTemporary
hourlyRate
Job role subclasses
Contract of employment subclasses

Tables or entities?





Mapping ER to Tables

- → Step 1: Mapping of Regular Entity Types.
- → Step 2: Mapping of Weak Entity Types.
- → Step 3: Mapping of Binary 1:1 Relationship Types.
  - → Foreign key approach
  - → Merged relation approach
  - → Cross-reference or relationship relation approach
- → Step 4: Mapping of Binary 1:N Relationship Types.
  - → The foreign key approach.
  - → The relationship relation approach.
- → Step 5: Mapping of Binary M:N Relationship Types.
- → Step 6: Mapping of Multivalued Attributes.
- → Step 7: Mapping of N-ary Relationship Types.

Figure 9.3
Illustration of some mapping steps.
(a) *Entity* relations after step 1.
(b) Additional *weak entity* relation after step 2.
(c) *Relationship* relations after step 5.
(d) Relation representing multivalued attribute after step 6.

(a)

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary

DEPARTMENT

Dname	Dnumber

PROJECT

Pname	Pnumber	Plocation

(b)

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

(c)

WORKS_ON

Essn	Pno	Hours

(d)

DEPT_LOCATIONS

Dnumber	Dlocation

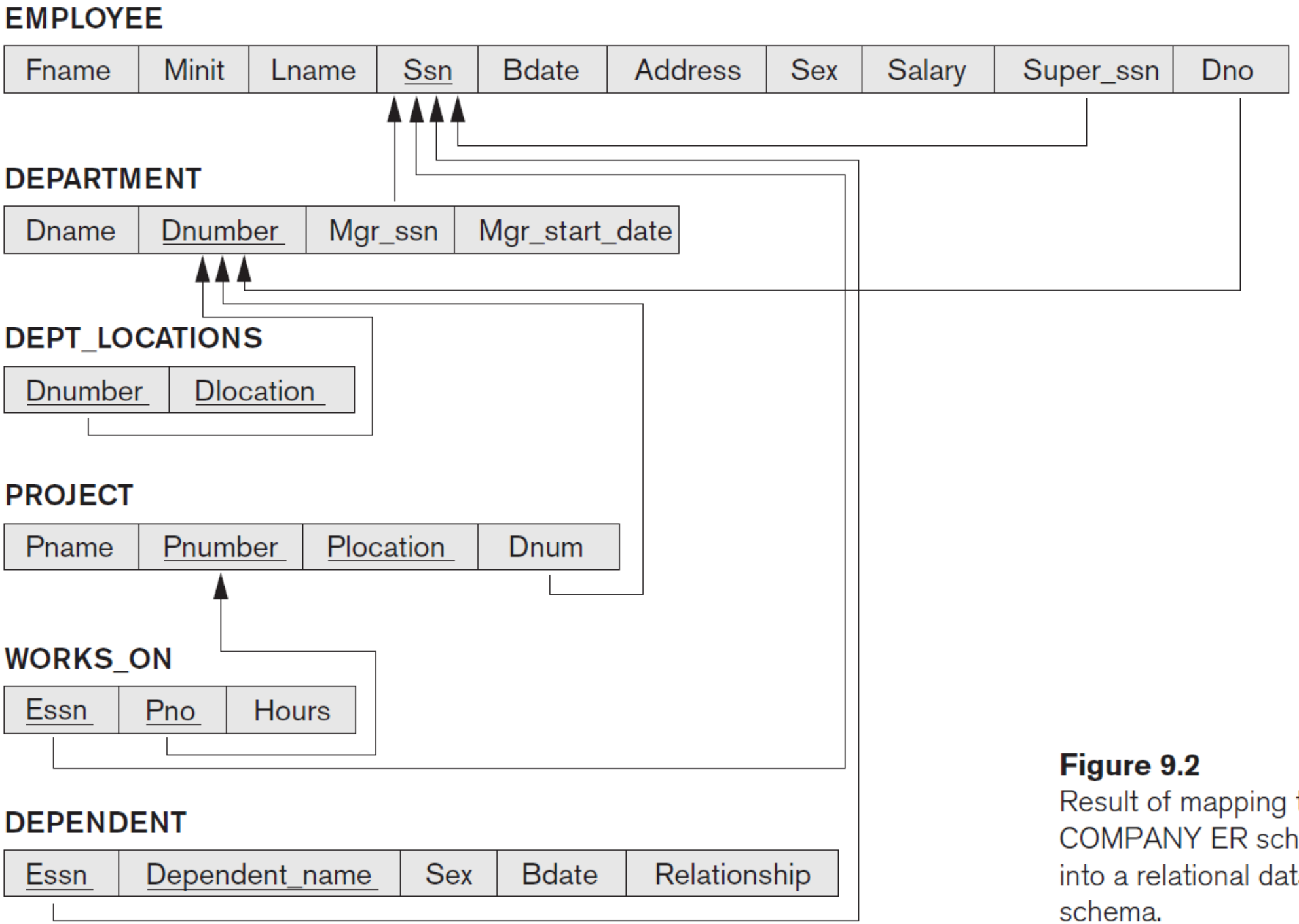
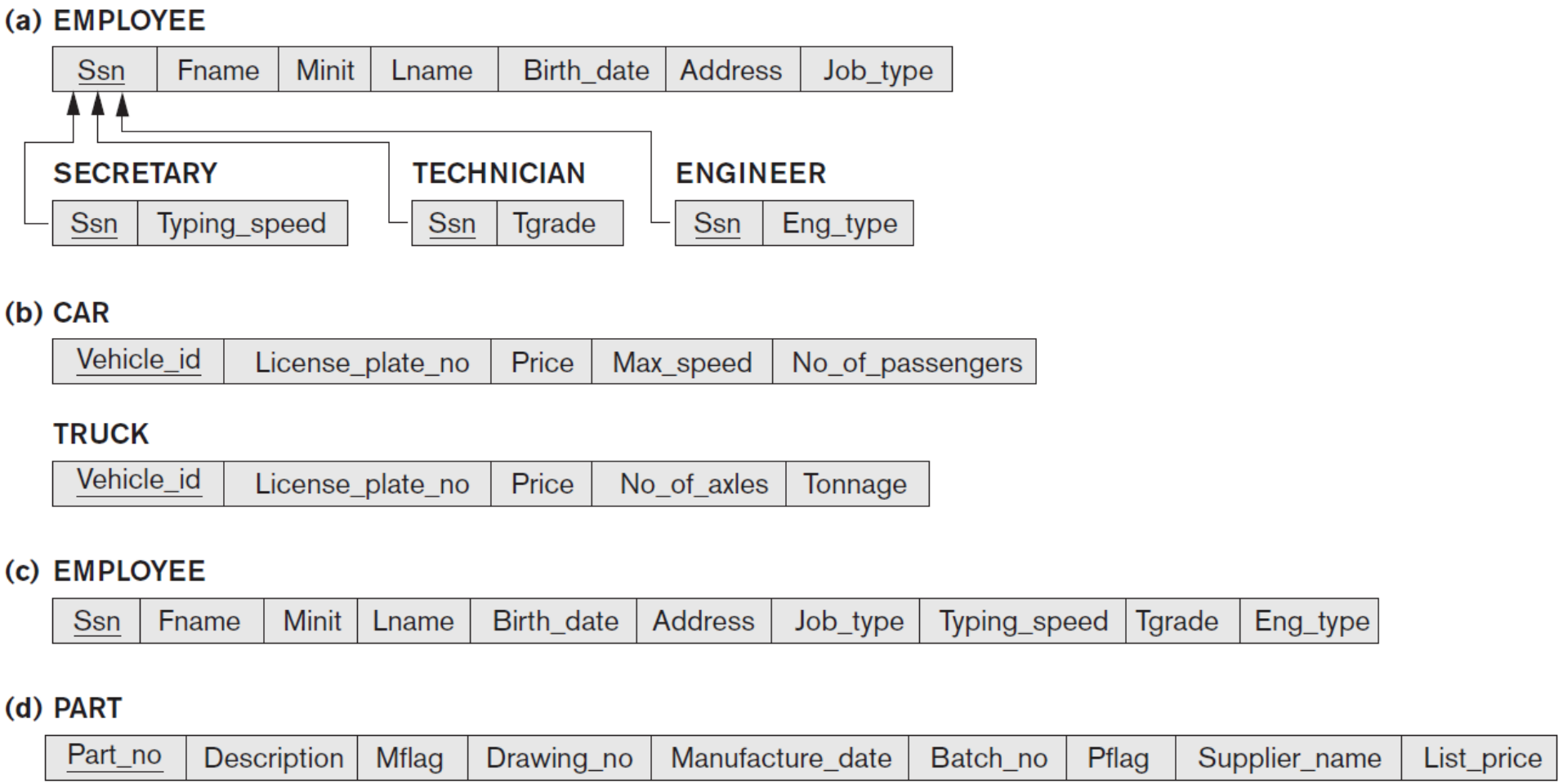


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

Maturing the UML diagram to tables (mixed example!)



In relation to UP – Repeated

Phase	Artifact
Business Modelling	Domain Model (None from DM)
Requirements	(None from DM)
Analysis	ER, EER, UML (Entities only)
Design	UML (Tables)
Implementation	SQL Creation Script
Test	(None from DM)
Deployment	(None from DM)

ER model vs Relational Model (Tables)

Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	*Entity* relation
1:1 or 1:N relationship type	Foreign key (or *relationship* relation)
M:N relationship type	*Relationship* relation and *two* foreign keys
n-ary relationship type	*Relationship* relation and *n* foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

Process Recap

ER / EER / UML (Entity)

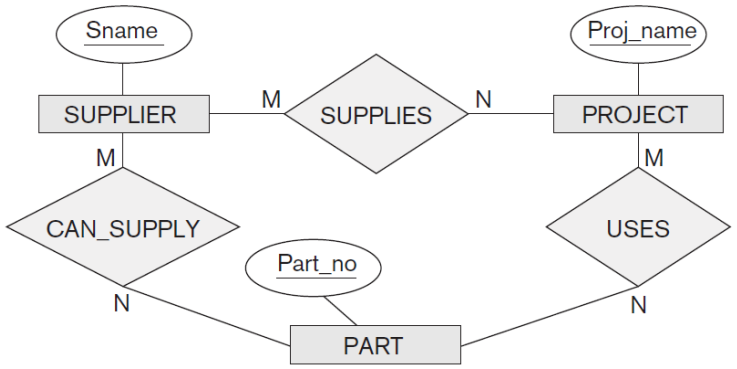


Table format / UML (Table)

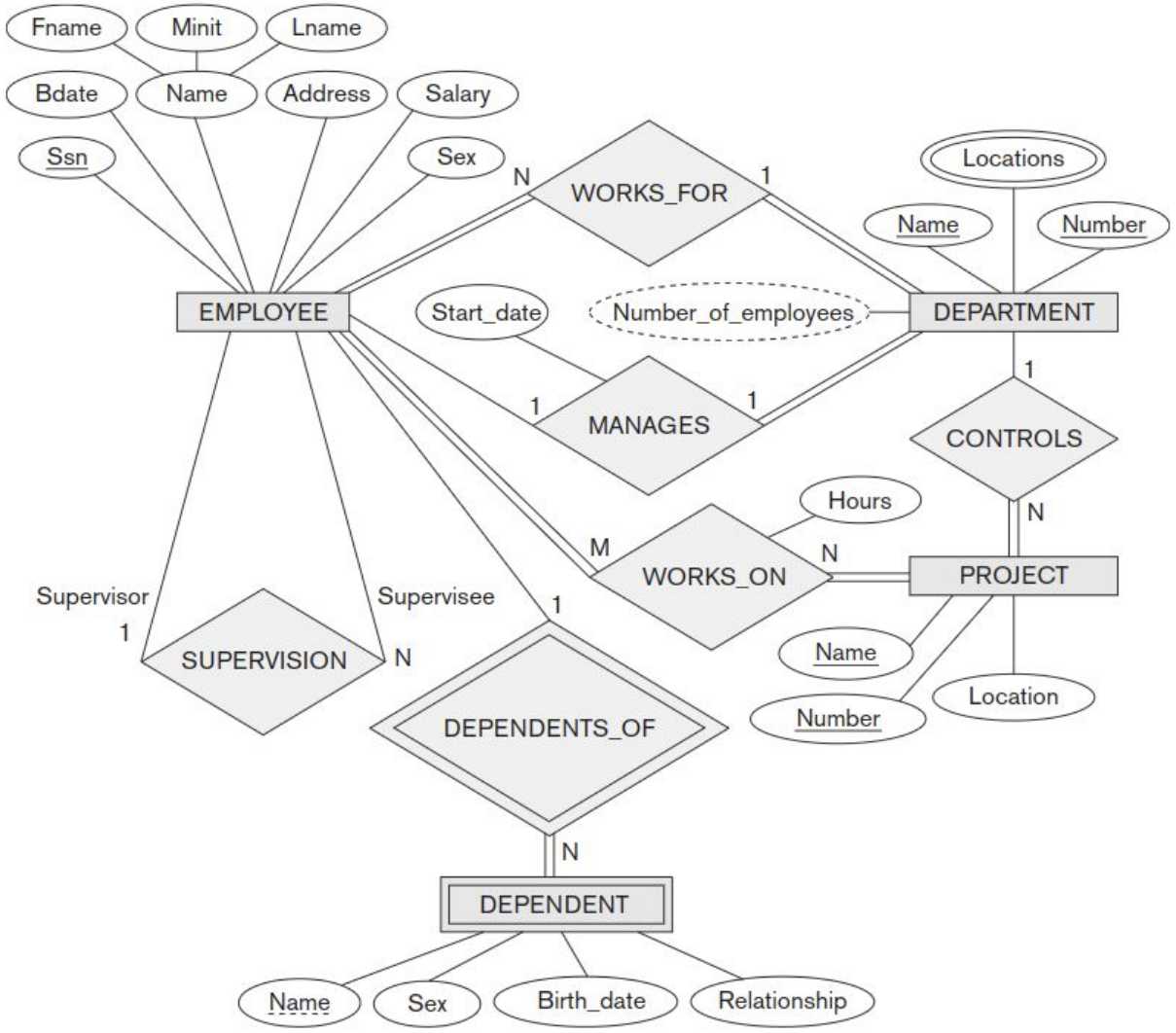
Remember << Table >>

EMPLOYEE

Fname	Minit	Lname	Ssn

SQL Creation Script

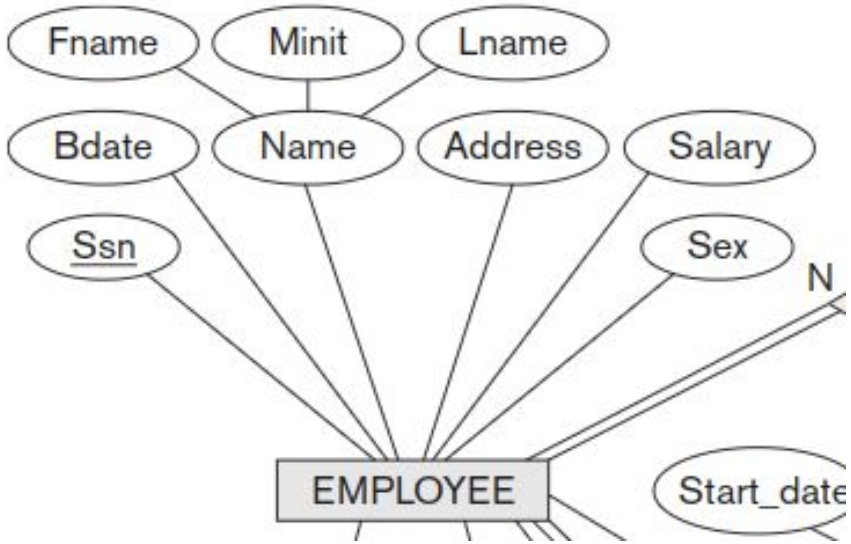
```sql
CREATE TABLE account(
    user_id serial PRIMARY KEY,
    username VARCHAR (50) UNIQUE NOT NULL,
    password VARCHAR (50) NOT NULL,
    email VARCHAR (355) UNIQUE NOT NULL,
    created_on TIMESTAMP NOT NULL,
    last_login TIMESTAMP
);
```



Mapping ER to Tables

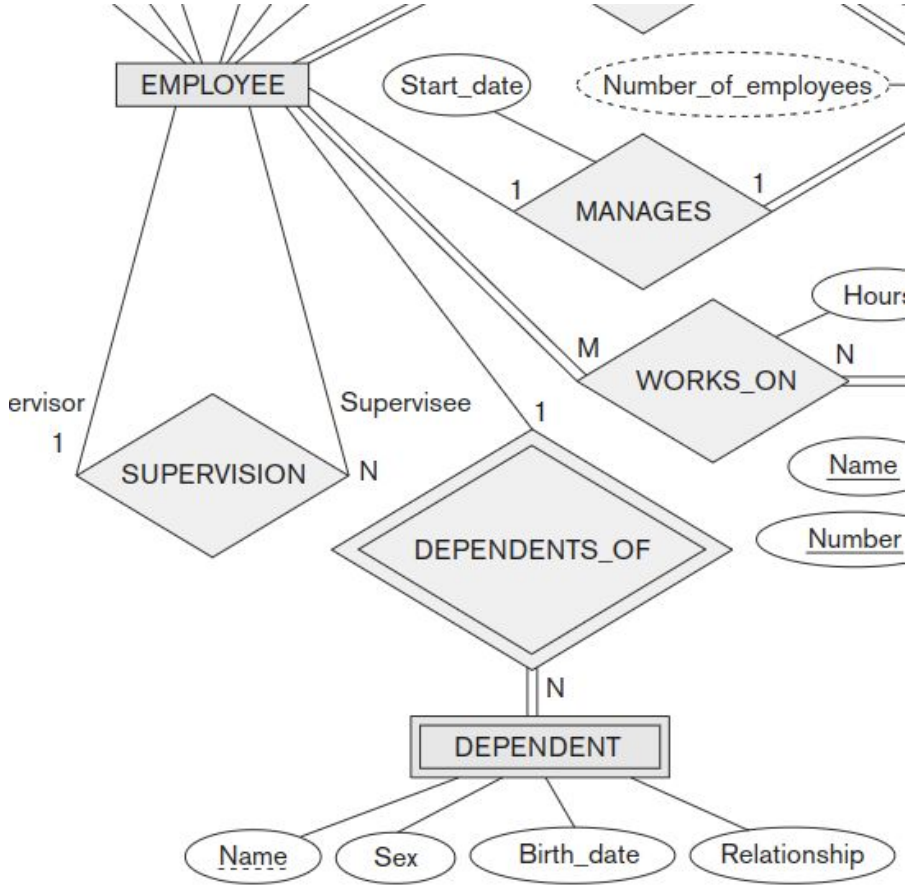
- → Step 1: Mapping of Regular Entity Types.
- → Step 2: Mapping of Weak Entity Types.
- → Step 3: Mapping of Binary 1:1 Relationship Types.
  - → Foreign key approach
  - → Merged relation approach
  - → Cross-reference or relationship relation approach
- → Step 4: Mapping of Binary 1:N Relationship Types.
  - → The foreign key approach.
  - → The relationship relation approach.
- → Step 5: Mapping of Binary M:N Relationship Types.
- → Step 6: Mapping of Multivalued Attributes.
- → Step 7: Mapping of N-ary Relationship Types.

Regular/Strong entities




```sql
create table employee
(
    ssn     int primary key,
    bdate   date,
    fname   varchar,
    minit   varchar,
    lname   varchar,
    address varchar,
    salary  real,
    sex     varchar
);
```

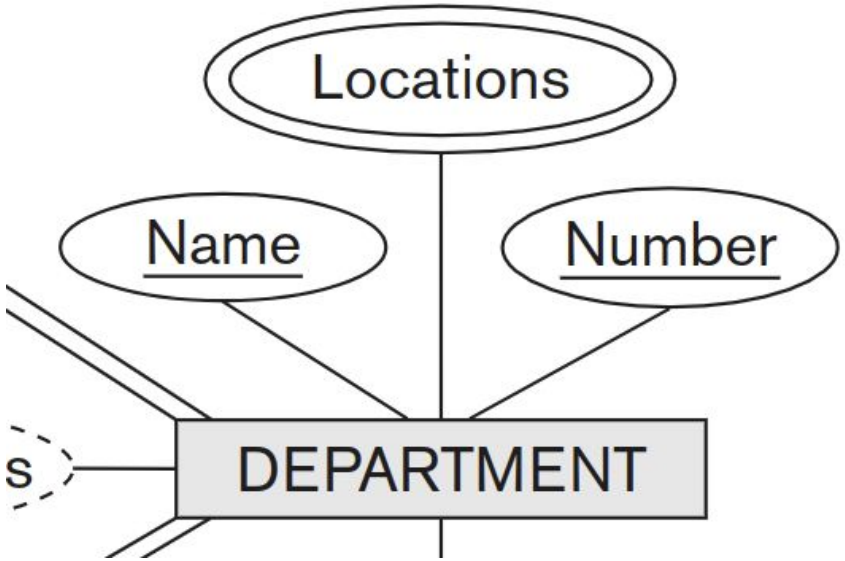
Weak entities




```sql
create table dependent
(
    employee_ssn int references employee (ssn),
    name         varchar,
    sex          varchar,
    birth_date   date,
    relationship varchar,
    primary key (employee_ssn, name)
);
```

Multi-valued attributes

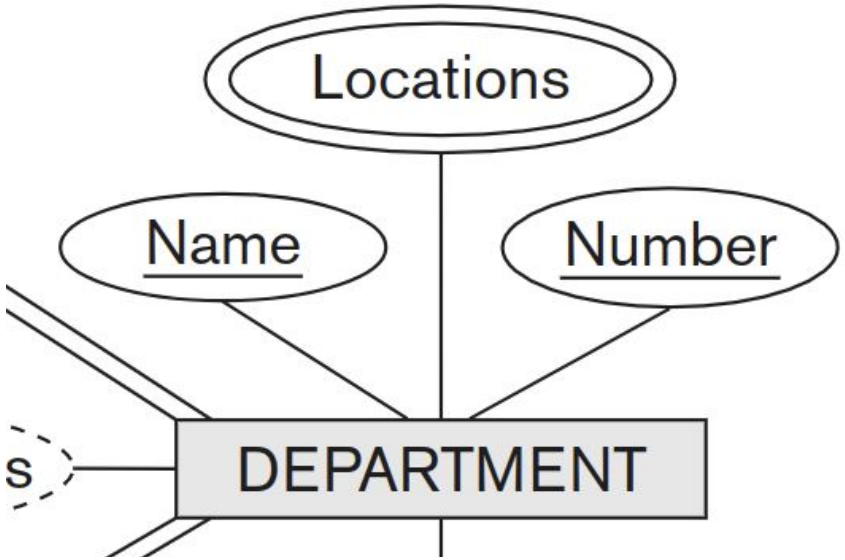
```sql
create table department
(
    number int,
    name   varchar unique ,
    primary key (number, name)
);
```




```sql
create table department_locations
(
    department_number int,
    department_name varchar,
    location varchar,
    foreign key (department_number, department_name) references department(number, name),
    primary key (department_number,department_name,location)
);
```

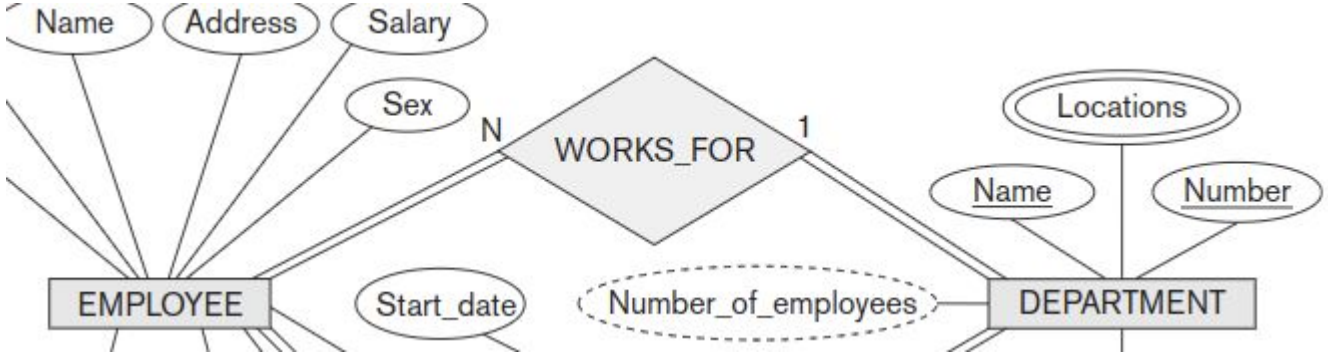
Multi-valued attributes

```sql
create table department
(
    number int,
    name   varchar unique ,
    primary key (number, name)
);
```



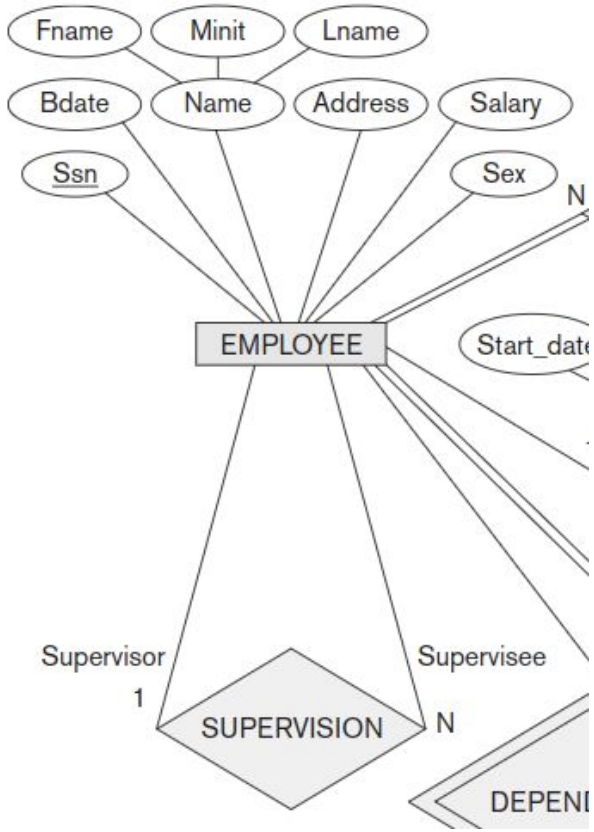

```sql
create table department_locations
(
    department_number int,
    department_name varchar,
    location varchar,
    foreign key (department_number, department_name) references department(number, name),
    primary key (location)
);
```

Total participation



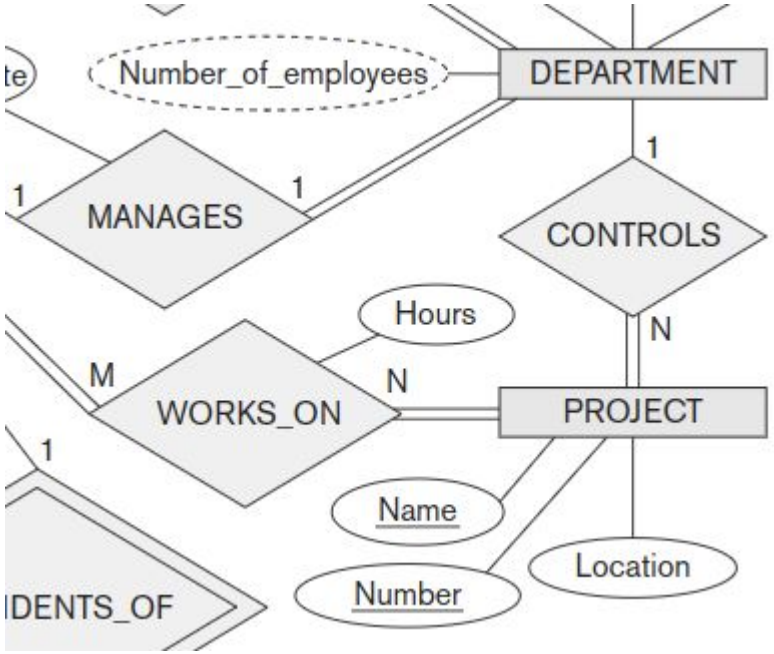

```sql
create table employee
(
    ssn     int primary key,
    bdate   date,
    fname   varchar,
    minit   varchar,
    lname   varchar,
    address varchar,
    salary  real,
    sex     varchar,
    department_number int,
    department_name varchar,
    foreign key (department_number,department_name) references department(number, name)
);
```

Recursive relationship




```sql
create table employee
(
    ssn     int primary key,
    bdate   date,
    fname   varchar,
    minit   varchar,
    lname   varchar,
    address varchar,
    salary  real,
    sex     varchar,
    department_number int,
    department_name varchar,
    foreign key (department_number,department_name) references department(number, name),
    super_ssn int references employee(ssn)
```

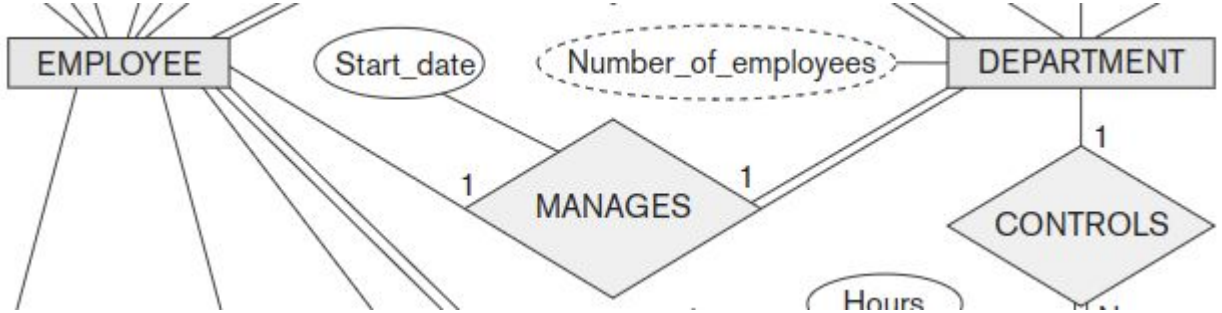
Many-to-many + more




```sql
create table project
(
    name     varchar,
    number   int,
    location varchar,
    department_number int,
    department_name varchar,
    foreign key (department_number, department_name) references department (number, name),
    primary key (name,number)
);

create table works_on
(
    employee_ssn   int references employee (ssn),
    project_name   varchar,
    project_number int,
    hours          int,
    foreign key (project_name, project_number) references project (name, number),
    primary key (employee_ssn, project_name, project_number)
);
```

One-to-one




```sql
create table department
(
    number int,
    name   varchar unique,
    manager_ssn int references employee(ssn),
    manager_start_date date,
    primary key (number, name)
);
```

Spot the problem

```
13  create table department
14  (
15      number int,
16      name   varchar unique,
17      manager_ssn int references employee(ssn),
18      manager_start_date date,
19      primary key (number, name)
20  );
22  create table employee
23  (
24      ssn              int primary key,
25      bdate            date,
26      fname            varchar,
27      minit            varchar,
28      lname            varchar,
29      address          varchar,
30      salary           real,
31      sex              varchar,
32      department_number int,
33      department_name   varchar,
34      foreign key (department_number, department_name) references department (number, name),
35      super_ssn        int references employee (ssn)
36  );
```

Spot the problem

```
13  create table department
14  (
```

```
[2022-03-07 11:57:56] [42P01] ERROR: relation "employee" does not exist
```

```
17      manager_ssn int references employee(ssn),
18      manager_start_date date,
19      primary key (number, name)
20  );
22  create table employee
23  (
24      ssn             int primary key,
25      bdate           date,
```

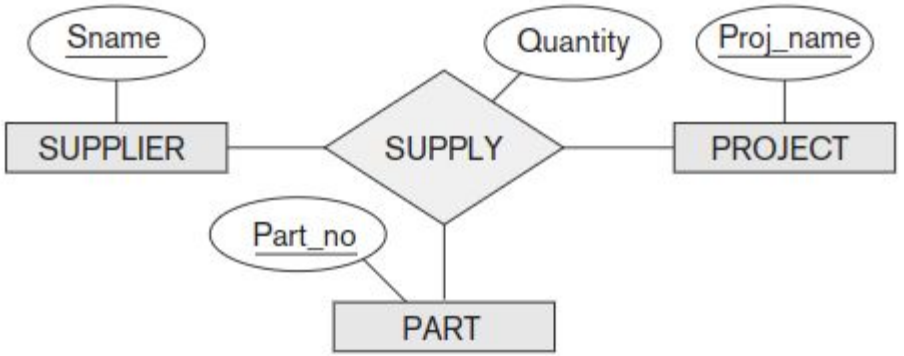
```
[2022-03-07 12:00:20] [42P01] ERROR: relation "department" does not exist
```

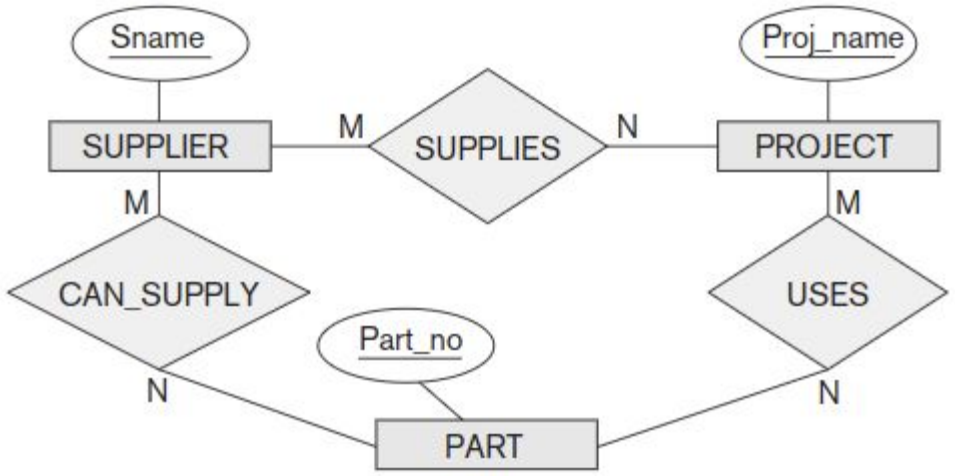
```
30      salary          real,
31      sex             varchar,
32      department_number int,
33      department_name   varchar,
34      foreign key (department_number, department_name) references department (number, name),
35      super_ssn       int references employee (ssn)
36  );
```

A solution

```
36 ✔ create table manages_department
37   (
38       manager_ssn       int unique references employee (ssn),
39       department_number int unique ,
40       department_name   varchar unique ,
41       foreign key (department_number, department_name) references department (number, name),
42       primary key (manager_ssn, department_number, department_name)
43   );
```

N-ary





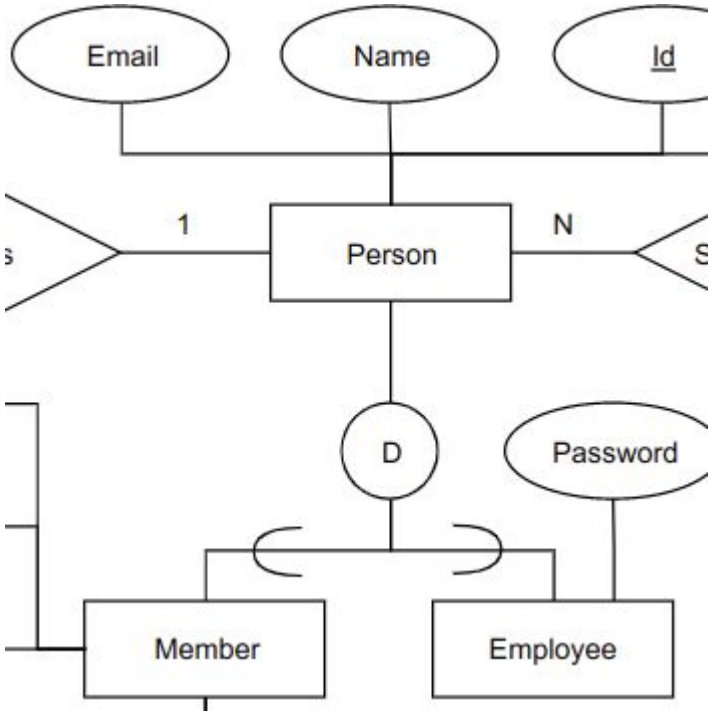

```sql
create table supplier
(
    name varchar primary key
);

create table project
(
    name varchar primary key
);

create table part
(
    number int primary key
);

create table supply
(
    supplier_name varchar references supplier(name),
    project_name varchar references project(name),
    part_number int references part(number),
    primary key (supplier_name,project_name,part_number),
    quantity int
);
```

Inheritance



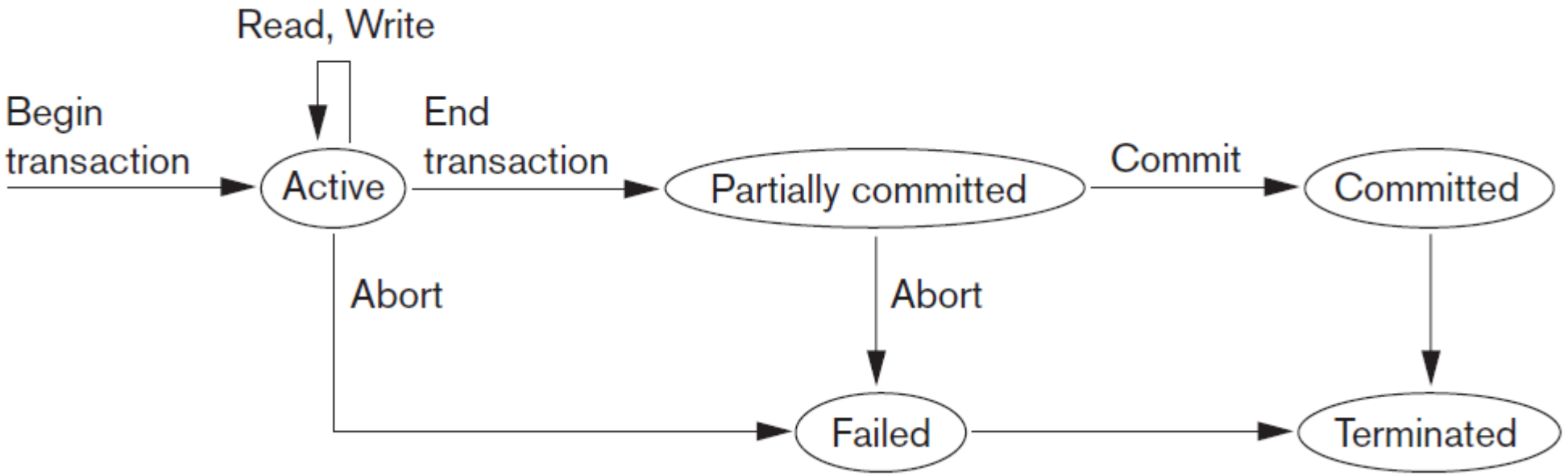

```
110 create table person(
111     id serial primary key,
112     name varchar,
113     email varchar
114 );
115
116 create table member(
117     person_id int references person(id) primary key
118 );
119
120 create table employee(
121         person_id int references person(id) primary key,
122         password varchar
123 );
```

Transactions

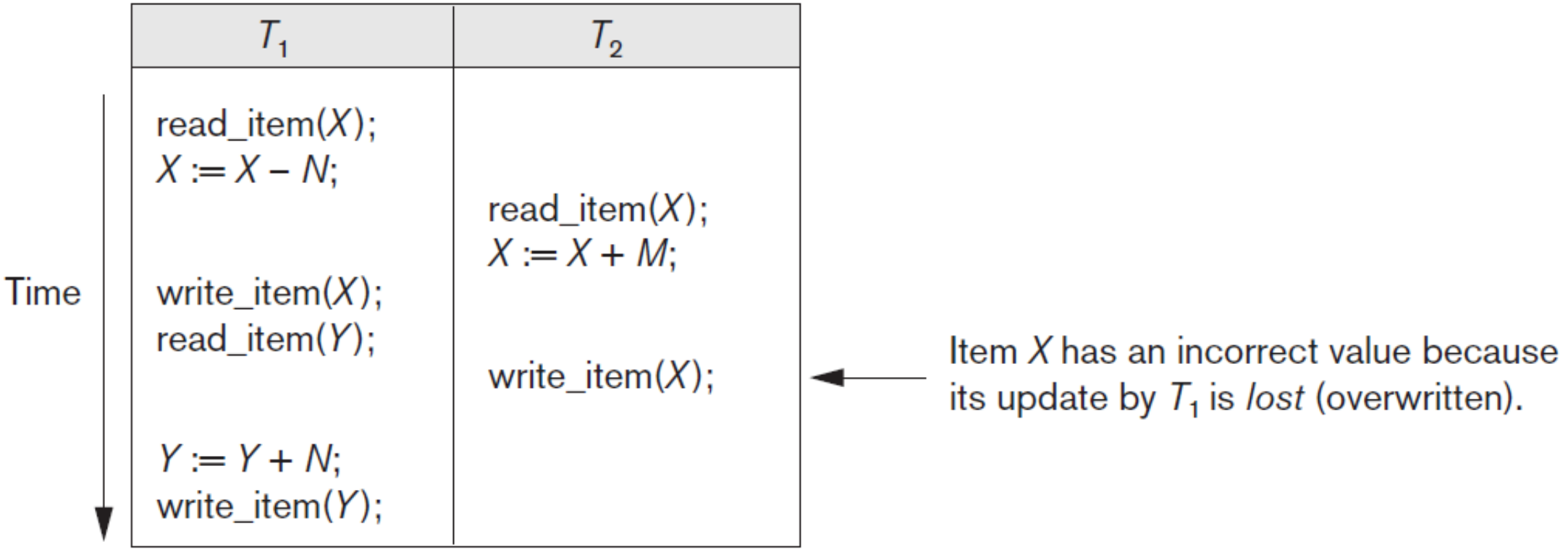
```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

- → What are transactions?
  - → An all or nothing operation
  - → A way to ensure consistency when multiple users are using a database system
  - → Entire transaction is executed, and not committed to the database before a COMMIT command is executed.

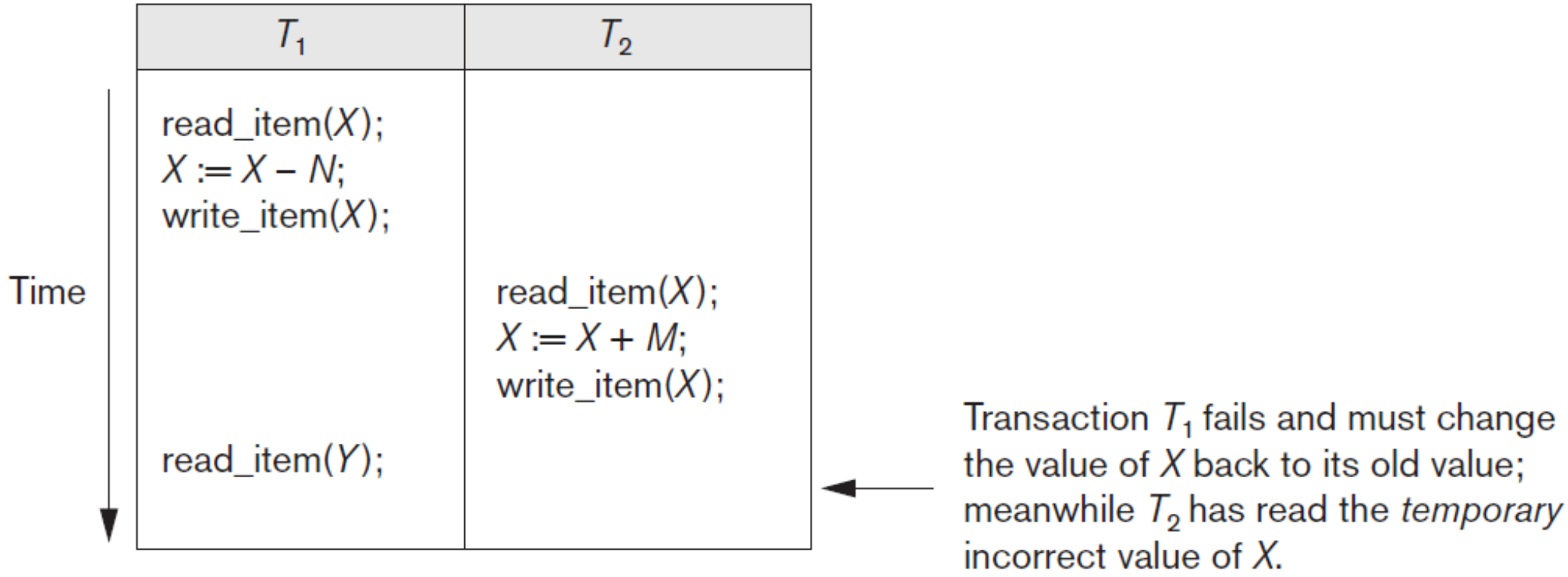
Commit Process



The Lost Update Problem



The Temporary Update (or Dirty Read) Problem



The Incorrect Summary Problem

T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	• • •

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

read_item(*Y*);
Y := *Y* + *N*;
write_item(*Y*);	

← T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Serial transaction

→ Typically the default.

→ Blocks other queries/transactions until one is finished.

(a)

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

(b)

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule B

Nonserial, and Conflict-Serializable Schedules

→ Combines transactions

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

Schedule C

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule D

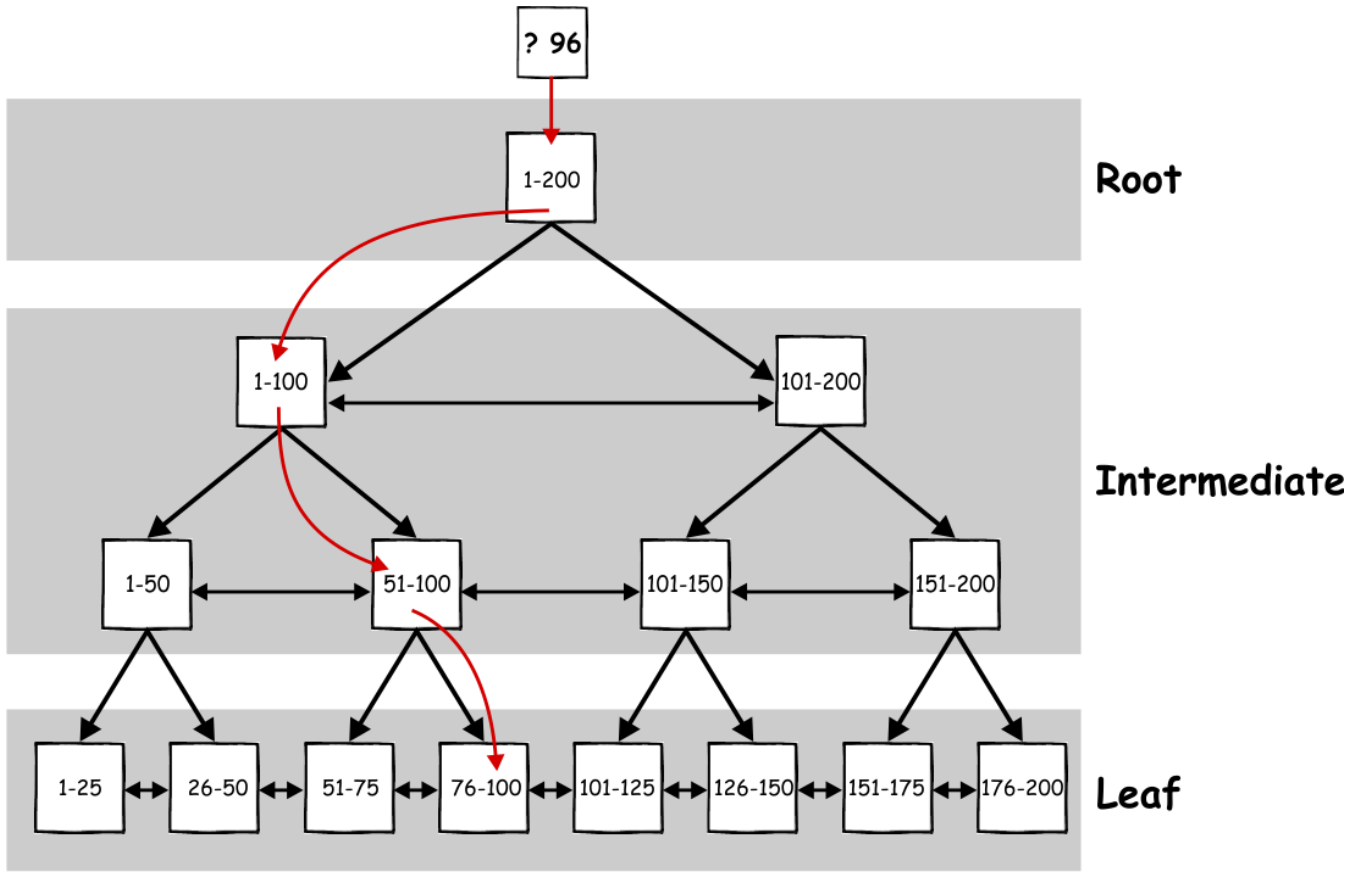
Transaction Commands

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

- → Commands:
  - → BEGIN
  - → SAVEPOINT
  - → ROLLBACK
  - → COMMIT
- → Watch out for not stalling the server with multiple BEGIN's that are never rolled back or committed.

Indexes

Indexes



- → What is an index?
  - → You already use it in programing
    - → Arrays, Maps, hash tables.
- → Used for performance optimization
- → Great when often finding records by something else than your primary key
- → Takes up more space on the disk

Creating Indexes

```
CREATE TABLE books(
    id serial PRIMARY KEY,
    title VARCHAR (250) NOT NULL,
    isbn VARCHAR (20) UNIQUE,
    price float
);

-- insert 2 million books

-- searching for book using ISBN instead of id - return time 2 min
SELECT * FROM books WHERE isbn = '978-12-92097-61-9';

-- slow result with that many entries

-- Creating
CREATE INDEX ON books(isbn);

-- now searching again - return time 5 seconds
SELECT * FROM books WHERE isbn = '978-12-92097-61-9';
```

Stored Procedures

Stored Procedures
(and Functions)

```
//Function PSM1:
0) CREATE FUNCTION Dept_size(IN deptno INTEGER)
1) RETURNS VARCHAR [7]
2) DECLARE No_of_emps INTEGER ;
3) SELECT COUNT(*) INTO No_of_emps
4) FROM EMPLOYEE WHERE Dno = deptno ;
5) IF No_of_emps > 100 THEN RETURN "HUGE"
6) ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
7) ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
8) ELSE RETURN "SMALL"
9) END IF ;
```

- → Several Languages can be used, depending on what database is used.
  - → Examples include: Java, Python, C# etc.
- → Most support SQL/PSM
  - → Completely portable, high-performance transaction-processing language.
  - → Generally supported directly inside SQL
- → The main functional difference between a function and a stored procedure is that a function returns a result, whereas a stored procedure does not.
- → All examples used after this slide are in PL/pgSQL, as we are using PostgreSQL

IF Structure

```
IF <condition> THEN <statement list>
    ELSEIF <condition> THEN <statement list>
    ...
    ELSEIF <condition> THEN <statement list>
    ELSE <statement list>
END IF ;
```

WHILE Structure

```
WHILE <condition> DO
        <statement list>
END WHILE ;
```

```
REPEAT
        <statement list>
UNTIL <condition>
END REPEAT ;
```

FOR Structure

```
FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
        <statement list>
END FOR ;
```

Stored Procedure Example 1/2

```
CREATE OR REPLACE PROCEDURE update_department_size(department_number INTEGER)
AS $$
DECLARE
    number_of_department_members integer := 0;
BEGIN
    SELECT COUNT(*) INTO number_of_department_members
    FROM department_members WHERE department_id = department_number;

    UPDATE departments SET number_of_members = number_of_department_members
    WHERE id = department_number;
END; $$
LANGUAGE plpgsql;
```

Stored Procedure Example 2/2

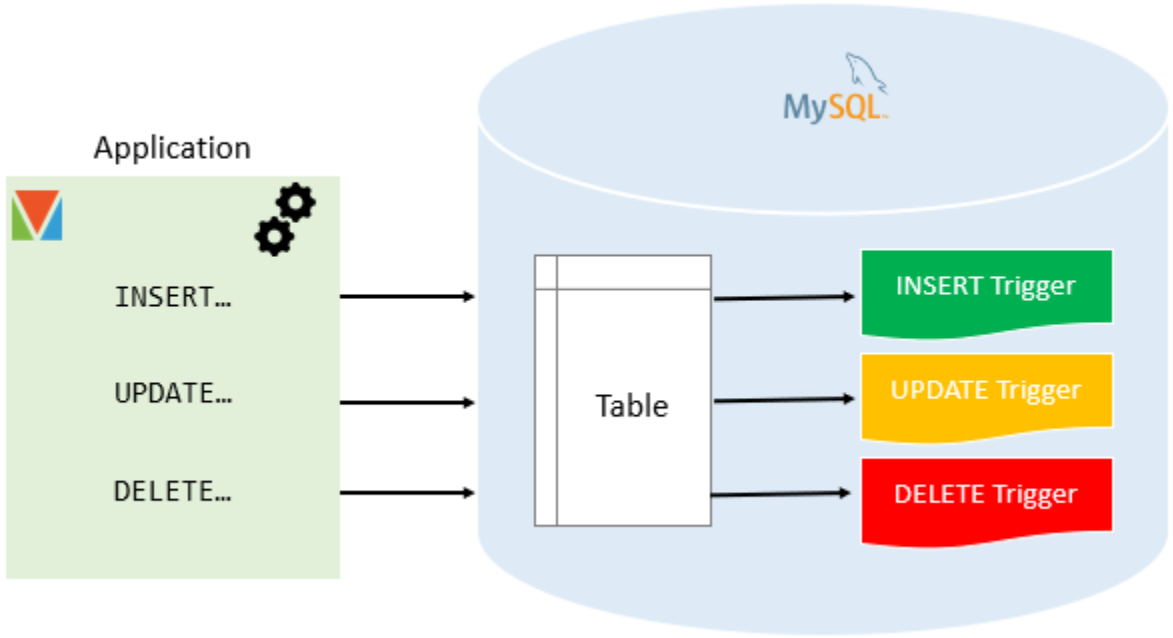
```
CREATE OR REPLACE PROCEDURE update_all_department_sizes()
AS $$
DECLARE
    departments CURSOR FOR SELECT DISTINCT(id) AS id FROM departments;
BEGIN
    FOR department in departments LOOP
        CALL update_department_size( department_number: department.id);
    END LOOP;
END; $$
LANGUAGE plpgsql;
```

Functions & Triggers

Function Example

```
CREATE OR REPLACE FUNCTION update_all_department_sizes_trigger()
    RETURNS TRIGGER
AS $$
BEGIN
    CALL update_all_department_sizes();
    RETURN NEW;
END; $$
LANGUAGE plpgsql;
```

Triggers



→ A way to execute functions or stored procedures if a change is happening in the database

Trigger Example 1

```
CREATE OR REPLACE FUNCTION update_all_department_sizes_trigger()
    RETURNS TRIGGER
AS $$
BEGIN
    CALL update_all_department_sizes();
    RETURN NEW;
END; $$
LANGUAGE plpgsql;
```

```
CREATE TRIGGER update_number_of_members_trigger
    AFTER INSERT OR DELETE ON department_members
    EXECUTE PROCEDURE update_all_department_sizes_trigger();
```

Trigger Examples 2

```
CREATE OR REPLACE FUNCTION log_last_name_changes()
  RETURNS trigger AS
$BODY$
BEGIN
   IF NEW.last_name <> OLD.last_name THEN
       INSERT INTO employee_audits(employee_id,last_name,changed_on)
       VALUES(OLD.id,OLD.last_name,now());
   END IF;

   RETURN NEW;
END;
$BODY$
```

```
CREATE TRIGGER last_name_changes
  BEFORE UPDATE
  ON employees
  FOR EACH ROW
  EXECUTE PROCEDURE log_last_name_changes();
```

Triggered by:

```
INSERT INTO employees (first_name, last_name)
VALUES ('John', 'Doe');

INSERT INTO employees (first_name, last_name)
VALUES ('Lily', 'Bush');
```

Types of triggers

When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
AFTER	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
	TRUNCATE	—	—

Trigger types example 1/2

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE PROCEDURE check_account_update();
```

- → A: Execute the function check_account_update whenever a row of the table accounts is about to be updated
- → B: The same, but only execute the function if column balance is specified as a target in the UPDATE command
- → C: This form only executes the function if column balance has in fact changed value

Trigger types example 2/2

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE PROCEDURE log_account_update();
```

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE PROCEDURE view_insert_row();
```

→ A: Call a function to log updates of accounts, but only if something changed.

→ B: Execute the function view_insert_row for each row to insert rows into the tables underlying a view

Data Model	Example Databases
Key-Value ("Key-Value Databases," p. 81)	BerkeleyDB
	LevelDB
	Memcached
	Project Voldemort
	Redis
	Riak
Document ("Document Databases," p. 89)	CouchDB
	MongoDB
	OrientDB
	RavenDB
	Terrastore
Column-Family ("Column-Family Stores," p. 99)	Amazon SimpleDB
	Cassandra
	HBase
	Hypertable
Graph ("Graph Databases," p. 111)	FlockDB
	HyperGraphDB
	Infinite Graph
	Neo4J
	OrientDB

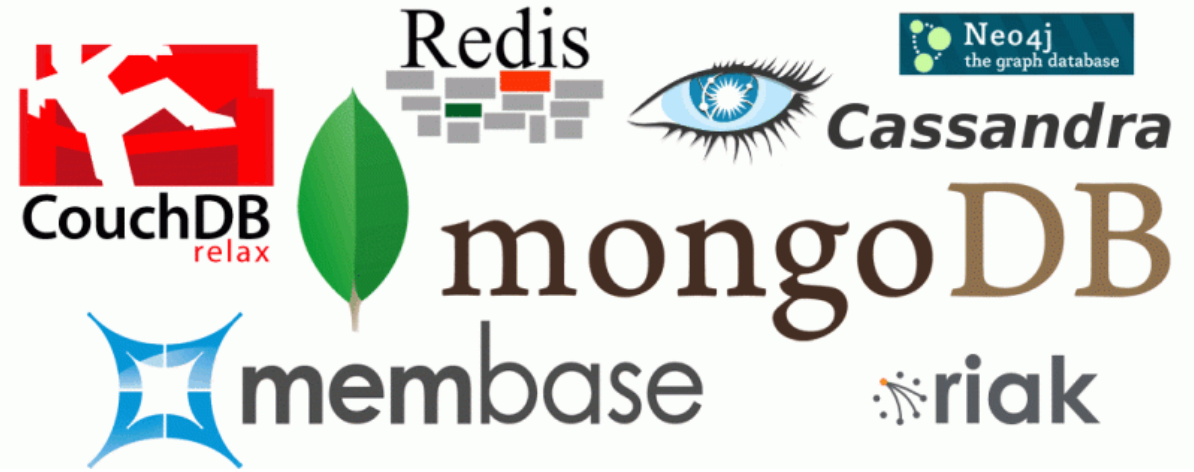
Source: NoSQL Distilled, by P. Sadalage and M. Fowler

NoSQL Databases

→ NoSQL means **N**ot **O**nly **SQL**

→ NoSQL covers multiple types of databases

Why NoSQL?



- → Fits well to many data types and application areas
- → Dynamic Schema! (Bad and Good)
- → Easy to scale
  - → Data Size increases (Big Data)
- → Often easy to replicate
- → High Performance
- → Less powerful query languages
- → Versioning

A comment on database sizes (Relational DBs)

		id	Symbol ▲ 1	Price	Timestamp
☐	Ret Kopi Slet	16777216	BTCUSDT	23494.4	1677554256
☐	Ret Kopi Slet	33554432	BTCUSDT	27888	1679411028
☐	Ret Kopi Slet	50331648	BTCUSDT	29282.3	1682851218
☐	Ret Kopi Slet	65536	BTCUSDT	20976.3	0
☐	Ret Kopi Slet	16842752	BTCUSDT	23407.1	1677560723
☐	Ret Kopi Slet	33619968	BTCUSDT	28347.6	1679418043
☐	Ret Kopi Slet	50397184	BTCUSDT	29203.9	1682858445
☐	Ret Kopi Slet	131072	BTCUSDT	21120.8	0
☐	Ret Kopi Slet	16908288	BTCUSDT	23393.2	1677567301
☐	Ret Kopi Slet	33685504	BTCUSDT	28070	1679424984
☐	Ret Kopi Slet	50462720	BTCUSDT	29255	1682865889

Viser rækkerne 0 - 24 (5977768 i alt, Forespørgsel tog 24.4428 sekunder.)

About 6.000.000 rows

A comment on database sizes (Relational DBs)

✔ MySQL returnerede ingen data (fx ingen rækker). (Forespørgsel tog 18.7534 sekunder.)

```
SELECT * from price_logs WHERE Symbol LIKE "BTCUSD*" LIMIT 20;
```

Takes 18,7 seconds to run

✔ MySQL returnerede ingen data (fx ingen rækker). (Forespørgsel tog 87.5783 sekunder.)

```
CREATE INDEX symbol_index on price_logs(Symbol);
```

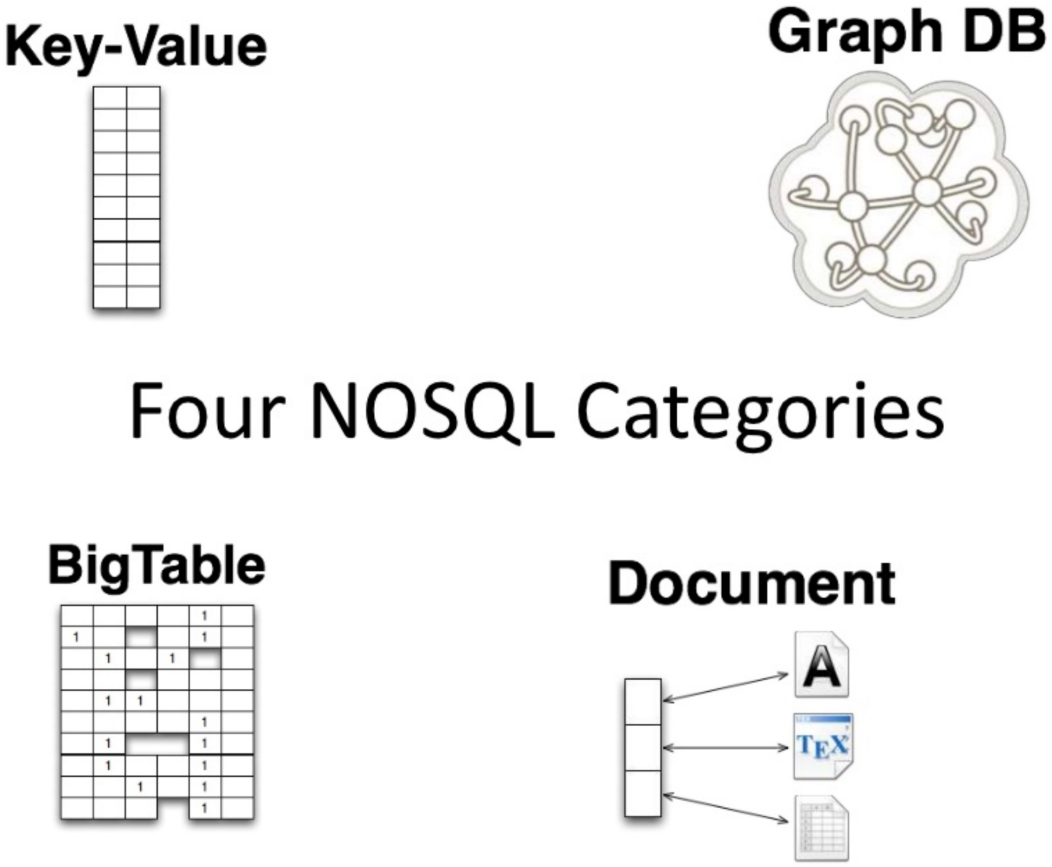
Create an index to optimize the 'like' clause, which takes 87,5783 seconds to create in memory.

✔ MySQL returnerede ingen data (fx ingen rækker). (Forespørgsel tog 0.0002 sekunder.)

```
SELECT * from price_logs WHERE Symbol LIKE "BTCUSD*" LIMIT 20;
```

After optimization, the query takes 0,0002 seconds to run (or 0.2 milliseconds)

What types exist?



→ Document-based NoSQL systems

→ NoSQL key-value stores

→ Column based or wide column-based NoSQL systems

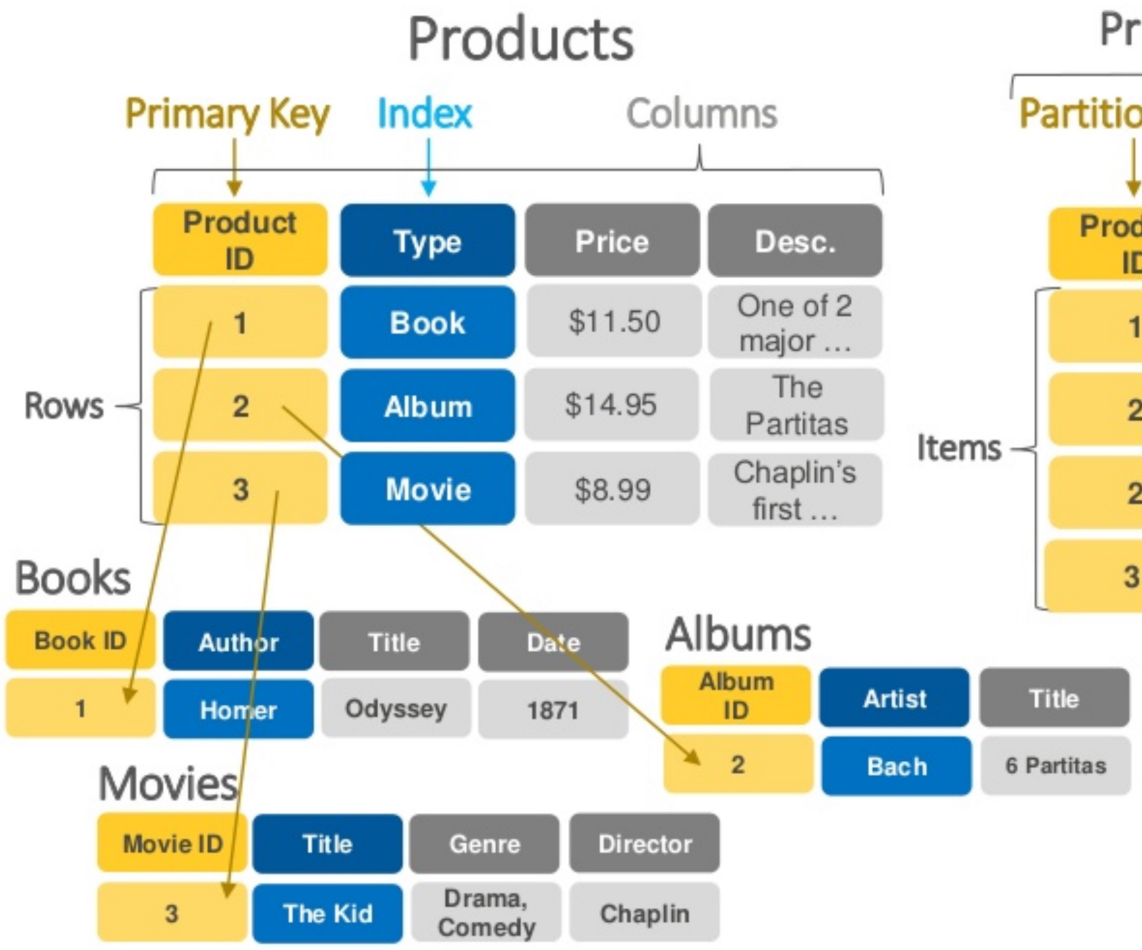
→ Graph based NoSQL systems

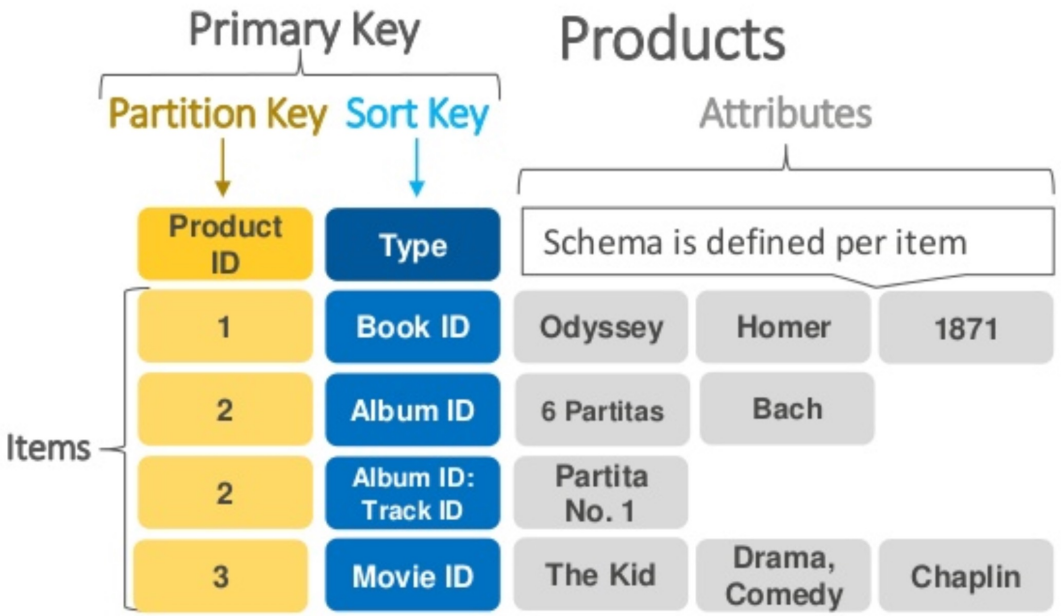
→ Hybrid NoSQL systems

  → Object databases

  → XML databases

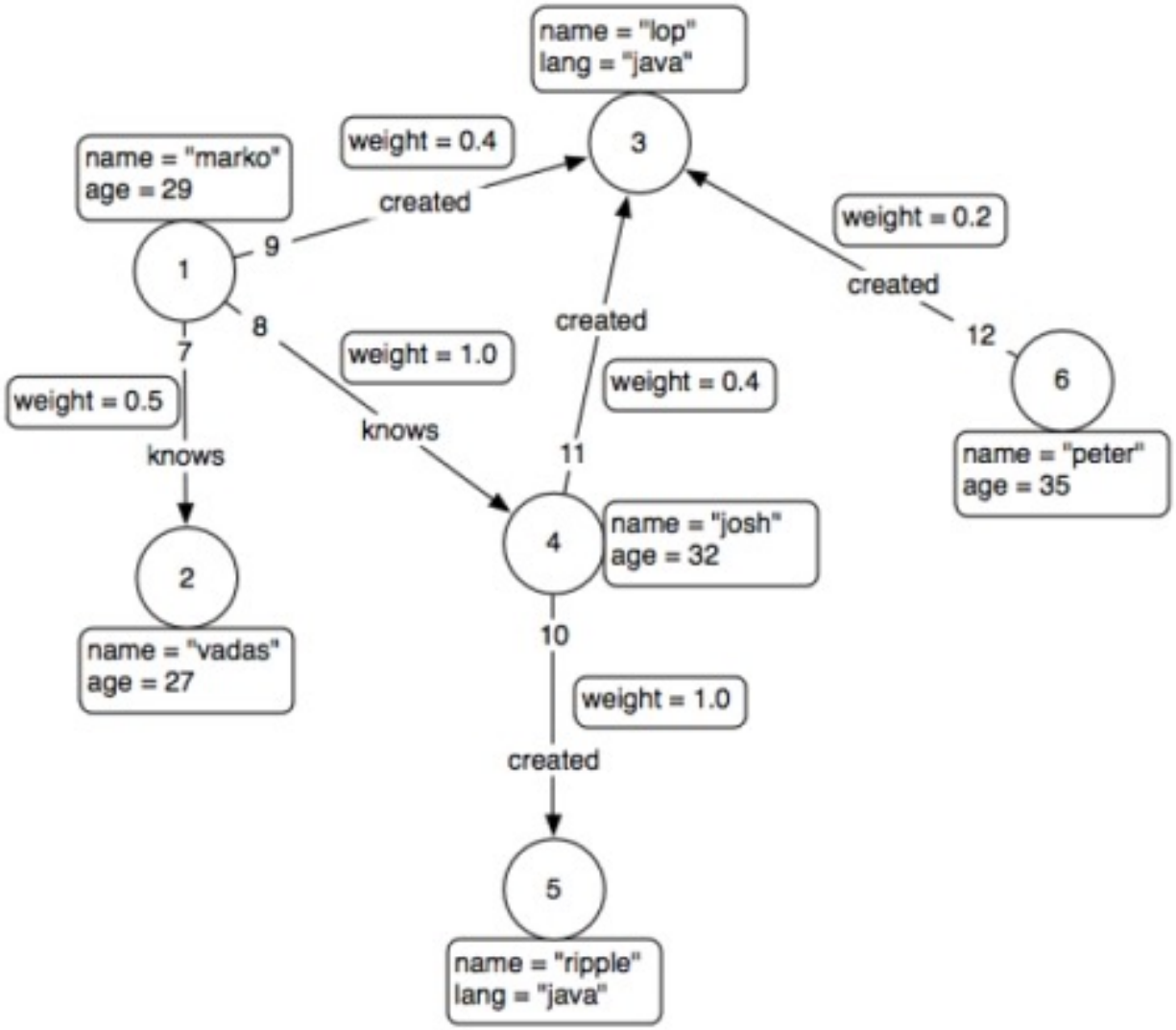
Relational vs. Key Value



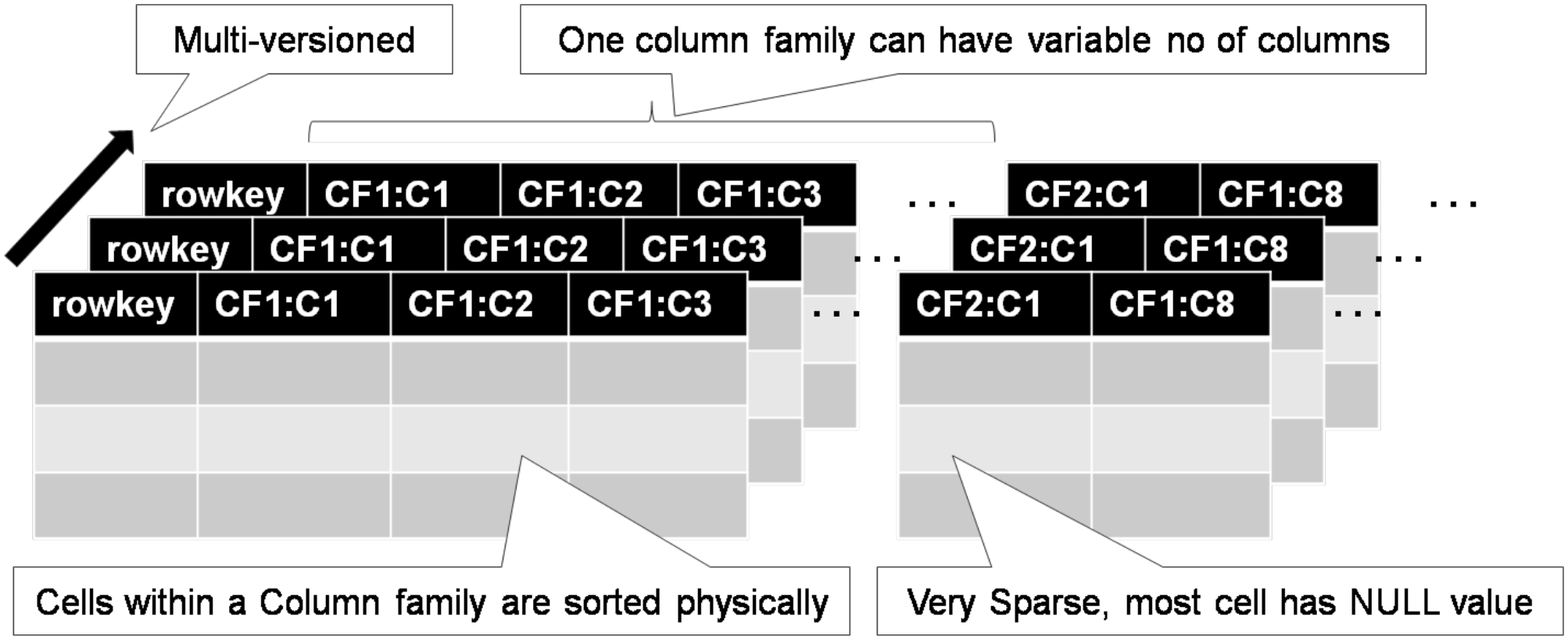


- Implemented in different ways, but all key/value based
- DynamoDB Example (in diagram above)
  - No sequences
  - Data found based on partition key and sort key

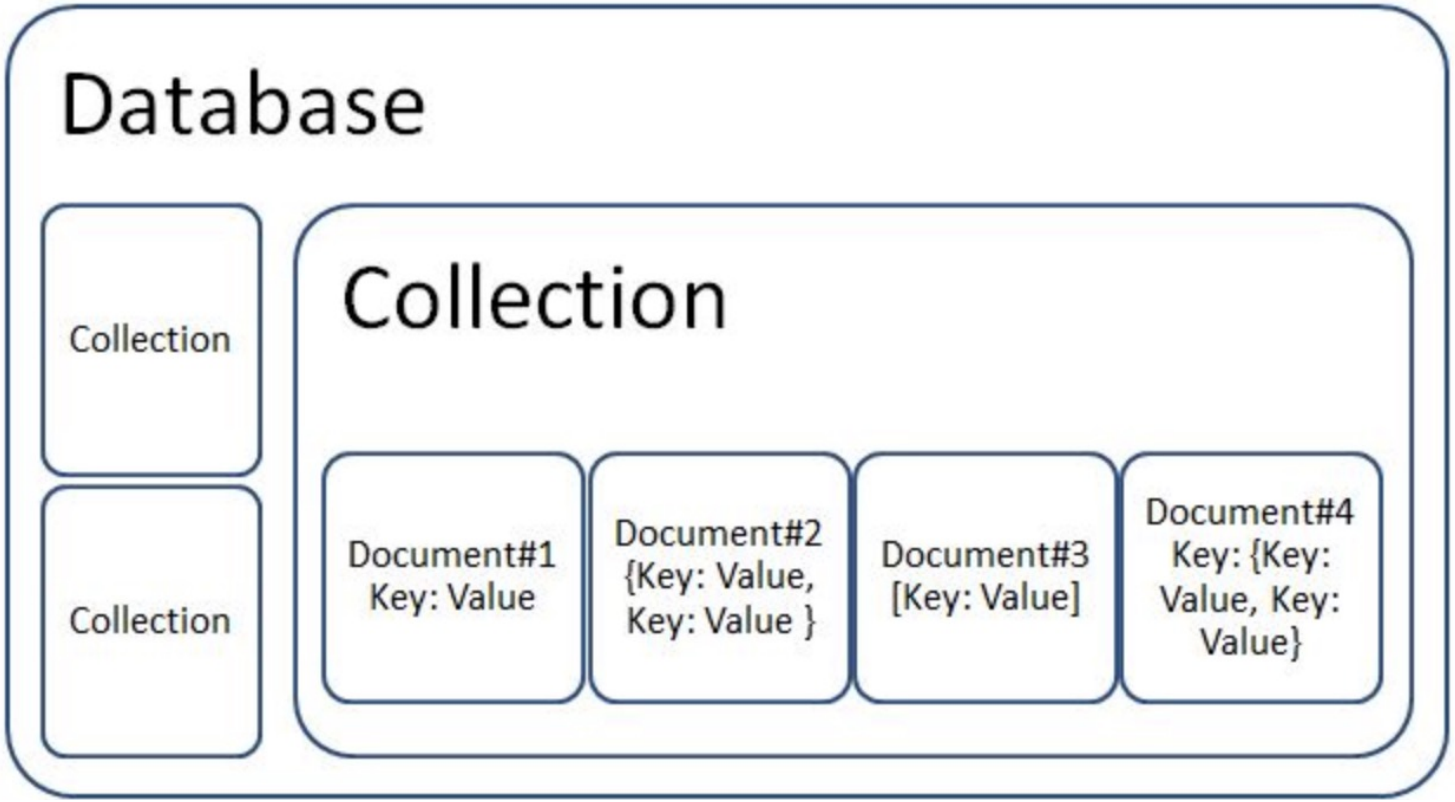
Graph Based


name = "lop"
lang = "java"
3
weight = 0.4
name = "marko"
age = 29
created
weight = 0.2
9
created
1
created
12
8
6
7
weight = 1.0
weight = 0.4
weight = 0.5
knows
knows
11
name = "peter"
age = 35
4
name = "josh"
age = 32
2
10
name = "vadas"
age = 27
weight = 1.0
created
5
name = "ripple"
lang = "java"

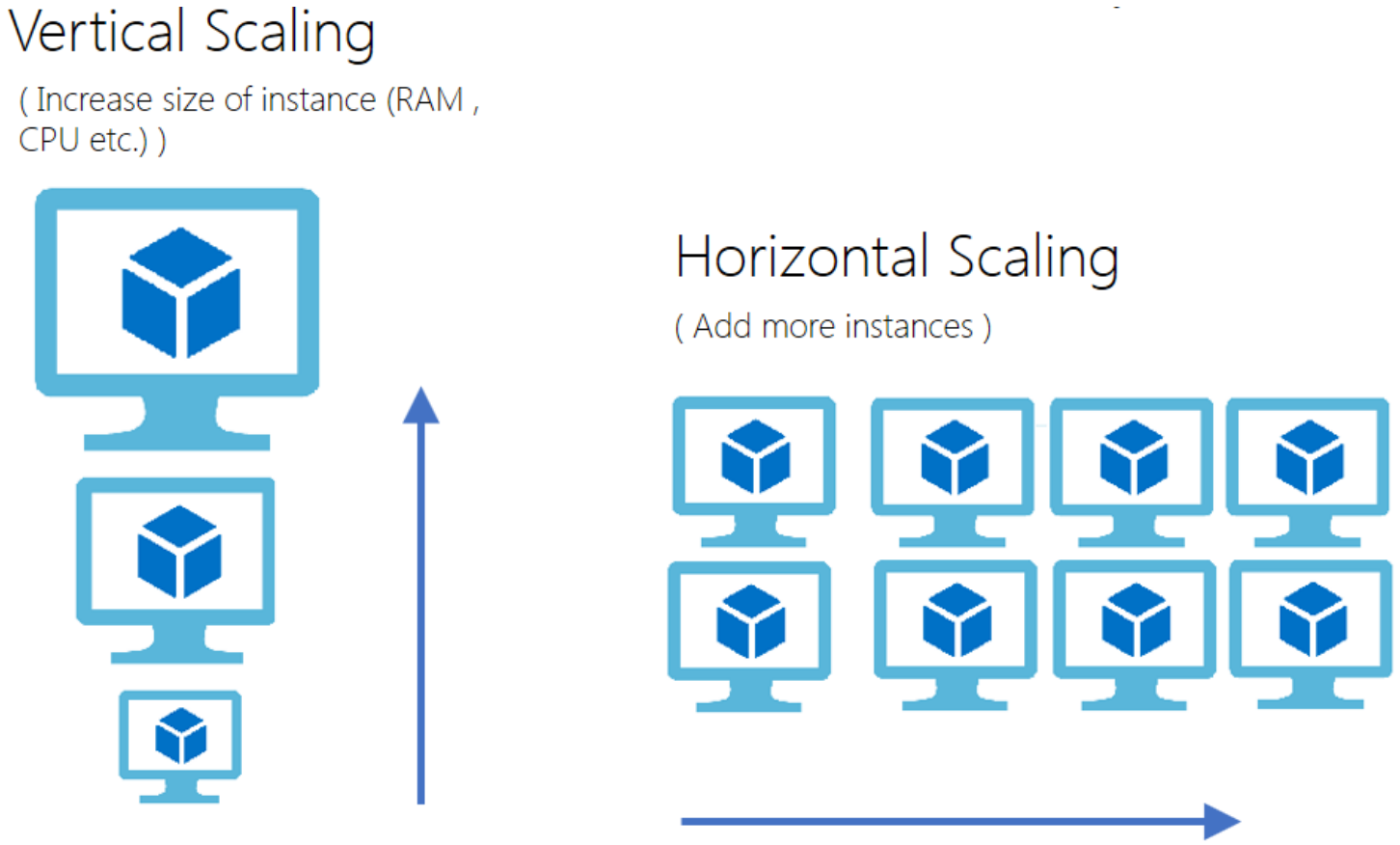
Big Table


Multi-versioned
One column family can have variable no of columns
rowkey
CF1:C1
CF1:C2
CF1:C3
CF2:C1
CF1:C8
Cells within a Column family are sorted physically
Very Sparse, most cell has NULL value

Document based



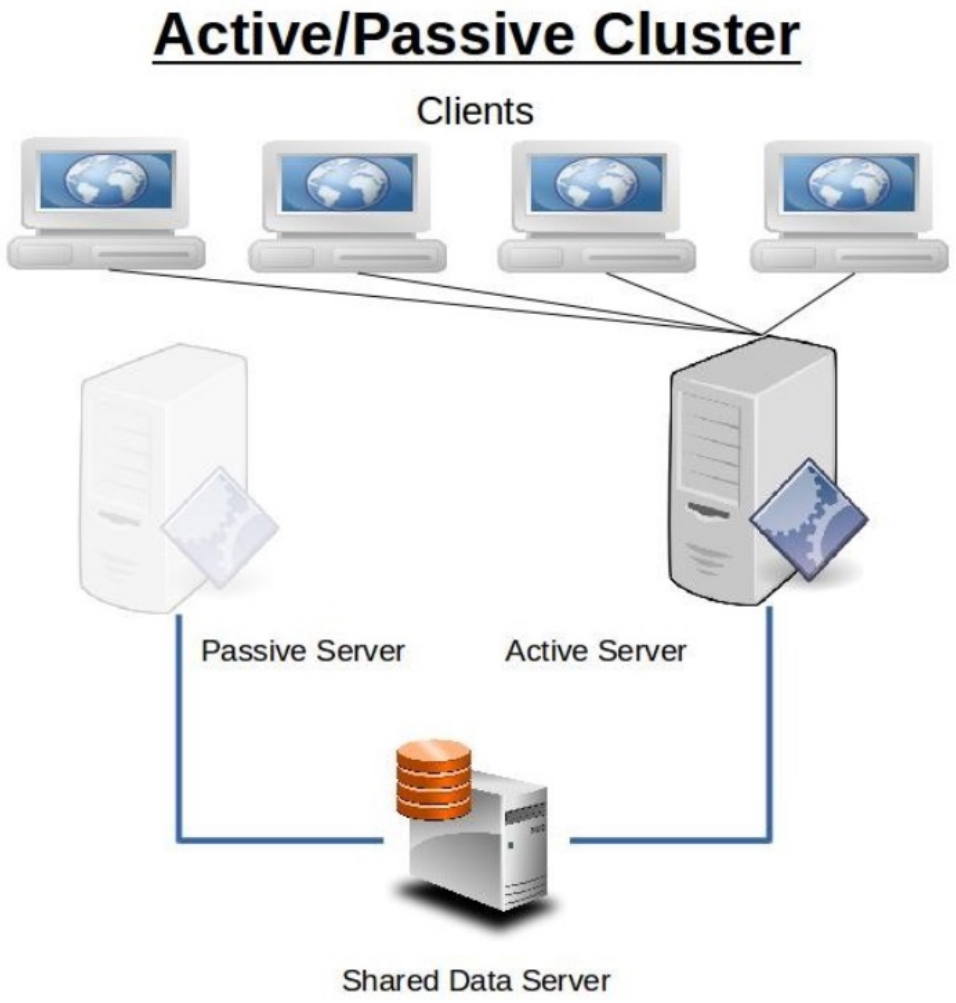
Horizontal vs Vertical Scaling

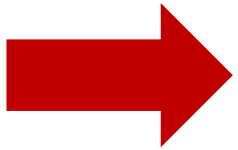


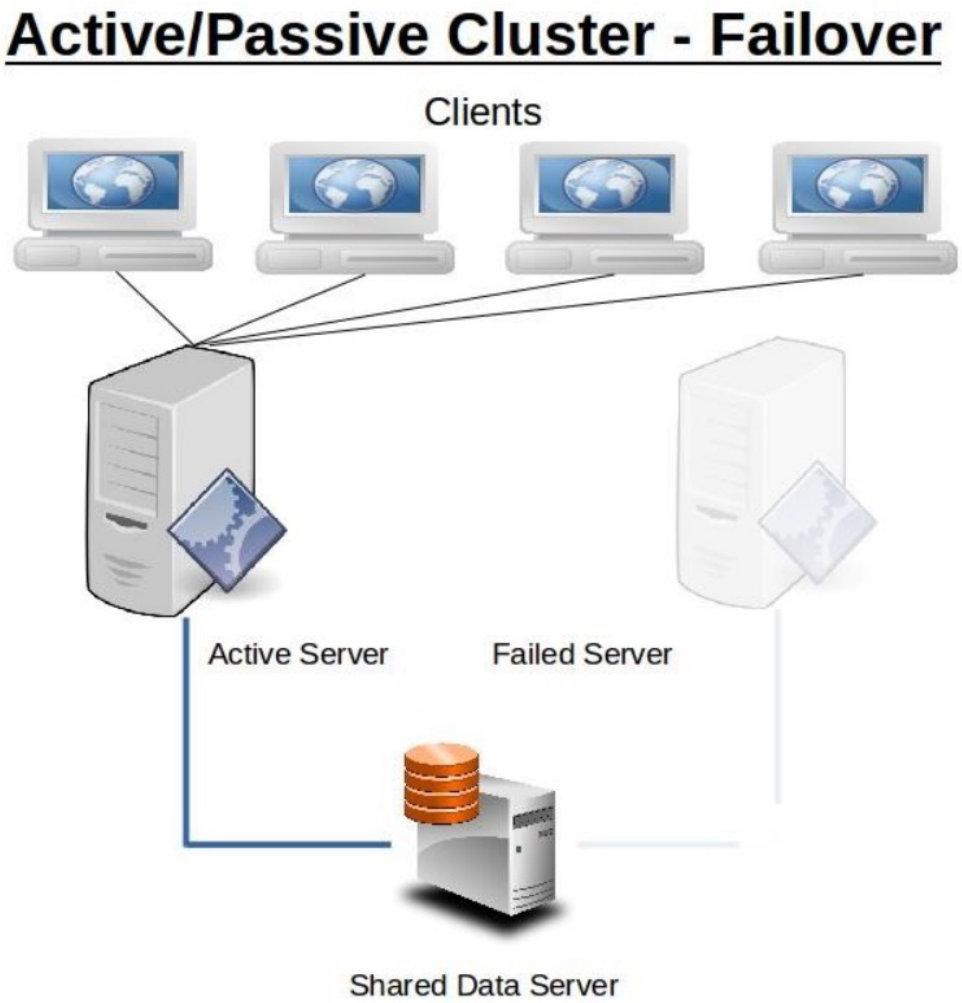
- → Vertical
  - → Increasingly Expensive to get larger hardware
  - → Requires no code changes
- → Horizontal
  - → Less expensive hardware
  - → Allows for large-scale systems
  - → Need to change software for a distributed architecture
    - → Leads to more complex code
  - → Need for Load Balancing

→ Do you know examples of these?

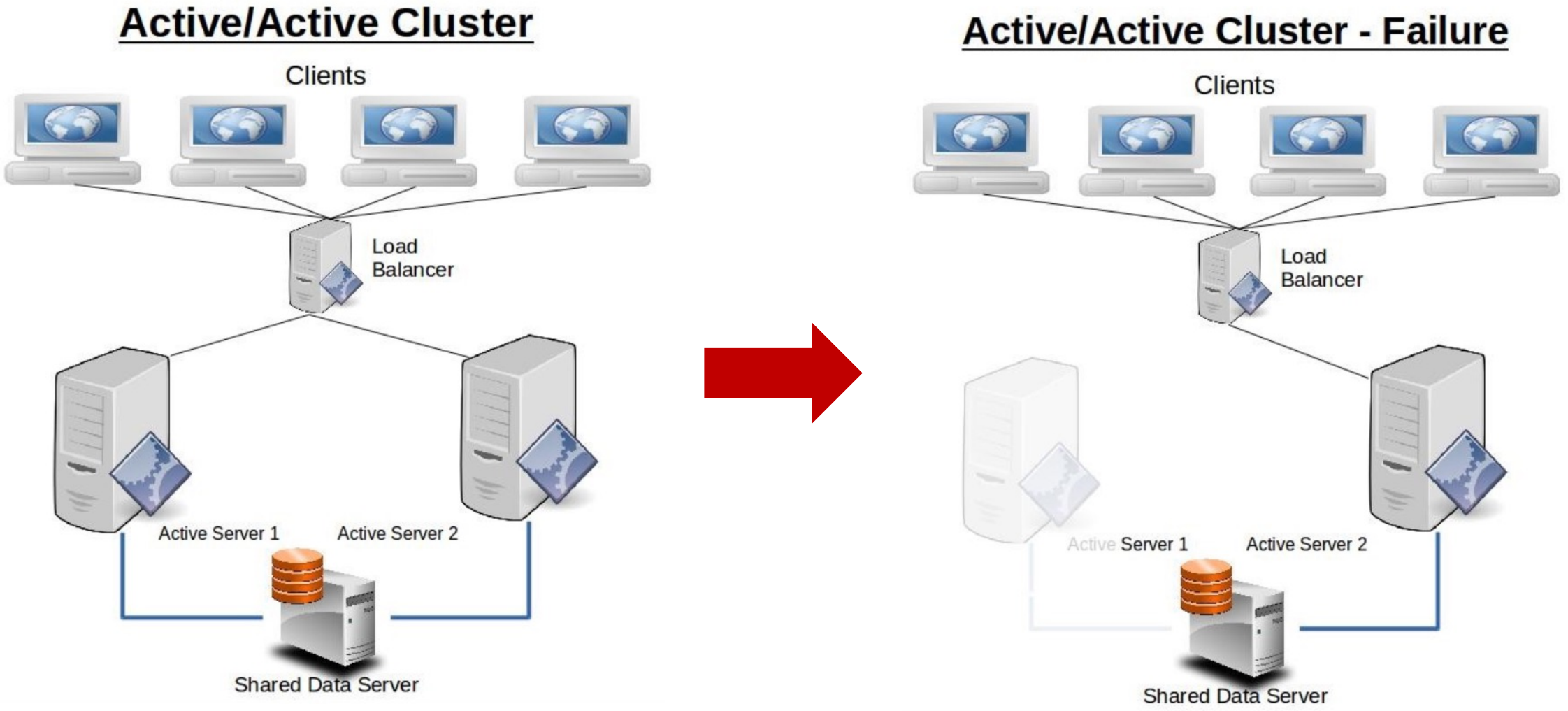
High Availability 1/3



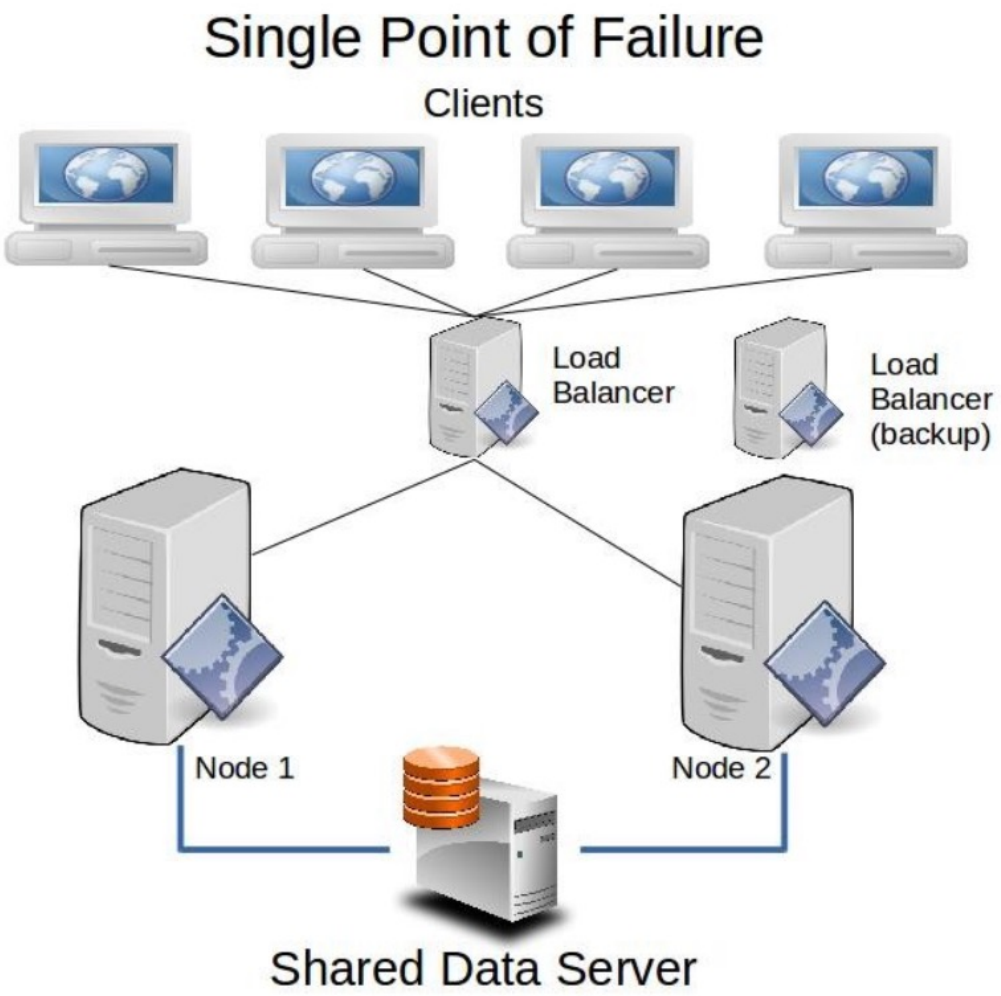


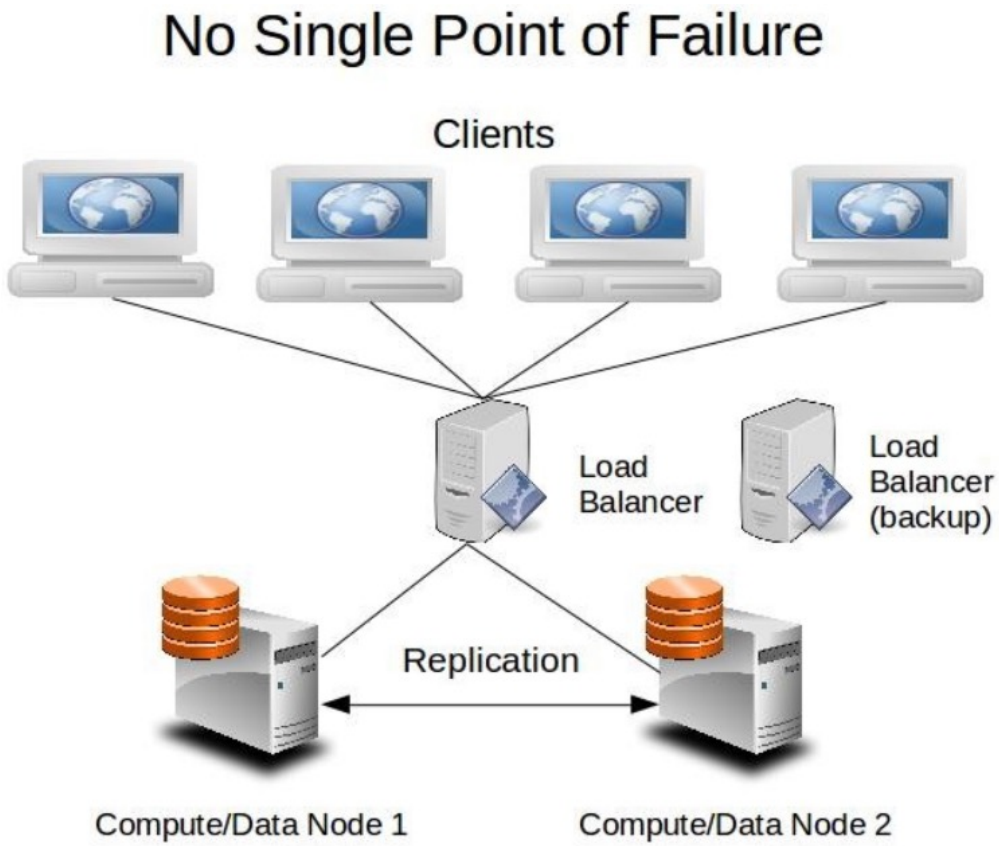


High Availability 2/3

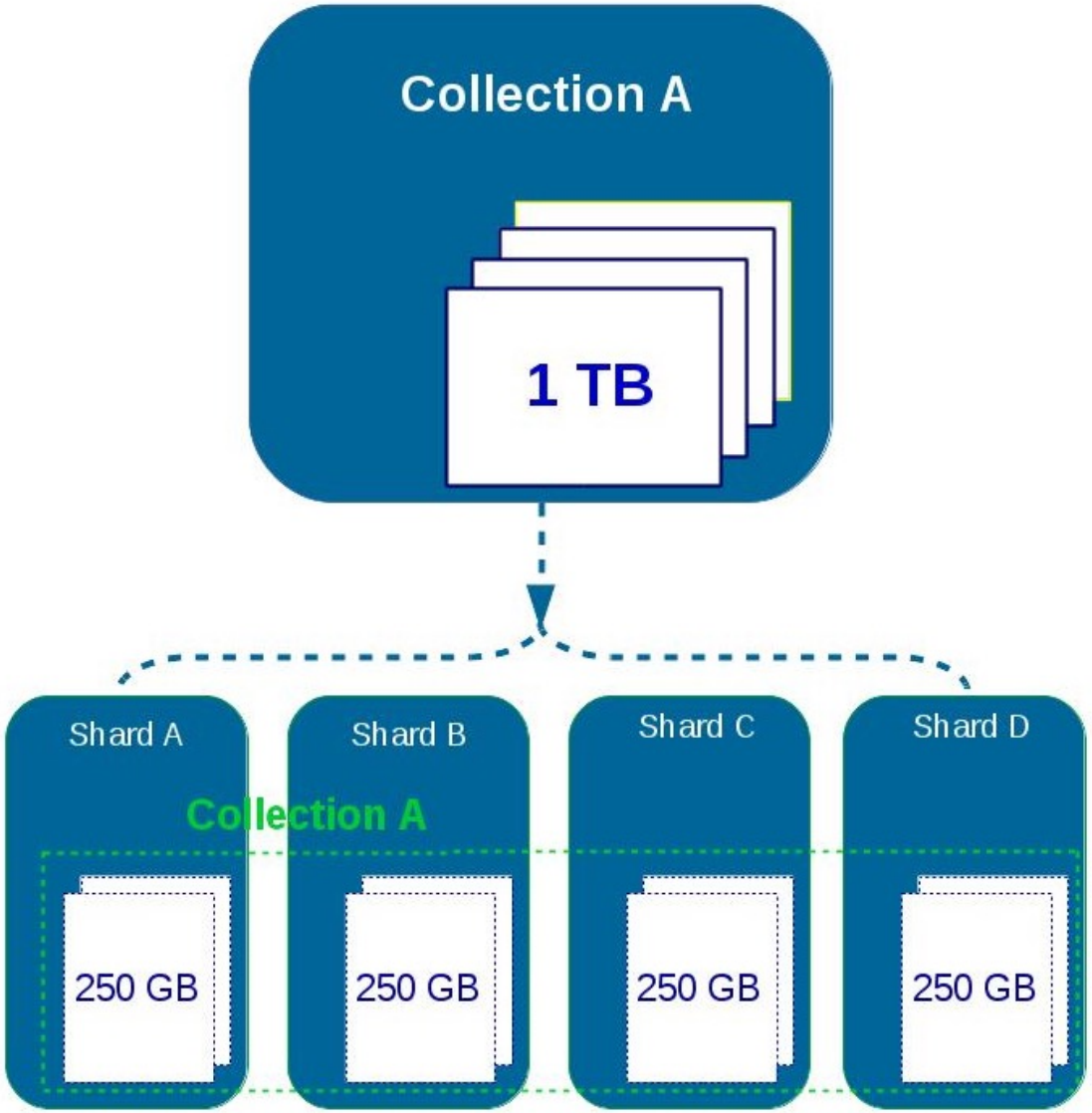


High Availability 3/3



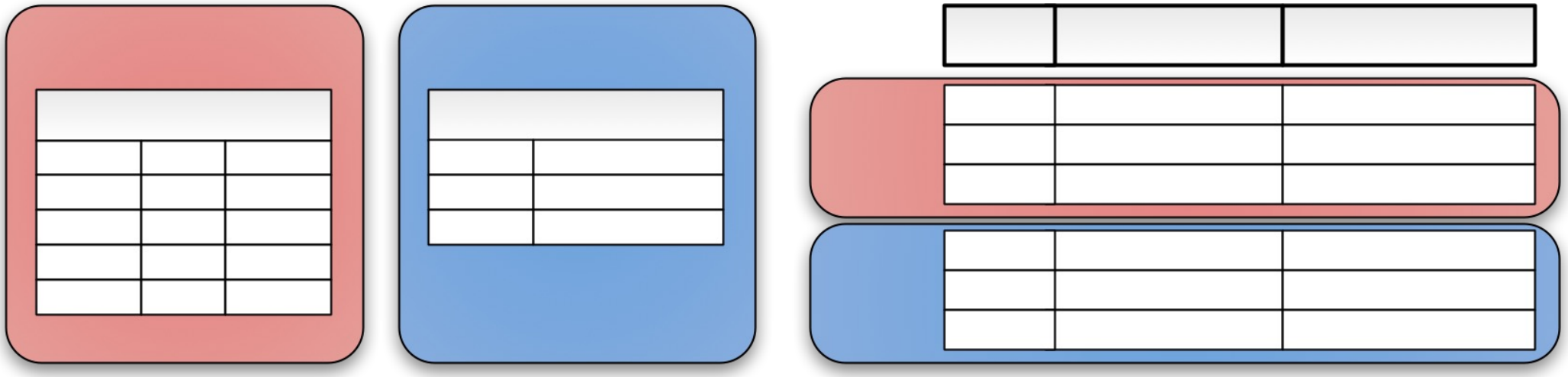


Partitioning (Sharding)

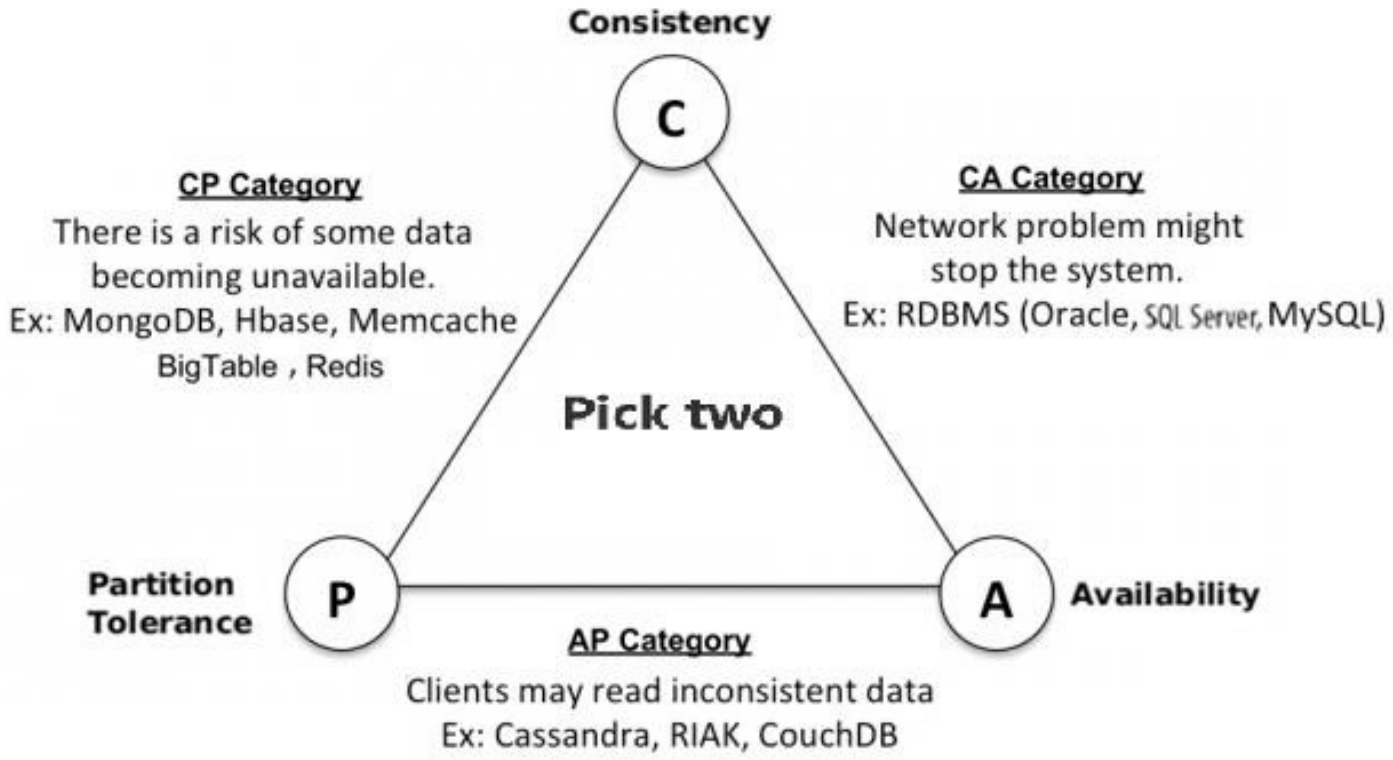


- → Splitting up the data between multiple servers
- → Together they represent the entire collection
- → Allows for better throughput as calculations and search are shared
- → Makes sense for Big Data Applications

Partitioning (Sharding)



The CAP Theorem – Distributed Databases

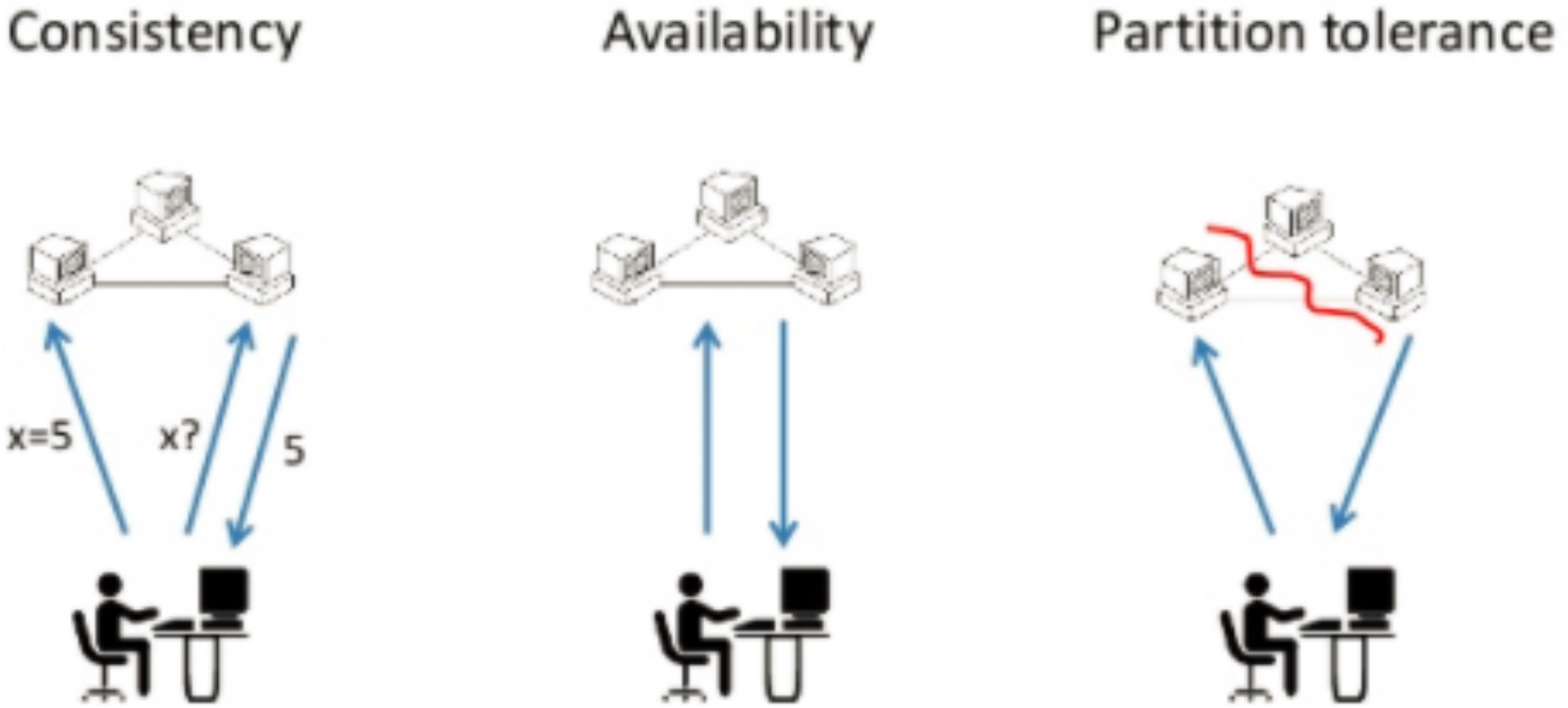


Consistency: Every read recieves the most recent write or an error. All nodes see the same data at all times.

Availability: System is always operational dispite invividual server status. Every Request recives a (non-error) repsonse, without the gurantee that it contains the most recent write

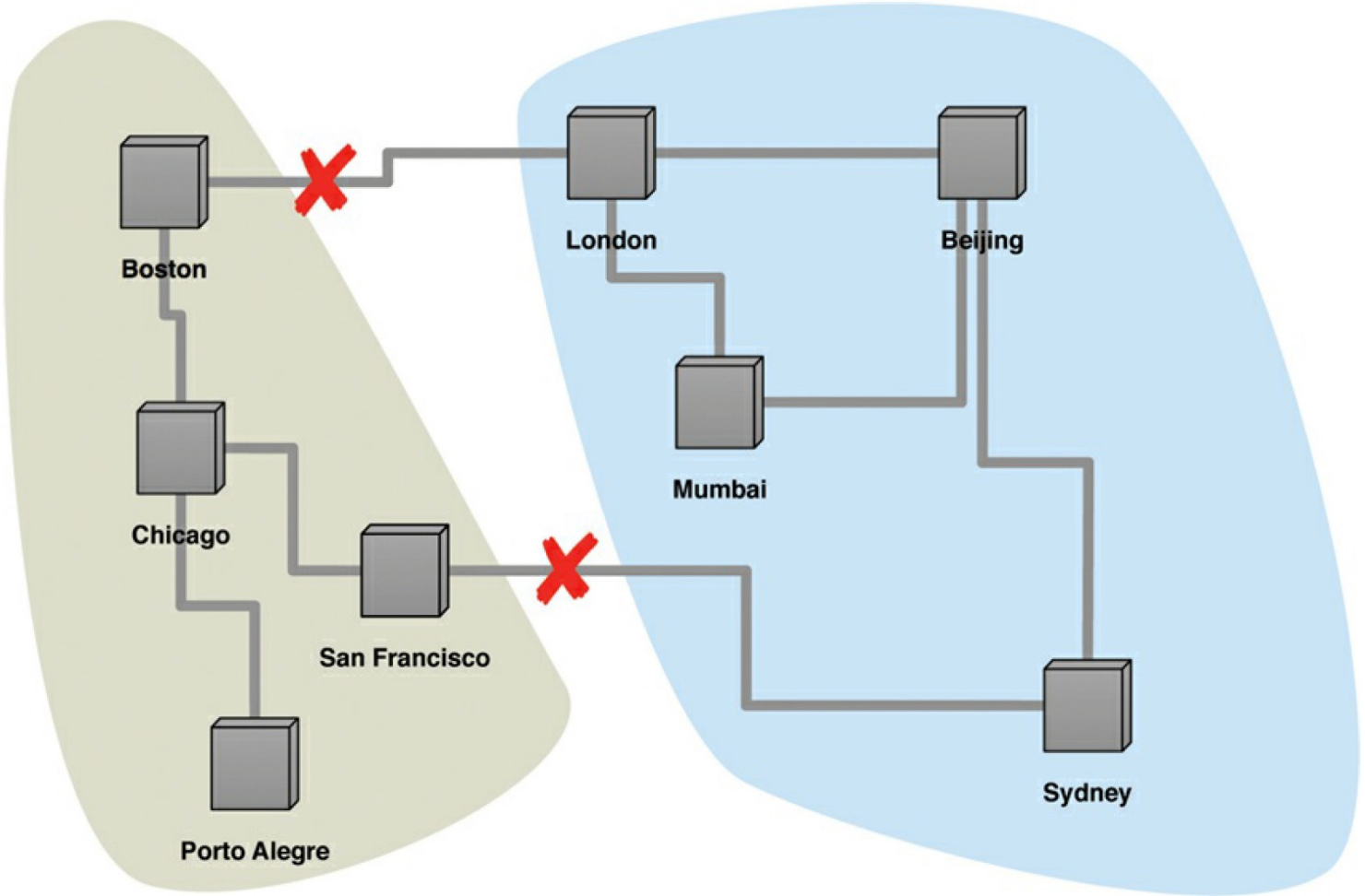
Partition Tolerance: The system continues to operate dispte of partition failure (a node dies with part of the data)

Confusing? Indeed - Read the FAQ: https://github.com/henryr/cap-faq



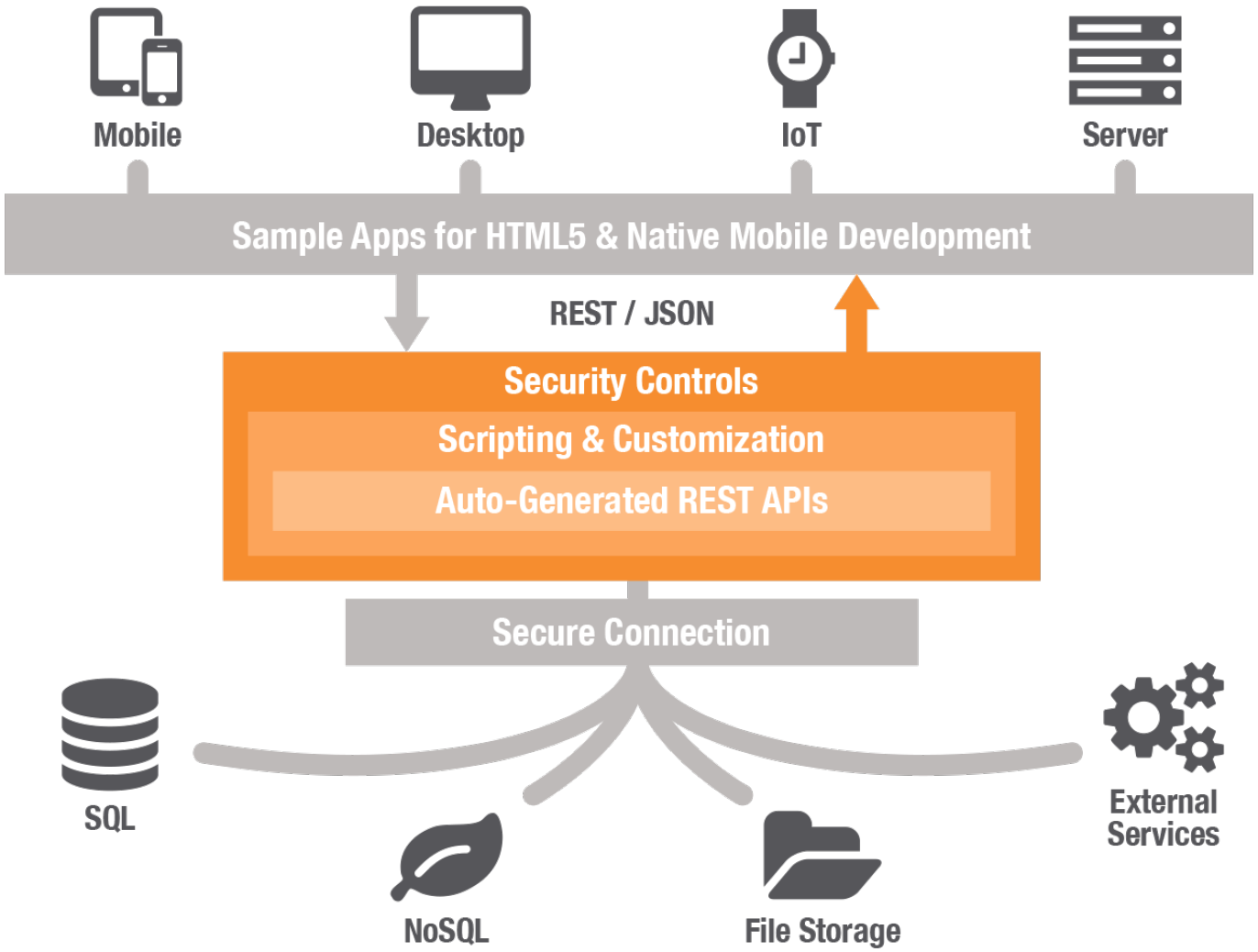
Source: https://toppertips-bx67a.ondigitalocean.app/cap-theorem/

A CAP Theorem example



What is JSON?

- JSON is stands for “JavaScript Object Notation”,
- It’s used to store and exchange data as an alternative solution for XML.
- Easy to parse, and easy to read and write for a human.
- Based on JavaScript Object Literals, but it’s a text format.
- JSON is language independent; means you can use parse and generate JSON data in other programming languages.



How is JSON used?

- → Websites communicating with the backend
- → Mobile applications communicating with the backend
- → Temporary storage format of objects
- → …

Object Declaration

```
{
    "id": 2456,
    "name": "John Doe",
    "cpr": "111111-1111",
    "married": true
}
```

- → An object starts with { and ends with }.
- → Inside is a collection of Key/Values separated by comma
  - → Be aware it is a syntax error to have a comma after the last key/value
- → Each key is a string, so it uses the double quotes - “key”
- → A : mark separates a key and a value
- → Each value can be of several types. In the example we have number, string, and boolean.

Array Declaration

```
[
    "Odense",
    "Aarhus",
    "San Francisco"
]
```

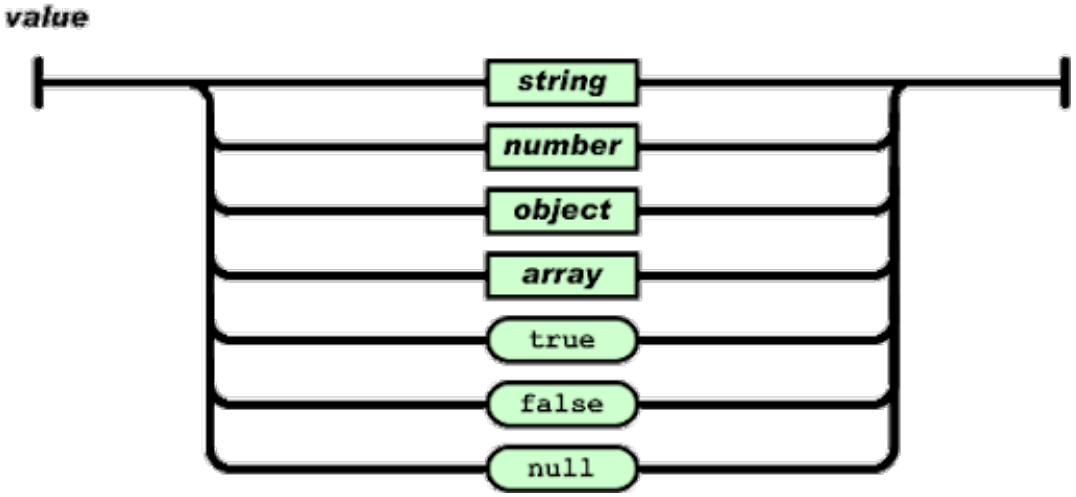
→ Arrays are ordered lists, and starts from 0

→ Initiated with [and ends with]

→ Values are separated by comma

→ An array can also hold objects

JSON Values



→ string; “string”
→ number; 1
→ boolean; true or false
→ null; null/left out
→ an object; {“key”: “value”}
→ an array; [1, “2”, null, true]

Putting it together

```
{
    "id": 2456,
    "name": "John Doe",
    "cpr": "111111-1111",
    "married": true,
    "relationships": [
        {
            "name": "Jane Doe",
            "relationshipType": "Wife"
        },
        {
            "name": "Danny Doe",
            "relationshipType": "Son"
        }
    ],
    "livedIn": ["Odense", "Aarhus"]
}
```

What is RegEx?

- → RegEx stands for Regular Expressions
- → A way to create a filter pattern
- → What if my data is not a database, but unstructured data?
- → Uses:
  - → Finding specific text in unstructured data
  - → Finding the right item
  - → Verify structure
  - → Do replacements
- → Practical Example:
  - → Validating input fields (websites, apps, etc.)
    - → Passwords, emails…
  - → Identifying spam websites (ScanNet)

RegEx in DM

→ Why is RegEx a part of Data Management?

→ Relates to getting data (just as we do from databases)

→ Valuable in relation to unstructured data

RegEx Example

/([A-Z])\w+/g

Text | Tests NEW | 29 matches (1.0ms)

RegExr was created by gskinner.com, and is proudly hosted by Media Temple.

Edit the Expression & Text to see matches. Roll over matches or the expression for details. PCRE & JavaScript flavors of RegEx are supported. Validate your expression with Tests mode.

The side bar includes a Cheatsheet, full Reference, and Help. You can also Save & Share with the Community, and view patterns you create or favorite in My Patterns.

Explore results with the Tools below. Replace & List output custom results. Details lists capture groups. Explain describes your expression in plain English.

See tool here: https://regexr.com – examples today will be in VS Code

Anchors

Anchors	
^	Matches at the start of string or start of line if multi-line mode is enabled. Many regex implementations have multi-line mode enabled by default.
$	Matches at the end of string or end of line if multi-line mode is enabled. Many regex implementations have multi-line mode enabled by default.
\A	Matches at the start of the search string.
\Z	Matches at the end of the search string, or before a newline at the end of the string.
\z	Matches at the end of the search string.
\b	Matches at word boundaries.
\B	Matches anywhere but word boundaries.

Character Classes

Character Classes *Can also be used in bracket expressions.*	
.	Matches any character except newline. Will also match newline if single-line mode is enabled.
\s	Matches white space characters.
\S	Matches anything but white space characters.
\d	Matches digits. Equivalent to [0-9].
\D	Matches anything but digits. Equivalent to [^0-9].
\w	Matches letters, digits and underscores. Equivalent to [A-Za-z0-9_].
\W	Matches anything but letters, digits and underscores. Equivalent to [^A-Za-z0-9_].
\xff	Matches ASCII hexadecimal character ff.
\x{ffff}	Matches UTF-8 hexadecimal character ffff.
\cA	Matches ASCII control character ^A. Control characters are case insensitive.
\132	Matches ASCII octal character 132.

Groups

Groups	
`(foo|bar)`	Matches pattern foo or bar.
`(foo)`	Define a group (or sub-pattern) consisting of pattern foo. Matches within the group can be referenced in a replacement using a backreference.
`(?<foo>bar)`	Define a named group named "foo" consisting of pattern bar. Matches within the group can be referenced in a replacement using the backreference $foo.
`(?:foo)`	Define a passive group consisting of pattern foo. Passive groups cannot be referenced in a replacement using a backreference.
`(?>foo+)bar`	Define an atomic group consisting of pattern foo+. Once foo+ has been matched, the regex engine will not try to find other variable length matches of foo+ in order to find a match followed by a match of bar. Atomic groups may be used for performance reasons.

Bracket Expressions

Bracket Expressions	
`[adf]`	Matches characters a or d or f.
`[^adf]`	Matches anything but characters a, d and f.
`[a-f]`	Match any lowercase letter between a and f inclusive.
`[A-F]`	Match any uppercase letter between A and F inclusive.
`[0-9]`	Match any digit between 0 and 9 inclusive. Does not support using numbers larger than 9, such as [10-20].

Quantifiers

- RegExs are greedy!
- {min, max} / {min,} / {, max} / {exact}
- ? -> {0,1}
- + -> {1,0}
- * -> {0,}

Quantifiers	
*	0 or more. Matches will be as large as possible.
*?	0 or more, lazy. Matches will be as small as possible.
+	1 or more. Matches will be as large as possible.
+?	1 or more, lazy. Matches will be as small as possible.
?	0 or 1. Matches will be as large as possible.
??	0 or 1, lazy. Matches will be as small as possible.
{2}	2 exactly.
{2,}	2 or more. Matches will be as large as possible.
{2,}?	2 or more, lazy. Matches will be as small as possible.
{2,4}	2, 3 or 4. Matches will be as large as possible.
{2,4}?	2, 3 or 4, lazy. Matches will be as small as possible.

Special Characters

Special Characters	
\	Escape character. Any metacharacter to be interpreted literally must be escaped. For example, \? matches literal ?. \\ matches literal \.
\n	Matches newline.
\t	Matches tab.
\r	Matches carriage return.
\v	Matches form feed/page break.

Assertions

Assertions	
`foo(?=bar)`	Lookahead assertion. The pattern foo will only match if followed by a match of pattern bar.
`foo(?!bar)`	Negative lookahead assertion. The pattern foo will only match if not followed by a match of pattern bar.
`(?<=foo)bar`	Lookbehind assertion. The pattern bar will only match if preceded by a match of pattern foo.
`(?<!foo)bar`	Negative lookbehind assertion. The pattern bar will only match if not preceded by a match of pattern foo.

POSIX Character Classes

POSIX Character Classes Must be used in bracket expressions, e.g. [a-z[:upper:]]	
`[:upper:]`	Matches uppercase letters. Equivalent to A-Z.
`[:lower:]`	Matches lowercase letters. Equivalent to a-z.
`[:alpha:]`	Matches letters. Equivalent to A-Za-z.
`[:alnum:]`	Matches letters and digits. Equivalent to A-Za-z0-9.
`[:ascii:]`	Matches ASCII characters. Equivalent to \x00-\x7f.
`[:word:]`	Matches letters, digits and underscores. Equivalent to \w.
`[:digit:]`	Matches digits. Equivalent to 0-9.
`[:xdigit:]`	Matches characters that can be used in hexadecimal codes. Equivalent to A-Fa-f0-9.
`[:punct:]`	Matches punctuation.
`[:blank:]`	Matches space and tab. Equivalent to [\t].
`[:space:]`	Matches space, tab and newline. Equivalent to \s.
`[:cntrl:]`	Matches control characters. Equivalent to [\x00-\x1F\x7F].
`[:graph:]`	Matches printed characters. Equivalent to [\x21-\x7E].
`[:print:]`	Matches printed characters and spaces. Equivalent to [\x21-\x7E].

Replacement Backreferences

Replacement Backreferences Used in replacements	
`$3 or \3 or`	Matched string within the third non-passive group.
`$0 or $& or \0`	Entire matched string.
`$foo` `${foo}`	Matched string within the group named "foo".

Case Modifiers

Modifiers May be grouped together, e.g. (?ixm)	
`(?i)`	Case insensitive mode. Make the remainder of the pattern or sub-pattern case insensitive.
`(?m)`	Multi-line mode. Make $ and ^ in the remainder of the pattern or subpattern match before/after newline.
`(?s)`	Single-line mode. Make the . (dot) in the remainder of the pattern or subpattern match newline.
`(?x)`	Free spacing mode. Ignore white space in the remainder of the pattern or subpattern.

Recursive Backreferences

Recursive Backreferences Used in patterns to reference captured text or sub-patterns from a capture group. Only available in some regex implementations.	
`\3 or \k<3>`	Re-match the text previously matched within the third non-passive group.
`\k<foo>`	Re-match the text previously matched within the group named "foo".
`\g<3>`	Re-execute the subpattern within the third non-passive group.
`\g<foo>`	Re-execute the sub-pattern within the group named "foo".

Expression Flags

- Placed after the /expression/
- (bad) JavaScript Example: /\w+@\w+.\w{2,3}/g
  - Not all languages use /expression/g flags
  - See java example on next page for for case insensitive example

Expression Flags	
`i`	With this flag the search is case-insensitive: no difference between A and a.
`g`	With this flag the search looks for all matches, without it – only the first match is returned.
`m`	
`s`	Enables “dotall” mode, that allows a dot . to match newline character \n.
`u`	Enables full Unicode support. The flag enables correct processing of surrogate pairs.
`y`	“Sticky” mode: searching at the exact position in the text.

RegEx examples in Java

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("w3schools", Pattern.CASE_INSENSITIVE);
    Matcher matcher = pattern.matcher("Visit W3Schools!");
    boolean matchFound = matcher.find();
    if(matchFound) {
      System.out.println("Match found");
    } else {
      System.out.println("Match not found");
    }
  }
}
// Outputs Match found
```


Se more here: https://www.w3schools.com/java/java_regex.asp