Information Systems
(Informationssysteme)

Jens Teubner, TU Dortmund
jens.teubner@cs.tu-dortmund.de

Summer 2016
Part IV

Database Design
Database Design Process

Database systems are very good at handling your data... once your data is in a form that can be digested by the system.

How do we get from a real-world problem to a database schema?

1 **requirements analysis** — Meet with customers, understand their problem.

2 **conceptual design** — Develop a high-level model for the data that should be stored in the database, typically using an **ER diagram**.
3 logical design — Convert the conceptual design into the data model of the chosen DBMS. Result is a conceptual schema (↗ slide 17).

4 schema refinement — Refine obtained conceptual schema, e.g., using normalization (see later).

5 physical design — Develop a physical schema that meets the application’s performance needs.

Note: In practice, you’ll have to re-iterate some or all of these steps multiple times until you reach a satisfactory design.
Main Goal: Understand user’s needs.

- Meet and discuss with user groups; study existing documentation and/or applications.
- Listen and watch out for real-world entities that should be reflected in the database and how they relate and interact with each others.
- Make sure you understand the user’s needs:
  - Note down your understanding in a way that you can discuss with your users (informal notation; prose text).
  - Re-iterate with users to make sure your understanding matches the needs of the users.
“As one of the largest cocktail bars in town, we are really proud of our large collection of cocktail recipes. For each cocktail (recipe) we would like to store its name, a short description, instructions how to make the cocktail, and an information how long that cocktail is already in our database. Each cocktail consists of a number of ingredients, which have a name, a short textual characterization of their flavor, and an information about the amount of alcohol they contain.

It is also important to know which supplier offers which of the ingredients (and at which price). Our supplier list contains addresses and URLs for each supplier. Sometimes, we even know a direct contact person that belongs to a supplier, including his/her name, phone number, and email address.”
Rule of thumb:

- Mark **subjects** in a customer’s description that describe **concepts** (or “entities”) that should be stored in the database.
  - *E.g.*, **cocktails** (or recipes) and **ingredients** should be stored in the database.

- Mark **verbs** that indicate **relationships** between concepts.
  - *E.g.*, cocktails **consist of** ingredients (or: ingredients **are contained in** cocktails).

- In addition, watch out for **attributes** that further characterize a concept/entity.
  - *E.g.*, name, description, etc. characterize cocktails; name, flavor, and alcohol percentage characterize ingredients.
Conceptual Database Design:

- High-level description of data to be stored in database.
- Typically uses a rather **formalized notation**.
  - Typically: **Entity-Relationship Model (ER Model)** and **ER Diagrams**.
  - Clear notation, yet **independent** of the data model used by the specific database system.

- The ER Model helps to
  - **communicate** with users (and verify the model) and
  - **translate** into a conceptual schema for the used DBMS.
    (We will learn rules how, e.g., an ER Diagram can mechanically be translated into a relational database schema.)
An **entity** is an object in the real world that is distinguishable from other objects.

An **entity set** is a collection of similar entities.

We represent an entity set in an ER Diagram as a **rectangle**.

An entity is described using a set of **attributes**.

We use **ellipses** to represent attributes.
The **domain** of an attribute describes its **possible values**.

*E.g.*,

- Name: *strings of length 30*
- Description: *strings of length 200*
- Since: *date value greater than Jan 1, 1970*
- Instructions: *strings of length 500*
A relationship is an association among two (or more) entities.

A relationship set is a collection of similar relationships.

We represent relationship sets as diamonds.

Relationships can carry attributes, too.
Entities, Relationships, and Sets Thereof

**Entity Set**  
Cocktails

**Entity**  
Dry Martini  
Paradise  
Screwdriver

**Relationship**  
1 dash

**Entity Set**  
Ingredients

**Entity**  
Vermouth Dry  
Gin  
Apricot Brandy  
Orange Juice  
Vodka

**Relationship Set**  
Consists of

© Jens Teubner · Information Systems · Summer 2016
More Relationships

Relationships can also associate two entities within the same entity set. 
E.g., some ingredients can be substituted by another one (when an ingredient has run out of stock):

And there can be multiple relationship sets between the same entity sets:
n-Ary Relationships

Relationships can be n-ary:

Students \(\xrightarrow{\text{takes exam}}\) Professors \(\xrightarrow{\text{takes exam}}\) Courses
Generally, an entity is **uniquely identified** by the values of its attributes.

Sometimes, a **subset** of attributes is enough to uniquely identify an entity.

→ e.g., Student ID; SSN; course number + semester; etc.

We call a minimal set of identifying attributes a **key**.

We use **underlining** to mark the (set of) key attributes.
If there is no simple identifying (set of) attribute(s), it is often useful to introduce an **artificial key attribute** (e.g., an integer number).

If there are multiple **candidate keys**, typically one is designated to be the **primary key**.
(Having a simple, designated key also eases translation to relational algebra, which we will look at later.)
What about keys for relationships?
Very often, the participation of entities in a relation set can be further constrained:

- Each cocktail consists of **at least one** ingredient.
- A contact person for a supplier is **optional**. But there must be **at most one** contact person per supplier.

**In other words:**

- In the **consists-of** relationship set,
  - each **cocktail** participates 1..∞ times,
  - each **ingredient** participates 0..∞ times.
- In the **contact-person-for** relationship set,
  - each **supplier** participates 0..1 times,
  - each **contact person** participates 1 time.
Use \((min, max)\) notation to specify such constraints.

→ Specify minimum and maximum number of times that each entity may participate in the relationship set.

\[\text{Cocktails} \quad (1, \ast) \quad \text{consists of} \quad (0, \ast) \quad \text{Ingredients}\]

\[\text{Suppliers} \quad (0, 1) \quad \text{contact person for} \quad (1, 1) \quad \text{Contact Persons}\]

**Typically:** Use ‘\(\ast\)’ instead of ‘\(\infty\)’. 
0, 1, and * are certainly seen most often in ER Diagrams. But other values can make sense, too.

Describe the meaning of these constraints in natural language.
An alternative, often-seen notation is to label relationship sets as either 1:1, 1:N (or N:1), or N:M.

(min, max) notation:

Alternative:

The semantics “one type of glass can be used for N different cocktails” is counter-intuitive to that of the (min, max) notation!
“Weak entities” are entities that can exist only in combination with a “strong entity”.

Since weak entities depend on their “strong” counterpart, they do not themselves have a unique key.

→ Use key of partner to form a complete key.
→ Here: \( \langle \text{Building No, Room No} \rangle \) together identify a room.
**Generalization**  Factor out common characteristics to build a common “supertype.”

**Specialization**  Derive new, specialized entity sub-types, possibly by adding new characteristics (such as new attributes).

There is no real standard notation to express generalization/specialization.
The **Unified Modeling Language (UML)** has emerged as a modeling language for a wide range of design tasks.

- In UML, **class diagrams** are closest to ER Diagrams.
- Unlike entity sets in the ER, UML classes can contain **methods**.

<table>
<thead>
<tr>
<th>class name</th>
<th>Cocktail</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>+ name</td>
</tr>
<tr>
<td></td>
<td>+ inDbSince</td>
</tr>
<tr>
<td>methods</td>
<td>+ description</td>
</tr>
<tr>
<td></td>
<td>+ instructions</td>
</tr>
<tr>
<td></td>
<td>+ printShoppingList()</td>
</tr>
</tbody>
</table>

- UML is more intended for (in-memory) application design. Entities/objects are, therefore, identified via **pointers**, not through explicit **key attributes**.
UML associations take the role of relationship sets in ER Models.

Often, associations are **directed**:

→ Can navigate from object to object only in one direction then.
Remaining question: How can we turn that into a database schema?
Mapping an **entity set** into a relation is straightforward.

→ Each attribute of the entity set becomes an attribute of the table.
Observe that we use the concept of **keys** also in the relational world.

- A **minimal set of fields** that **uniquely identifies a tuple (row)** in a table is called a **(candidate) key**.
  - In **any legal instance** of the relation, two distinct tuples cannot have identical values in all the fields of a key.
  - No subset of the key is a unique identifier for a tuple.

**The key constraint** is a property of the **schema**. A column that just happens to contain unique values in the current table **instance** **is not** a key.

- Again, among multiple candidate keys, we typically select one **primary key**.
Key Constraints

In database systems, keys can be declared together with the schema of the table. *E.g.*, in SQL:\(^5\)

```
CREATE TABLE Ingredients ( IngrID INTEGER NOT NULL, 
    Name CHAR(30), 
    Alcohol DECIMAL(3,1), 
    Flavor CHAR(20), 
    PRIMARY KEY (IngrID) )
```

The DBMS will enforce such constraints and reject any modifications that would violate the key constraint.

\(^5\)Fields marked as **NOT NULL** cannot be left blank for any row; key columns must be declared **NOT NULL**. **DECIMAL**\((m,n)\) is a decimal number type with \(m\) digits total and \(n\) digits after the decimal.
An \textit{n-ary relationship} in the ER Model is an \textit{n-tuple} of entities. That is, the corresponding \textit{relationship set} can be thought of as the set

\[
\{(e_1, \ldots, e_n) | e_1 \in E_1, \ldots, e_n \in E_n\}
\]

(ignoring relationship attributes for a moment).

We \textbf{could} use that representation to express relationships in the relational world:

<table>
<thead>
<tr>
<th>ConsistsOf</th>
</tr>
</thead>
<tbody>
<tr>
<td>CockID</td>
</tr>
<tr>
<td>---</td>
</tr>
</tbody>
</table>

- Observe that we re-named the ‘Name’ fields, because column names must be unique within one table.
Relationships in the Relational World

Such an encoding would incur significant \textbf{redundancy} and \textbf{storage overhead}.

- \textit{E.g.}, every cocktail redundantly appears at least once in the \textit{ConsistsOf} relation.

Remember that the ‘Cocktail ID’ already \textbf{uniquely determines} all remaining cocktail properties:

\[ \text{CockID} \rightarrow \text{CName} \times \text{Since} \times \text{Descr} \times \text{Instr} \ . \]

Columns \textit{CName}, \textit{Since}, \textit{Descr}, and \textit{Instr} can thus safely be \textbf{omitted} in \textit{ConsistsOf}.

\rightarrow When needed, the information can always be \textbf{looked up} in \textit{Cocktails} (with help of the \textit{CockID} value).
Likewise, we can also omit all non-key columns of *Ingredients*:

<table>
<thead>
<tr>
<th>ConsistsOf</th>
</tr>
</thead>
<tbody>
<tr>
<td>CockID</td>
</tr>
<tr>
<td>IngrID</td>
</tr>
</tbody>
</table>

Let us now put the *relationship attributes* back:

<table>
<thead>
<tr>
<th>ConsistsOf</th>
</tr>
</thead>
<tbody>
<tr>
<td>CockID</td>
</tr>
<tr>
<td>IngrID</td>
</tr>
<tr>
<td>Quantity</td>
</tr>
</tbody>
</table>
Assuming Cocktails, Ingredients, and ConsistsOf are stored in an SQL database, how could we re-construct the original, full ConsistsOf relation?
Foreign Keys

Such “lookups” occur very often in relational databases.

- The concept is, in fact, a **corner stone** of relational databases.
- Columns *CockID* and *IngrID* in *ConsistsOf* are called **foreign keys**. They refer to *Cocktails* and *Ingredients* (respectively).

Foreign keys can be declared in SQL, too:

```sql
CREATE TABLE ConsistsOf (  
  CockID INTEGER NOT NULL,  
  IngrID INTEGER NOT NULL,  
  FOREIGN KEY (CockID) REFERENCES Cocktails,  
  FOREIGN KEY (IngrID) REFERENCES Ingredients )
```

→ Foreign keys refer to the **primary key** of the respective relation.
Relational database systems use regular, user-accessible attribute values to reference between tuples. → No “pointers” or other internal data structures.

Remember physical data independence:
- Tuples can freely be moved to new locations in memory/on disk without breaking tuple associations.
- Foreign key references can always be “followed”\(^6\) in both directions.

\(^6\)SQL is declarative and does not really offer navigation primitives.
Which columns form a key in `ConsistsOf`?

<table>
<thead>
<tr>
<th>ConsistsOf</th>
</tr>
</thead>
<tbody>
<tr>
<td>CockID</td>
</tr>
<tr>
<td>IngrID</td>
</tr>
<tr>
<td>Quantity</td>
</tr>
</tbody>
</table>

Does `ConsistsOf` have a key at all?
Participation Constraints

For some relationship sets we know that an entity may appear at most once in the set:

\[
\begin{array}{c}
\text{Suppliers} \quad (0, 1) \quad \text{contact person for} \quad (1, 1) \quad \text{Contact Persons}
\end{array}
\]

The respective columns in the resulting relation thus must be keys.

Here:
- \textit{SupplID} is a key candidate in the \textit{ContactPersonFor} relation.
- \textit{ContactID} is a key candidate in the \textit{ContactPersonFor} relation.
Tables that have the **same key** can be **merged** into one:

<table>
<thead>
<tr>
<th>Cocktails</th>
<th></th>
<th>ServedIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>CockID</td>
<td>Name</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

merge

→

<table>
<thead>
<tr>
<th>Cocktails</th>
</tr>
</thead>
<tbody>
<tr>
<td>CockID</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Which of the participation constraints in

\[
\text{Cocktails} \xrightarrow{(1, 1)} \text{Served In} \xleftarrow{(0, \ast)} \text{Glasses}
\]

does the merged relation implement?
For 1:1 relationships, there are various options to merge relations:

1. No relations could be merged.
2. \textit{ContactPersons} could be merged with \textit{ContactPersonFor}.
3. \textit{Suppliers} could be merged with \textit{ContactPersonFor}.
4. All three relations could be merged into one.
Merging Relations

Suppose we chose option 3:

<table>
<thead>
<tr>
<th>SuppID</th>
<th>Name</th>
<th>Address</th>
<th>WWW</th>
<th>ContactID</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

(where `ContactID` is a foreign key on the `ContactPersons` relation).

What if there is no contact person for a given supplier?

- In such a case, we’d want to leave the `ContactID` empty.
- This can be done by setting `ContactID` to null.
Null Values

**Null values** play an important role in relational databases.

They are used to model a variety of real-world scenarios:

- A value exists (in the real world), but is **not known**.
  - A supplier might have a WWW URL, but we don’t know it.

- **No value exists**.
  - A supplier might not have a WWW URL.

- The attribute is **not applicable** for this tuple.
  - In a *Persons* relation, the *Semester* field only applies to students.

- . . .

**Null** is a **special value**, distinct from **any other value** in the column’s domain.

  - **Null** is **not** the numeric value 0 and **not** the empty string.
Behavior of Null Values

In operations and predicates, think of null as “unknown”:

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>unknown</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>unknown</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>unknown</td>
<td>false</td>
</tr>
</tbody>
</table>

- **Arithmetic operations** with null evaluate to null (null + 42 → null).
- **Comparisons** with null evaluate to null (Semester < null → null).

✍ Exercise to try at home in SQL: Given the “Presidents” database used in the lecture, find all presidents that are still alive (i.e., their DEATH_AGE is set to NULL).
In SQL, null values are expressed using the keyword NULL:

```
INSERT INTO Suppliers VALUES
(4711, 'Shop Rite', '31 Main St', NULL, NULL)
```

Null values can be allowed or disallowed for particular columns.

→ See SQL example on slide 67.

→ Key columns must not contain null values.
The allowance of null values is another knob to restrict participation constraints in the relational world. 

*E.g.*, ContactPersonFor:

![Diagram](null_values_participation_constraints_diagram.png)

Column *ContactID* in relation *Suppliers*:

- `NULL` (null values allowed) \(\Rightarrow\) \(\square \equiv (0, 1)\)
- `NOT NULL` (null values disallowed) \(\Rightarrow\) \(\square \equiv (1, 1)\)

By marking *ContactID* in *Suppliers* as a **key**, we can further constrain the maximum participation of *ContactPersons* in the relationship set.
To translate an ER Diagram into a relational schema:

1. Map all **entity sets** to a **relation**.
2. Identify a **primary key** in each resulting **relation**.
3. Map all **relationship sets** to a **relation**.
4. Identify **foreign key** constraints in all those relations.
5. Refine the resulting schema by **merging** relations with **same key**.
   - Often, there is some degree of freedom in that step.
   - (Dis)allow **null values** to reflect **participation constraints**

Generally, **not all** participation constraints can losslessly be modeled with only (foreign) key constraints and constraints on null values.