Database Management
Database Design

By: Meyer Tanuan (2022), Rick Kozak (2019), Glenn Paulley (2015), John Mckay (2011)

A database is a model

A database is a model of “things” that exist in the real world

- Sometimes these “things” are called “entities”
- Sometimes they are called “objects”

Some of these “things” are tangible

- Students
- Automobiles
- Warehouses
- Manufactured goods

Other “things” are intangible

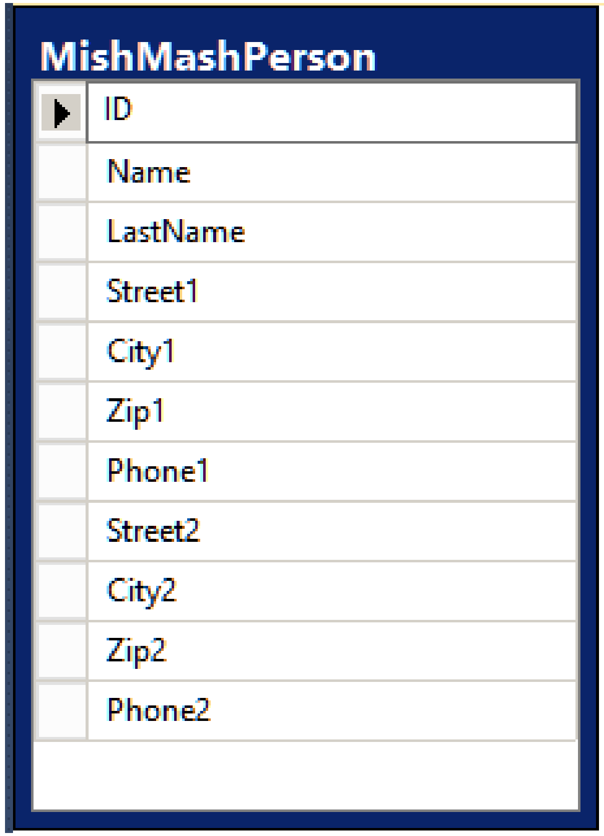
- Courses
- Insurance policies
- E-Books

The components of our model

- Database must consist of tables
- Tables consist of columns (fields)
- Table contain data records (rows)
- Each record in the database has unique identifier (primary key)
- Records in the other tables can reference other records by mentioning their primary keys. (foreign keys)

Requirements of our model

- Minimization of Storage
  - because of the ability to make references there is no need to store the same data more than once (e.g., Person table)
- Proper referencing
  - there is a need to keep the integrity of references



Data Example

Person A has a name, home address and office address. Each address has a separate phone number.

- In our minds, all this data is joined together
- It is very tempting to design one table which holds all this data

First Rule: Separate data

There are two different entities described in our table: Person and Address. We need to separate them.

Second Rule: Atomicity

- The data in each table must be atomic
  - i.e., it cannot be divided further.
- No field can contain the enumeration of data

Third Rule: Relationship

- If we have one home address per person and one work address per person, then to connect them, we can put a foreign key to person in the Address table OR a foreign key to address in the Person table. It doesn’t matter.
- If more than one person works at the same location, but only ever works in one place, then we need to put the foreign key in the Person table.

Third Rule: Relationship

- What if we have more than one person living at the same address?
- And we have persons living at more than one address?
- There are three approaches to solving this (M:M) problem:

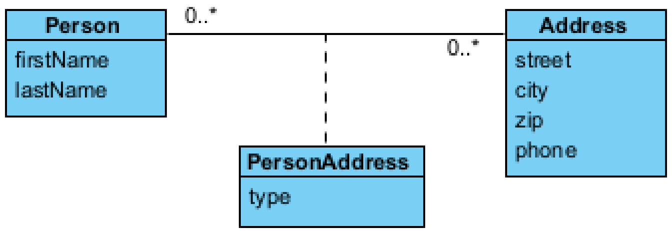
(1) Adding multiple address IDs into the Person table

(2) Adding multiple records for each person each with unique address id

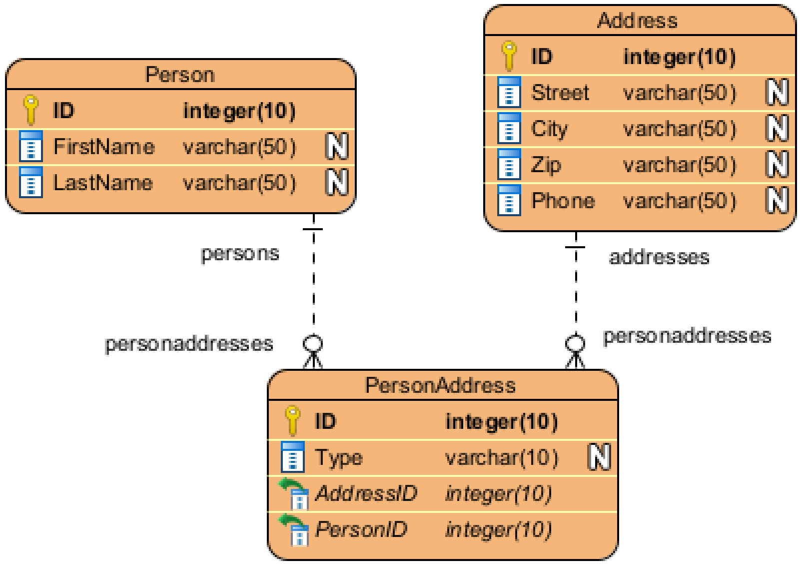
(3) Creating a separate **‘join’ or ‘linking’ table** to express this relationship

Third Rule: Relationship

We need to create another **PersonAddress** table which will contain references to the person and address primary keys as **foreign keys** (AddressID, PersonID)



UML Class Diagram



ERD

Third Rule: Relationship

ID	FirstName	LastName
1	Joe	Smith
2	Jane	Smith
NULL	*NULL*	*NULL*

ID	Street	City	Zip	Phone
1	123 Street	AnyCity	12345	123456789
2	987 Street	theCity	98765	987654321
3	654 Street	theCity	65465	656565656
NULL	*NULL*	*NULL*	*NULL*	*NULL*

PersonID	AddressID	Type
1	1	Home
1	2	Work
2	1	Home
2	3	Work
NULL	*NULL*	*NULL*

More formally …

We model the relationships, or associations, amongst the objects in the database

- Is the association/relationship optional?
- What is the cardinality of the relationship?
  - One-to-one
  - Many-to-one
  - One-to-many
  - Many-to-many
- In addition, are there additional attributes, or properties, that are dependent upon the existence of a relationship?
  - Example: a “grade” is a property of the relationship between a student and a course (or perhaps a section of a course)

Fourth Rule: Do not repeat (within reason)

If database contains certain repeating records it is possible to combine them into the Lookup table

PersonID	AddressID	AddressTypeID
1	1	1
1	2	2
2	1	1
2	3	2

ID	Description
1	Home
2	Office

Said another way…

- Ideally, we prefer not to store the same information about an entity, or object, more than once
- Often, redundant information leads to inconsistencies where duplicated copies diverge in value
- Maintaining duplicated information is not cost-free:
  - Greater storage required
  - Greater application complexity
  - Risk of inconsistency to the business
  - There are tradeoffs to be considered
  - Occasionally, deliberate redundancy can yield benefits, such as improved application performance
  - Choices must be made with due consideration

Another example

Suppose we have the table:

Student (<u>studentID</u>, SIN, studentName, <u>courseID</u>, courseName, grade)

NULL values are not allowed

- This design suffers from insert, update, and delete anomalies that ideally we would like to avoid:
  - INSERT – we cannot insert a new student without that student taking at least one course; similarly, cannot insert a new course without the course having at least one student
  - UPDATE – If a student changes their name, it will be necessary to modify each and every tuple (row) for that student, and modify the name stored in each row.
  - DELETE – If we delete the last tuple (row) for a student taking that particular course, we lose all of the information about that student

Another example

Solution: decompose the table “**Student**” into multiple tables:

- **Student**: facts about a student
  - Student ID
  - Student name
  - SIN
- **Course**: facts about a course
  - Course ID
  - Course name
- **StudentCourse**: facts about the relationship of a student enrolled in a specific course
  - Student ID
  - Course ID
  - Grade

Deciding on Keys

In DB design, one must differentiate between:

- What is an **entity**/object
  - And for each entity, what are the candidate key(s) for that entity?
- What is an **attribute** of a specific entity/object
- What are the **relationships** that involve this entity/object?

Example: Student

- A student is a “thing” and merits an entity/object to represent it
- Possible candidate keys:
  - SIN number
  - Student number
  - Student name
  - Internally generated number
- Possible attributes:
  - Name, DOB, address, email, etc.

Normalization

- The Normal Forms (1NF, 2NF, 3NF, 4NF)
- Benefits of Normalization
- When to denormalize

Video: https://www.linkedin.com/learning/relational-databases-essential-training/relational-database-normal-forms

First Normal Form (1NF)

The value stored at the intersection of each row and column (cell) must be a **scalar** value. A table must not contain any **repeating columns**.

- Every database record must have the same number of columns
  - In another words: columns cannot be omitted, omitted values in the columns are represented by NULL
  - Each cell must hold a single value
- In addition, every record must have a key (singular or composite)
- If database adheres to this rule it is in **1NF**.

Un-normalized student enrolment data

- Student enrolment data: courses and grades columns with non-scalar values:
- **Enrolment** (studentName, schoolTerm, courses, grades)
  - (“**Jay Smith**”, “Winter 2022”, “**PROG8080, INFO8000**”, “**82, 78**”)
  - (“**Jay Smith**”, “Spring 2022”, “**PROG8090**”, “**75**”)

1NF

- Attempt to fix the problem: **replace columns with non-scalar values**
- How to track more than two courses?
  - add new columns (e.g., course3, grade3)
- Side-effect: **repeating columns**
- **Enrolment** (studentName, schoolTerm, [(course1, grade1), (course2, grade2)])
  - (“Jay Smith”, “Winter 2022”, “PROG8080”, 82, “INFO2070”, 78)
  - (“Jay Smith”, “Spring 2022”, “PROG8090”, 75, NULL, NULL)

1NF

- In order to fix the problem, we need to eliminate repeating columns:
- **Enrolment** (studentName, schoolTerm, **course**, **grade**)
  - (“Jay Smith”, “Winter 2022”, “PROG8080”, 82)
  - (“Jay Smith”, “Winter 2022”, “INFO2070”, 78)
  - (“Jay Smith”, “Spring 2022”, “PROG8090”, 75)

The segment is in **1NF**.

New side-effect: **internal redundancy** (i.e., redundant studentName and schoolTerm values).

Second Normal Form (2NF)

Every non-key column must depend on the **entire** primary key.

- Every column (field) of the record must be related to the **key**, identifying this record.
- If the **key** is composite, no column can be related to only part of the key
- Consider the table with columns:

  studentId, courseId, grade, professor

- Grade refers to both parts of the key, professor only relates to the course .

2NF

- In order to fix the problem, we need to decompose the previous table to:

studentId, courseId, grade

courseId, professor

- This combination of two tables adheres to **2NF**. All non-key columns refer to the whole key and not parts
- The segment is in **2NF**.

Third Normal Form (3NF)

Every non-key column must depend ***only*** on the primary key.

- Third normalization rule states that no non-key column of the table can define or be defined by another non-key column of the same table

employeeId, departmentId, locationId

- Department and location are obviously connected but connection between employee and location is not necessarily direct (i.e., employee can work at more than one location)

3NF

In order to fix this problem, we need to decompose the previous table:

<u>employeeId</u>, departmentId

<u>departmentId</u>, locationId

This combination of tables adheres to **3NF**. It is said sometimes that such decomposition brings the whole database to a **3NF**.

Fourth Normal Form (4NF)

A table must not have more than one *multivalued dependency*, where the primary key has a one-to-many relationship to non-key columns.

Keys related to the independent qualities of entity with another key must be stored in different tables.

Consider the following table normalized using **3NF** rule:

employeeId, skillId, languageId

4NF

Because **skills** (Database, C#, Java) and **languages** (English, Russian, Mandarin, Hindi) are independent, the data is required to be stored in two separate tables:

employeeId, skillId

employeeId, languageId

If performed so, the database appears in **fourth normal form**

Designers usually stop at the **third normal form**, because the number of tables can expand drastically in the **fourth normal form**

Benefits of normalization

- More tables and the database has more clustered indexes. That makes data **retrieval more efficient**.
- Each table contains information about a single entity, and each index has fewer columns (usually one) and fewer rows. That makes **DML more efficient**.
- Each table has fewer indexes, which makes **DML more efficient**.
- Data redundancy is minimized, which simplifies maintenance and **reduces storage**.

Source: Murach's SQL Server 2019 for Developers

When to denormalize

- When a column from a joined table is used repeatedly in search criteria.
- If a table is updated infrequently.
- Include columns with derived values when those values are used frequently in search conditions.

Source: Murach's SQL Server 2019 for Developers

custId	orderNum	custName	phone	qty	itemNum	itemDesc	listPrice	extendedPrice
435	11997	John Edwards	(519) 447-9283	2	99334-1	Socks	12.00	24.00
435	11998	John Edwards	(519) 447-9283	1	99334-2	T-Shirt	14.95	14.95
435	11998	John Edwards	(519) 447-9283	3	99734-1	Belt	17.95	53.85
497	23415	Sharon Jones	(519) 582-5994	2	99311-5	Handbag	34.95	69.90
497	23416	Sharon Jones	(519) 582-5994	2	99312-1	Gloves	15.25	30.50

Task

Modify this to first, second, and third normal forms

Sample Solution

- **UNF** (un-normalized form):
  - **CustomerOrders** (custId, **orderNum**, custName, phone, qty, **itemNum**, itemDesc, listPrice, extendedPrice)
- **1NF**:
  - Same as UNF (no repeating columns, all cells have scalar values)
    - If another item is added to the same orderNum, a new row is added to **CustomerOrders**.
  - Composite primary key (PK): **orderNum + itemNum**
- **2NF**:
  - **Item** (itemNum, listPrice, itemDesc)
    - listPrice, itemDesc depends on itemNum (part of composite PK)
  - **Order** (orderNum, custId, custName, phone)
    - custId, custName, phone depends on orderNum (part of composite PK)
  - **OrderItem** (orderNum, itemNum, qty, extendedPrice)
    - Rename **CustomerOrders** to **OrderItem**
- **3NF**:
  - **Customer** (custId, custName, phone)
    - custName and phone (non-key columns) depends on custId (non-key column) of **Order**
  - **Order** (orderNum, custId)
  - No change to **Item** and **OrderItem** in 2NF