To Access Page

Normalization

Relational Database Normalization Process

Illogically or inconsistently stored data can cause a number of problems. In a relational database, a logical and efficient design is just as critical. A poorly designed database may provide erroneous information, may be difficult to use, or may even fail to work properly.

Most of these problems are the result of two bad design features called: redundant data and anomalies. **Redundant data** is unnecessary reoccurring data (repeating groups of data). **Anomalies** are any occurrence that weakens the integrity of your data due to irregular or inconsistent storage (delete, insert and update irregularity, that generates the inconsistent data).

Basically, normalization is the **process of efficiently organising data** in a database. There are two main objectives of the normalization process: **eliminate redundant data** (storing the same data in more than one table) and **ensure data dependencies make sense** (only storing related data in a table). Both of these are valuable goals as they **reduce the amount of space a database consumes** and **ensure that data is logically stored**.

The process of designing a relational database includes making sure that **a table contains only data directly related to the primary key**, that **each data field contains only one item of data**, and that **redundant** (duplicated and unnecessary) **data is eliminated**. The task of a database designer is to structure the data in a way that eliminates unnecessary duplication(s) and provides a rapid search path to all necessary information. This process of specifying and defining tables, keys, columns, and relationships in order to create an efficient database is called **normalization**.

Normalization is part of successful database design. Without normalization, database systems can be inaccurate, slow, and inefficient and they might not produce the data you expect.

We use the normalization process to design efficient and functional databases. By normalizing, we store data where it logically and uniquely belongs. The normalization process involves a few steps and **each step is called a form**. Forms range from the first normal form (1NF) to fifth normal form (5NF). There is also one higher level, called domain key normal form (DK/NF). However, we will cover the first 3 forms.

When normalising a database you should achieve four goals:

1. **Arranging data into logical groups** such that each group describes a small part of the whole
2. **Minimizing the amount of duplicated data** stored in a database
3. Building a database in which you can **access and manipulate the data quickly and efficiently** without compromising the integrity of the data storage
4. Organising the data such that, when you modify it, you **make the changes in only one place**

NOTE: Sometimes database designers refer to these goals in terms such as **data integrity**, **referential integrity**, or **keyed data access**.

Normalization is a complex process with many specific rules and different intensity levels. In its full definition, normalization is the process of **discarding repeating groups, minimizing redundancy, eliminating composite keys for partial dependency** and **separating non-key attributes**.

In simple terms, the rules for normalization can be summed up in a single phrase: **"Each attribute (column) must be a fact about the key, the whole key, and nothing but the key."** Said another way, each table should describe only one type of entity (information).

NOTE: 'Data' and 'information' are terms used interchangeably in everyday speech, but they do mean different things. **Data** are row facts. **Information** is organised and processed data. When you associate the data with other data items, you can create information.

A properly normalised design allows you to:

- **Use storage space efficiently**
- **Eliminate redundant data**
- **Reduce or eliminate inconsistent data**
- **Ease the database maintenance burden**

A bad database design usually include:

- **Repetition of information**
- **Inability to represent certain information**
- **Loss of information**
- **Difficulty to maintain information**

When you normalise a database, you start from the general and work towards the specific, applying certain tests (checks) along the way. Some users call this process **decomposition**. It means decomposing (dividing/breaking down) a 'big' un-normalise table (file) into several smaller tables by:

- **Eliminating insertion, update and delete anomalies**
- **Establishing functional dependencies**
- **Removing transitive dependencies**
- **Reducing non-key data redundancy**

As mention before, relational database theorists have developed a series of guidelines for ensuring that databases are normalized. They have divided normalization into several rules called **'normal forms'**:

- **Un-normalised data** = repeating groups, inconsistent data, delete and insert anomalies.
- **First Normal Form (1NF) = ELIMINATE REPEATING GROUPS (make a separate table for each set of related attributes, and give each table a primary key)**. A table is in 1NF if there are **no duplicated rows in the table**, if **data is atomic** (each cell is single-valued; there are not repeating groups or arrays), if **entries in a column are of the same kind** (type), and if **each row contains the same columns**. The order of the rows is irrelevant. The order of the columns is irrelevant. However, the requirement that there be no duplicated rows in the table means that **the table has a primary key** (singe or composite).

1NF meets the following criteria:

1. Each table contains **all atomic data items, no repeating groups,** and **a designated primary key (no duplicated rows)**

For example, in a table that describes a student, the student details and classes (modules) should not be stored in one data field, separated by commas. Rather the classes (modules) data should be moved to their own table, which should include a link, back to the student table (called a foreign key).

The following table **is not in 1NF** (it is **un-normalized**):

STUDENTS AND CLASSES (MODULES) DETAILS TABLE
Brian Smith, NBC Services, London, Introduction to PCs, Introduction to Windows, Introduction to the Internet, Microsoft Word Advanced, Microsoft Excel Advanced, Microsoft Access Advanced
Laura Grey, ABC Corporation, Nottingham, Introduction to PCs, Introduction to Windows, Introduction to the Internet, Microsoft Word Advanced, Microsoft Excel Advanced, Microsoft Access Advanced, VB.NET Windows-Based Applications, VB.NET Web Applications, VB.NET XML Web Services and Server Components, Solution Architecture, SQL Server 2000
John Brown, NBC Services, London, Microsoft Word Advanced, Microsoft Excel Advanced, Microsoft Access Advanced
Peter Black, ABC Corporation, Nottingham, VB.NET Windows-Based Applications, VB.NET Web Applications, VB.NET XML Web Services and Server Components, Solution Architecture, SQL Server 2000

The following tables **are in 1NF**:

STUDENTS TABLE

STUDENT ID	STUDENT NAMES	COMPANY NAME	COMPANY LOCATION

1	Brian Smith	NBC Services	London
2	Laura Grey	ABC Corporation	Nottingham
3	John Brown	IBM Limited	Birmingham
4	Peter Black	MCI Software	Edinburgh

TAKEN CLASSES (MODULES) TABLE

STUDENT ID	CLASS ID	CLASSES (MODULES)
1	1	Introduction to PCs
1	2	Introduction to Windows
1	3	Introduction to the Internet
1	4	Microsoft Word Advanced
1	5	Microsoft Excel Advanced
1	6	Microsoft Access Advanced
2	7	Introduction to PCs
2	8	Introduction to Windows
2	9	Introduction to the Internet
2	10	Microsoft Word Advanced
2	11	Microsoft Excel Advanced
2	12	Microsoft Access Advanced
2	13	VB.NET Windows-Based Applications
2	14	VB.NET Web Applications
2	15	VB.NET XML Web Services and Server Components
2	16	Solution Architecture
2	17	SQL Server 2000
3	18	Microsoft Word Advanced
3	19	Microsoft Excel Advanced
3	20	Microsoft Access Advanced
4	21	VB.NET Windows-Based Applications

4	22	VB.NET Web Applications
4	23	VB.NET XML Web Services and Server Components
4	24	Solution Architecture
4	25	SQL Server 2000

NOTE: In the original table each student name is followed by the company they work for (pays their course fees), and any class (module) name that the student is currently taking. To answer the question who is taking the 'Introduction to the Internet' course (module) we need to perform an awkward scan of the list looking for references to the 'Introduction to the Internet' class (module). Separating the repeating groups of databases from the students' information results in **1NF**. The 'student ID' in the 'Classes (Module) Table' matches the primary key in the 'Student Table', providing a foreign key for relating the two tables. Now it is easier to see and answer the question - Who is taking the 'Introduction to the Internet' class (module)? – Students with IDs 1 and 2 (Brian Smith and Laura Grey).

SUMMARY: First normal form (1NF) is the "basic" level of normalization and generally corresponds to the definition of any database, namely:

- It contains two-dimensional tables with rows and columns
- Each column corresponds to a sub-object or an attribute of the object represented by the entire table.
- Each row represents a unique instance of that sub-object or attribute and must be different in some way from any other row (that is, no duplicate rows are possible).
- All entries in any column must be of the same kind. For example, in the column labeled"Student Names" only student names are permitted.

Second Normal Form (2NF) = ELIMINATE REDUNDANT DATA (if an attribute depends on only part of multi-valued key, remove it to a separate table). A table is in 2NF if it **met all database requirements for 1NF**, and if **each non-key attribute is fully functionally dependent on the whole primary key**; data, which does not directly dependent on table's primary key must be moved into another table. To test for functional dependency you have to establish whether primary key determines each non-key attribute. No subset of the key can determine the attribute's value.

2NF meets the following criteria:

1. Each table contains **all atomic data items, no repeating groups**, and **a designated primary key (no duplicated rows)**

2. Each table has all **non-primary key attributes fully functionally dependant on the whole primary key**

NOTE: In the 'Classes (Modules)' table, the primary key is made up of the 'class ID' and 'student ID'. This make sense since they are different for every member/class (module) combination, but the class (module) name depends only on the 'class ID'; the same class (module) name appears redundantly every time its associated ID appears in the 'Classes (Modules) table:

- 'Introduction to PCs' class appears twice with IDs 1 and 7 (taken by 2 students with IDs 1 and 2)
- 'Introduction to Windows' class appears twice with IDs 2 and 8 (taken by 2 students with IDs 1 and 2)
- 'Introduction to the Internet' class appears twice with IDs 3 and 9 (taken by 2 students with IDs 1 and 2)
- 'Microsoft Word Advanced' class appears three times with IDs 4, 10 and 18 (taken by 3 students with IDs 1, 2, and 3)
- 'Microsoft Excel Advanced' class appears three times with IDs 5, 11 and 19 (taken by 3 students with IDs 1, 2, and 3)
- 'Microsoft Access Advanced' class appears three times with IDs 6, 12 and 20 (taken by 3 students with IDs 1, 2, and 3)
- 'VB.NET Windows-Based Applications' class appears twice with IDs 13 and 21 (taken by 2 students with IDs 2 and 4)
- 'VB.NET Web Applications' class appears twice with IDs 14 and 22 (taken by 2 students with IDs 2 and 4)
- 'VB.NET XML Web Services and Server Components' class appears twice with IDs 15 and 23 (taken by 2 students with IDs 2 and 4)
- 'Solution Architecture' class appears twice with IDs 16 and 24 (taken by 2 students with IDs 2 and 4)
- 'SQL Server 2000' class appears twice with IDs 17 and 25 (taken by 2 students with IDs 2 and 4)

So, what we have is – several students are taking the same classes (modules), which are under different 'class IDs'. This can cause an update or/and a delete anomaly. To avoid these problems, we need **2NF**. To achieve this, we need to separate the attributes depending on both parts of the primary key from those depending only on the 'class ID'. This results in two tables – 'Classes (Modules)' and 'Taken Classes (Modules)' tables.

NOTE: The student ID is the primary key. Student names are fully functionally dependant upon the student ID – this data is specific to each other. The class ID is the primary key. Class (module) names are fully functionally dependant upon the class (module) ID - this data is specific to each other. However, the classes (modules) taken are dependant upon each student and may differ from a student to a student.

The following example tables **are in 2NF**:

STUDENTS TABLE

STUDENT ID	STUDENT NAMES	COMPANY NAME	COMPANY LOCATION

1	Brian Smith	NBC Services	London
2	Laura Grey	ABC Corporation	Nottingham
3	John Brown	IBM Limited	Birmingham
4	Peter Black	MCI Software	Edinburgh

CLASSES (MODULES) TABLE

CLASS ID	CLASSES (MODULE)
1	Introduction to PCs
2	Introduction to Windows
3	Introduction to the Internet
4	Microsoft Word Advanced
5	Microsoft Excel Advanced
6	Microsoft Access Advanced
7	VB.NET Windows-Based Applications
8	VB.NET Web Applications
9	VB.NET XML Web Services and Server Components
10	Solution Architecture
11	SQL Server 2000

TAKEN CLASSES (MODULES) TABLE

STUDENT ID	CLASS ID
1	1
1	2
1	3
1	4
1	5
1	6
2	1
2	2
2	3
2	4

2	5
2	6
2	7
2	8
2	9
2	10
2	11
3	4
3	5
3	6
4	7
4	8
4	9
4	10
4	11

NOTE: Each 'class ID' determines a particular 'class (module) name' (its value) - by establishing a class unique identifier value, we can find out its name as well. So, each class (module) name is functionally dependant on and decided by its 'class ID'. Each 'student ID' determines a specific 'student name' (his/her value) - by establishing a student unique identifier value, we can find out his/her name as well. So, each student name is functionally dependant on and decided by his/her 'student ID'.

SUMMARY: Second normal form (2NF) - At this level of normalization, each column in a table that is not a determiner of the contents of another column must itself be a function of the other columns in the table. For example, in a table with three columns containing customer ID, product sold, and price of the product when sold, the price would be a function of the customer ID (entitled to a discount) and the specific product.

Third Normal Form (3NF) = ELIMINATE COLUMNS NOT DEPENDANT ON KEY (if attributes do not contribute to a description of the key remove them to a separate table). A table is in 3NF if it **met all database requirements for both 1NF and 2NF**, and if **all transitive dependencies are eliminated** (each column must depend directly on the primary key; all attributes that are not dependant upon the primary key must be eliminated

(e.g. attributes that can be derived from data contained in other fields and tables must be removed)).

3NF meets the following criteria:

1. Each table contains **all-atomic data items**, no repeating groups, and a designated primary key
2. Each table has all **non-primary key attributes fully functionally dependant on the whole primary key**
3. All **transitive dependencies are removed** from each table

NOTE: We need to find a determinant as any column or collection of columns that determine another column. By this definition, primary key, or any candidate key, will be a determinant.

As we could see the 'Students' table satisfy the 1NF since it contains no repeating groups. It also satisfies the 2NF since it does not have a multi-valued (composite) key. However, the company name and company location are relevant to a specific company, not a member. To achieve **3NF** they must be moved into a separate table – 'Companies' table. Since they describe a company, 'company ID' (code) becomes the key of the new table.

STUDENTS TABLE

STUDENT ID	STUDENT NAMES	COMPANY ID
1	Brian Smith	NBC
2	Laura Grey	ABC
3	John Brown	IBM
4	Peter Black	MCI

COMPANIES TABLE

COMPANY ID	COMPANY NAME	COMPANY LOCATION
NBC	NBC Services	London
AS	Access Solutions	London
ABC	ABC Corporation	Nottingham
MNB	Moor, Norman and Brown Association	Leeds
IBM	IBM Limited	Birmingham
MCI	MCI Software	Edinburgh

The motivation for this is again the same – to prevent the update and/or delete anomalies.

NOTE: A transitive dependency can be described as follows: **"If A determines B, and B determines C, then A determines C."**

You achieve the 3NF when you resolve all **transitive dependencies**. And once again, you have to test the attributes in each table, but this time you test and check to see whether, within a table, any non-key attribute determines the value of another non-key attribute. (NOTE: Such a determination defines transitive dependency, which may cause additional redundancy.) Each non-key attribute must be fully functionally dependent on the entire primary key, and not on any other non-key attribute – no transitive dependencies exist among attributes.

When your database is in 3NF you can **analyze each table independently** of all others in the database, and then deduce a normal form for that table. However, the success of the database design and normalization hinges on what kind of relationship each table has to each other table, and on the correct expression of each relationship.

If you ensure that each table is in 3NF, you **avoid problems that can arise when users update/insert/delete data**. You must make sure that you've stored each non-primary key attribute only once to remove all redundant data and to **remove the possibility of unsynchronised data**, which can damage data recovery.

> **SUMMARY: Third normal form (3NF)**. At the second normal form, modifications are still possible because a change to one row in a table may affect data that refers to this information from another table. For example, using the customer table just cited, removing a row describing a customer purchase (because of a return perhaps) will also remove the fact that the product has a certain price. In the third normal form, these tables would be divided into two tables so that product pricing would be tracked separately.

IMPORTANT NOTE: You must be able to **reconstruct the original flat view of the data**. If you violate this rule, you will have defeated the purpose of normalising the database.

A poorly normalized database and poorly normalized tables can cause problems ranging from **excessive disk I/O** and **subsequent poor system performance** to **inaccurate data**. An improperly normalized condition can result in **extensive data redundancy**, which puts a burden on all programs that modify the data. From a business perspective, the expense of bad normalization is **inadequate** and **weak operating systems**, and **inaccurate**, **incorrect**, or **missing data**. Applying normalization techniques to database design helps create **efficient systems that produce accurate data and reliable information**.

A table is in **Boyce-Codd normal form (BCNF)** if it is **in 3NF** and if **every determinant is a candidate key** - a combination of attributes that can be uniquely used to identify a database record; each table may have one or more candidate keys; one of these candidate keys is selected as the table primary key. So, a table is in BCNF only **if every determinant is a candidate key**.

NOTE: Most of the 3NF relations are also the BCNF relations, but **3NF is not BCNF if**:

1. Candidate keys in the table are composite keys (they are not single attributes)
2. There is more than one candidate key in the table, and
3. The keys are not disjoint (some attributes in the keys are common)

Fourth Normal Form (4NF) = ISOLATE INDEPENDENT MULTIPLE RELATIONSHIPS (no table may contain two or more 1:M or M:M relationships that are not directly related). This applies to only 1:M and M:M relationships. To be in **Fourth Normal Form (4NF)**, a relation must first be in **Boyce-Codd normal form**. Additionally, a given relation may not contain more than one multi-valued attribute. There should not exist any non-trivial multi-valued dependencies in a table. To move from BCNF to 4NF, remove any independently multi-valued components to two new 'parent' entities.

For example, if an employee can have many skills and many dependents, move the skill and dependent information to separate tables since they repeat and they are independent of each other.

Fifth Normal Form (5NF) = ISOLATE SEMANTICALLY RELATED MULTIPLE RELATIONSHIPS (there may be practical constraints on information that justify separating logically related M:M relationships).

By now, you have seen that normalization results in splitting tables from one table into two or more tables to eliminate data inconsistency. One benefit (advantage) of this splitting is that designer could always reconstruct the original (flat) table by joining the once created during normalization process. **Fifth normal form (5NF)** defers from the definitions of the previous normal forms – **it defines a goal to be reached**, rather than the resolution (declaration) of a particular anomaly. The goal to be reached with 5NF is to **keep splitting the tables** until either of the following two conditions is reached:

1. Further splitting would result in tables that could NOT be joined to recreate original (flat) table
2. The only splits left are trivial

Domain/key normal form (DKNF) = A table is in DKNF if every constraint on the table is a logical consequence of the defining of keys and domains.

Constraint -> a rule governing static values of attributes

Key -> row unique identifier

Domain -> description of an attribute's allowed values

A key uniquely identifies each row in a table. A domain is the set of permissible values for an attribute. By enforcing key and domain restrictions, the database is assured of being **freed from modification anomalies**. DKNF is the normalization level that most designers aim to achieve.

As you could see normalization is process of **structuring relational database schema** such that most ambiguity is removed. The stages of normalization process are referred to as 'normal' forms' and progress **from the least restrictive** (1NF) **through the most restrictive** (5NF).

However, most database designers do not attempt to implement anything higher than 3NF or BCNF.

So, in order to fully normalise a relational database we have to **determine dependency** ((column) **relationships** between different tables). To be dependant, the columns must relate (affect one another).

After dividing the information into tables and identifying primary key fields, you need a way to tell the system **how to bring related information back together again in meaningful ways**. To do this, you define relationships between tables. Relationship is an **association between common data fields (columns) in two tables**. A relationship works by **matching data** in key fields. In most cases, these matching fields are the primary key from one table and a foreign key in the other table.

The kind of relationships, that the system creates, depends on how the related fields are defined. When you physically join two tables by connecting fields with related information, you create a relationship that is recognized by the system (e.g. Access). The specified relationship is important. It tells system **how to find and display information from fields in two or more tables**. The program needs to know whether to look for only one record in a table or to look for several records on the basis of the relationship.

A database is an **organised, integrated collection of data items**. The integration is important; data items relate to other data items, and groups of related data items (called entities) relate to other entities. The relationships between entities can be one of three types, one-to-one (1:1), one-to-many (1:M), and many-to-many (M:N):

1. **One-to-one (1:1)** – each instance of one entity relates to only one instance of the second entity - each record in Table A can have only one matching record in Table B, and each record in Table B can be related to only one record in Table A. This type of relationship is not frequently used in database systems, but it can be very useful way to link two tables together. However, the information related in this way could be in one table. You might use a one-to-one relationship to divide a table with many fields in order to isolate part of a table for security reasons, or to store information that applies only to a subset of the main table, or for efficient use of space. A one-to-one relationship is created if both of the related fields are primary keys or have unique indexes.
2. **One-to-many (1:M)** - each instance of one entity relates to one or more instances of the second entity; this is the most common type of relationship and it is used to relate one record from the 'primary' table with many records in the 'related' table. In a one-to-many relationship, a record ('parent') in Table A can have many matching records ('children') in Table B, but a record ('child') in Table B has only one matching record ('parent') in Table A. This kind of relationship is created if only one of the related fields is a primary key or has a unique index.
3. **Many-to-many (M:M)** – many instances of one entity relate to many instances of the second entity; this relationship is used to relate many

records in the table A with many records in the table B. A record ('parent') in Table A can have many matching records ('children') in Table B, and a record ('child') in Table B can have many matching records ('parents') in Table A. It is the hardest relationship to understand and it is not correct. By breaking it into two one-to-many relationships and creating a new (junction/link) table to stand between the two existing tables will enable correct and appropriate relationship setting. A many-to-many relationship is really two one-to-many relationships with a junction/link table. NOTE: Link table usually has the composite primary key that consists of the foreign keys from both tables A and B.

The relationships are usually displayed (represented) through a technique called **entity relationship modeling (ERM)**. Entities can be of two types: **noun-type** entities (people, places, and things) and **verb-type** entities (actions and interactions between the noun-type entities). **Entity relationship modeling** is a way to graphically represent the structural database design and to model the informational requirements. The result of the entity modeling efforts is an **entity-relationship diagram (ERD)**. Entities in an entity-relational diagram eventually **become tables** in a database.

NOTE: Although, there are many different entity-modeling methodologies, there are two commonly accepted rules:

1. **Entity names are always expressed in the singular** (you can identify the type of relationship between singular entity names more easily than you can determine the relationship between plural entity names)
2. **Entity names are always capitalized**

A **primary key (PK)** is an attribute (data column) in a table that serves a special purpose. The data items that make up this attribute are unique; no two data item values in this data column (can be) are the same. The primary key value serves to uniquely identify each row in the table. Each table must have an explicitly designated primary key. Each table has only one primary key; however, it can include more than one attribute (called a composite or concatenated primary key).

A **foreign key (FK)** is an attribute (data field) that forms an implied link between two tables that are in a 1:M relationship. The foreign key, which is a column in the table of the many, is usually a primary key in the table of the one. Foreign keys represent a type of controlled redundancy.

When tables are linked (joined) together, one table is usually called '**parent**' or '**primary**' table ('**one end**' in the 1:M relationship and '**one end**' (primarily created table) in the 1:1 relationship) and another table is called '**child**' or '**related**' table ('**many end**' in the 1:M relationship and '**one end**' (subsequently created table) in the 1:1 relationship). This is known as a **parent-child relationship** between tables. Records in a primary table cannot be modified or deleted if there are related records in the 'child' table - there will not be an orphan (related) record without a parent (primary) record. Also, a new record cannot be added to the related table if there is no associated record in the primary table.

In addition to specifying relationships between two tables in a database, you also set up referential integrity rules that help in maintaining a degree of

accuracy between tables. Setting **referential integrity rules** would prevent unwanted and accidental deletions and modifications of the 'parent' records that relate to records in the 'child' table. This type of problem could be catastrophic for any system. The referential integrity rules keep the relationships between tables intact and unbroken in a relational database management system - referential integrity prohibits you from changing existing data in ways that invalidate and harm the links between tables.

NOTE: Referential integrity operates strictly on the basis of the tables' key fields. It checks each time a key field, whether primary or foreign, is added, changed or deleted. If any of these listed actions creates an invalid relationship between two tables, it is said to **violate referential integrity**. Referential integrity is a system of rules that Microsoft Access uses to ensure that relationships between records in related tables are valid, and that you don't accidentally delete or incorrectly change related data.

Data integrity:

- Ensures the **quality of data** within a database
- Is about the actual **values that are stored and used** in an application's data structures
- Ensures that an application exert deliberate **control on every process that uses your data** to ensure the continued correctness of the information
- Is applied through the careful implementation of several **key concepts** such as normalizing data, defining business rules, providing referential integrity and validating the data

There are 4 types of the data integrity:

1. **Entity Integrity** ensures that each row (record) is a unique instance in a particular table by enforcing the integrity of the primary key or the identifier column(s) of a table (e.g. ID, Reference Code, etc).
2. **Domain Integrity** ensures validity of entries (data input) for a column through the data type, the data format and the range of possible values (e.g. date, time, age, etc.).
3. **Referential Integrity** preserves the defined relationships between tables when records are added, modified or deleted by ensuring that the key values are consistent across tables; such consistency requires that there are no references to non-existent values and if a key value changes, all references to it change consistently through database, otherwise a key value cannot be changed.
4. **User-Defined Integrity** enables specific (required) business rule(s) to be defined and established in order to provide correct and consistent control of an application's data access (e.g. who can have permissions to modify data, how generated reports should look like, which data can be modified, etc.)