RDM Best Practices.

Naming Convention. Data Dictionary. Data Types. Constraints. Indexes.

Contents

1. Naming Rules Convention
2. Data Dictionary
3. Choosing Data Types
4. Data Integrity and Constraints
5. Indexes

© Yuriy Shamshin, 2020

1. Naming Rules Convention

Naming convention descript important information about an item's identity and contents. A coherent naming rule can prevent many errors and frustrations and adds to the understanding of the structure of the database schema. Using a naming convention are more to do with human factors than any system limitations.

You'll want to name all the obvious database elements:

- Tables
- Views
- Columns
- Keys – including the primary key, alternate keys, and foreign keys
- Schemas

But don't leave out the less-visible items:

- Tablespaces
- Constraints
- References
- Indexes
- Stored procedures
- Triggers
- Sequences
- Variables

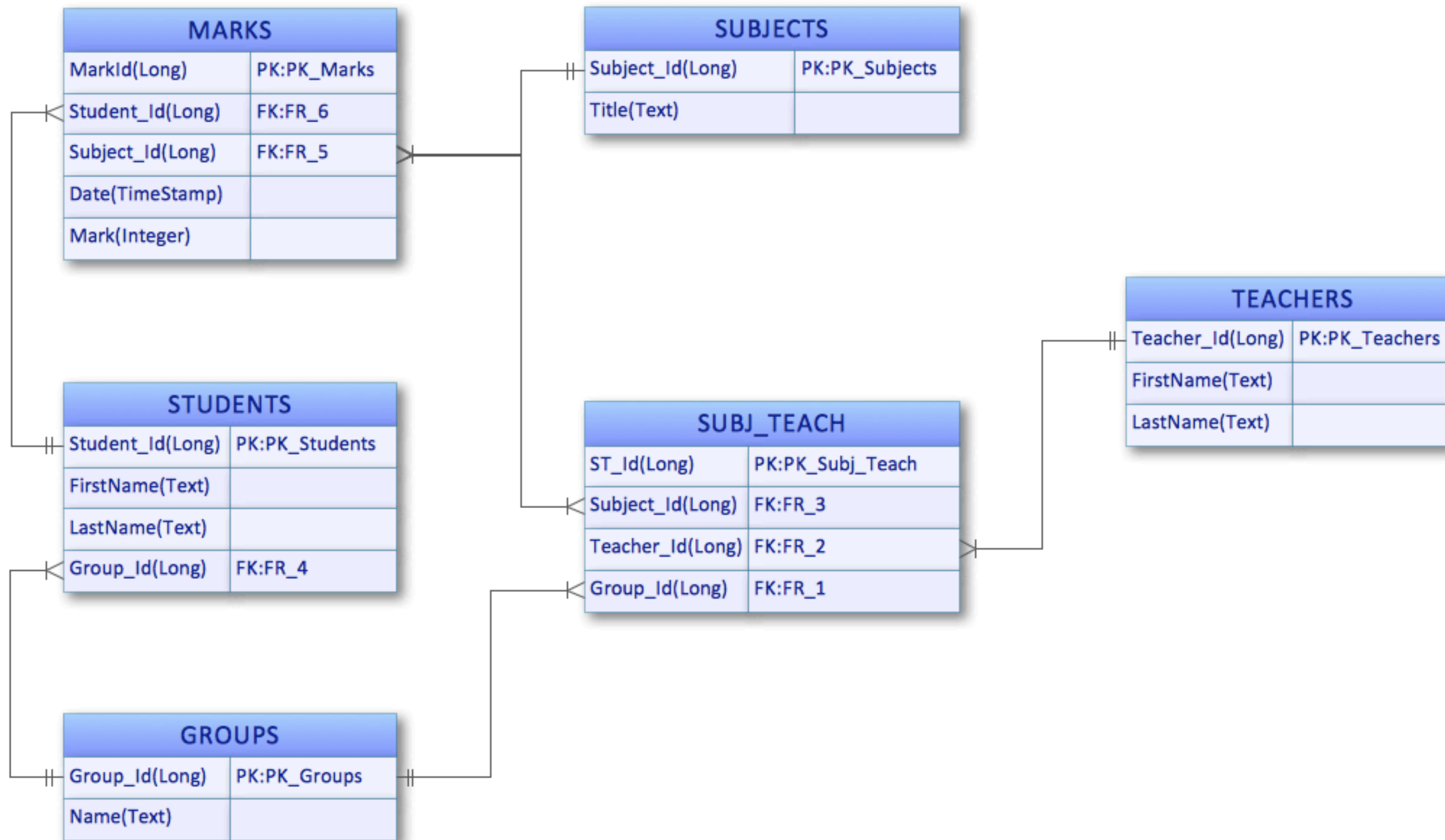
Naming Conventions + Constancy = Best Practice

Naming conventions are a matter of personal choice. What you finally decide on is only valuable when decisions are consistent and documented. A well-designed model adheres to a naming convention, but the occasional exception can be permitted if it can also be justified.

Article "How I Write SQL, Part 1: Naming Conventions" https://launchbylunch.com/posts/2014/Feb/16/sql-naming-conventions/

© Yuriy Shamshin, 2020

1.1. Entity Relationship Diagram Example – University Session



1.2. ERD/RDM Name Construction Rules

1. Names are constructed **only of Latin alphabetic characters and underscore**; no national, numerals, punctuation, or other special characters (such as spaces or dashes) are allowed. **Underscore** ensures the name remains readable even where case is not preserved.
   For MySQL: [0-9, a-z, A-Z, _, $]

2. Limit the all **names** to **32-64 characters**.
   For MySQL: 64 characters, read this https://dev.mysql.com/doc/refman/8.0/en/identifiers.html

3. For **table names** the all letter is **capitalized**, or capitalized first letter only
   Examples: STUDENTS, MARKS, SUBJ_TEACH or Students, Subjects, Marks, Subj_Teach.

4. Table names are **plural**, field name is **singular**.
   Examples: table called MARKS (Marks); fields called Mark, Date (mark, date).

5. Tables and fields should be **unique** within the database schema. For fields use the prefix with a 2 or 3 character of the table name.
   Examples: STUDENTS and TEACHERS would have a field called stdFirstName and tshFirstName

6. **Simple field names** are constructed using **full words** or commonly accepted **acronym** and **abbreviation** to provide as much intuitive meaning as possible.
   Examples: Title, Name, Mgmnt (Management), NY (New-York).

7. For composite name use naming in the **object/modifier/class** format.
   Examples: PersonFirstName, ProductSalesQuantity.

8. For easy reading of the **composite name**, do not use **lowercase** and **kebab-case**, but use **PascalCase** (the first letter of each word in a phrase is written with a capital letter), or **camelCase**, or **snake_case** (with an underscore "_"), or **mixing_Case**.
   Examples bad: sitememo, conversiontomillion, site-memo, conversion-to-million
   Examples right: site_memo, conversion_to_million, siteMemo, conversionToMillion, zipCode, SiteMemo, ConversionToMillion.

9. The **PK field** of each non-associative table is identified by the **table root name** and **a suffix of ID** (or _ID, or _Id). This allows distinction between key and non-key (data) fields. Or Primary Key Fields indicate by appending _pk.
   Examples: MarkId in MARKS table, or Location_ID as the PK field of the LOCATIONS, or Patient_ID_pk for PATIENTS.

10. The presence of _ID (or _ID_fk) suffix as **FK fields** in a table also clearly indicates that one or more non-key fields are available in a related table of the same name.

Examples: student_ID and Subject_ID in MARKS table.

11. Designate the **index** as **idx_**<TableName>.
    Examples: PATIENTS would have a primary key index called idx_Patients.

12. Designate the View as prefix **vw_**<TableName>.
    Examples: PATIENTS would have a primary key index called idx_Patients

13. Your **constraint** conventions might look like this:

    - pk_TableName for primary key constraints
    - fk_TableName_ReferencedTableName[_n] for foreign key constraints
    - uq_TableName_columnName[_columnName2...] for unique constraints
    - ck_TableName_columnName (or ck_TableName_n) for check constraints

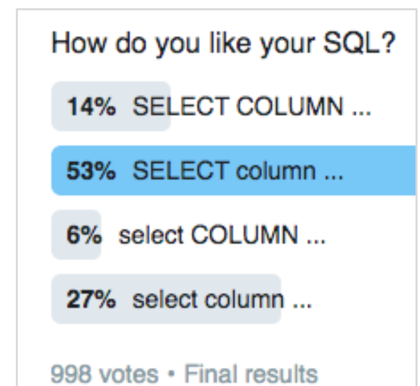
14. Avoid using SQL and database engine-specific **keywords** as names. Reserved words are permitted as identifiers if you quote them.
    - For MySQL read the https://dev.mysql.com/doc/refman/8.0/en/keywords.html
    - For Postgre SQL read the https://www.postgresql.org/docs/9.3/sql-keywords-appendix.html
    - For Oracle read the https://docs.oracle.com/database/121/SQLRF/ap_keywd.htm#SQLRF022
    - For MS SQL read the https://docs.microsoft.com/en-us/sql/t-sql/language-elements/reserved-keywords-transact-sql

    Examples: CREATE TABLE interval (begin INT, end INT); vs CREATE TABLE `interval` (begin INT, end INT);

1.3. SQL Name Convention Best Practice

Type all SQL statements in **lowercase**, being consistent with capitalisation improves the caching of SQL statements. A common variant is to put only SQL keywords in capitals.

```
SELECT
   em_employee_id_pk,
   em_employee,
   ab_start_date
FROM
   employees em,
   absences ab
WHERE
   absences.ab_employee_id=employees.em_employee_id_pk;
```



© Yuriy Shamshin, 2020

2. Data Dictionary

Write and maintain a data dictionary for all data elements - you should be defining the business meaning of each data item.

Data Dictionary - Owner Registration Information

Entity: Owner This table contains information about the people who own a registered vehicle

Field Name	Description	Type	Specifications	Default	Required	Unique	Key(s)
DLID	Drivers License Number	Character	9 numeric characters		Yes	Yes	PK
Last Name	Owner's Last Name	Character	25 alpha-numeric characters		Yes	No	
First Name	Owner's First Name	Character	20 alpha-numeric characters		Yes	No	
Middle Name	Owner's Middle Name/Initial	Character	25 alpha-numeric characters		No	No	
DOB	Owner's Date of Birth	Date	'MM/DD/YYYY' format		Yes	No	
DayPhone	Owner's Daytime Phone Number	Integer	10 digits; Area Code and Phone Number		Yes	No	
MailAddr1	First line of Owner's Mailing Address	Character	30 alpha-numeric characters		Yes	No	
MailAptNo	Owner's Apartment Number	Character	10 alpha-numeric characters		No	No	
MailAddr2	Second line of Owner's Mailing Address	Character	30 alpha-numeric characters		No	No	
MailCity	Mailing Address City/Town	Character	30 alpha-numeric characters		Yes	No	
MailState	Mailing Address State	Character	2 alpha characters, valid State acronym	'NY'	Yes	No	
MailZip	Mailing Address Zip Code	Character	9 numeric characters		Yes	No	
MailCounty	Mailing Address County	Integer	FIPS County Code		Yes	No	FK

Entity: Vehicle This table contains information about registered vehicles

Field Name	Description	Type	Specifications	Default	Required	Unique	Key(s)
VIN	Vehicle Identification Number	Character	14 alpha-numeric characters		Yes	Yes	PK
Year	Vehicle Year	Character	4 numeric digits, no leading zeros		Yes	No	
Make	Vehicle Make	Character	20 alpha-numeric characters		Yes	No	
BodyType	Type of Vehicle Body	Character	20 alpha-numeric characters		Yes	No	FK
OthBodyType	'Other' Type of Vehicle Body Description	Character	20 alpha-numeric characters; required if BodyType = 'Other'		Yes	No	
Color	Color of vehicle	Character	20 alpha-numeric characters		Yes	No	
Weight	Weight of vehicle (Unladen), to nearest lb.	Integer	up to 6 digits		Yes	No	
Power	Fuel source	Character	10 alpha characters		Yes	No	FK
Cylinders	Number of engine cylinders	Character	10 alpha-numeric characters		Yes	No	
MaxGrWgt	Trailer and Commercial Vehicle Max. Gross Weight	Character	30 alpha-numeric characters		Yes	No	
SeatingCap	Livery Vehicle Seating Capacity	Character	30 alpha-numeric characters		Yes	No	
Odometer	Odometer Reading at time of Acquisition	Character	2 alpha characters, valid State acronym		Yes	No	
Display	Number of Odometer Reading Digits Displayed	Integer	1 numeric character (5, 6, 7)		Yes	No	
Axles	Commercial Vehicle Number of Axles	Integer	2 numeric characters		Yes	No	

3. Choosing Data Types in Mysql

When creating tables in Mysql, we define data types and additional rules for columns and tables (size, restrictions, indexes):

```
CREATE TABLE `users` (
  id int unsigned NOT NULL AUTO_INCREMENT,
  email varchar(128) NOT NULL,
  first_name varchar(32) NOT NULL,
  last_name varchar(32) NOT NULL,
  gender char(1) NOT NULL,
  PRIMARY KEY (id),
  UNIQUE KEY email (email)
) ENGINE=InnoDB;
```

3.1. Short version for Data Type selection

How to choose the "right" data types?

Very simple - you need to comply with the rule **"The less is the better"**.

The less space the values in the table take, the easier it will be for the database to read and write them, and the less disk space is required.

- Avoid using NULL values.
- Select the minimum required integer numeric types (TINYINT / SMALLINT / MEDIUMINT instead of INT).
- Use FLOAT / DOUBLE instead of DECIMAL for approximate numbers (for using CPU hard engine).
- Select CHAR for lines of approximately the same length.
- For other lines - select VARCHAR.
- Select TEXT for very long text value only.
- Do not use TEXT / BLOB for sorting and indexing.
- Use ENUM instead of fixed-line strings (for example, a list of countries or days of the week or month names).
- Use TIMESTAMP to set the time of events (registration, sending messages, etc.).
- For other dates, use DATETIME.

© Yuriy Shamshin, 2020

3.2. NULL VALUE

Mysql's NULL value is a special value. There are special functions for working with it. To process it, additional logic is needed. A good rule of thumb is to avoid using this value. Instead, you can use empty values for strings or zeros for numbers:

```
CREATE TABLE `users` (
  ...
  age tinyint NOT NULL DEFAULT 0,
  gender char(1) NOT NULL DEFAULT '',
  ...);
```

However, do not take this as a limitation. In some cases, it is convenient to use NULL to indicate the absence of a value. For example:

```
CREATE TABLE `users` (
  ...
  signuped_at datetime NULL DEFAULT NULL,
  ...);
```

3.3. INTEGER NUMBERS

For all numerical columns, be sure to calculate the maximum value. There are 5 integer types in Mysql:

- TINYINT: 8 bit, max 127
- SMALLINT: 16 bit, max 32 676
- MEDIUM: 24 bit, max 8 388 607
- INT: 32 bit, max 2 147 483 647 (2 x 109)
- BIGINT: 64 bit, max 9 x 1018

Imagine that you are using the INT type for a column that stores the user's age. Then, as the tinyint type is enough for you, you use 32-8=24 bit more. For each line. If you have 10 thousand users, you are wasting your time: 24/8*10000=30KiB. If there are 10 million users, then 30MiB.

Choose the minimum data type based on the maximum column value. It may not be so much for a disk, but it is critical for RAM.

```
CREATE TABLE `users` (
  ...
  age TINYINT NOT NULL DEFAULT 0,
  ...);
```

3.4. Unsigned numbers

If a negative number is not relevant for the column, use UNSIGNED values. Then the max value will be twice as large, but the min will be zero:

- UNSIGNED TINYINT: 8 bit, max 255
- UNSIGNED SMALLINT: 16 bit, max 65 535
- UNSIGNED MEDIUM: 24 bit, max 16 x 106
- UNSIGNED INT: 32 bit, max 4 x 109
- UNSIGNED BIGINT: 64 bit, max 18 x 1018

3.5. Numeric Type Length

In Mysql, you can specify the column length after specifying a numeric type:

```
CREATE TABLE `users` (
  ...
  phone INT(7) NOT NULL DEFAULT 0,
  ...);
```

This has no effect on either the column size or the maximum number. Just never use length for numeric types.

3.6. Decimal numbers

To store real numbers, MySQL suggests using the DECIMAL type:

```
CREATE TABLE `planets` (
  ...
  distance_to_sun DECIMAL(40,3) NOT NULL DEFAULT 0,
  ...);
```

The number of digits in total and their number after the decimal point (may be zero) are indicated in parentheses. Since processors do not support mathematical operations with similar numbers, Mysql does all the calculations on its side. **So that’s very slow**.

3.7. Float / Double

Unlike DECIMAL, the type FLOAT is approximate (stores an inaccurate number). At the same time, the processor can work with this type directly. In addition, FLOAT takes up less space than DECIMAL for storing the same values. Use FLOAT / DOUBLE instead of DECIMAL if you don't need very accurate numbers.

© Yuriy Shamshin, 2020

3.8. VARCHAR / CHAR

When choosing string types, the minimum rule also applies. Estimate the maximum string length and set a limit. **If the values in the text column are similar in length, select CHAR, otherwise - VARCHAR.**

Type CHAR - type of fixed length. This means that for any row the same number of bytes will always be allocated:

```
CREATE TABLE `planets` (
  ...
  state CHAR(2) NOT NULL DEFAULT '', # a place will always be allocated for 2 characters (even if its value is empty)
  ...);
```

VARCHAR - type of variable length. In such a column, the string will occupy exactly its length (in the number of characters). However, Mysql will add another 1 or 2 bytes to store the length of the string itself:

```
CREATE TABLE `planets` (
  ...
  first_name VARCHAR(32) NOT NULL DEFAULT '', # the column will contain from 1 to 32 characters depending on the value
  ...);
```

3.9. TEXT / BLOB

Types TEXT and BLOB differ from each other only in that for the second type Mysql does not do encoding conversions (stores as it is).

Do not use TEXT / BLOB types for sorting columns.

3.10. ENUM

String values can be one of the previously known values (for example, a list of countries). If the list is fixed (new countries do not appear every day), it will be convenient to use the ENUM type. It allows you to specify a list of values and use only them in the rows of the table.

```
CREATE TABLE users(
...
      region ENUM('EU', 'USA', 'ASIA') NOT NULL
...);
```

The advantage of this type is that it writes the value number instead of the value itself on each line. This provides huge space savings.

© Yuriy Shamshin, 2020

3.11. DATETIME / TIMESTAMP

Both date formats allow you to store date and time values up to seconds. However, there are differences between them:

- DATETIME occupies 8 bytes and allows you to store dates from 1001 to 9999.
- TIMESTAMP takes 4 bytes and allows you to store dates from 1970 to 2038.

Use the TIMESTAMP format to set the dates of events (which is why it was created). For example, the time of user registration or posting a comment.

Data Type	“Zero” Value
DATE	'0000-00-00'
TIME	'00:00:00'
DATETIME	'0000-00-00 00:00:00'
TIMESTAMP	'0000-00-00 00:00:00'
YEAR	0000

© Yuriy Shamshin, 2020

4. Data Integrity and Constraints

4.1. Data Integrity Rules

Data integrity is one of the cornerstones of the relational model. Simply stated data integrity means that the data values in the database are correct and consistent. You should also apply the integrity rules to check the integrity of your design:

Attribute Integrity Rule: Although not part of the relational model, most database software enforce attribute integrity through the use of domain information.

A domain is a valid set of values for an attribute which enforce that values from an insert or update make sense. Each attribute in the model should be assigned domain information which includes:

- Data Type—Basic data types are integer, decimal, or character. Most data bases support variants of these plus special data types for date and time.
- Length—This is the number of digits or characters in the value. For example, a value of 5 digits or 40 characters.
- Date Format—The format for date values such as dd/mm/yy or yy/mm/dd.
- Range—The range specifies the lower and upper boundaries of the values the attribute may legally have.
- Constraints—Are special restrictions on allowable values. For example, the Beginning_Pay_Date for a new employee must always be the first work day of the month of hire.
- Null support—Indicates whether the attribute can have null values.
- Default value (if any)—The value an attribute instance will have if a value is not entered.

Entity Integrity Rule: The primary key cannot contain NULL. Otherwise, it cannot uniquely identify the row. For composite key made up of several columns, none of the column can contain NULL. Most RDBMS check and enforce this rule.

Referential Integrity Rule: Each foreign key value must be matched to a primary key value in the table referenced (or parent table). Most RDBMS can be setup to perform the check and ensure the referential integrity.

- You can insert a row with a foreign key in the child table only if the value exists in the parent table.
- If the value of the key changes in the parent table (e.g., the row updated or deleted), all rows with this foreign key in the child table(s) must be handled accordingly. You could either (a) disallow the changes; (b) cascade the change (or delete the records) in the child tables accordingly; (c) set the key value in the child tables to NULL.

Business logic Integrity: Beside the above two general integrity rules, there could be integrity (validation) pertaining to the business logic, e.g., zip code shall be 5-digit within a certain ranges, delivery date and time shall fall in the business hours; quantity ordered shall be equal or less than quantity in stock, etc. These could be carried out in validation rule (for the specific column) or programming logic.

4.2. Constraints

Constraints enforce limits to the data or type of data that can be inserted/updated/deleted from a table. The whole purpose of constraints is to maintain the **data integrity** during an update/delete/insert into a table.

Types of constraints

1. Data Type and Length (The length is the number of digits or characters in the value. For example, a value of 5 digits or 40 characters.)
2. NULL (Indicates whether the attribute can have null values)
3. UNIQUE
4. AUTO_INCREMENT
5. DEFAULT (if any, the value an attribute instance will have if a value is not entered)
6. CHECK and Range (The range specifies the lower and upper boundaries of the values the attribute may legally have)
7. PRIMARY KEY
8. FOREIGN KEY
9. Domain constraints
10. Triggers

4.3. Null / Not Null

NOT NULL constraint makes sure that a column does not hold NULL value. When we don't provide value for a particular column while inserting a record into a table, it takes NULL value by default. By specifying NULL constraint, we can be sure that a particular column(s) cannot have NULL values.

```
CREATE TABLE STUDENTS(
std_ID INT NOT NULL,
stdName VARCHAR (35) NOT NULL,
stdAge INT NOT NULL,
stdAddress VARCHAR (235),
PRIMARY KEY (std_ID)
);
```

4.4. Unique

UNIQUE Constraint enforces a column or set of columns to have unique values. If a column has a unique constraint, it means that particular column cannot have duplicate values in a table.

```
CREATE TABLE STUDENTS(
std_ID INT NOT NULL,
stdName VARCHAR (35) NOT NULL UNIQUE,
stdAge INT NOT NULL,
stdAddress VARCHAR (235) UNIQUE,
PRIMARY KEY (std_ID)
);
```

The UNIQUE constraint creates an index such that, all values in the index column must be unique.

4.5. AUTO_INCREMENT

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table. Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

```
CREATE TABLE STUDENTS(
std_ID INT NOT NULL AUTO_INCREMENT,
stdName VARCHAR (35) NOT NULL UNIQUE,
stdAge INT NOT NULL,
stdAddress VARCHAR (235) UNIQUE,
PRIMARY KEY (std_ID)
);
```

By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record. To let the AUTO_INCREMENT sequence start with another value, use the following SQL statement:

```
ALTER TABLE STUDENTS AUTO_INCREMENT=100;
```

4.6. DEFAULT

The DEFAULT constraint provides a default value to a column when there is no value provided while inserting a record into a table.

```
CREATE TABLE STUDENTS(
std_ID INT NOT NULL,
stdName VARCHAR (35) NOT NULL,
```

```
stdAge INT NOT NULL,
ExamFee INT DEFAULT 10,
stdAddress VARCHAR (235),
PRIMARY KEY (std_ID)
);
```

4.7. Check and Range

The CHECK constraint is used for specifying range of values for a particular column of a table. When this constraint is being set on a column, it ensures that the specified column must have the value falling in the specified range.

```
CREATE TABLE STUDENTS(
std_ID INT NOT NULL,
stdName VARCHAR (35) NOT NULL,
stdAge INT NOT NULL,
ExamFee INT DEFAULT 10 CHECK(ExamFee > 2),
stdAddress VARCHAR (235),
PRIMARY KEY (std_ID)
);
```

4.8. Primary Key

Primary key uniquely identifies each record in a table. It must have unique values (cannot have duplicate) and cannot contain nulls.

```
CREATE TABLE ORDERS(
Customer_ID INT NOT NULL,
Product_ID INT NOT NULL,
OrderQuantity INT NOT NULL,
PRIMARY KEY (Customer_ID, Product_ID)
);
```

When we have only one attribute as primary key, like we see in the first example of STUDENTS table. we can define the key like this as well:

```
CREATE TABLE STUDENTS(
std_ID INT NOT NULL PRIMARY KEY,
```

© Yuriy Shamshin, 2020

```
stdName VARCHAR (35) NOT NULL UNIQUE,
stdAge INT NOT NULL,
stdAddress VARCHAR (235) UNIQUE
);
```

4.9. Foreign Key

Example: Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

```
CREATE TABLE COUNTRYS(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR (128) NOT NULL UNIQUE
);

CREATE TABLE REGIONS(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR (128) NOT NULL UNIQUE,
  country_id int NOT NULL,
  FOREIGN KEY (country_id) REFERENCES countrys (id)       # No Action is equivalent to RESTRICT
);
CREATE TABLE CITYS(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR (128) NOT NULL UNIQUE,
  region_id int NOT NULL,
  FOREIGN KEY (region_id) REFERENCES regions (id)
  ON UPDATE CASCADE ON DELETE RESTRICT
);
```

In **COUNTRYS** table, **id** is the primary key which is set as foreign key in **REGIONS** table. The value that is entered in **country_id** which is set as foreign key in **REGIONS** table must be present in **COUNTRYS** table where it is set as primary key. This prevents invalid data to be inserted into **country_id** column of **REGIONS** table. If you try to INSERT any incorrect data, DBMS will return error and will not allow you to insert the data.

© Yuriy Shamshin, 2020

4.9.1. Behaviour of Foriegn Key Column on INSERT/DELETE/UPDATE.

A foreign key creates a hierarchical relationship between two associated entities. The entity containing the FK is the child, or dependent, and the table containing the PK from which the FK values are obtained is the parent. In order to maintain referential integrity between the parent and child as data is inserted or deleted/updated from the database certain insert and delete/update rules must be considered.

Insert Rules:

- **Dependent**. The dependent insert rule permits insertion of child entity instance only if matching parent entity already exists.
- **Automatic**. The automatic insert rule always permits insertion of child entity instance. If matching parent entity instance does not exist, it is created.
- **Nullify**. The nullify insert rule always permits the insertion of child entity instance. If a matching parent entity instance does not exist, the foreign key in child is set to null.
- **Default**. The default insert rule always permits insertion of child entity instance. If a matching parent entity instance does not exist, the foreign key in the child is set to previously defined value.
- **Customized**. The customized insert rule permits the insertion of child entity instance only if certain customized validity constraints are met.
- **No Effect**. This rule states that the insertion of child entity instance is always permitted. No matching parent entity instance need exist, and thus no validity checking is done.
- **Error**. If we don't use any of the above, then we cannot insert data in the child table from for which data in the main table not exists. We will get an error if we try to do so.

Delete/Update Rules:

- **Restrict**. The restrict delete rule permits deletion of parent entity instance only if there are no matching child entity instances.
- **Cascade**. The cascade delete rule always permits deletion of a parent entity instance and deletes all matching instances in the child entity.
- **Nullify**. The nullify delete rules always permits deletion of a parent entity instance. If any matching child entity instances exist, the values of the foreign keys in those instances are set to null.
- **Default**. The default rule always permits deletion of a parent entity instance. If any matching child entity instances exist, the value of the foreign keys are set to a predefined default value.
- **Customized**. The customized delete rule permits deletion of a parent entity instance only if certain validity constraints are met.
- **No Effect**. The no effect delete rule always permits deletion of a parent entity instance. No validity checking is done.
- **Error**. If we don't use any of the above, then we cannot delete/update data from the main table for which data in child table exists. We will get an error if we try to do so.

4.10. Domain constraints

Each table has certain set of columns and each column allows a same type of data, based on its data type. The column does not accept values of any other data type. Domain constraints are **user defined data type** and we can define them like this:

Domain Constraint = data type + Constraints (NOT NULL / UNIQUE / PRIMARY KEY / FOREIGN KEY / CHECK / DEFAULT)

For example I want to create a table “student_info” with “stu_id” field having value greater than 100, I can create a domain and table like this:

```
create domain id_value int
constraint id_test
check(value > 100);

create table student_info (
stu_id id_value PRIMARY KEY,
stu_name varchar(30),
stu_age int
);
```

4.11. Triggers

In SQL, a trigger is a stored program invoked automatically in response to an event such as insert, update, or delete that occurs in the associated table. For example, you can define a trigger that is invoked automatically before a new row is inserted into a table.

The SQL standard defines two types of triggers: row-level triggers and statement-level triggers.

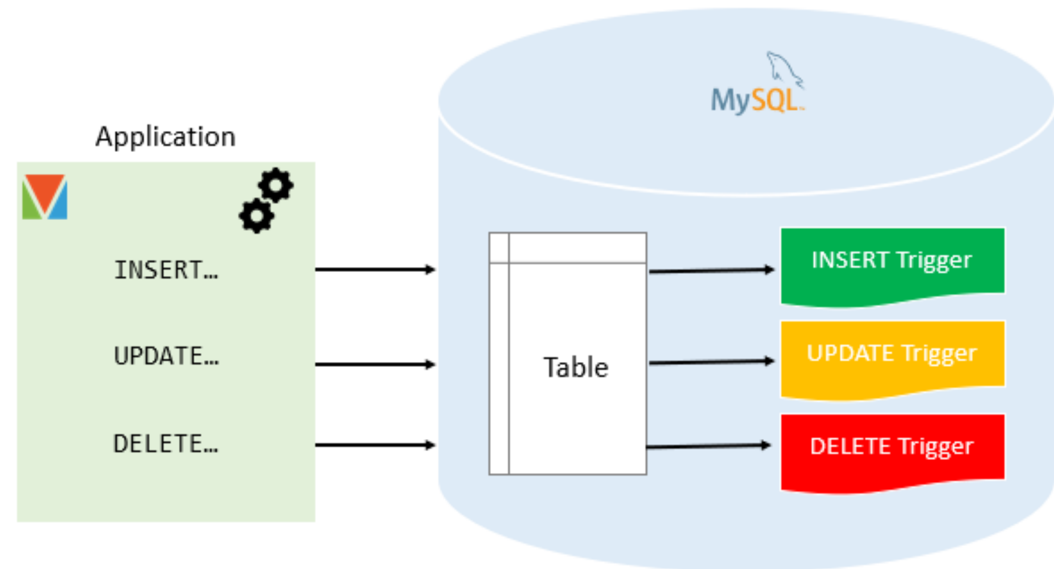
- A row-level trigger is activated for each row that is inserted, updated, or deleted. For example, if a table has 100 rows inserted, updated, or deleted, the trigger is automatically invoked 100 times for the 100 rows affected.

- A statement-level trigger is executed once for each transaction regardless of how many rows are inserted, updated, or deleted.

MySQL supports only row-level triggers. It doesn’t support statement-level triggers.

Here is the basic syntax of the Create Trigger statement:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON table_name FOR EACH ROW
trigger_body;
```



Advantages of triggers

- Triggers provide another way to check the integrity of data.
- Triggers handle errors from the database layer.
- Triggers give an alternative way to run scheduled tasks. By using triggers, you don’t have to wait for the scheduled events to run because the triggers are invoked automatically *before* or *after* a change is made to the data in a table.
- Triggers can be useful for auditing the data changes in tables.

Disadvantages of triggers

- Triggers can only provide extended validations, not all validations. For simple validations, you can use other constraints.
- Triggers can be difficult to troubleshoot because they execute automatically in the DB, which may not invisible to the client App.
- Triggers may increase the overhead of the MySQL Server.

5. INDEXES

5.1. COLUMN INDEXING

An index is a structured file that speeds up data access for SELECT, but may slow down INSERT, UPDATE, and DELETE.

- **Without** an index structure, to process a SELECT query with a matching criterion (e.g., SELECT * FROM Customers WHERE customerName ='Bill Gates') the database engine needs to compare **every records** in the table.
- A **specialized** index (e.g., in BTREE structure) could reach the record **without comparing every records**.
- However, the **index** needs to be **rebuilt** whenever a record is changed, which results in overhead associated with using indexes.
- Most RDBMS builds index on the primary key **automatically**.

Index can be defined on a single column, a set of columns (called concatenated index), or part of a column (e.g., first 10 characters of a VARCHAR(100)) (called partial index). You could built more than one index in a table.

For example, if you often search for a customer using either customerName or phoneNumber, you could speed up the search by building an index on column customerName, as well as phoneNumber.

5.2. WHEN TO CREATE INDEXES?

- Indexes should be created as slow queries are detected. This will help slow-log in MySQL. Requests that run more than 1 second are the first candidates for optimization.
- Start creating indexes with the most common queries. A query that runs a second, but 1000 times a day does more damage than a 10-second query that runs several times a day.
- Do not create indexes on tables with fewer than a few thousand records. For such sizes, the gain from using the index will be almost invisible.
- Do not create indexes for the future, for example, in a development environment. Indexes should be set exclusively for the form and type of load of a working system.
- Delete unused indexes.

© Yuriy Shamshin, 2020