**DASC 5333**

9/24/2024

**Transforming UML Class Diagrams to Relational Models**

by K. Yue

**1. Transforming OO Model to the Relational Model**

* Once a *conceptual* OO data model (UML) is constructed, it needs to be mapped for implementation in the selected *logical* database model. See [IntroDataModeling.html](https://dcm.uhcl.edu/yue/courses/joinDB/Fall2024/notes/model/IntroDataModeling.html).
* If a relational DB is used, the mapping will be from OO (classes, attributes, associations, etc.) to relational schema (relations, attributes, keys, etc.)
* Note that the relational model and the OO model are *very different*, even though diagrams representing the two models look similar (there are only a finite number of common suitable shapes).
* Computer-Aided Software Engineering (CASE) tools or DB modeling tools may provide varying degrees of facilities for automatic generating relational schema and corresponding SQL statements.
* However, it is important to understand the mechanism that a tool uses to generate the relational schema and make adjustment if needed.

**1.1 Model Transformation**

The problem is: Source Model --> Target Model.

* Source Model: UML
	1. Basic elements:
		1. Class
		2. Attribute: can be multi-valued.
		3. Association
	2. Secondary elements:
		1. Object
		2. Multiplicity
		3. Data type
		4. Default value
		5. Constraint
		6. Stereotype for RDB extensions: e.g., candidate keys, primary keys, unique, derived, nullability, etc.
		7. ...
* Target Model: Relational Model
	1. Basic elements:
		1. Relation
		2. Attribute (column/field): It should be single-valued (atomic). Otherwise, first normal form (1NF) is not satisfied.
	2. Secondary elements:
		1. Row (tuple)
		2. Data type
		3. Nullability
		4. Constraint
		5. Candidate key
		6. Primary key
		7. Foreign key
		8. Index
		9. ...
* There are only two basic elements in the targeted model to consider. Data in the relational model can be stored only in two ways:
	1. relation:
		1. more flexible,
		2. can hold attributes to store a collection of logically related data
		3. more complex.
	2. attribute:
		1. should be single-valued (if good design, i.e., the first normal form, is to be assured.)
		2. Simple
		3. Thus, if attributes are sufficient in model transformation, they are preferred.
* Different RDBMS provide different features.
	+ 1. Thus, the targeted RDB model is not universal.
		2. It is necessary to define vendor-specific transformation rules.

**2. Transformation Rules**

* We present a set of mapping rules below. It is not meant to be complete or universal.
* Examples of comparable transformation rules:
	+ A relative simple one: <http://web.fe.up.pt/~ssn/2010/lbaw/slides/lbaw-uml2rel.eng.pdf> a
	+ A more elaborated one based on agile methodology: [http://www.agiledata.org/essays/mappingObjects.html](http://www.agiledata.org/essays/mappingObjects.html#Figure2IncludingShadowInformation).
* Do not mechanically follow these rules. Instead, understand the rationale behind the rules and adapt.
* All OO model details should be implemented in the targeted model in some ways.
	+ database level: preferred.
	+ middle layer level
	+ application level

**2.1 Classes**

**C1**. A class C is mapped or transformed to a relation RC.

1. Relations may later be merged and/or reorganized in design refinement and performance tuning.
2. The relation may use the same name as the class.
3. As a result, all information of an object of class C is stored as a row in RC.

**Rationale:**

1. A class is a logical unit for encapsulating related data and a relation has the same property.

**2.2 Attributes**

**Basic:**

**ATT1. Single-Valued Attributes.**Map all *single*-valued attributes (with simple data types) of a class C as attributes of RC, the relation for the class C.

**ATT2**. **Multi-Valued Attributes.** For *each* *multi-valued* attribute A, Email, of the class C, Student, create a *new* relation RCA (StudentEmail) containing the attribute A (Email) and the primary key, RCId (Sid), of the relation RC (which implements the class C).

1. (RCId, A) is a composite candidate key.
2. RCId (Sid) is a foreign key referencing RC(RCId), Student(Sid)
3. A surrogate key, such as RCA\_Id, may be created to serve as the *simple* primary key.
4. The name of RCA should be meaningfully selected.

Example:



Student object 1:

1. Sid: 100
2. LName: Yue
3. Emails: yue@uhcl.edu, yue@gmail.com
4. Organizations: IEEE, ACM, AIS

Relation student:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sid (PK)** | **LName** | Emails? Not good |  |  |
| 100 | Yue | yue@uhcl.edu, yue@gmail.comNot single-valued. |  |  |
|  |  |  |  |  |



Good enough: 3 columns: Email1, Email2, Email3.

:

Cannot just use a fixed number of columns.

A new table: StudentEMail

|  |  |  |
| --- | --- | --- |
| **Sid (FK)** | **Email (single-valued)** | **SEId** |
| 100 | yue@uhcl.edu  | 1 |
| 100 | yue@gmail.com | 2 |
|  |  | 3 |

Student Organizations:

|  |  |
| --- | --- |
| **SID** | **Organizations** |
| 100 | IEEE |
| 100 | ACM |
| 100 | AIS |
| 200 | IEEE |

**Secondary:**

**ATT3.** **Zero in Multiplicity.** If the multiplicity of an attribute is specified, to handle the case of 0:

1. If 0 is allowed in the UML model (e.g., 0..1, 0..\* in the UML class diagram), the attribute is nullable. Add the NULL specifier in the column definition in the RDBMS. (NULL is usually the default)
2. If 0 is not allowed, add the NOT NULL specifier in the column definition.

**ATT4. Default Values.**The default value of an attribute can be directly implemented in SQL DDL.

**ATT5. Data Types.**Data type mapping should be handled adequately, effectively, and consistently.

1. In later modeling phases, one may use SQL data types of the targeted DBMS in the class diagram.
2. If available, consider using user-defined data types in the targeted DBMS.

***Example:***

UML for toyu.student:



Relational schema (in HW assignment format):

|  |  |
| --- | --- |
| **Relation** | Student(*StuId, fname, lname*, major, minor, *ach*, advisor) |
| Candidate Keys | [1] StuId |
| Foreign Keys | [1] major references department(deptCode), [2] minor references department(deptCode), [3] advisor references faculty(facId) |
| Nullable Attributes | major, minor, advisor, *ach* |
| Non-nullable Attributes | stuId, fname, lname |
| Notes |   |

Relational schema in SQL (with more implementation details):

CREATE TABLE IF NOT EXISTS Student    (
    stuId        INT NOT NULL,
    fname        VARCHAR(30) NOT NULL,
    lname        VARCHAR(30) NOT NULL,
    major        CHAR(4) NULL,
    minor        CHAR(4) NULL,
   -- ach: accumulated credit hours, including transferred credits.
    ach          INTEGER(3) NULL DEFAULT 0,
    advisor      INT NULL,
    CONSTRAINT Student\_stuId\_pk PRIMARY KEY(stuId),
   -- an artificial example of a CHECK constraint.
    CONSTRAINT Student\_ach\_cc CHECK ((ach>=0) AND (ach < 250)),
    CONSTRAINT Student\_major\_fk FOREIGN KEY (major)
        REFERENCES Department(deptCode) ON DELETE CASCADE,
    CONSTRAINT Student\_minor\_fk FOREIGN KEY (minor)
        REFERENCES Department(deptCode) ON DELETE CASCADE,
    CONSTRAINT Student\_advisor\_fk FOREIGN KEY (advisor)
        REFERENCES Faculty(facId)
);

***Example:***

Multi-valued attributes: consider the class Member with the following attributes:

1. Member\_Id: <<PK>>
2. Screen\_Name <<unique>>
3. Hobbies[0..\*]
4. Medals[0..\*]



Reasonable relation schema: three relations used

1. Member(MemberId, ScreenName):
	1. CK: [1] MemberId, [2] *ScreenName*
2. Hobby(HobbyId, MemberId, Hobby):
	1. CK: [1] HobbyId, [2] MemberId, Hobby
	2. FK: [1] MemberId references Member(MemberId)
	3. A surrogate key, HobbyId, is created as the primary key.
3. Medal(MedalId, MemberId, Medal):
	1. CK: [1] MedalId, [2] MemberId, Medal.
	2. FK: [1] MemberId references Member(MemberId)
	3. A surrogate key, MedalId, is created as the primary key.

All columns in the tables above are not nullable.

**ATT6. Single-Valued Composite Data Types.** A single-valued attribute of a *composite* data type (such as set, list, array) can be mapped in various ways.

1. If there is a comparable composite data type in the targeted DBMS, it can be implemented as an attribute of that data type in the relation.
	* The relation will no longer be in the first normal form.
	* Care should be taken in handling the difference in data type mapping.
2. Otherwise, regard the attribute as a multi-valued attribute and apply rule ATT2.

**ATT7. Derived Attributes.** For a derived attribute A:

1. It can be implemented and stored as an attribute of the relation.
	* Mechanisms, such as triggers or stored procedures, should be used to ensure data consistency. The derived column should be consistent with the data that derives its value.
2. It may not be stored as a column directly in any relation.
	* Mechanisms, such as virtual columns, views or stored functions, may be used to provide standard access to the derived attributes.

***Example:***

A class Rectangle has three attributes:

* Length
* Width
* \Area or <<derived>>: derived.



What may the relational schema look like?

One solution:

Rectangle(RectangleId, Length, Width)

with a view Rect define as

SELECT DISTINCT RectangleId, Length, Width, Length \* Width as Area
FROM Rectangle;

Alternatively, use*virtual* columns: a column that is computed in real-time by an expression and not stored.

CREATE or replace TABLE rectangle (
  width DOUBLE,
  height DOUBLE,
  area DOUBLE AS (width \* height) virtual
);

**2.3 Keys and Constraints**

**KC1. Primary Key Identified.** If a relation R implements a class C or an association (class) AC, and C or AC has identified the PK, set it as the primary key of R.

**KC2. Candidate Keys.** If a relation R implements a class C or an association (class) AC, and C or AC has candidate keys K's, set **all** K's as candidate keys of R.

* If no primary key has been identified, select a CK to serve as the primary key, and apply KC1.
* A candidate key can be implemented by using the 'unique' and non-null constraint together in SQL.

**KC3. No Primary Key Identified.**If a relation R implements a class C or an association (class) A, and C or AC has no candidate key, create a *surrogate* primary key K for R.

1. This is needed as every relation must have at least one candidate key.

***Example:***

for the class Department in toyu:



Relational schema (in HW assignment format):

|  |  |
| --- | --- |
| **3** | Department(deptCode, deptName, schoolCode, numStaff) |
| Candidate Keys | [1] deptCode, [2] deptName |
| Foreign Keys | [1] schoolCode references School(schoolCode) |
| Nullable Attributes | schoolCode, numStaff |
| Non-nullable Attributes | deptCode, deptName |
| Notes |   |

Relational schema in SQL (with more implementation details):

CREATE TABLE IF NOT EXISTS Department (
    deptCode    CHAR(4) NOT NULL,
    deptName    VARCHAR(30) *NOT NULL,*
    schoolCode  CHAR(3) NULL,
    numStaff  TINYINT NULL,
    CONSTRAINT Department\_deptCode\_pk PRIMARY KEY (deptCode),
   -- alternate keys: [1] deptName
    *CONSTRAINT Department\_name\_ck UNIQUE (deptName),*
    CONSTRAINT Department\_schoolCode\_fk FOREIGN KEY (schoolCode)
        REFERENCES School(schoolCode)
);

**KC4.** For a stereotype:

1. Some may be directly implemented in SQL DDL, e.g., PK, CK, unique, etc.
2. Otherwise, it is necessary to consider where and how it is implemented.

**2.4 Associations**

**A1. Many-to-one Association.** For a many to one association A between C1, Team (the class with the one multiplicity) and Cm, Project, add a column R1\_Id (TeamId) into the relation Rm, Proje t (which implements Cm).

1. Assume that R1\_Id, TeamId, is the primary key of the relation R1 (which implements C1).
2. R1\_Id, TeamId is a foreign key in Rm referencing R1(R1\_Id): team(TeamId)
3. The name R1\_Id may be renamed.
4. R1\_Id is not null in Rm iff (if and only if) 0 is not allowed (i.e., 1..1) for C1.
5. Any single-valued attribute of the association is mapped to a column in Rm.
6. If the association A is an association class, single-valued attributes of A can be stored as attributes of R1.
7. If you have composite or multi-valued attributes of the relationship, you should consider promoting the association to a regular class in your UML class diagram.



Many to one Association from Project to Team.

Team 1 has projects,10 and 11.

|  |  |  |  |
| --- | --- | --- | --- |
| TeamId (PK) | TName | … | ProjectIds? No, not single-valued |
| 1 | DS\_team |  | 10, 11? |
|  |  |  |  |

Project: Cm

|  |  |  |  |
| --- | --- | --- | --- |
| PId | PName | … | TeamId? Yes, FK |
| 10 | Ice cream |  | 1 |
| 11 | Donut |  | 1 |
| … |  |  |  |
| 13 |  |  | 0: not ok |

|  |  |
| --- | --- |
| 3 | Team(TeamId, TName, Description, Since, OwnerTRMemberId) |
| Candidate Keys | [1] TeamId |
| Foreign Keys | [1] OwnerTRMemberId references TRMember(TRMemberId) |
| Nullable Attributes | Description |
| Non-nullable Attributes | TeamId, TName, Since, OwnerTRMemberId |
| Notes |  |

|  |  |
| --- | --- |
| 12 | Project(ProjectId, PName, Description, ExpCompTime, ActCompTime, CurrentStatusId, ParentProjectId, TeamId, CreatorManagerId, AssigneeTMId) |
| Candidate Keys | [1] ProjectId |
| Foreign Keys | [1] CurrentStatusId references Status(StatusId), [2] CreatorManagerId references Manger(ManagerId), [3] AssigneeTMId references TeamMember(TMId), [4] ParentProjectId references Project(ProjectId), [5] TeamId references Team(TeamId) |
| Nullable Attributes | ParentProjectId, AssigneeTMId |
| Non-nullable Attributes | ProjectId, PName, Description, ExpCompTime, ActCompTime, TeamId, CreatorManagerId, CurrentStatusId |
| Notes |  |

***Example:***

For:



We have the three numbered associations implemented by the three foreign keys below.

|  |  |
| --- | --- |
| **4** | Faculty(facId, fname, lname, *deptCode*, rank) |
| Candidate Keys | [1] facId |
| Foreign Keys | [1] deptCode references Department(deptCode) |
| Nullable Attributes | rank, *deptCode* |
| Non-nullable Attributes | facId, fname, lname |
| Notes |   |
| **6** | Class(classId, *courseId*, semester, year, *facId*, room) |
| Candidate Keys | [1] classId |
| Foreign Keys | [1] courseId references Course(courseId), [2] facId references Faculty(facId) |
| Nullable Attributes | room |
| Non-nullable Attributes | classId, *courseId*, semester, year, *facId* |
| Notes |   |

**A2. Many-to-many Association.** For a many-to-many association (including association classes) between classes CA, TeamMember and CB Role, create a*new* relation RAB TeamMemberRold(RA\_Id: TMId, RB\_Id: RoleId).

1. (RA\_Id, RB\_Id) (TMId, RoleId) is a candidate key.
2. RA\_Id references RA(RA\_Id) as a foreign key.
3. RB\_Id references RB(RB\_Id) as a foreign key.
4. An additional surrogate key, such as RAB\_Id, can be created.



TM M1 roles R1 and R2
TM M2 roles R1 and R3.

Role

|  |  |  |
| --- | --- | --- |
| **RoleId** | **RLNum** | TMIds? No, not single-valued |
| R1 | 2 | M1, M2 |
| R2 | 3 |  |
| R3 | 5 |  |

TeamMember:

|  |  |  |  |
| --- | --- | --- | --- |
| TMId | … | RoleIds? No, not single-valued |  |
| M1 |  | R1, R2 |  |
| M2 |  | R1, R3 |  |
|  |  |  |  |

New Table:

|  |  |
| --- | --- |
| 10 | TeamMemberRole(TMRId, TMId, RoleId) |
| Candidate Keys | [1] TMRId, [2] TMId, RoleId |
| Foreign Keys | [1] TMId references TeamMember(TMId), [2] RoleId references Role(RoleId) |
| Nullable Attributes |  |
| Non-nullable Attributes | TMRId, TMId, RoleId |
| Notes | [1] TMRId is created as a surrogate primary key. |

T

|  |  |  |
| --- | --- | --- |
|  | **TMId (FK)** | **RoleId (FK)** |
|  | M1 | R1 |
|  | M1 | R2 |
|  | M2 | R1 |
|  | M2 | R3 |
|  |  |  |

|  |  |
| --- | --- |
| 7 | Role(RoleId, RLNum) |
| Candidate Keys | [1] RoleId |
| Foreign Keys | [1] RLNum references RoleLevel(RLNum) |
| Nullable Attributes |  |
| Non-nullable Attributes | RoleId, RLNum |
| Notes | [1] RoleId is created as a surrogate primary key. [2] We used three relations to implement the three classes Role, UserDefinedRole and StandardRole. It is possible to use only one relation. |

|  |  |
| --- | --- |
| 5 | TeamMember(TMId, TRMemberId, TeamId, JointTime) |
| Candidate Keys | [1] TMId, [2] TRMemberId, TeamId, JointTime |
| Foreign Keys | [1] TRMemberId references TRMember(TRMemberId), [2] TeamId references Team(TeamId) |
| Nullable Attributes |  |
| Non-nullable Attributes | TMId, TRMemberId, TeamId, JointTime |
| Notes | [1] TMId is created as a surrogate primary key. |

***Example:***

For:



We have:

|  |  |
| --- | --- |
| **8** | Enroll(stuId, classId, grade, n\_alerts) |
| Candidate Keys | [1] stuId, classId |
| Foreign Keys | [1] stuId references Student(stuId), [2] classId references Class(classId), [3] grade references Grade(grade) |
| Nullable Attributes | grade, n\_alerts |
| Non-nullable Attributes | stuId, classId |

**A3. One-to-one Association.** For a one to one association between classes CA and CB, there are several options:

1. Treat CA as C1 and CB as Cm and apply A1.
2. Treat CA as Cm and CB as C1 and apply A1.
3. Merge the two relations RA and RB into one. (In this case, you may want to refactor the class diagram.)

**A4. N-ary Associations.** For any n-ary association (n>2), a new relation is needed.

1. You should consider using binary associations instead.
2. A ternary association can be modeled as a regular class with three binary associations with the participating classes in the ternary association.

***Example:***

Consider the ternary association between the classes Supplier, Part, and Warehouse with an association attribute quantity.

A supplier S1 supplies 50 Part P1 to Warehouse W1.

It can reasonably be replaced by a new class and three binary associations.



Supply(SupplyId, SupplierId, PartId, WarehouseId, Quantity):

1. CK: [1] SupplyId, [2] SupplierId, PartId, WarehouseId
2. FK: [1] SupplierId references Supplier(SupplierId), [2] PartId references Part(PartId), [3] WarehouseId references Warehouse(WarehouseId).
3. All attributes in Supply is not nullable.

***Example:***

Checkout the UML diagram and relation schema for [toyu](https://dcm.uhcl.edu/yue/courses/joinDB/Fall2024/notes/toyu/toyu.html) and [swim](https://dcm.uhcl.edu/yue/courses/joinDB/Fall2024/notes/swim/swim.html).

**Introduction to SQL and MySQL**

by K. Yue

**1. Introduction**

* SQL (Structured Query Language): defacto standard for relational databases.
* SQL-like languages are also used in non-relational DBMS.
* Contains core specifications and extensions. Latest SQL standard: 2016.
* Not using a *pure* relational model: e.g.
	1. Use the terms row, column, and table instead of tuple, attribute, and relation.
	2. The results may not be a set.
* Mostly based on Tuple Relational Calculus (TRC) and a little on Relational Algebra (RA).
* SQL is mostly declarative.
* DBMS vendor-specific extensions are common.
* SQL Contains:
	1. Data Definition Language (DDL): define the relation schema (structure)
	2. Data Manipulation Language (DML): manipulate data; CRUD:
		1. Create: Insert
		2. Read
		3. Update
		4. Delete
	3. Data Administration Language: for DB administration such as user and security management.

**2. MySQL**

Use [toyu](https://dcm.uhcl.edu/yue/courses/joinDB/Fall2024/notes/toyu/toyu.html), a drastically simplified university, as examples.

**2.1 DDL:**

* Make sure that you are familiar with the core SQL Data Definition Language (DDL) commands. Refer to, for example: <http://www.w3schools.com/sql/default.asp>.
* MariaDB DDL: <https://mariadb.com/kb/en/sql-statements/>
* MySQL DDL manual: <https://dev.mysql.com/doc/refman/8.1/en/sql-statements.html>.
* Basic DDL: some examples
	1. CREATE TABLE
	2. CREATE DATABASE: a database contains a collection of related tables for an application.
	3. CREATE VIEW: a view is a virtual table for users to access a subset of a database.
	4. CREATE INDEX: an index is a data structure to enhance access performance of specific queries.
	5. CREATE PROCEDURE
	6. CREATE FUNCTION
	7. CREATE TRIGGER: a trigger is event-driven procedural code activated by events.
	8. ALTER (No ALTER TRIGGER and ALTER INDEX)
	9. DROP

Example:

BNF:

* <postal-address>, <street-address>: non-terminals; expand to the right hand side of a rule.
* “.”, “.”, “Sr.”: terminals.
* Start the root non-terminal: <postal-address>
* Expand the root to all terminals: cover the statement => syntactically correct.

<postal-address> ::= <name-part> <street-address> <zip-part>

 <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL> | <personal-part> <name-part>

 <personal-part> ::= <first-name> | <initial> "."

 <street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

 <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""

 <opt-apt-num> ::= "Apt" <apt-num> | ""

In MYSQL: BNF standard notation for CFG (Context Free Grammar)

Non-terminals: *select\_expr, table\_references*Terminals: SELECT, ALL, DISTINCT,..
[]: optional
|: or
… 0 or more

SELECT

 [ALL | DISTINCT | DISTINCTROW ]

 [HIGH\_PRIORITY]

 [STRAIGHT\_JOIN]

 [SQL\_SMALL\_RESULT] [SQL\_BIG\_RESULT] [SQL\_BUFFER\_RESULT]

 [SQL\_NO\_CACHE] [SQL\_CALC\_FOUND\_ROWS]

 *select\_expr* [, *select\_expr*]

* Note that in MySQL, DATABASE and SCHEMA having the same meaning. Hierarchy:
	1. MySQL: Database = Schema: contains a collection of tables.
	2. Postgres:
		1. A database contains a collection of schema.
		2. A schema contains a collection of tables.
* Constraints: to implement certain constraints in your data model.
	1. NOT NULL: attributes cannot have an null value.
	2. UNIQUE: KEY; the set of attributes must be unique for each row:
	3. PRIMARY KEY: unique, not null, and used for the physical structure of the relation.
	4. FOREIGN KEY
	5. CHECK: for a Boolean condition on the columns.
	6. DEFAULT: define a default value.
* Some other options:
	1. AUTO INCREMENT: automatic increment an integer if a value is not specified. Used for id.

***Example:***

Experimenting with the CREATE TABLE command. Execute the following code and ensure that you understand the result. For example,

* A temporary table is not persistent. It is created for a SQL client session. Its scope is the client session.

DROP SCHEMA IF EXISTS tinker;
CREATE SCHEMA tinker;
USE tinker;

CREATE TABLE s2
SELECT \* FROM toyu.student;

SELECT \*
FROM s2;

CREATE TEMPORARY TABLE s3
SELECT \* FROM toyu.student;

SELECT \*
FROM s3;

CREATE TABLE s4 LIKE toyu.student;

SELECT \*
FROM s4;

INSERT INTO s4
SELECT \* FROM toyu.student;

SELECT \*
FROM s4;

SHOW TABLES;

-- Note that keys and constraints of student are missing in s2 and S3.
DESC student;
DESC s2;
DESC s3;
DESC s4;

DROP TABLE s2;
DROP TABLE s3;
DROP TABLE s4;

SHOW TABLES;

DROP SCHEMA IF EXISTS tinker;

Column names may include special characters. For example, you cannot use the name 'first name' directly as column name, as spaces are interpreted as separator. You will need special syntax. For example:

1. In MySQL, use back-quote: `first name`
2. In MS SQL Server, use []: [first name]
* For each column, there is a data type and optional specifiers (such as NULL, NOT NULL, default values, etc.)
* Additional constraints and indexes can be defined.
* In general, some important considerations in creating tables:
	1. What are the columns?
	2. What are the data types of the columns?
		+ The right domain: be restrictive.
		+ Performance consideration.
	3. Nullability of columns
	4. Primary key
	5. Candidate keys
	6. Foreign keys: can they be enforced by the selected storage engine?
	7. Indexes: performance tuning.
	8. Additional constraints: check whether they are enforced by the storage engine.

***Example:***

* MySQL only supports foreign key constraint in the InnoDB database engine.
* Older versions of MySQL ignore the 'check' clause.

**2.2 Data types**

* Data types in MySQL are rich: <https://dev.mysql.com/doc/refman/8.1/en/data-types.html>
* Beside simple data types, other noticeable data types:
	1. JSON: JavaScript Object Notation
	2. Spatial: support OpenGIS Geometry Model
	3. BLOB: Binary large object
	4. TEXT: long character strings (VARCHAR is limited to 255, extensible to 64K).

**3. DML**

* Basically declarative.

**3.1 Writing to the DB**

* Basic *update* commands (write):
	1. INSERT
	2. UPDATE
	3. DELETE

INSERT INTO <<table>> [<<columns>>]
VALUES <<expression>>

* If column names are missing, the proper column order during table creation will be used.
* Column names using default values or auto-increment values should not be included in the INSERT statement if they are used.
* NULL and DEFAULT can be used as values in INSERT.
* One may also insert values from a select statement. E.g.

INSERT INTO <<table>> [<<columns>>]
<<select statement>>

* The DELETE statement includes a condition for selecting the rows for deletion.

DELETE FROM <<table>>
WHERE <<condition>>

* The update statement is used to update rows and may have an update condition to identify the rows to be updated.

UPDATE <<table>>
SET <<update assignments>>
[WHERE <<update condition>>]

***Example:***

UPDATE Student
SET major = 'ITEC'
WHERE StuId = 100000;

* The update and delete statements can be used to affect multiple rows so be very careful.

***Example:***

-- All students will be majoring in CSCI
UPDATE Student
SET major = 'CSCI';

* Once changed, the effect is permanent. There is no 'undo' command.

***Example:***

Note the order of the insertions in createtoyu.sql below.

INSERT INTO Grade(grade, gradePoint) VALUES
    ('A',4),('A-',3.6667),('B+',3.3333),('B',3),('B-',2.6667),
    ('C+',2.3333),('C',2),('C-',1.6667),
    ('D+',1.3333),('D',1),('D-',0.6667),('F',0),
   ('P', NULL), ('IP', NULL), ('WX', NULL);

INSERT INTO School(schoolCode, schoolName) VALUES
    ('BUS','Business'),
   ('EDU','Education'),
    ('HSH','Human Sciences and Humanities'),
    ('CSE','Science and Engineering');

INSERT INTO Department(deptCode, deptName, schoolCode, numStaff) VALUES
    ('ACCT','Accounting','BUS',10),
    ('ARTS','Arts','HSH',5),
    ('CINF','Computer Information Systems','CSE',5),
    ('CSCI','Computer Science','CSE',12),
    ('ENGL','English','HSH',12),
    ('ITEC','Information Technology','CSE',4),
    ('MATH','Mathematics','CSE',7);

INSERT INTO Faculty(facId, fname, lname, deptCode, `rank`) VALUES
    (1011,'Paul','Smith','CSCI','Professor'),
    (1012,'Mary','Tran','CSCI','Associate Professor'),
    (1013,'David','Love','CSCI',NULL),
    (1014,'Sharon','Mannes','CSCI','Assistant Professor'),
    (1015,'Daniel','Kim','CINF','Professor'),
    (1016,'Andrew','Byre','CINF','Associate Professor'),
    (1017,'Deborah','Gump','ITEC','Professor'),
    (1018,'Art','Allister','ARTS','Assistant Professor'),
    (1019,'Benjamin','Yu','ITEC','Lecturer'),
    (1020,'Katrina','Bajaj','ENGL','Lecturer'),
    (1021,'Jorginlo','Neymar','ACCT','Assistant Professor');

INSERT INTO Course(courseId, rubric, number, title, credits) VALUES
    (2000,'CSCI',3333,'Data Structures',3),
    (2001,'CSCI',4333,'Design of Database Systems',3),
    (2002,'CSCI',5333,'DBMS',3),
    (2020,'CINF',3321,'Introduction to Information Systems',3),
    (2021,'CINF',4320,'Web Application Development',3),
    (2040,'ITEC',3335,'Database Development',3),
    (2041,'ITEC',3312,'Introduction to Scripting',3),
    (2060,'ENGL',1410,'English I',4),
    (2061,'ENGL',1311,'English II',3),
    (2080,'ARTS',3311,'Hindu Arts',3),
    (2090,'ACCT',3333,'Managerial Accounting',3);

INSERT INTO Class(classId, courseId, semester, year, facId, room) VALUES
    (10000,2000,'Fall',2019,1011,'D241'),
    (10001,2001,'Fall',2019,1011,'D242'),
    (10002,2002,'Fall',2019,1012,'D136'),
    (10003,2020,'Fall',2019,1014,'D241'),
    (10004,2021,'Fall',2019,1014,'D241'),
    (10005,2040,'Fall',2019,1015,'D237'),
    (10006,2041,'Fall',2019,1019,'D217'),
    (10007,2060,'Fall',2019,1020,'B101'),
    (10008,2080,'Fall',2019,1018,'D241'),
    (11000,2000,'Spring',2020,1011,'D241'),
    (11001,2001,'Spring',2020,1012,'D242'),
    (11002,2002,'Spring',2020,1013,'D136'),
    (11003,2020,'Spring',2020,1016,'D217'),
    (11004,2061,'Spring',2020,1018,'B101');

INSERT INTO Student(stuId, fname, lname, major, minor, ach, advisor) VALUES
    (100000,'Tony','Hawk','CSCI','CINF',40,1011),
    (100001,'Mary','Hawk','CSCI','CINF',35,1011),
    (100002,'David','Hawk','CSCI','ITEC',66,1012),
    (100003,'Catherine','Lim','ITEC','CINF',20,NULL),
    (100004,'Larry','Johnson','ITEC',NULL,66,1017),
    (100005,'Linda','Johnson','CINF','ENGL',13,1015),
    (100006,'Lillian','Johnson','CINF','ITEC',18,1016),
    (100007,'Ben','Zico',NULL,NULL,16,NULL),
    (100008,'Bill','Ching','ARTS',NULL,90,NULL),
    (100009,'Linda','King','ARTS','CSCI',125,1018),
   (100111,'Cathy','Johanson',NULL,NULL,0,1018);

INSERT INTO Enroll(stuId, classId, grade, n\_alerts) VALUES
    (100000,10000,'A',0),
    (100001,10000,NULL,NULL),
    (100002,10000,'B-',3),
    (100000,10001,'A',2),
    (100001,10001,'A-',0),
    (100000,10002,'B+',1),
    (100002,10002,'B+',2),
    (100000,10003,'C',0),
    (100002,10003,'D',4),
    (100004,10003,'A',0),
    (100005,10003,NULL,NULL),
    (100000,10004,'A-',1),
    (100004,10004,'B+',NULL),
    (100005,10004,'A-',0),
    (100006,10004,'C+',NULL),
    (100005,10005,'A-',0),
    (100006,10005,'A',NULL),
    (100005,10006,'B+',NULL),
    (100007,10007,'F',4),
    (100008,10007,'C-',0),
    (100007,10008,'A-',0),
    (100000,11001,'D',4);

Note the explicit use of NULL, which is a keyword in SQL.

***Example:***

Execute the following code and ensure that you understand the result.

INSERT INTO student VALUES
   (100010,'Bun','Yue',null,null,50,null),
   (100011,'Paul','Harris','CSCI','ITEC',23,1015);

SELECT \* FROM student;

INSERT INTO student VALUES
   (100010,'Bun','Yue',null,null,50,null),
   (100011,'Paul','Harris','CSCI','ITEC',23,1015);

INSERT INTO student VALUES
   (100020,'Bunno','Yue','GEOG',null,50,null);
INSERT INTO student VALUES
   (100021,'Bunna','Yue',null,'GEOG',50,null);
INSERT INTO student VALUES
   (100022,'Bunno','Yue',null,null,50,8888);

-- Remove the two new rows.
DELETE FROM Student
WHERE stuId = 100010 OR stuId = 100011;

SELECT \* FROM student;

**3.2 Querying with the SELECT Statement**

* SELECT is the basic data retrieval statement in SQL
* Not to be confused with the select statement in Relational Algebra (RA).
* Basic format, with*conceptual* steps.

SELECT DISTINCT <<result\_columns>> -- [3] construct result columns
FROM <<source\_tables>> -- [1] conceptually join sources to form a large table
WHERE <<conditions\_for\_inclusion>> -- [2] Filter rows from [1]

1. <<source\_tables>>: the source tables to gather the result data
2. <<conditions\_for\_inclusion>>: the conditions to be satisfied for results to be included and possibly the conditions how the tables should be joined together.
3. <<result\_columns>>: the result columns or expressions desired to be displayed.
* Built-in functions and operators: <https://dev.mysql.com/doc/refman/8.1/en/built-in-function-reference.html>
* Some examples of common functions:
	1. BETWEEN lower\_range AND upper\_range
	2. IN: membership test for a set/table (binary operation)
	3. EXISTS: not an empty set (unary operation)
	4. IF: a ternary operation
	5. LIKE: inexact string matching.
		+ wild cards:
			- % match any and all following characters.
			- \_: match any one character.

***Example:***

Execute the following code and ensure that you understand the result.

-- operators:
-- student with credits in a range.
SELECT DISTINCT \*
FROM Student
WHERE credits BETWEEN 30 AND 70;

-- student in selected majors
SELECT DISTINCT \*
FROM Student
WHERE major IN ('CSCI', 'CINF', 'ITEC');

-- student enrolled in some classes.
SELECT DISTINCT \*
FROM Student AS s
WHERE EXISTS
(SELECT \*  -- a subquery
FROM Enroll AS e
WHERE e.stuId = s.stuId);
-- or
SELECT DISTINCT s.\*
FROM Student AS s INNER JOIN Enroll AS e USING (stuId);

-- students not enrolled in any class.
SELECT DISTINCT \*
FROM student AS s
WHERE s.stuId NOT IN (SELECT DISTINCT e.stuID FROM enroll AS e);

-- students wiht a 'k' in their last name.
SELECT DISTINCT s.\*
FROM student AS s
WHERE s.lname LIKE '%k%';

-- case sensitive version.
SELECT DISTINCT s.\*
FROM student AS s
WHERE s.lname LIKE BINARY '%k%';

-- case sensitive version: a more complicated take.
-- The mysql client sends the query using cp850.
-- The default character set of MySQL server is utf8mb4.
-- It is thus necessary to set the @@character\_set\_connection
-- in order to use collate if MySQL client is used.
-- If HeidiSQL is used, it is not necessary.
SET @@character\_set\_connection=utf8mb4;

SELECT DISTINCT s.\*
FROM student AS s
WHERE s.lname LIKE '%k%' COLLATE utf8mb4\_bin;

-- LIKE compares the whole string.
SELECT DISTINCT s.\*
FROM student AS s
WHERE s.lname LIKE 'ng';

-- student with last name of four characters, with ng the last two.
SELECT DISTINCT s.\*
FROM student AS s
WHERE s.lname LIKE '\_\_ng';

**3.3 Joins**

* When multiple tables are needed for a query, it is common that foreign keys are used to connect the tables.
* It is thus necessary to ensure that the equality of the foreign key with the referenced key of the parent table.
* A popular style is shown in the example below.

***Example:*** one popular SQL style

SELECT DISTINCT s.fname, s.lname, c.classId, e.grade
FROM student AS s, enroll AS e, class AS c
WHERE s.stuId = e.stuId -- Join condition
AND e.classId = c.classId -- Join condition
AND c.semester = 'Fall' -- problem condition
AND c.year = 2019; -- problem condition

Result:

mysql> SELECT DISTINCT s.fname, s.lname, c.classId, e.grade
    -> FROM student s, enroll e, class c
    -> WHERE s.stuId = e.stuId   -- Join condition
    -> AND e.classId = c.classId -- Join condition
    -> AND c.semester = 'Fall'   -- problem condition
    -> AND c.year = 2019;        -- problem condition
+---------+---------+---------+-------+
| fname   | lname   | classId | grade |
+---------+---------+---------+-------+
| Tony    | Hawk    |   10000 | A     |
| Mary    | Hawk    |   10000 | NULL  |
| David   | Hawk    |   10000 | B-    |
| Tony    | Hawk    |   10001 | A     |
| Mary    | Hawk    |   10001 | A-    |
| Tony    | Hawk    |   10002 | B+    |
| David   | Hawk    |   10002 | B+    |
| Tony    | Hawk    |   10003 | C     |
| David   | Hawk    |   10003 | D     |
| Larry   | Johnson |   10003 | A     |
| Linda   | Johnson |   10003 | NULL  |
| Tony    | Hawk    |   10004 | A-    |
| Larry   | Johnson |   10004 | B+    |
| Linda   | Johnson |   10004 | A-    |
| Lillian | Johnson |   10004 | C+    |
| Linda   | Johnson |   10005 | A-    |
| Lillian | Johnson |   10005 | A     |
| Linda   | Johnson |   10006 | B+    |
| Ben     | Zico    |   10007 | F     |
| Bill    | Ching   |   10007 | C-    |
| Ben     | Zico    |   10008 | A-    |
+---------+---------+---------+-------+
21 rows in set (0.00 sec)

**3.3.1 Inner Join**

* In the SELECT statement, the FROM clause allows the results of JOIN statements in the table references.
* Using the JOIN operations in the FROM clause is the preferred technique:
	1. Potentially faster performance: better optimization by DB engines, especially when using indexes.
	2. Better style: separation of join conditions and query semantic conditions.
	3. Easier changes between different joins.
* There are many kind of joins, as discussed below.
* You may use the Explain statement in MySQL to find out the execution plan.

***Example:***

Execute the following code and ensure that you understand the result.

SELECT DISTINCT s.fname, s.lname, c.classId, e.grade
FROM student AS s, enroll AS e, class AS c
WHERE s.stuId = e.stuId -- Join condition
AND e.classId = c.classId -- Join condition
AND c.semester = 'Fall' -- problem condition
AND c.year = 2019; -- problem condition

SELECT DISTINCT s.fname, s.lname, c.classId, e.grade
FROM student AS s INNER JOIN enroll e ON (s.stuId = e.stuId) -- Join condition
    INNER JOIN class AS c ON (e.classId = c.classId) -- Join condition
WHERE c.semester = 'Fall' -- Problem condition
AND c.year = 2019; -- Problem condition

-- alternative: using the USING clause.
SELECT DISTINCT s.fname, s.lname, c.classId, e.grade
FROM student AS s INNER JOIN enroll e USING (stuId) -- Join condition
    INNER JOIN class AS c USING (classId) -- Join condition
WHERE c.semester = 'Fall' -- Problem condition
AND c.year = 2019; -- Problem condition

-- the ON clause is more general and can be more effective.
SELECT DISTINCT s.fname, s.lname, c.classId, e.grade
FROM student AS s INNER JOIN enroll e ON (s.stuId = e.stuId) -- Join condition
    INNER JOIN class AS c
   ON (e.classId = c.classId -- Join condition
      AND c.semester = 'Fall' -- Problem condition
      AND c.year = 2019); -- Problem condition

**3.3.2 Left and Right Join**

* Left joins are the most popular joins besides (inner) joins.
* R1 LEFT JOIN R2: same as INNER JOIN, except that for a tuple t1 in R1 without a matching tuple in R2, t1 will be kept in the result with attributes from R2 being null.
	+ All rows in the left table will be in the result at least once.
* A right join is the mirror image of a left join.

***Example***

Execute the following code and ensure that you understand the result.

-- List the names of the students with their minors (in full name).
-- Student with no department not listed.
SELECT DISTINCT CONCAT(s.fname, ' ', s.lname) AS student,
    d.deptName AS `minor department`
FROM student AS s INNER JOIN department AS d ON (s.minor = d.deptCode);

-- List the names of the students with their minors (in full name).
SELECT DISTINCT CONCAT(s.fname, ' ', s.lname) AS student,
    d.deptName AS `minor department`
FROM student AS s LEFT JOIN department AS d ON (s.minor = d.deptCode);

-- List the names of the students with their minors (in full name).
-- more readable form.
SELECT DISTINCT CONCAT(s.fname, ' ', s.lname) AS student,
    IFNULL (d.deptName, 'N/A') AS `minor department`
FROM student s LEFT JOIN department d ON (s.minor = d.deptCode);

* Joins are procedural. Join orders can be important. Use parenthesis to enforce the desired order.

***Example:***

(R1 LEFT JOIN R2) RIGHT JOIN R3
-- may give different result than
R1 LEFT JOIN (R2 RIGHT JOIN R3)

***Example:***

Problem: List student information and the CSCI class information. Include all students, leaving blanks when appropriate
(i.e., no CSCI courses enrolled by the student).

+--------+-----------------+---------+-------------+-------+
| stuId  | student         | classId | CSCI course | grade |
+--------+-----------------+---------+-------------+-------+
| 100000 | Tony Hawk       | 10000   | CSCI 3333   | A     |
| 100000 | Tony Hawk       | 10001   | CSCI 4333   | A     |
| 100000 | Tony Hawk       | 10002   | CSCI 5333   | B+    |
| 100000 | Tony Hawk       | 11001   | CSCI 4333   | D     |
| 100001 | Mary Hawk       | 10000   | CSCI 3333   |       |
| 100001 | Mary Hawk       | 10001   | CSCI 4333   | A-    |
| 100002 | David Hawk      | 10000   | CSCI 3333   | B-    |
| 100002 | David Hawk      | 10002   | CSCI 5333   | B+    |
| 100003 | Catherine Lim   |         |             |       |
| 100004 | Larry Johnson   |         |             |       |
| 100005 | Linda Johnson   |         |             |       |
| 100006 | Lillian Johnson |         |             |       |
| 100007 | Ben Zico        |         |             |       |
| 100008 | Bill Ching      |         |             |       |
| 100009 | Linda King      |         |             |       |
| 100111 | Cathy Johanson  |         |             |       |
+--------+-----------------+---------+-------------+-------+
16 rows in set (0.001 sec)

***Example:*** (advanced)

Execute the following code and ensure that you understand the result.

-- List student information and the CSCI class information.
SELECT DISTINCT s.stuId,
   CONCAT(s.fname, ' ', s.lname) AS student,
   e.classId,
   CONCAT(co.rubric, ' ', co.number) AS `CSCI course`,
   e.grade
FROM student AS s INNER JOIN enroll AS e USING (stuId)
   INNER JOIN class AS c USING (classId)
   INNER JOIN course AS co USING (courseId)
WHERE co.rubric = 'CSCI';

-- List student information and the CSCI class information.
-- Include all students, leaving blanks when appropriate
-- (i.e. no CSCI courses enrolled by the student).

-- These do not do the job. Why?
SELECT DISTINCT s.stuId,
   CONCAT(s.fname, ' ', s.lname) AS student,
   IFNULL(e.classId, '') AS classId,
   IFNULL(CONCAT(co.rubric, ' ', co.number), '') AS `CSCI course`,
   IFNULL(e.grade, '') AS grade
FROM student AS s LEFT JOIN enroll AS e USING (stuId)
   LEFT JOIN class AS c USING (classId)
   LEFT JOIN course AS co USING (courseId)
WHERE co.rubric = 'CSCI';

SELECT DISTINCT s.stuId,
   CONCAT(s.fname, ' ', s.lname) AS student,
   IFNULL(e.classId, '') AS classId,
   IFNULL(CONCAT(co.rubric, ' ', co.number), '') AS `CSCI course`,
   IFNULL(e.grade, '') AS grade
FROM student AS s LEFT JOIN enroll AS e USING (stuId)
   LEFT JOIN class AS c USING (classId)
   LEFT JOIN course AS co ON (c.courseId = co.courseId AND co.rubric = 'CSCI' );

-- This works. Note the LEFT JOIN and RIGHT JOIN.
SELECT DISTINCT s.stuId,
   CONCAT(s.fname, ' ', s.lname) AS student,
   IFNULL(e.classId, '') AS classId,
   IFNULL(CONCAT(co.rubric, ' ', co.number), '') AS `CSCI course`,
   IFNULL(e.grade, '') AS grade
FROM enroll AS e INNER JOIN class AS c USING (classId)
   INNER JOIN course AS co ON (c.courseId = co.courseId AND co.rubric = 'CSCI' )
   RIGHT JOIN student AS s USING (stuId);

* Note:
	+ The inclusion of the condition co.rubric = 'CSCI' in the INNER JOIN condition.
	+ The student table should be joined the last using RIGHT JOIN.

**3.4 Subqueries**

* A SQL subquery is a nested/inner subquery within a SQL statement or another query (for SELECT, INSERT, UPDATE or, DELETE).
* Subqueries usually appear in the FROM clause (as derived tables) and the WHERE clause.

***Example***

Execute the following code and ensure that you understand the result.

-- subqueries in the WHERE course
-- students not enrolled in any class.
SELECT DISTINCT \*
FROM student AS s
WHERE s.stuId NOT IN (SELECT DISTINCT e.stuID FROM enroll AS e);

-- student with the maximum number of ach.
SELECT DISTINCT MAX(ach)
FROM student;

-- student within 60 credits of the maximum number of ach any student may have.
SELECT DISTINCT s.stuId,
   CONCAT(s.fname, ' ', s.lname) AS student,
   s.ach AS credits
FROM student AS s
WHERE s.ach + 60 >=
   (SELECT DISTINCT MAX(ach) FROM student);

-- subqueries as derived tables.
SELECT DISTINCT s.stuId,
   CONCAT(s.fname, ' ', s.lname) AS student,
   s.ach AS credits
FROM student AS s INNER JOIN
   (SELECT DISTINCT MAX(ach) AS max FROM student) AS m -- an alias is required.
WHERE s.ach + 60 >= m.max;

**3.5 Common Table Expressions (CTE)**

* Supported by MySQL 8.0 and forward.
* Allow the definition of temporary common tables in a sequence before the body of a SELECT statement.
	+ WITH t1 AS (definition of t1, a query...), t2 AS (...), ..., tn AS () SELECT ...
* A table defined in CTE can be used immediately until the end of the SELECT statement.
* Support a more natural way to implement *algorithmic solutions*, an (n+1) step solutions.
	+ step 1 to n: constructions of the common tables t1, t2, ..., tn
	+ step (n+1): the body of the SELECT statement.
* Allow recursion.
* May degrade performance.
* It is generally better than subqueries in the FROM clauses.
	+ Tables in CTE can be used immediately after their definitions.
	+ More natural order.
	+ Can use recursion.

***Example:***

-- CTE
WITH  t1 AS
   (SELECT MAX(ach) AS max FROM student)
SELECT s.stuId,
   s.ach AS `ach credits`,
   t1.max - s.ach AS `diff from max credits of all`
FROM student AS s, t1
ORDER BY `ach credits` DESC;

-- multiple common tables (not efficient; used as demonstration.)
WITH t1 AS
   (SELECT MAX(ach) AS max FROM student),
t2 AS
   (SELECT s.stuId,
      s.ach AS `ach credits`,
      t1.max - s.ach AS diff,
      s.major
    FROM student AS s, t1)
SELECT t2.stuId, t2.`ach credits`,
   t2.diff AS `diff from max credits of all`,
   d.deptName AS department
FROM t2 LEFT JOIN department d ON (t2.major = d.deptCode)
ORDER BY t2.`ach credits` DESC;

For those interesting in recursive CTE, here is an example. Recursive CTE will not be in the examinations.

Create and populate a simple relation that stores EmpId of an employee and the EmpId of the immediate supervisor.

CREATE SCHEMA CTETinker;
USE SCHEMA CTEtinker;
CREATE OR REPLACE TABLE Employee (
    EmpId CHAR(7) NOT NULL,
    SupervisorEmpId CHAR(7) NULL,
    CONSTRAINT Emp\_EmpId\_pk PRIMARY KEY (EmpId),
    CONSTRAINT Emp\_SupervisorEmpId\_fk FOREIGN KEY (SupervisorEmpId)
        REFERENCES Employee(EmpId)
);

INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES
   ('E3', null);
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES
   ('E15', 'E3');
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES
   ('E50', 'E15');
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES
   ('E75', 'E50');
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES
   ('E100', 'E75');
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES
   ('E102', 'E75');
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES
   ('E70', 'E50');
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES
   ('E103', 'E70');

SELECT \* FROM Employee;

Result:

MariaDB [temp]> SELECT \* FROM Employee;
+-------+-----------------+
| EmpId | SupervisorEmpId |
+-------+-----------------+
| E3    | NULL            |
| E50   | E15             |
| E15   | E3              |
| E70   | E50             |
| E75   | E50             |
| E103  | E70             |
| E100  | E75             |
| E102  | E75             |
+-------+-----------------+
8 rows in set (0.002 sec)

A recursive CTE SQL to get all supervisors of employee 'E100':

WITH RECURSIVE Super(SEId) AS
(  SELECT SupervisorEmpId AS SEId FROM Employee AS e WHERE e.EmpId = 'E100' -- initial condition/action
   UNION ALL -- union all: add rows created by the recursive action to the result, table Super.
   SELECT e.SupervisorEmpId AS SEId -- recursive action
      FROM Employee AS e INNER JOIN Super
      WHERE e.EmpId = Super.SEId
      AND e.SupervisorEmpId IS NOT NULL
      -- exit condition: when the recursive action returns an empty table.
)
SELECT \*
FROM Super;

Result:

+------+
| SEId |
+------+
| E75  |
| E50  |
| E15  |
| E3   |
+------+

DROP SCHEMA IF EXISTS CTEtinker;

**3.6 GROUP BY and HAVING**

* Useful for group reports: one result row per group, not per row as in the regular SELECT statement without GROUP BY.
* Allow aggregate functions (also known as *group functions* and column functions) to be performed by the groups defined.
* Output one row per *group*.
* A group is defined by an unique value of the columns in the group by clause.
* Example aggregate functions: MAX, MIN, AVG, COUNT, SUM, GROUP\_CONCAT, etc. See: <https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html>
* The HAVING clause allows using group functions in the condition. The WHERE clause does not allow using group functions.
* Using GROUP BY, the columns of the SELECT clause can only have:
	1. Columns named in GROUP BY.
	2. Aggregate functions on other columns in the tables.
	3. Constant expressions.
* A number can be used in the GROUP BY and the ORDER BY clauses to refer to the positions of the result columns in the select clauses.

Thus, the conceptual steps and framework for the SELECT statement become

SELECT DISTINCT <<result\_columns>> -- [5] construct result columns
FROM <<source\_tables>> -- [1] conceptually join tables to form a large table to produce initial rows
WHERE <<conditions\_for\_inclusion>> -- [2] Filter initial rows
GROUP BY <<group\_by\_columns>>
         --[3] group initial rows into groups by values of the group\_by\_column. A group becomes a new row.
HAVING <<conditions for filtering group>> -- [4] filter groups
ORDER BY <<columns>>; -- [6] Order the result of [5].

***Example:***

-- Student names and number of classes enrolled.
-- More than 2 classes to be included in the result.
SELECT CONCAT(s.fname, ' ', s.lname) AS student,
   COUNT(e.classId) AS `Enrolled classes`
FROM student AS s INNER JOIN enroll e ON (s.stuId = e.stuId)
GROUP BY student
HAVING `Enrolled classes` > 2
ORDER BY `Enrolled classes` DESC;

***Exercises***:

[1] Write a query to generate the student names and number of courses enrolled, including those not enrolled?

+-----------------+------------------+
| name            | Enrolled classes |
+-----------------+------------------+
| Tony Hawk       |                6 |
| Linda Johnson   |                4 |
| David Hawk      |                3 |
| Ben Zico        |                2 |
| Larry Johnson   |                2 |
| Mary Hawk       |                2 |
| Lillian Johnson |                2 |
| Bill Ching      |                1 |
| Catherine Lim   |                0 |
| Linda King      |                0 |
+-----------------+------------------+
10 rows in set (0.00 sec)

Solution:

SELECT CONCAT(s.fname, ' ', s.lname) AS student,
    COUNT(e.classId) AS `Enrolled classes`
FROM student AS s LEFT JOIN enroll e ON (s.stuId = e.stuId)
GROUP BY student
ORDER BY `Enrolled classes` DESC;

[2] Can you write a query to generate the following output?

+----------+------------------------------+------------+----------+----------+
| deptCode | deptName                     | numFaculty | numMajor | numMinor |
+----------+------------------------------+------------+----------+----------+
| ACCT     | Accounting                   |          1 |        0 |        0 |
| ARTS     | Arts                         |          1 |        2 |        0 |
| CINF     | Computer Information Systems |          2 |        2 |        3 |
| CSCI     | Computer Science             |          4 |        3 |        1 |
| ENGL     | English                      |          1 |        0 |        2 |
| ITEC     | Information Technology       |          2 |        2 |        2 |
| MATH     | Mathematics                  |          0 |        0 |        0 |
+----------+------------------------------+------------+----------+----------+
7 rows in set (0.00 sec)

Solution:

WITH ma AS
   (SELECT s.major AS deptCode, COUNT(s.stuId) AS numMajor
    FROM student AS s
    GROUP BY s.major),
mi AS
   (SELECT s.minor AS deptCode, COUNT(s.stuId) AS numMinor
    FROM student AS s
    GROUP BY s.minor),
f AS
   (SELECT f.deptCode, COUNT(f.facId) AS numFaculty
    FROM faculty AS f
    GROUP BY f.deptCode)
SELECT d.deptCode,
   d.deptName,
   IFNULL(f.numFaculty, 0) AS numFaculty,
   IFNULL(ma.numMajor, 0) AS numMajor,
   IFNULL(mi.numMinor, 0) AS numMinor
FROM department AS d LEFT JOIN ma USING (deptCode)
   LEFT JOIN mi USING (deptCode)
   LEFT JOIN f USING (deptCode);

**3.7 Window Functions**

* MySQL 8.x supports Window functions.
* A window function performs a computation on a set of rows (a window frame) in which the current row is in the window frame.
* It is not a clause.
* Unlike the GROUP BY clause, it does*not* form groups.
* The OVER clause is used to define the window frame.
* OVER(): all rows are in the window frame.
* OVER(PARTITION BY X): each X value defines a window frame.
* Many aggregate functions can be used by Window functions.
* Modern DBMS support a rich set of Window functions.

***Example:***

WITH temp AS
(SELECT DISTINCT sc.schoolName AS college, d.deptName AS department,
   COUNT(s.stuId) As deptMajor
FROM school AS sc INNER JOIN department AS d ON (sc.schoolCode = d.schoolCode)
   LEFT JOIN student AS s ON (s.major = d.deptCode)
GROUP BY college, department)
SELECT temp.college, temp.department,
   temp.deptMajor AS `major in department`,
   SUM(deptMajor) OVER(PARTITION BY college) AS `major in college`,
   SUM(deptMajor) OVER() AS `major in university`
FROM temp;

Please execute to see the output.

Adding row number and rank:

WITH ma AS
(SELECT s.major AS deptCode, COUNT(s.stuId) AS numMajor
FROM student AS s
GROUP BY s.major),
mi AS
(SELECT s.minor AS deptCode, COUNT(s.stuId) AS numMinor
FROM student AS s
GROUP BY s.minor),
f AS
(SELECT f.deptCode, COUNT(f.facId) AS numFaculty
FROM faculty AS f
GROUP BY f.deptCode)
SELECT ROW\_NUMBER() OVER () AS `#`,
   RANK() OVER (ORDER BY f.numFaculty DESC) AS `# in descending number of faculty`,
   d.deptCode,
   d.deptName,
   IFNULL(f.numFaculty, 0) AS numFaculty,
   IFNULL(ma.numMajor, 0) AS numMajor,
   IFNULL(mi.numMinor, 0) AS numMinor
FROM department AS d LEFT JOIN ma USING (deptCode)
   LEFT JOIN mi USING (deptCode)
   LEFT JOIN f USING (deptCode);