Study Unit 3
Creating Database and Database Objects

Learning Outcomes

By the end of this unit, you should be able to:

1. Construct a logical model from a data model
2. Implement a physical model from a logical model
3. Formulate SQL DML insert, delete, update statements
4. Formulate SQL DML select statement to implement and query database systems
5. Implement business rules via views

Overview

In this study unit, we look into implementing a database.

We start with the data model from Study Unit 2. The steps and rules to transform a data model into a logical model are then covered. Constructing a logical model is mechanical, and is based on the rules for the transformation that you should become familiar with.

SQL Data Definition Language statements that create, alter and drop database objects are introduced next. The implementation of the logical model, via SQL Data Definition Language statements that create both the database and tables, is also demonstrated.

Once the database is implemented, SQL Data Manipulation Language statements are used to perform operations such as insert, delete and update on the database. We also formulate SQL Data Manipulation Language select statement to query the database.

Finally, views or virtual tables are covered. There are many uses of views, one of which is to provide client with an abstraction of the data, and so a client that depends on the abstraction can be isolated from changes to the implementation of the underlying database tables.

This study unit covers chapters 6, 7 and 2 of the course text. It is estimated that the student will spend about 9 hours to read the course text chapters 6, 7 and 2 in conjunction with the study notes, to work out the activities, and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered. It is advisable to use the study notes to guide the reading of the chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Chapter 1 The Logical Model

After the correct data requirements are captured in a data model, the next step is to transform the data model into a logical model, that is, into relations with the necessary integrity constraints. Integrity constraints are grouped into the following categories:

- Entity integrity constraints
  The entities in the data model are made into relations and their identifiers are the primary keys of the relations.

  The entity constraints specify that primary keys must be unique and cannot be null (that is, not having a value).

- Referential integrity constraints
  The relationships between entities are implemented using foreign keys. When two entities are related, one relation is made a parent and the other relation a child. The primary key of the parent is duplicated into the child as a foreign key.

  The referential integrity constraints specify that any value for a foreign key must be one of the values for the primary key, or alternatively, the foreign key value must be null (that is, not having a value).

- Key constraints
  All the keys of the relations must be specified as some key constraints.
  - The primary key constraint specifies the primary key.
  - The foreign key constraints specify the foreign keys and the corresponding parent relations.
  - The alternative key constraints are specified for candidate keys that have not been chosen as the primary keys, but nevertheless, have values that are unique in the relation.

- Domain integrity and Data constraints
  The attributes of a relation are the attributes of an entity, including its identifier.

  The domain integrity constraints specify the constraints for the values of the attributes such as whether values are required (null status), the data type, the valid values or the format of the valid values.

The logical model is platform-independent. Although the course text refers to relations as tables, we will use the terms table, column and row only in chapter 2 of this study unit, when the relations are implemented on a specific DBMS. The terms relation, attribute and tuple are used in this chapter.

Activity 1

Reproduced from Question 6.2 of the course text.

What is the difference between the logical and physical design of a database?

Refer to Figure 3.1 for the steps to transform a data model into a logical model.

Transforming a Data Model into a Database Design
1. Create a table for each entity:
– Specify the primary key (consider surrogate keys, as appropriate)
– Specify alternate keys
– Specify properties for each column:
• Null status
• Data type
• Default value (if any)
• Data constraints (if any)
– Verify normalization
2. Create relationships by placing foreign keys
– Relationships between strong entities (1:1, 1:N, N:M)
– Identifying relationships with ID-dependent entities (intersection tables, association patterns, multivalued attributes, archetype/instance patterns)
– Relationships between a strong entity and a weak but non-ID-dependent entity (1:1, 1:N, N:M)
– Mixed relationships
– Relationships between supertype/subtype entities
– Recursive relationships (1:1, 1:N, N:M)
3. Specify logic for enforcing minimum cardinality:
– O-O relationships
– M-O relationships
– O-M relationships
– M-M relationships

Figure 3.1 Steps for transforming a data model to a logical model

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 6.1)

1.1 Create a Relation for Each Entity

Each entity becomes a relation in the logical model. The identifier of the entity becomes its primary key in the relation. The prmary key identifies tuples in a relation.

If the identifier is not suitable as primary key, for reasons such as the identifier consists of several attributes and the values for the attributes are lengthy and non-numeric, then a surrogate key can be introduced at this time.

If there are other candidate keys but not chosen as primary key, they are made into alternate keys.

Each attribute is described in terms of the following properties:

- Whether the attribute can have null value

  Identify whether it is required that a value be provided to the attribute.

  If a value is not required for an attribute, then the attribute can have null value, that is, its null status is null. Otherwise, its null status is not null.

- Data type

  Identify the generic data type, such as alphanumerice, decimal etc.

  The actual data types supplied by the DBMS will be used in the implementation to a physical model. The physical model is discussed in chapter 2 of this study unit.

- Default values

  Identify whether an attribute has a default value.

  For example, an attribute for a date attribute may have the default value, today.

- Data constraints

  Identify whether there are constraints to the values that an attribute can have.

  For example, a valid value for an attribute for cost is within the range 10 and 20.

Each relation formed through the transformation must be normalised to BCNF or 4NF using the procedure in Section 2.2, Study Unit 1.

Normalisation is necessary to remove modification anomalies due to functional and multivalued dependencies. This happens when the entities in the data model are not single-themed, that is, the entities are about more than one thing.

Performing normaliaton may result in more relations in the logical model.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 280-292.

1.2 Create Relationships by Placing Foreign Keys

Relationships in a data model show up as foreign keys in the logical model. One entity becomes a parent and the other a child.

The term parent is used on the relation whose primary key is duplicated in another relation as foreign key. The relation that has the foreign key is called the child.

If a relationship is identifying, the identifier of the parent is already in the identifier of the child.

If the relationship is non-identifying, the identifier of the parent is duplicated in the child as foreign key.

Once the parent-child relationships are established, a referential integrity constraint, also called foreign key constraint, is specified for each foreign key. The foreign key constraint is specified in the child relation.

Apply the following standard rules to decide which relation is the child in a non-identifying relationship:

- 1:1 relationship

  If minimal cardinality of both ends of the relationship are the same (both ends are mandatory or both ends are optional), any of the two relations can be parent and the other a child.

  If minimum cardinality on the two ends of the relationship are different, **always** make the relation for the entity on the optional side of the relationship the child.

- 1:N relationship

Consider only the maximum cardinality for this case, and **always** make the relation for the entity on the N-side (or many side) of the relationship the child.

The parent is the relation for the entity on the 1-side of the relationship.

- N:M relationship

  Always create another relation for each N:M relationship to resolve the N:M relationship into two 1:N relationships. The new relation is also called the intersection table (or intersection relation).

  The intersection table is a child to both entities in the N:M relationship. The intersection table consists of only a primary key which is the composite of the primary keys of both its parents.

We will now consider how to decide which relation is a child for the various design patterns discussed in Study Unit 2:

- Strong relationships

  - 1:1 strong entity relationship

    If the minimal cardinality of the relationship on both ends are the same, place the primary key of one relation in the other relation as foreign key. That is, pick one relation as parent and the other relation as child.

    If the minimal cardinality of the relationship on both sides are different, the relation on the mandatory side is always the parent and the relation on the optional side is always the child.

  - 1:N strong entity relationship

    The relation on the one-side is always the parent and the relation on the many-side is always the child.

  - N:M strong entity relationship

  Both entities of the N:M relationship cannot be a child. Instead, introduce an intersection table, and resolve the N:M relationship into two 1:N relationships. The intersection table is on the many-side for both 1:N

relationships. Therefore, always make the intersection table a child of the relations for both entities of the N:M relationship.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 292-295.

Activity 2

Modified from Question 6.19 of the course text.

Describe two ways to represent a 1:1 strong entity relationship using the example in Activity 3 a) of Study Unit 2. The scenario is reproduced here:

In the Real Estate Agency example, each AGENT must use an agency car when on agency business. Further, to keep costs down, the agency keeps exactly enough cars for the agents. Therefore, each AGENT must have a CAR, and each CAR must be assigned to an AGENT. This is a 1:1, M-M relationship.

- ID-dependent relationship
    - Association pattern and the associative entity

      The identifier of the associative entity already contains the identifiers of the two strong entity in the association pattern.

      Therefore, the relation for an associative entity is always the child, and relations for the two strong entity in the association pattern are always the parents.

- Multivalued attribute pattern

  The identifier of the entity for the repeating group or multivalued attribute contains the identifier of the entity that has the multivalued attribute.

  Therefore, the relation for the entity for the repeating group or multivalued attribute is always the child. The parent is always the entity that has the repeating group or multivalued attribute.

- Archetype/Instance Pattern

  The instance is always the child and the archetype is always the parent.

  The identifier of the instance contains the identifier of the archetype if the relationship is identifying.

  If the relationship is non-identifying, put the primary key of the archetype into the relation for the instance as foreign key.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 295-300.

- Mixed identifying and non-identifying relationship

  The weak entity that has an identifying relationship with a strong entity is always a child to this strong entity.

  The weak entity also has a non-identifying relationship with another strong entity. The weak entity is considered as strong with respect to this other strong entity. Therefore, treat this non-identifying relationship like any other 1:1, 1:N or N:M strong relationship.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 300-302.

- For-Use Pattern

  The subtype is always a child of the supertype. The primary key of the supertype is duplicated in the subtype as foreign key.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 300-302.

- Recursive relationship

  Treat the recursive relationship like any other 1:1, 1:N or N:M strong relationship. Therefore, we look at the maximum cardinality to decide which relation is child.

  - 1:1 recursive relationship

    In a recursive relationship, an instance of the entity is related to another instance of the same entity, and so, the relation is both parent and child.

    Copy the primary key of the relation into itself as a foreign key. Name the foreign key differently from the primary key, as attribute names must be unique for a relation to be in 1NF.

    To decide on the attribute name for the foreign key, consider how the foreign key is related to each instance in the relation. The foreign key is named according to the role played by the parent.

For example, consider the recursive relationship, isMarriedTo, described in Section 2.5.1, Study Unit 2, depicted in Figure 2:19, reproduced here as Figure 3.2:

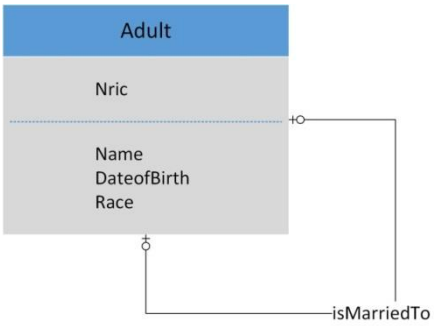

Figure 3.2 1:1 recursive relationship

In the case of this 1:1 recursive relationship, siMarriedTo, the foreign key can be named either as:

- isMarriedToThisAdult

  The foreign key records the person whom the instance of the entity married to.

- marriesThisAdult

  The foreign key records the person whom the instance of an entity marries.

- 1:N recursive relationship

Similar to 1:N strong relationship, copy the primary key of the relation on the one-side of the relationship into itself as a foreign key. Name the foreign key differently from the primary key, as attribute names must be unique for a relation to be in 1NF.

Name the foreign key according to the role played by the parent.

For example, consider the recursive relationship, manages, described in Section 2.5.2, Study Unit 2, depicted in Figure 2:20, reproduced here as Figure 3.3:

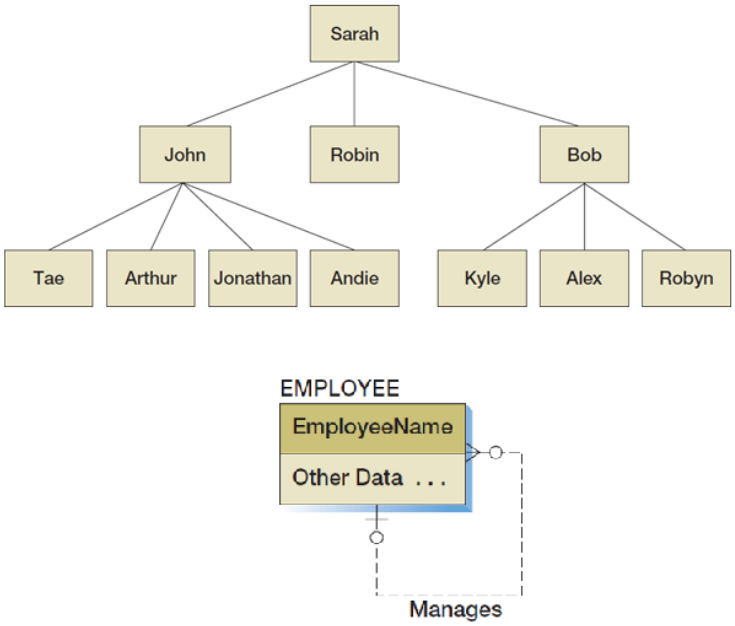

Figure 3.3 1:N recursive relationship

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 5-40 and Figure 5-41)

As the role played by the one-side is manager, the foreign key can be named manager.

- N:M recursive relationship

  Note that the N:M recursive relationship is also resolved as two 1:N relationship with the introduction of an intersection table.

  Put the primary key of the relation on the both sides of the N:M relationship into the intersection table as composite primary key and as two foreign keys.

  Name each foreign key according to the role played by each parent.

  For example, consider the recursive relationship, consists, described in Section 2.5.3, Study Unit 2, depicted in Figure 2:21, reproduced here as Figure 3.4:

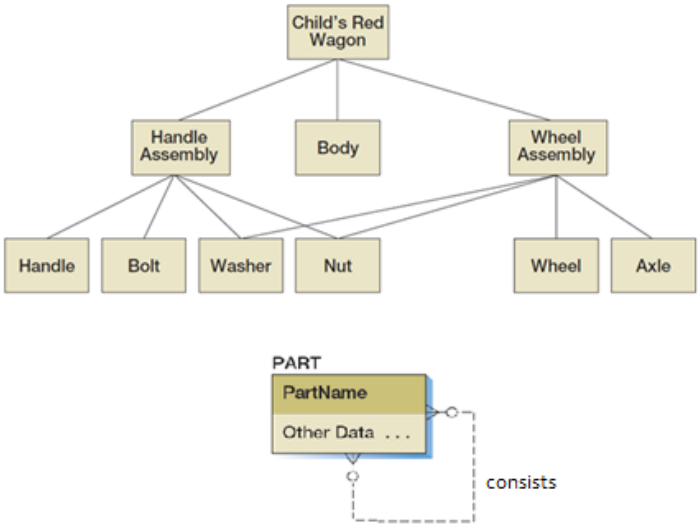

Figure 3.4 N:M recursive relationship

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 5-42 and Figure 5-43)

As the role played by the one parent is the container (as it consists) and the other role played by the other parent is the contained (as it is consisted in), the foreign keys can be named containerPart (or simply, part) and containedPart.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 303-308.

Activity 3

Reproduced from Question 6.22 of the course text.

What is the trick for representing N:M recursive relationships?

Activity 4

Modified from Question 6.33 of the course text.

Describe how to represent this 1:N recursive relationship.
Customers of the All Stuff Warehouse may refer one or more other customers to the business. Therefore, there are many referred customers with exactly one referring customer, who has also been referred by another customer. However, customers may not have been referred, and customers do not have to make referrals.

1.3 Specify Logic for Enforcing Minimum Cardinality

The minimum cardinality of a relation is enforced by specifying (programming) logic, unlike for maximum cardinality.

The maximum cardinality of a relation is easily enforced using the unique constraint.

- If the maximum cardinality is 1, simply make the foreign key unique

  The foreign key values are not repeated in the child table. This means each parent tuple can be related to at most 1 child tuple.

- If the maximum cardinality is many, do not impose a unique constraint on the foreign key

  The foreign key values can be repeated in the child table. This means each parent tuple can be related to many child tuples.

The minimum cardinality of a relation is easily enforced when the child is optional, but is difficult for mandatory child. The minimum cardinality for optional or mandatory parent is easily enforced by defining the appropriate null status for the foreign key.

1.3.1 Optional or mandatory parent

When a parent is optional, the null status for the foreign key is null, and when a parent is mandatory, the null status for the foreign key is not null,

In addition, whether parent is optional or mandatory, the child relation must always specify what should happen to its tuples when their parent tuple is updated or deleted.

A child relation can specify the following policies with regards to update or delete operations on the parent relation:

- Cascading actions

  When a child relation specifies cascading delete/update, then child tuples are also deleted/updated when a parent tuple is deleted/update.

- No action (prohibit)

  For example, when a child relation specifies no action for delete/update, then a parent tuple cannot be deleted/updated if there are affected child tuples.

- Set to default value or set to null

  When a child relation specifies a default for delete/update, then when a parent tuple is deleted/updated, the foreign key of the related child tuples is set to the default value or to null.

When a child tuple is inserted or updated, the referential integrity constraint imposes that the foreign key must exist for mandatory parent or be null if the parent is optional.

If there is no suitable parent tuple for the case of mandatory parent, the insert or update is prohibited. If there is no suitable parent tuple for the case of optional parent, simply set the foreign key to null.

Refer to Figure 3.5 for a summary of actions to enforce minimum cardinality for required parent.

Parent Required	Action on Parent	Action on Child
Insert	None.	Get a parent. Prohibit.
Modify key or foreign key	Change children's foreign key values to match new value (**cascade update**). Prohibit.	OK, if new foreign key value matches existing parent. Prohibit.
Delete	Delete children (**cascade delete**). Prohibit.	None.

Figure 3.5 Actions when parent is required

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 6.29a)

1.3.2 Optional or mandatory child

- Optional child

  There is no action required to enforce an optional child.

  When the child is optional, a parent tuple need not be related to any child tuple.

  This means that nothing need to be done on the parent relation when a child tuple is updated or deleted, as by definition of parent, the parent relation does not track the relationship, and does not have the foreign key.

  When a parent tuple is inserted, there is no need for its primary key to appear as a foreign key in the child relation as the child is optional. Thus, again, no action is needed for a new parent tuple.

- mandatory child

  When the child is mandatory, a parent tuple must be related to at least one child tuple.

  When a child tuple is updated or deleted, it is possible for a parent tuple to become childless. Therefore, before the child tuple can be updated or deleted, (programming) logic must ensure that it is not the last child for any parent tuple. Otherwise, the (programming) logic must prohibit the operation.

  When a parent tuple is inserted, (programming) logic is needed to make the foreign key of one child tuple to refer to the primary key of the newly-inserted tuple.

  As (programming) logic must be specified to enforce mandatory child, wherever possible, always make the mandatory side a parent and the optional side a child.

  In general, the (programming) logic is encoded in triggers, which is covered in Study Unit 4

Refer to Figure 3.6 for a summary of actions to enforce minimum cardinality for required child.

Child Required	Action on Parent	Action on Child
Insert	Get a child. Prohibit.	None.
Modify key or foreign key	Update the foreign key of (at least one) child. Prohibit.	If not last child, OK. If last child, prohibit or find a replacement.
Delete	None.	If not last child, OK. If last child, prohibit or find a replacement.

Figure 3.6 Actions when child is required

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 6.29b)

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 309-316.

Activity 5

Reproduced from Question 6.20 of the course text.

Describe the difference between an O-M relationship and an M-M relationship.

Activity 6

Reproduced from Question 6.45 of the course text.

Explain why the DBMS cannot be used to enforce required children.

Activity 7

Reproduced from Question 6.48 of the course text.

Explain the need for each of the design decisions in shown here:

Relationship Minimum Cardinality	Design Decisions to Be Made	Design Documentation
M-O	• Update cascade or prohibit? • Delete cascade or prohibit? • Policy for obtaining parent on insert of child	Referential integrity (RI) actions plus documentation for policy on obtaining parent for child insert.
O-M	• Policy for obtaining child on insert of parent • Primary key update cascade or prohibit? • Policy for update of child foreign key • Policy for deletion of child	Use Figure 6-28(b) as a boilerplate.
M-M	All decisions for M-O and O-M above, plus how to process trigger conflict on insertion of first instance of parent/child and deletion of last instance of parent/child.	For mandatory parent, RI actions plus documentation for policy on obtaining parent for child insert. For mandatory child, use Figure 6-28(b) as a boilerplate. Add documentation on how to process trigger conflict.

1.4 View Ridge Art Gallery

Study the View Ridge summary of data requirement for an art acquisition application, as shown in Figure 3.7.

Summary of View Ridge Gallery Database Requirements
Track customers and their interest in specific artists
Record the gallery's purchases
Record customer's purchases
Report how fast an artist's works have sold and at what margin
Show the artists represented by the gallery on a Web page
Show current inventory on a Web page
Show all the works of art that have appeared in the gallery on Web pages

Figure 3.7 View Ridge summary of data requirements

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 6.36)

From the data requirements, we can identify these entities: customer, artist, purchase, work of arts.

The data model shown in Figure 3.8, captures the data requirements. The data model uses the relation name, work for work of arts and the relation name, trans for purchase.

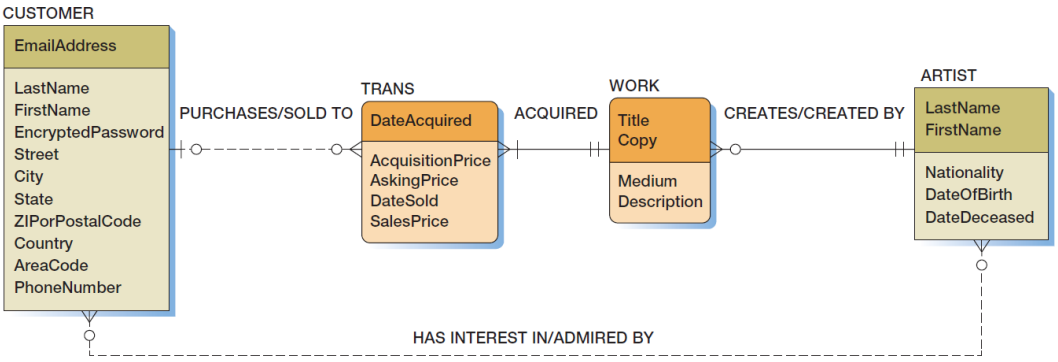

Figure 3.8 View Ridge data model

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 6-37)

Using the transformation steps outlined in Section 1, the logical model is derived, and shown in Figure 3.9. Note an intersection table, CUSTOMER_ARTIST_INT has been added to resolve the N:M relationship between customer and artist.

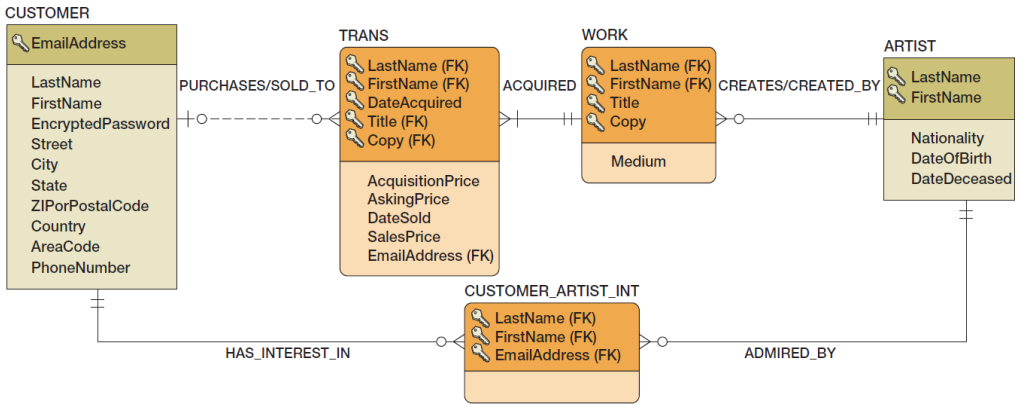

Figure 3.9 View Ridge logical model

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 6-38)

As the primary keys for the relations in the logical model are cumbersome, surrogate keys are introduced into the relations of the logical model, as shown in Figure 3.10.

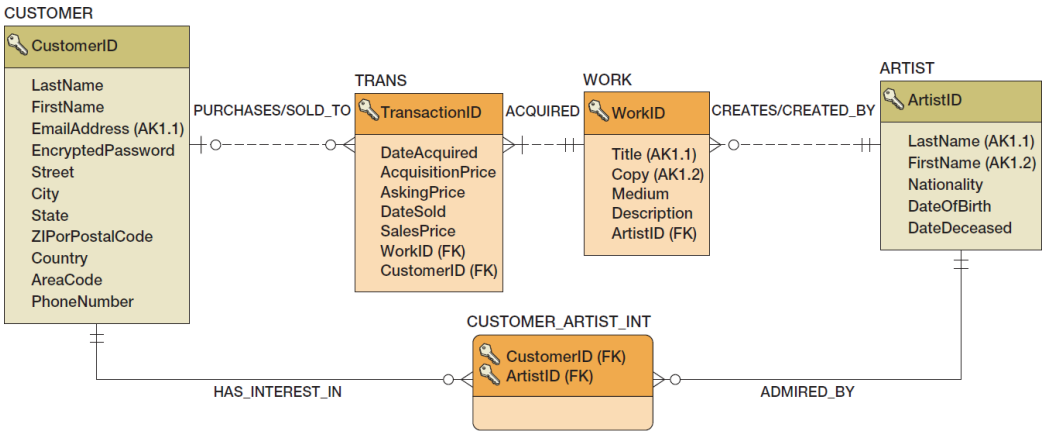

Figure 3.10 View Ridge logical model with surrogate keys

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 6-39)

A database will be implemented for it in chapter 2 of this study unit, as an example of a physical scheme.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 316-325.

Chapter 2 Physical Model

An implementation of the logical model is a physical model or the database. A database is one of the four components of a database system.

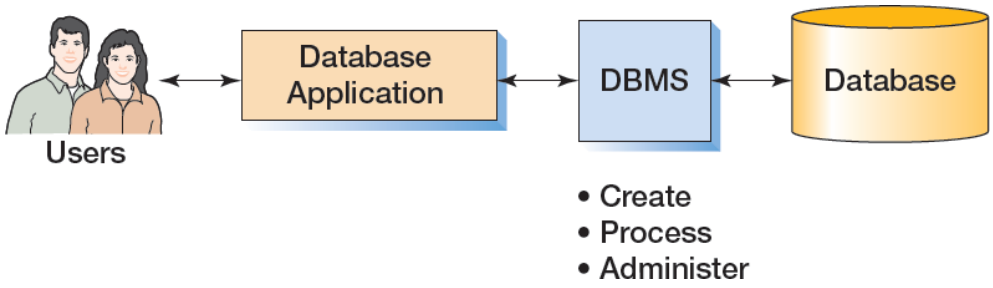

Figure 3.11 Components of a Database System

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 1-8)

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 43-49.

The implementation of the logical model to the physical model (or database) is done using a Database Management System or DBMS. A DBMS is a software that creates databases.

DBMSs are complex and are usually written by software companies. Some examples of DBMSs are Microsoft (MS) Access, Microsoft (MS) SQL Server, IBM DB2, Oracle and mySQL.

The DBMS stores data records for the databases on external storage. The data records are organised such that data applications can make queries to access them.

Database applications can request for data to be inserted, updated or deleted from database, all using Structured Query Language (SQL), a standard language that DBMSs understand. SQL statements are case-insensitive.

We create the database using Microsoft (MS) SQL Server.

2.1 SQL DDL statements

SQL Data Definition Language statements include

- CREATE statements to create database objects such as tables and views
- ALTER statements to modify existing database objects
- DROP statements to remove database objects

2.1.1 SQL DDL Create, Alter and Drop Database statements

- `create database` *`databaseName optional options`*

  This statement is a command to the DBMS to create a database with the database name, *databaseName*.

  For example, to create a database named vrg, issue the statement below for the DBMS to execute:

  ```
  create database vrg
  ```

- `alter database` *`databaseName optional options`*

  This statement is a command to the DBMS to make changes to a database with the database name, *databaseName* with the updated options.

- `drop database` *`databaseName`*

This statement is a command to the DBMS to delete a database with the database name, *databaseName*.

2.1.2 SQL DDL Create, Alter and Drop Table statements

The tables are the physical model of the relations in a logical model.

- Create table statement

```
create table tableName (
columnName datatype nullStatus,
…
constraint constraintName constraintDefinition,
…
)
```

After a database has been created, tables can be created for it.

The `create table` statement is a command to the DBMS to create a table with the table name, *tableName* for a database. We can specify which database using the `use` statement before issuing the `create table` statement.

```
use databaseName
```

The attributes of relations become the columns of the respective tables. Their data types, default values and data constraints are specified in the `create table` statement.

The primary keys of relations become the primary keys of the respective tables. Foreign keys defined in relations become foreign keys in the respective tables. Child relations become child tables.

Two examples of `create table` statements are shown in Figure 3.12.

  - The first `create table` statement creates the table Artist.

    The surrogate key is ArtistId, as shown in the logical model in Figure 3.10.

    The values of a surrogate key are generated by the DBMS, in the case of SQL Server, using the `identity` column ArtistId. In the example,

`identity(1,1)` means the value 1 is given to the first row, the value 2 is given to the second row, and so on.

The identity column, ArtistId is made into a primary key using the `primary key` constraint.

A `unique` constraint is specified for lastName and firstName as they form a composite candidate key, that is, the combination of the values for lastName and firstName is unique in the relation, and hence, in the table Artist.

Constraints should be given names, e.g., ArtistPK and ArtistAK1, so that they can be dropped easily if they are no longer applicable. This happens because over time, the database may need to be changed to cater to new data requirements.

Besides specifying key constraints, we can also specify data constraints. Data constraints use the keyword `check`.

For example, four data constraints are specified in the table:

- ```
  constraint NationalityValues CHECK
  (Nationality IN ('Canadian', 'English',
  'French','German', 'Mexican', 'Russian',
  'Spanish','United States'))
  ```

  This constraint is a domain check.

  It restricts the values for the column Nationality to only this set of values: Canadian, English, French, German, Mexican, Russian, Spanish and United States.

- constraint *BirthValueCheck* check (*DateofBirth* < *DateOfDeceased*)

  This constraint is a intra-relation constraint.

  It checks two columns in the same table. In this case, it checks that the DateofBirth must be earlier than the DateOfDeceased.

- constraint *ValidBirthYear* check (*DateofBirth* Like *'[1-2][0-9][0-9][0-9][0-9]'*)
  ```

This constraint is a format check.

It checks that the first digit of DateOfBirth must either be a 1 or 2 and the subsequent 3 digits must be a digit from 0 to 9.

- constraint *ValidDeathYear* check (*DateOfDeceased* Like *'[1-2][0-9][0-9][0-9][0-9]'*)

  Likewise, this constraint is a format check.

  It checks that the first digit of DateOfDeceased must either be a 1 or 2 and the subsequent 3 digits must be a digit from 0 to 9.

Other types of `check` constraints include

- a range check

  e.g., the constraint *ValidBirthYear* can be rewritten as:

  constraint *ValidBirthYear* check (*DateofBirth* between *1000 and 2999)*

- an inter-relation constraint

  An inter-relation constraint is a constraint that involves more than one relation. We cover inter-relation checks in Study Unit 4.

- The second `create table` statement creates the table Work.

  The table Work is a child table. It has a duplicated column ArtistId, duplicated from the table Artist. Thus, the parent of table Work is the table Artist.

  Notice that the null status of the duplicated column ArtistId in table Work is not null. This implements the required or mandatory parent constraint, that each row in Work must be related, at the minimum, to one row in Artist.

  Besides specifying the parent and the column(s) of parents, the `foreign key` constraint also specifies the minimal cardinality actions,

in this case, the primary key in the parent table, Artist cannot be deleted or updated if there are affected child rows in the table Work.

To give the default constraint a name, the column definition for the column [Description] may be rewritten as:

```
[Description]        varchar(1000)    null      constraint
desc_def  default 'Unknown provenance'
```

`desc_def` is the name for the `default` constraint.

```
CREATE TABLE ARTIST (
       ArtistID            Int              NOT NULL IDENTITY(1,1),
       LastName            Char(25)         NOT NULL,
       FirstName           Char(25)         NOT NULL,
       Nationality         Char(30)         NULL,
       DateOfBirth         Numeric(4,0)     NULL,
       DateDeceased        Numeric(4,0)     NULL,
       CONSTRAINT       ArtistPK         PRIMARY KEY(ArtistID),
       CONSTRAINT       ArtistAK1        UNIQUE(LastName, FirstName),
       CONSTRAINT       NationalityValues   CHECK
                           (Nationality IN ('Canadian', 'English', 'French',
                            'German', 'Mexican', 'Russian', 'Spanish',
                            'United States')),
       CONSTRAINT       BirthValuesCheck    CHECK (DateOfBirth < DateDeceased),
       CONSTRAINT       ValidBirthYear      CHECK
                           (DateOfBirth LIKE '[1-2][0-9][0-9][0-9]'),
       CONSTRAINT       ValidDeathYear      CHECK
                           (DateDeceased LIKE '[1-2][0-9][0-9][0-9]')
       );

CREATE TABLE WORK (
       WorkID              Int              NOT NULL IDENTITY(500,1),
       Title               Char(35)         NOT NULL,
       Copy                Char(12)         NOT NULL,
       Medium              Char(35)         NULL,
       [Description]       Varchar(1000)    NULL DEFAULT 'Unknown provenance',
       ArtistID            Int              NOT NULL,
       CONSTRAINT       WorkPK           PRIMARY KEY(WorkID),
       CONSTRAINT       WorkAK1          UNIQUE(Title, Copy),
       CONSTRAINT       ArtistFK         FOREIGN KEY(ArtistID)
                           REFERENCES ARTIST(ArtistID)
                              ON UPDATE NO ACTION
                              ON DELETE NO ACTION
       );
```

Figure 3.12 SQL Statements to Create the Artist and Work with Default Values and Data Constraints

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 7-12)

Activity 8

Reproduced from Question 7.3 of the course text.

Explain the meaning of the following expression: IDENTITY (4000, 5).

Activity 9

Modified from Question 7.14 of the course text.

Given the Create table statement for Customer_01:

```
CREATE TABLE CUSTOMER_01(
  EmailAddress       VARCHAR(100)         NOT NULL,
  LastName           VARCHAR(25)          NOT NULL,
  FirstName          VARCHAR(25)          NOT NULL,
  CONSTRAINT         CUSTOMER_01_PK       PRIMARY KEY(EmailAddress)
);
```

Write the Create table statement for the relation:
SALE_01 (SaleID, DateOfSale, EmailAddress, SaleAmount)
where EmailAddress in SALE_01 must exist in EmailAddress in CUSTOMER_01

The relationship from SALE_01 to CUSTOMER_01 is N:1, O-M.
In this database, CUSTOMER_01 and SALE_01 records are never deleted. You also will need to decide how to implement the ON UPDATE referential integrity action.

Activity 10

Reproduced from Question 7.17 of the course text.

Could we have created the SALE_01 table before creating the CUSTOMER_01 table? If not, why not?

- Make changes to a table

  `alter table tableName change`

  This statement is a command to the DBMS to make a change to a table, *tableName*.

  A change to a table includes:

  - adding a column or a constraint

    Format:

    `Alter table tableName add columnName datatype nullStatus defaultValueConstraint`

    `Alter table tableName add constraint constraintName constraintDefinition`

  - altering or changing a column characteristics

    `Alter table tableName alter columnName datatype nullStatus defaultValueConstraint`

  - dropping a column or a constraint

    `Alter table tableName drop column columnName`

    `Alter table tableName drop constraint constraintName`

- Remove a table

```
drop table tableName
```

This statement is a command to the DBMS to delete a table with the table name, *tableName*.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 335-351.

2.2 SQL DDL statements

SQL Data Manipulation Language includes statements to manipulate data in a database, that is, to insert data into a table, to update data in a table, to delete data from a table and to make queries.

2.2.1 SQL DML Insert statement

The SQL DML `insert` statement is a command to the DBMS to add a new row to a table.

Format:

```
insert into tableName(columnName1, …, columnNamen)  values(value1, …, valuen)
```

For example, to add a row with certain values into the table Artist, issue the SQL command:

```
insert into Artist (LastName, FirstName, Nationality, DateOfBirth, DateDeceased) VALUES ('Tamayo', 'Rufino', 'Mexican', 1899, 1991)
```

Notice that no value is provided for the identity column, ArtistId.

Inserted rows are stored as records on the secondary storage, in the order of a clustered index. The default clustered index is the primary key.

Indexes can be created, altered and dropped.

As there can be at most one clustered index per table, other indexes created are non-clustered. Non-clustered indexes are implemented on separate files to point to the physical records; they do not rearrange the positions of the records on the secondary storage.

Activity 11

Modified from Question 7.22 of the course text.

Write and run a set of SQL INSERT statements to populate the CUSTOMER_01 table with the data shown in the table here:

	EmailAddress	LastName	FirstName
1	Chris.Bancroft@somewhere.com	Bancroft	Chris
2	Katherine.Goodyear@somewhere.com	Goodyear	Katherine
3	Robert.Shire@somewhere.com	Shire	Robert

2.2.2 SQL DML Update statement

The SQL DML `update` statement is a command to the DBMS to modify existing row(s) in a table.

Format:

`update` *tableName* `set` *columnName*$_1$ `=` *value*$_1$`, …,` *columnName*$_n$ `=` *value*$_n$

`where` *condition*

The `where` *condition* is optional. Leaving it out will cause every row to be updated.

An example of an `update` statement is shown here:

```
update Artist set Nationality = 'United States' where LastName = 'Smith'
```

An `update` statement may result in zero or more rows being updated.

When the DBMS executes the statement, the nationality of artist(s) whose last name is Smith is updated to United States.

Activity 12

Write and run an SQL UPDATE statement to change the email address and first name of row 2 in CUSTOMER_01 to Catherine.Goodyear@somewhere.com and Catherine instead :

	EmailAddress	LastName	FirstName
1	Chris.Bancroft@somewhere.com	Bancroft	Chris
2	Katherine.Goodyear@somewhere.com	Goodyear	Katherine
3	Robert.Shire@somewhere.com	Shire	Robert

Write another SQL UPDATE statement to revert row 2 to the previous data.

2.2.3 SQL DML Delete statement

The SQL DML `delete` statement is a command to the DBMS to remove existing row(s) in a table.

Format:

```
delete from tableName where condition
```

The `where condition` is optional. Leaving it out will cause every row to be deleted.

An example of an `delete` statement is shown here:

```
delete from Artist where LastName = 'Smith'
```

A delete statement may result in zero or more rows being deleted.

When the DBMS executes the statement, all artists with last name Smith are deleted from the table Artist.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 352-361.

Activity 13

Write and run an SQL DELETE statement to delete row 2 in CUSTOMER_01:

	EmailAddress	LastName	FirstName
1	Chris.Bancroft@somewhere.com	Bancroft	Chris
2	Katherine.Goodyear@somewhere.com	Goodyear	Katherine
3	Robert.Shire@somewhere.com	Shire	Robert

Explain whether the row can be deleted.

2.2.4 SQL DML Select statement

The SQL DML `select` statement is a command to the DBMS to retrieve data in table(s). There are many variations of the `select` statement:

- Single table query

  Format:

  `select` *columnName*$_a$ *[as alias*$_a$*]*, …, *columnName*$_{a+n}$ *[as alias*$_{a+n}$*]*
  `from` *tableName*

```
where where-condition
order by columnNamec [DESC], …, columnNamec+p [DESC]
```

An example of such a query is shown here:

```
select FirstName as 'First Name', LastName as 'Last Name'
from Artist
where Nationality = 'United States'
order by LastName DESC, FirstName
```

Output:

	First Name	Last Name
1	Mark	Tobey
2	John Singer	Sargent
3	Paul	Horiuchi
4	Morris	Graves

When the DBMS executes the statement, it produces a resultset containing the first name and last name of artists with nationality United States from the table Artist.

If columns are given aliases, the column headers show the aliases instead of the column names.

The default display order in the `order by` clause is ASC or ascending. If DESC is specified, the display order is in descending order.

For this query, the resultset is displayed in descending order of the last name of artists, and in ascending order of first name if the artists have the same last name.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 76-107.

Activity 14

Write and run an SQL SELECT statement to show all rows in CUSTOMER_01:

	EmailAddress	LastName	FirstName
1	Chris.Bancroft@somewhere.com	Bancroft	Chris
2	Katherine.Goodyear@somewhere.com	Goodyear	Katherine
3	Robert.Shire@somewhere.com	Shire	Robert

- Non-correlated Subquery

  Format:

```
select columnNamea [as aliasa], …, columnNamea+n [as aliasa+n]
from tableName
where where-condition and
      columnNamei in (
            select columnNamei
            from tableName2
            where where-condition
      )
```

  The non-correlated subquery is used when the table in the `from` clause has all the required columns for display, but the table in the `from` clause does not include the column required in the `where` condition.

  Instead, the required column is found in another table, *tableName*$_2$. Thus, the inner query needs to handle the `where` condition.

  In a non-correlated subquery, the inner query can be processed independently of the outer query. The inner query is first evaluated to return a set of values which is used in the `where` clause of the outer query.

  For example,

```
select LastName, FirstName
from Artist
where dateOfBirth < 2000 and
```

```
      ArtistId in (
            select artistId
            from Work
            where medium = 'High Quality Limited Print'
      )
order by LastName DESC, FirstName
```

Output:

	LastName	FirstName
1	Tobey	Mark
2	Sargent	John Singer
3	Miro	Joan
4	Matisse	Henri
5	Klee	Paul
6	Kandinsky	Wassily
7	Horiuchi	Paul
8	Graves	Morris
9	Chagall	Marc

The table in the `select-from` clause, Artist has all the required columns for display, but it does not include the column required in the `where` condition, `where medium = 'High Quality Limited Print'`. Thus, we use an inner query to handle that `where` condition.

First, the inner query returns a set of artistId for artists whose work includes the medium high quality limited print. Next, the outer query picks up artist rows with birth year < 2000 and with artist id is in the set returned by the inner query - the set of artists whose work includes the medium high quality limited print.

Note that the table in the inner query and the table in the outer query must have at least a common column, or a column with the same domain. That common column links the tables in the outer and inner queries.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 119-121.

Activity 15

Insert this row into CUSTOMER_01.

	EmailAddress	LastName	FirstName
1	Chris.Croft@somewhere.com	Croft	Chris

This customer has no sales record yet.

Write and run an SQL SELECT statement to show customers who have not made any sales.

- Correlated Subquery

  Format:

```
select columnNamea [as aliasa], …, columnNamea+n [as aliasa+n]
from tableName t1
where columnNamei in (
      select columnNamei
      from tableName t2
      where where-conditionInvolvingt1Andt2
      )
```

  The correlated subquery is used when we need to determine whether some columns can become a new primary key of the table.

  Notice that both inner query and outer query use the same table, *tableName* but the table is referred to as *t1* in the outer query, and as *t2* in the inner query. In addition, the `where` condition involves both *t1* and *t2*.

  Because the `where` condition involves both *t1* and *t2*, the inner query cannot be processed independently of the outer query. The inner query is processed

as many times as the number of rows in the outer query. Each time the inner query is processed, it uses one row of the table in the outer query and checks it against all the rows in the inner query in the inner `where` condition. This processing repeats until there is not more row in the outer query that has not been processed in the inner query.

For example,

```
SELECT W1.Title, W1.Copy
FROM  WORK W1
WHERE W1.Title IN
      (SELECT W2.Title
       FROM WORK W2
       WHERE W1.Title = W2.Title AND W1.WorkID <> W2.WorkID)
```

This correlated subquery checks whether the title column in Work is unique.

To evaluate this subquery, the first row of `W1` is held, and then checked against all rows in `W2`, and if there is another row in `W2` with matching title as the row in `W1` but with different workid, the row in `W1` will show up in the resultset.

This is then repeated for the second row of `W1`, the third and so on, until all rows of `W1` are processed.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 429-433.

- Inner Join

Format:

```
select columnNamea [as aliasa], …, columnNamea+n [as aliasa+n]
from tableName1 t1 join tableName2 t2 on t1.someColumn =
t2.someColumn
where where-condition
```

The join is used when multiple tables must be combined to get all the required columns for display.

The multiple tables are specified in the `from` clause. The tables must be linked through common columns. An example of such a query is shown here:

```
select distinct LastName, FirstName, Title, Medium
from Artist A join Work W on A.artistId = W.artistId
order by Title
```

Output:

LastName	FirstName	Title	Medium
Klee	Paul	Angelus Novus	High Quality Limited Print
Tobey	Mark	Blue Interior	Tempera on card
Tobey	Mark	Broadway Boggie	High Quality Limited Print
Sargent	John Singer	Claude Monet Painting	High Quality Limited Print
Horiuchi	Paul	Color Floating in Time	High Quality Limited Print
Kandinsky	Wassily	Der Blaue Reiter	High Quality Limited Print
Tobey	Mark	Farmer's Market #2	High Quality Limited Print
Tobey	Mark	Forms in Progress I	Color aquatint
Tobey	Mark	Forms in Progress II	Color aquatint
Chagall	Marc	I and the Village	High Quality Limited Print
Horiuchi	Paul	Into Time	High Quality Limited Print
Miro	Joan	La Lecon de Ski	High Quality Limited Print
Horiuchi	Paul	Memories IV	Casein rice paper collage
Graves	Morris	Mid-Century Hibernation	High Quality Limited Print
Graves	Morris	Night Bird	Watercolor on Paper
Kandinsky	Wassily	On White II	High Quality Limited Print
Sargent	John Singer	Spanish Dancer	High Quality Limited Print
Graves	Morris	Sunflower	Watercolor and ink
Graves	Morris	Surf and Bird	Gouache
Graves	Morris	Surf and Bird	High Quality Limited Print
Matisse	Henri	The Dance	High Quality Limited Print
Chagall	Marc	The Fiddler	High Quality Limited Print
Miro	Joan	The Tilled Field	High Quality Limited Print
Tobey	Mark	The Woven World	Color lithograph
Tobey	Mark	Universal Field	High Quality Limited Print
Tobey	Mark	Untitled Number 1	Monotype with tempera
Matisse	Henri	Woman with a Hat	High Quality Limited Print
Horiuchi	Paul	Yellow Covers Blue	Oil and collage

The DBMS executes the query by matching each row in the Artist with every row in Work. If a row in Artist has a matching artistId as a row in Work, the columns of both tables are combined. If a row in Artist does not have a

matching artistId as a row in Work, or vice versa, those rows in both tables are not combined and will not show up in the resultset.

From this combined row, the last name and first name of the artist and the title and medium of Work are picked up. The keyword `distinct` removes duplicate rows from the resultset.

The time complexity of a join is $O(n^2)$ whereas that of a non-correlated subquery is O(n) where n is the number of rows in the table. Thus, a non-correlated subquery is more efficient, and hence, preferred whenever there is a subquery equivalent for a join.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 122-131.

- Outer Join

  Format:

  ```
  select columnNamea [as aliasa], …, columnNamea+n [as aliasa+n]
  from tableName1 t1 joinType join tableName2 t2 on t1.someColumn
  = t2.someColumn
  where where-condition
  ```

  joinType is either `left`, `right` or `full`.

  The difference between an inner join and an outer join is that when there is no match for a row in one table, the row can appear in the resultset, but with null for the columns in the other table.

  When the *joinType* is `left`, all rows in *tableName1* shows up in the resultset, and if there is no match, the columns in *tableName2* show null.

  When the *joinType* is `right`, all rows in *tableName2* shows up in the resultset, and if there is no match, the columns in *tableName1* show null.

When the *joinType* is `full`, all rows in $tableName_1$ and $tableName_2$ shows up in the resultset, and if there is no match, the columns in the other table show null.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 131-134.

Activity 16

Write and run an SQL SELECT statement to show customer data combined with their sales data. Show all customers even if they have not made any sales. Display the result in ascending order of email address as shown here.

	EmailAddress	LastName	FirstName	DateOfSale	SaleAmount
1	Chris.Bancroft@somewhere.com	Bancroft	Chris	2015-01-14	56.50
2	Chris.Croft@somewhere.com	Croft	Chris	NULL	NULL
3	Katherine.Goodyear@somewhere.com	Goodyear	Katherine	2015-01-17	34.25
4	Robert.Shire@somewhere.com	Shire	Robert	2015-01-14	234.00
5	Robert.Shire@somewhere.com	Shire	Robert	2015-01-16	123.00

- Group by and having clauses

  Format:

```
select columnNamea [as aliasa], …, columnNamea+n [as aliasa+n]
from tableNames
where where-condition
group by someColumns
having having-condition
order by columnNamec [DESC], …, columnNamec+p [DESC]
```

Note that the `from` clause may include table `join` and the `where` condition may include a subquery.

The `group by` clause is used when the rows in the dataset must be grouped so that statistics about each group can be obtained, using the built-in functions: sum, avg, max, min and count.

The `having` clause is used when the groups must satisfy some condition, before they can show up in in the resultset.

An example of such a query is shown here:

```
select LastName, FirstName, count(Title) as 'Number of Works'
from Artist A join Work W on A.artistId = W.artistId
where medium = 'High Quality Limited Print' and WorkId in (
      select WorkId
      from Trans
      where dateSold is null
      )
group by LastName, FirstName, Medium
having count(Title) >= 2
order by LastName, FirstName
```

Assume that these data rows are currently in the database:

	lastname	firstname	title	transactio...	medium	dateSold
1	Graves	Morris	Mid-Century Hibernation	181	High Quality Limited Print	NULL
2	Graves	Morris	Surf and Bird	252	High Quality Limited Print	NULL
3	Graves	Morris	Surf and Bird	253	High Quality Limited Print	NULL
4	Graves	Morris	Surf and Bird	254	High Quality Limited Print	NULL
5	Horiuchi	Paul	Color Floating in Time	229	High Quality Limited Print	NULL
6	Klee	Paul	Angelus Novus	126	High Quality Limited Print	NULL
7	Sargent	John Singer	Spanish Dancer	226	High Quality Limited Print	NULL
8	Tobey	Mark	Farmer's Market #2	155	High Quality Limited Print	NULL
9	Tobey	Mark	Universal Field	228	High Quality Limited Print	NULL

The example query produces this output:

	LastName	FirstName	Number of Works
1	Graves	Morris	4
2	Tobey	Mark	2

The inner query picks up unsold work (that is, *`dateSold is null`*). The `group by` clause puts the rows obtained from joining the tables Artist and Work into groups, based on whether they have the same last name, first name, and medium (`group by` *`LastName, FirstName, Medium`*).

The built-in count counts the number of unsold work titles that appear in each group, that is, the number of unsold work titles for each artist on a particular medium.

The having clause is a condition imposed on the group, in this case, the counts of unsold work titles must be 2 or more for the group statistics to show up in the resultset.

Without the `having` clause, the example query shows all all the groups, as shown in this output:

	LastName	FirstName	Number of Works
1	Graves	Morris	4
2	Horiuchi	Paul	1
3	Klee	Paul	1
4	Sargent	John Singer	1
5	Tobey	Mark	2

Note that a `group by` clause is independent on whether there is a `having` clause.

The `group by` clause is always necessary whenever a built-in function is used in the `select` clause and there are other columns in the `select` clause that the built-in function is not applied.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 107-118.

Activity 17

Write and run an SQL SELECT statement to show customer who have made at least 2 sales. Show the number of sales they have made and the combined total of sales as shown here:

	EmailAddress	LastNa...	FirstNa...	Number of Sales	Total Sales
1	Robert.Shire@somewhere.com	Shire	Robert	2	357.00

- `select` statements with set operators

  Format:

  `selectStatement`$_1$ *`setOperator selectStatement`*$_2$

  where the *`setOperator`* is either

  - `union`

    The result is all the rows resulting from either of `select` statements.

  - `intersect`

    The result is all the rows that are common for both `select` statements.

  - `except`

    The result is all in rows in the first `select` statement that do not result from the second `select` statement.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 134-137.

- `select` clause within SQL DML `insert`, `update` and `delete` statements
  - Bulk `insert` statement

    Format:

    ```
    insert into tableName(columnName1, …, columnNamen)

        select columnName1, …, columnNamen

    from tableNames
    where where-condition
    group by someColumns
    having having-condition
    order by columnNamec  [DESC], …, columnNamec+p [DESC]
    ```

    Note that the `from` clause may include table `join` and the `where` condition may include a subquery.

    This `insert` statement will cause all rows in the resultset of the `select` statement to be inserted into `tableName`.

  - `Update` statement
    - Format:

      ```
      update tableName set columnName1 = value1 , …,
      columnNamen = valuen
      from tableName t1 join tableName2 t2 on
      t1.someColumn = t2.someColumn
      where conditionOntableName2
      ```

      Note that the existing rows in *tableName* are updated if they satisfy the condition specified for another table, *tableName*$_2$.

    - Format:

      ```
      update tableName set columnName1 = (select
      columnName1 from tableName2 where condition1), …,
      columnNamen = (select columnNamen from tableNamen
      where conditionn)
      where conditionOntableName
      ```

      Note that the existing rows in *tableName* are updated with values found in other tables, *tableName*$_1$, …, *tableName*$_n$.

  - `delete` statement

Format:

```
delete from tableName
from tableName t1 join tableName2 t2 on t1.someColumn =
t2.someColumn
where conditionOntableName2
```

Note that the existing rows in *tableName* are deleted if they satisfy the condition specified for another table, *tableName*$_2$.

2.3 Views

A view is a virtual table, and forms one abstraction of the database.

Some uses of views are listed in Figure 3.13. Views are very useful as they can restrict access to only relevant data. Furthermore, when table implementation changes, the same abstraction still be presented, and so, the client code is not affected.

Uses of SQL Views
Hide columns or rows.
Display results of computations.
Hide complicated SQL syntax.
Layer built-in functions.
Provide level of isolation between table data and users' view of data.
Assign different processing permissions to different views of the same table.
Assign different triggers to different views of the same table.

Figure 3.13 Uses of SQL Views

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 7-17)

2.3.1 SQL DDL Create, Alter and Drop View statements

- `Create View`

A view is created with a `select` statement. The rows and columns of a view come from the physical tables or other views in the database.

Format:

```
create view viewName as
selectStatement
```

Once a view is defined, it is queried like a physical table.

Whenever a `select` statement on the view is executed, the view accesses the present content of the database. Thus, a the view presents the current state of the database.

Note that the row ordering for a table (both physical and virtual) is meaningless. Therefore, standard SQL does not permit an `order by` clause in the `select` statement for view definition.

- `Alter View`

  To alter or make changes to a view after it has been created, replace the keyword `create` with `alter`.

  Format:

  ```
  alter view viewName as
  selectStatement
  ```

- `Drop View`

  A view can be deleted

  Format:

  ```
  drop view viewName
  ```

Activity 18

Reproduced from Question 7.38 of the course text.

Write an SQL statement to create a view named Customer01DataView based on the CUSTOMER_01 table. Include the values of EmailAddress, LastName as CustomerLastName, and FirstName as CustomerFirstName, as shown here:

	EmailAddress	CustomerLastName	CustomerFirstName
1	Chris.Bancroft@somewhere.com	Bancroft	Chris
2	Katherine.Goodyear@somewhere.com	Goodyear	Katherine
3	Robert.Shire@somewhere.com	Shire	Robert

Run this statement to create the view, and then test the view by writing and running an appropriate SQL SELECT statement.

2.3.2 Inline View

An inline view can simplify a `select` statement. It is defined in the `from` clause and can be used only in the `select` statement associated with that `from` clause.

For example, the query makes use of an inline view, `sumTransCust`.

```
select lastName, firstName, sum(salesPrice)
from customer c join trans t on c.customerId = t.customerId
group by lastName, firstName
having sum(salesPrice) >=
      (select max(custTotalPurchase)
       from
            (select sum(salesPrice)  as custTotalPurchase
             from trans t
             group by customerId
            ) as sumTransCust
      )
```

Output:

	lastName	firstName	Max Sales Amount
1	Janes	Jeffrey	97500.00

Note that the inline view `sumTransCust` is defined in the `from` clause of the `select` statement:

```
select max(custTotalPurchase)
```

Thus, it is valid only for that `select` statement.

The inline view `sumTransCust` and its columns is not valid for the `select` statement:

```
select lastName, firstName, sum(salesPrice)
from customer c join trans t on c.customerId = t.customerId
```

The example query picks out the customers (last name and first name) who have spent the most amount of money purchasing art works from View Ridge Gallery.

The inline view computes the sum of the sales price in table trans grouped by customer id. The maximum of the sums is obtained by using the built-in function, `max` to pick up the highest sum of the rows in the inline view.

The maximum is then compared against the sum for each customer, to get the customer with the maximum sum of sale prices.

2.3.3 Updating Views

A view can be updated as if it is a physical table, causing the underlying rows in the physical tables to be updated. However, there is a caveat: the underlying rows must be identifiable. If they cannot be identified, the view is not updatable.

Figure 3.14 lists the guidelines for updating views.

Updatable Views
View based on a single table with no computed columns and all non-null columns present in the view.
View based on any number of tables, with or without computed columns, and INSTEAD OF trigger defined for the view.
Possibly Updatable Views
Based on a single table, primary key in view, some required columns missing from view, update and delete may be allowed. Insert is not allowed.
Based on multiple tables, updates may be allowed on the most subordinate table in the view if rows of that table can be uniquely identified.

Figure 3.14 Uses of SQL Views

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 7-19)

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 361-371.

Summary

In this study unit, we transformed a data model to a logical model, in preparation for implementation on a relational database.

Constructing a logical model from a data model is mechanical, the transformation steps and rules are standard. Each entity becomes a relation. Identifiers are primary keys. Foreign keys are introduced for the relationships. In a 1:1 relationship, the child relation is preferably the entity on the optional side. In a 1:N relationship, the child relation is the entity on the many side. In a N:M relationship, an intersection table, the child relation is introduced. The minimal cardinality actions are specified with the foreign key constraints on the child relation.

SQL DDL create statements are used to create a database and its tables to implement a logical model. Various data constraints can be defined for each column.

SQL DML statements such as insert, update and delete allow the data in the database to be manipulated. SQL DML select statements allow the data to be queried.

Views or virtual tables are abstraction of the database that have many uses such as hiding the implementation details from users. Views help isolate client code from being impacted by changes in database.

References

Book

Author(s)	Year	Book Title	Edition	Publisher
Kroenke, D., & Auer, D. J.	2016	*Database Processing – Fundamentals, Design, and Implementation*	14	Pearson

Quiz

1. Which statement is true about transforming entity to table?

a. Make the attributes of the entity the rows in the database table.

b. Examine each entity according to normalization criteria before creating a table from it.

c. Ensure every table has at least one non-key column.

*d. Make entities and attributes the tables and their columns.

2. Which statement is true about foreign keys?

*a. In a relational database design, all relationships are expressed by creating a foreign key.

b. To represent a 1:1 relationship between two strong entities, it is preferable that the key of optional table is placed into the other table.

c. In a 1:1 relationship, the null status Not Null is how the DBMS enforce uniqueness on the foreign key.

d. When placing a foreign key for a 1:N relationship, the key of either table can be used as the foreign key in the other table.

3. Which SQL keyword is required to make a query?

a. ORDER BY

*b. SELECT

c. DISTINCT

d. WHERE

4. Which SQL keyword is used with built-in functions to group together rows that have the same value in a specified column?

*a. GROUP BY

b. ORDER BY

c. SORT BY

d. HAVING

5. Which SQL keyword makes a new table and describes its columns?

a. SET

*b. CREATE

c. SELECT

d. ALTER

6. Given the SQL statement:

```
CREATE TABLE Student (
StdNo          int             NOT NULL,
Name           char(35)        NOT NULL,
Email          varchar(64)     NOT NULL,
DateOfBirth    date            NOT NULL,

CONSTRAINT     StudentPK       PRIMARY KEY (StdNo),
CONSTRAINT     StudentAK       UNIQUE (Email)
);
```

What conclusion can be made about the column Email?

a. Email is the primary key

b. Email is a foreign key

*c. Email is a candidate key

d. Email is a surrogate key

Formative Assessment

1. Which statement is true about the referential integrity constraint policy that guarantees that a row in a child table always has a required entry in a parent table?

a. This referential integrity constraint is specified in the parent table.

Incorrect. The foreign key constraint is specified in the child table. Refer to textbook page 310.

b. This referential integrity constraint is enforced when adding a new row to a parent table.

Incorrect. It is enforced when adding a new row to a child table. Refer to textbook page 390.

c. This referential integrity constraint specifies the minimum cardinality enforcement actions.

Correct! Minimum cardinality enforcement action specifies the action required for the different minimum cardinality requirement for parent and child tables. Refer to textbook page 390.

d. This referential integrity constraint cannot be enforced in all DBMS products

Incorrect. Required parent is supported by some DBMS simply by having a foreign key constraint specified in the child table. Refer to textbook page 390.

2. Given the tables

OrderItem(orderNo, sku, quantityOrdered, Date)

Product(sku, description, quantityOnHand)

What type of SQL select statement is required to display product sku and the number of times the product has been ordered?

a. A subquery

Incorrect. A subquery can display data from one table. The query requires data from both tables to be displayed. Refer to textbook page 127.

b. A join

Correct! Join can display data from both tables. Refer to textbook page 122-127.

c. A select statement on just the Product table

Incorrect. A select statement on Product can display data from the Product table only. But to display the number of times the product was ordered requires a COUNT on the rows in the OrderItem table. Refer to textbook page 127.

d. All three are possible.

@ Incorrect. Only join can display data from both tables. Refer to textbook page 127.

3. Which statement is false about recursive relationships?

a. The rows of a single table can play two different roles.

Incorrect. This statement is true. Each row can play the roles of the parent and a child when there is a recursive relationship. Refer to textbook page 304

b. The techniques for representing the tables are the same as for non-recursive relationships except the rows are in the same table.

Incorrect. This statement is true. Think of the entity as two tables: the parent and a child. Refer to textbook page 303.

c. Recursive relationships can be 1:1, 1:N, or N:M relationships.

Incorrect. This statement is true. Recursive relationships are like relationships between strong entities, except the relationship is between itself. Refer to textbook page 303.

d. A new table must be defined to represent all recursive relationships.

Correct! This statement is false. Recursive relationships are like relationships between strong entities, except the relationship is between itself. Introduce a new table only for N:M relationships. Refer to textbook page 304.

4. Which SQL keyword imposes restrictions on a table, data and relationship?

a. CREATE

Incorrect. CREATE TABLE defines a table and its columns. Restrictions are defined using the CONSTRAINT keyword. Refer to textbook page 338.

b. SELECT

Incorrect. SELECT is the keyword for making query. Refer to textbook page 77.

c. ALTER

Incorrect. ALTER is used to make changes to an existing table. Refer to textbook page 349.

d. CONSTRAINT

Correct! The CONSTRAINT keyword is used to specify restrictions. Refer to textbook pages 342-343.

5. What advantage is not gained from implementing a view?

a. Greater security

Incorrect. This is an advantage of implementing views. Views can hide rows and columns. In addition, different processing permissions can be assigned to different views of the same table. Refer to textbook pages 352-353.

b. User code is minimally affected by changes in the database

Incorrect. This is an advantage of implementing views. Views provide a level of isolation between tables and users' view of data. Refer to textbook pages 352-353.

c. Simplify code in SQL query

Incorrect. This is an advantage of implementing views. A view can hide complicated SQL syntax and layer built-in functions. Refer to textbook pages 352-353.

d. Allow multi-layer SQL queries to be written

Correct! This is not an advantage of implementing views. Rather, views may be implemented over views, thus layering levels of views. Refer to textbook pages 352-353.

Solutions or Suggested Answers

Activity 1

The logical or conceptual design describes the tables and relations that are deducted from the ER-model. The physical design however, adds constructor specifics. Therefore one logical design can be transformed into several constructor specific database implementations.

Activity 2

A 1:1 strong entity relationship can be represented by placing one of the primary keys as a foreign key in either of the two entities. Thus, there are two ways to represent the relationship. However, one of these may be preferred depending on maximum cardinality requirements.

The relationship between CAR and AGENT in the example is a 1:1 strong entity relationship. The relationship can be represented by:

(1) Placing the primary key of AGENT in CAR as a foreign key, or

(2) Placing the primary key of CAR in AGENT as a foreign key.

Activity 3

The trick for representing N:M recursive relationships is to decompose the N:M relationship into two 1:N relationships. We do this by creating an intersection table, just as we did for N:M relationships between strong entities.

Activity 4

A 1:N recursive relationship is represented just like a 1:N strong entity relationship. We place the primary key of the entity acting as the parent entity (the "1" entity) in the entity itself acting as child entity (the "N" entity) as a foreign key.

This is like a 1:N strong entity relationship. The relationship can be represented by placing the primary key of CUSTOMER in CUSTOMER as the foreign key Referrer. Name the foreign key column referring customer.

Activity 5

O-M stands for Optional Mandatory. This means that the parent relation is not obliged while the child is. Example: an employee is a person that works optional for a department, although a department can't exist without employees. M-O stands for Mandatory Optional. This means that the parent table is obliged but the child is not. Example: an invoice detail row can't exist without an invoice header.

Activity 6

The DBMS cannot be used to enforce required children because there is no straightforward way to make sure that appropriate child row foreign keys exist, nor that relationships stay valid upon INSERT, UPDATE and DELETE actions.

Activity 7

(1) M-O design decisions – The M-O design decisions are based on a mandatory parent.

If the primary key is a surrogate key (or equivalent), updates should be prohibited since the key will never change. Otherwise, cascade updates from the parent primary key to the child foreign key.

If the parent and child entities are strong entities, prohibit carrying the deletion of the parent into the child. If the child is a weak entity, cascade the deletion when the parent is deleted.

When a child row is inserted, it must have a valid foreign key value and the policy will specify how the foreign key value is specified.

(2) O-M design decisions — The O-M design decisions are based on a mandatory child.

If the child is required, there must be a child row associated with a parent row as soon as the parent row is created. The policy will specify how this is done.

If a parent has a required child, a change to the parent key will have to be associated with the change of at least one child foreign key. However, if the parent primary key is a surrogate, it will never change or updates should be prohibited.

The foreign key of the child can be changed (indicating a reassignment to another parent) except that the last child cannot be reassigned because this would strand the parent without a required child.

The same logic applies to deletions in the child. The last associated child of a parent cannot be deleted unless the parent is also deleted.

(3) M-M design decisions — The M-M design decisions are based on both a mandatory parent and a mandatory child.

All of the above considerations apply. However, the requirements that a parent obtain a required child and that a child obtain a required parent simultaneously will create a conflict that will require application logic (in triggers) to be designed to handle the conflict.

A similar problem occurs when deleting the last associated rows from required parents and children, and similar application logic will have to be designed to handle the situation.

Activity 8

The IDENTITY keyword is used to modify a column name, and is used to specify surrogate keys. The first number parameter after IDENTITY specifies the starting value for the surrogate key, and the second number specifies the increment value for each additional record. Thus a column named RelationID and modified by IDENTITY (4000, 5) will be a surrogate key named RelationID with an initial value of 4000 (for the first record in the relation), and with following values incremented by 5: 4000, 4005, 4010, etc.

Activity 9

```
CREATE TABLE SALE_01(
  SaleID           INT                NOT NULL IDENTITY(20150001, 1),
  DateOfSale       DATE               NOT NULL,
  EmailAddress     VARCHAR(100)       NOT NULL,
  SaleAmount       NUMERIC(7,2)       NOT NULL,
  CONSTRAINT       SALE_PK_01         PRIMARY KEY(SaleID),
  CONSTRAINT       S_01_C_01_FK       FOREIGN KEY(EmailAddress)
                          REFERENCES CUSTOMER_01(EmailAddress)
                                ON UPDATE CASCADE
  );
```

Activity 10

No, because the primary key CUSTOMER_01.EmailAddress must be created before the foreign key SALE_01.EmailAddress.

Activity 11

```
INSERT INTO CUSTOMER_01 (EmailAddress, LastName, FirstName)
       VALUES('Robert.Shire@somewhere.com','Shire', 'Robert');
INSERT INTO CUSTOMER_01 (EmailAddress, LastName, FirstName)
       VALUES('Katherine.Goodyear@somewhere.com','Goodyear', 'Katherine');
INSERT INTO CUSTOMER_01 (EmailAddress, LastName, FirstName)
       VALUES('Chris.Bancroft@somewhere.com','Bancroft', 'Chris');
```

Activity 12

```
update CUSTOMER_01
set EmailAddress = 'Catherine.Goodyear@somewhere.com',
FirstName = 'Catherine'
where EmailAddress = 'Katherine.Goodyear@somewhere.com'

update CUSTOMER_01
set EmailAddress = 'Katherine.Goodyear@somewhere.com',
FirstName = 'Katherine'
where EmailAddress = 'Catherine.Goodyear@somewhere.com'
```

Activity 13

```
delete from CUSTOMER_01
where EmailAddress = 'Katherine.Goodyear@somewhere.com'
```

As there is a child row for Katherine.Goodyear@somewhere.com in Sales_01, and the default minimal cardinality action is on delete no action, the delete operation will be unsuccessful when executed.

Activity 14

```
select * from CUSTOMER_01
```

Activity 15

```
INSERT INTO CUSTOMER_01 (EmailAddress, LastName, FirstName)
VALUES('Chris.Croft@somewhere.com','Croft', 'Chris');

select * from CUSTOMER_01 c
where c.EmailAddress not in (
select EmailAddress from SALE_01)
```

Activity 16

```
select c.EmailAddress, LastName, FirstName, DateOfSale  , SaleAmount
from CUSTOMER_01 c left join SALE_01 s on c.EmailAddress = s.EmailAddress
order by c.EmailAddress
```

Activity 17

```
select c.EmailAddress, LastName, FirstName, count(DateOfSale) as 'Number of
Sales' , sum(SaleAmount) as 'Total Sales'
from CUSTOMER_01 c left join SALE_01 s on c.EmailAddress = s.EmailAddress
group by c.EmailAddress, LastName, FirstName
having count(DateOfSale) >= 2
```

Activity 18

```
CREATE VIEW Customer01DataView AS
      SELECT EmailAddress,
             LastName as CustomerLastName,
             FirstName as CustomerFirstName
      FROM   CUSTOMER_01;
```