Chapter 10 ER Modelling

ADRIENNE WATT

One important theory developed for the entity relational (ER) model involves the notion of functional dependency (FD). The aim of studying this is to improve your understanding of relationships among data and to gain enough formalism to assist with practical database design.

Like constraints, FDs are drawn from the semantics of the application domain. Essentially, *functional dependencies* describe how individual attributes are related. FDs are a kind of constraint among attributes within a relation and contribute to a good relational schema design. In this chapter, we will look at:

- The basic theory and definition of functional dependency
- The methodology for improving schema designs, also called normalization

Relational Design and Redundancy

Generally, a good relational database design must capture all of the necessary attributes and associations. The design should do this with a minimal amount of stored information and no redundant data.

In database design, redundancy is generally undesirable because it causes problems maintaining consistency after updates. However, redundancy can sometimes lead to performance improvements; for example, when redundancy can be used in place of a *join* to connect data. A *join* is used when you need to obtain information based on two related tables.

Consider Figure 10.1: customer 1313131 is displayed twice, once for account no. A-101 and again for account A-102. In this case, the customer number is not redundant, although there are deletion anomalies with the table. Having a separate customer table would solve this problem. However, if a branch address were to change, it would have to be updated in multiple places. If the customer number was left in the table as is, then you wouldn't need a branch table and no join would be required, and performance is improved .

accountNo	balance	customer	branch	address	assets
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	600	9876543	Round Hil	Horseneck	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hil	Horseneck	8000000

Bank Accounts

Figure 10.1. An example of redundancy used with bank accounts and branches.

Insertion Anomaly

An *insertion anomaly* occurs when you are inserting inconsistent information into a table. When we insert a new record, such as account no. A-306 in Figure 10.2, we need to check that the branch data is consistent with existing rows.

accountNo	balance	customer	branch	address	assets
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	600	9876543	Round Hil	Horseneck	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hil	Horseneck	8000000
A-306	800	1111111	Round Hil	Horseneck	8000800

Insertion anomaly - Insert account A-306 at Round Hill

Figure 10.2. Example of an insertion anomaly.

Update Anomaly

If a branch changes address, such as the Round Hill branch in Figure 10.3, we need to update all rows referring to that branch. Changing existing information incorrectly is called an *update anomaly*.

accountNo	balance	customer	branch	address	assets
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	600	9876543	Round Hil	Palo Alto	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hil	Horseneck	8000000

Update Anomaly - Round Hill branch address

Figure 10.3. Example of an update anomaly.

Deletion Anomaly

A *deletion anomaly* occurs when you delete a record that may contain attributes that shouldn't be deleted. For instance, if we remove information about the last account at a branch, such as account A-101 at the Downtown branch in Figure 10.4, all of the branch information disappears.

accountNo	balance	customer	branch	address	assets
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	600	9876543	Round Hill	Horseneck	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000

Deletion anomaly - Bank Account

Figure 10.4. Example of a deletion anomaly.

The problem with deleting the A-101 row is we don't know where the Downtown branch is located and we lose all information regarding customer 1313131. To avoid these kinds of update or deletion problems, we need to decompose the original table into several smaller tables where each table has minimal overlap with other tables.

Each bank account table must contain information about one entity only, such as the Branch or Customer, as displayed in Figure 10.5.

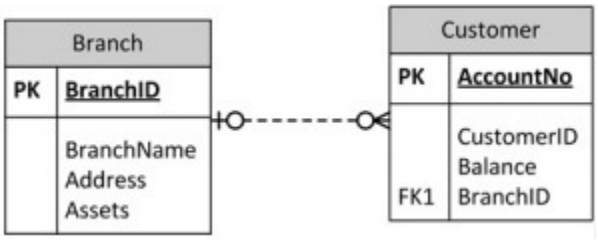


Figure 10.5. Examples of bank account tables that contain one entity each, by A. Watt.

Following this practice will ensure that when branch information is added or updated it will only affect one record. So, when customer information is added or deleted, the branch information will not be accidentally modified or incorrectly recorded.

Example: employee project table and anomalies

Figure 10.6 shows an example of an employee project table. From this table, we can assume that:

1. EmpID and ProjectID are a composite PK.
2. Project ID determines Budget (i.e., Project P1 has a budget of 32 hours).

EmpID	Budget	ProjectID	Hours
S75	32	P1	7
S75	40	P2	3
S79	32	P1	4
S79	27	P3	1
S80	40	P2	5
	17	P4	

Figure 10.6. Example of an employee project table, by A. Watt.

Next, let's look at some possible anomalies that might occur with this table during the following steps.

1. Action: Add row {S85,35,P1,9}
2. Problem: There are two tuples with conflicting budgets
3. Action: Delete tuple {S79, 27, P3, 1}
4. Problem: Step #3 deletes the budget for project P3
5. Action: Update tuple {S75, 32, P1, 7} to {S75, 35, P1, 7}
6. Problem: Step #5 creates two tuples with different values for project P1's budget
7. Solution: Create a separate table, each, for Projects and Employees, as shown in Figure 10.7.

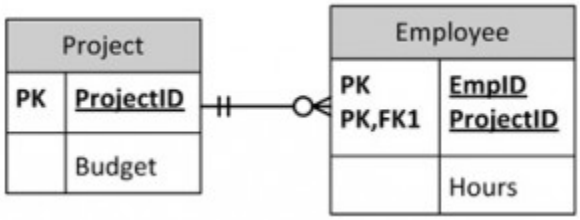


Figure 10.7. Solution: separate tables for Project and Employee, by A. Watt.

How to Avoid Anomalies

The best approach to creating tables without anomalies is to ensure that the tables are normalized, and that's accomplished by understanding functional dependencies. FD ensures that all attributes in a table belong to that table. In other words, it will eliminate redundancies and anomalies.

Example: separate Project and Employee tables

Project table

ProjectID	Budget
P1	32
P2	40
P3	27
P4	17

Employee table

EmpID	ProjectID	Hours
S75	P1	7
S75	P2	3
S79	P1	4
S79	P3	1
S80	P2	5

Figure 10.8. Separate Project and Employee tables with data, by A. Watt.

By keeping data separate using individual Project and Employee tables:

1. No anomalies will be created if a budget is changed.
2. No dummy values are needed for projects that have no employees assigned.
3. If an employee's contribution is deleted, no important data is lost.
4. No anomalies are created if an employee's contribution is added.

Key Terms

deletion anomaly: occurs when you delete a record that may contain attributes that shouldn't be deleted

functional dependency (FD): describes how individual attributes are related

insertion anomaly: occurs when you are inserting inconsistent information into a table

join: used when you need to obtain information based on two related tables

update anomaly: changing existing information incorrectly

Exercises

1. Normalize Figure 10.9.

Attribute Name	Sample Value	Sample Value	Sample Value
StudentID	1	2	3
StudentName	John Smith	Sandy Law	Sue Rogers
CourseID	2	2	3
CourseName	Programming Level 1	Programming Level 1	Business
Grade	75%	61%	81%
CourseDate	Jan 5th, 2014	Jan 5th, 2014	Jan 7th, 2014

Figure 10.9. Table for question 1, by A. Watt.

2. Create a logical ERD for an online movie rental service (no many to many relationships). Use the following description of operations on which your business rules must be based:The online movie rental service classifies movie titles according to their type: comedy, western, classical, science fiction, cartoon, action, musical, and new release. Each type contains many possible titles, and most titles within a type are available in multiple copies. For example, note the following summary:TYPE TITLE
   Musical My Fair Lady (Copy 1)
   My Fair Lady (Copy 2)
   Oklahoma (Copy 1)
   Oklahoma (Copy 2)
   Oklahoma (Copy 3)
   etc.
3. What three data anomalies are likely to be the result of data redundancy? How can such anomalies be eliminated?

Also see *Appendix B: Sample ERD Exercises*

Attribution

The following material was written by Adrienne Watt:

1. Example: employee project table and anomalies
2. How to Avoid Anomalies
3. Key Terms
4. Exercises

Chapter 11 Functional Dependencies

ADRIENNE WATT

A *functional dependency* (FD) is a relationship between two attributes, typically between the PK and other non-key attributes within a table. For any relation R, attribute Y is functionally dependent on attribute X (usually the PK), if for every valid instance of X, that value of X uniquely determines the value of Y. This relationship is indicated by the representation below :

X ———-> Y

The left side of the above FD diagram is called the *determinant*, and the right side is the *dependent*. Here are a few examples.

In the first example, below, SIN determines Name, Address and Birthdate. Given SIN, we can determine any of the other attributes within the table.

SIN ———-> Name, Address, Birthdate

For the second example, SIN and Course determine the date completed (DateCompleted). This must also work for a composite PK.

SIN, Course ———> DateCompleted

The third example indicates that ISBN determines Title.

ISBN ———-> Title

Rules of Functional Dependencies

Consider the following table of data r(R) of the relation schema R(ABCDE) shown in Table 11.1.

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b1	C2	d2	e1
a3	b2	C1	d1	e1
a4	b2	C2	d2	e1
a5	b3	C3	d1	e1

Table R

Table 11.1. Functional dependency example, by A. Watt.

As you look at this table, ask yourself: *What kind of dependencies can we observe among the attributes in Table* R? Since the values of A are unique (a1, a2, a3, etc.), it follows from the FD definition that:

$A \rightarrow B, \quad A \rightarrow C, \quad A \rightarrow D, \quad A \rightarrow E$

- It also follows that $A \rightarrow BC$ (or any other subset of ABCDE).
- This can be summarized as $A \rightarrow BCDE$.
- From our understanding of primary keys, A is a primary key.

Since the values of E are always the same (all e1), it follows that:

$A \rightarrow E, \quad B \rightarrow E, \quad C \rightarrow E, \quad D \rightarrow E$

However, we cannot generally summarize the above with $ABCD \rightarrow E$ because, in general, $A \rightarrow E, \quad B \rightarrow E, \quad AB \rightarrow E$.

Other observations:

1. Combinations of BC are unique, therefore $BC \rightarrow ADE$.
2. Combinations of BD are unique, therefore $BD \rightarrow ACE$.
3. If C values match, so do D values.
    1. Therefore, $C \rightarrow D$
    2. However, D values don't determine C values
    3. So C does not determine D, and D does not determine C.

Looking at actual data can help clarify which attributes are dependent and which are determinants.

Inference Rules

Armstrong's axioms are a set of inference rules used to infer all the functional dependencies on a relational database.

They were developed by William W. Armstrong. The following describes what will be used, in terms of notation, to explain these axioms.

Let R(U) be a relation scheme over the set of attributes U. We will use the letters X, Y, Z to represent any subset of and, for short, the union of two sets of attributes, instead of the usual X U Y.

Axiom of reflexivity

This axiom says, if Y is a subset of X, then X determines Y (see Figure 11.1).

$$\text{If } Y \subseteq X, \text{ then } X \rightarrow Y$$

Figure 11.1. Equation for axiom of reflexivity.

For example, **PartNo —> NT123** where X (PartNo) is composed of more than one piece of information; i.e., Y (NT) and partID (123).

Axiom of augmentation

The axiom of augmentation, also known as a partial dependency, says if X determines Y, then XZ determines YZ for any Z (see Figure 11.2).

$$\textbf{If } X \rightarrow Y, \textbf{ then } XZ \rightarrow YZ \textbf{ for any } Z$$

Figure 11.2. Equation for axiom of augmentation.

The axiom of augmentation says that every non-key attribute must be fully dependent on the PK. In the example shown below, StudentName, Address, City, Prov, and PC (postal code) are only dependent on the StudentNo, not on the StudentNo and Grade.

StudentNo, Course —> StudentName, Address, City, Prov, PC, Grade, DateCompleted

This situation is not desirable because every non-key attribute has to be fully dependent on the PK. In this situation, student information is only partially dependent on the PK (StudentNo).

To fix this problem, we need to break the original table down into two as follows:

- Table 1: StudentNo, Course, Grade, DateCompleted
- Table 2: StudentNo, StudentName, Address, City, Prov, PC

Axiom of transitivity

The axiom of transitivity says if X determines Y, and Y determines Z, then X must also determine Z (see Figure 11.3).

$$\textbf{If } X \rightarrow Y \textbf{ and } Y \rightarrow Z \textbf{, then } X \rightarrow Z$$

Figure 11.3. Equation for axiom of transitivity.

The table below has information not directly related to the student; for instance, ProgramID and ProgramName should have a table of its own. ProgramName is not dependent on StudentNo; it's dependent on ProgramID.

StudentNo —> StudentName, Address, City, Prov, PC, ProgramID, ProgramName

This situation is not desirable because a non-key attribute (ProgramName) depends on another non-key attribute (ProgramID).

To fix this problem, we need to break this table into two: one to hold information about the student and the other to hold information about the program.

- Table 1: StudentNo —> StudentName, Address, City, Prov, PC, ProgramID
- Table 2: ProgramID —> ProgramName

However we still need to leave an FK in the student table so that we can identify which program the student is enrolled in.

Union

This rule suggests that if two tables are separate, and the PK is the same, you may want to consider putting them together. It states that if X determines Y and X determines Z then X must also determine Y and Z (see Figure 11.4).

$$\textbf{If } X \rightarrow Y \textbf{ and } X \rightarrow Z \textbf{ then } X \rightarrow YZ$$

Figure 11.4. Equation for the Union rule.

For example, if:

- SIN —> EmpName
- SIN —> SpouseName

You may want to join these two tables into one as follows:

SIN –> EmpName, SpouseName

Some database administrators (DBA) might choose to keep these tables separated for a couple of reasons. One, each table describes a different entity so the entities should be kept apart. Two, if SpouseName is to be left NULL most of the time, there is no need to include it in the same table as EmpName.

Decomposition

Decomposition is the reverse of the Union rule. If you have a table that appears to contain two entities that are determined by the same PK, consider breaking them up into two tables. This rule states that if X determines Y and Z, then X determines Y and X determines Z separately (see Figure 11.5).

$$\text{If } X \rightarrow YZ \text{ then } X \rightarrow Y \text{ and } X \rightarrow Z$$

Figure 11.5. Equation for decompensation rule.

Dependency Diagram

A dependency diagram, shown in Figure 11.6, illustrates the various dependencies that might exist in a *non-normalized table*. A non-normalized table is one that has data redundancy in it.

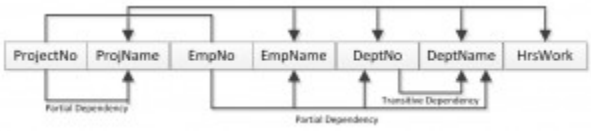


Figure 11.6. Dependency diagram.

The following dependencies are identified in this table:

- ProjectNo and EmpNo, combined, are the PK.
- Partial Dependencies:
  - ProjectNo —> ProjName
  - EmpNo —> EmpName, DeptNo,
  - ProjectNo, EmpNo —> HrsWork
- Transitive Dependency:
  - DeptNo —> DeptName

Key Terms

Armstrong's axioms: a set of inference rules used to infer all the functional dependencies on a relational database**DBA**: database administrator
decomposition: a rule that suggests if you have a table that appears to contain two entities that are determined by the same PK, consider breaking them up into two tables

dependent: the right side of the functional dependency diagram

determinant: the left side of the functional dependency diagram

functional dependency (FD): a relationship between two attributes, typically between the PK and other non-key attributes within a table

non-normalized table: a table that has data redundancy in it

Union: a rule that suggests that if two tables are separate, and the PK is the same, consider putting them together

Exercises

See Chapter 12.

Attributions

This chapter of *Database Design* (including images, except as otherwise noted) is a derivative copy of Armstrong's axioms by Wikipedia the Free Encyclopedia licensed under Creative Commons Attribution-ShareAlike 3.0 Unported

The following material was written by Adrienne Watt:

1. some of Rules of Functional Dependencies
2. Key Terms

Chapter 12 Normalization

ADRIENNE WATT

Normalization should be part of the database design process. However, it is difficult to separate the normalization process from the ER modelling process so the two techniques should be used concurrently.

Use an entity relation diagram (ERD) to provide the big picture, or macro view, of an organization's data requirements and operations. This is created through an iterative process that involves identifying relevant entities, their attributes and their relationships.

Normalization procedure focuses on characteristics of specific entities and represents the micro view of entities within the ERD.

What Is Normalization?

Normalization is the branch of relational theory that provides design insights. It is the process of determining how much redundancy exists in a table. The goals of normalization are to:

- Be able to characterize the level of redundancy in a relational schema
- Provide mechanisms for transforming schemas in order to remove redundancy

Normalization theory draws heavily on the theory of functional dependencies. Normalization theory defines six normal forms (NF). Each normal form involves a set of dependency properties that a schema must satisfy and each normal form gives guarantees about the presence and/or absence of update anomalies. This means that higher normal forms have less redundancy, and as a result, fewer update problems.

Normal Forms

All the tables in any database can be in one of the normal forms we will discuss next. Ideally we only want minimal redundancy for PK to FK. Everything else should be derived from other tables. There are six normal forms, but we will only look at the first four, which are:

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)
- Boyce-Codd normal form (BCNF)

BCNF is rarely used.

First Normal Form (1NF)

In the *first normal form*, only single values are permitted at the intersection of each row and column; hence, there are no repeating groups.

To normalize a relation that contains a repeating group, remove the repeating group and form two new relations.

The PK of the new relation is a combination of the PK of the original relation plus an attribute from the newly created relation for unique identification.

Process for 1NF

We will use the **Student_Grade_Report** table below, from a School database, as our example to explain the process for 1NF.

Student_Grade_Report (StudentNo, StudentName, Major, CourseNo, CourseName, InstructorNo, InstructorName, InstructorLocation, Grade)

- In the Student Grade Report table, the repeating group is the course information. A student can take many courses.
- Remove the repeating group. In this case, it's the course information for each student.
- Identify the PK for your new table.
- The PK must uniquely identify the attribute value (StudentNo and CourseNo).
- After removing all the attributes related to the course and student, you are left with the student course table (**StudentCourse**).
- The Student table (**Student**) is now in first normal form with the repeating group removed.
- The two new tables are shown below.

Student (StudentNo, StudentName, Major)

StudentCourse (StudentNo, CourseNo, CourseName, InstructorNo, InstructorName, InstructorLocation, Grade)

How to update 1NF anomalies

StudentCourse (StudentNo, CourseNo, CourseName, InstructorNo, InstructorName, InstructorLocation, Grade)

- To add a new course, we need a student.
- When course information needs to be updated, we may have inconsistencies.
- To delete a *student*, we might also delete critical information about a course.

Second Normal Form (2NF)

For the *second normal form*, the relation must first be in 1NF. The relation is automatically in 2NF if, and only if, the PK comprises a single attribute.

If the relation has a composite PK, then each non-key attribute must be fully dependent on the entire PK and not on a subset of the PK (i.e., there must be no partial dependency or augmentation).

Process for 2NF

To move to 2NF, a table must first be in 1NF.

- The Student table is already in 2NF because it has a single-column PK.
- When examining the Student Course table, we see that not all the attributes are fully dependent on the PK; specifically, all course information. The only attribute that is fully dependent is grade.
- Identify the new table that contains the course information.
- Identify the PK for the new table.
- The three new tables are shown below.

Student (StudentNo, StudentName, Major)

CourseGrade (StudentNo, CourseNo, Grade)

CourseInstructor (CourseNo, CourseName, InstructorNo, InstructorName, InstructorLocation)

How to update 2NF anomalies

- When adding a new instructor, we need a course.
- Updating course information could lead to inconsistencies for instructor information.
- Deleting a course may also delete instructor information.

Third Normal Form (3NF)

To be in *third normal form*, the relation must be in second normal form. Also all transitive dependencies must be removed; a non-key attribute may not be functionally dependent on another non-key attribute.

Process for 3NF

- Eliminate all dependent attributes in transitive relationship(s) from each of the tables that have a transitive relationship.
- Create new table(s) with removed dependency.
- Check new table(s) as well as table(s) modified to make sure that each table has a determinant and that no table contains inappropriate dependencies.
- See the four new tables below.

Student (StudentNo, StudentName, Major)

CourseGrade (StudentNo, CourseNo, Grade)

Course (CourseNo, CourseName, InstructorNo)

Instructor (InstructorNo, InstructorName, InstructorLocation)

At this stage, there should be no anomalies in third normal form. Let's look at the dependency diagram (Figure 12.1) for this example. The first step is to remove repeating groups, as discussed above.

Student (StudentNo, StudentName, Major)

StudentCourse (StudentNo, CourseNo, CourseName, InstructorNo, InstructorName, InstructorLocation, Grade)

To recap the normalization process for the School database, review the dependencies shown in Figure 12.1.

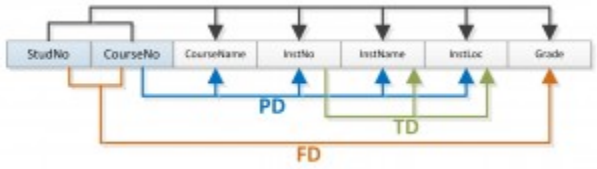


Figure 12.1 Dependency diagram, by A. Watt.

The abbreviations used in Figure 12.1 are as follows:

- PD: partial dependency
- TD: transitive dependency
- FD: full dependency (Note: FD typically stands for **functional** dependency. Using FD as an abbreviation for full dependency is only used in Figure 12.1.)

Boyce-Codd Normal Form (BCNF)

When a table has more than one candidate key, anomalies may result even though the relation is in 3NF. *Boyce-Codd normal form* is a special case of 3NF. A relation is in BCNF if, and only if, every determinant is a candidate key.

BCNF Example 1

Consider the following table (**St_Maj_Adv**).

Student_id	Major	Advisor
111	Physics	Smith
111	Music	Chan
320	Math	Dobbs
671	Physics	White
803	Physics	Smith

The *semantic rules* (business rules applied to the database) for this table are:

1. Each Student may major in several subjects.
2. For each Major, a given Student has only one Advisor.
3. Each Major has several Advisors.
4. Each Advisor advises only one Major.
5. Each Advisor advises several Students in one Major.

The functional dependencies for this table are listed below. The first one is a candidate key; the second is not.

1. Student_id, Major ——> Advisor
2. Advisor ——> Major

Anomalies for this table include:

1. Delete – student deletes advisor info
2. Insert – a new advisor needs a student
3. Update – inconsistencies

Note: No single attribute is a candidate key.

PK can be Student_id, Major or Student_id, Advisor**.**

To reduce the **St_Maj_Adv** relation to BCNF, you create two new tables:

1. **St_Adv** (Student_id, Advisor)
2. **Adv_Maj** (Advisor, Major)

St_Adv table

Student_id	**Advisor**
111	Smith
111	Chan
320	Dobbs
671	White
803	Smith

Adv_Maj table

Advisor	**Major**
Smith	Physics
Chan	Music
Dobbs	Math
White	Physics

BCNF Example 2

Consider the following table (**Client_Interview)**.

ClientNo	InterviewDate	InterviewTime	StaffNo	RoomNo
CR76	13-May-02	10.30	SG5	G101
CR56	13-May-02	12.00	SG5	G101
CR74	13-May-02	12.00	SG37	G102
CR56	1-July-02	10.30	SG5	G102

FD1 – ClientNo, InterviewDate –> InterviewTime, StaffNo, RoomNo (PK)

FD2 – staffNo, interviewDate, interviewTime –> clientNO (candidate key: CK)

FD3 – roomNo, interviewDate, interviewTime –> staffNo, clientNo (CK)

FD4 – staffNo, interviewDate –> roomNo

A relation is in BCNF if, and only if, every determinant is a candidate key. We need to create a table that incorporates the first three FDs (**Client_Interview2** table) and another table (**StaffRoom** table) for the fourth FD.

Client_Interview2 table

ClientNo	InterviewDate	InterViewTime	StaffNo
CR76	13-May-02	10.30	SG5
CR56	13-May-02	12.00	SG5
CR74	13-May-02	12.00	SG37
CR56	1-July-02	10.30	SG5

StaffRoom table

StaffNo	InterviewDate	RoomNo
SG5	13-May-02	G101
SG37	13-May-02	G102
SG5	1-July-02	G102

Normalization and Database Design

During the normalization process of database design, make sure that proposed entities meet required normal form before table structures are created. Many real-world databases have been improperly designed or burdened with anomalies if improperly modified during the course of time. You may be asked to redesign and modify existing databases. This can be a large undertaking if the tables are not properly normalized.

Key Terms and Abbrevations

Boyce-Codd normal form (BCNF): a special case of 3rd NF

first normal form (1NF): only single values are permitted at the intersection of each row and column so there are no repeating groups

normalization: the process of determining how much redundancy exists in a table

second normal form (2NF): the relation must be in 1NF and the PK comprises a single attribute

semantic rules: business rules applied to the database

third normal form (3NF): the relation must be in 2NF and all transitive dependencies must be removed; a non-key attribute may not be functionally dependent on another non-key attribute

Exercises

Complete chapters 11 and 12 before doing these exercises.

1. What is normalization?
2. When is a table in 1NF?
3. When is a table in 2NF?
4. When is a table in 3NF?
5. Identify and discuss each of the indicated dependencies in the dependency diagram shown in Figure 12.2.

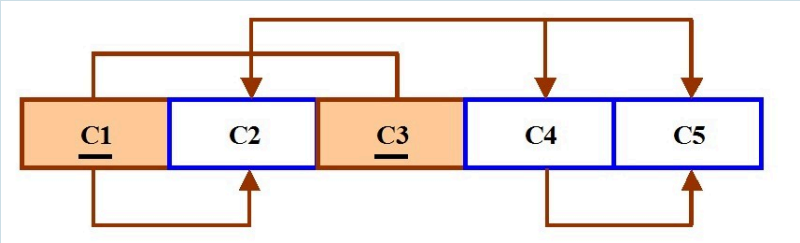


Figure 12.2 For question 5, by A. Watt.

6. To keep track of students and courses, a new college uses the table structure in Figure 12.3. Draw the dependency diagram for this table.

Attribute Name	Sample Value	Sample Value	Sample Value
StudentID	1	2	3
StudentName	John Smith	Sandy Law	Sue Rogers
CourseID	2	2	3
CourseName	Programming Level 1	Programming Level 1	Business
Grade	75%	61%	81%
CourseDate	Jan 5th, 2014	Jan 5th, 2014	Jan 7th, 2014

Figure 12.3 For question 6, by A. Watt.

7. Using the dependency diagram you just drew, show the tables (in their third normal form) you would create to fix the problems you encountered. Draw the dependency diagram for the fixed table.
8. An agency called Instant Cover supplies part-time/temporary staff to hotels in Scotland. Figure 12.4 lists the time spent by agency staff working at various hotels. The national insurance number (NIN) is unique for every member of staff. Use Figure 12.4 to answer questions (a) and (b).

NIN	ContractNo	Hours	eName	hNo	hLoc
1135	C1024	16	Smith J.	H25	East Killbride
1057	C1024	24	Hocine D.	H25	East Killbride
1068	C1025	28	White T.	H4	Glasgow
1135	C1025	15	Smith J.	H4	Glasgow

Figure 12.4 For question 8, by A. Watt.

   1. This table is susceptible to update anomalies. Provide examples of insertion, deletion and update anomalies.
   2. Normalize this table to third normal form. State any assumptions.

9. Fill in the blanks:

   1. _____ produces a lower normal form.
   2. Any attribute whose value determines other values within a row is called a(n) _____.
   3. An attribute that cannot be further divided is said to display _____.
   4. _____ refers to the level of detail represented by the values stored in a table's row.
   5. A relational table must not contain _____ groups.

Also see *Appendix B: Sample ERD Exercises*

Bibliography

Nguyen Kim Anh, *Relational Design Theory*. OpenStax CNX. 8 Jul 2009 Retrieved July 2014 from http://cnx.org/contents/606cc532-0b1d-419d-a0ec-ac4e2e2d533b@1@1

Russell, Gordon. Chapter 4 – Normalisation. *Database eLearning*. N.d. Retrived July 2014 from db.grussell.org/ch4.html

Chapter 13 Database Development Process

ADRIENNE WATT

A core aspect of software engineering is the subdivision of the development process into a series of phases, or steps, each of which focuses on one aspect of the development. The collection of these steps is sometimes referred to as the *software development life cycle (SDLC)*. The software product moves through this life cycle (sometimes repeatedly as it is refined or redeveloped) until it is finally retired from use. Ideally, each phase in the life cycle can be checked for correctness before moving on to the next phase.

Software Development Life Cycle – Waterfall

Let us start with an overview of the *waterfall model* such as you will find in most software engineering textbooks. This waterfall figure, seen in Figure 13.1, illustrates a general waterfall model that could apply to any computer system development. It shows the process as a strict sequence of steps where the output of one step is the input to the next and all of one step has to be completed before moving onto the next.

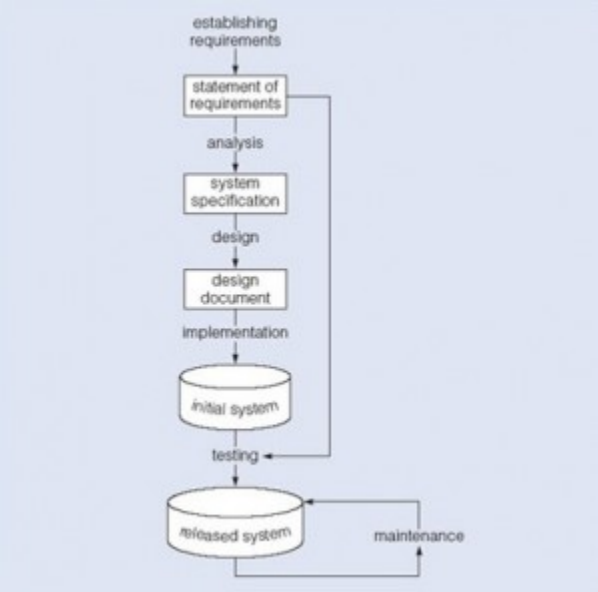


Figure 13.1. Waterfall model.

We can use the *waterfall process* as a means of identifying the tasks that are required, together with the input and output for each activity. What is important is the scope of the activities, which can be summarized as follows:

- *Establishing requirements* involves consultation with, and agreement among, stakeholders about what they want from a system, expressed as a statement of requirements.
- *Analysis* starts by considering the statement of requirements and finishes by producing a system specification. The specification is a formal representation of what a system should do, expressed in terms that are independent of how it may be realized.
- *Design* begins with a system specification, produces design documents and provides a detailed description of how

a system should be constructed.

- *Implementation* is the construction of a computer system according to a given design document and taking into account the environment in which the system will be operating (e.g., specific hardware or software available for the development). Implementation may be staged, usually with an initial system that can be validated and tested before a final system is released for use.
- *Testing* compares the implemented system against the design documents and requirements specification and produces an acceptance report or, more usually, a list of errors and bugs that require a review of the analysis, design and implementation processes to correct (testing is usually the task that leads to the waterfall model iterating through the life cycle).
- *Maintenance* involves dealing with changes in the requirements or the implementation environment, bug fixing or porting of the system to new environments (e.g., migrating a system from a standalone PC to a UNIX workstation or a networked environment). Since maintenance involves the analysis of the changes required, design of a solution, implementation and testing of that solution over the lifetime of a maintained software system, the waterfall life cycle will be repeatedly revisited.

Database Life Cycle

We can use the waterfall cycle as the basis for a model of database development that incorporates three assumptions:

1. We can separate the development of a database – that is, specification and creation of a schema to define data in a database – from the user processes that make use of the database.
2. We can use the three-schema architecture as a basis for distinguishing the activities associated with a schema.
3. We can represent the constraints to enforce the semantics of the data once within a database, rather than within every user process that uses the data.

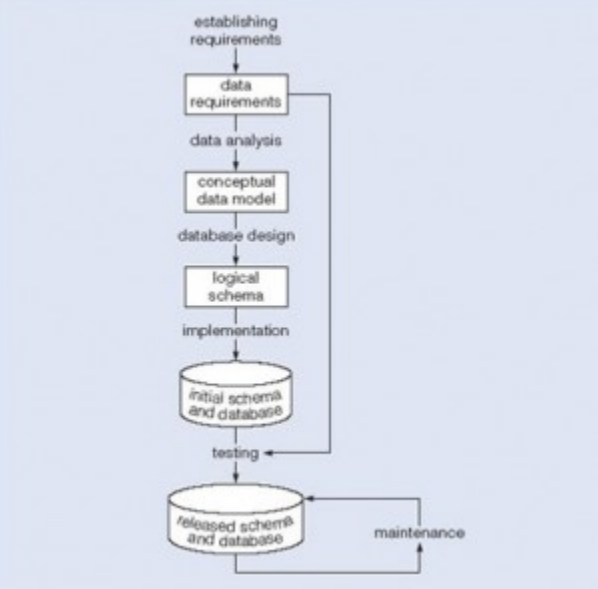


Figure 13.2. A waterfall model of the activities and their outputs for database development.

Using these assumptions and Figure 13.2, we can see that this diagram represents a model of the activities and their outputs for database development. It is applicable to any class of DBMS, not just a relational approach.

Database application development is the process of obtaining real-world requirements, analyzing requirements, designing the data and functions of the system, and then implementing the operations in the system.

Requirements Gathering

The first step is *requirements gathering*. During this step, the database designers have to interview the customers (database users) to understand the proposed system and obtain and document the data and functional requirements. The result of this step is a document that includes the detailed requirements provided by the users.

Establishing requirements involves consultation with, and agreement among, all the users as to what persistent data they want to store along with an agreement as to the meaning and interpretation of the data elements. The data administrator plays a key role in this process as they overview the business, legal and ethical issues within the organization that impact on the data requirements.

The *data requirements document* is used to confirm the understanding of requirements with users. To make sure that it is easily understood, it should not be overly formal or highly encoded. The document should give a concise summary of all users' requirements – not just a collection of individuals' requirements – as the intention is to develop a single shared database.

The requirements should not describe how the data is to be processed, but rather what the data items are, what attributes they have, what constraints apply and the relationships that hold between the data items.

Analysis

Data analysis begins with the statement of data requirements and then produces a conceptual data model. The aim of analysis is to obtain a detailed description of the data that will suit user requirements so that both high and low level properties of data and their use are dealt with. These include properties such as the possible range of values that can be permitted for attributes (e.g., in the school database example, the student course code, course title and credit points).

The conceptual data model provides a shared, formal representation of what is being communicated between clients and developers during database development – it is focused on the data in a database, irrespective of the eventual use of that data in user processes or implementation of the data in specific computer environments. Therefore, a conceptual data model is concerned with the meaning and structure of data, but not with the details affecting how they are implemented.

The conceptual data model then is a formal representation of what data a database should contain and the constraints the data must satisfy. This should be expressed in terms that are independent of how the model may be implemented. As a result, analysis focuses on the questions, "What is required?" not "How is it achieved?"

Logical Design

Database design starts with a conceptual data model and produces a specification of a logical schema; this will determine the specific type of database system (network, relational, object-oriented) that is required. The relational representation is still independent of any specific DBMS; it is another conceptual data model.

We can use a relational representation of the conceptual data model as input to the logical design process. The output of this stage is a detailed relational specification, the logical schema, of all the tables and constraints needed to satisfy the description of the data in the conceptual data model. It is during this design activity that choices are made as to which tables are most appropriate for representing the data in a database. These choices must take into account various design criteria including, for example, flexibility for change, control of duplication and how best to represent the constraints. It is the tables defined by the logical schema that determine what data are stored and how they may be manipulated in the database.

Database designers familiar with relational databases and SQL might be tempted to go directly to implementation after they have produced a conceptual data model. However, such a direct transformation of the relational representation to SQL tables does not necessarily result in a database that has all the desirable properties: completeness, integrity, flexibility, efficiency and usability. A good conceptual data model is an essential first step towards a database with these properties, but that does not mean that the direct transformation to SQL tables automatically produces a good database. This first step will accurately represent the tables and constraints needed to satisfy the conceptual data model description, and so will satisfy the completeness and integrity requirements, but it may be inflexible or offer poor usability. The first design is then flexed to improve the quality of the database design. *Flexing* is a term that is intended to capture the simultaneous ideas of bending something for a different purpose and weakening aspects of it as it is bent.

Figure 13.3 summarizes the iterative (repeated) steps involved in database design, based on the overview given. Its main purpose is to distinguish the general issue of what tables should be used from the detailed definition of the constituent parts of each table – these tables are considered one at a time, although they are not independent of each other. Each iteration that involves a revision of the tables would lead to a new design; collectively they are usually referred to as *second-cut designs*, even if the process iterates for more than a single loop.

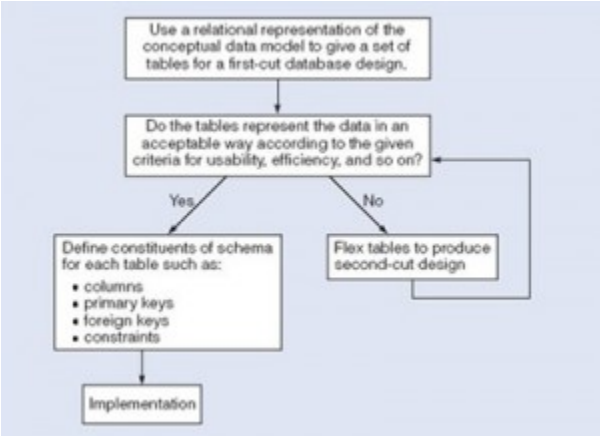


Figure 13.3. A summary of the iterative steps involved in database design.

First, for a given conceptual data model, it is not necessary that all the user requirements it represents be satisfied by a single database. There can be various reasons for the development of more than one database, such as the need for independent operation in different locations or departmental control over "their" data. However, if the collection of

databases contains duplicated data and users need to access data in more than one database, then there are possible reasons that one database can satisfy multiple requirements, or issues related to data replication and distribution need to be examined.

Second, one of the assumptions about database development is that we can separate the development of a database from the development of user processes that make use of it. This is based on the expectation that, once a database has been implemented, all data required by currently identified user processes have been defined and can be accessed; but we also require flexibility to allow us to meet future requirements changes. In developing a database for some applications, it may be possible to predict the common requests that will be presented to the database and so we can optimize our design for the most common requests.

Third, at a detailed level, many aspects of database design and implementation depend on the particular DBMS being used. If the choice of DBMS is fixed or made prior to the design task, that choice can be used to determine design criteria rather than waiting until implementation. That is, it is possible to incorporate design decisions for a specific DBMS rather than produce a generic design and then tailor it to the DBMS during implementation.

It is not uncommon to find that a single design cannot simultaneously satisfy all the properties of a good database. So it is important that the designer has prioritized these properties (usually using information from the requirements specification); for example, to decide if integrity is more important than efficiency and whether usability is more important than flexibility in a given development.

At the end of our design stage, the logical schema will be specified by SQL data definition language (DDL) statements, which describe the database that needs to be implemented to meet the user requirements.

Implementation

Implementation involves the construction of a database according to the specification of a logical schema. This will include the specification of an appropriate storage schema, security enforcement, external schema and so on. Implementation is heavily influenced by the choice of available DBMSs, database tools and operating environment. There are additional tasks beyond simply creating a database schema and implementing the constraints – data must be entered into the tables, issues relating to the users and user processes need to be addressed, and the management activities associated with wider aspects of corporate data management need to be supported. In keeping with the DBMS approach, we want as many of these concerns as possible to be addressed within the DBMS. We look at some of these concerns briefly now.

In practice, implementation of the logical schema in a given DBMS requires a very detailed knowledge of the specific features and facilities that the DBMS has to offer. In an ideal world, and in keeping with good software engineering practice, the first stage of implementation would involve matching the design requirements with the best available implementing tools and then using those tools for the implementation. In database terms, this might involve choosing vendor products with DBMS and SQL variants most suited to the database we need to implement. However, we don't live in an ideal world and more often than not, hardware choice and decisions regarding the DBMS will have been made well in advance of consideration of the database design. Consequently, implementation can involve additional flexing of the design to overcome any software or hardware limitations.

Realizing the Design

After the logical design has been created, we need our database to be created according to the definitions we have produced. For an implementation with a relational DBMS, this will probably involve the use of SQL to create tables and constraints that satisfy the logical schema description and the choice of appropriate storage schema (if the DBMS permits that level of control).

One way to achieve this is to write the appropriate SQL DDL statements into a file that can be executed by a DBMS so that there is an independent record, a text file, of the SQL statements defining the database. Another method is to work interactively using a database tool like SQL Server Management Studio or Microsoft Access. Whatever mechanism is used to implement the logical schema, the result is that a database, with tables and constraints, is defined but will contain no data for the user processes.

Populating the Database

After a database has been created, there are two ways of populating the tables – either from existing data or through the use of the user applications developed for the database.

For some tables, there may be existing data from another database or data files. For example, in establishing a database for a hospital, you would expect that there are already some records of all the staff that have to be included in the database. Data might also be brought in from an outside agency (address lists are frequently brought in from external companies) or produced during a large data entry task (converting hard-copy manual records into computer files can be done by a data entry agency). In such situations, the simplest approach to populate the database is to use the import and export facilities found in the DBMS.

Facilities to import and export data in various standard formats are usually available (these functions are also known in some systems as loading and unloading data). Importing enables a file of data to be copied directly into a table. When data are held in a file format that is not appropriate for using the import function, then it is necessary to prepare an application program that reads in the old data, transforms them as necessary and then inserts them into the database using SQL code specifically produced for that purpose. The transfer of large quantities of existing data into a database is referred to as a *bulk load*. Bulk loading of data may involve very large quantities of data being loaded, one table at a time so you may find that there are DBMS facilities to postpone constraint checking until the end of the bulk loading.

Guidelines for Developing an ER Diagram

Note: These are general guidelines that will assist in developing a strong basis for the actual database design (the logical model).

1. Document all entities discovered during the information-gathering stage.
2. Document all attributes that belong to each entity. Select candidate and primary keys. Ensure that all non-key attributes for each entity are full-functionally dependent on the primary key.
3. Develop an initial ER diagram and review it with appropriate personnel. (Remember that this is an iterative process.)
4. Create new entities (tables) for multivalued attributes and repeating groups. Incorporate these new entities (tables) in the ER diagram. Review with appropriate personnel.

5. Verify ER modeling by normalizing tables.

Key Terms

analysis: starts by considering the statement of requirements and finishes by producing a system specification**bulk load**: the transfer of large quantities of existing data into a database
data requirements document:used to confirm the understanding of requirements with the user

design: begins with a system specification, produces design documents and provides a detailed description of how a system should be constructed

establishing requirements: involves consultation with, and agreement among, stakeholders as to what they want from a system; expressed as a statement of requirements

flexing: a term that captures the simultaneous ideas of bending something for a different purpose and weakening aspects of it as it is bent

implementation: the construction of a computer system according to a given design document

maintenance: involves dealing with changes in the requirements or the implementation environment, bug fixing or porting of the system to new environments

requirements gathering: a process during which the database designer interviews the database user to understand the proposed system and obtain and document the data and functional requirements

second-cut designs:the collection of iterations that each involves a revision of the tables that lead to a new design

software development life cycle (SDLC): the series of steps involved in the database development process

testing: compares the implemented system against the design documents and requirements specification and produces an acceptance report

waterfall model: shows the database development process as a strict sequence of steps where the output of one step is the input to the next

waterfall process: a means of identifying the tasks required for database development, together with the input and output for each activity (see *waterfall model*)

Exercises

1. Describe the waterfall model. List the steps.
2. What does the acronym SDLC mean, and what does an SDLC portray?
3. What needs to be modified in the waterfall model to accommodate database design?
4. Provide the iterative steps involved in database design.

Attribution

The following material was written by Adrienne Watt:

1. Key Terms
2. Exercises

Chapter 14 Database Users

ADRIENNE WATT

End Users

End users are the people whose jobs require access to a database for querying, updating and generating reports.

Application user

The application user is someone who accesses an existing application program to perform daily tasks.

Sophisticated user

Sophisticated users are those who have their own way of accessing the database. This means they do not use the application program provided in the system. Instead, they might define their own application or describe their need directly by using query languages. These specialized users maintain their personal databases by using ready-made program packages that provide easy-to-use menu driven commands, such as MS Access.

Application Programmers

These users implement specific application programs to access the stored data. They must be familiar with the DBMSs to accomplish their task.

Database Administrators (DBA)

This may be one person or a group of people in an organization responsible for authorizing access to the database, monitoring its use and managing all of the resources to support the use of the entire database system.

Key Terms

application programmer: user who implements specific application programs to access the stored data
application user: accesses an existing application program to perform daily tasks.

database administrator (DBA): responsible for authorizing access to the database, monitoring its use and managing all the resources to support the use of the entire database system

end user: people whose jobs require access to a database for querying, updating and generating reports

sophisticated user: those who use other methods, other than the application program, to access the database

There are no exercises provided for this chapter.

Chapter 15 SQL Structured Query Language

ADRIENNE WATT & NELSON ENG

Structured Query Language (SQL) is a database language designed for managing data held in a relational database management system. SQL was initially developed by IBM in the early 1970s (Date 1986). The initial version, called SEQUEL (Structured English Query Language), was designed to manipulate and retrieve data stored in IBM's quasi-relational database management system, System R. Then in the late 1970s, Relational Software Inc., which is now Oracle Corporation, introduced the first commercially available implementation of SQL, Oracle V2 for VAX computers.

Many of the currently available relational DBMSs, such as Oracle Database, Microsoft SQL Server (shown in Figure 15.1), MySQL, IBM DB2, IBM Informix and Microsoft Access, use SQL.

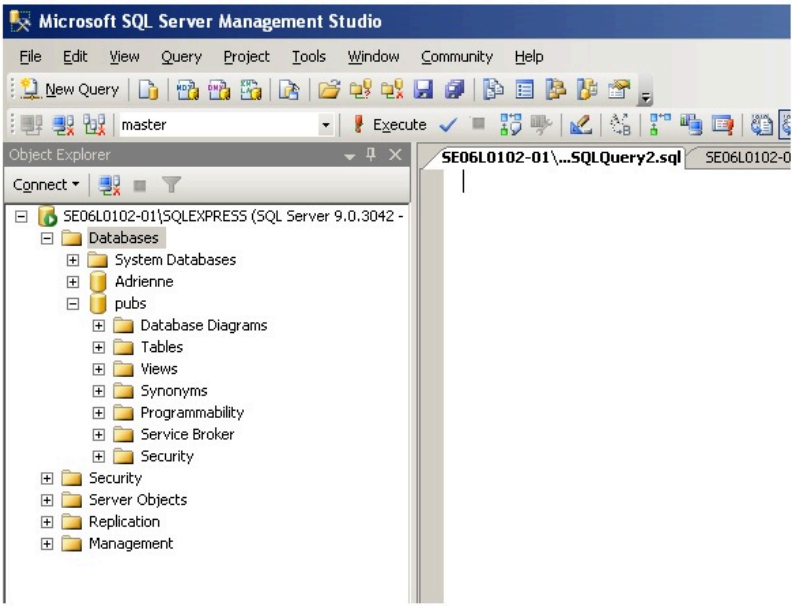


Figure 15.1. Example of Microsoft SQL Server, by A. Watt.

In a DBMS, the SQL database language is used to:

- Create the database and table structures
- Perform basic data management chores (add, delete and modify)
- Perform complex queries to transform raw data into useful information

In this chapter, we will focus on using SQL to create the database and table structures, mainly using SQL as a data definition language (DDL). In Chapter 16, we will use SQL as a data manipulation language (DML) to insert, delete, select and update data within the database tables.

Create Database

The major SQL DDL statements are CREATE DATABASE and CREATE/DROP/ALTER TABLE. The SQL statement CREATE is used to create the database and table structures.

Example: CREATE DATABASE SW

A new database named **SW** is created by the SQL statement CREATE DATABASE SW. Once the database is created, the next step is to create the database tables.

The general format for the CREATE TABLE command is:

```
CREATE TABLE <tablename>
(
ColumnName, Datatype, Optional Column Constraint,
ColumnName, Datatype, Optional Column Constraint,
Optional table Constraints
);
```

Tablename is the name of the database table such as **Employee**. Each field in the CREATE TABLE has three parts (see above):

1. ColumnName
2. Data type
3. Optional Column Constraint

ColumnName

The ColumnName must be unique within the table. Some examples of ColumnNames are FirstName and LastName.

Data Type

The data type, as described below, must be a system data type or a user-defined data type. Many of the data types have a size such as CHAR(35) or Numeric(8,2).

Bit –Integer data with either a 1 or 0 value

Int –Integer (whole number) data from -2^31 (-2,147,483,648) through 2^31 – 1 (2,147,483,647)

Smallint –Integer data from 2^15 (-32,768) through 2^15 – 1 (32,767)

Tinyint –Integer data from 0 through 255

Decimal –Fixed precision and scale numeric data from -10^38 -1 through 10^38

Numeric –A synonym for **decimal**

Timestamp –A database-wide unique number

Uniqueidentifier –A globally unique identifier (GUID)

Money – Monetary data values from -2^63 (-922,337,203,685,477.5808) through 2^63 – 1 (+922,337,203,685,477.5807), with accuracy to one-ten-thousandth of a monetary unit

Smallmoney –Monetary data values from -214,748.3648 through +214,748.3647, with accuracy to one-ten-thousandth of a monetary unit

Float –Floating precision number data from -1.79E + 308 through 1.79E + 308

Real –Floating precision number data from -3.40E + 38 through 3.40E + 38

Datetime –Date and time data from January 1, 1753, to December 31, 9999, with an accuracy of one-three-hundredths of a second, or 3.33 milliseconds

Smalldatetime –Date and time data from January 1, 1900, through June 6, 2079, with an accuracy of one minute

Char –Fixed-length non-Unicode character data with a maximum length of 8,000 characters

Varchar –Variable-length non-Unicode data with a maximum of 8,000 characters

Text –Variable-length non-Unicode data with a maximum length of 2^31 – 1 (2,147,483,647) characters

Binary –Fixed-length binary data with a maximum length of 8,000 bytes

Varbinary –Variable-length binary data with a maximum length of 8,000 bytes

Image – Variable-length binary data with a maximum length of 2^31 – 1 (2,147,483,647) bytes

Optional Column Constraints

The Optional ColumnConstraints are NULL, NOT NULL, UNIQUE, PRIMARY KEY and DEFAULT, used to initialize a value for a new record. The column constraint NULL indicates that null values are allowed, which means that a row can be created without a value for this column. The column constraint NOT NULL indicates that a value must be supplied when a new row is created.

To illustrate, we will use the SQL statement CREATE TABLE EMPLOYEES to create the employees table with 16 attributes or fields.

```
USE SW
CREATE TABLE EMPLOYEES
(
EmployeeNo              CHAR(10)        NOT NULL        UNIQUE,
DepartmentName          CHAR(30)        NOT NULL        DEFAULT “Human Resources”,
FirstName                CHAR(25)        NOT NULL,
```

```
LastName              CHAR(25)        NOT NULL,
Category              CHAR(20)        NOT NULL,
HourlyRate            CURRENCY        NOT NULL,
TimeCard              LOGICAL         NOT NULL,
HourlySalaried        CHAR(1)         NOT NULL,
EmpType               CHAR(1)         NOT NULL,
Terminated            LOGICAL         NOT NULL,
ExemptCode            CHAR(2)         NOT NULL,
Supervisor            LOGICAL         NOT NULL,
SupervisorName        CHAR(50)        NOT NULL,
BirthDate             DATE            NOT NULL,
CollegeDegree         CHAR(5)         NOT NULL,
CONSTRAINT            Employee_PK     PRIMARY KEY(EmployeeNo
);
```

The first field is EmployeeNo with a field type of CHAR. For this field, the field length is 10 characters, and the user cannot leave this field empty (NOT NULL).

Similarly, the second field is DepartmentName with a field type CHAR of length 30. After all the table columns are defined, a table constraint, identified by the word CONSTRAINT, is used to create the primary key:

```
CONSTRAINT    EmployeePK    PRIMARY KEY(EmployeeNo)
```

We will discuss the constraint property further later in this chapter.

Likewise, we can create a Department table, a Project table and an Assignment table using the CREATE TABLE SQL DDL command as shown in the below example.

```
USE SW
CREATE TABLE DEPARTMENT
(
DepartmentName Char(35)  NOT NULL,
BudgetCode     Char(30)  NOT NULL,
OfficeNumber   Char(15)  NOT NULL,
Phone          Char(15)  NOT NULL,
CONSTRAINT DEPARTMENT_PK PRIMARY KEY(DepartmentName)
);
```

In this example, a project table is created with seven fields: ProjectID, ProjectName, Department, MaxHours, StartDate, and EndDate.

```
USE SW
CREATE TABLE PROJECT
(
ProjectID       Int  NOT NULL IDENTITY (1000,100),
ProjectName     Char(50) NOT NULL,
Department      Char(35) NOT NULL,
MaxHours        Numeric(8,2)  NOT NULL DEFAULT 100,
StartDate       DateTime NULL,
EndDate         DateTime NULL,
CONSTRAINT      ASSIGNMENT_PK  PRIMARY KEY(ProjectID)
);
```

In this last example, an assignment table is created with three fields: ProjectID, EmployeeNumber, and HoursWorked. The assignment table is used to record who (EmployeeNumber) and how much time(HoursWorked) an employee worked on the particular project(ProjectID).

```
USE SW
CREATE TABLE ASSIGNMENT
(
ProjectID       Int  NOT NULL,
EmployeeNumber  Int  NOT NULL,
HoursWorked     Numeric(6,2)  NULL,
);
```

Table Constraints

Table constraints are identified by the CONSTRAINT keyword and can be used to implement various constraints described below.

IDENTITY constraint

We can use the optional column constraint IDENTITY to provide a unique, incremental value for that column. Identity columns are often used with the PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to a column with a tinyint, smallint, int, decimal or numeric data type. This constraint:

- Generates sequential numbers
- Does not enforce entity integrity

- Only one column can have the IDENTITY property
- Must be defined as an integer, numeric or decimal data type
- Cannot update a column with the IDENTITY property
- Cannot contain NULL values
- Cannot bind defaults and default constraints to the column

For IDENTITY[(seed, increment)]

- Seed – the initial value of the identity column
- Increment – the value to add to the last increment column

We will use another database example to further illustrate the SQL DDL statements by creating the table tblHotel in this HOTEL database.

```
CREATE TABLE  tblHotel
(
HotelNo          Int            IDENTITY (1,1),
Name             Char(50)       NOT NULL,
Address          Char(50)       NULL,
City             Char(25)       NULL,
)
```

UNIQUE constraint

The UNIQUE constraint prevents duplicate values from being entered into a column.

- Both PK and UNIQUE constraints are used to enforce entity integrity.
- Multiple UNIQUE constraints can be defined for a table.
- When a UNIQUE constraint is added to an existing table, the existing data is always validated.
- A UNIQUE constraint can be placed on columns that accept nulls. *Only one row can be NULL.*
- A UNIQUE constraint automatically creates a unique index on the selected column.

This is the general syntax for the UNIQUE constraint:

```
[CONSTRAINT constraint_name]
UNIQUE [CLUSTERED | NONCLUSTERED]
(col_name [, col_name2 [..., col_name16]])
[ON segment_name]
```

This is an examle using the UNIQUE constraint.

```
CREATE TABLE EMPLOYEES
(
EmployeeNo                CHAR(10)          NOT NULL          UNIQUE,
)
```

FOREIGN KEY constraint

The FOREIGN KEY (FK) constraint defines a column, or combination of columns, whose values match the PRIMARY KEY (PK) of another table.

- Values in an FK are automatically updated when the PK values in the associated table are updated/changed.
- FK constraints must reference PK or the UNIQUE constraint of another table.
- The number of columns for FK must be same as PK or UNIQUE constraint.
- If the WITH NOCHECK option is used, the FK constraint will not validate existing data in a table.
- No index is created on the columns that participate in an FK constraint.

This is the general syntax for the FOREIGN KEY constraint:

```
[CONSTRAINT constraint_name]
[FOREIGN KEY (col_name [, col_name2 [..., col_name16]])]
REFERENCES [owner.]ref_table [(ref_col [, ref_col2 [..., ref_col16]])]
```

In this example, the field HotelNo in the tblRoom table is a FK to the field HotelNo in the tblHotel table shown previously.

```
USE HOTEL
GO
CREATE TABLE  tblRoom
(
HotelNo          Int                NOT NULL ,
RoomNo  Int                NOT NULL,
Type               Char(50)         NULL,
Price              Money            NULL,
PRIMARY KEY (HotelNo, RoomNo),
FOREIGN KEY (HotelNo) REFERENCES tblHotel
)
```

CHECK constraint

The CHECK constraint restricts values that can be entered into a table.

- It can contain search conditions similar to a WHERE clause.
- It can reference columns in the same table.
- The data validation rule for a CHECK constraint must evaluate to a boolean expression.
- It can be defined for a column that has a rule bound to it.

This is the general syntax for the CHECK constraint:

```
[CONSTRAINT constraint_name]
CHECK [NOT FOR REPLICATION] (expression)
```

In this example, the Type field is restricted to have only the types 'Single', 'Double', 'Suite' or 'Executive'.

```
USE HOTEL
GO
CREATE TABLE  tblRoom
(
HotelNo          Int                  NOT NULL,
RoomNo  Int                  NOT NULL,
Type                Char(50)         NULL,
Price       Money                     NULL,
PRIMARY KEY (HotelNo, RoomNo),
FOREIGN KEY (HotelNo) REFERENCES tblHotel
CONSTRAINT Valid_Type
CHECK (Type IN ('Single', 'Double', 'Suite', 'Executive'))
)
```

In this second example, the employee hire date should be before January 1, 2004, or have a salary limit of $300,000.

```
GO
CREATE TABLE SALESREPS
(
Empl_num     Int Not Null
CHECK (Empl_num BETWEEN 101 and 199),
Name             Char (15),
Age      Int       CHECK (Age >= 21),
```

```
Quota                Money                CHECK (Quota >= 0.0),
HireDate      DateTime,
CONSTRAINT  QuotaCap CHECK ((HireDate < “01-01-2004”) OR (Quota <=300000))
)
```

DEFAULT constraint

The DEFAULT constraint is used to supply a value that is automatically added for a column if the user does not supply one.

- A column can have only one DEFAULT.
- The DEFAULT constraint cannot be used on columns with a timestamp data type or identity property.
- DEFAULT constraints are automatically bound to a column when they are created.

The general syntax for the DEFAULT constraint is:

```
[CONSTRAINT constraint_name]
DEFAULT {constant_expression | niladic-function | NULL}
[FOR col_name]
```

This example sets the default for the city field to ‘Vancouver’.

```
USE HOTEL
ALTER TABLE tblHotel
Add CONSTRAINT df_city DEFAULT ‘Vancouver’ FOR City
```

User Defined Types

User defined types are always based on system-supplied data type. They can enforce data integrity and they allow nulls.

To create a user-defined data type in SQL Server, choose types under “Programmability” in your database. Next, right click and choose ‘New’ –>‘User-defined data type’ or execute the sp_addtype system stored procedure. After this, type:

```
sp_addtype ssn, 'varchar(11)', 'NOT NULL'
```

This will add a new user-defined data type called SIN with nine characters.

In this example, the field EmployeeSIN uses the user-defined data type SIN.

```
CREATE TABLE SINTable
(
EmployeeID       INT Primary Key,
EmployeeSIN    SIN,
CONSTRAINT CheckSIN
CHECK (EmployeeSIN LIKE
' [0-9][0-9][0-9] – [0-9][0-9] [0-9] – [0-9][0-9][0-9] ')
)
```

ALTER TABLE

You can use ALTER TABLE statements to add and drop constraints.

- ALTER TABLE allows columns to be removed.
- When a constraint is added, all existing data are verified for violations.

In this example, we use the ALTER TABLE statement to the IDENTITY property to a ColumnName field.

```
USE HOTEL
GO
ALTER TABLE  tblHotel
ADD CONSTRAINT unqName UNIQUE (Name)
```

Use the ALTER TABLE statement to add a column with the IDENTITY property such as ALTER TABLE TableName.

```
ADD
ColumnName        int    IDENTITY(seed, increment)
```

DROP TABLE

The DROP TABLE will remove a table from the database. Make sure you have the correct database selected.

```
DROP TABLE tblHotel
```

Executing the above SQL DROP TABLE statement will remove the table tblHotel from the database.

Key Terms

DDL: abbreviation for *data definition language*
DML: abbreviation for *data manipulation language*

SEQUEL: acronym for *Structured English Query Language*; designed to manipulate and retrieve data stored in IBM's quasi-relational database management system, System R

Structured Query Language (SQL): a database language designed for managing data held in a relational database management system

Exercises

1. Using the information for the Chapter 9 exercise, implement the schema using Transact SQL (show SQL statements for each table). Implement the constraints as well.
2. Create the table shown here in SQL Server and show the statements you used.
   Table: Employee

ATTRIBUTE (FIELD) NAME	DATA DECLARATION
EMP_NUM	CHAR(3)
EMP_LNAME	VARCHAR(15)
EMP_FNAME	VARCHAR(15)
EMP_INITIAL	CHAR(1)
EMP_HIREDATE	DATE
JOB_CODE	CHAR(3)

3. Having created the table structure in question 2, write the SQL code to enter the rows for the table shown in Figure 15.1.

	EMP_NUM	EMP_LNAME	EMP_FNAME	EMP_INITIAL	EMP_HIREDATE	JOB_CODE
▶	101	News	John	G	08-Nov-00	502
	102	Senior	David	H	12-Jul-89	501
	103	Arbough	June	E	01-Dec-96	500
	104	Ramoras	Anne	K	15-Nov-87	501
	105	Johnson	Alice	K	01-Feb-93	502
	106	Smithfield	William		22-Jun-04	500
	107	Alonzo	Maria	D	10-Oct-93	500
	108	Washington	Ralph	B	22-Aug-91	501
	109	Smith	Larry	W	18-Jul-97	501

Figure 15.2. Employee table with data for questions 4-10, by A. Watt.

Use Figure 15.2 to answer questions 4 to 10.

4. Write the SQL code to change the job code to 501 for the person whose personnel number is 107. After you have completed the task, examine the results, and then reset the job code to its original value.
5. Assuming that the data shown in the Employee table have been entered, write the SQL code that lists all attributes for a job code of 502.
6. Write the SQL code to delete the row for the person named William Smithfield, who was hired on June 22, 2004, and whose job code classification is 500. (*Hint*: Use logical operators to include all the information given in this problem.)
7. Add the attributes EMP_PCT and PROJ_NUM to the Employee table. The EMP_PCT is the bonus percentage to be paid to each employee.
8. Using a single command, write the SQL code that will enter the project number (PROJ_NUM) = 18 for all employees whose job classification (JOB_CODE) is 500.
9. Using a single command, write the SQL code that will enter the project number (PROJ_NUM) = 25 for all employees whose job classification (JOB_CODE) is 502 or higher.
10. Write the SQL code that will change the PROJ_NUM to 14 for those employees who were hired before January 1, 1994, and whose job code is at least 501. (You may assume that the table will be restored to its original condition preceding this question.)

Also see *Appendix C: SQL Lab with Solution*

References

Date, C.J. *Relational Database Selected Writings.* Reading: Mass: Addison-Wesley Publishing Company Inc., 1986, p. 269-311.

Chapter 16 SQL Data Manipulation Language

ADRIENNE WATT & NELSON ENG

The SQL data manipulation language (DML) is used to query and modify database data. In this chapter, we will describe how to use the SELECT, INSERT, UPDATE, and DELETE SQL DML command statements, defined below.

- *SELECT* – to query data in the database
- *INSERT* – to insert data into a table
- *UPDATE* – to update data in a table
- *DELETE* – to delete data from a table

In the SQL DML statement:

- Each clause in a statement should begin on a new line.
- The beginning of each clause should line up with the beginning of other clauses.
- If a clause has several parts, they should appear on separate lines and be indented under the start of the clause to show the relationship.
- Upper case letters are used to represent reserved words.
- Lower case letters are used to represent user-defined words.

SELECT Statement

The SELECT statement, or command, allows the user to extract data from tables, based on specific criteria. It is processed according to the following sequence:

```
SELECT DISTINCT item(s)
FROM  table(s)
WHERE  predicate
GROUP BY  field(s)
ORDER BY fields
```

We can use the SELECT statement to generate an employee phone list from the Employees table as follows:

```
SELECT  FirstName, LastName, phone
FROM Employees
ORDER BY LastName
```

This action will display employee's last name, first name, and phone number from the Employees table, seen in Table 16.1.

Last Name	First Name	Phone Number
Hagans	Jim	604-232-3232
Wong	Bruce	604-244-2322

Table 16.1. Employees table.

In this next example, we will use a Publishers table (Table 16.2). (You will notice that Canada is mispelled in the *Publisher Country* field for Example Publishing and ABC Publishing. To correct mispelling, use the UPDATE statement to standardize the country field to Canada – see UPDATE statement later in this chapter.)

Publisher Name	Publisher City	Publisher Province	Publisher Country
Acme Publishing	Vancouver	BC	Canada
Example Publishing	Edmonton	AB	Cnada
ABC Publishing	Toronto	ON	Canda

Table 16.2. Publishers table.

If you add the publisher's name and city, you would use the SELECT statement followed by the fields name separated by a comma:

```
SELECT PubName, city
FROM Publishers
```

This action will display the publisher's name and city from the Publishers table.

If you just want the publisher's name under the display name city, you would use the SELECT statement with *no comma* separating pub_name and city:

```
SELECT PubName city
FROM Publishers
```

Performing this action will display only the pub_name from the Publishers table with a "city" heading. If you do not include the comma, SQL Server assumes you want a new column name for pub_name.

SELECT statement with WHERE criteria

Sometimes you might want to focus on a portion of the Publishers table, such as only publishers that are in Vancouver. In this situation, you would use the SELECT statement with the WHERE criterion, i.e., WHERE city = 'Vancouver'.

These first two examples illustrate how to limit record selection with the WHERE criterion using BETWEEN. Each of these examples give the same results for store items with between 20 and 50 items in stock.

Example #1 uses the quantity, *qty BETWEEN 20 and* 50.

```
SELECT StorID, qty, TitleID
FROM Sales
WHERE qty BETWEEN 20 and 50  (includes the 20 and 50)
```

Example #2, on the other hand, uses *qty* >=20 *and qty* <=50 .

```
SELECT StorID, qty, TitleID
FROM Sales
WHERE qty >= 20 and qty  <= 50
```

Example #3 illustrates how to limit record selection with the WHERE criterion using NOT BETWEEN.

```
SELECT StorID, qty, TitleID
FROM Sales
WHERE qty NOT BETWEEN 20 and 50
```

The next two examples show two different ways to limit record selection with the WHERE criterion using IN, with each yielding the same results.

Example #4 shows how to select records using *province*= as part of the WHERE statement.

```
SELECT *
FROM Publishers
WHERE province = 'BC' OR province = 'AB' OR province = 'ON'
```

Example #5 select records using *province IN* as part of the WHERE statement.

```
SELECT *
FROM Publishers
WHERE province IN ('BC', 'AB', 'ON')
```

The final two examples illustrate how NULL and NOT NULL can be used to select records. For these examples, a

Books table (not shown) would be used that contains fields called Title, Quantity, and Price (of book). Each publisher has a Books table that lists all of its books.

Example #6 uses NULL.

```
SELECT price, title
FROM Books
WHERE price IS NULL
```

Example #7 uses NOT NULL.

```
SELECT price, title
FROM Books
WHERE price IS NOT NULL
```

Using wildcards in the LIKE clause

The LIKE keyword selects rows containing fields that match specified portions of character strings. LIKE is used with char, varchar, text, datetime and smalldatetime data. A *wildcard* allows the user to match fields that contain certain letters. For example, the wildcard province = 'N%' would give all provinces that start with the letter 'N'. Table 16.3 shows four ways to specify wildcards in the SELECT statement in regular express format.

%	Any string of zero or more characters
_	Any single character
[ ]	Any single character within the specified range (e.g., [a-f]) or set (e.g., [abcdef])
[^]	Any single character not within the specified range (e.g., [^a – f]) or set (e.g., [^abcdef])

Table 16.3. How to specify wildcards in the SELECT statement.

In example #1, LIKE 'Mc%' searches for all last names that begin with the letters "Mc" (e.g., McBadden).

```
SELECT LastName
FROM Employees
WHERE LastName LIKE 'Mc%'
```

For example #2: LIKE '%inger' searches for all last names that end with the letters "inger" (e.g., Ringer, Stringer).

```
SELECT LastName
FROM Employees
WHERE LastName LIKE '%inger'
```

In, example #3: LIKE '%en%' searches for all last names that have the letters "en" (e.g., Bennett, Green, McBadden).

```
SELECT LastName
FROM Employees
WHERE LastName LIKE '%en%'
```

SELECT statement with ORDER BY clause

You use the ORDER BY clause to sort the records in the resulting list. Use ASC to sort the results in ascending order and *DESC* to sort the results in descending order.

For example, with ASC:

```
SELECT *
FROM Employees
ORDER BY HireDate ASC
```

And with DESC:

```
SELECT *
FROM Books
ORDER BY type, price DESC
```

SELECT statement with GROUP BY clause

The *GROUP BY* clause is used to create one output row per each group and produces summary values for the selected columns, as shown below.

```
SELECT type
FROM Books
GROUP BY type
```

Here is an example using the above statement.

```
SELECT type AS 'Type', MIN(price) AS 'Minimum Price'
FROM Books
WHERE royalty > 10
GROUP BY type
```

If the SELECT statement includes a WHERE criterion where *price is not null*,

```
SELECT type, price
FROM Books
WHERE price is not null
```

then a statement with the GROUP BY clause would look like this:

```
SELECT type AS 'Type', MIN(price) AS 'Minimum Price'
FROM Books
WHERE price is not null
GROUP BY type
```

Using COUNT with GROUP BY

We can use COUNT to tally how many items are in a container. However, if we want to count different items into separate groups, such as marbles of varying colours, then we would use the COUNT function with the GROUP BY command.

The below SELECT statement illustrates how to count groups of data using the COUNT function with the GROUP BY clause.

```
SELECT COUNT(*)
FROM Books
GROUP BY type
```

Using AVG and SUM with GROUP BY

We can use the AVG function to give us the average of any group, and SUM to give the total.

Example #1 uses the AVG FUNCTION with the GROUP BY type.

```
SELECT AVG(qty)
FROM Books
GROUP BY type
```

Example #2 uses the SUM function with the GROUP BY type.

```
SELECT SUM(qty)
FROM Books
GROUP BY type
```

Example #3 uses both the AVG and SUM functions with the GROUP BY type in the SELECT statement.

```
SELECT 'Total Sales' = SUM(qty), 'Average Sales' = AVG(qty), stor_id
FROM Sales
GROUP BY StorID ORDER BY  'Total Sales'
```

Restricting rows with HAVING

The HAVING clause can be used to restrict rows. It is similar to the WHERE condition except HAVING can include the aggregate function; the WHERE cannot do this.

The HAVING clause behaves like the WHERE clause, but is applicable to groups. In this example, we use the HAVING clause to exclude the groups with the province 'BC'.

```
SELECT au_fname AS 'Author"s First Name', province as 'Province'
FROM Authors
GROUP BY au_fname, province
HAVING province <> 'BC'
```

INSERT statement

The *INSERT statement* adds rows to a table. In addition,

- INSERT specifies the table or view that data will be inserted into.
- Column_list lists columns that will be affected by the INSERT.
- If a column is omitted, each value must be provided.
- If you are including columns, they can be listed in any order.
- VALUES specifies the data that you want to insert into the table. VALUES is required.
- Columns with the IDENTITY property should not be explicitly listed in the column_list or values_clause.

The syntax for the INSERT statement is:

```
INSERT [INTO] Table_name | view name [column_list]
DEFAULT VALUES | values_list | select statement
```

When inserting rows with the INSERT statement, these rules apply:

- Inserting an empty string (' ') into a varchar or text column inserts a single space.
- All char columns are right-padded to the defined length.
- All trailing spaces are removed from data inserted into varchar columns, except in strings that contain only spaces. These strings are truncated to a single space.
- If an INSERT statement violates a constraint, default or rule, or if it is the wrong data type, the statement fails and SQL Server displays an error message.

When you specify values for only some of the columns in the column_list, one of three things can happen to the columns that have no values:

1. A default value is entered if the column has a DEFAULT constraint, if a default is bound to the column, or if a default is bound to the underlying user-defined data type.
2. NULL is entered if the column allows NULLs and no default value exists for the column.
3. An error message is displayed and the row is rejected if the column is defined as NOT NULL and no default exists.

This example uses INSERT to add a record to the publisher's Authors table.

```
INSERT INTO Authors
VALUES('555-093-467', 'Martin', 'April', '281 555-5673', '816 Market St.,' , 'Vancouver', 'BC', 'V7G3P4', 0)
```

This following example illustrates how to insert a partial row into the Publishers table with a column list. The country column had a default value of Canada so it does not require that you include it in your values.

```
INSERT INTO Publishers (PubID, PubName, city, province)
VALUES ('9900', 'Acme Publishing', 'Vancouver', 'BC')
```

To insert rows into a table with an IDENTITY column, follow the below example. Do not supply the value for the IDENTITY nor the name of the column in the column list.

```
INSERT INTO jobs
VALUES ('DBA', 100, 175)
```

Inserting specific values into an IDENTITY column

By default, data cannot be inserted directly into an IDENTITY column; however, if a row is accidentally deleted, or there are gaps in the IDENTITY column values, you can insert a row and specify the IDENTITY column value.

```
IDENTITY_INSERT option
```

To allow an insert with a specific identity value, the IDENTITY_INSERT option can be used as follows.

```
SET IDENTITY_INSERT jobs ON
INSERT INTO jobs  (job_id, job_desc, min_lvl, max_lvl)
VALUES (19, 'DBA2', 100, 175)
SET IDENTITY_INSERT jobs OFF
```

Inserting rows with a SELECT statement

We can sometimes create a small temporary table from a large table. For this, we can insert rows with a SELECT statement. When using this command, there is no validation for uniqueness. Consequently, there may be many rows with the same pub_id in the example below.

This example creates a smaller temporary Publishers table using the CREATE TABLE statement. Then the INSERT with a SELECT statement is used to add records to this temporary Publishers table from the publis table.

```
CREATE TABLE dbo.tmpPublishers (
PubID char (4) NOT NULL ,
PubName varchar (40) NULL ,
city varchar (20) NULL ,
province char (2) NULL ,
country varchar (30) NULL  DEFAULT ('Canada')
)
INSERT  tmpPublishers
SELECT * FROM Publishers
```

In this example, we're copying a subset of data.

```
INSERT tmpPublishers (pub_id, pub_name)
SELECT PubID, PubName
FROM Publishers
```

In this example, the publishers' data are copied to the tmpPublishers table and the country column is set to Canada.

```
INSERT tmpPublishers (PubID, PubName, city, province, country)
SELECT PubID, PubName, city, province, 'Canada'
FROM Publishers
```

UPDATE statement

The *UPDATE statement* changes data in existing rows either by adding new data or modifying existing data.

This example uses the UPDATE statement to standardize the country field to be Canada for all records in the Publishers table.

```
UPDATE Publishers
SET country = 'Canada'
```

This example increases the royalty amount by 10% for those royalty amounts between 10 and 20.

```
UPDATE roysched
SET royalty = royalty + (royalty * .10)
WHERE royalty BETWEEN 10 and 20
```

Including subqueries in an UPDATE statement

The employees from the Employees table who were hired by the publisher in 2010 are given a promotion to the highest job level for their job type. This is what the UPDATE statement would look like.

```
UPDATE Employees
SET job_lvl =
(SELECT max_lvl FROM jobs
WHERE employee.job_id = jobs.job_id)
WHERE DATEPART(year, employee.hire_date) = 2010
```

DELETE statement

The *DELETE statement* removes rows from a record set. DELETE names the table or view that holds the rows that will be deleted and only one table or row may be listed at a time. WHERE is a standard WHERE clause that limits the deletion to select records.

The DELETE syntax looks like this.

```
DELETE [FROM] {table_name | view_name }
[WHERE clause]
```

The rules for the DELETE statement are:

1. If you omit a WHERE clause, all rows in the table are removed (except for indexes, the table, constraints).
2. DELETE cannot be used with a view that has a FROM clause naming more than one table. (Delete can affect only one base table at a time.)

What follows are three different DELETE statements that can be used.

1. Deleting all rows from a table.

```
DELETE
FROM Discounts
```

2. Deleting selected rows:

```
DELETE
FROM Sales
WHERE stor_id = '6380'
```

3. Deleting rows based on a value in a subquery:

```
DELETE FROM Sales
WHERE title_id IN
(SELECT title_id FROM Books WHERE type = 'mod_cook')
```

Built-in Functions

There are many built-in functions in SQL Server such as:

1. *Aggregate*: returns summary values
2. *Conversion:* transforms one data type to another
3. *Date*: displays information about dates and times
4. *Mathematical*: performs operations on numeric data
5. *String*: performs operations on character strings, binary data or expressions
6. *System*: returns a special piece of information from the database
7. *Text and image*: performs operations on text and image data

Below you will find detailed descriptions and examples for the first four functions.

Aggregate functions

Aggregate functions perform a calculation on a set of values and return a single, or summary, value. Table 16.4 lists these functions.

FUNCTION	DESCRIPTION
AVG	Returns the average of all the values, or only the DISTINCT values, in the expression.
COUNT	Returns the number of non-null values in the expression. When DISTINCT is specified, COUNT finds the number of unique non-null values.
COUNT(*)	Returns the number of rows. COUNT(*) takes no parameters and cannot be used with DISTINCT.
MAX	Returns the maximum value in the expression. MAX can be used with numeric, character and datetime columns, but not with bit columns. With character columns, MAX finds the highest value in the collating sequence. MAX ignores any null values.
MIN	Returns the minimum value in the expression. MIN can be used with numeric, character and datetime columns, but not with bit columns. With character columns, MIN finds the value that is lowest in the sort sequence. MIN ignores any null values.
SUM	Returns the sum of all the values, or only the DISTINCT values, in the expression. SUM can be used with numeric columns only.

Table 16.4 A list of aggregate functions and descriptions.

Below are examples of each of the aggregate functions listed in Table 16.4.

Example #1: AVG

```
SELECT AVG (price) AS 'Average Title Price'
FROM Books
```

Example #2: COUNT

```
SELECT COUNT(PubID) AS 'Number of Publishers'
FROM Publishers
```

Example #3: COUNT

```
SELECT COUNT(province) AS 'Number of Publishers'
FROM Publishers
```

Example #3: COUNT (*)

```
SELECT COUNT(*)
FROM Employees
WHERE job_lvl = 35
```

Example #4: MAX

```
SELECT MAX (HireDate)
FROM Employees
```

Example #5: MIN

```
SELECT MIN (price)
FROM Books
```

Example #6: SUM

```
SELECT SUM(discount) AS 'Total Discounts'
FROM Discounts
```

Conversion function

The conversion function transforms one data type to another.

In the example below, a price that contains two 9s is converted into five characters. The syntax for this statement is SELECT 'The date is ' + CONVERT(varchar(12), getdate()).

```
SELECT CONVERT(int, 10.6496)
SELECT title_id, price
FROM Books
WHERE CONVERT(char(5), price) LIKE '%99%'
```

In this second example, the conversion function changes data to a data type with a different size.

```
SELECT title_id, CONVERT(char(4), ytd_sales) as 'Sales'
FROM Books
WHERE type LIKE '%cook'
```

Date function

The date function produces a date by adding an interval to a specified date. The result is a datetime value equal to the date plus the number of date parts. If the date parameter is a smalldatetime value, the result is also a smalldatetime value.

The DATEADD function is used to add and increment date values. The syntax for this function is DATEADD(datepart, number, date).

```
SELECT DATEADD(day, 3, hire_date)
FROM Employees
```

In this example, the function DATEDIFF(datepart, date1, date2) is used.

This command returns the number of datepart "boundaries" crossed between two specified dates. The method of counting crossed boundaries makes the result given by DATEDIFF consistent across all data types such as minutes, seconds, and milliseconds.

```
SELECT DATEDIFF(day, HireDate, 'Nov 30 1995')
FROM Employees
```

For any particular date, we can examine any part of that date from the year to the millisecond.

The date parts (DATEPART) and abbreviations recognized by SQL Server, and the acceptable values are listed in Table 16.5.

DATE PART	ABBREVIATION	VALUES
Year	yy	1753-9999
Quarter	qq	1-4
Month	mm	1-12
Day of year	dy	1-366
Day	dd	1-31
Week	wk	1-53
Weekday	dw	1-7 (Sun.-Sat.)
Hour	hh	0-23
Minute	mi	0-59
Second	ss	0-59
Millisecond	ms	0-999

Table 16.5. Date part abbreviations and values.

Mathematical functions

Mathematical functions perform operations on numeric data. The following example lists the current price for each book sold by the publisher and what they would be if all prices increased by 10%.

```
SELECT Price, (price * 1.1) AS 'New Price', title
FROM Books
SELECT 'Square Root' = SQRT(81)
SELECT 'Rounded' = ROUND(4567.9876,2)
SELECT FLOOR (123.45)
```

Joining Tables

Joining two or more tables is the process of comparing the data in specified columns and using the comparison results to form a new table from the rows that qualify. A join statement:

- Specifies a column from each table
- Compares the values in those columns row by row
- Combines rows with qualifying values into a new row

Although the comparison is usually for equality – values that match exactly – other types of joins can also be specified. All the different joins such as inner, left (outer), right (outer), and cross join will be described below.

Inner join

An *inner join* connects two tables on a column with the same data type. Only the rows where the column values match are returned; unmatched rows are discarded.

Example #1

```
SELECT jobs.job_id, job_desc
FROM jobs
INNER JOIN Employees ON employee.job_id = jobs.job_id
WHERE jobs.job_id < 7
```

Example #2

```
SELECT authors.au_fname, authors.au_lname, books.royalty, title
FROM authorsINNER JOIN titleauthor ON authors.au_id=titleauthor.au_id
INNER JOIN books ON titleauthor.title_id=books.title_id
GROUP BY authors.au_lname, authors.au_fname, title, title.royalty
ORDER BY authors.au_lname
```

Left outer join

A *left outer join* specifies that all left outer rows be returned. All rows from the left table that did not meet the condition specified are included in the results set, and output columns from the other table are set to NULL.

This first example uses the new syntax for a left outer join.

```
SELECT publishers.pub_name, books.title
FROM Publishers
LEFT OUTER JOIN Books On publishers.pub_id = books.pub_id
```

This is an example of a left outer join using the old syntax.

```
SELECT publishers.pub_name, books.title
FROM Publishers, Books
```

```
WHERE publishers.pub_id *= books.pub_id
```

Right outer join

A *right outer join* includes, in its result set, all rows from the right table that did not meet the condition specified. Output columns that correspond to the other table are set to NULL.

Below is an example using the new syntax for a right outer join.

```
SELECT titleauthor.title_id, authors.au_lname, authors.au_fname
FROM titleauthor
RIGHT OUTER JOIN authors ON titleauthor.au_id = authors.au_id
ORDERY BY au_lname
```

This second example show the old syntax used for a right outer join.

```
SELECT titleauthor.title_id, authors.au_lname, authors.au_fname
FROM titleauthor, authors
WHERE titleauthor.au_id =* authors.au_id
ORDERY BY au_lname
```

Full outer join

A *full outer join* specifies that if a row from either table does not match the selection criteria, the row is included in the result set, and its output columns that correspond to the other table are set to NULL.

Here is an example of a full outer join.

```
SELECT books.title, publishers.pub_name, publishers.province
FROM Publishers
FULL OUTER JOIN Books ON books.pub_id = publishers.pub_id
WHERE (publishers.province <> “BC” and publishers.province <> “ON”)
ORDER BY books.title_id
```

Cross join

A *cross join* is a product combining two tables. This join returns the same rows as if no WHERE clause were specified. For example:

```
SELECT au_lname, pub_name,
FROM Authors CROSS JOIN Publishers
```

Key Terms

aggregate function: returns summary values**ASC**: ascending order
conversion function: transforms one data type to another

cross join: a product combining two tables

date function: displays information about dates and times

DELETE statement: removes rows from a record set

DESC: descending order

full outer join: specifies that if a row from either table does not match the selection criteria

GROUP BY: used to create one output row per each group and produces summary values for the selected columns

inner join: connects two tables on a column with the same data type

INSERT statement: adds rows to a table

left outer join: specifies that all left outer rows be returned

mathematical function: performs operations on numeric data

right outer join: includes all rows from the right table that did not meet the condition specified

SELECT statement: used to query data in the database

string function: performs operations on character strings, binary data or expressions

system function: returns a special piece of information from the database

text and image functions: performs operations on text and image data

UPDATE statement: changes data in existing rows either by adding new data or modifying existing data

wildcard: allows the user to match fields that contain certain letters.

Exercises

For questions 1 to 18 use the PUBS sample database created by Microsoft. To download the script to generate this database please go to the following site: http://www.microsoft.com/en-ca/download/details.aspx?id=23654.

1. Display a list of publication dates and titles (books) that were published in 2011.
2. Display a list of titles that have been categorized as either traditional or modern cooking. Use the Books table.
3. Display all authors whose first names are five letters long.
4. Display from the Books table: type, price, pub_id, title about the books put out by each publisher. Rename the column type with "Book Category." Sort by type (descending) and then price (ascending).
5. Display title_id, pubdate and pubdate plus three days, using the Books table.
6. Using the datediff and getdate function determine how much time has elapsed in months since the books in the Books table were published.
7. List the title IDs and quantity of all books that sold more than 30 copies.
8. Display a list of all last names of the authors who live in Ontario (ON) and the cities where they live.
9. Display all rows that contain a 60 in the payterms field. Use the Sales table.
10. Display all authors whose first names are five letters long , end in O or A, and start with M or P.
11. Display all titles that cost more than $30 and either begin with T or have a publisher ID of 0877.
12. Display from the Employees table the first name (fname), last name (lname), employe ID(emp_id) and job level (job_lvl) columns for those employees with a job level greater than 200; and rename the column headings to: "First Name," "Last Name," "IDENTIFICATION#" and "Job Level."
13. Display the royalty, royalty plus 50% as "royalty plus 50" and title_id. Use the Roysched table.
14. Using the STUFF function create a string "12xxxx567" from the string "1234567."
15. Display the first 40 characters of each title, along with the average monthly sales for that title to date (ytd_sales/12). Use the Title table.
16. Show how many books have assigned prices.
17. Display a list of cookbooks with the average cost for all of the books of each type. Use the GROUP BY.

Advanced Questions (Union, Intersect, and Minus)

1. The relational set operators UNION, INTERSECT and MINUS work properly only if the relations are union-compatible. What does union-compatible mean, and how would you check for this condition?
2. What is the difference between UNION and UNION ALL? Write the syntax for each.
3. Suppose that you have two tables, Employees and Employees_1. The Employees table contains the records for three employees: Alice Cordoza, John Cretchakov, and Anne McDonald. The Employees_1 table contains the records for employees: John Cretchakov and Mary Chen. Given that information, what

is the query output for the UNION query? List the query output.

4. Given the employee information in question 3, what is the query output for the UNION ALL query? List the query output.
5. Given the employee information in question 3, what is the query output for the INTERSECT query? List the query output.
6. Given the employee information in question 3, what is the query output for the EXCEPT query? List the query output.
7. What is a cross join? Give an example of its syntax.
8. Explain these three join types:

    1. left outer join
    2. right outer join
    3. full outer join

9. What is a subquery, and what are its basic characteristics?
10. What is a correlated subquery? Give an example.
11. Suppose that a Product table contains two attributes, PROD_CODE and VEND_CODE. The values for the PROD_CODE are: ABC, DEF, GHI and JKL. These are matched by the following values for the VEND_CODE: 125, 124, 124 and 123, respectively (e.g., PROD_CODE value ABC corresponds to VEND_CODE value 125). The Vendor table contains a single attribute, VEND_CODE, with values 123, 124, 125 and 126. (The VEND_CODE attribute in the Product table is a foreign key to the VEND_CODE in the Vendor table.)
12. Given the information in question 11, what would be the query output for the following? Show values.

    1. A UNION query based on these two tables
    2. A UNION ALL query based on these two tables
    3. An INTERSECT query based on these two tables
    4. A MINUS query based on these two tables

Advanced Questions (Using Joins)

1. Display a list of all titles and sales numbers in the Books and Sales tables, including titles that have no sales. Use a join.
2. Display a list of authors' last names and all associated titles that each author has published sorted by the author's last name. Use a join. Save it as a view named: Published Authors.
3. Using a subquery, display all the authors (show last and first name, postal code) who receive a royalty of 100% and live in Alberta. Save it as a view titled: AuthorsView. When creating the view, rename the author's last name and first name as 'Last Name' and 'First Name'.
4. Display the stores that did not sell the title *Is Anger the Enemy?*
5. Display a list of store names for sales after 2013 (Order Date is greater than 2013). Display store name and order date.

6. Display a list of titles for books sold in store name “News & Brews.” Display store name, titles and order dates.
7. List total sales (qty) by title. Display total quantity and title columns.
8. List total sales (qty) by type. Display total quantity and type columns.
9. List total sales (qty*price) by type. Display total dollar value and type columns.
10. Calculate the total number of types of books by publisher. Show publisher name and total count of types of books for each publisher.
11. Show publisher names that do not have any type of book. Display publisher name only.

Appendix A University Registration Data Model Example

Here is a statement of the data requirements for a product to support the registration of and provide help to students of a fictitious e-learning university.

An e-learning university needs to keep details of its students and staff, the courses that it offers and the performance of the students who study its courses. The university is administered in four geographical regions (England, Scotland, Wales and Northern Ireland).

Information about each student should be initially recorded at registration. This includes the student's identification number issued at the time, name, year of registration and the region in which the student is located. A student is not required to enroll in any courses at registration; enrollment in a course can happen at a later time.

Information recorded for each member of the tutorial and counseling staff must include the staff number, name and region in which he or she is located. Each staff member may act as a counselor to one or more students, and may act as a tutor to one or more students on one or more courses. It may be the case that, at any particular point in time, a member of staff may not be allocated any students to tutor or counsel.

Each student has one counselor, allocated at registration, who supports the student throughout his or her university career. A student is allocated a separate tutor for each course in which he or she is enrolled. A staff member may only counsel or tutor a student who is resident in the same region as that staff member.

Each course that is available for study must have a course code, a title and a value in terms of credit points. A course is either a 15-point course or a 30-point course. A course may have a quota for the number of students enrolled in it at any one presentation. A course need not have any students enrolled in it (such as a course that has just been written and offered for study).

Students are constrained in the number of courses they can be enrolled in at any one time. They may not take courses simultaneously if their combined points total exceeds 180 points.

For assessment purposes, a 15-point course may have up to three assignments per presentation and a 30-point course may have up to five assignments per presentation. The grade for an assignment on any course is recorded as a mark out of 100.

The university database below is one possible data model that describes the above set of requirements. The model has several parts, beginning with an ERD and followed by a written description of entity types, constraints, and assumptions.

Design Process

See Figure A.1.

1. The first step is to determine the kernels. These are typically nouns: Staff, Course, Student and Assignment.
2. The next step is to document all attributes for each entity. This is where you need to ensure that all tables are properly normalized.
3. Create the initial ERD and review it with the users.

4. Make changes if needed after the ERD review.
5. Verify the ER model with users to finalize the design.

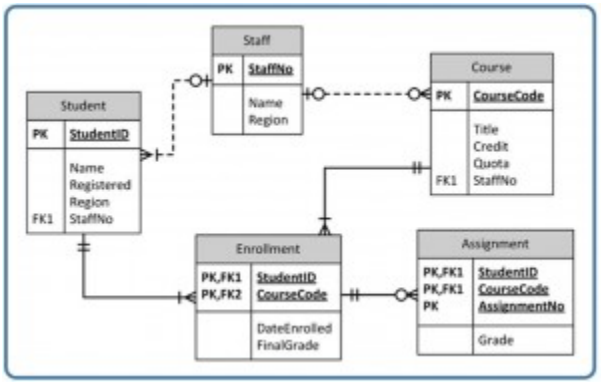


Figure A.1. University ERD. A data model for a student and staff records system by A. Watt.

Entity

Student (StudentID, Name, Registered, Region, StaffNo)

Staff (StaffNo, Name, Region) – This table contains instructors and other staff members.

Course (CourseCode, Title, Credit, Quota, StaffNo)

Enrollment (StudentlD, CourseCode, DateEnrolled, FinalGrade)

Assignment (StudentID, CourseCode, AssignmentNo, Grade)

Constraints

- A staff member may only tutor or counsel students who are located in the same region as the staff member.
- Students may not enroll for more than 180 points worth of courses at any one time.
- The attribute Credit (of Course) has a value of 15 or 30 points.
- A 30-point course may have up to five assignments; a 15-point course may have up to three assignments.
- The attribute Grade (of Assignment) has a value that is a mark out of 100.

Assumptions

- A student has at most one enrollment in a course as only current enrollments are recorded.
- An assignment may be submitted only once.

Relationships (includes cardinality)

Using Figure A.2, note that a student (record) is associated with (enrolled) with a minimum of 1 to a maximum of many courses.

Each enrollment must have a valid student.

Note: Since the StudentID is part of the PK, it can't be null. Therefore, any StudentID entered, must exist in the Student table at least once to a maximum of 1 time. This should be obvious since the PK cannot have duplicates.

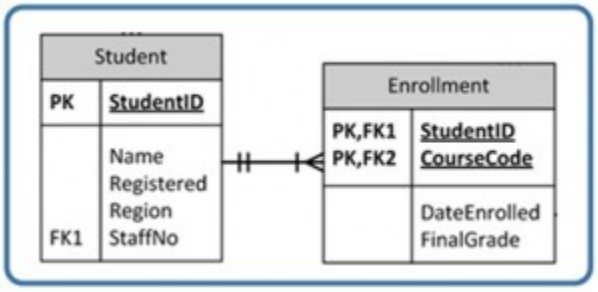


Figure A.2 by A. Watt.

Refer to Figure A.3. A staff record (a tutor) is associated with a minimum of 0 students to a maximum of many students.

A student record may or may not have a tutor.

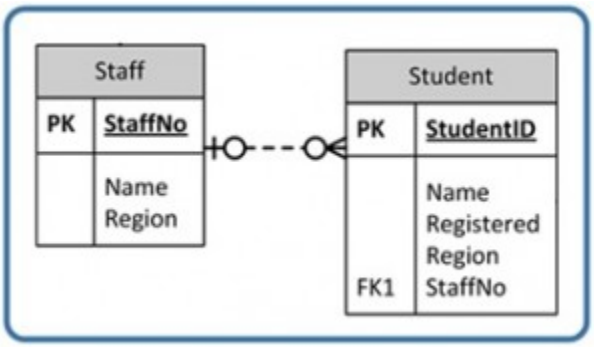


Figure A.3 by A. Watt.

Note: The StaffNo field in the Student table allows null values – represented by the 0 on the left side. However, if a StaffNo exists in the student table it must exist in the Staff table maximum once – represented by the 1.

Refer to Figure A.4. A staff record (instructor) is associated with a minimum of 0 courses to a maximum of many courses.

A course may or may not be associated with an instructor.

Note: The StaffNo in the Course table is the FK, and it can be null. This represents the 0 on the left side of the relationship. If the StaffNo has data, it has to be in the Staff table a maximum of once. That is represented by the 1 on the left side of the relationship.

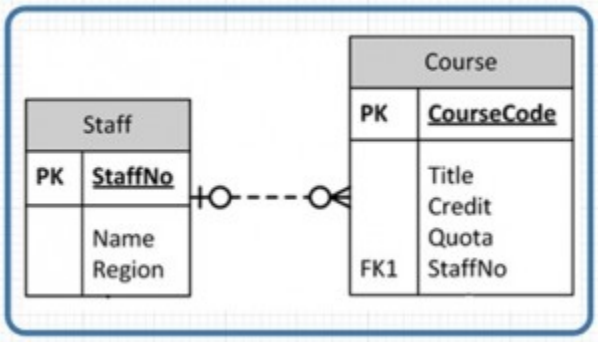


Figure A.4 by A. Watt.

Refer to Figure A.5. A course must be offered (in enrollment) at least once to a maximum of many times.

The Enrollment table must contain at least 1 valid course to a maximum of many.

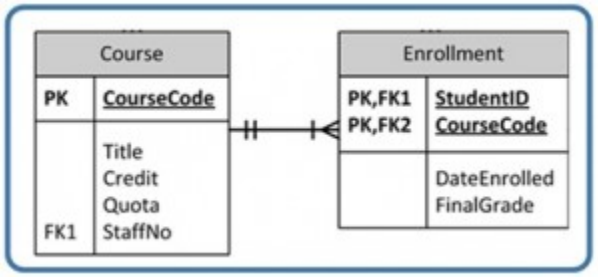


Figure A.5 by A. Watt.

Refer to Figure A.6. An enrollment can have a minimum of 0 assignments or a maximum of many.

An assignment must be associated with at least 1 with a maximum of 1 enrollment.

Note: Every record in the Assignment table must contain a valid enrollment record. One enrollment record can be associated with multiple assignments.

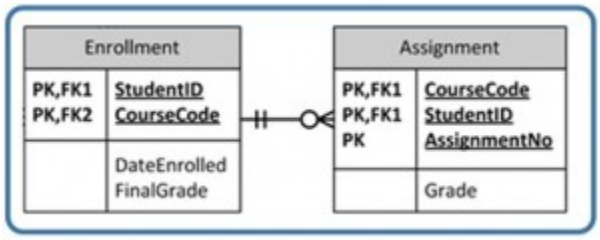


Figure A.6 by A. Watt.

Attribution

This is an adaptation, not a derivation as the author wrote half of it. Source: http://openlearn.open.ac.uk/mod/oucontent/view.php?id=397581&section=8.2

Appendix B Sample ERD Exercises

Exercise 1

Manufacturer

A manufacturing company produces products. The following product information is stored: product name, product ID and quantity on hand. These products are made up of many components. Each component can be supplied by one or more suppliers. The following component information is kept: component ID, name, description, suppliers who supply them, and products in which they are used. Use Figure B.1 for this exercise.

Create an ERD to show how you would track this information.

Show entity names, primary keys, attributes for each entity, relationships between the entities and cardinality.

Assumptions

- A supplier can exist without providing components.
- A component does not have to be associated with a supplier.
- A component does not have to be associated with a product. Not all components are used in products.
- A product cannot exist without components.

ERD Answer

Component(CompID, CompName, Description) PK=CompID

Product(ProdID, ProdName, QtyOnHand) PK=ProdID

Supplier(SuppID, SuppName) PK = SuppID

CompSupp(CompID, SuppID) PK = CompID, SuppID

Build(CompID, ProdID, QtyOfComp) PK= CompID, ProdID

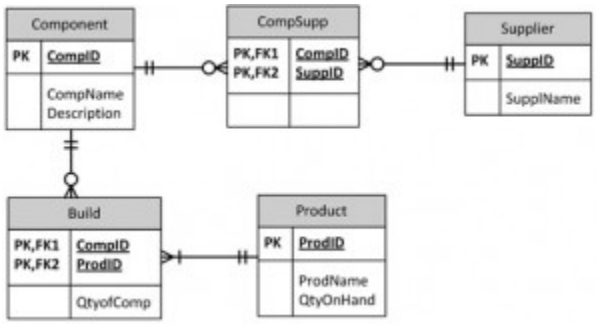


Figure B.1 by A. Watt.

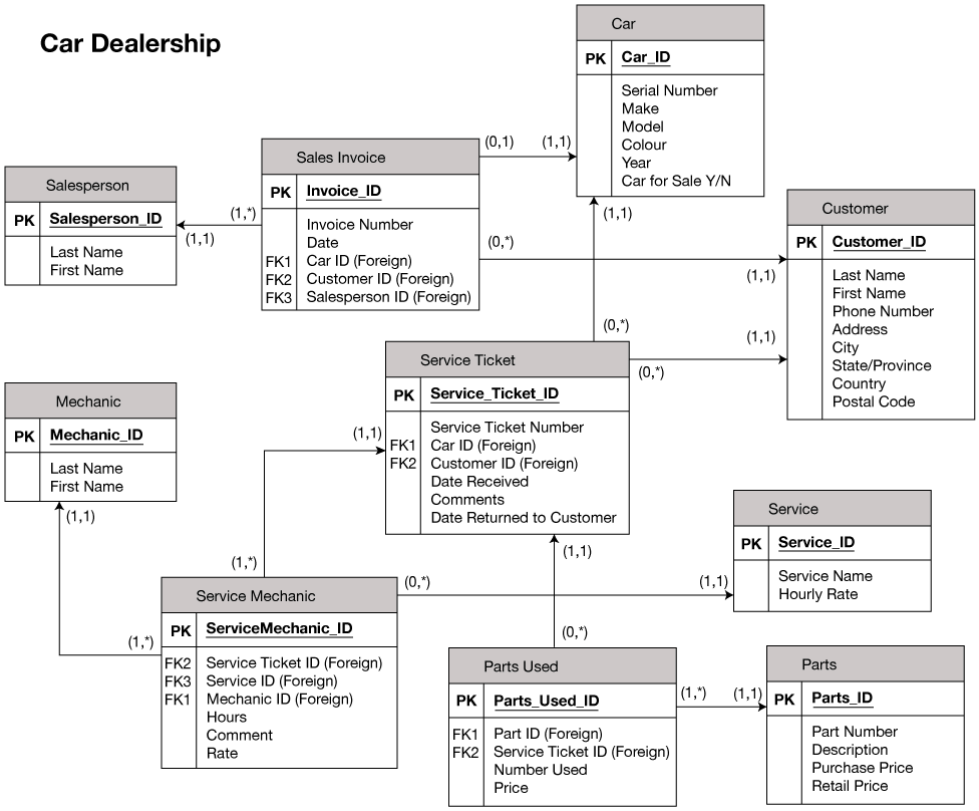
Exercise 2

Car Dealership

Create an ERD for a car dealership. The dealership sells both new and used cars, and it operates a service facility (see Figure B.2). Base your design on the following business rules:

- A salesperson may sell many cars, but each car is sold by only one salesperson.
- A customer may buy many cars, but each car is bought by only one customer.
- A salesperson writes a single invoice for each car he or she sells.
- A customer gets an invoice for each car he or she buys.
- A customer may come in just to have his or her car serviced; that is, a customer need not buy a car to be classified as a customer.
- When a customer takes one or more cars in for repair or service, one service ticket is written for each car.
- The car dealership maintains a service history for each of the cars serviced. The service records are referenced by the car's serial number.
- A car brought in for service can be worked on by many mechanics, and each mechanic may work on many cars.
- A car that is serviced may or may not need parts (e.g., adjusting a carburetor or cleaning a fuel injector nozzle does not require providing new parts).

ERD Answer


Car Dealership
Car
PK Car_ID
Serial Number
Make
Model
Colour
Year
Car for Sale Y/N
Sales Invoice
PK Invoice_ID
Invoice Number
Date
FK1 Car ID (Foreign)
FK2 Customer ID (Foreign)
FK3 Salesperson ID (Foreign)
Salesperson
PK Salesperson_ID
Last Name
First Name
Customer
PK Customer_ID
Last Name
First Name
Phone Number
Address
City
State/Province
Country
Postal Code
Service Ticket
PK Service_Ticket_ID
Service Ticket Number
FK1 Car ID (Foreign)
FK2 Customer ID (Foreign)
Date Received
Comments
Date Returned to Customer
Mechanic
PK Mechanic_ID
Last Name
First Name
Service
PK Service_ID
Service Name
Hourly Rate
Service Mechanic
PK ServiceMechanic_ID
FK2 Service Ticket ID (Foreign)
FK3 Service ID (Foreign)
FK1 Mechanic ID (Foreign)
Hours
Comment
Rate
Parts Used
PK Parts_Used_ID
FK1 Part ID (Foreign)
FK2 Service Ticket ID (Foreign)
Number Used
Price
Parts
PK Parts_ID
Part Number
Description
Purchase Price
Retail Price
(0,1)
(1,1)
(1,*)
(1,1)
(0,*)
(1,1)
(1,1)
(0,*)
(1,1)
(0,*)
(1,1)
(1,*)
(1,1)
(1,*)
(0,*)
(1,1)
(1,1)
(0,*)
(1,*)
(1,1)

Appendix C SQL Lab with Solution

Download the following script: OrdersAndData.sql.

Part 1 – DDL

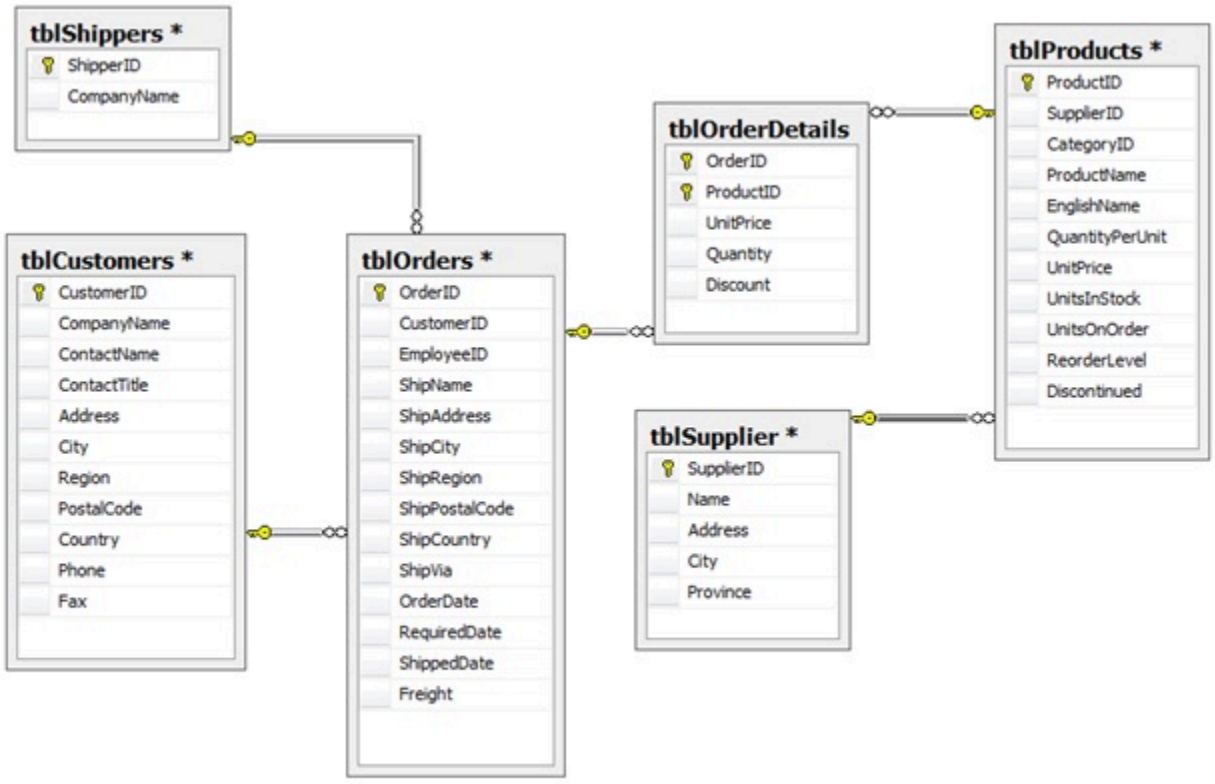


Figure C.1. ERD for Orders and Data.

1. Use the script OrdersAndData.sql that creates the tables and adds the data for the Orders and Data ERD in Figure C.1.
2. Create a database called Orders. Modify the script to integrate the PK and referential integrity. Show the CREATE TABLE statements with the modifications including the constraints given in step 3.
3. Add the following constraints:

- tblCustomers table: Country – default to Canada
- tblOrderDetails: Quantity – > 0
- tblShippers: CompanyName must be unique.
- tblOrders: ShippedDate must be greater than order date.

```
CREATE DATABASE Orders
Go
Use Orders
Go
```

```
Use Orders
Go
CREATE TABLE [dbo].[tblCustomers]
[CustomerID]       nvarchar(5) NOT NULL,
[CompanyName]      nvarchar(40) NOT NULL,
[ContactName]      nvarchar(30) NULL,
[ContactTitle]     nvarchar(30) NULL,
[Address]          nvarchar(60) NULL,
[City]             nvarchar(15) NULL,
[Region]           nvarchar(15) NULL,
[PostalCode]       nvarchar(10) NULL,
[Country]          nvarchar(15) NULL
Constraint     df_country DEFAULT 'Canada',
[Phone]            nvarchar(24) NULL,
[Fax]              nvarchar(24) NULL,
Primary Key (CustomerID)
);
```

```
CREATE TABLE [dbo].[tblSupplier] (
[SupplierID]     int NOT NULL,
[Name]           nvarchar(50) NULL,
[Address]        nvarchar(50) NULL,
[City]           nvarchar(50) NULL,
[Province]       nvarchar(50) NULL,
Primary Key (SupplierID)
);
```

```
CREATE TABLE [dbo].[tblShippers] (
[ShipperID]       int NOT NULL,
[CompanyName]     nvarchar(40) NOT NULL,
Primary Key (ShipperID),<
```

```
CONSTRAINT uc_CompanyName UNIQUE (CompanyName)
);
```

```
CREATE TABLE [dbo].[tblProducts] (
[ProductID]          int NOT NULL,
[SupplierID]         int NULL,
[CategoryID]         int NULL,
[ProductName]        nvarchar(40) NOT NULL,
[EnglishName]        nvarchar(40) NULL,
[QuantityPerUnit]    nvarchar(20) NULL,
[UnitPrice]          money NULL,
[UnitsInStock]       smallint NULL,
[UnitsOnOrder]       smallint NULL,
[ReorderLevel]       smallint NULL,
[Discontinued]       bit NOT NULL,
Primary Key (ProductID),
Foreign Key (SupplierID) References tblSupplier
);
```

```
CREATE TABLE [dbo].[tblOrders] (
[OrderID]           int NOT NULL,
[CustomerID]        nvarchar(5) NOT NULL,
[EmployeeID]        int NULL,
[ShipName]          nvarchar(40) NULL,
[ShipAddress]       nvarchar(60) NULL,
[ShipCity]          nvarchar(15) NULL,
[ShipRegion]        nvarchar(15) NULL,
[ShipPostalCode]    nvarchar(10) NULL,
[ShipCountry]       nvarchar(15) NULL,
[ShipVia]           int NULL,
[OrderDate]         smalldatetime NULL,
[RequiredDate]      smalldatetime NULL,
[ShippedDate]       smalldatetime NULL,
[Freight]           money NULL
Primary Key (OrderID),
Foreign Key (CustomerID) References tblCustomers,
Foreign Key (ShipVia) References tblShippers,
Constraint valid_ShipDate CHECK (ShippedDate > OrderDate)
);
```

```
CREATE TABLE [dbo].[tblOrderDetails] (
[OrderID]       int NOT NULL,
[ProductID]     int NOT NULL,
[UnitPrice]     money NOT NULL,
[Quantity]      smallint NOT NULL,
[Discount]      real NOT NULL,
Primary Key (OrderID, ProductID),
Foreign Key (OrderID) References tblOrders,
Foreign Key (ProductID) References tblProducts,
Constraint Valid_Qty Check (Quantity > 0)
);
Go
```

Part 2 – Create the Following SQL Statements

1. Show a list of customers and the orders they generated during 2014. Display customer ID, order ID, order date and date ordered.

```
Use Orders
Go
SELECT CompanyName, OrderID, RequiredDate as 'order date', OrderDate as 'date ordered'
FROM tblcustomers  JOIN tblOrders on tblOrders.CustomerID = tblCustomers.CustomerID
WHERE Year(OrderDate) = 2014
```

2. Using the ALTER TABLE statement, add a new field (Active) in the tblcustomer. Default it to True.

```
ALTER TABLE tblCustomers
ADD Active bit DEFAULT ('True')
```

3. Show all orders purchased before September 1, 2012. Display company name, date ordered and total amount of order (include freight).

```
SELECT tblOrders.OrderID, OrderDate as 'Date Ordered', sum(unitprice*quantity*(1-discount))+ freight as
'Total Cost'
```

```
FROM tblOrderDetails join tblOrders on tblOrders.orderID = tblOrderDetails.OrderID
WHERE OrderDate < 'September 1, 2012'
GROUP BY tblOrders.OrderID, freight, OrderDate
```

4. Show all orders that have been shipped via Federal Shipping. Display OrderID, ShipName, ShipAddress and CustomerID.

```
SELECT OrderID, ShipName, ShipAddress, CustomerID
FROM tblOrders join tblShippers on tblOrders.ShipVia = tblShippers.ShipperID
WHERE CompanyName= 'Federal Shipping'
```

5. Show all customers who have not made purchases in 2011.

```
SELECT CompanyName
FROM tblCustomers
WHERE CustomerID not in
(  SELECT CustomerID
FROM  tblOrders
WHERE Year(OrderDate) = 2011
)
```

6. Show all products that have never been ordered.

```
SELECT ProductID from tblProducts
Except
SELECT ProductID from tblOrderDetails
```

OR

```
SELECT Products.ProductID,Products.ProductName
FROM Products LEFT JOIN [Order Details]
ON Products.ProductID = [Order Details].ProductID
WHERE [Order Details].OrderID IS NULL
```

7. Show OrderIDs for customers who reside in London. Use a subquery. Display CustomerID, CustomerName and OrderID.

```
SELECT Customers.CompanyName,Customers.CustomerID,OrderID
FROM Orders
LEFT JOIN Customers ON Orders.CustomerID = Customers.CustomerID
WHERE Customers.CompanyName IN
(SELECT CompanyName
FROM Customers
WHERE City = 'London')
```

8. Show products supplied by Supplier A and Supplier B. Display product name and supplier name.

```
SELECT ProductName, Name
FROM tblProducts JOIN tblSupplier on tblProducts.SupplierID = tblSupplier.SupplierID
WHERE Name Like 'Supplier A' or Name Like 'Supplier B'
```

9. Show all products that come in boxes. Display product name and QuantityPerUnit.

```
SELECT EnglishName, ProductName,  QuantityPerUnit
FROM tblProducts
WHERE QuantityPerUnit like '%box%'
ORDER BY EnglishName
```

Part 3 – Insert, Update, Delete, Indexes

1. Create an Employee table. The primary key should be EmployeeID (autonumber). Add the following fields: LastName, FirstName, Address, City, Province, Postalcode, Phone, Salary. Show the CREATE TABLE statement and the INSERT statements for the five employees. Join the employee table to the tblOrders. Show the script for creating the table, setting constraints and adding employees.

```
Use Orders
CREATE TABLE [dbo].[tblEmployee](
EmployeeID Int IDENTITY NOT NULL ,
FirstName varchar (20) NOT NULL,
```

```
LastName varchar (20) NOT NULL,
Address varchar (50),
City varchar(20), Province varchar (50),
PostalCode char(6),
Phone char (10),
Salary Money NOT NULL,
Primary Key (EmployeeID)
```

```
Go
INSERT into tblEmployees
Values ('Jim', 'Smith', '123 Fake', 'Terrace', 'BC', 'V8G5J6', '2506155989', '20.12'),
('Jimmy', 'Smithy', '124 Fake', 'Terrace', 'BC', 'V8G5J7', '2506155984', '21.12'),
('John', 'Smore', '13 Fake', 'Terrace', 'BC', 'V4G5J6', '2506115989', '19.12'),
('Jay', 'Sith', '12 Fake', 'Terrace', 'BC', 'V8G4J6', '2506155939', '25.12'),
('Jig', 'Mith', '23 Fake', 'Terrace', 'BC', 'V8G5J5', '2506455989', '18.12');
Go
```

2. Add a field to tblOrders called TotalSales. Show DDL – ALTER TABLE statement.

```
ALTER TABLE tblOrders
ADD Foreign Key (EmployeeID) references tblEmployees (EmployeeID)
```

3. Using the UPDATE statement, add the total sale for each order based on the order details table.

```
UPDATE tblOrders
Set TotalSales = (select sum(unitprice*quantity*(1-discount))
FROM tblOrderDetails
WHERE tblOrderDetails.OrderID= tblOrders.OrderID
GROUP BY OrderID
```

About the Authors

Primary Author: Adrienne Watt

Adrienne Watt holds a computer systems diploma (BCIT), a bachelor's degree in technology (BCIT) and a master's degree in business administration (City University).

Since 1989, Adrienne has worked as an educator and gained extensive experience developing and delivering business and technology curriculum to post-secondary students. During that time, she ran a successful software development business. In the business, she worked as an IT professional in a variety of senior positions including project manager, database designer, administrator and business analyst. Recently she has been exploring a wide range of technology-related tools and processes to improve delivery methods and enhance learning for her students.

Contributing Author: Nelson Eng

Nelson Eng completed his bachelor's degree in commerce (accounting and management information systems) from the University of British Columbia and a master's degree in computer science from the University of Western Ontario. He spent 11 years as a computer systems coordinator with Science World of B.C. and 14 years as a computer science and information systems instructor with Douglas College.

Versioning History

This page provides a record of edits and changes made to this book since its initial publication. Whenever edits or updates are made, we make the required changes in the text and provide a record and description of those changes here. If the change is minor, the version number increases by 0.1. However, if the edits involve substantial updates, the version number goes up to the next full number. The files on our website always reflect the most recent version, including the print-on-demand copy.

If you find an error in this book, please fill out the Report an Error form.

Version	Date	Change	Details
1.1	October 24, 2014	Book published in the B.C. Open Textbook Collection.	
1.2	February 4, 2019	Book update to conform to BCcampus style.	Added a Versioning History page, provided missing publication information, updated the book cover, updated the About the Book chapter, and updated the copyright statement.
1.3	June 11, 2019	Updated the book's theme.	The styles of this book have been updated, which may affect the page numbers of the PDF and print copy.
1.4	January 27, 2022	Text error.	E.F. Codd was incorrectly referenced as C.F. Todd in Chapter 7 The Relational Data Model.