**Chapter 8**

**Need for database redesign**

• **Database redesign** is necessary:

– To fix mistakes made during the initial database design.

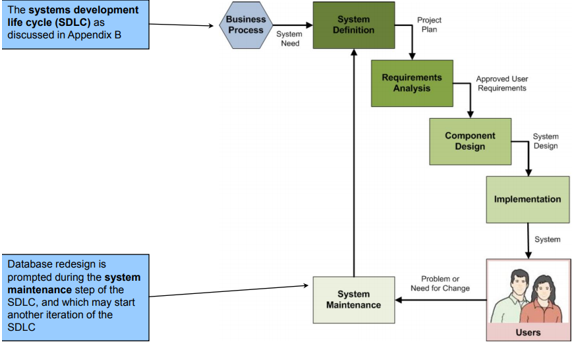
– To adapt the database to changes in system requirements.

• Because information systems and organizations create each other, a new information system will cause changes in systems requirements:

– When a new system is installed, users can behave in new ways.

– As the users behave in the new ways, they will want changes to the system to accommodate their new behaviors.

**System Maintenance in the SDLC**



**Table Showing Constraint Assumption Violation**



**Correlated Subqueries**

A **correlated subquery** looks similar to a regular subquery.

• A regular subquery can be processed from the bottom up.

• For a correlated subquery, the processing is **nested**, i.e., a row from an upper query statement is used in comparison with rows in a lower-level query.

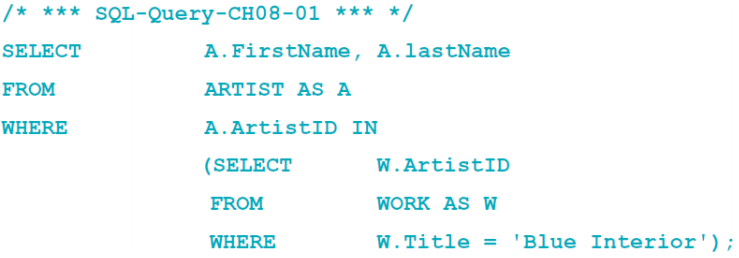
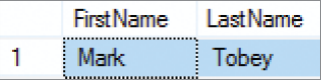
**Non-Correlated Subquery**

We used the following type of subquery in Chapter 2.

• It contains two separate tables in the levels of the query.

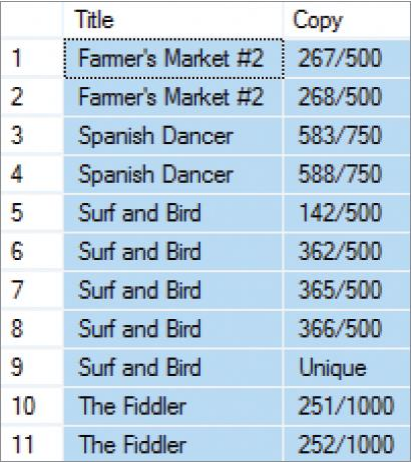
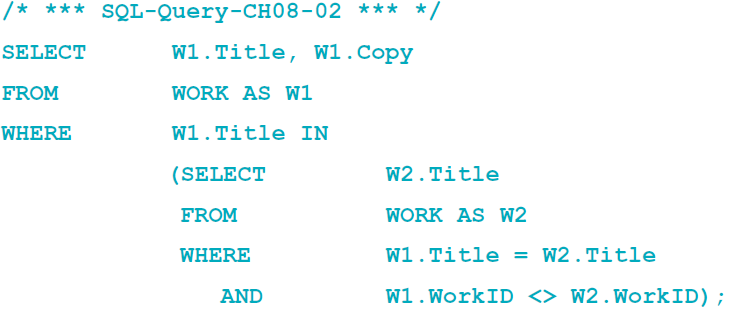
– ARTIST in the top level query

– WORK in the subquery

**Correlated Subquery to find row with the same title**

The following is a correlated subquery - It contains the same tables in both levels of the query.



**The Difference between regular and correlated subqueries**

WHERE w1.Title = w2.Title AND w1.WorkID <> w2.WorkID;

**A Common Trap**

SELECT w1.Title, w1.Copy FROM WORK AS w1 WHERE w1.WorkID IN (SELECT w2.WorkID FROM WORK AS w2 WHERE w1.Title = w2.Title AND w1.WorkID <> w2.WorkID);

**Checking Functional Dependencies**

The following correlated subquery can be used to check for any rows that violate the functional dependency

**Department -> DeptPhone**

SELECT e1.EmployeeNumber, e1.Department, e1.DeptPhone From EMPLOYEE AS e1 WHERE e1.Department IN

(SELECT e2.Department FROM EMPLOYEE AS e2 WHERE e1.Department = e2.Department AND e1.DeptPhone <> e2.DeptPhone);

**Checking Functional Dependencies Results**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **EmployeeNumber** | **Department** | **DeptPhone** |
| **1** | **200** | **Finance** | **834-2100** |
| **2** | **300** | **Finance** | **834-21000** |

**SQL EXIST and NOT EXISTS Comparison Operators 1**

|  |  |
| --- | --- |
| **SQL Comparison Operators** | |
| **Operator** | **Meaning** |
| **EXISTS** | **Is a non-empty set of values** |
| **NOT EXISTS** | **Is an empty set** |

**SQL EXISTS and NOT EXISTS Comparison Operators 2**

Using the SQL EXISTS comparison operator and the SQL NOT EXISTS comparison operator (both discussed in Chapter 2) we can create specialized forms of correlated subqueries.

– An EXISTS condition is true if any row in the subquery meets the specified conditions.

– A NOT EXISTS condition is true only if all rows in the subquery do not meet the specified condition.

• The use of a double NOT EXISTS can be used to find rows that have some specified condition to every row of a table.

**Checking Functional Dependencies**

Here is the code to check the previous functional dependency using the SQL EXISTS comparison operator:

SELECT e1.EmployeeNumber, e1.Department, e1.DeptPhone FROM EMPLOYEE AS e1 WHERE EXISTS

(SELECT e2.Department FROM EMPLOYEE AS e2 WHERE e1.Department = e2.Department AND e1.DeptPhone <> e2.DeptPhone);

**Double NOT EXISTS**

SELECT a.FirstName, a.LastName FROM ARTIST AS a WHERE NOT EXISTS

(SELECT c.CustomerID FROM CUSTOMER AS c WHERE NOT EXISTS

(SELECT CAI.CustomerID FROM CUSTOMER\_ARTIST\_INT AS CAI WHERE c.CustomerID = CAI.CustomerID and a.ArtistID = CAI.ArtistID));

|  |  |  |
| --- | --- | --- |
|  | FirstName | LastName |
|  |  |  |

**Database Redesign**

Three principles for database redesign:

-Measure twice and cut once: **understand** the current structure and contents of the database before making any structure changes.

– **Test** the new changes on a test database before making real changes.

– Create a complete backup of the operational database before making any structure changes.

• **Technique: Reverse Engineering (RE)**

**Reverse Engineering (RE)**

Reverse engineering (RE) is the process of reading and producing a data model from a database schema.

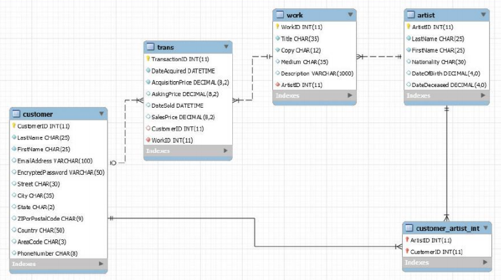
• A reverse engineered (RE) data model:

– Provides a basis to begin the database redesign project.

– Is neither truly a conceptual nor an internal schema as it has characteristics of both.

– Should be carefully reviewed because it almost always has missing information.

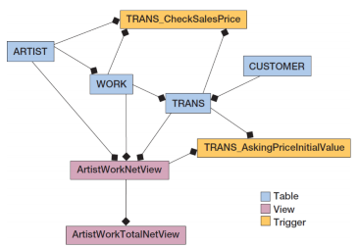
**Reverse Engineered Data Model**



**Dependency Graphs**

Dependency graphs are diagrams used to portray the dependency of one element on another.

**Composite Dependency Graph [Incomplete]**



**Database Backup and Test Databases**

Before making any changes to an operational database:

– A complete backup of the operational database should be made.

– Any proposed changes should be thoroughly tested.

• Three different copies of the database schema used in the redesign process:

– A small test database for initial testing

– A large test database for secondary testing

– The operational database

**Database Redesign Changes**

• Changing tables and columns

|  |  |  |  |
| --- | --- | --- | --- |
| – Changing table names  table columns | – Adding and dropping | – Changing data type or constraints | – Adding and dropping constraints |

• Changing relationships

– Changing cardinalities

– Adding and deleting relationships

– Adding and removing relationships for denormalization

**Changing Table Names**

• Although SQL or DBMS specific commands exist, there is no good command to change a table name except in the most simple cases.

– The table needs to be re-created under the new name, tested, and the old table is dropped.

• Changing a table name has a surprising number of potential consequences.

– Therefore, using views defined as table aliases is more appropriate.

– Only views that define the aliases would need to be changed when the source table name is changed.

**Adding Columns 1**

• To add NULL columns to a table: ALTER TABLE WORK ADD DateCreated Date NULL;

**Adding Columns 2**

• Other column constraints, e.g., DEFAULT or UNIQUE, may be included with the column definition.

ALTER TABLE WORK ADD DateCreated Date NULL DEFALT ‘01/01/1900’;

• Newly added DEFAULT constraint will be applied to only new rows, existing rows will have null values.

UPDATE WORK SET DateCreated = ‘01/01/1900’ WHERE DateCreated IS NULL;

**Adding Columns 3**

• Three steps to add a NOT NULL column:

|  |  |  |
| --- | --- | --- |
| – Add the column as NULL. | – Add data to every row. | – Alter the column constraint to NOT NULL. |

ALTER TABLE WORK ALTER COLUMN DateCreated Date NOT NULL;

**Dropping Columns**

To drop nonkey columns:

ALTER TABLE WORK DROP COLUMN DateCreated;

• To drop a foreign key column, the foreign key constraint must first be dropped.

• To drop the primary key, all foreign keys using the primary key must first be dropped; followed by dropping the primary key constraint.

**Changing Data Type or Constraints**

• Use the SQL ALTER TABLE ALTER COLUMN statement to change data types and constraints.

• For some changes, data will be lost or the DBMS may refuse the change.

• To change a constraint from NULL to NOT NULL, all rows must have a value first.

**Changing Data Type or Constraints**

• Converting more specific data type e.g., date, money, and numeric, to char, or varchar will usually succeed.

– Changing a data type from char or varchar to a more specific type can be a problem.

• Example:

ALTER TABLE ARTIST ALTER COLUMN DateOfBirth Numeric(4,0) NULL;

**Adding and Dropping Constraints**

Use the SQL ALTER TABLE ADD (DROP) CONSTRAINT statement to add (remove) constraints

• Example

ALTER TABLE ARTIST ADD CONSTRAINT NumericBirthYearCheck CHECK (DateOfBirth>1900 AND DateOfBirth<2100);

**Changing Minimum Cardinalities**

• On the parent side:

– To change from zero to one, change the foreign key constraint from NULL to NOT NULL.

• Can only be done if all the rows in the table have a value.

– To change from one to zero, change the foreign key constraint from NOT NULL to NULL.

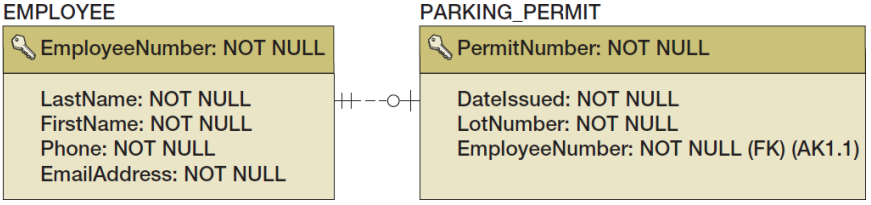
• On the child side:

– Add (to change from zero to one) or drop (to change from one to zero) triggers that enforce the constraint.

**Changing Maximum Cardinalities: 1:1 to 1:N**

• If the foreign key is in the correct table, remove the unique constraint on the foreign key column.

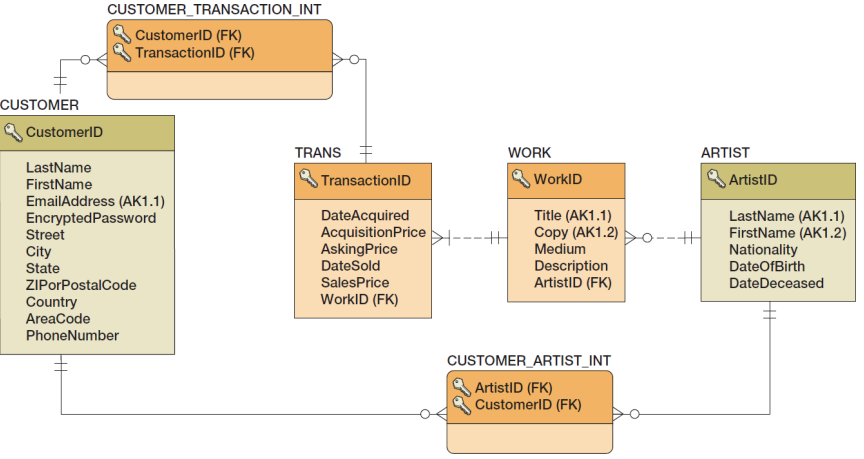
• If the foreign key is in the wrong table, move the foreign key to the correct table and do not place a unique constraint on that table.

**Changing Maximum Cardinalities: 1:1 to 1:N Example**

**Changing Maximum Cardinalities: 1:N to N:M**

• Build a new intersection table and move the key and foreign key values to the intersection table

**Changing Maximum Cardinalities: 1:N to N:M Example**



**Reducing Cardinalities**

• Reducing cardinalities may result in data loss.

• Reducing N:M to 1:N:

– Create a foreign key in the parent table and move one value from the intersection table into that foreign key.

• Reducing 1:N to 1:1:

– Remove any duplicates in the foreign key and then set a uniqueness constraint on that key.

**Adding and Deleting Relationships**

• Adding new tables and relationships:

– Add the tables and relationships using CREATE TABLE statements with FOREIGN KEY constraints.

– If an existing table has a child relationship to the new table, add a FOREIGN KEY constraint using the existing table.

• Deleting relationships and tables:

– Drop the foreign key constraints and then drop the tables.

**Forward Engineering**

• **Forward engineering** is the process of applying data model changes to an existing database.

• Results of forward engineering should be tested before using it on an operational database.

• Some tools will show the SQL that will execute during the forward engineering process:

– If so, that SQL should be carefully reviewed.