LESSON

Designing and Building a Relational Database

Visit our website

Introduction

WELCOME TO THE DESIGNING AND BUILDING A RELATIONAL DATABASE LESSON!

In this lesson, we introduce the relational database and related concepts, including data independence, data modelling, and types of database languages. We will also discuss how to evaluate and design good table structures with normalisation to control data redundancies, and therefore avoid data anomalies, and look at the concept, role, and importance of indexing. At the end there is a task for you to put into practice what you have learned.

INTENDED LEARNING OUTCOMES

You will know that you have succeeded at this task if, by the time you complete it, you are able to:

- Understand and explain key data concepts
- Understand and explain key relational database concepts
- Understand and explain normalisation
- Normalise data to 3NF

Refer to the self-assessment table provided after the programming task near the end of the document for a checklist to assess your own degree of achievement of each of these learning outcomes. Try to gauge your own learning and address any weak areas before submitting your task, which will become part of your formative assessment portfolio.

This lesson addresses Unit Standard 114048: Create database access for a computer application using structured query language.

Specific Outcomes:

1. **SO1: Review the requirements for database access for a computer application using SQL (Range: Piecemeal development, Data sharing, Integration, Abstraction, Data Independence, Data models, Data Definition Language, Data Manipulation Language, Data Control, Language (at least 4))**
2. **SO2: Design database access for a computer application using SQL. (Range: Simple indexes, Multi-level indexes, Index-sequential file, Tree structures, SQL CREATE INDEX and DROP INDEX statements (at least 3))**
3. **SO3: Write program code for database access for a computer application using SQL. (Range: Row types, User-defined types, User defined routines, Reference types, Collection types, Support for large objects, Stored procedures, Triggers (at least 3))**

4. **SO4: Test programs for a computer application that accesses a database using SQL. (Range: Logic tests; Access functions; Debugging.)**
5. **SO5: Document programs for a computer application that accesses a database using SQL. (Range: Clarity; Simplicity; Ease of Use)**

Review the requirements for database access for a computer application using SQL (SO1)

There are a number of concepts you need to understand in order to be able to properly consider the requirements for database access for a computer application using SQL. These are explained below, drawing from the Bella Visage Learner Guide for the ***National Certificate: Information Technology (Systems Development)***.

1.1 DATA CONCEPTS

ABSTRACTION

Abstraction is the process of separating ideas from specific instances of those ideas at work. Computational structures are defined by their meanings (semantics), while hiding away the details of how they work. Abstraction tries to factor out details from a common pattern so that programmers can work close to the level of human thought, leaving out details which matter in practice, but are immaterial to the problem being solved. For example, a system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer; low-level abstraction layers expose details of the computer hardware where the program runs, while high-level layers deal with the business logic of the program.
Abstraction captures only those details about an object that are relevant to the current perspective; in both computing and in mathematics, numbers are concepts in programming languages. Numbers can be represented in myriad ways in hardware and software, but, irrespective of how this is done, numerical operations will obey identical rules.

Abstraction can apply to control or to data: **Control abstraction** is the abstraction of actions while **data abstraction** is that of data structures.

- Control abstraction involves the use of subprograms and related concepts' control flows
- Data abstraction allows handling data bits in meaningful ways. For example, it is the basic motivation behind data typing (strings, integers, floats, etc).

The recommendation that programmers use abstractions whenever suitable in order to avoid duplication (usually of code) is known as the **abstraction principle**. The same term is used to describe the requirement that a programming language provide suitable abstractions.

DATA INDEPENDENCE

One of the biggest advantages databases offer is data independence. It means we can change the conceptual schema at one level without affecting the data at another level. We can change the structure of the database without affecting the data required by users and programs. Data independence is essentially the principle that each higher level of the data architecture is immune to changes to the next, lower level of the architecture. The logical scheme stays unchanged even though the storage space or type of some data may change for reasons of optimisation or reorganisation. Here, the external schema does not change. Internal schema changes may be required if physical schema are reorganised. Physical data independence is present in most databases and file environments in which aspects such as the hardware storage of data encoding, the exact location of data on the disk, the merging of records, and so on, are hidden from the user.

Types of data independence

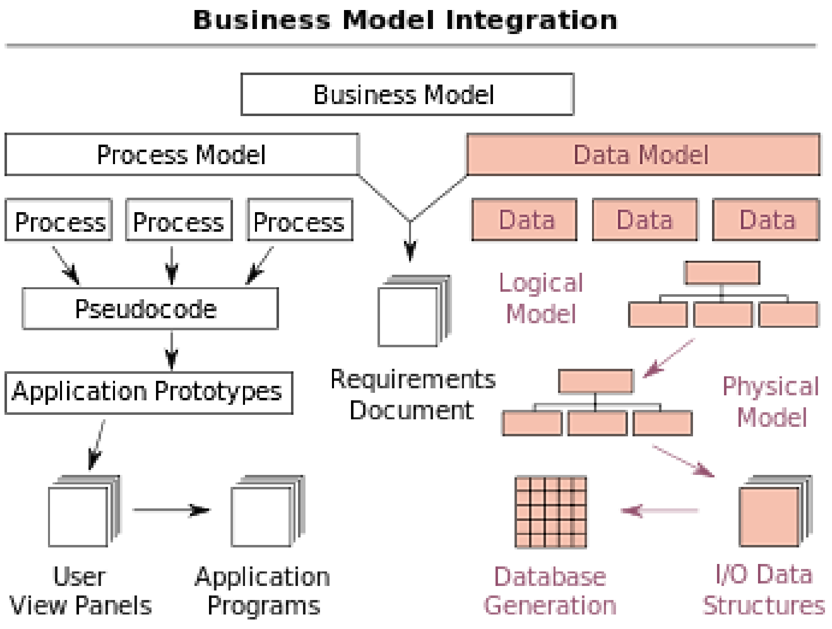
Data can be considered physically independent or logically independent.

- **Physical data independence** is the ability to modify a physical schema without causing application programs to require rewriting. Modifications at the physical level are occasionally necessary to improve performance. When physical data independence exists, this means we change the physical storage/level without affecting the conceptual or external view of the data. The new changes are absorbed by mapping techniques.

- **Logical data independence** is the ability to modify the logical data schema without causing application programs to require rewriting. Modifications at the logical level are necessary whenever the logical structure of the database is altered (for example, when a new account type such as money-market accounts are added to a banking system). Logical Data independence means that we can e.g. add some new fields to a database requiring adding columns to one or more tables, or remove fields requiring removing some columns from one or more tables, without the user view and associated programs requiring any changes. Logical data independence is more difficult to achieve than physical data independence, as application programs are heavily dependent on the logical structure of the data that they access.

DATA MODELS

The term “data models” is used in two related senses. In the first, a data model is a description of the objects represented by a computer system together with their properties and relationships; these are typically "real world" objects such as products, suppliers, customers, and orders. In the second sense, a data model is a collection of concepts and rules used in defining how data are stored and related,: for example the relational model uses relations and tuples, while the network model uses records, sets, and fields.



The figure above is an example of the interaction between process and data models. Here, the data model is based on data, data relationships, data semantics (the meaning of the data), and data constraints. A data model provides the details of information to be stored, and is of primary use when the final product is the generation of computer software code.

Data models are often used as an aid to communication between business people defining the requirements for a computer system and technical people defining the design in response to those requirements. They are used to show the data needed and created by business processes.

A data model is essentially a navigation tool for both business and IT professionals, which uses a set of symbols and text to precisely explain a subset of real information to improve communication within the organisation and thereby lead to a more flexible and stable application environment.

A data model explicitly determines the structure of data. Data models are specified in a data modelling notation, which is often graphical in form.

A data model can be sometimes referred to as a data structure, especially in the context of programming languages. Data models are often complemented by function models, especially in the context of enterprise models.

DATABASE LANGUAGES

Database languages are special-purpose languages, which either define data, manipulate data, or query data (three types).

Data Definition Language (DDL)

A data definition language (DDL) defines data types and the relationships among them.

Data manipulation language (DML)

A data manipulation language (DML) performs tasks such as inserting, updating, updating, or deleting data in a database. Performing read-only queries of data is sometimes also considered a component of DMLs. DMLs are divided into two types, procedural programming and declarative programming. DMLs were initially only used within computer programs, but with the advent of SQL have come to be used interactively by database administrators and end users.

Query languages

A query language, such as **SQL** (Structured Query Language) facilitates searching for information and computing derived information, enabling users to ask questions of the database and get answers in which data specific to the question is presented in such a way as to provide information.

SQL

You will learn much more about SQL in the next unit, entitled **Introduction to SQL**. Here, we are just going to provide a brief overview of SQL in the context of a DML/DDL and a query language. SQL is used to retrieve and manipulate data in a relational database. As a DML, SQL consists of data change statements. which modify stored data but not the schema or database objects. Manipulation of persistent database objects, e.g., tables or stored procedures, via the SQL schema statements rather than the data stored within them, is considered to be part of a separate DDL. In SQL these two categories are similar in their detailed syntax, data types, expressions etc., but distinct in their overall function.

Data manipulation languages have their functional capability organised by the initial word in a statement, which is almost always a verb. In the case of SQL, these verbs are:

- SELECT … FROM … WHERE …
- INSERT INTO … VALUES …
- UPDATE … SET … WHERE …
- DELETE FROM … WHERE …

The purely read-only SELECT query statement is classed with the 'SQL-data' statements and so is considered by the standard to be outside of DML. The INTO form, however, is considered to be DML because it manipulates (i.e. modifies) data. In common practice though, this distinction is not made and SELECT is widely considered to be part of DML.

Most SQL database implementations extend their SQL capabilities by providing imperative functionality, i.e. procedural languages. Examples of these are Oracle's PL/SQL and DB2's SQL_PL.

DMLS tend to have many different flavours and capabilities between database vendors. There have been a number of standards established for SQL by ANSI, but vendors still provide their own extensions to the standard while often also not implementing the entire standard.

1.2 RELATIONAL DATABASE CONCEPTS

WHAT IS A RELATIONAL DATABASE?

As mentioned in the first lesson, a relational database is a database that organises data as a set of formally described tables. Data can then be accessed or reassembled from these tables in many different ways without having to reorganise the database tables.

Each table in a relational database has a unique name and may relate to one or more other tables in the database through common values. These tables contain rows (records) and columns (fields). Each row contains a unique instance of data for the categories defined by the columns. For example, a table that describes a *Customer* can have columns for *Name, Address, Phone Number,* and so on. Rows are sometimes referred to as tuples and columns are sometimes referred to as attributes. A relationship is a link between two tables. Relationships make it possible to find data in one table that pertains to a specific record in another table.

Tables in a relational database often contain a primary key, which is an index for a column or group of columns used as a unique identifier for each row in the table (you will learn more about indexing, and indices, in the next section). For example, a *Customer* table might have a column called *CustomerID* that is unique for every row. This makes it easy to keep track of a record over time and to associate a record with records in other tables.

Tables may also contain foreign keys, which are columns that link to primary key columns working as indices in other tables, thereby creating a relationship. For example, the *Customers* table might have a foreign key column called *SalesRep* that links to *EmployeeID*, which is the primary key in the *Employees* table.

A Relational Database Management System (RDBMS) is the software used for creating, manipulating, and administering a database. Most commercial RDBMSes use the Structured Query Language (SQL). SQL queries are the standard way to access data from a relational database. SQL queries can be used to create, modify, and delete tables as well as select, insert, and delete data from existing tables. You will learn more about SQL in a later lesson.

REQUIREMENTS FOR DATABASE ACCESS FOR A COMPUTER APPLICATION USING SQL (SO2)

There are a number of concepts you need to understand in order to be able to properly design database access for a computer application using SQL. These are explained below, drawing from the Bella Visage Learner Guide for the ***National Certificate: Information Technology (Systems Development)***.

INDEXING

Simple indices

Given the fundamental importance of indices in databases, it is unfortunate how often the proper design of indices is neglected. It often turns out that the programmer understands detail, but not the broad picture of what indices do.

What is an index?

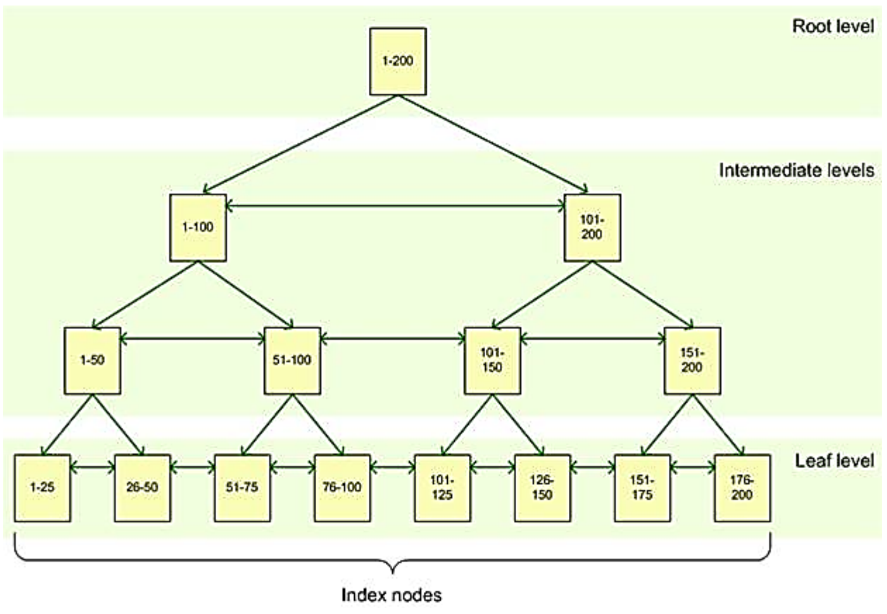
One of the most important routes to high performance in a SQL database is the index. Indices speed up the querying process by providing swift access to rows in the data tables, similarly to the way a book's index helps you find information quickly within that book. Most of this information applies to indices in software like SQL Server, and the underlying structure remains relatively the same across versions thereof, and similar programs.

Index Structures

Indices are created on columns in tables or views. The index provides a fast way to look up data based on the values within those columns. For example, if you create an index on the primary key and then search for a row of data based on one of the primary key values, SQL first finds that value in the index, and then uses the index to quickly locate the entire row of data. Without the index, a table scan would have to be performed in order to locate the row, which can have a significant effect on performance.

You can create indices on most columns in a table or a view. You can also create indices on XML columns, but those indices are slightly different from the basic index and are beyond the scope of this lesson.

An index is made up of a set of pages (index nodes) that are organised in a B-tree structure. This structure is hierarchical in nature, with the root node at the top of the hierarchy and the leaf nodes at the bottom, as shown in the figure below.



B-tree structure of a SQL Server index

Index Design

As beneficial as indices can be, they must be designed carefully. Because they can take up significant disk space, you don't want to implement more indices than necessary. In addition, indices are automatically updated when the data rows themselves are updated, which can lead to additional overhead and can affect performance. As a result, index design should take into account a number of considerations.

Database considerations

As mentioned above, indices can enhance performance because they can provide a quick way for the query engine to find data. However, you must also take into account whether and how much you're going to be inserting, updating, and deleting data. When you modify data, the indices must also be modified to reflect the changed data, which can significantly affect performance. You should consider the following guidelines when planning your indexing strategy:

- For tables that are heavily updated, use as few columns as possible in the index, and don't over-index the tables.

- If a table contains a lot of data but data modifications are low, use as many indices as necessary to improve query performance. However, use indices judiciously on small tables because the query engine might take longer to navigate the index than to perform a table scan.

- For clustered indices, try to keep the length of the indexed columns as short as possible. Ideally, try to implement your clustered indices on unique columns that do not permit null values. This is why the primary key is often used for the table's clustered index, although query considerations should also be taken into account when determining which columns should participate in the clustered index.

- The uniqueness of values in a column affects index performance. In general, the more duplicate values you have in a column, the more poorly the index performs. On the other hand, the more unique each value, the better the performance. When possible, implement unique indices.

- For composite indices, take into consideration the order of the columns in the index definition. Columns that will be used in comparison expressions in the WHERE clause (such as WHERE FirstName = 'Charlie') should be listed first. Subsequent columns should be listed based on the uniqueness of their values, with the most unique listed first.

- You can also index computed columns if they meet certain requirements. For example, the expression used to generate the values must be deterministic (which means it always returns the same result for a specified set of inputs).

Planning for Queries

Another consideration when setting up indices is how the database will be queried - in other words, how users will be able to ask questions of the database and get resultant, answering, data returned to them by the query language. As mentioned above, you must take into account the frequency of data modifications. In addition, you should consider the following guidelines:

- Try to insert or modify as many rows as possible in a single statement, rather than using multiple queries.

- Create no clustered indices on columns used frequently in your statement's predicates and join conditions.

- Consider indexing columns used in exact-match queries.

Multi-Level indices

The indexing described so far involves ordered index files. A binary search is applied to the index to locate pointers to disk blocks or records in the file having a specific index field value.

A binary search (recall the *Sorting and Searching* unit) requires $\log_2 b_i$ block accesses for an index with b_i blocks. Each step of the binary search reduces the part of the index file we search by a factor of 2. (Each step divides the search space by 2, or halves the search space).

With multilevel indices, the idea is to reduce the part of the index that we continue to search by a larger factor, the blocking factor of the index, where the bfr_i is greater than 2. The blocking factor of the index, bfr_i is called the fan-out, or fo of the multilevel index.

Searching a multilevel index requires approximately ($\log_{fo} b_i$) block accesses which is a smaller number than for a binary search if the fan-out is larger than 2. If the fan out is equal to 2, there is no difference in the number of block accesses.

The index file is called the first level of a multilevel index. It is an ordered file with a distinct value for each key value K(i). Therefore we can create a *primary index* for the first level. This index to the first level is called the second level of the multilevel index.

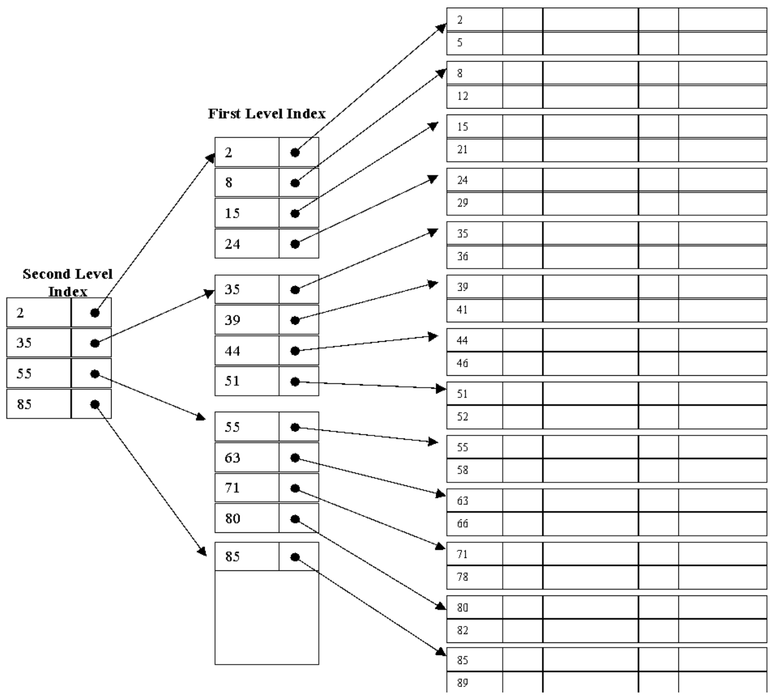
The second level is a primary index, therefore we can use block anchors, and the second level has *one entry for each block* in the first level index. The blocking factor for all other levels is the same as for the first level, because the size of each index entry is the same. Each entry has one field value, and one block address.

If the first level has r_1 entries, and the blocking factor (which is also the fan out) for the index is bfr_i = fo, then the first level needs $\lceil r_1/fo \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index. If necessary, this process can be repeated at the second level. The third level, which is an index for the second level, has an entry for each second level block, so the number of third level entries is $r_3 = \lceil r_2/fo \rceil$. We only require a second level only if the first level needs more than one block of disk storage, and we require a third level only if the second level requires more than one block as well. This process can be repeated until all the index entries at some level *t* fit in a single block.

The block at the t^{th} level is the top-level index. Each level reduces the number of entries from the previous level by a factor of fo (the index fan out or blocking factor of the index). A multilevel index with r1 first level entries will have approximately t levels, where:

$t = \lceil \log_{fo}(r_1) \rceil$.

The multilevel indices can be used on any type of index, primary, clustering, or secondary, as long as the first level index has distinct values for K(i), and fixed length entries.



Example of a two level Primary Index

Example Question

Assume the dense secondary index from example 2 is converted into a multilevel index. The index blocking factor is bfr_i = 68 index entries per block. This value is also the fan out for the multilevel index. The number of first level blocks, b_1 was calculated as 442.
How many second level blocks?

$b_2 = \lceil b_1/fo \rceil = \lceil 442/68 \rceil = 7$ blocks.

How many third level blocks?

$b_3 = \lceil b_2/fo \rceil = \lceil 7/68 \rceil = 1$ block.

Because all of the index entries at the third level can fit into one block, the third level is the top level of the index, and t = 3. (Remember: t is the number of levels required)

To access a record by searching the multilevel index, we must access one block at each level, plus one block from the data file, so we need t + 1 = 3 + 1 = 4 block accesses. In example 2, using a single level index with a binary search, we required 10 block accesses.

In this example, a dense secondary index was used, meaning there was an index entry for each record in the table. You could also have a multilevel primary index which is non-dense (for example if the first level is a primary index).

In this case, we must access the data block from the file before we can determine whether the record being searched for is in the file. With a dense index, this can be determined by accessing the first index level, without having to access the data file, since there is an entry in the index for each record in the file.

Using multi-level indices, we can reduce the number of block accesses required to search for a record given its indexing field value. Insertions and deletions are still a problem because all the index levels are physically ordered files.

ISAM (Indexed Sequential Access Method) is a file management system developed at IBM that allows records to be accessed either sequentially (in the order they were entered) or randomly (with an index). Each index defines a different ordering of the records. An employee database may have several indices, based on the information being sought. For example, a name index may order employees alphabetically by last name, while a department index may order employees by their department. A key is specified in each index. For an alphabetical index of employee names, the last name field would be the key.

ISAM was developed prior to VSAM (Virtual Storage Access Method) and relational databases.

A tree

A widely used abstract data type (ADT) or data structure implementing this ADT is the tree data structure that simulates a hierarchical tree, with a root value and sub trees of children, represented as a set of linked nodes. You will have encountered this under abstract data types in the *Sorting and Searching* unit. As you learned previously, a tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

Alternatively, a tree can be defined abstractly as a whole (globally) as an ordered

tree, with a value assigned to each node. Both these perspectives are useful: while. a tree can be analysed mathematically as a whole, when actually represented as a data structure it is usually represented and worked with separately by node (rather than as a list of nodes and an adjacency list of edges between nodes, as one may represent a digraph, for instance). For example, looking at a tree as a whole, one can talk about "the parent node" of a given node, but in general as a data structure a given node only contains the list of its children, but does not contain a reference to its parent (if any).

2.2 ADVANCED RELATIONAL DATABASE CONCEPTS

DATA REDUNDANCY

When the same data is stored in different places for no reason, this is called data redundancy. This could cause issues when the same versions of data are not updated consistently. This could cause different versions of the same information to be kept in a database, which could lead to misinformation when trying to retrieve the most recent data. This is known as data inconsistency. Redundancy could also negatively impact the security of the data because having multiple versions of something increases the risk of someone being able to access it without authorisation.

Redundancies can also lead to abnormalities in the data, known as anomalies. **Data anomalies** can occur for when changes to the data are unsuccessful. For example:

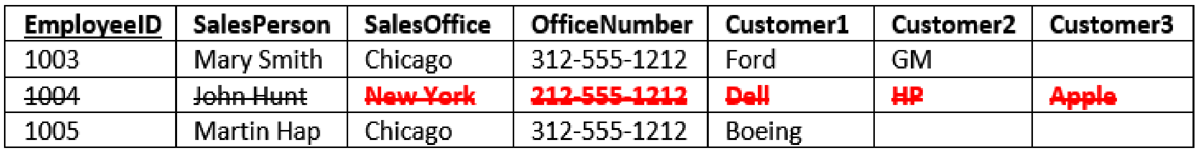
- *Update anomalies:* Occur when the same information is recorded in multiple rows. For example, in an Employee table, if the office number changes, then there are multiple updates that need to be made. If these updates are not successfully completed across all rows, then an inconsistency occurs.

EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	**312-555-1212**	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	**312-555-1212**	Boeing		

- *Insertion anomalies:* When there is data we cannot record until we know information for the entire row. For example, we cannot record a new sales office until we also know the salesperson because, in order to create the record, we need to provide a primary key. In our case, this is the EmployeeID.

EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		
???	**???**	**Atlanta**	**312-555-1212**			

- *Deletion anomalies:* When deletion of a row can cause more than one set of facts to be removed. For example, if John Hunt retires, deleting that row causes us to lose information about the New York office.



EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
~~1004~~	~~John Hunt~~	~~New York~~	~~212-555-1212~~	~~Dell~~	~~HP~~	~~Apple~~
1005	Martin Hap	Chicago	312-555-1212	Boeing		

NORMALISATION

The aim of normalisation is to correct the tables in a relational database to reduce anomalies by eliminating redundancies. To achieve normalisation, we work through up to six stages called normal forms: 1NF (first normal form), 2NF, 3NF, etc. up to 6NF. However, it is rarely necessary to go through all six stages. It is most common to go through the first three (1NF to 3NF). Therefore, in the example below, we will be working through the first three stages. By working through these stages, each version of the tables becomes structurally better than the previous stage.

By the end of normalisation, the final tables should show the following:

- Each table represents a single subject. For example, an Employee table will only contain data that directly pertains to employees.
- No data is unnecessarily stored in more than one table.
- All attributes in a table are dependent on a primary key.

Conversion to First Normal Form (1NF)

Normalisation starts with a simple three-step procedure.

Step 1: Eliminate the repeating groups
To start, the data must be formatted to a table. It is important here that there are no repeating groups of data. Each cell must contain a maximum of one value.

Table name: MUSIC

SONG_NAME	SONG_ARTIST	SONG_ALBUM	SONG_GENRE	REC_LBL	SONG_PRDR
Hello	Adele	25	Soul	XL Recordings	Greg Kurstin
Sunflower	Post Malone	Spiderman: Into the Spider-Verse	Pop	Republic	Carter Lang
Wow.	Post Malone	Hollywood's Bleeding	Hip-Hop	Republic	Louis Bel
One	Ed Sheeran	x	Folk-pop	Asylum	Jake Gosling

Bad Guy	Billie Eilish	When We Fall Asleep, Where Do We Go?	Alternative	Dark Room	Finneas O'Connell
One	Metallica	...And Justice for All	Metal	One on One Studios	Metallica
Hello	Eminem	Relapse	Hip-Hop	Aftermath Shady Interscope	Dr Dre
7 Rings	Ariana Grande	Thank U, Next	Pop	Republic	Tommy Brown
Thank U, Next					

Take a look at the table above. Note that some songs could reference more than one data entry. For example, a song named "Hello" could refer to one of two songs, each with a different artist. To make sure there is no repeated data, we need to fill in the blanks to make sure there are no null values in the table. Looking at the last row, the empty cells contain the same information as the row above because the two songs come from the same album. Therefore, the filled-in table will look like this:

Table name: 1NF_MUSIC

SONG_NAME	SONG_ARTIST	SONG_ALBUM	SONG_GENRE	REC_LBL	SONG_PRDR
Hello	Adele	25	Soul	XL Recordings	Greg Kurstin
Sunflower	Post Malone	Spiderman: Into the Spider-Verse	Pop	Republic	Carter Lang
Wow.	Post Malone	Hollywood's Bleeding	Hip-hop	Republic	Louis Bel
One	Ed Sheeran	x	Folk-pop	Asylum	Jake Gosling
Bad Guy	Billie Eilish	When We Fall Asleep, Where Do We Go?	Alternative	Dark Room	Finneas O'Connell
One	Metallica	...And Justice for All	Metal	One on One Studios	Metallica
Hello	Eminem	Relapse	Hip-Hop	Aftermath	Dr. Dre

				Shady Interscope	
7 Rings	Ariana Grande	Thank U, Next	Pop	Republic	Tommy Brown
Thank U, Next	Ariana Grande	Thank U, Next	Pop	Republic	Tommy Brown

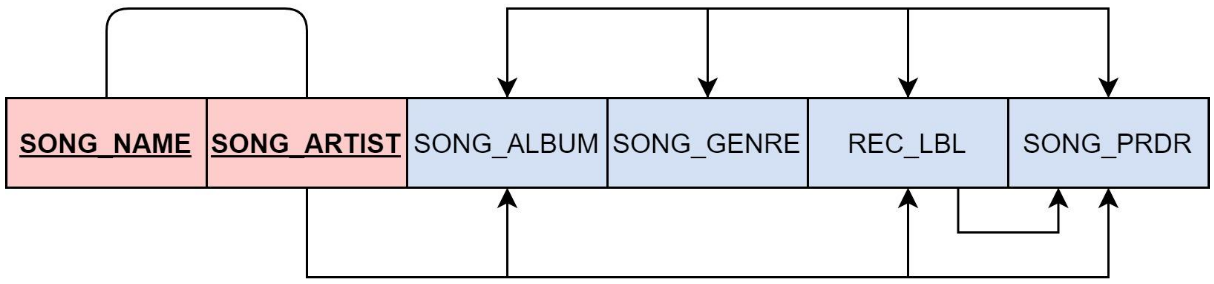
Step 2: Identify the primary key
Next, we need to identify the primary key. The primary key needs to be able to identify one row of the table, thereby identifying all the remaining rows, or attributes, when needed. For example, SONG_NAME is not a good primary key because "One" and "Hello" each identify 2 different rows. Similarly, SONG_ARTIST would not be a good primary key because "Ariana Grande" and "Post Malone" each identify 2 rows. Therefore, we could use a combination of SONG_NAME and SONG_ARTIST to be able to narrow down our search to a single row. Now, if you know that SONG_NAME = "Hello and SONG_ARTIST = "Adele", the entries for the attributes can only be "25", "Soul", "XL Recordings", and "Greg Kurstin"

Step 3: Identify all dependencies
The example above, step 1 shows us the following relationship:

<u>SONG_NAME</u>, <u>ARTIST_NAME</u> → SONG_ALBUM, SONG_GENRE, REC_LBL, SONG_PRDR

This means that SONG_ALBUM, SONG_GENRE, REC_LBL and SONG_PRDR are dependent on the combination of <u>SONG_NAME</u> and <u>ARTIST_NAME</u>. However, looking at the table, we could also see that the song artist determines the album name, the record label and the record producer (<u>SONG_ARTIST</u> → SONG_ALBUM, REC_LBL, SONG_PRDR). These are known as partial dependencies because only one of the two primary keys is needed to determine the other attributes.

Finally, knowing the record label means knowing the song's producer (REC_LBL → SONG_PRDR). This is known as a partial dependency because it is not based on a primary attribute. For a visual representation of this, look at the dependency diagram below:



Note that:

1. The primary key attributes are underlined and shaded in a different colour.

2. The arrows above the attributes indicate all desirable dependencies. Desirable dependencies are those based on the primary key. Note that the entity's attributes are dependent on the combination of SONG_NAME and ARTIST_NAME.

3. The arrows below the diagram indicate partial dependencies and transitive dependencies. These are less desirable than those based on the primary key

We have now put the table in 1NF because:
- All of the key attributes are defined
- There are no repeating groups in the table
- All attributes are dependent on the primary key

Conversation to Second Normal Form (2NF)

The relational database design can be improved by converting the database into a format known as the second normal form. Here, we try to rid the tables of partial dependencies, as they can cause anomalies.

Step 1: Write each key component on a separate line
Write each key component on a separate line, then write the original (composite) key on the last line:

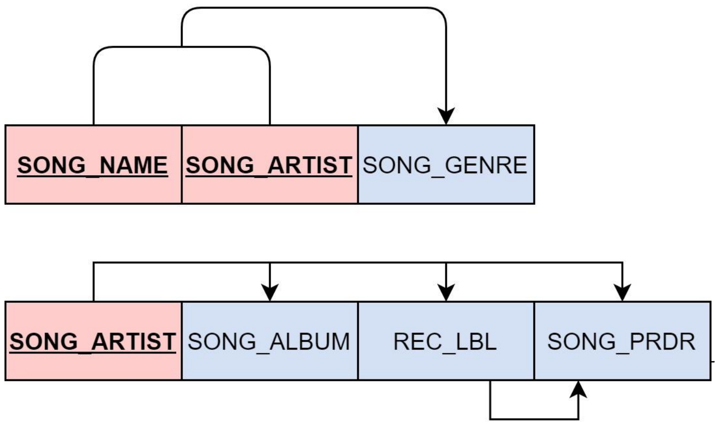
SONG_NAME
SONG_ARTIST
SONG_NAME SONG_ARTIST

You may have noticed from the diagram above that SONG_ARTIST is not a primary key on its own for any attribute, therefore giving it its own new table isn't necessary. Therefore, the keys that will have their own tables are SONG_NAME and SONG_NAME SONG_ARTIST. We will name these two tables ARTIST and GENRE (because the table is used to determine the genre) respectively.

Step 2: Assign corresponding dependent attributes
Then use the dependency diagram above to determine those attributes that are dependent on other attributes by looking at the arrows underneath the diagram. With this, we create three new tables:

ARTIST (SONG_ARTIST, SONG_ALBUM, REC_LBL, SONG_PRDR)
GENRE (SONG_NAME, SONG_ARTIST, SONG_GENRE)

The dependency diagram below shows the result of Steps 1 and 2.



Note how there are no longer any partial dependencies that could cause anomalies in the tables. However, there is still transitive dependency in the Employee table.

We have now put the table into 2NF because:

- It is in 1NF
- It includes no partial dependencies because no attribute in each table is dependent on only a part of a primary key

Conversation to Third Normal Form (3NF)

We will now eliminate the transitive dependencies.

Step 1: Identify each new determinant
Write the determinant of each transitive dependency as a primary key in its own table. In this example, the determinant will be REC_LBL.

Step 2: Identify the dependent attributes
As mentioned, SONG_PRDR is dependent on REC_LBL (REC_LBL → SONG_PRDR). Therefore, we can name the new table Label.

Step 3: Remove the dependent attributes from transitive dependencies
Eliminate all dependent attributes in the transitive relationship from each of the tables that have such a relationship. In this example, we eliminate SONG_PRDR from the ARTIST table shown in the dependency diagram above to leave the ARTIST table dependency definition as SONG_ARTIST → SONG_ALBUM, REC_LBL.

Notice that the REC_LBL remains in the ARTISTtable to serve as the foreign key. You can now draw a new dependency diagram to show all of the tables you have defined in the steps above. Then check the new tables as well as the tables you modified in Step 3 to make sure that each table has a determinant and that the tables don't contain inappropriate dependencies. The new dependency diagram should look as follows:

Table name: GENRE

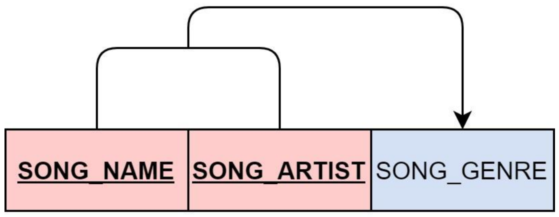


Table name: ARTIST

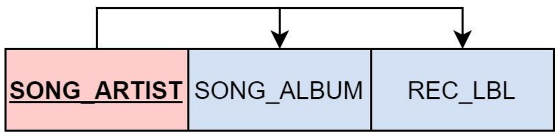
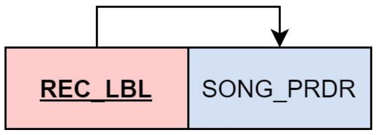


Table name: LABEL



This conversation has eliminated the original ARTIST table's transitive dependency. The tables are now said to be in third normal form (3NF).

We have now put the table into 3NF because:

- It is in 2NF
- It contains no transitive dependencies

Compulsory Task

Answer the following questions:

- Define a database and the three database language types covered in this lesson (8)
- Using the INVOICE table given below, draw its dependency diagram and identify all dependencies (including transitive and partial dependencies). You can assume that the table does not contain any repeating groups and that an invoice number references more than one product.
  Hint: This table uses a composite primary key

INV_NUM	PROD_NUM	SALE_DATE	PROD_LABEL	VEND_CODE	VEND_NAME	QUANT_SOLD	PROD_PRICE
211347	AX-P126VBB	15-Jun-2018	Lawnmower	111	Mowers and Bulbs	1	R4099.90
211347	GF-5106D2F	15-Jun-2018	Lightbulb	111	Mowers and Bulbs	8	R24,99
211347	RB-163698G	15-Jun-2018	Blue paint	063	All Things Paint	1	R80,90
211348	AX-P126VBB	15-Jun-2018	Lawnmower	111	Mowers and Bulbs	2	R4099.90
211349	TF-7465023Q	17-Jun-2018	2m ladder	207	Tall Things	1	R699,99

- Draw new dependency diagrams to show the data in 2NF.
- Draw new dependency diagrams to show the data in 3NF.
- Use this table to illustrate one of each kind of anomaly described in the task. In a text file called anomalies.txt, how you would change the table and which anomaly that change would create.

Self-assessment table

Intended learning outcome (SAQA US 11048)	Assess your own proficiency level for each intended learning outcome. Check each box when the statement becomes true- I can successfully:				
Review the requirements for database access for a computer application using SQL (SO1)	Understand and explain Abstraction	Understand and explain Data independence (physical and logical)	Understand and explain Data models	Understand and explain DDLs and DMLs	
Design database access for a computer application using SQL (SO2)	Understand and use simple indices	Understand and use multi-level indices	Understand index-sequential files	Understand tree structures	
Intended learning outcomes (Hyperion extension)	**Assess your own proficiency level for each intended learning outcome. Check each box when the statement becomes true- I can successfully:**				
Relational database concepts	Understand and explain Tables	Understand and explain Relationships between Tables	Understand and explain Data Redundancy	Understand and explain Data anomalies	Understand and explain the Primary and foreign keys and their roles in relating Tables
Normalisation	Define normalisation	Convert to 1NF	Convert to 2NF	Convert to 3NF	

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this lesson, task, or this course as a whole, can be improved, or think we’ve done a good job?

Click here to share your thoughts anonymously.

REFERENCES

Coronel, C., Morris, S., & Rob, P. (2011). *Database systems* (9th ed., pp. 4-26). Boston: Cengage Learning.

Coronel, C., Morris, S., & Rob, P. (2011). *Database systems* (9th ed., pp. 174-218). Boston: Cengage Learning.

Bella Visage Incorporated. (2021). 48872 National Certificate: Information Technology (Systems Development). Learner Guide.