NAPIER UNIVERSITY
EDINBURGH

Database Systems

Student Notes

CO22001/CO72010

Version 3.2

SCHOOL OF COMPUTING

Database eLearning

This site is focused on online database learning, revolving around an electronic textbook. Chapter links to the textbook are as follows:

Other Sites: Linux Admin Tutorials

Chapter 1 - Introduction

So you want to learn about databases? This document is a good starting point, and is used at University level to teach computing students. Use it in conjunction with the online resources, like the quiz and the SQL tutorial environment.

This Document

Welcome to Database Resources. I have been authoring online material to teach databases for many years, and this site is the latest attempt to bring online database learning to the public.

The site revolves around this electronic document, which contains all the theory required to pass a introduction to databases module at University level. The site also has practical exercises, including an online quiz for testing your knowledge, actual University exam papers for you to formally test yourself and gain exam experience, and an online SQL environment for executing your own SQL from the safety of your own home.

For reference, the version of the notes you see on this site are based on v3.1 of the printed notes.

The SQL environment gives you access to an Oracle database, and allows you to write your own SQL and execute it as part of an extensive practical exercise. Free registration allows you to revisit the tutorials and continue on from where you left off.

There are additional resources for those involved in teaching databases, including a downloadable version of the notes and powerpoint slides for formal lectures.

Usage

This document is for use with a variety of University courses running throughout the world. The document forms a good introduction to the basics of database systems for university students. At Napier University the modules which use this material include:

- CO22001 – Database Systems. This is a 2nd year module for computing students.
- CS22010 – Database Systems 2. This is the old name for CO22001.
- CO72010 – Database Systems. This is a postgraduate module taught on some of our postgraduate conversion courses.

You are free to make use of this site for personal learning purposes only. To make use of the material found here for financial gain you must gain written permission from myself. Suggestions and corrections welcomed.

Dr Gordon Russell (g.russell@napier.ac.uk)

Acknowledgments:

Andrew Cumming
John Old

Introduction

Contents

Relational database systems have become increasingly popular since the late 1970's. They offer a powerful method for storing data in an application-independent manner. This means that for many enterprises the database is at the core of the I.T. strategy. Developments can progress around a relatively stable database structure which is secure, reliable, efficient, and transparent.

In early systems, each suite of application programs had its own independent master file. The duplication of data over master files could lead to inconsistent data.

Efforts to use a common master file for a number of application programs resulted in problems of integrity and security. The production of new application programs could require amendments to existing application programs, resulting in `unproductive maintenance'.

Data structuring techniques, developed to exploit random access storage devices, increased the complexity of the insert, delete and update operations on data. As a first step towards a DBMS, packages of subroutines were introduced to reduce programmer effort in maintaining these data structures. However, the use of these packages still requires knowledge of the physical organization of the data.

The Database Approach

A database system is a computer-based system to record and maintain information. The information concerned can be anything of significance to the organisation for whose use it is intended.

The contents of a database can hold a variety of different things. To make database design more straight-forward, databases contents are divided up into two concepts:

- Schema
- Data

The Schema is the structure of data, whereas the Data are the "facts". Schema can be complex to understand to begin with, but really indicates the rules which the Data must obey.

Imagine a case where we want to store facts about employees in a company. Such facts could include their name, address, date of birth, and salary. In a database all the information on all employees would be held in a single storage "container", called a *table*. This table is a tabular object like a spreadsheet page, with different employees as the rows, and the facts (e.g. their names) as columns... Let's call this table EMP, and it could look something like:

Name	Address	Date of Birth	Salary
Jim Smith	1 Apple Lane	1/3/1991	11000
Jon Greg	5 Pear St	7/9/1992	13000
Bob Roberts	2 Plumb Road	3/2/1990	12000

From this information the *schema* would define that EMP has four components, "NAME","ADDRESS","DOB","SALARY". As designers we can call the columns what we like, but making them meaningful helps. In addition to the name, we want to try and make sure that people dont accidentally store a name in the DOB column, or some other silly error. Protecting the database against rubbish data is one of the most important database design steps, and is what much of this course is about. From what we know about the facts, we can say things like:

- NAME is a string, and needs to hold at least 12 characters.
- ADDRESS is a string, and needs to hold at least 12 characters.
- DOB is a date... The company forbids people over 100 years old or younger than 18 years old working for them.
- SALARY is a number. It must be greater than zero.

Such rules can be enforced by a database. During the design phase of a database schema these and more complex rules are identified and where possible implemented. The more rules the harder it is to enter poor quality data.

User Types

When considering users of a Database system, there are three broad classes to consider:

1. the application programmer, responsible for writing programs in some high-level language such as COBOL, C++, etc.
2. the end-user, who accesses the database via a query language
3. the database administrator (DBA), who controls all operations on the database

Database Architecture

DBMSs do not all conform to the same architecture.

- The three-level architecture forms the basis of modern database architectures.
- this is in agreement with the ANSI/SPARC study group on Database Management Systems.
- ANSI/SPARC is the American National Standards Institute/Standard Planning and Requirement Committee).
- The architecture for DBMSs is divided into three general levels:
- external
- conceptual
- internal

Three level database architecture

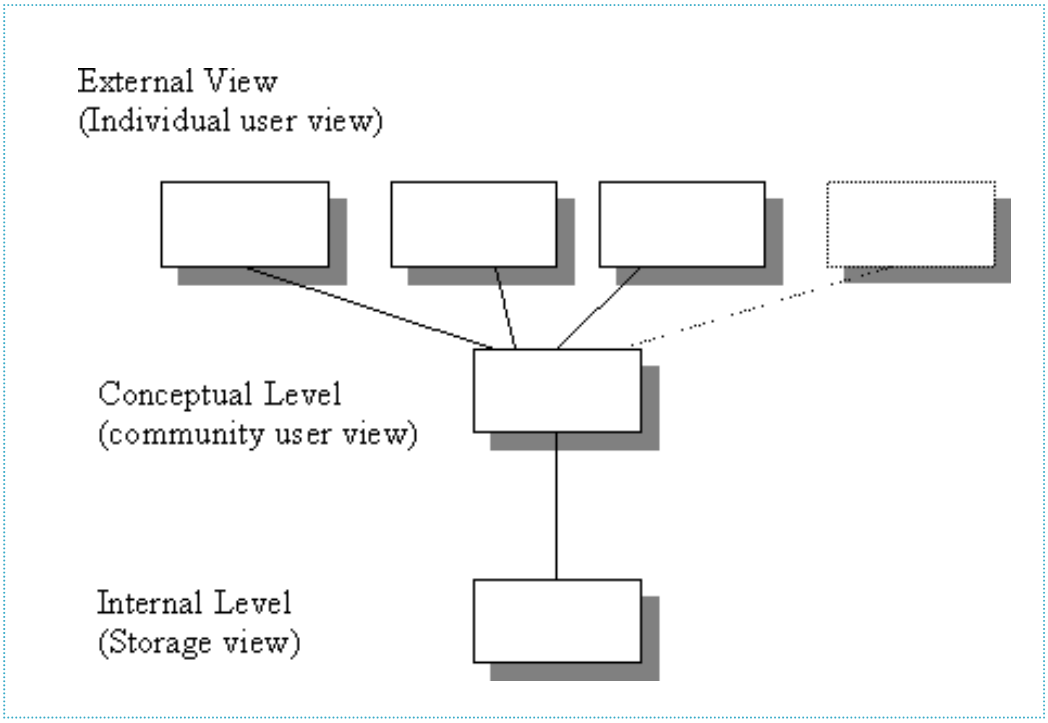


Figure 1: Three level architecture

1. the external level : concerned with the way individual users see the data
2. the conceptual level : can be regarded as a community user view a formal description of data of interest to the organisation, independent of any storage considerations.
3. the internal level : concerned with the way in which the data is actually stored

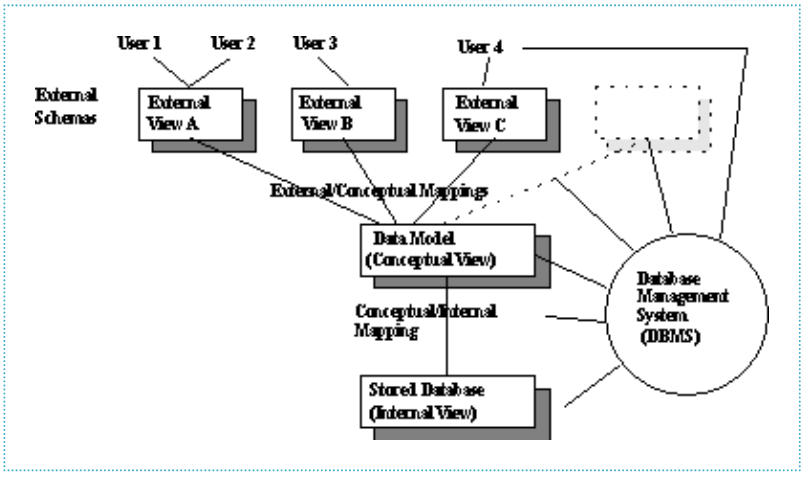


Figure : How the three level architecture works

External View

A user is anyone who needs to access some portion of the data. They may range from application programmers to casual users with adhoc queries. Each user has a language at his/her disposal.

The application programmer may use a high level language (e.g. COBOL) while the casual user will probably use a query language.

Regardless of the language used, it will include a data sublanguage DSL which is that subset of the language which is concerned with storage and retrieval of information in the database and may or

may not be apparent to the user.

A DSL is a combination of two languages:

- a data definition language (DDL) - provides for the definition or description of database objects
- a data manipulation language (DML) - supports the manipulation or processing of database objects.

Each user sees the data in terms of an external view: Defined by an external schema, consisting basically of descriptions of each of the various types of external record in that external view, and also a definition of the mapping between the external schema and the underlying conceptual schema.

Conceptual View

- An abstract representation of the entire information content of the database.
- It is in general a view of the data as it actually is, that is, it is a `model' of the `realworld'.
- It consists of multiple occurrences of multiple types of conceptual record, defined in the conceptual schema.
- To achieve data independence, the definitions of conceptual records must involve information content only.
- storage structure is ignored
- access strategy is ignored
- In addition to definitions, the conceptual schema contains authorisation and validation procedures.

Internal View

The internal view is a low-level representation of the entire database consisting of multiple occurrences of multiple types of internal (stored) records.

It is however at one remove from the physical level since it does not deal in terms of physical records or blocks nor with any device specific constraints such as cylinder or track sizes. Details of mapping to physical storage is highly implementation specific and are not expressed in the three-level architecture.

The internal view described by the internal schema:

- defines the various types of stored record
- what indices exist
- how stored fields are represented
- what physical sequence the stored records are in

In effect the internal schema is the storage structure definition.

Mappings

- The conceptual/internal mapping:
    - defines conceptual and internal view correspondence
    - specifies mapping from conceptual records to their stored counterparts
- An external/conceptual mapping:
    - defines a particular external and conceptual view correspondence
- A change to the storage structure definition means that the conceptual/internal mapping must

be changed accordingly, so that the conceptual schema may remain invariant, achieving physical data independence.
- A change to the conceptual definition means that the conceptual/external mapping must be changed accordingly, so that the external schema may remain invariant, achieving logical data independence.

DBMS

The database management system (DBMS) is the software that:

- handles all access to the database
- is responsible for applying the authorisation checks and validation procedures

Conceptually what happens is:

1. A user issues an access request, using some particular DML.
2. The DBMS intercepts the request and interprets it.
3. The DBMS inspects in turn the external schema, the external/conceptual mapping, the conceptual schema, the conceptual internal mapping, and the storage structure definition.
4. The DBMS performs the necessary operations on the stored database.

Database Administrator

The database administrator (DBA) is the person (or group of people) responsible for overall control of the database system. The DBA's responsibilities include the following:

- deciding the information content of the database, i.e. identifying the entities of interest to the enterprise and the information to be recorded about those entities. This is defined by writing the conceptual schema using the DDL
- deciding the storage structure and access strategy, i.e. how the data is to be represented by writing the storage structure definition. The associated internal/conceptual schema must also be specified using the DDL
- liaising with users, i.e. to ensure that the data they require is available and to write the necessary external schemas and conceptual/external mapping (again using DDL)
- defining authorisation checks and validation procedures. Authorisation checks and validation procedures are extensions to the conceptual schema and can be specified using the DDL
- defining a strategy for backup and recovery. For example periodic dumping of the database to a backup tape and procedures for reloading the database for backup. Use of a log file where each log record contains the values for database items before and after a change and can be used for recovery purposes
- monitoring performance and responding to changes in requirements, i.e. changing details of storage and access thereby organising the system so as to get the performance that is `best for the enterprise'

Facilities and Limitations

The facilities offered by DBMS vary a great deal, depending on their level of sophistication. In general, however, a good DBMS should provide the following advantages over a conventional system:

- Independence of data and program - This is a prime advantage of a database. Both the database and the user program can be altered independently of each other thus saving time and

money which would be required to retain consistency.
- Data shareability and nonredundance of data - The ideal situation is to enable applications to share an integrated database containing all the data needed by the applications and thus eliminate as much as possible the need to store data redundantly.
- Integrity - With many different users sharing various portions of the database, it is impossible for each user to be responsible for the consistency of the values in the database and for maintaining the relationships of the user data items to all other data item, some of which may be unknown or even prohibited for the user to access.
- Centralised control - With central control of the database, the DBA can ensure that standards are followed in the representation of data.
- Security - Having control over the database the DBA can ensure that access to the database is through proper channels and can define the access rights of any user to any data items or defined subset of the database. The security system must prevent corruption of the existing data either accidently or maliciously.
- Performance and Efficiency - In view of the size of databases and of demanding database accessing requirements, good performance and efficiency are major requirements. Knowing the overall requirements of the organisation, as opposed to the requirements of any individual user, the DBA can structure the database system to provide an overall service that is `best for the enterprise'.

Data Independence

- This is a prime advantage of a database. Both the database and the user program can be altered independently of each other.
- In a conventional system applications are datadependent. This means that the way in which the data is organised in secondary storage and the way in which it is accessed are both dictated by the requirements of the application, and, moreover, that knowledge of the data organisation and access technique is built into the application logic.
- For example, if a file is stored in indexed sequential form then an application must know
    - that the index exists
    - the file sequence (as defined by the index)

The internal structure of the application will be built around this knowledge. If, for example, the file was to be replaced by a hash-addressed file, major modifications would have to be made to the application.

Such an application is data-dependent - it is impossible to change the storage structure (how the data is physically recorded) or the access strategy (how it is accessed) without affecting the application, probably drastically. The portions of the application requiring alteration are those that communicate with the file handling software - the difficulties involved are quite irrelevant to the problem the application was written to solve.

- it is undesirable to allow applications to be data-dependent - different applications will need different views of the same data.
- the DBA must have the freedom to change storage structure or access strategy in response to changing requirements without having to modify existing applications.
- Data independence can be defines as
  `The immunity of applications to change in storage structure and access strategy'.

Data Redundancy

In non-database systems each application has its own private files. This can often lead to redundancy in stored data, with resultant waste in storage space. In a database the data is integrated.

The database may be thought of as a unification of several otherwise distinct data files, with any redundancy among those files partially or wholly eliminated.

Data integration is generally regarded as an important characteristic of a database. The avoidance of redundancy should be an aim, however, the vigour with which this aim should be pursued is open to question.

Redundancy is

- direct if a value is a copy of another
- indirect if the value can be derived from other values:
    - simplifies retrieval but complicates update
    - conversely integration makes retrieval slow and updates easier
- Data redundancy can lead to inconsistency in the database unless controlled.
- the system should be aware of any data duplication - the system is responsible for ensuring updates are carried out correctly.
- a DB with uncontrolled redundancy can be in an inconsistent state - it can supply incorrect or conflicting information
- a given fact represented by a single entry cannot result in inconsistency - few systems are capable of propagating updates i.e. most systems do not support controlled redundancy.

Data Integrity

This describes the problem of ensuring that the data in the database is accurate...

- inconsistencies between two entries representing the same `fact' give an example of lack of integrity (caused by redundancy in the database).
- integrity constraints can be viewed as a set of assertions to be obeyed when updating a DB to preserve an error-free state.
- even if redundancy is eliminated, the DB may still contain incorrect data.
- integrity checks which are important are checks on data items and record types.

Integrity checks on data items can be divided into 4 groups:

1. type checks
    - e.g. ensuring a numeric field is numeric and not a character - this check should be performed automatically by the DBMS.
2. redundancy checks
    - direct or indirect (see data redundancy) - this check is not automatic in most cases.
3. range checks
    - e.g. to ensure a data item value falls within a specified range of values, such as checking dates so that say (age > 0 AND age < 110).
4. comparison checks
    - in this check a function of a set of data item values is compared against a function of another set of data item values. For example, the max salary for a given set of employees must be less than the min salary for the set of employees on a higher salary scale.

A record type may have constraints on the total number of occurrences, or on the insertions and deletions of records. For example in a patient database there may be a limit on the number of xray results for each patient or the details of a patients visit to hospital must be kept for a minimum of 5 years before it can be deleted

- Centralized control of the database helps maintain integrity, and permits the DBA to define

validation procedures to be carried out whenever any update operation is attempted (update covers modification, creation and deletion).

- Integrity is important in a database system - an application run without validation procedures can produce erroneous data which can then affect other applications using that data.

Chapter 2 - Database Analysis

Basic database analysis techniques, Entity Relationship modelling, and mapping ER diagrams to relations.

Database Analysis

Contents

This unit it concerned with the process of taking a database specification from a customer and implementing the underlying database structure necessary to support that specification.

Introduction

Data analysis is concerned with the NATURE and USE of data. It involves the identification of the data elements which are needed to support the data processing system of the organization, the placing of these elements into logical groups and the definition of the relationships between the resulting groups.

Other approaches, e.g. D.F.Ds and Flowcharts, have been concerned with the flow of data-dataflow methodologies. Data analysis is one of several data structure based methodologies Jackson SP/D is another.

Systems analysts often, in practice, go directly from fact finding to implementation dependent data analysis. Their assumptions about the usage of properties of and relationships between data elements are embodied directly in record and file designs and computer procedure specifications. The introduction of Database Management Systems (DBMS) has encouraged a higher level of analysis, where the data elements are defined by a logical model or `schema' (conceptual schema). When discussing the schema in the context of a DBMS, the effects of alternative designs on the efficiency or ease of implementation is considered, i.e. the analysis is still somewhat implementation dependent. If we consider the data relationships, usages and properties that are important to the business without regard to their representation in a particular computerised system using particular software, we have what we are concerned with, implementationindependent data analysis.

It is fair to ask why data analysis should be done if it is possible, in practice to go straight to a computerised system design. Data analysis is time consuming; it throws up a lot of questions. Implementation may be slowed down while the answers are sought. It is more expedient to have an experienced analyst `get on with the job' and come up with a design straight away. The main difference is that data analysis is more likely to result in a design which meets both present and future requirements, being more easily adapted to changes in the business or in the computing equipment. It can also be argued that it tends to ensure that policy questions concerning the organisations' data are answered by the managers of the organisation, not by the systems analysts. Data analysis may be thought of as the `slow and careful' approach, whereas omitting this step is `quick and dirty'.

From another viewpoint, data analysis provides useful insights for general design principals which will benefit the trainee analyst even if he finally settles for a `quick and dirty' solution.

The development of techniques of data analysis have helped to understand the structure and meaning of data in organisations. Data analysis techniques can be used as the first step of extrapolating the complexities of the real world into a model that can be held on a computer and be accessed by many users. The data can be gathered by conventional methods such as interviewing people in the organisation and studying documents. The facts can be represented as objects of interest. There are a number of documentation tools available for data analysis, such as entityrelationship diagrams. These are useful aids to communication, help to ensure that the work is carried out in a thorough manner, and ease the mapping processes that follow data analysis. Some of the documents can be used as source documents for the data dictionary.

In data analysis we analyse the data and build a systems representation in the form of a data model (conceptual). A conceptual data model specifies the structure of the data and the processes which use that data.

Data Analysis = establishing the nature of data.

Functional Analysis = establishing the use of data.

However, since Data and Functional Analysis are so intermixed, we shall use the term Data Analysis to cover both.

Building a model of an organisation is not easy. The whole organisation is too large as there will be too many things to be modelled. It takes too long and does not achieve anything concrete like an information system, and managers want tangible results fairly quickly. It is therefore the task of the data analyst to model a particular view of the organisation, one which proves reasonable and accurate for most applications and uses. Data has an intrinsic structure of its own, independent of processing, reports formats etc. The data model seeks to make explicit that structure

Data analysis was described as establishing the nature and use of data.

Database Analysis Life Cycle

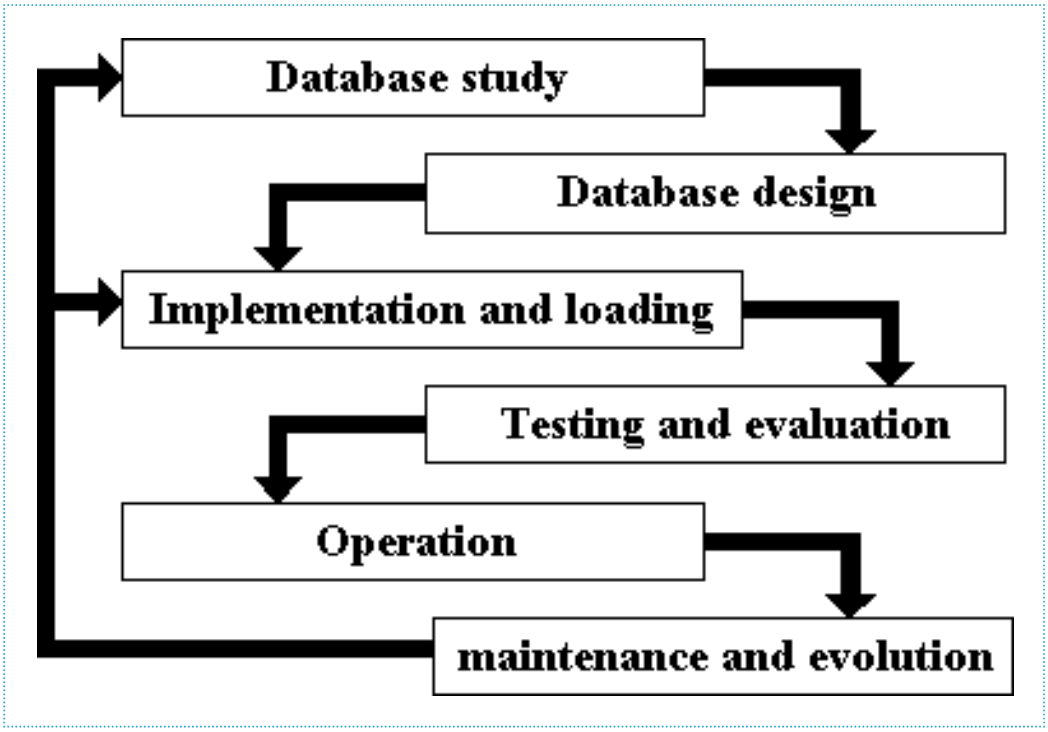


Figure : Database Analysis Life Cycle

When a database designer is approaching the problem of constructing a database system, the logical steps followed is that of the database analysis life cycle:

- **Database study** - here the designer creates a written specification in words for the database system to be built. This involves:
  - analysing the company situation - is it an expanding company, dynamic in its requirements, mature in nature, solid background in employee training for new internal products, etc. These have an impact on how the specification is to be viewed.
  - define problems and constraints - what is the situation currently? How does the company deal with the task which the new database is to perform. Any issues around the current method? What are the limits of the new system?
  - define objectives - what is the new database system going to have to do, and in what way must it be done. What information does the company want to store specifically, and what does it want to calculate. How will the data evolve.
  - define scope and boundaries - what is stored on this new database system, and what it stored elsewhere. Will it interface to another database?
- **Database Design** - conceptual, logical, and physical design steps in taking specifications to physical implementable designs. This is looked at more closely in a moment.
- **Implementation and loading** - it is quite possible that the database is to run on a machine which as yet does not have a database management system running on it at the moment. If this is the case one must be installed on that machine. Once a DBMS has been installed, the database itself must be created within the DBMS. Finally, not all databases start completely empty, and thus must be loaded with the initial data set (such as the current inventory, current staff names, current customer details, etc).
- **Testing and evaluation** - the database, once implemented, must be tested against the specification supplied by the client. It is also useful to test the database with the client using mock data, as clients do not always have a full understanding of what they thing they have specified and how it differs from what they have actually asked for! In addition, this step in the life cycle offers the chance to the designer to fine-tune the system for best performance. Finally, it is a good idea to evaluate the database in-situ, along with any linked applications.
- **Operation** - this step is where the system is actually in real usage by the company.

- **Maintenance and evolution** - designers rarely get everything perfect first time, and it may be the case that the company requests changes to fix problems with the system or to recommend enhancements or new requirements.
    - Commonly development takes place without change to the database structure. In elderly systems the DB structure becomes fossilised.

Three-level Database Model

Often referred to as the three-level model, this is where the design moves from a written specification taken from the real-world requirements to a physically-implementable design for a specific DBMS. The three levels commonly referred to are `Conceptual Design', `Data Model Mapping', and `Physical Design'.

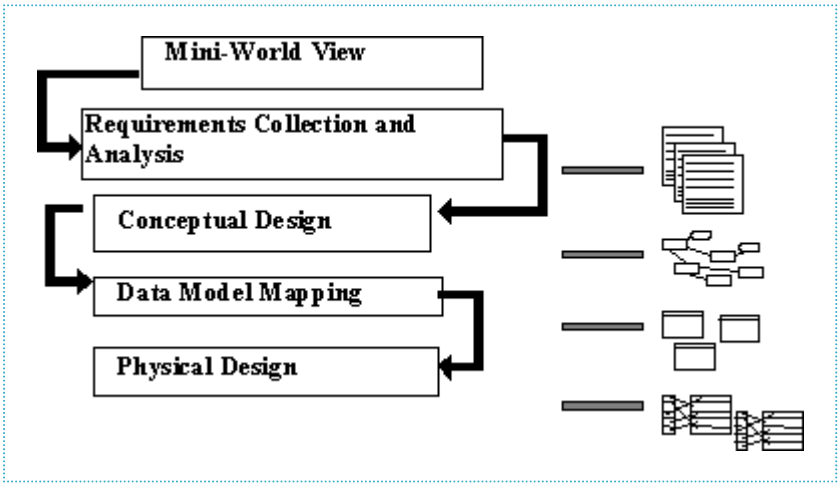


Figure : Logic behind the three level architecture

The specification is usually in the form of a written document containing customer requirements, mock reports, screen drawings and the like, written by the client to indicate the requirements which the final system is to have. Often such data has to be collected together from a variety of internal sources to the company and then analysed to see if the requirements are necessary, correct, and efficient.

Once the Database requirements have been collated, the Conceptual Design phase takes the requirements and produces a high-level data model of the database structure. In this module, we use ER modelling to represent high-level data models, but there are other techniques. This model is independent of the final DBMS which the database will be installed in.

Next, the Conceptual Design phase takes the high-level data model it taken and converted into a conceptual schema, which is specific to a particular DBMS class (e.g. relational). For a relational system, such as Oracle, an appropriate conceptual schema would be relations.

Finally, in the Physical Design phase the conceptual schema is converted into database internal structures. This is specific to a particular DBMS product.

Basics

Entity Relationship (ER) modelling

- is a design tool
- is a graphical representation of the database system

- provides a high-level conceptual data model
- supports the user's perception of the data
- is DBMS and hardware independent
- had many variants
- is composed of entities, attributes, and relationships

Entities

- An entity is any object in the system that we want to model and store information about
- Individual objects are called entities
- Groups of the same type of objects are called entity types or entity sets
- Entities are represented by rectangles (either with round or square corners)

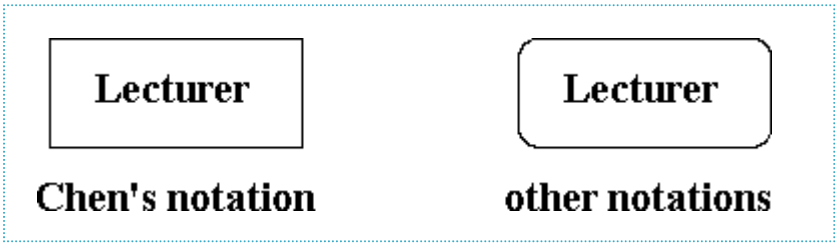


Figure: Entities

- There are two types of entities; weak and strong entity types.

Attribute

- All the data relating to an entity is held in its attributes.
- An attribute is a property of an entity.
- Each attribute can have any value from its domain.
- Each entity within an entity type:
  - May have any number of attributes.
  - Can have different attribute values than that in any other entity.
  - Have the same number of attributes.
- Attributes can be
- simple or composite
- single-valued or multi-valued
- Attributes can be shown on ER models
- They appear inside ovals and are attached to their entity.
- Note that entity types can have a large number of attributes... If all are shown then the diagrams would be confusing. Only show an attribute if it adds information to the ER diagram, or clarifies a point.

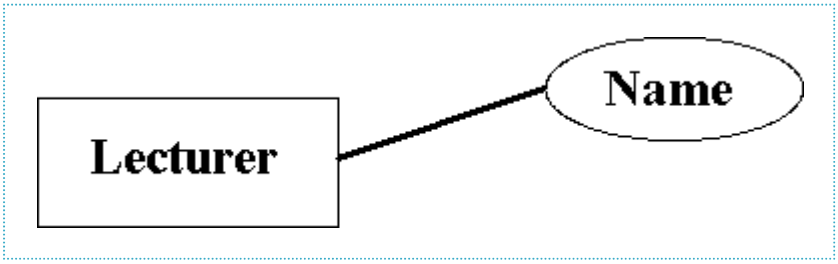


Figure : Attributes

Keys

- A key is a data item that allows us to uniquely identify individual occurrences or an entity type.
- A candidate key is an attribute or set of attributes that uniquely identifies individual occurrences or an entity type.
- An entity type may have one or more possible candidate keys, the one which is selected is known as the primary key.
- A composite key is a candidate key that consists of two or more attributes
- The name of each primary key attribute is underlined.

Relationships

- A *relationship type* is a meaningful association between entity types
- A *relationship* is an association of entities where the association includes one entity from each participating entity type.
- Relationship types are represented on the ER diagram by a series of lines.
- As always, there are many notations in use today...
- In the original Chen notation, the relationship is placed inside a diamond, e.g. managers manage employees:

Figure : Chens notation for relationships

- For this module, we will use an alternative notation, where the relationship is a label on the line. The meaning is identical

Figure : Relationships used in this document

Degree of a Relationship

- The number of participating entities in a relationship is known as the degree of the relationship.
- If there are two entity types involved it is a *binary* relationship type

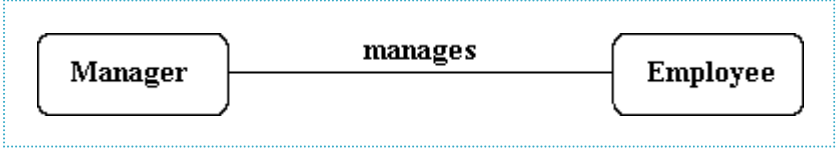


Figure : Binary Relationships

- If there are three entity types involved it is a *ternary* relationship type

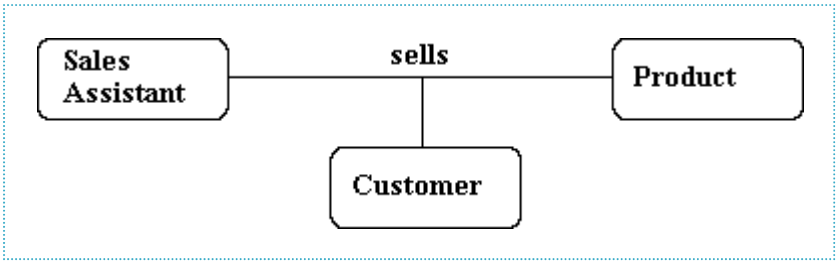


Figure : Ternary relationship

- It is possible to have a n-ary relationship (e.g. quaternary or unary).
- Unary relationships are also known as a *recursive* relationship.

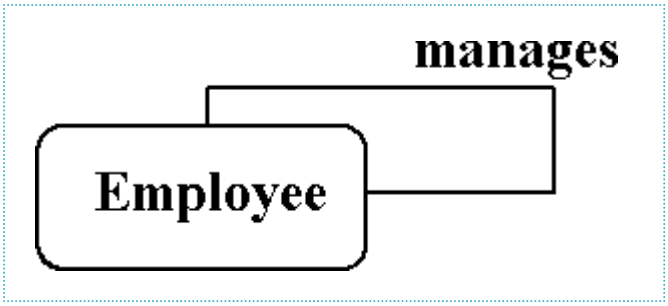


Figure : Recursive relationship

- It is a relationship where the same entity participates more than once in different roles.
- In the example above we are saying that employees are managed by employees.
- If we wanted more information about who manages whom, we could introduce a second entity type called manager.

Degree of a Relationship

- It is also possible to have entities associated through two or more distinct relationships.

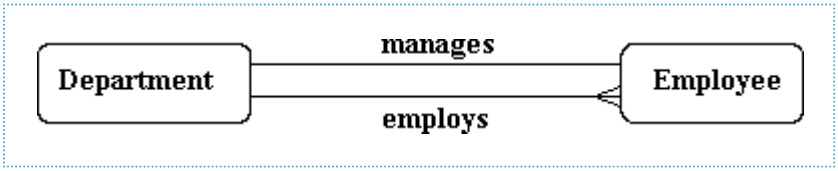


Figure : Multiple relationships

- In the representation we use it is not possible to have attributes as part of a relationship. To support this other entity types need to be developed.

Replacing ternary relationships

When ternary relationships occurs in an ER model they should always be removed before finishing the model. Sometimes the relationships can be replaced by a series of binary relationships that link pairs of the original ternary relationship.

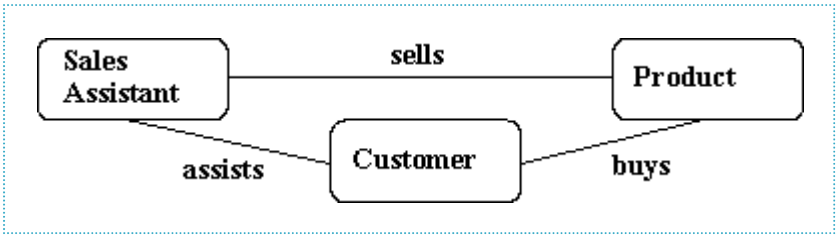


Figure : A ternary relationship example

- This can result in the loss of some information - It is no longer clear which sales assistant sold a customer a particular product.
- Try replacing the ternary relationship with an entity type and a set of binary relationships.

Relationships are usually verbs, so name the new entity type by the relationship verb rewritten as a noun.

- The relationship *sells* can become the entity type *sale*.

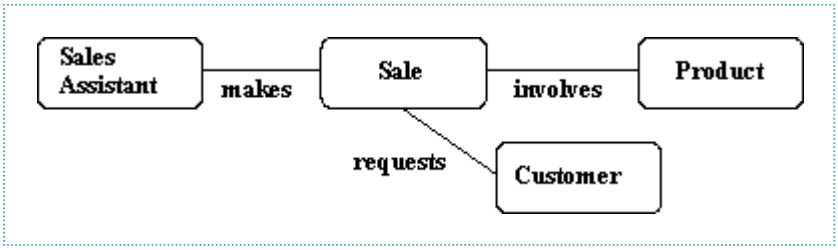


Figure : Replacing a ternary relationship

- So a sales assistant can be linked to a specific customer and both of them to the sale of a particular product.
- This process also works for higher order relationships.

Cardinality

- Relationships are rarely one-to-one
- For example, a manager usually manages more than one employee
- This is described by the *cardinality* of the relationship, for which there are four possible categories.
- One to one (1:1) relationship
- One to many (1:m) relationship
- Many to one (m:1) relationship
- Many to many (m:n) relationship
- On an ER diagram, if the end of a relationship is straight, it represents 1, while a "crow's foot" end represents many.
- A one to one relationship - a man can only marry one woman, and a woman can only marry one man, so it is a one to one (1:1) relationship

Figure : One to One relationship example

- A one to may relationship - one manager manages many employees, but each employee only has one manager, so it is a one to many (1:n) relationship

Figure : One to Many relationship example

- A many to one relationship - many students study one course. They do not study more than one course, so it is a many to one (m:1) relationship

Figure : Many to One relationship example

- A many to many relationship - One lecturer teaches many students and a student is taught by many lecturers, so it is a many to many (m:n) relationship

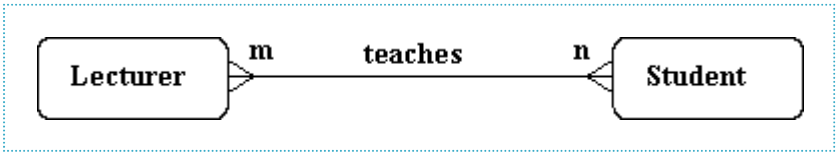


Figure : Many to Many relationship example

Optionality

A relationship can be optional or mandatory.

- If the relationship is mandatory
- an entity at one end of the relationship must be related to an entity at the other end.
- The optionality can be different at each end of the relationship
- For example, a student must be on a course. This is mandatory. To the relationship `student studies course' is mandatory.
- But a course can exist before any students have enrolled. Thus the relationship `course is_studied_by student' is optional.
- To show optionality, put a circle or `0' at the `optional end' of the relationship.
- As the optional relationship is `course is_studied_by student', and the optional part of this is the student, then the `O' goes at the student end of the relationship connection.

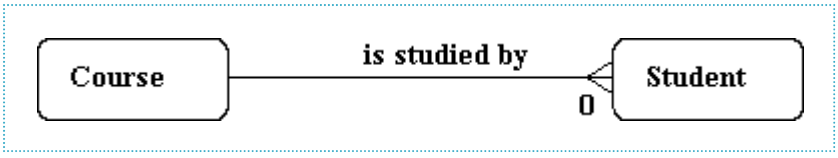


Figure : Optionality example

- It is important to know the optionality because you must ensure that whenever you create a new entity it has the required mandatory links.

Entity Sets

Sometimes it is useful to try out various examples of entities from an ER model. One reason for this is to confirm the correct cardinality and optionality of a relationship. We use an `entity set diagram' to show entity examples graphically. Consider the example of `course is_studied_by student'.

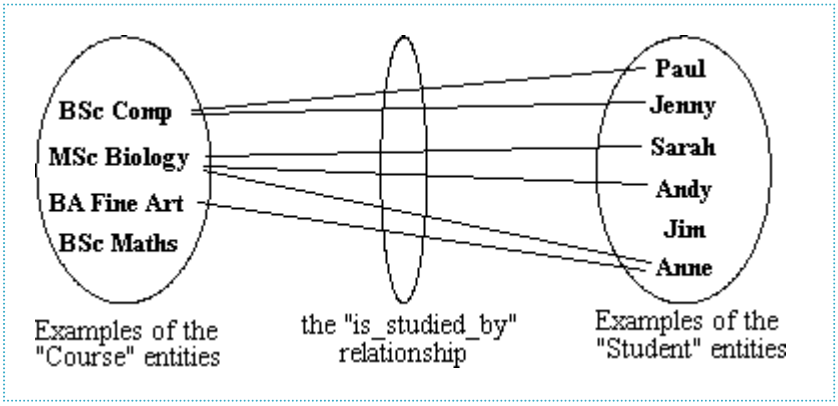


Figure : Entity set example

Confirming Correctness

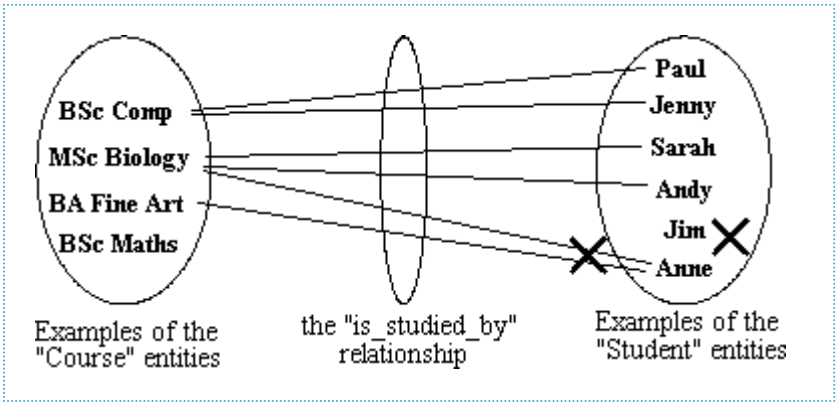


Figure : Entity set confirming errors

- Use the diagram to show all possible relationship scenarios.
- Go back to the requirements specification and check to see if they are allowed.
- If not, then put a cross through the forbidden relationships
- This allows you to show the cardinality and optionality of the relationship

Deriving the relationship parameters

To check we have the correct parameters (sometimes also known as the degree) of a relationship, ask two questions:

1. One course is studied by how many students? Answer = `zero or more'.
    - This gives us the degree at the `student' end.
    - The answer `zero or more' needs to be split into two parts.
    - The `more' part means that the cardinality is `many'.
    - The `zero' part means that the relationship is `optional'.
    - If the answer was `one or more', then the relationship would be `mandatory'.
2. One student studies how many courses? Answer = `One'
    - This gives us the degree at the `course' end of the relationship.

- The answer `one' means that the cardinality of this relationship is 1, and is `mandatory'
- If the answer had been `zero or one', then the cardinality of the relationship would have been 1, and be `optional'.

Redundant relationships

Some ER diagrams end up with a relationship loop.

- check to see if it is possible to break the loop without losing info
- Given three entities A, B, C, where there are relations A-B, B-C, and C-A, check if it is possible to navigate between A and C via B. If it is possible, then A-C was a redundant relationship.
- Always check carefully for ways to simplify your ER diagram. It makes it easier to read the remaining information.

Redundant relationships example

- Consider entities `customer' (customer details), `address' (the address of a customer) and `distance' (distance from the company to the customer address).

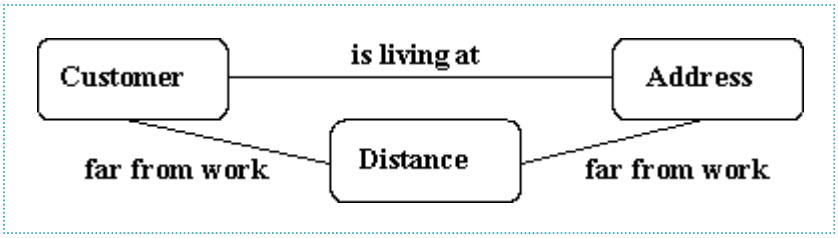


Figure : Redundant relationship

Splitting n:m Relationships

A many to many relationship in an ER model is not necessarily incorrect. They can be replaced using an intermediate entity. This should only be done where:

- the m:n relationship hides an entity
- the resulting ER diagram is easier to understand.

Splitting n:m Relationships - Example

Consider the case of a car hire company. Customers hire cars, one customer hires many card and a car is hired by many customers.

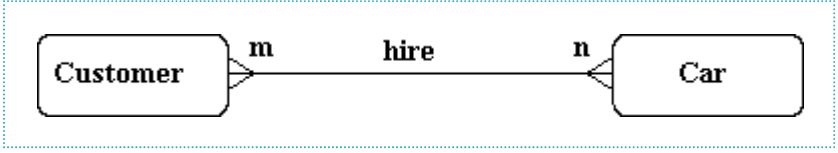


Figure : Many to Many example

The many to many relationship can be broken down to reveal a `hire' entity, which contains an attribute `date of hire'.

Figure : Splitting the Many to Many example

Constructing an ER model

Before beginning to draw the ER model, read the requirements specification carefully. Document any assumptions you need to make.

1. Identify entities - list all potential entity types. These are the object of interest in the system. It is better to put too many entities in at this stage and them discard them later if necessary.
2. Remove duplicate entities - Ensure that they really separate entity types or just two names for the same thing.
    - Also do not include the system as an entity type
    - e.g. if modelling a library, the entity types might be books, borrowers, etc.
    - The library is the system, thus should not be an entity type.
3. List the attributes of each entity (all properties to describe the entity which are relevant to the application).
    - Ensure that the entity types are really needed.
    - are any of them just attributes of another entity type?
    - if so keep them as attributes and cross them off the entity list.
    - Do not have attributes of one entity as attributes of another entity!
4. Mark the primary keys.
    - Which attributes uniquely identify instances of that entity type?
    - This may not be possible for some weak entities.
5. Define the relationships
    - Examine each entity type to see its relationship to the others.
6. Describe the cardinality and optionality of the relationships
    - Examine the constraints between participating entities.
7. Remove redundant relationships
    - Examine the ER model for redundant relationships.

ER modelling is an iterative process, so draw several versions, refining each one until you are happy with it. Note that there is no one right answer to the problem, but some solutions are better than others!

Entity Relationship Modelling - 2

Contents

Overview

- construct an ER model
- understand the problems associated with ER models
- understand the modelling concepts of Enhanced ER modelling

Country Bus Company

A Country Bus Company owns a number of busses. Each bus is allocated to a particular route, although some routes may have several busses. Each route passes through a number of towns. One or more drivers are allocated to each stage of a route, which corresponds to a journey through some or all of the towns on a route. Some of the towns have a garage where busses are kept and each of the busses are identified by the registration number and can carry different numbers of passengers, since the vehicles vary in size and can be single or double-decked. Each route is identified by a route number and information is available on the average number of passengers carried per day for each route. Drivers have an employee number, name, address, and sometimes a telephone number.

Entities

- Bus - Company owns busses and will hold information about them.
- Route - Buses travel on routes and will need described.
- Town - Buses pass through towns and need to know about them
- Driver - Company employs drivers, personnel will hold their data.
- Stage - Routes are made up of stages
- Garage - Garage houses buses, and need to know where they are.

Relationships

- A bus is allocated to a route and a route may have several buses.
- Bus-route (m:1) is serviced by
- A route comprises of one or more stages.

- route-stage (1:m) comprises
- One or more drivers are allocated to each stage.
- driver-stage (m:1) is allocated
- A stage passes through some or all of the towns on a route.
- stage-town (m:n) passes-through
- A route passes through some or all of the towns
- route-town (m:n) passes-through
- Some of the towns have a garage
- garage-town (1:1) is situated
- A garage keeps buses and each bus has one `home' garage
- garage-bus (m:1) is garaged

Draw E-R Diagram

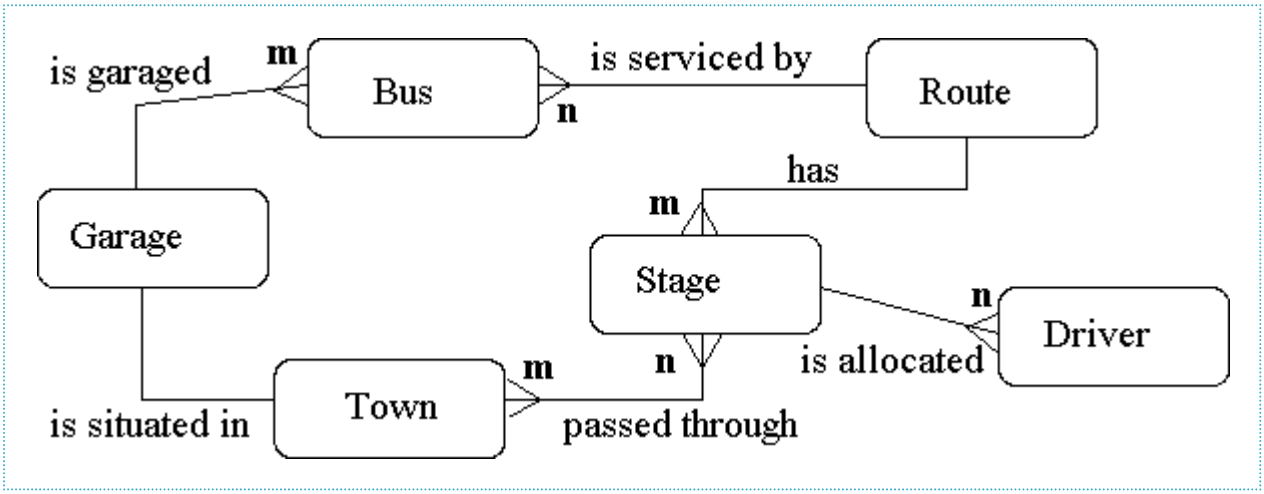


Figure : Bus Company

Attributes

- Bus (reg-no,make,size,deck,no-pass)
- Route (route-no,avg-pass)
- Driver (emp-no,name,address,tel-no)
- Town (name)
- Stage (stage-no)
- Garage (name,address)

Problems with ER Models

There are several problems that may arise when designing a conceptual data model. These are known as connection traps.

There are two main types of connection traps:

1. fan traps
2. chasm traps

Fan traps

A fan trap occurs when a model represents a relationship between entity types, but the pathway

between certain entity occurrences is ambiguous. It occurs when 1:m relationships fan out from a single entity.

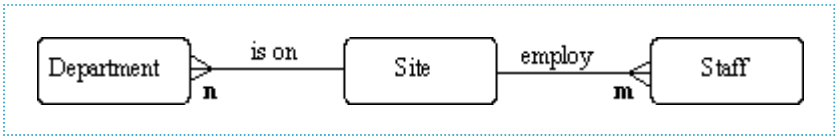


Figure : Fan Trap

A single site contains many departments and employs many staff. However, which staff work in a particular department?

The fan trap is resolved by restructuring the original ER model to represent the correct association.

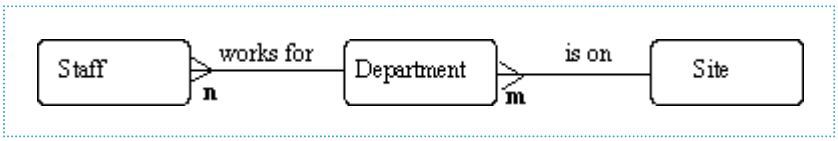


Figure : Resolved Fan Trap

Chasm traps

A chasm trap occurs when a model suggests the existence of a relationship between entity types, but the pathway does not exist between certain entity occurrences.

It occurs where there is a relationship with partial participation, which forms part of the pathway between entities that are related.

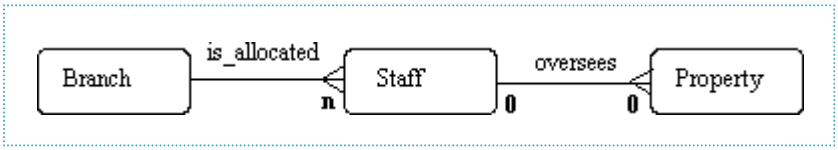


Figure : Chasm Trap

- A single branch is allocated many staff who oversee the management of properties for rent. Not all staff oversee property and not all property is managed by a member of staff.
- What properties are available at a branch?
- The partial participation of Staff and Property in the oversees relation means that some properties cannot be associated with a branch office through a member of staff.
- We need to add the missing relationship which is called `has' between the Branch and the Property entities.
- You need to therefore be careful when you remove relationships which you consider to be redundant.

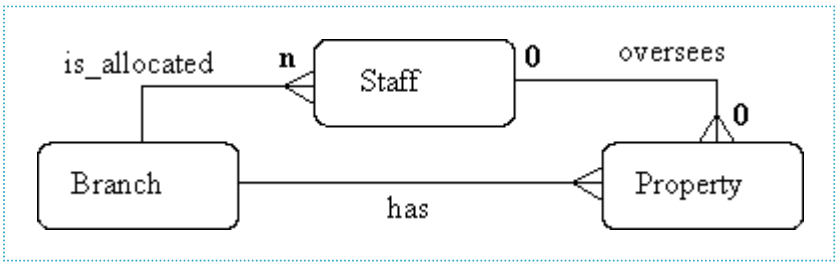


Figure : Resolved Chasm Trap

Enhanced ER Models (EER)

The basic concepts of ER modelling is not powerful enough for some complex applications... We require some additional semantic modelling concepts:

- Specialisation
- Generalisation
- Categorisation
- Aggregation

First we need some new entity constructs.

- Superclass - an entity type that includes distinct subclasses that require to be represented in a data model.
- Subclass - an entity type that has a distinct role and is also a member of a superclass.

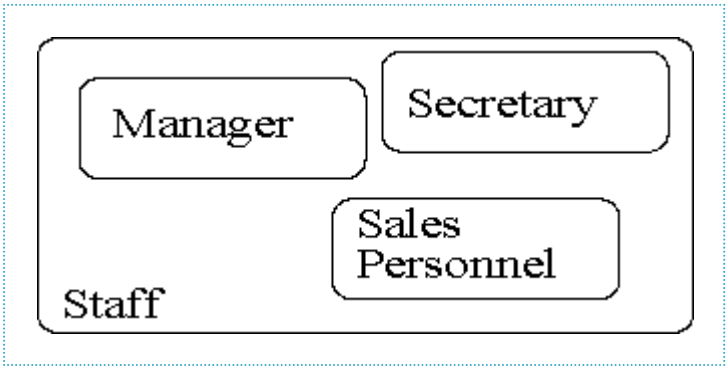


Figure : Superclass and subclasses

Subclasses need not be mutually exclusive; a member of staff may be a manager and a sales person.

The purpose of introducing superclasses and subclasses is to avoid describing types of staff with possibly different attributes within a single entity. This could waste space and you might want to make some attributes mandatory for some types of staff but other staff would not need these attributes at all.

Specialisation

This is the process of maximising the differences between members of an entity by identifying their distinguishing characteristics.

- Staff(staff_no,name,address,dob)
- Manager(bonus)
- Secretary(wp_skills)
- Sales_personnel(sales_area, car_allowance)

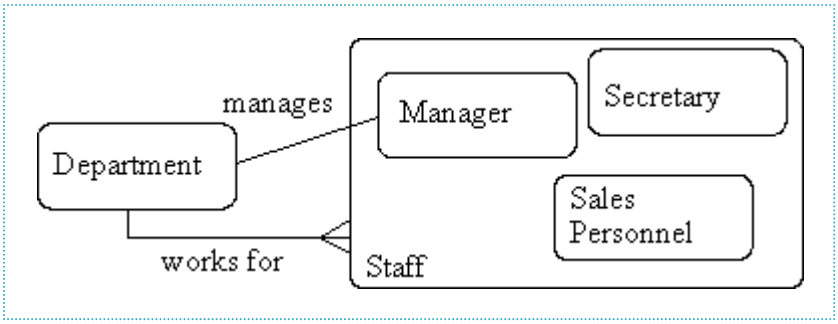


Figure : Specialisation in action

- Here we have shown that the manages relationship is only applicable to the Manager subclass, whereas the works_for relationship is applicable to all staff.
- It is possible to have subclasses of subclasses.

Generalisation

Generalisation is the process of minimising the differences between entities by identifying common features.

This is the identification of a generalised superclass from the original subclasses. This is the process of identifying the common attributes and relationships.

For instance, taking:

```
car(regno,colour,make,model,numSeats)
motorbike(regno,colour,make,model,hasWindshield)
```

And forming:

```
vehicle(regno,colour,make,model,numSeats,hasWindshielf)
```

In this case *vehicle* has numSeats which would be NULL if the vehicle was a motorbike, and has hasWindshield which would be NULL if it was a car.

Categorisation

Left as an exercise to research.

Aggregation

Left as an exercise to research.

Mapping ER Models into Relations

Contents

Overview

- map 1:1 relationships into relations
- map 1:m relationships into relations
- map m:n relationships into relations
- differences between mapping optional and mandatory relationships.

What is a relation?

A relation is a table that holds the data we are interested in. It is two-dimensional and has rows and columns.

Each entity type in the ER model is mapped into a relation.

- The attributes become the columns.
- The individual entities become the rows.

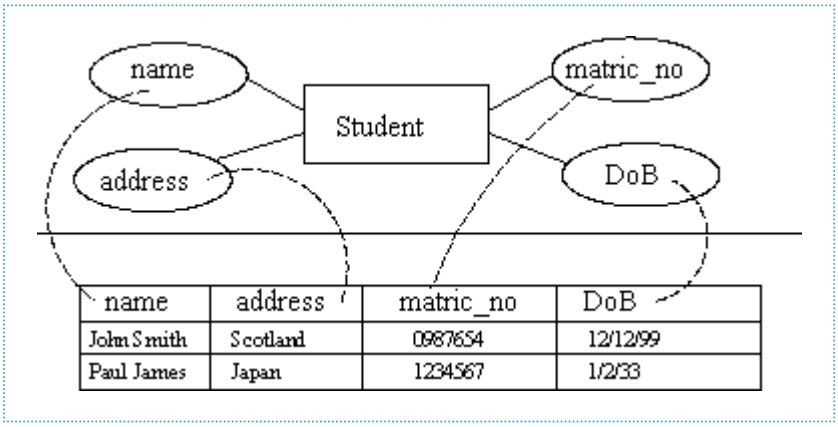


Figure : a relation

Relations can be represented textually as:

```
tablename(primary key, attribute 1, attribute 2, ... , foreign key)
```

If matric_no was the primary key, and there were no foreign keys, then the table above could be represented as:

```
student(matric no, name, address, date_of_birth)
```

When referring to relations or tables, cardinality is considered to the the number of rows in the relation or table, and arity is the number of columns in a table or attributes in a relation.

Foreign keys

A foreign key is an attribute (or group of attributes) that is the primary key to another relation.

- Roughly, each foreign key represents a relationship between two entity types.
- They are added to relations as we go through the mapping process.
- They allow the relations to be linked together.
- A relation can have several foreign keys.
- It will generally have a foreign key from each table that it is related to.
- Foreign keys are usually shown in italics or with a wiggly underline.

Preparing to map the ER model

Before we start the actual mapping process we need to be certain that we have simplified the ER model as much as possible.

This is the ideal time to check the model, as it is really the last chance to make changes to the ER model without causing major complications.

Mapping 1:1 relationships

Before tackling a 1:1 relationship, we need to know its optionality.

There are three possibilities the relationship can be:

1. mandatory at both ends
2. mandatory at one end and optional at the other
3. optional at both ends

Mandatory at both ends

If the relationship is mandatory at both ends it is often possible to subsume one entity type into the other.

- The choice of which entity type subsumes the other depends on which is the most important entity type (more attributes, better key, semantic nature of them).
- The result of this amalgamation is that all the attributes of the `swallowed up' entity become attributes of the more important entity.
- The key of the subsumed entity type becomes a normal attribute.
- If there are any attributes in common, the duplicates are removed.

- The primary key of the new combined entity is usually the same as that of the original more important entity type.

When not to combine

There are a few reason why you might not combine a 1:1 mandatory relationship.

- the two entity types represent different entities in the `real world'.
- the entities participate in very different relationships with other entities.
- efficiency considerations when fast responses are required or different patterns of updating occur to the two different entity types.

If not combined...

If the two entity types are kept separate then the association between them must be represented by a foreign key.

- The primary key of one entity type comes the foreign key in the other.
- It does not matter which way around it is done but you should not have a foreign key in each entity.

Example

- Two entity types; staff and contract.
- Each member of staff must have one contract and each contract must have one member of staff associated with it.
- It is therefore a mandatory relations at both ends.

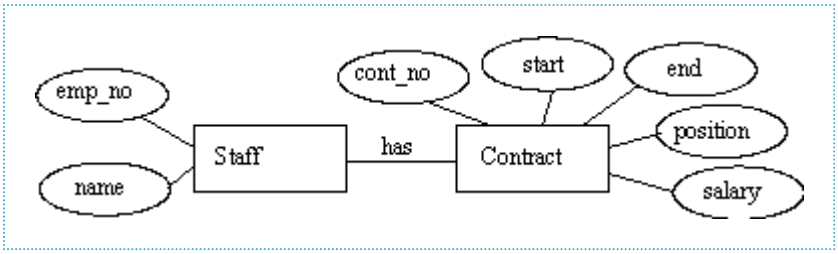


Figure : 1:1 mandatory relationship

- These to entity types could be amalgamated into one.

```
Staff(emp_no, name, cont_no, start, end, position, salary)
```

- or kept apart and a foreign key used

```
Staff(emp_no, name, contract_no)
Contract(cont_no, start, end, position, salary)
```

- or

```
Staff(emp_no, name)
Contract(cont_no, start, end, position, salary, emp_no)
```

Mandatory <->Optional

The entity type of the optional end may be subsumed into the mandatory end as in the previous example.

It is better NOT to subsume the mandatory end into the optional end as this will create null entries.

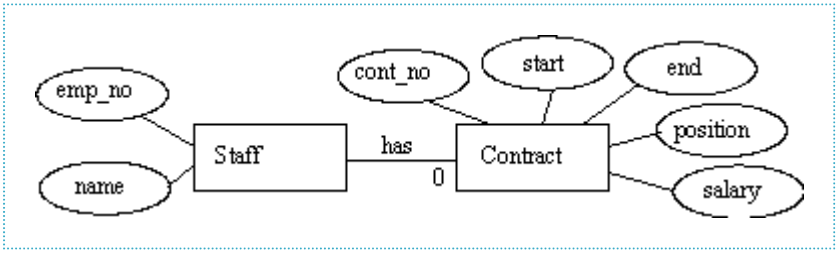


Figure : 1:1 with 1 optional end

If we add to the specification that each staff member may have at most one contract (thus making the relation optional at one end).

- Map the foreign key into Staff - the key is null for staff without a contract.

```
Staff(emp_no, name, contract_no)
Contract(cont_no, start, end, position, salary)
```

- Map the foreign key into Contract - *emp_no* is mandatory thus never null.

```
Staff(emp_no, name)
Contract(cont_no, start, end, position, salary, emp_no)
```

Example

Consider this example:

- Staff "Gordon", empno 10, contract no 11.
- Staff "Andrew", empno 11, no contract.
- Contract 11, from 1st Jan 2001 to 10th Jan 2001, lecturer, on £2.00 a year.

Foreign key in Staff:

Contract Table:

Cont_no	Start	End	Position	Salary
11	1st Jan 2001	10th Jan 2001	Lecturer	£2.00

Staff Table:

Empno	Name	Contract No
10	Gordon	11
11	Andrew	**NULL**

However, Foreign key in Contract:

Contract Table:

Cont_no	Start	End	Position	Salary	Empno
11	1st Jan 2001	10th Jan 2001	Lecturer	£2.00	10

Staff Table:

Empno	Name
10	Gordon
11	Andrew

As you can see, both ways store the same information, but the second way has no NULLs.

Mandatory <->Optional - Subsume?

The reasons for not subsuming are the same as before with the following additional reason.

- very few of the entities from the mandatory end are involved in the relationship. This could cause a lot of wasted space with many blank or null entries.

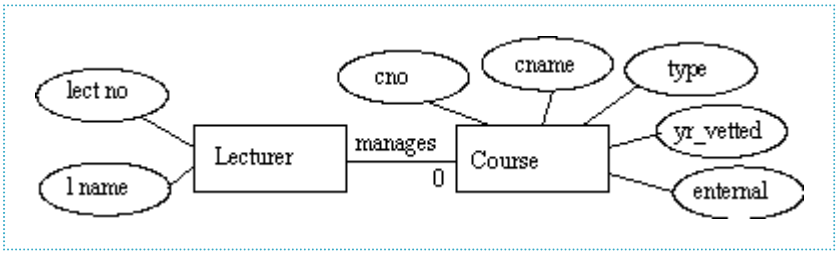


Figure : 1 optional end

- If only a few lecturers manage courses and Course is subsumed into Lecturer then there would be many null entries in the table.

```
Lecturer(lect_no, l_name, cno, c_name, type, yr_vetted, external)
```

- It would be better to keep them separate.

```
Lecturer(lect_no, l_name)
Course(cno, c_name, type, yr_vetted, external,lect_no)
```

Summary...

So for 1:1 optional relationships, take the primary key from the `mandatory end' and add it to the `optional end' as a foreign key.

So, given entity types A and B, where A <->B is a relationship where the A end it optional, the result would be:

```
A (primary key,attribute,...,foreign key to B)
B (primary key,attribute,...)
```

Optional at both ends...

Such examples cannot be amalgamated as you could not select a primary key. Instead, one foreign key is used as before.

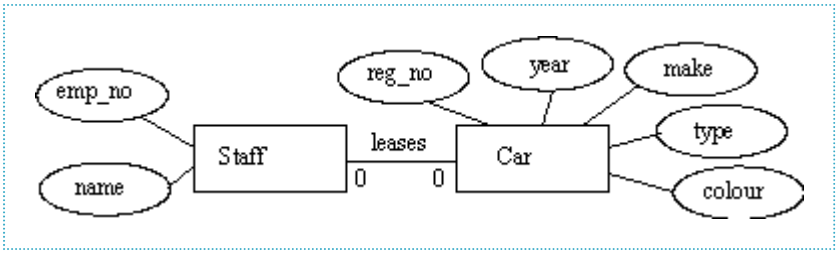


Figure : 2 optional end

- Each staff member may lease up to one car
- Each car may be leased by at most one member of staff
- If these were combined together...

```
Staff_car(emp_no, name, reg_no, year, make, type, colour)
```

what would be the primary key?

- If emp_no is used then all the cars which are not being leased will not have a key.
- Similarly, if the reg_no is used, all the staff not leasing a car will not have a key.
- A compound key will not work either.

Mapping 1:m relationships

To map 1:m relationships, the primary key on the `one side' of the relationship is added to the `many side' as a foreign key.

For example, the 1:m relationship `course-student':

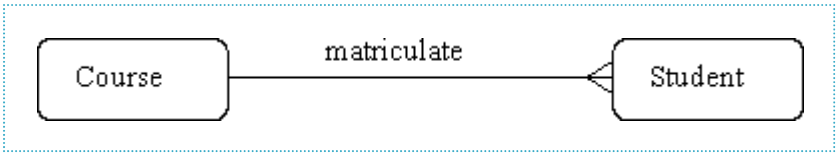


Figure : Mapping 1:m relationships

- Assuming that the entity types have the following attributes:

```
Course(course_no, c_name)
Student(matric_no, st_name, dob)
```

- Then after mapping, the following relations are produced:

```
Course(course_no, c_name)
Student(matric_no, st_name, dob, course_no)
```

- If an entity type participates in several 1:m relationships, then you apply the rule to each relationship, and add foreign keys as appropriate.

Mapping n:m relationships

If you have some m:n relationships in your ER model then these are mapped in the following

manner.

- A new relation is produced which contains the primary keys from both sides of the relationship
- These primary keys form a composite primary key.

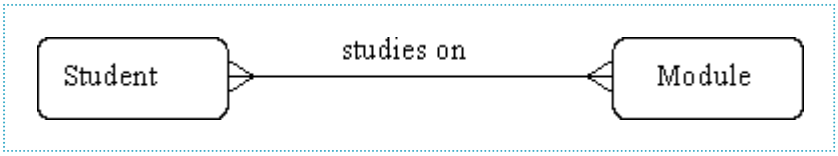


Figure : Mapping n:m relationships

- Thus

```
Student(matric_no, st_name, dob)
Module(module_no, m_name, level, credits)
```

- becomes

```
Student(matric_no, st_name, dob)
Module(module_no, m_name, level, credits)
Studies(matric_no,module_no)
```

This is equivalent to:

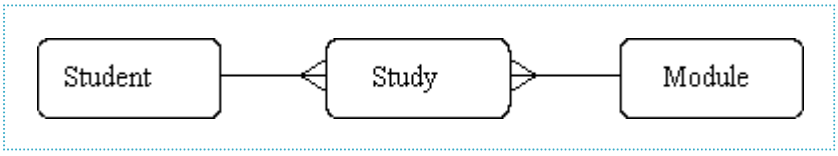


Figure : After Mapping a n:m relationship

```
Student(matric_no,st_name,dob)
Module(module_no,m_name,level,credits)
Study()
```

Summary

- 1-1 relationships
  Depending on the optionality of the relationship, the entities are either combined or the primary key of one entity type is placed as a foreign key in the other relation.
- 1-m relationships
  The primary key from the `one side' is placed as a foreign key in the `many side'.
- m-n relationships
  A new relation is created with the primary keys from each entity forming a composite key.

Advanced ER Mapping

Contents

Overview

- map parallel relationships into relations
- map unary relationships into relations
- map superclasses and subclasses into relations

Mapping parallel relationships

Parallel relationships occur when there are two or more relationships between two entity types (e.g. employees own and service cars).

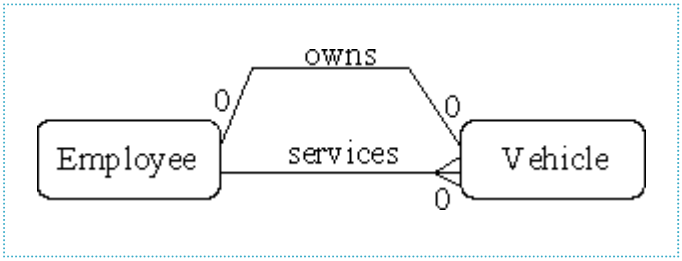


Figure : Parallel Relationships

- In order to distinguish between the two roles we can give the foreign keys different names.
- Each relationship is mapped according to the rules, and we end up with two foreign keys added to the Vehicle table.
- So we add the *employee_no* as the *owner_no* in order to represent the `owns' relationship.
- We then add the *employee_no* as the *serviced_by* attribute in order to represent the `services' relationship.
- Before mapping

```
Employee(employee_no,...)
Vehicle(registration_no,...)
```

- After mapping

```
Employee(employee_no,...)
Vehicle(registration_no,owner_no,serviced_by,...)
```

Mapping 1:m in unary relationships

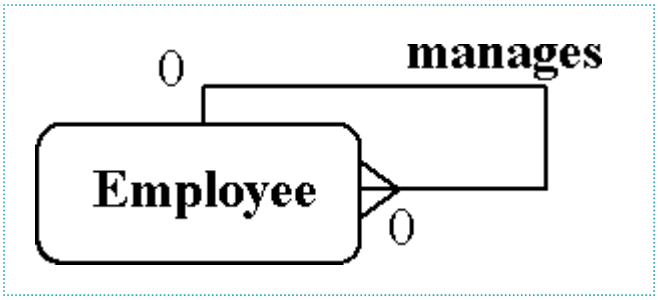


Figure : Mapping recursive relationships

- Employees manage employees
- Each employee has an employee_no with is the primary key
- We represent the manages relationship by adding a *manager_no* as a foreign key.
- This is in fact the employee_no of the manager.
- It is given a different name to clearly convey what it represents, and to ensure that all the entity type's attributes have unique names, as to do otherwise would be invalid.
- After mapping

```
Employee(employee_no,manager_no, name,...)
```

- So in general, for unary 1:n relationships, the foreign key is the primary key of the same table, but is given a different name.
- Note that the relationship is optional in both directions because not all staff can be managers, and the top manager is not managed by anybody else.

Mapping superclasses and subclasses

There are three ways of implementing superclasses and subclasses and it depends on the application which will be the most suitable.

Only the first method is a true reflection of the superclasses and subclasses and if either of the other methods is preferential then the model should not have subclasses.

1. One relation for the superclass and one relation for each subclass.
2. One relation for each subclass.
3. One relation for the superclass.

Example

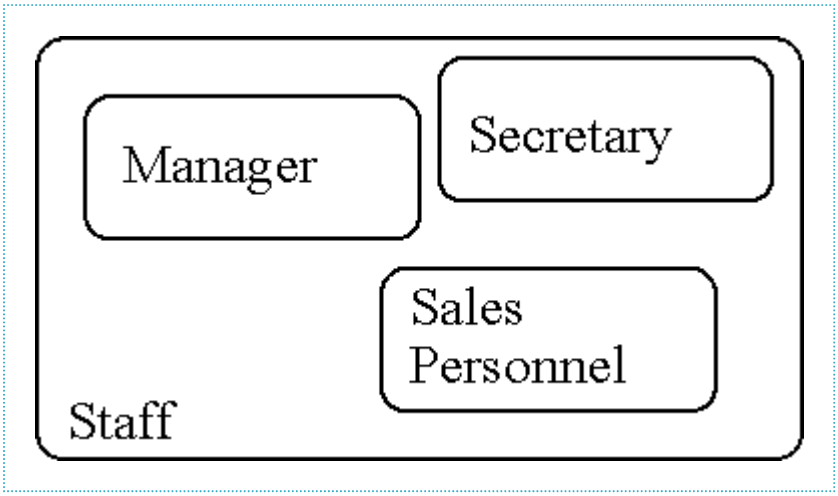


Figure : Superclass/Subclass mapping example

```
Staff(staff_no,name,address,dob)
Manager(bonus)
Secretary(wp_skills)
Sales_personnel(sales_area, car_allowance)
```

One relation for the superclass and one relation for each subclass:

```
Staff(staff_no,name,address,dob)
Manager(staff_no,bonus)
Secretary(staff_no,wp_skills)
Sales_personnel(staff_no,sales_area, car_allowance)
```

The primary key of the superclass is mapped into each subclass and becomes the subclasses primary key. This represents most closely the EER model. However it can cause efficiency problems as there needs to be a lot of joins if the additional information is often needed for all staff.

One relation for each subclass:

```
Manager(staff_no,name,address,dob,bonus)
Secretary(staff_no,name,address,dob,wp_skills)
Sales_personnel(staff_no,name,address,dob,sales_area, car_allowance)
```

All attributes are mapped into each subclass. It is equivalent to having three separate entity types and no superclass.

It is useful if there are no overlapping entities and there are no relationships between the superclass and other entity types. It is poor if the subclasses are not disjoint as there is data duplication in each relation which can cause problems with consistency.

One relation for the superclass:

```
Staff(staff_no,name,address,dob, bonus, wp_skills, sales_area, car_allowance)
```

This represents a single entity type with no subclasses.

This is no good if the subclasses are not disjoint or if there are relationships between the subclasses and the other entities.

In addition, there will be many null fields if the subclasses do not overlap a lot. However, it avoids any joins to get additional information about each member of staff.

Chapter 3 - SQL

Sections covering basic SQL usage.

Structured Query Language

Contents

In the other chapters of this course consideration is given to producing a good design for a database structure or schema. In this chapter the focus is on applying this schema to a database management system, and then using that DBMS to allow storage and retrieval of data.

To communicate with the database system itself we need a language. SQL is an international standard language for manipulating relational databases. It is based on an IBM product. SQL is short for Structured Query Language.

SQL can create schemas, delete them, and change them. It can also put data into schemas and remove data. It is a data handling language, but it is not a programming language.

SQL is a DSL (Data Sub Language), which is really a combination of two languages. These are the Data Definition Language (DDL) and the Data Manipulation Language (DML). Schema changes are part of the DDL, while data changes are part of the DML. We will consider both parts of the DSL in this discussion of SQL.

Database Models

A data model comprises

- a data structure
- a set of integrity constraints
- operations associated with the data structure

Examples of data models include:

- hierarchic
- network
- relational

Models other than the relational database module used to be quite popular. Each model type is appropriate to particular types of problem. The Relational model type is the most popular in use today, and the other types are not discussed further.

Relational Databases

The relational data model comprises:

- relational data structure
- relational integrity constraints
- relational algebra or equivalent (SQL)

- SQL is an ISO language based on relational algebra
- relational algebra is a mathematical formulation

Relational Data Structure

A relational data structure is a collection of tables or relations.

- A relation is a collection of rows or tuples
- A tuple is a collection of columns or attributes
- A domain is a pool of values from which the actual attribute values are taken.

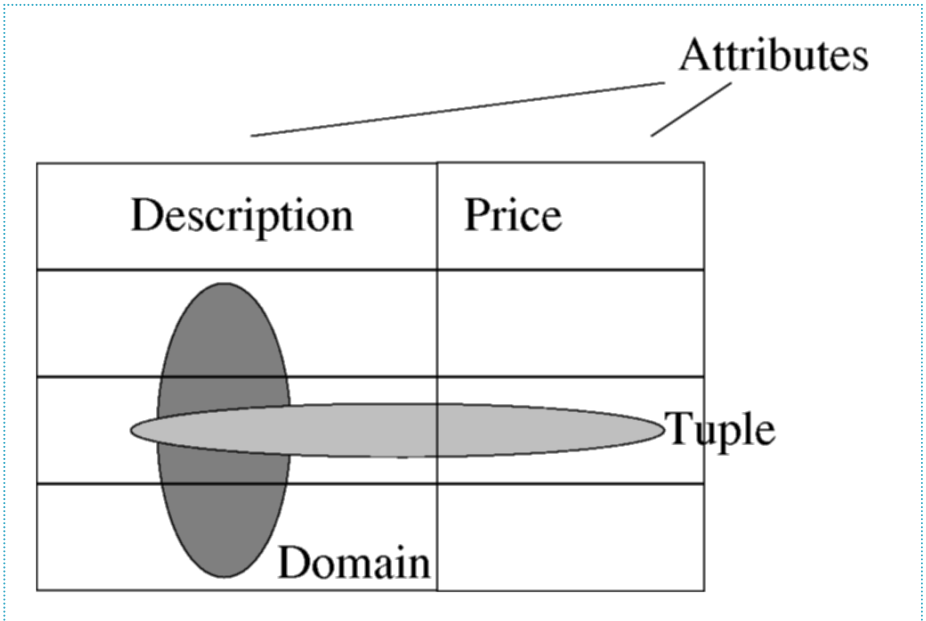


Figure : Tuples and Domains

Domain and Integrity Constraints

- Domain Constraints
  - limit the range of domain values of an attribute
  - specify uniqueness and `nullness' of an attribute
  - specify a default value for an attribute when no value is provided.

- Entity Integrity
  - every tuple is uniquely identified by a unique non-null attribute, the primary key.
- Referential Integrity
  - rows in different tables are correctly related by valid key values (`foreign' keys refer to primary keys).

Structure of a Table

In the design process tables are defined, and the relationships between tables identified. Remember a relationship is just a link between two concepts. Consider a table holding "drivers" and a table holding "car" information... Each car is owned by a driver, and therefore there is a link between "car" and "driver" to indicate which driver owns which car.

In the subsequent pages we will refer back to this driver and car arrangement. To make the examples easier, lets create some example data.

CAR

The CAR table has the following structure:

- REGNO : The registration number of the car
- MAKE : The manufacturer of the car
- COLOUR: The colour of the car
- PRICE : The price of the car when it was bought new

DRIVER

The DRIVER table has the following structure:

- NAME : The full name of the driver
- DOB : The data of birth of the driver

Relationship between CAR and DRIVER

The DRIVER and the CAR has a relationship between them of N:1. This indicates that a CAR can have only 1 DRIVER, but that a DRIVER can own more than 1 CAR simultaneously.

Figure : ER Diagram of DRIVER and CAR

In the design section we can see that this requires a FOREIGN KEY in the CAR end of the relationship. This foreign key allows us to implement the relationship in the database. We will call this field OWNER.

Example Data

DRIVER

NAME	DOB
Jim Smith	11 Jan 1980
Bob Smith	23 Mar 1981
Bob Jones	3 Dec 1986

CAR

REGNO	MAKE	COLOUR	PRICE	OWNER
F611 AAA	FORD	RED	12000	Jim Smith
J111 BBB	SKODA	BLUE	11000	Jim Smith
A155 BDE	MERCEDES	BLUE	22000	Bob Smith
K555 GHT	FIAT	GREEN	6000	Bob Jones
SC04 BFE	SMART	BLUE	13000	

Columns or Attributes

Each column is given a name which is unique within a table

Each column holds data of one specified type. E.g.

```
integer               decimal
character text        data
 -- the range of values can be further constrained
```

If a column of a row contains no data, we say it is NULL. For example, a car just off the production line might not have an owner in the database until someone buys the car. A NULL value may also indicate that the value is unavailable or inappropriate . This might be the case for a car which is being destroyed or a car where two people are arguing in court that they are both the owner.

Some important rules:

- All rows of a table must be different in some way from all other rows.
- Sometimes a row is referred to as a *Tuple*.
- **Cardinality** is the number of ROWS in a table.
- **Arity** is the number of COLUMNS in a table.

Primary Keys

A table requires a key which uniquely identifies each row in the table. This is entity integrity.

The key could have one column, or it could use all the columns. It should not use more columns than necessary. A key with more than one column is called a *composite* key.

A table may have several possible keys, the candidate keys, from which one is chosen as the primary key.

No part of a primary key may be NULL.

If the rows of the data are not unique, it is necessary to generate an artificial primary key.

In our example, DRIVER has a primary key of NAME, and CAR has a primary key of REGNO. This database will break if there are two drivers with the same name, but it gives you an idea what the primary key means...

Note that if for some reason JIM SMITH decided to change his name to "BRIAN SMITH", then not only would this have to be changed in DRIVER, but it would also have to be changed in CAR. If you changed it only in DRIVER, there would be some foreign keys pointing to DRIVER looking for a driver who does not exist. This would be an error called a REFERENTIAL INTEGRITY error, and the DBMS stops you making changes to the database which would result in such an error.

SQL Basics

Basic SQL Statements include:

- CREATE - a data structure
- SELECT - read one or more rows from a table
- INSERT - one or more rows into a table
- DELETE - one or more rows from a table
- UPDATE - change the column values in a row
- DROP - a data structure

In the remainder of this section only simple SELECT statements are considered.

Simple SELECT

The syntax of a SELECT statement is :

```
SELECT column FROM tablename
```

This would produce all the rows from the specified table, but only for the particular column mentioned. If you want more than one column shown, you can put in multiple columns separating them with commas, like:

```
SELECT column1,column2,column3 FROM tablename
```

If you want to see all the columns of a particular table, you can type:

```
SELECT * FROM tablename
```

Lets see it in action on CAR...

```
SELECT * FROM car;
```

REGNO	MAKE	COLOUR	PRICE	OWNER
F611 AAA	FORD	RED	12000	Jim Smith
J111 BBB	SKODA	BLUE	11000	Jim Smith
A155 BDE	MERCEDES	BLUE	22000	Bob Smith
K555 GHT	FIAT	GREEN	6000	Bob Jones
SC04 BFE	SMART	BLUE	13000	

```
SELECT regno FROM car;
```

REGNO
F611 AAA
J111 BBB
A155 BDE
K555 GHT
SC04 BFE

```
SELECT colour,owner FROM car;
```

COLOUR	OWNER
RED	Jim Smith
BLUE	Jim Smith
BLUE	Bob Smith
GREEN	Bob Jones
BLUE	

In SQL, you can put extra space characters and return characters just about anywhere without changing the meaning of the SQL. SQL is also case-insensitive (except for things in quotes). In addition, SQL in theory should always end with a ';' character. You need to include the ';' if you have two different SQL queries so that the system can tell when one SQL statement stops and another one starts. If you forget the ';' the online interface will put one in for you. For these reasons all of the following statements are identical and valid.

```
SELECT REGNO FROM CAR;

SELECT REGNO FROM CAR

Select REGNO from CAR

select regno FROM car


SELECT
    regno
 FROM             car;
```

Comments

Sometimes you might want to write a comment in somewhere as part of an SQL statement. A comment in this case is a simple piece of text which is meaningful to yourself, but should be ignored by the database. The characters '--', when they appear in a query, indicate the start of a comment. Everything after that point is ignored until the end of that line. The following queries are all equivalent.

```
SELECT regno
FROM   car;

SELECT regno -- The registration number
FROM   car   -- The car storage table
;
```

Warning: You cannot put a comment immediately after a ';'. Comments are only supported within the text of an SQL statement. The following will cause SQL errors:

```
SELECT regno
FROM   car;   -- Error here as comment is after the query

-- Error here as comment is before the start of the query
SELECT regno
FROM   car;
```

SELECT filters

Displaying all the rows of a table can be handy, but if we have tables with millions of rows then this type of query could take hours. Instead, we can add "filters" onto a SELECT statement to only show specific rows of a table. These filters are written into an optional part of the SELECT statement, known as a WHERE clause.

```
SELECT columns
FROM table
WHERE rule
```

The "rule" section of the WHERE clause is checked for every row that a select statement would normally show. If the whole rule is TRUE, then that row is shown, whereas if the rule is FALSE, then that row is not shown.

The rule itself can be quite complex. The simplest rule is a single equality test, such as "COLOUR = 'RED'".

Without the WHERE rule would show:

```
SELECT regno from CAR;
```

REGNO
F611 AAA
J111 BBB
A155 BDE
K555 GHT
SC04 BFE

From the database we know that only F611 AAA is RED, and the rest of the cars are either BLUE or

GREEN. Thus a rule *COLOUR = 'RED'* is only true on the row with F611 AAA, and false elsewhere. With everything in a query:

```
SELECT regno from CAR
WHERE colour = 'RED';
```

REGNO
F611 AAA

An important point to note is that queries are case sensitive between the quotes. Thus 'RED' will work, but 'red' will produce nothing. The case used in the quotes must match perfectly the case stored in the table. SQL is not forgiving and if you forget you can be scratching you head for hours trying to fix it.

Note also that "colour" does not have to appear on the SELECT line as a column name. It can if you want to see the colour, but there is no requirement for it to be there. Therefore this will work too:

```
SELECT regno,colour from CAR
WHERE colour = 'RED';
```

REGNO	COLOUR
F611 AAA	RED

Comparisons

SQL supports a variety of comparison rules for use in a WHERE clause. These include =,!=,<>, <, <=, >, and >=.

Examples of a single rule using these comparisons are:

WHERE colour = 'RED'	The *colour* attribute must be RED
WHERE colour != 'RED'	The *colour* must be a colour OTHER THAN RED
WHERE colour <> 'RED'	The same as !=
WHERE PRICE > 10000	The price of the car is MORE THAN 10000
WHERE PRICE >= 10000	The price of the car is EQUAL TO OR MORE THAN 10000
WHERE PRICE < 10000	The price of the car is LESS THAN 10000
WHERE PRICE <= 10000	The price of the car is EQUAL TO OR LESS THAN 10000

Note that when dealing with strings, like RED, you must say 'RED'. When dealing with numbers, like 10000, you can say '10000' or 10000. The choice is yours.

Dates

Date rules are some of the hardest rules to get right in writing SQL, yet there is nothing particularly complex about them. The hard part is working out what it means to be GREATER THAN a particular date.

In date calculations, you can use all the normal comparators.

```
SELECT name,dob from driver
```

NAME	DOB
Jim Smith	11 Jan 1980
Bob Smith	23 Mar 1981
Bob Jones	3 Dec 1986

```
SELECT name,dob from driver
WHERE DOB = '3 Dec 1986'
```

NAME	DOB
Bob Jones	3 Dec 1986

In other comparators, it is important to realise that a date gets bigger as you move into the future, and smaller as you move into the past. Thus to say 'DATE1 < DATE2' you are stating that DATE1 occurs before DATE2 on a calender. For example, to find all drivers who were born on or after the 1st Jan 1981 you would do:

```
SELECT name,dob from driver
WHERE DOB >= '1 Jan 1981'
```

NAME	DOB
Bob Smith	23 Mar 1981
Bob Jones	3 Dec 1986

The syntax for dates does change slightly on difference database systems, but the syntax '1 Jan 2000' works in general on all systems. Oracle also allows dates like '1-Jan-2000' and '1-Jan-00'. If you specify a year using only the last two digits, Oracle uses the current date to compute the missing parts of the year, converting '00' to '2000'. Do not get confused by saying '87' for '1987' and ending up with '2087'!

BETWEEN

Sometimes when you are dealing with dates you want to specify a range of dates to check. The best way of doing this is using BETWEEN. For instance, to find all the drivers born between 1995 and 1999 you could use:

```
SELECT name,dob from driver
WHERE DOB between '1 Jan 1985' and '31 Dec 1999'
```

NAME	DOB
Bob Jones	3 Dec 1986

Note that the dates have day of the month and month in them, and not just the year. In SQL, all dates must have a month and a year. If you try to use just a year the query will fail.

BETWEEN works for other things, not just dates. For instance, to find cars worth between 5000 and 10000, you could execute:

```
SELECT regno
```

```
FROM car
where price between 5000 and 10000;
```

REGNO	PRICE
K555 GHT	

NULL

The NULL value indicates that something has no real value. For this reason the normal value comparisons will always fail if you are dealing with a NULL. If you are looking for NULL, for instance looking for cars without owners using OWNER of CAR, all of the following are wrong!

```
SELECT regno from CAR WHERE OWNER = NULL            WRONG!
SELECT regno from CAR WHERE OWNER = 'NULL'          WRONG!
```

Instead SQL has a special comparison operator called IS which allows us to find NULL values. There is also an opposite to IS, called IS NOT, which finds all the values which are not NULL. So finding all the regnos of cars with current owners would be (note that if they have an owner, then the owner has a value and thus is NOT NULL):

```
SELECT REGNO from CAR
WHERE OWNER is not NULL
```

REGNO
F611 AAA
J111 BBB
A155 BDE
K555 GHT

And finding cars without owners would be:

```
SELECT REGNO from CAR
WHERE OWNER is NULL
```

REGNO
SC04 BFE

LIKE

When dealing with strings, sometimes you do not want to match on exact strings like *='RED'*, but instead on partial strings, substrings, or particular patterns. This could allow you, for instance, to find all cars with a colour starting with 'B'. The LIKE operator provides this functionality.

The LIKE operator is used in place of an '=' sign. In its basic form it is identical to '='. For instance, both of the following statements are identical:

```
SELECT regno FROM car WHERE colour = 'RED';
SELECT regno FROM car WHERE colour LIKE 'RED';
```

The power of LIKE is that it supports two special characters, '%' and '-'. These are equivalent to the

'*' and '?' wildcard characters of DOS. Whenever there is an '-' character in the string, any character will match. Whenever there is an '%' character in the string, 0 or more characters will match. Consider these rules:

name LIKE 'Jim Smith'	Matches 'Jim Smith'
name LIKE '_im Smith'	Matches things like 'Jim Smith' or 'Tim Smith'
name LIKE '___ Smith'	Matches 'Jim Smith' and 'Bob Smith'
name LIKE '% Smith'	Matches 'Jim Smith' and 'Bob Smith'
name LIKE '% S%'	Matches 'Jim Smith' and 'Bob Smith'
name LIKE 'Bob %'	Matches 'Bob Jones' and 'Bob Smith'
name LIKE '%'	Matches anything not null

Note however that LIKE is more powerful than a simple '=' operator, and thus takes longer to run. If you are not using any wildcard characters in a LIKE operator then you should always replace LIKE with '='.

Logical Operators and Aggregation

Contents

Logical Operators

In the previous section we saw how a single rule could be added to a query using a WHERE clause. While this is useful, usually more than a single rule is required to produce the correct result. To support multiple rules we need to make use of NOT, AND, OR and parentheses.

AND

The basic way of supporting multiple rules in a single query is by making use of AND. AND provides a way of connecting two rules together such that ALL the rules must be true before the row is shown. Lets make use again of the CAR table:

REGNO	MAKE	COLOUR	PRICE	OWNER
F611 AAA	FORD	RED	12000	Jim Smith
J111 BBB	SKODA	BLUE	11000	Jim Smith
A155 BDE	MERCEDES	BLUE	22000	Bob Smith
K555 GHT	FIAT	GREEN	6000	Bob Jones
SC04 BFE	SMART	BLUE	13000	

Consider the case where a police eye witness spots a car driving away from a crime. The witness reports that the car was BLUE and had the character '5' somewhere in the REGNO field. Taking these rules seperately...

```
SELECT regno from CAR
WHERE colour = 'BLUE';
```

REGNO
J111 BBB
A155 BDE
SC04 BFE

```
SELECT regno from CAR
WHERE regno LIKE '%5%'
```

REGNO
A155 BDE
K555 GHT

We are looking for a REGNO in common to both these results, which means the car we are looking for is 'A155 BDE'. Rather than doing this ourselves we want the computer to identify the right car in a single query. The two rules in question are linked together with an AND.

```
SELECT regno from CAR
WHERE colour = 'BLUE' AND regno LIKE '%5%'
;
```

REGNO
A155 BDE

Remember that the layout of the SQL is independent of spaces and newlines, so this query is identical to:

```
SELECT regno
FROM   CAR
WHERE  colour = 'BLUE'
AND    regno LIKE '%5%'
;
```

You can link as many rules together as you like. So for instance if the witness said that the car was BLUE, had a 5 in the registration number, and that someone said the car was owned by Bob, we could write a query:

```
SELECT regno
FROM   CAR
WHERE  colour = 'BLUE'
AND    regno LIKE '%5%'
AND    owner LIKE 'Bob %'
;
```

OR

AND allows us to link rules together such that all rules must be true to see that row. Think of AND as 'As well as'. Sometimes we want to say 'or that' or 'either' rather than 'As well as'. To do this we use OR. For instance, lets say that the police witness said that the car colour was either RED or BLUE, and they were not sure which. If you said:

```
WHERE colour = 'RED' AND colour = 'BLUE'
```

then no rows would be produced, as you are saying you want rows where the colour is both RED and BLUE at the same time (RED as well as BLUE). What we need is *either* RED OR BLUE.

```
SELECT REGNO,COLOUR from CAR
WHERE colour = 'RED'
OR    colour = 'BLUE';
```

REGNO	COLOUR
F611 AAA	RED
J111 BBB	BLUE
A155 BDE	BLUE
SC04 BFE	BLUE

NOT

The NOT operator does the opposite of whatever comparison is being done. NOT is not frequently needed, as there is usually an opposite operator already. For instance, if you wanted the opposite of:

```
WHERE colour = 'RED'
```

You could say

```
WHERE colour != 'RED'
```

Using NOT you could also say

```
WHERE NOT colour = 'RED'
```

While not particularly useful in these simple examples, NOT comes into its own once you start to use parentheses.

Precedence

AND, OR, and NOT become more complex to understand when you mix them together in a single query. The problem is that the rules are combined together, not in the order you write them, but in the order of their precedence. This states that NOT is done first, then AND, and finally OR. This can make a BIG difference to your queries!

Consider the case of the police witness. Lets say that the car being looked for had a 5 in the registration number, and was either RED or BLUE.

```
SELECT REGNO,COLOUR from CAR
WHERE colour = 'RED'      -- 1
OR    colour = 'BLUE'     -- 2
AND   regno LIKE '%5%'    -- 3
;
```

REGNO	COLOUR
F611 AAA	RED
A155 BDE	BLUE

In this query, rule 3 and rule 2 and ANDed together first, as they have a higher precedent. Only then

is rule 1 ORed in. Thus this query says "The car is BLUE with a 5 in the regno" OR "the car is RED". What was wanted was to have rules 1 and 2 done first, and then 3, so that the query says "The car is either RED or BLUE" AND "the car had a 5 in the regno". To do this we need to use parenthesis.

Parenthesis

Parenthesis, or brackets, are used to instruct the database which rules should be done first. The database uses a simple ruleset to understand brackets. If you have any brackets, then the rule in the brackets is done first. If you have brackets within brackets, then the inner brackets are done first. In the example above, the right query can be generated as:

```
SELECT REGNO,COLOUR from CAR
WHERE (colour = 'RED'
OR    colour = 'BLUE')
AND   regno LIKE '%5%'
;
```

REGNO	COLOUR
A155 BDE	BLUE

The following queries are all identical in function to the above query...

```
SELECT REGNO,COLOUR from CAR
WHERE (colour = 'RED' OR colour = 'BLUE')
AND   regno LIKE '%5%';

SELECT REGNO,COLOUR from CAR
WHERE ( (colour = 'RED' OR colour = 'BLUE')
AND   regno LIKE '%5%');
```

Do not use brackets where they are not needed, as it makes the query harder for users to understand whats going on.

DISTINCT

Lets say you want a list of all the colours of cars in the database. The COLOUR field of CAR gives you this, and thus:

```
SELECT colour FROM car;
```

COLOUR
RED
BLUE
BLUE
GREEN
BLUE

This result was not the ideal one wanted. BLUE for some reason appears 3 times. It does this because BLUE appears 3 times in the original data. Sometimes this duplication is what is wanted, other times we want only to see the colours appearing once. To tell the database to show the rows only once, you can use the keyword DISTINCT. This appears immediately after the word SELECT.

DISTINCT effectively means that all rows which appear must be unique, and any duplicate rows will be deleted.

```
SELECT DISTINCT colour FROM car;
```

COLOUR
BLUE
GREEN
RED

ORDER BY

When a query is executed the results are displayed in an almost random order. The order is dependent on how the database management system was written. This is fine usually, but sometimes giving the data out in a particular order would make the data must more useful. There is a special clause, ORDER BY, which can be added to the end of a query to give the data a particular order.

```
SELECT make FROM car;
```

MAKE
FORD
SKODA
MERCEDES
FIAT
SMART

To order alphabetically (which in SQL is known as ascending or ASC) you can use ORDER BY or ORDER BY ASC.

```
SELECT make FROM car
ORDER BY make;
```

MAKE
FIAT
FORD
MERCEDES
SKODA
SMART

This is identical to

```
SELECT make FROM car
ORDER BY make ASC;
```

To order things in the reverse ordering, you can use ORDER BY DESC.

```
SELECT make FROM car
ORDER BY make DESC;
```

MAKE
SMART
SKODA
MERCEDES
FORD
FIAT

For complex orderings involving more than one column, you can specify multiple columns in the ORDER BY statement, simply by separating each column name with a comma. Thus a query to sort cars by colour and then make would look like:

```
SELECT make,colour FROM car
ORDER BY colour,make;
```

MAKE	COLOUR
SKODA	BLUE
SMART	BLUE
MERCEDES	BLUE
FIAT	GREEN
FORD	RED

IN

- IN (list of values) determines whether a specified value is in a set of one or more listed values.

List the registration numbers of cars which are either SKODA or SMART

```
SELECT regno,make
FROM car
WHERE make = 'SKODA' or make='SMART'
;
```

REGNO	MAKE
J111 BBB	SKODA
SC04 BFE	SMART

This can be rewritten using IN.

```
SELECT regno,make
FROM car
WHERE make IN ('SKODA','SMART')
;
```

A good way to think of IN is to consider it as "is one of the following".

Aggregate Functions

Operators exist in SQL to give results based on the statistics of a group of values stored in the

database. Such operators include "what is the maximum number" and "what is the average". These functions are called SET or AGGREGATE functions.

AVERAGE

To calculate the average of a column you use the AVG function.

```
SELECT price FROM car;
```

PRICE
12000
11000
22000
6000
13000

```
SELECT avg(price) FROM car;
```

avg(price)
12800

SUM

To calculate the SUM of all values in a column you use the SUM function.

```
SELECT sum(price) FROM car;
```

sum(price)
64000

MAX

To calculate the maximum or biggest value present in a particular column you can use the MAX function.

```
SELECT max(price) FROM car;
```

sum(price)
22000

MIN

To calculate the minimum or smallest value present in a particular column you can use the MIN function.

```
SELECT min(price) FROM car;
```

sum(price)
6000

COUNT

To work out how many rows are in a particular query result you can use the COUNT function.

Using "count(column)" counts how many rows exist in the answer where that column is NOT NULL. Using "count(*)" counts how many rows exist independent of NULL values.

```
SELECT count(price) FROM car;
```

sum(price)
5

In this case, the following SQL produces the same answer.

```
SELECT count(*) FROM car;
```

COUNT DISTINCT

Sometimes you do not want to count how many rows are in a particular column, but how many different values are stored in a column. There is a special variation of count which allows you to do that, known as *COUNT DISTINCT*. Its syntax is a little unusual...

```
SELECT count(colour) from car;
```

sum(price)
5

```
SELECT count(DISTINCT colour) from car;
```

sum(price)
3

GROUP BY aggregation

The aggregate functions are excellent when all you want is a single number answer. Frequently what is needed is statistical analysis in groups. For instance, what is the maximum cost of a car given its colour. Here we are wanting two columns, one the car colour, and the second column the highest cost. Intuitively one might think:

```
SELECT colour,max(price)
FROM   car
;
```

If you were to run this query it would produce a "group by" error.

Instead, what you have to do is consider all aggregate functions in your query, and over which columns they are going to be grouped. In this case we are grouping on colour, and want the maximum price within each "colour" group. To tell the computer this we use GROUP BY.

```
SELECT colour,price
FROM   car
;
```

COLOUR	PRICE
RED	12000
BLUE	11000
BLUE	22000
GREEN	6000
BLUE	13000

```
SELECT colour,max(price)
FROM   car
GROUP BY colour
;
```

COLOUR	max(PRICE)
RED	12000
BLUE	22000
GREEN	6000

If you are ever confused by what to put in the GROUP BY, then here is a simple rule which is 99% accurate... If you have a SELECT line with aggregate functions, then you need a GROUP BY listing all the column names from the SELECT line which are not used by the functions. In this example "price" and "colour" are columns from SELECT, but as "price" is used in MAX, only "colour" needs to go into the GROUP BY statement.

HAVING

One annoying feature of SQL is that aggregate functions are executed at almost the last stage of the query process. This makes writing queries like "Which owners own more than 1 car" quite complex. Ideally we would like to write:

```
SELECT owner from car where count(owner) > 1;
```

The problem is that this does not work! Aggregate functions cannot appear in a WHERE clause, so this query is illegal... To get around this you can have the HAVING clause. HAVING works in an identical way to WHERE, except that it runs very late in the process and allows aggregate functions. It is also VERY expensive for the database to use, so do not use it until it is absolutely essential.

Our query can now be rewritten thus:

```
SELECT owner,count(regno)
FROM car
GROUP BY owner
HAVING count(regno)>1;
```

This query also shows how many cars the owner owns. You do not have to have the function in the HAVING on the SELECT line. The following also works:

```
SELECT owner
FROM car
GROUP BY owner
HAVING count(regno)>1;
```

If you remember *count(*)* counts how many rows there are in the answer. With a GROUP BY, it counts how many rows are in each group. The difference between a count with * or with a column name is that using a column name makes the count ignore NULL entries in that column, whereas with * NULL entries are counted too. In our example, REGNO is never NULL, so the query is also identical to:

```
SELECT owner
FROM car
GROUP BY owner
HAVING count(*)>1;
```

JOINs and VIEWs

Contents

Multiple source tables

Sometimes you will need to write a query which uses more than a single table. This is perfectly acceptable in SQL, but needs a little care... It is very easy to produce multi-table queries which produce mostly rubbush.

The basic concept for producing multi-table queries is that all the tables you need must be listed in the FROM clause of the query. For example, lets try to write a query which lists the owner name, date of birth, and the registration number, for each car in the database. REGNO is in CAR, but DOB is in DRIVER. Therefore both tables are needed. The basic query looks like:

```
SELECT name,dob,regno
FROM   car,driver
;
```

The order in which the tables appear in the FROM line are irrelevent. However, this query does not produce the right answer. The reason for this is that the DBMS does not understand how to relate one table to the other.

JOIN condition

In order to usefully join multiple tables together we need to explain to the database how they are joined. The FROM clause takes all rows in all the tables listed, and forms a new table which contains all combinations of the original rows. Most of the time this results in rubbish. Look at this example.

```
SELECT *
FROM   car
;
```

REGNO	MAKE	COLOUR	PRICE	OWNER
F611 AAA	FORD	RED	12000	Jim Smith
J111 BBB	SKODA	BLUE	11000	Jim Smith
A155 BDE	MERCEDES	BLUE	22000	Bob Smith
K555 GHT	FIAT	GREEN	6000	Bob Jones
SC04 BFE	SMART	BLUE	13000	

```
SELECT *
FROM   driver
;
```

NAME	DOB
Jim Smith	11 Jan 1980
Bob Smith	23 Mar 1981
Bob Jones	3 Dec 1986

```
SELECT *
FROM   car,driver
;
```

REGNO	MAKE	COLOUR	PRICE	OWNER	NAME	DOB
F611 AAA	FORD	RED	12000	Jim Smith	Jim Smith	11 Jan 1980
J111 BBB	SKODA	BLUE	11000	Jim Smith	Jim Smith	11 Jan 1980
A155 BDE	MERCEDES	BLUE	22000	Bob Smith	Jim Smith	11 Jan 1980
K555 GHT	FIAT	GREEN	6000	Bob Jones	Jim Smith	11 Jan 1980
SC04 BFE	SMART	BLUE	13000		Jim Smith	11 Jan 1980
F611 AAA	FORD	RED	12000	Jim Smith	Bob Smith	23 Mar 1981
J111 BBB	SKODA	BLUE	11000	Jim Smith	Bob Smith	23 Mar 1981
A155 BDE	MERCEDES	BLUE	22000	Bob Smith	Bob Smith	23 Mar 1981
K555 GHT	FIAT	GREEN	6000	Bob Jones	Bob Smith	23 Mar 1981
SC04 BFE	SMART	BLUE	13000		Bob Smith	23 Mar 1981
F611 AAA	FORD	RED	12000	Jim Smith	Bob Jones	3 Dec 1986
J111 BBB	SKODA	BLUE	11000	Jim Smith	Bob Jones	3 Dec 1986
A155 BDE	MERCEDES	BLUE	22000	Bob Smith	Bob Jones	3 Dec 1986
K555 GHT	FIAT	GREEN	6000	Bob Jones	Bob Jones	3 Dec 1986
SC04 BFE	SMART	BLUE	13000		Bob Jones	3 Dec 1986

In our query, we are only interested in table combinations which obey the rules of the FOREIGN KEY relationship which joins these two tables. If you remember, the PRIMARY KEY of DRIVER (NAME) was copied into CAR as a FOREIGN KEY (named OWNER). Thus this FROM generated table needs to be filtered so that only the rows where *NAME = OWNER* appear. Note that this FROM generated table containing all the combinations of the listed tables is known as the *cartesian*

cross product. We will return to the subject of the cross product in the relational algebra chapter.

Now, in order to get our query working properly, we put in the JOIN condition *NAME = OWNER*. There are two basic ways to do this, which we will call *traditional* and *modern*. Both ways are usually referred to as an INNER JOIN.

Traditional JOIN

To put the join condition *NAME = OWNER* into a query using the traditional approach is simply to list it in the WHERE clause as a rule. So...

```
SELECT *
FROM   car,driver
WHERE  owner = name
;
```

REGNO	MAKE	COLOUR	PRICE	OWNER	NAME	DOB
F611 AAA	FORD	RED	12000	Jim Smith	Jim Smith	11 Jan 1980
J111 BBB	SKODA	BLUE	11000	Jim Smith	Jim Smith	11 Jan 1980
A155 BDE	MERCEDES	BLUE	22000	Bob Smith	Bob Smith	23 Mar 1981
K555 GHT	FIAT	GREEN	6000	Bob Jones	Bob Jones	3 Dec 1986

Modern JOIN

To put the join condition *NAME = OWNER* into a query using the modern approach, you rewrite the FROM line to say:

```
FROM table1 JOIN table2 ON (rules)
```

So in our example:

```
SELECT *
FROM   car JOIN driver ON (owner = name)
;
```

REGNO	MAKE	COLOUR	PRICE	OWNER	NAME	DOB
F611 AAA	FORD	RED	12000	Jim Smith	Jim Smith	11 Jan 1980
J111 BBB	SKODA	BLUE	11000	Jim Smith	Jim Smith	11 Jan 1980
A155 BDE	MERCEDES	BLUE	22000	Bob Smith	Bob Smith	23 Mar 1981
K555 GHT	FIAT	GREEN	6000	Bob Jones	Bob Jones	3 Dec 1986

OUTER JOIN

You might have noticed a result in the previous query (when there were no join conditions) where there was a NULL in the OWNER field. This is for a car with no current owner. Once the join condition was inserted into the query the rows with NULL owners were filtered out. This is usually exactly what is desired, but sometimes we want the join condition to be obeyed if the fields are not NULL, and the rules to be broken when there is a NULL. Such JOINs are called OUTER JOINS. In the modern JOIN syntax you simply insert either the word LEFT or the word RIGHT in front of the word JOIN.

To decide if the right word is LEFT of RIGHT, you have to consider where the NULL values will be. In our example query, the NULL value is in the OWNER field, which belongs to the CAR table. The current JOIN is:

```
FROM car JOIN driver on (owner = name)
      ^    ^   ^
      |    |   |
      |    |   +-------- To the right of JOIN
      |    +-------------- The JOIN statement
      +------------------------ To the left of JOIN
```

As the CAR table has the NULL values, and CAR appears to the left of the word JOIN in the query, the right keyword to use is LEFT JOIN. The query becomes:

```
SELECT *
FROM   car LEFT JOIN driver ON (owner = name)
;
```

REGNO	MAKE	COLOUR	PRICE	OWNER	NAME	DOB
F611 AAA	FORD	RED	12000	Jim Smith	Jim Smith	11 Jan 1980
J111 BBB	SKODA	BLUE	11000	Jim Smith	Jim Smith	11 Jan 1980
A155 BDE	MERCEDES	BLUE	22000	Bob Smith	Bob Smith	23 Mar 1981
K555 GHT	FIAT	GREEN	6000	Bob Jones	Bob Jones	3 Dec 1986
SC04 BFE	SMART	BLUE	13000			

The OUTER JOIN fills in the missing data (for the things which do not satisfy the rules) with NULLs. Note that if you swap CAR and DRIVER around in the JOIN statement you can write it as a RIGHT JOIN just as easily...

```
SELECT *
FROM   driver RIGHT JOIN car ON (owner = name)
;
```

The order of the rules in ON have no significance in deciding what is right and what is left.

FULL OUTER JOIN

First, assume that we have added a new row to DRIVER, so that it now reads as:

NAME	DOB
Jim Smith	11 Jan 1980
Bob Smith	23 Mar 1981
Bob Jones	3 Dec 1986
David Davis	1 Oct 1975

Now, David Davis does not own a car, and thus never appears in a normal inner JOIN. In an outer join, we can have:

```
SELECT *
FROM   car LEFT JOIN driver ON (owner = name)
;
```

REGNO	MAKE	COLOUR	PRICE	OWNER	NAME	DOB
F611 AAA	FORD	RED	12000	Jim Smith	Jim Smith	11 Jan 1980
J111 BBB	SKODA	BLUE	11000	Jim Smith	Jim Smith	11 Jan 1980
A155 BDE	MERCEDES	BLUE	22000	Bob Smith	Bob Smith	23 Mar 1981
K555 GHT	FIAT	GREEN	6000	Bob Jones	Bob Jones	3 Dec 1986
SC04 BFE	SMART	BLUE	13000			

```
SELECT *
FROM   car RIGHT JOIN driver ON (owner = name)
;
```

REGNO	MAKE	COLOUR	PRICE	OWNER	NAME	DOB
F611 AAA	FORD	RED	12000	Jim Smith	Jim Smith	11 Jan 1980
J111 BBB	SKODA	BLUE	11000	Jim Smith	Jim Smith	11 Jan 1980
A155 BDE	MERCEDES	BLUE	22000	Bob Smith	Bob Smith	23 Mar 1981
K555 GHT	FIAT	GREEN	6000	Bob Jones	Bob Jones	3 Dec 1986
					David Davis	1 Oct 1975

In some relatively unusual queries, it might be useful if we see all the rows which obey the join condition, followed by the rows left over from each of the tables involved in the join. This is called a FULL OUTER JOIN and is written in SQL as *FULL JOIN*.

```
SELECT *
FROM   car FULL JOIN driver ON (owner = name)
;
```

REGNO	MAKE	COLOUR	PRICE	OWNER	NAME	DOB
F611 AAA	FORD	RED	12000	Jim Smith	Jim Smith	11 Jan 1980
J111 BBB	SKODA	BLUE	11000	Jim Smith	Jim Smith	11 Jan 1980
A155 BDE	MERCEDES	BLUE	22000	Bob Smith	Bob Smith	23 Mar 1981
K555 GHT	FIAT	GREEN	6000	Bob Jones	Bob Jones	3 Dec 1986
SC04 BFE	SMART	BLUE	13000			
					David Davis	1 Oct 1975

Naming

In some complex queries the DBMS may not be able to identify what table an attribute belongs to. For instance, joining two tables ALPHA and BRAVO, where both tables have a column called NAME. Running the following:

```
SELECT name from ALPHA,BRAVO
```

would produce an error. The problem is when you say "name" is it the one in ALPHA or the one in BRAVO? Instead you have to make the query more explicit.

What you are allowed to do is in front of a column name you can say with table that column belongs to. If you wanted to say "name" in ALPHA, you could say *alpha.name*. Now it is clear what table the

column belongs to, and the query will work:

```
SELECT alpha.name from ALPHA,BRAVO
```

Aliases

Sometimes you can be dealing with large table names, and finding you have to put the table name in front of many of the attribute names. This can be a lot of typing. SQL allows you to pretend that a table is called something else for the duration of your query. This is called aliasing. So instead of

```
SELECT car.owner from car
```

you can write

```
SELECT c.owner FROM car c
```

In this way aliases provide a shorthand way to refer to tables. In a more complex example:

```
SELECT c.regno,c.owner,d.dob
FROM   car c JOIN driver d ON (c.owner = d.name)
;
```

Remember you only have to use aliases if you want to, and decorate attributes with the table names when the computer cannot work out which table attribute you are talking about.

Self Joins

Self-joins, or Equijoins, are where you want the query to use the same table more than once, but each time you use it for a different purpose.

Consider the question "Who drives a car the same colour as Bob Smith"?

```
SELECT colour FROM car WHERE owner = 'Bob Smith';
```

colour
BLUE

```
SELECT owner FROM car
WHERE colour = 'BLUE'
AND   owner != 'Bob Smith'
AND   owner NOT NULL
```

owner
Jim Smith

To run this query, we need to use CAR twice. First to find the colour, and then to find the other owners. Thus CAR is used for two different purposes. To combine these queries together, we need to use CAR twice. To make this work, we need to use table aliases to make CAR appear to be two different tables. After that, its easy!

```
SELECT other.owner
FROM  car bobsmith, car other
WHERE bobsmith.colour = other.colour -- join on the colour
```

```
AND   bobsmith.owner = 'Bob Smith'   -- In bobsmith look only for Bob Smith
AND   bobsmith.owner != other.owner  -- OTHER cannot be Bob Smith
AND   other.owner NOT NULL           -- Exclude cars without owners
```

owner
Jim Smith

VIEWs

When writing queries, the query can get longer and longer. In addition, sometimes you find that a single query uses the same rules in two different parts of the query. In programming languages you would move the duplicated code into some sort of library of subroutines. In SQL, the idea of a subroutine is called a VIEW.

A VIEW can be created in the DBMS, and each view represents a particular SQL query. Once the view is created, the query it represents is hidden from the user, and instead the view appears to be just another table. The contents of the VIEW always remains identical to the result of running the query which the view represents.

Lets say you want a query to tell you how many drivers there are and how many cars exist in the database. You could run two different queries:

```
SELECT count(*) from DRIVER;
SELECT count(*) from CAR;
```

Instead, lets put each of them in a VIEW

```
CREATE VIEW count1 (total) AS SELECT count(*) from DRIVER;
CREATE VIEW count2 (total) AS SELECT count(*) from CAR;

SELECT * from count1;
```

total
3

```
SELECT * from count2;
```

total
5

```
SELECT count1.total,count2.total from count1,count2;
```

total	total
3	5

DROP View

Once you are finished with your VIEW, you can delete it. The command to do this is *DROP VIEW viewname*. Continuing our count1 and count2 example, to delete the count1 view you would type:

```
DROP VIEW count1;
```

Subqueries and Schema

Contents

Subqueries

One SELECT statement can be used inside another, allowing the result of executing one query to be used in the WHERE rules of the other SELECT statement. Where one SELECT statement appears within another SELECT statement's WHERE clause it is known as a SUBQUERY.

One limitation of subqueries is that it can only return one attribute. This means that the subquery can only have one attribute in its SELECT line. If you supply more than one attribute the system will report an error.

Subqueries are generally used in situations where one might normally use a self join or a view. Subqueries tend to be much easier to understand.

Simple Example

Who in the database is older than Jim Smith?

```
SELECT dob FROM driver WHERE name = 'Jim Smith'
```

dob
11 Jan 1980

```
SELECT name FROM driver WHERE dob > '11 Jan 1980';
```

name
Bob Smith
Bob Jones

```
SELECT name
FROM driver
WHERE dob > (SELECT dob FROM driver WHERE name = 'Jim Smith')
```

This subquery works well, and is simple to understand, but you must take care that the subquery returns only 1 row. If there were two people called Jim Smith, the query would return two different dates of birth, and this would break the query. To get around this problem, we use ANY or ALL.

ANY and ALL

This allows us to handle subqueries which return multiple rows. You still must only have a subquery which has only a single column. If you put ANY in front of a query, then the rule you provide must be true for at least 1 of the rows returned. If you put ALL in front of the subquery, then your rule must be true for all the rows returned.

Question: What cars are the same colour as a car owned by Jim Smith?

Jim Smith owns two cars, and their colours are RED and BLUE. We want to know what cars are EITHER RED or BLUE...

```
SELECT regno FROM car
WHERE colour = ANY (SELECT colour FROM car WHERE owner = 'Jim Smith')
;
```

Question: List the drivers younger than all the people who own a blue car.

This is really looking for the age of people who own a BLUE car (2 people) and listing drivers who are younger than both of these people.

```
SELECT name,dob
FROM driver
WHERE dob < ALL (
              SELECT dob
              FROM car join driver on (owner=name)
              WHERE colour = 'BLUE'
      )
;
```

IN and NOT IN for subqueries

Just like IN could be used with something like ('BLUE','BLACK'), a subquery returns a similar construct which can similarly be accessed using IN

Question: Which cars the same colour as one of Jim Smith's cars?

```
SELECT regno FROM car
WHERE colour IN (SELECT colour FROM car WHERE owner = 'Jim Smith')
;
```

Question: Which cars do not have the same colour as one of Jim Smith's cars?

```
SELECT regno FROM car
WHERE colour NOT IN (SELECT colour FROM car WHERE owner = 'Jim Smith')
;
```

EXISTS

In almost all cases, when a question involves uniqueness then you can do it with a subquery and EXISTS or NOT EXISTS. The EXISTS operator is a simple test, which is TRUE if the subquery returns at least 1 row, and FALSE if it return 0 rows. NOT EXISTS does the opposite.

Question: List the colours which are only used once in the database.

```
SELECT colour
FROM car a
WHERE exists (
        select colour             -- does not matter what is selected
        from car b                -- As we use CAR twice, call this one b
        where a.colour = b.colour -- CAR rows with the same colour as a
        and   a.regno != b.regno  -- but a car different to the one in a
     );
```

Remember that the rules are processed for each row of a. So the query looks at row 1 of a, runs the subquery, and decides if the colour is unique. It then moves to row 2 of a and reruns the subquery.

UNION

Sometimes it is desirable to merge the results of two queries together to form a single output table. This is known as UNION. UNION only works if each query in the statement has the same number of columns, and each of the corresponding columns are of the same type.

Question: List all the drivers in the DRIVER table, and show how many cars each of them own. If a driver owns no cars, the total should be 0. We will assume that David Davis has been added to the DRIVER table, but that he owns no cars.

```
SELECT name,count(*)
FROM   driver JOIN car on (name = owner)
```

NAME	count(*)
Jim Smith	2
Bob Smith	1
Bob Jones	1

This does not show David Davis, but we could write a query to find people who own no cars using NOT IN and a subquery.

```
SELECT name,0
```

```
FROM   driver
WHERE  name not in (select owner from car);
```

name	0
David Davis	0

Now, we can merge these two results together using UNION, and thus:

```
SELECT name,count(*)
FROM   driver JOIN car on (name = owner)
UNION
SELECT name,0
FROM   driver
WHERE  name not in (select owner from car)
```

NAME	count(*)
Jim Smith	2
Bob Smith	1
Bob Jones	1
David Davis	0

Changing Data

So far we have just looked at SELECT but we need to be able to do other operations as follows:

- INSERT - which writes new rows into a database
- DELETE - which deletes rows from a database
- UPDATE - which changes values in existing rows

INSERT

The INSERT command allows you to put new rows into a table.

```
INSERT INTO table_name
 [(column_list)] VALUES (value_list)
```

The column_list lists columns to be assigned values. It can be omitted if every column is to be assigned a value. The value_list is a set of literal values giving the value for each column in column_list or CREATE TABLE order.

```
insert into driver
  values ('Jessie James','31 Nov 1892');
insert into driver (name,dob)
  values ('John Johnstone','1 Aug 1996');
```

Usually you do not have to specify the columns in the insert statement, but doing so is useful in case someone changes the table at some point in the future. By mentioning the column names you are certain that the values specified are going into the correct columns.

DELETE

The DELETE command allows you to remove rows from a table.

```
DELETE FROM table_name [WHERE condition];
```

the rows of table_name which satify the condition are deleted.

Example:

```
DELETE FROM car        -- Delete all rows from CAR
;

DELETE from car
WHERE  owner is null   -- Delete any row where a car has no owner
;
```

UPDATE

UPDATE allows you to write queries which change data already in a table. It cannot add more rows or take rows away.

```
UPDATE table_name
 SET column_name = expression,{column_name=expression}
 [WHERE condition]
```

For example, lets set all BLUE cars to GREEN.

```
UPDATE car SET colour = 'GREEN'
WHERE colour = 'BLUE';
```

This next example shows how the update calculation can be an expression. Lets add VAT of 17.5% to all prices in the CAR table. This is equivalent to multiplying the car price by 1.175.

```
UPDATE car SET price = price * 1.175
```

View Manipulation

When is the contents of a view calculated? The process of the DBMS calculating the contents of a view is called 'materialising the view'. In theory this could be:

- When it is defined or
- when it is accessed

If it is the former then subsequent inserts, deletes and updates would not be visible. If the latter then changes will be seen.

Some systems allow you to chose when views are materialised. Most do not, and views are materialised whenever they are accessed, thus all changes to the tables on which the view query is based can instantly be seen.

VIEW update, insert and delete

Can we change the data viewed through a view?

- Yes, provided the primary key of all the base tables which make up the view are present in the view.

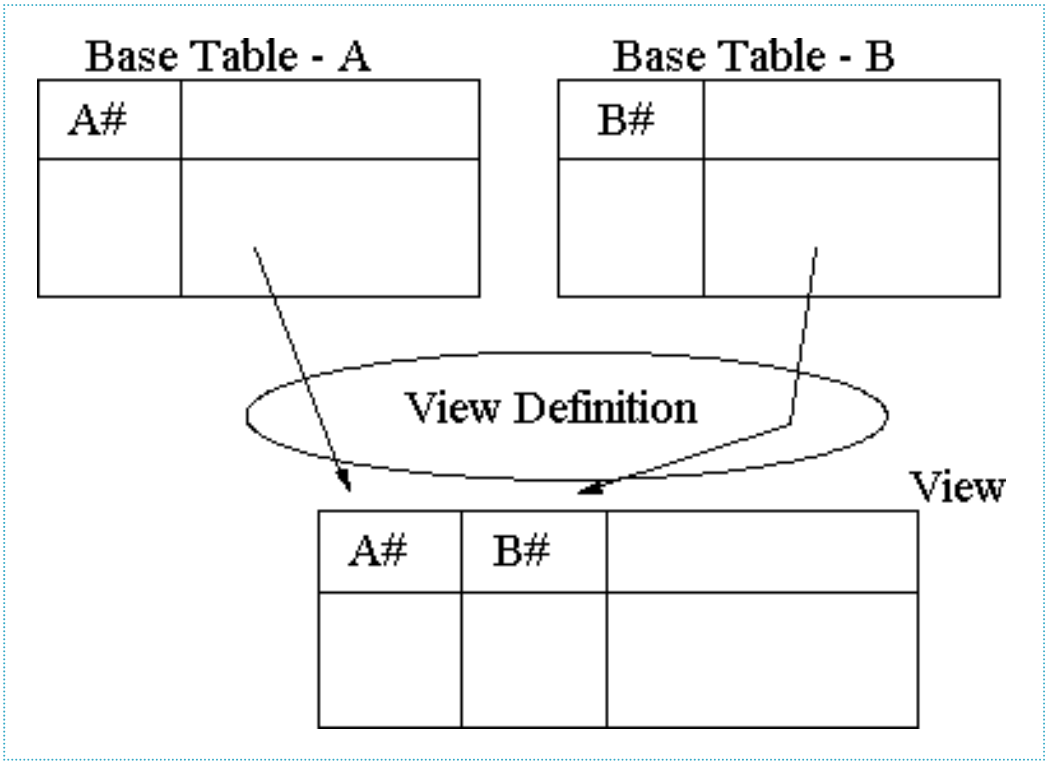


Figure : VIEW which can be updated

- The following view cannot be changed because we have no means of knowing which row of B to modify

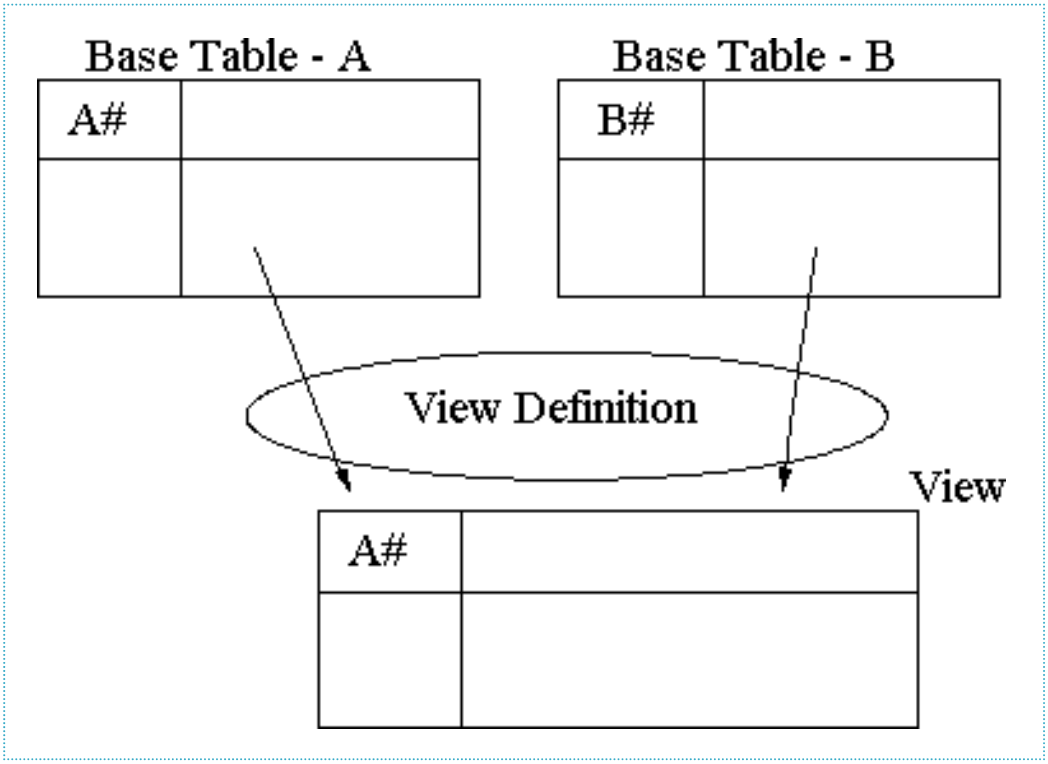


Figure : VIEW which cannot be updated

Controlling Schema

Up to this point we have assumed that the database has already been created. However, someome

must be able to create the schema to allow table structures to be defined. In the ER diagram design phase, the process will take you from written specifications to sets of relations, including foreign key definitions. At that point the relations have to be rewritten into schema creation queries.

CREATE TABLE

CREATE TABLE allows the user to create the table schemas. It has a relatively simple structure, consisting of the column names and the type of each column. We have not really mentioned column types, but there are quite a few different types in an DBMS. The important ones are:

- INTEGER - A column to hold numbers. Numbers with decimal points are not permitted. Examples: 5, 6, 10006.
- REAL - A column to hold numbers which have decimal points. Examples could be pounds and pence. Examples 5.6, 1000.35567.
- DECIMAL - A column to hold numbers which can have decimal points. It is used as DECIMAL(n) or DECIMAL(n,m), where n is the size of the number allowed before the decimal point, and m is the size allowed after the decimal point. If you do not specify m, it is assumed to be 0.
- VARCHAR - ASCII characters with a length ranging from 0 characters up to some limit. It is usually used as VARCHAR(n), where n is the maximum number of characters which can be stored. Examples include 'Hello' and 'surprise birthday'.
- CHAR - ASCII characters with a fixed length. It is usually used as CHAR(n), where n is the fixed length of the string. If you try to write a string into a CHAR which is shorter than n, spaces are added to the end of your string. For example, with CHAR(5), storing 'Hia' results in 'Hia '.
- DATE - A column which holds a day/month/year date. Examples include '1 Jan 2003' and '31 Dec 1885'.

The actual syntax of the statement is:

```
CREATE TABLE tablename (
   colname type Optionaladditionalinfo
   ,colname type Optionaladditionalinfo
   ,colname type Optionaladditionalinfo
   ,optionaladditionalinfo
);
```

At the end of each column definition you can have some additional info. This could be range rules or key information. Common ones to use include

- REFERENCES - This field is a foreign key which refers to the specified table and key. An example could be
  A INTEGER REFERENCES B(C)
  which would indicate that column A (an integer) is a foreign key which refers to a table called B, and relates to column C in B (C should be the primary key in a properly designed database).
- PRIMARY KEY - This column is the primary key.
- NOT NULL - This column must have a value.

At the end of the definition you can have some other types of optional additional information. There is a significant number of possibilities here, but the main ones include:

- PRIMARY KEY (column1,column2,...) - If the table is has a composite primary key (more than 1 column makes up the key) then you must define the key in this way at the end of the definition.
- FOREIGN KEY (column1,column2,...) REFERENCES othertable - If the table has a

relationship with another table which has a composite key, then the columns in this table which form the foreign keys must be listed using this syntax.

Example

As some examples, lets defined the DRIVER and CAR tables.

```
CREATE TABLE driver (
  name          varchar(30)     PRIMARY KEY
 ,dob           DATE            NOT NULL
);

CREATE TABLE car (
  regno         VARCHAR(8)      PRIMARY KEY
 ,make          VARCHAR(20)
 ,colour        VARCHAR(30)
 ,price         DECIMAL(8,2)
 ,owner         VARCHAR(30)     REFERENCES driver(name)
);
```

Or, using the additional information aspects of the syntax, the following statements create the same table structures.

```
CREATE TABLE driver (
  name          varchar(30)
 ,dob           DATE            NOT NULL
 ,PRIMARY KEY (name)
);

CREATE TABLE car (
  regno         VARCHAR(8)
 ,make          VARCHAR(20)
 ,colour        VARCHAR(30)
 ,price         DECIMAL(8,2)
 ,owner         VARCHAR(30)
 ,PRIMARY KEY(regno)
 ,FOREIGN KEY(owner) REFERENCES driver
);
```

DROP TABLE

Eventually there may come a time when you want to remove a table. The basic syntax is:

```
DROP TABLE tablename
```

The only difficulty in dropping tables is that you cannot drop a table if another table refers to it via a foreign key relationship. This would break the referential integrity rules. Thus in our example we can drop CAR and then DRIVER, but we cannot drop DRIVER first.

```
DROP TABLE car;
DROP TABLE driver;
```

ALTER TABLE

Most database management systems allow you to alter the definition of a table after it has been constructed, using the ALTER TABLE command. There are many variants to this, and far too much to discuss in this introduction. One simple example would be if there was a need to add a column to DRIVER to indicate the driver's address. This could be done by:

```
ALTER TABLE driver ADD address varchar(50);
```

Order of Evaluation

In summary, consider the following information, which depicts the various options of the SELECT statement, and the approximate order in which each statement is evaluated:

```
 SELECT   [distinct] column_names       7,6 eliminate unwanted data
 FROM     table_list                    1   Cartesian Product
[WHERE    conditions ]                  2   Filter rows
[GROUP BY colum_list                    3   Group Rows
  [HAVING conditions ]]                 4   eliminate unwanted groups
[ORDER BY column_list [DESC]]           5   Sort rows
```

Chapter 4 - Normalisation

Normalisation techniques for relations and raw data examples, covering unnormalised forms through to 5th Normal Form.

Normalisation

Contents

What is normalisation?

Normalisation is the process of taking data from a problem and reducing it to a set of relations while ensuring data integrity and eliminating data redundancy

- Data integrity - all of the data in the database are consistent, and satisfy all integrity constraints.
- Data redundancy – if data in the database can be found in two different locations (direct redundancy) or if data can be calculated from other data items (indirect redundancy) then the data is said to contain redundancy.

Data should only be stored once and avoid storing data that can be calculated from other data already held in the database. During the process of normalisation redundancy must be removed, but not at the expense of breaking data integrity rules.

If redundancy exists in the database then problems can arise when the database is in normal operation:

- When data is inserted the data must be duplicated correctly in all places where there is redundancy. For instance, if two tables exist for in a database, and both tables contain the employee name, then creating a new employee entry requires that both tables be updated with the employee name.
- When data is modified in the database, if the data being changed has redundancy, then all versions of the redundant data must be updated simultaneously. So in the employee example a change to the employee name must happen in both tables simultaneously.

The removal of redundancy helps to prevent insertion, deletion, and update errors, since the data is only available in one attribute of one table in the database.

The data in the database can be considered to be in one of a number of `normal forms'. Basically the

normal form of the data indicates how much redundancy is in that data. The normal forms have a strict ordering:

1. 1st Normal Form
2. 2nd Normal Form
3. 3rd Normal Form
4. BCNF

There are other normal forms, such as 4th and 5th normal forms. They are rarely utilised in system design and are not considered further here.

To be in a particular form requires that the data meets the criteria to also be in all normal forms before that form. Thus to be in 2nd normal form the data must meet the criteria for both 2nd normal form and 1st normal form. The higher the form the more redundancy has been eliminated.

Integrity Constraints

An integrity constraint is a rule that restricts the values that may be present in the database. The relational data model includes constraints that are used to verify the validity of the data as well as adding meaningful structure to it:

- entity integrity :

The rows (or tuples) in a relation represent entities, and each one must be uniquely identified. Hence we have the primary key that must have a unique non-null value for each row.

- referential integrity :

This constraint involves the foreign keys. Foreign keys tie the relations together, so it is vitally important that the links are correct. Every foreign key must either be null or its value must be the actual value of a key in another relation.

Understanding Data

Sometimes the starting point for understanding data is given in the form of relations and functional dependancies. This would be the case where the starting point in the process was a detailed specification of the problem. We already know what relations are. Functional dependancies are rules stating that given a certain set of attributes (the determinant) determines a second set of attributes.

The definition of a functional dependency looks like A->B. In this case B is a single attribute but it can be as many attributes as required (for instance, X->J,K,L,M). In the functional dependency, the determinant (the left hand side of the -> sign) can determine the set of attributes on the right hand side of the -> sign. This basically means that A selects a particular value for B, and that A is unique. In the second example X is unique and selects a particular set of values for J,K,L, and M. It can also be said that B is functionally dependent on A. In addition, a particular value of A ALWAYS gives you a particular value for B, but not vice-versa.

Consider this example:

R(matric_no, firstname, surname, tutor_number, tutor_name)

tutor_number -> tutor_name

Here there is a relation R, and a functional dependency that indicates that:

- instances of tutor_number are unique in the data
- from the data, given a tutor_number, it is always possible to work out the tutor_name.
- As an example tutor number 1 may be “Mr Smith”, but tutor number 10 may also be “Mr Smith”. Given a tutor number of 1, this is ALWAYS “Mr Smith”. However, given the name “Mr Smith” it is not possible to work out if we are talking about tutor 1 or tutor 10.

There is actually a second functional dependency for this relation, which can be worked out from the relation itself. As the relation has a primary key, then given this attribute you can determine all the other attributes in R. This is an implied functional dependency and is not normally listed in the list of functional dependents.

Extracting understanding

It is possible that the relations and the determinants have not yet been defined for a problem, and therefore must be calculated from examples of the data. Consider the following Student table.

Student - an unnormalised tablewith repeating groups

matric_no	Name	date_of_birth	subject	grade
960100	Smith, J	14/11/1977	Databases Soft_Dev ISDE	C A D
960105	White, A	10/05/1975	Soft_Dev ISDE	B B
960120	Moore, T	11/03/1970	Databases Soft_Dev Workshop	A B C
960145	Smith, J	09/01/1972	Databases	B
960150	Black, D	21/08/1973	Databases Soft_Dev ISDE Workshop	B D C D

The subject/grade pair is repeated for each student. 960145 has 1 pair while 960150 has four. Repeating groups are placed inside another set of parentheses. From the table the following relation is generated:

Student(matric_no, name, date_of_birth, (subject, grade))

The repeating group needs a key in order that the relation can be correctly defined. Looking at the data one can see that grade repeats within matric_no (for instance, for 960150, the student has 2 D grades). However, subject never seems to repeat for a single matric_no, and therefore is a candidate key in the repeating group.

Whenever keys or dependencies are extracted from example data, the information extracted is only as good as the data sample examined. It could be that another data sample disproves some of the key selections made or dependencies extracted. What is important however is that the information extracted during these exercises is correct for the data being examined.

Looking at the data itself, we can see that the same name appears more than once in the name column. The name in conjunction with the date_of_birth seems to be unique, suggesting a functional dependency of:

name, date_of_birth -> matric_no

This implies that not only is the matric_no sufficient to uniquely identify a student, the student's name combined with the date of birth is also sufficient to uniquely identify a student. It is therefore possible to have the relation Student written as:

Student(matric_no, name, date_of_birth, (subject, grade))

As guidance in cases where a variety of keys could be selected one should try to select the relation with the least number of attributes defined as primary keys.

Flattened Tables

Note that the student table shown above explicitly identifies the repeating group. It is also possible that the table presented will be what is called a flat table, where the repeating group is not explicitly shown:

Student #2 - Flattened Table

matric_no	name	date_of_birth	Subject	grade
960100	Smith, J	14/11/1977	Databases	C
960100	Smith, J	14/11/1977	Soft_Dev	A
960100	Smith, J	14/11/1977	ISDE	D
960105	White, A	10/05/1975	Soft_Dev	B
960105	White, A	10/05/1975	ISDE	B
960120	Moore, T	11/03/1970	Databases	A
960120	Moore, T	11/03/1970	Soft_Dev	B
960120	Moore, T	11/03/1970	Workshop	C
960145	Smith, J	09/01/1972	Databases	B
960150	Black, D	21/08/1973	Databases	B
960150	Black, D	21/08/1973	Soft_Dev	D
960150	Black, D	21/08/1973	ISDE	C
960150	Black, D	21/08/1973	Workshop	B

The table still shows the same data as the previous example, but the format is different. We have removed the repeating group (which is good) but we have introduced redundancy (which is bad).

Sometimes you will miss spotting the repeating group, so you may produce something like the following relation for the Student data.

Student(matric_no, name, date_of_birth, subject, grade)

matric_no -> name, date_of_birth
name, date_of_birth -> matric_no

This data does not explicitly identify the repeating group, but as you will see the result of the normalisation process on this relation produces exactly the same relations as the normalisation of the version that explicitly does have a repeating group.

First Normal Form

- First normal form (1NF) deals with the `shape' of the record type
- A relation is in 1NF if, and only if, it contains no repeating attributes or groups of attributes.
- Example:
- The Student table with the repeating group is not in 1NF
- It has repeating groups, and it is called an `unnormalised table'.

Relational databases require that each row only has a single value per attribute, and so a repeating group in a row is not allowed.

To remove the repeating group, one of two things can be done:

- either flatten the table and extend the key, or
- decompose the relation- leading to First Normal Form

Flatten table and Extend Primary Key

The Student table with the repeating group can be written as:

Student(matric_no, name, date_of_birth, (subject, grade))

If the repeating group was flattened, as in the Student #2 data table, it would look something like:

Student(matric_no, name, date_of_birth, subject, grade)

Although this is an improvement, we still have a problem. matric_no can no longer be the primary key - it does not have an unique value for each row. So we have to find a new primary key - in this case it has to be a compound key since no single attribute can uniquely identify a row. The new primary key is a compound key (matrix_no + subject).

We have now solved the repeating groups problem, but we have created other complications. Every repetition of the matric_no, name, and data_of_birth is redundant and liable to produce errors.

With the relation in its flattened form, strange anomalies appear in the system. Redundant data is the main cause of insertion, deletion, and updating anomalies.

Insertion anomaly:

With the primary key including subject, we cannot enter a new student until they have at least one subject to study. We are not allowed NULLs in the primary key so we must have an entry in both matric_no and subject before we can create a new record.

- This is known as the insertion anomaly. It is difficult to insert new records into the database.
- On a practical level, it also means that it is difficult to keep the data up to date.

Update anomaly

If the name of a student were changed for example Smith, J. was changed to Green, J. this would require not one change but many one for every subject that Smith, J. studied.

Deletion anomaly

If all of the records for the `Databases' subject were deleted from the table,we would inadvertently lose all of the information on the student with matric_no 960145. This would be the same for any student who was studying only one subject and the subject was deleted. Again this problem arises from the need to have a compound primary key.

Decomposing the relation

- The alternative approach is to split the table into two parts, one for the repeating groups and one of the non-repeating groups.
- the primary key for the original relation is included in both of the new relations

Record

matric_no	**subject**	**grade**
960100	Databases	C
960100	Soft_Dev	A
960100	ISDE	D
960105	Soft_Dev	B
960105	ISDE	B
...	...	...
960150	Workshop	B

Student

matric_no	name	date_of_birth
960100	Smith,J	14/11/1977
960105	White,A	10/05/1975
960120	Moore,T	11/03/1970
960145	Smith,J	09/01/1972
960150	Black,D	21/08/1973

- We now have two relations, Student and Record.
- Student contains the original non-repeating groups
- Record has the original repeating groups and the matric_no

Student(matric_no, name, date_of_birth)
Record(matric_no, subject, grade)

Matric_no remains the key to the Student relation. It cannot be the complete key to the new Record relation - we end up with a compound primary key consisting of matric_no and subject. The matric_no is the link between the two tables - it will allow us to find out which subjects a student is

studying . So in the Record relation, matric_no is the foreign key.

This method has eliminated some of the anomalies. It does not always do so, it depends on the example chosen

- In this case we no longer have the insertion anomaly
- It is now possible to enter new students without knowing the subjects that they will be studying
- They will exist only in the Student table, and will not be entered in the Record table until they are studying at least one subject.
- We have also removed the deletion anomaly
- If all of the `databases' subject records are removed, student 960145 still exists in the Student table.
- We have also removed the update anomaly

Student and Record are now in First Normal Form.

Second Normal Form

Second normal form (or 2NF) is a more stringent normal form defined as:

A relation is in 2NF if, and only if, it is in 1NF and every non-key attribute is fully functionally dependent on the whole key.

Thus the relation is in 1NF with no repeating groups, and all non-key attributes must depend on the whole key, not just some part of it. Another way of saying this is that there must be no partial key dependencies (PKDs).

The problems arise when there is a compound key, e.g. the key to the Record relation - matric_no, subject. In this case it is possible for non-key attributes to depend on only part of the key - i.e. on only one of the two key attributes. This is what 2NF tries to prevent.

Consider again the Student relation from the flattened Student #2 table:

```
Student(matric_no, name, date_of_birth, subject, grade )
```

- There are no repeating groups
- The relation is already in 1NF
- However, we have a compound primary key - so we must check all of the non-key attributes against each part of the key to ensure they are functionally dependent on it.
- matric_no determines name and date_of_birth, but not grade.
- subject together with matric_no determines grade, but not name or date_of_birth.
- So there is a problem with potential redundancies

A dependency diagram is used to show how non-key attributes relate to each part or combination of parts in the primary key.

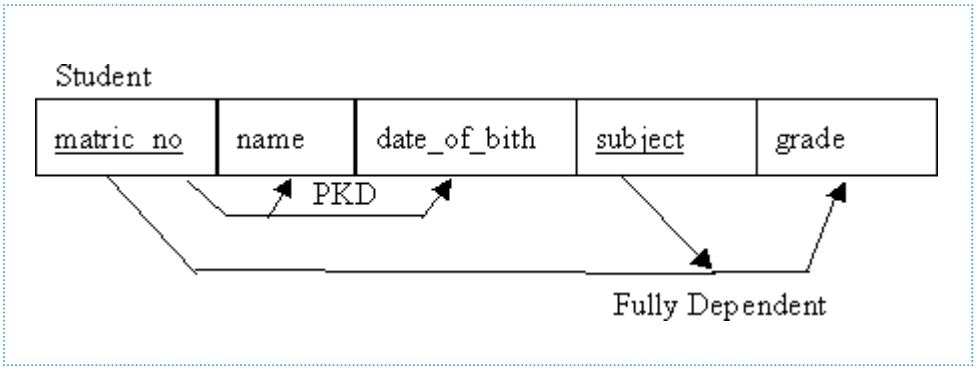
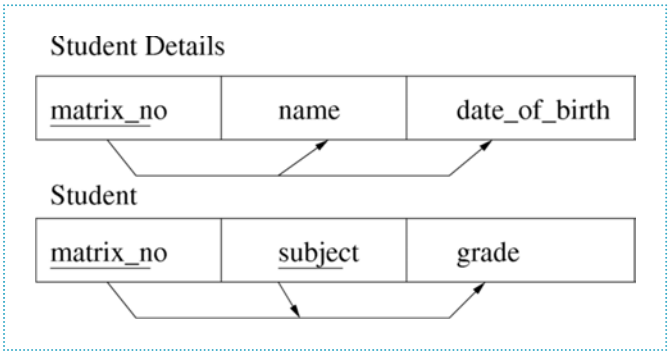


Figure : Dependency Diagram

- This relation is not in 2NF
- It appears to be two tables squashed into one.
- the solution is to split the relation up into its component parts.
- separate out all the attributes that are solely dependent on matric_no
- put them in a new Student_details relation, with matric_no as the primary key
- separate out all the attributes that are solely dependent on subject.
- in this case no attributes are solely dependent on subject.
- separate out all the attributes that are solely dependent on matric_no + subject
- put them into a separate Student relation, keyed on matric_no + subject



All attributes in each relation are fully functionally dependent upon its primary key

These relations are now in 2NF

Figure : Dependencies after splitting

Interestingly this is the same set of relations as when we recognized that there were repeating terms in the table and directly removed the repeating terms. It should not really matter what process you followed when normalizing, as the end result should be similar relations.

Third Normal Form

3NF is an even stricter normal form and removes virtually all the redundant data :

- A relation is in 3NF if, and only if, it is in 2NF and there are no transitive functional dependencies
- Transitive functional dependencies arise:
- when one non-key attribute is functionally dependent on another non-key attribute:
- FD: non-key attribute -> non-key attribute
- and when there is redundancy in the database

By definition transitive functional dependency can only occur if there is more than one non-key field, so we can say that a relation in 2NF with zero or one non-key field must automatically be in 3NF.

project_no	manager	address
p1	Black,B	32 High Street
p2	Smith,J	11 New Street
p3	Black,B	32 High Street
p4	Black,B	32 High Street

Project has more than one non-key field so we must check for transitive dependency:

- address depends on the value in the manager column
- every time B Black is listed in the manager column, the address column has the value `32 High Street'. From this the relation and functional dependency can be implied as:

  Project(project_no, manager, address)

  manager -> address
- in this case address is transitively dependent on manager. Manager is the determinant - it determines the value of address. It is transitive functional dependency only if all attributes on the left of the "->" are not in the key but are all in the relation, and all attributes to the right of the "->" are not in the key with at least one actually being in the relation.
- Data redundancy arises from this
- we duplicate address if a manager is in charge of more than one project
- causes problems if we had to change the address- have to change several entries, and this could lead to errors.
- The solution is to eliminate transitive functional dependency by splitting the table
- create two relations - one with the transitive dependency in it, and another for all of the remaining attributes.
- split Project into Project and Manager.
- the determinant attribute becomes the primary key in the new relation
- manager becomes the primary key to the Manager relation
- the original key is the primary key to the remaining non-transitive attributes
- in this case, project_no remains the key to the new Projects table.

Project

project_no	manager
p1	Black,B
p2	Smith,J
p3	Black,B
p4	Black,B

Manager

manager	address
Black,B	32 High Street
Smith,J	11 New Street

- Now we need to store the address only once
- If we need to know a manager's address we can look it up in the Manager relation
- The manager attribute is the link between the two tables, and in the Projects table it is now a foreign key.
- These relations are now in third normal form.

Summary: 1NF

- A relation is in 1NF if it contains no repeating groups
- To convert an unnormalised relation to 1NF either:
- Flatten the table and change the primary key, or
- Decompose the relation into smaller relations, one for the repeating groups and one for the non-repeating groups.
- Remember to put the primary key from the original relation into both new relations.
- This option is liable to give the best results.

Summary: 2NF

- A relation is in 2NF if it contains no repeating groups and no partial key functional dependencies
- Rule: A relation in 1NF with a single key field must be in 2NF
- To convert a relation with partial functional dependencies to 2NF. create a set of new relations:
- One relation for the attributes that are fully dependent upon the key.
- One relation for each part of the key that has partially dependent attributes

Summary: 3NF

- A relation is in 3NF if it contains no repeating groups, no partial functional dependencies, and no transitive functional dependencies
- To convert a relation with transitive functional dependencies to 3NF, remove the attributes involved in the transitive dependency and put them in a new relation
- Rule: A relation in 2NF with only one non-key attribute must be in 3NF
- In a normalised relation a non-key field must provide a fact about the key, the whole key and nothing but the key.
- Relations in 3NF are sufficient for most practical database design problems. However, 3NF does not guarantee that all anomalies have been removed.

Normalisation - BCNF

Contents

Overview

- normalise a relation to Boyce Codd Normal Form (BCNF)
- Normalisation example

Boyce-Codd Normal Form (BCNF)

- When a relation has more than one candidate key, anomalies may result even though the relation is in 3NF.
- 3NF does not deal satisfactorily with the case of a relation with overlapping candidate keys
- i.e. composite candidate keys with at least one attribute in common.
- BCNF is based on the concept of a *determinant*.
- A determinant is any attribute (simple or composite) on which some other attribute is fully functionally dependent.
- A relation is in BCNF is, and only if, every determinant is a candidate key.

Consider the following relation and determinants.

R(a,b,c,d)
a,c -> b,d
a,d -> b

Here, the first determinant suggests that the primary key of R could be changed from a,b to a,c. If this change was done all of the non-key attributes present in R could still be determined, and therefore this change is legal. However, the second determinant indicates that a,d determines b, but a,d could not be the key of R as a,d does not determine all of the non key attributes of R (it does not determine c). We would say that the first determinate is a candidate key, but the second determinant is not a candidate key, and thus this relation is not in BCNF (but is in 3rd normal form).

Normalisation to BCNF - Example 1

Patient No	Patient Name	Appointment Id	Time	Doctor
1	John	0	09:00	Zorro
2	Kerr	0	09:00	Killer
3	Adam	1	10:00	Zorro
4	Robert	0	13:00	Killer
5	Zane	1	14:00	Zorro

Lets consider the database extract shown above. This depicts a special dieting clinic where the each patient has 4 appointments. On the first they are weighed, the second they are exercised, the third their fat is removed by surgery, and on the fourth their mouth is stitched closed… Not all patients need all four appointments! If the Patient Name begins with a letter before "P" they get a morning appointment, otherwise they get an afternoon appointment. Appointment 1 is either 09:00 or 13:00, appointment 2 10:00 or 14:00, and so on. From this (hopefully) make-believe scenario we can extract the following determinants:

DB(Patno,PatName,appNo,time,doctor)

Patno -> PatName
Patno,appNo -> Time,doctor
Time -> appNo

Now we have to decide what the primary key of DB is going to be. From the information we have, we could chose:
DB(Patno,PatName,appNo,time,doctor) (example 1a)
or
DB(Patno,PatName,appNo,time,doctor) (example 1b)

Example 1a - DB(Patno,PatName,appNo,time,doctor)

- 1NF Eliminate repeating groups.

None:

DB(Patno,PatName,appNo,time,doctor)

- 2NF Eliminate partial key dependencies

```
DB(Patno,appNo,time,doctor)
R1(Patno,PatName)
```

- 3NF Eliminate transitive dependencies

```
None: so just as 2NF
```

- BCNF Every determinant is a candidate key
  DB(Patno,appNo,time,doctor)
  R1(Patno,PatName)
- Go through all determinates where ALL of the left hand attributes are present in a relation and

at least ONE of the right hand attributes are also present in the relation.
- Patno -> PatName
  Patno is present in DB, but not PatName, so not relevant.
- Patno,appNo -> Time,doctor
  All LHS present, and time and doctor also present, so relevant. Is this a candidate key? Patno,appNo IS the key, so this is a candidate key. Thus this is OK for BCNF compliance.
- Time -> appNo
  Time is present, and so is appNo, so relevant. Is this a candidate key. If it was then we could rewrite DB as:
  DB(Patno,appNo,time,doctor)
  This will not work, as you need both time and Patno together to form a unique key. Thus this determinate is not a candidate key, and therefore DB is not in BCNF. We need to fix this.
- BCNF: rewrite to
  DB(Patno,time,doctor)
  R1(Patno,PatName)
  R2(time,appNo)

time is enough to work out the appointment number of a patient. Now BCNF is satisfied, and the final relations shown are in BCNF.

Example 1b - DB(Patno,PatName,appNo,time,doctor)

- 1NF Eliminate repeating groups.

None:

DB(Patno,PatName,appNo,time,doctor)

- 2NF Eliminate partial key dependencies

```
DB(Patno,time,doctor)
R1(Patno,PatName)
R2(time,appNo)
```

- 3NF Eliminate transitive dependencies

```
None: so just as 2NF
```

- BCNF Every determinant is a candidate key

```
DB(Patno,time,doctor)
R1(Patno,PatName)
R2(time,appNo)
```

- Go through all determinates where ALL of the left hand attributes are present in a relation and at least ONE of the right hand attributes are also present in the relation.
- Patno -> PatName
  Patno is present in DB, but not PatName, so not relevant.
- Patno,appNo -> Time,doctor
  Not all LHS present, so not relevant.
- Time -> appNo
  Time is present, and so is appNo, so relevant. This is a candidate key. However, Time is currently the key for R2, so satisfies the rules for BCNF.

- BCNF: as 3NF
  DB(Patno,time,doctor)
  R1(Patno,PatName)
  R2(time,appNo)

Summary - Example 1

This example has demonstrated three things:

- BCNF is stronger than 3NF, relations that are in 3NF are not necessarily in BCNF
- BCNF is needed in certain situations to obtain full understanding of the data model
- there are several routes to take to arrive at the same set of relations in BCNF.
- Unfortunately there are no rules as to which route will be the easiest one to take.

Example 2

```
Grade_report(StudNo,StudName,(Major,Adviser,
 (CourseNo,Ctitle,InstrucName,InstructLocn,Grade)))
```

- Functional dependencies

```
StudNo -> StudName
CourseNo -> Ctitle,InstrucName
InstrucName -> InstrucLocn
StudNo,CourseNo,Major -> Grade
StudNo,Major -> Advisor
Advisor -> Major
```

- Unnormalised

```
Grade_report(StudNo,StudName,(Major,Advisor,
 (CourseNo,Ctitle,InstrucName,InstructLocn,Grade)))
```

- 1NF Remove repeating groups

```
Student(StudNo,StudName)
StudMajor(StudNo,Major,Advisor)
StudCourse(StudNo,Major,CourseNo,
  Ctitle,InstrucName,InstructLocn,Grade)
```

- 2NF Remove partial key dependencies

```
Student(StudNo,StudName)
StudMajor(StudNo,Major,Advisor)
StudCourse(StudNo,Major,CourseNo,Grade)
Course(CourseNo,Ctitle,InstrucName,InstructLocn)
```

- 3NF Remove transitive dependencies

```
Student(StudNo,StudName)
StudMajor(StudNo,Major,Advisor)
StudCourse(StudNo,Major,CourseNo,Grade)
Course(CourseNo,Ctitle,InstrucName)
Instructor(InstructName,InstructLocn)
```

- BCNF Every determinant is a candidate key

- Student : only determinant is StudNo
- StudCourse: only determinant is StudNo,Major
- Course: only determinant is CourseNo
- Instructor: only determinant is InstrucName
- StudMajor: the determinants are
- StudNo,Major, or
- Adviser

Only StudNo,Major is a candidate key.

- BCNF

```
Student(StudNo,StudName)
StudCourse(StudNo,Major,CourseNo,Grade)
Course(CourseNo,Ctitle,InstrucName)
Instructor(InstructName,InstructLocn)
StudMajor(StudNo,Advisor)
Adviser(Adviser,Major)
```

Problems BCNF overcomes

STUDENT	MAJOR	ADVISOR
123	PHYSICS	EINSTEIN
123	MUSIC	MOZART
456	BIOLOGY	DARWIN
789	PHYSICS	BOHR
999	PHYSICS	EINSTEIN

- If the record for student 456 is deleted we lose not only information on student 456 but also the fact that DARWIN advises in BIOLOGY
- we cannot record the fact that WATSON can advise on COMPUTING until we have a student majoring in COMPUTING to whom we can assign WATSON as an advisor.

In BCNF we have two tables:

STUDENT	ADVISOR
123	EINSTEIN
123	MOZART
456	DARWIN
789	BOHR
999	EINSTEIN

ADVISOR	MAJOR
EINSTEIN	PHYSICS
MOZART	MUSIC
DARWIN	BIOLOGY
BOHR	PHYSICS

Returning to the ER Model

- Now that we have reached the end of the normalisation process, you must go back and compare the resulting relations with the original ER model
- You may need to alter it to take account of the changes that have occurred during the normalisation process Your ER diagram should always be a prefect reflection of the model you are going to implement in the database, so keep it up to date!
- The changes required depends on how good the ER model was at first!

Normalisation Example

Library

Consider the case of a simple video library. Each video has a title, director, and serial number. Customers have a name, address, and membership number. Assume only one copy of each video exists in the library. We are given:

```
video(title,director,serial)
customer(name,addr,memberno)
hire(memberno,serial,date)

  title->director,serial
  serial->title
  serial->director
  name,addr -> memberno
  memberno -> name,addr
  serial,date -> memberno
```

What normal form is this?

- No repeating groups, so at least 1NF
- 2NF? There is a composite key in hire. Investigate further... Can memberno in hire be found with just serial or just date. NO. Therefore relation is in at least 2NF.
- 3NF? serial->director is a non-key dependency. Therefore the relations are currently in 2NF.

Convert from 2NF to 3NF.

```
Rewrite
   video(title,director,serial)
To
   video(title,serial)
   serial(serial,director)

Therefore the new relations become:
   video(title,serial)
   serial(serial,director)
   customer(name,addr,memberno)
   hire(memberno,serial,date)
```

In BCNF? Check if every determinant is a candidate key.

```
   video(title,serial)
       Determinants are:
           title->director,serial  Candidate key
           serial->title           Candidate key
       video in BCNF
```

```
serial(serial,director)
    Determinants are:
        serial->director          Candidate key
    serial in BCNF

customer(name,addr,memberno)
    Determinants are:
       name,addr -> memberno      Candidate key
       memberno -> name,addr      Candidate key
    customer in BCNF

hire(memberno,serial,date)
    Determinants are:
       serial,date -> memberno    Candidate key
    hire in BCNF
```

Therefore the relations are also now in BCNF.

Chapter 5 - Relational Algebra

Relational algebra introduction, including a full algebraic syntax.

Relational Algebra

Contents

In order to implement a DBMS, there must exist a set of rules which state how the database system will behave. For instance, somewhere in the DBMS must be a set of statements which indicate than when someone inserts data into a row of a relation, it has the effect which the user expects. One way to specify this is to use words to write an `essay' as to how the DBMS will operate, but words tend to be imprecise and open to interpretation. Instead, relational databases are more usually defined using Relational Algebra.

Relational Algebra is :

- the formal description of how a relational database operates
- an interface to the data stored in the database itself
- the mathematics which underpin SQL operations

Operators in relational algebra are not necessarily the same as SQL operators, even if they have the same name. For example, the SELECT statement exists in SQL, and also exists in relational algebra. These two uses of SELECT are not the same. The DBMS must take whatever SQL statements the user types in and translate them into relational algebra operations before applying them to the database.

Terminology

- Relation - a set of tuples.
- Tuple - a collection of attributes which describe some real world entity.
- Attribute - a real world role played by a named domain.
- Domain - a set of atomic values.
- Set - a mathematical definition for a collection of objects which contains no duplicates.

Operators - Write

- INSERT - provides a list of attribute values for a new tuple in a relation. This operator is the same as SQL.
- DELETE - provides a condition on the attributes of a relation to determine which tuple(s) to remove from the relation. This operator is the same as SQL.
- MODIFY - changes the values of one or more attributes in one or more tuples of a relation, as identified by a condition operating on the attributes of the relation. This is equivalent to SQL UPDATE.

Operators - Retrieval

There are two groups of operations:

- Mathematical set theory based relations:
  UNION, INTERSECTION, DIFFERENCE, and CARTESIAN PRODUCT.
- Special database operations:
  SELECT (not the same as SQL SELECT), PROJECT, and JOIN.

Relational SELECT

SELECT is used to obtain a subset of the tuples of a relation that satisfy a *select condition*.

For example, find all employees born after 1st Jan 1950:

$$SELECT_{dob\ '01/JAN/1950'}(employee)$$

Relational PROJECT

The PROJECT operation is used to select a subset of the attributes of a relation by specifying the names of the required attributes.

For example, to get a list of all employees surnames and employee numbers:

$$PROJECT_{surname,empno}(employee)$$

SELECT and PROJECT

SELECT and PROJECT can be combined together. For example, to get a list of employee numbers for employees in department number 1:

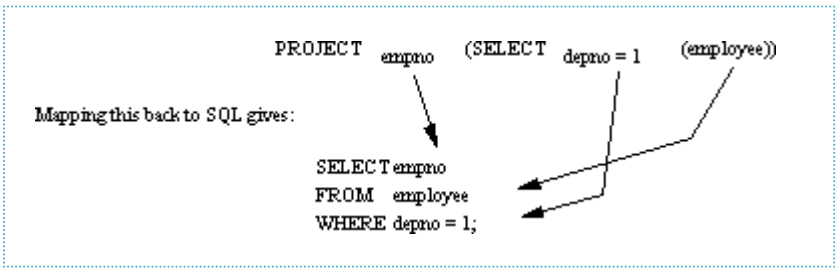


Figure : Mapping select and project

Set Operations - semantics

Consider two relations R and S.

- UNION of R and S
  the union of two relations is a relation that includes all the tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
- INTERSECTION of R and S
  the intersection of R and S is a relation that includes all tuples that are both in R and S.
- DIFFERENCE of R and S
  the difference of R and S is the relation that contains all the tuples that are in R but that are not in S.

SET Operations - requirements

For set operations to function correctly the relations R and S must be union compatible. Two relations are union compatible if

- they have the same number of attributes
- the domain of each attribute in column order is the same in both R and S.

UNION Example

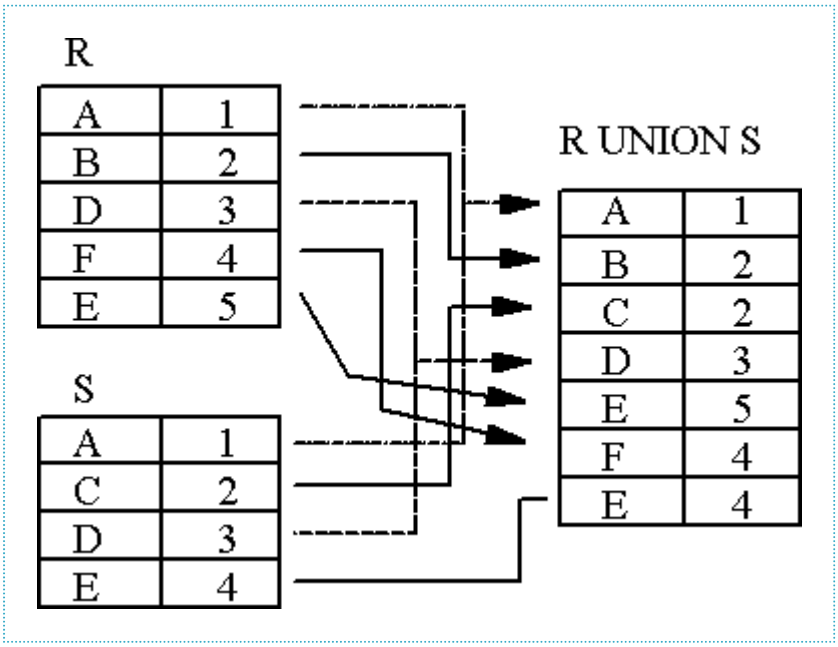


Figure : UNION

INTERSECTION Example

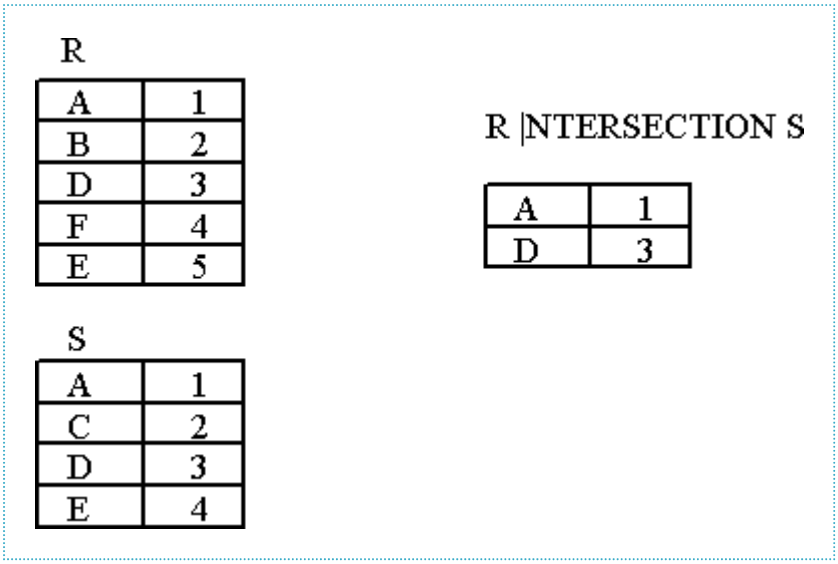


Figure : Intersection

DIFFERENCE Example

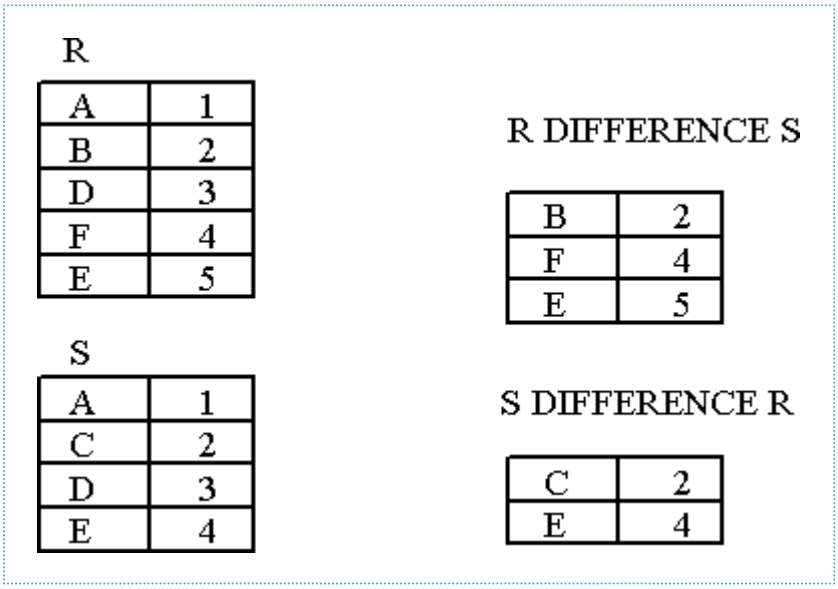


Figure : DIFFERENCE

CARTESIAN PRODUCT

The Cartesian Product is also an operator which works on two sets. It is sometimes called the CROSS PRODUCT or CROSS JOIN.

It combines the tuples of one relation with all the tuples of the other relation.

CARTESIAN PRODUCT example

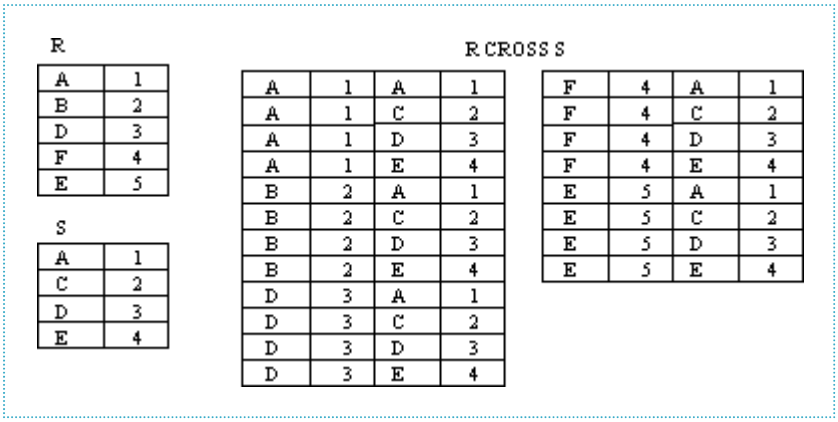


Figure : CARTESIAN PRODUCT

JOIN Operator

JOIN is used to combine related tuples from two relations:

- In its simplest form the JOIN operator is just the cross product of the two relations.
- As the join becomes more complex, tuples are removed within the cross product to make the result of the join more meaningful.
- JOIN allows you to evaluate a join condition between the attributes of the relations on which the join is undertaken.

The notation used is

$$R \text{ JOIN}_{\text{join condition}} S$$

JOIN Example

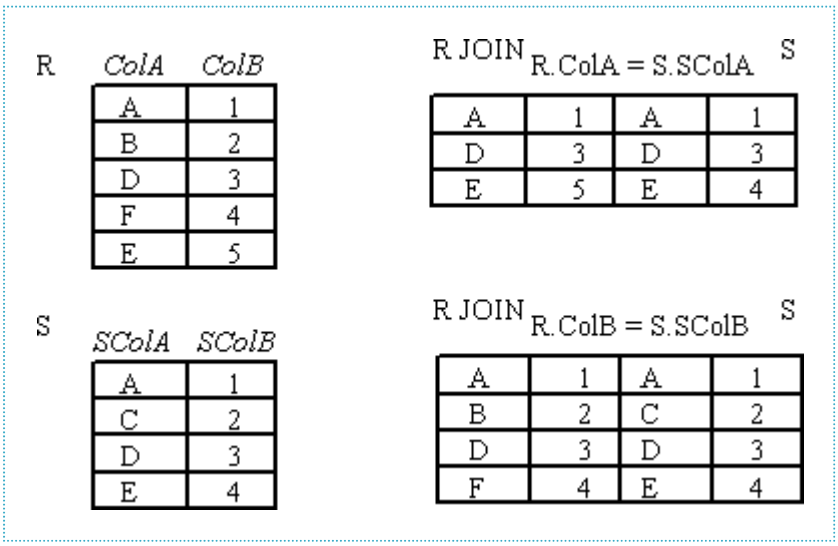


Figure : JOIN

Natural Join

Invariably the JOIN involves an equality test, and thus is often described as an equi-join. Such joins result in two attributes in the resulting relation having exactly the same value. A `natural join' will

remove the duplicate attribute(s).

- In most systems a natural join will require that the attributes have the same name to identify the attribute(s) to be used in the join. This may require a renaming mechanism.
- If you do use natural joins make sure that the relations do not have two attributes with the same name by accident.

OUTER JOINs

Notice that much of the data is lost when applying a join to two relations. In some cases this lost data might hold useful information. An outer join retains the information that would have been lost from the tables, replacing missing data with nulls.

There are three forms of the outer join, depending on which data is to be kept.

- LEFT OUTER JOIN - keep data from the left-hand table
- RIGHT OUTER JOIN - keep data from the right-hand table
- FULL OUTER JOIN - keep data from both tables

OUTER JOIN example 1

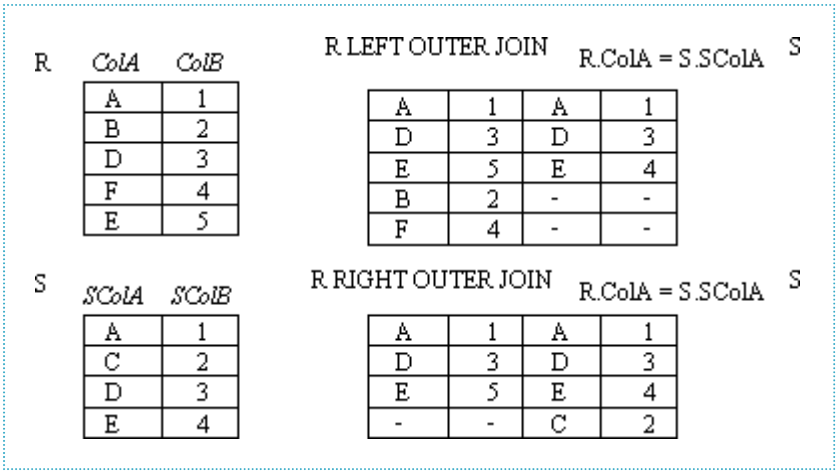


Figure : OUTER JOIN (left/right)

OUTER JOIN example 2

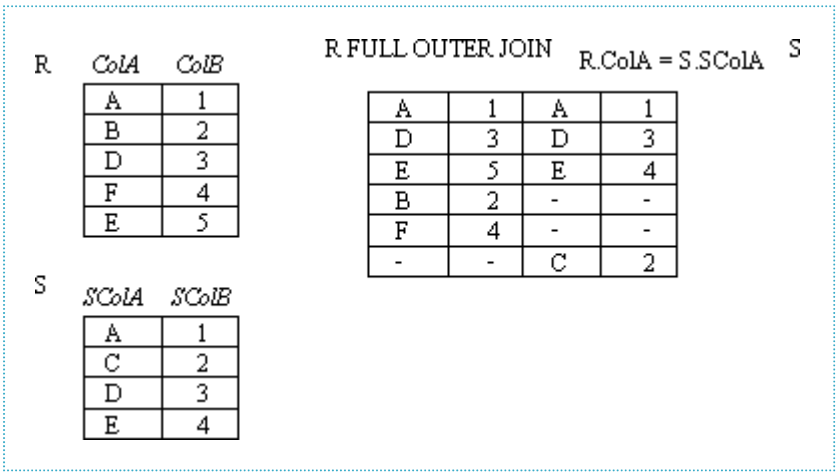


Figure : OUTER JOIN (full)

Relational Algebra - Example

Contents

Consider the following SQL to find which departments have had employees on the `Further Accounting' course.

```
SELECT DISTINCT dname
FROM  department, course, empcourse, employee
WHERE cname = `Further Accounting'
  AND course.courseno = empcourse.courseno
  AND empcourse.empno = employee.empno
  AND employee.depno = department.depno;
```

The equivalent relational algebra is

$$\text{PROJECT}_{\text{dname}}\ (\text{department JOIN}_{\text{depno = depno}}\ (\text{PROJECT}_{\text{depno}}\ (\text{employee JOIN}_{\text{empno = empno}}\ (\text{PROJECT}_{\text{empno}}\ (\text{empcourse JOIN}_{\text{courseno = courseno}}\ (\text{PROJECT}_{\text{courseno}}\ (\text{SELECT}_{\text{cname = `Further Accounting'}}\ \text{course})))))))$$

Symbolic Notation

From the example, one can see that for complicated cases a large amount of the answer is formed from operator names, such as PROJECT and JOIN. It is therefore commonplace to use symbolic notation to represent the operators.

- SELECT ->σ (sigma)
- PROJECT -> π(pi)
- PRODUCT -> ×(times)
- JOIN -> |×| (bow-tie)
- UNION -> ∪ (cup)
- INTERSECTION -> ∩(cap)
- DIFFERENCE -> - (minus)
- RENAME ->ρ (rho)

Usage

The symbolic operators are used as with the verbal ones. So, to find all employees in department 1:

$SELECT_{depno = 1}(employee)$
becomes $\sigma_{depno = 1}(employee)$

Conditions can be combined together using ^ (AND) and v (OR). For example, all employees in department 1 called `Smith':

$SELECT_{depno = 1 \ ^ \ surname = `Smith'}(employee)$
becomes $\sigma_{depno = 1 \ ^ \ surname = `Smith'}(employee)$

The use of the symbolic notation can lend itself to brevity. Even better, when the JOIN is a natural join, the JOIN condition may be omitted from |x|. The earlier example resulted in:

$PROJECT_{dname}$ (department $JOIN_{depno = depno}$ (
$PROJECT_{depno}$ (employee $JOIN_{empno = empno}$ (
$PROJECT_{empno}$ (empcourse $JOIN_{courseno = courseno}$ (
$PROJECT_{courseno}$ ($SELECT_{cname = `Further Accounting'}$ course)))))))

becomes

π_{dname} (department |×| (
π_{depno} (employee |×| (
π_{empno} (empcourse |×| (
$\pi_{courseno}$ ($\sigma_{cname = `Further Accounting'}$ course)))))))

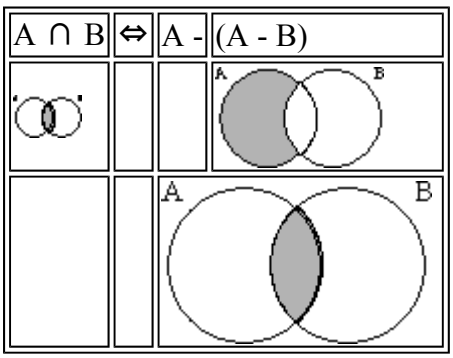
Rename Operator

The rename operator returns an existing relation under a new name. $\rho_A(B)$ is the relation B with its name changed to A. For example, find the employees in the same Department as employee 3.

$\rho_{emp2.surname,emp2.forenames}$ (
$\sigma_{employee.empno = 3 \ ^ \ employee.depno = emp2.depno}$ (
employee × (ρ_{emp2}employee)
)
)

Derivable Operators

- Fundamental operators:σ, π, ×, ∪, -, ρ
- Derivable operators: |×|, ∩



Equivalence

$A |\times|_c B \Leftrightarrow \pi_{a1,a2,...aN}(\sigma_c(A \times B))$

- where c is the join condition (eg A.a1 = B.a1),
- and a1,a2,...aN are all the attributes of A and B without repetition.

c is called the join-condition, and is usually the comparison of primary and foreign key. Where there are N tables, there are usually N-1 join-conditions. In the case of a natural join, the conditions can be missed out, but otherwise missing out conditions results in a cartesian product (a common mistake to make).

Equivalences

The same relational algebraic expression can be written in many different ways. The order in which tuples appear in relations is never significant.

- $A \times B \Leftrightarrow B \times A$
- $A \cap B \Leftrightarrow B \cap A$
- $A \cup B \Leftrightarrow B \cup A$
- (A - B) is not the same as (B - A)
- $\sigma_{c1}(\sigma_{c2}(A)) \Leftrightarrow \sigma_{c2}(\sigma_{c1}(A)) \Leftrightarrow \sigma_{c1 \wedge c2}(A)$
- $\pi_{a1}(A) \Leftrightarrow \pi_{a1}(\pi_{a1,etc}(A))$
  where etc represents any other attributes of A.
- many other equivalences exist.

While equivalent expressions always give the same result, some may be much easier to evaluate that others.

When any query is submitted to the DBMS, its query optimiser tries to find the most efficient equivalent expression before evaluating it.

Comparing RA and SQL

- Relational algebra:
- is closed (the result of every expression is a relation)
- has a rigorous foundation
- has simple semantics
- is used for reasoning, query optimisation, etc.
- SQL:
- is a superset of relational algebra
- has convenient formatting features, etc.
- provides aggregate functions
- has complicated semantics
- is an end-user language.

Comparing RA and SQL

Any relational language as powerful as relational algebra is called relationally complete.

A relationally complete language can perform all basic, meaningful operations on relations.

Since SQL is a superset of relational algebra, it is also relationally complete.

Chapter 6 - Implementations

Coverage of concurrency requirements, concurrency control through locking and transactions, and storage structure implementation approaches.

Concurrency using Transactions

Contents

The goal in a `concurrent' DBMS is to allow multiple users to access the database simultaneously without interfering with each other.

A problem with multiple users using the DBMS is that it may be possible for two users to try and change data in the database simultaneously. If this type of action is not carefully controlled, inconsistencies are possible.

To control data access, we first need a concept to allow us to encapsulate database accesses. Such encapsulation is called a `Transaction'.

Transactions

- Transaction (ACID)
- unit of logical work and recovery
  - **A** - atomicity (for integrity)
  - **C** - consistency preservation
  - **I** - isolation
  - **D** - durability
- Available in SQL
- Some applications require nested or long transactions

After work is performed in a transaction, two outcomes are possible:

- Commit - Any changes made during the transaction by this transaction are committed to the database.
- Abort - All the changes made during the transaction by this transaction are not made to the database. The result of this is as if the transaction was never started.

Transaction Schedules

A transaction schedule is a tabular representation of how a set of transactions were executed over time. This is useful when examining problem scenarios. Within the diagrams various nomenclatures are used:

- READ(*a*) - This is a read action on an attribute or data item called `a'.
- WRITE(*x*,*a*) - This is a write action on an attribute or data item called `a', where the value `x' is written into `a'.
- tn (e.g. t1,t2,t10) - This indicates the time at which something occurred. The units are not important, but t*n* always occurs before t*n+1*.

Consider transaction A, which loads in a bank account balance X (initially 20) and adds 10 pounds to it. Such a schedule would look like this:

Time	Transaction A
t1	TOTAL:=READ(X)
t2	TOTAL:=TOTAL+10
t3	WRITE(TOTAL,X)

Now consider that, at the same time as transaction A runs, transaction B runs. Transaction B gives all accounts a 10% increase. Will X be 32 or 33?

Time	Transaction A	Value TOTAL	Transaction B	Value BALANCE
t1			BALANCE:=READ(X)	20
t2	TOTAL:=READ(X)	20		
t3	TOTAL:=TOTAL+10	30		
t4	WRITE(TOTAL,X)	30		
t5			BALANCE:=BALANCE*110%	22
t6			WRITE(BALANCE,X)	22

Whoops... X is 22! Depending on the interleaving, X can also be 32, 33, or 30. Lets classify erroneous scenarios.

Lost Update scenario.

Time	Transaction A	Transaction B
t1	X=READ(R)	
t2		Y=READ(R)
t3	WRITE(X,R)	
t4		WRITE(Y,R)

Transaction A's update is lost at t4, because Transaction B overwrites it. B missed A's update at t3 as it got the value of R at t2.

Uncommitted Dependency

Time	Transaction A	Transaction B
t1		WRITE(X,R)
t2	Y=READ(R)	
t3		ABORT

Transaction A is allowed to READ (or WRITE) item R which has been updated by another transaction but not committed (and in this case ABORTed).

Inconsistency

Time	X	Y	Z	Transaction A		Transaction B
				Action	SUM	
t1	40	50	30	SUM:=READ(X)	40	
t2	40	50	30	SUM+=READ(Y)	90	
t3	40	50	30			ACC1=READ(Z)
t4	40	50	20			WRITE(ACC1-10,Z)
t5	40	50	20			ACC2=READ(X)
t6	50	50	20			WRITE(AC2+10,X)
t7	50	50	20			COMMIT
t7	50	50	20	SUM+=READ(Z)	110	
				SUM should have been 120...		

Serialisability

- A `schedule' is the actual execution sequence of two or more concurrent transactions.
- A schedule of two transactions T1 and T2 is `serialisable' if and only if executing this schedule has the same effect as either T1;T2 or T2;T1.

Precedence Graph

In order to know that a particular transaction schedule can be serialized, we can draw a precedence graph. This is a graph of nodes and vertices, where the nodes are the transaction names and the vertices are attribute collisions.

The schedule is said to be serialised if and only if there are no cycles in the resulting diagram.

Precedence Graph : Method

To draw one;

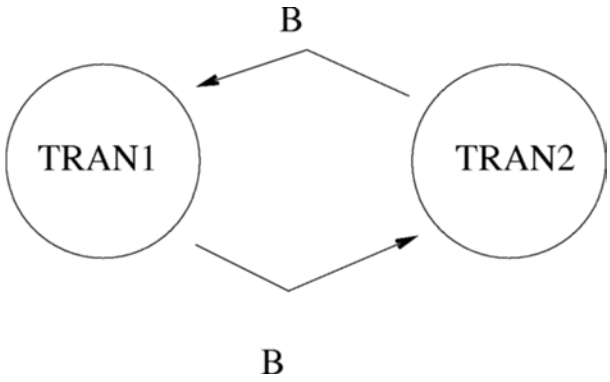
- Draw a node for each transaction in the schedule
- Where transaction A writes to an attribute which transaction B has read from, draw an line pointing from B to A.

- Where transaction A writes to an attribute which transaction B has written to, draw a line pointing from B to A.
- Where transaction A reads from an attribute which transaction B has written to, draw a line pointing from B to A.

Example 1

Consider the following schedule:

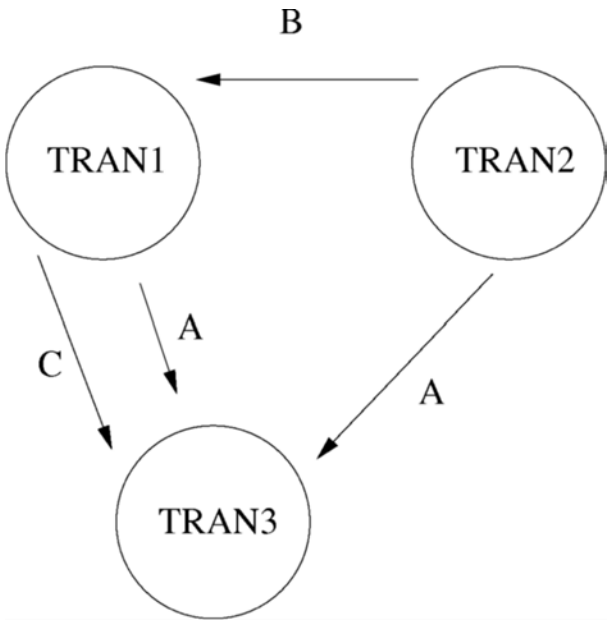
Time	TRAN1	TRAN2
t1	READ(A)	
t2	READ(B)	
t3		READ(A)
t4		READ(B)
t5	WRITE(x,B)	
t6		WRITE(y,B)



Example 2

Consider the following schedule:

Time	TRAN1	TRAN2	TRAN3
t1	READ(A)		
t2	READ(B)		
t3		READ(A)	
t4		READ(B)	
t5			WRITE(x,A)
t6	WRITE(v,C)		
t7	WRITE(w,B)		
t8			WRITE(z,C)



Concurrency

Contents

Locking

A solution to enforcing serialisability?

- read (shareable) lock
- write (exclusive) lock
- coarse granularity
- easier processing
- less concurrency
- fine granularity
- more processing
- higher concurrency

Many systems use locking mechanisms for concurrency control. When a transaction needs an assurance that some object will not change in some unpredictable manner, it acquires a lock on that object.

- A transaction holding a read lock is permitted to read an object but not to change it.
- More than one transaction can hold a read lock for the same object.
- Usually, only one transaction may hold a write lock on an object.
- On a transaction schedule, we use `S' to indicate a shared lock, and `X' for an exclusive write lock.

Locking - Uncommitted Dependency

Locking solves the uncommitted dependency problem.

Time	Transaction A	Transaction B	Lock on R
t1		WRITE(p,R)	- = X
t2	READ R (WAIT)		X
t3	...wait...	ABORT	X = -
t4	READ R (CONT)		- = S

Deadlock

Deadlock can arise when locks are used, and causes all related transactions to WAIT forever...

time	Transaction A	Transaction B	Lock State	
			X	**Y**
t1	WRITE(p,X)		- = X	-
t2		WRITE(q,Y)	X	- = X
t3	READ(Y) (WAIT)		X	X
t3	...WAIT...	READ(X) (WAIT)	X	X
t3	...WAIT...	...WAIT...	X	X

The `lost update' senario results in deadlock with locks. So does the `inconsistency' scenario.

time	Transaction A	Transaction B	Lock R
t1	READ(R)		- = S
t2		READ(R)	S = S
t3	WRITE(p,R) (WAIT)		S
t3	...wait...	WRITE(q,R) (WAIT)	S
t3	...wait...	...wait...	S

Deadlock Handling

- Deadlock avoidance
  - pre-claim strategy used in operating systems
  - not effective in database environments.
- Deadlock detection
  - whenever a lock requests a wait, or on some perodic basis.
  - if a transaction is blocked due to another transaction, make sure that that transaction is not blocked on the first transaction, either directly or indirectly via another transaction.

Deadlock Resolution

If a set of transactions is considered to be deadlocked:

- choose a victim (e.g. the shortest-lived transaction)
- rollback `victim' transaction and restart it.
- The rollback terminates the transaction, undoing all its updates and releasing all of its locks.
- A message is passed to the victim and depending on the system the transaction may or may not be started again automatically.

Two-Phase Locking

The presence of locks does not guarantee serialisability. If a transaction is allowed to release locks before the transaction has completed, and is also allowed to acquire more (or even the same) locks

later then the benifit or locking is lost.

If all transactions obey the `two-phase locking protocol', then all possible interleaved executions are guaranteed serialisable.

The two-phase locking protocol:

- Before operating on any item, a transaction must acquire at least a shared lock on that item. Thus no item can be accessed without first obtaining the correct lock.
- After releasing a lock, a transaction must never go on to acquire any more locks.

The technical names for the two phases of the locking protocol are the `lock-acquisition phase' and the `lock-release phase'.

Other Database Consistency Methods

Two-phase locking is not the only approach to enforcing database consistency. Another method used in some DMBS is timestamping. With timestamping, there are no locks to prevent transactions seeing uncommitted changes, and all physical updates are deferred to commit time.

- locking synchronises the interleaved execution of a set of transactions in such a way that it is equivalent to some serial execution of those transactions.
- timestamping synchronises that interleaved execution in such a way that it is equivalent to a particular serial order - the order of the timestamps.

Timestamping rules

The following rules are checked when transaction T attempts to change a data item. If the rule indicates ABORT, then transaction T is rolled back and aborted (and perhaps restarted).

- If T attempts to read a data item which has already been written to by a younger transaction then ABORT T.
- If T attempts to write a data item which has been seen or written to by a younger transaction then ABORT T.

If transaction T aborts, then all other transactions which have seen a data item written to by T must also abort. In addition, other aborting transactions can cause further aborts on other transactions. This is a `cascading rollback'.

Recovery

Contents

A database might be left in an inconsistent state when:

- deadlock has occurred.
- a transaction aborts after updating the database.
- software or hardware errors.
- incorrect updates have been applied to the database.

If the database is in an inconsistent state, it is necessary to recover to a consistent state. The basis of recovery is to have backups of the data in the database.

Recovery: the dump

The simplest backup technique is `the Dump'.

- entire contents of the database is backed up to an auxiliary store.
- must be performed when the state of the database is consistent - therefore no transactions which modify the database can be running
- dumping can take a long time to perform
- you need to store the data in the database twice.
- as dumping is expensive, it probably cannot be performed as often as one would like.
- a cut-down version can be used to take `snapshots' of the most volatile areas.

Recovery: the transaction log

A technique often used to perform recovery is the transaction log or journal.

- records information about the progress of transactions in a log since the last consistent state.
- the database therefore knows the state of the database before and after each transaction.
- every so often database is returned to a consistent state and the log may be truncated to remove committed transactions.
- when the database is returned to a consistent state the process is often referred to as `checkpointing'.

Deferred Update

Deferred update, or NO-UNDO/REDO, is an algorithm to support ABORT and machine failure scenarios.

- While a transaction runs, no changes made by that transaction are recorded in the database.
- On a commit:
- The new data is recorded in a log file and flushed to disk
- The new data is then recorded in the database itself.
- On an abort, do nothing (the database has not been changed).
- On a system restart after a failure, REDO the log.

Example

Consider the following transaction TRAN1

Transaction TRAN1
read(A)
write(10,B)
write(20,C)
Commit

Using deferred update, the process is:

Time	**Action**	**Log**
t1	START	-
t2	read(A)	-
t3	write(10,B)	B = 10
t4	write(20,C)	C = 20
t5	COMMIT	COMMIT

	Before			**After**		
DISK			B = 6			B = 10
	A = 5	C = 2		A = 5	C = 20	

If the DMBS fails and is restarted:

- The disks are physically or logically damaged then recovery from the log is impossible and instead a restore from a dump is needed.
- If the disks are OK then the database consistency must be maintained. Writes to the disk which was in progress at the time of the failure may have only been partially done.
- Parse the log file, and where a transaction has been ended with `COMMIT' apply the data part of the log to the database.
- If a log entry for a transaction ends with anything other than COMMIT, do nothing for that transaction.
- flush the data to the disk, and then truncate the log to zero.
- the process or reapplying transaction from the log is sometimes referred to as `rollforward'.

Immediate Update

Immediate update, or UNDO/REDO, is another algorithm to support ABORT and machine failure scenarios.

- While a transaction runs, changes made by that transaction can be written to the database at any time. However, the original and the new data being written must both be stored in the log BEFORE storing it on the database disk.
- On a commit:
- All the updates which has not yet been recorded on the disk is first stored in the log file and then flushed to disk.
- The new data is then recorded in the database itself.
- On an abort, REDO all the changes which that transaction has made to the database disk using the log entries.
- On a system restart after a failure, REDO committed changes from log.

Example

Using immediate update, and the transaction TRAN1 again, the process is:

Time	Action	LOG
t1	START	-
t2	read(A)	-
t3	write(10,B)	Was B == 6, now 10
t4	write(20,C)	Was C == 2, now 20
t5	COMMIT	COMMIT

	Before			During			After		
DISK			B = 6			B = 10			B = 10
	A = 5	C = 2		A = 5	C = 2		A = 5	C = 20	

If the DMBS fails and is restarted:

- The disks are physically or logically damaged then recovery from the log is impossible and instead a restore from a dump is needed.
- If the disks are OK then the database consistency must be maintained. Writes to the disk which was in progress at the time of the failure may have only been partially done.
- Parse the log file, and where a transaction has been ended with `COMMIT' apply the `new data' part of the log to the database.
- If a log entry for a transaction ends with anything other than COMMIT, apply the `old data' part of the log to the database.
- flush the data to the disk, and then truncate the log to zero.

Rollback

The process of undoing changes done to the disk under immediate update is frequently referred to as rollback.

- Where the DBMS does not prevent one transaction from reading uncommitted modifications (a `dirty read') of another transaction (i.e. the uncommitted dependency problem) then aborting the first transaction also means aborting all the transactions which have performed these dirty reads.
- as a transaction is aborted, it can therefore cause aborts in other dirty reader transactions,

which in turn can cause other aborts in other dirty reader transaction. This is referred to as `cascade rollback'.

DBMS Implementation

Contents

Implementing a DBMS

A database management system handles the requests generated from the SQL interface, producing or modifying data in response to these requests. This involves a multilevel processing system.

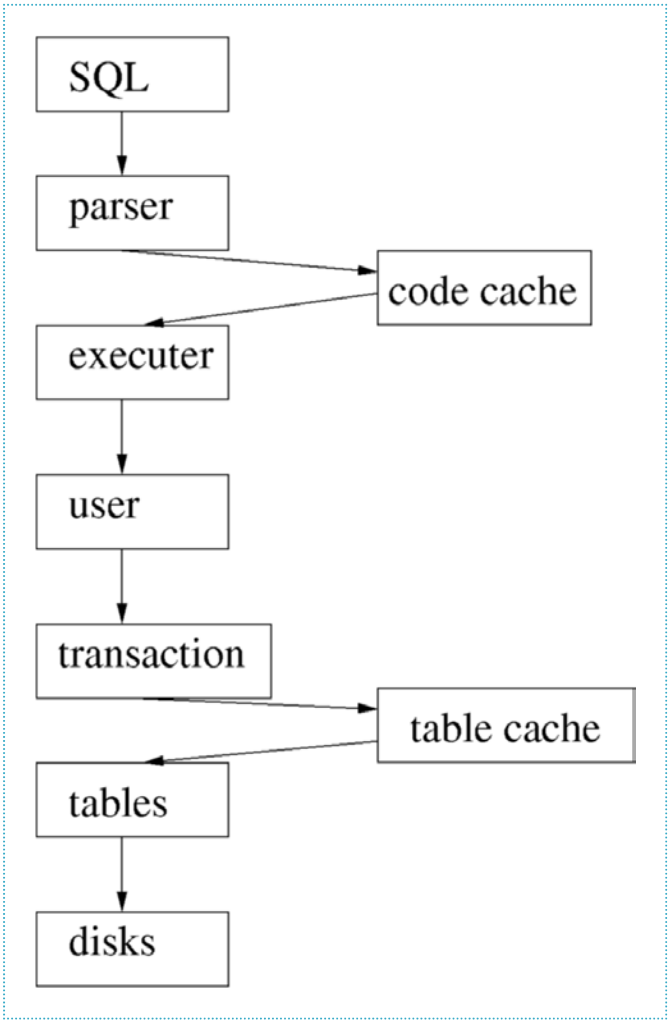


Figure : DBMS Execution and Parsing

This level structure processes the SQL submitted by the user or application.

- Parser: The SQL must be parsed and tokenised. Syntax errors are reported back to the user. Parsing can be time consuming, so good quality DBMS implementations cache queries after they have been parsed so that if the same query is submitted again the cached copy can be used instead. To make the best use of this most systems use placeholders in queries, like:

  ```
  SELECT empno FROM employee where surname = ?
  ```

  The '?' character is prompted for when the query is executed, and can be supplied separately from the query by the API used to inject the SQL. The parameter is not part of the parsing process, and thus once this query is parsed once it need not be parsed again.
- Executer: This takes the SQL tokens and basically translates it into relational algebra. Each relational algebra fragment is optimised, and the passed down the levels to be acted on.
- User: The concept of the user is required at this stage. This gives the query context, and also allows security to be implemented on a per-user basis.
- Transactions: The queries are executed in the transaction model. The same query from the same user can be executing multiple times in different transactions. Each transaction is quite separate.
- Tables : The idea of the table structure is controlled at a low level. Much security is based on the concept of tables, and the schema itself is stored in tables, as well as being a set of tables itself.
- Table cache: Disks are slow, yet a disk is the best way of storing long-term data. Memory is much faster, so it makes sense to keep as much table information as possible in memory. The disk remains synchronised to memory as part of the transaction control system.
- Disks : Underlying almost all database systems is the disk storage system. This provides storage for the DBMS system tables, user information, schema definitions, and the user data itself. It also provides the means for transaction logging.

The 'user' context is handled in a number of different ways, depending on the database system being used. The following diagram gives you an idea of the approach followed by two different systems, Oracle and MySQL.

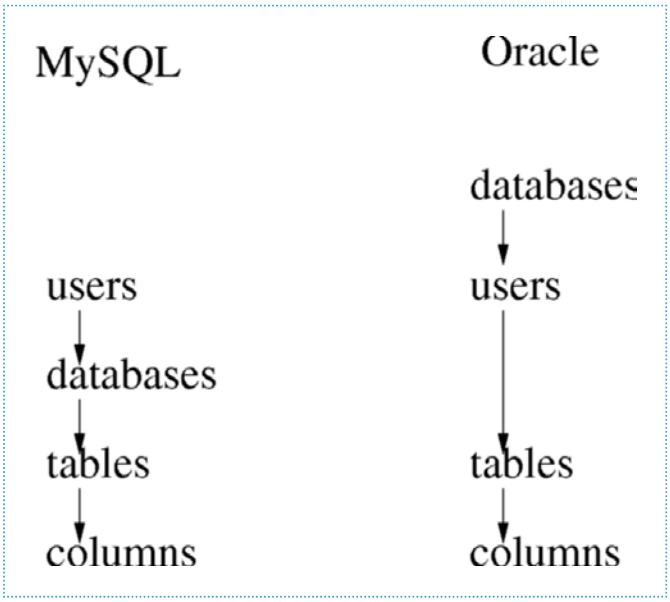


Figure : Users and Tablespaces

All users in a system have login names and passwords. In Oracle, during the connection phase, a database name must be provided. This allows one Oracle DBMS to run multiple databases, each of which is effectively isolated from each other.

Once a user is connected using a username and password, MySQL places the user in a particular tablespace in the database. The name of the tablespace is independent of the same. In Oracle, tablespaces and usernames are synonymous, and thus you should really be thinking of different usernames for databases that serve different purposes. In MySQL the philosophy is more like a username is a person, and that person may want to do a variety of tasks.

Once in a tablespace, a number of tables are visible, and in each table columns are visible.

In both approaches, tables in other tablespaces can be accessed. MySQL effectively sees a tablespace and a database being the same concept, but in Oracle the two ideas are kept slightly more separate. However, the syntax remains the same. Just as you can access column owner of table CAR, if it is in your own tablespace, by saying

```
SELECT car.owner FROM car;
```

You can access table CAR in another tablespace (lets call it vehicles) by saying:

```
SELECT vehicles.car.owner FROM vehicles.car;
```

The appearance of this structure is similar in concept to the idea of file directories. In a database the directories are limited to "folder.table.column", although "folder" could be a username, a tablename, or a database, depending on the philosophy of the database management system. Even then, the concept is largely similar.

Disk and Memory

The tradeoff between the DBMS using Disk or using main memory should be understood...

Issue	Main Memory VS Disk
Speed	Main memory is at least 1000 times faster than Disk
Storage Space	Disk can hold hundreds of times more information than memory for the same cost
Persistence	When the power is switched off, Disk keeps the data, main memory forgets everything
Access Time	Main memory starts sending data in nanoseconds, while disk takes milliseconds
Block Size	Main memory can be accessed 1 word at a time, Disk 1 block at a time

The DBMS runs in main memory, and the processor can only access data which is currently in main memory. The handling of the differences between disk and main memory effectively is at the heart of a good quality DBMS.

Disk Arrangements

Efficient processing of the DBMS requests requires efficient handling of disk storage. The important aspects of this include:

- Index handling

- Transaction Log management
- Parallel Disk Requests
- Data prediction

With indexing, we are concerned with finding the data we actually want quickly and efficiently, without having to request and read more disk blocks than absolutely necessary. There are many approaches to this, but two of the more important ones are hash tables and binary trees.

When handling transaction logs, the discussion we have had so far has been on the theory of these techniques. In practice, the separation of data and log is much more blurred. We will look at one technique for implementing transaction logging, known as shadow paging.

Finally, the underlying desire of a good DBMS is to never be in a position where no further work can be done until the disk gives us some data. Instead, by using prediction, prefetching, and parallel disk operations, it is hoped that CPU time becomes the limiting factor.

Hash tables

A Hash table is one of the simplest index structures which a database can implement. The major components of a hash index is the "hash function" and the "buckets". Effectively the DBMS constructs an index for every table you create that has a PRIMARY KEY attribute, like:

```
CREATE TABLE test (
  id   INTEGER PRIMARY KEY
 ,name varchar(100)
);
```

In table test, we have decided to store 4 rows...

```
insert into test values (1,'Gordon');
insert into test values (2,'Jim');
insert into test values (4,'Andrew');
insert into test values (3,'John');
```

The algorithm splits the places which the rows are to be stored into areas. These areas are called buckets. If a row's primary key matches the requirements to be stored in that bucket, then that is where it will be stored. The algorithm to decide which bucket to use is called the hash function. For our example we will have a nice simple hash function, where the bucket number equals the primary key. When the index is created we have to also decide how many buckets there are. In this example we have decided on 4.

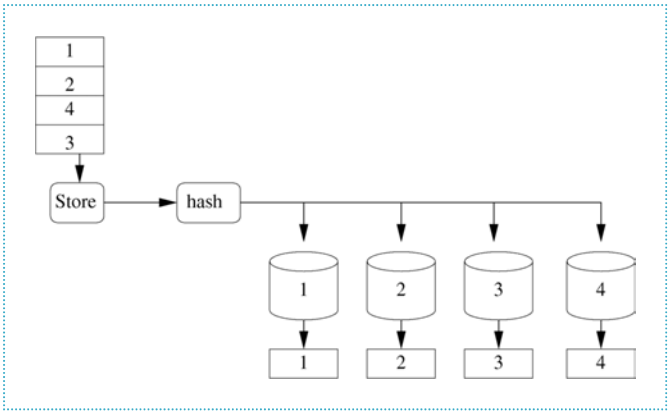


Figure : Hash Table with no collisions

Now we can find id 3 quickly and easily by visiting bucket 3 and looking into it. But now the buckets are full. To add more values we will have to reuse buckets. We need a better hash function based on mod 4. Now bucket 1 holds ids (1,5,9...), bucket 2 holds (2,6,10...), etc.

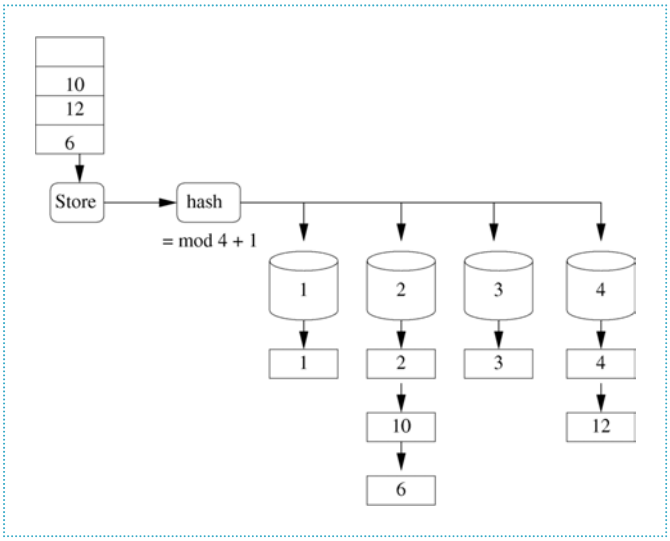


Figure : Hash Table with collisions

We have had to put more than 1 row in some of the buckets. This is called a hash collision. The more collisions we have the longer the collision chain and the slower the system will get. For instance, finding id 6 means visiting bucket 2, and then finding id 2, then 10, and then finally 6.

In DBMS systems we can usually ask for a hash index for a table, and also say how many buckets we thing we will need. This approach is good when we know how many rows there is likely to be. Most systems will handle the hash table for you, adding more buckets over time if you have made a mistake. It remains a popular indexing technique.

Binary Tree

Binary trees is the latest approach to providing indexes. It is much cleverer than hash tables, and attempts to solve the problem of not knowing how many buckets you might need, and that some collision chains might be much longer than others. It attempts to create indexes such that all rows can be found in a similar number of steps through the storage blocks.

The state of the art in binary tree technology is called B+ Trees. With B+ tree, the order of the original data is maintained in its creation order. This allows multiple B+ tree indices to be kept for the same set of data records.

- the lowest level in the index has one entry for each data record.
- the index is created dynamically as data is added to the file.
- as data is added the index is expanded such that each record requires the same number of index levels to reach it (thus the tree stays `balanced').
- the records can be accessed via an index or sequentially.

Each index node in a B+ Tree can hold a certain number of keys. The number of keys is often referred to as the `order'. Unfortunately, `Order 2' and `Order 1' are frequently confused in the database literature. For the purposes of our coursework and exam, `Order 2' means that there can be a maximum of 2 keys per index node. In this module, we only ever consider order 2 B+ trees.

B+ Tree Example

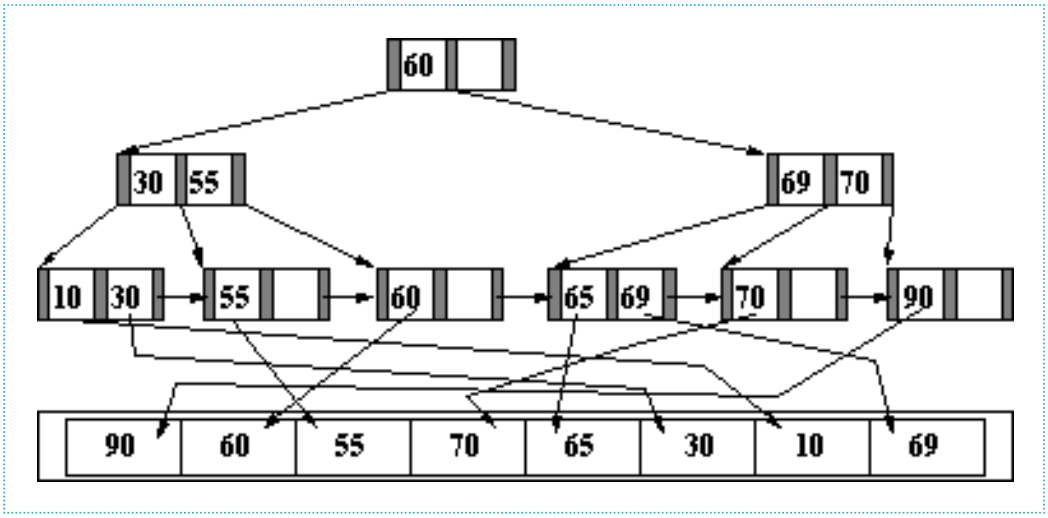


Figure : Completed B+ Tree

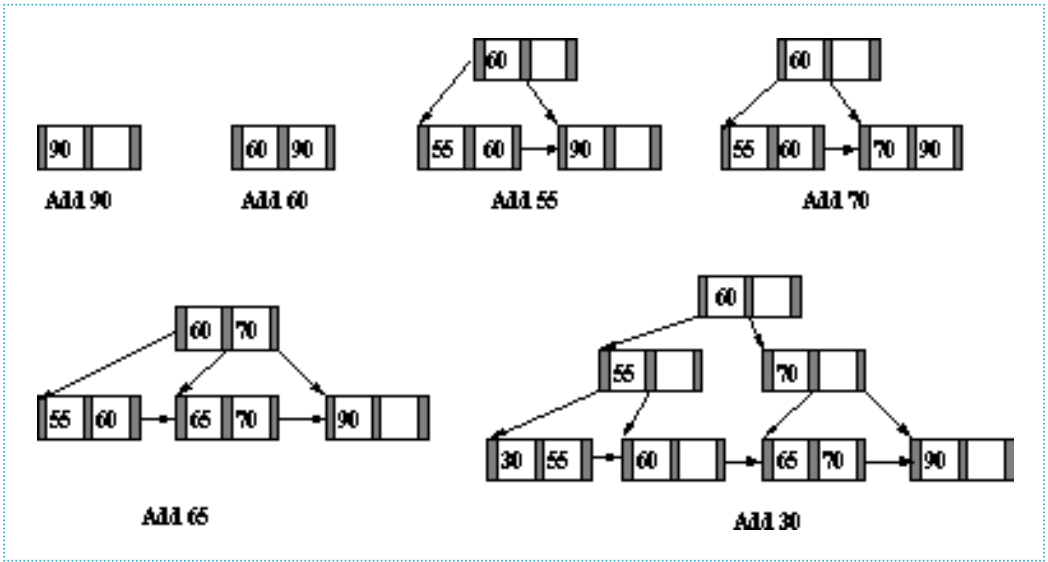


Figure : Initial Stages of B+ Tree

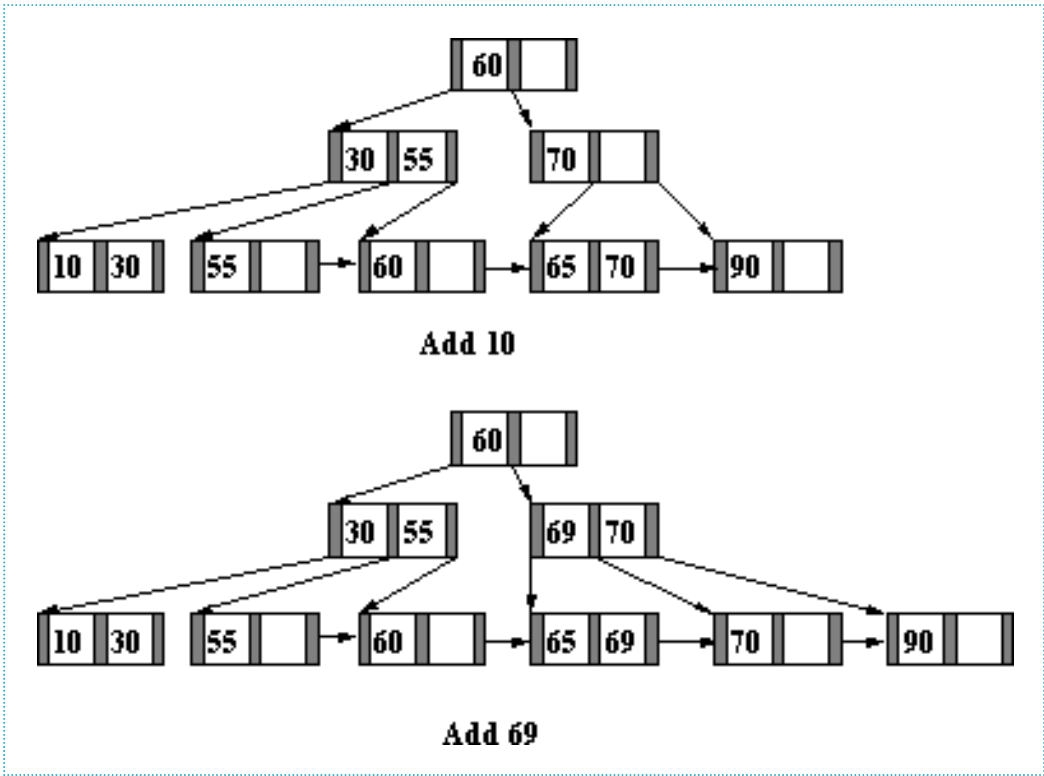


Figure : Final Stages of B+ Tree

Index Structure and Access

- The top level of an index is usually held in memory. It is read once from disk at the start of queries.
- Each index entry points to either another level of the index, a data record, or a block of data records.
- The top level of the index is searched to find the range within which the desired record lies.
- The appropriate part of the next level is read into memory from disc and searched.
- This continues until the required data is found.
- The use of indices reduce the amount of file which has to be searched.

Costing Index and File Access

- The major cost of accessing an index is associated with reading in each of the intermediate levels of the index from a disk (milliseconds).
- Searching the index once it is in memory is comparatively inexpensive (microseconds).
- The major cost of accessing data records involves waiting for the media to recover the required blocks (milliseconds).
- Some indexes mix the index blocks with the data blocks, which means that disk accesses can be saved because the final level of the index is read into memory with the associated data records.

Use of Indexes

- A DBMS may use different file organisations for its own purposes.
- A DBMS user is generally given little choice of file type.
- A B+ Tree is likely to be used wherever an index is needed.
- Indexes are generated:
    - (Probably) for fields specified with `PRIMARY KEY' or `UNIQUE' constraints in a

CREATE TABLE statement.
    - For fields specified in SQL statements such as CREATE [UNIQUE] INDEX indexname ON tablename (col [,col]...);
- Primary Indexes have unique keys.
- Secondary Indexes may have duplicates.
- An index on a column which is used in an SQL `WHERE' predicate is likely to speed up an enquiry.
- this is particularly so when `=' is involved (equijoin)
- no improvement will occur with `IS [NOT] NULL' statements
- an index is best used on a column with widely varying data.
- indexing a column of Y/N values might slow down enquiries.
- an index on telephone numbers might be very good but an index on area code might be a poor performer.
- Multicolumn index can be used, and the column which has the biggest range of values or is the most frequently accessed should be listed first.
- Avoid indexing small relations, frequently updated columns, or those with long strings.
- There may be several indexes on each table. Note that partial indexing normally supports only one index per table.
- Reading or updating a particular record should be fast.
- Inserting records should be reasonably fast. However, each index has to be updated too, so increasing the indexes makes this slower.
- Deletion may be slow.
- particularly when indexes have to be updated.
- deletion may be fast if records are simply flagged as `deleted'.

Shadow Paging

The ideas proposed for implementing transactions are prefectly workable, but such an approach would not likely be implemented in a modern system. Instead a disk block transaction technique would more likely be used. This saves much messing around with little pieces of information, while maintaining disk order and disk clustering.

Disk clustering is when all the data which a query would want has been stored close together on the disk. In this way when a query is executed the DBMS can simple "scoop" up a few tracks from the disk and have all the data it needs to complete the query. Without clustering, the disk may have to move over the whole disk surface looking for bits of the query data, and this could be hundreds of times slower than being able to get it all at once. Most DBMS systems perform clustering techniques, either user-directed or automatically.

With shadow paging, transaction logs do not hold the attributes being changed but a copy of the whole disk block holding the data being changed. This sounds expensive, but actually is highly efficient. When a transaction begins, any changes to disk follow the following procedure:

1. If the disk block to be changed has been copied to the log already, jump to 3.
2. Copy the disk block to the transaction log.
3. Write the change to the original disk block.

On a commit the copy of the disk block in the log can be erased. On an abort all the blocks in the log are copied back to their old locations. As disk access is based on disk blocks, this process is fast and simple. Most DBMS systems will use a transaction mechanism based on shadow paging.

Disk Parallelism

When you look at an Oracle database implementation, you do not see one file but several...

```
ls -sh /u02/oradata/grussell/

2.8M control01.ctl
2.8M control02.ctl
2.8M control03.ctl
 11M redo01.log
 11M redo02.log
 11M redo03.log
351M sysaux01.dbf
451M system01.dbf
3.1M temp01.dbf
 61M undotbs01.dbf
 38M users01.dbf
```

Each of these files has a separate function in Oracle, and requests can be fired to each of them in parallel. The transaction logs are called redo logs. The activesql interface is stored completely in users01. In my case all the files are in a single directory on a single disk, but each of the files could be on a different disk, meaning that the seek times for each file could be in parallel.

Caching of the files is also going on behind the scenes. For instance, the activesql tables only take up 38MB, and thus can live happly in memory. When queries come in the cache is accessed first, and if there is a need to go to disk then not only is the data requested read, but frequently data nearby that block is also read. This is called prefetching, and is based on the idea that if I need to go to disk for something, I might as well get more than I need. If it turns out that the other stuff is not needed, then not much time or resource was wasted, but if the other stuff is needed in the near future, the DBMS gains a huge performance hit. Algorithms help to steer the preloading strategy to its best possible probability of loading useful data.

Lastly, the maximum performance of a database is achieved only when there are many queries which can run in parallel. In this case data loaded for one query may satisfy a different query. The speed of running 1 query on an empty machine may not be significantly different from running 100 similar queries on the machine.

Chapter 7 - Database Connectivity

Very Basic introduction to the approaches of using SQL in a programming language.

- Application Links

Application Links

Contents

Some relational database products provide a full programming environment. Such systems include Access, ORACLE, Paradox. At one time these integrated environments were labelled as 4GLs. This term quickly became overused, and the term 4GL has largely fallen into disuse.

Typically such systems will include:

- A database engine.
- A mechanism for creating tables and entering raw data into the tables. Such a table editor will often provide a means of establishing foreign keys and simple format restrictions.
- A tool for creating applications. Often the user will be isolated from the raw data entry mechanism. A protected environment can be built by a database designer; users are then presented with a simplified view of the data. Some kind of programming language is usually built in.
- Various mechanisms for producing reports

The database engine is not normally visible to the user or even the programmer - indeed it should be possible for the programmer to switch between engines relatively painlessly.

Some concerns

4GL systems can lead to the rapid development of relatively powerful applications. However:

- The very speed of development can cause long term difficulties - things that start as prototypes tend to become products.
- The proprietary nature of the products they are based on can cause constant update problems.
- A particular product may tie systems into specific operating systems (witness the number of dumb terminals sitting alongside PC's)
- Vendors bring out new versions regularly, developers rarely have the luxury of working on the

latest version. Program maintenance on obsolete versions of a language can be irritating
- As systems come and go it can be difficult to find expertise

Databases in other languages

Rather than developing in a proprietary, specialist language (VBA, dBase, PL) we can develop in a well established, general purpose language (C, C++, Pascal) and link to a database engine.

There are several common means of achieving this

- SQL embedding
- using an API (application programmer interface) such as ODBC
- visual programming approach (Visual Basic, Delphi...)

Each of these approaches involves the notion of a cursor.

Cursors

A cursor may be viewed as a pointer into a relational table (or view). It will usually be possible for the programmer to step forwards and backwards through the table. Individual fields may be accessible as

- Text boxes
- Program variables
- API function calls

API calls

A database API (application program interface) is a set of function definitions that allow an application program to connect to an SQL server. Typical API instructions include:

- connect – identify the machine to connect to and the user name and password of the account to be used
- execute – send an SQL statement to the server. This function often returns a "handle" or "cursor" if data is to be returned from the SQL statement
- fetch – get one row of the data returned by a select statement
- advance – move the cursor on to the next row
- test – check if we are at the last row
- close – close the connection to the database and release any resources used by the connection

Having a single standard API can give many advantages to the programmer. Ideally a programmer can write and compile a program using a particular database product (such as Microsoft SQL Server) then switch to another database vendor (such as Oracle) with only a trivial change to the code. As both manufacturers provide an implementation of the API the code should work equally well in both cases.

- programmers who use non-standard SQL lose this flexibility
- malevolent database vendors trick programmers into using non-standard SQL so that the code is "locked into" a single product.

The following example is for Delphi. A similar functionality is available in VB and other languages.

```
Table1.First;
while not Table1.EOF do
begin
  Memo1.lines.Add(Table1.FieldByName('NAME').AsString);
  Table1.Next;
end;
```

This is a typical routine for reading from a table. The cursor is placed at the beginning of the file, inside the loop two actions take place:

- Data is read from the current record and processed in some way
- The cursor is moved on to the next record

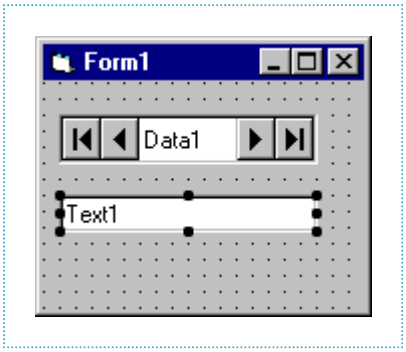
The loop terminates when the cursor attempts to move on from the final record.

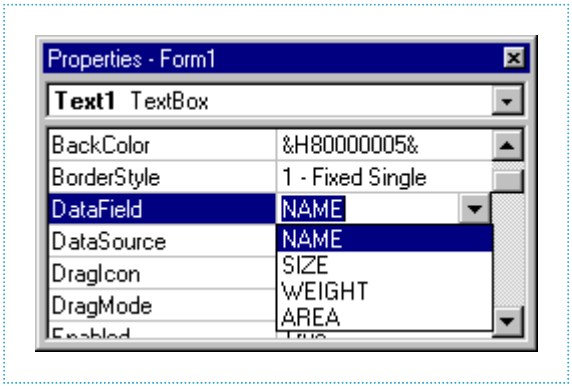
Note

- The order of the elements in the table is governed by an IndexFile property of the Table1 value
- It is possible to change values by assigning the "pseudo-variable"; Table1.FieldByName ('NAME').AsString
- Table1.Fields[x] may be used in place of FieldByName('xyz')

Data Linked Visual Components

This example shows a data-linked text box in Visual Basic. Other components are possible. Other languages have similar mechanisms.





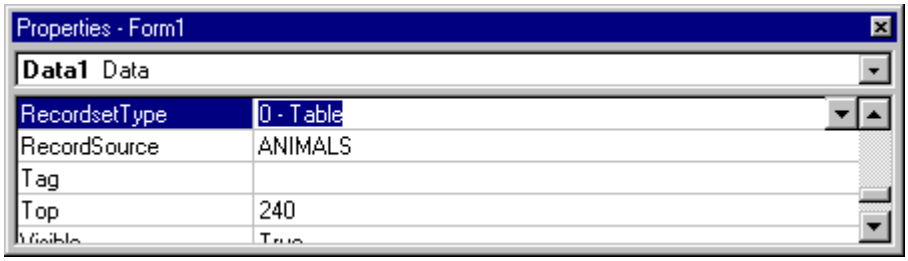


Figure : Visual Components

Notes:

- The data source Data1 has it's "record source" property set to a pre-existing table "ANIMALS"
- Data1 acts as a cursor - the arrows permit users to move backwards and forward through the table at run time
- Text1 is a data-linked or data-aware component. The property Text1.DataSource is set to Data1, the property Text1.DataField is set to "NAME"
- As the cursor moves the data displayed in Text1 is updated to reflect the current row
- Text1 may also be set up to automatically update the database if the user performs an edit.

Using spreadsheets

Many spreadsheets permit primitive relational operators:

```
=VLOOKUP(B1, Sheet2!$A$1:$B$3, 2)
```

We can have many of the advantages of a relational database within a spreadsheet by sticking to a few rules:

- Store one record per row (1NF)
- Rely on VLOOKUP index into other tables
- Maintain key order

Using PHP and MySQL

```
selene(63)% /usr/local/mysql/bin/mysql -h
zeus -u andrew -p
mysql> use andrew
mysql> show tables;
+------------------+
| Tables_in_andrew |
+------------------+
| one              |
| cia              |
+------------------+
2 rows in set (0.05 sec)

mysql> select * from cia where
population>200000000;

+---------------+----------------+---------+------------+---------------+
| name          | region         | area    | population | gdp           |
+---------------+----------------+---------+------------+---------------+
| China         | Asia           | 9596960 | 1261832482 | 4800000000000 |
```

```
| India          | Asia           | 3287590 | 1014003817 | 1805000000000 |
| Indonesia      | Southeast Asia | 1919440 |  224784210 |  610000000000 |
| United States  | North America  | 9629091 |  275562673 | 9255000000000 |
+----------------+----------------+---------+------------+---------------+
4 rows in set (0.11 sec)
```

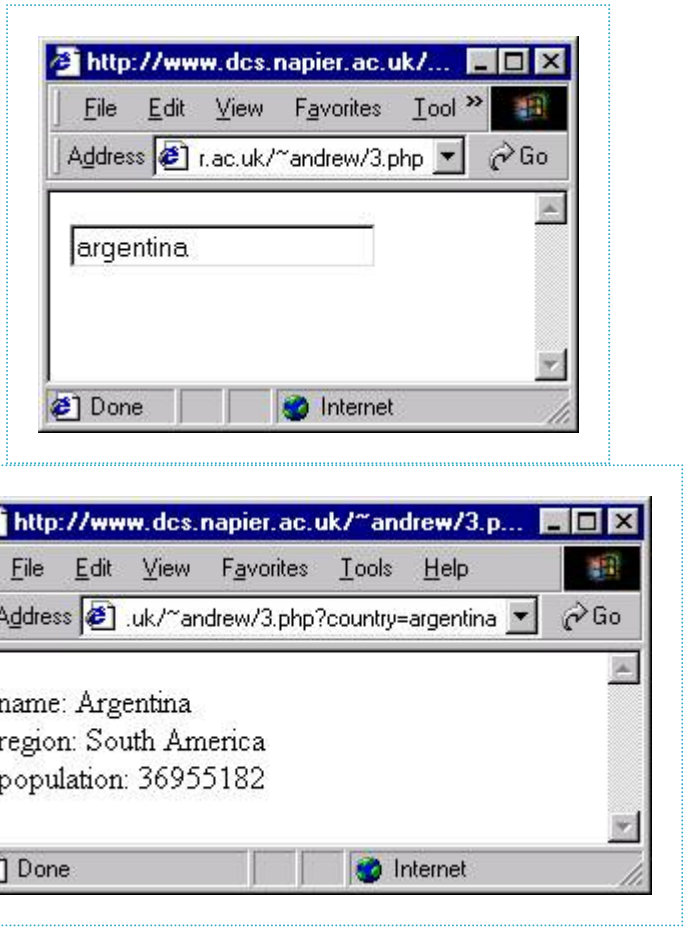


Figure : CGI Example

```
<?php
if ($country) {
  $link = mysql_connect("zeus","andrew","******") or die("Could not connect");
  mysql_select_db("andrew") or die("Could not select database");
  $query = "SELECT name, region, population FROM cia WHERE name='$country'";
  $result = mysql_query($query) or die("Query failed");
  while ($row = mysql_fetch_array($result)) {
    extract($row);
    print "name: $name<br>\n";
    print "region: $region<br>\n";
    print "population: $population<br>\n";
  }
  print "</table>\n";
  mysql_free_result($result);
  mysql_close($link);
}else{
  print "<form><input name='country'></form>\n";
}
?>
```

SQL Embedding

The following code is a simple embedded SQL program, written in C. The program illustrates many, but not all, of the embedded SQL techniques. The program prompts the user for an order number,

retrieves the customer number, salesperson, and status of the order, and displays the retrieved information on the screen.

```
main()
{
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      int OrderID;         /* Employee ID (from user)         */
      int CustID;            /* Retrieved customer ID         */
      char SalesPerson[10]   /* Retrieved salesperson name      */
      char Status[6]         /* Retrieved order status        */
   EXEC SQL END DECLARE SECTION;

   /* Set up error processing */
   EXEC SQL WHENEVER SQLERROR GOTO query_error;
   EXEC SQL WHENEVER NOT FOUND GOTO bad_number;

   /* Prompt the user for order number */
   printf ("Enter order number: ");
   scanf ("%d", &OrderID);

   /* Execute the SQL query */
   EXEC SQL SELECT CustID, SalesPerson, Status
      FROM Orders
      WHERE OrderID = :OrderID
      INTO :CustID, :SalesPerson, :Status;

   /* Display the results */
   printf ("Customer number:  %d\n", CustID);
   printf ("Salesperson: %s\n", SalesPerson);
   printf ("Status: %s\n", Status);
   exit();

query_error:
   printf ("SQL error: %ld\n", SQLCA.SQLCODE);
   exit();

bad_number:
   printf ("Invalid order number.\n");
   exit();
}
```

Advantages of a standard API

An ideal API is one that connects many development platforms to many database implementations. It allows application designers to give their users access to many databases; it allows database manufacturers to provide an interface to many application platforms.

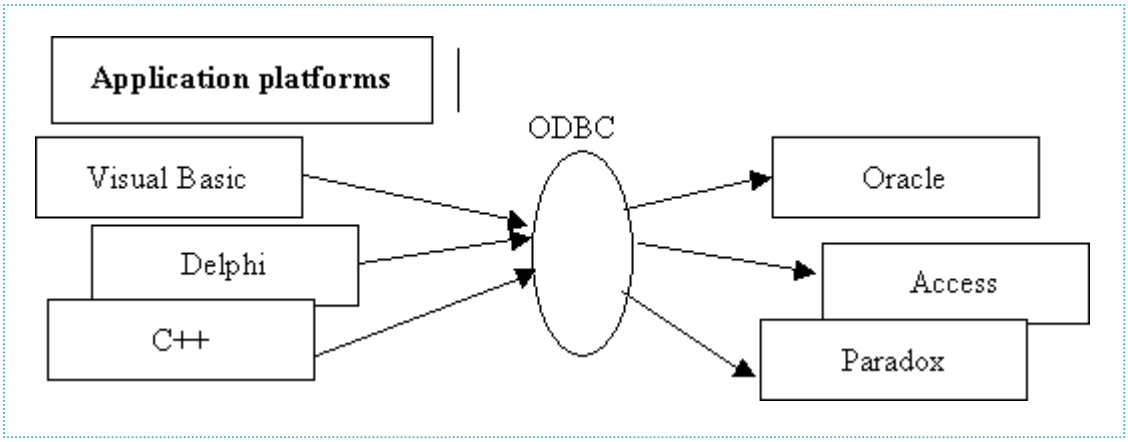


Figure : Standard API via ODBC

Without a standard each language would have to provide an interface to each database implementation.

Popular APIs

ODBC - Open Database Connectivity

Specified by Microsoft and principally associated with the MS Windows platform, ODBC includes a basic set of routines to connect to a database engine. ODBC connections can be set up from the control panel on a windows machine.

This is a popular and successful API. Most programming languages can form an ODBC connection without reference to the underlying database product. Most database products can fulfill the API.

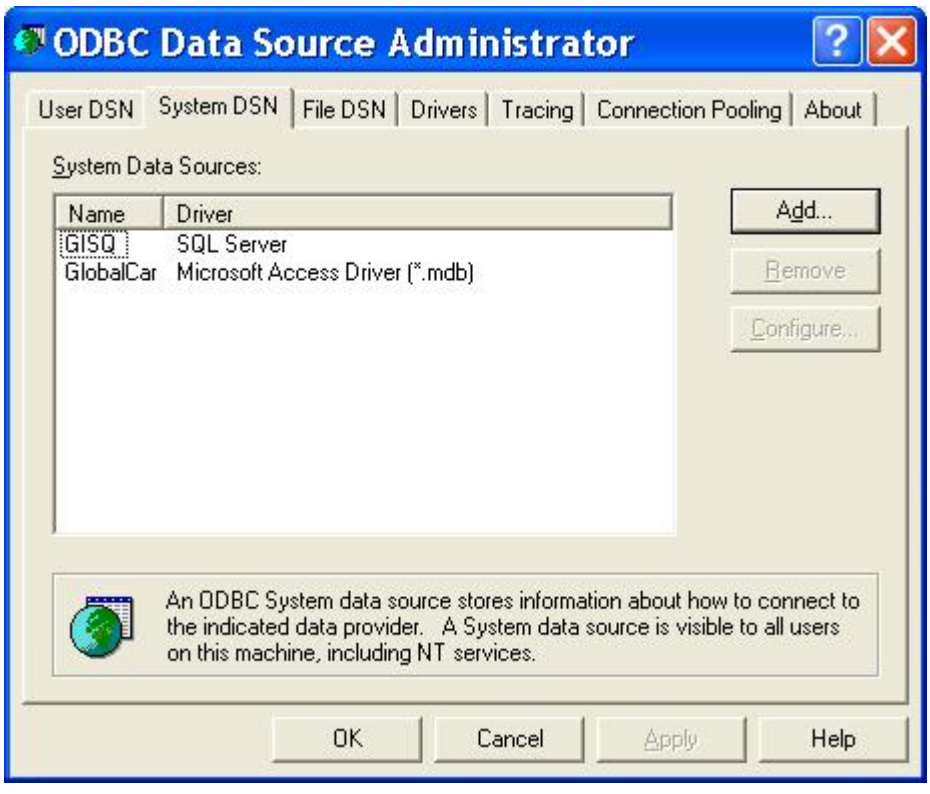


Figure : ODBC Data Source Administrator

Using SQLBindCol (ODBC)

The application binds columns by calling **SQLBindCol**. This function binds one column at a time. With it, the application specifies:

- The column number. Column 0 is the bookmark column; this column is not included in some result sets. All other columns are numbered starting with the number 1. It is an error to bind a higher numbered column than there are columns in the result set; this error cannot be detected until the result set has been created, so it is returned by **SQLFetch**, not **SQLBindCol**.
- The C data type, address, and byte length of the variable bound to the column. It is an error to specify a C data type to which the SQL data type of the column cannot be converted; this error might not be detected until the result set has been created, so it is returned by **SQLFetch**, not **SQLBindCol**.
- The address of a length/indicator buffer. The length/indicator buffer is optional. It is used to return the byte length of binary or character data or return SQL_NULL_DATA if the data is NULL.

When **SQLBindCol** is called, the driver associates this information with the statement. When each row of data is fetched, it uses the information to place the data for each column in the bound application variables.

For example, the following code binds variables to the SalesPerson and CustID columns. Data for the columns will be returned in *SalesPerson* and *CustID*. Because *SalesPerson* is a character buffer, the application specifies its byte length (11) so the driver can determine whether to truncate the data. The byte length of the returned title, or whether it is NULL, will be returned in *SalesPersonLenOrInd*.

Because *CustID* is an integer variable and has fixed length, there is no need to specify its byte length; the driver assumes it is **sizeof(SQLUINTEGER)**. The byte length of the returned customer ID data, or whether it is NULL, will be returned in *CustIDInd*. Note that the application is only interested in whether the salary is NULL, because the byte length is always **sizeof (SQLUINTEGER)**.

```
SQLCHAR         SalesPerson[11];
SQLUINTEGER     CustID;
SQLINTEGER      SalesPersonLenOrInd, CustIDInd;

SQLRETURN       rc;

SQLHSTMT        hstmt;

// Bind SalesPerson to the SalesPerson column and CustID to the CustID column.
SQLBindCol(hstmt, 1, SQL_C_CHAR, SalesPerson, sizeof(SalesPerson),
            &SalesPersonLenOrInd);
SQLBindCol(hstmt, 2, SQL_C_ULONG, &CustID, 0, &CustIDInd);

// Execute a statement to get the sales person/customer of all orders.
SQLExecDirect(hstmt, "SELECT SalesPerson, CustID FROM Orders ORDER BY SalesPerson
            SQL_NTS);

// Fetch and print the data. Print "NULL" if the data is NULL. Code to check if r
// equals SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
   if (SalesPersonLenOrInd == SQL_NULL_DATA)
            printf("NULL");
```

```
   else
            printf("%10s   ", SalesPerson);

   if (CustIDInd == SQL_NULL_DATA)
         printf("NULL\n");
   else
         printf("%d\n", CustID);
}

// Close the cursor.
SQLCloseCursor(hstmt);
```

JDBC

JDBC provides a similar level of functionality to ODBC but is specific to the Java programming language.

The following is an example of a Java program using JDBC:

```
/* CIA.java
  From http://sqlzoo.net By Andrew Cumming
*/
import java.sql.*;
public class CIA{
 public static void main(String[] args){
   Connection myCon;
   Statement myStmt;
   try{
     Class.forName("com.mysql.jdbc.Driver").newInstance();
     # Connect to an instance of mysql with the follow details:
     # machine address: pc236nt.napier.ac.uk
     # database      : gisq
     # user name     : scott
     # password      : tiger
     myCon = DriverManager.getConnection(
             "jdbc:mysql://pc236nt.napier.ac.uk/gisq"
             "scott","tiger");
     myStmt = myCon.createStatement();
     ResultSet result = myStmt.executeQuery(
        "SELECT name FROM cia WHERE population>200000000");
     while (result.next()){
       System.out.println(result.getString("name"));
     }
     myCon.close();
   }
   catch (Exception sqlEx){
     System.err.println(sqlEx);
   }
 }
}
```

Such a program may be compiled with the command:

```
javac CIA.java
```

It may be executed with the command:

```
 java -cp mysql-connector-java-2.0.14-bin.jar:. CIA
```

In the code shown a connection is made to the mysql database using the getConnection method – we

specify the machine on which mysql is running (pc236nt.napier.ac.uk), the mysql database (gisq), the user name (scott) and the password (tiger).

Having created an SQL statement we get a ResultSet object by executing the SQL statement over the connection. For the statement given we get four rows each with a single column – these are the countries China, India, United States and Indonesia.

The ResultSet object is a cursor that points to a single row of the result table. Initially the cursor is considered to be pointing to before the first row, the method result.next() is the first instruction – this moves it on to the first row.

We can retrieve data from the ResultSet using the getString method. The getString method takes either an attribute name (as in this case) or an integer indicating the position of the attribute. In either case the value of that field is returned as a string. Similar methods such as getInteger are available.

DBI/DBD

An API which is growing in popularity is the DBI/DBD interface. This is an attempt to offer a standard programmers interface to executing SQL from a variety of languages. It has many similarities to ODBC, but without some of the complexities. It is a popular database linking API for Perl.

The following is a fragment of Perl example code for finding out the surname of employees with a particular department number.

```
my $dbh = DBI->connect("dbname","username","password");
my $depno = 3;
my $cmd = $dbh->prepare("SELECT surname FROM employee where depno=?");
my $res = $cmd->execute($depno);
while (my ($name) = $res->fetchrow_array()) {
  prnt "The employee name is $name\n";
}
```

Using ASP

ASP programming allows data from a database to be displayed on web pages. The ASP script (often VBScript or JScript) is interpreted at the Web server. The web client (the browser) receives plain HTML.

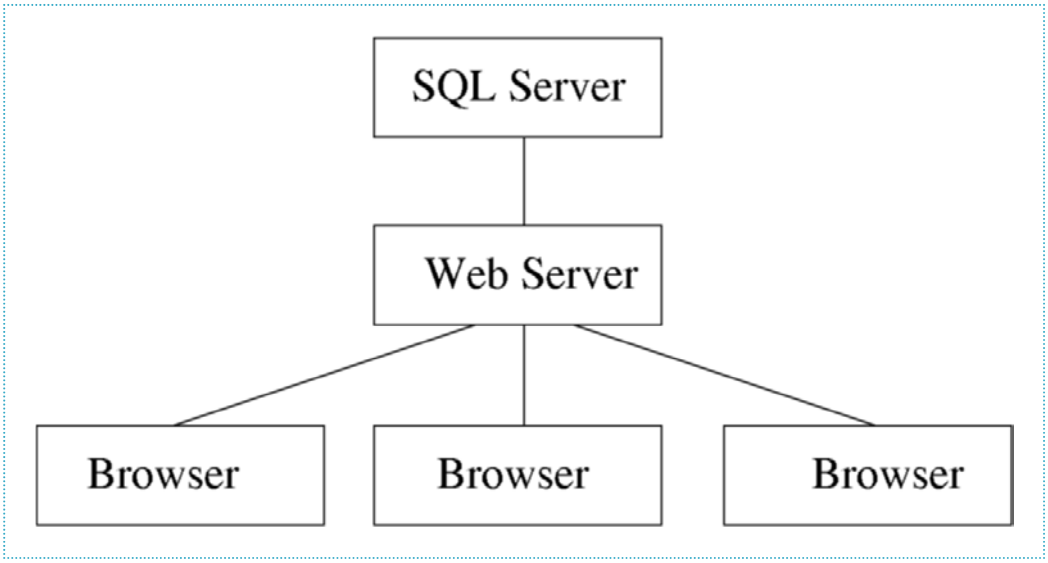


Figure : Using ASP

Typically the following sequence of events takes place:

- A web surfer requests an ASP page by linking to www.xyz.com/page1.asp, this user is likely to be using Netscape or IE or similar - the browser does **not** need any special plugins or applets
- The Web Server (probably IIS or PWS) receives the request, examines page1.asp in the local file space and interprets it - the page makes reference to database db1.mdb - this causes a request to the RDBMS
- The RDBMS (probably SQL Server) gets the SQL request and returns the results.

A sample ASP code

```
<%SQL="SELECT carName FROM Cars ORDER BY carName"
  set conn = server.createobject("ADODB.Connection")conn.open "parking"
  set cars=conn.execute(SQL) %>
<% do while not cars.eof %>
  <%= cars(0) %> <br>
  <%cars.movenext
    loop%>
<% cars.close %>
```

If you are interested in trying out ASP yourself, you will need to get an account on an IIS server. You can try running your own server using PWS from Microsoft. Another option is to get a free account from an online site such as www.brinkster.com

Efficiency Issues

No matter how the connection to the database is made, care should be taken to ensure that the connection is handled efficiently. The application connecting to a database is typically executing on a different machine to the database server. This has many advantages – and is essential if the database is to be shared by more than one user machine. However having the application program and the database server on different machines introduces a significant performance cost.

Establishing a connection, including time to log on and verify passwords can be costly. Communication between the application machine and the server must go across a network – this is usually considerably slower than the disk transfer rate. Sending SQL statements to the server is relatively trivial however sending the rows of the results back may be significant. Programmers should take care to request only the data they need, lazy programmers may be tempted to request “SELECT * FROM table” when “SELECT id FROM table” would do. In the first case all fields may be sent across the network for every row examined – this can be very expensive especially if there are many fields or some of them are lengthy.

As an example consider a web site configured to dynamically create and serve web pages from stored database information. When the server program is generating web pages it may be that database connections are being created rapidly to solve relatively trivial requests. It is not uncommon for this type of web server to be spending far more time opening and closing connections than anything else. In such cases connection pooling may help. The web-server creates a pool of connections and keeps these open between requests. This may be transparent to the application programmer.

Chapter 8 - MetaData

The role of the database administrator, security in a database, and the data dictionary.

Metadata, Security, and the DBA

Contents

Metadata

So far in the DBMS we have looked at table schema for our database design. We have also considered views, and in many ways these act like tables. This table theme extends to all parts of a DBMS. In particular, Oracle makes this theme quite explicit.

In Oracle, everything is a table. Not only the things we think of as tables, but also the system things like user information. The philosophy is simple... implement the concept of a table and we have everything we need to build a DBMS. This includes security concepts; secure the table concept and everything is similarly secure.

Oracle has a special tablespace, called SYS, which holds all the system information. Various security levels protect SYS, so dependent on your access rights you may or may not be able to see all the tables held there. SYS in total holds hundreds of tables. The list below gives a few of these table names.

```
USER_OBJECTS
TAB
USER_TABLES
USER_VIEWS
ALL_TABLES
USER_TAB_COLUMNS
USER_CONSTRAINTS
USER_TRIGGERS
USER_CATALOG
DBA_USERS
```

For example, the DBA_USERS table holds user information.

```
SQL> describe dba_users;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 USERNAME                                  NOT NULL VARCHAR2(30)
 USER_ID                                   NOT NULL NUMBER
 PASSWORD                                           VARCHAR2(30)
 ACCOUNT_STATUS                            NOT NULL VARCHAR2(32)
 LOCK_DATE                                          DATE
 EXPIRY_DATE                                        DATE
 DEFAULT_TABLESPACE                        NOT NULL VARCHAR2(30)
 TEMPORARY_TABLESPACE                      NOT NULL VARCHAR2(30)
 CREATED                                   NOT NULL DATE
 PROFILE                                   NOT NULL VARCHAR2(30)
 INITIAL_RSRC_CONSUMER_GROUP                        VARCHAR2(30)
 EXTERNAL_NAME                                      VARCHAR2(4000)
```

The DBA_USERS table holds the username of users, an ID number unique for each user, their login password, the tablespace where their personal tables and views are created, a space for calculating the results of queries (temporary tablespace), plus many more internal details.

An example of a table holding the more internal features of the DBMS is the USER_CONSTRAINTS table. This (extensively) documents the constraints which exist between tables in the database. A summary of the attributes is shown below.

```
SQL> describe user_constraints;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 OWNER                                     NOT NULL VARCHAR2(30)
 CONSTRAINT_NAME                           NOT NULL VARCHAR2(30)
 CONSTRAINT_TYPE                                    VARCHAR2(1)
 TABLE_NAME                                NOT NULL VARCHAR2(30)
 ......
 DEFERRABLE                                         VARCHAR2(14)
 DEFERRED                                           VARCHAR2(9)
 ......
 LAST_CHANGE                                        DATE
 ......
```

Here a row links the owner of the constraint to a constraint name and type. This constraint is on a table name. The date when this change was made is recorded. Oracle allows constraint checking to be put off till the end of a transaction, and this is known as DEFERRING. If a constraint can be deferred then it will be DEFERRABLE, and if it is currently deferred that too can be recorded.

```
select owner,table_name,constraint_name,constraint_type
from all_constraints
where owner = 'DBRW'
and table_name in ('EMPLOYEE','JOBHISTORY','DEPARTMENT')
;
```

OWNER	TABLE_NAME	CONSTRAINT_NAME	CON
DBRW	DEPARTMENT	SYS_C0010801	P
DBRW	EMPLOYEE	SYS_C0010803	P
DBRW	EMPLOYEE	SYS_C0010804	R
DBRW	JOBHISTORY	SYS_C0010807	P
DBRW	JOBHISTORY	SYS_C0010808	R

From the tutorials you may remember these tables. The constraints indicate that the DEPARTMENT has only a PRIMARY KEY constraint. EMPLOYEE and JOBHISTORY also have primary key constraints, but also have some foreign key referential integrity constraints (R). The constraint names are automatically generated when the tables are created. These names can be useful, as attempting to delete table DEPARTMENT results in a error indicating that this would violate constraint SYS_C0010804, and using this table shows that DEPARTMENT must have a foreign key relationship from EMPLOYEE, and therefore EMPLOYEE must be dropped first.

There is a great deal of metadata in a DBMS, extending well beyond the implementation of the user schema. This includes support for application links and schema documentation (such as comments). Exploring this metadata can give a valuable insight into DMBS construction and performance issues.

Security

Security of the database involves the protection of the database against:

- unauthorised disclosures
- alteration
- destruction

The protection which security gives is usually directed against two classes of user

- Stop people without database access from having any form of access.
- Stop people with database access from performing actions on the database which are not required to perform their duties.

There are many aspects to security

- Legal, social and ethical aspects

Legally there is the Data Protection Act, which places restrictions on databases which contain information on living people. This was created to protect the public from data contained on a computer, about themselves, to which the public had previously no legal right of access. Information on computers can be wrong, and decisions made on wrong information concerns the public and additionally is of no benefit to the company holding the data. The act supports the idea of the public querying data, and indicating errors in that data.

However, just because a database is legal does not make it socially or ethically acceptable. Collating medical records on computer for a hospital is acceptable, but not having enough security to prevent insurance companies accessing the database and using that as a basis for rejecting life assurance applications could be considered questionable. Frequently it is best to place the tightest restrictions on who can access data, and where necessary security is deliberately relaxed to allow only legitimate queries to take place.

- Physical controls

Security often begins with physical controls. If a person cannot enter the building where the database runs and is accessed, then that person cannot access the database. Usually the construction of security is a layered approach, where a person bent on accessing the database must penetrate multiple levels of security. The simple precaution of having all the database access points behind locked doors can only add to the security of the system.

- Policy questions

Security of a database is often the enforcement in the database of the company policy. All companies should have a policy statement, listing what is acceptable and what is not. Companies with weak policy statements will often have the weakest security. At a minimum, it should be policy that data stored in the database should not be made available to outside agents without written consent from a Managing Director. Without a policy statement, it is hard to argue that an employee has actually done anything wrong...

- Operational problems

If only a single person has access to a database, security is certainly higher than if many people have access. However, if all the people in the UK had to phone the same one person to find out what their bank balance was the whole system would quickly become unworkable. Security considerations often have to be balanced against operational issues.

- Hardware controls

No matter how secure the database actually is, if a person can simply steal the hard drive on which the database is stored, then that person can access the database at leisure. This case is obvious, but less obvious security failures, such as taking a copy of a backup tape of the database, can be harder to safeguard against.

- Operating system security

Most DBMS's run on top of an operating system (OS). Examples of OS's include Window 95, Windows NT, and Unix. The database may be secure from within the DBMS, but if the database can also be accessed from the OS using simple file handling programs, then a clear weakness in the security model exists.

- Database system security

Within the DBMS itself, if anyone can access anything then having any other sort of security seems pointless. The use of user accounts and password protection of user identities is a good starting point to improve security. User identities is also an aid to accountability. Protection of certain elements of the database with respect to certain users (or user groups) should always be considered where potentially confidential data is being stored. It is DBMS security which is the focus of this discussion.

Granularity of DBMS Security

The unit of data used in specifying security in the database can be, for example;

- the entire database
- a set of relations
- individual relation
- a set of tuples in a relation
- individual tuple
- a set of attributes of all tuples
- an attribute of an individual tuple.

DBMS-level Protection

- Data encryption:

Often it is hard to prevent people from copying the database and then hacking into the copy at another location. It is easier to simply make copying the data a useless activity by encrypting the data. This means that the data itself is unreadable unless you know a secret code. The encrypted data in combination with the secret key is needed to use the DBMS.

- Audit Trails:

If someone does penetrate the DBMS, it is useful to find out how they did it and what was accessed or altered. Audit Trails can be set up selectively to minimise disk usage, identify system weaknesses, and finger naughty users.

User-level Security for SQL

- Each user has certain access rights on certain objects.
- Different users may have different access rights on the same object.

In order to control the granularity of access rights, users can

- Have rights of access (authorisations) on a table
- Have rights of access on a view. Using views, access rights can be controlled horizontal and vertical subsets on a table, and on dynamically generated data from other tables.

The GRANT command

GRANT is used to grant privileges to users

```
GRANT privileges ON tablename
  TO { grantee ... }
  [ WITH GRANT OPTION ]
```

Possible privileges are:

- SELECT - user can retrieve data
- UPDATE - user can modify existing data
- DELETE - user can remove data
- INSERT - user can insert new data

- REFERENCES - user can make references to the table

The WITH GRANT OPTION permits the specified user can grant privileges which that user possesses on that table to other users. This is a good way to permit other users to look after permissions for certain tables, such as allowing a manager to control access to a table for his or her subordinates.

grantee need not be a username or a set of usernames. It is permitted to specify PUBLIC, which means that the privileges are granted to everyone.

```
GRANT SELECT ON userlist TO PUBLIC;
```

GRANT and VIEWs

When a view is created is when the security of the view is checked. Thus if there was sufficient security for the view to execute when it was created, then the view will always work no matter what additional GRANTs are made. This can be used to restrict columns and rows from a user.

```
GRANT select ON employee to jim;
create view empjim as
select empno, surname,forenames from employee;
GRANT select on empjim to jim;
REVOKE select ON employee from jim;
```

You can also restrict rows of a table to particular users by their username or other feature. In Oracle, the username of the current user is returned by the function USER. Thus the following creates a single table, but gives each user of the view the ability to look at only rows where the username matches their username.

```
CREATE table checker (
  username      varchar(200)
 ,secretinfo    varchar(100)
);
CREATE view userview as
  select * from checker
  where username = USER
;

select * from userview -- shows rows where the username matches.
;
```

The Database Administrator

As system controls increase usability of the system decreases. Actually it is perfectly possible to have an efficient and reliable system which no one can use effectively. This is never the explicit goal of the DBA, but there is a danger that it is an implicit goal.

The person who looks after the database needs to balance all needs of the users, whether they know they need it or not. No user wants security, for instance, yet if someone hacks in and deletes all their work the DBA becomes the target. Perfectly designed security is completely invisible to the valid user, but is automatic and total for the invalid user.

Security is not the only issue of importance for the DBA. They are also concerned with:

- System performance and tuning
- Data backup and recovery

- Product and tool selection, installation, and maintainance
- System documentation
- Support
- Education
- Fortune Telling / Future Prediction

A good DBA is almost never seen. The fact is that if you have to phone the DBA then the DBA has failed. The system will be monitored continuously, and problems detected and fixed before they are noticed by users. Long term issues, such as data growth, diversification, the addition of new projects, do need to be discussed with the DBA, but the DBA should be able to detect most issues anyway and handle them transparently from the users and developers.

Chapter 9 - Offline Tutorials

Mostly this site is involved with online tutorials, but a number of paper-based tutorials are also included in the online book.

- SQL Tutorial 1 - Intro material covering activeSQL tutorials 1 and 2.
- SQL Tutorial 2 - Material covering activeSQL tutorial 3.
- SQL Tutorial 3 - Material covering activeSQL tutorial 4.
- SQL Tutorial 4 - Material covering activeSQL tutorial 5.
- jobs database ER diagram
- dressmaker database ER diagram
- musician database ER diagram
- Tutorial - ER Diagram Examples 1-2
- Tutorial - ER Diagram Examples 3-5
- Tutorial - Normalisation

Tutorial 1

Contents

SQL with Oracle

SQL (Structured Query Language) is the structured query language used to manipulate relational databases. The concept of relational database was first proposed by Codd in 1970, and a language to extract and manipulate the data in it was developed theoretically during the following years. All SQL statements are instructions to the database. SQL is a non-procedural language, which means that commands are not executed step by step according to how they were written, but they are retained in memory, read through and executed in the most effective way. In relational database terminology, SQL provides 'automatic navigation' to the data in the database.

In these notes, you see an indication of two possible interfaces to SQL, namely sqlplus and activeSQL. Sqlplus is the standard Oracle interface while activeSQL is an experimental interface which Napier students have access to. For our tutorials we will all be using activeSQL.

Command Endings

Note that all SQL commands typed into sqlplus MUST end in a ; (semicolon) character. It will not work without it. The activeSQL interface is more forgiving, but even then if you enter more than 1 command into the interface at a time you MUST separate the commands with a semicolon.

When entering SQL, you can have as many space characters and return characters as you like. They are completely ignored by Oracle. Sqlplus will, when you hit return, tell you what line you are currently on. These numbers are not part of the command so do not let them confuse you. In sqlplus, if you hit return twice (return on a blank line) the current command is cancelled.

In activeSQL, no command is executed until you hit the submit button.

SELECT

SQL command SELECT is used to retrieve information from a table. SELECT informs Oracle which table(s) to use and which column(s) and row(s) to retrieve.

The asterisk can be used to denote all fields.

To list all fields and all records from the table **employee**.

```
SELECT *
FROM employee
;
```

To display only the fields empno and depno but all records

```
SELECT empno,depno
FROM employee;
```

SPECIAL TABLE

There is a special table called CAT, which contains the name and type of all tables in your namespace. Running

```
SELECT * from CAT:
```

produces the name and type of all local tables. This includes the 5 tables used in these tutorials: employee, empcourse, jobhistory, course, and department.

To find out about a particular table, you can look at the commands which created it, or you can use the DESCRIBE command. This tells you the attributes of the table in question. In sqlplus, this description does not include the Foreign Keys (the links to other tables - more of this in tutorial 2) but in activeSQL Foreign Keys ARE shown. For instance:

```
DESCRIBE employee;

          empno               integer        primary key
          surname             varchar(15)
          forenames           varchar(30)
          dob                 date
          address             varchar(50)
          telno               varchar(20)
          depno               integer        references department(depno)
```

varchar(20) indicates a string which can be up to 20 characters long.

Date indicates that field is an Oracle date.

Integer indicates that that attribute is a number.

Depno integer References department(depno) tells us that depno in employee is a link to the department table's depno attribute. In this confusing case there are two attributes called depno, one in employee and one in department. These attributes are different attributes in different tables.

Predicates

Search conditions are made up of predicates. These are then combined together with ANDs, ORs, and NOTs.

There are seven types of predicate:

- comparison
- BETWEEN predicate

- IN predicate
- LIKE predicate
- ANY or ALL predicate
- EXISTS predicate
- IS NULL

Comparisons

The comparisons available are

= equal to
!= not equal to
> greater than
>= greater than or equal to
< less than
<= less than or equal to

List the fields empno, surname, telno of all employees who have a surname Wright. Notice the quote marks required for character constants. Note also that anything within the quotes is case sensitive.

```
SELECT empno, surname, telno
FROM employee
WHERE surname = 'Wright'
```

List all current salaries in the range £20000 to £30000, listing their empno values.

```
SELECT empno, enddate, salary
FROM jobhistory
WHERE enddate IS NULL
AND salary >= 20000
AND salary <= 30000
```

List all the employees working in the company on January 1st 1980 and their position

```
SELECT empno, position, startdate, enddate
FROM jobhistory
WHERE (startdate < '01-JAN-1980' AND enddate > '01-JAN-1980')
OR (startdate < '01-JAN-1980' AND enddate IS NULL)
```

BETWEEN (Inclusive)

List all the courses which occurred during 1988

```
SELECT *
FROM course
WHERE cdate BETWEEN '01-JAN-1988' AND '31-DEC-1988'
```

List all the courses which did not occur in 1988

```
SELECT *
FROM course
WHERE cdate NOT BETWEEN '01-JAN-1988' AND '31-DEC -1988'
```

LIKE

The LIKE predicate provides the only pattern matching capability in SQL for the character data types. It takes the following form

```
columnname [NOT] LIKE pattern-to-match
```

The pattern match characters are the percent sign (%) to denote 0 or more arbitrary characters, and the underscore (_) to denote exactly one arbitrary character.

List the employee numbers and surnames of all employees who have a surname beginning with C.

```
SELECT empno,surname
FROM employee
WHERE surname LIKE 'C%'
```

List all course numbers and names for any course to do with accounting.

```
SELECT courseno,cname
FROM course
WHERE cname LIKE '%ccount%'
```

List all employees who have r as the second letter of their forename.

```
SELECT surname, forenames
FROM employee
WHERE forenames LIKE '_r%'
```

IS NULL

List all employees numbers and their current position

```
SELECT empno,position
FROM jobhistory
WHERE enddate IS NULL
```

The remaining predicates will be dealt with at a later stage.

Set Functions

A set function is a function that operates on an entire column of values, not just a single value.

List the total wage bill for the company at the moment.

```
SELECT SUM(salary)
FROM jobhistory
WHERE enddate IS NULL
```

This will retrieve the total salary for employees, where the enddate is empty or NULL.

The following are the set functions supported

Table 1: Set Functions

Name	Description	
COUNT	Count of occurrences	
SUM	Summation	
AVG	Average (Sum/Count)	zero if Count = zero
MAX	Maximum value	
MIN	Minimum value	

Find the number of employees working currently.

```
SELECT COUNT(*)
FROM jobhistory
WHERE enddate IS NULL
```

The COUNT(*) function is used to count rows in a table, and is the exception to the following rule.

NULL values are ignored by the set functions.

Count how many jobs that employee number 25 has had previously.

```
SELECT COUNT(enddate)
FROM jobhistory
WHERE empno = 25
```

Count how many jobs employee number 25 has had, including current job.

```
SELECT COUNT(startdate)
FROM jobhistory
WHERE empno = 25
```

Calculate the average salary for all employees.

```
SELECT AVG(salary)
FROM jobhistory
WHERE enddate IS NULL
```

Note that a 'column label' might be usefully added. This will be output in uppercase format unless enclosed in double quotes as follows :--

Find out the greatest salary.

```
SELECT MAX(salary)    "Highest Salary"
FROM jobhistory
WHERE enddate IS NULL
```

Ordering rows of a query result

The order in which the selected rows are displayed is changed by adding an ORDER BY clause to the end of your SELECT command. The ordering is done numerically or alphabetically and can be ascending or descending.

List all the employee numbers and salaries, ordered by their salary.

```
SELECT empno, salary
```

```
FROM jobhistory
WHERE enddate IS NULL
ORDER BY salary
```

To order by descending order you need to add DESC in the ORDER BY command

```
SELECT empno, salary
FROM jobhistory
WHERE enddate IS NULL
ORDER BY salary DESC
```

Prevention of duplicate rows

If you print all the jobs in the jobhistory table you will get duplicate rows.

```
SELECT position
FROM jobhistory
```

To print out only one for each different job you need to add DISTINCT in the SELECT clause.

```
SELECT DISTINCT position
FROM jobhistory
```

Counting unique rows

Often you would like to count how many different rows exist in the result of a query. Doing:

```
SELECT DISTINCT count(position)
FROM jobhistory
```

result in an answer which is no different from the same query without the DISTINCT. This is caused by the fact that count() is done before DISTINCT, and therefore in this case DISTINCT does nothing. What is actually needed is a way of forcing DISTINCT to be done before the count. This can be achieved by doing:

```
SELECT count(DISTINCT position)
FROM jobhistory
```

Tutorial S2

Contents

Joining tables

Often the information required is contained in more than one table. You can specify more than one table in the FROM clause. For example the use of the two tables **employee** and **jobhistory** in the FROM clause will create a larger table with each row in **employee** combined with each row in **jobhistory**. Each of these new rows will have all the columns from the **employee** table and the **jobhistory** table. If there are 3 rows in employee and 5 rows in jobhistory this will create a new table of 3 times 5 (i.e. 15) rows. This is known as a **Cartesian product**.

A Cartesian product will contain many rows of no practical interest, such as rows containing the employee and jobhistory details for two different employees. It is therefore necessary to have some restriction on the join. Here a likely requirement is that the empno field in the **employee** table matches the empno of the **jobhistory** table. Each row in the resulting table will then contain employee and jobhistory data for only one employee.

List the employees number, surname, and current job title.

```
SELECT employee.empno, surname, position
FROM employee, jobhistory
WHERE enddate IS NULL
AND employee.empno = jobhistory.empno
```

A more modern syntax of this would be.

```
SELECT employee.empno, surname, position
FROM employee JOIN jobhistory ON (employee.empno = jobhistory.empno)
WHERE enddate IS NULL
```

Notice that the fields which are not unique must be explicitly referred to by use of the table name and a fullstop followed by the fieldname. For instance empno occurs in both the **employee** table and the **jobhistory** table and so it must be explicitly referred to. This also means that it must be explicitly referred to in the SELECT clause even though the values are the same for employee.empno and jobhistory.empno.

You can use more than two tables in the FROM clause. There is no theoretical limit, however there will be some limit placed on you by the system itself. If you have N tables in the FROM clause then you will normally need (N – 1) join conditions.

Aliases or correlation names.

Although table prefixes prevent ambiguity in a query, they can be tedious to enter. You can define temporary labels in the FROM clause and use them elsewhere in the query. Such temporary labels are sometimes known as temporary table aliases.

List the employee number, surname and department of each employee.

```
SELECT e.empno, surname, dname
FROM employee e JOIN department d ON (e.depno = d.depno)
```

Notice that the table **employee** is given an alias e, and **department** an alias d. This can then be used during the query. It is also possible to use the actual name. Notice also that the join is on the two tables employee and department.

Equi-joins and non-equi joins

When you join the table **department** to the table **employee**, the join condition specifies the relationship between them. Such joins are known as **equi-joins** because the comparison operator is the equals operator. Any join that does not use this operator is known as a **non-equi join**.

GROUP BY

Conceptually GROUP BY rearranges the table designated in the FROM clause into partitions or groups, such that within any one group all rows have the same value for the GROUP BY field(s).

List the departments and their total current salary bill

```
SELECT depno, sum(salary) "Salary"
FROM employee JOIN jobhistory ON (employee.empno=jobhistory.empno)
WHERE enddate IS NULL
GROUP BY depno
```

In the above example, table employee is grouped so that one group contains all the rows for department 1, another contains all the rows for department 2, and so on.

The sum(salary) "Salary" renames the column Salary.

Each expression in the SELECT clause must be single-valued per group (i.e. it can be one of the GROUP BY fields or an arithmetic expression involving such a field), or a constant, or a function such as SUM that operates on all values of a given field within a group and reduces those values to a single value.

The purpose of such grouping is generally to allow some set function to be computed for each group.

HAVING

The GROUP BY clause may be qualified by a HAVING clause. The HAVING clause restricts the groups which are selected in the output. The groups that do not meet the search condition are eliminated.

Each expression in the HAVING clause must also be single-valued per group.

List the number of people who have been on each course numbered 1 to 6

```
SELECT courseno, COUNT (*)
FROM empcourse
GROUP BY courseno
HAVING courseno BETWEEN 1 AND 6
```

Execution of queries

From a conceptual standpoint, the subselect is evaluated in the following manner: First, the Cartesian product of all tables identified in the FROM clause is formed. From that product, rows not satisfying the search condition specified in the WHERE clause are eliminated. Next, the remaining rows are grouped in accordance with the specifications of the GROUP BY clause. Groups not satisfying the search condition in the HAVING clause are then eliminated. Then, the expressions specified in the SELECT clause are evaluated. Finally, the ORDER BY clause, if present, is evaluated and, if the key word DISTINCT has been specified, any duplicate rows are eliminated from the result table.

Joining Tables to Themselves - Self joins

Sometimes a table must be joined to itself. In this case, any references to fieldnames become ambiguous and aliases must be used to uniquely identify required fields.

List the surname and forename of all the employees who work in the same department as employee number 16. In this case two “versions” of the employee table must be used, one for employees other than 16, and one for employee 16 :-

```
SELECT x.surname, x.forenames
FROM employee x, employee y
WHERE x.depno = y.depno
AND y.empno = 16
AND x.empno != 16
```

You need to have one version of the table **employee** so that you can find the department number of employee 16. In the above example this table is called **y**. You then look through another version of the table employee, here called**x**, to find people who are in the same department. Finally, you do not want employee number 16 to be displayed, so you should eliminate this case by adding x.empno != 16.

Notice you have to make sure that you do not get the different tables confused and display y.surname and y.forenames since this will just display the surname and forename of employee 16 as many times as there are employees in their department. If there is any risk of confusion you are advised to avoid cryptic labels and use meaningful labels , for example replace "x", "y" with "others", "emp16"; :-

```
SELECT others.surname, others.forenames
FROM employee others, employee emp16
WHERE others.depno = emp16.depno
AND emp16.empno = 16
AND others.empno != 16
```

Tutorial S3

Contents

Subqueries

Nesting of queries is accomplished in SQL by means of a search condition feature known as the subquery. A subquery is a subselect used in a predicate of a search condition. Multiple levels of nesting are permitted. It is often possible to frame a query either by using subqueries or by using joins between the tables.

Some students find subqueries easier to understand than using joins. So if you had difficulty with joins in tutorial 2 you might find this tutorial a lot easier.

The following example was given in tutorial 2 using a **self join** :-

List the surname and forename of all the employees who work in the same department as employee number 16.

```
SELECT x.surname, x.forenames
FROM employee x, employee y
WHERE x.depno = y.depno
AND y.empno = 16
AND x.empno != 16
```

This could be implemented using a **subquery** as :-

```
SELECT surname, forenames
FROM employee
WHERE depno =
   (SELECT depno
    FROM employee
    WHERE empno = 16)
AND empno != 16
```

The subquery in the brackets is evaluated first. The value in SELECT clause is then placed in the outer query, which is then evaluated. So that if the subquery established that employee 16 worked in department number 5, the following outer query would then be evaluated.

```
SELECT surname, forenames
FROM employee
WHERE depno = 5
AND empno != 16
```

The SELECT clause of a SUBQUERY can return ONLY ONE field name which may be associated with zero, one or many values.

Notice also in the previous example that although there are two different occurrences of the table **employee**, they need not be given aliases. This is because the definition of employee in each FROM clause above, is only referred to locally within the predicates of the corresponding SELECT clause.

Aliases may be optionally used as shorthand or to clarify statements. However, at times, it is essential to use an alias, for example to reference a table defined in an outer query. In the following example, if there was no explicit reference to x.depno in the subquery, then it would assumed to be implicitly qualified by y.depno.

Unqualified columns in a subquery are looked up in the tables of that subquery, then in the table of the next enclosing query and so on.

The overall query is evaluated by letting x take each of its permitted values in turn (i.e. letting it range over the **employee** table), and for each such value of x, evaluating the subquery.

This type of query must be done by using subqueries and cannot be done just using joins.

List the employee's number, name and department for any employee with a current salary greater than the average current salary for their department.

```
SELECT x.empno, x.surname, x.depno
FROM employee x, jobhistory
WHERE enddate IS NULL
AND x.empno = jobhistory.empno
AND salary >
   (SELECT AVG(salary)
    FROM employee y, jobhistory
    WHERE y.empno = jobhistory.empno
    AND enddate IS NULL
    AND y.depno = x.depno)
```

Notice that there need be no correlation names for the **jobhistory** tables as they are only used locally and therefore are implicit.

The following examples cover predicates which are used in combination with subqueries. They specify how values returned by a subquery are to be used in the outer WHERE clause.

ANY and ALL

Any or ALL can be inserted between the comparison operator (=, !=, >, >=, <, <=) and the subquery.

List the employees who earn more than any employee in Department 5 :-.

```
SELECT employee.empno, surname, salary
FROM employee, jobhistory
WHERE enddate IS NULL
AND employee.empno = jobhistory.empno
AND salary > ANY
     (SELECT salary
      FROM employee, jobhistory
      WHERE enddate IS NULL
      AND depno=5
      AND employee.empno = jobhistory.empno)
```

The lowest salary in department 5 is £17000, employee 29, the main query then returns employees who earn more than £17000.

List the employees who earn more than all the employees in Department 5 :-

```
SELECT employee.empno, surname, salary
FROM employee, jobhistory
WHERE enddate IS NULL
AND employee.empno = jobhistory.empno
AND salary > ALL
    (SELECT salary
     FROM employee, jobhistory
     WHERE enddate IS NULL
     AND depno=5
     AND employee.empno = jobhistory.empno)
```

Since the greatest salary in department 5 is £29000 , employee number 28, the main query returns all employees who earn more than £29000.

IN and NOT IN

Subqueries can return a list of values. IN and NOT IN are used to check if values are in this list.

List all the employee numbers of anyone who has been on a course in 1988.

```
SELECT empno
FROM empcourse
WHERE courseno IN
      (SELECT courseno
       FROM course
       WHERE cdate BETWEEN '01-JAN-1988' AND '31-DEC-1988')
```

Notice the subquery must contain a reference to exactly one column in its SELECT clause.

EXISTS and NOT EXISTS

EXISTS evaluates to true if and only if the set represented by the subquery is nonempty.

List all the employees who have at least one other employee currently doing the same job as them.

```
SELECT x.empno, surname, x.position
FROM jobhistory x, employee
WHERE x.empno = employee.empno
AND x.enddate IS NULL
AND EXISTS
  (SELECT *
   FROM jobhistory y
   WHERE y.enddate IS NULL
   AND y.position = x.position
   AND x.empno != y.empno)
```

UNION of subqueries

A query may be composed of two or more queries with the operator UNION.

UNION returns all the distinct rows returned by either of the queries it applies to. This means it removes all duplicates.

UNION ALL returns all rows returned by either of the queries it applies to. Duplicates allowed.

List all employees who are in department 4 or 5.

```
SELECT forenames, surname
FROM employee
WHERE depno = 4
UNION
SELECT forename, surname
FROM employee
WHERE depno = 5
```

This UNION could have been done more concisely by using a IN clause.

```
SELECT forenames, surname
FROM employee
WHERE depno IN (4, 5)
```

However, this is not as easy if the two parts of the query are from different tables.

List all employees who were born before 1960 or who earn more than £25000.

```
SELECT forenames, surname
FROM employee
WHERE dob < '01-JAN-1960'
UNION
SELECT forenames, surname
FROM employee, jobhistory
WHERE employee.empno = jobhistory.empno
AND enddate IS NULL
AND salary > 25000
```

Any employee who meets both conditions is listed only once.

Tutorial 4

Contents

VIEWS and Miscellany

VIEWS

CREATE VIEW viewname AS defines a virtual table. A query appears after the AS statement, and the result of executing this query appears as a new table called viewname. However, the data resulting from executing the AS statement is not stored directly in the database. Only the view definition is stored.

Each time a view table is used in an SQL statement, the statement operates on the view's base tables to generate the required view content. Views are therefore dynamic and their contents change automatically as base tables change.

Views can be usefully employed for intermediate tables, and may replace subenquires in order to simplify complex queries

All SELECTs on views are fully supported. Updates, inserts and deletes on views are, however, subject to several rules. Although in this tutorial we make no attempt to update or modify tables, it is important to realise what these modification rules are.

View modifications are not allowed if

- View was created from more than one table.
- View was created from a non-updatable view.
- Any column in the view is derived or is an aggregate function.

Furthermore, inserts are not allowed if

- Any column in the base table was declared as NOT NULL is not present in the view.

Create a view that contains each employees' surname, salary and department name.

```
CREATE VIEW empdepsal(ename, sal, dept)
AS
SELECT e.surname, j.salary, d.dname
FROM employee e, jobhistory j, department d
WHERE e.empno = j.empno
AND e.depno = d.depno
AND enddate IS NULL
```

Removal of a VIEW

This is just the same syntax as dropping a TABLE.

```
DROP VIEW empdepsal
```

differences between sqlplus and activeSQL

There is one important difference between sqlplus and activeSQL. In sqlplus, you have your OWN oracle account, and you do not share this with anyone. When you create a view in Oracle is stays around until you explicitly delete it. Thus you can reuse a view for more than one purpose without having to redefine it.

In activeSQL, you share your namespace with all the other activeSQL users. ActiveSQL tries to make sure that this never causes interference involving the other users. However, one thing it does do is insist that your views are deleted immediately after they are used. If you run a query involving a view, activeSQL will delete that view automatically on your behalf before displaying the results of your query. Thus to use a view in two questions, you must create the view in EACH question. You will not lose marks for reusing a view in two or more questions by copying the view definition into the answer to all the questions. Try to come up with viewname which are likely to be different to your colleagues names - identical name for views are unlikely to cause any problems but different name will definitely NOT cause problems.

No matter what interface you use, it is good practice to delete a view once you are finished with it. Forgetting to delete the view yourself will cost you marks.

Outer Join

One problem which comes up frequently in advanced SQL is losing data in a query where some of the relationships involve NULL. For instance, lets say we want to list ALL empnos in the employee table against how many courses they have been on.

Initially you might simply say:

```
select   employee.empno,count(courseno)
from     employee,empcourse
where    employee.empno = empcourse.empno
group by employee.empno;
```

It looks very reasonable, but running the query produces:

EMPNO	COUNT(COURSENO)
1	2
2	2
7	2
8	2
14	2
15	2
19	2
21	1
22	2

So what happened to all the other empno entries? For instance there is an employee 3, but it does not appear in the table. As employee 3 has not been on any courses, empcourse.empno does not have the value 3, and thus that row of employee is ignored. Whats the solution? There are two possibilities, one using UNION and one (much nicer) solution using OUTER JOIN.

With UNION, we can join together two separate queries and make it appear like a single result table. We can use this to join two queries together, one which is the query above with all the employees with courses, and one query which is all the employees who never did courses. This second query must return the empno attribute, and also a count attribute with a value of 0 (these employees have done 0 courses). Actually, this is quite easy:

```
select   employee.empno,0
from     employee
where    employee.empno not in
         (select empno from empcourse)
group by employee.empno;
```

To join them together list both queries one after another with the word UNION between them. Thus:

```
select   employee.empno,count(courseno)
from     employee,empcourse
where    employee.empno = empcourse.empno
group by employee.empno
UNION
select   employee.empno,0
from     employee
where    employee.empno not in
         (select empno from empcourse)
group by employee.empno;
```

Magic!

Although this works well, it is rather complex and long. Another way is to use OUTER JOIN. This is the same as a normal join, except we warn Oracle that, if there is no value at one side of the join, just pretend there is one. Where we want to allow values to be missing the Oracle syntax as (+) after the attribute which can have no value.

In our case the problem is with

```
employee.empno = empcourse.empno
```

Here, empcourse.empno does not have all the values of employee.empno, and to make the query work we tell oracle to keep going even if there is no equivalent empcourse.empno value. Thus we change this line to:

```
employee.empno = empcourse.empno(+)
```

The whole query changes to :

```
select   employee.empno,count(courseno)
from     employee,empcourse
where    employee.empno = empcourse.empno(+)
group by employee.empno;
```

You could have also wrote:

```
  where empcourse.empno(+) = employee.empno
```

There are complex rules as to how many (+) symbols you can put to the left of an = sign, but in general you can put as many as you like on the right hand side. Trying to utilise more than one (+) in a single SQL statement is strictly for experts only. It is easy to get into a position where Oracle refuses to execute such queries!

As you can see, the is only a few characters longer than the original broken query, and thus is much less complex than the UNION solution. The problem people find with OUTER JOIN is knowing where to put the (+). If you are going to use OUTER JOIN do not just randomly put (+) in your query and then move it until it works. Try to have a logical approach to its placement, or you will be sorry!

Arithmetic operation on dates etc.

Oracle allows you to do simple arithmetic operations on dates.

SYSDATE returns the current date and time.

You can add and subtract number constants as well as other dates from dates. Oracle interprets number constants as numbers of days. For example, **SYSDATE -7** is one week ago.

ADD_MONTHS(d, n) returns the date plus n months.

LAST_DAY(d) returns the last day of the month that contains the date d.

MONTHS_BETWEEN(d1,d2) returns the number of months between dates d1 and d2. If d1 is later than d2, the result is positive, if earlier it is negative. If d1 and d2 are the same days of the month or both the last days of the month the result is an integer otherwise there is also a fractional part.

To calculate the number of days left in a month.

```
SELECT SYSDATE, LAST_DAY(SYSDATE) "Last", LAST_DAY(SYSDATE) - SYSDATE "Days left"
FROM DUAL;
```

Dual is a table automatically created by Oracle along with the data dictionary. It is accessible to all users. It has one column DUMMY and one row with the value X. Selecting from the **dual** table is useful for computing a constant expression with the SELECT command because **dual** has only one row the constant is returned only once.

How many days has each employee been in his or her current job?

```
SELECT empno, SYSDATE - startdate "No of days"
FROM jobhistory
WHERE enddate IS NULL;
```

This works out the number of days between the startdate and today's date. The heading of the column is No of days. However there is also a fractional part which is the fraction of the day.

ROUND(d) returns d rounded to the nearest day.

How many days has each employee been in his or her current job?

```
SELECT empno, ROUND(SYSDATE - startdate) "No of days"
FROM jobhistory
WHERE enddate IS NULL;
```

Rounds the result to the nearest day.

List how many months each employee has been in his or her current job?

```
SELECT empno, MONTHS_BETWEEN(SYSDATE , startdate) "No of months"
FROM jobhistory
WHERE enddate IS NULL;
```

NVL function

NVL returns the normal set function result unless that result is NULL, when it returns the second argument in the NVL function.

To list employees' positions with end dates if not null or else today's date :-

```
SELECT empno, NVL(enddate, SYSDATE)
FROM jobhistory
```

ER Diagram for JOBS

There are a number of tutorial questions on the JOBS database. This can be described with the following ER diagram:

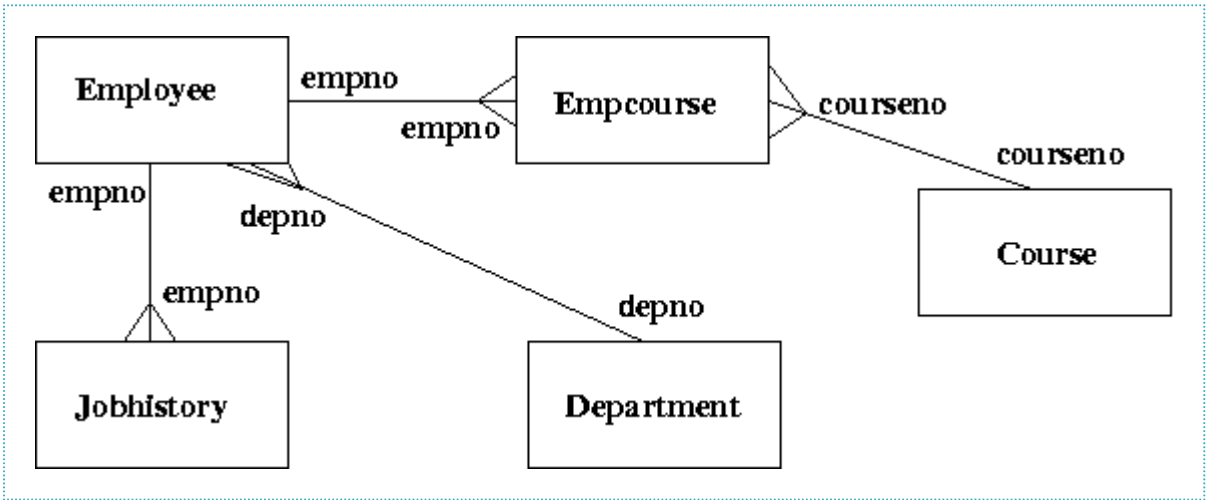


Figure : ER diagram for the JOBS database

ER Diagram for Dressmaker

Contents

There are a number of tutorial questions on the DRESSMAKER database. This can be described with the following ER diagram:

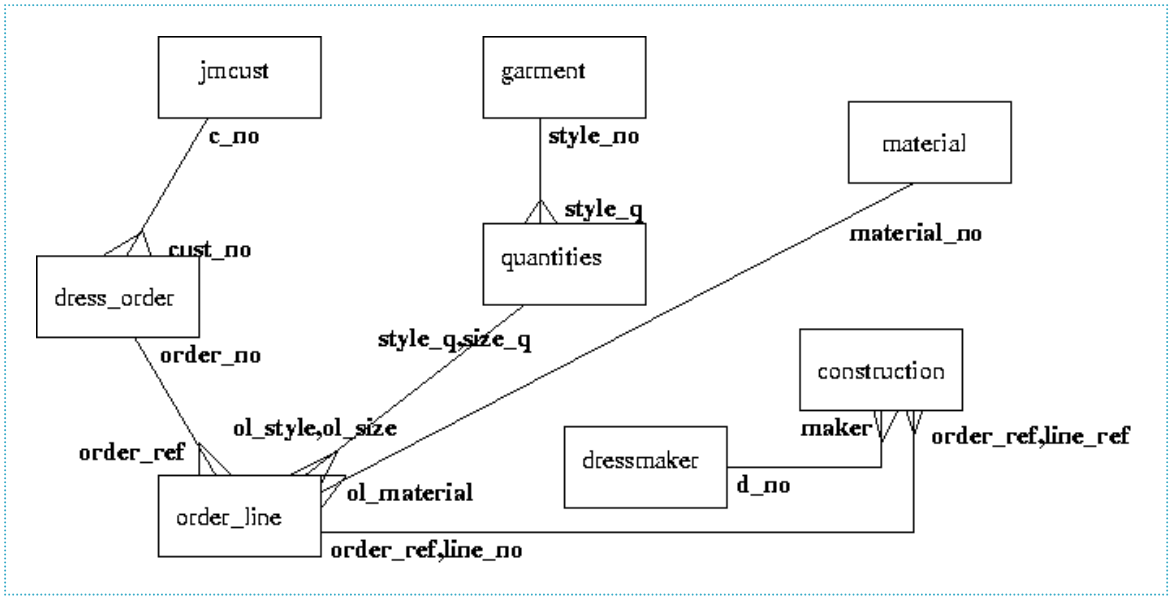


Figure: Dressmaker ER Diagram

The dressmaker tables make use of composite primary keys, and therefore has composite foreign keys. A relationship involving a composite key must include all the attributes involved. For instance, a query needing ORDER_LINE, CONSTRUCTION, and DRESSMAKER would need something like:

```
SELECT *
FROM order_line JOIN construction ON (
       order_line.order_ref = construction.order_ref
   AND order_line.line_no   = construction.line_ref
   ) JOIN dressmaker ON (dressmaker.d_no = construction.maker)
;
```

JMCUST

This table contains information on the customers who use the dressmaker company, including a unique id, the customer name and house number, and the customer's post code.

DRESS_ORDER

If a customer makes an order, it is recorded here. Each order has an order number, and an associated customer number. The date of the order is also recorded. Once all the items in the order have been completed, COMPLETED is set to Y, otherwise it is N. Only uppercase Y or N is used.

ORDER_LINE

Each order that a customer places is made up of 1 or more garments. Each garment of the order is recorded in this table. It is called ORDER_LINE as it represents a single element or line of an order sheet. Each garment in an order is given a unique number (line_no). ORDER_REF is the order number. Garments to be build need a style (trousers, shirts, etc), a size (10,12,etc) and a material (silk, cotton, etc).

QUANTITIES

QUANTITIES explains how much material is needed to build a particular garment. For instance style 4 in size 16 requires 1.5 linear feet of material. Material is sold in a roll, and so someone needs to measure 1.5 feet off the roll and give that to a dressmaker to make the garment.

GARMENT

Each garment has a style number, a description (e.g. trousers), a labour cost and some dressmaker notes (called notions). The labour cost indicates how much money has to be payed to a dressmaker for the time required to make this garment.

MATERIAL

The material to make a garment has a material number, and a fabric name (e.g. cotton). Each fabric may be available in different colours and fabric patterns (like stripes). The COST is the price of the material in linear feet. So one foot off the role of material costs so many pounds.

CONSTRUCTION

This allocates each item in an order to a particular dressmaker. It includes a start date (when it was allocated) and has a finish date of NULL when it is not finished, or the date when it was finished.

DRESSMAKER

Each dressmaker who works for the company is recorded here. Each dressmaker has a name and unique id, plus a house number and a post code. The dressmakers are all freelance, and thus get paid only on completion of a garment.

ER Diagram for Musician

There are a number of tutorial questions on the MUSICIAN database. This can be described with the following ER diagram:

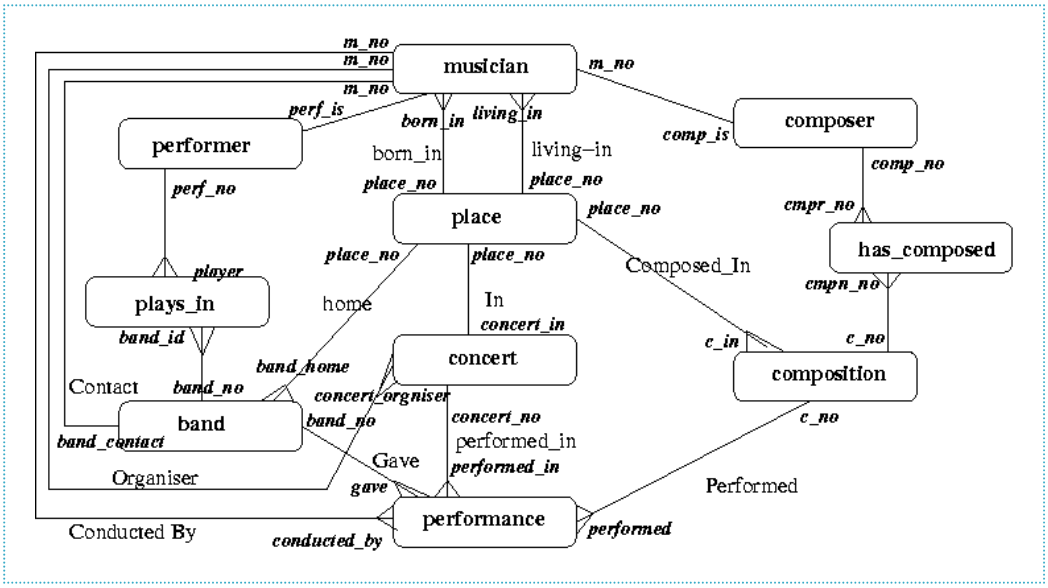


Figure : ER Diagram for the MUSICIANS database

Tutorial - ER Diagram Examples 1-2

Contents

Example 1

A publishing company produces scientific books on various subjects. The books are written by authors who specialize in one particular subject. The company employs editors who, not necessarily being specialists in a particular area, each take sole responsibility for editing one or more publications. A publication covers essentially one of the specialist subjects and is normally written by a single author. When writing a particular book, each author works with on editor, but may submit another work for publication to be supervised by other editors. To improve their competitiveness, the company tries to employ a variety of authors, more than one author being a specialist in a particular subject.

Example 2

A General Hospital consists of a number of specialized wards (such as Maternity, Paediatry, Oncology, etc). Each ward hosts a number of patients, who were admitted on the recommendation of their own GP and confirmed by a consultant employed by the Hospital. On admission, the personal details of every patient are recorded. A separate register is to be held to store the information of the tests undertaken and the results of a prescribed treatment. A number of tests may be conducted for each patient. Each patient is assigned to one leading consultant but may be examined by another doctor, if required. Doctors are specialists in some branch of medicine and may be leading consultants for a number of patients, not necessarily from the same ward.

Tutorial - ER Diagram Examples 3-5

Contents

Example 3

A database is to be designed for a Car Rental Co. (CRC). The information required includes a description of cars, subcontractors (i.e. garages), company expenditures, company revenues and customers. Cars are to be described by such data as: make, model, year of production, engine size, fuel type, number of passengers, registration number, purchase price, purchase date, rent price and insurance details. It is the company policy not to keep any car for a period exceeding one year. All major repairs and maintenance are done by subcontractors (i.e. franchised garages), with whom CRC has long-term agreements. Therefore the data about garages to be kept in the database includes garage names, addressees, range of services and the like. Some garages require payments immediately after a repair has been made; with others CRC has made arrangements for credit facilities. Company expenditures are to be registered for all outgoings connected with purchases, repairs, maintenance, insurance etc. Similarly the cash inflow coming from all sources - car hire, car sales, insurance claims - must be kept of file.CRC maintains a reasonably stable client base. For this privileged category of customers special credit card facilities are provided. These customers may also book in advance a particular car. These reservations can be made for any period of time up to one month. Casual customers must pay a deposit for an estimated time of rental, unless they wish to pay by credit card. All major credit cards care accepted. Personal details (such as name, address, telephone number, driving licence, number) about each customer are kept in the database.

Example 4

A database is to be designed for a college to monitor students' progress throughout their course of study. The students are reading for a degree (such as BA, BA(Hons) MSc, etc) within the framework of the modular system. The college provides a number of module, each being characterised by its code , title, credit value, module leader, teaching staff and the department they come from. A module is co-ordinated by a module leader who shares teaching duties with one or more lecturers. A lecturer may teach (and be a module leader for) more than one module. Students are free to choose any module they wish but the following rules must be observed: some modules require pre-requisites modules and some degree programmes have compulsory modules. The database is also to contain some information about students including their numbers, names, addresses, degrees they read for, and their past performance (i.e. modules taken and examination results).

Example 5

A relational database is to be designed for a medium sized Company dealing with industrial applications of computers. The Company delivers various products to its customers ranging from a single application program through to complete installation of hardware with customized software. The Company employs various experts, consultants and supporting staff. All personnel are employed on long-term basis, i.e. there are no short-term or temporary staff. Although the Company is somehow structured for administrative purposes (that is, it is divided into departments headed by

department managers) all projects are carried out in an inter-disciplinary way. For each project a project team is selected, grouping employees from different departments, and a Project Manager (also an employee of the Company) is appointed who is entirely and exclusively responsible for the control of the project, quite independently of the Company's hierarchy. The following is a brief statement of some facts and policies adopted by the Company.

Normalisation Tutorial

1. A college keeps details about a student and the various modules the student studied. These details comprise
   - regno - registration number
   - n - student name
   - a - student address
   - tno - tutor number
   - tna - tutor name
   - dc - diploma code
   - dn - diploma name
   - mc - module code
   - mn - module name
   - res - module exam result

   where

   ```
   details(regno,n,a,tno,tna,dc,dn,(mc,mn,res))
     dc -> dn
     tno -> tna
     mc,mn -> res
     n -> a
     mc -> mn
   ```

   Reduce the relation DETAILS to third normal form.

2. Classify the following relations as either UNNORMALISED, 1NF, 2NF or 3NF. If the relation is not in 3NF, normalise the relation to 3NF.
   1. ```
      EMPLOYEE(empno,empname,jobcode)
        empno -> empname
        empno -> jobcode
      ```

   2. ```
      EMPLOYEE(empno,empname,(jobcode,years))
        empno -> empname
        empno,jobcode -> years
      ```

   3. ```
      EMPLOYEE(empno,empname,jobcode,jobdesc)
        empno -> empname,jobcode
        jobcode -> jobdesc
      ```

   4. ```
      EMPLOYEE(empno,empname,project,hoursworked)
        empno -> empname
        empno,project -> hoursworked
      ```

3. Identify any repeating groups and functional dependences in the PATIENT relation. Show all the intermediate steps to derive the third normal form for PATIENT.

```
PATIENT(patno,patname,gpno,gpname,appdate,consultant,conaddr,sample)
```

patno	patname	gpno	gpname	appdate	consultant	conaddr	sample
01027	Grist	919	Robinson	3/9/2004	Farnes	Acadia Rd	blood
				20/12/2004	Farnes	Acadia Rd	none
				10/10/2004	Edwards	Beech Ave	urine
08023	Daniels	818	Seymour	3/9/2004	Farnes	Acadia Rd	none
				3/9/2004	Russ	Fir St	sputum
191146	Falken	717	Ibbotson	4/10/2004	Russ	Fir St	blood
001239	Burgess	818	Seymour	5/6/2004	Russ	Fir St	sputum
007249	Lynch	717	Ibbotson	9/11/2004	Edwards	Beach Ave	none

4. Reduce the following to BCNF, showing all the steps involved.

```
Supplier(sno,sname,saddress,(partno, partdesc,(custid,custname,custaddr,quan
  sno -> sname,saddr
  sno,partno -> partdesc
  sno,partno,custid -> quantity
  sname -> sno
  custid -> custname,custaddr
```

Suppliers supply many parts to many customers. Each customer deals with only one supplier. Supplier names are unique. Customer names are not unique.

5. Normalise the following relation to 3NF showing all the steps involved.

```
GP(gpno,cpname,gpadd,(patno,patname,patadd,patdob,(apptdate,apptime,diagnosi
  gpno -> gpname,gpadd
  patno -> patname,patadd, patdob
  patno,apptdate -> apptime,diagnosis
  diagnosis -> treatment
```

6. The table below shows an extract from a tour operator's data on travel agent bookings. Derive the third normal form of the data, showing all the intermediate steps.

batchno	agentno	agent name	holiday code	cost	quantity booked	airport code	airport name
1	76	Bairns travel	B563	363	10	1	Luton
			B248	248	20	12	Edinburgh
			B428	322	18	11	Glasgow
2	142	Active Holidays	B563	363	15	1	Luton
			C930	568	2	14	Newcastle
			A270	972	1	14	Newcastle
			B728	248	5	12	Edinburgh
3	76	Bairns travel	C930	568	11	1	Luton
			A430	279	15	11	Glasgow

7. A software consulting firm wishes to keep the following data for an employee and costing database:
    - employee number
    - employee name
    - employee address
    - salary
    - current job code
    - job history (job promotion code + year)
    - office location
    - telephone number
    - project number
    - project name
    - task number
    - task name
    - project budget
    - task expendature to date
    - department number
    - department name

    There are none, one or mor ejob promotion code/year entries per employee. The office location uniquely depends on the telephone number, and there may be more than one employee using the same telephone and more than one telephone in the one office. Tasks are numbered uniquely only within each project. An employee may be concurrently assigned to more than one project and task, but belongs to one department. Reduce this data to third normal form.

Chapter 10 - Appendix

This is a collection of some useful reference sections

Changes

Contents

This is the changelog for this document

Changes in V3.2

- General tidy and fixed broken characters in Relational Algebra section.

Changes in V3.1

- Fixes to typos and section nesting.
- Image figures are now handled in a more XHTML way.

Changes in V3.0

- Order of SQL and Data analysis sections switched around
- Embedded SQL rewritten as Application Links with considerable changes
- Introduction changed to have an intro to "what is a table"
- The Unit groupings have been removed
- SQL chapters have been completely rewritten and expanded
- Removed: Data Dictionary (some metadata in new section)
- Removed: DBA (some DBA in new section)
- New: DBMS Implementation. Uses some rewritten notes from Storage Structures
- New: Metadata. Includes material rewritten from security, plus is expanded to discuss metadata issues and some aspects of the DBA.
- Changes: Normalisation loses 4NF and 5NF, but gains a simple in-class example
- Exam walkthrough and exam advice removed. This is best done by picking up a past paper and working through that, or using the online tests. Do not use the old slides related to the exam walkthrough (which contained things like assertion/reason questions), as they are badly dated and misleading.
- The notes are not now available as a word document. All the notes are kept in html. I will endevour to produce a good quality printable output of the notes as a pdf, but this technology is limited at the moment.

Plan

CO22001 - Teaching Plan

This is a mock teaching plan for this module. The teaching plan which you actually follow may be different from this.

Week No	Lecture A	Mini A	Lecture B	Mini B	Tutorial
01	Introduction		SQL 1		Logging on / SQL1
02	ER 1		SQL 2		SQL2
03	ER 2		SQL 3		SQL3
04	ER 3		SQL 4	ER1-2	SQL4
05	ER 4		Norm1		ER Diagram 1 + 2
06	Trans		Norm2		C/Work
07	RelAl1		Concur		C/Work
08	RelAL2		Recovery		C/Work
09	Reading Week				Not Supervised
10	DB Security		App Links	ER3-5	Normalisation
11	*Exam Preperation*		DBMS Implement		Spare
12	*Discuss SQL Assess*		*Revision*		Not-Used
13	Revision Week				Not Supervised
14	Exam Week 1				
15	Exam Week 2				