1 Databases.

A db is a large collection of data stored on secondary storage devices. Any number of a.p.'s and online users can use the db at the same time. Usual operations inserting, removing, changing, fetching data

A db imposes a particular structure on the data based on the relationships / associations between data items. ie, structure is determined by the meaning and characteristics of the data and not according to how it will be used by programs.

There are certain standard methods of manipulation, based on the relationships between data rather than to physical addresses.

1.1 Main Components of a DB System.

Controlled by a system called a DBMS. Each DBMS has a special type of structure used to store the data and a special set of standard operations. It acts as an interface between the db and the a.p/user. The only way to access the db is via the DBMS. This simplifies the task of programming and helps control how data is accessed.

For the DBMS to interface the user & data, it must know the structure of the db, specified in a schema (a complete, logical description of an entire db).

The Database Administrator (DBA) is responsible for the db. The DBA will usually design the db and write the schema. The DBA

(1)

(2) will also control use of the db (monitoring, authorising users, imposing standards, etc).

Usually an organisation will only have one db storing all its data. However, most users and a.p's will only be concerned with some portion of the db, termed their <u>view</u>. To define a view, a <u>subschema</u> is written. In many systems the definition of a data item in a subschema may differ from its defn in the schema, which allows a user to view data differently from the way it is actually stored in the db.

1.2 WHY DID D.B SYSTEMS EVOLVE?

(1) Increasing emphasis on data and filing as opposed to number crunching
(2) "Software crisis" - too much software to write/maintain, too few/slow programmers to do this. Simplifies task of software writers.
(3) Growing importance of data manipulation resulted in increasingly sophisticated file management software
(4) Logical step to remove file handling from the program and make it Database Management Software

1.3 COMPARISON WITH FILING SYSTEMS.

Traditionally, data storage has four properties:
(i) files usually are designed to optimise the processing of a particular program or group of programs
[disadvantage - often results in duplication of data]
(ii) data is accessed by one run unit at a time
(iii) programmers need to be fully acquainted with the physical

Characteristics of the data

(iv) the data and ap's are mutually dependent (i.e., change one & you need to change the other)

DEFN A database is a centralised storage to cater for any application

The advantages of data bases are:

- multiple users who may visualise and use the data differently, all concurrently.
- data independence - db should be able to grow and change without affecting established ways of using the data. The needs of a new a.p should be satisfied by the existing data.
- easy & effecient usage - the DBO should strive for clarity (so users can easily know what data is available), flexibility, and ease of use. Data requests should be suitably fast and storage space should be minimized. Report generation languages may make ap's unnecessary.
- protection against loss or corruption - the DBO should preserve the integrity, privacy & security of data. Privacy ≡ rights of access, security ≡ preventing unauthorised changes integrity ≡ ensuring data is accurate.

Disadvantages of DBO

- in some cases, involves extra overload in terms of time/space
- lack of expertise & training
- expensive
- non-portable
- centralised (more vulnerable to corruption / failure / error)

1.4 A TYPICAL DATABASE LIFECYCLE

- determine the application and user information requirements
- define the data to be used
- analyse the logical data relationships
- analyse the ways in which the data will be used
- design the conceptual model
- choose the appropriate DBMS
- convert the conceptual model to a logical one as dictated by the DBMS
- design the physical database
- write and compile the schema
- define, write and compile individual subschemas as required
- load the database
- design and write the application programs
- monitor performance
- adjust physical database to optimise performance ("tuning")

↳ an iterative process occuring whenever new a.p's are required.

1.5 THE APPLICATION PROGRAM (AP)

The is an ordinary program with db facilities embedded. Most of the prog comprises statements in a HOST language (eg C, Cobol). The extra facilities embedded in the host are:

- a statement naming subschema and db being used.
- commands (in DML - the data management language) to the DBMS, which are freely interspersed with host -

- a user work area comprising variables used for receiving/transmitting db data. This is the only storage area common to the db and ap. Error variables also form part of the uwa, set by the DBMS to indicate success/failure of DML commands.

When an ap accesses a record:

- the ap issues a call to the DBMS using appropriate DML commands
- the DBMS examines subschema of ap to find description of record
- the DBMS fetches the db schema & by comparing this with the subschema determines which record is needed. The access is checked for authorisation by the privacy control. Any security violation is logged.
- the logical request is converted to a physical request by the DBMS, which reads the record into its system buffer.
- the DBMS compares schema & subschema and performs any necessary type transformations
- the DBMS transfers the data to the user work area.
- the DBMS provides the user with error status information
- the DBMS monitor collects user stats and updates the audit trail if the db was updated.

1.6 NORMALIZING DATA RELATIONSHIPS.

Normalization is a step-by-step process for analysing data into its constituent entities and attributes. Each relation consists of tuples, where each element in a tuple is specified by an attribute name. Attributes are either identifiers

(keys) or descriptives. Generally one identifier is sufficient within a relation, although sometimes more than one is necessary, in which case we refer to the identifying attributes as composite identifiers. In some cases we may be able to choose between identifiers - these are called candidate identifiers.

Consider for example, an order form, with an order #, supplier #, order date, delivery date, and then columns for part #, description, qty and price. We could represent this as:

ORDER (order #, supplier #, orderdate, deliverydate, (part #, description, qty, price), total price)

However, this is very inconvenient if we wish to access all orders of a particular part, say. The internal group representing the columns is a repeating group of attributes which we should separate out, to give:

ORDER (order #, supplier #, orderdate, deliverydat, total price)
ITEMORD (order #, part #, description, qty, price)

Thus the first step in the normalisation process is to remove repeating groups of attributes, recasting them as new entities. The identifier of the original entity must be made an attribute of each new entity, although it may not be part of the identifier of a new entity. Entities & attributes without repeating groups are said to be in first normal form.

Note that we do not require part description each time a part is ordered. The second step of normalisation removes this sort of redundant data.

DEFN: An attribute α is *functionally dependent* on another attribute β if knowing the value of β uniquely determines the value α is $\alpha \rightarrow \beta$

We have (order #, part #) → qty, and part # → description
(order #, part #) is the composite identifier for ITEMORD but description is functionally dependant on part # alone. This can be overcome by changing our entities to:

ORDER (order #, supplier #, orderdate, deliverydate, totalprice)
ITEMORD (order #, part #, qty, price)
PART (part #, description)

Thus, the second step in the normalisation process is to determine functional dependencies on identifying attributes, and rewrite the entities so that all (sets of non-identifying attributes are functionally dependent on the identifying attribute(s). If the identifier is composite, the other attributes must be functionally dependant on the *whole* of the composite identifier - it must not be possible for a part of the composite identifier to uniquely determine any other attribute in the entity. This is *second normal form*.

In the third step, we look for dependencies between non-identifying attributes only.

Say we have (order #, supplier #, supplier address)

Thus is is second normal form - both supplier # and supplier address can be uniquely determined from order #. However, supplier # → supplier address ie, there is transitive dependence between these. We wish to remove the functional dependencies between non-identifying attributes and make these attributes mutually independent by seperating the orginal attributes into new entities. The resulting model is said to be in 3rd normal form.

So, we could for example, split our entity (order #, supplier #, supplier address) into two:

(order #, supplier #) and (supplier #, supplier address)

A database in 3NF is very stable as it can grow without needing restructuring when new data needs to be inserted. The physical records in 3NF are also highly economical in space consumed.

Compared with 1NF and 2NF, 3NF carries the penalty of extra entity names, although the space saved by removing duplicate fields far outweighs this. The set of normal forms are query equivalent, ie, any query answerable in 1NF is answerable in 2NF and 3NF (after conversion). However, it is possible to extend a database in third normal form to contain information which cannot be validly represented in a db in 1NF or 2NF. It thus has a clear operational advantage.

2 RELATIONAL DATABASES.

A relational database is a collection of time-varying normalised relations of assorted degrees

The degree of a relation is the number of domains
The cardinality of a relation is the number of tuples

DEFN: An n-ary relation R (of degree n) is a set of n-tuples $(d_1, d_2, \ldots, d_n)$ such that $d_i \in D_i$, where the sets D_i are called the domains of R.

Data must be in 1NF for a relational database to be implemented (nb 3NF is in 2NF is in 1NF). Rel. db's show a high degree of data independence. No physical structuring is imposed. Attributes are used so it is not necessary to remember the ordering of attributes in the tuple; similarly, there is no explicit ordering of tuples in the relation. As relations are normalised (at least to 1NF) all values are atomic, there are no duplicate tuples in a relation, and no repeated groups.

2.1 INTEGRETY RULES.

Every relation has a key. If it has several candidate keys, one of these is primie

Rule 1 ENTITY INTEGRITY

No component of a primary key may be a null

Rule 2 REFERENTIAL INTEGRITY

If D is a primary domain (ie some primary key of some relation R is drawn from the domain of D) and A is a non-prime attribute in some relation R' defined on D, then values of A in R' must either be null or equal to V where V is (part of) the primary key value in some tuple of R.

A is called a foreign key in R'.

2.2 RELATIONAL LANGUAGES

There are two types - relational algebra, and relational calculus. Relational algebra is based on the concept of a relation as a set of tuples, and includes operators similar to algebraic set operators. It has three types of operators - project, join and divide. These three have been extended to include select. Rel. alg also includes operators for insertion, deletion and updating tuples in a relation.

Project specifies that certain attributes from a relation are to be copied to a new relation. Any resulting duplicate tuples are removed. For example, if we had a reln ITEM (item#, description, stock) we could get the stock of each item by Π ITEM (item#, stock)
Format: Π relation (attributes)

Join takes two relations and creates a new reln by matching on a common domain.
Format: relation 1 * relation 2 (attr 1 = attr 2)

Relational calculus allows any 1st order predicate calculus expression to be translated into one equivalent relational calculus operation. It is more common than rel. alg, due to its query language type nature

2.3 THE INGRES SYSTEM.

This is a relational calculus system. It provides the query language QUEL whose syntax is:

APPEND relation (targetlist) [WHERE pred]
DELETE rangevar [WHERE pred]
RANGE OF || rangevar IS relation ||
REPLACE rangevar (targetlist) [WHERE pred]
RETRIEVE [INTO relation] (targetlist) [WHERE pred] [SORT BY
    ||attribute [:D] ||] or
  RETRIEVE UNIQUE (targetlist) [WHERE pred] [SORT ....]
CREATE relation (|| attribute = format ||)
DESTROY || relation || or DESTROY {PERMIT / INTEGRITY} relation {|| integer || / ALL}
DEFINE INTEGRITY ON rangevar IS pred
DEFINE PERMIT || option || ON rangevar [|| attribute ||] TO
    user [AT term] [FROM t TO t'] [ON d TO d'] [WHERE pred]
COPY relation (|| attribute = format ||) {INTO / FROM} filename
INDEX ON relation IS indexname (|| attribute ||)
PRINT || relation ||
MODIFY relation TO structure [ON || attribute [:D] ||]
    [WHERE [FILLFACTOR = n] [, MINPAGES = n] [, MAXPAGES = n]]
    VIEWS ("subschemas")
DEFINE VIEW viewname (targetlist) [WHERE pred]

Relational calculus and algebra share basic ideas about data manipulation. These are:

1) operations manipulate files of data rather than records
2) operations state WHAT is to be done, not how
3) the tuple is not the unit of access
4) duplicate tuples are not delivered into the work area
5) relations retrieved can have their attributes in any desired order.
6) data can be retrieved by its value, not its address
7) retrieval is not restricted to access by key - any attribute will do
8) the order in which tuples are delivered into the work area can be specified
9) the number of tuples to be retrieved can be specified
10) library functions exist to perform simple functions on stored data
11) Any query can be translated into a single equivalent retrieval statement.

3 THE HIERARCHICAL MODEL.

A hierarchical database is conceptually made up of a forest of trees of the same type. The nodes of the trees are termed segments (analagous to records) - these are the units of access. All segments other than the root are called dependent segments. Unlike relational and network systems, a hierarchical system stores data in several databases, one for each different tree type.

A tree is defined as a collection of segments and links with the properties:

(i) a segment (type) can have any number of links emanating from it to child segments.

(ii) a segment type can have at most one link leading to it from a parent segment.

(iii) no link from a segment type back to the same segment is permitted (ie no loops)

3.1 TREE OCCURENCES.

For every tree type defined (ie for every "database"), there can be many tree occurences of that type, each having the structure defined by the tree type.

The rules for tree occurences are:

- there is only one occurence of the root segment
- there can be any number of occurences of dependent segments (including none) at the lower levels; each of these must be of the correct segment type.
- a dependent segment occurence has exactly one parent segment in the tree, (so no segment (except the root) can exist in the database without its parent)

3.2 SUBSCHEMAS & SCHEMAS.

Each database is defined by a DBD (Database description). Hence the collection of all DBD's for all the physical databases (forests) will together form the "schema".
An application program will describe its view of the database(s) in its PSB (Program spec. block), which is analogous to a subschema. A PSB consists of several PCB's (Prog. communication blocks), one for each tree type or database.

The rules restricting PSB definitions are:

(i) Each PCB must have the same root segment as the corresponding DBD.

(ii) Any dependant segment type in the DBD can be omitted in the corresponding PCB, if and only if all its descendants are also omitted in that PCB.

(iii) The fields of a PCB segment type can be in a different order to that in the corresponding DBD segment, and one or more fields can be omitted entirely. Thus a PCB is a sub-hierarchy of the corresponding DBD.

Segments / fields appearing in a PCB are called "sensitive" segments / fields. An a.p. can reference only sensitive segments.

3.3 HIERARCHICAL DML's.

Retrieval is in preorder. The main search commands are:

GET UNIQUE segment [WHERE (predicate)] (abbrev GU)
GET NEXT segment [WHERE (predicate)] (abbrev GN)
GET NEXT WITHIN PARENT (abbrev GNP)

If the data is to be manipulated, then GET HOLD must be used instead of GET. This can then be altered by INSERT, DELETE & REPLACE. In deletion, all segments (sensitive & non-sensitive) which are descendants are also deleted.

3.4 EVALUATION

Hierarchical databases form a natural and simple way of modelling hierarchical structures in the real world. More complex relations are problematic, however. Other disadvantages include:

- deletion of parent segments resulting in deletion of all dependant data
- updating a child segment means scanning ALL parents and their children, because of the redundancy that exists.
- inserting a child segment may require storing a "dummy" parent.

In general, the hierarchical model is simple and easy to use in simple cases. However it is incapable of representing complex and symmetrical relationships without enormous processing overheads or data redundancy.

3.5 Implementation

Root segments are usually hash or indexed sequential for fast access. Dependant segments are either followed through the tree (pointers), or are physically arranged in preorder. Linking can be hierarchical (each segment points to all its children) or child-twin (each segment points to its 1st child of each type, its next twin, and optionally also to its last child). Reverse pointers can also be used. The tree can "mix" both types of pointers. The DBA chooses from the above for each DBD (tree type).

4 The Network (CODASYL) Model

The network model was designed by the CODASYL DBTG (Data Base Task Group) to be compatible with COBOL. It consists of records arranged in sets, and can represent both hierarchical and network structures.

4.1 Data Constructs

Data item: unit of homogeneous data
Data aggregate: collection of homogeneous data items.
Vector data aggregate: ordered data aggregate
Repeating group: collection of data aggregates or items occurring multiple times
Record: collection of data items or aggregates (logical concept, not physical)
Set: collection of records
Set Member: record belonging to that set.

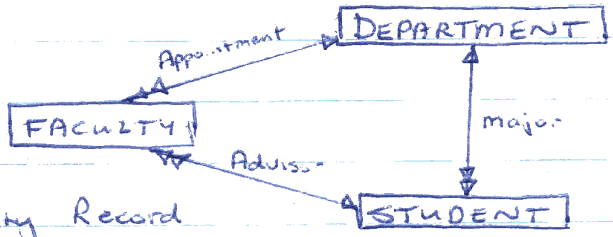
Set Owner: Record identifying a particular set occurence
Realm: subset of database records (≡ view)

A database consists of a collection of sets (of records)
A set comprises one owner record type, and any number of member record types, ie 1:∞ many relationships.
Many:many relations cannot be directly represented.

The rules for set formation and interconnection, plus a summary of set characteristics, is:

(1) A set is a collection of records
(2) There are an arbitrary number of sets in the database
(3) Each set has <u>one</u> owner type, and none or more member record types (can have "empty set" as member)
(4) Each owner record occurence defines a set occurence
(5) There are an arbitrary number of member record occurences in one set occurence
(6) Set records can be ordered
(7) Set records can be accessed directly by specifying the values of record data items
*(8) A record may be the member / owner of more than one set type.
*(9) A record may not be a member of two occurences of the same set type (⇒ no many:many rels)
*(10) A record may not be a member and an owner of the same set type.

A simple example is given overleaf



Department Record
Owner of appointment set
Owner of major set

Faculty Record
Owner of advisor set
Member of appointment set.

Student Record
Member of major set
Member of advisor set

Every record type defined has a name and a location mode clause associated with it. The latter can be one of four types:

(a) direct - record is located by directly supplying its key (→ address)

(b) calc - the key of the record is calculated using the values in some of its data items

(c) via <setname> set - the record is located by firstly locating a <setname> set occurence and then scanning its members.

(d) index sequential - using any field of the record as the search key.

Every set type has a set occurence selection clause associated with it, as well as a name, etc. The former can be one of two types:

(a) location mode of owner - the owner occurence is found as specified by the location mode.

(b) current of set - the current record of this set type is found, and the set it belongs to (as either owner or member) is the set occurence chosen (i.e. we locate the latest occurence, while in (a) we locate a specific set).

Every member of every set is given a membership class, either manual or automatic.

- automatic :- if a record type R is an automatic member of a set type S, then every R record occurence in the DB will be a member of some S occurence.
- manual :- participation of an R occurence in an S set occurence is optional.

4.2. CURRENCY INDICATORS

For each run-unit under its control, the DBMS keeps a table of currency indicators. These specify which record was most recently accessed by the run unit.

- current of run unit : the most recently referenced record
- current of R : the most recently accessed R occurence
- current of S : the most recently accessed record belonging to an S occurence.

The address of the current record is stored in the currency indicator for rapid access.

4.3 DMS 1100 COMMANDS