CS352 Lecture - The Hierarchical and Network Models

last revised September 5, 2002

Objectives:

1. To expose students to the “flavor” of the hierarchical and network models

I. Introduction

A. We have already noted that four of the five database models we will discuss have been used for commercial systems. All four models store entities in essentially the same way (as a series of bytes in a disk file somewhere). Where they differ is in how they represent relationships.

B. In this series of lectures, we will be introducing the two oldest models: the hierarchical model and the network model. While the relational model dominates the database world today, there are two good reasons for looking briefly at these older models:

1. They are still used, especially for very large “legacy” systems. In fact, for large applications they have a performance advantage that may be one of the factors helping to keep these systems alive. (Inertia is probably also a factor.)

2. We will see, when we get to object-oriented models later in the course, that there are some similarities between the way OO models handle relationships and the way these models do.

C. We will follow the course of history by looking at the hierarchical model first, then the network model. We will <u>not</u> attempt to delve into details of the DDL’s and DML’s - our goal is brief familiarity, not an ability to actually work with these models.

D. Since you already have some familiarity with the relational model (from CS211), it is important to note that both of these models have a very different “flavor” from the relational model.. Network DBMS's have a very different "flavor" than most relational DBMS's.

1. Relational DBMS's are typically designed to support two modes of operation:

a) An interactive mode, in which a user can formulate queries in an appropriate query language, without the benefit of a traditional program per se.

b) An embedded mode, in which query language statements are embedded in an applications program that uses the DBMS in much the same way it might use a subroutine library.

c) If anything, relational DBMS's tend to exhibit a bias toward the interactive model.

d) The hierarchical and network models, however, are designed for embedded use.

(1) Their data manipulation languages (DMLs) are clearly intended to be embedded in a higher-level-language program, and relies on the facilities of the host language to provide many of the IO and formatting functions that are needed when processing a query.

(2) A typical commercial product must therefore include provisions for mixing programming language and database constructs in the same program. This may be accomplished in one of two ways:

(a) A modified compiler that recognizes database statements and converts them into DBMS procedure calls.

(b) A preprocessor that converts embedded DML to explicit calls to DBMS procedures before the program is submitted to a regular compiler.

(3) In principle, hierarchical and network DML can be embedded in any higher-level language; but it is more suited to some than others. The host languages that fit most closely are languages like COBOL and PL/1.

II. The Hierarchical Model

A. The hierarchical model is the oldest DBMS model, but is still widely used. The best known commercial implementation is IBM's IMS, first developed in the late 1960's, but still in use today after more than 30 years. (Not many software products can make such a claim!)

B. The hierarchical model (and the network model as well) makes use of “records” to model entities and “links” to model relationships.

Note: the hierarchical model uses the terms “record” to refer to an individual record and “record type” to refer to a collection of records having the same format. This corresponds to our use of the terms "entity" and "entity set" in the ER model.

1. A link connects two records to one another (thus the model only directly supports binary relationships). A link can be implemented either by physical proximity on the storage medium or as a disk-based pointer - i.e. a pointer from one location on disk to another.

2. A link cannot have attributes - i.e. it is a simple connection between two records, nothing more.

3. If we are modeling a relationship with attributes, or that is not binary, or is many to many, we end up having to create a record type for the relationship, which is then linked to the participating records. (We will see an example of this shortly.)

   Thus, in the simplest case, a relationship in an ER diagram can be modeled directly by a link. In more complex cases, a relationship is modeled by a record that is linked to the records representing the participating entities.

C. The hierarchical model gets its name from that the fact that it requires that in the database scheme, the record <u>types</u> must be organized as trees (hierarchies).

1. Every record type is a member of exactly one primary hierarchy, and appears just once in that hierarchy.

2. The database may also contain secondary hierarchies, and a record type can be a member of any number of these - but can only appear at one place within any one secondary hierarchy.

3. There is a parent-child relationship between record types within any given hierarchy. Within a given hierarchy, there is one root record type. It, in turn, may be the parent of one or more child record types, each of which may, in turn, have its own children ... (Of course, the parent-child relationship will be represented by a link, so it should correspond to some relationship in the underlying reality)

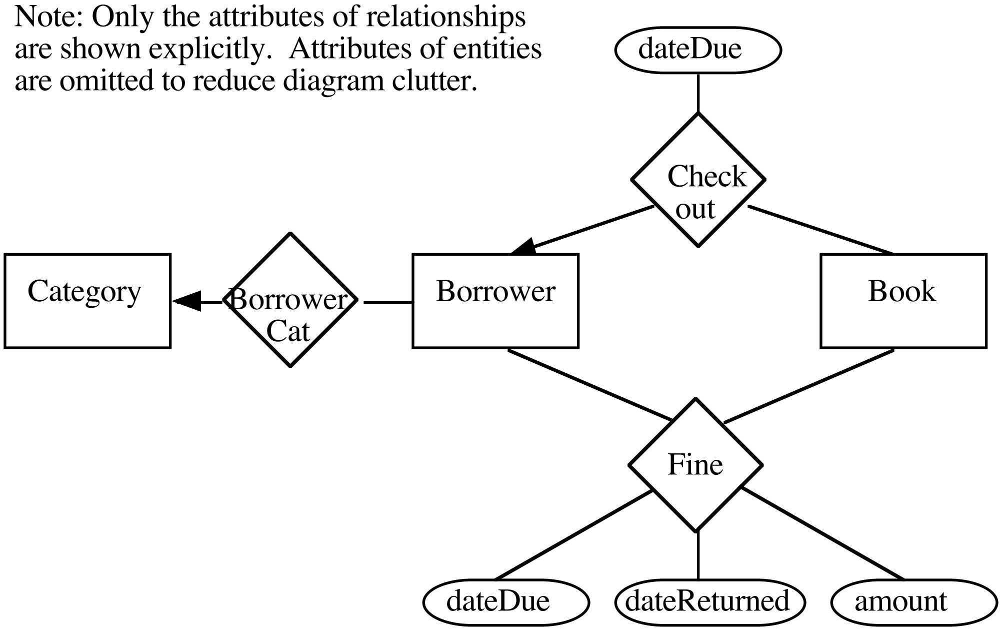
4. Hence, within any given hierarchy, a record type has exactly one parent record type (since it can appear only once in the hierarchy) - except, of course, the root record type has no parent.

5. This might seem, at first, to be quite restricting. Actually, it does not pose a significant problem, for two reasons:

a) The “real world” we are modeling often has a hierarchical structure to begin with - especially in the realm of business data processing, which is the “world” in which the hierarchical model was developed.

b) Realities that are non-hierarchical can be converted into a suitable hierarchical representation.

D. Example: Consider a simple database for a library represented by the following E-R diagram:


Note: Only the attributes of relationships are shown explicitly. Attributes of entities are omitted to reduce diagram clutter.
dateDue
Check out
Category
Borrower Cat
Borrower
Book
Fine
dateDue
dateReturned
amount

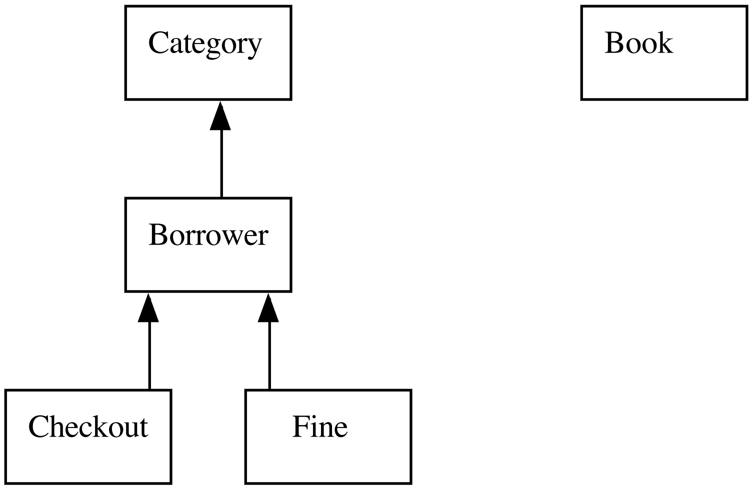
1. Each of the entity sets - Category, Borrower, and Book - will become a record type in the hierarchical scheme.

2. Two of the relationships - Checkout and Fine - will also need to be modeled by record types, because they have attributes. (There is a way to avoid this in the case of Checkout - but that would spoil the example!)

3. The Borrower-Cat relationship can be represented by a simple link in the hierarchical database scheme. In addition, we will need links to connect the record types we create for Checkout and Fine to Borrower and Book in each case.

4. This might be structured as a hierarchical database with two primary hierarchies.

   Primary Hierarchies

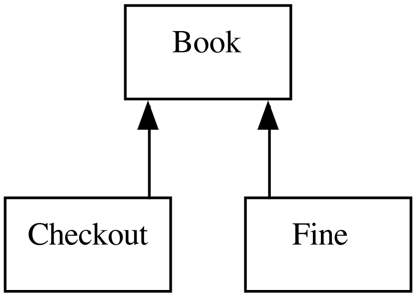
   Note that each record type has an arrow pointing to its parent type.

   Note: we need two primary hierarchies, because there is no way to legitimately incorporate Book in the first hierarchy: there is no relationship between Category and Book, and a Book cannot be reasonably said to be a "child" of a Borrower, Checkout, or Fine, since books can exist without ever having been related to a borrower, checkout, or fine. Moreover, if book were made a "child" of one of these as a "parent", it couldn't be a child of the other two.) Because we must include each record type in exactly one primary hierarchy, we need a second primary hierarchy.

5. In addition, we need a secondary hierarchy to represent the relationships between Book and Checkout and between Book and Fine. This needs to be a secondary hierarchy, because each of the record types involved has already appeared in a primary hierarchy, and in the primary hierarchy Borrower is the parent of Checkout and Fine, and a record type can have only one parent in any one hierarchy.

   (Note, then, how the problem of checkout needing two parents has been solved by PARTITIONING the structure into two hierarchies.)

Secondary Hierarchy



6. A diagram incorporating the above three hierarchies is called a TREE STRUCTURE diagram.

E. In addition to the requirement that the database <u>scheme</u> be hierarchical, this model also requires that the database <u>instance</u> have a hierarchical structure.

1. In particular, each record belonging to a "child" record type within any given hierarchy can relate to at most one record belonging to the "parent" type within that hierarchy. Thus:

a) Each Borrower relates to exactly one Category - which comports with reality.

b) Each Checkout and each Fine relates to exactly one Borrower and exactly one Book. (A Borrower can have many books checked out, but each Checkout record relates to one specific book; a Borrower can incur fines for many books, and a given book can give rise to fines for many borrowers, but each Fine record relates to one specific Borrower and Book.) (I.e. relationships must be either one to one or one to many going down the tree.)

c) What if we need a many to many relationship, or a many to one relationship going down the tree?

ASK

We have to turn the relationship into a record type, with links to corresponding entities in such a case.

Example: Even if the fine relationship in our original ER diagram didn't have attributes, it would still need to become a record type because it is many to many.

2. The database INSTANCE, then, is a forest of rooted trees, each of which corresponds to one of the primary hierarchies.

   a) The root of each tree is a dummy record which owns all the level one records, each of which owns any number of level 2 records, etc. (Note that the owned records need not all be of the same type. For example, with the above hierarchies it is possible for a single Borrower record to own both Checkout and Fine records.)

   b) Thus, each record - except for the dummy records at the roots of the hierarchies - is "owned" by some other record.

   Note, then, that in the instance corresponding to the first hierarchy each Borrower record is "owned" by some Category record, and each Checkout and each Fine is "owned" by some Borrower - we don't have (and can't have) unattached Borrowers, Fines, or Checkouts.

3. This leads, then, to a simple way of storing the database instance - a storage structure based on preorder traversal of the primary hierarchy/hierarchies.

   Example: suppose we have a library database instance with two categories: honcho and peon; five borrowers: Aardvark (peon), Cat (honcho), Dog (peon), Fox (peon), and Zebra (honcho); four books (two of which are checked out to Dog and one to Cat), and three fines (all owed by Aardvark). This instance might look like this:

   (dummy record)

Category:	honcho ...
Borrower:	Cat ...
Checkout:	due 11-15-02
Borrower:	Zebra ...
Category:	peon ...
Borrower:	Aardvark ...
Fine:	$ 1.45 ...
Fine:	$ 0.25 ...
Fine:	$ 0.40
Borrower:	Dog
Checkout:	due 11-10-02
Checkout:	due 11-10-02
Borrower:	Fox ...

   (dummy record)

Book:	QA76.093 ...
Book:	RZ12.905 ...
Book:	LM925.04 ...
Book:	AB123.40 ...

   Note: in the earliest days of the hierarchical model, disk capacities were relatively small, and it was common to store a large database on magnetic tape, an inherently sequential medium. This preorder structure lent itself quite well to this. When the database is stored on disk, it is not necessary that the records be physically stored in preorder, though they can be.

4. What about storing the secondary hierarchies? Rather than replicating the records, these are composed of <u>virtual records</u>

   a) When a given record must appear more than once in the database, a single master copy is made (containing all the attributes).

   b) All other occurrences are replaced by special records that point to the main record. These latter records can be very small (just a pointer), so space utilization problems are minimized; and since they contain no data inconsistency problems are also avoided.

   c) This is transparent to the user. When a user accesses a virtual record, the system gives him the data from the corresponding real record so he never knows the difference.

F. Hierarchical DML.

1. It is not our goal to cover hierarchical DML in detail, but some exposure to the DML used to manipulate a hierarchical database helps to give something of the "flavor" of the model.

   In contrast to the network and relational models, there is no industry standard query language for hierarchical databases. Rather, the query language is specific to a particular product. We will consider DL/I, the query language of IBM's IMS product.

2. Regardless of how the database is actually stored physically, DL/I is based on the preorder storage model we just considered

3. A key concept in the DML is the notion of a program work area.

   a) One piece of information in the program work area is a <u>currency pointer</u> for each tree, which points to the most-recently accessed record in that tree.

   b) In addition, for each record type, there is a <u>template</u> that holds the values from the most-recently accessed record of that type.

4. The following are simplified versions of some basic DL/I retrieval commands. Each manipulates the currency pointer for some tree, and

also fetches data from the database into the template for the appropriate record type. We will not consider data storage/modification commands; these are similar in style to those for the other models. Our examples will be based on the example instance we just looked at.

a) Get unique (GU) - gets the FIRST record of some specified type in some hierarchy, possibly matching some condition

Example: GU Borrower in the first primary hierarchy would get Cat ...

GU Borrower (last_name = “Aardvark”) in the first primary hierarchy would get Aardvark ...

b) Get next (GN) - gets the NEXT record of some specified type within some hierarchy following preorder from currency pointer for that hierarchy (i.e. the result of the last get).

Example: Following a GU that got Cat ...

GN Checkout in the first primary hierarchy would get 11-15-02

GN Borrower in the first primary hierarchy would get Zebra ...

c) Get next within parent (GNP) - like GN, but stays within subtree owned by result of last get

Example: Following a GU that got Cat, GN Checkout would get each of the three Checkout records in turn. If we wanted only checkouts to Cat, we could use GNP instead, and we would stop after the one pertaining to Cat.

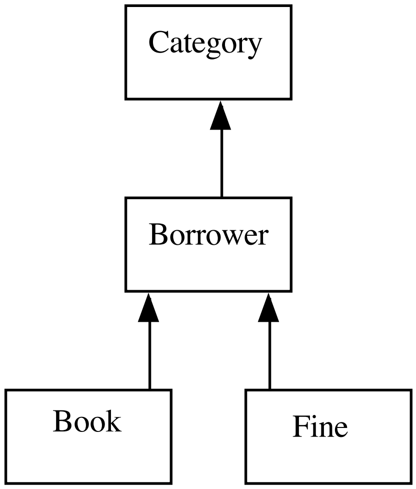
5. An important characteristic of hierarchical DML (and network DML as well) is that it is navigational - that is, to access the data one wants, one navigates to it.

Example: Print out the total of all fines which Aardvark owes:

•Navigate to the Borrower record for Aardvark (using GU)
• Use GNP to navigate to each of the Fine records for with Aardvark is the parent; add the amounts for each

When writing code to access a hierarchical database, one must remember where one is in the navigation of the data.

G. It should be apparent, from our discussion of the physical representation of the database, that avoiding secondary hierarchies is desirable (because of the need for virtual records.) It turns out that, by slightly changing the way we store information, we can actually represent our library database scheme as a single hierarchy.



1. Because a book can only be checked out to one borrower at a time, we can make date-due an attribute of Book, and eliminate the need for a separate Checkout record type.

2. We still have an apparent problem, though - what do we do with a book that is not checked out to someone - hence not the "child" of some borrower?

   ASK

   We can create a "dummy" borrower that owns all the books not checked out to someone else - so every Book is now associated with some Borrower.

3. We can eliminate the need for a relationship between Book and Fine by keeping necessary information about the Book (e.g. perhaps its title) in the Fine record.

III. The Network Model

A. We now turn to the network model, that was historically the successor to the hierarchical model (though IBM's hierarchical IMS product is still very much in use!) Although the network model can be viewed abstractly, it is normally thought of in connection with a particular scheme called the CODASYL DBTG model.

1. CODASYL is the Conference on Data Systems Languages, and is an industry group whose primary claim to fame is that it is the maintainer of the COBOL language.

2. In the late 1960's, CODASYL formed a database task group to seek to develop a standard for database systems that could be used by programs written in COBOL and other languages. The DBTG report of 1971 - and subsequent revisions - has become a defacto industry standard for network databases. Thus, for all intents and purposes a network DBMS is a DBMS based on the CODASYL DBTG proposal.

3. Note that the CODASYL DBTG model is an industry standard - not a particular product. There are many different implementations of this model by various computer manufacturers and third-party vendors, just as there are many implementations of a standardized language such as COBOL or C. (As of 2002, the two major implementations still being developed are Oracle's DBMS (not to be confused with its relational product, and Computer Associates' IDMS)

B. In many ways, the network model resembles the hierarchical model. Indeed, the key difference is the removal of the requirement that the database scheme be hierarchical - the network model supports a scheme that is an arbitrary graph structure.

Thus, a network database scheme consists of a collection of

1. Record types, which can be used to represent entity sets.

2. Links. Each link joins exactly two records, and thus, in its simplest form, can be used only to model binary relationships with no attributes.

   In the CODASYL DBTG form of the network model, links are restricted, as in the hierarchical model, to modeling one to one, one to many, or many to one relationships - many to many relationships cannot be modeled directly by links.

C. In the CODASYL DBTG model, one does not normally work directly with links, but rather with a higher-level construct called an "owner coupled set".

1. Because of the restriction on the mapping cardinality of links, in any linkage it is possible to speak of one of the two participating record types as being the OWNER type.

a) The owner type is determined as follows:

(1) If the linkage is one-to-many or many-to-one, then the "one" record type is the owner.

(2) If the linkage is one-to-one, then either record type can be the owner; the database designer must choose one to fulfill this role.

b) One point of notational confusion arises in connection with this:

(1) In drawing E-R diagrams, the standard convention is that when a relationship is one-to-one, we put arrows on both of the lines joining it to the entity sets involved; if one-to-many or many-to-one, then we put arrows on the line joining the relationship to the "one" entity set, and if many-to-many we use no arrows.

(2) However, in drawing data-structure diagrams, (also known as Bachman diagrams), the original convention was to have arrows point from the owner record type to the member record type - i.e. to the "many" record type. Some more recent writers (including the authors of our text) have turned this around to conform to E-R diagram usage.

2. The set of all links of a given type is called a DBTG-SET or OWNER-COUPLED SET. Within such a set, the subset consisting of all the links connecting to the same owner is called a SET OCCURRENCE. The non-owner records in a set occurrence are called MEMBER RECORDS.

   Example: The set of all links between Fines and Borrowers owing them is a DBTG-set; the set of links between Anthony Aardvark and the fines he owes is a set occurrence. The Anthony Aardvark record is the owner of this set occurrence; each individual fine is a member of this set occurrence.

3. Note that each record of a given type can participate in at most one set occurrence of a given type.

   Example: In the owner coupled set BorrowerFines, each Borrower record is the owner of the one occurrence it participates in. (This occurrence may consist only of the owner record; it may have no members if the Borrower has no Fines - but we still speak of the Borrower record as the owner of the set occurrence) Each of a borrower's fines is a member of that occurrence and no others.

(This property follows from the restriction against many to many links, and was the motivation for imposing this restriction, since it makes the physical representation of sets much cleaner, as we shall see.)

4. As was the case with the hierarchical model, in translating a scheme from an E-R model design to a network implementation, it sometimes becomes necessary to introduce new kinds of records to model certain kinds of relationships not directly capable of being modeled by DBTG sets based on simple links.

   When a relationship cannot be directly modeled by an owner-coupled set, it must be modeled by creating a new record type called a link-record.

   a) Note well: a link-record is to be distinguished from a link. It has all the properties of any other record type, but serves to model a relationship set, not an entity set.

   b) Each new link-record type record will be linked to exactly one record of each type participating in the relationship. (In each case, the other record type will therefore be the owner type.)

D. The CODASYL model supports a very efficient physical representation for owner coupled sets.

1. We have seen that, because links are never many-to-many, the set of all links of a given type can be partitioned into a set of occurrences, each of which contains one owner record plus all of the member records to which it is linked.

2. Further, each record participates in at most one set occurrence of a given type - either as owner or member.

3. How is a set occurrence represented on disk?

   a) The DBTG model is a logical model, so an implementation is not directly specified.

   b) But the model clearly has a natural representation (which was assumed in developing the model in the first place.)

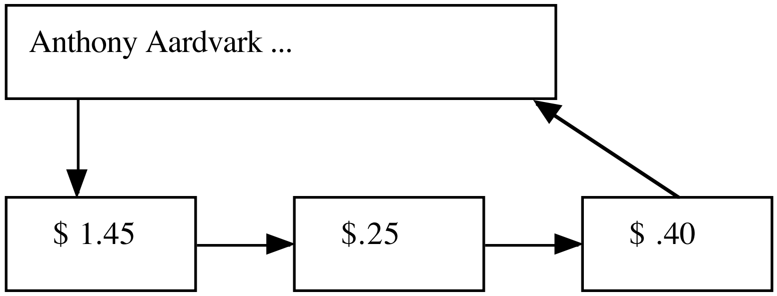
   c) Each record of a given type will contain two kinds of fields:

   (1) Fields for each of its attributes

(2) One or two fields for each type of link to which it can be a party. Each of these fields will hold some sort of physical pointer to another record.

d) In the simplest representation, all of the members of a given set occurrence are linked by physical pointers into a circularly linked list, containing one owner record and any number of member records.

Example: Suppose Anthony Aardvark owes fines for 1.45, .25, and .40. Then the owner-coupled set representing the relationship between him and his fines would look like this:



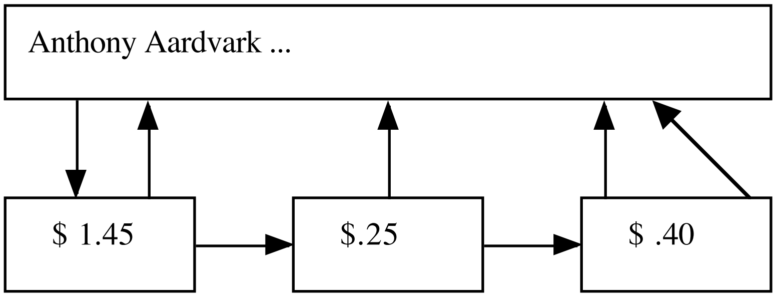
Note: the arrows here are used in the sense of pointer variables, not as in E-R diagrams.

(1) Because each record can only participate in one set occurrence of any given type, a single pointer field in each record for a given set type is always sufficient.

(2) The circular list establishes an ordering among the member records. From the owner record, it is possible to go directly to one member record, from that to the next etc. until one gets back around to the owner. This ordering is reflected in the language features that operate on sets.

(3) From any member record, one can get back to the owner record by following pointers - though this could be costly if the set occurrence has many records.

e) To speed movement from a member record in a set occurrence to the owner of that occurrence, some implementations include a second pointer field in each member record containing a pointer directly to the set owner - e.g.



f) This representation supports the basic operations that the DBTG model allows on sets. These operations are often called "navigating the database", since they involve a logical movement from one record to another.

(1) From the owner record of a set occurrence, find the first member of the set occurrence. [Note: if the owner record is not in a set occurrence - i.e. it owns no member records - this operation will set an error-flag instead.]

(2) From a member of a set occurrence, find the next member record (if there is one). [Note: if the next record would be the owner, meaning the current record is the last member of its set, then this operation will set an error-flag instead.]

(3) From any member of a set occurrence, find the owner record. [This must always succeed unless the database is corrupt.]

E. Categories of Set Membership

1. For each DBTG set, each record of the owner type is ipso-facto part of a set occurrence (the one which it owns) as soon as it is created. This set occurrence may or may not contain member records, though.

2. For each DBTG set, each record of the member type can be a member of at most one set occurrence, however, whether it has to be a member of one occurrence is dependent on the logic of the relationship being modeled. Therefore, for each owner-coupled set, part of the definition is a specification of insertion and retention rules for member records.

3. A set may specify one of two types of insertion for member records. This rule deals with how a member record initially gets into a set occurrence.

a) Automatic insertion specifies that a newly-created record of the member type is automatically inserted at the time of creation into an appropriate occurrence of that set (where the appropriate occurrence must be specified as part of the insertion process.)

   Example: For the owner-coupled set linking borrowers to fines, we would presumably specify automatic insertion, since it is not meaningful to have a fine in the database that no one owes.

b) Manual insertion specifies that a newly-created record is not a member of an occurrence of that set, but can be explicitly connected to one either at the time the record is created or at a later date.

   Example: If we used a DBTG set to model books checked out (ignoring the date due problem), we would specify manual insertion, since it is meaningful to allow a book to be in the database without being checked out to some borrower.

4. A set may specify one of three types of retention for member records. This rule deals with the possibility of removing a record from a given set occurrence to either transfer it to another set occurrence or leave it unconnected.

   a) Fixed retention specifies that once a record is a member of a particular set occurrence, it may not be removed from that set occurrence.

      (1) The only way to remove it from the set is to delete the record from the database altogether.

      (2) If the owner record is erased from the database, all member records it owns are also automatically erased from the database.

      Example: Fixed retention is appropriate for the set linking borrowers and fines they owe. Once a fine has been assessed against a given borrower, it would not be meaningful to move it to a different borrower.

   b) Mandatory retention specifies that a member record can be moved from one set occurrence of a given type to another; but it must be a member of some set occurrence.

      Example: Suppose we established various categories of borrowers, each with certain rules pertaining to number of books that can be out at once, length of loan etc. We might then establish a Category record type - one record for each category. Such records might be made owners of Borrower records - i.e. each Borrower record is a member of the set of borrowers of a certain Category.

In this case, it would be meaningful to move a given borrower from one category to another, but not to have a borrower that is not a member of any category.

c) Optional retention specifies that a member record can be removed from a set at will, and connected to another at some other time if desired.

Example: If we used a DBTG set to model books checked out, we would specify optional retention. At one moment a book may be "owned by" (checked out to) one Borrower, then it may be returned and thus "owned" by no one; then it may later be checked out to someone else.

5. Set insertion and set retention are orthogonal concepts, so six different combinations are possible. However, some combinations are not likely to be useful (e.g. manual insertion plus mandatory retention.) Other combinations reflect particular patterns in the underlying reality.

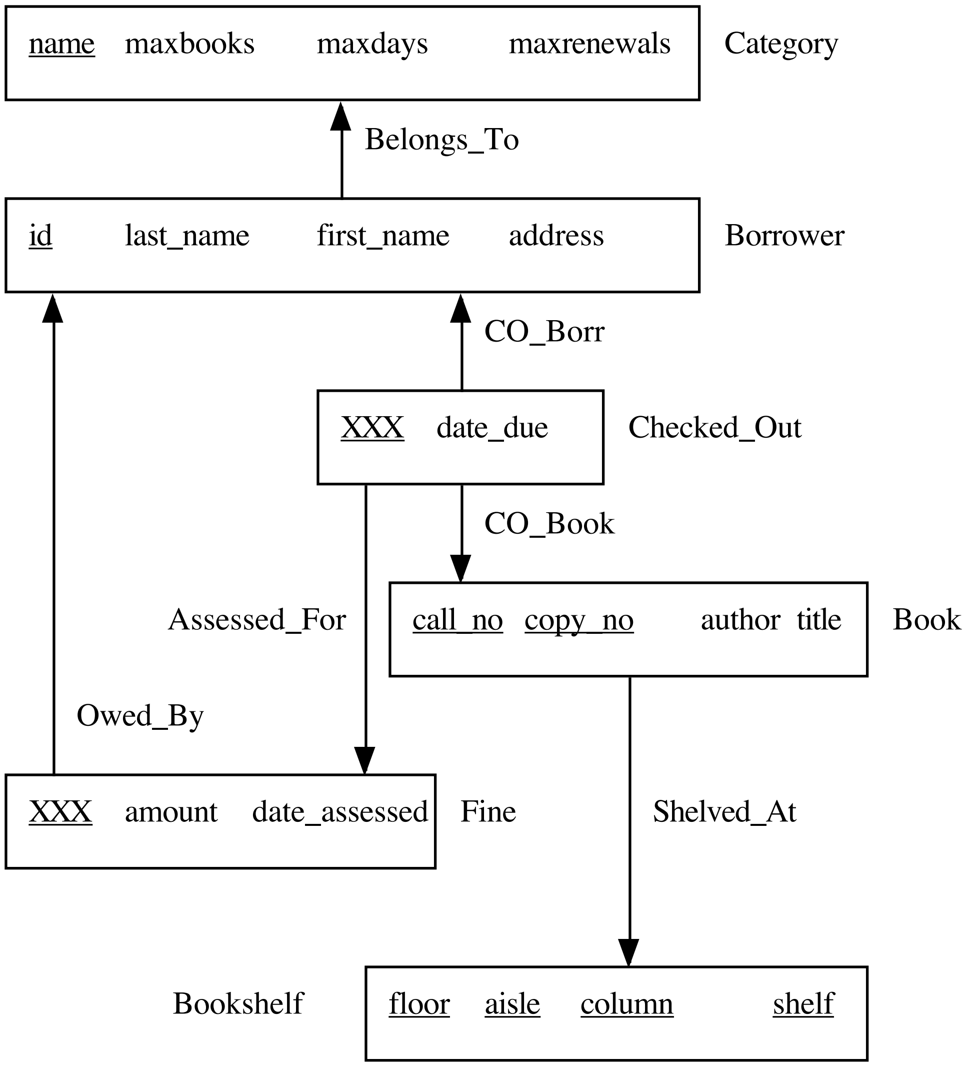
   a) In particular, if a relationship with a participation constraint (as discussed in conjunction with the ER model) is represented by a DBTG set, then the set will likely be declared with automatic insertion and mandatory retention.

   b) If a relationship with an existence constraint (as discussed in conjunction with the ER model) is represented by a DBTG set, then the set will likely be declared with automatic insertion and fixed retention.

   c) Most other relationships will, if modeled by a DBTG set, use a set declared with manual insertion and optional retention.

6. Note that each set has its own set of rules. Thus, if a given record type is a member record of two different sets, two different sets of rules may pertain to records of that type - one for each set.

F. Example of a Relatively Complete Library Database (using notation in the text, with arrows pointing from member to owner in sets.) (Note: this has been contrived a bit to illustrate various possibilities for sets!)



Note: primary keys for each record type are underlined. XXX means that a system-generated primary key is used.

1. Five entities are modeled by record types: Category, Borrower, Book, Fine, and Bookshelf.

2. Four relationships are modeled directly by a DBTG set: Belongs_To, Owed_By, Assessed_For, and Shelved_At.

3. One relationship is modeled by a record type plus two DBTG sets: Checked_Out plus CO_Borr, CO_Book.

4. The DBTG sets have the following properties:

a) Belongs_To: owner is Category, member is Borrower. Insertion is automatic, with mandatory retention. (Borrowers can move to a different category, but must be in one.)

b) Owed_By: owner is Borrower, member is Fine. Insertion is automatic, with fixed retention.

c) Assessed_For: owner is Fine, member is Checked_out. (This would allow a single Fine record to be created if several overdue books are returned simultaneously, representing the aggregate total of all amounts owed for the books. Insertion is manual, with fixed retention. (A check-out need not be the cause of a fine; but once it causes one, it stays attached to it.)

d) Shelved_At: owner is Bookshelf, member is Book. Insertion could be either automatic or manual, with mandatory retention. (Newly-arrived books need not have a space assigned, and books can be moved, but even a checked-out book has a place where it belongs.)

e) CO_Borr: owner is Borrower, member is Checked_Out. Insertion is automatic, with fixed retention.

f) CO_Book: owner is Book, member is Checked_Out. Insertion is automatic, with fixed retention.

G. The DBTG proposal includes both a DDL and a DML for working with network databases, both designed to be embedded in COBOL (though they have been adapted to other programming languages.) We concern ourselves just with the DML here. We will give only a subset of the options.

1. As was the case in the hierarchical model, there is a User Work Area (UWA) that contains

a) Record templates - one per record type in use

b) Record currency indicators - one per record type in use: defines a current record for that type

c) Set currency indicators - one per set: defines a current set occurrence for that set, plus either the owner or one of the members as the current record within the occurrence

d) Current of run unit (CRU) - the one record that has most recently been worked with (default if a record is not explicitly specified in a

DML command.)

e) Status variable - should be tested after every operation to see if if succeeded (unless one knows it cannot fail)

2. Commands to position the currency pointers on a particular record. Each finds a record according to some criteria and makes it the current record.

a) FIND FIRST record USING attribute(s)

(finds first record of the specified type that matches values in template for that type on specified attributes, and makes it CRU, current of record, and current of set for all sets it participates in)

b) FIND NEXT record USING attribute(s)

(finds first matching record after current of record for specified record type, and makes it CRU, current of record, and current of set for all sets it participates in)

c) FIND FIRST record WITHIN set
LAST
NEXT
PRIOR

(navigates within set occurrence of the specified type that is associated with the record that is CRU, and makes the record found CRU, current of its record type, and current of set for all sets it participates in)

d) FIND OWNER WITHIN set

(finds owner in occurrence of set associated with the record that is CRU, and makes it CRU and current of its record type as well as current of set for that set)

e) FIND record

(Makes current of record for specified record type CRU and current of set for all sets it participates in)

3. Commands to transfer data between the record which is CRU and the appropriate template in the UWA.

a) GET

b) MODIFY

c) ERASE

4. Command to create a new record (which becomes CRU, as well as current of record for its type and current of set for all sets it participates in)

   STORE record

5. Set manipulation commands - all affect the record which is CRU. CONNECT and RECONNECT establish a connection to the set occurrence which is current of set for the set specified.

   a) CONNECT TO set

   b) RECONNECT WITHIN set

   c) DISCONNECT FROM set

6. Sample queries (using a mixture of DBTG DML and C++-like pseudo code.)

   a) Is the book whose call number / copy number is QA76.093 / 1 checked out? (For simplicity, ignore possibility of keeping a checkout around after books is returned if there is a fine)

```
Book.call_no = “QA76.093”;
Book.copy_no = 1:
FIND FIRST Book USING call_no, copy_no;
FIND FIRST Checked_Out WITHIN CO_Book;
if (status is success)
   book is checked out
```

   b) List the names of all borrowers:

   (Omit GET when first writing on board, then ask students what’s missing)

```
FIND FIRST Borrower;
while (status is success)
{
  GET;
  write out Borrower.last_name, Borrower.first_name;
  FIND NEXT Borrower;
}
```

c) List the names of all borrowers having one or more books more than 7 days overdue. (Again, for simplicity, ignore possibility of keeping a checkout around after books is returned if there is a fine)

(Omit two GET's and FIND Borrower when writing on board, then ask students what's missing)

```
FIND FIRST Borrower;
while (status is success)
{
  boolean ok = true;
  FIND FIRST Checked_Out WITHIN CO_Borr;
  while (ok && status is success)
  {
      GET;
      if (Checked_Out.date_due is more than 7 days
                 before today)
          ok = false;
      else
          FIND NEXT Checked_Out WITHIN CO_Borr;
  }
  FIND Borrower;  // Note that a Checked_Out is
                  // typically CRU here - must make
                  // the Borrower be CRU
  if (! ok)
  {
      GET;
      write out Borrower.last_name,
                     Borrower.first_name;
  }
  FIND NEXT Borrower;
}
```

H. Repeating Groups in the DBTG Model

1. One feature of the DBTG model that sets it in contrast to the relational model is the use of REPEATING GROUPS within record types to solve certain kinds of problems. Consider the following: A given book typically has a single author. But some books have two authors; some have three, and a few have even more. How shall we model this?

   a) One approach would be to have multiple author attributes for each book record - say called Author1, Author2 ... - with the excess ones containing a null value.

      (1) This is a natural model.

      (2) But it is potentially highly space inefficient. For example, if some book has five authors, all book records need five author attributes. If most books have only one author, then most records contain four null slots.

      (3) Further, no matter how many slots you reserve within a book record (within reason!), there is always the possibility of a book coming along with one too many authors.

      (4) Also, it makes the task of finding books by a given author difficult, since several fields would have to be searched for the name.

      (5) This is not a good solution.

   b) Alternately, we could have a single "author" attribute in each book record, with a separate record for each author. Thus, our text book would be represented in the database by three records:

Database System Concepts ...	Korth, Henry F
Database System Concepts ...	Silberschatz, Abraham
Database System Concepts ...	Sudarshan, S

      (1) This raises the potential problems of data redundancy we noted earlier.

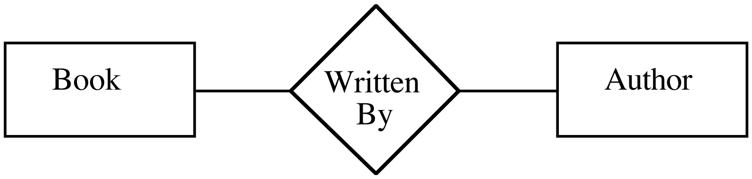
      (2) It also raises a particular problem for our library database: if a person checks out a certain book, to which of the records is the check-out record connected?

(a) Connecting it to all three is a nuisance, to say the least.

(b) But if we connect it to just one, then the database shows an entry for Database System Concepts that is not checked out to anyone - implying that it is available to be taken out, which it is not.

(3) This is not a good solution

c) We could redesign our model of book to separate out author, as follows:



(1) This requires creating a new record type for author, which is OK.

(2) But since the Written By relationship between Books and Authors is many to many, we also have to create a second link record type plus two sets to complete the model.

(3) This solution is acceptable, but costly. (It might become desirable, however, if we wanted to build a thorough author database with other information such as date of birth etc.)

d) Instead of taking one of these approaches, the DBTG model allows a record type to contain repeating groups. That is, in our example, the author attribute could be an ARRAY, with as many elements as necessary for the particular book. The record will also contain a count of the number of elements in the repeating group.

(1) This means that book records will be of varying size - somewhat of an implementation problem, but not insurmountable.

(2) The implementation would presumably require we declare some upper limit on the number of repetitions, but the limit could be made high since any one record contains only the slots it actually needs.

(3) In essence, this implements the first alternative we considered above, without the attendant disadvantages.

(4) Of course, in processing such a record we will now need to use a loop to process each member of the repeating group, in turn. (This is a programming necessity which cannot be avoided by any solution, of course.)

2. We might contrast this to the relational model's approach.

a) Relational database design rejects the notion of repeating groups. We will see that relational design is developed in terms of a hierarchy of normal forms (1NF, 2NF, 3NF, BCNF etc.) The basic requirement for 1NF (and thus for all the other normal forms) is that each record have a fixed format - i.e. no repeating groups.

b) Instead, the relational model could solve the problem by using two tables - one corresponding to the Book entity in the ER diagram we drew earlier, and one corresponding to the WrittenBy relationship. There would be no need for a separate Author table if we only stored the key attributes of an author (i.e. his/her name); if we wanted to store more information, we would add a third table with the name as the key and the other information.

c) This works nicely in the relational model, because the relational model has no difficulty with many-to-many relationships.