High-Level Database Models

Let us consider the process whereby a new database, such as our movie database, is created. Figure 1 suggests the process. We begin with a design phase, in which we address and answer questions about what information will be stored, how information elements will be related to one another, what constraints such as keys or referential integrity may be assumed, and so on. This phase may last for a long time, while options are evaluated and opinions are reconciled. We show this phase in Fig. 1 as the conversion of ideas to a high-level design.

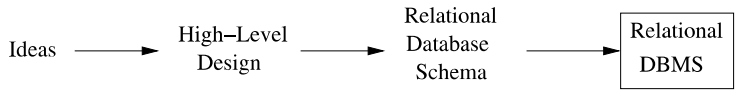

Figure 1: The database modeling and implementation process

Since the great majority of commercial database systems use the relational model, we might suppose that the design phase should use this model too. However, in practice it is often easier to start with a higher-level model and then convert the design to the relational model. The primary reason for doing so is that the relational model has only one concept — the relation — rather than several complementary concepts that more closely model real-world situations. Simplicity of concepts in the relational model is a great strength of the model, especially when it comes to efficient implementation of database operations. Yet that strength becomes a weakness when we do a preliminary design, which is why it often is helpful to begin by using a high-level design model.

There are several options for the notation in which the design is expressed. The first, and oldest, method is the "entity-relationship diagram," and here is where we shall start in Section 1. A more recent trend is the use of UML ("Unified Modeling Language"), a notation that was originally designed for

describing object-oriented software projects, but which has been adapted to describe database schemas as well. We shall see this model in Section 7. Finally, in Section 9, we shall consider ODL ("Object Description Language"), which was created to describe databases as collections of classes and their objects.

The next phase shown in Fig. 1 is the conversion of our high-level design to a relational design. This phase occurs only when we are confident of the high-level design. Whichever of the high-level models we use, there is a fairly mechanical way of converting the high-level design into a relational database schema, which then runs on a conventional DBMS. Sections 5 and 6 discuss conversion of E/R diagrams to relational database schemas. Section 8 does the same for UML, and Section 10 serves for ODL.

1 The Entity/Relationship Model

In the *entity-relationship model* (or *E/R model*). the structure of data is represented graphically, as an "entity-relationship diagram," using three principal element types:

1. Entity sets,
2. Attributes, and
3. Relationships.

We shall cover each in turn.

1.1 Entity Sets

An *entity* is an abstract object of some sort, and a collection of similar entities forms an *entity set*. An entity in some ways resembles an "object" in the sense of object-oriented programming. Likewise, an entity set bears some resemblance to a class of objects. However, the E/R model is a static concept, involving the structure of data and not the operations on data. Thus, one would not expect to find methods associated with an entity set as one would with a class.

Example 1: Let us consider the design of our running movie-database example. Each movie is an entity, and the set of all movies constitutes an entity set. Likewise, the stars are entities, and the set of stars is an entity set. A studio is another kind of entity, and the set of studios is a third entity set that will appear in our examples. □

1.2 Attributes

Entity sets have associated *attributes*, which are properties of the entities in that set. For instance, the entity set *Movies* might be given attributes such as *title* and *length*. It should not surprise you if the attributes for the entity

E/R Model Variations

In some versions of the E/R model, the type of an attribute can be either:

1. A primitive type, as in the version presented here.
2. A "struct," as in C, or tuple with a fixed number of primitive components.
3. A set of values of one type: either primitive or a "struct" type.

For example, the type of an attribute in such a model could be a set of pairs, each pair consisting of an integer and a string.

set *Movies* resemble the attributes of the relation `Movies` in our example. It is common for entity sets to be implemented as relations, although not every relation in our final relational design will come from an entity set.

In our version of the E/R model, we shall assume that attributes are of primitive types, such as strings, integers, or reals. There are other variations of this model in which attributes can have some limited structure; see the box on "E/R Model Variations."

1.3 Relationships

Relationships are connections among two or more entity sets. For instance, if *Movies* and *Stars* are two entity sets, we could have a relationship *Stars-in* that connects movies and stars. The intent is that a movie entity m is related to a star entity s by the relationship *Stars-in* if s appears in movie m. While binary relationships, those between two entity sets, are by far the most common type of relationship, the E/R model allows relationships to involve any number of entity sets. We shall defer discussion of these multiway relationships until Section 1.7.

1.4 Entity-Relationship Diagrams

An *E/R diagram* is a graph representing entity sets, attributes, and relationships. Elements of each of these kinds are represented by nodes of the graph, and we use a special shape of node to indicate the kind, as follows:

- Entity sets are represented by rectangles.
- Attributes are represented by ovals.
- Relationships are represented by diamonds.

Edges connect an entity set to its attributes and also connect a relationship to its entity sets.

Example 2: In Fig. 2 is an E/R diagram that represents a simple database about movies. The entity sets are *Movies*, *Stars*, and *Studios*.

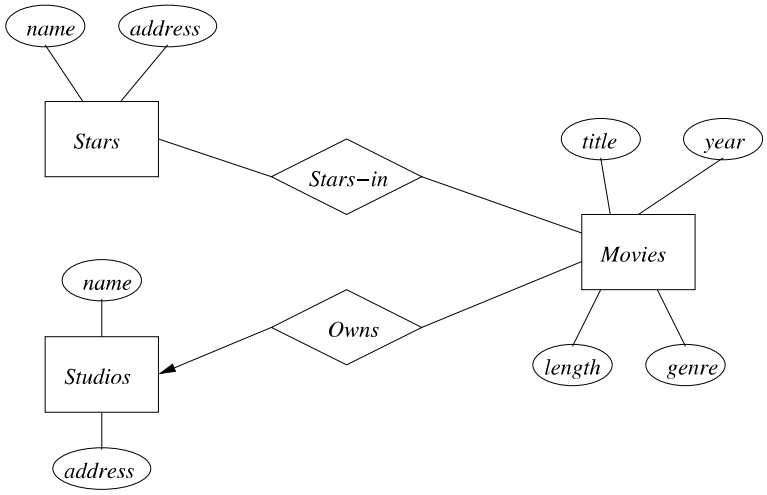

Figure 2: An entity-relationship diagram for the movie database

The *Movies* entity set has four of our usual attributes: *title*, *year*, *length*, and *genre*. The other two entity sets *Stars* and *Studios* happen to have the same two attributes: *name* and *address*, each with an obvious meaning. We also see two relationships in the diagram:

1. *Stars-in* is a relationship connecting each movie to the stars of that movie. This relationship consequently also connects stars to the movies in which they appeared.

2. *Owns* connects each movie to the studio that owns the movie. The arrow pointing to entity set *Studios* in Fig. 2 indicates that each movie is owned by at most one studio. We shall discuss uniqueness constraints such as this one in Section 1.6.

□

1.5 Instances of an E/R Diagram

E/R diagrams are a notation for describing schemas of databases. We may imagine that a database described by an E/R diagram contains particular data, an "instance" of the database. Since the database is not implemented in the E/R model, only designed, the instance never exists in the sense that a relation's

instances exist in a DBMS. However, it is often useful to visualize the database being designed as if it existed.

For each entity set, the database instance will have a particular finite set of entities. Each of these entities has particular values for each attribute. A relationship R that connects n entity sets $E_1, E_2, \ldots, E_n$ may be imagined to have an "instance" that consists of a finite set of tuples $(e_1, e_2, \ldots, e_n)$, where each e_i is chosen from the entities that are in the current instance of entity set E_i. We regard each of these tuples as "connected" by relationship R.

This set of tuples is called the *relationship set* for R. It is often helpful to visualize a relationship set as a table or relation. However, the "tuples" of a relationship set are not really tuples of a relation, since their components are entities rather than primitive types such as strings or integers. The columns of the table are headed by the names of the entity sets involved in the relationship, and each list of connected entities occupies one row of the table. As we shall see, however, when we convert relationships to relations, the resulting relation is not the same as the relationship set.

Example 3: An instance of the *Stars-in* relationship could be visualized as a table with pairs such as:

Movies	*Stars*
`Basic Instinct`	`Sharon Stone`
`Total Recall`	`Arnold Schwarzenegger`
`Total Recall`	`Sharon Stone`

The members of the relationship set are the rows of the table. For instance, (Basic Instinct, Sharon Stone) is a tuple in the relationship set for the current instance of relationship *Stars-in*. □

1.6 Multiplicity of Binary E/R Relationships

In general, a binary relationship can connect any member of one of its entity sets to any number of members of the other entity set. However, it is common for there to be a restriction on the "multiplicity" of a relationship. Suppose R is a relationship connecting entity sets E and F. Then:

- If each member of E can be connected by R to at most one member of F, then we say that R is *many-one* from E to F. Note that in a many-one relationship from E to F, each entity in F can be connected to many members of E. Similarly, if instead a member of F can be connected by R to at most one member of E, then we say R is many-one from F to E (or equivalently, one-many from E to F).

- If R is both many-one from E to F and many-one from F to E, then we say that R is *one-one*. In a one-one relationship an entity of either entity set can be connected to at most one entity of the other set.

- If R is neither many-one from E to F or from F to E, then we say R is *many-many*.

As we mentioned in Example 2, arrows can be used to indicate the multiplicity of a relationship in an E/R diagram. If a relationship is many-one from entity set E to entity set F, then we place an arrow entering F. The arrow indicates that each entity in set E is related to at most one entity in set F. Unless there is also an arrow on the edge to E, an entity in F may be related to many entities in E.

Example 4: A one-one relationship between entity sets E and F is represented by arrows pointing to both E and F. For instance, Fig. 3 shows two entity sets, *Studios* and *Presidents*, and the relationship *Runs* between them (attributes are omitted). We assume that a president can run only one studio and a studio has only one president, so this relationship is one-one, as indicated by the two arrows, one entering each entity set.

Figure 3: A one-one relationship

Remember that the arrow means "at most one"; it does not guarantee existence of an entity of the set pointed to. Thus, in Fig. 3, we would expect that a "president" is surely associated with some studio; how could they be a "president" otherwise? However, a studio might not have a president at some particular time, so the arrow from *Runs* to *Presidents* truly means "at most one" and not "exactly one." We shall discuss the distinction further in Section 3.3. □

1.7 Multiway Relationships

The E/R model makes it convenient to define relationships involving more than two entity sets. In practice, ternary (three-way) or higher-degree relationships are rare, but they occasionally are necessary to reflect the true state of affairs. A multiway relationship in an E/R diagram is represented by lines from the relationship diamond to each of the involved entity sets.

Example 5: In Fig. 4 is a relationship *Contracts* that involves a studio, a star, and a movie. This relationship represents that a studio has contracted with a particular star to act in a particular movie. In general, the value of an E/R relationship can be thought of as a relationship set of tuples whose components are the entities participating in the relationship, as we discussed in Section 1.5. Thus, relationship *Contracts* can be described by triples of the form (studio, star, movie).

Implications Among Relationship Types

We should be aware that a many-one relationship is a special case of a many-many relationship, and a one-one relationship is a special case of a many-one relationship. Thus, any useful property of many-one relationships holds for one-one relationships too. For example, a data structure for representing many-one relationships will work for one-one relationships, although it might not be suitable for many-many relationships.

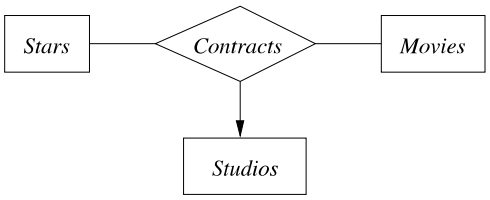

Figure 4: A three-way relationship

In multiway relationships, an arrow pointing to an entity set E means that if we select one entity from each of the other entity sets in the relationship, those entities are related to at most one entity in E. (Note that this rule generalizes the notation used for many-one, binary relationships.) Informally, we may think of a functional dependency with E on the right and all the other entity sets of the relationship on the left.

In Fig. 4 we have an arrow pointing to entity set *Studios*, indicating that for a particular star and movie, there is only one studio with which the star has contracted for that movie. However, there are no arrows pointing to entity sets *Stars* or *Movies*. A studio may contract with several stars for a movie, and a star may contract with one studio for more than one movie. □

1.8 Roles in Relationships

It is possible that one entity set appears two or more times in a single relationship. If so, we draw as many lines from the relationship to the entity set as the entity set appears in the relationship. Each line to the entity set represents a different *role* that the entity set plays in the relationship. We therefore label the edges between the entity set and relationship by names, which we call "roles."

Example 6: In Fig. 5 is a relationship *Sequel-of* between the entity set *Movies* and itself. Each relationship is between two movies, one of which is the sequel of the other. To differentiate the two movies in a relationship, one line is labeled by the role *Original* and one by the role *Sequel*, indicating the

Limits on Arrow Notation in Multiway Relationships

There are not enough choices of arrow or no-arrow on the lines attached to a relationship with three or more participants. Thus, we cannot describe every possible situation with arrows. For instance, in Fig. 4, the studio is really a function of the movie alone, not the star and movie jointly, since only one studio produces a movie. However, our notation does not distinguish this situation from the case of a three-way relationship where the entity set pointed to by the arrow is truly a function of both other entity sets. To handle all possible situations, we would have to give a set of functional dependencies involving the entity sets of the relationship.

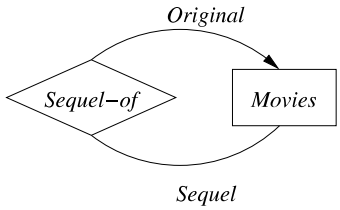

Figure 5: A relationship with roles

original movie and its sequel, respectively. We assume that a movie may have many sequels, but for each sequel there is only one original movie. Thus, the relationship is many-one from *Sequel* movies to *Original* movies, as indicated by the arrow in the E/R diagram of Fig. 5. □

Example 7: As a final example that includes both a multiway relationship and an entity set with multiple roles, in Fig. 6 is a more complex version of the *Contracts* relationship introduced earlier in Example 5. Now, relationship *Contracts* involves two studios, a star, and a movie. The intent is that one studio, having a certain star under contract (in general, not for a particular movie), may further contract with a second studio to allow that star to act in a particular movie. Thus, the relationship is described by 4-tuples of the form (studio1, studio2, star, movie), meaning that studio2 contracts with studio1 for the use of studio1's star by studio2 for the movie.

We see in Fig. 6 arrows pointing to *Studios* in both of its roles, as "owner" of the star and as producer of the movie. However, there are not arrows pointing to *Stars* or *Movies*. The rationale is as follows. Given a star, a movie, and a studio producing the movie, there can be only one studio that "owns" the star. (We assume a star is under contract to exactly one studio.) Similarly, only one studio produces a given movie, so given a star, a movie, and the star's studio, we can determine a unique producing studio. Note that in both

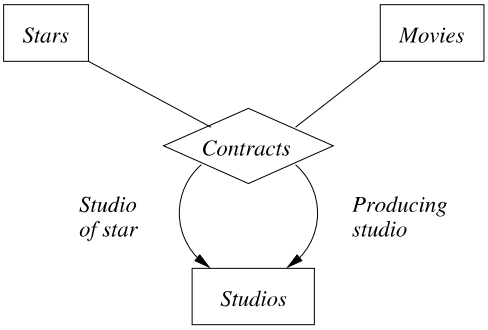

Figure 6: A four-way relationship

cases we actually needed only one of the other entities to determine the unique entity—for example, we need only know the movie to determine the unique producing studio—but this fact does not change the multiplicity specification for the multiway relationship.

There are no arrows pointing to *Stars* or *Movies*. Given a star, the star's studio, and a producing studio, there could be several different contracts allowing the star to act in several movies. Thus, the other three components in a relationship 4-tuple do not necessarily determine a unique movie. Similarly, a producing studio might contract with some other studio to use more than one of their stars in one movie. Thus, a star is not determined by the three other components of the relationship. □

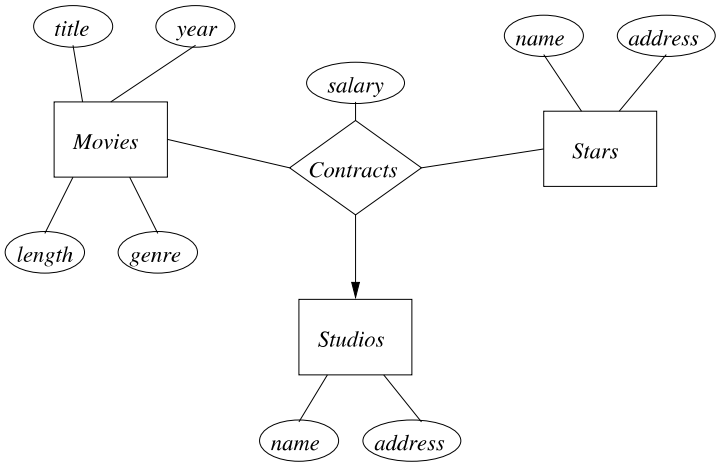

Figure 7: A relationship with an attribute

1.9 Attributes on Relationships

Sometimes it is convenient, or even essential, to associate attributes with a relationship, rather than with any one of the entity sets that the relationship connects. For example, consider the relationship of Fig. 4, which represents contracts between a star and studio for a movie.[1] We might wish to record the salary associated with this contract. However, we cannot associate it with the star; a star might get different salaries for different movies. Similarly, it does not make sense to associate the salary with a studio (they may pay different salaries to different stars) or with a movie (different stars in a movie may receive different salaries).

However, we can associate a unique salary with the (star, movie, studio) triple in the relationship set for the *Contracts* relationship. In Fig. 7 we see Fig. 4 fleshed out with attributes. The relationship has attribute *salary*, while the entity sets have the same attributes that we showed for them in Fig. 2.

In general, we may place one or more attributes on any relationship. The values of these attributes are functionally determined by the entire tuple in the relationship set for that relation. In some cases, the attributes can be determined by a subset of the entity sets involved in the relation, but presumably not by any single entity set (or it would make more sense to place the attribute on that entity set). For instance, in Fig. 7, the salary is really determined by the movie and star entities, since the studio entity is itself determined by the movie entity.

It is never necessary to place attributes on relationships. We can instead invent a new entity set, whose entities have the attributes ascribed to the relationship. If we then include this entity set in the relationship, we can omit the attributes on the relationship itself. However, attributes on a relationship are a useful convention, which we shall continue to use where appropriate.

Example 8: Let us revise the E/R diagram of Fig. 7, which has the salary attribute on the *Contracts* relationship. Instead, we create an entity set *Salaries*, with attribute *salary*. *Salaries* becomes the fourth entity set of relationship *Contracts*. The whole diagram is shown in Fig. 8.

Notice that there is an arrow into the *Salaries* entity set in Fig. 8. That arrow is appropriate, since we know that the salary is determined by all the other entity sets involved in the relationship. In general, when we do a conversion from attributes on a relationship to an additional entity set, we place an arrow into that entity set. □

1.10 Converting Multiway Relationships to Binary

There are some data models, such as UML (Section 7) and ODL (Section 9), that limit relationships to be binary. Thus, while the E/R model does not

[1] Here, we have reverted to the earlier notion of three-way contracts in Example 5, not the four-way relationship of Example 7.

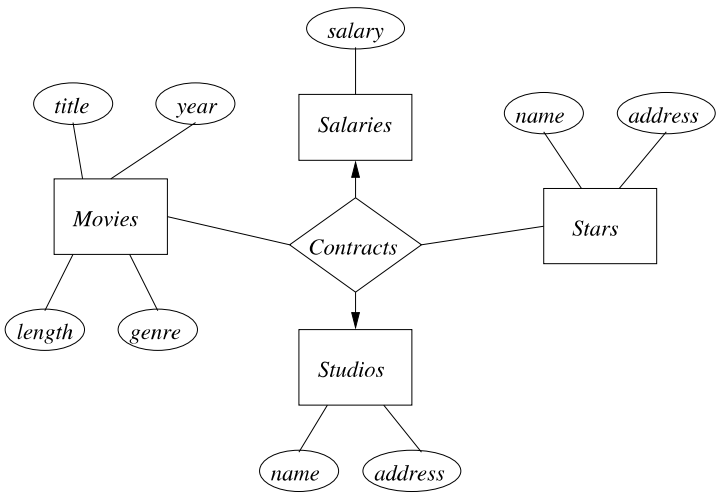

Figure 8: Moving the attribute to an entity set

require binary relationships, it is useful to observe that any relationship connecting more than two entity sets can be converted to a collection of binary, many-one relationships. To do so, introduce a new entity set whose entities we may think of as tuples of the relationship set for the multiway relationship. We call this entity set a *connecting* entity set. We then introduce many-one relationships from the connecting entity set to each of the entity sets that provide components of tuples in the original, multiway relationship. If an entity set plays more than one role, then it is the target of one relationship for each role.

Example 9: The four-way *Contracts* relationship in Fig. 6 can be replaced by an entity set that we may also call *Contracts*. As seen in Fig. 9, it participates in four relationships. If the relationship set for the relationship *Contracts* has a 4-tuple (studio1, studio2, star, movie) then the entity set *Contracts* has an entity e. This entity is linked by relationship *Star-of* to the entity *star* in entity set *Stars*. It is linked by relationship *Movie-of* to the entity *movie* in *Movies*. It is linked to entities *studio1* and *studio2* of *Studios* by relationships *Studio-of-star* and *Producing-studio*, respectively.

Note that we have assumed there are no attributes of entity set *Contracts*, although the other entity sets in Fig. 9 have unseen attributes. However, it is possible to add attributes, such as the date of signing, to entity set *Contracts*. □

1.11 Subclasses in the E/R Model

Often, an entity set contains certain entities that have special properties not associated with all members of the set. If so, we find it useful to define certain

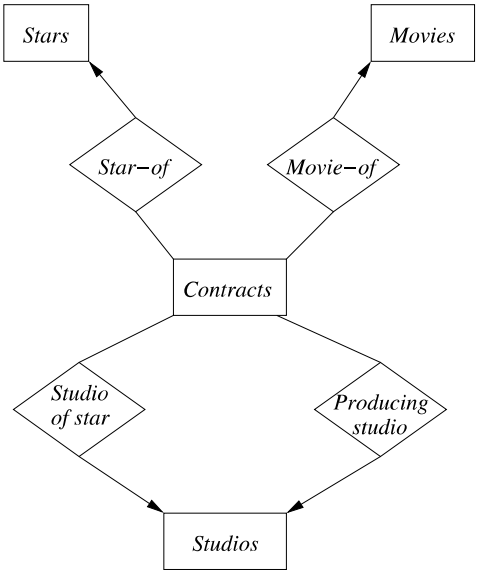

Figure 9: Replacing a multiway relationship by an entity set and binary relationships

special-case entity sets, or *subclasses*, each with its own special attributes and/or relationships. We connect an entity set to its subclasses using a relationship called *isa* (i.e., "an A is a B" expresses an "isa" relationship from entity set A to entity set B).

An isa relationship is a special kind of relationship, and to emphasize that it is unlike other relationships, we use a special notation: a triangle. One side of the triangle is attached to the subclass, and the opposite point is connected to the superclass. Every isa relationship is one-one, although we shall not draw the two arrows that are associated with other one-one relationships.

Example 10: Among the special kinds of movies we might store in our example database are cartoons and murder mysteries. For each of these special movie types, we could define a subclass of the entity set *Movies*. For instance, let us postulate two subclasses: *Cartoons* and *Murder-Mysteries*. A cartoon has, in addition to the attributes and relationships of *Movies*, an additional relationship called *Voices* that gives us a set of stars who speak, but do not appear in the movie. Movies that are not cartoons do not have such stars. Murder-mysteries have an additional attribute *weapon*. The connections among the three entity sets *Movies*, *Cartoons*, and *Murder-Mysteries* is shown in Fig. 10. □

While, in principle, a collection of entity sets connected by *isa* relationships could have any structure, we shall limit isa-structures to trees, in which there

Parallel Relationships Can Be Different

Figure 9 illustrates a subtle point about relationships. There are two different relationships, *Studio-of-Star* and *Producing-Studio*, that each connect entity sets *Contracts* and *Studios*. We should not presume that these relationships therefore have the same relationship sets. In fact, in this case, it is unlikely that both relationships would ever relate the same contract to the same studios, since a studio would then be contracting with itself.

More generally, there is nothing wrong with an E/R diagram having several relationships that connect the same entity sets. In the database, the instances of these relationships will normally be different, reflecting the different meanings of the relationships.

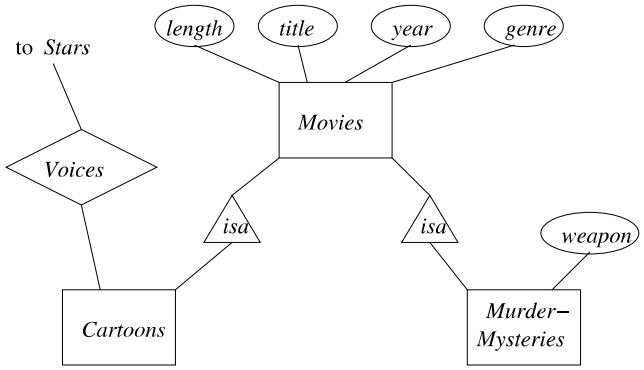

Figure 10: Isa relationships in an E/R diagram

is one *root* entity set (e.g., *Movies* in Fig. 10) that is the most general, with progressively more specialized entity sets extending below the root in a tree.

Suppose we have a tree of entity sets, connected by *isa* relationships. A single entity consists of *components* from one or more of these entity sets, as long as those components are in a subtree including the root. That is, if an entity e has a component c in entity set E, and the parent of E in the tree is F, then entity e also has a component d in F. Further, c and d must be paired in the relationship set for the *isa* relationship from E to F. The entity e has whatever attributes any of its components has, and it participates in whatever relationships any of its components participate in.

Example 11: The typical movie, being neither a cartoon nor a murder-mystery, will have a component only in the root entity set *Movies* in Fig. 10. These entities have only the four attributes of *Movies* (and the two relationships of *Movies* — *Stars-in* and *Owns* — that are not shown in Fig. 10).

The E/R View of Subclasses

There is a significant resemblance between "isa" in the E/R model and subclasses in object-oriented languages. In a sense, "isa" relates a subclass to its superclass. However, there is also a fundamental difference between the conventional E/R view and the object-oriented approach: entities are allowed to have representatives in a tree of entity sets, while objects are assumed to exist in exactly one class or subclass.

The difference becomes apparent when we consider how the movie *Roger Rabbit* was handled in Example 11. In an object-oriented approach, we would need for this movie a fourth entity set, "cartoon-murder-mystery," which inherited all the attributes and relationships of *Movies*, *Cartoons*, and *Murder-Mysteries*. However, in the E/R model, the effect of this fourth subclass is obtained by putting components of the movie *Roger Rabbit* in both the *Cartoons* and *Murder-Mysteries* entity sets.

A cartoon that is not a murder-mystery will have two components, one in *Movies* and one in *Cartoons*. Its entity will therefore have not only the four attributes of *Movies*, but the relationship *Voices*. Likewise, a murder-mystery will have two components for its entity, one in *Movies* and one in *Murder-Mysteries* and thus will have five attributes, including *weapon*.

Finally, a movie like *Roger Rabbit*, which is both a cartoon and a murder-mystery, will have components in all three of the entity sets *Movies*, *Cartoons*, and *Murder-Mysteries*. The three components are connected into one entity by the *isa* relationships. Together, these components give the *Roger Rabbit* entity all four attributes of *Movies* plus the attribute *weapon* of entity set *Murder-Mysteries* and the relationship *Voices* of entity set *Cartoons*. □

1.12 Exercises for Section 1

Exercise 1.1: Design a database for a bank, including information about customers and their accounts. Information about a customer includes their name, address, phone, and Social Security number. Accounts have numbers, types (e.g., savings, checking) and balances. Also record the customer(s) who own an account. Draw the E/R diagram for this database. Be sure to include arrows where appropriate, to indicate the multiplicity of a relationship.

Exercise 1.2: Modify your solution to Exercise 1.1 as follows:

a) Change your diagram so an account can have only one customer.

b) Further change your diagram so a customer can have only one account.

! c) Change your original diagram of Exercise 1.1 so that a customer can have a set of addresses (which are street-city-state triples) and a set of phones. Remember that we do not allow attributes to have nonprimitive types, such as sets, in the E/R model.

! d) Further modify your diagram so that customers can have a set of addresses, and at each address there is a set of phones.

Exercise 1.3: Give an E/R diagram for a database recording information about teams, players, and their fans, including:

1. For each team, its name, its players, its team captain (one of its players), and the colors of its uniform.
2. For each player, his/her name.
3. For each fan, his/her name, favorite teams, favorite players, and favorite color.

Remember that a set of colors is not a suitable attribute type for teams. How can you get around this restriction?

Exercise 1.4: Suppose we wish to add to the schema of Exercise 1.3 a relationship *Led-by* among two players and a team. The intention is that this relationship set consists of triples (player1, player2, team) such that player 1 played on the team at a time when some other player 2 was the team captain.

a) Draw the modification to the E/R diagram.

b) Replace your ternary relationship with a new entity set and binary relationships.

! c) Are your new binary relationships the same as any of the previously existing relationships? Note that we assume the two players are different, i.e., the team captain is not self-led.

Exercise 1.5: Modify Exercise 1.3 to record for each player the history of teams on which they have played, including the start date and ending date (if they were traded) for each such team.

! Exercise 1.6: Design a genealogy database with one entity set: *People*. The information to record about persons includes their name (an attribute), their mother, father, and children.

! Exercise 1.7: Modify your "people" database design of Exercise 1.6 to include the following special types of people:

1. Females.

2. Males.

3. People who are parents.

You may wish to distinguish certain other kinds of people as well, so relationships connect appropriate subclasses of people.

Exercise 1.8: An alternative way to represent the information of Exercise 1.6 is to have a ternary relationship *Family* with the intent that in the relationship set for *Family*, triple (person, mother, father) is a person, their mother, and their father; all three are in the *People* entity set, of course.

a) Draw this diagram, placing arrows on edges where appropriate.

b) Replace the ternary relationship *Family* by an entity set and binary relationships. Again place arrows to indicate the multiplicity of relationships.

Exercise 1.9: Design a database suitable for a university registrar. This database should include information about students, departments, professors, courses, which students are enrolled in which courses, which professors are teaching which courses, student grades, TA's for a course (TA's are students), which courses a department offers, and any other information you deem appropriate. Note that this question is more free-form than the questions above, and you need to make some decisions about multiplicities of relationships, appropriate types, and even what information needs to be represented.

! Exercise 1.10: Informally, we can say that two E/R diagrams "have the same information" if, given a real-world situation, the instances of these two diagrams that reflect this situation can be computed from one another. Consider the E/R diagram of Fig. 6. This four-way relationship can be decomposed into a three-way relationship and a binary relationship by taking advantage of the fact that for each movie, there is a unique studio that produces that movie. Give an E/R diagram without a four-way relationship that has the same information as Fig. 6.

2 Design Principles

We have yet to learn many of the details of the E/R model, but we have enough to begin study of the crucial issue of what constitutes a good design and what should be avoided. In this section, we offer some useful design principles.

2.1 Faithfulness

First and foremost, the design should be faithful to the specifications of the application. That is, entity sets and their attributes should reflect reality. You can't attach an attribute *number-of-cylinders* to *Stars*, although that attribute

would make sense for an entity set *Automobiles*. Whatever relationships are asserted should make sense given what we know about the part of the real world being modeled.

Example 12: If we define a relationship *Stars-in* between *Stars* and *Movies*, it should be a many-many relationship. The reason is that an observation of the real world tells us that stars can appear in more than one movie, and movies can have more than one star. It is incorrect to declare the relationship *Stars-in* to be many-one in either direction or to be one-one. □

Example 13: On the other hand, sometimes it is less obvious what the real world requires us to do in our E/R design. Consider, for instance, entity sets *Courses* and *Instructors*, with a relationship *Teaches* between them. Is *Teaches* many-one from *Courses* to *Instructors*? The answer lies in the policy and intentions of the organization creating the database. It is possible that the school has a policy that there can be only one instructor for any course. Even if several instructors may "team-teach" a course, the school may require that exactly one of them be listed in the database as the instructor responsible for the course. In either of these cases, we would make *Teaches* a many-one relationship from *Courses* to *Instructors*.

Alternatively, the school may use teams of instructors regularly and wish its database to allow several instructors to be associated with a course. Or, the intent of the *Teaches* relationship may not be to reflect the current teacher of a course, but rather those who have ever taught the course, or those who are capable of teaching the course; we cannot tell simply from the name of the relationship. In either of these cases, it would be proper to make *Teaches* be many-many. □

2.2 Avoiding Redundancy

We should be careful to say everything once only. Problems regarding redundancy and anomalies are typical of problems that can arise in E/R designs. However, in the E/R model, there are several new mechanisms whereby redundancy and other anomalies can arise.

For instance, we have used a relationship *Owns* between movies and studios. We might also choose to have an attribute *studioName* of entity set *Movies*. While there is nothing illegal about doing so, it is dangerous for several reasons.

1. Doing so leads to repetition of a fact, with the result that extra space is required to represent the data, once we convert the E/R design to a relational (or other type of) concrete implementation.

2. There is an update-anomaly potential, since we might change the relationship but not the attribute, or vice-versa.

We shall say more about avoiding anomalies in Sections 2.4 and 2.5.

2.3 Simplicity Counts

Avoid introducing more elements into your design than is absolutely necessary.

Example 14: Suppose that instead of a relationship between *Movies* and *Studios* we postulated the existence of "movie-holdings," the ownership of a single movie. We might then create another entity set *Holdings*. A one-one relationship *Represents* could be established between each movie and the unique holding that represents the movie. A many-one relationship from *Holdings* to *Studios* completes the picture shown in Fig. 11.

Figure 11: A poor design with an unnecessary entity set

Technically, the structure of Fig. 11 truly represents the real world, since it is possible to go from a movie to its unique owning studio via *Holdings*. However, *Holdings* serves no useful purpose, and we are better off without it. It makes programs that use the movie-studio relationship more complicated, wastes space, and encourages errors. □

2.4 Choosing the Right Relationships

Entity sets can be connected in various ways by relationships. However, adding to our design every possible relationship is not often a good idea. Doing so can lead to redundancy, update anomalies, and deletion anomalies, where the connected pairs or sets of entities for one relationship can be deduced from one or more other relationships. We shall illustrate the problem and what to do about it with two examples. In the first example, several relationships could represent the same information; in the second, one relationship could be deduced from several others.

Example 15: Let us review Fig. 7, where we connected movies, stars, and studios with a three-way relationship *Contracts*. We omitted from that figure the two binary relationships *Stars-in* and *Owns* from Fig. 2. Do we also need these relationships, between *Movies* and *Stars*, and between *Movies* and *Studios*, respectively? The answer is: "we don't know; it depends on our assumptions regarding the three relationships in question."

It might be possible to deduce the relationship *Stars-in* from *Contracts*. If a star can appear in a movie only if there is a contract involving that star, that movie, and the owning studio for the movie, then there truly is no need for relationship *Stars-in*. We could figure out all the star-movie pairs by looking at the star-movie-studio triples in the relationship set for *Contracts* and taking only the star and movie components, i.e., projecting *Contracts* onto *Stars-in*.

However, if a star can work on a movie without there being a contract — or what is more likely, without there being a contract that we know about in our database — then there could be star-movie pairs in *Stars-in* that are not part of star-movie-studio triples in *Contracts*. In that case, we need to retain the *Stars-in* relationship.

A similar observation applies to relationship *Owns*. If for every movie, there is at least one contract involving that movie, its owning studio, and some star for that movie, then we can dispense with *Owns*. However, if there is the possibility that a studio owns a movie, yet has no stars under contract for that movie, or no such contract is known to our database, then we must retain *Owns*.

In summary, we cannot tell you whether a given relationship will be redundant. You must find out from those who wish the database implemented what to expect. Only then can you make a rational decision about whether or not to include relationships such as *Stars-in* or *Owns*. □

Example 16: Now, consider Fig. 2 again. In this diagram, there is no relationship between stars and studios. Yet we can use the two relationships *Stars-in* and *Owns* to build a connection by the process of composing those two relationships. That is, a star is connected to some movies by *Stars-in*, and those movies are connected to studios by *Owns*. Thus, we could say that a star is connected to the studios that own movies in which the star has appeared.

Would it make sense to have a relationship *Works-for*, as suggested in Fig. 12, between *Stars* and *Studios* too? Again, we cannot tell without knowing more. First, what would the meaning of this relationship be? If it is to mean "the star appeared in at least one movie of this studio," then probably there is no good reason to include it in the diagram. We could deduce this information from *Stars-in* and *Owns* instead.

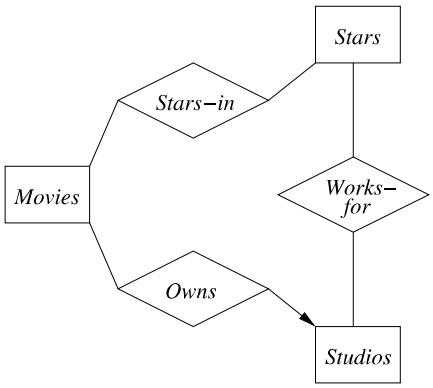

Figure 12: Adding a relationship between *Stars* and *Studios*

However, perhaps we have other information about stars working for studios that is not implied by the connection through a movie. In that case, a

relationship connecting stars directly to studios might be useful and would not be redundant. Alternatively, we might use a relationship between stars and studios to mean something entirely different. For example, it might represent the fact that the star is under contract to the studio, in a manner unrelated to any movie. As we suggested in Example 7, it is possible for a star to be under contract to one studio and yet work on a movie owned by another studio. In this case, the information found in the new *Works-for* relation would be independent of the *Stars-in* and *Owns* relationships, and would surely be nonredundant. □

2.5 Picking the Right Kind of Element

Sometimes we have options regarding the type of design element used to represent a real-world concept. Many of these choices are between using attributes and using entity set/relationship combinations. In general, an attribute is simpler to implement than either an entity set or a relationship. However, making everything an attribute will usually get us into trouble.

Example 17: Let us consider a specific problem. In Fig. 2, were we wise to make studios an entity set? Should we instead have made the name and address of the studio be attributes of movies and eliminated the *Studio* entity set? One problem with doing so is that we repeat the address of the studio for each movie. We can also have an update anomaly if we change the address for one movie but not another with the same studio, and we can have a deletion anomaly if we delete the last movie owned by a given studio.

On the other hand, if we did not record addresses of studios, then there is no harm in making the studio name an attribute of movies. We have no anomalies in this case. Saying the name of a studio for each movie is not true redundancy, since we must represent the owner of each movie somehow, and saying the name of the studio is a reasonable way to do so. □

We can abstract what we have observed in Example 17 to give the conditions under which we prefer to use an attribute instead of an entity set. Suppose E is an entity set. Here are conditions that E must obey in order for us to replace E by an attribute or attributes of several other entity sets.

1. All relationships in which E is involved must have arrows entering E. That is, E must be the "one" in many-one relationships, or its generalization for the case of multiway relationships.

2. If E has more than one attribute, then no attribute depends on the other attributes, the way *address* depends on *name* for *Studios*. That is, the only key for E is all its attributes.

3. No relationship involves E more than once.

If these conditions are met, then we can replace entity set E as follows:

a) If there is a many-one relationship R from some entity set F to E, then remove R and make the attributes of E be attributes of F, suitably renamed if they conflict with attribute names for F. In effect, each F-entity takes, as attributes, the name of the unique, related E-entity.[2] For instance, *Movies* entities could take their studio name as an attribute, should we dispense with studio addresses.

b) If there is a multiway relationship R with an arrow to E, make the attributes of E be attributes of R and delete the arc from R to E. An example of this transformation is replacing Fig. 8, where there is an entity set *Salaries* with a number as its lone attribute, by its original diagram in Fig. 7.

Example 18: Let us consider a point where there is a tradeoff between using a multiway relationship and using a connecting entity set with several binary relationships. We saw a four-way relationship *Contracts* among a star, a movie, and two studios in Fig. 6. In Fig. 9, we mechanically converted it to an entity set *Contracts*. Does it matter which we choose?

As the problem was stated, either is appropriate. However, should we change the problem just slightly, then we are almost forced to choose a connecting entity set. Let us suppose that contracts involve one star, one movie, but any set of studios. This situation is more complex than the one in Fig. 6, where we had two studios playing two roles. In this case, we can have any number of studios involved, perhaps one to do production, one for special effects, one for distribution, and so on. Thus, we cannot assign roles for studios.

It appears that a relationship set for the relationship *Contracts* must contain triples of the form (star, movie, set-of-studios), and the relationship *Contracts* itself involves not only the usual *Stars* and *Movies* entity sets, but a new entity set whose entities are *sets of* studios. While this approach is possible, it seems unnatural to think of sets of studios as basic entities, and we do not recommend it.

A better approach is to think of contracts as an entity set. As in Fig. 9, a contract entity connects a star, a movie and a set of studios, but now there must be no limit on the number of studios. Thus, the relationship between contracts and studios is many-many, rather than many-one as it would be if contracts were a true "connecting" entity set. Figure 13 sketches the E/R diagram. Note that a contract is related to a single star and to a single movie, but to any number of studios. □

2.6 Exercises for Section 2

Exercise 2.1: In Fig. 14 is an E/R diagram for a bank database involving customers and accounts. Since customers may have several accounts, and

[2] In a situation where an F-entity is not related to any E-entity, the new attributes of F would be given special "null" values to indicate the absence of a related E-entity. A similar arrangement would be used for the new attributes of R in case (b).

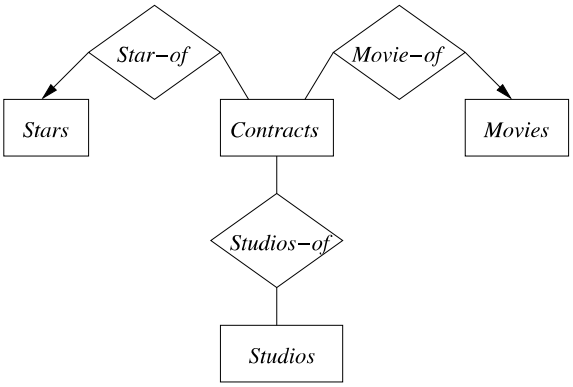

Figure 13: Contracts connecting a star, a movie, and a set of studios

accounts may be held jointly by several customers, we associate with each customer an "account set," and accounts are members of one or more account sets. Assuming the meaning of the various relationships and attributes are as expected given their names, criticize the design. What design rules are violated? Why? What modifications would you suggest?

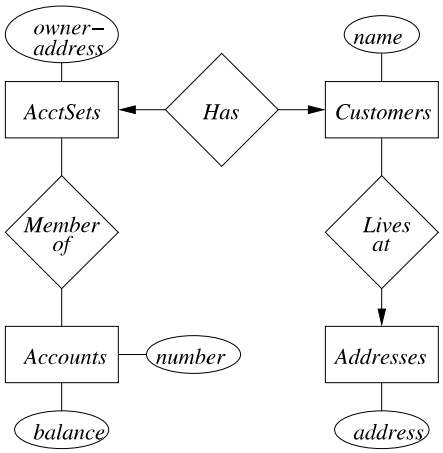

Figure 14: A poor design for a bank database

Exercise 2.2: Under what circumstances (regarding the unseen attributes of *Studios* and *Presidents*) would you recommend combining the two entity sets and relationship in Fig. 3 into a single entity set and attributes?

Exercise 2.3: Suppose we delete the attribute *address* from *Studios* in Fig. 7. Show how we could then replace an entity set by an attribute. Where would that attribute appear?

Exercise 2.4: Give choices of attributes for the following entity sets in Fig. 13 that will allow the entity set to be replaced by an attribute:

a) *Stars.*

b) *Movies.*

! c) *Studios.*

!! **Exercise 2.5:** In this and following exercises we shall consider two design options in the E/R model for describing births. At a birth, there is one baby (twins would be represented by two births), one mother, any number of nurses, and any number of doctors. Suppose, therefore, that we have entity sets *Babies*, *Mothers*, *Nurses*, and *Doctors*. Suppose we also use a relationship *Births*, which connects these four entity sets, as suggested in Fig. 15. Note that a tuple of the relationship set for *Births* has the form (baby, mother, nurse, doctor). If there is more than one nurse and/or doctor attending a birth, then there will be several tuples with the same baby and mother, one for each combination of nurse and doctor.

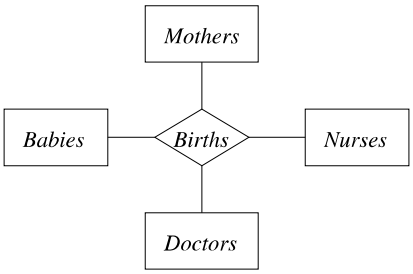

Figure 15: Representing births by a multiway relationship

There are certain assumptions that we might wish to incorporate into our design. For each, tell how to add arrows or other elements to the E/R diagram in order to express the assumption.

a) For every baby, there is a unique mother.

b) For every combination of a baby, nurse, and doctor, there is a unique mother.

c) For every combination of a baby and a mother there is a unique doctor.

! **Exercise 2.6:** Another approach to the problem of Exercise 2.5 is to connect the four entity sets *Babies*, *Mothers*, *Nurses*, and *Doctors* by an entity set

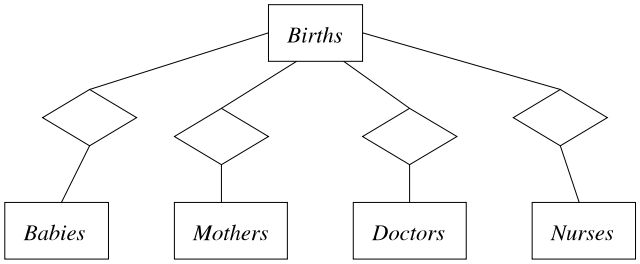

Figure 16: Representing births by an entity set

Births, with four relationships, one between *Births* and each of the other entity sets, as suggested in Fig. 16. Use arrows (indicating that certain of these relationships are many-one) to represent the following conditions:

a) Every baby is the result of a unique birth, and every birth is of a unique baby.

b) In addition to (a), every baby has a unique mother.

c) In addition to (a) and (b), for every birth there is a unique doctor.

In each case, what design flaws do you see?

!! Exercise 2.7 : Suppose we change our viewpoint to allow a birth to involve more than one baby born to one mother. How would you represent the fact that every baby still has a unique mother using the approaches of Exercises 2.5 and 2.6?

3 Constraints in the E/R Model

The E/R model has several ways to express the common kinds of constraints on the data that will populate the database being designed. Like the relational model, there is a way to express the idea that an attribute or attributes are a key for an entity set. We have already seen how an arrow connecting a relationship to an entity set serves as a "functional dependency." There is also a way to express a referential-integrity constraint, where an entity in one set is required to have an entity in another set to which it is related.

3.1 Keys in the E/R Model

A *key* for an entity set E is a set K of one or more attributes such that, given any two distinct entities e_1 and e_2 in E, e_1 and e_2 cannot have identical values for each of the attributes in the key K. If K consists of more than one attribute, then it is possible for e_1 and e_2 to agree in some of these attributes, but never in all attributes. Some important points to remember are:

- Every entity set must have a key, although in some cases — isa-hierarchies and "weak" entity sets (see Section 4), the key actually belongs to another entity set.

- There can be more than one possible key for an entity set. However, it is customary to pick one key as the "primary key," and to act as if that were the only key.

- When an entity set is involved in an isa-hierarchy, we require that the root entity set have all the attributes needed for a key, and that the key for each entity is found from its component in the root entity set, regardless of how many entity sets in the hierarchy have components for the entity.

In our running movies example, we have used *title* and *year* as the key for *Movies*, counting on the observation that it is unlikely that two movies with the same title would be released in one year. We also decided that it was safe to use *name* as a key for *MovieStar*, believing that no real star would ever want to use the name of another star.

3.2 Representing Keys in the E/R Model

In our E/R-diagram notation, we underline the attributes belonging to a key for an entity set. For example, Fig. 17 reproduces our E/R diagram for movies, stars, and studios from Fig. 2, but with key attributes underlined. Attribute *name* is the key for *Stars*. Likewise, *Studios* has a key consisting of only its own attribute *name*.

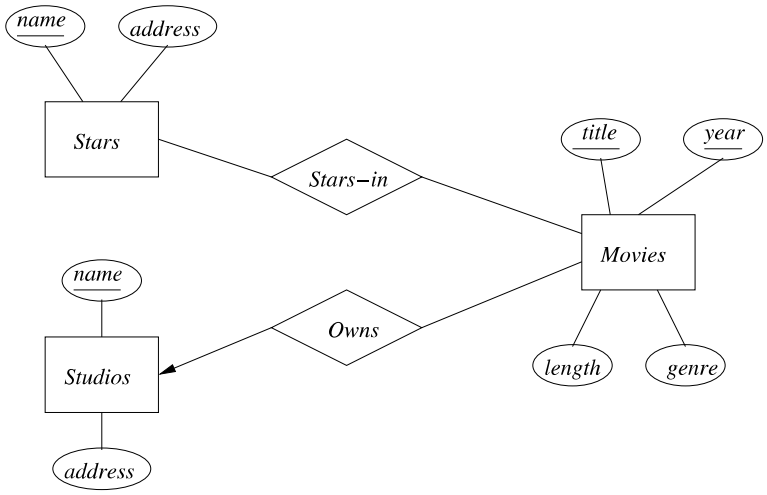

Figure 17: E/R diagram; keys are indicated by underlines

The attributes *title* and *year* together form the key for *Movies*. Note that when several attributes are underlined, as in Fig. 17, then they are each members of the key. There is no notation for representing the situation where there are several keys for an entity set; we underline only the primary key. You should also be aware that in some unusual situations, the attributes forming the key for an entity set do not all belong to the entity set itself. We shall defer this matter, called "weak entity sets," until Section 4.

3.3 Referential Integrity

Recall the properties of referential-integrity constraints. These constraints say that a value appearing in one context must also appear in another. For example, let us consider the many-one relationship *Owns* from *Movies* to *Studios* in Fig. 2. The many-one requirement simply says that no movie can be owned by more than one studio. It does *not* say that a movie must surely be owned by a studio, or that the owning studio must be present in the *Studios* entity set, as stored in our database. An appropriate referential integrity constraint on relationship *Owns* is that for each movie, the owning studio (the entity "referenced" by the relationship for this movie) must exist in our database.

The arrow notation in E/R diagrams is able to indicate whether a relationship is expected to support referential integrity in one or more directions. Suppose R is a relationship from entity set E to entity set F. A rounded arrowhead pointing to F indicates not only that the relationship is many-one from E to F, but that the entity of set F related to a given entity of set E is required to exist. The same idea applies when R is a relationship among more than two entity sets.

Example 19: Figure 18 shows some appropriate referential integrity constraints among the entity sets *Movies*, *Studios*, and *Presidents*. These entity sets and relationships were first introduced in Figs. 2 and 3. We see a rounded arrow entering *Studios* from relationship *Owns*. That arrow expresses the referential integrity constraint that every movie must be owned by one studio, and this studio is present in the *Studios* entity set.

Figure 18: E/R diagram showing referential integrity constraints

Similarly, we see a rounded arrow entering *Studios* from *Runs*. That arrow expresses the referential integrity constraint that every president runs a studio that exists in the *Studios* entity set.

Note that the arrow to *Presidents* from *Runs* remains a pointed arrow. That choice reflects a reasonable assumption about the relationship between studios

and their presidents. If a studio ceases to exist, its president can no longer be called a president, so we would expect the president of the studio to be deleted from the entity set *Presidents*. Hence there is a rounded arrow to *Studios*. On the other hand, if a president were fired or resigned, the studio would continue to exist. Thus, we place an ordinary, pointed arrow to *Presidents*, indicating that each studio has at most one president, but might have no president at some time. □

3.4 Degree Constraints

In the E/R model, we can attach a bounding number to the edges that connect a relationship to an entity set, indicating limits on the number of entities that can be connected to any one entity of the related entity set. For example, we could choose to place a constraint on the degree of a relationship, such as that a movie entity cannot be connected by relationship *Stars-in* to more than 10 star entities.

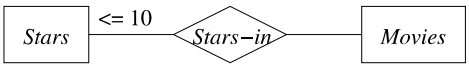

Figure 19: Representing a constraint on the number of stars per movie

Figure 19 shows how we can represent this constraint. As another example, we can think of the arrow as a synonym for the constraint "≤ 1," and we can think of the rounded arrow of Fig. 18 as standing for the constraint "$= 1$."

3.5 Exercises for Section 3

Exercise 3.1 : For your E/R diagrams of:

a) Exercise 1.1.

b) Exercise 1.3.

c) Exercise 1.6.

(*i*) Select and specify keys, and (*ii*) Indicate appropriate referential integrity constraints.

! Exercise 3.2 : We may think of relationships in the E/R model as having keys, just as entity sets do. Let R be a relationship among the entity sets $E_1, E_2, \ldots, E_n$. Then a *key* for R is a set K of attributes chosen from the attributes of $E_1, E_2, \ldots, E_n$ such that if $(e_1, e_2, \ldots, e_n)$ and $(f_1, f_2, \ldots, f_n)$ are two different tuples in the relationship set for R, then it is not possible that these tuples agree in all the attributes of K. Now, suppose $n = 2$; that is, R is a binary relationship. Also, for each i, let K_i be a set of attributes that is a key for entity set E_i. In terms of E_1 and E_2, give a smallest possible key for R under the assumption that:

a) R is many-many.

b) R is many-one from E_1 to E_2.

c) R is many-one from E_2 to E_1.

d) R is one-one.

!! Exercise 3.3: Consider again the problem of Exercise 3.2, but with n allowed to be any number, not just 2. Using only the information about which arcs from R to the E_i's have arrows, show how to find a smallest possible key K for R in terms of the K_i's.

4 Weak Entity Sets

It is possible for an entity set's key to be composed of attributes, some or all of which belong to another entity set. Such an entity set is called a *weak entity set*.

4.1 Causes of Weak Entity Sets

There are two principal reasons we need weak entity sets. First, sometimes entity sets fall into a hierarchy based on classifications unrelated to the "isa hierarchy" of Section 1.11. If entities of set E are subunits of entities in set F, then it is possible that the names of E-entities are not unique until we take into account the name of the F-entity to which the E entity is subordinate. Several examples will illustrate the problem.

Example 20: A movie studio might have several film crews. The crews might be designated by a given studio as crew 1, crew 2, and so on. However, other studios might use the same designations for crews, so the attribute *number* is not a key for crews. Rather, to name a crew uniquely, we need to give both the name of the studio to which it belongs and the number of the crew. The situation is suggested by Fig. 20. The double-rectangle indicates a weak entity set, and the double-diamond indicates a many-one relationship that helps provide the key for the weak entity set. The notation will be explained further in Section 4.3. The key for weak entity set *Crews* is its own *number* attribute and the *name* attribute of the unique studio to which the crew is related by the many-one *Unit-of* relationship. □

Example 21: A species is designated by its genus and species names. For example, humans are of the species *Homo sapiens*; *Homo* is the genus name and *sapiens* the species name. In general, a genus consists of several species, each of which has a name beginning with the genus name and continuing with the species name. Unfortunately, species names, by themselves, are not unique.

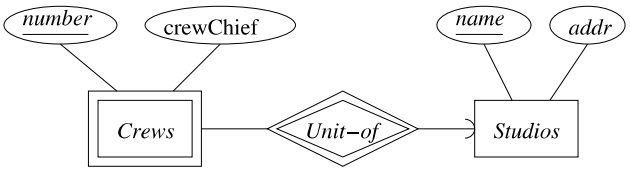

Figure 20: A weak entity set for crews, and its connections

Two or more genera may have species with the same species name. Thus, to designate a species uniquely we need both the species name and the name of the genus to which the species is related by the *Belongs-to* relationship, as suggested in Fig. 21. *Species* is a weak entity set whose key comes partially from its genus. □

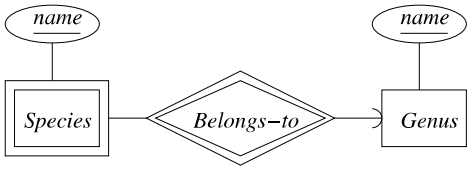

Figure 21: Another weak entity set, for species

The second common source of weak entity sets is the connecting entity sets that we introduced in Section 1.10 as a way to eliminate a multiway relationship.[3] These entity sets often have no attributes of their own. Their key is formed from the attributes that are the key attributes for the entity sets they connect.

Example 22: In Fig. 22 we see a connecting entity set *Contracts* that replaces the ternary relationship *Contracts* of Example 5. *Contracts* has an attribute *salary*, but this attribute does not contribute to the key. Rather, the key for a contract consists of the name of the studio and the star involved, plus the title and year of the movie involved. □

4.2 Requirements for Weak Entity Sets

We cannot obtain key attributes for a weak entity set indiscriminately. Rather, if E is a weak entity set then its key consists of:

1. Zero or more of its own attributes, and

[3]Remember that there is no particular requirement in the E/R model that multiway relationships be eliminated, although this requirement exists in some other database design models.

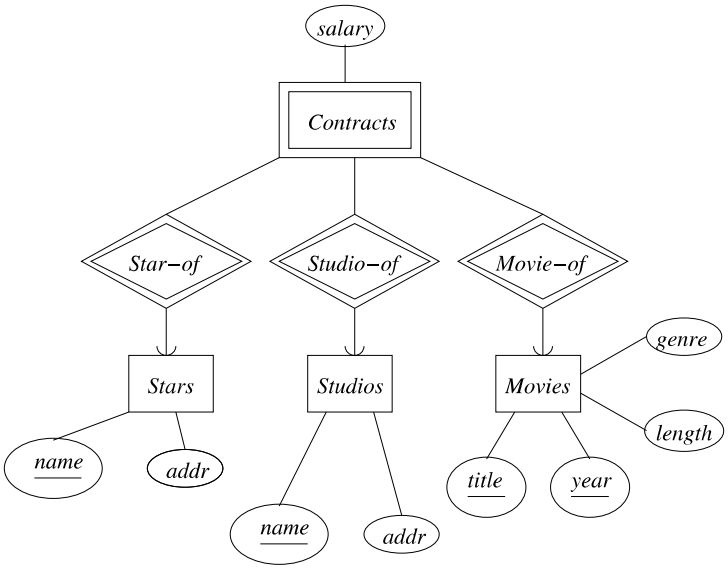

Figure 22: Connecting entity sets are weak

2. Key attributes from entity sets that are reached by certain many-one relationships from E to other entity sets. These many-one relationships are called *supporting relationships* for E, and the entity sets reached from E are *supporting entity sets*.

In order for R, a many-one relationship from E to some entity set F, to be a supporting relationship for E, the following conditions must be obeyed:

a) R must be a binary, many-one relationship[4] from E to F.

b) R must have referential integrity from E to F. That is, for every E-entity, there must be exactly one existing F-entity related to it by R. Put another way, a rounded arrow from R to F must be justified.

c) The attributes that F supplies for the key of E must be key attributes of F.

d) However, if F is itself weak, then some or all of the key attributes of F supplied to E will be key attributes of one or more entity sets G to which F is connected by a supporting relationship. Recursively, if G is weak, some key attributes of G will be supplied from elsewhere, and so on.

[4]Remember that a one-one relationship is a special case of a many-one relationship. When we say a relationship must be many-one, we always include one-one relationships as well.

e) If there are several different supporting relationships from E to the same entity set F, then each relationship is used to supply a copy of the key attributes of F to help form the key of E. Note that an entity e from E may be related to different entities in F through different supporting relationships from E. Thus, the keys of several different entities from F may appear in the key values identifying a particular entity e from E.

The intuitive reason why these conditions are needed is as follows. Consider an entity in a weak entity set, say a crew in Example 20. Each crew is unique, abstractly. In principle we can tell one crew from another, even if they have the same number but belong to different studios. It is only the data about crews that makes it hard to distinguish crews, because the number alone is not sufficient. The only way we can associate additional information with a crew is if there is some deterministic process leading to additional values that make the designation of a crew unique. But the only unique values associated with an abstract crew entity are:

1. Values of attributes of the *Crews* entity set, and
2. Values obtained by following a relationship from a crew entity to a unique entity of some other entity set, where that other entity has a unique associated value of some kind. That is, the relationship followed must be many-one to the other entity set F, and the associated value must be part of a key for F.

4.3 Weak Entity Set Notation

We shall adopt the following conventions to indicate that an entity set is weak and to declare its key attributes.

1. If an entity set is weak, it will be shown as a rectangle with a double border. Examples of this convention are *Crews* in Fig. 20 and *Contracts* in Fig. 22.
2. Its supporting many-one relationships will be shown as diamonds with a double border. Examples of this convention are *Unit-of* in Fig. 20 and all three relationships in Fig. 22.
3. If an entity set supplies any attributes for its own key, then those attributes will be underlined. An example is in Fig. 20, where the number of a crew participates in its own key, although it is not the complete key for *Crews*.

We can summarize these conventions with the following rule:

- Whenever we use an entity set E with a double border, it is weak. The key for E is whatever attributes of E are underlined plus the key attributes of those entity sets to which E is connected by many-one relationships with a double border.

We should remember that the double-diamond is used only for supporting relationships. It is possible for there to be many-one relationships from a weak entity set that are not supporting relationships, and therefore do not get a double diamond.

Example 23: In Fig. 22, the relationship *Studio-of* need not be a supporting relationship for *Contracts*. The reason is that each movie has a unique owning studio, determined by the (not shown) many-one relationship from *Movies* to *Studios*. Thus, if we are told the name of a star and a movie, there is at most one contract with any studio for the work of that star in that movie. In terms of our notation, it would be appropriate to use an ordinary single diamond, rather than the double diamond, for *Studio-of* in Fig. 22. □

4.4 Exercises for Section 4

Exercise 4.1: One way to represent students and the grades they get in courses is to use entity sets corresponding to students, to courses, and to "enrollments." Enrollment entities form a "connecting" entity set between students and courses and can be used to represent not only the fact that a student is taking a certain course, but the grade of the student in the course. Draw an E/R diagram for this situation, indicating weak entity sets and the keys for the entity sets. Is the grade part of the key for enrollments?

Exercise 4.2: Modify your solution to Exercise 4.1 so that we can record grades of the student for each of several assignments within a course. Again, indicate weak entity sets and keys.

Exercise 4.3: For your E/R diagrams of Exercise 2.6(a)–(c), indicate weak entity sets, supporting relationships, and keys.

Exercise 4.4: Draw E/R diagrams for the following situations involving weak entity sets. In each case indicate keys for entity sets.

a) Entity sets *Courses* and *Departments*. A course is given by a unique department, but its only attribute is its number. Different departments can offer courses with the same number. Each department has a unique name.

! b) Entity sets *Leagues*, *Teams*, and *Players*. League names are unique. No league has two teams with the same name. No team has two players with the same number. However, there can be players with the same number on different teams, and there can be teams with the same name in different leagues.

5 From E/R Diagrams to Relational Designs

To a first approximation, converting an E/R design to a relational database schema is straightforward:

- Turn each entity set into a relation with the same set of attributes, and
- Replace a relationship by a relation whose attributes are the keys for the connected entity sets.

While these two rules cover much of the ground, there are also several special situations that we need to deal with, including:

1. Weak entity sets cannot be translated straightforwardly to relations.
2. "Isa" relationships and subclasses require careful treatment.
3. Sometimes, we do well to combine two relations, especially the relation for an entity set E and the relation that comes from a many-one relationship from E to some other entity set.

5.1 From Entity Sets to Relations

Let us first consider entity sets that are not weak. We shall take up the modifications needed to accommodate weak entity sets in Section 5.4. For each non-weak entity set, we shall create a relation of the same name and with the same set of attributes. This relation will not have any indication of the relationships in which the entity set participates; we'll handle relationships with separate relations, as discussed in Section 5.2.

Example 24: Consider the three entity sets *Movies*, *Stars* and *Studios* from Fig. 17, which we reproduce here as Fig. 23. The attributes for the *Movies* entity set are *title*, *year*, *length*, and *genre*.

Next, consider the entity set *Stars* from Fig. 23. There are two attributes, *name* and *address*. Thus, we would expect the corresponding `Stars` relation to have schema `Stars(name, address)` and for

name	*address*
`Carrie Fisher`	`123 Maple St., Hollywood`
`Mark Hamill`	`456 Oak Rd., Brentwood`
`Harrison Ford`	`789 Palm Dr., Beverly Hills`

to be a typical instance. □

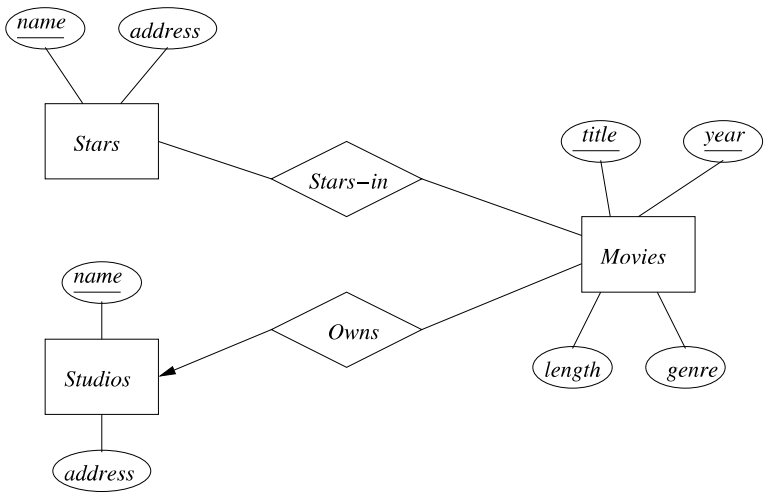

Figure 23: E/R diagram for the movie database

5.2 From E/R Relationships to Relations

Relationships in the E/R model are also represented by relations. The relation for a given relationship R has the following attributes:

1. For each entity set involved in relationship R, we take its key attribute or attributes as part of the schema of the relation for R.

2. If the relationship has attributes, then these are also attributes of relation R.

If one entity set is involved several times in a relationship, in different roles, then its key attributes each appear as many times as there are roles. We must rename the attributes to avoid name duplication. More generally, should the same attribute name appear twice or more among the attributes of R itself and the keys of the entity sets involved in relationship R, then we need to rename to avoid duplication.

Example 25: Consider the relationship *Owns* of Fig. 23. This relationship connects entity sets *Movies* and *Studios*. Thus, for the schema of relation `Owns` we use the key for *Movies*, which is *title* and *year*, and the key of *Studios*, which is *name*. That is, the schema for relation `Owns` is:

```
Owns(title, year, studioName)
```

A sample instance of this relation is:

title	*year*	*studioName*
Star Wars	1977	Fox
Gone With the Wind	1939	MGM
Wayne's World	1992	Paramount

We have chosen the attribute `studioName` for clarity; it corresponds to the attribute *name* of *Studios*. □

title	*year*	*starName*
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Gone With the Wind	1939	Vivien Leigh
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

Figure 24: A relation for relationship *Stars-In*

Example 26: Similarly, the relationship *Stars-In* of Fig. 23 can be transformed into a relation with the attributes `title` and `year` (the key for *Movies*) and attribute `starName`, which is the key for entity set *Stars*. Figure 24 shows a sample relation `Stars-In`. □

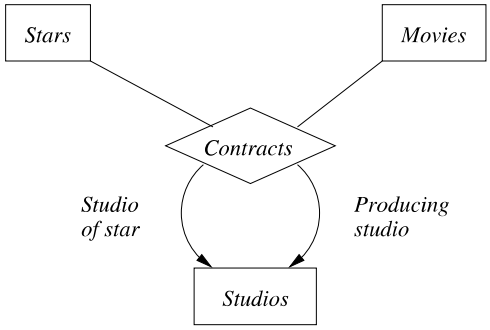

Figure 25: The relationship *Contracts*

Example 27: Multiway relationships are also easy to convert to relations. Consider the four-way relationship *Contracts* of Fig. 6, reproduced here as Fig. 25, involving a star, a movie, and two studios — the first holding the star's contract and the second contracting for that star's services in that movie. We represent this relationship by a relation `Contracts` whose schema consists of the attributes from the keys of the following four entity sets:

1. The key `starName` for the star.
2. The key consisting of attributes `title` and `year` for the movie.
3. The key `studioOfStar` indicating the name of the first studio; recall we assume the studio name is a key for the entity set `Studios`.
4. The key `producingStudio` indicating the name of the studio that will produce the movie using that star.

That is, the relation schema is:

```
Contracts(starName, title, year, studioOfStar, producingStudio)
```

Notice that we have been inventive in choosing attribute names for our relation schema, avoiding "name" for any attribute, since it would be unobvious whether that referred to a star's name or studio's name, and in the latter case, which studio role. Also, were there attributes attached to entity set *Contracts*, such as *salary*, these attributes would be added to the schema of relation `Contracts`. □

5.3 Combining Relations

Sometimes, the relations that we get from converting entity sets and relationships to relations are not the best possible choice of relations for the given data. One common situation occurs when there is an entity set E with a many-one relationship R from E to F. The relations from E and R will each have the key for E in their relation schema. In addition, the relation for E will have in its schema the attributes of E that are not in the key, and the relation for R will have the key attributes of F and any attributes of R itself. Because R is many-one, all these attributes are functionally determined by the key for E, and we can combine them into one relation with a schema consisting of:

1. All attributes of E.
2. The key attributes of F.
3. Any attributes belonging to relationship R.

For an entity e of E that is not related to any entity of F, the attributes of types (2) and (3) will have null values in the tuple for e.

Example 28: In our running movie example, *Owns* is a many-one relationship from *Movies* to *Studios*, which we converted to a relation in Example 25. The relation obtained from entity set *Movies* was discussed in Example 24. We can combine these relations by taking all their attributes and forming one relation schema. If we do, the relation looks like that in Fig. 26. □

title	*year*	*length*	*genre*	*studioName*
Star Wars	1977	124	sciFi	Fox
Gone With the Wind	1939	239	drama	MGM
Wayne's World	1992	95	comedy	Paramount

Figure 26: Combining relation `Movies` with relation `Owns`

Whether or not we choose to combine relations in this manner is a matter of judgement. However, there are some advantages to having all the attributes that are dependent on the key of entity set E together in one relation, even if there are a number of many-one relationships from E to other entity sets. For example, it is often more efficient to answer queries involving attributes of one relation than to answer queries involving attributes of several relations. In fact, some design systems based on the E/R model combine these relations automatically.

On the other hand, one might wonder if it made sense to combine the relation for E with the relation of a relationship R that involved E but was not many-one from E to some other entity set. Doing so is risky, because it often leads to redundancy, as the next example shows.

Example 29: To get a sense of what can go wrong, suppose we combined the relation of Fig. 26 with the relation that we get for the many-many relationship *Stars-in*; recall this relation was suggested by Fig. 24. Then the combined relation would look like Fig. 27. This relation has anomalies that we need to remove by the process of normalization. □

title	*year*	*length*	*genre*	*studioName*	*starName*
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Figure 27: The relation `Movies` with star information

5.4 Handling Weak Entity Sets

When a weak entity set appears in an E/R diagram, we need to do three things differently.

1. The relation for the weak entity set W itself must include not only the attributes of W but also the key attributes of the supporting entity sets. The supporting entity sets are easily recognized because they are reached by supporting (double-diamond) relationships from W.

2. The relation for any relationship in which the weak entity set W appears must use as a key for W all of its key attributes, including those of other entity sets that contribute to W's key.

3. However, a supporting relationship R, from the weak entity set W to a supporting entity set, need not be converted to a relation at all. The justification is that, as discussed in Section 5.3, the attributes of many-one relationship R's relation will either *be* attributes of the relation for W, or (in the case of attributes on R) can be added to the schema for W's relation.

Of course, when introducing additional attributes to build the key of a weak entity set, we must be careful not to use the same name twice. If necessary, we rename some or all of these attributes.

Example 30: Let us consider the weak entity set *Crews* from Fig. 20, which we reproduce here as Fig. 28. From this diagram we get three relations, whose schemas are:

```
Studios(name, addr)
Crews(number, studioName, crewChief)
Unit-of(number, studioName, name)
```

The first relation, `Studios`, is constructed in a straightforward manner from the entity set of the same name. The second, `Crews`, comes from the weak entity set *Crews*. The attributes of this relation are the key attributes of *Crews* and the one nonkey attribute of *Crews*, which is *crewChief*. We have chosen `studioName` as the attribute in relation `Crews` that corresponds to the attribute *name* in the entity set *Studios*.

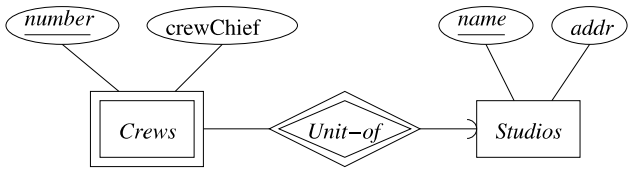

Figure 28: The crews example of a weak entity set

The third relation, `Unit-of`, comes from the relationship of the same name. As always, we represent an E/R relationship in the relational model by a relation whose schema has the key attributes of the related entity sets. In this case,

`Unit-of` has attributes `number` and `studioName`, the key for weak entity set *Crews*, and attribute `name`, the key for entity set *Studios*. However, notice that since *Unit-of* is a many-one relationship, the studio `studioName` is surely the same as the studio `name`.

For instance, suppose Disney crew #3 is one of the crews of the Disney studio. Then the relationship set for E/R relationship *Unit-of* includes the pair

(Disney-crew-#3, Disney)

This pair gives rise to the tuple

(3, Disney, Disney)

for the relation `Unit-of`.

Notice that, as must be the case, the components of this tuple for attributes `studioName` and `name` are identical. As a consequence, we can "merge" the attributes `studioName` and `name` of `Unit-of`, giving us the simpler schema:

```
Unit-of(number, name)
```

However, now we can dispense with the relation `Unit-of` altogether, since its attributes are now a subset of the attributes of relation `Crews`. □

The phenomenon observed in Example 30 — that a supporting relationship needs no relation — is universal for weak entity sets. The following is a modified rule for converting to relations entity sets that are weak.

- If W is a weak entity set, construct for W a relation whose schema consists of:
  1. All attributes of W.
  2. All attributes of supporting relationships for W.
  3. For each supporting relationship for W, say a many-one relationship from W to entity set E, all the *key* attributes of E.

  Rename attributes, if necessary, to avoid name conflicts.
- Do *not* construct a relation for any supporting relationship for W.

5.5 Exercises for Section 5

Exercise 5.1: Convert the E/R diagram of Fig. 29 to a relational database schema.

! Exercise 5.2: There is another E/R diagram that could describe the weak entity set *Bookings* in Fig. 29. Notice that a booking can be identified uniquely by the flight number, day of the flight, the row, and the seat; the customer is not then necessary to help identify the booking.

Relations With Subset Schemas

You might imagine from Example 30 that whenever one relation R has a set of attributes that is a subset of the attributes of another relation S, we can eliminate R. That is not exactly true. R might hold information that doesn't appear in S because the additional attributes of S do not allow us to extend a tuple from R to S.

For instance, the Internal Revenue Service tries to maintain a relation `People(name, ss#)` of potential taxpayers and their social-security numbers, even if the person had no income and did not file a tax return. They might also maintain a relation `TaxPayers(name, ss#, amount)` indicating the amount of tax paid by each person who filed a return in the current year. The schema of `People` is a subset of the schema of `TaxPayers`, yet there may be value in remembering the social-security number of those who are mentioned in `People` but not in `Taxpayers`.

In fact, even identical sets of attributes may have different semantics, so it is not possible to merge their tuples. An example would be two relations `Stars(name, addr)` and `Studios(name, addr)`. Although the schemas look alike, we cannot turn star tuples into studio tuples, or vice-versa.

On the other hand, when the two relations come from the weak-entity-set construction, then there can be no such additional value to the relation with the smaller set of attributes. The reason is that the tuples of the relation that comes from the supporting relationship correspond one-for-one with the tuples of the relation that comes from the weak entity set. Thus, we routinely eliminate the former relation.

a) Revise the diagram of Fig. 29 to reflect this new viewpoint.

b) Convert your diagram from (a) into relations. Do you get the same database schema as in Exercise 5.1?

Exercise 5.3: The E/R diagram of Fig. 30 represents ships. Ships are said to be *sisters* if they were designed from the same plans. Convert this diagram to a relational database schema.

Exercise 5.4: Convert the following E/R diagrams to relational database schemas.

a) Figure 22.

b) Your answer to Exercise 4.1.

c) Your answer to Exercise 4.4(a).

d) Your answer to Exercise 4.4(b).

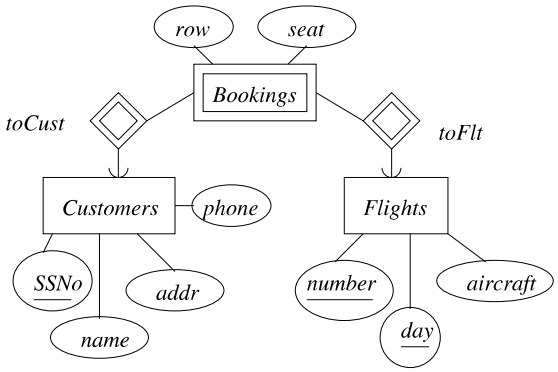

Figure 29: An E/R diagram about airlines

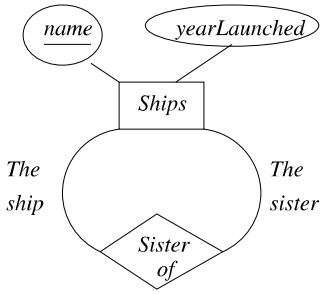

Figure 30: An E/R diagram about sister ships

6 Converting Subclass Structures to Relations

When we have an isa-hierarchy of entity sets, we are presented with several choices of strategy for conversion to relations. Recall we assume that:

- There is a root entity set for the hierarchy,
- This entity set has a key that serves to identify every entity represented by the hierarchy, and
- A given entity may have *components* that belong to the entity sets of any subtree of the hierarchy, as long as that subtree includes the root.

The principal conversion strategies are:

1. *Follow the E/R viewpoint.* For each entity set E in the hierarchy, create a relation that includes the key attributes from the root and any attributes belonging to E.

2. *Treat entities as objects belonging to a single class.* For each possible subtree that includes the root, create one relation, whose schema includes all the attributes of all the entity sets in the subtree.

3. *Use null values.* Create one relation with all the attributes of all the entity sets in the hierarchy. Each entity is represented by one tuple, and that tuple has a null value for whatever attributes the entity does not have.

We shall consider each approach in turn.

6.1 E/R-Style Conversion

Our first approach is to create a relation for each entity set, as usual. If the entity set E is not the root of the hierarchy, then the relation for E will include the key attributes at the root, to identify the entity represented by each tuple, plus all the attributes of E. In addition, if E is involved in a relationship, then we use these key attributes to identify entities of E in the relation corresponding to that relationship.

Note, however, that although we spoke of "isa" as a relationship, it is unlike other relationships, in that it connects components of a single entity, not distinct entities. Thus, we do not create a relation for "isa."

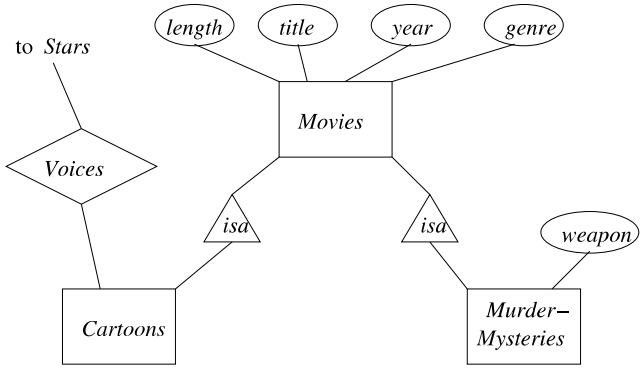

Figure 31: The movie hierarchy

Example 31: Consider the hierarchy of Fig. 10, which we reproduce here as Fig. 31. The relations needed to represent the entity sets in this hierarchy are:

1. `Movies(title, year, length, genre)`. This relation was discussed in Example 24, and every movie is represented by a tuple here.

2. `MurderMysteries(title, year, weapon)`. The first two attributes are the key for all movies, and the last is the lone attribute for the corresponding entity set. Those movies that are murder mysteries have a tuple here as well as in `Movies`.

3. `Cartoons(title, year)`. This relation is the set of cartoons. It has no attributes other than the key for movies, since the extra information about cartoons is contained in the relationship *Voices*. Movies that are cartoons have a tuple here as well as in `Movies`.

Note that the fourth kind of movie — those that are both cartoons and murder mysteries — have tuples in all three relations.

In addition, we shall need the relation `Voices(title, year, starName)` that corresponds to the relationship *Voices* between *Stars* and *Cartoons*. The last attribute is the key for *Stars* and the first two form the key for *Cartoons*.

For instance, the movie *Roger Rabbit* would have tuples in all four relations. Its basic information would be in `Movies`, the murder weapon would appear in `MurderMysteries`, and the stars that provided voices for the movie would appear in `Voices`.

Notice that the relation `Cartoons` has a schema that is a subset of the schema for the relation `Voices`. In many situations, we would be content to eliminate a relation such as `Cartoons`, since it appears not to contain any information beyond what is in `Voices`. However, there may be silent cartoons in our database. Those cartoons would have no voices, and we would therefore lose information should we eliminate relation `Cartoons`. □

6.2 An Object-Oriented Approach

An alternative strategy for converting isa-hierarchies to relations is to enumerate all the possible subtrees of the hierarchy. For each, create one relation that represents entities having components in exactly those subtrees. The schema for this relation has all the attributes of any entity set in the subtree. We refer to this approach as "object-oriented," since it is motivated by the assumption that entities are "objects" that belong to one and only one class.

Example 32: Consider the hierarchy of Fig. 31. There are four possible subtrees including the root:

1. *Movies* alone.

2. *Movies* and *Cartoons* only.

3. *Movies* and *Murder-Mysteries* only.

4. All three entity sets.

We must construct relations for all four "classes." Since only Murder-Mysteries contributes an attribute that is unique to its entities, there is actually some repetition, and these four relations are:

```
Movies(title, year, length, genre)
MoviesC(title, year, length, genre)
MoviesMM(title, year, length, genre, weapon)
MoviesCMM(title, year, length, genre, weapon)
```

If *Cartoons* had attributes unique to that entity set, then all four relations would have different sets of attributes. As that is not the case here, we could combine `Movies` with `MoviesC` (i.e., create one relation for non-murder-mysteries) and combine `MoviesMM` with `MoviesCMM` (i.e., create one relation for all murder mysteries), although doing so loses some information — which movies are cartoons.

We also need to consider how to handle the relationship *Voices* from *Cartoons* to *Stars*. If *Voices* were many-one from *Cartoons*, then we could add a `voice` attribute to `MoviesC` and `MoviesCMM`, which would represent the *Voices* relationship and would have the side-effect of making all four relations different. However, *Voices* is many-many, so we need to create a separate relation for this relationship. As always, its schema has the key attributes from the entity sets connected; in this case

```
Voices(title, year, starName)
```

would be an appropriate schema.

One might consider whether it was necessary to create two such relations, one connecting cartoons that are not murder mysteries to their voices, and the other for cartoons that *are* murder mysteries. However, there does not appear to be any benefit to doing so in this case. □

6.3 Using Null Values to Combine Relations

There is one more approach to representing information about a hierarchy of entity sets. If we are allowed to use `NULL` (the null value as in SQL) as a value in tuples, we can handle a hierarchy of entity sets with a single relation. This relation has all the attributes belonging to any entity set of the hierarchy. An entity is then represented by a single tuple. This tuple has `NULL` in each attribute that is not defined for that entity.

Example 33: If we applied this approach to the diagram of Fig. 31, we would create a single relation whose schema is:

```
Movie(title, year, length, genre, weapon)
```

Those movies that are not murder mysteries would have `NULL` in the `weapon` component of their tuple. It would also be necessary to have a relation `Voices` to connect those movies that are cartoons to the stars performing the voices, as in Example 32. □

6.4 Comparison of Approaches

Each of the three approaches, which we shall refer to as "straight-E/R," "object-oriented," and "nulls," respectively, have advantages and disadvantages. Here is a list of the principal issues.

1. It can be expensive to answer queries involving several relations, so we would prefer to find all the attributes we needed to answer a query in one relation. The nulls approach uses only one relation for all the attributes, so it has an advantage in this regard. The other two approaches have advantages for different kinds of queries. For instance:

   (a) A query like "what films of 2008 were longer than 150 minutes?" can be answered directly from the relation `Movies` in the straight-E/R approach of Example 31. However, in the object-oriented approach of Example 32, we need to examine `Movies`, `MoviesC`, `MoviesMM`, and `MoviesCMM`, since a long movie may be in any of these four relations.

   (b) On the other hand, a query like "what weapons were used in cartoons of over 150 minutes in length?" gives us trouble in the straight-E/R approach. We must access `Movies` to find those movies of over 150 minutes. We must access `Cartoons` to verify that a movie is a cartoon, and we must access `MurderMysteries` to find the murder weapon. In the object-oriented approach, we have only to access the relation `MoviesCMM`, where all the information we need will be found.

2. We would like not to use too many relations. Here again, the nulls method shines, since it requires only one relation. However, there is a difference between the other two methods, since in the straight-E/R approach, we use only one relation per entity set in the hierarchy. In the object-oriented approach, if we have a root and n children ($n + 1$ entity sets in all), then there are 2^n different classes of entities, and we need that many relations.

3. We would like to minimize space and avoid repeating information. Since the object-oriented method uses only one tuple per entity, and that tuple has components for only those attributes that make sense for the entity, this approach offers the minimum possible space usage. The nulls approach also has only one tuple per entity, but these tuples are "long"; i.e., they have components for all attributes, whether or not they are appropriate for a given entity. If there are many entity sets in the hierarchy, and there are many attributes among those entity sets, then a large fraction of the space could be wasted in the nulls approach. The straight-E/R method has several tuples for each entity, but only the key attributes are repeated. Thus, this method could use either more or less space than the nulls method.

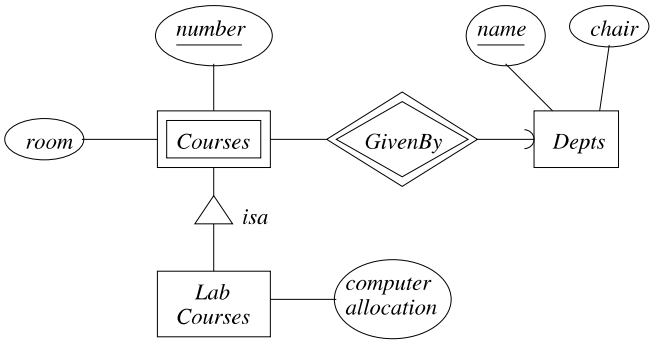

Figure 32: E/R diagram for Exercise 6.1

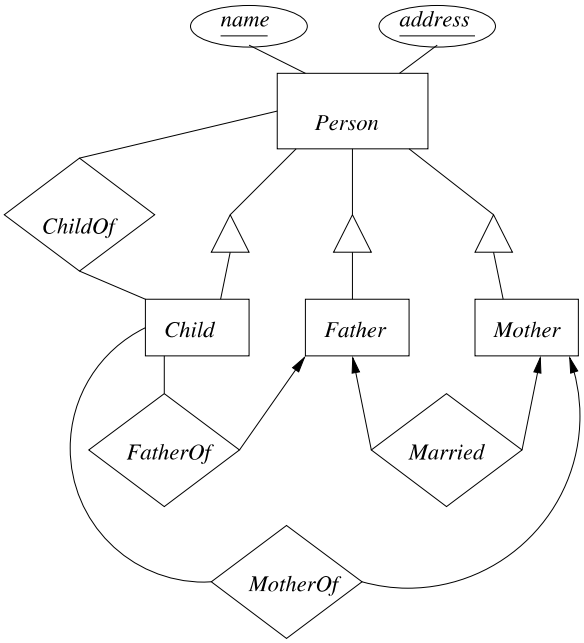

Figure 33: E/R diagram for Exercise 6.2

6.5 Exercises for Section 6

Exercise 6.1: Convert the E/R diagram of Fig. 32 to a relational database schema, using each of the following approaches:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

! Exercise 6.2: Convert the E/R diagram of Fig. 33 to a relational database schema, using:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

Exercise 6.3: Convert your E/R design from Exercise 1.7 to a relational database schema, using:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

! Exercise 6.4: Suppose that we have an isa-hierarchy involving e entity sets. Each entity set has a attributes, and k of those at the root form the key for all these entity sets. Give formulas for (i) the minimum and maximum number of relations used, and (ii) the minimum and maximum number of components that the tuple(s) for a single entity have all together, when the method of conversion to relations is:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

7 Unified Modeling Language

UML (*Unified Modeling Language*) was developed originally as a graphical notation for describing software designs in an object-oriented style. It has been extended, with some modifications, to be a popular notation for describing database designs, and it is this portion of UML that we shall study here. UML offers much the same capabilities as the E/R model, with the exception of multiway relationships. UML also offers the ability to treat entity sets as true classes, with methods as well as data. Figure 34 summarizes the common concepts, with different terminology, used by E/R and UML.

UML	E/R Model
Class	Entity set
Association	Binary relationship
Association Class	Attributes on a relationship
Subclass	Isa hierarchy
Aggregation	Many-one relationship
Composition	Many-one relationship with referential integrity

Figure 34: Comparison between UML and E/R terminology

7.1 UML Classes

A class in UML is similar to an entity set in the E/R model. The notation for a class is rather different, however. Figure 35 shows the class that corresponds to the E/R entity set *Movies* from our running example of this chapter.

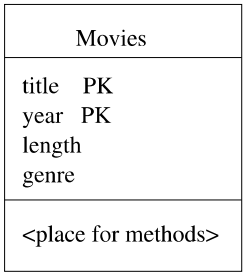

Figure 35: The *Movies* class in UML

The box for a class is divided into three parts. At the top is the name of the class. The middle has the attributes, which are like instance variables of a class. In our *Movies* class, we use the attributes *title, year, length*, and *genre*.

The bottom portion is for methods. Neither the E/R model nor the relational model provides methods. However, they are an important concept, and one that actually appears in modern relational systems, called "object-relational" DBMS.

Example 34: We might have added an instance method *lengthInHours*(). The UML specification doesn't tell anything more about a method than the types of any arguments and the type of its return-value. Perhaps this method returns *length*/60.0, but we cannot know from the design. □

In this section, we shall not use methods in our design. Thus, in the future, UML class boxes will have only two sections, for the class name and the attributes.

7.2 Keys for UML classes

As for entity sets, we can declare one key for a UML class. To do so, we follow each attribute in the key by the letters PK, standing for "primary key." There is no convenient way to stipulate that several attributes or sets of attributes are each keys.

Example 35: In Fig. 35, we have made our standard assumption that *title* and *year* together form the key for *Movies*. Notice that PK appears on the lines for these attributes and not for the others. □

7.3 Associations

A binary relationship between classes is called an *association*. There is no analog of multiway relationships in UML. Rather, a multiway relationship has to be broken into binary relationships, which as we suggested in Section 1.10, can always be done. The interpretation of an association is exactly what we described for relationships in Section 1.5 on relationship sets. The association is a set of pairs of objects, one from each of the classes it connects.

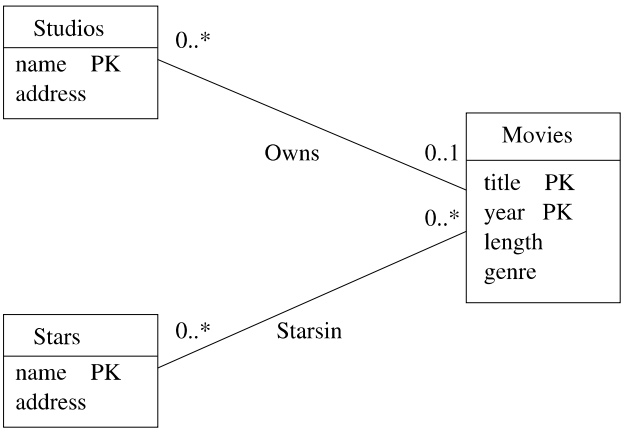

Figure 36: Movies, stars, and studios in UML

We draw a UML association between two classes simply by drawing a line between them, and giving the line a name. Usually, we'll place the name below the line. For example, Fig. 36 is the UML analog of the E/R diagram of Fig. 17. There are two associations, *Stars-in* and *Owns*; the first connects *Movies* with *Stars* and the second connects *Movies* with *Studios*.

Every association has constraints on the number of objects from each of its classes that can be connected to an object of the other class. We indicate these constraints by a label of the form $m..n$ at each end. The meaning of this label is that each object at the other end is connected to at least m and at most n objects at this end. In addition:

- A $*$ in place of n, as in $m..*$, stands for "infinity." That is, there is no upper limit.
- A $*$ alone, in place of $m..n$, stands for the range $0..*$, that is, no constraint at all on the number of objects.
- If there is no label at all at an end of an association edge, then the label is taken to be 1..1, i.e., "exactly one."

Example 36: In Fig. 36 we see $0..*$ at the *Movies* end of both associations. That says that a star appears in zero or more movies, and a studio owns zero or more movies; i.e., there is no constraint for either. There is also a $0..*$ at the *Stars* end of association *Stars-in*, telling us that a movie has any number of stars. However, the label on the *Studios* end of association *Owns* is 0..1, which means either 0 or 1 studio. That is, a given movie can either be owned by one studio, or not be owned by any studio in the database. Notice that this constraint is exactly what is said by the pointed arrow entering *Studios* in the E/R diagram of Fig. 17. □

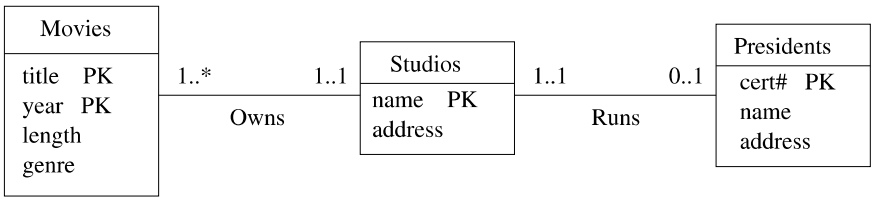

Figure 37: Expressing referential integrity in UML

Example 37: The UML diagram of Fig. 37 is intended to mirror the E/R diagram of Fig. 18. Here, we see assumptions somewhat different from those in Example 36, about the numbers of movies and studios that can be associated. The label $1..*$ at the *Movies* end of *Owns* says that each studio must own at least one movie (or else it isn't really a studio). There is still no upper limit on how many movies a studio can own.

At the *Studios* end of *Owns*, we see the label 1..1. That label says that a movie must be owned by one studio and only one studio. It is not possible for a movie not to be owned by any studio, as was possible in Fig. 36. The label 1..1 says exactly what the rounded arrow in E/R diagrams says.

We also see the association *Runs* between studios and presidents. At the *Studios* end we see label 1..1. That is, a president must be the president of one and only one studio. That label reflects the same constraint as the rounded arrow from *Presidents* to *Studios* in Fig. 18. At the other end of association *Runs* is the label 0..1. That label says that a studio can have at most one president, but it could not have a president at some time. This constraint is exactly the constraint of a pointed arrow. □

7.4 Self-Associations

An association can have both ends at the same class; such an association is called a *self-association*. To distinguish the two roles played by one class in a self-association, we give the association two names, one for each end.

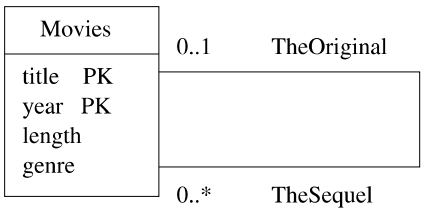

Figure 38: A self-association representing sequels of movies

Example 38: Figure 38 represents the relationship "sequel-of" on movies. We see one association with each end at the class *Movies*. The end with role *TheOriginal* points to the original movie, and it has label 0..1. That is, for a movie to be a sequel, there has to be exactly one movie that was the original. However, some movies are not sequels of any movie. The other role, *TheSequel* has label 0..$*$. The reasoning is that an original can have any number of sequels. Note we take the point of view that there is an original movie for any sequence of sequels, and a sequel is a sequel of the original, not of the previous movie in the sequence. For instance, *Rocky II* through *Rocky V* are sequels of *Rocky*. We do not assume *Rocky IV* is a sequel of *Rocky III*, and so on. □

7.5 Association Classes

We can attach attributes to an association in much the way we did in the E/R model, in Section 1.9.[5] In UML, we create a new class, called an *association class*, and attach it to the middle of the association. The association class has its own name, but its attributes may be thought of as attributes of the association to which it attaches.

Example 39: Suppose we want to add to the association *Stars-in* between *Movies* and *Stars* some information about the compensation the star received for the movie. This information is not associated with the movie (different stars get different salaries) nor with the star (stars can get different salaries for different movies). Thus, we must attach this information with the association itself. That is, every movie-star pair has its own salary information.

Figure 39 shows the association *Stars-in* with an association class called *Compensation*. This class has two attributes, *salary* and *residuals*. Notice

[5]However, the example there in Fig. 7 will not carry over directly, because the relationship there is 3-way.

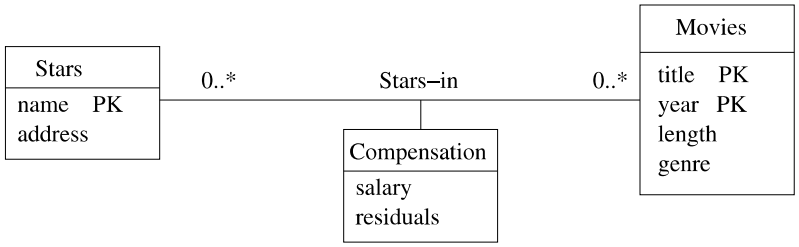

Figure 39: *Compensation* is an association class for the association *Stars-in*

that there is no primary key marked for *Compensation*. When we convert a diagram such as Fig. 39 to relations, the attributes of *Compensation* will attach to tuples created for movie-star pairs, as was described for relationships in Section 5.2. □

7.6 Subclasses in UML

Any UML class can have a hierarchy of subclasses below it. The primary key comes from the root of the hierarchy, just as with E/R hierarchies. UML permits a class C to have four different kinds of subclasses below it, depending on our choices of answer to two questions:

1. *Complete versus Partial.* Is every object in the class C a member of some subclass? If so, the subclasses are *complete*; otherwise they are *partial* or *incomplete*.

2. *Disjoint versus Overlapping.* Are the subclasses *disjoint* (an object cannot be in two of the subclasses)? If an object can be in two or more of the subclasses, then the subclasses are said to be *overlapping*.

Note that these decisions are taken at each level of a hierarchy, and the decisions may be made independently at each point.

There are several interesting relationships between the classification of UML subclasses given above, the standard notion of subclasses in object-oriented systems, and the E/R notion of subclasses.

- In a typical object-oriented system, subclasses are disjoint. That is, no object can be in two classes. Of course they inherit properties from their parent class, so in a sense, an object also "belongs" in the parent class. However, the object may not also be in a sibling class.

- The E/R model automatically allows overlapping subclasses.

- Both the E/R model and object-oriented systems allow either complete or partial subclasses. That is, there is no requirement that a member of the superclass be in any subclass.

Subclasses are represented by rectangles, like any class. We assume a subclass inherits the properties (attributes and associations) from its superclass. However, any additional attributes belonging to the subclass are shown in the box for that subclass, and the subclass may have its own, additional, associations to other classes. To represent the class/subclass relationship in UML diagrams, we use a triangular, open arrow pointing to the superclass. The subclasses are usually connected by a horizontal line, feeding into the arrow.

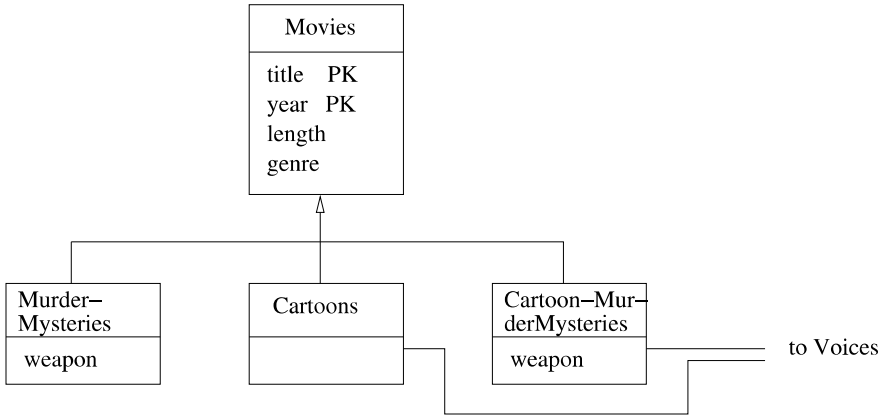

Figure 40: Cartoons and murder mysteries as disjoint subclasses of movies

Example 40: Figure 40 shows a UML variant of the subclass example from Section 1.11. However, unlike the E/R subclasses, which are of necessity overlapping, we have chosen here to make the subclasses disjoint. They are partial, of course, since many movies are neither cartoons nor murder mysteries.

Because the subclasses were chosen disjoint, there must be a third subclass for movies like *Roger Rabbit* that are both cartoons and murder mysteries. Notice that both the classes *MurderMysteries* and *Cartoon-MurderMysteries* have additional attribute *weapon*, while the two subclasses *MurderMysteries* and *Cartoon-MurderMysteries* have associations with the unseen class *Voices*. □

7.7 Aggregations and Compositions

There are two special notations for many-one associations whose implications are rather subtle. In one sense, they reflect the object-oriented style of programming, where it is common for one class to have references to other classes among its attributes. In another sense, these special notations are really stipulations about how the diagram should be converted to relations; we discuss this aspect of the matter in Section 8.3.

An *aggregation* is a line between two classes that ends in an open diamond at one end. The implication of the diamond is that the label at that end must

be 0..1, i.e., the aggregation is a many-one association from the class at the opposite end to the class at the diamond end. Although the aggregation is an association, we do not need to name it, since in practice that name will never be used in a relational implementation.

A *composition* is similar to an association, but the label at the diamond end must be 1..1. That is, every object at the opposite end from the diamond must be connected to exactly one object at the diamond end. Compositions are distinguished by making the diamond be solid black.

Example 41 : In Fig. 41 we see examples of both an aggregation and a composition. It both modifies and elaborates on the situation of Fig. 37. We see an association from *Movies* to *Studios*. The label 1..∗ at the *Movies* end says that a studio has to own at least one movie. We do not need a label at the diamond end, since the open diamond implies a 0..1 label. That is, a movie may or may not be associated with a studio, but cannot be associated with more than one studio. There is also the implication that *Movies* objects will contain a reference to their owning *Studios* object; that reference may be null if the movie is not owned by a studio.

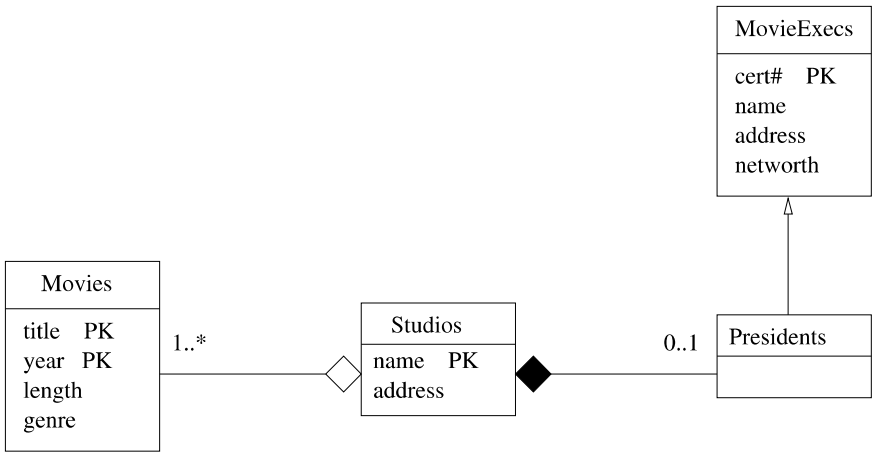

Figure 41: An aggregation from *Movies* to *Studios* and a composition from *Presidents* to *Studios*

At the right, we see the class *MovieExecs* with a subclass *Presidents*. There is a composition from *Presidents* to *Studios*. meaning that every president is the president of exactly one studio. A label 1..1 at the *Studios* end is implied by the solid diamond. The implication of the composition is that *Presidents* objects will contain a reference to a *Studios* object, and that this reference cannot be null. □

7.8 Exercises for Section 7

Exercise 7.1: Draw a UML diagram for the problem of Exercise 1.1.

Exercise 7.2: Modify your diagram from Exercise 7.1 in accordance with the requirements of Exercise 1.2.

Exercise 7.3: Repeat Exercise 1.3 using UML.

Exercise 7.4: Repeat Exercise 1.6 using UML.

Exercise 7.5: Repeat Exercise 1.7 using UML. Are your subclasses disjoint or overlapping? Are they complete or partial?

Exercise 7.6: Repeat Exercise 1.9 using UML.

Exercise 7.7: Convert the E/R diagram of Fig. 30 to a UML diagram.

! Exercise 7.8: How would you represent the 3-way relationship of *Contracts* among movies, stars, and studios (see Fig. 4) in UML?

! Exercise 7.9: Repeat Exercise 2.5 using UML.

Exercise 7.10: Usually, when we constrain associations with a label of the form $m..n$, we find that m and n are each either 0, 1, or $*$. Give some examples of associations where it would make sense for at least one of m and n to be something different.

8 From UML Diagrams to Relations

Many of the ideas needed to turn E/R diagrams into relations work for UML diagrams as well. We shall therefore briefly review the important techniques, dwelling only on points where the two modeling methods diverge.

8.1 UML-to-Relations Basics

Here is an outline of the points that should be familiar from our discussion in Section 5:

- *Classes to Relations.* For each class, create a relation whose name is the name of the class, and whose attributes are the attributes of the class.
- *Associations to Relations.* For each association, create a relation with the name of that association. The attributes of the relation are the key attributes of the two connected classes. If there is a coincidence of attributes between the two classes, rename them appropriately. If there is an association class attached to the association, include the attributes of the association class among the attributes of the relation.

Example 42: Consider the UML diagram of Fig. 36. For the three classes we create relations:

```
Movies(title, year, length genre)
Stars(name, address)
Studios(name, address)
```

For the two associations, we create relations

```
Stars-In(movieTitle, movieYear, starName)
Owns(movieTitle, movieYear, studioName)
```

Note that we have taken some liberties with the names of attributes, for clarity of intention, even though we were not required to do so.

For another example, consider the UML diagram of Fig. 39, which shows an association class. The relations for the classes *Movies* and *Stars* would be the same as above. However, for the association, we would have a relation

```
Stars-In(movieTitle, movieYear, starName, salary, residuals)
```

That is, we add to the key attributes of the associated classes, the two attributes of the association class *Compensation*. Note that there is no relation created for *Compensation* itself. □

8.2 From UML Subclasses to Relations

The three options we enumerated in Section 6 apply to UML subclass hierarchies as well. Recall these options are "E/R style" (relations for each subclass have only the key attributes and attributes of that subclass), "object-oriented" (each entity is represented in the relation for only one subclass), and "use nulls" (one relation for all subclasses). However, if we have information about whether subclasses are disjoint or overlapping, and complete or partial, then we may find one or another method more appropriate. Here are some considerations:

1. If a hierarchy is disjoint at every level, then an object-oriented representation is suggested. We do not have to consider each possible tree of subclasses when forming relations, since we know that each object can belong to only one class and its ancestors in the hierarchy. Thus, there is no possibility of an exponentially exploding number of relations being created.

2. If the hierarchy is both complete and disjoint at every level, then the task is even simpler. If we use the object-oriented approach, then we have only to construct relations for the classes at the leaves of the hierarchy.

3. If the hierarchy is large and overlapping at some or all levels, then the E/R approach is indicated. We are likely to need so many relations that the relational database schema becomes unwieldy.

8.3 From Aggregations and Compositions to Relations

Aggregations and compositions are really types of many-one associations. Thus, one approach to their representation in a relational database schema is to convert them as we do for any association in Section 8.1. Since these elements are not necessarily named in the UML diagram, we need to invent a name for the corresponding relation.

However, there is a hidden assumption that this implementation of aggregations and compositions is undesirable. Recall from Section 5.3 that when we have an entity set E and a many-one relationship R from E to another entity set F, we have the option — some would say the obligation — to combine the relation for E with the relation for R. That is, the one relation constructed from E and R has all the attributes of E plus the key attributes of F.

We suggest that aggregations and compositions be treated routinely in this manner. Construct no relation for the aggregation or composition. Rather, add to the relation for the class at the nondiamond end the key attribute(s) of the class at the diamond end. In the case of an aggregation (but not a composition), it is possible that these attributes can be null.

Example 43: Consider the UML diagram of Fig. 41. Since there is a small hierarchy, we need to decide how *MovieExecs* and *Presidents* will be translated. Let us adopt the E/R approach, so the `Presidents` relation has only the `cert#` attribute from *MovieExecs*.

The aggregation from *Movies* to *Studios* is represented by putting the key *name* for *Studios* among the attributes for the relation `Movies`. The composition from *Presidents* to *Studios* is represented by adding the key for *Studios* to the relation `Presidents` as well. No relations are constructed for the aggregation or the composition. The following are all the relations we construct from this UML diagram.

```
MovieExecs(cert#, name, address, netWorth)
Presidents(cert#, studioName)
Movies(title, year, length, genre, studioName)
Studios(name, address)
```

As before, we take some liberties with names of attributes to make our intentions clear. □

8.4 The UML Analog of Weak Entity Sets

We have not mentioned a UML notation that corresponds to the double-border notation for weak entity sets in the E/R model. There is a sense in which none is needed. The reason is that UML, unlike E/R, draws on the tradition of object-oriented systems, which takes the point of view that each object has its own *object-identity*. That is, we can distinguish two objects, even if they have the same values for each of their attributes and other properties. That object-identity is typically viewed as a reference or pointer to the object.

In UML, we can take the point of view that the objects belonging to a class likewise have object-identity. Thus, even if the stated attributes for a class do not serve to identify a unique object of the class, we can create a new attribute that serves as a key for the corresponding relation and represents the object-identity of the object.

However, it is also possible, in UML, to use a composition as we used supporting relationships for weak entity sets in the E/R model. This composition goes from the "weak" class (the class whose attributes do not provide its key) to the "supporting" class. If there are several "supporting" classes, then several compositions can be used. We shall use a special notation for a *supporting* composition: a small box attached to the *weak* class with "PK" in it will serve as the anchor for the supporting composition. The implication is that the key attribute(s) for the *supporting* class at the other end of the composition is part of the key of the weak class, along with any of the attributes of the weak class that are marked "PK." As with weak entity sets, there can be several supporting compositions and classes, and those supporting classes could themselves be weak, in which case the rule just described is applied recursively.

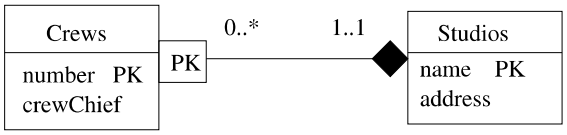

Figure 42: Weak class *Crews* supported by a composition and the class *Studios*

Example 44: Figure 42 shows the analog of the weak entity set *Crews* of Example 20. There is a composition from *Crews* to *Studios* anchored by a box labeled "PK" to indicate that this composition provides part of the key for *Crews*. □

We convert weak structures such as Fig. 42 to relations exactly as we did in Section 5.4. There is a relation for class *Studios* as usual. There is no relation for the composition, again as usual. The relation for class *Crews* includes not only its own attribute *number*, but the key for the class at the end of the composition, which is *Studios*.

Example 45: The relations for Example 44 are thus:

```
Studios(name, address)
Crews(number, crewChief, studioName)
```

As before, we renamed the attribute *name* of *Studios* in the `Crews` relation, for clarity. □

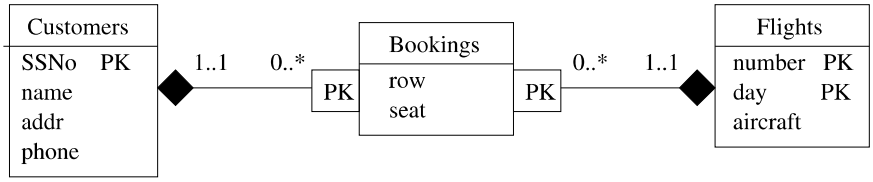

Figure 43: A UML diagram analogous to the E/R diagram of Fig. 29

8.5 Exercises for Section 8

Exercise 8.1: Convert the UML diagram of Fig. 43 to relations.

Exercise 8.2: Convert the following UML diagrams to relations:

a) Figure 37.

b) Figure 40.

c) Your solution to Exercise 7.1.

d) Your solution to Exercise 7.3.

e) Your solution to Exercise 7.4.

f) Your solution to Exercise 7.6.

! Exercise 8.3: How many relations do we create, using the object-oriented approach, if we have a three-level hierarchy with three subclasses of each class at the first and second levels, and that hierarchy is:

a) Disjoint and complete at each level.

b) Disjoint but not complete at each level.

c) Neither disjoint nor complete.

9 Object Definition Language

ODL (*Object Definition Language*) is a text-based language for specifying the structure of databases in object-oriented terms. Like UML, the class is the central concept in ODL. Classes in ODL have a name, attributes, and methods, just as UML classes do. Relationships, which are analogous to UML's associations, are not an independent concept in ODL, but are embedded within classes as an additional family of properties.

9.1 Class Declarations

A declaration of a class in ODL, in its simplest form, is:

```
class <name> {
   <list of properties>
};
```

That is, the keyword `class` is followed by the name of the class and a bracketed list of properties. A property can be an attribute, a relationship, or a method.

9.2 Attributes in ODL

The simplest kind of property is the *attribute*. In ODL, attributes need not be of simple types, such as integers and strings. ODL has a type system, described in Section 9.6, that allows us to form structured types and collection types (e.g., sets). For example, an attribute `address` might have a structured type with fields for the street, city, and zip code. An attribute `phones` might have a set of strings as its type, and even more complex types are possible. An attribute is represented in the declaration for its class by the keyword `attribute`, the type of the attribute, and the name of the attribute.

```
1)  class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Genres
              {drama, comedy, sciFi, teen} genre;
    };
```

Figure 44: An ODL declaration of the class `Movie`

Example 46: In Fig. 44 is an ODL declaration of the class of movies. It is not a complete declaration; we shall add more to it later. Line (1) declares `Movie` to be a class. Following line (1) are the declarations of four attributes that all `Movie` objects will have.

Lines (2), (3), and (4) declare three attributes, `title`, `year`, and `length`. The first of these is of character-string type, and the other two are integers. Line (5) declares attribute `genre` to be of enumerated type. The name of the enumeration (list of symbolic constants) is `Genres`, and the four values the attribute `genre` is allowed to take are `drama`, `comedy`, `sciFi`, and `teen`. An enumeration must have a name, which can be used to refer to the same type anywhere. □

Why Name Enumerations and Structures?

The enumeration-name `Genres` in Fig. 44 appears to play no role. However, by giving this set of symbolic constants a name, we can refer to it elsewhere, including in the declaration of other classes. In some other class, the *scoped name* `Movie::Genres` can be used to refer to the definition of the enumerated type of this name within the class `Movie`.

Example 47: In Example 46, all the attributes have primitive types. Here is an example with a complex type. We can define the class `Star` by

```
1)  class Star {
2)     attribute string name;
3)     attribute Struct Addr
           {string street, string city} address;
    };
```

Line (2) specifies an attribute `name` (of the star) that is a string. Line (3) specifies another attribute `address`. This attribute has a type that is a *record structure*. The name of this structure is `Addr`, and the type consists of two fields: `street` and `city`. Both fields are strings. In general, one can define record structure types in ODL by the keyword `Struct` and curly braces around the list of field names and their types. Like enumerations, structure types must have a name, which can be used elsewhere to refer to the same structure type. □

9.3 Relationships in ODL

An ODL relationship is declared inside a class declaration, by the keyword `relationship`, a type, and the name of the relationship. The type of a relationship describes what a single object of the class is connected to by the relationship. Typically, this type is either another class (if the relationship is many-one) or a collection type (if the relationship is one-many or many-many). We shall show complex types by example, until the full type system is described in Section 9.6.

Example 48: Suppose we want to add to the declaration of the `Movie` class from Example 46 a property that is a set of stars. More precisely, we want each `Movie` object to connect the set of `Star` objects that are its stars. The best way to represent this connection between the `Movie` and `Star` classes is with a *relationship*. We may represent this relationship by a line:

```
relationship Set<Star> stars;
```

in the declaration of class `Movie`. It says that in each object of class `Movie` there is a set of references to `Star` objects. The set of references is called `stars`. □

9.4 Inverse Relationships

Just as we might like to access the stars of a given movie, we might like to know the movies in which a given star acted. To get this information into `Star` objects, we can add the line

```
relationship Set<Movie> starredIn;
```

to the declaration of class `Star` in Example 47. However, this line and a similar declaration for `Movie` omits a very important aspect of the relationship between movies and stars. We expect that if a star S is in the `stars` set for movie M, then movie M is in the `starredIn` set for star S. We indicate this connection between the relationships `stars` and `starredIn` by placing in each of their declarations the keyword `inverse` and the name of the other relationship. If the other relationship is in some other class, as it usually is, then we refer to that relationship by its scoped name — the name of its class, followed by a double colon (`::`) and the name of the relationship.

Example 49: To define the relationship `starredIn` of class `Star` to be the inverse of the relationship `stars` in class `Movie`, we revise the declarations of these classes, as shown in Fig. 45 (which also contains a definition of class `Studio` to be discussed later). Line (6) shows the declaration of relationship `stars` of movies, and says that its inverse is `Star::starredIn`. Since relationship `starredIn` is defined in another class, its scoped name must be used.

Similarly, relationship `starredIn` is declared in line (11). Its inverse is declared by that line to be `stars` of class `Movie`, as it must be, because inverses always are linked in pairs. □

As a general rule, if a relationship R for class C associates with object x of class C with objects $y_1, y_2, \ldots, y_n$ of class D, then the inverse relationship of R associates with each of the y_i's the object x (perhaps along with other objects).

9.5 Multiplicity of Relationships

Like the binary relationships of the E/R model, a pair of inverse relationships in ODL can be classified as either many-many, many-one in either direction, or one-one. The type declarations for the pair of relationships tells us which.

1. If we have a many-many relationship between classes C and D, then in class C the type of the relationship is `Set<`D`>`, and in class D the type is `Set<`C`>`.[6]

[6] Actually, the `Set` could be replaced by another "collection type," such as list or bag, as discussed in Section 9.6. We shall assume all collections are sets in our exposition of relationships, however.

```
 1)  class Movie {
 2)     attribute string title;
 3)     attribute integer year;
 4)     attribute integer length;
 5)     attribute enum Genres
                {drama, comedy, sciFi, teen} genre;
 6)     relationship Set<Star> stars
                     inverse Star::starredIn;
 7)     relationship Studio ownedBy
                     inverse Studio::owns;
     };

 8)  class Star {
 9)     attribute string name;
10)     attribute Struct Addr
            {string street, string city} address;
11)     relationship Set<Movie> starredIn
                     inverse Movie::stars;
     };

12)  class Studio {
13)     attribute string name;
14)     attribute Star::Addr address;
15)     relationship Set<Movie> owns
                     inverse Movie::ownedBy;
     };
```

Figure 45: Some ODL classes and their relationships

2. If the relationship is many-one from C to D, then the type of the relationship in C is just D, while the type of the relationship in D is `Set<`C`>`.

3. If the relationship is many-one from D to C, then the roles of C and D are reversed in (2) above.

4. If the relationship is one-one, then the type of the relationship in C is just D, and in D it is just C.

Note that, as in the E/R model, we allow a many-one or one-one relationship to include the case where for some objects the "one" is actually "none." For instance, a many-one relationship from C to D might have a "null" value of the relationship in some of the C objects. Of course, since a D object could be associated with any set of C objects, it is also permissible for that set to be empty for some D objects.

Example 50: In Fig. 45 we have the declaration of three classes, `Movie`, `Star`, and `Studio`. The first two of these have already been introduced in Examples 46 and 47. We also discussed the relationship pair `stars` and `starredIn`. Since each of their types uses `Set`, we see that this pair represents a many-many relationship between `Star` and `Movie`.

`Studio` objects have attributes `name` and `address`; these appear in lines (13) and (14). We have used the same type for addresses of studios as we defined in class `Star` for addresses of stars.

In line (7) we see a relationship `ownedBy` from movies to studios, and the inverse of this relationship is `owns` on line (15). Since the type of `ownedBy` is `Studio`, while the type of `owns` is `Set<Movie>`, we see that this pair of inverse relationships is many-one from `Movie` to `Studio`. □

9.6 Types in ODL

ODL offers the database designer a type system similar to that found in C or other conventional programming languages. A type system is built from a basis of types that are defined by themselves and certain recursive rules whereby complex types are built from simpler types. In ODL, the basis consists of:

1. *Primitive types*: integer, float, character, character string, boolean, and *enumerations*. The latter are lists of symbolic names, such as `drama` in line (5) of Fig. 45.

2. *Class names*, such as `Movie`, or `Star`, which represent types that are actually structures, with components for each of the attributes and relationships of that class.

These types are combined into structured types using the following *type constructors*:

1. *Set.* If T is any type, then `Set<T>` denotes the type whose values are finite sets of elements of type T. Examples using the set type-constructor occur in lines (6), (11), and (15) of Fig. 45.

2. *Bag.* If T is any type, then `Bag<T>` denotes the type whose values are finite bags or *multisets* of elements of type T.

3. *List.* If T is any type, then `List<T>` denotes the type whose values are finite lists of zero or more elements of type T.

4. *Array.* If T is a type and i is an integer, then `Array<T,i>` denotes the type whose elements are arrays of i elements of type T. For example, `Array<char,10>` denotes character strings of length 10.

5. *Dictionary.* If T and S are types, then `Dictionary<T,S>` denotes a type whose values are finite sets of pairs. Each pair consists of a value of the *key type* T and a value of the *range type* S. The dictionary may not contain two pairs with the same key value.

Sets, Bags, and Lists

To understand the distinction between sets, bags, and lists, remember that a set has unordered elements, and only one occurrence of each element. A bag allows more than one occurrence of an element, but the elements and their occurrences are unordered. A list allows more than one occurrence of an element, but the occurrences are ordered. Thus, $\{1, 2, 1\}$ and $\{2, 1, 1\}$ are the same bag, but $(1, 2, 1)$ and $(2, 1, 1)$ are not the same list.

6. *Structures.* If $T_1, T_2, \ldots, T_n$ are types, and $F_1, F_2, \ldots, F_n$ are names of fields, then

   `Struct N {`T_1 F_1`,` T_2 F_2`,...,` T_n F_n`}`

   denotes the type named N whose elements are structures with n fields. The ith field is named F_i and has type T_i. For example, line (10) of Fig. 45 showed a structure type named `Addr`, with two fields. Both fields are of type `string` and have names `street` and `city`, respectively.

The first five types — set, bag, list, array, and dictionary — are called *collection types*. There are different rules about which types may be associated with attributes and which with relationships.

- The type of a relationship is either a class type or a single use of a collection type constructor applied to a class type.
- The type of an attribute is built starting with a primitive type or types.[7] We may then apply the structure and collection type constructors as we wish, as many times as we wish.

Example 51: Some of the possible types of attributes are:

1. `integer`.
2. `Struct N {string field1, integer field2}`.
3. `List<real>`.
4. `Array<Struct N {string field1, integer field2}, 10>`.

[7] Class types may also be used, which makes the attribute behave like a "one-way" relationship. We shall not consider such attributes here.

Example (1) is a primitive type, (2) is a structure of primitive types, (3) a collection of a primitive type, and (4) a collection of structures built from primitive types.

Now, suppose the class names `Movie` and `Star` are available primitive types. Then we may construct relationship types such as `Movie` or `Bag<Star>`. However, the following are illegal as relationship types:

1. `Struct N {Movie field1, Star field2}`. Relationship types cannot involve structures.
2. `Set<integer>`. Relationship types cannot involve primitive types.
3. `Set<Array<Star, 10>>`. Relationship types cannot involve two applications of collection types.

□

9.7 Subclasses in ODL

We can declare one class C to be a subclass of another class D. To do so, follow the name C in its declaration with the keyword `extends` and the name D. Then, class C inherits all the properties of D, and may have additional properties of its own.

Example 52: Recall Example 10, where we declared cartoons to be a subclass of movies, with the additional property of a relationship from a cartoon to a set of stars that are its "voices." We can create a subclass `Cartoon` for Movie with the ODL declaration:

```
class Cartoon extends Movie {
   relationship Set<Star> voices;
};
```

Also in that example, we defined a class of murder mysteries with additional attribute `weapon`.

```
class MurderMystery extends Movie {
   attribute string weapon;
};
```

is a suitable declaration of this subclass. □

Sometimes, as in the case of a movie like "Roger Rabbit," we need a class that is a subclass of two or more other classes at the same time. In ODL, we may follow the keyword `extends` by several classes, separated by colons.[8] Thus, we may declare a fourth class by:

[8] Technically, the second and subsequent names must be "interfaces," rather than classes. Roughly, an *interface* in ODL is a class definition without an associated set of objects.

```
class CartoonMurderMystery
    extends MurderMystery : Cartoon;
```

Note that when there is multiple inheritance, there is the potential for a class to inherit two properties with the same name. The way such conflicts are resolved is implementation-dependent.

9.8 Declaring Keys in ODL

The declaration of a key or keys for a class is optional. The reason is that ODL, being object-oriented, assumes that all objects have an object-identity, as discussed in connection with UML in Section 8.4.

In ODL we may declare one or more attributes to be a key for a class by using the keyword `key` or `keys` (it doesn't matter which) followed by the attribute or attributes forming keys. If there is more than one attribute in a key, the list of attributes must be surrounded by parentheses. The key declaration itself appears inside parentheses, following the name of the class itself in the first line of its declaration.

Example 53: To declare that the set of two attributes `title` and `year` form a key for class `Movie`, we could begin its declaration:

```
class Movie (key (title, year)) {
```

We could have used `keys` in place of `key`, even though only one key is declared. □

It is possible that several sets of attributes are keys. If so, then following the word `key(s)` we may place several keys separated by commas. A key that consists of more than one attribute must have parentheses around the list of its attributes, so we can disambiguate a key of several attributes from several keys of one attribute each.

The ODL standard also allows properties other than attributes to appear in keys. There is no fundamental problem with a method or relationship being declared a key or part of a key, since keys are advisory statements that the DBMS can take advantage of or not, as it wishes. For instance, one could declare a method to be a key, meaning that on distinct objects of the class the method is guaranteed to return distinct values.

When we allow many-one relationships to appear in key declarations, we can get an effect similar to that of weak entity sets in the E/R model. We can declare that the object O_1 referred to by an object O_2 on the "many" side of the relationship, perhaps together with other properties of O_2 that are included in the key, is unique for different objects O_2. However, we should remember that there is no requirement that classes have keys; we are never obliged to handle, in some special way, classes that lack attributes of their own to form a key, as we did for weak entity sets.

Example 54: Let us review the example of a weak entity set *Crews* in Fig. 20. Recall that we hypothesized that crews were identified by their number, and the studio for which they worked, although two studios might have crews with the same number. We might declare the class `Crew` as in Fig. 46. Note that we should modify the declaration of `Studio` to include the relationship `crewsOf` that is an inverse to the relationship `unitOf` in `Crew`; we omit this change.

```
class Crew (key (number, unitOf)) {
    attribute integer number;
    attribute string crewChief;
    relationship Studio unitOf
        inverse Studio::crewsOf;
};
```

Figure 46: A ODL declaration for crews

What this key declaration asserts is that there cannot be two crews that both have the same value for the `number` attribute and are related to the same studio by `unitOf`. Notice how this assertion resembles the implication of the E/R diagram in Fig. 20, which is that the number of a crew and the name of the related studio (i.e., the key for studios) uniquely determine a crew entity. □

9.9 Exercises for Section 9

Exercise 9.1: In Exercise 1.1 was the informal description of a bank database. Render this design in ODL, including keys as appropriate.

Exercise 9.2: Modify your design of Exercise 9.1 in the ways enumerated in Exercise 1.2. Describe the changes; do not write a complete, new schema.

Exercise 9.3: Render the teams-players-fans database of Exercise 1.3 in ODL, including keys, as appropriate. Why does the complication about sets of team colors, which was mentioned in the original exercise, not present a problem in ODL?

! Exercise 9.4: Suppose we wish to keep a genealogy. We shall have one class, `Person`. The information we wish to record about persons includes their name (an attribute) and the following relationships: mother, father, and children. Give an ODL design for the `Person` class. Be sure to indicate the inverses of the relationships that, like `mother`, `father`, and `children`, are also relationships from `Person` to itself. Is the inverse of the `mother` relationship the `children` relationship? Why or why not? Describe each of the relationships and their inverses as sets of pairs.

! Exercise 9.5: Let us add to the design of Exercise 9.4 the attribute `education`. The value of this attribute is intended to be a collection of the degrees obtained by each person, including the name of the degree (e.g., B.S.), the school, and the date. This collection of structs could be a set, bag, list, or array. Describe the consequences of each of these four choices. What information could be gained or lost by making each choice? Is the information lost likely to be important in practice?

Exercise 9.6: In Exercise 4.4 we saw two examples of situations where weak entity sets were essential. Render these databases in ODL, including declarations for suitable keys.

Exercise 9.7: Give an ODL design for the registrar's database described in Exercise 1.9.

!! Exercise 9.8: Under what circumstances is a relationship its own inverse? *Hint*: Think about the relationship as a set of pairs, as discussed in Section 9.4.

10 From ODL Designs to Relational Designs

ODL was actually intended as the data-definition part of a language standard for object-oriented DBMS's, analogous to the SQL `CREATE TABLE` statement. Indeed, there have been some attempts to implement such a system. However, it is also possible to see ODL as a text-based, high-level design notation, from which we eventually derive a relational database schema. Thus, in this section we shall consider how to convert ODL designs into relational designs.

Much of the process is similar to that we discussed for E/R diagrams in Section 5 and for UML in Section 8. Classes become relations, and relationships become relations that connect the key attributes of the classes involved in the relationship. Yet some new problems arise for ODL, including:

1. Entity sets must have keys, but there is no such guarantee for ODL classes.
2. While attributes in E/R, UML, and the relational model are of primitive type, there is no such constraint for ODL attributes.

10.1 From ODL Classes to Relations

As a starting point, let us assume that our goal is to have one relation for each class and for that relation to have one attribute for each property. We shall see many ways in which this approach must be modified, but for the moment, let us consider the simplest possible case, where we can indeed convert classes to relations and properties to attributes. The restrictions we assume are:

1. All properties of the class are attributes (not relationships or methods).

2. The types of the attributes are primitive (not structures or sets).

In this case, the ODL class looks almost like an entity set or a UML class. Although there might be no key for the ODL class, ODL assumes object-identity. We can create an artificial attribute to represent the object-identity and serve as a key for the relation; this issue was introduced for UML in Section 8.4.

Example 55: Figure 47 is an ODL description of movie executives. No key is listed, and we do not assume that `name` uniquely determines a movie executive (unlike stars, who will make sure their chosen name is unique).

```
class MovieExec {
   attribute string name;
   attribute string address;
   attribute integer netWorth;
};
```

Figure 47: The class `MovieExec`

We create a relation with the same name as the class. The relation has four attributes, one for each attribute of the class, and one for the object-identity:

```
MovieExecs(cert#, name, address, netWorth)
```

We use `cert#` as the key attribute, representing the object-identity. □

10.2 Complex Attributes in Classes

Even when a class' properties are all attributes we may have some difficulty converting the class to a relation. The reason is that attributes in ODL can have complex types such as structures, sets, bags, or lists. On the other hand, a fundamental principle of the relational model is that a relation's attributes have a primitive type, such as numbers and strings. Thus, we must find some way to represent complex attribute types as relations.

Record structures whose fields are themselves primitive are the easiest to handle. We simply expand the structure definition, making one attribute of the relation for each field of the structure.

```
class Star (key name) {
   attribute string name;
   attribute Struct Addr
       {string street, string city} address;
};
```

Figure 48: Class with a structured attribute

Example 56: In Fig. 48 is a declaration for class `Star`, with only attributes as properties. The attribute `name` is of primitive type, but attribute `address` is a structure with two fields, `street` and `city`. We represent this class by the relation:

```
Star(name, street, city)
```

The key is `name`, and the attributes `street` and `city` represent the structure `address`. □

10.3 Representing Set-Valued Attributes

However, record structures are not the most complex kind of attribute that can appear in ODL class definitions. Values can also be built using type constructors `Set`, `Bag`, `List`, `Array`, and `Dictionary` from Section 9.6. Each presents its own problems when migrating to the relational model. We shall only discuss the `Set` constructor, which is the most common, in detail.

One approach to representing a set of values for an attribute A is to make one tuple for each value. That tuple includes the appropriate values for all the other attributes besides A. This approach works, although it is likely to produce unnormalized relations, as we shall see in the next example.

```
class Star (key name) {
   attribute string name;
   attribute Set<
           Struct Addr {string street, string city}
       > address;
   attribute Date birthdate;
};
```

Figure 49: Stars with a set of addresses and a birthdate

Example 57: Figure 49 shows a new definition of the class `Star`, in which we have allowed stars to have a set of addresses and also added a nonkey, primitive attribute `birthdate`. The `birthdate` attribute can be an attribute of the `Star` relation, whose schema now becomes:

```
Star(name, street, city, birthdate)
```

Unfortunately, this relation exhibits some anomalies. If Carrie Fisher has two addresses, say a home and a beach house, then she is represented by two tuples in the relation `Star`. If Harrison Ford has an empty set of addresses, then he does not appear at all in `Star`. A typical set of tuples for `Star` is shown in Fig. 50.

name	*street*	*city*	*birthdate*
Carrie Fisher	123 Maple St.	Hollywood	9/9/99
Carrie Fisher	5 Locust Ln.	Malibu	9/9/99
Mark Hamill	456 Oak Rd.	Brentwood	8/8/88

Figure 50: Adding birthdates

Although `name` is a key for the class `Star`, our need to have several tuples for one star to represent all their addresses means that `name` is *not* a key for the relation `Star`. In fact, the key for that relation is {`name, street, city`}. Thus, the functional dependency

$$\texttt{name} \rightarrow \texttt{birthdate}$$

is a BCNF violation and the multivalued dependency

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

is a 4NF violation as well. □

There are several options regarding how to handle set-valued attributes that appear in a class declaration along with other attributes, set-valued or not. One approach is to separate out each set-valued attribute as if it were a many-many relationship between the objects of the class and the values that appear in the sets.

An alternative approach is to place all attributes, set-valued or not, in the schema for the relation, then use normalization techniques to eliminate the resulting BCNF and 4NF violations. Notice that any set-valued attribute in conjunction with any single-valued attribute leads to a BNCF violation, as in Example 57. Two set-valued attributes in the same class declaration will lead to a 4NF violation, even if there are no single-valued attributes.

10.4 Representing Other Type Constructors

Besides record structures and sets, an ODL class definition could use `Bag`, `List`, `Array`, or `Dictionary` to construct values. To represent a bag (multiset), in which a single object can be a member of the bag n times, we cannot simply introduce into a relation n identical tuples.[9] Instead, we could add to the relation schema another attribute `count` representing the number of times that

[9]To be precise, we cannot introduce identical tuples into relations of the abstract relational model. However, SQL-based relational DBMS's *do* allow duplicate tuples; i.e., relations are bags rather than sets in SQL. If queries are likely to ask for tuple counts, we advise using a scheme such as that described here, even if your DBMS allows duplicate tuples.

each element is a member of the bag. For instance, suppose that `address` in Fig. 49 were a bag instead of a set. We could say that 123 Maple St., Hollywood is Carrie Fisher's address twice and 5 Locust Ln., Malibu is her address 3 times (whatever that may mean) by

name	*street*	*city*	*count*
Carrie Fisher	123 Maple St.	Hollywood	2
Carrie Fisher	5 Locust Ln.	Malibu	3

A list of addresses could be represented by a new attribute `position`, indicating the position in the list. For instance, we could show Carrie Fisher's addresses as a list, with Hollywood first, by:

name	*street*	*city*	*position*
Carrie Fisher	123 Maple St.	Hollywood	1
Carrie Fisher	5 Locust Ln.	Malibu	2

A fixed-length array of addresses could be represented by attributes for each position in the array. For instance, if `address` were to be an array of two street-city structures, we could represent `Star` objects as:

name	*street1*	*city1*	*street2*	*city2*
Carrie Fisher	123 Maple St.	Hollywood	5 Locust Ln.	Malibu

Finally, a dictionary could be represented as a set, but with attributes for both the key-value and range-value components of the pairs that are members of the dictionary. For instance, suppose that instead of star's addresses, we really wanted to keep, for each star, a dictionary giving the mortgage holder for each of their homes. Then the dictionary would have address as the key value and bank name as the range value. A hypothetical rendering of the Carrie-Fisher object with a dictionary attribute is:

name	*street*	*city*	*mortgage-holder*
Carrie Fisher	123 Maple St.	Hollywood	Bank of Burbank
Carrie Fisher	5 Locust Ln.	Malibu	Torrance Trust

Of course attribute types in ODL may involve more than one type constructor. If a type is any collection type besides dictionary applied to a structure (e.g., a set of structs), then we may apply the techniques from Sections 10.3 or 10.4 as if the struct were an atomic value, and then replace the single attribute representing the atomic value by several attributes, one for each field of the struct. This strategy was used in the examples above, where the address is a struct. The case of a dictionary applied to structs is similar and left as an exercise.

There are many reasons to limit the complexity of attribute types to an optional struct followed by an optional collection type. We mentioned in Section 1.1 that some versions of the E/R model allow exactly this much generality in the types of attributes, although we restricted ourselves to attributes of

primitive type in the E/R model. We recommend that, if you are going to use an ODL design for the purpose of eventual translation to a relational database schema, you similarly limit yourself. We take up in the exercises some options for dealing with more complex types as attributes.

10.5 Representing ODL Relationships

Usually, an ODL class definition will contain relationships to other ODL classes. As in the E/R model, we can create for each relationship a new relation that connects the keys of the two related classes. However, in ODL, relationships come in inverse pairs, and we must create only one relation for each pair.

When a relationship is many-one, we have an option to combine it with the relation that is constructed for the class on the "many" side. Doing so has the effect of combining two relations that have a common key, as we discussed in Section 5.3. It therefore does not cause a BCNF violation and is a legitimate and commonly followed option.

10.6 Exercises for Section 10

Exercise 10.1: Convert your ODL designs from the following exercises to relational database schemas.

a) Exercise 9.1.

b) Exercise 9.2 (include all four of the modifications specified by that exercise).

c) Exercise 9.3.

d) Exercise 9.4.

e) Exercise 9.5.

! Exercise 10.2: Consider an attribute of type `Dictionary` with key and range types both structs of primitive types. Show how to convert a class with an attribute of this type to a relation.

Exercise 10.3: We mentioned that when attributes are of a type more complex than a collection of structs, it becomes tricky to convert them to relations; in particular, it becomes necessary to create some intermediate concepts and relations for them. The following sequence of questions will examine increasingly more complex types and how to represent them as relations.

a) A *card* can be represented as a struct with fields `rank` ($2, 3, \ldots, 10$, Jack, Queen, King, and Ace) and `suit` (Clubs, Diamonds, Hearts, and Spades). Give a suitable definition of a structured type `Card`. This definition should be independent of any class declarations but available to them all.

b) A *hand* is a set of cards. The number of cards may vary. Give a declaration of a class `Hand` whose objects are hands. That is, this class declaration has an attribute `theHand`, whose type is a hand.

! c) Convert your class declaration `Hand` from (b) to a relation schema.

d) A *poker hand* is a set of five cards. Repeat (b) and (c) for poker hands.

! e) A *deal* is a set of pairs, each pair consisting of the name of a player and a hand for that player. Declare a class `Deal`, whose objects are deals. That is, this class declaration has an attribute `theDeal`, whose type is a deal.

f) Repeat (e), but restrict hands of a deal to be hands of exactly five cards.

g) Repeat (e), using a dictionary for a deal. You may assume the names of players in a deal are unique.

!! h) Convert your class declaration from (e) to a relational database schema.

! i) Suppose we defined deals to be sets of sets of cards, with no player associated with each hand (set of cards). It is proposed that we represent such deals by a relation schema `Deals(dealID, card)`, meaning that the card was a member of one of the hands in the deal with the given ID. What, if anything, is wrong with this representation? How would you fix the problem?

Exercise 10.4: Suppose we have a class C defined by

```
class C (key a) {
    attribute string a;
    attribute T b;
};
```

where T is some type. Give the relation schema for the relation derived from C and indicate its key attributes if T is:

a) `Set<Struct S {string f, string g}>`

! b) `Bag<Struct S {string f, string g}>`

! c) `List<Struct S {string f, string }>`

! d) `Dictionary<Struct K {string f, string g}, Struct R {string i, string j}>`

11 Summary

- ✦ *The Entity-Relationship Model*: In the E/R model we describe entity sets, relationships among entity sets, and attributes of entity sets and relationships. Members of entity sets are called entities.
- ✦ *Entity-Relationship Diagrams*: We use rectangles, diamonds, and ovals to draw entity sets, relationships, and attributes, respectively.
- ✦ *Multiplicity of Relationships*: Binary relationships can be one-one, many-one, or many-many. In a one-one relationship, an entity of either set can be associated with at most one entity of the other set. In a many-one relationship, each entity of the "many" side is associated with at most one entity of the other side. Many-many relationships place no restriction.
- ✦ *Good Design*: Designing databases effectively requires that we represent the real world faithfully, that we select appropriate elements (e.g., relationships, attributes), and that we avoid redundancy — saying the same thing twice or saying something in an indirect or overly complex manner.
- ✦ *Subclasses*: The E/R model uses a special relationship *isa* to represent the fact that one entity set is a special case of another. Entity sets may be connected in a hierarchy with each child node a special case of its parent. Entities may have components belonging to any subtree of the hierarchy, as long as the subtree includes the root.
- ✦ *Weak Entity Sets*: These require attributes of some supporting entity set(s) to identify their own entities. A special notation involving diamonds and rectangles with double borders is used to distinguish weak entity sets.
- ✦ *Converting Entity Sets to Relations*: The relation for an entity set has one attribute for each attribute of the entity set. An exception is a weak entity set E, whose relation must also have attributes for the key attributes of its supporting entity sets.
- ✦ *Converting Relationships to Relations*: The relation for an E/R relationship has attributes corresponding to the key attributes of each entity set that participates in the relationship. However, if a relationship is a supporting relationship for some weak entity set, it is not necessary to produce a relation for that relationship.
- ✦ *Converting Isa Hierarchies to Relations*: One approach is to create a relation for each entity set with the key attributes of the hierarchy's root plus the attributes of the entity set itself. A second approach is to create a relation for each possible subset of the entity sets in the hierarchy, and create for each entity one tuple; that tuple is in the relation for exactly the set of entity sets to which the entity belongs. A third approach is to create only one relation and to use null values for those attributes that do not apply to the entity represented by a given tuple.

- *Unified Modeling Language*: In UML, we describe classes and associations between classes. Classes are analogous to E/R entity sets, and associations are like binary E/R relationships. Special kinds of many-one associations, called aggregations and compositions, are used and have implications as to how they are translated to relations.

- *UML Subclass Hierarchies*: UML permits classes to have subclasses, with inheritance from the superclass. The subclasses of a class can be complete or partial, and they can be disjoint or overlapping.

- *Converting UML Diagrams to Relations*: The methods are similar to those used for the E/R model. Classes become relations and associations become relations connecting the keys of the associated classes. Aggregations and compositions are combined with the relation constructed from the class at the "many" end.

- *Object Definition Language*: This language is a notation for formally describing the schemas of databases in an object-oriented style. One defines classes, which may have three kinds of properties: attributes, methods, and relationships.

- *ODL Relationships*: A relationship in ODL must be binary. It is represented, in the two classes it connects, by names that are declared to be inverses of one another. Relationships can be many-many, many-one, or one-one, depending on whether the types of the pair are declared to be a single object or a set of objects.

- *The ODL Type System*: ODL allows types to be constructed, beginning with class names and atomic types such as integer, by applying any of the following type constructors: structure formation, set-of, bag-of, list-of, array-of, and dictionary-of.

- *Keys in ODL*: Keys are optional in ODL. We can declare one or more keys, but because objects have an object-ID that is not one of its properties, a system implementing ODL can tell the difference between objects, even if they have identical values for all properties.

- *Converting ODL Classes to Relations*: The method is the same as for E/R or UML, except if the class has attributes of complex type. If that happens the resulting relation may be unnormalized and will have to be decomposed. It may also be necessary to create a new attribute to represent the object-identity of objects and serve as a key.

- *Converting ODL Relationships to Relations*: The method is the same as for E/R relationships, except that we must first pair ODL relationships and their inverses, and create only one relation for the pair.

12 References

The original paper on the Entity-Relationship model is [5]. Two books on the subject of E/R design are [2] and [7].

The manual defining ODL is [4]. One can also find more about the history of object-oriented database systems from [1], [3], and [6].

1. F. Bancilhon, C. Delobel, and P. Kanellakis, *Building an Object-Oriented Database System*, Morgan-Kaufmann, San Francisco, 1992.

2. Carlo Batini, S. Ceri, S. B. Navathe, and Carol Batini, *Conceptual Database Design: an Entity/Relationship Approach*, Addison-Wesley, Boston MA, 1991.

3. R. G. G. Cattell, *Object Data Management*, Addison-Wesley, Reading, MA, 1994.

4. R. G. G. Cattell (ed.), *The Object Database Standard: ODMG–99*, Morgan-Kaufmann, San Francisco, 1999.

5. P. P. Chen, "The entity-relationship model: toward a unified view of data," *ACM Trans. on Database Systems* **1**:1, pp. 9–36, 1976.

6. W. Kim (ed.), *Modern Database Systems: The Object Model, Interoperability, and Beyond*, ACM Press, New York, 1994.

7. B. Thalheim, "Fundamentals of Entity-Relationship Modeling," Springer-Verlag, Berlin, 2000.

Algebraic and Logical Query Languages

We now switch our attention from modeling to programming for relational databases. We start in this discussion with two abstract programming languages, one algebraic and the other logic-based. The algebraic programming language, relational algebra, let us see what operations in the relational model look like. However, there is more to the algebra. In this chapter, we extend set-based algebra to bags, which better reflect the way the relational model is implemented in practice. We also extend the algebra so it can handle several more operations than were described previously; for example, we need to do aggregations (e.g., averages) of columns of a relation.

We close the chapter with another form of query language, based on logic. This language, called "Datalog," allows us to express queries by describing the desired results, rather than by giving an algorithm to compute the results, as relational algebra requires.

1 Relational Operations on Bags

In this section, we shall consider relations that are bags (multisets) rather than sets. That is, we shall allow the same tuple to appear more than once in a relation. When relations are bags, there are changes that need to be made to the definition of some relational operations, as we shall see. First, let us look at a simple example of a relation that is a bag but not a set.

Example 1: The relation in Fig. 1 is a bag of tuples. In it, the tuple $(1, 2)$ appears three times and the tuple $(3, 4)$ appears once. If Fig. 1 were a set-valued relation, we would have to eliminate two occurrences of the tuple $(1, 2)$. In a bag-valued relation, we *do* allow multiple occurrences of the same tuple, but like sets, the order of tuples does not matter. □

A	B
1	2
3	4
1	2
1	2

Figure 1: A bag

1.1 Why Bags?

As we mentioned, commercial DBMS's implement relations that are bags, rather than sets. An important motivation for relations as bags is that some relational operations are considerably more efficient if we use the bag model. For example:

1. To take the union of two relations as bags, we simply copy one relation and add to the copy all the tuples of the other relation. There is no need to eliminate duplicate copies of a tuple that happens to be in both relations.

2. When we project relation as sets, we need to compare each projected tuple with all the other projected tuples, to make sure that each projection appears only once. However, if we can accept a bag as the result, then we simply project each tuple and add it to the result; no comparison with other projected tuples is necessary.

A	B	C
1	2	5
3	4	6
1	2	7
1	2	8

Figure 2: Bag for Example 2

Example 2: The bag of Fig. 1 could be the result of projecting the relation shown in Fig. 2 onto attributes A and B, provided we allow the result to be a bag and do not eliminate the duplicate occurrences of $(1, 2)$. Had we used the ordinary projection operator of relational algebra, and therefore eliminated duplicates, the result would be only:

A	B
1	2
3	4

Note that the bag result, although larger, can be computed more quickly, since there is no need to compare each tuple $(1, 2)$ or $(3, 4)$ with previously generated tuples. □

Another motivation for relations as bags is that there are some situations where the expected answer can only be obtained if we use bags, at least temporarily. Here is an example.

Example 3: Suppose we want to take the average of the A-components of a set-valued relation such as Fig. 2. We could not use the set model to think of the relation projected onto attribute A. As a set, the average value of A is 2, because there are only two values of A — 1 and 3 — in Fig. 2, and their average is 2. However, if we treat the A-column in Fig. 2 as a bag $\{1, 3, 1, 1\}$, we get the correct average of A, which is 1.5, among the four tuples of Fig. 2. □

1.2 Union, Intersection, and Difference of Bags

These three operations have new definitions for bags. Suppose that R and S are bags, and that tuple t appears n times in R and m times in S. Note that either n or m (or both) can be 0. Then:

- In the bag union $R \cup S$, tuple t appears $n + m$ times.
- In the bag intersection $R \cap S$, tuple t appears $\min(n, m)$ times.
- In the bag difference $R - S$, tuple t appears $\max(0, n - m)$ times. That is, if tuple t appears in R more times than it appears in S, then t appears in $R - S$ the number of times it appears in R, minus the number of times it appears in S. However, if t appears at least as many times in S as it appears in R, then t does not appear at all in $R - S$. Intuitively, occurrences of t in S each "cancel" one occurrence in R.

Example 4: Let R be the relation of Fig. 1, that is, a bag in which tuple $(1, 2)$ appears three times and $(3, 4)$ appears once. Let S be the bag

A	B
1	2
3	4
3	4
5	6

Then the bag union $R \cup S$ is the bag in which $(1, 2)$ appears four times (three times for its occurrences in R and once for its occurrence in S); $(3, 4)$ appears three times, and $(5, 6)$ appears once.

The bag intersection $R \cap S$ is the bag

A	B
1	2
3	4

with one occurrence each of $(1, 2)$ and $(3, 4)$. That is, $(1, 2)$ appears three times in R and once in S, and $\min(3, 1) = 1$, so $(1, 2)$ appears once in $R \cap S$. Similarly, $(3, 4)$ appears $\min(1, 2) = 1$ time in $R \cap S$. Tuple $(5, 6)$, which appears once in S but zero times in R appears $\min(0, 1) = 0$ times in $R \cap S$. In this case, the result happens to be a set, but any set is also a bag.

The bag difference $R - S$ is the bag

A	B
1	2
1	2

To see why, notice that $(1, 2)$ appears three times in R and once in S, so in $R - S$ it appears $\max(0, 3 - 1) = 2$ times. Tuple $(3, 4)$ appears once in R and twice in S, so in $R - S$ it appears $\max(0, 1 - 2) = 0$ times. No other tuple appears in R, so there can be no other tuples in $R - S$.

As another example, the bag difference $S - R$ is the bag

A	B
3	4
5	6

Tuple $(3, 4)$ appears once because that is the number of times it appears in S minus the number of times it appears in R. Tuple $(5, 6)$ appears once in $S - R$ for the same reason. □

1.3 Projection of Bags

We have already illustrated the projection of bags. As we saw in Example 2, each tuple is processed independently during the projection. If R is the bag of Fig. 2 and we compute the bag-projection $\pi_{A,B}(R)$, then we get the bag of Fig. 1.

If the elimination of one or more attributes during the projection causes the same tuple to be created from several tuples, these duplicate tuples are not eliminated from the result of a bag-projection. Thus, the three tuples $(1, 2, 5)$, $(1, 2, 7)$, and $(1, 2, 8)$ of the relation R from Fig. 2 each gave rise to the same tuple $(1, 2)$ after projection onto attributes A and B. In the bag result, there are three occurrences of tuple $(1, 2)$, while in the set-projection, this tuple appears only once.

Bag Operations on Sets

Imagine we have two sets R and S. Every set may be thought of as a bag; the bag just happens to have at most one occurrence of any tuple. Suppose we intersect $R \cap S$, but we think of R and S as bags and use the bag intersection rule. Then we get the same result as we would get if we thought of R and S as sets. That is, thinking of R and S as bags, a tuple t is in $R \cap S$ the minimum of the number of times it is in R and S. Since R and S are sets, t can be in each only 0 or 1 times. Whether we use the bag or set intersection rules, we find that t can appear at most once in $R \cap S$, and it appears once exactly when it is in both R and S. Similarly, if we use the bag difference rule to compute $R - S$ or $S - R$ we get exactly the same result as if we used the set rule.

However, union behaves differently, depending on whether we think of R and S as sets or bags. If we use the bag rule to compute $R \cup S$, then the result may not be a set, even if R and S are sets. In particular, if tuple t appears in both R and S, then t appears twice in $R \cup S$ if we use the bag rule for union. But if we use the set rule then t appears only once in $R \cup S$.

1.4 Selection on Bags

To apply a selection to a bag, we apply the selection condition to each tuple independently. As always with bags, we do not eliminate duplicate tuples in the result.

Example 5: If R is the bag

A	B	C
1	2	5
3	4	6
1	2	7
1	2	7

then the result of the bag-selection $\sigma_{C \geq 6}(R)$ is

A	B	C
3	4	6
1	2	7
1	2	7

That is, all but the first tuple meets the selection condition. The last two tuples, which are duplicates in R, are each included in the result. □

Algebraic Laws for Bags

An algebraic law is an equivalence between two expressions of relational algebra whose arguments are variables standing for relations. The equivalence asserts that no matter what relations we substitute for these variables, the two expressions define the same relation. An example of a well-known law is the commutative law for union: $R \cup S = S \cup R$. This law happens to hold whether we regard relation-variables R and S as standing for sets or bags. However, there are a number of other laws that hold when relational algebra is applied to sets but that do not hold when relations are interpreted as bags. A simple example of such a law is the distributive law of set difference over union, $(R \cup S) - T = (R - T) \cup (S - T)$. This law holds for sets but not for bags. To see why it fails for bags, suppose R, S, and T each have one copy of tuple t. Then the expression on the left has one t, while the expression on the right has none. As sets, neither would have t. Some exploration of algebraic laws for bags appears in Exercises 1.4 and 1.5.

1.5 Product of Bags

The rule for the Cartesian product of bags is the expected one. Each tuple of one relation is paired with each tuple of the other, regardless of whether it is a duplicate or not. As a result, if a tuple r appears in a relation R m times, and tuple s appears n times in relation S, then in the product $R \times S$, the tuple rs will appear mn times.

Example 6: Let R and S be the bags shown in Fig. 3. Then the product $R \times S$ consists of six tuples, as shown in Fig. 3(c). Note that the usual convention regarding attribute names that we developed for set-relations applies equally well to bags. Thus, the attribute B, which belongs to both relations R and S, appears twice in the product, each time prefixed by one of the relation names. □

1.6 Joins of Bags

Joining bags presents no surprises. We compare each tuple of one relation with each tuple of the other, decide whether or not this pair of tuples joins successfully, and if so we put the resulting tuple in the answer. When constructing the answer, we do not eliminate duplicate tuples.

Example 7: The natural join $R \bowtie S$ of the relations R and S seen in Fig. 3 is

A	B
1	2
1	2

(a) The relation R

B	C
2	3
4	5
4	5

(b) The relation S

A	$R.B$	$S.B$	C
1	2	2	3
1	2	2	3
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

(c) The product $R \times S$

Figure 3: Computing the product of bags

A	B	C
1	2	3
1	2	3

That is, tuple $(1, 2)$ of R joins with $(2, 3)$ of S. Since there are two copies of $(1, 2)$ in R and one copy of $(2, 3)$ in S, there are two pairs of tuples that join to give the tuple $(1, 2, 3)$. No other tuples from R and S join successfully.

As another example on the same relations R and S, the theta-join

$$R \bowtie_{R.B<S.B} S$$

produces the bag

A	$R.B$	$S.B$	C
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

The computation of the join is as follows. Tuple $(1, 2)$ from R and $(4, 5)$ from S meet the join condition. Since each appears twice in its relation, the number of times the joined tuple appears in the result is 2×2 or 4. The other possible join of tuples — $(1, 2)$ from R with $(2, 3)$ from S — fails to meet the join condition, so this combination does not appear in the result. □

1.7 Exercises for Section 1

Exercise 1.1: Let PC be the relation of Fig. 4, and suppose we compute the projection π_{speed}(PC). What is the value of this expression as a set? As a bag? What is the average value of tuples in this projection, when treated as a set? As a bag?

model	*speed*	*ram*	*hd*	*price*
1001	2.66	1024	250	2114
1002	2.10	512	250	995
1003	1.42	512	80	478
1004	2.80	1024	250	649
1005	3.20	512	250	630
1006	3.20	1024	320	1049
1007	2.20	1024	200	510
1008	2.20	2048	250	770
1009	2.00	1024	250	650
1010	2.80	2048	300	770
1011	1.86	2048	160	959
1012	2.80	1024	160	649
1013	3.06	512	80	529

Figure 4: Sample data for relation PC

Exercise 1.2: Repeat Exercise 1.1 for the projection π_{hd}(PC).

Exercise 1.3: This exercise refers to the following relations:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) The expression π_{bore}(Classes) yields a single-column relation with the bores of the various classes. For the data in Figs. 5 and 6, what is this relation as a set? As a bag?

! b) Write an expression of relational algebra to give the bores of the ships (not the classes). Your expression must make sense for bags; that is, the

class	*type*	*country*	*numGuns*	*bore*	*displacement*
Bismarck	bb	Germany	8	15	42000
Iowa	bb	USA	9	16	46000
Kongo	bc	Japan	8	14	32000
North Carolina	bb	USA	9	16	37000
Renown	bc	Gt. Britain	6	15	32000
Revenge	bb	Gt. Britain	8	15	29000
Tennessee	bb	USA	12	14	32000
Yamato	bb	Japan	9	18	65000

(a) Sample data for relation `Classes`

name	*date*
Denmark Strait	5/24-27/41
Guadalcanal	11/15/42
North Cape	12/26/43
Surigao Strait	10/25/44

(b) Sample data for relation `Battles`

ship	*battle*	*result*
Arizona	Pearl Harbor	sunk
Bismarck	Denmark Strait	sunk
California	Surigao Strait	ok
Duke of York	North Cape	ok
Fuso	Surigao Strait	sunk
Hood	Denmark Strait	sunk
King George V	Denmark Strait	ok
Kirishima	Guadalcanal	sunk
Prince of Wales	Denmark Strait	damaged
Rodney	Denmark Strait	ok
Scharnhorst	North Cape	sunk
South Dakota	Guadalcanal	damaged
Tennessee	Surigao Strait	ok
Washington	Guadalcanal	ok
West Virginia	Surigao Strait	ok
Yamashiro	Surigao Strait	sunk

(c) Sample data for relation `Outcomes`

Figure 5: Data for Exercise 1.3

name	*class*	*launched*
California	Tennessee	1921
Haruna	Kongo	1915
Hiei	Kongo	1914
Iowa	Iowa	1943
Kirishima	Kongo	1915
Kongo	Kongo	1913
Missouri	Iowa	1944
Musashi	Yamato	1942
New Jersey	Iowa	1943
North Carolina	North Carolina	1941
Ramillies	Revenge	1917
Renown	Renown	1916
Repulse	Renown	1916
Resolution	Revenge	1916
Revenge	Revenge	1916
Royal Oak	Revenge	1916
Royal Sovereign	Revenge	1916
Tennessee	Tennessee	1920
Washington	North Carolina	1941
Wisconsin	Iowa	1944
Yamato	Yamato	1941

Figure 6: Sample data for relation Ships

number of times a value b appears must be the number of ships that have bore b.

! Exercise 1.4: Certain algebraic laws for relations as sets also hold for relations as bags. Explain why each of the laws below hold for bags as well as sets.

a) The associative law for union: $(R \cup S) \cup T = R \cup (S \cup T)$.

b) The associative law for intersection: $(R \cap S) \cap T = R \cap (S \cap T)$.

c) The associative law for natural join: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$.

d) The commutative law for union: $(R \cup S) = (S \cup R)$.

e) The commutative law for intersection: $(R \cap S) = (S \cap R)$.

f) The commutative law for natural join: $(R \bowtie S) = (S \bowtie R)$.

g) $\pi_L(R \cup S) = \pi_L(R) \cup \pi_L(S)$. Here, L is an arbitrary list of attributes.

h) The distributive law of union over intersection:

$$R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$$

i) $\sigma_{C \text{ AND } D}(R) = \sigma_C(R) \cap \sigma_D(R)$. Here, C and D are arbitrary conditions about the tuples of R.

!! Exercise 1.5: The following algebraic laws hold for sets but not for bags. Explain why they hold for sets and give counterexamples to show that they do not hold for bags.

a) $(R \cap S) - T = R \cap (S - T)$.

b) The distributive law of intersection over union:

$$R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$$

c) $\sigma_{C \text{ OR } D}(R) = \sigma_C(R) \cup \sigma_D(R)$. Here, C and D are arbitrary conditions about the tuples of R.

2 Extended Operators of Relational Algebra

Section 1 introduced the modifications necessary to treat relations as bags of tuples rather than sets. The ideas of that section, combined with those of classical relational algebra, serve as a foundation for most of modern query languages. However, languages such as SQL have several other operations that have proved quite important in applications. Thus, a full treatment of relational operations must include a number of other operators, which we introduce in this section. The additions:

1. The *duplicate-elimination operator* δ turns a bag into a set by eliminating all but one copy of each tuple.

2. *Aggregation operators*, such as sums or averages, are not operations of relational algebra, but are used by the grouping operator (described next). Aggregation operators apply to attributes (columns) of a relation; e.g., the sum of a column produces the one number that is the sum of all the values in that column.

3. *Grouping* of tuples according to their value in one or more attributes has the effect of partitioning the tuples of a relation into "groups." Aggregation can then be applied to columns within each group, giving us the ability to express a number of queries that are impossible to express in the classical relational algebra. The *grouping operator* γ is an operator that combines the effect of grouping and aggregation.

4. *Extended projection* gives additional power to the operator π. In addition to projecting out some columns, in its generalized form π can perform computations involving the columns of its argument relation to produce new columns.

5. The *sorting operator* τ turns a relation into a list of tuples, sorted according to one or more attributes. This operator should be used judiciously, because some relational-algebra operators do not make sense on lists. We can, however, apply selections or projections to lists and expect the order of elements on the list to be preserved in the output.

6. The *outerjoin* operator is a variant of the join that avoids losing dangling tuples. In the result of the outerjoin, dangling tuples are "padded" with the null value, so the dangling tuples can be represented in the output.

2.1 Duplicate Elimination

Sometimes, we need an operator that converts a bag to a set. For that purpose, we use $\delta(R)$ to return the set consisting of one copy of every tuple that appears one or more times in relation R.

Example 8: If R is the relation

A	B
1	2
3	4
1	2
1	2

from Fig. 1, then $\delta(R)$ is

A	B
1	2
3	4

Note that the tuple $(1, 2)$, which appeared three times in R, appears only once in $\delta(R)$. □

2.2 Aggregation Operators

There are several operators that apply to sets or bags of numbers or strings. These operators are used to summarize or "aggregate" the values in one column of a relation, and thus are referred to as *aggregation* operators. The standard operators of this type are:

1. `SUM` produces the sum of a column with numerical values.

2. `AVG` produces the average of a column with numerical values.

3. `MIN` and `MAX`, applied to a column with numerical values, produces the smallest or largest value, respectively. When applied to a column with character-string values, they produce the lexicographically (alphabetically) first or last value, respectively.
4. `COUNT` produces the number of (not necessarily distinct) values in a column. Equivalently, `COUNT` applied to any attribute of a relation produces the number of tuples of that relation, including duplicates.

Example 9: Consider the relation

A	*B*
1	2
3	4
1	2
1	2

Some examples of aggregations on the attributes of this relation are:

1. `SUM(B)` $= 2 + 4 + 2 + 2 = 10$.
2. `AVG(A)` $= (1 + 3 + 1 + 1)/4 = 1.5$.
3. `MIN(A)` $= 1$.
4. `MAX(B)` $= 4$.
5. `COUNT(A)` $= 4$.

□

2.3 Grouping

Often we do not want simply the average or some other aggregation of an entire column. Rather, we need to consider the tuples of a relation in groups, corresponding to the value of one or more other columns, and we aggregate only within each group. As an example, suppose we wanted to compute the total number of minutes of movies produced by each studio, i.e., a relation such as:

studioName	*sumOfLengths*
Disney	12345
MGM	54321
...	...

Starting with the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

from our example database schema, we must group the tuples according to their value for attribute `studioName`. We must then sum the `length` column within each group. That is, we imagine that the tuples of `Movies` are grouped as suggested in Fig. 7, and we apply the aggregation `SUM(length)` to each group independently.

	studioName	
	Disney Disney Disney	
	MGM MGM	
	○ ○ ○	

Figure 7: A relation with imaginary division into groups

2.4 The Grouping Operator

We shall now introduce an operator that allows us to group a relation and/or aggregate some columns. If there is grouping, then the aggregation is within groups.

The subscript used with the γ operator is a list L of elements, each of which is either:

a) An attribute of the relation R to which the γ is applied; this attribute is one of the attributes by which R will be grouped. This element is said to be a *grouping attribute*.

b) An aggregation operator applied to an attribute of the relation. To provide a name for the attribute corresponding to this aggregation in the result, an arrow and new name are appended to the aggregation. The underlying attribute is said to be an *aggregated attribute*.

The relation returned by the expression $\gamma_L(R)$ is constructed as follows:

1. Partition the tuples of R into *groups*. Each group consists of all tuples having one particular assignment of values to the grouping attributes in the list L. If there are no grouping attributes, the entire relation R is one group.

2. For each group, produce one tuple consisting of:

   i. The grouping attributes' values for that group and

   ii. The aggregations, over all tuples of that group, for the aggregated attributes on list L.

Example 10: Suppose we have the relation

```
StarsIn(title, year, starName)
```

δ is a Special Case of γ

Technically, the δ operator is redundant. If $R(A_1, A_2, \ldots, A_n)$ is a relation, then $\delta(R)$ is equivalent to $\gamma_{A_1,A_2,\ldots,A_n}(R)$. That is, to eliminate duplicates, we group on all the attributes of the relation and do no aggregation. Then each group corresponds to a tuple that is found one or more times in R. Since the result of γ contains exactly one tuple from each group, the effect of this "grouping" is to eliminate duplicates. However, because δ is such a common and important operator, we shall continue to consider it separately when we study algebraic laws and algorithms for implementing the operators.

One can also see γ as an extension of the projection operator on sets. That is, $\gamma_{A_1,A_2,\ldots,A_n}(R)$ is also the same as $\pi_{A_1,A_2,\ldots,A_n}(R)$, if R is a set. However, if R is a bag, then γ eliminates duplicates while π does not.

and we wish to find, for each star who has appeared in at least three movies, the earliest year in which they appeared. The first step is to group, using `starName` as a grouping attribute. We clearly must compute for each group the `MIN(year)` aggregate. However, in order to decide which groups satisfy the condition that the star appears in at least three movies, we must also compute the `COUNT(title)` aggregate for each group.

We begin with the grouping expression

$$\gamma_{starName,\ \texttt{MIN}(year)\rightarrow minYear,\ \texttt{COUNT}(title)\rightarrow ctTitle}(\texttt{StarsIn})$$

The first two columns of the result of this expression are needed for the query result. The third column is an auxiliary attribute, which we have named `ctTitle`; it is needed to determine whether a star has appeared in at least three movies. That is, we continue the algebraic expression for the query by selecting for `ctTitle >= 3` and then projecting onto the first two columns. An expression tree for the query is shown in Fig. 8. □

2.5 Extending the Projection Operator

Let us reconsider the projection operator $\pi_L(R)$. In the classical relational algebra, L is a list of (some of the) attributes of R. We extend the projection operator to allow it to compute with components of tuples as well as choose components. In *extended projection*, also denoted $\pi_L(R)$, projection lists can have the following kinds of elements:

1. A single attribute of R.

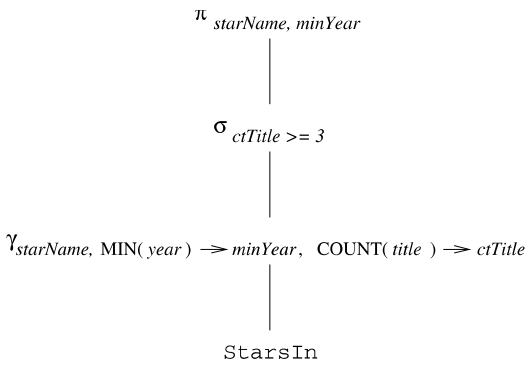

Figure 8: Algebraic expression tree for the query of Example 10

2. An expression $x \to y$, where x and y are names for attributes. The element $x \to y$ in the list L asks that we take the attribute x of R and *rename* it y; i.e., the name of this attribute in the schema of the result relation is y.

3. An expression $E \to z$, where E is an expression involving attributes of R, constants, arithmetic operators, and string operators, and z is a new name for the attribute that results from the calculation implied by E. For example, $a+b \to x$ as a list element represents the sum of the attributes a and b, renamed x. Element $c \,||\, d \to e$ means concatenate the presumably string-valued attributes c and d and call the result e.

The result of the projection is computed by considering each tuple of R in turn. We evaluate the list L by substituting the tuple's components for the corresponding attributes mentioned in L and applying any operators indicated by L to these values. The result is a relation whose schema is the names of the attributes on list L, with whatever renaming the list specifies. Each tuple of R yields one tuple of the result. Duplicate tuples in R surely yield duplicate tuples in the result, but the result can have duplicates even if R does not.

Example 11: Let R be the relation

A	B	C
0	1	2
0	1	2
3	4	5

Then the result of $\pi_{A,B+C \to X}(R)$ is

A	X
0	3
0	3
3	9

The result's schema has two attributes. One is A, the first attribute of R, not renamed. The second is the sum of the second and third attributes of R, with the name X.

For another example, $\pi_{B-A\to X,C-B\to Y}(R)$ is

X	Y
1	1
1	1
1	1

Notice that the calculation required by this projection list happens to turn different tuples $(0, 1, 2)$ and $(3, 4, 5)$ into the same tuple $(1, 1)$. Thus, the latter tuple appears three times in the result. □

2.6 The Sorting Operator

There are several contexts in which we want to sort the tuples of a relation by one or more of its attributes. Often, when querying data, one wants the result relation to be sorted. For instance, in a query about all the movies in which Sean Connery appeared, we might wish to have the list sorted by title, so we could more easily find whether a certain movie was on the list. We shall also see when we study query optimization how execution of queries by the DBMS is often made more efficient if we sort the relations first.

The expression $\tau_L(R)$, where R is a relation and L a list of some of R's attributes, is the relation R, but with the tuples of R sorted in the order indicated by L. If L is the list $A_1, A_2, \ldots, A_n$, then the tuples of R are sorted first by their value of attribute A_1. Ties are broken according to the value of A_2; tuples that agree on both A_1 and A_2 are ordered according to their value of A_3, and so on. Ties that remain after attribute A_n is considered may be ordered arbitrarily.

Example 12: If R is a relation with schema $R(A, B, C)$, then $\tau_{C,B}(R)$ orders the tuples of R by their value of C, and tuples with the same C-value are ordered by their B value. Tuples that agree on both B and C may be ordered arbitrarily. □

If we apply another operator such as join to the sorted result of a τ, the sorted order usually becomes meaningless, and the elements on the list should be treated as a bag, not a list. However, bag projections can be made to preserve the order. Also, a selection on a list drops out the tuples that do not satisfy the condition of the selection, but the remaining tuples can be made to appear in their original sorted order.

2.7 Outerjoins

A property of the join operator is that it is possible for certain tuples to be "dangling"; that is, they fail to match any tuple of the other relation in the

common attributes. Dangling tuples do not have any trace in the result of the join, so the join may not represent the data of the original relations completely. In cases where this behavior is undesirable, a variation on the join, called "outerjoin," has been proposed and appears in various commercial systems.

We shall consider the "natural" case first, where the join is on equated values of all attributes in common to the two relations. The *outerjoin* $R \overset{\circ}{\bowtie} S$ is formed by starting with $R \bowtie S$, and adding any dangling tuples from R or S. The added tuples must be padded with a special *null* symbol, $\perp$, in all the attributes that they do not possess but that appear in the join result. Note that $\perp$ is written `NULL` in SQL.

A	B	C
1	2	3
4	5	6
7	8	9

(a) Relation U

B	C	D
2	3	10
2	3	11
6	7	12

(b) Relation V

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$
$\perp$	6	7	12

(c) Result $U \overset{\circ}{\bowtie} V$

Figure 9: Outerjoin of relations

Example 13: In Fig. 9(a) and (b) we see two relations U and V. Tuple $(1, 2, 3)$ of U joins with both $(2, 3, 10)$ and $(2, 3, 11)$ of V, so these three tuples are not dangling. However, the other three tuples — $(4, 5, 6)$ and $(7, 8, 9)$ of U and $(6, 7, 12)$ of V — are dangling. That is, for none of these three tuples is there a tuple of the other relation that agrees with it on both the B and C components. Thus, in $U \overset{\circ}{\bowtie} V$, seen in Fig. 9(c), the three dangling tuples

are padded with $\perp$ in the attributes that they do not have: attribute D for the tuples of U and attribute A for the tuple of V. □

There are many variants of the basic (natural) outerjoin idea. The *left outerjoin* $R \overset{\circ}{\bowtie}_L S$ is like the outerjoin, but only dangling tuples of the left argument R are padded with $\perp$ and added to the result. The *right outerjoin* $R \overset{\circ}{\bowtie}_R S$ is like the outerjoin, but only the dangling tuples of the right argument S are padded with $\perp$ and added to the result.

Example 14: If U and V are as in Fig. 9, then $U \overset{\circ}{\bowtie}_L V$ is:

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$

and $U \overset{\circ}{\bowtie}_R V$ is:

A	B	C	D
1	2	3	10
1	2	3	11
$\perp$	6	7	12

□

In addition, all three natural outerjoin operators have theta-join analogs, where first a theta-join is taken and then those tuples that failed to join with any tuple of the other relation, when the condition of the theta-join was applied, are padded with $\perp$ and added to the result. We use $\overset{\circ}{\bowtie}_C$ to denote a theta-outerjoin with condition C. This operator can also be modified with L or R to indicate left- or right-outerjoin.

Example 15: Let U and V be the relations of Fig. 9, and consider

$$U \overset{\circ}{\bowtie}_{A>V.C} V$$

Tuples $(4, 5, 6)$ and $(7, 8, 9)$ of U each satisfy the condition with both of the tuples $(2, 3, 10)$ and $(2, 3, 11)$ of V. Thus, none of these four tuples are dangling in this theta-join. However, the two other tuples — $(1, 2, 3)$ of U and $(6, 7, 12)$ of V — are dangling. They thus appear, padded, in the result shown in Fig. 10. □

A	$U.B$	$U.C$	$V.B$	$V.C$	D
4	5	6	2	3	10
4	5	6	2	3	11
7	8	9	2	3	10
7	8	9	2	3	11
1	2	3	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	6	7	12

Figure 10: Result of a theta-outerjoin

2.8 Exercises for Section 2

Exercise 2.1: Here are two relations:

$R(A, B)$: $\{(0, 1),\ (2, 3),\ (0, 1),\ (2, 4),\ (3, 4)\}$

$S(B, C)$: $\{(0, 1),\ (2, 4),\ (2, 5),\ (3, 4),\ (0, 2),\ (3, 4)\}$

Compute the following: a) $\pi_{A+B,A^2,B^2}(R)$; b) $\pi_{B+1,C-1}(S)$; c) $\tau_{B,A}(R)$; d) $\tau_{B,C}(S)$; e) $\delta(R)$; f) $\delta(S)$; g) $\gamma_{A,\ \mathtt{SUM}(B)}(R)$; h) $\gamma_{B,\mathtt{AVG}(C)}(S)$; ! i) $\gamma_A(R)$; ! j) $\gamma_{A,\mathtt{MAX}(C)}(R \bowtie S)$; k) $R \overset{\circ}{\bowtie}_L S$; l) $R \overset{\circ}{\bowtie}_R S$; m) $R \overset{\circ}{\bowtie} S$; n) $R \overset{\circ}{\bowtie}_{R.B<S.B} S$.

! Exercise 2.2: A unary operator f is said to be *idempotent* if for all relations R, $f\big(f(R)\big) = f(R)$. That is, applying f more than once is the same as applying it once. Which of the following operators are idempotent? Either explain why or give a counterexample.

a) δ; b) π_L; c) σ_C; d) γ_L; e) τ.

! Exercise 2.3: One thing that can be done with an extended projection, but not with the original version of projection, is to duplicate columns. For example, if $R(A, B)$ is a relation, then $\pi_{A,A}(R)$ produces the tuple (a, a) for every tuple (a, b) in R. Can this operation be done using only the classical operations of relation algebra? Explain your reasoning.

3 A Logic for Relations

As an alternative to abstract query languages based on algebra, one can use a form of logic to express queries. The logical query language *Datalog* ("database logic") consists of if-then rules. Each of these rules expresses the idea that from certain combinations of tuples in certain relations, we may infer that some other tuple must be in some other relation, or in the answer to a query.

3.1 Predicates and Atoms

Relations are represented in Datalog by *predicates*. Each predicate takes a fixed number of arguments, and a predicate followed by its arguments is called an *atom*. The syntax of atoms is just like that of function calls in conventional programming languages; for example $P(x_1, x_2, \ldots, x_n)$ is an atom consisting of the predicate P with arguments $x_1, x_2, \ldots, x_n$.

In essence, a predicate is the name of a function that returns a boolean value. If R is a relation with n attributes in some fixed order, then we shall also use R as the name of a predicate corresponding to this relation. The atom $R(a_1, a_2, \ldots, a_n)$ has value `TRUE` if $(a_1, a_2, \ldots, a_n)$ is a tuple of R; the atom has value `FALSE` otherwise.

Notice that a relation defined by a predicate can be assumed to be a set. In Section 3.6, we shall discuss how it is possible to extend Datalog to bags. However, outside that section, you should assume in connection with Datalog that relations are sets.

Example 16: Let R be the relation

A	B
1	2
3	4

Then $R(1, 2)$ is true and so is $R(3, 4)$. However, for any other combination of values x and y, $R(x, y)$ is false. □

A predicate can take variables as well as constants as arguments. If an atom has variables for one or more of its arguments, then it is a boolean-valued function that takes values for these variables and returns `TRUE` or `FALSE`.

Example 17: If R is the predicate from Example 16, then $R(x, y)$ is the function that tells, for any x and y, whether the tuple (x, y) is in relation R. For the particular instance of R mentioned in Example 16, $R(x, y)$ returns `TRUE` when either

1. $x = 1$ and $y = 2$, or
2. $x = 3$ and $y = 4$

and returns `FALSE` otherwise. As another example, the atom $R(1, z)$ returns `TRUE` if $z = 2$ and returns `FALSE` otherwise. □

3.2 Arithmetic Atoms

There is another kind of atom that is important in Datalog: an *arithmetic atom*. This kind of atom is a comparison between two arithmetic expressions, for example $x < y$ or $x + 1 \geq y + 4 \times z$. For contrast, we shall call the atoms introduced in Section 3.1 *relational atoms*; both kinds are "atoms."

Note that arithmetic and relational atoms each take as arguments the values of any variables that appear in the atom, and they return a boolean value. In effect, arithmetic comparisons like $<$ or $\geq$ are like the names of relations that contain all the true pairs. Thus, we can visualize the relation "$<$" as containing all the tuples, such as $(1, 2)$ or $(-1.5, 65.4)$, whose first component is less than their second component. Remember, however, that database relations are always finite, and usually change from time to time. In contrast, arithmetic-comparison relations such as $<$ are both infinite and unchanging.

3.3 Datalog Rules and Queries

Operations similar to those of relational algebra are described in Datalog by *rules*, which consist of

1. A relational atom called the *head*, followed by
2. The symbol $\leftarrow$, which we often read "if," followed by
3. A *body* consisting of one or more atoms, called *subgoals*, which may be either relational or arithmetic. Subgoals are connected by `AND`, and any subgoal may optionally be preceded by the logical operator `NOT`.

Example 18: The Datalog rule

```
LongMovie(t,y) ← Movies(t,y,l,g,s,p) AND l ≥ 100
```

defines the set of "long" movies, those at least 100 minutes long. It refers to our standard relation `Movies` with schema

```
Movies(title, year, length, genre, studioName, producerC#)
```

The head of the rule is the atom $LongMovie(t, y)$. The body of the rule consists of two subgoals:

1. The first subgoal has predicate *Movies* and six arguments, corresponding to the six attributes of the `Movies` relation. Each of these arguments has a different variable: t for the `title` component, y for the `year` component, l for the `length` component, and so on. We can see this subgoal as saying: "Let (t, y, l, g, s, p) be a tuple in the current instance of relation `Movies`." More precisely, $Movies(t, y, l, g, s, p)$ is true whenever the six variables have values that are the six components of some one `Movies` tuple.
2. The second subgoal, $l \geq 100$, is true whenever the length component of a `Movies` tuple is at least 100.

The rule as a whole can be thought of as saying: $LongMovie(t, y)$ is true whenever we can find a tuple in `Movies` with:

a) t and y as the first two components (for `title` and `year`),

Anonymous Variables

Frequently, Datalog rules have some variables that appear only once. The names used for these variables are irrelevant. Only when a variable appears more than once do we care about its name, so we can see it is the same variable in its second and subsequent appearances. Thus, we shall allow the common convention that an underscore, _, as an argument of an atom, stands for a variable that appears only there. Multiple occurrences of _ stand for different variables, never the same variable. For instance, the rule of Example 18 could be written

$$\texttt{LongMovie(t,y)} \leftarrow \texttt{Movies(t,y,l,\_,\_,\_) AND l} \geq 100$$

The three variables g, s, and p that appear only once have each been replaced by underscores. We cannot replace any of the other variables, since each appears twice in the rule.

b) A third component l (for `length`) that is at least 100, and

c) Any values in components 4 through 6.

Notice that this rule is thus equivalent to the "assignment statement" in relational algebra:

$$\texttt{LongMovie} := \pi_{title,year}\big(\sigma_{length \geq 100}(\texttt{Movies})\big)$$

whose right side is a relational-algebra expression. □

A *query* in Datalog is a collection of one or more rules. If there is only one relation that appears in the rule heads, then the value of this relation is taken to be the answer to the query. Thus, in Example 18, `LongMovie` is the answer to the query. If there is more than one relation among the rule heads, then one of these relations is the answer to the query, while the others assist in the definition of the answer. When there are several predicates defined by a collection of rules, we shall usually assume that the query result is named `Answer`.

3.4 Meaning of Datalog Rules

Example 18 gave us a hint of the meaning of a Datalog rule. More precisely, imagine the variables of the rule ranging over all possible values. Whenever these variables have values that together make all the subgoals true, then we see what the value of the head is for those variables, and we add the resulting tuple to the relation whose predicate is in the head.

For instance, we can imagine the six variables of Example 18 ranging over all possible values. The only combinations of values that can make all the subgoals true are when the values of (t, y, l, g, s, p) in that order form a tuple of `Movies`. Moreover, since the $l \geq 100$ subgoal must also be true, this tuple must be one where l, the value of the `length` component, is at least 100. When we find such a combination of values, we put the tuple (t, y) in the head's relation `LongMovie`.

There are, however, restrictions that we must place on the way variables are used in rules, so that the result of a rule is a finite relation and so that rules with arithmetic subgoals or with *negated* subgoals (those with `NOT` in front of them) make intuitive sense. This condition, which we call the *safety* condition, is:

- Every variable that appears anywhere in the rule must appear in some nonnegated, relational subgoal of the body.

In particular, any variable that appears in the head, in a negated relational subgoal, or in any arithmetic subgoal, must also appear in a nonnegated, relational subgoal of the body.

Example 19: Consider the rule

```
LongMovie(t,y) ← Movies(t,y,l,_,_,_) AND l ≥ 100
```

from Example 18. The first subgoal is a nonnegated, relational subgoal, and it contains all the variables that appear anywhere in the rule, including the anonymous ones represented by underscores. In particular, the two variables t and y that appear in the head also appear in the first subgoal of the body. Likewise, variable l appears in an arithmetic subgoal, but it also appears in the first subgoal. Thus, the rule is safe. □

Example 20: The following rule has three safety violations:

```
P(x,y) ← Q(x,z) AND NOT R(w,x,z) AND x<y
```

1. The variable y appears in the head but not in any nonnegated, relational subgoal of the body. Notice that y's appearance in the arithmetic subgoal $x < y$ does not help to limit the possible values of y to a finite set. As soon as we find values a, b, and c for w, x, and z respectively that satisfy the first two subgoals, we are forced to add the infinite number of tuples (b, d) such that $d > b$ to the relation for the head predicate P.

2. Variable w appears in a negated, relational subgoal but not in a nonnegated, relational subgoal.

3. Variable y appears in an arithmetic subgoal, but not in a nonnegated, relational subgoal.

Thus, it is not a safe rule and cannot be used in Datalog. □

There is another way to define the meaning of rules. Instead of considering all of the possible assignments of values to variables, we consider the sets of tuples in the relations corresponding to each of the nonnegated, relational subgoals. If some assignment of tuples for each nonnegated, relational subgoal is *consistent*, in the sense that it assigns the same value to each occurrence of any one variable, then consider the resulting assignment of values to all the variables of the rule. Notice that because the rule is safe, every variable is assigned a value.

For each consistent assignment, we consider the negated, relational subgoals and the arithmetic subgoals, to see if the assignment of values to variables makes them all true. Remember that a negated subgoal is true if its atom is false. If all the subgoals are true, then we see what tuple the head becomes under this assignment of values to variables. This tuple is added to the relation whose predicate is the head.

Example 21 : Consider the Datalog rule

```
P(x,y) ← Q(x,z) AND R(z,y) AND NOT Q(x,y)
```

Let relation Q contain the two tuples $(1, 2)$ and $(1, 3)$. Let relation R contain tuples $(2, 3)$ and $(3, 1)$. There are two nonnegated, relational subgoals, $Q(x, z)$ and $R(z, y)$, so we must consider all combinations of assignments of tuples from relations Q and R, respectively, to these subgoals. The table of Fig. 11 considers all four combinations.

	Tuple for `Q(x,z)`	Tuple for `R(z,y)`	Consistent Assignment?	`NOT Q(x,y)` True?	Resulting Head
1)	$(1, 2)$	$(2, 3)$	Yes	No	—
2)	$(1, 2)$	$(3, 1)$	No; $z = 2, 3$	Irrelevant	—
3)	$(1, 3)$	$(2, 3)$	No; $z = 3, 2$	Irrelevant	—
4)	$(1, 3)$	$(3, 1)$	Yes	Yes	$P(1, 1)$

Figure 11: All possible assignments of tuples to $Q(x, z)$ and $R(z, y)$

The second and third options in Fig. 11 are not consistent. Each assigns two different values to the variable z. Thus, we do not consider these tuple-assignments further.

The first option, where subgoal $Q(x, z)$ is assigned the tuple $(1, 2)$ and subgoal $R(z, y)$ is assigned tuple $(2, 3)$, yields a consistent assignment, with x, y, and z given the values 1, 3, and 2, respectively. We thus proceed to the test of the other subgoals, those that are not nonnegated, relational subgoals. There is only one: `NOT Q(x,y)`. For this assignment of values to the variables, this subgoal becomes `NOT Q(1,3)`. Since $(1, 3)$ is a tuple of Q, this subgoal is false, and no head tuple is produced for the tuple-assignment (1).

The final option is (4). Here, the assignment is consistent; x, y, and z are assigned the values 1, 1, and 3, respectively. The subgoal `NOT Q(x,y)` takes on the value `NOT Q(1,1)`. Since $(1,1)$ is not a tuple of Q, this subgoal is true. We thus evaluate the head $P(x,y)$ for this assignment of values to variables and find it is $P(1,1)$. Thus the tuple $(1,1)$ is in the relation P. Since we have exhausted all tuple-assignments, this is the only tuple in P. □

3.5 Extensional and Intensional Predicates

It is useful to make the distinction between

- *Extensional* predicates, which are predicates whose relations are stored in a database, and
- *Intensional* predicates, whose relations are computed by applying one or more Datalog rules.

The difference is the same as that between the operands of a relational-algebra expression, which are "extensional" (i.e., defined by their *extension*, which is another name for the "current instance of a relation") and the relations computed by a relational-algebra expression, either as the final result or as an intermediate result corresponding to some subexpression; these relations are "intensional" (i.e., defined by the programmer's "intent").

When talking of Datalog rules, we shall refer to the relation corresponding to a predicate as "intensional" or "extensional," if the predicate is intensional or extensional, respectively. We shall also use the abbreviation *IDB* for "intensional database" to refer to either an intensional predicate or its corresponding relation. Similarly, we use abbreviation *EDB*, standing for "extensional database," for extensional predicates or relations.

Thus, in Example 18, `Movies` is an EDB relation, defined by its extension. The predicate *Movies* is likewise an EDB predicate. Relation and predicate `LongMovie` are both intensional.

An EDB predicate can never appear in the head of a rule, although it can appear in the body of a rule. IDB predicates can appear in either the head or the body of rules, or both. It is also common to construct a single relation by using several rules with the same IDB predicate in the head. We shall see an illustration of this idea in Example 24, regarding the union of two relations.

By using a series of intensional predicates, we can build progressively more complicated functions of the EDB relations. The process is similar to the building of relational-algebra expressions using several operators.

3.6 Datalog Rules Applied to Bags

Datalog is inherently a logic of sets. However, as long as there are no negated, relational subgoals, the ideas for evaluating Datalog rules when relations are sets apply to bags as well. When relations are bags, it is conceptually simpler to use

the second approach for evaluating Datalog rules that we gave in Section 3.4. Recall this technique involves looking at each of the nonnegated, relational subgoals and substituting for it all tuples of the relation for the predicate of that subgoal. If a selection of tuples for each subgoal gives a consistent value to each variable, and the arithmetic subgoals all become true,[1] then we see what the head becomes with this assignment of values to variables. The resulting tuple is put in the head relation.

Since we are now dealing with bags, we do not eliminate duplicates from the head. Moreover, as we consider all combinations of tuples for the subgoals, a tuple appearing n times in the relation for a subgoal gets considered n times as the tuple for that subgoal, each time in conjunction with all combinations of tuples for the other subgoals.

Example 22: Consider the rule

```
H(x,z) ← R(x,y) AND S(y,z)
```

where relation $R(A, B)$ has the tuples:

A	B
1	2
1	2

and $S(B, C)$ has tuples:

B	C
2	3
4	5
4	5

The only time we get a consistent assignment of tuples to the subgoals (i.e., an assignment where the value of y from each subgoal is the same) is when the first subgoal is assigned one of the tuples $(1, 2)$ from R and the second subgoal is assigned tuple $(2, 3)$ from S. Since $(1, 2)$ appears twice in R, and $(2, 3)$ appears once in S, there will be two assignments of tuples that give the variable assignments $x = 1$, $y = 2$, and $z = 3$. The tuple of the head, which is (x, z), is for each of these assignments $(1, 3)$. Thus the tuple $(1, 3)$ appears twice in the head relation H, and no other tuple appears there. That is, the relation

1	3
1	3

[1]Note that there must not be any negated relational subgoals in the rule. There is not a clearly defined meaning of arbitrary Datalog rules with negated, relational subgoals under the bag model.

is the head relation defined by this rule. More generally, had tuple $(1, 2)$ appeared n times in R and tuple $(2, 3)$ appeared m times in S, then tuple $(1, 3)$ would appear nm times in H. □

If a relation is defined by several rules, then the result is the bag-union of whatever tuples are produced by each rule.

Example 23: Consider a relation H defined by the two rules

```
H(x,y) ← S(x,y) AND x>1
H(x,y) ← S(x,y) AND y<5
```

where relation $S(B, C)$ is as in Example 22; that is, $S = \{(2, 3),\ (4, 5),\ (4, 5)\}$. The first rule puts each of the three tuples of S into H, since they each have a first component greater than 1. The second rule puts only the tuple $(2, 3)$ into H, since $(4, 5)$ does not satisfy the condition $y < 5$. Thus, the resulting relation H has two copies of the tuple $(2, 3)$ and two copies of the tuple $(4, 5)$. □

3.7 Exercises for Section 3

!! Exercise 3.1: The requirement we gave for safety of Datalog rules is sufficient to guarantee that the head predicate has a finite relation if the predicates of the relational subgoals have finite relations. However, this requirement is too strong. Give an example of a Datalog rule that violates the condition, yet whatever finite relations we assign to the relational predicates, the head relation will be finite.

4 Relational Algebra and Datalog

Each of the relational-algebra operators of Section 4 can be mimicked by one or several Datalog rules. In this section we shall consider each operator in turn. We shall then consider how to combine Datalog rules to mimic complex algebraic expressions. It is also true that any single safe Datalog rule can be expressed in relational algebra, although we shall not prove that fact here. However, Datalog queries are more powerful than relational algebra when several rules are allowed to interact; they can express recursions that are not expressable in the algebra (see Example 35).

4.1 Boolean Operations

The boolean operations of relational algebra — union, intersection, and set difference — can each be expressed simply in Datalog. Here are the three techniques needed. We assume R and S are relations with the same number of attributes, n. We shall describe the needed rules using `Answer` as the name of the head predicate in all cases. However, we can use anything we wish for the name of the result, and in fact it is important to choose different predicates for the results of different operations.

- To take the union $R \cup S$, use two rules and n distinct variables

  $$a_1, a_2, \ldots, a_n$$

  One rule has $R(a_1, a_2, \ldots, a_n)$ as the lone subgoal and the other has $S(a_1, a_2, \ldots, a_n)$ alone. Both rules have the head $Answer(a_1, a_2, \ldots, a_n)$. As a result, each tuple from R and each tuple of S is put into the answer relation.

- To take the intersection $R \cap S$, use a rule with body

  $$R(a_1, a_2, \ldots, a_n) \texttt{ AND } S(a_1, a_2, \ldots, a_n)$$

  and head $Answer(a_1, a_2, \ldots, a_n)$. Then, a tuple is in the answer relation if and only if it is in both R and S.

- To take the difference $R - S$, use a rule with body

  $$R(a_1, a_2, \ldots, a_n) \texttt{ AND NOT } S(a_1, a_2, \ldots, a_n)$$

  and head $Answer(a_1, a_2, \ldots, a_n)$. Then, a tuple is in the answer relation if and only if it is in R but not in S.

Example 24: Let the schemas for the two relations be $R(A, B, C)$ and $S(A, B, C)$. To avoid confusion, we use different predicates for the various results, rather than calling them all `Answer`.

To take the union $R \cup S$ we use the two rules:

1. `U(x,y,z)` $\leftarrow$ `R(x,y,z)`
2. `U(x,y,z)` $\leftarrow$ `S(x,y,z)`

Rule (1) says that every tuple in R is a tuple in the IDB relation U. Rule (2) similarly says that every tuple in S is in U.

To compute $R \cap S$, we use the rule

`I(a,b,c)` $\leftarrow$ `R(a,b,c) AND S(a,b,c)`

Finally, the rule

`D(a,b,c)` $\leftarrow$ `R(a,b,c) AND NOT S(a,b,c)`

computes the difference $R - S$. □

Variables Are Local to a Rule

Notice that the names we choose for variables in a rule are arbitrary and have no connection to the variables used in any other rule. The reason there is no connection is that each rule is evaluated alone and contributes tuples to its head's relation independent of other rules. Thus, for instance, we could replace the second rule of Example 24 by

$$\texttt{U(a,b,c)} \leftarrow \texttt{S(a,b,c)}$$

while leaving the first rule unchanged, and the two rules would still compute the union of R and S. Note, however, that when substituting one variable u for another variable v within a rule, we must substitute u for all occurrences of v within the rule. Moreover, the substituting variable u that we choose must not be a variable that already appears in the rule.

4.2 Projection

To compute a projection of a relation R, we use one rule with a single subgoal with predicate R. The arguments of this subgoal are distinct variables, one for each attribute of the relation. The head has an atom with arguments that are the variables corresponding to the attributes in the projection list, in the desired order.

Example 25: Suppose we want to project the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

onto its first three attributes — `title`, `year`, and `length`. The rule

$$\texttt{P(t,y,l)} \leftarrow \texttt{Movies(t,y,l,g,s,p)}$$

serves, defining a relation called P to be the result of the projection. □

4.3 Selection

Selections can be somewhat more difficult to express in Datalog. The simple case is when the selection condition is the `AND` of one or more arithmetic comparisons. In that case, we create a rule with

1. One relational subgoal for the relation upon which we are performing the selection. This atom has distinct variables for each component, one for each attribute of the relation.

2. For each comparison in the selection condition, an arithmetic subgoal that is identical to this comparison. However, while in the selection condition an attribute name was used, in the arithmetic subgoal we use the corresponding variable, following the correspondence established by the relational subgoal.

Example 26: The selection

$$\sigma_{length \geq 100 \texttt{ AND } studioName=\texttt{'Fox'}}(\texttt{Movies})$$

can be written as a Datalog rule

```
S(t,y,l,g,s,p) ← Movies(t,y,l,g,s,p) AND l ≥ 100 AND s = 'Fox'
```

The result is the relation S. Note that l and s are the variables corresponding to attributes `length` and `studioName` in the standard order we have used for the attributes of `Movies`. □

Now, let us consider selections that involve the `OR` of conditions. We cannot necessarily replace such selections by single Datalog rules. However, selection for the `OR` of two conditions is equivalent to selecting for each condition separately and then taking the union of the results. Thus, the `OR` of n conditions can be expressed by n rules, each of which defines the same head predicate. The ith rule performs the selection for the ith of the n conditions.

Example 27: Let us modify the selection of Example 26 by replacing the `AND` by an `OR` to get the selection:

$$\sigma_{length \geq 100 \texttt{ OR } studioName=\texttt{'Fox'}}(\texttt{Movies})$$

That is, find all those movies that are either long or by Fox. We can write two rules, one for each of the two conditions:

```
1. S(t,y,l,g,s,p) ← Movies(t,y,l,g,s,p) AND l ≥ 100
2. S(t,y,l,g,s,p) ← Movies(t,y,l,g,s,p) AND s = 'Fox'
```

Rule (1) produces movies at least 100 minutes long, and rule (2) produces movies by Fox. □

Even more complex selection conditions can be formed by several applications, in any order, of the logical operators `AND`, `OR`, and `NOT`. However, there is a widely known technique, which we shall not present here, for rearranging any such logical expression into "disjunctive normal form," where the expression is the disjunction (`OR`) of "conjuncts." A *conjunct*, in turn, is the `AND` of "literals," and a *literal* is either a comparison or a negated comparison.[2]

[2]See, e.g., A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, 1992.

We can represent any literal by a subgoal, perhaps with a NOT in front of it. If the subgoal is arithmetic, the NOT can be incorporated into the comparison operator. For example, NOT x $\geq$ 100 can be written as x < 100. Then, any conjunct can be represented by a single Datalog rule, with one subgoal for each comparison. Finally, every disjunctive-normal-form expression can be written by several Datalog rules, one rule for each conjunct. These rules take the union, or OR, of the results from each of the conjuncts.

Example 28: We gave a simple instance of this algorithm in Example 27. A more difficult example can be formed by negating the condition of that example. We then have the expression:

$$\sigma_{\texttt{NOT } (length \geq 100 \texttt{ OR } studioName = \texttt{'Fox'})}(\texttt{Movies})$$

That is, find all those movies that are neither long nor by Fox.

Here, a NOT is applied to an expression that is itself not a simple comparison. Thus, we must push the NOT down the expression, using one form of *DeMorgan's laws*, which says that the negation of an OR is the AND of the negations. That is, the selection can be rewritten:

$$\sigma_{(\texttt{NOT } (length \geq 100)) \texttt{ AND } (\texttt{NOT } (studioName = \texttt{'Fox'}))}(\texttt{Movies})$$

Now, we can take the NOT's inside the comparisons to get the expression:

$$\sigma_{length < 100 \texttt{ AND } studioName \neq \texttt{'Fox'}}(\texttt{Movies})$$

This expression can be converted into the Datalog rule

```
S(t,y,l,g,s,p) ← Movies(t,y,l,g,s,p) AND l < 100 AND s ≠ 'Fox'
```

□

Example 29: Let us consider a similar example where we have the negation of an AND in the selection. Now, we use the second form of DeMorgan's law, which says that the negation of an AND is the OR of the negations. We begin with the algebraic expression

$$\sigma_{\texttt{NOT } (length \geq 100 \texttt{ AND } studioName = \texttt{'Fox'})}(\texttt{Movies})$$

That is, find all those movies that are not both long and by Fox.

We apply DeMorgan's law to push the NOT below the AND, to get:

$$\sigma_{(\texttt{NOT } (length \geq 100)) \texttt{ OR } (\texttt{NOT } (studioName = \texttt{'Fox'}))}(\texttt{Movies})$$

Again we take the NOT's inside the comparisons to get:

$$\sigma_{length < 100 \texttt{ OR } studioName \neq \texttt{'Fox'}}(\texttt{Movies})$$

Finally, we write two rules, one for each part of the OR. The resulting Datalog rules are:

```
1. S(t,y,l,g,s,p) ← Movies(t,y,l,g,s,p) AND l < 100
2. S(t,y,l,g,s,p) ← Movies(t,y,l,g,s,p) AND s ≠ 'Fox'
```

□

4.4 Product

The product of two relations $R \times S$ can be expressed by a single Datalog rule. This rule has two subgoals, one for R and one for S. Each of these subgoals has distinct variables, one for each attribute of R or S. The IDB predicate in the head has as arguments all the variables that appear in either subgoal, with the variables appearing in the R-subgoal listed before those of the S-subgoal.

Example 30: Let us consider the two three-attribute relations R and S from Example 24. The rule

```
P(a,b,c,x,y,z) ← R(a,b,c) AND S(x,y,z)
```

defines P to be $R \times S$. We have arbitrarily used variables at the beginning of the alphabet for the arguments of R and variables at the end of the alphabet for S. These variables all appear in the rule head. □

4.5 Joins

We can take the natural join of two relations by a Datalog rule that looks much like the rule for a product. The difference is that if we want $R \bowtie S$, then we must use the same variable for attributes of R and S that have the same name and must use different variables otherwise. For instance, we can use the attribute names themselves as the variables. The head is an IDB predicate that has each variable appearing once.

Example 31: Consider relations with schemas $R(A, B)$ and $S(B, C, D)$. Their natural join may be defined by the rule

```
J(a,b,c,d) ← R(a,b) AND S(b,c,d)
```

Notice how the variables used in the subgoals correspond in an obvious way to the attributes of the relations R and S. □

We also can convert theta-joins to Datalog. Recall how a theta-join can be expressed as a product followed by a selection. If the selection condition is a conjunct, that is, the AND of comparisons, then we may simply start with the Datalog rule for the product and add additional, arithmetic subgoals, one for each of the comparisons.

Example 32: Consider the relations $U(A, B, C)$ and $V(B, C, D)$ and the theta-join:

$$U \bowtie_{A<D \text{ AND } U.B \neq V.B} V$$

We can construct the Datalog rule

```
J(a,ub,uc,vb,vc,d)  ←  U(a,ub,uc) AND V(vb,vc,d) AND
                       a < d AND ub ≠ vb
```

to perform the same operation. We have used ub as the variable corresponding to attribute B of U, and similarly used vb, uc, and vc, although any six distinct variables for the six attributes of the two relations would be fine. The first two subgoals introduce the two relations, and the second two subgoals enforce the two comparisons that appear in the condition of the theta-join. □

If the condition of the theta-join is not a conjunction, then we convert it to disjunctive normal form, as discussed in Section 4.3. We then create one rule for each conjunct. In this rule, we begin with the subgoals for the product and then add subgoals for each literal in the conjunct. The heads of all the rules are identical and have one argument for each attribute of the two relations being theta-joined.

Example 33: In this example, we shall make a simple modification to the algebraic expression of Example 32. The `AND` will be replaced by an `OR`. There are no negations in this expression, so it is already in disjunctive normal form. There are two conjuncts, each with a single literal. The expression is:

$$U \bowtie_{A<D \texttt{ OR } U.B \neq V.B} V$$

Using the same variable-naming scheme as in Example 32, we obtain the two rules

1. `J(a,ub,uc,vb,vc,d) ← U(a,ub,uc) AND V(vb,vc,d) AND a < d`
2. `J(a,ub,uc,vb,vc,d) ← U(a,ub,uc) AND V(vb,vc,d) AND ub ≠ vb`

Each rule has subgoals for the two relations involved plus a subgoal for one of the two conditions $A < D$ or $U.B \neq V.B$. □

4.6 Simulating Multiple Operations with Datalog

Datalog rules are not only capable of mimicking a single operation of relational algebra. We can in fact mimic any algebraic expression. The trick is to look at the expression tree for the relational-algebra expression and create one IDB predicate for each interior node of the tree. The rule or rules for each IDB predicate is whatever we need to apply the operator at the corresponding node of the tree. Those operands of the tree that are extensional (i.e., they are relations of the database) are represented by the corresponding predicate. Operands that are themselves interior nodes are represented by the corresponding IDB predicate. The result of the algebraic expression is the relation for the predicate associated with the root of the expression tree.

Example 34: Consider the algebraic expression

$$\pi_{title,year}\Big(\sigma_{length\geq 100}(\texttt{Movies}) \cap \sigma_{studioName=\texttt{'Fox'}}(\texttt{Movies})\Big)$$

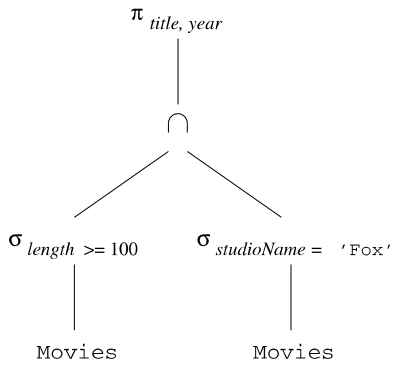

Figure 12: Expression tree

1. `W(t,y,l,g,s,p) ← Movies(t,y,l,g,s,p) AND l ≥ 100`
2. `X(t,y,l,g,s,p) ← Movies(t,y,l,g,s,p) AND s = 'Fox'`
3. `Y(t,y,l,g,s,p) ← W(t,y,l,g,s,p) AND X(t,y,l,g,s,p)`
4. `Answer(t,y) ← Y(t,y,l,g,s,p)`

Figure 13: Datalog rules to perform several algebraic operations

whose expression tree is presented in Fig. 12. There are four interior nodes, so we need to create four IDB predicates. Each of these predicates has a single Datalog rule, and we summarize all the rules in Fig. 13.

The lowest two interior nodes perform simple selections on the EDB relation `Movies`, so we can create the IDB predicates W and X to represent these selections. Rules (1) and (2) of Fig. 13 describe these selections. For example, rule (1) defines W to be those tuples of `Movies` that have a length at least 100.

Then rule (3) defines predicate Y to be the intersection of W and X, using the form of rule we learned for an intersection in Section 4.1. Finally, rule (4) defines the answer to be the projection of Y onto the `title` and `year` attributes. We here use the technique for simulating a projection that we learned in Section 4.2.

Note that, because Y is defined by a single rule, we can substitute for the Y subgoal in rule (4) of Fig. 13, replacing it with the body of rule (3). Then, we can substitute for the W and X subgoals, using the bodies of rules (1) and (2). Since the `Movies` subgoal appears in both of these bodies, we can eliminate one copy. As a result, the single rule

```
Answer(t,y) ← Movies(t,y,l,g,s,p) AND l ≥ 100 AND s = 'Fox'
```

suffices. □

4.7 Comparison Between Datalog and Relational Algebra

We see from Section 4.6 that every expression in basic relational algebra can be expressed as a Datalog query. There are operations in the extended relational algebra, such as grouping and aggregation from Section 2, that have no corresponding features in the Datalog version we have presented here. Likewise, Datalog does not support bag operations such as duplicate elimination.

It is also true that any single Datalog rule can be expressed in relational algebra. That is, we can write a query in the basic relational algebra that produces the same set of tuples as the head of that rule produces.

However, when we consider collections of Datalog rules, the situation changes. Datalog rules can express recursion, which relational algebra can not. The reason is that IDB predicates can also be used in the bodies of rules, and the tuples we discover for the heads of rules can thus feed back to rule bodies and help produce more tuples for the heads. We shall not discuss here any of the complexities that arise, especially when the rules have negated subgoals. However, the following example will illustrate recursive Datalog.

Example 35: Suppose we have a relation `Edge(X,Y)` that says there is a directed edge (arc) from node X to node Y. We can express the transitive closure of the edge relation, that is, the relation `Path(X,Y)` meaning that there is a path of length 1 or more from node X to node Y, as follows:

1. `Path(X,Y)` $\leftarrow$ `Edge(X,Y)`
2. `Path(X,Y)` $\leftarrow$ `Edge(X,Z) AND Path(Z,Y)`

Rule (1) says that every edge is a path. Rule (2) says that if there is an edge from node X to some node Z and a path from Z to Y, then there is also a path from X to Y. If we apply Rule (1) and then Rule (2), we get the paths of length 2. If we take the *Path* facts we get from this application and use them in another application of Rule (2), we get paths of length 3. Feeding those *Path* facts back again gives us paths of length 4, and so on. Eventually, we discover all possible path facts, and on one round we get no new facts. At that point, we can stop. If we haven't discovered the fact $Path(a, b)$, then there really is no path in the graph from node a to node b. □

4.8 Exercises for Section 4

Exercise 4.1: Let $R(a, b, c)$, $S(a, b, c)$, and $T(a, b, c)$ be three relations. Write one or more Datalog rules that define the result of each of the following expressions of relational algebra:

a) $R \cup S$.

b) $R \cap S$.

c) $R - S$.

d) $(R \cup S) - T$.

! e) $(R - S) \cap (R - T)$.

f) $\pi_{a,b}(R)$.

! g) $\pi_{a,b}(R) \cap \rho_{U(a,b)}\big(\pi_{b,c}(S)\big)$.

Exercise 4.2: Let $R(x, y, z)$ be a relation. Write one or more Datalog rules that define $\sigma_C(R)$, where C stands for each of the following conditions:

a) $x = y$.

b) $x < y$ `AND` $y < z$.

c) $x < y$ `OR` $y < z$.

d) `NOT` $(x < y$ `OR` $x > y)$.

! e) `NOT` $\big((x < y$ `OR` $x > y)$ `AND` $y < z\big)$.

! f) `NOT` $\big((x < y$ `OR` $x < z)$ `AND` $y < z\big)$.

Exercise 4.3: Let $R(a, b, c)$, $S(b, c, d)$, and $T(d, e)$ be three relations. Write single Datalog rules for each of the natural joins:

a) $R \bowtie S$.

b) $S \bowtie T$.

c) $(R \bowtie S) \bowtie T$. (*Note*: since the natural join is associative and commutative, the order of the join of these three relations is irrelevant.)

Exercise 4.4: Let $R(x, y, z)$ and $S(x, y, z)$ be two relations. Write one or more Datalog rules to define each of the theta-joins $R \bowtie_C S$, where C is one of the conditions of Exercise 4.2. For each of these conditions, interpret each arithmetic comparison as comparing an attribute of R on the left with an attribute of S on the right. For instance, $x < y$ stands for $R.x < S.y$.

! **Exercise 4.5:** It is also possible to convert Datalog rules into equivalent relational-algebra expressions. While we have not discussed the method of doing so in general, it is possible to work out many simple examples. For each of the Datalog rules below, write an expression of relational algebra that defines the same relation as the head of the rule.

a) `P(x,y) ← Q(x,z) AND R(z,y)`

b) `P(x,y) ← Q(x,z) AND Q(z,y)`

c) `P(x,y) ← Q(x,z) AND R(z,y) AND x < y`

5 Summary

- *Relations as Bags*: In commercial database systems, relations are actually bags, in which the same tuple is allowed to appear several times. The operations of relational algebra on sets can be extended to bags, but there are some algebraic laws that fail to hold.
- *Extensions to Relational Algebra*: To match the capabilities of SQL, some operators not present in the core relational algebra are needed. Sorting of a relation is an example, as is an extended projection, where computation on columns of a relation is supported. Grouping, aggregation, and outerjoins are also needed.
- *Grouping and Aggregation*: Aggregations summarize a column of a relation. Typical aggregation operators are sum, average, count, minimum, and maximum. The grouping operator allows us to partition the tuples of a relation according to their value(s) in one or more attributes before computing aggregation(s) for each group.
- *Outerjoins*: The outerjoin of two relations starts with a join of those relations. Then, dangling tuples (those that failed to join with any tuple) from either relation are padded with null values for the attributes belonging only to the other relation, and the padded tuples are included in the result.
- *Datalog*: This form of logic allows us to write queries in the relational model. In Datalog, one writes rules in which a head predicate or relation is defined in terms of a body, consisting of subgoals.
- *Atoms*: The head and subgoals are each atoms, and an atom consists of an (optionally negated) predicate applied to some number of arguments. Predicates may represent either relations or arithmetic comparisons such as $<$.
- *IDB and EDB Predicates*: Some predicates correspond to stored relations, and are called EDB (extensional database) predicates or relations. Other predicates, called IDB (intensional database), are defined by the rules. EDB predicates may not appear in rule heads.
- *Safe Rules*: Datalog rules must be safe, meaning that every variable in the rule appears in some nonnegated, relational subgoal of the body. Safe rules guarantee that if the EDB relations are finite, then the IDB relations will be finite.
- *Relational Algebra and Datalog*: All queries that can be expressed in core relational algebra can also be expressed in Datalog. If the rules are safe and nonrecursive, then they define exactly the same set of queries as core relational algebra.

6 References

The relational algebra comes from [2]. The extended operator γ is from [5].

Codd also introduced two forms of first-order logic called *tuple relational calculus* and *domain relational calculus* in one of his early papers on the relational model [3]. These forms of logic are equivalent in expressive power to relational algebra, a fact proved in [3].

Datalog, looking more like logical rules, was inspired by the programming language Prolog. The book [4] originated much of the development of logic as a query language, while [1] placed the ideas in the context of database systems.

More on Datalog and relational calculus can be found in [6] and [7].

1. F. Bancilhon, and R. Ramakrishnan, "An amateur's introduction to recursive query-processing strategies," *ACM SIGMOD Intl. Conf. on Management of Data*, pp. 16–52, 1986.

2. E. F. Codd, "A relational model for large shared data banks," *Comm. ACM* **13**:6, pp. 377–387, 1970.

3. E. F. Codd, "Relational completeness of database sublanguages," in *Database Systems* (R. Rustin, ed.), Prentice Hall, Englewood Cliffs, NJ, 1972.

4. H. Gallaire and J. Minker, *Logic and Databases*, Plenum Press, New York, 1978.

5. A. Gupta, V. Harinarayan, and D. Quass, "Generalized projections: a powerful approach to aggregation," *21st Intl. Conf. on Very Large Databases Systems*, pp. 358–369, 1995.

6. M. Liu, "Deductive database languages: problems and solutions," *Computing Surveys* **31**:1 (March, 1999), pp. 27–62.

7. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volumes I and II*, Computer Science Press, New York, 1988, 1989.

The Database Language SQL

The most commonly used relational DBMS's query and modify the database through a language called SQL (sometimes pronounced "sequel"). SQL stands for "Structured Query Language." The portion of SQL that supports queries has capabilities very close to that of relational algebra. However, SQL also includes statements for modifying the database (e.g., inserting and deleting tuples from relations) and for declaring a database schema. Thus, SQL serves as both a data-manipulation language and as a data-definition language. SQL also standardizes many other database commands.

There are many different dialects of SQL. First, there are three major standards. There is ANSI (American National Standards Institute) SQL and an updated standard adopted in 1992, called SQL-92 or SQL2. The most recent SQL-99 (previously referred to as SQL3) standard extends SQL2 with object-relational features and a number of other new capabilities. There is also a collection of extensions to SQL-99, collectively called SQL:2003. Then, there are versions of SQL produced by the principal DBMS vendors. These all include the capabilities of the original ANSI standard. They also conform to a large extent to the more recent SQL2, although each has its variations and extensions beyond SQL2, including some, but not all, of the features in the SQL-99 and SQL:2003 standards.

This chapter introduces the basics of SQL: the query language and database modification statements. We also introduce the notion of a "transaction," the basic unit of work for database systems. This study, although simplifed, will give you a sense of how database operations can interact and some of the resulting pitfalls.

From Chapter 6 of *A First Course in Database Systems*, Third Edition. Jeffrey Ullman.

The intent of this chapter and the following are to provide the reader with a sense of what SQL is about, more at the level of a "tutorial" than a "manual." Thus, we focus on the most commonly used features only, and we try to use code that not only conforms to the standard, but to the usage of commercial DBMS's. The references mention places where more of the details of the language and its dialects can be found.

1 Simple Queries in SQL

Perhaps the simplest form of query in SQL asks for those tuples of some one relation that satisfy a condition. Such a query is analogous to a selection in relational algebra. This simple query, like almost all SQL queries, uses the three keywords, `SELECT`, `FROM`, and `WHERE` that characterize SQL.

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Figure 1: Example database schema

Example 1: In this and subsequent examples, we shall use the movie database schema from Fig. 1.

As our first query, let us ask about the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

for all movies produced by Disney Studios in 1990. In SQL, we say

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

This query exhibits the characteristic select-from-where form of most SQL queries.

How SQL is Used

In this chapter, we assume a *generic query interface*, where we type SQL queries or other statements and have them execute. In practice, the generic interface is used rarely. Rather, there are large programs, written in a conventional language such as C or Java (called the *host language*). These programs issue SQL statements to a database, using a special library for the host language. Data is moved from host-language variables to the SQL statements, and the results of those statements are moved from the database to host-language variables.

- The `FROM` clause gives the relation or relations to which the query refers. In our example, the query is about the relation `Movies`.
- The `WHERE` clause is a condition, much like a selection-condition in relational algebra. Tuples must satisfy the condition in order to match the query. Here, the condition is that the `studioName` attribute of the tuple has the value `'Disney'` and the `year` attribute of the tuple has the value 1990. All tuples meeting both stipulations satisfy the condition; other tuples do not.
- The `SELECT` clause tells which attributes of the tuples matching the condition are produced as part of the answer. The `*` in this example indicates that the entire tuple is produced. The result of the query is the relation consisting of all tuples produced by this process.

One way to interpret this query is to consider each tuple of the relation mentioned in the `FROM` clause. The condition in the `WHERE` clause is applied to the tuple. More precisely, any attributes mentioned in the `WHERE` clause are replaced by the value in the tuple's component for that attribute. The condition is then evaluated, and if true, the components appearing in the `SELECT` clause are produced as one tuple of the answer. Thus, the result of the query is the `Movies` tuples for those movies produced by Disney in 1990, for example, *Pretty Woman*.

In detail, when the SQL query processor encounters the `Movies` tuple

title	*year*	*length*	*genre*	*studioName*	*producerC#*
`Pretty Woman`	`1990`	`119`	`true`	`Disney`	`999`

(here, 999 is the imaginary certificate number for the producer of the movie), the value `'Disney'` is substituted for attribute `studioName` and value 1990 is substituted for attribute `year` in the condition of the `WHERE` clause, because these are the values for those attributes in the tuple in question. The `WHERE` clause thus becomes

A Trick for Reading and Writing Queries

It is generally easist to examine a select-from-where query by first looking at the `FROM` clause, to learn which relations are involved in the query. Then, move to the `WHERE` clause, to learn what it is about tuples that is important to the query. Finally, look at the `SELECT` clause to see what the output is. The same order — from, then where, then select — is often useful when writing queries of your own, as well.

```
WHERE 'Disney' = 'Disney' AND 1990 = 1990
```

Since this condition is evidently true, the tuple for *Pretty Woman* passes the test of the `WHERE` clause and the tuple becomes part of the result of the query. □

1.1 Projection in SQL

We can, if we wish, eliminate some of the components of the chosen tuples; that is, we can project the relation produced by a SQL query onto some of its attributes. In place of the `*` of the `SELECT` clause, we may list some of the attributes of the relation mentioned in the `FROM` clause. The result will be projected onto the attributes listed.[1]

Example 2: Suppose we wish to modify the query of Example 1 to produce only the movie title and length. We may write

```
SELECT title, length
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

The result is a table with two columns, headed `title` and `length`. The tuples in this table are pairs, each consisting of a movie title and its length, such that the movie was produced by Disney in 1990. For instance, the relation schema and one of its tuples looks like:

title	*length*
Pretty Woman	119
...	...

□

[1] Thus, the keyword `SELECT` in SQL actually corresponds most closely to the projection operator of relational algebra, while the selection operator of the algebra corresponds to the `WHERE` clause of SQL queries.

Sometimes, we wish to produce a relation with column headers different from the attributes of the relation mentioned in the `FROM` clause. We may follow the name of the attribute by the keyword `AS` and an *alias*, which becomes the header in the result relation. Keyword `AS` is optional. That is, an alias can immediately follow what it stands for, without any intervening punctuation.

Example 3: We can modify Example 2 to produce a relation with attributes `name` and `duration` in place of `title` and `length` as follows.

```
SELECT title AS name, length AS duration
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

The result is the same set of tuples as in Example 2, but with the columns headed by attributes `name` and `duration`. For example,

name	*duration*
Pretty Woman	119
...	...

could be the first tuple in the result. □

Another option in the `SELECT` clause is to use an expression in place of an attribute. Put another way, the `SELECT` list can function like the lists in an extended projection. We shall see in Section 4 that the `SELECT` list can also include aggregates.

Example 4: Suppose we want output as in Example 3, but with the length in hours. We might replace the `SELECT` clause of that example with

```
SELECT title AS name, length*0.016667 AS lengthInHours
```

Then the same movies would be produced, but lengths would be calculated in hours and the second column would be headed by attribute `lengthInHours`, as:

name	*lengthInHours*
Pretty Woman	1.98334
...	...

□

Example 5: We can even allow a constant as an expression in the `SELECT` clause. It might seem pointless to do so, but one application is to put some useful words into the output that SQL displays. The following query:

Case Insensitivity

SQL is *case insensitive*, meaning that it treats upper- and lower-case letters as the same letter. For example, although we have chosen to write keywords like `FROM` in capitals, it is equally proper to write this keyword as `From` or `from`, or even `FrOm`. Names of attributes, relations, aliases, and so on are similarly case insensitive. Only inside quotes does SQL make a distinction between upper- and lower-case letters. Thus, `'FROM'` and `'from'` are different character strings. Of course, neither is the keyword `FROM`.

```
SELECT title, length*0.016667 AS length, 'hrs.' AS inHours
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

produces tuples such as

title	*length*	*inHours*
Pretty Woman	1.98334	hrs.
...	...	...

We have arranged that the third column is called `inHours`, which fits with the column header `length` in the second column. Every tuple in the answer will have the constant `hrs.` in the third column, which gives the illusion of being the units attached to the value in the second column. □

1.2 Selection in SQL

The selection operator of relational algebra, and much more, is available through the `WHERE` clause of SQL. The expressions that may follow `WHERE` include conditional expressions like those found in common languages such as C or Java.

We may build expressions by comparing values using the six common comparison operators: `=`, `<>`, `<`, `>`, `<=`, and `>=`. The last four operators are as in C, but `<>` is the SQL symbol for "not equal to" (`!=` in C), and `=` in SQL is equality (`==` in C).

The values that may be compared include constants and attributes of the relations mentioned after `FROM`. We may also apply the usual arithmetic operators, $+$, $*$, and so on, to numeric values before we compare them. For instance, $(\texttt{year} - 1930) * (\texttt{year} - 1930) < 100$ is true for those years within 9 of 1930. We may apply the concatenation operator `||` to strings; for example `'foo' || 'bar'` has value `'foobar'`.

An example comparison is

```
studioName = 'Disney'
```

SQL Queries and Relational Algebra

The simple SQL queries that we have seen so far all have the form:

```
SELECT L
FROM R
WHERE C
```

in which L is a list of expressions, R is a relation, and C is a condition. The meaning of any such expression is the same as that of the relational-algebra expression

$$\pi_L\big(\sigma_C(R)\big)$$

That is, we start with the relation in the `FROM` clause, apply to each tuple whatever condition is indicated in the `WHERE` clause, and then project onto the list of attributes and/or expressions in the `SELECT` clause.

in Example 1. The attribute `studioName` of the relation `Movies` is tested for equality against the constant `'Disney'`. This constant is string-valued; strings in SQL are denoted by surrounding them with single quotes. Numeric constants, integers and reals, are also allowed, and SQL uses the common notations for reals such as `-12.34` or `1.23E45`.

The result of a comparison is a boolean value: either `TRUE` or `FALSE`.[2] Boolean values may be combined by the logical operators `AND`, `OR`, and `NOT`, with their expected meanings. For instance, we saw in Example 1 how two conditions could be combined by `AND`. The `WHERE` clause of this example evaluates to true if and only if both comparisons are satisfied; that is, the studio name is `'Disney'` and the year is 1990. Here is an example of a query with a complex `WHERE` clause.

Example 6: Consider the query

```
SELECT title
FROM Movies
WHERE (year > 1970 OR length < 90) AND studioName = 'MGM';
```

This query asks for the titles of movies made by MGM Studios that either were made after 1970 or were less than 90 minutes long. Notice that comparisons can be grouped using parentheses. The parentheses are needed here because the precedence of logical operators in SQL is the same as in most other languages: `AND` takes precedence over `OR`, and `NOT` takes precedence over both. □

[2] Well there's a bit more to boolean values; see Section 1.7.

Representing Bit Strings

A string of bits is represented by `B` followed by a quoted string of 0's and 1's. Thus, `B'011'` represents the string of three bits, the first of which is 0 and the other two of which are 1. Hexadecimal notation may also be used, where an `X` is followed by a quoted string of hexadecimal digits (0 through 9, and a through f, with the latter representing "digits" 10 through 15). For instance, `X'7ff'` represents a string of twelve bits, a 0 followed by eleven 1's. Note that each hexadecimal digit represents four bits, and leading 0's are not suppressed.

1.3 Comparison of Strings

Two strings are equal if they are the same sequence of characters. Recall that strings can be stored as fixed-length strings, using `CHAR`, or variable-length strings, using `VARCHAR`. When comparing strings with different declarations, only the actual strings are compared; SQL ignores any "pad" characters that must be present in the database in order to give a string its required length.

When we compare strings by one of the "less than" operators, such as `<` or `>=`, we are asking whether one precedes the other in lexicographic order (i.e., in dictionary order, or alphabetically). That is, if $a_1a_2\cdots a_n$ and $b_1b_2\cdots b_m$ are two strings, then the first is "less than" the second if either $a_1 < b_1$, or if $a_1 = b_1$ and $a_2 < b_2$, or if $a_1 = b_1$, $a_2 = b_2$, and $a_3 < b_3$, and so on. We also say $a_1a_2\cdots a_n < b_1b_2\cdots b_m$ if $n < m$ and $a_1a_2\cdots a_n = b_1b_2\cdots b_n$; that is, the first string is a proper prefix of the second. For instance, `'fodder'` < `'foo'`, because the first two characters of each string are the same, `fo`, and the third character of `fodder` precedes the third character of `foo`. Also, `'bar'` < `'bargain'` because the former is a proper prefix of the latter.

1.4 Pattern Matching in SQL

SQL also provides the capability to compare strings on the basis of a simple pattern match. An alternative form of comparison expression is

```
s LIKE p
```

where s is a string and p is a *pattern*, that is, a string with the optional use of the two special characters `%` and `_`. Ordinary characters in p match only themselves in s. But `%` in p can match any sequence of 0 or more characters in s, and `_` in p matches any one character in s. The value of this expression is true if and only if string s matches pattern p. Similarly, s `NOT LIKE` p is true if and only if string s does not match pattern p.

Example 7: We remember a movie "Star something," and we remember that the something has four letters. What could this movie be? We can retrieve all such names with the query:

```
SELECT title
FROM Movies
WHERE title LIKE 'Star ____';
```

This query asks if the title attribute of a movie has a value that is nine characters long, the first five characters being `Star` and a blank. The last four characters may be anything, since any sequence of four characters matches the four _ symbols. The result of the query is the set of complete matching titles, such as *Star Wars* and *Star Trek*. □

Example 8: Let us search for all movies with a possessive ('s) in their titles. The desired query is

```
SELECT title
FROM Movies
WHERE title LIKE '%''s%';
```

To understand this pattern, we must first observe that the apostrophe, being the character that surrounds strings in SQL, cannot also represent itself. The convention taken by SQL is that two consecutive apostrophes in a string represent a single apostrophe and do not end the string. Thus, `''s` in a pattern is matched by a single apostrophe followed by an `s`.

The two `%` characters on either side of the `'s` match any strings whatsoever. Thus, any title with `'s` as a substring will match the pattern, and the answer to this query will include films such as *Logan's Run* or *Alice's Restaurant*. □

1.5 Dates and Times

Implementations of SQL generally support dates and times as special data types. These values are often representable in a variety of formats such as `05/14/1948` or `14 May 1948`. Here we shall describe only the SQL standard notation, which is very specific about format.

A *date* constant is represented by the keyword `DATE` followed by a quoted string of a special form. For example, `DATE '1948-05-14'` follows the required form. The first four characters are digits representing the year. Then come a hyphen and two digits representing the month. Note that, as in our example, a one-digit month is padded with a leading 0. Finally there is another hyphen and two digits representing the day. As with months, we pad the day with a leading 0 if that is necessary to make a two-digit number.

A *time* constant is represented similarly by the keyword `TIME` and a quoted string. This string has two digits for the hour, on the military (24-hour)

Escape Characters in LIKE expressions

What if the pattern we wish to use in a LIKE expression involves the characters % or _? Instead of having a particular character used as the escape character (e.g., the backslash in most UNIX commands), SQL allows us to specify any one character we like as the escape character for a single pattern. We do so by following the pattern by the keyword ESCAPE and the chosen escape character, in quotes. A character % or _ preceded by the escape character in the pattern is interpreted literally as that character, not as a symbol for any sequence of characters or any one character, respectively. For example,

```
s LIKE 'x%%x%' ESCAPE 'x'
```

makes x the escape character in the pattern x%%x%. The sequence x% is taken to be a single %. This pattern matches any string that begins and ends with the character %. Note that only the middle % has its "any string" interpretation.

clock. Then come a colon, two digits for the minute, another colon, and two digits for the second. If fractions of a second are desired, we may continue with a decimal point and as many significant digits as we like. For instance, TIME '15:00:02.5' represents the time at which all students will have left a class that ends at 3 PM: two and a half seconds past three o'clock.

Alternatively, time can be expressed as the number of hours and minutes ahead of (indicated by a plus sign) or behind (indicated by a minus sign) Greenwich Mean Time (GMT). For instance, TIME '12:00:00-8:00' represents noon in Pacific Standard Time, which is eight hours behind GMT.

To combine dates and times we use a value of type TIMESTAMP. These values consist of the keyword TIMESTAMP, a date value, a space, and a time value. Thus, TIMESTAMP '1948-05-14 12:00:00' represents noon on May 14, 1948.

We can compare dates or times using the same comparison operators we use for numbers or strings. That is, < on dates means that the first date is earlier than the second; < on times means that the first is earlier (within the same day) than the second.

1.6 Null Values and Comparisons Involving NULL

SQL allows attributes to have a special value NULL, which is called the *null value*. There are many different interpretations that can be put on null values. Here are some of the most common:

1. *Value unknown*: that is, "I know there is some value that belongs here but I don't know what it is." An unknown birthdate is an example.

2. *Value inapplicable*: "There is no value that makes sense here." For example, if we had a `spouse` attribute for the `MovieStar` relation, then an unmarried star might have `NULL` for that attribute, not because we don't know the spouse's name, but because there is none.

3. *Value withheld*: "We are not entitled to know the value that belongs here." For instance, an unlisted phone number might appear as `NULL` in the component for a `phone` attribute.

Recall how the use of the outerjoin operator of relational algebra produces null values in some components of tuples; SQL allows outerjoins and also produces `NULL`'s when a query involves outerjoins; see Section 3.8. There are other ways SQL produces `NULL`'s as well. For example, certain insertions of tuples create null values, as we shall see in Section 5.1.

In `WHERE` clauses, we must be prepared for the possibility that a component of some tuple we are examining will be `NULL`. There are two important rules to remember when we operate upon a `NULL` value.

1. When we operate on a `NULL` and any value, including another `NULL`, using an arithmetic operator like $\times$ or $+$, the result is `NULL`.

2. When we compare a `NULL` value and any value, including another `NULL`, using a comparison operator like `=` or `>`, the result is `UNKNOWN`. The value `UNKNOWN` is another truth-value, like `TRUE` and `FALSE`; we shall discuss how to manipulate truth-value `UNKNOWN` shortly.

However, we must remember that, although `NULL` is a value that can appear in tuples, it is *not* a constant. Thus, while the above rules apply when we try to operate on an expression whose value is `NULL`, we cannot use `NULL` explicitly as an operand.

Example 9: Let x have the value `NULL`. Then the value of $x + 3$ is also `NULL`. However, `NULL` + 3 is not a legal SQL expression. Similarly, the value of $x = 3$ is `UNKNOWN`, because we cannot tell if the value of x, which is `NULL`, equals the value 3. However, the comparison `NULL` = 3 is not correct SQL. □

The correct way to ask if x has the value `NULL` is with the expression `x IS NULL`. This expression has the value `TRUE` if x has the value `NULL` and it has value `FALSE` otherwise. Similarly, `x IS NOT NULL` has the value `TRUE` unless the value of x is `NULL`.

1.7 The Truth-Value `UNKNOWN`

In Section 1.2 we assumed that the result of a comparison was either `TRUE` or `FALSE`, and these truth-values were combined in the obvious way using the logical operators `AND`, `OR`, and `NOT`. We have just seen that when `NULL` values

Pitfalls Regarding Nulls

It is tempting to assume that NULL in SQL can always be taken to mean "a value that we don't know but that surely exists." However, there are several ways that intuition is violated. For instance, suppose x is a component of some tuple, and the domain for that component is the integers. We might reason that $0 * x$ surely has the value 0, since no matter what integer x is, its product with 0 is 0. However, if x has the value NULL, rule (1) of Section 1.6 applies; the product of 0 and NULL is NULL. Similarly, we might reason that $x - x$ has the value 0, since whatever integer x is, its difference with itself is 0. However, again the rule about operations on nulls applies, and the result is NULL.

occur, comparisons can yield a third truth-value: UNKNOWN. We must now learn how the logical operators behave on combinations of all three truth-values.

The rule is easy to remember if we think of TRUE as 1 (i.e., fully true), FALSE as 0 (i.e., not at all true), and UNKNOWN as 1/2 (i.e., somewhere between true and false). Then:

1. The AND of two truth-values is the minimum of those values. That is, x AND y is FALSE if either x or y is FALSE; it is UNKNOWN if neither is FALSE but at least one is UNKNOWN, and it is TRUE only when both x and y are TRUE.

2. The OR of two truth-values is the maximum of those values. That is, x OR y is TRUE if either x or y is TRUE; it is UNKNOWN if neither is TRUE but at least one is UNKNOWN, and it is FALSE only when both are FALSE.

3. The negation of truth-value v is $1 - v$. That is, NOT x has the value TRUE when x is FALSE, the value FALSE when x is TRUE, and the value UNKNOWN when x has value UNKNOWN.

In Fig. 2 is a summary of the result of applying the three logical operators to the nine different combinations of truth-values for operands x and y. The value of the last operator, NOT, depends only on x.

SQL conditions, as appear in WHERE clauses of select-from-where statements, apply to each tuple in some relation, and for each tuple, one of the three truth values, TRUE, FALSE, or UNKNOWN is produced. However, only the tuples for which the condition has the value TRUE become part of the answer; tuples with either UNKNOWN or FALSE as value are excluded from the answer. That situation leads to another surprising behavior similar to that discussed in the box on "Pitfalls Regarding Nulls," as the next example illustrates.

Example 10: Suppose we ask about our running-example relation

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Figure 2: Truth table for three-valued logic

```
Movies(title, year, length, genre, studioName, producerC#)
```

the following query:

```
SELECT *
FROM Movies
WHERE length <= 120 OR length > 120;
```

Intuitively, we would expect to get a copy of the `Movies` relation, since each movie has a length that is either 120 or less or that is greater than 120.

However, suppose there are `Movies` tuples with `NULL` in the `length` component. Then both comparisons `length <= 120` and `length > 120` evaluate to `UNKNOWN`. The `OR` of two `UNKNOWN`'s is `UNKNOWN`, by Fig. 2. Thus, for any tuple with a `NULL` in the `length` component, the `WHERE` clause evaluates to `UNKNOWN`. Such a tuple is *not* returned as part of the answer to the query. As a result, the true meaning of the query is "find all the `Movies` tuples with non-`NULL` lengths." □

1.8 Ordering the Output

We may ask that the tuples produced by a query be presented in sorted order. The order may be based on the value of any attribute, with ties broken by the value of a second attribute, remaining ties broken by a third, and so on. To get output in sorted order, we may add to the select-from-where statement a clause:

```
ORDER BY <list of attributes>
```

The order is by default ascending, but we can get the output highest-first by appending the keyword `DESC` (for "descending") to an attribute. Similarly, we can specify ascending order with the keyword `ASC`, but that word is unnecessary.

The `ORDER BY` clause follows the `WHERE` clause and any other clauses (i.e., the optional `GROUP BY` and `HAVING` clauses, which are introduced in Section 4).

The ordering is performed on the result of the `FROM`, `WHERE`, and other clauses, just before we apply the `SELECT` clause. The tuples of this result are then sorted by the attributes in the list of the `ORDER BY` clause, and then passed to the `SELECT` clause for processing in the normal manner.

Example 11: The following is a rewrite of our original query of Example 1, asking for the Disney movies of 1990 from the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

To get the movies listed by length, shortest first, and among movies of equal length, alphabetically, we can say:

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

A subtlety of ordering is that all the attributes of `Movies` are available at the time of sorting, even if they are not part of the `SELECT` clause. Thus, we could replace `SELECT *` by `SELECT producerC#`, and the query would still be legal. □

An additional option in ordering is that the list following `ORDER BY` can include expressions, just as the `SELECT` clause can. For instance, we can order the tuples of a relation $R(A, B)$ by the sum of the two components of the tuples, highest first, with:

```
SELECT *
FROM R
ORDER BY A+B DESC;
```

1.9 Exercises for Section 1

Exercise 1.1: If a query has a `SELECT` clause

```
SELECT A B
```

how do we know whether A and B are two different attributes or B is an alias of A?

Exercise 1.2: Write the following queries, based on our running movie database example

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

in SQL.

a) Find the address of MGM studios.

b) Find Sandra Bullock's birthdate.

c) Find all the stars that appeared either in a movie made in 1980 or a movie with "Love" in the title.

d) Find all executives worth at least $10,000,000.

e) Find all the stars who either are male or live in Malibu (have string `Malibu` as a part of their address).

Exercise 1.3: Let a and b be integer-valued attributes that may be `NULL` in some tuples. For each of the following conditions (as may appear in a `WHERE` clause), describe exactly the set of (a, b) tuples that satisfy the condition, including the case where a and/or b is `NULL`.

a) `a = 10 OR b = 20`

b) `a = 10 AND b = 20`

c) `a < 10 OR a >= 10`

! d) `a = b`

! e) `a <= b`

! **Exercise 1.4:** In Example 10 we discussed the query

```
SELECT *
FROM Movies
WHERE length <= 120 OR length > 120;
```

which behaves unintuitively when the length of a movie is `NULL`. Find a simpler, equivalent query, one with a single condition in the `WHERE` clause (no `AND` or `OR` of conditions).

2 Queries Involving More Than One Relation

Much of the power of relational algebra comes from its ability to combine two or more relations through joins, products, unions, intersections, and differences. We get all of these operations in SQL. The set-theoretic operations — union, intersection, and difference — appear directly in SQL, as we shall learn in Section 2.5. First, we shall learn how the select-from-where statement of SQL allows us to perform products and joins.

2.1 Products and Joins in SQL

SQL has a simple way to couple relations in one query: list each relation in the `FROM` clause. Then, the `SELECT` and `WHERE` clauses can refer to the attributes of any of the relations in the `FROM` clause.

Example 12: Suppose we want to know the name of the producer of *Star Wars*. To answer this question we need the following two relations from our running example:

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

The producer certificate number is given in the `Movies` relation, so we can do a simple query on `Movies` to get this number. We could then do a second query on the relation `MovieExec` to find the name of the person with that certificate number.

However, we can phrase both these steps as one query about the pair of relations `Movies` and `MovieExec` as follows:

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

This query asks us to consider all pairs of tuples, one from `Movies` and the other from `MovieExec`. The conditions on this pair are stated in the `WHERE` clause:

1. The `title` component of the tuple from `Movies` must have value `'Star Wars'`.

2. The `producerC#` attribute of the `Movies` tuple must be the same certificate number as the `cert#` attribute in the `MovieExec` tuple. That is, these two tuples must refer to the same producer.

Whenever we find a pair of tuples satisfying both conditions, we produce the `name` attribute of the tuple from `MovieExec` as part of the answer. If the data is what we expect, the only time both conditions will be met is when the tuple from `Movies` is for *Star Wars*, and the tuple from `MovieExec` is for George Lucas. Then and only then will the title be correct and the certificate numbers agree. Thus, `George Lucas` should be the only value produced. This process is suggested in Fig. 3. We take up in more detail how to interpret multirelation queries in Section 2.4. □

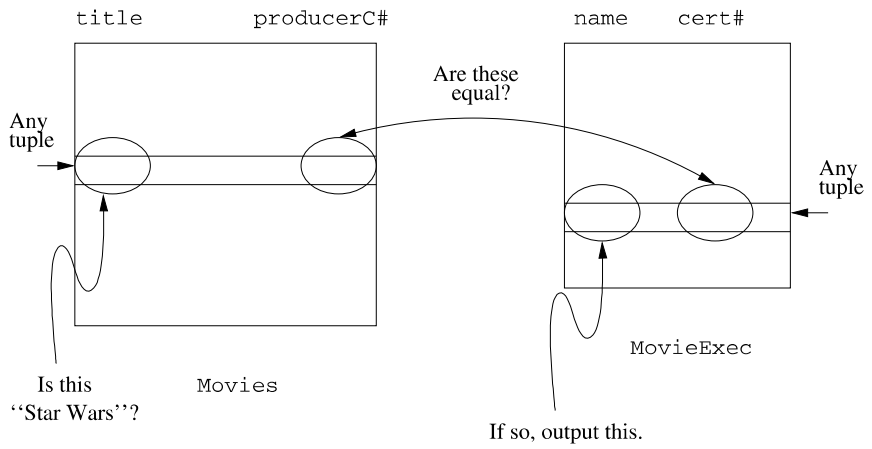

Figure 3: The query of Example 12 asks us to pair every tuple of `Movies` with every tuple of `MovieExec` and test two conditions

2.2 Disambiguating Attributes

Sometimes we ask a query involving several relations, and among these relations are two or more attributes with the same name. If so, we need a way to indicate which of these attributes is meant by a use of their shared name. SQL solves this problem by allowing us to place a relation name and a dot in front of an attribute. Thus $R.A$ refers to the attribute A of relation R.

Example 13: The two relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

each have attributes `name` and `address`. Suppose we wish to find pairs consisting of a star and an executive with the same address. The following query does the job.

```
SELECT MovieStar.name, MovieExec.name
FROM MovieStar, MovieExec
WHERE MovieStar.address = MovieExec.address;
```

In this query, we look for a pair of tuples, one from `MovieStar` and the other from `MovieExec`, such that their address components agree. The `WHERE` clause enforces the requirement that the `address` attributes from each of the two tuples agree. Then, for each matching pair of tuples, we extract the two `name` attributes, first from the `MovieStar` tuple and then from the other. The result would be a set of pairs such as

MovieStar.name	*MovieExec.name*
Jane Fonda	Ted Turner
...	...

□

The relation name, followed by a dot, is permissible even in situations where there is no ambiguity. For instance, we are free to write the query of Example 12 as

```
SELECT MovieExec.name
FROM Movies, MovieExec
WHERE Movie.title = 'Star Wars'
      AND Movie.producerC# = MovieExec.cert#;
```

Alternatively, we may use relation names and dots in front of any subset of the attributes in this query.

2.3 Tuple Variables

Disambiguating attributes by prefixing the relation name works as long as the query involves combining several different relations. However, sometimes we need to ask a query that involves two or more tuples from the same relation. We may list a relation R as many times as we need to in the `FROM` clause, but we need a way to refer to each occurrence of R. SQL allows us to define, for each occurrence of R in the `FROM` clause, an "alias" which we shall refer to as a *tuple variable*. Each use of R in the `FROM` clause is followed by the (optional) keyword `AS` and the name of the tuple variable; we shall generally omit the `AS` in this context.

In the `SELECT` and `WHERE` clauses, we can disambiguate attributes of R by preceding them by the appropriate tuple variable and a dot. Thus, the tuple variable serves as another name for relation R and can be used in its place when we wish.

Example 14: While Example 13 asked for a star and an executive sharing an address, we might similarly want to know about two stars who share an address. The query is essentially the same, but now we must think of two tuples chosen from relation `MovieStar`, rather than tuples from each of `MovieStar` and `MovieExec`. Using tuple variables as aliases for two uses of `MovieStar`, we can write the query as

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address
      AND Star1.name < Star2.name;
```

Tuple Variables and Relation Names

Technically, references to attributes in `SELECT` and `WHERE` clauses are *always* to a tuple variable. However, if a relation appears only once in the `FROM` clause, then we can use the relation name as its own tuple variable. Thus, we can see a relation name R in the `FROM` clause as shorthand for R `AS` R. Furthermore, as we have seen, when an attribute belongs unambiguously to one relation, the relation name (tuple variable) may be omitted.

We see in the `FROM` clause the declaration of two tuple variables, `Star1` and `Star2`; each is an alias for relation `MovieStar`. The tuple variables are used in the `SELECT` clause to refer to the `name` components of the two tuples. These aliases are also used in the `WHERE` clause to say that the two `MovieStar` tuples represented by `Star1` and `Star2` have the same value in their `address` components.

The second condition in the `WHERE` clause, `Star1.name < Star2.name`, says that the name of the first star precedes the name of the second star alphabetically. If this condition were omitted, then tuple variables `Star1` and `Star2` could both refer to the same tuple. We would find that the two tuple variables referred to tuples whose `address` components are equal, of course, and thus produce each star name paired with itself.[3] The second condition also forces us to produce each pair of stars with a common address only once, in alphabetical order. If we used `<>` (not-equal) as the comparison operator, then we would produce pairs of married stars twice, like

Star1.name	*Star2.name*
Paul Newman	Joanne Woodward
Joanne Woodward	Paul Newman
...	...

□

2.4 Interpreting Multirelation Queries

There are several ways to define the meaning of the select-from-where expressions that we have just covered. All are *equivalent*, in the sense that they each give the same answer for each query applied to the same relation instances. We shall consider each in turn.

[3] A similar problem occurs in Example 13 when the same individual is both a star and an executive. We could solve that problem by requiring that the two names be unequal.

Nested Loops

The semantics that we have implicitly used in examples so far is that of tuple variables. Recall that a tuple variable ranges over all tuples of the corresponding relation. A relation name that is not aliased is also a tuple variable ranging over the relation itself, as we mentioned in the box on "Tuple Variables and Relation Names." If there are several tuple variables, we may imagine nested loops, one for each tuple variable, in which the variables each range over the tuples of their respective relations. For each assignment of tuples to the tuple variables, we decide whether the `WHERE` clause is true. If so, we produce a tuple consisting of the values of the expressions following `SELECT`; note that each term is given a value by the current assignment of tuples to tuple variables. This query-answering algorithm is suggested by Fig. 4.

```
LET the tuple variables in the from-clause range over
        relations R1, R2, ..., Rn;
FOR each tuple t1 in relation R1 DO
    FOR each tuple t2 in relation R2 DO
            ...
        FOR each tuple tn in relation Rn DO
            IF the where-clause is satisfied when the values
            from t1, t2, ..., tn are substituted for all
            attribute references THEN
                evaluate the expressions of the select-clause
                according to t1, t2, ..., tn and produce the
                tuple of values that results.
```

Figure 4: Answering a simple SQL query

Parallel Assignment

There is an equivalent definition in which we do not explicitly create nested loops ranging over the tuple variables. Rather, we consider in arbitrary order, or in parallel, all possible assignments of tuples from the appropriate relations to the tuple variables. For each such assignment, we consider whether the `WHERE` clause becomes true. Each assignment that produces a true `WHERE` clause contributes a tuple to the answer; that tuple is constructed from the attributes of the `SELECT` clause, evaluated according to that assignment.

Conversion to Relational Algebra

A third approach is to relate the SQL query to relational algebra. We start with the tuple variables in the `FROM` clause and take the Cartesian product of their relations. If two tuple variables refer to the same relation, then this relation appears twice in the product, and we rename its attributes so all attributes have

An Unintuitive Consequence of SQL Semantics

Suppose R, S, and T are unary (one-component) relations, each having attribute A alone, and we wish to find those elements that are in R and also in either S or T (or both). That is, we want to compute $R \cap (S \cup T)$. We might expect the following SQL query would do the job.

```
SELECT R.A
FROM R, S, T
WHERE R.A = S.A OR R.A = T.A;
```

However, consider the situation in which T is empty. Since then $R.A = T.A$ can never be satisfied, we might expect the query to produce exactly $R \cap S$, based on our intuition about how "OR" operates. Yet whichever of the three equivalent definitions of Section 2.4 one prefers, we find that the result is empty, regardless of how many elements R and S have in common. If we use the nested-loop semantics of Figure 4, then we see that the loop for tuple variable T iterates 0 times, since there are no tuples in the relation for the tuple variable to range over. Thus, the if-statement inside the for-loops never executes, and nothing can be produced. Similarly, if we look for assignments of tuples to the tuple variables, there is no way to assign a tuple to T, so no assignments exist. Finally, if we use the Cartesian-product approach, we start with $R \times S \times T$, which is empty because T is empty.

unique names. Similarly, attributes of the same name from different relations are renamed to avoid ambiguity.

Having created the product, we apply a selection operator to it by converting the `WHERE` clause to a selection condition in the obvious way. That is, each attribute reference in the `WHERE` clause is replaced by the attribute of the product to which it corresponds. Finally, we create from the `SELECT` clause a list of expressions for a final (extended) projection operation. As we did for the `WHERE` clause, we interpret each attribute reference in the `SELECT` clause as the corresponding attribute in the product of relations.

Example 15: Let us convert the query of Example 14 to relational algebra. First, there are two tuple variables in the `FROM` clause, both referring to relation `MovieStar`. Thus, our expression (without the necessary renaming) begins:

$$\texttt{MovieStar} \times \texttt{MovieStar}$$

The resulting relation has eight attributes, the first four correspond to attributes `name`, `address`, `gender`, and `birthdate` from the first copy of relation `MovieStar`, and the second four correspond to the same attributes from the

other copy of `MovieStar`. We could create names for these attributes with a dot and the aliasing tuple variable — e.g., `Star1.gender` — but for succinctness, let us invent new symbols and call the attributes simply $A_1, A_2, \ldots, A_8$. Thus, A_1 corresponds to `Star1.name`, A_5 corresponds to `Star2.name`, and so on.

Under this naming strategy for attributes, the selection condition obtained from the `WHERE` clause is $A_2 = A_6$ and $A_1 < A_5$. The projection list is A_1, A_5. Thus,

$$\pi_{A_1,A_5}\Big(\sigma_{A_2=A_6 \text{ AND } A_1<A_5}\big(\rho_{M(A_1,A_2,A_3,A_4)}(\texttt{MovieStar}) \times \rho_{N(A_5,A_6,A_7,A_8)}(\texttt{MovieStar})\big)\Big)$$

renders the entire query in relational algebra. □

2.5 Union, Intersection, and Difference of Queries

Sometimes we wish to combine relations using the set operations of relational algebra: union, intersection, and difference. SQL provides corresponding operators that apply to the results of queries, provided those queries produce relations with the same list of attributes and attribute types. The keywords used are `UNION`, `INTERSECT`, and `EXCEPT` for $\cup$, $\cap$, and $-$, respectively. Words like `UNION` are used between two queries, and those queries must be parenthesized.

Example 16: Suppose we wanted the names and addresses of all female movie stars who are also movie executives with a net worth over $10,000,000. Using the following two relations:

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

we can write the query as in Fig. 5. Lines (1) through (3) produce a relation whose schema is `(name, address)` and whose tuples are the names and addresses of all female movie stars.

```
1)  (SELECT name, address
2)   FROM MovieStar
3)   WHERE gender = 'F')
4)       INTERSECT
5)  (SELECT name, address
6)   FROM MovieExec
7)   WHERE netWorth > 10000000);
```

Figure 5: Intersecting female movie stars with rich executives

Similarly, lines (5) through (7) produce the set of "rich" executives, those with net worth over $10,000,000. This query also yields a relation whose schema

Readable SQL Queries

Generally, one writes SQL queries so that each important keyword like `FROM` or `WHERE` starts a new line. This style offers the reader visual clues to the structure of the query. However, when a query or subquery is short, we shall sometimes write it out on a single line, as we did in Example 17. That style, keeping a complete query compact, also offers good readability.

has the attributes `name` and `address` only. Since the two schemas are the same, we can intersect them, and we do so with the operator of line (4). □

Example 17: In a similar vein, we could take the difference of two sets of persons, each selected from a relation. The query

```
(SELECT name, address FROM MovieStar)
    EXCEPT
(SELECT name, address FROM MovieExec);
```

gives the names and addresses of movie stars who are not also movie executives, regardless of gender or net worth. □

In the two examples above, the attributes of the relations whose intersection or difference we took were conveniently the same. However, if necessary to get a common set of attributes, we can rename attributes as in Example 3.

Example 18: Suppose we wanted all the titles and years of movies that appeared in either the `Movies` or `StarsIn` relation of our running example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

Ideally, these sets of movies would be the same, but in practice it is common for relations to diverge; for instance we might have movies with no listed stars or a `StarsIn` tuple that mentions a movie not found in the `Movies` relation.[4] Thus, we might write

```
(SELECT title, year FROM Movie)
    UNION
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

The result would be all movies mentioned in either relation, with `title` and `year` as the attributes of the resulting relation. □

[4] There are ways to prevent this divergence.

2.6 Exercises for Section 2

Exercise 2.1: Using the database schema of our running movie example

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

write the following queries in SQL.

a) Who were the male stars in *Titanic*?

b) Which stars appeared in movies produced by MGM in 1995?

c) Who is the president of MGM studios?

! d) Which movies are longer than *Gone With the Wind*?

! e) Which executives are worth more than Merv Griffin?

! Exercise 2.2: A general form of relational-algebra query is

$$\pi_L\Big(\sigma_C(R_1 \times R_2 \times \cdots \times R_n)\Big)$$

Here, L is an arbitrary list of attributes, and C is an arbitrary condition. The list of relations $R_1, R_2, \ldots, R_n$ may include the same relation repeated several times, in which case appropriate renaming may be assumed applied to the R_i's. Show how to express any query of this form in SQL.

! Exercise 2.3: Another general form of relational-algebra query is

$$\pi_L\Big(\sigma_C(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n)\Big)$$

The same assumptions as in Exercise 2.2 apply here; the only difference is that the natural join is used instead of the product. Show how to express any query of this form in SQL.

3 Subqueries

In SQL, one query can be used in various ways to help in the evaluation of another. A query that is part of another is called a *subquery*. Subqueries can have subqueries, and so on, down as many levels as we wish. We already saw one example of the use of subqueries; in Section 2.5 we built a union, intersection, or difference query by connecting two subqueries to form the whole query. There are a number of other ways that subqueries can be used:

1. Subqueries can return a single constant, and this constant can be compared with another value in a `WHERE` clause.

2. Subqueries can return relations that can be used in various ways in `WHERE` clauses.

3. Subqueries can appear in `FROM` clauses, followed by a tuple variable that represents the tuples in the result of the subquery.

3.1 Subqueries that Produce Scalar Values

An atomic value that can appear as one component of a tuple is referred to as a *scalar*. A select-from-where expression can produce a relation with any number of attributes in its schema, and there can be any number of tuples in the relation. However, often we are only interested in values of a single attribute. Furthermore, sometimes we can deduce from information about keys, or from other information, that there will be only a single value produced for that attribute.

If so, we can use this select-from-where expression, surrounded by parentheses, as if it were a constant. In particular, it may appear in a `WHERE` clause any place we would expect to find a constant or an attribute representing a component of a tuple. For instance, we may compare the result of such a subquery to a constant or attribute.

Example 19: Let us recall Example 12, where we asked for the producer of *Star Wars*. We had to query the two relations

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

because only the former has movie title information and only the latter has producer names. The information is linked by "certificate numbers." These numbers uniquely identify producers. The query we developed is:

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

There is another way to look at this query. We need the `Movies` relation only to get the certificate number for the producer of *Star Wars*. Once we have it, we can query the relation `MovieExec` to find the name of the person with this certificate. The first problem, getting the certificate number, can be written as a subquery, and the result, which we expect will be a single value, can be used in the "main" query to achieve the same effect as the query above. This query is shown in Fig. 6.

Lines (4) through (6) of Fig. 6 are the subquery. Looking only at this simple query by itself, we see that the result will be a unary relation with attribute

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# =
4)      (SELECT producerC#
5)       FROM Movies
6)       WHERE title = 'Star Wars'
        );
```

Figure 6: Finding the producer of *Star Wars* by using a nested subquery

`producerC#`, and we expect to find only one tuple in this relation. The tuple will look like (12345), that is, a single component with some integer, perhaps 12345 or whatever George Lucas' certificate number is. If zero tuples or more than one tuple is produced by the subquery of lines (4) through (6), it is a run-time error.

Having executed this subquery, we can then execute lines (1) through (3) of Fig. 6, as if the value 12345 replaced the entire subquery. That is, the "main" query is executed as if it were

```
SELECT name
FROM MovieExec
WHERE cert# = 12345;
```

The result of this query should be `George Lucas`. □

3.2 Conditions Involving Relations

There are a number of SQL operators that we can apply to a relation R and produce a boolean result. However, the relation R must be expressed as a subquery. As a trick, if we want to apply these operators to a stored table `Foo`, we can use the subquery `(SELECT * FROM Foo)`. The same trick works for union, intersection, and difference of relations. Notice that those operators, introduced in Section 2.5 are applied to two subqueries.

Some of the operators below — `IN`, `ALL`, and `ANY` — will be explained first in their simple form where a scalar value s is involved. In this situation, the subquery R is required to produce a one-column relation. Here are the definitions of the operators:

1. `EXISTS` R is a condition that is true if and only if R is not empty.

2. s `IN` R is true if and only if s is equal to one of the values in R. Likewise, s `NOT IN` R is true if and only if s is equal to no value in R. Here, we assume R is a unary relation. We shall discuss extensions to the `IN` and `NOT IN` operators where R has more than one attribute in its schema and s is a tuple in Section 3.3.

3. s `>` `ALL` R is true if and only if s is greater than every value in unary relation R. Similarly, the `>` operator could be replaced by any of the other five comparison operators, with the analogous meaning: s stands in the stated relationship to every tuple in R. For instance, s `<>` `ALL` R is the same as s `NOT IN` R.

4. s `>` `ANY` R is true if and only if s is greater than at least one value in unary relation R. Similarly, any of the other five comparisons could be used in place of `>`, with the meaning that s stands in the stated relationship to at least one tuple of R. For instance, s `=` `ANY` R is the same as s `IN` R.

The `EXISTS`, `ALL`, and `ANY` operators can be negated by putting `NOT` in front of the entire expression, just like any other boolean-valued expression. Thus, `NOT EXISTS` R is true if and only if R is empty. `NOT` s `>=` `ALL` R is true if and only if s is not the maximum value in R, and `NOT` s `>` `ANY` R is true if and only if s is the minimum value in R. We shall see several examples of the use of these operators shortly.

3.3 Conditions Involving Tuples

A tuple in SQL is represented by a parenthesized list of scalar values. Examples are `(123, 'foo')` and `(name, address, networth)`. The first of these has constants as components; the second has attributes as components. Mixing of constants and attributes is permitted.

If a tuple t has the same number of components as a relation R, then it makes sense to compare t and R in expressions of the type listed in Section 3.2. Examples are t `IN` R or t `<>` `ANY` R. The latter comparison means that there is some tuple in R other than t. Note that when comparing a tuple with members of a relation R, we must compare components using the assumed standard order for the attributes of R.

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# IN
4)      (SELECT producerC#
5)       FROM Movies
6)       WHERE (title, year) IN
7)           (SELECT movieTitle, movieYear
8)            FROM StarsIn
9)            WHERE starName = 'Harrison Ford'
             )
        );
```

Figure 7: Finding the producers of Harrison Ford's movies

Example 20: In Fig. 7 is a SQL query on the three relations

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
```

asking for all the producers of movies in which Harrison Ford stars. It consists of a "main" query, a query nested within that, and a third query nested within the second.

We should analyze any query with subqueries from the inside out. Thus, let us start with the innermost nested subquery: lines (7) through (9). This query examines the tuples of the relation `StarsIn` and finds all those tuples whose `starName` component is `'Harrison Ford'`. The titles and years of those movies are returned by this subquery. Recall that title and year, not title alone, is the key for movies, so we need to produce tuples with both attributes to identify a movie uniquely. Thus, we would expect the value produced by lines (7) through (9) to look something like Fig. 8.

title	*year*
Star Wars	1977
Raiders of the Lost Ark	1981
The Fugitive	1993
...	...

Figure 8: Title-year pairs returned by inner subquery

Now, consider the middle subquery, lines (4) through (6). It searches the `Movies` relation for tuples whose title and year are in the relation suggested by Fig. 8. For each tuple found, the producer's certificate number is returned, so the result of the middle subquery is the set of certificates of the producers of Harrison Ford's movies.

Finally, consider the "main" query of lines (1) through (3). It examines the tuples of the `MovieExec` relation to find those whose `cert#` component is one of the certificates in the set returned by the middle subquery. For each of these tuples, the name of the producer is returned, giving us the set of producers of Harrison Ford's movies, as desired. □

Incidentally, the nested query of Fig. 7 can, like many nested queries, be written as a single select-from-where expression with relations in the `FROM` clause for each of the relations mentioned in the main query or a subquery. The `IN` relationships are replaced by equalities in the `WHERE` clause. For instance, the query of Fig. 9 is essentially that of Fig. 7. There is a difference regarding the way duplicate occurrences of a producer — e.g., George Lucas — are handled, as we shall discuss in Section 4.1.

```
SELECT name
FROM MovieExec, Movies, StarsIn
WHERE cert# = producerC# AND
      title = movieTitle AND
      year = movieYear AND
      starName = 'Harrison Ford';
```

Figure 9: Ford's producers without nested subqueries

3.4 Correlated Subqueries

The simplest subqueries can be evaluated once and for all, and the result used in a higher-level query. A more complicated use of nested subqueries requires the subquery to be evaluated many times, once for each assignment of a value to some term in the subquery that comes from a tuple variable outside the subquery. A subquery of this type is called a *correlated* subquery. Let us begin our study with an example.

Example 21: We shall find the titles that have been used for two or more movies. We start with an outer query that looks at all tuples in the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

For each such tuple, we ask in a subquery whether there is a movie with the same title and a greater year. The entire query is shown in Fig. 10.

As with other nested queries, let us begin at the innermost subquery, lines (4) through (6). If `Old.title` in line (6) were replaced by a constant string such as `'King Kong'`, we would understand it quite easily as a query asking for the year or years in which movies titled *King Kong* were made. The present subquery differs little. The only problem is that we don't know what value `Old.title` has. However, as we range over `Movies` tuples of the outer query of lines (1) through (3), each tuple provides a value of `Old.title`. We then execute the query of lines (4) through (6) with this value for `Old.title` to decide the truth of the `WHERE` clause that extends from lines (3) through (6).

```
1)  SELECT title
2)  FROM Movies Old
3)  WHERE year < ANY
4)      (SELECT year
5)       FROM Movies
6)       WHERE title = Old.title
        );
```

Figure 10: Finding movie titles that appear more than once

The condition of line (3) is true if any movie with the same title as `Old.title` has a later year than the movie in the tuple that is the current value of tuple variable `Old`. This condition is true unless the year in the tuple `Old` is the last year in which a movie of that title was made. Consequently, lines (1) through (3) produce a title one fewer times than there are movies with that title. A movie made twice will be listed once, a movie made three times will be listed twice, and so on.[5] □

When writing a correlated query it is important that we be aware of the *scoping rules* for names. In general, an attribute in a subquery belongs to one of the tuple variables in that subquery's `FROM` clause if some tuple variable's relation has that attribute in its schema. If not, we look at the immediately surrounding subquery, then to the one surrounding that, and so on. Thus, `year` on line (4) and `title` on line (6) of Fig. 10 refer to the attributes of the tuple variable that ranges over all the tuples of the copy of relation `Movies` introduced on line (5) — that is, the copy of the `Movies` relation addressed by the subquery of lines (4) through (6).

However, we can arrange for an attribute to belong to another tuple variable if we prefix it by that tuple variable and a dot. That is why we introduced the alias `Old` for the `Movies` relation of the outer query, and why we refer to `Old.title` in line (6). Note that if the two relations in the `FROM` clauses of lines (2) and (5) were different, we would not need an alias. Rather, in the subquery we could refer directly to attributes of a relation mentioned in line (2).

3.5 Subqueries in `FROM` Clauses

Another use for subqueries is as relations in a `FROM` clause. In a `FROM` list, instead of a stored relation, we may use a parenthesized subquery. Since we don't have a name for the result of this subquery, we must give it a tuple-variable alias. We then refer to tuples in the result of the subquery as we would tuples in any relation that appears in the `FROM` list.

Example 22: Let us reconsider the problem of Example 20, where we wrote a query that finds the producers of Harrison Ford's movies. Suppose we had a relation that gave the certificates of the producers of those movies. It would then be a simple matter to look up the names of those producers in the relation `MovieExec`. Figure 11 is such a query.

Lines (2) through (7) are the `FROM` clause of the outer query. In addition to the relation `MovieExec`, it has a subquery. That subquery joins `Movies` and `StarsIn` on lines (3) through (5), adds the condition that the star is Harrison Ford on line (6), and returns the set of producers of the movies at line (2). This set is given the alias `Prod` on line (7).

[5] This example is the first occasion on which we've been reminded that relations in SQL are bags, not sets. There are several ways that duplicates may crop up in SQL relations. We shall discuss the matter in detail in Section 4.

```
1)  SELECT name
2)  FROM MovieExec, (SELECT producerC#
3)                   FROM Movies, StarsIn
4)                   WHERE title = movieTitle AND
5)                         year = movieYear AND
6)                         starName = 'Harrison Ford'
7)                  ) Prod
8)  WHERE cert# = Prod.producerC#;
```

Figure 11: Finding the producers of Ford's movies using a subquery in the `FROM` clause

At line (8), the relations `MovieExec` and the subquery aliased `Prod` are joined with the requirement that the certificate numbers be the same. The names of the producers from `MovieExec` that have certificates in the set aliased by `Prod` is returned at line (1). □

3.6 SQL Join Expressions

We can construct relations by a number of variations on the join operator applied to two relations. These variants include products, natural joins, theta-joins, and outerjoins. The result can stand as a query by itself. Alternatively, all these expressions, since they produce relations, may be used as subqueries in the `FROM` clause of a select-from-where expression. These expressions are principally shorthands for more complex select-from-where queries (see Exercise 3.9).

The simplest form of join expression is a *cross join*. For instance, if we want the product of the two relations

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

we can say

```
Movies CROSS JOIN StarsIn;
```

and the result will be a nine-column relation with all the attributes of `Movies` and `StarsIn`. Every pair consisting of one tuple of `Movies` and one tuple of `StarsIn` will be a tuple of the resulting relation.

The attributes in the product relation can be called $R.A$, where R is one of the two joined relations and A is one of its attributes. If only one of the relations has an attribute named A, then the R and dot can be dropped, as usual. In this instance, since `Movies` and `StarsIn` have no common attributes, the nine attribute names suffice in the product.

However, the product by itself is rarely a useful operation. A more conventional theta-join is obtained with the keyword `ON`. We put `JOIN` between two

relation names R and S and follow them by `ON` and a condition. The meaning of `JOIN...ON` is that the product of $R \times S$ is followed by a selection for whatever condition follows `ON`.

Example 23: Suppose we want to join the relations

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

with the condition that the only tuples to be joined are those that refer to the same movie. That is, the titles and years from both relations must be the same. We can ask this query by

```
Movies JOIN StarsIn ON
        title = movieTitle AND year = movieYear;
```

The result is again a nine-column relation with the obvious attribute names. However, now a tuple from `Movies` and one from `StarsIn` combine to form a tuple of the result only if the two tuples agree on both the title and year. As a result, two of the columns are redundant, because every tuple of the result will have the same value in both the `title` and `movieTitle` components and will have the same value in both `year` and `movieYear`.

If we are concerned with the fact that the join above has two redundant components, we can use the whole expression as a subquery in a `FROM` clause and use a `SELECT` clause to remove the undesired attributes. Thus, we could write

```
SELECT title, year, length, genre, studioName,
        producerC#, starName
FROM Movies JOIN StarsIn ON
        title = movieTitle AND year = movieYear;
```

to get a seven-column relation which is the `Movies` relation's tuples, each extended in all possible ways with a star of that movie. □

3.7 Natural Joins

A natural join differs from a theta-join in that:

1. The join condition is that all pairs of attributes from the two relations having a common name are equated, and there are no other conditions.
2. One of each pair of equated attributes is projected out.

The SQL natural join behaves exactly this way. Keywords `NATURAL JOIN` appear between the relations to express the $\bowtie$ operator.

Example 24: Suppose we want to compute the natural join of the relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

The result will be a relation whose schema includes attributes `name` and `address` plus all the attributes that appear in one or the other of the two relations. A tuple of the result will represent an individual who is both a star and an executive and will have all the information pertinent to either: a name, address, gender, birthdate, certificate number, and net worth. The expression

```
MovieStar NATURAL JOIN MovieExec;
```

succinctly describes the desired relation. □

3.8 Outerjoins

The outerjoin operator is a way to augment the result of a join by the dangling tuples, padded with null values. In SQL, we can specify an outerjoin; `NULL` is used as the null value.

Example 25: Suppose we wish to take the outerjoin of the two relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

SQL refers to the standard outerjoin, which pads dangling tuples from both of its arguments, as a *full* outerjoin. The syntax is unsurprising:

```
MovieStar NATURAL FULL OUTER JOIN MovieExec;
```

The result of this operation is a relation with the same six-attribute schema as Example 24. The tuples of this relation are of three kinds. Those representing individuals who are both stars and executives have tuples with all six attributes non-`NULL`. These are the tuples that are also in the result of Example 24.

The second kind of tuple is one for an individual who is a star but not an executive. These tuples have values for attributes `name`, `address`, `gender`, and `birthdate` taken from their tuple in `MovieStar`, while the attributes belonging only to `MovieExec`, namely `cert#` and `netWorth`, have `NULL` values.

The third kind of tuple is for an executive who is not also a star. These tuples have values for the attributes of `MovieExec` taken from their `MovieExec` tuple and `NULL`'s in the attributes `gender` and `birthdate` that come only from `MovieStar`. For instance, the three tuples of the result relation shown in Fig. 12 correspond to the three types of individuals, respectively. □

Variations on the outerjoin are also available in SQL. If we want a left- or right-outerjoin, we add the appropriate word `LEFT` or `RIGHT` in place of `FULL`. For instance,

```
MovieStar NATURAL LEFT OUTER JOIN MovieExec;
```

name	*address*	*gender*	*birthdate*	*cert#*	*networth*
Mary Tyler Moore	Maple St.	'F'	9/9/99	12345	100\cdots$
Tom Hanks	Cherry Ln.	'M'	8/8/88	NULL	NULL
George Lucas	Oak Rd.	NULL	NULL	23456	200\cdots$

Figure 12: Three tuples in the outerjoin of `MovieStar` and `MovieExec`

would yield the first two tuples of Fig. 12 but not the third. Similarly,

```
MovieStar NATURAL RIGHT OUTER JOIN MovieExec;
```

would yield the first and third tuples of Fig. 12 but not the second.

Next, suppose we want a theta-outerjoin instead of a natural outerjoin. Instead of using the keyword `NATURAL`, we may follow the join by `ON` and a condition that matching tuples must obey. If we also specify `FULL OUTER JOIN`, then after matching tuples from the two joined relations, we pad dangling tuples of either relation with `NULL`'s and include the padded tuples in the result.

Example 26: Let us reconsider Example 23, where we joined the relations `Movies` and `StarsIn` using the conditions that the `title` and `movieTitle` attributes of the two relations agree and that the `year` and `movieYear` attributes of the two relations agree. If we modify that example to call for a full outerjoin:

```
Movies FULL OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

then we shall get not only tuples for movies that have at least one star mentioned in `StarsIn`, but we shall get tuples for movies with no listed stars, padded with `NULL`'s in attributes `movieTitle`, `movieYear`, and `starName`. Likewise, for stars not appearing in any movie listed in relation `Movies` we get a tuple with `NULL`'s in the six attributes of `Movies`. □

The keyword `FULL` can be replaced by either `LEFT` or `RIGHT` in outerjoins of the type suggested by Example 26. For instance,

```
Movies LEFT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

gives us the `Movies` tuples with at least one listed star and `NULL`-padded `Movies` tuples without a listed star, but will not include stars without a listed movie. Conversely,

```
Movies RIGHT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

will omit the tuples for movies without a listed star but will include tuples for stars not in any listed movies, padded with `NULL`'s.

3.9 Exercises for Section 3

! Exercise 3.1 : Write the query of Fig. 10 without any subqueries.

! Exercise 3.2: Consider expression $\pi_L(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n)$ of relational algebra, where L is a list of attributes all of which belong to R_1. Show that this expression can be written in SQL using subqueries only. More precisely, write an equivalent SQL expression where no `FROM` clause has more than one relation in its list.

! Exercise 3.3: Write the following queries without using the intersection or difference operators:

a) The intersection query of Fig. 5.

b) The difference query of Example 17.

!! Exercise 3.4: We have noticed that certain operators of SQL are redundant, in the sense that they always can be replaced by other operators. For example, we saw that s `IN` R can be replaced by s `= ANY` R. Show that `EXISTS` and `NOT EXISTS` are redundant by explaining how to replace any expression of the form `EXISTS` R or `NOT EXISTS` R by an expression that does not involve `EXISTS` (except perhaps in the expression R itself). *Hint*: Remember that it is permissible to have a constant in the `SELECT` clause.

Exercise 3.5: For these relations from our running movie database schema

```
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

describe the tuples that would appear in the following SQL expressions:

a) `Studio CROSS JOIN MovieExec;`

b) `StarsIn NATURAL FULL OUTER JOIN MovieStar;`

c) `StarsIn FULL OUTER JOIN MovieStar ON name = starName;`

! Exercise 3.6: Using the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

write a SQL query that will produce information about all products — PC's, laptops, and printers — including their manufacturer if available, and whatever information about that product is relevant (i.e., found in the relation for that type of product).

Exercise 3.7: Using the two relations

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
```

write a SQL query that will produce all available information about ships, including that information available in the `Classes` relation. You need not produce information about classes if there are no ships of that class mentioned in `Ships`.

! Exercise 3.8: Repeat Exercise 3.7, but also include in the result, for any class C that is not mentioned in `Ships`, information about the ship that has the same name C as its class. You may assume that there is a ship with the class name, even if it doesn't appear in `Ships`.

! Exercise 3.9: The join operators (other than outerjoin) we learned in this section are redundant, in the sense that they can always be replaced by select-from-where expressions. Explain how to write expressions of the following forms using select-from-where:

a) `R CROSS JOIN S;`

b) `R NATURAL JOIN S;`

c) `R JOIN S ON` C`;`, where C is a SQL condition.

4 Full-Relation Operations

In this section we shall study some operations that act on relations as a whole, rather than on tuples individually or in small numbers (as do joins of several relations, for instance). First, we deal with the fact that SQL uses relations that are bags rather than sets, and a tuple can appear more than once in a relation. We shall see how to force the result of an operation to be a set in Section 4.1, and in Section 4.2 we shall see that it is also possible to prevent the elimination of duplicates in circumstances where SQL systems would normally eliminate them.

Then, we discuss how SQL supports the grouping and aggregation operator γ. SQL has aggregation operators and a `GROUP-BY` clause. There is also a "`HAVING`" clause that allows selection of certain groups in a way that depends on the group as a whole, rather than on individual tuples.

4.1 Eliminating Duplicates

As mentioned in Section 3.4, SQL's notion of relations differs from an abstract notion of relations. A relation, being a set, cannot have more than one copy of any given tuple. When a SQL query creates a new relation, the SQL system does not ordinarily eliminate duplicates. Thus, the SQL response to a query may list the same tuple several times.

Recall from Section 2.4 that one of several equivalent definitions of the meaning of a SQL select-from-where query is that we begin with the Cartesian product of the relations referred to in the `FROM` clause. Each tuple of the product is tested by the condition in the `WHERE` clause, and the ones that pass the test are given to the output for projection according to the `SELECT` clause. This projection may cause the same tuple to result from different tuples of the product, and if so, each copy of the resulting tuple is printed in its turn. Further, since there is nothing wrong with a SQL relation having duplicates, the relations from which the Cartesian product is formed may have duplicates, and each identical copy is paired with the tuples from the other relations, yielding a proliferation of duplicates in the product.

If we do not wish duplicates in the result, then we may follow the keyword `SELECT` by the keyword `DISTINCT`. That word tells SQL to produce only one copy of any tuple and is the SQL analog of applying the δ operator to the result of the query.

Example 27: Let us reconsider the query of Fig. 9, where we asked for the producers of Harrison Ford's movies using no subqueries. As written, George Lucas will appear many times in the output. If we want only to see each producer once, we may change line (1) of the query to

```
1)  SELECT DISTINCT name
```

Then, the list of producers will have duplicate occurrences of names eliminated before printing.

Incidentally, the query of Fig. 7, where we used subqueries, does not necessarily suffer from the problem of duplicate answers. True, the subquery at line (4) of Fig. 7 will produce the certificate number of George Lucas several times. However, in the "main" query of line (1), we examine each tuple of `MovieExec` once. Presumably, there is only one tuple for George Lucas in that relation, and if so, it is only this tuple that satisfies the `WHERE` clause of line (3). Thus, George Lucas is printed only once. □

4.2 Duplicates in Unions, Intersections, and Differences

Unlike the `SELECT` statement, which preserves duplicates as a default and only eliminates them when instructed to by the `DISTINCT` keyword, the union, intersection, and difference operations, which we introduced in Section 2.5, normally eliminate duplicates. That is, bags are converted to sets, and the set version of the operation is applied. In order to prevent the elimination of duplicates, we must follow the operator `UNION`, `INTERSECT`, or `EXCEPT` by the keyword `ALL`. If we do, then we get the bag semantics of these operators.

Example 28: Consider again the union expression from Example 18, but now add the keyword `ALL`, as:

The Cost of Duplicate Elimination

One might be tempted to place `DISTINCT` after every `SELECT`, on the theory that it is harmless. In fact, it is very expensive to eliminate duplicates from a relation. The relation must be sorted or partitioned so that identical tuples appear next to each other. Only by grouping the tuples in this way can we determine whether or not a given tuple should be eliminated. The time it takes to sort the relation so that duplicates may be eliminated is often greater than the time it takes to execute the query itself. Thus, duplicate elimination should be used judiciously if we want our queries to run fast.

```
(SELECT title, year FROM Movies)
    UNION ALL
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

Now, a title and year will appear as many times in the result as it appears in each of the relations `Movies` and `StarsIn` put together. For instance, if a movie appeared once in the `Movies` relation and there were three stars for that movie listed in `StarsIn` (so the movie appeared in three different tuples of `StarsIn`), then that movie's title and year would appear four times in the result of the union. □

As for union, the operators `INTERSECT ALL` and `EXCEPT ALL` are intersection and difference of bags. Thus, if R and S are relations, then the result of expression

$$R \texttt{ INTERSECT ALL } S$$

is the relation in which the number of times a tuple t appears is the minimum of the number of times it appears in R and the number of times it appears in S.

The result of expression

$$R \texttt{ EXCEPT ALL } S$$

has tuple t as many times as the difference of the number of times it appears in R minus the number of times it appears in S, provided the difference is positive.

4.3 Grouping and Aggregation in SQL

Consider the grouping-and-aggregation operator γ for our extended relational algebra. Recall that this operator allows us to partition the tuples of a relation

into "groups," based on the values of tuples in one or more attributes. We are then able to aggregate certain other columns of the relation by applying "aggregation" operators to those columns. If there are groups, then the aggregation is done separately for each group. SQL provides all the capability of the γ operator through the use of aggregation operators in `SELECT` clauses and a special `GROUP BY` clause.

4.4 Aggregation Operators

SQL uses the five aggregation operators `SUM`, `AVG`, `MIN`, `MAX`, and `COUNT`. These operators are used by applying them to a scalar-valued expression, typically a column name, in a `SELECT` clause. One exception is the expression `COUNT(*)`, which counts all the tuples in the relation that is constructed from the `FROM` clause and `WHERE` clause of the query.

In addition, we have the option of eliminating duplicates from the column before applying the aggregation operator by using the keyword `DISTINCT`. That is, an expression such as `COUNT(DISTINCT x)` counts the number of distinct values in column x. We could use any of the other operators in place of `COUNT` here, but expressions such as `SUM(DISTINCT x)` rarely make sense, since it asks us to sum the different values in column x.

Example 29: The following query finds the average net worth of all movie executives:

```
SELECT AVG(netWorth)
FROM MovieExec;
```

Note that there is no `WHERE` clause at all, so the keyword `WHERE` is properly omitted. This query examines the `netWorth` column of the relation

```
MovieExec(name, address, cert#, netWorth)
```

sums the values found there, one value for each tuple (even if the tuple is a duplicate of some other tuple), and divides the sum by the number of tuples. If there are no duplicate tuples, then this query gives the average net worth as we expect. If there were duplicate tuples, then a movie executive whose tuple appeared n times would have his or her net worth counted n times in the average. □

Example 30: The following query:

```
SELECT COUNT(*)
FROM StarsIn;
```

counts the number of tuples in the `StarsIn` relation. The similar query:

```
SELECT COUNT(starName)
FROM StarsIn;
```

counts the number of values in the `starName` column of the relation. Since duplicate values are not eliminated when we project onto the `starName` column in SQL, this count should be the same as the count produced by the query with `COUNT(*)`.

If we want to be certain that we do not count duplicate values more than once, we can use the keyword `DISTINCT` before the aggregated attribute, as:

```
SELECT COUNT(DISTINCT starName)
FROM StarsIn;
```

Now, each star is counted once, no matter in how many movies they appeared. □

4.5 Grouping

To group tuples, we use a `GROUP BY` clause, following the `WHERE` clause. The keywords `GROUP BY` are followed by a list of *grouping* attributes. In the simplest situation, there is only one relation reference in the `FROM` clause, and this relation has its tuples grouped according to their values in the grouping attributes. Whatever aggregation operators are used in the `SELECT` clause are applied only within groups.

Example 31: The problem of finding, from the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

the sum of the lengths of all movies for each studio is expressed by

```
SELECT studioName, SUM(length)
FROM Movies
GROUP BY studioName;
```

We may imagine that the tuples of relation `Movies` are reorganized and grouped so that all the tuples for Disney studios are together, all those for MGM are together, and so on. The sums of the length components of all the tuples in each group are calculated, and for each group, the studio name is printed along with that sum. □

Observe in Example 31 how the `SELECT` clause has two kinds of terms. These are the only terms that may appear when there is an aggregation in the `SELECT` clause.

1. Aggregations, where an aggregate operator is applied to an attribute or expression involving attributes. As mentioned, these terms are evaluated on a per-group basis.

2. Attributes, such as `studioName` in this example, that appear in the `GROUP BY` clause. In a `SELECT` clause that has aggregations, only those attributes that are mentioned in the `GROUP BY` clause may appear unaggregated in the `SELECT` clause.

While queries involving `GROUP BY` generally have both grouping attributes and aggregations in the `SELECT` clause, it is technically not necessary to have both. For example, we could write

```
SELECT studioName
FROM Movies
GROUP BY studioName;
```

This query would group the tuples of `Movies` according to their studio name and then print the studio name for each group, no matter how many tuples there are with a given studio name. Thus, the above query has the same effect as

```
SELECT DISTINCT studioName
FROM Movies;
```

It is also possible to use a `GROUP BY` clause in a query about several relations. Such a query is interpreted by the following sequence of steps:

1. Evaluate the relation R expressed by the `FROM` and `WHERE` clauses. That is, relation R is the Cartesian product of the relations mentioned in the `FROM` clause, to which the selection of the `WHERE` clause is applied.
2. Group the tuples of R according to the attributes in the `GROUP BY` clause.
3. Produce as a result the attributes and aggregations of the `SELECT` clause, as if the query were about a stored relation R.

Example 32: Suppose we wish to print a table listing each producer's total length of film produced. We need to get information from the two relations

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

so we begin by taking their theta-join, equating the certificate numbers from the two relations. That step gives us a relation in which each `MovieExec` tuple is paired with the `Movies` tuples for all the movies of that producer. Note that an executive who is not a producer will not be paired with any movies, and therefore will not appear in the relation. Now, we can group the selected tuples of this relation according to the name of the producer. Finally, we sum the lengths of the movies in each group. The query is shown in Fig. 13. □

```
SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name;
```

Figure 13: Computing the length of movies for each producer

4.6 Grouping, Aggregation, and Nulls

When tuples have nulls, there are a few rules we must remember:

- The value NULL is ignored in any aggregation. It does not contribute to a sum, average, or count of an attribute, nor can it be the minimum or maximum in its column. For example, COUNT(*) is always a count of the number of tuples in a relation, but COUNT(A) is the number of tuples with non-NULL values for attribute A.
- On the other hand, NULL is treated as an ordinary value when forming groups. That is, we can have a group in which one or more of the grouping attributes are assigned the value NULL.
- When we perform any aggregation except count over an empty bag of values, the result is NULL. The count of an empty bag is 0.

Example 33: Suppose we have a relation $R(A, B)$ with one tuple, both of whose components are NULL:

A	B
NULL	NULL

Then the result of:

```
SELECT A, COUNT(B)
FROM R
GROUP BY A;
```

is the one tuple (NULL, 0). The reason is that when we group by A, we find only a group for value NULL. This group has one tuple, and its B-value is NULL. We thus count the bag of values {NULL}. Since the count of a bag of values does not count the NULL's, this count is 0.

On the other hand, the result of:

```
SELECT A, SUM(B)
FROM R
GROUP BY A;
```

Order of Clauses in SQL Queries

We have now met all six clauses that can appear in a SQL "select-from-where" query: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`. Only the `SELECT` and `FROM` clauses are required. Whichever additional clauses appear must be in the order listed above.

is the one tuple (`NULL`, `NULL`). The reason is as follows. The group for value `NULL` has one tuple, the only tuple in R. However, when we try to sum the B-values for this group, we only find `NULL`, and `NULL` does not contribute to a sum. Thus, we are summing an empty bag of values, and this sum is defined to be `NULL`. □

4.7 `HAVING` Clauses

Suppose that we did not wish to include all of the producers in our table of Example 32. We could restrict the tuples prior to grouping in a way that would make undesired groups empty. For instance, if we only wanted the total length of movies for producers with a net worth of more than $10,000,000, we could change the third line of Fig. 13 to

```
WHERE producerC# = cert# AND networth > 10000000
```

However, sometimes we want to choose our groups based on some aggregate property of the group itself. Then we follow the `GROUP BY` clause with a `HAVING` clause. The latter clause consists of the keyword `HAVING` followed by a condition about the group.

Example 34: Suppose we want to print the total film length for only those producers who made at least one film prior to 1930. We may append to Fig. 13 the clause

```
HAVING MIN(year) < 1930
```

The resulting query, shown in Fig. 14, would remove from the grouped relation all those groups in which every tuple had a `year` component 1930 or higher. □

There are several rules we must remember about `HAVING` clauses:

- An aggregation in a `HAVING` clause applies only to the tuples of the group being tested.
- Any attribute of relations in the `FROM` clause may be aggregated in the `HAVING` clause, but only those attributes that are in the `GROUP BY` list may appear unaggregated in the `HAVING` clause (the same rule as for the `SELECT` clause).

```
SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;
```

Figure 14: Computing the total length of film for early producers

4.8 Exercises for Section 4

! Exercise 4.1 : In Example 27, we mentioned that different versions of the query "find the producers of Harrison Ford's movies" can have different answers as bags, even though they yield the same set of answers. Consider the version of the query in Example 22, where we used a subquery in the `FROM` clause. Does this version produce duplicates, and if so, why?

! Exercise 4.2: The γ operator of extended relational algebra does not have a feature that corresponds to the `HAVING` clause of SQL. Is it possible to mimic a SQL query with a `HAVING` clause in relational algebra? If so, how would we do it in general?

5 Database Modifications

To this point, we have focused on the normal SQL query form: the select-from-where statement. There are a number of other statement forms that do not return a result, but rather change the state of the database. In this section, we shall focus on three types of statements that allow us to

1. Insert tuples into a relation.
2. Delete certain tuples from a relation.
3. Update values of certain components of certain existing tuples.

We refer to these three types of operations collectively as *modifications*.

5.1 Insertion

The basic form of insertion statement is:

`INSERT INTO` $R(A_1, \ldots, A_n)$ `VALUES` $(v_1, \ldots, v_n)$`;`

A tuple is created using the value v_i for attribute A_i, for $i = 1, 2, \ldots, n$. If the list of attributes does not include all attributes of the relation R, then the tuple created has default values for all missing attributes.

Example 35: Suppose we wish to add Sydney Greenstreet to the list of stars of *The Maltese Falcon*. We say:

```
1) INSERT INTO StarsIn(movieTitle, movieYear, starName)
2) VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

The effect of executing this statement is that a tuple with the three components on line (2) is inserted into the relation `StarsIn`. Since all attributes of `StarsIn` are mentioned on line (1), there is no need to add default components. The values on line (2) are matched with the attributes on line (1) in the order given, so `'The Maltese Falcon'` becomes the value of the component for attribute `movieTitle`, and so on. □

If, as in Example 35, we provide values for all attributes of the relation, then we may omit the list of attributes that follows the relation name. That is, we could just say:

```
INSERT INTO StarsIn
VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

However, if we take this option, we must be sure that the order of the values is the same as the standard order of attributes for the relation.

- If you are not sure of the declared order for the attributes, it is best to list them in the `INSERT` clause in the order you choose for their values in the `VALUES` clause.

The simple `INSERT` described above only puts one tuple into a relation. Instead of using explicit values for one tuple, we can compute a set of tuples to be inserted, using a subquery. This subquery replaces the keyword `VALUES` and the tuple expression in the `INSERT` statement form described above.

Example 36: Suppose we want to add to the relation

```
Studio(name, address, presC#)
```

all movie studios that are mentioned in the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

but do not appear in `Studio`. Since there is no way to determine an address or a president for such a studio, we shall have to be content with value `NULL` for attributes `address` and `presC#` in the inserted `Studio` tuples. A way to make this insertion is shown in Fig. 15.

```
1)  INSERT INTO Studio(name)
2)      SELECT DISTINCT studioName
3)      FROM Movies
4)      WHERE studioName NOT IN
5)          (SELECT name
6)           FROM Studio);
```

Figure 15: Adding new studios

Like most SQL statements with nesting, Fig. 15 is easiest to examine from the inside out. Lines (5) and (6) generate all the studio names in the relation `Studio`. Thus, line (4) tests that a studio name from the `Movies` relation is none of these studios.

Now, we see that lines (2) through (6) produce the set of studio names found in `Movies` but not in `Studio`. The use of `DISTINCT` on line (2) assures that each studio will appear only once in this set, no matter how many movies it owns. Finally, line (1) inserts each of these studios, with `NULL` for the attributes `address` and `presC#`, into relation `Studio`. □

5.2 Deletion

The form of a deletion is

`DELETE FROM` R `WHERE` <condition>`;`

The Timing of Insertions

The SQL standard requires that the query be evaluated completely before any tuples are inserted. For example, in Fig. 15, the query of lines (2) through (6) must be evaluated prior to executing the insertion of line (1). Thus, there is no possibility that new tuples added to `Studio` at line (1) will affect the condition on line (4).

In this particular example, it does not matter whether or not insertions are delayed until the query is completely evaluated. However, suppose `DISTINCT` were removed from line (2) of Fig. 15. If we evaluate the query of lines (2) through (6) before doing any insertion, then a new studio name appearing in several `Movies` tuples would appear several times in the result of this query and therefore would be inserted several times into relation `Studio`. However, if the DBMS inserted new studios into `Studio` as soon as we found them during the evaluation of the query of lines (2) through (6), something that would be incorrect according to the standard, then the same new studio would not be inserted twice. Rather, as soon as the new studio was inserted once, its name would no longer satisfy the condition of lines (4) through (6), and it would not appear a second time in the result of the query of lines (2) through (6).

The effect of executing this statement is that every tuple satisfying the condition will be deleted from relation R.

Example 37: We can delete from relation

```
StarsIn(movieTitle, movieYear, starName)
```

the fact that Sydney Greenstreet was a star in *The Maltese Falcon* by the SQL statement:

```
DELETE FROM StarsIn
WHERE movieTitle = 'The Maltese Falcon' AND
      movieYear = 1942 AND
      starName = 'Sydney Greenstreet';
```

Notice that unlike the insertion statement of Example 35, we cannot simply specify a tuple to be deleted. Rather, we must describe the tuple exactly by a `WHERE` clause. □

Example 38: Here is another example of a deletion. This time, we delete from relation

```
MovieExec(name, address, cert#, netWorth)
```

several tuples at once by using a condition that can be satisfied by more than one tuple. The statement

```
DELETE FROM MovieExec
WHERE netWorth < 10000000;
```

deletes all movie executives whose net worth is low — less than ten million dollars. □

5.3 Updates

While we might think of both insertions and deletions of tuples as "updates" to the database, an *update* in SQL is a very specific kind of change to the database: one or more tuples that already exist in the database have some of their components changed. The general form of an update statement is:

`UPDATE` R `SET` <new-value assignments> `WHERE` <condition>`;`

Each new-value assignment is an attribute, an equal sign, and an expression. If there is more than one assignment, they are separated by commas. The effect of this statement is to find all the tuples in R that satisfy the condition. Each of these tuples is then changed by having the expressions in the assignments evaluated and assigned to the components of the tuple for the corresponding attributes of R.

Example 39: Let us modify the relation

```
MovieExec(name, address, cert#, netWorth)
```

by attaching the title `Pres.` in front of the name of every movie executive who is the president of a studio. The condition the desired tuples satisfy is that their certificate numbers appear in the `presC#` component of some tuple in the `Studio` relation. We express this update as:

```
1)  UPDATE MovieExec
2)  SET name = 'Pres. ' || name
3)  WHERE cert# IN (SELECT presC# FROM Studio);
```

Line (3) tests whether the certificate number from the `MovieExec` tuple is one of those that appear as a president's certificate number in `Studio`.

Line (2) performs the update on the selected tuples. Recall that the operator `||` denotes concatenation of strings, so the expression following the `=` sign in line (2) places the characters `Pres.` and a blank in front of the old value of the `name` component of this tuple. The new string becomes the value of the `name` component of this tuple; the effect is that `'Pres. '` has been prepended to the old value of `name`. □

6 Transactions in SQL

To this point, our model of operations on the database has been that of one user querying or modifying the database. Thus, operations on the database are executed one at a time, and the database state left by one operation is the state upon which the next operation acts. Moreover, we imagine that operations are carried out in their entirety ("atomically"). That is, we assumed it is impossible for the hardware or software to fail in the middle of a modification, leaving the database in a state that cannot be explained as the result of the operations performed on it.

Real life is often considerably more complicated. We shall first consider what can happen to leave the database in a state that doesn't reflect the operations performed on it, and then we shall consider the tools SQL gives the user to assure that these problems do not occur.

6.1 Serializability

In applications like Web services, banking, or airline reservations, hundreds of operations per second may be performed on the database. The operations initiate at any of thousands or millions of sites, such as desktop computers or automatic teller machines. It is entirely possible that we could have two operations affecting the same bank account or flight, and for those operations to overlap in time. If so, they might interact in strange ways.

Here is an example of what could go wrong if the DBMS were completely unconstrained as to the order in which it operated upon the database. This example involves a database interacting with people, and it is intended to illustrate why it is important to control the sequences in which interacting events can occur. However, a DBMS would not control events that were so "large" that they involved waiting for a user to make a choice. The event sequences controlled by the DBMS involve only the execution of SQL statements.

Example 40: The typical airline gives customers a Web interface where they can choose a seat for their flight. This interface shows a map of available

seats, and the data for this map is obtained from the airline's database. There might be a relation such as:

```
Flights(fltNo, fltDate, seatNo, seatStatus)
```

upon which we can issue the query:

```
SELECT seatNo
FROM Flights
WHERE fltNo = 123 AND fltDate = DATE '2008-12-25'
    AND seatStatus = 'available';
```

The flight number and date are example data, which would in fact be obtained from previous interactions with the customer.

When the customer clicks on an empty seat, say 22A, that seat is reserved for them. The database is modified by an update-statement, such as:

```
UPDATE Flights
SET seatStatus = 'occupied'
WHERE fltNo = 123 AND fltDate = DATE '2008-12-25'
    AND seatNo = '22A';
```

However, this customer may not be the only one reserving a seat on flight 123 on Dec. 25, 2008 and this exact moment. Another customer may have asked for the seat map at the same time, in which case they also see seat 22A empty. Should they also choose seat 22A, they too believe they have reserved 22A. The timing of these events is as suggested by Fig. 16. □

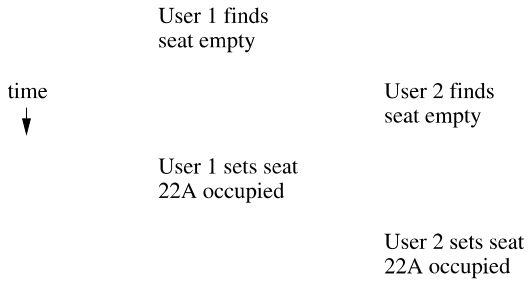

Figure 16: Two customers trying to book the same seat simultaneously

As we see from Example 40, it is conceivable that two operations could each be performed correctly, and yet the global result not be correct: both customers believe they have been granted seat 22A. The problem is solved in SQL by the notion of a "transaction," which is informally a group of operations that need to be performed together. Suppose that in Example 40, the query

Assuring Serializable Behavior

In practice it is often impossible to require that operations run serially; there are just too many of them, and some parallelism is required. Thus, DBMS's adopt a mechanism for assuring serializable behavior; even if the execution is not serial, the result looks to users as if operations were executed serially.

One common approach is for the DBMS to *lock* elements of the database so that two functions cannot access them at the same time. There is an extensive technology of how to implement locks in a DBMS. For example, if the transaction of Example 40 were written to lock other transactions out of the `Flights` relation, then transactions that did not access `Flights` could run in parallel with the seat-selection transaction, but no other invocation of the seat-selection operation could run in parallel.

and update shown would be grouped into one transaction.[6] SQL then allows the programmer to state that a certain transaction must be *serializable* with respect to other transactions. That is, these transactions must behave as if they were run *serially* — one at a time, with no overlap.

Clearly, if the two invocations of the seat-selection operation are run serially (or serializably), then the error we saw cannot occur. One customer's invocation occurs first. This customer sees seat 22A is empty, and books it. The other customer's invocation then begins and is not given 22A as a choice, because it is already occupied. It may matter to the customers who gets the seat, but to the database all that is important is that a seat is assigned only once.

6.2 Atomicity

In addition to nonserialized behavior that can occur if two or more database operations are performed about the same time, it is possible for a single operation to put the database in an unacceptable state if there is a hardware or software "crash" while the operation is executing. Here is another example suggesting what might occur. As in Example 40, we should remember that real database systems do not allow this sort of error to occur in properly designed application programs.

Example 41: Let us picture another common sort of database: a bank's account records. We can represent the situation by a relation

[6] However, it would be extremely unwise to group into a single transaction operations that involved a user, or even a computer that was not owned by the airline, such as a travel agent's computer. Another mechanism must be used to deal with event sequences that include operations outside the database.

```
Accounts(acctNo, balance)
```

Consider the operation of transferring $100 from the account numbered 123 to the account 456. We might first check whether there is at least $100 in account 123, and if so, we execute the following two steps:

1. Add $100 to account 456 by the SQL update statement:

```
UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;
```

2. Subtract $100 from account 123 by the SQL update statement:

```
UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;
```

Now, consider what happens if there is a failure after Step (1) but before Step (2). Perhaps the computer fails, or the network connecting the database to the processor that is actually performing the transfer fails. Then the database is left in a state where money has been transferred into the second account, but the money has not been taken out of the first account. The bank has in effect given away the amount of money that was to be transferred. □

The problem illustrated by Example 41 is that certain combinations of database operations, like the two updates of that example, need to be done *atomically*; that is, either they are both done or neither is done. For example, a simple solution is to have all changes to the database done in a local workspace, and only after all work is done do we *commit* the changes to the database, whereupon all changes become part of the database and visible to other operations.

6.3 Transactions

The solution to the problems of serialization and atomicity posed in Sections 6.1 and 6.2 is to group database operations into *transactions*. A transaction is a collection of one or more operations on the database that must be executed atomically; that is, either all operations are performed or none are. In addition, SQL requires that, as a default, transactions are executed in a serializable manner. A DBMS may allow the user to specify a less stringent constraint on the interleaving of operations from two or more transactions. We shall discuss these modifications to the serializability condition in later sections.

When using the *generic SQL interface* (the facility wherein one types queries and other SQL statements), each statement is a transaction by itself. However,

How the Database Changes During Transactions

Different systems may do different things to implement transactions. It is possible that as a transaction executes, it makes changes to the database. If the transaction aborts, then (unless the programmer took precautions) it is possible that these changes were seen by some other transaction. The most common solution is for the database system to lock the changed items until `COMMIT` or `ROLLBACK` is chosen, thus preventing other transactions from seeing the tentative change. Locks or an equivalent would surely be used if the user wants the transactions to run in a serializable fashion.

However, as we shall see starting in Section 6.4, SQL offers us several options regarding the treatment of tentative database changes. It is possible that the changed data is not locked and becomes visible even though a subsequent rollback makes the change disappear. It is up to the author of a transaction to decide whether it is safe for that transaction to see tentative changes of other transactions.

SQL allows the programmer to group several statements into a single transaction. The SQL command `START TRANSACTION` is used to mark the beginning of a transaction. There are two ways to end a transaction:

1. The SQL statement `COMMIT` causes the transaction to end successfully. Whatever changes to the database were caused by the SQL statement or statements since the current transaction began are installed permanently in the database (i.e., they are *committed*). Before the `COMMIT` statement is executed, changes are tentative and may or may not be visible to other transactions.

2. The SQL statement `ROLLBACK` causes the transaction to *abort*, or terminate unsuccessfully. Any changes made in response to the SQL statements of the transaction are undone (i.e., they are *rolled back*), so they never permanently appear in the database.

Example 42: Suppose we want the transfer operation of Example 41 to be a single transaction. We execute `BEGIN TRANSACTION` before accessing the database. If we find that there are insufficient funds to make the transfer, then we would execute the `ROLLBACK` command. However, if there are sufficient funds, then we execute the two update statements and then execute `COMMIT`. □

6.4 Read-Only Transactions

Examples 40 and 41 each involved a transaction that read and then (possibly) wrote some data into the database. This sort of transaction is prone to

Application- Versus System-Generated Rollbacks

In our discussion of transactions, we have presumed that the decision whether a transaction is committed or rolled back is made as part of the application issuing the transaction. That is, as in Examples 44 and 42, a transaction may perform a number of database operations, then decide whether to make any changes permanent by issuing `COMMIT`, or to return to the original state by issuing `ROLLBACK`. However, the system may also perform transaction rollbacks, to ensure that transactions are executed atomically and conform to their specified isolation level in the presence of other concurrent transactions or system crashes. Typically, if the system aborts a transaction then a special error code or exception is generated. If an application wishes to guarantee that its transactions are executed successfully, it must catch such conditions and reissue the transaction in question.

serialization problems. Thus we saw in Example 40 what could happen if two executions of the function tried to book the same seat at the same time, and we saw in Example 41 what could happen if there was a crash in the middle of a funds transfer. However, when a transaction only reads data and does not write data, we have more freedom to let the transaction execute in parallel with other transactions.

Example 43: Suppose we wrote a program that read data from the `Flights` relation of Example 40 to determine whether a certain seat was available. We could execute many invocations of this program at once, without risk of permanent harm to the database. The worst that could happen is that while we were reading the availability of a certain seat, that seat was being booked or was being released by the execution of some other program. Thus, we might get the answer "available" or "occupied," depending on microscopic differences in the time at which we executed the query, but the answer would make sense at some time. □

If we tell the SQL execution system that our current transaction is *read-only*, that is, it will never change the database, then it is quite possible that the SQL system will be able to take advantage of that knowledge. Generally it will be possible for many read-only transactions accessing the same data to run in parallel, while they would not be allowed to run in parallel with a transaction that wrote the same data.

We tell the SQL system that the next transaction is read-only by:

```
SET TRANSACTION READ ONLY;
```

This statement must be executed before the transaction begins. We can also inform SQL that the coming transaction may write data by the statement

```
SET TRANSACTION READ WRITE;
```

However, this option is the default.

6.5 Dirty Reads

Dirty data is a common term for data written by a transaction that has not yet committed. A *dirty read* is a read of dirty data written by another transaction. The risk in reading dirty data is that the transaction that wrote it may eventually abort. If so, then the dirty data will be removed from the database, and the world is supposed to behave as if that data never existed. If some other transaction has read the dirty data, then that transaction might commit or take some other action that reflects its knowledge of the dirty data.

Sometimes the dirty read matters, and sometimes it doesn't. Other times it matters little enough that it makes sense to risk an occasional dirty read and thus avoid:

1. The time-consuming work by the DBMS that is needed to prevent dirty reads, and
2. The loss of parallelism that results from waiting until there is no possibility of a dirty read.

Here are some examples of what might happen when dirty reads are allowed.

Example 44: Let us reconsider the account transfer of Example 41. However, suppose that transfers are implemented by a program P that executes the following sequence of steps:

1. Add money to account 2.
2. Test if account 1 has enough money.
   (a) If there is not enough money, remove the money from account 2 and end.[7]
   (b) If there is enough money, subtract the money from account 1 and end.

If program P is executed serializably, then it doesn't matter that we have put money temporarily into account 2. No one will see that money, and it gets removed if the transfer can't be made.

However, suppose dirty reads are possible. Imagine there are three accounts: $A1$, $A2$, and $A3$, with \$100, \$200, and \$300, respectively. Suppose transaction

[7] You should be aware that the program P is trying to perform functions that would more typically be done by the DBMS. In particular, when P decides, as it has done at this step, that it must not complete the transaction, it would issue a rollback (abort) command to the DBMS and have the DBMS reverse the effects of this execution of P.

T_1 executes program P to transfer \$150 from $A1$ to $A2$. At roughly the same time, transaction T_2 runs program P to transfer \$250 from $A2$ to $A3$. Here is a possible sequence of events:

1. T_2 executes Step (1) and adds \$250 to $A3$, which now has \$550.
2. T_1 executes Step (1) and adds \$150 to $A2$, which now has \$350.
3. T_2 executes the test of Step (2) and finds that $A2$ has enough funds (\$350) to allow the transfer of \$250 from $A2$ to $A3$.
4. T_1 executes the test of Step (2) and finds that $A1$ does not have enough funds (\$100) to allow the transfer of \$150 from $A1$ to $A2$.
5. T_2 executes Step (2b). It subtracts \$250 from $A2$, which now has \$100, and ends.
6. T_1 executes Step (2a). It subtracts \$150 from $A2$, which now has $-$\$50, and ends.

The total amount of money has not changed; there is still \$600 among the three accounts. But because T_2 read dirty data at the third of the six steps above, we have not protected against an account going negative, which supposedly was the purpose of testing the first account to see if it had adequate funds. □

Example 45: Let us imagine a variation on the seat-choosing function of Example 40. In the new approach:

1. We find an available seat and reserve it by setting `seatStatus` to `'occupied'` for that seat. If there is none, end.
2. We ask the customer for approval of the seat. If so, we commit. If not, we release the seat by setting `seatStatus` to `'available'` and repeat Step (1) to get another seat.

If two transactions are executing this algorithm at about the same time, one might reserve a seat S, which later is rejected by the customer. If the second transaction executes Step (1) at a time when seat S is marked occupied, the customer for that transaction is not given the option to take seat S.

As in Example 44, the problem is that a dirty read has occurred. The second transaction saw a tuple (with S marked occupied) that was written by the first transaction and later modified by the first transaction. □

How important is the fact that a read was dirty? In Example 44 it was very important; it caused an account to go negative despite apparent safeguards against that happening. In Example 45, the problem does not look too serious. Indeed, the second traveler might not get their favorite seat, or might even be told that no seats existed. However, in the latter case, running the transaction

again will almost certainly reveal the availability of seat S. It might well make sense to implement this seat-choosing function in a way that allowed dirty reads, in order to speed up the average processing time for booking requests.

SQL allows us to specify that dirty reads are acceptable for a given transaction. We use the `SET TRANSACTION` statement that we discussed in Section 6.4. The appropriate form for a transaction like that described in Example 45 is:

```
1)  SET TRANSACTION READ WRITE
2)      ISOLATION LEVEL READ UNCOMMITTED;
```

The statement above does two things:

1. Line (1) declares that the transaction may write data.

2. Line (2) declares that the transaction may run with the "isolation level" *read-uncommitted*. That is, the transaction is allowed to read dirty data. We shall discuss the four isolation levels in Section 6.6. So far, we have seen two of them: serializable and read-uncommitted.

Note that if the transaction is not read-only (i.e., it may modify the database), and we specify isolation level `READ UNCOMMITTED`, then we must also specify `READ WRITE`. Recall from Section 6.4 that the default assumption is that transactions are read-write. However, SQL makes an exception for the case where dirty reads are allowed. Then, the default assumption is that the transaction is read-only, because read-write transactions with dirty reads entail significant risks, as we saw. If we want a read-write transaction to run with read-uncommitted as the isolation level, then we need to specify `READ WRITE` explicitly, as above.

6.6 Other Isolation Levels

SQL provides a total of four *isolation levels*. Two of them we have already seen: serializable and read-uncommitted (dirty reads allowed). The other two are *read-committed* and *repeatable-read*. They can be specified for a given transaction by

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

or

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

respectively. For each, the default is that transactions are read-write, so we can add `READ ONLY` to either statement, if appropriate. Incidentally, we also have the option of specifying

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Interactions Among Transactions Running at Different Isolation Levels

A subtle point is that the isolation level of a transaction affects only what data *that* transaction may see; it does not affect what any other transaction sees. As a case in point, if a transaction T is running at level serializable, then the execution of T must appear as if all other transactions run either entirely before or entirely after T. However, if some of those transactions are running at another isolation level, then *they* may see the data written by T as T writes it. They may even see dirty data from T if they are running at isolation level read-uncommitted, and T aborts.

However, that is the SQL default and need not be stated explicitly.

The read-committed isolation level, as its name implies, forbids the reading of dirty (uncommitted) data. However, it does allow a transaction running at this isolation level to issue the same query several times and get different answers, as long as the answers reflect data that has been written by transactions that already committed.

Example 46: Let us reconsider the seat-choosing program of Example 45, but suppose we declare it to run with isolation level read-committed. Then when it searches for a seat at Step (1), it will not see seats as booked if some other transaction is reserving them but not committed.[8] However, if the traveler rejects seats, and one execution of the function queries for available seats many times, it may see a different set of available seats each time it queries, as other transactions successfully book seats or cancel seats in parallel with our transaction. □

Now, let us consider isolation level repeatable-read. The term is something of a misnomer, since the same query issued more than once is not quite guaranteed to get the same answer. Under repeatable-read isolation, if a tuple is retrieved the first time, then we can be sure that the identical tuple will be retrieved again if the query is repeated. However, it is also possible that a second or subsequent execution of the same query will retrieve *phantom* tuples. The latter are tuples that result from insertions into the database while our transaction is executing.

Example 47: Let us continue with the seat-choosing problem of Examples 45 and 46. If we execute this function under isolation level repeatable-read, then

[8] What actually happens may seem mysterious, since we have not addressed the algorithms for enforcing the various isolation levels. Possibly, should two transactions both see a seat as available and try to book it, one will be forced by the system to roll back in order to break the deadlock (see the box on "Application- Versus System-Generated Rollbacks" in Section 6.3).

a seat that is available on the first query at Step (1) will remain available at subsequent queries.

However, suppose some new tuples enter the relation `Flights`. For example, the airline may have switched the flight to a larger plane, creating some new tuples that weren't there before. Then under repeatable-read isolation, a subsequent query for available seats may also retrieve the new seats. □

Figure 17 summarizes the differences between the four SQL isolation levels.

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

Figure 17: Properties of SQL isolation levels

6.7 Exercises for Section 6

Exercise 6.1: This and the next exercises involve certain programs that operate on the two relations

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
```

from our running PC exercise. Sketch the following programs, including SQL statements and work done in a conventional language. Do not forget to issue `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` statements at the proper times and to tell the system your transactions are read-only if they are.

a) Given a speed and amount of RAM (as arguments of the function), look up the PC's with that speed and RAM, printing the model number and price of each.

b) Given a model number, delete the tuple for that model from both `PC` and `Product`.

c) Given a model number, decrease the price of that model PC by $100.

d) Given a maker, model number, processor speed, RAM size, hard-disk size, and price, check that there is no product with that model. If there is such a model, print an error message for the user. If no such model existed in the database, enter the information about that model into the `PC` and `Product` tables.

! Exercise 6.2: For each of the programs of Exercise 6.1, discuss the atomicity problems, if any, that could occur should the system crash in the middle of an execution of the program.

! Exercise 6.3: Suppose we execute as a transaction T one of the four programs of Exercise 6.1, while other transactions that are executions of the same or a different one of the four programs may also be executing at about the same time. What behaviors of transaction T may be observed if all the transactions run with isolation level `READ UNCOMMITTED` that would not be possible if they all ran with isolation level `SERIALIZABLE`? Consider separately the case that T is any of the programs (a) through (d) of Exercise 6.1.

!! Exercise 6.4: Suppose we have a transaction T that is a function which runs "forever," and at each hour checks whether there is a PC that has a speed of 3.5 or more and sells for under $1000. If it finds one, it prints the information and terminates. During this time, other transactions that are executions of one of the four programs described in Exercise 6.1 may run. For each of the four isolation levels — serializable, repeatable read, read committed, and read uncommitted — tell what the effect on T of running at this isolation level is.

7 Summary

- ✦ *SQL*: The language SQL is the principal query language for relational database systems. The most recent full standard is called SQL-99 or SQL3. Commercial systems generally vary from this standard.

- ✦ *Select-From-Where Queries*: The most common form of SQL query has the form select-from-where. It allows us to take the product of several relations (the `FROM` clause), apply a condition to the tuples of the result (the `WHERE` clause), and produce desired components (the `SELECT` clause).

- ✦ *Subqueries*: Select-from-where queries can also be used as subqueries within a `WHERE` clause or `FROM` clause of another query. The operators `EXISTS`, `IN`, `ALL`, and `ANY` may be used to express boolean-valued conditions about the relations that are the result of a subquery in a `WHERE` clause.

- ✦ *Set Operations on Relations*: We can take the union, intersection, or difference of relations by connecting the relations, or connecting queries defining the relations, with the keywords `UNION`, `INTERSECT`, and `EXCEPT`, respectively.

- ✦ *Join Expressions*: SQL has operators such as `NATURAL JOIN` that may be applied to relations, either as queries by themselves or to define relations in a `FROM` clause.

- *Null Values*: SQL provides a special value `NULL` that appears in components of tuples for which no concrete value is available. The arithmetic and logic of `NULL` is unusual. Comparison of any value to `NULL`, even another `NULL`, gives the truth value `UNKNOWN`. That truth value, in turn, behaves in boolean-valued expressions as if it were halfway between `TRUE` and `FALSE`.

- *Outerjoins*: SQL provides an `OUTER JOIN` operator that joins relations but also includes in the result dangling tuples from one or both relations; the dangling tuples are padded with `NULL`'s in the resulting relation.

- *The Bag Model of Relations*: SQL actually regards relations as bags of tuples, not sets of tuples. We can force elimination of duplicate tuples with the keyword `DISTINCT`, while keyword `ALL` allows the result to be a bag in certain circumstances where bags are not the default.

- *Aggregations*: The values appearing in one column of a relation can be summarized (aggregated) by using one of the keywords `SUM`, `AVG` (average value), `MIN`, `MAX`, or `COUNT`. Tuples can be partitioned prior to aggregation with the keywords `GROUP BY`. Certain groups can be eliminated with a clause introduced by the keyword `HAVING`.

- *Modification Statements*: SQL allows us to change the tuples in a relation. We may `INSERT` (add new tuples), `DELETE` (remove tuples), or `UPDATE` (change some of the existing tuples), by writing SQL statements using one of these three keywords.

- *Transactions*: SQL allows the programmer to group SQL statements into transactions, which may be committed or rolled back (aborted). Transactions may be rolled back by the application in order to undo changes, or by the system in order to guarantee atomicity and isolation.

- *Isolation Levels*: SQL defines four isolation levels called, from most stringent to least stringent: "serializable" (the transaction must appear to run either completely before or completely after each other transaction), "repeatable-read" (every tuple read in response to a query will reappear if the query is repeated), "read-committed" (only tuples written by transactions that have already committed may be seen by this transaction), and "read-uncommitted" (no constraint on what the transaction may see).

8 References

Many books on SQL programming are available. Some popular ones are [3], [5], and [7]. [6] is an early exposition of the SQL-99 standard.

SQL was first defined in [4]. It was implemented as part of System R [1], one of the first generation of relational database prototypes.

There is a discussion of problems with this standard in the area of transactions and cursors in [2].

1. M. M. Astrahan et al., "System R: a relational approach to data management," *ACM Transactions on Database Systems* **1**:2, pp. 97–137, 1976.

2. H. Berenson, P. A. Bernstein, J. N. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1–10, 1995.

3. J. Celko, *SQL for Smarties*, Morgan-Kaufmann, San Francisco, 2005.

4. D. D. Chamberlin et al., "SEQUEL 2: a unified approach to data definition, manipulation, and control," *IBM Journal of Research and Development* **20**:6, pp. 560–575, 1976.

5. C. J. Date and H. Darwen, *A Guide to the SQL Standard*, Addison-Wesley, Reading, MA, 1997.

6. P. Gulutzan and T. Pelzer, *SQL-99 Complete, Really*, R&D Books, Lawrence, KA, 1999.

7. J. Melton and A. R. Simon, *Understanding the New SQL: A Complete Guide*, Morgan-Kaufmann, San Francisco, 2006.