**Conceptual Database Design - Entity Relationship(ER) Modeling:**

Database Design Techniques

1. ER Modeling (Top down Approach)
2. Normalization (Bottom Up approach)

What is ER Modeling?

A graphical technique for understanding and organizing the data independent of the actual database implementation

We need to be familiar with the following terms to go further.

Entity

Any thing that has an independent existence and about which we collect data. It is also known as entity type. In ER modeling, notation for entity is given below.



Entity instance

Entity instance is a particular member of the entity type. Example for entity instance : A particular employee **Regular Entity**

An entity which has its own key attribute is a regular entity. Example for regular entity : Employee.

Weak entity

An entity which depends on other entity for its existence and doesn't have any key attribute of its own is a weak

entity.

Example for a weak entity : In a parent/child relationship, a parent is considered as a strong entity and the child is a weak entity.

In ER modeling, notation for weak entity is given below.



Attributes

Properties/characteristics which describe entities are called attributes. In ER modeling, notation for attribute is given below.

Domain of Attributes

The set of possible values that an attribute can take is called the domain of the attribute. For example, the attribute day may take any value from the set {Monday, Tuesday ... Friday}. Hence this set can be termed as the domain of the attribute day.

Key attribute

The attribute (or combination of attributes) which is unique for every entity instance is called key attribute.

E.g the employee\_id of an employee, pan\_card\_number of a person etc.If the key attribute consists of two or more attributes in combination, it is called a composite key.

In ER modeling, notation for key attribute is given below.



Simple attribute

If an attribute cannot be divided into simpler components, it is a simple attribute. Example for simple attribute : employee\_id of an employee.

Composite attribute

If an attribute can be split into components, it is called a composite attribute.

Example for composite attribute : Name of the employee which can be split into First\_name, Middle\_name, and Last\_name.

Single valued Attributes

If an attribute can take only a single value for each entity instance, it is a single valued attribute. example for single valued attribute : age of a student. It can take only one value for a particular student. **Multi-valued Attributes**

If an attribute can take more than one value for each entity instance, it is a multi-valued attribute. Multi-valued

example for multi valued attribute : telephone number of an employee, a particular employee may have multiple telephone numbers.

In ER modeling, notation for multi-valued attribute is given below.



Stored Attribute

An attribute which need to be stored permanently is a stored attribute Example for stored attribute : name of a student

Derived Attribute

An attribute which can be calculated or derived based on other attributes is a derived attribute.

Example for derived attribute : age of employee which can be calculated from date of birth and current date. In ER modeling, notation for derived attribute is given below.

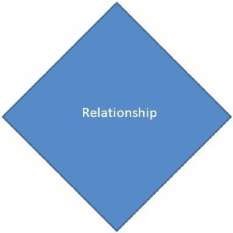


Relationships

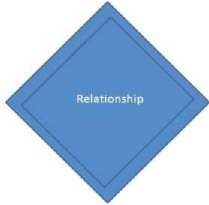
Associations between entities are called relationships

Example : An employee works for an organization. Here "works for" is a relation between the entities employee and organization.

In ER modeling, notation for relationship is given below.



However in ER Modeling, To connect a weak Entity with others, you should use a weak relationship notation as given below



Degree of a Relationship

Degree of a relationship is the number of entity types involved. The n-ary relationship is the general form for degree n. Special cases are unary, binary, and ternary ,where the degree is 1, 2, and 3, respectively.

Example for unary relationship : An employee ia a manager of another employee Example for binary relationship : An employee works-for department.

Example for ternary relationship : customer purchase item from a shop keeper

Cardinality of a Relationship

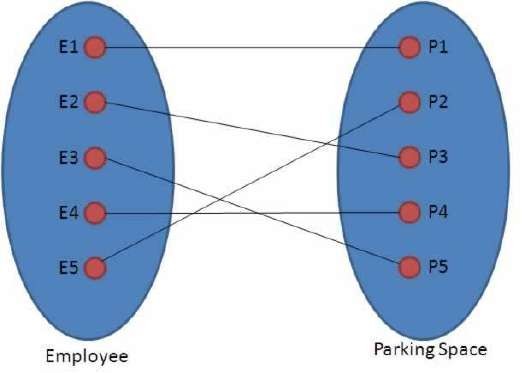
Relationship cardinalities specify how many of each entity type is allowed. Relationships can have four possible connectivities as given below.

1. One to one (1:1) relationship
2. One to many (1:N) relationship
3. Many to one (M:1) relationship
4. Many to many (M:N) relationship

The minimum and maximum values of this connectivity is called the cardinality of the relationship

Example for Cardinality – One-to-One (1:1)

Employee is assigned with a parking space.



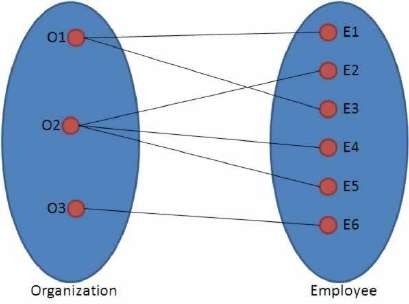
One employee is assigned with only one parking space and one parking space is assigned to only one employee. Hence it is a 1:1 relationship and cardinality is One-To-One (1:1)

In ER modeling, this can be mentioned using notations as given below



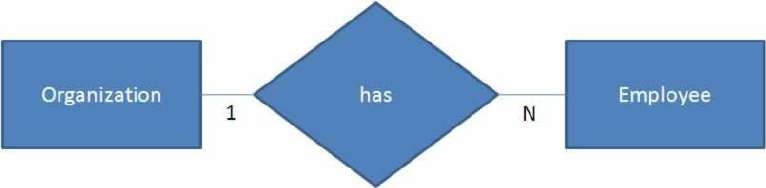
Example for Cardinality – One-to-Many (1:N)

Organization has employees



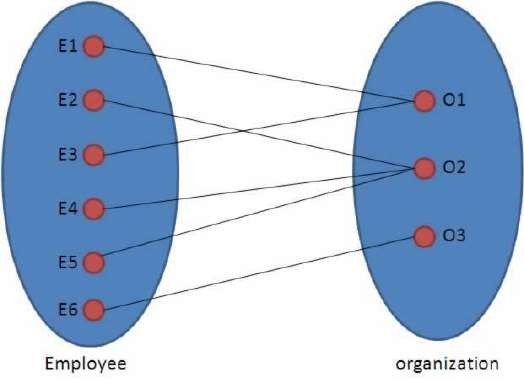
One organization can have many employees , but one employee works in only one organization. Hence it is a 1:N relationship and cardinality is One-To-Many (1:N)

In ER modeling, this can be mentioned using notations as given below



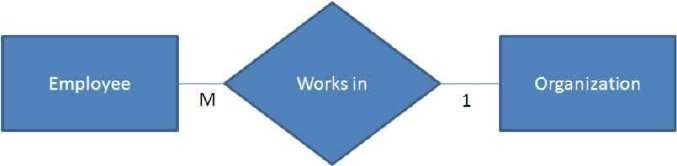
Example for Cardinality – Many-to-One (M :1)

It is the reverse of the One to Many relationship. employee works in organization



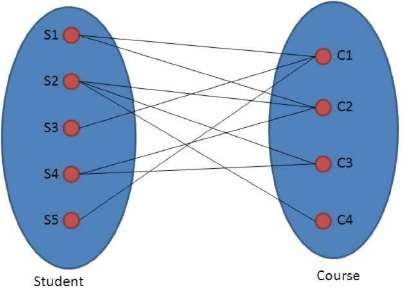
One employee works in only one organization But one organization can have many employees. Hence it is a M:1 relationship and cardinality is Many-to-One (M :1)

In ER modeling, this can be mentioned using notations as given below.



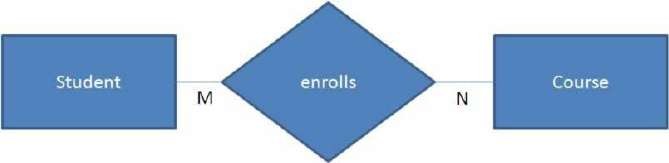
Cardinality – Many-to-Many (M:N)

Students enrolls for courses



One student can enroll for many courses and one course can be enrolled by many students. Hence it is a M:N relationship and cardinality is Many-to-Many (M:N)

In ER modeling, this can be mentioned using notations as given below



Relationship Participation

1. **Total**

In total participation, every entity instance will be connected through the relationship to another instance of the other participating entity types

1. Partial

Example for relationship participation

Consider the relationship - Employee is head of the department.

Here all employees will not be the head of the department. Only one employee will be the head of the department. In other words, only few instances of employee entity participate in the above relationship. So employee entity's participation is partial in the said relationship.

However each department will be headed by some employee. So department entity's participation is total in the said relationship.

**Advantages and Disadvantages of ER Modeling ( Merits and Demerits of ER Modeling ) Advantages**

1. ER Modeling is simple and easily understandable. It is represented in business users language and it can be understood by non-technical specialist.
2. Intuitive and helps in Physical Database creation.
3. Can be generalized and specialized based on needs.
4. Can help in database design.
5. Gives a higher level description of the system.

**Disadvantages**

1. Physical design derived from E-R Model may have some amount of ambiguities or inconsistency.
2. Sometime diagrams may lead to misinterpretations

###### Relational Model

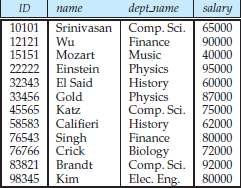
The relational model is today the primary data model for commercial data processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model. In this, we first study the fundamentals of the relational model. A substantial theory exists for relational databases.

Structure of Relational Databases:

A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example, consider the *instructor* table of Figure:1.5, which stores information about instructors. The table has four column headers: *ID*, *name*, *dept name*, and *salary*. Each row of this table records information about an instructor, consisting of the instructor’s *ID*, *name*, *dept name*, and *salary*. Similarly, the *course* table of Figure 1.6 stores information about courses, consisting of a *course id*, *title*, *dept name*, and *credits*, for each course. Note that each instructor is identified by the value of the column *ID*, while each course is identified by the value of the column *course id*.

Figure 1.7 shows a third table, *prereq*, which stores the prerequisite courses for each course. The table has two columns, *course id* and *prereq id*. Each row consists of a pair of course identifiers such that the second course is a prerequisite for the first course.

Thus, a row in the *prereq* table indicates that two courses are *related* in the sense that one course is a prerequisite for the other. As another example, we consider the table *instructor*, a row in the table can be thought of as representing the relationship between a specified *ID* and the corresponding values for *name*,*dept name*, and *salary* values.



**Figure 1.5: The *instructor* relation (2.1)**

In general, a row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In mathematical terminology, a *tuple* is simply a sequence (or list) of values. A relationship between *n* values is represented mathematically by an *n-tuple* of values, i.e., a tuple with *n* values, which corresponds to a row in a table.

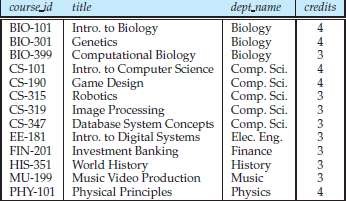
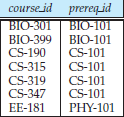


Figure: 1.6: The *course* relation (2.2)



**Figure: 1.7: The *prereq* relation**. **(2.3)**

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.

Examining Figure 1.5, we can see that the relation *instructor* has four attributes:

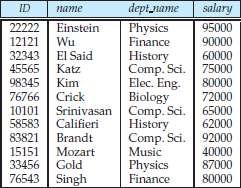
***ID*, *name*, *dept name*, and *salary*.**

We use the term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set of rows. The instance of *instructor* shown in Figure 1.5 has 12 tuples, corresponding to 12 instructors.

In this topic, we shall be using a number of different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a university. They do not include all the data an actual university database would contain, in order to simplify our presentation.

The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 1.5, or are unsorted, as in Figure 1.8, does not matter; the relations in the two figures are the same, since both contain the same set of tuples. For ease of exposition, we will mostly show the relations sorted by their first attribute. For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute. Thus, the domain of the *salary* attribute of the *instructor* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible instructor names.

We require that, for all relations *r*, the domains of all attributes of *r* be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units.



**Figure: 1.8: Unsorted display of the *instructor* relation.** (2-4)

For example, suppose the table *instructor* had an attribute *phone number*, which can store a set of phone numbers corresponding to the instructor. Then the domain of *phone number* would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely the individual phone numbers

in the set.

The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the *phone number* attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code and a local number, we would be treating it as a nonatomic value. If we treat each phone number as a single indivisible unit, then the attribute *phone number* would have an atomic domain.

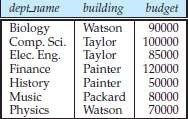
The **null** value is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the null value to signify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus should be eliminated if at all possible. We shall assume null values are absent initially.

Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given

instant in time. The concept of a relation corresponds to the programming-language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

In general, a relation schema consists of a list of attributes and their corresponding domains. The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time;



**Figure 1.9: The *department* relation**.(2-5)

similarly the contents of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change. Although it is important to know the difference between a relation schema and a relation instance, we often use the same name, such as *instructor*, to refer to both the schema and the instance. Where required, we explicitly refer to the schema or to the instance, for example “the *instructor* schema,” or “an instance of the *instructor* relation.” However, where it is clear whether we mean the schema or the instance, we simply use the relation name.

Consider the *department* relation of Figure 1.9. The schema for that relation is

***department* (*dept name*, *building*, *budget*)**

Note that the attribute *dept name* appears in both the *instructor* schema and the *department* schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations.

For example, suppose we wish to find the information about all the instructors who work in the Watson building. We look first at the *department* relation to find the *dept name* of all the departments housed in Watson. Then, for each such department, we look in the *instructor* relation to find the information about the instructor associated with the corresponding *dept name*.

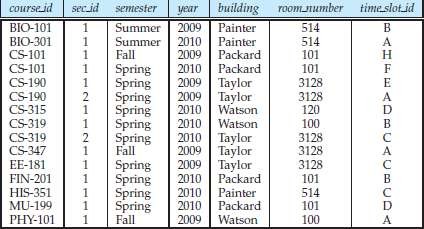
Let us continue with our university database example. Each course in a university may be offered multiple times, across different semesters, or even within a semester.We need a relation to describe each individual

offering, or section, of the class. The schema is

***section* (*course id*, *sec id*, *semester*, *year*, *building*, *room number*, *time slot id*)**

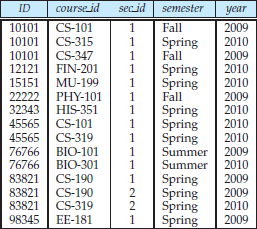
Figure 1.10 shows a sample instance of the *section* relation. We need a relation to describe the association between instructors and the class sections that they teach. The relation schema to describe this association is

***teaches* (*ID*, *course id*, *sec id*, *semester*, *year*)**



**Figure 1.10: The *section* relation.(2-6)**

Figure 1.11 shows a sample instance of the *teaches* relation. As you can imagine, there are many more relations maintained in a real university database. In addition to those relations we have listed already, *instructor*, *department*, *course*, *section*, *prereq*, and *teaches*,we use the following relations in this text:



**Figure: 1.11: The *teaches* relation.(2-7)**

Keys

* *student* (*ID*, *name*, *dept name*, *tot cred*)
* *advisor* (*s id*, *i id*)
* *takes* (*ID*, *course id*, *sec id*, *semester*, *year*, *grade*)
* *classroom* (*building*, *room number*, *capacity*)
* *time slot* (*time slot id*, *day*, *start time*, *end time*)

We must have a way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can *uniquely identify* the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from

another. Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name. Formally, let *R* denote the set of attributes in the schema of relation *r*. If we say that a subset *K* of *R* is a *superkey* for *r* , we are restricting consideration to instances of relations *r* in which no two distinct tuples have the same values on all attributes in *K*. That is, if *t*1 and *t*2 are in *r* and *t*1 = *t*2, then *t*1*.K* = *t*2*.K*.

A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *instructor*. If *K* is a superkey, then so is any superset of *K*. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept name* is sufficient to distinguish among members of the *instructor* relation. Then, both *{ID}* and

*{name*, *dept name}* are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, *{ID*, *name}*, does not form a candidate key, since the attribute *ID* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time. The designation of a key represents a constraint in the real-world enterprise being modeled.

Primary keys must be chosen with care. As we noted, the name of a person is obviously not sufficient, because there may be many people with the same name. In the United States, the social-security number attribute of a person would be a candidate key. Since non-U.S. residents usually do not have social-security numbers, international enterprises must generate their own unique identifiers.

An alternative is to use some unique combination of other attributes as a key. The primary key should be chosen such that its attribute values are never, or very rarely, changed. For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined. A relation, say *r*1, may include among its attributes the primary key of another relation, say *r*2. This attribute is called a **foreign key** from *r*1, referencing *r*2.

The relation *r*1 is also called the **referencing relation** of the foreign key dependency, and *r*2 is called the **referenced relation** of the foreign key. For example, the attribute *dept name* in *instructor* is a foreign key from *instructor*, referencing *department*, since *dept name* is the primary key of *department*. In any database instance, given any tuple, say *ta*, from the *instructor* relation, there must be some tuple, say *tb*, in the *department* relation such that the value of the *dept name* attribute of *ta* is the same as the value of the primary key, *dept name*, of *tb*.

Now consider the *section* and *teaches* relations. It would be reasonable to require that if a section exists for a course, it must be taught by at least one instructor; however, it could possibly be taught by more than one instructor. To enforce this constraint, we would require that if a particular (*course id*, *sec id*, *semester*, *year*) combination appears in *section*, then the same combination must appear in *teaches*. However, this set of values does not form a primary key for *teaches*, since more than one instructor may teach one such section. As a result, we cannot declare a foreign key constraint from *section* to *teaches* (although we can define a foreign key constraint in the other direction, from *teaches* to *section*).

The constraint from *section* to *teaches* is an example of a **referential integrity constraint**; a referential integrity constraint requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

**Schema Diagrams**

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**. Figure 1.12 shows the schema diagram for our university organization. Each relation appears as a box, with the relation name at the top in blue, and the attributes listed inside the box. Primary key attributes are shown underlined. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

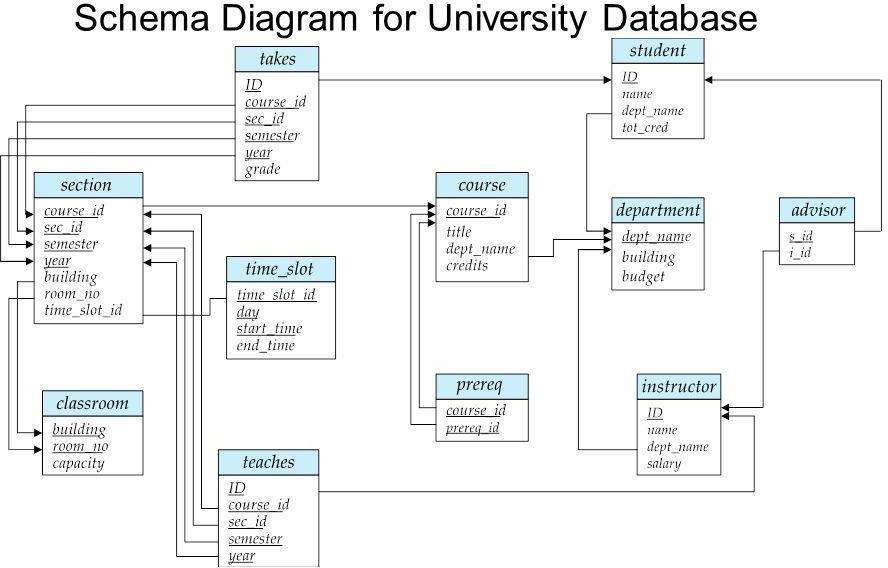


Figure 1.12 : Schema diagram for the university database.

Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams. We will study a different diagrammatic representation called the entity-relationship diagram.

Relational Algebra

PRELIMINARIES

In defining relational algebra and calculus, the alternative of referring to fields by position is more convenient than referring to fields by name: Queries often involve the computation of intermediate results, which are themselves relation instances, and if we use field names to refer to fields, the definition of query language constructs must specify the names of fields for all intermediate relation instances. This can be tedious and is really a secondary issue because we can refer to fields by position anyway. On the other hand, field names make queries more readable.

Due to these considerations, we use the positional notation to formally define relational algebra and calculus. We also introduce simple conventions that allow intermediate relations to ‘inherit’ field names, for convenience.

We present a number of sample queries using the following schema:

Sailors (*sid:* integer, *sname:* string, *rating:* integer, *age:* real) Boats (*bid:* integer, *bname:* string, *color:* string)

Reserves (*sid:* integer, *bid:* integer, *day:* date)

The key fields are underlined, and the domain of each field is listed after the field name. Thus *sid* is the key for Sailors, *bid* is the key for Boats, and all three fields together form the key for Reserves. Fields in an instance of one of these relations will be referred to by name, or positionally, using the order in which they are listed above.

In several examples illustrating the relational algebra operators, we will use the in- stances *S*1 and *S*2 (of Sailors) and *R*1 (of Reserves) shown in Figures 4.1, 4.2, and 4.3, respectively,

RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the re- lational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) rela- tion instances as arguments and returns a relation instance as the result. This property makes it easy to compose operators to form a complex query—a relational algebra expression is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions. We describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention, in the following sections.Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query. The procedural nature of the algebra allows us to think of an algebra expression as a recipe, or a plan, for evaluating a query, and relational systems in fact use algebra expressions to represent query evaluation plans.

**Selection and Projection**

Relational algebra includes operators to *select* rows from a relation (*σ*) and to *project* columns (*π*). These operations allow us to manipulate data in a single relation. Con- sider the instance of the Sailors relation shown in Figure 4.2, denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the *σ* operator. The expression,

*σrating>*8(*S*2)

evaluates to the relation shown in Figure 4.4. The subscript *rating>8* specifies the selection criterion to be applied while retrieving tuples.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | ***sname*** | ***rating*** |
| *yuppy* | *9* |
| *Lubber* | *8* |
| *Guppy* | *5* |
| *Rusty* | *10* |
| ***sid*** | ***sname*** | ***rating*** | ***age*** |  | |
| *28* | *Yuppy* | *9* | *35.0* |
| *58* | *Rusty* | *10* | *35.0* |

Figure 4.4 *σrating>*8(*S*2) Figure 4.5*πsname,rating*(*S*2)

The selection operator *σ* specifies the tuples to retain through a *selection condition*. In general, the selection condition is a boolean combination (i.e., an expression using the logical connectives ∧ and ∨) of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the comparison operators *<, <*=*,* =*,* =*, >*=, or *>*. The reference to an attribute can be by position (of the form *.i* or *i*) or by name (of the form *.name* or *name*). The schema of the result of a selection is the schema of the input relation instance

The projection operator *π* allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using *π*. The expression *πsname,rating*(*S*2)

Suppose that we wanted to find out only the ages of sailors. The expression

*πage*(*S*2)

a single tuple with *age*=*35.0* appears in the result of the projection. This follows from the definition of a relation as a *set* of tuples. In practice, real systems often omit the expensive step of eliminating *duplicate tuples*, leading to relations that are multisets. However, our discussion of relational algebra and calculus assumes that duplicate elimination is always done so that relations are always sets of tuples.

We can compute the names and ratings of highly rated sailors by combining two of the preceding queries. The expression

*πsname,rating*(*σrating>*8(*S*2))

***age***

|  |  |
| --- | --- |
| ***sname*** | ***rating*** |
| yuppy | 9 |
| Rusty | 10 |

35.0

55.5

Figure 4.6 *πage*(*S*2) Figure 4.7 *πsname,rating*(*σrating>*8(*S*2))

Set Operations

The following standard operations on sets are also available in relational algebra: *union* (*U*),

*intersection* (*∩*), *set-difference* (*−*), and *cross-product* (*×*).

* Union: *R u S* returns a relation instance containing all tuples that occur in *either* relation instance *R* or relation instance *S* (or both). *R* and *S* must be *union- compatible*, and the schema of the result is defined to be identical to the schema of *R*.
* Intersection: *R ∩ S* returns a relation instance containing all tuples that occur in *both R* and *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.
* Set-difference: *R − S* returns a relation instance containing all tuples that occur in *R* but not in *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.
* Cross-product: *R × S* returns a relation instance whose schema contains all the fields of *R* (in the same order as they appear in *R*) followed by all the fields of *S* (in the same order as they appear in *S*). The result of *R × S* contains one tuple

〈 *r, s* 〉 (the concatenation of tuples *r* and *s*) for each pair of tuples *r* ∈ *R, s* ∈ *S*.

The cross-product opertion is sometimes called Cartesian product.

We now illustrate these definitions through several examples. The union of *S*1 and *S*2 is shown in Figure 4.8. Fields are listed in order; field names are also inherited from *S*1. *S*2 has the same field names, of course, since it is also an instance of Sailors.In general, fields of *S*2 may have different names; recall that we require only domains to match. Note that the result is a *set* of tuples. Tuples that appear in both *S*1 and *S*2 appear only once in *S*1 ∪ *S*2. Also, *S*1 ∪ *R*1 is not a valid operation because the two relations are not union-compatible. The intersection of *S*1 and *S*2 is shown in Figure 4.9, and the set-difference *S*1 *− S*2 is shown in Figure 4.10.

|  |  |  |  |
| --- | --- | --- | --- |
| ***sid*** | ***sname*** | ***rating*** | ***age*** |
| *22* | *Dustin* | *7* | *45.0* |
| *31* | *Lubber* | *8* | *55.5* |
| *58* | *Rusty* | *10* | *35.0* |
| *28* | *Yuppy* | *9* | *35.0* |
| *44* | *Guppy* | *5* | *35.0* |

Figure 4.8 *S*1 ∪ *S*2

|  |  |  |  |
| --- | --- | --- | --- |
| *sid* | *sname* | *rating* | *age* |
| *31* | *Lubbe* | *8* | *55.5* |
| *58* | *Rusty* | *10* | *35.0* |

|  |  |  |  |
| --- | --- | --- | --- |
| *sid* | *sname* | *rating* | *age* |
| 22 | Dustin | 7 | 45.0 |

Figure 4.9 *S*1 *∩ S*2 Figure 4.10 *S*1 *− S*2

The result of the cross-product *S*1 *× R*1 is shown in Figure 4.11 The fields in *S*1

*× R*1 have the same domains as the corresponding fields in *R*1 and *S*1. In Figure 4.11 *sid* is listed in parentheses to

emphasize that it is not an inherited field name; only the corresponding domain is inherited.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| (*sid*) | *sname* | *rating* | *age* | (*sid*) | *bid* | *day* |
| 22 | *Dustin* | *7* | *45.0* | 22 | *101* | *10/10/96* |
| 22 | *Dustin* | *7* | *45.0* | 58 | *103* | *11/12/96* |
| 31 | *Lubber* | *8* | *55.5* | 22 | *101* | *10/10/96* |
| 31 | *Lubber* | *8* | *55.5* | 58 | *103* | *11/12/96* |
| 58 | *Rusty* | *10* | *35.0* | 22 | *101* | *10/10/96* |
| 58 | *Rusty* | *10* | *35.0* | 58 | *103* | *11/12/96* |

**Renaming**

Figure 4.11 *S*1 *× R*1

We introduce a renaming operator *ρ* for this purpose. The expression *ρ*(*R*(*F* )*, E*) takes an arbitrary relational algebra expression *E* and returns an instance of a (new) relation called *R*. *R* contains the same tuples as the result of *E*, and has the same schema as *E*, but some fields are renamed. The field names in relation *R* are the same as in *E*, except for fields renamed in the *renaming list F.*

For example, the expression *ρ*(*C*(1 *→ sid*1*,* 5 *→ sid*2)*, S*1 *× R*1) returns a relation that contains the tuples shown in Figure 4.11 and has the following schema: C(*sid1:* integer, *sname:* string, *rating:* integer, *age:* real, *sid2:* integer, *bid:* integer,*day:* dates).

It is customary to include some additional operators in the algebra, but they can all be defined in terms of the operators that we have defined thus far. (In fact, the renaming operator is only needed for syntactic convenience, and even the *∩* operator is redundant; *R*

*∩ S* can be defined as *R −* (*R − S*).) We will consider these additional operators,and their definition in terms of the basic operators, in the next two subsections.

**Joins**

The *join* operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products.

joins have received a lot of attention, and there are several variants of the join operation.

Condition Joins

The most general version of the join operation accepts a *join condition c* and a pair of relation instances as arguments, and returns a relation instance. The *join condition* is identical to a *selection condition* in form. The operation is defined as follows:

*R* 𝝰⊳*c S* = *σc*(*R × S*)

Thus 𝝰⊳ is defined to be a cross-product followed by a selection. Note that the condition

*c* can (and typically *does*) refer to attributes of both *R* and *S*. The reference to an

attribute of a relation, say *R*, can be by position (of the form *R.i*) or by name (of the form *R.name*).As an example, the result of *S*1 𝝰⊳*S*1*.sid<R*1*.sid R*1 is shown in Figure 4.12. Because *sid* appears in both *S*1 and *R*1, the corresponding fields in the result of the cross- product *S*1 *× R*1 (and therefore in the result of *S*1 𝝰⊳*S*1*.sid<R*1*.sid R*1) are unnamed. Domains are inherited from the corresponding fields of *S*1 and *R*1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| (*sid*) | *sname* | *rating* | *age* | (*sid*) | *bid* | *day* |
| 22 | *Dustin* | *7* | *45.0* | 58 | *103* | *11/12/96* |
| 31 | *Lubber* | *8* | *55.5* | 58 | *103* | *11/12/96* |

Figure 4.12 *S*1 𝝰⊳*S*1*.sid<R*1*.sid R*1

Equijoin

A common special case of the join operation *R* 𝝰⊳ *S* is when the *join condition* con- sists solely of equalities (connected by ∧) of the form *R.name*1 = *S.name*2, that is, equalities between two fields in *R* and *S*. In this case, obviously, there is some redun- dancy in retaining both attributes in the result. For join conditions that contain only such equalities, the join operation is refined by doing an additional projection in which *S.name*2 is dropped. The join operation with this refinement is called equijoin.

The schema of the result of an equijoin contains the fields of *R* (with the same names and domains as in *R*) followed by the fields of *S* that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from *R* and *S*, they are unnamed in the result relation.

We illustrate *S*1 𝝰⊳*R.sid*=*S.sid R*1 in Figure 4.13. Notice that only one field called *sid*

appears in the result.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *sid* | *sname* | *rating* | *age* | *bid* | *day* |
| *22* | *Dustin* | *7* | *45.0* | *101* | *10/10/96* |
| *58* | *Rusty* | *10* | *35.0* | *103* | *11/12/96* |

Natural Join

Figure 4.13 *S*1 𝝰⊳*R.sid*=*S.sid R*1

A further special case of the join operation *R* 𝝰⊳ *S* is an equijoin in which equalities are specified on *all* fields having the same name in *R* and *S*. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. We call this special case a *natural join*, and it has the nice property that the result is guaranteed not to have two fields with the same name.

The equijoin expression *S*1 𝝰⊳*R.sid*=*S.sid R*1 is actually a natural join and can simply be denoted as *S*1 𝝰⊳ *R*1, since the only common field is *sid*. If the two relations have no attributes in common, *S*1 𝝰⊳ *R*1 is simply the cross-product.

Division

The division operator is useful for expressing certain kinds of queries, for example: “Find the names of sailors who have reserved all boats.” Understanding how to use the basic operators of the algebra to define division is a useful exercise. However,

the division operator does not have the same importance as the other operators—it is not needed as often, and database systems do not try to exploit the semantics of division by implementing it as a distinct operator (as, for example, is done with the join operator).

We discuss division through an example. Consider two relation instances *A* and *B* in which *A* has (exactly) two fields *x* and *y* and *B* has just one field *y*, with the same domain as in *A*. We define the *division* operation *A/B* as the set of all *x* values (in the form of unary tuples) such that for *every y* value in (a tuple of) *B*, there is a tuple

〈*x,y*〉in *A*.

Another way to understand division is as follows. For each *x* value in (the first column of) *A*, consider the set of *y* values that appear in (the second field of) tuples of *A* with that *x* value. If this set contains (all *y* values in) *B*, the *x* value is in the result of *A/B*.

An analogy with integer division may also help to understand division. For integers *A* and *B*, *A/B* is the largest integer *Q* such that *Q* ∗ *B ≤ A*. For relation instances *A* and *B*, *A/B* is the largest relation instance *Q* such that *Q × B* ⊆ *A*.

Division is illustrated in Figure 4.14. It helps to think of *A* as a relation listing the parts supplied by suppliers, and of the *B* relations as listing parts. *A/Bi* computes suppliers who supply *all* parts listed in relation instance *Bi*.

Expressing *A/B* in terms of the basic algebra operators is an interesting exercise, and the reader should try to do this before reading further. The basic idea is to compute all *x* values in *A* that are not *disqualified*. An *x* value is *disqualified* if by attaching a

*y* value from *B*, we obtain a tuple 〈 *x,y* 〉 that is not in *A*. We can compute disqualified tuples using the algebra expression

Thus we can define *A/B* as

*πx*((*πx*(*A*) *× B*) *− A*)

*πx*(*A*) *− πx*((*πx*(*A*) *× B*) *− A*)

To understand the division operation in full generality, we have to consider the case when both *x* and *y* are replaced by a set of attributes.

More Examples of Relational Algebra Queries

We illustrate queries using thei nstances *S*3 of Sailors, *R*2 of Reserves, and *B*1 of Boats, shown in Figures 4.15,4.16,and4.17, respectively.

|  |  |  |  |
| --- | --- | --- | --- |
| ***sid*** | ***sname*** | ***rating*** | ***age*** |
| 22 | Dustin | 7 | 45.0 |
| 29 | Brutus | 1 | 33.0 |
| 31 | Lubber | 8 | 55.5 |
| 32 | Andy | 8 | 25.5 |
| 58 | Rusty | 10 | 35.0 |
| 64 | Horatio | 7 | 35.0 |
| 71 | Zorba | 10 | 16.0 |
| 74 | Horatio | 9 | 35.0 |
| 85 | Art | 3 | 25.5 |
| 95 | Bob | 3 | 63.5 |

|  |  |  |
| --- | --- | --- |
| ***sid*** | ***bid*** | ***day*** |
| 22 | 101 | 10/10/98 |
| 22 | 102 | 10/10/98 |
| 22 | 103 | 10/8/98 |
| 22 | 104 | 10/7/98 |
| 31 | 102 | 11/10/98 |
| 31 | 103 | 11/6/98 |
| 31 | 104 | 11/12/98 |
| 64 | 101 | 9/5/98 |
| 64 | 102 | 9/8/98 |
| 74 | 103 | 9/8/98 |

Figure 4.15An Instance *S*3 of Sailors Figure 4.16An Instance *R*2 of Reserves

|  |  |  |
| --- | --- | --- |
| *bid* | *bname* | *color* |
| *101* | *Interlak* | *blue* |
| *102* | *Interlak* | *red* |
| *103* | *Clipper* | *green* |
| *104* | *Marine* | *red* |

Figure 4.17 An Instance *B*1 of Boats

***(Q1) Find the names of sailors who have reserved boat 103.***

This query can be written as follows:

*πsname*((*σbid*=103*Reserves*) 𝝰⊳*Sailors*)

We first compute the set of tuples in Reserves with *bid* = 103 and then take the natural join of this set with Sailors. This expression can be evaluated on instances of Reserves and Sailors. Evaluated on the instances *R*2 and *S*3, it yields a relation that contains just one field, called *sname*, and three tuples 〈*Dustin*〉, 〈*Horatio*〉,and

*Lubber* 〉 . (Observe that there are two sailors called Horatio, and only one of them has reserved a red boat.)

We can break this query into smaller pieces using the renaming operator *ρ*:

*ρ*(*T emp*1*, σbid*=103*Reserves*) *ρ*(*T emp*2*, T emp*1 𝝰⊳ *Sailors*) *πsname*(*Temp*2)

*T emp*1 denotes an intermediate relation that identifies reservations of boat 103. *T emp*2 is another intermediate relation, and it denotes sailors who have made a reservation in the set *T emp*1. The instances of these relations when evaluating this query on the instances *R*2 and *S*3 are illustrated in Figures 4.18 and 4.19. Finally, we extract the *sname* column from *T emp*2.

|  |  |  |
| --- | --- | --- |
| ***sid*** | ***bid*** | ***day*** |
| 22 | 103 | 10/8/98 |
| 31 | 103 | 11/6/98 |
| 74 | 103 | 9/8/98 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***sid*** | ***sname*** | ***rating*** | ***age*** | ***bid*** | ***day*** |
| 22 | Dustin | 7 | 45.0 | 103 | 10/8/98 |
| 31 | Lubber | 8 | 55.5 | 103 | 11/6/98 |
| 74 | Horatio | 9 | 35.0 | 103 | 9/8/98 |

Figure 4.18Instance of *T emp*1 Figure 4.19 Instance of *T emp*2

*πsname*(*σbid*=103(*Reserves*𝝰⊳*Sailors*))

The DBMS translates an SQL query into (an extended form of) relational algebra, and then looks for other algebra expressions that will produce the same answers but are cheaper to evaluate. If the user’s query is first translated into the expression

*πsname*(*σbid*=103(*Reserves*𝝰⊳*Sailors*)) a good query optimizer will find the equivalent expression

*πsname*((*σbid*=103*Reserves*) 𝝰⊳ *Sailors*)

***(Q2) Find the names of sailors who have reserved a red boat.***

*πsname*((*σcolor*=*′red′ Boats*) 𝝰⊳ *Reserves* 𝝰⊳ *Sailors*

This query involves a series of two joins. First we choose (tuples describing) red boats. Then we join this set with Reserves (natural join, with equality specified on the *bid* column) to identify reservations of red boats. Next we join the resulting intermediate

relation with Sailors (natural join, with equality specified on the *sid* column) to retrieve the names of sailors who have made reservations of red boats. Finally, we project the sailors’ names. The answer, when evaluated on the instances *B*1, *R*2 and *S*3, contains the names Dustin, Horatio, and Lubber.An equivalent expression is:

*πsname*(*πsid*((*πbidσcolor*=*′red′ Boats*) 𝝰⊳ *Reserves*) 𝝰⊳ *Sailors*)

The reader is invited to rewrite both of these queries by using *ρ* to make the interme- diate relations explicit and to compare the schemas of the intermediate relations. The second expression generates intermediate relations with fewer fields (and is therefore likely to result in intermediate relation instances with fewer tuples, as well). A rela- tional query optimizer would try to arrive at the second expression if it is given the first.

***(Q3) Find the colors of boats reserved by Lubber.***

*πcolor*((*σsname*=*′Lubber′ Sailors*) 𝝰⊳ *Reserves* 𝝰⊳ *Boats*)

This query is very similar to the query we used to compute sailors who reserved red boats. On instances *B*1, *R*2, and *S*3, the query will return the colors gren and red.

***(Q4) Find the names of sailors who have reserved at least one boat****.*

*πsname*(*Sailors* 𝝰⊳ *Reserves*)

***(Q5) Find the names of sailors who have reserved a red or a green boat.***

*ρ*(*T empboats,* (*σcolor*=*′red′ Boats*) *U* (*σcolor*=*′green′ Boats*))

*πsname*(*Tempboats* 𝝰⊳*Reserves* 𝝰⊳*Sailors*)

We identify the set of all boats that are either red or green (Tempboats, which contains boats with the *bid*s 102, 103, and 104 on instances *B*1, *R*2, and *S*3). Then we join with Reserves to identify *sid*s of sailors who have reserved one of these boats; this gives us *sid*s 22, 31, 64, and 74 over our example instances. Finally, we join (an intermediate relation containing this set of *sid*s) with Sailors to find the names of Sailors with these *sid*s. This gives us the names Dustin, Horatio, and Lubber on the instances *B*1, *R*2, and *S*3. Another equivalent definition is the following:

*ρ*(*T empboats,* (*σcolor*=*′red′*∨ *color*=*′green′ Boats*))

*πsname*(*Tempboats* 𝝰⊳*Reserves* 𝝰⊳ *Sailors*)

***(Q6) Find the names of sailors who have reserved a red and a green boat***

*ρ*(*T empboats*2*,* (*σcolor*=*′red′ Boats*) *∩* (*σcolor*=*′green′*

*Boats*)) *πsname*(*Tempboats*2 𝝰⊳ *Reserves* 𝝰⊳ *Sailors*)

However, this solution is incorrect—it instead tries to compute sailors who have re- served a boat that is both red and green. (Since *bid* is a key for Boats, a boat can be only one color; this query will always return an empty answer set.) The correct approach is to find sailors who have reserved a red boat, then sailors who have reserved a green boat, and then take the intersection of these two sets:

*ρ*(*T empred, πsid*((*σcolor*=*′red′ Boats*) 𝝰⊳ *Reserves*))

*ρ*(*T empgreen, πsid*((*σcolor*=*′green′ Boats*) 𝝰⊳ *Reserves*))

*πsname*((*Tempred ∩ Tempgreen*) 𝝰⊳ *Sailors*)

The two temporary relations compute the *sid*s of sailors, and their intersection identifies sailors who have reserved both red and green boats. On instances *B*1, *R*2, and *S*3, the *sid*s of sailors who have reserved a red boat are 22, 31, and 64. The *sid*s of sailors who have reserved a green boat are 22, 31, and 74. Thus, sailors 22 and 31 have reserved both a red boat and a green boat; their names are Dustin and Lubber.

This formulation of Query Q6 can easily be adapted to find sailors who have reserved red *or* green boats (Query Q5); just replace *∩* by ∪ :

*ρ*(*T empred, πsid*((*σcolor*=*′red′ Boats*) 𝝰⊳ *Reserves*))

*ρ*(*T empgreen, πsid*((*σcolor*=*′green′ Boats*) 𝝰⊳ *Reserves*))

*πsname*((*Tempred* ∪ *Tempgreen*) 𝝰⊳*Sailors*)

In the above formulations of Queries Q5 and Q6, the fact that *sid* (the field over which we compute union or intersection) is a key for Sailors is very important. Consider the following attempt to answer Query Q6:

*ρ*(*T empred, πsname*((*σcolor*=*′red′ Boats*) 𝝰⊳*Reserves* 𝝰⊳ *Sailors*))

*ρ*(*T empgreen, πsname*((*σcolor*=*′green′ Boats*) 𝝰⊳*Reserves* 𝝰⊳

*Sailors*)) *T empred ∩T empgreen*

This attempt is incorrect for a rather subtle reason. Two distinct sailors with the same name, such as Horatio in our example instances, may have reserved red and

green boats, respectively. In this case, the name Horatio will (incorrectly) be included in the answer even though no one individual called Horatio has reserved a red boat and a green boat. The cause of this error is that *sname* is being used to identify sailors (while doing the intersection) in this version of the query, but *sname* is not a key.

***(Q7) Find the names of sailors who have reserved at least two boats.*** *ρ*(*Reservations, πsid,sname,bid*(*Sailors* 𝝰⊳ *Reserves*)) *ρ*(*Reservationpairs*(1 *→ sid*1*,* 2 *→ sname*1*,* 3 *→ bid*1*,* 4 *→ sid*2*,* 5 *→ sname*2*,*6 *→ bid*2)*,Reservations × Reservations*) *πsname*1*σ*(*sid*1=*sid*2) *∩* (*bid*1=*bid*2)*Reservationpairs*

First we compute tuples of the form 〈 *sid,sname,bid* 〉 , where sailor *sid* has made a reservation for boat *bid*; this set of tuples is the temporary relation Reservations. Next we find all pairs of Reservations tuples where the same sailor has made both reservations and the boats involved are distinct. Here is the central idea: In order to show that a sailor has reserved two boats, we must find two Reservations tuples involving the same sailor but distinct boats. Over instances *B*1, *R*2, and *S*3, the sailors with *sid*s 22, 31, and 64 have each reserved at least two boats. Finally, we project the names of such sailors to obtain the answer, containing the names Dustin, Horatio, and Lubber.

Notice that we included *sid* in Reservations because it is the key field identifying sailors, and we need it to check that two Reservations tuples involve the same sailor. As noted in the previous example, we can’t use *sname* for this purpose.

***(Q8) Find the sids of sailors with age over 20 who have not reserved a red boat.***

*πsid*(*σage>*20*Sailors*) *−πsid*((*σcolor*=*′red′ Boats*) 𝝰⊳ *Reserves* 𝝰⊳ *Sailors*)

This query illustrates the use of the set-difference operator. Again, we use the fact that *sid* is the key for Sailors. We first identify sailors aged over 20 (over instances *B*1, *R*2, and *S*3, *sid*s 22, 29, 31, 32, 58, 64, 74, 85, and 95) and then discard those who have

reserved a red boat (*sid*s 22, 31, and 64), to obtain the answer (*sid*s 29, 32, 58, 74,

85, and 95). If we want to compute the names of such sailors, we must first compute their *sid*s (as shown above), and then join with Sailors and project the *sname* values.

***(Q9) Find the names of sailors who have reserved all boats.***

The use of the word *all* (or *every*) is a good indication that the division operation might be applicable:

*ρ*(*T empsids,* (*πsid,bidReserves*)*/*(*πbidBoats*))

*πsname*(*Tempsids* 𝝰⊳ *Sailors*)

The intermediate relation Tempsids is defined using division, and computes the set of *sid*s of sailors who have reserved every boat (over instances *B*1, *R*2, and *S*3, this is just *sid* 22). Notice how we define the two relations that the division operator (*/*) is applied to—the first relation has the schema *(sid,bid)* and the second has the schema *(bid)*. Division then returns all *sid*s such that there is a tuple 〈 *sid,bid* 〉 in the first relation for each *bid* in the second. Joining Tempsids with Sailors is necessary to associate names with the selected *sid*s; for sailor 22, the name is Dustin.

***(Q10) Find the names of sailors who have reserved all boats called Interlake****.*

*ρ*(*T empsids,* (*πsid,bidReserves*)*/*(*πbid*(*σbname*=*′Interlake′ Boats*))) *πsname*(*Tempsids* 𝝰⊳

*Sailors*)

The only difference with respect to the previous query is that now we apply a selection to Boats, to ensure that we compute only *bid*s of boats named *Interlake* in defining the second argument to the division operator. Over instances *B*1, *R*2, and *S*3, Tempsids evaluates to *sids* 22 and 64, and the answer contains their names, Dustin and Horatio.