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 5.7: Result of a theta-outerjoin

5.2.8 Exercises for Section 5.2

Exercise 5.2.1: 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 5.2.2:** 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 5.2.3:** One thing that can be done with an extended projection, but not with the original version of projection that we defined in Section 2.4.5, 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 from Section 2.4? Explain your reasoning.

5.3 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.

5.3.1 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 5.3.6, 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 5.16: 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 5.17: If R is the predicate from Example 5.16, 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 5.16, $R(x,y)$ 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. □

5.3.2 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 5.3.1 *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.

5.3.3 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 5.18: 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 5.18 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:

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

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 5.18, `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`.

5.3.4 Meaning of Datalog Rules

Example 5.18 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 5.18 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 5.19: Consider the rule

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

from Example 5.18. 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 5.20: 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 5.21: 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. 5.8 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 5.8: All possible assignments of tuples to $Q(x,z)$ and $R(z,y)$

The second and third options in Fig. 5.8 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. □

5.3.5 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 5.18, `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 5.24, 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.

5.3.6 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 5.3.4. 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 5.22: 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 5.23: 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 5.22; 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 $(4,5)$. □

5.3.7 Exercises for Section 5.3

Exercise 5.3.1: Write each of the queries of Exercise 2.4.1 in Datalog. You should use only safe rules, but you may wish to use several IDB predicates corresponding to subexpressions of complicated relational-algebra expressions.

Exercise 5.3.2: Write each of the queries of Exercise 2.4.3 in Datalog. Again, use only safe rules, but you may use several IDB predicates if you like.

!! **Exercise 5.3.3:** 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.

5.4 Relational Algebra and Datalog

Each of the relational-algebra operators of Section 2.4 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 5.35).