Objectives:

1. To introduce fundamental concepts of the relational model, including terminology, keys, nulls, views
2. To introduce the relational algebra
3. To introduce the relational calculus

Materials:

1. Transparencies of figures 3.8 and 3.9 in book
2. Transparency/handout of library example database used in class
3. Handout of natural language queries against this database
4. Transparency of cartesian join of checked_out and book tables

I. Introduction

A. The Relational Data Model is one of the data models that have been used for commercial DBMS's. Although it was first proposed in the 1970's, it has been the subject of intensive research that continues to today, and has become the dominant model in actual use. It will, therefore, be the major focus of this course.

B. One of the major advantages of the relational model is that it has a solid mathematical foundation. The relational model is based on the mathematical theory of relations. As a result, a large body of mathematical tools is available to support work with this model.

C. The Relational Model represents both entities and relationships via tables, in the way we discussed in connection with the entity-relationship model. (That is, an entity-relationship scheme that has been converted to tables is, in fact, essentially a relational scheme.)

1. This means that the relational model draws no distinction between entities and relationships; both are simply tables as far as the relational model is concerned, and are processed in exactly the same way.

2. However, a distinction between entities and relationships can be useful in the process of designing a database scheme. Thus, one can use the entity-relationship model as a design tool, with the relational model serving as a means of implementation. The conversion from design to implementation is straight-forward since, as we have already shown, both entities and relationships can easily be represented by tables.
3. One important place where the relational model imposes requirements not present in the entity-relationship model is that the relational model requires that attributes (table columns) be **atomic** and **single-valued**.

   a) The former requirement can be easily met by allocating one table column to each component of a composite attribute.

   b) The latter requirement necessitates creating a separate table to store the different values of the multivalued attribute - we will discuss this further when we discuss multivalued dependencies in connection with database design.

**II. Basic Terminology**

A. Because the relational model is grounded in the mathematical theory of relations, writers often use mathematical terms when discussing the relational model. However, since mathematical terminology can be intimidating to some, there is an alternate, non-mathematical set of terms that can also be used.

B. Terminology drawn from mathematical relations

1. A relational database is a collection of relations.

2. Formally, a relation is a set of ordered tuples of some arity.

   a) Note that the term tuple is not as unfamiliar as it sounds. For example, if a woman gives birth to two children, we say she has had twins; but if she has five, we say she has had quintuplets; if six, sextuplets etc.

   b) The arity of a tuple is the number of components in it - e.g.

   ( aardvark, anthony) is a tuple of arity 2

   ( 123-45-6789, aardvark, anthony, $30000, 999-99-9999) is a tuple of arity 5

   c) Within any given relation, all the tuples have the same arity.

   d) When we say that the tuples are ordered, we mean that the order of listing the components is important - e.g.

   (aardvark, anthony) is not the same tuple as (anthony, aardvark)
e) Since a relation is a set, the order of the tuples themselves is immaterial - e.g. the following are the same relation

(aardvark, anthony)
(elephant, emily)

and

(elephant, emily)
(aardvark, anthony)

3. The components of a tuple are generally called attributes.
   Mathematically, each attribute of a tuple is given a number - e.g. in (aardvark, anthony), aardvark is attribute 1 and anthony is attribute 2. (Note that, by convention, 1-origin indexing is used.) However, in relational databases, we normally give the attributes names (perhaps last_name and first_name, in this case.)

4. Further, each attribute of a tuple is drawn from a specific domain, or set of possible values.

   Example: the last_name attribute above is drawn from the set of all possible last names

   In many cases, actual relational DBMS software only allows the domain to be specified in terms of a basic data type - e.g. "integer" or "string of 30 characters".

   a) Some systems do, however, allow the enforcement of domain constraints beyond basic data type.

   b) In such a system, proper specification of domains could prevent semantic errors, such as mistakenly storing a ZIP+4 ZIP code (9 digits) into a social security number field!

5. The specification of the structure of tuples in a relation is called a relation scheme.

Example: suppose we had a relation in which the first attribute of each tuple was drawn from the set of ASCII characters and the second from the set of integers. (Perhaps this relation comprises an ASCII code table.) We could describe the relation scheme for this relation as follows:
ASCII_code_table_scheme = ( character: char, code: integer)

or, omitting the domain specifiers:

ASCII_code_table_scheme = ( character, code)

(Note the analogy to a type in a programming language.)

6. A specific relation on some scheme is called a relation instance or an instance of that scheme

(Note the analogy to the value of a variable in a programming language)

7. A relation instance on some scheme is, in fact, a subset of the cartesian product of the domains of its attributes. (It can be any subset, including an improper subset or the empty subset)

Example: Suppose we had a relation on the following scheme:

( One_digit_integer: 0..9, flag: boolean)

The domain of the first attribute would be

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

and that of the second would be

{ false, true }

The Cartesian product of these domains is:

<table>
<thead>
<tr>
<th>0</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>true</td>
</tr>
<tr>
<td>9</td>
<td>true</td>
</tr>
</tbody>
</table>

Any relation on this scheme would be a subset of this cartesian product. E.g. the table which indicates whether a given integer is odd would be the subset:
C. Alternate terminology

1. We may also speak of a relational database as a collection of tables. (A table, then is another name for a relation.)

2. Each table, in turn, consists of some number of rows and columns. (A row, then, is another name for a tuple; and a column is another name for an attribute.)

3. The relation-scheme is often represented as headings on the columns - e.g.

<table>
<thead>
<tr>
<th>last_name</th>
<th>first_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>aardvark</td>
<td>anthony</td>
</tr>
<tr>
<td>cat</td>
<td>charlene</td>
</tr>
</tbody>
</table>

D. Keys

1. The terminology we developed for keys in connection with the entity-relationship model carries over directly to the relational model.

   a) Because a relation is a set, the tuples comprising it must be distinct (no two can be identical)

   b) A superkey for a relation is a set of attributes which serve to distinguish any tuple in the relation from all others - i.e. each tuple in the relation will have a different set of values for superkey attributes than any other tuple in the relation (some attributes may have the same value as long as at least one differs)

   c) A candidate key is a superkey which has no proper subset that is also a superkey.

   d) A primary key is one particular candidate key chosen by the designer to serve as the basis for identifying tuples.
2. Because of the way we converted entities to tables, it will always be true that any relation will have a primary key (though perhaps this primary key will, in fact, be all of the attributes together.) This is because we converted weak entities to tables in which we included not only the attributes of the weak entity but also the primary key of the strong entity on which its existence depends.

3. An important notion in relational database design is the notion of a foreign key. When a column or group of columns in one table is the primary key of some other table, we call this a foreign key (the key of some other table is present in this table.) Foreign keys are, of course, the way we represent relationships in relational tables.

E. Nulls

1. One interesting question that arises in database design is how are we to handle a situation where we don't have values available for all the attributes of an entity.

Example: Suppose we need to add a borrower to our borrower-scheme, but for some reason we don't know the person's phone number. (Perhaps it's unlisted, or perhaps he forgot to give it to us when he filled out his forms.)

2. For cases like this, most DBMS's provide a special value called NULL that can be stored in a field. When we print out the field, NULL will normally print as spaces; but storing NULL in a field is NOT the same as storing a string of spaces (i.e. the DBMS software distinguishes between NULL and "").

3. One property of NULL is that it is never treated as any value during a query - the principle being that, since NULL means we don't know a value, it should never participate in any query. Thus, if we compare an attribute whose value is NULL to some value, the result of the comparison is always false, regardless of which comparison (=, <, >, <=, >= or <> ) we use. In fact, it is even the case that NULL = NULL is false, and so is NULL <> NULL. Further, a NULL value never participates in any aggregate operations such as average.

   a) Example: suppose we are doing a natural join between two relations which each contain a phone-number attribute. If a tuple in the first relation contains a NULL phone-number, it will not join with any tuple in the second relation - even one that also has a NULL phone-number. (Since NULL = NULL is false!)
b) Example: suppose we are calculating the average salary for all
employees in a company. If the salary for some employee is stored
as NULL, then - for the purposes of taking the average - it will be
as if that employee were not present. (Notice that this gives a
different result than if we took the salary to be 0.)

4. In designing a database, it will sometimes be necessary to specify that
certain fields CANNOT ever contain a NULL value. This will certainly
be true of any field that is part of the primary key, and may be true of
other fields as well. Most DBMS's allow the designer to specify that a
given field cannot be NULL.

F. An important concept is the notion of a view. We will introduce these
later, in conjunction with the relational algebra (once we have learned the
concepts needed to specify a view) - but a view is a fundamental notion in
a relational database, not specific to the relational algebra.

G. The book gives a diagrammatic form for representing a relational schema
(which is quite similar to a UML diagram, though not identical)

TRANSPARENCIES: Figure 3.8 from book (an E-R diagram), plus
Figure 3.9 from book (an equivalent relational schema)

Note the explicit representation of foreign keys

III. Query Languages

A. For the rest of this series of lectures, we will make use of the following
very simple example database for a small library. The structure of this
database has been contrived to allow us to illustrate most kinds of query
operations we might want to perform. (Relations have been kept
unrealistically small, both in terms of the number of rows and in terms of
the number of columns, to simplify the examples done in class. In
particular, we assume that the library only has one copy of any given
book.) Note: primary key attributes are underlined

borrower(borrower_id, last_name, first_name)

12345    aardvark    anthony
20147    cat    charlene
89754    dog    donna
60984    fox    frederick
54872    zebra    zelda
B. One of the main reasons for having a database is to be able to answer queries, such as the following that might be posed against our example:

HANDOUT

Who is the borrower whose borrower id is 12345?
List the names of all borrowers
What is the title of the book whose call number is QA76.093?
List the titles of all books that are currently checked out.
List the names of all borrowers having one or more books overdue.
List the names of all employees who earn more than their supervisor.

List the names of all people connected with the library - whether borrowers, employees, or both.

List the names of all borrowers who are not employees.

List all books needed as course reserves that are currently checked out to someone.

List the names of employees together with their supervisor’s name.

List the call numbers of all overdue books, together with the number of days they are overdue.

What is the average salary of all employees?

Print a list of borrower id's and the number of books each has out

List the titles of all books, together with the borrower id of the person (if any) who has the book out.

1. Queries such as these are termed natural language queries. Of course, most commercial DBMS's cannot accept queries like this, though natural language query capability is a major area of current research in AI. Instead, most DBMS's use a more formal query language. Even those with natural language capabilities typically consist of a regular query language plus a natural language front end - e.g.

   ![Diagram of natural language query process]

   Natural language query → "Front end" software → Actual DBMS query language equivalent → DBMS software

2. All DBMS's support one or more query languages. Typically, these can be used in two ways:
   
   a) Interactively, by a user.

   b) Embedded within programs written in traditional programming languages.
C. Query languages are of two general types: procedural and non-procedural.

1. A procedural language requires the user to spell out the series of steps needed to satisfy his query.

2. A non-procedural language allows the user to simply specify what he/she wants, with the DBMS figuring out the steps needed to get it.

Example: Suppose one wanted to find the roots of the following equation:

\[ x^2 + 3x + 2 = 0 \]

In a non-procedural language, one would simply specify the equation and indicate in some way that an appropriate value of x is wanted. ("Find the value of x such that \( x^2 + 3x + 2 = 0 \))

In a procedural language, one would specify steps like:

\begin{align*}
\text{Compute discriminant} &= 3^2 - 4 * 1 * 2 \\
\text{Compute root1} &= (-3 + \sqrt{\text{discriminant}}) / 2 \\
\text{Compute root2} &= (-3 - \sqrt{\text{discriminant}}) / 2
\end{align*}

Note that our sample English queries were all of a non-procedural sort. A major part of the challenge in developing a natural language front-end for a DBMS is the process of converting a non-procedural query to a correct series of procedural steps.

3. Commercial languages may include mixtures of procedural and non-procedural features.

D. Another way to classify query languages is as formal and commercial

1. A formal query language uses mathematical notation and concepts, and is not readily intelligible to the average user.

2. Commercial languages use various forms of syntactic sugar to enhance readability and ease of use.

3. Though commercial languages are what is used in actual DBMS applications, formal languages are worthy of some study:
a) All commercial languages build on underlying formal language concepts.

b) Formal languages provide a vehicle for research (especially the proving of various theorems) that generates results useful in the development of commercial languages.

4. The book discussed three formal query languages:

a) The relational algebra - a procedural formal query language. This underlies most (but not all) commercial DBMS products - including the SQL language that widely used as a cross-vendor standard.

b) Two variants of the relational calculus - both non-procedural formal query languages

IV. The Relational Algebra

A. Queries in the relational algebra are formulated in terms of six basic operations on relations. Three operate on a single relation to give a new relation; and three operate on pairs of relations to give a new relation.

1. The operation of SELECTION (denoted by the Greek letter sigma - $\sigma$) selects rows of a table that meet certain criteria. The result is a new table with (generally) fewer rows than the original table, but with the same columns.

Example: What is the relational algebra expression corresponding to our first English query: "Who is the borrower whose id is 12345?"?

ASK

$\sigma$ borrower

borrower_id = 12345

What is the result of this query?

ASK

12345 aardvark anthony

Note, then, that selection reduces the table by squeezing out rows that do not satisfy the selection condition.
2. The operation of PROJECTION (denoted by the Greek letter pi - $\pi$) chooses only specific columns from all the rows of a table.

Example: What is the relational algebra expression corresponding to our second query: "List the names of all borrowers"?

ASK

$\pi$ borrower
last_name
first_name

What is the result of this query?

ASK

aardvark  anthony
cat  charlene
dog  donna
fox  frederick
zebra  zelda

Example: our third query ("What is the title of the book whose call number is QA76.093?") could be satisfied by a selection from book to get the row for the book in question, followed by a projection on the book-title attribute. (I.e. composing two relational algebra operations) - written as

$\pi$ book
title   call_number = QA76.093

What is the result of this query?

ASK

Wenham Zoo Guide

Note, then, that projection reduces the table by squeezing out unwanted columns.

a) One possible problem with a projection operation is that it could produce a result with duplicate rows, by projecting out the attribute that distinguishes them.
Example: Project book on the author attribute. We now have two books with the same author.

b) This is a problem, since a relation is a set, and a set cannot have the same element appear twice. Thus, mathematically, the result of a projection has any such duplications eliminated. However, the SQL language (which is based on relational algebra) does not do this automatically, because of the time involved - though there is a SQL query language feature to specify that duplicates are to be eliminated.

3. The operation JOIN takes the cartesian product of two tables, and is used when a query involves information contained in more than one table. (A cartesian product means pairing each row from the first table with each row from the second.) Join is denoted by a large "X".

Example: Our next two queries would require a join.

“List the titles of all books that are currently checked out” requires information from the checked_out table (to find out the call number(s) of book(s) currently checked out) and from the book table (to get the titles)

“List the names of all borrowers having one or more books overdue” requires information from the checked_out table (to find out the ids of borrowers having books overdue) and from the borrower table (to get their names)

a) Taken by itself, the join produces a rather large result.

Example: consider checked_out X book, required for our first query:

<table>
<thead>
<tr>
<th>borrower id</th>
<th>call number</th>
<th>date-due number</th>
<th>title</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>89754</td>
<td>RZ12.905</td>
<td>11-10-02</td>
<td>QA76.093 Wenham Zoo Guide</td>
<td>elephant</td>
</tr>
<tr>
<td>89754</td>
<td>RZ12.905</td>
<td>11-10-02</td>
<td>RZ12.905 Fire Hydrants</td>
<td>dog</td>
</tr>
<tr>
<td>89754</td>
<td>RZ12.905</td>
<td>11-10-02</td>
<td>LM925.04 21 Ways to Cook</td>
<td>koala</td>
</tr>
<tr>
<td>89754</td>
<td>RZ12.905</td>
<td>11-10-02</td>
<td>AB123.40 Karate</td>
<td>koala</td>
</tr>
<tr>
<td>89754</td>
<td>LM925.04</td>
<td>11-10-02</td>
<td>QA76.093 Wenham Zoo Guide</td>
<td>elephant</td>
</tr>
<tr>
<td>89754</td>
<td>LM925.04</td>
<td>11-10-02</td>
<td>RZ12.905 Fire Hydrants</td>
<td>dog</td>
</tr>
<tr>
<td>89754</td>
<td>LM925.04</td>
<td>11-10-02</td>
<td>LM925.04 21 Ways to Cook</td>
<td>koala</td>
</tr>
<tr>
<td>89754</td>
<td>LM925.04</td>
<td>11-10-02</td>
<td>AB123.40 Karate</td>
<td>koala</td>
</tr>
<tr>
<td>20147</td>
<td>AB123.40</td>
<td>11-15-02</td>
<td>QA76.093 Wenham Zoo Guide</td>
<td>elephant</td>
</tr>
<tr>
<td>20147</td>
<td>AB123.40</td>
<td>11-15-02</td>
<td>RZ12.905 Fire Hydrants</td>
<td>dog</td>
</tr>
<tr>
<td>20147</td>
<td>AB123.40</td>
<td>11-15-02</td>
<td>LM925.04 21 Ways to Cook</td>
<td>koala</td>
</tr>
<tr>
<td>20147</td>
<td>AB123.40</td>
<td>11-15-02</td>
<td>AB123.40 Karate</td>
<td>koala</td>
</tr>
</tbody>
</table>
(1) Because the checked_out table contains 3 rows, and the book table 4 rows, the result of the join contains $3 \times 4 = 12$ rows.

(2) Because the checked_out table has three columns and the book table has three columns, the result of the join has $3 + 3 = 6$ columns.

(3) This will always be true for a simple join: the result has as many rows as the product of the numbers of rows of the two tables, and as many columns as the sum of the numbers of columns of the two tables. (For the former reasons, simple join is sometimes called product or times.)

(4) Question: For the second query - requiring a join of checked_out and borrower) how many rows would the result table have? How many columns?

**ASK**

$3 \times 5 = 15$ rows

$3 + 3 = 6$ columns

b) However, when we do a join, generally only certain rows are meaningful.

Example: in the above, only the rows in which the two values of call number are the same tell us anything worth knowing - i.e. the fact that book RZ12.905 is checked out means that the title of RZ12.905 is of interest. On the other hand, the row that pairs RZ12.905 with QA76.093 means nothing. Thus, a join is almost always immediately followed by a select.

c) In doing the select, of course, one must use attribute names from both relations in the join. Since there is no rule that each relation must have unique attribute names (and, in fact, there are good reasons for having common names between relations), one must often qualify attribute names by the name of the relation they come from.

Example: Meaningful information would be produced by
\[ \pi \ (\text{checked\_out} \ X \ \text{book}) \]
\[ \text{checked\_out.call\_number} = \text{book.call\_number} \]

Yielding:

<table>
<thead>
<tr>
<th>borrower id</th>
<th>call number</th>
<th>date due</th>
<th>title</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>89754</td>
<td>RZ12.905</td>
<td>11-10-02</td>
<td>Fire Hydrants ... dog</td>
<td></td>
</tr>
<tr>
<td>89754</td>
<td>LM925.04</td>
<td>11-10-02</td>
<td>21 Ways to Cook .. dog</td>
<td></td>
</tr>
<tr>
<td>20147</td>
<td>AB123.40</td>
<td>11-15-02</td>
<td>Karate koala</td>
<td></td>
</tr>
</tbody>
</table>

If we now project this on title, we get the answer to our query about title of books checked out:

\[ \pi \ (\text{checked\_out} \ X \ \text{book}) \]
\[ \text{title} \ \text{checked\_out.call\_number} = \text{book.call\_number} \]

(Not that we do not need to qualify the title attribute, since it appears in only one relation.)

What is the result of this query?

**ASK**

- Fire Hydrants I Have Known
- 21 Ways to Cook a Cat
- Karate

**d)** In similar fashion, our second query (“List the names of all borrowers having one or more books overdue.”) can be answered by using a join with a select and project as above, but with additional selection conditions.

**ASK**

\[ \pi \ (\text{checked\_out} \ X \ \text{borrower}) \]
\[ \text{last\_name, first\_name} \ \text{checked\_out.borrower\_id} = \text{borrower.borrower\_id} \wedge \text{date\_due < today} \]

4. The operation RENAME takes a given table and gives it a new name, possibly also renaming its attributes as well. Rename is denoted by the Greek letter rho - \( \rho \).
a) This operation is most often useful in conjunction with joins, especially when doing a join between a table and itself.

Example: Our query “List the names of all employees who earn more than their supervisor” requires us to join the employee table with itself. We can do this by

\[
\pi \quad (\text{employee} \times \text{employee})
\]

\[
\text{employee.} \quad \text{employee.supervisor.ssn} = \quad \text{supervisor.ssn} \\
\text{employee.} \quad \text{employee.salary} > \quad \text{supervisor.salary}
\]

The rename operation here causes the second copy of employee to be called supervisor for the purposes of qualifying names in the select and project.

What is the result of this query?

ASK

    elephant    emily
    fox         frederick

b) Actually, as the example above suggests, when we need to qualify column names, it is often convenient to rename the tables in a join to simplify the names involved - e.g. the above could be simplified to:

\[
\pi \quad (\text{employee} \times \text{employee})
\]

\[
\text{e.last_name} \quad \text{e.supervisor.ssn} = \quad \text{e.ssn} \\
\text{e.first_name} \quad \text{e.salary} > \quad \text{s.salary}
\]

c) The rename operation can also be used to rename the columns of a table - we will see an example of this shortly

5. The operation UNION takes two relations on the same scheme and combines them into one, eliminating duplicate tuples.

Example: Our query “List the names of all people connected with the library - whether borrowers, employees, or both.” requires combining information from the borrower table and the employee table. A join is not what we want - in no sense would pairing a borrower and an
employee make sense - we want to know the names of people who are either or both. What query would do the job?

**ASK**

\[
(\pi \text{ borrower}) \bigcup (\pi \text{ employee})
\]

\[
\text{last_name, last_name, first_name first_name}
\]

What would be the result of this query?

**ASK**

<table>
<thead>
<tr>
<th>aardvark</th>
<th>anthony</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>charlene</td>
</tr>
<tr>
<td>dog</td>
<td>donna</td>
</tr>
<tr>
<td>fox</td>
<td>frederick</td>
</tr>
<tr>
<td>zebra</td>
<td>zelda</td>
</tr>
<tr>
<td>buffalo</td>
<td>boris</td>
</tr>
<tr>
<td>elephant</td>
<td>emily</td>
</tr>
</tbody>
</table>

(In some order - not necessarily this!)

a) The symbol is the familiar set algebra symbol for union: \bigcup

b) Because relations are sets, duplicate rows that would result from a union are discarded - e.g., in the above, suppose an someone were both a borrower and an employee - then their name would appear only once, even though it is in both tables. (Though for efficiency reason, many relational DBMS’s allow you to avoid this operation and accept duplicates, since discarding duplicates takes additional processing.)

Example: in the above, there is only one row for aardvark and fox

c) Union frequently arises when similar tables are created as a result of generalization or specialization.

d) Union only makes sense when the participating tables have the same or similar schemes. In fact, they must have the same arity, and corresponding attributes must be drawn from the same domain.

In the above, we projected the last_name and first_name attributes before taking the union, since salary and supervisor are unique to employees, and borrower_id and ssn, while they play similar roles, are different.
e) Sometimes, we may have a case where tables being combined by union have similar attributes with different names. In this case, we may need to rename individual attributes.

Example: Suppose we wanted our list to include an “id” attribute in both cases - choosing either the borrower_id or the ssn as appropriate. This could be done by:

\[
(\pi \text{ borrower}) \bowtie (\pi \text{ employee})
\]

\[
t1(\text{id, borrower_id, last_name, first_name}) 
\]
\[
t2(\text{id, ssn, last_name, first_name})
\]

(Note that we use \( \bowtie \) to rename the “id” attributes to both be called id in the result.)

(Note that, in this case, a person who was both a borrower and an employee would definitely appear twice in the result - one with each id!)

6. The operation of DIFFERENCE takes two relations on the same scheme and returns a relation having only those tuples which appear in the first but not the second.

Example: Consider the query “List the names of all borrowers who are not employees.” How would this be accomplished?

\[
(\pi \text{ borrower}) \setminus (\pi \text{ employee})
\]

What would be the result of this query?

\[
\text{ASK}
\]

\[
\text{ASK}
\]

a) The symbol is the familiar set algebra symbol for difference: -
b) As with union, the difference operation only makes sense if the tables involved have the same arity, and corresponding attributes come from the same domain.

B. There are three other relational operations that are sometimes included in a query language. Strictly speaking, they are not necessary, since they can be expressed in terms of the six primitives (selection, projection, join, rename, union, and difference). But they are useful enough to include as operations in their own right anyway.

1. The operation of INTERSECTION is similar to union and difference, in that it combines two relations having the same scheme. But where union produces all tuples that occur in either table, and difference gives those tuples in the first relation but not the second, intersection gives only those tuples that occur in BOTH tables.

Example: How would we answer the question “List all books needed as course reserves that are currently checked out to someone”?

\[ \text{ASK} \]
\[
(\pi \text{reserve_book}) \text{ call_number} \quad \Box \quad (\pi \text{checked_out}) \text{ call_number}
\]

What would be the result of this query?

\[ \text{ASK} \]
\[
\text{AB123.40}
\]

(Joining could now be used to get additional information such as the Title of the book and/or the borrower who has it.)

a) The symbol for intersection is the familiar set algebra symbol \( \Box \)

b) If intersection is not available in a query language, it can be implemented from the other primitives as follows:

\[
to \text{ form the intersection of relations R1 and R2, compute} \]
\[
R1 - (R1 - R2)
\]

1) \( R1 - R2 \) computes all tuples in \( R1 \) but not in \( R2 \)
(2) therefore R1 - (R1 - R2) leaves only those tuples from R1 which also occur in R2 - i.e. all tuples occurring in BOTH R1 and R2.

c) As with union and difference, intersection only makes sense if both tables have the same arity and corresponding attributes come from the same domains. (Note how we used projection in the example to facilitate this.)

2. The operation called NATURAL JOIN is a specialized version of the join that is often useful.

a) Recall, when we first introduced join, that we said it is often the case that when we perform a join between two tables that have some attribute(s) in common, we are only interested in the rows where the attributes have the same value.

Example: when we joined checked_out to book to answer the query “List the titles of all books checked out”, we were only interested in rows where the call_number from checked_out was the same as call_number from book - hence our join was followed by a select that looked like this:

```
\[
\text{select (checked_out X book) where checked_out.call_number = book.call_number}
\]
```

Yielding:

<table>
<thead>
<tr>
<th>borrower_id</th>
<th>call_number</th>
<th>date_due</th>
<th>call_number</th>
<th>title</th>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>89754</td>
<td>RZ12.905</td>
<td>11-10-02</td>
<td>RZ12.905</td>
<td>Fire Hydrants ...</td>
<td>dog</td>
</tr>
<tr>
<td>89754</td>
<td>LM925.04</td>
<td>11-10-02</td>
<td>LM925.04</td>
<td>21 Ways to Cook . .</td>
<td>dog</td>
</tr>
<tr>
<td>20147</td>
<td>AB123.40</td>
<td>11-15-02</td>
<td>AB123.40</td>
<td>Karate koala</td>
<td></td>
</tr>
</tbody>
</table>

We call an operation like this an EQUIJOIN.

b) Now suppose we wanted all the information - not just the title of the book. Would the above be a good answer to our question?

ASK

Not really - when the two relations in the join have some attribute in common, it makes sense to include it only once in the resultant relation. (In the above, call_number appears twice). Thus, it would
make sense to follow the selection is often followed immediately by projection.

Example:

\[
\pi_{\text{checked} \_ \text{out}}\ 
\begin{array}{c}
\text{call} \_ \text{number} \\
\text{borrower} \_ \text{id} \\
\text{date} \_ \text{due} \\
\text{title} \\
\text{author}
\end{array}\ 
\begin{array}{c}
\text{checked} \_ \text{out} \\
\text{call} \_ \text{number} = \\
\text{book} \\
\text{call} \_ \text{number}
\end{array}
\]

(c) Because this combination of operations occurs so frequently, it is often useful to define special kind of join, known as the NATURAL JOIN - of which the join we just considered is actually an example.

(1) A natural join is typically done between two tables having one or more attribute names in common. Only those rows in the joined table which agree on the common attributes are kept, and only one copy of each common attribute appears in the final result. (This is, in fact, the definition of the natural join.)

(2) The natural join is denoted by the symbol \(|X|\)

Example: the last table we constructed is in fact

\[
\text{checked-out} \ |X| \text{book}
\]

(3) Note that natural join is not a new operator, but rather a combination of already defined operations join, select, and project. However, we distinguish it for two reasons:

(a) simplicity of notation.

(b) Efficiency: Some DBMS’s support a natural join operation in their query language. (E.g. mySQL does, but IBM’s DB2 does not.) It is more efficient for a DBMS to do the selection and projection as it does the join, rather than doing a full cartesian product and then paring it down!

(Of course, a "smart" DBMS will recognize natural join possibilities even in the user does a full cartesian product and then does a select and/or project on it. In fact, even though
IBM’s db2 does not have a natural join primitive operation, it does recognize and optimize when this pattern occurs.

(4) Note that natural join is defined in terms of attribute names. This presupposes that when columns in two different tables have the same name, they refer to the same thing. If the different names mean different things, the result could be undesirable - e.g. suppose our employee table included a “title” attribute which was the employee’s job title - then

book \( \Join \) employee

would give a table listing employees whose job title is the same as the title of some book!

3. Rarely, we will need to do a join operation where some comparison other than strict equality of values of attributes of the same name is needed.

Example: our query “List the names of all employees together with their supervisor’s name.” Here, we want to join the employee table with itself, on the condition that the supervisor_ssn from the “left” copy matches the regular ssn from the “right” copy.

This can be done using a conventional cartesian product, followed by selection (and then projection):

\[
\pi_{\text{e.last_name, e.first_name, s.last_name, s.first_name}} (\text{employee} \times \text{employee})
\]

\[
e_{\text{last_name}} e_{\text{first_name}} s_{\text{last_name}} s_{\text{first_name}}
\]

a) We sometimes give such a cartesian product followed immediately by a selection a theta join - which we can denote as follows:

\[
\pi_{\text{e.last_name, e.first_name, s.ssn}} (\text{employee} \times \text{employee})
\]

\[
e_{\text{last_name}} e_{\text{first_name}} s_{\text{ssn}}
\]

b) In the book, the notation used for theta join implies that it always involves a natural join - which is not the case.
The operation of DIVISION is a rather unusual and complex one, but one that is useful in certain situations - namely queries in which a phrase "for all" or "every" occurs.

a) To get a meaningful example, we will depart from our library database to use student registration as an illustration.

Suppose we have a relation

\[
courses_{\text{taken}}(\text{student}\_\text{id}, \text{course}\_\text{id})
\]

which lists ALL the courses a student has successfully completed, and another relation

\[
core_{\text{requirements}}(\text{course}\_\text{id})
\]

which lists all the core courses required. (To simplify, we assume no options in the core - i.e. no requirement of the form "take either xxxxx or yyyyy").

Now, suppose we wish to form a list of students eligible to graduate (at least as far as the core is concerned.) These will be those students who have taken EVERY course appearing in core_requirements. It turns out that the division operation gives us what we want - i.e.

\[
courses_{\text{taken}} / core_{\text{requirements}}
\]

will give us a relation on the scheme (student_id) listing only those students who have taken EVERY core_requirement.

b) In general, the operation \( r / s \) is possible only if the scheme for \( s \) (\( S \)) is a proper subset of the scheme for \( r \) (\( R \)). The scheme of the result is those attributes of \( r \) not appearing in \( s \): \( R-S \). (E.g. (course_id) is a subset of (student_id, course_id), and the result-scheme (student_id) is (student_id, course_id) - (course_id).

c) The operation is called division because it is, in some sense, the inverse of cartesian product. To see this, note that if we now take the result of \( courses_{\text{taken}} / core_{\text{requirements}} \) and join it with core_requirements, we will get back a relation on the original scheme for courses_taken that is a subset of courses_taken - including only the entries for students having fulfilled all the core.
(The entries in the original relation that are not in the product

\(\text{courses\_taken/core\_requirements) X core\_requirements}\)

may be thought of as the "remainder" of the division.)

d) If a DBMS does not provide division as a primitive, the operation
can be implemented from the other primitives as follows (assuming
a relation \(r\) on scheme \(R\) and a relation \(s\) on scheme \(S\), with \(R \subseteq S\))

\[ r / s ::= (\pi_{r}) - \pi_{\left(((\pi_{r}) X s) - r\right)} \]

\(R-S\) \(R-S\) \(R-S\)

(1) the first projection gives us a list of all tuples that could appear
in the quotient

(2) from this, we subtract the second projection, which gives us the
ineligible tuples.

(a) \((\pi_{r}) X s\) gives us the tuples that would appear in
\(R-S\)

\(r\) if every tuple in \(r\) were eligible to participate in the result -
i.e. a relation in which every value in \(\pi_{r}\) \(r\) is, in fact,
paired with every value in \(s\) \(R-S\).

(b) Subtracting \(r\) from this gives us the tuples that are missing
from \(r\) for this to be true. Projecting this on \(R-S\) gives us the
ineligible tuples on \(R-S\).

C. Sometimes, a purportedly relational DBMS will implement some, but not
all, of the relational primitives. Such a system is not, then, fully relational.
Relational-like systems can be classified as follows:

1. TABULAR systems store data as tables, but do not fully support all
three of the most basic operations select, project, and join.

2. MINIMALLY RELATIONAL systems support only select, project,
and join operations.

3. RELATIONALLY COMPLETE systems support the primitives select,
project, join, rename, union, and difference. The other operations
(natural join, intersection and division) may also be provided; but if not
they can always be built up from the primitives.
D. There are some additional operations that extend the basic relational algebra, which are sometimes called “extended relational algebra”.

1. One such extension is generalized projection - allow a projection expression to contain computations based on column values, not just column values themselves.

Example:

List the call numbers of all overdue books, together with the number of days they are overdue.

\[ \pi \text{ call_number} \text{ checked_out} \text{ date_due < today} \text{ today - date_due} \]

What is the result of this query?

2. A related extension is allowing the use of AGGREGATE FUNCTIONS to produce a single summary value from a set of tuples.

   a) The simplest case is applying functions like sum, max, min, average to a single column, or count to an entire table.

   Example: What is the average salary of all employees?

   \[ G \text{ employee} \text{ average(salary)} \]

   b) An aggregate function can also be used with a grouping operator to produce summaries by groups

   Example: Print a list of borrower id's and the number of books each has out

   \[ G \text{ checked_out} \text{ borrower_id} \text{ count(call_number)} \]
What is the result of the above query?

**ASK**

|   |    |  
|---|----|---|
| 89754 |   2 |
| 20147 |   1 |

3. Another such extension is the OUTER JOIN.

a) The ordinary join operation will omit from the result any row from either table that does not join with some row from the other table.

Example:

List the titles of all books, together with the borrower id of the person (if any) who has the book out.

As a first attempt, we could use the following approach:

\[ \pi \text{ title, borrower}\_id \text{ book} \mid \text{checked}\_out \]

What is the result of this query?

**ASK**

- Fire Hydrants I Have Known
- 21 Ways to Cook a Cat
- Karate

Note that there is no row in this table for Wenham Zoo Guide. Why?

**ASK**

Wenham Zoo Guide is not checked out, so there is no row in the checked\_out table for the row in book to join with.

b) If we want to be sure all books are included, even if not checked out, we can use the outer join:

\[ \pi \text{ title, borrower}\_id \text{ book} \mid \text{checked}\_out \]

26
which will pair any tuple in book that does not join with any row in checked_out with a “manufactured” row of all nulls

c) The outer join comes in three versions

(1) The left outer join - which we used above - joins any tuple in the left relation that does not join with anything in the right relation with a dummy tuple of all nulls. Any tuple in the right relation that does not match a tuple in the left is absent from the result.

(2) The right outer join - denoted $\text{leftouter}$ - joins any tuple in the right relation that does not join with anything in the left relation with a dummy tuple of all nulls. Any tuple in the left relation that does not match a tuple in the right is absent from the result.

(3) The full outer join - denoted $\text{outer}$ combines the above - hence no tuple is omitted.

d) The example we looked at considers the outer join in conjunction with a natural join. It is also possible to do an outer join in conjunction with a theta join.

Example: Earlier, we considered a query for “List the names of all employees together with their supervisor’s name.”

\[
\pi \text{ employee } (\text{employee} \bowtie \text{employee})
\]
\[
e.\text{last_name} \quad e.\text{supervisor_ssn} = e.\text{ssn} \\
e.\text{first_name} \\
s.\text{last_name} \\
s.\text{first_name}
\]

This query does not produce a row for aardvark as an employee, because aardvark’s supervisor is null. (Presumably, he’s the head honcho.) If we used a theta outer join instead, we would get aardvark listed with no supervisor:

\[
\pi \text{ employee } (\text{employee} \bowtie \text{employee})
\]
\[
e.\text{last_name} \\
e.\text{first_name} \\
s.\text{last_name} \\
s.\text{first_name}
\]
E. Modifying the Database

1. The relational algebra, as we have developed it, only deals with queries, but provides no way to alter the database. To add this capability, we need only add the assignment operator. 

2. Deletion: general form is

   relation - query expression for row(s) 

Example: Delete all books written by dog

   author = dog

3. Insertion: general form is

   relation relation row(s) to be added 

Example: Add 65432 raccoon ralph to the database:

   borrower - borrower (65432, raccoon, ralph)

Example: Suppose we merge with another library, which has a relation

   patron(patron_id, given_name, surname, middle_initial)

We could add all of these patrons to our borrower table with a single operation - note use of rename and project with reordering of columns to make schemes conform:

   borrower - borrower (patron)
   p(borrower_id, last_name, first_name)

4. Update: general form is relation π relation formulas for changes 

   or

   relation changed tuples (relation - tuples to be changed)
Example: Give all employees a 5% pay raise

\[
\text{employee} \left[ \pi \text{last_name, first_name, salary}\times 1.05, \text{supervisor_ssn} \right]
\]

F. Views

1. In our introduction to the course, we noted that a database can be discussed at three levels: physical, logical, and view. So far, our discussion of relational databases has all been at the logical level.

2. Using relational algebra, it is possible to define a view by using a CREATE VIEW name AS expression, where expression is a relational algebra operation that defines the view.

Example: we might want to allow some users to access a list of employee names without seeing the ssn of the employee or the employee’s supervisor or salaries. We could do this by

\[
\text{create view emp as } \pi \text{last_name, first_name}
\]

or, we might want to let a supervisor see full information on only those employees he/she supervises

\[
\text{create view aardvarks_emps as } \text{employee} \\
\text{supervisor_ssn = 123-45-6789}
\]

3. Note that any relational algebra expression can be used to define a view - thus views can also be bigger than the tables they are built from.

Example: As a first cut at designing a database for a library, we might be tempted to create a single scheme embodying everything we know:

\[
\text{books_out} = \\
(\text{call_number, title, author, date_due,} \\
\text{borrower_id, last_name, first_name, phone_number})
\]

However, such a scheme runs into all kinds of issues with anomalies:
a) How can we insert a borrower into the database until the borrower has a book out? How can we insert a book that is not checked out into the database? (insertion anomalies)

b) What happens to information about the borrower when the borrower returns the last book he/she has checked out? What happens to information about a book that is checked out when it is returned? (deletion anomalies)

c) What happens if a borrower moves and his/her phone number changes, but we forget to change all the rows in the table pertaining to this borrower? (update anomalies)

d) For this reason, the logical database design will likely partition the data into at least three relations - corresponding to an ER diagram structure with book and borrower entities plus a checked-out relationship between them. Nonetheless, while there are compelling reasons for not using such a table for the logical database model, we could create such a scheme (if we want it) as a view:

create view books_out as book |X| checked_out |X| borrower

4. Views can be used in queries just like the actual tables comprising the database - and can even be used in the definition of other views. However, numerous problems arise if we try to update the database through a view - hence such updates are generally either forbidden or heavily restricted.

Example: if we try to do an insert through a view that involves a projection, what do we insert into the columns that are projected out?

Example: if we try to do a delete through a view that involves a natural join, do we delete both of the tuples that joined to yield the tuple being deleted, or just one? If the latter, which one?

5. Views - especially those based on natural joins - can give rise to significant performance issues due to the time needed to compute the view when it is needed. For this reason, some DBMS’s allow the definition of materialized views. (In DB2 these are called summary tables.) A materialized view is stored in the database like any other table (and so does not need to be recomputed when needed); but the DBMS keeps it up to date when the underlying tables change, like any other view.
V. The Relational Calculus

A. The relational calculus is an alternative to the relational algebra as a basis for query languages. It is derived from predicate calculus.

1. A predicate is an assertion that we require to be true. When we formulate a query in the relational calculus, we specify a predicate that the object(s) we are looking for must satisfy.

2. Unlike relational algebra - which is procedural - relational calculus is non-procedural - i.e. we specify what requirements the result must satisfy, not how to compute it.

B. There are two variants of the relational calculus: the TUPLE RELATIONAL CALCULUS and the DOMAIN RELATIONAL CALCULUS.

1. In the tuple relational calculus, variables represent tuples, and predicates are formulated in terms of attributes of a tuple variable.

   Ex: Find book tuples for which author = dog:

   \[ \{ t \mid t \in book \land t[\text{author}] = \text{dog} \} \]

2. In the domain relational calculus, variables represent individual attributes, and complete tuples are represented as lists of attributes.

   Ex: The above query:

   \[ \{ \langle \text{call\_number}, \text{title}, \text{author} \rangle \mid \langle \text{call\_number}, \text{title}, \text{author} \rangle \in book \land \text{author} = \text{dog} \} \]

3. In either case, we represent a query by a predicate which we want the result to satisfy.

C. It is important to see that there are definite analogues between operations in relational algebra and predicates in relational calculus.

1. In fact, it can be shown that the systems are of equivalent power, in the sense that any query that can be formulated in the relational algebra (without the extensions we have discussed) can be formulated in either of the relational calculi, and any safe formula (we define this later) in either of the relational calculi is equivalent to some query in the relational algebra.
2. Thus, it is possible for one system to support query languages of both types by internally translating from a query of one type into the language used internally (most often a variant of relational algebra, since this is procedural in nature.)

D. We will now consider examples of relational calculus equivalents to each basic relational algebra operation.

1. Relational algebra SELECTION:

\[
\begin{align*}
\text{\textbackslash \textbackslash} & \text{ book} \\
\text{author} & = \text{ dog}
\end{align*}
\]

See the two examples above

2. Relational algebra PROJECTION:

\[
\pi \text{ borrower} \\
\text{last_name} \\
\text{first_name}
\]

a) tuple r.c.:

\[
\{ t \mid \text{\textbackslash \textbackslash} s \text{ \textbackslash \textbackslash} \text{ borrower} \land t[\text{last_name}] = s[\text{last_name}] \land t[\text{first_name}] = s[\text{first_name}] \}
\]

where t is on the new scheme (last_name, first_name)

b) domain r.c.:

\[
\{ \langle \text{last_name, first_name} \rangle \mid \text{\textbackslash \textbackslash} \text{ borrower_id} \\
\langle \text{borrower_id, last_name, first_name} \rangle \}
\]

3. Relational algebra NATURAL JOIN: checked_out \text{\textbackslash\textbackslash} borrower

a) i. tuple r.c.:

\[
\{ t \mid \text{\textbackslash \textbackslash} u, v \text{ \textbackslash \textbackslash} \text{ checked_out} \land v \text{ \textbackslash \textbackslash} \text{ borrower} \land
\]

\[
\begin{align*}
u[\text{borrower_id}] & = v[\text{borrower_id}] \\
t[\text{borrower_id}] & = u[\text{borrower_id}] \\
t[\text{call_number}] & = u[\text{call_number}] \\
t[\text{date_due}] & = u[\text{date_due}] \\
t[\text{last_name}] & = v[\text{last_name}] \\
t[\text{first_name}] & = v[\text{first_name}]
\end{align*}
\}
\]
where \( t \) is on the new scheme

\[(\text{borrower} \_\text{id}, \text{call} \_\text{number}, \text{date} \_\text{due}, \text{last} \_\text{name}, \text{first} \_\text{name})\]

b) domain r.c.:

\[
\{ \langle \text{borrower} \_\text{id}, \text{call} \_\text{number}, \text{date} \_\text{due}, \text{last} \_\text{name}, \text{first} \_\text{name} \rangle | \\
\langle \text{borrower} \_\text{id}, \text{call} \_\text{number}, \text{date} \_\text{due} \rangle \sqcap \text{checked} \_\text{out} \sqcap \\
\langle \text{borrower} \_\text{id}, \text{last} \_\text{name}, \text{first} \_\text{name} \rangle \sqcap \text{borrower} \}
\]

4. Relational algebra UNION: Suppose we have two tables that have the same scheme - say student\_borrower and fac\_staff\_borrower - and want a table including all persons in either group:

a) tuple r.c.:

\[
\{ t \mid (t \sqcap \text{student\_borrower}) \lor (t \sqcap \text{fac\_staff\_borrower}) \}
\]

b) domain r.c.:

\[
\{ \langle \text{borrower} \_\text{id}, \text{last} \_\text{name}, \text{first} \_\text{name} \rangle | \\
(\langle \text{borrower} \_\text{id}, \text{last} \_\text{name}, \text{first} \_\text{name} \rangle \sqcap \text{student\_borrower}) \lor \\
(\langle \text{borrower} \_\text{id}, \text{last} \_\text{name}, \text{first} \_\text{name} \rangle \sqcap \text{fac\_staff\_borrower}) \}
\]

5. Relational algebra DIFFERENCE: Suppose, instead, we have a general borrower table - which contains information on all borrowers - plus a student\_borrower table, which contains only student borrowers, and we want to list borrowers who are not students. In relational algebra, this would be borrower \(-\text{student\_borrower}

a) tuple r.c.:

\[
\{ t \mid (t \sqcap \text{borrower}) \land \neg (t \sqcap \text{student\_borrower}) \}
\]

b) domain r.c.:

\[
\{ \langle \text{borrower} \_\text{id}, \text{last} \_\text{name}, \text{first} \_\text{name} \rangle | \\
(\langle \text{borrower} \_\text{id}, \text{last} \_\text{name}, \text{first} \_\text{name} \rangle \sqcap \text{borrower}) \land \\
\neg (\langle \text{borrower} \_\text{id}, \text{last} \_\text{name}, \text{first} \_\text{name} \rangle \sqcap \text{student\_borrower}) \}
\]

6. A more complete example:

Find the last name of all borrowers who have an overdue book
\[ \pi \text{ checked_out } \exists \text{X} \text{ borrower} \]
\[
\text{last_name} \quad \text{date_due} < \\
\text{-- whatever today is --}
\]

Ask class to do in each relational calculus:

tuple:

\[
\{ t \mid \exists u,v \\
( u \text{ checked_out }^\land v \text{ borrower }^\land \\
t[\text{last_name}] = v[\text{last_name}]^\land \\
u[\text{borrower_id}] = v[\text{borrower_id}]^\land \\
u[\text{date_due}] < "\text{September 18, 2002"} ) \}
\]

domain:

\[
\{ <\text{last_name}> \mid \exists \text{borrower_id, call_number, date_due, first_name} \\
( <\text{borrower_id, call_number, date_due}> \text{ checked_out }^\land \\
<\text{borrower_id, last_name, first_name}> \text{ borrower }^\land \\
\text{date_due} < "\text{September 18, 2002"} ) \}
\]

E. Syntax of relational calculus (either type)

1. Relational calculus expressions have the general form

\[
\{ \text{tuple variable } \mid \text{formula} \}
\]
or

\[
\{ \text{<domain variables>} \mid \text{formula} \}
\]

2. Formulas are constructed out of building blocks known as atomic formulas. The following shows the various types of atomic formula permitted in each variant of the relational calculus:

tuple r.c.

\[
\text{tuple-var} \ [\text{Relation}}
\]

\[
=, \neq,
\]

\[
\text{tuple-var[some-attribute]} < \quad \text{tuple-var[some-attribute]} \\
> \quad \text{constant} \\
\leq, \geq
\]

34
domain r.c.

$$< \text{domain-vars} > \quad \Box \text{Relation}$$

$$= \quad \neq$$

domain-var $$< \quad$$ domain var

$$> \quad$$ constant

$$\leq \quad$$

$$\geq$$

3. In either variant, a formula is either an atomic formula, or is of one of the following forms:

- \( \neg \) formula
- (formula)
- formula \( \lor \) formula
- formula \( \land \) formula

\( \Box \ x \ (P(x)) \) - where P is a formula and x is a variable free in P

\( \exists \ x \ (P(x)) \) - where P is a formula and x is a variable free in P

note - a variable is said to be free in a formula if it is not the subject of one of the quantifiers \( \Box \) or \( \exists \) WITHIN the formula.

4. The free variables in the overall formula are the variables that become the ultimate objects of our query - i.e. the result of our query is the set of all values for the free variable(s) that satisfy the formula.

F. One important property of relational calculus formulas is SAFETY. A relational calculus formula is said to be safe iff it does not require us to inspect infinitely many objects.

1. Example: the query \( \{ t \mid (t R) \} \) is not safe. For any relation R, there are infinitely many tuples that are not members of that relation!

2. Unsafe formulas are most often the result of improper use of not. In general, the set of all values NOT satisfying some predicate is infinite; so when not is used it must be coupled with an additional condition that narrows the scope of consideration - e.g. if P(x) and Q(x) are safe formulas then

\( \neg \ Q(x) \)
is unsafe, but

\[ P(x) \land \neg Q(x) \]

is safe.

3. The most common way to ensure that a formula is safe is to include a formula that restricts consideration to tuples from a particular relation - i.e. a formula of the form \( t \in R \) or \( <a,b,c> \in R \).

4. Notice that the issue of safety is unique to the relational calculus. One cannot formulate an unsafe query in the relational algebra - hence only safe relational calculus formulas have relational algebra equivalents.