CS352 Lecture - The Relational Model

last revised July 25, 2008

Objectives:

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

Materials:

1. Projectable/handout of library example database used in class (scheme and instance)
2. Handout of natural language queries against this database
3. Projectable 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 the previous lecture.

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 both entities and relationships can easily be represented by tables. (We will look at this more formally when we talk about database design.)

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.

      Example: A person's address is not atomic. In the US, we normally consider it to consist of a street address, city, state, and ZIP code. Therefore, in setting up a relation scheme for a person, we will generally have separate attributes for each. (Sometimes even two attributes for street address, separating it into the street address proper and an apartment number or the like.)

   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.

      Example: Many people have more than one phone number.

D. We looked at relational databases briefly in CS211. Some of what we look at here will be review of concepts we saw then (just in case someone might have forgotten something - which would never happen, of course :-) but much of what we will discuss here will be new.

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:

0	false
1	false
2	false
3	false
4	false
...	
8	true
9	true

   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:

0	false
1	true
2	false
3	true
4	false
...	
8	false
9	true

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.

last_name	first_name
aardvark	anthony
cat	charlene

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. When we discuss the E-R model in more detail in conjunction with database design, we will see that there is a straight-forward way to to convert entities to tables, which will guarantee that any relation will have a primary key (though perhaps this primary key will, in fact, be all of the attributes together.)

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 " ").

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

   Have class turn to Figure 2.8 from book (doesn't scan well)

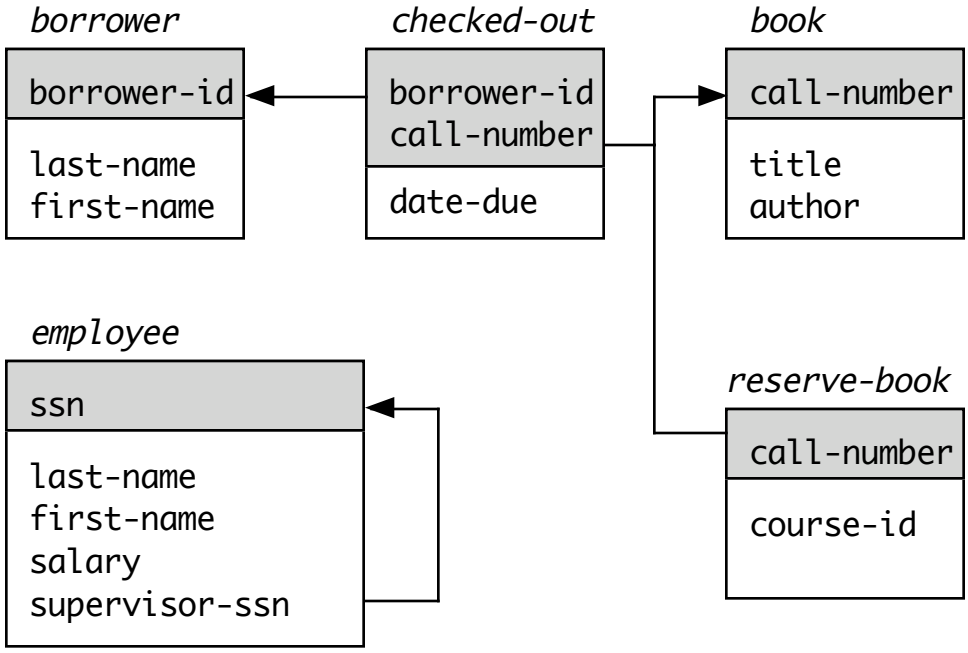
   Note the explicit representation of foreign keys

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. (This is similar to the one we used in CS211, but has two more tables and allows illustrating several more operations.)

   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).

Schema diagram:




```
Simplifying assumptions for this example:

1) author of a book is single-valued
2) there is only one copy of a book with a given call number
3) a given book can only be on reserve for a single course
4) course-id is presumably a foreign key in a table not shown
```

Example instance: (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

book(call_number, title, author)

QA76.093	Wenham Zoo Guide	elephant
RZ12.905	Fire Hydrants I Have Known	dog
LM925.04	21 Ways to Cook a Cat	dog
AB123.40	Karate	koala

checked_out(borrower_id, call_number, date_due)

89754	RZ12.905	11-10-02
89754	LM925.04	11-10-02
20147	AB123.40	11-15-02

reserve_book(call_number, course_id)

QA76.093	BY123
AB123.40	PE075

employee(ssn, last_name,first_name, salary, supervisor_ssn)

123-45-6789	aardvark	anthony	$40000	null
567-89-1234	buffalo	boris	$30000	123-45-6789
890-12-3456	elephant	emily	$50000	123-45-6789
111-11-1111	fox	frederick	$45000	567-89-1234

PROJECT, HANDOUT - SHOWING CONTENT AND SCHEME

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.

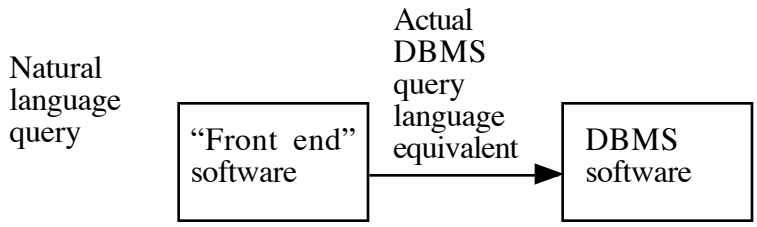
List the call numbers of all overdue books, together with the number of days they are overdue.

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 an area of 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.



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:

   Compute discriminant = 3^2 - 4 * 1 * 2
   Compute root1 = (-3 + sqrt(discriminant)) / 2
   Compute root2 = (-3 - sqrt(discriminant)) / 2

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 discusses two formal query languages, one here and one later:

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) The relational calculus -a non-procedural formal query language which is discussed in chapter 5.

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 - σ) 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

σ 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 - π) 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

   π 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

   π σ 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. Why? ASK

   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. That is, the projection operation can also result in squeezing rows out of the result.

c) 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 CARTESIAN PRODUCT (often called JOIN) 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:

```
borrower  call      date-due call     title              author
id        number              number

89754     RZ12.905 11-10-02 QA76.093 Wenham Zoo Guide   elephant
89754     RZ12.905 11-10-02 RZ12.905 Fire Hydrants ...  dog
89754     RZ12.905 11-10-02 LM925.04 21 Ways to Cook .. dog
89754     RZ12.905 11-10-02 AB123.40 Karate             koala
89754     LM925.04 11-10-02 QA76.093 Wenham Zoo Guide   elephant
89754     LM925.04 11-10-02 RZ12.905 Fire Hydrants ...  dog
89754     LM925.04 11-10-02 LM925.04 21 Ways to Cook .. dog
89754     LM925.04 11-10-02 AB123.40 Karate             koala
20147     AB123.40 11-15-02 QA76.093 Wenham Zoo Guide   elephant
20147     AB123.40 11-15-02 RZ12.905 Fire Hydrants ...  dog
20147     AB123.40 11-15-02 LM925.04 21 Ways to Cook .. dog
20147     AB123.40 11-15-02 AB123.40 Karate             koala
```

PROJECT

(1) Because the checked_out table contains 3 rows, and the book table 4 rows, the result of the join contains 3 x 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 reason, 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 x 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

σ (checked_out X book)
checked_out.call_number =
book.call_number

Yielding:

```
borrower  call      date_due call     title              author
id        number              number

89754     RZ12.905 11-10-02 RZ12.905 Fire Hydrants ...  dog
89754     LM925.04 11-10-02 LM925.04 21 Ways to Cook .. dog
20147     AB123.40 11-15-02 AB123.40 Karate             koala
```

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

π σ (checked_out X book)
title checked_out.call_number =
book.call_number

(Note 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

π σ (checked_out X borrower)
last_name, checked_out.borrower_id =
first_name borrower.borrower_id ^
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 - ρ.

   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

      ```
      π                σ                          (employee X ρ  employee)
      employee.        employee.supervisor_ssn =          supervisor
       last_name         supervisor.ssn ^
      employee.        employee.salary >
       first_name        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:

      ```
      π                σ                          (ρ employee X ρ employee)
       e.last_name      e.supervisor_ssn    =    e                 s
       e.first_name       s.ssn ^
                        e.salary > 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 <u>not</u> 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

(π borrower) ∪ (π employee)
last_name, last_name,
first_name first_name

What would be the result of this query?

ASK

aardvark	anthony
cat	charlene
dog	donna
fox	frederick
zebra	zelda
buffalo	boris
elephant	emily

(In some order - not necessarily this!)

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

b) Because relations are sets, duplicate rows that would result from a union are discarded - e.g., in the above, suppose an someone were <u>both</u> 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 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.

d) 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:

(ρ t1(id, last_name, first_name) π borrower_id, last_name, first_name) ∪ (ρ t2(id, last_name, first_name) π ssn, last_name, first_name employee)

(Note that we use ρ 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 the not second.

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

ASK

(π borrower) – (π employee)
last_name, first_name last_name, first_name

What would be the result of this query?

ASK

cat	charlene
dog	donna
zebra	zelda

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 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"?

   ASK

   $$(\pi_{\text{call\_number}} \text{ reserve\_book}) \cap (\pi_{\text{call\_number}} \text{ checked\_out})$$

   What would be the result of this query?

   ASK

   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 ∩

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

      to 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. (So often useful that the book seems to avoid using the term JOIN for the Cartesian Product - though it is, in fact, used that way in SQL. It is important to distinguish between JOIN and NATURAL JOIN).

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:

$\sigma_{\text{checked\_out.call\_number} = \text{book.call\_number}}$ (checked_out X book)

Yielding:

borrower id	call number	date_due	call number	title	author
89754	RZ12.905	11-10-02	RZ12.905	Fire Hydrants ...	dog
89754	LM925.04	11-10-02	LM925.04	21 Ways to Cook ..	dog
20147	AB123.40	11-15-02	AB123.40	Karate	koala

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 immediately follow the selection by projection.

Example:

π	σ	(checked_out X book)
checked_out.	checked_out.	
call_number	call_number =	
borrower_id	book.	
date_due	call_number	
title		
author		

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

checked-out |X| 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 if the user does a full cartesian product and then does a select and/or project on it. This will be true even if the joining columns have different names. For example, 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 |X| 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):

$$\pi_{\substack{e.last\_name \\ e.first\_name \\ s.last\_name \\ s.first\_name}} \left(\sigma_{e.supervisor\_ssn = s.ssn} \left(\rho_{e}\, employee \times \rho_{s}\, employee \right) \right)$$

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

$$\pi_{\substack{e.last\_name \\ e.first\_name \\ s.last\_name \\ s.first\_name}} \left(\rho_{e}\, employee \; \vartheta_{e.supervisor\_ssn = s.ssn} \; \rho_{s}\, employee \right)$$

   b) In the book, the notation used for theta join implies that it always involves a natural join - which is not necessarily the case.

4. 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_taken(student_id, course_id)

which lists ALL the courses a student has successfully completed,

and another relation

core_requirements(course_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_taken / core_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_taken / core_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

(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 $R \supset S$

$$r / s ::== (\pi_{R-S}(r)) - \pi_{R-S}(((\pi_{R-S}(r)) \times s) - r)$$

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

   ASK

   $\pi_{\text{call\_number},\ \text{today} - \text{date\_due}}(\sigma_{\text{date\_due} < \text{today}}(\text{checked\_out}))$

   What is the result of this query?

   ASK

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?

      $\mathcal{G}$ employee
      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

      $\mathcal{G}$ checked_out
      borrower_id count(call_number)

      What is the result of the above query?

      ASK

      89754 2
      20147 1

3. Another such extension is the OUTER JOIN, which is a variant of the natural or theta join.

   a) An ordinary natural 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:

      π book |X| checked_out
      title
      borrower_id

      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:

π book ⟕ checked_out
title
borrower_id

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 ⟖ - 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 ⟗ 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."

π σ (ρ employee X ρ employee)
e.last_name e.supervisor_ssn = e s
e.first_name s.ssn
s.last_name
s.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:

$$\pi_{\substack{e.last\_name \\ e.first\_name \\ s.last\_name \\ s.first\_name}} (\rho_{e}\,employee \overline{\underline{\bowtie}}_{e.supervisor\_ssn = s.ssn} \rho_{s}\,employee)$$

D. The book discusses how the various operations handle the special value NULL. 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.

1. 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!)

2. 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.)

3. 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

.

E. Modifying the Database

1. The relational algebra, as we have developed it, only deals with queries, but provides not way to alter the database. To add this capability, we need only add the assignment operator $\leftarrow$.

2. Deletion: general form is

relation ← relation - query expression for row(s)

Example: Delete all books written by dog

book ← book - σ (author = dog) book

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 ∪ (ρ p(borrower_id, last_name, first_name) patron)

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

or

relation ← changed tuples ∪ (relation - tuples to be changed)

Example: Give all employees a 5% pay raise

employee ← π (last_name, first_name, salary * 1.05, supervisor_ssn) employee