CS 325 - DB Reading Packet 3: "Introduction to the Relational Model and Relational Operations"

Sources:

- Kroenke, "Database Processing: Fundamentals, Design, and Implementation", 7th edition, Chapter 1, Prentice Hall, 1999.
- Connolly and Begg, "Database Systems: A Practical Approach to Design Implementation and Management", 3rd Edition, Addison-Wesley.
- Ricardo, "Databases Illuminated", Jones and Bartlett.
- Sunderraman, "Oracle 9i Programming: A Primer", Addison-Wesley.
- Ullman, "Principles of Database Systems", 2nd Edition, Computer Science Press.

Introduction to the Relational Model and Relational Operations

As we have already mentioned, the **relational model** was first developed by **E.F. Codd**, at IBM, in 1970, based on a branch of mathematics. And, as we also mentioned, while it was considered a breakthrough in terms of theory form the start, it was also considered to be impractical during its early years: a DBMS implementing a relational database would require too much computer overhead, and could not possibly be fast or efficient enough to "ever" be practical. But, fortunately, hardware and OS efficiency did improve enough, and hardware and memory became cheap enough, for that overhead to become affordable after all.

And why is there so much overhead? Because a relational DBMS, or RDBMS, is abstracting out a LOT of the physical details by providing the very powerful, elegantly simple **relational** view of one's data.

There are several reasons, then, that the relational model is important, besides its elegance:

- the relational model can be used to express DBMS-independent database designs, since the constructs of the relational model are so broad and general;
- the relational model is the basis for a whole important category of DBMS products!

The Relational Model

In the relational model, a **relational database** is a collection of **relations**.

And, what is a **relation**?

Formally, it is "a subset of the Cartesian product of a list of domains" (Ullman, p. 19). You might recall that a Cartesian product of two sets is a set containing every (ordered) pairing of one thing from the first set with one thing from the second set -- that is, the Cartesian product of the sets {A, B, C} and {1, 2} is the set { (A, 1), (A, 2), (B, 1), (B, 2), (C, 1), (C, 2) }.

But -- how does this lead to relations? Let's expand on this relation definition a bit.

In Sunderraman, a relation scheme/schema is defined as a finite sequence of unique **attribute** names -- for example,

```
employees = (empl_id, empl_name, empl_addr, empl_salary)
```

Each attribute name `A` is associated with a **domain**, `dom(A)`, a set of values, which includes a special value `null`. (Be careful - `null` is special! It means the **lack** of any value, rather than any particular value.) `dom(A)`, then, is the set of values attribute `A` can possibly have -- for example, `dom(empl_id)` might be the set of integers between 1000 and 9999 inclusive, while `dom(empl_salary)` might be the set of real numbers between 0 and 100000, inclusive, along with the special value null.

Given a **relation scheme/schema** $R = (A_1, A_2, \ldots A_n)$, then, a **relation** `r` on the relation scheme/schema `R` is defined as any finite subset of the Cartesian product

$dom(A_1)$ x $dom(A_2)$ x ... x $dom(A_n)$

So, for the `employees` scheme/schema given above, a relation `r` on that schema `employees` would be **any** subset of combinations of an `empl_id`, an `empl_name`, an `empl_addr`, and an `empl_salary`. An example relation under this relational schema could be something like:

```
{ (1111, 'Jones', '111 Humbold Lane', 20000),
  (2222, 'Smith', '123 Lumberjack Ave', 25000) }
```

See how that looks, well, rather tabular?

But, notice that a relation, in the relational model, is really a **set**, and a set of what? For an employees relation, it is a set of those combinations of an `empl_id`, an `empl_name`, an `empl_addr`, and an `empl_salary`. That is, the example relation given above has **two** elements: the element `(1111, 'Jones', '111 Humbold Lane', 20000)` and the element `(2222, 'Smith', '123 Lumberjack Ave', 25000)`. In the relational model, each of the elements in a relation is also referred to as a **tuple** (which, when pronounced, rhymes with "couple").

So -- the above was a discussion of the formal definition of a **relation**. What, then, is the formal definition of a **relational database scheme or schema**? It is a finite set of relation schemes/schemas $\{R_1, R_2, \ldots R_m\}$. Or, in our parlance from the last lecture, it is a collection of **relation structures**!

And, what is a **relational database** on a relational database scheme/schema `D`? It is a set of relations $\{r_1, r_2, \ldots r_m\}$ where each r_i is a relation on the corresponding relation scheme/schema R_i.

Those, then, are the formal definitions of a relation scheme/schema, a relation, a relational database scheme/schema, and a relational database!

Different Terms for the Same Thing

Because relational databases "grew" out of work in quite separate areas, you'll find different terminology for the same things that still persists in different writings about databases. You should be comfortable with all of these terms, so you'll know what is meant whichever is used.

Above, you've seen the terms **relation**, **tuple**, and **attribute**. Those are indeed the terms that mathematicians would use for these major ideas within the relational model. But those with a more "mainframe" background might see parallels between those terms and **file, record,** and **field** (but they would always mean a record-based file, not a stream-based file!). And most others would more recognize these as **table**, **row**, and **column**.

We will deliberately avoid the term **file** for referring to a table or relation (the whole point of a DBMS is to hide file details from the user!). But when we say **table**, we do mean a collection of rows and columns that meets the definition of a relation as we've just discussed. You'll frequently see **row** and **tuple** used interchangeably for one of the sets within such a table or relation -- and you'll even occasionally see **record**. And you'll frequently see **column**, **attribute**, and even **field** used interchangeably for, well, a column within a table or relation.

So, you should be comfortable with what is meant, regardless of which of these terms we are using.

Natural Restrictions on Tables Arising from the Definition of a Relation

Notice that, if you follow the line of the relational model definitions, then certain things must ALWAYS be true about the resulting relations, the resulting tabular beasties:

- Assume that a **cell** of a relation/table is the intersection of a row and a column -- that is, a cell is a single attribute/column value within a tuple/row. There can only be **one** "value" per "cell"; that is, the cells of the relation must be **single-valued**. There can be **no** multi-valued cells!
    - (Consider those definitions we just discussed -- if you are building subsets of Cartesian products of domains, there is no provision for including more than one value of an attribute's domain within one of the elements, one of the tuples, of the resulting set!)
- Each column in the table - each attribute in the relation - should be a characteristic with a particular domain of possible "valid" values (including both physical and semantic definitions). That is, a given column cannot contain values from multiple domains -- you cannot have values of different types within the same column! All of the entries in any column/attribute/field must be of the same "kind" -- and note that a domain often includes the special (lack of a) value `null`.
    - aside: the **physical** definition of a domain might be the data type you use to represent the values in that domain; the **semantic** definition is a more logical description of the allowed values, what they "mean".
    - So, while a physical definition of `dom(salary)` might be a floating point type, the semantic definition would be that it includes values that can be employee salaries, the amount an employee makes in, say, a year. One would not then expect a negative value to be part of the semantic definition of `dom(salary)`, even if DBMS limitations lead you to provide a physical definition of `dom(salary)` that permits such values. You might need to limit the possible values of that column in some other fashion. So, all of the values in a column are required to have the same "meaning", the same domain.
- **No duplicate rows** are allowed! Consider: relations are sets, and sets, by definition, don't have duplicate members -- an element either is a set member, or it isn't. And since a relation is sets of tuples/rows/records, it then follows that a relation has NO duplicate tuples/rows/records.
- And, as the order of elements in a set is not important -- `{A, B, C}` and `{C, B, A}` and `{B, A, C}` are all considered depictions of the same set -- so the order of rows/tuples/records in a table/relation is also not considered important.
    - That said, sometimes people would like a depiction of a relation ordered in a certain way -- and so SQL will provide a means for requesting such a depiction, as we'll see in a later

reading packet. But, in terms of the mathematical concept of a relation, the order of rows/tuples/records is not significant (and in terms of the abstraction provided by a relational DBMS, you should not care in what order a relation's rows/tuples/records happen to be stored).

- Each column/attribute/field must have a unique name (within a table/relation/record).
  - This is not saying that you cannot have the same column name in two different tables -- it is saying that, within a single table, you cannot have two columns, two attributes, with the same name.
  - If you then consider a column's/attribute's/field's "full" name to be `<relation_name>.<attribute_name>`, then, since you cannot have two relations with the same name within a database, then even within a database each column/attribute/field will end up also having a unique name.
- But, again as in sets, the order of the attributes/columns/fields does not matter, either.

So, note that each of the following are depictions of the **same** relation, a `dept` relation, whose relation structure form can be given as:

```
dept(dept_name, DEPT_NUM, dept_loc)
```

dept_name	dept_num	dept_loc
Accounting	10	New York
Research	20	Dallas
Sales	30	Chicago
Operations	40	Boston

dept_num	dept_name	dept_loc
40	Operations	Boston
30	Sales	Chicago
20	Research	Dallas
10	Accounting	New York

This particular relation has 4 tuples/rows/records, made up of 3 attributes/columns/fields each.

A Few More Words on Relations and Relational Schemas

Now, when you are actually creating your own relations, you very likely will not store random tuples, random combinations of values from the attribute domains -- rather, you are likely to choose values from the domains of each attribute that pertain to some actual "thing", or portion of an actual "thing" -- for example, the `empl_id`, `empl_name`, `empl_addr`, and `empl_salary` for an actual employee in your "world"! So, a tuple represents a **relationship** among a set of values.

And, if you give the **name** of a relation/table, followed by, in parentheses, the (unique) names of each column/attribute/field (capitalizing or otherwise indicating the primary key attributes), that is what we call **relation structure** form.

Strictly speaking, relation structures **plus** constraints on allowable data values is a **relational database schema**. We'll be giving a less-mathematical definition later on, but we'll find these definitions really don't conflict too badly -- the less-mathematical definition will just state some of these constraints on allowable data values more specifically.

Functional Dependencies

Both for this packet's purposes and for the purposes of explaining normalization in a future packet, it is useful to discuss the concept of **primary key** a bit more formally. We'll start with the concept of a **functional dependency**.

A **functional dependency** is a relationship between or among attributes. If, given the value of one attribute, `A`, you can look up/uniquely obtain the value of another, `B`, then we say that `B` is **functionally dependent** on `A`.

That is, consider an `employees` relation:

```
employees(EMPL_ID, empl_name, empl_addr, empl_salary)
```

If you are given the value of a particular `empl_id`, you can look up a single `empl_salary` that corresponds to that `empl_id` value. However, multiple employees might have the same salary. You should not reasonably expect to always obtain a single `empl_id` given a particular salary.

(Note that, in determining functional dependencies, it is more important what is reasonable given the **meanings** and possible domains of the attributes within one's "world" rather than the particular rows that may be in a relation at some particular time. If there is ever a period of time in which all of the employee salaries happen to be different, that would **not** suddenly allow us to say that `empl_id` is functionally dependent on `empl_salary`.)

So, we say that attribute `B` is **functionally dependent** on attribute `A` if the value of `A` **determines** the value of `B` -- if, knowing the value of `A`, we can **uniquely determine** the value of `B`.

Some have argued that the storage and retrieval of functional dependencies is a major reason for having a database!

Functional dependencies are written using the following notation:

```
A -> B
```

```
empl_id -> empl_salary
```

read as:

- `A` **functionally determines** B; `empl_id` **functionally determines** `empl_salary`
- `A` **determines** `B`; `empl_id` **determines** `empl_salary`
- `B` is **functionally dependent** on `A`; `empl_salary` is **functionally dependent** on `empl_id`

The attribute on the left-hand-side of the arrow in this notation is called the **determinant** -- `A` and `empl_id` are the determinants in these two functional dependencies.

Note that functional dependencies might involve **groups** of attributes -- consider the relation:

```
Course_grades(STUD_ID, SECTION_NUM, Grade)
```

A student may take several different course sections -- you cannot say that `Stud_id -> Grade`. And, likewise, a section contains many students -- you cannot say that `Section_num -> Grade`. But if the assumption is that a student can only be registered in a given section of a course once, it is quite reasonable to say that:

```
(Stud_id, Section_num) -> Grade
```

That is, the combination/pair of a `stud_id` and a `section_num` does uniquely determine a `Grade`.

Here is a small self-check: make sure that these make sense to you:

- If `X -> (Y, Z)`, then it is also true that `X -> Y` and `X -> Z`.
- However, if `(X, Y) -> Z`, then you **cannot** reasonably assume that `X -> Z` or that `Y -> Z`.

Keys

We'll now discuss a series of important **key** definitions.

A **superkey** is a set of one or more attributes that **uniquely identifies** a tuple. Again, this is based on what is reasonable given the "world", what is true in the day-to-day operations of that "world" (sometimes called its **business rules**), and the domains of the data, not whatever might happen to be the contents of particular rows at one time.

Since a relation, by definition, cannot contain duplicate rows, then every relation has the superkey consisting of all of its attributes (an entire row uniquely determines itself!) But sometimes subsets of attributes may be a superkey, also. Since `empl_id` uniquely identifies an `Employees` tuple -- no two employees are given the same `empl_id` -- then the set consisting of just `empl_id` is a superkey for the `employees` relation, also.

Consider the `Course_grades` relation -- the pair `(Stud_id, Sect_num)` uniquely determines a row, as does the trio `(Stud_id, Sect_num, Grade)` -- so, both of those are superkeys of the `Course_grades` relation. However, the set of just `Course_id` is not a superkey for `Course_grades` -- many students take a course, so many rows may contain a given `Sect_num`. Likewise, the set of just `Stud_id` is not a superkey for `Course_grades`, either -- students often take more than one course, so several rows may contain a given `Stud_id`.

One more time: it is an important point that whether an attribute, or set of attributes, is a superkey (and whether a functional dependency exists between attributes) is determined by the user, really -- by their model, their semantics, their business rules. If you had a school where students were allowed to take one course, ever, then in that odd world `Stud_id` might indeed be a superkey for `Course_grades`. When in doubt, it is not a good idea to just look at a small sample of data, or to guess -- when in doubt, ask the user(s)!

So, most relations (although not all) have more than one superkey.

If a superkey is **minimal** -- if no proper subset of its attributes is also a superkey -- then such **minimal superkeys** are also called **candidate keys**. For the relation `Course_grades`, the trio `(Stud_id, Sect_num, Grade)` is a superkey, but not a candidate key, since it is not minimal: its subset `(Stud_id, Sect_num)` is also a superkey. The pair `(Stud_id,` Sect_num) is a candidate key, though, since it is a minimal superkey: neither Stud_id nor Sect_num is a superkey.

(Remember that whether a superkey is minimal does **not** have to do just with how many attributes it includes -- a two-attribute or three-attribute superkey can indeed be minimal, also. All that matters is that **no** subset of attributes within that collection of attributes is also minimal.)

The **primary key**, then, can be defined as the candidate key that you choose for a particular relation. (Get it? The candidate keys are the candidates for primary key!) Or, you should select a primary key from amongst that relation's candidate keys -- the primary key is expected to be selected from amongst those candidate keys. Some tables may indeed have multiple candidate keys -- but the designer only selects one of those to be primary key. (Remember, a candidate key may still be a collection of attributes! But no subset of those attributes is also a superkey.)

Once you choose a relation's primary key, and define it as the primary key in that relation's definition, then a DBMS that supports **entity integrity** will enforce that primary key. That is, such a DBMS will not permit the insertion of a row with the same primary key as an existing row -- it will thus ensure that the primary key really does uniquely determine a row within that table. As another part of enforcing entity integrity, such a DBMS will also not permit any attribute that is part of the primary key to contain the special null lack-of-a-value.

Just because the database field has such a hodge-podge of terminology -- remember those **physical indexes** we discussed in an earlier packet, that some DBMSs support as part of a database to allow you to specify that certain attributes should be set up to be more-quickly searchable? Sometimes these physical indexes are also called **physical keys**! But as these do not include the idea of uniquely determining a row, on those rare occasions that we refer to these, we'll stick with either physical index or index for these.

What, (formally), then, is a **foreign key**? It is an attribute or set of attributes within one relation that is at least a candidate key of another relation -- although, in practice, many DBMSs (including Oracle) require it to be the primary key of some other relation. So, in practice, when the primary key of one relation is stored in a second relation, that is called a foreign key. The relation that a foreign key references -- that the foreign key is "from", if you will -- is often called the "parent" relation, and the relation with a foreign key is often called the "child" relation.

The importance of foreign keys is that they enable us to **link** tables by using controlled redundancy -- the primary key of one table appears as the foreign key in the table to which it is linked. In a future reading packet, we will discuss how and when to insert foreign keys as part of the conversion from a database model to a database design/schema, because there are very specific rules one follows in putting them in, based on the model.

So, let's say that you have a parts relation, with characteristics of parts that some store sells, and its primary key is parts_num. There might also be an orders relation, representing orders of parts. If each order can only be for one part, then you might decide to add the parts' relation's primary key to the orders relation as a foreign key -- you might make parts_num an attribute in the orders relation, and also define it as a foreign key referencing parts. So, since each orders tuple includes the attribute part_num, that relates each order to the parts tuple with more information about the part being ordered.

Consider -- should it be reasonable, then, for an orders tuple to contain a part_num that is not in the parts relation? It should not be, and it could be useful for a DBMS to prevent this. How might it prevent this? It could prevent the adding of an orders tuple with a part_num that is not in the parts relation -- it could also prevent the removing of a parts tuple if there are orders of that part, and it could prevent modifying an `orders` tuple's `part_num` to a `part_num` not in the `parts` relation.

If you can define a foreign key in a set of relations, and the DBMS enforces that -- that is, if the DBMS does not allow a value for a foreign key that is not already in the "parent" relation for that key, and it does not allow a "parent" tuple to be deleted if any "child" tuples have that tuple's primary key as its foreign key -- then that DBMS is said to be enforcing **referential integrity**. Or, to put this another way: referential integrity means that if a foreign key contains a value (if it is not null), then that value must refer to an existing tuple in another relation.

Relational Operations

There are operations in **relational algebra** that turn out to be very useful for querying data in relational databases. What is relational algebra? It is algebra in which the variables represent relations (in contrast to, say, numeric algebra, in which the variables represent numbers). Just like you can talk about operations on numbers in numeric algebra -- and how operations such as addition on numbers and multiplication on numbers result in numbers -- in relational algebra, relational operations on relations result in relations. That is, where the arithmetic operators in numeric algebra manipulate numbers, in relational algebra the relational operators manipulate relations to form new relations (which can in turn be operated on by other relational operators). You can build compound relational expressions as you can build compound arithmetic expressions!

In practice, then, this is the basis for ad-hoc queries -- with relational operators, the only limit you have to what you can ask about a well-designed set of relations is your imagination (and your relational operations skill)!

Some of these operators are so-called **set-theoretic** operators -- they derive from set theory, but here the sets are sets of tuples (since, remember, that is what tuples are!). These include **union**, **difference**, **intersection**, and **Cartesian product**.

The **relation-theoretic** operations are particular to relational algebra: **rename**, **selection**, **projection**, **join**s of various types, and **division**.

Any relational DBMS worthy of the name had **better** support at least the operations of selection, projection, and the particular joins of natural and/or equi-joins!! So, in this packet, we will concentrate on those three (and also Cartesian product along the way, as you have to understand it to understand natural joins and equi-joins).

The selection relational operation

Be careful not to confuse the selection relational operation with the (unfortunately named) `select` statement in SQL! They are **not** synonymous, as we will discuss later on.

The selection operation selects **just the specified rows** from a table (just the specified tuples from a relation). That is, given a select operator and a relation and some criterion, the result is the relation consisting of just those rows from the given relation that meet that criterion. (Note that, if a relation is a subset of the Cartesian product of a list of domains, then a further subset of that subset is still a relation!)

(You could think of selection as being a kind of "horizontal" filter, as taking a "horizontal" subset of a relation.)

For example, say that you have a `Student` relation as follows:

`Student:`

Stud_ID	Stud_Name	Stud_Major	Stud_Grade_Level	Stud_Age
123	Jones	History	JR	21
158	Parks	Math	GR	26
105	Anderson	Management	SR	27
271	Smith	History	JR	19

Then the result of the selection operation on the `Student` relation of rows in which `Age < 25` would be:

`Student WHERE Age < 25`

Stud_ID	Stud_Name	Stud_Major	Stud_Grade_Level	Stud_Age
123	Jones	History	JR	21
271	Smith	History	JR	19

The projection relational operation

The projection operation grabs **just the specified columns/attributes** from a relation, and eliminates any duplicate rows in what results, so that the result will still be a relation.

(You could think of projection as being a kind of "vertical" filter, as taking a "vertical" subset of a relation.)

Or, it grabs specified **columns** from a table to create a new table, eliminating any duplicate rows before the final result.

For example, say that you still have a `Student` relation as follows:

`Student:`

Stud_ID	Stud_Name	Stud_Major	Stud_Grade_Level	Stud_Age
123	Jones	History	JR	21
158	Parks	Math	GR	26
105	Anderson	Management	SR	27
271	Smith	History	JR	19

Then the result of the projection operation of the `Student_Major` and `Stud_grade_level` attributes of the `Student` table would be:

`Student[Student_Major, Stud_Grade_level]`

Stud_Major	Stud_Grade_Level
History	JR
Math	GR
Management	SR

Do you see how the original `Student` table had 4 rows, but this projection only has 3? That's because a duplicate row was indeed eliminated from this projection's result (there were two rows with `(History, JR)`).

The Cartesian product operation

Now, because of their importance, I would prefer show equi-join and natural join next. However, one needs Cartesian product to explain what equi-join and natural join mean, and so it must be discussed before discussing equi-join and natural join.

The Cartesian product operation on two relations concatenates every tuple/row from one relation to every tuple/row from another relation, forming a third relation. That is, it produces a relation consisting of **all possible pairs of rows** from 2 relations. Notice that the Cartesian product of relation `A`, with `m` rows, and relation `B`, with `n` rows, will be a relation containing `m*n` rows!

For example, say that you have a `Price` relation as follows:

`Price:`

Prod_Code	Price
AA	5.99
BB	22.75

And, say that you have a `Location` relation as follows:

`Location:`

Store	Aisle	Shelf
23	W	5
24	K	9
25	Z	6

Then the Cartesian product of `Price` and `Location` would be:

`Price x Location`

Prod_Code	Price	Store	Aisle	Shelf
AA	5.99	23	W	5
AA	5.99	24	K	9
AA	5.99	25	Z	6
BB	22.75	23	W	5
BB	22.75	24	K	9
BB	22.75	25	Z	6

The equi-join and natural join operations

The join operations in general create a relation that is the Cartesian product of two relations with certain tuples removed. We are going to concentrate today on the most important join operations, equi-join and natural join (but note that there are others as well!).

In an equi-join or a natural join, you have a **join condition** that is an **equality** -- a condition specifying that a column or columns in one table have the same value as the column or columns in another table. You then request a join operation on those two tables based on that equality.

The first step of any join (at least conceptually) is to perform a Cartesian product of the two tables being joined.

Then, for an equi-join, you perform the second step: a relational selection of the result, selecting only those rows for which the join condition is true. This result is called the equi-join.

But, if you have selected rows for which an equality-based join condition is true, can you see that two columns will have the same contents? (The columns have different names -- `table1.attrib`, `table2.attrib` -- but the same contents.) So, a natural join performs a third step, a further projection operation, projecting all of the columns of the result except for one of the "duplicate-contents" columns.

For example, say that you have `Student` and `Enrollment` relations as follows:

`Student:`

Stud_ID	Stud_Name	Stud_Major	Stud_Grade_Level	Stud_Age
123	Jones	History	JR	21
158	Parks	Math	GR	26
105	Anderson	Management	SR	27
271	Smith	History	JR	19

`Enrollment:`

Stud_ID	Class_name	Position_Num
123	H350	1
105	BS490	3
123	BA490	7

Say that you wish to compute the equi-join and natural join of these tables based on the join condition `(Student.Stud_id = Enrollment.Stud_id).`

First, compute the Cartesian product of these two tables:

`Student x Enrollment`

Student. Stud_id	Stud_Name	Stud_Major	Stud_Grade _Level	Stud_Age	Enrollment. Stud_ID	Class_Name	Position_ Num
123	Jones	History	JR	21	123	H350	1
123	Jones	History	JR	21	105	BS490	3
123	Jones	History	JR	21	123	BA490	7
158	Parks	Math	GR	26	123	H350	1
158	Parks	Math	GR	26	105	BS490	3
158	Parks	Math	GR	26	123	BA490	7
105	Anderson	Management	SR	27	123	H350	1
105	Anderson	Management	SR	27	105	BS490	3

Student. Stud_id	**Stud_Name**	**Stud_Major**	**Stud_Grade _Level**	**Stud_Age**	**Enrollment. Stud_ID**	**Class_Name**	**Position_ Num**
105	Anderson	Management	SR	27	123	BA490	7
271	Smith	History	JR	19	123	H350	1
271	Smith	History	JR	19	105	BS490	3
271	Smith	History	JR	19	123	BA490	7

Second, perform a selection on this result of only those rows for which `(Student.stud_id = Enrollment.stud_id)`; the resulting relation will be:

Student. Stud_id	**Stud_Name**	**Stud_Major**	**Stud_Grade _Level**	**Stud_Age**	**Enrollment. Stud_ID**	**Class_Name**	**Position_ Num**
123	Jones	History	JR	21	123	H350	1
123	Jones	History	JR	21	123	BA490	7
105	Anderson	Management	SR	27	105	BS490	3

This is the equi-join of these two tables on the join condition `(Student.stud_id = Enrollment.stud_id)`. (Do you see how, if the join condition involves columns with the same domain, the resulting equi-join essentially combines related information into a single relation? Above, each row contains a student's information combined with the information about one of that student's course enrollments.)

Do you also see how the columns `Student.Stud_id` and `Enrollment.Stud_id` have exactly the same contents? (And must, since we selected the rows based on that very equality!)

Then, the natural join of these two tables on this join condition would include the third step of now projecting all of the columns in this result except for one of the "duplicate-contents" columns (and it doesn't matter which one of the two is omitted). So, for example, the resulting natural join in this case could be:

Student. Stud_id	**Stud_Name**	**Stud_Major**	**Stud_Grade _Level**	**Stud_Age**	**Class_Name**	**Position_ Num**
123	Jones	History	JR	21	H350	1
123	Jones	History	JR	21	BA490	7
105	Anderson	Management	SR	27	BS490	3

So, either an equi-join or a natural join basically "combines" relations based on common attributes. Note that if you use a join condition on columns whose domains do not have the same meaning, the result likely will be meaningless!

It has been said that equi-join/natural join are the source of the power of relational databases, allowing the use of **independent** relations linked by **common** attributes (really, by carefully-chosen foreign keys!). In good practice, join conditions usually involve comparing primary or foreign keys in one table to their corresponding foreign keys in another table.

Also note: doesn't the above sound horribly inefficient? If so, note that above is **conceptually** what equi-join and natural join **mean** -- the actual algorithm used by a DBMS to perform such joins will be different than that described here, and much more efficient, although the results will be the same.

And that is where we will conclude for this packet. (We will discuss a few of the set-theoretic operations on relations in a later packet.)