DDL, Database Constraints, & Basics Of SQL Queries

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. DDL: CREATE TABLE & Domains

2. CREATE TABLE & Defining Constraints

3. SQL: A Relational Query Language

Some material introducing SQL queries derived from lectures created by Professor Zvi Kedem, NYU Courant: used with permission

Context

- We've discussed the relational data model
- Then took a detour to discuss core Java concurrency concepts
  - Apologies: but (probably?) necessary to get you started with the course project
- Today
  - Return (briefly) to the topic of relational database constraints
  - Discuss these constraints in the context of creating database table
  - Other database table commands (DDL)
  - Row manipulation commands (DML)
  - Begin discussion of SQL queries (part of DML)

DDL: CREATE TABLE & Domains

Constraints: A Quick Review

- We discussed various types of constraints that can be enforced by a relational database
  - Domain constraints
  - Key constraints
  - Referential integrity
  - Entity integrity
- We didn't really discuss how these constraints are defined
- To do so, we have to pause before diving into querying existing tables
  - And look at how tables are created, deleted, and modified
  - By "modify", I refer to altering a table's structure, in contrast to altering a table's existing tuples
- Relational algebra has nothing to do with such operations
- They were introduced into SQL when relational databases moved from "research laboratory" to "commercial use" ☺

SQL History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory in the 1970s
  - Structured English QUEry Language
- Renamed Structured Query Language (SQL)
- Standardization efforts under ANSI and ISO
  - SQL-86, SQL-89, SQL-92, SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
  - The standardization process has not stood still!
- Commercial systems offer most, if not all, SQL-92 features
  - Plus varying feature sets from later standards and special proprietary features
  - Warning: Not all examples here may work on your particular system
  - That's what "manuals" are for ... ☺

I've made an effort to refer to language constructs that are supported by PostgreSQL

- DDL (Data Definition Language) statements are used to build and modify the structure of your tables
  - Also: to modify the structure of other database objects
- Reminder: this has nothing to do with relational algebra!
- SQL DDL allows you to define DBMS information about relations ("tables") such as:
  - The schema of each relation
  - The domain of values associated with each attribute
  - Integrity constraints
  - Other important DBMS constructs (to be discussed in depth, later) ...
    - The set of indices to be maintained for each relation
    - Security and authorization information for each relation
    - The physical storage structure of each relation on disk

- First thing you should learn when working with any new DBMS is its CRUD syntax
  - Create
  - Retrieve (or “Read”)
  - Update
  - Delete
- For relational databases:
  - DDL can be thought of as “CRUD for tables”
  - DML (“Data Manipulation Language”) is “CRUD for rows”
- Let’s do the “easy” CRUD DDL operations first …
  - Delete & Update

Deleting A Table

```
1  CREATE TABLE author (
2          author_id INT NOT NULL PRIMARY KEY, firstname VARCHAR (50), lastname VARCHAR
               (50)
3  );
4
5  CREATE TABLE page (
6          page_id serial PRIMARY KEY, title VARCHAR (255) NOT NULL, CONTENT TEXT,
7          author_id INT NOT NULL,
8          FOREIGN KEY (author_id) REFERENCES author (author_id)
9  );
```

- (Apologies, but I must precede the "delete table" example with a "create table" example: we'll discuss "create table" in detail later)
- Q: what happens when you issue DROP TABLE AUTHOR?
- A: because "page" table depends on "author" table, POSTGRESQL issues a *"cannot drop table author because other objects depend on it"* error message ☹

```
1  DROP TABLE [IF EXISTS] table_name [CASCADE | RESTRICT]
```

- Use the CASCADE (opposite of the RESTRICT default) to change this behavior
- Use the IF EXISTS sub-clause to avoid error messages such as *"table "author" does not exist.*

Updating (“Altering”) A Table

- You can change the structure of an existing database table through the ALTER command
- This is pretty boring ☹
  - Pretty much everything that you can do when creating a table (hold that thought) can be altered if you change your mind
  - Read the manual for details
- Examples:
  - Add a column, drop a column, rename a column, or change a column’s data type.
  - Set a default value for the column.
  - Add a CHECK constraint to a column
  - Rename a table

```
1  ALTER TABLE links ADD COLUMN active boolean
2  ALTER TABLE links RENAME COLUMN title TO link_title;
3  ALTER TABLE links RENAME TO url;
```

Example of CREATE TABLE Command

```
create table instructor
    (ID char(5), name varchar(20),
     dept_name varchar(20), salary numeric(8,2))
```

In general, an SQL relation is defined using the CREATE TABLE command:

create table *r*

(A_1 D_1, A_2 D_2, ..., A_n D_n,
(integrity-constraint$_1$),
...,
(integrity-constraint$_k$))

- *r* is the name of the relation
- Each A_i is an attribute name in the schema of relation *r*
- Each D_i is the data type of values in the domain of attribute A_i

What Are These Domains?

Remember: in the bad old days, wasted bits cost your company money!

- char(n): Fixed length character string, with user-specified length *n*
- varchar(n): Variable length character strings, with user-specified maximum length *n*
- int: Integer (a finite subset of the integers that is machine-dependent)
- smallint: Small integer (a machine-dependent subset of the integer domain type)
- numeric(p,d): Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point.
- real, double precision: Floating point and double-precision floating point numbers, with machine-dependent precision
- float(n) Floating point number, with user-specified precision of at least *n* digits

Example: $numeric(3, 1)$, allows 44.5 to be stored exactly, but not 444.5 or 0.32

Dates & Time (I)

- The “domains” on the previous slides should feel familiar from a programming language perspective
  - Can be viewed as “space saving” refinements of Java primitives
  - Keeping in mind: “the more restrictive the type, the more likely you are to cause a domain violation” ☹
- Now we look at “data & time” related types
  - Compare to `java.util.Date` & `java.time` packages
- date: these are dates, containing a (4 digit) year, month and date
  - `date '2005-7-27'`

Dates & Time (II)

- time: these are “time of day”, in hours, minutes and seconds
  - `time '09:00:30'` or `time '09:00:30.75'`
- timestamp: these are “date plus time of day” constructs
  - `timestamp '2005-7-27 09:00:30.75'`
- interval: these represent a “period of time”
  - `interval '1' day`
  - When subtracting a date/time/timestamp value from another, you get an interval value
  - Similarly: can add an interval value to date/time/timestamp values

```
1  SELECT
2          now(),
3          now() - INTERVAL '1 year 3 hours 20 minutes'
4               AS "3 hours 20 minutes ago of last year";
```

Large-Object Types

- "Large object" types have two purposes
  - An "escape" hatch for state that doesn't have a pre-existing type in the DBMS
  - A technique for handling large strings efficiently
- Example: photos, videos, CAD files
- blob (binary large object): object is a large collection of uninterpreted binary data
  - Interpretation is left to an application outside of the database system
- clob (character large object): object is a large collection of character data
- Queries that return a "large object" are really returning a pointer, not the large object itself
  - Instantiated on demand ...

User-Defined Types (I)

- The ability to create “new data types” is very powerful
  - You’re so used to this from Java that you make take it for granted ☺
- Relational databases allow us to create new types and have their semantics enforced at runtime
  - Like any other attribute domain

```
1  create type Dollars as numeric (12,2)
2
3  create table department
4     (dept_name varchar (20), building varchar (15),
          budget Dollars);
```

- CREATE TYPE even allows you to define composite object

```
1  CREATE TYPE full_address AS
2    ( city VARCHAR(90),
3      street VARCHAR(90)
4    )
```

User-Defined Types (II)

- The CREATE DOMAIN construct in SQL-92 is another way to create user-defined domain types
- Example

```
create domain person_name char(20) not null
```

- Types and domains are similar ...
- Domains can have constraints, such as NOT NULL specified on them.
- Example:

```
create domain degree_level varchar(10)
constraint degree_level_test check (value in ('
    Bachelors', 'Masters', 'Doctorate'));
```

	CREATE DOMAIN	CREATE TYPE
Scalar (Single Field) Type	✓	✓
Complex (Composite) Type	✗	✓
Enumeration Type	✓	✓
DEFAULT Value	✓	✗
NULL and NOT NULL Constraint	✓	✗
CHECK Constraint	✓	✗

CREATE TABLE & Defining Constraints

- You define table constraints when you create a database table
  - The constraints apply to rows that will be inserted (later) into the table
- The system enforces these constraints at the appropriate CRUD life-cycle events for the table's rows
- We haven't yet discussed DML syntax, so we'll refer to the CRUD "semantics"

Possible Constraint Violations For “Create Row”

- When creating a new row, the command may violate any of the following constraints
- Domain constraint: if one of the attribute values provided for the new row is not of the specified attribute domain
- Key constraint: if the value of a key attribute in the new row already exists in another row in the relation
- Referential integrity: if a foreign key value in the new row references a key value that does not exist in the referenced relation
- Entity integrity: if the primary key value is null in the new row

Possible Constraint Violations For “Delete Row”

- When deleting a row, the only issue is whether the operation will violate referential integrity
- If the primary key value of the tuple being deleted is referenced by other tuples in the database ...
  - This implies that your “innocent” delete operation breaks the previous defined constraint between the “referenced” tuple and the “referencing” tuple
- The runtime behavior of this violation depends on the details of the foreign key constraint
  - (Read your manual for syntax and possible other options)
  - RESTRICT option: reject the deletion
  - CASCADE option: Delete any rows referencing the deleted row
  - SET NULL option: set the foreign keys of the referencing tuples to NULL
  - SET DEFAULT: Set the referencing attributes(s) to the default values for their domains

Possible Constraint Violations For "Update Row"

- When updating a row, any change to an existing attribute value may violate a domain constraint and a NOT NULL constraint
- Depending on the attribute being modified, other constraints may also be violated
- The semantics of changing a primary key value is the same as "delete the tuple, followed by create a tuple"
  - Because a primary key provides identity for a given tuple
- Implication: the previously discussed issue for "create" and "delete" both come into play
  - Note: the CASCADE option for an update implies: *"update the value of the referencing column to the new value of the referenced column"*
- Changing the value of a foreign key may violate referential integrity (because the DBMS must make sure that the new value refers to an existing tuple in the referenced relation
- Typically, "create" DDL allows separate options for dealing with violations of "update" versus "delete" referential integrity

Examples

- not null
- primary key $(A_1, \ldots, A_n)$
- foreign key $(A_m, \ldots, A_n)$ references r

```
create table instructor (
      ID              char(5),
      name            varchar(20) not null,
      dept_name       varchar(20),
      salary          numeric(8,2),
      primary key (ID),
      foreign key (dept_name) references department);
```

As we've discussed, primary key declaration on an attribute automatically ensures "not null"

Primary Key Integrity Constraint

- As we discussed, any number of attributes can comprise a primary key
- They must all be non-null
- Semantics consideration: you're asserting that this combination of attributes will be unique for all instances of this relation

Foreign Key Integrity Constraint

```
create table instructor (
      ID              char(5),
      name            varchar(20) not null,
      dept_name       varchar(20),
      salary          numeric(8,2),
      primary key (ID),
      foreign key (dept_name) references department);
```

- You're asserting that the values of the foreign key's attributes correspond to primary key attributes of a tuple in the referenced relation
  - You're also asserting that *"this semantic will be true for all current and future tuples in the referencing relation"*
- In this example, asserting that relation *department* has primary key consisting only of *dept_name*
- Key point: without this constraint, a perfectly fine "insert" statement could result in an instructor becoming a member of a non-existent department
  - Not getting paid ☹

Not Null Integrity Constraint

- Constraint excludes the value NULL from the specified attribute's domain
- The implications of this are considerable: requires that we discuss the "null" concept at greater length
  - Later ...
- Constraints are enforced at runtime: execution of statements that violate an integrity constraint will result in
  - An error code returned to the client
  - The statement will not be applied to the database

Unique Constraints

- The "create" DDL can also specify

```
1  CREATE TABLE person (
2          id SERIAL  PRIMARY KEY,
3          first_name VARCHAR (50),
4          last_name  VARCHAR (50),
5          email      VARCHAR (50),
6    UNIQUE(email)
7  );
```

- (Alternatively, can specify EMAIL VARCHAR (50) UNIQUE)
- Can also specify a UNIQUE constraint on multiple columns
- In effect, this constraint states that the set of specified attributes form a candidate key
  - No two tuples in the relation can be "equal" on all of the specified attributes
- Unlike a declared primary key, candidate keys are permitted to be NULL (unless you explicitly declare NOT NULL)

CHECK Clause

- The CHECK (P) clause specifies a predicate *P* that must be satisfied by every tuple in a relation
- This is a "cheap way" to build refinements into a "type system"
  - Example: CHECK BUDGET > 0
- Here is fancier example ...

```
create table section
    (course_id varchar (8),
     sec_id varchar (8),
     semester varchar (6),
     year numeric (4,0),
     building varchar (15),
     room_number varchar (7),
     time slot id varchar (4),
     primary key (course_id, sec_id, semester, year),
     check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

Note: POSTGRESQL does support CHECK constraints, but does not (yet) support assertions (Textbook 4.4.8)

SQL: A Relational Query Language

Introduction

- We're about to begin the study of *"how to query a relational database"*
- Per textbook (Chapter 2.5 & 2.6) we should be discussing the relational algebra first, before we begin SQL
  - Most "relational database" textbooks follow this sequence
  - We're going to do things differently ☺
- Key point: you are very unlikely to interact with a commercial relational database with anything but SQL
  - I just don't think it's worth class time to discuss the relational algebra & calculus
- The ideas behind the relational algebra are very important to us because SQL is (for the most part) a pretty good realization of the algebra
  - So: apologies in advance if I do refer to the algebra from time to time

A Very Brief Digression Into "Programming Languages" Material

- Some students initially have trouble using SQL because ...
  - ...SQL is definitely a type of programming language
  - But: SQL programming differs considerably from e.g., programming in Java (at the least the way we present Java in your introductory courses)
- I'll try to provide some context in the next few slides to help with the adjustment

Imperative Programming: Back To COM 1300

- Imperative programming describes the programming process as a
  1. Series of steps that describes in precise detail what the computer must do and when it has to do it
  2. Process that transforms a set of state $A_0 \rightarrow A_1 \ldots \rightarrow A_n$, describing each of these transformations as a sequence of n steps
- Alternatively: imperative programming describes computation *as a sequence of statements that change the state of the program, evolving the program's state to reach a certain goal*
- Imperative programming is often contrasted to declarative programming

Declarative Programming (I)

> Imperative programming tells the computer how to compute: declarative programming tells the computer what to compute

- Key point: the above point is insightful but should be viewed as a "fractal" definition
- For example: did you ever "tell the computer how to compute 3/4.2"?
- Of course not!
- You told the program what to compute, and waited patiently for the result ☺
- Implication: "declarative" programming at one level of abstraction is "imperative" programming at a higher level of abstraction …

Declarative Programming (II)

- SQL is a declarative language
  - Example: a SELECT statement specifies what data to return
  - The database's query optimizer and execution engine are responsible for figuring how to determine & retrieve the relevant data
- Q: so where does functional programming fit in to this picture?
- A: (donning asbestos gloves) ☺ functional programming is a subset of declarative programming
  - Functional programming is about computations that only involve a "pure" functional transformation of data
  - MAP/REDUCE programs are a good example of FP
  - Most SQL queries are a good example of FP

Relational Query Languages (I)

- Query languages: Enable retrieval of data from a database
  - This implies that CUD operations are not part of a query language
- That said: the DML statements that are part of SQL do include commands that "create", "update", and "delete" tuples ☺
  - Today's lecture discusses only "query"
  - We'll only discuss CUD operations at the end of our SQL lectures
- Key observation: relational model naturally supports simple, powerful query languages
  - The model is based on a strong formal foundation based on logic and set theory
  - Because of the rigorous specification, allows for considerable optimization (that's the "declarative" aspect discussed a few slides back)

Relational Query Languages (II)

- Query languages should not be confused with programming languages
- Query languages are:
  - Not: Turing complete (we'll touch on this later)
  - Not: suitable for complex calculations
- Note: these limitations are very closely tied to the fact that SQL is a functional language ☺
  - The limitations of SQL are typically overcome by embedding SQL in "host languages" such as C, Cobol, Java

Relational Algebra: Analogy

In mathematics, a basic algebraic operation is any one of the traditional operations of arithmetic, which are addition, subtraction, multiplication, division, raising to an integer power, and taking roots (fractional power). These operations may be performed on numbers, in which case they are often called arithmetic operations. They may also be performed, in a similar way, on variables, algebraic expressions, and, more generally on elements of algebraic structures, such as groups and fields.

Algebraic Operations

- The Relational Algebra defines a set of operations on relations
- These operations take relations as input
- These operations return relations as output

Basic Query Structure (I)

- Today: only dealing with the following possible clauses in a SELECT statement
  - SELECT
  - FROM
  - WHERE
- The generic structure of an SQL query can be written as:

select $A_1, A_2, \ldots A_n$ from $r_1, r_2, \ldots r_m$ where P

Here:

- A_i represents an attribute
- r_i represents a relation
- P is a predicate

Basic Query Structure (II)

The next slides will drill down into the semantics of the clauses in this basic query structure

select $A_1, A_2, \ldots A_n$ from $r_1, r_2, \ldots r_m$ where P

Logical processing order:

1. FROM: compute the Cartesian product of the relations
2. WHERE: apply the predicate to restrict the combinations generated by the Cartesian product to those that are relevant for our query's semantics
3. SELECT: remove extraneous attributes from the output relation

Projection: Select Subset of Columns

R	A	B	C	D
	1	10	100	1000
	1	20	100	1000
	1	20	200	1000

SQL

```sql
SELECT B, A, D
FROM R;
```

	B	A	D
	10	1	1000
	20	1	1000
	20	1	1000

- Note: duplicates are allowed! And can be removed
- Relational algebra symbol: π

How Would You Implement Projection?

Thought exercise only!

- R is a file of records
- Each record is tuple
- Each record consists of fields (values of attributes)

1. Create a new output file
2. Iterate over the records of R
3. Create a new record by selecting only the specified fields of each record
4. Append the new record to the output file

Selection: Select Subset of Rows

R	A	B	C	D
	5	5	7	4
	5	6	5	7
	4	5	4	4
	5	5	5	5
	4	6	5	3
	4	4	3	4
	4	4	4	5
	4	6	4	6

SQL

```
1  SELECT *  /* ''star'' means all columns */
2  FROM R
3  WHERE A <= C AND D = 4; /* WHERE clause is a predicate */
4  SELECT B, A, D
5  FROM R;
```

	A	B	C	D
	5	5	7	4
	4	5	4	4

- Relational algebra symbol: σ

How Would You Implement Selection?

Thought exercise only!

- R is a file of records
- Each record is tuple
- Each record consists of fields (values of attributes)

1. Create a new output file
2. Iterate over the records of R
3. Check if current record satisfies some conditions on the values of the field
   - Does “applying the predicate to the record” return `true`?
4. Only if “true”, append the record to the new file

Selection Predicates

- Predicates are a Boolean expression consisting of
- NOT, AND, OR clauses applied to atomic conditions
- Where atomic conditions are a comparison between
  - Two column names (this is like "dereferencing a programming variable")
  - A column name and a constant
- If no predicate is specified, then all tuples satisfy the predicate
  - We say that *"All tuples match"*

```
1    WHERE  mycol > 100 AND item = 'Hammer'
2
3    WHERE  mycol IS NULL OR mycol = 100
```

Cartesian Product: All Pairs of Rows

R	A	B
	1	10
	2	10
	2	20

S	C	B	D
	40	10	10
	50	20	10

SQL

```
1  SELECT A, R.B, C, S.B, D
2  FROM R, S;  /* The ‘‘comma’’ stands for Cartesian product */
```

	A	R.B	C	S.B	D
	1	10	40	10	10
	1	10	50	20	10
	2	10	40	10	10
	2	10	50	20	10
	2	20	40	10	10
	2	20	50	20	10

- Relational algebra symbol: $\times$

How Would You Implement Cartesian Product?

Thought exercise only!

- R and S are files of records
- Each record is tuple
- Each record consists of fields (values of attributes)

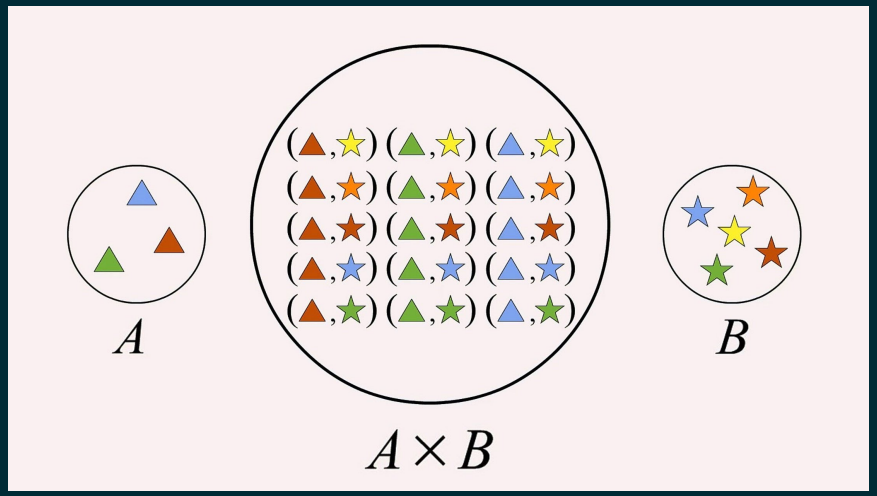
1. Create a new output file
2. Iterate over the records of R (“outer loop”)
3. Iterate over the records of S (“inner loop”)
4. Combine the record from R with the record from S
5. Append the “combined” record to the output file

Cartesian Product Syntax

> *For the Cartesian product to be defined, the two relations involved must have disjoint headers - that is, they must not have a common attribute name*
>
> *Wikipedia, "Set Operators"*

- SQL removes this limitation by using "dot notation" to create unique attribute names
  - $relation_1.foo$ *versus* $relation_2.foo$

Cartesian Product Semantics



- Relational CP has slightly different semantics from vanilla set theory
- As shown in the above figure, vanilla CP creates a set of two-tuples whose first element is an element $a \in A$ and whose second element is an element $b \in B$
- Relational CP creates a set of flattened tuples consisting of tuples from A "concatenated" with tuples from B
- $A \times B = \{(a_1, a_2, \ldots a_n, b_1, b_2, \ldots b_n) | ((a_1, a_2, \ldots a_n) \in A, (b_1, b_2, \ldots b_n) \in B\}$

Natural Join versus Cartesian Product

- Textbook (chapter 2) introduces both "natural join" and "Cartesian product" operations
  - Natural join: "pairs of rows that have the same value on all attributes that have the same name"
    - Symbol is $\bowtie$ (Latex symbol: backslash "bowtie", in math mode)
  - Cartesian product: "pairs of rows regardless of whether or not they have the same values on common attributes"
- Today's lecture focuses on "fundamental operations"
- "natural join" is syntactic sugar sprinkled on top of the "Cartesian product" operation
  - Includes an implicit Selection operation to only return tuples whose values match on common attributes

Query Scenario: I

R	Size	Room#
	140	1010
	150	1020
	140	1030

S	ID#	Room#	YOB
	40	1010	1982
	50	1020	1985

- Relation *R* stores information about rooms and their room size
- Relation *S* stores information about employees, and includes data about their work location
- Task: Return the room size of all employee work locations
- How would you formulate this query?
  - Alternatively: how do you combine the necessary relational algebra operations?

Query Scenario: II

```
1  WHERE R.Room# = S.Room#;
```

- All we really need is to select tuples that match this predicate
- Problem: WHERE can be applied to only one tuple at a time ☹
- In other words: to apply the WHERE clause in the way we want, we must first “combine the relations into one”

Query Scenario: III

- We need to combine tuples from the two relations
- (In subsequent lectures, we'll discuss why we had to separate them in the first place ☺)
- That's a "Cartesian product" operation
  - We could combine tuples from *n* relations, not limited to 2
- Applying the Cartesian product produces this relation

	Size	R.Room#	ID#	S.Room#	YOB
	140	1010	40	1010	1982
	140	1010	50	1020	1985
	150	1020	40	1010	1982
	150	1020	50	1020	1985
	140	1030	40	1010	1982
	140	1030	50	1020	1985

Query Scenario: IV

```
1  SELECT ID#, R.Room#, Size
2  FROM R, S
3  WHERE R.Room# = S.Room#;
```

- Once we have a single relation (from the Cartesian product)
- We apply the WHERE clause to get the output relation

	ID#	R.Room#	Size
	40	1010	140
	50	1020	150

Note: Natural join version would implicitly include the `WHERE R.Room# = S.Room#` clause

Today's Lecture: Wrapping it Up

DDL: CREATE TABLE & Domains

CREATE TABLE & Defining Constraints

SQL: A Relational Query Language

Readings

- Textbook discusses DDL & DML in 3.1, 3.2, and 3.9
- Textbook discusses various types of database constraints in Chapter 4.3, and 4.4
- Textbook discusses SQL data types & domains in Chapter 4.5
- Textbook discusses the structure of a basic SQL query in Chapter 3.3