

# CSC343 Notes

Max Xu

January 11, 2026

## Contents

<b>1 Day 1: Intro (Jan 07, 2026)</b>	<b>2</b>
<b>2 Day 2: The Relational Model (Jan 09, 2026)</b>	<b>4</b>
<b>3 Day 3: The Relational Algebra (Jan 14, 2026)</b>	<b>6</b>
3.1 Pre-Class .....	6

## §1 Day 1: Intro (Jan 07, 2026)

Why study databases:

- Interesting concepts and techniques
- Employment reasons
- Research: open problem (e.g. VLDB)

The word ‘database’ is used colloquially. To be precise, some terminology:

**Definition 1.1** (Database). The data itself.

**Definition 1.2** (DBMS). **Database Management System** (DBMS) is software, for creating (entering data into the tool) and managing large amounts of data efficiently. It must persist data (e.g. to survive natural disasters), and ensure safety and consistency in data (avoid race conditions).

DBMS is based off some data model which specifies the

- structure of the data
- constraints on the data
- allowed operations on the data

In addition, the DMBS must also

1. specify logical structure (explicitly and enforce it), maintaining indices (metadata)
2. be able to query and modify the data
3. be performant (optimization of queries)
4. be durability
5. allow for concurrent access

DBMS manages buffers and disk space (instead of the OS!), as the DBMS is more informed about what needs to be done.

Our course focuses on the **relational data model** (very old, and strict), which is based on relations from math. In past classes, we learned that pointers are messy, and almost everything required custom code. A relational data model provides simplicity: the spirit of a relational database is that each cell contains only 1 ‘piece’.

We will be using **PostgreSQL**. It’s (and more generally relational databases are) meant for ‘rectangular information’. You must formally define a schema, and strictly adhere to it. All rows should have data for all columns. If you must have holes, use another structure to get around this.

There exist other data models not covered in this course, being semi-structure data model (JSON, XML) and unstructured data, and graph data model, where information represented in nodes and edges, queries defined by paths.

This class is about *using* DBMSs. We will be

- defining schemas and instances
- writing queries
- connecting PostgreSQL to code written in a general-purpose language (Python)
- understanding the rigorous underlying principles (some proofs)
- focusing on the relational model, to build foundations

The logical next step after this course is **CSC443**, where you will understand the design choices and implementation side of things.

## §2 Day 2: The Relational Model (Jan 09, 2026)

Recall that the relational model is based on the concept of a relation/table. We will explore this notion this class.

Let  $D_1, \dots, D_n$  be domains. The **Cartesian product**  $D_1 \times \dots \times D_n$  is the set

$$\{\langle d_1, \dots, d_n \rangle : d_1 \in D_1, \dots, d_n \in D_n\}$$

where  $\langle \cdot \rangle$  is called a **tuple/ordered list**. A **relation** on  $D_1, \dots, D_n$  is a subset of  $D_1 \times \dots \times D_n$ .

Notice that a relation is a set, thus, in our database,

- there are no duplicates
- order of the tuples do not matter

Real DBMS do allow for duplicates, amongst other conditions.

The empty set  $\{\}$  and the entire Cartesian product itself are both examples of relations.

To bridge the gap between math/database terminology, the following pairs are equivalent in their meaning:

- relation/table
- attribute/column
- tuple/row
- arity of a relation/number of attributes
- cardinality of a relation/number of tuples

### Remark 2.1

As for empty/optional values, they are denoted by `None` in PostgreSQL. We consider it ‘bad design’ to have a column with a lot of `None` values, and will go over techniques to avoid this later.

The **schema** is the structure of a relation. The **instance** is the content of the relation. Database schema and database instances are sets of relation schema and sets of relation instances respectively.

Instances change frequently, but schema should rarely change. Most databases store the current version of the data. Those that also store the history are called ‘temporal’ databases.

We now want to declare constraints on our relation. We do so using keys.

**Definition 2.2 (Keys).** A **key** for a relation is a set of attributes  $\{a_1, \dots, a_n\}$  where

1. Uniqueness: There do not exist distinct tuples  $t_1, t_2$ , where  $\bigwedge_{i=1}^n t_1.a_i = t_2.a_i$

2. Minimality: no subset of  $\{a_1, \dots, a_n\}$  satisfy uniqueness

A key defines a constraint on the integrity of the data, originating from **domain experts**. The instance must conform to said constraints. If we could encounter duplicate rows, we can invent additional attributes to ensure that keys exist.

Any superset of a key is called a **superkey** (useful in theory). We mostly declare keys in practice. Immediately, every key is a superkey.

**Definition 2.3** (Foreign Key). A foreign key is a subset of attributes of some relation  $R_1$ , such that it is also a key for relation  $R_2$ .

Let  $R$  be a relation, and  $A$  a list of attributes in  $R$ . Define  $R[A]$  as the set of all attributes from  $R$ , with only the attributes in list  $A$ .

**Foreign key constraints** are declared as  $R_1[X] \subseteq R_2[Y]$ , where  $X, Y$  have the same arity, and  $Y$  must be a key in  $R_2$ . This is an instance **referential integrity constraint** (though not all referential integrity constraints are foreign key constraints).

## §3 Day 3: The Relational Algebra (Jan 14, 2026)

### 3.1 Pre-Class

A **query** on a set of relations produces a relation as a result. We perform queries on our database, which is a set of relations.

The semantics of relational algebra says that duplicates from queries are always eliminated, as opposed to SQL, which operates with multisets/bags. Some queries include:

**Relation Name** Obtains a copy of the relation of the same name. Returns an entire relation.

**Select** Picks certain rows from relation  $R$  with condition  $cond$ . Written as  $\sigma_{cond} R$ . Returns a subset of said relation.

$\sigma_{cond}$  Expr functions similarly, where Expr is any expression of the algebra.

**Project**  $\Pi_{attr_1, attr_2, \dots}$  Expr picks the columns with attribute  $attr_1, attr_2, \dots$  respectively from the expression Expr.

**Cross-product** Combines two expressions, written  $Expr_1 \times Expr_2$ . This is the standard Cartesian product for sets.

When two relations have the same attribute name, we add the relation name as a prefix to distinguish them.

**Natural Join** Similar to cross-product, except it enforces equality on all attributes with the same name, and eliminates one copy of duplicate attributes. Written  $Expr_1 \bowtie Expr_2$ .

**Theta Join** Let  $\theta$  be some condition.  $Expr_1 \bowtie_\theta Expr_2 \equiv \sigma_\theta(Expr_1 \times Expr_2)$ .

The natural join and theta join add no expressive power to the language.