CSC343 Notes

Max Xu

January 14, 2026

Contents

§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:

(a) specify logical structure (explicitly and enforce it), maintaining indices (metadata)

(b) be able to query and modify the data

(c) be performant (optimization of queries)

(d) be durability

(e) 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

(i) defining schemas and instances

(i) writing queries

(i) connecting PostgreSQL to code written in a general-purpose language (Python)

(i) understanding the rigorous underlying principles (some proofs)

(i) 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 $D_1, \cdots, D_n$ be domains. The **Cartesian product** $D_1 \times \cdots \times D_n$ is the set

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

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

A relation is a set, thus, in our database,

(i) there are no duplicates

(ii) 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 `NULL` in PostgreSQL. We consider it 'poor design' to have a column where `NULL` values frequently appear, 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 $\{a_1, \cdots, a_n\}$ where

(i) Uniqueness: There do not exist distinct tuples t_1, t_2, where for all indices i, $t_1.a_i = t_2.a_i$

(ii) Minimality: no subset of $\{a_1, \cdots, a_n\}$ satisfies uniqueness

In other words, no two tuples may be identical in the fields specified by a key. Some notation: Artists($\underline{a_1, a_2}, a_3, a_4$) means that $\{a_1, a_2\}$ is a key for the relation Example with attributes a_1, a_2, a_3, a_4. If they were not adjacent, explicitly stating the key set also works.

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 of a **referential integrity constraint** (though not all referential integrity constraints are foreign key constraints).

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

Pre class readings: this video.

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.

We will drop these assumptions when we work with SQL, but for now, we assume the following:

(i) Relations are sets (no duplicates)

(ii) Every cell is filled (every tuple has all attributes)

Some types of queries include:

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

Select Removes rows from R that do not satisfy condition *cond*. Written as $\sigma_{cond}\, R$. Returns a subset of said relation.
σ_{cond} Expr functions similarly, where Expr is any expression of the algebra.

Project $\Pi_{attr1,attr2,\ldots}$ Expr picks the columns with attribute $attr_1, attr_2, \cdots$ respectively from the expression Expr.

Cross-product Combines two expressions, written $\text{Expr}_1 \times \text{Expr}_2$. 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.

Theta Join Let θ be some condition. $\text{Expr}_1 \bowtie_\theta \text{Expr}_2 \equiv \sigma_\theta(\text{Expr}_1 \times \text{Expr}_2)$.

Natural Join Written $\text{Expr}_1 \bowtie \text{Expr}_2$, it is similar to the cross-product, but with additional steps when Expr_1 and Expr_2 have shared attributes. In the resulting relation, for each attribute *attr*, tuples whose $\text{Expr}_1.attr$ and $\text{Expr}_2.attr$ differ are removed. A projection then occurs, consolidating $\text{Expr}_1.attr$ and $\text{Expr}_2.attr$ into the same column.

The natural join and theta join add no additional expressive power to the language. They are defined for convenience.

So far, we have encountered 3 kinds of integrity constraints.

(i) **Key Constraint**

e.g. Roles($\underline{mid, aid}, character$)

(ii) **Referential Integrity Constraint**

e.g. Artists[aid] $\subseteq$ Roles[aid]. $\{aid\}$ may or may not be a key for Artists.

- **Foreign Key Constraint**

  Artists[aid] $\subseteq$ Roles[aid], where $\{aid\}$ is a key for Artists.

In schema design, the schema should represent the data well and avoid redundancy.