CS 143

Warren Kim

Contents

Overview

1.1 Purpose of a Database

We will be studying (mostly) Relational DataBase Management Systems (RDBMS).

Definition: Database

A **database** abstracts how data is stored, maintained, and processed. It is a system that uses advanced data structures to store and index data.

A database abstracts away the data integrity and file management aspect of CRUD operations. Moreover, a database provides us with a single location for all of the data, even if the database itself is distributed.

1.2 Abstraction Layers

There are three layers of abstraction: physical, logical, and view.

Definition: Physical Abstraction

The **physical abstraction** defines the data and its relationships to other data within the database.

Definition: Logical Abstraction

The **logical abstraction** deals with how we interface with the database.

Definition: View Abstraction

The **view abstraction** refers to specific use cases and filters the data from the logical abstraction.

We start by learning the logical abstraction.

1.3 Instances and Schema

Definition: Schema and Instance

A **schema**[a] is the overall design of a database. It defines the structure of the data as well as how it is organized.

An **instance** of a database is the actual set of data stored in the database at a particular moment in time.

[a]Note: schema can also refer to a relation (table).

1.4 Data Models

Data models define how we design databases and interact with data. We want to answer the following:

(i) How do we define data?

(ii) How do we encode relationships among data?

(iii) How do we impose constraints on data?

Data models are either an Implementation model or a Design mechanism. Implementation models build databases from the ground up while design mechanisms are implemented as features in a database. We discuss five major types (an several niche ones).

1.4.1 Relational

In a relational model, all data is stored as a ***relation***[1]. Rows represent individual n-tuple units (***records***). Columns represent (typed) ***attributes*** common to all records in the relations.

1.4.2 Entity-Relationship (ER)

An entity-relationship model uses a collection of basic objects (***entities***) and define ***relationships*** among them.

1.4.3 Object-Oriented

The object-oriented model is similar to OOP with encapsulation, methods, adn object identity. It was originally an implementation model but is now a design mechanism.

1.4.4 Document (Semi-Structured)

A document model stores records as ***documents***, which do ***not*** have an enforced schema. This allows for more versatility in the type of data stored in the database.

1.4.5 Network/Hierarchical/Graphical

A graph model is analogous to how we think. Records are stored as ***nodes*** and relationships between records as ***edges***.

1.4.6 Vector

A vector model stores records as ***vectors*** in $\mathbb{R}^n$, and are stored in a way that enables efficient retrieval and comparison (e.g. nearest neighbor[s]).

1.4.7 Key-Value

A key-value model stores data as a key-value pair (typically using a hash function). In this model, data typically lives in RAM as opposed to disk.

1.5 Database Languages

There are two main semantic systems when working with databases:

(i) Data Manipulation Language (DML)

(ii) Data Definition Language (DDL)

Note that a relational model typically uses SQL for both DDL and DML.

[1]Note: tables are an implementation of relations.

1.5.1 Data Manipulation Language

DML's can either be procedural or declarative.

Definition: Query

A ***query*** is a written expression to retrieve or manipulate data.

Aside: A Note on SQL

SQL is a declarative language, and as such, it is hard to perform sequential or nontrivial[a] computations in SQL. To remedy this, a common option is to write an ***ETL job*** in another language (pick one). We **E**xtract the data from the database (using a connection driver), **T**ransform the data using another lanuage (pick one!), and **L**oad the data into a new table using the same driver. We can schedule these jobs using something like `cron`.

[a]Nontrivial: Any computation where we have to specify *how* to perform the computation.

1.5.2 Data Definition Language

DDL's specify a schema: a collection of attribute names and data types, consistency constraints, and optionally storage structure and access methods. There are four types of consistency constraints:

(i) Domain constraints define the domain of an attribute (e.g. `tinyint`, `enum`, etc.).

(ii) Assertions are business rules that must hold true (e.g. an enforced prerequisite for a class must be present in your transcript before you can add a class to your study list).

(iii) Authorization determines who can do what (e.g. full CRUD, read-only, etc.).

(iv) Referential integrity ensures that links from one table to another must be defined (Suppose we have two relations R, R'. If there is a link $f : R \to R'$, then f is surjective).

1.6 Data Storage and Querying

Definition: Storage Manager

A ***storage manager*** that abstracts away how the data is laid out on disk.

A storage manager is helpful because reading data from disk to RAM is *slow*, and the storage manager handles swapping[2] and makes retrieval efficient.

Definition: Query Manager

A ***query manager*** takes the DML statements and organize them into a *query plan*[a] that "compiles" a query (using relational algebra) and executes the instruction(s).

[a]Note: The query plan dictates the performance of a query.

1.7 Defining a Schema

A schema can be written as relation($\underline{\texttt{attribute}_1}$, ..., $\texttt{attribute}_\texttt{n}$) where underlined attributes represent the primary key.

[2]Swapping: Virtual memory in CS111!

Keys

Aside: A Note on Context and Instance

Based on **context** means that the given data is a subset of the complete dataset.
Based on **instance** means that we treat the given data as the complete dataset.

2.1 Superkey

Definition: Superkey

A **superkey** is a set of one or more attributes that uniquely identifies a record (tuple) and distinguishes it from all other records in the relation.

Formally, let R be a relation with a set $S = \{a_1, a_2, \ldots, a_n : a \text{ is an attribute of } R\}$. A **superkey** is a subset $s \subseteq S$ such that s uniquely identifies each n-tuple in R.

The superkey $s = S = \{a_1, a_2, \ldots, a_n\} = \bigcup_{i=1}^{n} \{a_i\}$ is called the ***trivial superkey***. Additionally, $\emptyset$ is not a superkey. Further note that for every relation R, there exists at most $2^n - 1$ superkeys where n is the number of attributes.

2.2 Candidate Key

Definition: Candidate Key

A **candidate key** is a superkey such that no subset of the candidate key is a superkey; i.e. it is the minimal superkey.

Formally, let R be a relation with a set $S = \{a_1, a_2, \ldots, a_n : a \text{ is an attribute of } R\}$. A **candidate key** is a superkey $s \subseteq S$ such that for every propery subset $t \subsetneq s$, t is not a superkey.

Candidate keys may vary in length, and the attributes of a candiate key may be `NULL` as long as it uniquely identifies an n-tuple in the relation.

2.3 Primary Key

Definition: Primary Key and Composite Key

A **primary key** is a candidate key (chosen by the database designer) to enforce uniqueness for a particular use case.

The primary key is typically chosen to be the minimal candidate key for simplicity. The attributes of a primary key may not be `NULL`.

2.4 Foreign Key

Definition: Foreign Key

A **foreign key** is a set of attributes that links tuples of two relations.

Formally, let R, R' be relations with sets $S = \{a_1, a_2, \ldots, a_n : a \text{ is an attribute of } R\}, S' = \{a'_1, a'_2, \ldots, a'_n : a' \text{ is an attribute of } R'\}$. A **foreign key** is a key $s \subseteq S$ of R that maps to the primary key $p \subseteq S'$ of R'.

Foreign keys are used to enforce referential integrity constraints; i.e. foreign keys in a relation R are used to protect data in R from being orphaned and/or inconsistent. Given two relations R, R' related via a foreign key, R' is said to be the *referring* relation and R the *referred* relation.
Let two relations R, S be related via a foreign key, where S is the *referring* relation and R is the *referred* relation. Suppose we want to remove an n-tuple $r \in R$. Then there are two cases:

Case 1 If there is no $s \in S$ such that $s \mapsto r$, we simply remove r.

Case 2 If there is at least one $s \in S$ such that $s \mapsto r$, we can either throw an error to prevent the deletion of r or *cascade*[1] the delete.

[1]Cascade: Delete r and all $s \in S$ that refer to r.

Relational Algebra

3.1 Selection

Definition: Selection

Selection retrieves a subset of tuples from a *single* relation R that satisfies some predicate ψ and returns a new relation $R' \subseteq R$, and is defined by

$$\sigma_{\psi}(R) = R' = \{t \in R : \psi(t)\}$$

where ψ is a boolean predicate on attributes and values with respect to a unary or binary operator[a]

[a] We may use the following operators: $\{=, \neq, <, >, \leq, \geq, \neg\}$.

We can build complex predicates using conjunction $\wedge$ (and) or disjunction $\vee$ (or).

Note: that selection σ is the most analogous to `WHERE` in SQL.

Below are a list of examples of selection, assuming all attributes and relations are well-defined:

(i) $\sigma_{(\texttt{dislikes}<\texttt{likes})}(\texttt{youtube\_video})$

(ii) $\sigma_{(\texttt{cat\_id}=17)}(\texttt{youtube\_video})$

(iii) $\sigma_{([\texttt{dislikes}<\texttt{likes}]\wedge[\texttt{views}>1000000]\wedge[\texttt{cat\_id=24}])}(\texttt{youtube\_video})$

(iv) $\sigma_{(\texttt{dislikes}<\texttt{likes})}(\sigma_{(\texttt{views}>1000000)}(\sigma_{(\texttt{cat\_id}=24)}(\texttt{youtube\_videos})))$

Note that *(iii)* and *(iv)* are equivalent.

3.2 Projection

Definition: Projection

Projection extracts attributes from a set of tuples and removes duplicates. Given a relation R, n-tuple t, and a set of attributes $a_1, \cdots, a_n$,

$$\Pi_{a_1,\cdots,a_n}(R) = \{t[a_1, \cdots, a_n] : t \in R\}$$

Projection is usually the last (outermost) operation done on a relation.

Aside: Projection?

We call it a projection because we are collapsing an n-tuple down to an $(n-k)$-tuple. That is, we take the n-tuples in a relation R_n and collapse them into a set of $(n-k)$-tuples in a new relation R'_{n-k}.

Here, R_n is a relation with n attributes.

Projections can be generalized to "create" new attributes or rename attributes using the $\rightarrow$ notation.

Example

We can apply arbitrary expressions to existing attributes (and create another one) by doing $\Pi_{\texttt{likes/(likes+dislikes)}\to\texttt{interactions}}(R)$ or rename attributes by doing $\Pi_{\texttt{likes}\to\texttt{thumbs\_up}}(R)$

Example

Consider the following relation R with $\Pi_{A,B}(R)$:

$$\begin{array}{c|c|c} A & B & C \\ \hline \alpha & \beta & \delta \\ \alpha & \beta & \gamma \\ \alpha & \beta & \lambda \end{array} \xrightarrow{Extract A,B} \begin{array}{c|c} A & B \\ \hline \alpha & \beta \\ \alpha & \beta \\ \alpha & \beta \end{array} \xrightarrow{\Pi_{A,B}(R)} \begin{array}{c|c} A & B \\ \hline \alpha & \beta \end{array}$$

Note: that projection Π is the most analogous to `SELECT DISTINCT` in SQL.

3.3 Cartesian Product

Definition: Cartesian Product

A **Cartesian product** forms all possible pairs of tuples. Given relations R, S,

$$R \times S = \{(r, s) : r \in R \wedge s \in S\}$$

Example

Suppose we have two relations R, S defined below:

$$R := \begin{array}{c|c} A & B \\ \hline \alpha & \beta \\ \alpha & \gamma \end{array}, S := \begin{array}{c|c} A & B \\ \hline \alpha & \gamma \\ \Delta & \eta \end{array}$$

Then,

$$R \times S = \begin{array}{c|c|c|c} A & B & C & D \\ \hline \alpha & \beta & \alpha & \gamma \\ \beta & \gamma & \Delta & \eta \\ \Delta & \eta & \alpha & \gamma \\ \Delta & \eta & \beta & \gamma \end{array}$$

Cartesian products are *very* expensive since they require a lot of compute power, ram, and disk space.

3.4 Aggregation

Definition: Aggregation

The aggregation operator (γ or $\mathcal{G}$) is a function on groups of tuples in a relation to summarize them. Common ones include: `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`, etc. Given a relation R,

$${}_A\gamma_F(R) = {}_A\mathcal{G}_F(R)$$

where $A :=$ {attributes to group by}, $F :=$ {functions to apply}

3.5 Rename

Definition: Rename

The rename operator (ρ) renames relations or attributes. Given a relation with name R, renaming a relation looks like $\rho_{R'}(R)$, where R' is the new name. Renaming an attribute in R looks like $\rho_{a'/a}(R)$, where a' is the new name.

Note: We must rename one of the R's when doing $R \times R$. That is, we must have $\rho_{R'}(R) \times R$ or $R \times \rho_{R'}(R)$.

3.6 Set Operations

Definition: Set Operations (Union, Intersection, Set Difference)

Let R, S be two sets of tuples. Then, union is defined to be

$$R \cup S = \{r_1, \cdots, r_{|R|}, s_1, \cdots s_{|S|} : r_i \in R \vee s_j \in S\}$$

intersection is defined as

$$R \cap S = \{t : t \in R \wedge t \in S\}$$

and set difference is defined as

$$R - S = R \setminus S = \{t : t \in R \wedge t \notin S\}$$

3.7 Order of Precedence

The order of precedence from highest to lowest is as follows:

$$(\sigma, \Pi, \rho), (\times, \bowtie), \cap, (\cup, -)$$

Week 3

4.1 Join

Definition: Join

A **join** merges tuples from two relations R, S based on some contextually related attribute(s) in both relations. The resulting relation contains tuples of the form (r, s) where $r \in R, s \in S$.

A **join key** is the set of attribute(s) that are used to join R and S. Note that a join key is ***completely unrelated*** to do with uniqueness.

There are two types of joins: natural and theta.

4.2 Natural Join

Definition: Natural Join

A **natural join** $\bowtie$ is a join where the join key is determined by the RDBMS. The simplest natural join is defined using the cartesian product:

$$R \bowtie S = \Pi_{R \cup S} \left(\sigma_{R.k=S.k}(R \times S) \right) = \{(r, s) : r \in R \wedge s \in S \wedge (r[k] = s[k])\}$$

The natural join is also characterized as an *equijoin*.

Edge Cases

1. If we have two relations R, S that have no common attributes ($k = \emptyset$), then

$$R \bowtie S = \{(r, s) : r \in R \wedge s \in S \wedge (r[k] = s[k])\} = R \times S$$

   because the empty set is unique.

2. If we have two relations R, S that have common attributes but no matches, then

$$R \bowtie S = \{(r, s) : r \in R \wedge s \in S \wedge (r[k] = s[k])\} = \emptyset$$

   because $r[k] = s[k]$ is always false.

4.3 Theta Join

Definition: Theta Join

A **theta join** $\bowtie_\theta$ is a join where the join key and condition are specified. Mathematically,

$$R \bowtie_\theta S = \sigma_\theta(R_1 \times R_2) = \{(r, s) : r \in R \wedge s \in S \wedge \theta((r, s))\}$$

where θ is the join condition.

Note: For any join, if there is a name clash in either the relation or attribute(s), we must alias them.

Example

Suppose we have two relations R, S defined below:

$$R := \begin{array}{c|c} \text{A} & \text{B} \\ \hline \alpha & \beta \\ \beta & \beta \end{array}, \; S := \begin{array}{c|c} \text{A} & \text{B} \\ \hline \beta & \alpha \\ \alpha & \gamma \end{array}$$

Define $\theta := R.A = S.A$. Then,

$$R \bowtie_\theta S = R \bowtie_{R.A=S.A} S = \begin{array}{c|c|c|c} \text{A} & \text{B} & \text{C} & \text{D} \\ \hline \alpha & \beta & \alpha & \gamma \\ \beta & \beta & \beta & \alpha \end{array}$$

4.4 Inner Joins

Definition: Theta Join

An **inner join** between two relations R, S is a theta join that omits elements that do not satisfy the join condition θ.

PostgreSQL Data Types

The ANSI SQL standard defines the following data types:

(i) numeric

(ii) string/text

(iii) binary

(iv) dates and times

Aside: Promoted

It is always good practice to only promote types to increaese precision, and never demote.

5.1 Numbers

Numeric data types have the following forms:

(i) int(eger) (4 bytes)

(ii) smallint (2 bytes)

(iii) bigint (8 bytes)

(iv) numeric(n, d), where n is the number of digits and p is the number of digits that appear after the decimal point.

(v) real, double precision

(vi) float(n), where n is the precision.

5.2 Strings

String data types have the following forms:

(i) char(n) is a fixed-length character array of length n.

(ii) varchar(n) is a variable-length character array of length $\leq n$.

5.3 Strings

String data types have the following forms:

(i) date (YYYY-MM-DD)

(ii) time (HH:MM:SS)

(iii) timestamp (YY-MM-DD)