CS561: Database Management Systems Notes

Steven DeFalco

Fall 2023

Contents

1 Introduction

Database management systems (DBMS) consist of **data**, **software** (programs such as data interfaces), and **environments** (operating systems). DBMS contains information about a particular enterprise. They are collections of interrelated data, a set of programs to access the data, and an environment that is both *convenient* and *efficient* to use. The *user* should only have to define what it is that they want from the database, whereas the *database* is responsible for defining how this query can be fulfilled; relational databases are good at this.

The three primary data models are ***entity-relationship model*** (diagrams), ***relational model*** (relational algebra), ***relational database*** (SQL).

Drawbacks to using **file systems** to store data include the following:

- data *redundancy* and *inconsistency* (multiple formats, duplication of information, etc.)
- difficulties in *accessing* data
- data isolation (multiple files and formats)
- concurrency issues (among multiple users)
- *integrity* problems
- atomicity of updates
- security problems (hard to provide varied levels of user access)

There are varying ***levels of abstraction*** in a database. The ***physical level*** defines how a record is stored. The ***logical level*** describes data stored in the database and the relationships among data. The ***view level*** is a way to hide details of data types and information for security purposes.

The ***schema*** is the logical structure of the database; this is analagous to type information of a variable in a program. ***Physical schema*** refer to database design at the physical level. ***Logical schema*** refer to database design at the logical level. An ***instance*** is the actual content of the database at a particular time; this is analagous to the value of a variable. ***Physical data independence*** is the ability to modify the physical schema without changing the logical schema.

Remark The schema of a table is the attributes of the table. For example, the schema of a table whose column titles are "A,B,C,D" is simply A,B,C,D.

Data manipulation languages (DML) are languages for accesing and manipulating the data organized in a DBMS. ***Procedural languages*** are ones in which the user specifies what data is required and how to get that data.

Declarative (nonprocedural) languages are ones in which the user specifies what data is required without specifying how to get such data. **SQL** is the most widely used query language.

A ***data definition language (DDL)*** is the specific notation for defining the database schema. The ***DDL compiler*** generates a set of tables stored in a data dictionary. Data dictionary contains metadata.

A ***relational database*** is based on the relational data model. Data and relationships among the data are represented by a collection of tables. These include both a **DML** and **DDL**. The most common relational database systems employ the **SQL** query language.

2 Entity-Relationship Model

A *database* can be modeled as a collection of entities or a relationship among entities. An ***entity*** is an object that exists and is distinguishable from other objects (e.g. specific person, company, even, plant). These *entities* have ***attributes*** (e.g. people hvae names and addresses). An ***entity set*** is a set of entities of the same type that share the same properties (e.g. set of all persons, companies). In the ER-model we refer to specific objects as *entities* which have *attributes* and are all a part of the entire *entity set*.

A ***relationship*** is an association among several entties. A ***relationship set*** is a mathematical relation among $n \geq 2$ entties, each taken from entity sets.

$$\{(e_1, e_2, \dots e_n) \,|\, e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

where $(e_1, e_2, \dots e_n)$ is a relationship.

2.1 Attributes

An ***attribute*** can also be property of a relationship set. For instance, the *depositer* relationship set between entity sets *customer* and *account* may have the attribute *access-date*. Relationshup sets that involve, two entity sets are ***binary*** (degree two). Relationship sets may involve more than two entity sets. Relationships between more than two entity sets are rare (i.e. most are binary).

An ***entity*** is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set. ***Domain*** is the set of permitted values for each attribute. The **types of attributes** include the following:

- *simple* (atomic) and *composite* attributes
- *single-valued* and *multi-valued* attributes

- *derived* attributes (can be computed from other attributes)

When attributes are *simple* and *single-valued*, then we say that the data is in ***First Normal Form***.

2.2 Keys

A ***super key*** of an entity set is a set of one or more attributes whose values uniquely determine each entity. Once you have defined a *super key*, you can add attributes and it is still considered a *super key*. A ***candidate key*** of an entity set is a minimal super key (e.g. *cusomer_id* is a candidate key of *customer*). Candidate keys *only* contain the necessary attributes to make something unique. Although several candidate keys may exist, one of the candidate keys is selected to be the ***primary key***.

The combination of primary keys of the participating entity sets forms a super key of relationship set. This means a pair of entity sets can have at most one relationship for each access.

A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a ***foreign key***. Foreign key is the primary key of the parent table. Only values occurring in the primary key attribute of the **referenced relation** may occur in the foreign key attribute of the **referencing relation**.

2.3 E-R Diagrams

Rectangles represent entity sets. Diamonds represent relationship sets. Lines link attributes to entity sets and entity sets to relationship sets. Ellipses represent attributes: double ellipses represent multivalued attributes while dashed ellipses denote derived attributes. Underline indicates primary key attributes.

3 Relational Model

Formally, given sets $D_1, D_2, \ldots D_n$ a ***relation*** r is a subset of

$$D_1 \times D_2 \times \cdots \times D_n$$

Thus, a relation is a set of n-uples $(a_1, a_2, \ldots a_n)$ where each $a_i \in D_i$.

Each attribute of a relation has a name. The set of allowed valued for each attribtue is called the ***domain*** of the attribute. Attribute values are (normally) required to be ***atomic***; that is, indivisible. Domain is said to be atomic if all its members are atomic. The special value *null* is a member of every domain. The null value causes complications in the definition of many operations.

$A_1, A_2, \ldots A_n$ are attributes. $R = (A_1, A_2, \ldots, A_n)$ is a ***relation schema***. $r(R)$ denotes a relation r on the relation schema R.

The current values (***relation instance***) of a relation is specified by a table. An element t of r is a *tuple*, represented by a row in a table. Relations are unordered and thus the roder of tuples is irrelevant (tuples may be stored in an arbitrary order).

A ***database*** consists of multiple relations. Information about an enterprise is broken up into parts, with each relation storing one part of the information.
The two types of database management systems are OLTP (***Online Transactional Processing***) and OLAP (***Online Analytical Processing***). OLTP databases are update/change oriented (*most* common databases are this category): write oriented. OLAP databases are for viewing and analyzing the data: read oriented. In OLTP, a good table is a table about *one thing only*: ***third normal form***. For example, a good OLTP table will have a table containing information about customers only and customers only. OLTP tables are normalized, but OLAP tables must be denormalized and this can be achieved by using ***join operations***.

3.1 Query Languages

A ***query language*** is a language in which users request information from the database. ***Pure languages*** are relational algebra, tuple relational calculus, or domain relational calculus. *Pure langauges* form the underlying basis of query languages that poeple use.

3.2 Relational Algebra

Relational algebra is a procedural language with six basic operators:

- select: σ — **WHERE** in SQL
- project: $\prod$ — **SELECT** in SQL
- union: $\cup$ — **UNION** in SQL
- set difference: $-$ — **EXCEPT** in SQL
- cartesian product: $\times$ — **, (comma)** in SQL
- rename: ρ — **AS** in SQL

The operators take one or two relations as inputs and produce a new relation as a result.

Example 3.1 Translate the following relational algebra to SQL...

- $\sigma_{A=B \ \hat{} \ D>5}(r)$
  `SELECT * FROM r WHERE A=B and D>5`

- $\prod_{A,C}(r)$
  ```
  SELECT A,C FROM r
  ```
- $r \cup s$
  ```
  SELECT * FROM r
  UNION
  SELECT * FROM s
  ```
- $r - S$
  ```
  SELECT * FROM r
  EXCEPT
  SELECT * FROM s
  ```

When writing a ***SQL query*** always pull all your data togther into a single view. This means that you will start with a `FROM` clause. In OLTP, all of the tables are denormalized into a single table so that when making queries, you can have a simple `FROM` clause where the data is already in a single view. ***Join*** is a more selective version of ***cartesian product***. **Join** is essentially a cartesian product followed by a select operation. Both **join** and **cartesian product** are $\mathcal{O}(n^2)$, but **join** will very likely run faster in most cases.

Definition 3.1 (Rename Operation) ALlows us to name, and therefore to refer to, the results of relational algebra expressions. For example,

$$\rho_x(E)$$

returns the expressio E under the name X. If a relational algebra expression E has an arit n, then

$$\rho_{x(A_1,A_2,\ldots,A_N)}(E)$$

returns the result fo expression E under the name X, and with the attributes renamed to $A_1, A_2, \ldots, A_n$.

How to write a query (SQL / rel. algebra) based on English Description...

1. Identify *data elements*: attributes and columns (e.g. `cusomername`)
2. Identify the *sources* of the data elements in **Step 1** (e.g. `customer`)
3. Identify some *meaningful* relationships between/among the elements in **Step 2** (join/cartesian product followed by selection)

Example 3.2 (Query Translation Practice) Find all loans of over $ 1200.

$$\sigma_{\text{amount }>1200}(loan)$$

```
SELECT *
FROM loan
```

```
WHERE amt > 1200.
```

Find the loan number for each loan of an amount greater than $ 1200.

$$\prod_{\text{loan\_number}} (\sigma_{\text{amount}>1200}(loan))$$

```
SELECT loan_numb
FROM loan
WHERE amt > 1200.
```

Pushing down of selection operation is the idea where we perform selection as early as possible to attempt to make the table resulting from a join smaller. This can be used to help optimize SQL queries. Try to minimize the size of the table (reduce the number of rows and columns) that is generated with a join. Push down of the selection operation reduces the number of rows. Projection operation reduces the number of columns. ***Heuristic optimization*** is this process and is done by the DBMS.

Definition 3.2 (Set-Intersection Operation ($r \cap s$)) Set intersection is defined as $r \cap s = \{t \mid t \in r \text{ and } t \in s\}$. Where we assume that r,s have the same *arity* and attributes of r and s are compatible. Note that $r \cap s = r - (r - s)$.

Definition 3.3 (Natural-Join Operation ($r \bowtie s$)) Let r and s be relations on schemas R and S repsectively. Then $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:

- Consider each pair of tuples t_r and r and t_s from s.
- If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result where
  - t has the same value as t_r on r
  - t has the same value as t_s on s

Example 3.3 (Natural-Join Example) Let $R = (A, B, C, D)$ and $S = (E, B, D)$.
Result schema $= (A, B, C, D, E)$
$r \bowtie s$ is defined as $\prod_{r.A,r.B,r.C,r.D,r.E}(\sigma_{r.B=s.B \wedge r.D=s.D}(r \times s))$

```
SELECT *
FROM r natural join s
```

or

```
SELECT y.A, y.B, y.C, y.D, S.E
FROM Y,S
WHERE Y.B = S.B
and Y.D = S.D
```

Definition 3.4 (Division Operation ($r \mid s$)) This operation is suited to queries that include the phrase "for all". Let r and s be relations on schemas R and S respectively where

- $R = (A_1, \ldots, A_m, B_1, \ldots B_n)$
- $S = (B_1, \ldots, B_n)$

The result of $r \mid s$ is a relation on schema $R - S = (A_1, \ldots, A_m)$

$$r \mid s = \{t \mid t \in \prod_{R-S}(r) \wedge \forall u \in s(tu \in r)\}$$

where tu means the concatenation of tuples t and u to produce a single tuple.

Remark Let $q = r \mid s$, then q is the largest relatin satisfying $q \times s \subseteq r$.

Definition 3.5 (Generalized Projection) Extends the prjection operation by allowing arithmetic functions to be used in the projection list

$$\prod_{F_1, F_2, \ldots, F_n}(E)$$

where E is any relational-algebra expression. Each of $F_1, F_2, \ldots F_n$ are arithmetic expressions involving constants and attributes in the schema of E.

An ***aggregation function*** takes a collection of values and returns a single value as a result. For example, average value, minimum value, maximum value, sum of values, and number of values are all aggregation functions.

Definition 3.6 (Aggregate Operation)

$$_{G_1, G_2, \ldots, G_n}g_{F_1(A_1), F_2(A_2, \ldots, F_n(A_n))}(E)$$

where E is any relational-algebra expression.

- $G_1, G_2, \ldots, G_n$ is a list of attributes on which to group (can be empty)
- Each F_i is an aggregate function
- Each A_i is an attribute name

Example 3.4 (Group By Examples)

- $g_{\text{sum}(c)}(r)$

```
SELECT sum(c)
FROM r
```

- $_{branch\_name}g_{\text{sum}(balance)}(account)$

```
SELECT branch_name, sum(balance)
FROM account
Group By branch_name
```

The result of an aggregation does not have a name, but we can use the rename operation to give it a name. For convenience, we permit renaming as a part of aggregate operation:

$$_{branch\_name}\mathcal{G}_{\text{sum}(balance) \text{ as } sum_balance}(account)$$

Definition 3.7 (Outer Join) an extension of the join operation that avoids loss of information. ***Outer Join*** computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join. This uses *null* values which signify that the value is unknown or does not exist. All comparisons involving *null* are false by definition.

Within this, there is left and right outer join. In ***left outer join***, the results include those from inner join and the entities from the first entity set that are not included in result of an inner join. Values that are missing (the reason they are not included in the inner join) will have null-values. There is also ***full outer join*** which is the union of both left and right inner join.

It is possible for tuples to have a null value, denoted by *null*, for some of their attributes. *null* signified an unknown value or that a value does not exist. The result of any arithmetic expression involving *null* is *null*. Aggregate functions simply ignore null values. For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same.

Comparisons with null values return the special truth value: ***unknown***. There are special rules involving the *unknown* truth value:

- OR

$$\begin{aligned}(unknown \text{ or } true) &= true,\\ (unknown \text{ or } false) &= unknown\\ (unknown \text{ or } unknown) &= unknown\end{aligned}$$

- AND

$$\begin{aligned}(true \text{ and } unknown) &= unknown,\\ (false \text{ and } unknown) &= false\\ (unknown \text{ and } unknown) &= unknown\end{aligned}$$

- (**not** *unknown*) = *unknown*

Definition 3.8 (Deletion) A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the data base. Can only delete whole tuples; cannot delete values on only particular values. A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.

Definition 3.9 (Insertion) To insert data into a relation, we either:

- specify a tuple to be inserted
- write a query whose result is a set of tuples to be inserted

In relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$

where r is a relation and E is a relational algebra expression. The insertion of a single tuple is expressed by letting E be a constant relation containing one tuple.

Definition 3.10 (Updating) A mechanism to change a value in a tuple without charting all values in the tuple. Use the generalized projection operator to do this task

$$r \leftarrow \prod_{F_1, F_2, \ldots, F_l} (r)$$

Each F_i is either

- The l^{th} attribute of r, if the l^{th} attribute is not updated, or,
- if the attribute to be updated F_i is an expression, involving only constants and the attributes of r, which gives the new value for the attribute