CENG315

Information Managment Lecture Notes

Ege Özkan

February 1, 2021

Contents

Editor's Note

Within the book, there are some instances where a text is presented within square brackets. These are the Editor's Commentary, these may be thoughts that popped into my mind while taking notes, additional information or clarifications. Now, nothing in the book is guaranteed to be correct, but most of it is at least information I transcribed from an expert, so Editor‘s Notes are extra likely to be wrong, enjoy.

Chapter 1

Introduction - October 15, 2020

1.1 Databases and Database Systems

Databases hold data. Database systems are software systems that manages the records in a database. There are five fundemental requirements for a database system.

- Database systems must be persistent, data must be storable and remain for the future.
- Databases must be able to handle getting large.
- Databases should be sharable, multiple users should be able to reach it at the same time.
- Databases must be kept accurate.
- Databases must be usable.

1.1.1 Record Storage

Databases can be made persistent in different ways.

Storing database records in text files

- Simplest approach.
- One file per record type.
- Each record could be a line of text, with its values seperated by tabs.

1	joe	2020
2	amy	2013
3	lee	2000

Its advantages are the database system has to do very little, and a use could easily examine and modify the files with a text, but it is slow. *(!*)*

1.1.2 Data Models and Schema

Data models are different ways to express connections between records while Schemas are the implementations of these methdos for a specific database.

File-system v. Relational

In the file system modal, each record type has a file, with one record per line, programs that read and write to the file is responsible for understanding this. In the relational data model, each record type has its **table** and each record has **fields** for each value. User access to the database happens via this record and field model and records that fit certain conditions can be quarried.

These models are at different levels of abstraction, relational model is a **conceptual**

model, since there is no need to know **how** schemas are specified and implemented, the conceptual schema describes what the data *is*. Whereas the file-system is called a **physical model**, physical schemas say how the data is *implemented*.

Physical Data Independence

A conceptial schema is certainly nicer to use than a physical scheme. Operations on a conceptual schema is implemented by the database schema. Database system has a **database catalog** that contains descriptions of the physical and conceptual schemas. Givven an SQL querry, the database system translates the conceptual abstraction to the physical one and interract with it on the users behalf. If the user does not have to deal with the physical level, this is called the Physical Data Independence.

It is easy to use, quaries are optimized automatically and it is isolated from changes to the physical schema.

Logical Data Independence

The set of tables personalized for a particular user is called the user's **external schema**. If users can be given their own external schema in a database system, it is told that this Database System supports Logical Data Independence.

It has three benefits:

- Each users gets a customized external schema, they see only the information they need.
- The user is isolated from changes to conceptual schema.
- It is safer.

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
COURSE(CId, Title, DeptID)
SECTION(SectId, CourseId, Prof, Year)
DEPARTMENT(DId, Name)
```

Figure 1.1: An example schema

1.2 Relational Databases

The relational modal is a conceptual model since its schemas do not depend on the pyhsical level.

1.2.1 Tables

The database is organized into **tables**, which contain zero or more **records** (ie: table rows), and at least one **fields** (ie: the columns of the table.) Each record has a value for each field, and all fields has a specific **type**. Often, when discussing tables, the type information ignored.

Null Values

A `null` value denotes a value that *does not exist* or is *unknown*. It occur if the data collection is incomplete or if data has not arrived yet.

1.2.2 Superkeys and Keys

In the relational model, the access to data is not handled by indices. Instead, a record must be referenced by specifying field values. Since not all values are guranteed to be unique for all users, a unique identifier field is called a **superkey** to distinguish it. Adding a field to a superkey, will generate another superket. A **key** is a superkey with the property that no subset of its fields is a super key.

Primary Keys

In the Schema at Figure 1.1's, `STUDENT` table `SId` is a key. Whereas in `SECTION` there may be multiple keys if each professor teaches only one class. Therefore, since a table may have multiple keys, a key is chosen as a **Primary Key**, whose values *should never be null*, and who is used to refeer to each record.

For instance, in Figure 1.1, `STUDENT` table, `SId` can be the primary key. This is no coincidance, IDs are most times fit to be primary keys.

Foreign Keys

The information in a database is split among tables, these are not isolated from each other, a **foreign key** is a field (or fields) of one table which corresponds to the primary key of another table. For instance, in Schema at Figure 1.1, `CourseId` of the `SECTION` table is a foreign key.

Foreign Keys can be used to create logical connections between different types of records. In the Schema at Figure 1.1, `CourseId` of the `SECTION` table creates a logical connection between the `SECTION` table and `COURSE` table, since the objects these represent in real life, Sections and Courses are bound by a logical connection as well. (Each section is a section of a course).

Foreign Keys and Referential Integrity

The specification of a foreign key asserts **referential integrity**. Which requires each non-null foreign key value to be the key value of some record. Database system must ensure that if the primary keys of a table is modified in some ways, the foreign keys in other tables refeering to primary keys must also be updated accordingly, or set to `null` in worst case scenerio.

1.2.3 Constraints

A **constraint** describes the allowable states that fields can have in a table. There are four important kinds of constraints. **Null Value Constraints** limit fields to not have `null` values. **Key constraints** specify that two records cannot have the same value. **Referential integrity constraints** specify referential integrity, finally **integrity constraints**.

Integrity constraints

These constraints encodes *business rules.* They can detect bad data entry and can enfore the *rules* of the organization. They may apply to tables, individual records or the entire database.

1.2.4 Table Specification in SQL

Listing 1.1: the SQL specification of the STUDENT table

```
create table STUDENT (
        SId int not null,
        SName varchar(10) not null,
        MajorId int,
        GradYear int,

        primary key (SId),
        foreign key (MajorId) references DEPT
                on update cascade
                on delete set null,
        check (SId > 0),
        check (GradYear >= 1863)
)
```

In Listing 1.1 we can see constraints and fields. The action specified with the `on delete` and `on update` keywords can be one of the following:

Cascade causes the same query to apply to each foreign key record.

Set null causes the foreign key values to be set to null.

Set default causes the foreign key values to be set to their default value.

No action causes query to be rejectd if there exists and effected value with the foreign key.

Part I

Theoretical Foundations & Database Design

Chapter 2

Relational Algebra - October 22, 2020

ID	Name	Dept. Name	Salary
22222	Einstein	Physics	95000
12212	Tesla	Physics	4354

Table 2.1: Instructors.

2.1 Structure of Relational Databases

Databases are structured with atrributes and values as tuples corresponding to those attributes.

2.1.1 Attributes

The domain of teh attribute is a set of allowed values. Attribute values are normally required to be **atomic**.

The **null** value is a special value that signifes that the value is unknown, or does not exist, it is a member of every domain. However, it causes complications.

2.1.2 Schema vs Instance

A database schema is the logical structure of the database. `instructor(ID, name, dept_name, salary)`. A database instance is the snapshot of the database in a given time.

Using common attributes in relation schemas is one of *(!*)*. There is also need for a *(!*)*.

2.1.3 Keys

A **superkey** is a set of one or more attributes that allow us to identify uniquely a tuple in relation. Let $L \subset R$, superkey K is a **candidate key** if K is minimal. One of the candidate keys is selected to be **primary key**, they should be chosen such that its attribute values are never or very rarely changed.

Foreign key constraint states that value in one relation must appear in another. **Referencing relation** is the relation that refeers to another and **Referenced relation** is the reference that is being referenced.

2.2 Relational Query Languages

A **query language** is a language in which a user requests infromation from the database. **Relational algebra** provides a set of operations that take one or more relations as input and return a relation as an output.

2.3 Operations of Relational Algebra

Relational algebra provides operations that take relations as input and returns relations as output.

2.3.1 Select Operation

Select operator selects $\sigma_p(r)$ (or `select(r, p)` to denote the selection of rows (horizontal selection) to denote selection on relation r with respect to predicate p.

For instance, $\sigma_{A=B \wedge D>5}(r)$ would select tuples of relation r, such that its A and B attributes are equal and values of D attribute is greater than 5

On the Table 6.1, $\sigma_{\text{dept\_name="Physics"}(\text{instructor})}$ would return a tuple of instructors whose department is Physics.

Selection predicate can take comparasions using $=, \neq, >, \geq, <, \leq$ and multiple predicates can be combined using **connectives.** $\wedge, \vee$ and $\neg$.

For instance on the department table with schema `department(dept_name, building, budget)`, $\sigma_{\text{dept\_name}=\texttt{building}}(\text{department})$ would return departments whose names equal to their building's name.

2.3.2 Project Operation

An unary operation that returns its argument relation with certain attributes left out. $\Pi_{A_1,A_2,A_3,\ldots,A_k}(r)$ or `project(r,` $A_1, A_2, A_3, \ldots, A_k$`)` where A_n are attribute names and r is a relation.

In essance project operation returns tuples with only the values whose attributes are listed in the operation.

2.3.3 Composition of Relational Operations

Since the result of a relational operation are itself a relations, operations can be given as input to other operations, ie: they can be composed together into a **relational-algebra expression**, finding the names ofa ll instructors in the physics department can be done by:

$$\Pi_{\texttt{name}}(\sigma_{\texttt{dept\_name="Physics"}}(\texttt{instructor})) \tag{2.1}$$

2.3.4 Cartesian Product Operation

Composes two relations together to a single product, `instructor` $\times$ `teaches` relation, where `instructor(id, name, dept_name, salary)` and `teaches(id, course_id, year` results in the relation `instructor`$\times$`teaches(instructor.id name, teaches.id, course_id, year)`

However, as one can see, common attributes are not joined, therefore the cartessian product may not (and most likely will not) result in logical results.

When to attribute names are the same, they can be distinguished by attaching the name of the relation prior to the attribute name.

2.3.5 Join Operation

To avoid the mistake of illogical results, one can write:

$$\sigma_{\texttt{instructor.id}=\texttt{teaches.id}}(\texttt{instructor}\times\texttt{teaches}). \quad (2.2)$$

The join operator is the equivalent of this expression. **Natural join** operation is denoted by $\bowtie$ Outputs of the rows from the two input relations that have the same value on all atributes that have the same name is joined.

Consider relations r and s, let θ be a predicate on attributes in the schema $r \cup s$. The join operation $r \bowtie_\theta s$ is defined as $r \bowtie_\theta s = \sigma_\theta(r \times s)$

Such as $\texttt{teaches} \bowtie_{\texttt{teaches.id}=\texttt{instructor.id}} (\texttt{instructor})$ is equivalent to $\sigma_{\texttt{instructor.id}=\texttt{teaches.id}}(\texttt{instructor} \times \texttt{teaches})$

2.3.6 Union Operation

The union operation $r \cup s$ combines two relations as long as they have the same **arity** (number of attributes) and the attribute domains are compatible. (Same indexed attributes have the same domain.)

The expression $\pi_{\texttt{course\_id}}(\sigma_{\texttt{semester}=\texttt{"Fall"}\wedge\texttt{year}=2017}(section) \cup \pi_{\texttt{course\_id}}(\sigma_{\texttt{semester}=\texttt{"Spring"}\wedge\texttt{year}=2018}(section)$ on the relation `section` with schema `section(course_id, sec_id, semester, year, building, room, number, time_slot_id)` wil select `course_id` row of the course that are though on Fall 2017 **or** Fall 2018.

2.3.7 Set Intersection Operation

Set intersection $s \cap r$ works exactly the same (and have the same assumptions.), but instead of working like *or*, it works like **and**.

2.3.8 Set Difference Operation

Set differnce $s - r$ works similar to intersection and union, but it selects those tuples that are on the first relation and **not** on the second relation.

2.3.9 Rename Operation

Given the relational algebra expression E, the expression $\rho_x(E)$ returns the expression E under the name X.

It can also return an output whose attribute names are changed when they are listed $\rho_{x\{A_1,A_2,\ldots,A_n\}}(r)$.

2.3.10 Assignment Operation

The assignment operation $\leftarrow$ works like assignment in a programming language, relation algebra expressions can be assigned to temporary relation variables.

$$\begin{aligned} \texttt{Physics} &\leftarrow \sigma_{\texttt{dept\_name}=\texttt{"Physics"}}(\texttt{instructor}) \\ \texttt{Musics} &\leftarrow \sigma_{\texttt{dept\_name}=\texttt{"Musics"}}(\texttt{instructor}) \\ &\texttt{Musics} \cup \texttt{Physics} \end{aligned}$$

2.3.11 Equivalent Queries

Since there is more than one way to write a query in relational algebra, queries that are not identical may be **equivalent**, they give the same result on any database.

Alternative Notation

On a related note, queries can be written with the alternative notation shown. For instance, `select(p, r)` instead of $\sigma_p(r)$

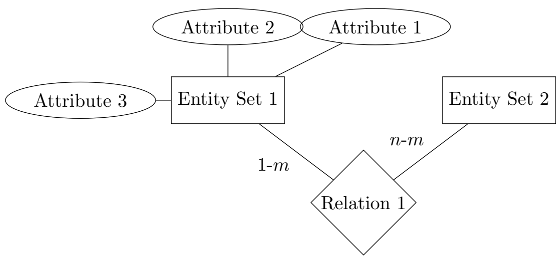
Chapter 3

Database Design - November 12, 2020

3.1 Design Phases

(!)*, First the database needs must be understood, after the database is designed conceptually. The final design is done in two phases, logical and phusical design, logical design is deciding on the schema, and phiscal design is choosing the implementation.

3.2 Entity Relationship Model



Models and enterprise as a collection of **entities** and **relationships**. It is also called the ER diagram. It consists of three basic structures, **entity sets**, **relationship sets** and

Entity a thing or an object in the enterprise that is distinguishable from other objects, described by a set of *attributes*.

Relationship An association among several entities.

Since entities are represented by a set of attributes, a subset of the attributes form a **primary key** of the entity set, uniquelly identifying each member of the sets.

Entity sets are represented in a similar fashion to UML class diagrams, with its attributes being the variables of the class. In the alternative notation, they are represented as rectangles, with its attributes (shown with elipses) tied to them. This alternative notation is shown in the picture at the start of subsection 3.2 (From https://texample.net/tikz/examples/er-diagram/)

Complex Attributes

Attributes can be grouped as simple and composite attributes, composite attributes can be divided into subparts. They may also be grouped as single-valued and multi-valued attributes, multivalued attributes may take more than one value at one time. Finally, a **derived** attribute is an attribute that can be derived from other attributes.

Composite attributes are shown as nested values in the UML-like notation. In the alternative notation, they are arguments bound to other arguments.

Relationship Sets

A relationship set is a mathametical relationship between two entity sets. Relationship sets are represented using diamonds between two entity sets. **Roles** are used to differ between two occurances of the same entity set in different rules, for instance, a course may be a prerequisette and the course name itself.

Relationship sets have **Degree**s, binary relationships involve two entity sets, which are most of them. But their degree may be higher.

Cardinality of a relationship refeers to the number of entities connected in each entity set by a relationship, a one-to-one relationship cocurs when the cardinality of a relationship is **contrainted** to at most one. The side(s) that is constrainted to at most one of themselves has a arrow head pointed at them in their connection to the relationship.

Cardinality constraints of relationships may be one-to-one, many-to-one, one-to-many or many-to-many.

The **Participation** is denoted with a double line or a single line, the **total participation**, indicated by a dobule line, means that every entity in the entity set participates in at least one relationship in the relationship setü while **partial participation** means that some entities may not participate in a relationship in the relationship set.

A line may have a text on it, of the form $l..h$, where l is the minimum and h is the maximum cardinality. If an asterix $(*)$ is given for the maximum, that implies that there is no limit. A minimum value of 1 implies maximum cardinality.

In Ternary and above relationship sets, only one arrow is allowed to denote cardinality.

Primary Key for Entity Sets

By definition, individual entites are distinct, no entities in an entity set can have all their attributes the same, at least one attribute must differ, the primary key is the one attribute that distinguishes between all entities.

Primary Key for Relationship Sets

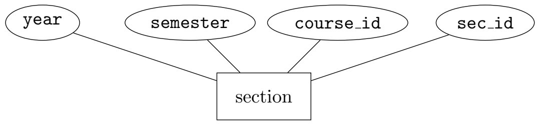
To distinguish among the various relationships of a relationship set, individual priamry keys of the entities in the relationship set denote the primary key for a relationship set is denoted by the union of primary keys of its entity sets.

The implication here is that, depending on the cardinality, for one-to-many relationships, the many side's keys are the minimal superkey and therefore , for many-to-many, the union of the keys take this role and for one-to-one, any one of the attributes may be chosen. *?**

In conclusion, the idea is to chose the primary key from the side that repeats the least. The idea is *how we can represent a connection using the least amount of keys?* We choose the many side, because, in one-to-many or many-to-one, because each many item will have **at most** one corresponding one item. On the other side, the choice literally does not matter for one-to-one, and on the other side of this, we have many-to-many where we need both sides to adequitelly identify a relationship, since everyone can have multiple onnections.

Weak Entity Sets

A weak entity is an entity that cannot be uniquely identified by its attributes alone.



A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**, the part of the primary key of this entity set is the primary key of the entity set it depends on as a **discriminator**. An entity set that is not a weak entity set is termed a **strong entity set**. Every weak entity must have a entity set it **existently depends** on.

In ER diagrams, a weak entity set is depicted via a double rectangle. Its discrimanotrs are underlined.

[Weak entity sets are simply entity sets that are dependant on other entity sets to exist.]

3.3 Reducing ER Diagrams to Relational Schemas

As a first approximation:

1. Turn each entity set into a relation with the same set of attributes.
2. Replace a relationship set by a relation whose attributes are the keys for the connected entity sets (and any descriptive attributes of the relationship sets).

Weak entity sets change this somewhat.

3.3.1 Representing entity sets

.

A strong entity set reduces to a schema with the same attributed, ie: A student entity set with attributes ID, name, and tot_cred becomes *student(<u>ID</u>, name, tot_cred)* A weak entity set becomes a table that includes a column for the rpimary key of the identifying strong entity set.

Composite Attributes are represented by dividing each composite part to normal attributes. [Composite attributes reduce to their subattributes.] Derived attributes are omitted completely.

Multivalued attributes map to brand new schemas, whose members are multiple values these attributes take. For instance, a student with a multivalued key phone number, maps to a phone number schema, whose members are all student's phone numbers, where a student may have multiple of them.

Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the many side, containing the priamry of the one side. In one-to-one relationships, any side can be chosen as the many, albait, if the participation is not total, `NULL` values will occur.

Relations between weak entity sets and their corresponding strong entity sets are omitted as well, since they become redundant.

3.3.2 Specialization

Specialization is special entity set structure where a (weak) entity set is used as a subclass-like structure of another entity set. It can be overlapping (entity may occur in multiple specializations) or it may be disjoint, where this cannot occur.

Specializations are represented in schemas by creating a schema for the higher level entity, and then forming another schema for each lower-level entity set, include the primary key of the higher level entity set. Another method is to form a sechema for each entity set and include all local and inherited values.

The drawback in the first is more queries being spent to look for a single entities records, and the for the second method more space being taken redunantly.

Completeness Constraint

Completeness constraint state wheter or not each entity in the higher level set must belong to a lower level entity set. Total Completeness means that it must, and Partial means it is not a must. The partial generalization is the default, when denoting a total generalization, a dashed line is drawn from the arrow, and on it the word *total* is written.

3.4 Design Problems

There are certain design problems that may occur while designing a database system.

Entities vs Attributes

Certain attributes may be converted to entities on their own right if one wishes to store additional information about a specific attribute.

Entities vs Relationship

A guidline in deciding wheter or not something is an entity or a relationship is by asking if it is an *actions*. Actions that occur between two of entities are relationships. Arguments directly related to relationships must become relationship attributes.

Redunantant Atttributes

Avoid repeating information. ER Diagrams *are not* schemas, foreign keys are not needed to be shown if there is a relationship between them instead.

Chapter 4

Database Theory for Relational Databases - November 19, 2020

4.1 Features of Good Relational Design

In a database we talked about the previous section, `instructor(ID, name, dept_name, salary)` and `department(dept_name, building, budget)` was two different schemas in this database. If we were to combine these two schemas into a relation, there would be a repetition of information, since instructors of the same departments will write budget data more than once. It also introduces the need to use `null` values, (if one adds a new department with no instructors.)

This is because, for this example, keeping two different tables is good. But in some cases, for instance `employee(ID, name, street, city, salary)` schema, decomposed into `employee1(ID, name)` and `employee2(name, street, city, salary)`, it might be impossible to reconstruct the original employee relation if more than one employee with the same name exists. These sorts of decompositions are called **loosy decomposition**. While a decomposition that can be reconstructed back to its original form is a **loseless composition**.

In conclusion, for the decomposition of a relation of R to R_1 and R_2, if $R_1 \bowtie R_2 = R$ it is lossless, otherwise, lossy.

4.1.1 Functional Dependencies

Suppose a schema of `Student(SSN, SName, address, HScode, HSname, HScity, GPA, priority)` suppose that the `priority` is determined by the `GPA`. If `GPA> 3.8, priority = 1,` `3.3 < GPA <= 3.8, priority = 2` and `GPA <= 3.3, priority = 3`. It can be concluded that *two tuples with the same GPA have the same priority.*

$\forall t, u \in$ `student` : t.`GPA` $=$ u.`GPA` $\Rightarrow$ t.`priority` $=$ u.`priority` Then, it is said that `GPA` $\rightarrow$ `priority` (`priority` is functionally dependent on `GPA`).

In general:

$$\text{if } \forall t, u \in R, t[A_1, A_2, \ldots, A_n] = u[A_1, A_2, \ldots, A_n] \rightarrow t[B_1, B_2, B_m] = u[B_1, B_2, \ldots, B_m] \text{ then } A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m \quad (4.1)$$

$X \rightarrow Y$ is an assertion about a relation R. By convention, X, Y, Z represets sets of attributes, A, B, C represents sinlge attributes, and by convention $\{A, B, C\}$ may be written as ABC.

4.1.2 Rules for Functional Dependencies

Splitting Right Sides of FDs

if $X \to A_1A_2 \dots A_n$ holds for R exactly when each of $X \to A_1, X \to A_2, \dots X \to A_n$ hold for R, in general:

$$A \to BC \Rightarrow A \to B \wedge A \to C \tag{4.2}$$

Combining Rule

The inverse of the splitting rule.

$$A \to B \wedge A \Rightarrow C \Rightarrow A \to BC \tag{4.3}$$

Triviality

$X \to Y$ is a nontrivial functional dependency if $Y \nsubseteq X$ otherwise, it is a trivial functional dependency. Moreover, if $X \to Y$ is a **trivial functional dependency** then $X \to X \cup Y$ and also $X \to X \cap Y$.

Transitivity of FDs

If $X \to Y$ and $Y \to Z$, then $X \to Z$.

Closure of Attributes

The set of **all** functional dependencies logically implied by X is the closure of X, given relation, a set of FDs, a set of attributes X, find Y suvh that $X \to Y$.

The algorithm used for this purpose, starts with a set of attributes X, and a set of FDs of relation R. The closure X^+:

- If necessary, split the FDs of the R, so each FD in R has a single attribute on the right.
- Start with the set itself.
- Repeat until there is no change. If $X \to Y$ and X is in the set, then add Y to the set.

For instance, given FDs $A \to B$ and $B \to D$, closure of A evolves as:

- $A^+ = \{A\}$
- $A^+ = \{A, B\}$
- $A^+ = \{A, B, D\}$

4.1.3 Keys of Relations

K is a **superkey** if they functionally determnie all other attributes. In other words, if $K^+ = X$ where X is all attributes of R, then K is a **superkey**.

Consider in scheme `Customers(name, addr, drinksLiked, manf, favDrink` if `name` $\to$ `addr, favDrink` and `drinksLiked` $\to$ `manf`, here $\{$`name, drinksLiked`$\}$ is the superkey since its closure is all the attributes of the relation and also since its closure is all attrbiutes.

A key is a superkey if none of its strict subsets is also a superkey. Also consider that all of the supersets of a superkey is a superkey itself.

4.1.4 Projecting Functional Dependencies

Normalization refeers to the process where one breaks a relatino schema into two or more schemas, imagine a relation R of attributes $ABCD$, with FDs $AB \to C, C \to D$ and

$D \to A$. If one decomposes R into ABC, AD not only will $AB \to C$ will hold, but also $C \to A$.

Start with given FDs and find all *nontrivial* FDs that follow from the given FDs, then restrict to those FDs that involve only attributes of the projected schema.

With inputs of two relationships R, R_1 where R_1 is decomposed from R, a set of FDs that hold in R.

1. Let T be the eventual output set of FDs, initally, it is empty.
2. For each set of attributes X that is a subset of atrributes of R_1, compute X^+
3. Add to T all nontrivial FDs $X \to A$ such that $A \in X^+$ and an attribute of R_1.
4. However, drop from T, $XY \to A$ whenever we discover $X \to A$, because $XY \to A$ follows from $X \to A$ in any projection.
5. Finally use these FDs.

There are a few tricks here, one does not need to compute the empty set an its closure, and if X^+ determines all attributes, then so does its supersets.

Example

For instance to $R(ABCD)$ FDs $A \to B, B \to C, C \to D$ project onto $R_1(ACD)$, we start from singletons and move onto biggers subsets.

- $A^+ = ABCD$, thus $A \to C$ and $A \to D$ holds in R_1, note that $A \to B$ is true in R but makes no sense in R_1. Since A^+ includes all attributes of R_1, there is no need to consider the supersets of A.
- $C^+ = CD$, thus $C \to D$ holds in R_1.
- $D^+ = D$ is trivial, and yields no nontrivial FDs.

Thus, FDs for R_1 are $A \to C$ and $C \to D$, and of course, $A \to D$ from the transitivity rule.

[So *that* is why they thought us predicate logic in Discrete Structures...]

4.2 Anomalies

Problems that arise in databases due to poor design are called **anomalies**.

Redundancy Repeated information in several tuples.

Update Anomalies When a change in one ypule leaves the same inforamtion unchanged in another tuple.

Deletion Anomalies Losing information when deleting.

Consider for the schema `Customers(name, addr, drinksLiked, manf, favDrink`, if a customer likes more than one drink, the `favDrink` and `addr` will repeat unncesserilly.

Moreover, this bad design is also open to update and deletion anomalies. Consider, if a customer changes their address, what if the programmer does not remember updating all tuples containing them; or if no one likes coke, one loses track of the fact that Coca-Cola manufactures Coke.

4.3 Boyce-Codd Normal Form

The goal of decomposition is to replace a relation (that exhibits some anomalies) with several relations that do not exhibit anomalies.

The condition under which the anomalies discussed can be guranteed not to exist is called Boyce-Codd Normal Form, or BCNF.

We say R is in BCNF if whenever $X \to Y$ is a nontrivial FD that holds in R, X is a superkey.

In `Customers(name, addr, drinksLiked, manf, favDrink`, FD's are `name` $\to$ `addr favDrink`, `drinksLiked` $\to$ `manf`. Since `name` is not a superkey but appears in a FD, it violates BCNF.

Now consider `Customers(name, manf, manfAddr)`, FDs are `name` $\to$ `manf` and `manf` $\to$ `manfAddr`, here the second FD violates BCNF.

We can replace an R that is not in BCNF with two schemas:

1. $R_1 = X^+$
2. $R_2 = R - (X^+ - X)$

And then projecting R's FDs onto R_1, R_2.

Example

For instance, in `Customers(name, addr, drinksLiked, manf, favDrink`, and `name` $\to$ `addr`, `name` $\to$ `favDrink` and `drinksLiked` $\to$ `manf`.

- Pick BCNF violation `name` $\to$ `addr`
- Close the left side: $\{\texttt{name}\}^+ = \{\texttt{name}, \texttt{addr}, \texttt{favDrink}\}$
- This yields two decomposed relations `Customers1(name, addr, favDrink)` and `Customers2(name, drinksLiked, manf)`.

Now, check if `Customers1` and `Customers2` is in BCNF.

- If we get the closures for Customers1, $\texttt{name}^+ = \{\texttt{name}, \texttt{addr}, \texttt{favDrink}\}$, $\texttt{addr}^+ = \{\texttt{addr}\}$, $\texttt{favDrink}^+ = \{\texttt{favDrink}\}$. Only relevant FD (non-trivial ones) is `name` $\to$ `addr` and `name` $\to$ `favDrink`, which is in BCNF since `name` is a superkey.
- But, Customer2, `Customers1(name, addr, favDrink)`**does** violate VCNF, since the only nontrivial fd is `drinksLiked` $\to$ `manf` but `drinksLiked` is not a superkey by itself.

We decompose Customers2 to `Customers2(drinksLiked, manf)` and `Customers4(name, drinksLiked).` The resulting decomposition of `Customers2(name, drinksLiked, manf)`is:

- `Customers1(`*`name`*`, addr, favDrink`, which telss us about customers
- `Customers3(`*`drinksLiked`*`, manf)` which tells us about drinks. `Customers4(`*`name`*`,` *`drinksLiked`*`)` which tells us about the relationship between customers and the drinks they like.

4.4 3rd Normal Form

There is a possibly, when, decomposing a relation, because not all FDs are carried onto all decomposed relations, it is possible that, although no relations violate their own FDs, the database as a whole may violate one of the original FDs.

The 3rd Normal Form (3NF) modifies BCNF to fix this issue. An attribute is called **prime** if it is a member of any key. $X \to A$ violates 3NF if and only if X is not a superkey, and also A is not a prime.

There are two important properties of a decomposition:

1. **Lossless Join** it should be possible to project the orignal relations onto the decomposed schema, and then rebuild them.

2. **Dependency Preservation** It should be possible to check in the projected relations wheter all FDs hold.

BCNF does not preserve dependencies all the time, 3NF is a weaker normal form that allows some redunancy but also gurantees dependency preservation. It also gurantees lossless join.

In 3NF, the need for null values may arise from time to time, and there is also a problem of repetition of information.

3NF Synthesis Algorithm

There is a need for **minimal basis** for FD, a minimal basis for FDs are:

1. Right sides are single attributes.

2. No FDs can be removed.

3. No attribute can be removed from a left side.

To get a minimal basis:

1. Split right sides.

2. Repeatedly try to remove an FD and see if the remaining FDs are equivalent to the original.

3. Repeatedly try to remove an attribute from a left side and see if the resulting FDs are equivalent to the original.

Then, we can create schemas by giving one relation for each FD in the minimal basis. Schema is the union of the left and right sides. And also, if none of them are a key, also the key for a relation.

In a relation $R = ABCD$, FDs are $A \to B$ and $A \to C$, and then AB and AC relations are decomposed from FDs, plus AD for a key.

Chapter 5

Design Theory for Relational Databases (Cont'd) - November 16, 2020

Imagine a system with the relation `Apply(SSN, cName, hobby)`, we have no Functional dependencies for this relation, the only key is the all attributes of the relation. The relation is in the BCNF.

Is this a good design to hold student collage applications? Imagine a database such as:

SSN	cName	Hobby
123	IYTE	tennis
123	IYTE	swimming
123	EGE	tennis
123	EGE	swimming

Table 5.1: Instructors.

Sweet Jesus, this is terrifying, look at this monstrosity, if a student with 4 hobbies applies to 5 collages, that would create 20 tuples alone! This is a terrible design.

5.1 Multivalued Dependency

a Multivalued Dependency (MVD) on R, denoted $X \rightarrow\rightarrow Y$ says that if two tuples of R agree on all attributes of X, then their components in Y may be swapped and the result will be two tuples that are also in the relation.

For instance, `SSN` $\rightarrow\rightarrow$ `cName`, we swap collage names wherever the `SSN` of th e student is the same, and the resulting tuples will also be in the relationship.

For instance in `Customers(name, addr, phones, drinksLiked)`, here `phones` and `drinksLiked` are independent, which will create redundant tuples, we can just say `name` $\rightarrow\rightarrow$ `phones` and `name` $\rightarrow\rightarrow$ `drinksLiked`.

Every FD is an MVD

Keep in mind that every Functional Dependency is a Multivalued Dependency as well. If $X \rightarrow Y$, then by definition $X \rightarrow\rightarrow Y$, since swapping Ys between two tuples that agree on X doesn't change the tuples.

Complementation

If $X \rightarrow\rightarrow Y$, and Z is all the other attributes $X \rightarrow\rightarrow Z$.

For instance, in the `Apply(SSN, cName, hobby)`, since `SSN` $\rightarrow\rightarrow$ `cName`, automatically `SSN` $\rightarrow\rightarrow$ `hobby`. [Since swapping `cName` values is equal to swapping `hobby` values in reverse.]

Splitting Doesn't Hold

Like the FDs, left side of an MVD cannot be generally split. **Unlike** FDs, the right side cannot be split either. If $A \rightarrow\rightarrow CD$, *does not necessarily mean* $A \rightarrow\rightarrow C$ and $A \rightarrow\rightarrow D$.

5.2 the Fourth Normal Form

The Separation of independent facts is what 4NF is about. The redundancy that comes from MVNs cannot be fixed with BCNF. 4NF treats MVDs as FDs while decomposition.

A relation is in 4NF if: whenever $X \rightarrow\rightarrow Y$ is a nontrivial MVD, then X is a superkey. Nontrivial MVD means that:

1. Y is not a subset of X.
2. X and Y are not, together, all the attributes.

Superkeys are still determined by FDs only.

Connection to BCNF.

Since every FD is an MVD, if R is in 4NF it is certainly BCNF, but R could be in BCNF and not in 4NF.

If $X \rightarrow\rightarrow Y$ is a 4NF violation for relation R, we can decompose R using the same technique for BCNF.

Algorithm.

Until all relations are in 4NF:

- Pick any R' with nontrivial $X \rightarrow\rightarrow Y$ that violates 4NF.
- Decompose R' into $R_1(X, Y) and R_2(X, \text{rest})$.
- Compute FDs and MVDs for R_1 and R_2.
- Compute keys for R_1 and R_2.

Example

In the `Apply(SSN, cName, hobby)`, MVDs are:

1. `SSN` $\rightarrow\rightarrow$ `cName`
2. `SSN` $\rightarrow\rightarrow$ `hobby`

The key is $\{$`SSN`, `cName`, `hobby`$\}$ and all dependencies violate 4NF.

The MVN (1) and (2) violates 4NF because `SSN` is not a superkey.

Decompose using `SSN` $\rightarrow\rightarrow$ `cName`:

1. `Apply1(`*`SSN, cName)`*, no MVDs or FDs, and in 4NF.
2. `Apply2(`*`SSN, hobby`*`)`, no MVDs or FDs, and in 4NF.

Part II

Structured Query Language

Chapter 6

SQL - November 26, 2020

This chapter is the start of an overview of the **Structured Query Language**, SQL. [A declarative language, semi-standardised in ISO/IEC 9075] More specifically, the dialect of SQL used in the Oracle Database Systems.

Most (if not all) SQL statements can be among all dialects, and even when one-to-one compatibility is unavailable

6.1 Statements

6.1.1 Select Statement

Used to guery tables:

```
SELECT * FROM dept;
SELECT deptno, loc FROM dept;
SELECT ename, sal, sal+300 FROM emp;
```

Here, `SELECT` is used to query the table denoted by `FROM`, `SELECT` statement is followed by the column names (seğerated by commas) that are wished to be retrieved.

Using the `*` sign will select all the columns.

Observe that by using arithmetic operations, we are able to view manipulated data as well. Keep in mind that `sal+300` **does not** change the table itself.

Operator Precedence

Operator precedence follows normal precedence rules, parenthesis maybe used to clarify precedence when need be.

Null Values

`NULL` values in are values that do not exist, they are not zero. They may also create problems in arithmetic operations, returning `NULL` themselves.

Column Aliases

```
SELECT ename as name FROM emp;
SELECT ename "Name" FROM emp;
```

`AS` keyword is optional, an alias may follow the column name itself. But without double quotations, the entire word will be capitalised.

Duplicate Rows

```
SELECT deptno FROM emp;
SELECT DISTINCT deptno FROM emp;
```

By default, queries will display the duplicate rows also. The, `DISTINCT` keyword can be used to get rid of these.

Limiting Rows

```sql
SELECT ename, job, deptno
FROM emp
WHERE job='CLERK';
```

```sql
SELECT ename, job, deptno
FROM emp
WHERE sal<=comm;
```

The `WHERE` clause is an optional clause that be used to filter rows. Observe that the `WHERE` clause can also be used with comparison operator. Do keep in mind that `NULL` values return `NULL` here too.

```sql
SELECT ename, job,
        deptno FROM emp WHERE
        sal BETWEEN 1000 and 2000;
SELECT ename FROM
        emp WHERE mgr
        IN(7902, 7566, 7788);
SELECT ename FROM
        emp WHERE ename
        LIKE '_A%';
SELECT  ename, mgr
        FROM empt WHERE
        mgr IS NULL
```

	Function
`BETWEEN ... AND`	Return True values between two values.
`IN(LIST)`	Return True if values in a list.
`LIKE`	Pattern matching, % denotes zero or many, _ denotes one character.
`IS NULL`	Returns True if value is `NULL`

Table 6.1: Other Comparison Operators

Pattern matching characters can be combined, in the example above, names of the employees whose name start with a single character, than an M, and then one or more characters will return.

Furthermore, logical operators `AND`, `OR` and `NOT` is defined in SQL.

Character Strings and Dates

Character strings and date values are represented via single quotation marks! They are case sensitive. Column names, clauses, table names are **not** case sensitive.

Date format is DD-MM-YYYY by default.

ORDER BY Clause

The order of rows returned in a query result is undefined, `ORDER BY` clause, alongside the `ASC` (default) and `DESC` keywords can be used to order the rows of a query result.

```sql
SELECT ename, job,
        deptno FROM emp
        ORDER BY hiredate DESC;
SELECT ename, job, deptno,
        sal*12 annsal
        FROM emp ORDER BY annsal;
```

As can be seen in the second example, column aliases can be used to order clauses as well.

```sql
SELECT ename, job, deptno
FROM emp
ORDER BY deptno, sal DESC;
```

Multiple columns can be sorted, the order of `ORDER BY` list is the order of the sort, the first option is sorted first, and if they are equal, **then** the second column is used, and so forth. One can also sort by a column that is not in the `SELECT` list.

Chapter 7

SQL (Cont‘d) - December 3, 2020

7.1 Single Row Functions

Case Conversion Functions

Consider

```
SELECT ename, job,
FROM emp
WHERE ename = 'blake';
```

However, this will return nothing, since all of the employee names are in capital letters, we can use `UPPER` to account for this.

```
SELECT ename, job,
FROM emp
WHERE ename = UPPER('blake');
```

There are other functions of this sort, such as `CONCAT(str, str)`, `LENGTH(str)`, `SUBSTR(str, start, stop)` that can be used for numerous purposes.

Arithmetic Operators with Dates

```
SELECT ename,
        (SYSDATE - hiredate)
        /7 WEEKS,
FROM emp;
```

As you we can see, the arithmetic operations work with date types.

7.2 Multiple Tables

So far, we only worked on a single table at a time, however, queries can be run on multiple tables, easily.

7.2.1 Join

```
SELECT  table1.column, table2.colum
FROM    table1, tabl2
WHERE   table1.column1 = table2.column2;
```

Column names are prefixed with the table name.

Equijoin

```
SELECT  emp.empno, emp.ename, emp.deptno
            dept.deptno, dept.loc
FROM    emp, dept
WHERE   emp.deptno=dept.deptno
```

Joining More Than Two Tables

If we have `employee(`*`person-name`*`, street, city)`, `works(`*`person-name`*`, company-name, salary)`, `company(`*`company-name`*`, city)` and `managers(`*`person-name`*`, manager-name)`. If we wish to find the names of all employees who live in the same city as the company for which they work.

```sql
SELECT  e.person-name
FROM    employee e,
            works w, company c
WHERE   e.person-name
            = w.person-name
        AND w.company.name =
            c.company-name
        AND e.city = c.city
```

Here we joined three tables to achieve our task! [Observe that the aliases can be used in the `FROM tab`]. [I also like how we can use aliases *before* we declare them, this means SQL as a language necessitates a two-pass compiler at least! (Like most languages, honestly, except C, but hey, that's C for ya.)]

Non-Equijoins

Joins do not need equalities to work. Non-Equijoins are joins that do not rise from equalities.

```sql
SELECT  e.name, e.sal, s.grade
FROM    emp e, salgrade s
WHERE   e.sal
BETWEEN s.losal AND s.hisal
```

Outer Joins

If a row does not satisfy a join condition, the row will not appear in the query results. One can use the outer join to also see rows that do not usually meet the join condition.

```sql
SELECT  table1.column, table2.column
FROM    table1, table2
WHERE   table1.column(+) = table2.column;
```

The side with the *missing information* is given the outer join [unary postfix] operator `(+)`. [The fact that it is unary and postfix is not related, but you do not get the chance to see a unary postfix operator a lot of times, I *had* to point it out.]

	Meaning
`AVG`	Average
`MIN`	Minimum
`MAX`	Maximum
`COUNT`	Number of rows
`NVL`	Include `NULL` values
`SUM`	Sum of Numbers

Table 7.1: Some group functions.

Self Joins

Joining a table to itself is called a self join. In the `EMP` table, each employer has a manager, but managers are also employees! (So they are in the same table)

```sql
SELECT  worker.ename
            || ' works for '
            || manager.ename
FROM    emp worker, emp manager
WHERE   worker.mgr, manager.empno;
```

7.3 Group Functions

Group functions operate on sets of rows to give one result per group. Such as the maximum value of a field in a table

Group functions include `AVG`, `COUNT`, `MAX`, `MIN`, `STDDEV` all group functions except `COUNT(*)` ignore `NULL` values. Group functions are used as `function_name(column_name)`.

They include:

`COUNT`, when written without a column name, also includes the `NULL` values. `COUNT` can also be used with the `DISTINCT` keyword, such as `SELECT COUNT(DISTINCT colname)`, in this form, only the *unique* rows shall be counted.

The `NVL(column_name, assume)` function can be used to interpret `NULL` values as the `assume`, thus forcing group functions to include `NULL`.

7.3.1 Creating Groups of Data

Sometimes, a table may be needed to be divided into smaller groups. Perhaps we want to divide employees by their groups, then we want to get the average salary for each department.

```
SELECT   col , groupf(col)
FROM     table
[WHERE   condition]
[GROUP BY        col]
[ORDER BY        col];
```

For instance, using this syntax our example can be calculated via:

```
SELECT   deptno , AVG(sal)
FROM     emp
GROUP BY         deptno
```

Remember, any column not inside a group function in the `SELECT` clause, **must** appear inside the `GROUP BY` clause. *However*, the column in the `GROUP BY` clause does not need to appear inside the `SELECT` clause, (albeit the results would lack meaning).

Grouping by More Than One Column

If we wish to sum salaries in the `EMP` table, for each job, grouped by their department, we can do this as follows:

```
SELECT   deptno , job , sum(sal)
FROM     emp,
GROUP BY         deptno , job;
```

Order does matter here.

Excluding Group Results

The `HAVING` clause, places following the `GROUP BY` clause can be used to restrict the results to certain results meeting a condition.

```
SELECT   deptno , max(sal)
FROM     emp,
GROUP BY         deptno
HAVING   max(sal)>2900;
```

Will only show the those departments whose maximum salary is greater than 2900.

Nesting Group Functions

Group functions can be nested to a depth of two.

7.4 Subqueries

Suppose that we have the quert, "Who has a salary greater than Jones'", we have, in fact a subquery of "What is Jones' query", and this query's result is used in the main query.

```
SELECT   cols
FROM     table
WHERE    exr  (SELECT cols
                         FROM table
                         WHERE exr)
```

For instance, for our original query.

```
SELECT   ename
FROM     emps
WHERE    sal >
 (SELECT sal
FROM emps
WHERE ename='JOHN')
```

The subquery is executed first, since its result is used in the main query. Subqueries can be **Single-Row**, **Multi-Row** and **Multi-Column**.

`ANY` and `ALL` operator can be used to distribute a logical condition to the results of a multiple-row subqueries.

7.5 Manipulating Data

7.5.1 Adding a Row

`INSERT` statement can be used to add one row at a time to a table.

```
INSERT INTO
        dept (deptno, dname, loc)
VALUES
        (50, 'DEVELOPMENT', 'DETROIT');
```

Columns are listed optionally. If all values are provided in the default order of the columns in the table, a new entity with these values will be created.

Inserting Rows with Null Names

In the implicit method, simply omit the column from the column list, its value will be `NULL`. The explicit method is to specify the `NULL` value with the `NULL` keyword.

Inserting Special Values

There are certain special functions that can be used to insert special values, such as `SYSDATE` which returns the system date.

Copying Rows

Instead of using the values clause, by writing a subquery selecting rows from another table, one can copy rows from another table.

7.5.2 Updating a Row

The `UPDATE` statement can be used to update a row.

```
UPDATE emp
SET    deptno = 20
WHERE empno = 7782
```

If one omits the `WHERE` clause, *all* the rows will be modified. One-to-one matching between subqueries and queries can be used to update a table as well.

7.5.3 Removing a Row

the `DELETE` statement can be used to delete rows.

```
DELETE FROM    department
WHERE    dname='DEVELOPMENT';
```

Omitting the `WHERE` clause would delete all the rows of the table.

7.6 `COMMIT` and `ROLLBACK` Statements

`COMMIT` and `ROLLBACK` is used for the following purposes:

- Ensure data consistency.
- Preview data changes before making changes permanent.
- Group logically related operations.

Following a **Transaction**, one can `COMMIT` the transaction, making it permanent, or `ROLLBACK` back to a **savepoint** or to the last `COMMIT`

Please keep in mind that `COMMIT` and `ROLLBACK` is unusable in Oracle's Web-based Database Engine, as it auto-commits statements.

State of the Data Before `COMMIT` or `ROLLBACK`

- The previous state of the data can be recovered.
- The *current* user can review the results of the DML operations by using the `SELECT` statements.
- Other users cannot interact with the changed rows, they are *locked.*
- Other users cannot see the changes, yet.

State of the Data After `COMMIT`

- Dta changes are made permanent.
- The previous state of data is permanently lost.
- All users can see the results.
- Locks are released.
- Savepoints are erased.

State of the Data After `ROLLBACK`

- All pending changes are discarded, data changes are undone.
- The previous state of the data is restored.
- Locks on affected rows are released.

Chapter 8

SQL (Cont‘d) - December 17, 2020

8.1 Table Creation and Modification

`CREATE TABLE` and `ALTER TABLE` statements can be used to create new tables and modify existing tables respectively.

8.1.1 Table Creation

The `CREATE TABLE` statement can be used to create a table in SQL.

```
CREATE TABLE tableName
        (colName colType(colSize),
        colName2 colType(colSize),
        ...
        );
```

Data Types in Oracle SQL

`CREATE TABLE` with a Subquery

Using the `AS` keyword, `CREATE TABLE` can be used with a subquery. [Similar to piping output in Unix systems.], when getting data from another table this way, the constraints are preserved.

8.1.2 Table Modification

`ALTER TABLE` statement can be used in conjunction with the `ADD` and `MODIFY` keywords to add new columns or modify existing columns of a table.

Data Type	Description
`VARCHAR2(size)`	Variable-length character data
`CHAR(size)`	Fixed-length character data
`NUMBER(p, s)`	Variable length numeric data
`DATA`	Data and time values.
`LONG`	variable-length character data up to 2GBs
`CLOB`	Single-byte character data up to 4 gigabytes.
`RAW`, `LONG RAW`	Raw binary data
`BLOB`	Binary data up to 4 gigabytes
`BFILE`	Binary data stored in an external file; up to 4GBs.

Table 8.1: Data types in Oracle SQL, different SQL Dialects support different types.

The table modification can update the type of an existing column.

8.2 Constraints

During table creations, SQL can be used to add constraints to certain columns or tables. **Column-level constraints** are written after a column type/size in the table creation. Whereas, **Table-level constraints** are written after writing all the columns, the table-level constraints follow the `CONSTRAINT` keyword and may refer to multiple columns.

the `NOT NULL` Constraint

Ensures that `NULL` values are not permitted for the column. It *cannot* be defined at the table level. [In fact, this behaviour is shared when checking for equalities. `NULL` does not equal to any value including another `NULL`, similar to the behaviour of `NaN` in IEEE 754.]

the `UNIQUE KEY` Constraint

Ensures that the column cannot hold the two (or more) copies of the same value. Two values holding `NULL` *can* be inserted. The `UNIQUE KEY` constraint can be defined at either the table level or the column level.

the `PRIMARY KEY` Constraint

This constraint acts both as a `NOT NULL` constraint and a `UNIQUE KEY` constraint. The values inserted at this column must be unique *and* must not be `NULL`.

the `FOREIGN KEY` Constraint

The `FOREIGN KEY` constraint is used for columns that hold the values of a primary key of another table. It is sometimes refereed to as the Referential Integrity Constraint.

When a column is defined to be a `FOREIGN KEY`, the values it hold *must* exist in the primary key of another table.

It has a unique syntax, and is used in conjunction with the `REFERENCES`.

```
FOREIGN KEY (nativeCol)
REFERENCES foreignTable (foreignCol);
```

If followed by the `ON DELETE CASCADE`, should the row containing the primary key in the parent table is deleted, the records it references in the child table is deleted as well. It should be noted that, if not followed by this keyword, it won't be possible to delete the records whose referenced by other tables in the parent table. (§`ORA-02292`).

the `CHECK` Constraint

Defines a condition that each row must satisfy.

8.2.1 Constraint Modification

Constraints may be added, or dropped; enable or disabled to a table using the `ALTER TABLE` constraint. If used in conjunction with the `ADD CONSTRAINT` keyword, it adds a constraint; if instead used with the `DROP CONSTRAINT` keyword, it drops a constraint instead.

the `CASCADE` Constraint

Using the `CASCADE` keyword following a `DROP CONSTRAINT` of a `PRIMARY KEY`, it also drops the `FOREIGN KEY` constraints associated with this column.

Attempting to drop a primary key constraint with foreign key constraints put on it from other tables without `CASCADE` will result in an error (§`ORA-02273`)

Disabling and Enabling Constraints

Primary keys can be activated or disabled using `DISABLE` and `ENABLE` keywords.

Viewing Constraints

Constraints can be viewed from the special table `user_constraints` table. The columns associated with these constraints can also be viewed from the `user_cons_columns` special table.

Part III

Databases

Chapter 9

Integrity and Security in Databases - December 17, 2020

Integrity The database should not become incorrect or inconsistent to an inadvertent update (such as a typo)

Security The database should not be able to be reached by the unauthorised people.

Constraint The allowable states that the tables in the database may be in.

An **integrity constraint** encodes business rules about the organization. It can detect bad data entry, and it can enforce the rules of the organization. It may apply to an individual record, one or more table(s), or an entire database.

Consider `STUDENT(SId, SName, GradYear, MajorId)`. An integrity constraint may be that the `GradYear` must at least be `1863`. Another rule may be that "A professor teaches at most two sections a year" `SECTION(SectId, CourseId, Prof, YearsOffered)`, applies to this table as a whole. A constraint such as "a student cannot take a course more than once", applies to multiple tables simultaneously.

`CREATE TABLE` can only be used to specify a fraction of possible integrity constraints. Only those that apply to the individual records in a table. One needs to use **assertions** if they wish to have more complex constraints.

```sql
create assertion SmallSections
check (not exists
          select e.SectionId
          from ENROLL e
          group by e.SectionId
          having count(e.EId) > 30)
```

Here, we can see the Standard SQL specification of the integrity constraint "No section can have more than 30 students". An Assertion can denote any constraint whose violation can be expressed as a query.

The `NOT EXISTS` operator returns true when a query returns no records.

9.1 Triggers

Oracle SQL **does not have** a `CREATE ASSERTION` statement. Instead, it has **Triggers**. A trigger specifies an action that the database system invokes automatically whenever a specified event occurs. A trigger has three essential parts:

1. an **Event**, the update statement that initiates the activity.

2. a **Trigger**, the predicate that determines whether the trigger will fire.

3. an **Action**, what happens if the trigger does fire.

THe Standard SQL defines triggers as:

```
CREATE TRIGGER trigger_name
{BEFORE|AFTER} {INSERT|
               DELETE|UPDATE}
      [OF col_name] ON tablaName
[REFERENCING <old or
      new values alias>]
[FOR EACH ROW]
[WHEN condition]
action
```

A trigger action is bracketed by the keywords `BEGIN` and `END`. The Oracle SQL defines triggers as:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE|AFTER} {INSERT|DELETE|UPDATE}
      [OF col_name] ON tableName
[REFERENCING old as <oldrow>
      new as <newrow>]
[FOR EACH ROW]
[ENABLE/DISABLE]
[WHEN condition]
BEGIN
action statements
END
```

The oracle also defines `:new.column-name` and `:old.column-name` automatically without using `REFERENCING as well`. Triggers in Oracle SQL can also be disabled or enabled. Enabled triggers automatically fires, disabled triggers do not.

[It is a bit interesting that unlike other standardised languages, SQL Standard is not adhered to by its vendors. Although some variations exists in some standardised languages, the syntax deviations seen between different dialects of SQL is not seen in any other standardised language, neither in C, C++ or ADA, for instance.]

9.1.1 Triggers and Constraints

Both triggers and constraints can constraint data input, constraints are easier to write and less error-prone than triggers that enforce the same rules. A constraint rejects any update that violates it, when a trigger *fixes* the bad update. Constraints are also less error prone, and one should prefer constraints whenever possible. But, some rules can only be conveyed properly by triggers

9.2 Authorisation

Constraints and triggers help keep the database from becoming accidentally corrupted. Now, we need to make sure that malicious users cannot corrupt the database intentionally or look at private data.

SQL acts on a system of privileges. Different users can have different privileges for specific tables. A statement such as `grant update on ENROLL to Einstein` gives the update privileges to the user Einstein.

Roles

SQL has a concept of roles. [Similar to Unix concept of a user group] A role, is a category of users, and privileges can be granted to a role as a whole.

For the syntactical purposes, the `GRANT` sees roles and users as the same. A user can be assigned multiple roles at the same time.The privileges of a user will be the union of the privileges of its roles.

Column Privileges

Privileges can also be assigned on a column-level, such as `GRANT SELECT (StudentId, Grade) on ENROLL to dean, professor`. The users with these rolls will be able to select these two columns on the `ENROLL` table specifically.

Users with the `update, insert` and `delete` privileges also need to have `select` privileges for the fields in the `WHERE` clause. They do not need this privilege if their query does not contain a `WHERE` clause.

Granting Constraint Privileges

The `grant reference` privilege, grants the privilege to create constraints on a table.

Grant-Option Privileges

The `with grant option` clause, added to the end of a `grant` statement gives the user who is granted a privilege the privilege to grant that same privilege to other users.

9.2.1 Mandatory Access Control

the SQL authorisation mechanism used by SQL is called **discretionary access control**. The creator of a table has authorisation on their tables, and may give access to others at his discretion.

A database system that uses discretionary access control depends on its users to be trustworthy. If a user is not trustworthy, the user can just copy information they have access to and release it to the public.

Contrasting with Discretionary Access Control, Mandatory Access Control assigns privileges to data, instead of tables. Each record and user is assigned to a classification level, a user is authorised to see a record if the user's classification level is equal or higher to the data's classification level.

This access control is predominantly used in high-security environments such as the military, and is not preferred in commercial settings.

Chapter 10

Views - December 24, 2020

[Merry Christmas and Happy Hanukkah to any friends that may be reading this! (And Yule as well, 'suppose.) the Year is almost over, eh? I sure am happy that *this* year is ending, let us hope for a better one for the next.]

In the first lecture, three schema levels, **Physical Schemas** (files, indexes), **Conceptual Schemas** (user-neutral data) and **External Schemas** (user-specific tables).

A **view** is essentially a named query, A user can refer to a view as if it were a table.
Views are useful for:

- Hiding some data from some users.
- Making certain queries easier or more natural.
- Increasing the modularity of database access.

10.1 Virtual Views

Virtual Views are the most often used [and historically the first] views. In this chapter, when the term view is used without any descriptive, it refers to a view.

View data is not stored on the disk, it is a **virtual relation** based on the result set of a `SELECT` statement. Views are customized presentations of data in one or more tables or other views, they can be thought of as stored queries, they derive data from other tables.

Views in the Standard SQL and Oracle SQL can be created as follows [this paragraph is here just to pad the code example to the next column so it doesn't split and look ugly]

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```

[View performance is comparable to normal queries in Oracle DBMS, there is no difference since they are *stored queries.* (§9673409)) In Microsoft SQL, they are actually a bit faster due to caching (referred to as *indexing*) (§439061).]

Views can be created from two separate tables, by giving multiple tables in the `FROM` clause.

Updatable Views

A view is updatable if every record in the view has a unique corresponding record in some underlying database. In practice, this reefers to single-table view that do not involve aggregations, and multi-table views are *in* *general* not updatable.

This problem arises from the fact that, when deleting (for instance) the system can delete the record from any of the tables constituting the table, this ambiguity leads to DBMS not allowing deletion altogether.

In conclusion, views that do not have one to one relationships such as due to multi-tables or aggregation functions (§7.3).

Inserting to Views

When inserting new values, the constraints on the fields of the view rows must be taken into account, the said fields must be a part of the view since only then can we insert new values, (if the field is `NOT NULL` for instance, we need to specify its value) Therefore, just because a table is updatable does not mean it is (always) modifiable.

`DROP`

The `DROP VIEW` will drop the view itself, which can be done on any view whether or not it is updatable. It does not affect the underlying table.

However, using the `DROP TABLE` on the underlying table makes the view unusable, since the Views depend on their underlying tables for reference.

Read Only Views

`WITH READ ONLY` keyword, added to the end of a view declaration makes that view not updatable irregardless of the conditions of the tables creation.

10.2 Materialised Views

Materialised Views are a form of controlled redundancy. They are tables containing the outputs of a view, and unlike virtual views they are stored on disk, whenever the underlying data is updated, so should the corresponding materialised views.

In certain situations, Query Optimiser may decide to use a materialised view rather than the original table.

Materialising *frequently-accessed* views whose outputs are *expensive to recompute* is beneficial, however it is a bad idea to materialise a table if it is *frequently-updated* and has a *large output*.

Rather than completely resetting the materialised view, the SQL systems generally incrementally update these views.

10.2.1 Materialized Views in Oracle

When a materialized view is defined, its refresh option can be specified. Either `ON COMMIT`, which refreshes one of the materialized view's fact tables commit. `ON DEMAND` uses a manual refresh action.

Furthermore, the what type of refresh is done can be specified. `COMPLETE` (Total redefining), `FAST` (Incremental), `FORCE` (Try fast refresh, if doesn't work apply apply a `COMPLETE` refresh.), `NEVER` (Oracle Refresh Mechanisms do not apply on the table).

10.2.2 Why Use Materialised Views

Not only does Materialised Views has all of the benefits of the normal Views, but they also **improve query performance**, if the workload is designed correctly.

Chapter 11

Transactions & Recovery - January 7, 2021

A transaction is a group of operations that behaves as a single operation, a transaction is seen as a cohesive unit from the outside but actually consists of multiple operations.

11.1 Requirements

There are certain requirements for a transactions. These are called ACID properties, and every DBMS that has transactions must guarantee these, somehow.

Atomicity

The transactions are **atomic**, if one operation of a transaction fails, none of the operations within a transaction will be reflected in the database.

Either all of its operations succeed (commit) or they fail (rollback).

Consistency

The **consistency requirement** says that the database must be consistent in the start and end of the transaction, but may be temporarily inconsistent during execution.

Every transaction leaves the database consistent.

Isolation

Since if another transaction is executing concurrently with a transaction it may see an inconsistent database, the **isolation requirement** says that a transaction should not be aware of other transactions.

A transaction behaves as if it is the only thing running on the system.

This can be trivially achieved by making transactions execute serially.

Durability

The **durability requirement** says that once the user has been notified that a transaction has been completed, the updates to the database by the transaction must persist even if there are software or hardware failures.

Changes made by a committed transaction are guaranteed to be permanent.

11.2 Transaction

A transaction is initiated either by a high-level data-manipulation language such as SQL, or by a programming language such as C++ or

Java.

In SQL, the `BEGIN TRANSACTION` keyword starts a transaction, and a `COMMIT` keyword typically ends it.

Typically, multiple transactions are allowed to run concurrently in a system allowing better transaction throughput and reducing average response time for transactions, *Concurrency control schemes* are necessary to achieve isolation.

11.2.1 Schedules

A **schedule** is a sequence of instructions that specify the chronological order in which instructions of concurrent transactions are executed.

A transaction that successfully completes its execution will have a commit instruction as its last statement, otherwise its last statement will be a rollback instruction.

Serializability & Conflicts

A schedule is said to serialisable if it is equivalent to a serial schedule.

In the simplified view of transactions consisting of only write and read instructions, any pair of instructions (I_i, I_j) where both of them refer to the same variable, if both are not read statements than they are conflicting.

If I_j and I_i are consecutive in a schedule and they do not conflict, they shall also not conflict if they are swapped, If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, S and S' are **conflict equivalent**. If a schedule S is conflict equivalent to a serial schedule, than they are **conflict serializable**.

Precedence Graph

A schedule with a set of instructions $T_1, T_2, ..., T_n$ A precendence graph is drawn whose nodes are transactions and whose edges arise when a conflict arise, the direction of an edge is determined from the transaction with preceding instruction.

A schedule is conflict serialisable if and only if its precedence graph is acyclic, if it is acyclic, the serialisable order is obtained by a topological sorting of the graph, a linear order consistent with the partial

A schedule is correct only if it is conflict serialisable due to the isolation requirement of the ACID.

Recoverable Schedules

Schedules must be recoverable to make sure that if transaction A sees the effects of the transaction B, and B is rolled back, then so should A.

A schedule is recoverable if two transactions T_j and T_i, and T_j reads data T_i wrote, then T_i s commit must occur first.

A cascading rollback occurs when rollback of one transaction triggers the rollback of another, these are undesirable since they lead to the undoing of significant amount of work, for each pair of transactions T_i, T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j. This is called a **cascadeless schedule**, all cascadeless schedules are also recoverable schedules.

11.2.2 Transaction Management

A transaction starts by executing a `BEGIN TRANSACTION` operation and is terminated by a `COMMIT` or `ROLLBACK` operation, a `COMMIT` establishes as commit point and a `ROLLBACK` returns to the previous commit point.

Transaction managment primarily deals with Recovery and Concurrency.

11.3 Recovery

Recovery is the restoration of the database to a correct state after a failure has occurred. It is based on *redundancy*, physical redundancy is desirable, logical redundancy is not. A recovery *log file* keeps the previous and after state.

A recovery is performed each time the database system starts up, its purpose is to restore the database to a reasonable state.

Reasonable State

- All uncompleted transactions should be rolled back.
- All committed transactions should have their modifications written to disk.

11.3.1 Undo-Redo Recovery Algorithm

One of the main algorithms, Undo-Redo Algorithm assumes that all updated records for an uncompleted transaction will be in the log file.

It **undoes** the modifications made by uncommitted transactions, and **redoes** the modifications made by committed transactions.

11.3.2 Checkpointing

Log file size will increase drastically with time, the recovery algorithms stops considering the logs when it reaches a checkpoint.

A checkpoint is taken by forcing the contents of the main-memory buffers out to the disk and forcing a special checkpoint record out to the log, it contains all transactions that were in progress at the time the checkpoint was taken.

Chapter 12

Concurrency - January 14, 2020

In a multi-process environment multiple transactions may occur *concurrently*, this however comes with three fundamental problems.

[Not all DBMS' allow concurrency, SQLite being the most notorious example of this, not allowing write concurrency and only allowing limited read concurrency.]

the Lost Update Problem: A second transaction reads the state of the database prior to the first one writing a change and then stomps on the first one's change with its own update.

the Uncommitted Dependency Problem: A second transaction relies on a change that has not yet been committed and is rolled back after the second transaction has begun.

the Inconsistent Analysis Problem: An error that occurs when totals are calculated during interleaved updates.

12.1 Conflicts

If A and B are concurrent transactions problems can occur if A and B want to read or write the same database object tuple t.

A	B	Problems
Read	Write	Inconsistent Analysis
Write	Read	Uncommitted Change
Write	Write	Lost Update

Table 12.1: Types of conflicts and problems they cause.

	S	X
S	true	false
X	false	false

Table 12.2: Lock matrix showing the compatibility of lock types.

A problem only arises if at least one of the transactions is a write, since two reads cannot interfere with each other.

12.2 Locking

Transactions lock a portion of the database to prevent these problems, there are two types of locks:

Exclusive Lock (X) Locked by writing transaction, locks all other transactions.

Shared Lock (S) Locked by reading transaction, disallows writes.

A transaction is granted a lock if and only if the requested lock is compatible with locks already on the item by other transactions, any number of transactions can hold shared locks on an item but only one can hold an exclusive lock (and if an item has an exclusive lock on it, it cannot have shared locks.)

12.2.1 Locking Protocol

Locks can be put using different protocols.

Two-Phase Locking Protocol

This protocol ensures conflict-serialisable schedules. It consists of two phases, the **Growing Phase**, where a transaction may obtain but may not release locks, and the **Shrinking Phase** where transaction may release but may not obtain locks.

12.2.2 Deadlock

The term **deadlock** refers to when two transactions acquire locks on items each other needs, making it impossible for the both to make progress.

This situation can occur in most locking protocols, it can be solved by rolling back and releasing the locks of either one of the deadlocked transactions.

Starvation

If a transaction is waiting for an X-lock on an item while a sequence of other transactions are requesting S-locks on the same item, the original transaction is repeatedly rolled backed due to deadlocks.

This is called **starvation**, and is the sign of a badly designed concurrency control manager.

Deadlock Prevention Strategies

Wait-Die Transaction 2 waits if it is older than 1, otherwise it is rolled back.

Wound-Wait Transaction 2 rolls back transaction 1 if it is older.

Timeout-Based A transaction waits for a specified amount of time for a lock, starvation may occur.

The oldest transactions survives and the rolled back transaction is restarted with the original timestamp.

Deadlock Detection

A system must use a dadlock detection and recovery schemes, **wait-for-graph** represents vertices as transactions and edges as lock waiting, when a cycle occurs in this graph, it means a deadlock has occurred.

12.3 Isolation

Transactions are isolated to certain **Isolation Levels**.

Read Uncommitted Uncommitted records may be read.

Read committed Only committed records can be read, but successive reads may return different values.

Repeatable Read Ensures that a read is repeatable throughout the transaction.

Serialisable Ensures serialisable execution.

As the level of isolation increases, the interference and the level of concurrency decreases.

These levels apply to **data reading** only. Certain problems may arise in isolation.

Dirty Read A transaction reads a value written by another transaction that hasn't been committed yet.

Nonrepeatable read A transaction reads the same object twice during execution and finds different values despite not changing it.

Phantom Read A transaction re-executes a query returning a set of roews that satisfy a search condition and finds that the set of rows satisfying the condition has changed as a result of another recently committed transaction.

As the level of isolation increases, all these problems are tackled one by one.