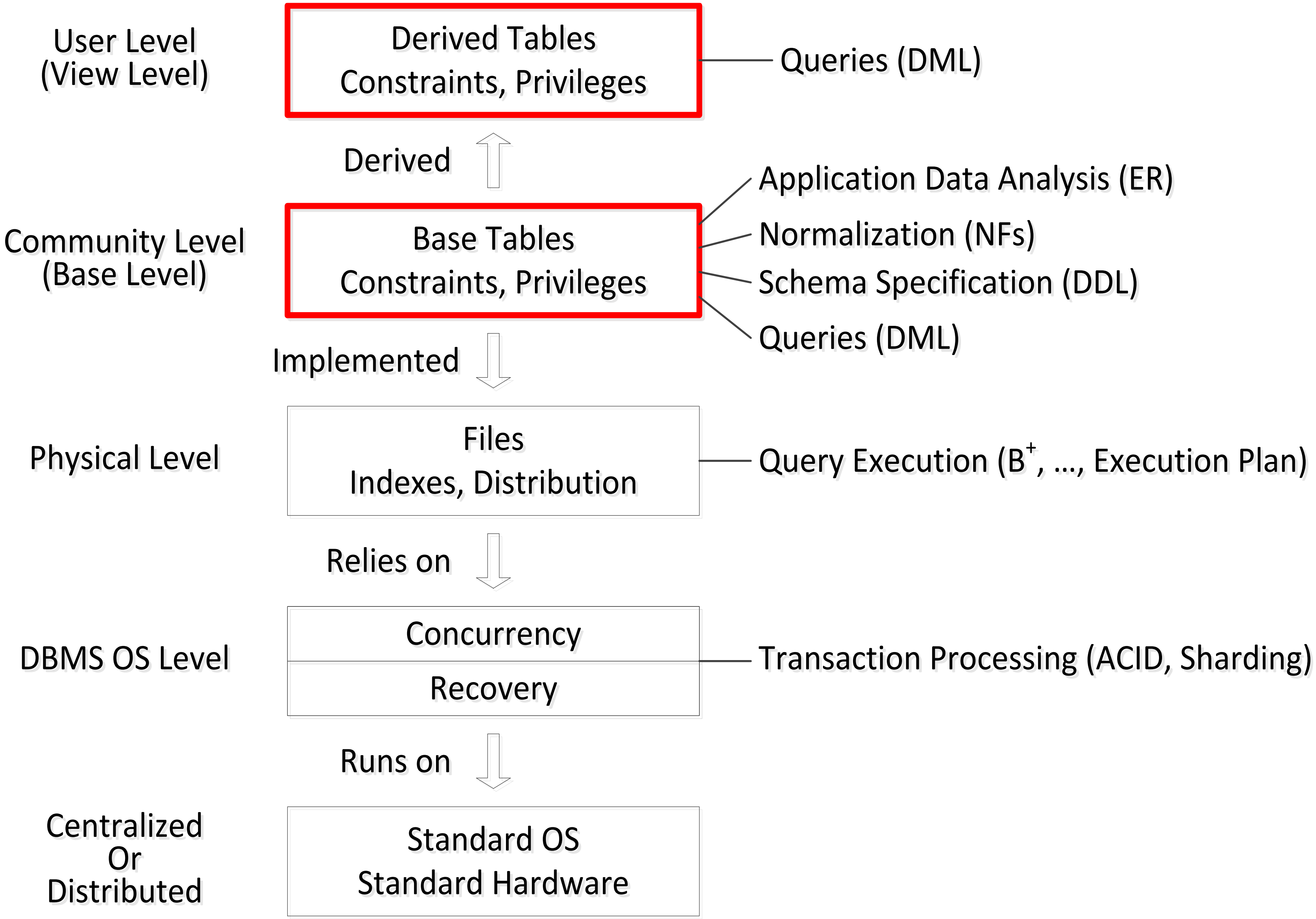
Unit 6
SQL: Data Definition Language
And Data Control Language
For Relational Databases

DDL and DCL in Context



User Level
(View Level)

Derived Tables
Constraints, Privileges

Queries (DML)

Derived

Community Level
(Base Level)

Base Tables
Constraints, Privileges

Application Data Analysis (ER)
Normalization (NFs)
Schema Specification (DDL)
Queries (DML)

Implemented

Physical Level

Files
Indexes, Distribution

Query Execution (B^+, …, Execution Plan)

Relies on

DBMS OS Level

Concurrency
Recovery

Transaction Processing (ACID, Sharding)

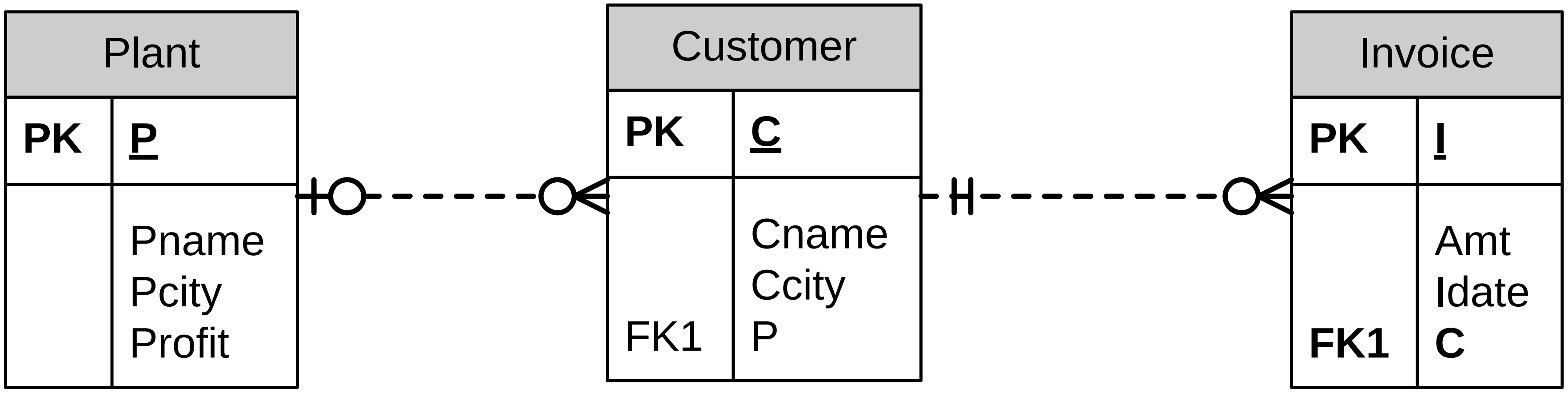
Runs on

Centralized
Or
Distributed

Standard OS
Standard Hardware

Introduction

The Tables To Be Defined And Some More



Plant	
PK	**P**
	Pname Pcity Profit

Customer	
PK	**C**
	Cname
	Ccity
FK1	P

Invoice	
PK	**I**
	Amt
	Idate
FK1	**C**

- This is the database we will define
- We do not pay attention to domains of attributes as there is not much interesting in that
- |-0 means zero or one (customer has zero or one plant). |-| means exactly one (invoice goes to one customer).

Defining A Relational Database

- We will focus only on some of the basic capabilities for defining a relational database
- The standard is very extensive and provides for a rich repertoire of useful capabilities
- We touch on the basics.
- But enough for defining reasonable-complexity databases

CREATE TABLE (Specifying A Relation Schema)

Basic Definition

-
  ```
  CREATE TABLE Plant (
   P CHAR(10),
   Pname CHAR VARYING(10),
   Pcity CHAR VARYING(10),
   Profit NUMBER
  );
  ```

- This is a minimal definition
  - Name of the table
  - Names of the columns
  - Domains of the columns

Basic Definition

-
  ```
  CREATE TABLE Customer (
   C CHAR(10),
   Cname CHAR VARYING(10),
   Ccity CHAR VARYING(10),
   P CHAR(10)
  );
  ```

- This is a minimal definition
  - Name of the table
  - Names of the columns
  - Domains of the columns

Basic Definition

-
  ```
  CREATE TABLE Invoice (
   I CHAR(10),
   Amt NUMBER,
   Idate DATE,
   C CHAR(10)
  );
  ```

- This is a minimal definition
  - Name of the table
  - Names of the columns
  - Domains of the columns

Permitted Data Types (Data Domains)

- SQL standard specifies permitted data types, which can be roughly grouped into several families
  - Integers (small or long)
  - Real numbers (standard or double length and with various precisions)
  - Character strings (fixed or variable length)
  - Bit strings (fixed or variable length)
  - Dates and times (various specifications with various time “granularity”)

- Systems have different implementations and modifications of the standard

Notation

- In some of the slides, new concepts will be introduced
- The SQL specifications will be in red color and bold to draw attention to them

Minimum Specification For Plant

-
  ```
  CREATE TABLE Plant (
   P CHAR(10) NOT NULL,
   Pname CHAR VARYING(10),
   Pcity CHAR VARYING(10),
   Profit NUMBER,
   CONSTRAINT C_20 PRIMARY KEY (P),
   CONSTRAINT C_30 UNIQUE (Pcity, Profit),
   CONSTRAINT C_40 CHECK ( Pcity <> Pname ),
   CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR
  (Profit > 1000) )
  );
  ```

- This is a minimal definition
  - Name of the table
  - Names of the columns
  - Domains of the columns

Not Null

-
  ```
  CREATE TABLE Plant (
   P CHAR(10) NOT NULL,
   Pname CHAR VARYING(10),
   Pcity CHAR VARYING(10),
   Profit NUMBER,
   CONSTRAINT C_20 PRIMARY KEY (P),
   CONSTRAINT C_30 UNIQUE (Pcity, Profit),
   CONSTRAINT C_40 CHECK ( Pcity <> Pname ),
   CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR
  (Profit > 1000) )
  );
  ```

- Specifies that the values in these columns (could be more than one such column) must not be NULL

Constraints

-
  ```
  CREATE TABLE Plant (
   P CHAR(10) NOT NULL,
   Pname CHAR VARYING(10),
   Pcity CHAR VARYING(10),
   Profit NUMBER,
   CONSTRAINT C_20 PRIMARY KEY (P),
   CONSTRAINT C_30 UNIQUE (Pcity, Profit),
   CONSTRAINT C_40 CHECK ( Pcity <> Pname ),
   CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR
  (Profit > 1000) )
  );
  ```

- Some constraint on the tables
  - Constraint name, here C_20, is not required, but it is a very good idea to give unique names to a constraint, so it can be later DROPPed or ALTERed by referring to it by its name
  - Constraint name should reflect something about the constraint, but to save space I used short names

Primary Key

-
  ```
  CREATE TABLE Plant (
   P CHAR(10) NOT NULL,
   Pname CHAR VARYING(10),
   Pcity CHAR VARYING(10),
   Profit NUMBER,
   CONSTRAINT C_20 PRIMARY KEY (P),
   CONSTRAINT C_30 UNIQUE (Pcity, Profit),
   CONSTRAINT C_40 CHECK ( Pcity <> Pname ),
   CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR
  (Profit > 1000) )
  );
  ```

- The column P is the primary key (only one possible)
  - This requires that it must not be NULL (this is not necessary to state in some systems, as the primary key condition automatically forces it by SQL standard)
- Primary key could be several columns, e.g., PRIMARY KEY(Pcity, Profit); but not in our example

Unique

-
  ```
  CREATE TABLE Plant (
   P CHAR(10) NOT NULL,
   Pname CHAR VARYING(10),
   Pcity CHAR VARYING(10),
   Profit NUMBER,
   CONSTRAINT C_20 PRIMARY KEY (P),
   CONSTRAINT C_30 UNIQUE (Pcity, Profit),
   CONSTRAINT C_40 CHECK ( Pcity <> Pname ),
   CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR
  (Profit > 1000) )
  );
  ```

- The “subtuple” Pcity,Pname is a candidate key
  - There is no requirement, in general, about any of its column being not NULL
  - But for every value of the pair there is at most one tuple in Plant
  - To reiterate: all the columns of the primary key must not be NULL

Check (And Unknown)

-
  ```
  CREATE TABLE Plant (
   P CHAR(10) NOT NULL,
   Pname CHAR VARYING(10),
   Pcity CHAR VARYING(10),
   Profit NUMBER,
   CONSTRAINT C_20 PRIMARY KEY (P),
   CONSTRAINT C_30 UNIQUE (Pcity, Profit),
   CONSTRAINT C_40 CHECK ( Pcity <> Pname ),
   CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR
  (Profit > 1000) )
  );
  ```
- Every tuple must satisfy this condition
- The condition is ***satisfied***, when it is either
  - ***TRUE, or***
  - ***UNKNOWN*** (so if Pcity is Null, this condition is satisfied)
- ***Recall in SQL SELECT queries: UNKNOWN implies not satisfied, effectively FALSE; here effectively TRUE***

Check

-
  ```
  CREATE TABLE Plant (
   P CHAR(10) NOT NULL,
   Pname CHAR VARYING(10),
   Pcity CHAR VARYING(10),
   Profit NUMBER,
   CONSTRAINT C_20 PRIMARY KEY (P),
   CONSTRAINT C_30 UNIQUE (Pcity, Profit),
   CONSTRAINT C_40 CHECK ( Pcity <> Pname ),
   CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR
  (Profit > 1000) )
  );
  ```

- This is: (Pcity = 'Chicago') → (Profit > 1000)

  By standard rules of Boolean operators (propositional calculus)

Check (And Unknown)

-
  ```
  CREATE TABLE Plant (
   P CHAR(10) NOT NULL,
   Pname CHAR VARYING(10),
   Pcity CHAR VARYING(10),
   Profit NUMBER,
   CONSTRAINT C_20 PRIMARY KEY (P),
   CONSTRAINT C_30 UNIQUE (Pcity, Profit),
   CONSTRAINT C_40 CHECK ( Pcity <> Pname ),
   CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR
  (Profit > 1000) )
  );
  ```
- Returning to semantics of UNKNOWN and OR, this constraint has to evaluate to TRUE or UNKNOWN to be satisfied, so we need

  (Pcity is not Chicago or IS NULL) or (Profit is greater than 1000 or IS NULL)
- So for Chicago the profit is greater than 1000 or IS NULL

Defaults

-
  ```
  CREATE TABLE Customer (
   C CHAR(10) NOT NULL,
   Cname CHAR VARYING(10) DEFAULT  NULL,
   Ccity CHAR VARYING(10),
   P CHAR(10) DEFAULT  'Main',
   CONSTRAINT C_60 PRIMARY KEY (C),
   CONSTRAINT C_70 FOREIGN KEY (P) REFERENCES
  Plant(P) ON DELETE SET NULL
  );
  ```

- It is possible to specify defaults
  - E.g., when a tuple is inserted and only C and Ccity are specified, the system knows to specify NULL for Cname and Main for P

Foreign Key

-
  ```
  CREATE TABLE Customer (
   C CHAR(10) NOT NULL,
   Cname CHAR VARYING(10) DEFAULT NULL,
   Ccity CHAR VARYING(10),
   P CHAR(10) DEFAULT 'Main',
   CONSTRAINT C_60 PRIMARY KEY (C),
   CONSTRAINT C_70 FOREIGN KEY (P) REFERENCES
  Plant(P) ON DELETE SET NULL
  );
  ```

- P in Customer has to reference the primary key of Plant (in this example) or a UNIQUE (not in this example)
- This means that one of two conditions is satisfied
  - P has a non NULL value and this value of P appears in Plant
  - P is NULL

    Of course, if P were specified as NOT NULL, this could not be the case

On Delete Set Null

-
  ```
  CREATE TABLE Customer (
   C CHAR(10) NOT NULL,
   Cname CHAR VARYING(10) DEFAULT NULL,
   Ccity CHAR VARYING(10),
   P CHAR(10) DEFAULT 'Main',
   CONSTRAINT C_60 PRIMARY KEY (C),
   CONSTRAINT C_70 FOREIGN KEY (P) REFERENCES
  Plant(P) ON DELETE SET NULL
  );
  ```

- P in Customer has to reference the primary key of Plant or a UNIQUE
- But note, that P in Customer is not required to be NOT NULL
- We have a specification that if P is listed in some tuple of Customer and that tuple with that P is deleted from Plant then that value of P in Customer is replaced by NULL

Not Null

-
  ```
  CREATE TABLE Invoice (
   I CHAR(10) NOT NULL,
   Amt NUMBER,
   Idate DATE,
   C CHAR(10) NOT NULL,
   CONSTRAINT C_80 PRIMARY KEY (I),
   CONSTRAINT C_90 FOREIGN KEY (C) REFERENCES
  Customer(C) ON DELETE CASCADE
  );
  ```

- NOT NULL can be specified also for columns not in the primary key

On Delete Cascade

-
  ```
  CREATE TABLE Invoice (
   I CHAR(10) NOT NULL,
   Amt NUMBER,
   Idate DATE,
   C CHAR(10) NOT NULL,
   CONSTRAINT C_80 PRIMARY KEY (I),
   CONSTRAINT C_90 FOREIGN KEY (C) REFERENCES
  Customer(C) ON DELETE CASCADE
  );
  ```

- We have a specification that if C listed in some tuple of Invoice is deleted from Customer (that is the tuple with this value of primary key is deleted), all the tuples with this value of C in Invoice must be deleted

Maintenance of Referential Integrity

- In order to maintain referential integrity constraints, the system will reject any operation that will violate it.
    - There are subtle interactions if NULLs are present; we will not discuss them here
- CREATE TABLE Invoice (

```
CREATE TABLE Invoice (
 I CHAR(10) NOT NULL,
 Amt NUMBER,
 Idate DATE,
 C CHAR(10) NOT NULL,
 CONSTRAINT C_80 PRIMARY KEY (I),
 CONSTRAINT C_90 FOREIGN KEY (C) REFERENCES
Customer(C)  ON . . .
);
```

Maintenance of Referential Integrity On Update

- This constraint “will act” when:
  - An **INSERT** or an **UPDATE** on Invoice is attempted that would produce there a value of C that does not exist in Customer.
  - A **DELETE** or an **UPDATE** on Customer is attempted that will leave tuples in Invoice in which the value of C does not appear in any tuple of Customer.
- The default is **NO ACTION**, that is the above will not be permitted
- We will briefly discuss other options in case of UPDATEs of Customer and skip what happens in other cases
  - **CASCADE**: the new value of the primary key (or UNIQUE if the foreign key refers to UNIQUE attribute or attributes) is copied to the foreign key
  - **SET NULL**: the new value of the foreign key (or UNIQUE) is NULL (if NULL is allowed)
  - **SET DEFAULT**: the new value of the foreign key (or UNIQUE) is a specified default value (which of course has to appear in Customer)

Starting With A Basic Definition

- It is generally a good idea to start with a basic definition and augment it with constraints later
- We see how this is done

Basic Definition

```
CREATE TABLE Plant (
 P CHAR(10) NOT NULL,
 Pname CHAR VARYING(10),
 Pcity CHAR VARYING(10),
 Profit NUMBER
);
```

Basic Definition

-
  ```
  CREATE TABLE Customer (
   C CHAR(10) NOT NULL,
   Cname CHAR VARYING(10) DEFAULT (NULL),
   Ccity CHAR VARYING(10),
   P CHAR(10) DEFAULT ('Main')
  );
  ```

Basic Definition

-
  ```
  CREATE TABLE Invoice (
   I CHAR(10) NOT NULL,
   Amt NUMBER,
   Idate DATE,
   C CHAR(10) NOT NULL
  );
  ```

More On Specifying A Relation Schema

Altering The Definition To Add Constraints

- ALTER TABLE Plant ADD CONSTRAINT C_20 PRIMARY KEY (P);
- ALTER TABLE Customer ADD CONSTRAINT C_60 PRIMARY KEY (C);
- ALTER TABLE Invoice ADD CONSTRAINT C_80 PRIMARY KEY (I);
- ALTER TABLE Customer ADD CONSTRAINT C_70 FOREIGN KEY (P) REFERENCES Plant(P) ON DELETE SET NULL;
- ALTER TABLE Invoice ADD CONSTRAINT C_90 FOREIGN KEY (C) REFERENCES Customer(C) ON DELETE CASCADE;

Altering The Definition To Add Constraints

- ALTER TABLE Plant ADD CONSTRAINT C_30 UNIQUE (Pcity, Profit);
- ALTER TABLE Plant ADD CONSTRAINT C_40 CHECK (Pcity <> Pname);
- ALTER TABLE Plant ADD CONSTRAINT C_50 CHECK ((Pcity <> 'Chicago') OR (Profit > 1000));

Adding Constraint Not Currently Satisfied

- What happens if a table is ALTERed with a CONSTRAINT that is currently not satisfied by the table?

```
drop table TEST_UNKNOWN;
create table TEST_UNKNOWN (
 X CHAR VARYING(10),
 Y CHAR VARYING(10)
 );
Insert into TEST_UNKNOWN values('a','b');
select *
from TEST_UNKNOWN;
alter table TEST_UNKNOWN add constraint
TEST_UNKNOWN check (X=Y);
Insert into TEST_UNKNOWN values('c','d');
select *
from TEST_UNKNOWN;
```

Adding Constraint Not Currently Satisfied

- The constraint is rejected and the system proceeds

```
Table dropped.
Table created.
1 row created.
X          Y
---------- ----------
a          b
alter table TEST_UNKNOWN add constraint
TEST_UNKNOWN check (X=Y)
*
ERROR at line 1:
ORA-02293: cannot validate (KEDEM.TEST_UNKNOWN) -
check constraint violated
1 row created.
X          Y
---------- ----------
a          b
c          d
```

FOREIGN KEY Can Reference The Table Itself (We Have Seen This Before)

- We store information about women
- With each woman, we store her mother, if the mother is known
- The mother is a woman

-
  ```
  create table WOMAN (
   SSN CHAR(9) not null,
   NAME CHAR VARYING(10) default (null),
   MOTHER CHAR(9) default (null)
  );
  ```

-
  ```
  alter table WOMAN add constraint C_01 primary key
  (SSN);
  alter table WOMAN add constraint C_02 foreign key
  (MOTHER) references WOMAN(SSN);
  ```

Referencing Unique

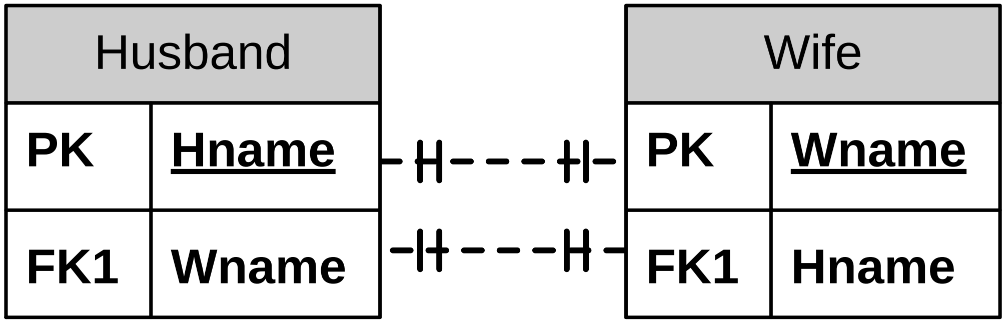
- Foreign key can also refer to UNIQUE and not only to PRIMARY KEY
- So we could also add to our database such a constraint, for which we look at an example

-
  ```
  CREATE TABLE Test (
   TestID CHAR(10) NOT NULL,
   TestPname CHAR VARYING(10),
   TestPcity CHAR VARYING(10),
   TestProfit NUMBER
  );
  ```

-
  ```
  ALTER TABLE Test ADD CONSTRAINT C_99 FOREIGN
  KEY (TestPcity, TestProfit) REFERENCES Plant(Pcity,
  Profit);
  ```

Sometimes It Is Necessary To Define Tables First And Then Add Constraints

- If you define a foreign key constraint, it cannot refer to a table that has not yet been designed
- Consider the following Visio diagram for some types of marriages



Husband	
PK	<u>Hname</u>
FK1	Wname

Wife	
PK	<u>Wname</u>
FK1	Hname

- You have “circular” dependencies
  - You cannot fully define Husband before Wife
  - You cannot fully define Wife before Husband
- Therefore

1. Produce basic definitions for Husband and Wife
2. Alter them by adding constraints later

When Are Constraints Checked?

- Essentially, each row of the TABLE has to satisfy the constraint
- Constraints are checked as tables are modified (immediately or deferred until later, generally until the end of a transaction)
- The actual checking is done either before/after each statement or at the end of a transaction
  - It is done at the end of a transaction to allow changes that cannot be done in a single statement
  - For example if Total = Checking + Savings and money is moved from Checking to Savings this constraint could be violated in the middle of the move, but must be satisfied before and after the move
- So as part of specification of a constraint one can specify
  - ***NOT DEFERRABLE*** (this is the default), or
  - ***DEFERRABLE***

Assertions

Assertions

- Assertion is like a CHECK constraint, ***but*** it is not attached to a TABLE definition; it is “free floating”

-
  ```
  CREATE ASSERTION Assertion01
  CHECK
  ( (SELECT COUNT (*) FROM Plant)
    +  (SELECT COUNT (*) FROM Customer)
                       < 1000 );
  ```

- Assertions are more natural than previously described constraints, especially when referring to several tables

- However, they are frequently not implemented
- It is very difficult to implement them both correctly and efficiently

UNIQUE And PRIMARY KEY

UNIQUE And PRIMARY KEY

- Back to our old example
- CREATE TABLE City
Country NOT NULL,
State,
Name NOT NULL,
Longitude NOT NULL,
Latitude NOT NULL
);

- A city can be identified in one of two ways
  - By its geographic location: Longitude and Latitude
  - By its official "hierarchy of names": Country, State, Name

- It may be the case that some countries are **not** partitioned into states (or equivalent units)
  - For them it is natural to allow State to be NULL, as opposed to faking something

UNIQUE And PRIMARY KEY

- The following is OK

-
  ```
  CREATE TABLE City
  Country NOT NULL,
  State,
  Name NOT NULL,
  Longitude NOT NULL,
  Latitude NOT NULL,
  UNIQUE (Country, State, Name),
  PRIMARY KEY (Longitude, Latitude) );
  ```

UNIQUE And PRIMARY KEY

- The following ***is not OK***
- CREATE TABLE City
  Country NOT NULL,
  State,
  Name NOT NULL,
  Longitude NOT NULL,
  Latitude NOT NULL,
  PRIMARY KEY (Country, State, Name),
  UNIQUE (Longitude, Latitude));

- ***Because State could be NULL, not permitted in primary key***
- We will see why primary keys should not contain NULLs (there are other reasons for this too)

UNIQUE And PRIMARY KEY

- Small database
  - CREATE TABLE City_Population
    Country NOT NULL,
    State,
    Name NOT NULL,
    Longitude NOT NULL,
    Latitude NOT NULL,
    Population,
    PRIMARY KEY (Country, State, Name),
    UNIQUE (Longitude, Latitude));
  - CREATE TABLE City_Size
    Country NOT NULL,
    State,
    Name NOT NULL,
    Longitude NOT NULL,
    Latitude NOT NULL,
    Size,
    PRIMARY KEY (Country, State, Name),
    UNIQUE (Longitude, Latitude));

UNIQUE And PRIMARY KEY

- We want to combine information about cities from both tables

- SELECT *
  FROM City_Population, City_Size
  WHERE (City_Population.Country = City_Size.Country
  AND City_Population.State = City_Size.State
  AND City_Population.Name = City_Size.Name) ;
- We will not get anything for cities in countries that are not partitioned into states!
- Because the result of comparison of say (Monaco, NULL, Monaco-Ville) = (Monaco, NULL, Monaco-Ville) is UNKNOWN
- Therefore, we cannot have (Country,State,Name) as PRIMARY KEY

Workaround

- The following can be done if we want to use UNIQUE set of attributes for joining in our example

-
  ```
  SELECT *
  FROM City_Population, City_Size
  WHERE City_Population.Country = City_Size.Country
  AND City_Population.Name = City_Size.Name
  AND ( City_Population.State = City_Size.State
     OR (City_Population.State IS NULL
      AND City_Size.State IS NULL ) );
  ```
- But this is very burdensome and potentially easily forgotten

AUTOINCREMENT for the Primary Key

Auto-incrementing Primary Keys

- **Primary keys are critical to the maintenance of the database**
  - The primary key, in effect, identifies a specific row in a table (assume no duplicate rows, for simplicity)
  - It is used typically for binary many-to-one mappings
- Frequently, it is desirable that primary keys **do not** reflect anything in the real world
  - Do not use, Social Security Numbers, Passport Numbers, or similar
- Frequently, it is desirable that primary keys occupy little space (this is important) and that they be “meaningless”
  - To have less space devoted to foreign keys (which are typically copies of primary keys)
  - To have less space devoted to indexes as we will see in a future unit; **this is important for performance**
  - Skip to page 65.

Oracle Version

- The simplest case is to start with 1 and increment by 1
- ***Replace***

  -
    ```
    CREATE TABLE Plant (
     P CHAR(10) NOT NULL,
     Pname CHAR VARYING(10),
     Pcity CHAR VARYING(10),
     Profit NUMBER,
     PRIMARY KEY (P)
    );
    ```

    by

  -
    ```
    CREATE TABLE Plant (
     Plant_Id NUMBER GENERATED ALWAYS AS IDENTITY,
     P CHAR(10) NOT NULL,
     Pname CHAR VARYING(10),
     Pcity CHAR VARYING(10),
     Profit NUMBER,
     PRIMARY KEY (Plant_Id),
     UNIQUE (P),
    );
    ```

Oracle Version

- Plant_Id will be **auto-generated**
- The system needs to be told that it will serve as a primary key, otherwise it will not know that
- Auto-generated primary key must be a number and a single column
- When we insert the first tuple into Plant, specifying some values of P, Pname, Pcity, Profit, then 1 will be automatically inserted into Plant_Id
- So if we insert (276, Main, Chicago, 0.5), the system will insert (1, 276, Main, Chicago, 0.5)
- When we insert the second tuple into Plant, specifying some values of P, Pname, Pcity, Profit, then 2 will be automatically inserted into Plant_Id
- So if we insert (93, Small, Boston, 0.6), the system will insert (2, 93, Small, Boston, 0.6)

Oracle Version

- Standard users cannot modify such auto-generated primary keys

- There are various option for the autoincrement keys
  - User can specify, e.g., start with 7 and increment by 12
  - User can specify the value of the autoincrement key for a new tuple (not ALWAYS by the systems as in the example)

- Other attributes, and not only the primary key, can also be auto-generated

Oracle Version

```
CREATE TABLE R(
A_ID NUMBER GENERATED ALWAYS AS IDENTITY,
PRIMARY KEY (A_ID),
A CHAR(30) NOT NULL,
UNIQUE (A)
);

INSERT INTO R (A) VALUES('ALICE’);
INSERT INTO R (A) VALUES('BOB');

SELECT *
FROM R;

      A_ID A
---------- ------------------------------
         1 ALICE
         2 BOB
```

- The old primary key was A

Oracle Version

```
UPDATE R
SET (A_ID) = 5
WHERE A = 'ALICE’  ;
```

```
SET (A_ID) = 5
     *
ERROR at line 2:
ORA-32796: cannot update a generated always identity column
```

```
SELECT *
FROM R ;
```

```
      A_ID A
---------- ------------------------------
         1 ALICE
         2 BOB
```

- Cannot change the values of the primary key in the table

Oracle Version

```
CREATE TABLE S(
B_ID NUMBER GENERATED ALWAYS AS IDENTITY,
B CHAR(30) NOT NULL,
UNIQUE (B),
C NUMBER,
FOREIGN KEY (C) REFERENCES R(A_ID)
);

INSERT INTO S (B) VALUES('CAROL’);
INSERT INTO S (B) VALUES('DON');

SELECT *
FROM S;

      B_ID B                                       C
---------- ------------------------------ ----------
         1 CAROL
         2 DON
```

- Note foreign key attribute “to connect to” table R

Oracle Version

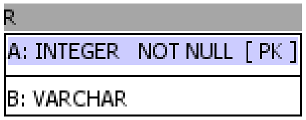
```
UPDATE S
SET (C) =  (SELECT R.A_ID
             FROM R
             WHERE R.A = 'ALICE’)
WHERE B ='DON' ;
```

```
SELECT *
FROM S ;
```

```
 B_ID B                                       C
---------- ------------------------------ ----------
         1 CAROL
         2 DON                                     1
```

- “Connecting” Don to Alice

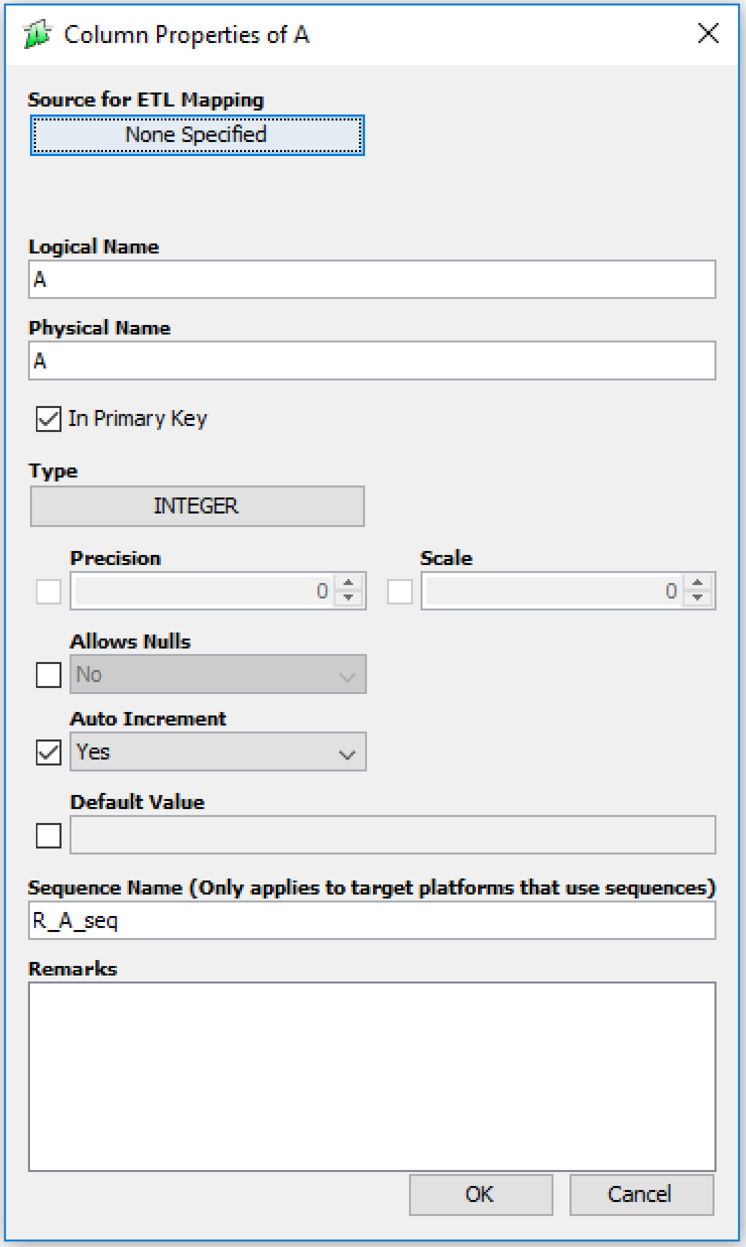
SQL Power Architect Definition



R

A: INTEGER NOT NULL [PK]

B: VARCHAR



Column Properties of A

Source for ETL Mapping
None Specified

Logical Name
A

Physical Name
A

☑ In Primary Key

Type
INTEGER

□ **Precision** 0

□ **Scale** 0

□ **Allows Nulls** No

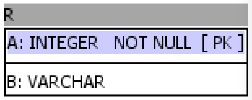
☑ **Auto Increment** Yes

□ **Default Value**

Sequence Name (Only applies to target platforms that use sequences)
R_A_seq

Remarks

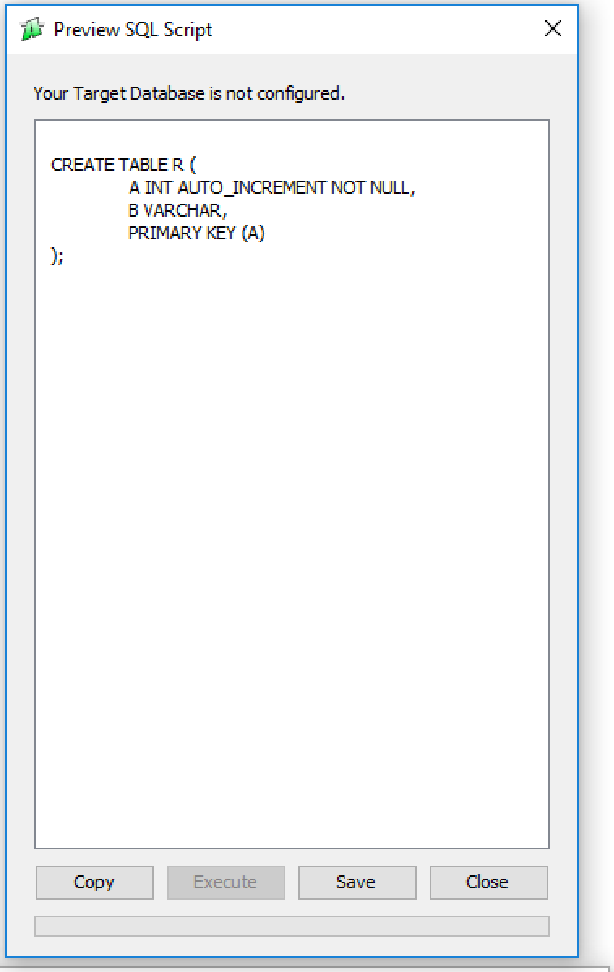
SQL Power Architect Generated SQL For MySQL



R

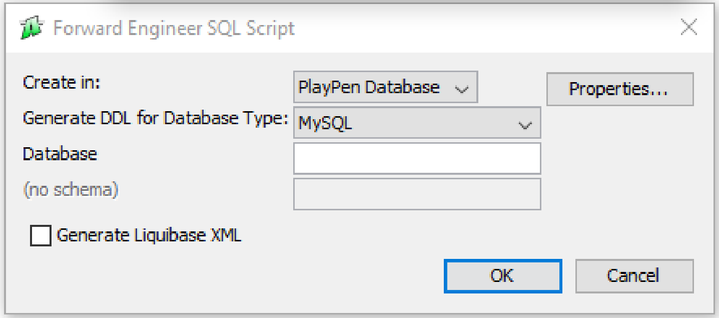
A: INTEGER NOT NULL [PK]

B: VARCHAR



Your Target Database is not configured.

```
CREATE TABLE R (
          A INT AUTO_INCREMENT NOT NULL,
          B VARCHAR,
          PRIMARY KEY (A)
);
```



Create in: PlayPen Database

Generate DDL for Database Type: MySQL

Database

(no schema)

□ Generate Liquibase XML

More On UNKNOWN

Example On UNKNOWN Using Oracle

- We define a table and insert tuples into it
- Our second tuple is interesting
  - For DDL and DML INSERT this satisfies the constraint (actually UNKNOWN)
  - For DML SELECT this does not satisfy the constraint (actually UNKNOWN)

```
create table TEST_UNKNOWN (
 X CHAR VARYING(10),
 Y CHAR VARYING(10),
 check (X=Y)
 );

Insert into TEST_UNKNOWN values('a','a');
Insert into TEST_UNKNOWN values('b',null);
Insert into TEST_UNKNOWN values('c','d');
```

Example On UNKNOWN Using Oracle

```
Table created.

1 row created.

1 row created.

Insert into TEST_UNKNOWN values('c','d')
*
ERROR at line 1:
ORA-02290: check constraint (KEDEM.SYS_C00111308)
violated
```

- Note that the third tuple does not satisfy the constraint for DDL

Example On UNKNOWN Using Oracle

```
select *
from TEST_UNKNOWN;

X          Y
---------- ----------
a          a
b
```

- The table containing tuples that satisfy the condition X = Y has two tuples

Example On UNKNOWN Using Oracle

```
select *
from TEST_UNKNOWN
where x=y;

X          Y
---------- ----------
a          a
```

- One tuple is output when condition X = Y is specified

VIEWs

Views

- We now proceed to the definition of the user level, that is to the definition of views
- Generally speaking, a view consists of “continuously current” table that is derived by means of a SELECT statement from other tables
- For example, we could write

```
CREATE VIEW GoodPlant
AS SELECT *
FROM Plant
WHERE Profit > 0.0;
```

- We could now execute a query against the view

```
SELECT P
FROM GoodPlant
WHERE City = 'Chicago';
```

- This will give all P for Chicago where Profit is positive

Views Versus Snapshots

- View is not a snapshot, which is static
- View can be thought of as a macro.
- Therefore we should think of the following procedure for computing the answer to the last query:
- The system computes the value of the table GoodPlant
- The system executes the query against the table GoodPlant
- In practice, the system may compute the answer differently (e.g. as two selects), however, the result will be equivalent to the canonical procedure described above

Views Defined by Queries

- In general, almost any query definition could be used to define a view, so we could have:

```
CREATE VIEW Customer_In_The_City
AS SELECT Cname
FROM Plant, Customer
WHERE Pcity = Ccity
AND Plant.C = Customer.C;
```

- Views can also be defined WITH **CHECK** OPTION, which we will discuss later.

Updating Views

- Views, in principle, can be updated just like the base tables
- However, all updates to views must be reflected in a correct update to the base table.
- Let us start with the view

```
CREATE VIEW GoodPlant
AS SELECT *
FROM Plant
WHERE Profit > 0.0;
```

- Then it is clear what should be inserted into the table Plant if the following is issued:

```
INSERT INTO GoodPlant
VALUES (675, 'Major', 'Philadelphia', .25);
```

Updating Views While Forcing Defaults

- Consider now the view

```
CREATE VIEW SomePlant
AS SELECT P, Pname, City
FROM Plant;
```

- Then, if the value of Profit can be NULL or has a defined default value, it is clear what should be inserted into the table Plant if the following is issued:

```
INSERT INTO SomePlant
VALUES (675, 'Major', 'Philadelphia');
```

Update To View Not Reflected In It

- Consider the view

```
CREATE VIEW Plant_In_Chicago
AS SELECT *
FROM Plant
WHERE City = 'Chicago';
```

- According to SQL the following update is valid

```
INSERT INTO Plant_In_Chicago
VALUES (897,'Minor','Philadelphia',.1);
```

- It is reflected properly in the base table Plant, however, it does not show in the view, of course

Checking for Updates Not Reflected in View

- Instead, if we define the view

```
CREATE VIEW Plant_In_Chicago
AS SELECT *
FROM Plant
WHERE City = 'Chicago'
WITH CHECK OPTION;
```

- Then the update

```
INSERT INTO Plant_In_Chicago
VALUES (897,'Minor','Philadelphia',.1);
```

will be rejected

Some Views Cannot Be Updated

- Consider the view

```
CREATE VIEW Profit_On_Date
AS SELECT Profit, Date
FROM Plant, Invoice, Customer
WHERE Plant.P = Customer.P
AND Invoice.C = Customer.C;
```

- There is no meaning to the update

```
INSERT INTO Profit_On_Date
VALUES (0.9,2009-02-01);
```

- Why?
  - Because there is no well-defined way for reflecting this update in the base tables
  - Several tables would need to be modified in a non-deterministic fashion

Some Views That Cannot Be Updated

- Consider the view

  ```
  CREATE VIEW Avg_Amt
  AS SELECT AVG(Amt)
  FROM Invoice
  WHERE Idate = ‘2009-02-01’;
  ```

- It is not permitted to issue:

  ```
  INSERT INTO Avg_Amt
  VALUES (75);
  ```

  - There is no way of changing the base tables in a well-defined way.

Some Views That Cannot Be Updated

- Consider the view

```
CREATE VIEW Cities_With_Plant
AS SELECT Pcity
FROM Plant;
```

- It is not permitted to issue

```
INSERT INTO Cities_With_Plant
VALUES ('Palm Beach');
```

  - P cannot have a NULL value, as it was the primary key

Views That Are Updatable In Standard SQL

- The following are the major conditions (there are others) that must be true for an updatable view

  - Is drawn from one TABLE

    No joins, unions, differences, intersections

  - If the underlying TABLE is a view, it must be updatable

  - The SELECTed columns are column references (each column at most once and without DISTINCT) and not values or aggregates

  - No GROUP BY
  - Skip to page 88.

TRIGGERs

Using A Trigger To Update A View

-
  ```
  CREATE TABLE r (
  a CHAR (10) NOT NULL,
  b CHAR (10) NOT NULL,
   PRIMARY KEY (a)
  );
  ```

-
  ```
  CREATE TABLE s (
  a CHAR (10) NOT NULL,
  c CHAR (10) NOT NULL,
  PRIMARY KEY (a)
  );
  ```

-
  ```
  CREATE VIEW t AS
  SELECT r.a AS a, r.b AS b, s.c AS c
  FROM r, s
  WHERE r.a = s.a;
  ```

Using A Trigger To Update A View

-
  ```
  CREATE TRIGGER trigger02
  INSTEAD OF UPDATE ON t
  REFERENCING NEW AS new
  BEGIN UPDATE s
  SET c = :new.c
  WHERE a = :old.a;
  END trigger02;
  .
  RUN
  ```

-
  ```
  UPDATE t
  SET c = 'q'
  WHERE a = '2';
  ```

Using A Trigger To Update A View

- Tables R, S, and view T before update on the view

```
A          B
---------- ----------
1          e
2          f

A          C
---------- ----------
1          m
2          n
3          o

A          B          C
---------- ---------- ----------
1          e          m
2          f          n
```

Using A Trigger To Update A View

- Tables R, S, and view T after update on the view using trigger02

```
A          B
---------- ----------
1          e
2          f

A          C
---------- ----------
1          m
2          q
3          o

A          B          C
---------- ---------- ----------
1          e          m
2          f          q
```

Using A Trigger To Update (?) A View

- Triggers will allow you to do very strange things

Using A Trigger To Update (?) A View

-
  ```
  CREATE TRIGGER trigger03
  INSTEAD OF UPDATE ON t
  REFERENCING NEW AS new
  BEGIN UPDATE r
  SET b = :new.c
  WHERE a = :old.a;
  END trigger03;
  .
  RUN
  ```

-
  ```
  UPDATE t
  SET c = 'q'
  WHERE a = '2';
  ```

Using A Trigger To Update (?) A View

- Tables R, S, and view T before update on the view

```
A          B
---------- ----------
1          e
2          f

A          C
---------- ----------
1          m
2          n
3          o

A          B          C
---------- ---------- ----------
1          e          m
2          f          n
```

Using A Trigger To Update (?) A View

- Tables R, S, and view T after update on the view using trigger03

```
A          B
---------- ----------
1          e
2          q

A          C
---------- ----------
1          m
2          n
3          o

A          B          C
---------- ---------- ----------
1          e          m
2          q          n
```

Modifying Objects

ALTER , DROP, REPLACE

- In general, if an object is CREATEd, in can subsequently be
  - **ALTER**ed (some features are changed)
  - **DROP**ped (removed)

- Allowing such modifications is why it is generally a good idea to name constraints, assertions, triggers, etc, while creating them

Privileges

Privileges

- Privileges can be granted to user or PUBLIC for
  - Operations
  - References

    on
  - Base tables
  - Views
- These are technically part of ***Data Control Language*** or ***DCL***

Types of Privileges

- Data-Specific
  - Select
  - Insert
  - Update
  - Delete
  - References
- Object-Specific
  - Create
  - Drop
  - Alter
- Execution
  - Execute (a PL/SQL procedure)

Examples of Privileges

- A typical instruction is:
  - **GRANT** SELECT, INSERT
    ON Customer
    TO Li, Brown;
- Privileges can be restricted to columns:
  - **GRANT** SELECT
    ON Customer.City
    TO Li, Brown;
- It is possible to grant all privileges by:
  - **GRANT** ALL
    ON Customer
    TO Li, Brown;

Passing Privileges

- It is possible to allow the users to pass the privileges to other users by issuing:
  - **GRANT** SELECT, INSERT
    ON Customer
    TO Li, Brown
    WITH GRANT OPTION;
- Then Li can issue
  - **GRANT** SELECT
    ON Customer.City
    TO JONES;

Privilege To Reference

- It is possible to allow a user to use columns in a table as foreign keys referring to primary keys or UNIQUE in a table to which the user has no privileges:
  - **GRANT ALL**
    ON Invoice
    TO Li;
  - **GRANT REFERENCES** (C)
    ON Customer
    TO Li;
- This privilege must be explicitly granted because Li may be able to check if a particular C appears in Customer
  - To check if C = 123456789 appears in Customer, Li attempts to INSERT an Invoice from C = 123456789
  - If C = 123456789 does not appear in Customer, the database will complain about violation of FOREIGN KEY constraint
  - If C = 123456789 appears in Customer, the database will not complain about violation of FOREIGN KEY constraint
  - This is how Li can check this and that's why this needs to be explicitly permitted if Li can insert tuples into Invoice

Privilege To Reference

- We assumed that C is a semantically meaningful number, such as SSN
  - That’s why we wanted to keep it confidential whether the person with that SSN was in Customer
- If we use an autoincrement primary key the knowledge of the primary key does not provide too much useful semantic information (still: chronology + number of rows)
  - For example, the primary key could be just a row number, as we have done previously
  - Then, by adding tuples to Invoice Li can only learn (useless information) whether there is information about a customer in a particular row of Customer, without learning anything about the customer in that row
- So little information will leak even if we somebody has
  - All the privileges on Invoice
  - The privilege to reference on Customer

Privileges On Views

- It is possible to grant privileges on views.
  - Of course, the privilege must be meaningful. That is a privilege to update can be given only on a view that can be updated, etc.

Revoking Privileges

- Privileges can be revoked

- There are various way to specify what happens with privileges granted by somebody from whom a privilege is taken away

Key Ideas

Key Ideas

- CREATE for defining tables
  - Specifying domains
  - PRIMARY KEY
  - UNIQUE
  - FOREIGN KEY
  - NOT NULL
  - CHECK
  - DEFAULT
- UNKNOWNs
- Maintenance of referential integrity
- Constraint checking
  - NOT DEFERRABLE
  - DEFERRABLE
- ASSERTIONs

Key Ideas

- Auto-generated primary keys
- Triggers on INSERT, UPDATE, DELETE, firing BEFORE, AFTER, INSTEAD
- Views
- Updating a view with an SQL UPDATE
- Updating a view with an INSTEAD TRIGGER
- ALTER, DROP, REPLACE
- Privileges