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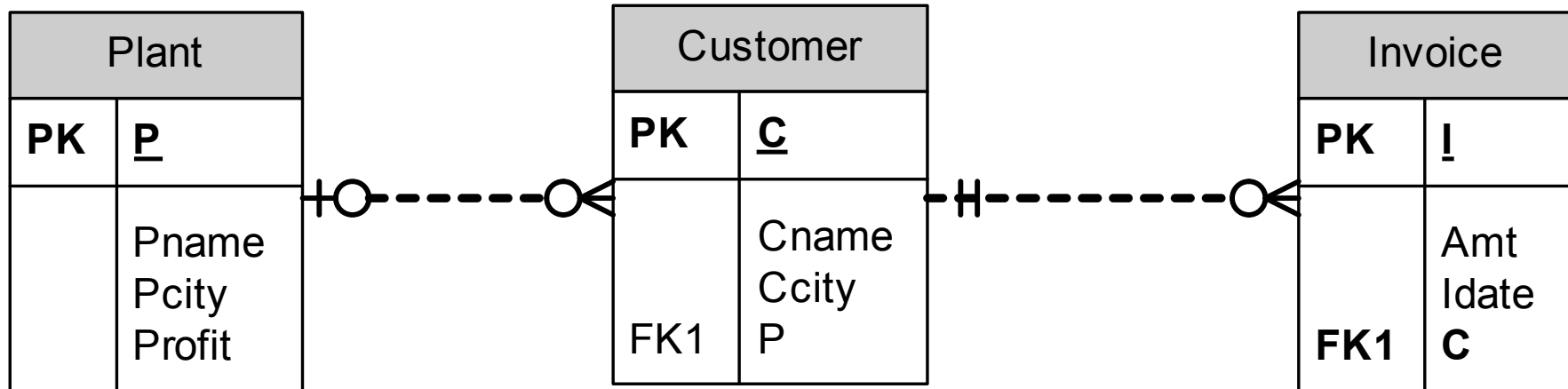## SQL: Data Definition Language
And Data Control Language
For Relational Databases

The Tables To Be Defined
And Some More

Plant	
PK	<u>P</u>
	Pname Pcity Profit

Customer	
PK	<u>C</u>
	Cname Ccity
FK1	P

Invoice	
PK	<u>I</u>
	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 this

Defining A Relational Database

◆ We will see only 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 can only touch on some of them

◆ But enough for defining reasonable complexity databases

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, 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
- 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 DML: UNKNOWN implies "not safisfied"*

14

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 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** ON DELETE SET NULL
 );

◆ P in Customer has to reference the primary key of Plant

◆ 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 **ON DELETE SET NULL**
   );

◆ P in Customer has to reference the primary key of Plant

◆ But note that P in Customer is not required to be NOT NULL

◆ We have a specification that if P listed in some tuple of Customer is deleted from Plant (that is the tuple with this value of primary key is deleted), then that value of P in Plant is automatically 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 ON DELETE CASCADE
  );

◆ NOT NULL can be specified 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 **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 (
   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 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 is copied to the foreign key
  - **SET NULL**: the new value of the foreign key is NULL
  - **SET DEFAULT**: the new value of the foreign key 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
);

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 ON DELETE SET NULL;

- ◆ ALTER TABLE Invoice ADD CONSTRAINT C_90 FOREIGN KEY (C) REFERENCES Customer 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));

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 diagram

Husband	
PK	Hname
FK1	Wname

Wife	
PK	Wname
FK1	Hname
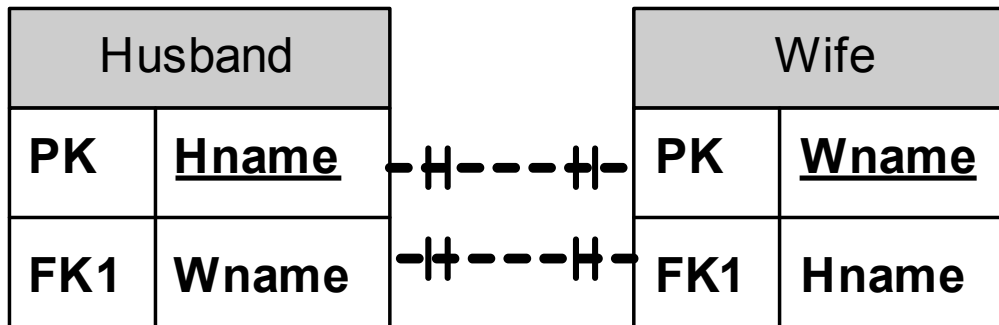
◆ You have "circular" dependencies
- You cannot fully define Husband before Wife
- You cannot fully define Wife before Husband

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

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 divided 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

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 after each statement or at the end of a transaction

- It is done at the end, 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

◆ 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, e.g., Oracle

◆ It is very difficult to implement them both correctly and efficiently

Triggers

◆ These are actions that can be taken before/after/instead INSERT, UPDATE, or DELETE

◆ Triggers are both complex and powerful, we just touch briefly on them here

◆ We will discuss in this unit:
- **AFTER** (next)
- **INSTEAD** (later)

◆ Assume that after a new Customer is inserted into the database, if Cname is Xiu, the system will "automatically" CREATE a new plant in the city Xiu lives, with "properties related to Xiu," which we will understand by looking at the example

◆ Let us look at (tested in Oracle)
- The exact trigger in Oracle
- A partial trace of the execution in Oracle

Defining A Trigger

◆ **CREATE TRIGGER** Trigger01
AFTER INSERT ON Customer
REFERENCING NEW AS newcustomer
FOR EACH ROW
WHEN (newcustomer.Cname = 'Xiu')
BEGIN
INSERT INTO Plant **VALUES**(:newcustomer.C,
'Xiu_Plant', :newcustomer.Ccity, NULL);
END Trigger01;
.
RUN;

◆ This was the exact Oracle syntax

◆ **NEW** refers to added rows

◆ If rows were deleted (not in our example!), we could refer to
them as **OLD**

38

Our Database

◆ Customer and Plant before Insert

```
C           CNAME       CCITY       P
----------  ----------  ----------  ----------
1000        Doe         Boston      901
2000        Yao         Boston      902
3000        Doe         Chicago     903
4000        Doe         Seattle
5000        Brown       Denver      903
6000        Smith       Seattle     907
7000        Yao         Chicago     904
8000        Smith       Denver      904
9000        Smith       Boston      903

P           PNAME       PCITY           PROFIT
----------  ----------  ----------  ----------
901         Alpha       Boston           45000
902         Beta        Boston           56000
903         Beta        Chicago
904         Gamma       Chicago          51000
905         Delta       Denver           48000
906         Epsilon     Miami            51000
907         Beta        Miami            65000
908         Beta        Boston           51000
```

Insertion

◆ INSERT INTO Customer
VALUES(1001,'Xiu','Boston',null);

◆ Note that the INSERT statement could have inserted many tuples into Customer, for instance, if a whole table was inserted into Customer
- We had an example of such "candidate customers" being inserted into Customer, once Good became Yes

Our Database

◆ Customer and Plant after Insert

```
C          CNAME      CCITY      P
---------- ---------- ---------- ----------
1000       Doe        Boston     901
2000       Yao        Boston     902
3000       Doe        Chicago    903
4000       Doe        Seattle
5000       Brown      Denver     903
6000       Smith      Seattle    907
7000       Yao        Chicago    904
8000       Smith      Denver     904
9000       Smith      Boston     903
1001       Xiu        Boston


P          PNAME      PCITY          PROFIT
---------- ---------- ---------- ----------
901        Alpha      Boston          45000
902        Beta       Boston          56000
903        Beta       Chicago
904        Gamma      Chicago         51000
905        Delta      Denver          48000
906        Epsilon    Miami           51000
907        Beta       Miami           65000
908        Beta       Boston          51000
1001       Xiu_Plant  Boston
```

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;
```

◆ 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 procedure call

◆ Therefore we should think of the following procedure for computing the answer to the last query:

  1. The system computes the value of the table GoodPlant

  2. The system executes the query against the table GoodPlant

◆ In actuality, the system may compute the answer differently, 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);

as 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');

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

Views That Are Updateable 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 updateable

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

- No GROUP BY

Some Views That Should Be Updateable

◆ It may make sense to update views that the SQL standard does not allow to update and it is now sometimes permissible; that is, in some implementations

◆ If we have two tables

- R(SSN, Salary)
- S(SSN, Address)

◆ Consider the view

CREATE VIEW RS
AS SELECT R.SSN AS SSN, Salary, Address
FROM R, S
WHERE R.SSN = S.SSN ;

◆ And it is perfectly clear what to do if a new employee is inserted into RS: i.e., how to reflect this in R and in S

Updating Views

◆ SQL prohibits this

◆ But Oracle actually will execute correctly

◆ But Oracle will do very strange things too when you attempt to update views in strange ways

◆ The standard mechanism for updating views when it makes sense uses **INSTEAD** 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';

60

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
```

ALTER , DROP, REPLACE

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

◆ Sometimes it can be
- **REPLACE**d (by a new object)

◆ This is why it is generally a good idea to name constraints, assertions, triggers, etc, while creating them

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

- ◆ Select
- ◆ Insert
- ◆ Update
- ◆ Delete
- ◆ References

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 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 = 1 appears in Customer, Li attempts to INSERT an Invoice from C = 1

- If C = 1 does not appear in Customer, the database will complain about violation of FOREIGN KEY constraint

- If C = 1 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 it needs to be explicitly permitted

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

◆ 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

◆ ASSERTION

Key Ideas

◆ Trigger "on" INSERT, UPDATE, DELETE, "firing" BEFORE, AFTER, INSTEAD

◆ Views

◆ Updating views with SQL UPDATE

◆ Updating views with INSTEAD TRIGGERs

◆ ALTER, DROP, REPLACE

◆ Privileges:
- Select
- Insert
- Update
- Delete
- References