INTEGRITY
CONSTRAINTS
Review

Three things managed by a DBMS

1. Data organization
   - E/R Model
   - Relational Model

2. Data Retrieval
   - Relational Algebra
   - SQL

3. Data Integrity
   - Integrity Constraints
Integrity Constraints

Purpose: prevent semantic inconsistencies in data

<table>
<thead>
<tr>
<th>cname</th>
<th>svngs</th>
<th>check</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>100</td>
<td>200</td>
<td>250</td>
</tr>
</tbody>
</table>

4 kinds of IC’s:

1. Key Constraints
2. Attribute Constraints
3. Referential Integrity Constraints
4. Global Constraints

No entry for Kenmore...???
IC’s

What are they?
- predicates on the database
- must always be true (\(\vdash\), checked whenever db gets updated)

There are the following 4 types of IC’s:

Key constraints (1 table)
- e.g., 2 accts can’t share the same acct_no

Attribute constraints (1 table)
- e.g., 2 accts must have nonnegative balance

Referential Integrity constraints (2 tables)
- E.g. bnames associated w/ loans must be names of real branches

Global Constraints (n tables)
- E.g., a loan must be carried by at least 1 customer with a svngs acct
Key Constraints

Idea: specifies that a relation is a set, not a bag

SQL examples:

1. Primary Key:

   CREATE TABLE branch(
     bname  CHAR(15)  PRIMARY KEY,
     bcity  CHAR(20),
     assets INT);

   or

   CREATE TABLE depositor(
     cname  CHAR(15),
     acct_no CHAR(5),
     PRIMARY KEY(cname, acct_no));

2. Candidate Keys:

   CREATE TABLE customer (  
     ssn   CHAR(9)   PRIMARY KEY,
     cname  CHAR(15),
     address CHAR(30),
     city   CHAR(10),
     UNIQUE (cname, address, city);
Key Constraints

Effect of SQL Key declarations
  PRIMARY (A1, A2, .., An) or
  UNIQUE (A1, A2, ..., An)

Insertions: check if any tuple has same values for A1, A2, .., An as any
  inserted tuple. If found, **reject insertion**
Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)
  1. 1 primary key per table, several unique keys allowed.
  2. Only primary key can be referenced by “foreign key” (ref integrity)
  3. DBMS may treat primary key differently
     (e.g.: create an index on PK)
Attribute Constraints

Idea:

- Attach constraints to values of attributes
- Enhances types system (e.g.: $\geq 0$ rather than integer)

In SQL:

1. NOT NULL
   e.g.: CREATE TABLE branch(
         bname  CHAR(15)  NOT NULL,
         ....
      )
   Note: declaring bname as primary key also prevents null values

2. CHECK
   e.g.: CREATE TABLE depositor(
         ....
         balance int NOT NULL,
         CHECK( balance $\geq 0$),
         ....
      )
   affect insertions, update in affected columns
Attribute Constraints

Domains: can associate constraints with DOMAINS rather than attributes

e.g. instead of:

```
CREATE TABLE depositor(
    ....
    balance INT NOT NULL,
    CHECK (balance >= 0)
)
```

One can write:

```
CREATE DOMAIN bank-balance INT (
    CONSTRAINT not-overdrawn
    CHECK (value >= 0),
    CONSTRAINT not-null-value
    CHECK( value NOT NULL));
```

```
CREATE TABLE depositor (  
    ....
    balance    bank-balance,
)
```

Advantages?
Attribute Constraints

Advantage of associating constraints with domains:

1. can avoid repeating specification of same constraint for multiple columns

2. can name constraints
   e.g.: CREATE DOMAIN bank-balance INT (CONSTRAINT not-overdrawn CHECK (value >= 0), CONSTRAINT not-null-value CHECK( value NOT NULL));

allows one to:
1. add or remove:
   ALTER DOMAIN bank-balance
   ADD CONSTRAINT capped
   CHECK( value <= 10000)

2. report better errors (know which constraint violated)
Referential Integrity Constraints

Idea: prevent “dangling tuples” (e.g.: a loan with a bname, Kenmore, when no Kenmore tuple in branch)

Ref Integrity:
ensure that:
  foreign key value $\rightarrow$ primary key value

(note: need not to ensure $\leftarrow$, i.e., not all branches have to have loans)
Referential Integrity Constraints

Referencing Relation (e.g. loan)

Referenced Relation (e.g. branch)

In SQL:

```sql
CREATE TABLE branch(
    bname CHAR(15) PRIMARY KEY
    ....)

CREATE TABLE loan (
    ....
    FOREIGN KEY bname REFERENCES branch);
```

Affects:

1) Insertions, updates of referencing relation
2) Deletions, updates of referenced relation
Referential Integrity Constraints

what happens when we try to delete this tuple?

Ans: 3 possibilities

1) reject deletion/update

2) set \[ t_i[c], t_j[c] = NULL \]

3) propagate deletion/update
   
   DELETE: delete \[ ti, tj \]
   
   UPDATE: set \[ ti[c], t_j[c] \] to updated values
Referential Integrity Constraints

CREATE TABLE A ( ..... 
FOREIGN KEY c REFERENCES B action 
 .......... )

Action: 1) left blank (deletion/update rejected) (or NO ACTION)

2) ON DELETE SET NULL/ ON UPDATE SET NULL
sets ti[c] = NULL, tj[c] = NULL

3) ON DELETE CASCADE
deletes ti, tj
ON UPDATE CASCADE
sets ti[c], tj[c] to new key values
Global Constraints

Idea: two kinds

1) single relation (constraints spans multiple columns)
   E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

2) multiple relations: CREATE ASSERTION

SQL examples:
   1) single relation: All Bkln branches must have assets > 5M

   CREATE TABLE branch (..........bcity CHAR(15),
                        assets INT,
                        CHECK (NOT(bcity = ‘Bkln’) OR assets > 5M))

   Affects:
      insertions into branch
      updates of bcity or assets in branch
Global Constraints

SQL example:
2) Multiple relations: every loan has a borrower with a savings account

CHECK (NOT EXISTS (  
    SELECT *  
    FROM loan AS L  
    WHERE NOT EXISTS(  
        SELECT *  
        FROM borrower B, depositor D, account A  
        WHERE B.cname = D.cname AND  
        D.acct_no = A.acct_no AND  
        L.lno = B.lno))

Problem: Where to put this constraint? At depositor? Loan? ....

Ans: None of the above:
    CREATE ASSERTION loan-constraint  
    CHECK( ..... )  
    Checked with EVERY DB update!
    very expensive.....
Global Constraints

Issues:

1) How does one decide what global constraint to impose?

2) How does one minimize the cost of checking the global constraints?

Ans: Functional dependencies.

but before we go there
<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Where declared</th>
<th>Affects...</th>
<th>Expense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Constraints</td>
<td>CREATE TABLE (PRIMARY KEY, UNIQUE)</td>
<td>Insertions, Updates</td>
<td>Moderate</td>
</tr>
<tr>
<td>Attribute Constraints</td>
<td>CREATE TABLE CREATE DOMAIN (Not NULL, CHECK)</td>
<td>Insertions, Updates</td>
<td>Cheap</td>
</tr>
<tr>
<td>Referential Integrity</td>
<td>Table Tag (FOREIGN KEY .... REFERENCES ....)</td>
<td>1. Insertions into referencing rel’n 1,2: like key constraints. Another reason to index/sort on the primary keys 3,4: depends on</td>
<td>1,2: like key constraints. Another reason to index/sort on the primary keys 3,4: depends on a. update/delete policy chosen b. existence of indexes on foreign key</td>
</tr>
<tr>
<td>Global Constraints</td>
<td>Table Tag (CHECK) or outside table (CREATE ASSERTION)</td>
<td>1. For single rel’n constraint, with insertion, deletion of relevant attrs 2. For assertions w/ every db modification</td>
<td>1. cheap 2. very expensive</td>
</tr>
</tbody>
</table>
Triggers (Active database)

- **Trigger**: A procedure that starts automatically if specified changes occur to the DBMS
- Analog to a "daemon" that monitors a database for certain events to occur
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run) [Optional]
  - Action (what happens if the trigger runs)
- Semantics:
  - When event occurs, and condition is satisfied, the action is performed.
### Triggers – Event, Condition, Action

- **Events** could be:

  ```sql
  BEFORE|AFTER | INSERT | UPDATE | DELETE ON <tableName>
  ```

  e.g.:  ```sql
  BEFORE INSERT ON Professor
  ```

- **Condition** is SQL expression or even an SQL query (query with non-empty result means **TRUE**)

- **Action** can be many different choices:
  - SQL statements, and even DDL and transaction-oriented statements like “commit”.  

Example Trigger

Assume our DB has a relation schema:

Professor (pNum, pName, salary)

We want to write a trigger that:

Ensures that any new professor inserted has salary $\geq 60000$
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

for what context  

BEGIN

check for violation here 

END;
Example Trigger

CREATE TRIGGER minSalary BEFORE INSERT ON Professor

FOR EACH ROW

BEGIN

Violation of Minimum Professor Salary?

END;
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

FOR EACH ROW

BEGIN

IF (:new.salary < 60000) THEN RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Professor Salary'); END IF;

END IF;

END;
Details of Trigger Example

- **BEFORE INSERT ON Professor**
  - This trigger is checked before the tuple is inserted

- **FOR EACH ROW**
  - specifies that trigger is performed for each row inserted

- **:new**
  - refers to the new tuple inserted

- **If (:new.salary < 60000)**
  - then an application error is raised and hence the row is not inserted; otherwise the row is inserted.

- **Use error code: -20004;**
  - this is in the valid range
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW
WHEN (new.salary < 60000)
BEGIN
    RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Professor Salary');
END;

- Conditions can refer to old/new values of tuples modified by the statement activating the trigger.
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

REFERENCING NEW as newTuple

FOR EACH ROW

WHEN (newTuple.salary < 60000)

BEGIN

    RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Professor Salary');

END;
Example Trigger

CREATE TRIGGER updSalary

    BEFORE UPDATE ON Professor

REFERENCING OLD AS oldTuple NEW as newTuple

FOR EACH ROW

WHEN (newTuple.salary < oldTuple.salary)

BEGIN

    RAISE_APPLICATION_ERROR (-20004, 'Salary Decreasing !!');

END;

- Ensure that salary does not decrease
CREATE TRIGGER youngSailorUpdate
    AFTER INSERT ON SAILORS
    REFERENCING NEW TABLE AS NewSailors
    FOR EACH STATEMENT
    INSERT
        INTO YoungSailors(sid, name, age, rating)
    SELECT sid, name, age, rating
    FROM NewSailors N
    WHERE N.age <= 18
Row vs Statement Level Trigger

- **Row** level: activated once per modified tuple
- **Statement** level: activate once per SQL statement

- **Row** level triggers can access new data, statement level triggers cannot always do that (depends on DBMS).

- **Statement** level triggers will be more efficient if we do not need to make row-specific decisions
Example: Consider a relation schema

\texttt{Account (num, amount)}

where we will allow creation of new accounts only during normal business hours.
CREATE TRIGGER MYTRIG1
BEFORE INSERT ON Account
FOR EACH STATEMENT  --- is default
BEGIN
   IF (TO_CHAR(SYSDATE,'dy') IN ('sat','sun'))
      OR
      (TO_CHAR(SYSDATE,'hh24:mi') NOT BETWEEN '08:00'
       AND '17:00')
   THEN
      RAISE_APPLICATION_ERROR(-20500,'Cannot create new account now !!');
   END IF;
END;
When to use BEFORE/AFTER

- Based on efficiency considerations or semantics.

- Suppose we perform statement-level after insert, then all the rows are inserted first, then if the condition fails, and all the inserted rows must be “rolled back”

- Not very efficient !!
CREATE TRIGGER salaryRestrictions
AFTER INSERT OR UPDATE ON Professor
FOR EACH ROW
BEGIN
IF (INSERTING AND :new.salary < 60000) THEN
   RAISE_APPLICATION_ERROR (-20004, 'below min salary'); END IF;
IF (UPDATING AND :new.salary < :old.salary)
THEN RAISE_APPLICATION_ERROR (-20004, 'Salary Decreasing !!'); END IF;
END;
Summary : Trigger Syntax

CREATE TRIGGER <triggerName>
BEFORE|AFTER INSERT|DELETE|UPDATE
[OF <columnList>] ON <tableName>|<viewName>
[REFERENCING [OLD AS <oldName>] [NEW AS <newName>]]
[FOR EACH ROW] (default is “FOR EACH STATEMENT”)
[WHEN (<condition>)]
<PSM body>;
Constraints versus Triggers

- **Constraints** are useful for database consistency
  - Use IC when sufficient
  - More opportunity for optimization
  - Not restricted into insert/delete/update

- **Triggers** are flexible and powerful
  - Alerters
  - Event logging for auditing
  - Security enforcement
  - Analysis of table accesses (statistics)
  - Workflow and business intelligence ...

- But can be hard to understand ......
  - Several triggers (Arbitrary order → unpredictable !?)
  - Chain triggers (When to stop ?)
  - Recursive triggers (Termination?)
Functional Dependencies

An example:

<table>
<thead>
<tr>
<th></th>
<th>bname</th>
<th>Lno</th>
<th>cname</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Downtown</td>
<td>L-170</td>
<td>Jones</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>Downtown</td>
<td>L-170</td>
<td>Smith</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>Redwood</td>
<td>L-230</td>
<td>Turner</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>Perry</td>
<td>L-234</td>
<td>Hayes</td>
<td>1500</td>
</tr>
<tr>
<td></td>
<td>Redwood</td>
<td>L-13</td>
<td>Johnson</td>
<td>1000</td>
</tr>
</tbody>
</table>

Observe:
tuples with the same value for Lno will always have the same value for amt

We write: \( \text{Lno} \rightarrow \text{amt} \) (Lno “determines” amt, or
amt is functional determined by Lno)

True or false?

\( \text{amt} \rightarrow \text{Lno} \) ?
\( \text{Lno} \rightarrow \text{cname} \) ?
\( \text{Lno} \rightarrow \text{bname} \)?
\( \text{bname} \rightarrow \text{Lno} \) ?

can’t always decide by looking at populated db’s
Functional Dependencies

In general:

A1 A2 .... An → B

set of attributes  single attribute

Formally:

if 2 tuples “agree” on their values for A1, A2,...,An they will also agree on their values for B

Formally:

∀ t, u:

How do we decide what constraints to impose?

Consider loan-info(bname, lno, cname, amt) with FDs: 
\[ \text{lno} \rightarrow \text{bname} \]

How do we ensure that \( \text{lno} \rightarrow \text{bname} \)?

CREATE ASSERTION lno-bname
CHECK ( NOT EXIST
  (SELECT *
   FROM loan-info l1, loan-info l2
   WHERE ?))

? == l1.lno = l2.lno AND l1.bname <> l2.bname

FD’s tell us what global constraints to impose....
How to derive them?

1. Key constraints (e.g.: bname a key for branch
   bname → bname
   bname → bcity
   bname → assets)

   we can write: bname → bname bcity assets

Q: Define “superkeys” in terms of FD’s:
   A: Any set of attributes in a relation that functionally determines all attributes in the relation

Q: Define “candidate key” in terms of FD’s:
   A: Any superkey such that the removal of any attribute leaves a set that does not functionally determine all attributes.
Functional Dependencies

How to derive them?
(1) Key constraints
(2) Laws of physics.... e.g.: time room → course

(3) Trial-and-error...
    Given R=(A, B,C) try each of the following to see if they make sense:
    A→ B   AB→C
    A→C   AC→B          What about:  AB → A ?
    B→A   BC→A          B → B ?
    B→C
    C→A
    C→B

    Just say:
    ...plus all of the trivial dependencies
(2) Avoiding the expense
Recall: Ino \(\rightarrow\) bname preserved by:

\[
\text{CREATE ASSERTION Ino-bname}
\]
\[
\text{CHECK ( NOT EXIST }
\]
\[
\text{(SELECT *}
\]
\[
\text{FROM loan-info l1, loan-info l2}
\]
\[
\text{WHERE l1.Ino = l2.Ino AND}
\]
\[
\text{l1.bname <> l2.bname))}
\]

Is it necessary to have an assertion for every FD’s?

Ans: Luckily, no. Can preprocess FD set
  • some FD’s can be eliminated
  • some FD’s can be combined
Combining FD’s:

a. `cname \rightarrow ccity`

CREATE ASSERTION name-city
CHECK ( NOT EXIST
(SELECT *
FROM customer c1, customer c2
WHERE c1.cname = c2.cname AND
    c1.ccity <> c2.ccity))

b. `cname \rightarrow cstreet`

CREATE ASSERTION name-street
CHECK ( NOT EXIST
(SELECT *
FROM customer c1, customer c2
WHERE c1.cname = c2.cname AND
    c1.cstreet <> c2.cstreet))

combine into: `cname \rightarrow ccity, cstreet`

CREATE ASSERTION name-city-street
CHECK ( NOT EXIST
(SELECT *
FROM customer c1, customer c2
WHERE c1.cname = c2.cname AND
    ??))

?? = ((c1.ccity <> c2.ccity) OR (c1.cstreet <> c2.cstreet))
Determining unnecessary FD’s:

Consider \( \text{cname} \rightarrow \text{cname} \)

CREATE ASSERTION name-name
   CHECK ( NOT EXIST
      (SELECT *
         FROM customer c1, customer c2
         WHERE c1.cname = c2.cname AND
         c1.cname <> c2.cname))

cannot possibly be violated!

Note: \( X \rightarrow Y \) s.t. \( Y \subseteq X \)
a “trivial dependency” (true, regardless of attributes involved)

So: Don’t create assertions for trivial dependencies
Functional Dependencies

Determining unnecessary FD’s:
even non-trivial FD’s can be unnecessary

e.g. a. \( \text{Ino} \rightarrow \text{bname} \)

\[
\text{CREATE ASSERTION Ino-bname}
\]
\[
\text{CHECK ( NOT EXIST}
\]
\[
\text{(SELECT * FROM loan-info } l1, \text{ loan-info } l2
\]
\[
\text{WHERE } l1.\text{Ino} = l2.\text{Ino AND}
\]
\[
\text{l1.bname }\not\sim l2.\text{bname})
\]

b. \( \text{bname} \rightarrow \text{assets} \)

\[
\text{CREATE ASSERTION Ino-bname}
\]
\[
\text{CHECK ( NOT EXIST}
\]
\[
\text{(SELECT *}
\]
\[
\text{FROM loan-info } l1, \text{ loan-info } l2
\]
\[
\text{WHERE } l1.\text{bname} = l2.\text{bname AND}
\]
\[
\text{l1.assets }\not\sim l2.\text{assets})
\]
c. \( lno \rightarrow assets \)

\[
\text{CREATE ASSERTION Ino-bname}
\]

\[
\text{CHECK ( NOT EXIST}
\]

\[
(\text{SELECT} \ *
\]

\[
\text{FROM loan-info l1, loan-info l2}
\]

\[
\text{WHERE l1.Ino} = \ l2.Ino \ \text{AND}
\]

\[
l1.assets \neq l2.assets)
\]

But if (a) and (b) succeed, then c must also