Summary

- Relational Model (Chapter 2)
  - Basics
  - Keys
  - Relational operations
  - Relational algebra basics
- SQL (Chapter 3)
  - Setting up the PostgreSQL database
  - Data Definition (3.2)
  - Basics (3.3-3.5)
  - Null values (3.6)
  - Aggregates (3.7)
  - Advanced operators

Integrity Constraints

- Predicates on the database
- Must always be true (checked whenever db gets updated)

- There are 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., ***accts must have nonnegative balance***
  - Referential Integrity constraints (2 tables)
    E.g. ***bnames*** associated w/ ***loans*** must exist
  - Global Constraints (***n*** tables)
    E.g., all ***loans*** must be carried by at least 1 ***customer*** with a savings acct

Key Constraints

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

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 for tuples with same values for A1, A2, .., An as inserted tuple. If found, **reject**

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.: >= 0 rather than integer)

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 >= 0),
        ....
        )
```

affects insertions, updates in affected columns

Attribute Constraints

Domains:

associate constraints with DOMAINS rather than attributes

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

Associating constraints with domains:

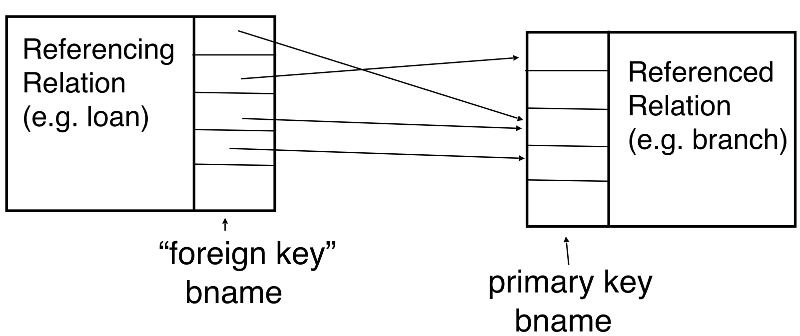
1. can avoid repeating specification of same constraint
   for multiple columns

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));

Advantages:
1. Can add or remove:
   ALTER DOMAIN bank-balance
   ADD CONSTRAINT capped CHECK(value <= 10000)
2. report better errors (know which constraint violated)

Referential Integrity Constraints

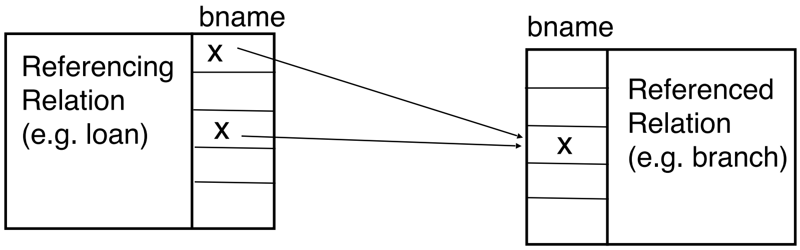
Idea: prevent "dangling tuples" (e.g.: a loan with a bname, *Kenmore*, when no *Kenmore* tuple in branch)



Referential Integrity:
- ensure that local value exists as primary key in other table
- the local value is just a pointer that *refers* to a value in other table

(note: don't need to ensure ←, i.e., not all branches have to have loans)

Referential Integrity Constraints



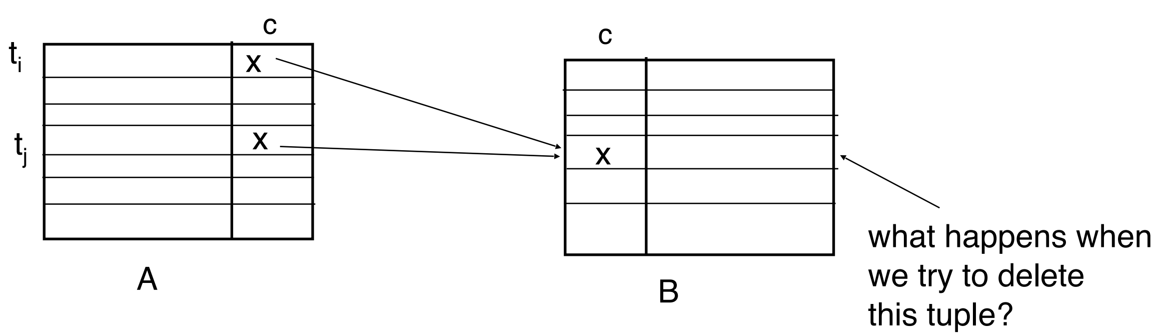
In SQL:

```
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



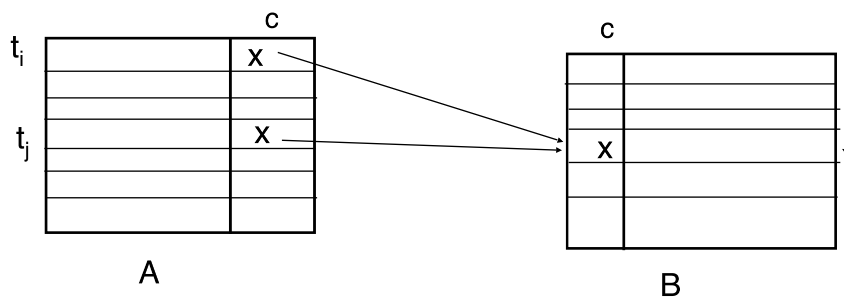
Ans: 3 possibilities

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], tj[c] to updated values

Referential Integrity Constraints



what happens when we try to delete this tuple?

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

Action: 1) left blank (deletion/update rejected)

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

1) single relation (constraints spans multiple columns)

- E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

SQL examples:

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

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.....

Summary: Integrity Constraints

Constraint Type	Where declared	Affects...	Expense
Key Constraints	CREATE TABLE (PRIMARY KEY, UNIQUE)	Insertions, Updates	Moderate
Attribute Constraints	CREATE TABLE CREATE DOMAIN (Not NULL, CHECK)	Insertions, Updates	Cheap
Referential Integrity	Table Tag (FOREIGN KEY .... REFERENCES ....)	1.Insertions into referencing rel'n 2. Updates of referencing rel'n of relevant attrs 3. Deletions from referenced rel'n 4. Update of referenced rel'n	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
Global Constraints	Table Tag (CHECK) or outside table (CREATE ASSERTION)	1. For single rel'n constraint, with insertion, deletion of relevant attrs 2. For assesrtions w/ every db modification	1. cheap 2. very expensive

Today's Plan

- SQL (Chapter 3, 4)
  - Views (4.2)
  - Triggers (5.3)
  - Transactions (4.3)
  - Integrity Constraints (4.4)
  - Functions and Procedures (5.2), Authorization (4.6), Ranking (5.5)
  - Return to / Finishing the Relational Algebra
  - E/R Diagrams

SQL Functions

- Function to count number of instructors in a department

```
create function dept_count (dept_name varchar(20))
returns integer AS $$
begin
    declare d_count  integer;
    select count (* ) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
 end
 $$
```

- Can use in queries:

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

SQL Procedures

- Same function as a procedure in plpgsql:

```
CREATE PROCEDURE dept_count_proc(IN dept_name VARCHAR(20), OUT d_count INTEGER)
  LANGUAGE plpgsql
  AS $$
  BEGIN
    SELECT COUNT(*) INTO d_count
    FROM instructor
    WHERE instructor.dept_name = dept_name;
  END;
  $$;
```

- But use differently:

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

HOWEVER: Syntax can be wildly different across different systems

- Was put in place by DBMS systems before standardization
- Hard to change once customers are already using

We Have Recursion in SQL

- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
    select course_id, prereq_id
    from prereq
  union
    select rec_prereq.course_id, prereq.prereq_id,
    from rec_prereq, prereq
    where rec_prereq.prereq_id = prereq.course_id
  )
select ∗
from rec_prereq;
```

Makes SQL Turing Complete (i.e., you can write any program in SQL)

But: Just because you can, doesn't mean you should

Ranking

- Ranking is done in conjunction with an order by specification.
- Consider: *student_grades(ID, GPA)*

```
Syntax:
RANK() OVER (
    [PARTITION BY partition_expression, ... ]
    ORDER BY sort_expression [ASC | DESC], ...
)
```

- Find the rank of each student.

  select *ID*, **rank**() **over** (**order by** *GPA* **desc**) **as** *s_rank*
  from *student_grades*
  order by *s_rank*

- Equivalent to:

  select *ID*, (1 + (**select count**(*)
  from *student_grades B*
  where *B.GPA* > *A.GPA*)) **as** *s_rank*
  from *student_grades A*
  order by *s_rank*;

Authorization/Security

- GRANT and REVOKE keywords
  - **GRANT privilege_type ON object_type object_name TO role_name;**
  - **GRANT SELECT ON TABLE students TO user1;**
  - **GRANT ALL ON TABLE employees TO user2;**
  - **REVOKE SELECT ON TABLE students FROM user1;**
- Can provide select, insert, update, delete privileges
- Can also create "Roles" and do security at the level of roles
- Some databases support doing this at the level of individual "tuples"
  - MS SQL Server: https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver15
  - PostgreSQL: https://www.postgresql.org/docs/10/ddl-rowsecurity.html

Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  - Transactions are started implicitly and terminated by one of
    - commit work: makes all updates of the transaction permanent in the database
    - rollback work: undoes all updates performed by the transaction.
- Motivating example
  - Transfer of money from one account to another involves two steps:
    - deduct from one account and credit to another
  - If one steps succeeds and the other fails, database is in an inconsistent state
  - Therefore, either both steps should succeed or neither should
- If any step of a transaction fails, all work done by the transaction can be undone by rollback work.
- Rollback of incomplete transactions is done automatically, in case of system failures

Transactions (Cont.)

- In most database systems, each SQL statement that executes successfully is automatically committed.
  - Each transaction would then consist of only a single statement
  - Automatic commit can usually be turned off, allowing multi-statement transactions, but how to do so depends on the database system
  - Another option in SQL:1999: enclose statements within

```
begin atomic
   …
end
```

Triggers

- A ***trigger*** is a statement that is executed automatically by the system as a side effect of a modification to the database.

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - 1. setting the account balance to zero
  - 2. creating a loan in the amount of the overdraft
  - 3. giving this loan a loan number identical to the account number of the overdrawn account

Trigger Example in SQL:1999

create trigger *overdraft-trigger* **after update on** *account*
  referencing new row as *nrow*
  for each row
  when *nrow.balance* < 0
  begin atomic
    actions to be taken
  end

Trigger Example in SQL:1999

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
        insert into borrower
          (select customer-name, account-number
           from depositor
           where nrow.account-number = depositor.account-number);
       insert into loan values
          (nrow.account-number, nrow.branch-name, nrow.balance);
       update account set balance = 0
       where account.account-number = nrow.account-number
end
```


Triggers…

- External World Actions
  - How does the DB ***order*** something if the inventory is low ?
- Syntax
  - Every system has its own syntax
- Careful with triggers
  - Cascading triggers, Infinite Sequences…
- More Info/Examples:
  - https://www.tutorialspoint.com/postgresql/postgresql_triggers.htm
  - Google "create trigger postgresql"

Context

- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Models
  - Conceptual representation of the data
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies