Today’s Plan

- SQL (Chapter 3, 4)
  - Views (4.2)
  - Transactions (4.3)
  - Integrity Constraints (4.4)
  - Triggers (5.3)
  - Functions and Procedures (5.2), Recursive Queries (5.4), Authorization (4.6), Ranking (5.5)

- Some Complex SQL Examples
Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

```
create view v as <query expression>
```

where:

- `<query expression>` is any legal expression

  The view name is represented by `v`

- Can be used in any place a normal table can be used
- For users, there is no distinction in terms of using it
Example Queries

- A view consisting of branches and their customers

```
create view all-customers as
  (select branch-name, customer-name
       from depositor, account
       where depositor.account-number = account.account-number)
union
  (select branch-name, customer-name
       from borrower, loan
       where borrower.loan-number = loan.loan-number)
```

Find all customers of the Perryridge branch
```
select customer-name
  from all-customers
  where branch-name = 'Perryridge'
```
Views

- Is it different from DBMS’s side?
  - Yes; a view may or may not be *materialized*
  - Pros/Cons?

- Updates into views have to be treated differently
  - In most cases, disallowed.
## Views vs Tables

<table>
<thead>
<tr>
<th>Creating</th>
<th>Create view V as (select * from A, B where …)</th>
<th>Create table T as (select * from A, B where …)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used</td>
<td>In any select query. Only some update queries.</td>
<td>It’s a new table. You can do what you want.</td>
</tr>
<tr>
<td>Maintained as</td>
<td>1. Evaluate the query and store it on disk as if a table. 2. Don’t store. Substitute in queries when referenced.</td>
<td>It’s a new table. Stored on disk.</td>
</tr>
<tr>
<td>What if a tuple inserted in A?</td>
<td>1. If stored on disk, the stored table is automatically updated to be accurate. 2. If we are just substituting, there is no need to do anything.</td>
<td>T is a separate table; there is no reason why DBMS should keep it updated. If you want that, you must define a trigger.</td>
</tr>
</tbody>
</table>
Views vs Tables

- Views strictly supercede “create a table and define a trigger to keep it updated”

- Two main reasons for using them:
  - Security/authorization
  - Ease of writing queries
    - E.g. IndividualMedals table
    - The way we are doing it, the IndividualMedals table is an instance of “creating table”, and not “creating view”
    - Creating a view might have been better.

- Perhaps the only reason to create a table is to force the DBMS to choose the option of “materializing”
  - That has efficiency advantages in some cases
  - Especially if the underlying tables don’t change
Update of a View

- Create a view of all loan data in loan relation, hiding the amount attribute
  ```
  create view branch-loan as
  select branch-name, loan-number
  from loan
  ```
- Add a new tuple to branch-loan
  ```
  insert into branch-loan
  values ('Perryridge', 'L-307')
  ```
- This insertion must be represented by the insertion of the tuple
  ```
  ('L-307', 'Perryridge', null)
  ```
  into the loan relation

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.
- Many SQL implementations allow updates only on simple views (without aggregates) defined on a single relation
Today’s Plan

- SQL (Chapter 3, 4)
  - Views (4.2)
  - Transactions (4.3)
  - Integrity Constraints (4.4)
  - Triggers (5.3)
  - Functions and Procedures (5.2), Recursive Queries (5.4), Authorization (4.6), Ranking (5.5)

- Some Complex SQL Examples
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
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**
Today’s Plan

- SQL (Chapter 3, 4)
  - Views (4.2)
  - Transactions (4.3)
  - Integrity Constraints (4.4)
  - Triggers (5.3)
  - Functions and Procedures (5.2), Recursive Queries (5.4), Authorization (4.6), Ranking (5.5)

- Some Complex SQL Examples
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
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
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:**
  - Google: “create trigger” oracle download-uk
Today’s Plan

SQL (Chapter 3, 4)
- Views (4.2)
- Transactions (4.3)
- Integrity Constraints (4.4)
- Triggers (5.3)
- Functions and Procedures (5.2), Recursive Queries (5.4), Authorization (4.6), Ranking (5.5)

Some Complex SQL Examples
Next:

- Integrity constraints
- ??
- Prevent semantic inconsistencies
IC’s

- Predicates on the database
  - Must always be true (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., *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., *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

SQL examples:

1. **Primary Key:**
   CREATE TABLE branch(
     bname  CHAR(15)  PRIMARY KEY,
     bcity      CHAR(20),
     assets    INT);

   or

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

2. **Candidate Keys:**
   CREATE TABLE customer ( 
     ssn     CHAR(9)    PRIMARY KEY,
     cnamen  CHAR(15),
     address CHAR(30),
     city          CHAR(10),
     UNIQUE (cname, address, city));
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)

How would you implement something like this?
Idea:
- Attach constraints to values of attributes
- Enhances types system (e.g.: >= 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 >= 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)
Idea: prevent “dangling tuples” (e.g.: a loan with a bname, Kenmore, when no Kenmore tuple in branch)

Referential Integrity:
ensure that:
foreign key value → primary key value

(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

What happens when we try to delete this tuple?

Ans: 3 possibilities

1) reject deletion/update

2) set \( t_i[c], t_j[c] = \text{NULL} \)

3) propagate deletion/update
   DELETE: delete \( t_i, t_j \)
   UPDATE: set \( t_i[c], t_j[c] \) to updated values
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

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.....
## Summary: Integrity Constraints

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Where declared</th>
<th>Affects...</th>
<th>Expense</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Constraints</strong></td>
<td>CREATE TABLE (PRIMARY KEY, UNIQUE)</td>
<td>Insertions, Updates</td>
<td>Moderate</td>
</tr>
<tr>
<td><strong>Attribute Constraints</strong></td>
<td>CREATE TABLE CREATE DOMAIN (Not NULL, CHECK)</td>
<td>Insertions, Updates</td>
<td>Cheap</td>
</tr>
</tbody>
</table>
| **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 assertions w/ every db modification | 1. cheap  
2. very expensive |
### Summary: Integrity Constraints

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

- SQL (Chapter 3, 4)
  - Views (4.2)
  - Transactions (4.3)
  - Integrity Constraints (4.4)
  - Triggers (5.3)
  - Functions and Procedures (5.2), Recursive Queries (5.4), Authorization (4.6), Ranking (5.5)

- Some Complex SQL Examples
SQL Functions

- Function to count number of instructors in a department
  ```sql
  create function dept_count (dept_name varchar(20))
  returns integer
  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
  ```sql
  select dept_name, budget
  from department
  where dept_count (dept_name) > 12
  ```
SQL Procedures

- Same function as a procedure
  ```sql
  create procedure dept_count_proc (in dept_name varchar(20),
                                   out d_count integer)
  begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
  end
  ```

- But use differently:
  ```sql
  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 it
Recursion in SQL

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

```sql
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: \(\text{student\_grades}(ID, \text{GPA})\)

- Find the rank of each student.

  ```sql
  SELECT ID, rank() OVER (ORDER BY GPA DESC) AS s_rank
  FROM student_grades
  ORDER BY s_rank
  ```

- Equivalent to:

  ```sql
  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 select on instructor to U_1, U_2, U_3`
  - `revoke select on branch from U_1, U_2, U_3`

- 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”
  - PostgreSQL: [https://www.postgresql.org/docs/10/ddl-rowsecurity.html](https://www.postgresql.org/docs/10/ddl-rowsecurity.html)
Today’s Plan

- SQL (Chapter 3, 4)
  - Views (4.2)
  - Transactions (4.3)
  - Integrity Constraints (4.4)
  - Triggers (5.3)
  - Functions and Procedures (5.2), Recursive Queries (5.4), Authorization (4.6), Ranking (5.5)

- Some Complex SQL Examples
Fun with SQL

- [https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/](https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/)
  - Long slide-deck linked off of this page
  - Complex SQL queries showing how to do things like: do Mandelbrot, solve subset sum problem etc.

- The MADlib Analytics Library or MAD Skills, the SQL; [https://arxiv.org/abs/1208.4165](https://arxiv.org/abs/1208.4165)

1. Everything is a Table

```sql
1  SELECT *
2  FROM (SELECT *
3       FROM person
4  ) t
```

Everything is a table. In PostgreSQL, even functions are tables:

```sql
1  SELECT *
2  FROM substring('abcde', 2, 3)
```
2. Recursion can be very powerful

WITH RECURSIVE t(v) AS (  
  SELECT 1  -- Seed Row  
  UNION ALL  
  SELECT v + 1 -- Recursion  
  FROM t  
)  
SELECT v  
FROM t  
LIMIT 5

It yields

```
| v |
---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
```
### 3. Window Functions

```sql
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

<table>
<thead>
<tr>
<th>depname</th>
<th>empno</th>
<th>salary</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>develop</td>
<td>11</td>
<td>5200</td>
<td>5020.0000000000000000000</td>
</tr>
<tr>
<td>develop</td>
<td>7</td>
<td>4200</td>
<td>5020.0000000000000000000</td>
</tr>
<tr>
<td>develop</td>
<td>9</td>
<td>4500</td>
<td>5020.0000000000000000000</td>
</tr>
<tr>
<td>develop</td>
<td>8</td>
<td>6000</td>
<td>5020.0000000000000000000</td>
</tr>
<tr>
<td>develop</td>
<td>10</td>
<td>5200</td>
<td>5020.0000000000000000000</td>
</tr>
<tr>
<td>personnel</td>
<td>5</td>
<td>3500</td>
<td>3700.0000000000000000000</td>
</tr>
<tr>
<td>personnel</td>
<td>2</td>
<td>3900</td>
<td>3700.0000000000000000000</td>
</tr>
<tr>
<td>sales</td>
<td>3</td>
<td>4800</td>
<td>4866.666666666666666667</td>
</tr>
<tr>
<td>sales</td>
<td>1</td>
<td>5000</td>
<td>4866.666666666666666667</td>
</tr>
<tr>
<td>sales</td>
<td>4</td>
<td>4800</td>
<td>4866.666666666666666667</td>
</tr>
</tbody>
</table>

(10 rows)
4. Correlation Coefficient

```sql
SET ARITHABORT ON;

DECLARE @OurData TABLE
(
    x NUMERIC(18,6) NOT NULL,
    y NUMERIC(18,6) NOT NULL
);

INSERT INTO @OurData
    (x, y)
SELECT
    x, y
FROM (VALUES
    (1,32), (1,23), (3,50), (11,37), (-2,39), (10,44), (27,32), (25,16), (20,23),
    (4,5), (30,41), (28,2), (31,52), (29,12), (50,40), (43,18), (10,65), (44,26),
    (35,15), (24,37), (52,66), (59,46), (64,95), (79,36), (24,66), (69,58), (88,56),
    (61,21), (100,60), (62,54), (10,14), (22,40), (52,97), (81,26), (37,58), (93,71),
    (64,82), (24,33), (112,49), (64,90), (53,90), (132,61), (104,35), (60,52),
    (29,50), (85,116), (95,104), (131,37), (139,38), (8,124)
) f(x,y)

SELECT
    ((Sy * Sxx) - (Sx * Sxy))
/ ((N * (Sxx)) - (Sx * Sx)) AS a,
    ((N * Sxy) - (Sx * Sy))
/ ((N * Sxx) - (Sx * Sx)) AS b,
    ((N * Sxy) - (Sx * Sy))
/ SQRT((
    ((N * Sxx) - (Sx * Sx))
    * ((N * Syy) - (Sy * Sy)))) AS r
FROM

(SELECT SUM([@OurData].x) AS Sx, SUM([@OurData].y) AS Sy,
    SUM([@OurData].x * [@OurData].x) AS Sxx,
    SUM([@OurData].x * [@OurData].y) AS Sxy,
    SUM([@OurData].y * [@OurData].y) AS Syy,
    COUNT(*) AS N
FROM @OurData
) sums;
```
5. Page Rank

- Recursive algorithm to assign weights to the nodes of a graph (Web Link Graph)
- Weight for a node depends on the weights of the nodes that point to it
- Typically done in iterations till “convergence”
- Not obvious that you can do it in SQL, but:
  - Each iteration is just a LEFT OUTERJOIN
  - Stopping condition is trickier
- Other ways to do it as well

https://devnambi.com/2013/pagerank.html
declare @DampingFactor decimal(3,2) = 0.85 --set the damping factor
, @MarginOfError decimal(10,5) = 0.001 --set the stable weight
, @TotalNodeCount int
, @IterationCount int = 1

-- we need to know the total number of nodes in the system
set @TotalNodeCount = (select count(*) from Nodes)

-- iterate!
WHILE EXISTS ( select * 
FROM dbo.Notes 
WHERE HasConverged = 0 
)
BEGIN

UPDATE n SET
NodeWeight = 1.0 - @DampingFactor + isnull(x.TransferWeight, 0.0)

-- a node has converged when its existing weight is the same as the weight it would be given
-- (plus or minus the stable weight margin of error)
, HasConverged = case when abs(n.NodeWeight - (1.0 - @DampingFactor + isnull(x.TransferWeight, 0.0))) < @MarginOfError then 1
else 0 end

FROM Nodes n
LEFT OUTER JOIN
(

-- Here's the weight calculation in place
SELECT
   e.TargetNodeId
, TransferWeight = sum(n.NodeWeight / n.NodeCount) * @DampingFactor

FROM Nodes n
INNER JOIN Edges e
ON n.NodeId = e.SourceNodeId
GROUP BY e.TargetNodeId

) as x
ON x.TargetNodeId = n.NodeId

-- for demonstration purposes, return the value of the nodes after each iteration
SELECT @IterationCount as IterationCount
, *
FROM Nodes

set @IterationCount += 1

END