Strings

• The LIKE operator compares strings to patterns
  – An underscore, _, matches any character
    • 'abc' LIKE 'a_c' is true
  – The percent sign, %, matches any sequence of zero or more characters
    • 'abc' LIKE 'a%' is true
  – LIKE always matches the entire string
    • So it is the same as equals if there is not a wildcard in the pattern
      – 'abc' LIKE 'abc' is true
      – 'abc' LIKE 'ab' is false

• NOT LIKE gives the opposite result of LIKE
Strings

• PostgreSQL adds some non-standard options
  – ILIKE does case-insensitive matching
    • 'abc' ILIKE 'AB_' is true
  – SIMILAR TO does regular expression matching
    • SQL standard but not widely supported
    • Regular expression syntax is different from POSIX standard
  – ~, ~*, !~, !~* do regular expression matching using POSIX standard
Strings and Indexes

• Case-insensitive matching (ILIKE, ~*, !~*) cannot use indexes

• Patterns with a wildcard at the beginning cannot use indexes

• Patterns that start with actual characters can use indexes
  – Can use index: WHERE str LIKE 'ab%'
  – Can **NOT** use index: WHERE str LIKE '%ab'
  – Can **NOT** use index: WHERE str ILIKE 'abc'
Functions

• Many DBMSs support the creation and use of functions/procedures
  – No common standard
  – Oracle has PL/SQL
    • Procedural Language/Structured Query Language
  – PostgreSQL has PL/pgSQL
    • Procedural Language/PostgreSQL Structured Query Language
    • Modeled closely on Oracle's PL/SQL
  – Microsoft and Sybase have T-SQL
    • Transact-SQL
Functions

- PostgreSQL actually has three general types of functions
  - Query language functions
    - Written in SQL
  - Procedural language functions
    - Written in PL/pgSQL, PL/Tcl, or some other supported language
  - C-language functions
    - Written in C and compiled to a dynamically-loadable object file outside of the database server
Built-in Functions

• PostgreSQL has a large number of built-in functions
  – Some of these are truly "built-in"
    • Written in C and part of the DBMS
  – Others are just "pre-defined"
    • Written in SQL or PL/pgSQL
Built-in Functions

• We've seen some built-in functions
  – Aggregates
    • count(), avg(), min(), max()
    • Aggregates operate over a row set
Built-in Functions

• Some functions take no arguments
  – Can be used anywhere a value could be used
    • SELECT now();
    • INSERT INTO carts (id, mod_time) VALUES (default, now());
    • UPDATE carts SET mod_time = now() WHERE id = 123;
Built-in Functions

• Some functions take a value or field
  – `SELECT lower(name) FROM users;`
  • `lower()` converts a string to lowercase

• Need to be careful when used in the WHERE clause
  – Can make a query not able to use an index
  – Can **NOT** use index: `WHERE lower(name) = 'tom';`
Creating Functions

• Basic function creation is as show below
  – Many options exist
  – Different languages can be used
  – Entire function body is a string, must be quoted

    CREATE FUNCTION name(args) RETURNS type AS 'body'
    ' LANGUAGE lang;
Creating Functions

- In order to avoid having to escape quotation marks, $$ is often used for quoting

```sql
CREATE FUNCTION name(args) RETURNS type AS $$
    body
$$ LANGUAGE lang;
```
Creating Functions

• CREATE OR REPLACE can be used to avoid having to drop an existing function before re-creating
  – Also keeps function attributes intact
  – Arguments and return types cannot be changed via CREATE OR REPLACE
• Must drop function and CREATE

CREATE OR REPLACE FUNCTION name(args) RETURNS type AS $$
body
$$ LANGUAGE lang;
SQL Functions

• SQL functions execute a list of SQL statements

```sql
CREATE FUNCTION clean_employees() RETURNS void AS $$
    DELETE FROM employees WHERE salary < 0;
$$ LANGUAGE SQL;
```

• Can call the function using SELECT

```sql
SELECT clean_employees();
```
SQL Functions

- Arguments are referenced using $1, $2, etc.
- Last selected value is returned

CREATE FUNCTION debit(integer, numeric) RETURNS numeric AS $$
  UPDATE bank
  SET balance = balance - $2
  WHERE accountno = $1;

  SELECT balance FROM bank WHERE accountno = $1;
$$ LANGUAGE SQL;

SELECT debit(17, 50.0);
SQL Functions

- Arguments can be composite (row) types

```sql
CREATE TABLE emp (  
    name text,  
    salary numeric,  
    age integer  
);

INSERT INTO emp VALUES ('Bill', 4200, 45);

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$  
    SELECT $1.salary * 2 AS salary;  
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream FROM emp  
    WHERE emp.name = 'Bill';

<table>
<thead>
<tr>
<th>name</th>
<th>dream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill</td>
<td>8400</td>
</tr>
</tbody>
</table>
SQL Functions

• Can also return composite types

CREATE FUNCTION new_emp() RETURNS emp AS $$
SELECT text 'None' AS name,
    1000.0 AS salary,
    25 AS age;
$$ LANGUAGE SQL;

SELECT new_emp();

     new_emp
--------------------------
   (None,1000.0,25)

SELECT * FROM new_emp();

    name | salary | age
-----------
    None | 1000.0 | 25
SQL Functions

• The arguments can be given names, but the names are ignored for SQL functions
  – Still need to use $1, $2, etc.
  – Other types of functions can reference arguments by name

CREATE FUNCTION debit(account_no integer, debit_amount numeric)
RETURNS numeric AS $$
UPDATE bank
  SET balance = balance - $2
WHERE accountno = $1;

SELECT balance FROM bank WHERE accountno = $1;
$$ LANGUAGE SQL;
SQL Functions

• SQL functions can have OUT parameters
  – OUT parameters are set rather than read by the function
  – SQL functions do not explicitly set the OUT parameters

  • Columns and types of result of last query must match the OUT parameters

CREATE FUNCTION sum_and_product (x int, y int, OUT sum int, OUT product int) AS 'CREATE FUNCTION sum_and_product (x int, y int, OUT sum int, OUT product int) AS 'SELECT $1 + $2, $1 * $2
  ' LANGUAGE SQL;

SELECT * FROM sum_and_product(11, 42);

<table>
<thead>
<tr>
<th>sum</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>462</td>
</tr>
</tbody>
</table>
SQL Functions

• Arguments can be given default values

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3) RETURNS int AS $$
  SELECT $1 + $2 + $3;
$$ LANGUAGE SQL;

SELECT foo(10, 20, 30);

  foo
-----
  60

SELECT foo(10, 20);

  foo
-----
  33

SELECT foo();
ERROR: function foo() does not exist
```
SQL Functions

- SQL functions can be used in the FROM clause of queries

```sql
CREATE TABLE foo (  
    fooid int,  
    foosubid int,  
    fooname text
);

INSERT INTO foo VALUES (1, 1, 'Joe');  
INSERT INTO foo VALUES (1, 2, 'Ed');  
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

<table>
<thead>
<tr>
<th>fooid</th>
<th>foosubid</th>
<th>fooname</th>
<th>upper</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Joe</td>
<td>JOE</td>
</tr>
</tbody>
</table>
```
SQL Functions

• Note that in the previous example the function only returned one row
  – To return all the rows of the last query in the function, declare the function as returning a set

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
  SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

<table>
<thead>
<tr>
<th>fooid</th>
<th>foosubid</th>
<th>fooname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Ed</td>
</tr>
</tbody>
</table>
Views

- A view is like a read-only table defined by a query

CREATE VIEW myview AS
  SELECT city, temp_lo, temp_hi, prcp, date, location
  FROM weather, cities
  WHERE city = 'Saint Louis';
Views

• Views can be used in queries as if they were tables

```
SELECT * FROM myview;
```

• Views are computed "on-the-fly" when used
  – The query defining the view is basically included as an inner select when executing the calling query

• Views can be used almost anywhere a table can be used
  – And views can be built from other views
Triggers

• Triggers allow functions to automatically be called when certain events happen on a table
  – Can be defined as happening BEFORE or AFTER an INSERT, UPDATE, or DELETE
  – Can execute the triggered function per row or per statement
    • e.g., if a DELETE statement would delete 10 rows then a trigger defined as
      – FOR EACH ROW would execute 10 time
      – FOR EACH STATEMENT would execute once
Triggers

- *AT* must be BEFORE, AFTER, or INSTEAD OF
- *EVENT* must be INSERT, UPDATE, DELETE, or TRUNCATE
- *TYPE* must be ROW or STATEMENT
- The WHEN clause is optional
- SQL function cannot be used. Must use PL/pgSQL (or another supported language) instead

```sql
CREATE TRIGGER name AT EVENT
  ON table
  FOR EACH TYPE
  WHEN condition
  EXECUTE PROCEDURE function(args);
```
Triggers

• Example: when an update is done on the accounts table, call function check_account_update() for each affected row

CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
Triggers

- Update triggers can be restricted to only fire when certain columns are updated
  – Note the addition of "OF balance"

CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
Triggers

• The WHEN clause can be used to check if something has changed
  – OLD and NEW can be used to refer to the old values and new values for each row

```sql
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE PROCEDURE check_account_update();
```
PL/pgSQL

• Declaration of PL/pgSQL functions (name, arguments, return type, language tag) is similar to SQL functions
  – But PL/pgSQL can do much more than execute a simple list of SQL statements
    • Includes conditional branching, loops, etc.
  – Unlike SQL functions, argument names can be used to refer to arguments instead of just $1, $2, etc.
PL/pgSQL

- PL/pgSQL is a block structured language
  - The complete body of a function must be a block as shown below
  - The labels and DECLARE section are optional

```sql
<<label>>
DECLARE
    declarations
BEGIN
    statements
END label;
```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity; -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity; -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity; -- Prints 50
    END;
RAISE NOTICE 'Quantity here is %', quantity; -- Prints 50
RETURN quantity;
END;
$$ LANGUAGE plpgsql;
OUT Parameters

- OUT parameters can be set instead of just returning a matching record

```
CREATE FUNCTION sum_and_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```
Assignment

• variable := expression;

  tax := subtotal * 0.06;
  my_record.user_id := 20;
If

• The ELSIF and ELSE parts are optional

    IF boolean-expression THEN
        statements
    ELSIF boolean-expression THEN
        statements
    ELSIF boolean-expression THEN
        statements
    ...
    ELSE
        statements
    END IF;
Case

• Only one WHEN is required
• ELSE is optional

CASE search-expression
  WHEN expression, expression ... THEN
  statements
  WHEN expression, expression ... THEN
  statements
...
ELSE
  statements
END CASE;
PL/pgSQL

- PL/pgSQL also includes loops, returns, exceptions, ways to execute SQL statements, and many other features
Note that RETURN just breaks out of the current block, it does not exit the function — unless called at the outermost level

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS $$
BEGIN
  LOOP
    -- first try to update the key
    UPDATE db SET b = data WHERE a = key;
    IF found THEN
      RETURN;
    END IF;
    -- not there, so try to insert the key
    -- if someone else inserts the same key concurrently, -- we could get a unique-key failure
    BEGIN
      INSERT INTO db(a,b) VALUES (key, data);
      RETURN;
    EXCEPTION WHEN unique_violation THEN
      -- Do nothing, and loop to try the UPDATE again.
    END;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```
Trigger Functions

• A function to be used as a trigger must be defined as taking no arguments and returning type trigger

• When called as a trigger, several special variables are defined, including
  – NEW: holds the new row data for INSERT/UPDATE row-level triggers
  – OLD: holds the old row data for UPDATE/DELETE row-level triggers
    • Both NEW and OLD are null for statement-level triggers or triggers of the wrong type
CREATE TABLE emp (  
    empname text, salary integer, last_date timestamp, last_user text  
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
BEGIN
  -- Check that empname and salary are given
  IF NEW.empname IS NULL THEN
    RAISE EXCEPTION 'empname cannot be null';
  END IF;
  IF NEW.salary IS NULL THEN
    RAISE EXCEPTION '% cannot have null salary', NEW.empname;
  END IF;

  -- Who works for us when she must pay for it?
  IF NEW.salary < 0 THEN
    RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
  END IF;

  -- Remember who changed the payroll when
  NEW.last_date := current_timestamp;
  NEW.last_user := current_user;
  RETURN NEW;
END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();