

Object-Relational Databases and OR Extensions

University of California, Berkeley School of Information IS 257: Database Management

Lecture Outline



- Object-Relational DBMS
 - OR features in Oracle and MySQL
 - Functions and Triggers
 - OR features in PostgreSQL
- Extending OR databases (examples from PostgreSQL)

Lecture Outline



- Object-Relational DBMS
 - OR features in Oracle and MySQL
 - Functions and Triggers
 - OR features in PostgreSQL
- Extending OR databases (examples from PostgreSQL)

Object Relational Databases



- Background
- Object Definitions
 - inheritance
- User-defined datatypes
- User-defined functions

Object Relational Databases



- Began with UniSQL/X unified object-oriented and relational system
- Some systems (like OpenODB from HP) were Object systems built on top of Relational databases
- Postgres project at Berkeley
- Miro/Montage/Illustra built on Postgres.
- Informix Buys Illustra. (DataBlades)
- Oracle Hires away Informix Programmers. (Cartridges)
- Informix bought by IBM (you can still get it)

Object Relational Data Model



- Class, instance, attribute, method, and integrity constraints
 - Class ≈ Relation, instance ≈ tuple, attribute...
- OID per instance
- Encapsulation
- Multiple inheritance hierarchy of classes
- Class references via OID object references
- Set-Valued attributes
- Abstract Data Types

Object Relational Extended SQL (Illustra)



- CREATE TABLE tablename {OF TYPE Typename}|{OF NEW TYPE typename} (attr1 type1, attr2 type2,...,attrn typen) {UNDER parent_table_name};
- CREATE TYPE typename (attribute_name type_desc, attribute2 type2, ..., attrn typen);
- CREATE FUNCTION functionname (type_name, type_name) RETURNS type_name AS sql_statement

Object-Relational SQL in ORACLE



 CREATE (OR REPLACE) TYPE typename AS OBJECT (attr_name, attr_type, ...);

CREATE TABLE OF typename;

Example



- CREATE TYPE ANIMAL_TY AS OBJECT (Breed VARCHAR2(25), Name VARCHAR2(25), Birthdate DATE);
 - Creates a new type
- CREATE TABLE Animal of Animal_ty;
 - Creates "Object Table"

Constructor Functions



- INSERT INTO Animal values
 (ANIMAL_TY('Mule', 'Frances',
 TO_DATE('01-APR-1997', 'DD-MM-YYYY')));
 - Insert a new ANIMAL_TY object into the table,
 i.e., the type name is a "constructor"

Selecting from an Object Table



Just use the columns in the object...

SELECT Name from Animal;

More Complex Objects



- CREATE TYPE Address_TY as object (Street VARCHAR2(50), City VARCHAR2(25), State CHAR(2), zip NUMBER);
- CREATE TYPE Person_TY as object (Name VARCHAR2(25), Address ADDRESS_TY);
- CREATE TABLE CUSTOMER (Customer_ID NUMBER, Person PERSON_TY);

What Does the Table Look like?



- DESCRIBE CUSTOMER;
- NAME
 TYPE
- -----
- CUSTOMER_ID
- PERSON

NUMBER

NAMED TYPE

Inserting



INSERT INTO CUSTOMER VALUES (1, PERSON_TY('John Smith', ADDRESS_TY('57 Mt Pleasant St.', 'Finn', 'NH', 111111)));

Selecting from Abstract Datatypes



- SELECT Customer_ID from CUSTOMER;
- SELECT * from CUSTOMER;

```
CUSTOMER_ID PERSON(NAME, ADDRESS(STREET, CITY, STATE ZIP))

1 PERSON_TY('JOHN SMITH', ADDRESS_TY('57...
```

Selecting from Abstract Datatypes



- SELECT Customer_id, person.name from Customer;
- SELECT Customer_id, person.address.street from Customer;

Updating



UPDATE Customer SET
 person.address.city = 'HART' where
 person.address.city = 'Briant';

MySQL



- So far, no data type definitions in MySQL
 - But would not be surprised to see them before too long
 - There are already spatial extensions and types
 - User-defined data types are in the current SQL standard, so they will probably make it into MySQL eventually
 - But user-defined functions and triggers are in MySQL

User-Defined Functions (Oracle)



 CREATE [OR REPLACE] FUNCTION function (argname [IN | OUT | IN OUT] datatype ...) RETURN datatype (IS | AS) {block | external body}

Example



Create Function BALANCE_CHECK
(Person_name IN Varchar2) RETURN NUMBER is BALANCE NUMBER(10,2)

BEGIN

```
SELECT sum(decode(Action, 'BOUGHT', Amount, 0)) - sum(decode(Action, 'SOLD', amount, 0)) INTO BALANCE FROM LEDGER where Person = PERSON_NAME;
RETURN BALANCE;
END;
```

Example



Select NAME, BALANCE_CHECK(NAME) from Worker;

Would return the name and balance for each worker

Functions and Procedures - MySQL



- CREATE [DEFINER = { user | CURRENT_USER }]
 PROCEDURE sp_name ([proc_parameter[,...]])
 [characteristic ...] routine_body
- CREATE [DEFINER = { user | CURRENT_USER }]
 FUNCTION sp_name ([func_parameter[,...]]) RETURNS type [characteristic ...] routine_body
- proc_parameter: [IN | OUT | INOUT] param_name type
- func_parameter: param_name type
- type: Any valid MySQL data type
- charactéristic: LÁNGUAGE ŚQL | [NOT]
 DETERMINISTIC | { CONTAINS SQL | NO SQL |
 READS SQL DATA | MODIFIES SQL DATA } | SQL
 SECURITY { DEFINER | INVOKER } | COMMENT 'string'
- routine_body: Valid SQL procedure statement

Defining a MySQL procedure



```
mysql> delimiter //
mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
-> BEGIN
-> SELECT COUNT(*) INTO param1 FROM t;
-> END//
Query OK, 0 rows affected (0.00 sec)
mysql> delimiter;
mysql> CALL simpleproc(@a);
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @a;
+----+
 (a)a
+----+
3
+----+
1 row in set (0.00 \text{ sec})
```

Defining a MySQL Function



mysql> CREATE FUNCTION hello (s CHAR(20))
RETURNS CHAR(50) DETERMINISTIC

-> RETURN CONCAT('Hello, ',s,'!');

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT hello('world');

```
+-----+
| hello('world') |
+-----+
| Hello, world! |
+-----+
1 row in set (0.00 sec)
```

DETERMINISTIC means the function always produces the same result for the same input parameters

TRIGGERS



A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database.

TRIGGERS (Oracle)



 Create TRIGGER UPDATE_LODGING INSTEAD OF UPDATE on WORKER_LODGING for each row

BEGIN

```
if :old.name <> :new.name then update
  worker set name = :new.name where
  name = :old.name;
end if;
if :old.lodging <> ... etc...
```

Triggers in MySQL



CREATE

```
[DEFINER = { user | CURRENT_USER }]
TRIGGER trigger_name trigger_time
trigger_event
ON tbl_name FOR EACH ROW
trigger_stmt
```

- trigger_event can be INSERT, UPDATE, or DELETE
- trigger_time can be BEFORE or AFTER.

Triggers in MySQL



```
CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE test4( a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY, b4 INT DEFAULT
delimiter |
CREATE TRIGGER testref BEFORE INSERT ON test1 FOR EACH ROW
  BEGIN
     INSERT INTO test2 SET a2 = NEW.a1;
     DELETE FROM test3 WHERE a3 = NEW.a1;
     UPDATE test4 SET b4 = b4 + 1 WHERE a4 =
  NEW.a1;
END I
delimiter;
```

Triggers in MySQL (cont)



```
mysgl> INSERT INTO test3 (a3) VALUES
(NULL), (NULL), (NULL), (NULL), (NULL), (NULL), (NULL), (NULL),
(NULL);
mysql> INSERT INTO test4 (a4) VALUES (0), (0), (0), (0), (0), (0), (0),
(0), (0);
mysql> INSERT INTO test1 VALUES
  -> (1), (3), (1), (7), (1), (8), (4), (4);
mysql> SELECT * FROM test1;
+----+
| a1
+____+
  3
  8
  4
```

Triggers in MySQL (cont.)



```
mysql> SELECT * FROM test2;
```

```
+----+
a2
+----+
  3
```

```
mysql> SELECT * FROM test3;
```

```
+----+

| a3 |

+----+

| 2 |

| 5 |

| 6 |

| 9 |

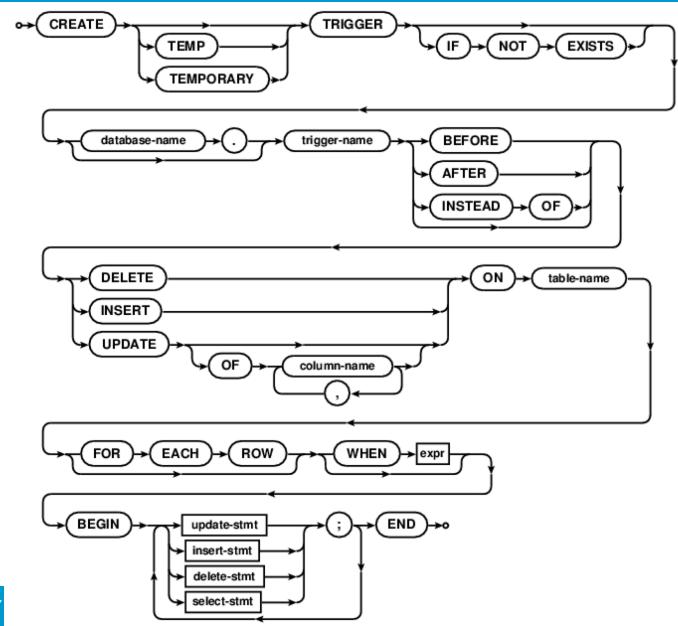
| 10 |

+----+
```

```
mysql> SELECT * FROM test4;
+---+
| a4 | b4
     3 |
```

Triggers in SQLite





Lecture Outline



- Object-Relational DBMS
 - OR features in Oracle and MySQL
 - OR features in PostgreSQL
- Extending OR databases (examples from PostgreSQL)

PostgreSQL



- Derived from POSTGRES
 - Developed at Berkeley by Mike Stonebraker and his students (EECS) starting in 1986
- Postgres95
 - Andrew Yu and Jolly Chen adapted
 POSTGRES to SQL and greatly improved the code base
- PostgreSQL
 - Name changed in 1996, and since that time the system has been expanded to support all SQL standard features, plus unique extensions

PostgreSQL Classes



 The fundamental notion in Postgres is that of a class, which is a named collection of object instances. Each instance has the same collection of named attributes, and each attribute is of a specific type. Furthermore, each instance has a permanent object identifier (OID) that is unique throughout the installation. Because SQL syntax refers to tables, we will use the terms table and class interchangeably. Likewise, an SQL row is an instance and SQL columns are attributes.

Creating a Class



 You can create a new class by specifying the class name, along with all attribute names and their types:

```
CREATE TABLE weather (
city varchar(80),
temp_lo int, -- low temperature
temp_hi int, -- high temperature
prcp real, -- precipitation
date date
);
```

PostgreSQL



- Postgres can be customized with an arbitrary number of user-defined data types.
 Consequently, type names are not syntactical keywords, except where required to support special cases in the SQL92 standard.
- So far, the Postgres CREATE command looks exactly like the command used to create a table in a traditional relational system. However, we will presently see that classes have properties that are extensions of the relational model.

PostgreSQL



- All of the usual SQL commands for creation, searching and modifying classes (tables) are available. With some additions...
- Inheritance
- Non-Atomic Values
- User defined functions and operators



```
CREATE TABLE cities (
              text,
  name
  population
            float,
  altitude int -- (in ft)
CREATE TABLE capitals (
  state
             char(2)
) INHERITS (cities);
```



```
ray=# create table cities (name varchar(50), population float,
     altitude int);
CREATE TABLE
ray=# \d cities
           Table "public.cities"
  Column | Type | Modifiers
name | character varying(50) |
population | double precision
altitude | integer
ray=# create table capitals (state char(2)) inherits (cities);
CREATE TABLE
ray=# \d capitals
          Table "public.capitals"
  Column |
                   Type | Modifiers
name | character varying (50) |
population | double precision
 altitude | integer
 state | character(2)
Inherits: cities
```



- In Postgres, a class can inherit from zero or more other classes.
- A query can reference either
 - all instances of a class
 - or all instances of a class plus all of its
 descendants



 For example, the following query finds all the cities that are situated at an attitude of 500ft or higher:

```
SELECT name, altitude
  FROM cities
 WHERE altitude > 500;
+----+
|name | altitude |
+----+
|Las Vegas | 2174
+----+
|Mariposa | 1953
 _____+
```



 On the other hand, to find the names of all cities, including state capitals, that are located at an altitude over 500ft, the query is:

```
SELECT c.name, c.altitude
FROM cities* c
WHERE c.altitude > 500;
```

which returns:

```
+-----+
|name | altitude |
+-----+
|Las Vegas | 2174
+-----+
|Mariposa | 1953 |
+-----+
|Madison | 845 |
+------+
```



- The "*" after cities in the preceding query indicates that the query should be run over cities and all classes below cities in the inheritance hierarchy
- Many of the PostgreSQL commands (SELECT, UPDATE and DELETE, etc.) support this inheritance notation using "*"

Non-Atomic Values



- One of the tenets of the relational model is that the attributes of a relation are atomic
 - I.e. only a single value for a given row and column (I.e., 1st Normal Form)
- Postgres does not have this restriction: attributes can themselves contain subvalues that can be accessed from the query language
 - Examples include arrays and other complex data types.

Non-Atomic Values - Arrays



 Postgres allows attributes of an instance to be defined as fixed-length or variable-length multidimensional arrays. Arrays of any base type or user-defined type can be created. To illustrate their use, we first create a class with arrays of base types.

Non-Atomic Values - Arrays



- The preceding SQL command will create a class named SAL_EMP with a text string (name), a one-dimensional array of int4 (pay_by_quarter), which represents the employee's salary by quarter and a two-dimensional array of text (schedule), which represents the employee's weekly schedule
- Now we do some INSERTSs; note that when appending to an array, we enclose the values within braces and separate them by commas.

Inserting into Arrays



```
INSERT INTO SAL EMP
  VALUES ('Bill',
  '{10000, 10000, 10000, 10000}',
  '{{"meeting", "lunch"}, {}}');
INSERT INTO SAL EMP
  VALUES ('Carol',
  '{20000, 25000, 25000, 25000}',
  '{{"talk", "consult"}, {"meeting"}}');
```

Querying Arrays



 This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name
  FROM SAL EMP
  WHERE SAL EMP.pay_by_quarter[1] <>
  SAL EMP.pay_by_quarter[2];
+----+
name
|Carol |
```

Querying Arrays



 This query retrieves the third quarter pay of all employees:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;
+-----+
|pay_by_quarter |
+-----+
|10000 |
+-----+
|25000 |
+-----+
```

Querying Arrays



 We can also access arbitrary slices of an array, or subarrays. This query retrieves the first item on Bill's schedule for the first two days of the week.

Lecture Outline



- Object-Relational DBMS
 - OR features in Oracle
 - OR features in PostgreSQL
- Extending OR databases (examples from PostgreSQL)

PostgreSQL Extensibility



- Postgres is extensible because its operation is catalogdriven
 - RDBMS store metadata, or information about databases, tables, columns, etc., in what are commonly known as system catalogs.
 (Some systems call this the data dictionary).
- One key difference between Postgres and standard RDBMS is that Postgres stores much more information in its catalogs
 - not only information about tables and columns, but also information about its types, functions, access methods, etc.
- These classes can be modified by the user, and since Postgres bases its internal operation on these classes, this means that Postgres can be extended by users
 - By comparison, conventional database systems can only be extended by changing hardcoded procedures within the DBMS or by loading modules specially-written by the DBMS vendor.

Postgres System Catalogs



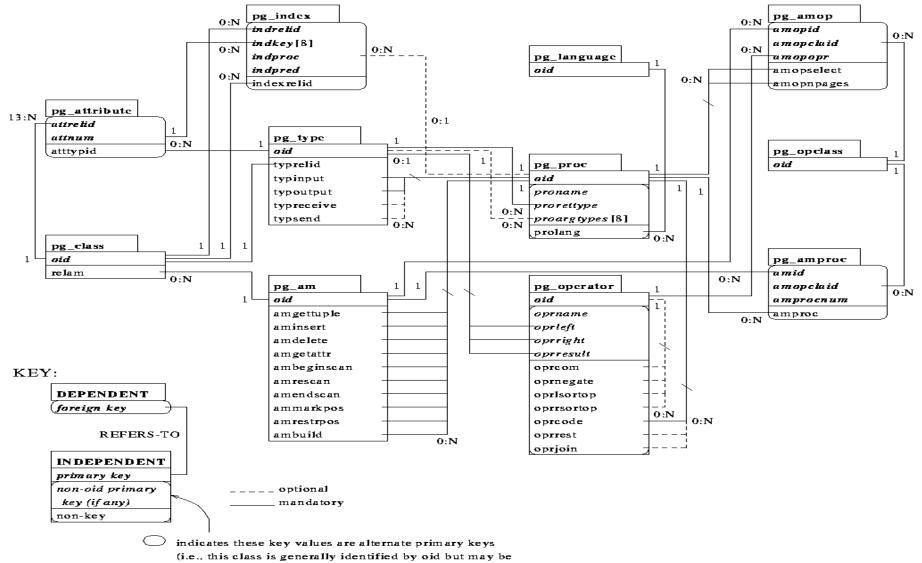


Figure 3. The major POSTGRES system catalogs.

identified by the non-oid primary key in other contexts).

User Defined Functions



 CREATE FUNCTION allows a Postgres user to register a function with a database.
 Subsequently, this user is considered the *owner* of the function

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
RETURNS rtype
AS {SQLdefinition}
LANGUAGE 'langname'
[ WITH ( attribute [, ...] ) ]

CREATE FUNCTION name ( [ ftype [, ...] ] )
RETURNS rtype
AS obj_file , link_symbol
LANGUAGE 'C'
[ WITH ( attribute [, ...] ) ]
```

Simple SQL Function



CREATE FUNCTION one() RETURNS int4
 AS 'SELECT 1 AS RESULT'
 LANGUAGE 'sql';

SELECT one() AS answer;

answer

1

A more complex function



 To illustrate a simple SQL function, consider the following, which might be used to debit a bank account:

```
create function TP1 (int4, float8) returns int4
as 'update BANK set balance = BANK.balance - $2
where BANK.acctountno = $1;
select balance from bank
where accountno = $1; ' language 'sql';
```

 A user could execute this function to debit account 17 by \$100.00 as follows:

```
select (x = TP1(17,100.0));
```

SQL Functions on Composite Types



 When creating functions with composite types, you have to include the attributes of that argument. If EMP is a table containing employee data, (therefore also the name of the composite type for each row of the table) a function to double salary might be...

```
CREATE FUNCTION double_salary(EMP) RETURNS integer AS 'SELECT $1.salary * 2 AS salary; 'LANGUAGE SQL;
```

```
SELECT name, double_salary(EMP) AS dream FROM EMP WHERE EMP.cubicle ~= point '(2,1)';
```

name | dream

----+-----

Sam | 2400

Notice the use of the syntax \$1.salary to select one field of the argument row value. Also notice how the calling SELECT command uses a table name to denote the entire current row of that table as a composite value.

SQL Functions on Composite Types



 It is also possible to build a function that returns a composite type. This is an example of a function that returns a single EMP row:

CREATE FUNCTION new_emp() RETURNS EMP

AS 'SELECT text "None" AS name, 1000 AS salary, 25 AS age,

point "(2,2)" AS cubicle; LANGUAGE SQL;

External Functions



 This example creates a C function by calling a routine from a user-created shared library. This particular routine calculates a check digit and returns TRUE if the check digit in the function parameters is correct. It is intended for use in a CHECK contraint.

```
CREATE FUNCTION ean_checkdigit(bpchar, bpchar) RETURNS bool
AS '/usr1/proj/bray/sql/funcs.so' LANGUAGE 'c';
CREATE TABLE product (
id char(8) PRIMARY KEY,
eanprefix char(8) CHECK (eanprefix ~ '[0-9]{2} [0-9]{5}')
REFERENCES brandname(ean_prefix),
eancode char(6) CHECK (eancode ~ '[0-9]{6}'),
CONSTRAINT ean CHECK (ean_checkdigit(eanprefix, eancode)));
```

Creating new Types



 CREATE TYPE allows the user to register a new user data type with Postgres for use in the current data base. The user who defines a type becomes its owner. typename is the name of the new type and must be unique within the types defined for this database.

```
CREATE TYPE typename ( INPUT = input_function, OUTPUT = output_function
, INTERNALLENGTH = { internallength | VARIABLE } [,
EXTERNALLENGTH = { externallength | VARIABLE } ]
[, DEFAULT = "default" ]
[, ELEMENT = element ] [, DELIMITER = delimiter ]
[, SEND = send_function ] [, RECEIVE = receive_function ]
[, PASSEDBYVALUE ] )
```

New Type Definition



 This command creates the box data type and then uses the type in a class definition:

```
CREATE TYPE box (INTERNALLENGTH = 8,
    INPUT = my_procedure_1, OUTPUT =
    my_procedure_2);
```

CREATE TABLE myboxes (id INT4, description box);

New Type Definition



- In the external language (usually C) functions are written for
- Type input
 - From a text representation to the internal representation
- Type output
 - From the internal representation to a text representation
- Can also define function and operators to manipulate the new type



 A C data structure is defined for the new type:

```
typedef struct Complex {
   double x;
   double y;
} Complex;
```



```
Complex *
   complex in(char *str)
     double x, y;
     Complex *result;
     if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2) {
       elog(WARN, "complex in: error in parsing");
       return NULL;
     result = (Complex *)palloc(sizeof(Complex));
     result->x = x;
     result->y = y;
     return (result);
```



```
char *
   complex out(Complex *complex)
     char *result;
     if (complex == NULL)
       return(NULL);
     result = (char *) palloc(60);
     sprintf(result, "(%g,%g)", complex->x,
             complex->y);
     return(result);
```



Now tell the system about the new type...

```
CREATE FUNCTION complex_in(opaque)
  RETURNS complex
  AS 'PGROOT/tutorial/obj/complex.so'
  LANGUAGE 'c';
CREATE FUNCTION complex_out(opaque)
  RETURNS opaque
  AS 'PGROOT/tutorial/obj/complex.so'
  LANGUAGE 'c';
CREATE TYPE complex (
  internallength = 16,
  input = complex_in,
  output = complex_out);
```

Operator extensions



```
CREATE FUNCTION complex_add(complex,
 complex)
 RETURNS complex
  AS '$PWD/obj/complex.so'
  LANGUAGE 'c';
CREATE OPERATOR + (
  leftarg = complex,
  rightarg = complex,
  procedure = complex add,
  commutator = + );
```

Create tables using the type



 CREATE TABLE test_complex (a complex, b complex);

 INSERT INTO test_complex (a,b) values (....);

Now we can do...



SELECT (a + b) AS c FROM test_complex;

- **+----**+
- |C |
- +----+
- |(5.2,6.05)
- +----+
- |(133.42,144.95)|
- **+----**+

•

Creating new Aggregates



```
CREATE AGGREGATE complex_sum (
  sfunc1 = complex add,
  basetype = complex,
  stype1 = complex,
  initcond1 = '(0,0)');
SELECT complex sum(a) FROM test complex;
    +----+
    complex sum
    |(34,53.9)
    +----+
```

Rules System



CREATE RULE name AS ON event
 TO object [WHERE condition]
 DO [INSTEAD] [action | NOTHING]

 Rules can be triggered by any event (select, insert, update, delete, etc.) as opposed to triggers that can only apply to insert, update, delete and truncate

Triggers in PostgreSQL



- CREATE [CONSTRAINT] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [OR ...]} ON table name [FROM referenced table name] [NOT DEFERRABLE | DEFERRABLE] { INITIALLY IMMEDIATE | INITIALLY DEFERRED [FOR [EACH] { ROW | STATEMENT }] [WHEN (condition) | EXECUTE PROCEDURE function name (arguments)
- where event can be one of: INSERT UPDATE [
 OF column_name [, ...]] DELETE TRUNCATE

Views as Rules



 Views in Postgres are implemented using the rule system. In fact there is absolutely no difference between a

CREATE VIEW myview AS SELECT * FROM mytab;

compared against the two commands

CREATE TABLE myview (same attribute list as for mytab);

CREATE RULE "_RETmyview" AS ON SELECT TO myview DO INSTEAD

SELECT * FROM mytab;

Extensions to Indexing



- Access Method extensions in Postgres
- GiST: A Generalized Search Trees
 - Joe Hellerstein, UC Berkeley

Indexing in OO/OR Systems



- Quick access to user-defined objects
- Support queries natural to the objects
- Two previous approaches
 - Specialized Indices ("ABCDEFG-trees")
 - redundant code: most trees are very similar
 - concurrency control, etc. tricky!
 - Extensible B-trees & R-trees (Postgres/Illustra)
 - B-tree or R-tree lookups only!
 - E.g. 'WHERE movie.video < 'Terminator 2'

GiST Approach



- A generalized search tree. Must be:
- Extensible in terms of queries
- General (B+-tree, R-tree, etc.)
- Easy to extend
- Efficient (match specialized trees)
- Highly concurrent, recoverable, etc.

GiST Applications



- New indexes needed for new apps...
 - find all supersets of S
 - find all molecules that bind to M
 - your favorite query here (multimedia?)
 - Keyword text indexes?
- ...and for new queries over old domains:
 - find all points in region from 12 to 2

find all text elements estimated relevant to a query string