SQL vs PostgreSQL

Checks in PostgreSQL

- Tuple-based checks may only refer to attributes of that relation
- Attribute-based checks may only refer to the name of the attribute
- *No subqueries allowed!*
- Use triggers for more elaborate checks

Assertions in PostgreSQL

- *Assertions are not implemented!*
- Use attribute-based or tuple-based checks where possible
- Use triggers for more elaborate checks

Triggers in PostgreSQL

- PostgreSQL does not allow events for only certain columns
- Rows and tables are called OLD and NEW (no REFERENCING ... AS)
- PostgreSQL only allows to execute a *function* as the action statement

The Trigger – SQL

```
CREATE TRIGGER PriceTrig
  AFTER UPDATE OF price ON Sells
  REFERENCING
    OLD ROW AS ooo
    NEW ROW AS nnn
  FOR EACH ROW
  WHEN (nnn.price > ooo.price + 10)
  INSERT INTO RipoffBars
    VALUES (nnn.bar);
```

The event – only changes to prices

Updates let us talk about old and new tuples

We need to consider each price change

Condition: a raise in price > 10

When the price change is great enough, add the bar to RipoffBars

The Trigger – PostgreSQL

CREATE TRIGGER PriceTrigger

AFTER UPDATE ON Sells

FOR EACH ROW

EXECUTE PROCEDURE checkRipoff();

The event – any changes to Sells

Updates have fixed references OLD and NEW

We need to consider each price change

Conditions moved into function

Always check for a ripoff using a function

The Function – PostgreSQL

```
CREATE FUNCTION CheckRipoff()
  RETURNS TRIGGER AS $$BEGIN
   IF NEW.price > OLD.price+10 THEN
     INSERT INTO RipoffBars
     VALUES (NEW.bar);
   END IF;
   RETURN NEW;
  END$$ LANGUAGE plpgsql;
```

Conditions moved into function

When the price change is great enough, add the bar to RipoffBars

Updates have fixed references OLD and NEW

Functions in PostgreSQL

- CREATE FUNCTION name([arguments]) RETURNS [TRIGGER type] AS $$function definition$$ LANGUAGE lang;
- Example:

```
CREATE FUNCTION add(int,int)
RETURNS int AS $$select $1+$2;$$
LANGUAGE SQL;
```

-
```
CREATE FUNCTION add(i1 int,i2 int)
RETURNS int AS $$BEGIN RETURN
i1 + i2; END;$$ LANGUAGE plpgsql;
```

Example: Attribute-Based Check

```
CREATE TABLE Sells (
  bar    CHAR(20),
  beer   CHAR(20)    CHECK (beer IN
         (SELECT name FROM Beers)),
  price  INT CHECK (price <= 100)
);
```

Example: Attribute-Based Check

```
CREATE TABLE Sells (
  bar    CHAR(20),   beer CHAR(20),
  price  INT CHECK (price <= 100));
CREATE FUNCTION CheckBeerName() RETURNS
  TRIGGER AS $$BEGIN IF NOT NEW.beer IN
  (SELECT name FROM Beers) THEN RAISE
  EXCEPTION 'no such beer in Beers';
  END IF; RETURN NEW; END$$
  LANGUAGE plpgsql;
CREATE TRIGGER BeerName AFTER UPDATE OR
  INSERT ON Sells FOR EACH ROW
  EXECUTE PROCEDURE CheckBeerName();
```

Example: Assertion

- In Drinkers(name, addr, phone) and Bars(name, addr, license), there cannot be more bars than drinkers

```
CREATE ASSERTION LessBars CHECK (
  (SELECT COUNT(*) FROM Bars) <=
  (SELECT COUNT(*) FROM Drinkers)
);
```

Example: Assertion

```
CREATE FUNCTION CheckNumbers()
  RETURNS TRIGGER AS $$BEGIN IF
   (SELECT COUNT(*) FROM Bars) >
   (SELECT COUNT(*) FROM Drinkers)
  THEN RAISE EXCEPTION '2manybars';
  END IF; RETURN NEW; END$$
  LANGUAGE plpgsql;
CREATE TRIGGER NumberBars AFTER
  INSERT ON Bars EXECUTE PROCEDURE
  CheckNumbers();
CREATE TRIGGER NumberDrinkers AFTER
  DELETE ON Drinkers EXECUTE PROCEDURE
  CheckNumbers();
```

Views

Views

- A *view* is a relation defined in terms of stored tables (called *base tables*) and other views
- Two kinds:
  1. *Virtual* = not stored in the database; just a query for constructing the relation
  2. *Materialized* = actually constructed and stored

Declaring Views

- Declare by:

  CREATE [MATERIALIZED] VIEW
      <name> AS <query>;
- Default is virtual
- PostgreSQL has no direct support for materialized views

Materialized Views

- Problem: each time a base table changes, the materialized view may change
  - Cannot afford to recompute the view with each change
- Solution: Periodic reconstruction of the materialized view, which is otherwise "out of date"

Example: A Data Warehouse

- Bilka stores every sale at every store in a database
- Overnight, the sales for the day are used to update a *data warehouse* = materialized views of the sales
- The warehouse is used by analysts to predict trends and move goods to where they are selling best

Virtual Views

- only a query is stored
- no need to change the view when the base table changes
- expensive when accessing the view often

Example: View Definition

- CanDrink(drinker, beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
    SELECT drinker, beer
    FROM Frequents, Sells
    WHERE Frequents.bar = Sells.bar;
```

Example: View Definition

- CanDrink(drinker, beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
    SELECT drinker, beer
    FROM Frequents NATURAL JOIN Sells;
```

Example: View Definition

- CanDrink(drinker, beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE TABLE CanDrink
      (drinker TEXT, beer TEXT);
CREATE RULE "_RETURN" AS ON SELECT
      TO CanDrink DO INSTEAD
      SELECT drinker, beer
      FROM Frequents NATURAL JOIN Sells;
```

Example: Accessing a View

- Query a view as if it were a base table
- Example query:

```
SELECT beer FROM CanDrink
WHERE drinker = 'Peter';
```

- The *rule* "_RETURN" will rewrite this to:

```
SELECT beer FROM (SELECT
drinker, beer FROM Frequents
NATURAL JOIN Sells) AS CanDrink
where drinker = 'Peter';
```

Modifying Virtual Views

- Generally, it is impossible to modify a virtual view, because it does not exist
- But a *rule* lets us interpret view modifications in a way that makes sense
- Example: the view Synergy has (drinker, beer, bar) triples such that the bar serves the beer, the drinker frequents the bar and likes the beer

Example: The View

```
CREATE VIEW Synergy AS
  SELECT Likes.drinker, Likes.beer, Sells.bar
  FROM Likes, Sells, Frequents
  WHERE Likes.drinker = Frequents.drinker
    AND Likes.beer = Sells.beer
    AND Sells.bar = Frequents.bar;
```

Pick one copy of each attribute

Natural join of Likes, Sells, and Frequents

Example: The View

```
CREATE VIEW Synergy AS
  SELECT drinker, beer, bar
  FROM Likes NATURAL JOIN Sells
      NATURAL JOIN Frequents;
```

Interpreting a View Insertion

- We cannot insert into Synergy – it is a virtual view
- But we can use a rule to turn a (drinker, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequents
  - Sells.price will have to be NULL

The Rule

```
CREATE RULE ViewRule AS
  ON INSERT TO Synergy
  DO INSTEAD (
     INSERT INTO Likes VALUES
     (NEW.drinker, NEW.beer);
     INSERT INTO Sells(bar, beer) VALUES
     (NEW.bar, NEW.beer);
     INSERT INTO Frequents VALUES
     (NEW.drinker, NEW.bar);
   );
```

Example: Assertion

```
CREATE FUNCTION CheckNumbers()
  RETURNS TRIGGER AS $$BEGIN IF
   (SELECT COUNT(*) FROM Bars) >
   (SELECT COUNT(*) FROM Drinkers)
  THEN RAISE EXCEPTION '2manybars';
  END IF; RETURN NEW; END$$
  LANGUAGE plpgsql;
CREATE TRIGGER NumberBars AFTER
  INSERT ON Bars EXECUTE PROCEDURE
  CheckNumbers();
CREATE TRIGGER NumberDrinkers AFTER
  DELETE ON Drinkers EXECUTE PROCEDURE
  CheckNumbers();
```

Example: Assertion

```
CREATE FUNCTION CheckNumbers()
  RETURNS TRIGGER AS $$BEGIN IF
   (SELECT COUNT(*) FROM Bars) >
   (SELECT COUNT(*) FROM Drinkers)
  THEN RETURN NULL;
  END IF; RETURN NEW; END$$
  LANGUAGE plpgsql;
CREATE TRIGGER NumberBars AFTER
  INSERT ON Bars EXECUTE PROCEDURE
  CheckNumbers();
CREATE TRIGGER NumberDrinkers AFTER
  DELETE ON Drinkers EXECUTE PROCEDURE
  CheckNumbers();
```

Example: Assertion

```
CREATE RULE CheckBars AS
  ON INSERT TO Bars
  WHEN (SELECT COUNT(*) FROM Bars) >=
  (SELECT COUNT(*) FROM Drinkers)
  DO INSTEAD NOTHING;

CREATE RULE CheckDrinkers AS
  ON DELETE TO Drinkers
  WHEN (SELECT COUNT(*) FROM Bars) >=
  (SELECT COUNT(*) FROM Drinkers)
  DO INSTEAD NOTHING;
```

Transactions

Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time
  - Both queries and modifications
- Unlike operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions

Example: Bad Interaction

- You and your domestic partner each take $100 from different ATM's at about the same time
  - The DBMS better make sure one account deduction does not get lost
- Compare: An OS allows two people to edit a document at the same time; If both write, one's changes get lost

Transactions

- *Transaction* = process involving database queries and/or modification
- Normally with some strong properties regarding concurrency
- Formed in SQL from single statements or explicit programmer control

ACID Transactions

- *ACID transactions* are:
  - *Atomic:* Whole transaction or none is done
  - *Consistent:* Database constraints preserved
  - *Isolated:* It appears to the user as if only one process executes at a time
  - *Durable:* Effects of a process survive a crash
- Optional: weaker forms of transactions are often supported as well

COMMIT

- The SQL statement COMMIT causes a transaction to complete
  - database modifications are now permanent in the database

ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*
  - No effects on the database
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it

Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that C.Ch. sells only Od.Cl. for 20 and Er.We. for 30
- Peter is querying Sells for the highest and lowest price C.Ch. Charges
- C.Ch. decides to stop selling Od.Cl. And Er.We., but to sell only Tuborg at 35

Peter’s Program

- Peter executes the following two SQL statements called (min) and (max) to help us remember what they do

(max) SELECT MAX(price) FROM Sells
WHERE bar = ’C.Ch.’;

(min) SELECT MIN(price) FROM Sells
WHERE bar = ’C.Ch.’;

Cafe Chino’s Program

- At about the same time, C.Ch. executes the following steps: (del) and (ins)

(del) DELETE FROM Sells

WHERE bar = ’C.Ch.’;

(ins) INSERT INTO Sells

VALUES(’C.Ch.’, ’Tuborg’, 35);

Interleaving of Statements

- Although (max) must come before (min), and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Peter’s and/or Cafe Chino’s statements into transactions

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min)

C.Ch. Prices:	{20, 30}	{20,30}		{35}
Statement:	(max)	(del)	(ins)	(min)
Result:	30			35

- Peter sees MAX < MIN!

Fixing the Problem

- If we group Peter's statements (max) (min) into one transaction, then he cannot see this inconsistency
- He sees C.Ch.'s prices at some fixed time
  - Either before or after they changes prices, or in the middle, but the MAX and MIN are computed from the same prices

Another Problem: Rollback

- Suppose C.Ch. executes (del)(ins), not as a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement
- If Peter executes his statements after (ins) but before the rollback, he sees a value, 35, that never existed in the database

Solution

- If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen

Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time
- Only one level (“serializable”) = ACID transactions
- Each DBMS implements transactions in its own way

Choosing the Isolation Level

- Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

Serializable Transactions

- If Peter = (max)(min) and C.Ch. = (del)(ins) are each transactions, and Peter runs with isolation level SERIALIZABLE, then he will see the database either before or after C.Ch. runs, but not in the middle

Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, not how others see it
- Example: If Cafe Chino Runs serializable, but Peter does not, then Peter might see no prices for Cafe Chino
  - i.e., it looks to Peter as if he ran in the middle of Cafe Chino's transaction

Read-Commited Transactions

- If Peter runs with isolation level READ COMMITTED, then he can see only committed data, but not necessarily the same data each time.
- Example: Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Cafe Chino commits
  - Peter sees MAX < MIN

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time
  - But the second and subsequent reads may see *more* tuples as well

Example: Repeatable Read

- Suppose Peter runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min)
  - (max) sees prices 20 and 30
  - (min) can see 35, but must also see 20 and 30, because they were seen on the earlier read by (max)

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never)
- Example: If Peter runs under READ UNCOMMITTED, he could see a price 35 even if Cafe Chino later aborts

Indexes

Indexes

- *Index* = data structure used to speed access to tuples of a relation, given values of one or more attributes
- Could be a hash table, but in a DBMS it is always a balanced search tree with giant nodes (a full disk page) called a *B-tree*

Declaring Indexes

- No standard!
- Typical syntax (also PostgreSQL):

```
CREATE INDEX BeerInd ON
  Beers(manf);
CREATE INDEX SellInd ON
  Sells(bar, beer);
```

Using Indexes

- Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index
- Example: use BeerInd and SellInd to find the prices of beers manufactured by Albani and sold by Cafe Chino (next slide)

Using Indexes

```
SELECT price FROM Beers, Sells
WHERE manf = 'Albani' AND
   Beers.name = Sells.beer AND
   bar = 'C.Ch.';
```

1. Use BeerInd to get all the beers made by Albani
2. Then use SellInd to get prices of those beers, with bar = 'C.Ch.'

Database Tuning

- A major problem in making a database run fast is deciding which indexes to create
- Pro: An index speeds up queries that can use it
- Con: An index slows down all modifications on its relation because the index must be modified too

Example: Tuning

- Suppose the only things we did with our beers database was:
  1. Insert new facts into a relation (10%)
  2. Find the price of a given beer at a given bar (90%)
- Then SellInd on Sells(bar, beer) would be wonderful, but BeerInd on Beers(manf) would be harmful

Tuning Advisors

- A major research area
  - Because hand tuning is so hard
- An advisor gets a *query load*, e.g.:

1. Choose random queries from the history of queries run on the database, or
2. Designer provides a sample workload

Tuning Advisors

- The advisor generates candidate indexes and evaluates each on the workload
  - Feed each sample query to the query optimizer, which assumes only this one index is available
  - Measure the improvement/degradation in the average running time of the queries

Summary 7

More things you should know:

- Constraints, Cascading, Assertions
- Triggers, Event-Condition-Action
- Triggers in PostgreSQL, Functions
- Views, Rules
- Transactions

Real SQL Programming

SQL in Real Programs

- We have seen only how SQL is used at the generic query interface – an environment where we sit at a terminal and ask queries of a database
- Reality is almost always different: conventional programs interacting with SQL

Options

1. Code in a specialized language is stored in the database itself (e.g., PSM, PL/SQL)
2. SQL statements are embedded in a *host language* (e.g., C)
3. Connection tools are used to allow a conventional language to access a database (e.g., CLI, JDBC, PHP/DB)

Stored Procedures

- PSM, or "*persistent stored modules*," allows us to store procedures as database schema elements
- PSM = a mixture of conventional statements (if, while, etc.) and SQL
- Lets us do things we cannot do in SQL alone

Basic PSM Form

```
CREATE PROCEDURE <name> (
    <parameter list> )
  <optional local declarations>
  <body>;
```

- Function alternative:

```
CREATE FUNCTION <name> (
    <parameter list> ) RETURNS <type>
```

Parameters in PSM

- Unlike the usual name-type pairs in languages like C, PSM uses mode-name-type triples, where the *mode* can be:
  - IN = procedure uses value, does not change value
  - OUT = procedure changes, does not use
  - INOUT = both

Example: Stored Procedure

- Let's write a procedure that takes two arguments *b* and *p*, and adds a tuple to Sells(bar, beer, price) that has bar = 'Cafe Chino', beer = *b*, and price = *p*
  - Used by Cafe Chino to add to their menu more easily

The Procedure

```
CREATE PROCEDURE ChinoMenu (
    IN   b     CHAR(20),
    IN p       REAL
)
INSERT INTO Sells
VALUES('C.Ch.', b, p);
```

Parameters are both read-only, not changed

The body --- a single insertion

Invoking Procedures

- Use SQL/PSM statement CALL, with the name of the desired procedure and arguments
- Example:

```
CALL ChinoMenu('Eventyr', 50);
```

- Functions used in SQL expressions wherever a value of their return type is appropriate