CSE 530A

Inheritance and Partitioning

Washington University
Fall 2013
Inheritance

• PostgreSQL provides *table inheritance*
  – SQL defines type inheritance, PostgreSQL's table inheritance is different

• A table can inherit its definition from one or more table
  – Inherits its column definitions from its parent tables
  – Can optionally add more columns
Inheritance

• Example

CREATE TABLE cities (  
    name text,  
    population float,  
    altitude int -- in feet  
);  

CREATE TABLE capitals (  
    state char(2)  
) INHERITS (cities);  

– The capitals table has all of the same columns as the cities table, plus the extra state column
Inheritance

• Inserts should be done to whichever table is most appropriate
• Queries (and updates and deletes) can be applied to either a single table or a table and all of its descendants
Inheritance

- By default, queries search a table and all of its descendants
  - Including the ONLY keyword after FROM causes it to *not* include descendant tables

```
SELECT name, altitude
FROM cities
WHERE altitude > 500;
```

<table>
<thead>
<tr>
<th>name</th>
<th>altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Las Vegas</td>
<td>2174</td>
</tr>
<tr>
<td>Mariposa</td>
<td>1953</td>
</tr>
<tr>
<td>Madison</td>
<td>845</td>
</tr>
</tbody>
</table>

```
SELECT name, altitude
FROM ONLY cities
WHERE altitude > 500;
```

<table>
<thead>
<tr>
<th>name</th>
<th>altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Las Vegas</td>
<td>2174</td>
</tr>
<tr>
<td>Mariposa</td>
<td>1953</td>
</tr>
</tbody>
</table>
Inheritance

• Can also put a * after a table name to explicitly include descendants
  – Not necessary as this is the default behavior

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

<table>
<thead>
<tr>
<th>name</th>
<th>altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Las Vegas</td>
<td>2174</td>
</tr>
<tr>
<td>Mariposa</td>
<td>1953</td>
</tr>
<tr>
<td>Madison</td>
<td>845</td>
</tr>
</tbody>
</table>
Inheritance

• Inserts will not be routed to child tables
  – The following INSERT will fail because the cities tables not have a state column

  INSERT INTO cities (name, population, altitude, state)
  VALUES ('New York', NULL, NULL, 'NY');
Inheritance

- Indexes (including uniqueness constraints) and foreign key constraints only apply to single tables
  - They do not apply across parents and children

- For example, declaring `cities.name` as UNIQUE or PRIMARY KEY would not keep duplicates from being inserted into `capitals`
  - Even if `capitals.name` is also declared UNIQUE the same value could be inserted into both `cities` and `capitals`
Inheritance

• If we declared a foreign key as `REFERENCES cities(name)` in another table then that column could not also reference the `capitals table`
  – Currently no solution for this other than to not declare the foreign key constraint
Partitioning

- PostgreSQL provides the ability to split one large table into smaller pieces
  - Other DBMSs have similar capabilities, but the process and syntax varies considerably
- Partitioning splits a table into smaller tables based on some column or columns
  - But still provides a logical view of the entire table
  - Uses table inheritance to do this
Partitioning

• Why partition?
  – Performance
    • If most queries only access a single partition then scanning the entire larger table can be avoided
      – Also avoids the overhead of leading all indexes with the partition key
    • Bulk loads and deletes can be done by adding and removing partitions
      – Can avoid locking or the cleanup generated by large deletes
Partitioning

• Implementing partitioning
  1. Create the master table
     • This table will not actually contain any data
       – No point in creating indexes or uniqueness constraints
  2. Create the child (partition) tables
     • The child tables inherit their definition from the master table
     • The child tables should not have any extra columns
  3. Add constraints to the partition tables to limit the partition key columns to the allowed values for each partition
     • Ensure there is no overlap, so that any row can be added to only one table
Partitioning

• Implementing partitioning
  4. For each partition table, create any desired indexes and constraints
     • This includes the primary key constraints
  5. Define a trigger or rule to redirect inserts into the master table to the appropriate partition
Partitioning

• Example: Suppose we are collecting sales information. We might partition in data by month.
  – Conceptually, we might want a table like:

```sql
CREATE TABLE sales (  
    order_nbr BIGINT NOT NULL,
    ...
    sale_date TIMESTAMP WITHOUT TIME ZONE  
;)
```
Partitioning

- After creating the master table, we then create a partition for each active month
  - Include constraint to enforce the partitioning

```
CREATE TABLE sales_y2013m01 (  
    CHECK ( sale_date >= DATE '2013-01-01'  
            AND sale_date < DATE '2013-02-01' )  
) INHERITS (sales);

CREATE TABLE sales_y2013m02 (  
    CHECK ( sale_date >= DATE '2013-02-01'  
            AND sale_date < DATE '2013-03-01' )  
) INHERITS (sales);

...
Partitioning

• We probably want indexes on the partition key

CREATE INDEX sales_y2013m01_date ON sales_y2013m01 (sale_date);
CREATE INDEX sales_y2013m02_date ON sales_y2013m02 (sale_date);
...

Partitioning

- Set up a trigger so that `INSERT INTO sales ...` redirects the data into the appropriate child table
  - If data is always going to be inserted into the most recent table then we can use a simple trigger function

```sql
CREATE OR REPLACE FUNCTION sales_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO sales_y2013m11 VALUES (NEW.*);
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER insert_sales_trigger
BEFORE INSERT ON sales
FOR EACH ROW EXECUTE PROCEDURE sales_insert_trigger();
```
Partitioning

• We can try to make the function smarter

```
CREATE OR REPLACE FUNCTION sales_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
  IF ( NEW.sale_date >= DATE '2013-01-01' AND
      NEW.sale_date < DATE '2013-02-01' ) THEN
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
  ELSIF ( NEW.sale_date >= DATE '2013-02-01' AND
          NEW.sale_date < DATE '2013-03-01' ) THEN
    INSERT INTO measurement_y2006m03 VALUES (NEW.*);
  ...  
  ELSIF ( NEW.sale_date >= DATE '2013-11-01' AND
          NEW.sale_date < DATE '2013-12-01' ) THEN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
  ELSE
    RAISE EXCEPTION 'Date out of range. Fix the trigger function!';
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```
Partitioning

- Note that each `IF` case must match the table constraints exactly
- Will still need to update the trigger function periodically
  - Over time, new partitions may be added and old partitions removed
    - Can use DROP TABLE to instantly drop a child table (after trigger function has been changed)
    - Could also just alter the table to no longer inherit from the master table without dropping to keep the data around
Constraint Exclusion

• Constraint exclusion is a query optimization that improves performance for partitioned tables

SELECT count(*) FROM sales WHERE sale_date >= DATE '2013-11-01';

– Without constraint exclusion, the query would cause all child tables of the sales table to be searched
– With constraint exclusion, the query planner examines the constraints on the child tables to determine which need to be searched
Constraint Exclusion

- Limitations
  - Constraint exclusion only works when the WHERE clause contains constants
    - Comparison against things like `now()` or `CURRENT_TIMESTAMP` cannot be optimized
      - The planning can't know which partitions will match when the query is actually run