DM505 Database Design and Programming
DM576 Database Systems

Panagiotis Tampakis

ptampakis@imada.sdu.dk

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
- Virtual View:
  - Advantage → it will always return the latest data to you
  - Disadvantage → performance

Materialized Views

- Problem: each time a base table changes, the materialized view may change
  - Cannot afford to recompute the view with each change
  - Solution: Save the results of the underlying query in a table
  - Advantage → Performance
  - Disadvantage → the returned data is only as up to date as the last time the materialized view has been refreshed.

Materialized Views

- Reconstruction of the materialized view, which is otherwise “out of date”
  - manual
  - on a set schedule
  - based on the database detecting a change in data from one of the underlying tables
  - Postgres supports **manual** and **trigger-based** reconstruction.

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;
```

Modifying Virtual Views

- Generally, it is impossible to modify a virtual view, because it does not exist
- SQL provides some rules for when modifications to a view axe permitted.
- Roughly, they permit modifications on views that are defined by selecting (using SELECT, not SELECT DISTINCT) some attributes from one relation R

Modifying Virtual Views

- Rules:
  - The WHERE clause must not involve R in a subquery.
  - The FROM clause can only consist of one occurrence of R and no other relation.
  - The list in the SELECT clause must include enough attributes that for every tuple inserted into the view.
    - For example, it is not permitted to project out an attribute that is declared NOT NULL and has no default.

Modifying Virtual Views

- Alternatively, we can use an *instead of trigger* 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

Pick one copy of each attribute

```
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;
```

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 an *instead of trigger* 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

Instead of Triggers

```
CREATE TRIGGER ViewRule
INSTEAD OF INSERT ON Synergy
REFERENCING NEW ROW AS NEW
FOR EACH ROW
BEGIN
     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);
END;
```

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 is executed, or nothing
  - *Consistent:* Brings the Database from one consistent state to another by preserving constraints
  - *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

BEGIN TRANSACTION

- The SQL statement BEGIN TRANSACTION marks the beginning of a transaction
  - all following statements are part of the transaction
  - until either a COMMIT or a ROLLBACK statements occurs

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) should 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 change 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'd see a value (35) that never existed in the database

Solution

- If Cafe Chino 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

- Runing 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 may look 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

- Just 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

Overview of Isolation Levels

Isolation Level	Dirty Reads	Nonrepeat-able Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

Multi-Version Concurrency Control (MVCC)

- Several real systems do not implement locks
- MVCC
  - is a non-locking concurrency control method to provide concurrent access to the database.

Multi-Version Concurrency Control (MVCC)

- Keeps multiple copies of each data item.
  - each user sees a *snapshot* of the database at a particular instant in time.
    - In case of writes it doesn't overwrite the original data item with new data,
    - instead creates a newer version of the data item. Thus there are multiple versions stored
  - Any changes made by a writer will not be seen by other users of the database until the changes have been completed

Multi-Version Concurrency Control (MVCC)

- MVCC uses
  - timestamps (TS),
  - and incrementing transaction IDs, to achieve transactional consistency
- Example

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

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 often 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 uses 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 11

More things you should know:

- Views
- Transactions
- Begin, Commit, Rollback
- Indexes