Constraints, Views & Indexes

Introduction to databases
CSCC43 Winter 2012
Ryan Johnson

Thanks to Manos Papagelis, John Mylopoulos, Arnold Rosenbloom
and Renee Miller for material in these slides

INTEGRITY CONSTRAINTS

Running Example

Gamers(name, address, age)
Platforms(name, manf, price)
Games(name, genre, studio)
plays(gamer, game, rating)
owns(gamer, platform)
available(game, platform, price, releasedate)

Underline = key (tuples cannot have the same value in all key attributes)
– Excellent example of a constraint

Kinds of Constraints

• Keys
• Foreign-key or referential-integrity constraints
  – Inter-relation constraints
• Value-based constraints
  – Constrain values of a particular attribute
• Tuple-based constraints
  – Relationship among components
• Assertions
Declaring Keys

• An attribute or list of attributes may be declared **PRIMARY KEY** or **UNIQUE**
  - Identify a set of attributes that uniquely identify each tuple
  - Place **PRIMARY KEY** or **UNIQUE** after the type in the declaration of the attribute.

  ```
  CREATE TABLE Games (
      name CHAR(20) PRIMARY KEY,
      studio CHAR(20),
      genre CHAR(20)
  );
  ```

**PRIMARY KEY vs. UNIQUE**

• There can be only one **PRIMARY KEY** for a relation, but several **UNIQUE** attributes
• No attribute of a **PRIMARY KEY** can ever be NULL in any tuple. But attributes declared **UNIQUE** may have NULL’s, and there may be several tuples with NULL.

Declaring multi-attribute keys

• A key declaration can appear as element in the list of elements of a CREATE TABLE statement
• This form is essential if the key consists of more than one attribute

  ```
  CREATE TABLE Available ( 
      game CHAR(20),
      platform VARCHAR(20),
      price REAL,
      PRIMARY KEY (game, platform));
  ```

  Game and platform together are the key for Available

Foreign keys

• Values appearing in attributes of one relation must appear together with certain attributes of another relation

  **Example:**
  We might expect that a value in Plays.gamer also appears as value in **Gamer.name**

  ```
  Gamer(name, address, age)
  Plays(gamer, game, rating)
  ```
Expressing foreign keys

- Use keyword `REFERENCES`, either:
  - After an attribute (for one-attribute keys)
    
    ```sql
    REFERENCES <relation> (<attribute>)
    ```
  - As an element of the schema:
    
    ```sql
    FOREIGN KEY (<list-of-attributes>)
    REFERENCES <relation> (<list-of-attributes>)
    ```

- Referenced attributes must be declared `PRIMARY KEY` or `UNIQUE`

Example: as schema element

```sql
CREATE TABLE Games (  
  name CHAR(20) PRIMARY KEY,
  studio CHAR(20),
  genre CHAR(20)
);
CREATE TABLE Available (  
  game CHAR(20) REFERENCES Games(name),
  platform VARCHAR(20),
  price REAL
);
```

Example: with attribute

```sql
CREATE TABLE Games (  
  name CHAR(20) PRIMARY KEY,
  studio CHAR(20),
  genre CHAR(20)
);
CREATE TABLE Available (  
  game CHAR(20) REFERENCES Games(name),
  platform VARCHAR(20),
  price REAL
);
```

Enforcing foreign key constraints

- If there is a foreign-key constraint from relation `R` to relation `S`, two violations are possible:
  - Insert/update to `R` introduces values not found in `S`
  - Deletion/update to `S` causes some tuples of `R` to “dangle”

- Example: suppose `R = Available`, `S = Games`
  - An insert/update to `Available` cannot introduce a nonexistent game
  - A deletion/update to `Games` cannot simply remove a game named in some tuples of `Available`
  - Several ways to handle the latter (next slide)
Policies for protecting foreign keys

- **DEFAULT**: Reject the modification
  - Deleted game in Games: reject if used in Available tuples
  - Updated game in Games: reject if used in Available tuples
- **CASCADE**: Make the same changes in Available
  - Deleted game in Games: delete Available tuples that use it
  - Updated game in Games: update Available tuples that use it
- **SET NULL**: Change the game to NULL
  - Deleted game in Games: set NULL values in Available tuples
  - Updated game in Games: set NULL values in Available tuples

Example

- Delete the Pac-man tuple from Games
  - CASCADE: delete all tuples from Available that have game='Pac-man'
  - SET NULL: update tuples from Available with game='Pac-man' to have game=NULL
- Update the ‘Pac-man’ tuple to ‘Pac man’
  - CASCADE: change all Available tuples having game='Pac-man' to game='Pac man'
  - SET NULL: update tuples from Available with game='Pac-man' to have game=NULL

Choosing a policy

- When we declare a foreign key, we may choose policies **SET NULL** or **CASCADE** independently for deletions and updates
- Follow the foreign-key declaration by:
  
  ON [UPDATE, DELETE][SET NULL, CASCADE]

- Two such clauses may be used, otherwise, the default (reject) is used.

Example: setting foreign key policies

```sql
CREATE TABLE Available (  
  platform CHAR(20),  
  game CHAR(20),  
  price REAL,  
  FOREIGN KEY(game)  
  REFERENCES Games(name)  
  ON DELETE SET NULL  
  ON UPDATE CASCADE
);
```
Attribute-based checks

- Constraints on the value of a particular attribute
- Add **CHECK**(<condition>) to the declaration for the attribute
- Condition is any scalar boolean expression
  - arithmetic/logical expressions
  - queries returning one tuple having one attribute
- The condition may use the name of the attribute, but *any other relation or attribute name must be in a subquery*

Timing of checks

- Attribute-based checks are performed only when a value for that attribute is inserted or updated
- Example:
  - **CHECK** (price <= 5.00)
    - Checks every new price and rejects the modification (for that tuple) if the price is more than $5
  - **CHECK** (game IN(SELECT name FROM Games))
    - Not checked if a game is later deleted from Games (unlike foreign-keys)

Example: attribute-based check

```sql
CREATE TABLE Available (  
  platform CHAR(20),
  game CHAR(20)  
  CHECK (game IN(SELECT name FROM Games)),
  price REAL  
  CHECK (price <= 5.00 )  
);```

Tuple-based checks

- **CHECK** (<condition>) may be added as a relation-schema element
  - The condition may refer to any attribute of the relation, but other attributes or relations require a subquery
  - Checked on insert or update only

Example: Only PS3 can sell game for more than $100:

```sql
CREATE TABLE Available (  
  platform CHAR(20),
  game CHAR(20),  
  price REAL,  
  CHECK (platform = 'PS3' OR price <= 100.00)  
);```
**Assertions**

- Permit the definition of constraints over whole tables, rather than individual tuples
  - useful to express generic inter-relational constraints
  - An assertion associates a name to a check clause.
- Syntax:
  ```sql
  CREATE ASSERTION AssertName CHECK (Condition)
  ```

**Example: assertion**

- "Every platform supports at least one game":
- Syntax:
  ```sql
  CREATE ASSERTION MinGamesPerPlatform
  CHECK (NOT EXISTS (SELECT name FROM Platform
  LEFT JOIN Available ON name=platform
  GROUP BY name HAVING count(game) = 0))
  ```
- Caveat: assertions are expensive to enforce
  - Especially if specified with expensive queries
  - Many systems do not support them at all

**Enforcement policies**

- Integrity constraints (checks, assertions) may be checked immediately when a change takes place to a relation, or at the end of a transaction
  - The first case may result in a partial rollback
  - The latter triggers a (full) rollback.
- This topic is discussed in more detail in **CSCD43**
Views

• A **view** is a relation defined in terms of stored tables (called **base tables**) and other views.

• Two kinds:
  – *Virtual* = not stored in the database; just a query for constructing the relation
    ```sql
    CREATE VIEW <name> AS <query>;
    ```
  – *Materialized* = actually constructed and stored
    ```sql
    CREATE MATERIALIZED VIEW <name> AS <query>;
    ```

Example: View Definition

**CanPlay**(gamer, game) is a view “containing” the gamer-game pairs such that the gamer owns at least one platform that supports the game:

```sql
CREATE VIEW CanPlay AS
    SELECT gamer, game
    FROM Owns, Available
    WHERE Owns.platform = Available.platform;
```

Example: Accessing a View

Query a view as if it were a base table:

```sql
SELECT game
FROM CanPlay
WHERE gamer = 'Sally';
```

Notes on Views

• **Data independence** (hide schema from apps)
  – DB team splits CustomerInfo into Customer and Address
  – Add a view so web apps don’t break

• **Data hiding** (access data on need-to-know basis)
  – Doctor outsources patient billing to third party
  – View only allows access to billing-related patient info

• **Code reuse**
  – Similar subquery appears multiple times in a query
  – View shortens code, improves readability, reduces bugs, ...
  – Bonus: query optimizer often does a better job!
Example: Views and Queries

Employee(RegNo, FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the department with highest salary expenditures" (without using a view):
SELECT Dept
FROM Employee
GROUP BY Dept
HAVING sum(Salary) >= ALL 
   SELECT sum(Salary) FROM Employee GROUP BY Dept)

NOTE: Not all DBMS support the ALL keyword

Example: Views and Queries (cont.)

"Find the department with highest salary expenditures" (using a view):
CREATE VIEW SalBudget (Dept, SalTotal) AS
SELECT Dept, sum(Salary)
FROM Employee
GROUP BY Dept
SELECT Dept
FROM SalBudget
WHERE SalTotal = (SELECT max(SalTotal) FROM SalBudget)

Updates on Views

- Generally, it is impossible to modify a virtual view because it doesn’t exist
- Can’t we “translate” updates on views into “equivalent” updates on base tables?
  - Not always (in fact, not often)
  - Most systems prohibit most view updates

Example: The View

CREATE VIEW Synergy AS
SELECT Plays.gamer, Plays.game, Available
FROM Plays, Available, Owns
WHERE Plays.gamer = Owns.gamer
   AND Plays.game = Available.game
   AND Available.platform = Owns.platform;

Natural join of Plays, Available, and Owns
Pick one copy of each attribute
Interpreting a View Insertion

- We cannot insert into Synergy - it is a virtual view
- Idea: Try to translate a \(\text{gamer}, \text{game}, \text{platform}\) triple into three insertions of projected pairs, one for each of Plays, Available, and Owns.
  - Available.price will have to be NULL.
  - There isn’t always a unique translation

Need for SQL Triggers - Not discussed

Materialized Views

- **Problem**: each time a base table changes, materialized views become stale
  - Unattractive to regenerate from scratch with each change
- **Solution**: Periodic reconstruction of the materialized view, accept “out of date” results

Hot research area: *incremental* view maintenance

Example: A Data Warehouse

- Wal-Mart 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
Index

- **Problem**: needle in haystack
  - Find all phone numbers with first name ‘Mary’
  - Find all phone numbers with last name ‘Li’
- **Index**: auxiliary database structure which provides random access to data
  - Index a set of attributes. No standard syntax! Typical is:
    ```sql
    CREATE INDEX indexName ON TableName(AttributeList);
    ```
  - Random access to any indexed attribute
    (e.g., retrieve a single tuple out of billions in <5 disk accesses)
  - Similar to a hash table, but in a DBMS it is a balanced search tree with giant nodes (a full disk page) called a **B-tree**

Another Example: Using Index

```sql
CREATE INDEX GamesInd ON Games(studio);
CREATE INDEX AvailableInd ON Available(platform, game);

Query: Find the prices of games manufactured by Valve and available on XBox

SELECT price FROM Games, Available
WHERE studio = 'Valve' AND Games.name = Available.game
  AND platform = 'XBox';
```

DBMS uses:
- **GamesInd** to quickly find games made by Valve
- **SellInd** to quickly find prices for games when platform = ‘XBox’

Example: Using Index

```sql
SELECT fname
FROM people
WHERE lname = 'Papagelis'
```

- **Without an index**:
The DBMS must look at the `lname` column on every row in the table (this is known as a **full table scan**)
- **With an index** (defined on attribute `lname`):
The DBMS simply follows the **B-tree** data structure until the ‘Papagelis’ entry has been found

  This is much less computationally expensive than a full table scan

Database tuning

- **How to make a database run fast?**
  - Decide which indexes to create
  - Decide which views to materialize
- **Pro**: Speeds up queries that can use them
- **Con**: Slows down all modifications to affected relations (have to keep index/view in sync)
- **Also**:
  - Index-only access (all needed columns available in index)
  - Clustered vs unclustered indices (D43 topic)
Example: Database Tuning

• Suppose the only things we did with our games database was:
  – Insert new games into the database (10%).
  – Find the price of a given game on a given platform (90%).
• Then
  – SellInd on Available(platform, game) would be wonderful
  – GamesInd on Games(studio) would be harmful

Make common case fast

Tuning is HARD

• Automatic tuning advisors: major research area
• Input: DB schema and a query load, e.g.:
  – Choose random queries from the history of queries run, or
  – Designer provides a sample workload
• Advisor searches space of candidate indexes
  – Predict space, performance impact wrt workload
  – Even larger search space than for query optimization!
• Output: optimal set of indices for workload