Chapter 4: Semantic Data Control

- View management
- Security control
- Integrity control

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of the last year course.

Semantic Data Control

- Semantic data control typically includes view management, security control, and semantic integrity control.
- Informally, these functions must ensure that **authorized** users perform **correct** operations on the database, contributing to the maintenance of database **integrity**.
- In RDBMS semantic data control can be achieved in a uniform way
  - views, security constraints, and semantic integrity constraints can be defined as rules that the system automatically enforces

View Management

- Views enable full logical data independence
- Views are virtual relations that are defined as the result of a query on base relations
- Views are typically not materialized
  - Can be considered a dynamic window that reflects all relevant updates to the database
- Views are very useful for ensuring data security in a simple way
  - By selecting a subset of the database, views **hide** some data
  - Users cannot see the hidden data

View Management in Centralized Databases

- A view is a relation that is derived from a base relation via a query.
- It can involve selection, projection, aggregate functions, etc.
- **Example:** The view of system analysts derived from relation EMP

```
CREATE VIEW SYSAN(ENO,ENAME) AS
SELECT ENO,ENAME
FROM EMP
WHERE TITLE="Syst. Anal."
```

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

SYSAN

ENO	ENAME
E2	M.Smith
E5	B.Casey
E8	J.Jones

View Management in Centralized Databases ...

- Queries expressed on views are translated into queries expressed on base relations
- **Example**: “Find the names of all the system analysts with their project number and responsibility?”
  - Involves the view SYSAN and the relation ASG(ENO,PNO,RESP,DUR)

```
SELECT ENAME, PNO, RESP
FROM SYSAN, ASG
WHERE SYSN.ENO = ASG.ENO
```

is translated into

```
SELECT ENAME,PNO,RESP
FROM EMP, ASG
WHERE EMP.ENO = ASG.ENO
AND TITLE = "Syst. Anal."
```

ENAME	PNO	RESP
M.Smith	P1	Analyst
M.Smith	P2	Analyst
B.Casey	P3	Manager
J.Jones	P4	Manager

- Automatic query modification is required, i.e., ANDing query qualification with view qualification

View Management in Centralized Databases . . .

- All views can be queried as base relations, but not all view can be updated as such
  - Updates through views can be handled automatically only if they can be propagated correctly to the base relations
  - We classify views as updatable or not-updatable

- **Updatable view:** The updates to the view *can* be propagated to the base relations without ambiguity.

```
CREATE VIEW SYSAN(ENO,ENAME) AS
SELECT ENO,ENAME
FROM EMP
WHERE TITLE="Syst. Anal."
```

  - e.g, insertion of tuple (201,Smith) can be mapped into the insertion of a new employee (201, Smith, “Syst. Anal.”)
  - If attributes other than TITLE were hidden by the view, they would be assigned the value *null*

View Management in Centralized Databases . . .

- **Non-updatable view:** The updates to the view *cannot* be propagated to the base relations without ambiguity.

```
CREATE VIEW EG(ENAME,RESP) AS
SELECT ENAME,RESP
FROM EMP, ASG
WHERE EMP.ENO=ASG.ENO
```

  - e.g, deletion of (Smith, ”Syst. Anal.”) is ambiguous, i.e., since deletion of “Smith” in EMP and deletion of “Syst. Anal.” in ASG are both meaningful, but the system cannot decide.

- Current systems are very restrictive about supportin gupdates through views
  - Views can be updated only if they are derived from a single relation by selection and projection
  - However, it is theoretically possible to automatically support updates of a larger class of views, e.g., joins

View Management in Distributed Databases

- Definition of views in DDBMS is similar as in centralized DBMS
  - However, a view in a DDBMS may be derived from fragmented relations stored at different sites
- Views are conceptually the same as the base relations, therefore we store them in the (possibly) distributed directory/catalogue
  - Thus, views might be centralized at one site, partially replicated, fully replicated
  - Queries on views are translated into queries on base relations, yielding distributed queries due to possible fragmentation of data
- Views derived from distributed relations may be costly to evaluate
  - Optimizations are important, e.g., snapshots
  - A snapshot is a static view
    - does not reflect the updates to the base relations
    - managed as temporary relations: the only access path is sequential scan
    - typically used when selectivity is small (no indices can be used efficiently)
    - is subject to periodic recalculation

Data Security

- **Data security** protects data against unauthorized acces and has two aspects:
  - Data protection
  - Authorization control

Data Protection

- **Data protection** prevents unauthorized users from understanding the physical content of data.
- Well established standards exist
  - Data encryption standard
  - Public-key encryption schemes

Authorization Control

- **Authorization control** must guarantee that only authorized users perform operations they are allowed to perform on the database.
- Three actors are involved in authorization
  - **users**, who trigger the execution of application programms
  - **operations**, which are embedded in applications programs
  - **database objects**, on which the operations are performed
- Authorization control can be viewed as a triple *(user, operation type, object)* which specifies that the user has the right to perform an operation of operation type on an object.
- Authentication of (groups of) users is typically done by username and password
- Authorization control in (D)DBMS is more complicated as in operating systems
  - In a file system: data objects are files
  - In a DBMS: Data objects are views, (fragments of) relations, tuples, attributes

Authorization Control . . .

- Grand and revoke statements are used to authorize triplets (user, operation, data object)
  - **GRANT** `<operations>` **ON** `<object>` **TO** `<users>`
  - **REVOKE** `<operations>` **ON** `<object>` **TO** `<users>`
- Typically, the creator of objects gets all permissions
  - Might even have the permission to GRANT permissions
  - This requires a recursive revoke process
- Privileges are stored in the directory/catalogue, conceptually as a matrix

	EMP	ENAME	ASG
Casey	UPDATE	UPDATE	UPDATE
Jones	SELECT	SELECT	SELECT WHERE RESP $\neq$ “Manager”
Smith	NONE	SELECT	NONE

- Different materializations of the matrix are possible (by row, by columns, by element), allowing for different optimizations
  - e.g., by row makes the enforcement of authorization efficient, since all rights of a user are in a single tuple

Distributed Authorization Control

- Additional problems of **authorization control in a distributed environment** stem from the fact that objects and subjects are distributed:
  - remote user authentication
  - managmenet of distributed authorization rules
  - handling of views and of user groups
- **Remote user authentication** is necessary since any site of a DDBMS may accept programs initiated and authorized at remote sites
- Two solutions are possible:
  - (username, password) is replicated at all sites and are communicated between the sites, whenever the relations at remote sites are accessed
    - beneficial if the users move from a site to a site
  - All sites of the DDBMS identify and authenticate themselves similarly as users do
    - intersite communication is protected by the use of the site password;
    - (username, password) is authorized by application at the start of the session;
    - no remote user authentication is required for accessing remote relations once the start site has been authenticated
    - beneficial if users are static

Semantic Integrity Constraints

- A database is said to be **consistent** if it satisfies a set of constraints, called **semantic integrity constraints**
- Maintain a database consistent by enforcing a set of constraints is a difficult problem
- Semantic integrity control evolved from procedural methods (in which the controls were embedded in application programs) to declarative methods
  - avoid data dependency problem, code redundancy, and poor performance of the procedural methods
- Two main types of constraints can be distinguished:
  - **Structural constraints**: basic semantic properties inherent to a data model e.g., unique key constraint in relational model
  - **Behavioral constraints**: regulate application behavior e.g., dependencies (functional, inclusion) in the relational model
- A semantic integrity control system has 2 components:
  - Integrity constraint specification
  - Integrity constraint enforcement

Semantic Integrity Constraint Specification

- **Integrity constraints specification**
  - In RDBMS, integrity constraints are defined as **assertions**, i.e., expression in tuple relational calculus
  - Variables are either universally ($\forall$) or existentially ($\exists$) quantified
  - Declarative method
  - Easy to define constraints
  - Can be seen as a query qualification which is either true or false
  - Definition of database consistency clear
  - 3 types of integrity constraints/assertions are distinguished:
    * predefined
    * precompiled
    * general constraints

- In the following examples we use the following relations:
  EMP(ENO, ENAME, TITLE)
  PROJ(PNO, PNAME, BUDGET)
  ASG(ENO, PNO, RESP, DUR)

Semantic Integrity Constraint Specification ...

- **Predefined constraints** are based on simple keywords and specify the more common contraints of the relational model
- Not-null attribute:
  - e.g., Employee number in EMP cannot be null
    ```
    ENO NOT NULL IN EMP
    ```
- Unique key:
  - e.g., the pair (ENO,PNO) is the unique key in ASG
    ```
    (ENO, PNO) UNIQUE IN ASG
    ```
- Foreign key:
  - e.g., PNO in ASG is a foreign key matching the primary key PNO in PROJ
    ```
    PNO IN ASG REFERENCES PNO IN PROJ
    ```
- Functional dependency:
  - e.g., employee number functionally determines the employee name
    ```
    ENO IN EMP DETERMINES ENAME
    ```

Semantic Integrity Constraint Specification ...

- **Precompiled constraints** express preconditions that must be satisfied by all tuples in a relation for a given update type
- General form:
  CHECK ON `<relation>` [**WHEN** `<update type>`] `<qualification>`
- Domain constraint, e.g., constrain the budget:
  CHECK ON `PROJ(BUDGET>500000` **AND** `BUDGET≤1000000)`
- Domain constraint on deletion, e.g., only tuples with budget 0 can be deleted:
  CHECK ON `PROJ` **WHEN DELETE** `(BUDGET = 0)`
- Transition constraint, e.g., a budget can only increase:
  CHECK ON `PROJ (NEW.BUDGET > OLD.BUDGET` **AND**
  `NEW.PNO = OLD.PNO)`
  - OLD and NEW are implicitly defined variables to identify the tuples that are subject to update

Semantic Integrity Constraint Specification ...

- **General constraints** may involve more than one relation
- General form:
  CHECK ON <variable>:<relation> (<qualification>)
- Functional dependency:
  CHECK ON e1:EMP, e2:EMP
  (e1.ENAME = e2.ENAME **IF** e1.ENO = e2.ENO)
- Constraint with aggregate function:
  e.g., The total duration for all employees in the CAD project is less than 100
  CHECK ON g:ASG, j:PROJ
  (**SUM**(g.DUR **WHERE** g.PNO=j.PNO) < 100
  IF j.PNAME="CAD/CAM")

Semantic Integrity Constraints Enforcement

- **Enforcing semantic integrity constraints** consists of rejecting update programs that violate some integrity constraints
- Thereby, the major problem is to find **efficient algorithms**
- Two methods to enforce integrity constraints:
  - **Detection:**
    1. Execute update $u : D \rightarrow D_u$
    2. If D_u is inconsistent then compensate $D_u \rightarrow D'_u$ or undo $D_u \rightarrow D$

    * Also called *posttest*
    * May be costly if undo is very large
  - **Prevention:**
    Execute $u : D \rightarrow D_u$ only if D_u will be consistent

    * Also called *pretest*
    * Generally more efficient
    * Query modification algorithm by Stonebraker (1975) is a preventive method that is particularly efficient in enforcing domain constraints.
      · Add the assertion qualification (constraint) to the update query and check it immediately for each tuple

Semantic Integrity Constraints Enforcement ...

- **Example**: Consider a query for increasing the budget of CAD/CAM projects by 10%:

```
UPDATE PROJ
SET BUDGET = BUDGET * 1.1
WHERE PNAME = ‘‘CAD/CAM’’
```

and the domain constraint

```
CHECK ON PROJ (BUDGET >= 50K AND BUDGET <= 100K)
```

The query modification algorithm transforms the query into:

```
UPDATE PROJ
SET BUDGET = BUDGET * 1.1
WHERE PNAME = ‘‘CAD/CAM’’
      AND NEW.BUDGET >= 50K
      AND NEW.BUDGET <= 100K
```

Distributed Constraints

- Three classes of **distributed integrity constraints/assertions** are distinguished:
  - **Individual** assertions
    - Single relation, single variable
    - Refer only to tuples to be updated independenlty of the rest of the DB
    - e.g., domain constraints
  - **Set-oriented** assertions
    - Single relation, multi variable (e.g., functional dependencies)
    - Multi-relation, multi-variable (e.g., foreign key constraints)
    - Multiple tuples form possibly different relations are involved
  - Assertions involving **aggregates**
    - Special, costly processing of aggregates is required

Distributed Constraints

- Particular difficulties with distributed constraints arise from the fact that relations are fragmented and replicated:
  - Definition of assertions
  - Where to store the assertions?
  - How to enforce the assertions?

Distributed Constraints

- **Definition and storage** of assertions
  - The definition of a new integrity assertion can be started at one of the sites that store the relations involved in the assertion, but needs to be propagated to sites that might store fragments of that relation.
  - Individual assertions
    - The assertion definition is sent to all other sites that contain fragments of the relation involved in the assertion.
    - At each fragment site, check for compatibility of assertion with data
    - If compatible, store; otherwise reject
    - If any of the sites rejects, globally reject
  - Set-oriented assertions
    - Involves joins (between fragments or relations)
    - Maybe necessary to perform joins to check for compatibility
    - Store if compatible

Distributed Constraints

- **Enforcement** of assertions in DDBMS is more complex than in centralized DBMS
- The main problem is to decide where (at which site) to enforce each assertion?
  - Depends on type of assertion, type of update, and where update is issued
- Individual assertions
  - Update = insert
    - enforce at the site where the update is issued (i.e., where the user inserts the tuples)
  - Update = delete or modify
    - Send the assertions to all the sites involved (i.e., where qualified tuples are updated)
    - Each site enforce its own assertion
- Set-oriented assertions
  - Single relation
    - Similar to individual assertions with qualified updates
  - Multi-relation
    - Move data between sites to perform joins
    - Then send the result to the query master site (the site the update is issued)

Conclusion

- Views enable full logical data independence
  - Queries expressed on views are translated into queries expressed on base relationships
  - Views can be updatable and non-updatable
- Three aspects are involved in authorization: (user, operation, data object)
- Semantic integrity constraints maintain database consistency
  - Individual assertions are checked at each fragment site, check for compatibility
  - Set-oriented assertions involve joins between fragments and optimal enforcement of the constraints is similar to distributed query optimization
  - Constraint detection vs. constraint prevention