SEMANTICS DATA CONTROL

- involves
  - view management
  - security control
  - integrity control
- objective
  - ensure that authorized users perform correct operations on the database, contributing to the maintenance of the database integrity

VIEW MANAGEMENT

View – virtual relation

- generated from base relation(s) by a query
- not stored as base relations

Consider relation E

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.

VIEW MANAGEMENT (CONT)

Example:

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

SYSAN:

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

VIEW MANAGEMENT (CONT)

Views can be manipulated as base relations

Example:

SELECT	ENAME, JNO, RESP
FROM	SYSAN, G
WHERE	SYSAN.ENO = G.ENO

QUERY MODIFICATION

Queries expressed on views must be re-expressed as queries on base relations

Example: The previous query on the view becomes

SELECT	ENAME, JNO, RESP
FROM	E, G
WHERE	E.ENO = G.ENO
AND	TITLE = “Syst. Anal.”

VIEW MANAGEMENT (CONT)

- to restrict access

```
CREATE  VIEW    ESAME
AS      SELECT  *
        FROM    E E1, E E2
        WHERE   E1.TITLE = E2.TITLE
        AND     E1.ENO = USER
```

- query issued by user J. Joe with ENO="E1"

```
SELECT  *
FROM    ESAME
```

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E6	L. Chu	Elect. Eng.

VIEW UPADTES

- updatable

CREATE	**VIEW**	SYSAN(ENO, ENAME)
AS	**SELECT**	ENO, ENAME
	FROM	E
	WHERE	TITLE = “Syst. Anal.”

- non-updatable

CREATE	**VIEW**	EG(ENAME, RESP)
AS	**SELECT**	ENAME, RESP
	FROM	E, G
	WHERE	E.ENO = G.ENO

View Updatability Problem: How to determine the updatabilities of a view?

View Update Problem: How to translate a view update to some base relation update(s)?

VIEW MANAGEMENT IN DDBMS

- views might be derived from fragments
- view definition storage should be treated as database storage
- query modification results in a distributed query
- view evaluations might be costly if base relations are distributed
  - use **snapshots**
    - static views – do not reflect the updates to the base relations
    - managed as temporary relations – only access path is sequential scan
    - bad selectivity – snapshots behave as pre-calculated answers
    - periodic recalculation
  - use **materialized views**
    - a view can be materialized by storing the tuples of the view as a relation in the database
    - materialized views can be used to speed up the processing of complex queries

VIEW MANAGEMENT IN DDBMS (cont)

* a materialized view gets out-of-date whenever the underlying base relations are modified.
* the process of making a materialized view consistent with the base relations from which it is derived is called **view maintenance**
* a view can be maintained in an immediate or a deferred maner
* **immediate view maintenance** algorithms maintain the view inside the transaction that modifies the base relations
  can significantly slow down update transactions
* **deferred view maintenance** algorithms maintain the view outside the transaction that modifies the base relations
* **incremental view maintenance** algorithms avoid the overhead of recomputing the whole view, they only compute the changes to a view
* view maintenance problem is still an active research topic in data warehouse research

DATA SECURITY

- data protection
  - prevent the physical content of data to be understood by unauthorized users
  - encryption/decryption
    * data encryption standard
    * public-key encryption
- authorization control
  - only authorized users perform operations they are allowed to on the database
    * identification of subjects and objects
    * authentication of subjects
    * granting of rights (authorization matrix)

SEMANTIC INTEGRITY CONTROL

maintain database consistency by enforcing a set of constraints defined on the database

- 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 in the relational model
- two components
  - integrity constraint specification
  - integrity constraint enforcement

SEMANTIC INTEGRITY CONTROL (CONT)

- procedural
  - control embedded in each application program
- declarative
  - assertions in predicate calculus
    - ∗ easy to define constraints
    - ∗ definition of database consistency clear
    - ∗ inefficient to check assertions for each update
      - · limit the search space
      - · decrease the number of data accesses and assertions
      - · preventive strategies
      - · checking at compile time
  - how to enforce the integrity constraints
    - ∗ use **triggers**
      event-condition-action (ECA)
    - ∗ use **incremental checking** methods instead of full checking methods
    - ∗ need to consider multiple updates in a transaction

CONSTRAINT SPECIFICATION LANGUAGE

Predefined constraints

- specify the more common constraints of the relational model
  - not-null attribute
    ENO **NOT NULL IN** E
  - unique key
    (ENO,JNO) **UNIQUE IN** G
  - foreign key
    JNO **IN** G **REFERENCES** JNO **IN** J
  - functional dependency
    ENO **IN** E **DETERMINES** ENAME

CONSTRAINT SPECIFICATION LANGUAGE (CONT)

precompiled constraints

- express preconditions that must be satisfied by all tuples in a relation for a given update type

  (INSERT, DELETE, MODIFY)

  NEW - ranges over new tuples to be inserted

  OLD - ranges over old tuples to be deleted

  general form

  CHECK ON <relation>

  [WHEN <update type>] <qualification>

CONSTRAINT SPECIFICATION LANGUAGE (CONT)

precompiled constraints

- domain constraint
  CHECK ON J(BUDGET $\geq$ 500000 **AND**
  BUDGET $\leq$ 1000000)
- domain constraint on deletion
  CHECK ON J **WHEN DELETE** (BUDGET = 0)
- transition constraint
  CHECK ON J (**NEW**.BUDGET $>$ **OLD**.BUDGET
  AND NEW.JNO = **OLD**.JNO)

CONSTRAINT SPECIFICATION LANGUAGE (CONT)

General constraints

- constraints that must always be true. Formulae of tuple relational calculus where all variables are quantified.

General Form

CHECK ON <variable>:<relation>,(<qualification>)

- functional dependency
  CHECK ON e1:E, e2:E
  (e1.ENAME = e2.ENAME **IF** e1.ENO = e2.ENO)
- constraint with aggregate function
  CHECK ON g:G, j:J
  (**SUM**(g.DUR **WHERE** g.JNO = j.JNO) $<$ 100
  IF j.JNAME="CAD/CAM")

INTEGRITY ENFORCEMENT

two methods

- detection

  Execute update $u : D \to D_u$
  if D_u is inconsistent then
  compensate $D_u \to D'_u$
  or
  undo $D_u \to D$

- preventive
  execute $u : D \to D_u$ only if D_u will be consistent
  - determine valid programs
  - determine valid states

QUERY MODIFICATION

- preventive
- add the assertion qualification to the update query
- only applicable to tuple calculus formulae with universally quantified variables

Example:

UPDATE	J
SET	BUDGET = BUDGET*1.1
WHERE	JNAME = “CAD/CAM”

becomes

UPDATE	J
SET	BUDGET = BUDGET*1.1
WHERE	JNAME = “CAD/CAM”
AND	**NEW.**BUDGET $\geq$ 500000
AND	**NEW.**BUDGET $\leq$ 1000000

COMPILED ASSERTIONS

Triple (R, T, C) where

R relation

T update type (insert,delete,modify)

C assertion on differential relations

Example: Foreign key assertion (referential constraint)

$\forall$ g $\in$ G, $\exists$ j $\in$ J : g.JNO = j.JNO

Compiled assertions:

(G,**INSERT**,C1), (J,**DELETE**,C2), (J,**MODIFY**,C3)

where

C1: $\forall$ **NEW** $\in G_+$, $\exists$ j $\in$ J: **NEW**.JNO = j.JNO

C2: $\forall$ g $\in$ G, $\forall$ **OLD** $\in$ J-: g.JNO $\neq$ **OLD**.JNO

C3: $\forall$ g $\in$ G, $\forall$ **OLD** $\in$ J-, $\exists$ **NEW** $\in J_+$:

g.JNO $\neq$ **OLD**.JNO **OR OLD**.JNO = **NEW**.JNO

Note: J_+ and J_- are differential relations of J. J_+ contains the tuples inserted into J and J_- contains the tuples deleted from J.

DIFFERENTIAL RELATIONS

Given relation R and update u

R_+	contains tuples inserted by u
R_-	contains tuples deleted by u

Types of u

insert	R_- empty
delete	R_+ empty
modify	$R_+ \cup (R - R_-)$

DIFFERENTIAL RELATIONS (CONT)

Algorithm
Input: Relation R, update u, compiled assertions C_i

Step 1: Generate differential relations R_+ and R_-
Step 2: Retrieve the tuples of R_+ and R_- which
do not satisfy C_i
Step 3: If retrieval is not successful, then the assertion
is valid

Example:
u is delete on J. Enforcing (J, DELETE, C2):
$retrieve\ all$ tuples of J_-
$into$ RESULT
$where$ not(C2)
If RESULT = $\emptyset$, the assertion is verified

DISTRIBUTED INTEGRITY CONTROL

problems:

- definition of constraints
  - consideration for fragments
- where to store
  - replication
  - non-replicated: fragments
- enforcement
  - minimize costs

TYPES OF DISTRIBUTED ASSERTIONS

- individual assertions
  - single relation, single variable
    - domain constraint
- set oriented assertions
  - single relation, multi-variable
    - functional dependency
  - multi-relation, multi-variable
    - foreign key
- assertions involving aggregates

DISTRIBUTED INTEGRITY CONTROL

- assertion definition
  - similar to the centralized techniques
  - transform the assertions to compiled assertions
- assertion storage
  - individual assertions
    - ∗ one relation, only fragments
    - ∗ at each fragment site, check for compatibility
    - ∗ if compatible, store; otherwise reject
    - ∗ if some site rejects, globally reject
    - ∗ see text book example
  - set-oriented assertions
    - ∗ involves joins (between fragments or relations)
    - ∗ may be necessary to perform joins to check for compatibility
    - ∗ store if compatible
    - ∗ see text book example

DISTRIBUTED INTEGRITY CONTROL (CONT)

- assertion enforcement
  - where to enforce each assertion?
    - type of assertion
    - type of update and where update is issues
  - individual assertions
    - update = insert
      - enforce at the site where the update is issues
    - update = qualified update (delete or modify)
      - send the assertions to all the sites involved
      - execute the qualification to obtain R_+ and R_-
      - each site enforces its own assertions (relevant at that site)
  - set-oriented assertions
    - single relation
      - similar to individual assertions with qualified updates
    - multi-relation
      - move data between sites to perform joins
      - send the result to the query master site
      - see text book example