Semantic Data Control

- View management
- Security control
- Integrity control
- Existing partitioning solutions

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."
```
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 view SYSAN and relation ASG(ENO,PNO,RESP,DUR)

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

  translates into

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

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

All views can be queried as base relations, but not all views can be updated as such.

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

```sql
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

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

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

Current systems are very restrictive about supporting updates 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

In Oracle a view is **not updatable** if the defining query expression contains any of the following constructs:

- A set operator
- A DISTINCT operator
- An aggregate or analytic function
- A GROUP BY clause
- Recursive views
- A subquery in a SELECT list
- Joins except if tables are key preserved (a key-preserved table has its key columns preserved through a SQL join) and one base table tuple appears at most once in the view.

- Oracle: `select table_name, column_name, updatable, insertable, deletable from user_updatable_columns;`
**View Management in Distributed Databases/1**

- 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 base relations. The view definition is stored in the (possibly) distributed directory/catalogue.
  - Views definitions 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

**View Management in Distributed Databases/2**

- 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
      - can be managed as temporary relations
      - is subject to periodic recalculation
- **Materialized views** store the computed result of a view to improve performance.
- Updates of materialized views are made faster by making materialized views
  - incrementally maintainable (update by considering changes only; no full recomputation)
  - self-maintainable (update by considering only the view and delta relations)

**Data Security**

- **Data security** protects data against unauthorized access and has two aspects:
  - Data protection
  - Authorization control
- **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** must guarantee that only authorized users perform operations they are allowed to perform on the database.

**Authorization Control/1**

- Three actors are involved in authorization
  - users, who trigger the execution of application programs
  - 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 than 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/2

- Grant and revoke statements are used to authorize triplets (user, operation, data object)
  - GRANT <operations> ON <object> TO <users>
  - REVOKE <operations> ON <object> TO <users>
- Examples:
  - GRANT CREATE ANY INDEX TO Robert;
  - REVOKE ALL PRIVILEGES FROM Robert;
  - GRANT SELECT ON emp TO tom;
- Typically, the creator of objects gets all permissions
  - Might have the permission to GRANT permissions
  - This requires a recursive revoke process

Authorization Control/3

Oracle offers about 200 privileges:

<table>
<thead>
<tr>
<th>Privilege</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE SESSION</td>
<td>EXECUTE ANY OPERATOR</td>
</tr>
<tr>
<td>CREATE [ANY] TABLE</td>
<td>EXECUTE ANY PROCEDURE</td>
</tr>
<tr>
<td>CREATE ANY INDEX</td>
<td>EXECUTE ANY RULE</td>
</tr>
<tr>
<td>CREATE [ANY] VIEW</td>
<td>EXPORT FULL DATABASE</td>
</tr>
<tr>
<td>CREATE [ANY] TYPE</td>
<td>IMPORT FULL DATABASE</td>
</tr>
<tr>
<td>CREATE [ANY] OPERATOR</td>
<td>FLASHBACK ANY TABLE</td>
</tr>
<tr>
<td>CREATE [ANY] PROCEDURE</td>
<td>GRANT ANY OBJECT PRIVILEGE</td>
</tr>
<tr>
<td>CREATE [ANY] TRIGGER</td>
<td>GRANT ANY PRIVILEGE</td>
</tr>
<tr>
<td>CREATE [ANY] CLUSTER</td>
<td>GRANT ANY ROLE</td>
</tr>
<tr>
<td>CREATE [ANY] SEQUENCE</td>
<td>SELECT ANY DICTIONARY</td>
</tr>
<tr>
<td>CREATE [PUBLIC] [ANY] SYNONYM</td>
<td>SELECT ANY SEQUENCE</td>
</tr>
<tr>
<td>CREATE [PUBLIC] DATABASE LINK</td>
<td>SELECT ANY TRANSACTION</td>
</tr>
<tr>
<td>CREATE USER</td>
<td>INSERT ANY TABLE</td>
</tr>
<tr>
<td>CREATE ANY CONTEXT</td>
<td>DELETE ANY TABLE</td>
</tr>
<tr>
<td>CREATE [ANY] DIMENSION</td>
<td>UPDATE ANY TABLE</td>
</tr>
</tbody>
</table>

Authorization Control/4

- Privileges are stored in the directory/catalogue, conceptually as a matrix

<table>
<thead>
<tr>
<th>EMP</th>
<th>ENAME</th>
<th>ASG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Casey</td>
<td>UPDATE</td>
<td>UPDATE</td>
</tr>
<tr>
<td>Jones</td>
<td>SELECT</td>
<td>SELECT WHERE RSP ≠ 'Mgr'</td>
</tr>
<tr>
<td>Smith</td>
<td>NONE</td>
<td>SELECT</td>
</tr>
</tbody>
</table>

- Roles can be used to group privileges.

Distributed Authorization Control/1

- Additional problems of authorization control in a distributed environment stem from the fact that objects and subjects are distributed:
  - remote user authentication
  - management of access 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
Distributed Authorization Control/2

Two solutions are possible:
- (username, password) is replicated at all sites and are communicated between the sites whenever a site modifies user information
  - beneficial if the users move from a site to a site
- (username, password) is not replicated; 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 (i.e., a user accesses DDB from the same site always)

Semantic Integrity Constraints/1

A database is said to be consistent if it satisfies a set of constraints, called semantic integrity constraints.

An integrity constraint is a closed first order formula (i.e., all variables are quantified). The DDBMS must ensure that such constraints are always satisfied.

Conceptually, an integrity constraint can be checked by negating it and checking if the query we get by negating the integrity constraint is empty (satisfied) or not (not satisfied).

IC: \( \forall X, \forall Y (\text{sal}(X, Y) \Rightarrow \text{emp}(X)) \)
Q: \( \neg \forall X, \forall Y (\text{sal}(X, Y) \Rightarrow \text{emp}(X)) \)
\( = \text{sal}(X, Y) \wedge \neg \text{emp}(X) \)
\( = \text{select } X, Y \text{ from sal where } X \text{ not in (select } X \text{ from emp)} \)

Keeping a database consistent by enforcing a set of constraints is a difficult problem (in terms of performance)

Semantic Integrity Constraints/2

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/1

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 which is either true (returns a result) or false (returns no result)
- 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/2

- **Predefined constraints** are based on simple keywords and specify the more common constraints 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/3

- **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/4

- **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')
    ```

### Constraints in SQL/1

- **unique, not null, primary key, foreign key, check**
  ```
  CREATE TABLE customer ( 
    SSN NUMBER(10) 
    CustName CHAR(20) UNIQUE, 
    CustStreet CHAR(30), 
    CustCity CHAR(30) NOT NULL, 
    CustCntry CHAR(3)), 
    PRIMARY KEY (SSN), 
    FOREIGN KEY (CustCntry) REFERENCES country, 
    CHECK (CustCntry IN ('US','CA'))
  )
  ```
- **SOTA:** subqueries in the check constraints are often not supported for performance reasons, e.g., CHECK (EXIST (SELECT * FROM Partners WHERE CustCntry = PartnerName))
Constraints in SQL/2

- General assertion have not been implemented until now:

  ```sql
  CREATE ASSERTION sum_constraint CHECK
  (NOT EXISTS (SELECT * FROM branch
  WHERE
  ( SELECT SUM(amount)
  FROM loan
  WHERE loan.BName = branch.BName)
  >=
  ( SELECT SUM(amount)
  FROM account
  WHERE loan.BName = branch.BName)))
  ```

Semantic Integrity Constraints Enforcement/1

- 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, cf next slide) is a preventive method that is 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/2

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

  ```sql
  UPDATE PROJ
  SET BUDGET = BUDGET * 1.1
  WHERE PNAME = 'CAD/CAM'
  ```

  and the domain constraint

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

  The query modification algorithm of Stonebraker transforms the query into:

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

Distributed Constraints/1

- Three classes of distributed integrity constraints/assertions are distinguished:
  - **Individual** assertions
    - Single relation, single variable
    - Refer only to tuples to be updated independently 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
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?

**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 satisfaction of assertion
  - If satisfied store; otherwise reject
  - If any of the sites rejects, globally reject
- Set-oriented assertions
  - Involves joins (between fragments or relations)
  - Store if satisfied
  - Typically satisfaction must be determined by evaluating query fully (e.g., join all fragments)

**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 modification, and where modification is issued
  - Idea is to check consistency incrementally (we know that DDBS was consistent before)
- Individual assertions
  - Modification = insert
    - enforce at the site where the modification is issued (i.e., the user inserts the tuples)
  - Modification = delete or modify
    - Execute modification on all involved sites (where relation is stored)
    - Send the assertions to all sites involved (i.e., where tuples are modified)
    - Each site enforce its own assertion

**Set-oriented assertions**

- Single relation, e.g., functional dependency ENO \(\Rightarrow\) ENAME
  - Similar to individual assertions with modifications
  - Master site sends temporary modification to all sites, each site checks with local fragment of relation, and sends result of compatibility check back
- Multi-relation
  - Move data between sites to perform joins
  - Then send the result to the query master site (the site the modification is issued)
Current Partitioning Solutions

- We illustrate current partitioning solutions, which allow to get some distributed functionality.
- Partitioning allows a table or index to be subdivided into smaller pieces (called partitions).
- Partitioning refers to disjoint pieces, i.e., no replication.
- PostgreSQL uses inheritance with constraint exclusion to implement partitioning.
- With constraint exclusion the query planner attempts to identify partitions that do not have to be queried.

Partitioning in PostgreSQL

```
CREATE TABLE measurement (city_id INT NOT NULL, logdate DATE NOT NULL, temp INT);
```

```
CREATE TABLE measurement_y2006m02 (CHECK (logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01')) INHERITS (measurement);
```

```
CREATE TABLE measurement_y2006m03 (CHECK (logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01')) INHERITS (measurement);
```

Range Partitioning in Oracle

- Maps data to partitions based on ranges of partition key values.
- Most common type of partitioning and is often used with dates.
- Each partition has a VALUES LESS THAN clause, which specifies a noninclusive upper bound for the partitions.

```
CREATE TABLE orders (order_id NUMBER PRIMARY KEY, order_date DATE NOT NULL, customer_id NUMBER NOT NULL, shipper_id NUMBER) PARTITION BY RANGE (order_date) (PARTITION y1 VALUES LESS THAN (TO_DATE('01-JAN-2006', 'DD-MON-YYYY')), PARTITION y2 VALUES LESS THAN (TO_DATE('01-JAN-2007', 'DD-MON-YYYY')), PARTITION y3 VALUES LESS THAN (TO_DATE('01-JAN-2008', 'DD-MON-YYYY')));
```

List Partitioning in Oracle

- List partitioning allows to explicitly control how rows map to partitions.
- A list of discrete values for the partitioning key is specified in the description for each partition.
- List partitioning can partition unordered sets of data.

```
CREATE TABLE sales_list (salesman_id NUMBER(5), salesman_name VARCHAR2(30), sales_state VARCHAR2(20), sales_amount NUMBER(10), sales_date DATE) PARTITION BY LIST (sales_state) (PARTITION sales_west VALUES ('California', 'Hawaii'), PARTITION sales_east VALUES ('New York', 'Florida'), PARTITION sales_other VALUES (DEFAULT));
```
Hash Partitioning in Oracle

- Hash partitioning is used if sizes of ranges are unknown or skewed

```
CREATE TABLE sales_hash (  
salesman_id NUMBER(5),  
salesman_name VARCHAR2(30),  
sales_amount NUMBER(10),  
week_no NUMBER(2))  
PARTITION BY HASH(salesman_id)  
PARTITIONS 4  
STORE IN (ts1, ts2, ts3, ts4);
```

Reference Partitioning in Oracle

- Reference partitioning deals with two tables in a one-to-many relationship.
- Requires a foreign key

```
CREATE TABLE order_items (  
order_id NUMBER NOT NULL,  
product_id NUMBER NOT NULL,  
price NUMBER,  
quantity NUMBER,  
CONSTRAINT order_items_fk  
FOREIGN KEY (order_id) REFERENCES orders)  
PARTITION BY REFERENCE (order_items_fk);
```

Conclusion/1

- Semantic data control includes views, authorization and integrity constraints.
- The techniques are very similar to centralized solutions; the main issue (also for centralized solutions) is performance.
- 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

Conclusion/2

- Three aspects are involved in authorization: (user, operation, data object).
- Semantic integrity constraints maintain database consistency:
  - Individual assertions are checked for each fragment
  - Set-oriented assertions involve joins between fragments and optimal enforcement of the constraints is similar to distributed query optimization
  - Constraint detection vs. constraint prevention
  - SQL foresees assertions but they are not yet supported by DBMS