Principles of Distributed Database Systems

M. Tamer Özsu

Patrick Valduriez

Outline

- Introduction
- Distributed and Parallel Database Design
- Distributed Data Control
- Distributed Query Processing
- Distributed Transaction Processing
- Data Replication
- Database Integration – Multidatabase Systems
- Parallel Database Systems
- Peer-to-Peer Data Management
- Big Data Processing
- NoSQL, NewSQL and Polystores
- Web Data Management

Outline

- Distributed Data Control
    - View management
    - Data security
    - Semantic integrity control

Semantic 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.

Outline

- Distributed Data Control
    - View management
    - Data security
    - Semantic integrity control

View Management

View – virtual relation

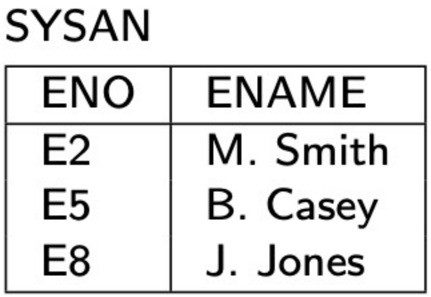
- generated from base relation(s) by a query
- not stored as base relations

Example :

```
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

Views can be manipulated as base relations

Example :

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

Query Modification

Queries expressed on views

↓

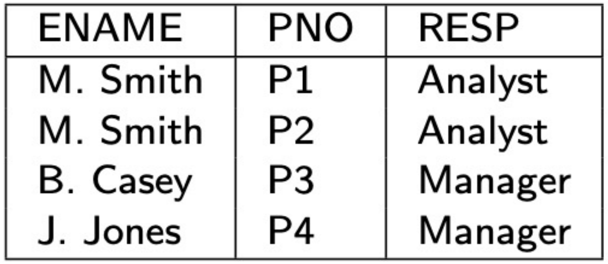
Queries expressed on base relations

Example :

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

↓

```
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

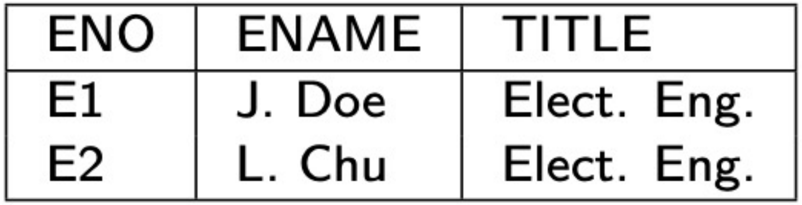
View Management

- To restrict access

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

- Query

```
SELECT *
FROM   ESAME
```



ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	L. Chu	Elect. Eng.

View Updates

- Updatable

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

- Non-updatable

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

View Management in Distributed DBMS

- 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 materialized views

Materialized View

- Origin: snapshot in the 1980's
  - Static copy of the view, avoid view derivation for each query
  - But periodic recomputing of the view may be expensive
- Actual version of a view
  - Stored as a database relation, possibly with indices
- Used much in practice
  - DDBMS: No need to access remote, base relations
  - Data warehouse: to speed up OLAP
    - Use aggregate (SUM, COUNT, etc.) and GROUP BY

Materialized View Maintenance

- Process of updating (refreshing) the view to reflect changes to base data
  - Resembles data replication but there are differences
    - View expressions typically more complex
    - Replication configurations more general
- View maintenance policy to specify:
  - When to refresh
  - How to refresh

When to Refresh a View

- Immediate mode
  - As part of the updating transaction, e.g. through 2PC
  - View always consistent with base data and fast queries
  - But increased transaction time to update base data
- Deferred mode (preferred in practice)
  - Through separate refresh transactions
    - No penalty on the updating transactions
  - Triggered at different times with different trade-offs
    - Lazily: just before evaluating a query on the view
    - Periodically: every hour, every day, etc.
    - Forcedly: after a number of predefined updates

How to Refresh a View

- Full computing from base data
  - Efficient if there has been many changes
- Incremental computing by applying only the changes to the view
  - Better if a small subset has been changed
  - Uses differential relations which reflect updated data only

Differential Relations

Given relation R and update u

R^+ contains tuples inserted by u

R^- contains tuples deleted by u

Type of u

insert R^- empty

delete R^+ empty

modify $R^+ \cup (R - R^-)$

Refreshing a view V is then done by computing

$V^+ \cup (V - V^-)$

computing V^+ and V^- may require accessing base data

Example

```
  EG =         SELECT DISTINCT ENAME, RESP
               FROM   EMP, ASG
               WHERE  EMP.ENO=ASG.ENO

 EG+=          (SELECT DISTINCT ENAME, RESP
               FROM   EMP, ASG+
               WHERE  EMP.ENO=ASG+.ENO) UNION
               (SELECT DISTINCT ENAME, RESP
               FROM   EMP+, ASG
               WHERE  EMP+.ENO=ASG.ENO) UNION
               (SELECT DISTINCT ENAME, RESP
               FROM   EMP+, ASG+
               WHERE  EMP+.ENO=ASG+.ENO)
```

Techniques for Incremental View Maintenance

- Different techniques depending on:
  - View expressiveness
    - Non recursive views: SPJ with duplicate elimination, union and aggregation
    - Views with outerjoin
    - Recursive views
- Most frequent case is non recursive views
  - Problem: an individual tuple in the view may be derived from several base tuples
    - Example: tuple ⟨M. Smith, Analyst⟩ in EG corresponding to
      - ⟨E2, M. Smith, … ⟩ in EMP
      - ⟨E2,P1,Analyst,24 ⟩ and ⟨E2,P2,Analyst,6⟩ in ASG
    - Makes deletion difficult
  - Solution: Counting

Counting Algorithm

- Basic idea
  - Maintain a count of the number of derivations for each tuple in the view
  - Increment (resp. decrement) tuple counts based on insertions (resp. deletions)
  - A tuple in the view whose count is zero can be deleted
- Algorithm
  1. Compute V^+ and V^- using V, base relations and diff. relations
  2. Compute positive in V^+ and negative counts in V^-
  3. Compute $V^+ \cup (V - V^-)$, deleting each tuple in V with count=0
- Optimal: computes exactly the view tuples that are inserted or deleted

Exploiting Data Skew

- Basic idea
  - Partition the relations on heavy / light values for join attributes
    - Threshold depends on data size and user parameter
  - Maintain the join of different parts using different plans
    - Most cases done using delta processing (Counting)
    - Few cases require pre-materialization of auxiliary views
  - Rebalance the partitions to reflect heavy ↔ light changes
    - Reasons for change:
      - Much more/less occurrences of a value than before
      - The heavy/light threshold changes due to change in data size
    - Update times are *amortized* to account for occasional rebalancing

Example: Triangle Count

$$\sum_{a,b,c} R(a,b) \cdot S(b,c) \cdot T(c,a)$$

- Data model
  - Relations are functions mapping tuples to multiplicities
  - Updates also map tuples to multiplicities
- Triangle count query
  - Joins relations *R*, *S* and *T* on common variables
  - Aggregates away all variables *a*, *b* and *c*
  - Sums over the product of the multiplicities of matching tuples
- Next: Maintenance under single-tuple update to *R*
  - Single-tuple update ΔR maps (a', b') to multiplicity m
  - If $m > 0$ $(m < 0$ $)$ then the update is an insert (delete)

Naïve Maintenance for Triangle Count

- Compute from scratch

$$newR := R + \Delta R$$

$$\sum_{a,b,c} newR(a,b) \cdot S(b,c) \cdot T(c,a)$$

- Maintenance time: *O(N*$^{1.5}$*)*
  - Assuming the input relations have size *O(N)*
  - Using existing worst-case optimal join algorithms

- No extra space needed

Delta Processing for Triangle Count

- Compute the change

$$\sum_{a,b,c} R(a,b)\cdot S(b,c)\cdot T(c,a)$$
$$+$$
$$\Delta R(a',b')\cdot \sum_{c} S(b',c)\cdot T(c,a')$$

- Maintenance time: *O(N)*
  - Intersect the set of *c* values paired with *b'* in *S* and with *a'* in *T*

- No extra space needed

Materialized View for Triangle Count

- Compute the change using materialized views

$$\text{Pre-materialize } V_{ST}(b,a) = \sum_c S(b,c) \cdot T(c,a)$$

$$\sum_{a,b,c} R(a,b) \cdot S(b,c) \cdot T(c,a)$$

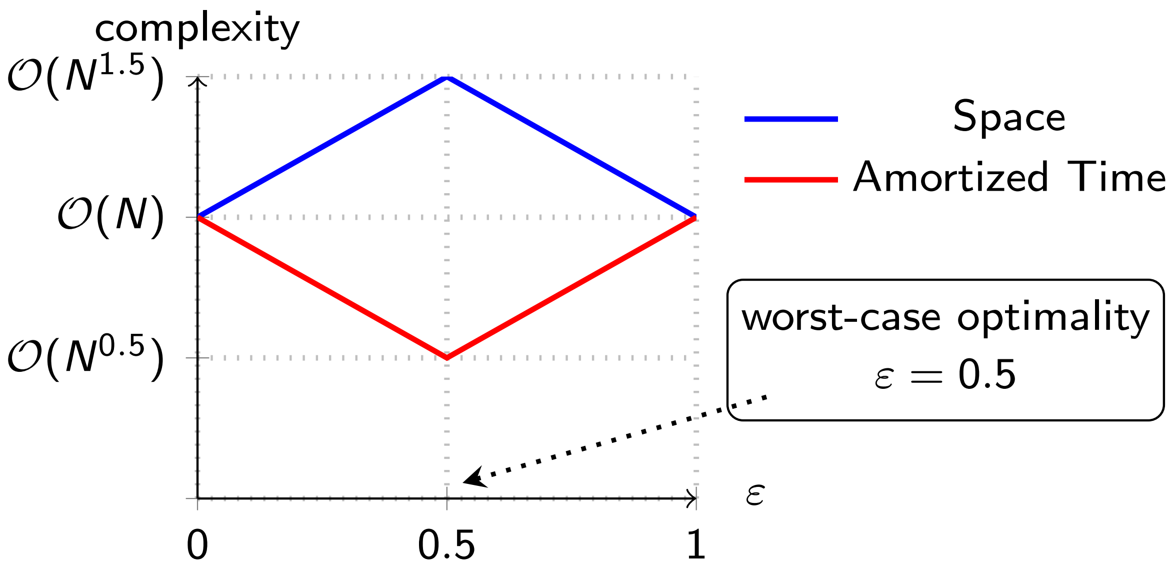
$$+$$

$$\Delta R(a',b') \cdot V_{ST}(b',a')$$

- Maintenance time:
  - ❑ Updates to *R*: *O(1)* time to look up in V_{ST}
  - ❑ Updates to *S* and *T*: *O(N)* time to maintain V_{ST}
- Extra $O(N^2)$ space needed for the view V_{ST}

Data Skew for Triangle Count

- For $\varepsilon \in [0,1]$, the triangle count can be maintained with $O(N^{\max\{\varepsilon,1-\varepsilon\}})$ update time and $O(N^{1+\min\{\varepsilon,1-\varepsilon\}})$ space.

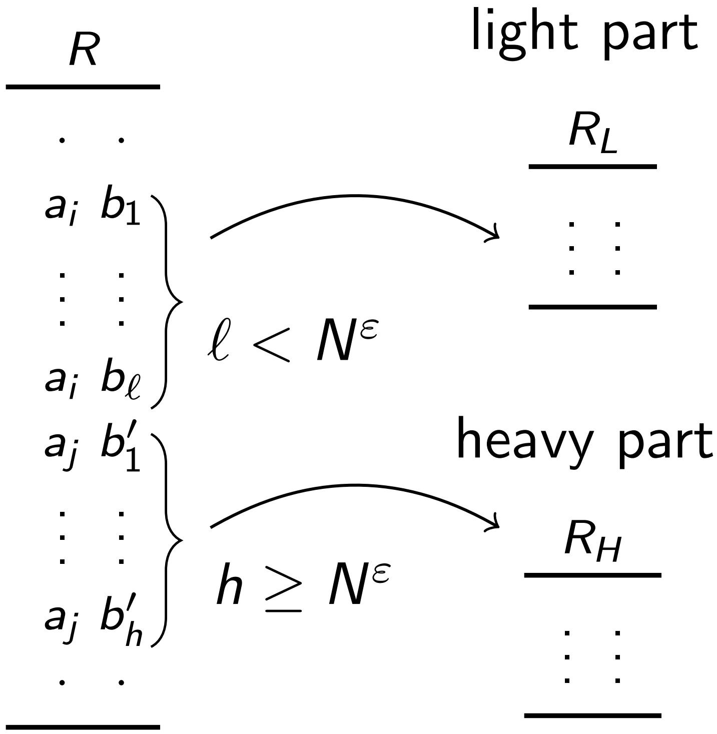


worst-case optimality
$\varepsilon = 0.5$

- No algorithm can attain $O(N^{0.5-\gamma})$ for any $\gamma > 0$.

Heavy/Light Partitioning of Relations

- Partition *R* on *a* into a light part R_L and a heavy part R_H
    - ❑ $R_L = \{t \ \in R \mid |\sigma_{a=t.a}R| < N^{\varepsilon}\}$
    - ❑ $R_H = \{t \mid t \in R, t \notin R_L\}$
- Cardinality bounds
    - ❑ For every value *a'*: $|\sigma_{a=a'}R_L| < N^{\varepsilon}$
    - ❑ $|\pi_a R_H| \leq N^{1-\varepsilon}$
- Also partition *S* on *b* and *T* on *c*



Maintenance for Skew-Aware Views

$$\sum_{U,V,W\in\{L,H\}} \sum_{a,b,c} R_U(a,b)\cdot S_V(b,c)\cdot T_W(c,a)$$

- For joins of light parts only or heavy parts only
  - Maintenance using delta processing (Counting)

- For joins of a heavy part with a light part
  - Maintenance using pre-materialized views

- Next: Consider one skew-aware view at a time
  - Single-tuple update $\Delta R(a', b')$ to *R*

Case 1: Light-Light Interaction

- Skew-aware views (any partition of *R*)

$$\sum_{a,b,c} R(a,b) \cdot S_L(b,c) \cdot T_L(c,a)$$

- Maintenance under update $\Delta R(a', b')$

$$\Delta R(a', b') \cdot \sum_{c} S_L(b', c) \cdot T_L(c, a')$$

  - There are at most N^{ε} *c* values paired with *b'*
  - For each such value *c*, we check (*c,a'*) in T_L in O(1)

- Maintenance time: $O(N^{\varepsilon})$

Case 2: Heavy-Heavy Interaction

- Skew-aware view (any partition of *R*)

$$\sum_{a,b,c} R(a,b) \cdot S_H(b,c) \cdot T_H(c,a)$$

- Maintenance under update $\Delta R(a',b')$

$$\Delta R(a',b') \cdot \sum_{c} T_H(c,a') \cdot S_H(b',c)$$

  - There are at most $N^{1-\varepsilon}$ *c* values paired with *a'* in T_H
  - For each such value *c*, we check (*b'*,*c*) in S_H in O(1)

- Maintenance time: $O(N^{1-\varepsilon})$

Case 3: Light-Heavy Interaction

- Skew-aware view (any partition of *R*)

$$\sum_{a,b,c} R(a,b) \cdot S_L(b,c) \cdot T_H(c,a)$$

- Two possible maintenance plans

$$\Delta R(a',b') \cdot \sum_{c} S_L(b',c) \cdot T_H(c,a') \qquad \Delta R(a',b') \cdot \sum_{c} T_H(c,a') \cdot S_L(b',c)$$

1. There are at most N^{ε} *c* values paired with *b'* in S_L
2. There are at most $N^{1-\varepsilon}$ *c* values paired with *a'* in T_H

- Maintenance time: $O(\min\{N^{\varepsilon}, N^{1-\varepsilon}\}) = O(N^{\min\{\varepsilon,1-\varepsilon\}})$

Case 4: Heavy-Light Interaction

- Skew-aware view (any partition of *R*)

$$\sum_{a,b,c} R(a,b) \cdot S_H(b,c) \cdot T_L(c,a)$$

- Maintenance under update $\Delta R(a', b')$
    - Materialize auxiliary view

$$V_{ST}(b,a) = \sum_{c} S_H(b,c) \cdot T_L(c,a)$$

    - Lookup in the view

$$\Delta R(a', b') \cdot V_{ST}(b', a')$$

- Maintenance time
    - *O(1)* for the skew-aware view
    - $O(N^{\max\{\varepsilon, 1-\varepsilon\}})$ for the auxiliary view
- Size of auxiliary view: $O(N^{1+\min\{\varepsilon, 1-\varepsilon\}})$

View Self-maintainability

- A view is self-maintainable if the base relations need not be accessed
  - Not the case for the Counting algorithm
- Self-maintainability depends on views' expressiveness
  - Most SPJ views are often self-maintainable wrt. deletion and modification, but not wrt. insertion
  - Example: a view *V* is self-maintainable wrt to deletion in *R* if the key of *R* is included in *V*

Outline

- Distributed Data Control
  - View management
  - Data security
  - Semantic integrity control

Data Security

- Data protection
  - Prevents the physical content of data to be understood by unauthorized users
  - Uses encryption/decryption techniques (Public key)
- Access control
  - Only authorized users perform operations they are allowed to on database objects
  - Discretionary access control (DAC)
    - Long been provided by DBMS with authorization rules
  - Multilevel access control (MAC)
    - Increases security with security levels

Discretionary Access Control

- Main actors
  - Subjects (users, groups of users) who execute operations
  - Operations (in queries or application programs)
  - Objects, on which operations are performed
- Checking whether a subject may perform an op. on an object
  - Authorization= (subject, op. type, object def.)
  - Defined using GRANT OR REVOKE
  - Centralized: one single user class (admin.) may grant or revoke
  - Decentralized, with op. type GRANT
    - More flexible but recursive revoking process which needs the hierarchy of grants

Problem with DAC

- A malicious user can access unauthorized data through an authorized user
- Example
  - User A has authorized access to *R* and *S*
  - User B has authorized access to *S* only
  - B somehow manages to modify an application program used by A so it writes *R* data in *S*
  - Then B can read unauthorized data (in *S*) without violating authorization rules
- Solution: multilevel security based on the famous Bell and Lapuda model for OS security

Multilevel Access Control

- Different security levels (*clearances*)
  - *Top Secret* > *Secret* > *Confidential* > *Unclassified*
- Access controlled by 2 rules:
  - No read up
    - subject *S* is allowed to read an object of level *L* only if *level(S)* ≥ *L*
    - Protect data from unauthorized disclosure, e.g. a subject with secret clearance cannot read top secret data
  - No write down:
    - subject *S* is allowed to write an object of level *L* only if *level(S)* ≤ *L*
    - Protect data from unauthorized change, e.g. a subject with top secret clearance can only write top secret data but not secret data (which could then contain top secret data)

MAC in Relational DB

- A relation can be classified at different levels:
  - Relation: all tuples have the same clearance
  - Tuple: every tuple has a clearance
  - Attribute: every attribute has a clearance
- A classified relation is thus multilevel
  - Appears differently (with different data) to subjects with different clearances

Example

PROJ*: classified at attribute level

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	DB Develop.	C	135000	S	New York	S
P3	S	CAD/CAM	S	250000	S	New York	S

PROJ* as seen by a subject with confidential clearance

PNO	SL1	PNAME	SL2	BUDGET	SL3	LOC	SL4
P1	C	Instrumentation	C	150000	C	Montreal	C
P2	C	DB Develop.	C	Null	C	Null	C

Distributed Access Control

- Additional problems in a distributed environment
  - Remote user authentication
    - Typically using a directory service
      - Should be replicated at some sites for availability
  - Management of DAC rules
    - Problem if users' group can span multiple sites
      - Rules stored at some directory based on user groups location
      - Accessing rules may incur remote queries
  - Covert channels in MAC

Covert Channels

- Indirect means to access unauthorized data
- Example
  - Consider a simple DDB with 2 sites: C (confidential) and S (secret)
  - Following the "no write down" rule, an update from a subject with secret clearance can only be sent to S
  - Following the "no read up" rule, a read query from the same subject can be sent to both C and S
  - But the query may contain secret information (e.g. in a select predicate), so is a potential covert channel
- Solution: replicate part of the DB
  - So that a site at security level *L* contains all data that a subject at level *L* can access (e.g. S above would replicate the confidential data so it can entirely process secret queries)

Outline

- Distributed Data Control
  - View management
  - Data security
  - Semantic integrity control

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

- Procedural
  - Control embedded in each application program
- Declarative
  - Assertions in predicate calculus
  - Easy to define constraints
  - Definition of database consistency clear
  - But inefficient to check assertions for each update
    - Limit the search space
    - Decrease the number of data accesses/assertion
    - Preventive strategies
    - Checking at compile time

Constraint Specification Language

Predefined constraints

specify the more common constraints of the relational model

- Not-null attribute

```
ENO NOT NULL IN EMP
```

- Unique key

```
(ENO, PNO) UNIQUE IN ASG
```

- Foreign key

  A key in a relation *R* is a foreign key if it is a primary key of another relation *S* and the existence of any of its values in *R* is dependent upon the existence of the same value in *S*

```
PNO IN ASG REFERENCES PNO IN PROJ
```

- Functional dependency

```
ENO IN EMP DETERMINES ENAME
```

Constraint Specification Language

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

Precompiled constraints

- Domain constraint

```
CHECK ON PROJ (BUDGET≥500000 AND BUDGET≤1000000)
```

- Domain constraint on deletion

```
CHECK ON PROJ WHEN DELETE (BUDGET = 0)
```

- Transition constraint

```
CHECK ON PROJ (NEW.BUDGET > OLD.BUDGET AND
                    NEW.PNO = OLD.PNO)
```

Constraint Specification Language

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:EMP, e2:EMP
   (e1.ENAME = e2.ENAME IF e1.ENO = e2.ENO)
```

- Constraint with aggregate function

```
CHECK ON g:ASG, j:PROJ
   (SUM(g.DUR WHERE g.PNO = j.PNO) < 100 IF
       j.PNAME = "CAD/CAM")
```

Integrity Enforcement

Two methods

- Detection

  Execute update *u*: $D \rightarrow D_u$

  If D_u is inconsistent then

      if possible: compensate $D_u \rightarrow D_u'$

      else

        undo $D_u \rightarrow D$

- Preventive

  Execute *u*: $D \rightarrow D_u$ 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

```
UPDATE  PROJ
SET     BUDGET = BUDGET*1.1
WHERE   PNAME = "CAD/CAM"
                ↓
UPDATE  PROJ
SET     BUDGET = BUDGET*1.1
WHERE   PNAME = "CAD/CAM"
AND     NEW.BUDGET ≥ 500000
AND     NEW.BUDGET ≤ 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

$\forall g \in \text{ASG}, \exists j \in \text{PROJ} : g.\text{PNO} = j.\text{PNO}$

Compiled assertions:

(ASG, **INSERT**, C1), (PROJ, **DELETE**, C2), (PROJ, **MODIFY**, C3)

where

C1: $\forall \mathbf{NEW} \in \text{ASG}+ \quad \exists j \in \text{PROJ}: \text{NEW.PNO} = j.\text{PNO}$

C2: $\forall g \in \text{ASG}, \forall \mathbf{OLD} \in \text{PROJ}^{-} : g.\text{PNO} \neq \mathbf{OLD}.\text{PNO}$

C3: $\forall g \in \text{ASG}, \forall \mathbf{OLD} \in \text{PROJ}^{-} \; \exists \mathbf{NEW} \in \text{PROJ}^{+}:$

$g.\text{PNO} \neq \mathbf{OLD}.\text{PNO} \text{ OR } \mathbf{OLD}.\text{PNO} = \mathbf{NEW}.\text{PNO}$

Differential Relations

Given relation R and update u

R^+ contains tuples inserted by u

R^- contains tuples deleted by u

Type of u

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

Differential Relations

Algorithm:

Input: Relation R, update u, compiled assertion 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 (EMP, DELETE, C2) :

```
    retrieve all tuples of EMP-
    into RESULT
    where not(C2)
```

If RESULT = { }, 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 all the sites reject, globally reject
    - Set-oriented assertions
        - Involves joins (between fragments or relations)
        - May be necessary to perform joins to check for compatibility
        - Store if compatible

Distributed Integrity Control

- Assertion Enforcement
  - Where to enforce each assertion depends on
    - Type of assertion
    - Type of update and where update is issued
  - Individual Assertions
    - If update = insert
      - Enforce at the site where the update is issued
    - If update = qualified
      - Send the assertions to all the sites involved
      - Execute the qualification to obtain R^+ and R^-
      - Each site enforces its own assertion
  - Set-oriented Assertions
    - Single relation
      - Similar to individual assertions with qualified updates
    - Multi-relation
      - Move data to perform joins; then send the result to query master site

Conclusion

- Solutions initially designed for centralized systems have been significantly extended for distributed systems
  - Materialized views and group-based discretionary access control
- Semantic integrity control has received less attention and is generally not well supported by distributed DBMS products
- Full data control is more complex and costly in distributed systems
  - Definition and storage of the rules (site selection)
  - Design of enforcement algorithms which minimize communication costs