DISTRIBUTED DATABASES AND FRAGMENTATION

Fragmentation Types

Horizontal

- Fragments ***Ri*** contains tuples with the same schema as the table ***R***.
- Each fragment can be seen as the result of a selection applied on ***R***.
- In order to reconstruct ***R***:

$$R = R_1 \cup R_2 \cup \cdots$$

Vertical

- Schemas for fragments ***Ri*** have a subset of the attributes of the table ***R.***
- Each fragment can be seen as the result of a projection applied on R.
- In order to reconstruct ***R***:

$$R = R_1 \times R_2 \times \cdots$$

Transparency Levels

- Fragmentation Transparency
- Allocation Transparency
- Language Transparency
- No Transparency

Football Players

Consider the database fragmented according to the Country where a team belongs:

PLAYER (Name, Team, Country, Role)

GOAL (PlayerName, GameDate, Minute, GoalType, Description)

The GOAL table describes the goals scored by the various players during their career and has a fragmentation which is derived from that of PLAYER. Assume that names identify players.

Describe at the three transparency levels the required SQL statements for **changing the player's team**. Assume that you know the player's name and the destination team and country, while his role is unchanged.

Two different cases

(a) The old and new team are in the same country

(b) The old and new team are in different countries

(a) Marco Borriello Milan → Rome

(b) Zlatan Ibrahimovich Barcelona → Milan

(a) The old and new team are in the same country

Borriello Milan → Rome

Fragmention transparency

upadate Player
set Team = “Rome”
where name = “Marco Borriello”

Allocation transparency

upadate *ItalianPlayer*
set Team = “Rome”
where name = “Marco Borriello”

(b) The old and new team are in different countries

Zlatan Ibrahimovich Barcelona → Milan

Fragmention transparency

upadate Player
set Team = “Milan”
where name = “Zlatan Ibrahimovich”

Allocation transparency

Imagine that for the horizontal fragmentation we created:

```
create table GoalsScoredEverSinceByPlayersCurrentlyEnrolledInItaly
    (  PlayerName varchar(50)  references Player(Name)
                                on update cascade on delete no action
     ... )

create table GoalsScoredEverSinceByPlayersCurrentlyEnrolledInSpain
    (  PlayerName varchar(50)  references Player(Name)
                                on update cascade on delete no action
     ... )
```

1. Insert Zlatan's data into → **ItalianPlayer**
2. Insert all of Zlatan's goals into the italian goal record (rom the spanish one) → **Goals...Italy**
3. Delete goals from the → **Goals...Spain**
4. Delete Zlatan from → **SpanishPlayer**

1. insert into **ItalianPlayer**

   select Name, ***‘Milan', ‘Italy',*** Role

   from **SpanishPlayer**

   where Name = ‘Zlatan Ibrahimovic‘

2. insert into **Goals..Italy**

   select *

   from **Goals..Spain**

   where PlayerName = ‘Zlatan Ibrahimovic‘

3. delete from **Goals..Spain**

   where PlayerName = ‘Zlatan Ibrahimovic‘

4. delete from **SpanishPlayer**

   where Name = ‘Zlatan Ibrahimovic‘

L

Consider the database:

Production (<u>SerialNumber</u>, PartType,
Model, Quantity, Machine)

Pickup (<u>SerialNumber</u>, <u>Lot</u>, Customer,
SalesPerson, Amount)

Customer (<u>Name</u>, City, Address)

SalesPerson (<u>Name</u>, City, Address)

Design the horizontal fragmentation of the tables **Production** and **Pickup** depending on the **PartType** value (‘*Keyboard*’, ‘*Screen*’, ‘*CPU-box*’ e ‘*Cable*’).

Assume four Production centres (one for each component) located in *Milan*, *Turin*, *Rome, Naples*,

and of **Customer** and **SalesPerson** on three Cities: *Milan*, *Turin*, *Rome*.

The distributed DB contains the following tables

Production**MiKbr** (<u>SerialNumber</u>, PartType, Model, Quantity, Machine)

Production**ToScr** (<u>SerialNumber</u>, PartType, Model, Quantity, Machine)

Production**RmCpu** (<u>SerialNumber</u>, PartType, Model, Quantity, Machine)

Production**NaCbl** (<u>SerialNumber</u>, PartType, Model, Quantity, Machine)

Production =
ProductionMiKbr ∪ **ProductionToScr** ∪ **ProductionRmCpu** ∪ **ProductionNaCpl**

Pickup**MiKbr** (SerialNumber, Lot, Client, SalesPerson, Amount)

Pickup**ToScr** (SerialNumber, Lot, Client, SalesPerson, Amount)

Pickup**RmCpu** (SerialNumber, Lot, Client, SalesPerson, Amount)

Pickup**NaCbl** (SerialNumber, Lot, Client, SalesPerson, Amount)

Pickup = PickupMiKbr $\cup$ **PickupToScr** $\cup$ **PickupRmCpu** $\cup$ **PickupNaCpl**

Client**Mi** (<u>Name</u>, City, Address)
Client**To** (<u>Name</u>, City, Address)
Client**Rm** (<u>Name</u>, City, Address)

Client = ClientMi $\cup$ **ClientTo** $\cup$ **ClientRm**

SalesPerson**Mi** (Name, City, Address)
SalesPerson**To** (Name, City, Address)
SalesPerson**Rm** (Name, City, Address)

SalesPerson = SalesPersonMi $\cup$ SalesPersonTo $\cup$ SalesPersonRm

Express the following queries at transparency levels of fragmentation, allocation and language:

Determine the available quantity of the product ’77Y6878’

Transparency of fragmentation

SELECT Quantity
FROM Production
WHERE SerialNumber='77Y6878'

Transparency of allocation

```
SELECT Quantity
FROM ProductionMiKbr
WHERE SerialNumber='77Y6878'
UNION
 SELECT Quantity
 FROM ProductionToScr
 WHERE SerialNumber='77Y6878'
UNION
 SELECT Quantity
 FROM ProductionRmCpu
 WHERE SerialNumber='77Y6878'
UNION
 SELECT Quantity
 FROM ProductionNaCbl
 WHERE SerialNumber='77Y6878')
```

Transparency of allocation

```
SELECT Quantity
FROM ProductionMiKbr
WHERE SerialNumber='77Y6878'
IF :empty then
 (SELECT Quantity
  FROM ProductionToScr
  WHERE SerialNumber='77Y6878'
  UNION
  SELECT Quantity
  FROM ProductionRmCpu
  WHERE SerialNumber='77Y6878'
  UNION
  SELECT Quantity
  FROM ProductionNaCbl
  WHERE SerialNumber='77Y6878')
```

Transparency of language

```
SELECT Quantity
FROM ProductionMiKbr@Milan
WHERE SerialNumber='77Y6878'
IF :empty then
 (SELECT Quantity
  FROM ProductionToScr@Turin
  WHERE SerialNumber='77Y6878'
  UNION
  SELECT Quantity
  FROM ProductionRmCpu@Rome
  WHERE SerialNumber='77Y6878'
  UNION
  SELECT Quantity
  FROM ProductionNaCbl@Naples
  WHERE SerialNumber='77Y6878')
```

Determine the clients who have bought a lot from the SalesPerson ‘Bianchi’, who has an office in Rome

Transparency of fragmentation

SELECT Customer
FROM Pickup
WHERE SalesPerson=‘Bianchi’

Transparency of allocation

SELECT Client FROM PickupMiKbr
WHERE SalesPerson='Bianchi'
union
SELECT Client FROM PickupToScr
WHERE SalesPerson='Bianchi'
union
SELECT Client FROM PickupRmCpu
WHERE SalesPerson='Bianchi'
union
SELECT Client FROM PickupNaCbl
WHERE SalesPerson='Bianchi'

Transparency of language

```
SELECT Client FROM PickupMiKbr@Milan
WHERE SalesPerson='Bianchi'
union
SELECT Client FROM PickupToScr@Turin
WHERE SalesPerson='Bianchi'
union
SELECT Client FROM PickupRmCpu@Rome
WHERE SalesPerson='Bianchi'
union
SELECT Client FROM PickupNaCbl@Naples
WHERE SalesPerson='Bianchi'
```

Determine the machines used for the production of the parts type ‘Keyboard’ sold to the Customer ‘Rossi’

Transparency of fragmentation

```
SELECT Machine
FROM Production P1 JOIN Pickup P2
    ON
(P1.SerialNumber=P2.SerialNumber)
WHERE Client = 'Rossi' AND
            PartType = 'Keyboard'
```

Transparency of allocation

This time the query is easier because only one fragment is totally dedicated to components of type 'Keyboard'

```
SELECT Machine
FROM ProductionMiKbr P1
  JOIN PickupMiKbr P2
  ON (P1.SerialNumber=P2.SerialNumber)
WHERE Client = 'Rossi'
```

Transparency of language

Also this query is easier because only one fragment is totally dedicated to components of type ‘Keyboard’

```
SELECT Machine
FROM ProductionMiKbr@Milan P1
  JOIN PickupMiKbr@Milan P2
  ON (P1.SerialNumber=P2.SerialNumber)
WHERE Client = ‘Rossi’
```

Update the Address of SalesPerson ‘Rossi’, moving from ‘via Po 45’ in ‘Milan’ to ‘Viale Trastevere 150’ in Rome’ (ignore the moving of related data in other tables)

Transparency of fragmentation

```
UPDATE SalesPerson
SET City = ‘Rome’,
      Address=‘ViaIeTrastevere 150’
WHERE Name = ‘Rossi’
```

Transparency of allocation

Delete followed by an Insert

DELETE FROM SalesPerson**Mi**
WHERE Name = ‘Rossi’

Insert INTO SalesPerson**Rm**
VALUES(‘Rossi’, ‘Rome’,
‘Viale Trastevere 150’)

Transparency of language

Delete followed by an Insert

DELETE FROM SalesPersonMi@Milan
WHERE Name = ‘Rossi’

Insert INTO SalesPersonRm@Rome
VALUES(‘Rossi’, ‘Rome’,
‘Viale Trastevere 150’)

Calcolate the sum of amounts of the orders received in Milan, Turin, Rome and Naples (note that the aggregate functions are also distributable)

Transparency of fragmentation

```
SELECT SUM(Amount)
FROM Pickup
```

Transparency of allocation

We need to identify the contribute of each fragment.
To do a sum of sums it is necessary to use a view.

Transparency of allocation

```
CREATE VIEW TotalAm(Tot) AS
SELECT SUM (Amount)
FROM PickupMiKbr
UNION ALL
SELECT SUM (Amount)
FROM PickupToScr
UNION ALL
SELECT SUM (Amount)
FROM PickupRmCpu
UNION ALL
SELECT SUM (Amount)
FROM PickupNaCbl
```

SELECT SUM(Tot) FROM TotalAm

Transparency of allocation

```
CREATE VIEW TotalAm(Tot) AS
SELECT SUM (Amount)
FROM PickupMiKbr@Milan
UNION ALL
SELECT SUM (Amount)
FROM PickupToScr@Turin
UNION ALL
SELECT SUM (Amount)
FROM PickupRmCpu@Rome
UNION ALL
SELECT SUM (Amount)
FROM PickupNaCbl@Naples
```

SELECT SUM(Tot) FROM TotalAm

Progettazione Fisica

Progettazione Fisica

- *Ingresso*:
  - Schema logico della base di dati
  - Caratteristiche del sistema scelto
  - Previsioni sul carico applicativo (queries)

- *Uscita*:
  - Strutture fisiche utilizzate (struttura primaria per ciascuna relazione, eventuali indici secondari)

Progettazione Fisica

- Operazioni più costose:
  - **Selezione** (accesso ad uno o più record sulla base di uno o più attributi)
  - **Join**

Queste operazioni sono molto più efficienti se esistono indici sui campi interessati (*primari* o *secondari*)

Progettazione Fisica

- **Strategie**:
  - La chiave primaria sarà quasi sempre coinvolta in operazioni di selezione o di join
    => spesso utile costruire un indice
    => valutare se utilizzarlo come primario

  - Indici su altri attributi spesso coinvolti in selezioni o join.
    - *B+-tree*: accesso **logaritmico**, utile per intervalli
    - *Hash*: accesso **diretto**, non utile per intervalli

Approccio Sistematico

- Si supponga di avere le operazioni $O_1, O_2, \ldots, O_n$
- Ciascuna con la frequenza $f_1, f_2, \ldots, f_n$
- Per ogni operazione è possibile definire un costo di esecuzione c_i (*numero di accessi a memoria secondaria*)
- Il costo può variare a seconda delle strutture fisiche scelte

La progettazione fisica = minimizzare il costo complessivo:

$$\sum_{i=1}^{n} c_i \, f_i$$

Consideriamo la relazione IMPIEGATO(Matricola, Cognome, Nome, DataNascita) con un numero di tuple pari a N = 10 000 000 abbastanza stabile nel tempo (pur con inserimenti e cancellazioni) e una dimensione di ciascuna tupla pari a L = 100 byte, di cui K = 2 byte per la chiave (Matricola) e C=15 byte per Cognome.

Supponiamo di avere a disposizione un DBMS che permetta strutture fisiche disordinate, ordinate e hash e che preveda la possibilità di definire indici secondari e un sistema operativo che utilizzi blocchi di dimensione B = 2000 byte con puntatori a blocchi di P = 4 byte.

Supponiamo che le operazioni principali siano le seguenti:

- O_1 – ricerca sul numero di matricola con frequenza f_1 = 2000 volte al minuto
- O_2 – ricerca sul cognome (o una sua sottostringa iniziale, abbastanza selettiva, in media una sottostringa identifica S = 10 tuple) con frequenza f_2 = 100 volte al minuto

Effettuare la progettazione fisica per identificare le strutture primarie e secondarie.

Considerazioni:

1. E’ comunque necessaria una struttura ad accesso diretto per matricola e cognome, visto che una scansione sequenziale sarebbe troppo costosa.
2. Non è possibile usare una struttura hash per Cognome, perché si vuole cercare per sottostringa (si può quindi considerare un indice primario)
3. Per la matricola si può utilizzare una struttura hash (ricerca diretta e struttura stabile nel tempo), oppure un indice (primario o secondario) in alternativa.

Abbiamo quindi 2 alternative da valutare:

- Struttura hash su Matricola - indice secondario su Cognome
- Indice primario su Cognome - secondario su Matricola

Ora possiamo calcolare i costi delle operazioni nei due casi, successivamente moltiplicare i costi per le frequenze per trovare l’alternativa migliore.

- $C_{1,A}$ – accesso diretto utilizzando l'hash (costo = 1)
- $C_{2,A}$ – richiede la visita dell'albero di Cognome + accessi diretti ai dati.

Fanout nodi intermedi dell'albero su cognome
~100 = (2000 bytes / (15 byte + 4 byte))

Profondità albero su cognome
$\log_{100}$ 10 000 000 = 3.5 (quindi **4 accessi** per raggiungere foglie)

+ (mediamente) S = 10 accessi alla struttura primaria per recuperare le tuple (ogni ricerca accede in media a10 tuple)

costo totale = 14

- $C_A = C_{1,A} \times f_1 + C_{2,A} \times f_2 =$

$= 1 \times 2000 + 14 \times 100 =$ **3400**

- $C_{1,B}$ – richiede la visita dell'albero di Matricola (secondario) + accesso

  Fanout nodi intermedi dell'albero su Matricola
  ~330 = (2000 bytes / (2 byte + 4 byte))

  Profondità albero su cognome
  $\log_{330} 10\,000\,000 = 2.78$ (quindi **3 accessi** per raggiungere foglie)

  + 1 accessi alla struttura primaria (chiave primaria)

  costo totale = 4

- $C_{2,B}$ – richiede la visita dell'albero di Cognome (primario)

  Fanout nodi intermedi dell'albero su Cognome ~= 100

  Le foglie contengono 2000 byte / 100 byte = 20
  Quindi ci sono 10 000 000 / 20 = 500 000 foglie

  Profondità albero su cognome (senza foglie)
  $\log_{100}$ 500 000 = 2.85 (quindi **3 accessi + 1** per raggiungere foglie)

  costo totale = 4

- $C_B = C_{1,B} \times f_1 + C_{2,B} \times f_2 =$

  $= 4 \times 2000 + 4 \times 100 =$ **8400**

Quindi viene scelta l’alternativa A (3400 accessi al minuto < 8400)

Cosa succederebbe se f1 e f2 fossero invertite (f1 = 100, f2 = 2000) ?

Quindi viene scelta l'alternativa A (3400 accessi al minuto < 8400)

Cosa succederebbe se f1 e f2 fossero invertite (f1 = 100, f2 = 2000) ?

- $C_A = C_{1,A} \times f_1 + C_{2,A} \times f_2 =$

  $= 1 \times 100 + 14 \times 2000 = \mathbf{28100}$

- $C_B = C_{1,B} \times f_1 + C_{2,B} \times f_2 =$

  $= 4 \times 100 + 4 \times 2000 = \mathbf{8400}$

In questo caso si sceglierebbe l'alternativa B.