Introduction to Data Management CSE 344

Lecture 19: Views

Announcements

- WQ6 is due tomorrow
- Homework 6 is due on Thursday
- AWS code for HW8 has been sent
  - Please keep this code carefully until you get instructions for setting up accounts for HW8.
  - Do not share the code with anyone, do not run any jobs now.
  - Each student gets $100 credit for HW8, sufficient for HW8.
  - **DO NOT** overuse/forget to stop server. **BE VERY CAREFUL**
- Today:
  - Views
    - Reading 8.1, Additional reading: 8.2, 8.5
  - Transactions (slides/notes in Lecture 20, already uploaded)

Views

- A view in SQL =
  - A table computed from other tables, s.t., whenever the base tables are updated, the view is updated too
- More generally:
  - A view is derived data that keeps track of changes in the original data
- Compare:
  - A function computes a value from other values, but does not keep track of changes to the inputs

Purchase(customer, product, store)
Product(pname, price)

StorePrice(store, price)

A Simple View

Create a view that returns for each store
the prices of products purchased at that store

```
CREATE VIEW  StorePrice AS
   SELECT DISTINCT x.store, y.price
   FROM  Purchase x, Product y
   WHERE x.product = y.pname
```

This is like a new table
StorePrice(store,price)

Purchase(customer, product, store)
Product(pname, price)

StorePrice(store, price)

We Use a View Like Any Table

- A "high end" store is a store that sell some products over 1000.
- For each customer, return all the high end stores that they visit.

```
SELECT DISTINCT u.cust, u.store
FROM Purchase u, StorePrice v
WHERE u.store = v.store
    AND v.price > 1000
```

Types of Views

- Virtual views
  - Used in databases
  - Computed only on-demand – slow at runtime
  - Always up to date

- Materialized views
  - Used in data warehouses
  - Pre-computed offline – fast at runtime
  - May have stale data (must recompute or update)
  - Indexes *are* materialized views

- A key component of physical tuning of databases is the selection of materialized views and indexes

Purchase(customer, product, store)
Product(pname, price)

StorePrice(store, price)

Query Modification

For each customer, find all the high end stores that they visit.

```
CREATE VIEW  StorePrice AS
  SELECT DISTINCT x.store, y.price
  FROM  Purchase x, Product y
  WHERE x.product = y.pname
```

```
SELECT DISTINCT u.cust, u.store
FROM Purchase u, StorePrice v
WHERE u.store = v.store
   AND v.price > 1000
```

Purchase(customer, product, store)
Product(pname, price)

StorePrice(store, price)

Query Modification

For each customer, find all the high end stores that they visit.

```
CREATE VIEW  StorePrice AS
   SELECT DISTINCT x.store, y.price
   FROM  Purchase x, Product y
   WHERE x.product = y.pname
```

```
SELECT DISTINCT u.cust, u.store
FROM Purchase u, StorePrice v
WHERE u.store = v.store
    AND v.price > 1000
```

Modified query:

```
SELECT DISTINCT u.cust, u.store
FROM Purchase u,
  (SELECT DISTINCT x.store, y.price
   FROM  Purchase x, Product y
   WHERE x.product = y.pname) v
WHERE u.store = v.store
    AND v.price > 1000
```

Purchase(customer, product, store)
Product(pname, price)

StorePrice(store, price)

Query Modification

For each customer, find all the high end stores that they visit.

```
SELECT DISTINCT u.customer, u.store
FROM Purchase u, Purchase x, Product y
WHERE u.store = x.store
    AND y.price > 1000
    AND x.product = y.pname
```

Notice that Purchase occurs twice. Why?

Modified query:

```
SELECT DISTINCT u.customer, u.store
FROM Purchase u,
  (SELECT DISTINCT x.store, y.price
   FROM  Purchase x, Product y
   WHERE x.product = y.pname) v
WHERE u.store = v.store
    AND v.price > 1000
```

Modified and unnested query:

Purchase(customer, product, store)
Product(pname, price)

StorePrice(store, price)

Further Virtual View Optimization

Retrieve all stores whose name contains ACME

```
CREATE VIEW  StorePrice AS
  SELECT DISTINCT x.store, y.price
  FROM  Purchase x, Product y
  WHERE x.product = y.pname
```

```
SELECT DISTINCT v.store
FROM StorePrice v
WHERE v.store like '%ACME%'
```

Purchase(customer, product, store)
Product(pname, price)

StorePrice(store, price)

Further Virtual View Optimization

Retrieve all stores whose name contains ACME

```
CREATE VIEW  StorePrice AS
  SELECT DISTINCT x.store, y.price
  FROM  Purchase x, Product y
  WHERE x.product = y.pname
```

```
SELECT DISTINCT v.store
FROM StorePrice v
WHERE v.store like '%ACME%'
```

Modified query:

```
SELECT DISTINCT v.store
FROM
 (SELECT DISTINCT x.store, y.price
  FROM  Purchase x, Product y
  WHERE x.product = y.pname) v
WHERE v.store like '%ACME%'
```

Purchase(customer, product, store)
Product(pname, price)

StorePrice(store, price)

Further Virtual View Optimization

Retrieve all stores whose name contains ACME

```
SELECT DISTINCT x.store
FROM Purchase x, Product y
WHERE x.product = y.pname
   AND  x.store like '%ACME%'
```

We can further optimize! How?

Modified query:

```
SELECT DISTINCT v.store
FROM
 (SELECT DISTINCT x.store, y.price
  FROM  Purchase x, Product y
  WHERE x.product = y.pname) v
WHERE v.store like '%ACME%'
```

Modified and unnested query:

Purchase(customer, product, store)
Product(pname, price)

StorePrice(store, price)

Further Virtual View Optimization

Retrieve all stores whose name contains ACME

SELECT DISTINCT x.store
FROM Purchase x, ~~Product y~~
WHERE ~~x.product = y.pname~~
~~AND~~ x.store like ‘%ACME%’

Assuming Product.pname is a key
and Purchase.product is a foreign key

Modified and unnested query:

Final Query

```
SELECT DISTINCT x.store
FROM Purchase x
WHERE x.store like '%ACME%'
```

Applications of Virtual Views

- Can you guess some?

Applications of Virtual Views

- Increased physical data independence. E.g.
  - Vertical data partitioning
  - Horizontal data partitioning
- Logical data independence. E.g.
  - Change schemas of base relations (i.e., stored tables)
- Security
  - View reveals only what the users are allowed to know

Vertical Partitioning

Resumes

<u>**SSN**</u>	**Name**	**Address**	**Resume**	**Picture**
234234	Mary	Huston	Clob1…	Blob1…
345345	Sue	Seattle	Clob2…	Blob2…
345343	Joan	Seattle	Clob3…	Blob3…
432432	Ann	Portland	Clob4…	Blob4…

This is how dbms decides to store data

T1

<u>**SSN**</u>	**Name**	**Address**
234234	Mary	Huston
345345	Sue	Seattle
. . .		

T2

<u>**SSN**</u>	**Resume**
234234	Clob1…
345345	Clob2…

T3

<u>**SSN**</u>	**Picture**
234234	Blob1…
345345	Blob2…

T2.SSN is a key *<u>and</u>* a foreign key to **T1**.SSN. Same for **T3**.SSN

T1(ssn,name,address)
T2(ssn,resume)
T3(ssn,picture)

Resumes(ssn,name,address,resume,picture)

Vertical Partitioning

```
CREATE VIEW Resumes AS
  SELECT T1.ssn, T1.name, T1.address,
         T2.resume, T3.picture
  FROM   T1,T2,T3
  WHERE  T1.ssn=T2.ssn AND T1.ssn=T3.ssn
```

However, the user still sees the same view!

T1(<u>ssn</u>,name,address)
T2(<u>ssn</u>,resume)
T3(<u>ssn</u>,picture)

Resumes(<u>ssn</u>,name,address,resume,picture)

Vertical Partitioning

```
CREATE VIEW  Resumes  AS
  SELECT  T1.ssn, T1.name, T1.address,
          T2.resume, T3.picture
  FROM    T1,T2,T3
  WHERE   T1.ssn=T2.ssn AND T1.ssn=T3.ssn
```

```
SELECT address
FROM   Resumes
WHERE  name = 'Sue'
```

However, the user still sees the same view!

T1(ssn,name,address)
T2(ssn,resume)
T3(ssn,picture)

Resumes(ssn,name,address,resume,picture)

Vertical Partitioning

```
CREATE VIEW  Resumes  AS
   SELECT  T1.ssn, T1.name, T1.address,
           T2.resume, T3.picture
   FROM    T1,T2,T3
   WHERE   T1.ssn=T2.ssn AND T1.ssn=T3.ssn
```

However, the user still sees the same view!

```
SELECT address
FROM   Resumes
WHERE  name = 'Sue'
```

Modified query:

```
SELECT T1.address
FROM T1, T2, T3
WHERE T1.name = 'Sue'
    AND T1.SSN=T2.SSN
    AND T1.SSN = T3.SSN
```

T1(ssn,name,address)
T2(ssn,resume)
T3(ssn,picture)

Resumes(ssn,name,address,resume,picture)

Vertical Partitioning

```
CREATE VIEW  Resumes  AS
   SELECT  T1.ssn, T1.name, T1.address,
           T2.resume, T3.picture
   FROM    T1,T2,T3
   WHERE   T1.ssn=T2.ssn AND T1.ssn=T3.ssn
```

```
SELECT address
FROM   Resumes
WHERE  name = 'Sue'
```

Modified query:

SELECT T1.address
FROM T1, ~~T2, T3~~
WHERE T1.name = 'Sue'
~~AND T1.SSN=T2.SSN~~
~~AND T1.SSN = T3.SSN~~

Final query:

```
SELECT T1.address
FROM T1
WHERE T1.name = 'Sue'
```

Vertical Partitioning Applications

Advantages

Disadvantages

Vertical Partitioning Applications

Advantages

- Speeds up queries that touch only a small fraction of columns
- Single column can be compressed effectively, reducing disk I/O

Disadvantages

Vertical Partitioning Applications

Advantages

- Speeds up queries that touch only a small fraction of columns
- Single column can be compressed effectively, reducing disk I/O

Disadvantages

- Updates are expensive!
- Need many joins to access many columns
- Repeated key columns add overhead

Hot trend today for data analytics: e.g., Vertica startup acquired by HP
They use a highly-tuned column-oriented data store AND engine

Horizontal Partitioning

This is how dbms decides to store data

Customers

SSN	Name	City
234234	Mary	Houston
345345	Sue	Seattle
345343	Joan	Seattle
234234	Ann	Portland
--	Frank	Calgary
--	Jean	Montreal

CustomersInHouston

SSN	Name	City
234234	Mary	Houston

CustomersInSeattle

SSN	Name	City
345345	Sue	Seattle
345343	Joan	Seattle

. . . . .

CustomersInHouston(ssn,name,city)
CustomersInSeattle(ssn,name,city)
. . . . .

Customers(ssn,name,city)

Horizontal Partitioning

```
CREATE VIEW  Customers  AS
    CustomersInHouston
        UNION ALL
    CustomersInSeattle
        UNION ALL
     . . .
```

However, the user still sees the same view!

CustomersInHouston(<u>ssn</u>,name,city)
CustomersInSeattle(<u>ssn</u>,name,city)
. . . . .

Customers(<u>ssn</u>,name,city)

Horizontal Partitioning

CustomersInHouston

SSN	Name	City
234234	Mary	Houston

CustomersInSeattle

SSN	Name	City
345345	Sue	Seattle
345343	Joan	Seattle

```
SELECT name
FROM   Customers
WHERE  city = 'Seattle'
```

Which tables are inspected by the system ?

CustomersInHouston(<u>ssn</u>,name,city)
CustomersInSeattle(<u>ssn</u>,name,city)
. . . . .

Customers(<u>ssn</u>,name,city)

Horizontal Partitioning

CustomersInHouston

SSN	Name	City
234234	Mary	Houston

CustomersInSeattle

SSN	Name	City
345345	Sue	Seattle
345343	Joan	Seattle

```
SELECT name
FROM    Customers
WHERE   city = 'Seattle'
```

Which tables are inspected by the system ?

All tables!
The systems doesn't know that CustomersInSeattle.city = 'Seattle'

CustomersInHouston(<u>ssn</u>,name,city)
CustomersInSeattle(<u>ssn</u>,name,city)
. . . . .

Customers(<u>ssn</u>,name,city)

Horizontal Partitioning

Better: remove CustomerInHuston.city etc

```
CREATE VIEW  Customers  AS
    (SELECT SSN, name, ‘Houston’ as city
     FROM CustomersInHouston)
       UNION ALL
    (SELECT SSN, name, ‘Seattle’ as city
     FROM CustomersInSeattle)
       UNION ALL
     . . .
```

CustomersInHouston(**<u>ssn</u>,name**)
CustomersInSeattle(**<u>ssn</u>,name**)
. . . . .

Customers(<u>ssn</u>,name,city)

Horizontal Partitioning

```
SELECT  name
FROM    Customers
WHERE   city = 'Seattle'
```

```
SELECT name
FROM   CustomersInSeattle
```

Horizontal Partitioning Applications

- Performance optimization
  - Especially for data warehousing
  - E.g. one partition per month
  - E.g. archived applications and active applications

- Distributed and parallel databases

- Data integration