Introduction to Database Systems
CSE 414

Lecture 19: Views
Announcements

• New web quiz and homework out now. Due Monday and Wednesday next week.

• Today: Views
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
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)
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.customer, 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
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.customer, u.store
FROM Purchase u, StorePrice v
WHERE u.store = v.store
  AND v.price > 1000
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.customer, u.store
FROM Purchase u, StorePrice v
WHERE u.store = v.store
AND v.price > 1000

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
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
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%’
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%'
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%’
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%'
```

Modified and unnested query:

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

Final Query

Assuming `Product.pname` is a key and `Purchase.product` is a foreign key
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
T2.SSN is a key and a foreign key to T1.SSN. Same for T3.SSN
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
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'
**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
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'

Final query:
SELECT T1.address
FROM T1
WHERE T1.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

27
Vertical Partitioning Applications

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

2. 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

**Customers**

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Houston</td>
</tr>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
</tr>
<tr>
<td>345343</td>
<td>Joan</td>
<td>Seattle</td>
</tr>
<tr>
<td>234234</td>
<td>Ann</td>
<td>Portland</td>
</tr>
<tr>
<td>--</td>
<td>Frank</td>
<td>Calgary</td>
</tr>
<tr>
<td>--</td>
<td>Jean</td>
<td>Montreal</td>
</tr>
</tbody>
</table>

**CustomersInHouston**

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Houston</td>
</tr>
</tbody>
</table>

**CustomersInSeattle**

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
</tr>
<tr>
<td>345343</td>
<td>Joan</td>
<td>Seattle</td>
</tr>
</tbody>
</table>

...
CREATE VIEW Customers AS
  CustomersInHouston
    UNION ALL
  CustomersInSeattle
    UNION ALL
  . . .
CustomersInHouston(ssn, name, city)
CustomersInSeattle(ssn, name, city)

......

Horizontal Partitioning

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

Which tables are inspected by the system?
Horizontal Partitioning

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

Which tables are inspected by the system?

- All tables!
- The system doesn’t know that `CustomersInSeattle.city` = ‘Seattle’
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
...
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