Views

Slides based on “Database Management Systems” 3rd ed, Ramakrishnan and Gehrke
Views

- So far, we have looked at SQL tables
  - Relations that are persistent
  - Physically stored in the DBMS

- It is also possible to have virtual relations, or views
  - Defined by an expression which is a SQL query
  - Do not exist physically in DBMS
    - Although it is possible to used materialized views

- Views can be queried directly
  - In some cases, it is also possible to modify views
Levels of Abstraction

- View 1: Describes files and indexes used
- View 2: Defines logical data structure
- View 3: Views define how users see data
- Conceptual Schema
- Physical Schema
- External Schema

Data
Creating a view

View

CREATE VIEW RegionalSales (category, sales, state, sid) 
AS SELECT P.category, S.sales, L.state, S.id 
FROM Products P, Sales S, Locations L 
WHERE P.pid=S.pid AND S.locid=L.locid

Defining Query
(also referred to as View Subquery)

Base Tables
Sales data is crucial to business planning

- Sales table can be huge: one row for each sales transaction
- Each sale is for a certain product (id pid) in a store in a certain location (id locid)
- The sales table needs to be compact: just sid, pid, locid info instead of any human-readable info
- This view adds basic human-readable info onto each sales entry:
  - Sales row: (1234, 34, 103, 34.59) for sid 1234, pid 34, locid 103 sales dollars 34.59: hard to interpret!
  - View row: ('pets', 34.59, 'MA', 1234) shows this sale is in MA in the pets category
- FYI: this is a "star schema" commonly used in data warehousing: big sales table in the middle, FKs to "dimension tables" with the details on products, stores, etc.
Querying views

Querying Views

```sql
SELECT R.category, R.state, SUM(R.sales)
FROM RegionalSales R GROUP BY R.category, R.state
```

- Views are queried just like regular tables
  - A view is just another relation (albeit a virtual one)
  - Queries can involve both views and base tables
  - Helps to think of views in terms of analogy with window on data
Views as subqueries

Equivalent Query (without views)

```sql
SELECT R.category, R.state, SUM(R.sales)
FROM (SELECT P.category, S.sales, L.state, S.id
      FROM Products P, Sales S, Locations L
      WHERE P.pid = S.pid AND S.locid = L.locid) R
GROUP BY R.category, R.state
```

SubQuery
Why are views useful?(1/3)

- **Usability**

  - Certain information must be retrieved from many tables
  
  - View abstraction can get all info in one (virtual) table
  
  - Queries are much easier to write on a single table
  
  - Subqueries that are often used can be included in queries without need for nesting
Why are views useful? (2/3)

- Compatibility
  - Shield users and application developer from changes
  - What if a schema changes? Define view that looks like the old schema
  - Users/applications access view, no changes needed in queries
  - “Obsolete” tables are preserved using views
    - Note: to be fully functional, some important views need to be updateable—we'll see how this is possible.
Why are views useful? (3/3)

- **Security**
  - Restrict user access to certain data only
    - Managers and employees are given different “views” of same data
  - Both column- and row-level access control possible

- Column-wise: students can only access **Name** and **Age** columns from a **Student** table

- Row-wise: access only transactions above $10,000 value
The syntax of the CREATE VIEW statement

CREATE [OR REPLACE] [{FORCE|NOFORCE}] VIEW view_name 
  [(column_alias_1[, column_alias_2]...)]
AS
  select_statement
  [WITH {READ ONLY|CHECK OPTION}
   [CONSTRAINT constraint_name]]

A statement that creates a view of vendors

CREATE VIEW vendors_phone_list AS
  SELECT vendor_name, vendor_contact_last_name, 
    vendor_contact_first_name, vendor_phone
  FROM vendors 
  WHERE vendor_id IN (SELECT vendor_id FROM invoices)

This create view example uses only standard SQL syntax.
A CREATE VIEW statement that uses a join

CREATE OR REPLACE VIEW vendor_invoices AS
  SELECT vendor_name, invoice_number, invoice_date, invoice_total
  FROM vendors
    JOIN invoices
      ON vendors.vendor_id = invoices.vendor_id

This create view example (also from Murach) uses only standard SQL syntax except for the “OR REPLACE”, as do the next four slides.
A CREATE VIEW statement that uses a subquery

```
CREATE OR REPLACE VIEW top5_invoice_totals AS
  SELECT vendor_id, invoice_total
  FROM (SELECT vendor_id, invoice_total FROM invoices
         ORDER BY invoice_total DESC)
  WHERE ROWNUM <= 5
```

ROWNUM is Oracle-specific

Mysql: "LIMIT 5" after WHERE, GROUP BY, HAVING, ORDER BY clauses, if any

Standard: not until 2008, way too late to help
A statement that names all the view columns in its CREATE VIEW clause

```
CREATE OR REPLACE VIEW invoices_outstanding
  (invoice_number, invoice_date, invoice_total,
   balance_due)
AS
  SELECT invoice_number, invoice_date, invoice_total,
    invoice_total - payment_total - credit_total
  FROM invoices
  WHERE invoice_total - payment_total - credit_total > 0
```
A statement that names just the calculated column in its SELECT clause

```
CREATE OR REPLACE VIEW invoices_outstanding AS
  SELECT invoice_number, invoice_date, invoice_total,
       invoice_total - payment_total - credit_total
       AS balance_due
  FROM invoices
  WHERE invoice_total - payment_total - credit_total > 0
```
A CREATE VIEW statement that summarizes invoices by vendor

```
CREATE OR REPLACE VIEW invoice_summary AS
  SELECT vendor_name,
         COUNT(*) AS invoice_count,
         SUM(invoice_total) AS invoice_total_sum
  FROM vendors
    JOIN invoices
        ON vendors.vendor_id = invoices.vendor_id
  GROUP BY vendor_name
```
Modifying views

- Is it possible to insert, update, delete tuples in a view?
  - Views are virtual …
  - … so modifications must be reflected in the base tables

- Why modifying views is a subtle issue?
  - Difficulty of translating view modifications in a unique way of updating base tables
    - Must be an unambiguous way to trace the base table tuple to update

- Views can be modified subject to restrictions
  - These are called **updatable views**
  - Still, many views are not updatable
Updatable Views: portable cases

- SQL-92 provides formal definition of updatable view:

1. View involves a **single** relation $R$. If $R$ is a view, it must also be updatable (relaxed in SQL-99)

2. Aggregate operations are **not** present in the view definition, and GROUP BY aggregation is also not allowed.

3. The DISTINCT keyword is **not** specified in SELECT clause

4. All columns in subquery are simple columns, **not** expressions

5. The WHERE clause **must not** contain a subquery involving $R$

6. All attributes in $R$ that are not in the SELECT clause of the view must **not** have both NOT NULL restriction and no default
Requirements for creating updatable views:
current Oracle requirements (not a complete list)

- The select list can’t include a DISTINCT clause. (as in SQL-92)
- The select list can’t include an aggregate function. (as in SQL-92)
- The SELECT statement can’t include a GROUP BY or HAVING clause. (as in SQL-92)
- The view can’t include the UNION operator. (this would mean two tables, so also disallowed in SQL-92)

Allowed in current Oracle beyond SQL-92:
- Multiple tables can be joined together, if FKs ensure that the query results have the same PK as one of the tables.
- Items in the SELECT clause can be expressions
A CREATE VIEW statement that creates an updatable view in Oracle

```
CREATE OR REPLACE VIEW balance_due_view AS
  SELECT vendor_name, invoice_number,
         invoice_total, payment_total, credit_total,
         invoice_total - payment_total - credit_total
       AS balance_due
  FROM vendors JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
  WHERE invoice_total - payment_total - credit_total > 0
```

An UPDATE statement that uses the view

```
UPDATE balance_due_view
SET credit_total = 300
WHERE invoice_number = '989319-497'
```

The response from the system

```
1 rows updated
```

- Note the expression in the select list, as well as the JOIN.
- Why OK? The join result has the same PK as invoices
- This create-view is standard SQL (except for OR REPLACE) — the surprising thing is that it is updatable (under Oracle).
An UPDATE statement that attempts to use the view to update a calculated column

```
UPDATE balance_due_view
SET balance_due = 0
WHERE invoice_number = '989319-497';
```

The response from the system

```
SQL Error: ORA-01733: virtual column not allowed here
```

So the view is updatable unless you try to update the expression result. Makes sense.
Insert and Delete on Updatable Views

- Insertion can be done through the view and cause an insert on the base table with view column values
  - Other attributes in $R$ set to NULL (unless a not null column)
- Deletion also possible
  - Delete tuple from base table
- Both insertion and deletion may cause unexpected results!
Issues with deletion

CREATE VIEW TopStudents (sid, sname) AS SELECT sid, name FROM Students S WHERE S.gpa > 3.0;

- Now let's delete students named Johnson
  
  DELETE FROM TopStudents WHERE Name LIKE '%Johnson%';

- Only tuples in the view are subject to deletion (or update)!
  - Outside tuples must be inaccessible (views used for security, too)
  - DBMS appends WHERE clause in view definition to statement

  DELETE FROM Students WHERE Name LIKE '%Johnson%' AND S.gpa > 3.0;
Issues with insertion

Base table student(sid, name, age, gpa)

CREATE VIEW TopStudents (sid, sname)
AS SELECT sid, name
FROM Students S
WHERE S.gpa > 3.0;

Now let’s insert a new student
INSERT INTO TopStudents VALUES ('10','FirstLastName');

GPA is set to NULL

Tuple falls outside view definition!
Not a mistake, but update will not be reflected in view!
  WITH CHECK OPTION clause disallows such an insertion
One solution is to include GPA in view definition
A statement that creates an updatable view

```sql
CREATE OR REPLACE VIEW ibm_invoices AS
  SELECT invoice_number, invoice_date, invoice_total
  FROM invoices
  WHERE vendor_id = 34;
```

The contents of the view

<table>
<thead>
<tr>
<th>INVOICE_NUMBER</th>
<th>INVOICE_DATE</th>
<th>INVOICE_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 QF58872</td>
<td>25-FEB-08</td>
<td>116.54</td>
</tr>
<tr>
<td>2 QS45443</td>
<td>14-MAR-08</td>
<td>1083.58</td>
</tr>
</tbody>
</table>
An INSERT statement that fails due to columns with null values

```sql
INSERT INTO ibm_invoices
  (invoice_number, invoice_date, invoice_total)
VALUES
  ('RA23988', '31-JUL-14', 417.34)
```

The response from the system

SQL Error: ORA-01400:
cannot insert NULL into ("AP"."INVOICES"."INVOICE_ID")

If you want to insert into a view, make sure the view has all the not-null columns, including of course the PK column(s).
An updatable view that has a WITH CHECK OPTION clause

(a standard clause of SQL-92)

```sql
CREATE OR REPLACE VIEW vendor_payment AS
SELECT vendor_name, invoice_number, invoice_date,
payment_date, invoice_total, credit_total,
payment_total
FROM vendors
JOIN invoices
    ON vendors.vendor_id = invoices.vendor_id
WHERE invoice_total - payment_total - credit_total >= 0
WITH CHECK OPTION
```

If you use WITH CHECK OPTION…

- An error will occur if you modify a row so it’s no longer included in the view, or insert a row that doesn’t show up in the view.
A statement that displays a row from the view

```
SELECT * FROM vendor_payment
WHERE invoice_number = 'P-0608'
```

The result set

<table>
<thead>
<tr>
<th>VENDOR_NAME</th>
<th>INVOICE_NUMBER</th>
<th>INVOICE_DATE</th>
<th>PAYMENT_DATE</th>
<th>INVOICE_TOTAL</th>
<th>CREDIT_TOTAL</th>
<th>PAYMENT_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malloy Lithographing Inc</td>
<td>P-0608</td>
<td>11-APR-08</td>
<td>(null)</td>
<td>20551.18</td>
<td>1200</td>
<td>0</td>
</tr>
</tbody>
</table>
An UPDATE statement that updates the view

```
UPDATE vendor_payment
SET payment_total = 400.00,
    payment_date = '01-AUG-14'
WHERE invoice_number = 'P-0608'
```

The response from the system

```
1 rows updated
```

The row after the update (still visible in the view)

<table>
<thead>
<tr>
<th>VENDOR_NAME</th>
<th>INVOICE_NUMBER</th>
<th>INVOICE_DATE</th>
<th>PAYMENT_DATE</th>
<th>INVOICE_TOTAL</th>
<th>CREDIT_TOTAL</th>
<th>PAYMENT_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malloy Lithographing Inc</td>
<td>P-0608</td>
<td>11-APR-08</td>
<td>01-AUG-08</td>
<td>20551.18</td>
<td>1200</td>
<td>400</td>
</tr>
</tbody>
</table>

This update doesn’t move the row out of the view ($400 is only a partial payment here)
A statement that tries to update the a row in the view to one no longer in the view

UPDATE vendor_payment
SET payment_total = 30000.00,
    payment_date = '01-AUG-14'
WHERE invoice_number = 'P-0608';

The response from the system

SQL Error: ORA-01402:
view WITH CHECK OPTION where-clause violation

This is a huge payment, overpaying the balance due, so the row would be sent out of the view (the view lists invoices with positive balance due)
Deleting views

DROP VIEW RegionalSales;

- View deleted from the schema
  - Note that, underlying data still intact
  - Contrast this with DROP TABLE!
View Materialization

- Materialized views can help speed up popular queries
  - Result has to be maintained when base tables change
  - They are stored just like base tables
  - But their contents are not “independent”; they must constantly reflect base tables
  - The database system detects changes in the base tables and uses these "events" to update the materialized view
  - These table-change events can also be made available to other applications, a process known as "Change Data Capture" or CDC for short.
- MVs are very important in "data warehousing"
- Surprising fact: MVs can be used to implement total participation constraints beyond "fat arrow" cases. See article.
Example 1

Create view **ActorSummary** that lists for every actor the actor identifier, actor name, number of movies starred in, and the year of debut (i.e., the year of the earliest movie(s) the actor starred in). The view will have four columns with headings: **ID**, **ActorName**, **MovieCount** and **DebutYear**
Example 1

Create view **ActorSummary** that lists for every actor the actor identifier, actor name, number of movies starred in, and the year of debut (i.e., the year of the earliest movie(s) the actor starred in). The view will have four columns with headings: ID, ActorName, MovieCount and DebutYear

```sql
CREATE VIEW ActorSummary (ID, ActorName, MovieCount, DebutYear) AS

SELECT A.actor_id, A.name, COUNT(M.movie_id), MIN(M.year)
FROM Actors A, StarsIn S, Movies M
WHERE A.actor_id = S.actor_id AND S.movie_id = M.movie_id
GROUP BY A.actor_id, A.name;
```
Example 2

Employee (eid:integer, ename:string, age:integer, salary:real)
Works (eid:integer, did:integer, pct_time:integer)
Department (did:integer, dname:string, budget:real, managerid:integer)

Create a view ManagerSummary that lists for every department the department name, manager ID and manager name, manager salary and the number of employees in that department. The view will have five columns with headings: DeptName, MgrID, MgrName, MgrSalary and EmpCount.

???
Example 2

Create a view **ManagerSummary** that lists for every department the department name, manager ID and manager name, manager salary and the number of employees in that department. The view will have five columns with headings:

**DeptName**, **MgrID**, **MgrName**, **MgrSalary** and **EmpCount**.

```
CREATE VIEW ManagerSummary(DeptName, MgrName, MgrID, MgrSalary, EmpCount) AS

SELECT D.dname, D.managerid, E.ename, E.salary, COUNT(W.eid)
FROM Department D, Employee E, Works W
WHERE D.managerid = E.eid AND D.did = W.did
GROUP BY D.did, D.dname, D.managerid, E.ename, E.salary;
```