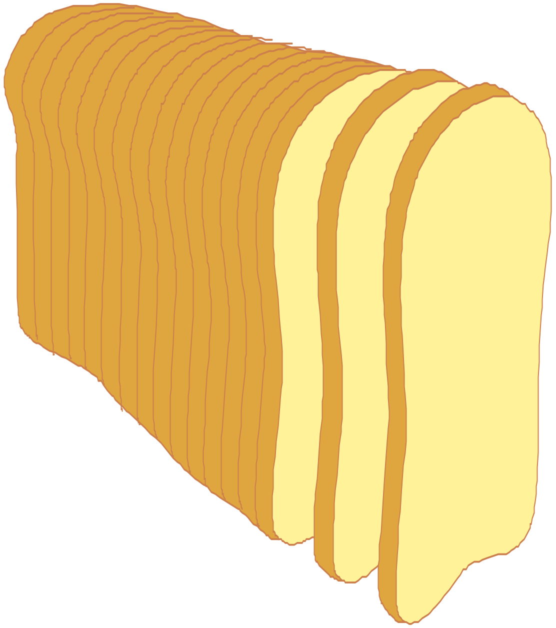
Partitioning
View Materialization

L14

Partitioning

- Breaking data into several physical units that can be handled separately
- Not a question of *whether* to do it in data warehouses but *how* to do it
- partitioning is key to effective implementation of a warehouse



Partitioning

- Partitioning is a feature designed to improve the performance of queries made against a very large table.
- It works by having more than one subset of data for the same table.
- All the rows are not directly stored in the table, but they are distributed in different partitions of this table.
- When you query the table looking for data in a single partition, or just a few, then due to the presence of these different subsets, you should receive a quicker response from the server.

Why Partition?

- Flexibility in managing data
- Smaller physical units allow
  - easy restructuring
  - free indexing
  - sequential scans if needed
  - easy reorganization
  - easy recovery
  - easy monitoring

Criterion for Partitioning

- Typically partitioned by
  - date
  - line of business
  - geography
  - organizational unit
  - any combination of above

Where to Partition?

- Application level or DBMS level
- Makes sense to partition at application level
  - Allows different definition for each year
    - Important since warehouse spans many years and as business evolves definition changes
  - Allows data to be moved between processing complexes easily

Vertical Partitioning

Acct. No	Name	Balance	Date Opened	Interest Rate	Address

Frequently accessed

Rarely accessed

Acct. No	Balance

Acct. No	Name	Date Opened	Interest Rate	Address

Smaller table and so less I/O

Partitioning

- **Improves performance**
- **Ease of management**
- **Fact tables are partitioned wrt dimensions**
- **Dimension tables may also be partitioned**

Partitioning

- **Horizontal Partitioning**
  - **Range**
  - **Hash**
  - **List**
  - **Composite**
- **Vertical Partitioning**
  - **Normalization**
  - **Row Splitting**

Partitioning: Thumb Rules

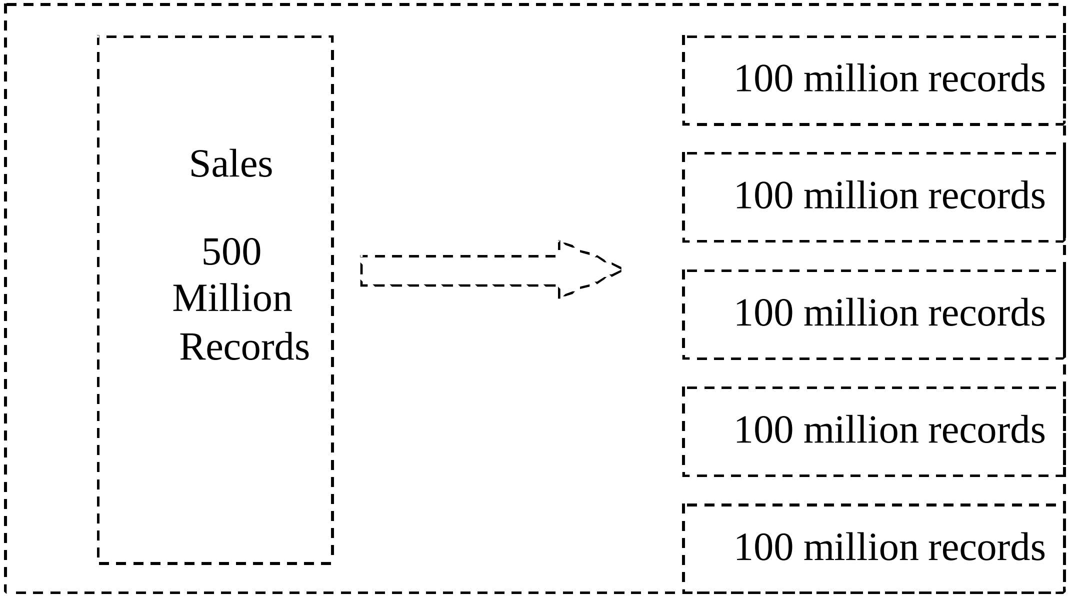
- **At no point of time there should be more than 500 tables in the data warehouse**
- **Do not partition on a dimension grouping that is likely to change within the lifecycle of the data warehouse**
    - **For example product, & location**

Partitioning: Advantages

- **Faster Access to Data**
- **Easy support for Data Purging**
- **Parallel DML**
- **Backup of Large Tables**

Partitioning Fact Tables

- **Sales Fact Table**



Partitioning Fact Tables

- **Wrt Time Dimension?**
  - **Equal Size**
  - **Unequal Size**
- **Wrt Product Dimension?**
- **Wrt Location Dimension?**
- **Wrt Size?**

Partitioning Fact Tables wrt TIME

- **Range Partitioning: Most commonly used**
- **Non-overlapping Partitions**
- **Example: Partition By Month**

Partitioning Fact Tables wrt TIME

```
CREATE TABLE sales_range
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
 sales_date    DATE)
PARTITION BY RANGE(sales_date)
 (
PARTITION sales_jan2000 VALUES LESS
THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
 PARTITION sales_feb2000 VALUES LESS
THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
PARTITION sales_mar2000 VALUES LES
THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
PARTITION sales_apr2000 VALUES LESS
THAN(TO_DATE('05/01/2000','DD/MM/YYYY')),
  );
```

Partitioning Fact Tables wrt TIME

```
create table sales (
lid number(2),
pid number(2),
sale_date date,
units number(3),
amount number(5)),
partition by range (sale_date)
(partition sale_1999 values less than
(to_date('01-JAN-2000','DD-MON-YYYY')),
partition sale_2000 values less than
(to_date('01-JAN-2001', 'DD-MON-YYYY')));
```

Partitioning Fact Tables wrt TIME

Seeing the partitions

select partition_name from user_tab_partitions where table_name='SALES';

Seeing the records in a partition

*select * from sales partition (sale_1999);*

*select * from sales partition (sale_2000);*

Now check what you get when you run the following command:

*select * from sales;*

What does it give you? And what you can infer from it?

Partitioning

- Table partitioning can make very large tables and indexes easier to manage, and improve the performance of appropriately filtered queries.
- strategies for partitioning tables
- partition functions and partition schemes
- Partitioning a large table divides the table and its indexes into smaller partitions, so that maintenance operations can be applied on a partition-by-partition basis, rather than on the entire table. In addition, the SQL Server optimizer can direct properly filtered queries to appropriate partitions rather than the entire table.

SQL Server 2008

- SQL Server 2008 provides a number of enhancements to partitioned tables and indexes:
- There are new wizards, the Create Partition Wizard and the Manage Partition Wizard, in SQL Server Management Studio, for creating and managing partitions and sliding windows.
- SQL Server 2008 supports switching partitions when partition-aligned indexed views are defined:
  - Management of indexed views on partitioned tables is much easier.
  - Aggregations in indexed views are preserved on existing partitions, and you only need to build new aggregates on the new partition before switching it into or out of the partitioned table.

Planning for Table Partitioning

- In order to successfully partition a large table, you must make a number of decisions. In particular, you need to:
- Plan the partitioning:
  - Decide which table or tables can benefit from the increased manageability and availability of partitioning.
  - Decide the column or column combination upon which to base the partition.
- Specify the partition boundaries in a partition function.
- Plan on how to store the partitions in filegroups using a partition scheme.

Choosing a Table to Partition

- There is no firm rule or formula that would determine when a table is large enough to be partitioned, or whether even a very large table would benefit from partitioning.
- Sometimes large tables may not require partitioning, if for example the tables are not accessed much and do not require index maintenance

- In general, any large table has maintenance costs that exceed requirements, or that is not performing as expected due to its size, might be a candidate for table partitioning. Some conditions that might indicate a table could benefit from partitioning are:
- Index maintenance on the table is costly or time-consuming and could benefit from reindexing partitions of the table rather than the whole table at once.
- Data must be aged out of the table periodically, and the delete process is currently too slow or blocks users trying to query the table.
- New data is loaded periodically into the table, the load process is too slow or interferes with queries on the table, and the table data lends itself to a partition column based on ascending date or time.

SQL Server

- SQL Server only supports one type of partitioning, which is Range Partitions. More specifically I should say 'Horizontal Range Partitions'. This the partitioning strategy in which data is partitioned based on the range that the value of a particular field falls in.

Partitioning

- when we partition a table, we define on which portion of a table a particular row will be stored.
- Now you must be wondering what would be the criterion that a specific row would be saved in a particular partition. There are actually rules defined for ranges of data to fill a particular partition.
- These ranges are based on a particular column; this is called the Partition Key. It should be noted that these ranges are non-overlapping. To achieve this, a *Partition Function* and a *Partition Scheme* is defined.

PARTITION FUNCTIONS

- The Partition Function is the function that defines the number of partitions.
- This is the first step in the implementation of partitioning for your database object.
- One partition function can be used to partition a number of objects.
- The type of partition is also specified in the partition function, which currently can only be 'RANGE'.

PARTITION FUNCTIONS

- Based on the fact about boundary values for partitions that which partition they should belong to, we can divide partition function into two types:
- **Left:** The first value is the maximum value of the first partition.
- **Right:** The first value is the minimum value of the second partition.
- The syntax for the creation of a partition function is as follows:

```
CREATE PARTITION FUNCTION partition_function_name
( input_parameter_type )
AS RANGE [ LEFT | RIGHT ]
FOR VALUES ( [ boundary_value [ ,...n ] ] ) [ ; ]
```

PARTITION SCHEME

- This is the physical storage scheme that will be followed by the partition. To define scheme, different file groups are specified, which would be occupied by each partition. It must be remembered that all partitions may also be defined with only one file group.
- After the definition of a partition function, a partition scheme is defined.
- The partition scheme just like specifying an alignment for data i.e. it specifies the specific file groups used during partitioning an object.

- The syntax for creating partition schema is as follows:

```
CREATE PARTITION SCHEME
partition_scheme_name
AS PARTITION partition_function_name
[ ALL ] TO ( { file_group_name | [ PRIMARY ] } [
,...n ] )
[ ; ]
```

PARTITIONED TABLE

- After creation of a partition scheme, a table may be defined to follow that scheme. In this case the table is called PARTITIONED. A partitioned table may have a partitioned index. Partition aligned index views may also be created for this table.
- For partitioning your existing table just drop the clustered index on your table and recreate it on the required partition scheme.

- CREATE TABLE
  [database_name . [schema_name] . | schema_name .] table_name
  ({ <column_definition> | <computed_column_definition> }
  [<table_constraint>] [,...n])
  [ON { partition_scheme_name (partition_column_name) | filegroup
  | "default" }]
  [{ TEXTIMAGE_ON { filegroup | "default" }] [;]

View Materialization

Dr. Yashvardhan Sharma
Department of Computer Science & Information Systems
BITS, Pilani

Topics

- View Materialization
- Which Views to Materialize?
- How to exploit Materialized Views to answer queries?
- View Maintenance

View Modification (Evaluate On Demand)

View

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

Query

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

Modified Query

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

View Materialization (Precomputation)

- Suppose we precompute RegionalSales and store it with a clustered B+ tree index on [category,state,sales].
- Then, previous query can be answered by an index-only scan.

SELECT R.state, SUM(R.sales) FROM RegionalSales R WHERE R.category="Laptop" GROUP BY R.state	SELECT R.state, SUM(R.sales) FROM RegionalSales R WHERE R. state="Wisconsin" GROUP BY R.category

Index on precomputed view is great!

Index is less useful (must scan entire leaf level).

Materialized Views

- A view whose tuples are stored in the database is said to be materialized
  - Provides fast access, like a (very high-level) cache.
  - Need to maintain the view as the underlying tables change.
  - Ideally, we want incremental view maintenance algorithms.
- Close relationship to Data Warehousing, OLAP,

What is a Materialized View?

- A database object that stores the results of a query
  - Marries the query rewrite features found in Oracle Discoverer with the data refresh capabilities of snapshots
- Features/Capabilities
  - Can be partitioned and indexed
  - Can be queried directly
  - Can have DML applied against it
  - Several refresh options are available
  - Best in read-intensive environments

Advantages and Disadvantages

- Advantages
  - Useful for summarizing, pre-computing, replicating and distributing data
  - Faster access for expensive and complex joins
  - Transparent to end-users
    - MVs can be added/dropped without invalidating coded SQL
- Disadvantages
  - Performance costs of maintaining the views
  - Storage costs of maintaining the views

Database Parameter Settings

- init.ora parameter
  - **`COMPATIBLE=8.1.0`** (or above)
- System or session settings
  - `query_rewrite_enabled={true|false}`
  - `query_rewrite_integrity={enforced|trusted|stale_tolerated}`
- Can be set for a session using
  - `alter session set query_rewrite_enabled=true;`
  - `alter session set query_rewrite_integrity=enforced;`
- Privileges which must be granted to users directly
  - **`QUERY_REWRITE`** - for MV using objects in own schema
  - **`GLOBAL_QUERY_REWRITE`** - for objects in other schemas

Query Rewrite Details

- query_rewrite_integrity Settings:
  - **`enforced`** – rewrites based on Oracle enforced constraints
    - Primary key, foreign keys
  - **`trusted`** – rewrites based on Oracle enforced constraints and known, but not enforced, data relationships
    - Primary key, foreign keys
    - Data dictionary information
    - Dimensions
  - **`stale_tolerated`** – queries rewritten even if Oracle knows the mv's data is out-of-sync with the detail data
    - Data dictionary information

Query Rewrite Details (cont'd)

- Query Rewrite Methods
  - Full Exact Text Match
    - Friendlier/more flexible version of text matching
  - Partial Text Match
    - Compares text starting at FROM clause
    - SELECT clause must be satisfied for rewrite to occur
  - Data Sufficiency
    - All required data must be present in the MV or retrievable through a join-back operation
  - Join Compatibility
    - All joined columns are present in the MV

Query Rewrite Details (cont'd)

- Grouping Compatibility
  - Allows for matches in groupings at higher levels than those defined MV query
  - Required if both query and MV contain a GROUP BY clause
- Aggregate Compatibility
  - Allows for interesting rewrites of aggregations
    - If SUM(x) and COUNT(x) are in MV, the MV may be used if the query specifies AVG(x)

Syntax For Materialized Views

```
CREATE MATERIALIZED VIEW <name>
 TABLESPACE <tbs name>  {<storage parameters>}
 <build option>
 REFRESH <refresh option> <refresh mode>
 [ENABLE|DISABLE] QUERY REWRITE
AS
 SELECT <select clause>;
```

- The **<build option>** determines when MV is built
  - **BUILD IMMEDIATE**: view is built at creation time
  - **BUILD DEFFERED**: view is built at a later time
  - **ON PREBUILT TABLE**: use an existing table as view source
    - Must set **QUERY_REWRITE_INTEGRITY** to **TRUSTED**

Materialized View Refresh Options

- Refresh Options
  - **COMPLETE** – totally refreshes the view
    - Can be done at any time; can be time consuming
  - **FAST** – incrementally applies data changes
    - A materialized view log is required on each detail table
    - Data changes are recorded in MV logs or direct loader logs
    - Many other requirements must be met for fast refreshes
  - **FORCE –** does a FAST refresh in favor of a COMPLETE
    - The default refresh option

Materialized View Refresh Modes

- Refresh Modes
  - **ON COMMIT** – refreshes occur whenever a commit is performed on one of the view's underlying detail table(s)
    - Available only with single table aggregate or join based views
    - Keeps view data transactionally accurate
    - Need to check alert log for view creation errors
  - **ON DEMAND** – refreshes are initiated manually using one of the procedures in the DBMS_MVIEW package
    - Can be used with all types of materialized views
    - Manual Refresh Procedures
      - `DBMS_MVIEW.REFRESH(<mv_name>, <refresh_option>)`
      - `DBMS_MVIEW.REFRESH_ALL_MVIEWS()`
  - **START WITH [NEXT] <date>** - refreshes start at a specified date/time and continue at regular intervals

Materialized View Example

```
CREATE MATERIALIZED VIEW items_summary_mv
 ON PREBUILT TABLE
 REFRESH FORCE  AS
 SELECT  a.PRD_ID, a.SITE_ID, a.TYPE_CODE, a.CATEG_ID,
         sum(a.GMS)       GMS,
         sum(a.NET_REV)   NET_REV,
         sum(a.BOLD_FEE)  BOLD_FEE,
         sum(a.BIN_PRICE) BIN_PRICE,
         sum(a.GLRY_FEE)  GLRY_FEE,
         sum(a.QTY_SOLD)  QTY_SOLD,
         count(a.ITEM_ID) UNITS
FROM  items a
GROUP BY  a.PRD_ID, a.SITE_ID, a.TYPE_CODE, a.CATEG_ID;

ANALYZE TABLE item_summary_mv COMPUTE STATISTICS;
```

Materialized View Example (cont'd)

```
-- Query to test impact of materialized view

select categ_id, site_id,
       sum(net_rev),
       sum(bold_fee),
       count(item_id)
  from items
 where prd_id in ('2000M05','2000M06','2001M07','2001M08')
   and site_id in (0,1)
   and categ_id in (2,4,6,8,1,22)
 group by categ_id, site_id

save mv_example.sql
```

Materialized View Example (cont'd)

```
SQL> ALTER SESSION SET QUERY_REWRITE_INTEGRITY=TRUSTED;
SQL> ALTER SESSION SET QUERY_REWRITE_ENABLED=FALSE;
SQL> @mv_example.sql

CATEG_ID SITE_ID SUM(NET_REV) SUM(BOLD_FEE) COUNT(ITEM_ID)
-------- ------- ------------ ------------- --------------
       1       0        -2.35             0              1
      22       0    -42120.87          -306          28085

Elapsed: 01:32:17.93

Execution Plan
----------------------------------------------------------
 0    SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=360829 Card=6 Bytes=120)
 1  0   SORT (GROUP BY) (Cost=360829 Card=6 Bytes=120)
 2  1     PARTITION RANGE (INLIST
 3  2       TABLE ACCESS (FULL) OF 'ITEMS' (Cost=360077
              Card=375154 Bytes=7503080)
```

Materialized View Example (cont’d)

```
SQL> ALTER SESSION SET QUERY_REWRITE_ENABLED=TRUE;
SQL> @mv_example.sql

CATEG_ID SITE_ID SUM(NET_REV) SUM(BOLD_FEE) COUNT(ITEM_ID)
-------- ------- ------------ ------------- --------------
       1       0        -2.35             0              1
      22       0    -42120.87          -306          28085
Elapsed: 00:01:40.47

Execution Plan
-------------------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=3749 Card=12 Bytes=276)
   1    0   SORT (GROUP BY) (Cost=3749 Card=12 Bytes=276)
   2    1     PARTITION RANGE (INLIST)
   3    2       TABLE ACCESS (FULL) OF 'ITEMS_SUMMARY_MV'
                  (Cost=3723 Card=7331 Bytes=168613)
```

Example of FAST REFRESH MV

```
CREATE MATERIALIZED VIEW LOG ON ITEMS
  TABLESPACE MV_LOGS   STORAGE(INITIAL 10M NEXT 10M) WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON CUSTOMERS
  TABLESPACE MV_LOGS   STORAGE(INITIAL 1M NEXT 1M) WITH ROWID;

CREATE MATERIALIZED VIEW cust_activity
 BUILD IMMEDIATE
 REFRESH FAST ON COMMIT
AS
 SELECT u.ROWID cust_rowid, l.ROWID item_rowid,
        u.cust_id, u.custname, u.email,
        l.categ_id, l.site_id, sum(gms), sum(net_rev_fee)
   FROM customers u, items l
  WHERE u.cust_id = l.seller_id
  GROUP BY u.cust_id, u.custname, u.email, l.categ_id, l.site_id;
```

Getting Information About an MV

Getting information about the key columns of a materialized view:

```
SELECT POSITION_IN_SELECT  POSITION,
       CONTAINER_COLUMN    COLUMN,
       DETAILOBJ_OWNER     OWNER,
       DETAILOBJ_NAME      SOURCE,
       DETAILOBJ_ALIAS     ALIAS,
       DETAILOBJ_TYPE      TYPE,
       DETAILOBJ_COLUMN    SRC_COLUMN
  FROM USER_MVIEW_KEYS
 WHERE MVIEW_NAME='ITEMS_SUMMARY_MV';
```

POS	**COLUMN**	OWNER	**SOURCE**	ALIAS	**TYPE**	SRC_COLUMN
1	PRD_ID	TAZ	ITEMS	A	**TABLE**	PRD_ID
2	SITE_ID	TAZ	ITEMS	A	**TABLE**	SITE_ID
3	TYPE_CODE	TAZ	ITEMS	A	**TABLE**	TYPE_CODE
4	CATEG_ID	TAZ	ITEMS	A	**TABLE**	CATEG_ID

Getting Information About an MV

Getting information about the aggregate columns of a materialized view:

```
SELECT POSITION_IN_SELECT  POSITION,
       CONTAINER_COLUMN    COLUMN,
       AGG_FUNCTION
  FROM USER_MVIEW_AGGREGATES
 WHERE MVIEW_NAME='ITEMS_SUMMARY_MV';
```

```
 POSITION     COLUMN                   AGG_FUNCTION
 --------     -----------------        ------------
        6     GMS                      SUM
        7     NET_REV                  SUM
        :        :                      :
       11     QTY_SOLD                 SUM
       12     UNITS                    COUNT
```

Issues in View Materialization

- What views should we materialize, and what indexes should we build on the precomputed results?
- Given a query and a set of materialized views, can we use the materialized views to answer the query?
- How frequently should we refresh materialized views to make them consistent with the underlying tables? (And how can we do this incrementally?)

View Materialization: Example

```
SELECT P.Category, SUM(S.sales)
FROM Product P, Sales S
WHERE P.pid=S.pid
GROUP BY P.Category
```

```
SELECT L.State, SUM(S.sales)
FROM Location L, Sales S
WHERE L.locid=S.locid
GROUP BY L.State
```

Both queries require us to join the Sales table with another table & aggregate the result

How can we use materialization to speed up these queries?

View Materialization: Example

- Pre-compute the two joins involved
  (product & sales & Location & sales)
- Pre-compute each query in its entirety
- OR let us define the following view:

```
CREATE VIEW TOTALSALES (pid, lid, total)
AS Select S.pid, S.locid, SUM(S.sales)
FROM Sales S
GROUP BY S.pid, S.locid
```

View Materialization: Example

- The View TOTALSALES can be materialized & used instead os Sales in our two example queries

```
SELECT P.Category, SUM(T.Total)
FROM Product P, TOTALSALES T
WHERE P.pid=T.pid
GROUP BY P.Category
```

```
SELECT L.State, SUM(T.Total)
FROM Location L, TOTALSALES T
WHERE L.locid=T.locid
GROUP BY L.State
```

View Maintenance

- A materialized view is said to be refreshed when it is made consistent with changes to its underlying tables
- Often referred to as VIEW MAINTENANCE
- Two issues:
  - HOW do we refresh a view when an underlying table is refreshed? Can we do it incrementally?
  - WHEN should we refresh a view in response to a change in the underlying table?

View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update
- A better option is to use **incremental view maintenance**
  - **Changes to database relations are used to compute changes to materialized view, which is then updated**
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Supported directly by the database

View Maintenance

- **Two steps:**
  - **Propagate:** Compute changes to view when data changes.
  - **Refresh:** Apply changes to the materialized view table.
- **Maintenance policy:** Controls when we do refresh.
  - **Immediate:** As part of the transaction that modifies the underlying data tables. (+ Materialized view is always consistent; - updates are slowed)
  - **Deferred:** Some time later, in a separate transaction. (- View becomes inconsistent; + can scale to maintain many views without slowing updates)