Partition Pruning

- The QP needs to be smart about partitions/MDC cells
- From Oracle docs, the idea: “Do not scan partitions where there can be no matching values”.
- Example: partitions of table t1 based on region_code:
  - PARTITION BY RANGE(region_code)
    - (PARTITION p0 VALUES LESS THAN (64),
      PARTITION p1 VALUES LESS THAN (128),
      PARTITION p2 VALUES LESS THAN (192),
      PARTITION p3 VALUES LESS THAN MAXVALUE);
- Query:
  - SELECT fname, lname, region_code, dob FROM t1
    WHERE region_code > 125 AND region_code < 130;
- QP should prune partitions p0 (region_code too low) and p3 (too high).
- But the capability is somewhat fragile in practice.

Parallelism is essential to huge DWs

<table>
<thead>
<tr>
<th>Shared Memory (least scalable)</th>
<th>Shared Disk (medium scalable)</th>
<th>Shared Nothing (most scalable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft SQL Server</td>
<td>Oracle RAC</td>
<td>Teradata</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Sybase IQ</td>
<td>IBM DB2</td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
<td>Netezza</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EnterpriseDB (Postgres)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Greenplum</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vertica</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MySQL Cluster</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAP HANA</td>
</tr>
</tbody>
</table>

Views and Materialized Views

Views: review of pp. 86-91

- **View** - rows are not explicitly stored, but computed as needed from view definition
- **Base table** - explicitly stored
CREATE VIEW

Given tables for these relations:

Students (ID, name, major)
Enrolled (ID, CourseID, grade)

Can create view:

CREATE VIEW B_Students (name, ID, CourseID) AS
SELECT S.name, S.ID, E.CourseID
FROM Students S, Enrolled E
WHERE S.ID = E.ID AND E.grade = 'B';

Now can use B_Students just as if it were a table, for queries

Could be used to shield D_students from view

Can grant select on view, but not on enrolled

---

Updatable Views

SQL-92: Must be defined on a single table using only selection and projection and not using DISTINCT.

SQL-1999: May involve multiple tables in SQL:1999 if each view field is from exactly one underlying base table and that table's PK is included in view; not restricted to selection and project, but cannot insert into views that use union, intersection, or set difference.

So B_Students is updatable by SQL99, and by Oracle 10.

---

Materialized Views

- What is a Materialized View?
  - Advantages and Disadvantages
  - Creating Materialized Views
  - Syntax, Refresh Modes/Options, Build Methods
  - Examples
  - Dimensions
    - What are they?
    - Examples

- 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

---

What is a Materialized View?

- A database object that stores the results of a query

- Features/Capabilities
  - Can be partitioned and indexed
  - Can be queried directly
  - Can have DML applied against it
  - Several refresh options are available (in Oracle)
  - Best in read-intensive environments

---

Similar to Indexes

- Designed to increase query Execution Performance.
- Transparent to SQL Applications allowing DBA's to create and drop Materialized Views without affecting the validity of Applications.
- Consume Storage Space.
- Can be Partitioned.
- Not covered by SQL standards
- But can be queried like tables
MV Support in DBs: from Wikipedia

- Materialized views were implemented first by the Oracle, and Oracle has the most features.
- In IBM DB2, they are called "materialized query tables";
- Microsoft SQL Server has a similar feature called "indexed views";
- MySQL doesn't support materialized views natively, but workarounds can be implemented by using triggers or stored procedures or by using the open-source application Flexviews.


<table>
<thead>
<tr>
<th>Table</th>
<th>View</th>
<th>Materialized View</th>
</tr>
</thead>
<tbody>
<tr>
<td>select * from T;</td>
<td>create view v as select * from t;</td>
<td>create materialized view mv as select * from t;</td>
</tr>
<tr>
<td>KEY</td>
<td>VAL</td>
<td>KEY</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Update to T is not propagated immediately to simple MV

<table>
<thead>
<tr>
<th>Table</th>
<th>View</th>
<th>Materialized View</th>
</tr>
</thead>
<tbody>
<tr>
<td>update t set val = upper(val);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>select * from T;</td>
<td>select * from V;</td>
<td>select * from MV;</td>
</tr>
<tr>
<td>KEY</td>
<td>VAL</td>
<td>KEY</td>
</tr>
<tr>
<td>1 A</td>
<td>1 A</td>
<td>1 a</td>
</tr>
<tr>
<td>2 B</td>
<td>2 B</td>
<td>2 b</td>
</tr>
<tr>
<td>3 C</td>
<td>3 C</td>
<td>3 c</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

MV “refresh“ command

<table>
<thead>
<tr>
<th>Table</th>
<th>View</th>
<th>Materialized View</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute dbms_mview.refresh('MV');</td>
<td></td>
<td></td>
</tr>
<tr>
<td>select * from T;</td>
<td>select * from V;</td>
<td>select * from MV;</td>
</tr>
<tr>
<td>KEY</td>
<td>VAL</td>
<td>KEY</td>
</tr>
<tr>
<td>1 A</td>
<td>1 A</td>
<td>1 A</td>
</tr>
<tr>
<td>2 B</td>
<td>2 B</td>
<td>2 B</td>
</tr>
<tr>
<td>3 C</td>
<td>3 C</td>
<td>3 C</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Materialized View Logs for fast refresh

- There is a way to refresh only the changed rows in a materialized view’s base table, called fast refreshing.
- For this, need a materialized view log (MLOG$_T$ here) on the base table t:
  
  ```
  create materialized view log on t;
  
  UPDATE t set val = upper(val) where KEY = 1;
  INSERT into t (KEY, val) values (5, 'e');
  
  select key, dmltype$$ from MLOG$_T$;
  
  KEY DMLTYPE$$
  1 U
  2 B
  3 C
  4
  ```

REFRESH FAST

- create materialized view mv REFRESH FAST as select * from t;
- select key, rowid from mv;
  
  KEY ROWID
  1 a
  2 b
  3 c
  4

- see same ROWIDs as above: nothing needed to be changed.
Now let’s update a row in the base table.

```sql
update t set val = 'XX' where key = 3;
commit;
execute dbms_mview.refresh (list => 'MV', method => 'F');
```

```sql
select key, val, rowid from mv;
```

<table>
<thead>
<tr>
<th>KEY</th>
<th>VAL</th>
<th>ROWID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>AAAM6+AAEAAAAAABA</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>AAAM6+AAEAAAAAABAB</td>
</tr>
<tr>
<td>3</td>
<td>XX</td>
<td>AAAM6+AAEAAAAAABAC</td>
</tr>
</tbody>
</table>

See update, same old ROWID

<table>
<thead>
<tr>
<th>T_KEY</th>
<th>ROW_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

So the MV row was updated based on the log entry

Adding Your Own Indexes

```sql
create materialized view mv refresh fast on commit as
select t.key, COUNT(*) ROW_COUNT from t group by t.key;
```

```sql
create index MY_INDEX on mv (T_KEY);
```

```sql
select index_name, i.uniqueness, ic.column_name from user_indexes i
inner join user_ind_columns ic on ( i.index_name )
where i.table_name = 'MV';
```

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>UNIQUENESS</th>
<th>COLUMN_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_SNAP$_MV</td>
<td>UNIQUE</td>
<td>SYS_NC00003$</td>
</tr>
<tr>
<td>MY_INDEX</td>
<td>NONUNIQUE</td>
<td>T_KEY</td>
</tr>
</tbody>
</table>

Prove that MY_INDEX is in use using SQL*Plus’s Autotrace feature

```sql
set autotrace on explain linesize 95
select * from mv where t_key = 2;
```

```sql
T_KEY ROW_COUNT
------- -----------
  2   2
```

MV on Join query

```sql
create materialized view log on t with rowid, sequence ;
create materialized view log on t2 with rowid, sequence
create materialized view mv refresh fast on commit enable query rewrite
as select t.key t_key, t.val t_val, t2.key t2_key, t2.amt t2_amt, t2.rowid t2_row_id from t, t2
where t.key = t2.key ;
create index mv_i1 on mv (t_row_id) ;
create index mv_i2 on mv (t2_row_id) ;
```

MV with aggregation

```sql
create materialized view log on t2 with rowid, sequence (t_key, amt) including new values ;
create materialized view mv refresh fast on commit enable query rewrite
as select t.key, sum(amt) as amt, sum(count(*)) as row_count ,
count(amt) as amt_count from t2 group by t_key ;
create index mv_i1 on mv (t_key) ;
```

MV with join and aggregation from Oracle DW docs

```sql
CREATE MATERIALIZED VIEW LOG ON products WITH SEQUENCE, ROWID (prod_id,prod_name,...) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON sales WITH SEQUENCE, ROWID (prod_id,cust_id,time_id,channel_id,promo_id,quantity_sold, amount_sold) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW product_sales_mv BUILD IMMEDIATE REFRESH FAST ENABLE QUERY REWRITE AS SELECT p.prod_name,SUM(s.amount_sold) AS dollar_sales, COUNT(*) AS cnt,COUNT(s.amount_sold) AS cnt_amt
FROM sales s, products p WHERE s.prod_id = p.prod_id GROUP BY p.prod_name ;
```
Dimensions

- A way of describing complex data relationships
- Used to perform query rewrites, but not required
- Defines hierarchical relationships between pairs of columns
  - Hierarchies can have multiple levels
  - Each child in the hierarchy has one and only one parent
  - Each level key can identify one or more attributes

- Dimensions should be validated using the DBMS_Olap.VALIDATE_DIMENSION package
- Bad row ROWIDs stored in table: mview$_exceptions

Example of Creating A Dimension

```sql
CREATE DIMENSION time_dim
  LEVEL CAL_DATE IS calendar.CAL_DATE
  LEVEL PRD_ID IS calendar.PRD_ID
  LEVEL QTR_ID IS calendar.QTR_ID
  LEVEL YEAR_ID IS calendar.YEAR_ID
  LEVEL WEEK_IN_YEAR_ID IS calendar.WEEK_IN_YEAR_ID
HIERARCHY calendar_rollup
  (CAL_DATE CHILD OF PRD_ID CHILD OF QTR_ID CHILD OF YEAR_ID)
HIERARCHY week_rollup
  (CAL_DATE CHILD OF WEEK_IN_YEAR_ID CHILD OF YEAR_ID)
ATTRIBUTE PRD_ID DETERMINES PRD_DESC
ATTRIBUTE QTR_ID DETERMINES QTR_DESC;
```

Example of Using Dimensions

---

**Step 1 of 4**
- Create materialized view (join-aggregate type)

```sql
CREATE MATERIALIZED VIEW items_mv
BUILD IMMEDIATE
REFRESH ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT l.slr_id, c.cal_date, sum(l.gms) gms
FROM items l, calendar c
WHERE l.end_date = c.cal_date
GROUP BY l.slr_id, c.cal_date;
```

**Step 2 of 4**: (not really required, for demonstration only)
- Execute query based on “quarter”, not “date”, without a time dimension

```sql
SQL> select c.qtr_id, sum(l.gms) gms
FROM items l, calendar c
WHERE l.end_date = c.cal_date
GROUP BY l.slr_id, c.qtr_id;
```

**Execution Plan**

```
SELECT STATEMENT Optimizer=CHOOSE (Cost=16174 Card=36258…)
SORT (GROUP BY) (Cost=16174 Card=36258 Bytes=1160256)
HASH JOIN (Cost=81 Card=5611339 Bytes=179562848)
TABLE ACCESS (FULL) OF 'CALENDAR' (Cost=2 Card=8017…)
TABLE ACCESS (FULL) OF 'ITEMS' (Cost=76 Card=69993…)
```

---

Example of Using Dimensions (cont’d)

**Step 3 of 4**: Create time dimension (see slide 24 for SQL)
**Dimension Created**
- **Step 4 of 4**: Rerun query based on “quarter” with time dimension

```sql
SQL> select c.qtr_id, sum(l.gms) gms
FROM items l, calendar c
WHERE l.end_date = c.cal_date
GROUP BY l.slr_id, c.qtr_id;
```

**Execution Plan**

```
SELECT STATEMENT Optimizer=CHOOSE (Cost=16174 Card=36258…)
SORT (GROUP BY) (Cost=16174 Card=36258 Bytes=1160256)
HASH JOIN (Cost=81 Card=5611339 Bytes=179562848)
TABLE ACCESS (FULL) OF 'CALENDAR' (Cost=2 Card=8017…)
TABLE ACCESS (FULL) OF 'ITEMS_MV' (Cost=7 Card=10962…)
```

---

DW Partitioning, Oracle case

- Clearly a win to partition fact table, big MVs by time intervals for roll-out, clustering effect
- Can sub-partition fact table by a dimension attribute, but need to modify queries to get QP to optimize
- Ex: partition by date intervals, product category
- Query: select p.subcategory, … from f where … (no mention of p.category)
- Modified query: select p.subcategory … where … AND category='Soft Drinks' --now QP uses partition pruning
- MVs are usually rolled-up, much smaller, don’t need effective partitioning so much
Summary

- Query Rewrite using dimension hierarchies apparently helps only Oracle MVs, not partition pruning.
- So put raw data in one fact table, partitioned for roll-out.
- Create MVs with various roll-ups, for queries, also partitioned by time.
- Add indexes to MVs.
- Note MVs are much smaller than raw fact tables.
- Every day (say) add data to raw fact table, refresh MVs.

Oracle OLAP Cube

- Another way to hold data, optimized for cube queries.
- Related to master tables: fact tables, dimensions.
- Excel can get data with MDX.
- Not itself a MV, but can be used like one.
- i.e. SQL queries can be automatically rewritten to use the OLAP cube, run faster.
- Other OLAP servers exist too.

Working cheaply: what about mysql?

- If your data can be fit into memory, you don’t need fancy software… so buy a terabyte of memory… no longer a crazy idea.
- Example: Dell’s PowerEdge FX2 FC830 (review June ‘15) can take up to 1.5TB memory, 4 CPU sockets for Xeon processors with 4-18 cores/CPU. Basic system (8GB memory) $8,300. Maybe $15K for 1TB compatible RAM (not sure).
- Have warehouse data in mysql on disk, comes into memory as accessed.
- Mysql has no MV's, but can compute a joined table periodically as needed for Excel.
- Use Excel for UI.