MATERIALIZED VIEWS – A REVIEW

Carl Dudley, University of Wolverhampton, UK

INTRODUCTION

Data warehouses typically contain a few very large tables. Joins and aggregate queries involving these tables can be very expensive. Materialized views (MVs) contain pre-calculated results of such queries and are designed to enhance their performance. The materialized views often contain summary data based on aggregations of the table data. The summaries are usually defined as aggregate views and/or join views, for example, a view containing sums of salaries of employees by job within department. The materialized views are not normally accessed directly. Instead, queries on base tables are automatically re-written by Oracle to use MVs. The MVs can be nested in that they can be based on other MVs.

In this way, the use of MVs can be completely transparent to applications. They can be created and dropped without affecting any functionality. Unlike normal views they physically contain their own data derived from database tables. This imposes an overhead. If the data in the materialized view needs to be up to date it will have to be refreshed (updated) as the base table data changes.

Materialized Views can also used to create replica snapshots of tables on remote databases. This feature is not covered in this paper.

CREATING MATERIALIZED VIEWS

```
CREATE MATERIALIZED VIEW emv
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
AS
SELECT deptno
      ,SUM(sal)
FROM emp
GROUP BY deptno;
```

This view is populated with data on creation (`BUILD IMMEDIATE` is the default). It is also refreshed (updated) on demand by wiping all of its data (truncating the view) and reconstructing the entire dataset. Queries against the `emp` table will *NOT* be rewritten to use the view.

```
CREATE MATERIALIZED VIEW emv
BUILD DEFERRED
REFRESH FAST ON COMMIT
ENABLE QUERY REWRITE
AS
SELECT deptno
      ,SUM(sal)
FROM emp
GROUP BY deptno;
```

This view will not be populated until it is first used in a query. It will be refreshed whenever a transaction on the base `emp` table commits, so imposing an overhead on normal processing. It will be available to the optimizer for use in query rewrite so that queries written against the `emp` table may be rewritten to use the view as appropriate.

QUERY REWRITE

The optimizer produces query plans with and without the use of MVs. It compares the costs of these plans and then selects the plan of the least cost. If the rewrite against the MV is found to be of lowest cost it will use the materialized view to produce the results. When query rewrite occurs, the underlying base tables are not used and performance is normally improved. A few carefully chosen MVs can support a large range of queries. Even when an MV does not contain all the required information, it can still be used in query optimization because Oracle can join it to base tables in order to satisfy the query.

Several factors affect whether query rewrite will occur

1. Enabling or disabling query rewrite can be achieved via the
   `CREATE` or `ALTER` statement for individual materialized views
2. `REWRITE` and `NOREWRITE` hints in individual SQL statements
   `SELECT /* +NOREWRITE */...`

   The `REWRITE_OR_ERROR` hint will prevent the processing of a statement if it is not rewritten
3. Rewrite integrity levels

   Some levels allow the use of stale data which can give different results to the original statement.
4. Dimensions and constraints

   Outside the scope of this paper, but details are in the Oracle documentation.

The accuracy of Query rewrite can be controlled using the `QUERY_REWRITE_INTEGRITY` parameter which can be set at system or session level.

```
ALTER SESSION
SET QUERY_REWRITE_INTEGRITY = ENFORCED|TRUSTED|STALE_TOLERATED;
```

`ENFORCED` (default)	MV data must be fresh and any constraints used must be enabled and validated
`TRUSTED`	Trusts data conforms to constraints that are not validated by Oracle
`STALE_TOLERATED`	Allows stale MVs to be used

Only the `ENFORCED` level guarantees totally accurate results, in that MVs will not be used if the base table changes and they are not refreshed. A `COMPILE_STATE` of `NEEDS_COMPILE` in `user_mviews` means Oracle cannot properly determine the staleness of the MV

```
SELECT mview_name,refresh_mode,staleness,compile_state
FROM user_mviews;
```

```
MVIEW_NAME                REFRESH_MODE STALENESS     COMPILE_STATE
------------------------- ------------ ------------- -------------
CAL_MONTH_SALES_MV        DEMAND       NEEDS_COMPILE NEEDS_COMPILE
FWEEK_PSCAT_SALES_MV      DEMAND       STALE         VALID
SUM_SALES_PSCAT_WEEK_MV   COMMIT       FRESH         VALID
```

Compiling the view can establish its status.

```
ALTER MATERIALIZED VIEW cal_month_sales_mv COMPILE;

SELECT mview_name,refresh_mode,staleness,compile_state
FROM user_mviews;

MVIEW_NAME                REFRESH_MODE STALENESS     COMPILE_STATE
------------------------- ------------ ------------- -------------
CAL_MONTH_SALES_MV        DEMAND       STALE         VALID
FWEEK_PSCAT_SALES_MV      DEMAND       STALE         VALID
SUM_SALES_PSCAT_WEEK_MV   COMMIT       FRESH         VALID
```

SAMPLE DATABASE USED IN TESTS

A standard `sales` table of 73475444 rows was used for the tests.

`sales` table columns :	`product_id`	`channel_id`	`time_id`	`customer_id`
	`promo_id`	`quantity_sold`	`amount_sold`	

Primary key : `product_id, channel_id, time_id, customer_id`

DATA DISTRIBUTION IN SALES TABLE

Column name	Distinct Values
Prod_id	10
Channel_id	5
Time_id	1460
Cust_id	175

QUERY REWRITE PERFORMANCE

SAMPLE MV ON SALES TABLE.

```
CREATE MATERIALIZED VIEW sales_prod_mv
BUILD IMMEDIATE
ENABLE QUERY REWRITE AS
SELECT prod_id
      ,SUM(amount_sold)
FROM sales
GROUP BY prod_id;
```

Sample Query

```
SELECT prod_id
      ,SUM(amount_sold)
FROM sales
GROUP BY prod_id;
```

This query is an ‘exact text match’ with the select statement in the view definition. When the view is present the following plan is observed.

```
----------------------------------------------------------------
|Id | Operation           | Name          |Rows|Bytes | Cost(%CPU)|
----------------------------------------------------------------
|  0| SELECT STATEMENT    |               |  10| 1872 |  3     (0)|
|  1|  MAT_VIEW REWRITE   | SALES_PROD_MV|  72| 1872 |  3     (0)|
|   |  ACCESS FULL        |               |    |      |           |
----------------------------------------------------------------
```

The elapsed time was 0.03 seconds

In the absence of the MV a very different situation arises with a full scan of the base table.

```
-----------------------------------------------------------------
|Id   | Operation          | Name  |Rows   | Bytes | Cost(%CPU) |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |    73 |   657 |  1182  (17)|
|   1 |  HASH GROUP BY     |       |    73 |   657 |  1182  (17)|
|   2 |   TABLE ACCESS FULL| SALES |   932K|  8196K|  1038   (5)|
-----------------------------------------------------------------
```

The elapsed time was 16.21 seconds. Obviously, the MV provides a vast improvement.

Explaining the Rewrite

Query rewrite occurs transparently, so checks have to be made to establish if it does happen. Checks can be made using :

1. `AUTOTRACE` (as shown above),
2. An oracle supplied procedure – `dbms_mview.explain_rewrite`

   This indicates reasons why the rewrite did not occur. Actually the rules governing this are extremely complex, so the information provided by this procedure is very useful. It populates the `rewrite_table` showing which materialized views will be used and produces comparative costings of the original and re-written query.

   Each Oracle account must have its own `rewrite_table` which can be created by running the `utlxrw` script found in the `rdbms/admin` directory.

Example 1.

```
BEGIN
  DBMS_MVIEW.EXPLAIN_REWRITE
     ('SELECT prod_id,SUM(prod_id)
       FROM sales GROUP BY prod_id');
END;
/
SELECT message
      ,original_cost
      ,rewritten_cost
FROM rewrite_table;

MESSAGE                           ORIGINAL_COST REWRITTEN_COST
--------------------------------- ------------- --------------
QSM-01009: materialized view,               180              3
prod_sum_mv,matched query text
```

The report shows an exact text match between the view definition and the query under test as the reason for using the MV. (There are many other criteria used by Oracle to establish whether the use of an MV is appropriate.)

Example 2.

```
BEGIN
  DBMS_MVIEW.EXPLAIN_REWRITE
   ('SELECT prod_id,AVG(amount_sold)
     FROM sales GROUP BY amount_sold');
END;
/

SELECT message
      ,original_cost
      ,rewritten_cost
FROM rewrite_table;

MESSAGE                                   ORIGINAL REWRITTEN
                                          _COST    _COST
----------------------------------------- -------- ---------
QSM-01086: dimension(s) not present or           0         0
not used in ENFORCED integrity mode

QSM-01065: materialized view, prod_sum_mv,       0         0
Cannot compute measure, AVG, in the query
```

This report shows that the required average cannot be obtained from the MV.

REFRESHING MATERIALIZED VIEWS

If changes are made to base tables, the MVs will need to be refreshed if correct information is to be produced. There are two ways in which this can be achieved.

1. `REFRESH ON COMMIT`
   This method keeps an MV up to date and synchronized with its base tables, but can have a performance impact on base table transactions. This is because the MV refresh will occur as part of the transaction. The `COMMIT` will not succeed until the refresh has finished.

2. `REFRESH ON DEMAND`
   This method requires a procedure to be invoked (on demand), which specifies the view to be refreshed.

   ```
   dbms_mview.refresh('emp_mv1')
   ```

   The procedure can take another argument to specify the type of refresh.

`COMPLETE`	performs a complete refresh by truncating the view and repopulating it
`FAST`	incrementally applies changes to the view, updating only the affected rows within the view
`FORCE`	performs a fast refresh if possible, if not it forces a complete refresh
`NEVER`	the materialized view will not be refreshed with refresh mechanisms.

Fast Refresh Operations - Materialized View Logs

Materialized view logs are required for fast refresh operations. These are defined using `CREATE MATERIALIZED VIEW LOG` on the base table. Only one log per table is allowed and the log cannot be explicitly named.

The log is named by Oracle as `mlog$_<base_table_name>.` MV logs are *not* created on the materialized view. Each inserted and deleted row in the base table creates a row in the log. An update on the base table may generate two rows in the log for both old and new values. It is normal to include the `ROWID` of affected rows in the log by specifying the `WITH ROWID` clause. In addition, logs that support aggregate materialized views must normally be created as follows.

1. Be defined with the `INCLUDING NEW VALUES` clause
2. Be defined with the `SEQUENCE` clause if the table is expected to have a mix of inserts/direct-loads, deletes, and updates
3. Contain every column in the table that is referenced in the materialized view.

There are many other requirements and restrictions as laid out in the Oracle Data Warehousing Guide.

Tuning Materialized Views for Fast Refresh and Query Rewrite

The procedure `dbms_advisor.tune_mview` can be used to optimize `CREATE MATERIALIZED VIEW` statements. It shows how to implement materialized views and any required logs. The output statements for its 'IMPLEMENTATION' information include:

1. CREATE MATERIALIZED VIEW LOG statements
   Shows how to create any missing materialized view logs required for fast refresh

2. ALTER MATERIALIZED VIEW LOG FORCE statements

Shows fixes for any materialized view log requirements such as missing filter columns, sequence, and so on, required for fast refresh

3. One or more CREATE MATERIALIZED VIEW statements

   Recommends and shows how to add additional required columns. For example, add `ROWID` column for materialized join view and add aggregate column for materialized aggregate view.

Tuning Materialized Views – Using the Advisor

The `dbms_advisor` package can be used to ascertain whether a specified view can undergo fast refresh. It populates a view called `user_tune_mview`.

Example 1

```
VARIABLE task_cust_mv VARCHAR2(30);
VARIABLE create_mv_ddl VARCHAR2(4000);

BEGIN
   :task_cust_mv := 'emp_mv1';
   :create_mv_ddl := 'CREATE MATERIALIZED VIEW emp_mv1
                      REFRESH FAST
                      ENABLE QUERY REWRITE AS
                      SELECT deptno,sum(sal)
                      FROM emp
                      GROUP BY deptno';
   DBMS_ADVISOR.TUNE_MVIEW(:task_cust_mv, :create_mv_ddl);
END;
```

The output from the routine is shown below

```
SELECT statement FROM user_tune_mview
WHERE task_name= :task_cust_mv AND script_type='IMPLEMENTATION';

STATEMENT
------------------------------------------------------------------
CREATE MATERIALIZED VIEW LOG ON "SCOTT"."EMP" WITH ROWID, SEQUENCE
   "SAL","DEPTNO")  INCLUDING NEW VALUES

ALTER MATERIALIZED VIEW LOG FORCE ON "SCOTT"."EMP" ADD ROWID, SEQUENCE
   "SAL","DEPTNO")  INCLUDING NEW VALUES

CREATE MATERIALIZED VIEW SCOTT.EMP_MV1   REFRESH FAST WITH ROWID ENABLE QUERY
REWRITE AS SELECT SCOTT.EMP.DEPTNO C1, SUM("SCOTT"."EMP"."SAL") M1,
COUNT("SCOTT"."EMP"."SAL") M2, COUNT(*) M3 FROM SCOTT.EMP GROUP BY
SCOTT.EMP.DEPTNO
```

For the view to be fast refreshed, the output recommends that the MV log is created and that a count of the aggregated column and of the rows is included in the view.

Example 2

```
BEGIN
   :task_cust_mv := 'emp_mv2';
   :create_mv_ddl := 'CREATE MATERIALIZED VIEW emp_mv2
                      REFRESH FAST -
                      ENABLE QUERY REWRITE AS
                      SELECT dept.deptno,dname,sum(sal)
                      FROM emp,dept
                      WHERE dept.deptno = emp.deptno
                      GROUP BY dept.deptno,dname'
   DBMS_ADVISOR.TUNE_MVIEW(:task_cust_mv, :create_mv_ddl);
END;

SELECT statement FROM user_tune_mview
WHERE task_name= :task_cust_mv AND script_type='IMPLEMENTATION';

STATEMENT
--------------------------------------------------------------------
CREATE MATERIALIZED VIEW LOG ON "SCOTT"."DEPT" WITH ROWID, SEQUENCE
("DEPTNO","DNAME")  INCLUDING NEW VALUES

ALTER MATERIALIZED VIEW LOG FORCE ON "SCOTT"."DEPT" ADD ROWID, SEQUENCE
("DEPTNO","DNAME")  INCLUDING NEW VALUES

CREATE MATERIALIZED VIEW LOG ON "SCOTT"."EMP" WITH ROWID, SEQUENCE
("SAL","DEPTNO")  INCLUDING NEW VALUES

ALTER MATERIALIZED VIEW LOG FORCE ON "SCOTT"."EMP" ADD ROWID, SEQUENCE
("SAL","DEPTNO")  INCLUDING NEW VALUES

CREATE MATERIALIZED VIEW SCOTT.EMP_MV2 REFRESH FAST WITH ROWID ENABLE
QUERY REWRITE AS SELECT SCOTT.DEPT.DNAME C1, SCOTT.DEPT.DEPTNO C2,
SUM("SCOTT"."EMP"."SAL") M1, COUNT("SCOTT"."EMP"."SAL") M2,
COUNT(*) M3 FROM SCOTT.EMP, SCOTT.DEPT WHERE SCOTT.DEPT.DEPTNO =
SCOTT.EMP.DEPTNO GROUP BY SCOTT.DEPT.DNAME, SCOTT.DEPT.DEPTNO
```

Output recommends logs on all base tables for fast refresh to be possible. (A log on the `emp` table already exists, but needs to be altered.)

MATERIALIZED VIEW CAPABILITIES

The `dbms_mview.explain_mview` procedure can be used to find and explain the capabilities (query rewrite, refresh modes) of an MV. Information is placed in the `mv_capabilities_table`. This table has one row for each reason why a certain action is not possible and needs to be built in each investigating account by executing the script `utlxmv.sql` found in the `rdbms/admin` directory.

```
EXECUTE dbms_mview.explain_mview('emp_mv1','6')
```

The optional second argument (in this case '6') is used as a statement identifier, allowing the `statement_id` column to be used to fetch rows for a given MV.

```
SELECT capability_name,possible,msgtxt
FROM mv_capabilities_table WHERE statement_id = '6';

CAPABILITY_NAME                P MSGTXT
------------------------------ - ----------------------------------------------
PCT                            N
REFRESH_COMPLETE               Y
REFRESH_FAST                   Y
REWRITE                        Y
PCT_TABLE                      N relation is not a partitioned table
REFRESH_FAST_AFTER_INSERT      Y
REFRESH_FAST_AFTER_ONETAB_DML  N SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ONETAB_DML  N COUNT(*) is not present in the select list
REFRESH_FAST_AFTER_ANY_DML     N mv log does not have sequence #
REFRESH_FAST_AFTER_ANY_DML     N see the reason why REFRESH_FAST_AFTER_ONETAB_DML
                                  is disabled
REFRESH_FAST_PCT               N PCT is not possible on any of the detail tables
                                  in the materialized view
REWRITE_FULL_TEXT_MATCH        Y
REWRITE_PARTIAL_TEXT_MATCH     Y
REWRITE_GENERAL                Y
REWRITE_PCT                    N general rewrite is not possible or PCT is not
                                  possible on any of the detail tables
PCT_TABLE_REWRITE              N relation is not a partitioned table
```

SUMMARY OF RESTRICTIONS ON FAST REFRESH OF MVS

Joins…

ROWIDs of all tables in FROM list must appear in the SELECT list of the query

Materialized view logs must exist on all tables involved in the join

Aggregates…

Only SUM, COUNT, AVG, STDDEV, VARIANCE, MIN and MAX are supported for fast refresh

COUNT(*) must be included in the view

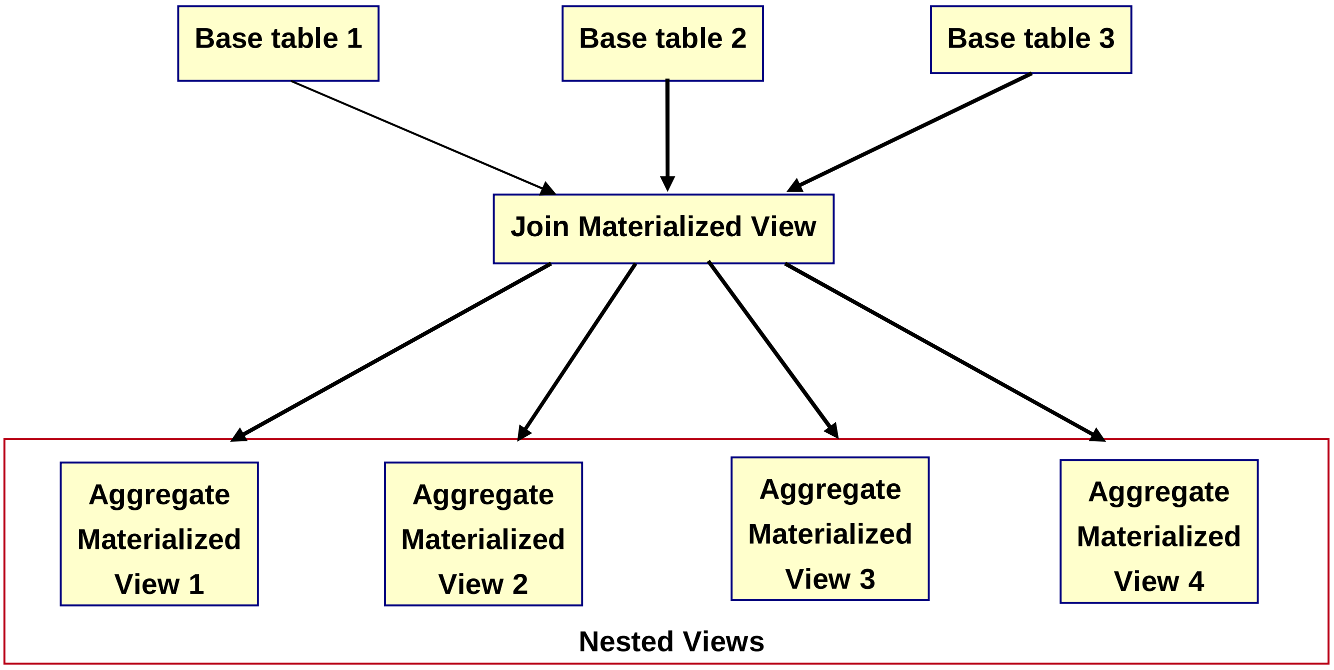
For each aggregate such as AVG(expr), the corresponding COUNT(expr) must be present

MECHANISM OF FAST REFRESH ON COMMIT

1. Create suitable materialized view logs on all base tables.
2. Build and populate materialized view with fast refresh
3. Any changes made to base tables are inserted into the MV log.
4. On transaction commit – these changes are propagated to the MV in a fast refresh action and the MV log entries are removed

NESTED MATERIALIZED VIEWS

This situation can be described as Nested MVs being based on parent (master) MVs. The master MV is typically a join MV. This may be large but contains pre-joined data. The nested views are aggregates based on the master view's joined data. This gives great flexibility in the construction of a suite of MVs as shown below.


Base table 1
Base table 2
Base table 3
Join Materialized View
Aggregate Materialized View 1
Aggregate Materialized View 2
Aggregate Materialized View 3
Aggregate Materialized View 4
Nested Views

The base objects must have a suitable materialized view log already defined with `ROWID`.

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON products WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON channels WITH ROWID;
```

The master MV can now be created.

```
CREATE MATERIALIZED VIEW master_nest_mv
BUILD IMMEDIATE
ENABLE QUERY REWRITE
AS
SELECT p.prod_id, p.prod_name, c.channel_id, c.channel_desc
FROM sales s, products p, channels c
WHERE s.prod_id = p.prod_id
AND s.channel_id = c.channel_id;
```

The master view itself must have a suitable materialized view log containing all the master view columns, `ROWID`, and all new values.

```
CREATE MATERIALIZED VIEW LOG ON master_nest_mv WITH ROWID
(prod_id, prod_name, channel_id, channel_desc)
INCLUDING NEW VALUES;
```

The nested MV can then be created on the master MV.

```
CREATE MATERIALIZED VIEW nested_pr_ch_mv
```

```
BUILD IMMEDIATE
ENABLE QUERY REWRITE
AS
SELECT channel_desc, prod_name, COUNT(prod_id)
FROM master_nest_mv
GROUP BY channel_desc, prod_name;
```

AUTOTRACE shows evidence of the use of the MV in queries, but does not report on the hierarchy.

```
SELECT p.prod_id, p.prod_name, c.channel_id, c.channel_desc
FROM sales s, products p, channels c
WHERE s.prod_id = p.prod_id
AND s.channel_id = c.channel_id;
```

```
-------------------------------------------------------------------------------
|Id| Operation                  |Name          |Rows|Bytes|Cost(%CPU)| Time    |
-------------------------------------------------------------------------------
| 0|SELECT STATEMENT            |              | 225| 9450| 3  (0)   |00:00:01|
| 1|MAT_VIEW REWRITE ACCESS FULL|NESTED_PR_CH_MV| 225| 9450| 3  (0)   |00:00:01|
-------------------------------------------------------------------------------
```

However, the explain_rewrite procedure gives full details.

```
BEGIN
   DBMS_MVIEW.EXPLAIN_REWRITE(
       'SELECT c.channel_desc,  p.prod_name, COUNT(p.prod_id)
        FROM sales s, products p, channels c
        WHERE s.prod_id = p.prod_id
        AND c.channel_id = s.channel_id
        GROUP BY channel_desc, prod_name'
        ,'NESTED_PR_CH_MV');
END;
/

SELECT message FROM rewrite_table;
MESSAGE
-----------------------------------------------------------------
QSM-01151: query was rewritten
QSM-01033: query rewritten with materialized view, NESTED_PR_CH_MV
QSM-01336: the materialized view you specified (NESTED_PR_CH_MV)
           was not used to rewrite the query
QSM-01033: query rewritten with materialized view, MASTER_NEST_MV
```

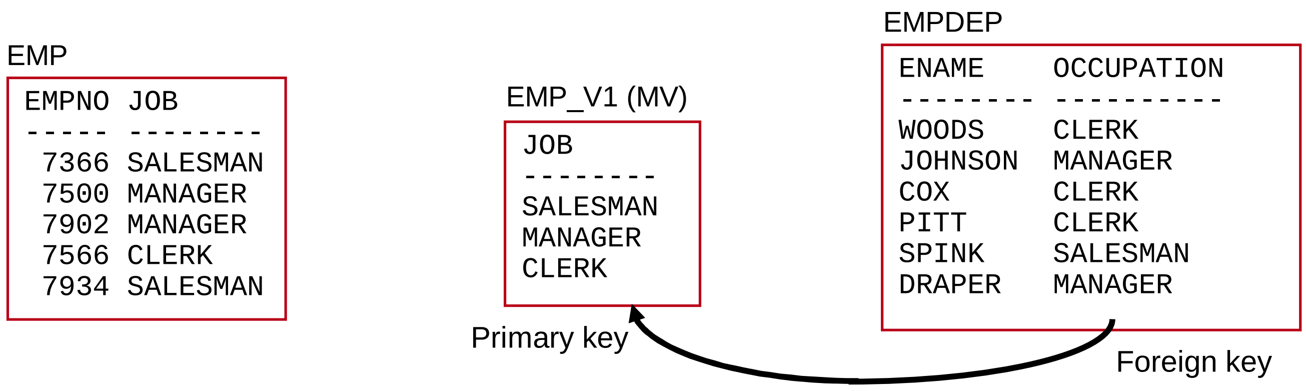
REFRESHING NESTED MVS

If the base tables are updated without refreshing the master MV, it will go stale. However any nested MVs will remain fresh with respect to the master. If the master MV is subsequently refreshed, the nested MVs will go stale until they themselves are refreshed. If the refresh on the master is performed with `NESTED => TRUE`, all views in the hierarchy are refreshed.

```
EXEC dbms_mview.refresh('master_nest_view',NESTED=>TRUE);
```

Using Materialized Views to allow Foreign Keys to Reference Non-unique Columns

There is sometimes a need to enforce a foreign key constraint on a column that refers to a non-unique column in another table. This problem can be solved by referencing a materialized view carrying only unique values of the referenced column.



EMP

```
EMPNO JOB
----- --------
 7366 SALESMAN
 7500 MANAGER
 7902 MANAGER
 7566 CLERK
 7934 SALESMAN
```

EMP_V1 (MV)

```
JOB
--------
SALESMAN
MANAGER
CLERK
```

EMPDEP

```
ENAME    OCCUPATION
-------- ----------
WOODS    CLERK
JOHNSON  MANAGER
COX      CLERK
PITT     CLERK
SPINK    SALESMAN
DRAPER   MANAGER
```

```
CREATE MATERIALIZED VIEW LOG ON emp
WITH ROWID,PRIMARY KEY,SEQUENCE(job)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW emp_v1
REFRESH FAST ON COMMIT
ENABLE QUERY REWRITE
AS SELECT job FROM emp GROUP BY job;

ALTER MATERIALIZED VIEW emp_v1 ADD PRIMARY KEY (job);

ALTER TABLE empdep ADD CONSTRAINT empdep_emp_v1
FOREIGN KEY (occupation) REFERENCES emp_v1;
```

The foreign key references the MV

```
UPDATE empdep SET occupation = 'X';
ORA-02291: integrity constraint (SCOTT.EMPDEP_EMP_V1) violated - parent key not
found
```

This erroneous update is prevented by referring to the unique values in the MV.

Enforcing Complex Constraints with Materialized Views

Example complex constraint :

Limit the total of `amount_sold` of a single product sold through a single channel to 563000000.

First create a materialized view, `prod_chan_mv`, which returns the maximum amount sold for any product on any channel.

```
CREATE MATERIALIZED VIEW prod_chan_mv
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
AS
SELECT prod_id, channel_id, SUM(amount_sold) sum_amount_sold
FROM sales
GROUP BY prod_id, channel_id;
```

`REFRESH FAST ON COMMIT` is necessary as the constraint must be checked each time a change to the sales table is committed.

Now find the current maximum amount sold across all products within channels.

```
SELECT prod_id, channel_id, sum_amount_sold
FROM prod_chan_mv
WHERE sum_amount_sold = (SELECT MAX(sum_amount_sold)
                         FROM prod_chan_mv);

   PROD_ID CHANNEL_ID SUM_AMOUNT_SOLD
---------- ---------- ---------------
        18          3       562565473
```

As the maximum amount sold was 562565473 a sensible limit (as an example) on the `sum_amount_sold` column in the MV could be 563000000.

```
ALTER TABLE prod_chan_mv
ADD CONSTRAINT amount_sold_check
CHECK (sum_amount_sold < 563000000)
DEFERRABLE;
```

The `DEFERRABLE` option ensures that the constraint is checked on commit rather than on update.

Insert a row into the base table with an `amount_sold` value of 8000000. This will violate the constraint on the MV.

```
INSERT INTO sales VALUES(18, 1, '01-JAN-2002', 3, 999, 1, 8000000);
```

The constraint is not enforced until the commit (and hence refresh) takes place.

```
SQL> COMMIT;
COMMIT
ERROR at line 1:
ORA-12048: error encountered while refreshing materialized
View "SH"."PROD_CHAN_MV"
```

```
ORA-02290: check constraint (SH.AMOUNT_SOLD_CHECK) violated
```

This scenario could lead to large numbers of rows being inserted in the MV when the constraint is repeatedly NOT violated when the base table is changed.

An attempt could be made to avoid storage overheads when enforcing constraints with MVs by using a `HAVING` clause. This could make it possible to collect into the MV only those rows that have `amount_sold` greater than 563000000.

```
CREATE MATERIALIZED VIEW prod_chan_mv
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
AS
SELECT prod_id
      ,channel_id
      ,SUM(amount_sold) sum_amount_sold
FROM sales
GROUP BY prod_id, channel_id
HAVING SUM(amount_sold) > 563000000;
```

Unfortunately, this is not possible, as the presence of the `HAVING` clause makes the view a 'complex MV' which cannot be refreshed on commit. This is still the case in Oracle11g. *If* it were possible to refresh on commit with a `HAVING` clause, a constraint could be added to the MV that checks that if the `prod_id` is null. The constraint would stop any of the collectable rows being committed into the view. The key column, `prod_id`, in the `products` table cannot be null. Any row successfully submitted to the materialized view must have a non-null value for `prod_id`. In this way, the MV would remain permanently empty whilst still providing the integrity check.

```
ALTER TABLE prod_chan_mv
ADD CONSTRAINT amount_sold_check_new
CHECK prod_id IS NULL;
```

INDEXES ON MATERIALIZED VIEWS

GROUPED (AGGREGATE) VIEWS

A single unique index is often created on an MV when it is built. If the MV is a grouped (aggregate) MV, the index is a function-based Btree, has a name of `I_SNAP$_<mv_name>`, and is built on a virtual `RAW` column(s) built into the MV

The number of columns in the index is equal to the number of grouping columns and the names of these columns are similar to `sys_nc00003$.`

The following statement generates two hidden MV columns that are based on the `prod_subcategory` and `week_ending_day` columns. An index is built on each of the hidden columns.

```
CREATE MATERIALIZED VIEW sales_mv AS
SELECT p.prod_subcategory, t.week_ending_day,
       SUM(s.amount_sold) AS sum_amount_sole
FROM sales s, products p, times t
WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY p.prod_subcategory, t.week_ending_day;
```

The following query shows the values in these columns (and their system-provided names). Note that the values are a 'RAW' representation of the values in the normal columns.

```
SELECT sys_nc00004$
      ,sys_nc00005$
      ,prod_subcategory
      ,week_ending_day
FROM sales_mv;

SYS_NC00004$             SYS_NC00005$     PROD_SUBCATEGORY WEEK_ENDING_DAY
------------------------ ---------------- ---------------- ---------------
43616D6572617300         77C6020101010100 Cameras          01-FEB-98
4D6F6E69746F727300       77C6031601010100 Monitors         22-MAR-98
4465736B746F702050437300 77C6030801010100 Desktop PCs      08-MAR-98
43616D636F726465727300   77C6010B01010100 Camcorders       11-JAN-98
43616D636F726465727300   77C6011901010100 Camcorders       25-JAN-98
4163636573736F7269657300 77C6020101010100 Accessories      01-FEB-98
4163636573736F7269657300 77C6030101010100 Accessories      01-MAR-98
486F6D6520417564696F00   77C6010B01010100 Home Audio       11-JAN-98
486F6D6520417564696F00   77C6031601010100 Home Audio       22-MAR-98
           :                      :             :                :
```

INDEXES ON NON-GROUPED MATERIALIZED VIEWS

If the MV has a one to one mapping of rows with the base table, and the base table has a primary key, then a normal Btree index is built on that column within the MV. If there is a primary key constraint on the base table the index is explicitly named with that of the constraint name (e.g PK_EMP). If there is no primary key on the base table, no index is built on the MV.

MANAGING UPDATE AND REFRESH OPERATIONS

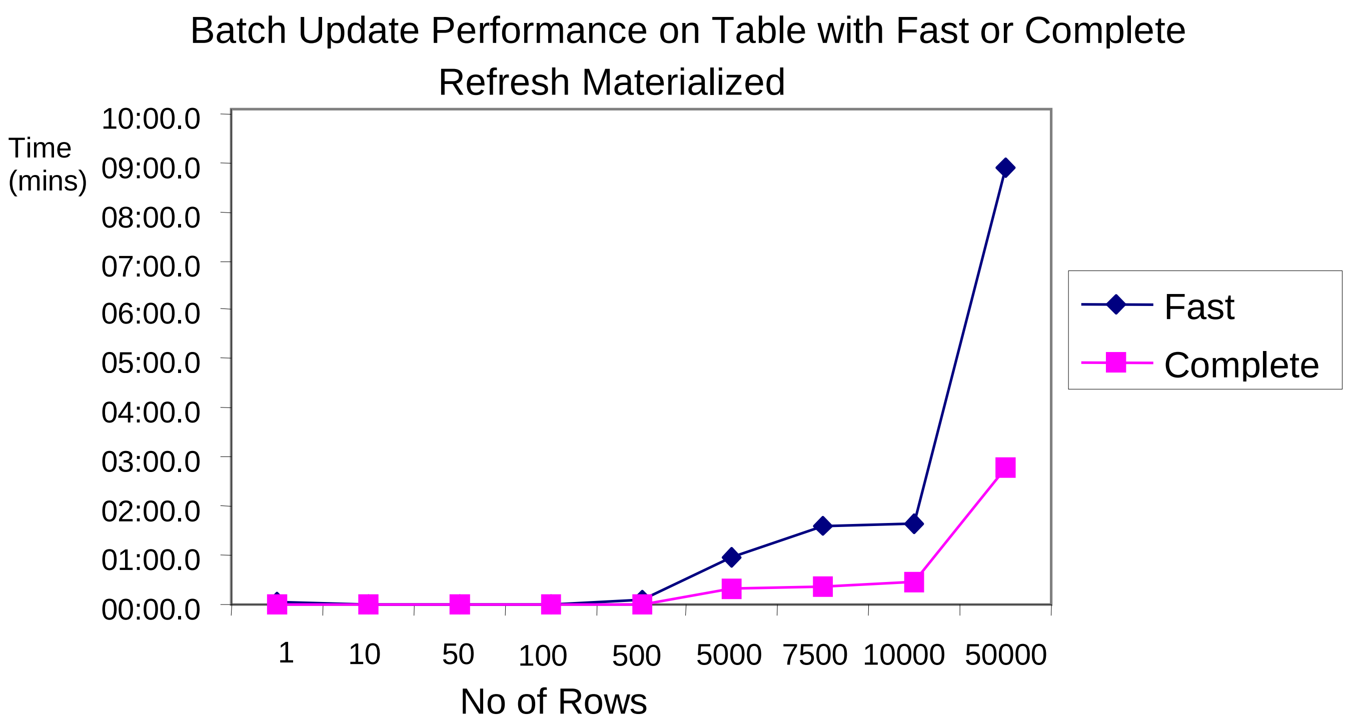
Tests were performed on a 918443 row `sales` table having one MV in fast or complete refresh mode. Updates to the base `sales` table were performed in batch mode (a single update statement affecting many rows).

Timings were collected for the actual update operation and also the subsequent refresh operation on the MV.

No. of rows updated	Update		Refresh	
	Fast	Complete	Fast	Complete
1	00:00.2	00:00.1	00:01.4	00:13.2
100	00:00.2	00:00.2	00:02.2	00:07.3
500	00:00.5	00:00.1	00:00.2	00:09.3
1000	00:00.7	00:00.1	00:01.5	00:07.7
5000	00:06.5	00:00.3	00:01.6	00:08.0
50000	00:58.0	00:19.1	00:39.1	00:10.5

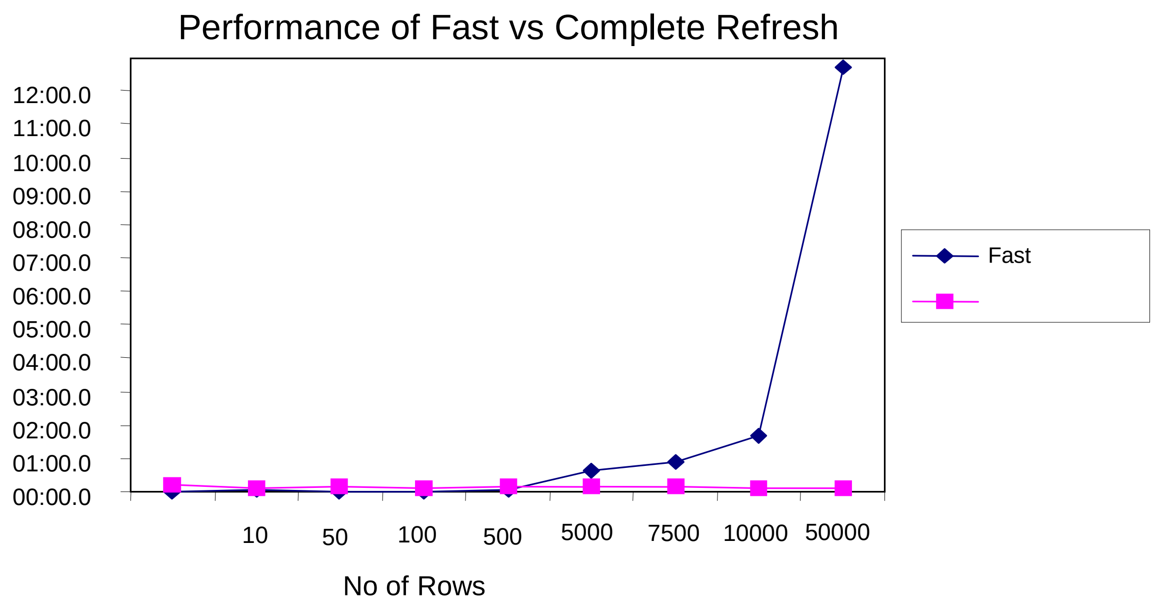
75000	00:35.7	00:22.3	00:53.5	00:08.5
100000	00:58.9	00:28.5	01:40.6	00:06.1
500000	08:52.9	02:47.7	12:40.8	00:07.2

COMPARISON OF UPDATE ACTIVITY WHEN USING FAST AND COMPLETE REFRESH MODES



The graph illustrates the overhead of maintaining the MV log on updating the base table(s) when fast refresh is required.

Comparison of Fast and Complete Refresh Performance



The MV is truncated during a complete refresh. This operation is comparatively faster for large scale updates.

Overheads on Online Transaction Processing

Repeated update transactions were made on a single row on the same base table.

```
BEGIN
  FOR i in 1..n LOOP
    UPDATE sales SET amount_sold = amount_sold + .01
    WHERE prod_id = 14
    AND cust_id = 50840
    AND time_id = '03-NOV-01'
    AND channel_id = 3;
    COMMIT;
  END LOOP;
END;
```

The `sales` table had an index to avoid repetitive full table scans masking the effect of refresh requirements on update performance.

```
ALTER TABLE sales ADD CONSTRAINT pk_sales
PRIMARY KEY (prod_id,cust_id,time_id,channel_id);
```

The update routine was timed, with and without an MV log on `sales`. The log was truncated after each test. Timings were also produced for the same routine without the `COMMIT` statement.

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID
   (cust_id
   ,prod_id
   ,channel_id
   ,time_id
   ,amount_sold
   ,quantity_sold)
INCLUDING NEW VALUES;
```

PERFORMANCE IMPACT OF MV LOGS

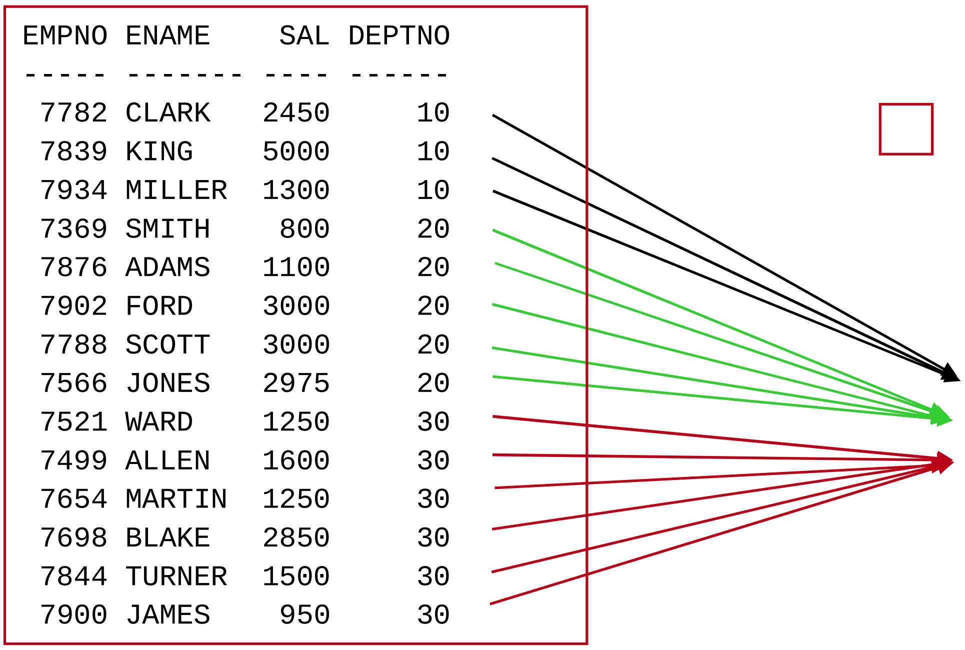
Number of iterations (n)	Time taken (seconds)			
	With commit		Without commit	
	No MV log	MV log present	No MV log	MV log present
500	0.21	0.25	0.03	0.15
1000	0.34	0.43	0.06	0.25
2500	0.62	0.96	0.23	0.51
5000	1.14	1.89	0.40	1.07
7500	1.67	2.81	0.61	1.58
10000	2.03	4.00	0.79	2.09
50000	11.10	18.59	3.93	10.62

The presence of the MV log can result in up to 250% performance overhead.

CONTENTION CAUSED BY MATERIALIZED VIEWS

Contention on a MV can arise when multiple concurrent transactions need to refresh the same row(s) in the materialized view. This obviously applies only to REFRESH ON COMMIT views in OLTP environments because the refresh takes place as part of the OLTP transaction. Aggregate MVs are prone to this type of contention.

The updates on base tables are 'condensed' into fewer rows in the MV which can cause locking and contention issues as transactions queue to update the grouped row in the MV.



```
EMPNO ENAME     SAL  DEPTNO
----- ------- ---- ------
 7782 CLARK    2450      10
 7839 KING     5000      10
 7934 MILLER   1300      10
 7369 SMITH     800      20
 7876 ADAMS    1100      20
 7902 FORD     3000      20
 7788 SCOTT    3000      20
 7566 JONES    2975      20
 7521 WARD     1250      30
 7499 ALLEN    1600      30
 7654 MARTIN   1250      30
 7698 BLAKE    2850      30
 7844 TURNER   1500      30
 7900 JAMES     950      30
```

Using Multiple Materialized Views

Consider two materialized views each covering only part of the base data. The base table contains `sales` data for `prod_id`s ranging from 10 to 19.

```
CREATE MATERIALIZED view prod_low_mv
BUILD IMMEDIATE
ENABLE QUERY REWRITE
AS
SELECT prod_id, SUM(amount_sold)
FROM sales
WHERE prod_id <= 15
GROUP BY prod_id;
```

Data held in view :

```
PROD_ID     SUM(AMOUNT_SOLD)
-------     ----------------
     14          733738663
     10         50236718.5
     15          734982650
     11         49879911.7
     13          662255842
     12         33032602.8
```

```
CREATE MATERIALIZED view prod_high_mv
BUILD IMMEDIATE
ENABLE QUERY REWRITE
AS
SELECT prod_id, SUM(amount_sold)
FROM sales
WHERE prod_id > 15
GROUP BY prod_id;
```

Data held in view :

```
PROD_ID SUM(AMOUNT_SOLD)
------- ----------------
```

```
                                         17          808449750
                                         16          220330955
                                         19         78746741.5
                                         18         1005874847
```

Now select the grouped data for all rows in the table.

```
SELECT prod_id, SUM(amount_sold)
FROM sales
WHERE prod_id > 0
GROUP BY prod_id;
```

Curiously, a `WHERE` clause is necessary for multiple views to be used in this way.

Oracle is able to use both views to deliver the complete output. Notice that the plan suggests a UNION-ALL operation is performed, but the same plan is shown even when duplicate rows are automatically eliminated due to the use of overlapping MVs.

Conclusion

Query Rewrite

MVs can return query results in a fraction of the time. They are considered for rewrite even if they contain only part of the data required by the query, however MVs will not (by default) be used for query rewrite if the base table(s) are updated. Query rewrite could use multiple materialized views to satisfy a single query, but there are some fairly severe restrictions in this area. There are useful tools to analyze the current situation and expose reasons why the rewrite does or does not occur.

Refreshing Materialized Views

This can be a major overhead in the use of MVs. Efficient refresh performance can be achieved using a fast refresh mechanism when the number of rows affected by a single transaction is small.. *BUT* if transaction rate is high, the refresh mechanism could have a significant effect. Materialized view logs are required for fast refresh, and these can also present overheads of storage and ultimately performance.

In summary, `REFRESH COMPLETE ON DEMAND` suits batch processing environments when the table updates are large and infrequent. `REFRESH FAST ON COMMIT` may suit an OLTP environment in which there are regular yet small table updates (often depending on real-time data), but careful testing is required as it could impose very significant overheads.

Versatility

MVs can also be used in the implementation of complex constraints, replication and the construction of sophisticated partitioning methods.