Materialized View:

أخذنا سابقاً مفهوم الـ View وطريقة إنشاؤها create view as select حيث أنها لم تكن تخزن الـ data بل كان عملية اسقاط على جدول آخر ما يعني أنه أمر logical فقط ففي حال تنفيذ الاستعلام التالي select * from view في هذه اللحظة يستدعي الاستعلام الأساسي الخاص بالـ view وينفذها فيتم تنفيذ الاستعلام المكتوب ضمنياً عند إنشاء الـ View .

أما في الـ MATERIALIZED VIEW يتم تخزين الـ data في table هو MATERIALIZED VIEW فيخزنها بطريقة يمكن أن يقوم بـ synchronize للـ table الأساسي بحيث أي تعديل عليه ينعكس على الـ data المخزنة في الـMATERIALIZED VIEW وذلك عند القيام بـ Refresh.

نستخدم الـ MATERIALIZED VIEW مع كلاً من :

- Aggregation تخزين الـ data الناتجة عن عمليات الـ Aggregation المعقدة .
- Join حيث يخفف الكثير من عمليات الـ Join .
- Aggregation and Join .

لكي نبني MATERIALIZED VIEW نحتاج إلى:

1. Log سيكون مهم فقط إذا كان لدينا MATERIALIZED VIEW له Refresh من نوع Complete .
2. الـ MATERIALIZED VIEW بحد ذاته.

تعليمة إنشاء MATERIALIZED VIEW :

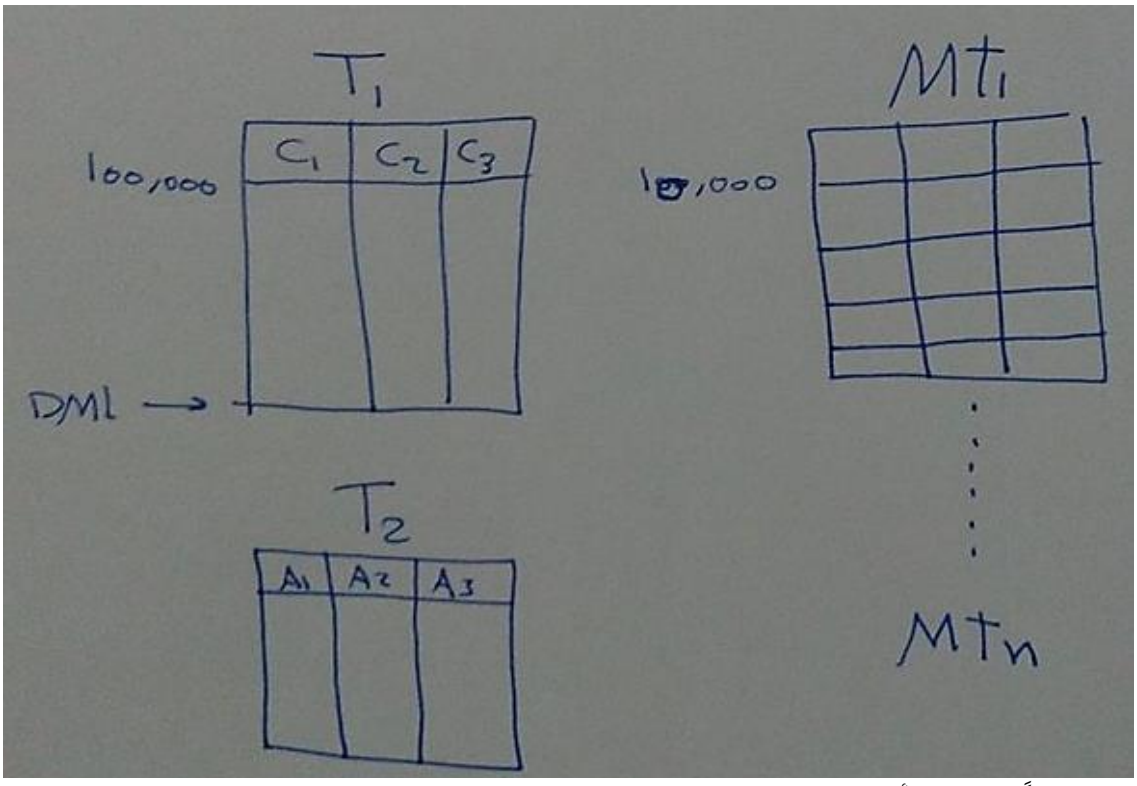
```
CREATE MATERIALIZED VIEW <schema.name>
TABLESPACE <tablespace_name>
BUILD <IMMEDIATE | DEFERRED>
REFRESH <COMPLETE | FAST | FORCE> ON <COMMIT | DEMAND>
<ENABLE QUERY REWRITE>
AS (<SQL statement>);
```

TABLESPACE : نقوم بتخزين الـ data فيه.

BUILD : IMMEDIATE يتم نقل الـ data إليه في هذه اللحظة ، DEFERRED عند القيام بـ Refresh.

REFRESH : له ثلاث أنواع COMPLETE , FAST , FORCE

لماذا نحتاج للـ REFRESH ؟



ليكن لدينا مثلاً جدولين لأول T1 والثاني T2 ونريد بناء MATERIALIZED VIEW يقوم بـ Aggregation وليكن مثلا الحقل c3 هو salary و نريد القيام بعملية Sum لهذا الحقل وبفرض لدينا 100000 row في T1 ثم قمنا بـ query نتيجتها 10000 row فسيتم تخزين هذه الـ data في MATERIALIZED VIEW وتكون موجودة بشكل Physically فيه ولن يحتاج عند جلبها مرة أخرى للذهاب إلى T1 .

في حال القيام بعملية DML على T1 (insert,update,delete) سيجري تغيير على الـ data ولكنه لن ينعكس على الـ MATERIALIZED VIEW حتى نقوم بـ Refresh .

"طأنواع الـ Refresh :

Complete Refresh: عندما نقوم بـ Refresh لـ MATERIALIZED VIEW يقوم بحذف الـ data التي فيه و يعيد قراءة الـ data من جديد من الجدول الأساسي و هي حركة ليست صحية ، **لماذا؟**

بفرض أن الـ MATERIALIZED VIEW مبني على n table وبينهم join و aggregation والـ data التي في داخلهم متغيرة بشكل كبير فكلما قمنا بعمل Refresh لـ MATERIALIZED VIEW سيعيد قراءة الـ data و يعيد الـ join والـaggregation وينقل الـ data إلى الـ MATERIALIZED VIEW .

الحل لهذه المشكلة (إعادة قراء الـ data من جديد بشكل كامل) هو Fast Refresh .

Fast Refresh : في هذا النوع لا نأخذ سوى الـ records المتغيرة فتتزامن الـ data كاملةً .

لكن كيف سنقوم بتحديد أن هذا الـ row حصل عليه تعديل ؟؟

نقوم ببناء Log من أجل كل table بنينا عليه الـ MATERIALIZED VIEW يحمل كامل تغيرات الـ Records الحاصلة عليه فعندما نقوم بـ Refresh لـ MATERIALIZED VIEW يذهب إلى الـ Log ويقرأ كل التعديلات على الجدول ويأخذ id الـ row المُعدّل فيحدث القيم الجديدة من الـ table الاساسي ثم يتم تفريغ الـ log .

وفي حال كان لدينا أكثر من MATERIALIZED VIEW لا يتم تفريغ الـ log حتى يأُخذ محتواه جميع الـ MATERIALIZED VIEW التي تعتمد عليه و لم يعد يحتاجه أحد منهم ، ولكل table يوجد log واحد فقط.

ملاحظة : هذه المعالجة تكون ضمنية في Oracle ولا يجب علينا القيام بها يدوياً .

Force Refresh : يختبر هل من الممكن أن يقوم بـ Fast refresh ؟ في حال عدم القدرة على ذلك يقوم بـ Complete Refresh

وتكون الـ Refresh : ON COMMIT أو ON DEMAND

ON COMMIT خيار خطير لأننا عندما نقوم بـ insert في T1 لن ينجح حتى نقوم بـ Refresh للـMATERIALIZED VIEW ، ولا يستخدم إلا نادراً لأنه من الممكن أن يكون MATERIALIZED VIEW في server آخر و لسبب ما كان هنالك عطل في الشبكة فلن تنجح الـ insert ، أي أنه لن يسجل الـ transition في الـ table حتى يضمن أنه سجل في الـ MATERIALIZED VIEW .

<ENABLE QUERY REWRITE>

قد نكون بنينا أكثر من MATERIALIZED VIEW بأنواع مختلفة ، ونريد تنفيذ query فعندما يريد الـ optimizer تحقيقها يتحقق فيما إذا كان هنالك MATERIALIZED VIEW تحقق هذه الـ query ، أي فرضاً كانت هذه الـ select على T1وT2 فيتحقق إذا كان هناك MATERIALIZED VIEW عليهما سوية و يحقق القيم التي كانت في الـ criteria الخاصة بهذه الـ select ، والأهم أن يتحقق إذا كان هذا الـ MATERIALIZED VIEW حصل له Refresh ، فيأخذ نتيجة هذه الـ query من الـ MATERIALIZED VIEW بدل من أن يأخذها من الـtables الأساسين .

وهو خيار مساعد جدا لأن التغيرات على الـ warehouse تكون قليلة فلن أقوم بـ refresh كل يوم لأننا نتعامل مع historical data أي نحن نقوم بترحيلها كل 6 أشهر أو كل سنة أو ... ، أي لا نتعامل مع online database .

كيف نبني الـ log ؟

```
CREATE MATERIALIZED VIEW LOG ON <schema.tablename>
WITH <SEQUENCE, PRIMARY KEY, ROWID> (filter columns)
<INCLUDING NEW VALUES>;
```

PRIMARY KEY : لضمان ربط التغيرات التي تحصل في الـ log مع الـ table و معرفة السطر الذي حصل عليه التعديل.

ROWID : إذا لم يكن هنالك PRIMARY KEY في الـ table ويتم توليده تلقائياً وهو unique على مستوى الـ DB .

SEQUENCE : يعطي قيمة متسلسلة لكامل تغيرات الـ records فنستطيع عندها القيام بـ query لاستعلام عن التغييرات بوضع order by sequence وهي أفضل من order by date لأن التاريخ ممكن أن يكون معدل .

filter columns : وهو خيار لنا ، لأن الـ optimizer لا يهمه أن تظهر أعمدة إضافية في الـ log لكن إذا أردنا إضافتها يمكننا ذلك .

INCLUDING NEW VALUES : هل نريد اظهار الـ new record والـ old record أم فقط الـ new .

نقوم ببناء table كما يلي :

```
CREATE TABLE salesman (
sm_id    NUMBER(5)     CONSTRAINT salesman_id_pk PRIMARY KEY,
sname    VARCHAR2(40) NOT NULL,
phoneno VARCHAR2(25) NOT NULL
);
```

و نبني عليه log فيه filter columns هو sname :

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

ويسمى الـ log بشكل افتراضي mlog$_salesman وهو اسم ثابت حسب الجدول .

ويبنى جدول أخر RUDB وهو نفس الـ log لكنه يبنى عندما يكون هنالك primary key ، و نستفيد منه في عمليات الـDML التي نقوم بها عالـ MATERIALIZED VIEW وهي عملية معقدة لن ندخل فيها .

بعد ذلك نقوم بـ insert على الـ table :

```
INSERT INTO salesman VALUES (1, 'Ahmad'   , '123123123');
INSERT INTO salesman VALUES (2, 'Mohamad', '456456456');
INSERT INTO salesman VALUES (3, 'Hamdii' , '789789789');
```

و إذا ذهبنا إلى الـ log سنجد 3Rows وسنجد ترتيب العمليات (sequence) و سنجد نوع العملية هو insert والقيمتين CHANGE-VECTORS , XID$ هما binary values يخزن فيه قيم الـ Rows حيث أن الـ Row ممكن أن يكون فيه مئات الـ columns فلا يمكن وضعها كلها في الـ log فنخزنها فيهما .

نبني table آخر مرتبط بالأول ومقسم إلى عدة partitions و نبني معه log :

```
CREATE TABLE sales_range (
sm_id       NUMBER(5)    NOT NULL,
amount      NUMBER(7)    NOT NULL,
total       NUMBER(10)   NOT NULL,
sales_date  DATE         NOT NULL,
CONSTRAINT salesman_id_fk FOREIGN KEY (sm_id) REFERENCES salesman (sm_id))
PARTITION BY RANGE (sales_date)
(PARTITION sales_jan2011 VALUES LESS THAN(TO_DATE('01/02/2011','DD/MM/YYYY')),
 PARTITION sales_feb2011 VALUES LESS THAN(TO_DATE('01/03/2011','DD/MM/YYYY')),
 PARTITION sales_mar2011 VALUES LESS THAN(TO_DATE('01/04/2011','DD/MM/YYYY')),
 PARTITION sales_apr2011 VALUES LESS THAN(TO_DATE('01/05/2011','DD/MM/YYYY'))
);

CREATE MATERIALIZED VIEW LOG ON sales_range
WITH SEQUENCE, ROWID (sm_id, amount, total)
INCLUDING NEW VALUES;
```

نلاحظ أن ال table لا يوجد فيه primary key لذلك لن يبني الـ RUDB مع الـ log ، والتعليمة في الـMATERIALIZED VIEW ستكون على الـ Row id .

سيكون الـ log فارغ بداية ، فنقوم ب insert لبعض الـ rows :

```
INSERT INTO sales_range VALUES (1, 10, 10000, TO_DATE('02/01/2011', 'DD/MM/YYYY'));
INSERT INTO sales_range VALUES (1, 12, 12000, TO_DATE('03/01/2011', 'DD/MM/YYYY'));
INSERT INTO sales_range VALUES (1, 20, 20000, TO_DATE('02/02/2011', 'DD/MM/YYYY'));
INSERT INTO sales_range VALUES (1, 10, 60000, TO_DATE('01/03/2011', 'DD/MM/YYYY'));
INSERT INTO sales_range VALUES (2, 10, 10000, TO_DATE('02/02/2011', 'DD/MM/YYYY'));
INSERT INTO sales_range VALUES (2, 30, 40000, TO_DATE('01/03/2011', 'DD/MM/YYYY'));
INSERT INTO sales_range VALUES (2, 70, 20000, TO_DATE('02/03/2011', 'DD/MM/YYYY'));
INSERT INTO sales_range VALUES (3, 40, 70000, TO_DATE('02/04/2011', 'DD/MM/YYYY'));
INSERT INTO sales_range VALUES (3, 20, 50000, TO_DATE('03/04/2011', 'DD/MM/YYYY'));
```

ثم نعود للـ log ونقوم بـ Refresh فنجد فيه هذه العمليات.

الآن بعد أن بنينا 2 tables سنبني الـ MATERIALIZED VIEW :

```
CREATE MATERIALIZED VIEW sum_sales_range
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
AS
SELECT sm.sname,
COUNT(*) AS count_sname,
SUM(sr.amount) AS sum_amount_sales,
SUM(sr.total) AS sum_total_sales
FROM salesman sm, sales_range sr
WHERE sr.sm_id = sm.sm_id
GROUP BY sm.sname;
```

إذا لم نبني log وأردنا بناء MATERIALIZED VIEW من نوع Fast سيفشل ،أما إذا كان من نوع Force فسينجح حيث يختبر الـ table عندما نقوم بـ Refresh هل هنالك log مبني عليه ؟ فنقرأ من هذا الـ log وإلا نقرأ الـ table كاملا.

الـ optimizer يرى الـ MATERIALIZED VIEW كـ object من نوع MATERIALIZED VIEW و كـ table .

إذا قمنا بإظهار محتوى الـ MATERIALIZED VIEW سنجد فيه data لأننا بنيناه من نوع immediate أما الـlogs ستكون فارغة لأن الـ MATERIALIZED VIEW قام بأخذ الـ data منها.

نقوم بتعديلات على الـ tables لنرى التغير على الـ log :

```
INSERT INTO sales_range VALUES (1, 50, 50000, TO_DATE('06/01/2011', 'DD/MM/YYYY'));
```

ونذهب إلى الـ log الخاص بهذا الـ table فنجد تغير من نوع insert بينما الـ log الأول لم يجري عليه أي تغيير .

وفي الـ MATERIALIZED VIEW لم تتغير الـ data إلى أن نقوم بـ Refresh له ، فيفرغ الـ log .

وتكون عملية الـ Refresh كالتالي :

```
EXECUTE DBMS_SNAPSHOT.REFRESH( 'SUM_SALES_RANGE','f');
```

نبني MATERIALIZED VIEW آخر على نفس الـ tables ونقوم ببعض التعديلات و نلحظها في الـ log .

والان أصبح لدينا 2 MATERIALIZED VIEW يشتركان بنفس الـlogs فأذا قمنا بـ Refresh للأول فلن يفرغ الـ log والتغير الوحيد اللذي حصل هو في التاريخ و يسجل أ ن الـ MATERIALIZED VIEW الأول قام بـ Refresh .

نقوم بـ insert جديدة لن يراها كلا الـ MATERIALIZED VIEW حتى نقوم بعمل Refresh لهما .

نقوم بـ Refresh لـ MATERIALIZED VIEW الثاني عندها سيفرغ ال log ولن يبقى سوى آخر عملية insert تنتظر الأول أن يقوم بـ Refresh فيفرغ الـ Log كاملاً .

- **Data Mining:**

الأداة التي نستخدمها تدعى WEKA

وهي تقرأ files من نوع (attribute relational file format) arff وهذا الملف نكتبه بالشكل:

@RELATION super-market (اسم الملف)

@ATTRIBUTE BREAD(0,1)

MILK(0,1) } 6 attribute

.

.

@DATA

1,1,1,1,1,1 يعني انه اشترى كل المواد

1,1,1,1,0,1

نعطيه اسم الملف و الـ attributes وقيم الـ attribute (0,1) ، ونعطيه الـ data التجريبية التي سيبني الخوارزمية عليها فاذا كانت القيمة 0 لـ attribute معين هذا يعني أن ضمن هذا الـ transaction لم يحصل شراء لهذه المادة .

في الـ mining لدينا اكثر من خوارزمية: Association ,Clustering, Classify

Apriori: هي من نوع Association

عند تنفيذها تعرض معلومات استطلاعية من الملف (6 attribute ، 13 transaction،....)

نضغط start فيعطيني rule ، و الـ Apriori يتكلم عن مبدأين اساسيين هما :

Support :يأخذ احتمال وجود الـ milk, egg, bread سوية على عدد الـ transaction الخاصة بهم ، وليس لديه وثوقية كبيرة.

Confidence :وثوقية أكبر ، يأخذ احتمال milk, egg, bread على احتمال milk, egg اي احتمال طرف كامل على احتمال الطرف اليساري من الاقتضاء ، وكلما كانت قيمته عالية كلما كانت وثوقية المعلومات كبيرة .

ونلاحظ هنا الـ Confidence هي 100%

والـ Support قيمته 10% يمكننا تعديلها الى 50% مثلا ، و الـ Confidence 90% سنحولها الى 80% والـ rules هم 10 سنحولها الى 20

وننفذ ونلاحظ التغير

والـ Association هي علاقات ، مثلا علاقات بين اشياء ، مجموعات أشخاص ،....

مثال عن الـ Classify هو لعبة التنس

حيث لدينا مجموعةمتغيرات وفقها نقرر هل سنلعب أم لا

الطقس و الرطوبة والرياح

فنأخذ تقاطع هذه المتغيرات لنستنتج

ولدينا اكثر من خوارزمية في الـ Classify : 1R , Decision Tree , Bays

نختار 1R ونريد من خلالها استنتاج من هو اكثر attribute (متغير) استطيع من خلاله استنتاج علاقة صحيحة ، فبالتنفيذ اعتمد هو على الـ outlook attribute وكان احتماله10/14 وهو اكثر attribute استطاع من خلاله استنتاج نتيجة صحيحة.

1R رغم بساطتها هي من اقوى الخوارزميات وقيمتها صحيحة بنسبة 90%.

Decision Tree : نأخذ الـ attribute الذي لديه القيمة الاكبر ونضعه في الـ root .

تمّ بعونِ اللّه انتهاء كتابة مقرّر قواعد الُمعطيات المتقدمة – قسم العملي

ما كان من صواب فَمِن اللّه ... وما كان من خطأ فمِن أنفسنا

لا تنسونا من صالح الدعاء ^_^

انتهى المقرر