'''

Question 1 – Solved in Oracle

'''

CREATE OR REPLACE TRIGGER DONT\_UPDATE\_JOHN\_DOE

BEFORE UPDATE ON SCORES\_IN

FOR EACH ROW

DECLARE

J\_DOE\_ID NUMBER;

BEGIN

SELECT eid INTO J\_DOE\_ID

FROM Employees

WHERE name='John Doe';

IF ( :NEW.pid = J\_DOE\_ID )

THEN

/\* Assumes that updating a score for John Doe is an illegal transaction \*/

raise\_application\_error( -20001, 'Cannot update scores for John Doe' );

END IF;

END;

'''

Question 2 – Solved in Oracle

Answer notes:

- It would be much more efficient to implement this as a row level trigger,

but Oracle throws an error when selecting from a mutating table (which is how

we would calculate the average of affected players when rows are upadted in the

scores\_in table).

- Ideally, this derived attribute would be calculated at query time to avoid data

redundancy/maintenance performance issues.

'''

ALTER TABLE Players ADD avg\_score NUMBER;

CREATE OR REPLACE TRIGGER AVG\_SCORE

AFTER UPDATE OR INSERT OR DELETE ON Scores\_In

DECLARE

/\* Update new averages\*/

CURSOR Averages IS

SELECT Players.eid AS pid, AVG(score) AS p\_avg

FROM Scores\_In, Players

WHERE Players.eid=Scores\_In.pid

GROUP BY Players.eid;

/\* Update any players who have no scores yet \*/

CURSOR Scoreless IS

SELECT eid AS pid

FROM Players

WHERE NOT EXISTS (

SELECT null

FROM Scores\_In

WHERE Scores\_In.pid=Players.eid );

BEGIN

FOR Average IN Averages LOOP

UPDATE Players SET Players.avg\_score = Average.p\_avg WHERE Players.eid=Average.pid;

END LOOP;

FOR OmittedPlayer IN Scoreless LOOP

UPDATE Players SET Players.avg\_score=0 WHERE Players.eid=OmittedPlayer.pid;

END LOOP;

END;

'''

Question 3.a – Solved in Oracle

- Note: Assumes that Stadiums represent the location earning money from a sale.

'''

DROP TABLE Sales CASCADE CONSTRAINTS;

CREATE TABLE Sales(

sale\_id NUMBER PRIMARY KEY,

gid NUMBER NOT NULL,

amount NUMBER,

FOREIGN KEY(gid) REFERENCES Games(gid)

);

/\* LOCATION 1 \*/

DROP TABLE Sales\_Loc\_1 CASCADE CONSTRAINTS;

CREATE TABLE Sales\_Loc\_1(

gid NUMBER,

average\_sale NUMBER,

total\_sale NUMBER,

num\_transactions NUMBER

);

CREATE OR REPLACE TRIGGER UPDATE\_SALES\_LOC\_1

AFTER INSERT OR DELETE OR UPDATE ON SALES

DECLARE

CURSOR Sales\_Loc\_1\_Cursor IS

SELECT S.gid AS gid,

AVG( S.amount ) AS average\_sale,

SUM( S.amount ) AS total\_sale,

COUNT(\*) AS num\_transactions

FROM Sales S, Games

WHERE S.gid=Games.gid AND

Games.sid=1

GROUP BY S.gid;

BEGIN

FOR Sales\_Stat IN Sales\_Loc\_1\_Cursor LOOP

UPDATE Sales\_Loc\_1

SET average\_sale=Sales\_Stat.average\_sale,

total\_sale=Sales\_Stat.total\_sale,

num\_transactions=Sales\_Stat.num\_transactions

WHERE gid=Sales\_Stat.gid;

/\* If row didn’t already exist, insert it. \*/

IF ( sql%rowcount = 0 ) THEN

INSERT INTO Sales\_Loc\_1

VALUES ( Sales\_Stat.gid, Sales\_Stat.average\_sale, Sales\_Stat.total\_sale, Sales\_Stat.num\_transactions );

END IF;

END LOOP;

END;

/\* LOCATION 2 \*/

DROP TABLE Sales\_Loc\_2 CASCADE CONSTRAINTS;

CREATE TABLE Sales\_Loc\_2(

gid NUMBER,

average\_sale NUMBER,

total\_sale NUMBER,

num\_transactions NUMBER

);

CREATE OR REPLACE TRIGGER UPDATE\_SALES\_LOC\_2

AFTER INSERT OR DELETE OR UPDATE ON SALES

DECLARE

CURSOR Sales\_Loc\_2\_Cursor IS

SELECT S.gid AS gid,

AVG( S.amount ) AS average\_sale,

SUM( S.amount ) AS total\_sale,

COUNT(\*) AS num\_transactions

FROM Sales S, Games

WHERE S.gid=Games.gid AND

Games.sid=2

GROUP BY S.gid;

BEGIN

FOR Sales\_Stat IN Sales\_Loc\_2\_Cursor LOOP

UPDATE Sales\_Loc\_2

SET average\_sale=Sales\_Stat.average\_sale,

total\_sale=Sales\_Stat.total\_sale,

num\_transactions=Sales\_Stat.num\_transactions

WHERE gid=Sales\_Stat.gid;

IF ( sql%rowcount = 0 ) THEN

INSERT INTO Sales\_Loc\_2

VALUES ( Sales\_Stat.gid, Sales\_Stat.average\_sale, Sales\_Stat.total\_sale, Sales\_Stat.num\_transactions );

END IF;

END LOOP;

END;

/\* LOCATION 3 \*/

DROP TABLE Sales\_Loc\_3 CASCADE CONSTRAINTS;

CREATE TABLE Sales\_Loc\_3(

gid NUMBER,

average\_sale NUMBER,

total\_sale NUMBER,

num\_transactions NUMBER

);

CREATE OR REPLACE TRIGGER UPDATE\_SALES\_LOC\_3

AFTER INSERT OR DELETE OR UPDATE ON SALES

DECLARE

CURSOR Sales\_Loc\_3\_Cursor IS

SELECT S.gid AS gid,

AVG( S.amount ) AS average\_sale,

SUM( S.amount ) AS total\_sale,

COUNT(\*) AS num\_transactions

FROM Sales S, Games

WHERE S.gid=Games.gid AND

Games.sid=3

GROUP BY S.gid;

BEGIN

FOR Sales\_Stat IN Sales\_Loc\_3\_Cursor LOOP

UPDATE Sales\_Loc\_3

SET average\_sale=Sales\_Stat.average\_sale,

total\_sale=Sales\_Stat.total\_sale,

num\_transactions=Sales\_Stat.num\_transactions

WHERE gid=Sales\_Stat.gid;

IF ( sql%rowcount = 0 ) THEN

INSERT INTO Sales\_Loc\_3

VALUES ( Sales\_Stat.gid, Sales\_Stat.average\_sale, Sales\_Stat.total\_sale, Sales\_Stat.num\_transactions );

END IF;

END LOOP;

END;

/\* LOCATION 4 \*/

DROP TABLE Sales\_Loc\_4 CASCADE CONSTRAINTS;

CREATE TABLE Sales\_Loc\_4(

gid NUMBER,

average\_sale NUMBER,

total\_sale NUMBER,

num\_transactions NUMBER

);

CREATE OR REPLACE TRIGGER UPDATE\_SALES\_LOC\_4

AFTER INSERT OR DELETE OR UPDATE ON SALES

DECLARE

CURSOR Sales\_Loc\_4\_Cursor IS

SELECT S.gid AS gid,

AVG( S.amount ) AS average\_sale,

SUM( S.amount ) AS total\_sale,

COUNT(\*) AS num\_transactions

FROM Sales S, Games

WHERE S.gid=Games.gid AND

Games.sid=4

GROUP BY S.gid;

BEGIN

FOR Sales\_Stat IN Sales\_Loc\_4\_Cursor LOOP

UPDATE Sales\_Loc\_4

SET average\_sale=Sales\_Stat.average\_sale,

total\_sale=Sales\_Stat.total\_sale,

num\_transactions=Sales\_Stat.num\_transactions

WHERE gid=Sales\_Stat.gid;

IF ( sql%rowcount = 0 ) THEN

INSERT INTO Sales\_Loc\_4

VALUES ( Sales\_Stat.gid, Sales\_Stat.average\_sale, Sales\_Stat.total\_sale, Sales\_Stat.num\_transactions );

END IF;

END LOOP;

END;

**3.b.i) Definitions**

*Data Cube -* A set of related views summarizing aggregate information from the fact table yet using different sets of group by parameters to represent the data presented in different ways

*POS Table -* Point of sale table. In this paper, this is the fact table

*Fact Table -* The non-summarized, complete set of sales information that summarized views use to aggregate their information

*Dimension table -* Tables that store related information to the fact table. For example, information about the stores in which transactions represented by the fact table take place.

*Summary Table -* A materialized view that, for the sake of answering aggregate queries quickly, caches aggregate information about the fact table.

*Summary Delta Table -* A table that collects differed changes in a summary table as a result of a change in the fact table. These are intended to save re-computation when applying the same set of changes to a set of summary tables all aggregating similar information from the fact table.

Summary Tables must be implemented as materialized views in data-warehousing applications because if they were implemented as standard views, the entirety of the fact table would need to be queried each time an aggregate function was queried. This would defeat the efficiency that these tables are intended to provide.

**3.b.ii)**

Triggers can be used to maintain summary and summary delta tables by being defined on the fact table in such a way that when information relevant to a particular summary table is inserted into the fact table, it can also be factored into the aggregation of the summary table. This saves on expensive re-querying from the fact table and provides real time updates to the information represented in a summary or summary delta table.

Summary delta tables are used when updating a summary table so that information being aggregated into summary tables from the fact table can be reused assuming that some portion of it is relevant to more than one summary table. The other large advantage to using summary delta tables is that the work of propagating new information from the fact table can be done in a summary delta table rather than in the summary table itself preventing the summary table from being locked while the computations are taking place. This leaves summary tables available for querying while updates are being calculated.

**3.b.iii)**

A data cube lattice structure is a structure established by creating a set of materialized views that represent a permutation of a subset of the total aggregate information stored in the base table. These views represent a vector of the information stored in the base table that they are derived from and are represented by the group by clauses that group the aggregate data. Creating this lattice structure is useful because queries against specific sets of information can be answered by searching a minimal set of data while keeping the entire set of information available if need be. Using the example from the data-warehousing paper, assume that the base table holds information about storeIDs, itemIDs and dates. If we wanted to answer and aggregate question about only the dates and items being sold, omitting store information, a materialized table would be available with no more and no less information than we were looking for.