DATABASES REPORT HOMEWORK – 2

By Noman Noor | Album number: 302343 Signed on 2020-05-31

DECLARATION:

'***I certify that this assignment is entirely my own work, performed independently and without any help from sources which are not allowed.***'

(Signed by Noman Noor)

NOMAN NOOR

2020-05-12

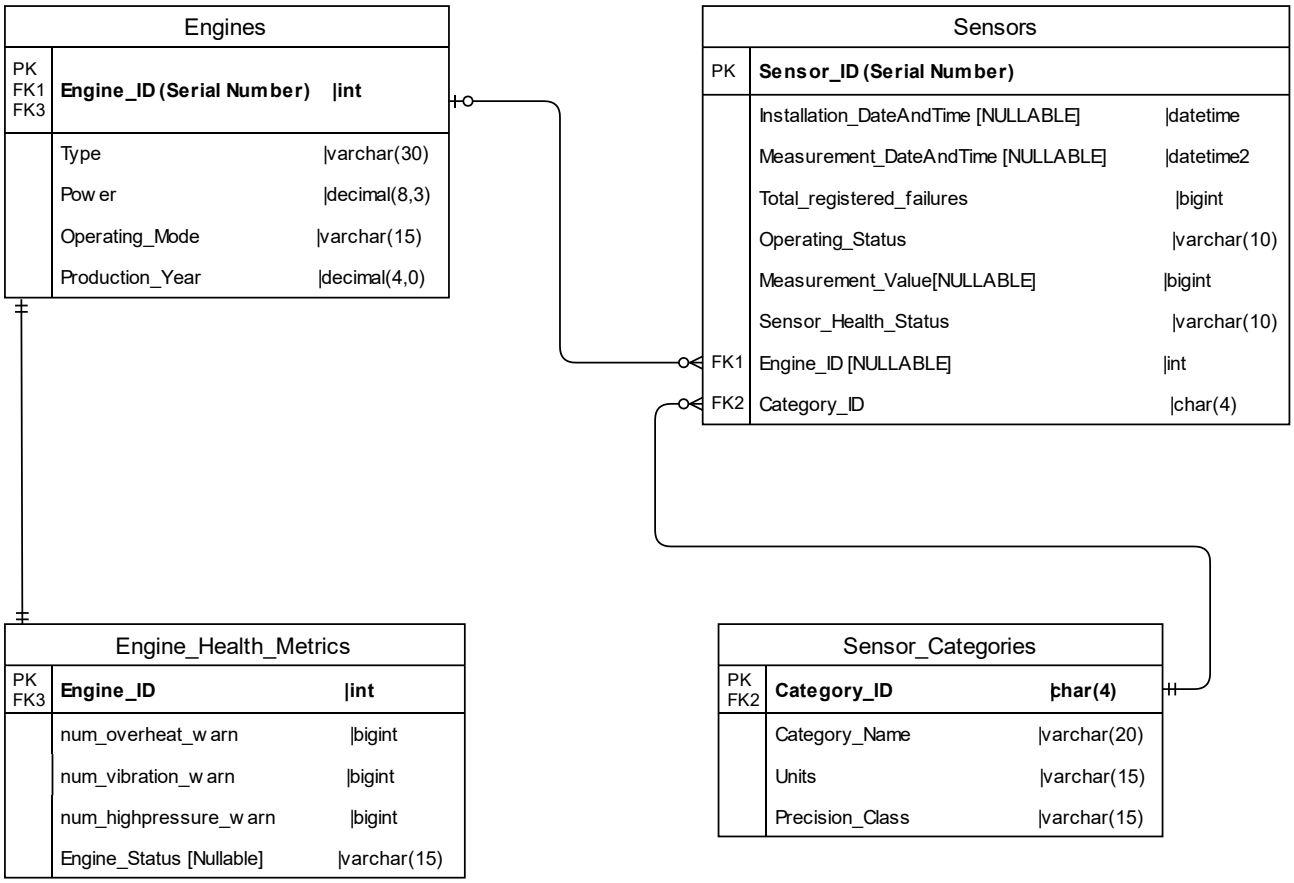
NOTE 1: All queries used here are included in the attached/mentioned .sql files.

NOTE 2: Everything in this code was done using manually typed SQL Queries (from designing DB (fk,pk,etc) to filling data). No helper utilities even from the SQL Server itself were used. Except I debugged the stored proc part by connecting the DB to Visual Studio and using it's debugging capabilities.

PART 1: Designing EngineDB:

I named the Database EngineDB and took special care of being consistent when it came to the names of rows (foreign keys especially). I also took special care when deciding on the primary keys.

The following is the Entity-Relationship Diagram of the database of Engines as I was asked to create based on the description provided:



Surprisingly, I did not need to create a table of the form table1_table2 to untangle any many-many relations as there are none, or composite primary keys, etc. However, I do know how to design more complicated Databases involving these and some other such needs.

I made Sensors' Installation and Measurement related fields nullable because a sensor may not be installed at all. However, later I realized letting Measurement_Based values be some dummy values (e.g.

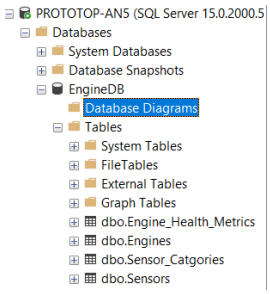
0) instead of NULL could have made the rest of the parts easier, or at times more efficient. Since I don't have contact with the client, I did the best to bypass any problems that arose by my assumptions.

On that note, it might have been better to have some extra columns or make some columns differently as I will explain in the upcoming sections wherever that change would have been most useful. Comments from my SQL files also contain these suggestions.

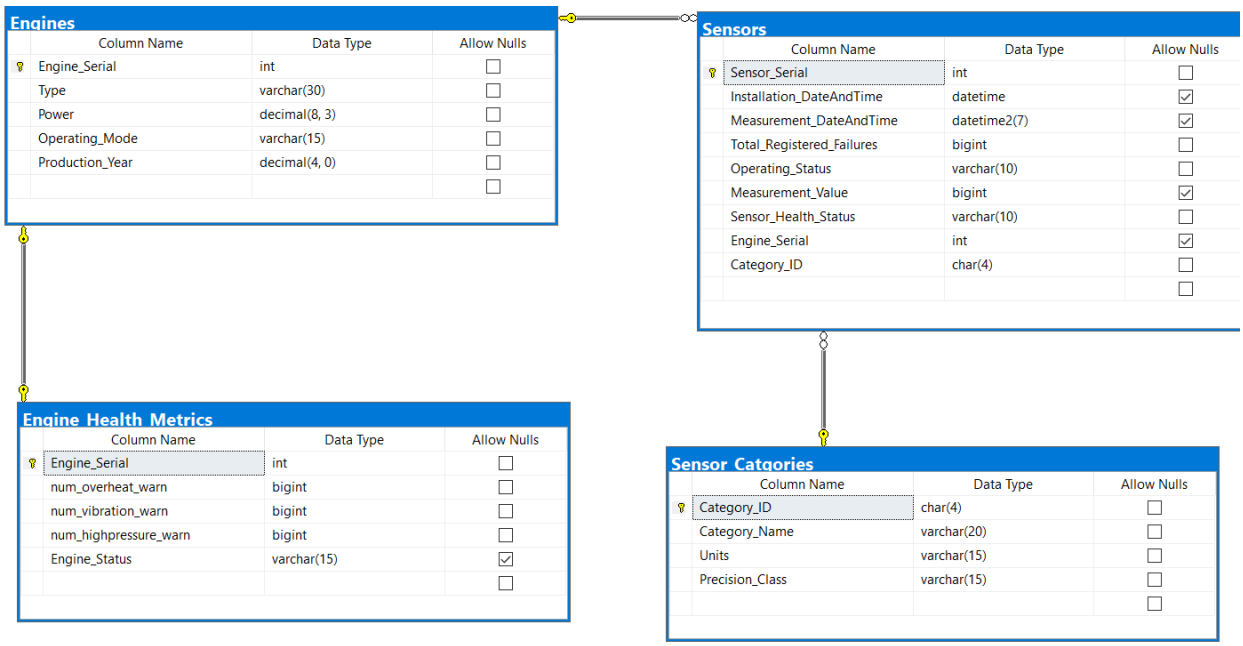
PART 2: Making and Filling EngineDB

MAKING THE DB:

Here's what the end result looks like after executing the script "[1]Create EngineDB.sql" (The comments in the script include an explanation for the program):



and



(Note:I did make the program solely using typed Queries, and this diagram was drawn by SQL Server based on my Database and I included this to show the conformity between what was planned and what I made.)

ADDING DATA:

I added 10 engines even though I was asked to add minimum 5 Engines, due to the 1-1 nature of its relationship with Engine_Health_Metrics and the fact that the rest of the tables were supposed have 10 entries. Also, I added more entries (and removed them back at times) in other parts when they were needed for testing the other parts.

Here's the final condition of each table after running the script named "[2]Data_Insertion_EngineDB.sql" I made for data insertion: (the script has been explained inside the .sql file using comments)

dbo.Engines:

	Engine_Serial	Type	Power	Operating_Mode	Production_Year
1	1	A	600.000	ACTIVE	1999
2	2	A	550.670	ACTIVE	1999
3	3	B	700.000	ACTIVE	2009
4	4	B	3000.000	ACTIVE	2019
5	5	C	10000.000	ACTIVE	2019
6	6	A	600.000	ACTIVE	2019
7	7	A	550.670	ACTIVE	2019
8	8	B	700.000	ACTIVE	2009
9	9	B	3000.000	ACTIVE	2019
10	10	C	10000.000	ACTIVE	2019

dbo.Engines_Health_Metrics:

	Engine_Serial	num_overheat_warn	num_vibration_warn	num_highpressure_warn	Engine_Status
1	1	0	0	0	GREEN
2	2	0	0	0	GREEN
3	3	0	0	0	GREEN
4	4	0	0	0	GREEN
5	5	0	0	0	GREEN
6	6	0	0	0	GREEN
7	7	0	0	0	GREEN
8	8	0	0	0	GREEN
9	9	0	0	0	GREEN
10	10	0	0	0	GREEN

dbo.Sensor_Categories:

	Category_ID	Category_Name	Units	Precision_Class
1	altd	Altitude	Metres	med
2	crnt	Current	Amperes	med
3	erpm	Engine RPM	RPM	high
4	humd	Humidity	%	low
5	intk	Air Intake	m^3	low
6	prsr	Pressure	psi	high
7	sped	Velocity	Knots/Hour	low
8	tmpr	Temperature	Celsius	high
9	vibr	Vibration Frequency	Hz	med
10	volt	Voltage	Volts	high

dbo.Sensors:

	Sensor_Seri...	Installation...	Measurem...	Total_Regis...	Operating_...	Measurem...	Sensor_He...	Engine_Seri...	Category_ID
▶	1	2020-01-12 ...	*NULL*	0	test	*NULL*	ok	1	altd
	2	2020-01-12 ...	*NULL*	0	offline	*NULL*	failure	2	volt
	3	2020-01-12 ...	*NULL*	0	online	*NULL*	ok	3	humd
	4	2020-01-01 ...	*NULL*	0	online	*NULL*	ok	4	crnt
	5	2020-01-01 ...	*NULL*	0	online	*NULL*	ok	5	erpm
	6	2020-01-01 ...	*NULL*	0	online	*NULL*	ok	6	intk
	7	2020-01-01 ...	*NULL*	0	online	*NULL*	ok	7	intk
	8	2020-01-01 ...	*NULL*	0	online	*NULL*	ok	8	sped
	9	2020-01-01 ...	*NULL*	0	online	*NULL*	ok	9	sped
	10	2020-01-01 ...	*NULL*	0	online	*NULL*	ok	10	sped
	11	2020-01-01 ...	2020-02-01 ...	100	online	120	ok	1	tmpr
	12	2020-01-01 ...	2020-01-01 ...	50	online	60	ok	1	vibr
	13	2020-01-01 ...	2020-03-01 ...	25	online	30	ok	1	prsr
	14	2020-01-01 ...	2020-02-01 ...	101	online	121	ok	2	tmpr
	15	2020-01-01 ...	2020-01-01 ...	51	online	61	ok	2	vibr
	16	2020-01-01 ...	2020-03-01 ...	26	online	31	ok	2	prsr
	17	2020-01-01 ...	2020-02-01 ...	103	online	123	ok	3	tmpr
	18	2020-01-01 ...	2020-01-01 ...	53	online	63	ok	3	vibr
	19	2020-01-01 ...	2020-03-01 ...	28	online	33	ok	3	prsr
	20	2020-01-01 ...	2020-02-01 ...	104	online	124	ok	4	tmpr
	21	2020-01-01 ...	2020-01-01 ...	54	online	64	ok	4	vibr
	22	2020-01-01 ...	2020-03-01 ...	29	online	34	ok	4	prsr
	23	2020-01-01 ...	2020-02-01 ...	105	online	125	ok	5	tmpr
	24	2020-01-01 ...	2020-01-01 ...	55	online	65	ok	5	vibr
	25	2020-01-01 ...	2020-03-01 ...	30	online	35	ok	5	prsr
	26	2020-01-01 ...	2020-02-01 ...	106	online	126	ok	6	tmpr
	27	2020-01-01 ...	2020-01-01 ...	56	online	66	ok	6	vibr
	28	2020-01-01 ...	2020-03-01 ...	31	online	36	ok	6	prsr
	29	2020-01-01 ...	2020-02-01 ...	107	online	127	ok	7	tmpr
	30	2020-01-01 ...	2020-01-01 ...	57	online	67	ok	7	vibr

	30	2020-01-01 ...	2020-01-01 ...	57	online	67	ok	7	vibr
	31	2020-01-01 ...	2020-03-01 ...	32	online	37	ok	7	prsr
	32	2020-01-01 ...	2020-02-01 ...	2108	online	2120	ok	8	tmpr
	33	2020-01-01 ...	2020-01-01 ...	258	online	260	ok	8	vibr
	34	2020-01-01 ...	2020-03-01 ...	233	online	230	ok	8	prsr
	35	2020-01-01 ...	2020-02-01 ...	10	online	20	ok	9	tmpr
	36	2020-01-01 ...	2020-01-01 ...	5	online	6	ok	9	vibr
	37	2020-01-01 ...	2020-03-01 ...	2	online	3	ok	9	prsr
	38	2020-01-01 ...	2020-02-01 ...	10	online	10	ok	10	tmpr
	39	2020-01-01 ...	2020-01-01 ...	10	online	10	ok	10	vibr
	40	2020-01-01 ...	2020-03-01 ...	10	online	10	ok	10	prsr

SQL Query TO RESET NUMBER OF WARNINGS TO 0:

(we use the script “[2_.5]ResetErrorsMetricTable_EngineDB.sql” to execute this)

A snippet of code from the above script is which achieves the required goal is:

```
SET num_highpressure_warn=0, num_overheat_warn=0, num_vibration_warn=0
WHERE num_highpressure_warn!=0 or num_overheat_warn!=0 or num_vibration_warn!=0;
```

Here, we set all the warnings to 0 in the first line for all sensors where all of them aren’t already 0.

PART 3: DESIGNING INDEX STRUCTURE

IDEOLOGY:

First of all, it would be a lot better if we could use the top down combined with bottom up approach (as taught in lectures) to find out what the users/managers actually need but I'll do my best to predict.

We note that all primary keys are already Clustered Indexes (as we didn't use the 'nonclustered' option when defining them). Therefore, we cannot create another Clustered Index as every table has PKs. It would not make sense doing it even if it did allow for such a possibility (also it’s contradictory).

Also, creating too many or using all of the remaining columns as non-clustered indexes wouldn't make sense (due to storage access issues and other performance issues due to need for rebuilding when CRUD operations are performed on the associated records especially for large databases (which is what our database mimics)).

Therefore, we look for columns that are used a lot (especially in Join clause, Where clause, or order by) which shouldn't get too many CRUD commands. Considering all the queries up to this part, it doesn't

make sense to have any indexes because the only command we have runs only on the initialization of the database. Hence only run once. However, one could predict Installation_DateAndTime, and some Foreign Keys to be good candidates. (Note that none of these get too many CRUDS). We do not use the Measurement_Value, etc. as they get a lot of updates and hence will require rebuilding index every time.

PICKING INDICES:

(We use the script "[3]Indices.sql" for executing this part. There are comments to help explain some parts of the code)

Using the Ideology given in the above sub-section:

We will set the **Installation_DateAndTime as a non-unique nonclustered index** [Since it is obviously not unique and might be mission-critical (e.g. servicing after some time (every year (by modulo 12=0), etc. and new sensors will not be added as often (assuming this from logic).]

Engine_Serial in Engine_Health_Metrics(FK3) could have been the perfect candidate for a "UNIQUE nonclustered index", but it's already a primary key of the table (hence a "UNIQUE CLUSTERED INDEX").

Apart from this, we set the all foreign keys which don't get too many CRUDS (except Engine_Serial in Engine_Health_Metrics(FK3) as it absolutely unique(exactly one-exactlyone relationship) as (NON-UNIQUE) nonclustered indices.

Looking at the next parts (4 and 5) we see that the all the indices we have already will suffice (e.g. every where clause only uses our indexes) and the indices we did create won't be rebuilt too many times because we picked them to ensure limited number of CRUDS affect the row itself.

ALSO, after re-checking the stored procedure, I realized Engines.Operating_Mode can be a good *(nonunique) nonclustered index* as Engines probably do not fail or change status as often (would need to talk to client to confirm though, but for now I'll assume this is true).

Therefore, the final indices are:

ROW NAME	TYPE OF INDEX
All Primary Keys	Unique Clustered Indices [DEFAULT]
Sensors.Engine_Serial (FK1)	Non-Clustered Indices (Non-Unique)
Sensors.Category_ID (FK2)	Non-Clustered Indices (Non-Unique)
Engines.Operating_Mode	Non-Clustered Indices (Non-Unique)

(Note: EngineHealthMetric's Engine_Status could be picked because of its mission-critical nature but it updates way too often so for now I will not do it, unless client needs something like this specifically upon being asked.)

PART 4: SQL QUERIES FOR BUISNESS PROBLEMS

(For this section we use the script "[4]SQL Queries for Business (Part4).sql". There are comments in it which explain some parts of the code.)

(NOTE: My code is very clumsy because I realized after doing this section when I had an error as I was writing this that said something about external references and realized just now, after finishing this since my mind is clear, that I could've used that in a lot of places to make it more efficient/pretty)

1: Prepare a report which will show all engines for which there was no failures ever reported by any of the sensors

```
update Sensors set Total_Registered_Failures=0 where Engine_Serial=8; --To help test if this works as my values were already 0 (I assumed this)
select * from Engines
where NOT EXISTS
(select Engine_Serial from Sensors where Sensors.Engine_Serial=Engines.Engine_Serial
and Sensors.Total_Registered_Failures>0)
--The number of errors are not nullable (0 by default) that's why we don't need to consider that. Otherwise we could have to use IS NULL also.
```

Here, we select engines for which there does not exist a corresponding Sensor where total failures are greater than 0. Note that Engines.Engine_Serial is an external reference to the initial "select * from engines where" the here the subquery works similar to "… where engines=…".

RESULT:

	Engine_Serial	Type	Power	Operating_Mode	Production_Year
1	8	B	700.000	ACTIVE	2009

2: Prepare a report where for each of the sensors category there will be shown average number of failures reported by sensors belonging to this category

```
select Sensor_Catgories.Category_Name ,
AVG(CAST(Sensors.Total_Registered_Failures AS DECIMAL(10,2))) AS 'Average Failures'
from Sensor_Catgories join Sensors
ON Sensor_Catgories.Category_ID=Sensors.Category_ID group by Category_Name
--the cast ensures that e.g.) instead of getting only 44.0 we get the accurate/real value 44.6
```

Here, we select the Category_Name and Average failures calculated in groups of category_name. The records of Sensors and Sensor_Categories are joined/linked based on same Category_ID. Basically what we get after the join is all sensors with the the details of their category next to them. Having the join enables us to use the “Sensors.Category_ID “after “ON” instead of a subquery.

RESULT:

	Category_Name	Average Failures
1	Air Intake	0.000000
2	Altitude	0.000000
3	Current	0.000000
4	Engine RPM	0.000000
5	Humidity	0.000000
6	Pressure	21.300000
7	Temperature	74.600000
8	Velocity	0.000000
9	Vibration Frequency	39.100000
10	Voltage	0.000000

3: Prepare a report which will identify an engine which sensor reported maximal total number of failures compared with total number of failures reported by individual sensors

```
select Engines.*,Sensors.Sensor_Serial,Sensors.Category_ID,Sensors.Measurement_Value
from Engines right join Sensors on Engines.Engine_Serial IN
(select TOP(1) Engines.Engine_Serial as EngSer from Engines,Sensors
where Sensors.Engine_Serial=Engines.Engine_Serial
group by Engines.Engine_Serial
order by sum(Sensors.Total_Registered_Failures) desc)
where Engines.Engine_Serial=Sensors.Engine_Serial
```

(Note that we could probably select all the engines with the max value if inside of top we added a subquery which returned the same number of engines with the maximum sum of failures with sliight modifications, but that is not what has been asked. The current scenario said select “an engine” so I thought anyone that matched would work.)

Here, we select first row of the engine and corresponding sensor (by joining them) which is sorted in descending order based on sum of total registered failures from Sensors calculated in groups of each Engines.Engine_Serial.

RESULT:

	Engine_Serial	Type	Power	Operating_Mode	Production_Year	Sensor_Serial	Category_ID	Measurement_Value
1	7	A	550.670	ACTIVE	2019	7	intk	NULL
2	7	A	550.670	ACTIVE	2019	29	tmpr	127
3	7	A	550.670	ACTIVE	2019	30	vibr	67
4	7	A	550.670	ACTIVE	2019	31	prsr	37

4: Prepare a report showing engines with average vibration readings significantly above (50% greater) compared to the average vibration readings

```
select * from Engines RIGHT JOIN
(select Sensors.Engine_Serial,AVG(Sensors.Measurement_Value) as avgmv
from Sensors GROUP BY Sensors.Engine_Serial,Category_ID HAVING
(avg(Sensors.Measurement_Value)>(select 1.5*AVG(Sensors.Measurement_Value) from Sensors where Sensors.Category_ID='vibr')) --the first Sensors.Measurement value
and                                                                                    -- is an external reference the second one is not due to the from in the subquery
Sensors.Category_ID='vibr') AS Sens  --Also, Alias was neccessary (AS Sens, it wouldn't work until then) maybe because of ambiguity with the external reference
ON Engines.Engine_Serial=Sens.Engine_Serial;
```

Here, the main part is that the second occurrence of avg() gives average of measurement values calculated in groups of engine_serial and Category_ID (where category_ID =0) [category_id is only present in group by because of the existence of aggregate and since I was also selecting Category_ID in the beginning], while the third occurrence of avg() in the subquery gives the average calculated for all sensors with the id vibr.

RESULT:

	Engine_Serial	Type	Power	Operating_Mode	Production_Year	Engine_Serial	avgmv
1	8	B	700.000	ACTIVE	2009	8	260

5: Identify an engine for which maximum readings of any sensor has never exceeded average reading for this sensor m on how many sensors an engine may have)

```
--the following value is inserted to help test the function
insert Sensors values
(100, CAST('2020-01-01T00:00:00.000' AS DateTime), CAST('2020-02-01T00:00:00.0000000' AS DateTime2), 100000000000, 'online', 12000000, 'ok', 1, 'tmpr')
--actual solution begins
Select Engines.*
from Engines,
(Select  tab1.Engine_Serial, count(tab1.Category_ID) as countcid from
Engines INNER JOIN
(select tb1.Engine_Serial,tb1.Category_ID from
(select Sensors.Engine_Serial, Sensors.Category_ID, MAX(Sensors.Measurement_Value) as maxmv from Sensors group by Sensors.Engine_Serial,Sensors.Category_ID) as tb1
INNER JOIN
(select Sensors.Category_ID, AVG(Sensors.Measurement_Value) as avgmvall from Sensors group by Sensors.Category_ID) as tb2
ON
(tb2.Category_ID=tb1.Category_ID and maxmv<avgmvall)) as tab1
ON(tab1.Engine_Serial=Engines.Engine_Serial)
group by tab1.Engine_Serial
having count(tab1.Category_ID)>=3) AS Table1
where(Engines.Engine_Serial=Table1.Engine_Serial)
--deleting the value previously inserted to test this
delete Sensors where Sensors.Sensor_Serial=100;
```

The part around the second inner join creates a table with details of an engine with the sensor type and its overall average value where the average of maximum value of each type of sensor for the engine is less than the average value of that specific type of sensor. The Table1 gotten by next join gets each engine and the number of times (using count() grouped by each engine) another category (corresponding to it) is in the list but only when the number of times is greater or equal (should be equal, but just in case I added greater or equal) to the number of sensor categories. Since here we only have 3 num_of_warnings columns, I set it to 3 (described as m in the question)

After this, Engines.* from Engines are selected where Engines.Engine_Serial=(table3).Engine_Serial. (This line is just a brief/layman's explanation.)

RESULT:

	Engine_Serial	Type	Power	Operating_Mode	Production_Year
1	2	A	550.670	ACTIVE	1999
2	3	B	700.000	ACTIVE	2009
3	4	B	3000.000	ACTIVE	2019
4	5	C	10000.000	ACTIVE	2019
5	6	A	600.000	ACTIVE	2019
6	7	A	550.670	ACTIVE	2019
7	9	B	3000.000	ACTIVE	2019
8	10	C	10000.000	ACTIVE	2019

PART 4: SQL QUERIES FOR BUISNESS PROBLEMS

(This part is achieved by executing "[5]FINAL Part5 SingleEngine CREATE STORED PROC.sql" (it is well documented with comments for explanations***) and "[6]FINAL Part5 MultipleEngine CREATE STORED PROC.sql"*** (it is self-explanatory)***)***

Check the above scripts for more information, as I'll only include essential snippets from them and explain their meaning.

Firstly, a few comments from my program that are necessary:

```
--I understood this pretty late that I had to iterate through all engines in this
procedure, so I just added another procedure which iterates through this procedure for
each engine. It seems just as efficient if not better and has the same functionality.
Apart from that it also gives you the option to monitor only specific engines, and is
simpler to modify/understand/edit/create.
--I also think it might have been impossible to achieve some things requested due to
inconsistencies if I had not done this (because returning when an engine was in service
would quit the whole procedure).
```

So, my solution was (explained in the comment above):

```
begin
    set nocount on


    DECLARE @Engine_Serial int
    DECLARE Engine CURSOR LOCAL FOR SELECT Engine_Serial FROM dbo.Engines
    OPEN Engine
    FETCH NEXT FROM Engine INTO @Engine_Serial
    WHILE @@FETCH_STATUS=0
    BEGIN
        exec dbo.spEnginesHealthMetrics_Update @Engine_Serial
        FETCH NEXT FROM Engine INTO @Engine_Serial
    END
    CLOSE Engine
    DEALLOCATE Engine
end
```

```
--I was also short on time since I received the project a little late and had a busy week
with classes,tests.

--EDIT: HAD TO CHANGE/DIVERT FROM THE DOCX HOW I DID THE FOLLOWING PART BECAUSE PROVIDED
DATABASE STRUCTURE ONLY ASKED FOR COLUMNS for overheat,vibration,highpressure warnings.
```

Basically, I had to make a counter/variable which would handle the number of errors from all other miscellaneous sensors combined, which would then be used in deciding the status.

EXPLANATIONS (of some essential parts):

```
IF  ((SELECT Operating_Mode FROM dbo.Engines WHERE Engines.Engine_Serial=@Engine_Serial)='STOP' or (SELECT Operating_Mode FROM dbo.Engines WHERE Engines.Engine_Serial=@Engine_Serial)='SERVICE')
BEGIN
    RETURN --I also think it might have been impossible to achieve some things requested due to inconsistencies if I had not done this in a procedure-inside-procedure fashion (because returning when an
END
```

The IF statement ensures that if the Engine has Operating_Mode as SERVICE or STOP, it terminates the procedure.

Then we loop/iterate through each sensor and detect if it is having a failure.

If yes, then:

```
IF((SELECT Sensor_Health_Status FROM Sensors WHERE Sensors.Sensor_Serial=@Sensor_Serial)='FAILURE')
BEGIN
    IF((DATEDIFF(HOUR,(SELECT Measurement_DateAndTime FROM Sensors WHERE Sensors.Sensor_Serial=@Sensor_Serial),GETUTCDATE()))<=2)       --
        BEGIN
            SET @Recent_Failure_Count = @Recent_Failure_Count+1
            UPDATE dbo.Sensors SET Total_Registered_Failures = Total_Registered_Failures+1 WHERE dbo.Sensors.Sensor_Serial=@Sensor_Serial
        END
```

The above increases total registered failure rate for a sensor and sets a recent failure count which is then later used in a way that if this is >=2 at the end of the loop of each sensor (but not the engine), we set engine status as amber.

```
IF((DATEDIFF(SECOND,(SELECT Measurement_DateAndTime FROM Sensors WHERE Sensors.Sensor_Serial=@Sensor_Serial),GETUTCDATE()))<=60 and (((SELECT Measurement_Value FROM Sensors WHERE Sensors.Sensor_Serial=@Sensor_Serial)
>2.2*(SELECT AVG(Measurement_Value) FROM Sensors GROUP BY Sensors.Category_ID having Sensors.Category_ID=(SELECT Category_ID FROM Sensors WHERE Sensor_Serial=@Sensor_Serial))) OR
(NOT EXISTS (SELECT Sensor_Serial FROM SENSORS WHERE (Category_ID IN (SELECT Category_ID FROM Sensors WHERE (Sensor_Serial=@Sensor_Serial)) AND (Sensor_Serial!=@Sensor_Serial) AND (Measurement_Value IS NOT NULL))))))
BEGIN
    IF((SELECT Sensors.Category_ID FROM Sensors WHERE Sensors.Sensor_Serial=@Sensor_Serial)='tmpr')        --Put tmpr first because I'm assuming overheating would be a more common problem therfore resources saved when t
    BEGIN
        IF(@HasWarningTempr =0) BEGIN SET  @HasWarningTempr=1 END
        UPDATE dbo.Engine_Health_Metrics SET num_overheat_warn = num_overheat_warn+1 WHERE Engine_Health_Metrics.Engine_Serial=@Engine_Serial
    END
    ELSE IF((SELECT Sensors.Category_ID FROM Sensors WHERE Sensors.Sensor_Serial=@Sensor_Serial)='prsr')
    BEGIN
        IF(@HasWarningPrsr =0) BEGIN SET  @HasWarningPrsr=1 END
        UPDATE dbo.Engine_Health_Metrics SET num_highpressure_warn = num_highpressure_warn+1 WHERE Engine_Health_Metrics.Engine_Serial=@Engine_Serial
    END
    ELSE IF((SELECT Sensors.Category_ID FROM Sensors WHERE Sensors.Sensor_Serial=@Sensor_Serial)='vibr')
        BEGIN
        IF(@HasWarningVibr=0) BEGIN SET  @HasWarningVibr=1 END
        UPDATE dbo.Engine_Health_Metrics SET num_vibration_warn = num_vibration_warn+1 WHERE Engine_Health_Metrics.Engine_Serial=@Engine_Serial
        END
    Else
        BEGIN SET @MiscSensor_New_Warning_Count = @MiscSensor_New_Warning_Count +1 SET @HasWarningMisc=1 END    --The reason I use IFs above is because there will be lots of times when the above categories will be update
END
```

The above part finds a sensors whose have never reported any failure in the past but reported one within last 60 seconds measurements above 220% of average readings for the corresponding sensors categories and then the procedure updates the metrics table record of the corresponding engine accordingly to increase the number of warnings value for the corresponding sensor category.

WE REACH THE END OF WHILE LOOP, after that:

```
--EDIT: HAD TO CHANGE/DIVERT FROM THE DOCX HOW I DID THE FOLLOWING PART BECAUSE PROVIDED DATABASE STRUCTURE ONLY ASKED FOR COLUMNS for overheat,vibration,highpressure warnings.
    --The next two lines make sure that emergency/warning parts can be reached even if the engine doesn't have any specific misc sensors. Though this will sacrifice the AMBER mode for those engines
    --This is by design as it is better to have just a red satus than just an amber in such mission critical-components
    --This should also serve as a reminder to install misc sensors as required, unlesss they don't need it. In such a case, a dummy sensor should be created and set to bypass this issue. This actually makes it a little safer.
IF(not exists(SELECT Category_ID FROM Sensors WHERE Category_ID NOT IN ('tmpr','prsr','vibr') and Engine_Serial=@Engine_Serial)) --Turns out you cant have the NOT IN part in brackets like I have the rest of conditions everywhere else in this program
    BEGIN SET @HasWarningMisc=1 SET @MiscSensor_New_Warning_Count=3 END
IF(3>=((SELECT num_overheat_warn FROM dbo.Engine_Health_Metrics WHERE Engine_Health_Metrics.Engine_Serial=@Engine_Serial)+(SELECT num_highpressure_warn FROM dbo.Engine_Health_Metrics WHERE Engine_Health_Metrics.Engine_Serial=@Engine_Serial)+(SELECT num
    BEGIN UPDATE dbo.Engine_Health_Metrics SET Engine_Status = N'GREEN' WHERE dbo.Engine_Health_Metrics.Engine_Serial=@Engine_Serial END

IF((((SELECT num_overheat_warn FROM dbo.Engine_Health_Metrics WHERE Engine_Health_Metrics.Engine_Serial=@Engine_Serial)+(SELECT num_highpressure_warn FROM dbo.Engine_Health_Metrics WHERE Engine_Health_Metrics.Engine_Serial=@Engine_Serial)+(SELECT num_v
    BEGIN UPDATE dbo.Engine_Health_Metrics SET Engine_Status = N'AMBER' WHERE dbo.Engine_Health_Metrics.Engine_Serial=@Engine_Serial END
IF((4=(@HasWarningTempr+@HasWarningPrsr+@HasWarningVibr+@HasWarningMisc) and ((3>(SELECT num_overheat_warn FROM dbo.Engine_Health_Metrics WHERE Engine_Health_Metrics.Engine_Serial=@Engine_Serial)) or (3>(SELECT num_highpressure_warn FROM dbo.Engine_Hea
BEGIN
    UPDATE dbo.Engine_Health_Metrics SET Engine_Status = N'RED' WHERE dbo.Engine_Health_Metrics.Engine_Serial=@Engine_Serial
    UPDATE dbo.Engines SET Operating_Mode = N'WARNING' WHERE dbo.Engines.Engine_Serial=@Engine_Serial
END
IF((4=(@HasWarningTempr+@HasWarningPrsr+@HasWarningVibr+@HasWarningMisc) and (not ((3>(SELECT num_overheat_warn FROM dbo.Engine_Health_Metrics WHERE Engine_Health_Metrics.Engine_Serial=@Engine_Serial)) or (3>(SELECT num_highpressure_warn FROM dbo.Engin
BEGIN
    UPDATE dbo.Engine_Health_Metrics SET Engine_Status = N'RED' WHERE dbo.Engine_Health_Metrics.Engine_Serial=@Engine_Serial
    UPDATE dbo.Engines SET Operating_Mode = N'EMEREGENCY' WHERE dbo.Engines.Engine_Serial=@Engine_Serial
END
```

[Note that the above part cannot be screenshotted fully (due to the long conditions which I should have organized better), therefore it might prove necessary to refer to the "FINAL Part5 SingleEngine CREATE STORED PROC.sql" script file to check the underlying logic.]

After processing the all the sensors , the above part sets

- metrics health to GREEN (if there is no more than 3 warning in total)
- metrics health to AMBER (if there are 3-5 warnings in at most 3 different categories)
- metrics health to RED and engine operating status to WARNING (if ther0065 are warnings in all categories)
- metrics health to RED and engine operating status to EMERGENCY (if there are at least 3 warnings in each category)