So this came as a surprise, when working on calculating simple statistics on my dataset, in particular min, max and median. First two are trivial. The last one was the one, that caught my attention.

While finding the fastest way on calculating the median (statistic: [median](https://en.wikipedia.org/wiki/Median)) for given dataset, I have stumbled upon an interesting thing.  While WINDOW function was performing super slow and calling R or Python using sp\_execute\_xternal\_script outperform window function as well, it raised couple of questions.

But first, I created a sample table and populate it sample rows:

DROP TABLE IF EXISTS t1;

GO

CREATE TABLE t1

(id INT IDENTITY(1,1) NOT NULL

,c1 INT

,c2 SMALLINT

,t VARCHAR(10)

)

SET NOCOUNT ON;

INSERT INTO t1 (c1,c2,t)

SELECT

x.\* FROM

(

SELECT

ABS(CAST(NEWID() AS BINARY(6)) %1000) AS c1

,ABS(CAST(NEWID() AS BINARY(6)) %1000) AS c2

,'text' AS t

) AS x

CROSS JOIN (SELECT number FROM master..spt\_values) AS n

CROSS JOIN (SELECT number FROM master..spt\_values) AS n2

GO 2

Query generated – in my case – little over 13 million records, just enough to test the performance.

So starting with calculating Median, but sorting first half and second half of rows respectively, the calculation time was surprisingly long:

-- Itzik Solution

SELECT (

(SELECT MAX(c1) FROM

(SELECT TOP 50 PERCENT c1 FROM t1 ORDER BY c1) AS BottomHalf)

+

(SELECT MIN(c1) FROM

(SELECT TOP 50 PERCENT c1 FROM t1 ORDER BY c1 DESC) AS TopHalf)

) / 2 AS Median

Before and after each run, I cleaned the stored execution plan. The execution on 13 million rows took – on my laptop – around 45 seconds.

Next query, for median calculation was a window function query.

SELECT DISTINCT

PERCENTILE\_CONT(0.5) WITHIN GROUP (ORDER BY c1)

OVER (PARTITION BY (SELECT 1)) AS MedianCont

FROM t1

To my surprise, the performance was even worse, and at this time, I have to say, I was running this on SQL Server 2017 with CU7. But luckily, I had a [SQL Server 2019 CTP 2.0](https://docs.microsoft.com/en-us/sql/sql-server/sql-server-ver15-release-notes?view=sqlallproducts-allversions) also installed and here, with no further optimization the query ran little over 1 second.

So the difference between the versions was enormous. I could replicate the same results by switching the database compatibility level from 140 to 150, respectively.

ALTER DATABASE SQLRPY

SET COMPATIBILITY\_LEVEL = 140;

GO

SELECT DISTINCT

PERCENTILE\_CONT(0.5) WITHIN GROUP (ORDER BY c1)

OVER (PARTITION BY (SELECT 1)) AS MedianCont140

FROM t1

ALTER DATABASE SQLRPY

SET COMPATIBILITY\_LEVEL = 150;

GO

SELECT DISTINCT

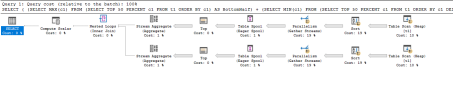
PERCENTILE\_CONT(0.5) WITHIN GROUP (ORDER BY c1)

OVER (PARTITION BY (SELECT 1)) AS MedianCont150

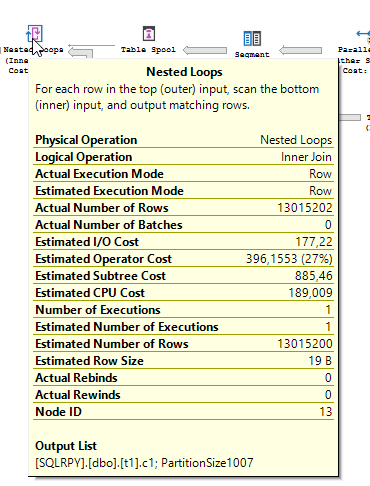
FROM t1

The answer was found in execution plan. When running window function under 140 compatibility level, execution plan decides to create nested loop two times, for both groups of upper and lower 50% of the dataset.

This plan is is somehow similar to understanding of 50% of upper and lower dataset but with only one nested loop:



Difference is that when running the window function calculation of median on SQL Server version 2017, the query optimizer decides to take row execution mode for built-in [window function with WITHIN GROUP.](https://docs.microsoft.com/en-us/sql/t-sql/functions/percentile-cont-transact-sql?view=sql-server-2017)

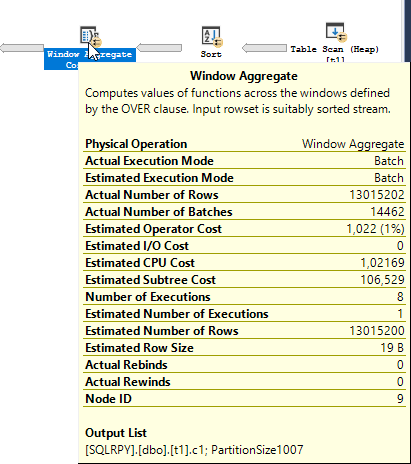


This was, as far as I knew, not an issue since SQL Server 2016, where batch mode operator for window aggregation was already used.

When switching to compatibility level 150 and running the same window function, the execution plan is, as expected:

comp150_1

And window aggregate uses batch mode:



When calculating Median using R:

sp\_Execute\_External\_Script

@language = N'R'

,@script = N'd <- InputDataSet

OutputDataSet <- data.frame(median(d$c1))'

,@input\_data\_1 = N'select c1 from t1'

WITH RESULT SETS (( Median\_R VARCHAR(100) ));

GO

or Python:

sp\_Execute\_External\_Script

@language = N'Python'

,@script = N'

import pandas as pd

dd = pd.DataFrame(data=InputDataSet)

os2 = dd.median()[0]

OutputDataSet = pd.DataFrame({''a'':os2}, index=[0])'

,@input\_data\_1 = N'select c1 from t1'

WITH RESULT SETS (( MEdian\_Python VARCHAR(100) ));

GO

both are executing and returning the results in about 5 seconds. So no bigger difference between R and Python when handling 13 million rows for calculating simple statistics.

To wrap up, If you find yourself in situation, where you need to calculate – as in my case – Median or any statistics, using window function within group, R or Python would be the fastest solutions, following T-SQL. Unless, you have the ability to use SQL Server 2019, T-SQL is your best choice.

Code and the plans, used in this blog post are available, as always at [Github](https://github.com/tomaztk/Window_aggregate_operator).