DataWarehouse Ingestion

## CONTEXT

Relational databases like Microsoft’s SQL Server Management Studio (SSMS) have a lot of potential for performing analytics on gridded climate data. Gridded climate data is commonly stored and distributed through highly compressed NetCDF (.nc) files that are unable to be read directly by SSMS. Therefore an ingestion process is needed to handle the extraction, transformation and loading of the gridded climate data into a relational database from a NetCDF file. This document outlines some lessons learned

This project has used the Bureau of Meteorology’s AWRA-L annual daily hydrology dataset to test the feasibility of using relational databases for gridded climate data. The data contained within these files exists as a three-dimensional array with positions corresponding to latitude, longitude, and date. Where possible, the ingestion process has been designed to be widely applicable for all gridded climate datasets. There are however some aspects relating to NetCDF structure and naming conventions that are specific to this dataset.

* Using np.flatten(), np.tile(), np.repeat()
* Df.to\_csv with pandas
* Data size constraints for pandas and csv editing

## ISSUES AND SOLUTIONS

The simplest version of the ingestion process would involve opening and extracting the NetCDF data into a tabular form (shown below) through some intermediate program, and then pushing it directly into an SSMS table – and this is what was originally attempted. Unfortunately, limitations in computational resources and software functionality meant that this approach would not be feasible.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **DateTime** | **Latitude** | **Longitude** | **Variable 1** | **Variable 2** | **…** | **Variable N** |
| 01/01/1911 | -10 | 112 | - | - | … | - |
| 01/01/1911 | -10 | 112.05 | - | - | … | - |
| ⁞ | ⁞ | ⁞ | ⁞ | ⁞ | ⁞ | ⁞ |
| 31/12/2015 | -44 | 153.95 | - | - | … | - |
| 31/12/2015 | -44 | 154 | - | - | … | - |

The majority of complications in this process are directly related to the shear size of this dataset

* Chose python
* The simplest version of this would be: (SQL server is unable to read netCDF)
* Read .nc into an array/table Datetime, lat, lon, variables in python and then push directly into SQL server (illustrate)
* Issues with this include:

### PRESERVING ORDER THROUGHOUT INGESTIONS

Tables in SSMS typically exist as unordered heaps of rows, unless they have an index. For the query performance of this datawarehouse it is important the rows are ordered by date and location. In a rowstore table this can be achieved by adding an index. Columnstore indexes however, work quite differently; in order to optimise query performance with the automatic segment elimination, row segments are required to be inserted and written to disk in order of date and location.

There are two main python modules that support pushing data to an SSMS table, **pyodbc** and **sqlalchemy**.

* + Sql alchemy and pyodbc modules both have the capability to push tabular data from python directly into a SSMS table, however neither are suitable for pushing into a columnstore table because order is not maintained in the process. Further, they also weren’t equipped to handle bulk loads, converging at around 50 minutes to push a single variable’s annual data into sql server. Even after the exploration of fastexecutemany settings and other tips to accelerate this
  + There is a workaround to the issue of ordering data for a columnstore table with these modules. It involves first loading the data as a rowstore heap, then adding and index to order it into a table and then swapping the index to a columnstore one. The issue here it that you have to switch off SSMS’s natural CPU parallelisation to ensure that the order is preserved – meaning that this process very costly for time.
  + In light of this issue, the most workable solution found was use a CSV file intermediate. This also enables the use of the BULK INSERT function which is significantly faster than at inserting data into a table than other methods. To achieve this requires the table to be defined with an additional unique identity column. Then a VIEW is created on the table covering all columns except for the identity column. When data is then bulk inserted into the VIEW, the identity column in the original table automatically updates and enforces an order of rows. Code example below:

CREATE TABLE [DW3].[OBT](

[ID] BIGINT IDENTITY(1,1), --

[DateTime] SMALLDATETIME,

[Latitude] DECIMAL(5,2),

[Longitude] DECIMAL(5,2),

[E0] FLOAT,

GO

CREATE VIEW DW3\_OBT ([DateTime], [Latitude], [Longitude], [E0]) AS

SELECT [DateTime], [Latitude], [Longitude], [E0] FROM [DW3].[OBT];

GO

BULK INSERT [DW3\_OBT]

FROM ''' + @FilePath + @FileName + '''

WITH(

FIRSTROW = 2,

FIELDTERMINATOR = '','',

ROWTERMINATOR = ''\n'',

KEEPNULLS

)'

* + Create view and bulk insert method was found to be the most practical
  + Explain the work around
* Data Size –
  + Managing datatypes can also help in SQL, but doesn’t offer major advantage in python of CSV steps
* RunTime
  + Efficiency is improved significantly by using the simpler functions in the numpy library
  + Processing the data in python as a numpy array is way faster
  + Less than 10 seconds using flatten, compared to up to 10 minutes using list iteration.
* Transaction log and tempdb
  + INSERTS will quickly fill up the transaction log and, with this volume of data are likely to
* NULL values
  + NULL values in CSV are written as “”
  + Data Size constraints in CSV editing approx. 10-15GB seems to be pushing limits on what can be processed in python
* Control

## PROCESS

## GENERAL RECOMMENDATIONS

* Iterate through small and manageable batches, clearing the transaction log after each
* Avoid having to do assembly/joins within SQL Server as these may cause tempdb to fill up and that is harder to clear
* Ensuring that data is inserted orderly into a columnstore table is important for segment elimination and optimisation. There are two work arounds.
* DateTime format vs split integer
* ETL vs ELT
  + For managing the transaction log issues I recommend ETL
  + Doesn’t necessarily save time, but it is much simpler inside SQL Server to just use a single BULK INSERT than to be worrying about having to join together tables and using staging tables.
  + It saves some complexity, but doesn’t really save time.
* Different table structures
* More control pythons running the SQL Load/Transform from python
* Unexplored options:
  + C-based language
  + BCP utility for SQL server