Factors that influence SNAP performance

Ingestion Performance

SNAP ingestion consists of two parts.

1. - Flattening the star schema (optimally when there is a star schema)
2. - Building an index

Design factors

- Qube design
  - column cardinality
    - High number of unique values in a column increases the time to build the index
    - When creating the star schema, create with stats and analyze the various statistics of the columns to determine potential impact.
  - Number of dimension columns
    - Time to index 100 columns is lesser than 1000 columns for example .
    - The time taken to build the index is also determined by the number of dimensions.
  - Star schema joins and flattening
    - The number of tables in the star schema influences the time to flatten prior to building the index. SNAP flattens the star schema and then builds an index for the dimensions. When there are more tables there is more time spent to flatten
    - The skewness of data in the join graph can influence the time to flatten.
    - Shuffle tuning is critical since the flattening is executed as a Spark job.
  - Partitioning strategy
    - Building a SNAP index on 100Gb non-paritioned table can have different resource requirements than the same table partitioned by say YEAR or other columns. SNAP ingestion can be done per partition and hence the performance of ingestion can be significantly improved by partitioning.
  - Settings on the Qube
    - SNAP prefers its segments(files that make up the index) to be around 250MB for normal sized Qubes(100-200 columns) or < 5million rows per file.
    - The setting “preferredSegmentSize”, when defining the OLAP Index, determines the size of the segment.
    - The setting “avgSizePerPartition” is a rough estimate of the uncompressed size of the SNAP partition upon indexing and is a function of the original data being ingested in SNAP.
      - For example if your original dataset is csv and you are ingesting 100GB of a partition, then avgSizePerPartition=20g based on our experience - not all datasets are the same and so this can vary but we can expect 70-80% compression.
      - If your dataset is Orc or Parquet, SNAP can be 1-1.5 times the partition size.
      - The ratio of avgSizePerPartition and preferredSegmentSize determines the number of physical SNAP files created per partition for the index.
      - Example indexing a 100GB csv partition
      - avgSizePerPartition=20g
      - preferredSegmentSize=250mb
      - Number of tasks running indexing is 20g/250mb= 80 and 80 files will be produced.

      - If the avgSizePerPartition=2g (by mistake for example) then 8 tasks will index the data and 8 files totalling 20Gb will be produced. This can cause the indexing to be much slower since each task is carrying a much heavier load due to reduction in parallelism
      - If the avgSizePerPartition=200g (again by mistake) then 800 tasks will be spawned and 800 files each of 25mb approximately will be produced. This is again very inefficient for SNAP since small files impact query performance.
      - The Spark UI will have information on number of tasks, size being processed etc while the indexing is running
      - To come up with the most efficient avgSizePerPartition it is best to index the data on a TABLE SAMPLE and estimate the size based on the sample index before doing it for all the data.

Run time

- Parallelism of the jobs.
  - Flattening the data involves creating multiple tasks to create files that are used as input into the indexing stage. At this stage, depending on the dataset being ingested, care must be taken to set spark.sql.shuffle.partitions appropriately so that few tasks don't end up processing all the data. Typically a task should be processing 250-500MB of data.
- Broadcast of small tables

  At the time of ingestion, large tables can be joined with smaller tables. In these cases spark.sql.autoBroadcastJoinThreshold can be set to a higher value to make use of broadcast joins and avoid sort merge on large clusters
- Off-heap memory

  The indexing process uses off-heap. Depending on the index size the off-heap has to be adjusted. Some times this can be 4GB to 6GB per core depending on the size of the partition being indexed.

Query Performance

- Number of cores available in the cluster
  - For a given index the number of waves determines the run time query performance
    - Example: An index has 1000 segments. The cluster has 100 cores. This means a query that executes on all segments(partition filters can reduce the number of segments scanned) will have 1000/100 = 10 Waves. If the median query segment time is 100ms then total query time will be approximately 100ms*10 = 1s.
      - The same query, when it has 200 cores will execute in 5 waves. So total time will be 0.5s.
- Per segment time
  - For a given query one can see the Task time(which is the time to query a given segment) on the Spark UI. The segment times should be ideally < 0.5s of the cube is properly designed and segments are memory mapped. Segment times can be high when there more than 5 million rows per segment.
    - Segment times can be large when
      - The segment has to be pulled from deep storage to local cache
      - The segment has to be read from disk

- Shuffle
  - SNAP queries have multiple stages - a Stage that queries the segment and stages that are executed in Spark.
    - The setting spark.sql.shuffle.partitions at query time should be set to 20 for most adHoc query usecases. (note when indexing this may not be sufficient and so this should be a session level query time setting)
- Executor memory
  - The executor memory is the Heap memory available to the Spark jobs. Typically for queries we set this as 2Gb per core.
- Off-heap memory
  - For some operations the off-heap is used. The off heap is a function of another setting + a constant. It is typically set atleast to number of cores per executor * the value below.
    - spark.sparklinedata.spmd.gByEngine.offheapsize=128mb
  - For example if the number of cores per executor is 4 then off-heap should be atleast 512mb(4* 128) and should be set to 1g to accomodate other spark use of the off-heap.
  - Off-heap memory is set as follows in the properties file - the setting MaxDirectMemorySize is the off-heap.

    spark.executor.extraJavaOptions= -XX:MaxDirectMemorySize=2g -XX:+UseG1GC -XX:MaxGCPauseMillis=100 -XX:OnOutOfMemoryError='kill -9 %p'
- Auto broadcast joins
  - For query, turn off auto broadcast. The driver can become a bottleneck in doing broadcast joins and can impact performance

Infrastucture

IO/Disk

- SNAP depends on Fast IO and memory mapping. SNAP downloads its segments (files) to a local cache. This local cache
  - should be across multiple disks
    - Should be SSDs
      - Should **not** be on a disk where other apps have a lot of IO (/tmp)

Nodes

Total RAM

HEAP	OFF-HEAP	SPACE FOR SEGMENTS IN MEMORY

The space for segments to be loaded in memory is typically SHARED across apps in a YARN cluster.

This could be an issue because SNAP will have to access the segments from DISK when the segments are not in memory. IO is typically 10x slower than RAM access. In a dedicated NODE this is not an issue because no other process will be using the RAM other than SNAP processes.

CPU

The number of CPUs allocated for a SNAP cluster should be a function of the number of segments constituting the SNAP indexes that are used in the cluster.

Example

Index 1 = 100 GB, 400 segments

Index 2 = 50 GB, 200 segments

Index 3 = 200 GB , 800 segments

Total across all Indexes = 1400 segments

Assuming median 100ms per segment query time the number of cores needed to achieve

- 1s per query performance = 1400 / (1s/100ms) = 140 cores.
  - 2s per query performance = 1400/ (2s/100ms)=70 cores.
    - 3s per query performance = 1400 / (3s/100ms) = 47 cores.

NOTE: 100ms per segment times are when the segment is in memory and is optimal and can be as low as 25-50ms depending on the use case.