A Report on Just-In-Time CSV Parser

Group SportyDS
https://github.com/swapnilauti/CSVParser

Swapnil Sudam Auti (sauti@buffalo.edu)
Sarang Dev (sarangde@buffalo.edu)
Pratik Patil (pratikvi@buffalo.edu)

Overview

In the ordinary database systems, querying a CSV file needs it to be parsed and indexed before any query could be executed. As a consequence, a lot of time is incurred in the initial loading, type checking, type conversions and indexing before the first query could be executed. In this project, we aim to build different parsing techniques which would enable us to dynamically parse the CSV file as and when the query is encountered. We also aim to reduce the time taken to execute the first query on the CSV file. By deferring the parsing, we also save time by parsing only on the columns/lookup values of interest. In other words, data which is not requested by any query would never be parsed.

At any given time, when a query on an untouched data in the CSV file is encountered, we aim to dynamically parse this new data. For that purpose, we build a data structure called Positional Maps which is a mapping from the location in the CSV file to the value. The column fetching or the value lookup is done with the help of this positional maps [2]. Positional maps are two dimensional arrays which help in retrieving the desired value from a block read of the file. During the initial file read, as we scan all the bytes, we simultaneously create this positional maps for every value in the CSV file, which would aid us in locating any cell within the file, thus significantly reducing the time taken to parse data for the subsequent queries.

The data value fetching strategy is coupled with Just-In-Time data structures (JITDs) [1] where we aim to dynamically index the column based on the workload. The physical representation of a JITD is defined by a set of generic components, or cogs, which capture the structure and semantics of the representation. The advantage of using JITDs is that a single JITD may implement and alternate between many different policies, rapidly adapting its behavior to fluctuating workload demands.

By combining JITDs with positional maps, we can create a prototype of CSV database which can effectively load and parse the file on demand as well as index the data on-the-fly. This combination can be effective for the read-only workloads.

Problem Statement

In traditional Databases the initial indexing of the data takes a lot of time, hence the data is not ready for querying immediately. There can be scenarios where the analytics on the data has to be run as soon as the data is ingested in the database. In order to make the data available for querying we can parse and index the data on-the-fly while processing the query. The aim of our project is to compare different parsing techniques and determine which technique would be better for a given scenario.

Implementation Details

We have implemented our programs in Java and have considered three parsing techniques to parse the CSV files:

Naïve Parser: Naïve parser is a prototype of traditional DB where all the data is indexed in the start. Our implementation of the Naïve Parser includes reading the entire CSV file upfront into the memory and then parse each cell and store it in a two dimensional array.

InFile Parser: InFile parser reads the file block by block into the memory. During the first query it creates a positional map for the entire CSV file and returns the data requested in the query. The subsequent queries make use of the positional map to map to the desired data value in the file. The block size can be controlled and can be tuned for good performance.

InMem Parser: InMem parser reads the file initially and stores it in memory in a byte array. Then whenever first query comes it creates a positional map and return the required data value. On subsequent queries, it uses the byte array and positional map to return the required data. The main difference between InMem and InFile is that InFile doesn't keep the file content in memory and only fetches value from the file via block reads.

In all the techniques discussed above, we have read the file in bytes so as to defer the casting lazily to the time when the values are required. Also for casting we have implemented a basic function to reduce the time used in extra checks in default casting provided by java libraries.

Metrics for Comparison

We are comparing the three parsing techniques i.e., InMem Parser, InFile Parser and Naïve Parser based on the following parameters:
1. Total time taken to load the whole CSV file
2. Parsing time for the first query
3. Maximum file size supported

Observations & Performance Analysis

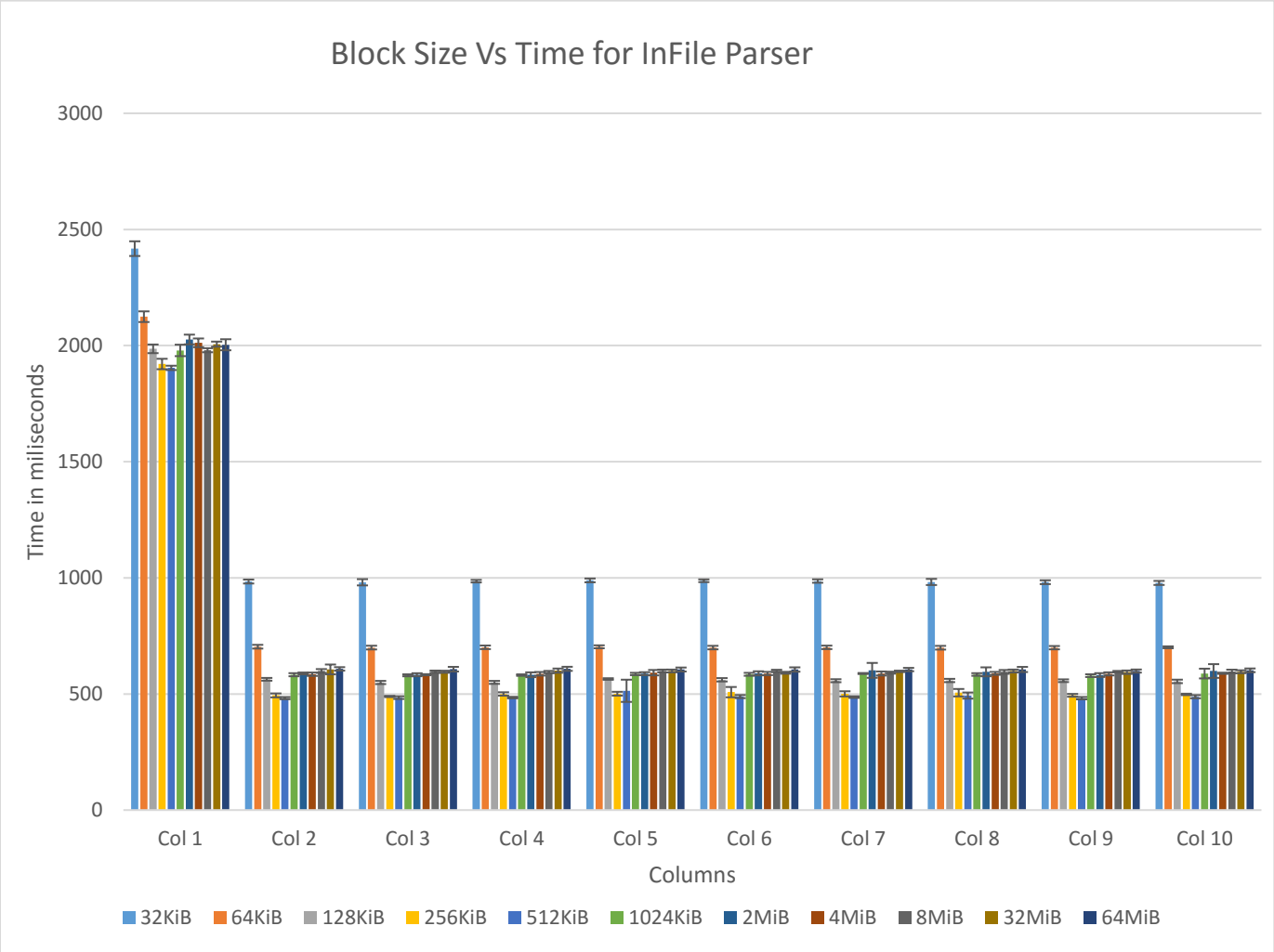
- **System Specifications:**
  Main Memory : 8 GB
  Processor : Intel i5-5200U CPU @ 2.20GHz
  L2 Cache: 0.5 MB
  L3 Cache: 3 MB
  JVM Heap Size Allocated : 4GB

- **File Specifications:**
  File Type : CSV file
  File Size : 300 MiB
  Number of rows: 1048576
  Number of columns: 10
  Data type contained: long

For comparison on total time taken to parse the whole file for different file sizes we have used file in sizes varying from 30 to 300 MiB in steps of 30 MiB having different number of rows. All the files were generated in python with each long value being 16 digits long. This was done to get a file of desired size.

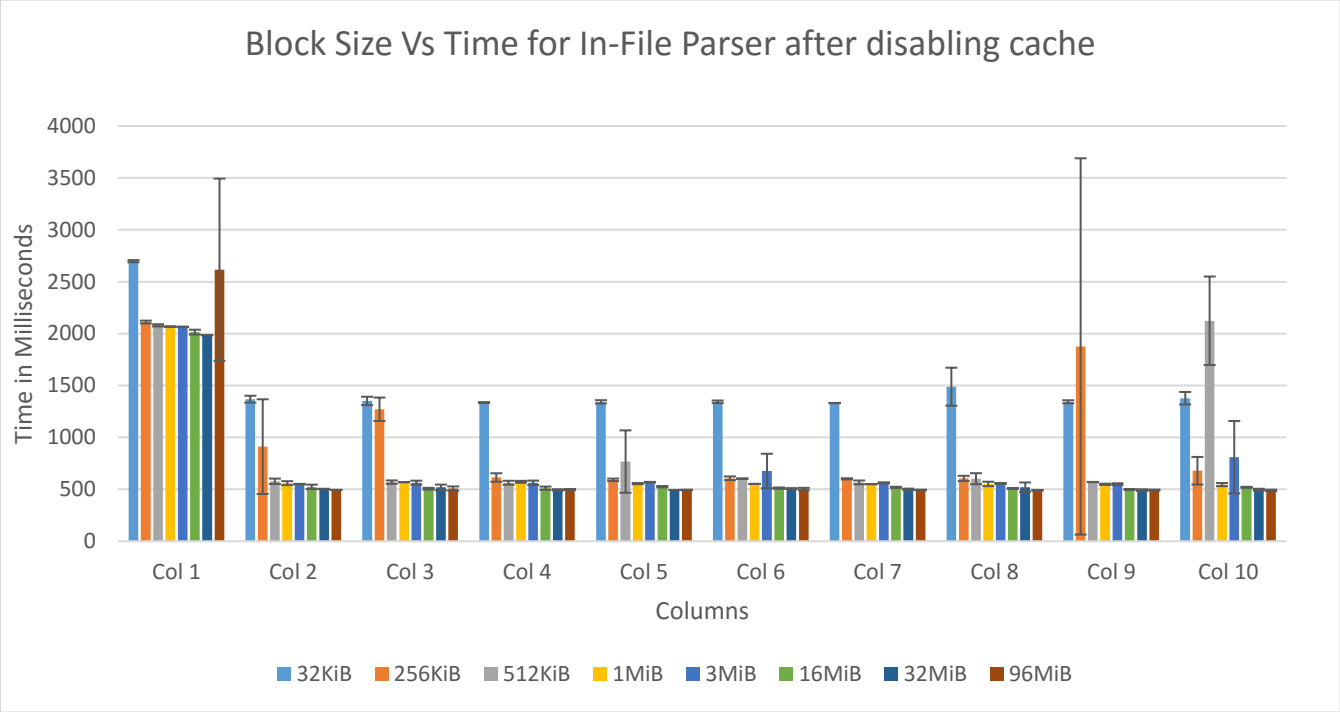
For all the experiments we have considered 2 initial runs as dummy runs in order to remove JIT interference and ten readings were taken and the average with a confidence interval of 95% was used in all the graphs. In all the readings we have taken care that the garbage collector didn't interfere with the reading. For this we have taken the heap size to be 4 GB and have used relatively lesser size files. We have also taken the Garbage Collector(GC) logs to verify that GC didn't run in between the observation times. Though there were a few readings in which it did interfere, but we have mentioned it in our observations.

Comparison of different block size for InFile Parser



The purpose of this experiment is to find an optimum block size in terms of time taken to parse the CSV file. Ideally, a very small block size (~32KiB) means more file I/O operations, thus more time taken to parse the CSV file. Whereas a large block size would mean less I/O and better performance in terms of time. But the performance dips after a certain block size(512KiB) slightly. The intermediate block size of 512 KiB takes the least time of all the block sizes to parse every column in the CSV file using InFile approach.

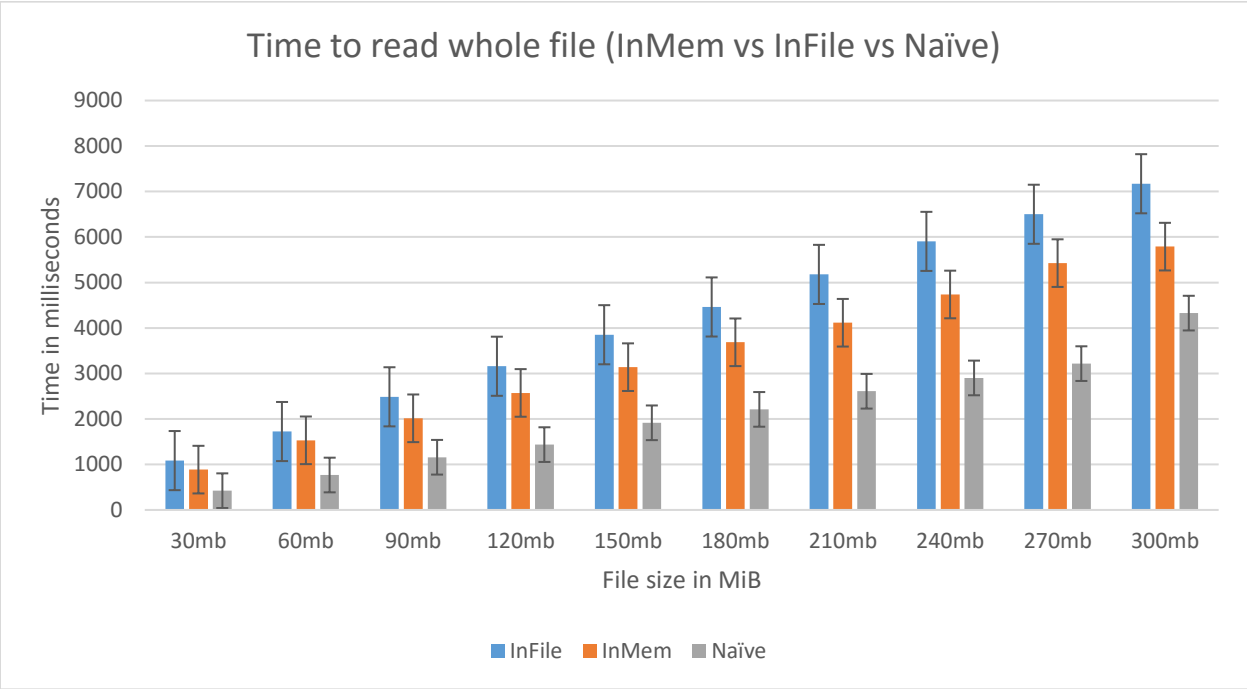
Comparison of different block size for InFile Parser After disabling Cache



The purpose of disabling all the caches (L1, L2 and L3) was to determine the impact of cache on the performance of the InFile parser with respect to different block sizes. For disabling the cache, we created a Linux module which sets the 30th bit in CR0 register to 0. Before running our experiment, we dynamically inserted the module by insmod command.

It is apparent that performance in terms of time improves as we increase the block size. This goes on to prove that the size of L2 cache for the machine on which we ran our experiments i.e. 512KiB is the prime reason for optimum performance for 512KiB block size (without disabling cache). The spikes in few readings (eg. Col1-96MiB, Col9-256KiB, etc.) is due to the GC that kicked in between our readings as found out from the GC logs.
Note: Since after disabling the cache, the system slows down significantly, we decided to reduce the iterations for the whole program to three and the number of block sizes as well.

Total time taken to parse the whole file

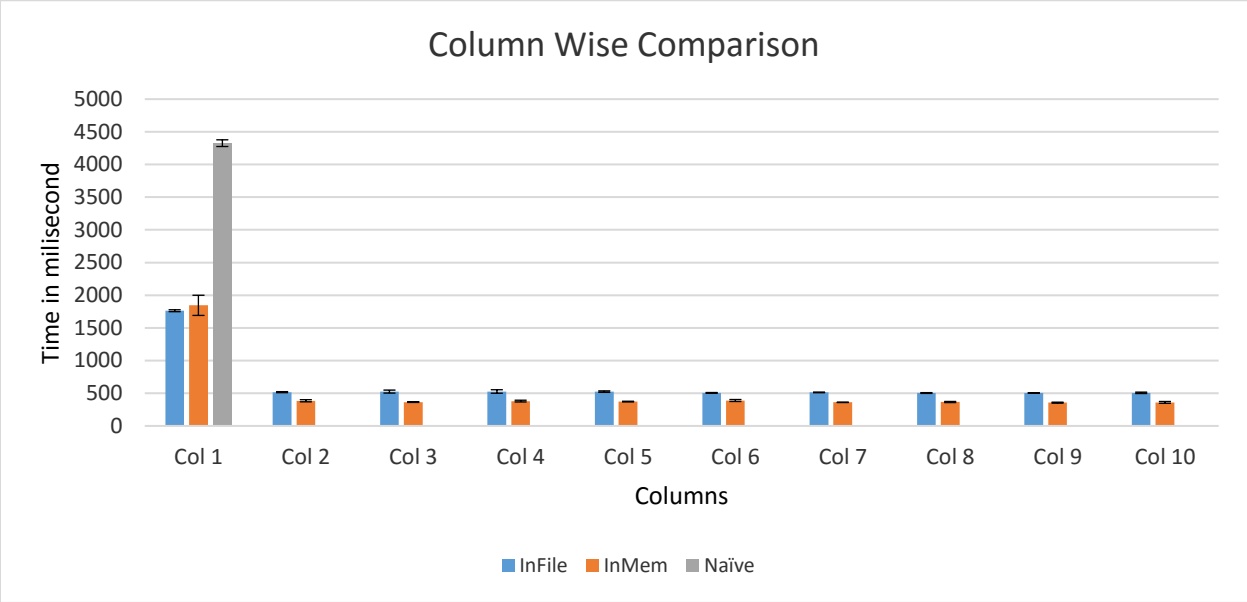


The total time taken to parse the complete CSV file when the file is read column wise is in following order:

$$\text{Time(Naïve)} < \text{Time(InMem)} < \text{Time(InFile)}$$

The three parsing techniques differ in the total number of times I/O operations are performed. InFile parser takes more time than the other techniques as maximum IO is performed (each time a column is loaded). The difference of total time between InFile and InMem will keep on increasing for larger files as the number of IO's for InFile will increase. Also InMem parser takes more time than Naïve parser as in case of Naïve, all the columns are loaded and parsed at once whereas for InMem the parsing is done separately for each column. For parsing each column in InMem the column has to be read from the file present in byte Array in memory. As the file size increases, the difference between the timings of Naïve and InMem also increases due more reads from the memory. Even after taking all the steps for GC, for the reading of 300 MiB file for Naïve parser, the GC kicked in between our readings as found out from the GC logs.

Parsing time for First Query



In this experiment, we compare the time taken to parse a 300MiB CSV file by each of the three parsers for first queries on all the columns one by one. We are considering a range scan query on the first column to be our first query for all the three techniques.
As evident from the graph above, the time taken for the first query is as follows:

$$\text{Time(InFile)} < \text{Time(InMem)} < \text{Time(Naïve)}$$

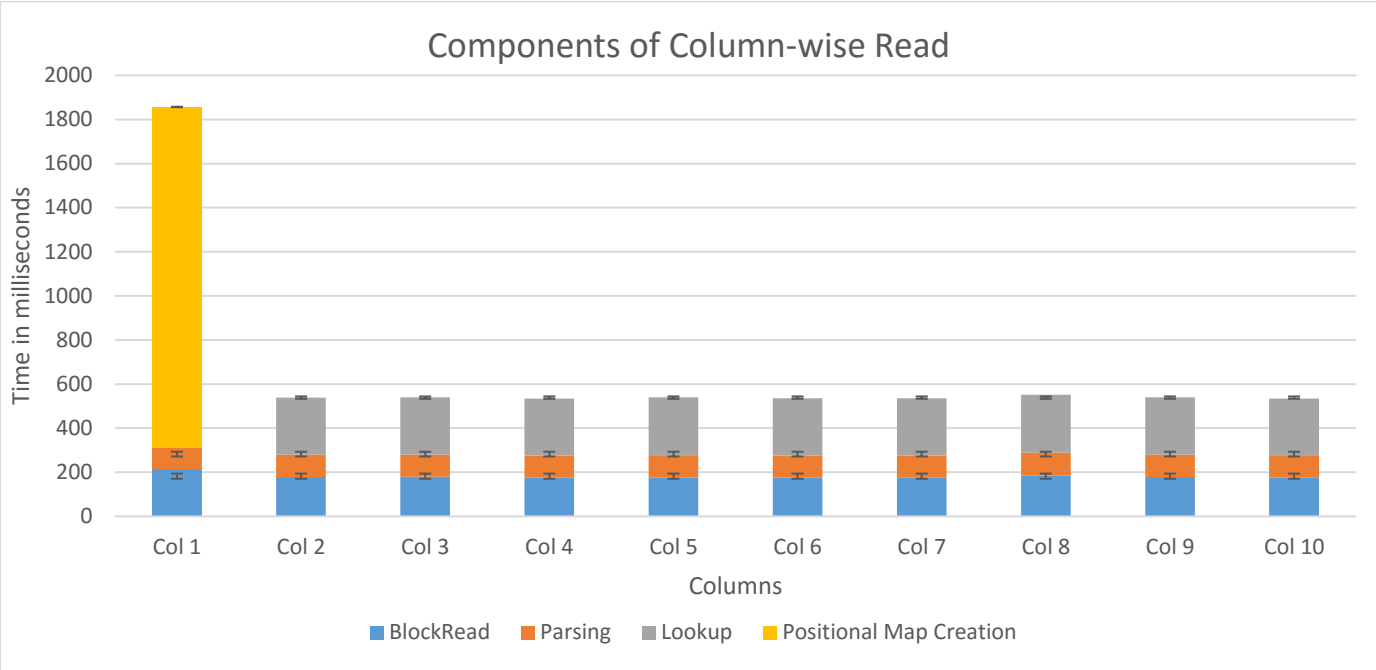
Since the Naïve Parser parses the entire CSV file before executing any query, it takes the maximum time. The InMem parser loads the entire CSV file into memory before answering any query but doesn't make type conversions unless the query on the particular field is encountered. Hence the time taken by InMem parser is less than that by Naïve parser. The InFile parser loads and make type conversions of only the fields of interest in a particular query, thus taking the minimum parsing time of all.

For all the subsequent queries on other columns, Naïve parser doesn't take any time as it has parsed all other columns in the first query on column 1. For InMem and InFile we can see:

$$\text{Time(InMem)} < \text{Time(InFile)}$$

This is due to the less file IO performed in case of InMem compared to InFile.

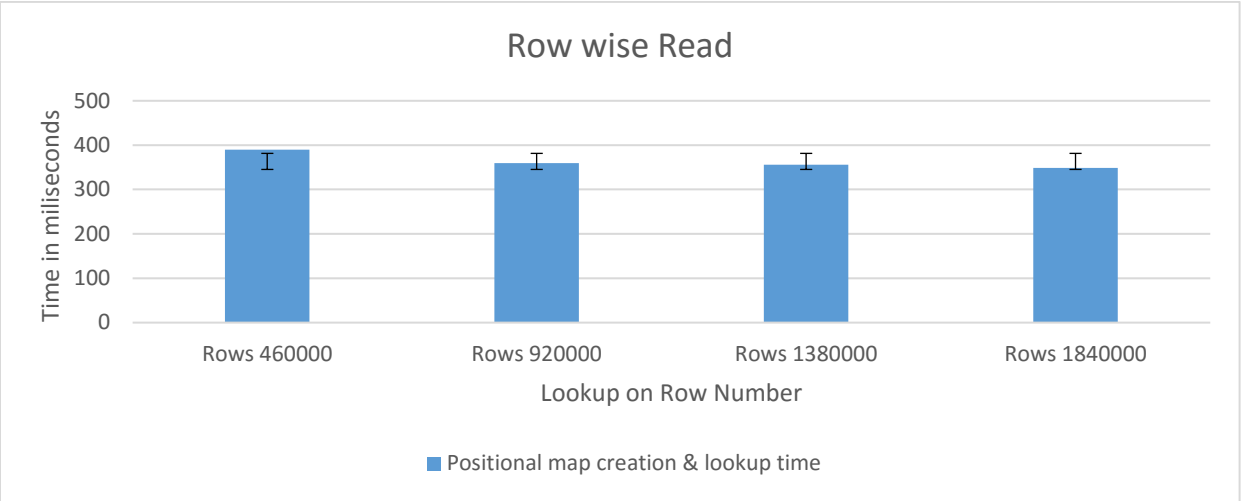
Analysis of Column-wise Read



This experiment does the in-depth analysis of the parsing using InFile parser. There are three components for the total time for a particular column fetch:
BlockRead time: Total time taken to read the file into blocks of size 512KiB
Lookup time: time taken to create byte array of value from the block read using positional map
Parsing time: Time taken to convert byte array into long data type for all values in the column.
Positional Map Creation time: Time taken to create positional maps. Since in our implementation we do lookup simultaneously while creating positional map for the first column, the lookup time is included in positional map creation time.

As expected, parsing and block read time are almost same for all the columns. The block read time is same for all the columns since for every column read, we read the entire file in blocks of fixed size(512KiB). The parsing time and lookup time from column 2 onwards is equivalent since the type of data in the file we considered is same across all the columns. The majority of the time for column 1 is taken by positional map creation which also includes lookup.

Analysis of Row-wise Read



This experiment is done to determine the time taken to fetch a cell value in the CSV file at a particular row. The first row-wise query was done on row corresponding to one-fourth of the file size (i.e. 460000) and the subsequent queries were on the rows incremented in steps of 460000. Since for every query, a partial positional map of same size is constructed, the time taken is almost similar in all the cases. For any row-wise query, only the desired cell value is parsed and positional map till that cell is created if it doesn't exist.

Query No	Row-wise query on row number	Time Taken (milliseconds)
1	1840000	1378
2	460000	0.127574

If the first query encountered is the row-wise query on the last row of the CSV file, the time taken is 1378 milliseconds. Subsequent row-wise queries for any row is mere 0.12 milliseconds. Such a huge difference is expected since for the second query since the location of the given cell in the file can be obtained from the positional map created during the first query. Hence we can assume that the time taken to create the positional map for the entire file is ~1378 milliseconds.

Volume Testing

This experiment is done to determine the maximum size of CSV file that can be handled by each of the three parsers before running out of Heap memory. Following are the observations made:
Breaking Point for all the parsing techniques (with 2GB heap size):

1. Naïve : 660 MiB
2. InFile : 600 MiB
3. InMem : 420 MiB

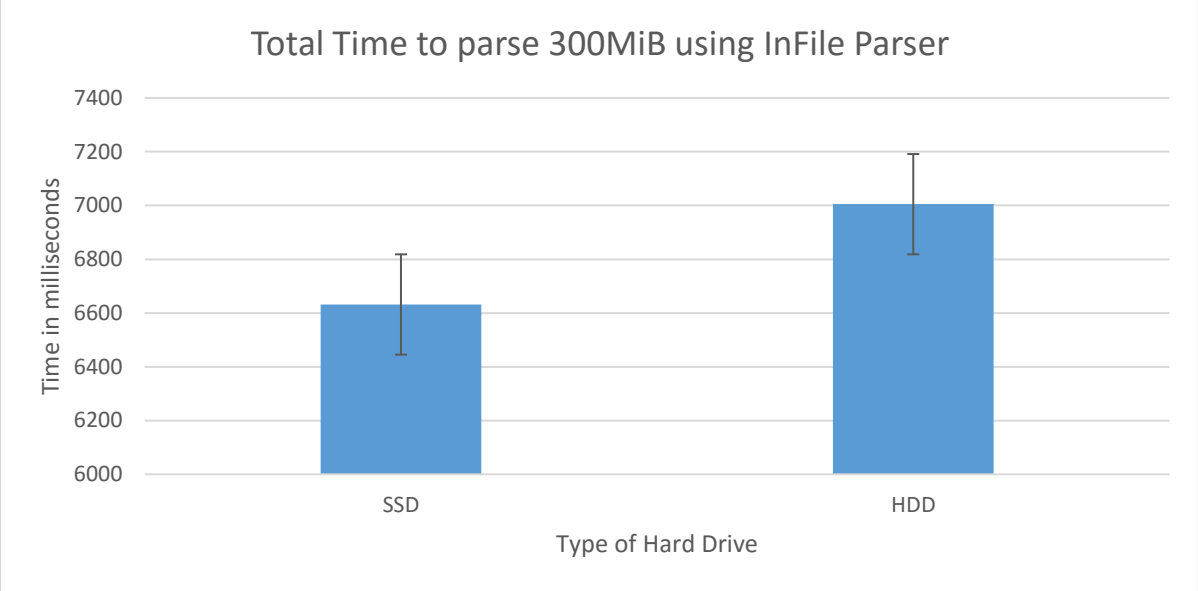
The maximum file size that could be handled by all the 3 parsers could be arranged as:

$$\text{MaxSize(InMem)} < \text{MaxSize(InFile)} < \text{MaxSize(Naïve)}$$

Since InMem Parser initially loads the entire CSV file into memory upfront in the form of an array of bytes, and then makes necessary type conversions on-the-fly and index them, if we parse the entire CSV file by querying all the columns, we end up having two copies of the same data in memory. InFile uses more memory than Naïve parser as we have an additional positional map of same dimension as that of the CSV file.

The file used in all the experiments above contains only 16 digit long values. We found that by having only single digit numbers in file the breaking point reduces by a factor of 16 as we are storing the values as long in JITD index.

Effect of type of Hard Drive used



HDD: Hard Disk Drive(HDD) consist of a bunch of magnetic discs on which data is stored.
SSD: Solid State Drive(SSD) store the data in the memory chips.

Unlike HDD, SSD has no moving component and thus have higher reading speed than HDD. As a result, the time taken to parse the entire CSV file using SSD is much less than that by HDD.

Conclusion

By conducting all the experiments, we came to following conclusions:

1. Naïve Parser takes the least time for complete parsing of the file and is also better in terms of maximum file size supported as seen from the volume testing. But the time taken to parse the file for the first query is quite large. Thus it can only be useful for small file sizes where the initial load time would not be high.
2. InFile parser takes maximum time for complete parsing of the file compared to other two techniques. But it supports larger file sizes and also takes the least time to parse the file for the first query. Thus it is ideal for large file sizes where it is required to query the CSV file as soon as it becomes available. It is also more suitable for files where the queries are run on selective columns, as only the required columns are parsed.
3. InMem parser takes more time than InFile parser but less than that of Naïve Parser for complete parsing of file. The time taken to parse the file for the first query is more than Naïve parser but less that that of InFile parser. But the file size supported in volume testing is the least compared to the other techniques. Thus, InMem should be preferred over InFile parser if we know that queries would be run on majority of the columns, as the time taken to parse the complete file is less than that of InFile Parser. But the support for the maximum file size would be a tradeoff for a given heap size as evident from the volume testing.

References

1. Just-In-Time Data Structures Oliver Kennedy, Lukasz Ziarek. In CIDR, 2015

2. NoDB: Efficient Query Execution on Raw Data Files Ioannis Alagiannis, Renata Borovica-Gajic, Miguel Branco, Stratos Idreos and Anastasia Ailamaki.