Implementing NoDB in action (standalone)

Team members: Suraj Kumar (18305R008), Tasneem Lightwala (183050004), Shreya Alva (183050003)

This project is based on the research paper ‘NoDB in Action: Adaptive Query Processing on Raw Data by Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, Anastasia Ailamaki’. The aim of NoDB is to minimize querying time for large files by reducing the need to load data by constructing adaptive indices on it.

Our project is a standalone implementation that demonstrates the efficacy of creating indices while reading data from csv files. Refer to our code in branch ‘suraj’. Our git repo is at: https://git.cse.iitb.ac.in/shreya/NoDB-CS631

To implement this, our approach was creating a linked list that contained address information for each node. This address information was encoded into a string, col_locations. A positional map kept track of what columns had their index information stored and where in col_locations. We would cease tokenizing the row when the positions needed were found. While augmenting the index for new columns, we chose the column closest to them as a starting point to tokenize the file.

To test our code, we compared it to the case where each line of the file was read and tokenized without an index (basic.c). We observed that while constructing indices for the first time, our code is slightly slower than the basic implementation, but once an initial index is in place, queries on these columns are faster than the basic implementation. For queries that fetch columns that are further in the csv, using an index has a clear advantage over reading the raw file without index.

The first query constructs the initial index and indices on daily_crowd, F1. It takes a considerable amount of time, but the future queries are faster.

These readings represent the mean time over 5 worst runs and were taken after giving the system a warm start, i.e. running queries several times so that any benefits offered by the caching are evened out for both paradigms. These queries have been run several time and the figures above capture the general trends we observed.

Query	**Execution with index (time in seconds)**	**Execution without index (time in seconds)**
select daily_crowd from train.csv (index already created on daily_crowd)	0.000175	0.016122
select post_day, target	0.026764	0.026108
select target, page_likes	0.018340	0.032942

To integrate our project with PostgreSQL we will have to do the following:

We trace the execution of a command in PostgreSQL: There is a sequence of function which is called while reading from files. There is a main method which is called by the command line interface or API in some programming language. Now this main method calls PostgresMain method which in turns calls exec_simple_query method which is the starting point of any query or transaction in SQL. Then there is a sequence of function called for lexical and syntax analysis (in lex and yacc) which comprises frontend for postgres.

The backend part constitutes of query planning module followed by query execution module. Before executing any query a portal is being made, then there is a ExecutorRun method which delegates the responsibilities of execution to series of function calls i.e. Standard_ExecutorRun, ExecutePlan, ExecuteProcNode, ExecuteForeignScan, ExecuteScan, ExecuteScanFetch etc.

For reading from files it's important to diverge parallely from method ExecutePlan as done in NoDB implementation. Now the fileIteratorForeignScan method iterates over series of functions for copying data from files (NextCopyFrom, NextCopyFromFields, CopyReadLine, CopyReadLineText, CopyLoadRawBuf, CopyGetData). The actual

reading of data is happening in function CopyGetData function. The end points of functions flow is depicted below. For the first time when a table is created there is a series of functions delegated to read and store the attributes of csv file (fileGetForeignRelSize, fileGetOptions, estimateSize, get_file_fdw_attribute_options etc).
Either we can write our own Foreign Data Wrapper or use some FDW wrappers provided as an extension to postgresql like postgres_fwd for reading data from different database or file_fdw for reading data from files. Usually the COPY command in postgres is written in such a way that it copies the data into tables but file_fdw in latest version of postgresql is overriding COPY definitions such that whenever we read data from external file it does not copy the data into tables. It is very much useful for data of very large size where response time to get results of a query is very high.

Foreign Data Wrapper (file_fdw) is used as proxy to read from csv file. It comes as an extension to postgresql (in contrib module). Firstly we need to compile and install this module and then create a proxy server using this. Now data can be read using sql. Here data files must be in a format that can be read by COPY FROM. To create an instance of file_fdw we use command CREATE EXTENSION file_fdw. Then we will create a server which acts as proxy to file system using command CREATE SERVER <server_name> FOREIGN DATA WRAPPER file_fdw. Then we can create the table using SQL query and we will specify the location of file corresponding to table using OPTIONS command in this step. Now queries can be fired directly on files using sql commands.

Effort distribution:

Everyone in this team had an equal contribution.

- Suraj:
  - wrapper to fire queries in standalone, code refactoring, debugging
- Tasneem:
  - laid out base implementation, debugging
- Shreya:
  - extended base implementation, debugging