**My approach for manipulating the dataset.**

At first glance the CSV dataset in its original form is having all 96 columns as Strings. We need to cast it to corresponding datatypes wherever needed.

For this we first need to load this into a dataframe of a spark cluster, then we can manipulate the columns in the dataframe.

Important and tricky things to consider while loading into a dataframe are

* Identify the delimiter, in our case its comma (","). And this delimiter is been used inside some of the columns as well, so need to ignore "," inside double quotes.
* There are some Newline characters (\n) inside some of the columns which will cause the parser to think it as end of the row and the next line will be treated as a new record.
* For this we need to identify a way to make the parser to ignore new line characters inside columns (double quotes)
* This is achieved by using an option called "multiLine" to true while reading csv. Since there are "\" characters also we need to escape them.
* Once we read the dataset into a dataframe, we can check if it has loaded correct by selecting any of the columns randomly, mainly pricing columns and compare the result with the actual dataset).
* Now all the columns were loaded as Strings by default. Since there are many other datatypes, need to clean the strings and cast them to corresponding datatypes.
* I had categorized the columns into 6 different datatypes, so that they can be readily used for calculations or aggregations by the downstream applications.
  + Integers - id, host\_id, host\_response\_rate, bedrooms, availability\_30, review\_scores\_value etc.
  + Double - All currency columns like price, cleaning\_fee, reviews\_per\_month (sometimes its a decimal)
* Boolean - host\_has\_profile\_pic, has\_availability, instant\_bookable etc.
  + - String which is comma separated values - amenities, host\_verifications etc.
    - Date columns - Date in String were converted to UNIX timestamp with MM/dd/yyyy format. host\_since, first\_review, last\_review etc.
    - Actual strings - Rest all columns were treated as Strings itself.
* For all the above categories (except actual strings), to maintain reusability four functions has been created to format them to corresponding datatypes.
  + format\_currency\_data - Its main purpose is to eliminate $, % , { symbols and cast them to Double.
  + format\_numeric\_data- Is same as format\_currency\_data but to cast to Integers.
  + format\_boolean\_data- convert the string with 't' to True and 'f' to false, which are Boolean types.
  + format\_amenities- Amenities and host\_verifications columns were stripped off with unnecessary characters and made as comma separated values, so that they can be exploded in the next processes.
* All these above Scala functions were registered as User Defined Functions in spark so that they can be easily referenced in any of the spark-sql Statements.
* All the string columns were formatted with "withColumn" and new columns were formed.
* All New columns will be ended with either "\_formatted” or "\_date" or "\_b" .
* Since the dataframe will be appended with old and new columns, create a sequence with non-redundant and formatted columns and create a final dataframe with columns from this sequence.
* Create a case class with all the columns and corresponding datatypes and then convert this final dataframe to dataset which is more sophisticated than dataframe and provides analysis errors at compile time itself opposed to dataframes which surprise us with errors at run time.

**Code interfacing with the source system and the data scientists' models and how the data could be continuously updated.**

I would like to create a data pipeline for end to end data flow.

The landing zone for all the input files will be a HDFS ETL folder using a Unix script to FTP files from source. A Spark job (with spark-submit) will be run once the file arrives by checking a "file\_watcher" which is 0-byte file to check if file arrived or not. This job formats the data and integrates with Hive to load the final dataframe into a staging tables in hive (which are truncate and load at every run)

This table will be stored as parquet format to effectively use the compression techniques and columnar storage. Out of the 96 columns from the staging table only certain columns has importance in price prediction, some of them are below:

* Price
* Cleaning\_fee
* Extra\_people\_fee
* Security\_deposit
* Guest\_included - since this is a potential search criterion while booking
* Location - Latitude, Longitude, city, neighborhood, Country
* Number of reviews
* Amenities
* host\_has\_profile\_pic
* Property\_type

In hive actual master/historical tables will be created and partitioned based on location/City. Bucketing will be performed if needed based on the query patterns used by data scientists.

Using HiveQL the data loaded in staging table will be queried based on ingestion dates and upsert statements can be used to either insert if new record or update if existing record. Which constitutes to incremental load.

With the up to date data being loaded every day, data science team can pull the latest data based on date limit criteria and run their models to predict changing prices. They can pull the formatted data from Hive tables.

**Scalability if confronted with terabytes of data.**

Imagining the data could be huge over time, I prefer to use AWS EMR/any cloud cluster which can auto scale (through CloudWatch) and high availability will be provided.

With respective code, I would like to use partitionBy while loading data into spark cluster so that data/rdds will be distributed across different nodes based on partitioned column which will help during reduce operations with minimal shuffling. Also, I prefer to use caching techniques like cache () or persist () so that the intermediate resultsets will be stored in memory/Disk and can be readily available without recomputing. Also Hive tables can be partitioned and clustered. Hive need to be configured to use map side joins whenever possible and avoid shuffling. Along with these some important configuration settings will be carefully applied like dynamicAllocation of Executors, number of executors and number of cores and memory for executor etc. And for Hive configurations will be made to use Tez/Spark engine instead of default MR.

**Deriving a price prediction model:**

Though I don’t belong to data science background, I have some views about predicting the prices. I would filter out unnecessary data from the resultset and load the necessary fields into dataframes. With the use case above, I prefer to use time any time-series model to predict the prices.

Many of the columns after formatting from data engineering team are straight forward, some columns like amenities still need some processing. I can group the amenities by host\_id and amenity\_name to find the number of occurrences of an amenity and consider the ones that are repeating the most number of times/common across all the hosts and take them as basic and powerful amenities while predicting prices.

From Given dataset, WIFI, TV, parking, washer etc. are more repetitive than others, so will take these as base and provide rankings for the hosts, by considering other parameters like location, guests allowed or not, how many beds and washrooms provided etc. Along with this we can observe the trend of prices over the seasons and with respective to the neighborhood.

Also, we can use Levenshtein and Soundex, fuzzylogic algorithms available in spark to depict the positivity of a review comment and give rankings which can also be considered in deriving the future price.