

NBA SportVU Tracking Data

Felipe Chamma, Felipe Ferreira, Alex Morris, Ryan Speed

Dataset

SportVU is a camera system hung from the rafters that collects tracking data at a rate of 25 Hertz (25 times per second), or equivalently every 40 milliseconds. We define a moment as one of the 40 millisecond snapshots. The system follows the ball and every player on the court, recording 11 total observations per moment. For each moment a coordinate (x, y) is recorded for each player, representing his location on the court. Where x ranges from 0 to 94 and y ranges from 0 to 50, the width and length of the court respectively.

From this system we were able to acquire data across 90 thousand moments, totaling approximately 1 million observations per game, for 2460 games; The entire 2014-2015 NBA season. The tracking data was very raw, and needed a significant amount of cleaning, processing, and manipulation before we could gain any insights.

Our Environment

Local Environment

For the local testing environment we used a MacBook Pro 15" with a 2.5Ghz Intel Core i7 and 16GB of high speed RAM. It also contains an extremely fast SSD .

We were running Postgres Server version 9.4.4.1.



Distributed Environment

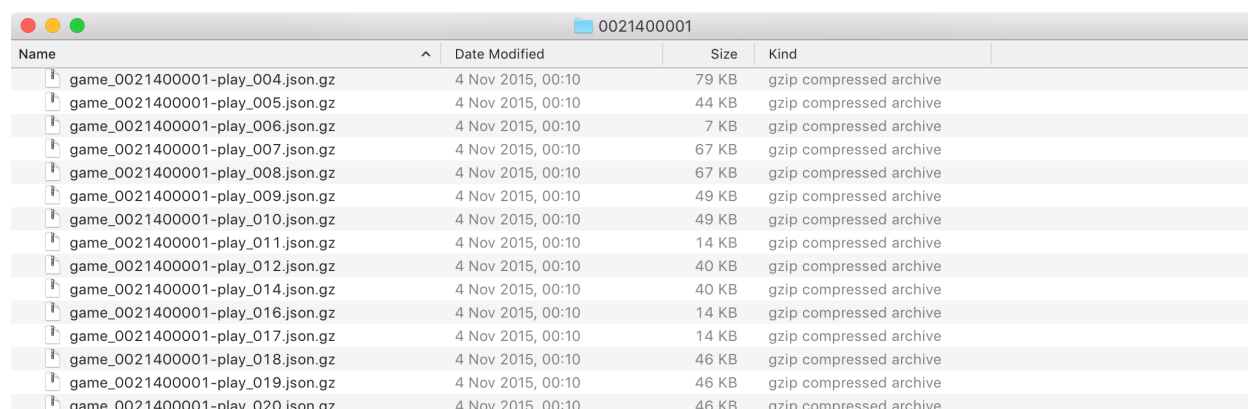
Our distributed environment was on AWS EMR and where we set up a cluster containing 6 nodes (1 master, 5 slave). The EC2 instance type of each node was an m3.2xlarge which is a memory optimized instance. We used the same cluster for both Hive and Spark parts of the project. The software we chose to install was Hive 1.0.0, Spark 1.6.1 and Zeppelin-Sandbox 0.5.6.

We carried out most of the querying in Zeppelin.

Pre Processing

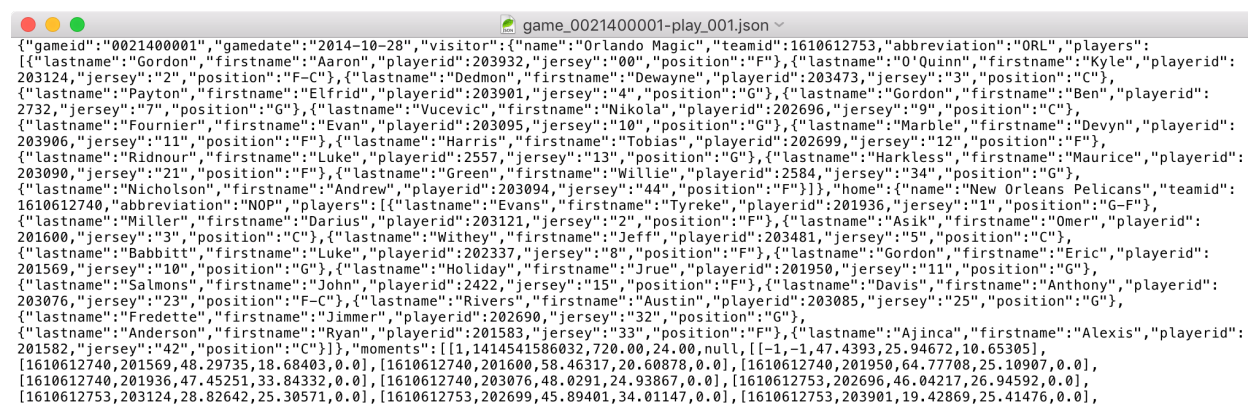
The Data contains a directory for each date, which contains a directory for each game on that date. Inside each game directory, there are 300 to 500 different JSON play files. We wrote a python script which could take each of these archives and convert it to 2 flat CSV files per game. See appendix for code.

The tracking data come directly from stats.nba.com, where a single game is 20-25 MB compressed. Each game contains approximately 1 million moment observations, as well as game and player metadata. The first month of the 2015 season is 6 GB compressed. We had to write a parsing function which would take the compressed JSON 'play' files stored S3 in the following hierarchy:



| Name | Date Modified | Size | Kind |
|----------------------------------|-------------------|-------|-------------------------|
| game_0021400001-play_004.json.gz | 4 Nov 2015, 00:10 | 79 KB | gzip compressed archive |
| game_0021400001-play_005.json.gz | 4 Nov 2015, 00:10 | 44 KB | gzip compressed archive |
| game_0021400001-play_006.json.gz | 4 Nov 2015, 00:10 | 7 KB | gzip compressed archive |
| game_0021400001-play_007.json.gz | 4 Nov 2015, 00:10 | 67 KB | gzip compressed archive |
| game_0021400001-play_008.json.gz | 4 Nov 2015, 00:10 | 67 KB | gzip compressed archive |
| game_0021400001-play_009.json.gz | 4 Nov 2015, 00:10 | 49 KB | gzip compressed archive |
| game_0021400001-play_010.json.gz | 4 Nov 2015, 00:10 | 49 KB | gzip compressed archive |
| game_0021400001-play_011.json.gz | 4 Nov 2015, 00:10 | 14 KB | gzip compressed archive |
| game_0021400001-play_012.json.gz | 4 Nov 2015, 00:10 | 40 KB | gzip compressed archive |
| game_0021400001-play_014.json.gz | 4 Nov 2015, 00:10 | 40 KB | gzip compressed archive |
| game_0021400001-play_016.json.gz | 4 Nov 2015, 00:10 | 14 KB | gzip compressed archive |
| game_0021400001-play_017.json.gz | 4 Nov 2015, 00:10 | 14 KB | gzip compressed archive |
| game_0021400001-play_018.json.gz | 4 Nov 2015, 00:10 | 46 KB | gzip compressed archive |
| game_0021400001-play_019.json.gz | 4 Nov 2015, 00:10 | 46 KB | gzip compressed archive |
| game_0021400001-play_020.json.gz | 4 Nov 2015, 00:10 | 46 KB | gzip compressed archive |

Where each JSON file has the following structure, with each player repeated at the start of the file and the moments for that given play in the bottom. There may be some overlap between plays so we have to deal with that to



```
{
  "gameid": "0021400001",
  "gamedate": "2014-10-28",
  "visitor": {
    "name": "Orlando Magic",
    "teamid": 1610612753,
    "abbreviation": "ORL",
    "players": [
      {
        "lastname": "Gordon",
        "firstname": "Aaron",
        "playerid": 203932,
        "jersey": "00",
        "position": "F",
        "home": {
          "name": "New Orleans Pelicans",
          "teamid": 1610612740,
          "abbreviation": "NOP",
          "players": [
            {
              "lastname": "Evans",
              "firstname": "Tyreke",
              "playerid": 201936,
              "jersey": "1",
              "position": "G-F",
              "moments": [
                [
                  141454586032,
                  720.00,
                  24.00,
                  null,
                  [-1, -1, 47.4393, 25.94672, 10.65305],
                  [1610612740, 201569, 48.29735, 18.68403, 0.0],
                  [1610612740, 201600, 58.46317, 20.60878, 0.0],
                  [1610612740, 201950, 64.77708, 25.10907, 0.0],
                  [1610612740, 201936, 47.45251, 33.84332, 0.0],
                  [1610612740, 203076, 48.0291, 24.93867, 0.0],
                  [1610612753, 202696, 46.04217, 26.94592, 0.0],
                  [1610612753, 203124, 28.82642, 25.30571, 0.0],
                  [1610612753, 202699, 45.89401, 34.01147, 0.0],
                  [1610612753, 203901, 19.42869, 25.41476, 0.0],
                ]
              ]
            }
          ]
        }
      }
    ]
  }
}
```

get accurate location data.

We wrote a Python script which pulls all these small compressed JSON files from S3, and parses them into a single CSV for players and a CSV for moments per game and

re-uploads them into separate directories on S3 for easy loading into HIVE and SQL tables. The code for this parsing can be found in the appendix.

Problem

Initially, we wanted to create a vector to represent each player's locations on the basketball court, then cluster the vectors. We were successful using python locally and using Spark, however the task was not suitable for a querying language such as hive. In order to properly test and compare different distribution systems, with our local environment, we shifted our approach. We are going to test an algorithm which implements a euclidean distance measure aggregated by player.

Given the data we are trying to calculate total distance ran per player using windowing function queries using Spark, Hive & locally on PostgreSQL. We were able to use all of the same preprocessing and data loading from the clustering problem.

General Approach

First we created two tables from the raw files. One called players which contained info about each player and the other called locations which represented the position of every player per game.

To calculate distance ran we run a query which carries out a window function over the x and y coordinates partitioned by game_id and ordered by timestamp. We then take square root of the sum of the squares of the differences of these values to get the distance traveled per time interval. Summing these distances up over each game id gives us the total distance traveled (in feet) per game. We can convert this to miles in our query simply by multiplying the value by 0.000189394.

$$\sum_{i=0}^{i=n} \sqrt{(x_{t_i} - x_{t_{i-1}})^2 + (y_{t_i} - y_{t_{i-1}})^2} \times 0.000189394$$

The general equation is given above, where i=0 to n represents all the moments for a given player in a given game.

PostgreSQL

The Postgres approach implements the general one just described. One drawback of using Postgres is that the data must be loaded as a single CSV. Therefore, we wrote a python script to combine all of the individual game files into one flat CSV. The final query can be found in the Appendix.

Hive

The Hive approach is virtually identical to PostgreSQL, with some intricacies with the syntax. Hive allows us to load data from an S3 bucket directly as multiple compressed text files so we don't have to combine them into a single text file. We also were able to partition the data by game, and distribute the tasks across nodes. Hive performed far better than Postgres, probably due to the fact that the data, being by game partitioned by game, is tailored for map reduce.

Spark

Using SparkSQL with a HiveContext we were able to further increase our query execution speed. Spark also allows us to load our data into an RDD directly from the S3 bucket. Spark ran the query the fastest. The HiveContext SQL object allowed for window functions so the final query was very similar to that of Hives. It's worth noting that Spark used many more resources when carrying out the query compared to Hive. Hive used around 7GB of RAM at peak, whereas Spark used the Clusters entire 105GB of available RAM.

Results

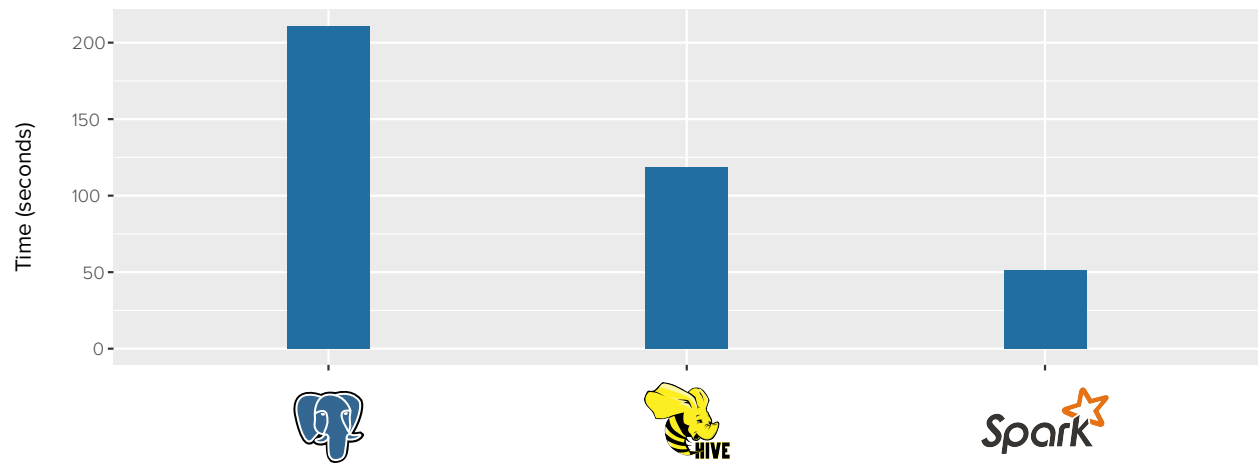
Below we can see each player in the first game and how many miles he ran. We can also see how each approach compares. Spark was the only one that was finished under a minute.

| Game ID | Player ID | Player Name | Position | Miles Run |
|----------|-----------|--------------------------|----------|-----------|
| 21400017 | 203484 | Kentavious Caldwell-Pope | G | 3.280599 |
| 21400020 | 203460 | Andre Roberson | G | 3.210647 |
| 21400022 | 2399 | Mike Dunleavy | F | 3.170411 |
| 21400020 | 203103 | Perry Jones | F | 3.114226 |
| 21400011 | 201167 | Arron Afflalo | G | 3.110374 |
| 21400022 | 2544 | LeBron James | F | 3.016054 |
| 21400022 | 201567 | Kevin Love | F | 2.992457 |
| 21400022 | 201149 | Joakim Noah | C | 2.986612 |
| 21400015 | 202083 | Wesley Matthews | G | 2.957845 |
| 21400014 | 201939 | Stephen Curry | G | 2.931559 |
| 21400012 | 203504 | Trey Burke | G | 2.919488 |

| | Spark | Hive | Postgres |
|------------------|---|--|--|
| Speed (1 day) | 51s | 118s | 211s |
| Speed (2 days) | 54s | 158s | 394s |
| Output Format | A | B- | B- |
| Resource Usage | C | A | C |
| Code Readability | B+ | A- | A- |
| Advantages | Fast, takes ordinary SQL code. Scales well. Can output as Python object for further analysis. | Becomes more efficient as data gets larger compared to PostgreSQL | Better for small datasets. Can be run locally machine |
| Disadvantages | Unusual syntax wrapper for running queries. Requires lots of memory | Slow on smaller subsets, not efficient for joining tables compared to other technologies | Struggles on large datasets. Loading from multiple files tedious |

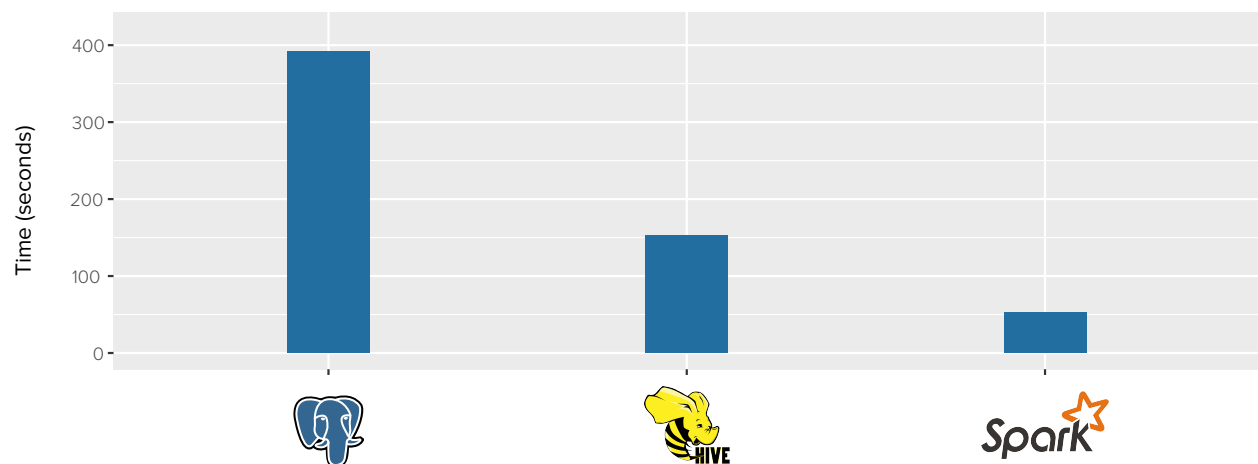
Running on a single days worth of data (9 million rows)

Here we can see all 3 get results in a reasonable amount of time. Although Spark is the fastest as mentioned previously it uses many many more resources than HIVE to get to its final answer.



Running on a two days worth of data (18 million rows)

This plot shows Postgres really starting to struggle vs the distributed approaches. What is interesting however is the factor at which the 3 had increased. For a 2x increase in data, Postgres's total query time increased by a factor of 1.8, Hive by a factor of 1.33, but Spark by just 1.05. This suggests that although Spark uses lots of memory, its certainly seems to be capable of dealing with larger datasets. Hive maybe able to get closer to Spark's performance with a little tuning and table partitioning however.



Challenges & Future

During every basketball game the teams change side after halftime, so in order to map the position of each player consistently on the defensive end and offensive end throughout the game, we need to flip the coordinates after each half. We also had to take care of that across games as well since they may start on a different ends of the court on each game.

Resources

We used the following website as a tutorial of how to work with S3 and hive directly.:

- <https://blog.mustardgrain.com/2010/09/30/using-hive-with-existing-files-on-s3/>

Appendix

Boto / Python script for cleaning data:

```
from boto.s3.connection import S3Connection
from boto.s3.key import Key
import tarfile
import re
import gzip
from cStringIO import StringIO
import json
import csv

conn = S3Connection(ACCESS_KEY, SECRET_KEY, host="s3-us-west-2.amazonaws.com")
bucket = conn.get_bucket("dcproject")
keys = bucket.list()

directory_pattern = re.compile(r'^(4)-{2}-{2}.tar.gz$')
play_pattern = re.compile(r'^.+json.gz$')

for key in bucket.list():
    if directory_pattern.match(key.name):
        fp = StringIO(key.get_contents_as_string())
        tar = tarfile.open(mode="r:tar", fileobj=StringIO(key.get_contents_as_string()))
        first = True
        current_game_id = False
        completed_players, completed_moments, player_rows, moment_rows = [], [], [], []

        game_pattern = re.compile(re.escape(key.name.split('.')[0]) + r'/(\d{10})$')

        game_ids = [x for x in tar.getnames() if game_pattern.match(x)]

        for game_directory in game_ids:
            completed_players, completed_moments, player_rows, moment_rows, first = [], [], [], [], True
            for member in tar.getnames():
                if re.match(re.escape(game_directory) + r'/game_\d{10}-play_\d{3}\.json\.gz$', member):
                    if len(completed_moments) > 7500:
                        completed_moments = completed_moments[len(completed_moments)-7500:]

                    play = StringIO(tar.extractfile(member).read())
                    play_data = json.loads(gzip.GzipFile(fileobj=play).read())

                    if first:
                        game_id = game_directory.split('/')[1]
                        home_players = play_data["home"]["players"]
                        visitor_players = play_data["visitor"]["players"]

                        for player in home_players:
                            if player["playerid"] not in completed_players:
                                completed_players.append(player["playerid"])
                                player_rows.append(player.values() + [play_data["home"]["teamid"]])

                        for player in visitor_players:
                            if player["playerid"] not in completed_players:
                                completed_players.append(player["playerid"])
                                player_rows.append(player.values() + [play_data["home"]["teamid"]])

                    first = False

                    for moment in play_data["moments"]:
                        if moment[1] not in completed_moments:
                            completed_moments.append(moment[1])
                            for players in moment[5]:
                                moment_rows.append([game_id] + moment[0:4] + players)

            moments = StringIO()
            moments_writer = csv.writer(moments)

            players = StringIO()
            players_writer = csv.writer(players)

            players_writer.writerows(player_rows)
            moments_writer.writerows(moment_rows)

            moments_compressed = StringIO()
            moments_string = gzip.GzipFile(fileobj=moments_compressed, mode='w').write(moments.getvalue())

            moment_k = Key(bucket)
            players_k = Key(bucket)

            moment_k.key = game_directory.split('/')[0] + '/moments/' + game_directory.split('/')[1] + '.csv.gz'
            players_k.key = game_directory.split('/')[0] + '/players/' + game_directory.split('/')[1] + '.csv'
```

```

moment_k.set_contents_from_string(moments_compressed.getvalue())
players_k.set_contents_from_string(players.getvalue())

moment_k.set_acl('public-read')
players_k.set_acl('public-read')
print "written " + game_directory + '_moments.csv.gz'

```

Python script to combine files for PostgreSQL:

```

import glob
import csv
players = glob.glob('~/.moments/*.csv')

rows = []
for i in players:
    with open(i, 'r') as my_file:
        filereader = csv.reader(my_file, delimiter=',')
        rows.extend(list(filereader))

With open('~/.moments/all_moments.csv', 'w') as output:
    writer = csv.writer(output, delimiter=',')
    writer.writerows(rows)

```

PostgreSQL Query

```

CREATE TABLE players (player_id INT, last_name varchar, jersey int, first_name VARCHAR, position VARCHAR, team_id int);

CREATE TABLE locations (game_id int, quarter int, unix_time VARCHAR,
    game_clock FLOAT, shot_clock FLOAT, team_id INT, player_id INT, x FLOAT, y FLOAT, z FLOAT);

COPY players FROM '~/.players/all_players.csv' DELIMITER ',' CSV;

COPY locations FROM '~/.all_moments.csv' DELIMITER ',' CSV;

SELECT * FROM (
    SELECT game_id, LHS.player_id, TEXTCAT(TEXTCAT(first_name, ' '), last_name) AS name, position, total_distance FROM (SELECT game_id, player_id, SUM(distance)
    total_distance FROM
    (SELECT player_id, game_id, unix_time, SQRT(POW(x_distance, 2) + POW(y_distance, 2))*0.000189394 distance FROM
    (SELECT player_id, game_id, unix_time, x, y, x - lag(x) OVER (PARTITION BY game_id, player_id ORDER BY unix_time) x_distance, y - lag(y) OVER (PARTITION BY
    game_id, player_id ORDER BY unix_time) y_distance FROM raw_locations) LHS) LHS2
    GROUP BY player_id, game_id) LHS LEFT JOIN raw_players RHS ON LHS.player_id = RHS.player_id) AGG ORDER BY total_distance DESC LIMIT 25;

```

Hive Query

```

CREATE EXTERNAL TABLE raw_players (player_id int, last_name STRING, jersey INT, first_name String, position String) ROW FORMAT DELIMITED FIELDS TERMINATED
BY ',' LOCATION 's3://dcproject/all_games/players/';

CREATE EXTERNAL TABLE raw_locations (game_id int, quarter int, unix_time STRING, game_clock FLOAT, shot_clock FLOAT, team_id INT, player_id INT, x FLOAT, y
FLOAT, z FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION 's3://dcproject/all_games/moments/';

SELECT * FROM (
    SELECT game_id, LHS.player_id, CONCAT(first_name, " ", last_name), position, total_distance FROM (SELECT game_id, player_id, SUM(distance) total_distance FROM
    (SELECT player_id, game_id, unix_time, SQRT(POW(x_distance, 2) + POW(y_distance, 2))*0.000189394 distance FROM
    (SELECT player_id, game_id, unix_time, x, y, x - lag(x) OVER (PARTITION BY game_id, player_id ORDER BY unix_time) x_distance, y - lag(y) OVER (PARTITION BY
    game_id, player_id ORDER BY unix_time) y_distance FROM raw_locations) LHS) LHS2
    GROUP BY player_id, game_id) LHS LEFT JOIN raw_players RHS ON LHS.player_id = RHS.player_id) AGG WHERE player_id <> -1 ORDER BY total_distance DESC LIMIT
50;

```

Spark Query

```

from pyspark.sql.types import *
players_raw = sc.textFile("s3://dcproject/2014-10-29/players/*.csv").repartition(100)

def parse_players(line):
    fields = line.split(",")
    player_id = int(fields[0])
    team = int(fields[5])
    player = fields[3] + " " + fields[1]
    position = str(fields[4])
    jersey = int(fields[2])
    return player_id, team, player, position, jersey

print players_raw.take(2)
players_info = players_raw.map(parse_players)

fields = [("player_id", IntegerType()), ("team_id", IntegerType()), ("player_name", StringType()), ("position", StringType()), ("jersey", IntegerType())]
player_schema = StructType([StructField(x[0], x[1], True) for x in fields])
schema_PlayersInfo = sqlc.createDataFrame(players_info, player_schema)

```

```

sqlc.registerDataFrameAsTable(schema_PlayersInfo, "raw_players")

locations_raw = sc.textFile("s3://dcproject/2014-10-29/moments/*.csv.gz").repartition(100)

def parse_locations(line):
    fields = line.split(",")
    game_id = int(fields[0])
    quarter = int(fields[1])
    unix_time = int(fields[2])
    try:
        game_clock = float(fields[3])
    except:
        game_clock = 0.0
    try:
        shot_clock = float(fields[4])
    except:
        shot_clock = 0.0

    team_id = int(fields[5])
    player_id = int(fields[6])

    try:
        x = float(fields[7])
    except:
        x = 0.0
    try:
        y = float(fields[8])
    except:
        y = 0.0
    try:
        z = float(fields[9])
    except:
        z = 0.0
    return game_id, quarter, unix_time, game_clock, shot_clock, team_id, player_id, x, y, z

locations_raw = locations_raw.map(parse_locations)

types = [("game_id", IntegerType()), ("quarter", IntegerType()), ("unix_time", IntegerType()), ("game_clock", FloatType()), ("shot_clock", DoubleType()), ("team_id", IntegerType()), ("player_id", IntegerType()), ("x", DoubleType()), ("y", DoubleType()), ("z", DoubleType())]
location_schema = StructType([StructField(x[0], x[1], True) for x in types])

schema_LocationInfo = sqlc.createDataFrame(locations_raw, location_schema)
sqlc.registerDataFrameAsTable(schema_LocationInfo, "raw_locations")

y = sc.parallelize([float(x) for x in range(51)])
x = sc.parallelize([float(x) for x in range(95)])

fields = [StructField("x", DoubleType(), True), StructField("y", DoubleType(), True)]
null_location_schema = sqlc.createDataFrame(x.cartesian(y), StructType(fields))
sqlc.registerDataFrameAsTable(null_location_schema, "court")

query = """
SELECT * FROM (
SELECT game_id, LHS.player_id, CONCAT(first_name, " ", last_name), position, total_distance FROM (SELECT game_id, player_id, SUM(distance) total_distance FROM
(SELECT player_id, game_id, unix_time, SQRT(POW(x_distance, 2) + POW(y_distance, 2))*0.000189394 distance FROM
(SELECT player_id, game_id, unix_time, x, y, x - lag(x) OVER (PARTITION BY game_id, player_id ORDER BY unix_time) x_distance, y - lag(y) OVER (PARTITION BY
game_id, player_id ORDER BY unix_time) y_distance FROM raw_locations) LHS) LHS2
GROUP BY player_id, game_id) LHS LEFT JOIN raw_players RHS ON LHS.player_id = RHS.player_id) AGG ORDER BY total_distance DESC LIMIT 25
"""
print sqlc.sql(query).take(25)

```