Analyzing (Baseball Data) for Streaks Using MapReduce

Ross Anderson

# Introduction

# Data Source

Data for this project was obtained from retrosheet.org. The data consists of historical play by play records from Major League Baseball games. The data goes as far back as 1940, however, prior to 1974, the records may be missing one or more games per year, so the records cannot be considered 100% complete,. Also, as the data does not go back to the start of Major League Baseball, they cannot be considered “official” and any records could not be considered official due to the data discrepancies.

# Data Pre-Processing Required

The data obtained from Retrosheet consists of an entire year’s worth or records in a zip file. The files consist of:

* One Event file for each ballpark (contains all of the game records that occurred in that ballpark)
  + File is named <4 digit year><3 Letter Team Abbreviation>.EV<League>
    - File extension “.EVA” for American League Teams
    - File extension “.EVN” for National League Teams
* One file for Translating Team Abbreviations to League and Team Name (Not used at this time)
  + File is named TEAM<4 digit year>
* One Roster file for each name for translating Player Ids to Player Names, as well as some characteristics of each player (Batting/Throwng hand, position)
  + File is named <Team Abbreviation><4 digit year>.ROS

To transform the event files provided into a CSV format that expands the records out to a more table oriented format, the utility BEVENT has been provided by retrosheet.org. For ease of use, I created a python script (in GitHub directory scripts/Retrosheet-Extract-Transform.py). This script unzips each year file, runs the BEVENT command and creates a <4 digit year>.EVAC file for American League teams, and <4 digit year>.EVNC file for National League teams.

# Data Description and Schema

After running the BEVENT command to expand the data, the data consists of a 96 column CSV file. The 96 columns represent batter/pitcher/each fielding position, score, outs, and numerous flags to indicate if the event was designated as an “at-bat”, if a double play occurred, if the record represents an plate appearance (steals, pickoffs, balks, and wild-pitches cause the plate appearance to be repeated to reflect the movement of runners). Full details of the column identifiers, and descriptions of how to use them are available at this path: <http://www.retrosheet.org/datause.txt> (See the BEVENT heading).

<Insert Diagram of Schema>

# Bad Data Issues

The only observed issue was while extracting data using the BEVENT utility, 4 instances of “too many assists on play” were reported. I did research one instance of this, and its an example of how difficult it can be to model such data. On 9/15/11, in the 4th inning of the Kansas City Royals/Chicago White Sox game, a double play was recorded with what appears to be 6 fielders getting an assist on what one can only assume was a hot box created where the players rotated through before tagging the runner. This issue would have been flagged as the resulting CSV file only has 5 columns for assists.

# MapReduce Algorithm/Architecture

The MapReduce architecture used consists of two MapReduce jobs. The first job is responsible for extracting data that could contribute to a statistical streak, and then analyzing the data for a streak. The second mapper sorts the streaks to obtain the top ones.

<High Level Diagram>

## Find Streaks Job - Mapper

Each record is read in (as mentioned earlier, the files are formatted as such that all events that took place in a American League park are in one file, all events that took place in a National League park are in a different file. As interleague play is possible in the years since 1997, the events need to be checked against both files for continuity of streaks, and to handle player moves). As each record is read, its confirmed to be the actual plate appearance. If it is the plate appearance, it will check for different types of events (Walks, Strikeouts, Hits, Extra Base Hits, Hit into Double Play, etc). Currently this only tracks a batters statistics, but a simple extension could include pitching, fielding or base running data. Any noted event is sent to the mapper, as well as each plate appearance. This is important to send, as if no plate appearance in a game for a given player recorded a particular event occurring, that would signal the end of the streak. The data is sent to the reducer as a composite key. The next paragraph details the composite key. The value sent to the reducer is a simple text key of the statistical event (Strikeout, Walk, Hit, etc).

## Find Streaks Job - Composite Key

To align with how the Reducer is setup, all of the records sent to the reducer need to be processed in sequential order. To accomplish this, and to assist in maintenance, a composite key was used. The composite key class consists of the Game Identifier, Player Id, Inning, and Score (Although currently unused, a unique value would need to be included to handle situations of batting around the order, the score would be sufficient to distinguish one plate appearance from another in the same inning). To ensure the records are sorted correctly for the reducer to find a streak, a custom compareTo() was defined to sort by: Player Id, then Game Id, then Inning, then Score.

## Find Streaks Job - Partitioner

To align with how the Reducer is setup, either all records would need to go to a single reducer, or records would need to be dispersed in such a way that all of the records for a given player were at the same reducer. To provide scalability, and usage of available resources, a partitioner was created to hash the Player Id field of the composite key output by the Mapper. The hash is modded by the number of reduce tasks, and helps to provide a more even distribution of the workload.

## Find Streaks Job - Reducer

The reducer of the streaks operates on the basis that all records arrive sorted by player, then game event. Each record compares the current player against the last player. If there is a difference, all of the actively tracked streaks need to be flushed as there will be no more records for the previous player. Then, each record compares the current game event to the last game event. If there is a difference, the previous game is no more, and each actively tracked streak needs to check for continuity. If the previous game does not match the last recorded event for the streak, that’s the flag to indicate the streak is over, and should be recorded. After establishing the player/game combination, it’s a matter of walking through each record for the combination, and updating any streaks. This leaves the last record processed by each reducer at risk of not being included/reported. To combat this, the cleanup() method was populated to flush any streaks that were active at the end of record processing.

The reducer was structured in such a way it uses the Value as the tracker. Any type of event the mapper can extract and place into the Value, the reducer can handle. This provides some flexibility for future growth in the reducer could be left as is, while the mapper could be extended to handle other items.

Once the mapper outputs a streak, its output as a Text key, and a NullWritable Value. The text key consists of <Event Name> : <Count of occurrences, 0 padded to 5 digits> : <Start Date - Finish Date> : Player Id. It should also be noted, that more than one occurrence had to be tracked for event to be output.

## Sort Streaks Job - Mapper

The sort streaks mapper is fairly straightforward. It is configured to take the incoming data as the TextInputFormat class. This is fitting as the Find Streaks Job Reducer output data as a Key only. As an alternative, a custom class could have been output into a sequence file, and processed here. As the data was viewed to be straightforward, and not highly susceptible to change, it was left as just Text. The net effect of the mapper is to take the values passed in, and move them to the Key, with the Value being NullWritable.

## Sort Streaks Job - Partitioner

To distribute the workload of outputting the streaks, a partitioner was employed to extract and hash the Event Name field, causing all of same events to be sent to the same partitioner. The necessity of using this partitioner vs., using a single reducer is debatable. There are currently about 1.3 million records processed by this job, and with 48 reducers, the reduce portion of the job appears to take less than 2 seconds. It should also be noted that even though there are 48 reducers available, the use of the partitioner on 14 different types of events, drives a maximum of 14 reducers to be used, largely defeating the purpose of using reducers as most are left to idle. If more stats were to be tracked, this may become more important, but at the current time, this is probably excessive for the amount of work accomplished.

## Sort Streaks Job - Custom Text Comparator

The way the Reducer was structured was to operate on the basis that the longest streak should be the first record. The default sort will not accomplish this as its in ascending order (recall from the Find Streaks Job - Reducer, the key starts with <Event Name>:<Streak length 0 padded>…). To get the sort, a custom text comparator was created. This comparator simply runs the default Text comparator, but inverts the result by multiplying it by -1. As the streak length was 0 padded, this works out. If it weren’t, either a composite key would need to be used, or the custom comparator could be extended to extract the integer value and sort by that.

## Sort Streaks Job - Reducer

With records coming into the Reducer sorted so the longest streaks are on the top, the reducer just needs to walk through the received values, and output the largest ones. To handle this, local variables are used to track the last type of event that was received, and how many of that event were output. For each incoming record, the event type is checked against the last processed event type. If they are the same, then check how many times the event has been output, and if the last value output is the same as the current value. The check for how many times acts as a “top” type of statement, while the check of the last value allows a streak like: 10, 9, 8, 8, 8… . If the top 3 were requested, one of the 8 values would be output, but the others would not be output.

### Use of Distributed Cache

As the records at the reducer still use the Player ID provided by Retrosheet, this isn’t ideal for a list intended to be human readable as the user would either have to guess based on the first four characters of the players lastname, first letter of their first name, and if more than one has ever played or coached, the index based on who played first with the same designation.

The mapping is trivial, and with it just being Player Id->Player Name, it’s a good fit for the map data structure. As the only point these values matter is the final output, the Sort Reducer is the ideal place to map these names in as not nearly as many lookups would need to be made. Unfortunately, I was unable to get the distributed cache working in my implementation. After reviewing the book, and the web, I feel this area may still be lacking in the Hadoop structure as there seems to be a heavy dependency on filename/path used in the code lining up with the filename/path used when the file was added to HDFS or passed in as part of starting the job.

# 3rd Party Libraries / Tools

The resulting CSV files from the BEVENT utility included a double quote qualifier on each non-numeric field. After pondering this, I decided a regular expression wasn’t going to handle this robustly, so I searched the web, and found opencsv which is an open source CSV library that can handle such qualifiers (http://opencsv.sourceforge.net/).

For development, I used the ClouderaVM, and the standard eclipse installation in that environment.

# Output Description

# Verification of Output

# Performance/Scale Characteristics

# Lessons Learned

# Conclusions