The Universal Sports Database

Lawrence Chang
Advisor: David Martin

Table of Contents

Abstract

With vast amounts of data in the world, organization becomes a challenge. The success of data driven web services (IMDb, YouTube, Google Maps, Wikipedia, et cetera) all hinge on their ability to present information in an intuitive manner with user friendly interfaces. One area that fails to have such a service is sports statistics. With the ubiquitous appeal of sports, having a solution to this problem can be universally beneficial. Many sites exist that have statistics of different sports, but there are limitations to all of them. Since there is very little continuity among all sports, statistics are represented disparately.

There are several problems with this approach. Any time there needs to be a change to the informational structure, the entire database and interface need to change. In addition, there can never be a single interface if there are different schemas for different sports, leading to a user unfriendly interface.

My system uses a unique schema that is capable of representing statistics from any sport, no matter how unique. Adding new statistics to a sport to reflect rule changes or adding a new sport altogether are seamless. In addition, the web interface is structured by Rails, which changes automatically with the schema.

Challenges included developing a universal sports schema and testing it sufficiently enough to prove its generality. Finding and extracting the data to populate the database also presented difficulties.

Introduction

As an avid fan of sports, it seemed fitting to write a thesis that incorporated technology with my interests. One thing that sports fans care a great deal about are statistics. These numbers, when put into context, are able to give objective information about a sporting event: results that are undeniable and real. As a result, not only are there mainstream sports sites where most people obtain information on current sporting events, but numerous sports statistics database services as well.

After exploring the sports data services available, it became apparent that every sport was presented in its own way. Databasesports.com, a "comprehensive" service that provides information on multiple sports, is subdivided by sport. One could easily find who had scored the most points in basketball (NBA and ABA), but anything that crossed sport boundaries was impossible. For instance, someone may be curious to know who had the most number of stolen bases, whether it was from Major League Baseball, Dominican League, or Japanese League. This person would be forced to find the leader from each respective league and manually figure out the answer. Another individual may wish to know the most number of interceptions in an American football game, including the National Football League, Arena Football League, XFL and the Canadian Football League. Once again, since these are considered different sports (different rules, regulations, teams), the information could only be found by manual compilation. Finally, if one wanted to find out who was the oldest active professional athlete in any sport: that would be a task for the ages.

The problem is that there is no clear way to represent sports statistics in a unified way. Aside from the aforementioned examples where the sports are essentially the same, the majority of sports are vastly different from one another. Every sport needs to be put in its own database

schema as well as have its own interface. As a result, there is no singular database scheme and no single interface.

This project sets out to resolve this limitation. The task was divided into three main parts: 1) create the universal sports schema, 2) implement the schema using at least three sufficiently different sports, and 3) populate the database with relevant data to prove the concept.

Much of the inspiration came from the Internet Movie Database service, or IMDb. This web site offers information about the entire American entertainment industry, from movies, actors, directors, and television shows. It is the ultimate resource for all things Hollywood. The best feature about IMDb is that its data is organized in a perfectly intuitive way. Say one were curious to know who played the mother of the leading actress in the newest Peter Jackson movie. One could simply search Peter Jackson and find a list of all his works; from these, select the correct film; and finally, look on the list to see the actress. This project attempts to bring a similar ease to finding anything sports data. The ultimate goal is to have the "IMDb of Sports."

System Overview

Database: SQLite
Interface: Ruby on Rails
Automation: Ruby
Data scraping: Hpricot, Firebug

Schema

a. Current schema organizations

Certain forms of data naturally span all sports, namely those that are unrelated to sports at all. These include information like an athlete's name, birth date, birth place, height and portrait. It is the more specific statistic types that cause problems with unification.

Based on existing sports database interfaces, it is not too difficult to see how their data is organized. On a basketball player's page, one sees a list of seasons played. For each season there are a number of statistics (total points, total rebounds, games played, minutes, et cetera). For each season there are corresponding links to the team and the league (Los Angeles Lakers 2001, NBA 1993, etc). Figure 1 shows a sample player's page. From this information one can make a reasonable assumption about the database organization employed by this service.

The main tables are *player*, *team by year*, and *league by year*. These tables would be in a database named *basketball*. Statistics can be recorded in a variety of ways. Giving currently available services the benefit of the doubt, let us assume their data is organized to not have redundancy. For example, only data from individual games are recorded. Then, cumulative or computed statistics are figured out on the fly (assists in a particular game versus assists in a season). In this scenario, there needs to be a table corresponding to a *game* and one corresponding to a *statistic* such as points, rebounds, and assists. Figure 2 shows a schema representation of such an organization.

Regular Season Stats

| Year	Age	Team	Lg	G	Min	Pts	PPG	FGM	FGA	FGP	FTM	FTA	FTP	3PM	3PA	3PP	ORB	DRI
1996-97	18	LAL	NBA	71	1103	539	7.6	176	422	.417	136	166	.819	51	136	.375	47	8
1997-98	19	LAL	NBA	79	2056	1220	15.4	391	913	.428	363	457	.794	75	220	.341	79	16
1998-99	20	LAL	NBA	50	1896	996	19.9	362	779	.465	245	292	.839	27	101	.267	53	21
1999-00	21	LAL	NBA	66	2524	1485	22.5	554	1183	.468	331	403	.821	46	144	.319	108	30
2000-01	22	LAL	NBA	68	2783	1938	28.5	701	1510	.464	475	557	.853	61	200	.305	104	29
2001-02	23	LAL	NBA	80	3064	2019	25.2	749	1597	.469	488	589	.829	33	132	.250	112	32
2002-03	24	LAL	NBA	82	3401	2461	30.0	868	1924	.451	601	713	.843	124	324	.383	106	45
2003-04	25	LAL	NBA	65	2447	1557	24.0	516	1178	.438	454	533	.852	71	217	.327	103	25
2004-05	26	LAL	NBA	66	2690	1819	27.6	573	1324	.433	542	664	.816	131	387	.339	95	29
2005-06	27	LAL	NBA	80	3274	2832	35.4	978	2173	.450	696	819	.850	180	518	.347	71	35
2006-07	28	LAL	NBA	77	3142	2430	31.6	813	1757	.463	667	768	.868	137	398	.344	75	36
2007-08	29	LAL	NBA	82	3193	2323	28.3	775	1690	.459	623	742	.840	150	415	.361	94	42
12 Season Totals				866	31573	21619	25.0	7456	16450	.453	5621	6703	.839	1086	3192	.340	1047	354

Figure 1
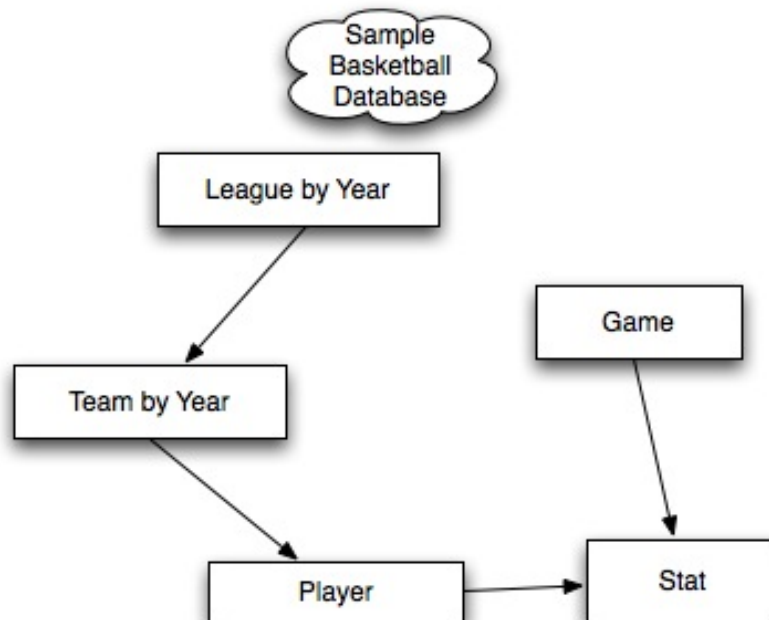
Figure 2

b. Proposed schema organization

Such existing models work fine when dealing with a single sport. However, imagine if the service wished to provide information on another sport, such as tennis. The programmers would have to create an entirely new database with new tables and a different organizational structure. In basketball, all games are played by two teams. In tennis, there are both singles and doubles matches. Would this lead to having two different tables representing a tennis match, perhaps *singles_match* and *doubles_match*? And what of the teams? Singles matches are one on one, where as doubles are two on two. How should games and teams and individuals be organized? There are numerous differences between the sports that create difficulty in terms of coherent data organization. How would this service deal with representing marathons, stock car racing or horse racing? The more varied sports become, the greater the difficulty of sustaining a sports database service.

Approaching this problem from the standpoint of database schema design, it lends itself to have a table representing a person or player. A team table is needed. A team, by definition, can have one or more players. Therefore, my schema could enforce the rule that all games are between teams. If a team represents an individual, conventions can be implemented to account for this, such as having a team with the same name as an individual or having a team with a null (nil in Ruby) name. Since a team can have multiple players, and a player can be a part of multiple teams (different seasons), an intermediary table needs to exist, perhaps called rosterspot. Teams can be associated with leagues (Los Angeles Lakers as a member of the NBA), but for simplicity I chose to neglect different leagues and assume all teams to be part of their respective sports (Los Angeles Lakers as a member of sport Basketball).

Dealing with the organization of personnel and sports and teams is not very difficult. To create the aforementioned scheme only one level of abstraction is needed: by having individuals as their own "teams". The problem becomes significantly more difficult when dealing with statistics from different types of sports. For Americans, the traditional sports involve scoring more points than the opponent. Then there are the sports where the time is the most important factor. And in others, placement determines the winners.

After much deliberation between my advisor and myself, the schema was produced provides enough abstraction to account for any sports statistic. The highest level abstraction is an object called *unit*. The *unit* table is used to describe what kind of statistic something is. The unit will denote how to compare other things of the same type. The most basic unit is *count*. A count denotes a statistic that is simply an accumulation. In American sports the most notable statistics are ones that are counted, such as home runs, touchdowns, and points. To encompass this unit with something that more accurately describes the statistic, we create the *statconcept*. A statconcept is an object that ties together a unit and sport. Home runs, goals, and yards are all examples of statconcepts. In traditional database terms, a statconcept has the foreign key to a unit. Finally, an actual statistic, which we will call *stat*, will record the value. A stat object will reference a statconcept object, a game object, and a player object. Figure 3 shows these relationships.
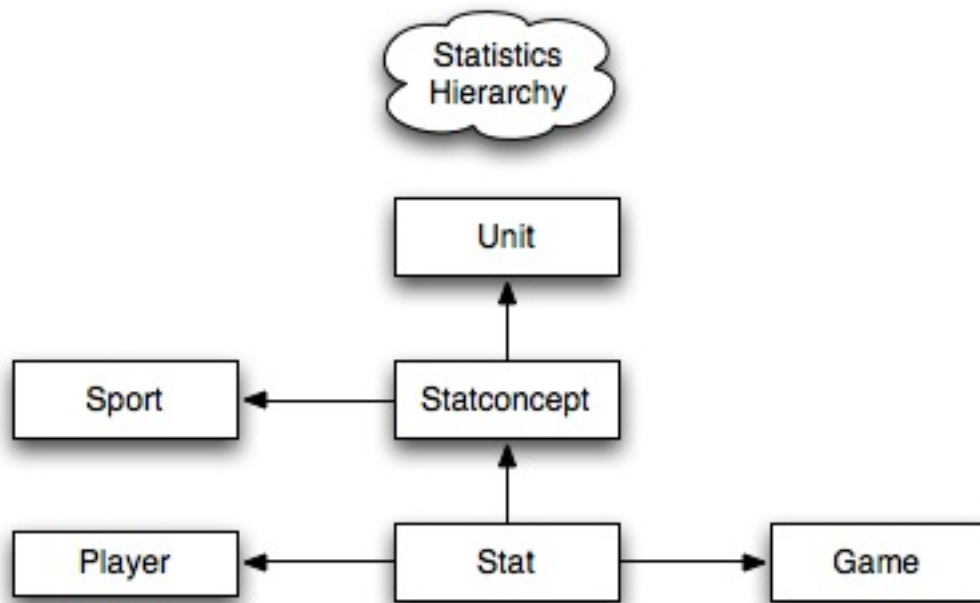
Figure 3

Using this schema, any numerical value can be recorded and reflected as a sport statistic.

Counts are obvious and trivial. With times, the units can be in seconds: minutes and hours can

be computed on the fly from this data. Placement is similar to count except lower numbers are
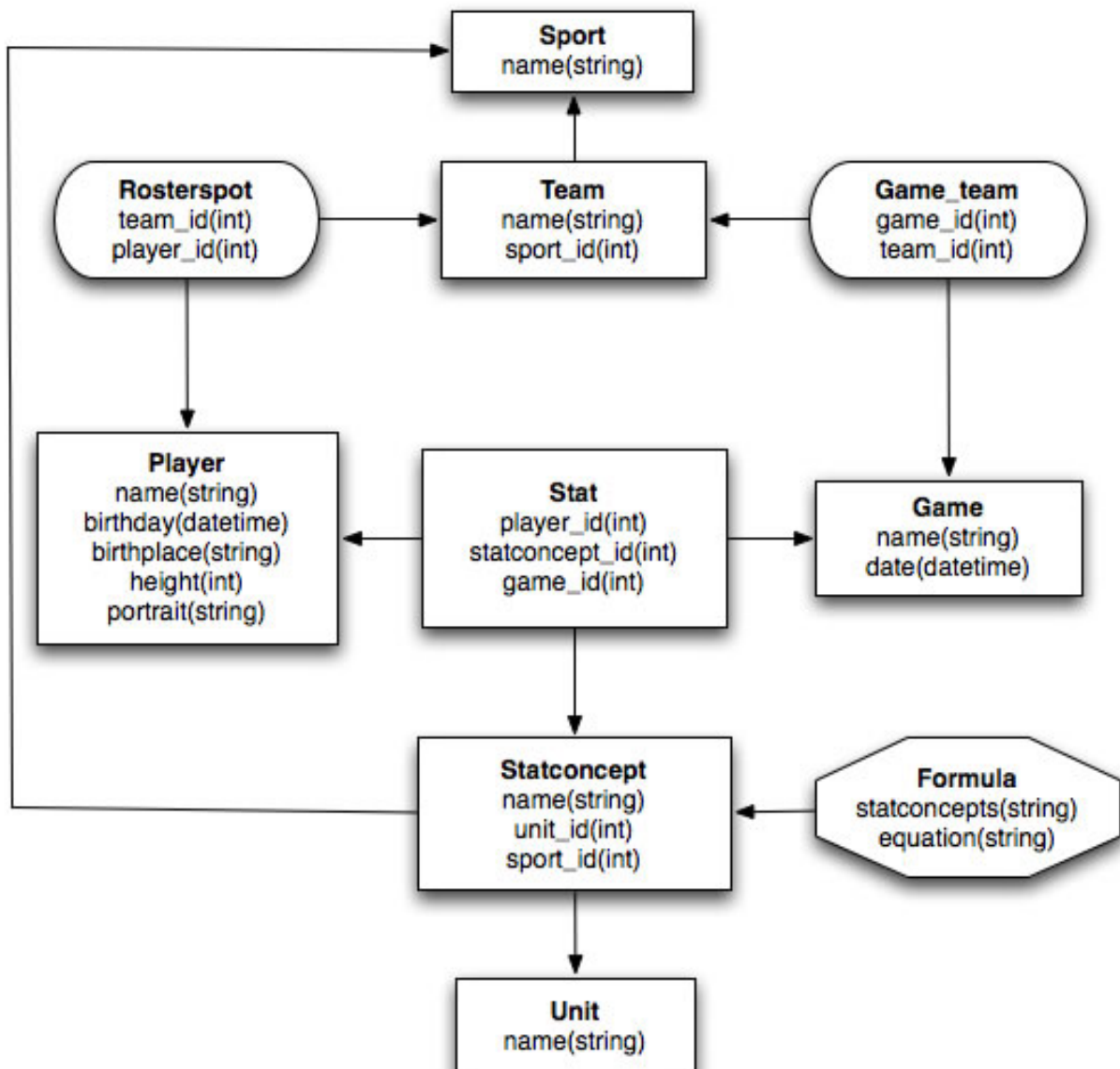
better.

Figure 4 shows the entire schema.

Figure 4

Implementation

To ensure the schema's universality, statistics were implemented for the sports basketball, tennis, and stock car racing. Stats composed of two or more other stats, or 'derived stats', are discussed at the end of this section.

Basketball is the more traditional among the three chosen sports in terms of statistics. The most common statistics associated with basketball are: points, minutes, field goals made, field goals attempted, free throws made, free throws attempted, three pointers made, three pointers attempted, offensive rebounds, defensive rebounds, assists, blocks, turn over's, and personal fouls. All of these statistics can be classified by the unit "count" since they are measured by counting the number of occurrences and compared strictly numerically. Statistics of these types can be associated with Statconcept's identified simply by the name. There is a Stat with the value '81', Player named 'Kobe Bryant', and Statconcept named 'points'. The Statconcept 'points' would be associated with sport 'Basketball' and Unit 'count'.

Attempting to create a statistical structure around the sport tennis poses several challenges. Like basketball, there are various aspects of the game that can be recorded by simply counting the number of occurrences. These include serves, aces, double faults, winners, at-net wins, at-net attempts, et cetera. However, these statistics are less important to tennis fans and thus rarely recorded past the lifetime of the match (game means something different in tennis). What is recorded are set scores and tie breaker scores.

Using the same strategy as basketball statistics does not work. There is no singular number to represent a set score for a match because there are varying numbers of sets. The method to solve this problem is to add a level of abstraction by dividing a match into distinct sets by number. A tennis match has at least one set played and at most five sets. A set either has a tiebreaker score or does not. Using these facts, all singles set scores can be represented using ten

Statconcept objects (five for set scores and five for tie breaker scores): SinglesSet1, SinglesSet2, etc, SinglesSetTie1, SinglesSetTie2, etc. These would correspond to unit object 'count'. Another ten would take care of doubles matches as well. Let us take a recent match between Rafael Nadal and Nikolay Davydenko at the Monte-Carlo semifinals as an example. Nadal won the best-of-three match in two sets with scores of 6-3 and 6-2. Each match is given a unique gameid, so we will use this as the name of the match (Game). The Stat's corresponding to the first set would be: Stat(1).player = Nadal, Stat(1).value = 6, Stat(1).Statconcept = SinglesSet1, Stat(2).player = Davydenko, Stat(2).value = 3, Stat(2).Statconcept = SinglesSet1. The second set is trivial: Stat(3).value = 6, Stat(3).Statconcept = SinglesSet2, etc. Both Stat objects would share the same 'game', perhaps Stat(1).game = Game.find_by_name(g), where g is the name of the match. If there had been a tiebreaker in the second set with score 7-4, there would be additional Stat objects to represent the data: Stat(4).Statconcept = SinglesSetTie2, Stat(4).value = 7, Stat(4).game = Game.find_by_name(g), Stat(5).Statconcept = SinglesSetTie2, Stat(5).value = 4.

Stock car racing's main body is the National Association for Stock Car Auto Racing, or NASCAR. There are various NASCAR racing series, the biggest being the Sprint Cup Series. During each race, a driver accumulates points that go toward winning the cup, as well as prize money. Times are not important; only the final position matters. Several statistics are numerical and can be 'count' stats. These include laps, money, points, laps led, bonus, and penalty. Then there are the statistics of position. We introduce a new Unit called 'position'. These stats are final position and starting position. Finally, various non-competitive "statistics" exist in NASCAR due to the nature of the sport and sponsorships. These are car number and car manufacturer. Another new Unit to account for these can introduced called 'non-competitive'.

It is trivial to understand how to implement the countable NASCAR statistics. Using unit 'position' define Statconcept objects 'start_position' and 'final_position'. At the 2008 Talladega 499 race, Dale Earnhardt Jr. started in pole position 9 and finished in final position 10. The Stat objects associated with these are: Stat(1).start_position = 9, Stat(2).final_position = 10. In addition, Earnhardt Jr. drives the number 88 race car made by Chevrolet. Using Unit 'non-competitive', we define Statconcepts 'car_number' and 'car_make'. The corresponding stats can be Stat(3).car_number = 88, Stat(4).car_make = 'Chevrolet'. All of these Stat objects would be tied to a Game object named for the event such as 'Talladega 499'.

Even with significantly different types of data, this schema is able to represent all statistics effectively. As previously mentioned, this schema has not yet handled statistics based on combinations of multiple statistics. These include common statistics such as a baseball player's batting average, basketball player's shooting percentage, or the ever useful baseball pitcher stat WHIP, which stands for Walks plus Hits per Inning Pitched. In staying with the spirit of not duplicating any data, these statistics must be derived from already existing data. Statconcepts that are of this form have another Unit type, thus we introduce 'derived'. However, this new unit does not tell a machine how to compute a derived stat. The name of the Statconcept should remain something humanly intelligible ('Batting average', 'WHIP', etc). To have a machine-intelligible representation of derived statistics, we create two new models: Formula and Formula_SC, which acts as a join table. Formula will have a string representation of a mathematical formula. Given a derived stat defined by three variables a,b,c and formula (a+b)/c, this can be translated to the string ". . + . /", a postfix representation. The dots correspond to the variables. The order in which to use the variables lies in the Formula_SC model. These objects are defined as "acts_as_list" in the model. Using the current example,

variable *a* is be defined as first, *b* as second, and *c* as third. When computing the derived stat, the corresponding Formula_SC object would be referenced whenever a '.' was encountered in the Formula's string field. Methods used would include *move_to_top*, which goes to the first of the ordered objects, and *increment_position* to move to the next object.

Let us take the WHIP statistic as an example. The Statconcept objects necessary are WalksAllowed, HitsAllowed, and InningsPitched. The Formula can be defined as 'WHIP' and have a string field that holds ". . + . /". The Formula_SC objects all have formula_id = Formula.find_by_name('WHIP'). The first object has statconcept_id = Statconcept.find_by_name('WalksAllowed'), the second by 'HitsAllowed' and third by 'InningsPitched'. These objects can be put into the proper order using the *insert_at* method.

Data scraping

To thoroughly test the schema it is crucial to use large data sets. To do this the schema was applied to three sufficiently varied sports' statistics. The three chosen were basketball, tennis, and stock car racing (NASCAR). (At the time of writing this I have only completed data entry for basketball and tennis.)

Several sources were reviewed for possible data collection. Most obvious were the existing database services on the web. However, many were limited to certain sports like basketball, football, baseball and hockey. There was also no guarantee about the accuracy of their data nor the timeliness of their updates. For accuracy, professional services like the Elias Sports Bureau were considered. ESB turns out to be too professional; their services are not freely available to the public. The third and eventual choice was ESPN.com. ESPN is renowned for their up to date and accurate sports information (they get their statistics from ESB). Another feature is the

breadth of their coverage, providing information on all the world's most popular sports. It was the best place to get the necessary information.

It was clear that an automated system was needed to collect new data while going back to older records for past data. One convenient feature about ESPN's stat pages is that they have links to navigate forward or backward one day. With this knowledge in hand, it was only a matter of writing a logical program flow that extracted data, inserted into my database, then moved to a different day. Figure 5 shows an example of a working script.
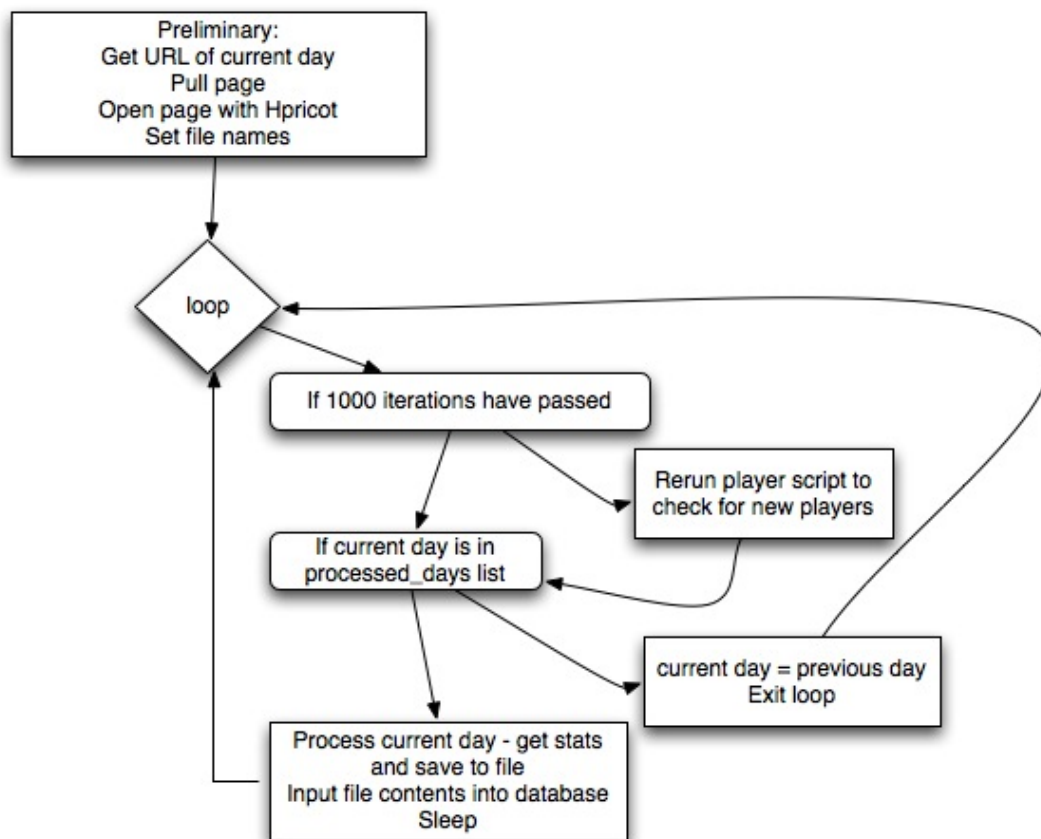

Figure 5

Data was extracted from web pages using Ruby's Hpricot library. The required libraries are rubygems, hpricot, and open-uri. An XML page is pulled from the designated URL and broken into the underlying tree structures. With this, XPath's (XML Path Language) can be specified to

reach specific parts of the XML tree. Because each statistics page is generally composed

identically, noting the XPath values on one page will usually correspond to the same data fields

on another page. The XPath's were figured out using a Firefox plug-in named FireBug. FireBug

breaks XML into a directory tree structure that shows where certain data is located.

Because Rails is used as the web application framework, coupled with the existence of

Hpricot, the scripts are written in Ruby. Very similar to the free flowing nature and simplicity of

MatLab, concepts can quickly be translated to code. Parts of the script pull and analyze the

HTML pages where data lie. The data (statistics) collected are then put into an XML file with

specifically chosen structures. Finally, another script is called and piped into the Ruby console

to read this file and input the data into the Rails database. This other script uses Rails commands

to create or edit the necessary data fields, as shown in Figure 6.

```
Stat.create :value => val, :game_id => Game.find_by_name(game).i
d, :player_id => Player.find_by_name(player).id, :statconcept_id => Statconcept.
find_by_name(sc).id
```

Figure 6

Engineering

Several requirements had to be met to produce a viable automated system:

1. The script can start at any time and not produce false data.

2. The script can be run over and over without issue.

3. The script can be stopped at any time.

To ensure that any automated script could start at any time, the initial starting URL is hard

coded into the program. In my master_tennis_script.rb program, the default starting URL is

ESPN's daily scores page (http://sports.espn.go.com/sports/tennis/dailyResults). Starting on the

current day, the script then moves backwards in time to all previous days. The URL for the

previous day would look something like …dailyResults?date=20080426, which would make the

current day URL equivalent to …dailyResults?date=20080427 (notice the date at the end). Once

the data from a day has been recorded, the date itself is saved into a text file. The script uses this text file to determine whether a date has been seen or not. By using *system(command)*, the script can easily call grep –q to determine whether a string exists in the file or not. The –q option returns a true or false result to the script.

There are various methods databases employ to validate data. Traditionally the checks are performed either by the interface code or internally within the database. Both methods are cumbersome and impractical. If the validation lives in the interface, business logic becomes public. If it lives in the database, any changes to business logic would require manually updating the database using SQL. Using Rails' MVC (Model-View-Controller) architectural pattern, all business logic can exist within the model, thus hiding it from people who should not have access to it while making it easy to edit if need be. With these safe guards in place inside the models, running the same script over and over will not result in duplicate data being saved to the database. Of course, this relies on the correctness of the logic. A good example of this logic can be shown through the Stat model. Individual statistics are data that have the greatest risk of duplication. If a script is run to capture last week's data, then run again to capture all data starting from this week, would last week's information not get doubled in the process? Figure 7 shows the code for the Stat model.

```
class Stat < ActiveRecord::Base
        validates_presence_of :player_id
        validates_presence_of :game_id
        validates_presence_of :statconcept_id
        validates_presence_of :value

        validates_uniqueness_of :statconcept_id, :scope => [:player_id, :game_id
]
end
```

Figure 7

A Stat object must tie in three different objects: player, game, and statconcept. This is accomplished by validating their existence whenever a Stat object is created. It also verifies that there exists a value because having a statistic without a value makes no sense. Using the three

other objects, the 'validates_uniqueness_of' line guarantees that no other stat object will have the same combination of sport, statconcept and game. The line literally states that there can be only one statconcept for a player in a given game. For example, a basketball player can not have two different point totals for the same game. These validations will prevent any duplicate data from being entered into the database, thus ensuring that the script can run over the same data multiple times and will input only unique data.

The issue of stopping the script is minimal but necessary to maintain an accurate database. If a certain day has been processed but the script is killed before the date is recorded, then the day will get processed again. This is not incorrect behavior since the Stat validations will maintain data integrity. If the program is killed during the Rails console data input, the behavior is still correct because re-running the script would get any missed data. The only problem that may arise is if a day is recorded as processed but the script is killed before it actually is. All that is needed to prevent such behavior is to always perform the action before recording it.

Interface

The interface is a web based tool that creates dynamic web pages using the database. This is all done by Rails. With the object oriented nature of Rails, the code to find specific data for display becomes trivial. Figure 8 gives some examples of using Rails objects to get specific data values:

> To get a player's birthday: *Player.find_by_name("Bryant, Kobe").bday*
>
> To get all stat's of a player:
>
> *Stat.find_all_by_player_id(Player.find_by_name("Armstrong, Lance").id))*

Figure 8

These methods can be called directly by the controller. The view corresponding to the controller then has access to all of its variables. The views themselves are able to execute ruby code, much like embedding PHP into standard HTML. Using standard for-each loops, a view can loop through and display a data from any list.

Problems

a. Hpricot and Firefox/Firebug normalization

One of the things that the Firefox browser does is perform HTML normalization. This process is necessary because not all web pages are formed correctly, thus Firefox compensates by inserting additional structure into the XML tree. Firebug, which is a plug-in for Firefox, then uses this normalized HTML to compute the XPaths.

The problem with this is that Hpricot does no such normalization. Instead it pulls pages directly from the server and uses them unaltered. Therefore, not all XPaths derived using Firebug will work in Hpricot. This issue threw me off for a while. I was also unclear on XPath definitions such as div classes being defined by div.classname, while div id's were defined by div#idname. After some helpful assistance from my advisor I was able to get Hpricot to work without using Firebug's auto generated XPaths but by manually looking at the tree structure to find the data I needed.

b. ESPN versus other sources

Although ESPN is the most comprehensive site I found in terms of breadth, they are not the best source when needing to collect data older than a few years. For basketball game scores, if the URL has a date that was not recorded (?date=19980809) the page automatically redirects to the current day's page. If I chose to use a more data oriented web site I would lose out on the breadth of sports.

c. Learning Rails on Ruby

Everyone knows that Ruby on Rails is the new way to develop data driven web services. People even go so far as to say that the days of PHP and MySQL embedded into HTML are over. What they fail to realize about Rails is that it is not the easiest thing to learn to use. Any programmer can easily pick up PHP and SQL to start pumping out web pages. With Rails, there is an enormous amount of structure and behind-the-scenes activity. Rails provides a great tool set to create, but learning to use the tools comes with a steep learning curve.

To start using Rails I followed numerous tutorials found online and in books. These have limited value. They are great to learn how to build simple applications that are similar to the tutorials, but do not provide enough information for someone to truly develop independently. That was the problem I faced. I could probably build a cookbook/recipe site with my eyes closed, but what I wanted to do was make a user friendly sports database.

After struggling to produce a few tangible results with Rails, most of which were basically enormous hacks, I shifted gears and began the data import/export process. During this time I had to become very familiar with Ruby since I was writing scripts and using Hpricot. Much of my work was done in the Ruby console, an environment very similar to a MatLab command line. After becoming comfortable with Ruby I found it much easier to start solving problems using Rails. The one advice I have for anyone attempting to learn Ruby on Rails is to start with Ruby. I do not think it is enough to just understand the syntax and loop structures of Ruby; one should be very comfortable with it.

d. The novelty of Ruby on Rails

Because Rails is still relatively new, significant changes are made to it by its core team frequently. This results in several obstacles for a beginner. First, there are fewer sources online

on particular bugs or problems. Second, tutorials get out of date quickly, thus the learning curve becomes steeper than it already is. Finally, there are not many people around who are experts on it, making it that much more difficult to learn.

Future Work

In order to have a convincing result about the capability of my universal schema, it is crucial to test it rigorously. Perhaps the testing should be perpetual since new sports can always be added. How can one be sure that every sport fits into the schema?

One of the most pressing areas in need of improvement is the data collection strategy. What the system is currently doing with ESPN is illegal, especially the areas where it hot-links player profile pictures straight from their servers. I still think a professional service like Elias Sports Bureau is the best place to get guaranteed accurate numbers.

Obviously there needs to be more than three sports supported. Once a reliable source of information can be found the system must be upgraded to support all popular sports. Thinking in terms of a business model, there can be a team of developers constantly updating the database schemas and adding new content for new sports.

The interface can be turned into something more like that of IMDb. There will be a search field. Any item can then be used to navigate (player names link to player profiles, team names point to team profiles, et cetera). There are various features that I wish I had the opportunity to implement. One is a side by side comparator, where two players can be compared stat by stat. Another is an advanced search tool. In this tool one could find very specific information such as "who hit more than three hundred home runs in their career?" Any sort of

statistical information desired could be obtained by using this tool. It would be the bread and butter of statistics fiends.

It would be interesting to see if some major sports authority (ESPN, Sports Illustrated) were interested in taking on a project like mine. They definitely would have the resources to make it happen.

In the end, I would like there to be a system like I initially envisioned: a comprehensive, intuitive sports database service comparable to IMDb.

References

databaseSports.com: The Largest Sports Statistics and History Database Online, <http://databasesports.com>

Entertainment and Sports Programming Network, <http://espn.go.com>

Fowler, *Rails Recipes*, Pragmatic Bookshelf, June 9, 2006

National Association for Stock Car Auto Racing, <http://www.nascar.com>

Ruby on Rails, <http://www.rubyonrails.org>

Thomas, Heinemeier Hansson, *Agile Web Development with Rails: Second Edition*, Pragmatic Bookshelf, December 14, 2006