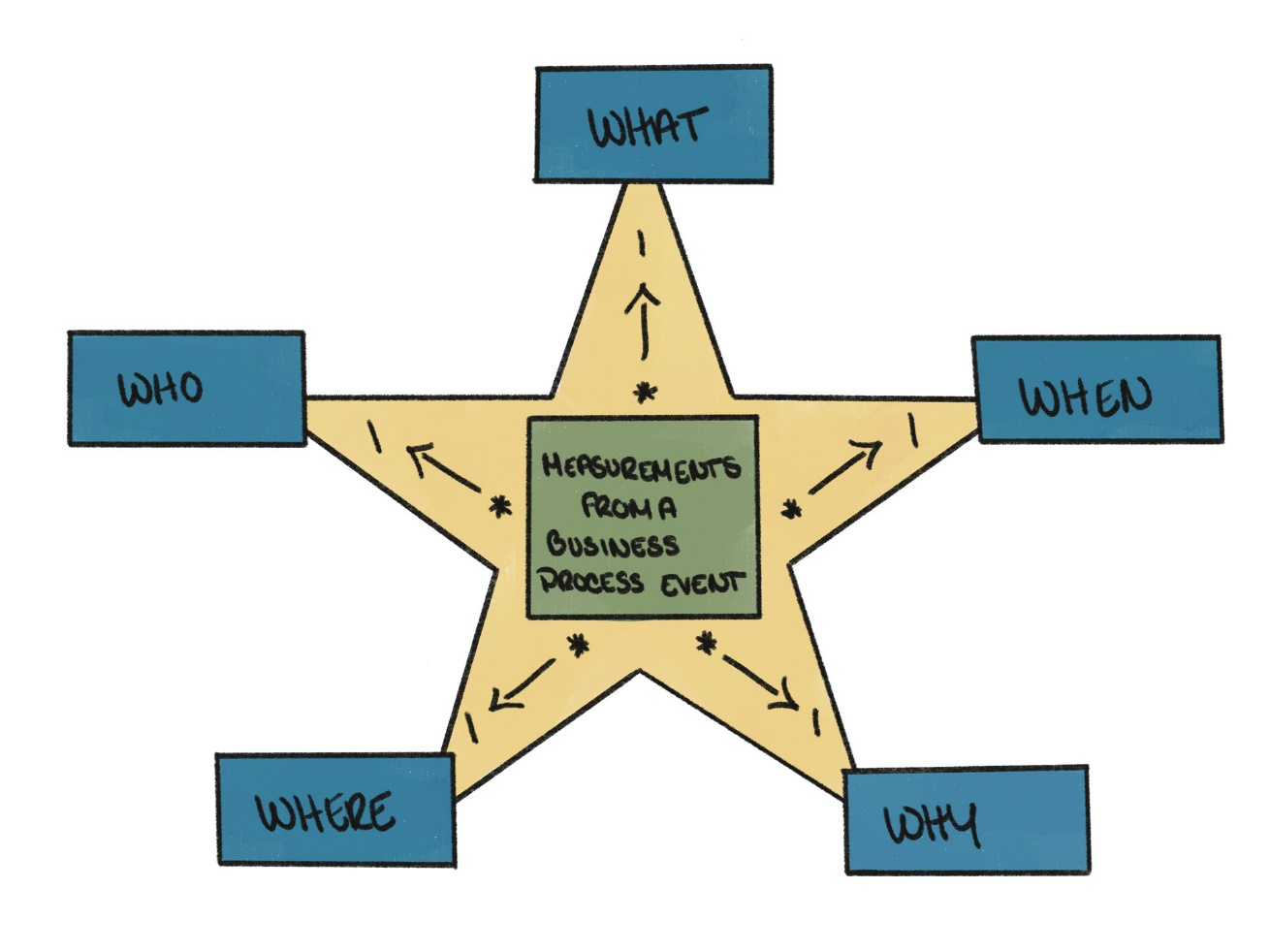
**Stars and Dimensions**

**Data Modeling for the Analytics**

Feb 25, 2020 | Bushra Anjum

Data modeling required for the online transactional processing (OLTP) systems’ databases is inherently different than the data modeling needed for analytics supported by a data warehouse. **The data modeling for the analytics is called “Dimensional Modeling.”** So what is this technique, and why do we need it? Let’s explore.

The concept of dimensional modeling was developed by *Ralph Kimball* and is relevant to any data practitioner in today’s data ecosystem. Dimensional data modeling is a database design technique used to restructure existing data for **easy and optimized querying**. The data model primarily consists of two types of tables, *fact tables* and *dimension tables*. A fact table contains facts, which are measurements from a business process event and correspond to an observable physical event such as a sale, purchase, processing, snapshotting, engagement, etc. Facts could be additive, semi-additive, or non-additive. A dimension table provides the context surrounding the business process event that generated those facts. Thus dimension tables have descriptive attributes and define “the 5 W’s and the H” (Who, What, When, Where, Why, and How) of the event.



Dimensional modeling results in **more intuitive data structures** for writing queries with good performance. The modeling principles, such as discussed [here](https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/kimball-techniques/dimensional-modeling-techniques/), make exponentially complicated data simple to use for a wide range of business problems and technical expertise. Each model, or star schema, is self-contained and self-explanatory. Hence interested **users of any technical background** can comprehend and run analytical queries against them without having to understand a mesh of normalized tables.

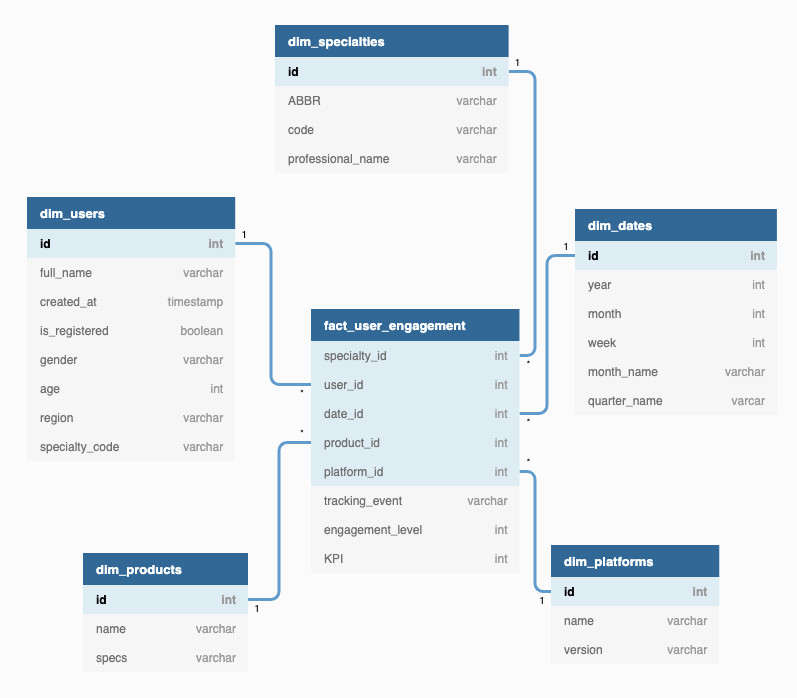
Relational data models, such as used by MySQL, Microsoft SQL Server, Oracle DB, IBM DB2, are optimized for real-time CRUD (create, read, update, delete) operations while ensuring ACID (atomic, consistent, isolated, durable) guarantees. Whereas, dimensional models are focused on user understanding and reduced data latencies while manipulating large volumes of data. For an easy reference, below are some of the differences between the transactional relational model and the dimensional model.

| **Transactional Relational Model** | **Dimensional Model** |
| --- | --- |
| Normalized (3NF) | Denormalized |
| Little to no redundant data | Redundant hierarchical data |
| Reflects the most current state | Requires historical context to be preserved |
| Optimized for CRUD operations on a small number of rows (KBs to MBs) | Optimized for retrieving and aggregating results spanning many, if not all, rows (GBs to PBs) |
| Works well with row stores | Works well with column stores |
| Data consumers are OLTP (online transaction processing systems) and software engineers | Data consumers are analysts, data scientists, software engineers, commercial, and business people |
| Protects transactional integrity (ACID guarantee) | Is the “simple and fast” source of actionable information |

Another advantage of the dimensional model is its extensibility. By design, **dimensions are modular**, and can and should be reused. This allows building a data warehouse in an agile, incremental fashion. As multiple business processes share dimensions, this enables overall conformity. Also, having one version of the truth, i.e., coherent dimensions, allow business users to analyze data better. Dimensional model integrates various business processes in an application-agnostic manner.

At Doximity, we use dimensional modeling extensively for our analysis and reporting. We have daily ETL (extract, transform, load) jobs running via Airflow that load production data from MySQL into fact and dimension tables in Snowflake. Further, we have hourly and daily tasks that utilize this modeled data and create interactive multi-level dashboards using Looker.

As an example, one such set of dashboards are built on the dimensional model of user engagement. A simplified version of the star schema is reproduced below. The fact table enables numerous aggregations on user engagement, by user attributes, medical specialties, Doximity products (newsfeed, digest, jobs, etc), platform (iOS, Android, web), and over multiple time frames. We also calculate advanced metrics such as daily active users, weekly active users, monthly active users, and quarterly active users using this data.



Notice how in this dimensional design, specialties and users are two **unlinked** tables that can be independently joined to user\_engagement through corresponding foreign keys in the fact table. In a normalized, transactional design, specialties would be **permanently linked** to users through a foreign key from the specialties table. Then, the user\_engagement table would link to the users table. Hence in a transactional design, any analysis consisting of only specialties based engagement will have to have an intermediate join with users. In a dimensional design, however, all contextual entities required for aggregation and analysis are directly linked to the fact table as independent dimensions. This creates some duplication (such as the field specialty code in users) but allows for both simplified design and performant joins, which are key benefits of dimensional design philosophy. Hence, in this case, specialties is an independent dimension that can be used to aggregate and query the engagement data without joining through users.

A diverse audience uses our dashboards. We have software and data engineers tracking daily metrics, commercial and business teams pursuing client interaction and interest, and product managers, senior managers, CTO, CPO, CEO interested in overall user engagement and product growth, and adoption. Dimensional data modeling allows us to cater to all these needs effectively.

This article was inspired by a talk I watched at Rubyconf, 2019 titled: [The Fewer the Concepts, the Better the Code](https://rubyconf.org/program#session-894)

How readable is the code we write? How well is someone else able to read it? Who is our audience, both now and in the future? It could be ourselves two months from now or perhaps a fellow colleague.

Martin Fowler once said:

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." [(link)](https://en.wikiquote.org/wiki/Martin_Fowler)

More often than not we think the humans Fowler refers to are skilled programmers who work in the same language I do. But what if I am the only engineer who knows Golang well enough to maintain that one legacy application written in Go? How do I write Golang in a way my fellow engineers can easily understand?

This question was the topic of David Copeland's 2019 RubyConf talk "The Fewer the Concepts, the Better the Code."

Take these two chunks of code as an example:

#### CODE SAMPLE 1

class Order

def total

items.map do |item|

item.price \* item.quantity

end.reduce(&:+)

end

end

#### CODE SAMPLE 2

class Order

def total

result = 0

for item in items do

result = result + (item.price \* item.quantity)

end

return result

end

end

Both snippets accomplish the same result, but are written differently. Which one is better? The answer is it depends.

To some, the second code sample may seem uglier than the first and would probably get bad marks from linters like Rubocop. However, for a programmer coming from another language the second snippet will be easier to understand for two reasons:

1. **CODE SAMPLE 1** requires knowledge of class, def, .map, ||, \*, .reduce(), &:+, as well as Ruby's implicit return. Whereas CODE SAMPLE 2 requires knowledge of fewer concepts: class, def, =, for, +, \*, return.
2. **CODE SAMPLE 2** not only has fewer concepts, but the concepts used are almost identical to those used in many other programming languages.

**The goal Copeland urges us to strive for is not to write ugly code that is cumbersome and simplistic. Rather, to remember the audience we’re writing code for might not be programmers exactly like us. We must consider the impact our code will have for programmers who are less experienced or come from an entirely different programming language.**

Here at Doximity we primarily develop in Ruby on Rails. We use the syntactic sugar that Ruby and Rails have built-in, and we love it. We would proudly write CODE SAMPLE 1 and commit it to our codebase because we know our fellow engineers are Ruby on Rails experts. If a fellow engineer is a beginner, we’ll take the time to teach them the concepts we’re using in our code instead of simplifying it. There is no harm in this approach since the other programmers maintaining this code, in terms of Ruby on Rails expertise, are like me.

However, if we were maintaining a legacy application written in Java, we would likely create change sets that are more similar in style to CODE SAMPLE 2, with fewer concepts that are simpler to understand for someone with a Ruby background.

I too share Copeland's thoughts here. We must consider the potential diversity of the audience reading our code and employ the appropriate mix of elegance and simplicity. The result will be code that is more easily maintained and tailored to the changing needs of our company.

**Working Remotely at Doximity**

Although headquartered in San Francisco, Doximity has a thriving remote work culture. While the office is the historical hub of activity, a large portion of our hiring over the last several months has been remote workers. Remote employees might take a back seat to on-site workers in a lot of offices, but, at Doximity, remote workers are first-class citizens.

#### Meetings and Communication

We strive to stay as lean and agile as possible and to create a culture that isn’t overly focused on meetings. That doesn’t mean we’re opposed to them — in fact it’s quite the opposite. Person-to-person communication is important and fosters a sense of connection and team building. To put that into a real-world perspective, you’re not going to have meetings all day, and the ones you do have are quick, focused, and important. As a company, we’ve invested in the infrastructure to make these more seamless and plan to keep that up as we grow.

A little bit about company structure: there are several different main teams, each working on a different product. For example, on the DocNews team, we begin the day with a standup meeting. Each participant is expected to talk about what was done the day before, what they'll be working on that day, and any blockers they may have encountered. It’s a good way to stay in contact with the team and, in addition to the intrinsic value of the standup, makes remote employees feel connected. Not every team does stand-ups but the infrastructure and options are there.

“Estimation meetings” are used to gauge how many points to assign to different stories. We talk about the feature and all give a thumbs up when we’re ready to estimate. Estimating points is done in a rock-paper-scissors-esque manner where we count to three and then put up fingers corresponding to the number of points we think a story should take. If we have different opinions on point assignments, we take time justify our position.

We recently started using a web app called [Stickies](https://stickies.io/) for our [retrospectives](https://engineering.doximity.com/articles/the-real-value-of-retrospectives). This lets remotes participate just like they were in the room. We also use live video chat during the meeting and collaborate using the app and Slack. [Read more about retrospectives](https://engineering.doximity.com/articles/the-real-value-of-retrospectives) at Doximity to get more insight into how they work for us.

Pair programming is also something we remoters do occasionally. This is generally up to the people who want to pair; we don’t force mandatory pairing time. Schedules permitting, coworkers are almost always up for pairing. We’ve mostly used Skype for this, but we also occasionally use tmux and a shared connection.

#### Hardware and Software

We use the typical tools to manage communication across our team:

##### [Slack](http://slack.com/)

We use Slack for pretty much everything. In-office text communication is done with Slack, which puts remote workers on the same playing field as everyone else. All of our deploys are done via in-house software which is controlled via bot. This lets any engineer have visibility into what's happening at all times. Most teams have their own channels (or several) where pertinent data is accessible, and the same applies to shared roles such as QA.

##### Everything Else

We make use of [Google Hangouts](http://google.com/hangouts), [Skype](http://skype.com/) and [appear](https://appear.in/) for video chats and screen sharing. Now that Slack is testing out video and audio chats, we may start using that more actively. Skype is slowly being phased out but we do have employees who prefer it.

Occasionally, we’ll use a remote tmux session for pairing. When that isn’t feasible, we use [Screenhero](https://screenhero.com/). Our [timezone](http://timezone.io/) account gives us a nice view of current times for remoters.

##### Hardware

Some of our conference rooms have recently been outfitted with [Chromeboxes](https://www.google.com/work/chrome/devices/for-meetings/), which have exceptional video and audio quality. These help tremendously and have the least amount of headaches when setting up.

#### In Person Visits

Even though working remote has tons of benefits, so does face-to-face interaction. Remote workers can visit the office every couple of months between our off-sites. We have two corporate lofts very close to the office where employees can stay.

New employees come to the office for their first week. This time functions as an introduction to the office, team, products, and the process. Developers are expected to make a commit and production deploy on their first day. There is also a series of presentations that teach about the company, culture, and introduce new hires to their coworkers.

Working in the office also means you can hang out at the J-Bar, an office staple. As the name suggests, there is alcohol available should you wish to partake, as well as a gorgeous view of San Francisco. It’s also where we do our lunch and learns, [meet-ups](http://www.meetup.com/Doximitys-Tech-Night/), and larger meetings. There is a camera and mic setup where remote participants can see and hear what’s going on as well as ask questions and interact with the presenter.

We also have off-sites every quarter. The off-site locations are different each quarter so we get to experience new places. With a diverse and remote team, off-sites are a chance to get everyone together in person. This helps to maintain the culture and provide an opportunity for everyone to realign and prioritize goals. We also share what we’ve learned and worked on the previous quarter and squash outstanding bugs in our bug bash.

A full development retrospective is done at each off-site. We try and concentrate on larger issues affecting all teams. I’m not trying to imply it’s all smooth sailing — it took a while to have a consensus on whether 80 or 120 characters is the proper line length for a Ruby file. (It’s 80, in case you were wondering.)

Automating workflows is key in setting your team up for success. We believe in building tools to help others focus on their craft rather than the nuances of process. These are the tools we rely on to improve our engineering workflow at Doximity:

## Setup

Jumping into new applications can be daunting. Learning the code and its business rules isn't enough. There are many dependencies to install and configure. Early on, we relied on README files to setup development environments. This worked well, but it was a manual process and consumed too much time. We are a Rails shop. Therefore, we rely on [bin/setup](https://robots.thoughtbot.com/bin-setup) to configure our environments.

The above three commands are all one should need to bootstrap our applications. Our bin/setup includes installing [brew](http://brew.sh/) and other system libraries. Here is a trimmed example of a few items from our bin/setup:

Once a team member is set up with an application, the next concern is merging commits into master. In our setup, they can rely on alerts to stay informed of errors. We encourage every new team member to deploy their code to production on day one. It’s vital they feel comfortable around our safeguards. Let's delve into some of them.

### Continous Integration

Whether we are running rSpec or minitest, we ensure every commit has an automated test suite ran against it. On completion, it's reported to the GitHub Pull Request and the respective Slack channels. Setting up Continuous Integration should be trivial with all the tools on the market. [CircleCI](https://circleci.com/) is our choice except for our mobile builds, for which we use [Jenkins](https://jenkins-ci.org/).

Pro Tip: [parallel\_tests](https://github.com/grosser/parallel_tests) can speed up builds while getting the most out of the VMs to save money.

### Instant Gratification

Having good test coverage is important, but if it takes hours to finish the build, it slows down your team. A fast build is a requirement. In our early days, our largest application’s test suite took 4 hours to run. Over the years, we’ve brought it down to 20 minutes by having a dedicated TestOps team who improved our test infrastructure. Smaller applications don't consume as much time, finishing in under 3 minutes. Consider how much time your team waits for a build to finish. That’s time users are deprived from the next new feature or a critical bug fix.

### Code Metrics

[CodeClimate](https://codeclimate.com/) can be an invaluable tool. It keeps your code lean and secure. Ensure your repository is hooked up to notify on Pull Requests so your team knows of issues before merging into master. [Brakeman](https://github.com/presidentbeef/brakeman) is also a good resource for analyzing security vulnerabilities.

Most aren't brave enough to deploy to production when their pull request looks like above.

## Delivery

Delivering your branch to production should be trivial -- but we all know it's not always that simple. Even with green automated tests, there is still a chance your branch can cause havoc in production. Perhaps it's the missing index which goes unnoticed in development. In the context of millions of rows in production, the impact can be drastic. Here is how we combat this.

### Staging and Production

Staging environments are common for testing before production deployment. Parity between these environments is critical. Ideally these environments replicate production as closely as possible from the NGINX configuration to production data stores. We use the following tools to build hundreds of environments in various clusters.

#### Packer

We ensure there are base images created via [Packer](https://www.packer.io/) templates with Ruby installations, speeding up the time it takes to bootstrap new environments.

#### Terraform

The biggest wrench in our toolbox is [Terraform](https://terraform.io/). We configure a lot with Terraform. Environment creation, IAM policies, S3 buckets, Route 53 records, Consul key/value stores, and many others. The following is a simplified version of our Terraform configuration with documentation:

Adding new environment involves the simple configuration shown below and running

#### Chef and Capistrano

Once Terraform finishes, [Chef](https://www.chef.io/chef) takes over provisioning everything that’s required by the Rails application. At that point, [Capistrano](http://capistranorb.com/) handles all our deployments.

### Deployments

Ensuring your team (whether it’s engineers or not) can deploy via ChatOps can be a huge time saver. Besides, it allows us to queue up dozens of deployments.

At Doximity, we rely on a combination of Slack, [Hubot](https://hubot.github.com/), and [Heaven](https://github.com/atmos/heaven) to handle deployments. For better visibility, we've customized Heaven with a user interface. It reports status checks on the individual environments along with deployment history.

Pro Tip: Lock deployments to production at Beer O’Clock.

#### Migrations

Migrating tables are common and they should be trivial to perform via ChatOps. With the combination of Capistrano, Hubot, and Heaven, this should be doable by everyone. Migrations have dangers such as locked tables and missing columns. This results in either downtime or errors for users. Relying on [Large Hadron Migrator](https://github.com/soundcloud/lhm), we perform live migrations without downtime for our users.

## Monitoring

Everyone on your team should be able to react to performance problems and address them with ease. After all, It’s Not Done Until It’s Fast. We use a combination of NewRelic for application reporting and [Sensu](https://sensuapp.org/) for server monitoring. [VictorOps](https://victorops.com/) alerts us when we aren’t by our computers. Otherwise, notifications are delivered to the Slack channel.

**Analyzing Complex Networks with R**

Imagine you stumble into a medical conference and decide you want to socialize with physicians. Your process may be to approach a random physician, speak to them for a bit, and then have them randomly select one of their colleagues whom you can speak to next. If you continued like this forever, how much time would you spend talking to each person? You might spend the most time talking with the person with the most connections, but depending on how the data is structured, that might not be the case. Let's take a look!

**Step 1:** Get a hold of the data you want to analyze and format it as an edge list in a csv. An [edge list](http://igraph.org/r/doc/graph_from_edgelist.html) has two columns which both represent nodes in the graph. For example, a row like 103, 105 might represent that physician 103 is connected to physician 105.

**Step 2:** [*Install R*](https://www.r-project.org/), as well as the *[igraph package](http://igraph.org/r/)*.

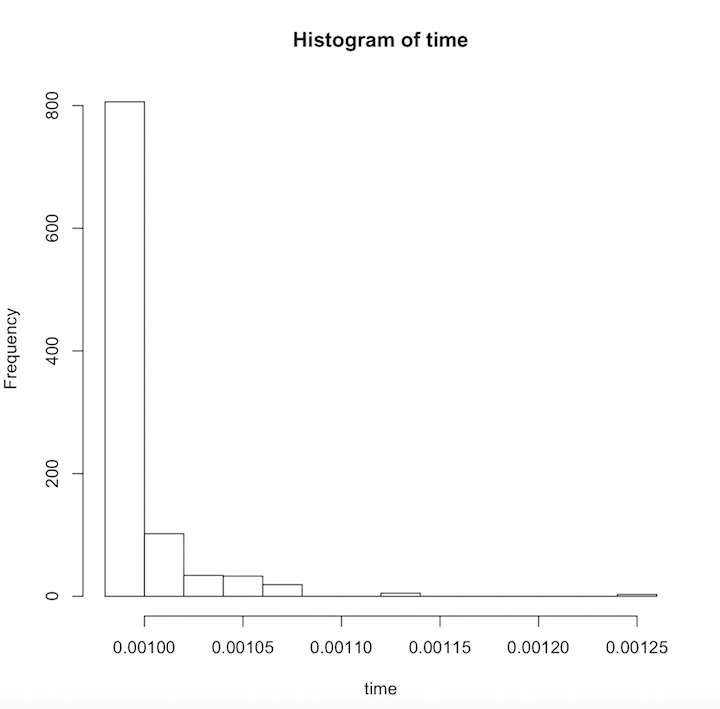
**Step 3:** Build a graph with the igraph package from your csv edge list. Running the function below prompts the user to select a csv file from the system and builds a graph.

**Step 4:** Manipulate the graph using some of the techniques from [*Google's page rank paper*](http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf) and build a [*transition matrix*](https://en.wikipedia.org/wiki/Stochastic_matrix).

Before moving to step 5, let's take a quick look at graph theory. An adjacency matrix is a representation of a graph where a spot in the matrix (for example, [103,105]) is 1 if physician 103 is connected to 105, and 0 if they aren't connected. To get the probability of randomly transitioning from any given node to any of its connections, you divide 1 by the number of connections for the physician. If physician 103 has 5 connections, then there would be a 1/5 or .20 chance of physician 103 directing you to 105. A matrix of all these probabilities is called a transition matrix.

But what happens if physician 103 is connected to 104 but 104 is connected to nobody? Or what if 104 is only connected to 105 and 105 is only connected to 104? These two examples, also known as leafs and periodic subgraphs, can distort the results of a random walk because the walker could never get to the rest of the graph. When Google was working on its page rank algorithm they came up with some tricks to ensure that the graph they analyzed would not have either of these properties. First, if a node has no outbound connections, then connect it to everything in the graph at equal probability. Second, the walker is given a 20% chance of jumping to anyone randomly at any time. The code to build such a transition matrix given a graph object is below:

**Step 5:** Compute the random walk probabilities for each physician. Now that we have a transition matrix to work with, we can use linear algebra to answer our original question. How much time will you spend talking to each person? It just so happens that when you're dealing with a real square matrix with positive entities, the eigenvector corresponding to its largest eigenvalue will give us exactly that information. Using the functions above the code to compute the dominant eigenvector looks like this:

Below is a graph of a small subset of Doximity physicians where the X axis shows the proportion of time spent talking to a physician and the Y axis shows how many physicians fell into that amount of time. The results follow a logarithmic trend where you would randomly chat with most physicians for about the same amount of time. A few physicians however stand out as people you would spend significantly more time chatting with.  


And just like that, you're able to quantify exactly how much time you would spend with each physician. This new piece of information is interesting on its own, but it can also be the start of many more fun data science exercises with R!