Crangraph - Final Report

James Lamb, Surya Nimmagadda, Jason Xie

Introduction

The R language is widely used among data scientists for developing statistical software and data analysis. The primary reason R has become so popular is the vast array of packages available at the cran repositories. In the last few years, the number of packages has grown exponentially. The packages continuously evolve with newer revisions coming out frequently. The constant evolution in terms of the number of packages and their versions poses an interesting problem of being able to maintain the dependency between the packages, which we would like to address with Crangraph.

In this project, we are using publicly-available metadata on R packages to build a dependency graph for the R language. The project uses the principles learnt in W205 for data storage and retrieval. Here we model new package releases as a stream and use infrastructure designed for streaming data storage (Kafka) and analysis (Storm). The data for this project are freely available from GitHub and CRAN and follow a known structure that should be manageable to parse. We created a front-end which shows the current state of the dependency graph in an interactive D3 graphic, allows users to view the evolution of the graph over time, and makes the dependency dataset available for easy access for other researchers to use.

Crangraph would be valuable to production data science shops who want to better understand the risks associated with building tools using certain R packages. For example, in many cases there are multiple R packages which can be used to accomplish a given task, and in the absence of other strong discriminating signals a team may want to choose a package nearer to the edge of the graph (i.e. less exposed to changes elsewhere in the package ecosystem and therefore less likely to contribute to software erosion).

Data Sources

All R packages approved for distribution via the Comprehensive R Archive Network (CRAN) must provide package metadata, such as version numbers, contributor names, and dependency lists, in a structured way . These metadata are bundled in with the package code files and distributed via CRAN. However, we can access these metadata without downloading and unpacking package tarballs. A read-only mirror of CRAN's 10,000 + packages is available on GitHub .These package-specific repos include one commit per new version.

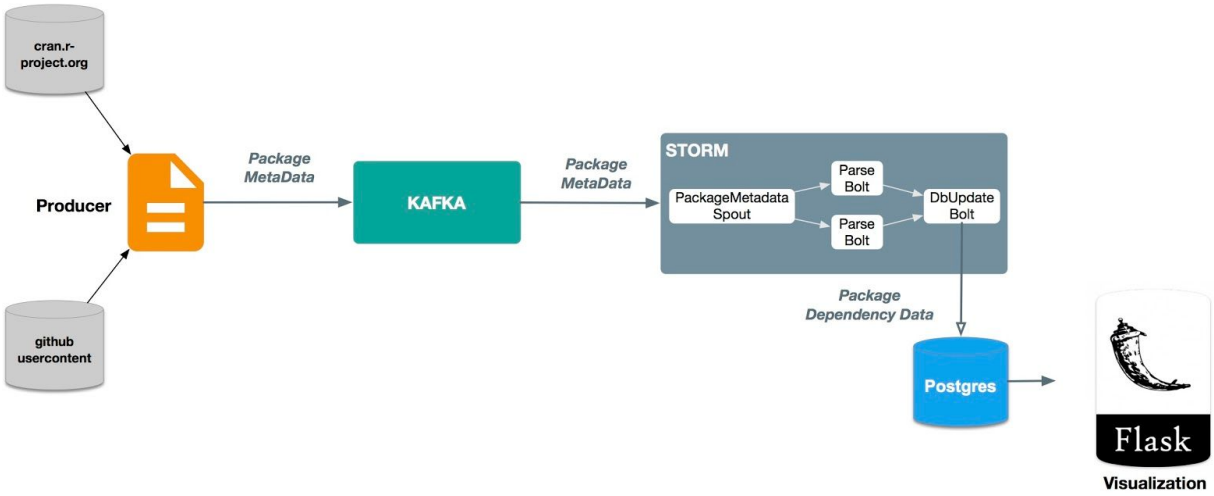
In addition, individual source code files can be scraped directly from GitHub with curl requests (without needing to unzip package tarballs). The metadata of interest for our project can be found in the package DESCRIPTION files, which follow a structure similar (but not identical) to YAML.

See, for example, https://raw.githubusercontent.com/cran/elastic/master/DESCRIPTION.

In this project, we model updates to packages (reflected in updates to these files) as a stream.

Architecture

The basic architecture and technology stack (described in more detail later in this report) is shown below.



Producer

To get package metadata into our application, we use a Kafka producer written in Python using the python-kafka library. This producer runs the following simple algorithm:

1. Pull the source of the CRAN package index
2. Scrape that page for a list of package names
3. Iterate over the list:
    a. For each package, pull DESCRIPTION file and put it directly onto a Kafka topic
4. Once all packages have been pulled and put on the topic, re-pull the package index and repeat

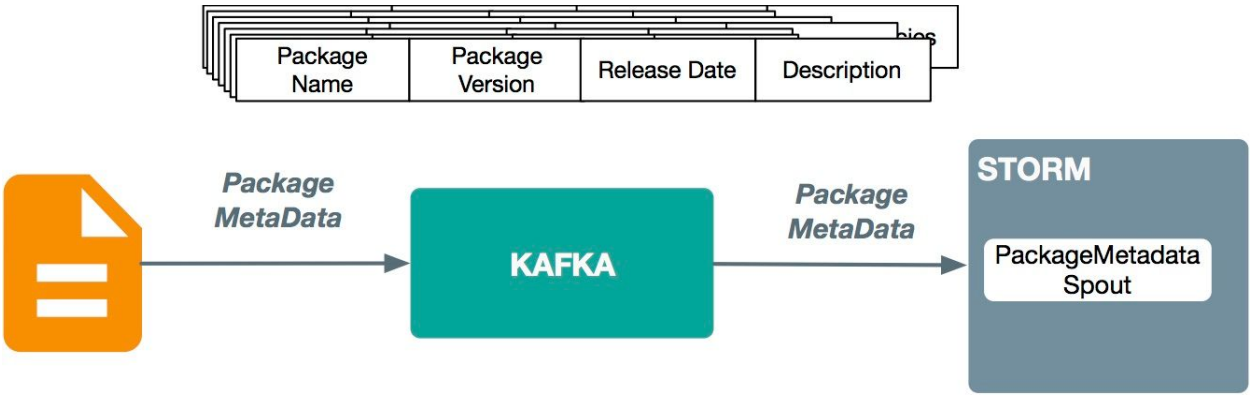
Through our own observation and experimentation, it takes about 1 hour to cycle through the entire package list, pull DESCRIPTION files, and get them on the topic. This implies that each

package could be checked for updates 24 times per day. This is more than sufficient coverage for users, and could even be slowed if the application became overworked.

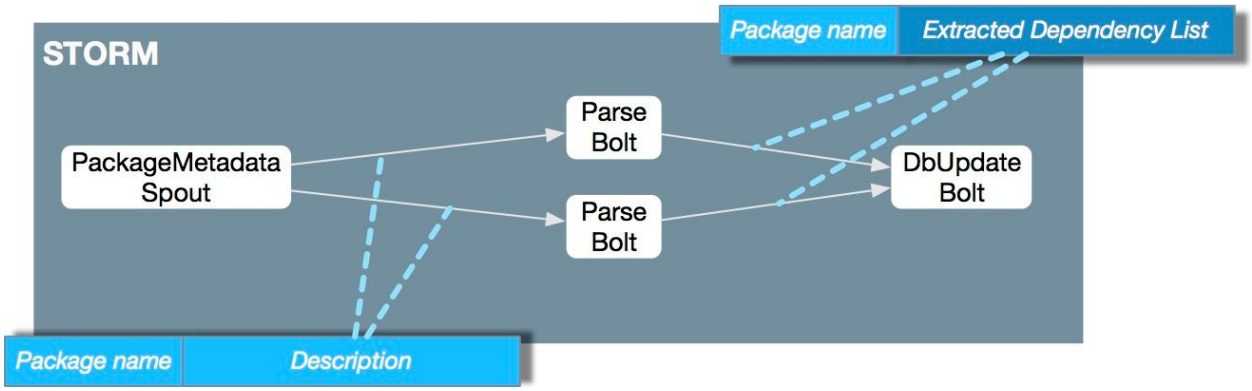
Data Storage, Retrieval and Processing

Crangraph is both compute and storage intensive. The amount of incoming data can be bursty in nature so it needs a scalable solution. To address this we have used a combination of Kafka, Apache Storm and Postgres.

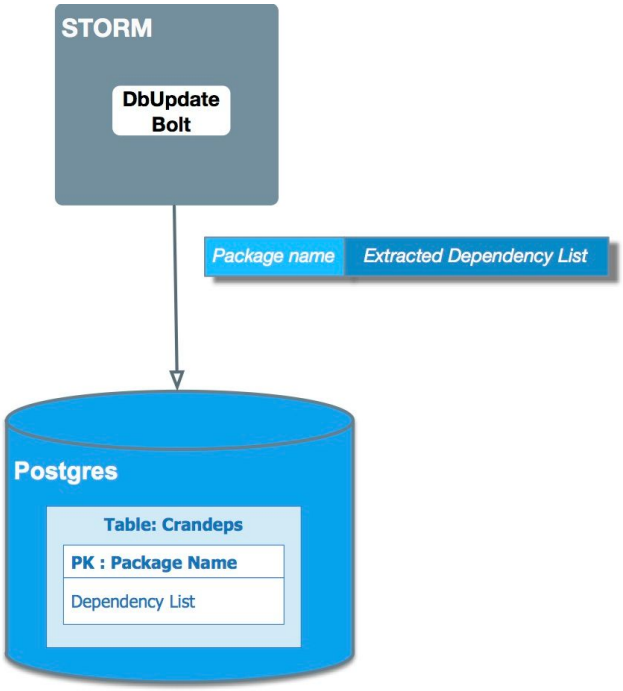
The producer sends the R package metadata as logs to Kafka server. At the other end, we connected a PackageMetadata spout that is configured as a Kafka consumer and implemented to fetch the package metadata from the Kafka server.



The PackageMetadata spout then emits the package name and raw DESCRIPTION file to the parse bolt. The parse bolt scrapes DESCRIPTION and extracts dependency list that is then given to the database update bolt.



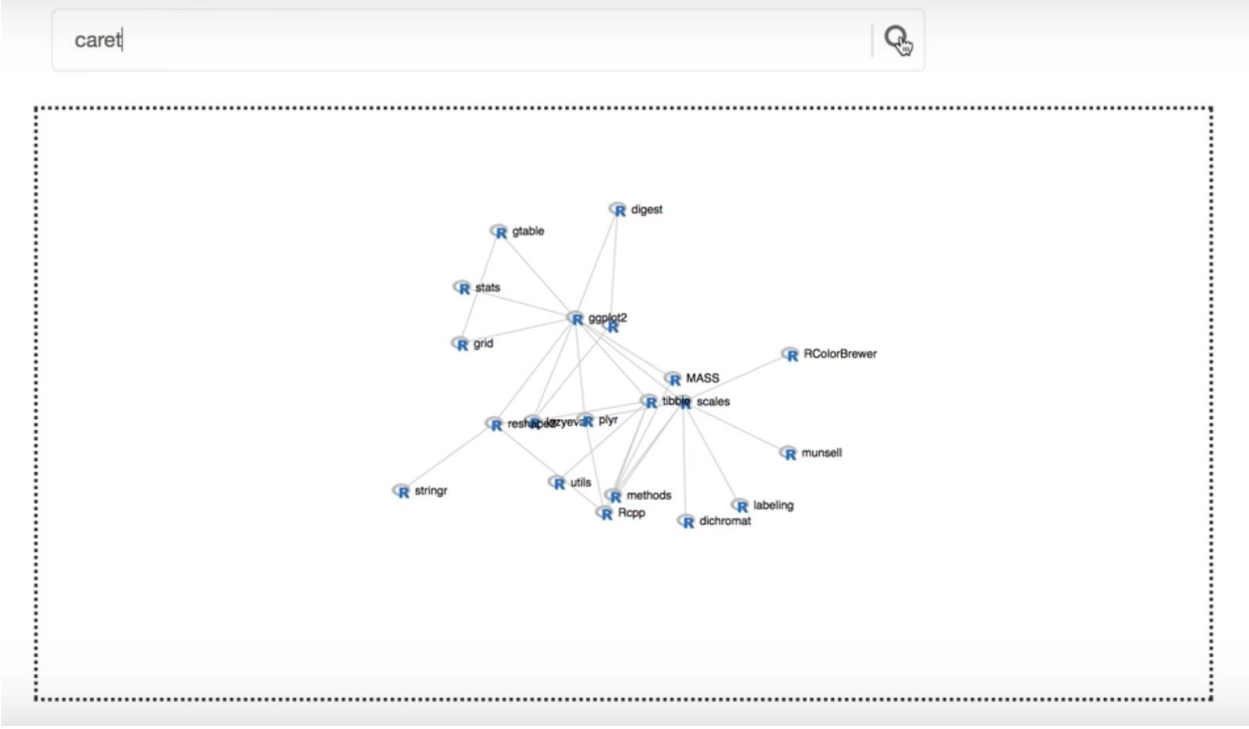
The database update bolt stores the parsed information in a postgres database named crangraph in a table named crandeps. The table stores the package name as primary key and the dependency list in JSON. To interface with postgres, psycopg2 is used within the database update bolt.

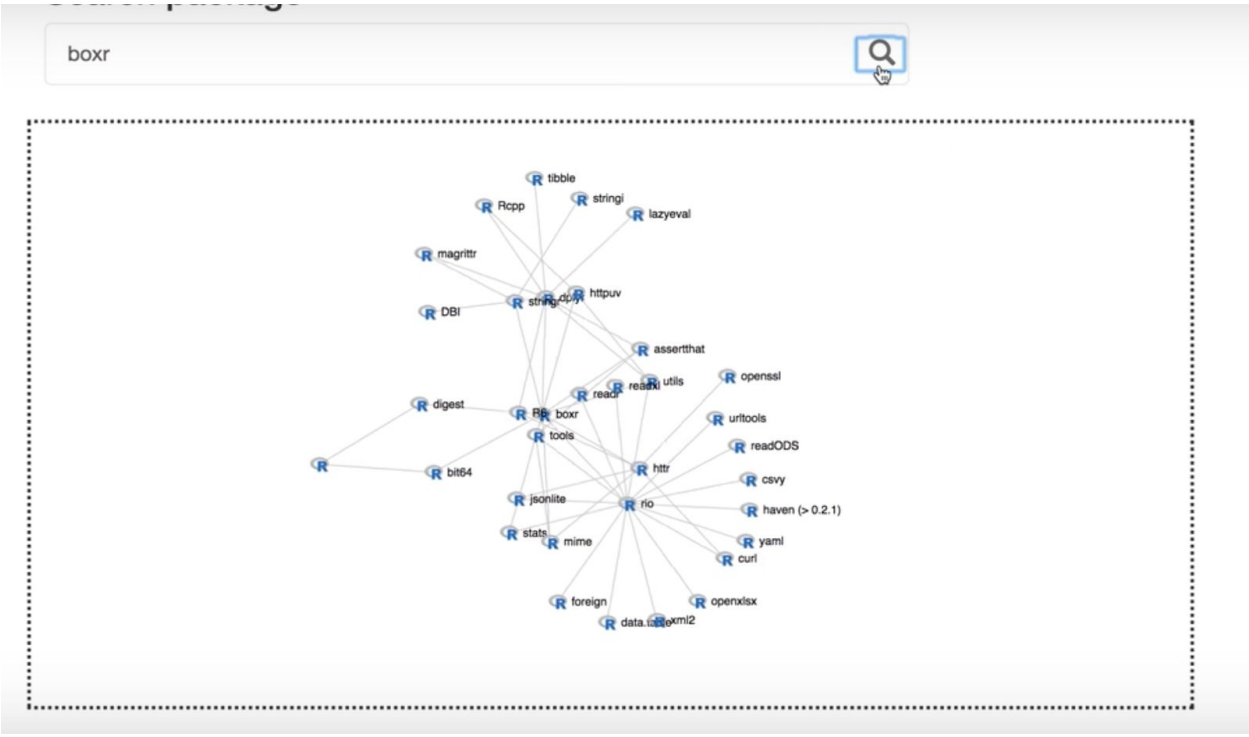


Most of our infrastructure (Kafka producer, Storm topology, Flask server) is orchestrated via Python programs. Most of the Python code in the application is package-ized, run in a dedicated conda environment, and documented using Sphinx. You can see the package documentation here.

Visualization

Crangraph's user interface is a simple web app which shows a force-directed graph of a package's 1st-degree and 2nd-degree dependencies.





For instance, say that we are interested in the package boxr. boxr directly requires rio, so rio is considered a first-degree dependency. As you can see in the second image above, this first-degree dependency exposes boxr to many second-degree dependencies, including openxlsx, xml2, yaml, and foreign.

How was it built?

The frontend of the dashboard is a D3 force-directed graph, meaning that we can drag and drop the graph.

The backend of the dashboard is built upon a python Flask server. The server serves 1) the D3 dashboard UI and 2) the neighbor graph data for a package (e.g. /data/car or /data/lme4). The whole Crangraph data is first loaded into a NetworkX graph object. Then we create a subgraph of a package, its neighbors and its neighbors' neighbors in the flask server.

Tools and Third-party Libraries

In this section, we describe the technology stack which forms the crangraph application.

Amazon AWS

To control the hardware and operating system available to the application, we use the following infrastructure provided by Amazon Web Services (AWS):

- Amazon Linux AMI → bare bones Linux operating system which we augment with setup scripts
- Amazon EC2 → compute service from Amazon which runs on the AMI
- Amazon EBS → Elastic Block Storage is used as a persistent storage layer mounted onto the application EC2. It's main purpose in this application is to serve as the physical storage for PostgreSQL

For more see: Amazon Linux AMI | Amazon EC2 | Amazon EBS

Apache Kafka

As mentioned above, we model R package updates as a stream. To make this stream available to downstream consumers in the application and to leave it flexible for other consumers to use in the future, crangraph places raw package metadata onto a Kafka topic and uses a Kafka consumer as the spout for its main Storm topology. We chose Kafka for this application because it is built to handle bursty streams and multiple data consumers.

We use Kafka with the out-of-the box defaults and create producers and consumers via a Python interface available in the kafka-python package.

For more see: Apache Kafka | kafka-python

Apache Storm

Once package metadata have been written to Kafka, they must be parsed into the format required by the UI and stored in our PostgreSQL serving database. Apache Storm is used to organize this processing and to distribute multiple tasks in the processing DAG across physical resources available to the application's EC2 instance.

crangraph's Storm topology is orchestrated almost entirely in Python using the streamparse library, but a small amount of config information is written in Clojure.

For more see: Apache Storm | streamparse

Conda

The AMI used in this project has a system distribution of python, but this is missing many of the components needed to run crangraph. To maintain tight control over the execution environment for the project's Python code, we use the dependency packaging tools provided in Continuum Analytics' Anaconda distribution. Namely, we use the Anaconda distribution of Python 2.7 and run all Python code in a custom conda environment created specifically for this project.

For more see: Anaconda | creating conda environments

Git / GitHub

Git is used to version control source code. GitHub is used as a remote repository for the application source code. This repository is cloned into the EC2 during application setup.

GitHub pages are used as a lightweight, free host for the project README and for the crangraph Python package documentation.

For more see: Git | GitHub pages | crangraph README | crangraph docs

JavaScript D3

The UI for this project is a simple webapp that allows on-demand rendering of subgraphs of the R dependency graph. That visualization is creating using the popular JavaScript D3 library.

For more see: D3 gallery | bl.ocks gallery

nginx

nginx is an open-source web server. In this application, it is responsible for handling communication between external clients (requests to the application) and the internal server running the crangraph Flask app.

For more see: About nginx

PostgreSQL

PostgreSQL is a popular open-source relational database. It is used as a serving DB for crangraph. In other words, application from the web front-end are translated into SQL queries against this database for dependency information on particular R packages. PostgreSQL allows

the storage of arbitrary JSON (dependency lists, in this case) which can be retrieved quickly from a table indexed by package name. PostgreSQL also has excellent support in Python, the main programming language used throughout this application

For more see: PostgreSQL | PostgreSQL on Linux | Psycopg2: Python + PostgreSQL

Python

Python is a popular general-purpose programming language. Because libraries exist for interacting with many technologies (including all of those mentioned in this report), we chose Python as the main library for this project. The Kafka producer / consumer, Storm Topology, serving web app, and other miscellaneous tools in crangraph were all written in Python.

This project uses Python 2.7 and relies on the use of conda environments for the maintenance and organization of dependencies.

For more see: Codecademy Python |

Sphinx

Much of the Python code supporting this project is wrapped in a Python package with several sub-packages. For example, all Bolts and Spouts in the crangraph Storm topology are stored and documented in a sub-package called crangraph.storm.

Sphinx is used to document the code in this package and to provide a front-end view that can be used to search the package. The package Sphinx docs are hosted on GitHub pages.

For more see: Sphinx | sphinx quickstart | crangraph Sphinx docs

Scaling Crangraph

1) Monitoring the resource usage and failure points
2) Scaling out our tech stack with microservices
    a) A large topology of Kafka and Storm nodes
    b) Distributed Graph database
    c) Graph-parallel visualizations (e.g. Spark GraphX)
3) Move to an image with more vCPUs then add more parsing bolts
4) Add better dedupe logic so we don't waste CPU cycles
5) Measure growth of logs and tune verbosity + persistence strategy
6) Give more resources (memory and compute) to webserver

Running the App + Source Code

All source code for crangraph is freely available from https://github.com/jameslamb/crangraph. To setup and run your own version of the application, follow the instructions provided in the project README.

It takes about 10 minutes to set up an EC2, mount an EBS volume, install all the necessary dependencies, and get the application started. It takes about an hour for the application to build the current state of the graph.

Future Extensions

1) Support arbitrary snapshots
    a) E.g. Crangraph at Feb 2014
2) Network metrics (link density, centrality, “influencer” packages)
3) Identify development communities
4) Detecting removals from CRAN
    a) We currently only allow additions of dependencies to the graph.
5) Animation of the evolution of Crangraph over time
6) Better app practices
    a) Security practices
    b) Error handling
    c) Strategy for deleting/archiving logs so they don't grow indefinitely
    d) Strategy for tuning how frequently we check for updates