How I parsed huge JSON files into an SQLite Database in under a second using the power of Crystal Language

Nathaniel Suchy June 1, 2019 Leave a comment

Recently I worked on a project with my friend **David Colombo**, he needed to take huge JSON files, map them onto an object, and then copy the data over from that object and insert it into a SQLite Database. This post describes the technical challenge of taking large amounts of JSON Data and inserting it into a SQLite Database without wait times.

Background

Previously, David had created a parser in **NodeJS** that reads through the **JSON** files containing structured data and inserts the data into **SQLite**. These JSON Files contained 1,000s of keys, and are 100s of megabytes each. Unfortunately due to limitations of the language and some anti-practices that JavaScript allows, the script took 14 hours to complete it's task. He attempted to write a parser in **Python**, except it was limited to 9999 keys and couldn't meet the project's expectations (in retrospect I'm sure there's a library or way to get around that limit, I'm not a Python Developer and cannot comment much on this potential limitation). He sent me a message asking if I was still working with **Ruby** and asked if it'd be any faster, I explained while I'm able to write a script in Ruby just fine, I had been experimenting with a language called **Crystal** (a language with Ruby-like syntax and C-like performance) and asked if he'd be open to trying it instead of Ruby. I also wasn't able to predict the performance of Ruby ahead of time and was not prepared to provide an answer on whether Ruby would be faster. Since I've been learning Crystal I decided to give the rewrite in it a shot.

A few requirements

We want the code to be long lived and require minimal changes (preferably no changes) to the project's dependencies, and we also wanted to avoid using third party libraries (since we risk having to replace them in the event the maintainers end the project) and only using Crystal's standard library was a goal. The standard library

does not support databases, so we decided to use the official shard for SQLite, it's maintained by the Crystal Core Team so it will probably be maintained well enough.

An object to represent the JSON Files

To maximize performance I decided to use a nested struct to contain the data using Crystal's `JSON.mapping()` (**https://crystal-lang.org/api/0.28.0/JSON.html#mapping**). The data wouldn't be changed, just copied into SQLite and stack memory is cheaper than heap memory so the drawbacks of structs were worth it in exchange for performance benefits.

```
module MyProgram
  extend self

  struct CVE_Data_Entity
    struct CVE_Items
      struct CVE
        struct DataMeta
          JSON.mapping(
            "id": {key: "ID", type: String, nilabl
            "assigner": {key: "ASSIGNER", type: St
          )
        end

        struct Affects
          struct Vendor
            struct VendorData
              struct Product
                struct Data
                  struct Version
                    struct Data
                      JSON.mapping(
                        "version_value": {type: St
                        "version_affected": {type:
                      )
                    end

                    JSON.mapping(
                      "version_data": {type: Array
                    )
                  end

                  JSON.mapping(
                    "product_name": {type: String,
                    "version": {type: Version, nil
                  )
                end

                JSON.mapping(
                  "product_data": {type: Array(Dat
                )
              end

              JSON.mapping(
                "vendor_name": {type: String, nila
                "product": {type: Product, nilable
              )
            end

            JSON.mapping(
              "vendor_data": {type: Array(VendorDa
            )
```

```
52              end
53
54              JSON.mapping(
55                "vendor": {type: Vendor, nilable: true
56              )
57            end
58
59            struct Problemtype
60              struct Data
61                struct Description
62                  JSON.mapping(
63                    "lang": {type: String, nilable: tr
64                    "value": {type: String, nilable: t
65                  )
66                end
67
68                JSON.mapping(
69                  "description": {type: Array(Descript
70                )
71              end
72
73              JSON.mapping(
74                "problemtype_data": {type: Array(Data)
75              )
76            end
77
78            struct References
79              struct Data
80                JSON.mapping(
81                  "url": {type: String, nilable: true}
82                  "name": {type: String, nilable: true
83                  "refsource": {type: String, nilable:
84                  "tags": {type: Array(String), nilabl
85                )
86              end
87
88              JSON.mapping(
89                "reference_data": {type: Array(Data),
90              )
91            end
92
93            struct Description
94              struct Data
95                JSON.mapping(
96                  "lang": {type: String, nilable: true
97                  "value": {type: String, nilable: tru
98                )
99              end
100
101             JSON.mapping(
102               "description_data": {type: Array(Data)
```

```
          )
        end

        JSON.mapping(
          "data_type": {type: String, nilable: tru
          "data_format": {type: String, nilable: t
          "data_version": {type: String, nilable:
          "cve_data_meta": {key: "CVE_data_meta",
          "affects": {type: Affects, nilable: true
          "problemtype": {type: Problemtype, nilab
          "references": {type: References, nilable
          "description": {type: Description, nilab
        )
      end

      struct Configurations
        struct Nodes
          struct CPE
            JSON.mapping(
              "vulnerable": {type: Bool, nilable:
              "cpe23Uri": {type: String, nilable:
            )
          end

          JSON.mapping(
            "operator": {type: String, nilable: tr
            "cpe_match": {type: Array(CPE), nilabl
          )
        end

        JSON.mapping(
          "cve_data_version": {key: "CVE_data_vers
          "nodes": {type: Array(Nodes), nilable: t
        )
      end

      struct Impact
        struct BaseMetricV3
          struct CvssV3
            JSON.mapping(
              "version": {type: String, nilable: t
              "vectorString": {type: String, nilab
              "attackVector": {type: String, nilab
              "attackComplexity": {type: String, n
              "privilegesRequired": {type: String,
              "userInteraction": {type: String, ni
              "scope": {type: String, nilable: tru
              "confidentialityImpact": {type: Stri
              "integrityImpact": {type: String, ni
              "availabilityImpact": {type: String,
              "baseScore": {type: Float64, nilable
```

```
154               "baseSeverity": {type: String, nilab
155             )
156           end
157
158           JSON.mapping(
159             "cvssV3": {type: CvssV3, nilable: true
160             "exploitabilityScore": {type: Float64,
161             "impactScore": {type: Float64, nilable
162           )
163         end
164
165         struct BaseMetricV2
166           struct CvssV2
167             JSON.mapping(
168               "version": {type: String, nilable: t
169               "vectorString": {type: String, nilab
170               "accessVector": {type: String, nilab
171               "accessComplexity": {type: String, n
172               "authentication": {type: String, nil
173               "confidentialityImpact": {type: Stri
174               "integrityImpact": {type: String, ni
175               "availabilityImpact": {type: String,
176               "baseScore": {type: Float64, nilable
177             )
178           end
179
180           JSON.mapping(
181             "cvssV2": {type: CvssV2, nilable: true
182             "severity": {type: String, nilable: tr
183             "exploitabilityScore": {type: Float64,
184             "impactScore": {type: Float64, nilable
185             "acInsufInfo": {type: Bool, nilable: t
186             "obtainAllPrivilege": {type: Bool, nil
187             "obtainUserPrivilege": {type: Bool, ni
188             "obtainOtherPrivilege": {type: Bool, n
189             "userInteractionRequired": {type: Bool
190           )
191         end
192
193         JSON.mapping(
194           "baseMetricV2": {type: BaseMetricV2, nil
195           "baseMetricV3": {type: BaseMetricV3, nil
196         )
197       end
198
199       JSON.mapping(
200         "cve": {type: CVE, nilable: true},
201         "configurations": {type: Configurations, n
202         "impact": {type: Impact, nilable: true},
203         "publishedDate": {type: String, nilable: t
204         "lastModifiedDate": {type: String, nilable
```

```
205         )
206       end
207
208       JSON.mapping(
209         "cve_data_type": {key: "CVE_data_type", type
210         "cve_data_format": {key: "CVE_data_format",
211         "cve_data_version": {key: "CVE_data_version"
212         "cve_data_numberofcves": {key: "CVE_data_num
213         "cve_data_timestamp": {key: "CVE_data_timest
214         "cve_items": {key: "CVE_Items", type: Array(
215       )
216     end
217   end
```

Improving the JSON parsing time

In Crystal's development mode parsing one of the JSON files (`2018.json` containing around 200MB of data) into the object took 30 seconds, in release mode it took 5 seconds. This performance was pretty good already but I'd like it to be faster as the datasets would get larger and larger over time. The first thing I tried was changing the `class` to a `struct` (which is being used now). That had minimal impact on performance. Next I changed from using the `File.open()` method to the `File.read()` method which improved the file read speed and brought the parse time down to under a second. From this we learned that it's much faster to open a file in read mode, than in read and write mode. When writing code we know to only ask for read permissions except when we also need to write to it. There are probably more file optimizations we could try, although that's a topic of it's own.

Inserting the data into SQLite

Gathering the data and attaching it into an object was only half the challenge, next I needed an efficient way to massively insert data into SQLite. At first I tried iterating over the various arrays and doing a lot of individual queries, on my Mac that still took around five minutes and much longer (never actually completed) on a Linux Laptop. I then learned I could **group these queries into one bulk transaction**. I came up with the following code that ran in under a second.

```
require "json"
require "sqlite3"
require "./cve_data_entity.cr"

module MyProgram
  VERSION = "0.1.0"

  filepath = "./src/example-json-files/example-full
  myobject = CVE_Data_Entity.from_json(File.read(fi

  DB.open "sqlite3://./src/example-json-files/dbnam
    db.transaction do |tx|
      tx.begin_transaction
      myobject.try(&.cve_items).try(&.each do |item
        # Insert General Information into the Datab
        cve_item_id = item.try(&.cve).try(&.cve_dat
        data_type = item.try(&.cve).try(&.data_type
        data_format = item.try(&.cve).try(&.data_fo
        data_version = item.try(&.cve).try(&.data_v
        published_date = item.try(&.publishedDate)
        last_modified_date = item.try(&.lastModifie
        tx.connection.exec("INSERT INTO GENERAL_INF

      end)
      tx.commit
    end
  end
end
```

Admittedly the `.try()` method calls can be a bit messy and we're looking into cleaner ways to write this type of code. One recommendation was to port the JSON `.dig()` method to my struct, in the future I might attempt that. If its difficult reading through the try logic right now, read about capturing blocks and procs in the **Crystal Documentation** first and it'll make more sense. This method is not ideal when working with larger amounts of code. Although other than the readability issues, there was not a huge performance impact.

In conclusion

By writing our parser and database insertion logic in Crystal we were forced to use **better coding practices**, we learned about **SQLite transactions**, and saved about 14 hours on our database's build time and brought it down to under a second. If you have a similar challenge in your organization, consider trying to solve it using **Crystal**.

Published by Nathaniel Suchy Software Engineer at Universal Layer LLC | 🏳️‍🌈 Non-binary Transgender Person (They/Them pronouns)
View more posts