Pages / ... / Analysis CArchive

Comparison of serialization formats

Created by 김동구 부장(실장) on May 03, 2022

- Pros & Cons per serialization format
- Speed and Size comparison
- JSON Performance
    - JsonCpp vs. RapidJSON
- Conclusion
- Reference sites

Pros & Cons per serialization format

name	Description	Pros	Cons	type

Apache Avro	- Row-oriented remote procedure call and data serialization framework developed within Apache's Hadoop project.
	- It uses JSON for defining data types and protocols, and serializes data in a compact binary format.
	- uses a schema to structure the data that is being encoded
	- two different types of schema language: Avro IDL(human editing), machine-readable based on JSON
	- be similar to Thrift and Protocol Buffers
		- a linguistic-neutral serialization of data
		- stores the schema in a file header, so the data is self-describing
		- Easy and fast data serialization and deserialization, which can provide very good ingestion performance
		- As with the Sequence files, the Avro files also contain synchronization markers to separate blocks. This makes it highly splittable
		- Files formatted in Avro are splittable and compressible and are therefore a good candidate for data storage in the Hadoop ecosystem
		- The schema used to read Avro files does not necessarily have to be the same as the one used to write the files.
			- Makes you think about the schema and data types
			- Its data is not human-readable
			- Not integrated into every programming language
				Schema-driven
				ex) cpx.json
				cpx is a C++ representation of the Avro schema.
				{
				"type": "record",
				"name": "cpx",
				"fields" : [
				{"name": "re", "type": "double"},
				{"name": "im", "type" : "double"}
				]
				}

Apache Parquet		- columnar format. Only the required columns will be retrieved/read, this reduces disk I/O. The concept is called projection pushdown
		- The scheme travels with the data, so the data is self-describing
		- Although it is designed for HDFS, data can be stored on other file systems such as GlusterFs or NFS
		- just files, which means it's easy to work, move, backup and replicate them
		- provides very good compression up to 75% when using even compression formats like snappy
		- As practice shows, this format is the fastest for read-heavy processes compared to other file formats
		- well suited for data storage solutions where aggregation on a particular column over a huge set of data is required
		- can be read and written using the Avro API and Avro Schema
		- provides predicate pushdown, thus reducing the further cost of transferring data from storage to the processing engine for filtering
			- The column-based design makes you think about the schema and data types
			- Parquet does not always have built-in support in tools other than Spark
			- not support data modification (Parquet files are immutable) and scheme evolution
				Schema-driven
				ex)
				enum PhoneType {
				HOME,
				WORK,
				MOBILE,
				OTHER
				}
				struct Phone {
				1: i32 id,
				2: string number,
				3: PhoneType type
				}
				service PhoneService {
				Phone findById(1: i32 id),
				list<Phone> findAll()
				}

FlatBuffers	- Free software library implementing a serialization format similar to Protocol Buffers, Thrift, Apache Avro, SBE, Cap'n Proto	- Supports "zero-copy" deserialization	- handling of FlatBuffers requires usually more code, and some operations are not possible (like some mutation operations).	Schema-driven
JSON	- open standard file format and data interchange format - language-independent data format	- supports hierarchical structures, simplifying the storage of related data in a single document and presenting complex relationships - Most languages provide simplified JSON serialization libraries or built-in support for JSON serialization/ deserialization - supports lists of objects, helping to avoid chaotic list conversion to a relational data model - widely used file format for NoSQL databases such as MongoDB, Couchbase and	- consumes more memory due to repeatable column names - Poor support for special characters - not very splittable - lacks indexing - less compact as compared to over binary formats	Schema-less

name	Description	Pros	Cons	type
		Azure Cosmos DB • Built-in support in most modern tools		
OpenDDL (OPen Data Description Language)	• general-purpose, human-readable, and strongly-typed data language for information exchange • can be used for a wide range of applications that include everything from simple configuration files to the exchange of complex information among many programs in an editable format. • It was originally developed as the basis for the Open Game Engine Exchange format.		• library license is GPL 3.0	
Protocol Buffers(protobuf)	• Free and open-source cross-platform data format used to serialize structured data	• Data is fully typed. • Protobuf supports schema evolution and batch/streaming processing. • Mainly used to serialize data, like AVRO. • Having a predefined and larger set of data types, messages serialized on Protobuf can be automatically validated by the code that is responsible to exchange them.	• not human-readable and human-editable • not good for the purposes of storing something e.g. a text document, or a database dump • Not splittable and not compressible. • No Map Reduce support. • Require a reference file with the schema. • Not designed to handle large messages. Since it doesn't support random access, you'll have to read the whole file, even if you only want to access a specific item.	Schema-driven
Apache Thrift	• Interface definition language and binary communication protocol used for defining and creating services for numerous programming languages	• Cross-language serialization with lower overhead than alternatives such as SOAP due to use of binary format. • No XML configuration files. • The language bindings feel natural. For example, Java uses ArrayList<String>. C++ uses std::vector<std::string>. • The application-level wire format and the serialization-level wire format are cleanly separated. They can be modified independently. • The predefined serialization styles include: binary, HTTP-friendly and compact binary. • Doubles as cross-language file serialization. • Soft versioning[clarify] of the protocol. Thrift does not require a centralized and explicit mechanism like major-version/minor-version. Loosely coupled teams can freely evolve RPC calls. • No build dependencies or non-standard software. No mix of incompatible software licenses.	• Long coding time • Large volume after coding • no documentation • not support server push	Schema-driven
Smile	• Computer data interchange format based on JSON • Format was specified in 2010 by Jackson JSON processor development team.	• More compact and more efficient to read and write		Schema-less
XML		• Supports batch/streaming processing. • Stores meta data along the data and supports the schema evolution. • Being a verbose format, It provides a good ratio of compression compare to JSON file or other text file format.	• Not splittable since XML has an opening tag at the beginning and a closing tag at the end. You cannot start processing at any point in the middle of those tags. • The redundant nature of the syntax causes higher storage and transportation cost when the volume of data is large. • XML namespaces are problematic to use. The support of namespace can be difficult to correctly implement in an XML parser. • Difficulties to parse requiring the selection and the usage of an appropriate DOM or SAX parser.	
MessagePack	• Computer data interchange format • binary form for representing simple	• more compact than JSON • allows binary data and non-UTF-8 encoded strings • any type can be a map key,	• imposes limitations on array and integer sizes a value of an Integer object is	Schema-less ex)

name	Description	Pros	Cons	type

	data structures like arras and associative arrays.	including types like maps and arrays, and, like YAML, numbers
		• more space-efficient than BSON
		• MessagePack is designed for efficient transmission over the wire
			limited from -(2^63) upto (2^64)-1
			maximum length of a Binary object is (2^32)-1
			maximum byte size of a String object is (2^32)-1
			String objects may contain invalid byte sequence and the behavior of a deserializer depends on the actual implementation when it received invalid byte sequence
			Deserializers should provide functionality to get the original byte array so that applications can decide how to handle the object
			maximum number of elements of an Array object is (2^32)-1
			maximum number of key-value associations of a Map object is (2^32)-1
			• MessagePack returns only a dynamically typed data structure and provides no automatic structure checks.
				#include <msgpack.hpp>
				#include <string>
				#include <iostream>
				#include <sstream>
				
				int main()
				{
				msgpack::type::tuple<int, bool, std::string> src(1, true, "example");
				
				// serialize the object into the buffer.
				// any classes that implements write(const char*,size_t) can be a buffer.
				std::stringstream buffer;
				msgpack::pack(buffer, src);
				
				// send the buffer ...
				buffer.seekg(0);
				
				// deserialize the buffer into msgpack::object instance.
				std::string str(buffer.str());
				
				msgpack::object_handle oh = msgpack::unpack(str.data(), str.size());
				
				// deserialized object is valid during the msgpack::object_handle instance is alive.
				msgpack::object deserialized = oh.get();
				
				// msgpack::object supports ostream.
				std::cout << deserialized << std::endl;
				
				// convert msgpack::object instance into the original type.
				// if the type is mismatched, it throws msgpack::type_error exception.
				msgpack::type::tuple<int, bool, std::string> dst;
				deserialized.convert(dst);
				
				// or create the new instance
				msgpack::type::tuple<int, bool, std::string> dst2 =
				deserialized.as<msgpack::type::tuple<int, bool, std::string> >();
				
				return 0;
				}

Cap'n Proto	• Data serialization format and Remote Procedure Call(RPC) framework for exchanging data between computer programs
	• The Cap'n Proto interface schema uses a C-like syntax and supports common primitives data types (booleans, integers, floats, etc.), compound types (structs, lists, enums), as well as generics and dynamic types.
	• supports Object Oriented features such as multiple inheritance, which has been criticized for its complexity.
		• Incremental reads: It is easy to start processing a Cap'n Proto message before you have received all of it since outer objects appear entirely before inner objects (as opposed to most encodings, where outer objects encompass inner objects).
		• Random access: You can read just one field of a message without parsing the whole thing.
		• mmap: Read a large Cap'n Proto file by memory-mapping it. The OS won't even read in the parts that you don't access.
		• Inter-language communication: Calling C++ code from, say, Java or Python tends to be painful or slow. With Cap'n Proto, the two languages can easily operate on the same in-memory data structure.
		• Inter-process communication: Multiple processes running on the same machine can share a Cap'n Proto message via shared memory. No need to pipe data through the kernel. Calling another process can be just as fast and easy as calling another thread.
		• Arena allocation: Manipulating Protobuf objects tends to be bogged down by memory allocation, unless you are very careful about object reuse. Cap'n Proto objects are always allocated in an "arena" or "region" style, which is faster and promotes cache locality.
		• Tiny generated code: Protobuf generates dedicated parsing and serialization code for every message type, and this code tends to be enormous. Cap'n Proto generated code is smaller by an order of magnitude or more. In fact, usually it's no more than some inline accessor methods!
		• Tiny runtime library: Due to the simplicity of the Cap'n Proto format, the runtime library can be much smaller.
		• Time-traveling RPC: Cap'n
				Schema-driven
				ex)
				@0xdbb9ad1f14bf0b36; # unique file ID, generated by `capnp id`
				struct Person {
				name @0 :Text;
				birthdate @3 :Date;
				email @1 :Text;
				phones @2 :List(PhoneNumber);
				
				struct PhoneNumber {
				number @0 :Text;
				type @1 :Type;
				enum Type {
				mobile @0;
				home @1;
				work @2;
				}
				}
				}
				
				struct Date {
				year @0 :Int16;
				month @1 :UInt8;
				day @2 :UInt8;
				}

name	Description	Pros	Cons	type
		Proto features an RPC system that implements time travel such that call results are returned to the client before the request even arrives at the server!		
CSV(Comma-Separated Values)	- With only commas separating values, CSV is very concise for a human-readable format.	- human-readable and easy to edit manually - provides a simple scheme - can be processed by almost all existing applications - easy to implement and parse - compact	- allows you to work with flat data. Complex data structures have to be processed separately from the format - No support for column types. No difference between text and numeric columns - no standard way to present binary data - Problems with CSV import (for example, no difference between NULL and quotes - Poor support for special characters - Lack of a universal standard	
YAS (Yet Another Serialization)	- is created as a replacement of boost.serialization - require C++ 11 support	- fast - header only library - YAS binary archives is endian independent.	- To do: protobuf/messagepack support limits objects versioning - weak documentation	ex) https://github.com/niXman/yas/blob/master/tests/base/include/serialize.hpp

cereal	- a header-only C++ 11 serialization library	- fast and compact
		- comes with full support for c++11
		- smart pointers (things like std::shared_ptr and std::unique_ptr) are supported.
		- unit tested
		- supports binary serialization, XML serialization, and JSON serialization
			- a consequence of this raw pointers and references are not supported.	Schema-less
				ex)
				#include <cereal/types/unordered_map.hpp>
				#include <cereal/types/memory.hpp>
				#include <cereal/archives/binary.hpp>
				#include <fstream>
				
				struct MyRecord
				{
				uint8_t x, y;
				float z;
				
				template <class Archive>
				void serialize(Archive & ar)
				{
				ar(x, y, z);
				}
				};
				
				struct SomeData
				{
				int32_t id;
				std::shared_ptr<std::unordered_map<uint32_t, MyRecord>> data;
				
				template <class Archive>
				void save(Archive & ar) const
				{
				ar(data);
				}
				
				template <class Archive>
				void load(Archive & ar)
				{
				static int32_t idGen = 0;
				id = idGen++;
				ar(data);
				}
				};
				
				int main()
				{
				std::ofstream os("out.cereal", std::ios::binary);
				cereal::BinaryOutputArchive archive(os);
				SomeData myData;
				archive(myData);
				return 0;
				}

name	Based on	Specification	Binary	Human-readable	Standard APIs	Supports zero-copy operations	license	Requirements
Apache Avro	N/A	Apache Avro Specification	Yes	Partial	C, C#, C++, Java, PHP, Python, Ruby	N/A	Apache License 2.0	- c++ compiler and runtime libraries - Boost library version 1.38 or later - CMake build tool version 2.6 or later - Python
Apache Parquet	N/A	Apache Parquet	Yes	No	Java, Python, C++	No	Apache License 2.0	
FlatBuffers	N/A	FlatBuffers GitHub	Yes	Apache Arrow	C++, Java, C#, Go, Python, Rust, JavaScript, PHP, C, Dart, Lua,	Yes	Apache License 2.0	

name	Based on	Specification	Binary	Human-readable	Standard APIs	Supports zero-copy operations	license	Requirements
					TypeScript			
JSON	JavaScript syntax	STD90 / RFC 8259(ancillary: RFC 6901, 6902), ECMA-404, ISO/IEC 21778:2017	No, but see BSON, Smile, UBJSON	Yes	Partial(Clarinet, JSONQuery/RQL, JSONPath), JSON-LD	No	JSON license	
OpenDDL	C, PHP	OpenDDL.org	No	Yes	OpenDDL library(implemented c++)	N/A	GPL 3.0	
Protocol Buffers(protobuf)	N/A	Developer Guide: Encoding	Yes	Partial	C++, Java, C#, Pytho, Go, Ruby, Objective-C, C, Dart, Perl, PHP, R, Rust, Scala, Swift, Julia, Erlang, D, Haskell, Action Script, Delphi, Elixir, Elm, Erlang, GopherJS, Haskell, Haxe, JavaScript, Kotlin, Lua, Matlab, Mercurt, OCaml, Prolog, Solidity, Typescript, Vala, Visual Basic	No	BSD	
Apache Thrift	N/A	Original whitepaper	Yes	Partial	C++. Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml, Delphi and other languages	N/A	Apache License 2.0	
Smile	JSON	Smile Format Specification	Yes	No	Partial(via JSON APIs implemented with Smile backend, on Jackson, Python) Java, C, Go, Javascript, Python, Rust	N/A	BSD 2-Clause	

XML	SGML	W3C Recommendations:
		1.0(Fifth Edition)
		1.1(Second Edition)
			Partial
			(Efficient XML Interchange, Binary XML, Fast Infoset, XSD base64 data)
				Yes	DOM, SAX, XQuery, XPath	N/A		

UBJSON	JSON, BSON	Ubjson.org	Yes	No	No	N/A	Apache 2.0	
MessagePack	JSON(loosely)	MessagePack format specification (github: https://github.com/msgpack/msgpack-c/tree/cpp_master)	Yes	No	No	Yes	Apache	• requires boost library.
Cap'n Proto		Capnproto.org					MIT	
CSV (Comma-serperated values)	N/A	RFC 4180 (among others)	No	Yes	No	No		
Cista++		https://github.com/felixguendling/cista					MIT	• C++17 compatible compiler • CMake
YAS (Yet Another Serialization)							Boost	C++ 11 support • Supported types of archives: binary text json(not fully comply) • Supported compilers: GCC 4.8.5 or higher(32/64 bit) MinGW: 4.8.5 or higher(32/64 bit) Clang: 3.5 or higher(32/64 bit) Intel: (untested) MSVC : 2017(in c++14 mode), or higher(32/64 bit) Emscripten: 1.38 (clang version 6.0.1)
cereal		https://uscilab.github.io/cereal/index.html	Yes	Yes (RapidJSON, RapidXML)	C++		BSD	• officially supports g++ 4.7.3, clang++ 3.3, and MSVC 2013 (or newer). • cmake_minimum_required(VERSION 3.6...3.15)

name	Based on	Specification	Binary	Human-readable	Standard APIs	Supports zero-copy operations	license	Requirements
								Narnia current use CMake(version 3.16.2)

Speed and Size comparison

(from https://github.com/thekvs/cpp-serializers)

Following results were obtained running 1000000 serialize-deserialize operations 50 times and then averaging results on a typical desktop computer with Intel Core i7 processor running Ubuntu 16.04.

For capnproto and flatbuffers since they already store data in a "serialized" form and serialization basically means getting pointer to the internal storage, we measure full build/serialize/deserialize cycle. In the case of other libraries we measure serialize/deserialize cycle of the already built data structure.

serializer name	version	object's size(bytes)	average total time(ms)	Rank of Object size per serializer	Rank of time per serializer	Human-readable
thrift-binary	0.12.0	17017	1190.22	6	5	No
thrift-compact	0.12.0	13378	3474.32	2	10	Partial
protobuf	3.7.0	16116	2312.78	4	8	Partial (If you use text_format, you can read.)
boost	1.69.0	17470	1195.04	9	6	No
MessagePack	3.1.1	13402	2560.6	3	9	No
cereal	1.2.2	17416	1052.46	7	4	Yes (If you use JSON Archive type, you can read)
avro	1.8.2	16384	4488.18	5	11	Partial
capnproto	0.7.0	17768	400.98	11	2	No
flatbuffers	1.10.0	17632	491.5	10	3	Apache Arrow
yas	7.0.2	17416	302.7	7	1	Partial. (JSON format is supported but not fully comply)
yas-compact	7.0.2	13321	2063.34	1	7	Partial. (JSON format is supported but not fully comply)

serializer name

Size : yas-compact < thrift-compact < MessagePack < protobuf < avro < thrift-binary < yas , cereal < boost < flatbuffers < capn proto

Time : yas < capn proto < flatbuffers < cereal < thrift-binary < boost < yas-compact < protobuf < MessagePack < thrift-compact < avro

JSON Performance

- CArchive is using JsonCpp library.(https://github.com/open-source-parsers/jsoncpp)

Libraries

43 libraries are successfully benchmarked. They are listed in alphabetic order: (Libraries section of https://github.com/miloyip/nativejson-benchmark)

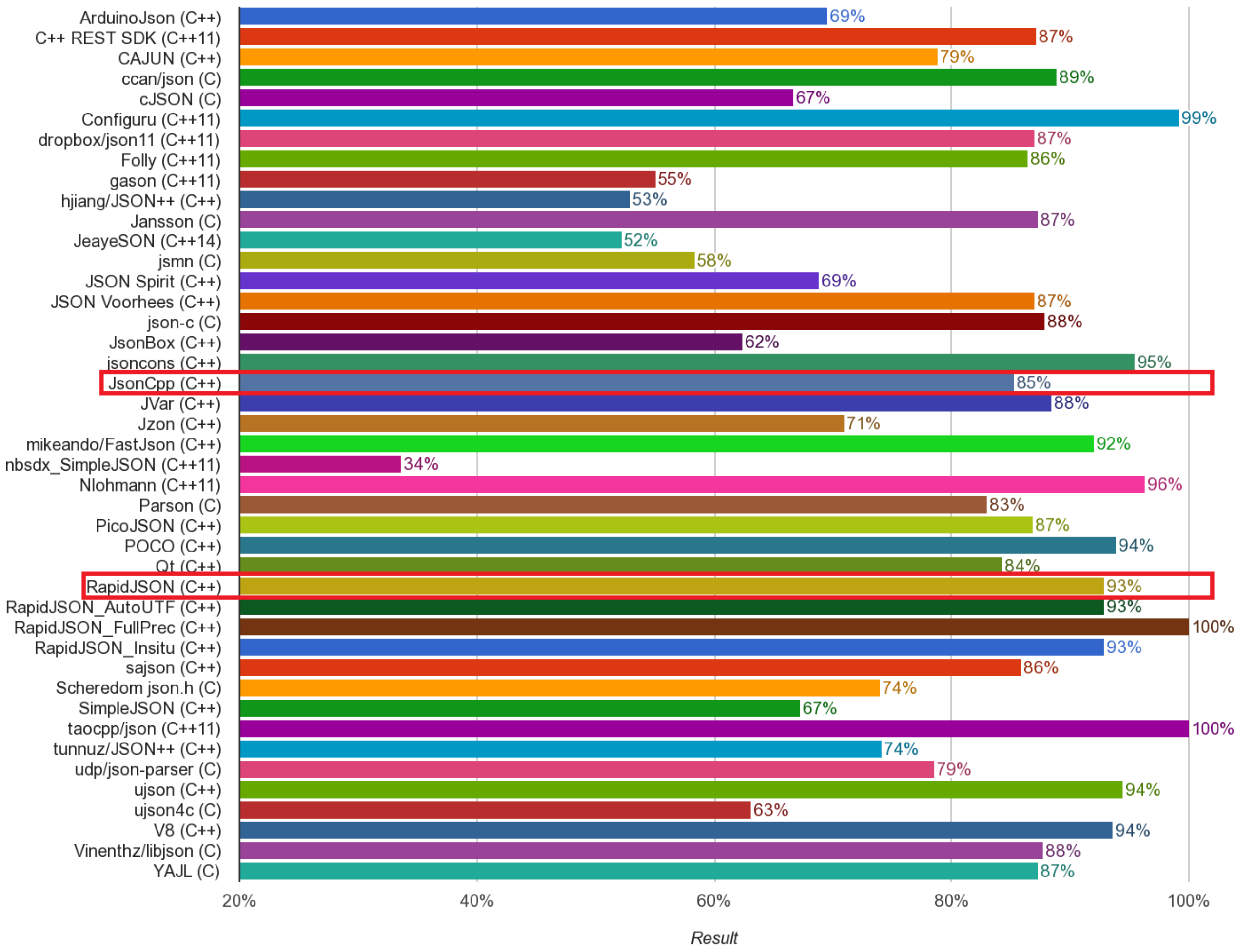
JsonCpp vs. RapidJSON

The followings are some snapshots from the results of MacBook Pro (Retina, 15-inch, Mid 2015, Corei7-4980HQ@2.80GHz) with clang 7.0 64-bit.

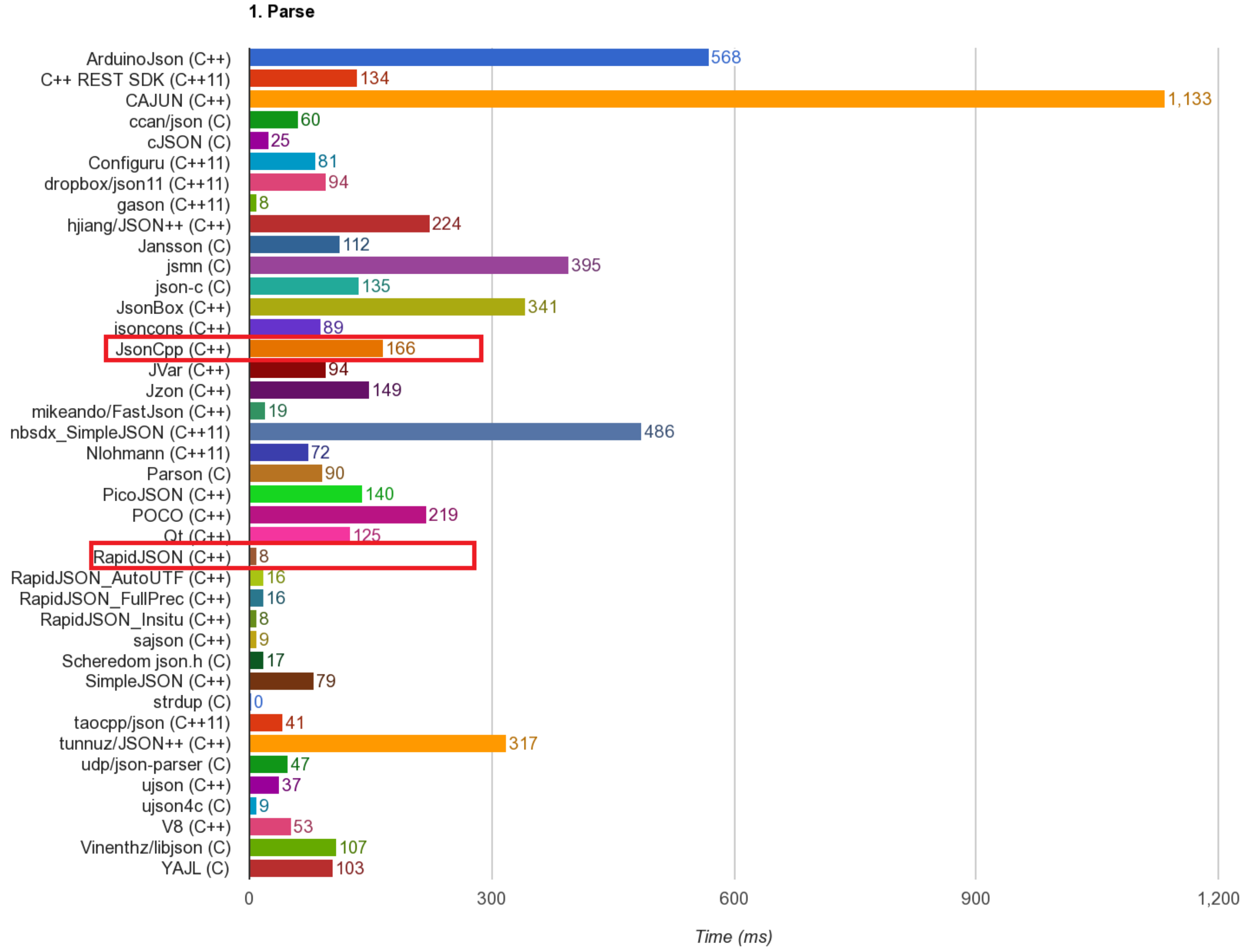
Overall

More detail information: https://rawgit.com/miloyip/nativejson-benchmark/master/sample/performance_Corei7-4980HQ@2.80GHz_mac64_clang7.0.html#1.%20Parse

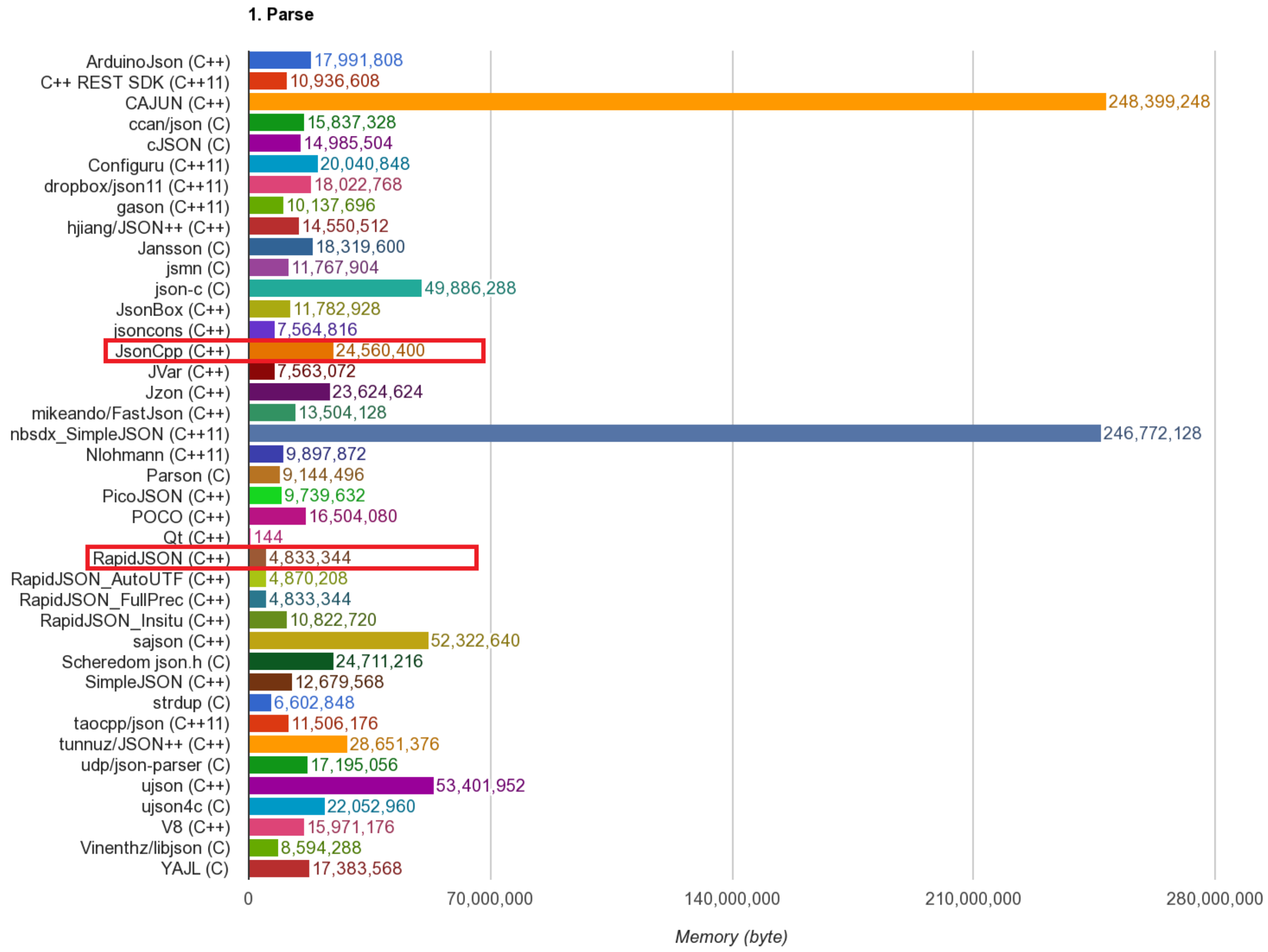
0. Overall



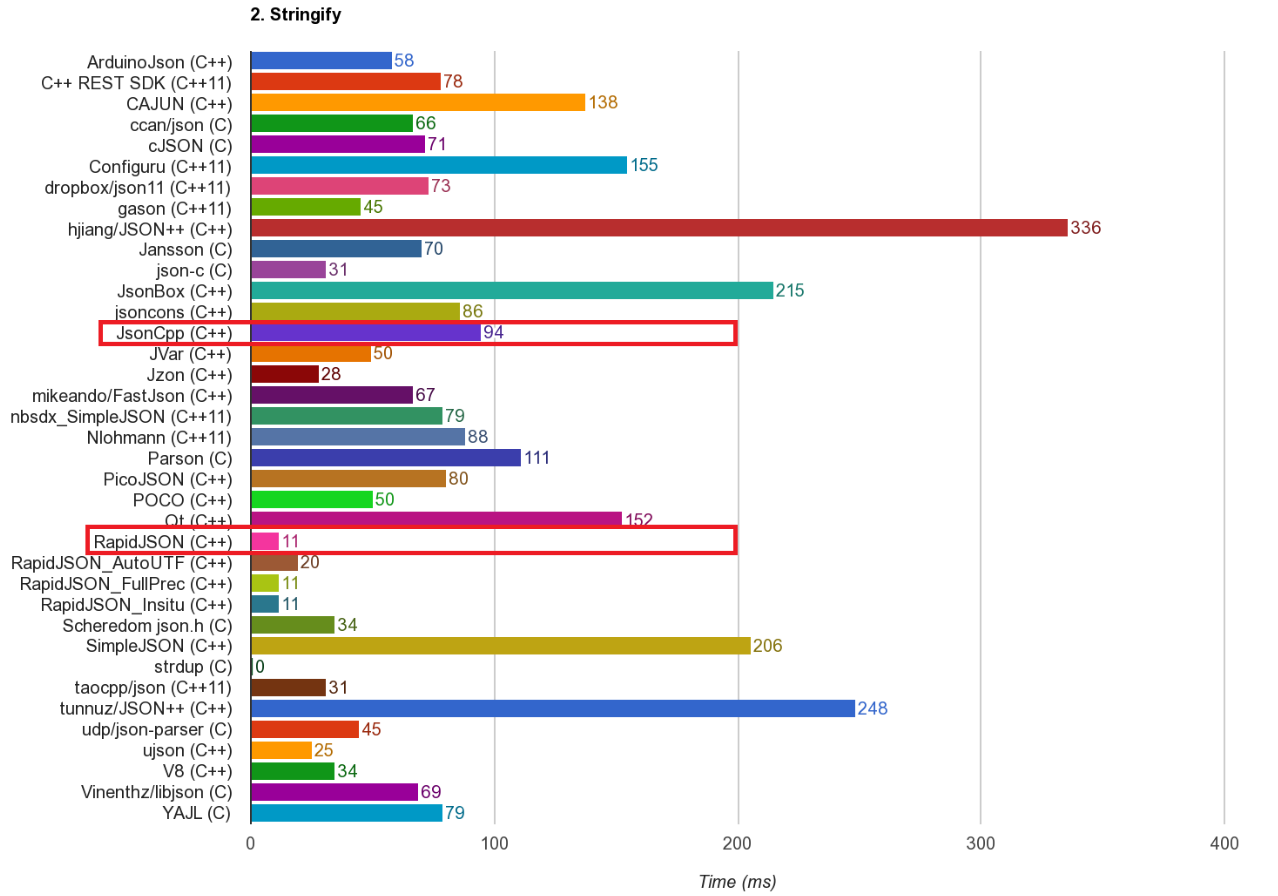
Parsing Time



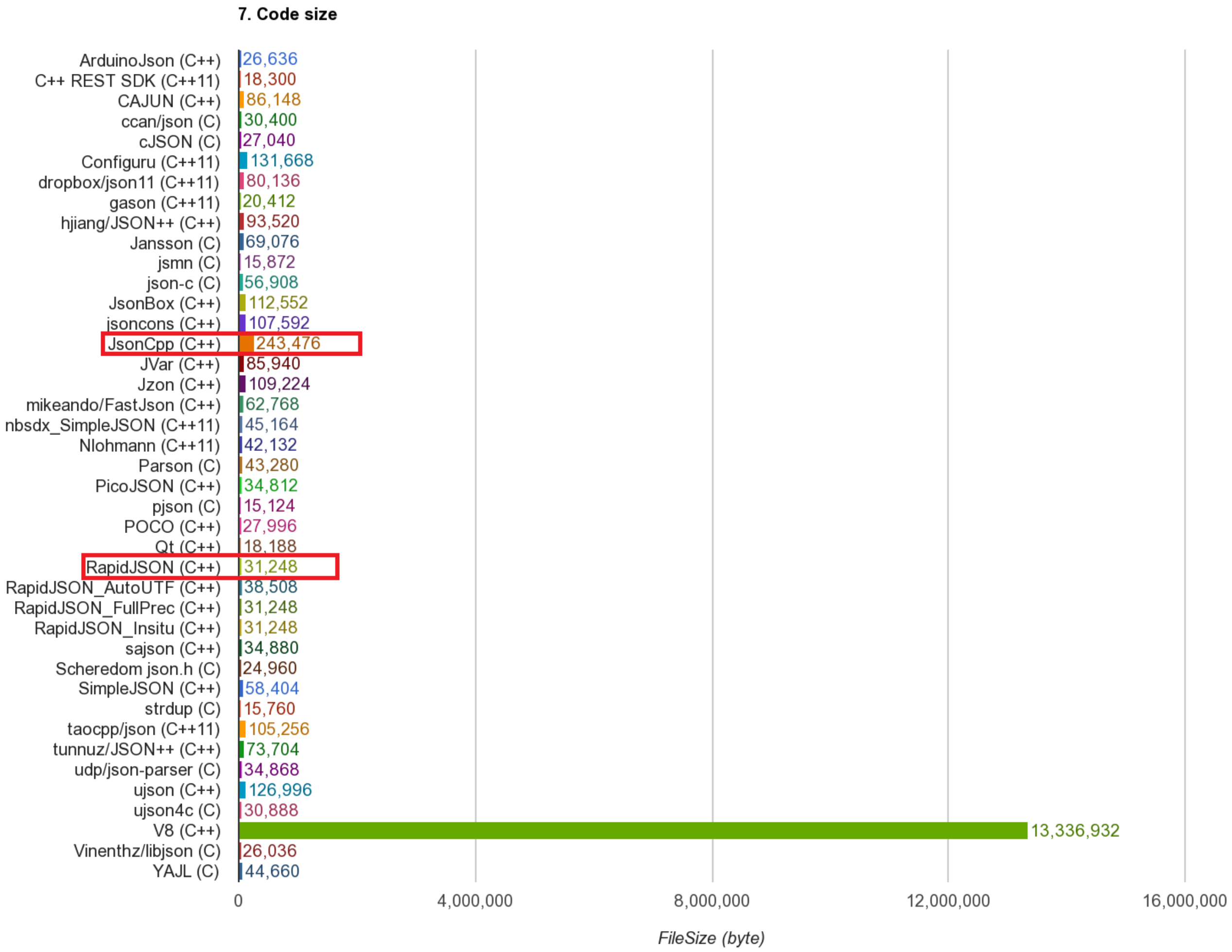
Parsing Memory



Stringify Time



Code size



Conclusion

According to serializer library benchmark, I found that we could use cereal library. Cereal serialization library supports JSON format. So it is human-readable.

And I searched the Json library performance benchmark result. According to the result, RapidJson library is faster and smaller than JsonCpp library.

If CArchive struct is used continuously, we can try to change native JSON library(JsonCpp library) to RapidJson.

Reference sites

https://blog.mbedded.ninja/programming/serialization-formats/a-comparison-of-serialization-formats/

https://en.wikipedia.org/wiki/Comparison_of_data-serialization_formats

https://github.com/thekvs/cpp-serializers

https://github.com/felixguendling/cpp-serialization-benchmark

https://uscilab.github.io/cereal/

https://github.com/miloyip/nativejson-benchmark

No labels

ATLASSIAN