Blaze: Compiling JSON Schema for 10× Faster Validation

Juan Cruz Viotti
jviotti@sourcemeta.com
Sourcemeta
Kuusalu, Estonia

Michael J. Mior
mmior@mail.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

ABSTRACT

JSON Schemas provide useful guardrails for developers of Web APIs to guarantee that the semi-structured JSON input provided by clients matches a predefined structure. This is important both to ensure the correctness of the data received as input and also to avoid potential security issues from processing input that is not correctly validated. However, this validation process can be time-consuming and adds overhead to every request. Different keywords in the JSON Schema specification have complex interactions that may increase validation time. Since popular APIs may process thousands of requests per second and schemas change infrequently, we observe that we can resolve some of the complexity ahead of time in order to achieve faster validation.

Our JSON Schema validator, Blaze, compiles complex schemas to an efficient representation in seconds to minutes, adding minimal overhead at build time. Blaze incorporates several unique optimizations to reduce the validation time by an average of approximately 10× compared existing validators on a variety of datasets. In some cases, Blaze achieves a reduction in validation time of multiple orders of magnitude compared to the next fastest validator. We also demonstrate that several popular validators produce incorrect results in some cases, while Blaze maintains strict adherence to the JSON Schema specification.

PVLDB Reference Format:
Juan Cruz Viotti and Michael J. Mior. Blaze: Compiling JSON Schema for 10× Faster Validation. PVLDB, 19(2): 279 - 291, 2025.
doi:10.14778/3773749.3773764

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sourcemeta-research/jsonschema-benchmark.

1 INTRODUCTION

Web APIs commonly accept payloads in the semi-structured JSON format that enable significant flexibility in the input that is accepted [13, 19, 20]. This is achieved with nested objects and arrays in JSON data structures that do not require the exact format to be clearly defined. Despite this flexibility, validating that data received by an API is in a specific format is important for both correctness and security. In order to validate payloads, such APIs commonly make use of the JSON Schema standard. JSON Schema allows API developers to validate the structure of a JSON payload including required properties, data types, and other features. The declarative approach of JSON Schema makes validation easier to write and maintain than explicit validation instructions. JSON Schemas are expressed in JSON format with keywords that restrict the set of JSON documents that are valid according to the schema. An example of a simple JSON Schema with both valid and invalid documents is shown in Figure 1. This particular example is straightforward, but interpreting schemas can be complex due to the interaction of various keywords in the JSON Schema specification.

There are a wide variety of tools and standards that make use of JSON Schema. For example, OpenAPI [1] is a commonly used format to define the structure of Web APIs. OpenAPI uses JSON Schema to describe request and response payloads. Validating requests according to JSON Schemas is a common practice, placing JSON Schema validation in the critical path of responding to a request. This means that any delay introduced by validation has an impact on latency. Numerous studies have shown that even small increases in latency can have a negative impact on the performance perceived by users [3, 18]. Our goal with Blaze is to minimize the latency of validating documents, even for large, complex schemas.

Attouche et al. [4] have shown that the latest dialect (2020-12) of the JSON Schema specification is PSPACE-complete with respect to the size of the schema. In particular, this is caused by the use of dynamic references, which are used to implement generic types or extend recursive schemas. However, they also identify that some validators incur a significant overhead from implementing dynamic references even when a particular schema does not actually make use of this feature. As we discuss further throughout the paper, we use open source schemas collected from GitHub to analyze the usage of JSON Schema in practice. Among the more than 31,000 schemas collected, we found only 10 instances of dynamic references. This means that some validators incur significant overhead for a feature of JSON Schema that is rarely used. We do not focus specifically on dynamic references in this work, but we do describe a method for converting some instances of dynamic references to static references during compilation. Our approach for doing so incurs no overhead at validation time. We also implement several other optimizations ahead of time through a precompilation process that transforms a provided schema into a list of instructions that enable efficient validation.

While validation happens often, schemas change relatively infrequently. This suggests that we can afford to invest extra time in the compilation process in order to achieve faster validation in the future. Across the schemas we analyzed from GitHub, the average time between commits to an individual schema over 65 days. Since schemas are changed infrequently, we can safely invest time in schema compilation to use for later validation. This is the

[1] https://www.openapis.org

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 2 ISSN 2150-8097.
doi:10.14778/3773749.3773764

```
{ "$schema": "https://json-schema.org/draft/2020-12/schema",
  "properties": {
    "firstName": {"type": "string", "maxLength": 100},
    "middleName": {"type": "string"},
    "lastName": {"type": "string", "maxLength": 100},
    "age": {"type": "integer", "minimum": 0}
  },
  "required": ["firstName", "lastName"] }
```

(a) JSON Schema example

```
{ "firstName": "Douglas",
  "lastName": "Crockford",
  "age": 69 }
```

(b) Valid document

```
{ "firstName": "Jason",
  // Missing lastName
  "age": 20 }
```

(c) Invalid document

Figure 1

approach that we take with Blaze. Our contributions in this paper are as follows: 1) an optimized low-level DSL for schema validation designed for semi-structured data validation, 2) a mapping from JSON Schema to our DSL that incorporates several unique optimizations, and 3) an execution engine that can efficiently validate documents according to instructions from our DSL.

Section 2 describes the schema validation language that is the result of schema compilation. We then describe how we compile from JSON Schema to this language in Section 3. Section 4 introduces several important optimizations to our compilation process and Section 5 describes the Blaze executor along with a complete example. Section 6 provides a comprehensive evaluation comparing Blaze to many existing evaluators using a variety of schemas. Finally, we discuss related and future work and conclude in Sections 7-9.

2 SCHEMA VALIDATION LANGUAGE

Like JSON Schema, by default, the DSL we define for validation is a constraint language [2]. In other words, the language is permissive, meaning that all values are accepted unless rejected by some instruction. To restrict the set of documents accepted, we introduce a list of instructions that potentially cause validation to fail if the document does not meet a specific set of conditions. Each instruction contains a location in the document it applies to and some assertion on the contents of that portion of the document. In the following sections, we first introduce basic instructions that apply to single values and then supplement these with instructions that loop over nested values and logically combine the results from multiple instructions. Note that by convention all our instructions start with uppercase letters while JSON Schema keywords start with lowercase letters. We also introduce some basic optimizations we apply to these instructions to improve performance. Further higher-level optimizations are discussed in Section 4.

2.1 Basic Instructions

The most basic instructions implemented in our validation language operate on single values. For example, the `TypeAny` instruction validates that a value is one of a given set of types. The instruction `TypeAny /foo ["string", "number"]` validates that a value of the property `"foo"` is either a string or a number. Similarly, `EqualsAny` validates that a value is one of a given set of specific values. Other instructions apply only to a specific type of value and are ignored for values of other types. These instructions have preconditions that check whether a value has a particular type before continuing execution. As we show in Section 3, this allows us to support JSON Schema keywords that only apply to specific types of values. The remaining basic instructions are summarized in Table 1. Each of these instructions have a precondition on their corresponding type.

Instruction	Type	Purpose
`DefinesAny`	object	specific properties exist
`PropertyDependencies`	object	if a property exist, other properties must also exist
`ObjectSize`	object	validates the number of properties in an object
`PropertyType`	object	an object has a property of a specific type
`Regex`	string	a specific regular expression matches
`StringSize`	string	validates the length of a string
`StringType`	string	validates complex string formats such as URIs
`Unique`	array	all array elements are unique
`ArraySize`	array	validates the size of an array
`Less/Greater/Equal`	number	validates the range of numeric values
`Divisible`	number	validates whether a number is divisible by a given value

Table 1: Basic instructions and their corresponding types

2.2 Loops

In several cases, we do not know in advance the set of values that must be evaluated to validate a schema. This occurs with arrays or objects where the set of keys is not predefined by the schema in advance. There are three specific scenarios we need to consider. Looping over object keys, object values, and array items. Looping over the keys of an object (ignoring values) is implemented using the `LoopKeys` instruction. Each key is validated against a given set of string instructions (e.g., the key must match a given regular expression or meet minimum/maximum length requirements). There are four different variants of looping over values of an object. `LoopProperties` validates *all* values according to a single set of instructions. `LoopPropertiesExcept` validates all values except those whose corresponding keys matching either a given regular expression or a static list of keys. `LoopPropertiesRegex` validates only values whose corresponding keys match a given regular expression. These first three variants loop over the schema, with each instruction matched against the instance being validated. The final two variants loop over the instance and then look up instructions to execute within the schema. `LoopPropertiesMatch` contains a set of keys and a list of instructions for corresponding values to validate against. The final loop instruction, `LoopPropertiesMatchClosed`, is used when all properties in an object must be explicitly defined in advance and the loop must validate all properties.

For arrays, there are two types of loop instructions. The first is `LoopItems` which validates all items in an array against a given set of instructions. A variant of this instruction, `LoopItemsFrom`, skips the first n items in the array, which as we show later, is useful for validating against some JSON Schema keywords.

2.3 Logical Operators

In some cases, it is desirable to combine instructions based on logical operators, such as requiring any of a given set of conditions to hold. Accordingly, we define a set of instructions that conditionally execute other instructions based on a set of conditions. Firstly, we have the `LogicalCondition` instruction that executes a different set of instructions based on whether a given condition is true or false. Second, we have a set of instructions to combine the results of

multiple steps using logical operators. These include `LogicalAnd`, `LogicalOr`, `LogicalXor`, and `LogicalNot`. These operators will short circuit where possible if failure to validate is guaranteed without evaluating all steps. For example, if the first instruction inside `LogicalAnd` evaluates to false, execution does not need to continue since the final result can already be determined.

2.4 Control Flow

It is common when validating semi-structured data to reuse part of a schema to validate similar structures in different locations with a document. Accordingly, we introduce the `ControlLabel` and `ControlJump` to enable this use case. The `ControlLabel` instruction contains a list of child instructions that are executed when the label is first encountered. The `ControlJump` instruction is able to return to these labeled instructions at any future point when the schema is evaluated. As we discuss in Section 3, we use these instructions to support references within a JSON Schema.

2.5 Instruction Set Optimization

Instruction	Condition
`WhenType`	Specific type
`WhenDefines`	Object that defines specific properties
`WhenArraySizeGreater`	Array with a minimum size
`WhenArraySizeEqual`	Array with a maximum size

Table 2: Logical condition optimizations

In general, we found that executing a smaller number of slightly more complex instructions leads to more efficient validation. This is largely because dynamic dispatch of individual instructions during validation can be costly. Accordingly, we introduce variants of several of our instructions analogous to the CISC approach of many modern processors [12]. For example, we define an instruction `StringBounds` that combines several separate instructions: `Type` to validate that the value is a string along with `StringSizeGreater` and `StringSizeLess` to validate that the length of the string is within a given range. In this vein, we also define variants of instructions ending with `Any` when only a single value is required. While the `EqualsAny` instruction checks if a value is one of a given list of values, the `Equals` instruction checks against a single value. This avoids a loop and yields modest efficiency gains. Another scenario we optimize is `LogicalCondition`. There are a number of specific conditions for which we define separate instructions as is shown in Table 2. We plan to explore further opportunities for optimization in the future. We note that we attempted to have fewer, more complex instructions, but we observed a negative impact on performance.

3 JSON SCHEMA COMPILATION

Our approach to convert JSON Schemas to our validation language follows a similar approach to the formalization of Attouche et al. for the 2020-12 dialect [4] of JSON Schema. Unless otherwise specified, all schemas we refer to use this most recent dialect. Blaze also supports JSON Schema dialects 4, 6, 7, and 2019-09 in a similar fashion to 2020-12 which we discuss here. In this work, we focus only on determining whether a document is valid and we ignore annotations and dynamic references. While Blaze supports both of these features, we save their discussion for future work. The formalization of Attouche et al. allows individual keywords to be evaluated sequentially subject to some ordering constraints [10]. We start with keywords that can be evaluated independent from others and then describe the operation of dependent keywords and how we handle references within documents.

3.1 Independent Keywords

Some of the keywords in JSON Schema can be evaluated against a value in any order without consideration for the effect of adjacent keywords. These keywords are referred to as *independent* keywords. We can exploit this independence to place instructions for more costly keywords first, saving processing time in case an earlier keyword causes validation to fail. For example, checking string length before validating a regular expression. There are two types of independent keywords. The first are *assertions* that have atomic values and define simple constraints. The second are applicators which themselves contain schemas to validate more complex values.

3.1.1 Assertions. For an example of an assertion, consider `minimum` that requires a numeric value be greater than a lower bound. In JSON Schema, many assertion only apply to a value of a given type. In this case, `minimum` only applies to numeric values. We therefore convert the application of the `minimum` keyword to the keyword `Greater` in our validation language, that also only applies to numeric values. Each assertion is mapped to instructions in our validation language that validate the assertion. Note that in the process of converting from assertions to instructions in our validation language, we also apply several static optimizations. For example, if the schema defines the type of a value to be anything other than `integer` or `number`, the `minimum` assertion is redundant can be ignored. No validation instructions generated since `minimum` only applies to numeric values. Such unnecessary assertions sometimes occur due to errors in authoring the schema. We plan to make use of these observations to develop a tool to highlight possible errors or optimizations in a schema.

3.1.2 Applicators. Several applicators are independent keywords, but the value of an applicator may be a schema. For example, `propertyNames` is an applicator that applies a schema to the keys of a JSON object. When compiling applicators, we recursively compile the corresponding subschema. Applicators may be nested recursively and our compilation process continues recursively as needed.

The applicators `anyOf`, `oneOf`, `allOf`, and `not` compile to the logical instructions `LogicalOr`, `LogicalXor`, `LogicalAnd`, and `LogicalNot` respectively. These applicators indicate that a location in the JSON document specify which of a number of schemas a given location in a document must match (or not). As mentioned previously, these instructions can short-circuit evaluation and skip instructions that are not necessary to determine whether a document is valid.

For conditional application of schemas based on `if`, `then`, and `else`, we handle all compilation upon encountering the `if` applicator. The `if` applicator specifies a schema. If the value at the instance location is valid according to the schema, then the schema specified under `then` is applied, otherwise the schema specified under `else` is applied. To compile into our validation language, we recursively compile each of these schemas and then generate

a `LogicalCondition` instruction that jumps to the appropriate schema for further validation. As a minor optimization, if there is no schema specified for `if`, then it will be considered true, and we can avoid compiling the `else` case.

For arrays, `contains` is an applicator that specifies a schema that must validate against at least one array element. The keywords `minContains` and `maxContains` can be used to control the number of required elements. We emit the `LoopContains` instruction to validate that the array contains the required number of elements. We note that several small optimizations are possible here. First, if `minContains` is `0` and `maxContains` is not set, we do not need to emit any instructions at all. Second, if the schema provided to `contains` is `true` (that is, all values are accepted), then we only need to validate the array length and `minContains` and `maxContains` have the same effect as `minItems` and `maxItems`. The other independent array applicator is `prefixItems` that validates that items in a prefix of an array match a given array of schemas. In this case, we recursively compile each schema and then generate an `ArrayPrefix` instruction that performs the validation of all items.

Objects also have several independent applicators that can be evaluated in any order. The `properties` and `patternProperties` applicators are very similar. Both specify schemas that must be valid against values inside an object. The `properties` applicator specifies string literals for keys while `patternProperties` specifies regular expressions to match keys. Note that if an expression listed under the `properties` keyword also matches a regular expression listed under `patternProperties`, then the value must be valid according to both schemas. This is what allows these two keywords to be evaluated independently. The schemas for each value in the `properties` and `patternProperties` applicators are recursively compiled and the `LoopProperties` and `LoopPropertiesExcept` instructions are used for their evaluation.

Finally, the `propertyNames` keyword specifies a schema that must be validated against object keys. In this case, we recursively compile this schema and generate a `LoopKeys` instruction to validate all keys in an object against the schema.

3.2 Dependent Keywords

There are two different types of dependent keywords: first-level and second-level dependent. Along with independent keywords, these define three tiers of keywords we must consider. Independent keywords can be evaluated in any order without considering any other keywords. First-level dependent keywords have some dependencies on other keywords. However, we introduce mechanisms in our compilation process to allow instructions compiled from these keywords to also be executed in any order. Second-level dependent keywords may depend on the evaluation of the remainder of the schema and cannot necessary be resolved statically.

3.2.1 First-Level Dependent Keywords. There are two first-level dependent keywords: `additionalProperties` (for objects) and `items` (for arrays). Both serve to validate values in objects (or arrays) that are not validated by other keywords. The `additionalProperties` applicator specifies a schema that every property not already validated by either `properties` or `patternProperties` must match. While the behavior of `additionalProperties` depends on the values of the `properties` and `patternProperties` keywords, we can resolve this dependency statically to also allow instructions generated from `additionalProperties` to be executed in any order. To do so, we examine keywords adjacent to `additionalProperties` and collect the static set of keys and the regular expressions that are used in the `properties` and `patternProperties` keywords. We then generate an instruction that skips these properties to validate the remaining additional properties.

However, we also optimize the common case where the value of `additionalProperties` is a Boolean. A value of `true` indicates that objects are permitted to have any additional properties without adhering to any particular schema. In this case, we generate no instructions for validation. When `additionalProperties` is set to `false`, only properties explicitly defined using the `properties` or `patternProperties` keywords are allowed. In this case, we modify the instructions generated for these keywords to fail if any additional properties are encountered.

The `items` applicator functions similarly for arrays. It validates all items in the array not already validated by `prefixItems`. In this case, we recursively compile the schema for the items and use the `LoopItems` instruction to validate the items. The first n items are skipped by using the `LoopItemsFrom` instruction if the `prefixItems` applicator is also used with n items. In this way, the behavior of `items` is dependent on `prefixItems`, but the two can still be evaluated in any order.

3.2.2 Second-Level Dependent Keywords. The two second-level dependent applicators `unevaluatedProperties` and `unevaluatedItems` serve a similar purpose. They provide a schema that values in arrays or objects must be validated against if they have not already been validated by another keyword elsewhere in the schema. This is similar to the first-level dependent keywords except that second-level dependent keywords can "see through" other applicators. Consider the example use the of `unevaluatedProperties` keyword in Figure 2 adapted from the JSON Schema documentation[2]. The properties `city` and `state` are contained inside the applicator `allOf`. If the schema used the similar keyword `additionalProperties` instead, the corresponding document would be invalid. This is because the keyword `additionalProperties` only applies to adjacent keywords such as `properties`. It would not allow the properties that are defined inside the `allOf` applicator.

In contrast, the behavior of `unevaluatedProperties` is defined by whether a property has been evaluated by another keyword.Most implementations add an *annotation* to an object key when it has been evaluated by one of these keywords, regardless of whether that evaluation occurred inside another applicator. Any key without such an annotation is then evaluated by `unevaluatedProperties`. However, we find that maintaining annotations can add significant overhead and that in many cases, they are not necessary. Instead, we can make a static pass over the schema to identify properties that will be evaluated by other keywords. In the case of the schema in Figure 2, we can generate instructions for `unevaluatedProperties` in the same way as for `additionalProperties` by statically identifying keys that are guaranteed to be evaluated. In the case where `unevaluatedProperties` is `true` or all properties are guaranteed to be evaluated by other keywords, we do not generate any instructions for `unevaluatedProperties`.

[2]https://json-schema.org/understanding-json-schema/reference/object

```
{"allOf": [{
    "type": "object",
    "properties": {
      "city": {"type": "string"},
      "state": {"type": "string"} }
    } }],
  "properties": {
    "name": {"type": "string"}
  },
  "unevaluatedProperties": false}
```

(a) JSON Schema

```
{
  "name": "Bob",
  "city": "Washington",
  "state": "DC"
}
```

(b) Valid document

Figure 2: Example use of `unevaluatedProperties`

3.3 Static References

When references are used in a schema, it is advantageous to allow the reuse of validation instructions. In fact, this is necessary for recursive references to avoid infinite loops in the generation of instructions. To handle references, we use the instructions `ControlLabel` and `ControlJump`. `ControlLabel` is used the first time the `$ref` keyword is encountered. The destination of the reference is recursively compiled into a set of instructions. Any subsequent uses of `$ref` throughout the schema use the `ControlJump` instruction to reuse the existing set of instructions. However, for non-recursive references, we note that jumping between instructions can add additional overhead by reducing cache efficiency. To mitigate this overhead, we eliminate the use of labels and jumps for non-recursive references that are repeated five or fewer times. In these cases, we simply repeat the instructions compiled from the destination of the reference. This avoids producing a significantly larger number of instructions for large numbers of repeated references. We plan to explore improvements to this heuristic in future work.

3.4 Dynamic References

As past work has identified [21], dynamic references introduce significant challenges into JSON Schema validation. Dynamic references are designed to allow targets of references to be changed when schemas are extended. This means that the target of a reference cannot easily be statically determined at schema compilation time. Instead, the resolution of the reference is dependent on the context determined at evaluation time. This complicates validation process and it has been shown that validation with dynamic references is PSPACE-complete [4, 21].

We leave a full explanation and evaluation of our approach to handling dynamic references to future work. However, we make two important observations. First, as others have observed [4], it is possible to remove dynamic references from a schema. In the general case, this can result in an exponential increase in the size of the schema. Our second observation, as we discussed in Section 1, is that dynamic references are very rarely used. Furthermore, when dynamic references are used, there are often very few possible contexts for each dynamic reference. In this work, we focus our evaluation on dynamic references with a single possible context. In this case, a dynamic reference can be directly replaced by a static reference since we are guaranteed that the single available evaluation context will be used. The dataset used in our evaluation in Section 6 contains two schemas, `openapi` and `cql2` that contain a dynamic reference with a single possible evaluation context that is transformed using this approach.

3.5 Correctness

As discussed previously, we base our formalization on that of Attouche et al. [4]. The key points addressed in their formulation are the order of keyword validation and the semantics of each individual keyword. We started by creating careful mappings from each JSON Schema keyword into instructions that validate the keyword. We carefully implement these instructions to ensure they follow the rules defined by the JSON Schema specification As described in Section 3, we are able to evaluate instructions for independent and first-level dependent keywords in any order. We guarantee that second-level dependent keywords are evaluated after their dependencies in order to ensure correct validation. As discussed in Section 6.1, we also validate our implementation against the official JSON Schema Test Suite [7]. The test suite contains hundreds of tests for each JSON Schema version specifically designed to test difficult edge cases. In particular, the test suite for the latest dialect supported by Blaze contains more than 1,200 tests. As we discuss later, even several popular validators fail at least one case in this test suite. Blaze passes every test case, increasing our confidence in its correctness. We leave the possibility of formal verification of our implementation as future work.

4 OPTIMIZATIONS

We implemented Blaze in approximately 11,000 lines of C++20. Our implementation is open source and available on GitHub [3] under the AGPL-3.0 license. Although our description of Blaze has focused on the most current dialect (2020-12), we also support all previous dialects currently in use (4, 6, 7, and 2019-09). We have put significant effort into optimizing each of the instructions executed by our validator. While several optimizations are common to general software development, several unique optimizations stem from observations about the nature of data present in both JSON Schemas and typical JSON documents. We describe these optimizations in the following subsections. Specifically, we optimize the hash function used in data structures in our validator, unrolling instructions containing loops, and optimizing specific patterns found in regular expressions.

4.1 Semi-perfect Hashing

One observation about both schemas and documents is that in most cases, the strings used as JSON keys are relatively short. In our corpus of schemas we use for evaluation, 95% of the keys defined in the schema are 13 characters or shorter. We make use of this fact to optimize hashing of these strings for comparison and define a hash function that results in no collisions for short strings. Unlike minimal perfect hash functions [11], we do not aim to avoid collisions in all cases, but focus on the most common case in practice.

The need to compare strings occurs frequently in schema validation for tasks such as checking the presence of required properties, whether a string matches a desired constant, and several other cases. Hash functions used in library implementations such as the default MurmurHash[4] in the C++ standard library, are designed for general use and are not optimized for any particular use case. We decide to optimize our hash functions for short strings at the expense of increased likelihood of collision for longer strings. In fact, our goal

[3] https://github.com/sourcemeta/blaze
[4] https://github.com/aappleby/smhasher

is to avoid the need to compare strings at all in the common case by making our hash function one-to-one for small strings.

We start by defining the output size of our hash function to be 256 bits. This is represented as four 64-bit bit integers (or two 128-bit integers on supported platforms), giving a total of 32 bytes. We use the final 31 bytes when hashing strings that are 31 bytes or less. In this case, the first byte is set to zero and the remaining bytes are copied directly from the string value. This first byte is used in the case where the string is larger than 31 bytes. In this case, the hash function is the sum of the size of the string and the first and last characters, modulo 255, plus one (ensuring the value is non-zero). This allows computing a hash in constant time for longer strings.

To compare hashed strings, we first check that the first byte of each hash is zero, indicating that the two hashes both correspond to strings of 31 bytes or less. At this point, we can compare the hash values directly to determine equality since the remaining bytes of the hash are exactly the bytes of strings. Furthermore, we can do this comparison with basic integer comparisons instead of the need to check character by character. If both strings are longer than 31 bytes, then it is still necessary to compare strings in the case of a hash collision. Since this hash function is efficient to evaluate, we store the hash of strings as part of the process of parsing documents. We then make use of this hash function and the optimized comparison anywhere strings are compared. We also note that since documents generally have a small number keys, we make use of a vector data structure to store keys instead of a hash map since looping over the small number of entries is more efficient than dealing with the indirection inherent in hash tables.

While we would expect using a one byte hash for longer strings would increase the rate of collisions, this case is rare given our analysis of the common size of strings. In the corpus of thousands of schemas we analyzed from GitHub, we found that over 98% of keys defined JSON Schemas were less 32 bytes. This approach also has the advantage that we can compare short strings (less than 32 bytes) by comparing only their hash values. We also note that the majority of JSON objects have a very small number of properties, so the rate of collisions is likely to remain low in practice. We consider a common use case of our hash function which is to build a hash table for properties in object. We looked at all JSON objects with more than one key across all of our test schemas and found that our hash function has a collision rate of less than 0.9%. MurmurHash has zero collisions on the same dataset. However, our hash function still achieves better overall performance since we can compute the hash value in constant time. Furthermore, when comparing short strings (less than 32 bytes), we only need to check their hash values and we can avoid string comparison entirely. This is because when using our hash function, any strings less than 32 bytes are guaranteed to be equal if their hash values are equal.

We provide an example of the use of our hash function in Figure 3. Note that our implementation packs hash values into large integers, but we represent values as byte arrays here for illustrative purposes. Since the first two strings are less than 32 bytes, they can be compared exactly by only comparing their hash values, avoiding string comparison entirely. The final two strings are longer than 31 bytes, so we only make use of the first byte, which is calculated based on the string length and the first and last characters. For these two strings, the hash value is the same since they have the same

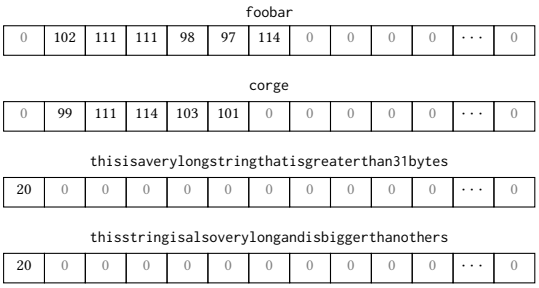

Figure 3: Semi-perfect hashing example

length and first and last characters. In this case, as with any hash function, we must compare the entire string in order to check for equality. However, as discussed previously, we expect this to be rare since most strings we encounter are short. We provide a detailed evaluation of the benefits of our hash function in Section 6.2.3.

4.2 Unrolling

While loop instructions can be useful for flexibility, as with traditional compiler optimization, we find that loop unrolling is sometimes a useful optimization. There are two cases where we apply unrolling in Blaze: property validation in objects and validation using references. We describe these cases below along with the heuristics we use to decide when they are employed. The heuristics were chosen based on observed performance on example test cases. We leave further tuning of these heuristics to future work.

When validating properties, there are two different approaches that we can take. The first was previously described in Section 2.2. In this case, we loop over all the properties of an object and look up the appropriate instruction to use for validation based on the property. However, when most properties are required, it can be more efficient to generate instructions that check each property individually instead. Specifically, we avoid generating a loop if there are 5 or fewer properties or if at least one quarter of the properties are required. We also have one additional heuristic that always unrolls loops inside of instructions generated from the `oneOf` or `anyOf` applicators. This increases the likelihood that these operators will be able to quickly short-circuit.

When a part of a schema (typically a definition) is referenced elsewhere, our default approach is to generate instructions for the subschema being referenced and then jump to these instructions as discussed in Section 3.3. However, as with loops, these jumps can be detrimental to CPU cache performance. Therefore, we can instead replace the jump instruction with the necessary instructions to validate according to the referenced subschema. We perform this replacement if there are no more than 5 references to a particular subschema. We also skip this optimization if there are recursive references since they cannot be implemented using unrolling.

4.3 Regular Expressions

Some features of JSON Schema rely on the evaluation of regular expressions. This includes, for example, the `pattern` keyword validating that strings match a particular pattern and `patternProperties` that serves a similar function for object keys. We decided to make

use of `Boost.Regex` as our regex engine after observing significantly better performance than `std::regex` for our use case. Furthermore, we enable precompilation that trades off time at regex construction for faster matching. We also identified multiple cases where regular expressions can be further optimized. For example, many schemas define regular expressions such as `.*`, that effectively allow all strings. In addition, expressions such as `.+` are used to identify non-empty strings. In both of these cases, we can avoid using a regex engine entirely. In the first case, where any value is accepted, we can completely remove the regular expression check. In the second case where the string must be non-empty, we can simply check the length of the string. Note that the JSON Schema specification does not fully specify the behavior of regular expression matching, so we have chosen to allow `.` to match any character (including line breaks). We expect this will not affect many schemas in practice since line breaks within JSON strings are uncommon.

In a slightly more complicated scenario, regexes such as `^x-` are common in schemas to indicate strings that start with a particular pattern. (This specific regular expression appeared several hundred times in our corpus of GitHub schemas.) In this case, we can also avoid using regular expressions by simply checking for a string prefix. Finally, we also identified a pattern of regular expressions such as `^.{3,5}$` that effectively indicate a string must be between length 3 and 5. We can again avoid the use of regular expression matching simply by checking the length of the string. We implemented special cases for each of these scenarios. In the future, we plan to explore the conversion of regular expressions to finite automata at compile time to further reduce the matching overhead.

4.4 Instruction Reordering

As discussed in Section 3, when compiling to our instruction set from JSON Schema, we have some flexibility on the order instructions are executed. For complex objects with many properties, we observed that it can be effective to evaluate properties with smaller subschemas first. This has the benefit of potentially allowing validation to fail more quickly while executing fewer instructions. This optimization is also particularly effective in the presence of applicators such as `oneOf`. Failing to validate as quickly as possible in this case means we can more quickly find the correct subschema to validate against. We currently perform reordering based solely on the size of subschemas. It is possible that there may be more effective orderings based on the specific data being validated. For example, we may find that it would be more efficient to place instructions for properties that commonly fail to validate first even if the corresponding subschema is larger. Currently we take a data-agnostic approach and we leave further optimizations as future work.

4.5 Reducing Memory Allocation

When allocating dynamic data structures such as vectors or hash maps, we prefer to preallocate a small number of entries. Since most data structures used in our implementation remain quite small, this means we can often avoid further allocations by using this small existing pool. Furthermore, we optimize for the case of repeated evaluations of the same schema by preallocating a data structure that can be reused for multiple validations without reallocation. This includes data structures such as pointers to the current location in both the schema and the document being examined. We plan to explore further memory optimizations in the future such as alternative allocators.

5 INSTRUCTION EXECUTION

In Blaze, unlike many validators, we do not interpret the schema, but instead precompile it into a set of instructions that can be efficiently executed. This section describes the Blaze executor.

5.1 Executor Implementation

The executor takes a compiled schema as input and executes the instructions against a JSON document to produce a Boolean indicating whether the document is valid. We first start by describing the structure used to represent each instruction in more detail. Each instruction first contains the type of instruction, which is one of the values specified in Section 2 as well as the location in the instance the instruction applies to. Depending on the instruction, there may also be an associated value such as the expected length of a string or a list of subinstructions used in cases such as validating items in an array according to a set of conditions.

The Blaze executor is driven by a loop over the instructions to be executed. Each instruction is executed by first looking up the value to be validated from the instance. Locations in JSON instances are expressed using JSON Pointer notation [14] which is used to identify a specific location within a document to be validated. Note that we make heavy use of the hash function discussed in Section 4.1 to quickly match properties in an object when traversing pointers. Several instructions may have preconditions to be validated before they are executed. For example, the `Greater` instruction discussed in Section 3.1.1 only applies to integers. In this case, if the corresponding value in the instance being validated is not an integer, the remainder of the instruction will be skipped. For instructions that contain children, recursion into subinstructions is accomplished by recursively calling the evaluation function in the executor in a loop with each subinstruction. If any subinstruction fails to validate, the loop over subinstructions is terminated early and failure to validate is returned. All instance locations specified within instructions are relative to their parent instruction. We provide a full example of compilation and execution of a schema in the following subsection.

5.2 Execution Example

This section provides an example of schema compilation and execution in Blaze using the schema in Figure 4a. The schema defines two optional properties `"foo"` and `"baz"`. Each instruction in Figure 4b contains both a JSON Pointer and (optionally) an associated value. Note that here, `<empty>` refers to the empty JSON pointer, indicating that the root of the document will be validated. Instructions may have an optional precondition, that is indicated before the instruction with a question mark. In order to validate the two properties, Blaze first generates the `LoopPropertiesMatchClosed` instruction. The `Closed` variant is selected since `additionalProperties` is set to `false` and only the properties specified will be permitted. This means that while the instruction loops over keys within the JSON object, any key without an associated validation instruction will cause validation to fail. Also note that this instruction is only executed if the value is of type `object` since this is a precondition for

this instruction. That is, any values that are not objects, will cause this instruction to be skipped. Importantly, validation does *not* fail if the precondition for an instruction is not met.

Within the loop are two instructions for validating each of the two properties. The `AssertionGreaterEqual` instruction is generated from the `minimum`. Since it has the precondition `number?`, it will only apply to numeric values. This precondition applies to the instance path `/baz`. This means that any non-numeric values will be accepted by this instruction which matches the semantics of the `minimum` keyword. If the precondition is valid (the value at `/baz` is a number), then the instruction checks if this value is at least 5. The second instruction in the loop is `AssertionType`. Since this instruction has no precondition, it applies to all values. Specifically, this instruction validates that the value at the instance path `/foo` has type `string`. Note that the order of the two instructions within the loop are unimportant since Blaze will look up the appropriate instruction based on the associated instance path.

Since the `LoopPropertiesMatchClosed` instruction includes an `object` type precondition, before the final instruction, any non-object values would pass validation. However, since the schema specifies that value must be an `object`, Blaze emits one final instruction. `AssertionType` behaves the same as above and validates that the entire value is an object. Assuming that this assertion evaluates to `true`, the entire document is considered valid.

Finally, we briefly walk through the execution of these instructions against the schema in Figure 4c. The first instruction, `LoopPropertiesMatchClosed` begins execution since our document meets the precondition of being an object. This instruction then loops over the all the key-value pairs in the object. The first key, `"foo"` corresponds to the `AssertionType` instruction. Since the value, `"baz"` is a string, it passes validation. The second key, `"baz"` corresponds to the `AssertionGreaterEqual` instruction. The value is a number, so it passes the precondition, and it is greater than 5, so this instruction also evaluates to `true`. Note that at this point, if the document had any other keys, the `LoopPropertiesMatchClosed` instruction would fail to validate since no additional properties are allowed. However, since our document only has two properties, this instruction evaluates to `true` and we move on to the final instruction. Since our document is an object, this instruction also evaluates to `true` and the entire document is considered valid.

6 EVALUATION

We have two main focuses with our experimental evaluation: validating the correctness of our implementation and comparing the validation performance with existing JSON Schema validators.

6.1 Correctness

In order to experimentally verify the correctness of our implementation, we make use of the official JSON Schema Test Suite [7]. This test suite is maintained by the authors of the JSON Schema specification and is designed to exercise corner cases in JSON Schema validation and contains several hundred tests for each version of the specification for a total of over 6,000 tests. These tests verify that each keyword is implemented correctly and also test interactions between relevant keywords. However, we note that the test suite does not cover all possible cases. Indeed, we later discuss some implementations that pass this test suite but produce errors in our evaluation. Through our evaluation, we were also able to discover and report bugs in other JSON Schema validators not captured by the official test suite. More than 20 JSON Schema validators publish ongoing reports via Bowtie [8], a system that automatically runs the test suite against supported implementations. Our implementation is one of only twelve that achieves a perfect score for the 2020-12 dialect, confirming the correctness of our compilation process. In addition to this test suite, we also have over 16,000 lines of manually written test cases across the different dialects supported by Blaze.

6.2 Performance

In order to test the performance of our validator, we need a number of schemas as well as documents corresponding to each schema. We collected our schemas from the JSON Schema Store [5], a repository of JSON Schemas for various configuration file formats. All of the schemas make use of dialect 7 of the JSON Schema specification with the exception of `cql2` and `openapi` which use the 2020-12 dialect. For several of these schemas, there is a convention used to name files which are designed to be valid according to the schema. For example, files that use the `babelrc` schema are typically named `.babelrc` or `babelrc.json`. We use these file names to search for matching files on open source projects on GitHub. After finding a set of files, we validate them according to the corresponding schema in order to ensure that each document indeed matches the schema. A summary of the schemas, their size, and the number and size of documents collected is shown in Table 3. Note that to measure the size of each schema, we exclude keywords such as `description` that have no effect on validation. All datasets are available in our benchmark repository[6]. We also make available measurements for several other implementations that did not meet our selection criteria. We note that none of these implementations are faster than Blaze on any of our test datasets. Experiments are run on a machine equipped with two 8-core 2.10 GHz Intel Xeon Silver 4110. We note that while multiple cores our available, our implementation of Blaze is currently single-threaded. We leave the possibility of optimizing for parallel execution as future work.

We compare against a wide variety of validators across multiple programming languages. Validators were selected based on those available in the Bowtie test system that either pass the entire JSON Schema Test Suite or are significantly popular. Note that we exclude the implementations dev.harrel.json-schema and io.openapi-processor.json-schema-validator since they performed an order of magnitude slower than other validators in the majority of cases. We have also included ajv and the Python jsonschema package since these are both very commonly used, despite producing incorrect results in some cases. A summary of all the implementations we compare with is in Table 4. We run each implementation five times on each dataset and measure the time to compile the schema as well as the time for validating all instances. Each instance is validated sequentially and validation does not exploit parallelism.

6.2.1 Compilation. As previously noted, Blaze trades off an upfront period of compilation for faster validation at runtime. There

[5] https://www.schemastore.org/json/
[6] https://github.com/sourcemeta-research/jsonschema-benchmark

```
{ "type": "object",
  "properties": {
    "foo": {"type": "string"},
    "baz": {"minimum": 5}
  },
  "additionalProperties": false }
```

(a) JSON Schema

```
object? LoopPropertiesMatchClosed <empty>
    number? AssertionGreaterEqual /baz, 5
    AssertionType /foo, string
AssertionType <empty>, object
```

(b) Compiled schema instructions

```
{
  "foo": "bar",
  "baz": 7
}
```

(c) Valid document

Figure 4: Example of JSON Schema compilation with Blaze

Name	# Docs	Schema Size (KB)	Avg. Doc. Size (B)
ansible-meta	333	36.1	312
aws-cdk	483	0.7	1145
babelrc	794	6.5	140
clang-format	133	54.2	336
cmake-presets	967	84.0	2721
code-climate	2484	5.9	282
cql2	109	17.9	125
cspell	981	125.6	817
cypress	981	16.0	401
deno	987	22.4	1018
dependabot	967	9.4	403
draft-04	563	4.0	12631
fabric-mod	911	11.1	691
geojson	500	45.0	52433
gitpod-configuration	986	13.1	354
helm-chart-lock	3888	1.5	342
importmap	964	0.6	630
jasmine	980	3.6	133
jsconfig	981	59.5	177
jshintrc	966	11.8	429
krakend	47	377.7	2431
lazygit	280	87.8	276
lerna	985	4.6	172
nest-cli	1025	18.9	290
omnisharp	987	13.5	595
openapi	107	32.5	165548
pre-commit-hooks	985	9.6	549
pulumi	3807	7.7	251
semantic-release	794	3.3	460
stale	961	3.7	466
stylecop	983	11.5	567
tmuxinator	382	4.4	628
ui5	942	94.1	487
ui5-manifest	611	383.5	2356
unreal-engine-uproject	859	10.6	394
vercel	710	37.2	406
yamllint	984	25.5	351

Table 3: Datasets used for validator evaluation

Implementation	Lang.	Version	Correct	AOT	Stars
Blaze (Ours)	C++	1.0.0	✓	✓	<100
ajv	JS	6.12.6	✗	✓	>10K
Boon	Rust	0.6	✓	✓	<100
Corvus	C#	4.0.12	✓	✓	~100
jsonschema	Go	6.0.1	✓	✓	~1K
jsonschema	Python	4.23.0	✗	✗	~5K
JsonSchema.NET	C#	7.2.3	✗	✓	~1K
JSV	Elixir	0.2.0	✓	✗	<10
KMP	Kotlin	0.3.0	✓	✗	<100
NetworkNT	Java	1.5.3	✓	✗	~900
json_schemer	Ruby	2.3.0	✓	✗	~400

Table 4: Implementation details for each validator

are existing validators supporting ahead-of-time (AOT) compilation, which we indicate in Table 4. However, many validators have very basic precompilation compared to Blaze. In several cases, precompilation consists of only parsing and validating the schema.

We performed compilation five times on each of the schemas in our dataset and report the average in Figure 5. As expected, we can see that the compilation time tends to increase relative to the size of the schema. We note that since compilation is done for the purpose of speeding up evaluation, validators which are slower to

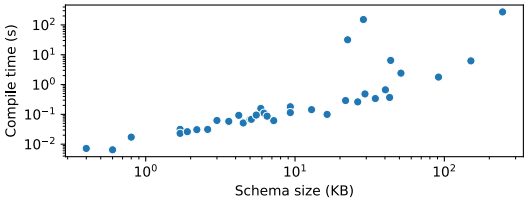

Figure 5: Blaze compilation time relative to schema size

compile may achieve a return on this investment after validating a sufficient number of documents. We have also not currently made any effort to optimize the compilation time of Blaze.

6.2.2 Validation. When measuring the validation runtime, we first measure the runtime immediately after compiling. We then perform a minimum of 100 more iterations of validation to warm up the implementation. Depending on the implementation, this will have the effect of warming caches, triggering JIT compilation, and other side effects that result in warm runs being faster. Warm runs reflect real-world situations such as an API gateway that validates a large number of incoming payloads according to a fixed schema. Runtimes for all implementations and datasets are reported in Table 5.

We note that several implementations produce failures on some schemas despite passing all the tests in the JSON Schema Test Suite. This is due to some tests being considered optional, which are not counted as failures since they exercise uncommon edge cases. For example, JSON Pointers that are used for references must be escaped if they contain either slashes or tildes. Several implementations do not perform this escaping properly, which is necessary for the `krakend` schema. Another common failure case is related to regular expressions. The JSON Schema specification indicates that regular expressions should be interpreted according to the ECMA-262 specification. Since some languages do not have an ECMA-262-compliant regex engine, some JSON Schema validators choose to use an alternative regex engine. While most regular expressions used in the schemas in our evaluation are supported across a wide variety of regex engines, there are others that fail to be interpreted correctly. We plan to explore the possibility of missing test cases in the JSON Schema specification as future work.

We show a summary of the performance of Blaze compared to other validators in Figure 6 (note the log scale). For this analysis, we exclude schemas where any implementation observed failure and sum the runtimes across the remaining 27 schemas. Blaze is faster than every other implementation by a minimum of 24% on every dataset. Before warmup, Blaze is more than ~10× faster than the next fastest implementation, Boon. After warmup, Blaze is

Dataset	ajv	Blaze	Boon	Corvus	json_schemer	jsonschema (Go)	jsonschema (Py)	JsonSchema.Net	JSV	KMP	NetworkNT
ansible-meta	21.4	**0.5**	26.1	439.1	370.8	66.9	846.1	780.3	82.0	175.0	188.8
	1.4	**0.5**	22.8	14.0	344.1	68.4	921.9	190.3	63.8	26.8	7.9
aws-cdk	1.8	**0.1**	0.5	30.2	41.2	4.2	27.1	178.4	23.7	43.8	17.4
	0.2	**0.1**	0.4	0.5	25.7	3.3	26.0	5.8	6.4	0.8	0.2
babelrc	6.9	**0.3**	0.8	63.8	73.8	11.2	92.4	257.2	16.1	71.6	28.8
	0.3	**0.2**	0.7	1.5	57.4	9.2	89.9	21.3	7.3	5.6	1.0
clang-format	15.5	**0.2**	0.3	338.7	20.2	3.5	27.5	310.7	14.3	42.0	160.2
	1.0	**0.2**	0.3	9.2	17.0	3.3	19.8	24.9	4.6	0.5	3.9
cmake-presets	279.3	**12.2**	260.9	673.7	13893.5	744.9	24338.0	11515.3	8185.7	1034.2	1343.1
	118.2	**8.0**	264.4	97.0	13591.5	751.6	24114.8	9151.3	5969.2	500.4	743.4
code-climate	†	**0.6**	1.9	34.6	132.9	8.9	124.5	373.5	21.4	67.5	29.3
	†	**0.4**	1.2	2.0	118.7	5.6	104.7	51.6	8.7	2.8	1.2
cql2	87.7	**0.6**	794.7	315.3	8607.0	375.3	43168.3	†	†	683.6	3911.7
	15.5	**0.4**	762.4	8.2	7255.4	365.7	40420.3	†	†	349.3	697.0
cspell	17.9	**1.9**	†	258.1	†	†	1049.7	1089.2	109.7	†	314.4
	2.5	**1.6**	†	7.1	†	†	1075.4	225.0	87.6	†	13.2
cypress	12.8	**0.3**	1.1	127.6	74.9	7.5	81.7	395.9	18.2	78.1	28.5
	1.6	**0.3**	0.9	3.2	67.1	8.9	78.7	36.8	8.6	6.8	1.1
deno	12.9	**1.1**	1.8	174.7	151.7	14.7	109.2	412.2	49.6	83.4	350.6
	1.4	**1.0**	1.4	8.0	127.9	14.1	114.4	58.5	27.6	6.2	17.3
dependabot	5.9	**0.8**	3.0	109.8	177.4	29.1	209.5	323.9	28.5	109.8	40.9
	0.8	1.0	2.7	7.5	192.2	24.8	193.0	43.4	17.1	6.2	2.6
draft-04	62.0	**10.7**	33.1	429.8	2167.4	162.3	4201.2	2152.2	†	288.2	256.4
	23.7	**10.7**	31.5	153.8	2083.0	140.6	4022.6	590.4	†	107.5	35.6
fabric-mod	16.6	**2.0**	5.9	144.1	305.9	57.9	740.0	593.5	171.1	143.4	296.1
	2.4	**1.9**	4.9	7.7	296.2	54.9	707.1	108.6	103.3	14.7	12.0
geojson	217.9	**44.3**	1492.5	1127.5	31812.3	2122.7	107120.1	37934.5	28436.3	10501.7	†
	53.2	**27.2**	1500.5	196.1	33601.6	2065.6	94707.2	33199.4	19140.4	9381.7	†
gitpod	10.1	**0.5**	1.7	102.7	152.0	16.1	140.1	346.9	22.8	90.2	132.4
	0.7	**0.4**	1.4	3.9	138.7	16.6	134.2	40.8	11.7	8.6	1.7
helm-chart-lock	8.3	**0.7**	6.3	40.1	428.5	42.5	464.6	460.1	135.6	132.2	45.0
	0.8	**0.6**	5.4	8.1	400.4	34.8	500.0	77.8	95.4	17.4	5.3
importmap	2.9	**0.1**	1.4	28.3	94.0	12.1	84.8	218.2	33.1	59.8	24.3
	0.3	**0.1**	1.2	0.8	63.4	12.4	86.8	14.3	18.8	1.6	1.1
jasmine	3.2	**0.3**	1.2	48.3	90.5	14.4	138.3	279.1	16.8	97.1	25.2
	0.3	**0.2**	1.1	2.0	75.6	13.6	132.9	23.1	8.8	6.6	1.1
jsconfig	17.8	**1.2**	3.7	285.9	188.1	34.5	356.9	5019.7	43.7	160.2	325.0
	3.1	**1.1**	2.9	9.1	183.3	29.3	349.2	4340.1	33.5	19.2	8.5
jshintrc	9.1	**1.6**	2.5	173.9	145.1	24.4	142.7	320.0	37.3	78.3	35.5
	2.6	**1.6**	2.5	19.7	141.1	20.0	149.8	38.8	26.3	5.5	2.1
krakend	†	**1.0**	1.4	646.8	95.6	12.7	123.5	431.0	39.3	†	†
	†	**0.6**	1.0	6.9	70.4	10.6	121.1	34.4	29.1	†	†
lazygit	33.7	**0.5**	0.9	359.9	78.5	6.2	87.7	396.0	19.9	75.8	203.0
	1.9	**0.3**	0.7	5.8	53.6	7.4	85.9	38.1	6.9	4.8	2.1
lerna	3.9	**0.4**	0.9	51.7	67.5	7.8	52.8	247.2	16.4	58.5	21.9
	0.4	**0.3**	0.8	1.4	53.5	8.8	56.0	17.5	6.5	1.9	0.9
nest-cli	7.2	**0.5**	2.0	103.9	126.3	14.8	186.0	440.8	22.3	100.0	37.7
	0.7	**0.4**	1.7	3.3	145.9	15.0	190.0	53.9	13.9	8.2	1.4
omnisharp	10.6	**1.3**	2.0	188.2	107.5	17.8	105.2	325.9	26.5	72.4	29.1
	1.5	**1.2**	2.0	7.0	99.8	13.9	109.0	51.3	15.5	6.4	1.3
openapi	†	**59.3**	94.6	540.2	3331.7	300.0	48583.0	4438.6	1944.4	†	1258.4
	†	**27.6**	92.3	72.1	3152.3	266.8	45347.2	2991.3	1506.9	†	462.0
pre-commit	15.2	**1.0**	4.2	138.1	210.3	33.6	358.8	432.6	114.9	117.8	39.5
	4.9	**0.9**	4.0	24.9	184.7	34.6	351.2	84.9	69.9	11.2	4.9
pulumi	14.8	**1.5**	4.7	99.3	319.8	31.4	396.0	850.7	111.3	144.9	70.6
	2.2	**1.3**	4.7	6.6	323.7	29.9	411.5	152.0	66.1	15.6	5.4
semantic-release	4.0	**0.3**	1.8	51.7	117.2	22.5	203.6	353.8	18.8	79.5	45.5
	0.4	**0.2**	1.6	1.2	103.1	17.2	211.1	45.5	9.9	5.6	1.5
stale	4.2	**0.3**	1.0	53.9	89.3	13.6	96.6	256.3	†	72.1	33.6
	0.3	**0.3**	0.9	1.7	81.4	14.2	98.1	17.8	†	2.2	1.0
stylecop	6.9	**1.1**	1.9	131.3	152.0	16.3	151.2	325.9	55.9	95.3	274.1
	1.1	**0.9**	1.5	4.5	126.8	15.5	139.6	43.4	33.9	6.8	9.9
tmuxinator	4.2	**0.3**	0.9	47.9	63.6	9.4	105.2	248.9	26.7	60.6	33.6
	0.5	**0.2**	0.8	1.3	45.3	7.2	111.8	16.9	7.8	2.5	0.8
ui5	111.6	**1.5**	6.3	719.6	323.8	44.4	443.0	3102.4	46.5	144.2	251.8
	4.7	**1.1**	5.9	18.3	298.0	39.4	453.3	1479.9	29.2	14.8	8.6
ui5-manifest	†	**19.1**	†	831.0	1172.3	†	2154.1	2856.8	702.5	315.4	†
	†	**7.3**	†	69.3	1267.5	†	2191.0	1144.9	481.7	51.2	†
unreal	21.3	**1.1**	2.7	164.4	195.4	27.8	415.4	510.8	83.5	157.4	266.3
	6.4	**1.0**	2.8	23.2	167.3	24.8	387.0	92.9	36.5	8.2	5.2
vercel	28.3	**0.6**	2.5	249.8	144.8	18.0	144.3	505.7	65.0	117.7	185.7
	1.5	**0.6**	1.6	6.6	129.0	13.2	138.8	70.2	25.8	6.6	2.7
yamllint	†	**0.1**	0.4	13.3	21.1	3.5	24.7	355.2	†	38.6	12.8
	†	**0.0**	0.3	0.2	20.7	2.0	22.6	55.8	†	0.5	0.2

The top number is the runtime in milliseconds for the first run after compilation. The bottom is the runtime for validating the same documents after several validation runs.

†The implementation produced an error on this dataset.

Table 5: Runtime results for implementations across datsets

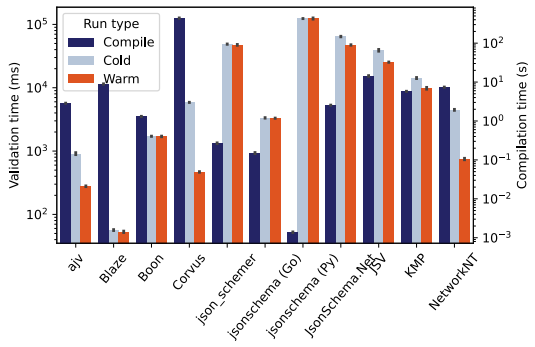

Figure 6: Compilation and validation time summary

approximately ~9× faster than the second fastest implementation, ajv while also being more correct than ajv.

Note that we include any initialization required before validation when measuring compile time. Implementations that do not use an explicit compilation step still measure a small compilation time. While the compilation time of Blaze is slower than many other validators, it still requires less than 10s on average. As noted earlier, we observed that schemas are infrequently changed (on the order of weeks). Since the compiled schema only needs to be updated when the original schema changes, the overhead is minimal.

6.2.3 Optimization. In addition to the benchmarks against other validators, we also performed an ablation study to understand the effect of each of our optimizations on the performance of Blaze. Specifically, we measure the performance with four separate optimizations disabled: our semi-perfect hash function (Section 4.1), instruction unrolling (Section 4.2), regular expression optimization (Section 4.3), and instruction reordering (Section 4.4). We note that while reducing memory allocations was important for the performance of Blaze, this is not an optimization that can easily be disabled since it is integral to our design. The relative speedup achieved by each optimization is shown for the single schema that is most affected by each optimization as well as the overall runtime for all 38 schemas in our test collection.

Although the average speedup for unrolling is relatively small (only ~3% overall), we see a benefit of nearly 47% on one schema, suggesting it can be very effective in certain cases. This specific example is a small schema with only a few properties defined, all of which are required. In this case, avoiding the loop significantly reduces validation time. We do also note that in the worst case, as with unrolling in compiler optimization, this optimization can reduce performance. Although Blaze employs heuristics to decide when unrolling is appropriate, we still see a significant performance reduction on several schemas, suggesting a need for improvement in these heuristics in future work. We also plan to make these optimizations configurable so users can decide which optimizations to use in the case our heuristics are ineffective.

The overall benefit from regex optimizations is minimal because many schemas do not make use of regular expressions. In the best case, we see a runtime reduction of over 41%. One common case is schemas using `"patternProperties": ".+"` to define a schema for non-empty keys. This is reduced to a string length check, avoiding the use of a regex engine entirely. We plan to explore further ways to optimize the use of regular expressions in future work.

We compare the performance of our hash function with the popular and widely-used MurmurHash3[7]. Our hash function along with instruction reordering are by far the most effective optimizations employed by Blaze, with an average improvement of almost 25% across all schemas. In the best case, we see almost a 49% percent reduction in runtime through the use of our hash function. This occurs in the `helm-chart-lock` schema which is quite small and uses few features of JSON Schema, meaning the validation time is dominated by hash table lookups. There is only a single case where use of our hash function results in a reduction in runtime. We plan to explore this case further in future work.

Our reordering optimization achieves a 95% reduction in runtime for one schema (`cql2`). This schema makes significant use of `oneOf`, which requires that a document be valid against only one of a set of schemas. These variants are identified by a string key `op` that has specific values for each variant. Reordering ensures that validating the `op` key occurs first. This means that the remainder of the schema only needs to be checked in the case where the correct `op` matches. Using a string key to identify a schema variant is a common use case we believe we can optimize further in the future.

We also measure the peak memory usage of each process during compilation and validation with the results shown in Figure 8. Boon exhibits the lowest memory usage, while Blaze uses approximately 43% more memory. This is approximately the same memory used by the Go jsonschema validator, but both use approximately half the memory of the next implementation and 85% less memory than Corvus, the heaviest implementation. We leave further analysis as well as any possible optimizations as future work.

7 RELATED WORK

We are not aware of any in-depth academic research into JSON Schema validation beyond the work of Attouche et al. [4]. However, there are many existing open-source validators that make use of precompilation as we have listed in Table 4. ajv[8] is a popular validator that is able to generate JavaScript code through a precompilation process to use for later validation. While ajv is sometimes comparable to Blaze in speed, it suffers from significant correctness issues and fails over 200 test cases in the official JSON Schema test suite. We believe that the speed of ajv comes primarily from the significant optimizations present in modern JavaScript runtimes. Indeed, we noticed when switching from the Node.js runtime to the Bun runtime used in our evaluation, the validation performance improved significantly. This observation matches what has been observed in prior work [1, 15]. Unlike ajv, Blaze maintains correctness in the validation process while also being faster to validate.

Corvus[9], jsonschema (Go)[10], and JsonSchema.Net [11] also perform precompilation. Corvus generates .NET code that compiled to produce the final validator while jsonschema constructs an internal data structure intended to optimize validation. While these

[7] https://github.com/aappleby/smhasher/wiki/MurmurHash3
[8] https://ajv.js.org
[9] https://github.com/corvus-dotnet/Corvus.JsonSchema
[10] https://github.com/santhosh-tekuri/jsonschema
[11] https://docs.json-everything.net/schema/basics/

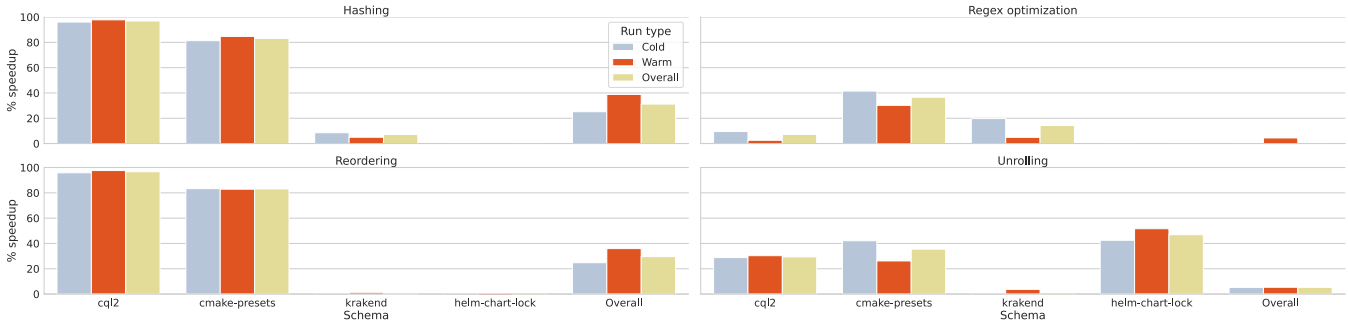

Figure 7: Performance benefits of various optimizations

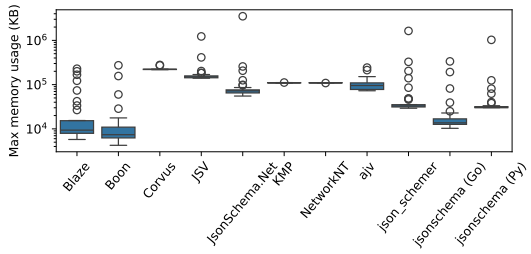

Figure 8: Memory usage during compilation and validation

implementations are among the fastest for warm runs, they do not incorporate many optimizations used in Blaze and remain significantly slower to validate on average. JsonSchema.Net performs static analysis of JSON Schema [9, 10] in order to reduce the amount of work necessary at validation time by identifying constraints that must apply to particular locations in a JSON document. Blaze extends this idea to significantly more in-depth analysis involving interactions between keywords and the order of instructions.

T there has been some past work at examining the structure of JSON Schemas in practice [5, 17, 21]. This work focused on specific research questions that were not applicable to our analysis. We used the Sourcegraph public code search API[12] to find files with the extension `.json` and containing a key `$schema` key indicating the document is a JSON Schema document. We downloaded all these schemas and validated them against their corresponding meta-schema in order to ensure each schema is valid. As mentioned previously, we collected approximately 31,000 schemas in total. This allowed us to answer such questions as "What is the distribution of key lengths defined in JSON Schemas?". We believe this corpus of schemas will be useful for further analysis.

Since XML is also a semi-structured data format, it is natural to compare corresponding validators. Indeed, others have shown that careful XML validator design improves performance [16]. However, JSON Schema validation is much more complex than XML Schema Definition (XSD) or Document Type Definition (DTD), two common formats for XML schemas. XML schema validation can occur in log-linear time [6], while JSON Schema validation is PSPACE-complete [4] in general. Due to the difference in expressiveness of the corresponding schema languages, JSON Schema validation is more complex and necessitates a different set of optimizations.

[12]https://sourcegraph.com/search

8 FUTURE WORK

Currently, the compilation process produces instructions that can be interpreted at runtime much more efficiently than operating using the original JSON Schema. However, we plan to explore precompiling the code necessary to validate each schema ahead of time. In addition to eliminating overhead from interpretation at runtime, this has the potential to leverage existing compiler optimizations to further improve performance. We also plan to explore further static optimizations to the generated schemas.

We believe there is potential to optimize schema compilation in a data-dependent way. Many validation instructions can be reordered while preserving correctness as we showed in Section 4.4. The fastest approach to validation will detect failure as early as possible. Depending on the specific schema and data being processed, different use cases might result in a higher likelihood of certain assertions failing as compared to others. If profiling suggests that a particular property is likely to fail validation, we can order instructions to validate that property first. This enables early failure detection, minimizing the number of executed instructions. We also plan to explore the use of data-dependent optimizations.

Finally, in this work we have focused only on indicating whether a document is valid according to a schema. In the case of an invalid document, it can be helpful to provide information on exactly why the document is not accepted according to the schema. This is particularly important in our case since we want to reference the user-provided schema ignoring any optimizations to instructions generated during the compilation process. While Blaze does have the option to provide helpful error messages to users for debugging purposes, here we focus purely on performance.

9 CONCLUSION

We have introduced Blaze, a JSON Schema validator that makes use of precompilation to optimize the validation process. Unlike many existing validators, Blaze achieves 100% correct validation behavior according to the JSON Schema specification. Blaze also validates documents a minimum of 20% faster than all other validators we tested on a wide variety of datasets and an average of 10× faster than the next fastest validator. We believe that there are many opportunities for further optimization.

REFERENCES

[1] Md Feroj Ahmod. 2023. *Javascript runtime performance analysis: Node and Bun*. Master's thesis. Tampere University, Faculty of Information Technology and Communication Sciences. https://trepo.tuni.fi/bitstream/handle/10024/149672/AhmodMdFeroj.pdf

[2] Henry Andrews. 2022. JSON Schema is a constraint system. Retrieved 2025-11-04 from https://modern-json-schema.com/json-schema-is-a-constraint-system

[3] Ioannis Arapakis, Xiao Bai, and Berkant Barla Cambazoglu. 2014. Impact of response latency on user behavior in web search. In *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, Gold Coast, Australia, 103–112.

[4] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2024. Validation of Modern JSON Schema: Formalization and Complexity. *Proc. ACM Program. Lang.* 8, POPL, Article 49 (Jan. 2024), 31 pages.

[5] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. An empirical study on the "usage of not" in real-world json schema documents. In *Conceptual Modeling: 40th International Conference*. Springer, 102–112.

[6] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. 2004. Incremental validation of XML documents. *ACM Trans. Database Syst.* 29, 4 (Dec. 2004), 710–751.

[7] Julian Berman, Karen Etheridge, Greg Dennis, Evgeny Poberezkin, Chase Sterling, Shawn Silverman, Henry Andrews, Santhosh Kumar Tekuri, Nikita Skovoroda, Jason Desrosiers, Johnny Graettinger, Zac Hatfield-Dodds, Ben Hutton, Francis Galiegue, Mark Jacobson, Kyle Fuller, Max Ignatenko, Ryan Miller Galamb, Austin Wright, Geraint, amosonn, Ethan, Hillel Arnold, Iain Beeston, Irene Knapp, marcus-kruse gcx, leadpony, Alvaro Gutierrez Perez, Andreas Gebhardt, and Dmitry Dygalo. 2023. *json-schema-org/JSON-Schema-Test-Suite: 23.1.0*. https://doi.org/10.5281/zenodo.7927399

[8] Julian Berman, Adwait Godbole, Agnivesh Chaubey, Dhruv Singh, harrel56, Sudip Mandal, Noman Dhoni, Matthew Adams, Oleg Smirnov, Santhosh Kumar Tekuri, Vishrut Aggarwal, Ashmit Jagtap, Anish Kshirsagar, Martin Hauner, Greg Dennis, Jason Desrosiers, Siddharth Singh, Juan Cruz Viotti, Zeel Rajodiya, Ludovic Dem, Daniel Parker, Akshay Bagai, Stefan Klessinger, sajal j25, Akanksha Kushwaha, XD, and SimonDMC. 2025. *bowtie-json-schema/bowtie: v2025.2.5*. https://doi.org/10.5281/zenodo.14834943

[9] Greg Dennis. 2023. The New JsonSchema.Net. Retrieved 2025-11-04 from https://blog.json-everything.net/posts/new-json-schema-net/

[10] Greg Dennis. 2023. Static Analysis of JSON Schema. Retrieved 2025-11-04 from https://json-schema.org/blog/posts/schema-static-analysis#keywords-that-have-dependencies

[11] Edward A Fox, Lenwood S Heath, Qi Fan Chen, and Amjad M Daoud. 1992. Practical minimal perfect hash functions for large databases. *Commun. ACM* 35, 1 (1992), 105–121.

[12] Alan D George. 1990. An overview of RISC vs. CISC. In *Proceedings The Twenty-Second Southeastern Symposium on System Theory*. IEEE Computer Society, 436–437.

[13] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 27 (Nov. 2023), 41 pages.

[14] Stefan Gössner, Glyn Normington, and Carsten Bormann. 2024. JSONPath: Query Expressions for JSON. RFC 9535. https://doi.org/10.17487/RFC9535

[15] I Kniazev and A Fitiskin. 2023. Choosing the right Javascript runtime: an in-depth comparison of Node.js and Bun. *Economic Sciences* 108 (2023), 72.

[16] Margaret G. Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, and Martha Mercaldi. 2006. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In *Proceedings of the 15th International Conference on World Wide Web* (Edinburgh, Scotland) *(WWW '06)*. Association for Computing Machinery, New York, NY, USA, 93–102.

[17] Benjamin Maiwald, Benjamin Riedle, and Stefanie Scherzinger. 2019. What Are Real JSON Schemas Like?. In *Advances in Conceptual Modeling*, Giancarlo Guizzardi, Frederik Gailly, and Rita Suzana Pitangueira Maciel (Eds.). Springer International Publishing, Cham, 95–105.

[18] Fiona Fui-Hoon Nah. 2004. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology* 23, 3 (2004), 153–163.

[19] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. 2021. An Analysis of Public REST Web Service APIs. *IEEE Transactions on Services Computing* 14, 4 (2021), 957–970.

[20] Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. 2023. RestGPT: Connecting Large Language Models with Real-World RESTful APIs. arXiv:2306.06624 [cs.CL]

[21] Claire Yannou-Medrala and Fabien Coelho. 2023. An Analysis of Defects in Public JSON Schemas. In *39èmes journées de la conférence BDA Gestion de Données – Principes, Technologies et Applications*. HAL, Montpellier, France, 13.