Filter Before You Parse: Faster Analytics on Raw Data

Shoumik Palkar, Firas Abuzaid, Peter Bailis, Matei Zaharia[†]

Stanford InfoLab, [†]Databricks Inc.

ABSTRACT

Exploratory big data applications often run on unstructured or semi-structured raw data formats, such as JSON files or text logs. These applications can spend 80–90% of their execution time parsing the data. In this paper, we propose a new approach for reducing this overhead: apply filters on the data's raw bytestream *before* parsing. This technique, which we call raw filtering, leverages the features of modern hardware and the high selectivity of queries found in many exploratory applications. With raw filtering, a user-specified query predicate is compiled into a set of filtering primitives called raw filters (RFs). RFs are fast, SIMD-based operators that occasionally yield false positives, but never any false negatives. We combine multiple RFs into an RF cascade to decrease the false positive rate and maximize parsing throughput. Because the best RF cascade is data-dependent, we propose an optimizer that dynamically selects the combination of RFs with the best expected throughput, achieving within 10% of the global optimum cascade while adding less than 1.2% overhead. We implement these techniques in a system called Sparser, which automatically manages a parsing cascade given a data stream in a supported format (e.g., JSON, Avro, Parquet, or log files) and a user query. We show that many real-world applications are highly selective and benefit from Sparser. Across diverse workloads, Sparser accelerates state-of-the-art parsers such as Mison and RapidJSON by 2–22× and improves end-to-end application performance by up to 9×.

1 Introduction

Many analytics workloads process data stored in unstructured or semi-structured formats, including JSON, XML, or binary formats such as Avro and Parquet [6, 48]. Rather than loading datasets in these formats into a DBMS, researchers have proposed techniques [3, 35–38, 46] for executing queries in situ over the raw data directly.

A key bottleneck in querying raw data is parsing the data itself. Parsers—especially for human-readable formats such as JSON—are typically expensive because they rely on state-machine-based algorithms that execute a series of instructions per byte of input [17, 53]. In contrast, modern CPUs are optimized for operations on multiple bytes in parallel (e.g., SIMD instructions). In response, researchers have recently developed new parsing methods that utilize modern hardware more effectively [38, 43]. One such example is the Mison JSON parser [38], which uses SIMD instructions to find special characters such as brackets and colons to build a *structural index* over a raw JSON string, enabling efficient field projection without deserializing the record completely. This approach delivers substantial speedups: we found that Mison can parse highly nested in-memory data at over 2GB/s per core, over 5× faster than RapidJSON [53], the fastest traditional state-machine-based parser available [33]. Even with these new techniques, however, we still observe a large memory-compute performance gap: a single core can scan a raw bytestream of JSON data 10× faster than Mison parses it. Perhaps surprisingly, similar gaps can even occur when parsing binary formats that require byte-level processing, such as Avro and Parquet.

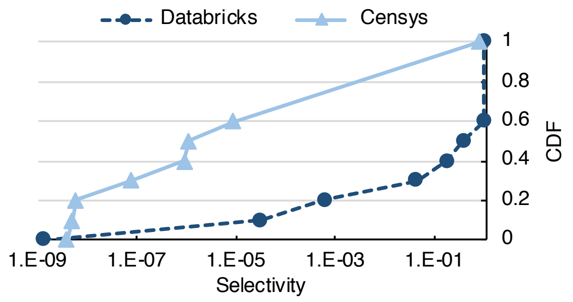

Figure 1: CDF of selectivities from (1) customer queries over JSON and CSV files running Spark SQL through Databricks' cloud service, and (2) researchers' queries over JSON data on the Censys search engine [25]. Both sets of queries are highly selective.

In this work, we accelerate raw data processing by exploiting a key property of many exploratory analytics workloads: these workloads often exhibit high selectivity. Figure 1 illustrates this, showing query selectivity from two cloud services: customer Spark SQL queries on JSON and CSV data profiled on Databricks' cloud service, and researchers' queries over Censys [25], a public search engine of Internet port scan data broadly used in the security community. 40% of the Spark queries select less than 20% of records, while the median Censys query selects only 0.001% of the records for further processing.

We propose *raw filtering*, a new approach that leverages modern hardware and the high selectivities of today's workloads to filter data *before* parsing it. Raw filtering uses a set of filtering primitives called *raw filters* (RFs), which are operators derived from a query predicate that filter records by inspecting a raw bytestream of data, such as UTF-8 strings for JSON or encoded binary buffers for Avro or Parquet. Rather than parsing records and evaluating query predicates exactly, RFs filter records by evaluating a format-agnostic filtering function over raw bytes with some false positives, but no false negatives.

To decrease the false positive rate, we can use an optimizer to compose multiple RFs into an *RF cascade* that incrementally filters the data. A full format-specific parser (e.g., Mison) then parses and verifies any remaining records. Raw filtering is thus complementary to existing work on fast projection [15, 23, 24, 28, 38]. With a well-optimized RF cascade, we show that raw filtering can accelerate even state-of-the-art parsers by up to 22× on selective queries.

Because RFs can produce false positives and still require running a full parser on the records that pass them, two key challenges arise in utilizing raw filtering efficiently. First, the RFs themselves have to be highly efficient, allowing us to run them without impacting overall parsing time. Second, the RF cascade optimizer must quickly find efficient cascades, which is challenging because the space of possible cascades is combinatorial, the passthrough rates of different

RFs are not independent, and the optimizer itself must not add high overhead. We discuss how we tackle these two challenges in turn.

Challenge 1: Designing Efficient Raw Filters. The first challenge is ensuring that RFs are hardware-efficient. Since RFs produce false positives, an inefficient design for these operators could increase total query execution time by adding the overhead of applying RFs without discarding many records. To address this challenge, we propose a set of SIMD-enabled RFs that process multiple bytes of input data per instruction. For example, the *substring search* RF searches for a byte sequence in raw data that indicates whether a record could pass a predicate. Consider evaluating the predicate `name = "Albert Einstein"` over JSON data. The substring RF could search over the raw data for the substring `"Albe"`, which fits in an AVX2 SIMD vector lane and allows searching 32 bytes in parallel. This is a valid RF that only produces false positives, because the string `"Albe"` must appear in any JSON record that satisfies the predicate. Without fully parsing the record, however, the RF may find cases where the substring comes from a different string (e.g., `"Albert Camus"`) or from the wrong field (e.g., `friend = "Albert Einstein"`). Likewise, a *key-value search* RF extends substring search to look for key-value pairs in the raw data (e.g., a JSON key and its value). While designing for hardware efficiency imposes some limitations on the predicates we can convert to RFs, our RFs can be applied to many queries in diverse workloads, and can stream through data 100× faster than existing parsers.

Challenge 2: Choosing an RF Cascade. The second challenge is selecting the RF cascade that gives us the highest expected parsing throughput: we need to determine which RFs to include, how many to include, and what order to apply them in. Naturally, the performance of each RF cascade depends on its execution cost, its false positive rate, and the execution cost of running the full parser. But determining these metrics is difficult, for two reasons: (1) the metrics are data-dependent, and (2) the passthrough rates of individual RFs in the cascade can be highly correlated with one another. For example, a cascade with the single substring RF `"Albe"` might not benefit from the additional substring RF `"Eins"`: whatever records match `"Albe"` are already likely to match `"Eins"` as well. In our evaluation, we show that modeling this interdependency between RFs is critical for performance, and can make a 2.5× difference compared to classical methods for predicate ordering that assume independence [9].

To address this challenge, we propose an efficient optimizer that uses SIMD to efficiently select a cascade while also accounting for RF interdependence. Our optimizer periodically takes a sample of the data stream and estimates the individual passthrough rates and execution costs of the valid RFs for the query on it. It stores the result of each RF on the sample records in a bitmap that allows us to rapidly compute the passthrough rate for any cascade of RFs using SIMD bitwise operators. With this approach, the optimizer can efficiently search through a large space of cascades and pick the one with the best expected throughput. We show that choosing the right cascade can make a 10× difference in performance, and that our optimizer only adds 1.2% overhead. We also show that updating the RF cascade periodically while processing the input data can make a 25× difference in execution time due to changing data properties.

Summary of Results. We implement raw filtering in a system called Sparser, which implements the SIMD RFs and optimizer described above. Sparser takes a user query predicate and a raw bytestream as input and returns a filtered bytestream to a downstream query engine. Our evaluation shows that Sparser outperforms standalone state-of-the-art parsers and accelerates real workloads when integrated in existing query processing engines such as Spark SQL [5]. On exploratory analytics queries over Twitter data from [38, 58], Sparser improves Mison's JSON parsing throughput up to 22×. When integrated into Spark SQL, Sparser accelerates distributed queries on a cluster by up to 9× end-to-end, including the time to load data from disk. Perhaps surprisingly, Sparser can even accelerate queries over binary formats such as Avro and Parquet by 1.3–5×. Finally, we show that raw filtering accelerates analytics workloads in other domains as well, such as filtering binary network packet captures and analyzing text logs from the Bro [11] intrusion detection system.

To summarize, our contributions are:

1. We introduce raw filtering, an approach that leverages the high query selectivity of many exploratory workloads to filter data before parsing it for improved performance. We also present a set of SIMD-based RFs optimized for modern hardware.
2. We present a fast optimizer that selects an efficient RF cascade in a data- and query-dependent manner while accounting for the potential interdependence between RFs.
3. We evaluate raw filtering in Sparser, and show that it complements existing parsers and accelerates realistic workloads by up to 9×.

2 Problem Statement and Goals

Raw filtering is motivated by the rising volumes of unstructured and semi-structured data, such as text logs, JSON, Avro [6] and Parquet [48]. With organizations embracing a "store everything" philosophy [34, 54] and an ever-increasing volume of machine generated data ranging from server diagnostic logs to network telemetry [10, 32, 52], accessing the records of relevance for a query is often expensive. Faster I/O devices such as SSDs, 100Gbps Ethernet, and RDMA have also shifted bottlenecks to the CPU [47]. As a result of these shifts, in cases where queries extract a small subset of the data, systems waste cycles parsing records that will eventually be discarded. This scenario—in which only a fraction of the data needs to be parsed—is common for many real-world exploratory workloads, as shown in our Databricks and Censys traces (Figure 1).

The overall goal of raw filtering is thus to maximize the throughput of parsing and filtering raw, serialized data, by applying query predicates before parsing data. A serialized record could be newline-separated UTF-8 JSON objects or a single line from a text log, for example. Raw filtering primarily accelerates exploratory workloads over unstructured and semi-structured textual formats, but we show in § 7 that it is also applicable to queries over binary formats such as Avro and Parquet. Raw filtering is most impactful for selective queries.

In settings where the same data is accessed repeatedly, users can load the data into a DBMS, build an index over it [3, 37]), or convert it into an efficient file format such as Parquet. Raw filtering thus only targets workloads where the data is not yet indexed, perhaps because it is accessed too infrequently to justify continuously building an index (e.g., for high-volume machine telemetry), or because the workloads themselves are performing data ingest. For example, at Databricks, customers commonly use Spark to convert ingested JSON and CSV data to Parquet, as shown by the large fraction of JSON/CSV jobs with selectivity 1 in Figure 1. Despite this, there are still a large number of selective queries on CSV and JSON directly (60% of the queries in Figure 1). In Censys, a purely exploratory workload, most queries are highly selective.

3 Overview

This section gives an overview of raw filtering and introduces Sparser, a system that addresses its challenges.

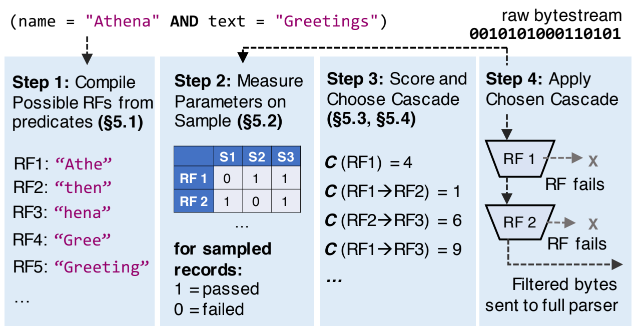
(name = "Athena" AND text = "Greetings")
raw bytestream
0010101000110101
Step 1: Compile Possible RFs from predicates (§5.1)
RF1: "Athe"
RF2: "then"
RF3: "hena"
RF4: "Gree"
RF5: "Greeting"
...
Step 2: Measure Parameters on Sample (§5.2)
S1
S2
S3
RF 1
0
1
1
RF 2
1
0
1
...
for sampled records:
1 = passed
0 = failed
Step 3: Score and Choose Cascade (§5.3, §5.4)
C (RF1) = 4
C (RF1→RF2) = 1
C (RF2→RF3) = 6
C (RF1→RF3) = 9
...
Step 4: Apply Chosen Cascade
RF 1
RF fails
RF 2
RF fails
Filtered bytes sent to full parser

Figure 2: Sparser's architecture. Sparser builds a cascade of raw filters to filter data before parsing it.

3.1 Raw Filtering

Raw filtering uses a primitive called a *raw filter* (RF) to discard data. Formally, an RF has the following two properties:

An RF operates over a raw bytestream. Raw filters do not require fully parsing records, and instead operate on an opaque sequence of bytes. For example, the bytestream could be UTF-8 strings encoded in JSON or CSV, or packed binary data encoded in Avro or Parquet.

An RF may produce false positives, but no false negatives. A raw filter searches for a sequence of bytes that originates from a predicate, but it offers no guarantee that finding a match will produce the same result as the original predicate. Therefore, for supported predicate types (which we discuss in § 3.3), raw filters can produce false positives, but no false negatives. Composing multiple RFs into an *RF cascade* [64] can further reduce—but not eliminate—false positives. Thus, an RF (or RF cascade) must be paired with a full, format-specific parser (e.g., RapidJSON or Mison) that can parse the records from the filtered bytestream and apply the query predicates.

We implement raw filtering in a system called Sparser, which addresses two main challenges with using RFs: designing an efficient set of filters that can leverage modern hardware, and selecting a composition of RFs—an RF cascade—that maximizes throughput.

3.2 System Architecture

Figure 2 summarizes Sparser's overall architecture. First, Sparser decomposes the input query predicate into a set of RFs. RFs in Sparser filter data by searching for byte sequences in the bytestream using SIMD instructions. Each RF has a false positive rate for passing records (§3.1) as well as an execution cost. The RFs' false positive rates and execution times are not known *a priori* since both are data-dependent. Sparser thus regularly estimates these quantities for the individual RFs by evaluating the individual RFs on a periodic sample of records from the input. These data-dependent factors guide Sparser in choosing which RFs to apply on the full bytestream.

Sparser chooses an RF cascade to execute over the full bytestream using an optimizer to search through the possible combinations of RFs and balance the runtime overhead of applying the RFs with their combined false positive rate. The main challenge in the optimizer is to accurately and efficiently model the joint false positive rates of the RFs, since the individual false positive rates are not independent and the system only measures individual RF rates. Directly measuring the execution overheads and passthrough rates of a combinatorial number of cascades would be prohibitively expensive; instead, the optimizer uses a SIMD-accelerated bitmap data structure to obtain joint probabilities using only the estimates of the *individual* RFs. The optimizer then uses a cost function to compute the expected parsing throughput of the cascade, using the execution costs of the

Filter	Example
Exact String Match	`WHERE user.name = 'Athena'`, `WHERE retweet_count = 5000`
Contains String	`WHERE text LIKE '\%VLDB\%'`
Key-Value Match	`WHERE user.verified = true`
Contains Key	`WHERE user.url != NULL`
Conjunctions	`WHERE user.name = 'Athena' AND user.verified = true`
Disjunctions	`WHERE user.name = 'Athena' OR user.verified = true`

Table 1: Filters supported in Sparser. Only equality-based filters are supported—numerical range filters (e.g., `WHERE num_retweets > 10`) are not supported.

individual RFs, the false positive rates of the RFs, and the execution cost of the full parser. The chosen RF cascade incrementally filters the bytestream using SIMD-based implementations of the RFs. The downstream query engine parses, filters, and processes records that pass the cascade.

3.3 Supported Predicates

Sparser supports several types of predicate expressions, summarized in Table 1. The system supports exact equality matches, substring matches, key-value matches, and key-presence matches. Predicates that check for the presence of a key are only valid for data formats such as JSON, where keys are explicitly present in the record. These supported predicates are also valid over binary data formats (e.g., Avro and Parquet).

Key names in the predicate can be nested (e.g., `user.name`): nested keys are a shorthand for checking whether each non-leaf key exists (a non-leaf key is any key that has a nested object as a value), and whether the value at the leaf matches the provided filter. For example, in the predicate `user.name = "Athena"`, Sparser checks for the existence of the field `"user"` and `"name"` (where `"name"` is a sub-field of `"user"`), and then checks if the value of the key `"name"` matches `"Athena"`.

Applications may also provide a value without an associated key. This is useful for binary formats, where key names often do not appear explicitly. Finally, users can specify arbitrary conjunctions (`AND` queries) and disjunctions (`OR` queries) of individual predicates.

3.4 System Limitations

Sparser has a number of limitations and non-goals. First, Sparser's RFs do not support every type of predicate expression that can be found in SQL. In particular, Sparser does not support range-based predicates over numerical values (e.g., `retweet_count > 15`) and inequality predicates for string values (e.g.,`name != "Athena"`). These queries are not supported because Sparser cannot implement them as RFs over a raw byte sequence efficiently. Nevertheless, we have found that Sparser's RFs still have broad applicability in many query workloads. Second, because the search space of cascades is combinatorial, Sparser bounds the maximum depth of the cascade to at most 4 RFs. We show in §7 that, despite the bounded search space, Sparser's SIMD-accelerated optimizer still produces cascades that accelerate parsing by over 20× in real workloads. Additionally, we show that, on these workloads, Sparser's parsing throughput is only up to 1.2% slower than searching the unbounded set of possible cascades. Finally, Sparser's speedups depend on high query selectivity. Sparser exhibits diminishing speedups on queries with low selectivity, but critically imposes no overhead on these queries (§7).

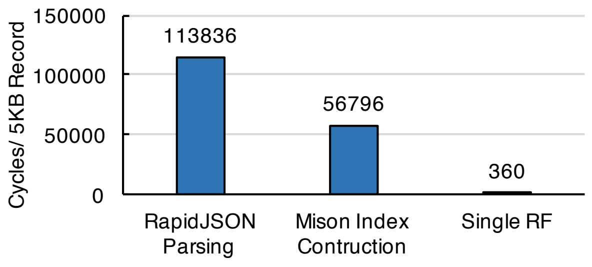

Figure 3: Comparison of CPU cycles spent parsing a single 5KB JSON record in L1 cache, using state-of-the-art parsers vs. applying a single RF that searches for a one-byte value with SIMD in the same buffer. The difference in performance is over 100×.

4 Sparser's Raw Filters

Recall that Sparser's overall objective is to discard records that fail the query predicate as fast as possible without parsing. A key challenge in Sparser is thus designing a set of efficient filtering operators that discards records by inspecting only a raw bytestream: an operator over a bytestream necessarily lacks access to format-specific structural information built via parsing (e.g., the offsets of specific fields [38]).

To address this challenge, Sparser utilizes a set of SIMD-accelerated raw filters (RFs) to discard data by searching for format-agnostic byte sequences in the input. The RFs use this design to take advantage of the throughput gap between parsing a record to extract structural information and scanning it to search for a byte sequence that fits in a single SIMD vector lane: Figure 3 illustrates this by comparing cycles spent parsing a 5KB record by two state-of-the-art JSON parsers and a SIMD-based search for a one-byte value over the same buffer.

An RF returns a boolean signal specifying whether it passed or failed a record depending on whether it found its byte sequence. Sparser produces several possible RFs for a given query and chooses a cascade using these RFs (§5).Sparser contains two RF primitives: *substring search* and *key-value search*.

4.1 Substring Search

The substring search RF is the main filtering primitive in Sparser. This RF searches for a byte sequence (i.e., a binary "substring") that indicates a record could pass a query predicate. Consider the predicate in Listing 1, which selects records where `text` and `name` match the specified value. Assuming the underlying bytestream encodes text data as UTF-8, the substring search RF can look for several number of bytes sequence that indicate a record could pass this filter: a few examples are `"VLDB"`, `"submissi"`, `"subm"`, and `" Athe"`. If the RF does not find any of these strings, it can discard the record with no false negatives because the predicate cannot be satisfied.

```
name = "Athena" AND text = "My submission to VLDB"
```

Listing 1: A sample query predicate that Sparser takes as input.

The substring search RF leverages SIMD instructions in the CPU by packing multiple copies of the byte sequence into a single vector register. This allows the RF to search over multiple bytes in parallel with a small sliding window. To illustrate this, consider the predicate in Listing 1 and the input string in Figure 4. The figure shows an example of an RF with the 4-byte UTF-8 sequence `"VLDB"`. This string is repeated eight times in a 32-byte vector register. To find all occurrences of the sequence, the operator needs to look at a 4-byte-wide sliding window. In the example, the fourth shift finds an occurrence of the sequence in the input, meaning that the RF passes the record containing the input string. Sparser considers 2-, 4- and 8-byte substrings with this RF.

```
Input   : “text”:“My VLDB 2018 submission”
Shift 1 : VLDBVLDBVLDBVLDBVLDBVLDBVLDBVLDB-------
Shift 2 : -VLDBVLDBVLDBVLDBVLDBVLDBVLDBVLDB------
Shift 3 : --VLDBVLDBVLDBVLDBVLDBVLDBVLDBVLDB-----
Shift 4 : ---VLDBVLDBVLDBVLDBVLDBVLDBVLDBVLDB----
```

Figure 4: An example of a substring search. The length of the substring denotes the number of shifts required. In this example, the fourth shift produces a match since one of the repeated 4-byte sequences in the vector register matches the input text.

```
Key: name Value: Athena Delimiter: ,
Ex 1: “name”: “Athena”,                (PASS)
Ex 2: “name”: “Actually, Athena”       (FAIL)
Ex 3: “text”: “My name is Athena”      (PASS, False positive)
```

Figure 5: Example of using the key-value search RF for the predicate `name = "Athena"`. The key is `name`, the value is `Athena`, and the delimiter is `‘,’`. The underlined regions are the parts of the input where the filter looks for the value after finding the key.

Note that the value of the `text` field in Figure 4 does not match the predicate on the `text` field in Listing 1, indicating that the RF in the example passed as a false positive. The RF does not return false negatives since it finds all occurrences of the sequence. The main advantage of the substring search RF is that, for a well-chosen sequence, the operator can reject inputs at nearly the speed of streaming data through a CPU core.

4.2 Key-Value Search

The key-value search RF searches for all co-occurrences of a key and a corresponding value within a record in the input data; this operator is useful for finding associations between two raw byte sequences. For example, if we search for `name = "Athena"`, we want to increase the probability that the `"Athena"` we find is associated with the field `"name"`. This operator is especially useful in cases where the key and value appear frequently independently, but rarely together. One example is searching for the value **`true`** in a JSON record, which likely appears somewhere in the record but not necessarily with the key specified by the query predicate.

This RF takes three parameters: a key, a value, and a delimiter set. Keys and values are byte sequences, as in the substring search operator. The delimiter set is a set of one-byte characters that signals the stopping point for a search; the value search term must appear after the key and before any of the delimiters. Thus, after finding an occurrence of the key, the operator searches for the value and terminates its search at the first occurring delimiter character found after the key. Sparser can search for the key, value, and stopping point using the packed-vector technique from the substring search RF for hardware efficiency, by either checking for a 2-, 4-, or 8-byte substring of both the key and value.

Unlike the substring search RF (which searches for arbitrary sequences and can be used for both **`LIKE`** and = predicates), the key-value search operator is only applicable for equality predicates; **`LIKE`** predicates are not supported. To understand why this constraint is necessary, Figure 5 provides an example of a key-value search on a JSON record that could yield false negatives if a **`LIKE`** predicate was specified instead of an equality predicate (i.e., `name` **`LIKE`** `Athena` instead of `name = "Athena"`). In this example, the operator first looks for the key `"name"`. After finding it, the operator looks to see if the key's corresponding value (the bytes before the delimiter `‘,’`) is exactly equal to `"Athena"`.

If the system permitted `LIKE` queries with the key-value search RF, then the second example in the figure, `"name": "Actually, Athena"`, would return a false negative, since the value search term `"Athena"` associated with the key `"name"` would never be found. Concretely, the problem is that the delimiter `','` can also appear within the value in this record, and it is impossible to distinguish between these two cases without a full parse of the entire key-value pair (and by extension, the full record)[1]. By only allowing equality predicates, false negatives cannot occur: if the search finds one of the delimiters but not the value search term, then either the delimiter appears after the entire value, or the value search term is not present. For the same reason, false negatives can also occur if the value search term and the delimiter set have any overlapping bytes; Sparser disallows this as well.

5 Sparser's Optimizer

Sparser's RFs provide an efficient but inexact mechanism for discarding records before parsing: these operators have high throughput but also produce false positives. To decrease the overall false positive rate while processing data, Sparser combines individual RFs into an RF cascade to maximize the overall filtering and parsing throughput. Finding the best cascade is challenging because a cascade's performance is both data- and query-dependent. Therefore, we present an optimizer that employs a cost model to score and select the best RF cascade. The optimizer takes as inputs a query predicate, a bytestream from the input file, and a full parser, and outputs an RF cascade to maximize the expected parsing throughput. Overall, the optimizer proceeds as follows:

1. Compile a set of possible RFs based on the clauses in the query predicate (§5.1).
2. Draw a sample of records from the input and measure data-dependent parameters such as the execution cost of the full parser, the execution costs of each RF, and the passthrough rates of the each RF on the sample (§5.2).
3. Generate a logical set of valid cascades to evaluate using the possible RFs (§5.3). A valid cascade does not produce false negatives.
4. Enumerate the possible valid RF cascades and select the best one using the previously estimated costs and passthrough rates (§5.4).

5.1 Compiling Query Predicates into Possible RFs

The first task in the optimizer is to convert the user-specified query predicate into a set of possible RFs. The query predicate is a boolean expression evaluated on each record: if a record causes the expression to evaluate to true, the record passes, and if the expression evaluates to false, the record may be discarded. By definition, the RFs generated by the optimizer for a given query must produce only false positives with respect to this boolean expression, but no false negatives (i.e., an RF may occasionally return true when the predicate evaluates to false, but never vice versa). The query predicate may also contain conjunctions and disjunctions that the optimizer must consider when generating RFs. Sparser thus takes follows steps to produce a set of possible RFs:

1. Convert the boolean query predicate to disjunctive normal form (i.e., of the form $(a \wedge b \ldots) \vee (c \wedge \ldots) \vee \ldots$). DNF allows Sparser's optimizer to systematically generate RF cascades that never produce false negatives. We refer to an expression with only conjunctions (e.g., $a \wedge b \wedge \ldots$) as a *conjunctive clause*.
2. Convert each *simple predicate* (i.e., predicates without conjunctions or disjunctions, such as equality or `LIKE` predicates) in the conjunctive clauses into one or more RFs. We elaborate on this procedure below using Listing 2 as an example.

```
(name = "Athena" AND text = "Greetings")
          OR name = "Jupiter"
```

Listing 2: An example predicate in DNF with two conjunctive clauses and three simple predicates.

Since each RF represents a search for a raw byte sequence, the conversion from a simple predicate to a set of RFs is format-dependent. For example, when parsing JSON, a predicate such as `name = "Athena"` in Listing 2 will produce both substring and key-value search RFs. However, for binary formats such as Avro and Parquet, field names (e.g., `text`) are typically not present in the data explicitly, which means that key-value search RFs would not be effective. Therefore, the optimizer only produces substring search RFs for these binary formats. For the sake of brevity, this section discusses only the JSON format (and assumes queries are over raw textual JSON data), which supports the full range of RFs available in Sparser.

For each simple predicate, Sparser produces a substring search RF for each 4- and 8-byte substring of each *token* in the predicate expression. A token is a single contiguous value in the underlying bytestream. Sparser generates 2-byte substring search RFs only if a token is less than 4-bytes long. As an example, the simple predicate `name = "Athena"` in Listing 2 contains two tokens `"name"` and `"Athena"`. For this simple predicate, the optimizer would generate the following substring search RFs: `"name"`, `"Athe"`, `"then"`, and `"hena"`.The optimizer additionally produces an RF that searches for each token in its entirety: `"name"` and `"Athena"` in this instance. Lastly, because `name = "Athena"` is an equality predicate, the optimizer generates key-value search RFs with the key `"name"` and the value set to each of the 4-byte substrings of `"Athena"`.

Each simple predicate is now associated with a set of RFs where each RF only produces false positives. If any RF in the set fails, the simple predicate also fails. By extension, each conjunctive clause is also associated with a set of RFs with the same property: for a conjunctive clause with n simple predicates, this set is $\bigcup_{i=1}^{n} r_i$, where r_i is the RF set of the ith simple predicate in the clause. This follows from the fact that RFs cannot produce false negatives: if any one RF in a conjunctive clause fails, some simple predicate failed, and so the full conjunctive clause must fail. To safely discard a record when processing a query with disjunctions, the optimizer must follow one rule when generating RF cascades: an RF from *each* conjunctive clause must fail to prevent false negatives. Returning to the example in Listing 2, the optimizer must ensure that an RF from *both* conjunctive clauses fails before discarding a record to prevent false positives.

5.2 Estimating Data-Dependent Parameters by Sampling

The next step in Sparser's optimizer is to estimate data-dependent parameters by drawing a sample of records from the input and executing the possible RFs from §5.1 and the full parser on the sample. Specifically, the optimizer requires the passthrough rates of the individual RFs, the runtime costs of executing the individual RFs, and the runtime cost of the full parser. This sampling technique is necessary because these parameters can vary significantly based on the format and dataset. For example, parsing a binary format such as Parquet requires fewer cycles than parsing a textual format such as JSON. Thus, for Parquet data, Sparser should choose a computationally inexpensive cascade to minimize runtime overhead, and the optimizer should capture that tradeoff.

[1]Extending the set of delimiters to include `'"'` would also yield false negatives for scenarios in which an escaped double-quote occurs within the entire value string

Algorithm 1 Estimating Data-Dependent Parameters by Sampling

```
1:  procedure ESTIMATE($records, candidateRFs$)
2:      $C \leftarrow$ len($candidateRFs$)
3:      $R \leftarrow$ len($records$)
4:      $ParserRuntime \leftarrow 0$                    ▷ Average parser runtime
5:      $RFRuntimes[C] \leftarrow 0$                    ▷ Average RF runtimes
6:      $B[C, R] \leftarrow 0_{C,R}$                      ▷ $C \times R$ matrix of bits
7:      for $(j, record) \in records$ do
8:          update running avg. $ParserRuntime$ with $parser(record)$
9:          for $(i, RF) \in candidateRF$ do
10:             update running avg. $RFRuntimes[i]$ with $RF$ on $record$
11:             if $RF \in record$ then
12:                 $B[i, j] \leftarrow 1$
        return $B$, $ParserRuntime$, $RFRuntimes$
```

To store the passthrough rates of the individual RFs, the optimizer uses a compact bit-matrix representation. This matrix stores a 1 at position i, j if the ith RF passes the jth record in the sample, and a 0 otherwise. Rather than storing the passthrough rate as a single numerical value per RF, each row in the bit-matrix compactly represents precisely which records in the sample passed for each RF as a bitmap. The optimizer leverages this data structure when scoring cascades with its cost model (§5.4) to compute the joint passthrough rates of multiple RFs efficiently.

Algorithm 1 summarizes the full parameter estimation procedure. The optimizer first initializes a $C \times R$ bit-matrix, where C is the number of possible RFs and R is the number of sampled records. For each sampled record, the optimizer updates an average of the full parser's running time in CPU cycles (e.g., using the x86 `rdtsc` instruction). Sparser can use any full parser, such as Mison. Then, for each RF, the optimizer applies the RF to the sampled record and measures the time the running time in CPU cycles. If RF i passes the record j, bit i, j is set to 1 in the matrix. After sampling, the optimizer has a populated matrix representing the records in the sample that passed for each RF, the average running time of each RF, and the average running time of the full parser.

5.3 Cascade Generation and Search Space

The third step in the optimizer is to generate valid RF cascades from the query predicate. Recall that, for a cascade to be valid in Sparser's optimizer, *at least one RF from each conjunctive clause in the query predicate must fail before discarding a record.* RF cascades are thus binary trees, where non-leaf nodes are RFs, leaf nodes are decisions (parse or discard), and edges represent whether the RF passes or fails a record. Figure 6 shows an example query predicate with examples of generated valid and invalid cascades. By considering at least one RF from every conjunctive clause, the optimizer only generates valid cascades, which may have false positives—but no false negatives—when evaluated on a given record from the input file.

The optimizer enumerates all cascades up to depth D that meet the above constraint. Our optimizer uses a pruning rule to prune the search space further by skipping cascades where two RFs from the same conjunction have overlapping substrings (e.g., a cascade which searches for `"Athena"` and `"Athe"`). If both candidate RFs appear in the same conjunction, the first string implies that the second will be found, and both must pass unconditionally for the conjunction to pass; thus, the optimizer should only consider one of the two. For completeness, the optimizer also considers the empty cascade (i.e., always parsing each record) to allow efficient formats such as Parquet to skip raw filtering altogether for queries that will exhibit no speedup. In our implementation, we set $D = max(\#\text{ Conjunctive Clauses}, 4)$ RFs, and generate up to 32 possible candidate RFs. For the queries in §7, we show that these choices

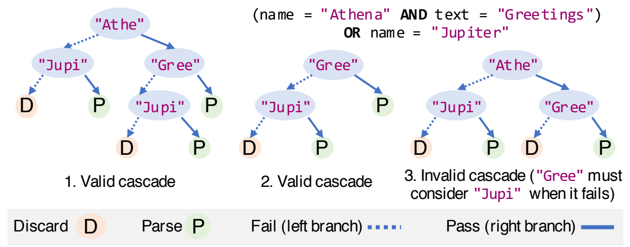

Figure 6: A set of RF cascades for the predicate in Listing 2. The third cascade does not check an RF from both conjunctive clauses on some paths and is thus invalid. The second cascade does not check all RFs in a conjunction but is still valid, since it checks one RF from each conjunctive clause.

Given a $C \times R$ bit-matrix of estimates B:	
$\Pr[a]$	$\texttt{popcnt}(B[a, \ldots])/R$
$\Pr[\neg a]$	$\texttt{popcnt}(\neg B[a, \ldots])/R$
$\Pr[a, .., z]$	$\texttt{popcnt}(B[a, \ldots] \wedge \ldots \wedge B[z, \ldots])/R$

Table 2: Estimating joint probabilities using the bit-matrix. $B[i, ...]$ indicates accessing the sampled bits for RF i. Bit i, j is set if RF i passed sampled record j. Bitwise operators ($\neg$, $\wedge$) use SIMD.

still generate cascades with overall parsing time within 10% of the globally optimal cascade.

5.4 Choosing the Best Cascade

Given a set of candidate cascades, Sparser's optimizer is left with one final task: choosing the best cascade. To make this choice, the optimizer evaluates the expected per-record CPU time of each cascade using a cost model, and selects the one with the lowest expected cost.

The cost of an RF cascade depends on c_i, the cost of executing the ith RF in given cascade, $\Pr[execute_i]$, the probability of executing the ith RF, as well as c_{parse} and $\Pr[execute_{parse}]$, which represent the respective cost and probability of executing the full parser. The optimizer measures the passthrough rates of the individual RFs in the previous step, as well as the execution times of the RFs and the full parser (c_i and c_{parse} respectively). However, for any RF i that relies on other RFs to pass or fail, $\Pr[execute_i]$ will be a joint probability. For example, in the example cascade $x \rightarrow y \rightarrow z$, the RF z will only execute after the first two RFs passed the record; therefore, $\Pr[execute_z] = \Pr[x, y]$, where $\Pr[x]$ and $\Pr[y]$ are the passthrough rates of x and y the optimizer measured in the previous step.

The challenge is that these joint probabilities are not necessarily independent (i.e., $\Pr[x, y] \neq \Pr[x] \Pr[y]$). For example, an RF that searches for the substring `"Gree"` may be highly correlated with an RF that searches for the substring `"ting"`, because both may indicate the presence of the string `"Greetings"`. Our evaluation shows that a strawman optimizer that does not consider these correlations achieves parsing throughputs $2.5\times$ lower than Sparser, because the strawman chooses an inferior cascade.

Another strawman solution is to estimate the joint passthrough rates of multiple RFs directly by executing RF cascades on the sample of records described in §5.2. However, executing each combination of RFs on the sample is inefficient, since this requires executing a combinatorial number of cascades.

Instead, Sparser's optimizer uses the bit-matrix representation (§5.2) to quickly estimate the joint passthrough rates using only sample-based measurements of the individual RFs. Recall that the matrix stores as a single bit whether an RF passes or fails each record in the sample (a 1 if the record passed the RF, and 0 otherwise). The passthrough rate of RF i is thus the number of 1s (i.e., the `popcnt`) of the ith row, or bitmap, in the matrix. Conversely, the probability of any RF i not passing a record is the number of 0s in row i. The joint passthrough rate of two RFs i and k is the number of 1s in the bitmap after taking the bitwise-and of the ith and kth bitmaps.

The key advantage to this approach is that these bitwise operations have SIMD support in modern hardware and complete in 1-3 cycles on 256-bit values on modern CPUs (roughly 1ns on a 3GHz processor). The matrix thus allows the optimizer to quickly estimate joint passthrough probabilities of RFs. This optimization allows Sparser to scale efficiently and accurately to handle complex user-specified query predicates that combine multiple predicate expressions. Table 2 summarizes the matrix operations.

With an efficient methodology to accurately compute the joint probabilities, the optimizer can now score each cascade and choose the one with the lowest cost. Let $R = \{r_1, \ldots, r_n\}$ be the set of RFs in the RF cascade. To evaluate C_R, the expected cost of the cascade on a single record, Sparser's optimizer computes the following:

$$C_R = \left(\sum_{i \in R} \Pr[execute_i] \cdot c_i \right) + \Pr[execute_{parse}] \cdot c_{parse}.$$

As an example, consider the first cascade in Figure 6. The probabilities of executing each RF in the cascade are:

$$\begin{aligned}
\Pr[execute_{\text{Athe}}] &= 1, \\
\Pr[execute_{\text{Gree}}] &= \Pr[\text{Athe}], \\
\Pr[execute_{\text{Jupi}}] &= \Pr[\neg\text{Athe}] + \Pr[\text{Athe}, \neg\text{Gree}], \\
\Pr[execute_{parse}] &= \Pr[\neg\text{Athe}, \text{Jupi}] + \\
&\quad \Pr[\text{Athe}, \text{Gree}] + \Pr[\text{Athe}, \neg\text{Gree}, \text{Jupi}].
\end{aligned}$$

The cost of the full cascade is therefore:

$$\sum_{i \in \{\text{Athe}, \text{Gree}, \text{Jupi}, parse\}} \Pr[execute_i] \times c_i.$$

§7.4 shows that, with the bit-matrix technique to compute joint probabilities, the optimizer adds at most 1.2% overhead in our benchmark queries, including the time to sample records and score cascades.

5.5 Periodic Resampling

Sparser occasionally recalibrates its cascade to account for data skew or sorting in the underlying input file. §7 shows that recalibration is important for minimizing parsing runtime over the *entire* input, because a cascade chosen at the beginning of the dataset may not be effective at the end. For instance, consider an RF that filters on a particular date, and the underlying input records are also sorted by date. The RF may be highly ineffective for one range of the file (e.g., the range of records that all match the given date in the filter) and very effective for other ranges. To address this issue, Sparser maintains an exponentially weighted moving average of its own parsing throughput. In our implementation, we update this average on every 100MB block of input data. If the average throughput deviates significantly (e.g., 20% in our implementation), Sparser reruns its optimizer algorithm to select a new RF cascade.

6 Implementation

We implemented Sparser's optimizer and RFs in roughly 4000 lines of C. Our implementation supports mapping query predicates to RFs for text logs, JSON, Avro, Parquet, and PCAP, the standard binary packet capture format [51]. RFs leverage Intel's AVX2 [8] vector extensions, though other architectures feature similar operators [44].

JSON. Our JSON implementation uses (and evaluates in §7) two existing state-of-the-art JSON parsers: Mison [38] and RapidJSON [53]. Sparser assumes that the input bytestream contains textual JSON records[2] (e.g., a Tweet from the Twitter Stream API) terminated by a newline character. Sparser uses SIMD to find the start of each record by searching for the newline, and applies the RF cascade on the raw byte buffer, where each RF searches until the following newline. If the record passes the full cascade, Sparser passes a pointer to the beginning of the record to the full parser. Otherwise, Sparser skips it and continues filtering the remaining bytestream.

Binary Formats. Sparser can also integrate with binary formats. We evaluate integration with Avro, Parquet, and PCAP in §7. For binary data, records are not explicitly delimited (e.g., by newlines), so Sparser does not know where to start or stop a search for a given RF. Rather than search line by line, Sparser treats the full input buffer as a single record and begins searching from the very beginning of the buffer. When an RF finds a match, Sparser uses a format-specific function for navigating and locating different records in the file. In our implementation, this function moves a pointer from the last processed record by the full parser to the record containing the match. The function also computes the end of the matched record (in most binary formats, this is the start of the record plus the record length, stored as part of the data) and returns both the pointer to the matching record and the length back to Sparser's search function to check the remaining RFs within the byte range. If all RF matches pass, Sparser calls the callback again and the record is processed just as before. Otherwise, Sparser resets its record-level state and continues. Sparser speeds up parsing binary formats by skipping over large blocks of data, processing only regions that contain RF matches (§7).

Integration with Spark. We also integrated Sparser with Spark [5] using Spark's Data Sources API, which provides a pluggable mechanism for accessing structured data though Spark SQL. The Data Sources API enables column pruning and filtering to be pushed down to the parser itself, in line with the core tenets of Sparser. To date, however, these features have primarily been used only in optimized columnar formats, such as Parquet, or for index construction to access individual rows in the underlying data source efficiently. In the API, Spark passes individual file partitions (which map to a filename, byte offset, and length) to a callback function; these arguments are then passed via the Java Native Interface (JNI) to call into Sparser's C library. This means that Sparser runs its calibration, raw filtering, and parsing steps on a per file-partition basis, rather than on a single file. Sparser reads, filters, and parses data, writing the extracted fields directly to an off-heap buffer allocated in Spark to store the parsed records. Spark treats the off-heap buffer as a standard DataFrame for the remainder of the query.

7 Evaluation

We evaluate Sparser and the raw filtering approach across a variety of workloads, datasets, and data formats. We find that:

- With raw filtering, Sparser accelerates diverse analytics workloads by filtering out records that do not need to be parsed.

[2]The JSON standard [22] allows any character to be escaped (e.g., the character `"A"` and its escaped Unicode representation, `"\u41"`, should be considered equal). For text with escaped values, users can configure Sparser to additionally search for the escaped Unicode strings. Our datasets in §7 do not include escaped characters.

Query Name	Query	Selectivity (%)
Twitter 1	`COUNT(*)WHERE text LIKE '%Donald Trump%'AND date LIKE '%Sep 13%'`	0.1324
Twitter 2	`user.id, SUM(retweet_count)WHERE text LIKE '%Obama%'GROUP BY user.id`	0.2855
Twitter 3	`id WHERE user.lang == 'msa'`	0.0020
Twitter 4	`distinct user.id WHERE text LIKE '%@realDonaldTrump%'`	0.3313
Censys 1	`COUNT(*)WHERE p23.telnet.banner.banner != null AND autonomous_system.asn = 9318`	0.0058
Censys 2	`COUNT(*)WHERE p80.http.get.body LIKE '%content=wordpress 3.5.1%'`	0.0032
Censys 3	`COUNT(*)WHERE autonomous_system.asn=2516`	0.0757
Censys 4	`COUNT(*)WHERE location.country = 'Chile'AND p80.http.get.status_code != null`	0.1884
Censys 5	`COUNT(*)WHERE p80.http.get.servers.server LIKE '%DIR-300%'`	0.1884
Censys 6	`COUNT(*)WHERE p110.pop3.starttls.banner != null OR p995.pop3s.tls.banner != null`	0.0001
Censys 7	`COUNT(*)WHERE p21.ftp.banner.banner LIKE '%Seagate Central Shared%'`	2.8862
Censys 8	`COUNT(*)WHERE p20000.dnp3.status.support=true`	0.0002
Censys 9	`asn, COUNT(ipnt)WHERE autonomous_system.name LIKE '%Verizon%'GROUP BY asn`	0.0002
Bro 1	`COUNT(*)WHERE record LIKE '%HTTP%'AND record LIKE '%Application%'`	15.324
Bro 2	`COUNT(*)WHERE record LIKE '%HTTP%'AND (record LIKE '%Java*dosexec%'OR record LIKE '%dosexec* Java%')`	1.1100
Bro 3	`COUNT(*)WHERE record LIKE '%HTTP%'AND record LIKE '%http*dosexec%'AND record LIKE '%GET%'`	0.5450
Bro 4	`COUNT(*)WHERE record LIKE '%HTTP%'AND (record LIKE '%80%'OR record LIKE '%6666%'OR record LIKE '%8888%'OR record LIKE '%8080%')`	12.294
PCAP 1	`* WHERE http.request.header LIKE '%GET%'`	81
PCAP 2	`* WHERE http.response AND http.content_type LIKE '%image/gif%'`	1.13
PCAP 3	`Flows WHERE tcp.port=110 AND pop.request.parameter LIKE '%user%'`	0.001
PCAP 4	`Flows WHERE http.header LIKE '%POST%'AND http.body LIKE '%password%'`	0.0095

Table 3: Queries used in the evaluation. §7.1 elaborates on the datasets and sources of the queries.

Sparser can improve the parsing throughput of state-of-the-art JSON parsers up to 22×. For distributed workloads, Sparser can improve the end-to-end runtime of Spark SQL queries up to 9×.

- Sparser can also accelerate the parsing throughput of binary formats such as Avro and Parquet by up to 5×. For queries over unstructured text logs, Sparser can reduce the query runtime by up to 4×.
- Sparser's optimizer improves parsing performance compared to strawman approaches, selecting RF cascades that are within 10% of the global optimum while only incurring a 1.2% runtime overhead during parsing.

7.1 Experimental Setup

We ran distributed Spark experiments on a 10-node Google Cloud Engine cluster using the `n1-highmem-4` instance type, where each worker had 4 vCPUs from an 2.2GHz Intel E5 v4 (Broadwell), 26GB of memory, and locally attached SSDs. We used Spark v2.2 for our cluster experiments. Single-node benchmarks ran on an Intel Xeon E5-2690 v4 CPU with 512GB of memory. All single-node experiments were single-threaded—we found that Sparser scales linearly with the number of cores for each workload, and omit these results for brevity. For all experiments, we compiled Sparser using `clang-4.0` with `-O3 -march=native` to enable use of AVX2.

Our experiments ran over the following real-world datasets and queries, with some experiments running over a subset of the data. Table 3 summarizes the queries and their selectivities.

Twitter Tweets. We used the Twitter Streaming API [63] to collect 68GB of tweets in the form of JSON over a period of three days. We benchmarked against 23GB of the data for our single-node experiments, and the entire dataset for our distributed experiments. We obtained the Twitter queries from [38, 58].

Censys Scan. We obtained a 652GB JSON dataset acquired from Censys [25], a search engine broadly used in the Internet security community. We benchmarked against 16GB of the data for our single-node experiments, and the entire dataset for our distributed experiments. Each record in the dataset represents an open port on the wide-area Internet. Censys data is highly nested: each data point is over 5KB in size. We obtained the queries over Censys data by sampling randomly from the 50,000 most popular queries to the engine.

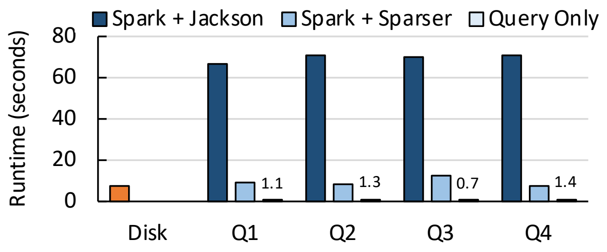

Figure 7: Twitter queries on Spark over JSON data end-to-end. The time to load the data from disk is shown on the far left. Spark loads and filters the data using Sparser into DataFrames and then uses Spark SQL to process the query.

Bro IDS Logs. Bro [11] is a widely deployed network intrusion detection system that generates ASCII logs while monitoring networks. Network security analysts perform post-hoc data analyses on these logs to find anomalies. We obtained a 10GB dataset of logs and a set of queries over them produced by packet traces from security forensics exercises [12–14].

Packet Captures. To evaluate Sparser's applicability in other domains, we obtained a 5GB trace of network traffic from a university network. Traffic is stored in standard binary file format called PCAP [50], which stores the binary representation of individual network packets. We selected queries for this trace from [21, 27, 29], which represent real workloads over captured network traffic, such as searching for insecure network connections.

7.2 End-to-End Workloads

We first study the performance of Sparser on a variety of real end-to-end applications, ranging from Spark SQL queries to network packet filtering and log analysis.

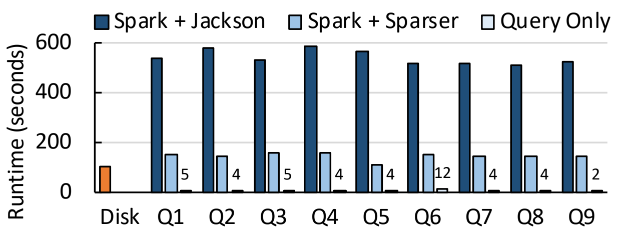

Figure 8: Censys queries on Spark over JSON data.

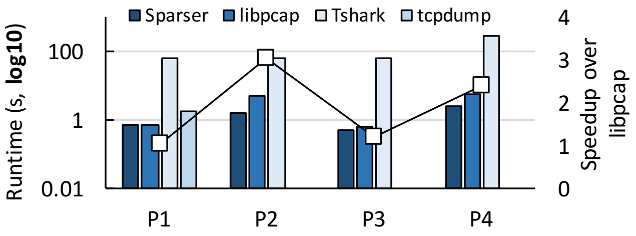

Figure 9: Packet filtering on binary `tcpdump` PCAP files. The line shows the speedup of Sparser over `libpcap`.

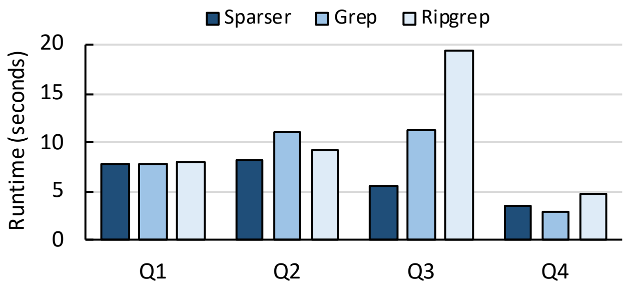

Figure 10: Performance of log analysis tasks on Bro IDS logs, where Sparser is used as a `grep`-like search tool.

Spark Queries. To benchmark Sparser's effectiveness parsing JSON in a production-quality query engine, we executed the four Twitter queries and nine Censys queries (all of which are over JSON data) from Table 3 on our 10-node Spark cluster and measured the end-to-end execution time.

Figures 7 and 8 show the end-to-end execution time of native Spark (which uses the Jackson JSON parser [31]) vs. Sparser integrated with Spark via the Data Source API [56]. Data is read from disk and passed to Sparser as chunks of raw bytes. Sparser runs its optimizer, chooses an RF cascade, and filters the batches of data, returning a filtered bytestream to Spark. Spark then parses the filtered bytestream into a Spark SQL DataFrame and processes the query. The presented execution time includes disk load, parsing (in Sparser, this includes both the optimizer's runtime and filtering), and querying. In each query, Sparser outperforms Spark without Sparser's raw filtering by at least 3×, and up to 9×. Speedups in these workloads comes from avoiding parsing—since each query has high selectivity, Sparser does not need to parse most of the records. We note that these queries see significant speedups end-to-end, even when including the time to load data from disk, showing that parsing data can have a substantial effect on total execution time.

Packet Filtering. To illustrate that raw filtering can accelerate a diverse set of analytics workloads, we apply it to filtering captured network traffic and evaluate its performance. Using Sparser, we implemented a simple packet-analysis library and benchmarked its throughput against `tshark` [61], a standard tool in the networking community for analyzing packet traces. We also compared against `tcpdump` [59] (a lightweight version of `tshark`) and a simple `libpcap` -based C program that hard-codes the four queries. The `libpcap` [39] library is the standard C library for parsing network packets: this baseline represents the packet parsing procedure without any overheads imposed by other systems.

Figure 9 shows the results: in the first query, each system (except `tshark`) performs similarly—the query's relatively low selectivity prevents Sparser from delivering large speedups. In the second query, Sparser performs roughly 3× faster than `libpcap`, which parses and searches each packet for a string. The `tcpdump` tool does not support searches inside a packet's payload. In the third and fourth queries, Sparser outperforms `libpcap` by 2.5×. Overall, Sparser accelerates these packet filtering workloads using its search technique, even compared to C-based implementations with almost no system overhead.

Bro Log File Analysis. Ad-hoc log analysis is a common task, especially for IT administrators maintaining servers, or security experts analyzing logs from intrusion detection systems such as Bro [11]. Tools such as `awk` and `grep` aid in these analyses; generally, the first step is to use these tools to filter for events of interest (e.g., errors, specific protocols, etc.). We collected a set of network forensic analysis workloads [12–14] and replaced the log filtering stage with Sparser's optimizer and raw filtering technique. Each query looks for a set of threat signatures and then performs a count with `wc`.

Figure 10 shows the results; compared to standard tools based on traditional string search algorithms (e.g., Boyer-Moore), Sparser improves performance by up to 4×. For the first, second, and fourth queries, the performance of all three systems perform similarly in this case, since both GNU `grep` and `ripgrep` (the fastest `grep` implementation we found [26]) are optimized and use vectorization. Sparser marginally outperforms both by searching for the most uncommon substring to discard rows faster.

In the third query, we search for one of three terms in each line, but some terms are much more selective than others. Sparser's optimizer identifies the most selective term and searches for it, while the other two systems naively search for the first term (the default policy when searching for multiple strings at once). Overall, Sparser is competitive with and sometimes faster than command-line string search tools such as `grep` and `ripgrep`, despite the fact that these tools are also designed for hardware efficiency and do not perform any parsing of the inputs.

7.3 Comparison with Other Parsers

JSON Parsing Performance. To evaluate Sparser's ability to accelerate JSON parsing, we integrated Sparser with RapidJSON [53] and Mison [38], two of the fastest C/C++-based parsers available. Because the Mison implementation is not open source, we implemented its algorithm as described in the paper using Intel AVX2, and benchmark only against the time to create its per-record index (i.e., we assume that using the index to extract the fields and evaluating the predicate is free). We believe this is a fair lower bound on its performance.

Figure 11 shows the parsing runtime of RapidJSON and Mison both with and without Sparser on the nine Censys queries; we benchmarked the queries on a 15GB sample of the full dataset on a single node. Because these queries have low selectivity, can accelerate these optimized parsers up to 22×, due to Sparser's ability to to efficiently filter records that do not need to be parsed.

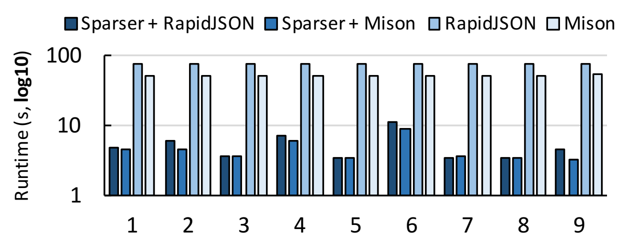

Figure 11: Parsing time for the nine Censys JSON queries compared against Mison and RapidJSON.

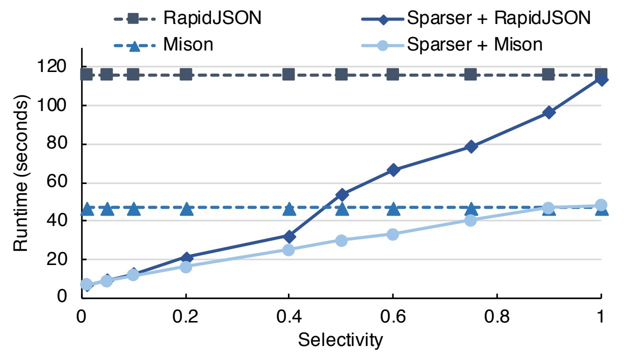

Figure 12: Effect of selectivity vs. parsing time for parsing the Twitter JSON data. We compare Sparser to both RapidJSON and Mison. As the selectivity decreases, The performance of the Sparser-enabled parser increases to match the runtime of the parser without Sparser.

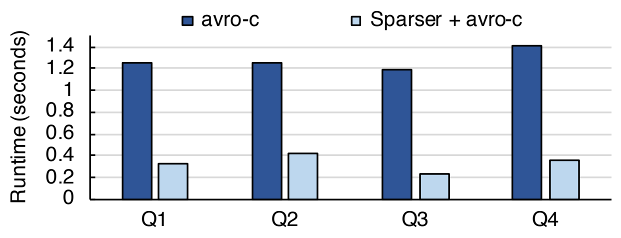

Figure 13: Parsing time for Avro data on the four Twitter queries using `avro-c` with and without Sparser.

Raw filtering is thus complementary to Mison's optimizations for projecting fields from JSON.

JSON Parsing Sensitivity to Selectivity. An underlying assumption of raw filtering is that many queries in exploratory workloads exhibit high selectivity. This raises an important question: how does raw filtering perform on queries with low selectivity? To study Sparser's sensitivity to selectivity, we benchmarked the parsing runtime of Sparser + {RapidJSON, Mison} on a synthetic query from the Twitter dataset, and varied the selectivity of the query from 0.01% to 100%.

Figure 12 shows the results, comparing Sparser's parsing time against RapidJSON and Mison at various filter selectivities. As the selectivity increases, the benefits of Sparser diminish. However, Sparser still outperforms both parsers by rejecting some records and adaptively tuning its cascade to choose fewer filters as the selectivity increases. In the worst case, when all the data is selected, Sparser always calls the downstream parser, resulting in no speedup.

Binary Formats: Avro and Parquet. In addition to human-readable formats such as JSON, many big data workloads operate over record-oriented binary formats such as Avro [6], or columnar binary formats such as Parquet [48]. Both of these formats are optimized to reduce storage and minimize the overhead of parsing data when loading it

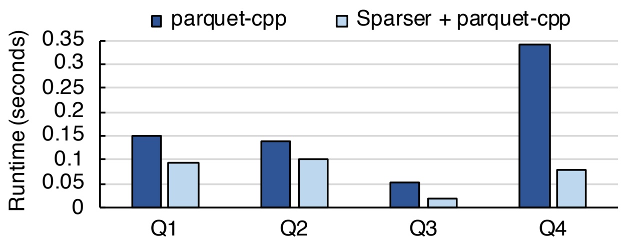

Figure 14: Parsing time for Parquet data on the four Twitter queries using `parquet-cpp` with and without Sparser.

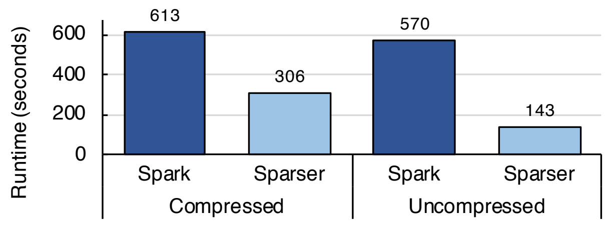

Figure 15: Sparser's runtime on Twitter Q1 on an uncompressed vs. `gzip`-compressed JSON file, benchmarked in Spark on a single node. Sparser speeds up end-to-end processing times even when the CPU must decompress data.

into memory. To evaluate Sparser's effectiveness on these formats, we converted the Twitter dataset (originally in JSON) to both Avro and Parquet files using Spark, and benchmarked the four Twitter queries on each format on a single node. Figure 13 summarizes the results for parsing Avro: compared to `avro-c`, an optimized C parser, Sparser's raw filtering reduces the end-to-end query time by up to 5×. For Parquet (Figure 14), Sparser improves the parsing time of the `parquet-cpp` library by up to 4.3× across the four queries.

Sparser is able to accelerate parsing on Avro and Parquet data because these formats can also be expensive to parse. In Avro, data is organized into blocks that contain records, where each record is stored as a sequence of fields without delimiters, and each variable-length field is prefixed with its length encoded as a variable-length integer [7]. Avro also prefixes each block of records with its corresponding byte length, but does not do so for each individual record. Therefore, to parse and filter Avro data, a parser must find each field one at a time to traverse through each record, and check the relevant fields in each record. However, with Sparser, we can skip entire blocks of records if the RFs do not match anywhere in the block. If the RFs do match, Sparser can skip checking the fields of every record in the block, and only check the fields in records where the RFs matched.

In Parquet, Sparser uses a similar strategy, but adapts it to Parquet's columnar format: we seek to the portions of the file that contain the columns of interest, search over those columns, and then seek to the matches within each column. Columns in Parquet are split across Row Groups [49], and we can seek to the Row Group that contains our potential match. Within each Row Group of a given column, the column data is stored across pages, which are typically 8KB each. Within each data page, the column values are stored using the same format found in Avro (i.e., length in bytes, followed by value), which therefore requires the same sort of incremental traversal over the values.

Speedups on Compressed Data. Data on disk is often compressed, so data loading often requires running a computationally expensive decompression algorithm, such as `gzip`. While there are some proposed solutions for directly querying compressed data [2], this step is unavoidable for general query processing. To show that

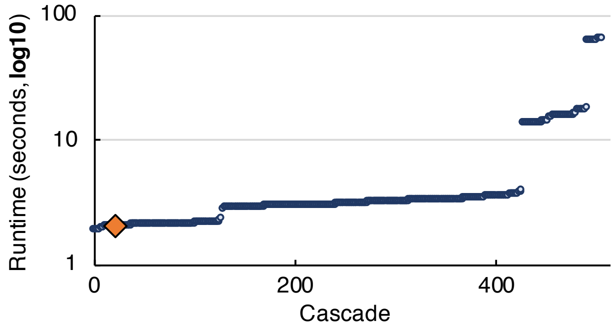

Figure 16: The performance of each cascade Sparser considers for Censys Q1, sorted by runtime from left to right. The difference between the best and worst of the 506 cascades is 35×. Sparser (the marked point) does not pick the globally best cascade, since it relies on sampling-based estimates of the RF probabilities. However, Sparser's selection is within 10% of the best cascade.

parsing is still an important factor even when data is compressed, we benchmarked Twitter Q1 on both an uncompressed and `gzip`-compressed version of the JSON data. Figure 15 shows the results of the end-to-end runtimes in Spark on a single node: even on compressed data, Sparser improves the end-to-end query runtime by 4× by minimizing the time spent parsing.

7.4 Evaluating Sparser's Optimizer

As described in Section 5, Sparser's optimizer examines the search space of possible RF cascades and selects the cascade that minimizes the expected parse time. Here, we examine the optimizer's impact on end-to-end query performance and measure its ability to select the appropriate RF cascade without incurring significant runtime overheads. We also evaluate the optimizer on multiple microbenchmarks to examine its key design decisions, such as the optimizer's modeling of joint passthrough rates and its periodic resampling strategy for handling skew in the input data.

Impact of the Optimizer. Sparser's optimizer uses a cost function to calculate the expected parsing time a given RF cascade; amongst many candidate cascades, the optimizer will select the one with the lowest expected cost. To show the overall impact of the optimizer, we ran the first Censys query on each cascade considered by the optimizer to study the difference in performance across the different candidates. Figure 16 shows the result, where each point represents a cascade, and cascades are sorted by runtime from left to right. Many of the cascades contain RFs that discard few records, which produces little improvement in end-to-end parsing time compared to a standard parser. However, the best cascades substantially reduce overall parsing times. Sparser selects one of the best cascades, showing that the optimizer effectively filters out poor RF combinations, and that Sparser's sampling-based measurement is sufficient to produce a cascade within 10% of the best-performing one.

To evaluate the effect of the bounded cascade depth and possible RFs (§5.3), we also ran all nine Censys queries without bounding the combinatorial search. For these queries, we found that only Q6 chose a cascade with depth greater than $D = 4$ (our maximum default depth), and the difference in runtime performance was 1.2%.

Optimizer Overhead. Table 4 summarizes the optimizer's average runtime across all nine Censys queries and includes a breakdown of the runtime across each of the optimizer's stages. On average, the optimizer spends 3ms end-to-end, including measuring the parser,

Runtime (ms)	Average	Min	Max
Query Time	5213	3339	11002
Optimizer	3.01	1.85	4.11
Measuring RFs	2.10	1.18	3.09
Measuring Parser	0.63	0.44	0.81
Scoring Cascades	0.28	0.05	0.77
Percent Overhead	0.67%	0.19%	1.18%

Table 4: Measurements from the optimizer on the nine Censys queries. On average, the optimizer takes 3ms to sample records, estimate RFs, measure the parser runtime, and search through and score cascades. Across all queries, the optimizer accounts for at most 1.2% of the total execution time.

	Cascade	Est. Sel.	Real Sel.	Runtime
Sparser	`"teln"` → `"30722"`	0.010%	0.031%	2.221s
Naive	`"teln"` → `"p23"`	0.090%	2.997%	4.454s

Table 5: Sparser's optimizer vs. a naive optimizer that assumes RFs are independent of one another. By capturing the interdependence betweens RFs, Sparser's optimizer can search for tokens that jointly minimize the passthrough rate, thus improving the parsing throughput.

measuring the RFs, and searching through cascades. We also measured the effects of the pruning rule that skips cascades with overlapping substrings and found that, on average, 86% of the cascades were not scored. Despite the combinatorial search space, the optimizer's pruning rule and use of bit-parallel operations enable it to account for less than 1.5% of the total running time on the Censys benchmark in the worst case.

Interdependence of RFs. The optimizer uses a bitmap-based data structure to store the passthrough rates of each individual RF, allowing it to quickly compute the joint passthrough rates of RF combinations. However, a strawman optimizer could also assume that RFs are independent of one another, and use only the individual passthrough rates to evaluate candidate cascades. To show the impact of capturing the correlations between RFs, we compared the performance of this strawman against Sparser's optimizer on Censys Q1. (For demonstration purposes in this experiment, we substitute `asn = 9318` with `asn = 30722`, a much more common value in the data.)

Table 5 summarizes the results of this experiment. Because the RF that searches for `"30722"` has a high passthrough rate (9.83%), the strawman optimizer instead searches for two tokens with smaller passthrough rates (3% each): `"teln"` (a substring of `"telnet"`) and `"p23"`. However, searching for both tokens adds marginal benefit compared to searching for only one of them—if the token `"teln"` is present, it is almost always accompanied by `"p23"` as well. Sparser's optimizer captures the co-occurrence rate of the two terms and instead searches for `"teln"` and `"30722"`. Although `"30722"` does appear frequently throughout the input on its own, it occurs much less frequently with `"teln"`. As a result, the cascade chosen by Sparser's optimizer is 2× faster than the naive optimizer's cascade when evaluating the query.

Key-Value RF. While the substring search RF has a straightforward application in most workloads, the key-value RF is useful for certain queries that search for uncommon associations between fields and values in the data. To measure its utility, we benchmarked Sparser both with and without the key-value RF on a synthetic query over the Twitter dataset. The query finds all tweets with `favorited = true` and has a small selectivity—only 0.002%. Figure 17 shows

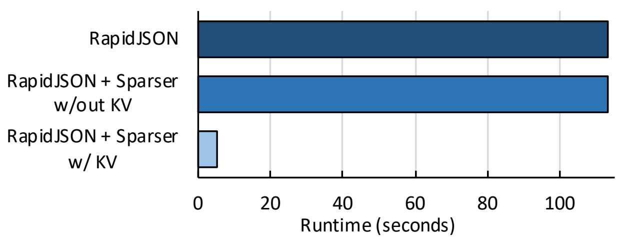

Figure 17: Advantage of the key-value RF in Sparser on a synthetic Twitter query that searches for tweets where `favorited = true`. Without the key-value RF, Sparser provides no additional speedups compared to RapidJSON.

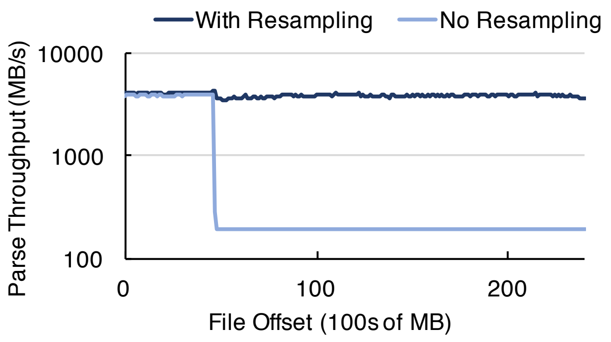

Figure 18: Resampling allows Sparser to adapt to changing distributions in the data. In the Twitter dataset, tweets are sorted by date, so Twitter Q1 (which filters based on date) benefits from resampling by detecting this change in the data.

the result: without the key-value RF, Sparser fails to outperform the standard parsers, since almost every record contains the terms `"favorited"` and `"true"`. The key-value RF associates both terms together when searching through the raw bytestream, thus enabling a 22× speedup.

Periodic Resampling. To study the impact of periodic resampling in Sparser's optimizer, we examine the first Twitter query from Table 3, which searches for tweets mentioning `"Donald Trump"` on a particular date. Because the tweets were collected as a stream, the `date` field has high temporal locality in the input file—the `date LIKE '%Sep 13%'` predicate selects all the data in some range, but none in the rest. We benchmarked Sparser both with and without its resampling step on this query, and Figure 18 shows the result. During the initial sampling, Sparser finds that the `date` predicate is highly selective and includes a substring RF based on `"Sep 13"`. However, in the range where the date does match, the RF no longer remains selective. With periodic resampling, Sparser detects this change and recalibrates its RF cascade to search for a substring of `"Donald Trump"`, rather than a substring of the date. By including this step in the optimizer, Sparser's parsing throughput over the entire input file is 25× faster than it would be otherwise.

8 Related Work

Processing Raw Data. Many researchers have proposed query engines over raw data formats. NoDB [3] proposes building indices incrementally over raw data to accelerate access to specific fields. Alagiannis et al. [4] and others [30] consider storage layouts and access patterns for query processing over raw data, and examine how to adapt to workloads online. ViDa introduces JIT-compiled access paths for adapting queries to underlying raw data formats [35–37]. Slalom [46] monitors access patterns to build indices for fast in-situ data access. SCANRAW uses parallelism to masks in-situ data access times via pipelining [18, 19], while Abouzied et al. [1] propose masking load times using MapReduce jobs. While these approaches propose full query engines over raw data, raw filtering focuses on the problem of filtering and loading it as quickly as possible using format-agnostic RFs and an optimizer. Existing raw processing systems can thus use raw filtering in a complementary manner to filter before downstream processing.

Parsers for Semi-Structured Data. For JSON parsing, the Mison JSON parser [38] is the closest to Sparser in that it takes both filtering expressions to apply to the data and a set of output fields to project as part of its API. Mison always begins by builds a structural index using SIMD and bit-parallel operators. The index finds special JSON characters such as colons and brackets to create a mapping from byte offset to field offset. Mison then builds another data structure called a pattern tree to speculatively jump to the desired field position using this structural index, and then applies predicates to the retrieved fields. We showed in §7 that just building the structural index in Mison is slower than rejecting RFs with Sparser on selective workloads. In addition, because Mison searches for format-specific delimiters to construct its index, its techniques are not applicable to binary formats that eschew delimiters, such as Avro and Parquet. Sparser is designed to work across both textual and binary formats, and speeds up queries across both. Nevertheless, since Sparser only filters data and Mison and other optimized JSON parsers [31, 53] extract values from data quickly, both approaches are highly complementary.

For XML, many approaches used optimized automata to parse and filter XML efficiently [16, 23, 24, 28]. In contrast, this work relies on SIMD instructions rather than automata to leverage data-parallelism in modern hardware. Similar to Mison, Parabix [15] uses SIMD instructions to parse XML, and Teubner et al. [60] and Moussalli et al. [41, 42] devise algorithms to leverage data-parallelism on GPUs and FPGAs to accelerate XML filtering and parsing. These systems still extract structural information about the format and, like with optimized JSON parsers, necessarily spend more time than an RF-based search for filtering data. Existing work on fast XML parsing is again complementary with raw filtering, because these systems can use raw filtering to filter data efficiently before parsing.

Predicate Ordering. Sparser's optimizer reorders predicates to optimize overall runtime and is inspired by a long lineage of work on predicate ordering in database systems. Babu et al. [9] propose a way to order conjunctive commutative filters to minimize runtime overhead by adaptively measuring selectivities and considering correlations across filters. The algorithms incur runtime overhead while filtering when accounting for correlations and explores the tradeoffs among ordering quality, decreased overhead, and algorithm convergence. Raw filtering instead uses a new SIMD-enabled optimizer to find an optimal depth-D ordering based on sampled selectivity estimates while always considering filter correlations, and also supports disjunctions of predicates. Scheufele et al. [55] propose an algorithm for optimal selection and join orderings but only consider the cost of individual predicates. Ma et al. [40] similarly order predicates using only their individual costs and selectivities, while Sparser considers correlations among the predicates. Lastly, Sparser's approach of combining multiple RFs into an RF cascade is inspired by previous work in computer vision, most notably the Viola-Jones object detector [64]. In Viola-Jones, a cascade is a single sequence of increasingly accurate but increasingly expensive classifiers; if any classifier is confident about the output, the cascade short-circuits evaluation, improving execution speed. In Sparser, an RF cascade is a binary tree, and the ordering of RFs in the tree is determined by their execution costs and joint passthrough rates.

Fast Substring Search. String and signature search algorithms are commonly used in network and security applications such as intrusion detection. DFC [20] is a recent algorithm for accelerating multi-pattern string search using small, cache-friendly data structures. Other work [57] accelerates multi-pattern string search using vector instructions or other optimizations [45, 62]. These signature search algorithms, however, are primarily designed for settings with thousands of signatures, where they must quickly match a packet against all the signatures using a finite state machine. In contrast, Sparser focuses on quickly rejecting records that do not match a small number of filters, allowing it to work effectively with a sequence of simple tests. Sparser also uses an optimizer to calibrate at runtime its selected RF cascade based on the input data.

9 Conclusion

We have presented raw filtering, a technique that accelerates one of the most expensive steps in data analytics applications—parsing unstructured or semi-structured data—by rejecting records that do not match a query without parsing them. We implement raw filtering in Sparser, which has two key components: a set of fast, SIMD-based *raw filter* (RF) operators, and an optimizer to efficiently select an RF cascade at runtime. Sparser accelerates existing high-performance parsers for semi-structured formats by 22× and provides up to an order-of-magnitude speedup on real-world analytics tasks, including Spark analytics queries and log mining.

10 References

[1] Abouzied, A., Abadi, D. J., and Silberschatz, A. Invisible loading: access-driven data transfer from raw files into database systems. In *Proceedings of the 16th International Conference on Extending Database Technology* (2013), ACM, pp. 1–10.

[2] Agarwal, R., Khandelwal, A., and Stoica, I. Succinct: Enabling queries on compressed data. In *NSDI* (2015), pp. 337–350.

[3] Alagiannis, I., Borovica, R., Branco, M., Idreos, S., and Ailamaki, A. Nodb: efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 241–252.

[4] Alagiannis, I., Idreos, S., and Ailamaki, A. H2o: A hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 1103–1114.

[5] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 1383–1394.

[6] Apache Avro. `https://avro.apache.org`.

[7] Apache Avro 1.8.1 Specification. `https://avro.apache.org/docs/1.8.1/spec.html`.

[8] Intel AVX2. `https://software.intel.com/en-us/node/523876`.

[9] Babu, S., Motwani, R., Munagala, K., Nishizawa, I., and Widom, J. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (2004), ACM, pp. 407–418.

[10] Bailis, P., Gan, E., Madden, S., Narayanan, D., Rong, K., and Suri, S. MacroBase: Prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 541–556.

[11] Bro. `https://www.bro.org/`.

[12] Bro Exchange 2013 Malware Analysis. `https://github.com/LiamRandall/BroMalware-Exercise`.

[13] Network Forensics with Bro. `http://matthias.vallentin.net/slides/bro-nf.pdf`, 2011.

[14] Understanding and Examining Bro Logs. `https://www.bro.org/bro-workshop-2011/solutions/logs/index.html`.

[15] Cameron, R. D., Herdy, K. S., and Lin, D. High Performance XML Parsing Using Parallel Bit Stream Technology. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds* (New York, NY, USA, 2008), CASCON '08, ACM, pp. 17:222–17:235.

[16] Candan, K. S., Hsiung, W.-P., Chen, S., Tatemura, J., and Agrawal, D. Afilter: adaptable xml filtering with prefix-caching suffix-clustering. In *Proceedings of the 32nd VLDB* (2006), VLDB Endowment, pp. 559–570.

[17] Writing a Really, Really Fast JSON Parser. `https://chadaustin.me/2017/05/writing-a-really-really-fast-json-parser/`, 2017.

[18] Cheng, Y., and Rusu, F. Parallel in-situ data processing with speculative loading. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 1287–1298.

[19] Cheng, Y., and Rusu, F. Scanraw: A database meta-operator for parallel in-situ processing and loading. *ACM Trans. Database Syst. 40*, 3 (Oct. 2015), 19:1–19:45.

[20] Choi, B., Chae, J., Jamshed, M., Park, K., and Han, D. Dfc: Accelerating string pattern matching for network applications. In *NSDI* (2016), pp. 551–565.

[21] Wireshark Filters. `http://www.lovemytool.com/blog/2010/04/top-10-wireshark-filters-by-chris-greer.html`.

[22] Crockford, D. The application/json media type for javascript object notation (json).

[23] Diao, Y., Altinel, M., Franklin, M. J., Zhang, H., and Fischer, P. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Transactions on Database Systems (TODS) 28*, 4 (2003), 467–516.

[24] Diao, Y., and Franklin, M. J. High-performance xml filtering: An overview of yfilter. *IEEE Data Eng. Bull. 26*, 1 (2003), 41–48.

[25] Durumeric, Z., Adrian, D., Mirian, A., Bailey, M., and Halderman, J. A. A search engine backed by internet-wide scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 542–553.

[26] Gallant, Andrew. ripgrep is faster than grep, ag, git grep, ucg, pt, sift. `https://blog.burntsushi.net/ripgrep`.

[27] TShark Tutorial and Filter Examples. `https://hackertarget.com/tshark-tutorial-and-filter-examples/`.

[28] HE, B., LUO, Q., AND CHOI, B. Cache-conscious automata for xml filtering. *IEEE Transactions on Knowledge and Data Engineering 18*, 12 (2006), 1629–1644.

[29] Analyze HTTP Requests with TShark. `http://kvz.io/blog/2010/05/15/analyze-http-requests-with-tshark/`.

[30] IDREOS, S., ALAGIANNIS, I., JOHNSON, R., AND AILAMAKI, A. Here are my data files. here are my queries. where are my results? In *Proceedings of 5th Biennial Conference on Innovative Data Systems Research* (2011), no. EPFL-CONF-161489.

[31] Jackson. `https://github.com/FasterXML/jackson`.

[32] JASTA, A. Observability at Twitter: Technical Overview, Part I. `https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-i.html`.

[33] nativejson-benchmark. `https://github.com/miloyip/nativejson-benchmark`.

[34] KAMBATLA, K., KOLLIAS, G., KUMAR, V., AND GRAMA, A. Trends in big data analytics. *Journal of Parallel and Distributed Computing 74*, 7 (2014), 2561–2573.

[35] KARPATHIOTAKIS, M., ALAGIANNIS, I., AND AILAMAKI, A. Fast queries over heterogeneous data through engine customization. *Proceedings of the VLDB Endowment 9*, 12 (2016), 972–983.

[36] KARPATHIOTAKIS, M., ALAGIANNIS, I., HEINIS, T., BRANCO, M., AND AILAMAKI, A. Just-in-time data virtualization: Lightweight data management with vida. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)* (2015).

[37] KARPATHIOTAKIS, M., BRANCO, M., ALAGIANNIS, I., AND AILAMAKI, A. Adaptive query processing on raw data. *Proceedings of the VLDB Endowment 7*, 12 (2014), 1119–1130.

[38] LI, Y., KATSIPOULAKIS, N. R., CHANDRAMOULI, B., GOLDSTEIN, J., AND KOSSMANN, D. Mison: a fast json parser for data analytics. *Proceedings of the VLDB Endowment 10*, 10 (2017), 1118–1129.

[39] libpcap. `http://www.tcpdump.org`.

[40] MA, L., AND AU, G. K.-O. Techniques for ordering predicates in column partitioned databases for query optimization, July 3 2014. US Patent App. 13/728,345.

[41] MOUSSALLI, R., HALSTEAD, R. J., SALLOUM, M., NAJJAR, W. A., AND TSOTRAS, V. J. Efficient xml path filtering using gpus. In *ADMS@ VLDB* (2011), Citeseer, pp. 9–18.

[42] MOUSSALLI, R., SALLOUM, M., NAJJAR, W., AND TSOTRAS, V. J. Massively parallel xml twig filtering using dynamic programming on fpgas. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on* (2011), IEEE, pp. 948–959.

[43] MÜHLBAUER, T., RÖDIGER, W., SEILBECK, R., REISER, A., KEMPER, A., AND NEUMANN, T. Instant loading for main memory databases. *Proceedings of the VLDB Endowment 6*, 14 (2013), 1702–1713.

[44] ARM NEON. `https://developer.arm.com/technologies/neon`.

[45] NORTON, M. Optimizing pattern matching for intrusion detection. *Sourcefire, Inc., Columbia, MD* (2004).

[46] OLMA, M., KARPATHIOTAKIS, M., ALAGIANNIS, I., ATHANASSOULIS, M., AND AILAMAKI, A. Slalom: Coasting through raw data via adaptive partitioning and indexing. *Proceedings of the VLDB Endowment 10*, 10 (2017), 1106–1117.

[47] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., CHUN, B.-G., AND ICSI, V. Making sense of performance in data analytics frameworks. In *NSDI* (2015), vol. 15, pp. 293–307.

[48] Apache Parquet. `https://parquet.apache.org`.

[49] apache/parquet-format. `https://github.com/apache/parquet-format`.

[50] Development/LibpcapFileFormat - The Wireshark Wiki. `https://wiki.wireshark.org/Development/LibpcapFileFormat`.

[51] Libpcap File Format. `https://wiki.wireshark.org/Development/LibpcapFileFormat`.

[52] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1816–1827.

[53] RapidJSON. `https://rapidjson.org`.

[54] SAGIROGLU, S., AND SINANC, D. Big data: A review. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on* (2013), IEEE, pp. 42–47.

[55] SCHEUFELE, W., AND MOERKOTTE, G. *Optimal ordering of selections and joins in acyclic queries with expensive predicates*. RWTH, Fachgruppe Informatik, 1996.

[56] Spark SQL Data Sources API: Unified Data Access for the Apache Spark Platform. `https://databricks.com/blog/2015/01/09/`.

[57] STYLIANOPOULOS, C., ALMGREN, M., LANDSIEDEL, O., AND PAPATRIANTAFILOU, M. Multiple pattern matching for network security applications: Acceleration through vectorization. In *Parallel Processing (ICPP), 2017 46th International Conference on* (2017), IEEE, pp. 472–482.

[58] TAHARA, D., DIAMOND, T., AND ABADI, D. J. Sinew: a SQL system for multi-structured data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 815–826.

[59] tcpdump. `http://www.tcpdump.org`.

[60] TEUBNER, J., WOODS, L., AND NIE, C. XLynx: an FPGA-based XML filter for hybrid XQuery processing. *ACM Transactions on Database Systems (TODS) 38*, 4 (2013), 23.

[61] TShark. `https://www.wireshark.org/docs/man-pages/tshark.html`.

[62] TUCK, N., SHERWOOD, T., CALDER, B., AND VARGHESE, G. Deterministic memory-efficient string matching algorithms for intrusion detection. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies* (2004), vol. 4, IEEE, pp. 2628–2639.

[63] Introduction to Twitter JSON. `https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json`.

[64] VIOLA, P., AND JONES, M. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the*

2001 IEEE Computer Society Conference on (2001), vol. 1, IEEE, pp. I–I.