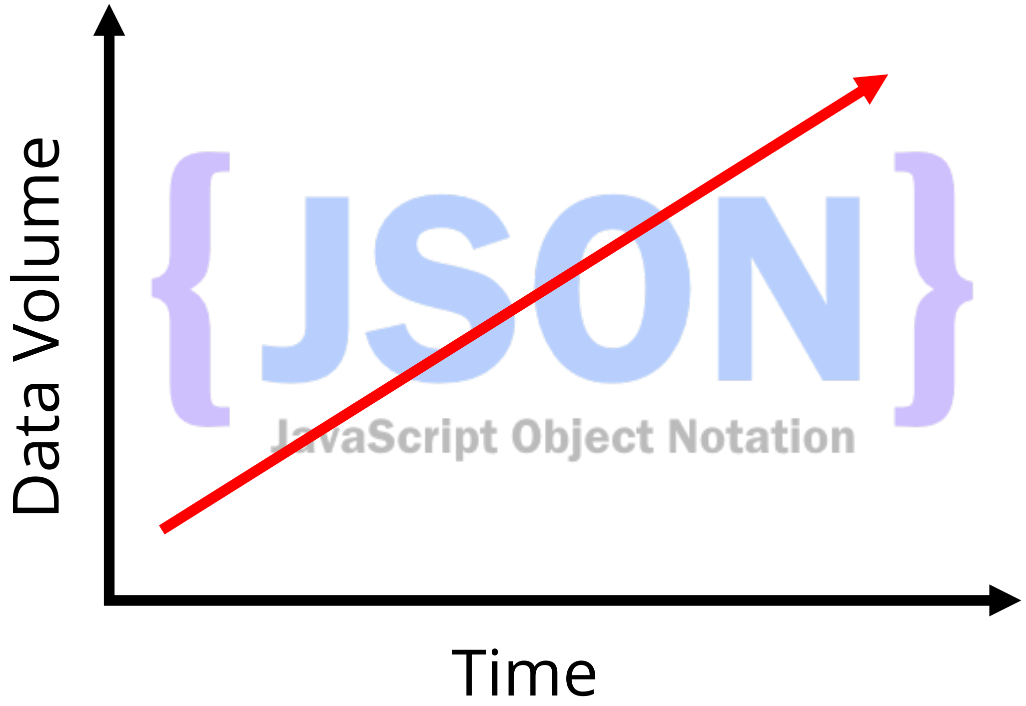
Sparser: Fast Analytics on Raw Data by Avoiding Parsing

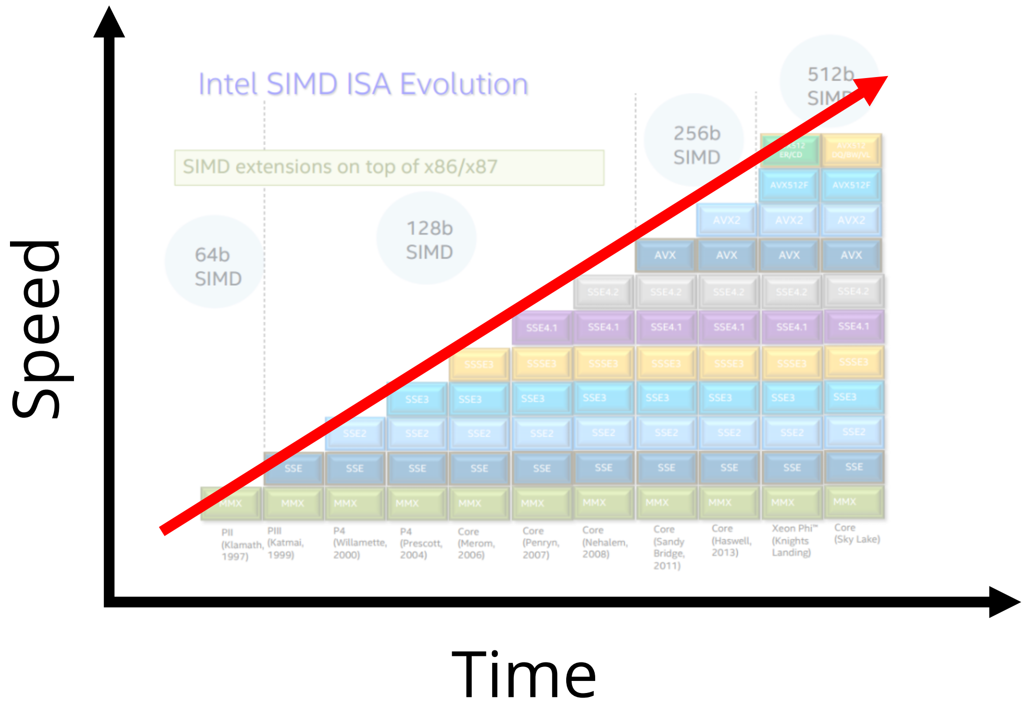
Shoumik Palkar, **Firas Abuzaid**, Peter Bailis, Matei Zaharia

STANFORD DAWN

Motivation

Bigger unstructured datasets and **faster** hardware.





Parsing unstructured data before querying it is often very slow.

Today: Spark Data Sources API

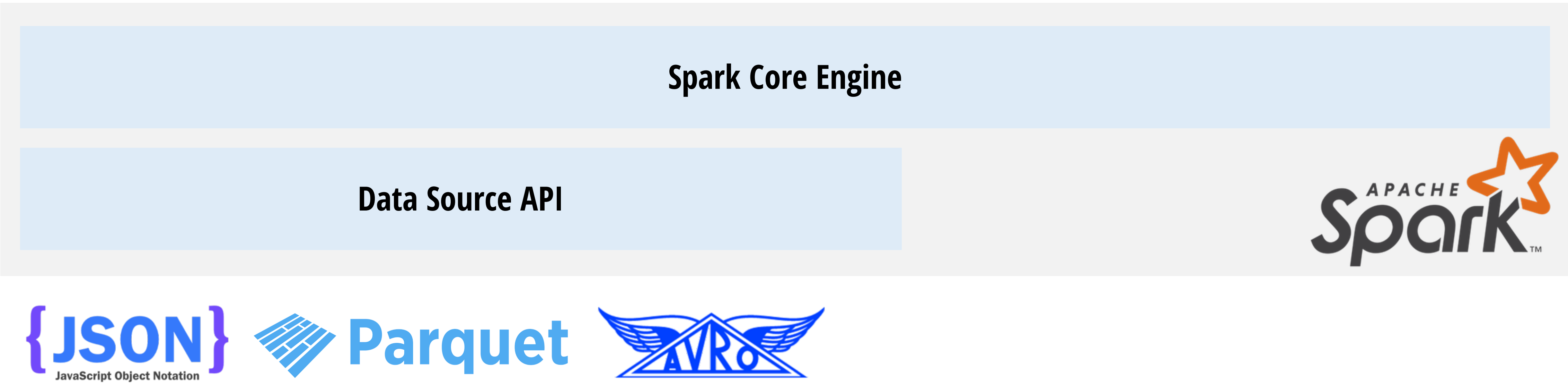
Push part of query into the data source.

Spark Core Engine

Today: Spark Data Sources API

Push part of query into the data source.

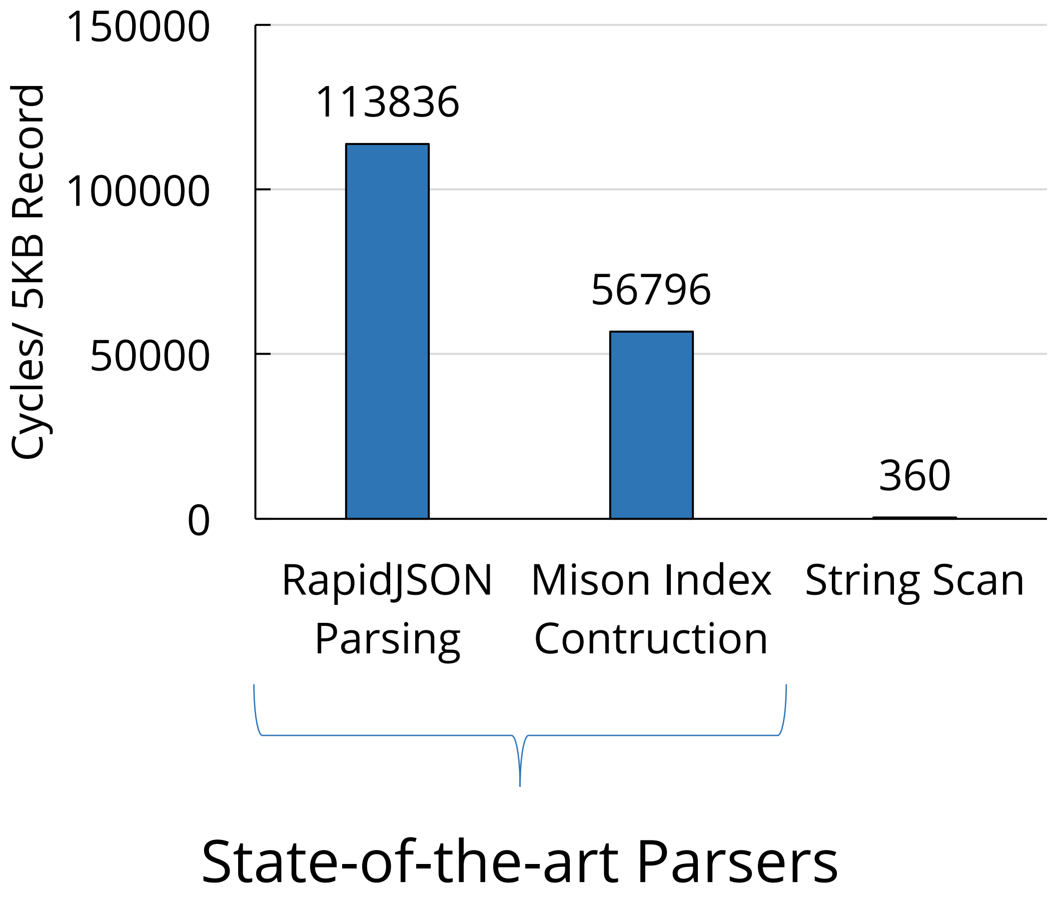
- + E.g., column pruning directly in Parquet data loader
- - Little support for unstructured formats (e.g., can't avoid JSON parsing)



Parsing: A Computational Bottleneck

Example: Existing state-of-the-art JSON parsers **100x slower** than scanning a string!

*Similar results on binary formats like Avro and Parquet

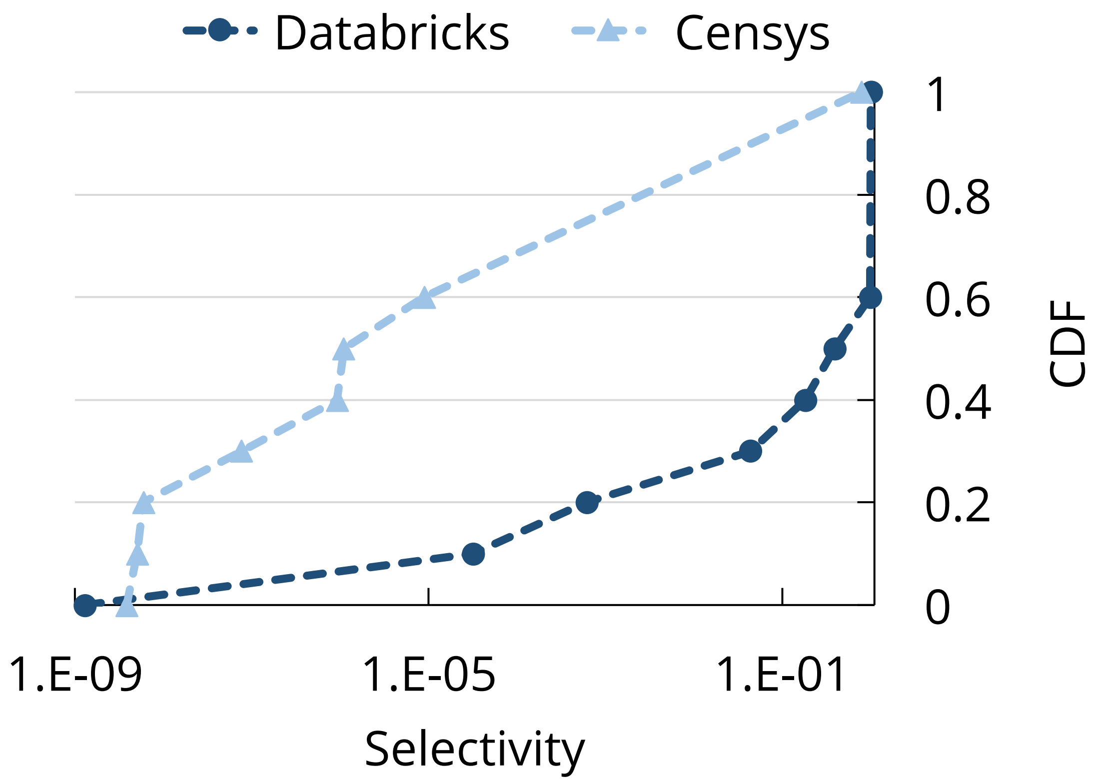


Parsing seems to be a necessary evil: how do we get around doing it?

Key Opportunity: High Selectivity

High selectivity especially true for **exploratory analytics.**

Data Source API provides access to query filters at data source!

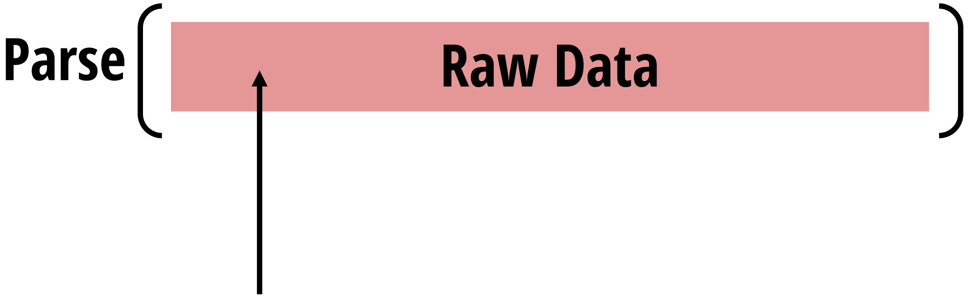


40% of customer Spark queries at Databricks **select < 20%** of data
99% of queries in Censys **select < 0.001%** of data

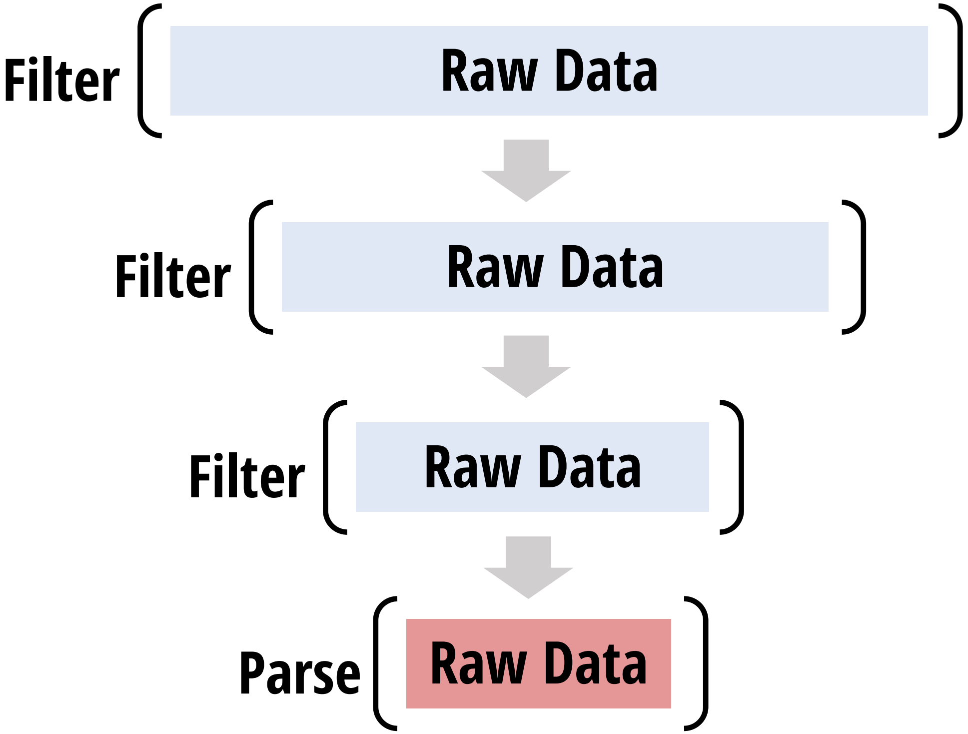
How can we exploit high selectivity to accelerate parsing?

Sparser: Filter Before You Parse

Sparser: **Filter before parsing** first using fast filtering functions with **false positives, but no false negatives**



Today:
parse full input → slow!

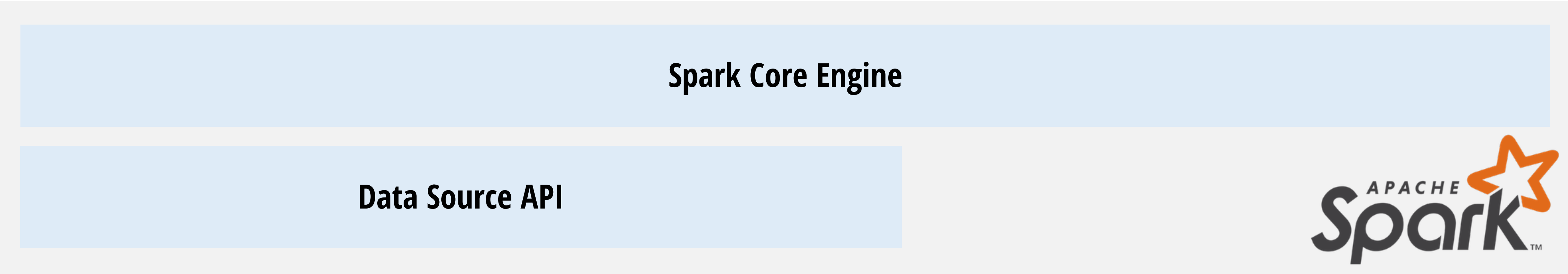


Demo

Count Tweets where text contains “Trump” and “Putin”

Sparser in Spark SQL

```
val df = spark.read.format("json")
```



Sparser in Spark SQL

```
val df = spark.read.format("edu.stanford.sparser.json")
```

Sparser Data Source Reader
(Also supports Avro, Parquet!)

Spark Core Engine

Data Source API

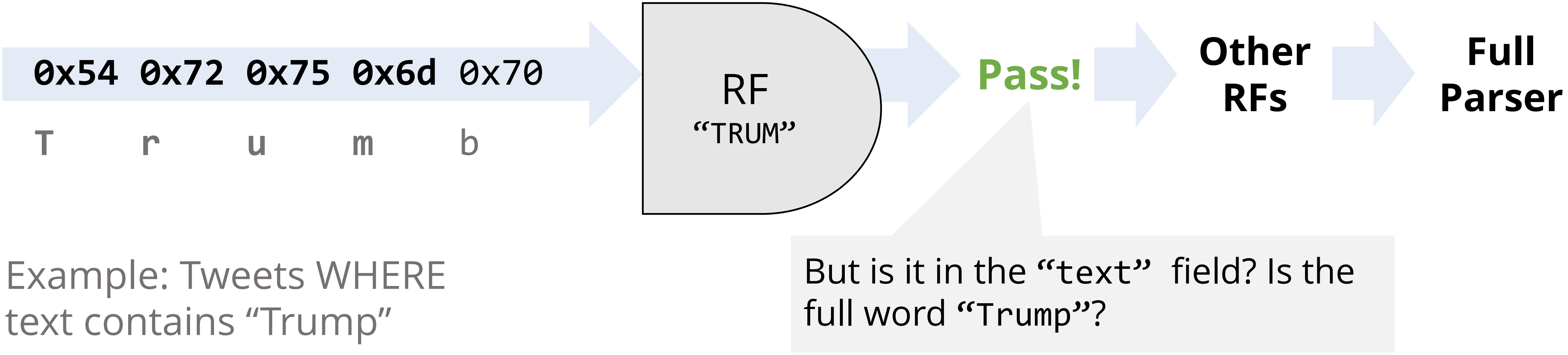
Sparser Filtering Engine

Sparser Overview

Sparser Overview

Raw Filter ("RF"): filtering function on a bytestream with **false positives** but **no false negatives.**

Use an **optimizer** to combine RFs into a **cascade.**



Example: Tweets WHERE text contains "Trump"

Key Challenges in Using RFs

1. How do we implement the RFs efficiently?
2. How do we efficiently choose the RFs to maximize parsing throughput for a given query and dataset?

Rest of this Talk

- Sparser’s API
- Sparser’s Raw Filter Designs **(Challenge 1)**
- Sparser’s Optimizer: Choosing a Cascade **(Challenge 2)**
- Performance Results on various file formats (e.g., JSON, Avro)

Sparser API

Sparser API

Filter	Example
Exact String Match	`WHERE user.name = “Trump”` `WHERE likes = 5000`

Sparser API

Filter	Example
Exact String Match	`WHERE user.name = “Trump”` `WHERE likes = 5000`
Contains String	`WHERE text contains “Trum”`

Sparser API

Filter	Example
Exact String Match	`WHERE user.name = "Trump"` `WHERE likes = 5000`
Contains String	`WHERE text contains "Trum"`
Contains Key	`WHERE user.url != NULL`

Sparser API

Filter	Example
Exact String Match	`WHERE user.name = "Trump"` `WHERE likes = 5000`
Contains String	`WHERE text contains "Trum"`
Contains Key	`WHERE user.url != NULL`
Conjunctions	`WHERE user.name = "Trump"` `AND user.verified = true`
Disjunctions	`WHERE user.name = "Trump"` `OR user.name = "Obama"`

Sparser API

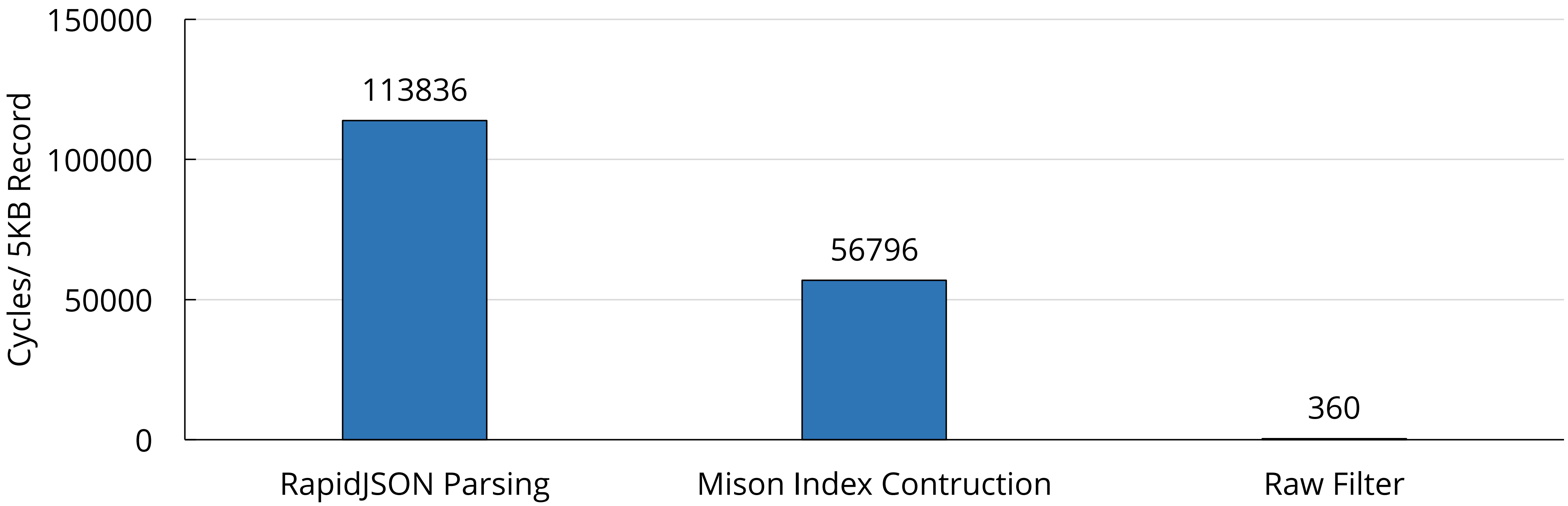
Filter	Example
Exact String Match	`WHERE user.name = “Trump”` `WHERE likes = 5000`
Contains String	`WHERE text contains “Trum”`
Contains Key	`WHERE user.url != NULL`
Conjunctions	`WHERE user.name = “Trump”` `AND user.verified = true`
Disjunctions	`WHERE user.name = “Trump”` `OR user.name = “Obama”`

Currently does not support numerical range-based predicates.

Challenge 1: Efficient RFs

Raw Filters in Sparser

SIMD-based filtering functions that pass or discard a record by inspecting raw bytestream



Example RF: Substring Search

Search for a small (e.g., 2, 4, or 8-byte) substring of a query predicate in parallel **using SIMD**

Example query: text contains “Trump”

```
Input   : “text”:“I just met Mr. Trumb!!!”
Shift 1 : TrumTrumTrumTrumTrumTrumTrum-------
Shift 2 : -TrumTrumTrumTrumTrumTrumTrum------
Shift 3 : --TrumTrumTrumTrumTrumTrumTrum-----
Shift 4 : ---TrumTrumTrumTrumTrumTrumTrum----
```

Length 4 Substring packed in SIMD register

False positives (found “Trumb” by accident),
but **no false negatives** (No “Trum” ⇒ No “Trump”)

On modern CPUs: compare 32 characters in parallel in ~4 cycles (2ns).

Other RFs also possible! Sparser selects them agnostic of implementation.

Key-Value Search RF

Searches for key, and if key is found, searches for value until some stopping point. Searches occur with SIMD.

- Only applicable for **exact matches**
- Useful for queries with common substrings (e.g., favorited=true)

```
Key: name Value: Trump Delimiter: ,
Ex 1: “name”: “Trump”,                    (Pass)
Ex 2: “name”: “Actually, Trump”           (Fail)
Ex 3: “name”: “My name is Trump”          (Pass, False positive)
```

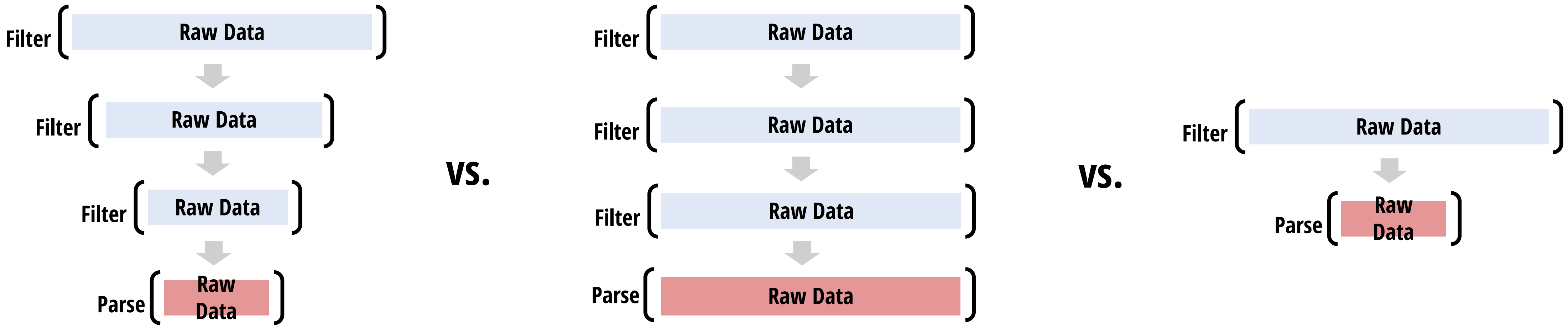
Second Example would result in **false negative** if we allow substring matches

Other RFs also possible! Sparser selects them agnostic of implementation.

Challenge 2: Choosing RFs

Choosing RFs

To decrease **false positive rate**, combine RFs into a cascade



Sparser uses an **optimizer** to choose a cascade

Sparser's Optimizer

`(name = "Trump" AND text contains "Putin")`

raw bytestream
0010101000110101

Step 1: Compile Possible RFs from predicates

RF1: "Trump"
RF2: "Trum"
RF3: "rump"
RF4: "Tr"
RF5: "Putin"
…

Step 2: Measure Params on Sample of Records

	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

C (RF1) = 4
C (RF1→RF2) = 1
C (RF2→RF3) = 6
C (RF1→RF3) = 9
…

Step 4: Apply Chosen Cascade

RF 1 → X
RF fails

RF 2 → X
RF fails

Filtered bytes sent to full parser

Sparser's Optimizer: Configuring RF Cascades

Three high level steps:

1. **Convert** a query predicate to a set of RFs
2. **Measure** the passthrough rate and runtime of each RF and the runtime of the parser on sample of records
3. **Minimize** optimization function to find min-cost cascade

1. Converting Predicates to RFs

Running Example

`(name = “Trump” AND text contains “Putin”)`

`OR name = “Obama”`

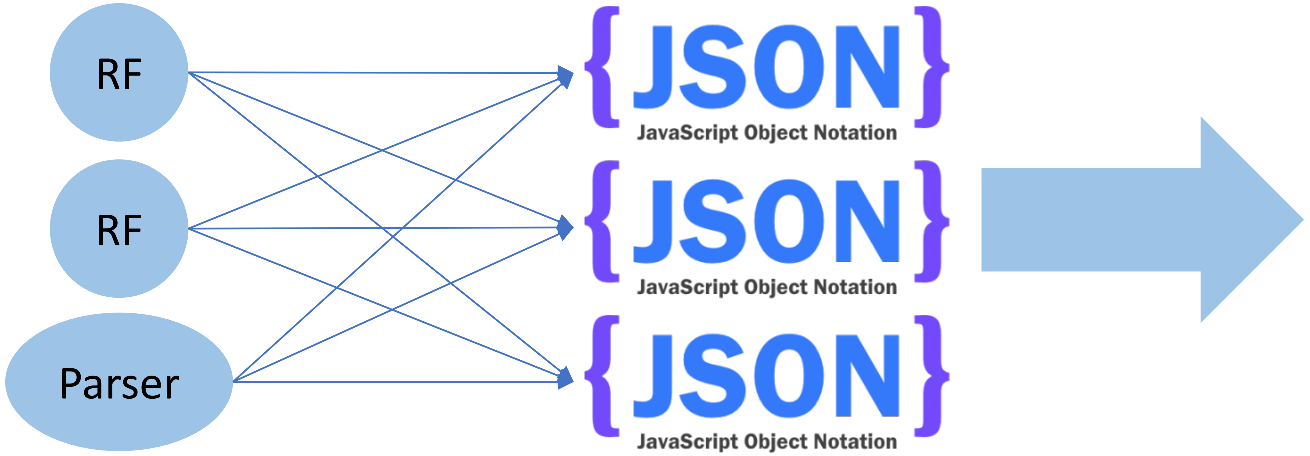
Possible RFs*

Substring Search: `Trum, Tr, ru, …, Puti, utin, Pu, ut, …, Obam, Ob, …`

* Translation from predicate to RF is format-dependent: examples are for JSON data

2. Measure RFs and Parser

Evaluate RFs and parser on a sample of records.



Passthrough rates* and runtimes

*Passthrough rates of RFs are **non-independent!** But too expensive to measure rate of each cascade (combinatorial search space)

2. Measure RFs and Parser

Passthrough rates of RFs are **non-independent!** But too expensive to rate of each cascade (combinatorial search space)

Solution: Store rates as **bitmaps.**

```
foreach sampled record i:
   foreach RF j:
      BitMap[i][j] = 1 if RF j passes on i
```

Pr[a] $\propto$ **1s** in Bitmap of RF a: 00**11**0**1**0**1**0**1**

Pr[b] $\propto$ **1s** in Bitmap of RF b: 000**11**0**11**0**1**

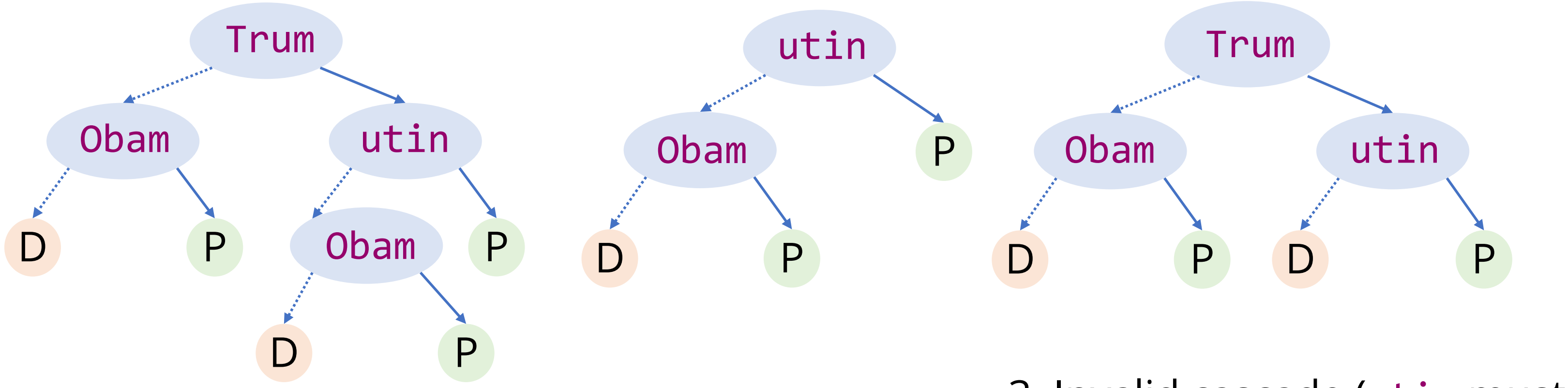
Pr[a,b] $\propto$ **1s** in **Bitwise-&** of a, b: 000**1**000**1**0**1**

3. Choosing Best Cascade

Running Example

`(name = "Trump" AND text contains "Putin")`

`OR name = "Obama"`



1. Valid cascade

2. Valid cascade

3. Invalid cascade (`utin` must consider `Obam` when it fails)

Discard D   Parse P   Fail (left branch)   Pass (right branch)

Cascade cost computed **by traversing tree from root to leaf**.

Each RF has a runtime/passthrough rate we measured in previous step.

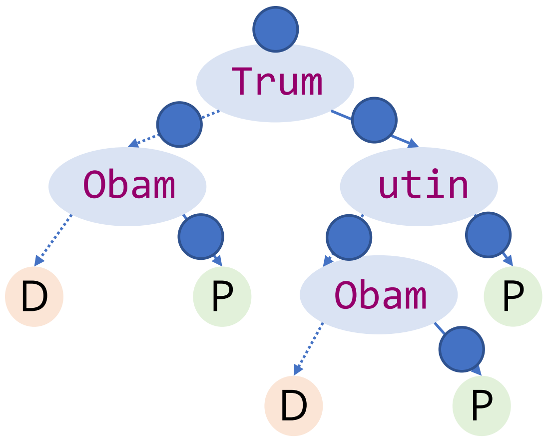
3. Choosing Best Cascade

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

Cost of Cascade with RFs 0,1,…,R

Probability and cost of executing RF *i*

Probability and cost of executing full parser



$$\Pr[execute_{Trum}] = 1$$

$$\Pr[execute_{utin}] = \Pr[\text{Trum}]$$

$$\Pr[execute_{Obam}] = \Pr[\neg\text{Trum}] + \Pr[\text{Trum}, \neg\text{utin}]$$

$$\Pr\left[execute_{parse}\right] = \Pr[\neg\text{Trum}, \text{Obam}] + \Pr[\text{Trum}, \text{utin}] + \Pr[\text{Trum}, \neg\text{utin}, \text{Obam}]$$

Choose cascade with minimum cost: Sparser considers up to min(32, # ORs) RFs and considers up to depth of 4 cascades.

Spark Integration

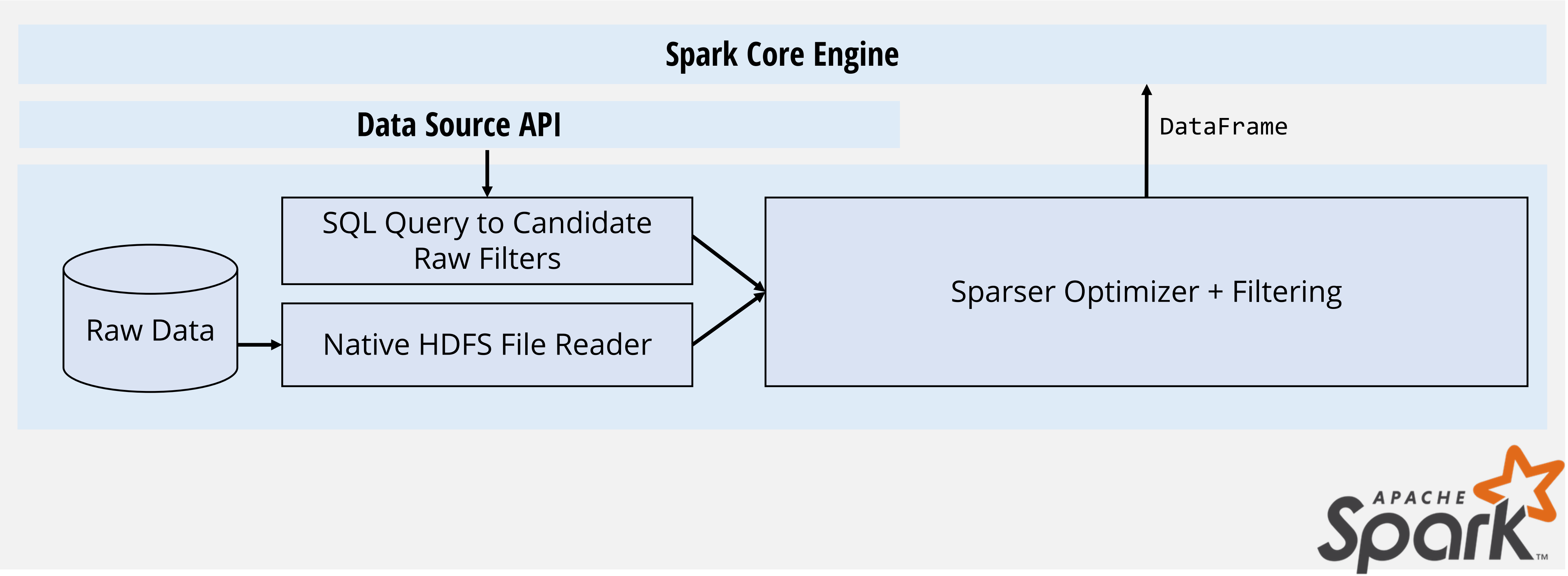
Sparser in Spark SQL

Spark Core Engine

Data Source API

Sparser Filtering Engine

Sparser in Spark SQL



Performance Results

Evaluation Setup

Datasets

Censys - Internet port scan data used in network security
Tweets - collected using Twitter Stream API

Distributed experiments

on 20-node cluster with 4 Broadwell vCPUs/26GB of memory, locally attached SSDs (Google Compute Engine)

Single node experiments

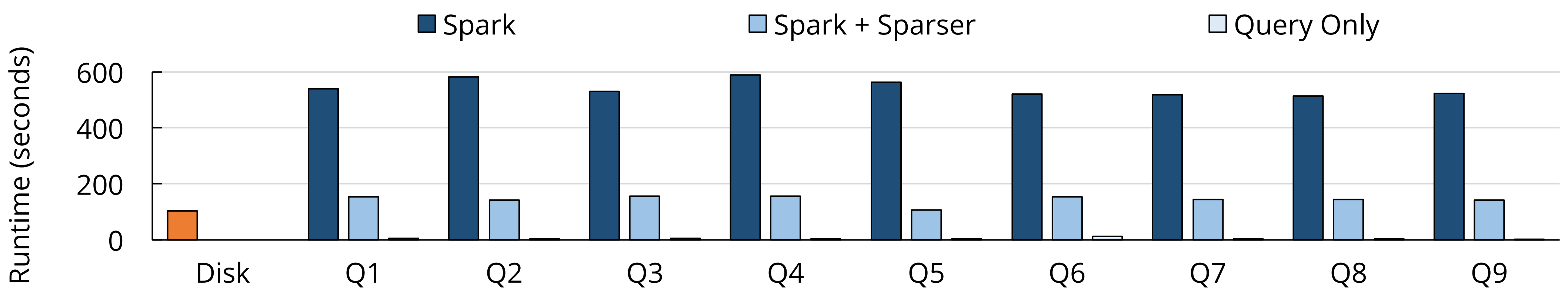
on Intel Xeon E5-2690 v4 with 512GB memory

Queries

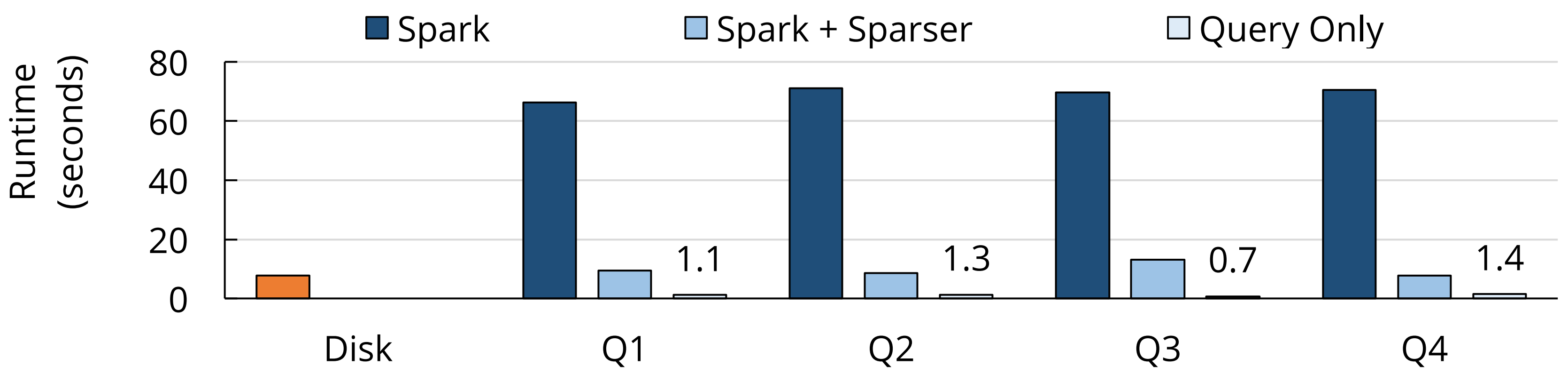
Query Name	Query	Selectivity (%)
Twitter 1	COUNT(*)WHERE text LIKE '%Donald Trump%'AND date LIKE '%Sep 13%'	0.1324
Twitter 2	user.id, SUM(retweet_count)WHERE text LIKE '%Obama%'GROUP BY user.id	0.2855
Twitter 3	id WHERE user.lang == 'msa'	0.0020
Twitter 4	distinct user.id WHERE text LIKE '%@realDonaldTrump%'	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

Twitter queries from other academic work. Censys queries sampled randomly from top-3000 queries. PCAP and Bro queries from online tutorials/blogs.

Results: Accelerating End-to-End Spark Jobs

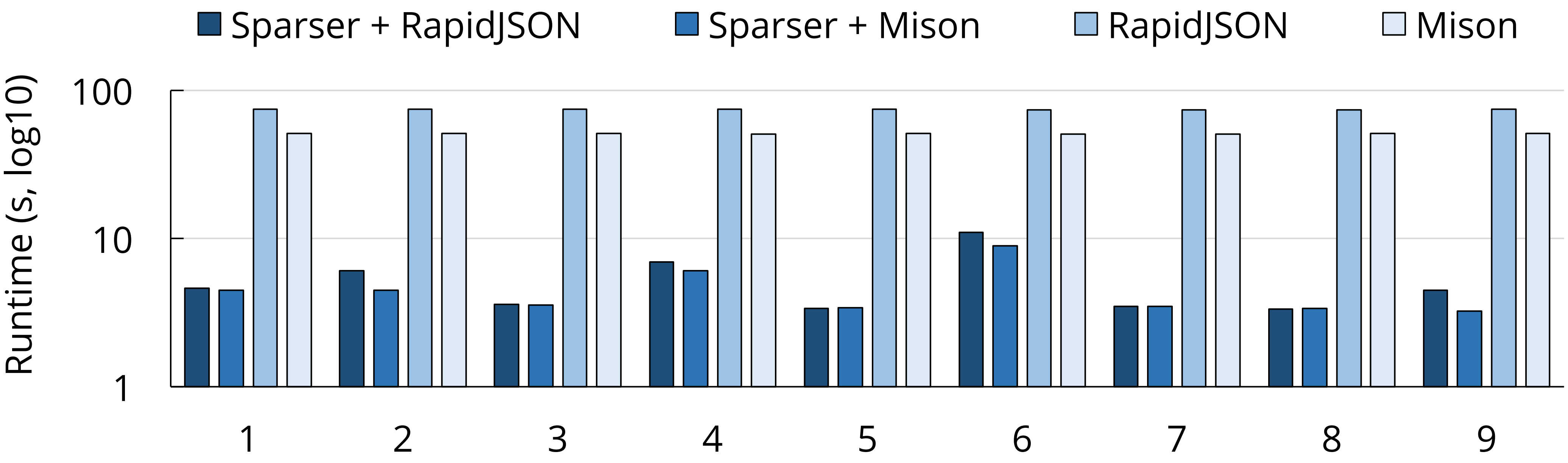


Censys queries on 652GB of JSON data: up to **4x speedup** by using Sparser.



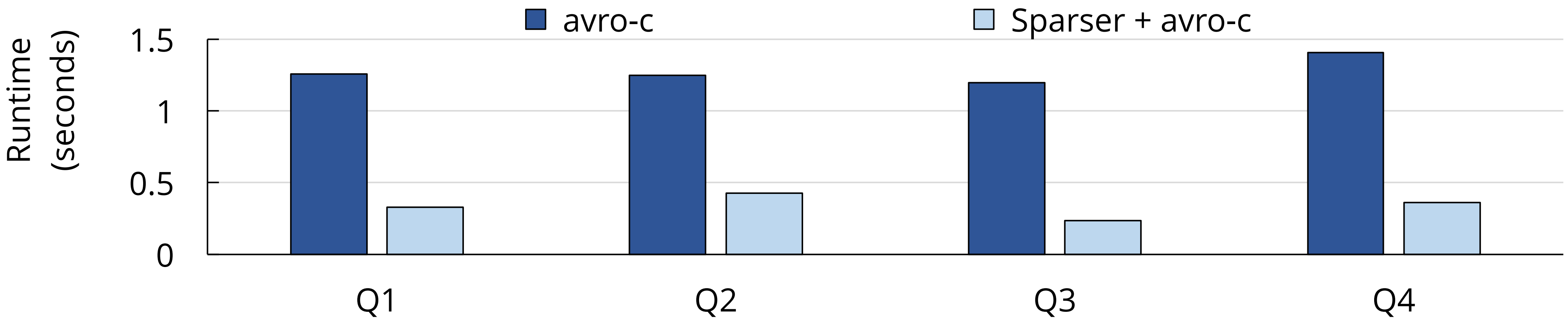
Twitter queries on 68GB of JSON data: up to **9x speedup** by using Sparser.

Results: Accelerating Existing JSON Parsers

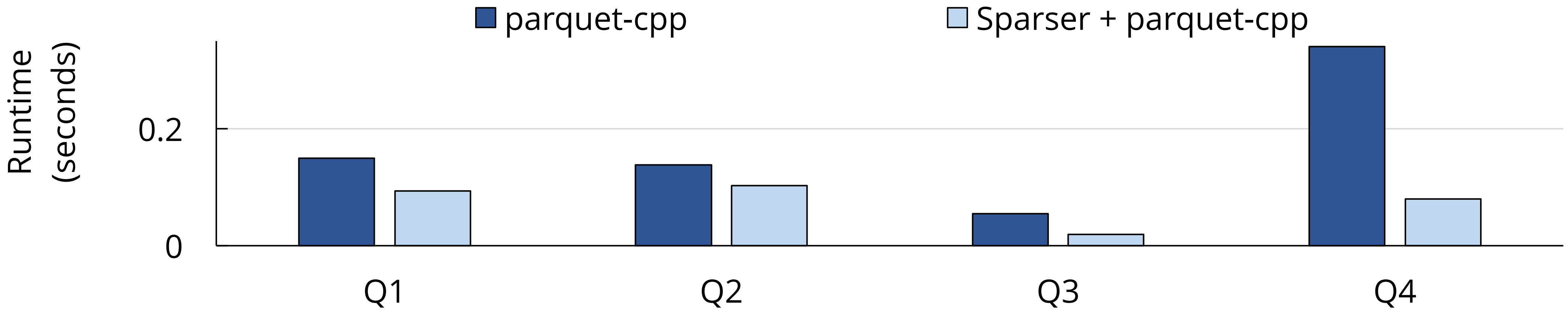


Censys queries compared against two state-of-the-art parsers (Mison based on SIMD): Sparser accelerates them by up to **22x**.

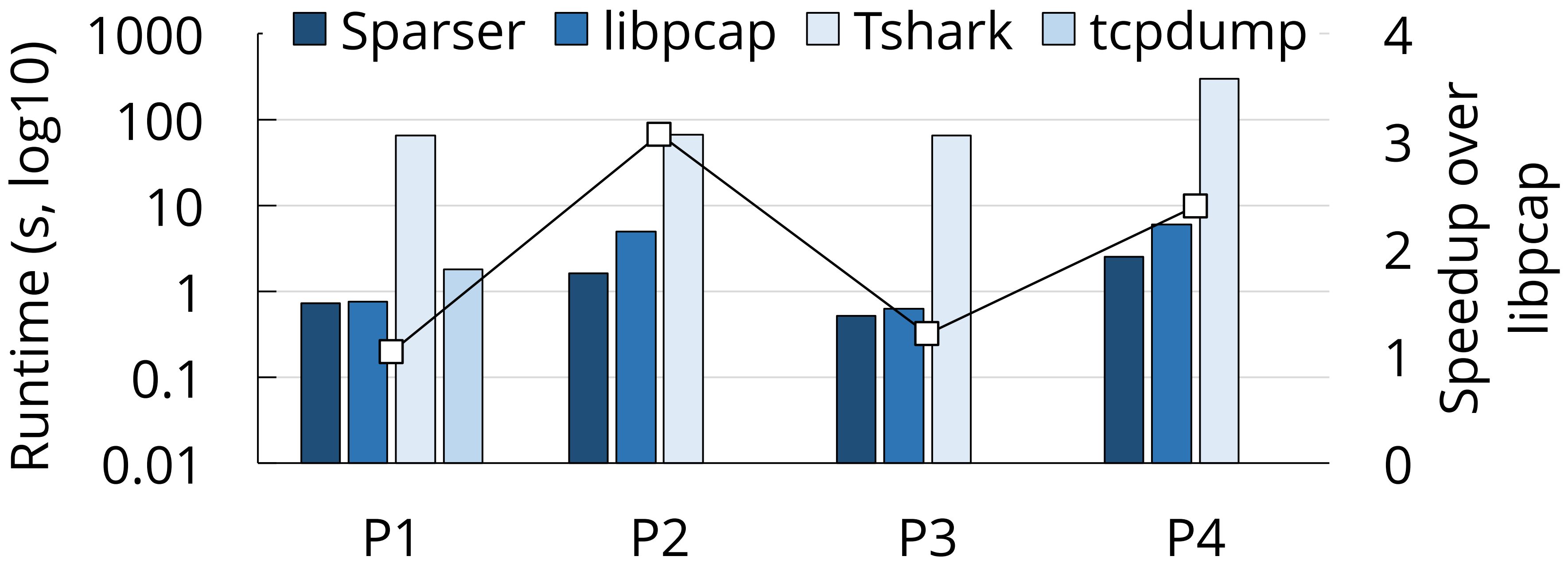
Results: Accelerating Binary Format Parsing



Sparser accelerates **Avro** (above) and **Parquet** (below) based queries by up to **5x** and **4.3x** respectively.

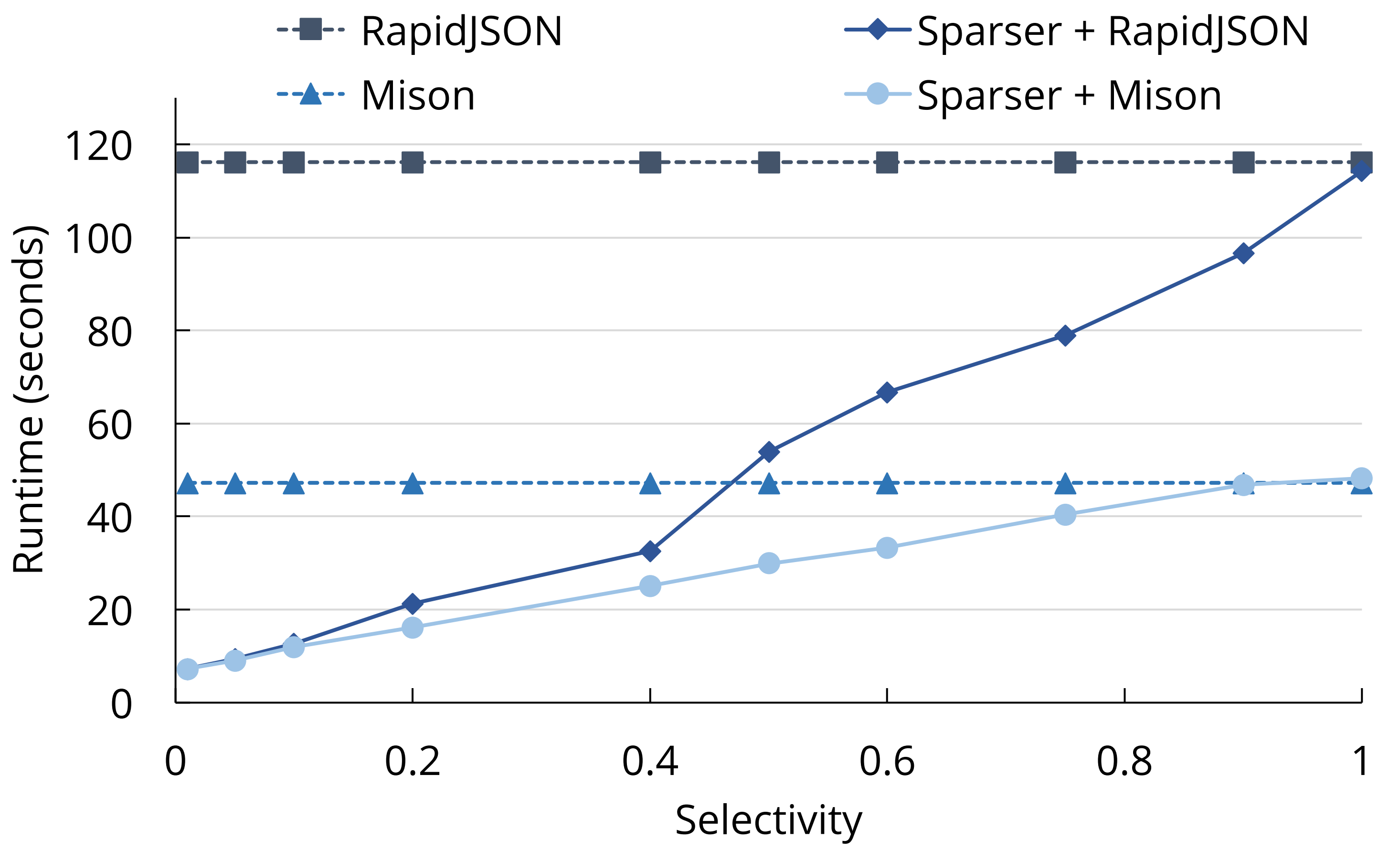


Results: Domain-Specific Tasks



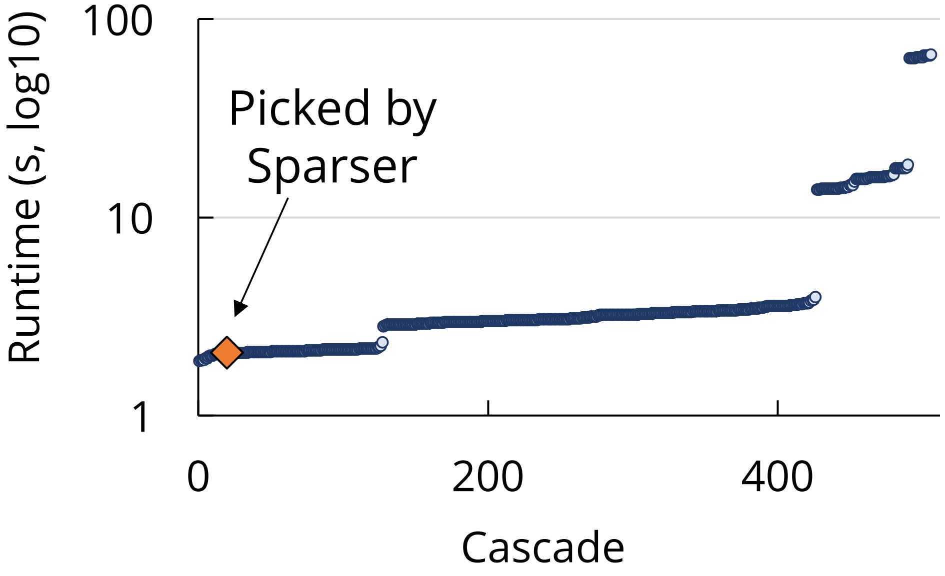
Sparser accelerates packet parsing by up to **3.5x** compared to standard tools.

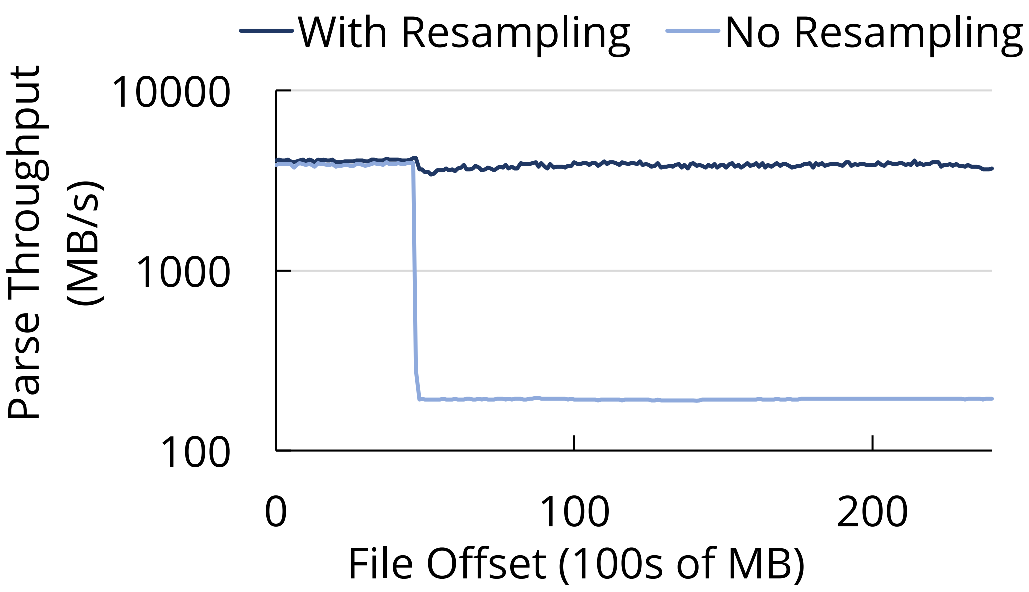
Results: Sparser's Sensitivity to Selectivity



Sparser's **performance degrades gracefully** as the selectivity of the query decreases.

Results: Nuts and Bolts





Sparser picks a cascade **within 5% of the optimal** one using its sampling-based measurement and optimizer.

Resampling to calibrate the cascade can improve end-to-end parsing time by up to **20x**.

Conclusion

Sparser:

- Uses **raw filters** to filter before parsing
- Selects a **cascade** of raw filters using an **efficient optimizer**
- Delivers up to **22x** speedups over existing parsers
- Will be **open source** soon!

Questions or Comments? Contact Us!

shoumik@cs.stanford.edu
fabuzaid@cs.stanford.edu

STANFORD DAWN