Characterizing Java Streams in the Wild

Eduardo Rosales, Matteo Basso, Andrea Rosà, Walter Binder

Universitá della Svizzera italiana (USI)

Alex Villazón, Adriana Orellana, Ángel Zenteno, Jhon Rivero

Universidad Privada Boliviana (UPB)

ICECCS 2022
March 30, 2022

Università
della
Svizzera
italiana

JAVA STREAMS

Università della Svizzera italiana

- **Stream API** (package java.util.stream)
  - Introduced since Java 8
  - Data processing
    - Functional programming
      - MapReduce style transformations
  - Two key abstractions:
    - **Stream**
    - **Stream pipeline**

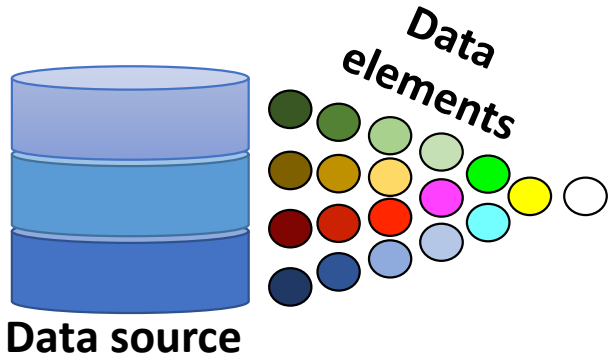
Università della Svizzera italiana

Stream – Java Code Example

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

Stream

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```



- A **stream** represents a sequence of **data elements** that comes from a **data source**

Data Source

```
transactionList.stream()
    .parallel()
    .filter(t -> t.getStatus() == Transaction.VALID)
    .map(Transaction::getID)
    .collect(Collectors.toSet());
```

Data source

Data source

`List`: transactionList

transactionList is a Collection

- Other data sources from which a stream can be created:
  - Arrays
  - Files
  - Strings
  - …

Source Collection Type

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

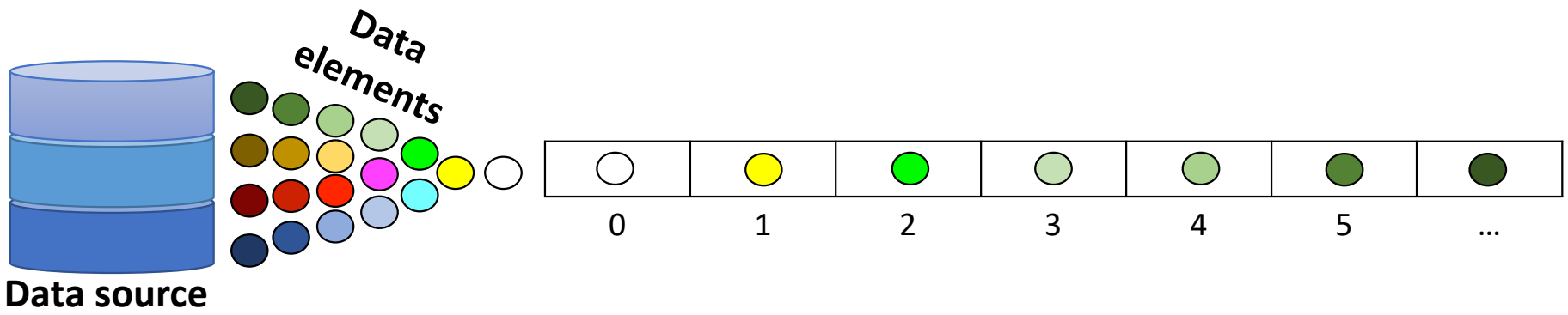
Data source

`List`:
transactionList

Source collection type:
e.g., `java.util.ArrayList`

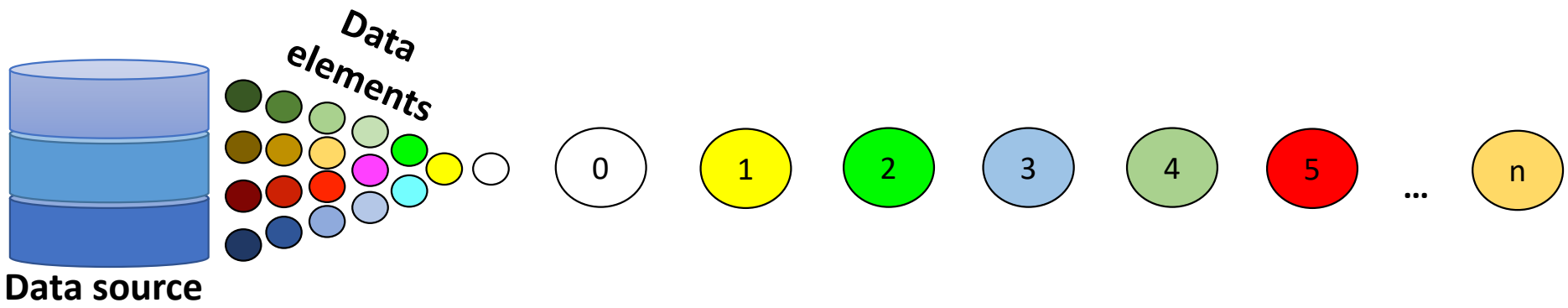
Characteristics of the Data Source - Encounter Order

- **Encounter Order:** whether the data source makes its elements available in a defined order (i.e., index order)



Characteristics of the Data Source - Sort Order

- **Sort Order:** whether the data source makes its elements available in a sort order



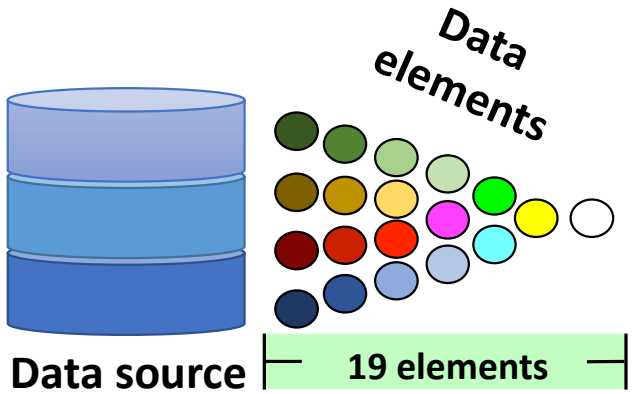
Università della Svizzera italiana

Characteristics of the Data Source

- **Concurrent:** whether the data source allows concurrent modification
- **Distinct:** whether the data source allows duplicates
- **Immutable:** whether the elements in the data source can be modified

Characteristics of the Data Source – Data-Source Size

- **Data-source size:** the total number of elements in the data source



Università della Svizzera italiana

Stream Types

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

Stream type:
`java.util.stream.Stream`

- Other stream types include:
  - `java.util.stream.IntStream`
  - `java.util.stream.LongStream`
  - `java.util.stream.DoubleStream`

Source Method

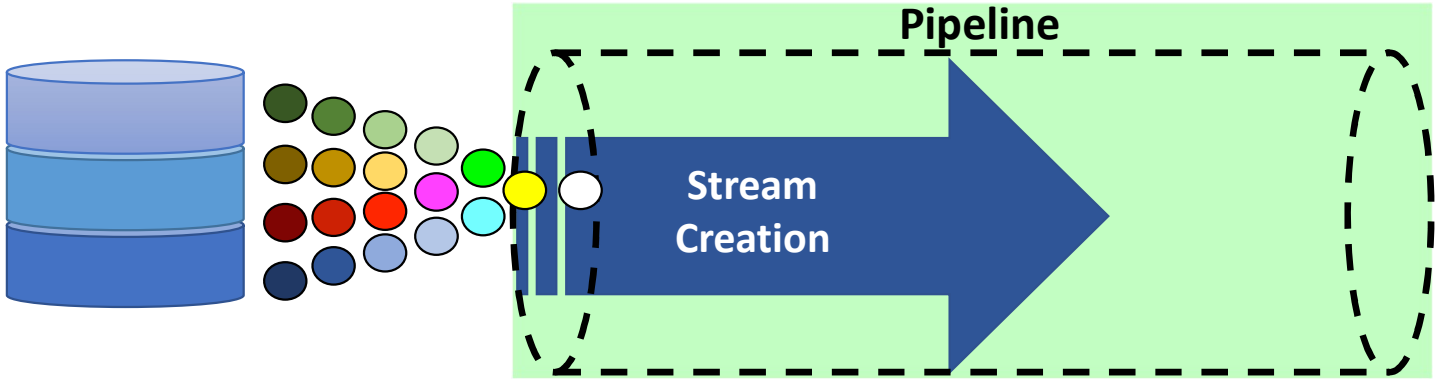
Source method:
`java.util.Collection.stream`

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

Pipeline

```
transactionList.stream()
    .parallel()
    .filter(t -> t.getStatus() == Transaction.VALID)
    .map(Transaction::getID)
    .collect(Collectors.toSet());
```

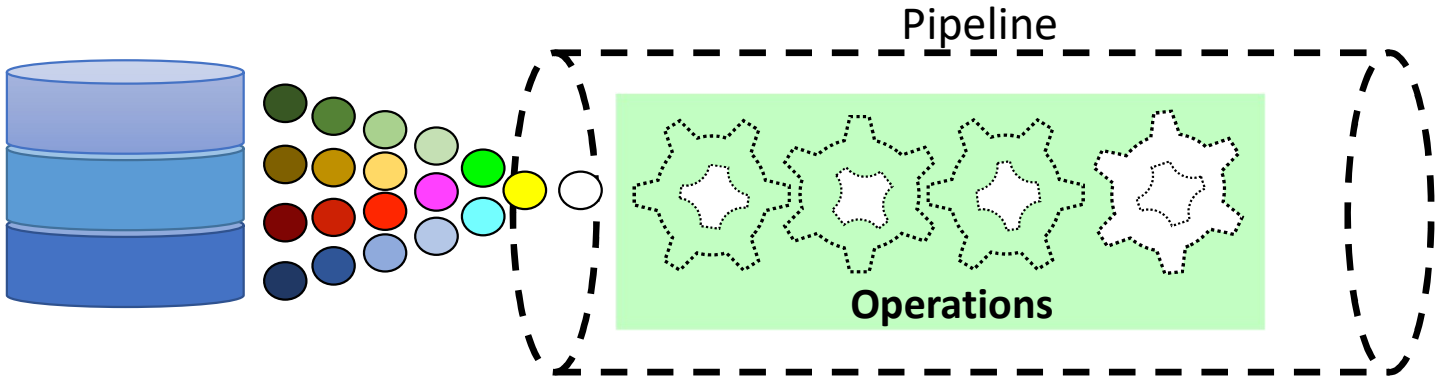
Stream creation



Operations

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

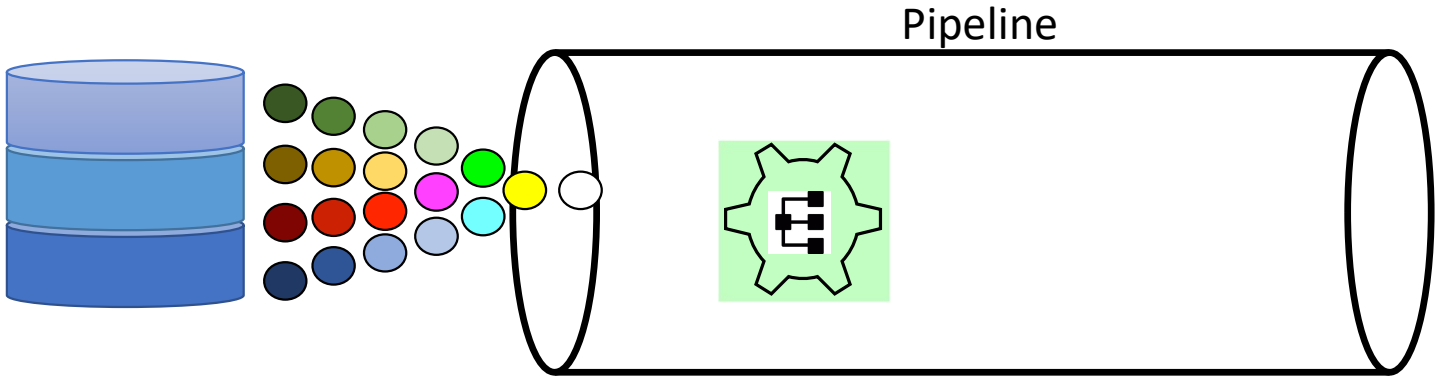
- The stream pipeline contains **operations**



Operation Appending

```
transactionList.stream()
    .parallel()
    .filter(t -> t.getStatus() == Transaction.VALID)
    .map(Transaction::getID)
    .collect(Collectors.toSet());
```

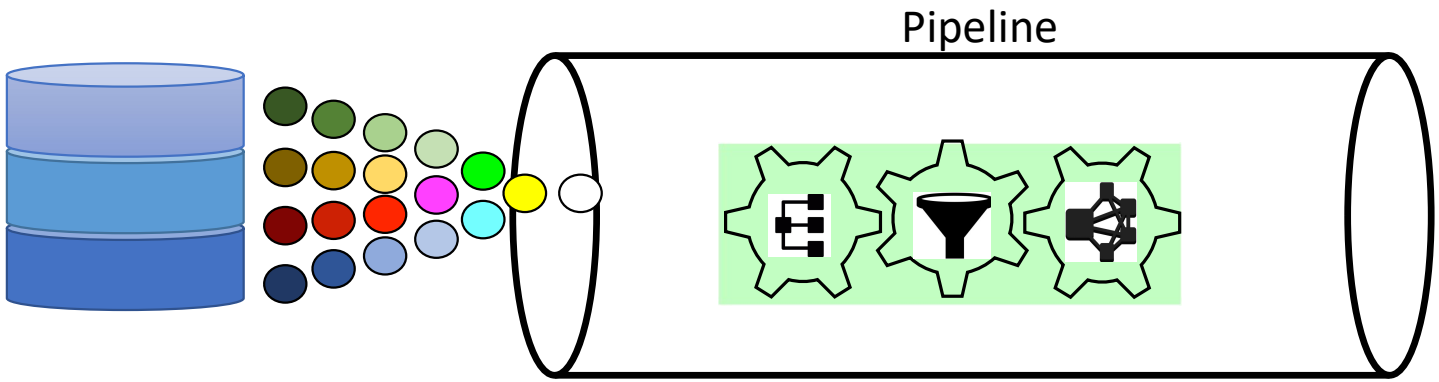
The operation is **appended** to the pipeline



Intermediate Operations

```
transactionList.stream()
      .parallel()
      .filter(t -> t.getStatus() == Transaction.VALID)
      .map(Transaction::getID)
      .collect(Collectors.toSet());
```

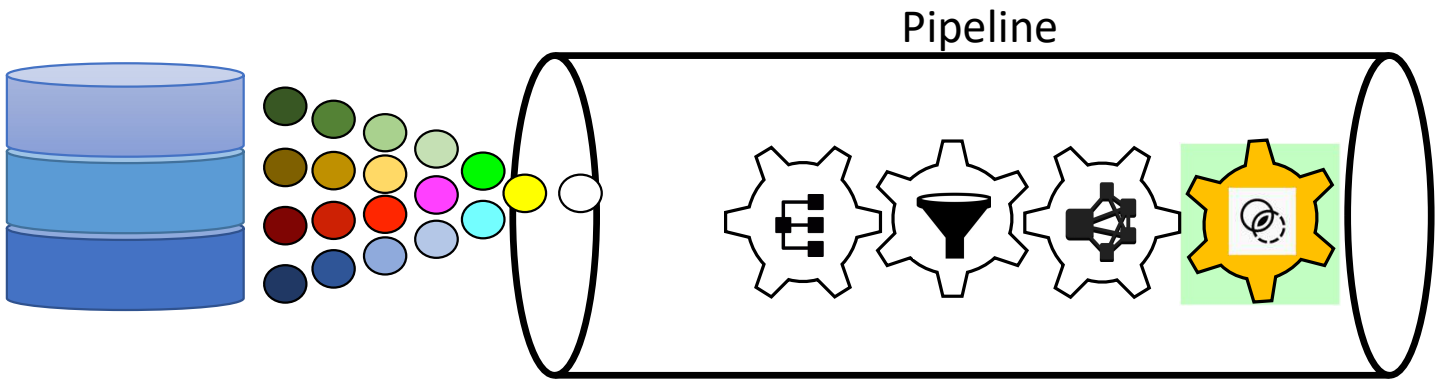
Intermediate operations



Terminal Operation

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

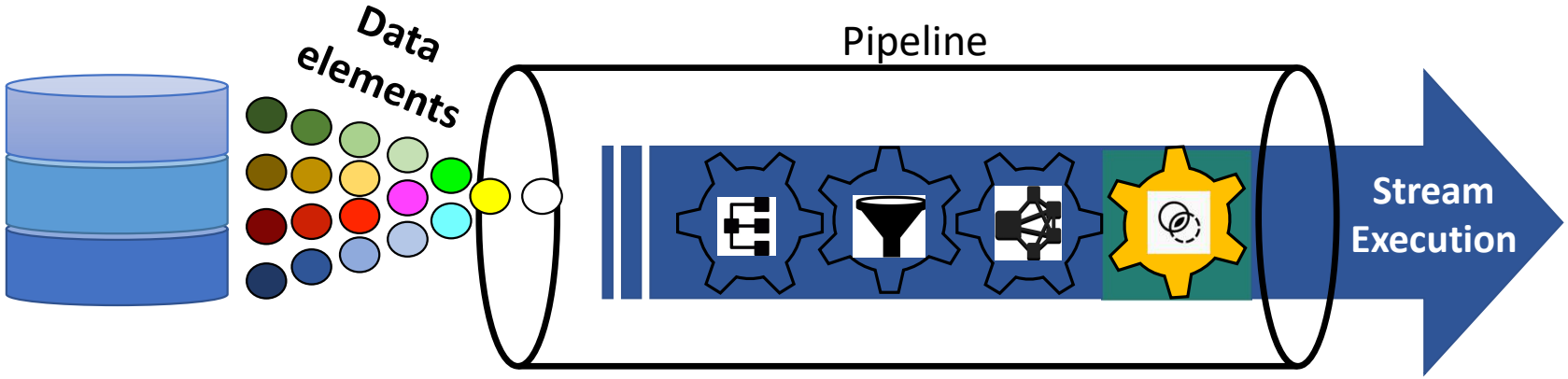
Terminal operation



Terminal Operation

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

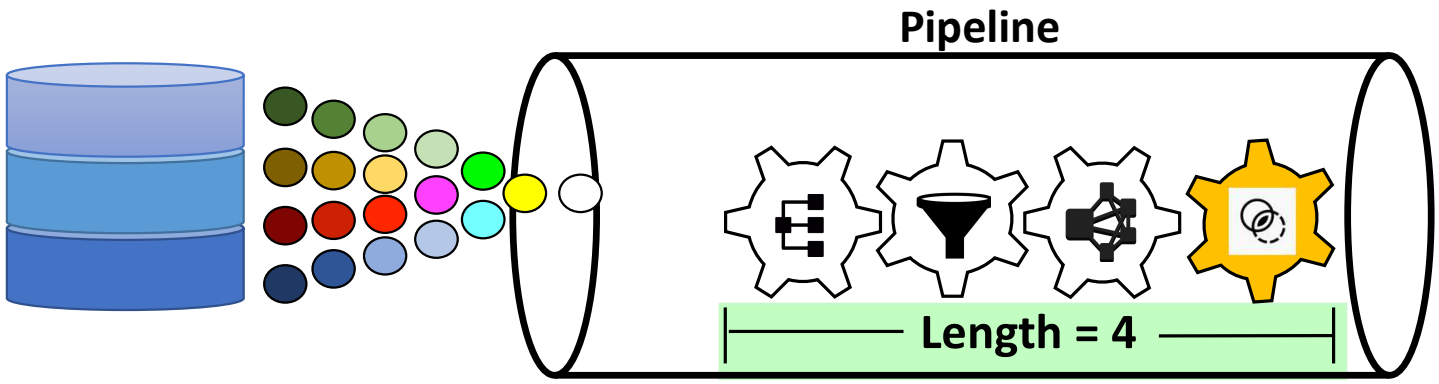
Terminal operation



Pipeline Length

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

There are 4 operations in the pipeline

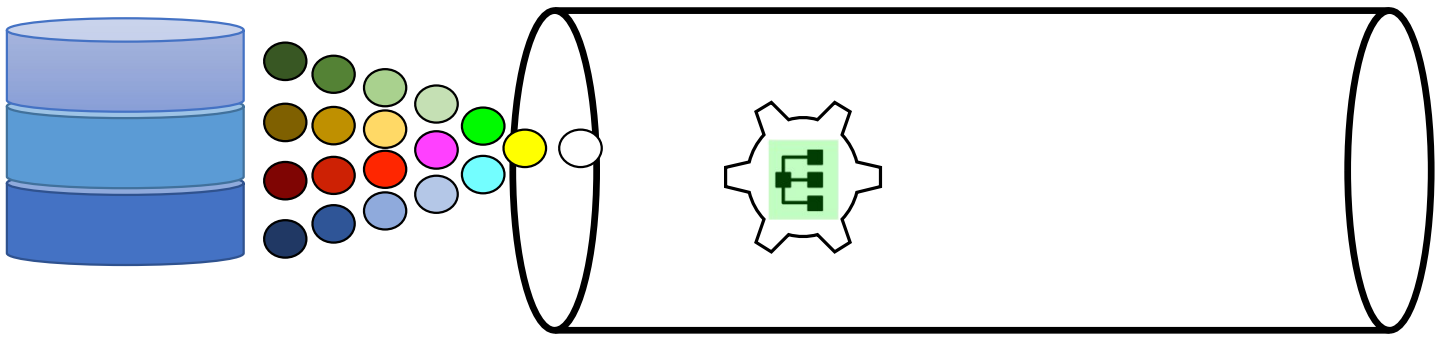


Execution Mode

```
transactionList.stream()
    .parallel()
    .filter(t -> t.getStatus() == Transaction.VALID)
    .map(Transaction::getID)
    .collect(Collectors.toSet());
```

Switches the **execution mode**

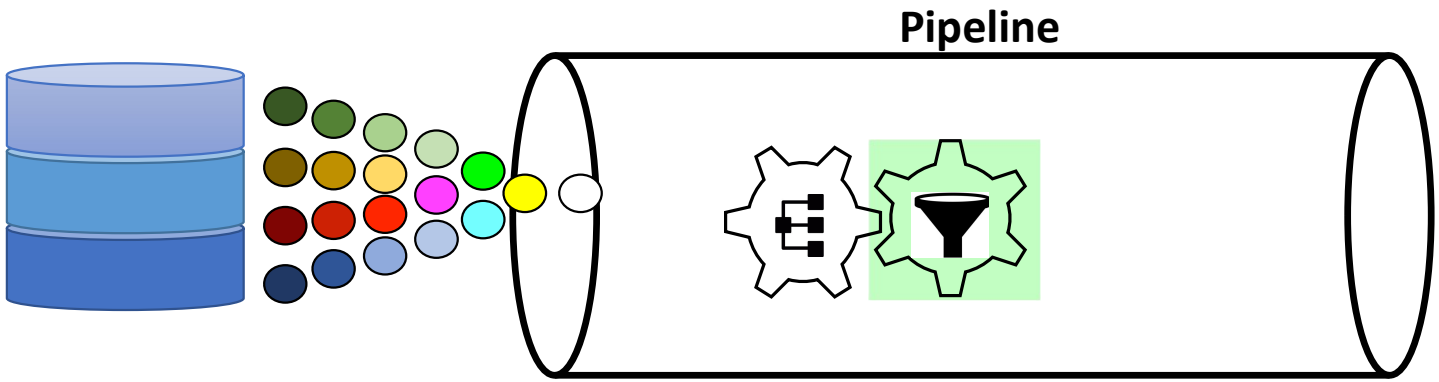
Execution Mode	
Sequential	Parallel



Intermediate Operations

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

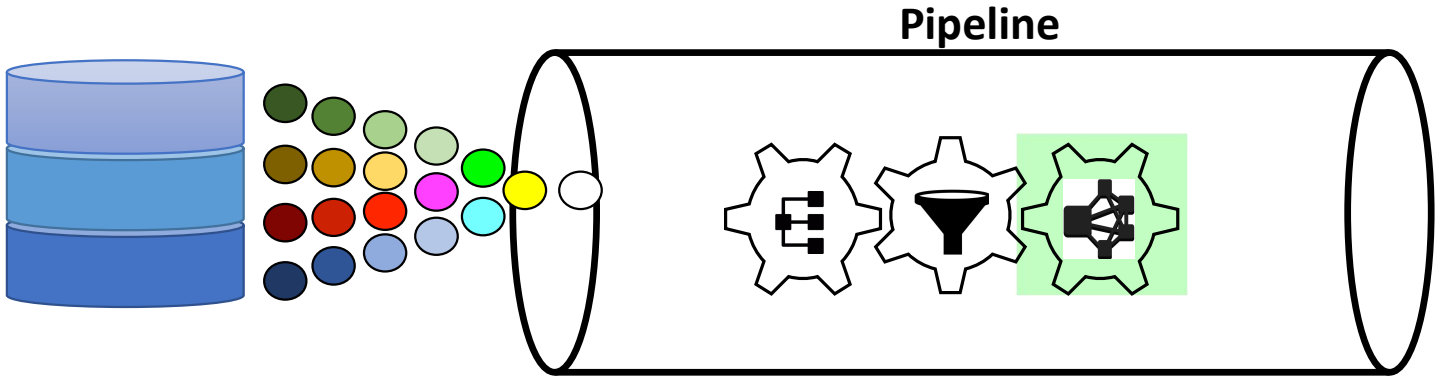
Filters valid transactions



Intermediate Operations

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

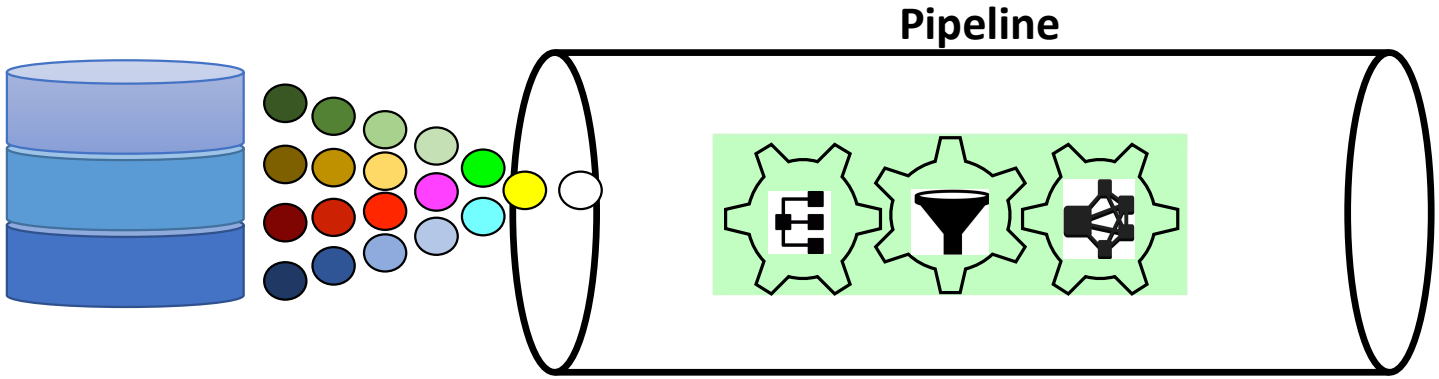
Extracts the IDs of valid transactions



Stateless Intermediate Operations

```
transactionList.stream()
    .parallel()
    .filter(t -> t.getStatus() == Transaction.VALID)
    .map(Transaction::getID)
    .collect(Collectors.toSet());
```

Stateless intermediate operations

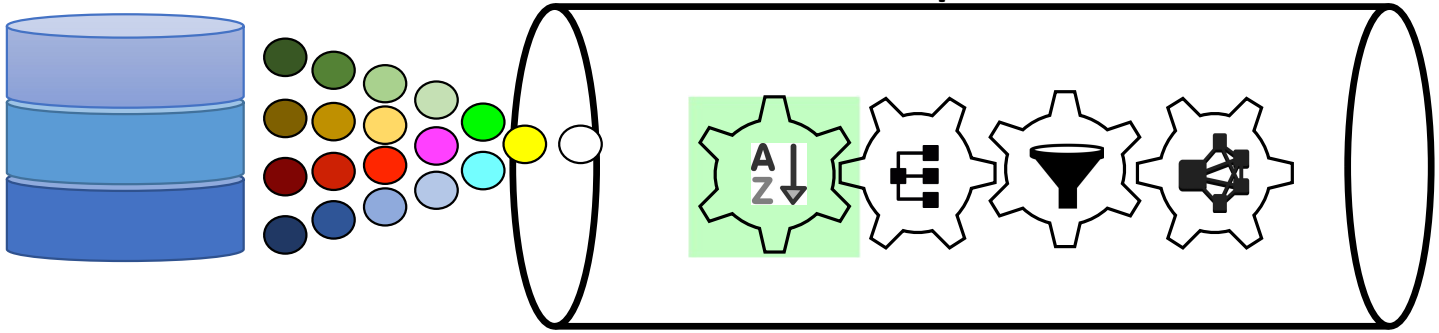


Stateful Intermediate Operations

```
transactionList.stream()
      .sorted()
      .parallel()
      .filter(t -> t.getStatus() == Transaction.VALID)
      .map(Transaction::getID)
      .collect(Collectors.toSet());
```

Stateful operation sorting the elements in the pipeline

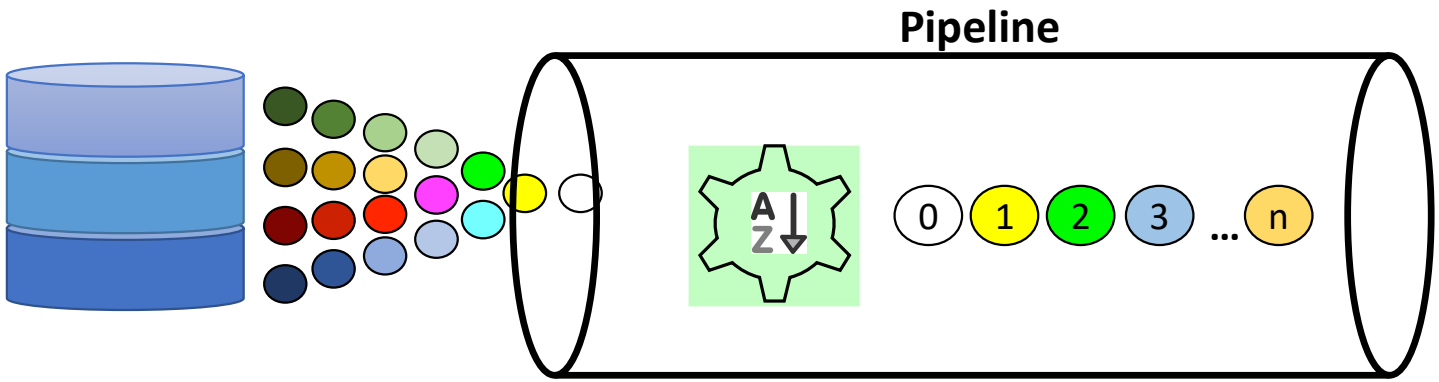
Pipeline



Stateful Intermediate Operations

```
transactionList.stream()
      .sorted()
      .parallel()
      .filter(t -> t.getStatus() == Transaction.VALID)
      .map(Transaction::getID)
      .collect(Collectors.toSet());
```

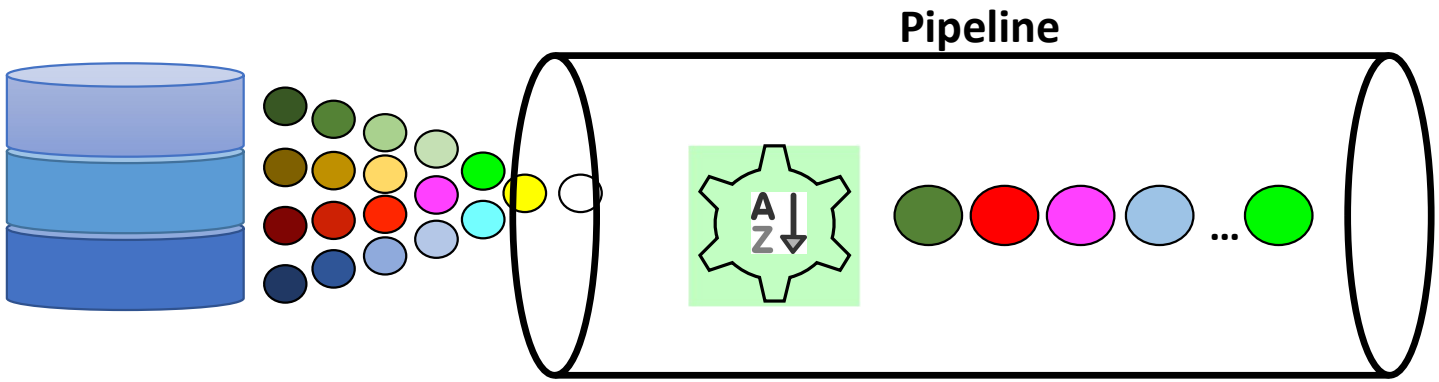
Stateful operations and ordering constraints may impair parallel performance



Stateful Intermediate Operations

De-orders the stream, which may improve parallel performance

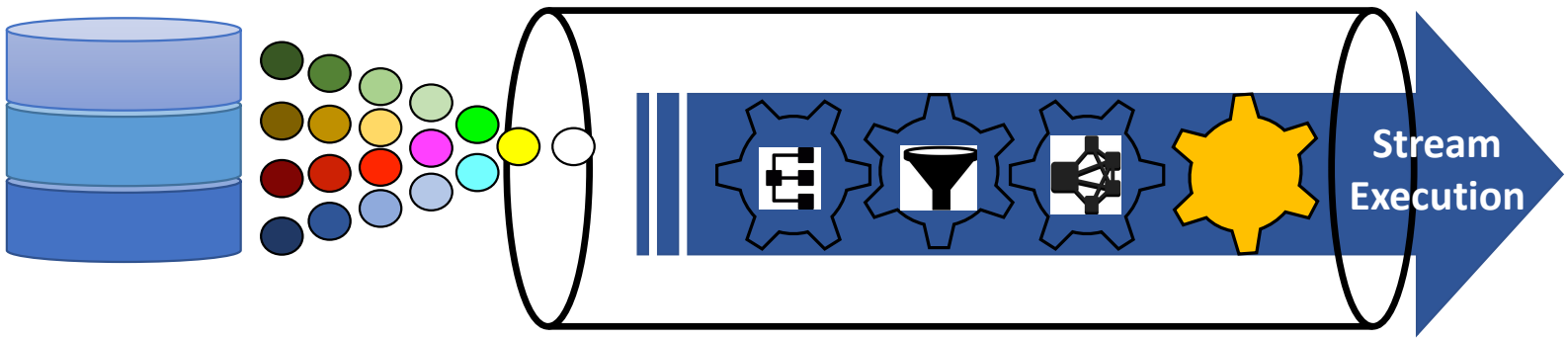
```
transactionList.stream()
      .unorderded()
      .parallel()
      .filter(t -> t.getStatus() == Transaction.VALID)
      .map(Transaction::getID)
      .collect(Collectors.toSet());
```



Stream Execution

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

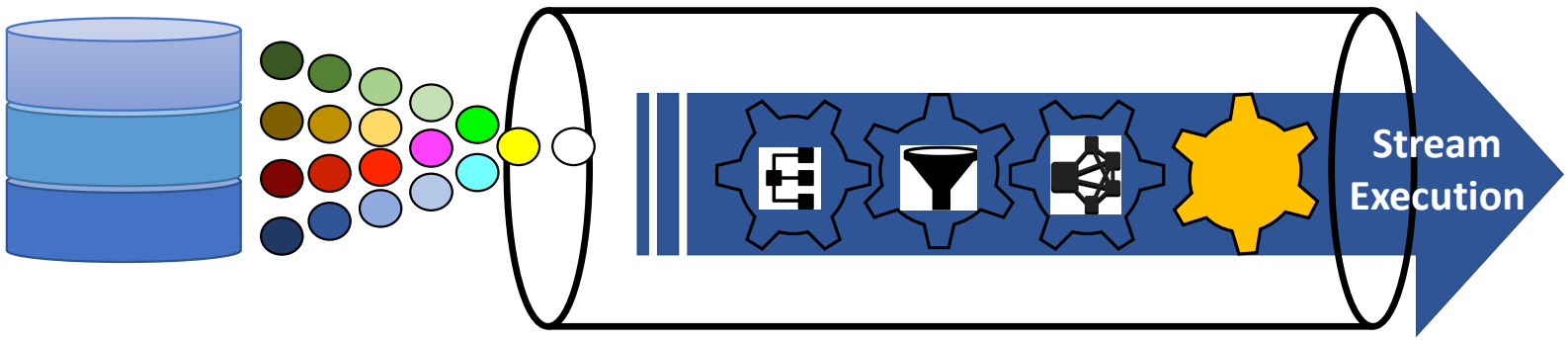
Stream execution



Stream Execution

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

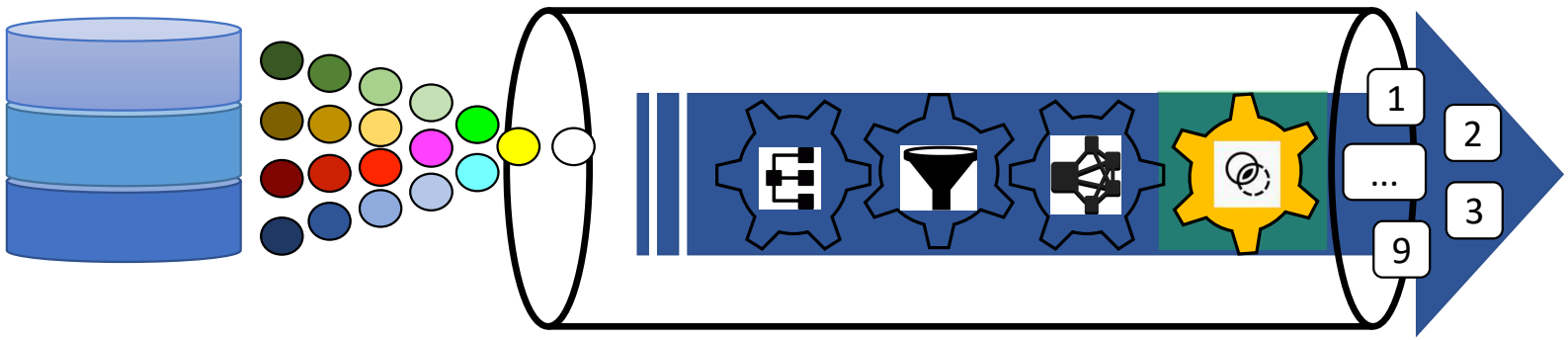
Performs a mutable reduction



Collector

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

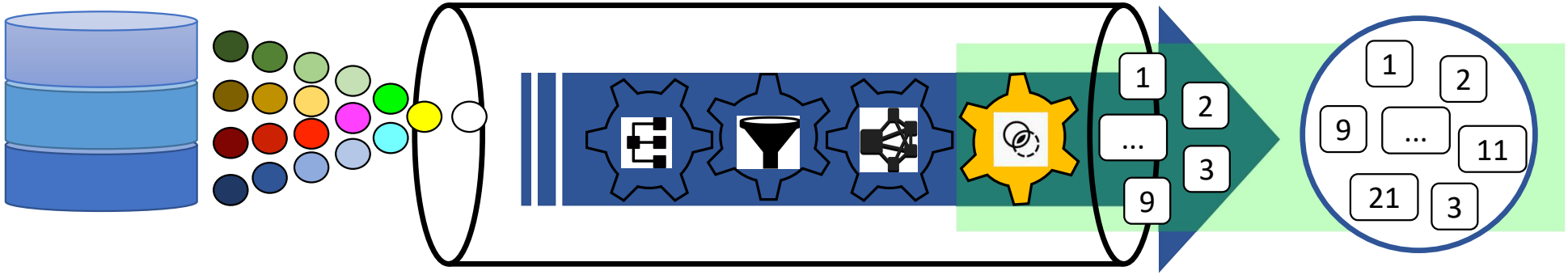
Collector



Collector

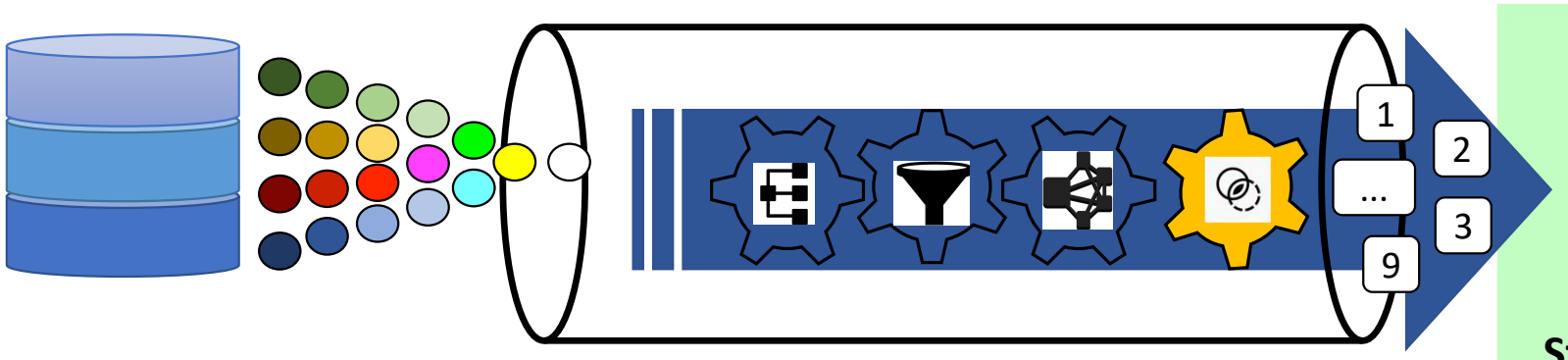
```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

Acumulates the IDs into a Set



Stream Result Type

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

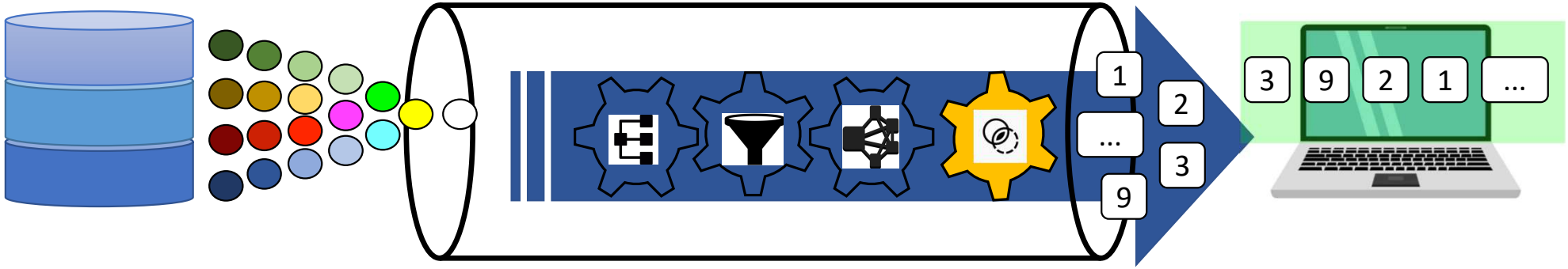


Stream result type
e.g., java.util.HashSet

Side Effects

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .forEach(System.out::print);
```

Prints the IDs of valid transactions



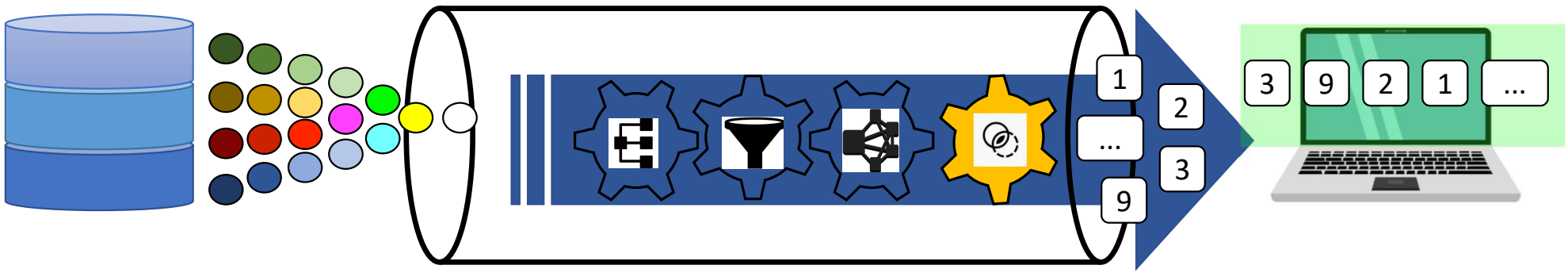
Side Effects

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .forEach(System.out::print);
```

Non-deterministic

3 9 2 1 ...

Prints the IDs of valid transactions



THE PROBLEM

Università della Svizzera italiana

The Problem

- Recent studies focused on the popularity of streams:
  - Tanaka et al. [1] mine 100 repositories to study the use of lambdas, streams, and the `java.util.Optional` class
  - Khatchadourian et al. [2] examine 34 projects to study the use of the Stream API
  - Nostas et al. [3] study 610 projects to do a partial replication of the study of Khatchadourian et al. at a larger scale

[1] Tanaka et al. *A Study on the Current Status of Functional Idioms in Java*. IEICE Trans. on Info. and Sys. 2019
[2] Khatchadourian et al. *An Empirical Study on the Use and Misuse of Java 8 Streams*. FASE'20.
[3] Nostas et al. *How Do Developers Use the Java Stream API*. ICCSA'21.

The Problem

- **Limitations:**
  - They consider only a small number of projects
  - They rely on manual code inspection and static analysis techniques
    - Disregarding the analysis of dynamic metrics unique to streams

There is yet limited evidence to make conclusions on how the Stream API is used

CONTRIBUTIONS

Università della Svizzera italiana

Contributions

- We present *Stream-Analyzer*, a novel tool specifically designed for the characterization of dynamic metrics specific to streams
  - Detects all forms of stream creation and execution as well as all operations available in the Stream API
  - Targets dynamic information and metrics that enable fine-grained characterization of stream processing
    - The selected metrics are suitable to be massively collected in the wild

Università della Svizzera italiana

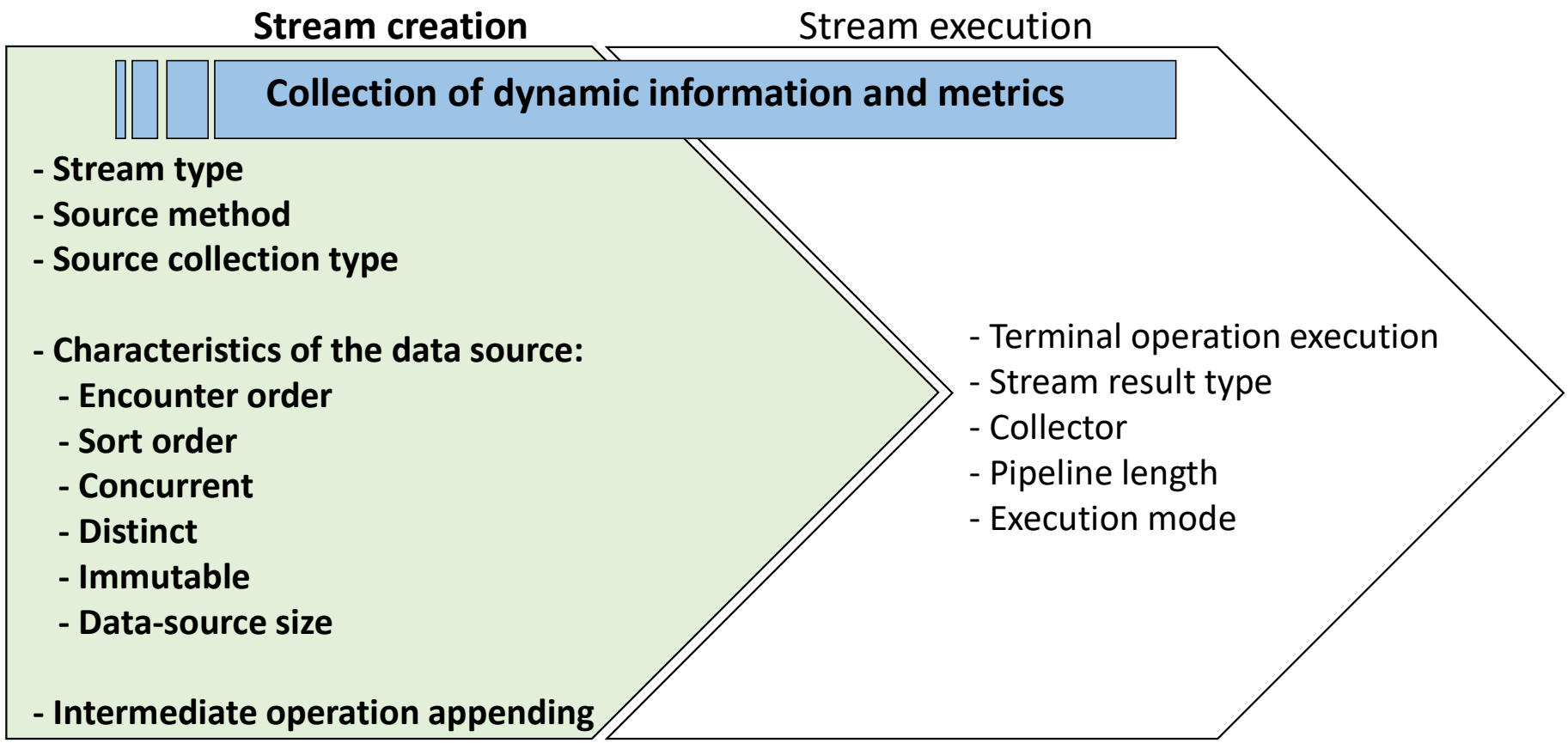
Contributions

- We use Stream-Analyzer to conduct the first large-scale empirical study on the use of Java streams
  - Stream-Analyzer runs on top of *NAB* [4], to enable a fully automated approach (no manual intervention, no manual code inspection)
- We target 1'8M open-source projects available in GitHub, of which 132K use the Stream API
  - Our study analyzes a total of 620M streams
- We confirm the findings of previous studies and report new dynamic information and metrics that related work overlooks

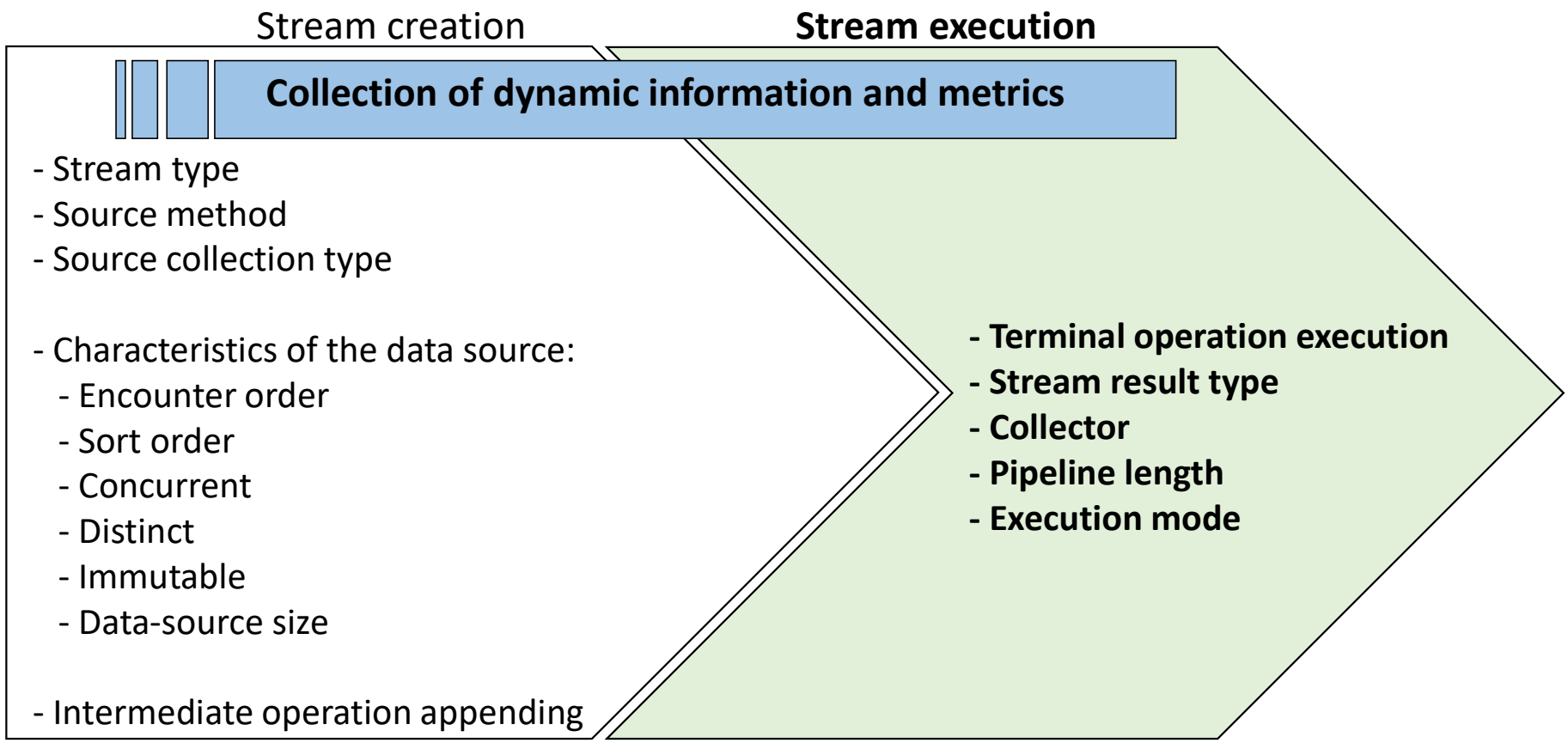
[4] Villazón et al. *Automated Large-Scale Multi-Language Dynamic Program Analysis in the Wild*. ECOOP'19.

Stream-Analyzer

Stream-Analyzer

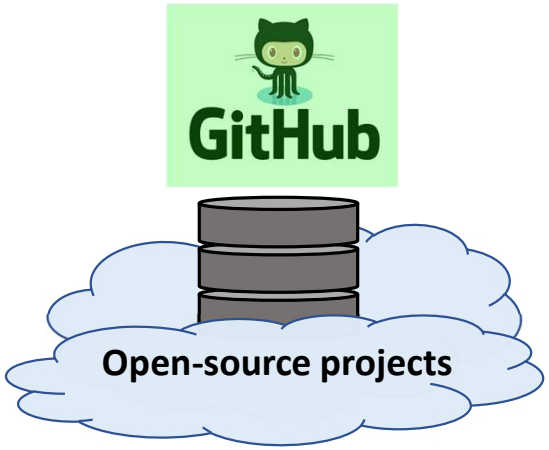


Stream-Analyzer

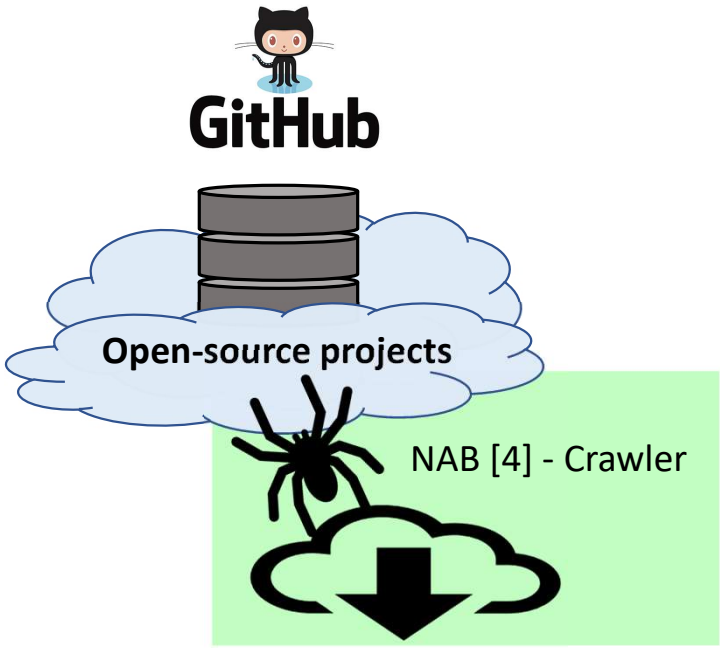


LARGE SCALE ANALYSIS

Stream-Analyzer + NAB

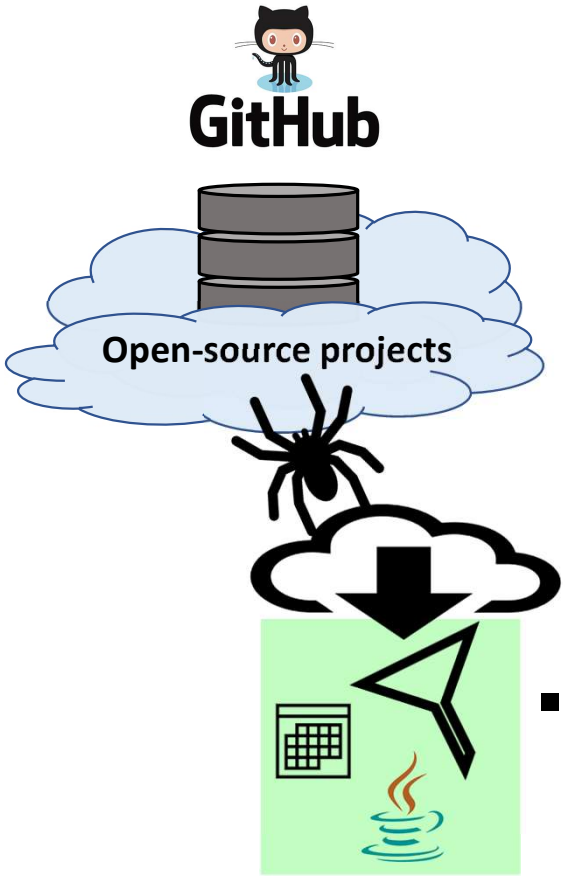


Stream-Analyzer + NAB



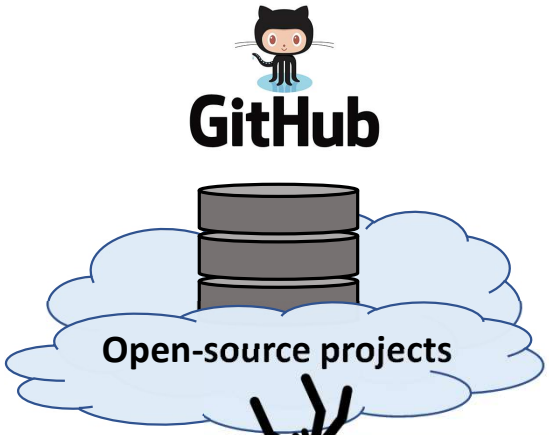
[4] Villazón et al. *Automated Large-Scale Multi-Language Dynamic Program Analysis in the Wild*. ECOOP'19.

Stream-Analyzer + NAB

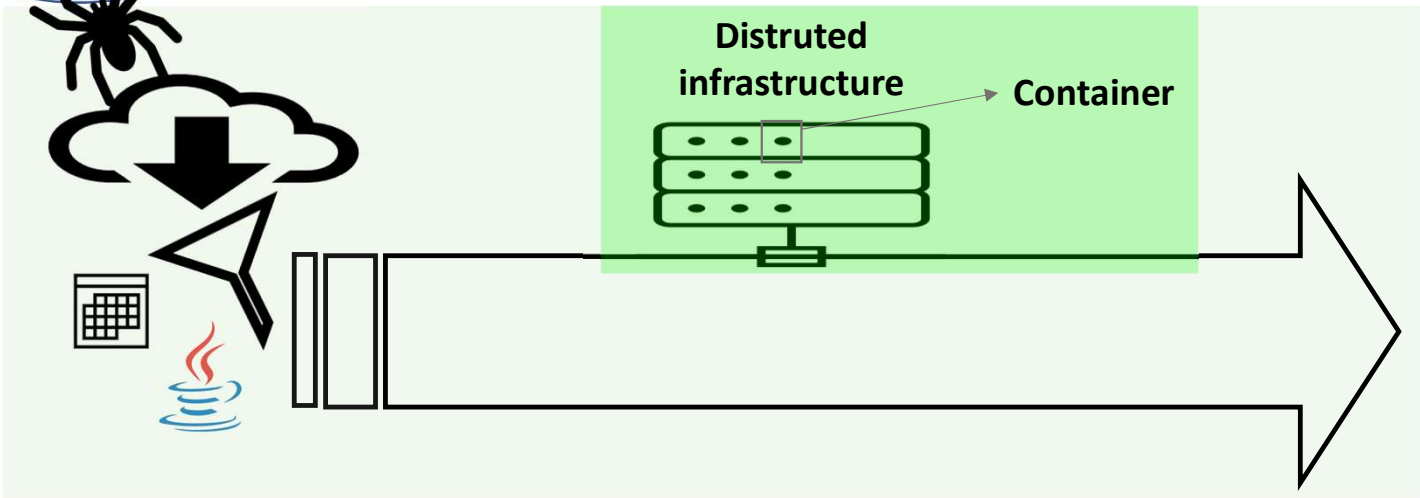


- **Timeframe:**
  - Java projects with at least one commit in year 2020

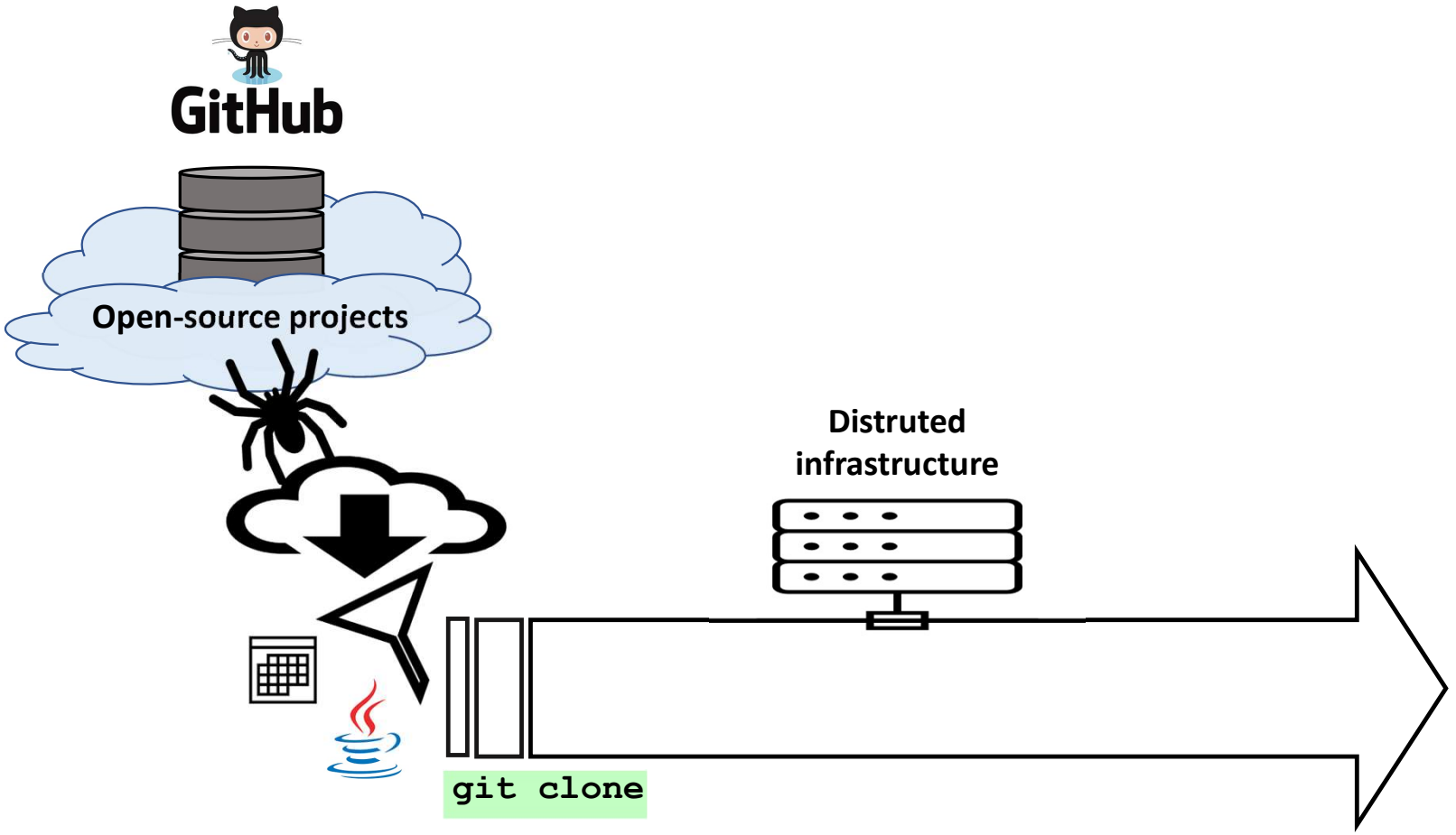
Stream-Analyzer + NAB



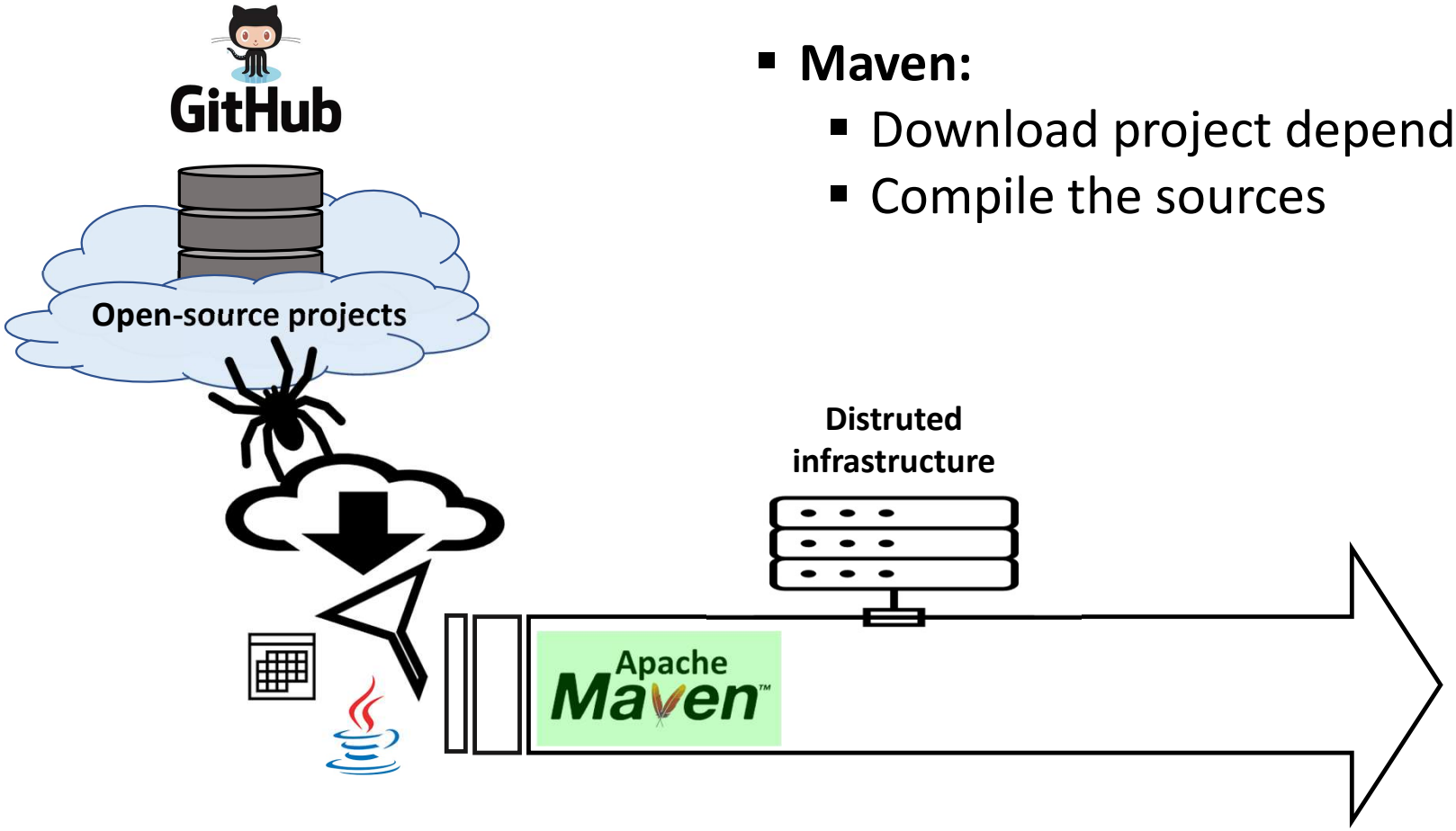
- **Containerization:**
  - Security: enables isolation
  - Performance: eases parallelization



Stream-Analyzer + NAB

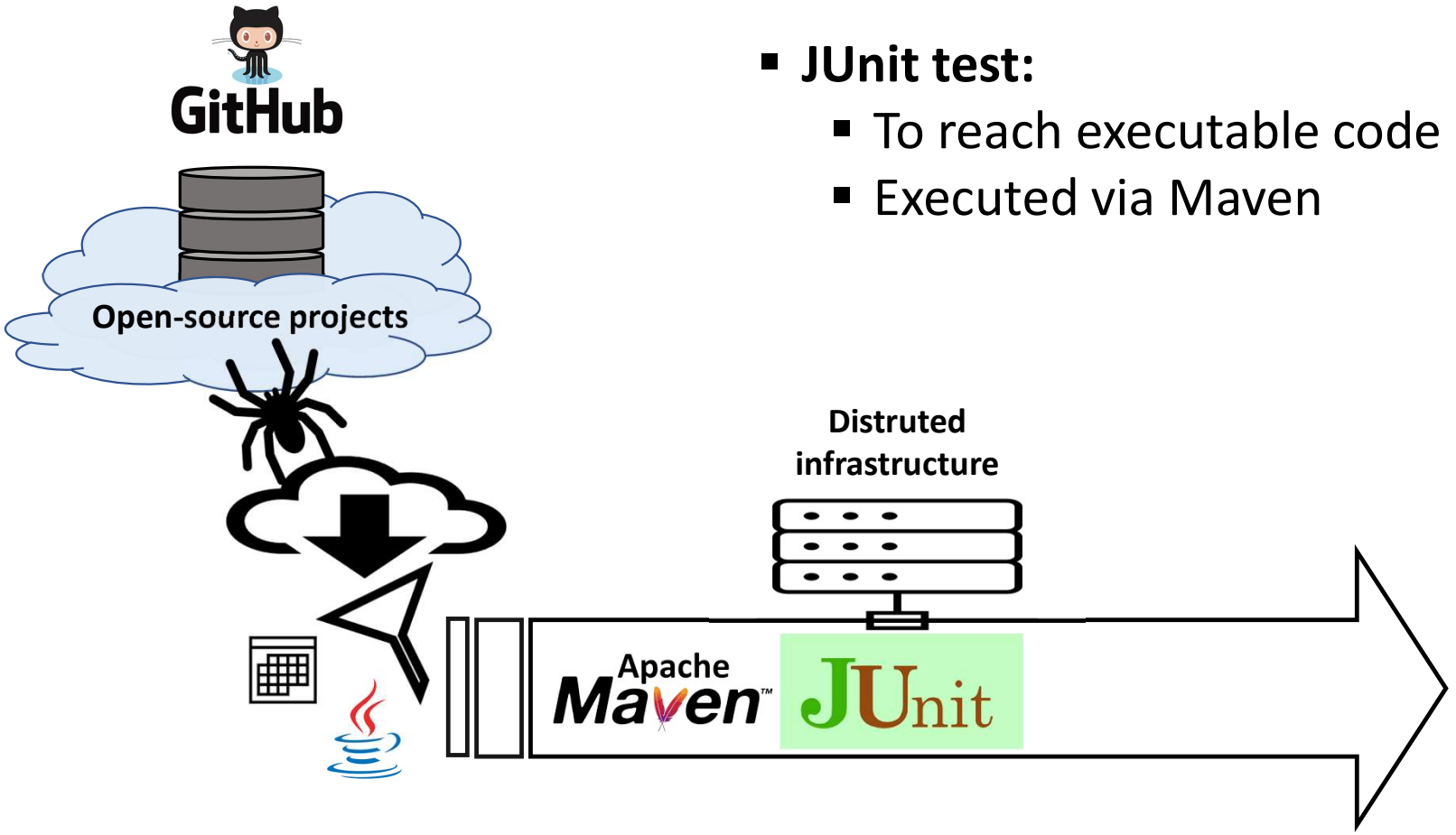


Stream-Analyzer + NAB



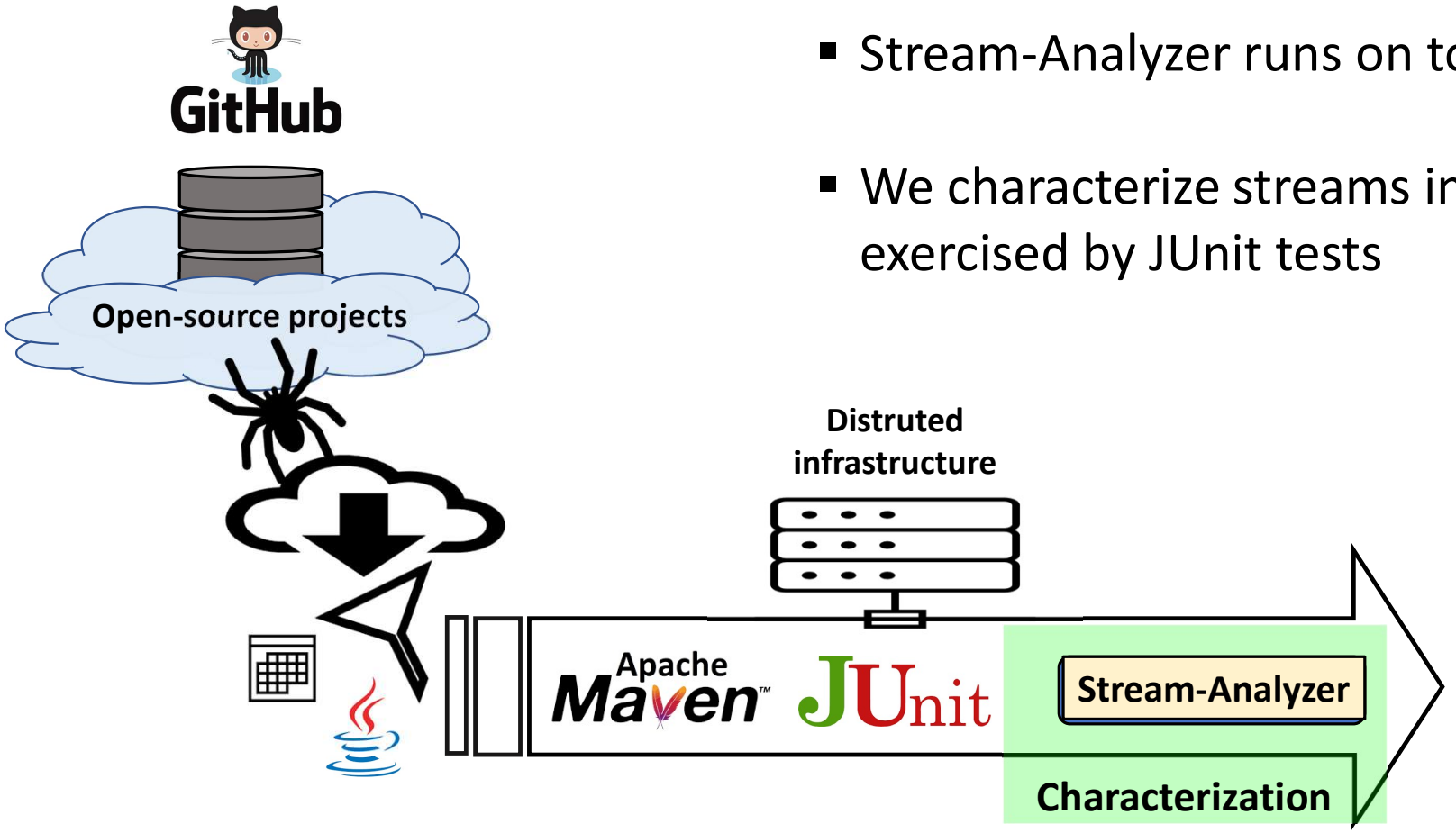
- **Maven:**
  - Download project dependencies
  - Compile the sources

Stream-Analyzer + NAB



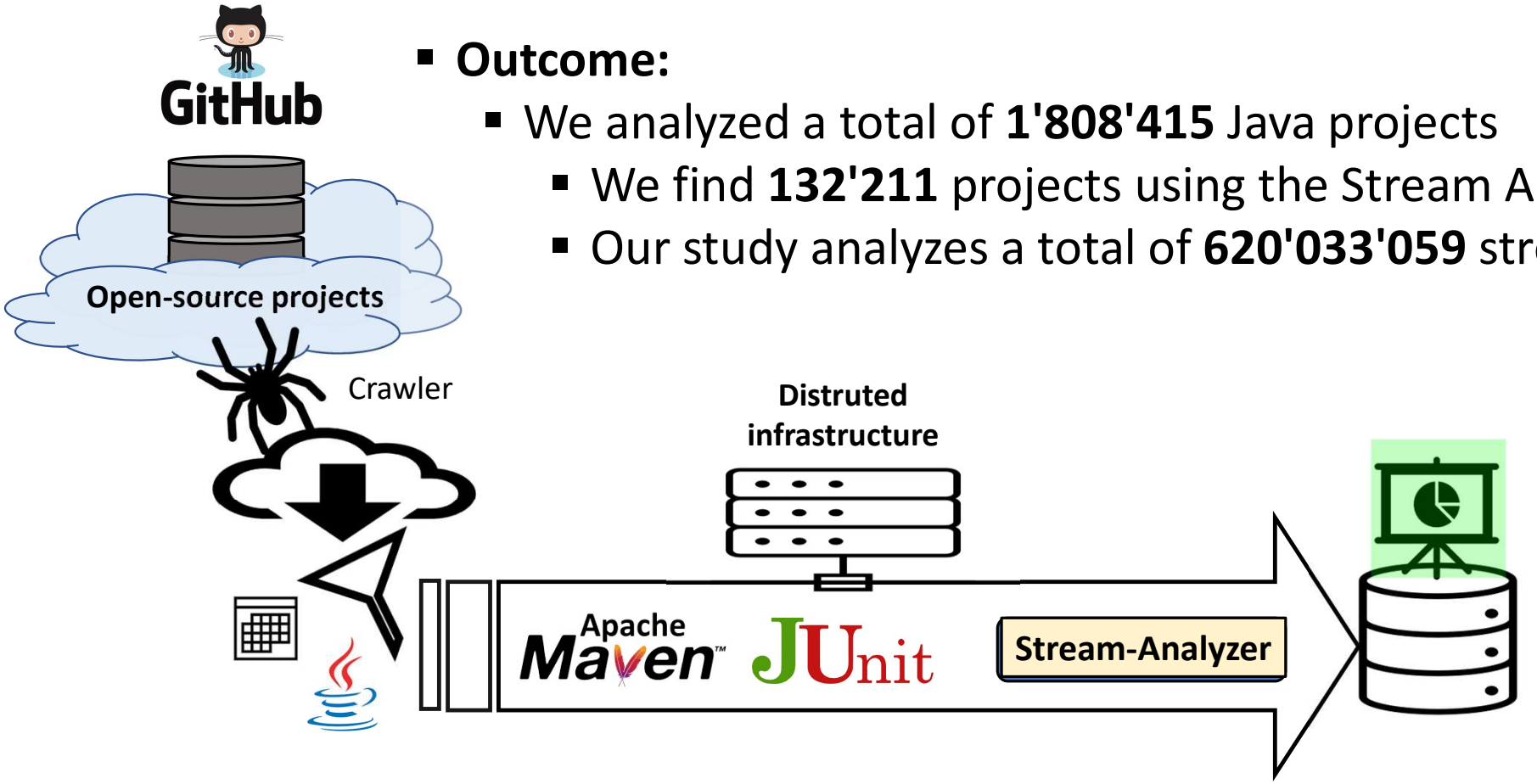
- **JUnit test:**
  - To reach executable code
  - Executed via Maven

Stream-Analyzer + NAB



- Stream-Analyzer runs on top of NAB
- We characterize streams in code exercised by JUnit tests

Stream-Analyzer + NAB



- **Outcome:**
  - We analyzed a total of **1'808'415** Java projects
    - We find **132'211** projects using the Stream API
    - Our study analyzes a total of **620'033'059** streams

RESULTS

Source Methods

- **Source methods:** the method used to create the stream

Source Method	Occurrences	%
java.util.Collection.stream	349'303'956	56.33
java.util.stream.IntStream.range	87'677'566	14.14
java.util.Arrays.stream	58'690'188	9.46
java.util.Collections$SynchronizedCollection.stream	27'069'317	4.37
java.util.stream.Stream.of	26'565'005	4.28
java.util.stream.Stream.concat	22'869'945	3.69
java.util.Random.ints	12'020'051	1.94
java.util.Collections$UnmodifiableCollection.stream	11'795'112	1.90
java.util.stream.Stream.empty	7'397'430	1.19
java.lang.CharSequence.chars	5'001'524	0.81

Finding 1: Streams are mainly created from collections, generators of integers, and arrays

Università della Svizzera italiana

Source Collection Type

- **Source collection type:** the type of the collection from which a stream is created, if any

Source Collection Type	Occurrences	%
java.util.ArrayList	258'713'462	41.72
java.util.HashSet	29'330'326	4.73
java.util.Collections$SynchronizedSet	25'569'216	4.12
java.util.LinkedList	12'246'909	1.98
java.util.LinkedHashSet	7'730'718	1.25
java.util.Arrays$ArrayList	7'480'889	1.21
java.util.Collections$EmptyList	5'759'041	0.93
java.util.HashMap$EntrySet	5'687'645	0.92
java.util.Collections$UnmodifiableCollection	5'466'774	0.88
java.util.Collections$UnmodifiableRandomAccessList	4'806'994	0.78

Finding 2: Streams generated from collections are mostly created from list and sets

Stream Types

- **Stream types:** the type of the stream

Stream Types	Occurrences	%
java.util.Stream	504'832'673	81.41
java.util.IntStream	114'970'830	18.54
java.util.LongStream	284'473	0.05
java.util.DoubleStream	5'627	0.001

Finding 3: Object-based stream types are far more popular than primitive-based types

Università della Svizzera italiana

Characteristics of the Data Source

- **Data source:** the source of data from which the stream is created

Data-source Characteristics	Occurrences	%
Encounter order	543'905'362	87.71
Sorted order	91'226'771	14.71
Concurrent	873'955	0.14
Distinct	197'900'117	31.91
Immutable	172'796'006	27.87

Università della Svizzera italiana

Characteristics of the Data Source

- **Data source:** the source of data from which the stream is created

Data-source Characteristics	Occurrences	%
Encounter order	543'905'362	87.71
Sorted order	91'226'771	14.71
Concurrent	873'955	0.14
Distinct	197'900'117	31.91
Immutable	172'796'006	27.87

Finding 4: Stream data sources often allow duplicates, have ordering constraints, are mutable, and do not support concurrent modification

Data-source Size

- **Data-source size:** the total number of elements in the data source upon stream creation

Size Range	Occurrences	%
0	51'585'754	13.02
$[10^0, 10^1)$	310'352'326	78.33
$[10^1, 10^2)$	29'085'769	7.34
$[10^2, 10^3)$	4'620'480	1.17
$[10^3, 10^4)$	557'893	0.14
$[10^4, 10^5)$	1692	$4 \cdot 10^{-4}$
$[10^5, 10^6)$	34	$1 \cdot 10^{-5}$
$[10^6, 10^7)$	5	$1 \cdot 10^{-6}$

Finding 5: Stream data sources typically have few elements

Intermediate Operations

- **Intermediate operations:** operations that produce other streams that can be further processed

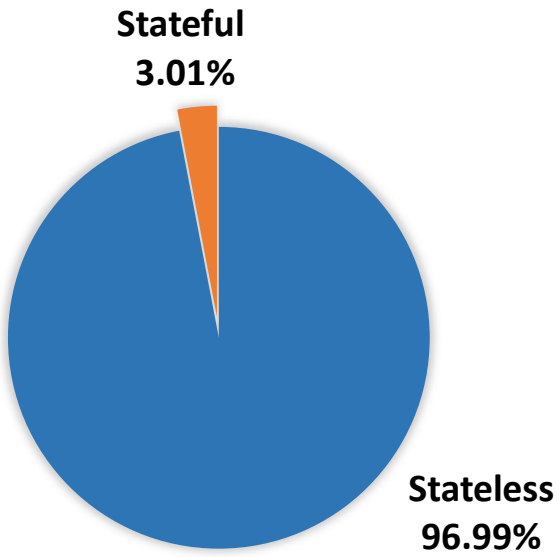
Intermediate Operation	Occurrences	%
map	427'009'693	51.57
filter	227'938'406	27.53
mapToObj	77'898'391	9.41
flatMap	34'512'362	4.17
mapToInt	27'327'892	3.30
limit	12'697'146	1.53
mapToLong	6'264'703	0.76
sorted	5'275'399	0.64
skip	4'041'440	0.49
distinct	2'877'764	0.35

- 827'977'847 intermediate operations detected
- `Unordered()` is not popular (less than 0.001%)

Finding 6: Mapping and filtering are by far the most popular intermediate operations

Intermediate Operations

- **Intermediate operations:** operations that produce other streams that can be further processed



- Pipelines containing only stateless operations can be processed in a single pass, potentially improving performance

Finding 7: Stateful intermediate operations are barely used

Terminal Operations

- **Terminal operations:** operations that trigger stream execution

Terminal Operation	Occurrences	%
collect	197'338'242	31.83
forEach	145'774'791	23.51
allMatch	68'072'088	10.98
findFirst	51'274'578	8.27
anyMatch	43'892'114	7.08
sum	38'716'266	6.24
findAny	38'505'248	6.21
reduce	9'720'348	1.57
toArray	7'555'733	1.22
count	6'033'536	0.97

- 620'033'509 terminal operations detected

Finding 8: Map-reduce-like data reductions via `collect` and iterative-style processing via `forEach` are the most common terminal operations

Terminal Operations

- **Terminal operations:** operations that trigger stream execution

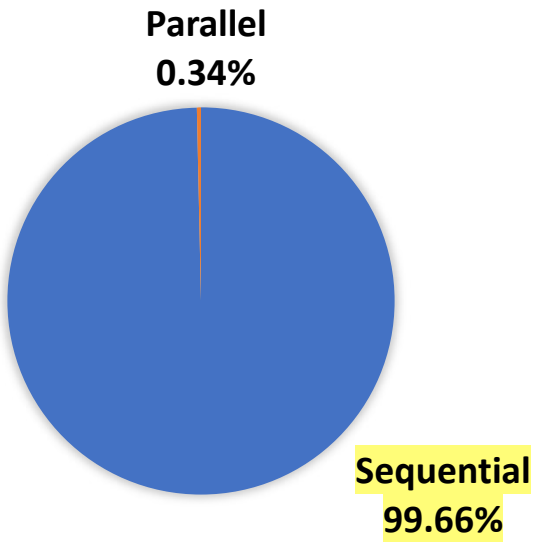
Terminal Operation	Occurrences	%
collect	197'338'242	31.83
forEach	145'774'791	23.51
allMatch	68'072'088	10.98
findFirst	51'274'578	8.27
anyMatch	43'892'114	7.08
sum	38'716'266	6.24
findAny	38'505'248	6.21
reduce	9'720'348	1.57
toArray	7'555'733	1.22
count	6'033'536	0.97

- Deterministic terminal operations are the most used (70.28%)

Finding 9: Deterministic terminal operations are more popular than nondeterministic ones

Execution Mode

- **Execution mode:** whether the stream is executed sequentially or in parallel



- 620'033'509 streams were executed
- 60'094 streams lack a terminal operation

Finding 10: Parallel streams are not popular

Pipeline Length

- **Length:** the total number of operations in the pipeline

Length	Occurrences	%
2	293'021'334	47.25
1	123'683'254	19.95
3	119'522'688	19.27
5	44'258'049	7.14
4	39'501'712	6.37
0	46'102	$7 \cdot 10^{-3}$
7	40'050	$6 \cdot 10^{-3}$
6	15'448	$2 \cdot 10^{-3}$
8	2'468	$4 \cdot 10^{-4}$
11	2'447	$4 \cdot 10^{-4}$
10	34	$5 \cdot 10^{-6}$
14	17	$3 \cdot 10^{-6}$

- The average pipeline length is only 2.34
- 46'102 streams have no operations in their pipelines.

Finding 11: Stream pipelines are typically composed of few operations

Collectors

- **Collector:** the collector to perform a mutable reduction

Collector	Occurrences	%
N/A	422'695'267	68.17
Collectors.toList	146'091'282	23.56
Collectors.toSet	15'855'229	2.56
Collectors.toCollection	15'550'665	2.51
Collectors.toMap	6'819'658	1.10
Collectors.joining	5'830'648	0.94
Collectors.collectingAndThen	5'495'747	0.89
Collectors.groupingBy	987'487	0.16
Collectors.counting	553'012	0.09
Collectors.reducing	121'687	0.02

- `groupingByConcurrent()` is not widely used (less than 0.001%)

Finding 12: Collectors that collect elements into lists and set are the most used

Stream Result Types

- **Stream Result Type:** the type of the output produced after the stream is executed, if any

Stream Result Type	Occurrences	%
java.util.ArrayList	148'567'911	23.96
void	145'918'994	23.53
boolean	112'159'247	18.09
java.util.Optional	99'032'395	15.97
int	29'961'760	4.83
java.util.HashSet	16'333'376	2.63
java.util.HashMap	15'464'414	2.49
int[]	9'576'981	1.54
java.lang.String	9'506'464	1.53
Object[]	8'745'210	1.41

Finding 13: Lists are the most popular data structure used to store the output of a stream

Università della Svizzera italiana

Summary

- Streams are mainly used to manipulate lists
- The data sources commonly contain few elements
- Pipelines typically contain few operations
- Lists and sets are widely used to store the result of the stream
- Parallel streams are concurrent reductions are rarely used
- MapReduce-style stream processing is the most popular one
  - Streams are also widely used for iterative-style collection processing

Università della Svizzera italiana

Conclusions

- We presented Stream-Analyzer, a novel tool to characterize dynamic metrics specific to streams
- We presented the first large-scale empirical study on the use of streams
  - Fully automated approach
  - 1.8M projects analyzed
    - Find 132K projects that use the Stream API
      - Analyze 620M streams
- Our results confirm the findings of previous studies at a much larger scale, considering runtime metrics previously overlooked

Università della Svizzera italiana

Future Work

- Consider a newer and longer time frame
- Consider other largest code-hosting facilities (e.g., GitLab, BitBucket)
- Consider other build systems (e.g., Apache Ant, Gradle)
- Find ways to detect and optimize suboptimal stream processing in the wild

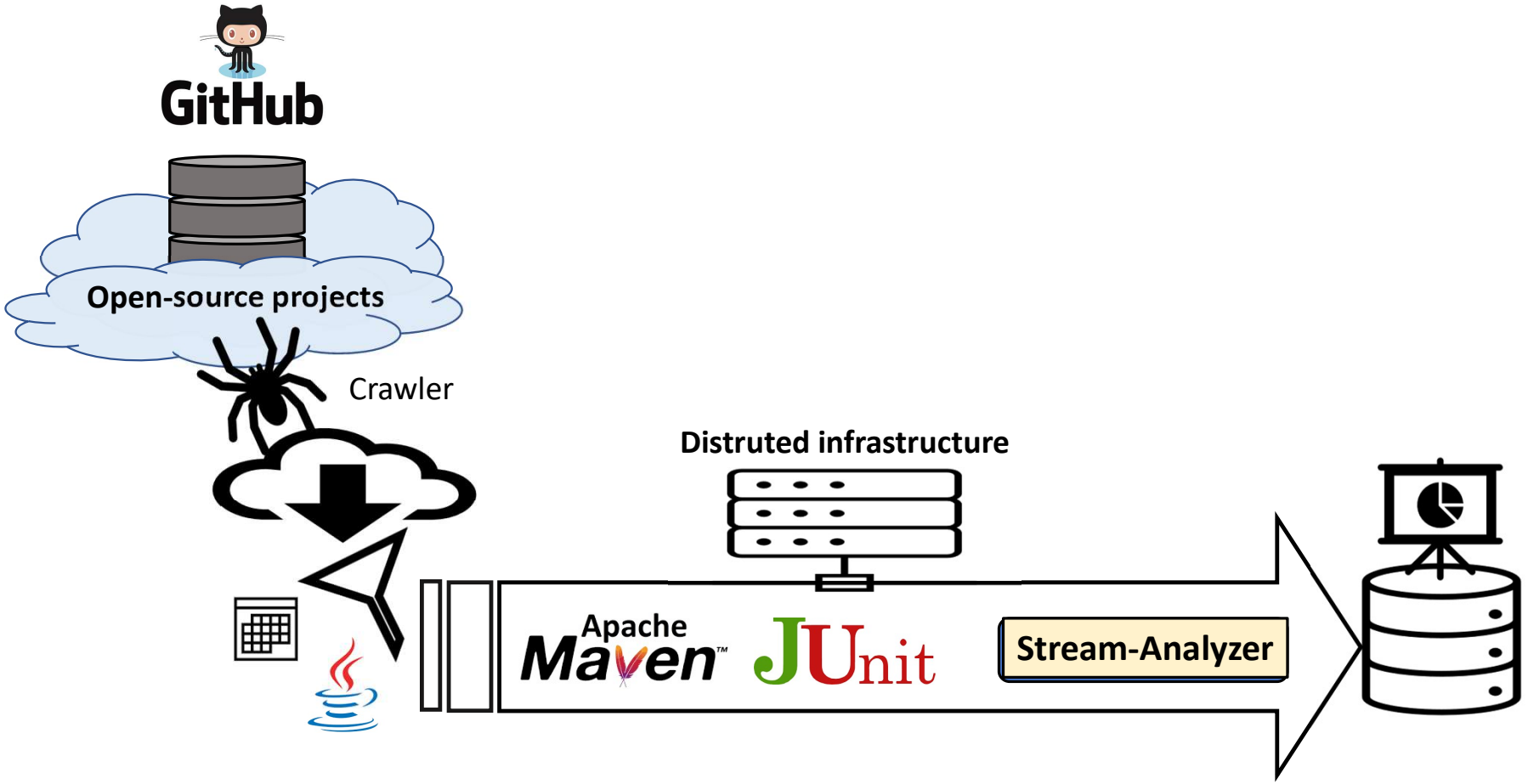
Università
della
Svizzera
italiana

Thanks for your attention

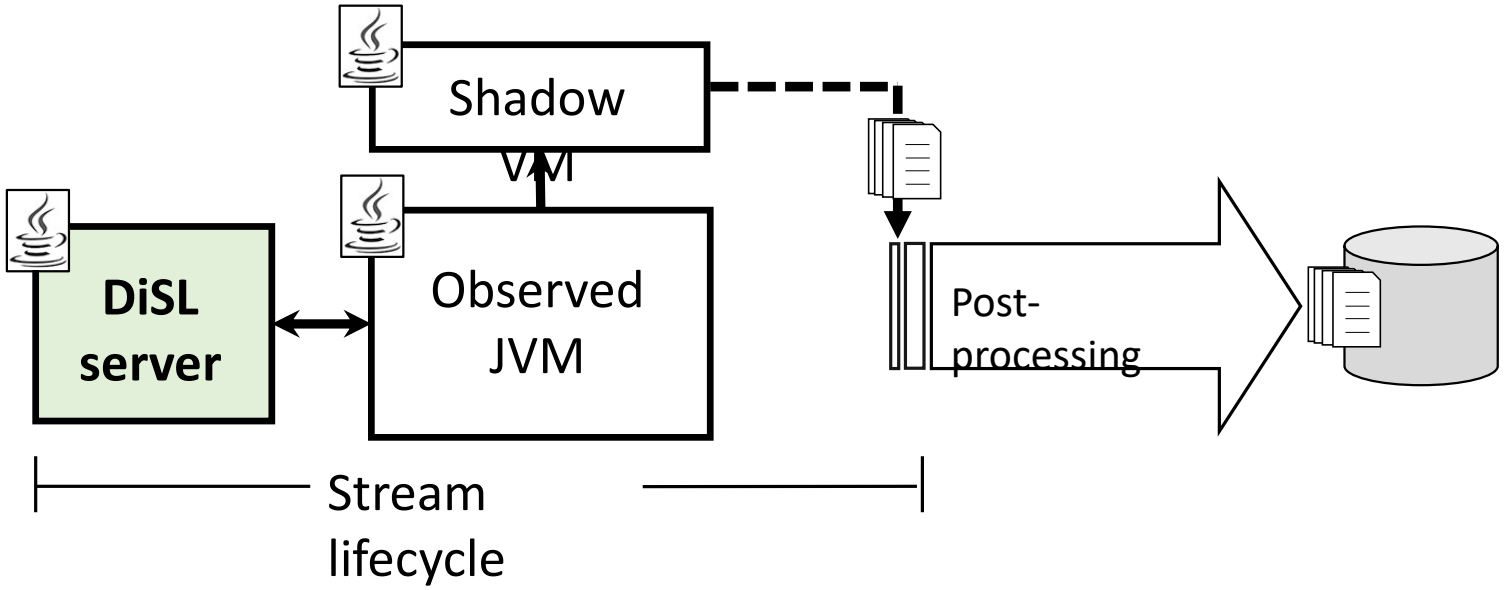
- **Eduardo Rosales:** `rosale@usi.ch`

BACKUP SLIDES

Stream-Analyzer - Implementation



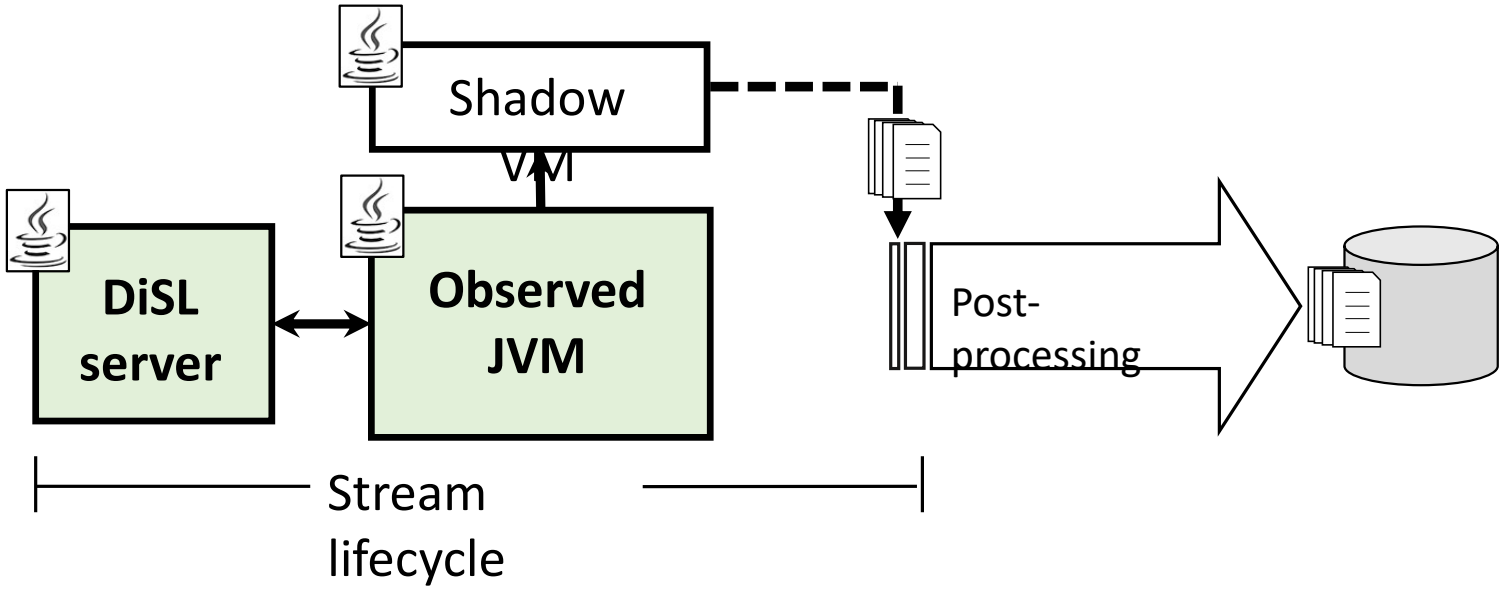
Stream-Analyzer - Implementation



- DiSL server [5]: performs load-time instrumentation
- DiSL offers full bytecode coverage
  - All streams are detected

[5] Marek et al. DiSL: *A Domain-specific Language for Bytecode Instrumentation*. AOSD'12.

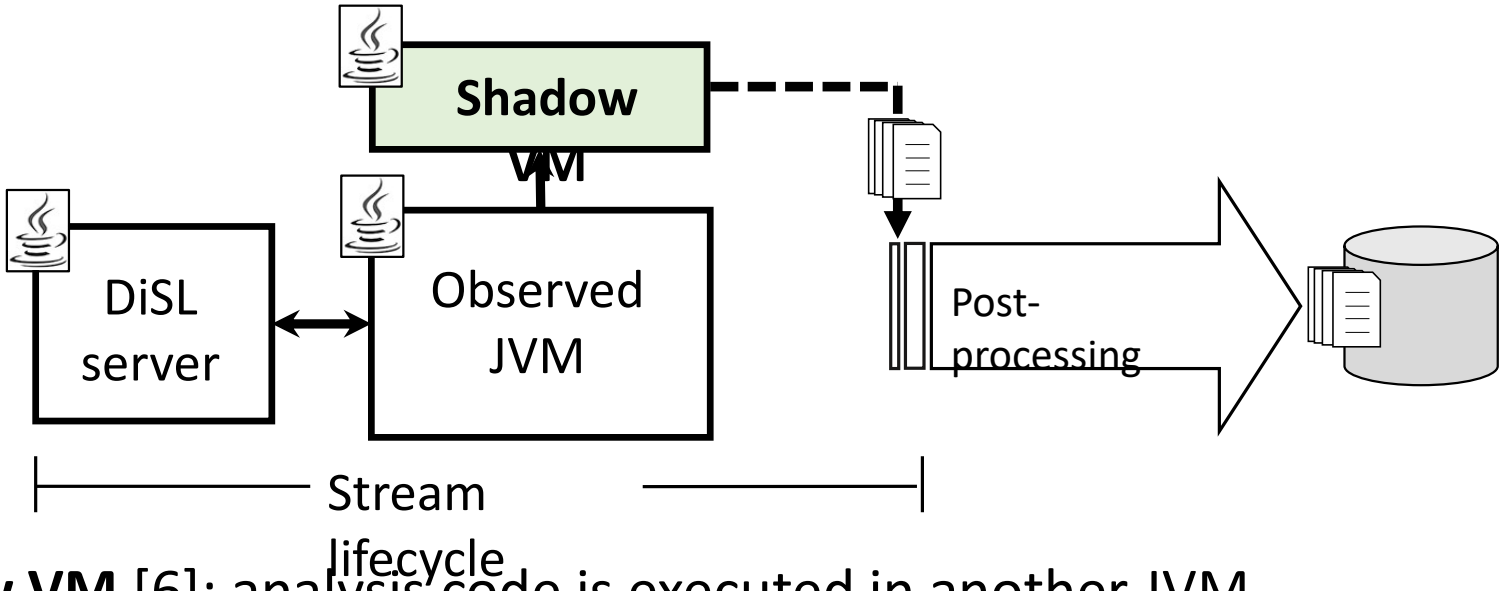
Stream-Analyzer - Implementation



- DiSL server [4]: performs load-time instrumentation
- DiSL offers full bytecode coverage
  - All streams are detected

[5] Marek et al. DiSL: *A Domain-specific Language for Bytecode Instrumentation*. AOSD'12.

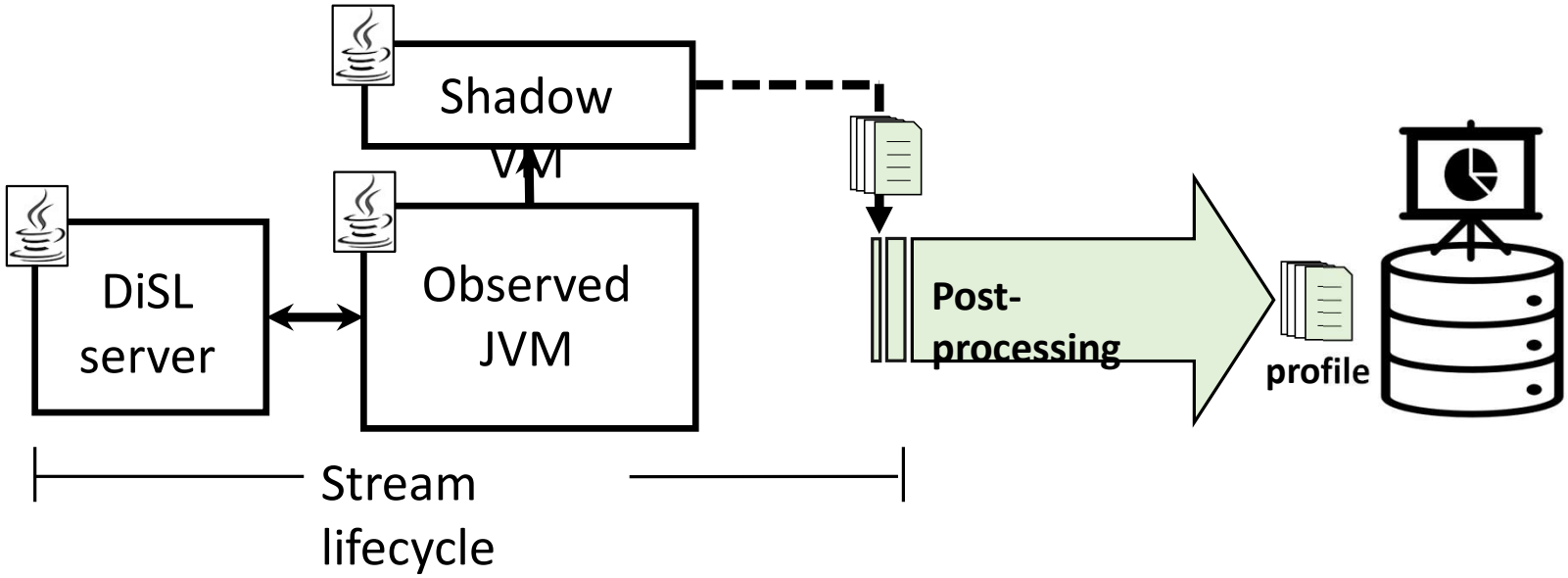
Stream-Analyzer - Implementation



- **Shadow VM** [6]: analysis code is executed in another JVM
  - The profiling logic is deployed in isolation to the observed application

[6] Marek et al. *ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform*. GPCE'13.

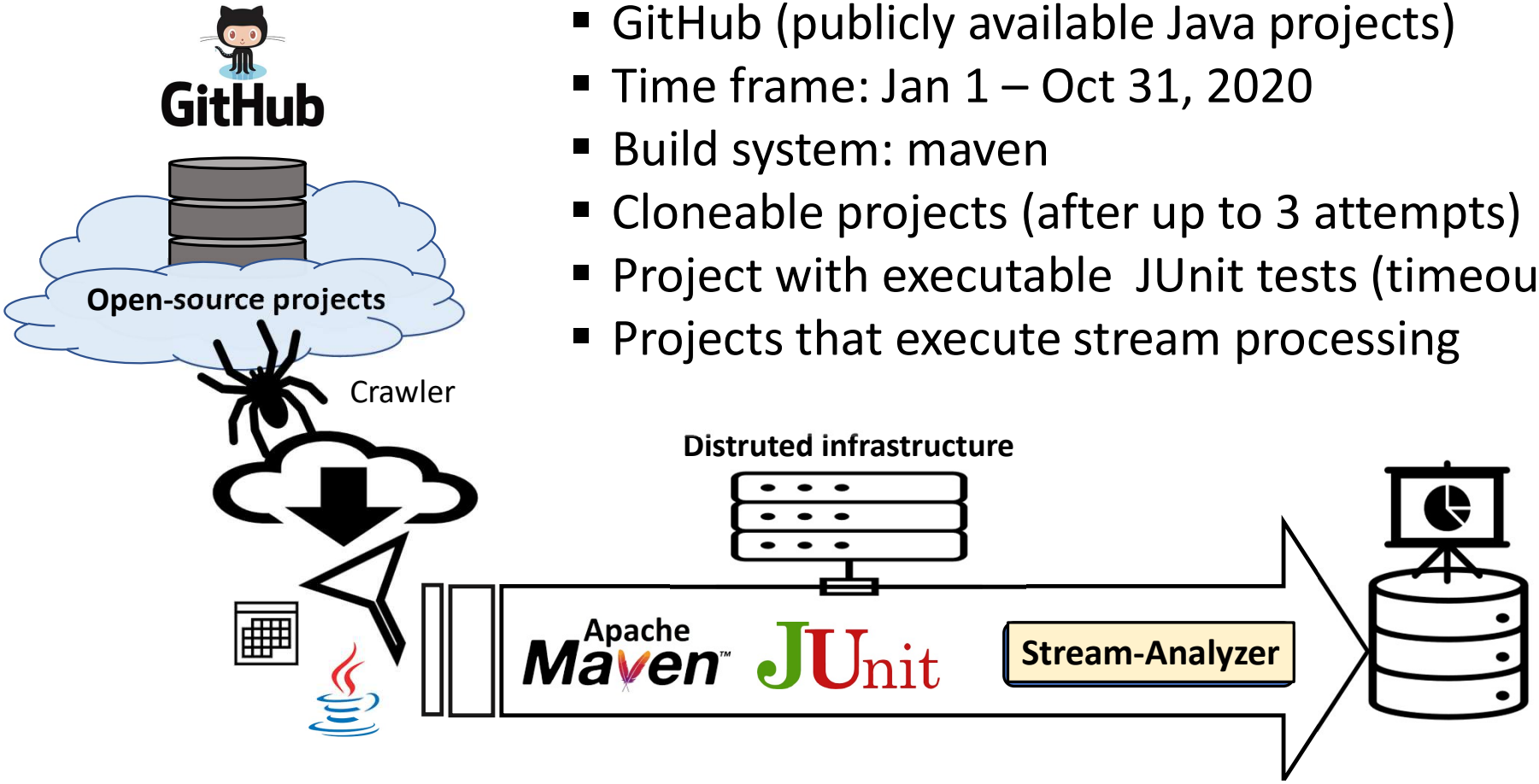
Stream-Analyzer - Implementation



- Offline post-processing produces a single integrated profile
- Profiles are stored on a database for further data analysis

Scope

- **Scope:**
  - GitHub (publicly available Java projects)
  - Time frame: Jan 1 – Oct 31, 2020
  - Build system: maven
  - Cloneable projects (after up to 3 attempts)
  - Project with executable JUnit tests (timeout 1 hour)
  - Projects that execute stream processing



Related Work

Study	Projects analyzed
Our work	620'033’059 projects (132’211 use streams)
Tanaka et al. [1]	100 projects (3 use streams)
Khatchadourian et al. [2]	34 projects using streams
Nostas et al. [3]	10’000 projects (610 use streams)

[1] Tanaka et al. *A Study on the Current Status of Functional Idioms in Java*. IEICE Trans. on Info. and Sys. 2019

[2] Khatchadourian et al. *An Empirical Study on the Use and Misuse of Java 8 Streams*. FASE’20.

[3] Nostas et al. *How Do Developers Use the Java Stream API*. ICCSA’21.

Università della Svizzera italiana

Related Work

Our study is the first to report the popularity of runtime info:

- Source methods
- Source collection types
- Stream types
- Pipeline lengths
- Stream result types

- Characteristics of stream data sources:
  - Size, ordering constraints, duplicate allowance, mutability, and support for concurrent modification

Università della Svizzera italiana

Discussion

- Feedback to developers of the Java class library
- Feedback to IDE and tool builders
- Basis to promote the discussion on how developers can better leverage the Stream API
- Feedback to educators training Java developers