VIRGINIA TECH

UCLA

Co-dependence Aware Fuzzing for Dataflow-based Big Data Analytics

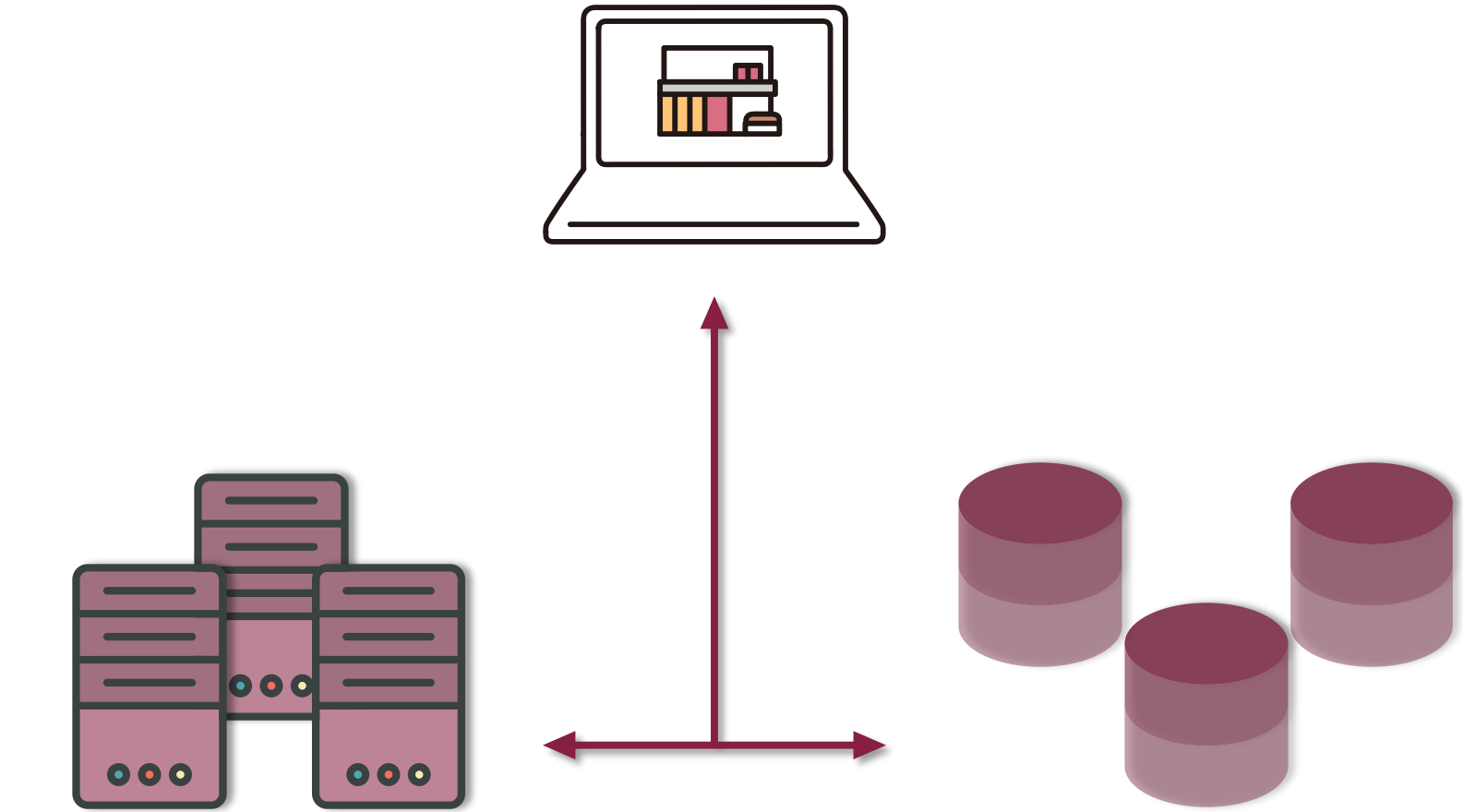
Ahmad Humayun, Miryung Kim, Muhmmad Ali Gulzar

Outline

- Problem: Test input generation for Big Data (DISC) applications
- Programs have complex interactions with data that create inter-data dependencies
- A general solution for data dependent programs

- ➢ Motivating Example
- ➢ Technical Challenges
- ➢ Approach Overview
- ➢ Evaluation

Data Intensive Scalable Computing (DISC)



Distributed Compute

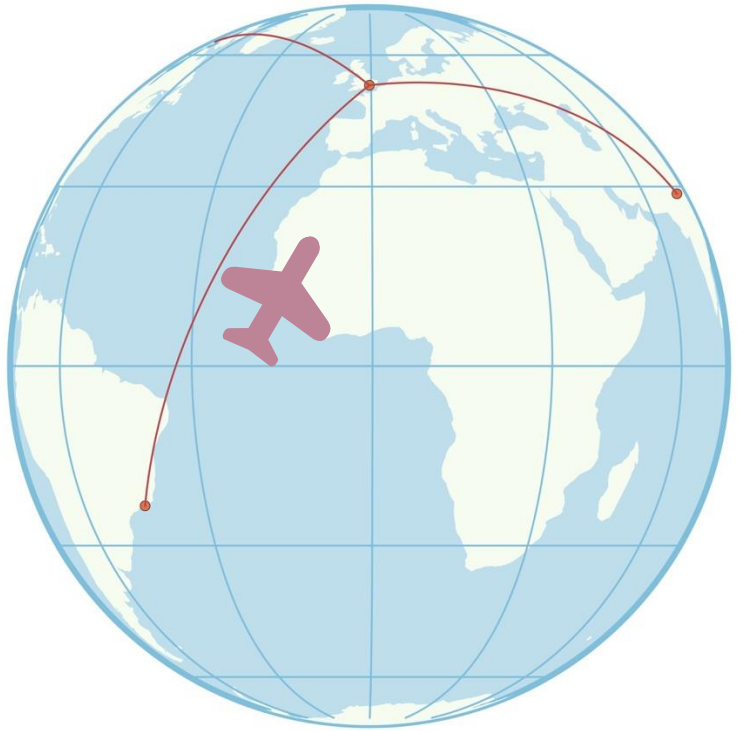
Distributed Data

DISC Application Example

- Compute the distance travelled by a flight, between two airports

flights.csv

```
32556, 08-22 07:25, 08-22 09:50, LAX, SFO
22019, 09-03 15:00, 09-03 02:50, ROA, SFO
31522, 09-03 08:55, 09-03 10:20, IAD, SFO
```



DISC Application Example

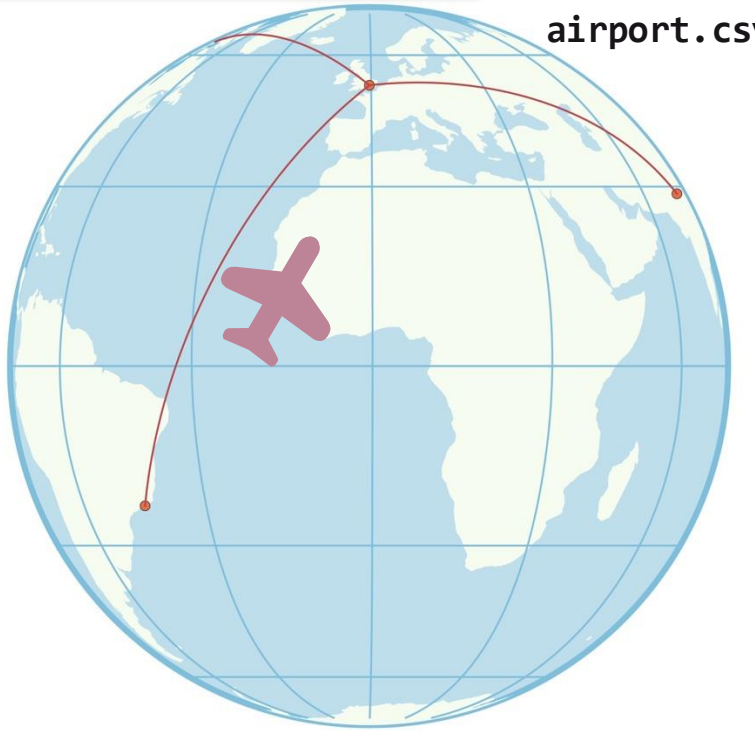
- Compute the distance travelled by a flight, between two airports

flights.csv

```
32556, 08-22 07:25, 08-22 09:50, LAX, SFO
22019, 09-03 15:00, 09-03 02:50, ROA, SFO
31522, 09-03 08:55, 09-03 10:20, IAD, SFO
```

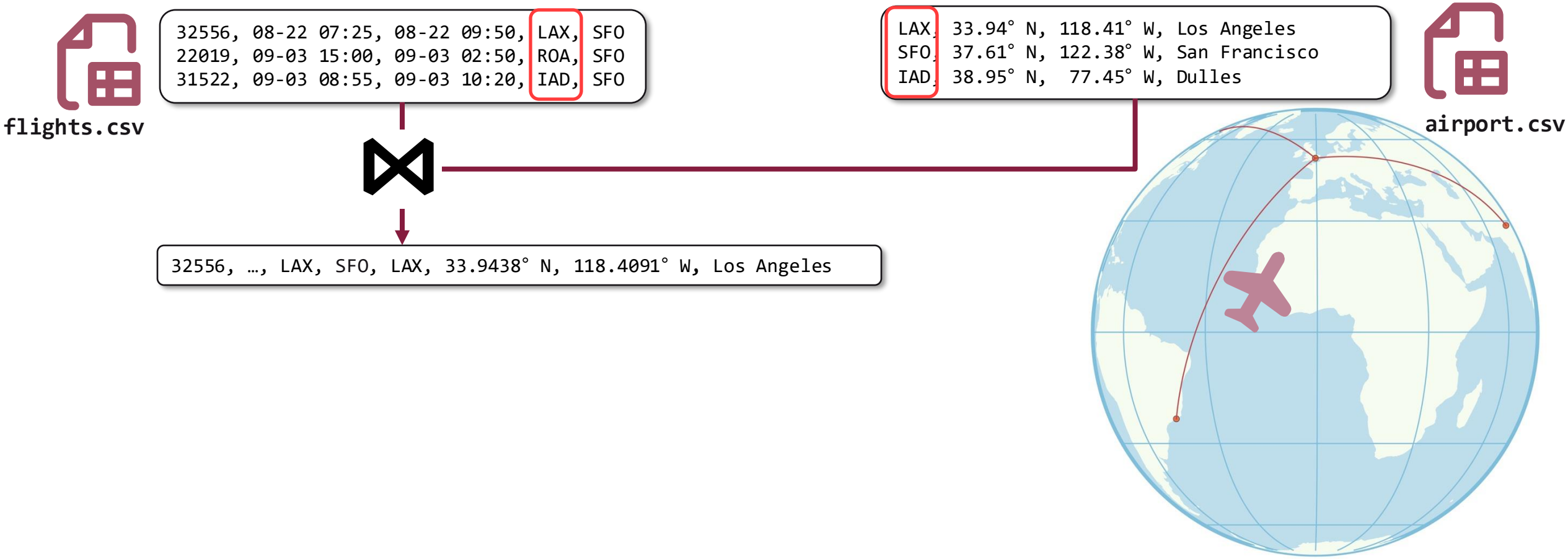
```
LAX, 33.94° N, 118.41° W, Los Angeles
SFO, 37.61° N, 122.38° W, San Francisco
IAD, 38.95° N,  77.45° W, Dulles
```

airport.csv



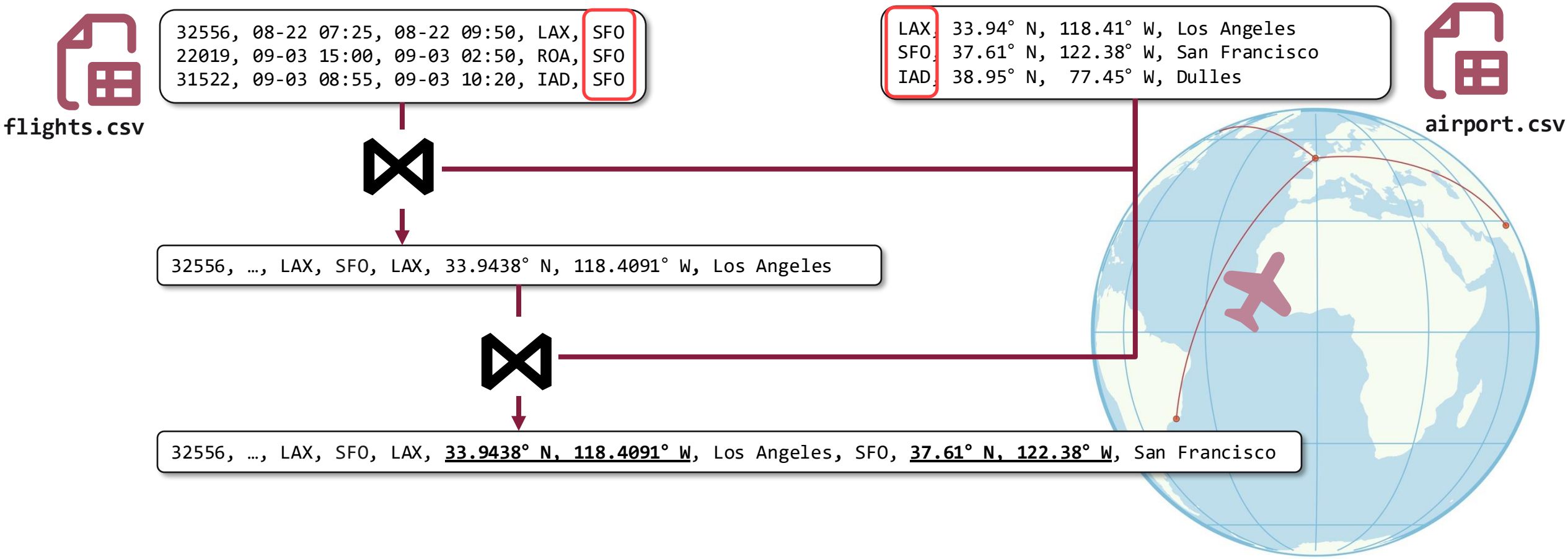
DISC Application Example

- Step 1: Join departure airport codes with airport codes in airports.csv



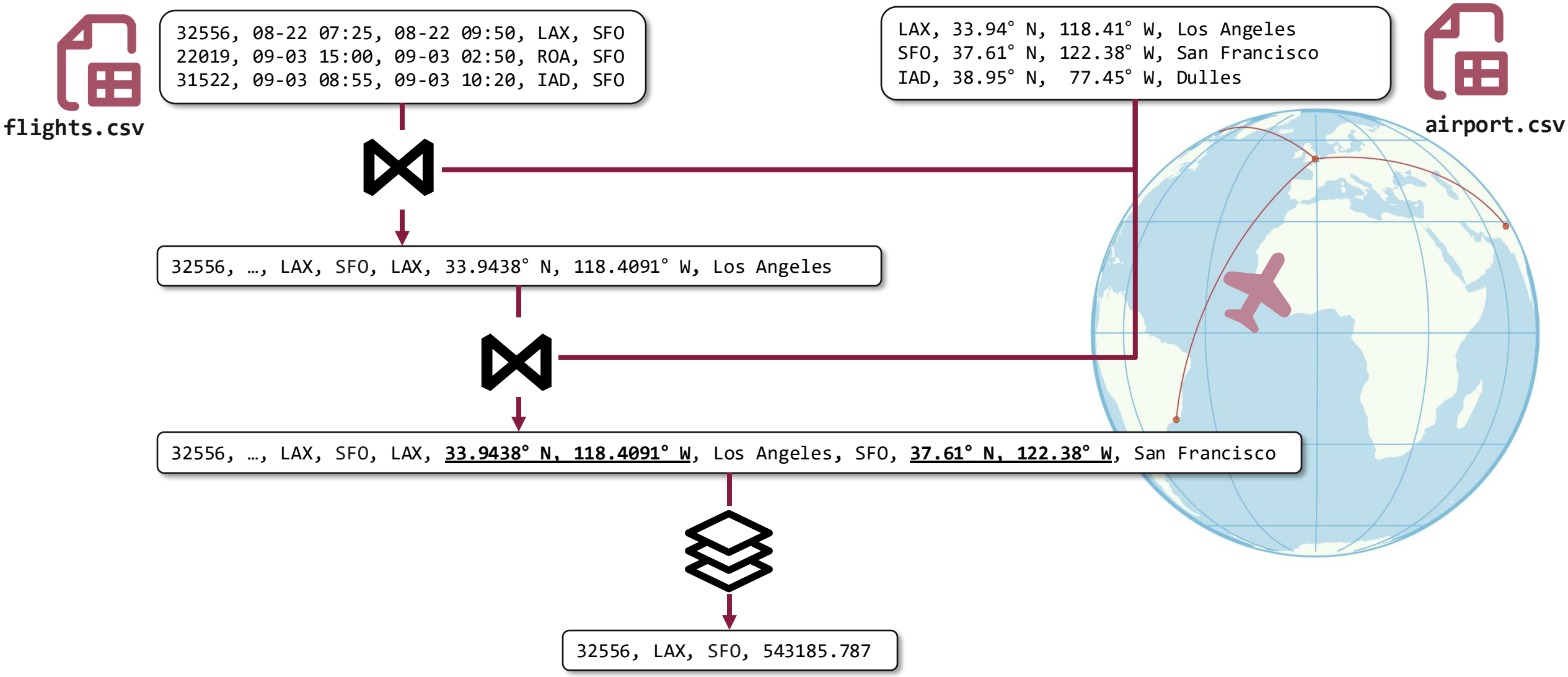
DISC Application Example

- Step 2: Join arrival airport codes with airport codes in airports.csv



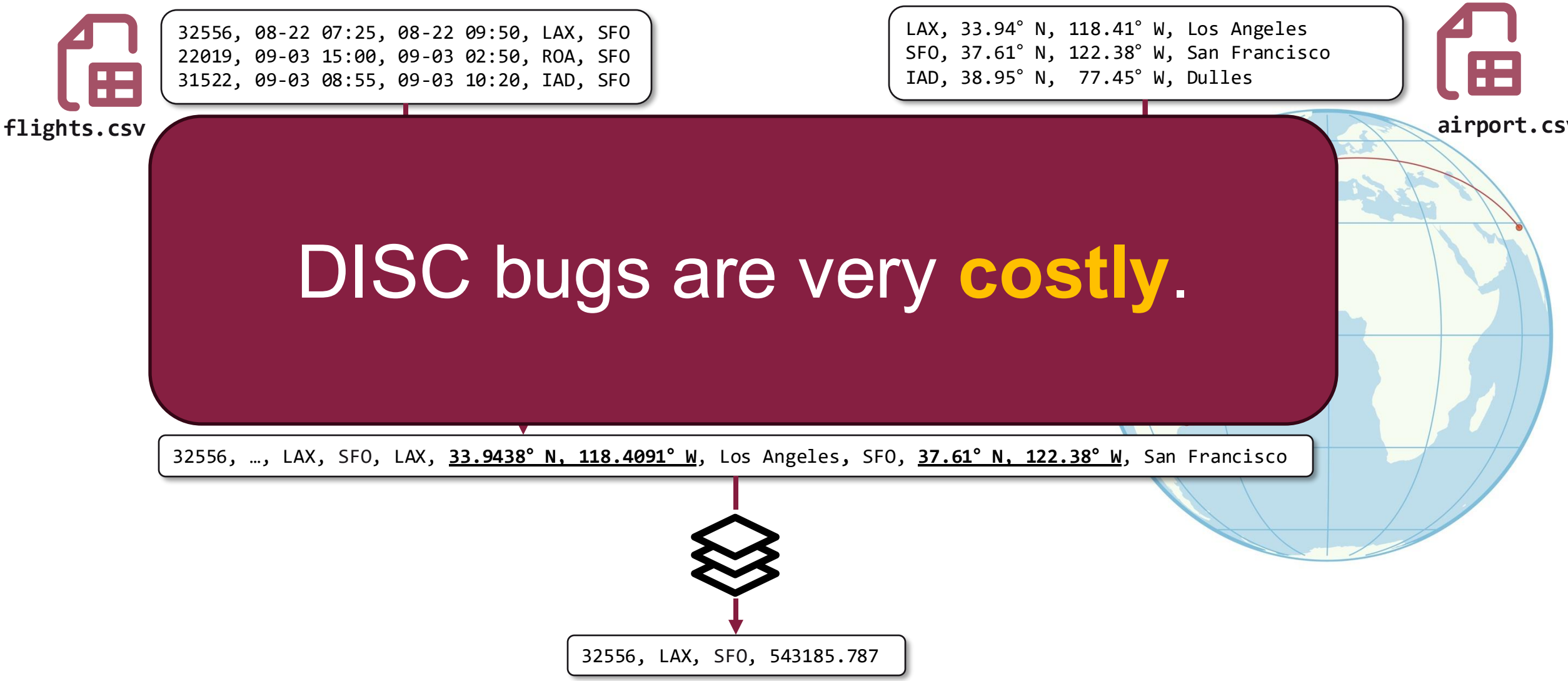
DISC Application Example

- Step 3: Use a map function to apply a formula using the longitude and lattitude values

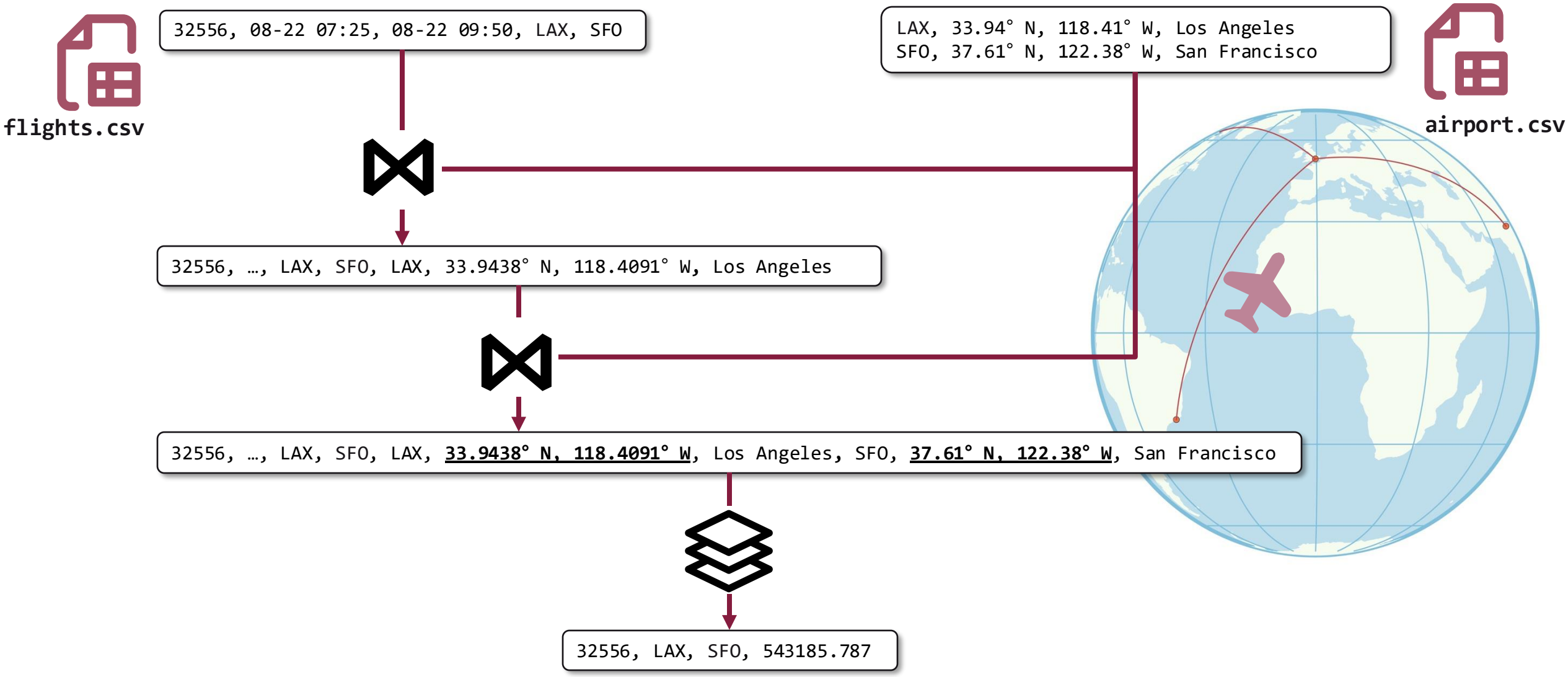


DISC Application Example

- Step 3: Use a map function to apply a formula using the longitude and lattitude values

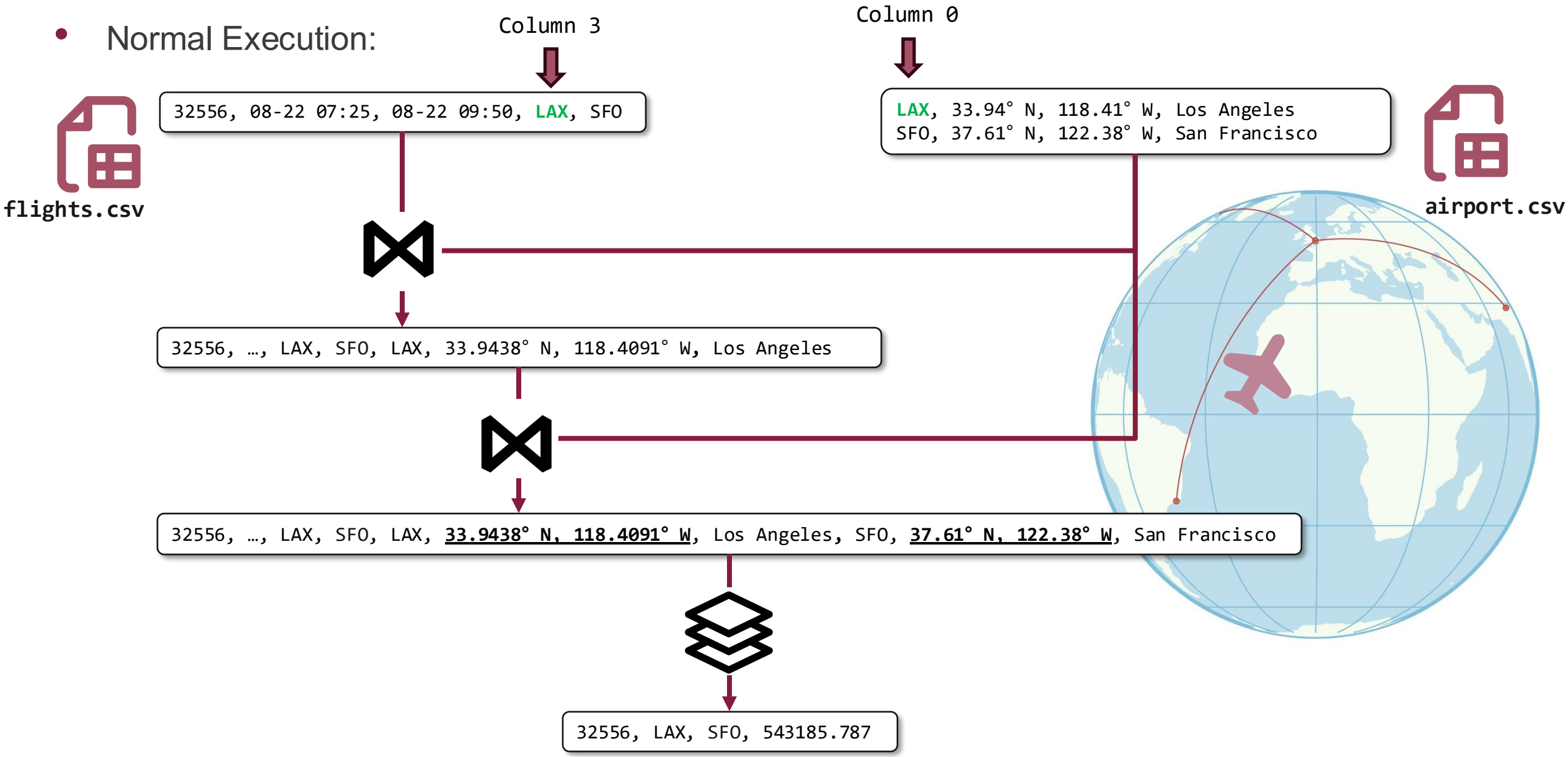


Challenges – Dataflow



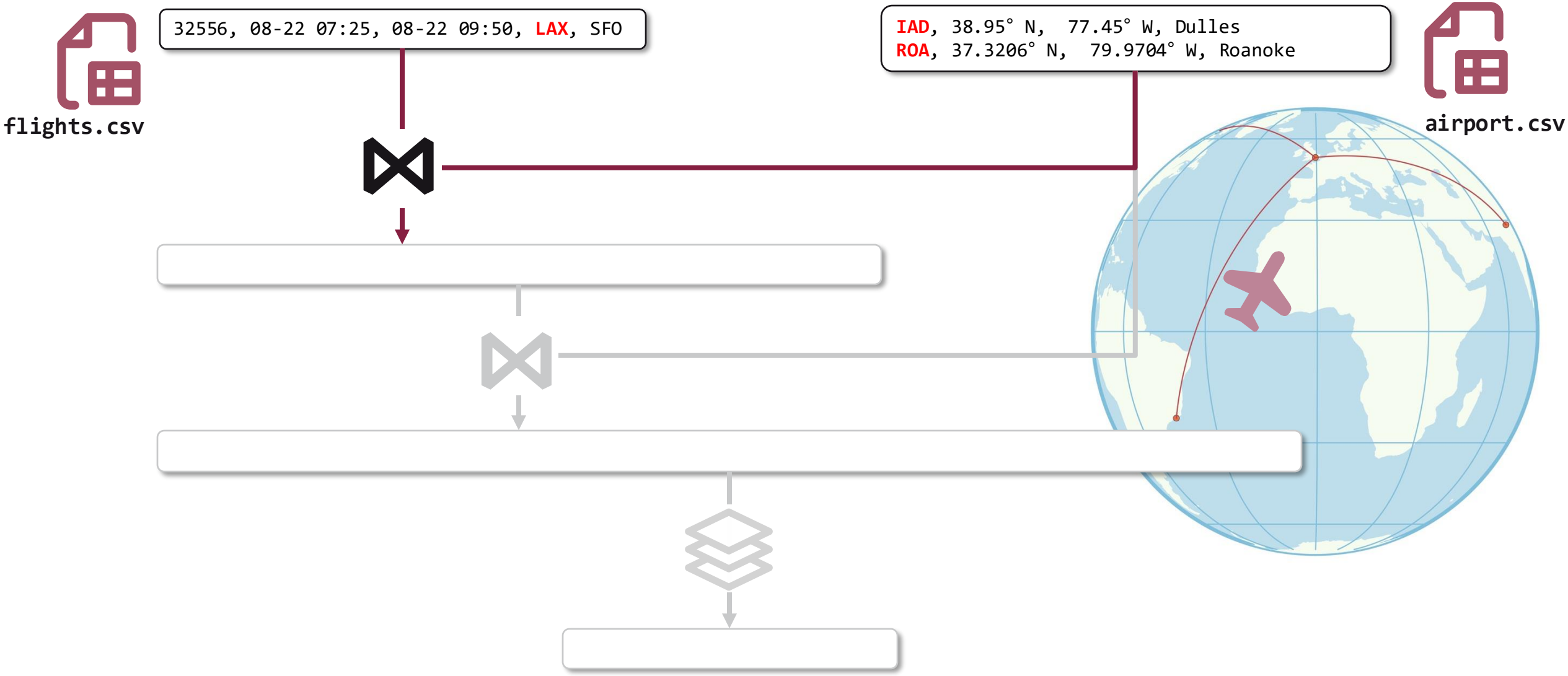
Challenges – Dataflow

- Normal Execution:



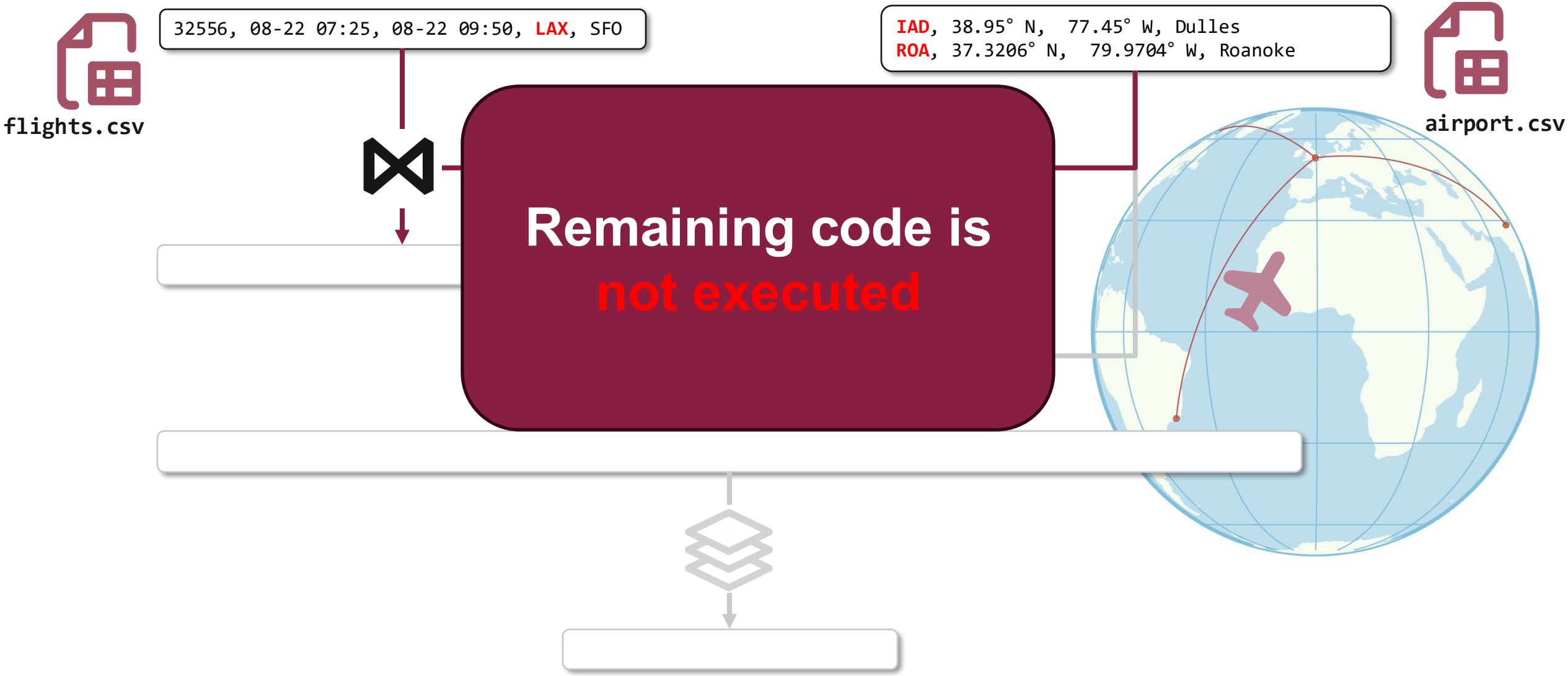
Challenges – Dataflow

- Dataflow terminates **prematurely**

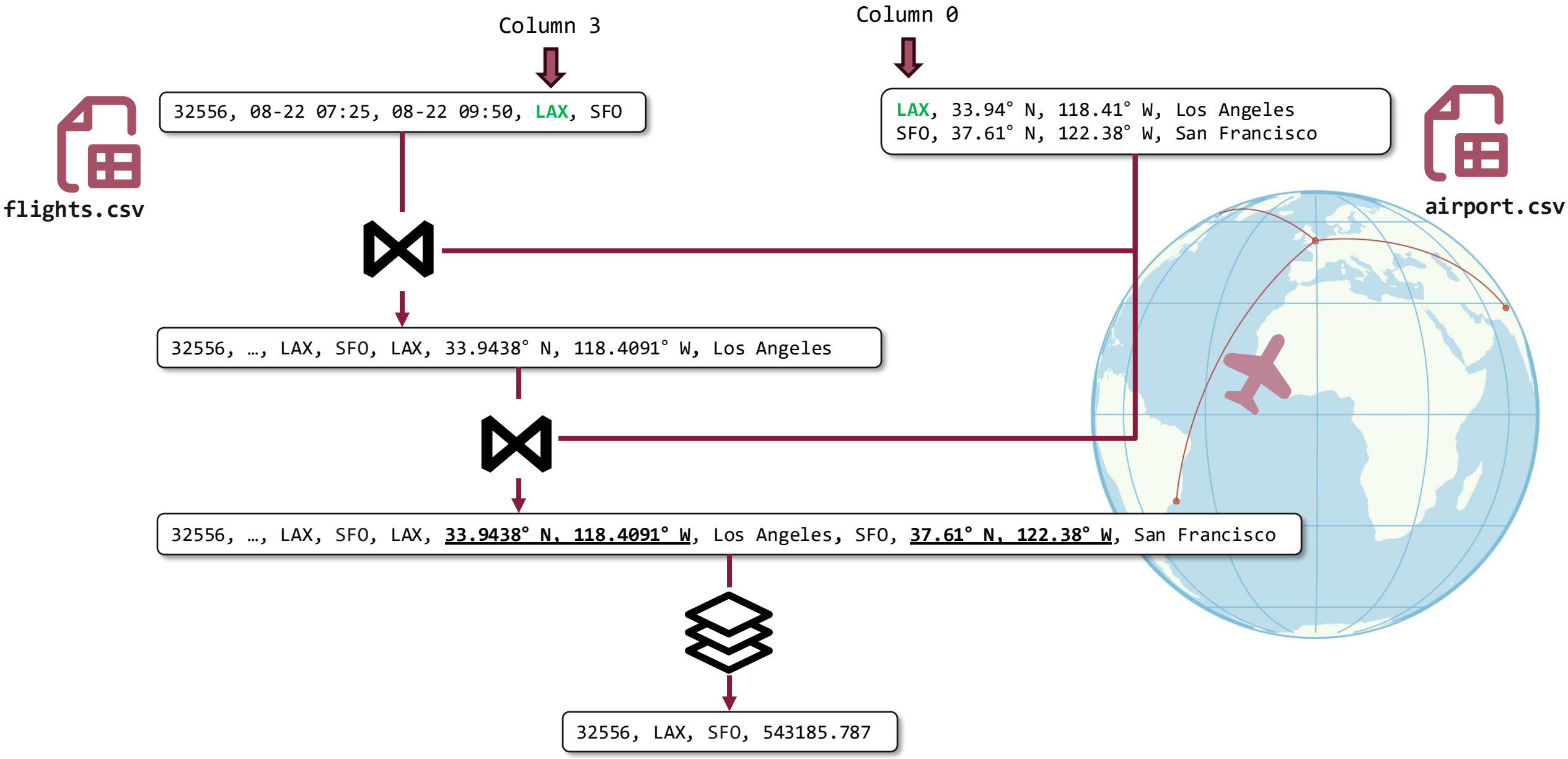


Challenges – Dataflow

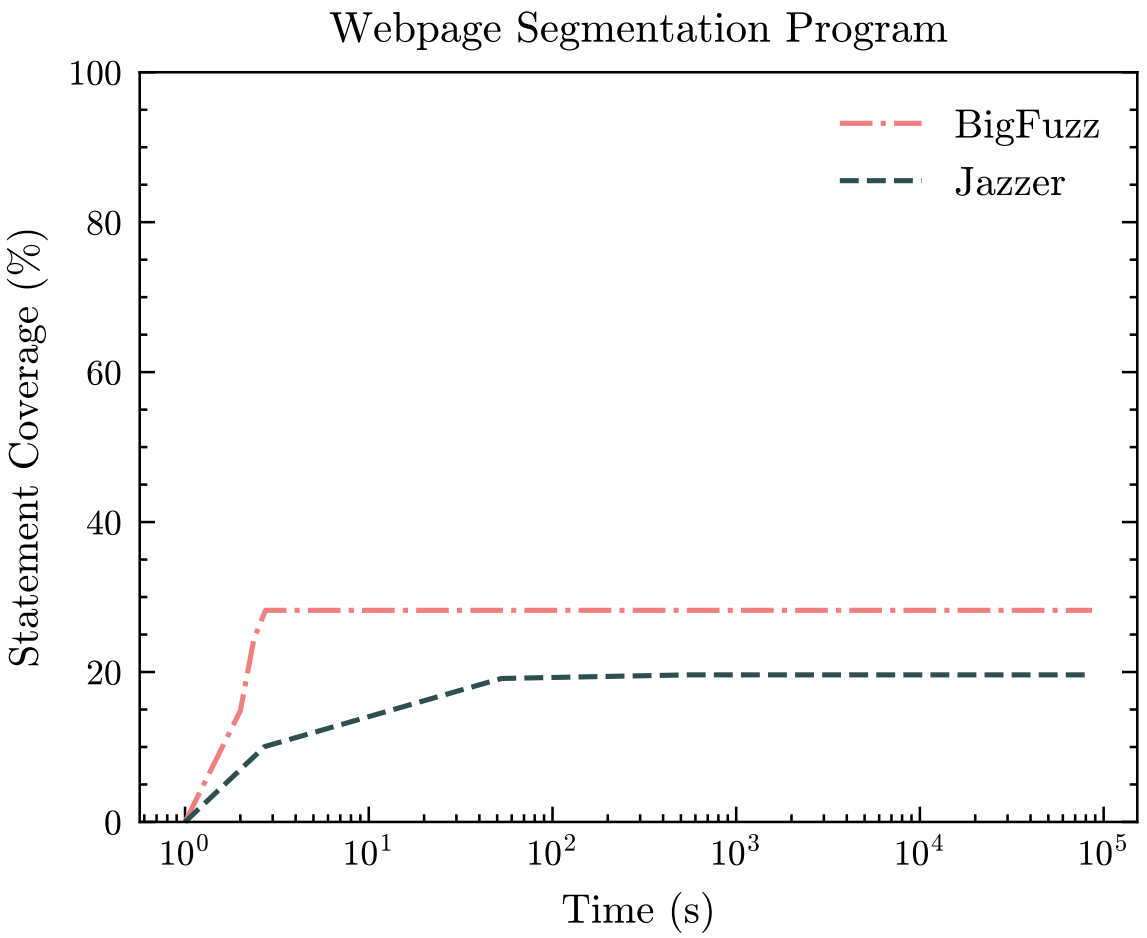
- Dataflow terminates **prematurely**



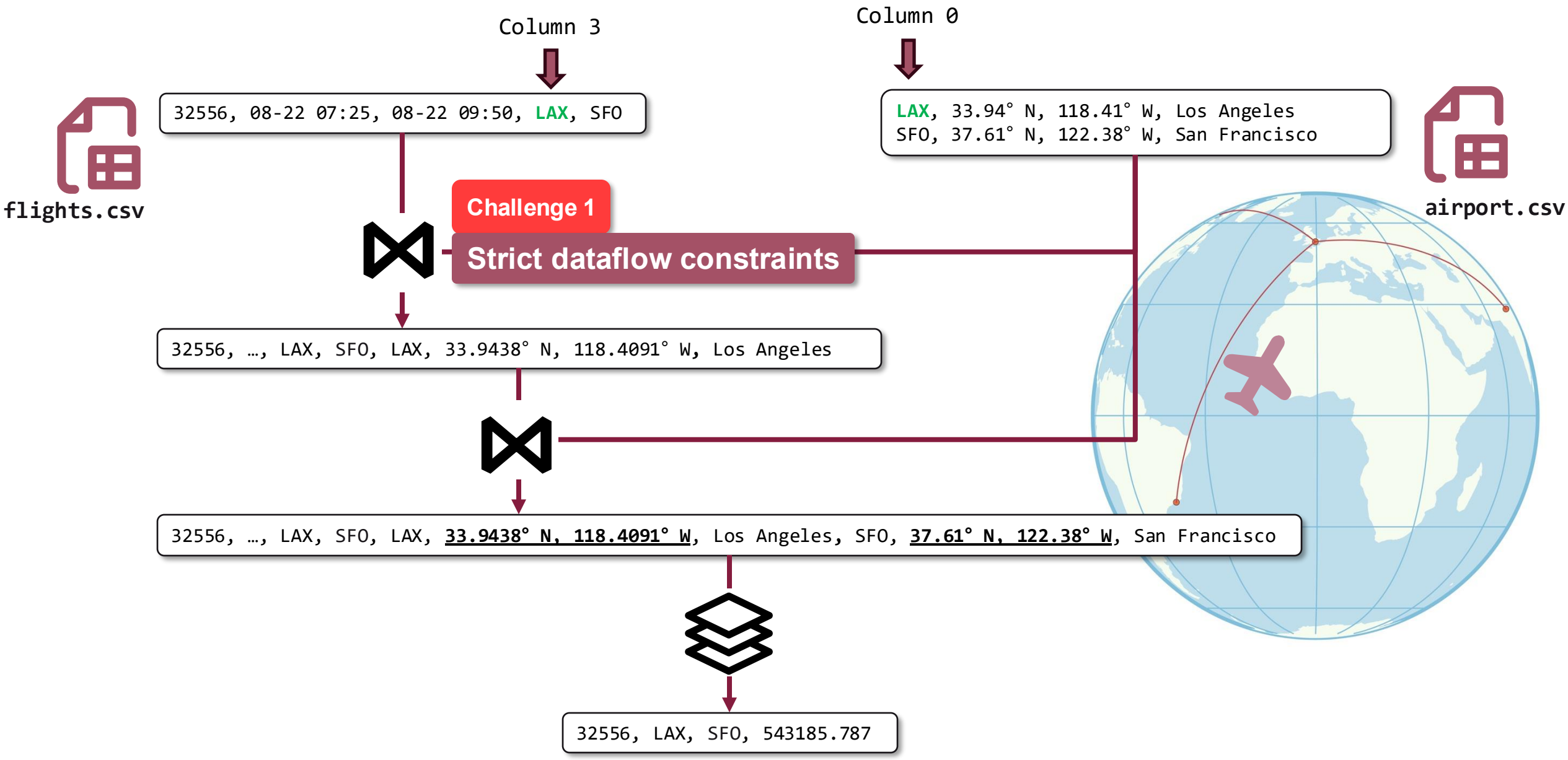
Challenges


Column 3
Column 0
flights.csv
airport.csv
32556, 08-22 07:25, 08-22 09:50, LAX, SFO
LAX, 33.94° N, 118.41° W, Los Angeles
SFO, 37.61° N, 122.38° W, San Francisco
32556, …, LAX, SFO, LAX, 33.9438° N, 118.4091° W, Los Angeles
32556, …, LAX, SFO, LAX, 33.9438° N, 118.4091° W, Los Angeles, SFO, 37.61° N, 122.38° W, San Francisco
32556, LAX, SFO, 543185.787

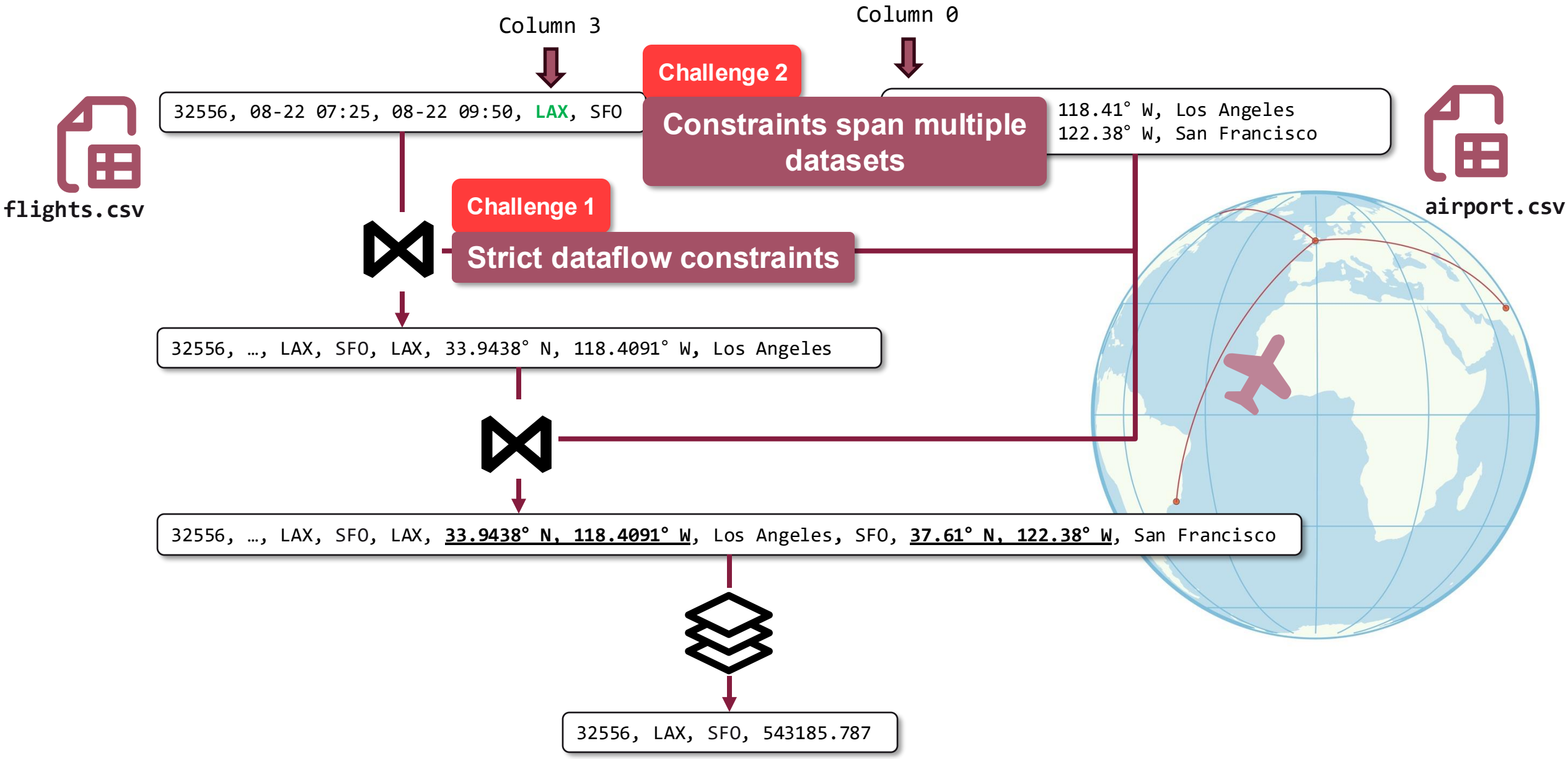
Current State of the Art



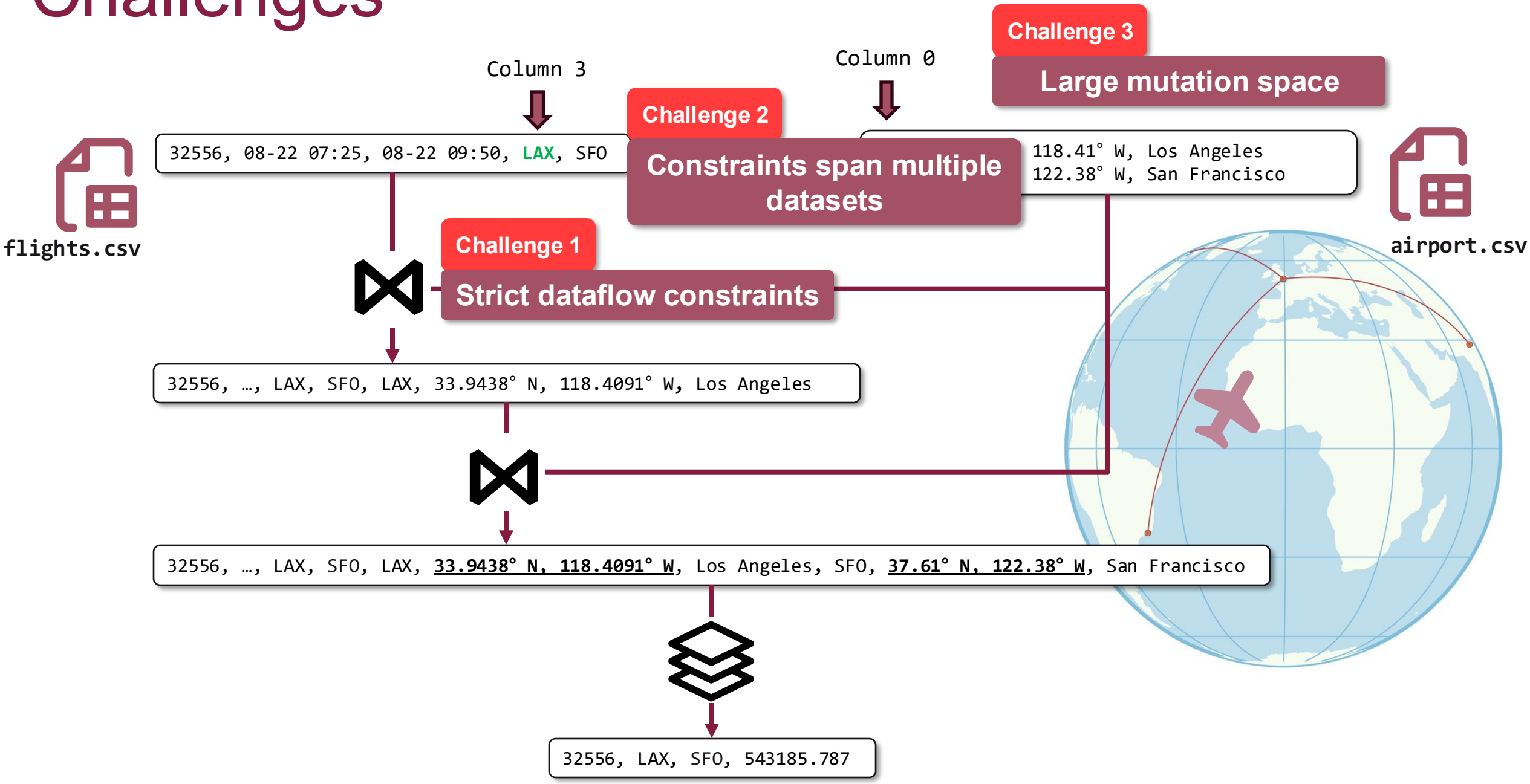
Challenges


Column 3
Column 0
flights.csv
32556, 08-22 07:25, 08-22 09:50, LAX, SFO
LAX, 33.94° N, 118.41° W, Los Angeles
SFO, 37.61° N, 122.38° W, San Francisco
airport.csv
Challenge 1
Strict dataflow constraints
32556, …, LAX, SFO, LAX, 33.9438° N, 118.4091° W, Los Angeles
32556, …, LAX, SFO, LAX, 33.9438° N, 118.4091° W, Los Angeles, SFO, 37.61° N, 122.38° W, San Francisco
32556, LAX, SFO, 543185.787

Challenges

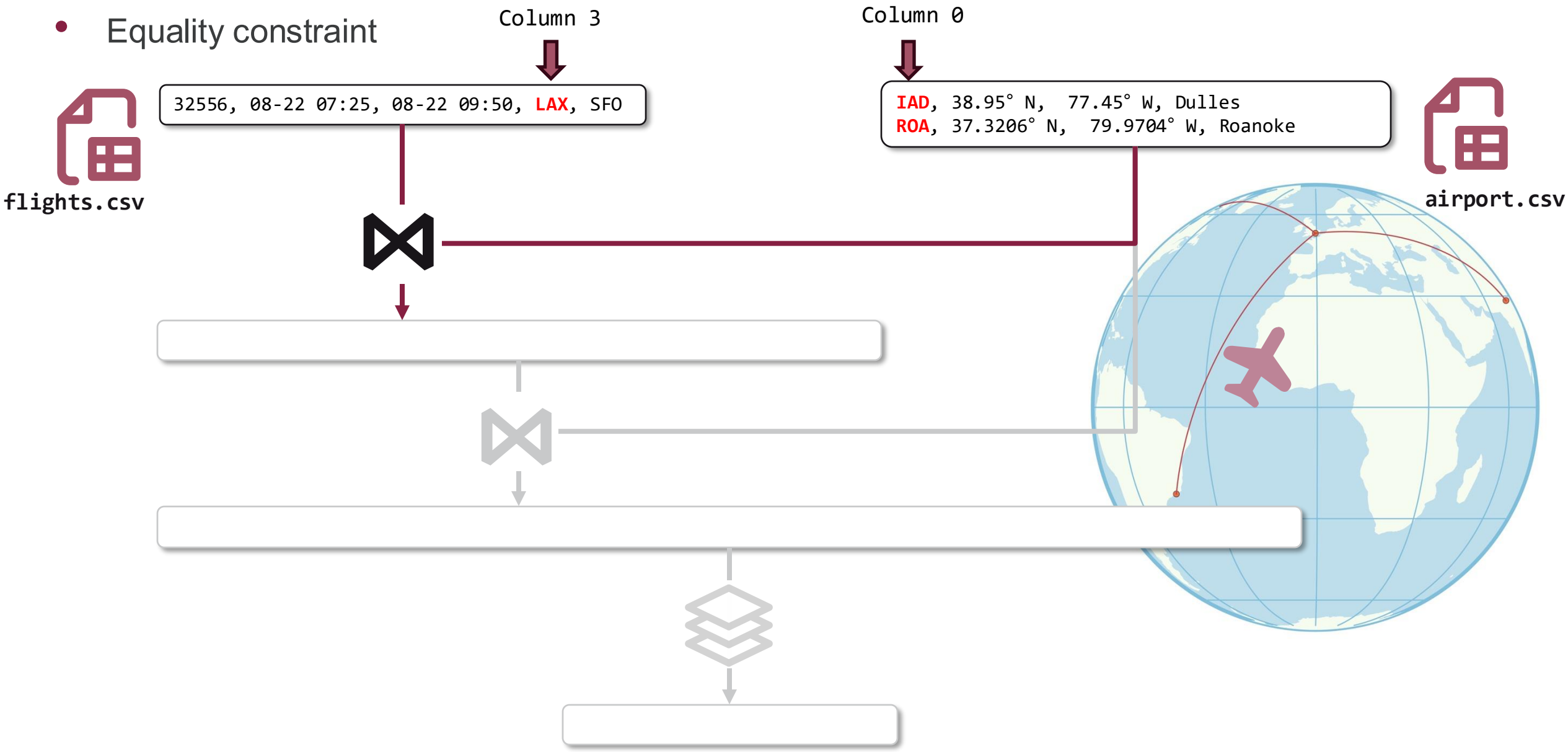

Column 3
Column 0
Challenge 2
Constraints span multiple datasets
flights.csv
32556, 08-22 07:25, 08-22 09:50, LAX, SFO
118.41° W, Los Angeles
122.38° W, San Francisco
airport.csv
Challenge 1
Strict dataflow constraints
32556, …, LAX, SFO, LAX, 33.9438° N, 118.4091° W, Los Angeles
32556, …, LAX, SFO, LAX, 33.9438° N, 118.4091° W, Los Angeles, SFO, 37.61° N, 122.38° W, San Francisco
32556, LAX, SFO, 543185.787

Challenges


Challenge 3
Large mutation space
Column 3
Column 0
Challenge 2
Constraints span multiple datasets
32556, 08-22 07:25, 08-22 09:50, LAX, SFO
118.41° W, Los Angeles
122.38° W, San Francisco
flights.csv
airport.csv
Challenge 1
Strict dataflow constraints
32556, …, LAX, SFO, LAX, 33.9438° N, 118.4091° W, Los Angeles
32556, …, LAX, SFO, LAX, 33.9438° N, 118.4091° W, Los Angeles, SFO, 37.61° N, 122.38° W, San Francisco
32556, LAX, SFO, 543185.787

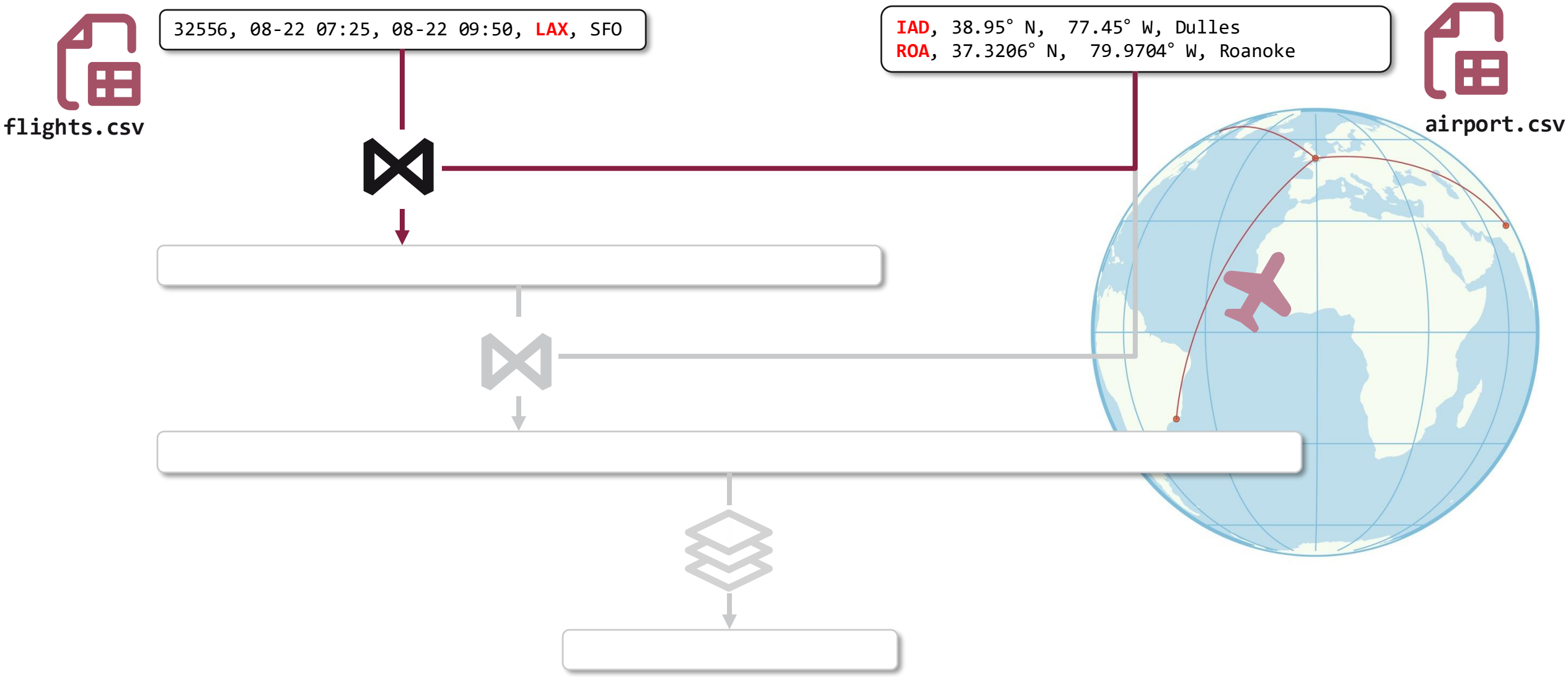
What is Co-Dependence? Intuition

- Equality constraint



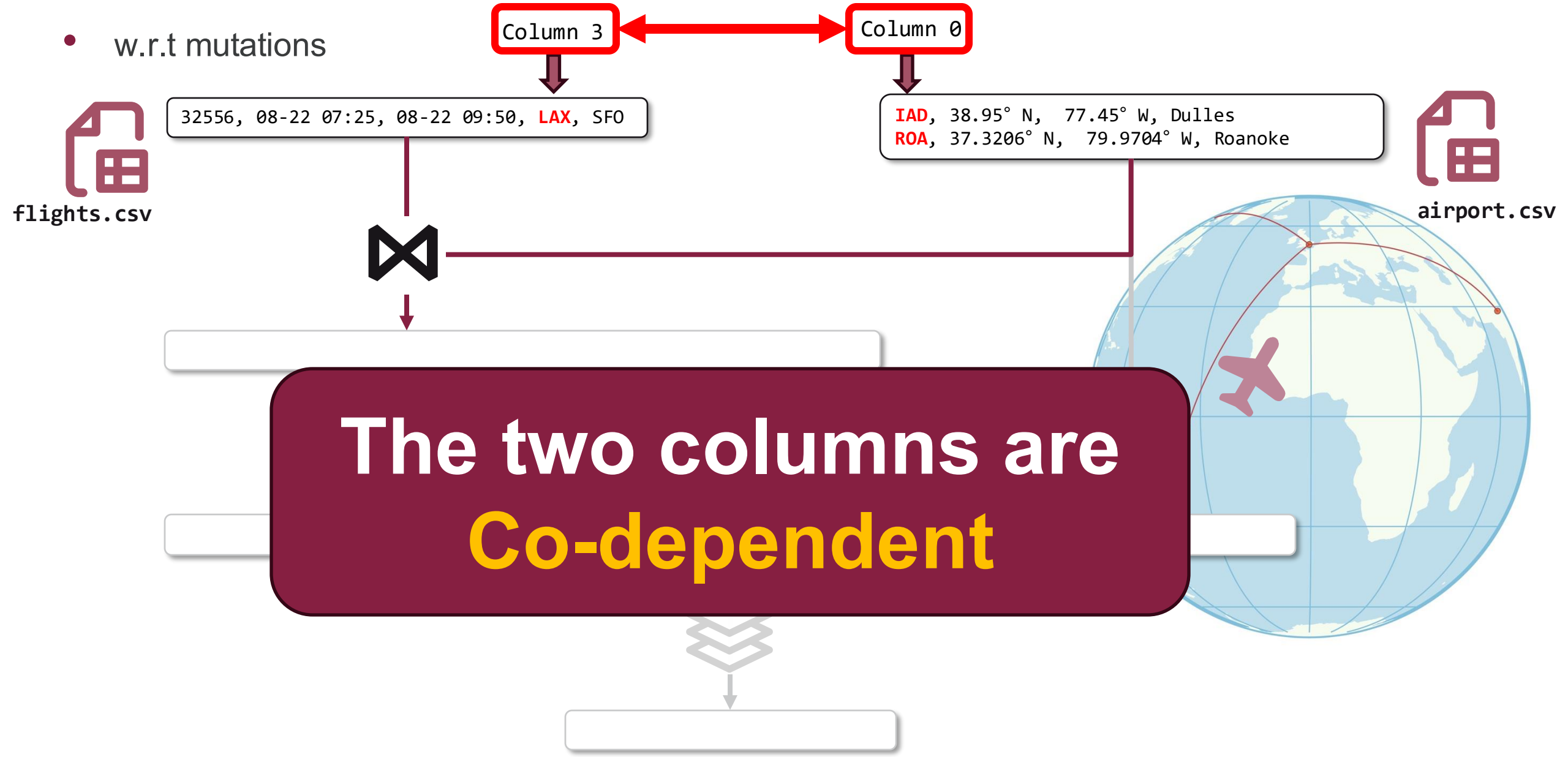
What is Co-Dependence?

- For optimal performance, fuzzers should respect this constraint



What is Co-Dependence?

- w.r.t mutations



What is Co-Dependence?

Column 3 ⟷ Column 0

flights.csv

32556, 0

ulles
N, Roanoke

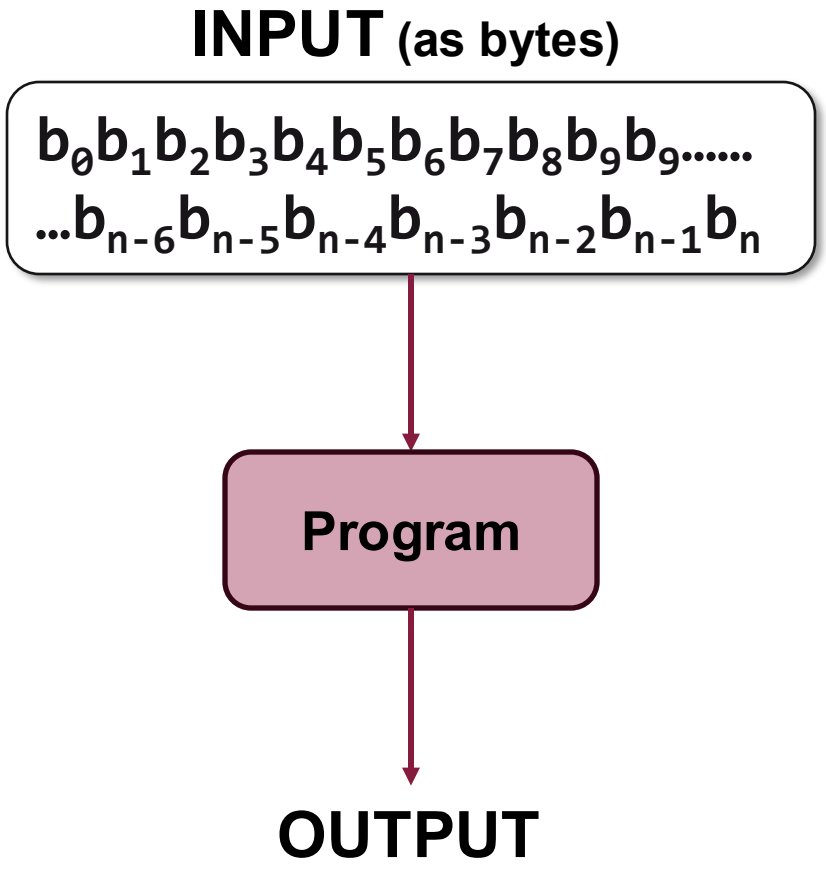
airport.csv

Can we generalize this idea?

The two columns are Co-dependent

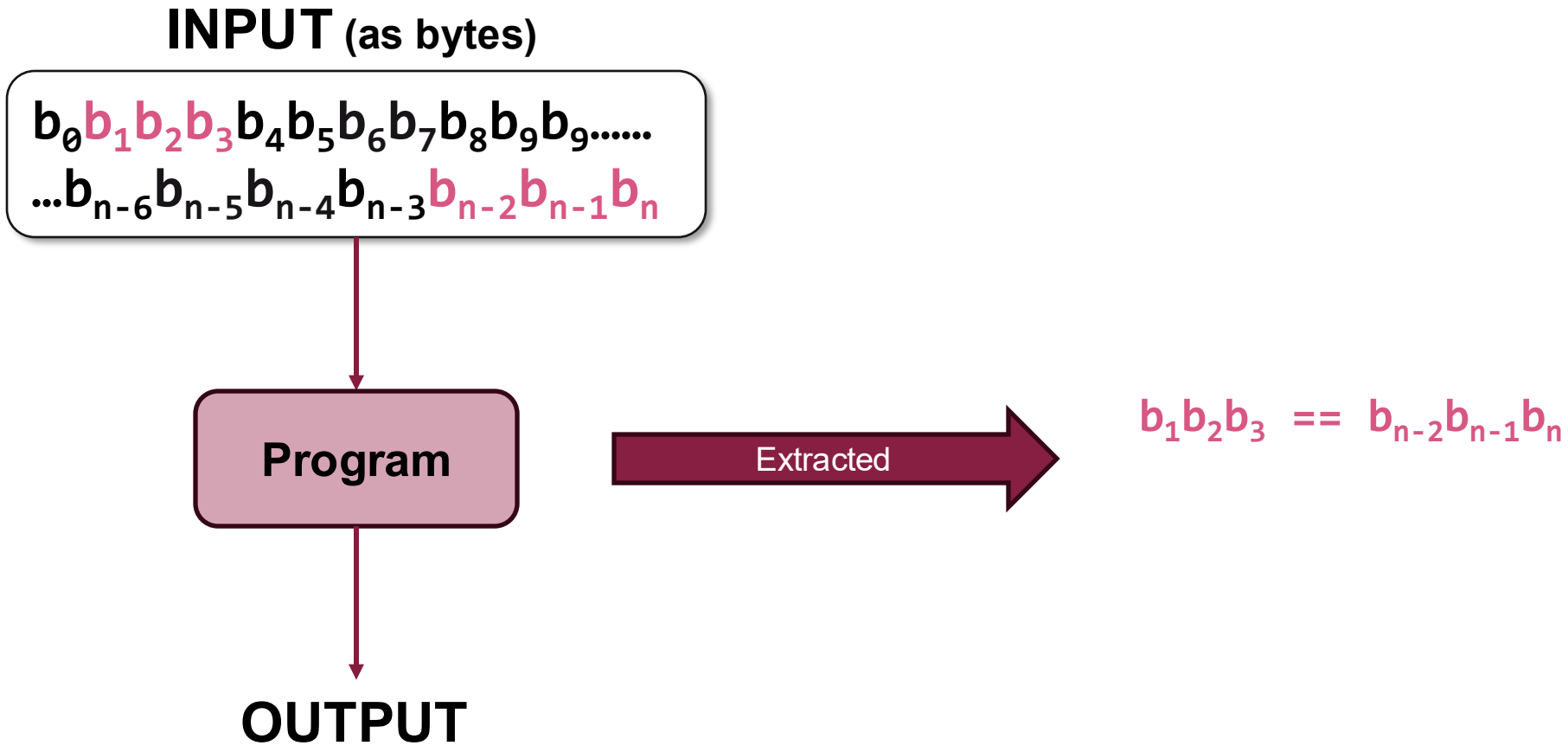
Generalizing Co-Dependence

- An input is a blob of bytes



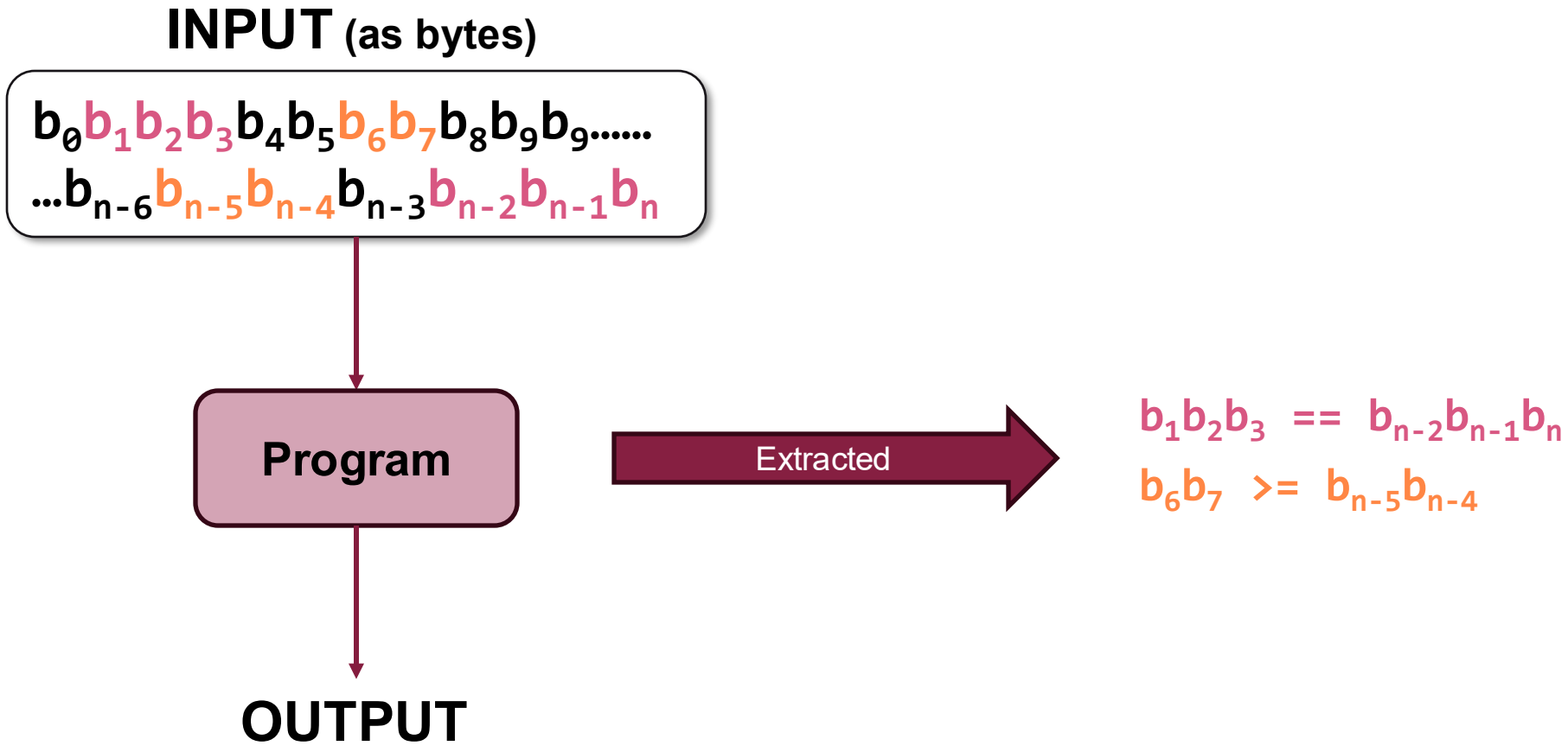
Generalizing Co-Dependence

- Two byte-regions of the input that are related to each other by some operation



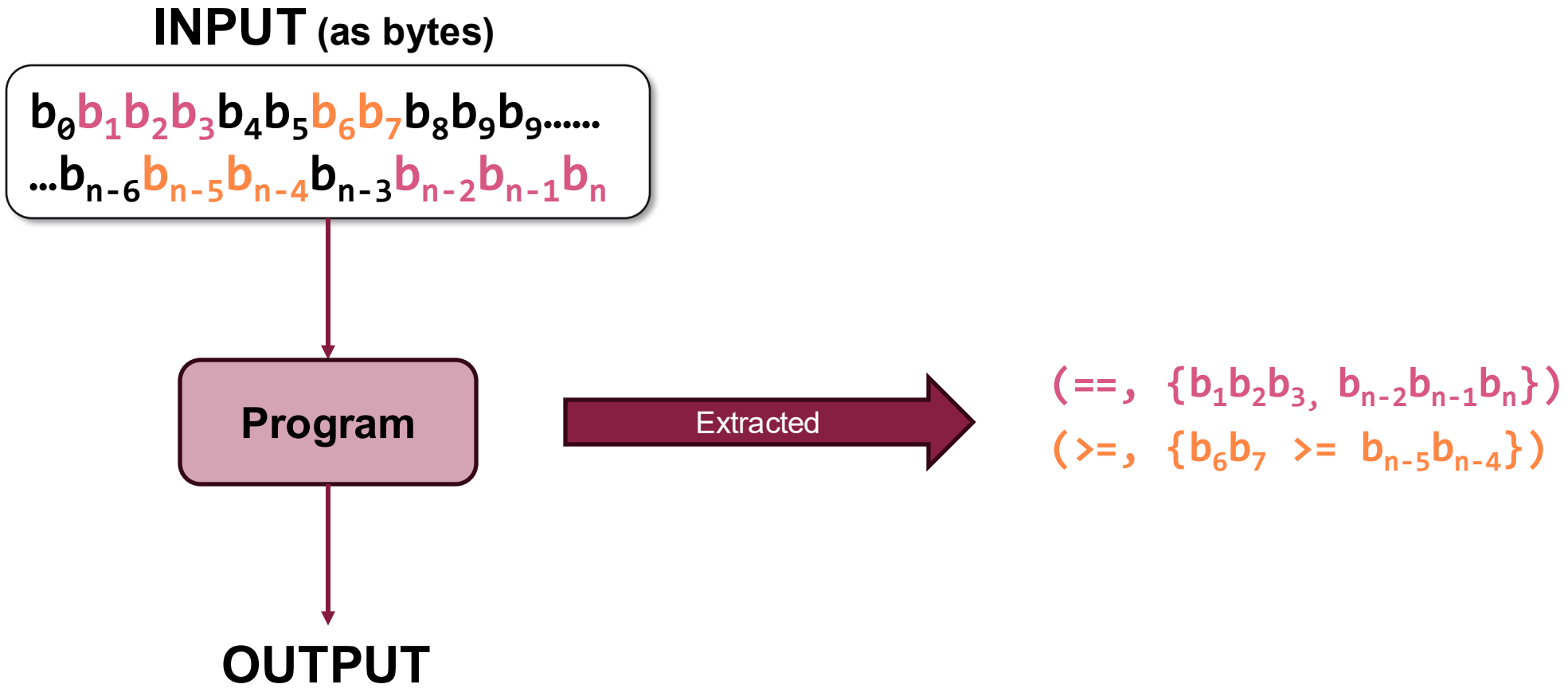
Generalizing Co-Dependence

- Two byte-regions of the input that are related to each other by some operation



Generalizing Co-Dependence

- Two byte-regions of the input that are related to each other by some operation



Generalizing Co-Dependence

other (*i.e.*, input and output of an operator). Thus, we define a DISC application's DFG, G, as

$$G = \langle O \cup N, E \rangle$$

where $O = \{o_1, o_2, ..., o_n\}$ is a set of operators, $N = \{n_1, n_2, ..., n_m\}$ is a set of datanodes. $E \subseteq (O \times N) \cup (N \times O)$ is a set of directed edges connecting operators with datanodes. An atomic unit of this DFG has three nodes and two edges *i.e.*, an operator with an incoming edge from an input datanode and an outgoing edge to an output datanode. Furthermore, a datanode n holds data in the form of a *byte sequence* $b_1 b_2 ... b_k$. Let $D(n)$ be the set of all possible subsequences of the byte sequence in a datanode n, *i.e.*, $\{b_1, b_2, ..., b_1 b_2, ..., b_1 ... b_k\}$. Input datasets of DISC applications are defined as S, a set of initial datanodes which are external inputs to the DFG. We combine regions in input datasets in S', a union of $D(n)$ across all input datanodes.

$$S' = \bigcup_{\forall n \in S} D(n)$$

Finally, we characterize co-dependency among input regions as a set of tuples, C

$$C = \{(o, R) \mid R \subseteq S', \forall o \in O\}$$

The first element, o, is an operator in the dataflow graph; and the second element, R, is a subset of the regions in the input datasets that are co-dependent due to operator o. Let $I(o)$ be the incoming data to an operator o. Since co-dependence can only occur between regions of the original datasets, we must extract R from $I(o)$, which can be any arbitrary byte sequence in the incoming datanode of operator o. To extract such information, we define *monitors* that are concretely explained in Section 3.1.

$$M_o : I(o) \rightarrow \mathcal{P}(S')$$

where $\mathcal{P}(S')$ is the powerset of S'. A monitor, M_o, is an operator-specific function that takes $I(o)$ as input and outputs a set of byte

Table 1: Summary of how each class of operators produces co-dependent regions in the input dataset. For simplicity, we use `row[1].col[3]` as a human-readable representation of input byte region $b_i, ...b_j$, where $0 < i < j < size(dataset)$

Operator Class	Sample Operators	Example of Identified Constraint	Mutation strategy

Fusions	`data1.join(data2)`
	`data1.intersection(data2)`
	`data1.cogroup(data2)`
		$M_{join}(\{data1, data2\})$.
		Possible output of M_{join}:
		`{data1.row[1].col[3], data2.row[23].col[0]}`
		`{data1.row[31].col[3], data2.row[52].col[0]}`
		Co-dependence tuples:
		`(== , {data1.row[1].col[3], data2.row[23].col[0]})`
		`(== , {data1.row[31].col[3], data2.row[52].col[0]})`
			Any mutation applied to `data1.row[1].col[3]` must also be applied to `data2.row[23].col[0]`. If no rows with matching keys exists, select a row from `data1` and copy `data1.col[3]` to `data2.col[0]`.

Aggregations	`data.aggregateByKey(udf)`
	`data.reduceByKey(udf)`
	`data.groupByKey()`
	`data.countByKey()`
		$M_{reduceByKey}(\{data\})$.
		Possible output of $M_{reduceByKey}$:
		`{data.row[2].col[2],`
		`data.row[43].col[2],`
		`data.row[63].col[2]}`
		Co-dependence tuple:
		`(== , {data.row[2].col[2],`
		`data.row[43].col[2],`
		`data.row[63].col[2]})`
			Any mutation applied to `data[row=2,col=2]` must also be applied to `data[row=43,col=2]`. Duplicate rows and apply same mutation to key columns of duplicates.

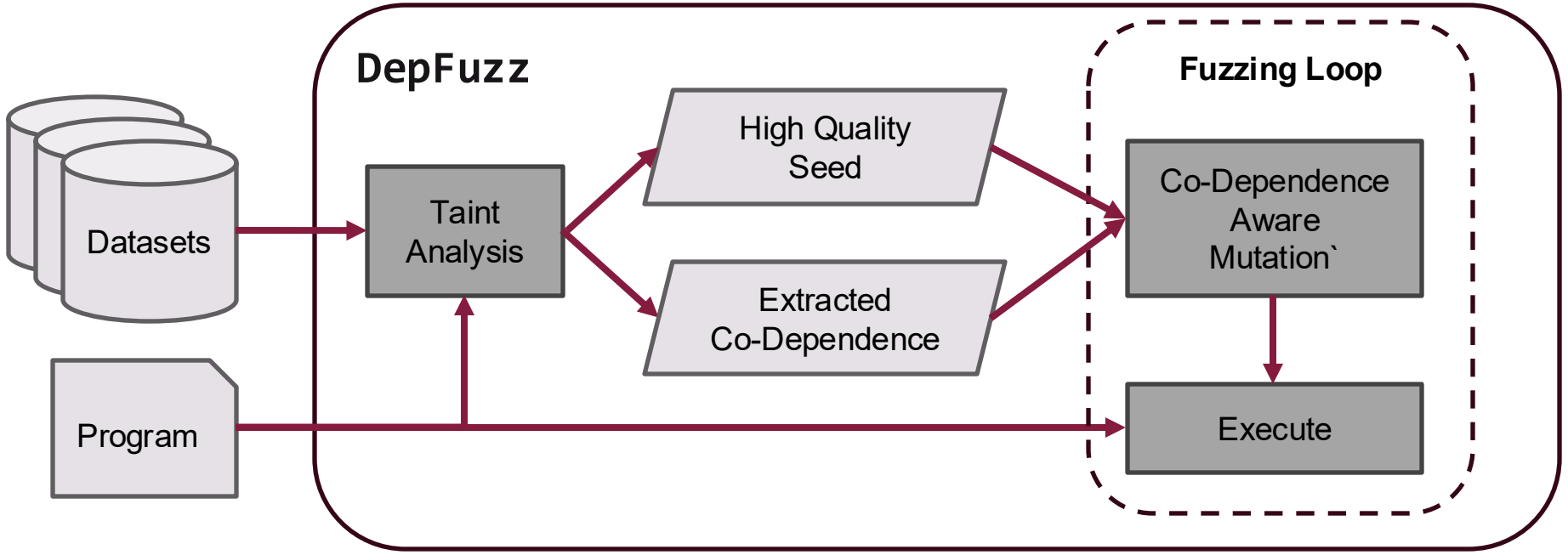
Filters	`data.filter(col0 > col5)`	$M_{filter}(\{data\})$.
		Possible output of M_{filter}:
		`{data.row[23].col[0], data2.row[23].col[5]}`
		`{data.row[31].col[0], data2.row[31].col[5]}`
		Co-dependence tuples:
		`(> , {data1.row[23].col[0], data2.row[23].col[5]})`
		`(> , {data1.row[31].col[3], data2.row[31].col[5]})`
			Any mutation applied to `data.col[0]` and `data.col[5]` must ensure that there is a true and false row for the predicate.

UDF Operators	`if(a.contains(b))`
	`if(a != b)`
	`if(a > b)`
		$M_{contains}(\{a, b\})$.
		Possible output of $M_{contains}$:
		`{data.row[1].col[0], data2.row[1].col[2]}`
		Co-dependence tuples:
		`(contains , {data1.row[1].col[0], data2.row[1].col[2]})`
			Any mutation applied to `data.col[0]` and `data.col[2]` must ensure that string a contains b for some mutations. It must also ensure it occasionally creates inputs that violate this.

Generalizing Co-Dependence

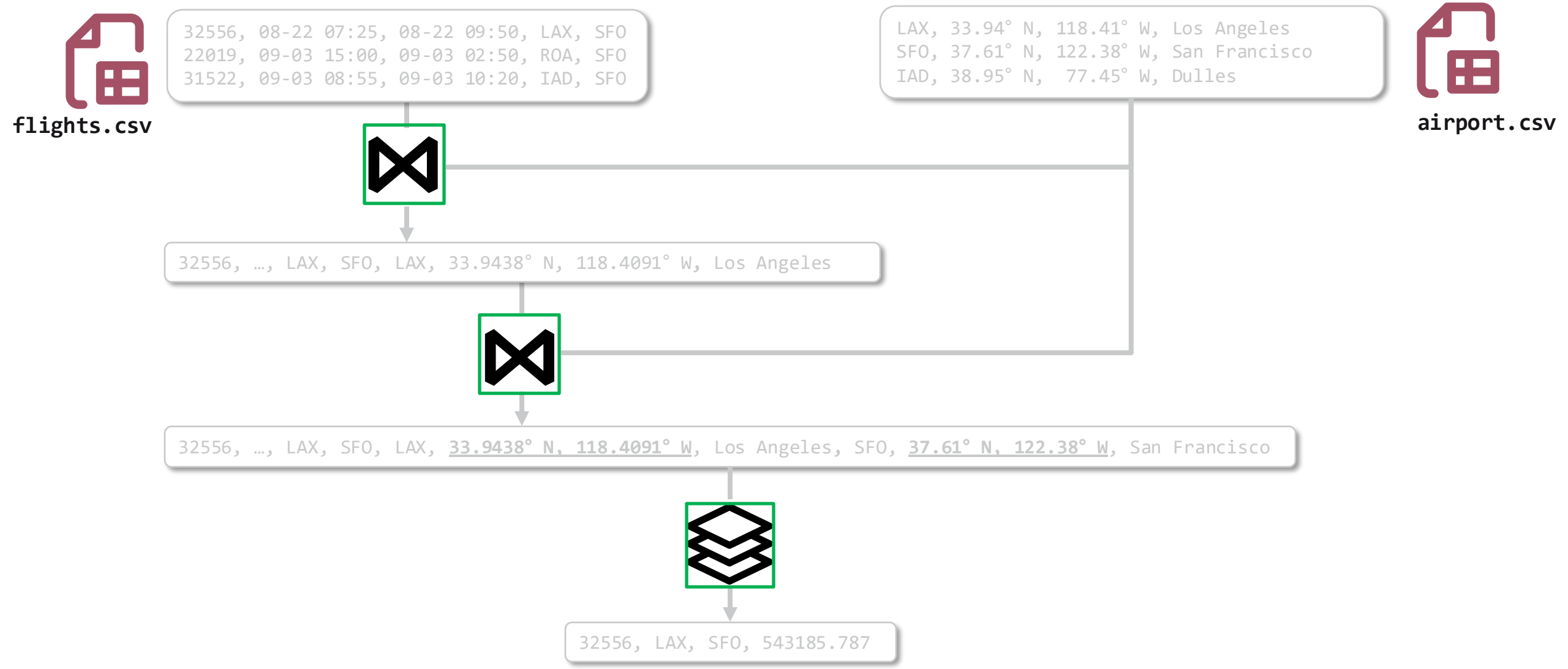
- Two byte-regions of the input that are related to each other by some operation

Approach Overview



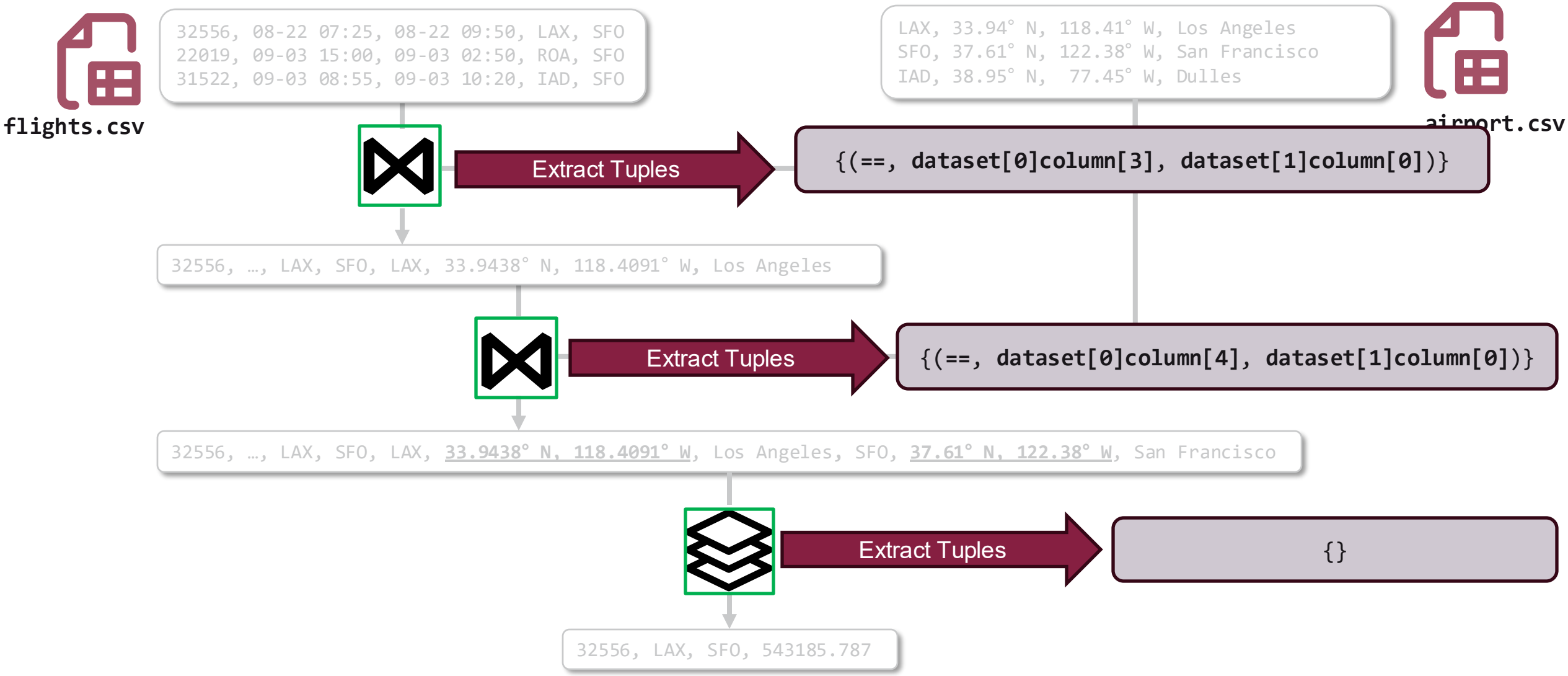
Capturing Co-Dependence

The **monitor** abstraction



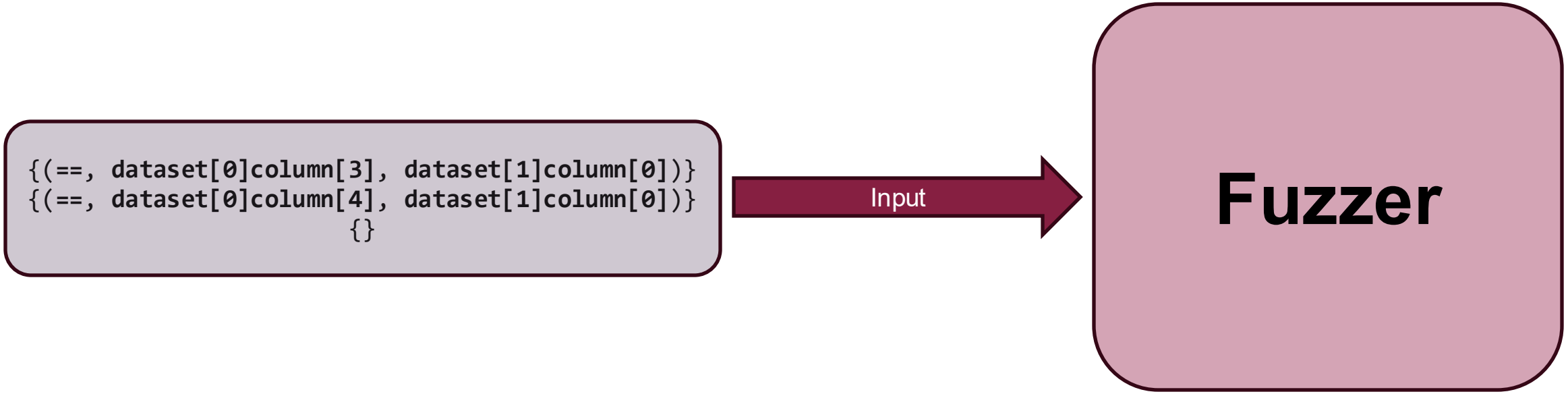
Capturing Co-Dependence

The **monitor** abstraction



Fuzzing

The fuzzer can use this representation perform better mutations



Representation

- This representation abstracts the code from the fuzzer

```
(Operator1, {Region A, Region B})
(Operator2, {Region B, Region C})
```

Representation

- Transitive co-dependencies can be merged

```
(==, {Region A, Region B})
(==, {Region B, Region C})
```

Representation

- Transitive co-dependencies can be merged

```
(==, {Region A, Region B})
(==, {Region B, Region C})
```

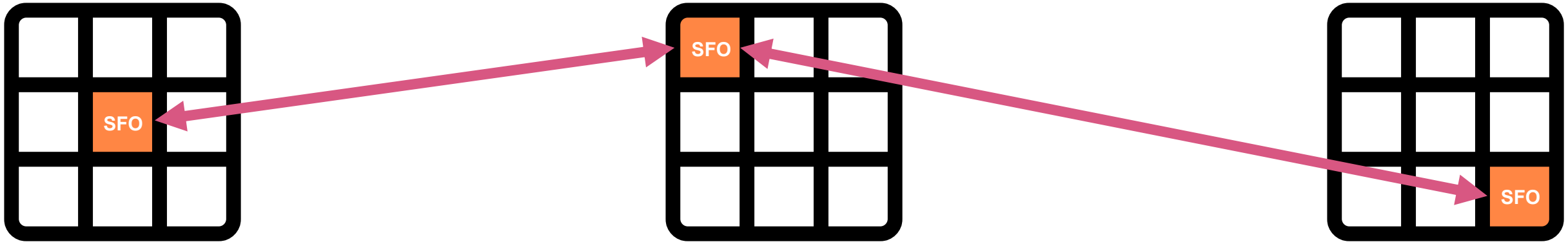
```
{(==, {Region A, Region B, Region C})}
```

Representation

- Transitive co-dependencies can be merged

```
(==, {Region A, Region B})
(==, {Region B, Region C})
```

```
{(==, {Region A, Region B, Region C})}
```

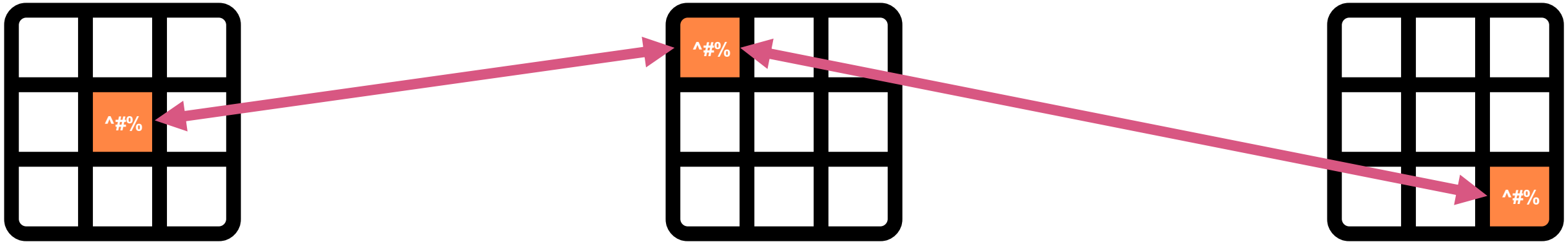


Representation

- Transitive co-dependencies can be merged

```
(==, {Region A, Region B})
(==, {Region B, Region C})
```

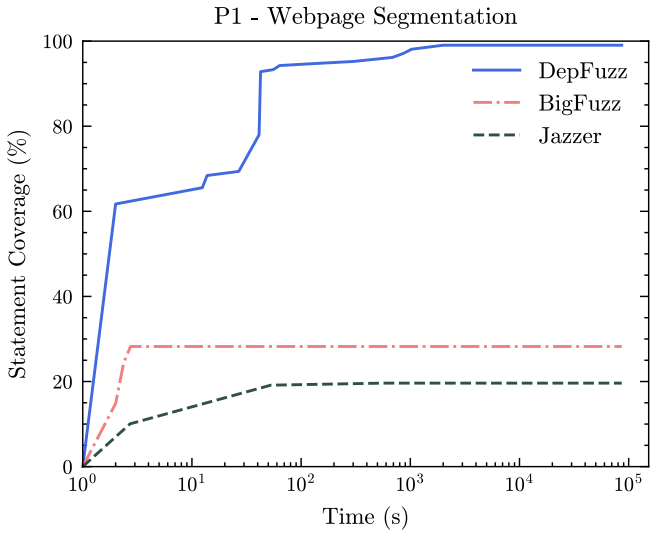
```
{(==, {Region A, Region B, Region C})}
```



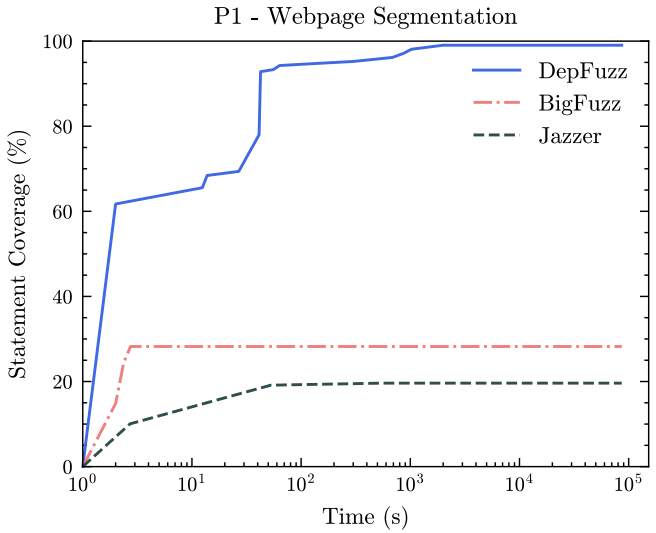
DepFuzz Evaluation

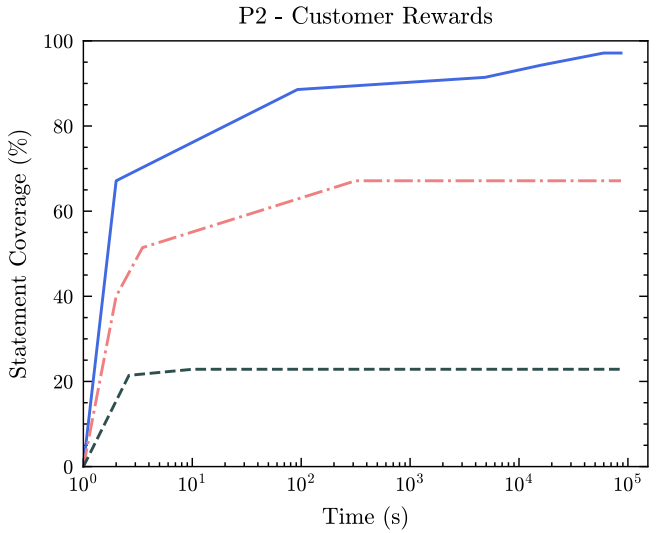
- DepFuzz is available on github: https://github.com/SEED-VT/DepFuzz
- Dockerized and easily runnable
- Baselines: BigFuzz and Jazzer
- Benchmark of 17 programs

Results – Coverage

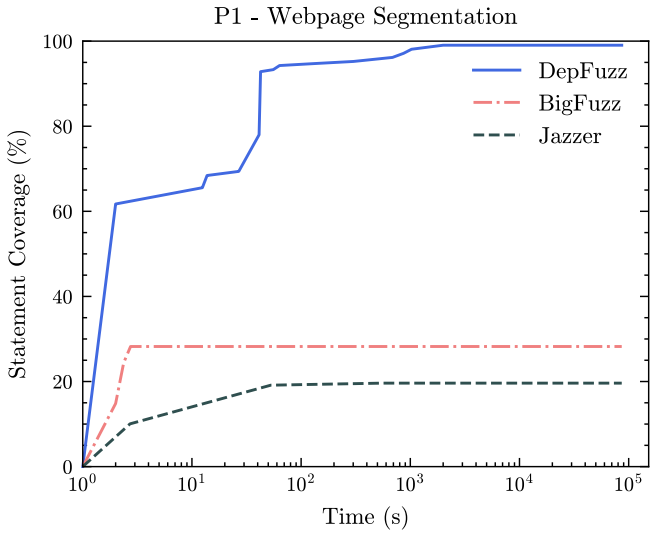


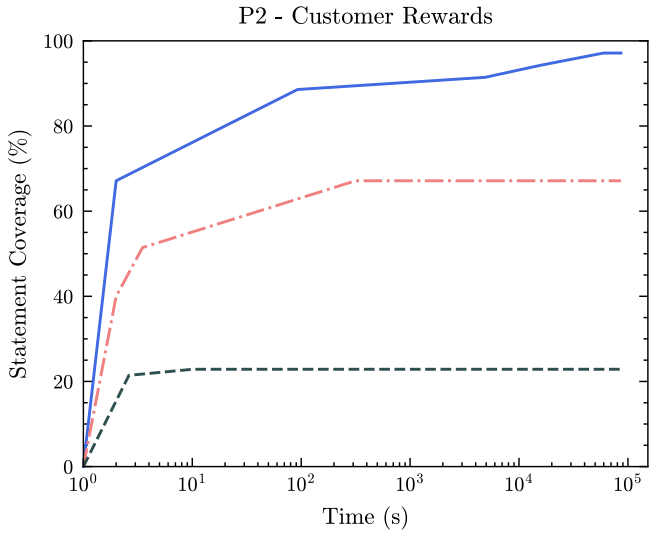
Results – Coverage

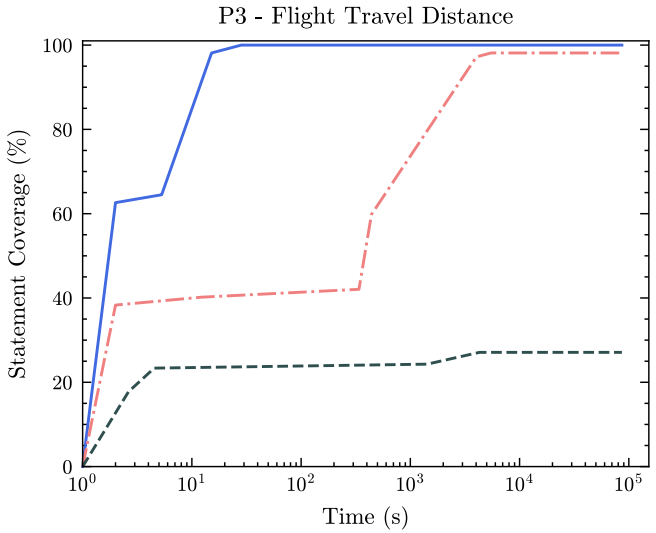




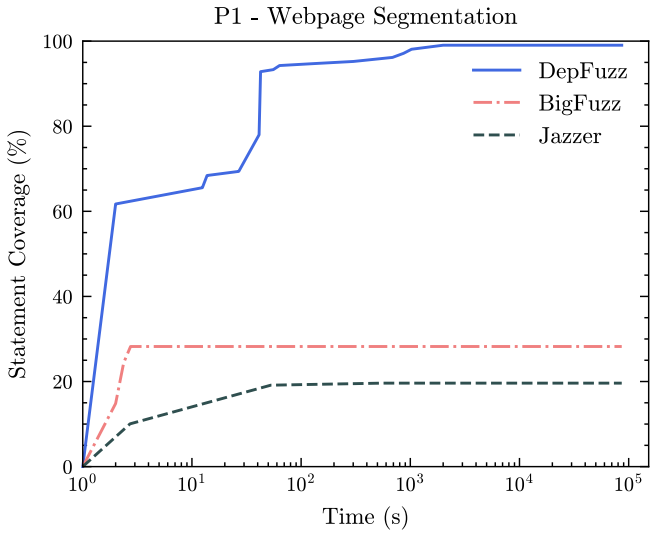
Results – Coverage

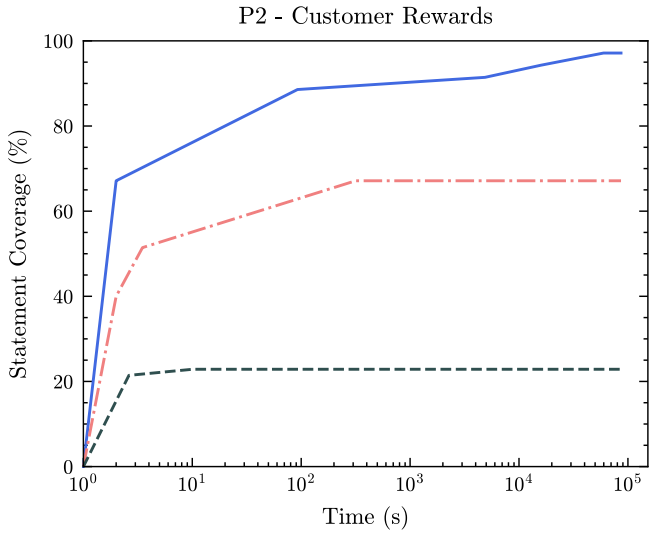


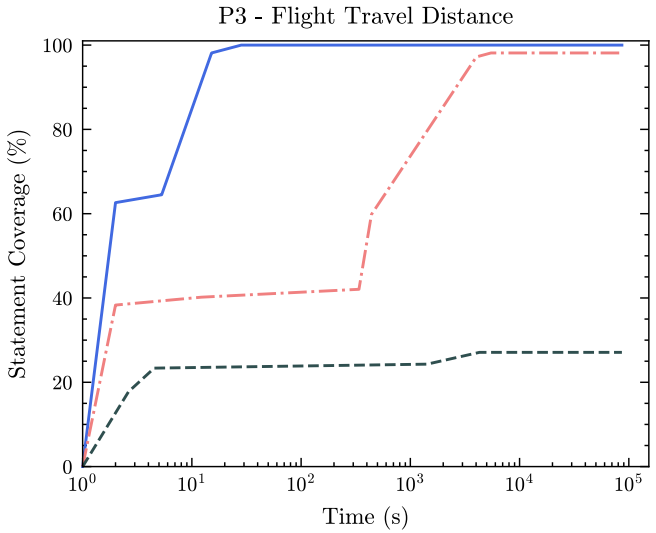




Results – Coverage

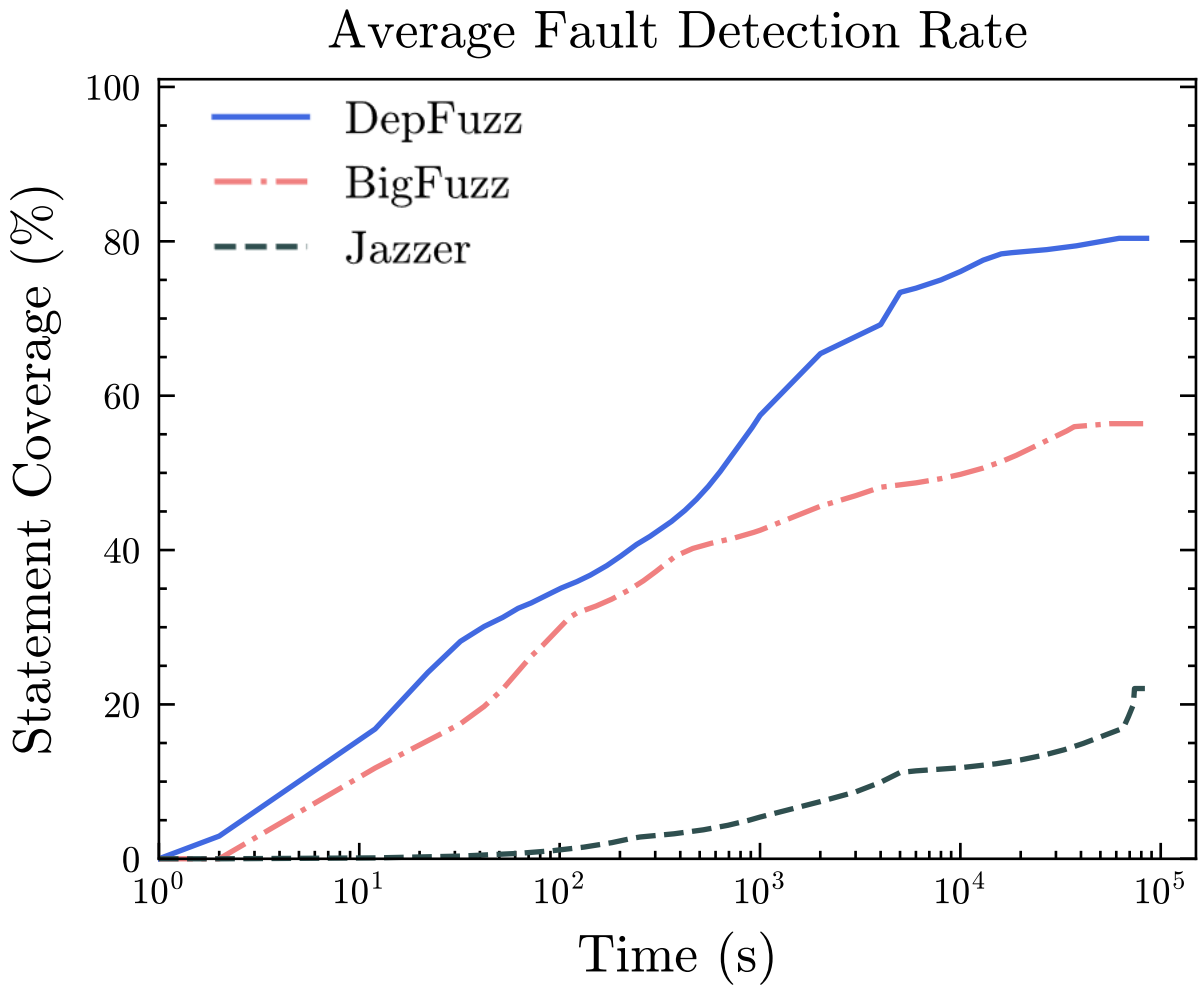




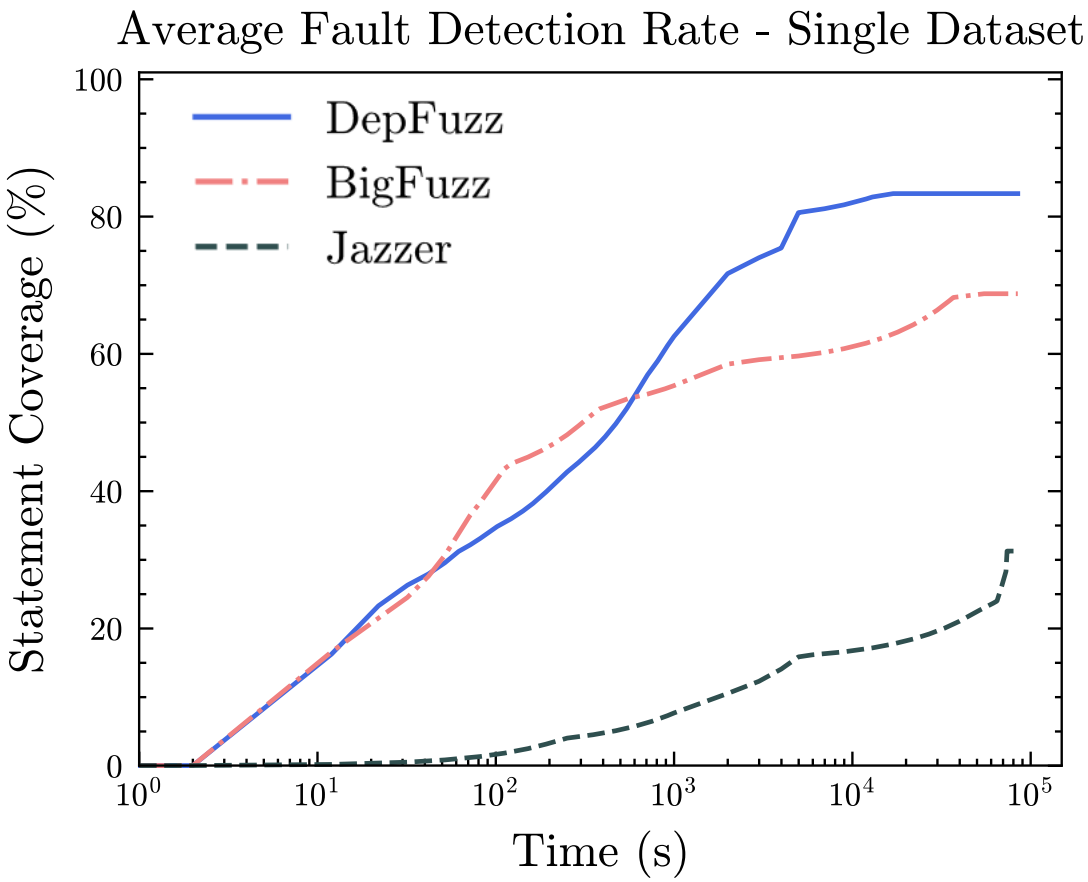


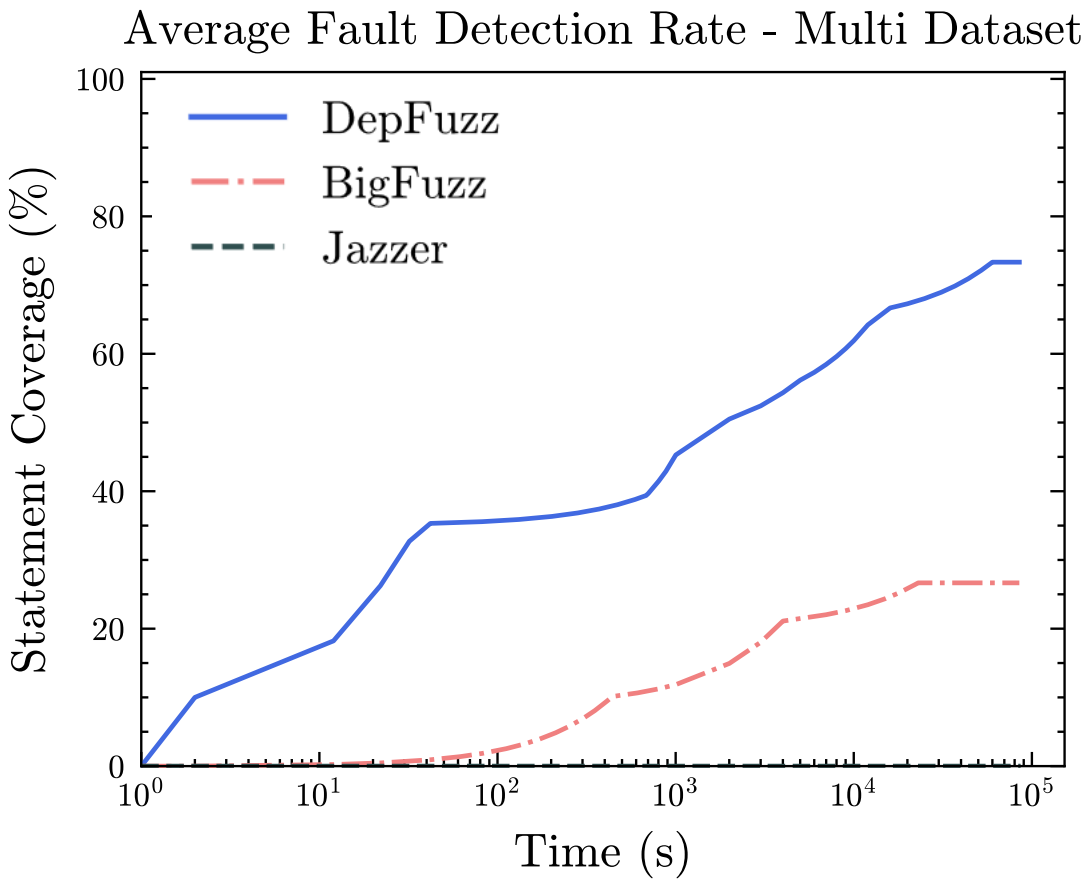
- On average, DepFuzz achieves 21% higher coverage than baselines.

Results – Fault Detection



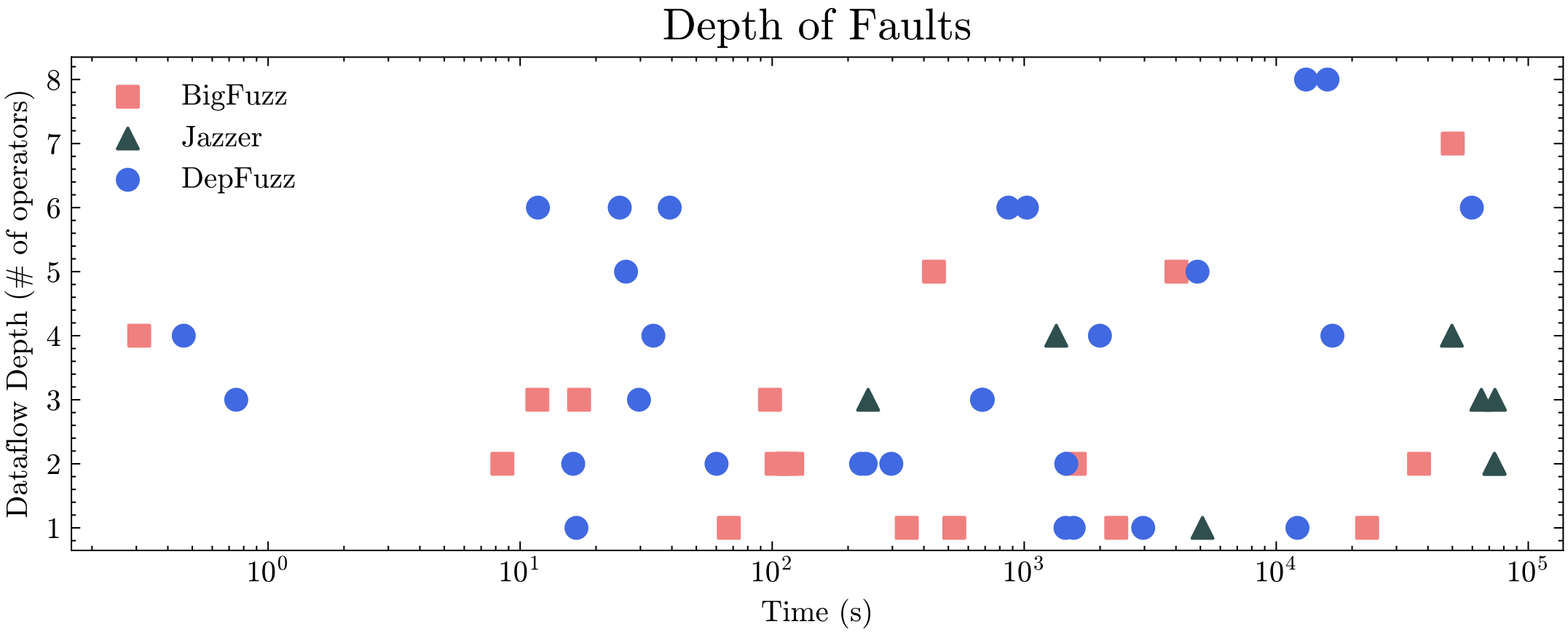
Results – Fault Detection





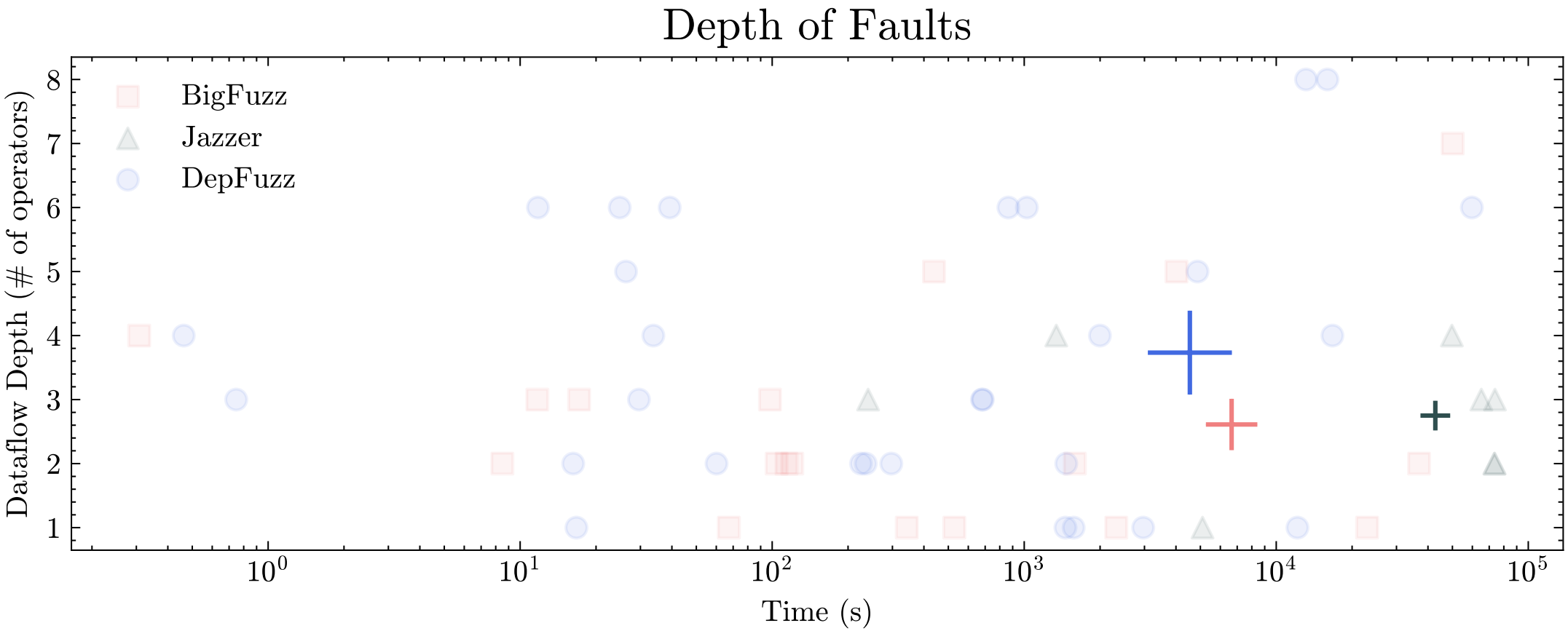
- On average, DepFuzz finds 2.6X more faults than baselines.

Results – Detected Fault Depth



- On average DepFuzz detects faults that are deeper

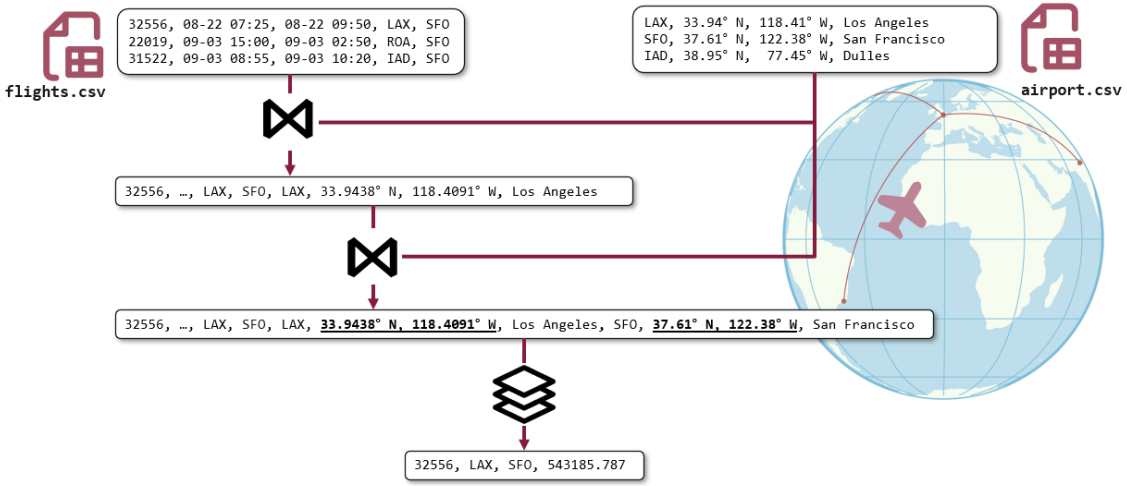
Results – Detected Fault Depth



- On average DepFuzz detects faults that are deeper

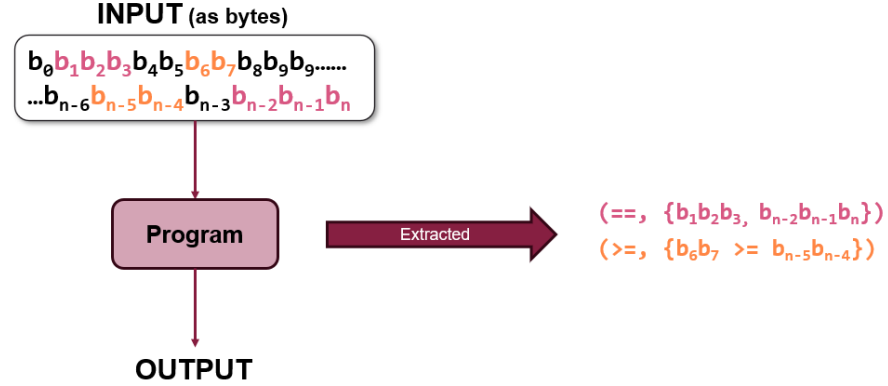
DISC Application Example

- Step 3: Use a map function to apply a formula using the longitude and lattitude values



Generalizing Co-Dependence

- Two byte-regions of the input that are related to each other by some operation

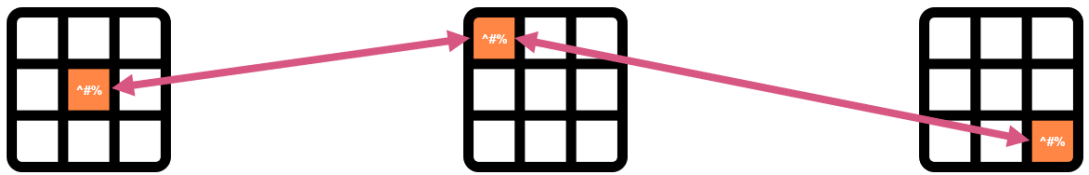


Representation

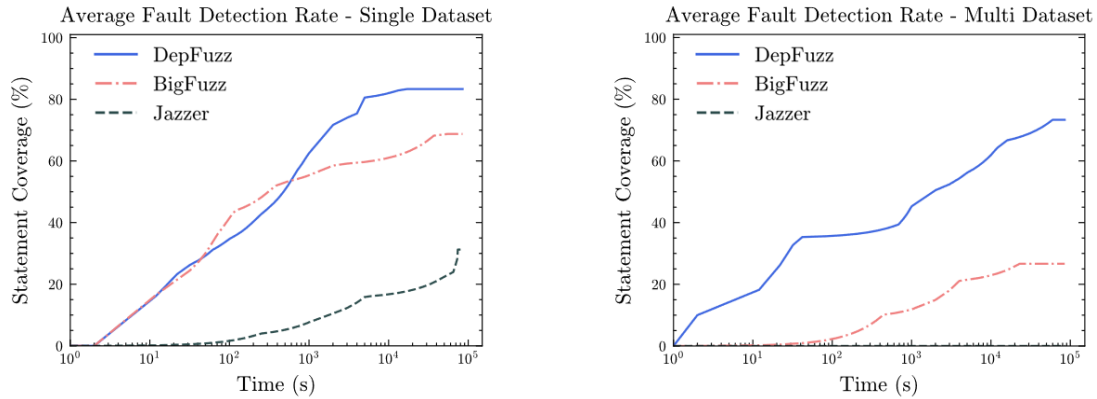
- Transitive co-dependencies can be merged

(==, {Region A, Region B})
(==, {Region B, Region C})

{(==, {Region A, Region B, Region C})}



Results – Fault Detection



- On average, DepFuzz finds 2.6X more faults than baselines.

VIRGINIA TECH

UCLA

Co-dependence Aware Fuzzing for Dataflow-based Big Data Analytics

Ahmad Humayun, Miryung Kim, Muhmmad Ali Gulzar