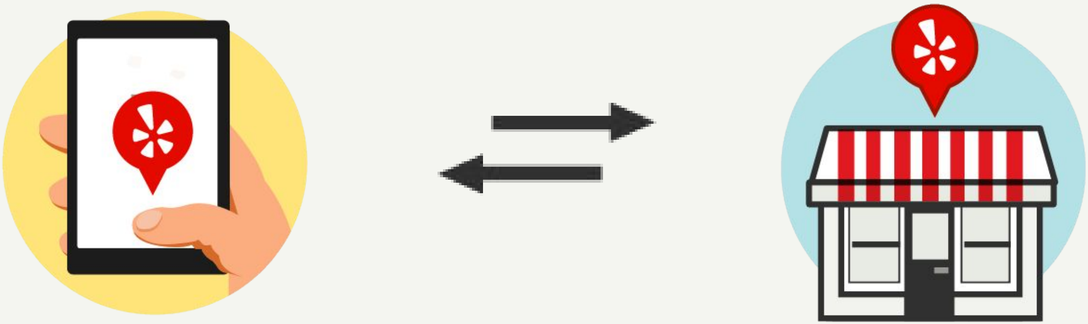
Building Stream Processing Applications

Amit Ramesh

Qui Nguyen

Yelp's Mission

Connecting people with great local businesses.



I. Why stream processing?

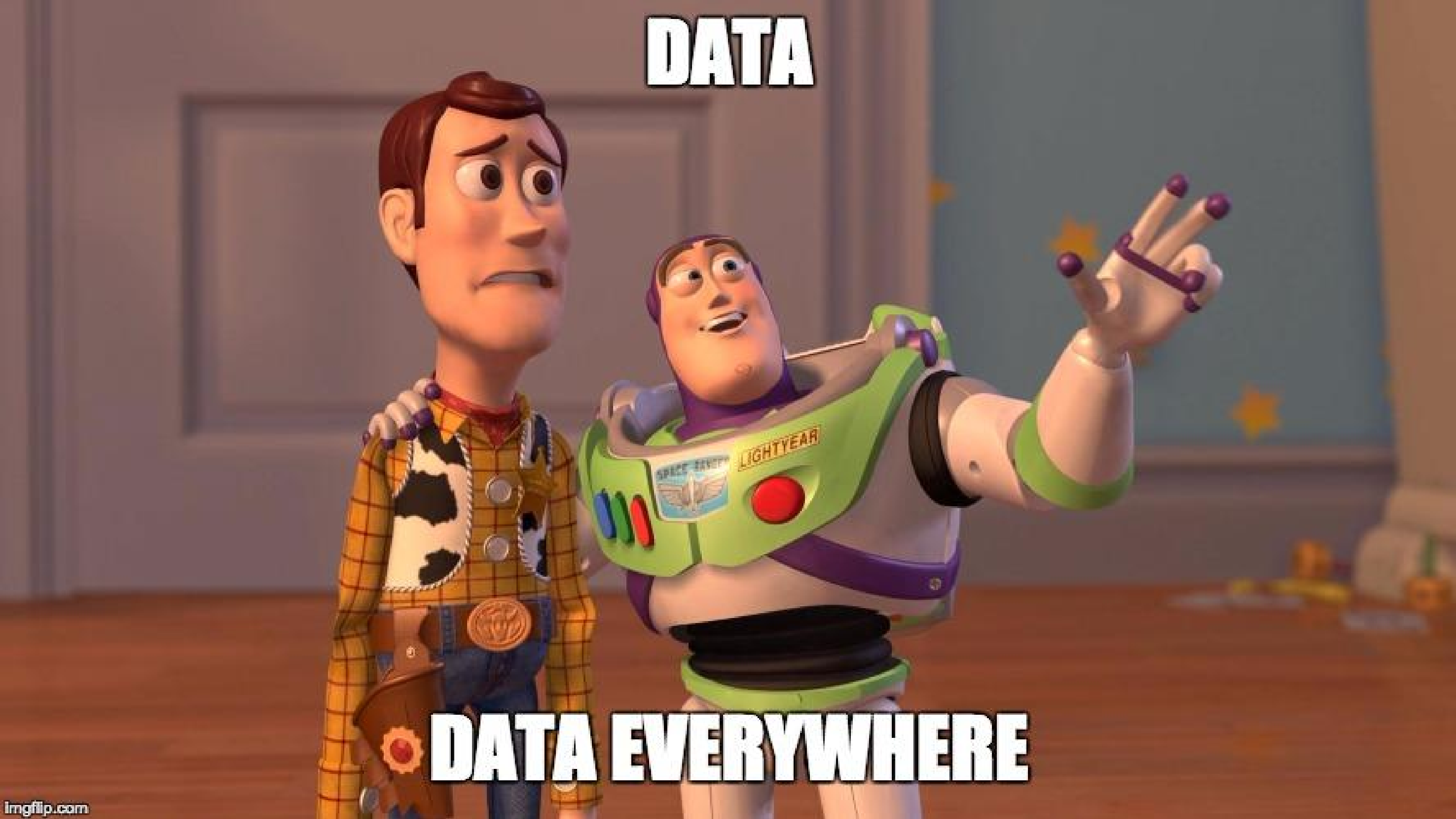
II. Putting an application together
    - Example problem
    - Components and data operations

III. Design principles and tradeoffs
    - Horizontal scalability
    - Handling failures
        - Idempotency
        - Consistency versus availability

I. **Why stream processing?**

II. Putting an application together
   - Example problem
   - Components and data operations

III. Design principles and tradeoffs
   - Horizontal scalability
   - Handling failures
     - Idempotency
     - Consistency versus availability


DATA
DATA EVERYWHERE
imgflip.com

Data processing

measurements from a sensor

clicking on ads

Data processing

measurements from a sensor

clicking on ads

average value in the last minute

total clicks on a day

Data processing: Batch or stream

Batch

Finite chunk of data

Operations defined over the entire input

Data processing: Batch or stream

Batch

Finite chunk of data

Operations defined over the entire input

Stream

Unbounded stream of events flowing in

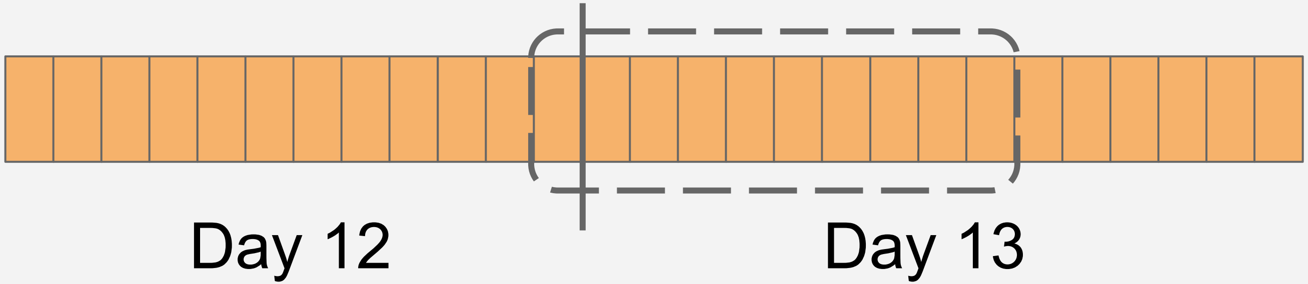
Events are processed continuously (possibly with state)

Why stream processing over batch?

- Lower latency on results
- Most data is unbounded, so streaming model is more flexible

Why stream processing over batch?

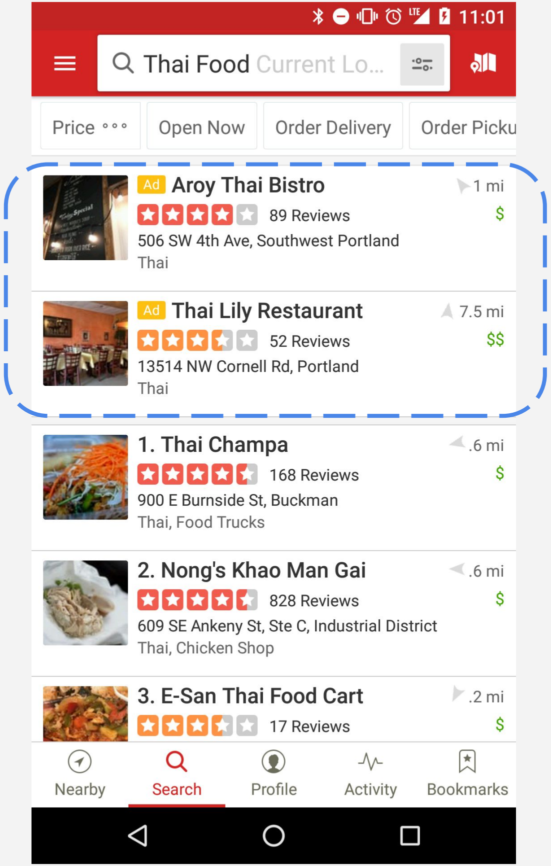
- Lower latency on results
- Most data is unbounded, so streaming model is more flexible



Our evolution

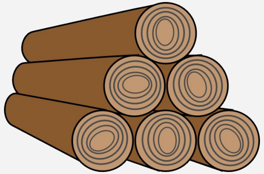
yelp

Our evolution



Our evolution

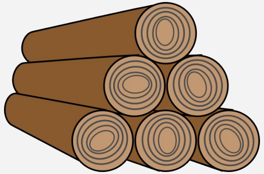
Our evolution



Our evolution

Our evolution

yelp



Spark Streaming

I. Why stream processing?

II. **Putting an application together**

Example problem

Components and data operations

III. Design principles and tradeoffs

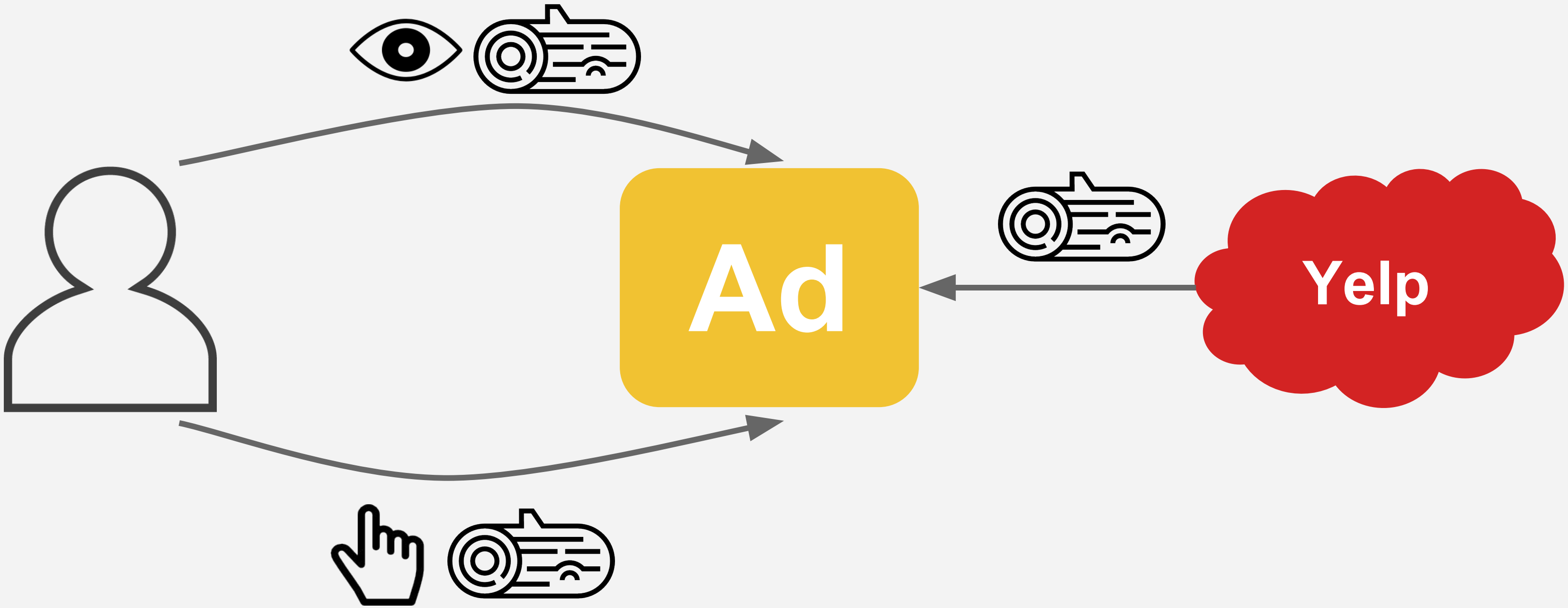
Horizontal scalability

Handling failures

Idempotency

Consistency versus availability

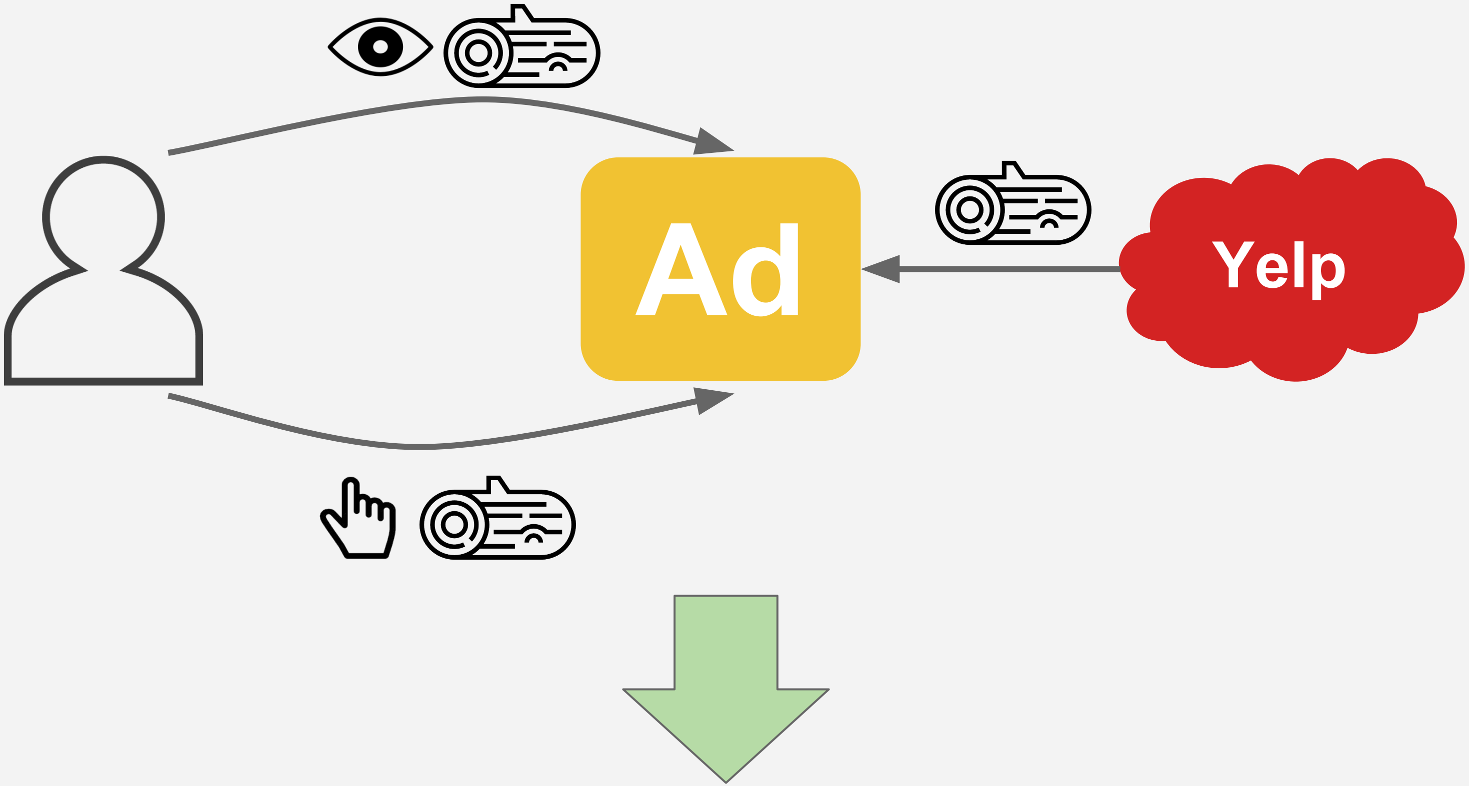
Example problem: ad campaign metrics



```
ad {
  id: 1200834,
  campaign_id: 2001,
  user_id: 9zkjacn81m,
  timestamp: 1490732147
}
```

```
view {
  id: 1200834,
  timestamp: 1490732150
}
```

```
click {
  id: 1200834,
  timestamp: 1490732168
}
```



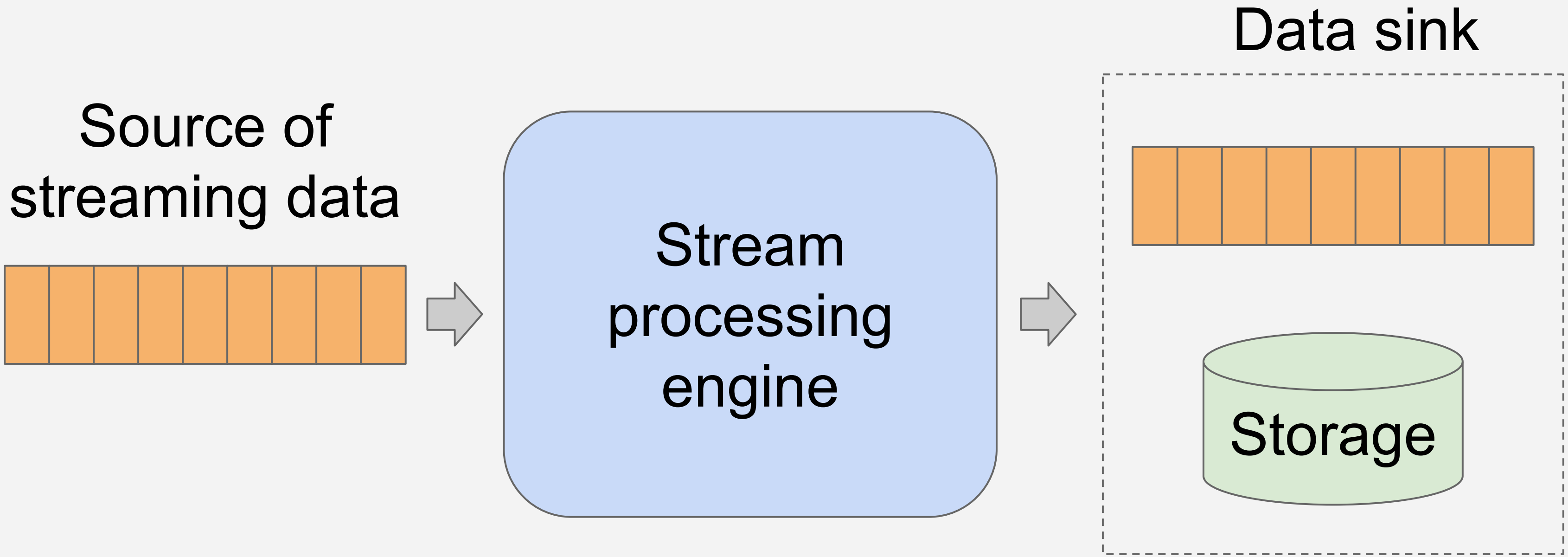
Metrics (views, clicks) for each
campaign over time

I. Why stream processing?

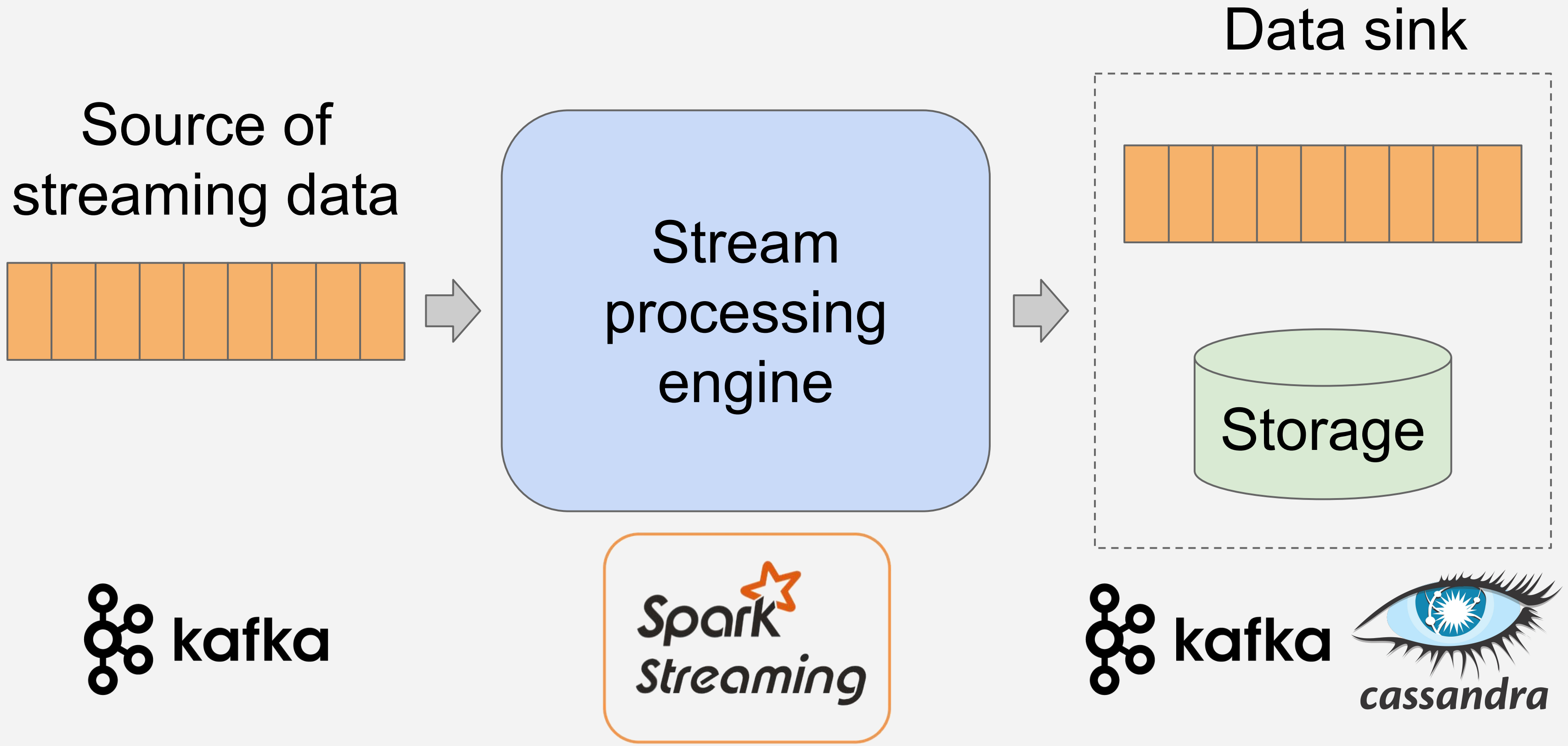
II. **Putting an application together**
   Example problem
   Components and data operations

III. Design principles and tradeoffs
   Horizontal scalability
   Handling failures
      Idempotency
      Consistency versus availability

Stream processing pipelines



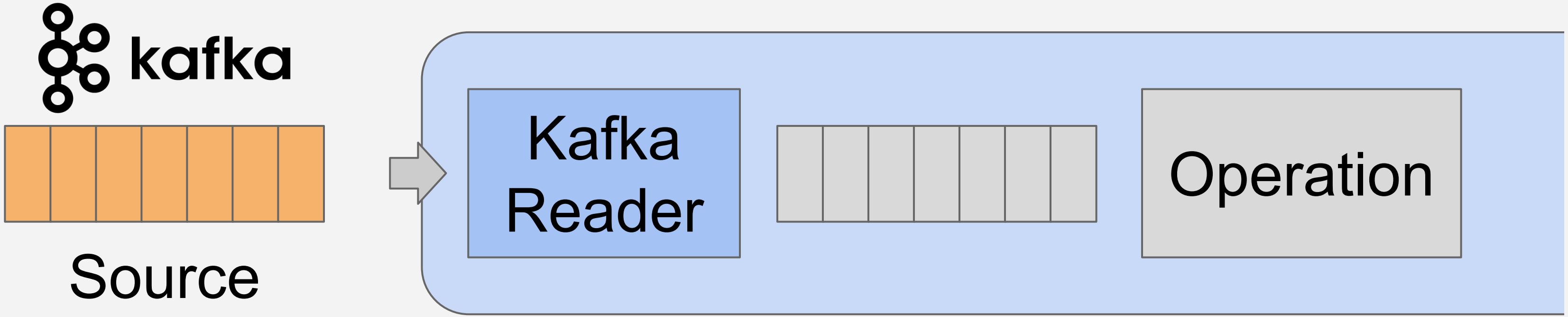
Stream processing pipelines



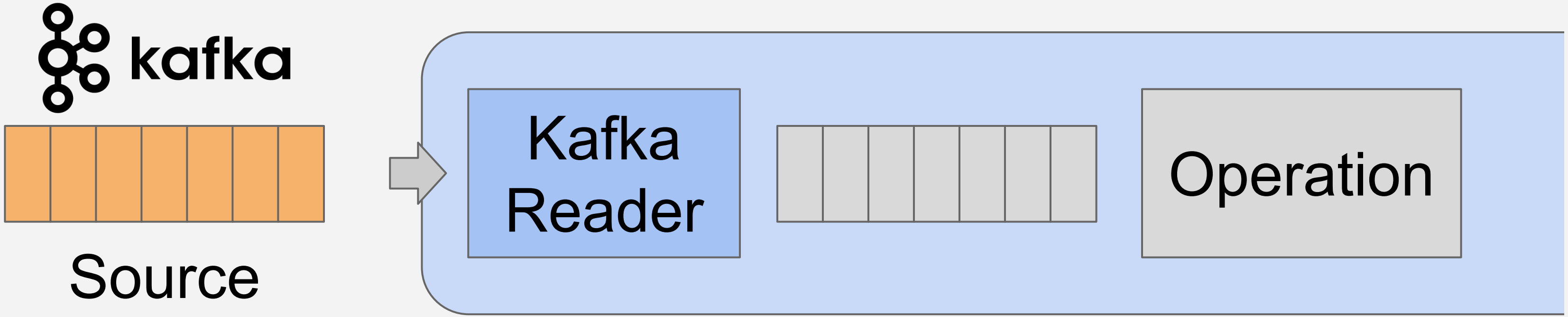
Types of operations

1. Ingestion
2. Stateless transforms
3. Stateful transforms
4. Keyed stateful transforms
5. Publishing

Operations: 1. Ingestion



Operations: 1. Ingestion



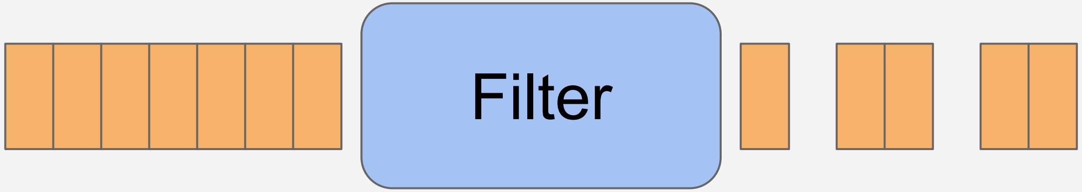

```
from pyspark.streaming.kafka import KafkaUtils

ad_stream = KafkaUtils.createDirectStream(
  streaming_context,
  topics=['ad_events'],
  kafkaParams={...},
)
```

Operations: 2. Stateless transforms

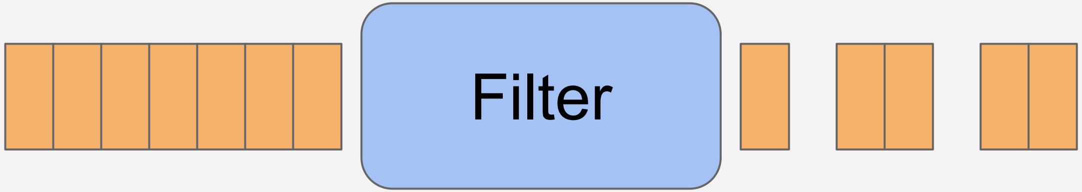
Operations: 2a. Stateless transforms

e.g., filtering



Operations: 2a. Stateless transforms

e.g., filtering

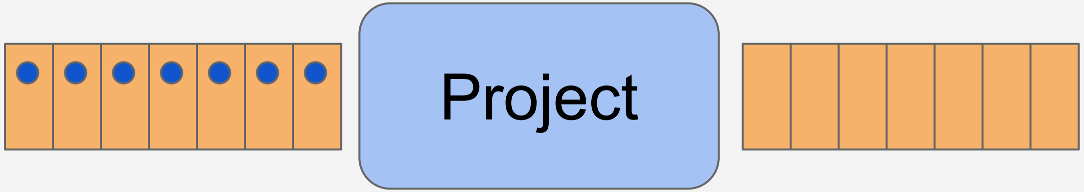



```
def is_not_from_bot(event):
  return event[‘ip’] not in bot_ips

filtered_stream = ad_stream.filter(is_not_from_bot)
```

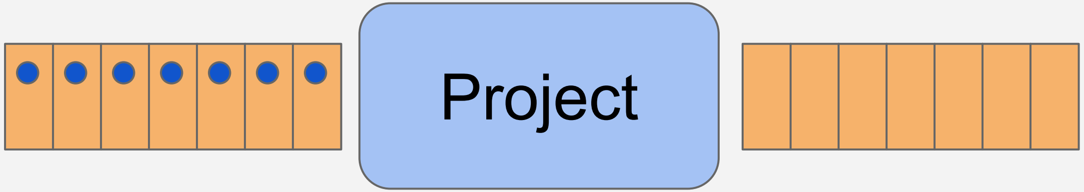
Operations: 2b. Stateless transforms

e.g., projection



Operations: 2b. Stateless transforms

e.g., projection



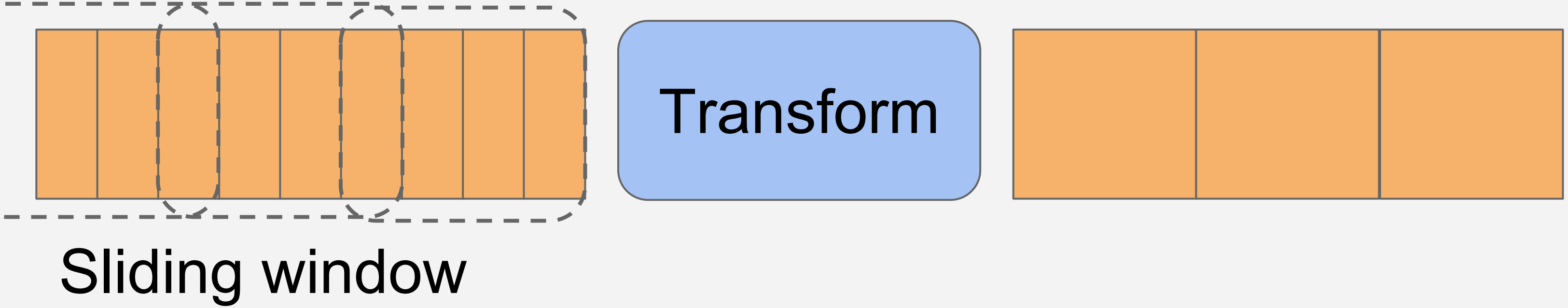

```
desired_fields = [‘ad_id’, ‘campaign_id’]

def trim_event(event):
  return {key: event[key] for key in desired_fields}

projected_stream = ad_stream.map(trim_event)
```

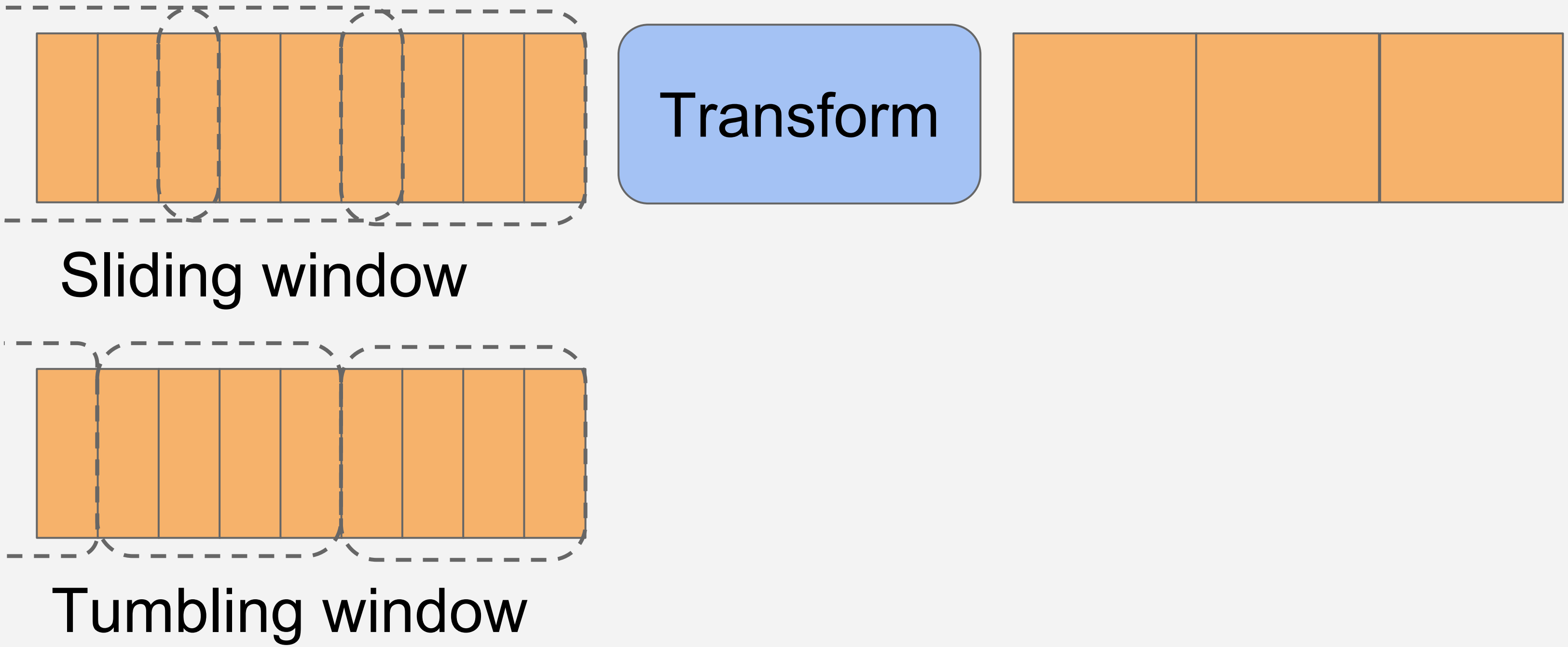
Operations: 3. Stateful transforms

On windows of data



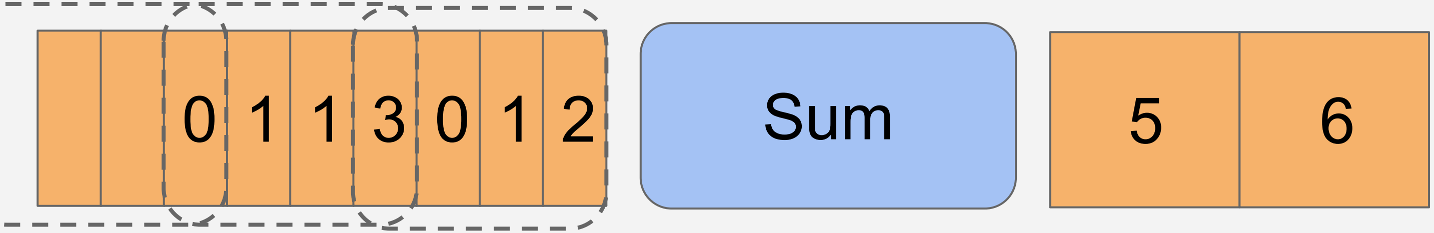
Operations: 3. Stateful transforms

On windows of data



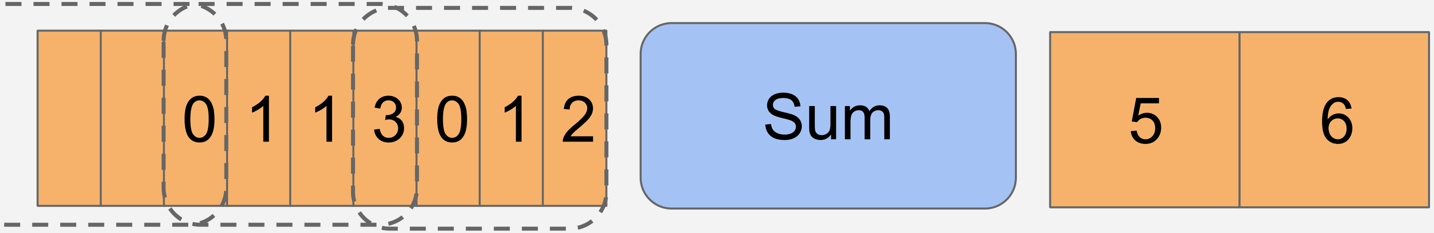
Operations: 3. Stateful transforms

e.g., aggregation



Operations: 3. Stateful transforms

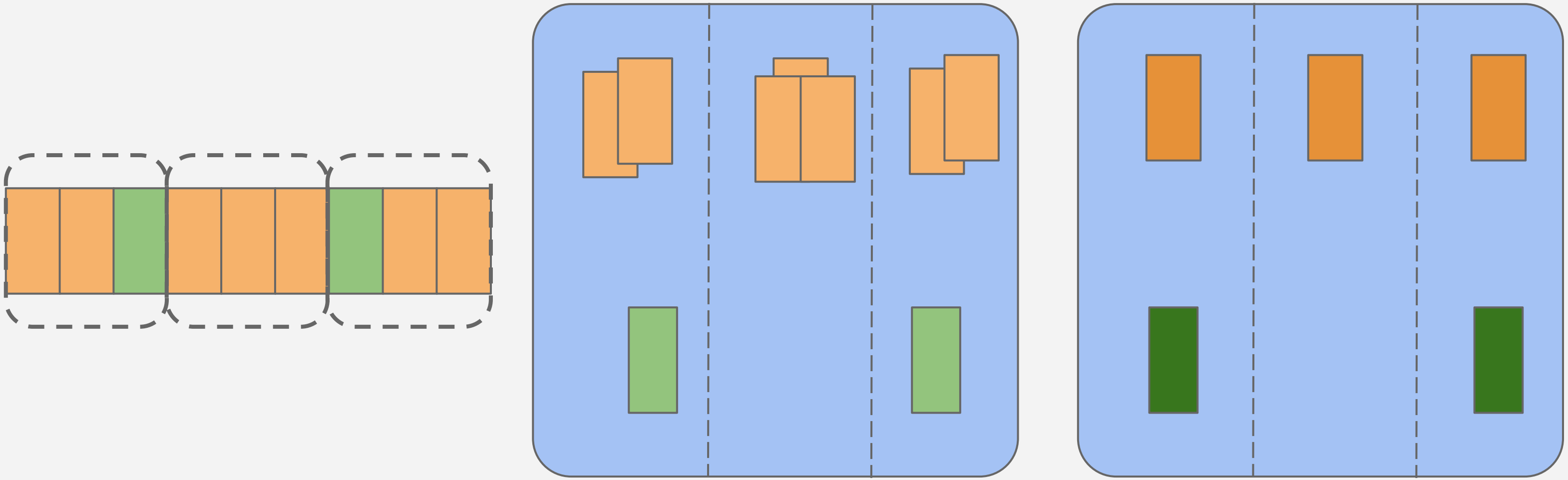
e.g., aggregation




```
aggregated_stream = event_stream.reduceByWindow(
  func=operator.add,
  windowLength=4,
  slideInterval=3,
)
```

Operations: 4. Keyed stateful transforms

Group events by key (shuffle) within each window before transform

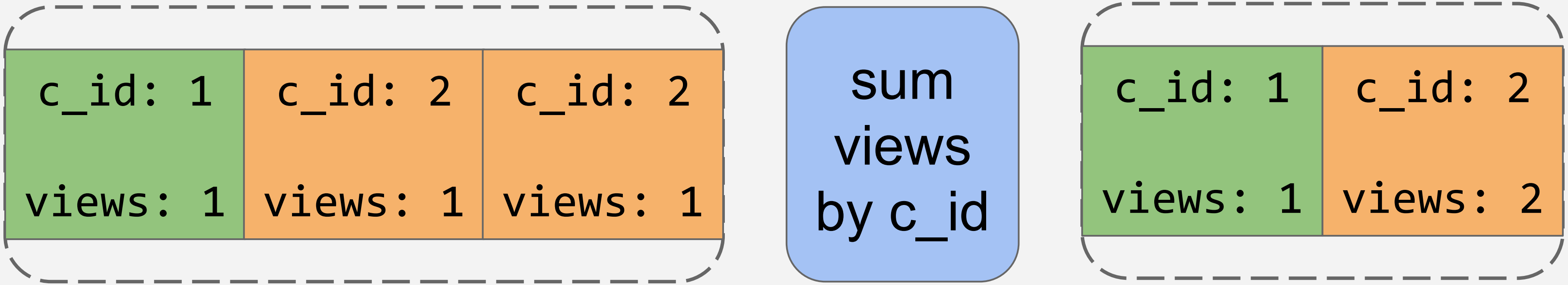


Shuffle

Transform

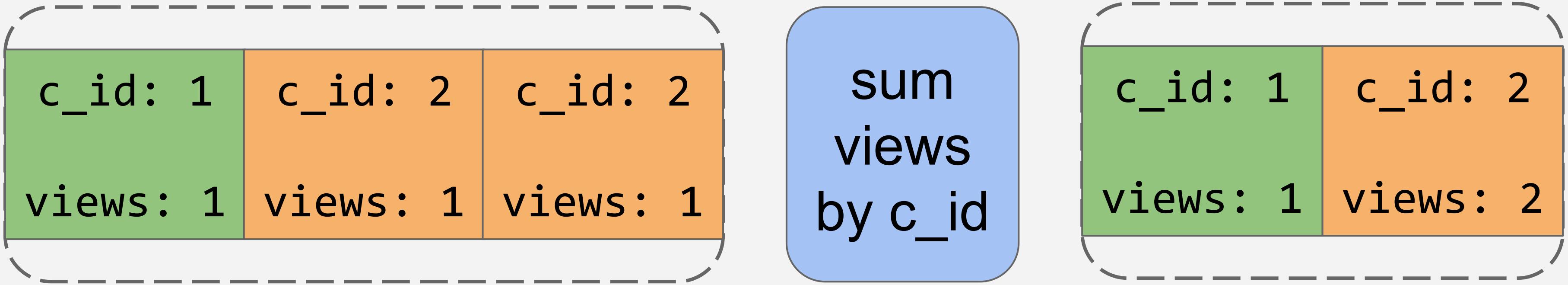
Operations: 4a. Keyed stateful transforms

e.g., aggregate views by campaign_id



Operations: 4a. Keyed stateful transforms

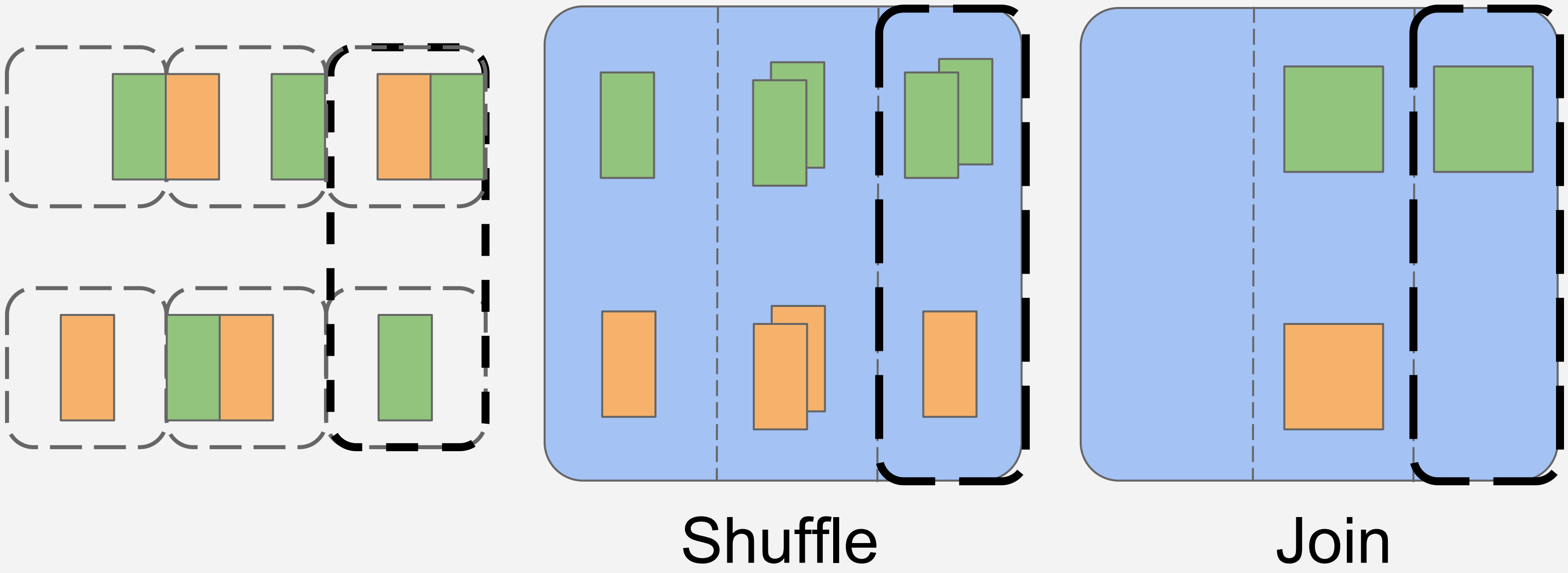
e.g., aggregate views by campaign_id




```
aggregated_views = view_stream.reduceByKeyAndWindow(
  func=operator.add,
  windowLength=3,
  slideInterval=3,
)
```

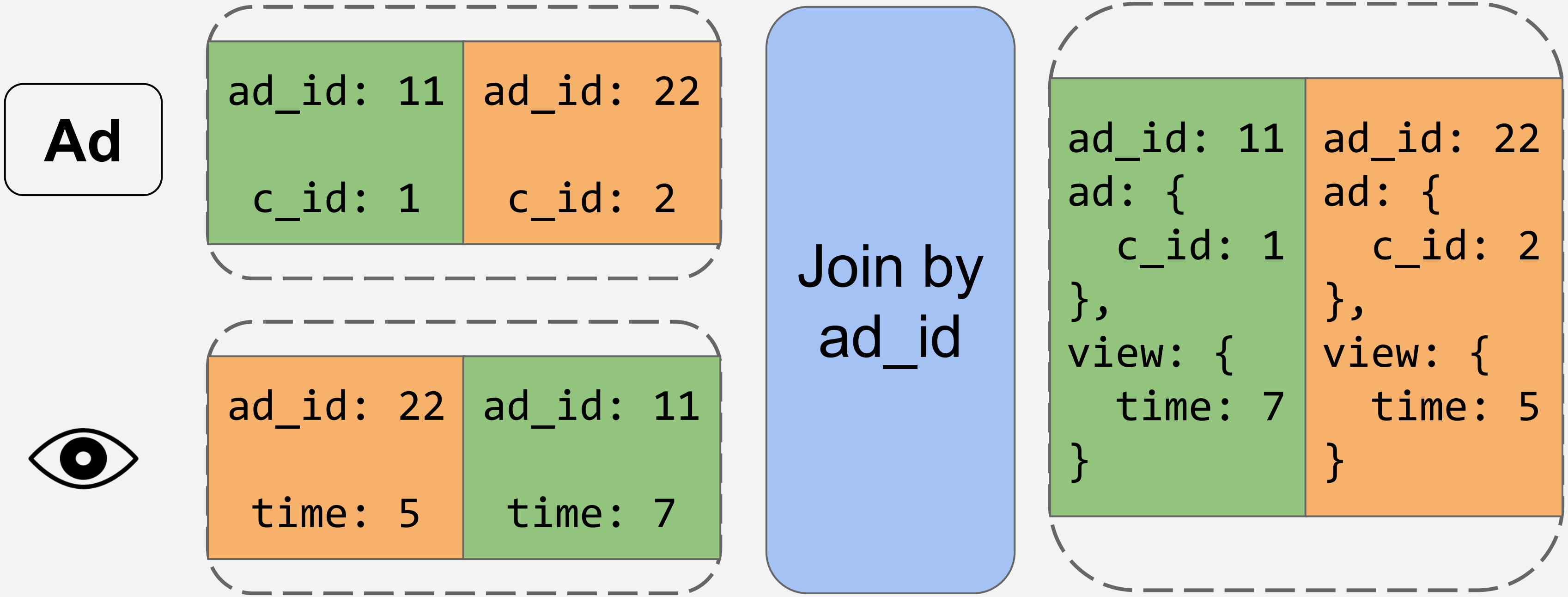
Operations: 4b. Keyed stateful transforms

Can also be on more than one stream, e.g., join by id



Operations: 4b. Keyed stateful transforms

e.g., join by ad_id

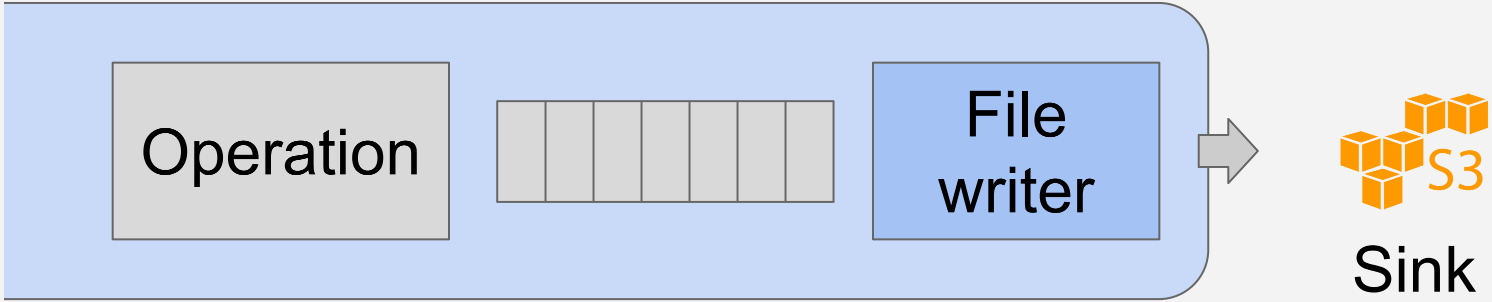


Operations: 4b. Keyed stateful transforms

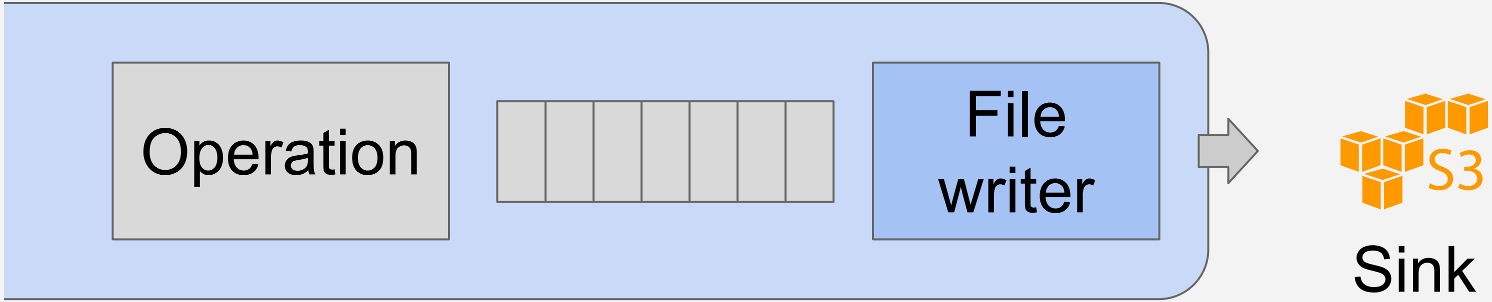
e.g., join by ad_id

```
windowed_ad_stream = ad_stream.window(
  windowLength=2,
  slideInterval=2,
)
windowed_view_stream = view_stream.window(
  windowLength=2,
  slideInterval=2,
)
joined_stream = windowed_ad_stream.join(
  windowed_view_stream,
)
```

Operations: 5. Publishing



Operations: 5. Publishing

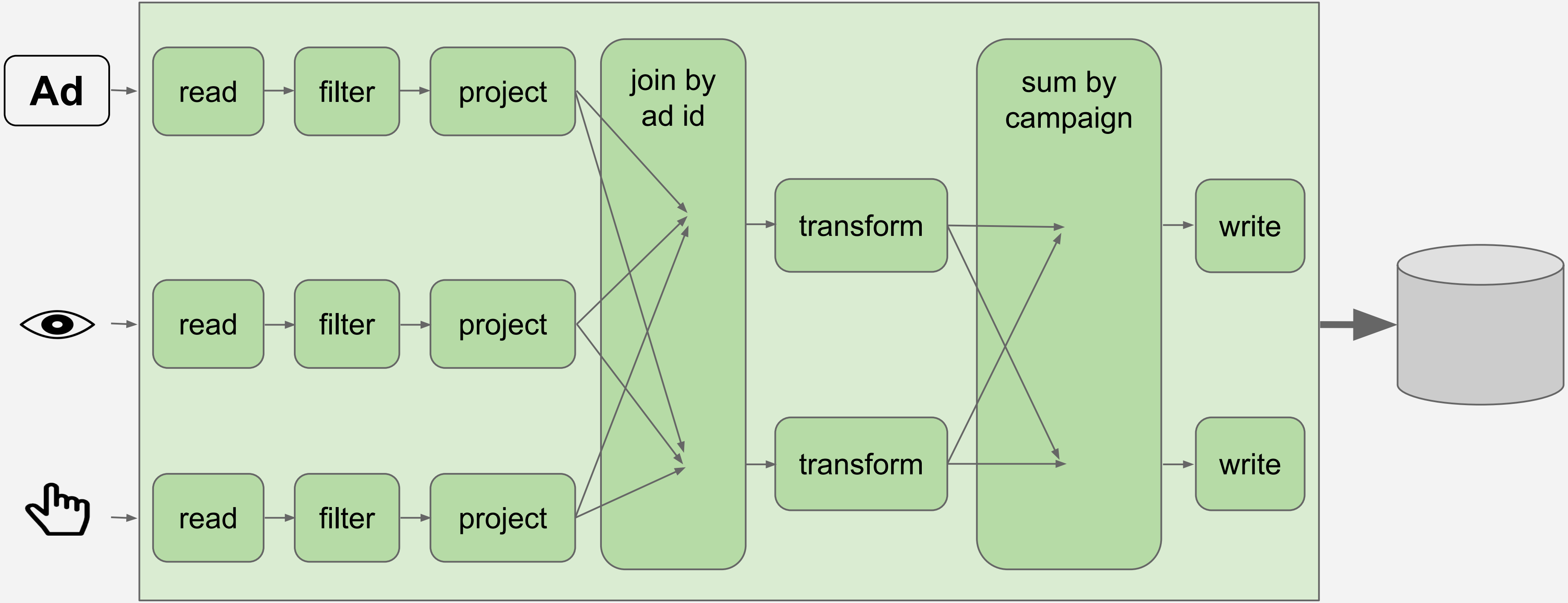



```
results_stream.saveAsTextFiles('s3://my.bucket/results/')
```

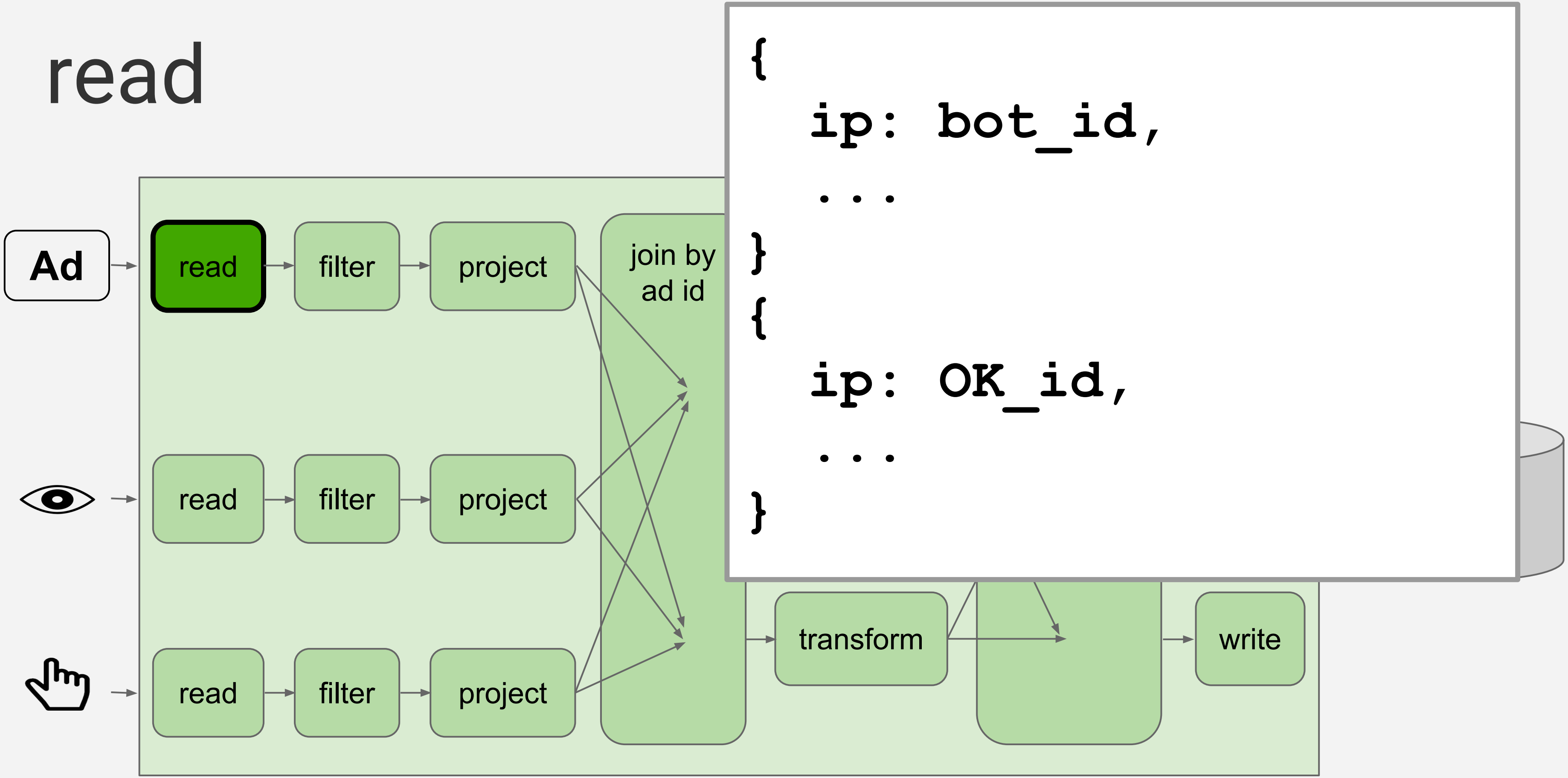
Operations: Summary

1. **Ingestion**
2. **Stateless transforms**: on single events
    a. Filtering
    b. Projections
3. **Stateful transforms**: on windows of events
4. **Keyed stateful transforms**
    a. On single streams, transform by key
    b. Join events from several streams by key
5. **Publishing**

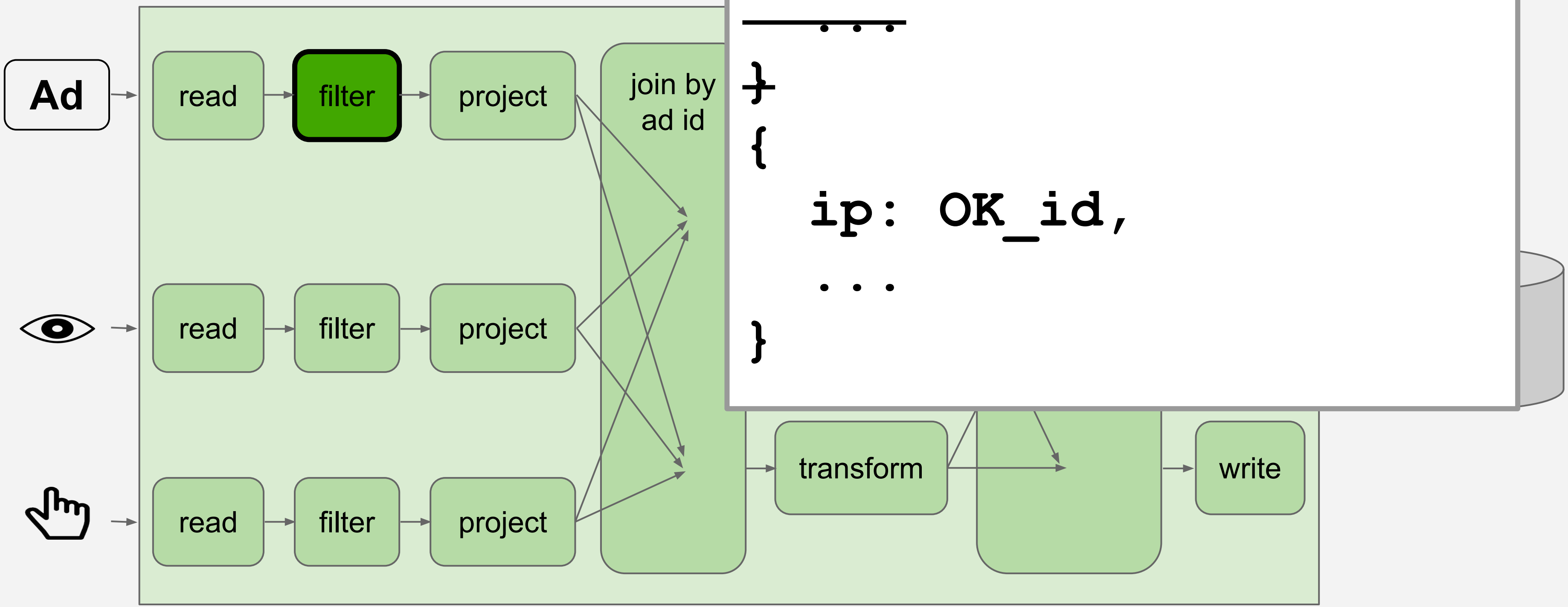
Putting it together: campaign metrics



read


Ad
read
filter
project
read
filter
project
read
filter
project
join by
ad id
transform
write
{
ip: bot_id,
...
}
{
ip: OK_id,
...
}

filter



~~{~~

~~ip: bot_id,~~

~~...~~

~~}~~

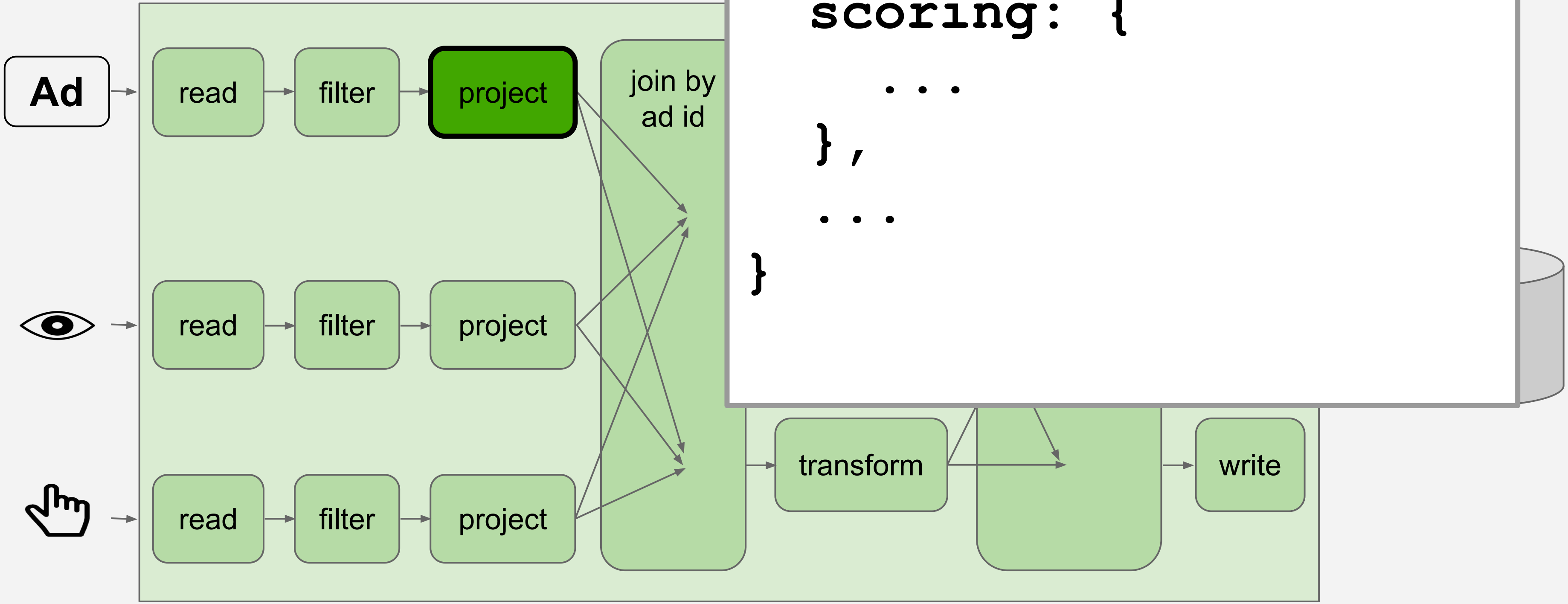
{

ip: OK_id,

...

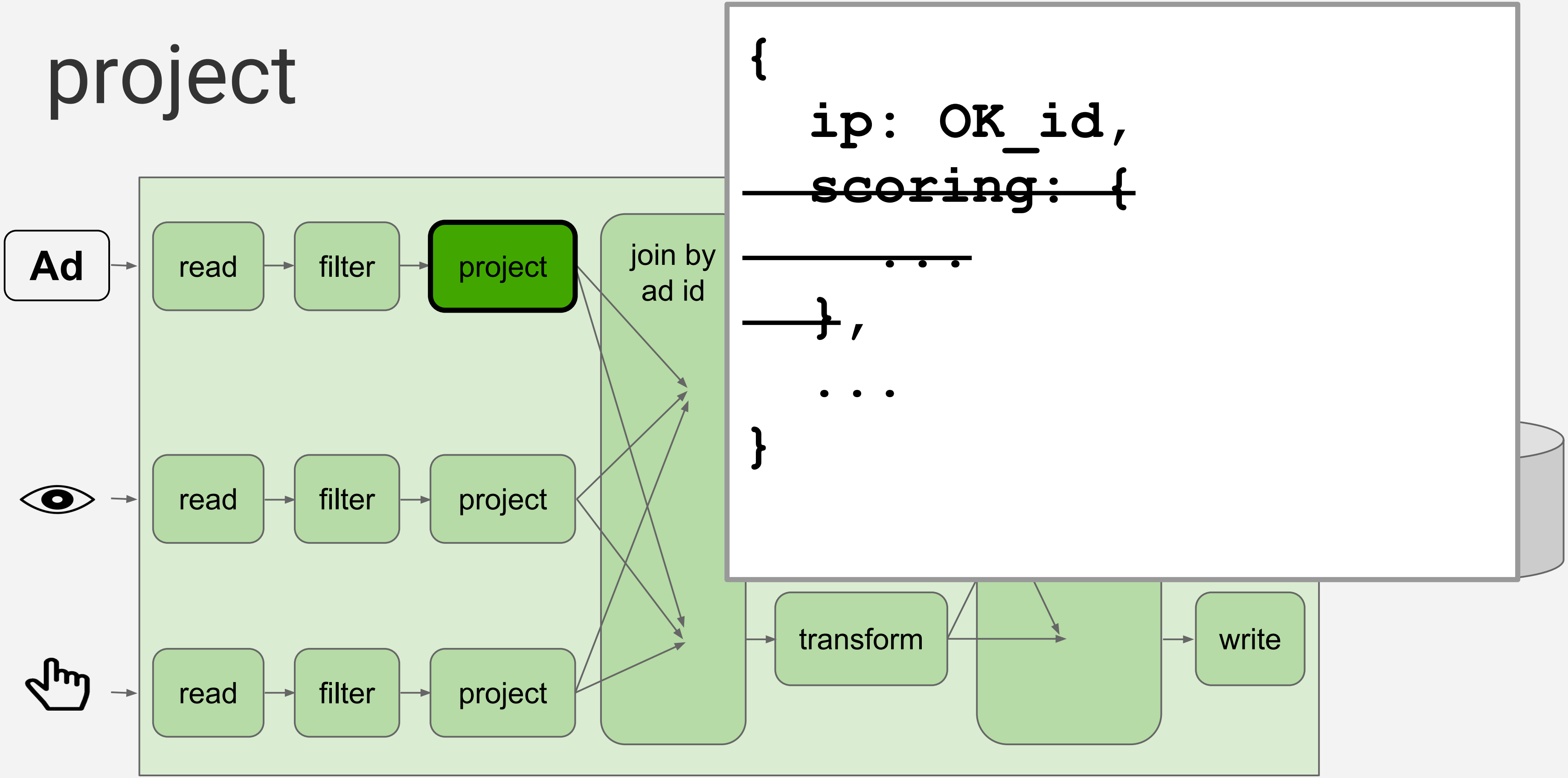
}

project



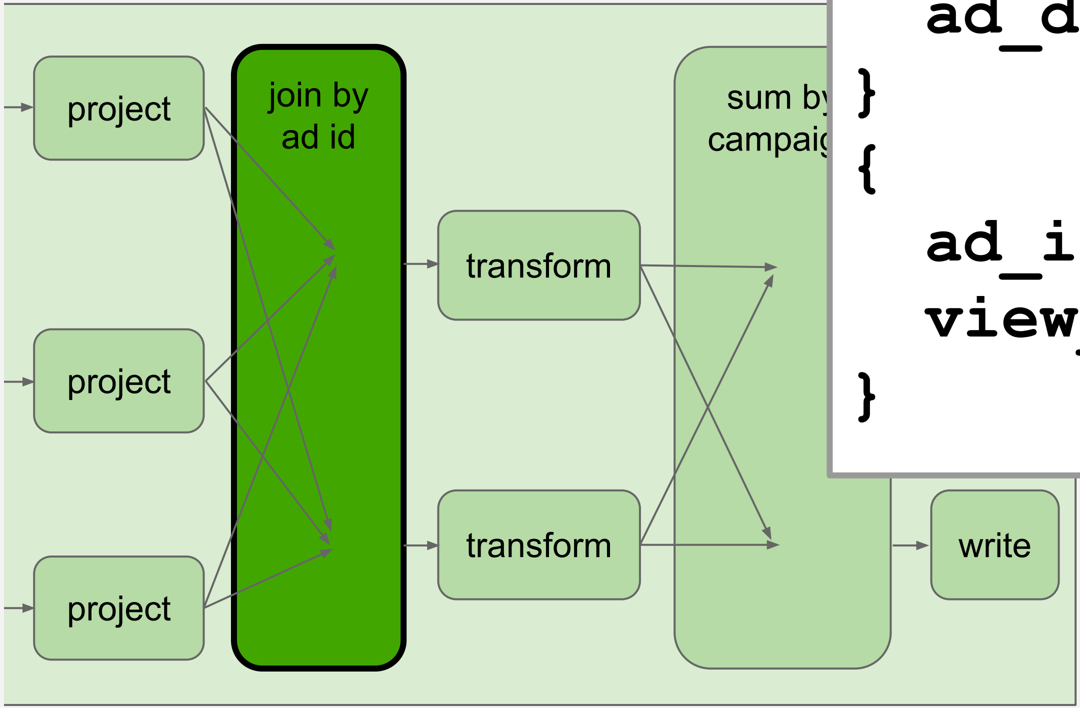

```
{
  ip: OK_id,
  scoring: {
     ...
  },
  ...
}
```

project



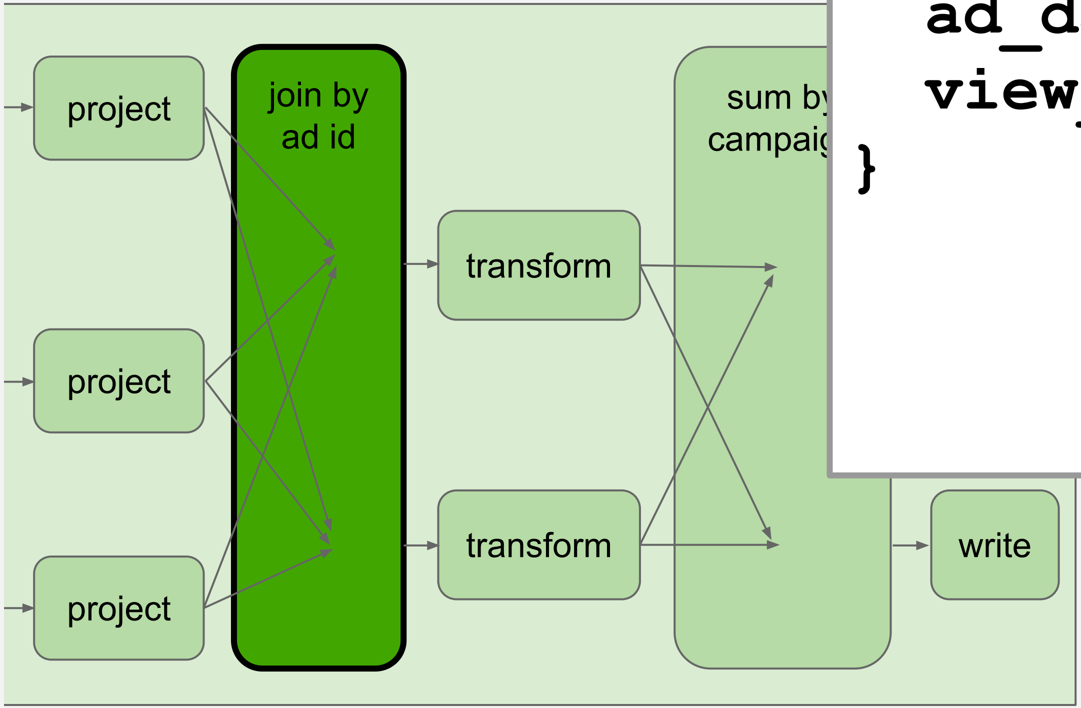
Ad
read
filter
project
read
filter
project
read
filter
project
join by ad id
transform
write
{
ip: OK_id,
~~scoring: {~~
~~...~~
~~}~~,
...
}

join by ad id



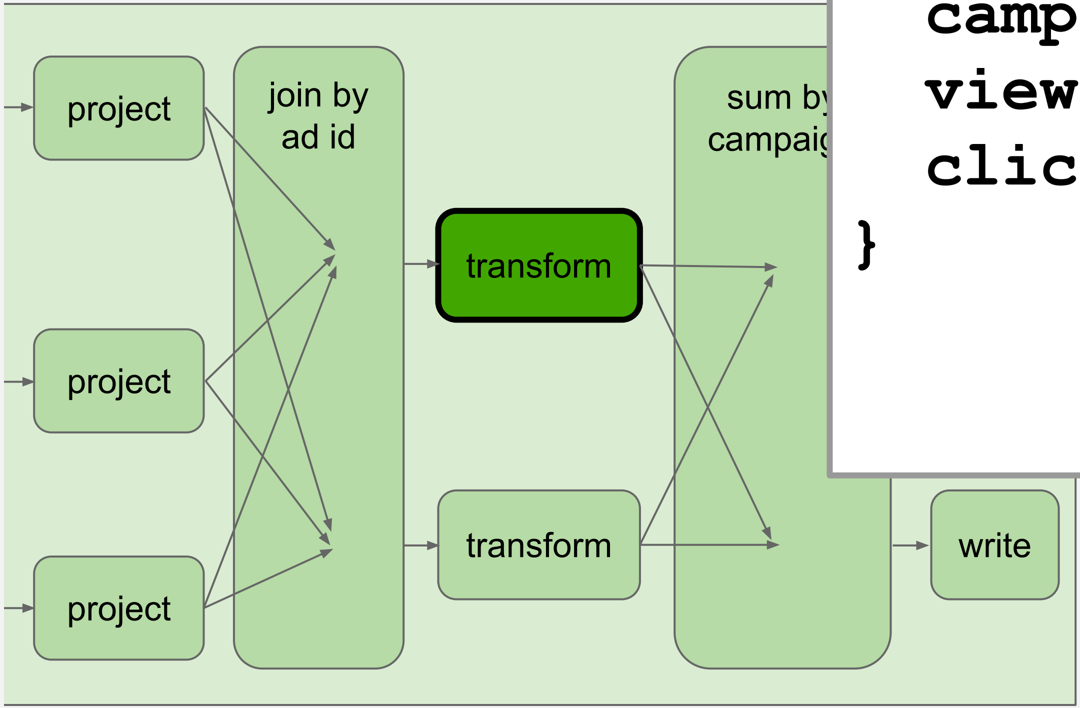

```
{
  ad_id: 1,
  ad_data: ...
}
{
  ad_id: 1,
  view_data: ...
}
```

join by ad id



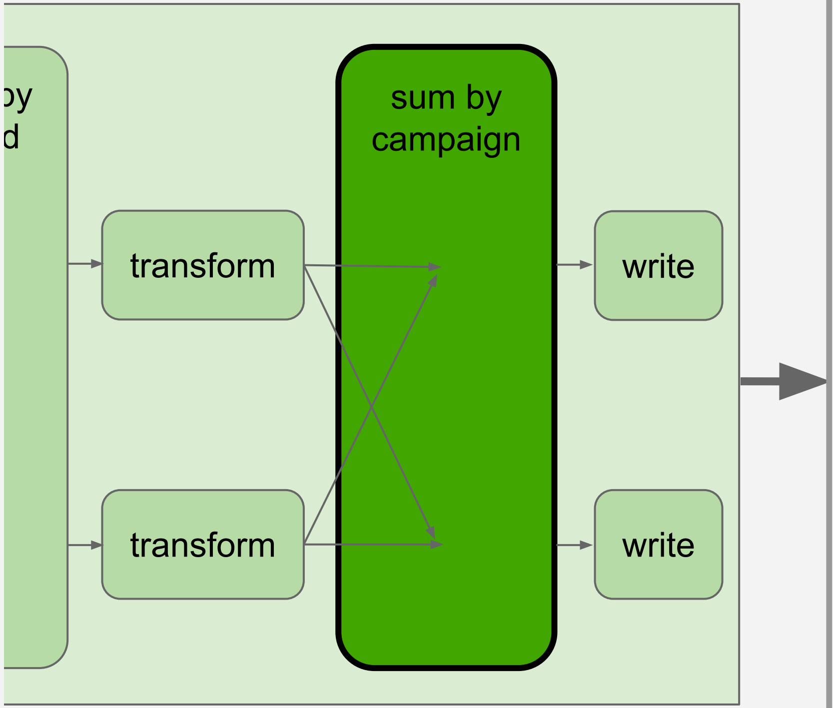

```
{
  ad_id: 1,
  ad_data: ...,
  view_data: ...,
}
```

transform



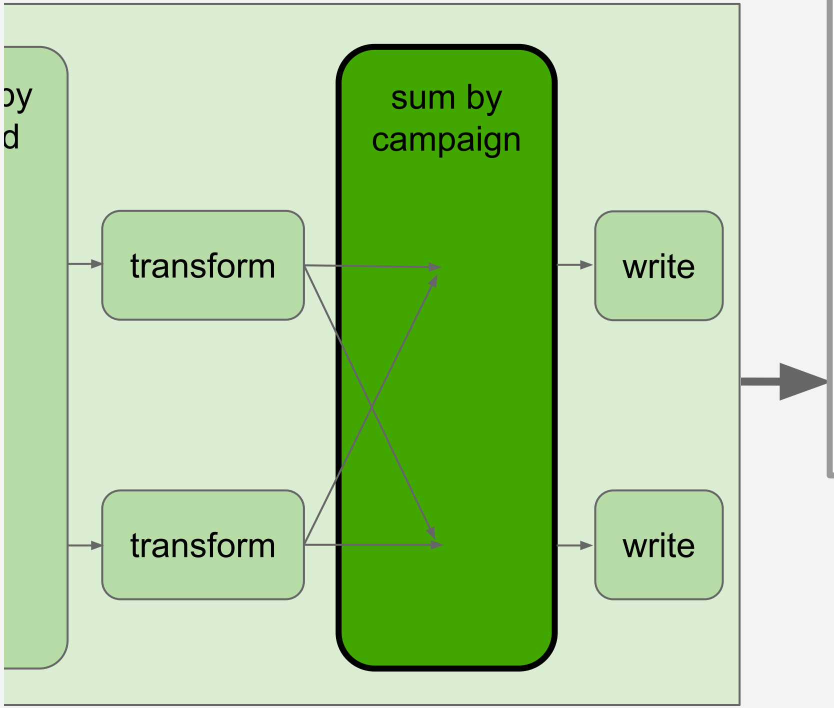

```
{
  ad_id: 1,
  campaign_id: 7,
  view: true,
  click: false
}
```

sum by campaign



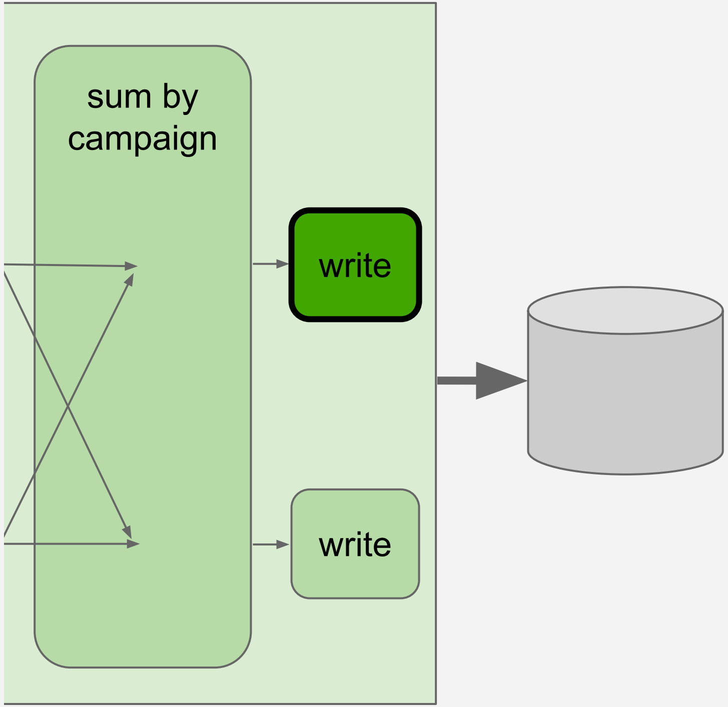

```
{
  ad_id: 1,
  campaign_id: 7,
  view: true,
  click: false
}
{
  ad_id: 23,
  campaign_id: 7,
  view: true,
  click: false
}
```

sum by campaign



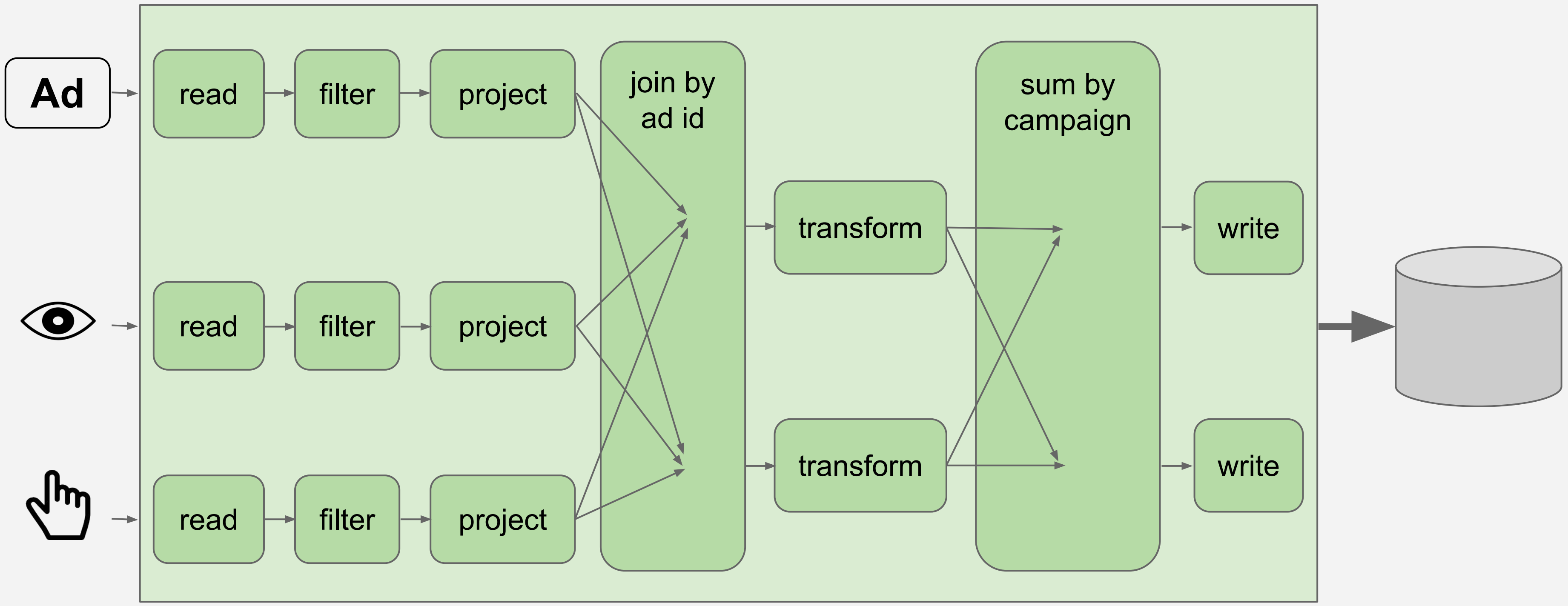

```
{
  campaign_id: 7,
  views: 2,
  clicks: 0
}
```

write




```
db.write(
   campaign_id=7,
   views=2,
   clicks=0,
)
```

Ad campaign metrics pipeline



I. Why stream processing?

II. Putting an application together

Example problem

Components and data operations

III. **Design principles and tradeoffs**

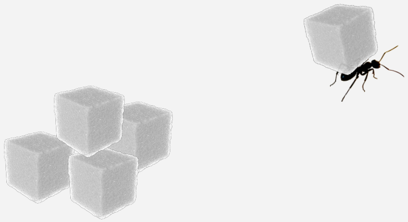
Horizontal scalability

Handling failures

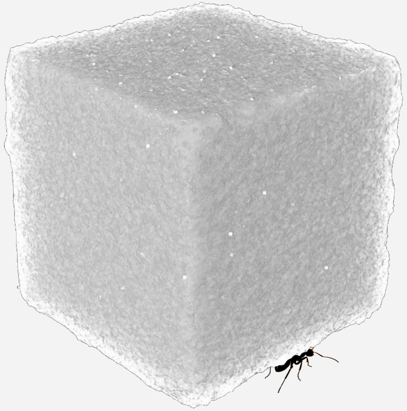
Idempotency

Consistency versus availability

Horizontal scalability: Basic idea

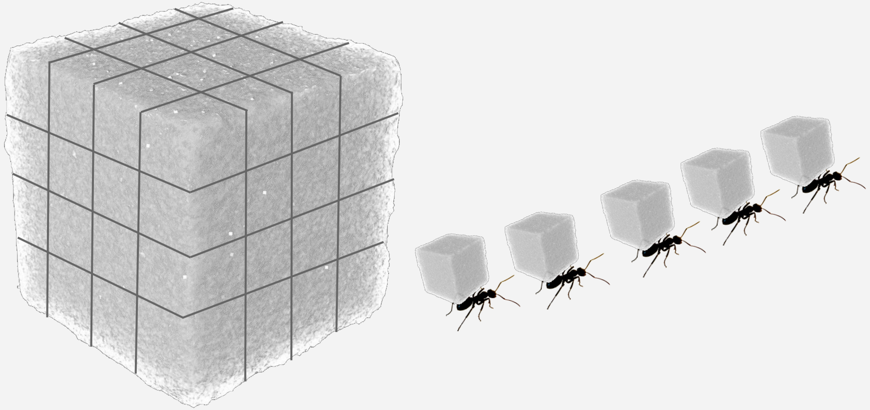


Horizontal scalability: Basic idea

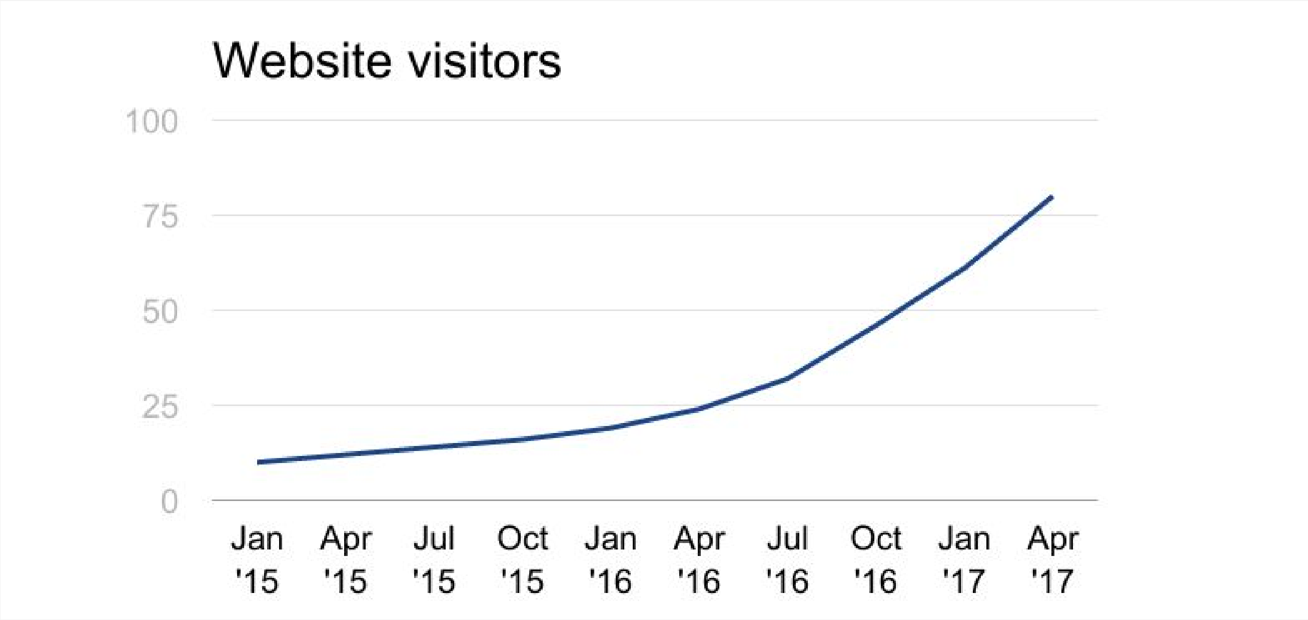


Horizontal scalability: Basic idea

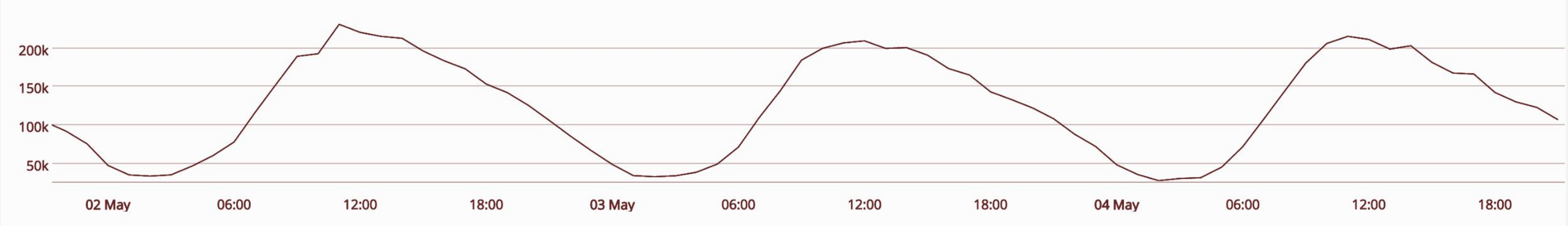
Horizontal scalability: Basic idea



Horizontal scalability: Why?

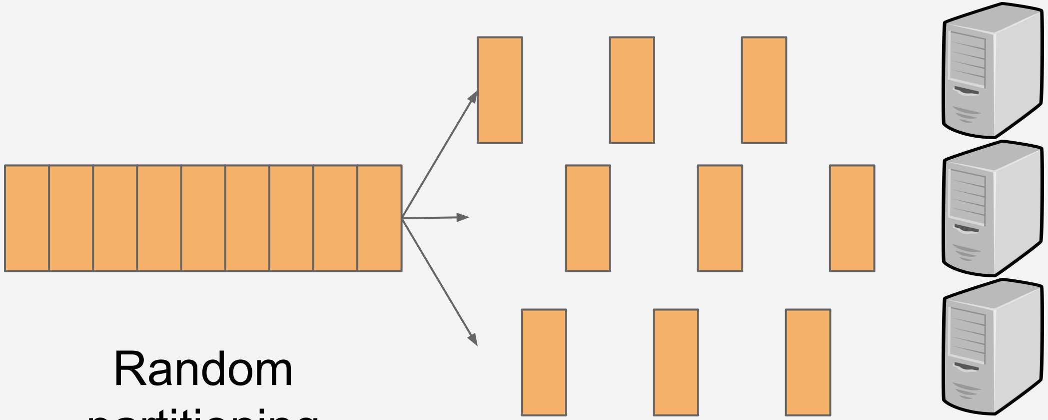


Horizontal scalability: Why?



Horizontal scalability: How?

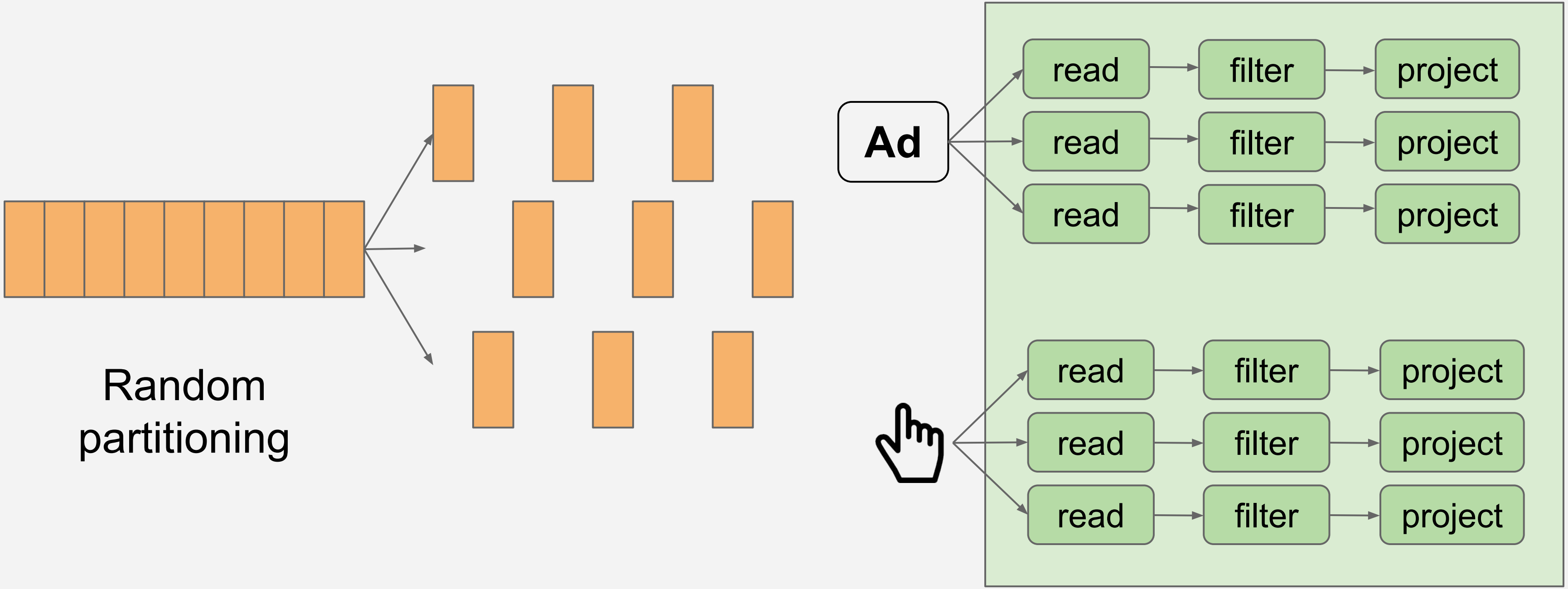
Partitioning



Random
partitioning

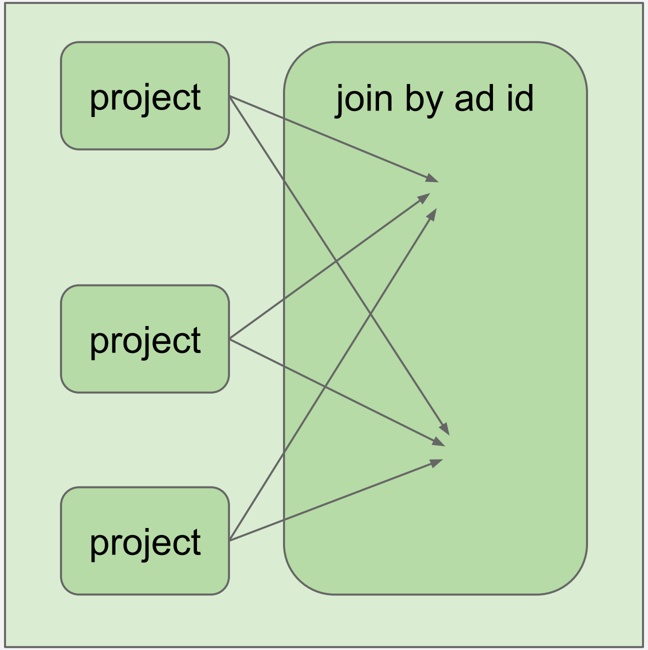
Horizontal scalability: How?

Partitioning



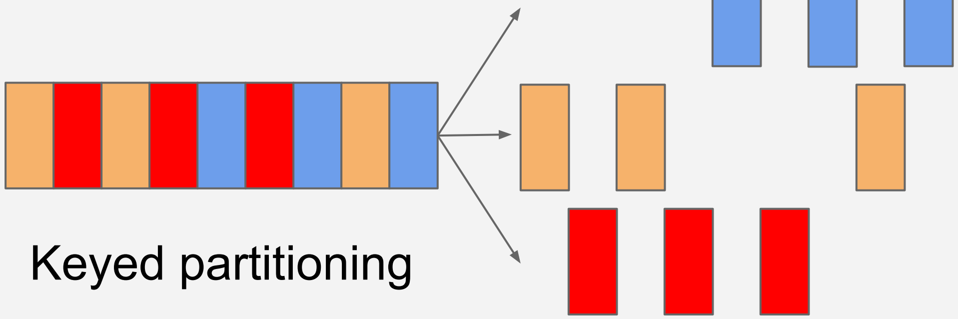
Horizontal scalability: How?

Partitioning

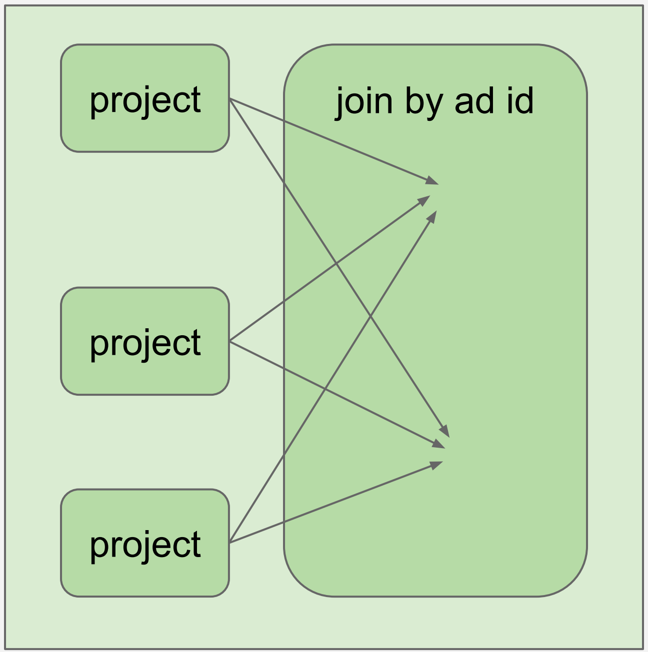


Horizontal scalability: How?

Partitioning

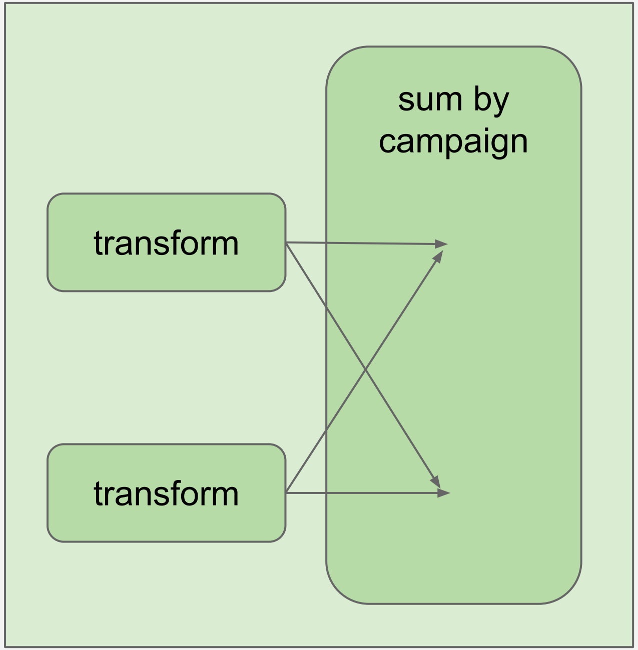


Keyed partitioning



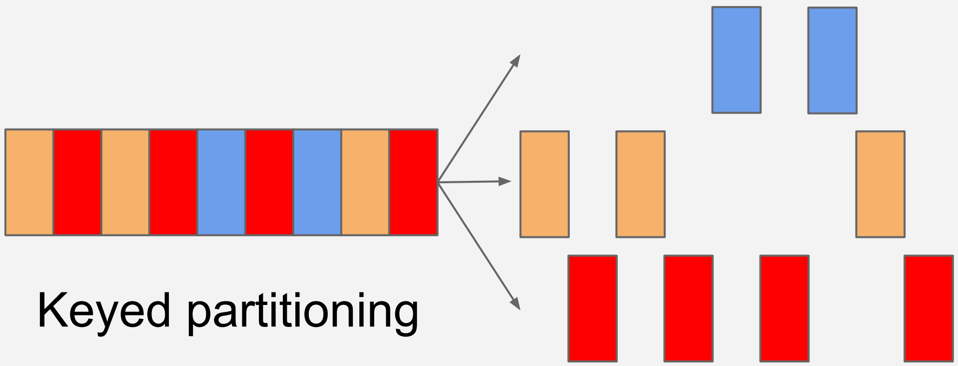
Horizontal scalability: watch out!

Hot spots / data skew

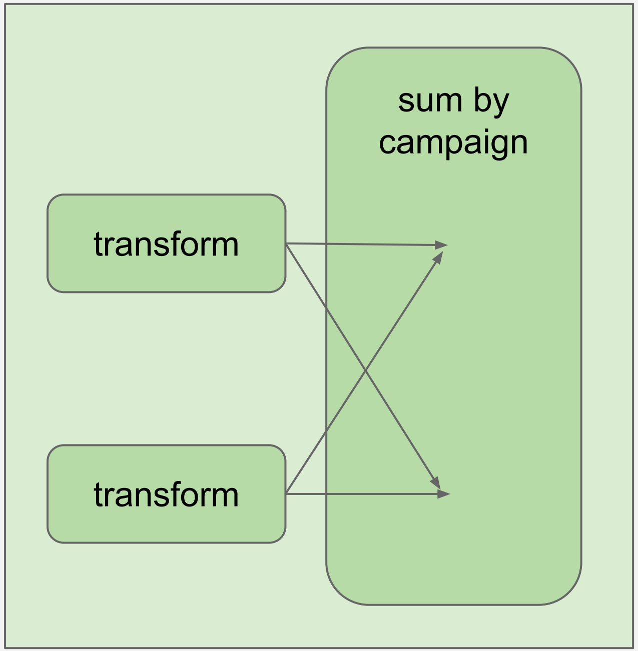


Horizontal scalability: watch out!

Hot spots / data skew



Keyed partitioning



Horizontal scalability: Summary

- Random partitioning for stateless transforms
- Keyed partitioning for keyed transformations
- Watch out for hot spots, and use appropriate mitigation strategy

I. Why stream processing?

II. Putting an application together

Example problem

Components and data operations

III. **Design principles and tradeoffs**

Horizontal scalability

Handling failures

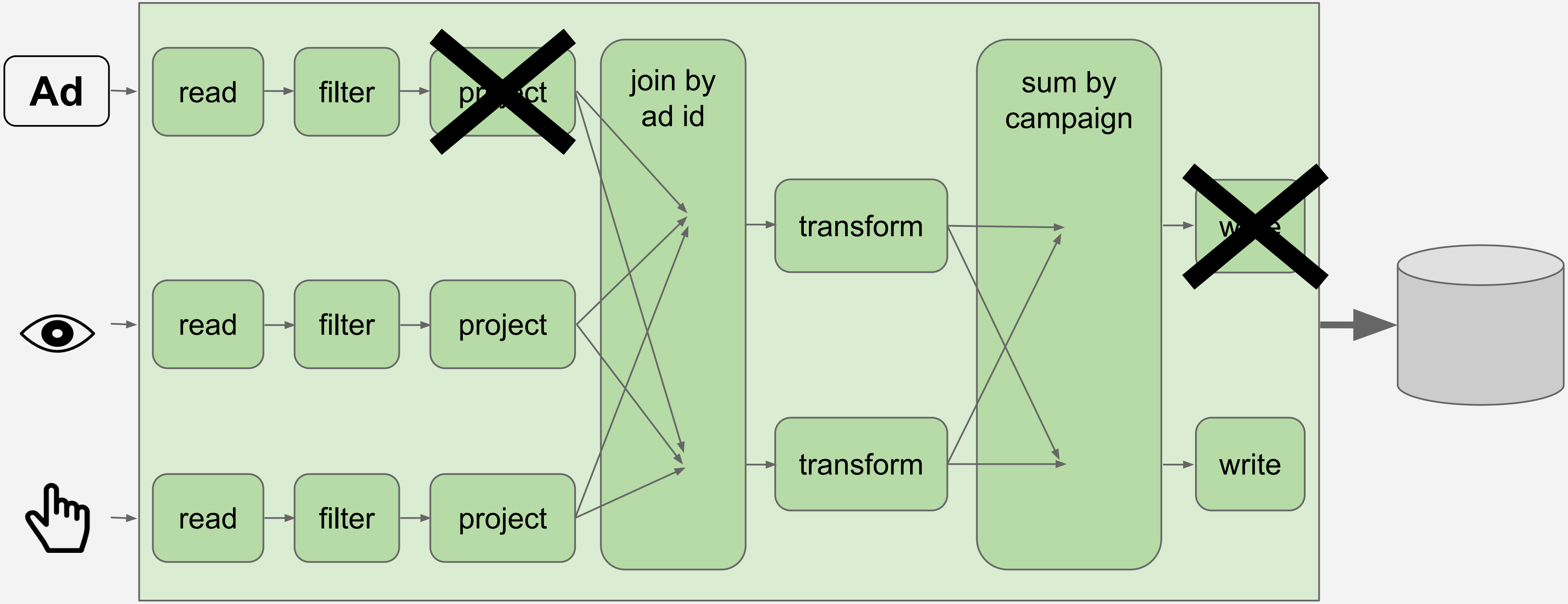
Idempotency

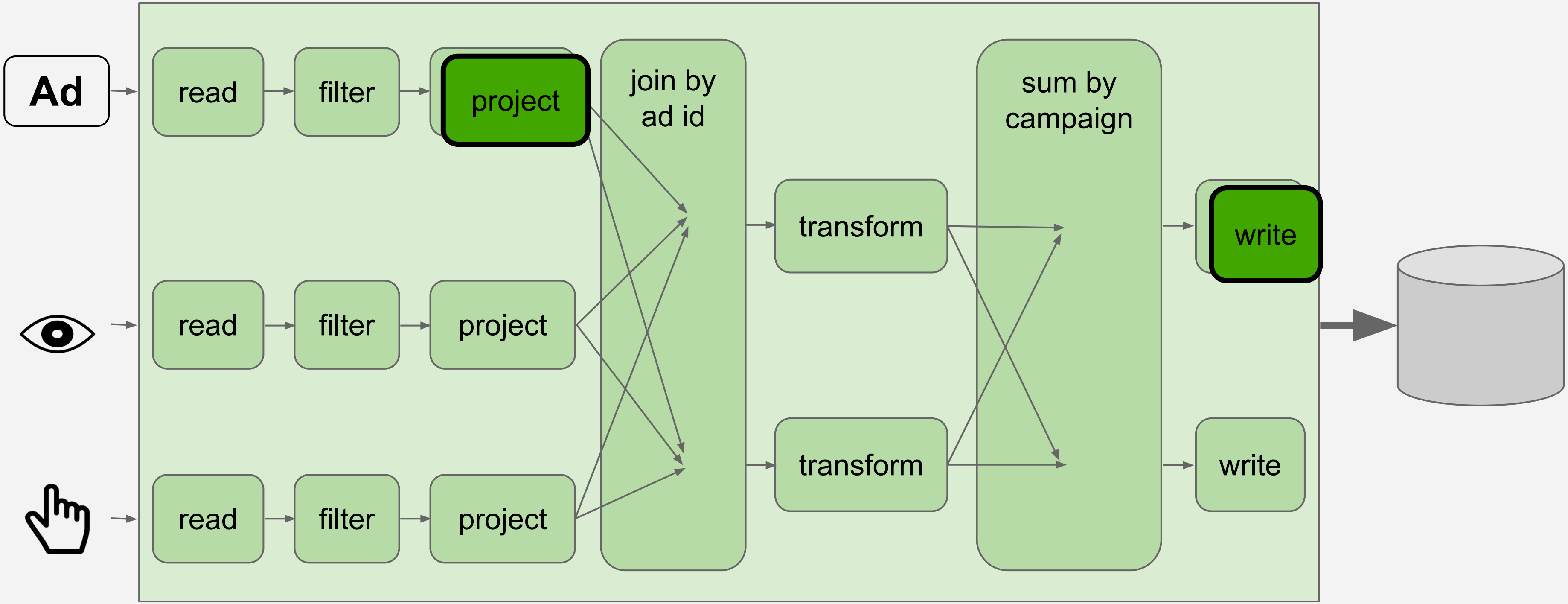
Consistency versus availability

Idempotency

Idempotency

An idempotent operation can be applied more than once and have the same effect.


Ad
read
filter
read
filter
project
read
filter
project
join by
ad id
transform
transform
sum by
campaign
write

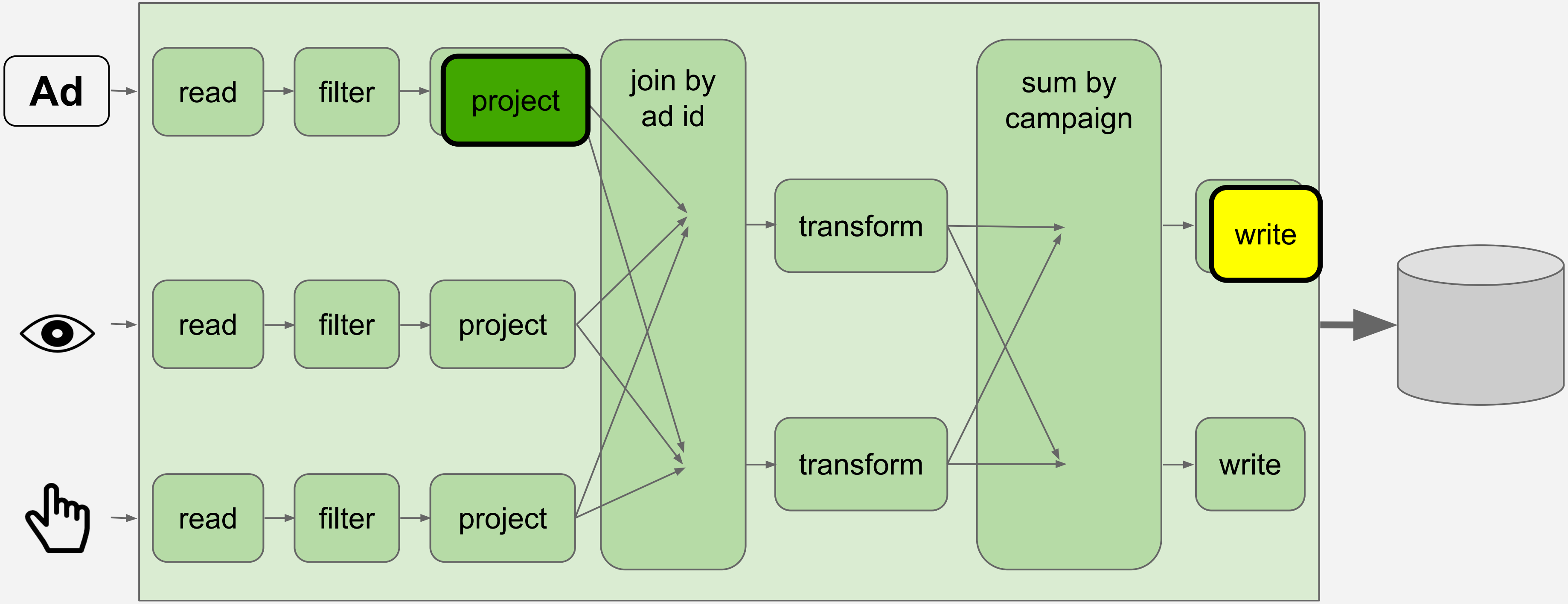

Ad
read
filter
project
read
filter
project
read
filter
project
join by
ad id
transform
transform
sum by
campaign
write
write

What operations are idempotent?

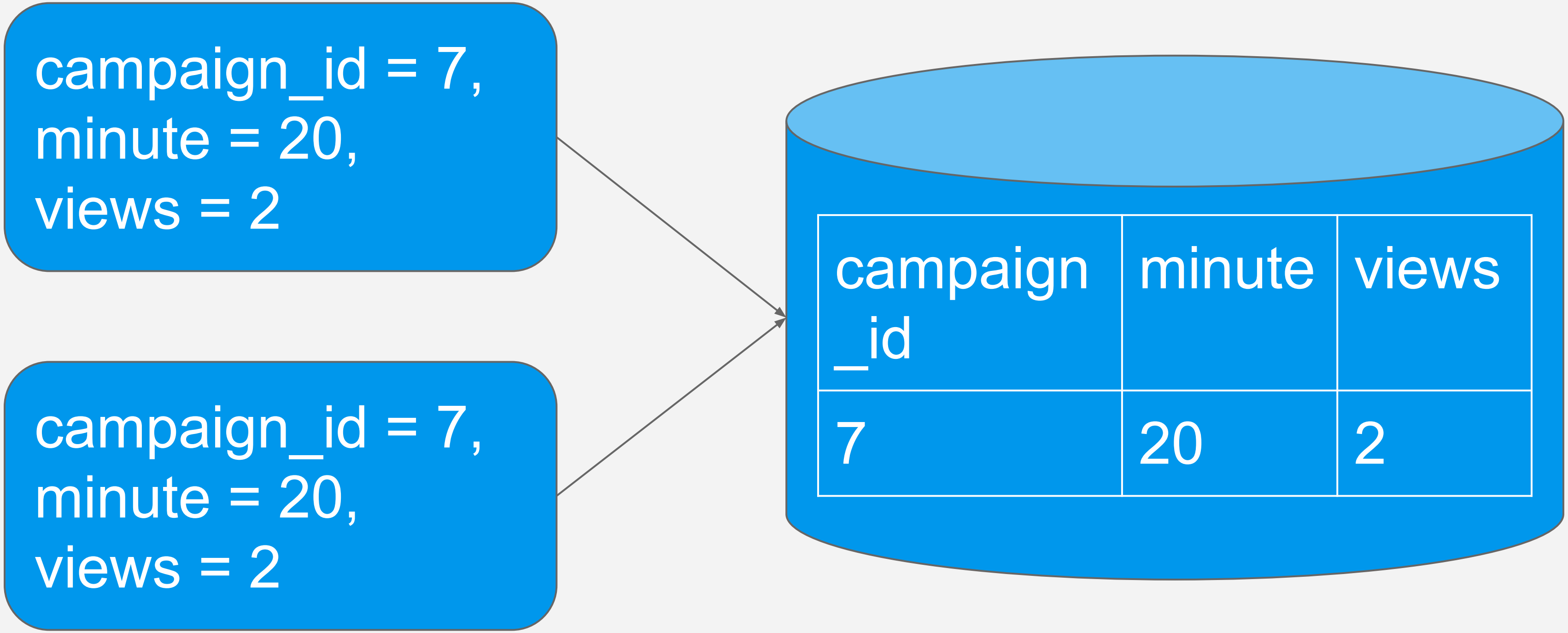
😃 Transforms: filters, projections, etc

No side effects!

🤔 Stateful operations


Ad
read
filter
project
read
filter
project
read
filter
project
join by
ad id
transform
transform
sum by
campaign
write
write

Idempotent writes with unique keys



Writes that aren’t idempotent

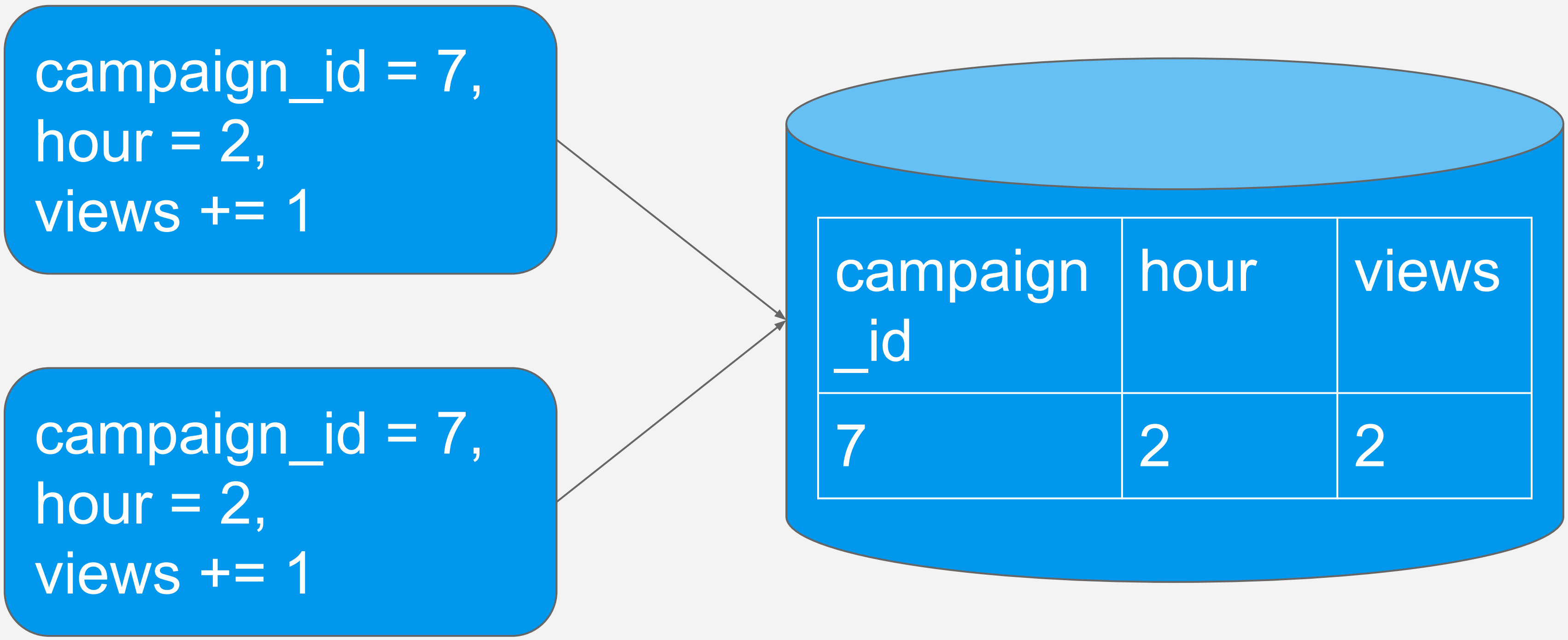
campaign_id	hour	views
7	2	0

Writes that aren’t idempotent

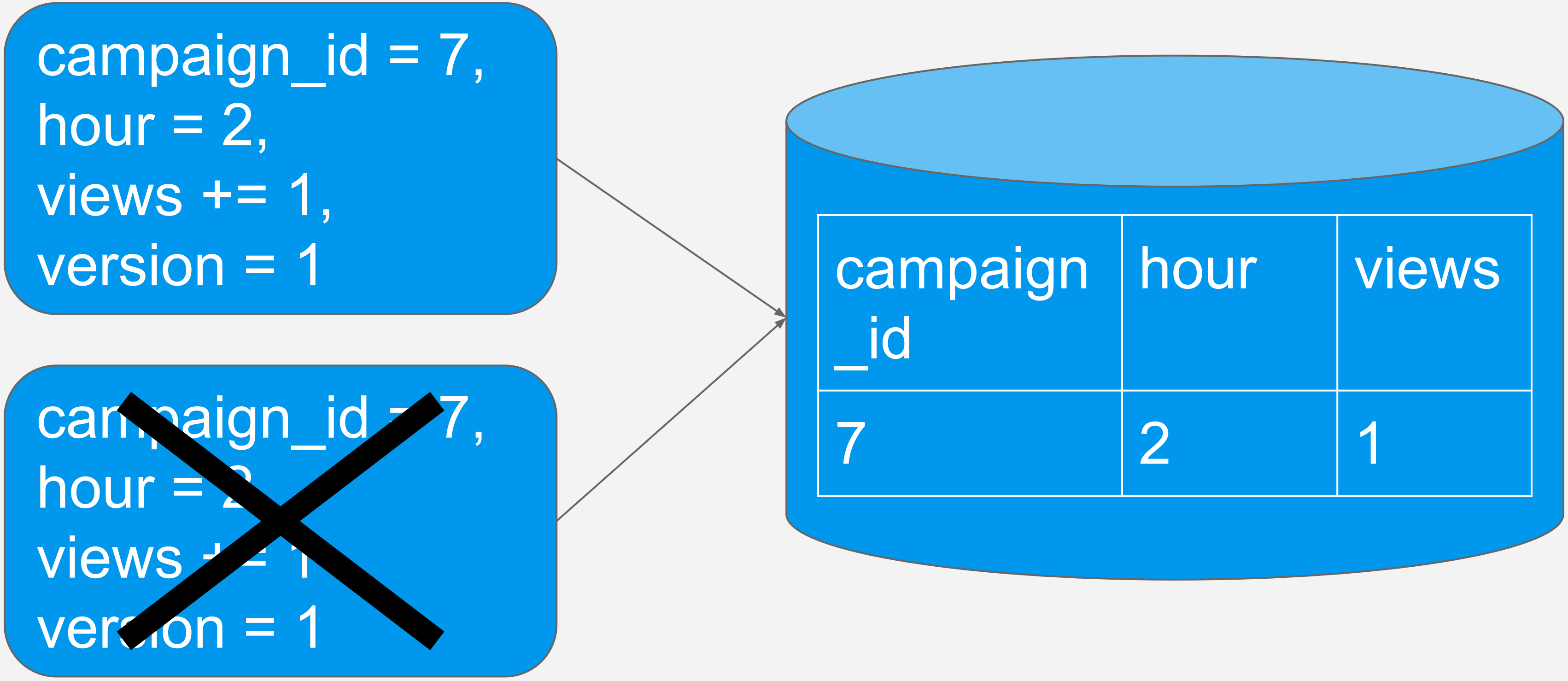
campaign_id = 7,
hour = 2,
views += 1

campaign_id	hour	views
7	2	1

Writes that aren't idempotent



Support for idempotency



Idempotency in streaming pipelines

Both in output to data sink and in local state (joining, aggregation)

Re-processing of events

- Some frameworks provide exactly once guarantees

Consistency vs. availability

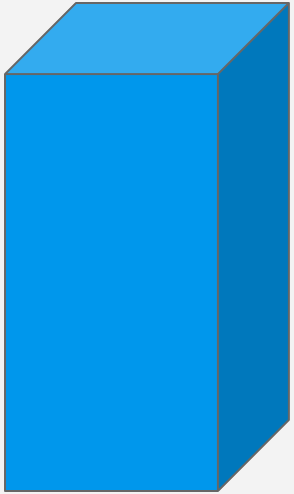
Always a tradeoff between **consistency** and **availability** when handling failures

Consistency

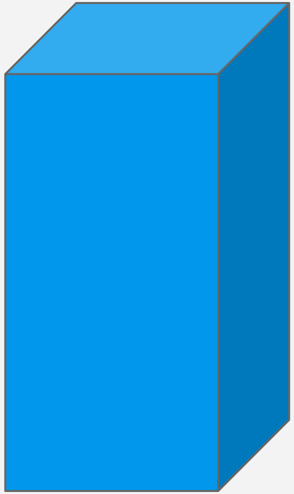
Every read sees a current view of the data.

Availability

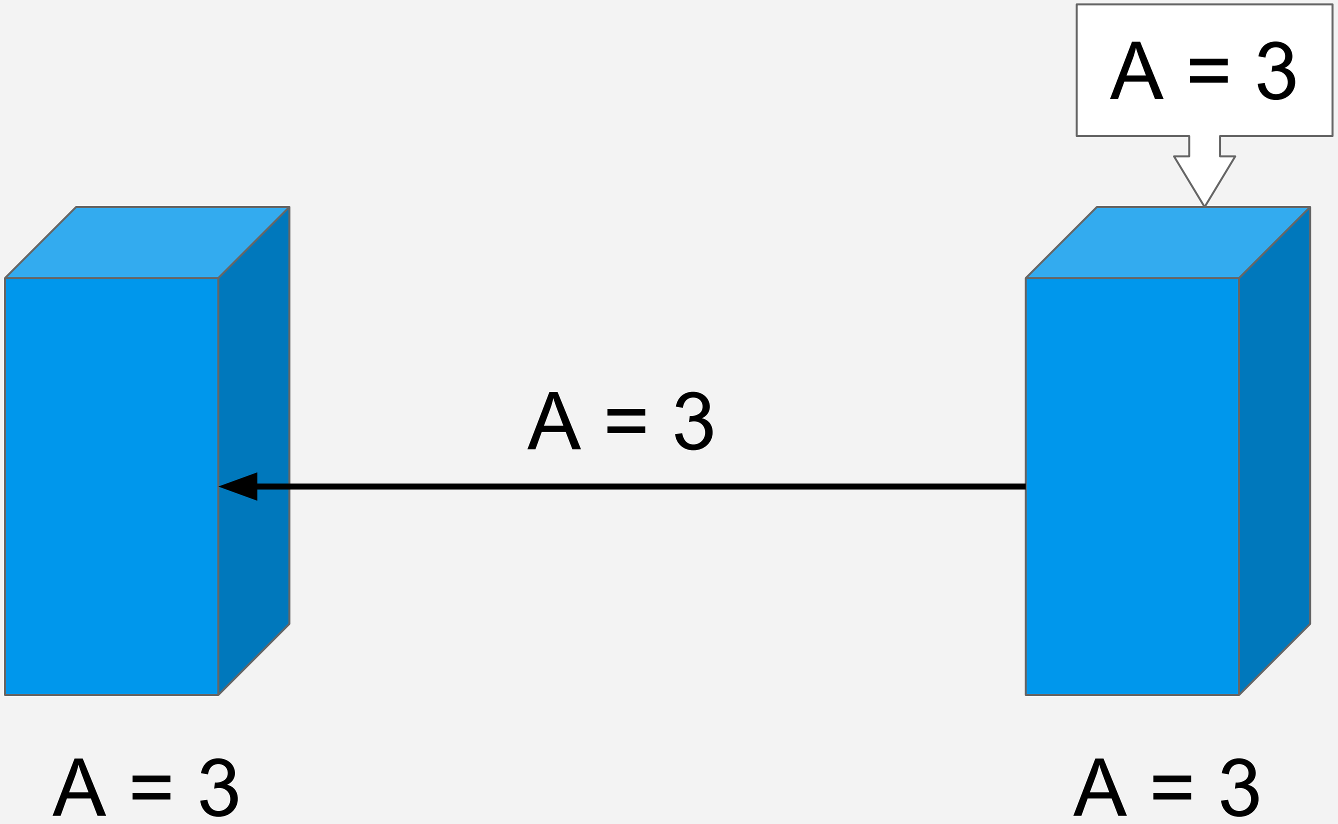
Capacity to serve requests

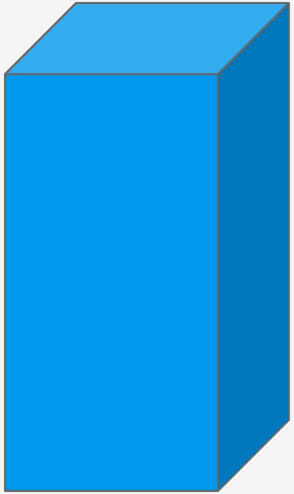


A = 9

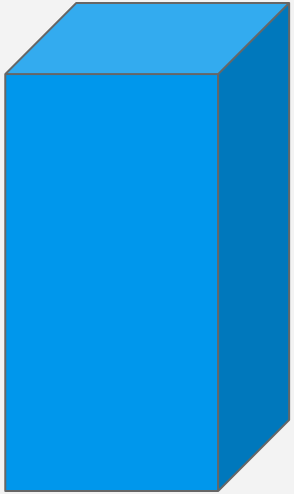


A = 9


A = 3
A = 3
A = 3
A = 3

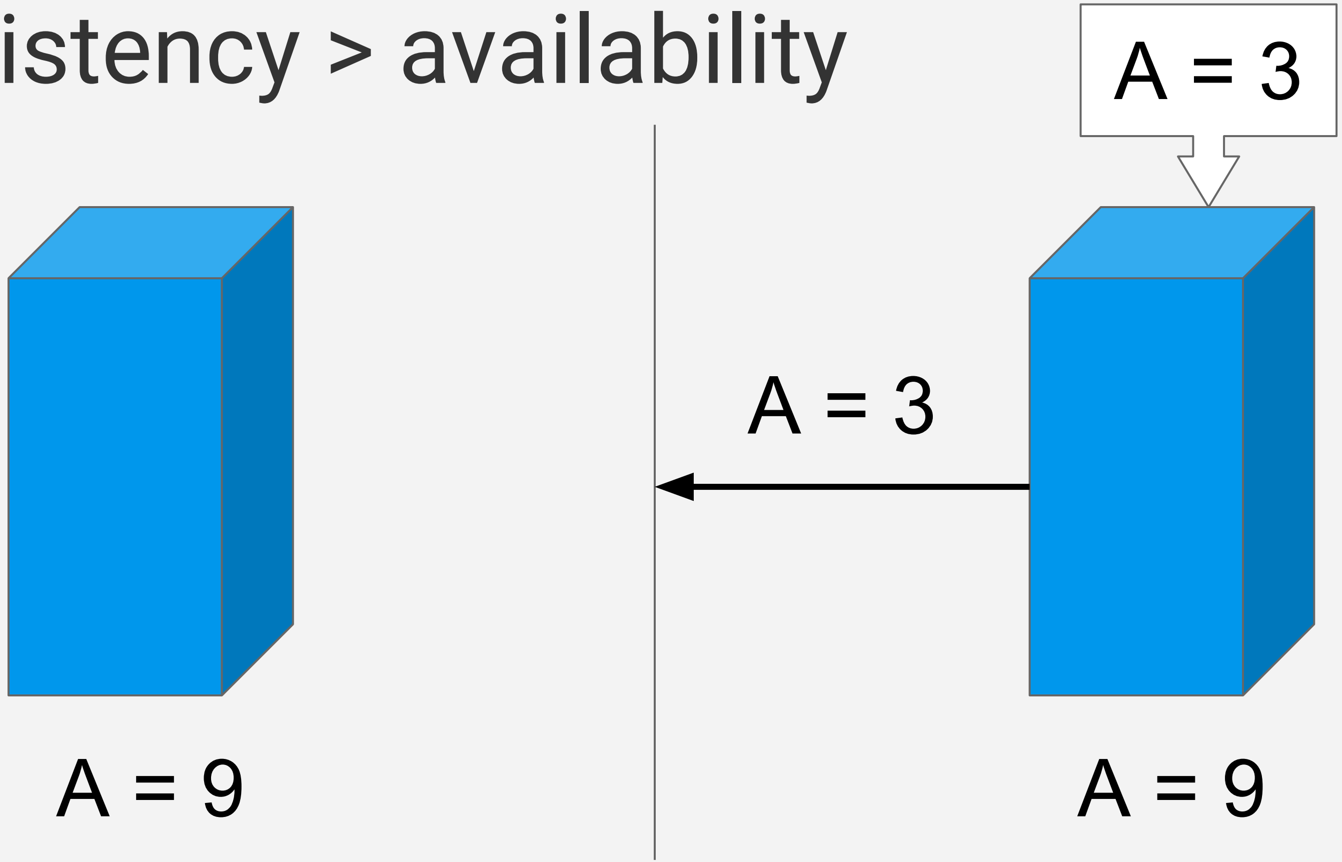


A = 9

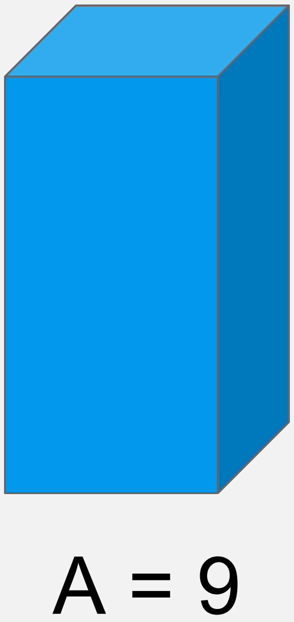


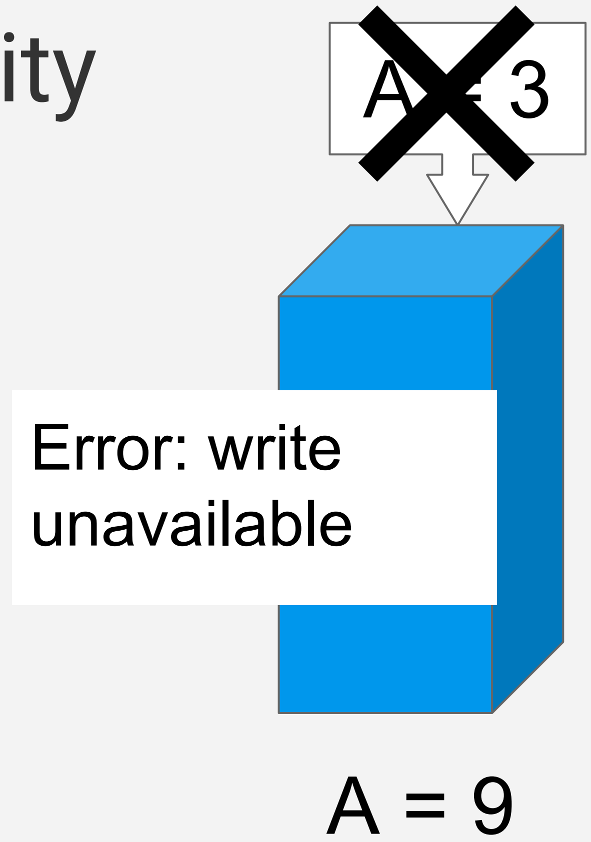
A = 9

Consistency > availability

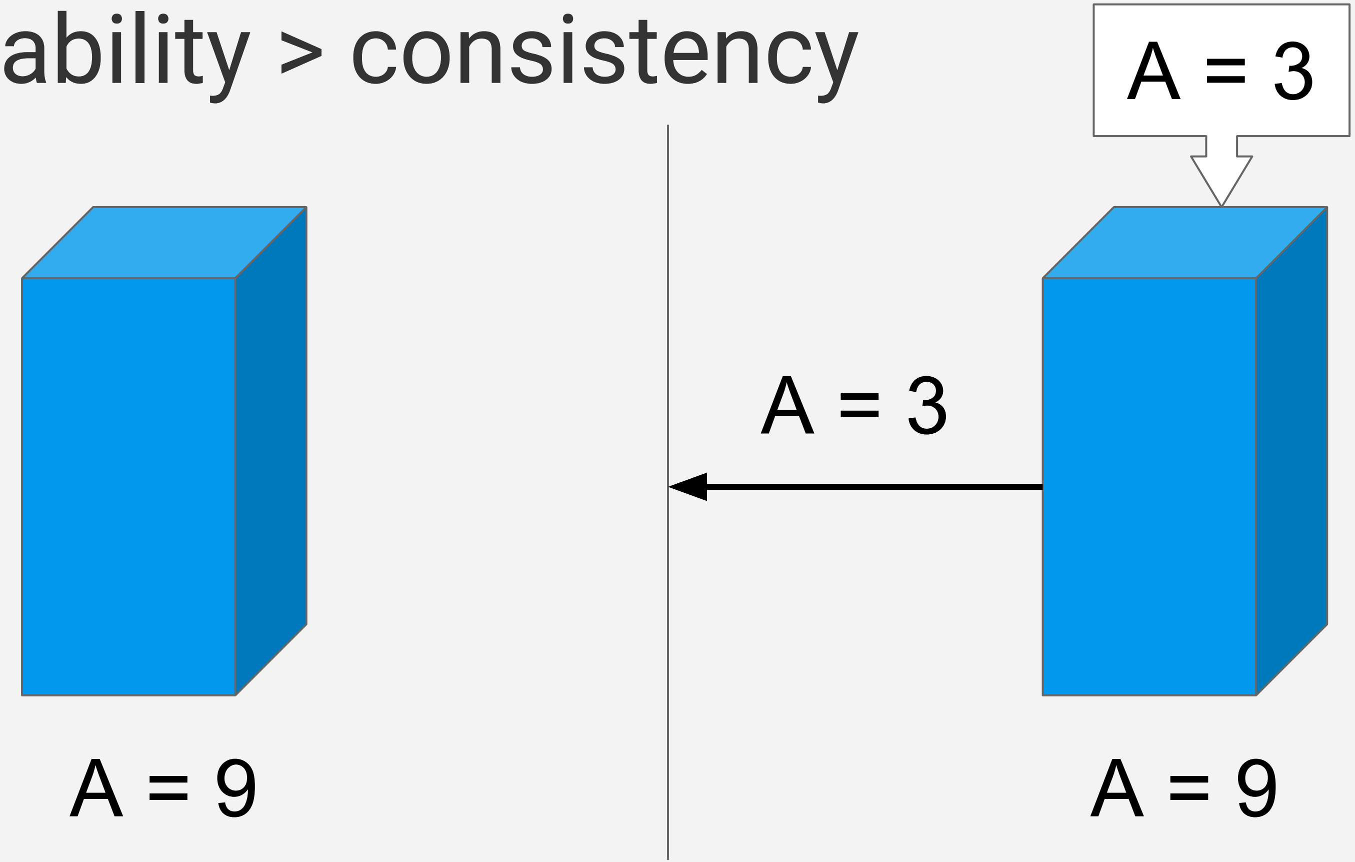


Consistency > availability

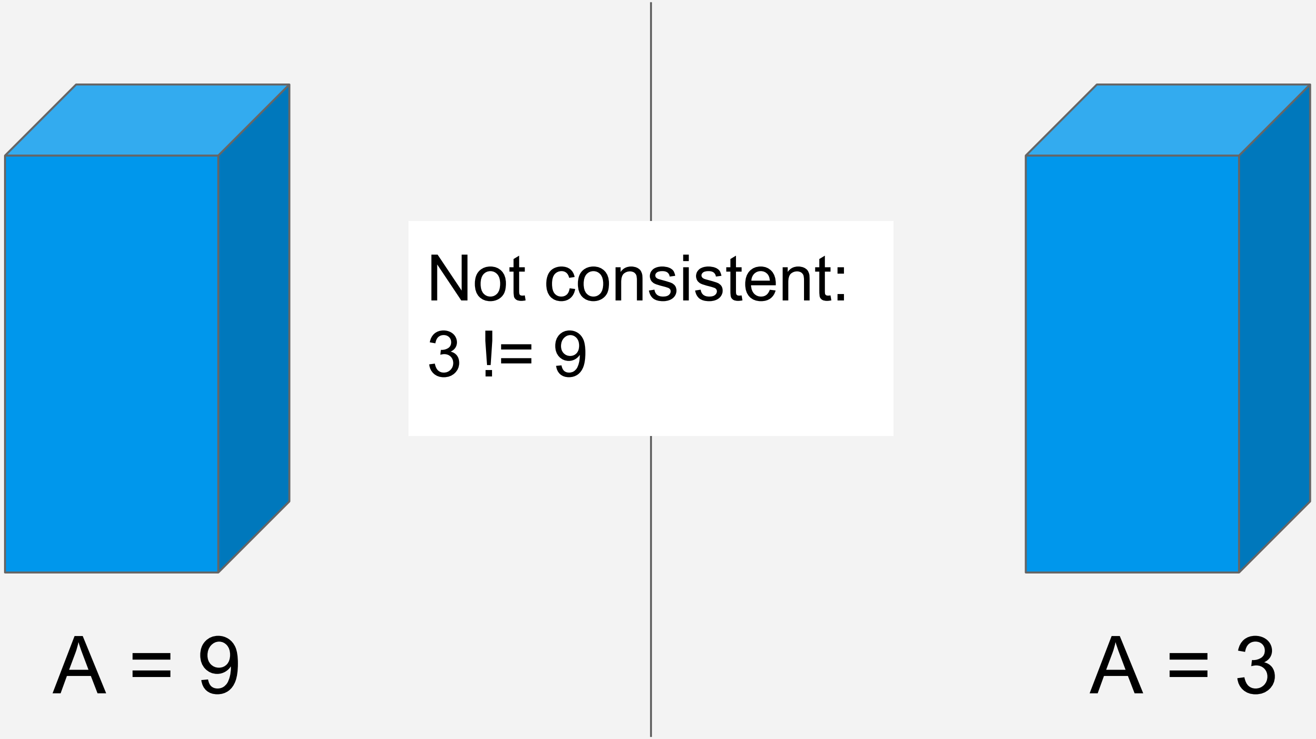




Availability > consistency

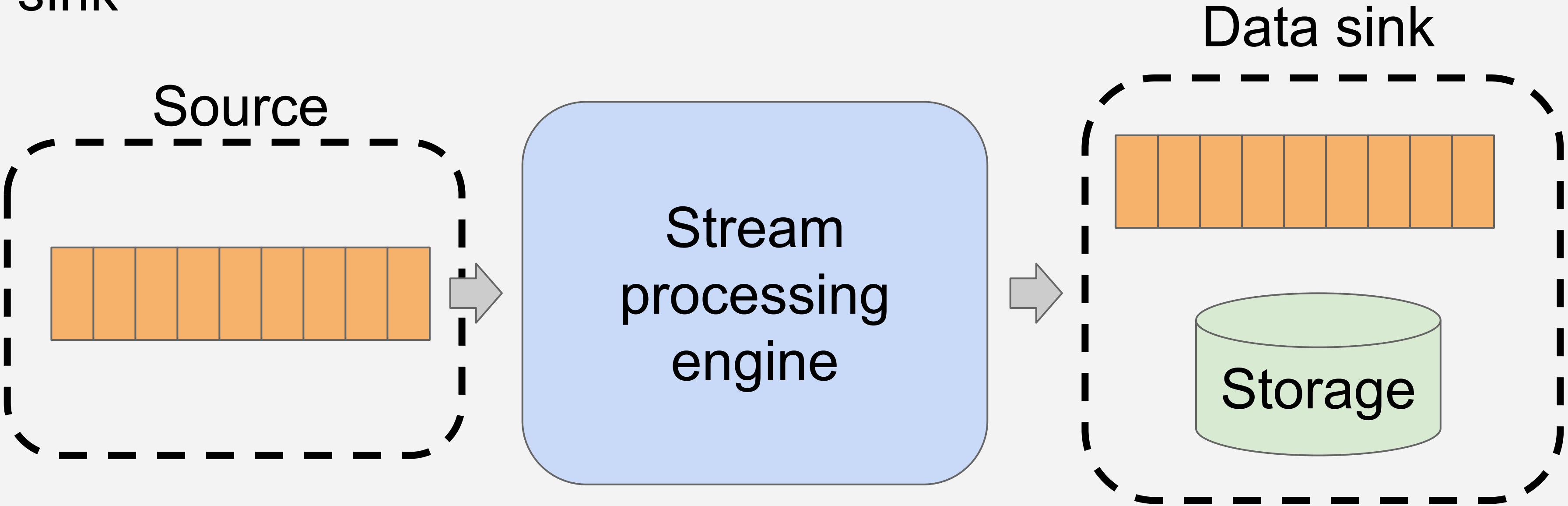


Availability > consistency



Prioritizing consistency or availability

Applies to systems for both your data source and data sink



Prioritizing consistency or availability

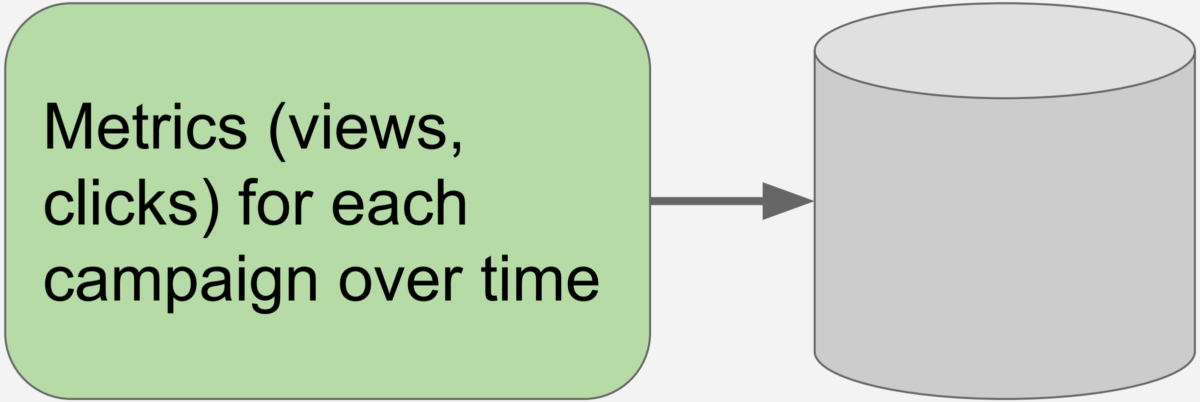
Applies to systems for both your data source and data sink

- Some systems pick one, be aware
- Others let you choose
  - ex. Cassandra - how many replicas respond to write?

Streaming applications run continuously

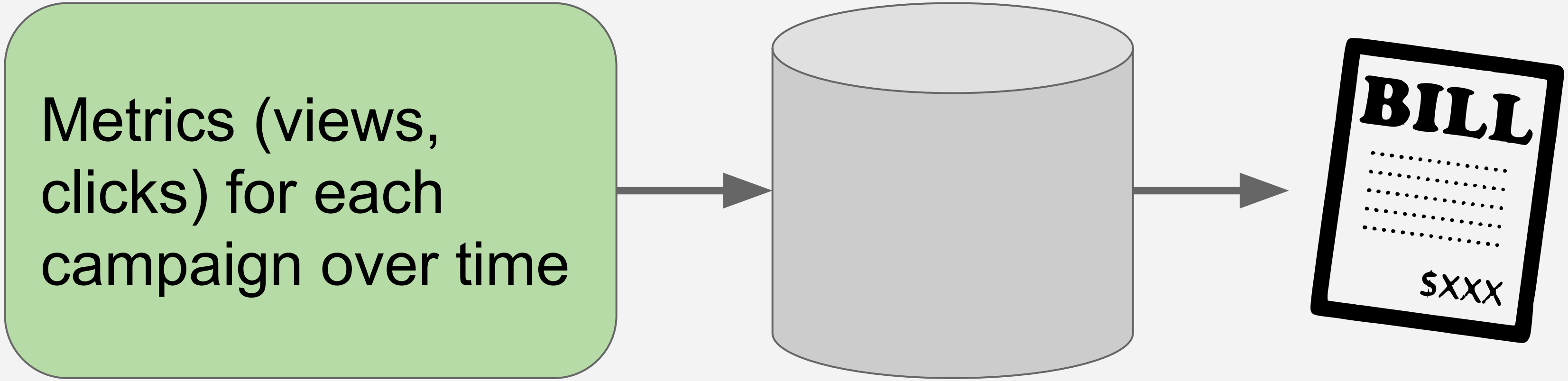
Prioritizing consistency or availability

Depends on the needs of your application



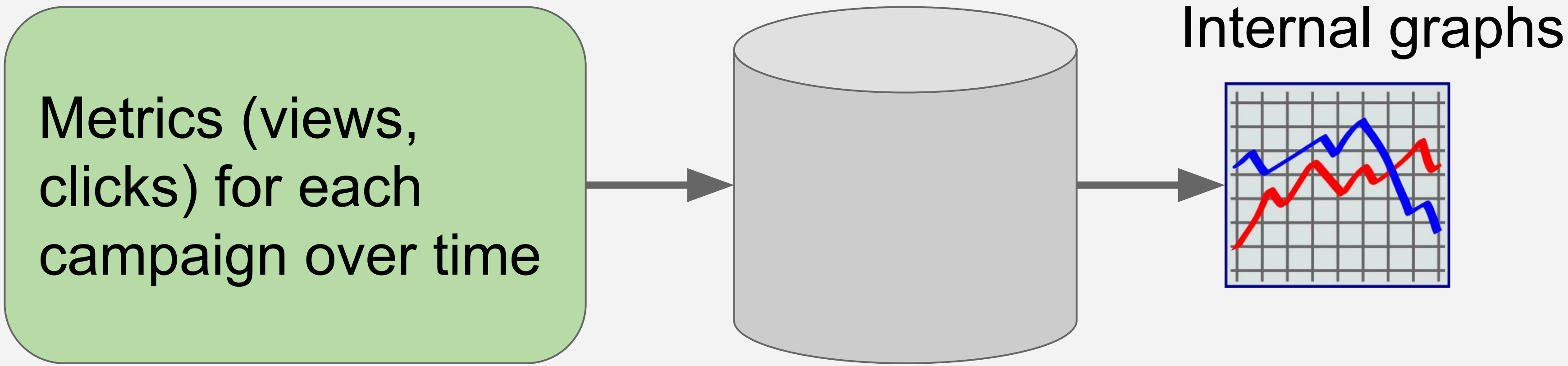
Prioritizing consistency or availability

More **consistency**



Prioritizing consistency or availability

More **availability**



Conclusion

- Stream processing: data processing with operations on events or windows of events
- Horizontal scalability, as data will grow and change over time
- Handle failures appropriately
  - Keep operations idempotent, for retries
  - Tradeoff between availability and consistency


We're Hiring!
www.yelp.com/careers/
EAT24

fb.com/YelpEngineers

@YelpEngineering

engineeringblog.yelp.com

github.com/yelp