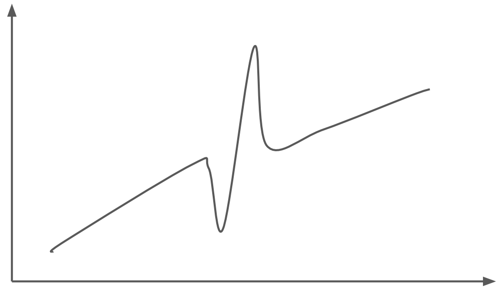
Eventually, time will kill your data pipeline

Berlin Buzzwords, 2019-06-17
Lars Albertsson
Mimeria

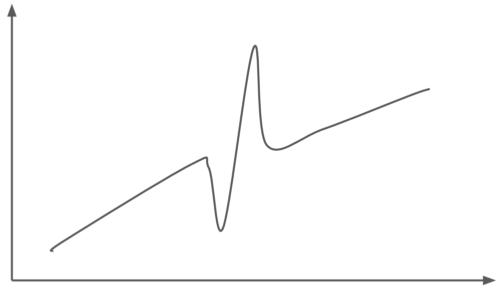
Out of joint



“Time is out of joint. O cursed spite,
That ever I was born to set it right.”

- Hamlet, prince of Denmark

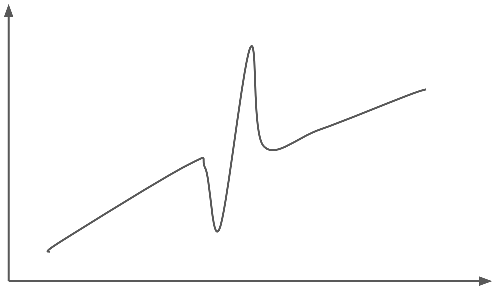
Out of joint



"Time is out of joint. O cursed spite,
That ever I was born to set it right."

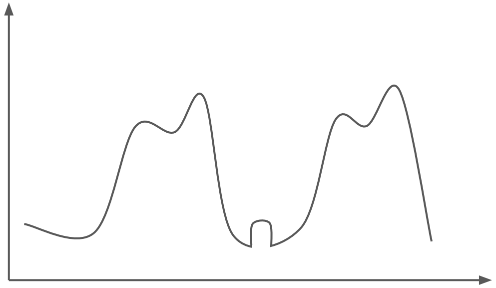
- Hamlet, prince of Denmark

Out of joint

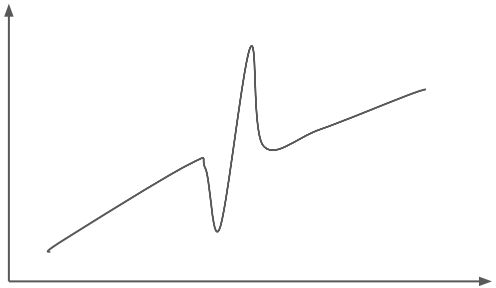


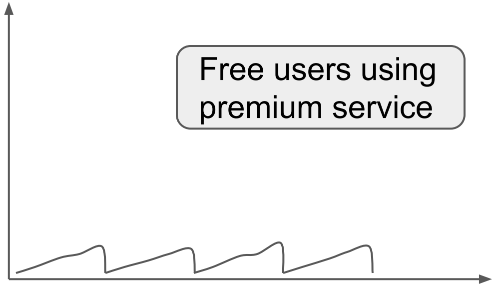
“Time is out of joint. O cursed spite,
That ever I was born to set it right.”

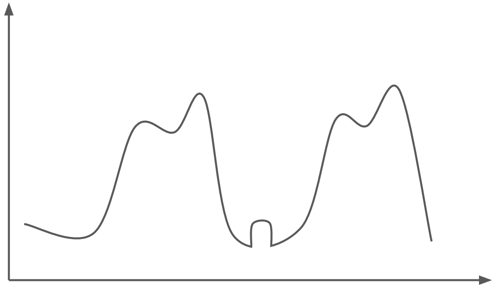
- Hamlet, prince of Denmark



Out of joint







Time will kill us all

- Time handling causes data processing problems
- Observed issues
- Principles, patterns, anti-patterns

Goals:

- Awareness, recognition
- Tools from my toolbox

Data categories, time angle

- Facts
  - Events, observations
  - Time stamped by clock(s)
  - Continuous stream

Data categories, time angle

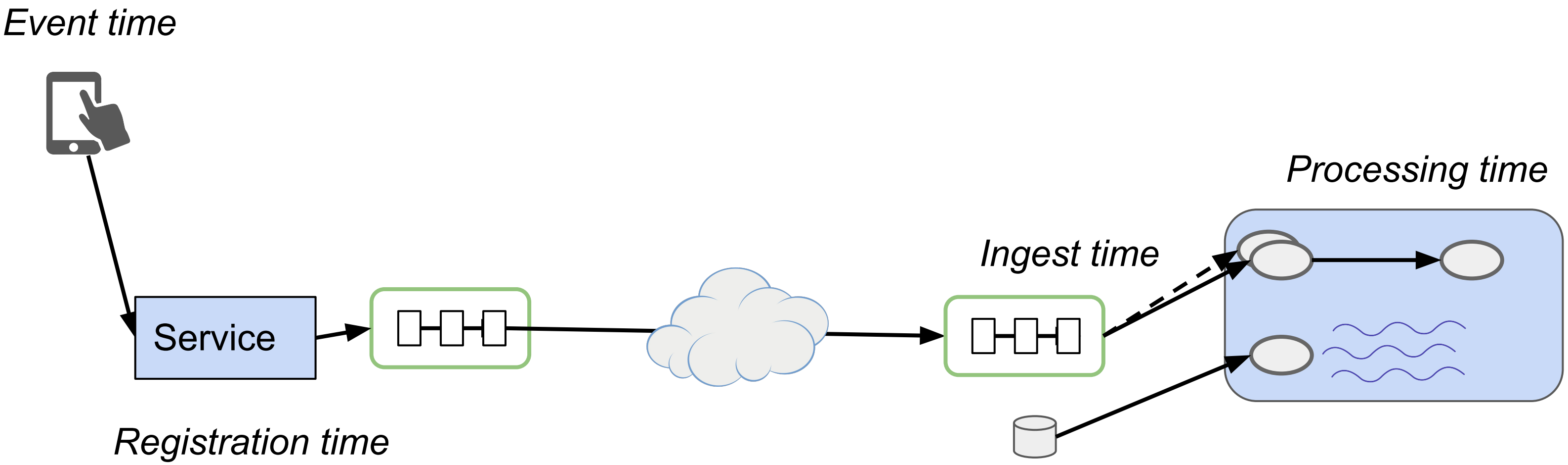
- Facts
  - Events, observations
  - Time stamped by clock(s)
  - Continuous stream

- State
  - Snapshot view of system state
  - ~~Lookup in live system.~~ Dumped to data lake.
  - Regular intervals (daily)

Data categories, time angle

- Facts
  - Events, observations
  - Time stamped by clock(s)
  - Continuous stream
- State
  - Snapshot view of system state
  - ~~Lookup in live system.~~ Dumped to data lake.
  - Regular intervals (daily)
- Claims
  - Statement about the past
  - Time window scope

Time scopes

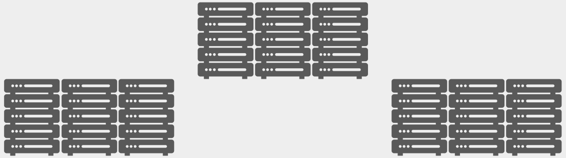


Domain time - scope of claim. “These are suspected fraudulent users for March 2019.”

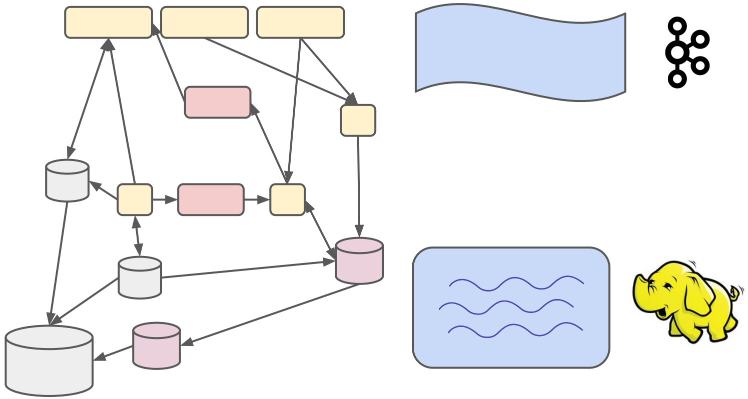
Clocks

- Computer clocks measure *elapsed time*
- Good clocks, bad clocks, wrong clocks

Clients - bad clocks
Time zone?



Legacy systems - good clocks
Time zone?



Controlled backend - good clocks, UTC

Calendars, naive

Maps time to astronomical and social domains

Naive calendar

- 365 days + leap years
- 12 months, weird number of days
- 7 day weeks
- 24 hours
- 60 minutes
- 60 seconds
- 24 hour time zones

Naive calendar properties

```
t + 1.day == t + 86400.seconds

(y + 45.years).asInt == y.asInt + 45

(date(y, 1, 1) + 364.days).month == 12

(t + 60.minutes).hour == (t.hour + 1) % 24

datetime(y, mon, d, h, m, s, tz).inZone(tz2).day in [d - 1, d, d + 1]

datetime(y, mon, d, h, m, s, tz).inZone(tz2).minute == m

date(y, 12, 29) + 1.day == date(y, 12, 30)

(date(y, 2, 28) + 2.days).month == 3
```

Can you find the counter examples?

Calendar reality

```
t + 1.day == t + 86400.seconds
```

Leap seconds

```
(y + 45.years).asInt == y.asInt + 45
```

Year zero

```
(date(y, 1, 1) + 364.days).month == 12
```

Julian / Gregorian calendar switch

```
(t + 60.minutes).hour == (t.hour + 1) % 24
```

Daylight savings time

```
datetime(y, mon, d, h, m, s, tz).inZone(tz2).day in [d - 1, d, d + 1]

datetime(y, mon, d, h, m, s, tz).inZone(tz2).minute == m
```

Time zones:
Span 26 hours
15 minute granularity

```
date(y, 12, 29) + 1.day == date(y, 12, 30)
```

Samoa crosses date line. Again.

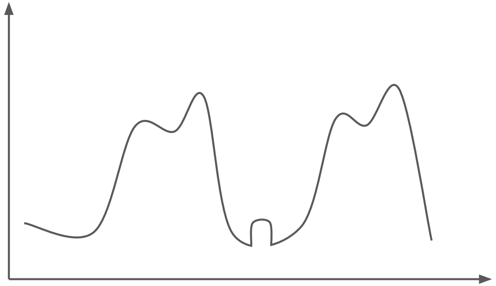
```
(date(y, 2, 28) + 2.days).month == 3
```

Sweden: 1712-02-30

Small problems in practice

Except DST

- Analytics quality
- Changes with political decision
- Might affect technical systems
- (Affects health!)



Leap seconds

Year zero

Julian / Gregorian calendar switch

Daylight savings time

Time zones:
Span 26 hours
15 minute granularity

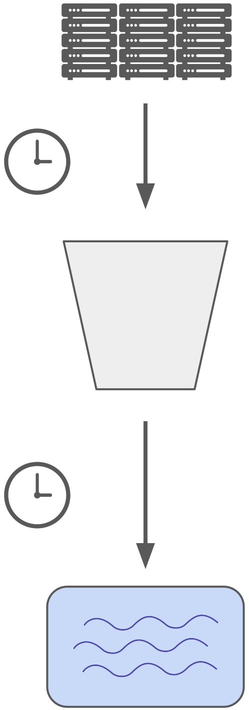
Samoa crosses date line. Again.

Sweden: 1712-02-30

Typical loading dock ingress

- File arrives every hour
- Ingest job copies to lake, applies data platform conventions
- Source system determines format, naming, and

  timestamps

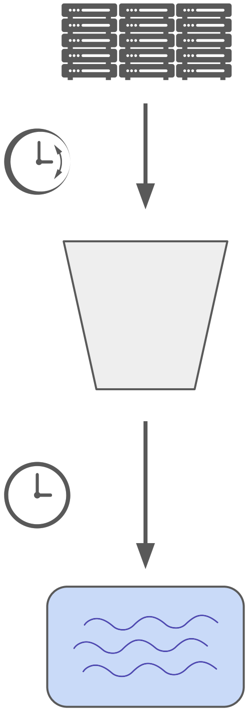


```
class SalesArrived(ExternalTask):
  """Receive a file with sales transactions every hour."""
  hour = DateHourParameter()

  def matches(self):
    return [u for u in GCSClient().listdir(f'gs://ingress/sales/')
            if re.match(rf'{self.hour:%Y%m%d%h}.*\.json', u)]

  def output(self):
    return GCSTarget(self.matches()[0])


@requires(SalesArrived)
class SalesIngest(Task):
  def output(self):
    return GCSFlagTarget('gs://lake/sales/'
      f'{self.hour:year=%Y/month=%m/day=%d/hour=%h}/')

  def run(self):
    _, base = self.input().path.rsplit('/', 1)
    dst = f'{self.output().path}{base}'
    self.output().fs.copy(src.path, dst)
    self.output().fs.put_string('',
      f'{self.output().path}{self.output().flag)}'


crontab:
*/10 * * * * luigi RangeHourly --of SalesIngest
```

Typical loading dock ingress

- File arrives every hour
- Ingest job copies to lake, applies data platform conventions
- Source system determines format, naming, and

  timestamps, ***incl. zone***
- Spring: halted ingest
  Autumn: data loss

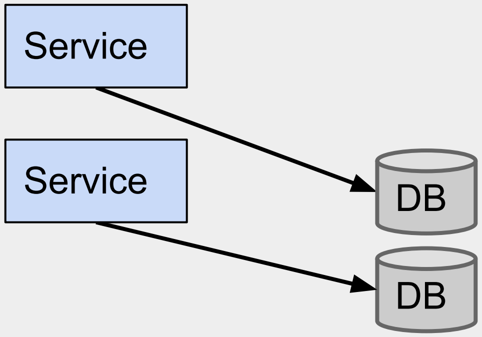


```
class SalesArrived(ExternalTask):
  """Receive a file with sales transactions every hour."""
  hour = DateHourParameter()

  def matches(self):
    return [u for u in GCSClient().listdir(f'gs://ingress/sales/')
            if re.match(rf'{self.hour:%Y%m%d%h}.*\.json', u)]

  def output(self):
    return GCSTarget(self.matches()[0])


@requires(SalesArrived)
class SalesIngest(Task):
  def output(self):
    return GCSFlagTarget('gs://lake/sales/'
      f'{self.hour:year=%Y/month=%m/day=%d/hour=%h}/')

  def run(self):
    _, base = self.input().path.rsplit('/', 1)
    dst = f'{self.output().path}{base}'
    self.output().fs.copy(src.path, dst)
    self.output().fs.put_string('',
      f'{self.output().path}{self.output().flag)}'


crontab:
*/10 * * * * luigi RangeHourly --of SalesIngest
```

Offline / online

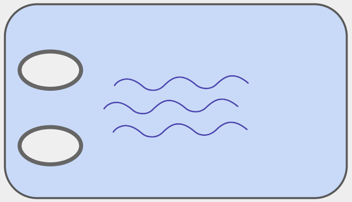
Online:

Processing time ~= event time



Offline:

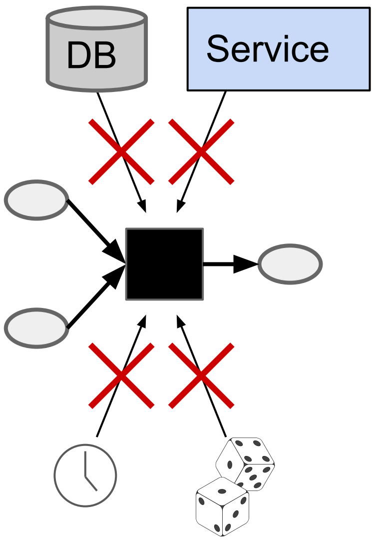
Processing time != event time



Batch job - functional principles

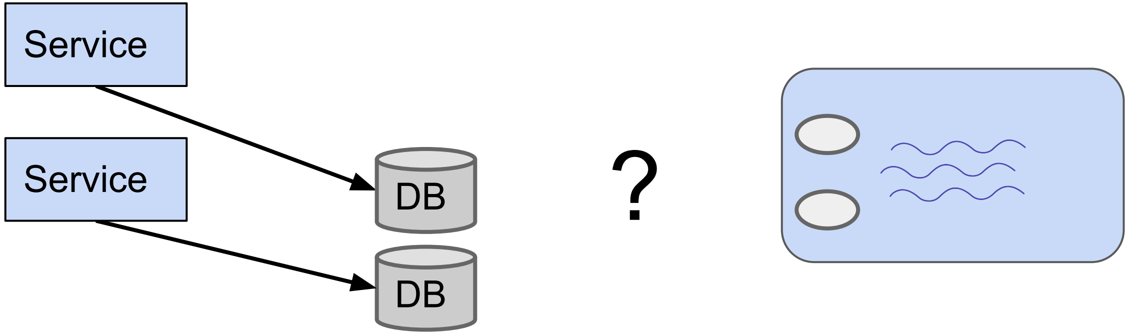
Job == function([input datasets]): [output datasets]

- Ideally: atomic, deterministic, idempotent
- No external factors → deterministic
  - No (mutable) database queries
  - No service lookup
  - Don't read wall clock
  - No random numbers
- Known, bounded input data
- No orthogonal concerns & input factors
  - ~~Invocation~~
  - ~~Scheduling~~
  - ~~Input / output location~~
- No side-effects



Database dumping

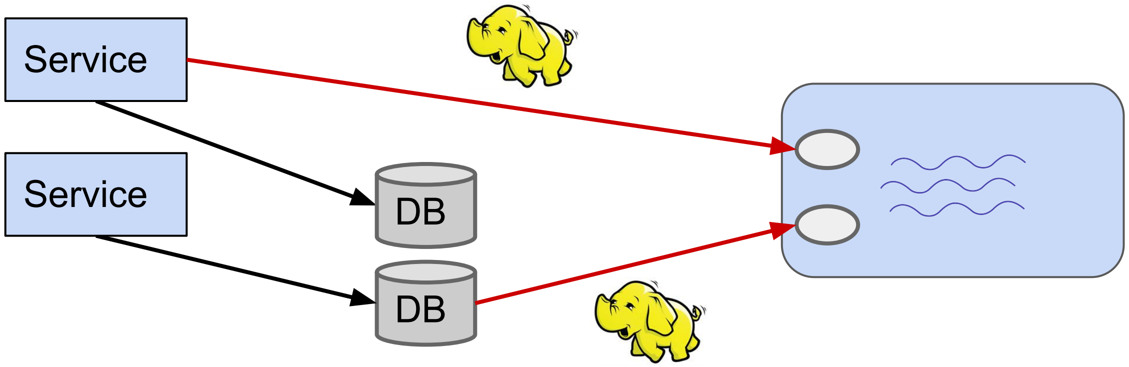
- Simple approach: Daily full table snapshot to cluster storage dataset.
- Easy on surface...



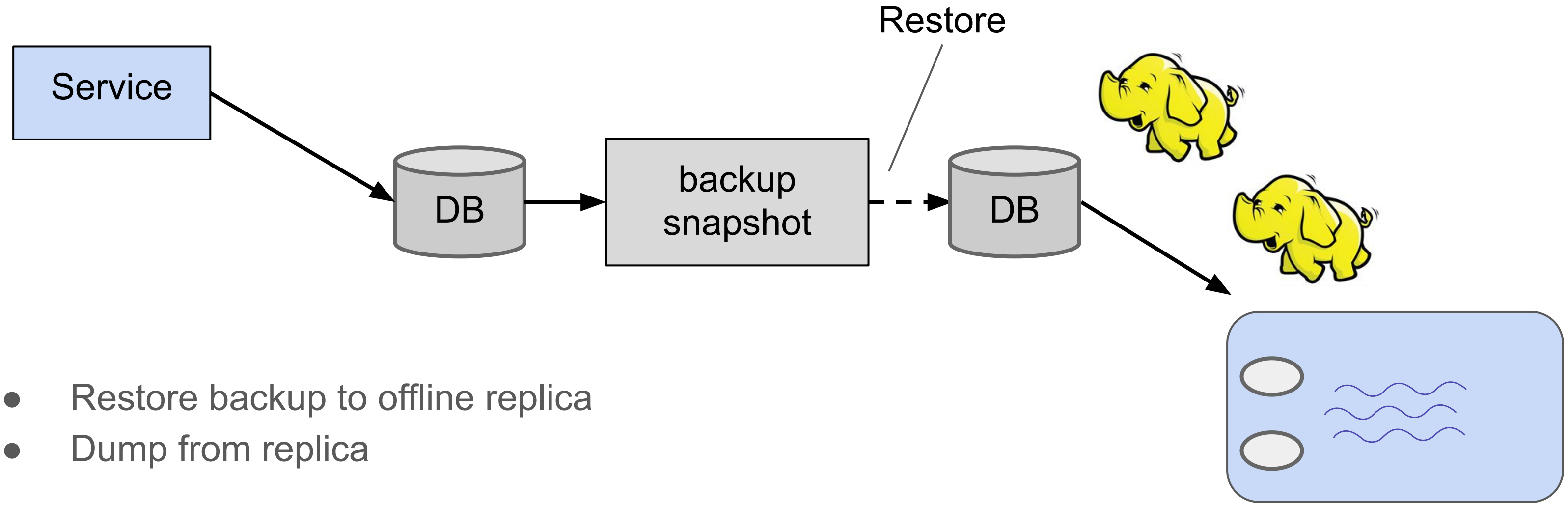
Anti-pattern: Death by elephant

- ~~Sqoop (dump with MapReduce) production DB~~
- ~~MapReduce from production API~~

Hadoop / Spark == internal DDoS service

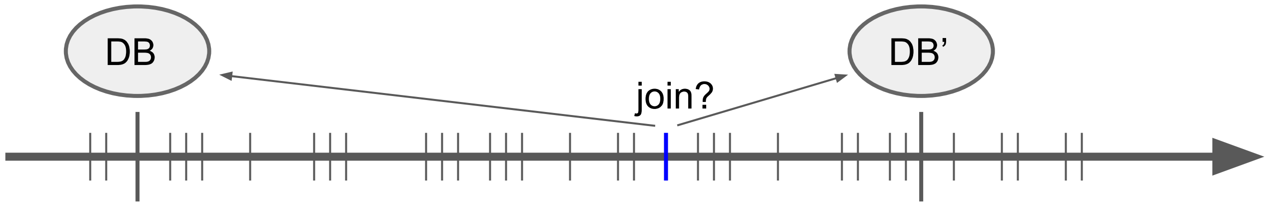


Pattern: Offline replica



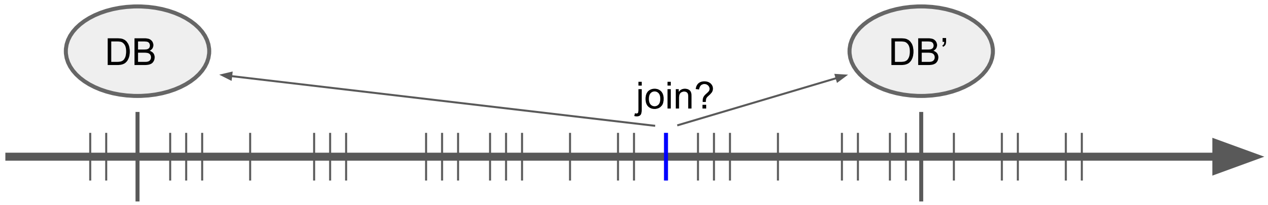
- Restore backup to offline replica
- Dump from replica

Using snapshots

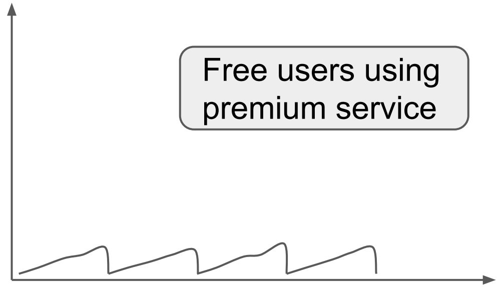


- join(event, snapshot) → always time mismatch
- Usually acceptable
  - In one direction

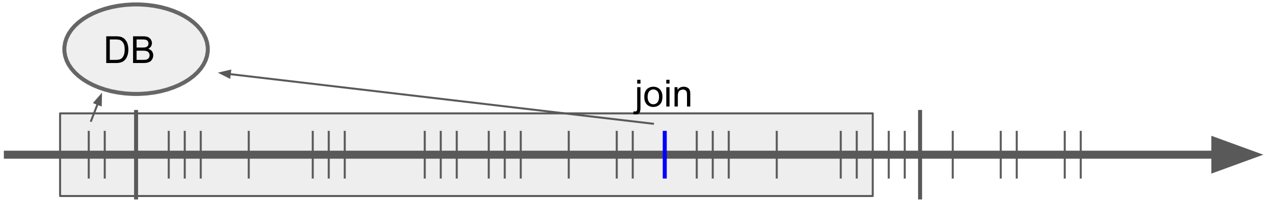
Using snapshots



- join(event, snapshot) → always time mismatch
- Usually acceptable
  - In one direction

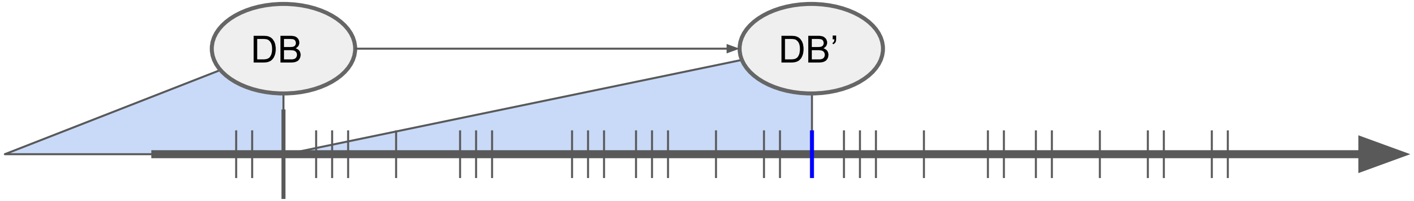


Window misalign



- Time mismatch in both directions

Event sourcing



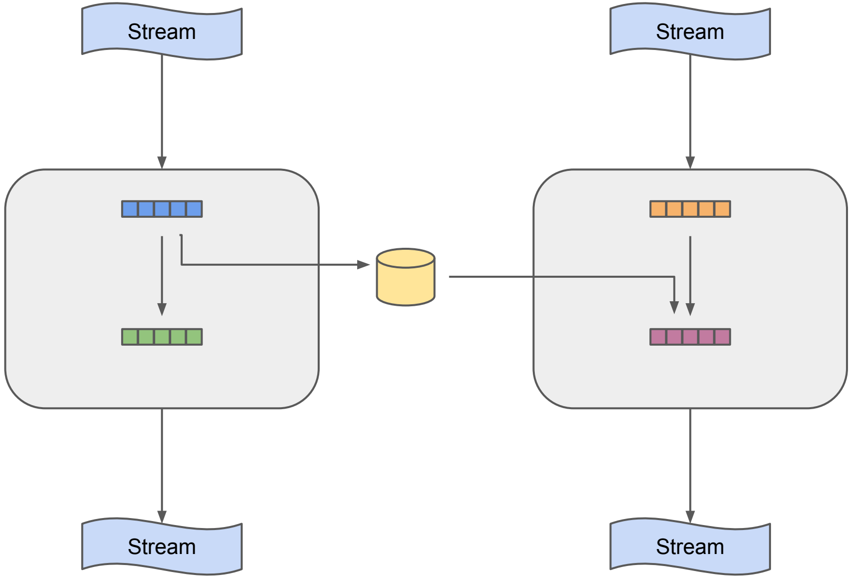
- Every change to unified log == source of truth
- snapshot(t + 1) = sum(snapshot(t), events(t, t+1))
- Allows view & join at any point in time
  - But more complex

Easier to express with streaming?

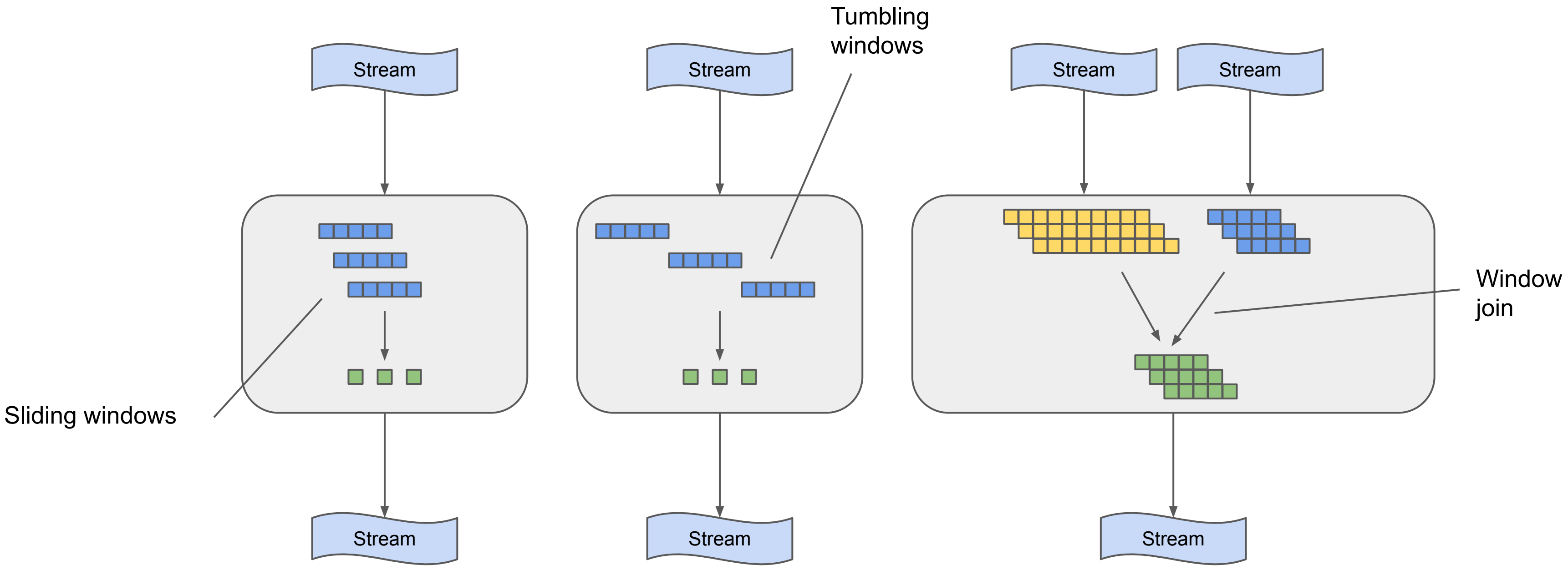
state + event → state'

- State is a view of sum of all events
    - Join with the sum table
    - Beautiful!
    - Not simple in practice

- Mixing event time and processing time
    - Every join is a race condition

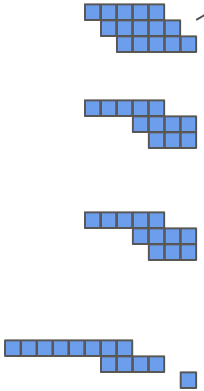


Stream window operations



Window over what?

- Number of events
- Processing time
- Registration time
- Event time



Less relevant for domain
Deterministic
Predictable semantics
Predictable resources

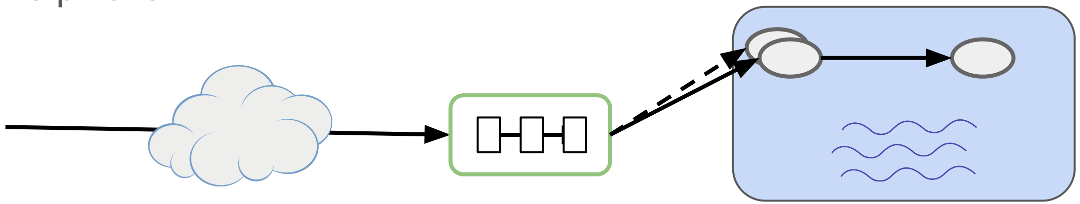
Less relevant for domain
Indeterministic
Unpredictable semantics
Predictable resources

More relevant for domain
Deterministic
Unpredictable semantics
Unpredictable resources

Event ingest

Batch is easier?

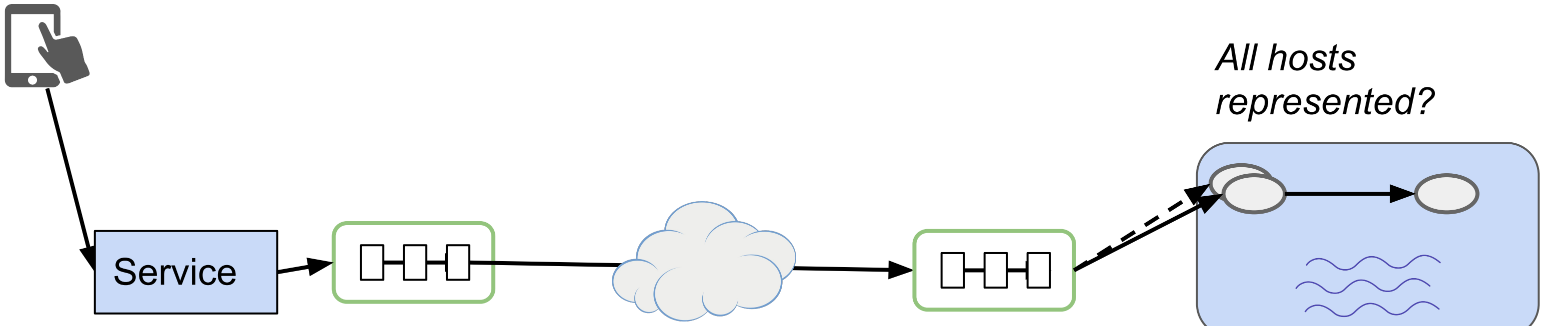
- Yes
- But, some pitfalls



Anti-pattern: Bucket by registration time

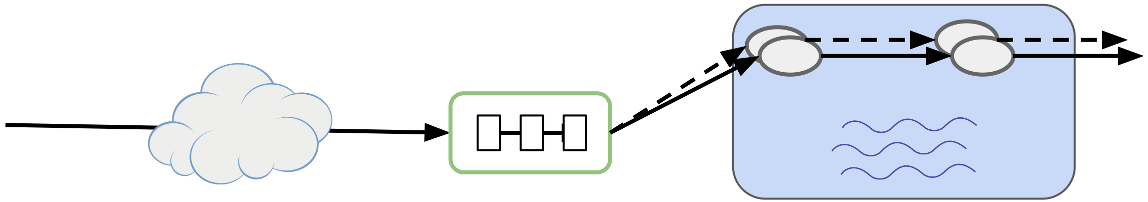
Ancient data collection:

- Events in log files, partitioned hourly
- Copy each hour of every host to lake



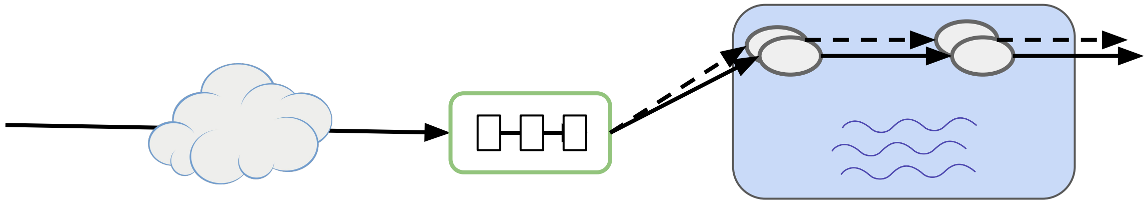
Anti-pattern: Reprocess

- Start processing optimistically
- Reprocess after x% new data has arrived



Supporting reprocessing

- Start processing optimistically
- Reprocess after x% new data has arrived

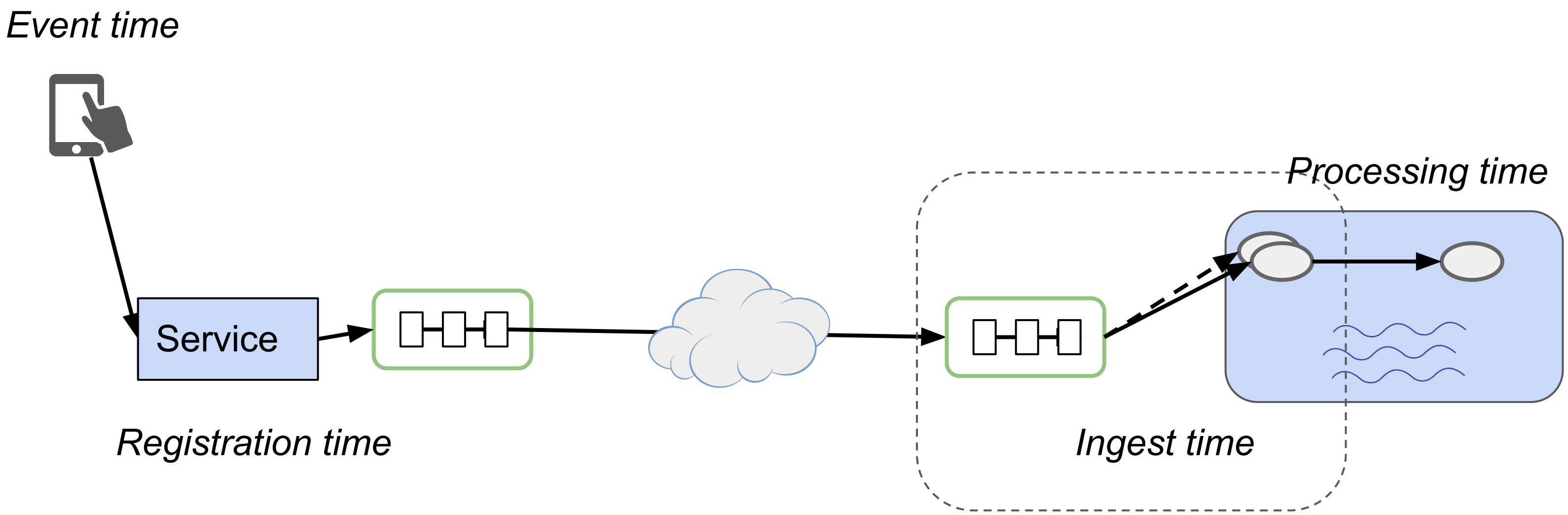


Solution space:

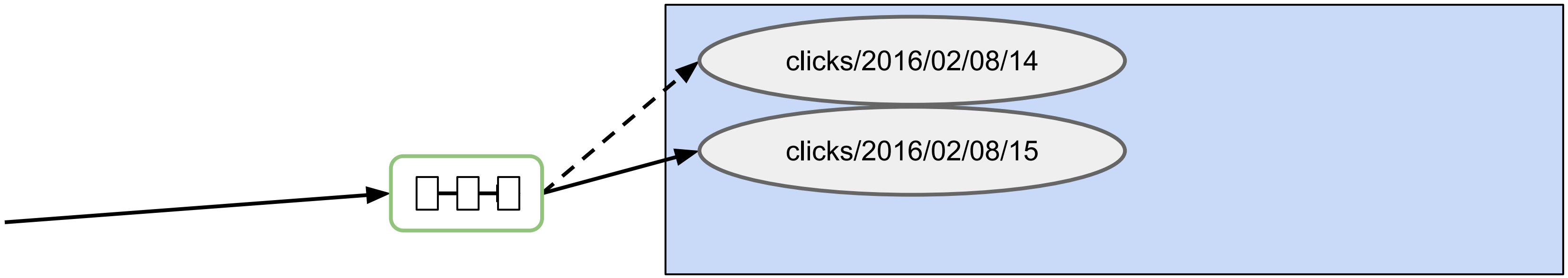
- Apache Beam / Google → stream processing ops
- Data versioning
- Provenance

Requires good (lacking) tooling

Event collection

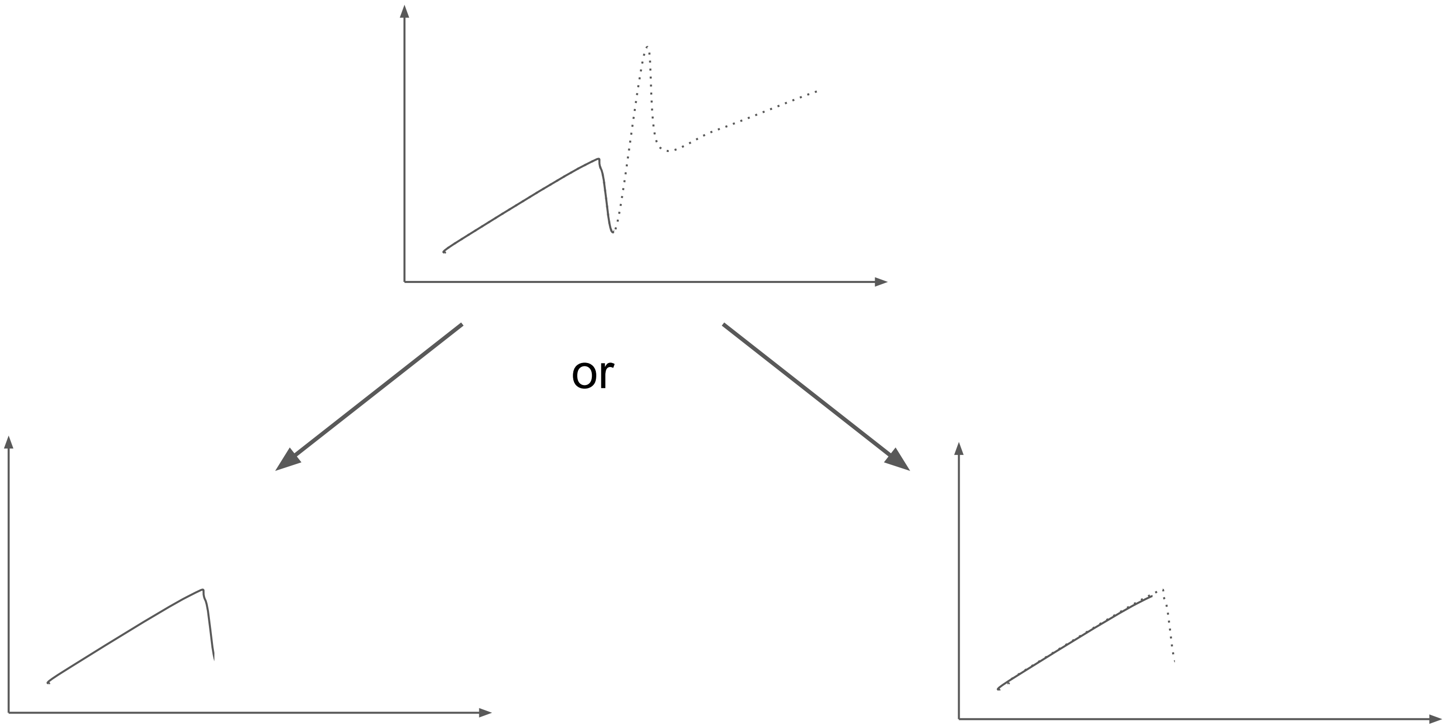


Pattern: Ingest time bucketing



- Bundle incoming events into datasets
  - Bucket on ingest / wall-clock time
  - Predictable bucketing, e.g. hour
- Sealed quickly at end of hour
- Mark dataset as complete
  - E.g. _SUCCESS flag

When data is late



Incompleteness recovery

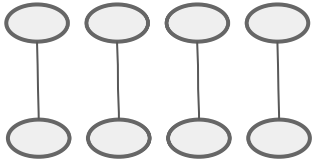
```
class OrderShuffle(SparkSubmitTask):
  hour = DateHourParameter()
  delay_hours = IntParameter()

  jar = 'orderpipeline.jar'
  entry_class = 'com.example.shop.OrderShuffleJob'

  def requires(self):
    # Note: This delays processing by N hours.
    return [Order(hour=hour) for hour in
      [self.hour + timedelta(hour=h) for h in
       range(self.delay_hours)]]

  def output(self):
    return HdfsTarget("/prod/red/order/v1/"
      f"delay={self.delay}/"
      f"{self.hour:%Y/%m/%d/%H}/")

  def app_options(self):
    return ["--hour", self.hour,
            "--delay-hours", self.delay_hours,
            "--order",
            ",".join([i.path for i in self.input()]),
            "--output", self.output().path]
```

```
val orderLateCounter = longAccumulator("order-event-late")

val hourPaths = conf.order.split(",")
val order = hourPaths
  .map(spark.read.avro(_))
  .reduce(a, b => a.union(b))

val orderThisHour = order
  .map({ cl =>
    # Count the events that came after the delay window
    if (cl.eventTime.hour + config.delayHours <
        config.hour) {
      orderLateCounter.add(1)
    }
    order
  })
  .filter(cl => cl.eventTime.hour == config.hour)
```

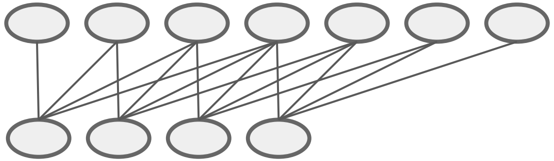
Fast data, complete data

```
class OrderShuffleAll(WrapperTask):
  hour = DateHourParameter()

  def requires(self):
    return [OrderShuffle(hour=self.hour, delay_hour=d)
            for d in [0, 4, 12]]


class OrderDashboard(mysql.CopyToTable):
  hour = DateHourParameter()

  def requires(self):
    return OrderShuffle(hour=self.hour,  delay_hour=0)


class FinancialReport(SparkSubmitTask):
  date = DateParameter()

  def requires(self):
    return [OrderShuffle(
            hour=datetime.combine(self.date, time(hours=h)),
            delay_hour=12)
            for h in range(24)]
```

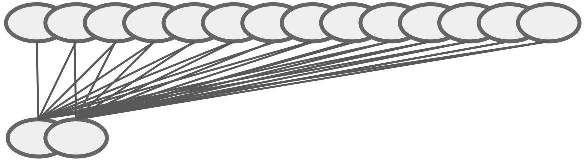
Delay: 0



Delay: 4

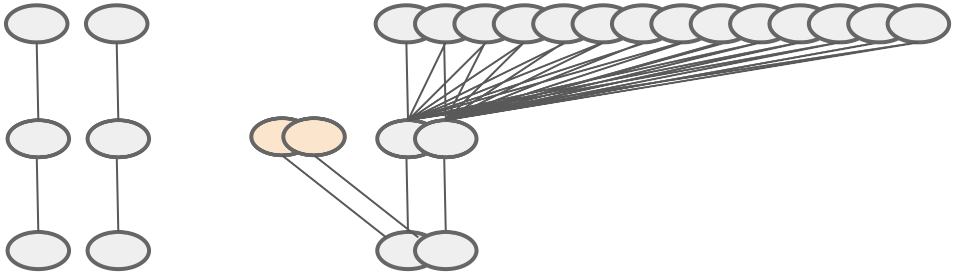


Delay: 12



Lambda in batch

```
class ReportBase(SparkSubmitTask):
  date = DateParameter()

  jar = 'reportpipeline.jar'
  entry_class = 'com.example.shop.ReportJob'

  def requires(self):
    return [OrderShuffle(
            hour=datetime.combine(self.date, time(hour=h)),
            delay_hour=self.delay)
            for h in range(24)]


class PreliminaryReport(ReportBase):
  delay = 0


class FinalReport(ReportBase):
  delay = 12

  def requires(self):
    return super().requires() + [SuspectedFraud(self.date)]
```



Human-delayed data

“These are my compensation claims for last January.”

- Want: accurate reporting
- Want: current report status

Anti-pattern: Reprocessing

Dual time scopes

- What we knew about month X at date Y
- Deterministic
  - Can be backfilled, audited
- May require sparse dependency trees

```
class ClaimsReport(SparkSubmitTask):
  domain_month = MonthParameter()
  date = DateParameter()

  jar = 'reportpipeline.jar'
  entry_class = 'com.example.shop.ClaimsReportJob'

  def requires(self):
    return [Claims(
                domain_month=self.domain_month,
                 date=d)
                for date in date_range(
                   self.domain_month + timedelta(month=1),
                   self.date)]
```

Recursive dependencies

- Use yesterday’s dataset + some delta
  - Starting point necessary
- Often convenient, but operational risk
  - Slow backfills
- Mitigation: recursive jumps
  - Depend on previous month + all previous days in this month

```
class StockAggregateJob(SparkSubmitTask):
  date = DateParameter()

  jar = 'stockpipeline.jar'
  entry_class = 'com.example.shop.StockAggregate'

  def requires(self):
    yesterday = self.date - timedelta(days=1)
    previous = StockAggregateJob( date=yesterday)
    return [StockUpdate(date=self.date), previous]
```

Recursive dependency strides

- Mitigation: recursive strides
- y/m/1 depends on y/m-1/1
- Others depend on y/m/1 + all previous days in this month

```
class StockAggregateStrideJob(SparkSubmitTask):
  date = DateParameter()

  jar = 'stockpipeline.jar'
  entry_class = 'com.example.shop.StockAggregate'

  def requires(self):
    first_in_month = self.date.replace(day=1)
    base = first_in_month - relativedelta(months=1) \
      if self.date.day == 1 else first_in_month
    return ([StockAggregateStrideJob(date=base)] +
            [StockUpdate(date=d) for d in
            rrule(freq=DAILY, dtstart=base, until=self.date)])
```

Business logic with history

Example: Forming sessions

- Valuable for
  - User insights
  - Product insights
  - A/B testing
  - ...

What is a session?

- Sequence of clicks at most 5 minutes apart?

- In order to emit sessions in one hour, which hours of clicks are needed?

What is a session?

- Sequence of clicks at most 5 minutes apart?
- Maximum length 3 hours?

- In order to emit sessions in one hour, which hours of clicks are needed?

What is a session?

- Sequence of clicks at most 5 minutes apart.
- Maximum length 3 hours.

Examples, window = 5am - 9am:

- Clicks at [6:00, 6:01, 6:03, 6:45, 6:47]?
  - Two sessions, 3 and 2 minutes long.

What is a session?

- Sequence of clicks at most 5 minutes apart.
- Maximum length 3 hours.

Examples, window = 5am - 9am:

- Clicks at [6:00, 6:01, 6:03, 6:45, 6:47]?
  - Two sessions, 3 and 2 minutes long.
- [5:00, 5:01, 5:03]?
  - One session, 3 minutes?

What is a session?

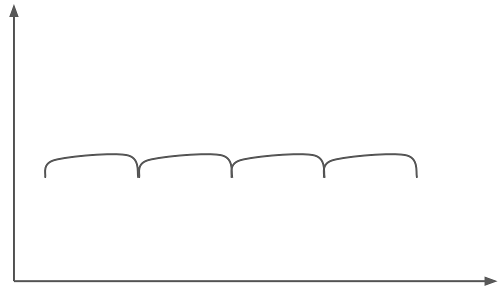
- Sequence of clicks at most 5 minutes apart.
- Maximum length 3 hours.

Examples, window = 5am - 9am:

- Clicks at [6:00, 6:01, 6:03, 6:45, 6:47]?
    - Two sessions, 3 and 2 minutes long.
- [5:00, 5:01, 5:03]?
    - One session, 3 minutes?
    - [(4:57), 5:00, 5:01, 5:03]?
    - [(2:01), (every 3 minutes), (4:57), 5:00, 5:01, 5:03]?

Occurrences with unbounded time spans

- E.g. sessions, behavioural patterns
- You often need a broader time range than expected
- You may need data from the future
- You may need infinite history
  - Recursive strides?
  - Introduce static limits, e.g. cut all sessions at midnight?
  - Emit counters to monitor assumptions.



Secrets of valuable data engineering

1. Avoid complexity and distributed systems
    - Your data fits in one machine.

2. Pick the slowest data integration you can live with
    - Batch >> streaming >> service.
    - Slow data → easy operations → high innovation speed

3. Functional architectural principles
    - Pipelines
    - Immutability
    - Reproducibility
    - Idempotency

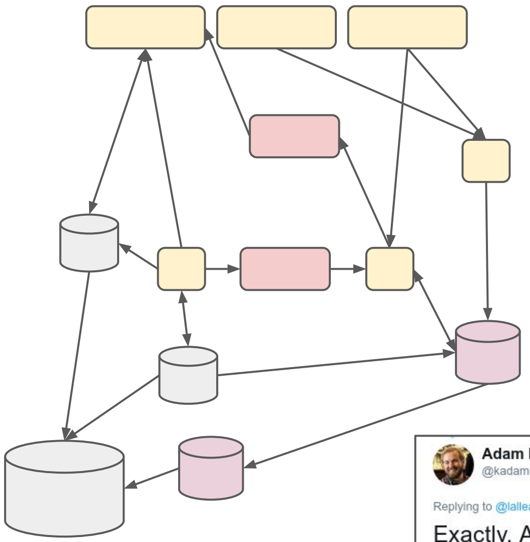
4. Master workflow orchestration

Team concurrency

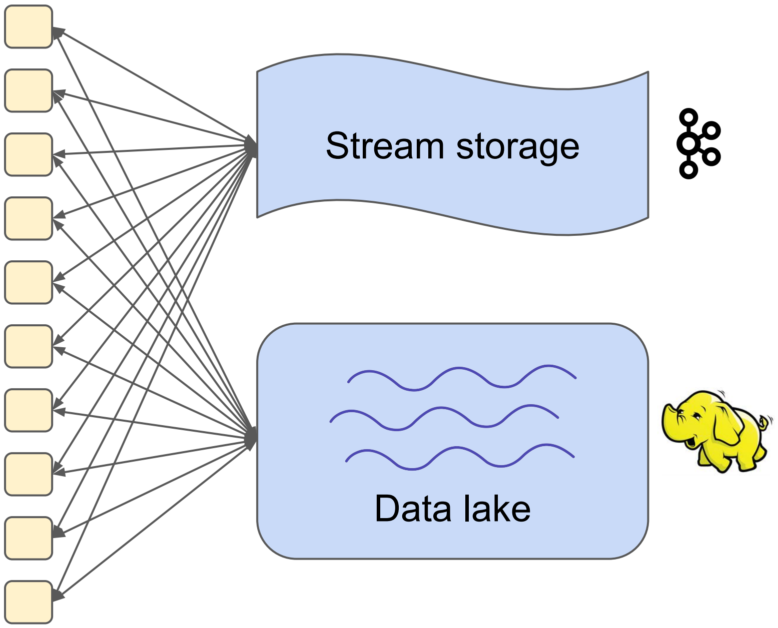
- Pipelines
  - Parallel development without risk
- Immutability
  - Data reusability without risk
- Reproducibility, idempotency
  - Preserving immutability
  - Reduces operational risk
  - Stable experiments
- Workflow orchestration
  - Data handover between systems & teams
  - Reduces operational overhead

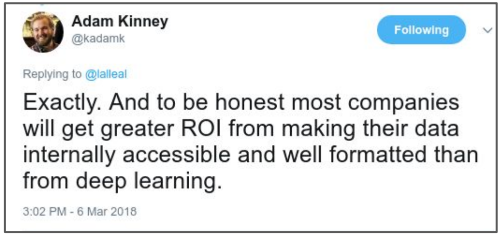
Data science reproducibility crisis!

The real value of big data



Data democratised




Adam Kinney
@kadamk
Following

Replying to @lalleal

Exactly. And to be honest most companies will get greater ROI from making their data internally accessible and well formatted than from deep learning.

3:02 PM - 6 Mar 2018

Resources, credits

Time libraries:

- Java: ~~Joda time~~ java.time
- Scala: chronoscala
- Python: dateutil, pendulum

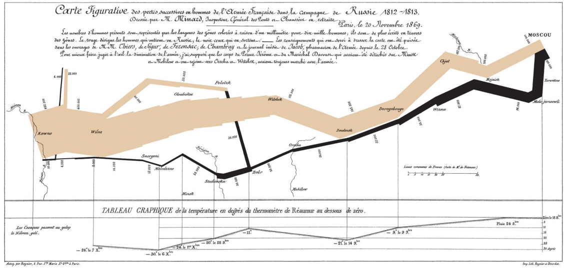
Presentation on operational tradeoffs:

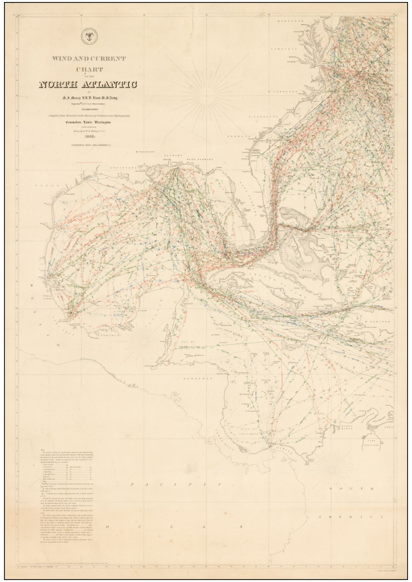
https://www.slideshare.net/lallea/data-ops-in-practice

Thank you,

- Konstantinos Chaidos, Spotify
- Lena Sundin, independent

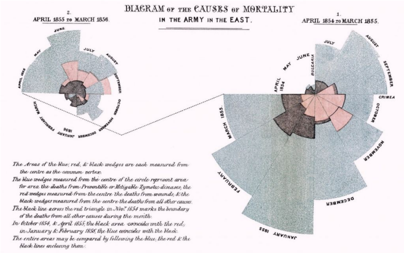
Laptop sticker



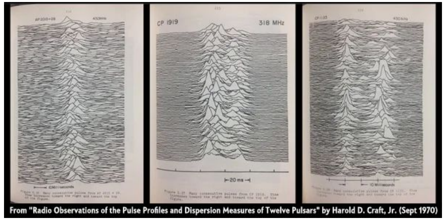


Vintage data visualisations, by Karin Lind.

- Charles Minard: Napoleon’s Russian campaign of 1812. Drawn 1869.
- Matthew F Maury: Wind and Current Chart of the North Atlantic. Drawn 1852.
- Florence Nightingale: Causes of Mortality in the Army of the East. Crimean war, 1854-1856. Drawn 1858.
  - Blue = disease, red = wounds, black = battle + other.



- Harold Craft: Radio Observations of the Pulse Profiles and Dispersion Measures of Twelve Pulsars, 1970
  - Joy Division:
    Unknown Pleasures, 1979
  - ~~“Joy plot”~~ → “ridge plot”



From “Radio Observations of the Pulse Profiles and Dispersion Measures of Twelve Pulsars” by Harold D. Craft, Jr. (Sept 1970)