MANAGING AND PROCESSING LARGE DATASETS

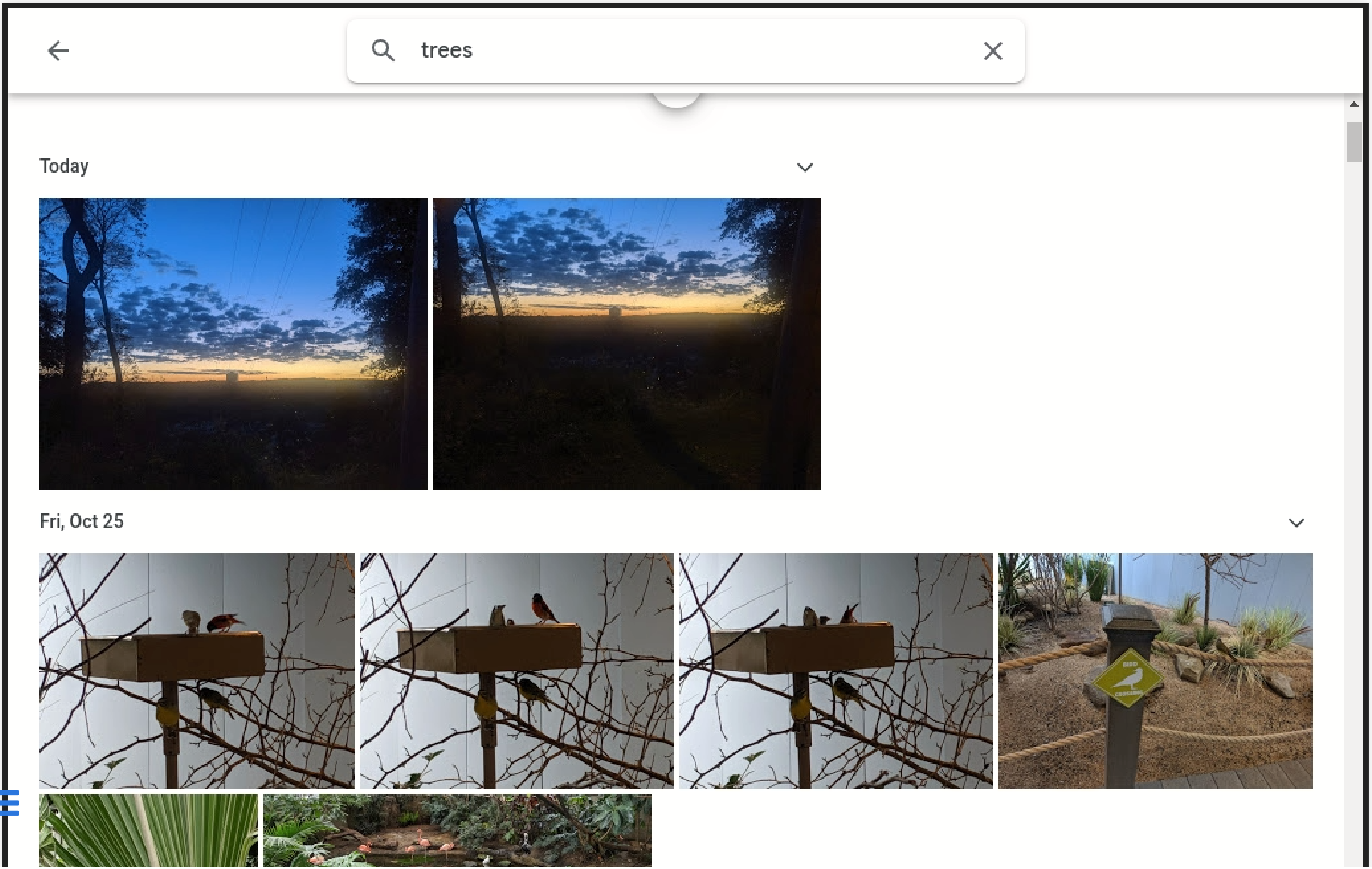
Christian Kaestner

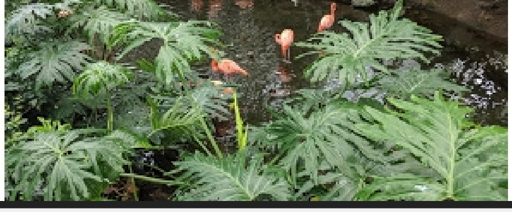
Required reading: Martin Kleppmann. Designing Data-Intensive Applications. OReilly. 2017. Chapter 1

LEARNING GOALS

- Organize different data management solutions and their tradeoffs
- Explain the tradeoffs between batch processing and stream processing and the lambda architecture
- Recommend and justify a design and corresponding technologies for a given system

CASE STUDY





Speaker notes

- Discuss possible architecture and when to predict (and update)
- in may 2017: 500M users, uploading 1.2billion photos per day (14k/sec)
- in Jun 2019 1 billion users

DATA MANAGEMENT AND PROCESSING IN ML-ENABLED SYSTEMS

KINDS OF DATA

- Training data
- Input data
- Telemetry data
- (Models)

all potentially with huge total volumes and high throughput

need strategies for storage and processing

DATA MANAGEMENT AND PROCESSING IN ML-ENABLED SYSTEMS

- Store, clean, and update training data
- Learning process reads training data, writes model
- Prediction task (inference) on demand or precomputed
- Individual requests (low/high volume) or large datasets?

- Often both learning and inference data heavy, high volume tasks

DISTRIBUTED X

- Distributed data cleaning
- Distributed feature extraction
- Distributed learning
- Distributed large prediction tasks
- Incremental predictions
- Distributed logging and telemetry

DATA STORAGE BASICS

- Relational vs document storage
- 1:n and n:m relations
- Storage and retrieval, indexes
- Query languages and optimization

RELATIONAL DATA MODELS

user_id	Name	Email	dpt
1	Christian	kaestner@cs.	1
2	Eunsuk	eskang@cmu.	1
2	Tom	...	2

dpt_id	Name	Address
1	ISR	...
2	CSD	...

```
select d.name from user u, dpt d where u.dpt=d.dpt_id
```

DOCUMENT DATA MODELS

```
{
    "id": 1,
    "name": "Christian",
    "email": "kaestner@cs.",
    "dpt": [
        {"name": "ISR", "address": "..."}
    ],
    "other": { ... }
}
```

```
db.getCollection('users').find({"name": "Christian"})
```

LOG FILES, UNSTRUCTURED DATA

```
2020-06-25T13:44:14,601844,GET /data/m/goyas+ghosts+2006/17.mpg
2020-06-25T13:44:14,935791,GET /data/m/the+big+circus+1959/68.mp
2020-06-25T13:44:14,557605,GET /data/m/elvis+meets+nixon+1997/17
2020-06-25T13:44:14,140291,GET /data/m/the+house+of+the+spirits+
2020-06-25T13:44:14,425781,GET /data/m/the+theory+of+everything+
2020-06-25T13:44:14,773178,GET /data/m/toy+story+2+1999/59.mpg
2020-06-25T13:44:14,901758,GET /data/m/ignition+2002/14.mpg
2020-06-25T13:44:14,911008,GET /data/m/toy+story+3+2010/46.mpg
```

TRADEOFFS

DATA ENCODING

- Semi-structured, schema-free (JSON, XML)
- Schema-based encoding (CSV, binary)
- Compact encodings

DISTRIBUTED DATA STORAGE

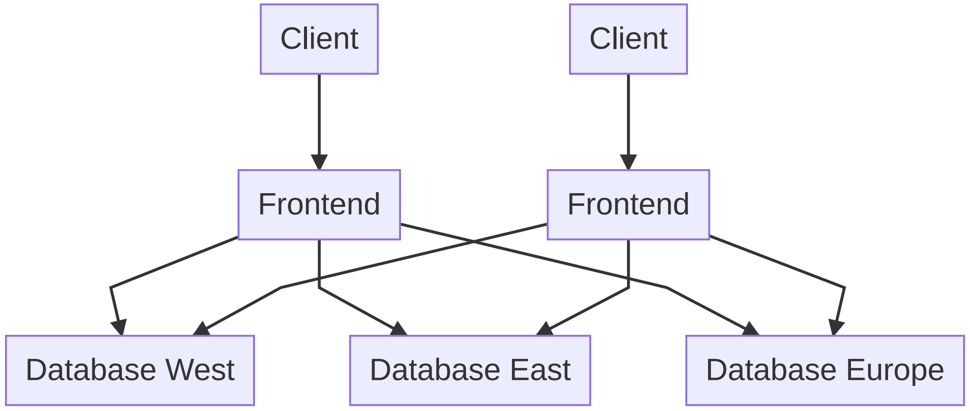
REPLICATION VS PARTITIONING

PARTITIONING

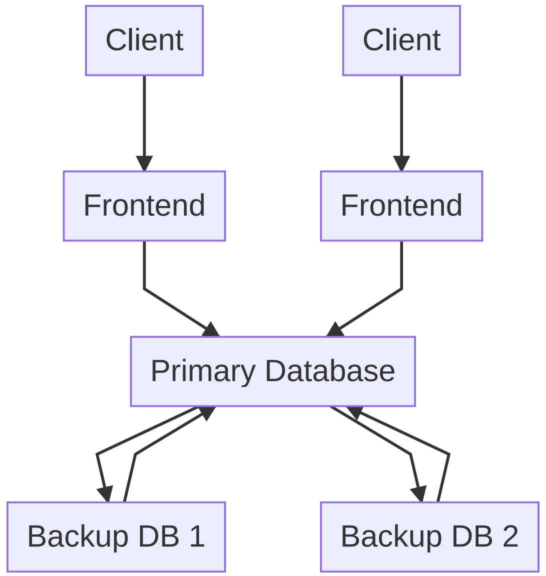
Divide data:

- Horizontal partitioning: Different rows in different tables; e.g., movies by decade, hashing often used
- Vertical partitioning: Different columns in different tables; e.g., movie title vs. all actors

Tradeoffs?



REPLICATION STRATEGIES: LEADERS AND FOLLOWERS



REPLICATION STRATEGIES: LEADERS AND FOLLOWERS

- Write to leader
    - propagated synchronously or async.
- Read from any follower
- Elect new leader on leader outage; catchup on follower outage

- Built in model of many databases (MySQL, MongoDB, ...)

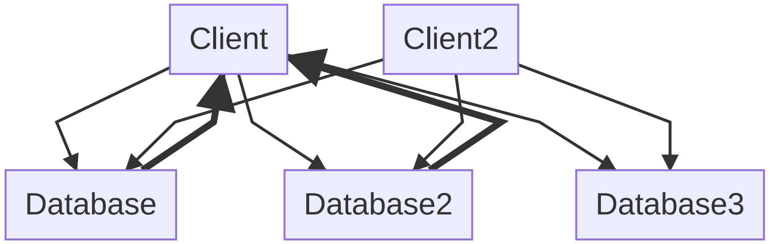
Benefits and Drawbacks?

MULTI-LEADER REPLICATION

- Scale write access, add redundancy
- Requires coordination among leaders
    - Resolution of write conflicts
- Offline leaders (e.g. apps), collaborative editing

LEADERLESS REPLICATION

- Client writes to all replica
- Read from multiple replica (quorum required)
    - Repair on reads, background repair process
- Versioning of entries (clock problem)
- e.g. Amazon Dynamo, Cassandra, Voldemort



TRANSACTIONS

- Multiple operations conducted as one, all or nothing
- Avoids problems such as
    - dirty reads
    - dirty writes
- Various strategies, including locking and optimistic+rollback
- Overhead in distributed setting

DATA PROCESSING (OVERVIEW)

- Services (online)
    - Responding to client requests as they come in
    - Evaluate: Response time
- Batch processing (offline)
    - Computations run on large amounts of data
    - Takes minutes to days
    - Typically scheduled periodically
    - Evaluate: Throughput
- Stream processing (near real time)
    - Processes input events, not responding to requests
    - Shortly after events are issued

BATCH PROCESSING

LARGE JOBS

- Analyzing TB of data, typically distributed storage
- Filtering, sorting, aggregating
- Producing reports, models, ...

```
cat /var/log/nginx/access.log |
    awk '{print $7}' |
    sort |
    uniq -c |
    sort -r -n |
    head -n 5
```

DISTRIBUTED BATCH PROCESSING

- Process data locally at storage
- Aggregate results as needed
- Separate pluming from job logic

MapReduce as common framework

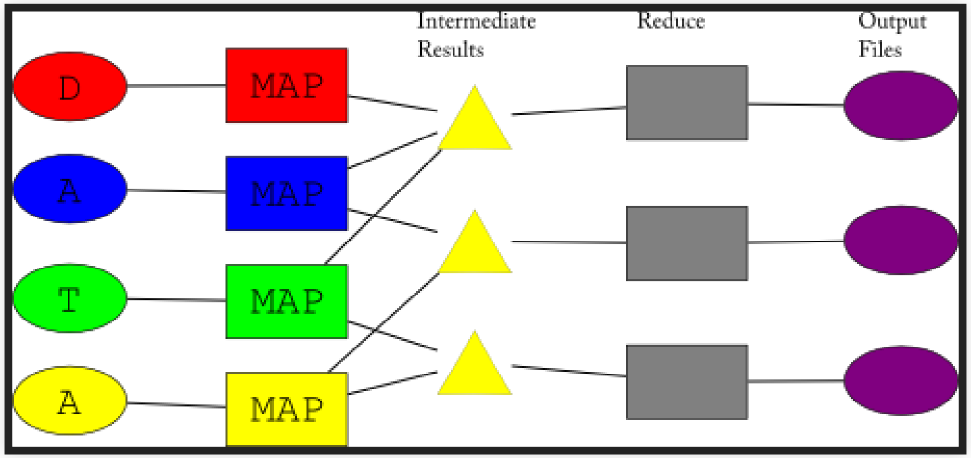


Image Source: Ville Tuulos (CC BY-SA 3.0)

MAPREDUCE -- FUNCTIONAL PROGRAMMING STYLE

- Similar to shell commands: Immutable inputs, new outputs, avoid side effects
- Jobs can be repeated (e.g., on crashes)
- Easy rollback
- Multiple jobs in parallel (e.g., experimentation)

MACHINE LEARNING AND MAPREDUCE

Speaker notes

Useful for big learning jobs, but also for feature extraction

DATAFLOW ENGINES (SPARK, TEZ, FLINK, ...)

- Single job, rather than subjobs
- More flexible than just map and reduce
- Multiple stages with explicit dataflow between them
- Often in-memory data
- Pluming and distribution logic separated

STREAM PROCESSING

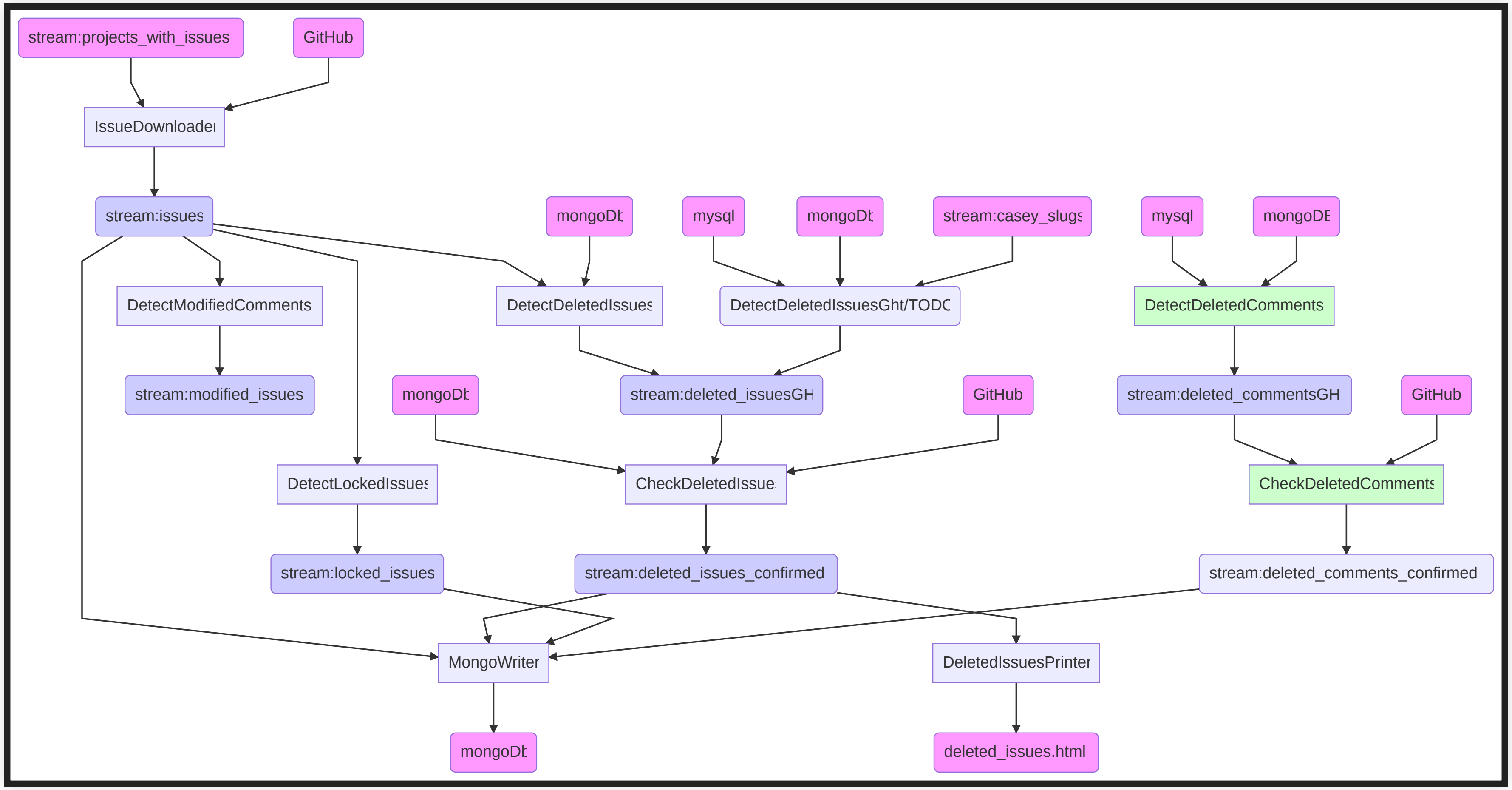
- Event-based systems, message passing style, publish subscribe

MESSAGING SYSTEMS

- Multiple producers send messages to topic
- Multiple consumers can read messages
- Decoupling of producers and consumers
- Message buffering if producers faster than consumers
- Typically some persistency to recover from failures
- Messages removed after consumption or after timeout
- With or without central broker
- Various error handling strategies (acknowledgements, redelivery, ...)

COMMON DESIGNS

Like shell programs: Read from stream, produce output in other stream. Loose coupling



9.3

STREAM QUERIES

- Processing one event at a time independently
- vs incremental analysis over all messages up to that point
- vs floating window analysis across recent messages
- Works well with probabilistic analyses

CONSUMERS

- Multiple consumers share topic for scaling and load balancing
- Multiple consumers read same message for different work
- Partitioning possible

DESIGN QUESTIONS

- Message loss important? (at-least-once processing)
- Can messages be processed repeatedly (at-most-once processing)
- Is the message order important?
- Are messages still needed after they are consumed?

STREAM PROCESSING AND AI-ENABLED SYSTEMS?

Speaker notes

Process data as it arrives, prepare data for learning tasks, use models to annotate data, analytics

EVENT SOURCING

- Append only databases
- Record edit events, never mutate data
- Compute current state from all past events, can reconstruct old state
- For efficiency, take state snapshots
- Similar to traditional database logs

```
createUser(id=5, name="Christian", dpt="SCS")
updateUser(id=5, dpt="ISR")
deleteUser(id=5)
```

BENEFITS OF IMMUTABILITY (EVENT SOURCING)

- All history is stored, recoverable
- Versioning easy by storing id of latest record
- Can compute multiple views
- Compare *git*

On a shopping website, a customer may add an item to their cart and then remove it again. Although the second event cancels out the first event from the point of view of order fulfillment, it may be useful to know for analytics purposes that the customer was considering a particular item but then decided against it. Perhaps they will choose to buy it in the future, or perhaps they found a substitute. This information is recorded in an event log, but would be lost in a database that deletes items when they are removed from the cart.

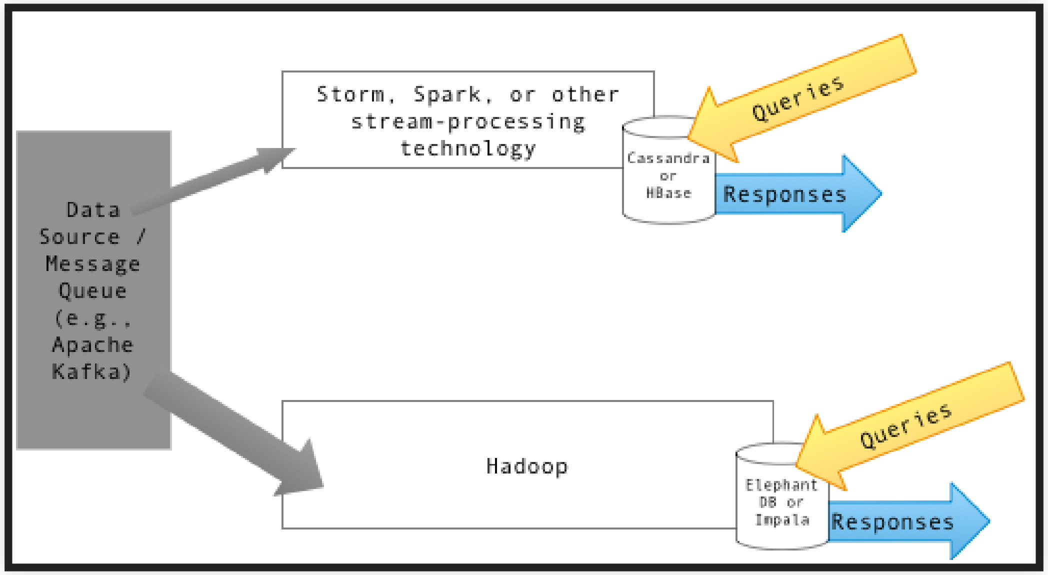
Source: Greg Young. CQRS and Event Sourcing. Code on the Beach 2014 via Martin Kleppmann. Designing Data-Intensive Applications. OReilly. 2017.

DRAWBACKS OF IMMUTABLE DATA

Speaker notes

- Storage overhead, extra complexity of deriving state
- Frequent changes may create massive data overhead
- Some sensitive data may need to be deleted (e.g., privacy, security)

THE LAMBDA ARCHITECTURE

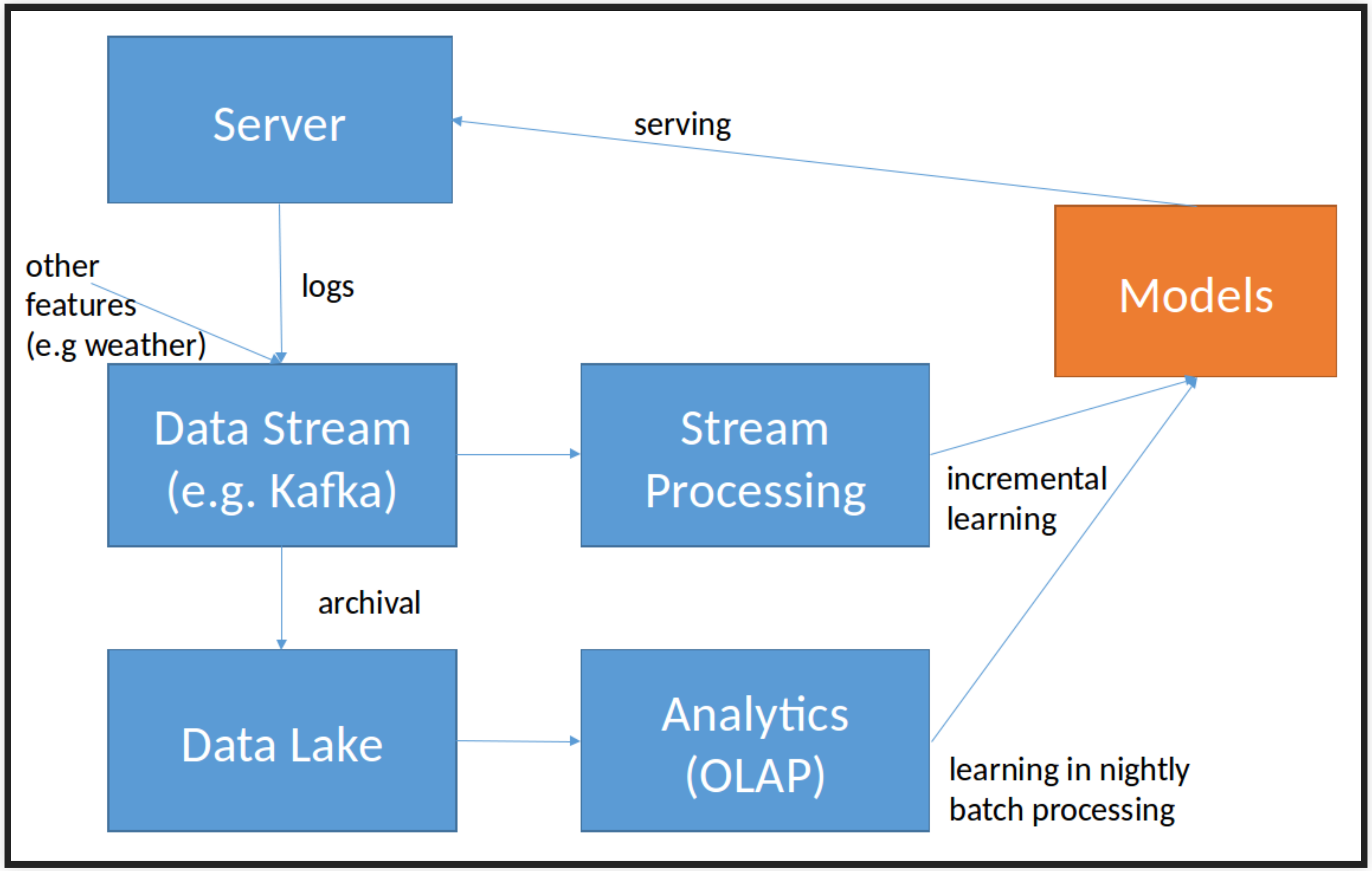

Storm, Spark, or other stream-processing technology
Queries
Cassandra or HBase
Responses
Data Source / Message Queue (e.g., Apache Kafka)
Hadoop
Queries
Elephant DB or Impala
Responses

LAMBDA ARCHITECTURE: 3 LAYER STORAGE ARCHITECTURE

- Batch layer: best accuracy, all data, recompute periodically
- Speed layer: stream processing, incremental updates, possibly approximated
- Serving layer: provide results of batch and speed layers to clients

- Assumes append-only data
- Supports tasks with widely varying latency
- Balance latency, throughput and fault tolerance

LAMBDA ARCHITECTURE AND MACHINE LEARNING



- Learn accurate model in batch job
- Learn incremental model in stream processor

DATA LAKE

- Trend to store all events in raw form (no consistent schema)
- May be useful later
- Data storage is comparably cheap

REASONING ABOUT DATAFLOWS

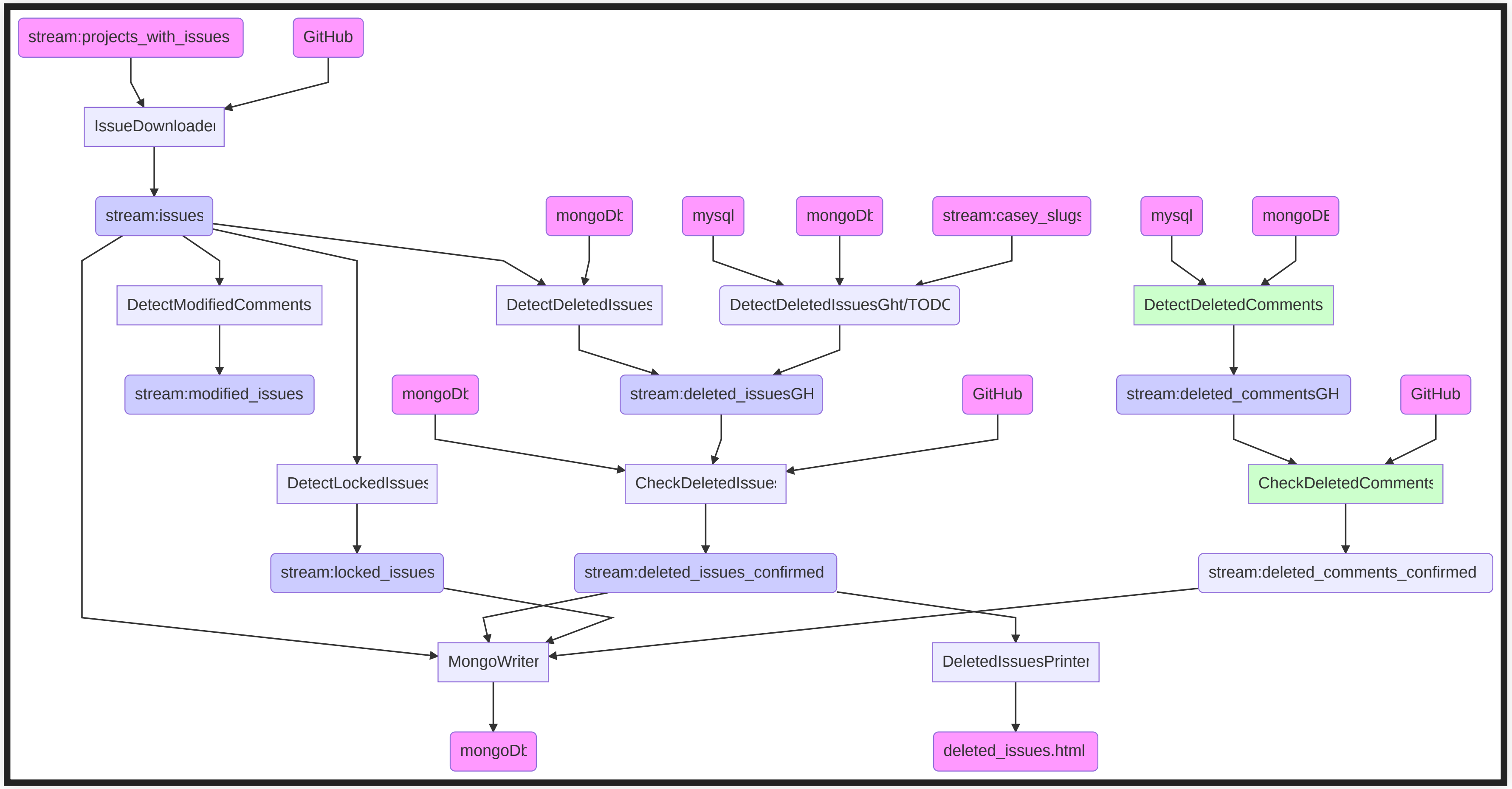
Many data sources, many outputs, many copies

Which data is derived from what other data and how?

Is it reproducible? Are old versions archived?

How do you get the right data to the right place in the right format?

Plan and document data flows


stream:projects_with_issues
GitHub
IssueDownloader
stream:issues
mongoDb
mysql
mongoDb
stream:casey_slugs
mysql
mongoDB
DetectModifiedComments
DetectDeletedIssues
DetectDeletedIssuesGht/TODO
DetectDeletedComments
stream:modified_issues
mongoDb
stream:deleted_issuesGH
GitHub
stream:deleted_commentsGH
GitHub
DetectLockedIssues
CheckDeletedIssues
CheckDeletedComments
stream:locked_issues
stream:deleted_issues_confirmed
stream:deleted_comments_confirmed
MongoWriter
DeletedIssuesPrinter
mongoDb
deleted_issues.html

SUMMARY

- Data storage strategies and their tradeoffs
- Partitioning and replication for scalability
- Design considerations: mutable vs immutable data
- Common design patterns (e.g., batch processing, stream processing, lambda architecture)

Recommended reading: Martin Kleppmann. Designing Data-Intensive Applications. OReilly. 2017.