Evolution:
From Distributed Cache to Feature Store

Yao Yue, IOP Systems

About me

- 10+ years of working on cache, creator of Pelikan
- 5+ years of performance engineering

- IOP Systems, ex-Twitter (not to be confused with X/Twitter)

- Systems thinking enthusiast

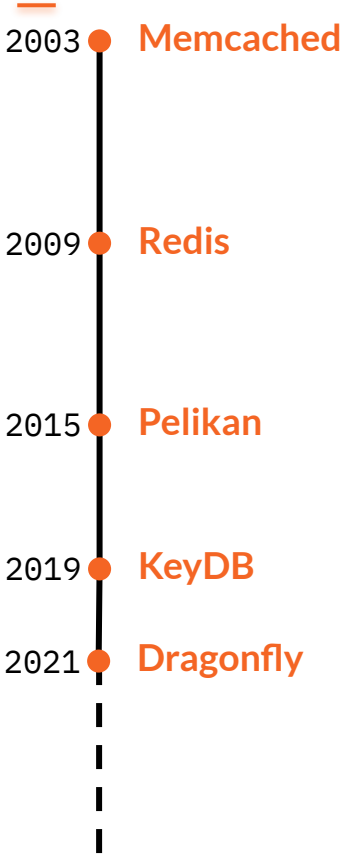


Fig. 1: Cache engine chronicles, most likely partial

Distributed Caching

- The emergence of distributed cache
  - Why
  - What
- Production adoption — distributed system grinding
  - Sharding and scaling
  - Availability
  - Performance
- Functionality
  - Data types and APIs

Fig. 2: Data processing chronicle, most definitely partial

Big Data Processing

- First there was data, a lot of data
- Transform data with a distributed system
- An increasingly-online world requires streaming
- Unification of the batch / streaming processing
  - Compute semantics unification: Lambda architecture
  - Access semantics unification: Kappa architecture

Feature store ecosystem

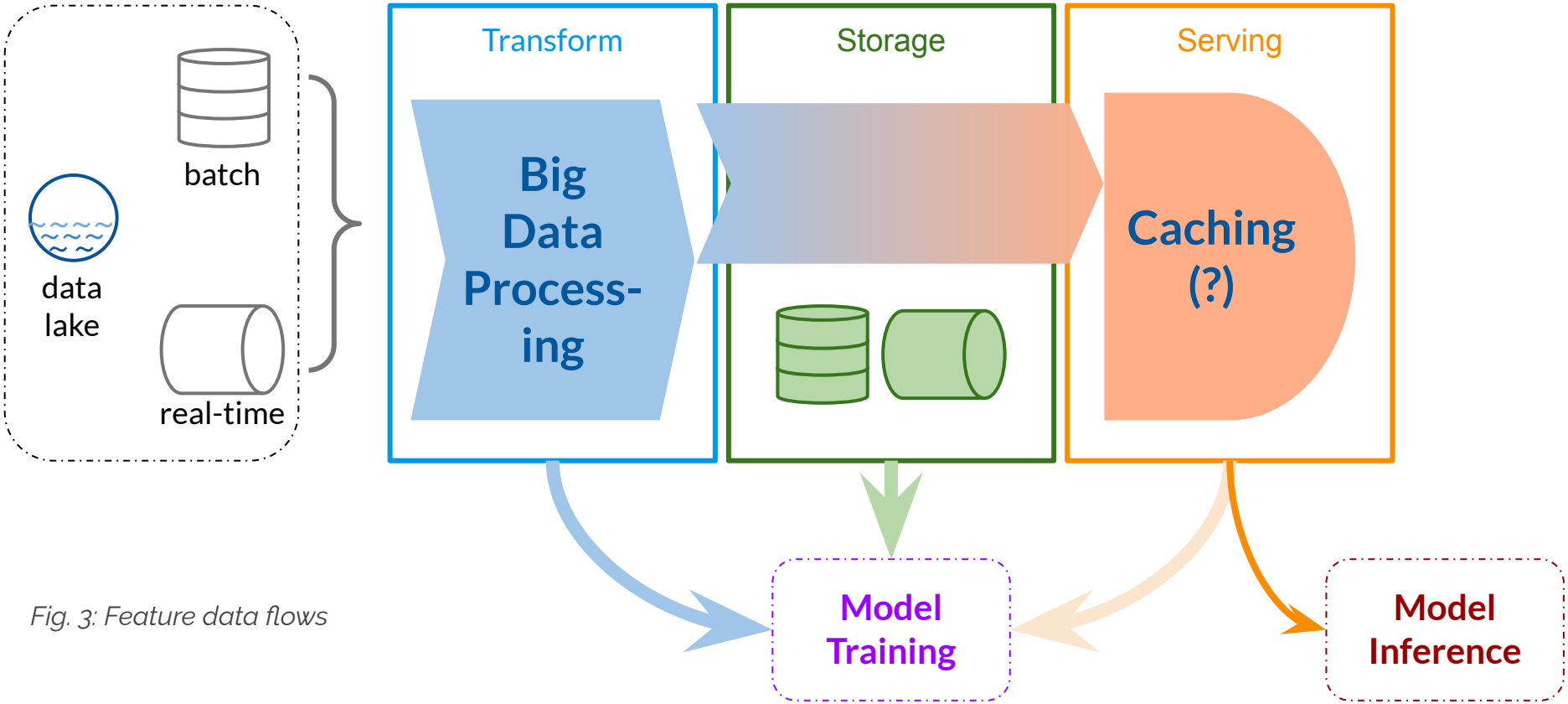
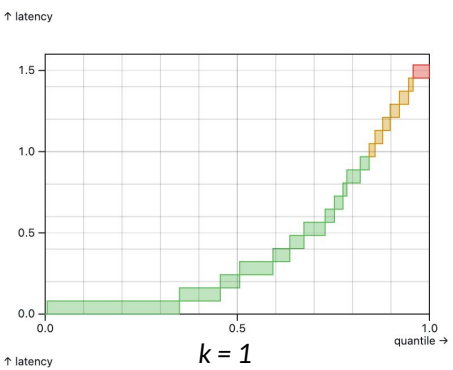
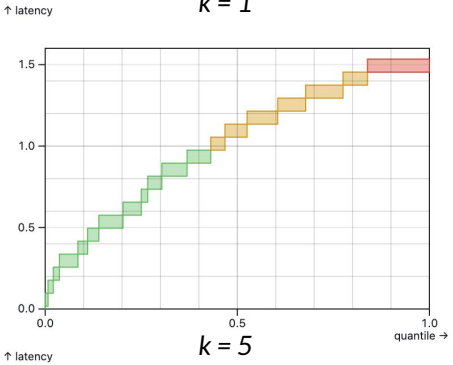


Fig. 3: Feature data flows

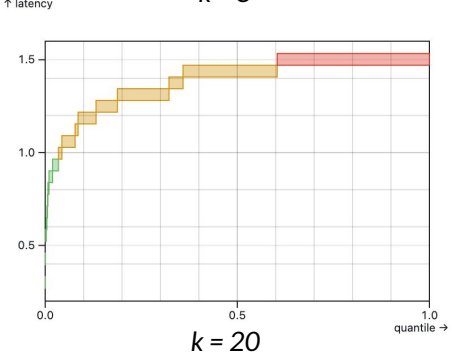
Bite-size big-data serving



k = 1



k = 5



k = 20

Identity-based feature retrieval

- A lot of data in a key-value format
- *Not* a canonical source
- Scatter-gather retrieval
  - Tail at scale: elevation of tail latencies

⇒ Distributed caching

- Data Layout
- Indexing
- API interface
- Threading Architecture

Divergent read-write patterns

- **Write**
  - Batch / Streaming writes, from data processing jobs
  - vs *point update* in transactional workloads
  - High volume, sequential, delay-tolerant
- Read
  - Point reads
  - Low volume, random, latency-sensitive
  - (Streaming reads in the future?)

- Entry Encoding
- Native operators
- Programmability

Feature-specific Considerations

- Limited basic types
  - Floating-point, fixed-point, integers
- Organizations
  - Maps/Records
  - Sorted lists/maps
- Queries
  - `get(Multi)Field`
  - `range`
  - `op() (max, min, avg, …)`

A few predictions on feature store evolution

- On storage
  - Embrace SSDs and write-friendly data layout, mainly for cost
  - Yes to durability, no to transactions
- On architecture
  - Local-level compute-storage separation
  - Message-passing centric internal communication
- Data import APIs
- High-level grammar for queries and functions

Questions?