Sparkle

Visualize the Things Scala Days SF 2015

@mighdoll   lee@nestlabs.com

About Me

Nest, Google, Typesafe. (opinions are my own)

Apple, General Magic, Twitter, WebTV, Microsoft, a few other startups

Scala fan

Today

1. Intro to sparkle
2. Performance & scaling with streams
3. Live layer and data architecture

Bonus: demos, development tips, Q&A

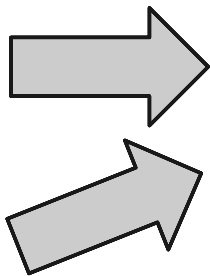
Sparkle

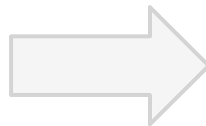
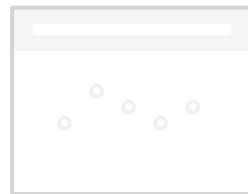
Tool for easily making zooming graphs

Platform for custom visualizations on live data

- Built on streams
- Big data, low latency

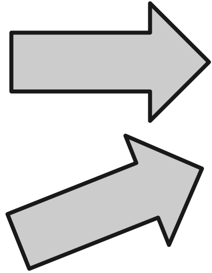
https://github.com/mighdoll/sparkle



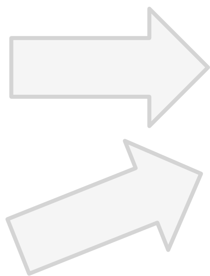
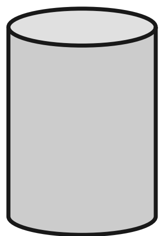
Loaders Fast Storage API Transform Stream Display

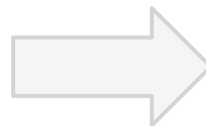
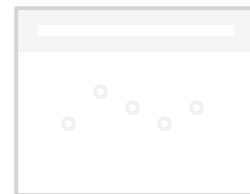
Loading Data



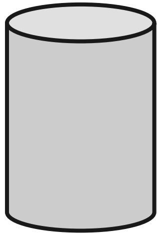
Loaders

- Several inputs, easy to add more
  - Files and directories (.csv / .tsv)
  - Kafka / Avro
  - HDFS bulk (*)
  - netcat (*)
- Loaders support subscribe/notify


Loaders

Fast Storage

API
Transform

Stream

Display

Sparkle data model

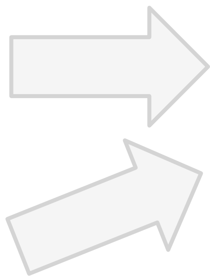


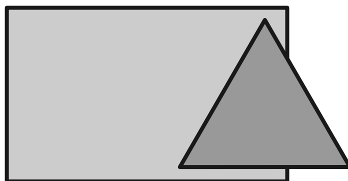
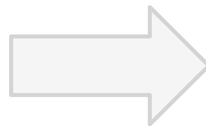
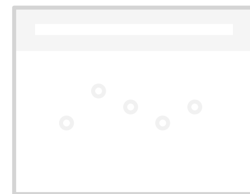
Fast Storage

Column oriented store

Immutable data model

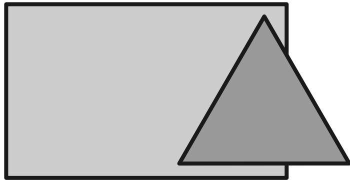
- Cassandra
- RamStore



Loaders Fast Storage **API Transform** Stream Display

Select and Transform



Select & Transform

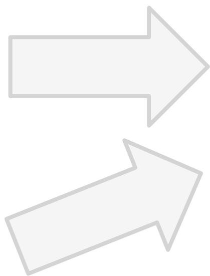
Select columns

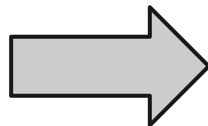
Apply standard transforms

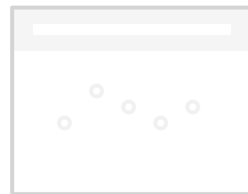
- Aggregation
- Time Aggregation

Extend with custom transforms

Fast

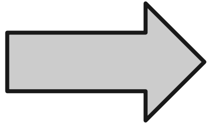






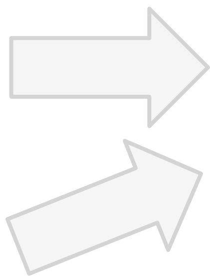
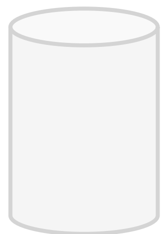
Loaders Fast Storage API Transform **Stream** Display

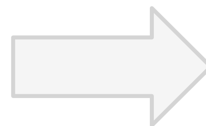
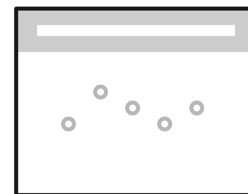
Sparkle Protocol



Stream

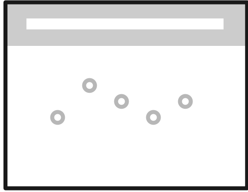
- Generic visualization protocol
- Designed for (Web)Sockets
- HTTP access too (no streaming)
- json encoded
- Documented

Loaders Fast Storage API Transform Stream **Display**

Javascript Client



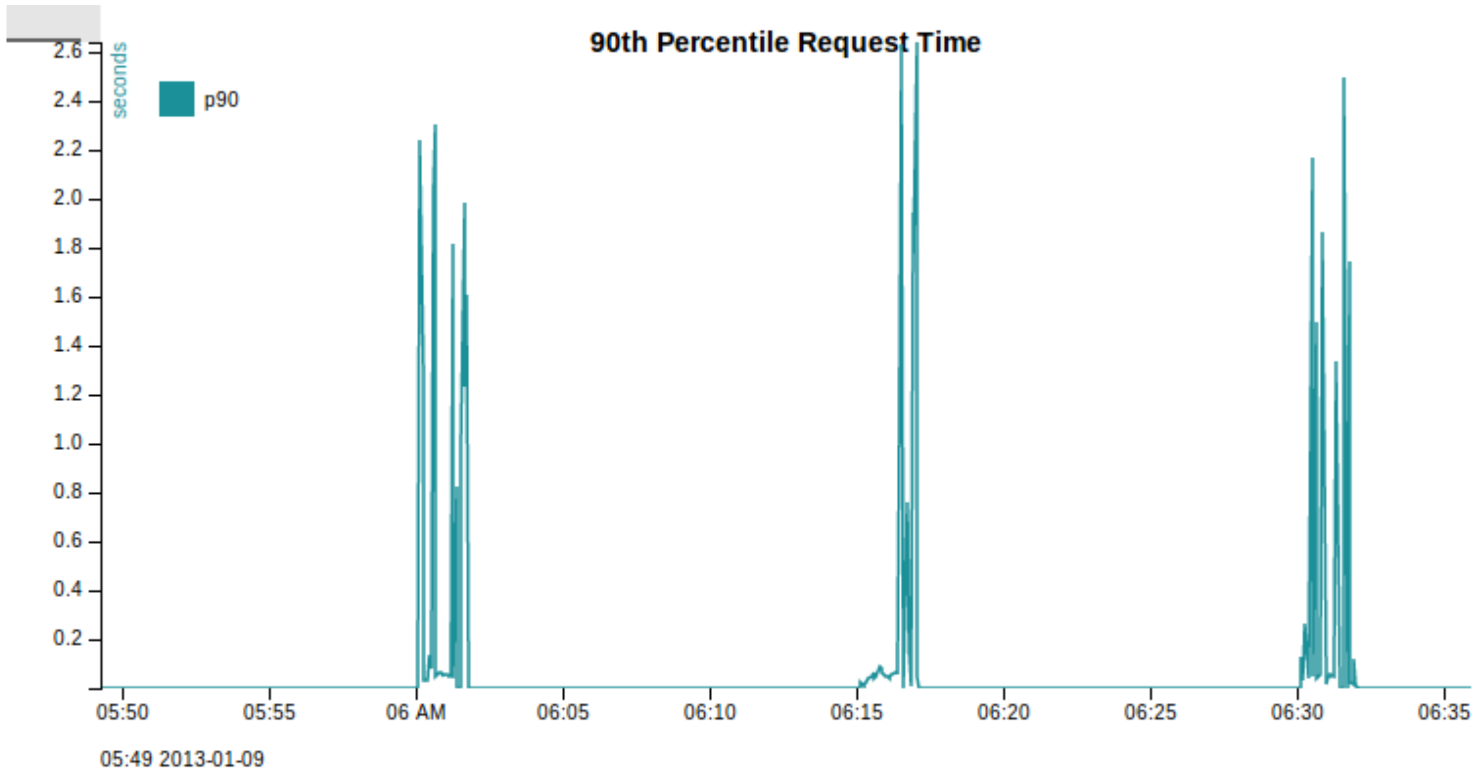
Display

- built on d3
- start with declarative javascript
- easy to customize / extend
- uses server api
  - (or standalone)

Simple Demo

quick graph of .tsv plot from command line

Simple Demo



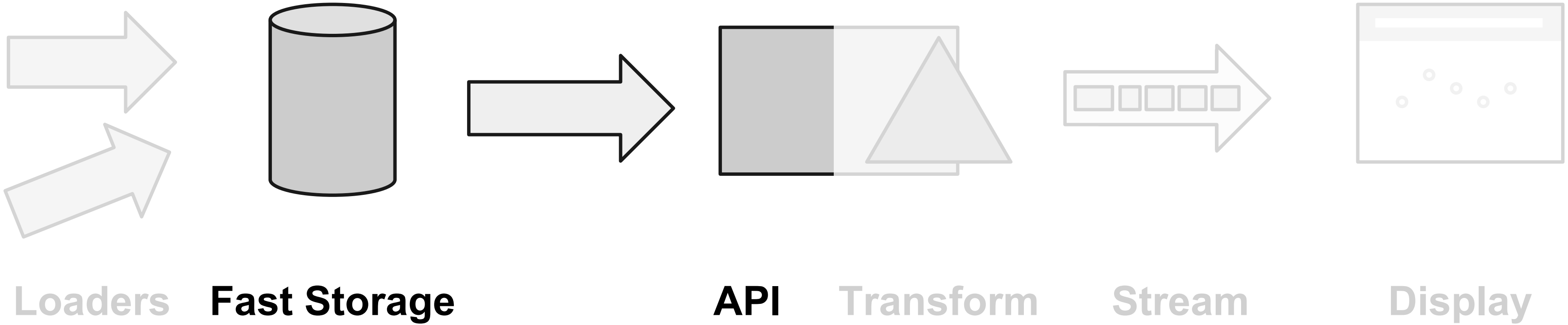
Code for basic graph

```
var charts = [ {
  title: "90th Percentile Request Time",
  groups: [ {
    label: "seconds",
    axis: sideAxis(),
    named: [ { name: "epochs/p90" } ]
  } ]
}];
```

Performance

Asynchronous Streams of Arrays

Perf Study: Read API



Phase I: Optimize Later

```
class Event[K, V](key: K, value: V)
def read(): Seq[Event[K, V]]
```

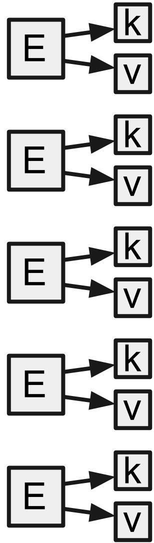
- **Easiest to understand, use, implement**

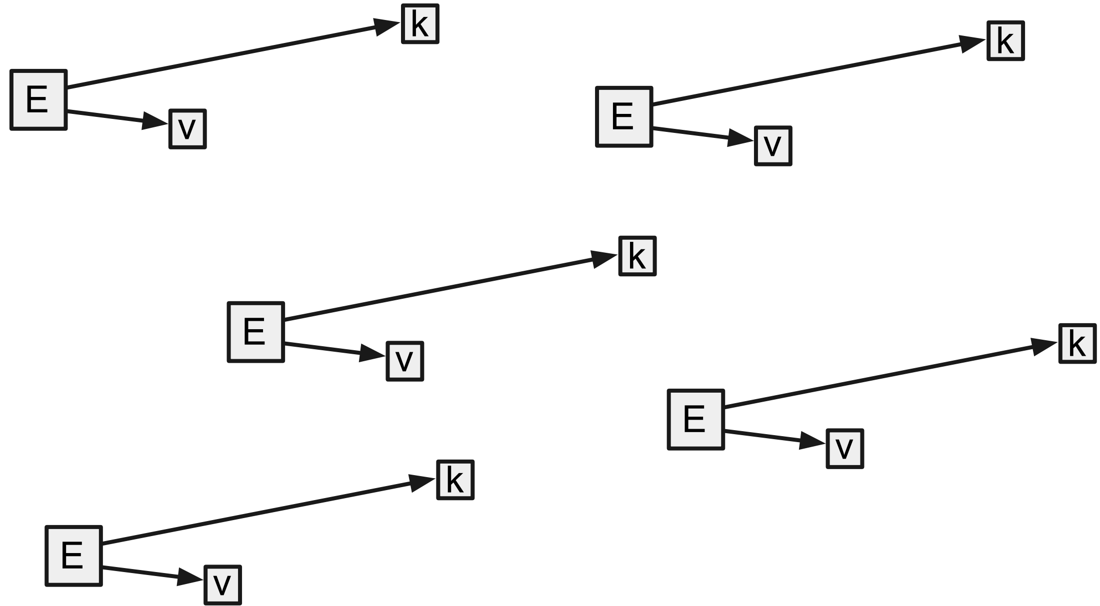
Tip: Perf Approach

1. Add measurements directly in the code
   - repeat single flow: end to end, major components
   - Async? measure synchronous portions and sum
2. Confirm/detail w/CPU profiling (YourKit)
3. Test throughput and latency under load
4. Review GC logs (Censum)

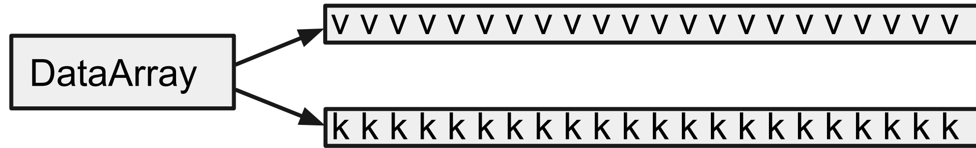
Graph to review perf numbers

GC / CPU utilization





DataArray - Saves Heap



- Arrays of primitives
- High density jvm storage
- Cache locality

Phase II: Array Blocks

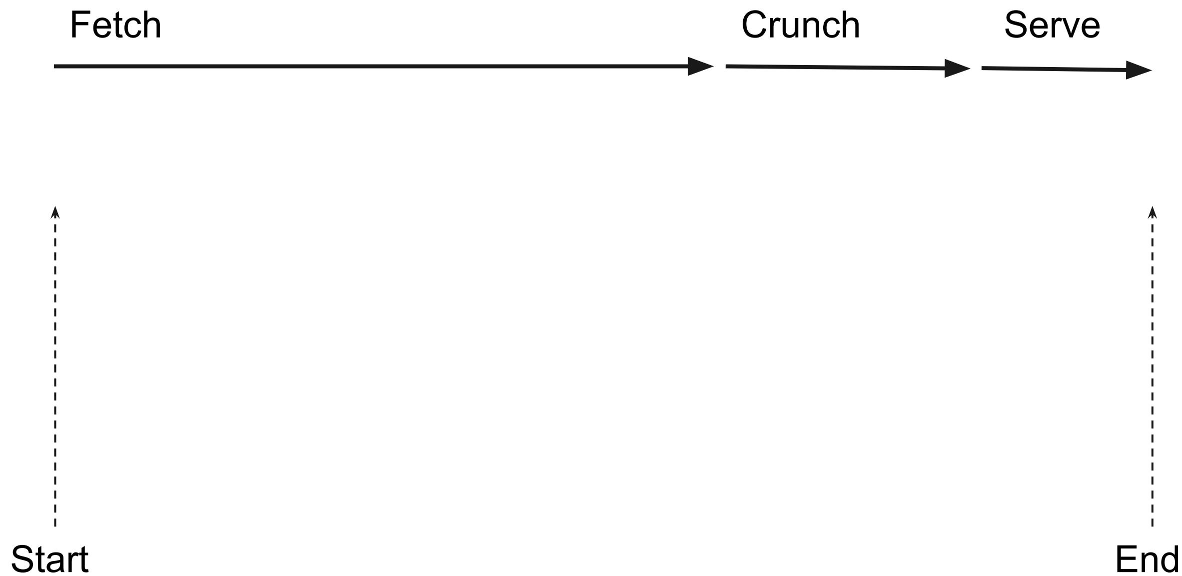
```
class DataArray[K: TypeTag, V: TypeTag]
   ( keys: Array[K], values: Array[V] )

def read(): DataArray[K, V]
```

Are we done yet?

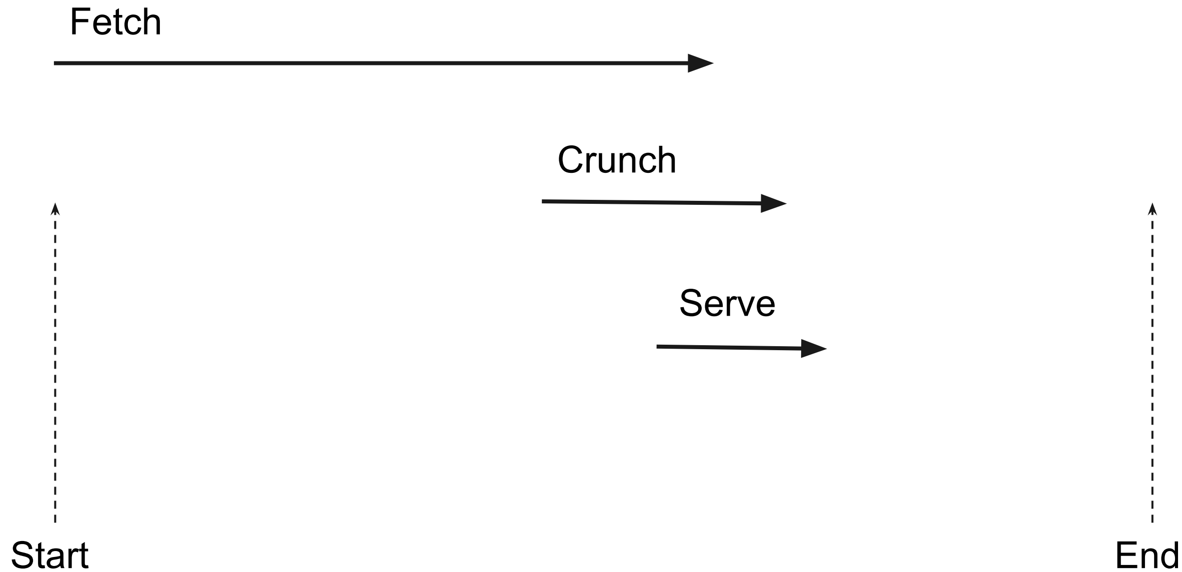
Dense arrays mean less garbage

Tighter loops, more CPU cache efficient

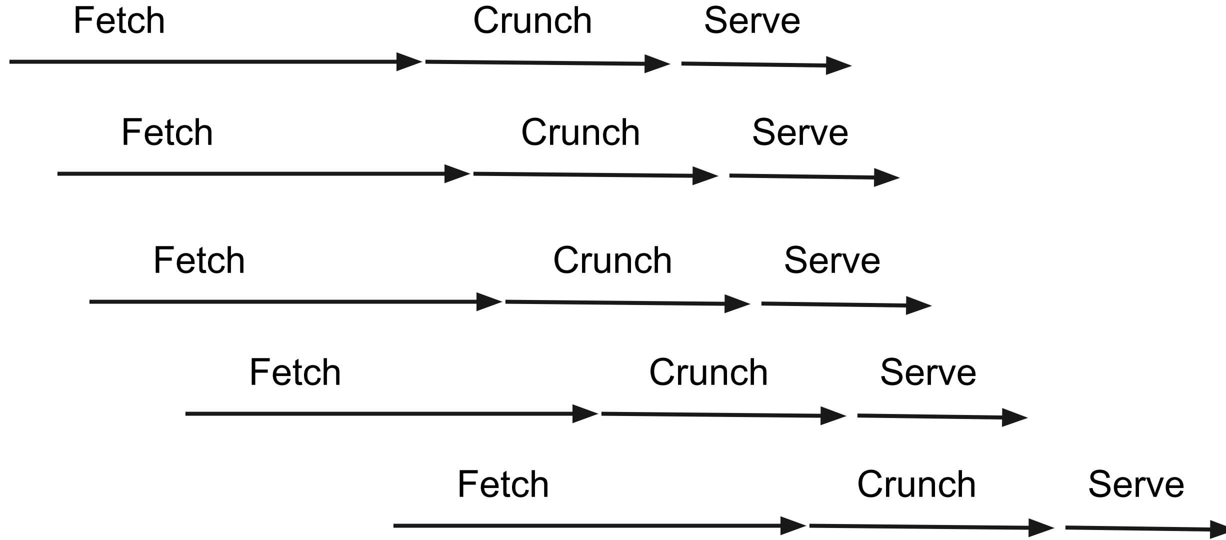
Latency



Overlap Pipeline Stages?



Consider Throughput



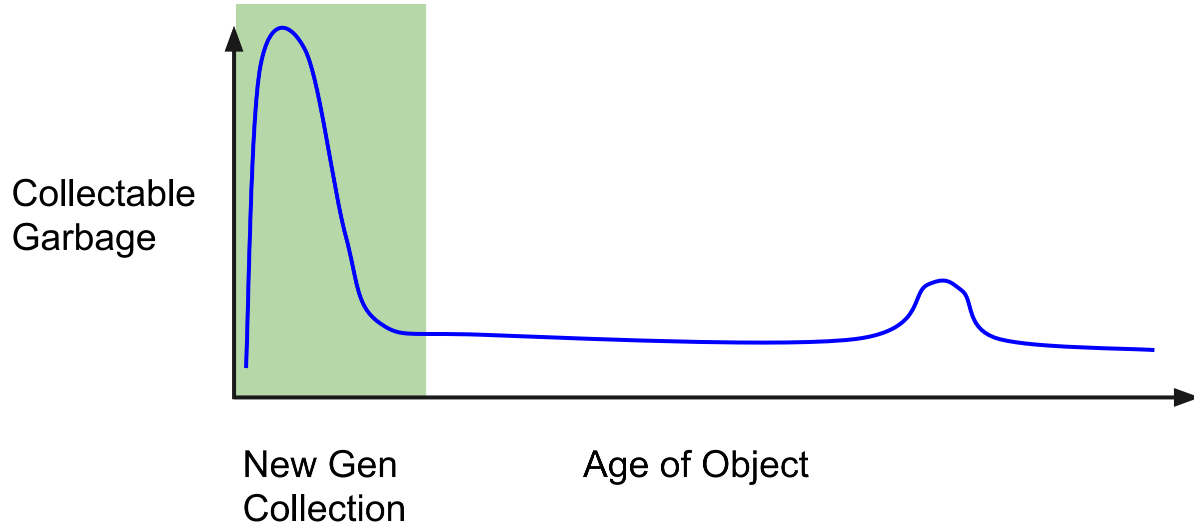
Throughput: Memory

Fetch Crunch Serve

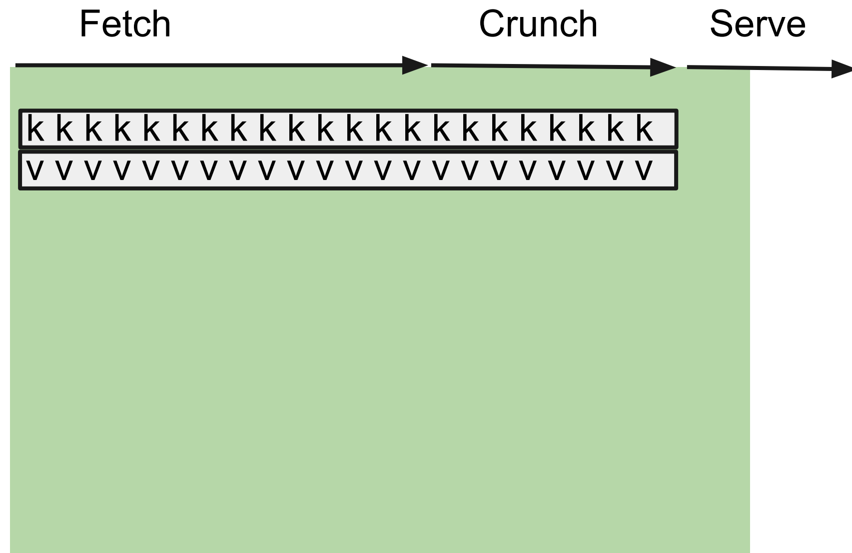
k k k k k k k k k k k k k k k k k k k k k k

v v v v v v v v v v v v v v v v v v v v v v

Generational Hypothesis

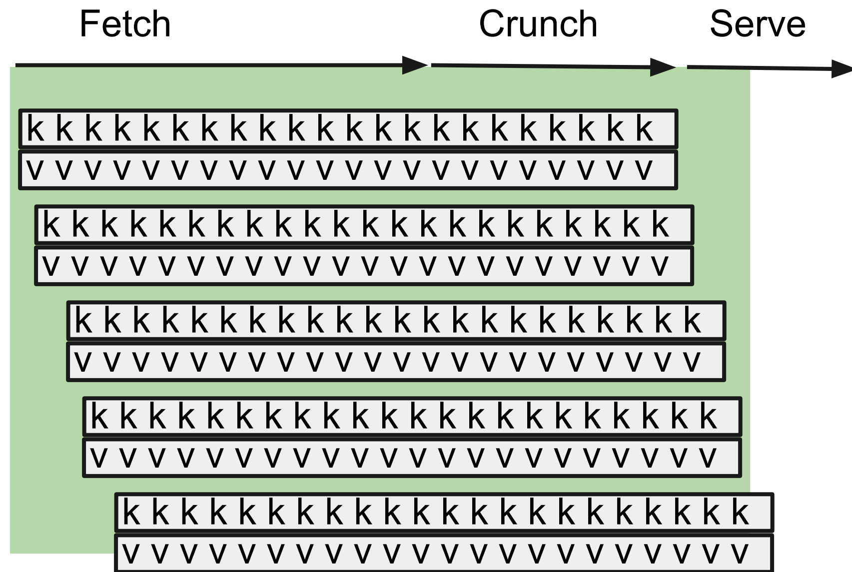


Throughput: Memory



New Gen
Collection

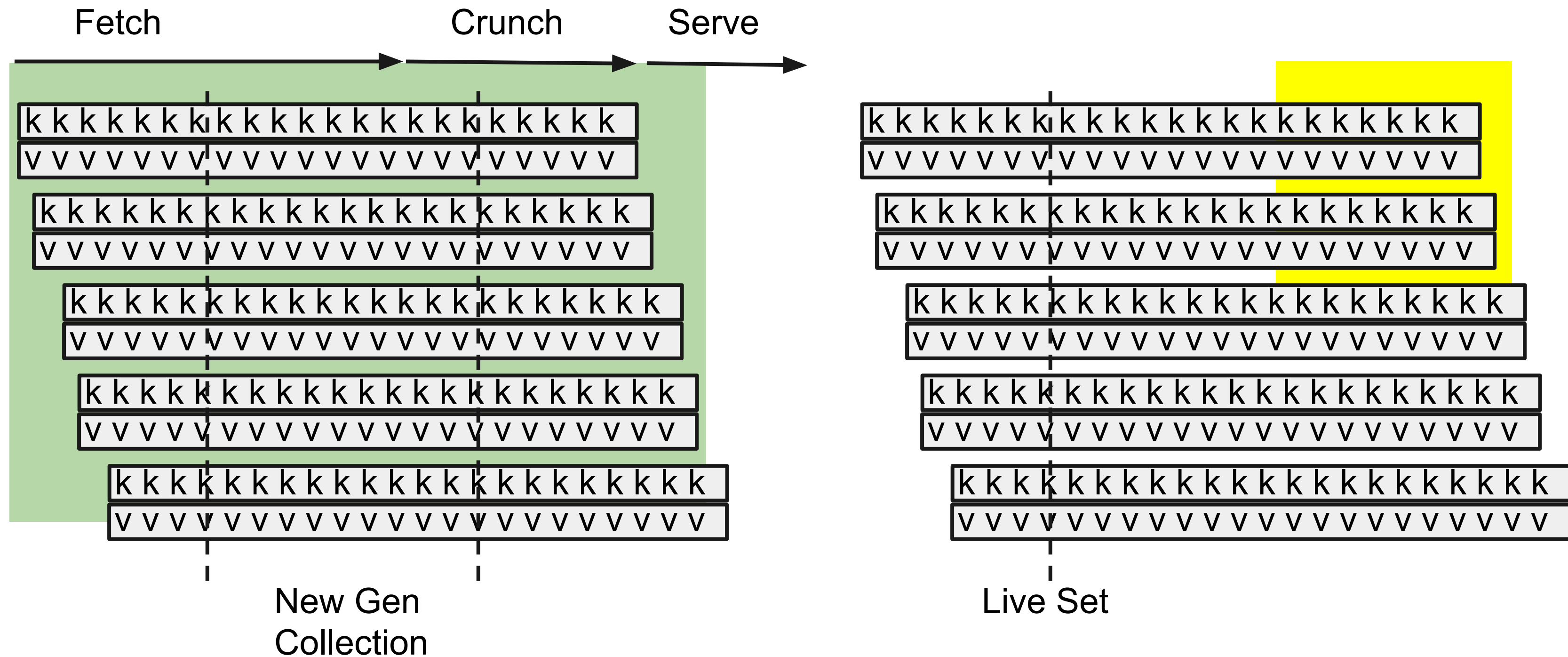
Throughput: Memory



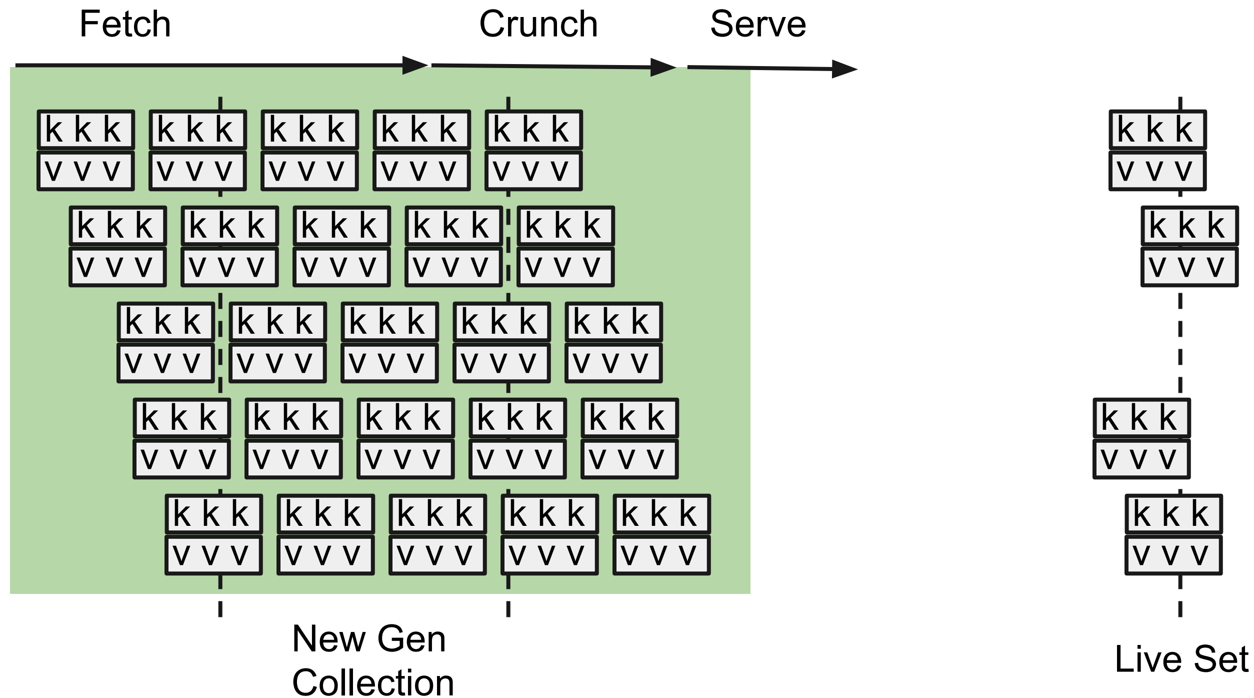
New Gen
Collection

JVM Heap

Throughput: Memory



Solution: Break Big Arrays



Phase III: Async Blocks

```
class DataStream[K: TypeTag, V: TypeTag]
   ( data: Observable[DataArray[K, V]] )

def read(): DataStream[K, V]
```

Go Reactive: save GC

Be more responsive, timely

- Reason enough to go reactive.

Another reason: reduce GC pressure.

- Transient working set is key for throughput

More Blocks

and more Streams

Blocks for Streaming Layer

Kafka is already block streaming internally

Encode your data block-wise anyway

- Encode/decode is more efficient
- Sets the stage for downstream consumers

Blocks for Cassandra

Partition-aligned CQL write batches

- 10x write throughput

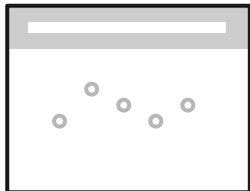
Store 1K blocks instead of (62) elements

- 10x write throughput
- 4x read throughput

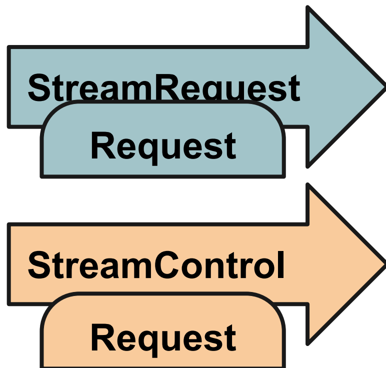
Stream to Graphing Client

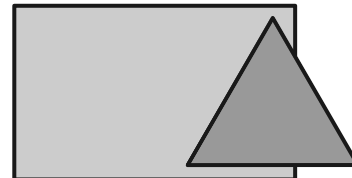
Overlap client processing / communication

- Lowers end to end latency
- Display starts sooner
- Enables live / progressive updates

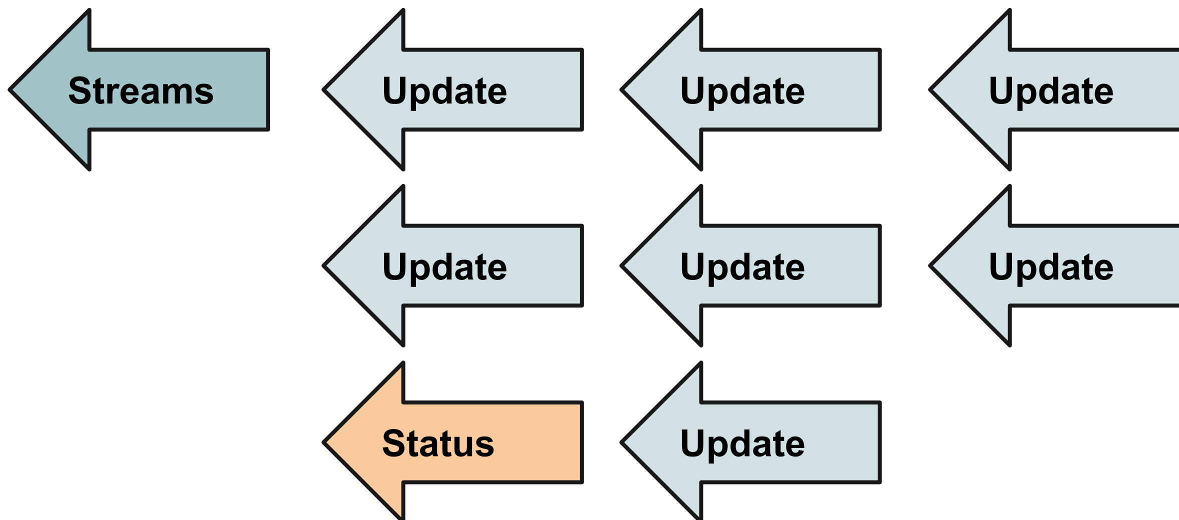


Client

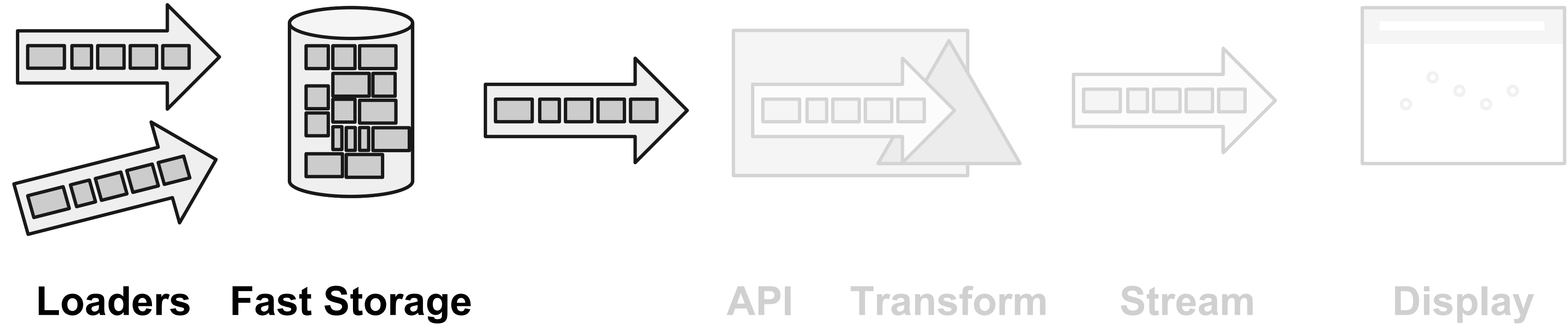




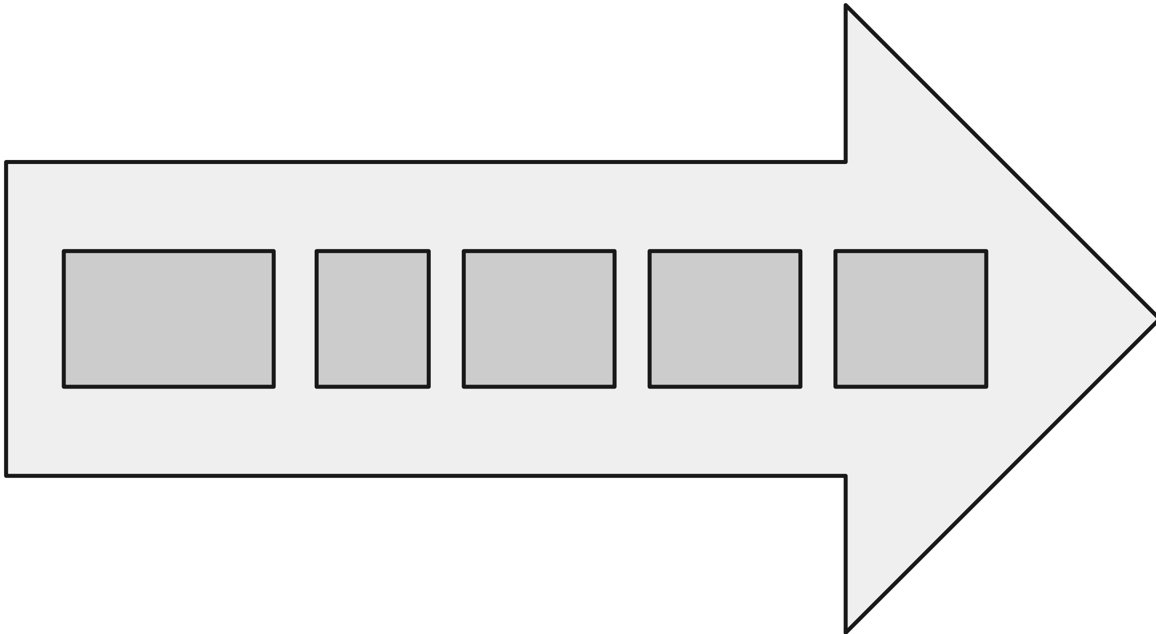
Server



Async Streams of Arrays



Async Streams of Arrays



Architecture

Lambda Architecture?

Streams are great for Sparkle

'Lambda Architecture' is about using streams

WDYT?

Lambda Architecture?

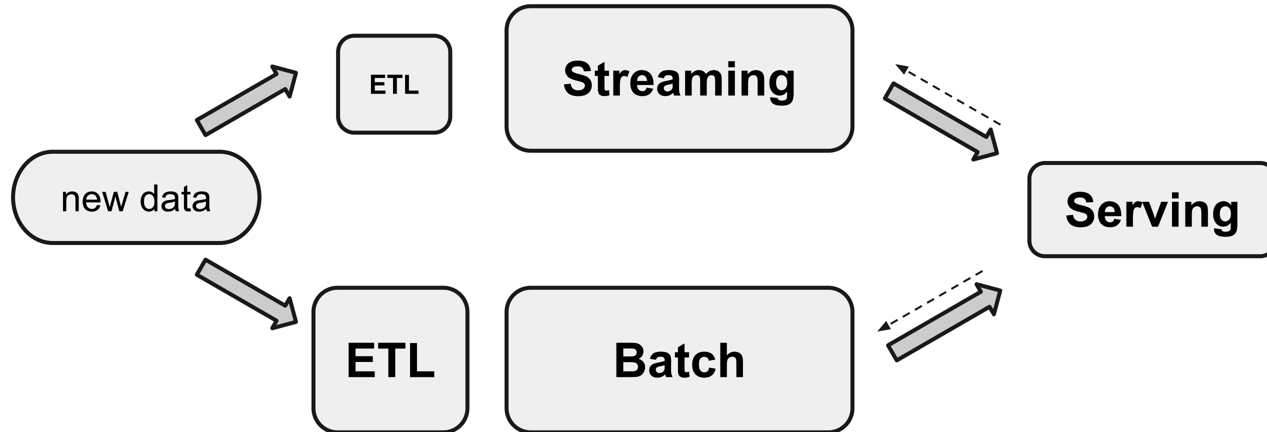
Queries as pure functions that take all data

- +1. we're all FP fans here too.

Batch… is too slow

So combine w/streaming, fast but approximate

Lambda Architecture



Lambda solves for latency

Problem: store + computation is batch slow

Solution: two pipes. streaming, slow/batch

New Problem: two pipes, two platforms, etc.

Streaming or batch: only 2 choices?

Low Latency Available Now

Ingest can be live

    write 1M items / second (RF=3)

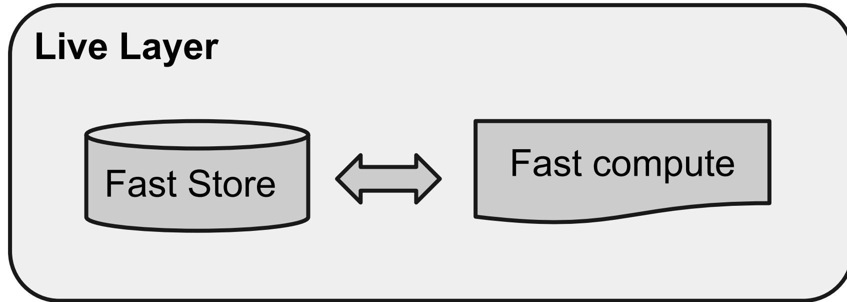
Processing can be live

    fetch + crunch 1M items < 250 msec

5-10x better looks feasible

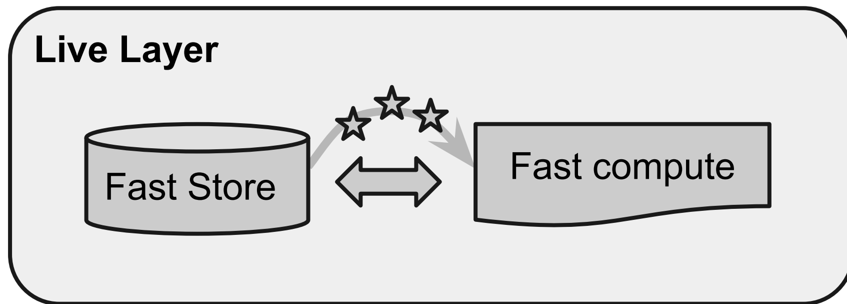
    not near IO bound

Introducing: Live Layer



High volumne
Low latency ingest
Low latency fetch
Transform quickly

Live with Notification



High volumne

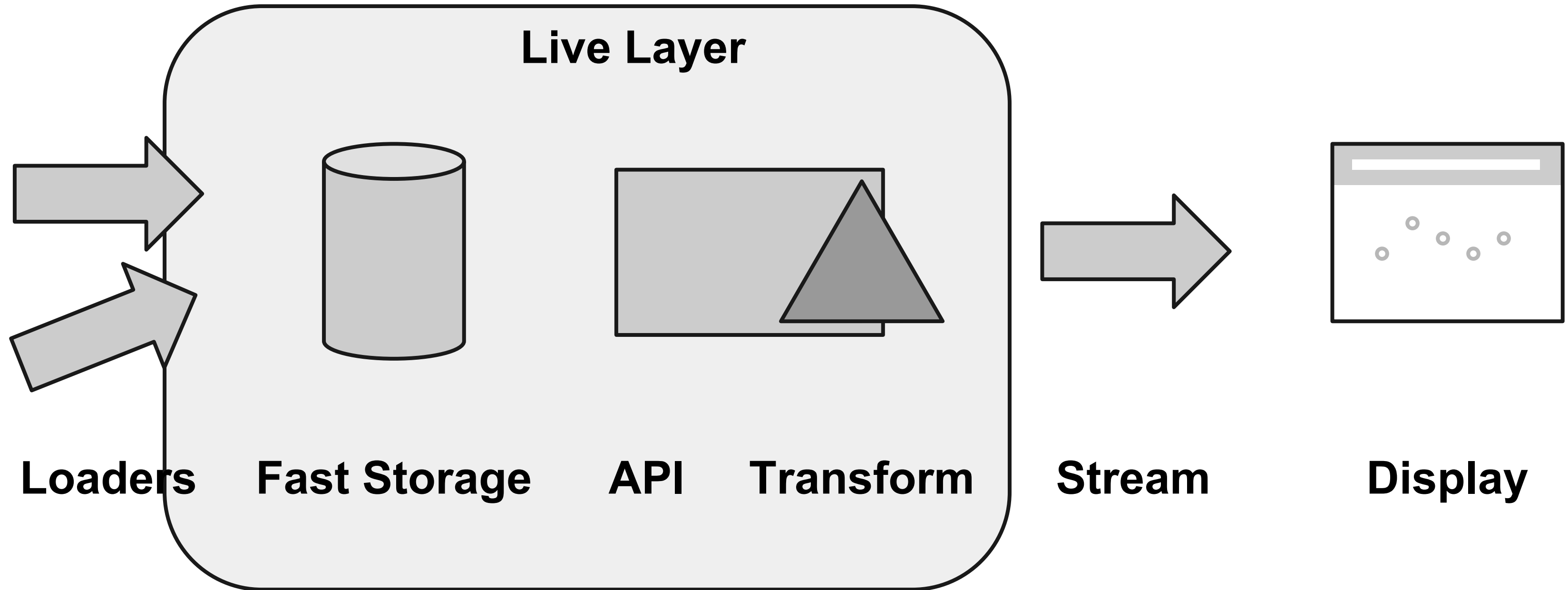
Low latency ingest

Low latency fetch

Transform quickly

Notification

(Sparkle has a Live Layer)



Live + Lambda?

Live: Enables On Demand

Grain aligned - compute live, on request

- Low latency response
- Fresh data
- Trigger as data arrives

Storage Grain

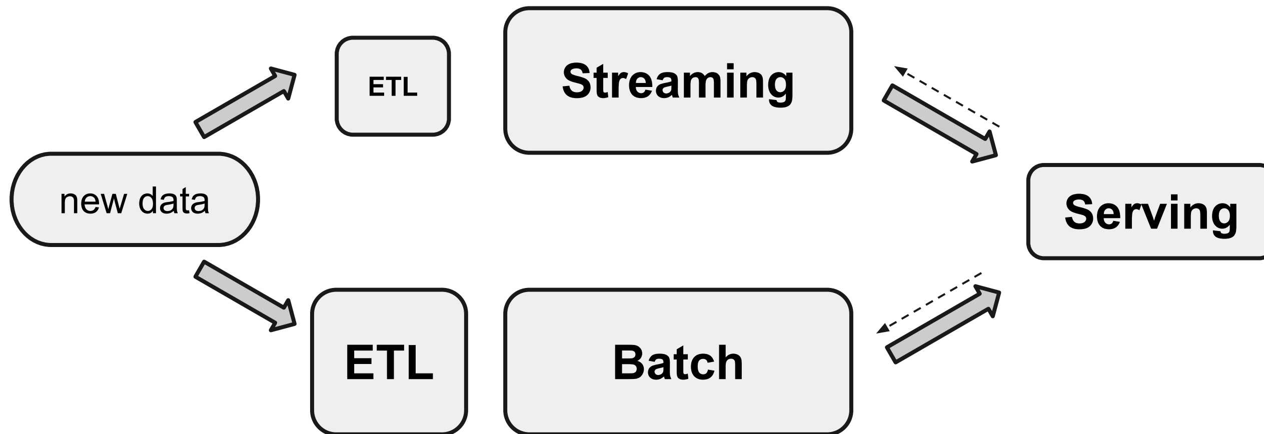
Example: time series server17.cpu.idle

With the grain: fast queries, scans

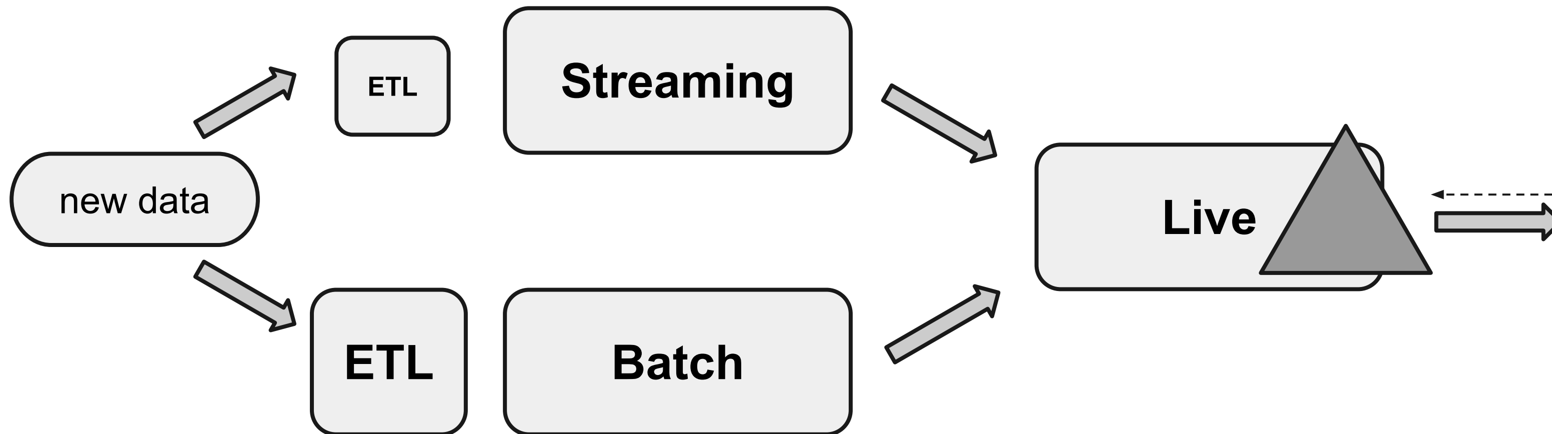
Writes against the grain: only 10x slower

Reads against the grain: cost grows linearly

Lambda Architecture



Live as Serving Layer



Live (vs. Stream Layer)

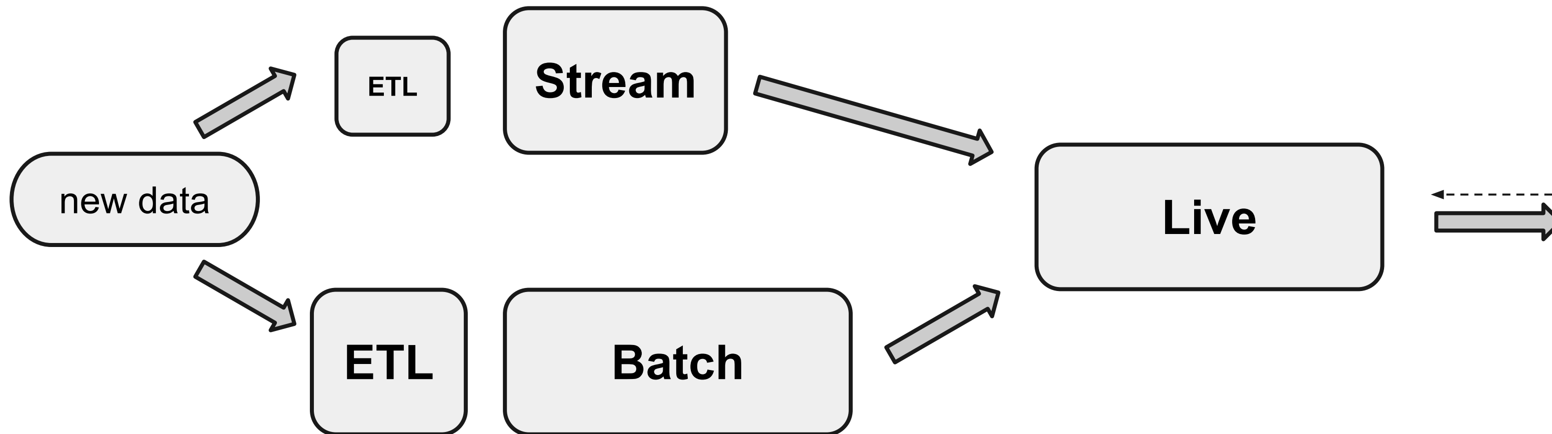
History fully available, not just a window

Efficient calculate views only if needed

Front End to streaming too (serving layer).

Rule of thumb: Per-entity stream can be live

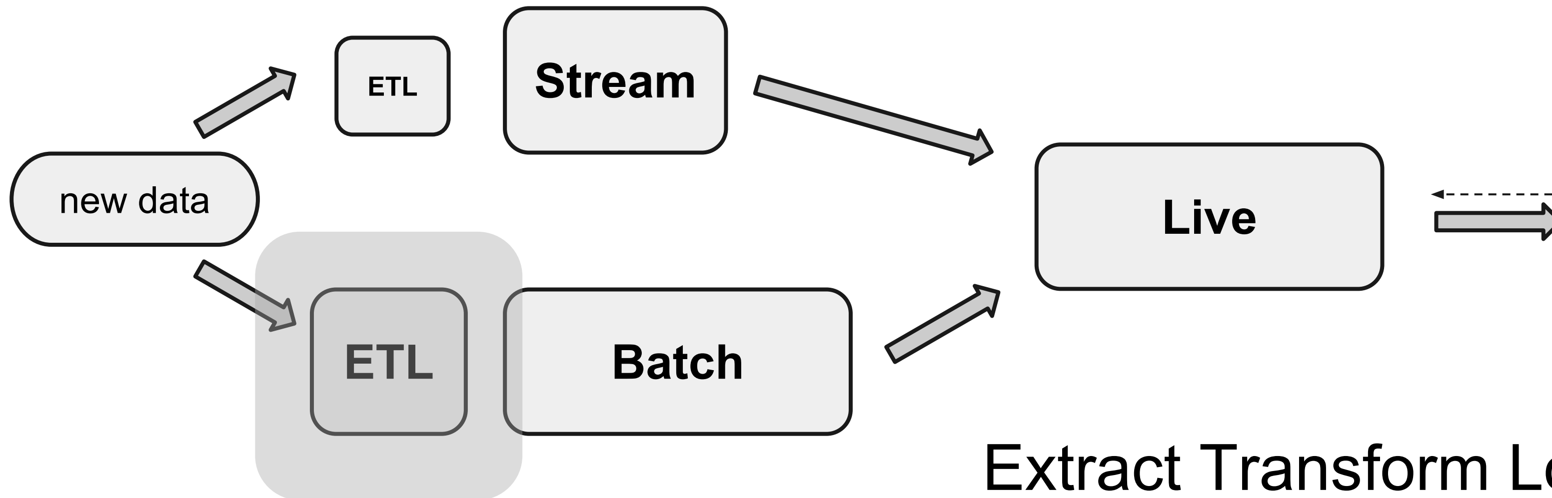
Live + Stream Layer



API for Live Data: Unifies

```
class TwoPartStream[K,V]
  ( initial: DataStream,
    ongoing: DataStream )

def readWithOngoing()
  : TwoPartStream[K,V]
```

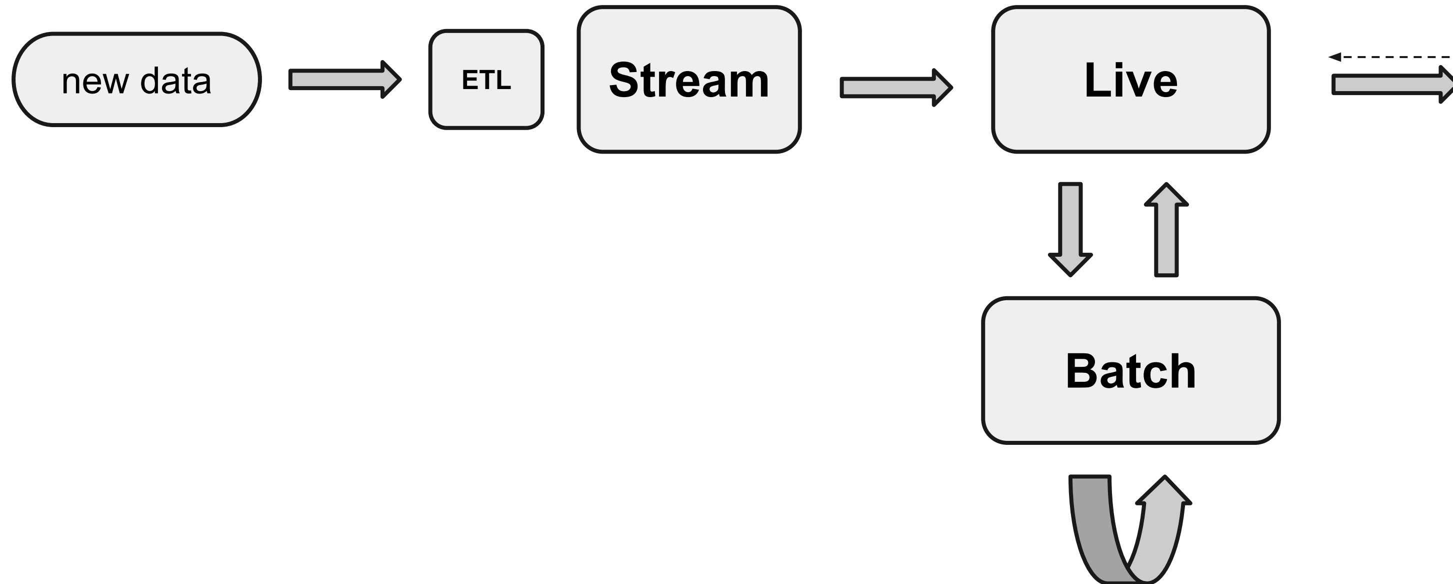
Simplifying ETL



Extract Transform Load

- Format conversion
- Grain alignment

Single Pipe + Batch



Live (vs. Batch Layer)

Flexible parameters, not fixed at batch time

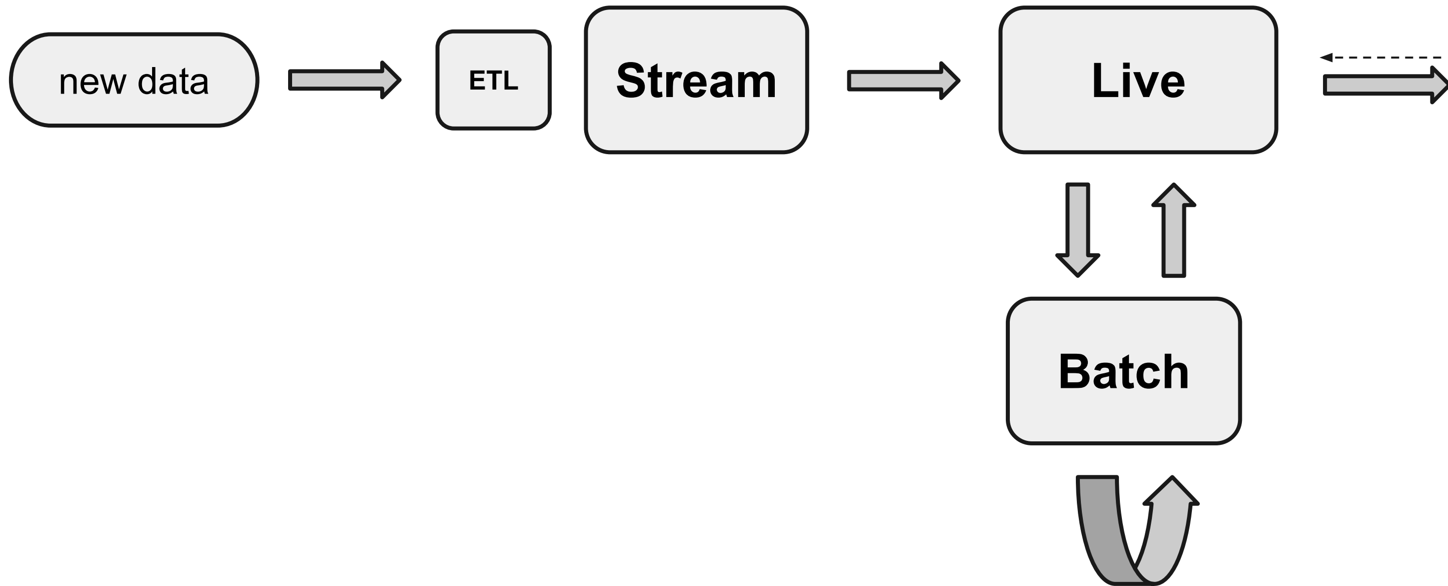
Agile w/o need to bulk reprocess

Fast responses broaden uses

Rule of thumb: Per-entity batch can now be live

+/- One pipe, still two storage+crunch systems

Single Pipe + Batch



Where to Transform Data?

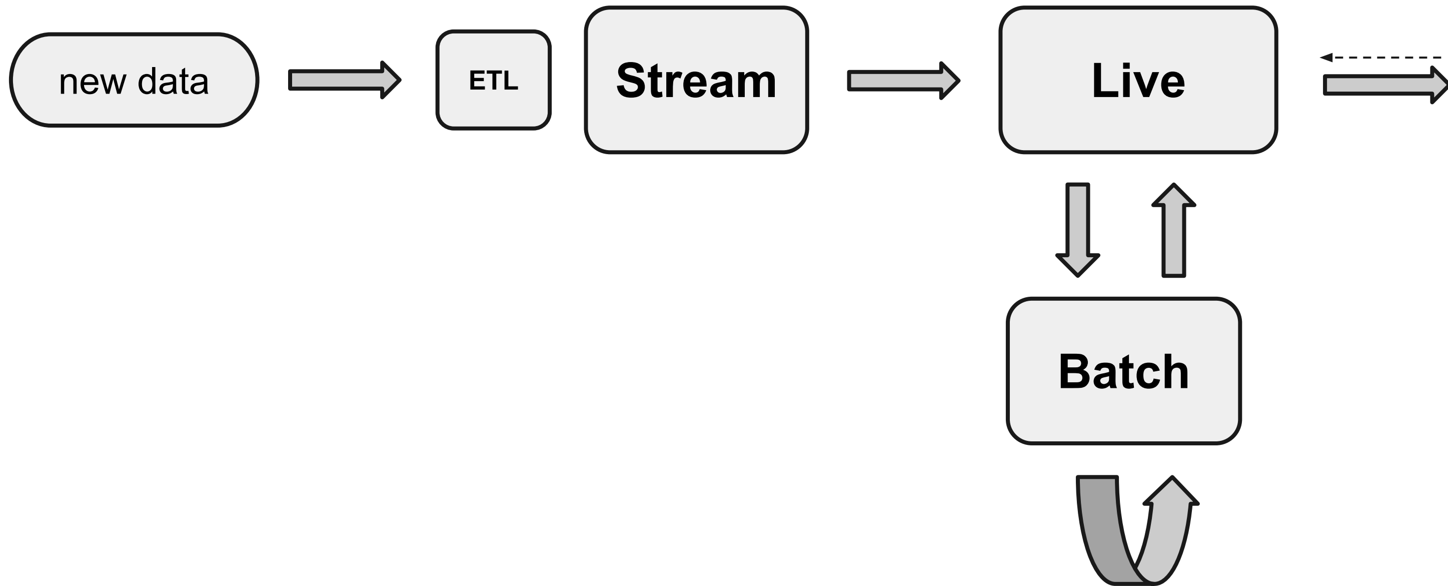
Streaming: ETL

Live: fast, with the grain

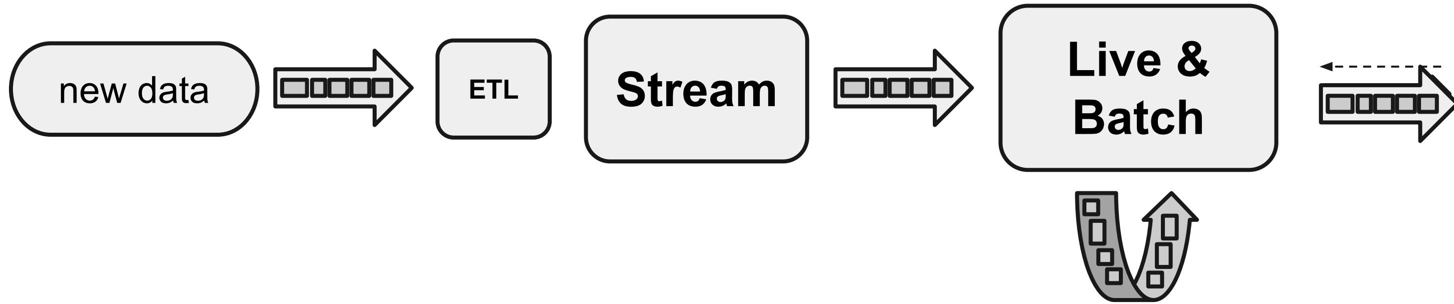
Batch: slow, against the grain

Streaming + Live: fast, against the grain

Single Pipe + Batch



Data Pipeline of the Future



Scala Console Demo

quick graphs from the scala repl

Spark Demo

query against the grain

batch parallel with spark

Sparkle

Tool for easily making zooming graphs

Platform for custom visualizations on live data

- Built on streams
- Generic visualization protocol
- Live data / big data

https://github.com/mighdoll/sparkle

Sparkle

Visualize the Things Scala Days SF 2015

@mighdoll   lee@nestlabs.com

Tips

Tip: Make tests as REPL

Make tests than can be run from the repl

Encourages simpler syntax

Creates a useful tool

Tip: Always make it better

Every commit makes the

Avoid Forbidden Island Syndrome

-> impassible continents

Strive for perfection: clarity, flexibility, efficiency

Scala: Refactoring FTW

- Language power: refactoring enabler
  - composition, abstraction, concision, clarity
- Types: safety net
  - 'works the first time it compiles' - oft heard, true, fun
  - 'works after refactoring' - more important
- Testing: smaller tests, better coverage
  - Bulk is drag
  - Best in class test libraries

Tip: Go Deep and Make

Not just a list of features

Or a deadline

A little learning is a dangerous thing;
Drink deep, or taste not the Pierian spring.

Strive to create a well-made thing.

Challenges

Type recovery

stored data has a fixed type

protocol requests reference data

but these types are unknown at compile time

Dynamic type recovery

serialize type tag

recover: match against known types

recover: match against needed type classes

```
tryNumeric[T: TypeTag]: Try[Numeric[T]]
```

Phase IV: DataStream

Specialization?

Stream Fusion?

n-arrays?

Scratch

Lambda Architecture