Using a sharded Akka distributed data cache as a Flink pipelines integration buffer

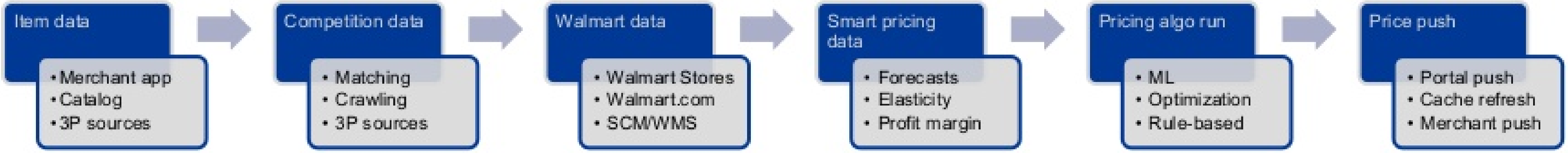
Andrew Torson, Walmart Labs

Agenda

- Smart Pricing @ Walmart
- Flink pipelines: integration patterns
- Technology deep-dive: distributed data cache as a Flink component
- Deployment and usage: how well does it perform and scale?
- Demo (if time permits)

Smart Pricing @ Walmart Labs

- **Algorithmic**: competitive analysis, economic modeling, cost/profit margin/sales/inventory data
- **Rich** ingestion: Walmart Stores, Walmart.com, Merchants, Competition, 3rd party data
- **Large** catalog size: ~ 10M 1st party catalog; ~100M 3rd party/marketplace catalog
- **Multi-strategy**: competitive leadership/revenue management/liquidation/trending/bidding
- **Real-time**: essential inputs ingestion (competition/trends/availability)
- Real-time: any 1P item price is refreshed within 5 minutes on all important input triggers
- Quasi real-time: push-rate-controlled 3P catalog price valuation
- Throughput control: probabilistic filtering/caching/micro-batching/streaming API/backpressure
- Regular batch jobs for quasi-static data: sales forecast/demand elasticity/marketplace statistics



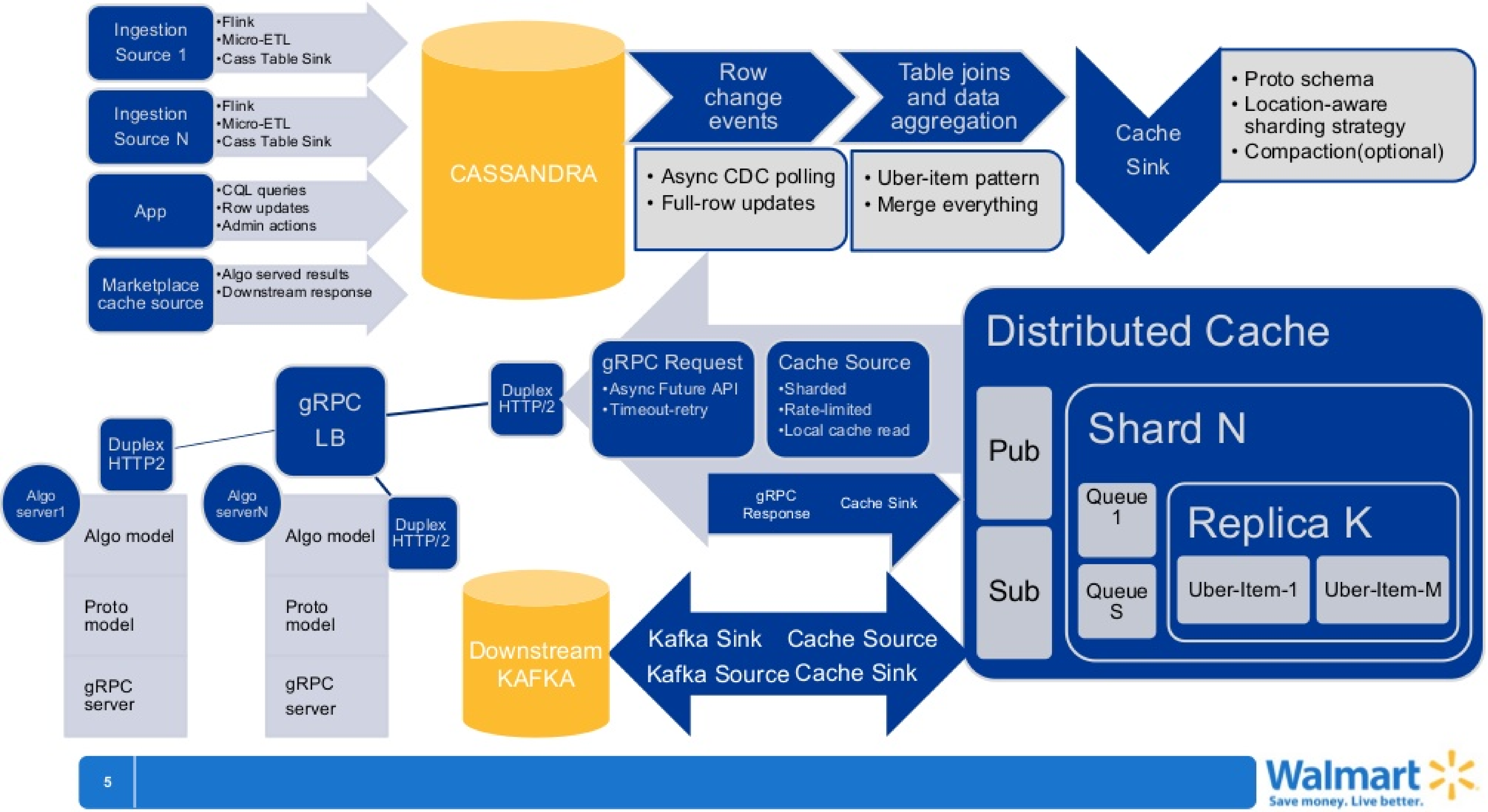
Tech Stack: Flink, Kafka, Akka, Cassandra, Redis, gRPC, Hive, ElasticSearch, Druid, PySpark, Scikit, Azkaban

Walmart Marketplace

- **Large:** more than 100M items at the moment; growing fast
- **Fluid:** vendor/item set up is automated in bulk; thousands of catalog change events/sec
- **Regulated:** items can be automatically de-listed if attributes are out of bounds
- **Real-time:** important attributes (like price bounds) are updated with minimum delay on significant catalog, marketing and logistics events ingested in real-time
- **Data rich:** a lot of ingestion data sources (item set-up, offers and promotions, competitive info, store prices, inventory availability, vendor system inputs, 3rd party analytics etc.)

Streaming pipeline characteristics and challenges:

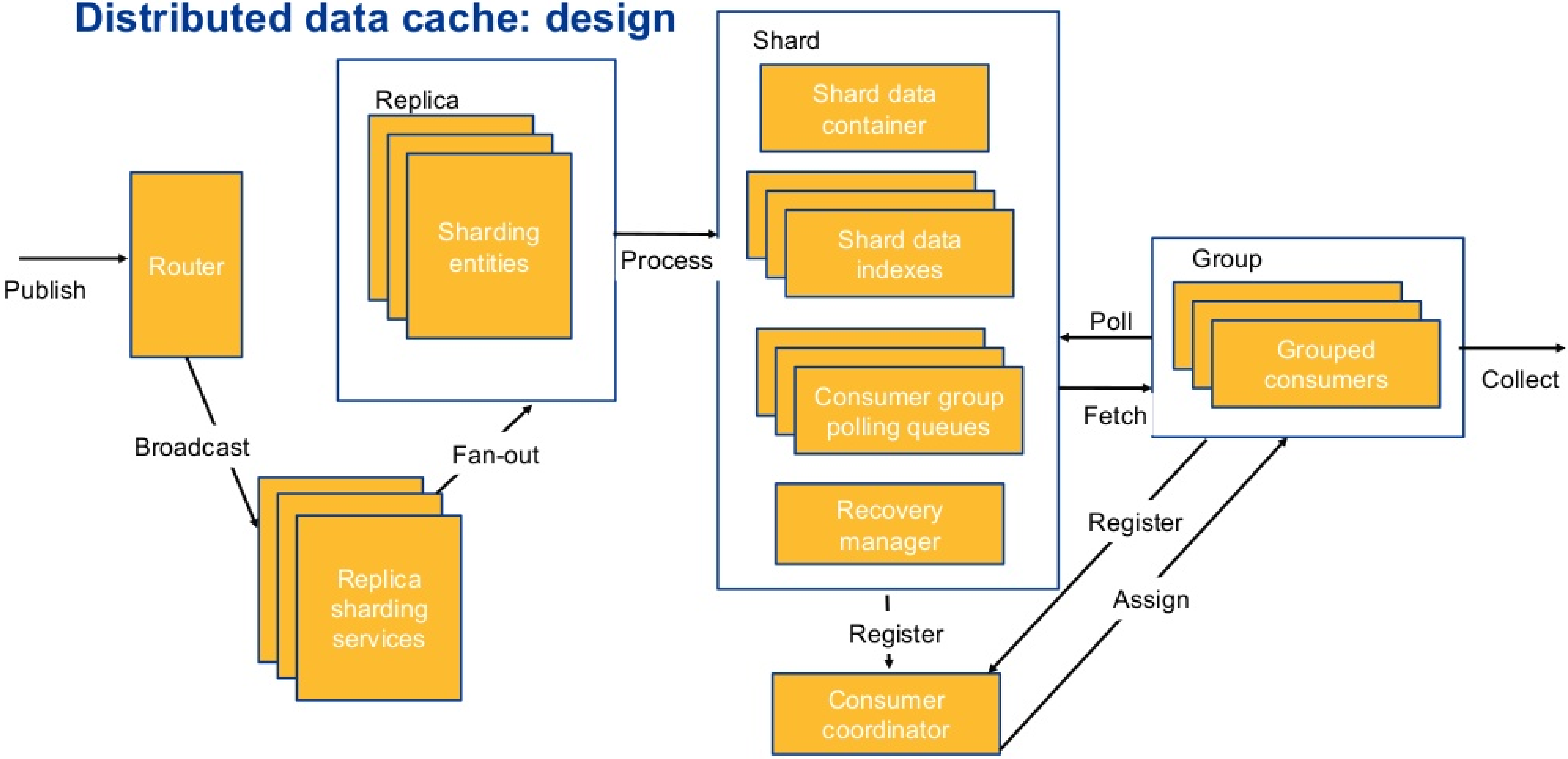
- **Ingestion firehose:** billions of events per day; loose/semi-structured schema
- **Streaming micro-services:** many analytical/intelligent processors including rule-based and ML
- **Backlog control:** backlogs can not be tolerated and must be under strict limits
- **Rate limiting/control:** downstream is slower than the upstream firehose; can not backpressure
- **Co-location/co-sharding:** most of the services should be co-located and co-sharded together


Ingestion Source 1
•Flink
•Micro-ETL
•Cass Table Sink
Ingestion Source N
•Flink
•Micro-ETL
•Cass Table Sink
App
•CQL queries
•Row updates
•Admin actions
Marketplace cache source
•Algo served results
•Downstream response
CASSANDRA
Row change events
Table joins and data aggregation
Cache Sink
• Async CDC polling
• Full-row updates
• Uber-item pattern
• Merge everything
• Proto schema
• Location-aware sharding strategy
• Compaction(optional)
Distributed Cache
Pub
Sub
Shard N
Queue 1
Queue S
Replica K
Uber-Item-1
Uber-Item-M
gRPC Request
•Async Future API
•Timeout-retry
Cache Source
•Sharded
•Rate-limited
•Local cache read
gRPC Response
Cache Sink
Duplex HTTP/2
gRPC LB
Duplex HTTP2
Duplex HTTP/2
Algo server1
Algo serverN
Algo model
Proto model
gRPC server
Algo model
Proto model
gRPC server
Downstream KAFKA
Kafka Sink
Cache Source
Kafka Source
Cache Sink
Walmart
Save money. Live better.

Flink pipelines: integration patterns

- **Micro-ETL:** firehose delivering data to Cassandra
  - No backpressure, no complex logic, no backlogging, low-latency writes
- **Cassandra CDC:** our own Akka-based sharded CDC micro-service
  - Serves all writes to Cassandra, sequentially within each shard
  - Micro-batched background polling CDC event publisher
- **Uber-item:** Protobuf-based container for easy merging of CDC events
  - Table joins share write schema; each table owns a subset of fields in it
  - Aggregations are served as collections fields
- **Sharded Cache:** Akka-based distributed data cache + consumer polling queues + pub-sub
  - Sharding is the key feature: it is a collection of independent replicated shards
  - Backlog is strictly limited: all events are mutating long-living key-value records within shards
  - Consumers can subscribe independently any-time: coordinator will assign shards for them to poll
  - Consumer reading rate is controlled through polling and can be much slower than the write rate
- **gRPC micro-services:** allow to use streaming HTTP/2 clients within Flink
  - Useful for algorithmic services integration
- **Kafka request/response**: a pair of Flink Kafka Sink/Source implementing duplex streaming
  - Useful for downstream services integration
  - Event-journaled via a Cassandra table for request-response correlation (reflects Kafka backlog)

Distributed data cache: design



Quick tour of Akka clustering patterns

- **Gossip:** de-centralized cluster formation, information dissemination and heartbeat monitoring
  - Good for state-sharing background services like cache replication
- **Singleton:** single instance-controlled actor instance deployed & maintained in a cluster
  - Good for coordination and registration
- **Router:** load-balanced instance-count controlled pool of actors deployed & maintained in a cluster
  - Good for load balancing and remote micro-services deployment
- **Sharding:** fan-out pattern creating a micro-service actor instance for each unique shard entity ID
  - Good for big data-set partitioning and parallel processing
  - Multiple composable micro-service instances can be co-located within a shard
- **Distributed data:** strong eventual-consistent data replication service (not fit for big data)
- **Distributed pub-sub:** cluster-wide event bus

You can build your own makeshift Kafka, Cassandra and Redis quickly with Akka

However, the devil is in the details and the proof is in the pudding

Flink wrappers : Source

```
public interface SourceFunction<T> extends Function, Serializable {

  void run(SourceContext<T> ctx) throws Exception;

  void cancel();

  interface SourceContext<T> {

     void collect(T element);

    Object getCheckpointLock();

    void close();
  }
}
```

- Just a couple of streaming sources readily-available
- No cache source
- Interface is very general
- No explicit rate-control or back-pressure
- Kafka source is a good reference for any polling source
- Checkpointing is essential: use standard ListState<> mechanism in Flink
- Expose monitoring metrics via Flink-metrics

We have implemented a new ShardedCacheSource<> component which is based on our Akka sharded cache machinery

Flink wrappers : Sink and AsyncMap

```
@Public
public interface SinkFunction<IN> extends Function, Serializable {

  default void invoke(IN value, Context context) throws Exception {
    invoke(value);
  }

  interface Context<T> {


    long currentProcessingTime();

    long currentWatermark();

    Long timestamp();
  }
}
```

- Many streaming sources readily-available
- Interface is very general: very easy to implement
- No explicit rate-control or back-pressure
- Expose monitoring metrics via Flink-metrics
- AsyncMap: basically a cache query followed by a flat-map
- AsyncMap: works like a Cassandra query (requires shardId and query field indexes)

We have implemented new ShardedCacheSink<> and ShardedAsyncMap<> components which are simple Akka clients using our Akka sharded cache machinery

Deployment and usage

- Standalone cluster or co-located within Flink JVMs
- RAM and IO are crucial, message compaction helps a lot but consumes CPU
- Monitoring is very important: expose metrics and consider tracing
- Akka actors may quietly and quickly get their mailboxes full
- Co-locating micro-services with Flink and Akka gets the full speed marks
- Locality-aware cache consumer assignment is important: local cache read and writes

- Resilience and HA are very good: Akka takes care of it
- Scalability is very good with sharding: tens of thousands of shards with million records in each without losing throughput
- Throughput performance is very good: cache source read rate beyond $1K messages/sec per Flink subtask; cache query latency 95% is under 3ms
- Akka distributed data background replication becomes slow in heavy traffic: had to reject it and write our own
- Full cache recovery may take tens of minutes with thousands of shards because of a bottleneck in Akka shard start-up coordination (one-time cost)

Questions?