deconstructing
LAMBDA

Philly ETE 2014 - Darach Ennis - @darachennis

A journey from speed at any cost - to unit cost at considerable scale

Philly ETE 2014 - Darach Ennis - @darachennis

UBIQUITI NETWORKS

EdgeMAX airMAX airFiber airVision UniFi mFi BUY

Celebrate the Great Outdoors

with 2 new UniFi Access Points
UAP-Outdoor+ with Multi-Lane RF Technology
and UAP-Outdoor AC

Read more

MULTI-LANE RF Technology

Wi Fi CERTIFIED

UBIQUITI world network

Sign Up Now!

THE FUTURE CAN'T WAIT

Ubiquiti Customer Stories

Watch UniFi Case Study Video

Enforta Deploys Ubiquiti's airMAX Technology to 150 Russian Cities, Bringing Broadband to Russia's Remote Regions - Learn more

2/7

UBIQUITI NETWORKS COMMUNITY

Discuss, find and share solutions with Ubiquiti users around the world.

Go to Community Site

SUPPORT

- Support Main
- Downloads
- Training
- Legacy Products

COMPANY

- About Us
- Contact Us
- Career
- Investors

UBIQUITI NEWS & UPDATES

Newsletter Sign Up

Terms of Service Privacy policy

small ~~**FAST**~~
DATA **guy**

Interested in Data Patterns and War Stories (aka: Data Architectures)

Philly ETE 2014 - Darach Ennis - @darachennis

Big Data

"The techniques and technologies for such data-intensive science are so different that it is worth distinguishing data-intensive science from computational science as a new, fourth paradigm"

- Jim Gray

The Fourth Paradigm: Data-Intensive Scientific Discovery. - Microsoft 2009

Scale vs Speed

“Premature optimisation is the root of all evil.”

- Donald Knuth

“Premature evil is the root of all optimisation.”

- Nitsan Wakart

DATA intensive science **@SCALE**

Philly ETE 2014 - Darach Ennis - @darachennis

Mechanical Sympathy

```
public class Zero {
    private Zero() { }

    public static final byte[] digest(byte[] rawData)
        throws NoSuchAlgorithmException {
        final MessageDigest digest = MessageDigest.getInstance("SHA-256");
        return digest.digest(rawData);
    }

    public static String string(byte[] rawData) {
        final StringBuilder sb = new StringBuilder();

        for(byte b : rawData) {
            sb.append(String.format("%02x", b& 0xFF));
        }

        return sb.toString();
    }

    // string(digest(foo)) -> ~41576/sec
}
```

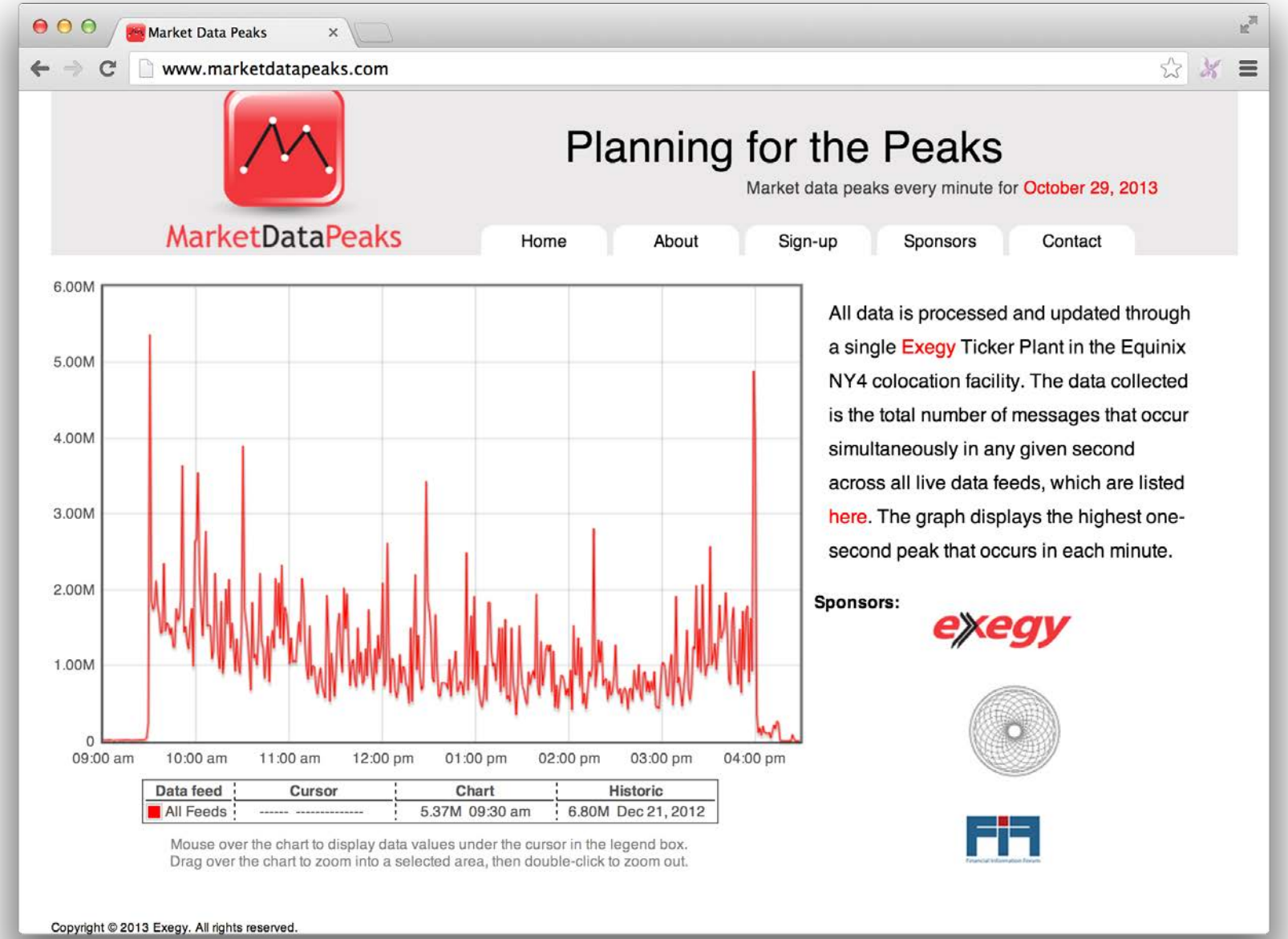
Mechanical Sympathy

```
public class One {
    private One() { }

    private static final char[] LUT = "0123456789abcdef".toCharArray();

    public static final byte[] digest(byte[] rawData)
        throws NoSuchAlgorithmException {
        final MessageDigest digest = MessageDigest.getInstance("SHA-256");
        return digest.digest(rawData);
    }

    public static String string(byte[] rawData) {
        final StringBuilder sb = new StringBuilder();

        for(byte b : rawData) {
            sb.append(LUT[(b & 0xF0) >>> 4]);
            sb.append(LUT[b & 0x0F]);
        }

        return sb.toString();
    }

    // string(digest(foo)) -> ~1071783/sec or a 25x speedup
}
```

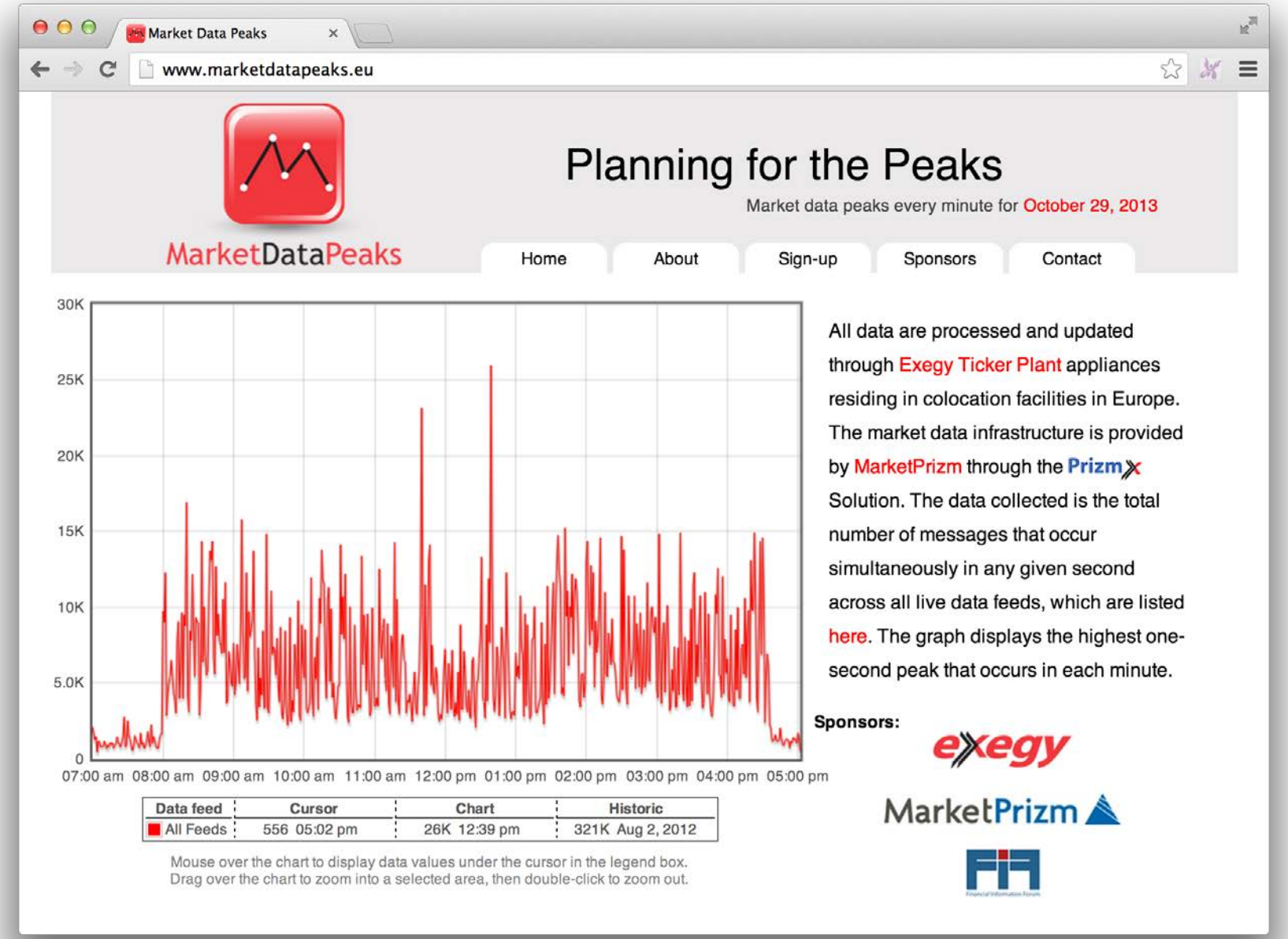
Mechanical Sympathy

```
private static final char[] LUT = "0123456789abcdef".toCharArray();
private static final ThreadLocal<MessageDigest> A = new ThreadLocal<MessageDigest>();
private static final ThreadLocal<StringBuilder> S = new ThreadLocal<StringBuilder>();

private static final MessageDigest a() {..
private static final StringBuilder s() {..

public static final byte[] digest(byte[] rawData)
    throws NoSuchAlgorithmException {
    final MessageDigest digest = a(); // Thread local, lazy alloc
    return digest.digest(rawData);
}

public static String string(byte[] rawData) {
    final StringBuilder sb = s(); // Thread local, lazy alloc, set length 0

    for(byte b : rawData) {
        sb.append(LUT[(b & 0xF0) >>> 4]);
        sb.append(LUT[b & 0x0F]);
    }

    return sb.toString();
}

// string(digest(foo)) -> 1365798/sec or 32x speedup
```

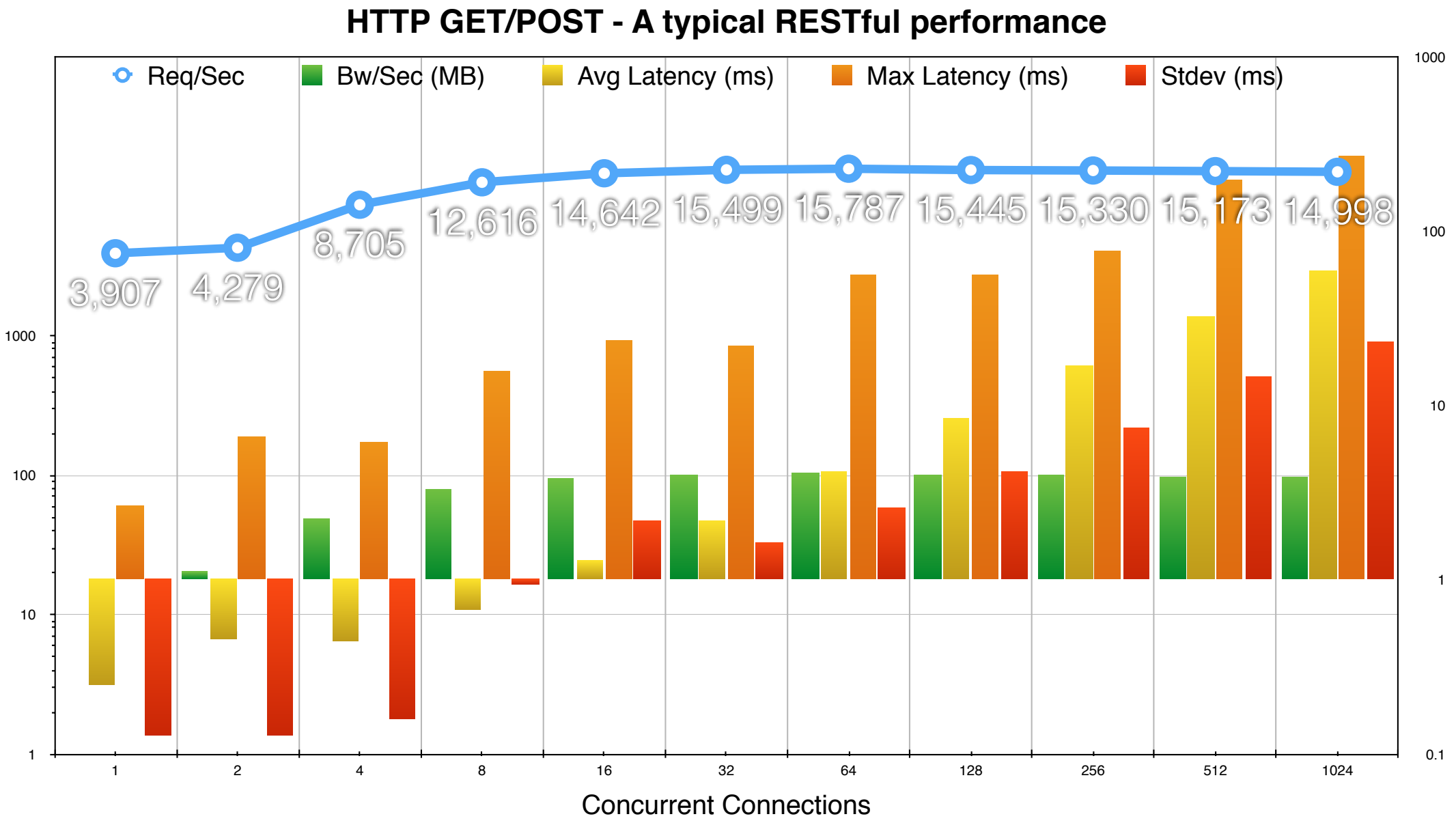
A Wall Street Second



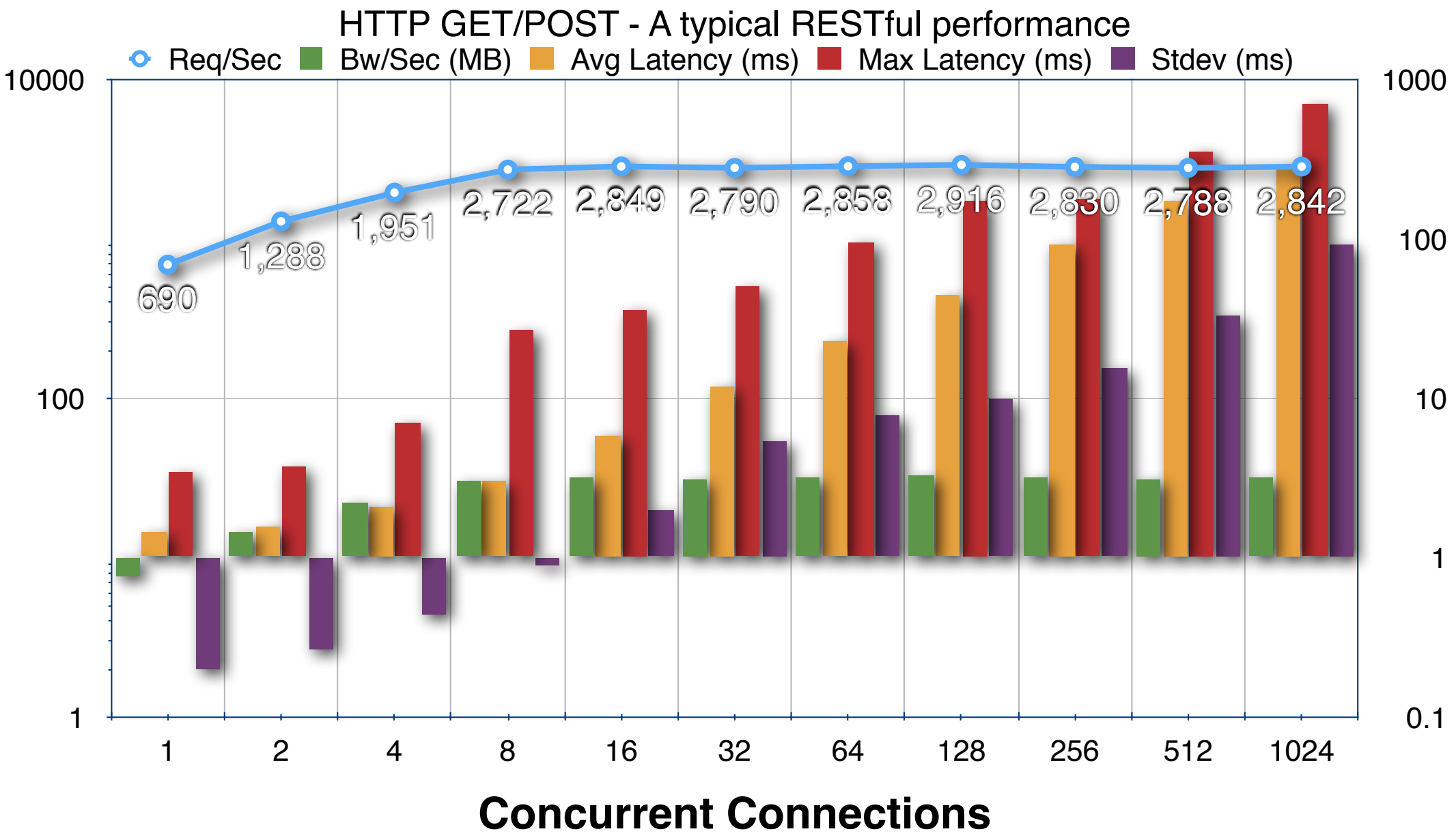
A Swiss Second



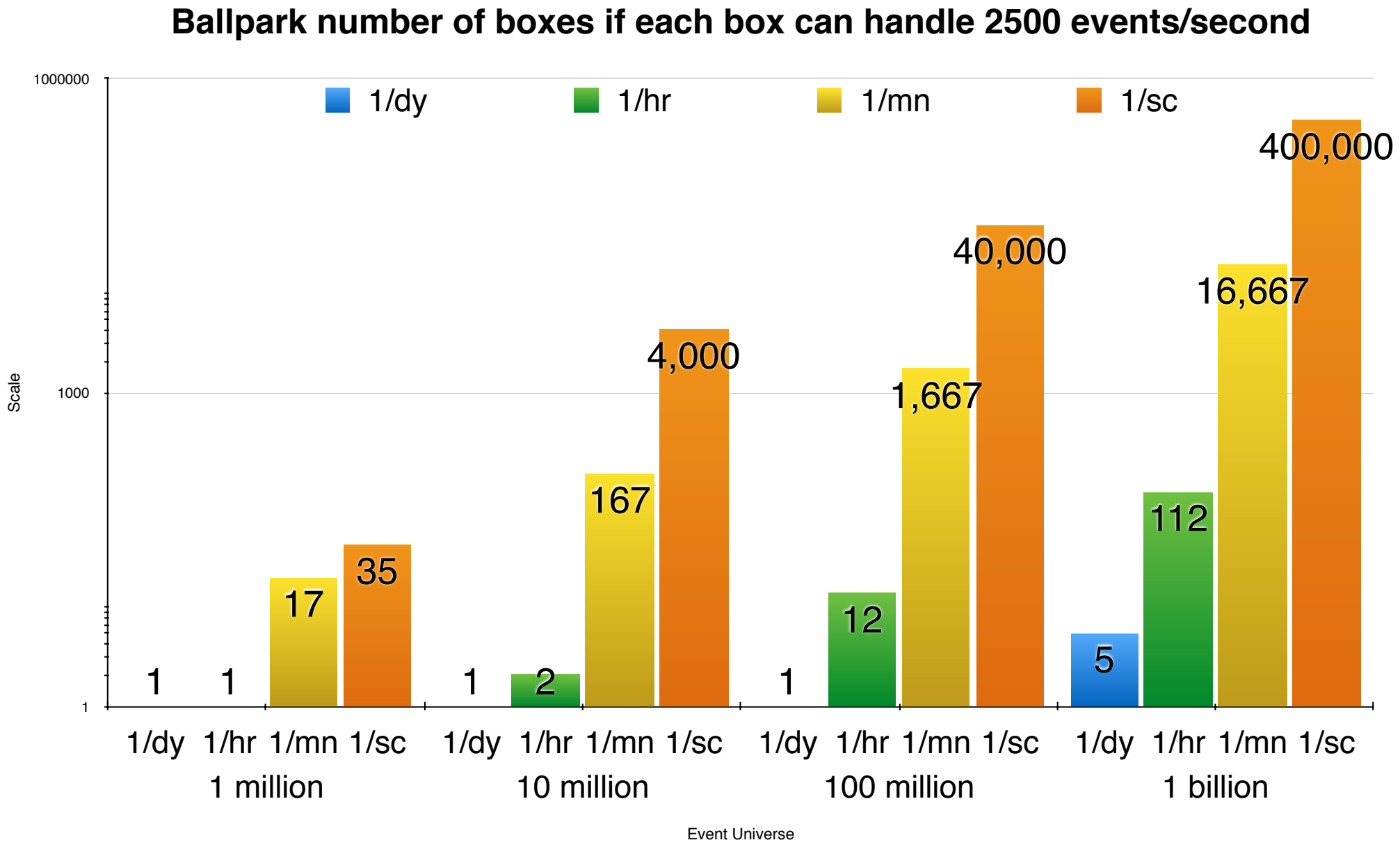
Small Data? <= 128bytes



Small Data? <= 1K



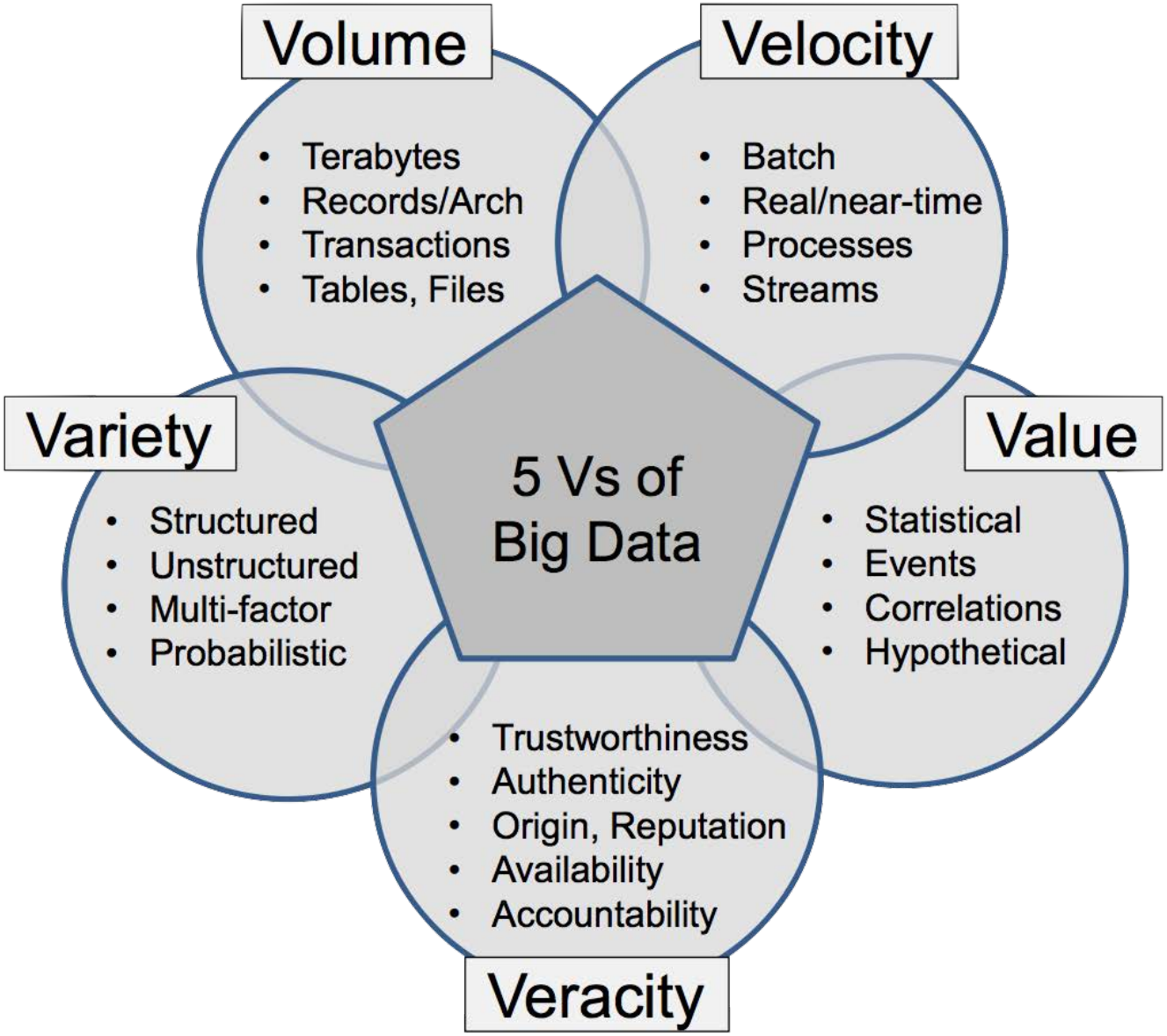
Big Events - 1Billion Sources



Data Sympathy?

Philly ETE 2014 - Darach Ennis - @darachennis

5 V's

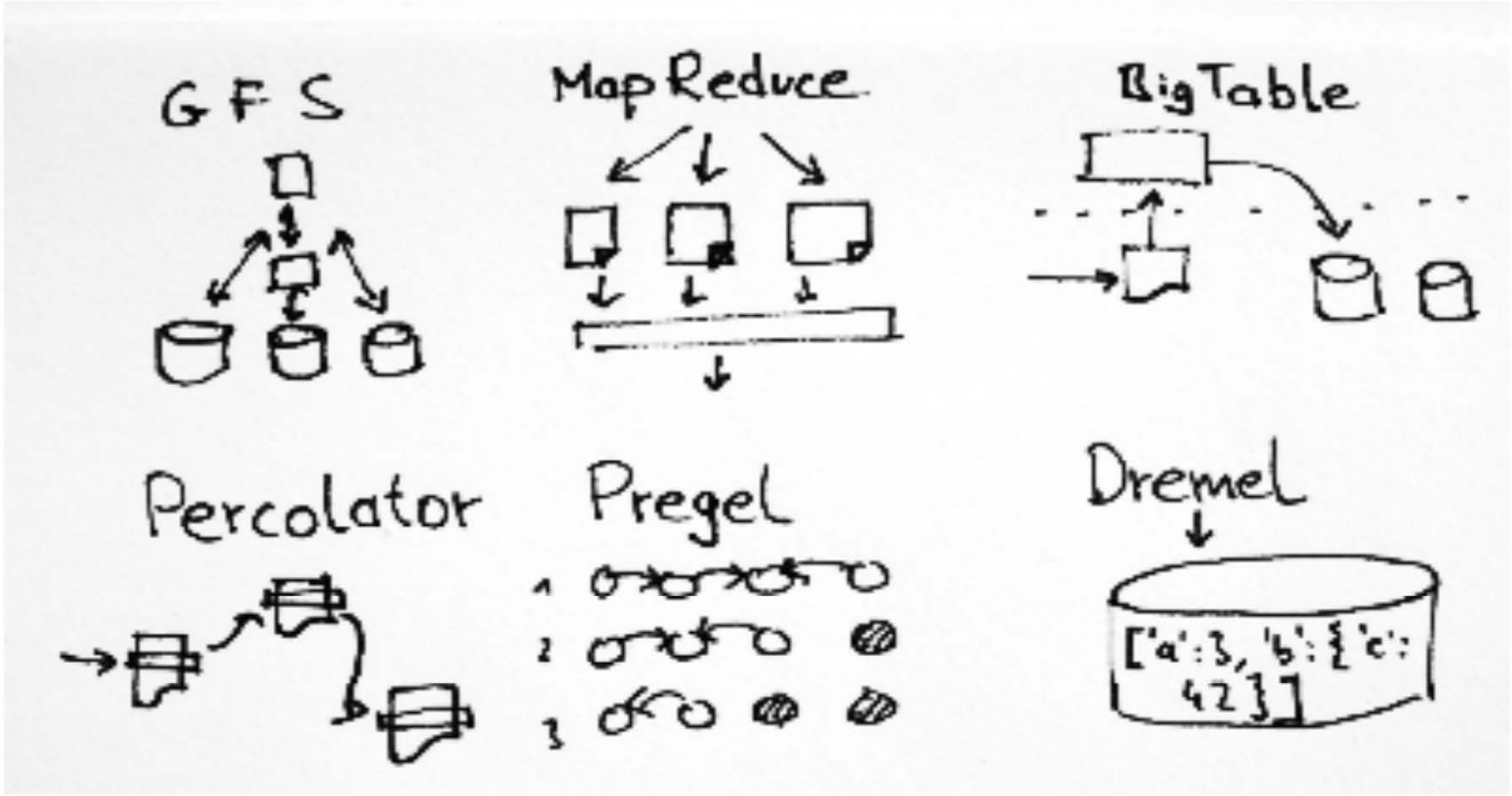

Volume
Terabytes
Records/Arch
Transactions
Tables, Files
Velocity
Batch
Real/near-time
Processes
Streams
Variety
Structured
Unstructured
Multi-factor
Probabilistic
5 Vs of
Big Data
Value
Statistical
Events
Correlations
Hypothetical
Trustworthiness
Authenticity
Origin, Reputation
Availability
Accountability
Veracity

5 V’s via [V-PEC-T]

- Business Factors
  - ‘Veracity’ - The What
  - ‘Value’ - The Why
- Technical Domain (Policies, Events, Content)
  - Volume, Velocity, Variety

Source: Ashwani Roy, Charles Cai - QCON London 2013 - http://bit.ly/1f2Pdf9

Big Data Industry History: Google's Papers



Source: Ashwani Roy, Charles Cai - QCON London 2013 - http://bit.ly/1f2Pdf9

Google's Big Data Papers: 2003 – 2006

GFS – Google File System

- 2003
- Distributed file system
- 3 x copies
- Commodity machines
- Colossus (2012)

MapReduce

- 2004
- Input → Map → Partition → Compare → Shuffle → Sort → Reduce → Output

BigTable

- 2006
- Distributed Key-Value column-family based database

Source: Ashwani Roy, Charles Cai - QCON London 2013 - http://bit.ly/1f2Pdf9

Google's Big Data Papers 2: 2010 - now

Percolator

- 2010
- Incremental update/ compute
- built on BigTable
- Adds transactions, locks, notifications
- SPFs: "Stream Processing Frameworks" + underlying database

Dremel

- 2010
- Online analytics and visualization
- SQL like language for structured data
- Each row is JSON object – in protobuf format
- Column based
- **Spanner (2012), BigQuery, F1**

Pregel

- 2010
- Scalable graph computing
- Worker threads → nodes → parallel "superstep" → messages → nodes → Aggregator/Combiners (global statistics)
- **PageRank**, shortest path, bipartite matching

YAHOO! S4 distributed stream computing platform — Apache incubator

Storm — Distributed and fault-tolerant realtime computation

Apache Drill — Apache incubator

cloudera — Ask Bigger Questions — **Impala**

Hortonworks — **Tez/Stinger**

Apache Giraph — Apache incubator

TITAN

Microsoft Trinity

Batch

The needs of the system outweigh the needs of individual events and queries running in flight or active within the system

Incremental

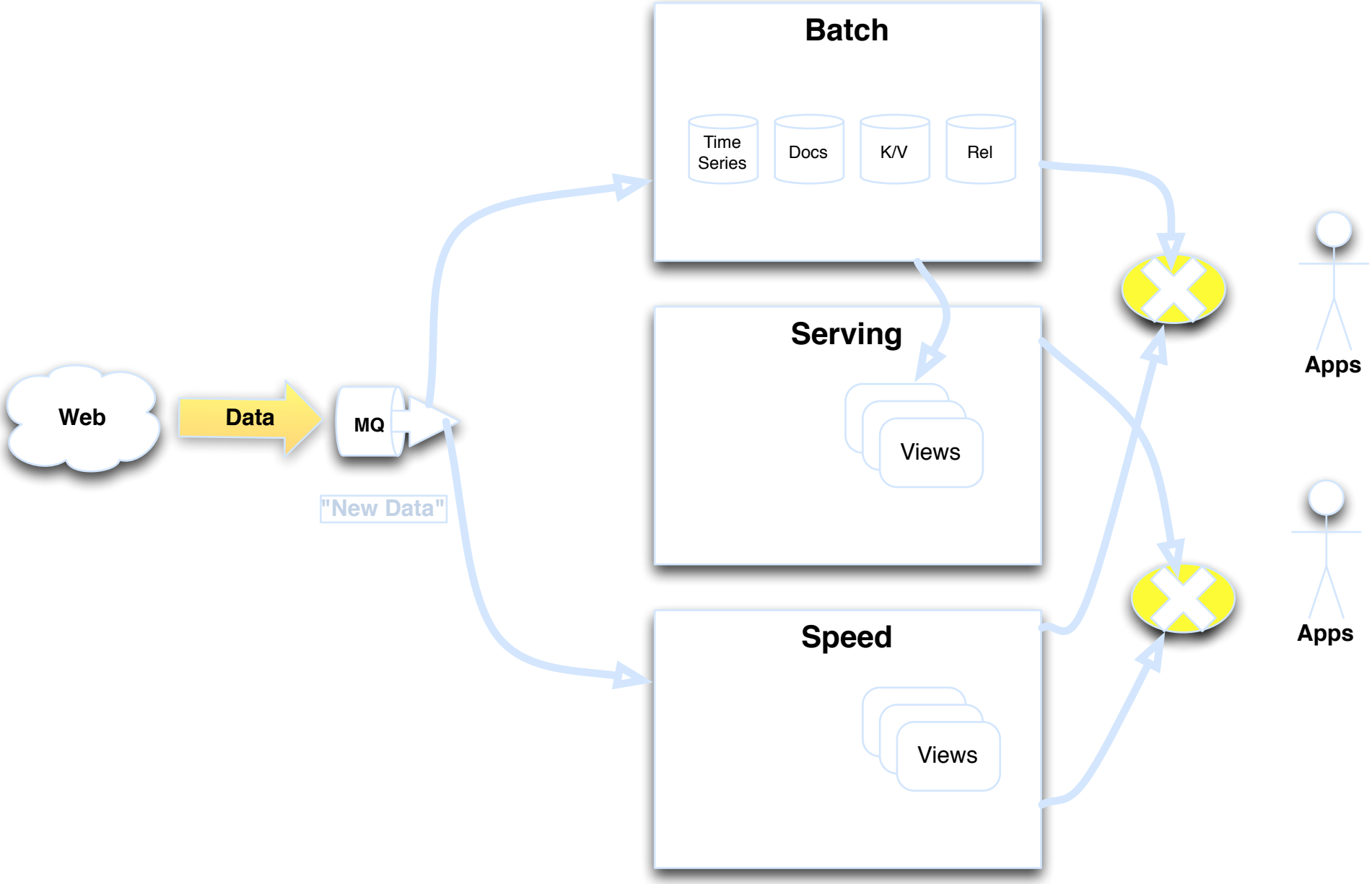
The needs of the individual event or query outweigh the needs of the aggregate events or queries in flight in the system

“Computing arbitrary functions on an arbitrary dataset in real-time is a daunting problem.”

- Nathan März

Lambda architecture is a twitter scale architecture. 5k msgs/sec inbound (tweets) on average (150k peak?) - <1k ‘small' data - Firehose outbound (broadcast problem, fairly easy to scale)

Lambda: http://bit.ly/Hs53Ur



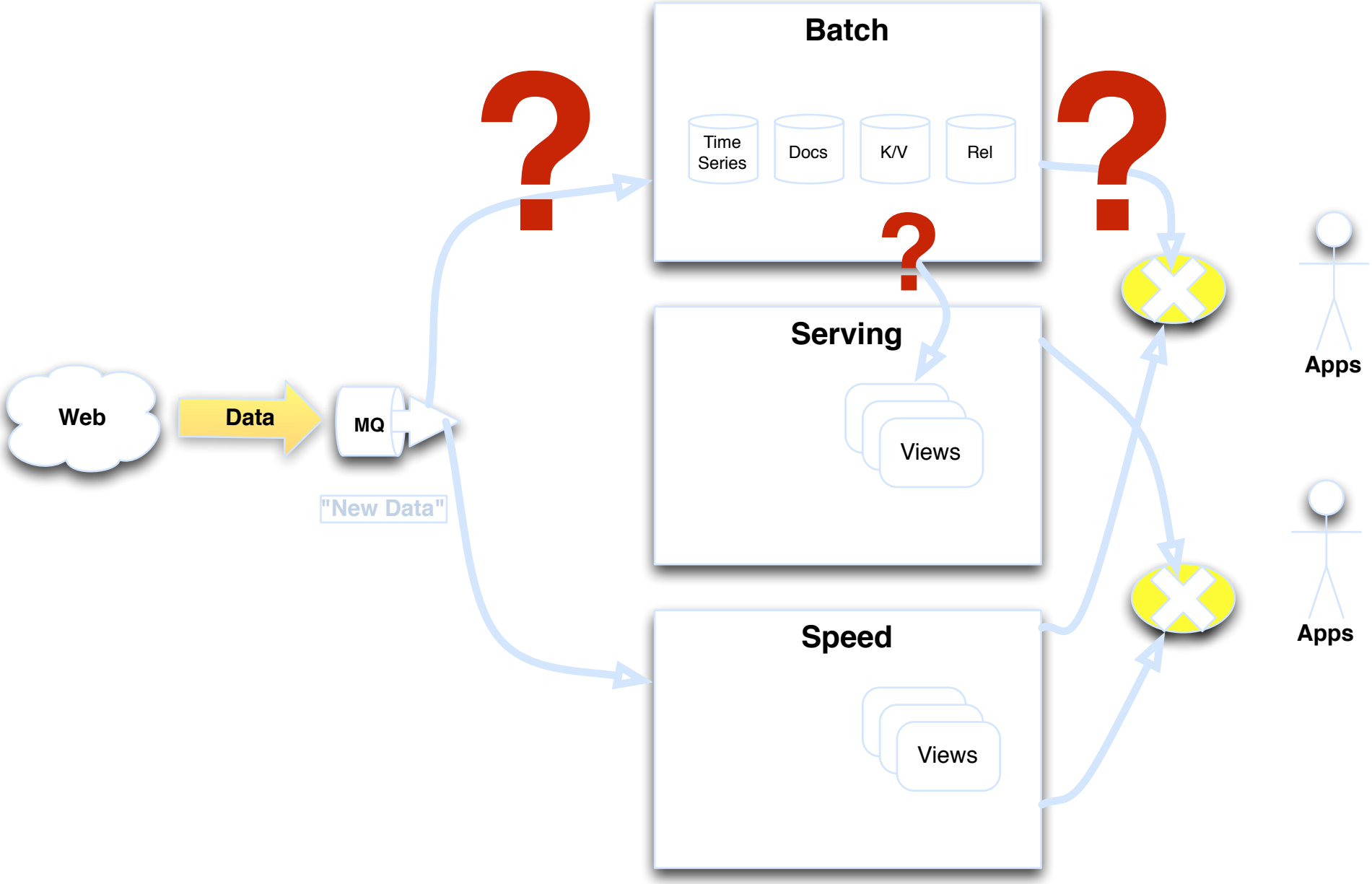
Lambda: A

All new data is sent to both the batch layer and the speed layer. In the batch layer, new data is appended to the master dataset. In the speed layer, the new data is consumed to do incremental updates of the realtime views.

Lambda: B

The master dataset is an immutable, append-only set of data. The master dataset only contains the **rawest** information that is **not derived** from any other information you have.

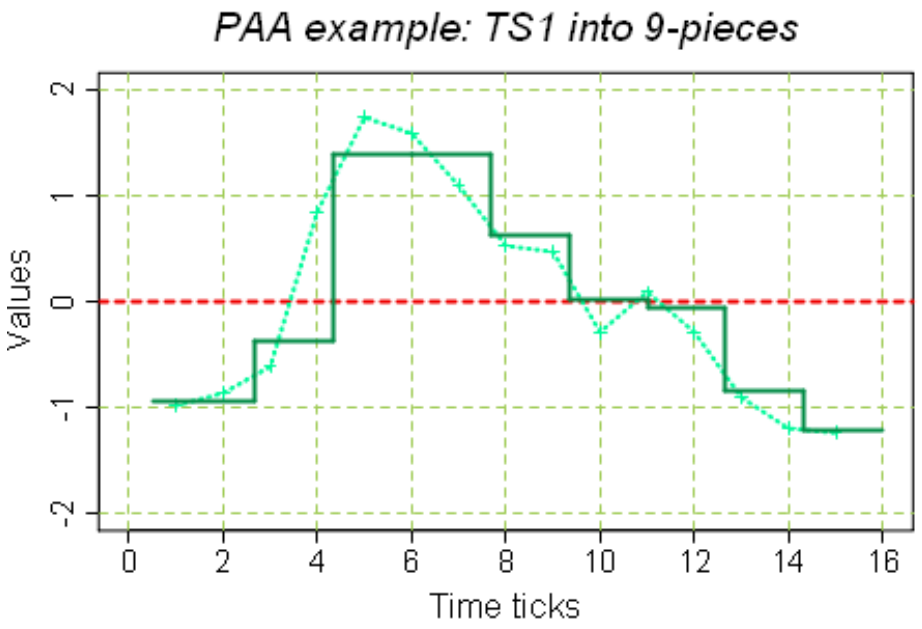
Lambda: http://bit.ly/Hs53Ur



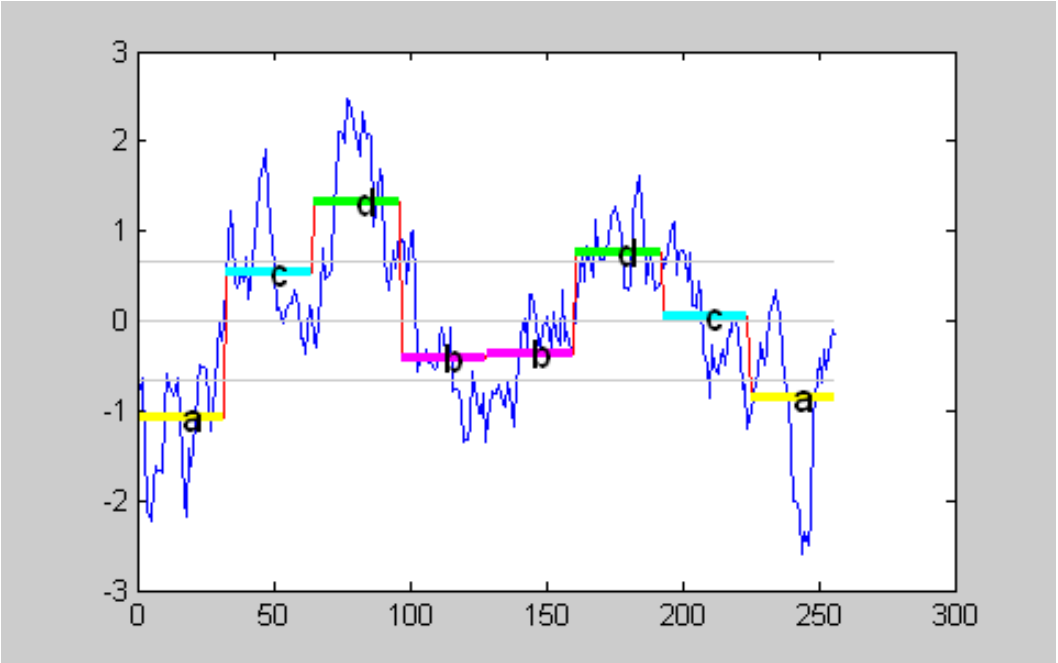
Enrich, Transform, Store
Extract, Transform, Load

- From A: “rawest … not derived"
  - In many environments it may be preferable to normalise data for later ease of retrieval (eg: Dremel, strongly typed nested records) to support scalable ad hoc query.
  - Derivation allows other forms of efficient retrieval eg: using **SAX** - Symbolic Aggregate Approximation, **PAA** - Piecewise Aggregate

SAX & PAA



Piecewise Aggregate Approximation



Symbolic Aggregate Approximation

1sc -> 1mn -> 1hr -> 1dy -> 1wk -> 1mh -> 1yr

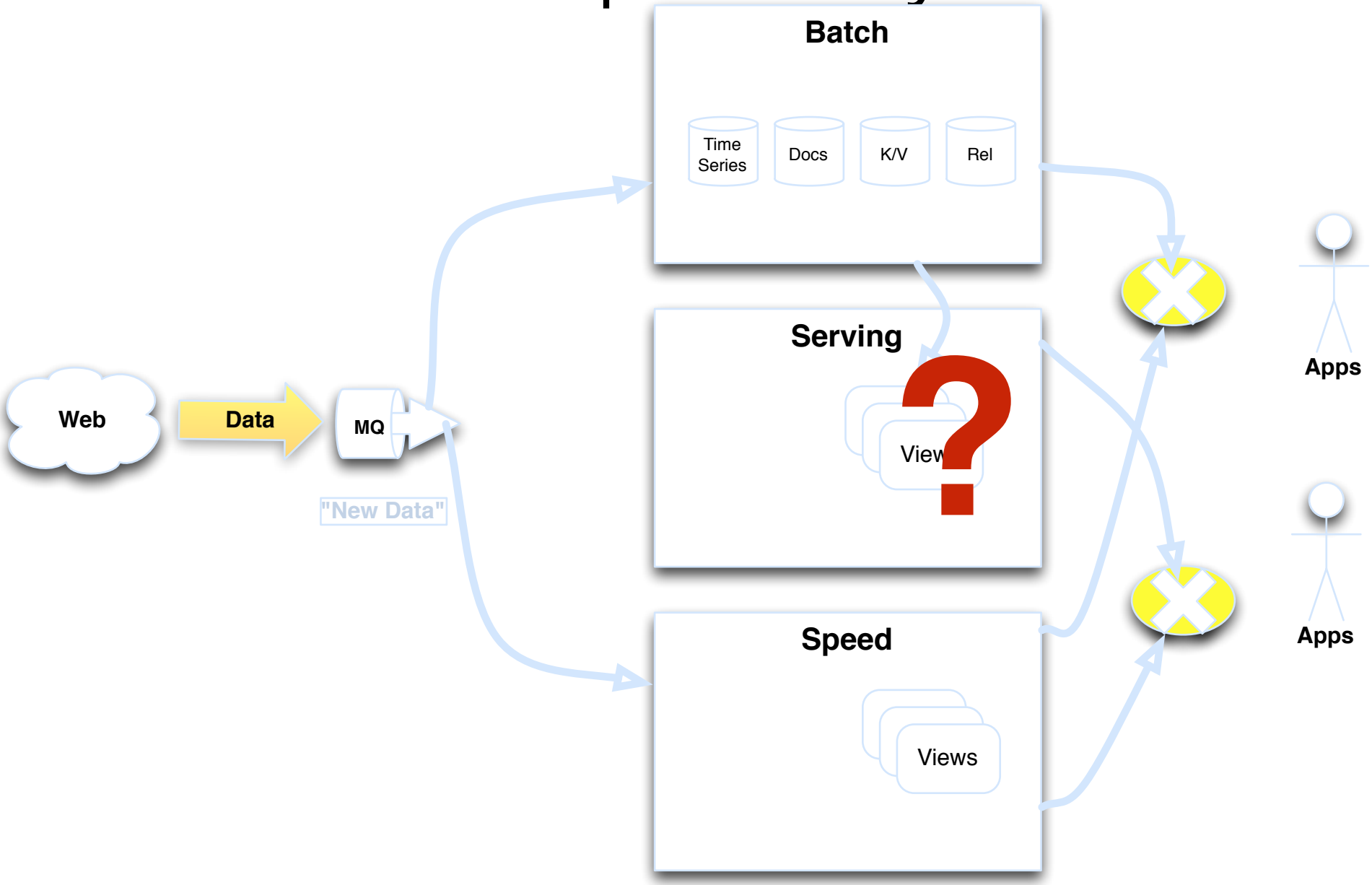
Lambda: C

The batch layer **precomputes** query functions from scratch. The results of the batch layer are called **batch views**. The batch layer runs in a while(true) loop and continuously recomputes the batch views from scratch. **The strength of the batch layer is its ability to compute arbitrary functions on arbitrary data.** This gives it the power to support any application.

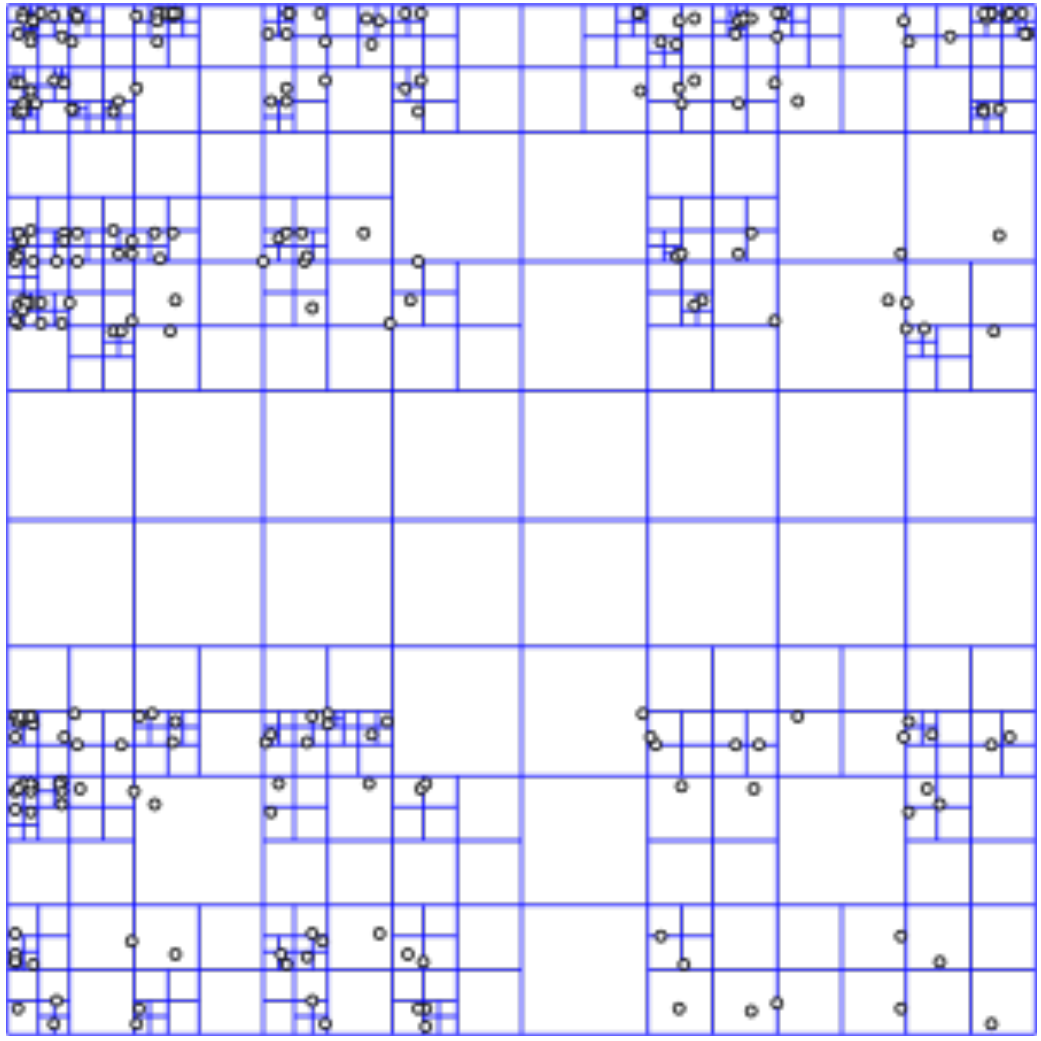
Lambda: D

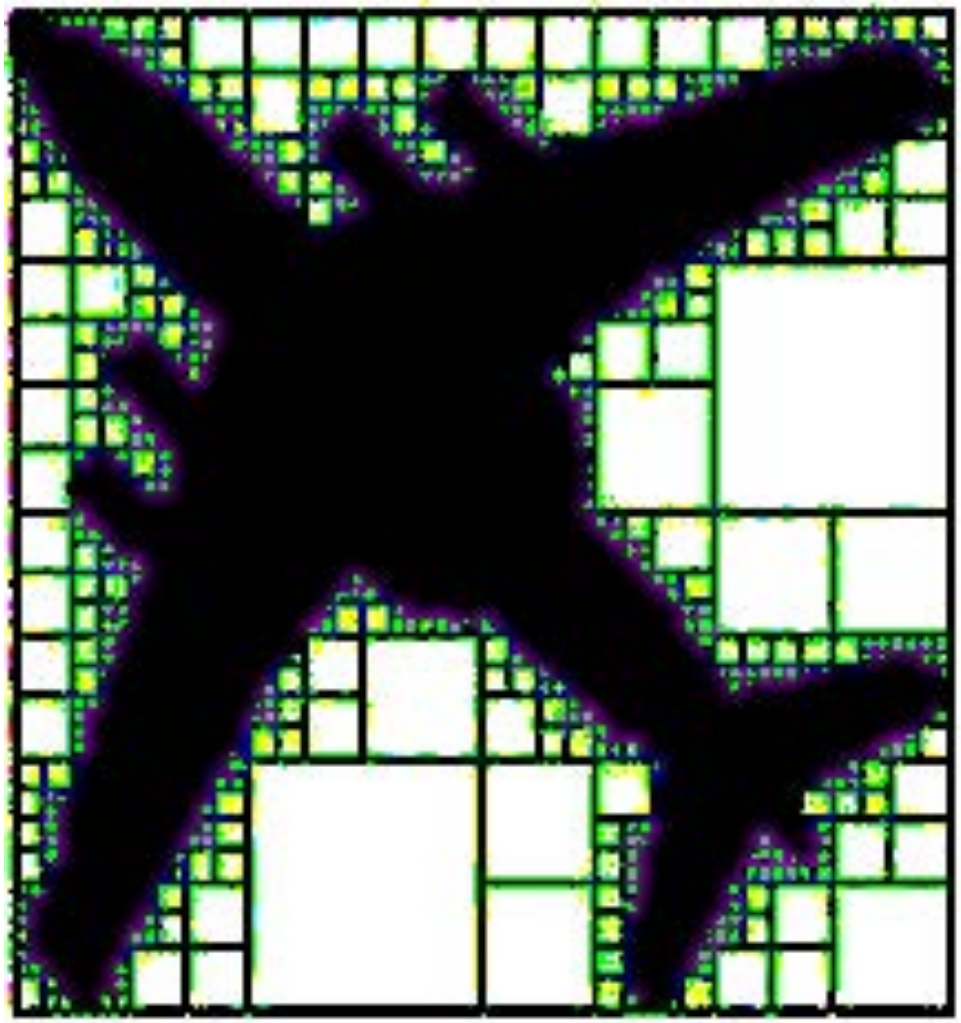
The serving layer indexes the batch views produced by the batch layer and makes it possible to get particular values out of a batch view very quickly. **The serving layer is a scalable database that swaps in new batch views as they're made available.** Because of the latency of the batch layer, the results available from the serving layer are always **out of date by a few hours**.

Lambda: http://bit.ly/Hs53Ur



Think ‘Statistical Compression'



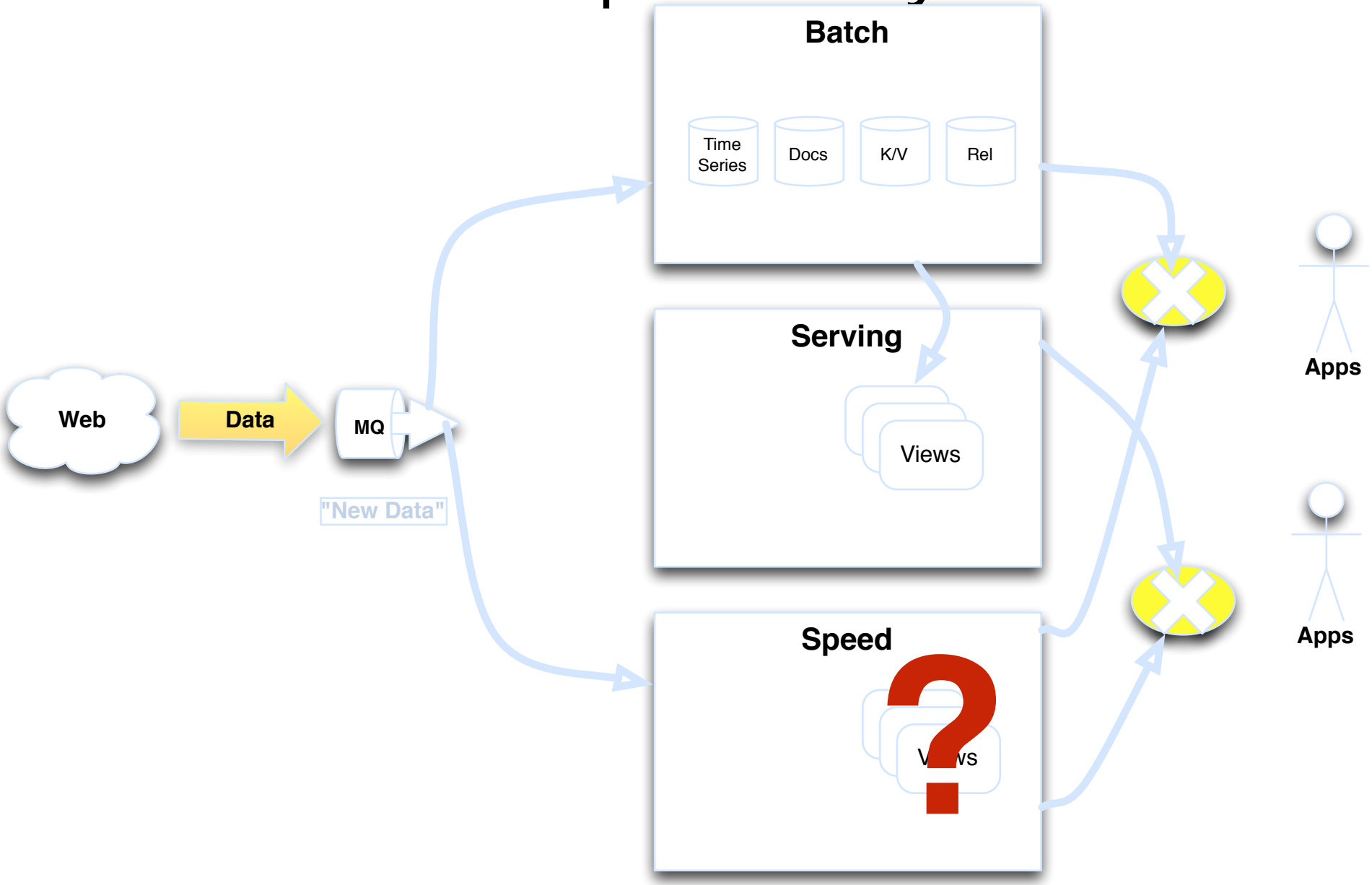


https://github.com/gornik/gorgeo - A geohash ES plugin

Lambda: E

The speed layer compensates for the high latency of updates to the serving layer. It uses fast incremental algorithms and read/write databases to produce realtime views that are always up to date. The speed layer only deals with recent data, because any data older than that has been absorbed into the batch layer and accounted for in the serving layer. **The speed layer is significantly more complex than the batch and serving layers, but that complexity is compensated by the fact that the realtime views can be continuously discarded as data makes its way through the batch and serving layers**. So, the potential negative impact of that complexity is greatly limited.

Lambda: http://bit.ly/Hs53Ur



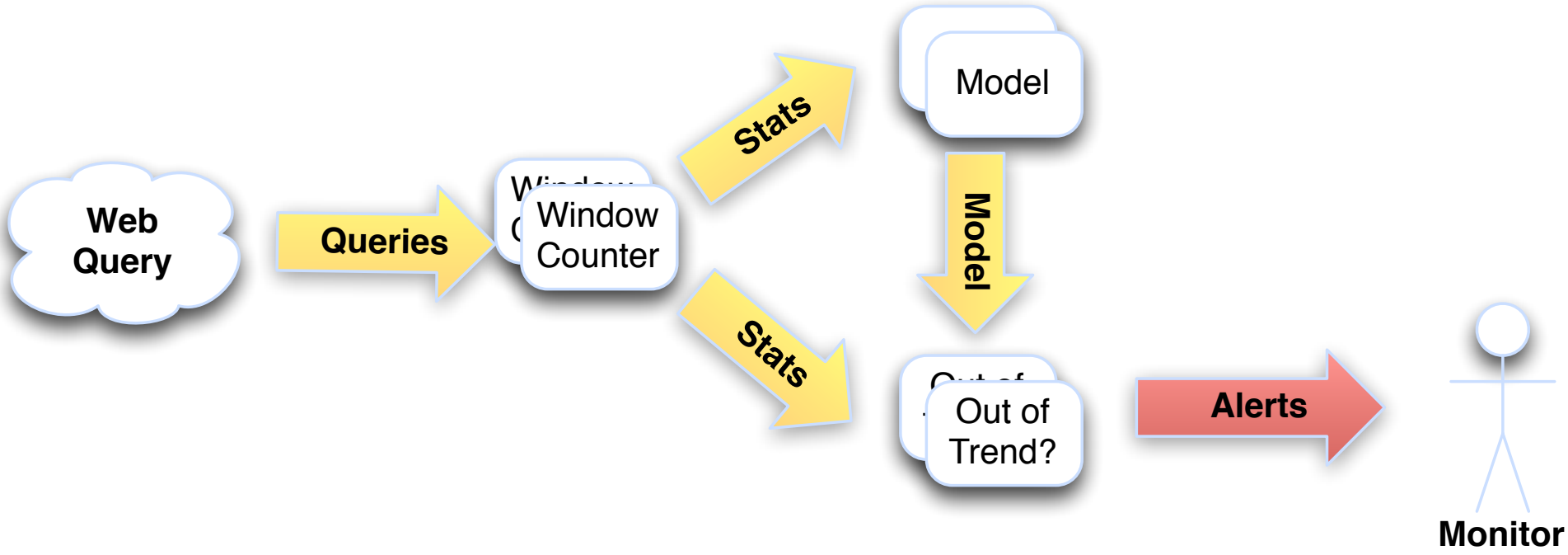
Use a DSP + CEP/ESP or ‘Scalable CEP'

- Storm/S4 + Esper/…
  - Embed a CEP/ESP within a Distributed Stream processing Engine
  - Use Drill for large scale ad hoc query [leverage nested records]

Lambda: F

Queries are resolved by getting results from both the batch and realtime views and **merging** them together.

Millwheel: http://bit.ly/1gWqNIC



Google's "Zeitgeist pipeline"

Lambda: Batch View

- **Precomputed** Queries are central to Complex Event Processing / Event Stream Processing architectures.

- Unfortunately, though, most DBMS's *still* offer only synchronous blocking RPC access to underlying data when asynchronous guaranteed delivery would be preferable for view construction leveraging CEP/ESP techniques.

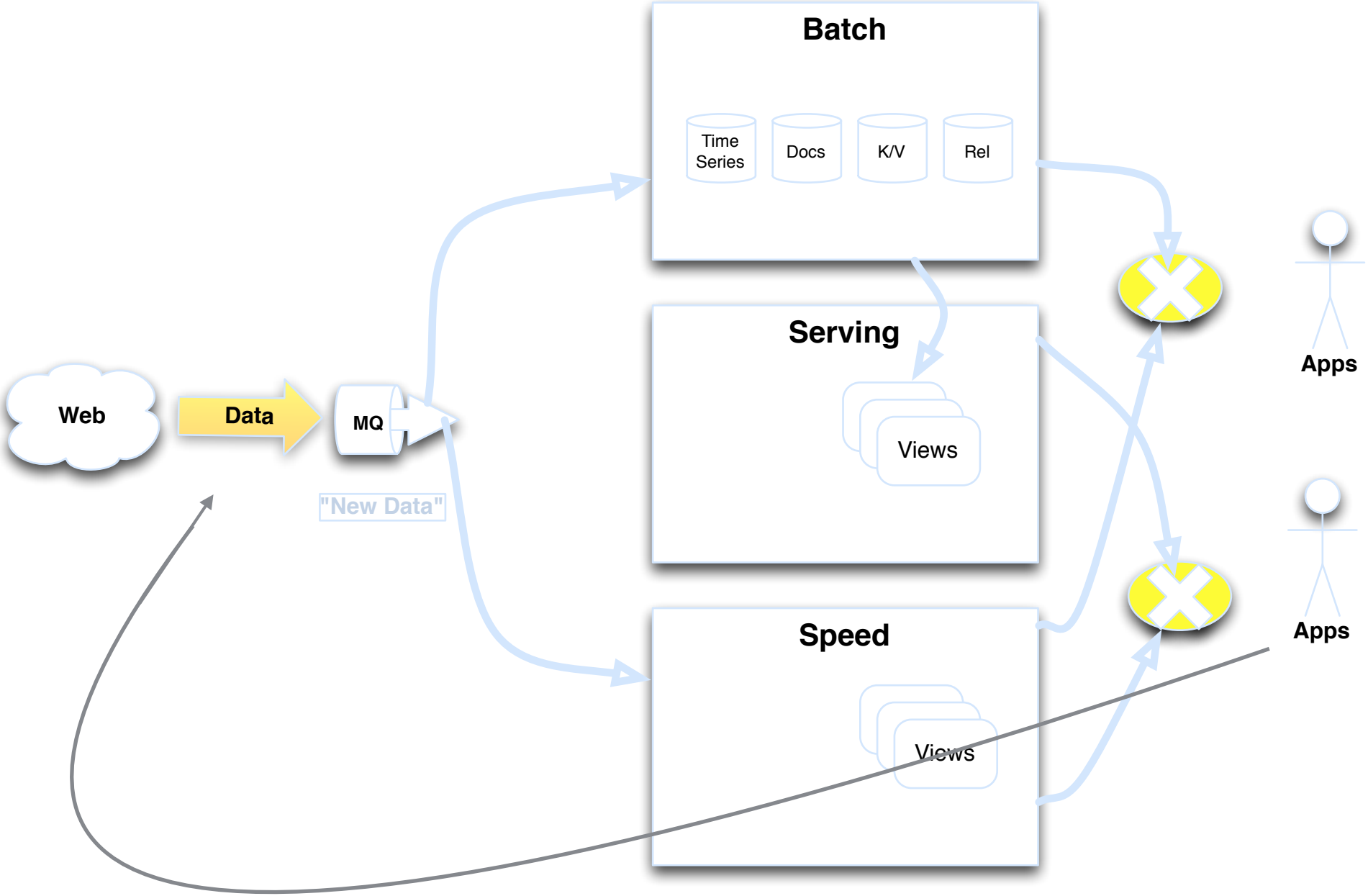
Lambda: Merging …

- Possibly one of the most difficult aspects of near real-time and historical data integration is combining flows sensibly.

- For example, is the order of interleaving across merge sources applied in a known deterministically recomputable order? If not, how can results be recomputed subsequently? Will data converge?

[cf: http://cs.brown.edu/research/aurora/hwang.icde05.ha.pdf]

Lambda: A start …



Lambda Architecture - An architectural pattern producing war stories is better than no **patterns** at all

Thanks.

Questions?

@darachennis