Datadog: A Real-Time Metrics Database for Trillions of Points/Day

Joel BARCIAUSKAS (https://twitter.com/JoelBarciauskas)
Director, Aggregation Metrics

SACON '20
### Trillions of points per day

| $10^4$ | Number of apps; 1,000’s hosts times 10’s containers |
| $10^3$ | Number of metrics emitted from each app/container |
| $10^0$ | 1 point a second per metric |
| $10^5$ | Seconds in a day (actually 86,400) |

$$10^4 \times 10^3 \times 10^5 = 10^{12}$$
Decreasing Infrastructure Lifecycle

Datacenter  Cloud/VM  Containers

Months/years  Seconds

λ
Increasing Granularity

- **Per User Device**
- **SLIs**
- **Application**
- **System**

Test duration by location & device

[SLI] Batch processing latency

Write requests (per second)
Tackling performance challenges

- Don't do it
- Do it, but don't do it again
- Do it less
- Do it later
- Do it when they're not looking
- Do it concurrently
- Do it cheaper

*From Craig Hanson and Pat Crain, and the performance engineering community - see [http://www.brendangregg.com/methodology.html](http://www.brendangregg.com/methodology.html)
Talk Plan

1. Our Architecture
2. Deep Dive On Our Datastores
3. Handling Synchronization
4. Approximation For Deeper Insights
5. Enabling Flexible Architecture
Talk Plan

1. Our Architecture
2. Deep Dive On Our Datastores
3. Handling Synchronization
4. Approximation For Deeper Insights
5. Enabling Flexible Architecture
Example Metrics Query 1

“What is the system load on instance i-xyz across the last 30 minutes”
A Time Series

<table>
<thead>
<tr>
<th>metric</th>
<th>system.load.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp</td>
<td>1526382440</td>
</tr>
<tr>
<td>value</td>
<td>0.92</td>
</tr>
<tr>
<td>tags</td>
<td>host:i-xyz, env:dev,...</td>
</tr>
</tbody>
</table>
Tags for all the dimensions

Host / container: system metrics by host
Application: internal cache hit rates, timers by module
Service: hits, latencies or errors/s by path and/or response code
Business: # of orders processed, $'s per second by customer ID
Pipeline Architecture

Metrics sources

Slack/Email/ PagerDuty etc

Customer Browser

Intake

Data Stores

Monitors and Alerts

Query System

Web frontend & APIs
Caching timeseries data

- Metrics sources
- Slack/Email/PagerDuty etc
- Customer Browser

Intake

- Monitors and Alerts
- Web frontend & APIs
- Query System
- Data Stores
- Query Cache
Performance mantras

• Don't do it
• **Do it, but don't do it again - cache as much as you can**
• Do it less
• Do it later
• Do it when they're not looking
• Do it concurrently
• Do it cheaper
Zooming in

Metrics sources

Slack/Email/PagerDuty etc

Customer Browser

Customer

Intake

Data Stores

Query System

Query Cache

Monitors and Alerts

Web frontend & APIs
Kafka for Independent Storage Systems

Intake

Incoming Data

Kafka Points

Kafka Tag Sets

Store 1

S3 Writer

Query System

Tag Index

Tag Describer

S3

Outgoing Data
Performance mantras

• Don't do it
• Do it, but don't do it again - cache as much as you can
• Do it less
• **Do it later - minimize upfront processing**
• Do it when they're not looking
• Do it concurrently
• Do it cheaper
Scaling through Kafka

**Partition** by customer, metric, tag set

- **Isolate** by customer
- **Scale concurrently** by metric
- *Building something more dynamic*

---

**Diagram:**

- **Incoming Data**
- **Intake**
  - Kafka partition:0
  - Kafka partition:1
  - Kafka partition:2
  - Kafka partition:3
- **Stores:**
  - Store 1
  - Store 2
Performance mantras

• Don't do it
• Do it, but don't do it again - cache as much as you can
• Do it less
• Do it later - minimize upfront processing
• Do it when they're not looking
• **Do it concurrently - spread data across independent, scalable data stores**
• Do it cheaper
Talk Plan

1. Our Architecture
2. Deep Dive On Our Datastores
3. Handling Synchronization
4. Approximation For Deeper Insights
5. Enabling Flexible Architecture
Trillions of points per day

| $10^4$ | Number of apps; 1,000’s hosts times 10’s containers |
| $10^3$ | Number of metrics emitted from each app/container |
| $10^0$ | 1 point a second per metric |
| $10^5$ | Seconds in a day (actually 86,400) |

$10^4 \times 10^3 \times 10^5 = 10^{12}$
### Per Customer Volume Ballparking

<table>
<thead>
<tr>
<th>(10^4)</th>
<th>Number of apps; 1,000’s hosts times 10’s containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10^3)</td>
<td>Number of metrics emitted from each app/container</td>
</tr>
<tr>
<td>(10^0)</td>
<td>1 point a second per metric</td>
</tr>
<tr>
<td>(10^5)</td>
<td>Seconds in a day (actually 86,400)</td>
</tr>
<tr>
<td>(10^1)</td>
<td>Bytes/point (8 byte float, amortized tags)</td>
</tr>
</tbody>
</table>

\[ = 10^{13} \] 10 Terabytes a Day For One Customer
## Cloud Storage Characteristics

<table>
<thead>
<tr>
<th>Type</th>
<th>Max Capacity</th>
<th>Bandwidth</th>
<th>Latency</th>
<th>Cost/TB for 1 month</th>
<th>Volatility</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM(^1)</td>
<td>4 TB</td>
<td>80 GB/s</td>
<td>0.08 us</td>
<td>$1,000</td>
<td>Instance Reboot</td>
</tr>
<tr>
<td>SSD(^2)</td>
<td>60 TB</td>
<td>12 GB/s</td>
<td>1 us</td>
<td>$60</td>
<td>Instance Failures</td>
</tr>
<tr>
<td>EBS io1</td>
<td>432 TB</td>
<td>12 GB/s</td>
<td>40 us</td>
<td>$400</td>
<td>Data Center Failures</td>
</tr>
<tr>
<td>S3</td>
<td>Infinite</td>
<td>12 GB/s(^3)</td>
<td>100+ ms</td>
<td>$21(^4)</td>
<td>11 nines durability</td>
</tr>
<tr>
<td>Glacier</td>
<td>Infinite</td>
<td>12 GB/s(^3)</td>
<td>hours</td>
<td>$4(^4)</td>
<td>11 nines durability</td>
</tr>
</tbody>
</table>

1. X1e.32xlarge, 3 year non convertible, no upfront reserved instance
2. i3en.24xlarge, 3 year non convertible, no upfront reserved instance
3. Assumes can highly parallelize to load network card of 100Gbps instance type. Likely does not scale out.
4. Storage Cost only
Volume Math

- 80 x1e.32xlarge DRAM
- $300,000 to store for a month
- This is with no indexes or overhead
- And people want to query much more than a month.
## Cloud Storage Characteristics

<table>
<thead>
<tr>
<th>Type</th>
<th>Max Capacity</th>
<th>Bandwidth</th>
<th>Latency</th>
<th>Cost/TB for 1 month</th>
<th>Volatility</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM(^1)</td>
<td>4 TB</td>
<td>80 GB/s</td>
<td>0.08 us</td>
<td>$1,000</td>
<td>Instance Reboot</td>
</tr>
<tr>
<td>SSD(^2)</td>
<td>60 TB</td>
<td>12 GB/s</td>
<td>1 us</td>
<td>$60</td>
<td>Instance Failures</td>
</tr>
<tr>
<td>EBS io1</td>
<td>432 TB</td>
<td>12 GB/s</td>
<td>40 us</td>
<td>$400</td>
<td>Data Center Failures</td>
</tr>
<tr>
<td>S3</td>
<td>Infinite</td>
<td>12 GB/s(^3)</td>
<td>100+ ms</td>
<td>$21(^4)</td>
<td>11 nines durability</td>
</tr>
<tr>
<td>Glacier</td>
<td>Infinite</td>
<td>12 GB/s(^3)</td>
<td>hours</td>
<td>$4(^4)</td>
<td>11 nines durability</td>
</tr>
</tbody>
</table>

1. X1e.32xlarge, 3 year non convertible, no upfront reserved instance
2. i3en.24xlarge, 3 year non convertible, no upfront reserved instance
3. Assumes can highly parallelize to load network card of 100Gbps instance type. Likely does not scale out.
4. Storage Cost only
## Cloud Storage Characteristics

<table>
<thead>
<tr>
<th>Type</th>
<th>Max Capacity</th>
<th>Bandwidth</th>
<th>Latency</th>
<th>Cost/TB for 1 month</th>
<th>Volatility</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM¹</td>
<td>4 TB</td>
<td>80 GB/s</td>
<td>0.08 us</td>
<td>$1,000</td>
<td>Instance Reboot</td>
</tr>
<tr>
<td>SSD²</td>
<td>60 TB</td>
<td>12 GB/s</td>
<td>1 us</td>
<td>$60</td>
<td>Instance Failures</td>
</tr>
<tr>
<td>EBS io1</td>
<td>432 TB</td>
<td>12 GB/s</td>
<td>40 us</td>
<td>$400</td>
<td>Data Center Failures</td>
</tr>
<tr>
<td>S3</td>
<td>Infinite</td>
<td>12 GB/s³</td>
<td>100+ ms</td>
<td>$21^4</td>
<td>11 nines durability</td>
</tr>
<tr>
<td>Glacier</td>
<td>Infinite</td>
<td>12 GB/s³</td>
<td>hours</td>
<td>$4^4</td>
<td>11 nines durability</td>
</tr>
</tbody>
</table>

1. X1e.32xlarge, 3 year non convertible, no upfront reserved instance
2. i3en.24xlarge, 3 year non convertible, no upfront reserved instance
3. Assumebes can highly parallelize to load network card of 100Gbps instance type. Likely does not scale out.
4. Storage Cost only
# Queries We Need to Support

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIBE TAGS</td>
<td>What tags are queryable for this metric?</td>
</tr>
<tr>
<td>TAG INDEX</td>
<td>Given a time series id, what tags were used?</td>
</tr>
<tr>
<td>TAG INVERTED INDEX</td>
<td>Given some tags and a time range, what were the time series ingested?</td>
</tr>
<tr>
<td>POINT STORE</td>
<td>What are the values of a time series between two times?</td>
</tr>
</tbody>
</table>
Performance mantras

• Don't do it
• Do it, but don't do it again - query caching
• **Do it less** - only index what you need
• Do it later - minimize upfront processing
• Do it when they're not looking
• Do it concurrently - use independent horizontally scalable data stores
• Do it cheaper
## Hybrid Data Storage

<table>
<thead>
<tr>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIBE TAGS</td>
</tr>
<tr>
<td>TAG INDEX</td>
</tr>
<tr>
<td>TAG INVERTED INDEX</td>
</tr>
<tr>
<td>POINT STORE</td>
</tr>
<tr>
<td>QUERY RESULTS</td>
</tr>
</tbody>
</table>
# Hybrid Data Storage

<table>
<thead>
<tr>
<th>System</th>
<th>Type</th>
<th>Persistence</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIBE TAGS</td>
<td>Local SSD</td>
<td>Years</td>
</tr>
<tr>
<td>TAG INDEX</td>
<td>DRAM</td>
<td>Cache (Hours)</td>
</tr>
<tr>
<td></td>
<td>Local SSD</td>
<td>Years</td>
</tr>
<tr>
<td>TAG INVERTED INDEX</td>
<td>DRAM</td>
<td>Hours</td>
</tr>
<tr>
<td></td>
<td>On SSD</td>
<td>Days</td>
</tr>
<tr>
<td></td>
<td>S3</td>
<td>Years</td>
</tr>
<tr>
<td>POINT STORE</td>
<td>DRAM</td>
<td>Hours</td>
</tr>
<tr>
<td></td>
<td>Local SSD</td>
<td>Days</td>
</tr>
<tr>
<td></td>
<td>S3</td>
<td>Years</td>
</tr>
<tr>
<td>QUERY RESULTS</td>
<td>DRAM</td>
<td>Cache (Days)</td>
</tr>
</tbody>
</table>
# Hybrid Data Storage

<table>
<thead>
<tr>
<th>System</th>
<th>Type</th>
<th>Persistence</th>
<th>Technology</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIBE TAGS</td>
<td>Local SSD</td>
<td>Years</td>
<td>LevelDB</td>
<td>High performing single node k,v</td>
</tr>
<tr>
<td>TAG INDEX</td>
<td>DRAM</td>
<td>Cache (Hours)</td>
<td>Redis</td>
<td>Very high performance, in memory k,v</td>
</tr>
<tr>
<td></td>
<td>Local SSD</td>
<td>Years</td>
<td>Cassandra</td>
<td>Horizontal scaling, persistent k,v</td>
</tr>
<tr>
<td>TAG INVERTED INDEX</td>
<td>DRAM</td>
<td>Hours</td>
<td>In house</td>
<td>Very customized index data structures</td>
</tr>
<tr>
<td></td>
<td>On SSD</td>
<td>Days</td>
<td>RocksDB + SQLite</td>
<td>Rich and flexible queries</td>
</tr>
<tr>
<td></td>
<td>S3</td>
<td>Years</td>
<td>Parquet</td>
<td>Flexible Schema over time</td>
</tr>
<tr>
<td>POINT STORE</td>
<td>DRAM</td>
<td>Hours</td>
<td>In house</td>
<td>Very customized index data structures</td>
</tr>
<tr>
<td></td>
<td>Local SSD</td>
<td>Days</td>
<td>In house</td>
<td>Very customized index data structures</td>
</tr>
<tr>
<td></td>
<td>S3</td>
<td>Years</td>
<td>Parquet</td>
<td>Flexible Schema over time</td>
</tr>
<tr>
<td>QUERY RESULTS</td>
<td>DRAM</td>
<td>Cache (Days)</td>
<td>Redis</td>
<td>Very high performance, in memory k,v</td>
</tr>
</tbody>
</table>
Performance mantras

- Don't do it
- Do it, but don't do it again - query caching
- Do it less - only index what you need
- Do it later - minimize upfront processing
- Do it when they're not looking
- Do it concurrently - use independent horizontally scalable data stores
- **Do it cheaper - match data latency requirements to cost**
Talk Plan

1. Our Architecture
2. Deep Dive On Our Datastores
3. **Handling Synchronization**
4. Approximation For Deeper Insights
5. Enabling Flexible Architecture
Alerts/Monitors Synchronization

• Required to prevent false positives
• Need all data for the evaluation time period is ready
Pipeline Architecture

Metrics sources

Slack/Email/PagerDuty etc

Customer Browser

Customer

Intake

Monitors and Alerts

Web frontend & APIs

Query System

Data Stores

Query Cache

Inject heartbeat here
Pipeline Architecture

- Customer Browser
- Metrics sources
- Slack/Email/PagerDuty etc

Intake
- Monitors and Alerts
- Web frontend & APIs
- Query System
- Data Stores
- Query Cache

Inject heartbeat here

And test it gets to here
Performance mantras

- Don't do it - build the minimal synchronization needed
- Do it, but don't do it again - query caching
- Do it less - only index what you need
- Do it later - minimize upfront processing
- Do it when they're not looking
- Do it concurrently - use independent horizontally scalable data stores
- Do it cheaper - match data latency requirements to cost
Talk Plan

1. Our Architecture
2. Deep Dive On Our Datastores
3. Handling Synchronization
4. Approximation For Deeper Insights
5. Enabling Flexible Architecture
Types of metrics

Counter, aggregate by sum
Ex: Requests, errors/s, total time spent (stopwatch)

Gauges, aggregate by last or avg
Ex: CPU/network/disk use, queue length
Aggregation for counters and gauges

<table>
<thead>
<tr>
<th></th>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
<th>$t_8$</th>
<th>$t_9$</th>
</tr>
</thead>
</table>

{0, 1, 0, 1, 0, 1, 0, 1, 0, 1}

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

{5, 5, 5, 5, 5, 5, 5, 5, 5, 5}

{0, 2, 4, 8, 16, 32, 64, 128, 256, 512}

Query output

Counters: {5, 40, 50, 1023}

Gauges (average): {0.5, 4, 5, 102.3}

Gauges (last): {1, 9, 5, 512}
Focus on outputs

These graphs are *both* aggregating 70k series
Output 20 to 2000 times less series than input
Pipeline Architecture

- Metrics sources
- Slack/Email/PagerDuty etc
- Customer Browser

Aggregation Points

- Intake
- Data Stores
- Monitors and Alerts
- Query System
- Web frontend & APIs
- Query Cache

Customer
Pipeline Architecture

Metrics sources

Slack/Email/PagerDuty etc

Customer Browser

Customer

Aggregation Points

Intake

Data Stores

Streaming Aggregator

Monitors and Alerts

Query System

Query Cache

Web frontend & APIs
Pipeline Architecture

Metrics sources
Slack/Email/PagerDuty etc
Customer Browser

Customer

Aggregation Points

Streaming Aggregator

Data Stores

Intake

Query System

Query Cache

Web frontend & APIs

Monitors and Alerts

No one's looking here!
Performance mantras

- Don't do it - build the minimal synchronization needed
- Do it, but don't do it again - query caching
- Do it less - only index what you need
- Do it later - minimize processing on path to persistence
- **Do it when they're not looking** - pre-aggregate
- Do it concurrently - use independent horizontally scalable data stores
- Do it cheaper - use hybrid data storage types and technologies
Distributions

Aggregate by percentile or SLO
(count of values above or below a threshold)

Ex: Latency, request size
Calculating distributions

<table>
<thead>
<tr>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
<th>$t_8$</th>
<th>$t_9$</th>
</tr>
</thead>
</table>

\{0, 1, 0, 1, 0, 1, 0, 1, 0, 1\}

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

\{5, 5, 5, 5, 5, 5, 5, 5, 5, 5\}

\{0, 2, 4, 8, 16, 32, 64, 128, 256, 512\}

\{0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 3, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 7, 8, 8, 9, 16, 32, 64, 128, 256, 512\}

$p_{50}$

$p_{90}$
Performance mantras

• Don't do it - build the minimal synchronization needed
• Do it, but don't do it again - query caching
• Do it less - only index what you need
• Do it later - minimize upfront processing
• Do it when they're not looking
• Do it concurrently - use independent horizontally scalable data stores
• Do it cheaper again?
Tradeoffs

Engineering triangle - fast, good or cheap

What's the universe of valid values? (inputs)

What are common queries? (outputs)
Sketches

Data structures designed for operating on streams of data

- Examine each item a limited number of times (ideally once)
- Limited memory usage (logarithmic to the size of the stream, or fixed)
You may know these sketches

HyperLogLog

- Cardinality / unique count estimation
- Used in Redis PFADD, PFCOUNT, PFMERGE

Others: Bloom filters (also for set membership), frequency sketches (top-N lists)
Approximation for distribution metrics

What's important for approximating distribution metrics?

- Good: accurate
- Fast: quick insertion & queries
- Cheap: bounded-size storage
Approximating a distribution
Bucketed histograms

Basic example from OpenMetrics / Prometheus

```plaintext
# HELP http_request_duration_seconds A histogram of the request duration.
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.05"} 24054
http_request_duration_seconds_bucket{le="0.1"} 33444
http_request_duration_seconds_bucket{le="0.2"} 100392
http_request_duration_seconds_bucket{le="0.5"} 129389
http_request_duration_seconds_bucket{le="1"} 133988
http_request_duration_seconds_bucket{le="+Inf"} 144320
http_request_duration_seconds_sum 53423
http_request_duration_seconds_count 144320
```
Bucketed histograms

Basic example from OpenMetrics / Prometheus

<table>
<thead>
<tr>
<th>Time spent</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 0.05 (50ms)</td>
<td>24054</td>
</tr>
<tr>
<td>&lt;= 0.1 (100ms)</td>
<td>33444</td>
</tr>
<tr>
<td>&lt;= 0.2 (200ms)</td>
<td>100392</td>
</tr>
<tr>
<td>&lt;= 0.5 (500ms)</td>
<td>129389</td>
</tr>
<tr>
<td>&lt;= 1s</td>
<td>133988</td>
</tr>
<tr>
<td>&gt; 1s</td>
<td>144320</td>
</tr>
</tbody>
</table>

median = ~158ms (using linear interpolation)

p99 = ?!
Bucketed histograms

Basic example from OpenMetrics / Prometheus

<table>
<thead>
<tr>
<th>Time spent</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 0.05 (50ms)</td>
<td>24054</td>
</tr>
<tr>
<td>&lt;= 0.1 (100ms)</td>
<td>33444</td>
</tr>
<tr>
<td>&lt;= 0.2 (200ms)</td>
<td>100392</td>
</tr>
<tr>
<td>&lt;= 0.5 (500ms)</td>
<td>129389</td>
</tr>
<tr>
<td>&lt;= 1s</td>
<td>133988</td>
</tr>
<tr>
<td>&gt; 1s</td>
<td>144320</td>
</tr>
</tbody>
</table>

median = ~158ms (using linear interpolation)
Rank and relative error
Rank and relative error
Good: relative error

Relative error bounds mean we can answer this: Yes, within 99% of requests are $\leq 500\text{ms} \pm 1\%$

Otherwise stated: 99% of requests are guaranteed $\leq 505\text{ms}$
Cheap: fixed storage size

With certain distributions, we may reach the maximum number of buckets (in our case, 4000)

- Roll up lower buckets - lower percentiles are generally not as interesting!*

*Note that we've yet to find a data set that actually needs this in practice
Fast: insertion & query

Each insertion is just two operations - find the bucket, increase the count (sometimes there's an allocation)

Queries look at the fixed number of buckets
DDSketch

DDSketch (Distributed Distribution Sketch) is open source

- Presented at VLDB2019 in August
- Open-source versions in several languages
  
  Python: [github.com/DataDog/sketches-py](https://github.com/DataDog/sketches-py)

  Java: [github.com/DataDog/sketches-java](https://github.com/DataDog/sketches-java)

  Go: [github.com/DataDog/sketches-go](https://github.com/DataDog/sketches-go)
Performance mantras

- Don't do it - build the minimal synchronization needed
- Do it, but don't do it again - query caching
- Do it less - only index what you need
- Do it later - minimize upfront processing
- Do it when they're not looking
- Do it concurrently - use independent horizontally scalable data stores
- Do it cheaper - leverage approximation
Talk Plan

1. Our Architecture
2. Deep Dive On Our Datastores
3. Handling Synchronization
4. Approximation For Deeper Insights
5. Enabling Flexible Architecture
Commutativity

"a binary operation is **commutative** if *changing the order* of the operands does not change the result"

Why is this important?
Commutativity

"a binary operation is **commutative** if *changing the order* of the operands does not change the result"

Why is this important?

Distribute aggregation work throughout the pipeline
Pipeline Architecture

- Customer
- Browser
- Metrics sources
- Slack/Email/PagerDuty etc
- Data Stores
- Query System
- Web frontend & APIs
- Monitors and Alerts
- Query Cache
- Streaming Aggregator

Aggregation Points
Performance mantras

• Don't do it - build the minimal synchronization needed
• Do it, but don't do it again - query caching
• Do it less - only index what you need
• Do it later - minimize upfront processing
• Do it when they're not looking - pre-aggregate
• Do it concurrently - use independent horizontally scalable data stores
• Do it cheaper - use hybrid data storage types and technologies and leverage approximation
• Don't do it - build the bare minimal synchronization needed
• Do it, but don't do it again - cache as much as you can
• Do it less - only index what you need
• Do it later - minimize upfront processing
• Do it when they're not looking - pre-aggregate where is cost effective
• Do it concurrently - use independent horizontally scalable data stores
• Do it cheaper - use hybrid data storage types and technologies and leverage approximation

Do exactly as much work as needed, and no more
Thank You
We’re living in a new era of constant sabotage, misinformation, and fear, in which everyone is a target, and you’re often the collateral damage in a growing conflict among states. From crippling infrastructure to sowing discord and doubt, cyber is now the weapon of choice for democracies, dictators, and terrorists.

David Sanger explains how the rise of cyberweapons has transformed geopolitics like nothing since the invention of the atomic bomb. Moving from the White House Situation Room to the dens of Chinese, Russian, North Korean, and Iranian hackers to the boardrooms of Silicon Valley, David reveals a world coming face-to-face with the perils of technological revolution—a conflict that the United States helped start when it began using cyberweapons against Iranian nuclear plants and North Korean missile launches. But now we find ourselves in a conflict we’re uncertain how to control, as our adversaries exploit vulnerabilities in our hyperconnected nation and we struggle to figure out how to deter these complex, short-of-war attacks.