Efficient Time Series

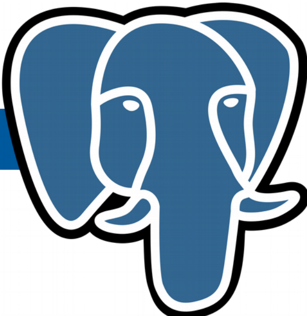
with PostgreSQL

Steve Simpson
steve@smpsn.net

FOSDEM PGDay 2018

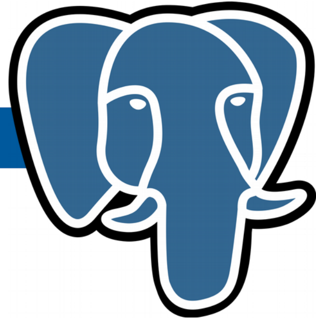
Abbaye de Abdij van
Leffe

Overview



Overview

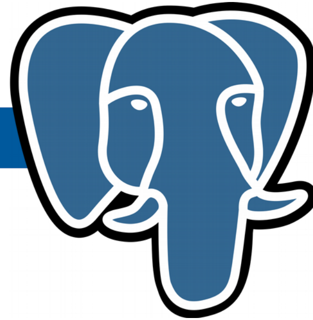
Background



Overview

Background

Complexity

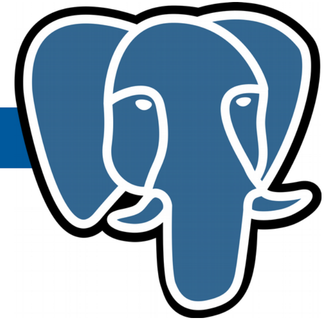


Overview

Background

Complexity

Time Series



Overview

Background

Complexity

Time Series

Schema

Indexing

Normalisation

Summarisation

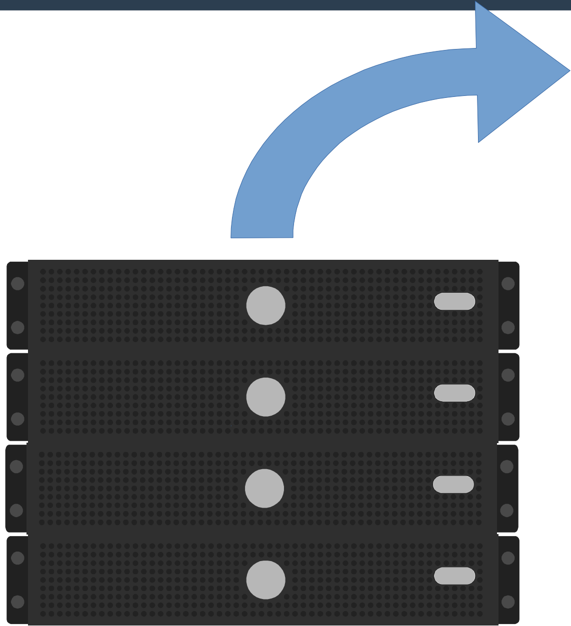
Partitioning

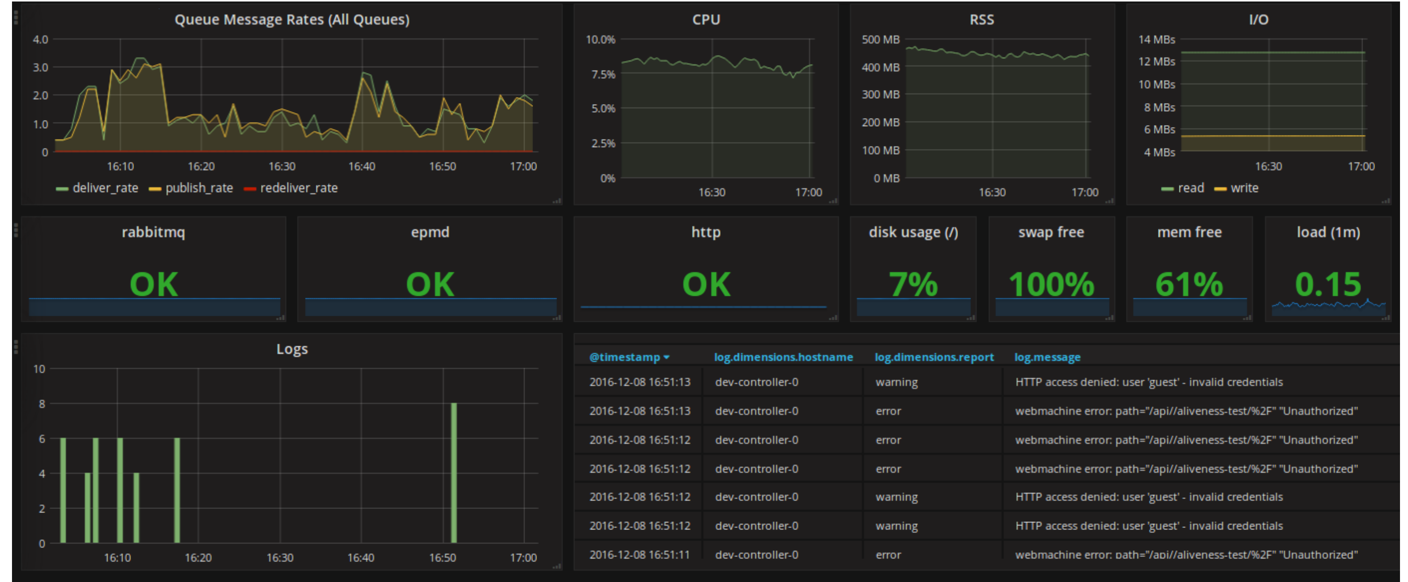
Background

Developers, Developers, Developers

Expensive Toys

Monitoring

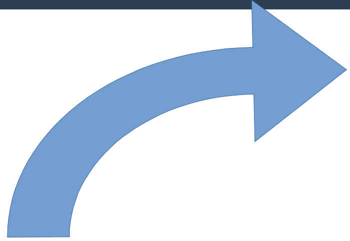


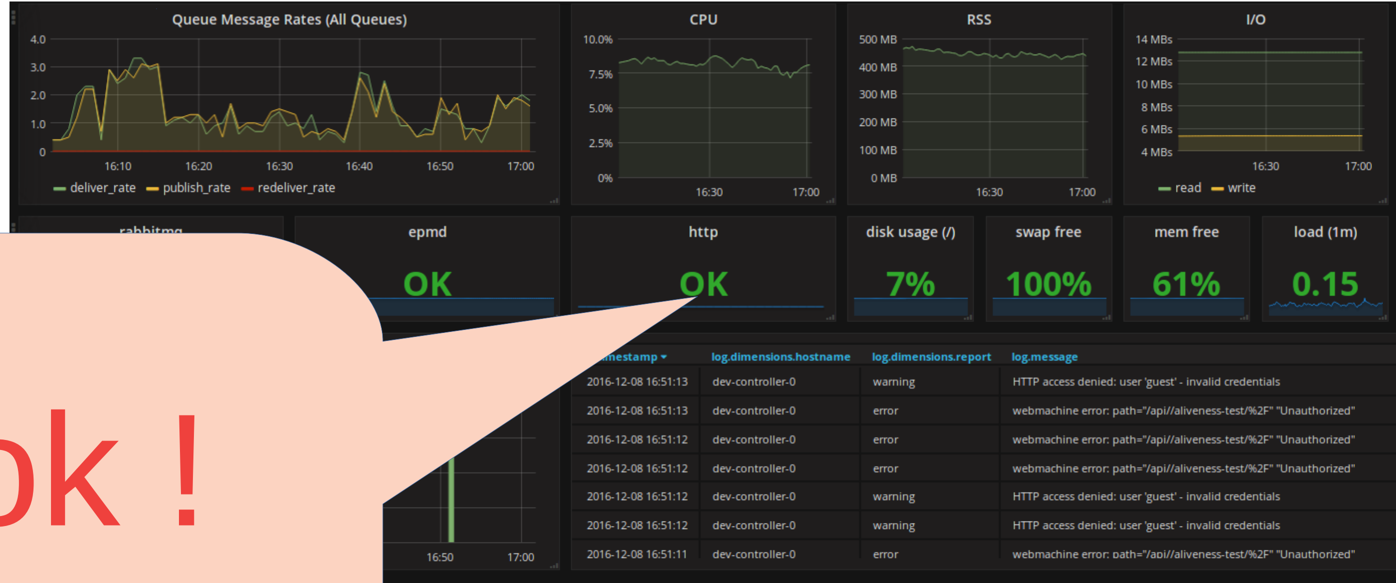


Visibility into the operation of hardware and software

e.g. web site, database, cluster, disk drive

Monitoring





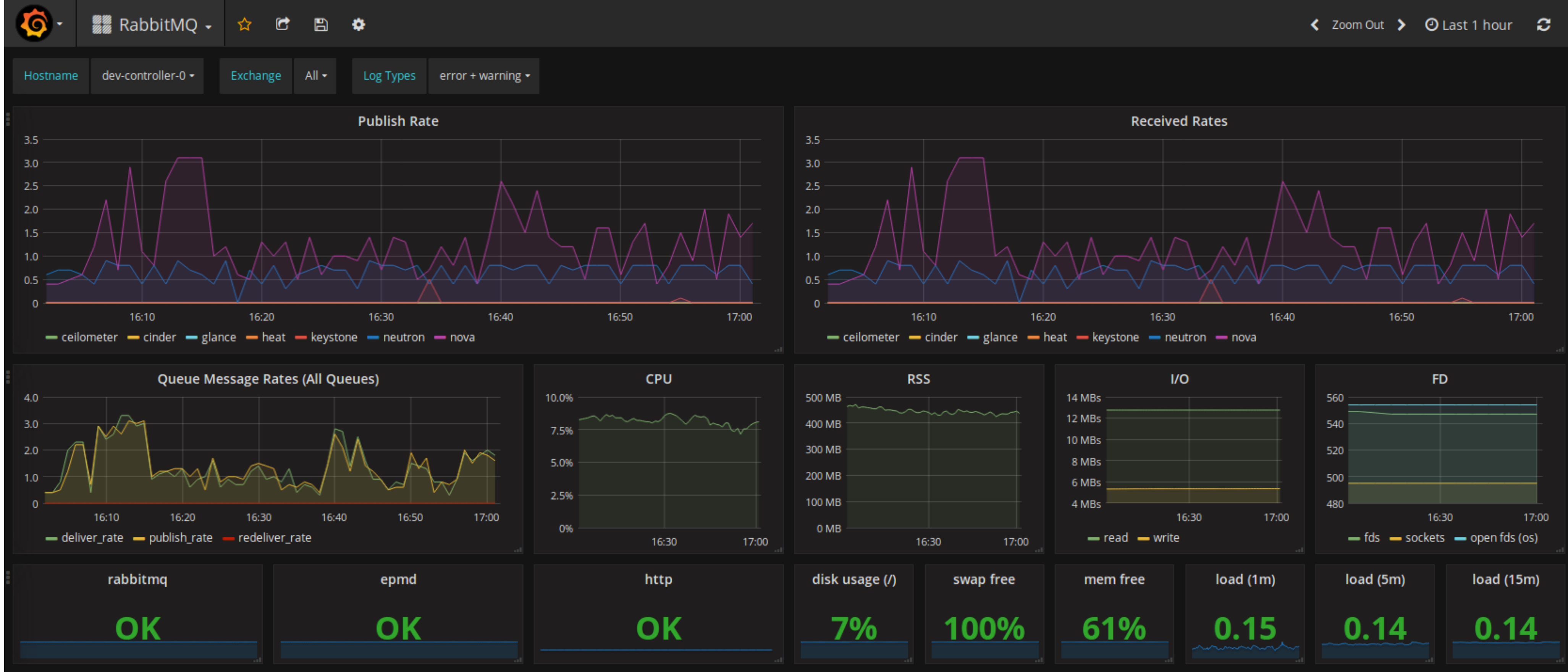
! Look !

! Graphs !

Vi… n of hardware and software

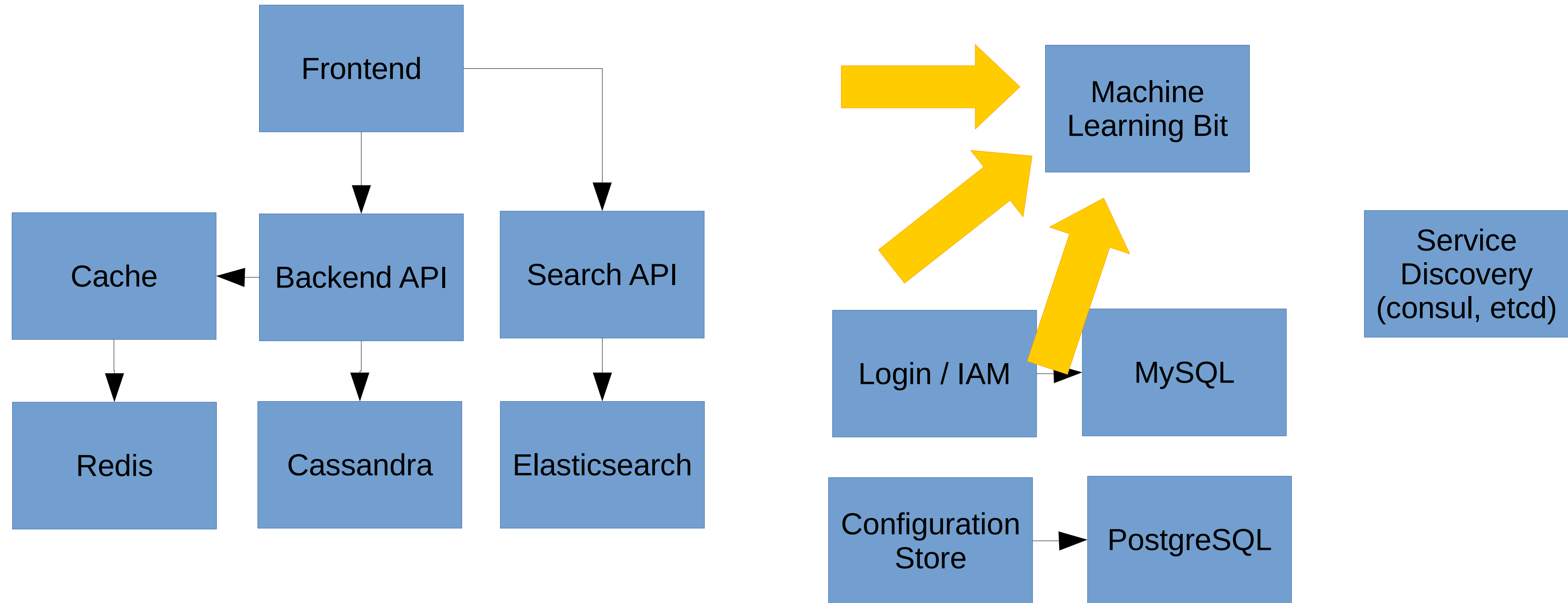
…ase, cluster, disk drive

Monitoring – Time Series

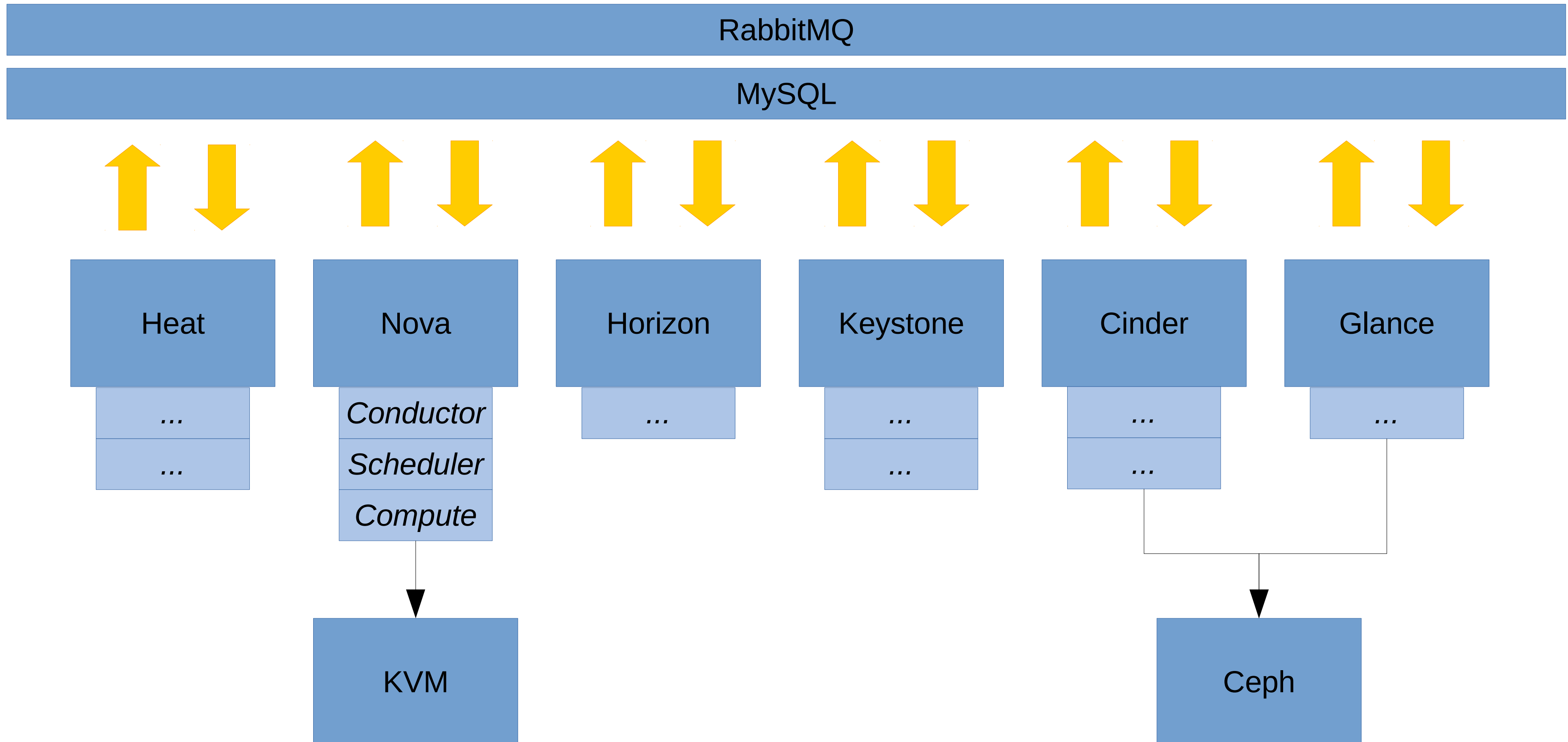


Complexity

“Microservices Hell”



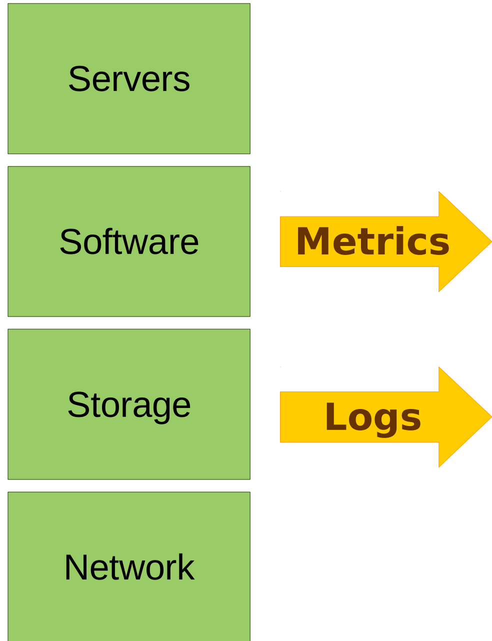
More Cowbell - OpenStack



More Cowbell - OpenStack

Ceph

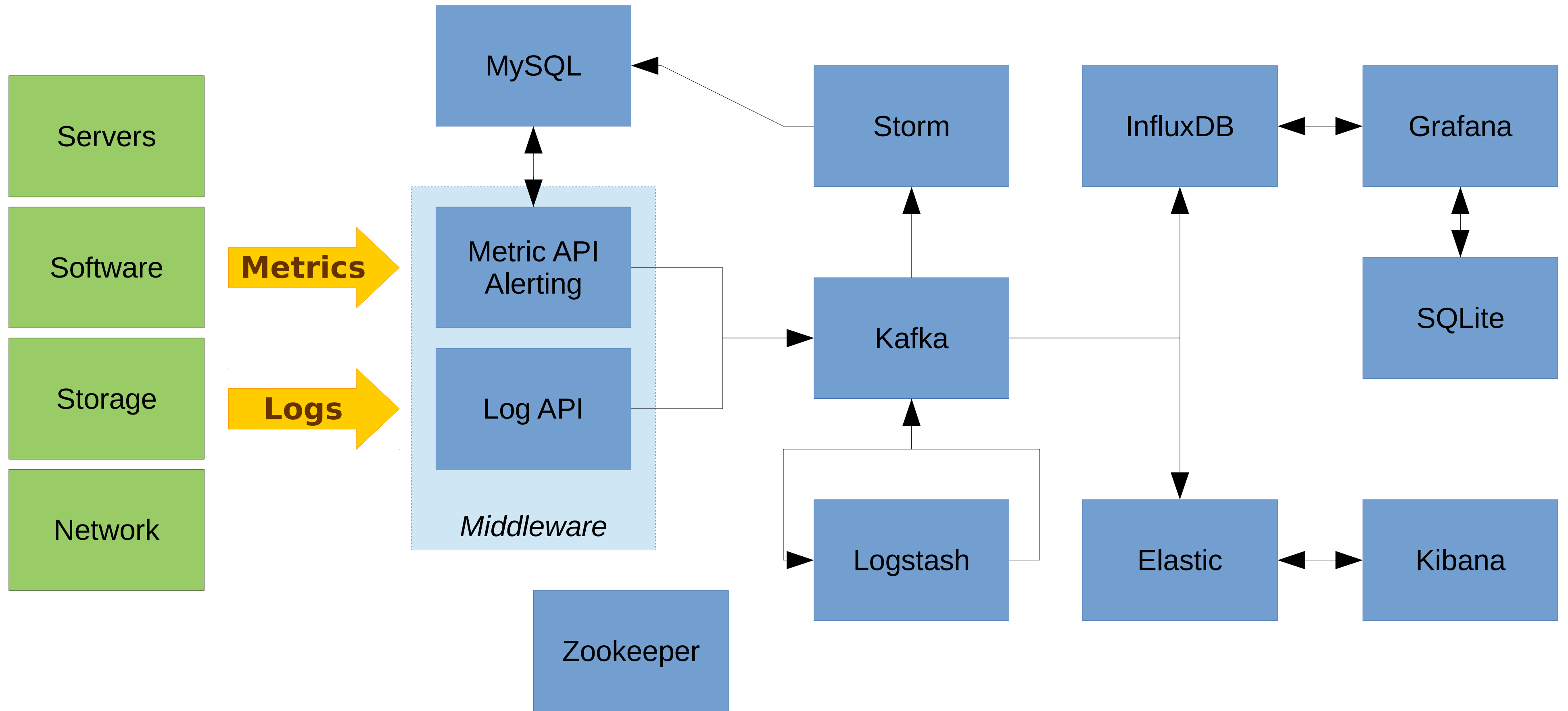
Monitoring



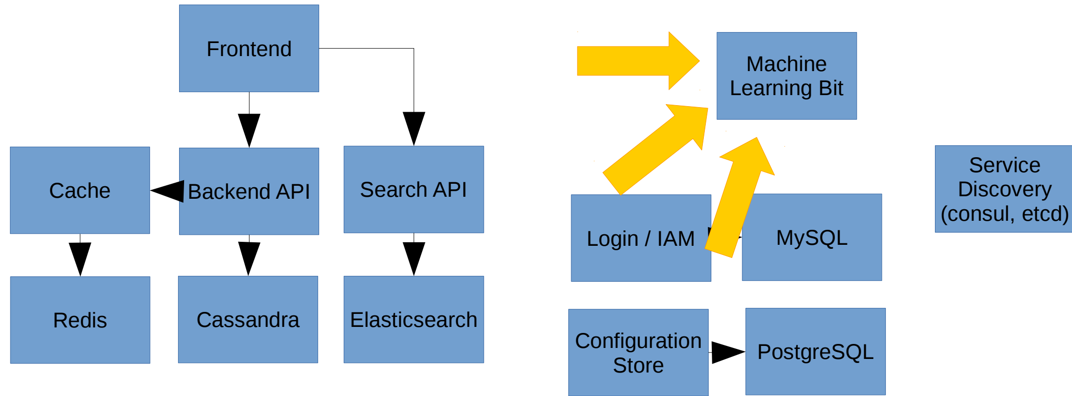
Metrics

Logs

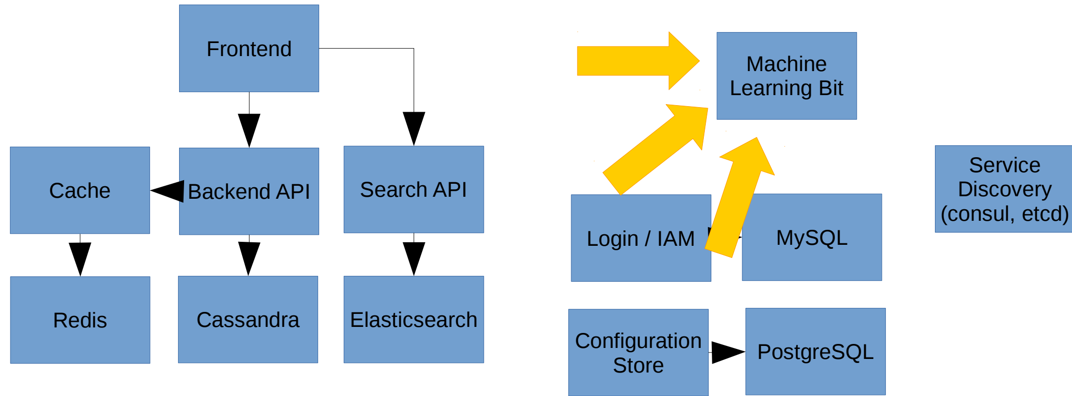
Monitoring



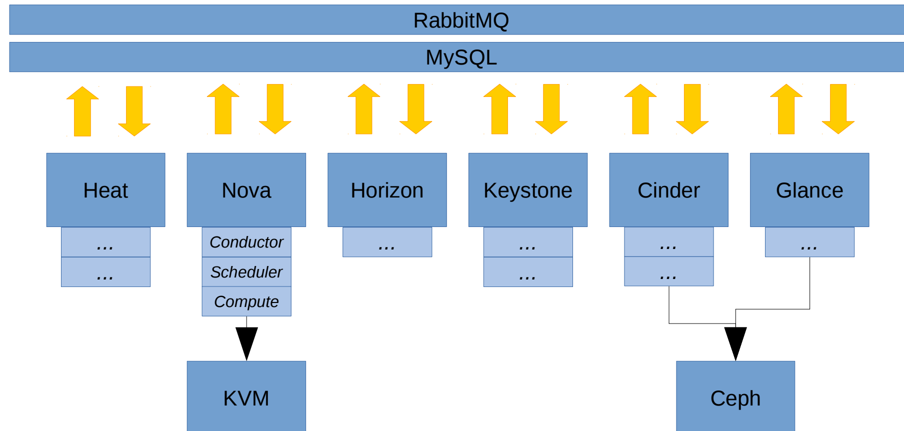
Recap



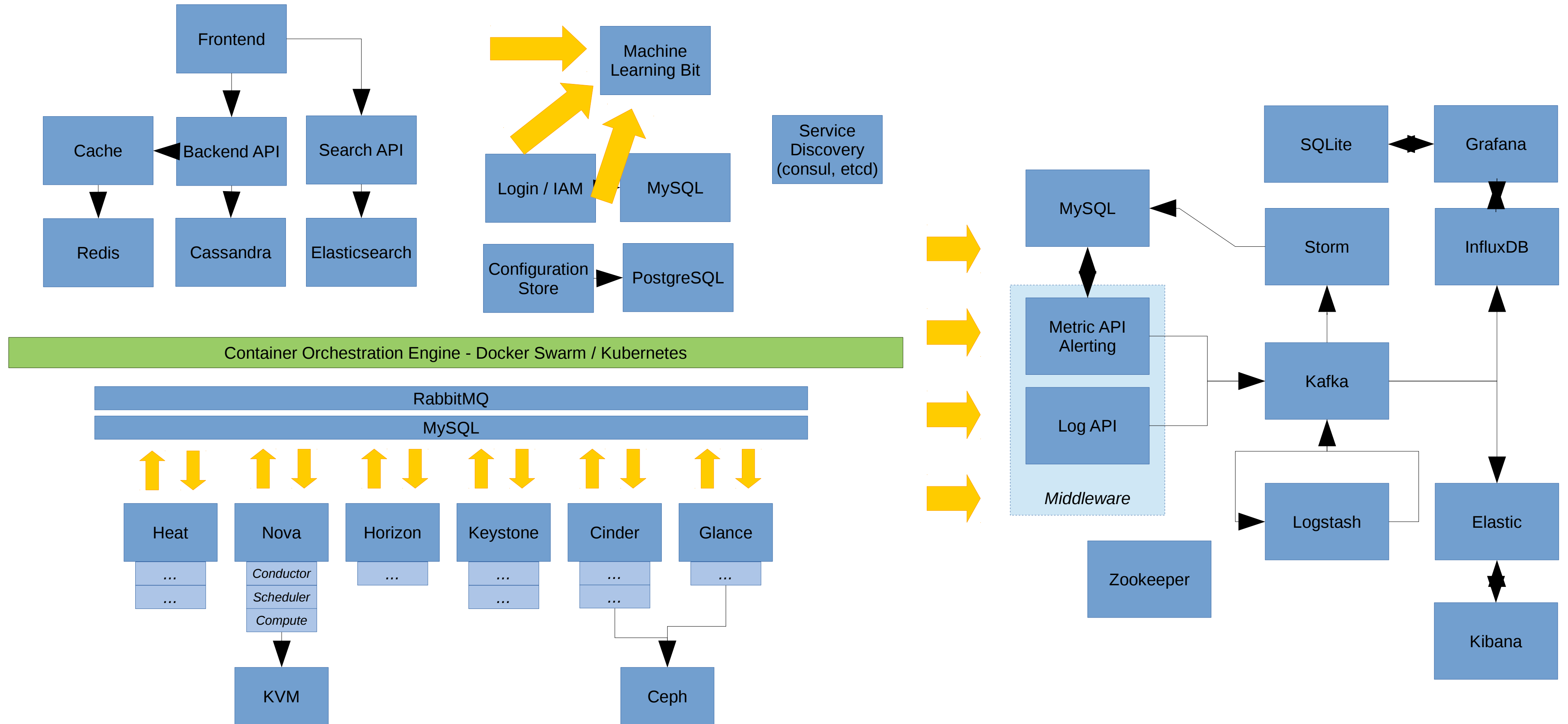
Recap



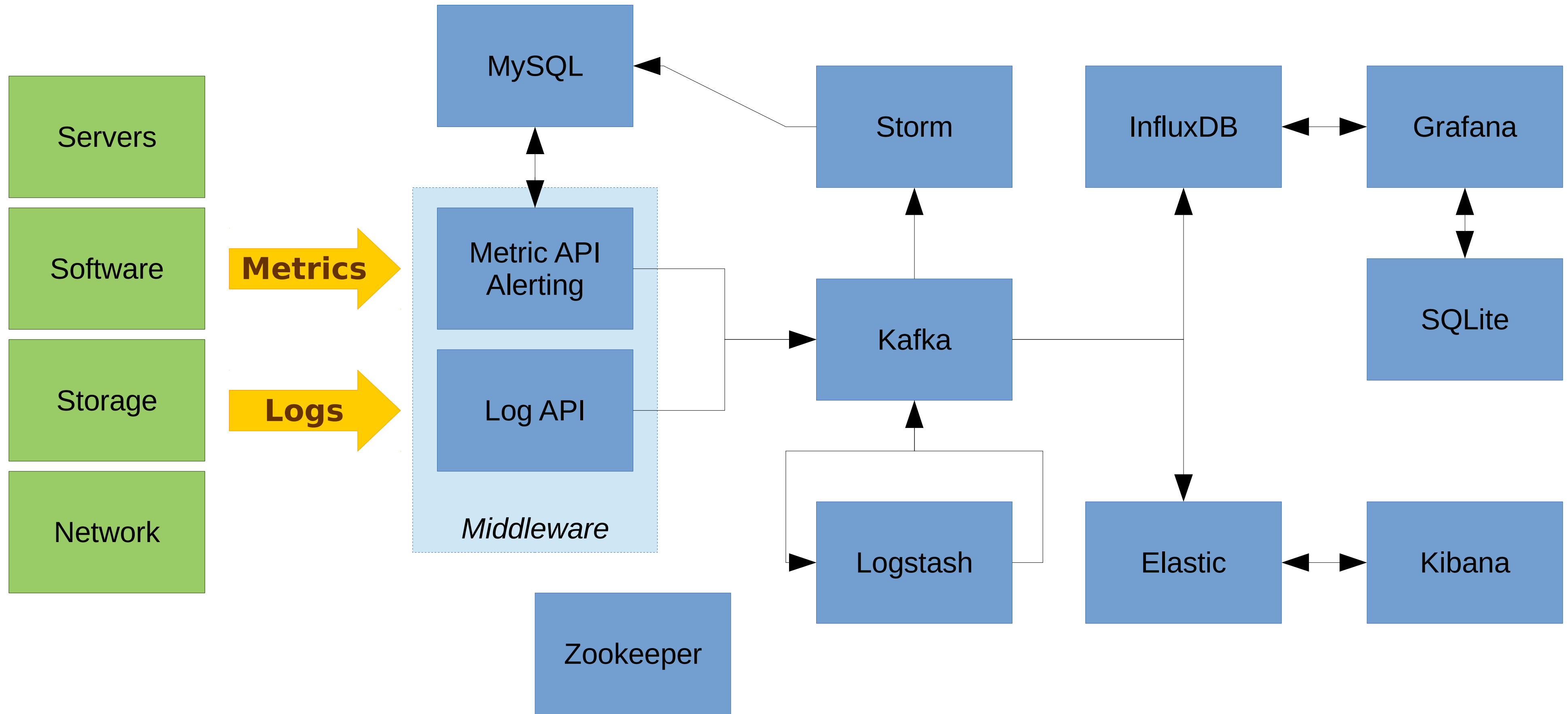
Container Orchestration Engine - Docker Swarm / Kubernetes



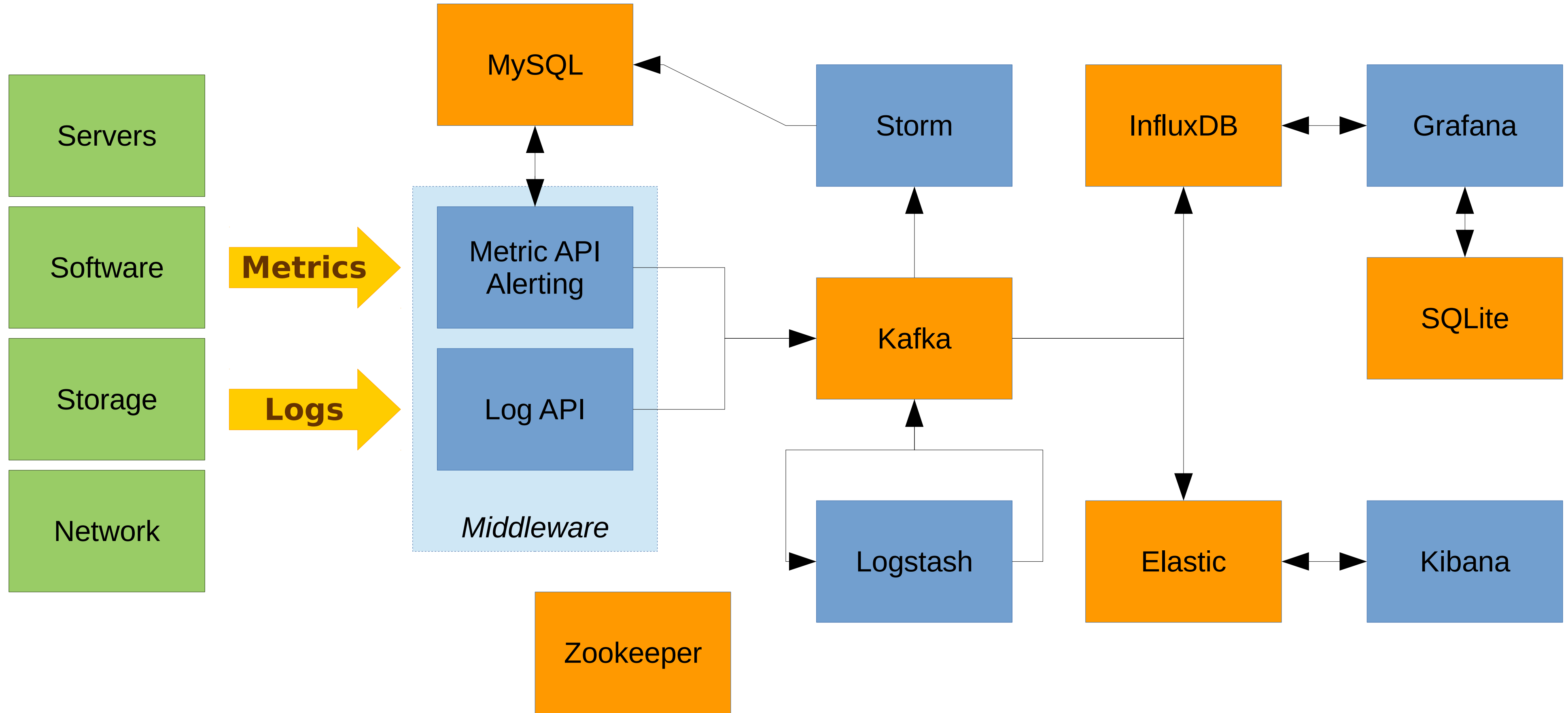
Recap



Monitoring



Monitoring



Monitoring

Commendable “right tool for the job” attitude, but…

At least, could some of the *persistence* be unified?

Fewer failure modes

Fewer backup strategies

Fewer redundancy/replication protocols

One set of consistent data semantics

Re-use existing operational knowledge

One simple question...

Is PostreSQL a Time Series Database?

Time Series

Time Series Databases – The Choice!

- Ganglia (RRDtool)
- Graphite (Whisper)
- OpenTSDB (HBase)
- KairosDB (Cassandra)
- InfluxDB
- Prometheus
- Gnocchi
- Atlas
- Heroic
- Hawkular (Cassandra)
- MetricTank (Cassandra)
- Riak TS (Riak)
- Blueflood (Cassandra)
- DalmatinerDB
- Druid
- BTrDB
- Warp 10 (Hbase)
- Tgres (PostgreSQL!)

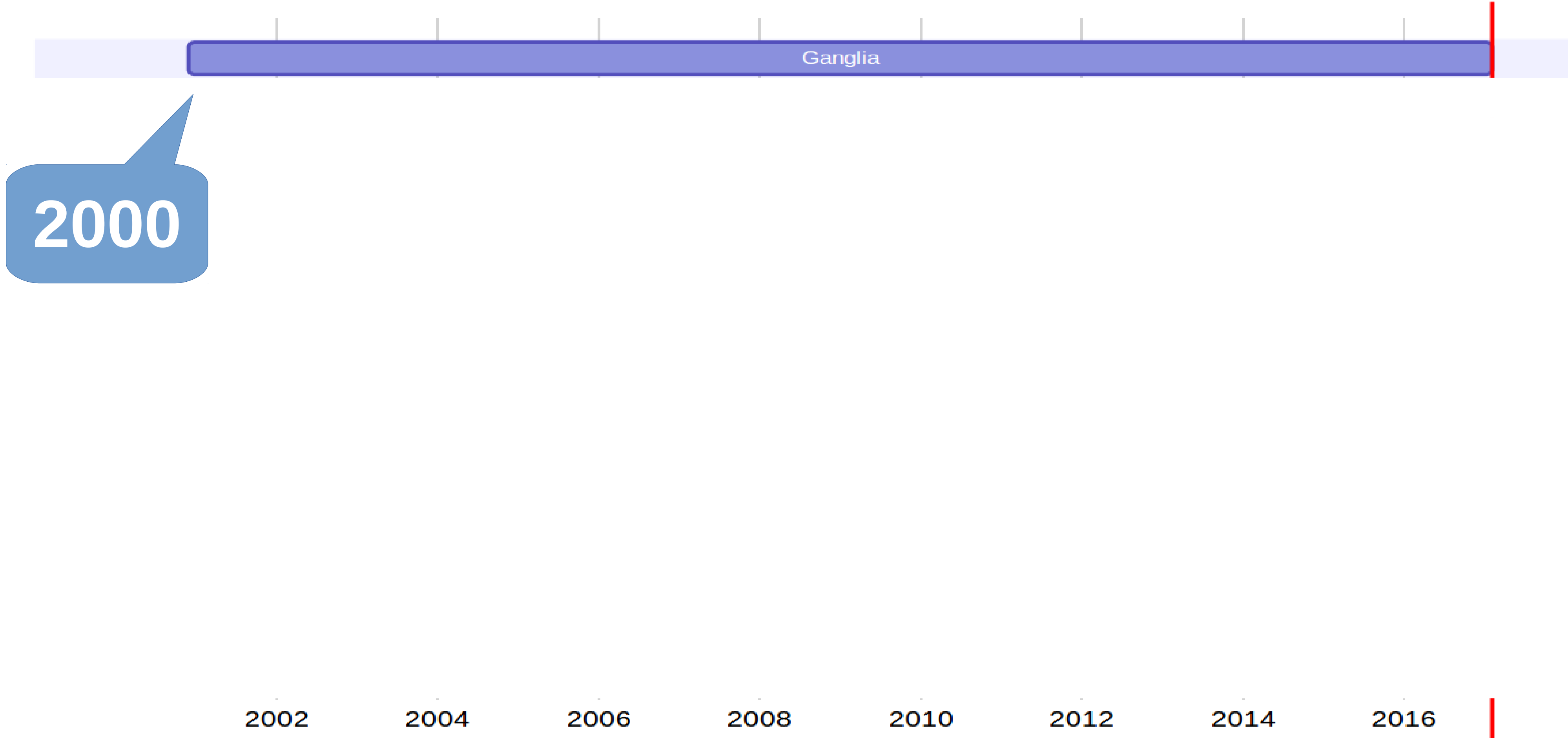
Time Series Databases – The Choice!

- Ganglia [Berkley]
- Graphite [Orbitz]
- OpenTSDB [Stubleupon]
- KairosDB
- InfluxDB
- Prometheus [SoundCloud]
- Gnocchi [OpenStack]
- Atlas [Netflix]
- Heroic [Spotify]
- Hawkular [Redhat]
- MetricTank [Raintank]
- Riak TS [Basho]
- Blueflood [Rackspace]
- DalmatinerDB
- Druid
- BTrDB
- Warp 10
- Tgres

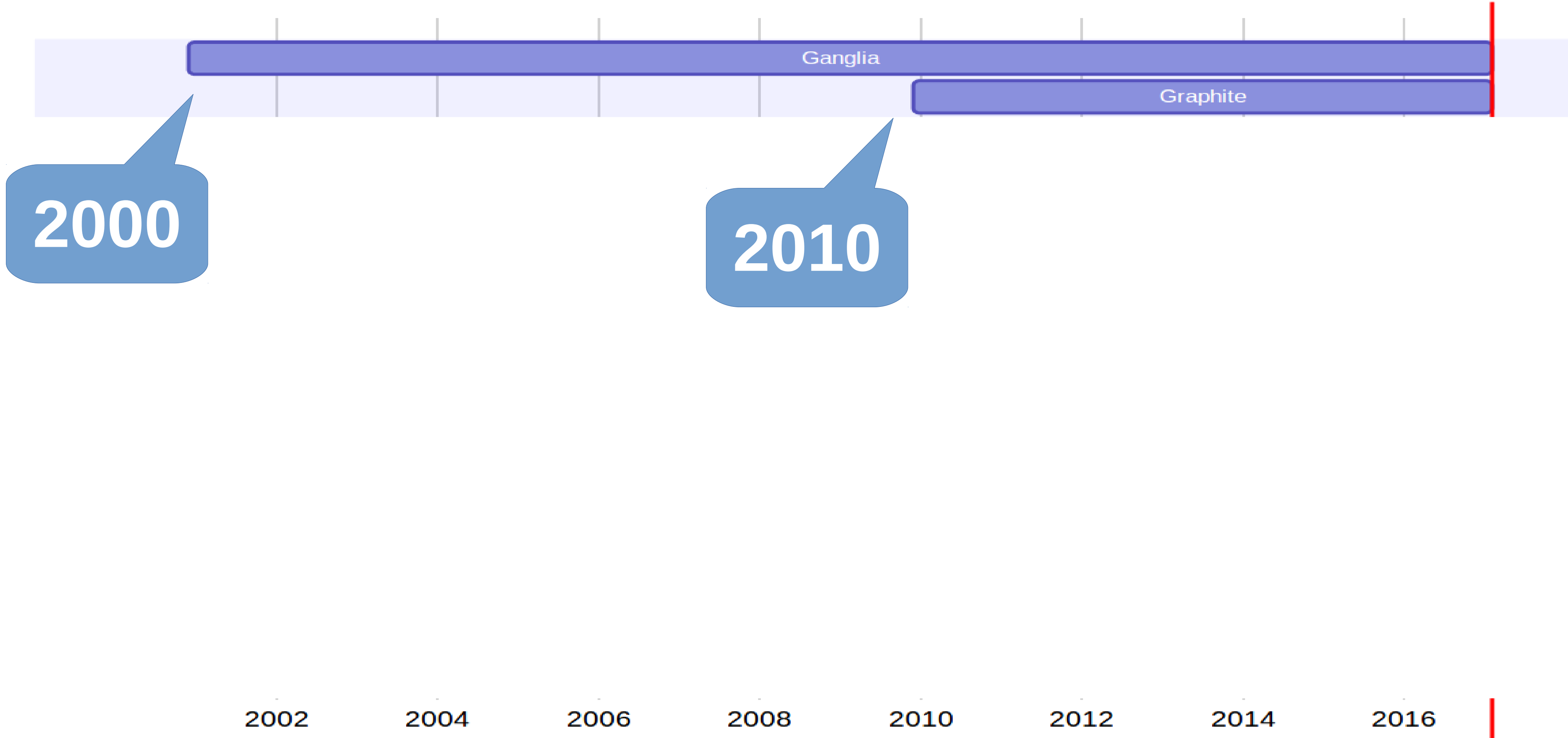
Time Series Databases

2002 2004 2006 2008 2010 2012 2014 2016

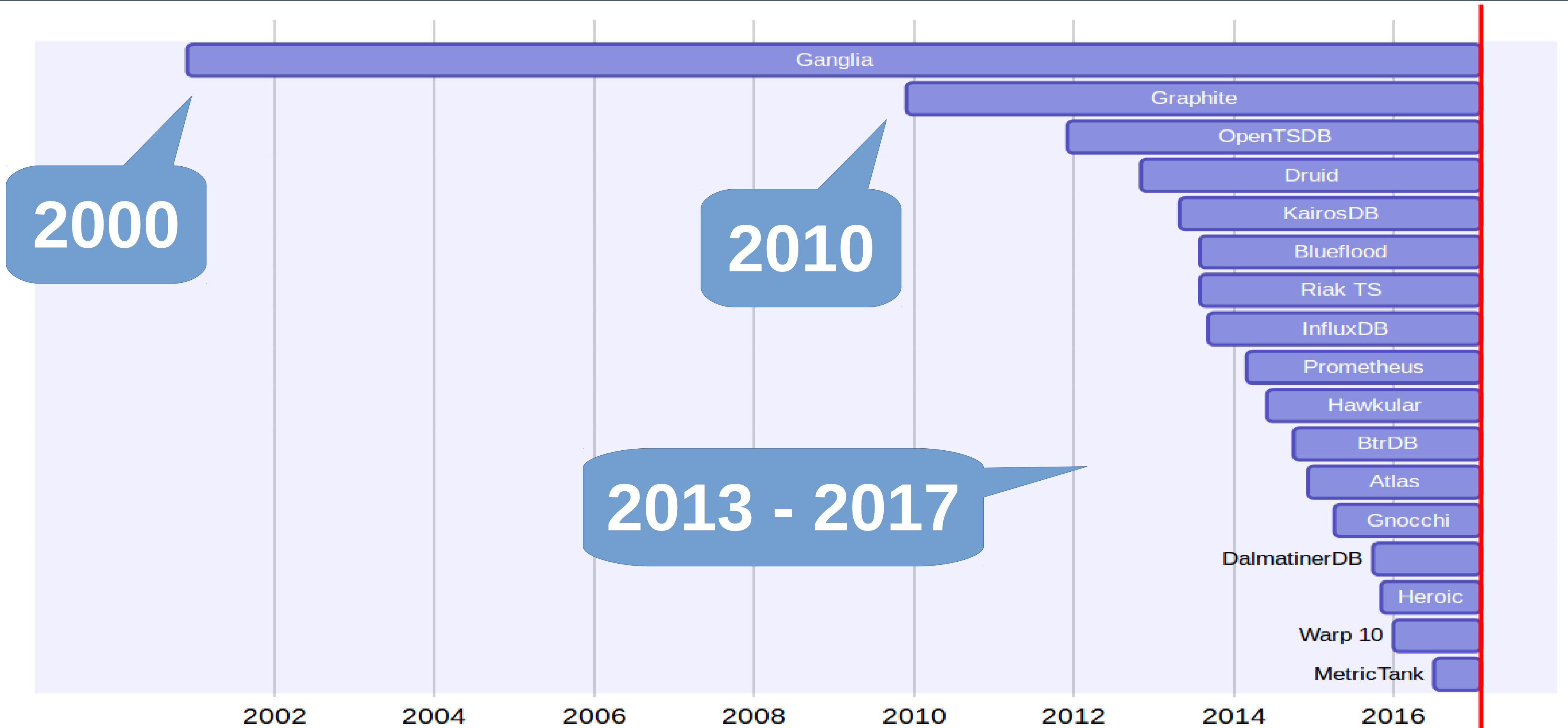
Time Series Databases



Time Series Databases



Time Series Databases



Time Series – Periodic

time	*value*
10:01	15%
10:02	100%
10:03	86%
10:04	60%
10:05	0%

Time Series – Periodic

Metric Meta

time	value	name	dimensions
10:01	15%	cpu	{ host:prod1 }
10:01	1%	cpu	{ host:prod2 }
10:01	24°	temp	{ sensor:rack }
10:02	100%	cpu	{ host:prod1 }
10:02	87%	cpu	{ host:prod2 }
10:02	26°	temp	{ sensor:rack }

Time Series – Sporadic

time	*value*	*name*	*dimensions*	*meta*
10:07	13	log	{ host:prod1 }	{ msg:… }
11:39	1	log	{ host:prod2 }	{ msg:… }
11:50	2	alarm	{ host:prod1 }	{ reason:… }
14:02	1	alarm	{ host:prod1 }	{ reason:… }

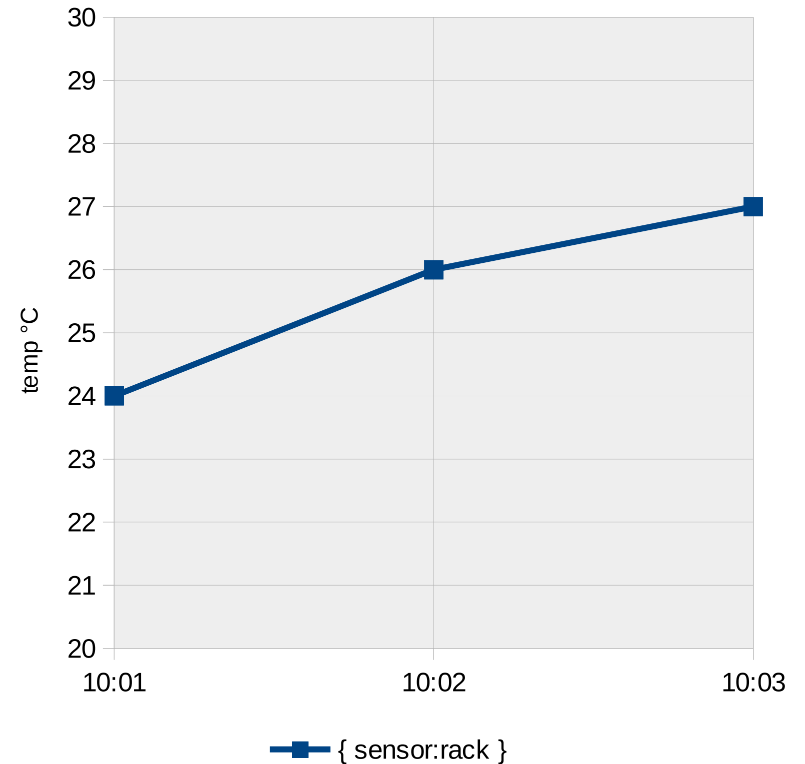
Time Series – Ingest Data

```
"metric": {
  "timestamp": 1232141412,
  "value": 42,
  "name": "cpu.percent",
  "dimensions": {
    "hostname": "dev-01"
  },
  "value_meta": { … }
}
```

- **JSON Encoded**
- **Timestamp (Unix/ISO/..)**
- **Measurement Value**
- **Metric Identification**
  - Name
  - Dimensions (or “Tags”)
- **Optional Event Data**

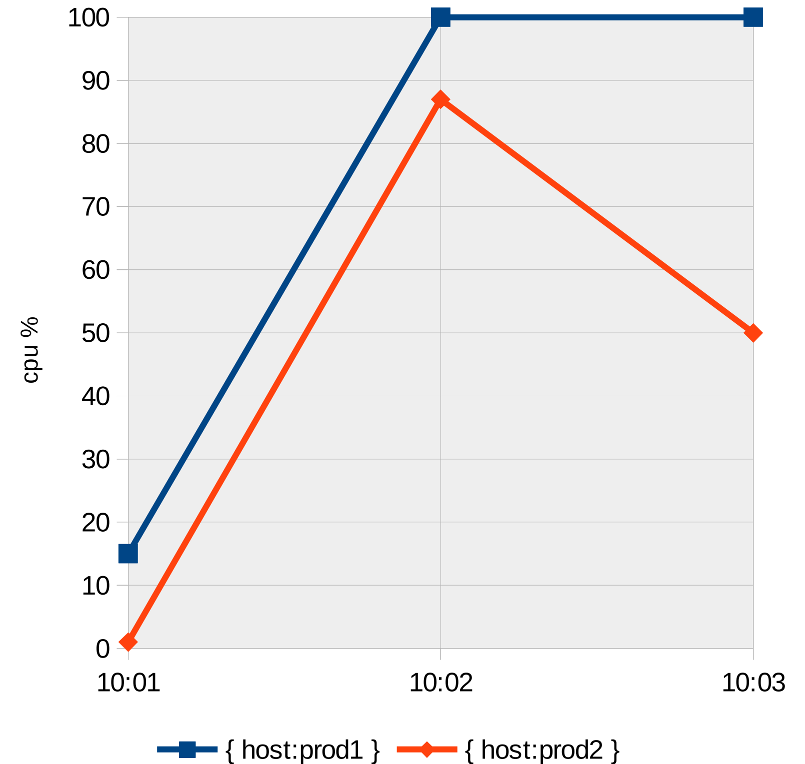
Time Series - Queries

time	value	name	dimensions
10:01	15%	cpu	{ host:prod1 }
10:01	1%	cpu	{ host:prod2 }
10:01	24°	temp	{ sensor:rack }
10:02	100%	cpu	{ host:prod1 }
10:02	87%	cpu	{ host:prod2 }
10:02	26°	temp	{ sensor:rack }
10:03	100%	cpu	{ host:prod1 }
10:03	50%	cpu	{ host:prod2 }
10:03	27°	temp	{ sensor:rack }



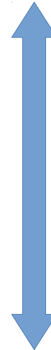
Time Series - Queries

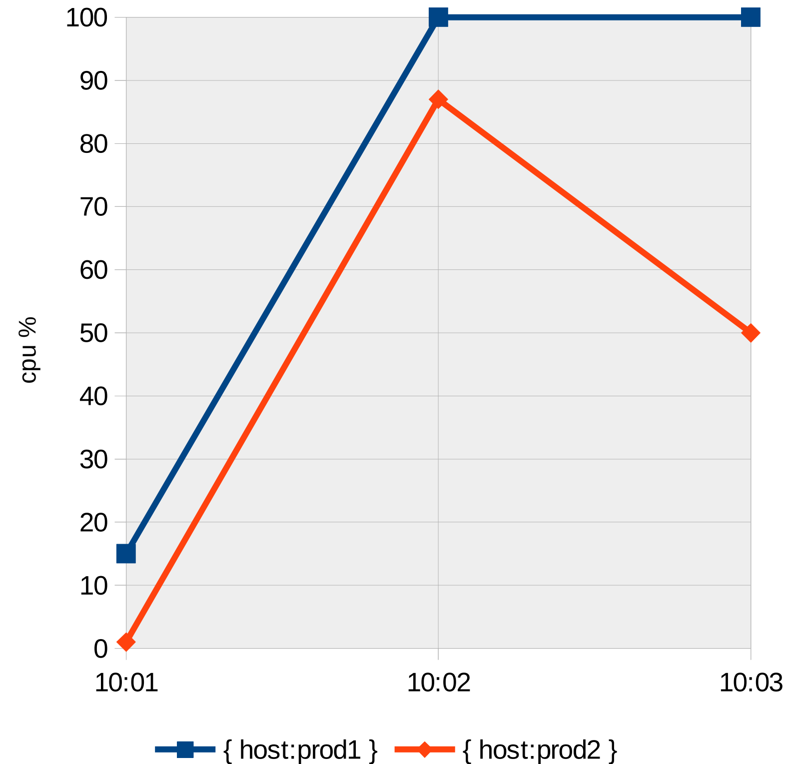
time	value	name	dimensions
10:01	15%	cpu	{ host:prod1 }
10:01	1%	cpu	{ host:prod2 }
10:01	24°	temp	{ sensor:rack }
10:02	100%	cpu	{ host:prod1 }
10:02	87%	cpu	{ host:prod2 }
10:02	26°	temp	{ sensor:rack }
10:03	100%	cpu	{ host:prod1 }
10:03	50%	cpu	{ host:prod2 }
10:03	27°	temp	{ sensor:rack }



Time Series – Queries

time	value	name	dimensions
10:00	?	?	?
10:00	?	?	?
10:00	?	?	?
10:01	15%	cpu	{ host:prod1 }
10:01	1%	cpu	{ host:prod2 }
10:01	24°	temp	{ sensor:rack }
10:02	100%	cpu	{ host:prod1 }
10:02	87%	cpu	{ host:prod2 }
10:02	26°	temp	{ sensor:rack }
10:03	100%	cpu	{ host:prod1 }
10:03	50%	cpu	{ host:prod2 }
10:03	27°	temp	{ sensor:rack }
10:04	?	?	?
10:04	?	?	?
10:04	?	?	?
10:05	?	?	?
10:05	?	?	?
10:05	?	?	?





Time Series – Scaling

time	*value*	*name*	*dimensions*
10:01	15%	cpu	{host:prod1}
10:01	1%	cpu	{host:prod2}
10:01	24°	temp	{sensor:rack}
10:02	100%	cpu	{host:prod1}
10:02	87%	cpu	{host:prod2}
10:02	26°	temp	{sensor:rack}
10:03	100%	cpu	{host:prod1}
10:03	50%	cpu	{host:prod2}
10:03	27°	temp	{sensor:rack}

- **Retention (Volume)**
  - How long is data kept
  - Data volume stored
- **Metric Amount**
  - Number of series names
  - Additional dimensions
- **Query Complexity**
  - Time range size
  - Number of metrics

Relational Model

Relational Model – Denormalised Schema

```
CREATE TABLE measurements (
  timestamp TIMESTAMPTZ,
  value FLOAT8,
  name VARCHAR,
  dimensions JSONB,
  value_meta JSON
);
```

- **Store in single table**
  - Trivial mapping of data
  - Row is a measurement
- **Native format time**
  - Normalise to UTC timezone
  - “Just use TIMESTAMPTZ”
- **Floating point value**
  - Typical for sensor use cases
  - Strict accuracy: use NUMERIC
    - e.g. Money!

Relational Model – Denormalised Schema

```
CREATE TABLE measurements (
  timestamp TIMESTAMPTZ,
  value FLOAT8,
  name VARCHAR,
  dimensions JSONB,
  value_meta JSON
);
```

- **Arbitrary length name**
  - Fixed length: no real effect
- **Key/value dimensions**
  - JSONB is binary encoded
  - Efficient access to children
  - Best if querying contents
- **Key/value metadata**
  - JSON just validated text
  - Best if purely payload

Relational Model - Series Listing Query

```
SELECT DISTINCT
  name,
  dimensions
FROM
  measurements
WHERE
  name = 'cpu.percent'
  dimensions @>
    '{"host": "dev-01"}'::JSONB
```

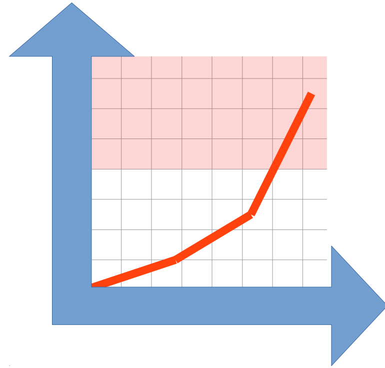
- **Select metric metadata**
- **Show each metric once**
- **Optionally filter**
  - All metrics with name
  - All metrics for a host

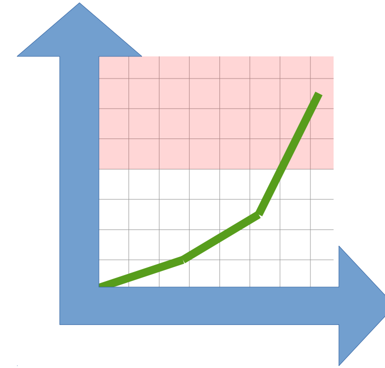
Relational Model – Single Series Query

```
SELECT
  TIME_ROUND(timestamp, 60),
  AVG(value)
FROM
  measurements
WHERE
  timestamp BETWEEN
    '2015-01-01Z00:00:00' AND
    '2015-01-01Z01:00:00'
  AND name =
    'cpu.percent'
  AND dimensions @>
    '{"host": "dev-01"}'::JSONB
GROUP BY
    1
```

- **Find measurements**
  - Specify time range
  - Specify metric name
  - Specify dimension value
- **Aggregate data points**
  - Round to desired interval
  - Group by that interval
  - Take average of all data points in that interval

Relational Model – Performance Analysis

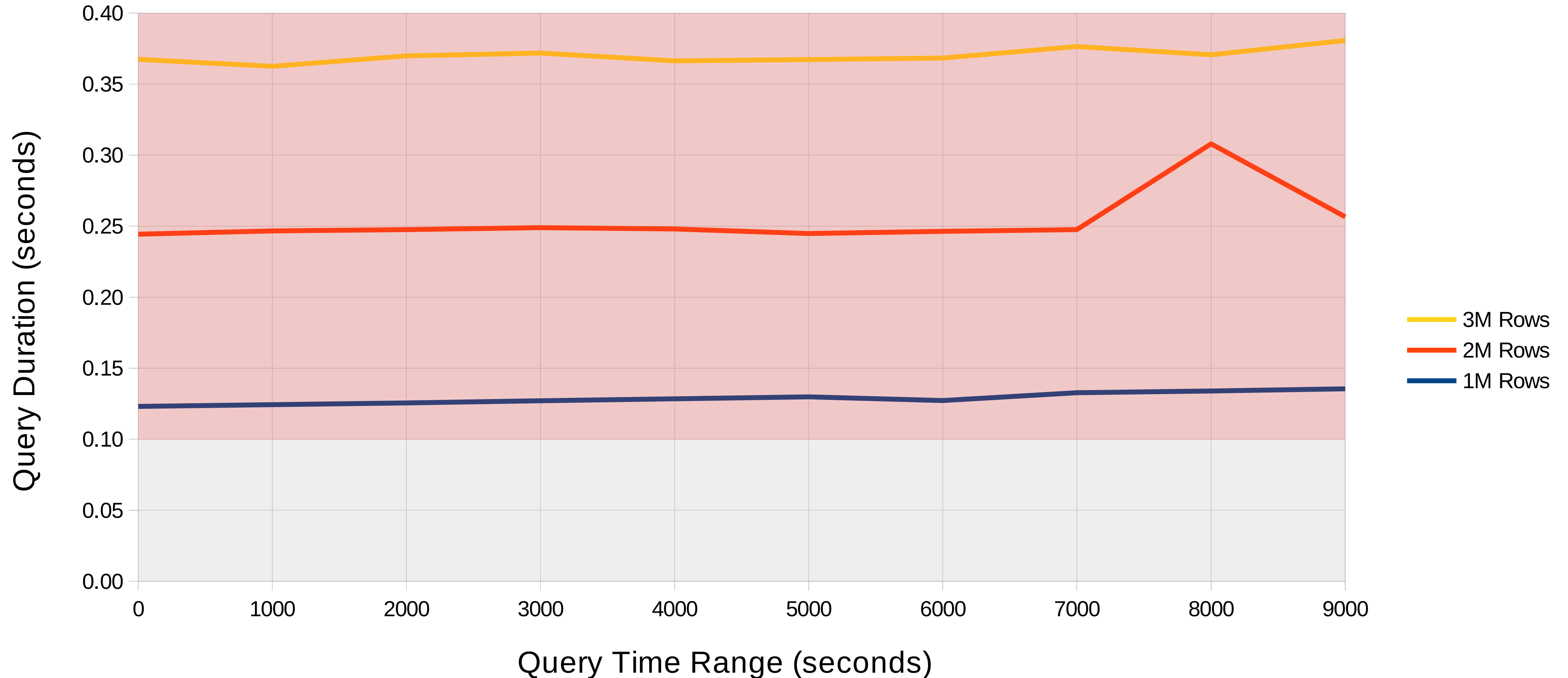




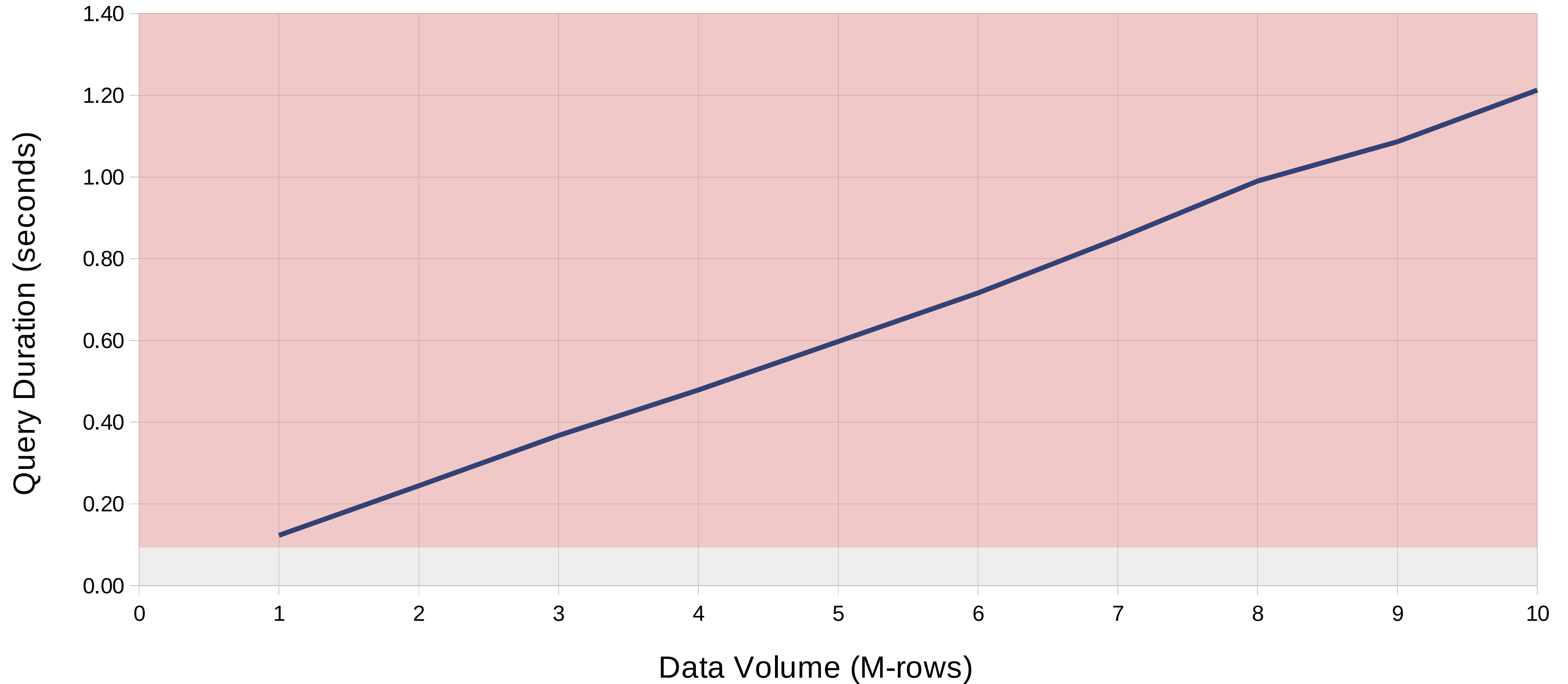
Target: <100ms

(Query Duration)

Relational Model – Single Series Query (vs Time Range)



Relational Model – Single Series Query (vs Data Volume)



Relational Model - Analysis

- ✔ **Query time fixed regardless of time range**
- ✔ **On target for < ~1M rows**

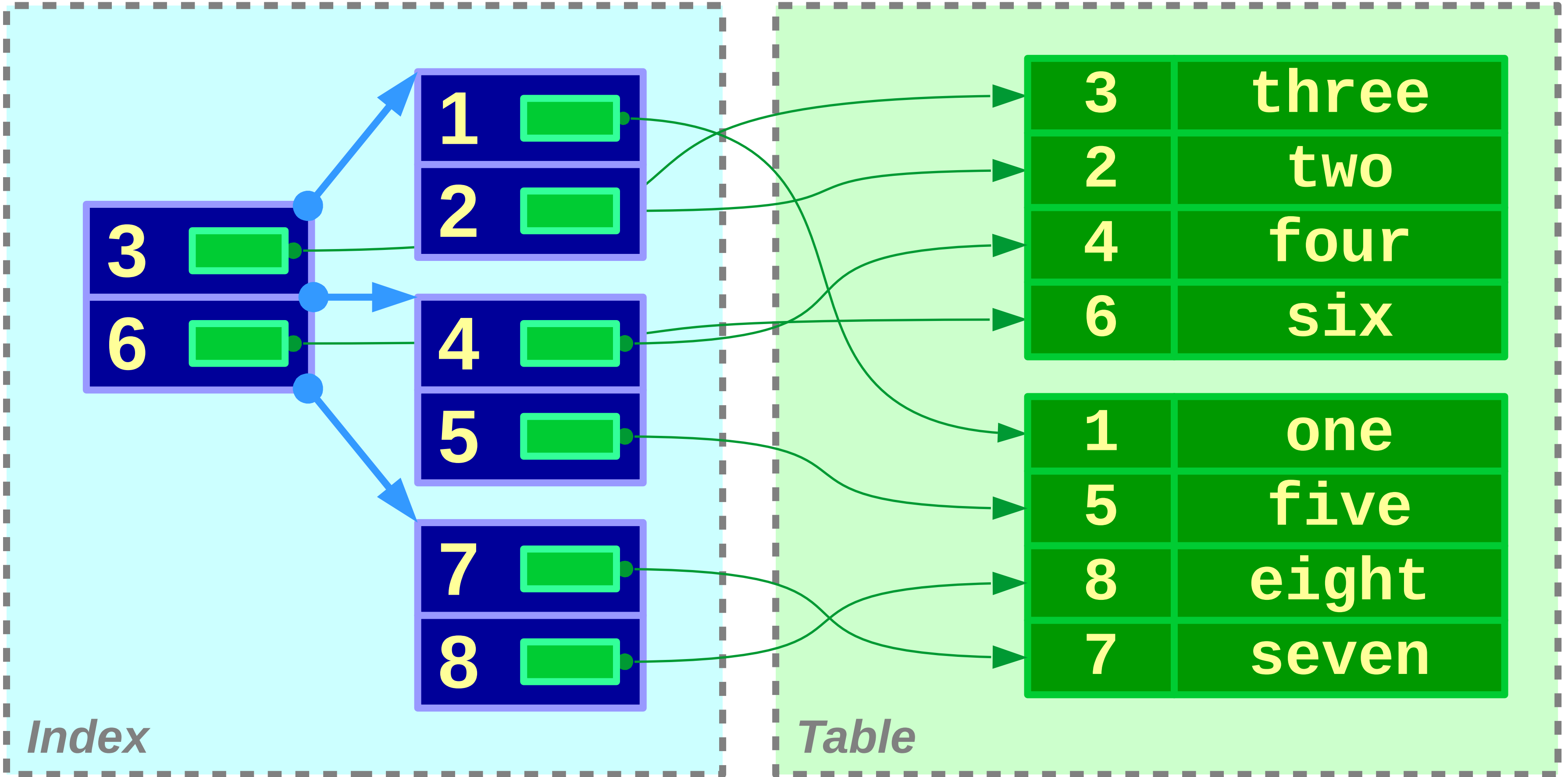
- ✗ **Query time scales linearly with data volume**
- ✗ **Every query reads every row**
  - ✗ Full table scan

Indexing

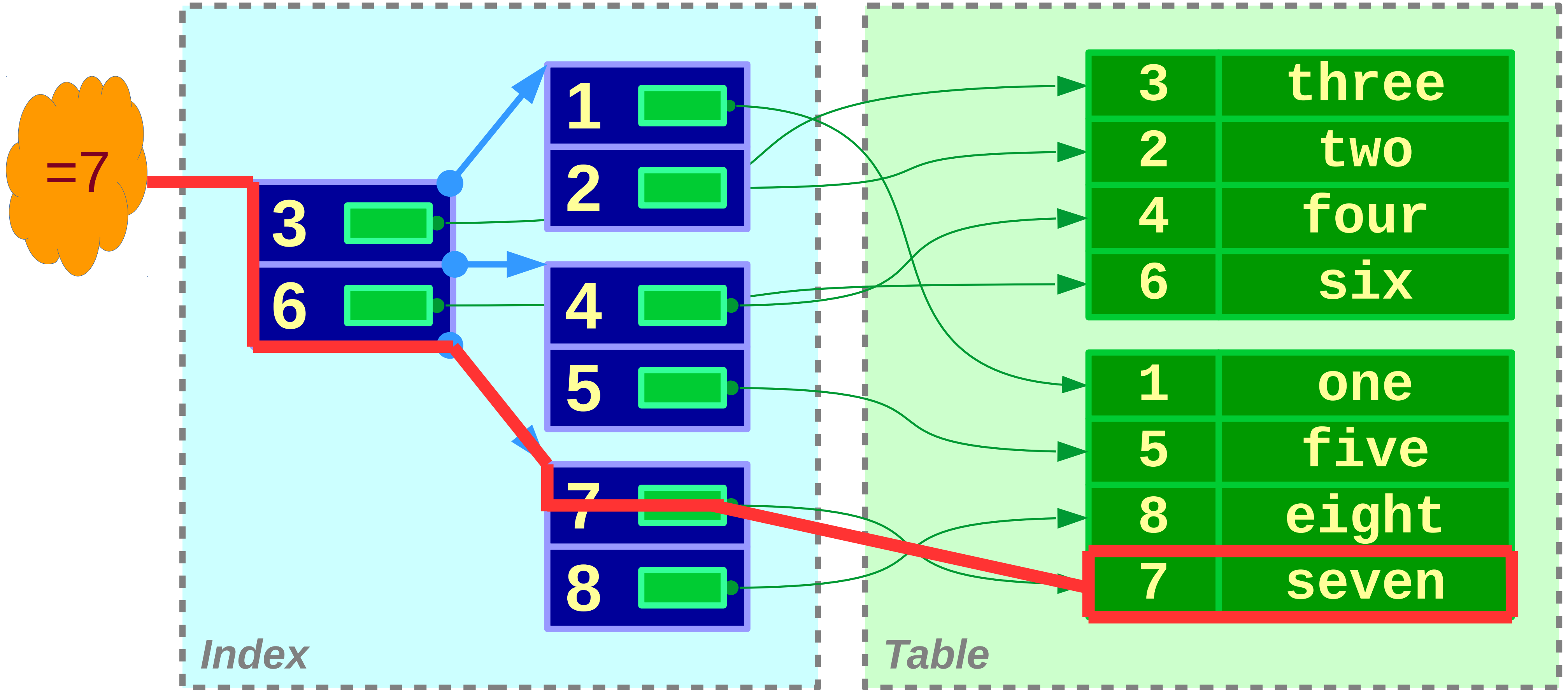
Indexing

- **Timestamps are essentially integers**
- **PostgreSQL has many index types**
  BTREE, HASH, BRIN, GIN, GIST
- **BTREE excellent for Equality and Between**

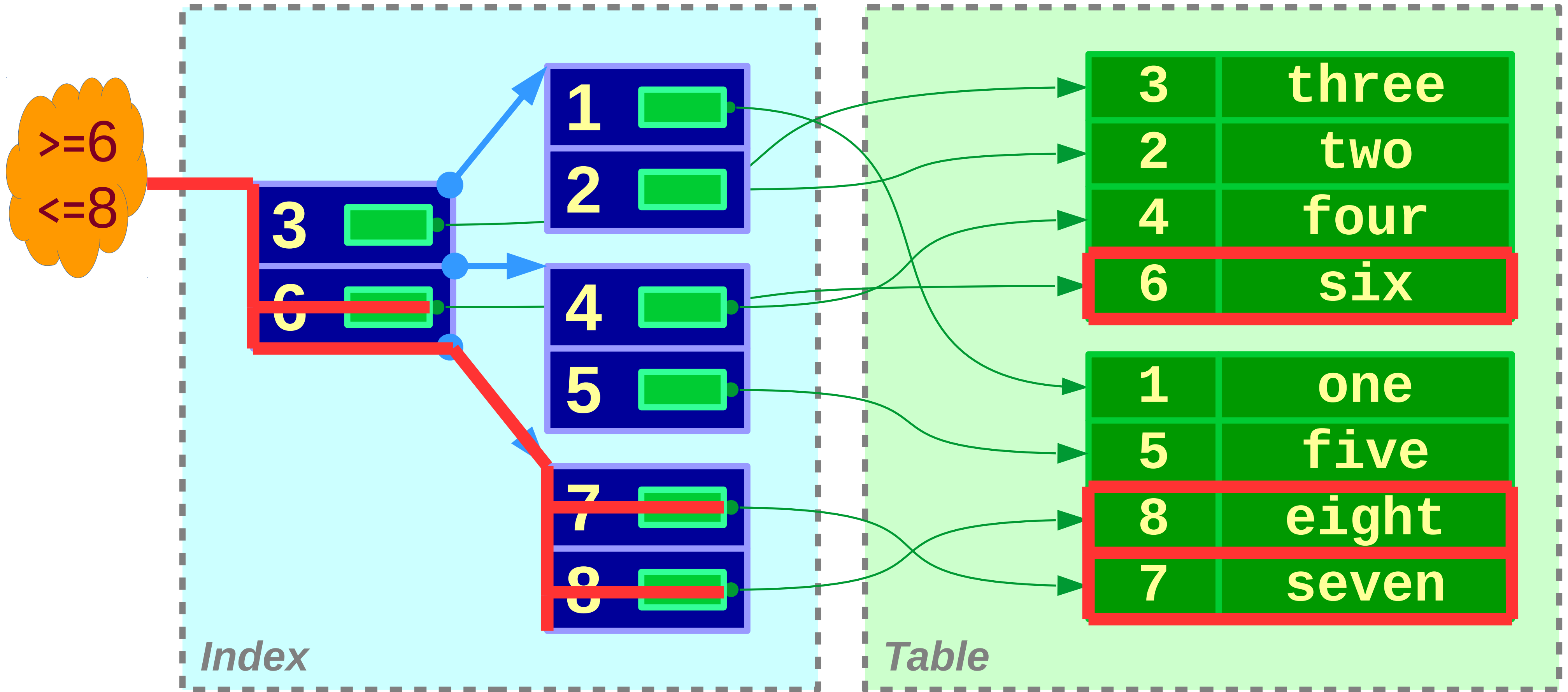
Indexing - BTREE



Indexing - BTREE



Indexing - BTREE



Indexing - Single Series Query

```
SELECT
  TIME_ROUND(timestamp, 60),
  AVG(value)
FROM
  measurements
WHERE
  timestamp BETWEEN
    '2015-01-01Z00:00:00' AND
    '2015-01-01Z01:00:00'
  AND name =
    'cpu.percent'
  AND dimensions @>
    '{"host": "dev-01"}'::JSONB
GROUP BY
    1
```

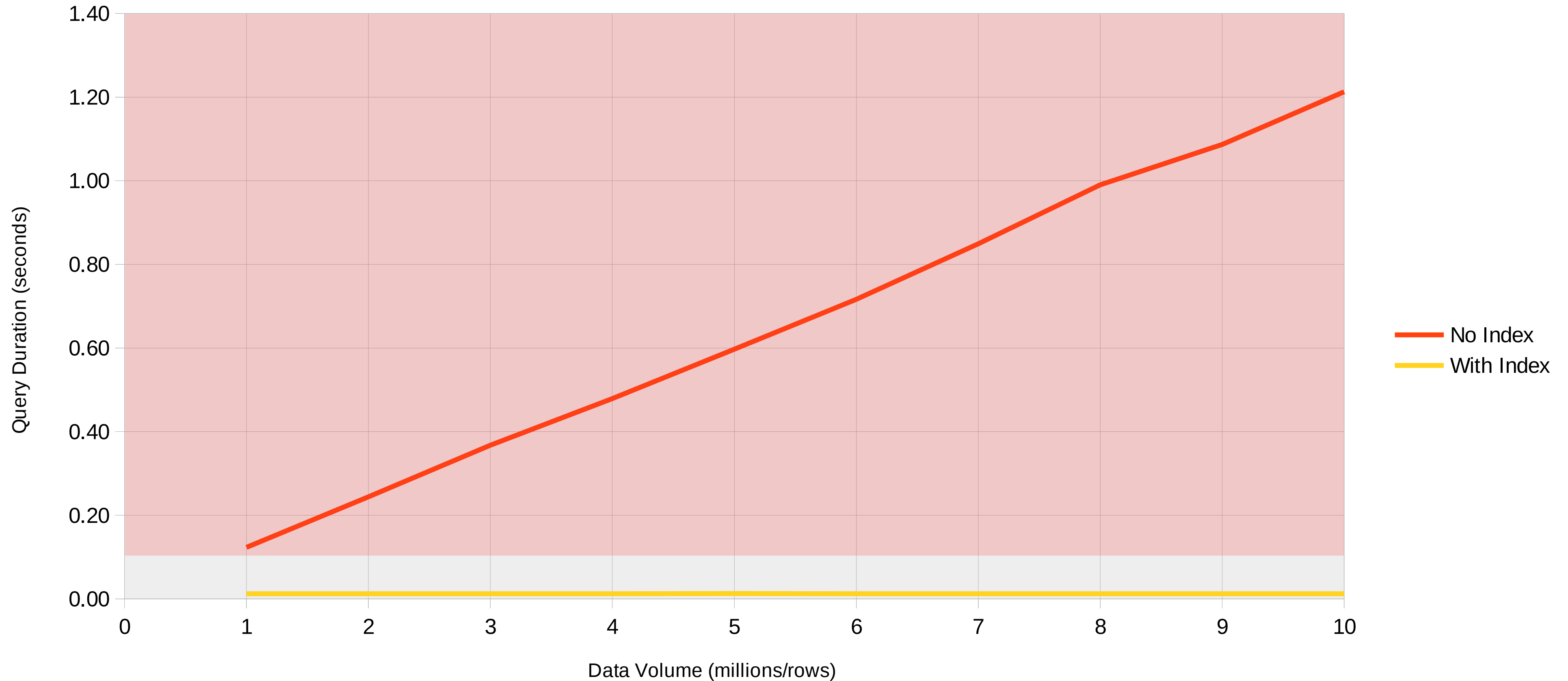
- **BETWEEN predicate**
- **Eliminates huge portion of table contents**
  - High selectivity
- **Excellent candidate for index creation**

Indexing - Timestamp

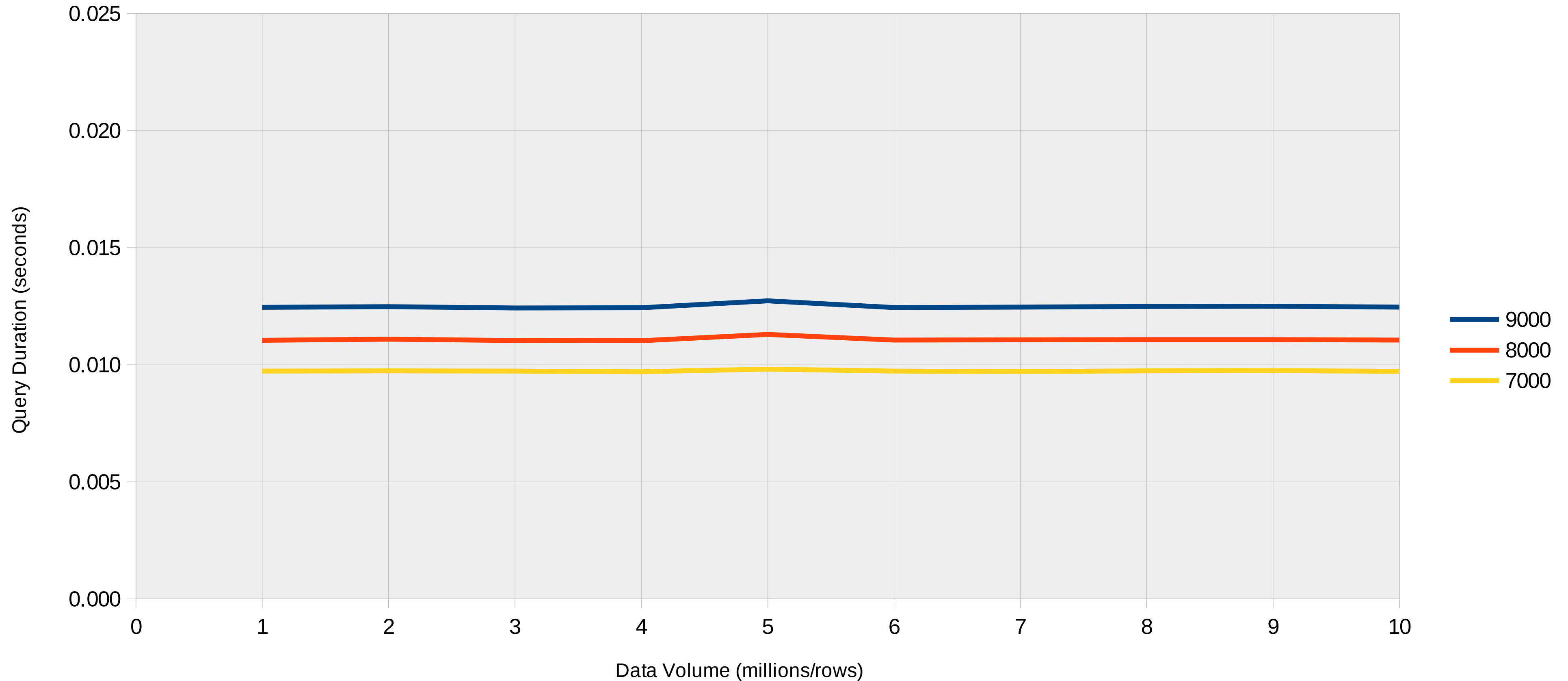
```
CREATE INDEX ON
  measurements
  USING BTREE
  (timestamp);
```

- **Specify table to index**
- **Specify index type**
  - Optional: BTREE is default
- **Specify column to index**

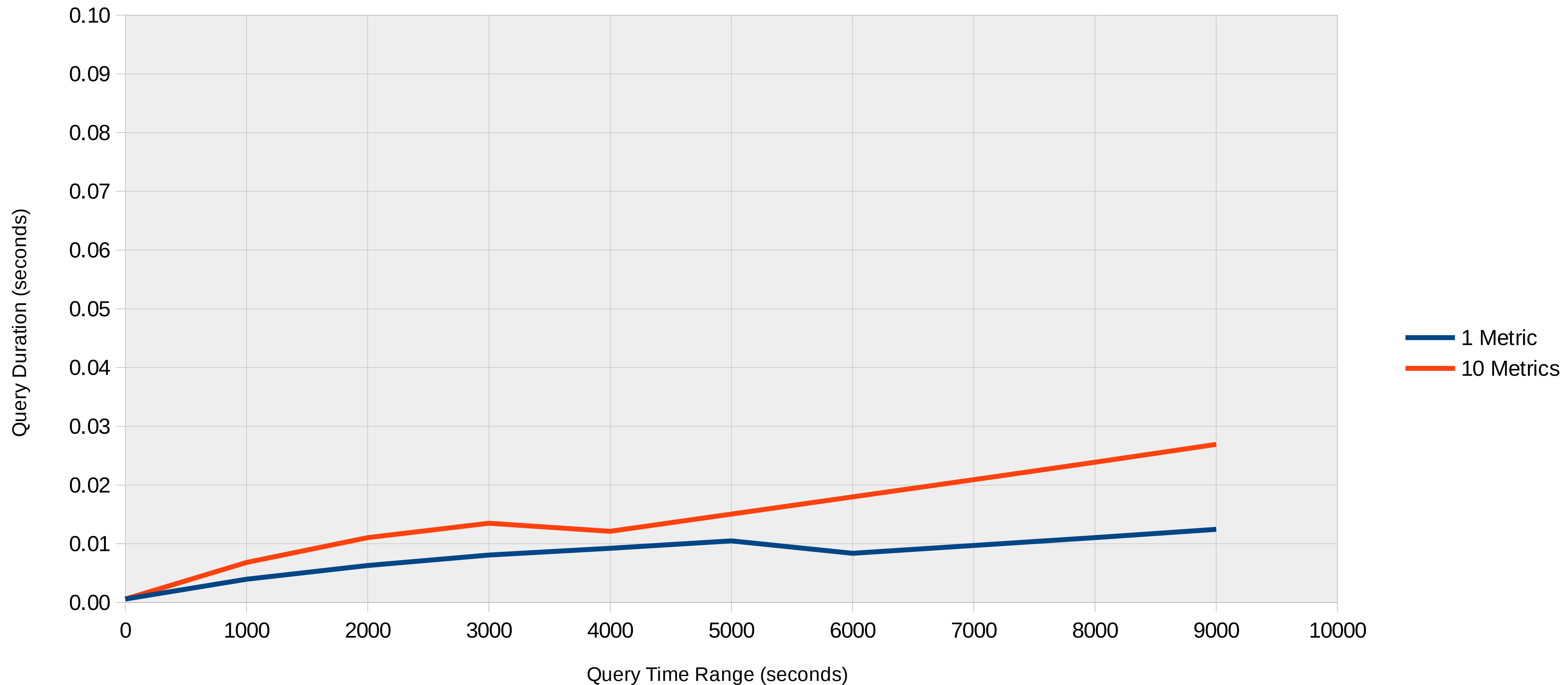
Indexing – Single Series Query (vs Data Volume)



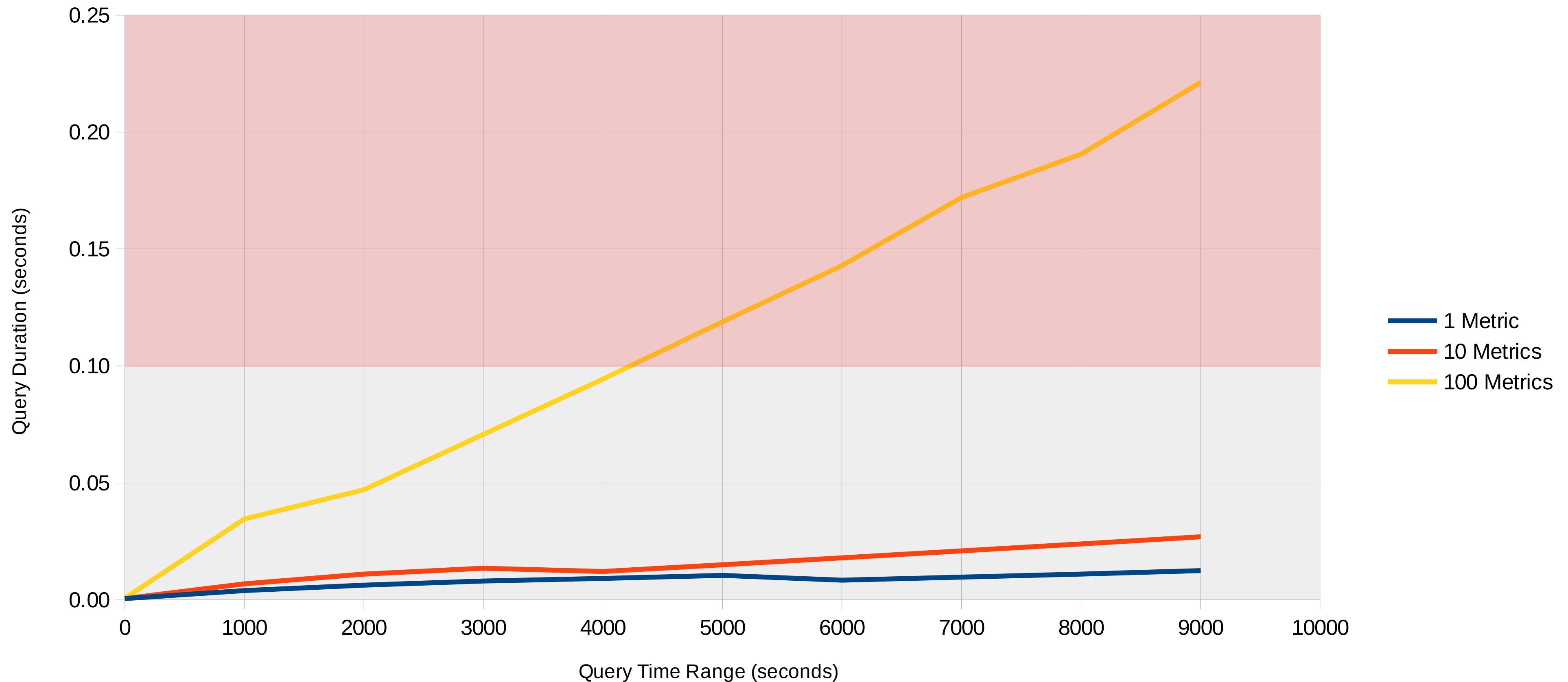
Indexing – Single Series Query (vs Data Volume)



Indexing – Single Series Query (vs Time Range, 10M Rows)



Indexing – Single Series Query (vs Time Range, 10M Rows)



Indexing - Analysis

- ✔ **Data Volume**
  - ✔ To 10M
- ✔ **Time Range**
  - ✔ To 9000s (10 Metrics)
- ✔ **Query time stable as Data Volume increases**

- ✗ **Time Range**
  - ✗ Over 4000s (100 Metrics)
- ✗ **Now apparent query duration increases as Time Range grows**
- ✗ **Increasing number of metrics drastically affects query duration**
  - ✗ Data for each uninteresting series must be filtered out

Indexing - Single Series Query

```
SELECT
  TIME_ROUND(timestamp, 60),
  AVG(value)
FROM
  measurements
WHERE
  timestamp BETWEEN
    '2015-01-01Z00:00:00' AND
    '2015-01-01Z01:00:00'
  AND name =
    'cpu.percent'
  AND dimensions @>
    '{"host": "dev-01"}'::JSONB
GROUP BY
    1
```

- **More indexing?**
  - name
  - dimensions

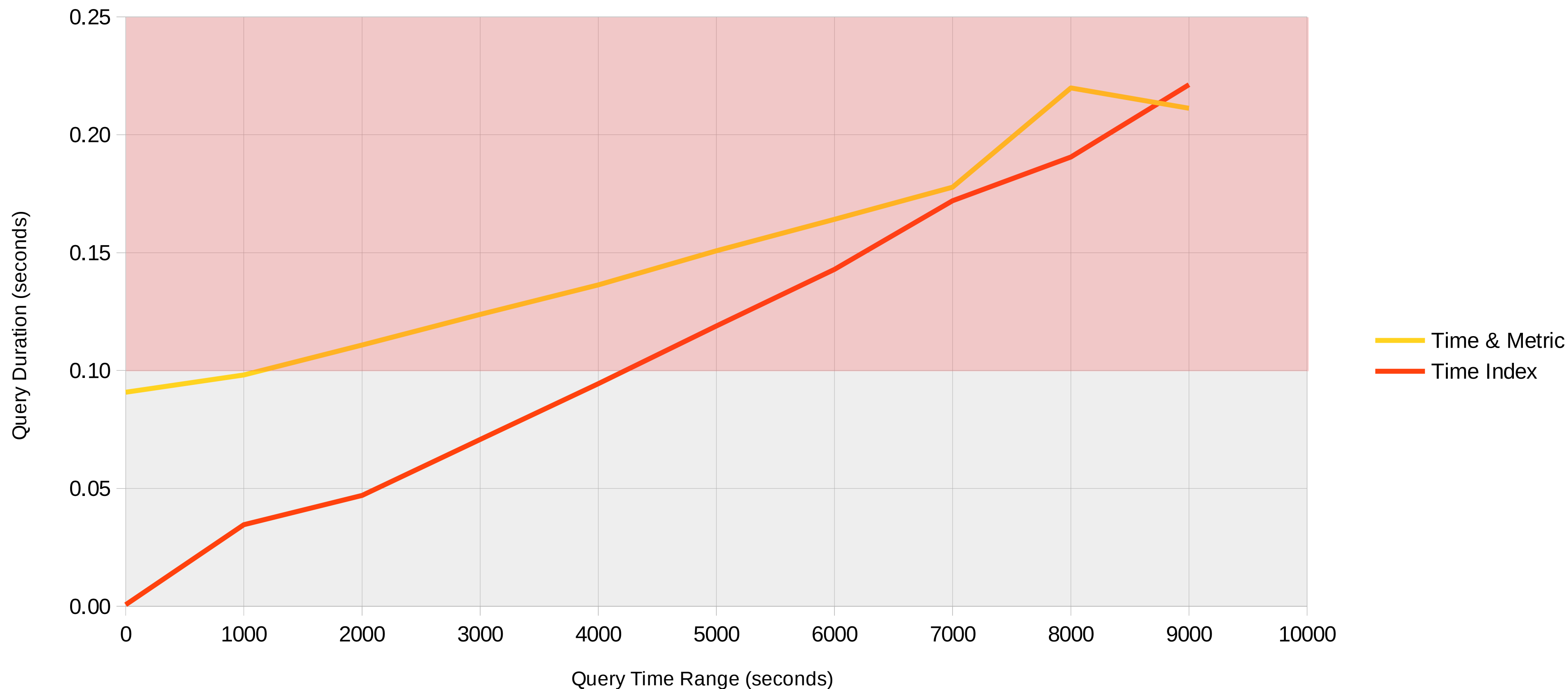
Indexing – Additional

```
CREATE INDEX ON
   measurements
   USING BTREE
   (name);

CREATE INDEX ON
   measurements
   USING GIN
   (dimensions);
```

- **Create new indexes on measurements table**
- **Specify *name***
  - Equality: Use BTREE
- **Specify *dimensions***
  - Containment: Use GIN
  - Find contents of JSON

Indexing – Series Query (vs Time Range, 10M Rows, 100 Metrics)



Indexing

✗ Oh dear

✗ Too much indexing can be harmful

Normalisation

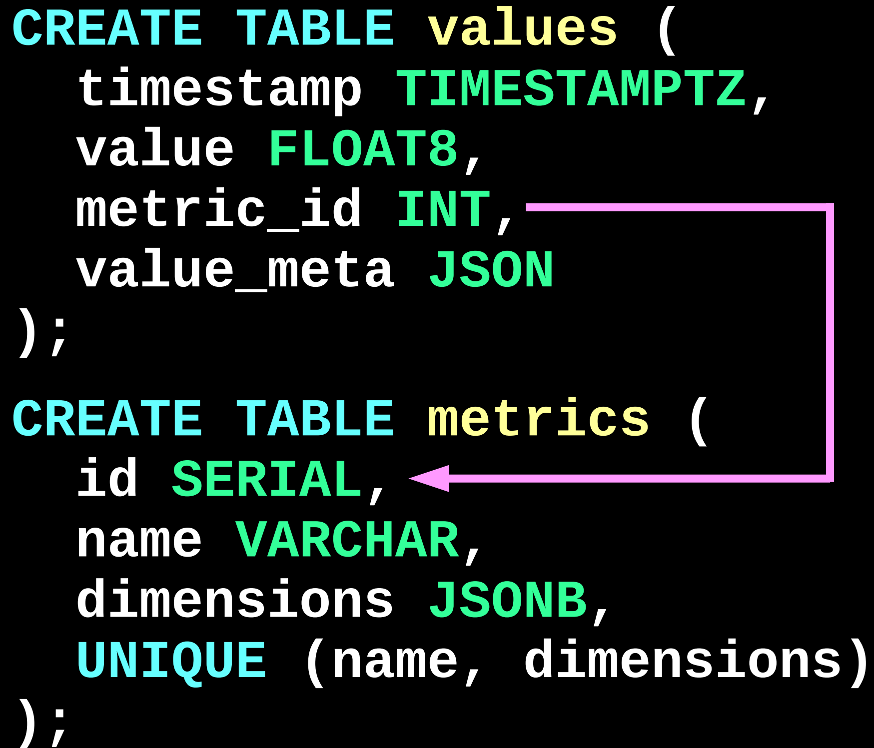
Normalisation

```sql
CREATE TABLE measurements (
  timestamp TIMESTAMPTZ,
  value FLOAT8,
  name VARCHAR,
  dimensions JSONB,
  value_meta JSON
);
```

```sql
CREATE TABLE values (
  timestamp TIMESTAMPTZ,
  value FLOAT8,
  metric_id INT,
  value_meta JSON
);

CREATE TABLE metrics (
  id SERIAL,
  name VARCHAR,
  dimensions JSONB,
  UNIQUE (name, dimensions)
);
```

Normalisation



- **Values stored by integer *id***
  - References entry in metric table
  - The name/dimensions for each metric are only stored once
  - Eliminates repeated bulky data in measurements table
- **Metric table defines *id***
  - SERIAL produces incrementing integers to allot id values
  - UNIQUE constraint is useful during normalisation
    - Implicitly creates suitable index

Normalisation – View

```
CREATE VIEW measurements
AS
  SELECT
    timestamp,
    value,
    name,
    dimensions,
    value_meta
  FROM
    values
    INNER JOIN
    metrics
    ON (metric_id = id);
```

- **Mimic *measurements***
  - Views can be queried in the same way as tables
- **Defined with SELECT**
  - Query to run which produces contents of view
- **Join normalised tables**
- **Can re-use same queries as before**

Normalisation - View Insert

```
CREATE RULE measurements_insert
  AS ON INSERT TO measurements
  DO INSTEAD
    INSERT INTO values (
      timestamp,
      value,
      metric_id,
      value_meta
    ) VALUES (
      NEW.timestamp,
      NEW.value,
      create_metric (
        NEW.name,
        NEW.dimensions),
      NEW.value_meta
    );
```

- **Can’t insert data into views by default**
- **Can specify an action to perform on INSERT**
- **Insert into *values***
- **Helper procedure to allocate *metric_id***
- **Normalisation is transparent for user**

Normalisation – Metric Lookup

```
CREATE FUNCTION create_metric (
  in_name VARCHAR,
  in_dims JSONB
) RETURNS INT LANGUAGE plpgsql AS $_$
DECLARE
  out_id INT;
BEGIN
  SELECT id INTO out_id
  FROM metrics AS m
  WHERE m.name = in_name AND
        m.dimensions = in_dims;
  IF NOT FOUND THEN
    INSERT INTO metrics
    ("name", "dimensions")
    VALUES (in_name, in_dims)
    RETURNING id INTO out_id;
  END IF;
  RETURN out_id;
END; $_$;
```

- **Stored procedure**
  - Take name/dimensions
  - Returns metric_id
- **Find existing metric**
  - Return existing id
- **If new, then INSERT**
  - Allocates new id
  - Return the new id

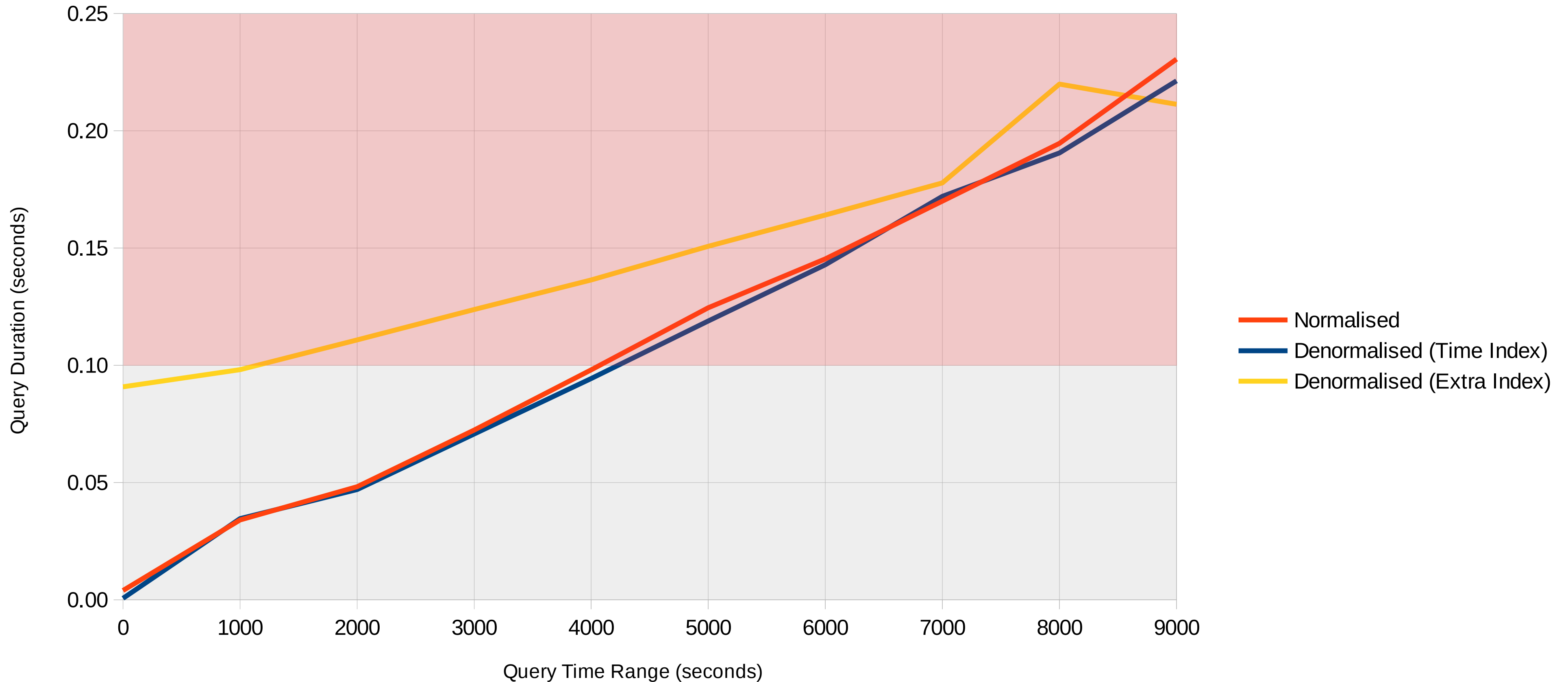
Normalisation - Indexing

```
CREATE INDEX ON
  values
  USING BTREE
  (timestamp);

CREATE INDEX ON
  values
  USING BTREE
  (metric_id);
```

- **Timestamp index**
  - Same as before
- **New index on metric_id**
  - Allow efficient filtering of metrics during JOIN
  - Serves similar purpose to existing metric indexing

Normalisation – Series Query (vs Time, 10M Rows, 100 Metrics)

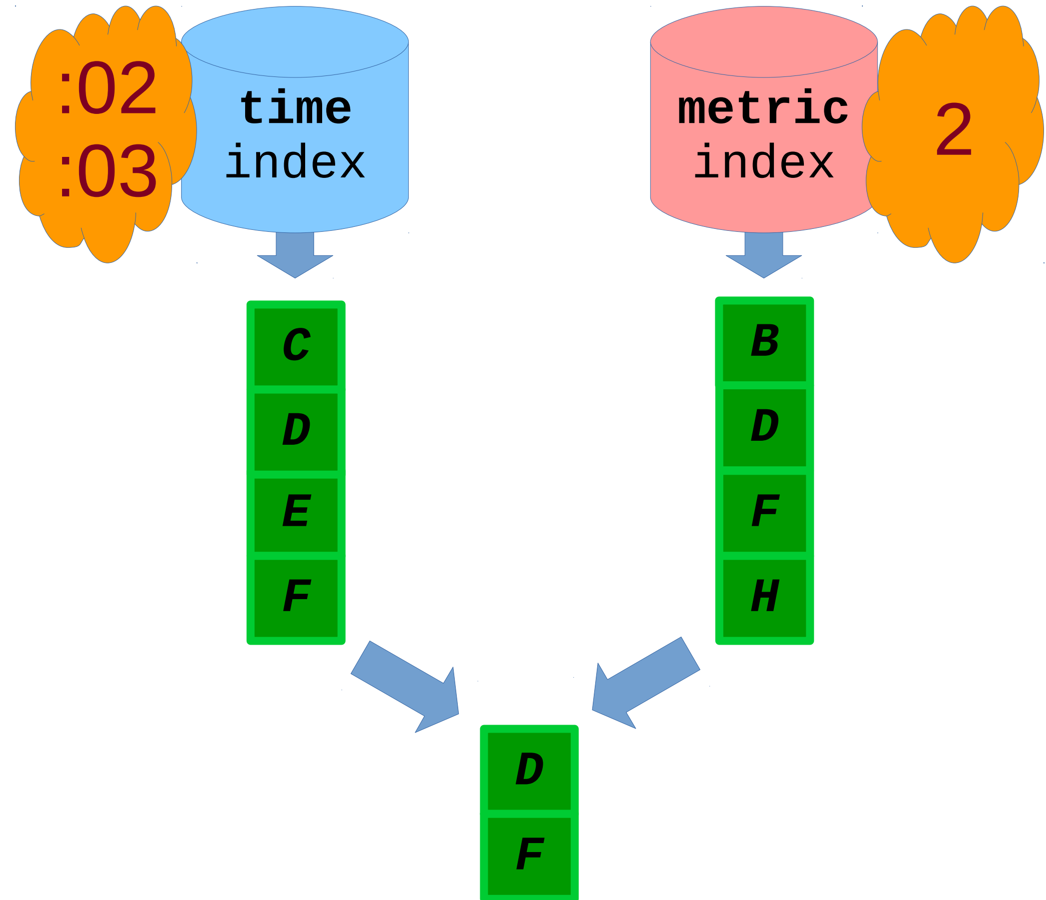


Normalisation

- ✔ Normalisation eliminated overhead of additional metric indexing
- ✗ The metric indexing still doesn’t have a positive effect

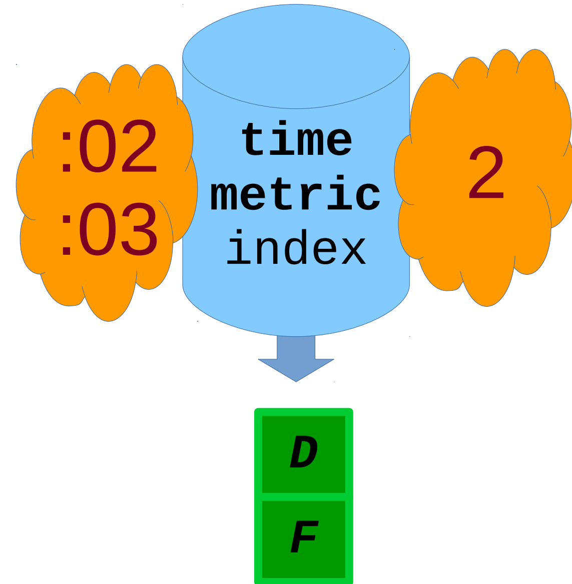
Normalisation – Bitmap Index Scan

	time	*value*	*metric*
A	10:01	.	1
B	10:01	.	2
C	10:02	.	1
D	10:02	.	2
E	10:03	.	1
F	10:03	.	2
G	10:04	.	1
H	10:04	.	2



Normalisation – Multi-Column Indexing

	time	*value*	*metric*
A	10:01	.	1
B	10:01	.	2
C	10:02	.	1
D	10:02	.	2
E	10:03	.	1
F	10:03	.	2
G	10:04	.	1
H	10:04	.	2



Normalisation – Multi-Column Indexing

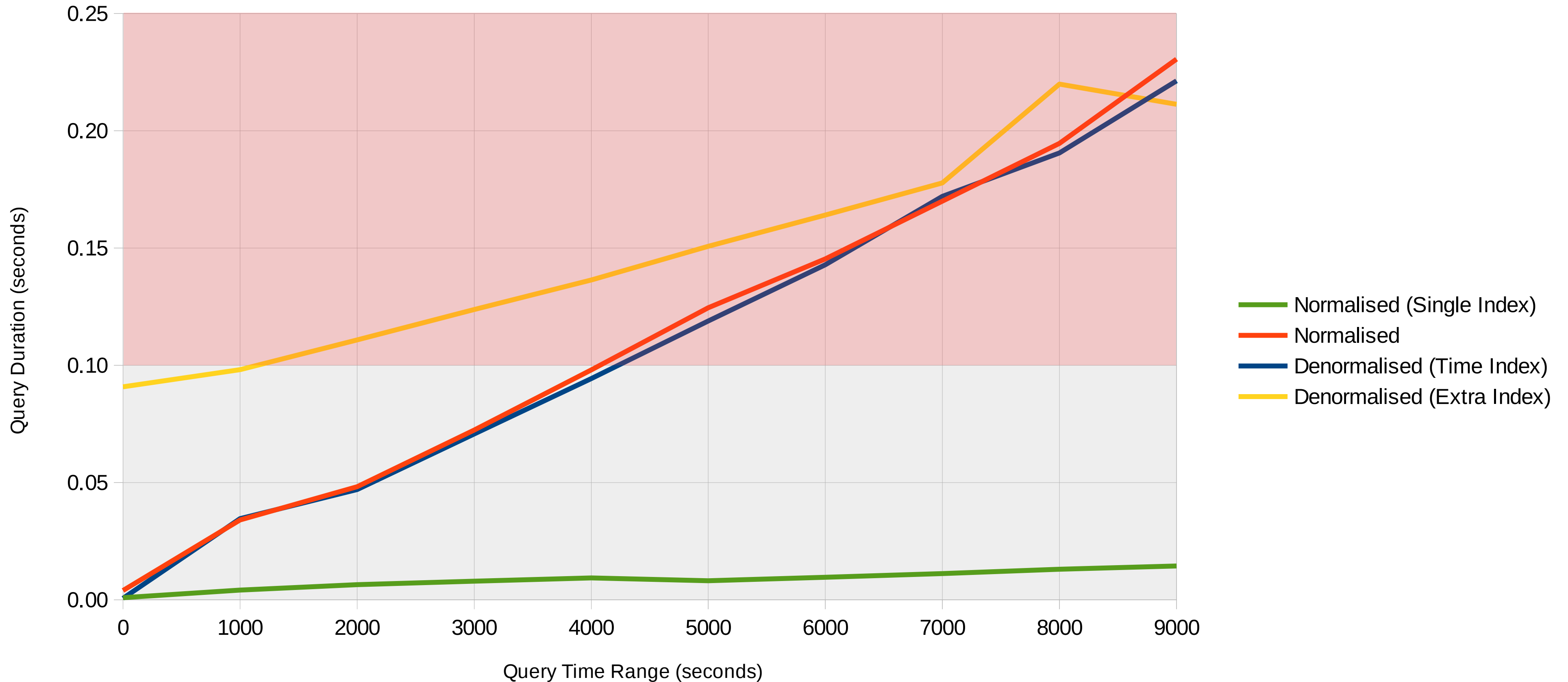
```sql
CREATE INDEX ON
  values
  USING BTREE
  (timestamp);

CREATE INDEX ON
  values
  USING BTREE
  (metric_id);
```

```sql
CREATE INDEX ON
  values
  USING BTREE
  (timestamp, metric_id);
```

```sql
CREATE INDEX ON
  values
  USING BTREE
  (metric_id, timestamp);
```

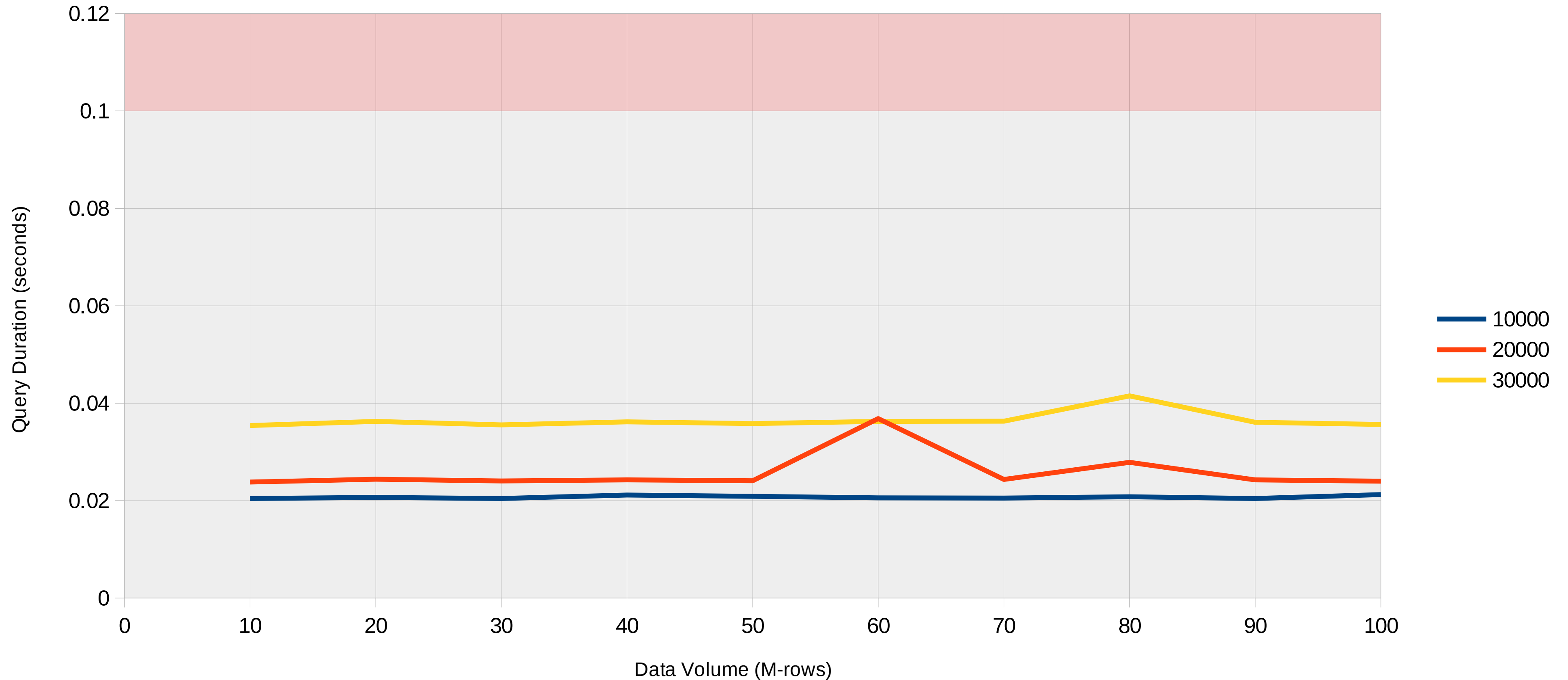
Normalisation – Series Query (vs Range, 10M Rows, 100 Metrics)



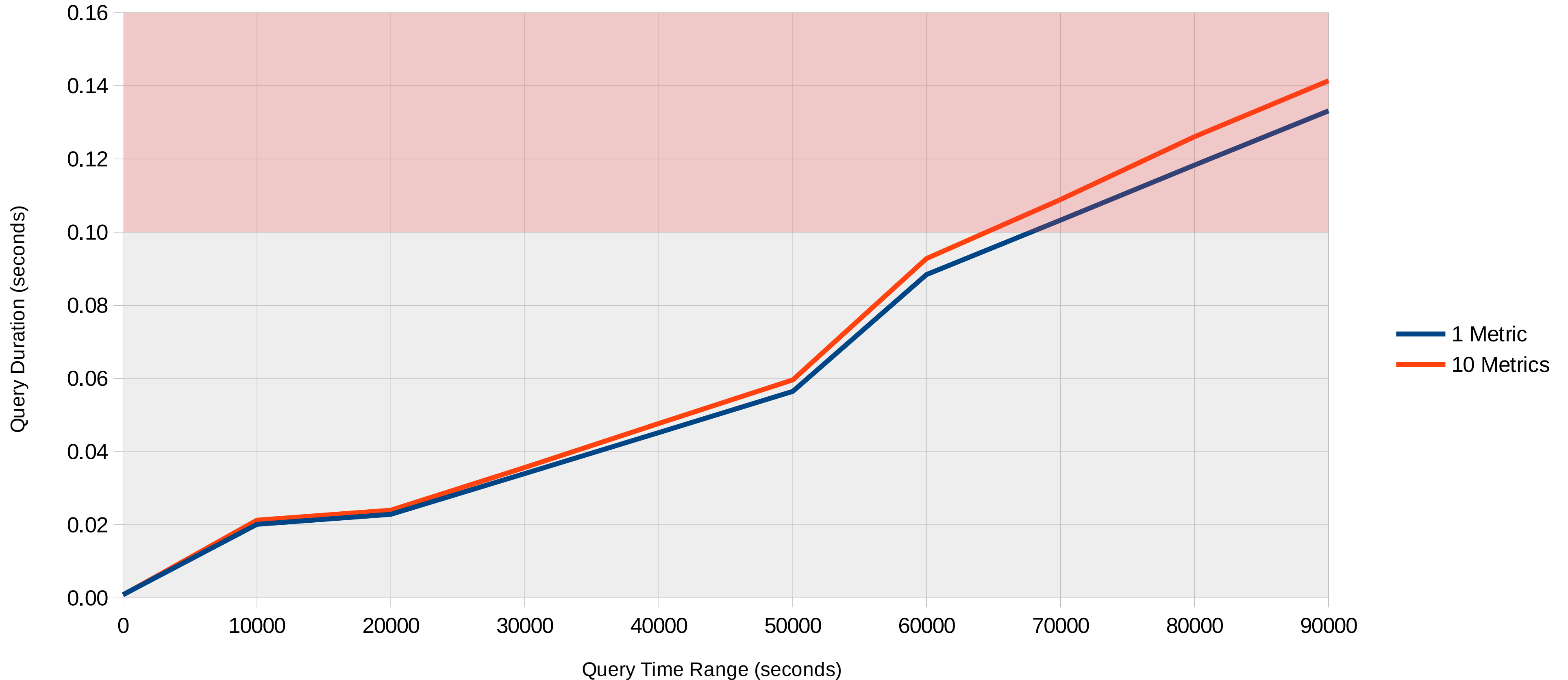
Normalisation

- **Increase volume 10M to 100M**
  - @ 1Hz / 100 Metrics: 1M seconds
    - Before: ~1.15 days
    - Now: ~11.5 days
- **Increase max time ranges from 9000s to 90,000s**
  - Before: 2.5 hours
  - Now: 1.04 days

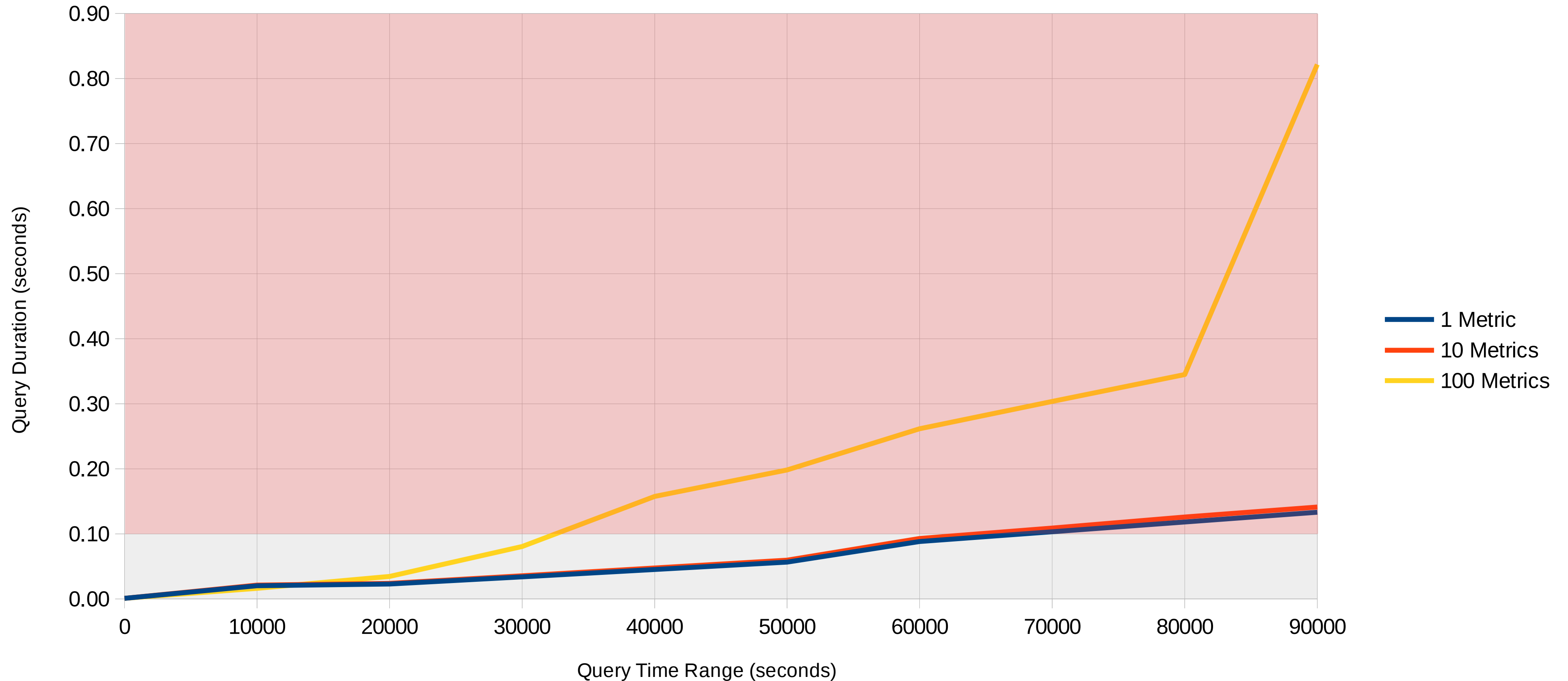
Normalisation – Series Query (vs Volume, 10 Metrics)



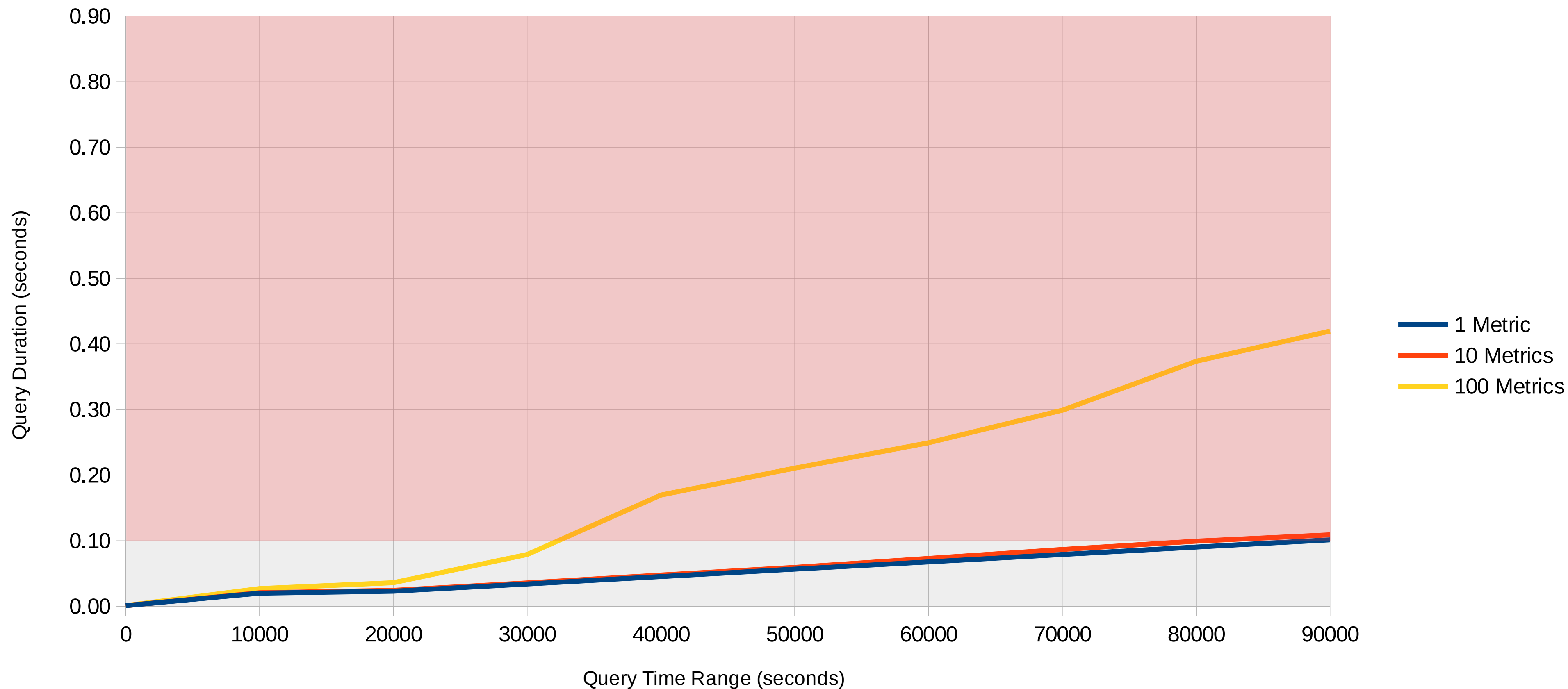
Normalisation – Series Query (vs Range, 100M Rows)



Normalisation – Series Query (vs Range, 100M Rows)



Normalisation – Series Query (vs Range, 100M Rows) (+Config)



Normalisation - Analysis

- ✔ **Data Volume**
  - ✔ To 100M
- ✔ **Time Range**
  - ✔ To 90,000s (10 Metrics)

- ✗ **Time Range**
  - ✗ Over 30,000s (100 Metrics)
  - ✗ Over 90,000s
- ✗ **Need a better strategy for servicing larger time ranges**

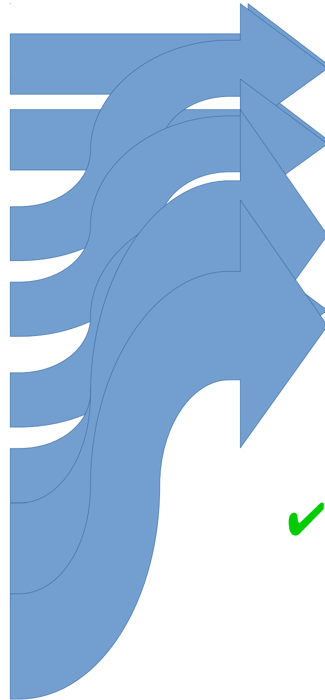
Summarising

Summarising - Problem

- **For 100 metrics, some queries miss target 100ms**
  - Over ~40Ks (~11 days)
- **Query might be returning up to 40K points**
- **Is this actually necessary?**
  - Especially if data is simply used for visualisation
  - An average 1080p monitor only has ~2000 pixels
- **Lets say 4000 points are enough, or even 400**

Summarising - Example

values		
time	*value*	*metric*
10:00	10	1
10:00	2	2
10:01	20	1
10:01	6	2
10:02	5	1
10:02	4	2
10:03	15	1
10:03	1	2



values_2		
time	*sum*	*metric*
10:00	30	1
10:00	8	2
10:02	20	1
10:02	5	2

- ✔ **Summary table just a fraction of the size**

Summarising

```
CREATE TABLE values_10 (
  timestamp TIMESTAMPTZ,
  metric_id INT,
  sum FLOAT8,
  count FLOAT8,
  min FLOAT8,
  max FLOAT8,

  UNIQUE (metric_id,
          timestamp)
);
```

- **Create values table**
  - Use for 10:1 summary
- **One entry/time period**
  - Per metric
  - UNIQUE provide indexing
- **Multiple aggregates**
  - SUM
  - COUNT
  - MIN
  - MAX

Summarising

```
CREATE VIEW summary_10 AS
  SELECT *
  FROM
    values_10
    INNER JOIN
    metrics
    ON (metric_id = id);
```

- **Create a view as before**
  - Only storing metric_id
- **Simplifies queries**
- **Joins metric definitions**

Summarising – Trigger Definition

```
CREATE FUNCTION
  summarise_10 ()
  RETURNS TRIGGER
  LANGUAGE plpgsql AS $_$
BEGIN
  :
END; $_$;

CREATE TRIGGER summarise_10_t
  AFTER INSERT ON values
  FOR EACH ROW
  EXECUTE PROCEDURE
    summarise_10 ();
```

- **Boilerplate**
- **Define trigger function**
  - Stored procedure
  - Contents omitted
- **Trigger to execute…**
  - On INSERT
  - To *values* table
  - Data passed to procedure

Summarising – Trigger Action

```
INSERT INTO values_10 VALUES (
  TIME_ROUND(NEW.timestamp, 10),
  NEW.metric_id,
  NEW.value,
  1,
  NEW.value,
  NEW.value
)
ON CONFLICT (metric_id,
             timestamp)
DO UPDATE SET
  sum   = sum   + EXCLUDED.sum,
  count = count + EXCLUDED.count,
  min = LEAST  (min,EXCLUDED.min),
  max = GREATEST(max,EXCLUDED.max)
;
```

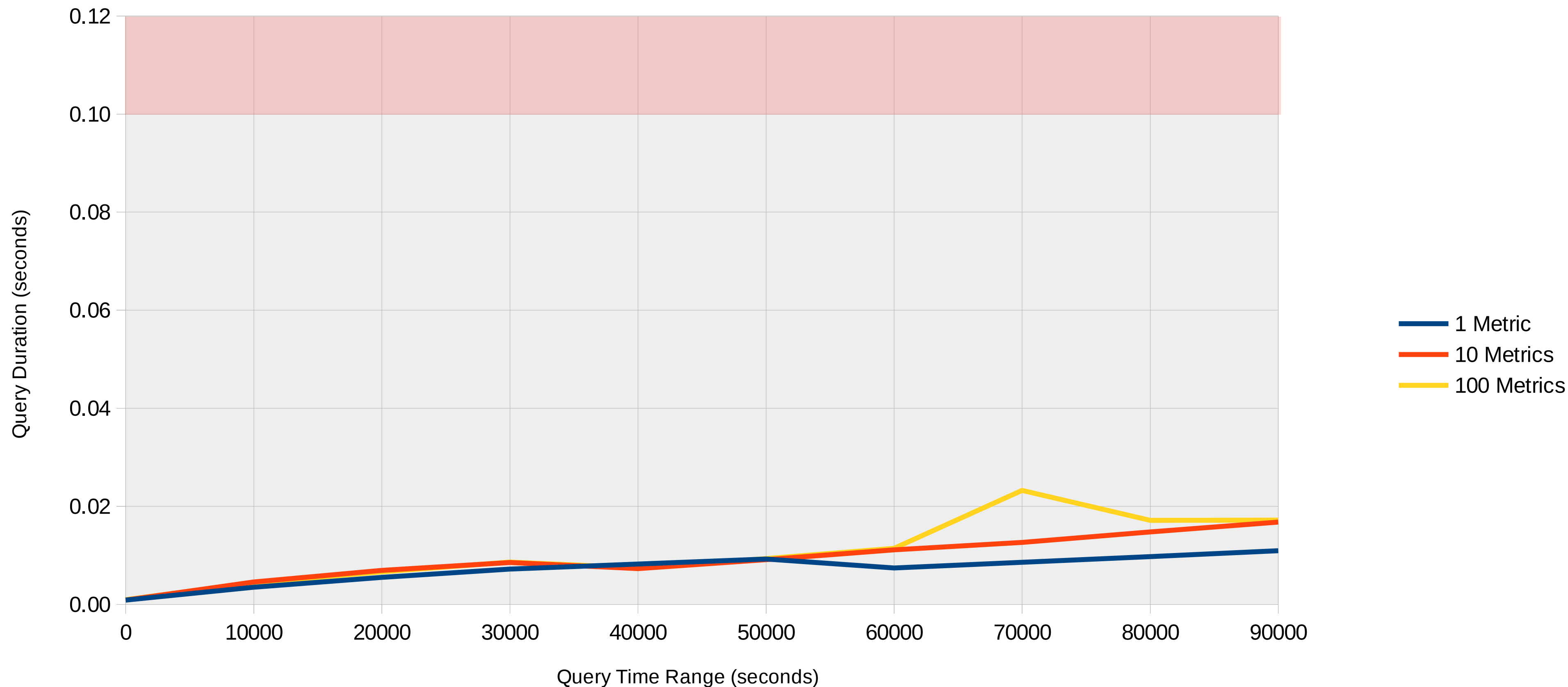
- **Insert into summary**
  - *NEW* is inserted data
- **Round time to period**
  - 10 seconds
- **Initial aggregate values**
- **If entry exists already**
- **Update instead**
  - *EXCLUDED* is current row
  - Combine new value with existing aggregate value

Summarising – Single Series Query

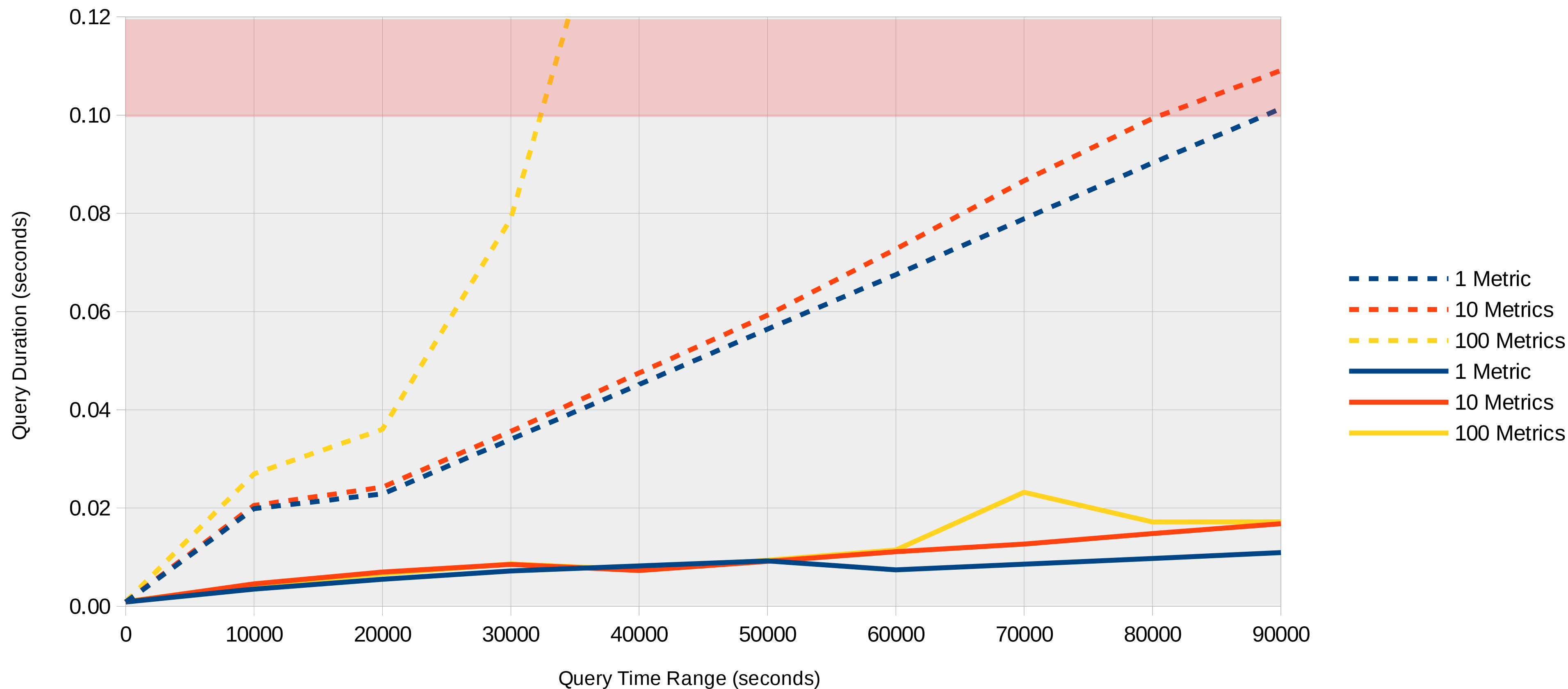
```
SELECT
  TIME_ROUND(timestamp, 60),
  (SUM(sum) / SUM(count)) AS avg
FROM
  summary_10
WHERE
  timestamp BETWEEN
    '2015-01-01Z00:00:00' AND
    '2015-01-01Z01:00:00'
  AND name =
    'cpu.percent'
  AND dimensions @>
    '{"host": "dev-01"}'::JSONB
GROUP BY
    1
```

- **Mostly unchanged**
- **Query summary table, not raw measurements**
- **Have to aggregate the partial aggregations**
  - `MIN: MIN(min)`
  - `MAX: MAX(max)`
  - `SUM: SUM(sum)`
  - `COUNT: SUM(count)`
  - `AVG: SUM(sum)/SUM(count)`

Summarising – Series Query (vs Range, 100M Rows)



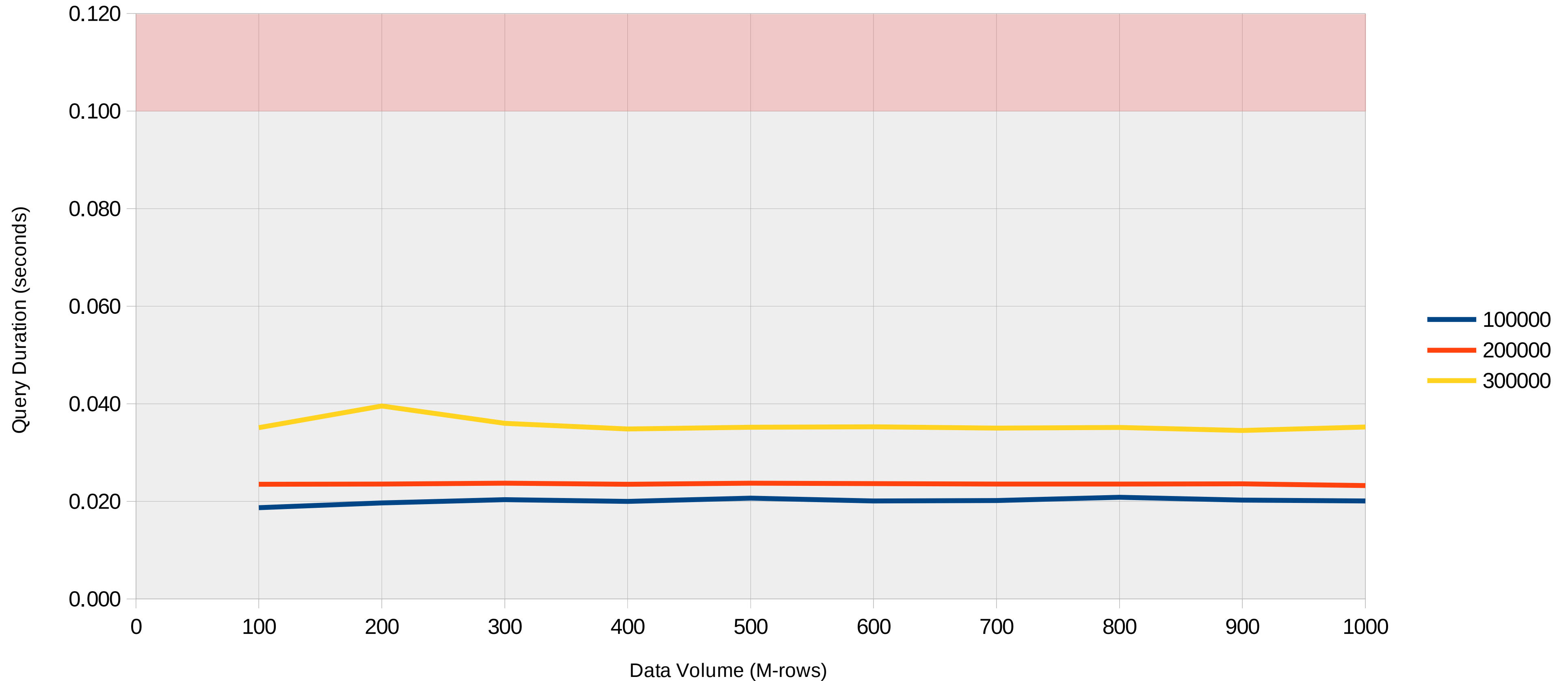
Summarising – Series Query (vs Range, 100M Rows)



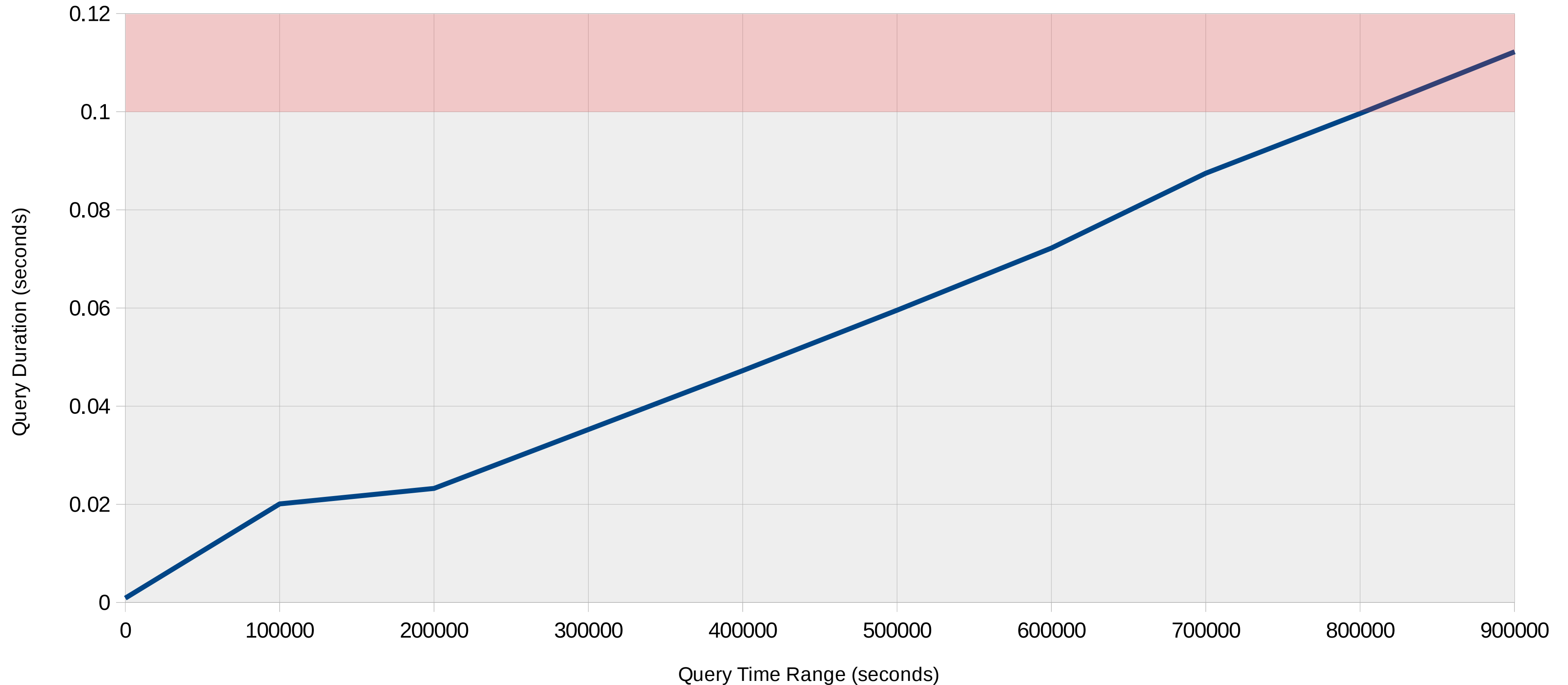
Summarising

- **Increase volume to 1BN**
  - @ 1Hz / 100 Metrics: 10M seconds
  - Before: 11.5 days
  - Now: ~115 days : 16½ weeks
- **Increase max time ranges from 90Ks to 900Ks**
  - Before: ~1.04 days
  - Now: ~10.4 days

Summarising – Series Query (vs Volume; 100M-1BN)



Summarising – Series Query (vs Range, 1BN Rows)



Summarising

- ✔ **Data Volume**
  - ✔ To 1BN – ~16 weeks
- ✔ **Time Range**
  - ✔ To ~10 days

- ✔ **To scale further? Try 100:1 summary**

Closing Notes

What Next?

- **Partitioning**
  - By timestamp and metric_id
  - Retention window (i.e. keep last 6 months)
- **Ingest optimisation**
  - Compute aggregates in batches (e.g. every 5min)
  - Non trigger-based summarisation
  - Possibility to abuse logical replication

Is It Worth It?

- **Yes**
  - *Reduce complexity* in terms of number of technologies
  - Reuse existing infrastructure and operational knowledge
  - No loss in consistency – ACID all the way
- **No?**
  - *Increase complexity* in terms of database design
  - Is an out-of-the box tool already available?

Thanks

steve@smpsn.net