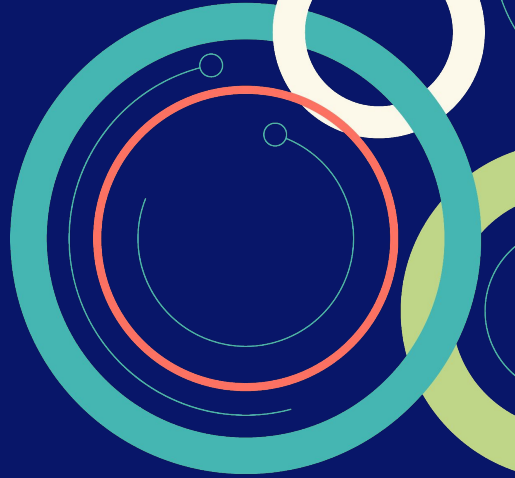


The Speed of Metrics, The Fidelity of Traces:

Architecting Post Collection Aggregation

Zack Owens
Principal Software Engineer, New Relic

Telemetry Fidelity is a Choice

What to measure is essential to observability practice

When you're in an incident or want to do long range analysis, these choices matter!

Today's problem: the choice you make at collection dictates what questions your observability practice can answer

Choice of Fidelity

Metrics

Cost effective to store

(Usually) fast to query or alert

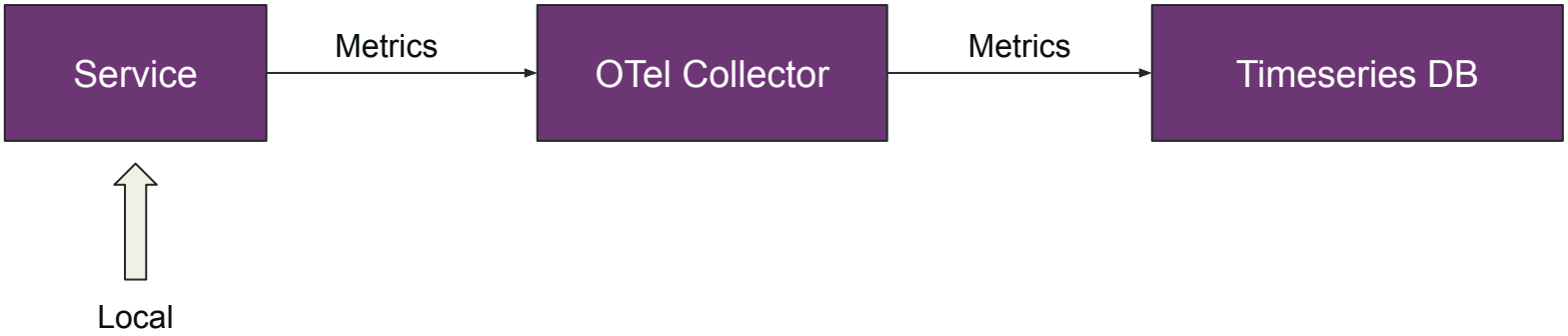
Problem: lacks the dimensions necessary to answer any observability question

“Raw Data” (Events/Logs/Spans/Traces)

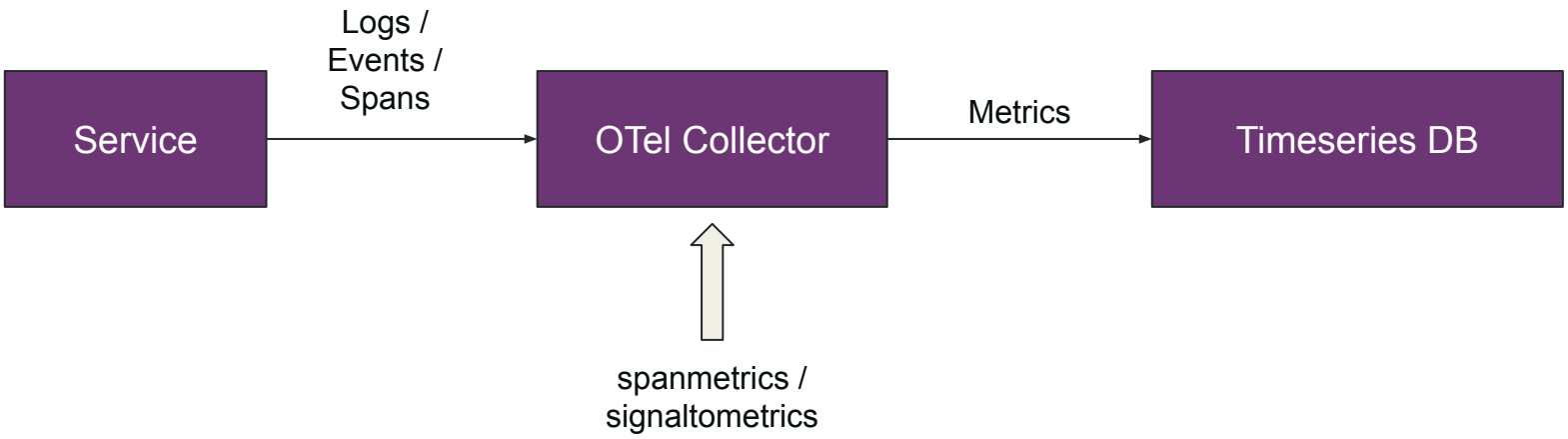
Rich context to answer ad-hoc questions

Problem: SLOW to query with large volumes of data

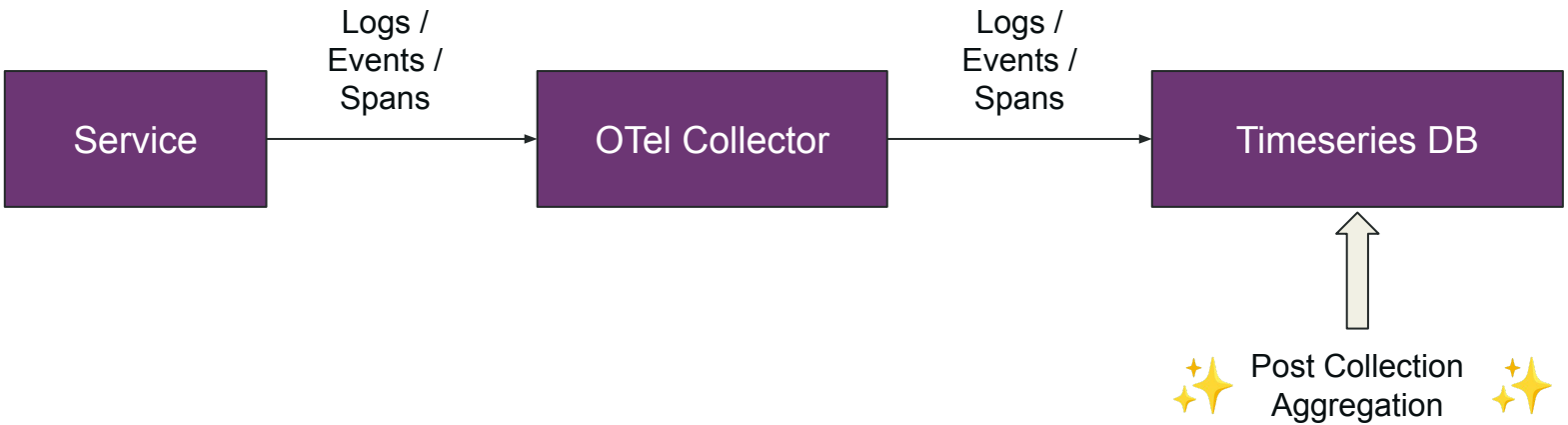
Where do we aggregate?



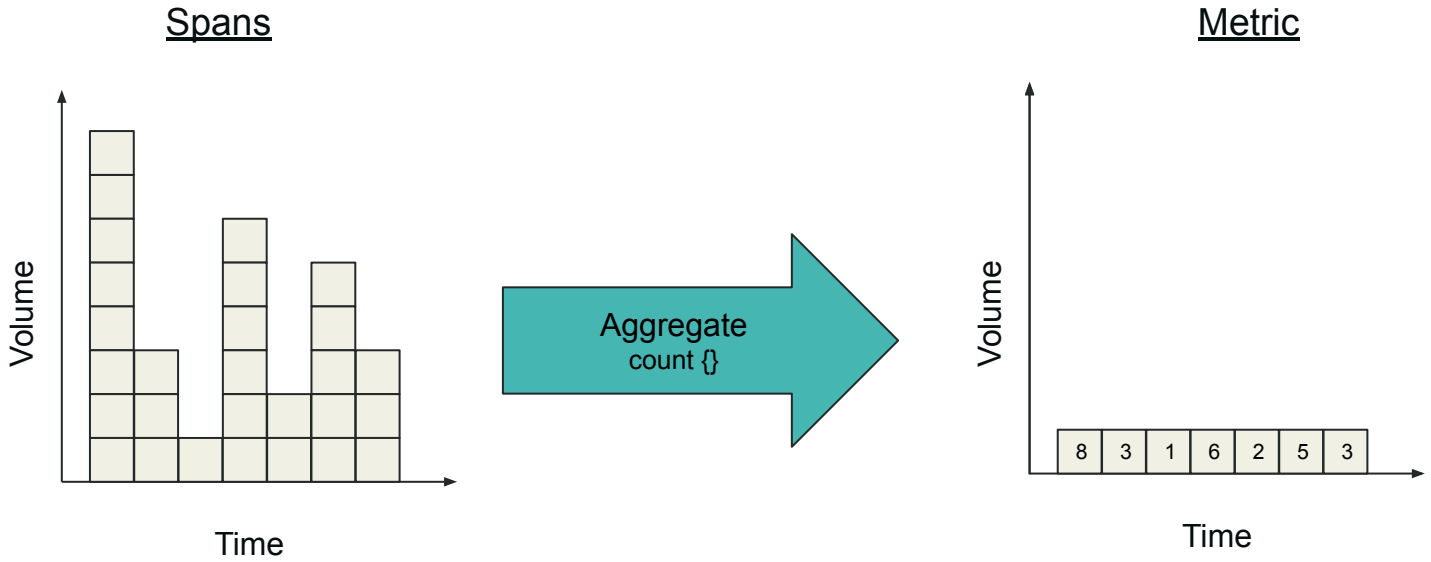
Where do we aggregate?



Where do we aggregate?

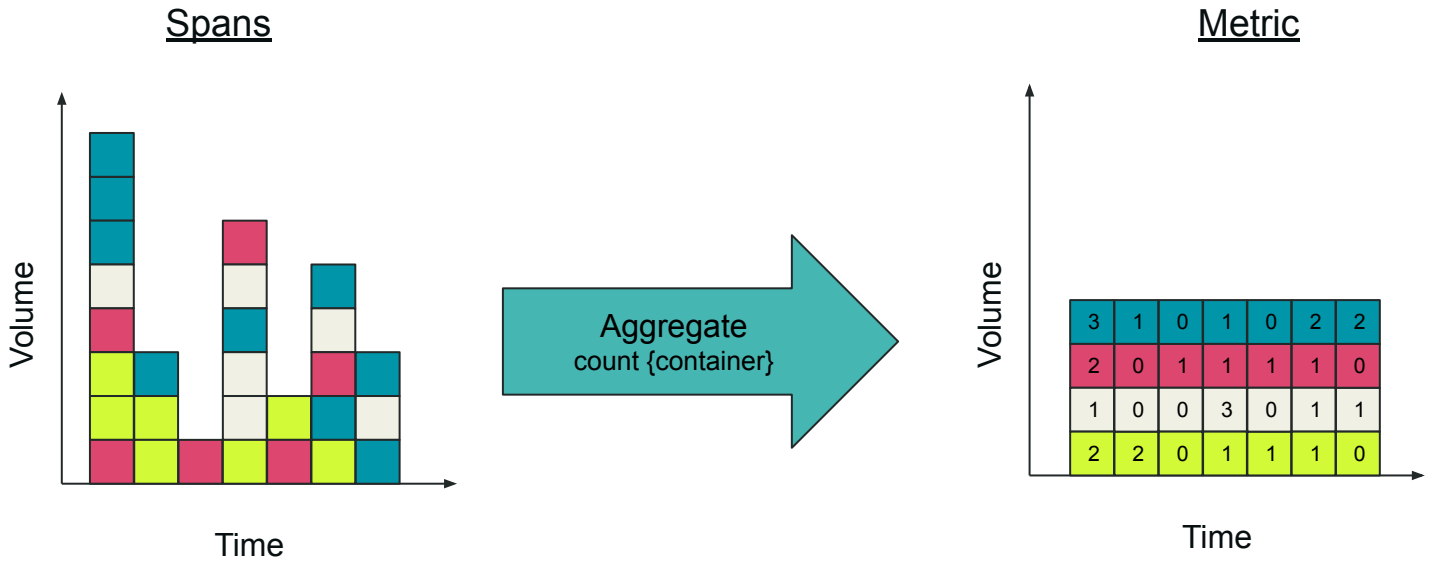


What is aggregation anyway?



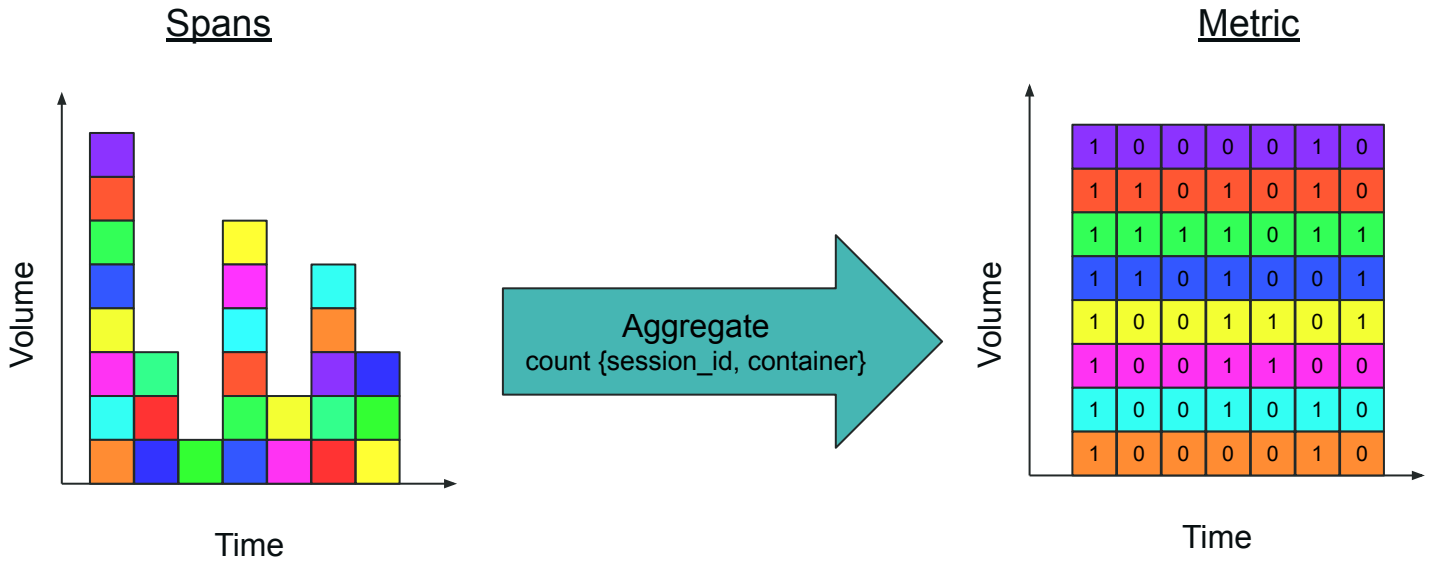
What is aggregation anyway?

With Dimensions



What is aggregation anyway?

High Cardinality



High Cardinality

High Cardinality has a price (storage + memory to aggregate)

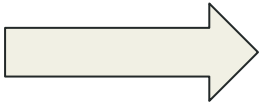
Might be better off storing raw data instead

Most cases: cardinality reduction for speed!

Mindset Shift

Before

End-to-end aggregation with reduction in dimensionality



After

Aggregation as a performance optimization over underlying data, keep high cardinality data

Advantages of Post Collection

Keep high fidelity raw data for “interesting” observability questions

Measure cardinality *before* layering on aggregation

Leading Timeseries DBs and Observability Vendors automatically apply aggregation

Post Collection Aggregation Options

"Events to Metrics"

Create metrics behind the scenes by aggregating raw telemetry into metrics

User decides at query time to use aggregates instead of raw data

Examples:

Clickhouse: Incremental Materialized Views

New Relic: Events to Metrics

Automatic Materialized View

Similar to indexes, aggregations build behind the scenes on raw data table

Database decides to use the aggregates at query time (no query syntax change!)

Examples:

Clickhouse: Projections

New Relic: Accelerated Views

Post Collection Aggregation

“Events to Metrics”

Use when your retention of aggregates exceeds retention of raw data

Can be useful for metric standardization

Automatic Materialized View

Default choice to accelerate common dashboards/alerts

Starting point to “drill into” data without switching contexts between aggregates and raw

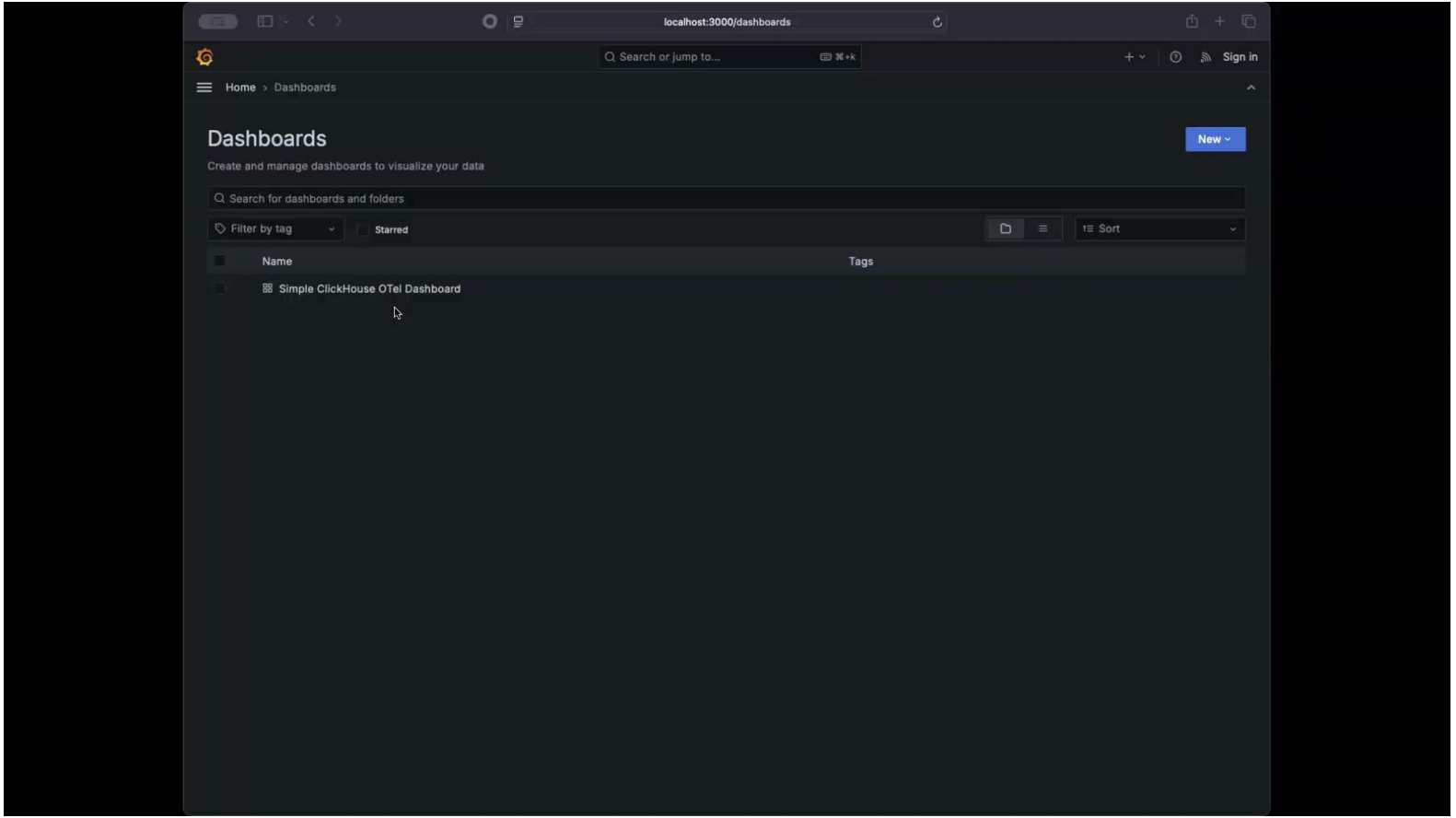
Ex: Clickhouse Projections


```
CREATE TABLE IF NOT EXISTS otel_traces
(
    Timestamp          DateTime64(9)                      CODEC(Delta, ZSTD(1)),
    TraceId            String                             CODEC(ZSTD(1)),
    SpanId             String                             CODEC(ZSTD(1)),
    ParentSpanId       String                             CODEC(ZSTD(1)),
    TraceState         String                             CODEC(ZSTD(1)),
    SpanName           LowCardinality(String)             CODEC(ZSTD(1)),
    SpanKind           LowCardinality(String)             CODEC(ZSTD(1)),
    ServiceName        LowCardinality(String)             CODEC(ZSTD(1)),
    ResourceAttributes Map(LowCardinality(String), String) CODEC(ZSTD(1)),
    ScopeName          String                             CODEC(ZSTD(1)),
    ScopeVersion       String                             CODEC(ZSTD(1)),
    SpanAttributes     Map(LowCardinality(String), String) CODEC(ZSTD(1)),
    Duration           Int64                              CODEC(ZSTD(1)),
    StatusCode         LowCardinality(String)             CODEC(ZSTD(1)),
    StatusMessage      String                             CODEC(ZSTD(1)),

    Events Nested (Timestamp  DateTime64(9), Name LowCardinality(String), Attributes Map(LowCardinality(String), String)) CODEC(ZSTD(1)),

    Links Nested (TraceId String, SpanId String, TraceState String, Attributes Map(LowCardinality(String), String)) CODEC(ZSTD(1)),

    INDEX idx_trace_id        TraceId                    TYPE bloom_filter(0.001) GRANULARITY 1,
    INDEX idx_res_attr_key    mapKeys(ResourceAttributes)   TYPE bloom_filter(0.01)  GRANULARITY 1,
    INDEX idx_res_attr_value  mapValues(ResourceAttributes) TYPE bloom_filter(0.01)  GRANULARITY 1,
    INDEX idx_span_attr_key   mapKeys(SpanAttributes)       TYPE bloom_filter(0.01)  GRANULARITY 1,
    INDEX idx_span_attr_value mapValues(SpanAttributes)     TYPE bloom_filter(0.01)  GRANULARITY 1,
    INDEX idx_duration        Duration                   TYPE minmax              GRANULARITY 1
)
ENGINE = MergeTree()
PARTITION BY toDate(Timestamp)
ORDER BY (ServiceName, SpanName, toUnixTimestamp(Timestamp), TraceId)
TTL toDateTime(Timestamp) + INTERVAL 30 DAY
SETTINGS index_granularity = 8192, ttl_only_drop_parts = 1;
```

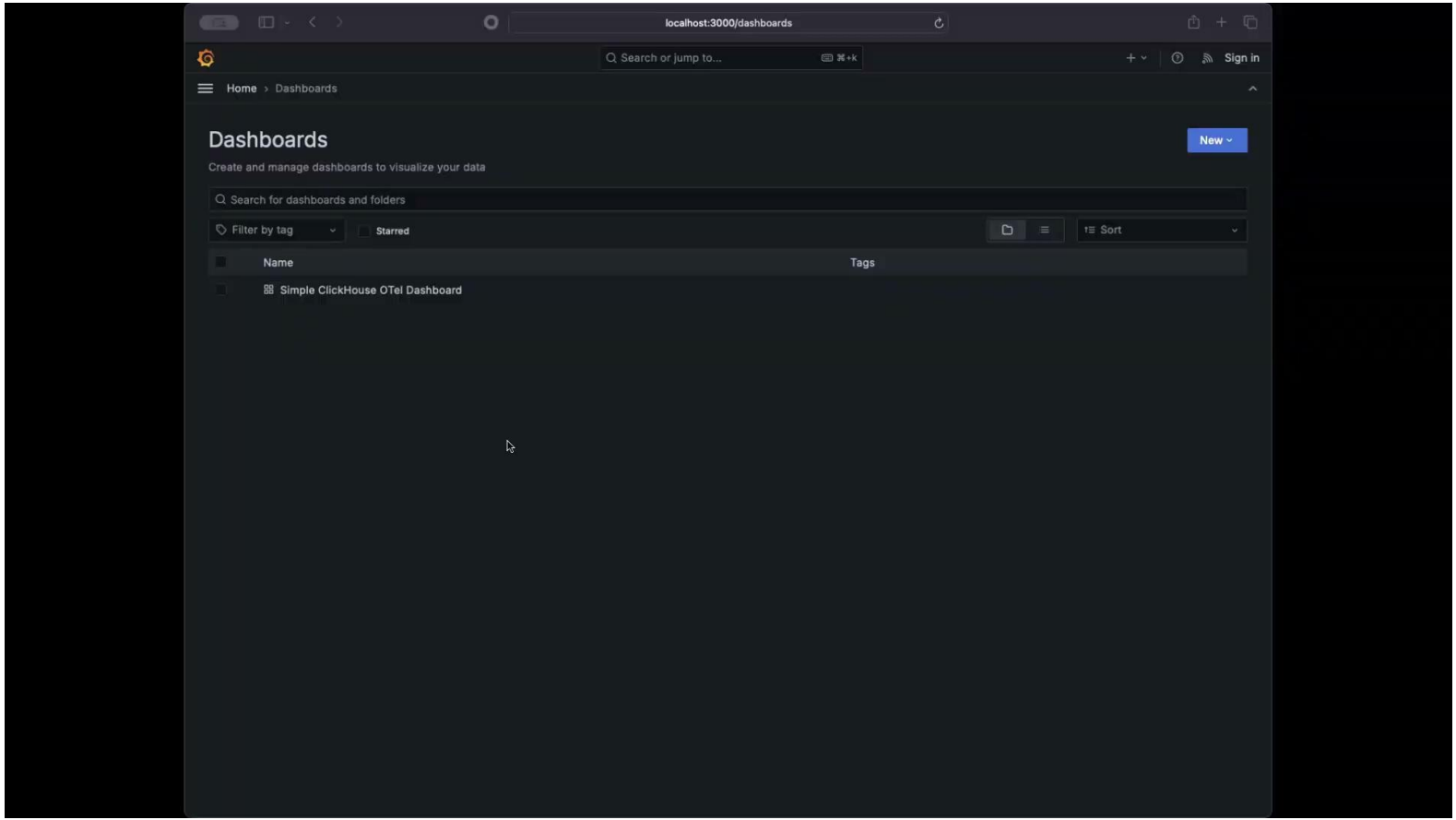

localhost:3000/dashboards
Search or jump to...
Sign in
Home › Dashboards
Dashboards
Create and manage dashboards to visualize your data
New
Search for dashboards and folders
Filter by tag
Starred
Sort
Name
Tags
Simple ClickHouse OTel Dashboard

Ex: Clickhouse Projections


```sql
-- add "minute" column to the table (simplifies aggregation)
ALTER TABLE otel_traces ADD COLUMN minute DateTime MATERIALIZED toStartOfMinute(Timestamp);
ALTER TABLE otel_traces MATERIALIZE COLUMN minute;

-- Add projection with count(), sum(Duration), p50, p95, p99
ALTER TABLE otel_traces ADD PROJECTION p_svc_minute
  (
      SELECT
          ServiceName,
          StatusCode,
          minute,
          count(),
          sum(Duration),
          quantile(0.5)(Duration),
          quantile(0.95)(Duration),
          quantile(0.99)(Duration)
      GROUP BY
          ServiceName, StatusCode, minute
  );

ALTER TABLE otel_traces MATERIALIZE PROJECTION p_svc_minute;
```


localhost:3000/dashboards
Search or jump to...
Sign in
Home › Dashboards
Dashboards
Create and manage dashboards to visualize your data
New
Search for dashboards and folders
Filter by tag
Starred
Sort
Name
Tags
Simple ClickHouse OTel Dashboard

```
EXPLAIN
SELECT
    minute AS time,
    ServiceName,
    quantile(0.99)(Duration) / 1000000. AS p99_ms
FROM otel_traces
WHERE minute >= (now() - toIntervalYear(6))
GROUP BY
    time,
    ServiceName
ORDER BY time ASC

Query id: 9bc83f04-330b-4a54-8be9-ba3c01ba1e10

    ┌─explain─────────────────────────────────────────────────────────────────┐
 1. │ Expression ((Project names + (Before ORDER BY + Projection) [lifted up part])) │
 2. │   Sorting (Sorting for ORDER BY)                                        │
 3. │     Expression ((Before ORDER BY + Projection))                         │
 4. │       Aggregating                                                       │
 5. │         Expression (Before GROUP BY)                                    │
 6. │           Expression                                                    │
 7. │             ReadFromMergeTree (default.otel_traces)                     │
    └─────────────────────────────────────────────────────────────────────────┘
```

```
EXPLAIN
SELECT
    minute AS time,
    ServiceName,
    quantile(0.99)(Duration) / 1000000. AS p99_ms
FROM otel_traces
WHERE minute >= (now() - toIntervalYear(6))
GROUP BY
    time,
    ServiceName
ORDER BY time ASC

Query id: b8873535-9a10-4d7d-8a0c-d6ea4092b8ec

    ┌─explain─────────────────────────────────────────────────────────────────┐
 1. │ Expression ((Project names + (Before ORDER BY + Projection) [lifted up part])) │
 2. │   Sorting (Sorting for ORDER BY)                                        │
 3. │     Expression ((Before ORDER BY + Projection))                         │
 4. │       Aggregating                                                       │
 5. │         Filter                                                          │
 6. │           ReadFromMergeTree (p_svc_minute)                              │
    └─────────────────────────────────────────────────────────────────────────┘
```

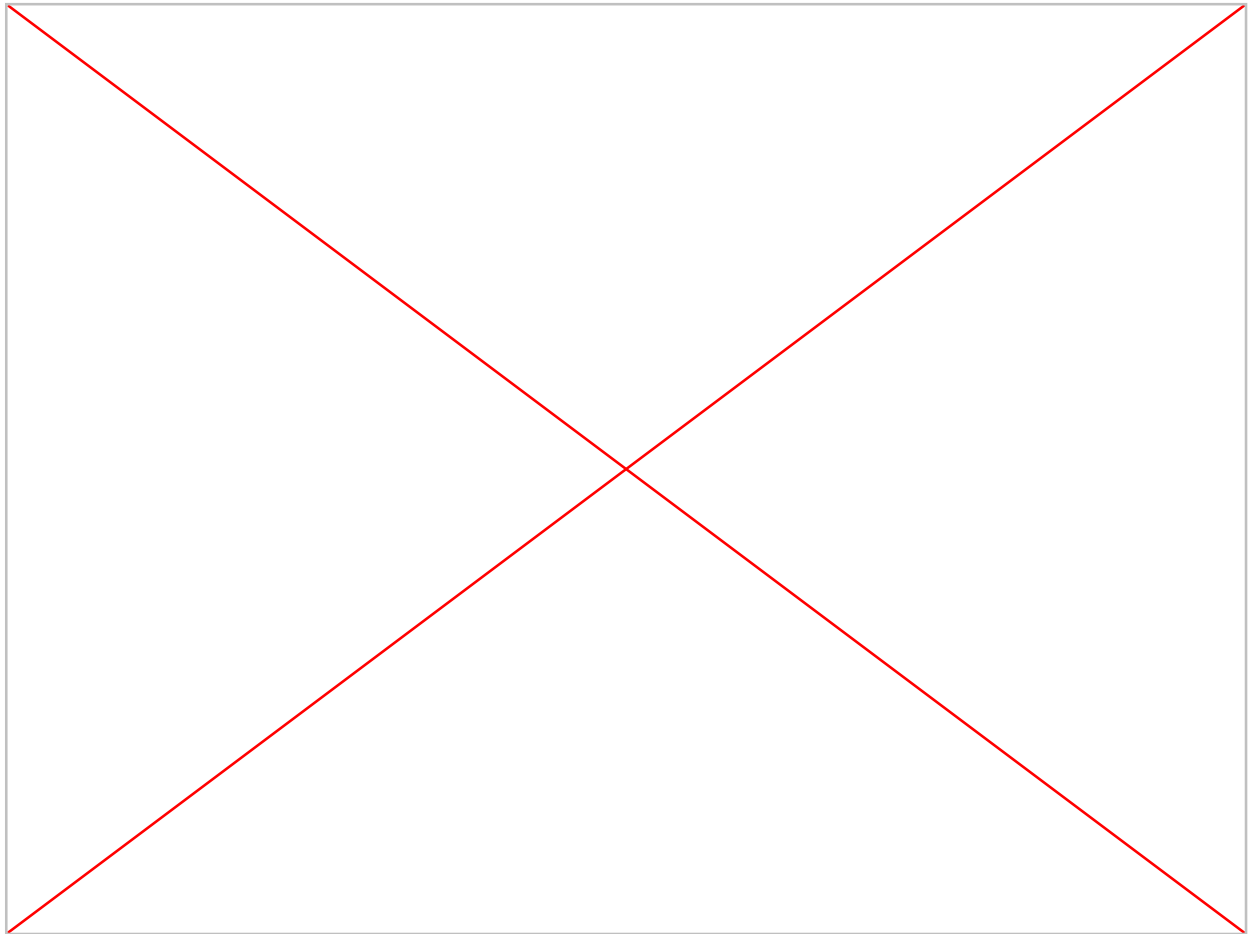
Performance Comparison

Before Projections

```
300 rows in set. Elapsed: 5.183 sec. Processed 360.00 million rows, 4.68 GB (69.45 million rows/s., 902.87 MB/s.)
Peak memory usage: 81.91 MiB.
```

After Projections

```
300 rows in set. Elapsed: 0.091 sec.
```



Measure Cardinality


```
0828bcd6562a :) SELECT
     count()                                                    AS raw_rows,

     -- Cardinality: ServiceName + StatusCode
     uniq(ServiceName, StatusCode)                              AS k_svc_status,
     k_svc_status * 360                                         AS proj_rows_svc_status,
     round(raw_rows / proj_rows_svc_status)                     AS shrink_svc_status,

     -- Cardinality: ServiceName + StatusCode + customer ID
     uniq(ServiceName, StatusCode, SpanAttributes['customer.id'])  AS k_plus_customer,
     k_plus_customer  * 360                                     AS proj_rows_plus_customer,
     round(raw_rows / proj_rows_plus_customer)                  AS shrink_plus_customer,

     -- Cardinality: ServiceName + Statuscode + user ID
     uniq(ServiceName, StatusCode, SpanAttributes['user.id'])   AS k_plus_user,
     k_plus_user      * 360                                     AS proj_rows_plus_user,
     round(raw_rows / proj_rows_plus_user)                      AS shrink_plus_user
FROM otel_traces
FORMAT Vertical;
```

Measure Cardinality

Raw Rows*: 3,600,000

```
Row 1:
──────
raw_rows:                  360000000
k_svc_status:              10
proj_rows_svc_status:      3600
shrink_svc_status:         100000
k_plus_customer:           10000
proj_rows_plus_customer:   3600000
shrink_plus_customer:      100
k_plus_user:               997416
proj_rows_plus_user:       359069760
shrink_plus_user:          1
```

Dimensions	Unique Groups	Projection Rows*	Reduction*
ServiceName, StatusCode	10	3,600	100,000:1
ServiceName, StatusCode, Customer ID	10,000	3,6600,000	100:1
ServiceName, StatusCode, User ID	~1,000,000	~360,000,000	1:1

*assuming 6 hour timeframe

Small Cost for Speed

Small write time performance cost of aggregation

Additional storage space for aggregates

Still paying cost to send & store raw data

Query matching can be temperamental (check EXPLAIN)

In Practice…

Recommendations:

- Add as much detail as possible to your spans/logs
- Ingest/store everything as long as useful
- For long range, known analysis create aggregates after collection

Future: automatic identification and creation of aggregates based on query patterns!

Q/A

async: zowens@newrelic.com