CS4221
Modern Databases I.
Time-Series and Streaming Databases

Yao LU

2024 Semester 2

National University of Singapore
School of Computing

Agenda

- Time series databases

  Labs on InfluxDB

- Streaming databases by NUS PhD alumni Yingjun Wu, CEO of RisingWave

  no Labs, explore on your own

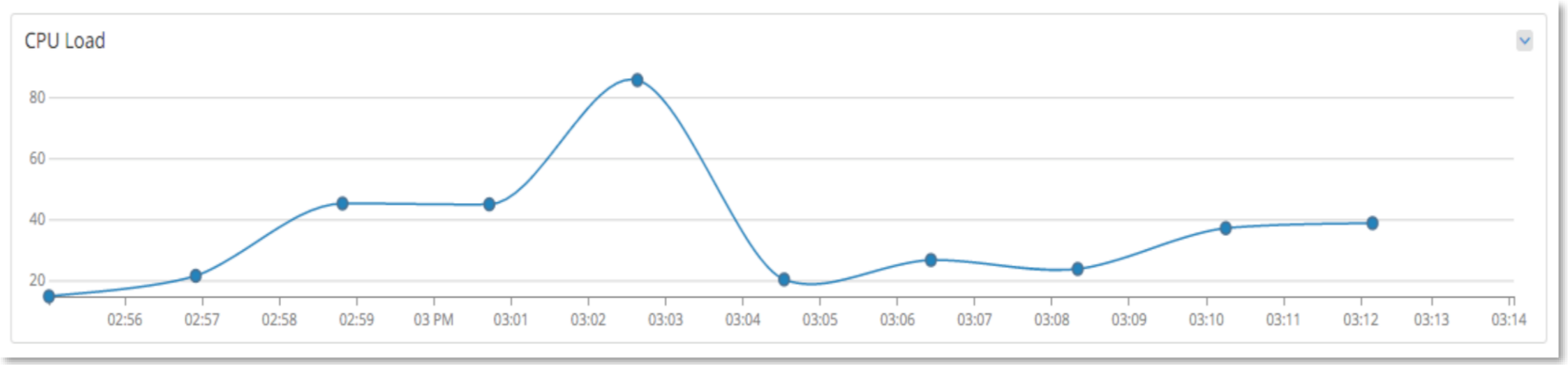
Time series data

- A time series is a series of data points indexed in time order.

- Ajay Kulkarni from TimescaleDB:
  - Time-series data: data that collectively represents how a system/process/behavior changes over time.
  - E.g., NYC taxi ride

tpep_pickup_datetime	tpep_dropoff_datetime	fare_amount	passenger_count	trip_distance
2018-01-01 00:00:17	2018-01-01 00:10:55	12	1	3.76
2018-01-01 00:00:16	2018-01-01 00:00:49	55	1	0
2018-01-01 00:00:15	2018-01-01 00:14:17	10.5	1	2.06
2018-01-01 00:00:15	2018-01-01 00:08:21	7	2	1.2
2018-01-01 00:00:14	2018-01-01 00:11:38	14	1	4
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

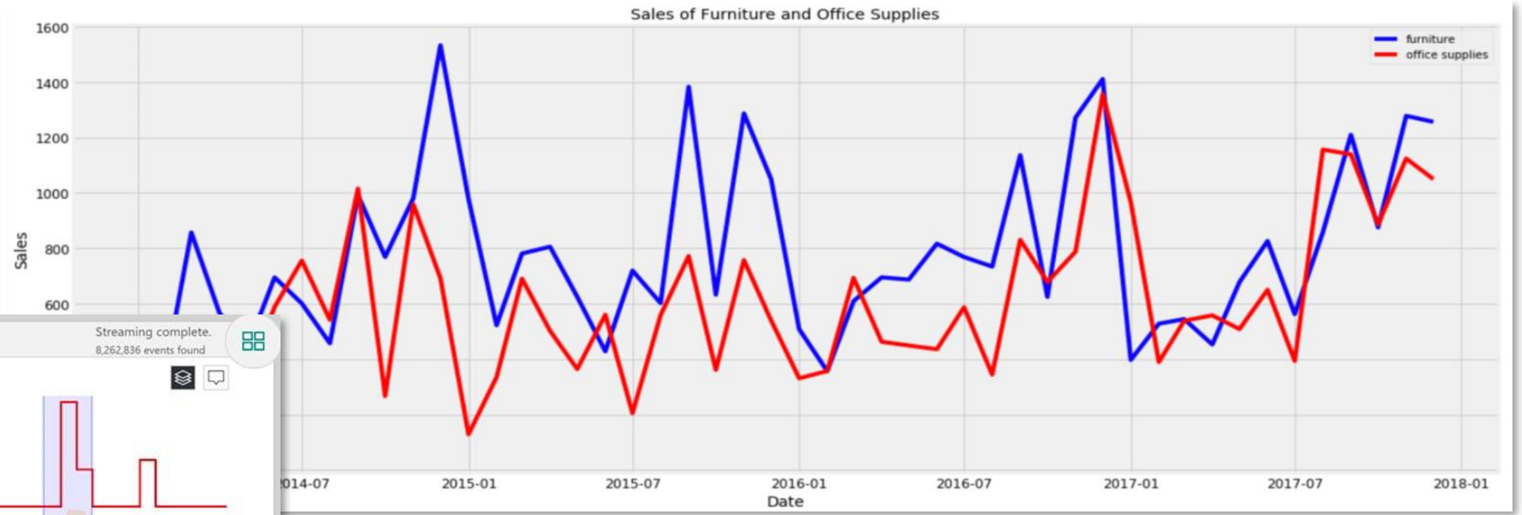
Time series data

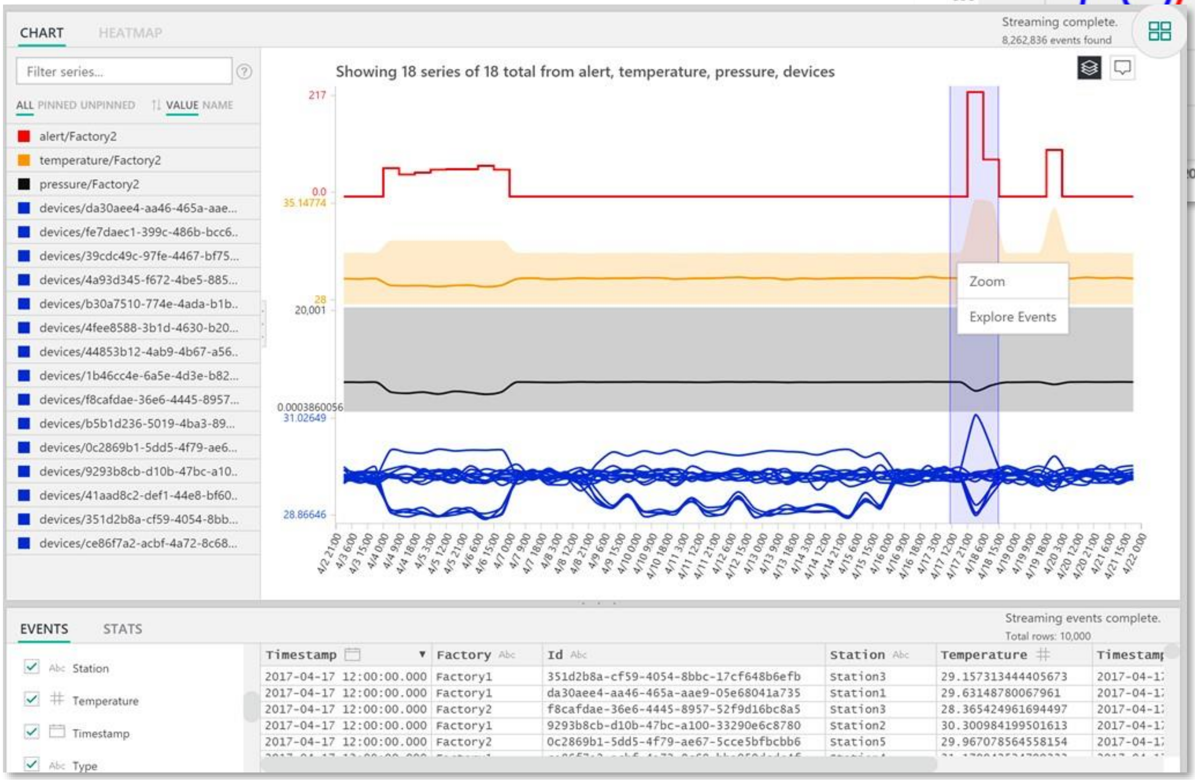


Server telemetry

Time series data

Analytics



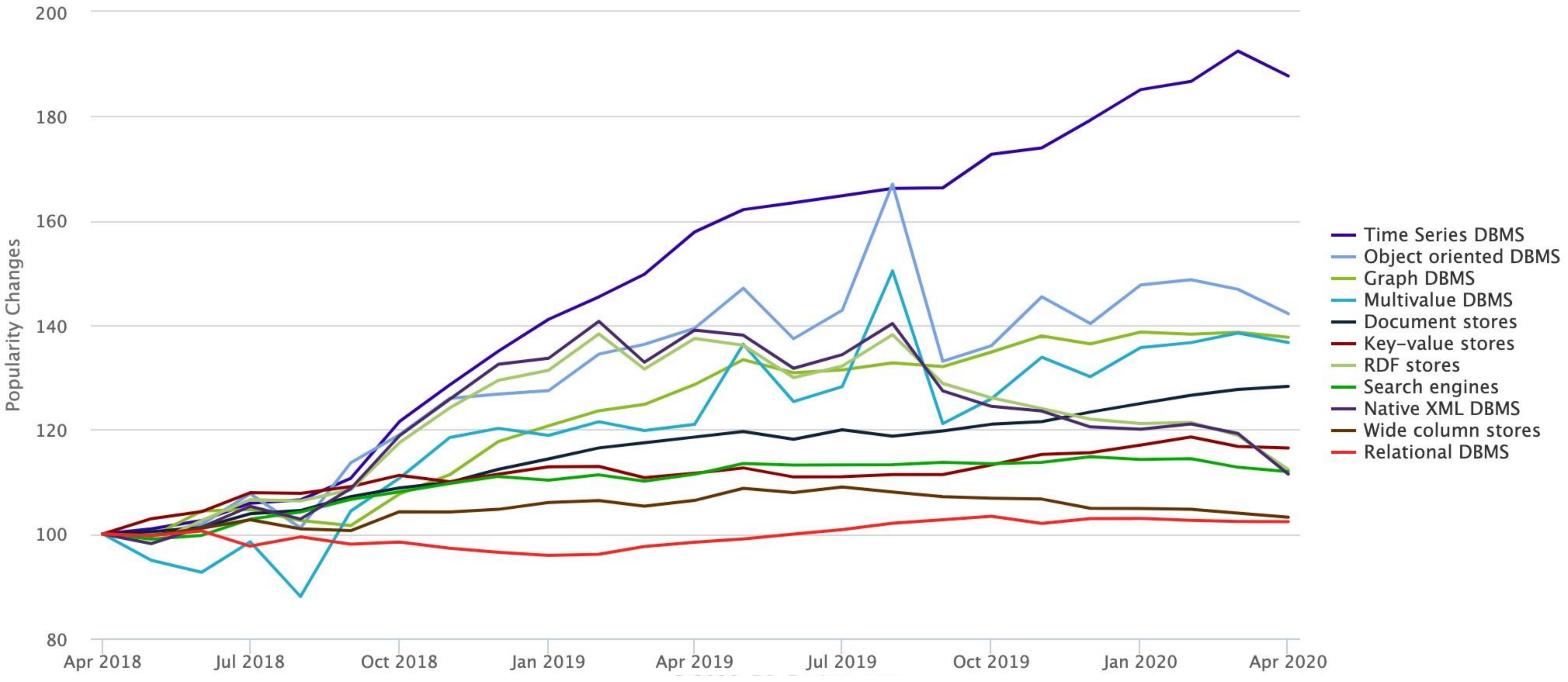


Internet of Things

- Data typically arrive in order form
- Data is append-only, in general
- Queries are always time range-based

- Special functions and operators
- Retention & continuous query

Popularity



Popularity Changes
200
180
160
140
120
100
80
Apr 2018
Jul 2018
Oct 2018
Jan 2019
Apr 2019
Jul 2019
Oct 2019
Jan 2020
Apr 2020
Time Series DBMS
Object oriented DBMS
Graph DBMS
Multivalue DBMS
Document stores
Key-value stores
RDF stores
Search engines
Native XML DBMS
Wide column stores
Relational DBMS

Most Popular TSDB

☐ include secondary database models — 33 systems in ranking, April 2020

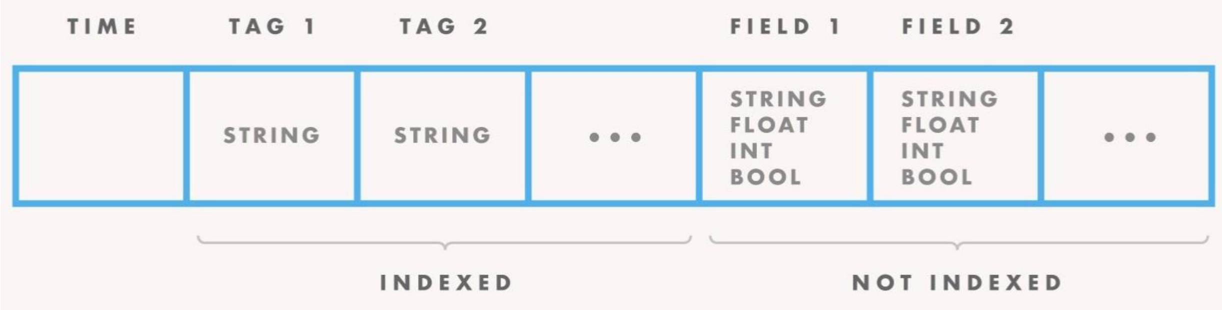
Rank			DBMS	Database Model	Score		
Apr 2020	Mar 2020	Apr 2019			Apr 2020	Mar 2020	Apr 2019
1.	1.	1.	InfluxDB	Time Series	21.62	-0.81	+4.40
2.	2.	2.	Kdb+	Time Series, Multi-model	5.27	-0.08	-0.57
3.	3.	↑ 4.	Prometheus	Time Series	4.25	+0.09	+1.34
4.	4.	↓ 3.	Graphite	Time Series	3.43	-0.01	+0.30
5.	5.	5.	RRDtool	Time Series	2.61	-0.10	-0.09
6.	6.	6.	OpenTSDB	Time Series	2.00	+0.02	-0.37
7.	↑ 8.	7.	Druid	Multi-model	1.92	+0.07	+0.28
8.	↓ 7.	8.	TimescaleDB	Time Series, Multi-model	1.87	-0.01	+0.92
9.	9.	↑ 11.	FaunaDB	Multi-model	0.87	-0.07	+0.50
10.	10.	↓ 9.	KairosDB	Time Series			
11.	11.	↑ 13.	GridDB	Time Series, Multi-model			
12.	12.		Alibaba Cloud TSDB	Time Series			
13.	13.	↓ 10.	eXtremeDB	Multi-model			
14.	14.	↓ 12.	Amazon Timestream	Time Series			
15.	15.	↑ 26.	DolphinDB	Time Series			
16.	16.	↓ 15.	IBM Db2 Event Store	Multi-model			

	InfluxDB	TimescaleDB
First Release	2013	2017
Development	From Scratch	Extension of PostgreSQL
Data Model	NoSQL	Relational
Query Language	Flux	SQL
Resilience	?	Inherit PostgreSQL
Performance	?	?

Data model

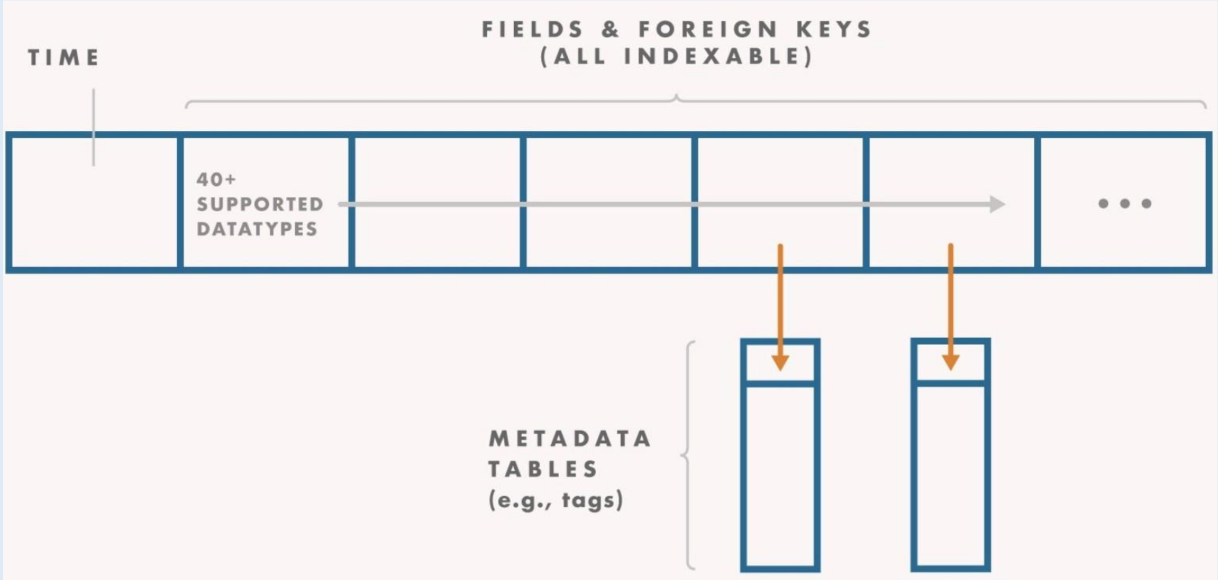
InfluxDB

- NoSQL, "Schema-less"



- Rigid & limited
  - Index on continuous field
  - Enforce data validation
- Schema-less

Timescale DB



- Relational data model
  - Normalize / denormalize
  - Index
  - Constraint check

Data model

InfluxDB

- Write data points via line protocol

field_key = field_value

Measurement

```
mem,host=host1 used_percent=23.43234543 1556892576842902000
mem,host=host1 used_percent=21.83599203 1556892777007291000
```

tag_key = tag_value (optional)

Unix timestamp

Timescale DB

- Write data
  1. Define schema
  2. INSERT INTO.. VALUES

Time	host	Mem_usage_GB	Mem_usage_percent
2019-08-18T00:00:00Z	host1	15.346	23.432
2019-08-18T00:06:00Z	host1	20.456	21.835

Data model

InfluxDB

- Querying data

Tags

_time	_measurement	host	_field	_value
2019-08-18T00:00:00Z	mem	host1	used_percent	23.432
2019-08-18T00:06:00Z	mem	host1	used_percent	21.835
2019-08-18T00:00:00Z	mem	host2	Usage_GB	10.873
2019-08-18T00:06:00Z	mem	host2	Usage_GB	9.235

Timescale DB

- Querying data

Time	host	usage_GB	usage_percent
2019-08-18T00:00:00Z	host1	15.346	23.432
2019-08-18T00:06:00Z	host1	20.456	21.835
2019-08-18T00:00:00Z	host2	10.873	18.345
2019-08-18T00:06:00Z	host2	9.235	19.032

Query language

InfluxDB

- Flux
  - Functional data scripting language
    - Anonymous function
    - Function composability

```
dataSet = from(bucket: "example-bucket")
  |> range(start: -5m)
  |> filter(fn: (r) =>
    r._measurement == "mem" and
    r._field == "used_percent"
  )
  |> drop(columns: ["host"])
```

Timescale DB

- SQL
  - Relational algebra

```
SELECT Time, used_percent
FROM mem
WHERE time > now() - interval '5 minutes'
```

Query language

InfluxDB

- Flux
  - Chaining operations
    - Build query in an incremental way

```
from(db:"telegraf")
|> range(start:-1h)
|> filter(fn: (r) => r._measurement == "foo")
|> exponentialMovingAverage(size:-10s)
```

EMV for each different stock symbol over time

Timescale DB

- SQL
  - Sub-query
  - Common table expression
    - With statement

```
SELECT date,
       symbol,
       exponential_moving_average(volume, 0.5) OVER (
         PARTITION BY symbol ORDER BY date ASC)
FROM telegraph
WHERE measurement = 'foo' and time > now() - '1 hour'
ORDER BY date, symbol;
```

Query language

InfluxDB

- Flux

```
// Memory used (in bytes)
memUsed = from(bucket: "telegraf/autogen")
  |> range(start: -1h)
  |> filter(fn: (r) =>
    r._measurement == "mem" and
    r._field == "used"
  )

// Total processes running
procTotal = from(bucket: "telegraf/autogen")
  |> range(start: -1h)
  |> filter(fn: (r) =>
    r._measurement == "processes" and
    r._field == "total"
    )

// Join memory used with total processes and calculate
// the average memory (in MB) used for running processes.
join(
    tables: {mem:memUsed, proc:procTotal},
    on: ["_time", "_stop", "_start", "host"]
  )
  |> map(fn: (r) => ({
    _time: r._time,
    _value: (r._value_mem / r._value_proc) / 1000000
  })
)
```

Timescale DB

- SQL
  - o Average memory used by each running process

```
SELECT time, (memUsed / procTotal / 1000000) as value
FROM measurements
WHERE time > now() - '1 hour';
```

Query language

InfluxDB

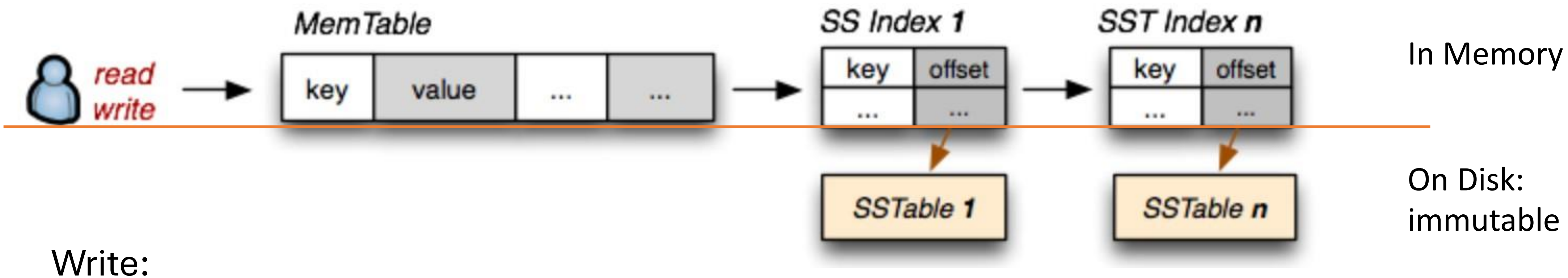
- Flux
  - Query executed in the same order as in the query statement
  - Require the user to write pushdown functions first in the query
    - Range, filter, group
  - Store and manipulate data in memory

Timescale DB

- SQL
  - Query optimizer
    - Re-ordering

InfluxDB: storage engine

Log Structured Merge Trees



Write:

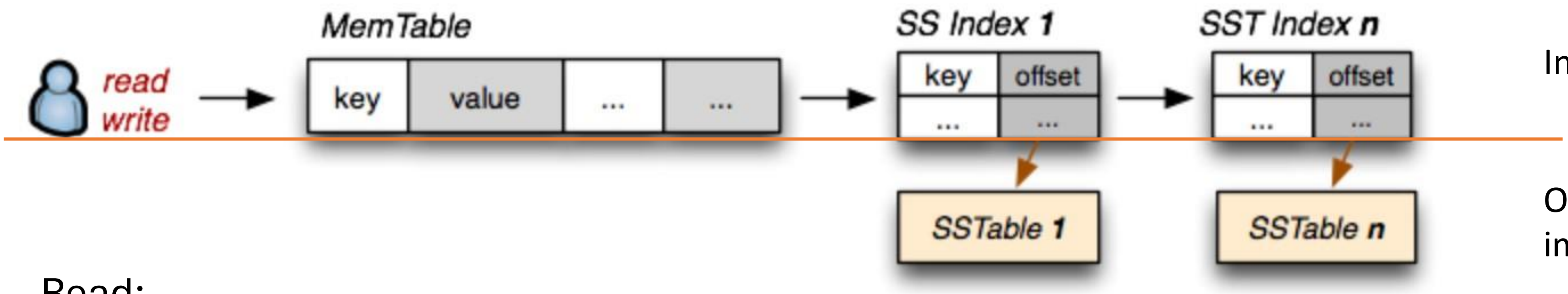
- All writes go directly to MemTable
- Periodically, the MemTable is flushed to disk as an SSTable (sorted string table)

Compaction

- Periodically, on-disk SSTables are merged and reorganized

InfluxDB: storage engine

Log Structured Merge Trees



In Memory

On Disk:
immutable

Read:

- First check the MemTable
- Then the SSTable index in sequence

Delete:

- If in MemTable, delete it
- Else, a tombstone record is appended

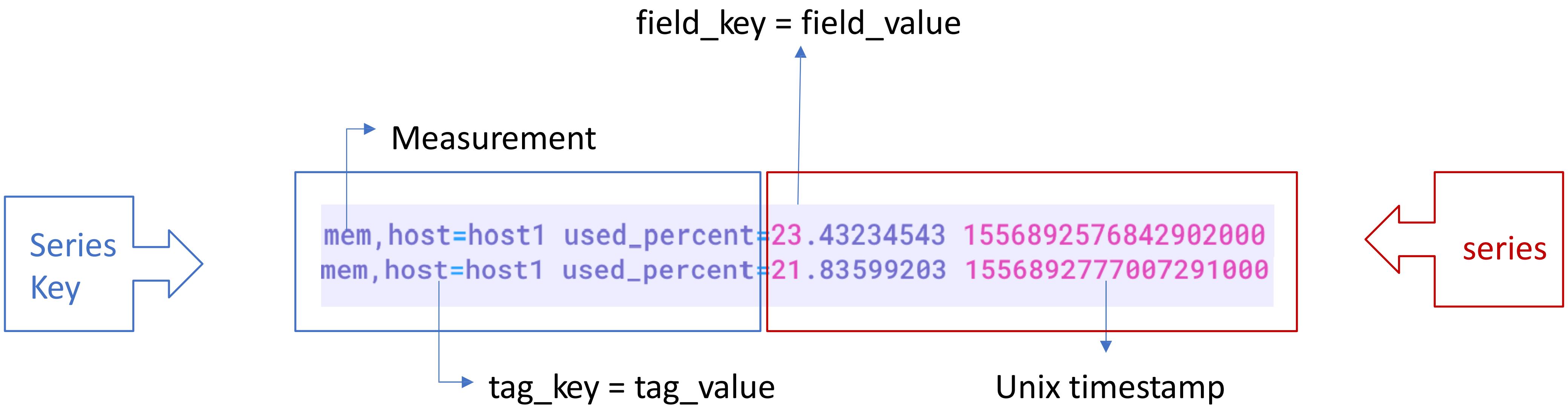
Time-series Data Characteristic:

- ➢Data retention
    - ➢Delete on large scale
    - ➢Split data into shards

InfluxDB: storage engine

- Write-Ahead Log (WAL)
  - Durability
- Cache -- memtable
  - In-memory representation of data in WAL
- Time Structured Merge file -- sstable
  - Compressed series data in columnar format
- Compactor
  - Converting less optimized Cache and TSM data into more read-optimized formats
- Compression
  - Encoder and decoder for specific data type

InfluxDB: TSM file



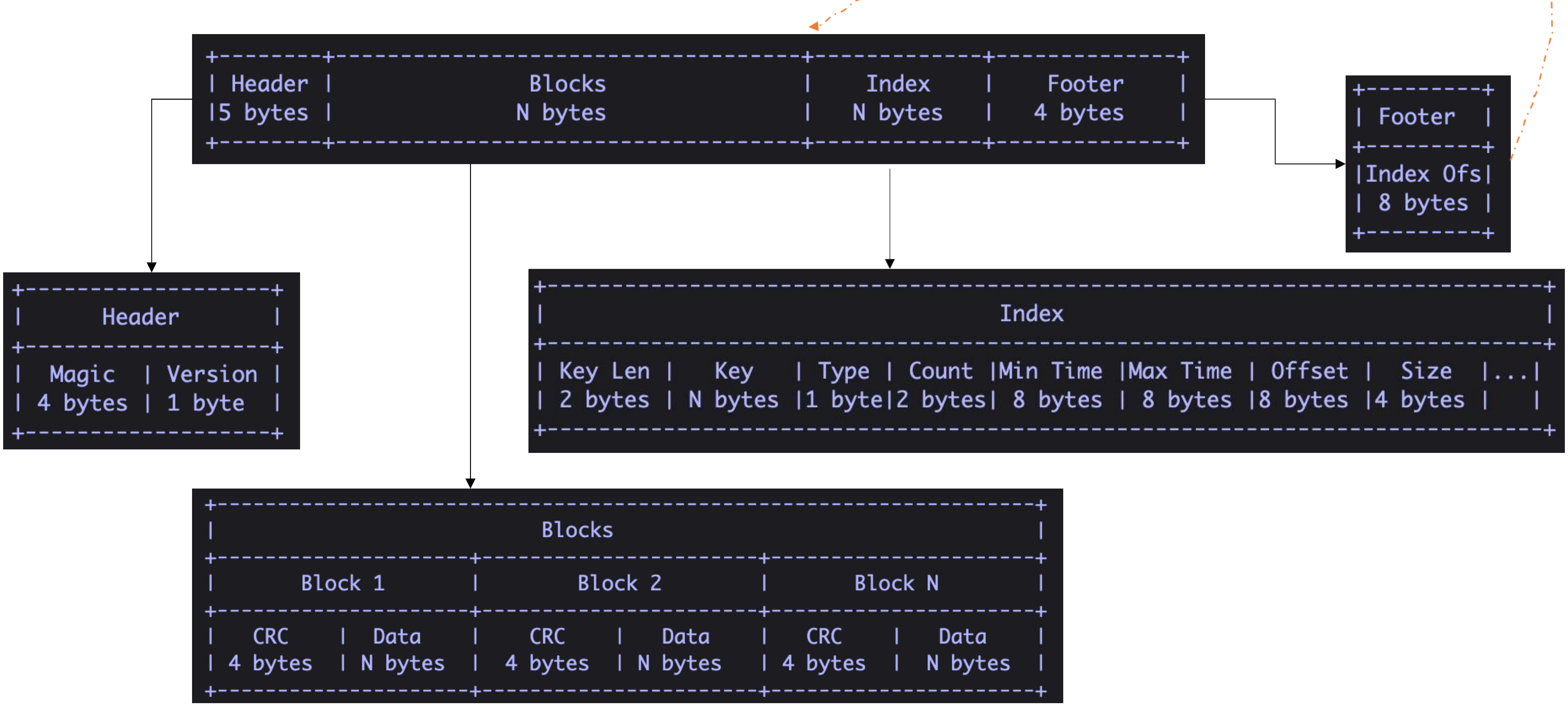
Series key : the measurement, tags, field_key

- Stored in TSM file and Time-series Index (TSI) Series

A sequence of (timestamp, field_value)

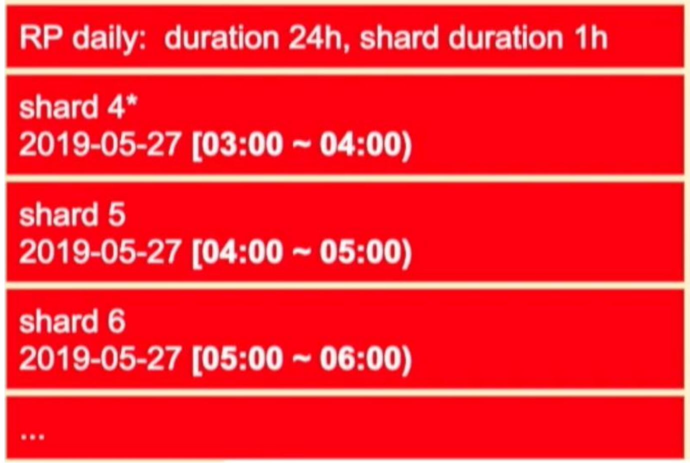
- Stored in TSM file

InfluxDB: TSM file

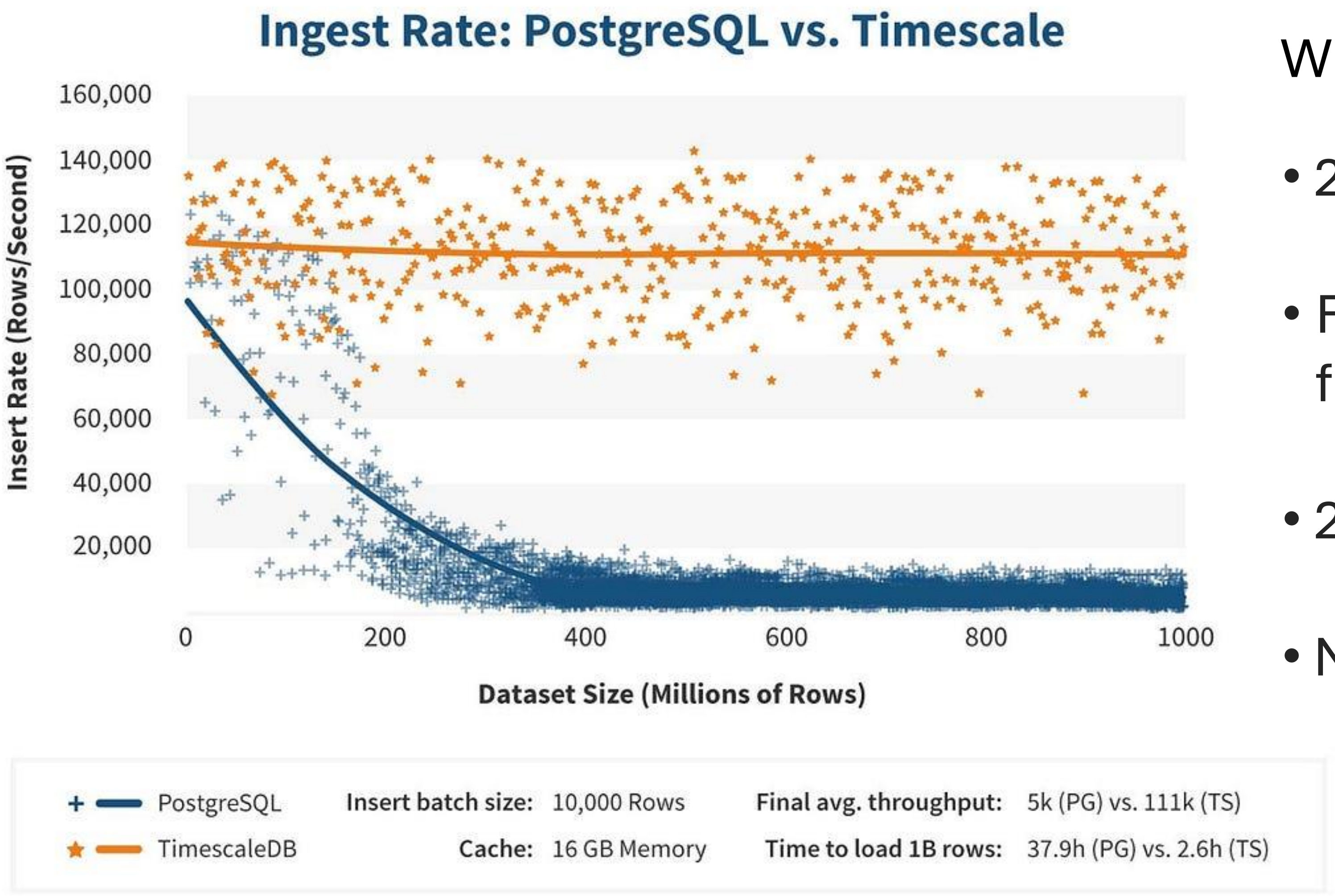


InfluxDB: misc.

- Data Compression
  - Columnar format / RLE (Regular interval) / Double Delta (Float)
- Time-Series Index
  - Stores series keys, grouped by measurement, tag, field key
    - Inverted Index
    - Read only the series that a query requires
- Shard (time-bounded)
  - A shard: a filesystem directory containing WAL, TSM, TSI
  - Retention policy



TimeScaleDB



Why not PostgreSQL? (claimed)

- 20x higher inserts at scale
- Faster time-based queries, ranging from 1.2 – 10,000x improvements
- 2000x faster deletes
- New time-centric functions

TimeScaleDB

- Whenever a new row of data is inserted into PostgreSQL, the database needs to update the indexes (e.g., B-trees) for each of the table's indexed columns. Once the indexes are too large to fit in memory this requires swapping one or more pages in from disk.

- TimescaleDB solves this through its heavily utilization and automation of time-space partitioning, even when running on a single machine. All writes to recent time intervals are only to tables that remain in memory.

TimeScaleDB

Query 1 — A simple query

```sql
SELECT date_trunc('minute', time) AS minute,
  MAX(usage_user)
FROM cpu
WHERE hostname = 'host_731'
  AND time >= '2016-01-01 02:17:08.646325 -7:00'
  AND time < '2016-01-01 03:17:08.646325 -7:00'
GROUP BY minute
ORDER BY minute ASC;
```

		TimescaleDB	PostgreSQL
Query #1	Max per minute for 1 device over 1 hour	2.04 ms	1.10 ms

TimeScaleDB

- Most simple queries (e.g., indexed lookups) that typically take <20ms, will be a few milliseconds slower on TimescaleDB, owing to the slightly larger planning time overhead.

- More complex queries that use time-based filtering or aggregations will be anywhere from 1.2x to 5x faster on TimescaleDB.

- Queries where we can leverage time-ordering will be significantly faster, anywhere from 450x to more than 14,000x faster in tests.

Performance

Query 2 — A simpler query over larger period of time

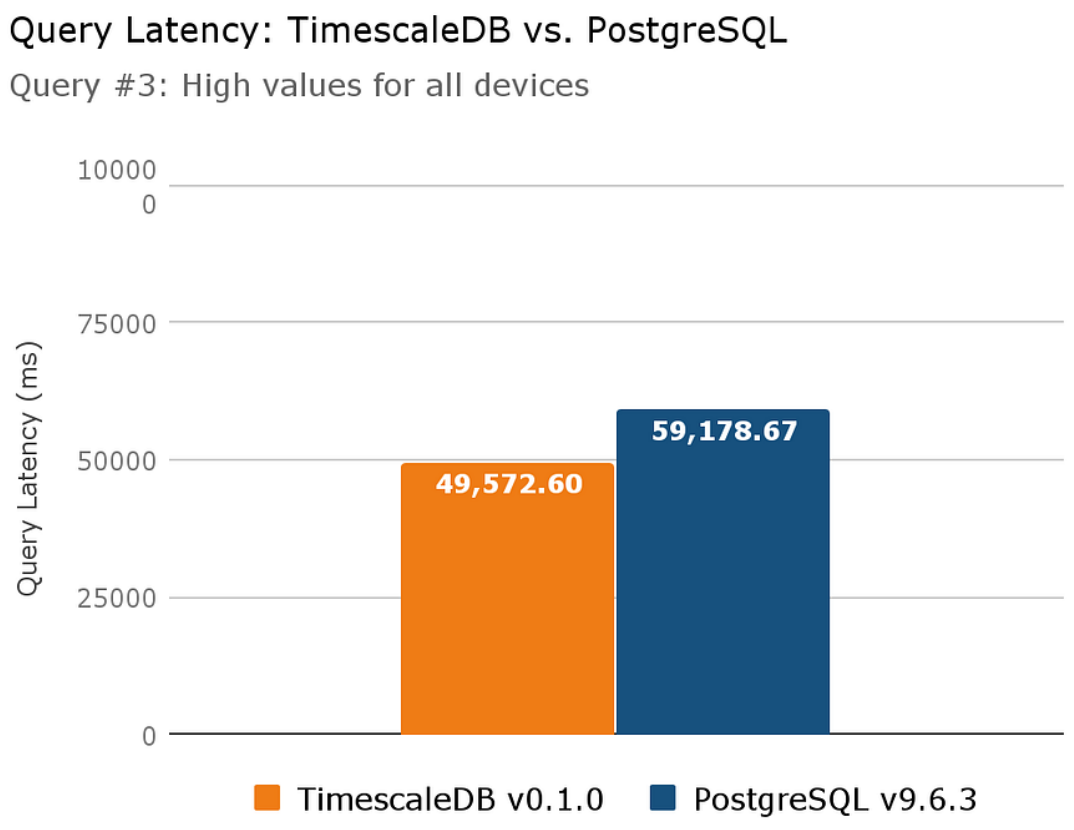
```sql
SELECT
  date_trunc('minute', time) AS minute,
  MAX(usage_user)
FROM cpu
WHERE hostname = 'host_731'
  AND time >= '2016-01-01 07:47:52'
  AND time < '2016-01-01 19:47:52'
GROUP BY minute
ORDER BY minute ASC
```

		TimescaleDB	PostgreSQL
Query #2	Max per minute for 1 device over 12 hour	14.29 ms	11.95 ms

Performance

Query 3 — Time-based filter

```sql
SELECT * FROM cpu
WHERE usage_user > 90.0
  AND time >= '2016-01-01 00:00:00'
  AND time < '2016-01-02 00:00:00'
```



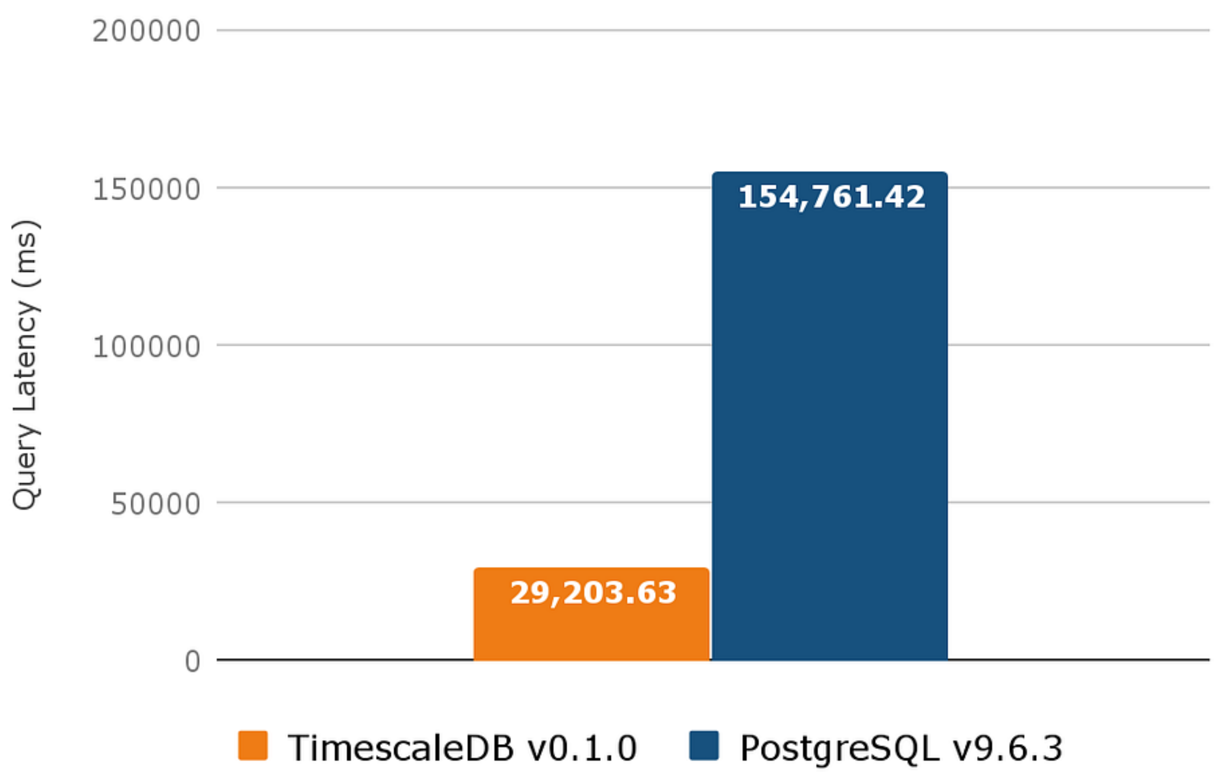
Performance

Query 4 — Time-based aggregation

```sql
SELECT date_trunc('hour', TIME)
  AS hour, hostname, avg(usage_user)
  AS mean_usage_user
FROM cpu
WHERE TIME >= '2016-01-01 00:00:00'
  AND TIME < '2016-01-02 00:00:00'
GROUP BY hour, hostname
ORDER BY hour
```

Query Latency: TimescaleDB vs. PostgreSQL

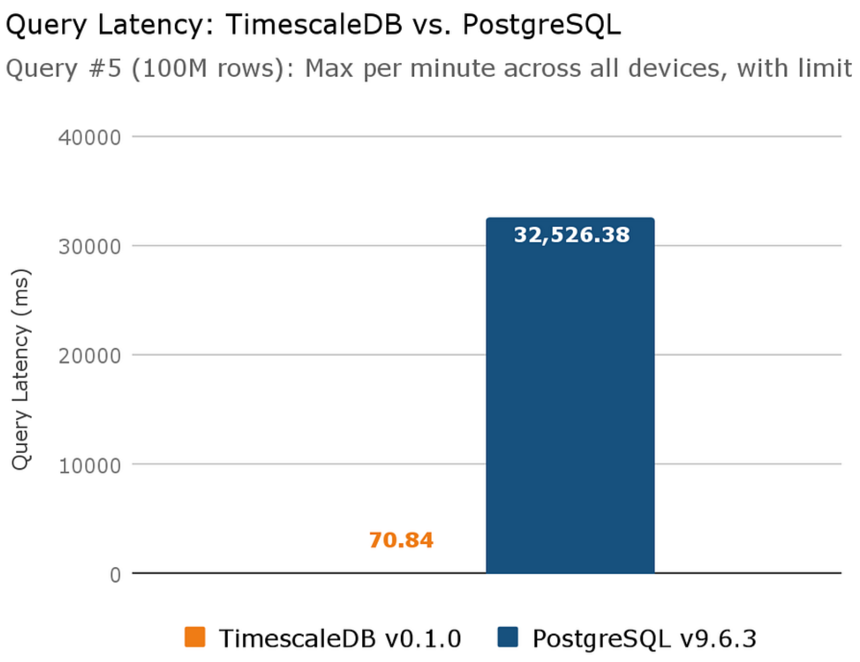
Query #4: Average per hour per every device

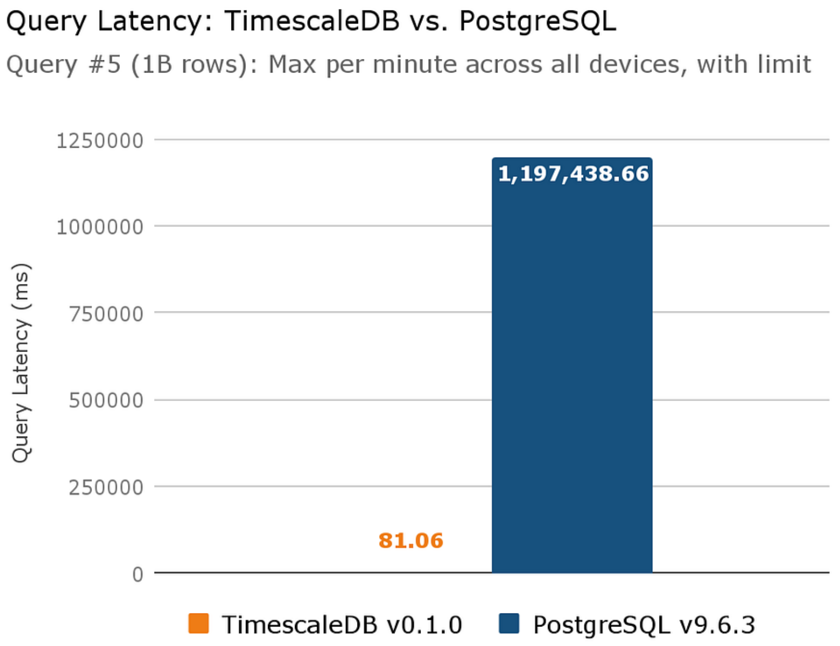


Performance

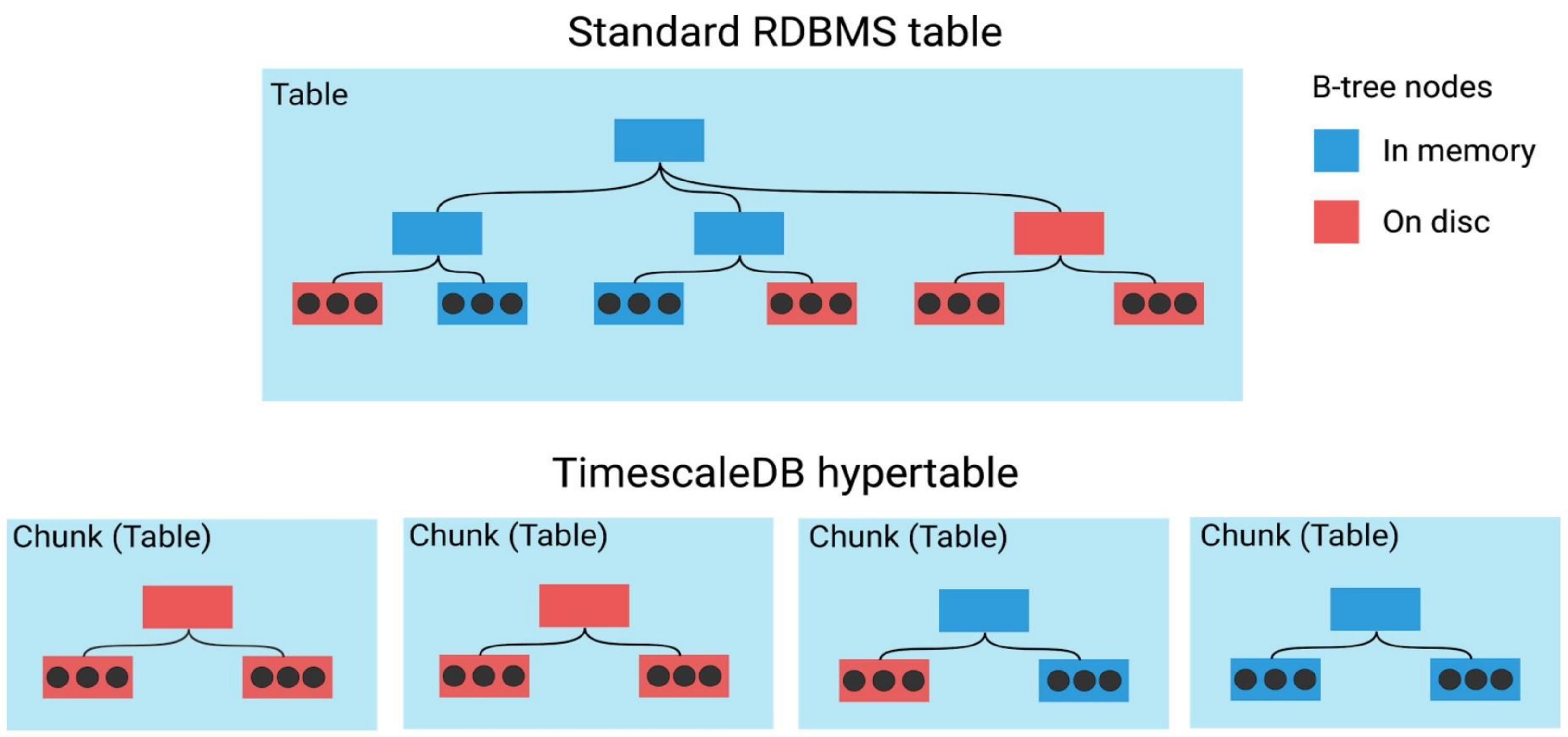
Query 5: Order by limit by using merge append

```sql
SELECT date_trunc('minute', time)
  AS minute, max(usage_user)
FROM cpu
WHERE time < '2016-01-01 19:47:52'
GROUP BY minute
ORDER BY minute DESC
LIMIT 5
```





Recent data is more accessed



Data normalisation

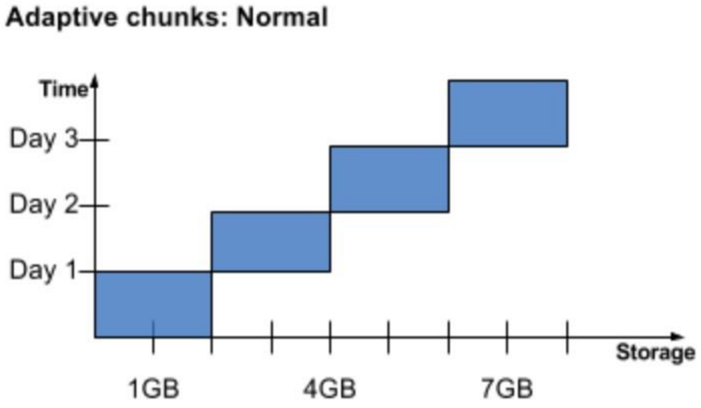
- Trading off between normalised & denormalised data storage.

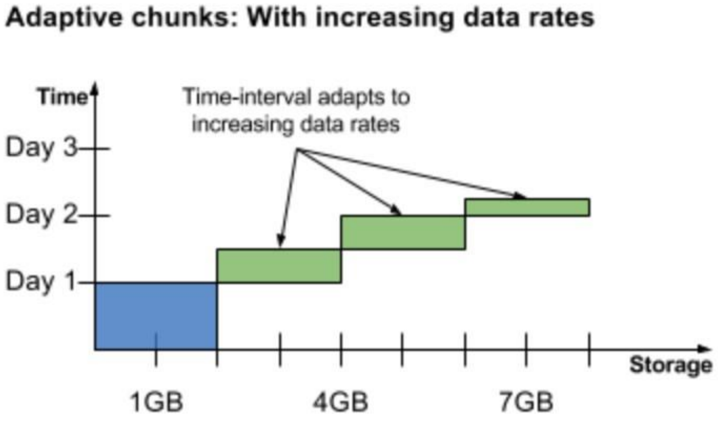
	Single Table Design	Multiple Table Design
Ease of Use	Easy	Somewhat easy
Multi-Tenancy / Privacy Regulations	Hard	Easy
Future-Proofness	Easy	Somewhat hard
Tooling Support	Easy	Hard

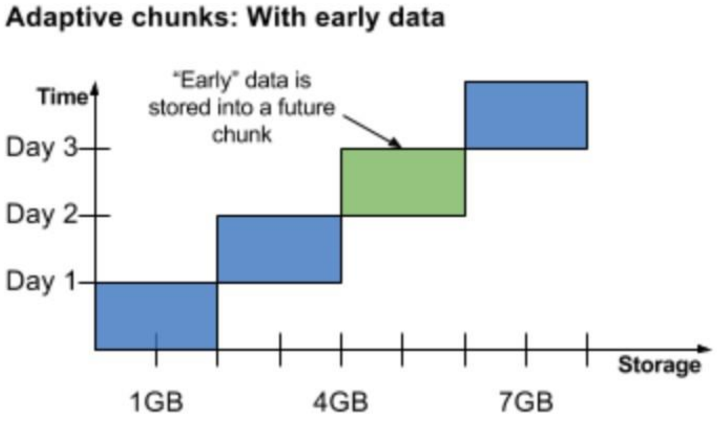
Adaptive time-space chunking

- Chunk by two dimensions
  - Time interval
  - Primary key (e.g., server/device/asset ID)

- Design choices
  - Fixed-duration interval
  - Fixed-size chunk
  - Adaptive interval > Hypertable in Timescale DB







Tuning & perf optimization in Timescale DB

- Inherit and adapt from PostgreSQL
  - Materialized views
  - Tiered storage
  - Compression

- Design choices & optimizations specific to time-series data & TimescaleDB
  - Partition pruning
  - Continuous aggregates

Compression (generic)

- **Dictionary compression / Page compression**
  - Compresses data by storing repeating values and common prefixes only once and then making references.
- **Compression in rowstores**
  - It takes fixed-length columns and makes them variable length, adding additional bytes for the overhead of tracking the changes being made.
- **Compression in columnstores**
  - Run length encoding >>
  - What if multiple columns?

Value
Mario
Sonic the Hedgehog
Sonic the Hedgehog
Sonic the Hedgehog
Mario
Yoshi
Ness
Pikachu
Pikachu
Sonic the Hedgehog
Yoshi
Link

Value	Value Count
Mario	1
Sonic the Hedgehog	3
Mario	1
Yoshi	1
Ness	1
Pikachu	2
Sonic the Hedgehog	1
Yoshi	1
Link	1

Compression in Timescale

- Simple but effective trick

```sql
ALTER TABLE metrics
SET (timescaledb.compress, timescaledb.compress_orderby='time');
```

Uncompressed chunk

Timestamp	Device ID	Status code	Temperature
12:00:01	A	0	70.11
12:00:01	B	0	69.7
12:00:02	A	0	70.12
12:00:02	B	0	69.69
12:00:03	A	0	70.14
12:00:03	B	4	69.7

Compressed chunk

Timestamp	Device ID	Status code	Temperature
[12:00:01, 12:00:01, 12:00:02, 12:00:02, 12:00:03, 12:00:03]	[A, B, A, B, A, B]	[0, 0, 0, 0, 0, 4]	[70.11, 69.70, 70.12, 69.69, 70.14, 69.70]

Batch (up to 1000 uncompressed tuples)

Delta-of-delta and simple-8b with run-length encoding

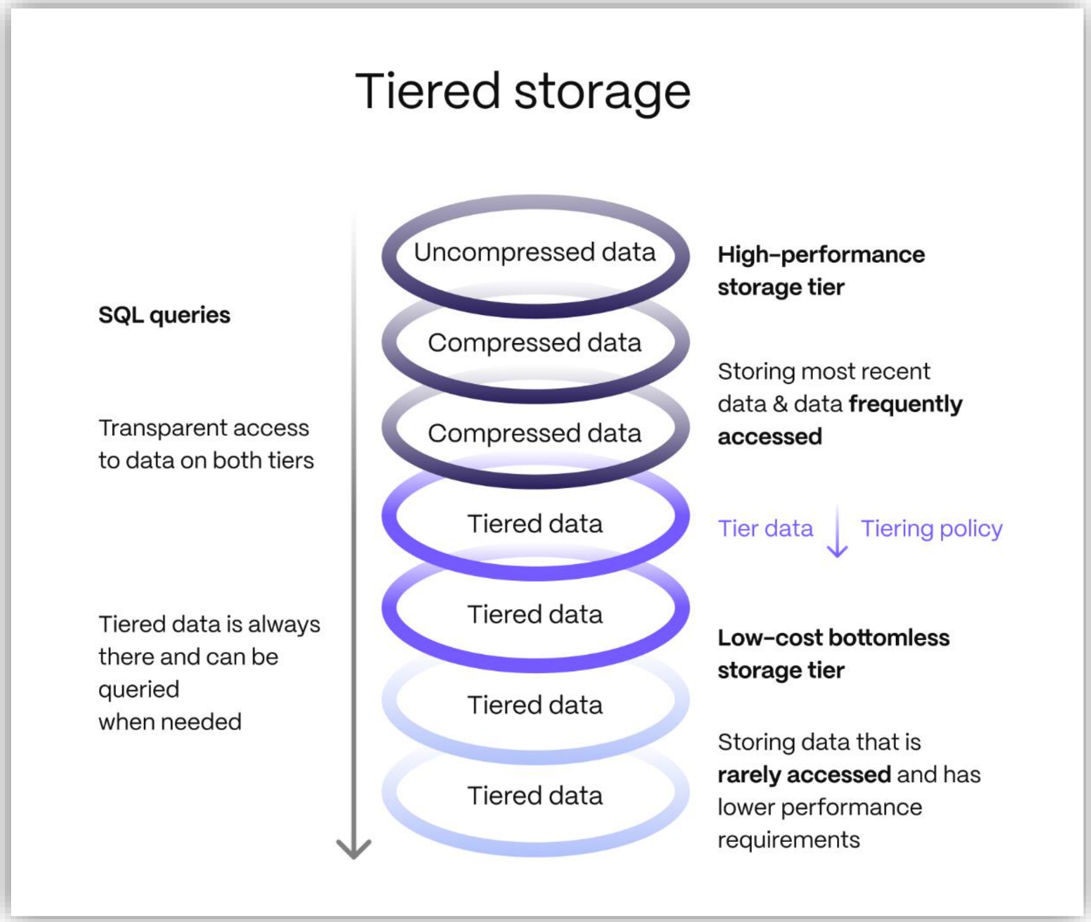
Dictionary compression

Gorilla compression

Tiered storage

- Moving “cold” data to cheaper & slower storage (e.g., AWS S3) saves costs (and your wallet).

- More useful for DB replicas.



Tiered storage

- User-defined tiering policy

```
SELECT add_tiering_policy('events', INTERVAL '1 month');
```

- Query runtime: transparent to the users

```
EXPLAIN
SELECT time_bucket('1 day', ts) as day,
        max(value) as max_reading,
        device_id
    FROM metrics
    JOIN devices ON metrics.device_id = devices.id
    JOIN sites ON devices.site_id = sites.id
WHERE sites.name = 'DC-1b'
GROUP BY day, device_id
ORDER BY day;
```

```
QUERY PLAN
----------------------------------------------------------
GroupAggregate
    Group Key: (time_bucket('1 day'::interval, _hyper_5666_706386_chunk.ts)),
_hyper_5666_706386_chunk.device_id
    -> Sort
        Sort Key: (time_bucket('1 day'::interval, _hyper_5666_706386_chunk.ts)),
_hyper_5666_706386_chunk.device_id
        -> Hash Join
            Hash Cond: (_hyper_5666_706386_chunk.device_id = devices.id)
            -> Append
    -> Seq Scan on _hyper_5666_706386_chunk
                -> Seq Scan on _hyper_5666_706387_chunk
                -> Seq Scan on _hyper_5666_706388_chunk
                -> Foreign Scan on osm_chunk_3334
            -> Hash
                -> Hash Join
                    Hash Cond: (devices.site_id = sites.id)
                    -> Seq Scan on devices
                    -> Hash
                        -> Seq Scan on sites
                            Filter: (name = 'DC-1b'::text)
```

- Partition pruning in much more effective for tiered storage!!

Tuning & perf optimization in Timescale DB

- Inherit and adapt from PostgreSQL
  - Materialized views
  - Tiered storage
  - Compression

- Design choices & optimizations specific to time-series data & TimescaleDB
  - Partition pruning
  - Continuous aggregates

Partition pruning in Timescale

```
SET enable_partition_pruning = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=188.76..188.77 rows=1 width=8)
   ->  Append  (cost=0.00..181.05 rows=3085 width=0)
         ->  Seq Scan on measurement_y2006m02  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m11  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2007m12  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

Each partition is scanned.

```
SET enable_partition_pruning = on;
```
(This is by default)

```
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=37.75..37.76 rows=1 width=8)
   ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
         Filter: (logdate >= '2008-01-01'::date)
```

The planner examines each partition to see if it needs be scanned, according to the WHERE clause.

Recall sequential vs index scans

- What if WHERE is on non-key column?

```sql
CREATE TABLE orders (
    order_id        serial,
    time            timestamptz,
    customer_id     int,
    order_total     float
);
```

```
# To fetch a single order from the hypertable, you would run a query like this:
SELECT * FROM orders WHERE order_id = 3942785;
                                        QUERY PLAN
---------------------------------------------------------------------------------------------
Gather  (cost=1000.00..509743.32 rows=148962 width=24)
   Workers Planned: 2
   ->  Parallel Append  (cost=0.00..493847.12 rows=62132 width=24)
         ->  Parallel Seq Scan on _hyper_4_280_chunk  (cost=0.00..1370.21 rows=294 width=24)
               Filter: (order_id = 3942785)
         ->  Parallel Seq Scan on _hyper_4_281_chunk  (cost=0.00..1370.21 rows=294 width=24)
               Filter: (order_id = 3942785)
… …
# Scanning 365 chunks in total

Time: 2176.563 ms (00:02,177)
```

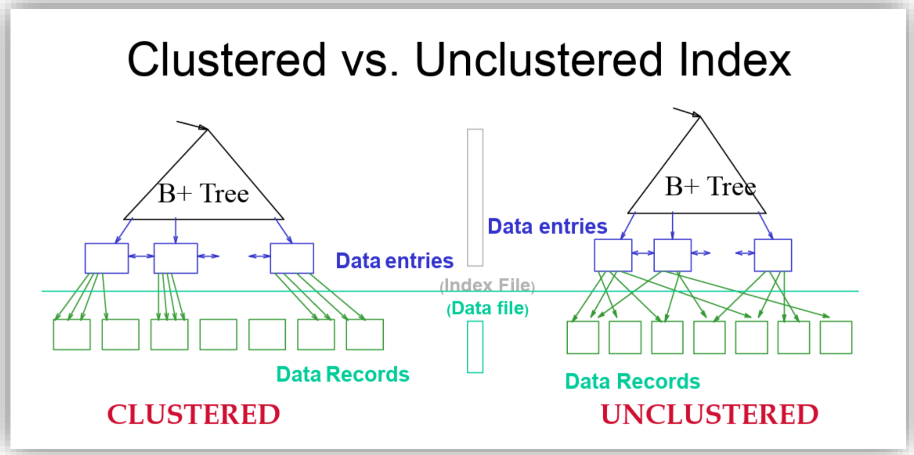
- Recall clustered index on key columns >> you end up in scanning all blocks

Any better?

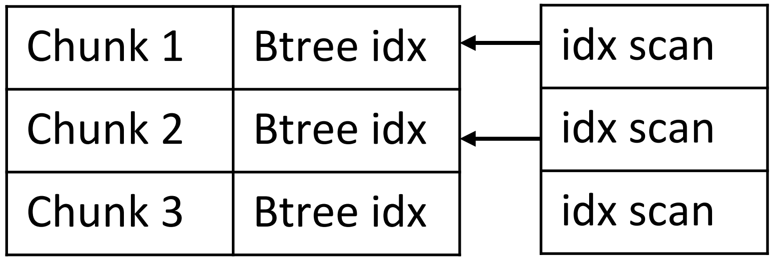
- We can create a non-clustered index on order_id

  at the cost of storage overhead (37% in this case)

(Recall clustered vs. non-clustered) index



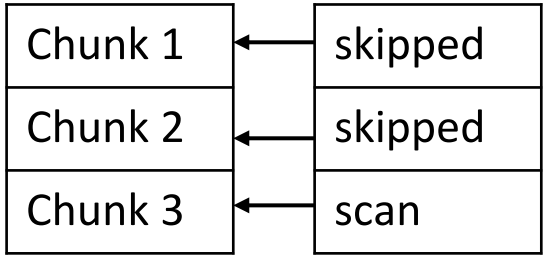

```
SELECT * FROM orders WHERE order_id = 3942785;
                                      QUERY PLAN


---------------------------------------------------------------------------------
-------------------
 Append  (cost=0.29..3043.28 rows=366 width=24)
    ->  Index Scan using _hyper_4_213_chunk_orders_order_id_idx on _hyper_4_213_chunk
(cost=0.29..8.31 rows=1 width=24)
          Index Cond: (order_id = 3942785)
    ->  Index Scan using _hyper_4_214_chunk_orders_order_id_idx on _hyper_4_214_chunk
(cost=0.29..8.31 rows=1 width=24)
          Index Cond: (order_id = 3942785)
… … ...
Time: 34.838 ms
```



Still not enough? Partition pruning to the rescue

- Suppose we can skip chunks/blocks according to the WHERE clause

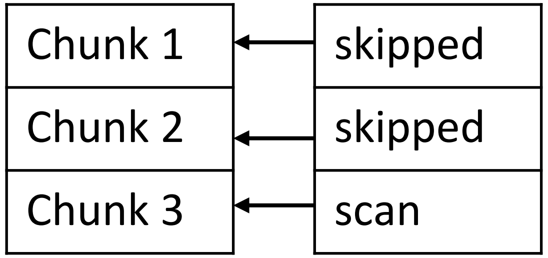
```
SELECT enable_chunk_skipping('orders', 'order_id');
ALTER TABLE orders SET (timescaledb.compress);
SELECT compress_chunk(show_chunks('orders'));

SELECT * FROM orders where order_id = 3942785;
                                           QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Custom Scan (DecompressChunk) on _hyper_4_254_chunk  (cost=0.15..3.30 rows=22000 width=24)
   Vectorized Filter: (order_id = 3942785)
   ->  Seq Scan on compress_hyper_5_1352_chunk  (cost=0.00..3.30 rows=22 width=148)
         Filter: ((_ts_meta_v2_min_order_id <= 3942785) AND (_ts_meta_v2_max_order_id >= 3942785))
(4 rows)
Time: 5.064 ms
```



Still not enough? Partition pruning to the rescue

- Suppose we can skip chunks/blocks according to the WHERE clause

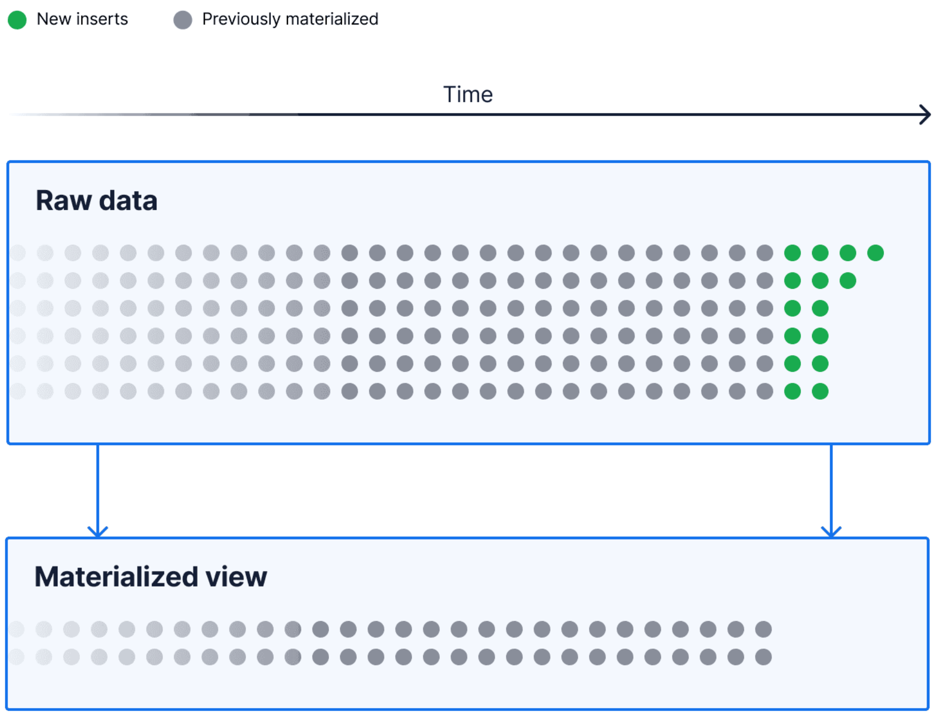
```
SELECT enable_chunk_skipping('orders', 'order_id');
ALTER TABLE orders SET (timescaledb.compress);
SELECT compress_chunk(show_chunks('orders'));

SELECT * FROM orders where order_id = 3942785;
                                            QUERY PLAN
---------------------------------------------------------------------------------------------------
 Custom Scan (DecompressChunk) on _hyper_4_254_chunk  (cost=0.15..3.30 rows=22000 width=24)
   Vectorized Filter: (order_id = 3942785)
   ->  Seq Scan on compress_hyper_5_1352_chunk  (cost=0.00..3.30 rows=22 width=148)
         Filter: ((_ts_meta_v2_min_order_id <= 3942785) AND (_ts_meta_v2_max_order_id >= 3942785))
(4 rows)
Time: 5.064 ms
```



- But how? Unfortunately, this is not implemented in PostgreSQL
- Rule based, ML-based etc. [K. Rong, Y. Lu, P. Bailis, S. Kandula, P. Levis. Approximate Partition Selection for Big-Data Workloads using Summary Statistics. VLDB 2020.]

Materialized views

- Recall views and materialized views from PostgreSQL
- REFERSH MATERIALIZED VIEW

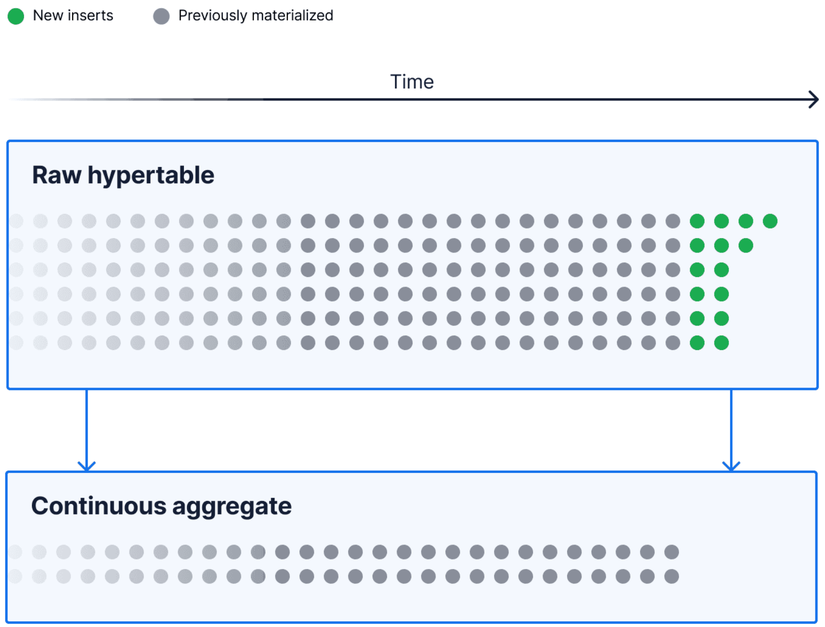


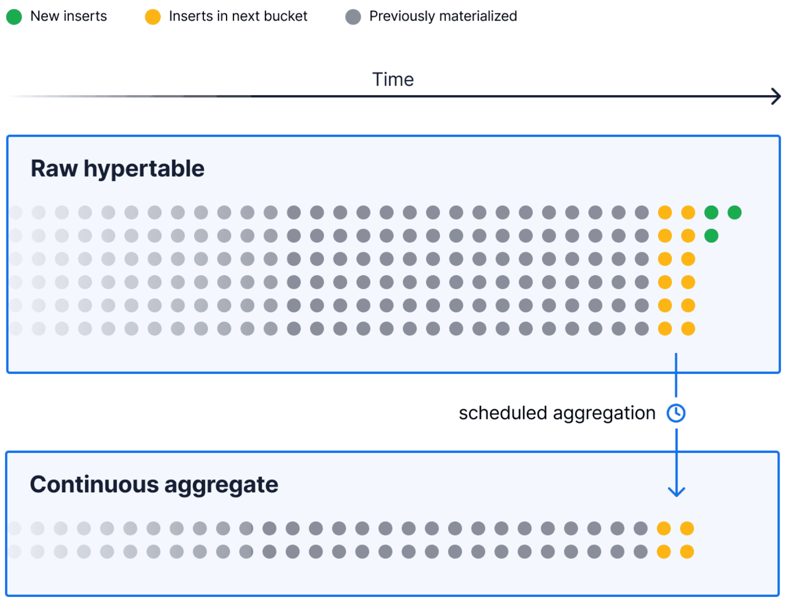
Continuous aggregates in Timescale

- Incremental, automatically updated materialized views (need update policy)

```
CREATE MATERIALIZED VIEW ohlc_cont
WITH (timescaledb.continuous) AS
SELECT time_bucket('15 min', time) bucket, symbol, first(price, time), max(price), min(price),
last(price, time)
FROM stocks_real_time
GROUP BY time_bucket('15 min', time), symbol;
```

```
SELECT add_continuous_aggregate_policy('ohlc_cont'::regclass, start_offset=>NULL, end_offset=>'15
mins'::interval,  schedule_interval=>'5 mins'::interval);
```



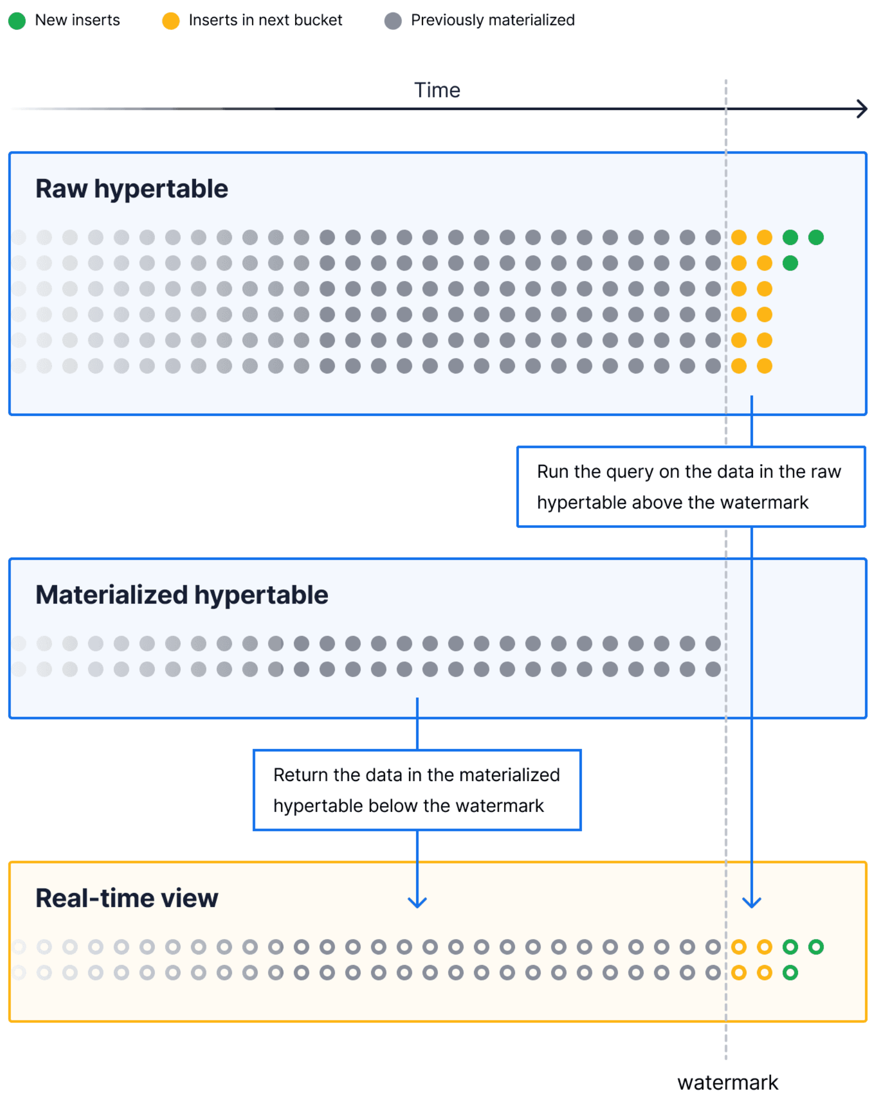


A step further: real-time views

Real-time continuous aggregates combine two parts:

- A *materialized hypertable*,
- A *real-time view,* which queries both the materialized hypertable and the raw hypertable (in the not-yet-aggregated region).

```
CREATE VIEW ohlc_cont AS  SELECT _materialized_hypertable_15.bucket,
    _materialized_hypertable_15.symbol,
    _materialized_hypertable_15.first,
    _materialized_hypertable_15.max,
    _materialized_hypertable_15.min,
    _materialized_hypertable_15.last
   FROM _timescaledb_internal._materialized_hypertable_15
  WHERE _materialized_hypertable_15.bucket < 
COALESCE(_timescaledb_internal.to_timestamp(_timescaledb_internal.cagg_watermark(15)), '-
infinity'::timestamp with time zone)
UNION ALL
 SELECT time_bucket('00:15:00'::interval, stocks_real_time."time") AS bucket,
    stocks_real_time.symbol,
    first(stocks_real_time."time", stocks_real_time.price) AS first,
    max(stocks_real_time.price) AS max,
    min(stocks_real_time.price) AS min,
    last(stocks_real_time."time", stocks_real_time.price) AS last
   FROM stocks_real_time
  WHERE stocks_real_time."time" >= 
COALESCE(_timescaledb_internal.to_timestamp(_timescaledb_internal.cagg_watermark(15)), '-
infinity'::timestamp with time zone)
  GROUP BY (time_bucket('00:15:00'::interval, stocks_real_time."time")), stocks_real_time.symbol;
```



Labs: try out InfluxDB (but not Timescale)

- We will use Timescale DB in later projects. Today's Labs let's quickly explore InfluxDB instead.

- Compare design choices and tuning strategies among PostgreSQL, Influx and Timescale.

Take home notes

- Design DB & optimizations according to your data model and use case!

- Timescale is fun to learn, as a use case in expanding PostgreSQL for a particular data model & use case.

- Many tricks and design patterns are transferrable.
  - Tiered storage
  - Partition pruning
  - Materialized views

Agenda

- Time series databases

  Labs on InfluxDB

- Streaming databases by NUS PhD alumni Yingjun Wu, CEO of RisingWave

  no Labs, explore on your own

Credits

- Silu Huang, Bytedance
- TimeScale Blogs