Performant Continuously Up to Date Materialized Aggregates

David Kohn

Solutions Architect / Software Engineer, Timescale

david@timescale.com · **github.com/timescale**

A Use Case: Stock Trades

```
postgres=# select * from trade_data order by trade_time limit 15 ;
          trade_time            | symbol |  price  | num_shares
--------------------------------+--------+---------+------------
 2019-06-14 09:30:00-04         | AAPL   |  200.32 |        157
 2019-06-14 09:30:00-04         | GE     |   11.08 |        276
 2019-06-14 09:30:00-04         | GOOG   | 1118.56 |         53
 2019-06-14 09:30:01.071429-04  | AAPL   |  200.05 |        160
 2019-06-14 09:30:02.142858-04  | AAPL   |  200.32 |        155
 2019-06-14 09:30:03.214287-04  | AAPL   |  199.98 |        157
 2019-06-14 09:30:03.75-04      | GE     |   11.41 |        270
 2019-06-14 09:30:04.285716-04  | AAPL   |   200.2 |        157
 2019-06-14 09:30:05.357145-04  | AAPL   |  199.73 |        155
 2019-06-14 09:30:06.428574-04  | AAPL   |  200.29 |        161
 2019-06-14 09:30:07.5-04       | GE     |   11.27 |        274
 2019-06-14 09:30:07.500003-04  | AAPL   |  200.25 |        154
 2019-06-14 09:30:08.571432-04  | AAPL   |  199.64 |        155
 2019-06-14 09:30:09.642861-04  | AAPL   |  200.05 |        156
 2019-06-14 09:30:10.71429-04   | AAPL   |  199.73 |        155
(15 rows)
```

Typical OHLC Query

```
SELECT time_bucket('15 min', trade_time),
   symbol,
   first(price, trade_time) as open,
   max(price) as high,
   min(price) as low,
   last(price, trade_time) as close,
   sum(num_shares) as total_volume
FROM trade_data
WHERE trade_time > now() - '30 days'::interval
GROUP BY time_bucket('15 min', trade_time), symbol
ORDER BY time_bucket('15 min', trade_time) DESC;
```

Typical OHLC Query

time_bucket	symbol	open	high	low	close	total_volume
2019-06-14 15:45:00-04	AAPL	195.44	203.37	193.25	197.09	57234
2019-06-14 15:45:00-04	GE	10.8	11.72	10.35	10.52	89445
2019-06-14 15:45:00-04	GOOG	1112.52	1134.02	1088.79	1100.5	9047
2019-06-14 15:45:00-04	GOOGL	1131.87	1140.7	1087.96	1123.17	31579
2019-06-14 15:45:00-04	IBM	140.08	143.13	136.12	138.48	8764
2019-06-14 15:30:00-04	AAPL	198.27	203.95	193.73	200.42	31154
2019-06-14 15:30:00-04	GE	11.2	11.74	10.37	10.64	31313
2019-06-14 15:30:00-04	GOOG	1128.93	1137.21	1094.49	1114.89	3517
2019-06-14 15:30:00-04	GOOGL	1113.12	1138.93	1093.73	1131.38	4477
2019-06-14 15:30:00-04	IBM	138.9	141.93	136.33	140.07	5326

How do we make it into an interactive dashboard?

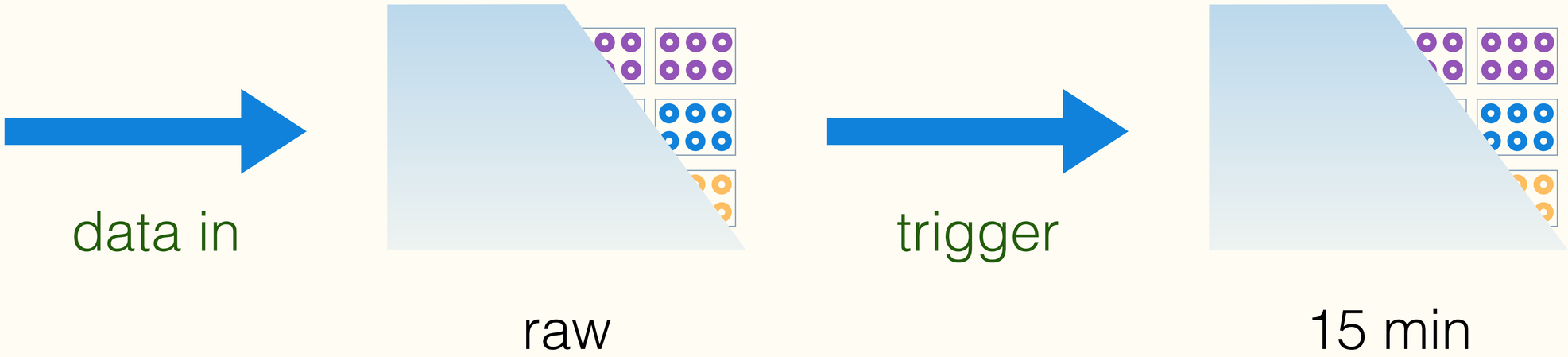
Let's try a Materialized View:

```
CREATE MATERIALIZED VIEW ohlc AS
SELECT time_bucket('15 min', trade_time),
  symbol,
  first(price, trade_time) as open,
  max(price) as high,
  min(price) as low,
  last(price, trade_time) as close,
  sum(num_shares) as total_volume
FROM trade_data
GROUP BY time_bucket('15 min', trade_time), symbol;
```

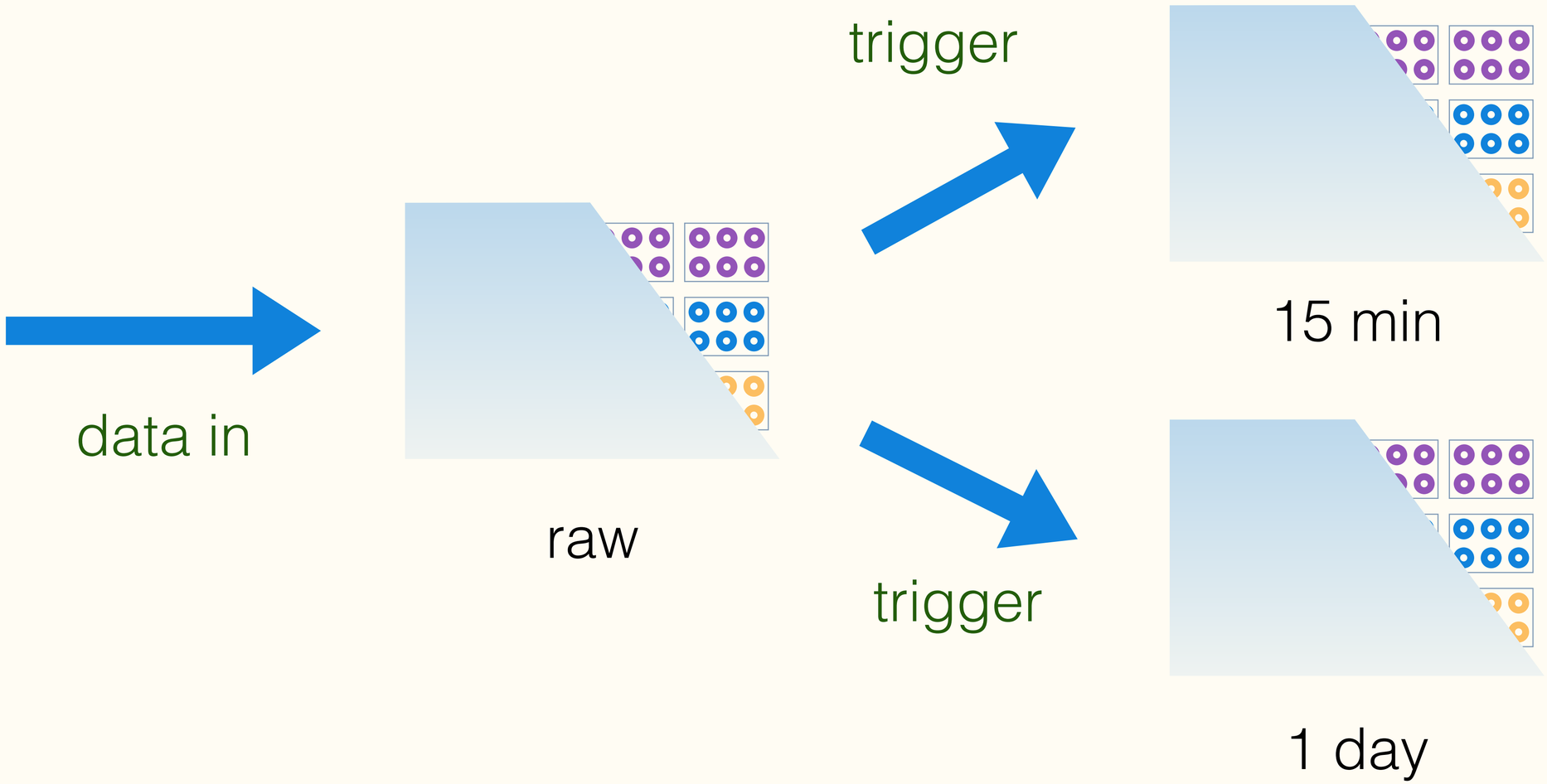
But…

- Need to run REFRESH MATERIALIZED VIEW manually or it will quickly be out of date.
- Recalculates the entire view on every refresh
  - Will REFRESH CONCURRENTLY help?
  - Very inefficient for insert-mostly, time-ordered workloads

What about triggers?



What about triggers?



What about triggers?

- Triggers can keep materializations up to date
- But they cause ***write amplification*** every row inserted (or updated/deleted) means a modification to each materialization a.k.a.
- Triggers can cause lock conflicts that further slow writes

The data industry is undergoing a **generational shift**

1970s-1990s

The relational database era for transactional processing

Oracle, DB2, SQL Server

2000s-2010s

The big data and non-relational era for analytics

Hadoop, Cassandra, MongoDB

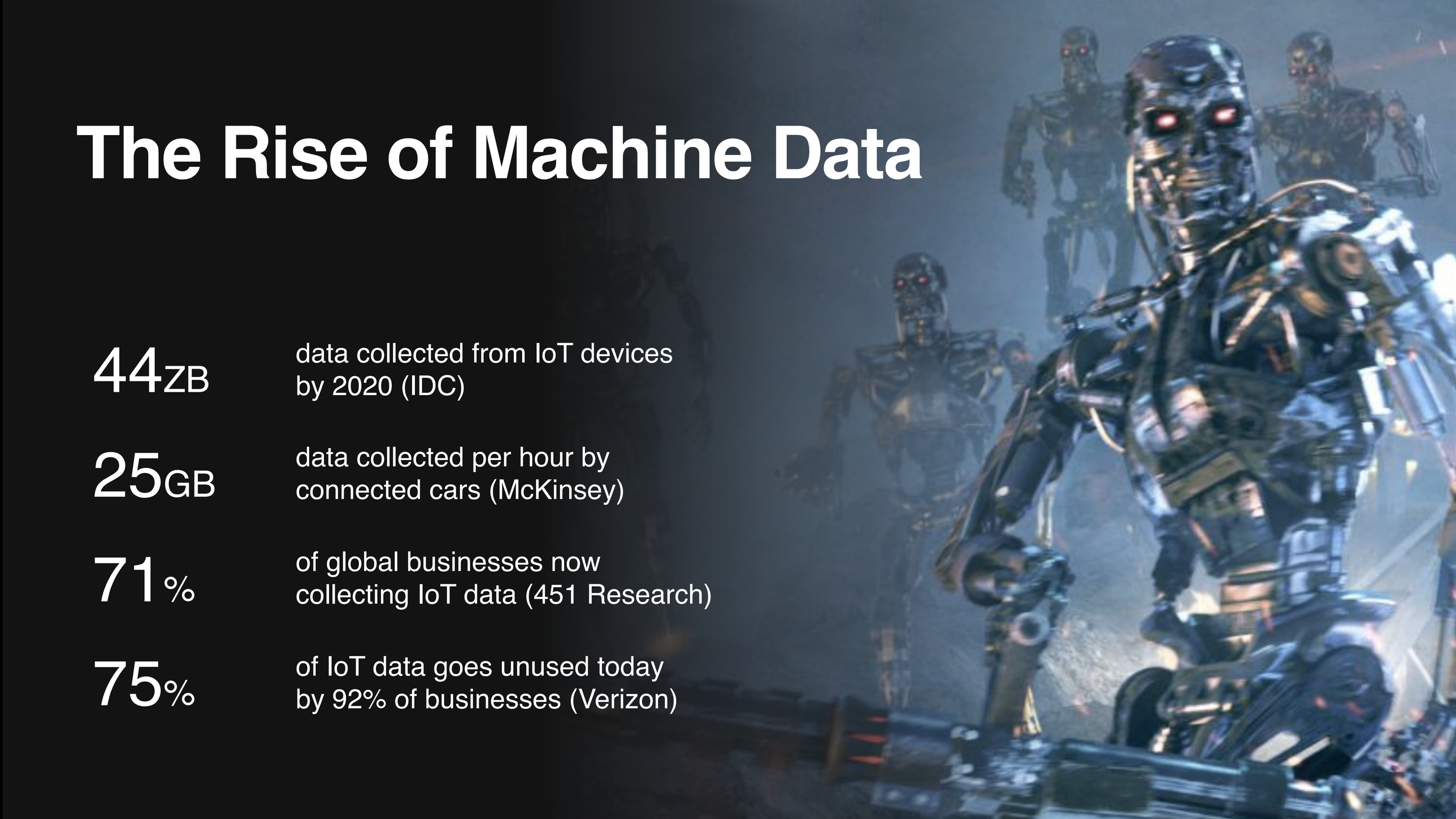
The Rise of Machine Data

44ZB data collected from IoT devices by 2020 (IDC)

25GB data collected per hour by connected cars (McKinsey)

71% of global businesses now collecting IoT data (451 Research)

75% of IoT data goes unused today by 92% of businesses (Verizon)



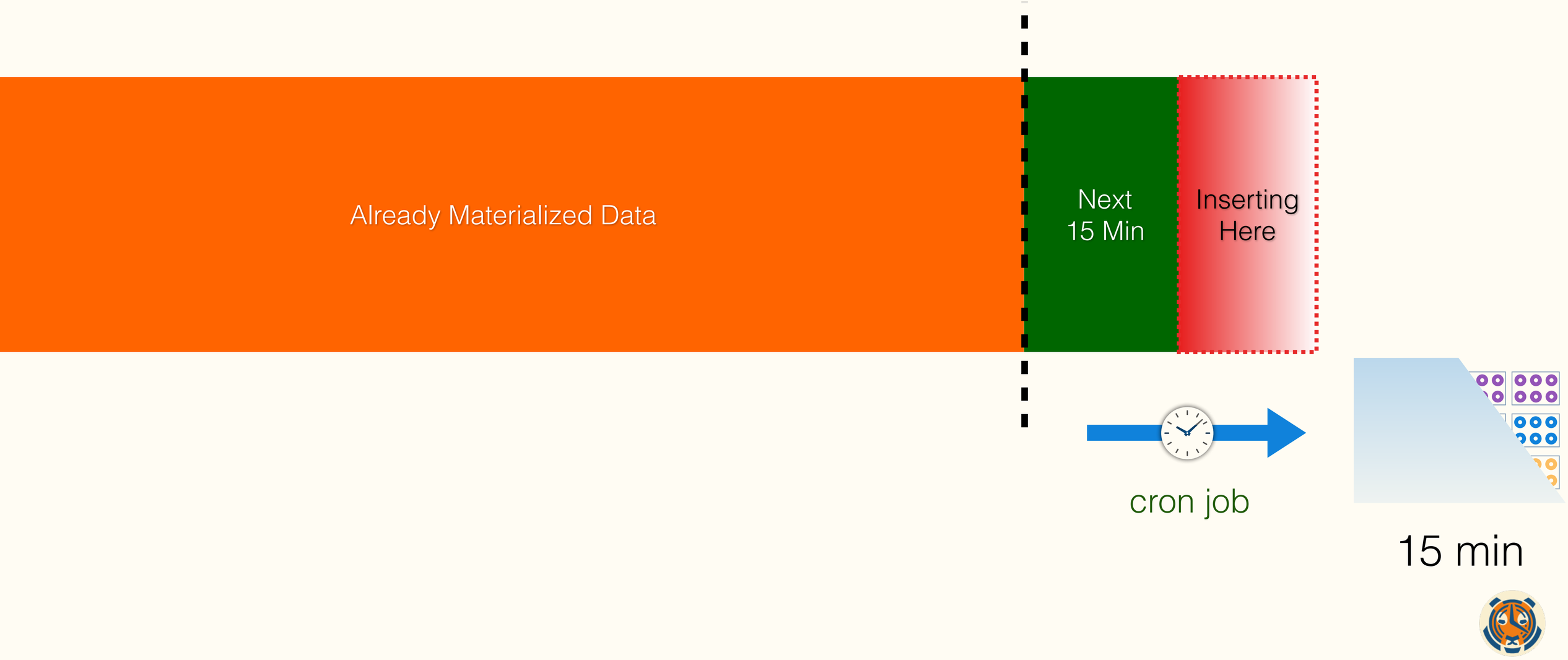
OLTP

- 👎 Primarily UPDATEs
- 👎 Writes randomly distributed
- 👎 Transactions to multiple primary keys

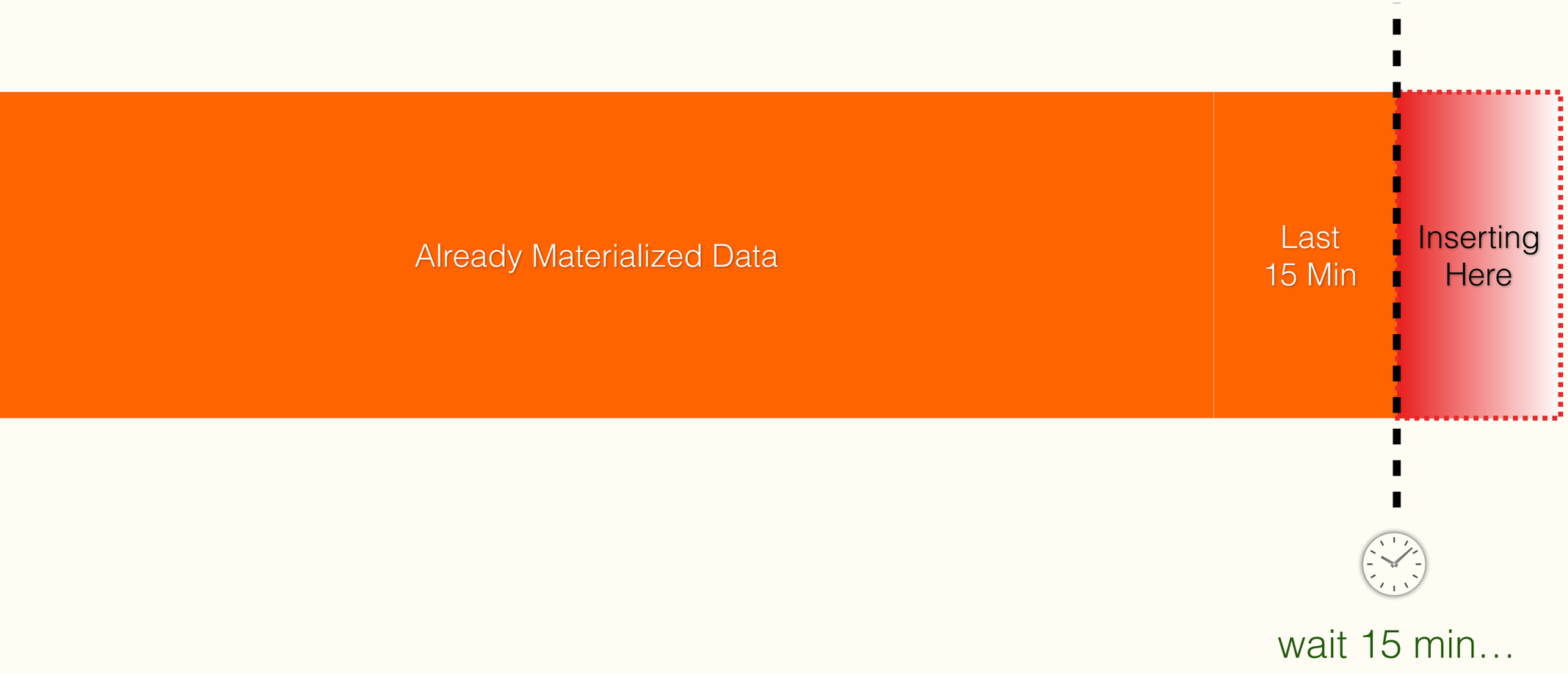
Time-series

- 👍 Primarily INSERTs
- 👍 Writes to recent time interval
- 👍 Writes primarily associated with a timestamp

Okay, okay, how about a cron job?



Okay, okay, how about a cron job?

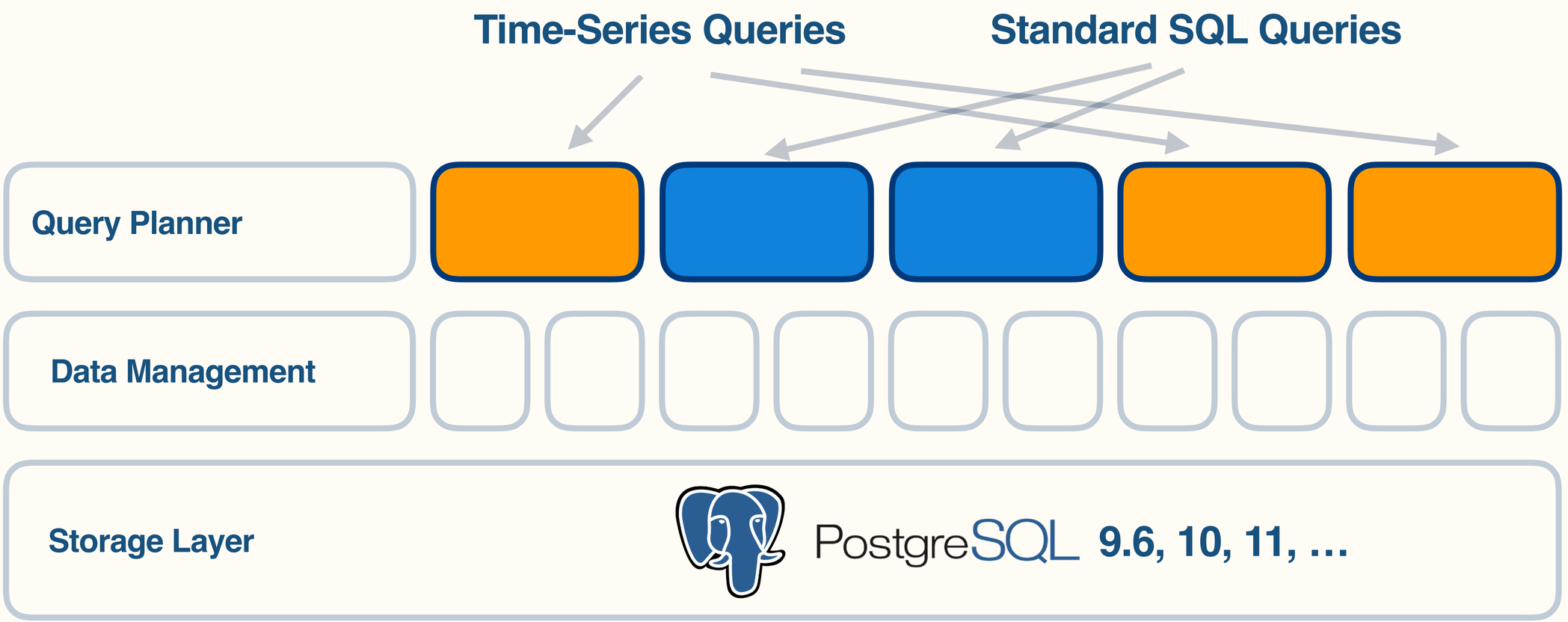


Okay, okay, how about a cron job?

- Deals with most inserts as they are usually to the most recent period
- Reduced write amplification because aggregates are computed once per period
- ***BUT...*** *what about late writes? Deletes? Updates?*
  - Synchronization issues often arise

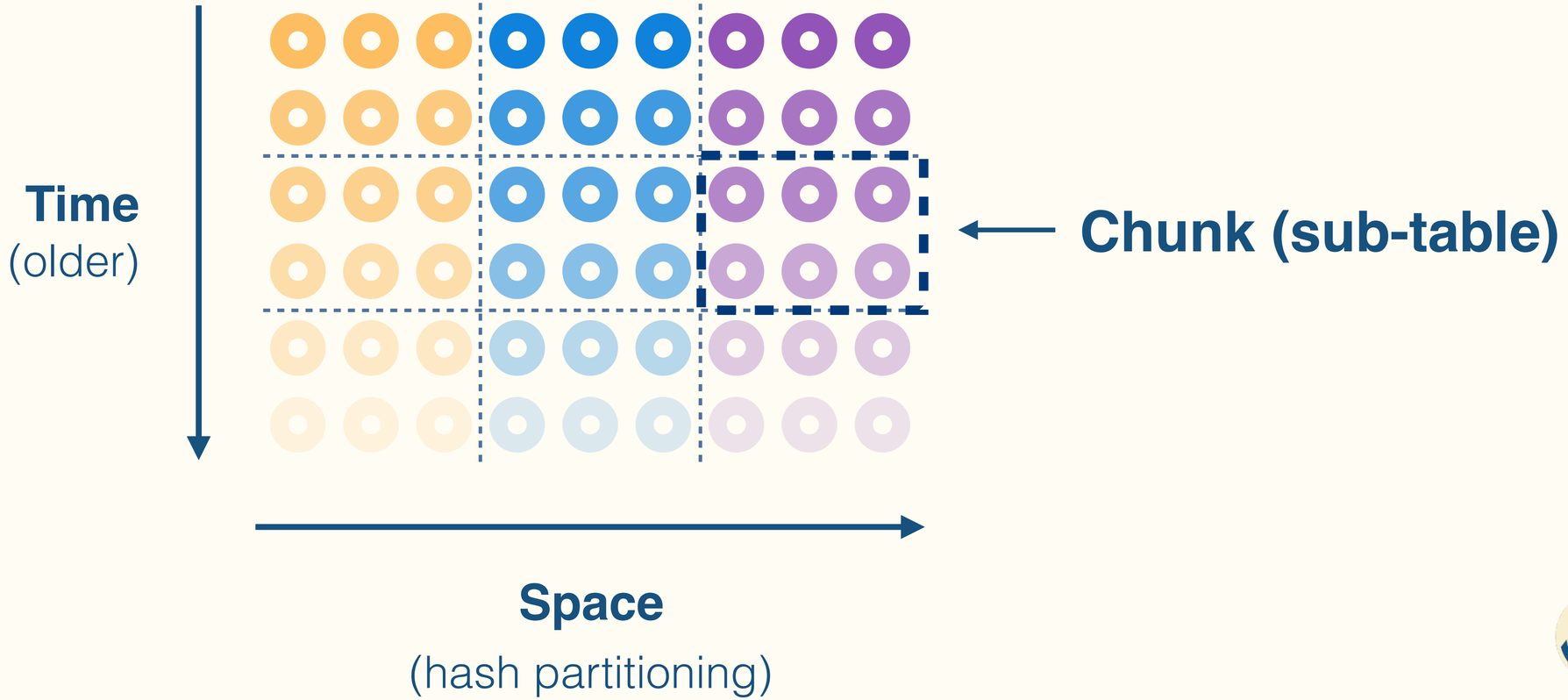
Enter **TimescaleDB**

The Extensibility of PostgreSQL

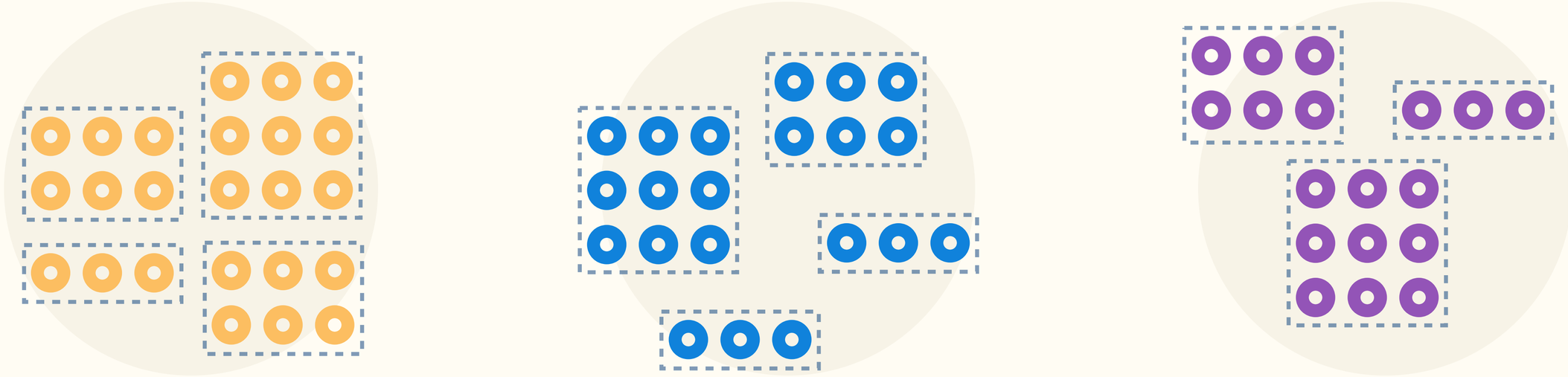


Time-space partitioning

(for both scaling up & out)



Chunks should be **“right-sized”**

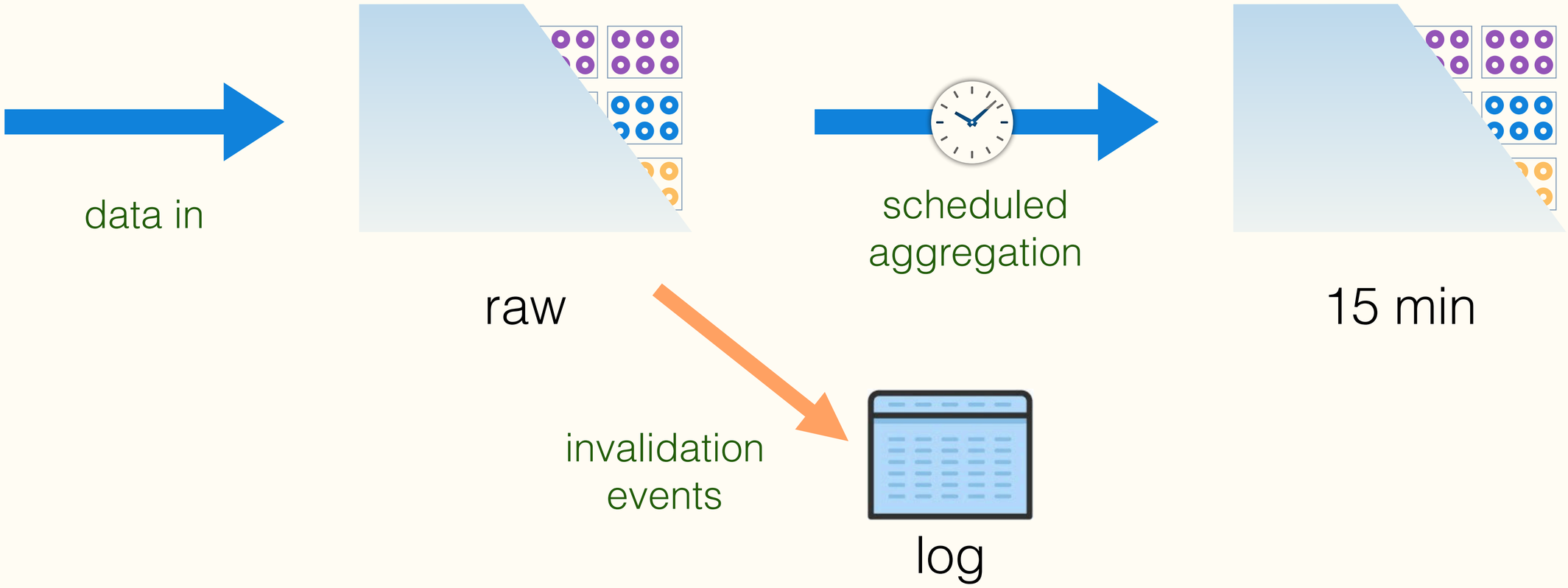


Recent (hot) chunks fit in memory

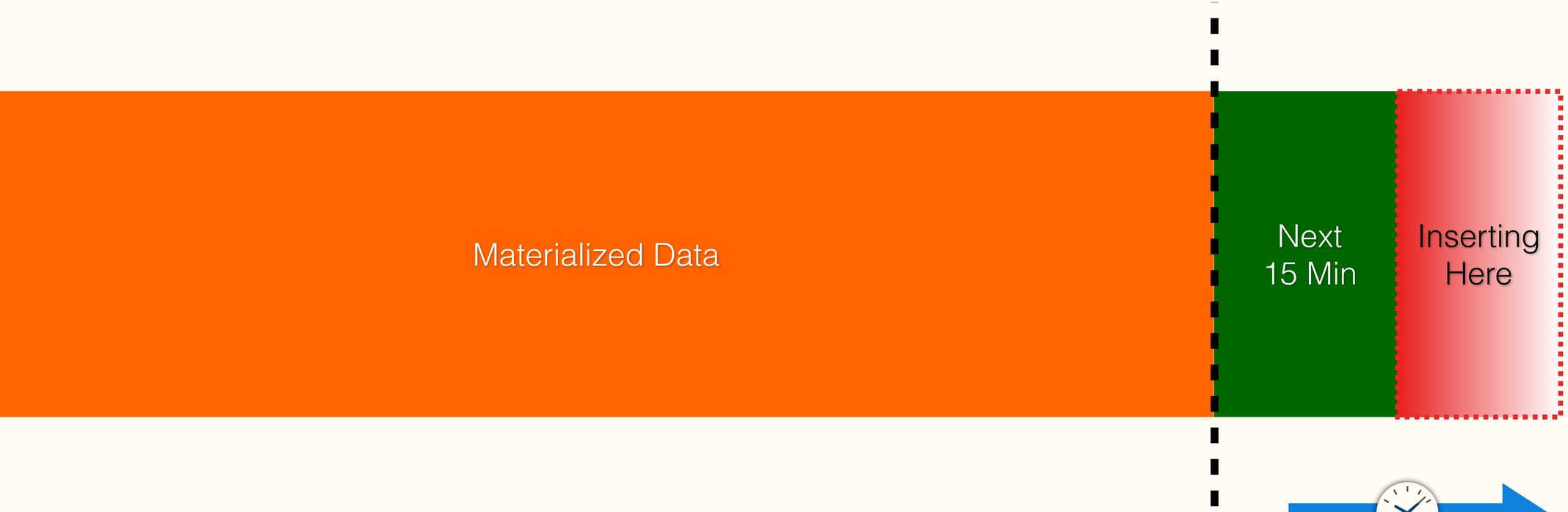
Automatic Space-time Partitioning

Continuous aggregates

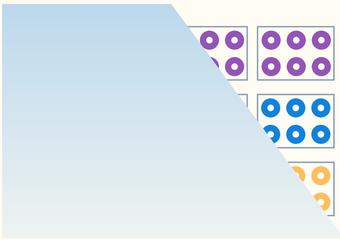
TimescaleDB Continuous Aggregates



TimescaleDB Continuous Aggregates

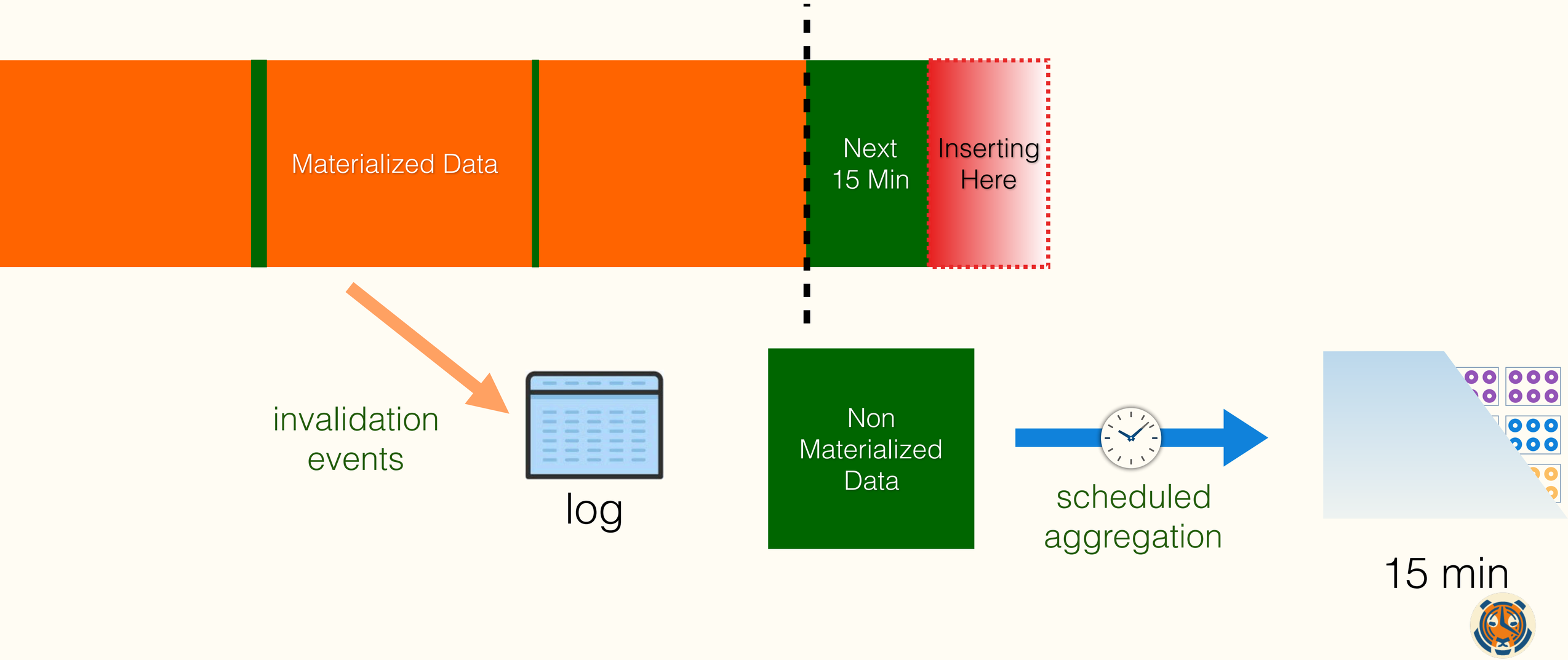


scheduled
aggregation



15 min

TimescaleDB Continuous Aggregates



TimescaleDB Continuous Aggregates

- Designed for high volume, mostly ordered, insert-mostly workloads
- Minimal write amplification, while maintaining correctness
  - None for writes more recent than threshold
  - One row per-statement invalidation overhead older than threshold
  - Meticulously avoid locking issues using PG transactional guarantees
- Maintained consistently without user intervention

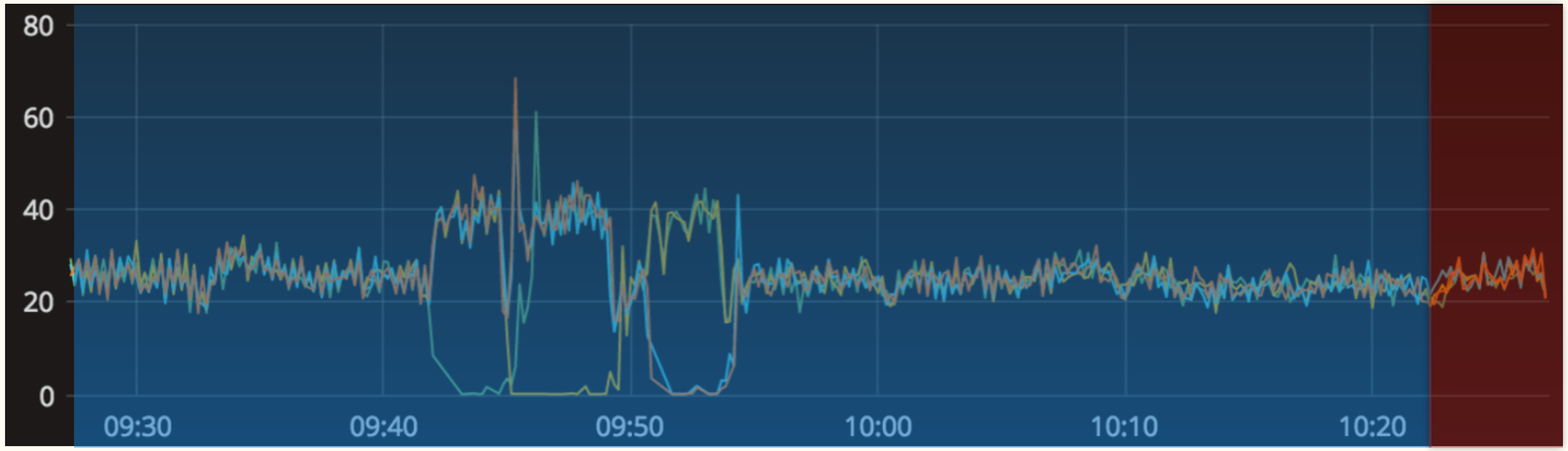
TimescaleDB Continuous Aggregates

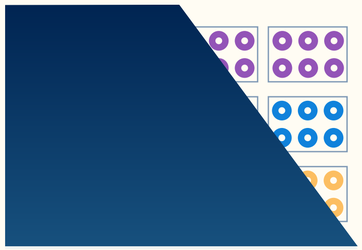
```
CREATE VIEW ohlc_continuous
WITH (timescaledb.continuous)
AS SELECT time_bucket('15 min', trade_time),
  symbol,
  first(price, trade_time) as open,
  max(price) as high,
  min(price) as low,
  last(price, trade_time) as close,
  sum(num_shares) as total_volume
FROM trade_data
GROUP BY time_bucket('15 min', trade_time), symbol;
```

Coming Soon!

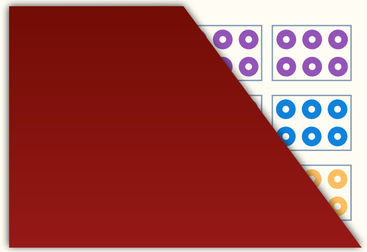
Single View Across Aggregated & Raw Data

```
SELECT * FROM ohlc_continuous;
```





15 min

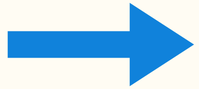


raw

In Development

Vision: Materialized View as an Index

```
SELECT symbol,
  max(high) as max_price
FROM ohlc_continuous
GROUP BY symbol;
```



```
SELECT symbol,
  max(price) as max_price
FROM trade_data
GROUP BY symbol;
```

- Why should your app need to know that the view exists?
- Indexes are transparent, why can’t a materialization be?

The Problem With Average

```
CREATE VIEW ohlc_continuous
WITH (timescaledb.continuous)
AS SELECT time_bucket('15 min', trade_time),
  symbol,
  first(price, trade_time) as open,
  max(price) as high,
  min(price) as low,
  last(price, trade_time) as close,
  sum(num_shares) as total_volume,
  avg(price) as avg_price
FROM trade_data
GROUP BY time_bucket('15 min', trade_time), symbol;
```

Partial Aggregation

- Example: for average store the sum and the count
  - Combine by summing each of them
  - Finalize by dividing the sum by the count.
- All ***parallelizable aggregates*** Postgres must have partial aggregation, combine and finalize functions defined
- Instead of storing the final state of the aggregate, we store partials and then combine and finalize at run time

Re-Grouping

```
SELECT time_bucket('2 hours', trade_time),
  symbol,
  avg(avg_price) as avg_price
FROM ohlc_continuous
GROUP BY time_bucket(‘2 hours', trade_time), symbol;
```

```
SELECT time_bucket('15 min', trade_time),
  avg(avg_price) as avg_price
FROM ohlc_continuous
GROUP BY time_bucket('15 min', trade_time);
```

In Development

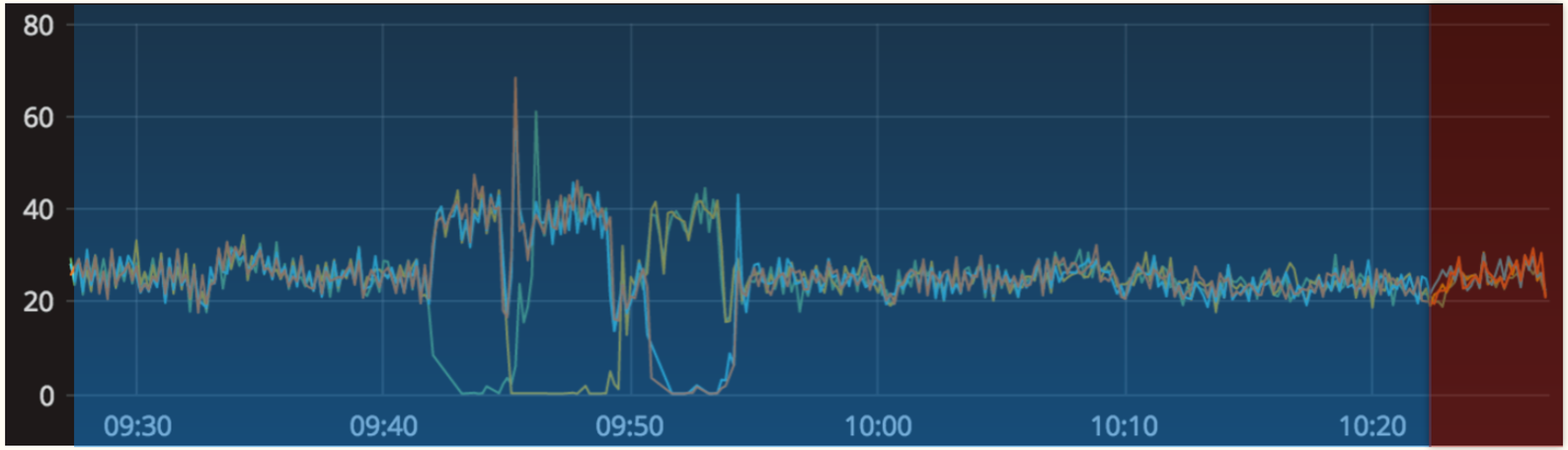
Re-Grouping With Exact Results

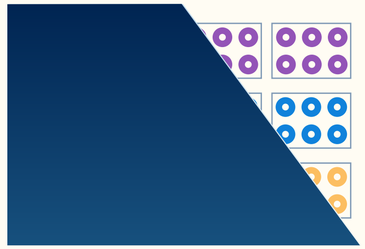
```
SELECT time_bucket_regroup('2 hours', trade_time),
  symbol,
  avg(avg_price) as avg_price
FROM ohlc_continuous
GROUP BY time_bucket_regroup(‘2 hours', trade_time), symbol;
```

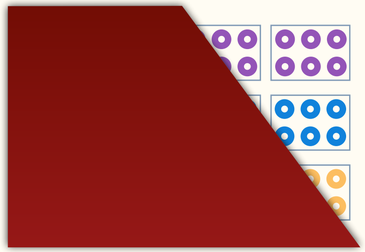
```
SELECT time_bucket_regroup('15 min', trade_time),
  avg(avg_price) as avg_price
FROM ohlc_continuous
GROUP BY time_bucket_regroup('15 min', trade_time);
```

Coming Soon!

Data Retention





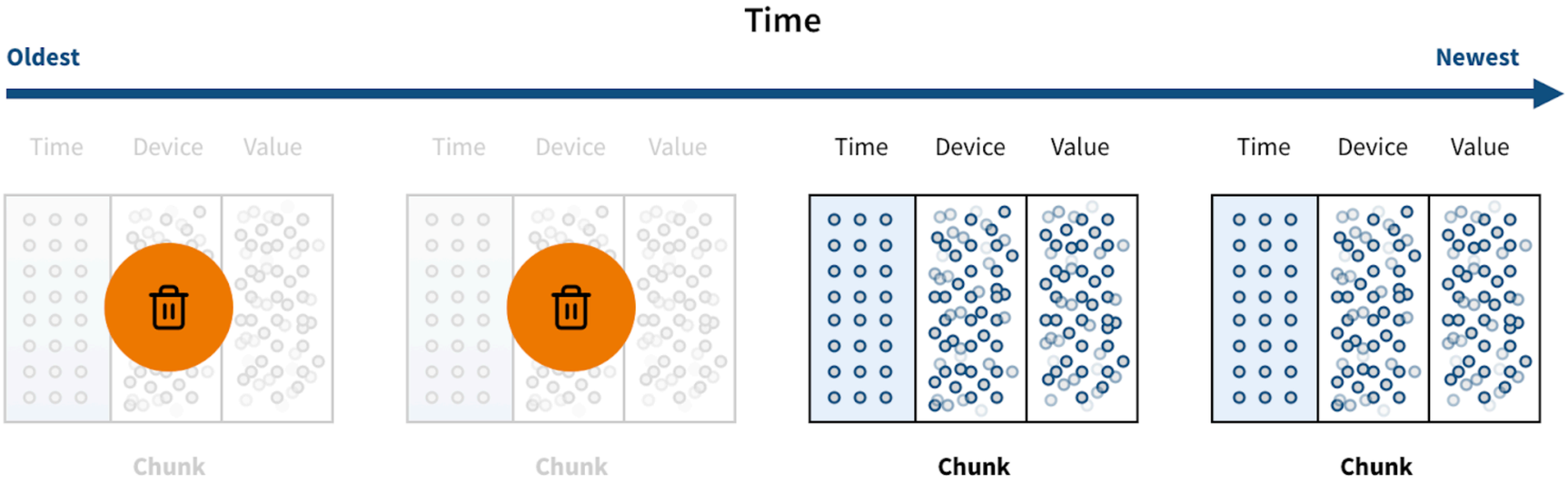


Granularity	15 min	raw
Retention	3 years	2 weeks

Powerful database **automation**

- Data reordering policies
- Data retention policies
- Data archival policies
- Data tiering policies
- Continuous aggregation policies

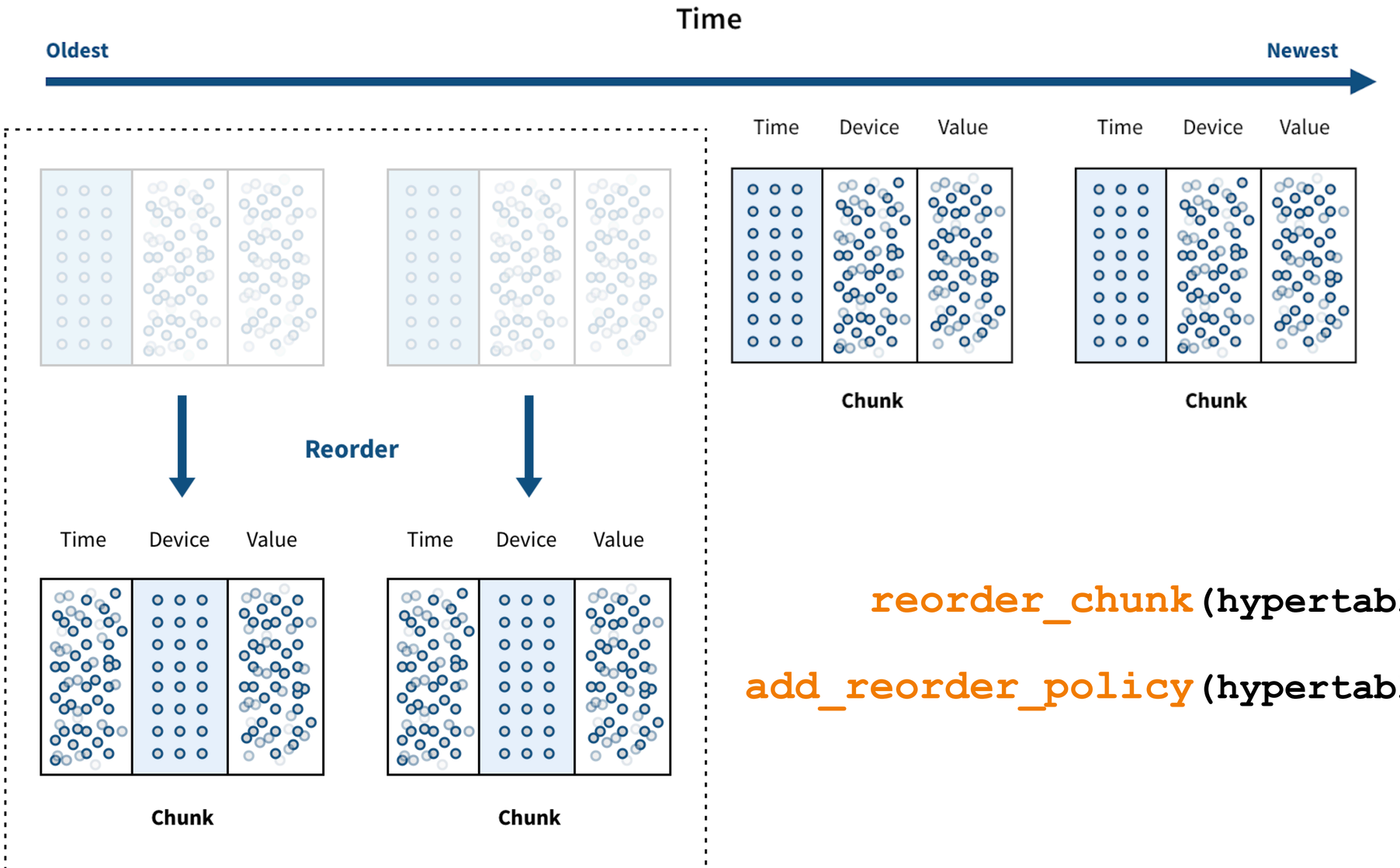
Automated **data retention**




```
SELECT drop_chunks(hypertable, interval);

SELECT add_drop_chunks_policy(hypertable, interval);
```

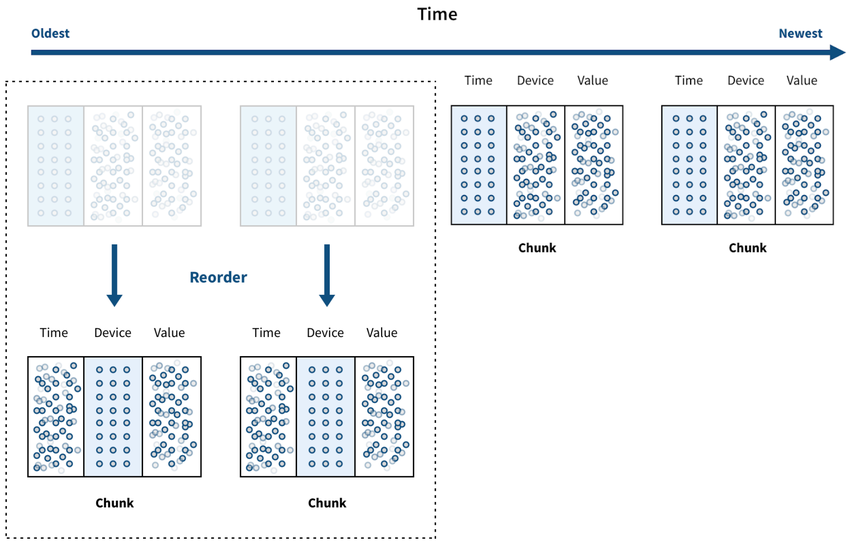
Automated **data reordering**




```
reorder_chunk(hypertable, index);

add_reorder_policy(hypertable, index);
```

Automated **data reordering**



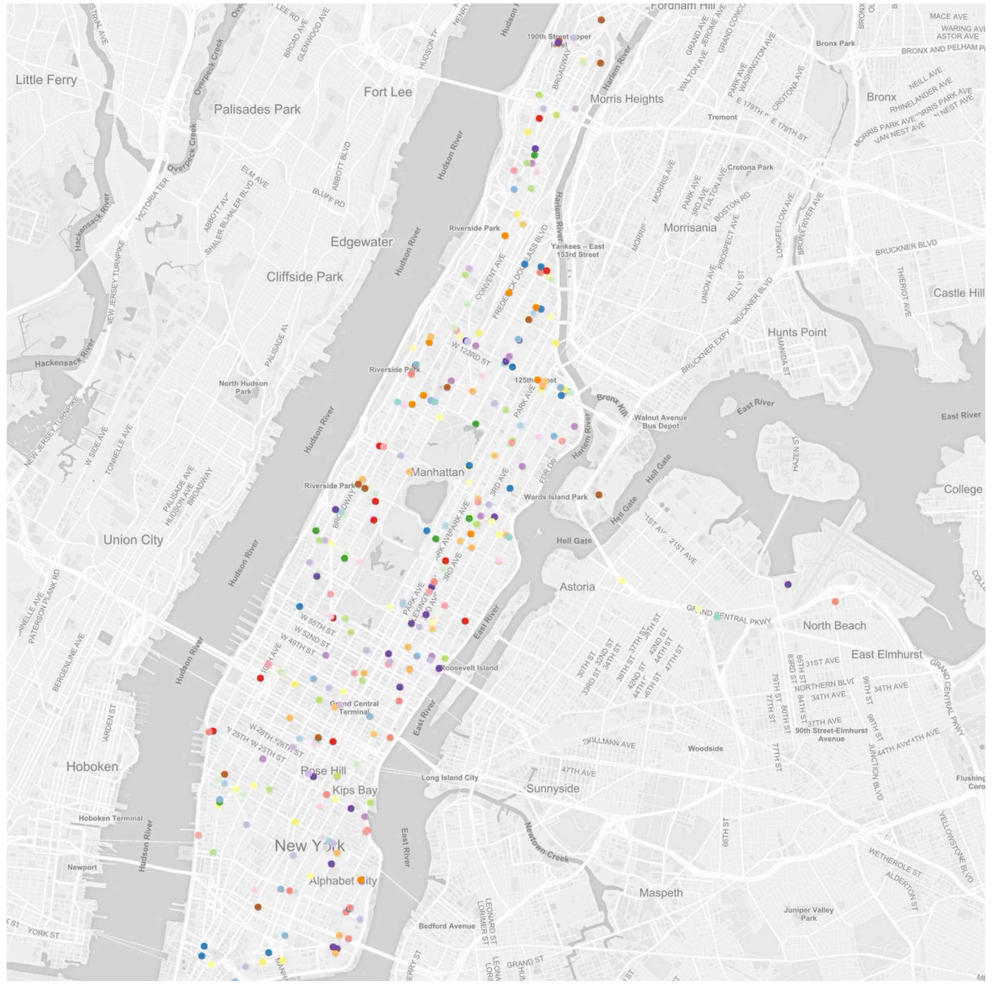

```
=> SELECT * FROM mta WHERE route_id = 'B39';
```

Heap Blocks: exact=20173; Execution Time: 12099 ms

```
=> SELECT reorder_chunk(…,'idx_mta_route');
```

```
=> SELECT * FROM mta WHERE route_id = 'B39';
```

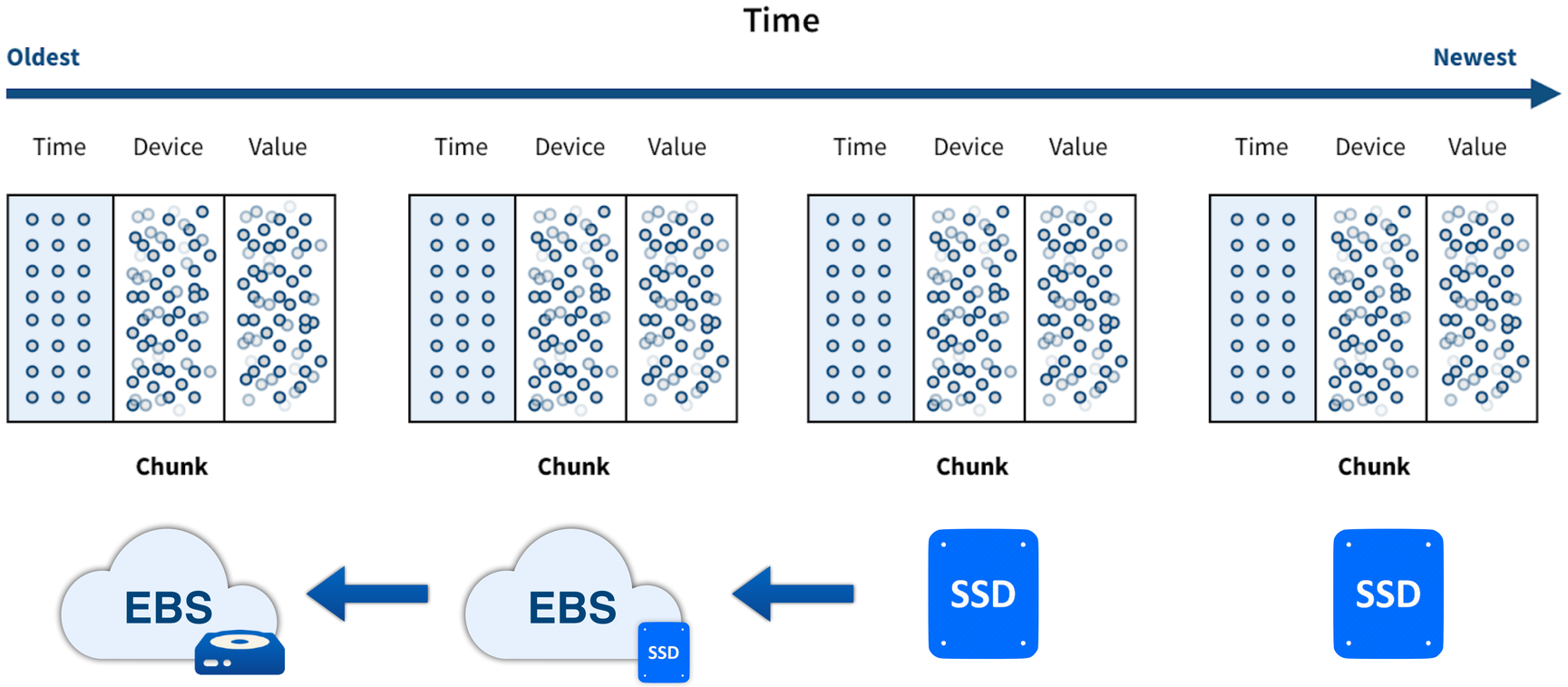
Heap Blocks: exact=250; Execution Time: 3.690 ms



https://github.com/timescale/mta-timescale

TimescaleDB 1.5+

Automated **data tiering**




```
SELECT migrate_chunks(hypertable, interval, to, from);

SELECT add_migrate_chunks_policy(hypertable, interval, to, from);
```

Source code

- github.com/timescale/timescaledb

Join the Community

- slack.timescale.com

TIMESCALE
Powered by aiven

30 days remaining in your free trial | Credits $300 | Billing $0.00 | Welcome Michael Freedman | Help | Logout

Current project:
mike-cf3c

- mike-cf3c
- + Create new project

- Services
- Events
- Members
- VPC
- Billing

Current services

+ Create a new service

Service	Plan	Cloud	Created
tsdb-ha-pair-google-cloud-1 TimescaleDB • Running	Timescale-pro-1024-io-optimized 4 CPU / 15 GB RAM / 1024 GB storage - high availability pair	Timescale / GCP: google-europe-west1 Europe, Belgium	10 minutes ago
grafana-aws-1 Grafana • Running	Dashboard-1 2 CPU / 1 GB RAM	Timescale / AWS: aws-us-east-2 United States, Ohio	19 minutes ago
	escale-basic-512-io-optimized U / 15 GB RAM / 512 GB storage	Timescale / AWS: aws-us-east-2 United States, Ohio	19 minutes ago
	escale-basic-512-io-optimized U / 8 GB RAM / 512 GB storage	Timescale / GCP: google-us-central1 United States, Iowa	20 minutes ago

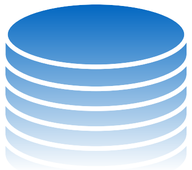
timescale.com/cloud-promo

TIMESCALE

In Development

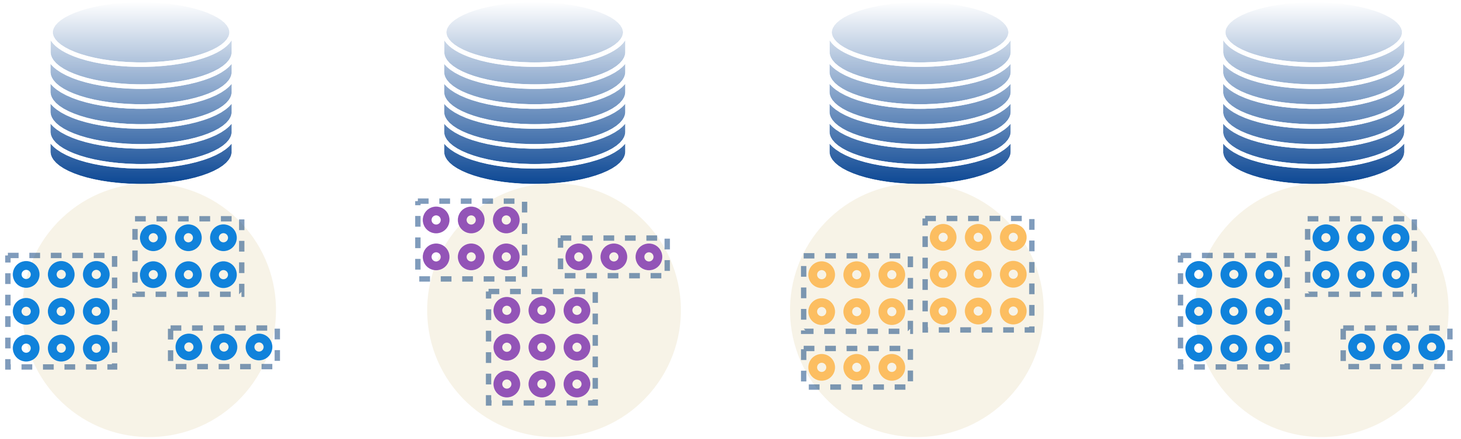
TimescaleDB scale-out clustering

"Front-end"
TimescaleDB



Cluster-wide catalog info, server → chunks

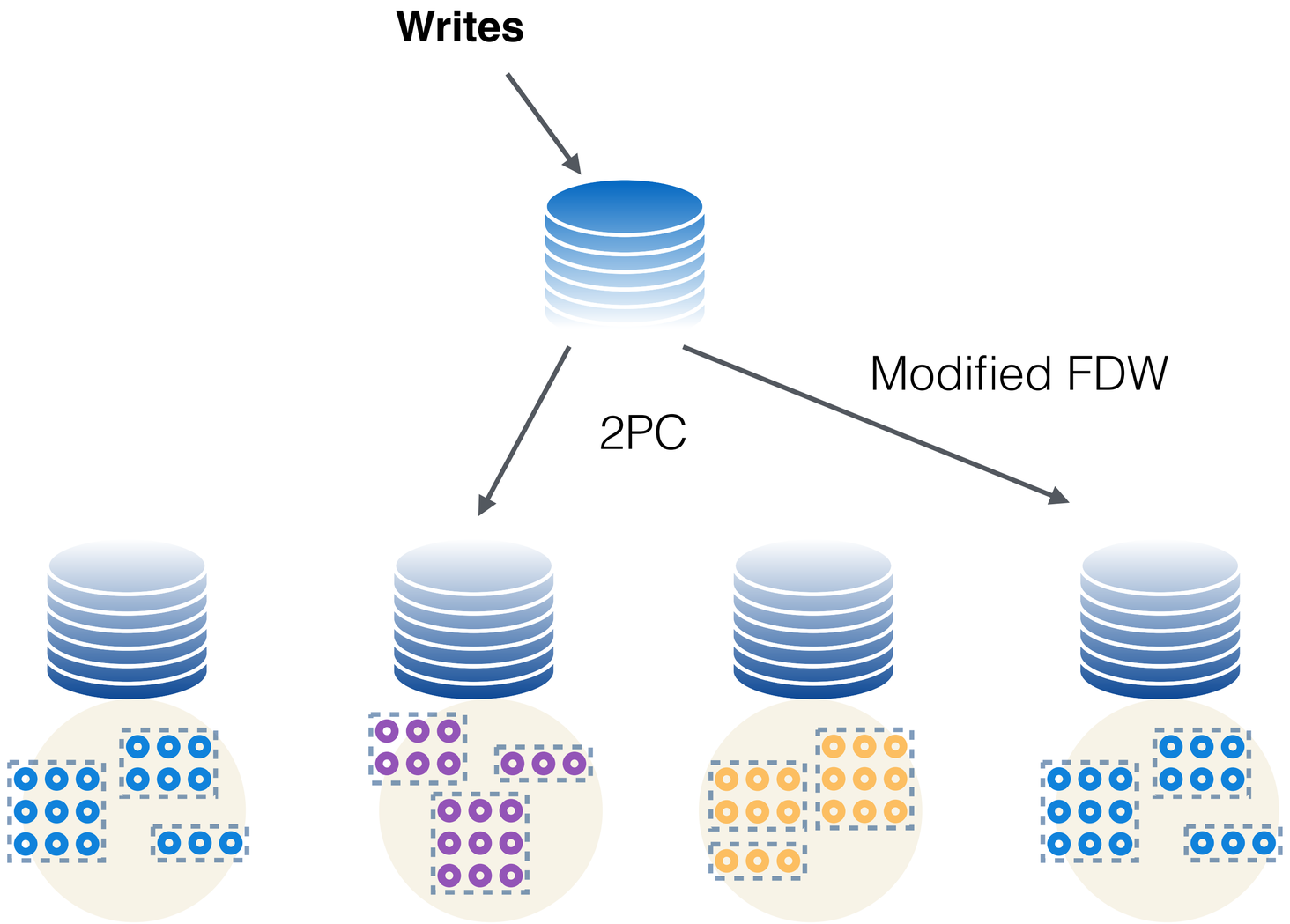
"Back-end"
TimescaleDB



Local catalog info

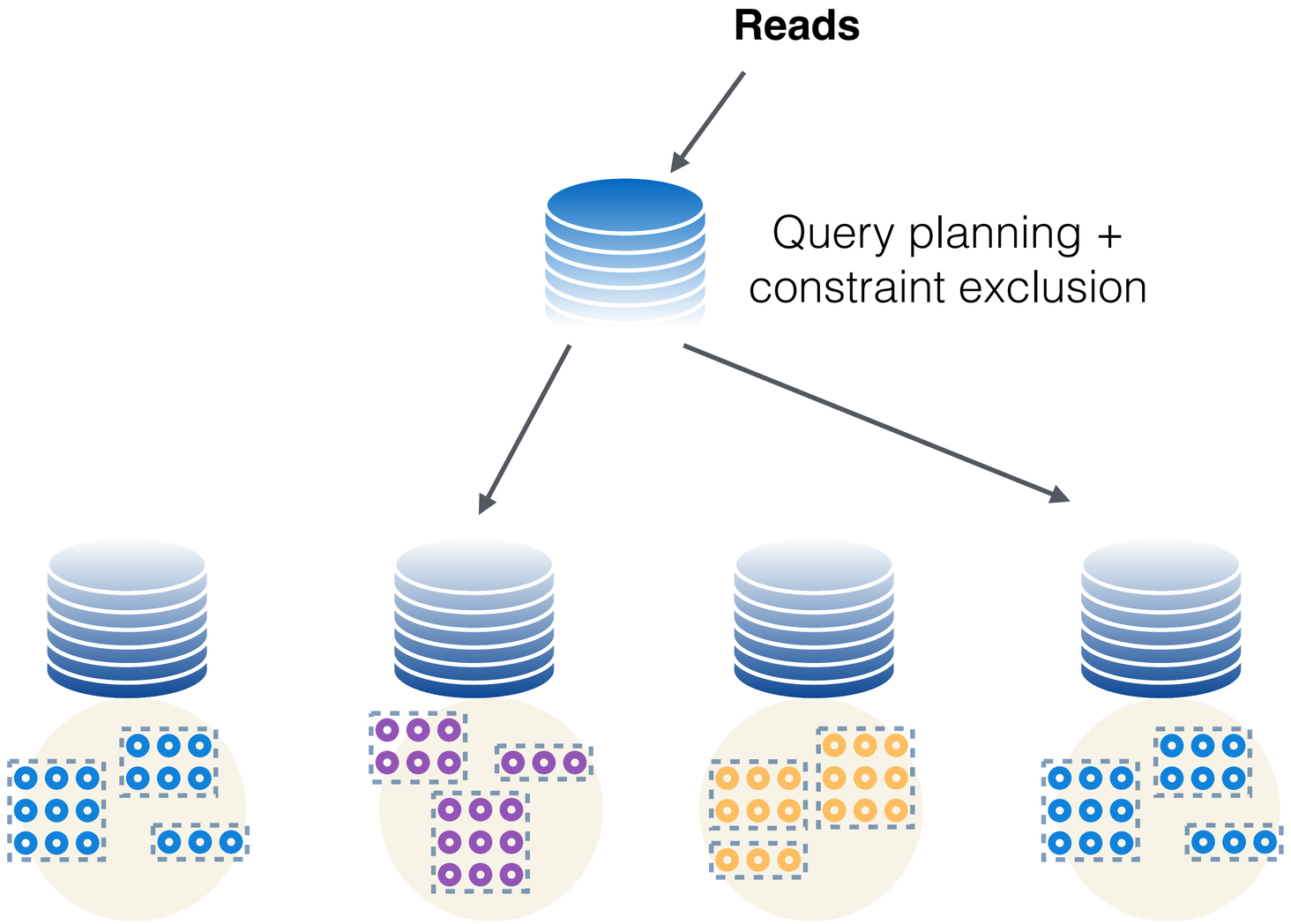
In Development

TimescaleDB scale-out clustering



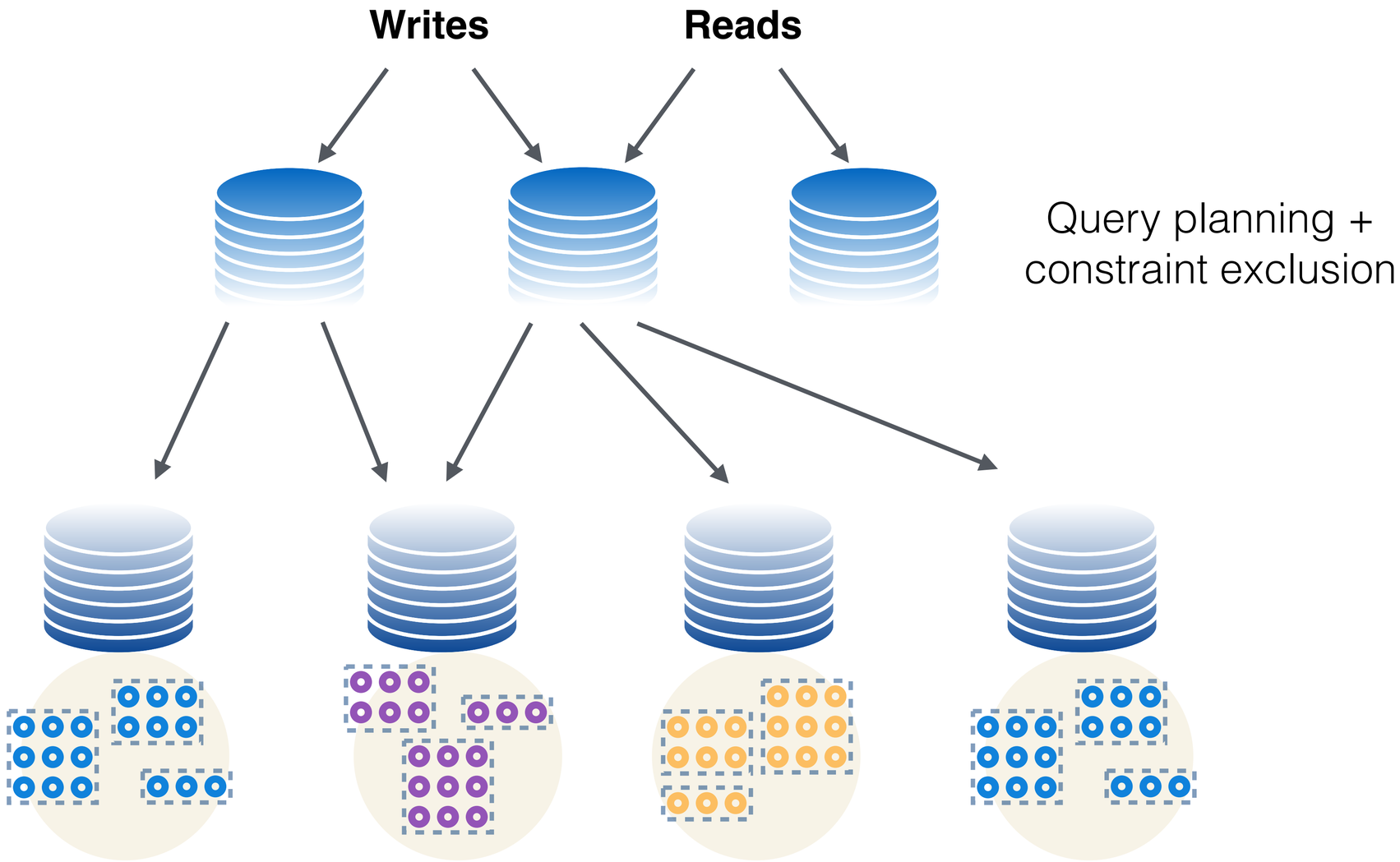
In Development

TimescaleDB scale-out clustering



In Development

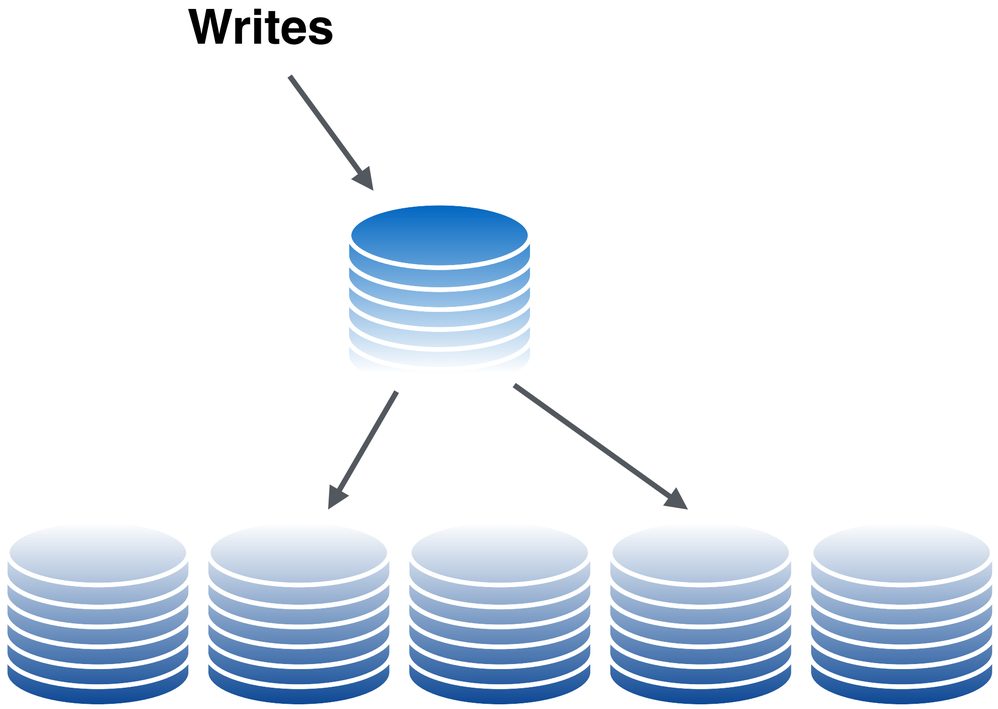
TimescaleDB scale-out clustering



Query planning +
constraint exclusion

In Development

Preliminary benchmarks

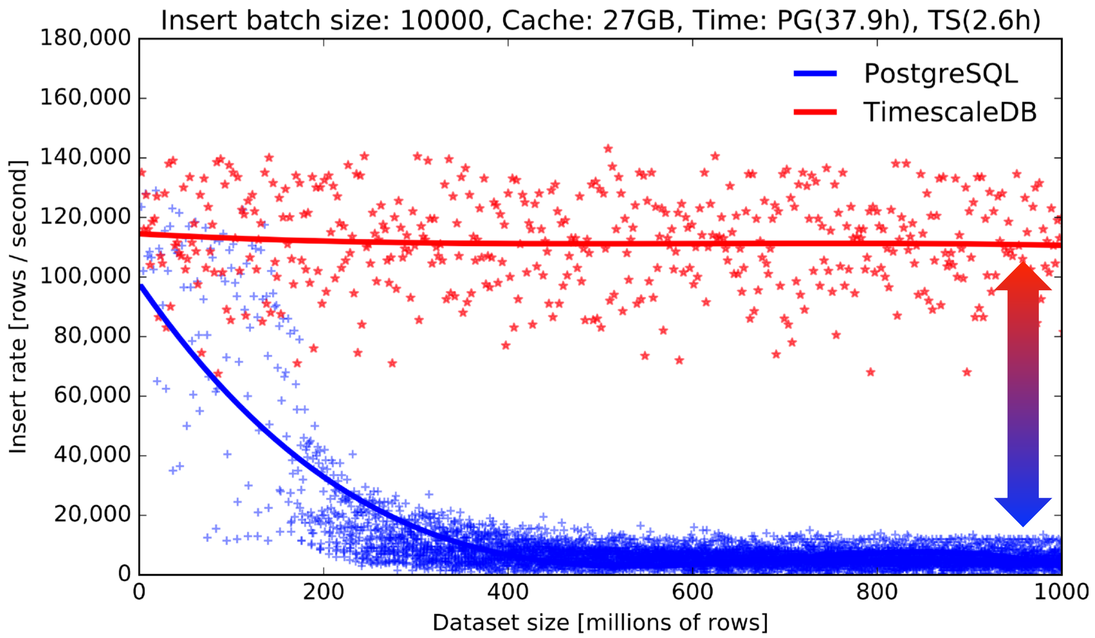


AWS m5.12xlarge, EBS storage

8 data servers, unreplicated	850K rows/s 8.5M metrics/s
8 data servers, 2-way replicated	700K rows/s 7M metrics/s

vs. PostgreSQL

20x Higher Inserts



Faster Queries

	Speedup
Table scans, simple column rollups	**0-20%**
GROUPBYs	**20-200%**
Time-ordered GROUPBYs	**400-10000x**
DELETEs	**2000x**