Microsoft

Scaling real-time analytics using Postgres in the cloud

Sai Krishna Srirampur Sai.Srirampur@microsoft.com
Colton Shepard Colton.Shepard@microsoft.com

Why PostgreSQL?

Proven Resilience and Stability

Thousands of Mission Critical Workloads

Open source

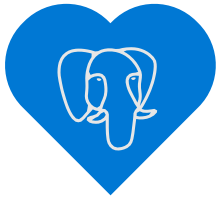
Large Developer Community and Extensible

Rich Feature Set: Solves multitude of use cases

Constraints
Extensions
PostGIS / Geospatial
HLL, TopN, Citus
Foreign data wrappers
JSONB

Rich SQL
CTEs
Window functions
Full text search
Datatypes

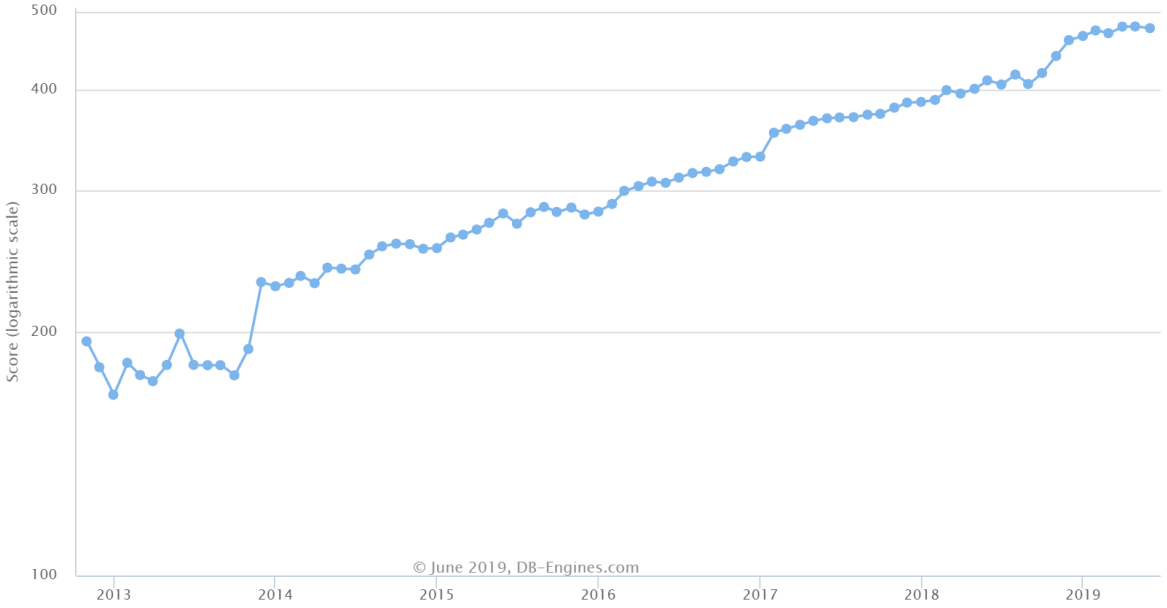
PostgreSQL is more popular than ever



One of the most **loved** and **wanted** databases in Stack Overflow's 2019 Developer Survey

Ranked 2018 **DBMS of the Year** by DB-Engines

DB-Engines' ranking of PostgreSQL popularity



https://insights.stackoverflow.com/survey/2019?utm_source=so-owned&utm_medium=blog&utm_campaign=dev-survey-2019&utm_content=launch-blog
https://db-engines.com/en/blog_post/76
https://db-engines.com/en/ranking_trend/system/PostgreSQL

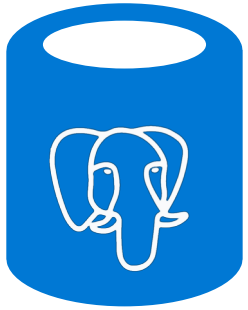
The cloud only makes PostgreSQL better

More and more organizations are shifting open source workloads to the cloud to benefit from key advantages:

- Improved manageability and security
- Improved performance and intelligence
- Global scalability

Azure Database for PostgreSQL

Azure Database for PostgreSQL is available in two deployment options



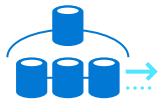
Enterprise-ready, fully managed community PostgreSQL with built-in HA and multi-layered security

Single Server

Fully-managed, single-node PostgreSQL

Example use cases

- Apps with JSON, geospatial support, or full-text search
- Transactional and operational analytics workloads
- Cloud-native apps built with modern frameworks



Hyperscale (Citus)

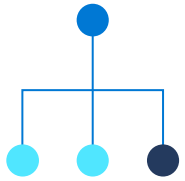
High-performance Postgres for scale out

Example use cases

- Scaling PostgreSQL multi-tenant, SaaS apps
- Real-time operational analytics
- Building high throughput transactional apps

The benefits of Azure Database for PostgreSQL

Build or migrate your workloads with confidence

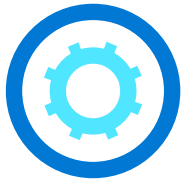


Fully managed and secure

Focus on your apps while Azure manages resource-intensive tasks, supports a large variety of Postgres versions and provides best-in industry indemnification coverage

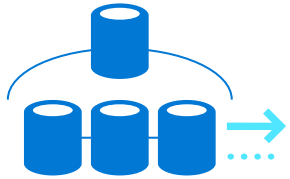
Intelligent performance optimization

Improve performance and reduce cost with customized recommendations



Flexible and open

Stay productive with your favorite Postgres extensions and leverage Microsoft's contributions to the Postgres community



High performance scale-out with Hyperscale (Citus)

Break free from the limits of single-node Postgres and scale out across hundreds of nodes

Single Server

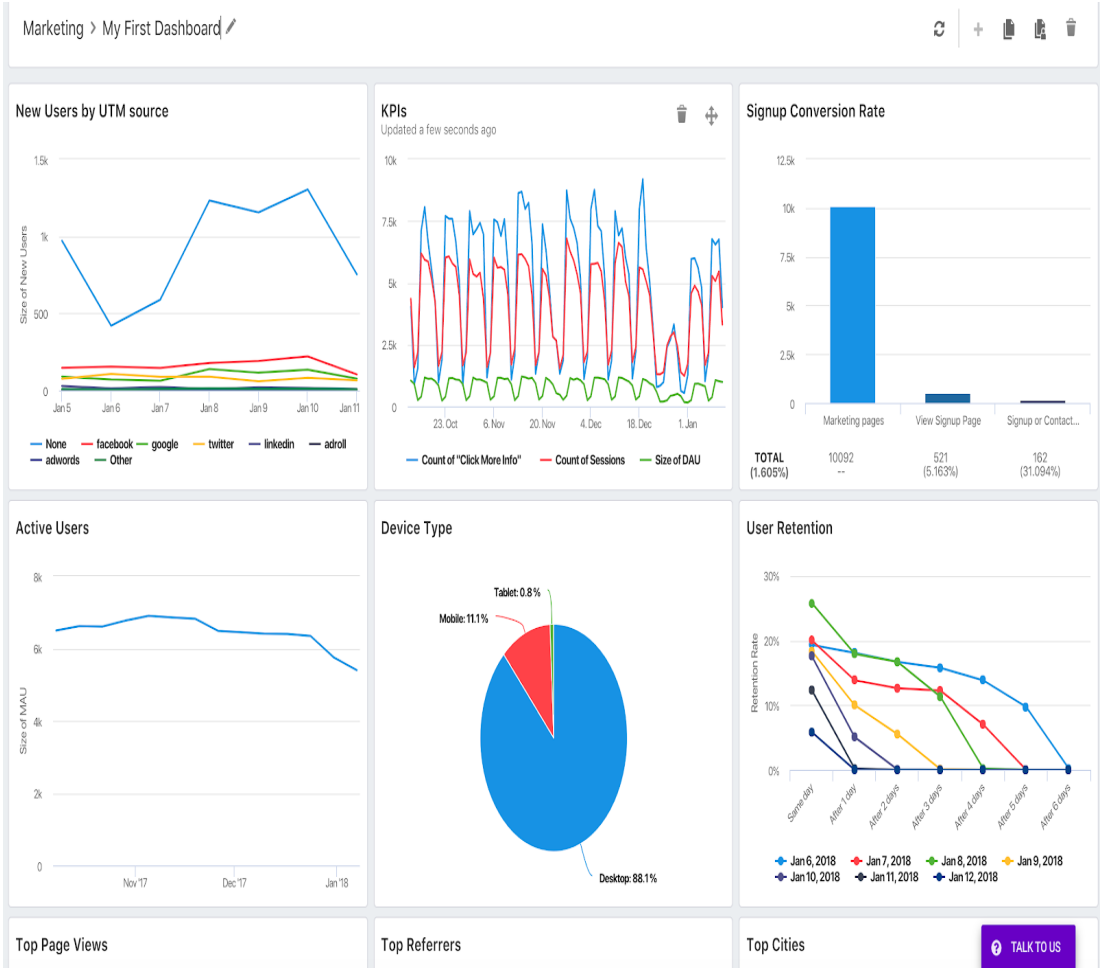
Hyperscale (Citus) NEW

Real time Analytics

Scenario: Real-time Analytics

- You offer a product or service which allows customers to run reports and analytics queries on-the fly on recent data.

- Examples:
- network telemetry
- clickstream analysis
- IoT



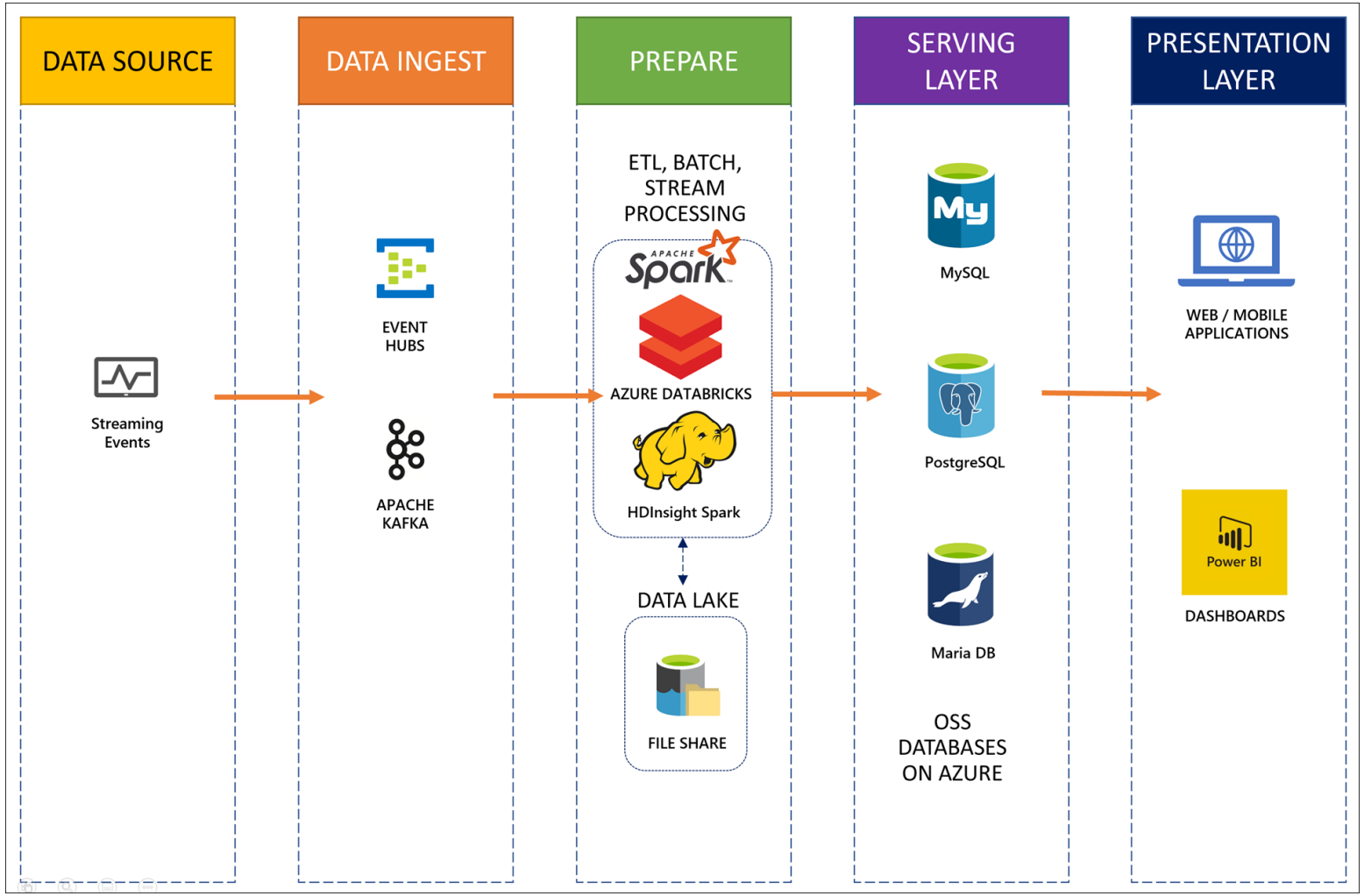
Common requirements for Real-time analytics applications

A real-time application generally:

- Generates large amounts of data
- Has sub-second response times
- Supports large number of concurrent users
- Reflects new data within minutes
- Supports advanced analytics

Architecting Real-time Analytics with Postgres in the cloud

Example Architecture for a real-time analytics application



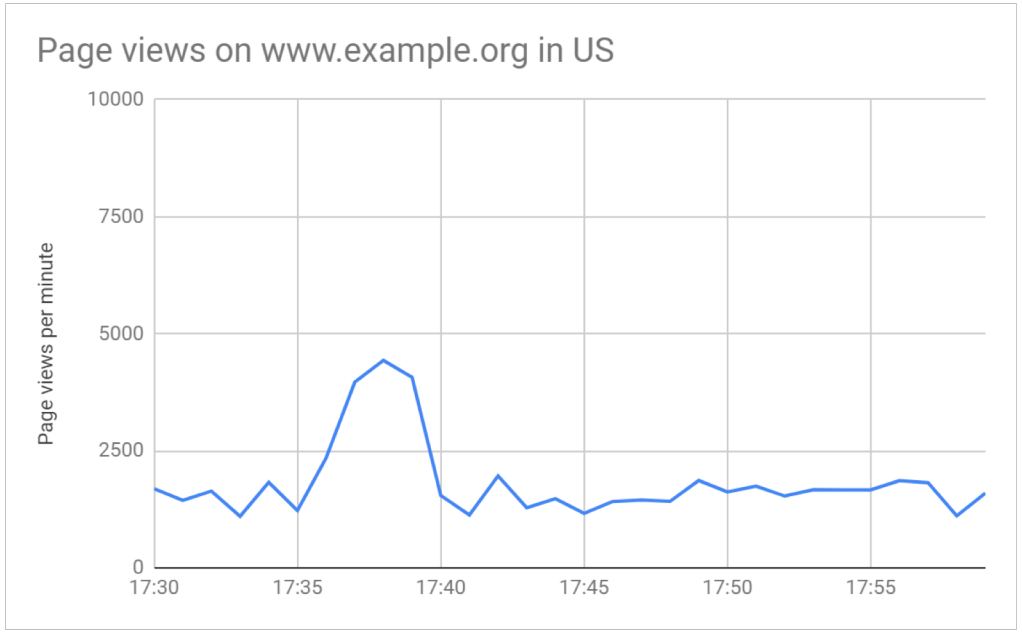
Typical Recipe for real-time analytics

- Ingest large volumes of data to a raw table
- Periodically aggregate events into a rollup table
- Have application query the rollup table

What is a rollup table?

- Pre-computed aggregates for a period and set of (group by) dimensions.

Period	Customer	Country	Site	Hit Count



Why Rollup tables?

- Fast (indexed) lookups of aggregates
- Compute-heavy work done periodically, in the background
- Rollups can be further aggregated
- Rollups are compact, can be kept over longer periods
- Advanced analytics using HLL, TopN
- Horizontal scale out using Citus

Typical Recipe for real-time analytics

- Ingest large volumes of data to a raw table
- Periodically aggregate events into a rollup table
- Have application query the rollup table

Schema for ingesting data into

```
CREATE TABLE events(
  event_id bigserial,
  event_time timestamptz default now(),
  customer_id bigint,
  event_type text,
  country text,
  browser text,
  device_id bigint,
  session_id bigint,
  details jsonb
);
```

Fast data loading - COPY

- COPY is the fastest way to bulk load data into Postgres
- With a few parallel streams, you can load millions of rows per second.

- Doesn’t need to be large batches.
- Can micro-batch in groups of 10s-100s of thousands of rows.

Best practices for data loading

- Use COPY
- Don’t use Indexes

But,

- You need indexes for querying that large table.

So,

- Avoid large indexes!

Ways to have small indexes

- Use one of the many index types Postgres supports as appropriate.
  - Eg. BRIN – efficient block based indexing for sorted data.
  - Small in size and bad for unsorted data, but tailor-made for range queries on time-based sorted data.
- Expire unneeded data to keep data and indexes small.
- Break up large table into smaller partitions.
  - That way you need to scan only relevant portions of data and not the entire index for ingestion.

Expiring old data

- DELETE FROM events WHERE event_time <= '2018-08-05’;
- If you want to have a more informed delete, you can do:
  - DELETE FROM events WHERE event_time <= '2018-08-05’ and not important;

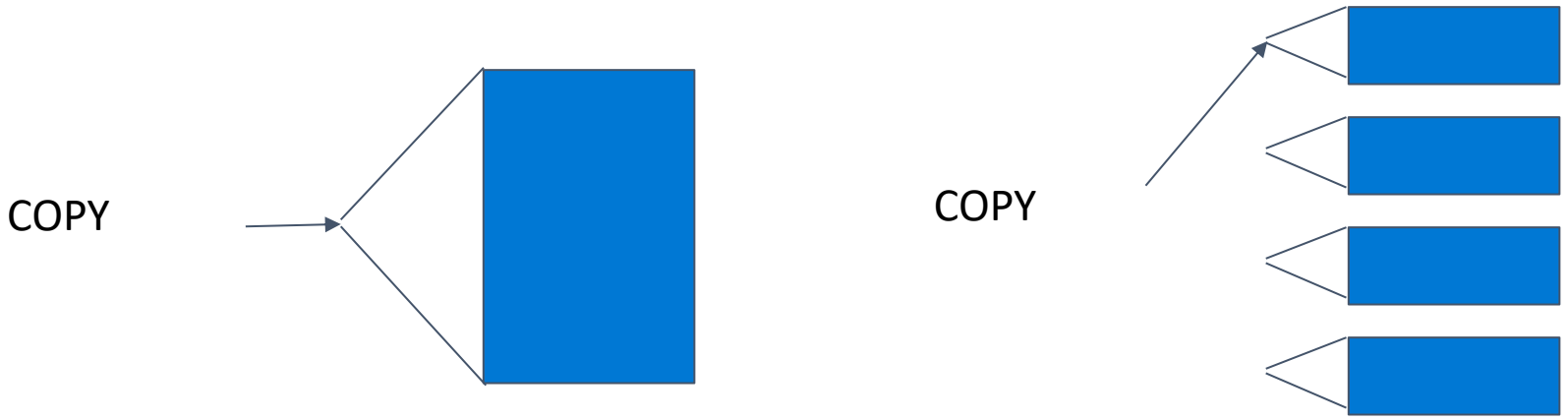
Bloat and Fragmentation

- Selective deletes create gaps and bloat.
- Autovacuum cleans them and new rows go into them.
- This leads to completely random rows and indexes – fragmentation
  - Affects load performance
  - Affects select performance

Partitioning

Keep your data sorted by bucketing it.

Partitioning keeps indexes small by dividing tables into partitions:



Benefits:

- **Drop old data quickly**, without bloat/fragmentation
- **Smaller indexes**
- **Partition pruning** for queries that filter by partition column

CREATE EXTENSION pg_partman

Defining a partitioned table:

```
CREATE TABLE events (...) PARTITION BY (event_time);
```

Setting up hourly partitioning with pg_partman:

```
SELECT partman.create_parent('public.events',
'event_time','native', 'hourly');
```

Now expiry becomes

- If you’re using partitioning, pg_partman can drop old partitions:

```
UPDATE partman.part_config
SET retention_keep_table = false, retention = '1 month'
WHERE parent_table = 'public.events';
```

Periodically run maintenance:

```
SELECT partman.run_maintenance(p_analyze := false);
```

Typical Recipe for real-time analytics

- Ingest large volumes of data to a raw table
- Periodically aggregate events into a rollup table
- Have application query the rollup table

Typical Structure of Rollups

```
CREATE TABLE rollup_by_period_and_dimensions (
    <period>
    <dimensions>
    <metrics>
    primary key (<dimensions>,<period>)
  );
```

Examples:

Period – 1min, 5min, 1hour

Dimensions – customer id, device id, location, country

Metrics – View count, byte count, number of distinct sessions

Choose granularity and dimensions

Time	Customer	Country	Aggregates

~100 rows per period/customer

Time	Customer	Site	Aggregates

~20 rows per period/customer

Time	Customer	Country	Site	Aggregates

~20*100=2000 rows per period/customer

Coarse-grained rollup for fast lookups

Coarse-grained rollup:

```
  CREATE TABLE hits_by_country (
       period timestamptz,
       customer_id bigint,
       country varchar(2),
       num_hits bigint,
       PRIMARY KEY (customer_id, country, period)
  );
```

Look up records by primary key columns:

```
  SELECT period, num_hits
  FROM hits_by_country
  WHERE customer_id = 1238 AND country = 'US'
  ORDER BY 1;
```

Fine-grained rollup for versatility

Fine-grained rollup:

```
  CREATE TABLE hits_by_country_site (
       period timestamptz,
       customer_id bigint,
       country varchar(2),
       site text,
       num_hits bigint,
       PRIMARY KEY (customer_id, country, site, period)
  );
```

Sum across all sites:

```
  SELECT period, sum(num_hits)
  FROM hits_by_country_site
  WHERE customer_id = 1238 AND country = 'US'
  GROUP BY period ORDER BY 1;
```

Build coarse grained from fine grained rollups

Can build coarse-grained rollup from a fine-grained rollup cheaply:

```
INSERT INTO hits_by_country_daily
SELECT period::date, country, sum(num_hits)
FROM hits_by_country_site
GROUP BY 1, 2;
```

Useful if you want to keep coarse-grained data for much longer.

Summary: Designing rollups

Find balance between query performance and table management.

1. Identify dimensions, metrics (aggregates)
2. Try rollup with all dimensions:
3. Test compression/performance (goal is **>5x smaller**)
4. If slow, split rollup table based on query patterns
5. Go to 3

Usually ends up with 5-10 rollup tables

Computing rollups

Append only vs Incremental

Use INSERT INTO rollup SELECT … FROM events … to populate rollup table.

Append-only aggregation (insert):

- Supports all aggregates, including exact distinct, percentiles
- Harder to handle late data

Incremental aggregation (upsert):

- Supports late data
- Cannot handle all aggregates (though can approximate using HLL, TopN)

Append-only aggregation

Aggregate events for a particular time period and append them to the rollup table, once all the data for the period is available.

```
INSERT INTO rollup
SELECT period, dimensions, aggregates
FROM events
WHERE event_time::date = '2018-09-04'
GROUP BY period, dimensions;
```

- Should keep track of which periods have been aggregated.
- If data comes late and has been rolled-up, you have to ignore it.

Incremental Aggregation

Aggregate new events and upsert into rollup table.

```
   INSERT INTO rollup
SELECT period, dimensions, aggregates
FROM events
WHERE event_id BETWEEN s AND e
GROUP BY period, dimensions
ON CONFLICT (dimensions, period) DO UPDATE
SET aggregates = aggregates + EXCLUDED.aggregates;
```

* Need to be able to incrementally build aggregates.
* Need to keep track of which events have been aggregated.

Keeping track of aggregated events

- Marking events as aggregated.
  - Causes write amplification and bloat.
- Using a staging table
  - Changes to ingestion pipeline, and higher overhead.
- Tracking sequence number
  - Recommended approach

Track sequence number

* Each event has a monotonically increasing sequence number i.
* Store sequence number S up to which all events were aggregated.

To aggregate:

- Draw a number from the sequence (E)
- Make sure there are no more in-flight transactions that are using sequence numbers <= E (briefly *block writes*)
- Incrementally aggregate all events with sequence numbers S < i <= E
- Set S = E

Function to do transactional rollup

```
CREATE FUNCTION do_aggregation()
RETURNS void LANGUAGE plpgsql AS $function$
DECLARE
       s bigint; e bigint;
BEGIN
   -- Get and update the rollup window
       SELECT * FROM safe_rollup_window('rollup') INTO s, e;

       INSERT INTO rollup SELECT period, dimensions, aggregates
       FROM events WHERE event_id BETWEEN s AND e
       GROUP BY period, dimensions
       ON CONFLICT (dimensions) DO UPDATE
       SET aggregates = aggregates + EXCLUDED.aggregates;
END; $function$;
```

Advanced aggregations – HLL and TopN

Some metrics can’t be rolled up easily

- Number of distinct IP addresses to visit your website.
- Top 10 IP addresses who have visited your website most.

- Problems
  - These metrics are slow to compute
  - Can’t be used incrementally
  - You cannot combine 1min rollups to produce 10min rollups

Solution: Use Approximations

- Data structures such as HLL and TopN
- Key properties:
  - Requires very little memory and storage
  - Merging them is commutative i.e. you can combine 5 min windows to 10 min ones
  - Use them with incremental aggregates
  - Doesn’t require centralized calculation in distributed systems

HLL

HyperLogLog starts by taking a hash of items counted:

```
hll_hash_text('54.33.98.12')
```

The hash function will produce a uniformly distributed bit string.
Unlikely patterns occurring indicates high cardinality.

Hash value with n 0-bits is observed → roughly 2^n distinct items

HyperLogLog divides values into m streams and averages the results.

HyperLogLog

- Approximation algorithm to count number of unique elements in a list.
- Stores a data structure whose cardinality indicates the number of distinct elements in the list.

Process

- First, hash the elements – hll_hash(number)
- Add other elements to the list by using hll_add(hashvalue)
- Combine a large number of elements by using hll_add_agg(column)
- Distinct count estimate is as simple as hll_cardinality(hll)

- To combine hlls together just do hll_union_agg(hll)

Incremental Aggregation using HLL

Use `hll_add_agg` and `hll_union` to do incremental rollups.

```
CREATE TABLE hourly_rollup (
  customer_id bigint not null,
  period timestamptz not null,
  unique_ips hll not null,
  PRIMARY KEY (customer_id, period)
);

INSERT INTO hourly_rollup
SELECT customer_id, date_trunc('hour', created_at), hll_add_agg(ip)
FROM page_views
WHERE event_id BETWEEN start_id AND end_id
GROUP BY 1, 2 ON CONFLICT (customer_id, period)
DO UPDATE SET unique_ips = hll_union(unique_ips, EXCLUDED.unique_ips);
```

Dashboard queries with HLL

Use `hll_union_agg` to merge HLL objects and `hll_cardinality` to extract distinct count.

```
-- HLL
SELECT period::date, hll_cardinality(hll_union_agg(unique_ips)) AS uniques
FROM hourly_rollup
WHERE customer_id = 1283 AND period >= now() - interval '1 week'
GROUP BY 1 ORDER BY 1;
```

```
   period    | uniques
-------------+---------
 2018-08-29  |   14712
 2018-08-30  |   33280
 …
(7 rows)
```

TopN

TopN keeps track of a set of counters (e.g. 1000) in JSONB with the explicit goal of determining the top N (e.g. 10) most heavy hitters.

```
{
  "184.31.49.1" : 1124712,
  "22.203.1.77" : 28371,
  "54.68.19.33" : 62183,
  …
}
```

Merging TopN objects

Like HLL, TopN objects can be merged over time periods, dimensions.

```
topn_union(tn1,tn2)
```

```
{                                                                    {
  "184.31.49.1" : 1124712,                                             "184.31.49.1" : 3407,
  "22.203.1.77" : 28371,                          +                    "22.203.1.77" : 22,
  "54.68.19.33" : 62183,                                               "54.68.19.33" : 1,
  …                                                                                  …
}                                                                    }
```

Incremental aggregation using TopN

Use `topn_add_agg` and `topn_union` to do incremental rollups.

```
CREATE TABLE heavy_hitters_hour (
  customer_id bigint not null,
  period timestamptz not null,
  top_ips jsonb not null,
  PRIMARY KEY (customer_id, period)
);

INSERT INTO heavy_hitters_hours
SELECT customer_id, date_trunc('hour', created_at), topn_add_agg(ip)
FROM page_views
WHERE event_id BETWEEN start_id AND end_id
GROUP BY 1, 2 ON CONFLICT (customer_id, period)
DO UPDATE SET top_ips = topn_union(top_ips, EXCLUDED.top_ips);
```

Dashboard queries with TopN

Use `topn_union_agg` to merge TopN objects, `topn` to extract top N counts.

```
-- Topn
SELECT (topn(topn_union_agg(top_ips), 10)).*
FROM heavy_hitters_hour
WHERE customer_id = 1283 AND period >= now() - interval '1 day';
```

```
    item      | frequency
--------------+-----------
 184.31.49.1  |   1124712
 54.68.19.33  |     62183
 …
(10 rows)
```

Cheap index look-up with aggregation across 24 rows.

Recipe for real-time analytics

- Ingest large volumes of data to a raw table
- Periodically aggregate events into a rollup table
- Have application query the rollup table
- Automate all of this with pg_cron

Automate jobs with pg_cron

- PostgreSQL extension which allows you to run cron within the database
- Makes it easy to schedule jobs without requiring external tools

Example: Delete old data at midnight using pg_cron:

```
 SELECT cron.schedule('0 0 * * *', $$
    DELETE FROM events
    WHERE event_time < date_trunc('day', now() -
interval '1 week')
  $$);
```

Periodic aggregation using pg_cron

Run aggregation every 5 minutes:

```
SELECT cron.schedule('*/5 * * * *', $$
    SELECT do_aggregation()
$$);
```

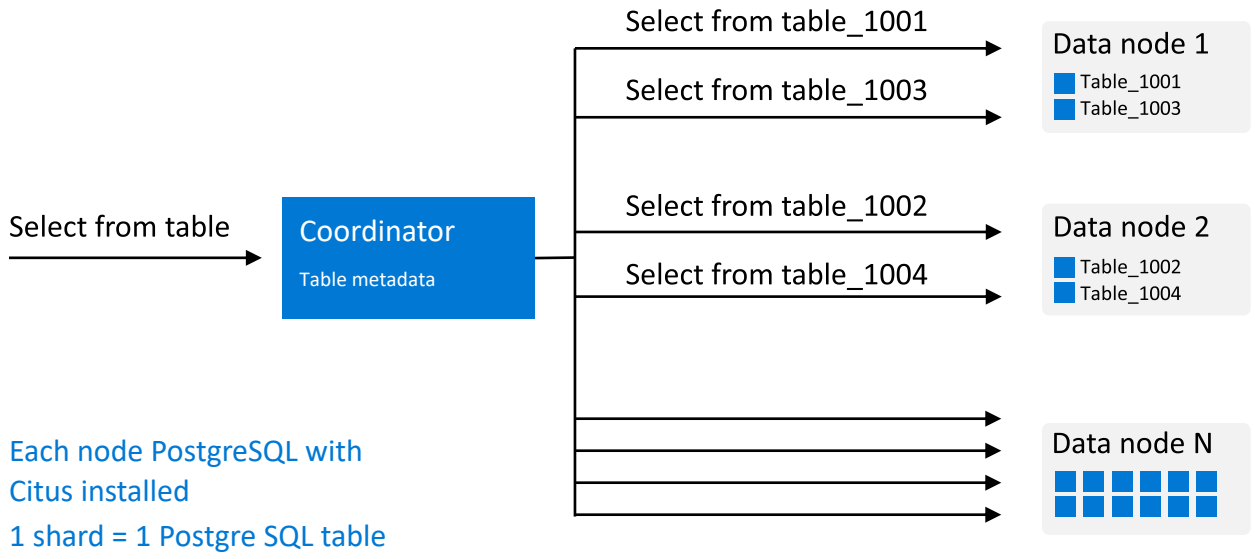
Scale out using Hyperscale (Citus)

Architecture

Shard your PostgreSQL database across multiple nodes to give your application more memory, compute, and disk storage

Easily add worker nodes to achieve horizontal scale, while being able to deliver parallelism even within each node

Scale out to 100s of nodes



Scaling with Hyperscale (Citus)

- All of this was for one node
- With Hyperscale (Citus), you can scale this entire pipeline to 10s-100s of nodes

- To distribute, run:
  - `SELECT create_distributed_table('events', 'customer_id');`

Multi-tenancy and colocation

Tenant ID provides a natural sharding dimension for many applications.

Citus automatically co-locates event and rollup data for the same

```
SELECT create_distributed_table('events', 'tenant_id');
SELECT create_distributed_table('rollup', 'tenant_id');
```

Aggregations can be done locally, without network traffic:

```
INSERT INTO rollup SELECT tenant_id, … FROM events …
```

Dashboard queries are always for a particular tenant:

```
SELECT … FROM rollup WHERE tenant_id = 1238 …
```

Or are parallelized when you want to compare tenants:

```
SELECT tenant_id,… FROM rollup GROUP BY tenant_id …
```

Benefits of Hyperscale (Citus)

Horizontally scale out a real-time analytics pipeline:

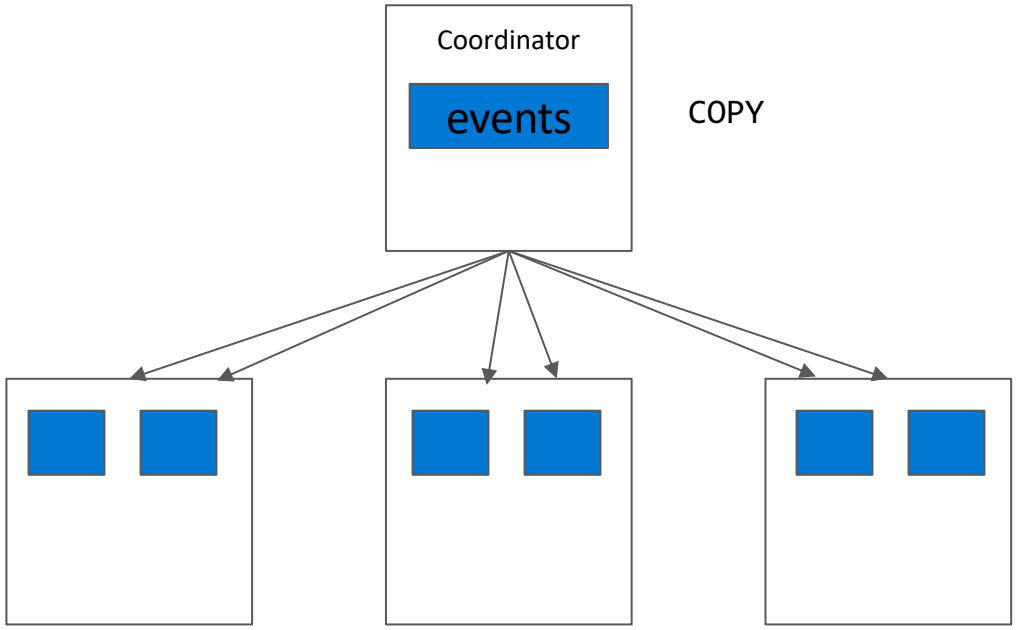
- High throughput COPY
- Parallel INSERT..SELECT, DELETE, SELECT, autovacuum
- Low-latency queries on rollup for a particular customer

In general:

- Can always have enough capacity (memory, storage, CPU) to meet performance goals
- Smaller tables, indexes through sharding (+ partitioning)

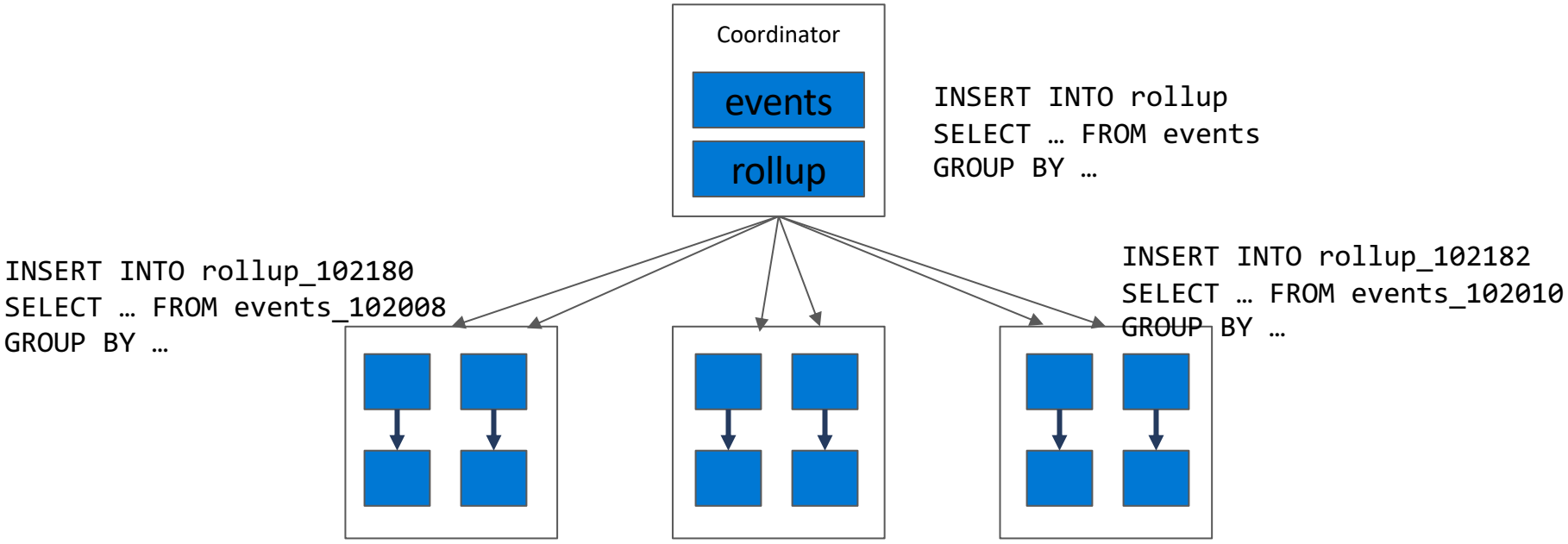
Data loading

- COPY asynchronously scatters rows to different shards



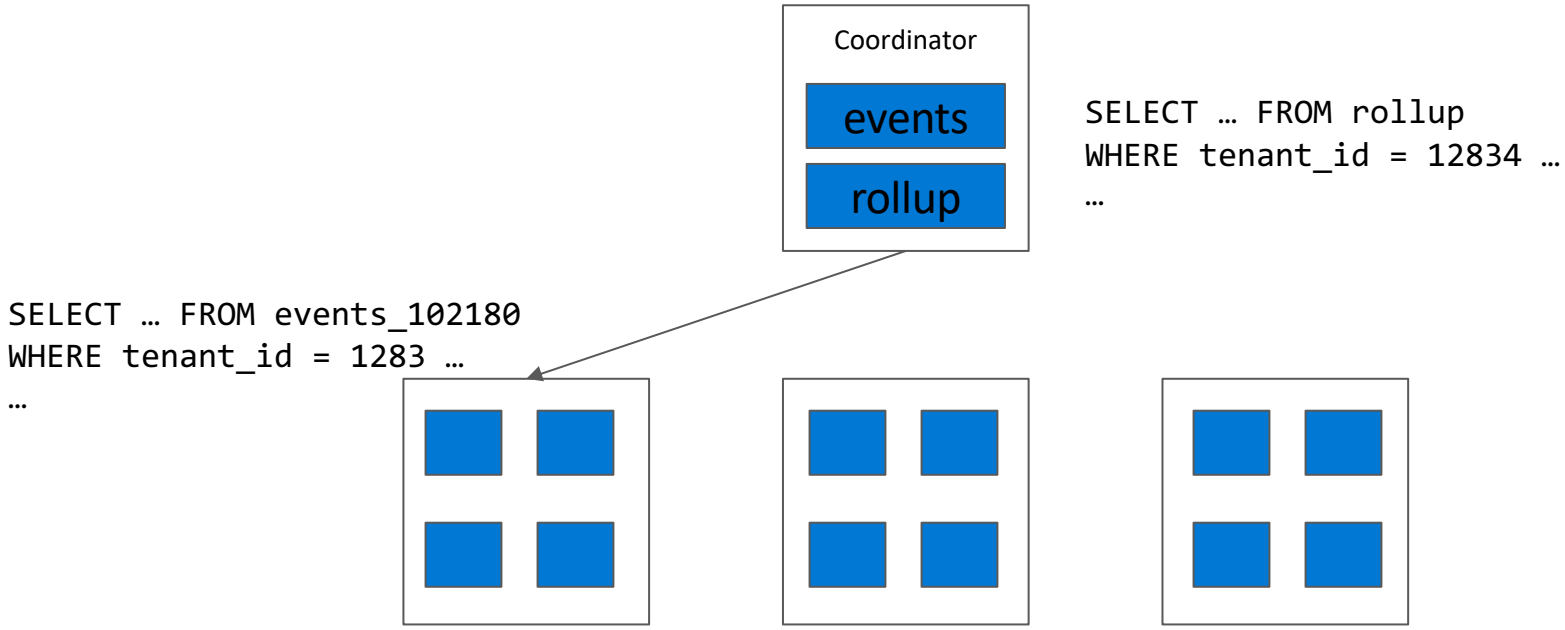
Aggregation and rollups

INSERT … SELECT can be parallelised across shards



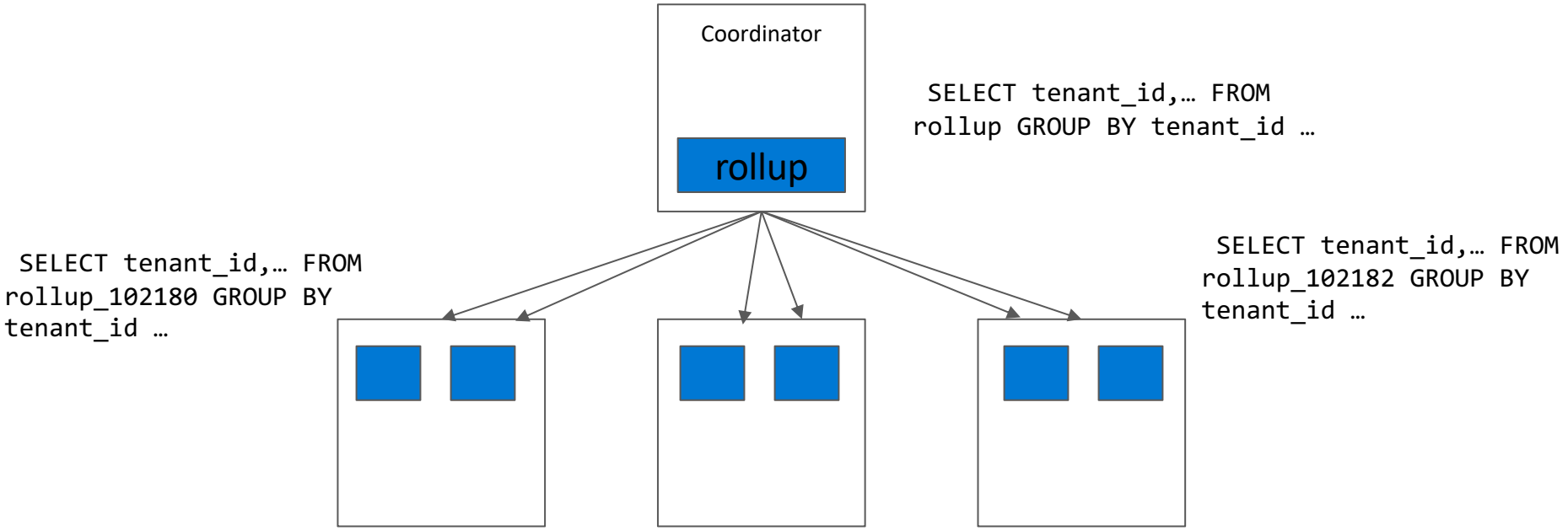
Querying rollups

SELECT on rollup for a particular customer (from the dashboard) can be routed to the appropriate shard.



Querying across tenants

SELECT queries across tenants can be parallelized across shards



Summary

To build a real-time application which

- Generates large amounts of data
- Has sub-second response times
- Supports large number of concurrent users
- Reflects new data within minutes
- Supports advanced analytics

You should use:

- **Azure Database for PostgreSQL** as your database engine
- **COPY** to load raw data into a table
- **BRIN** index to find new events during aggregation
- Ordered deletion or **partitioning** with pg_partman to expire old data
- **Rollup tables** built from raw event data
- **Incremental aggregation** if you can have late data
- **HLL** to incrementally approximate distinct count
- **TopN** to incrementally approximate heavy hitters
- **Hyperscale (Citus)** to scale out

Hands on Lab

Scenario: You = Cloud services provider helping businesses monitor their HTTP traffic

- Every time one of your clients receives an HTTP request, your service receives a log record.
- You want to ingest all records & create an HTTP operational analytics dashboard to give your clients insight, such as the number of HTTP errors their sites served.
- Fast queries (low latency) as there is a dashboard and real-time (at least minutely)
- Queries could be for analyzing multiple sites at once (OR) a single site at once.

http://tinyurl.com/yxreau7d

*** Write down session id ***

Instructions link

(if browser issues – disable popup blocker, try incognito)

Setup (page 1 to page 4)

- **Read Scenario (IMPORTANT)**
- Login to Azure Portal using your credentials
- Setup Cloud Shell
- Checkout Hyperscale(Citus) cluster
- Setup Firewall
- Test connecting to Hyperscale(Citus) cluster

Please pause at page 4

Design and Implementation (page 5 to page 9)

- Data model – table design and sharding
- Querying raw data
- Introducing Rollups
- Data Expiration
- Implementing HLL for Approximate Distinct Counts in Rollups
- Unstructured data with jsonb in Rollups

Thank You
Q&A

Sai Krishna Srirampur
Colton Shepard