Storage and Schema Optimizations

02

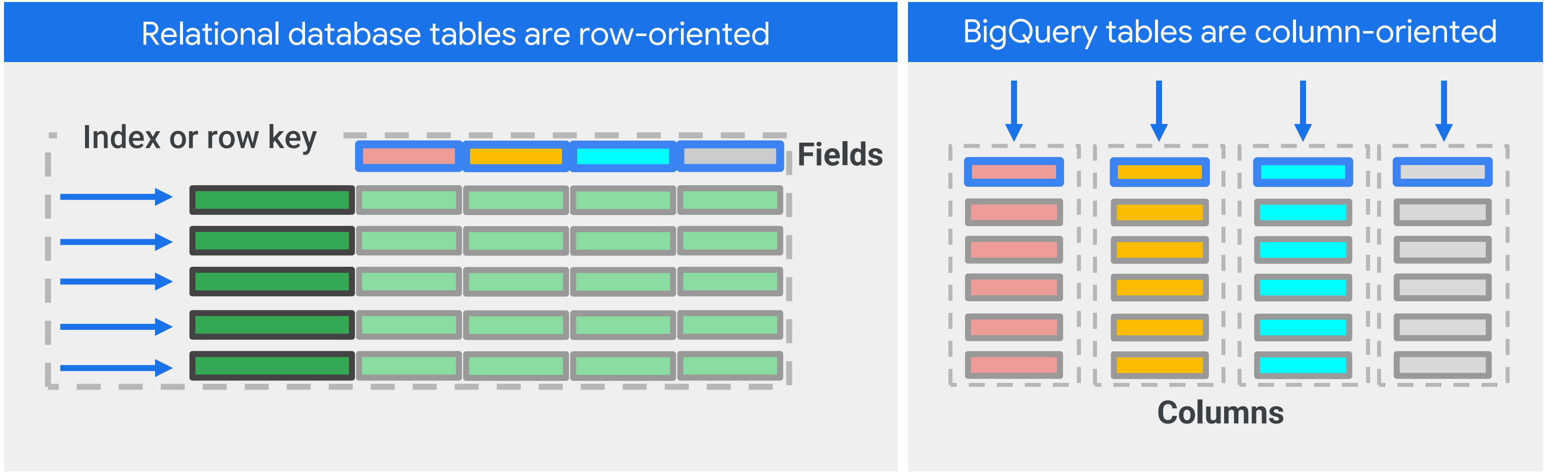
Topics

Google Cloud

01

BigQuery Storage

Recap: BigQuery splits tables into columnar files



Recap: Columnar files stored in Colossus; metadata stored in Cloud Spanner

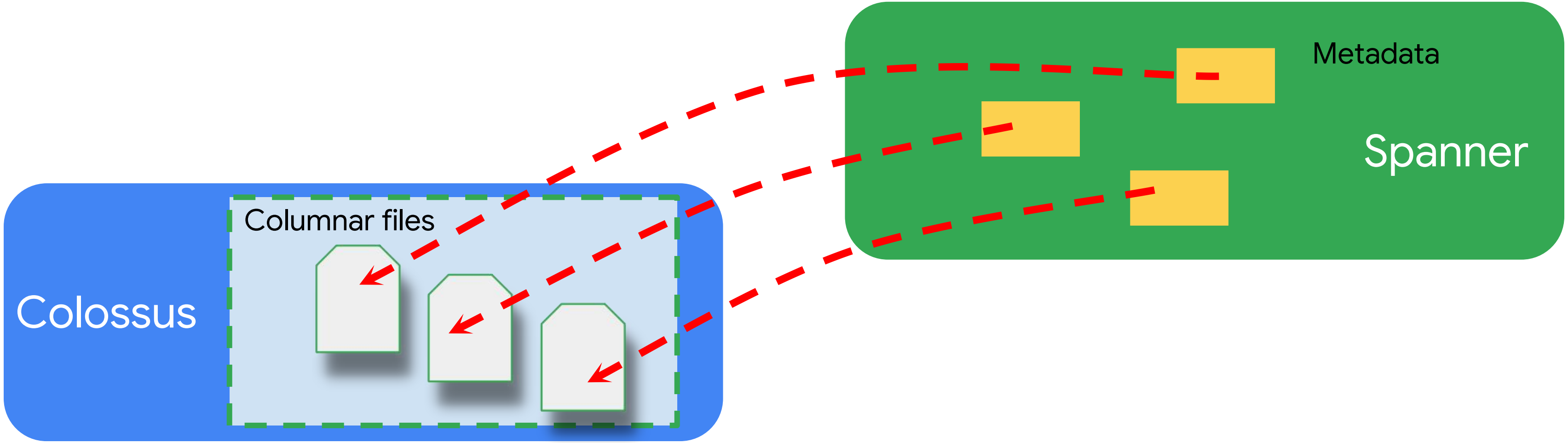
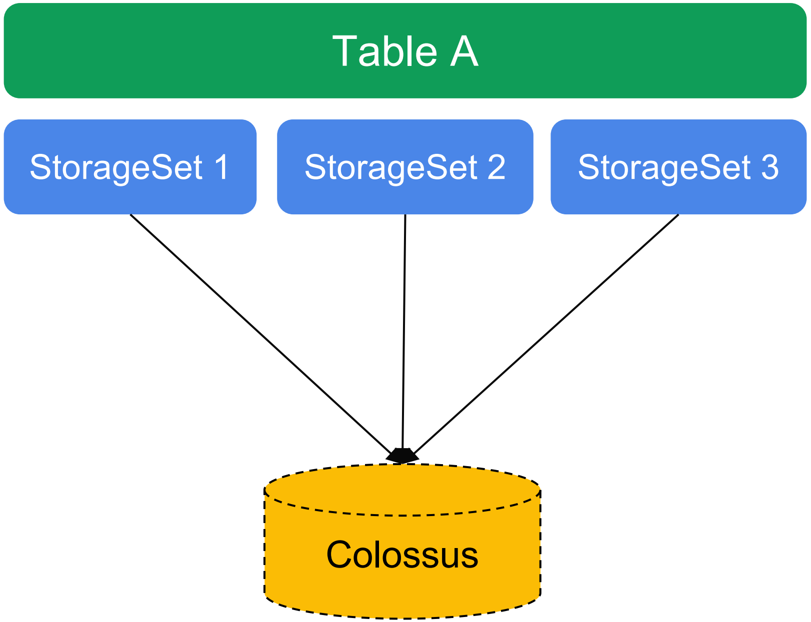


Table metadata and StorageSets



StorageSet (data commit) metadata contains:

- Colossus path information
- Number of inputs (columnar files)
  - Not one file per column; files are sets of rows.
- Partition key
  - Implication: one StorageSet per partition (ultimately)

Table metadata and StorageSets (continued)

StorageSet (data commit) metadata contains:

- State
  - PENDING (Preparing to commit)
  - COMMITTED (Live)
  - GARBAGE (Deleted or superseded by newer data)
- Data stats
  - Column info, sizes, data constraints/ranges

02

Partitioning and Clustering

Partitioned tables

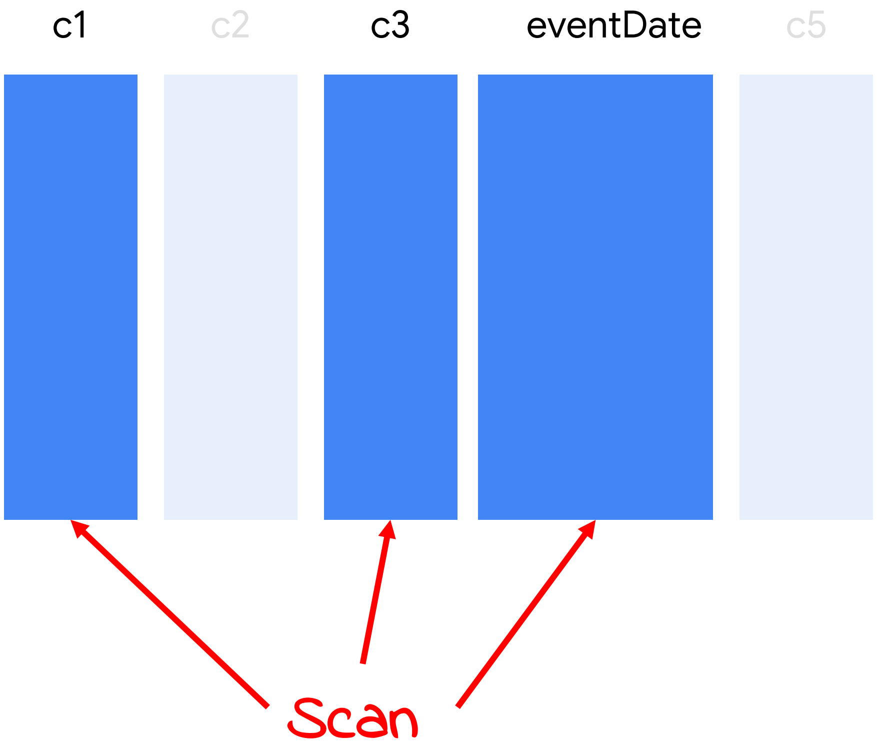
Fetching data from a non-partitioned table

SELECT **c1, c3**

FROM t1

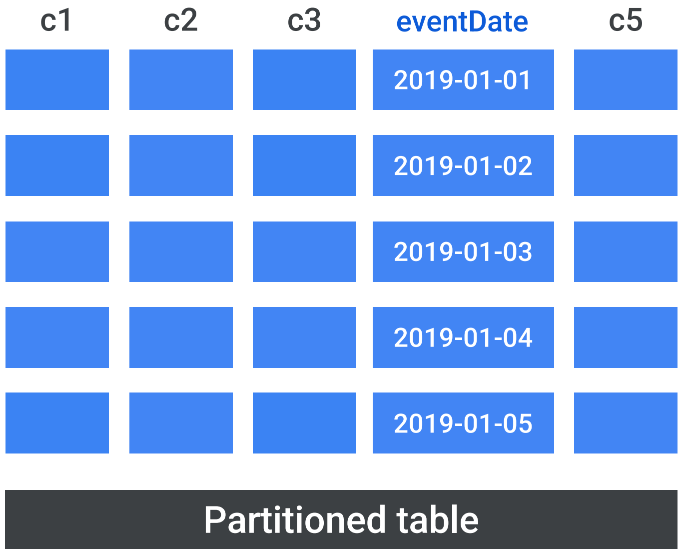
WHERE

eventDate BETWEEN "2019-01-03" AND "2019-01-04"



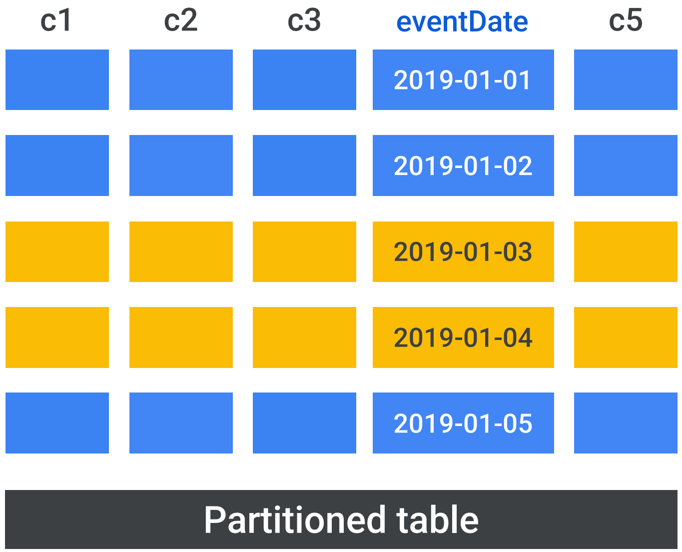
What if the table was partitioned?

Table is partitioned by the **eventDate** column.



Reduce cost and amount of data read by partitioning your tables

```
SELECT c1, c3
FROM t1
WHERE eventDate BETWEEN
"2019-01-03" AND "2019-01-04"
```



Partitioning

- Data is automatically partitioned at write time.
- Each partition behaves like its own table.
- Metadata is maintained for each partition.
- Provides strict guarantees for bytes scanned and billed.
    - Query cost is known upfront.
- Partitioning is available at table creation time only.

```
CREATE TABLE T1 (eventDate TIMESTAMP,
                 userId INTEGER,
                 itemId STRING,
                 ...,
                 ... )
PARTITION BY DATE(eventDate)
```

A common challenge with 1M+ record tables is querying the entire table for just one week's metrics

All ecommerce site visits

```
SELECT
  COUNT(transactionId) AS total_transactions,
  date
FROM
  `data-to-insights.ecommerce.all_sessions`
WHERE
  transactionId IS NOT NULL
  AND PARSE_DATE("%Y%m%d", date) >= '2018-01-01'
GROUP BY date
ORDER BY date DESC
```

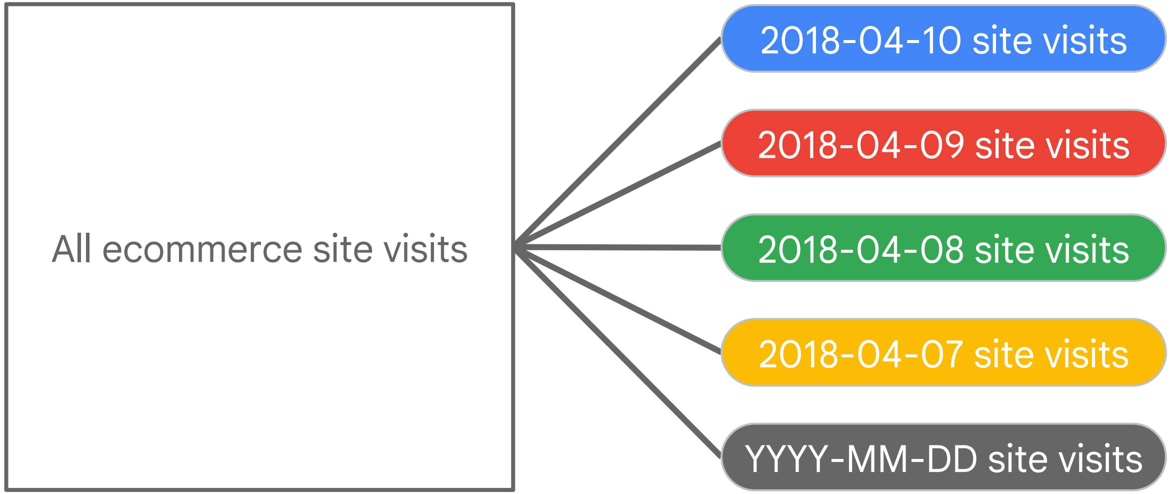
To satisfy the WHERE condition, our query must look at **every** date value to see whether it's after ‘2018-01-01’

All ecommerce site visits

```
SELECT
  COUNT(transactionId) AS total_transactions,
  date
FROM
  `data-to-insights.ecommerce.all_sessions`
WHERE
  transactionId IS NOT NULL
  AND PARSE_DATE("%Y%m%d", date) >= '2018-01-01'
GROUP BY date
ORDER BY date DESC
```

This query will process 205.9 MB when run.

A single table can be divided into logical partitions for performance



A single table can be divided into logical partitions for performance

```
CREATE OR REPLACE TABLE ecommerce.partitions
 PARTITION BY date_formatted
 OPTIONS(
   description="a table partitioned by date"
 ) AS

SELECT
  COUNT(transactionId) AS total_transactions,
  PARSE_DATE("%Y%m%d", date) AS date_formatted
FROM
  `data-to-insights.ecommerce.all_sessions`
WHERE
  transactionId IS NOT NULL
GROUP BY date
```

Now the exact same query will first reference the partition list before processing any data!

- 2018-04-10 site visits
- 2018-04-09 site visits
- 2018-04-08 site visits
- 2018-04-07 site visits
- YYYY-MM-DD site visits

```
SELECT
  total_transactions,
  date_formatted
FROM
  `data-to-insights.ecommerce.partitions`
WHERE date_formatted >= '2018-01-01'
ORDER BY date_formatted DESC
```

This query will process 0 B when run.

Why 0?

Our query knew there weren't any transactions after 2017-08-01 in our dataset by looking at our existing partitions

2018-04-10 site visits

2018-04-09 site visits

2018-04-08 site visits

2018-04-07 site visits

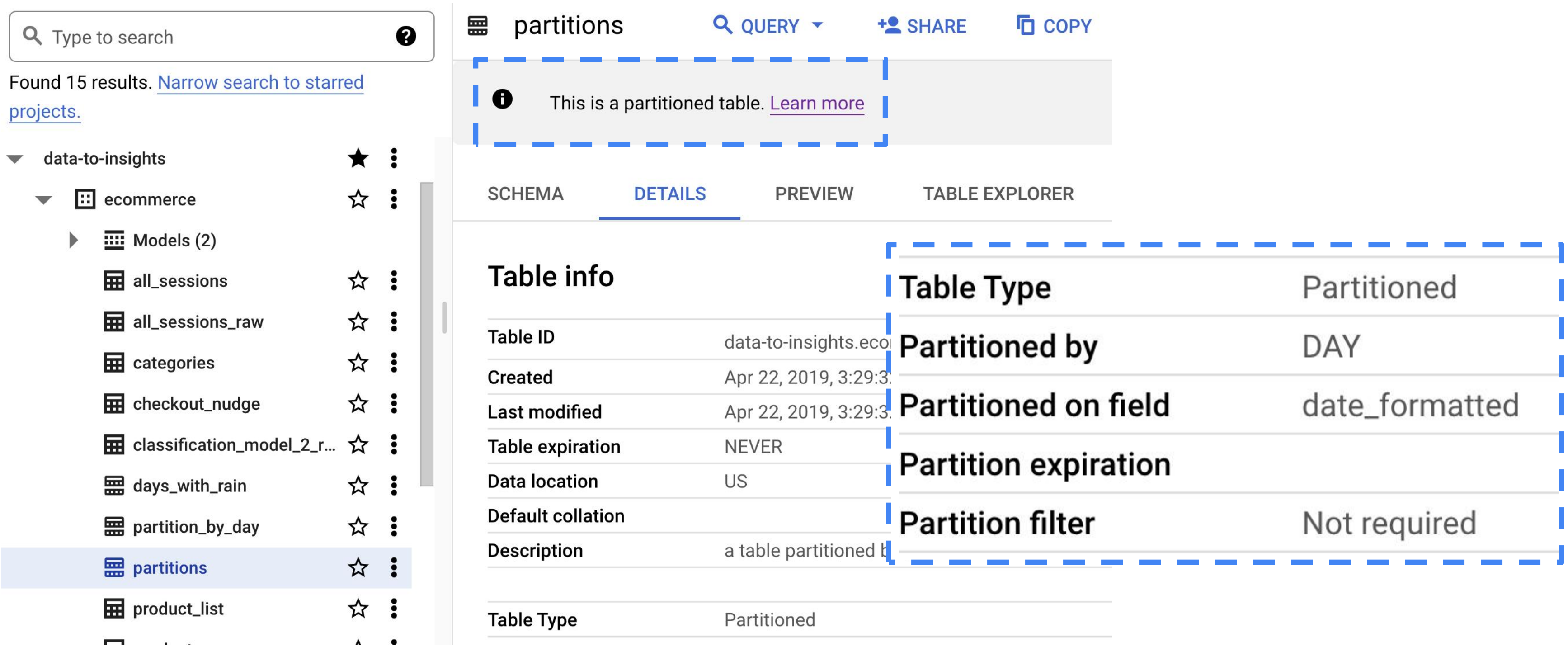
YYYY-MM-DD site visits

```
SELECT
  total_transactions,
  date_formatted
FROM
  `data-to-insights.ecommerce.partitions`
ORDER BY
  date_formatted DESC
```

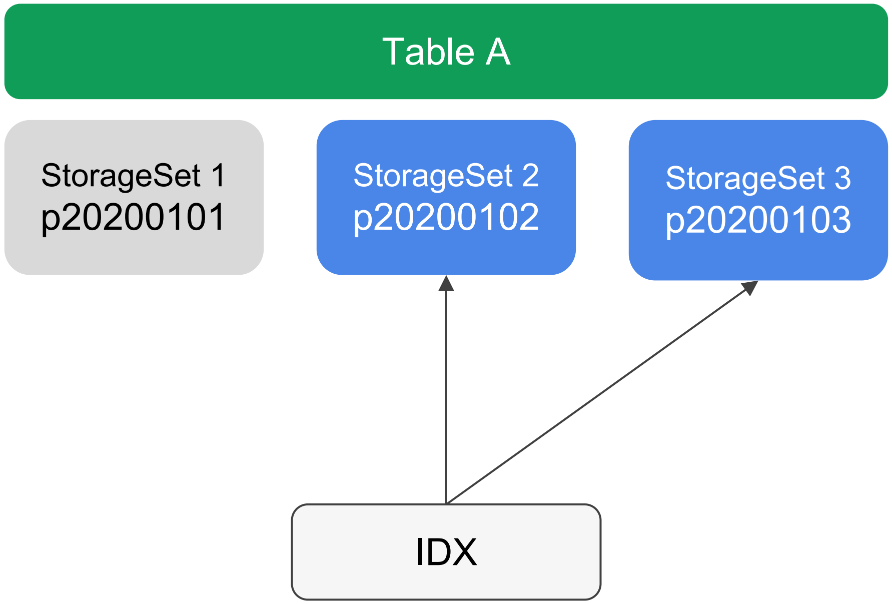
Row	total_transactions	date_formatted
1	279	2017-08-01
2	321	2017-07-31
3	139	2017-07-30
4	111	2017-07-29
5	249	2017-07-28
6	230	2017-07-27

The latest partition of data is from ***2017***-08-01, so no 2018 transactions would ever show up.

Partitioned tables in the BigQuery UI

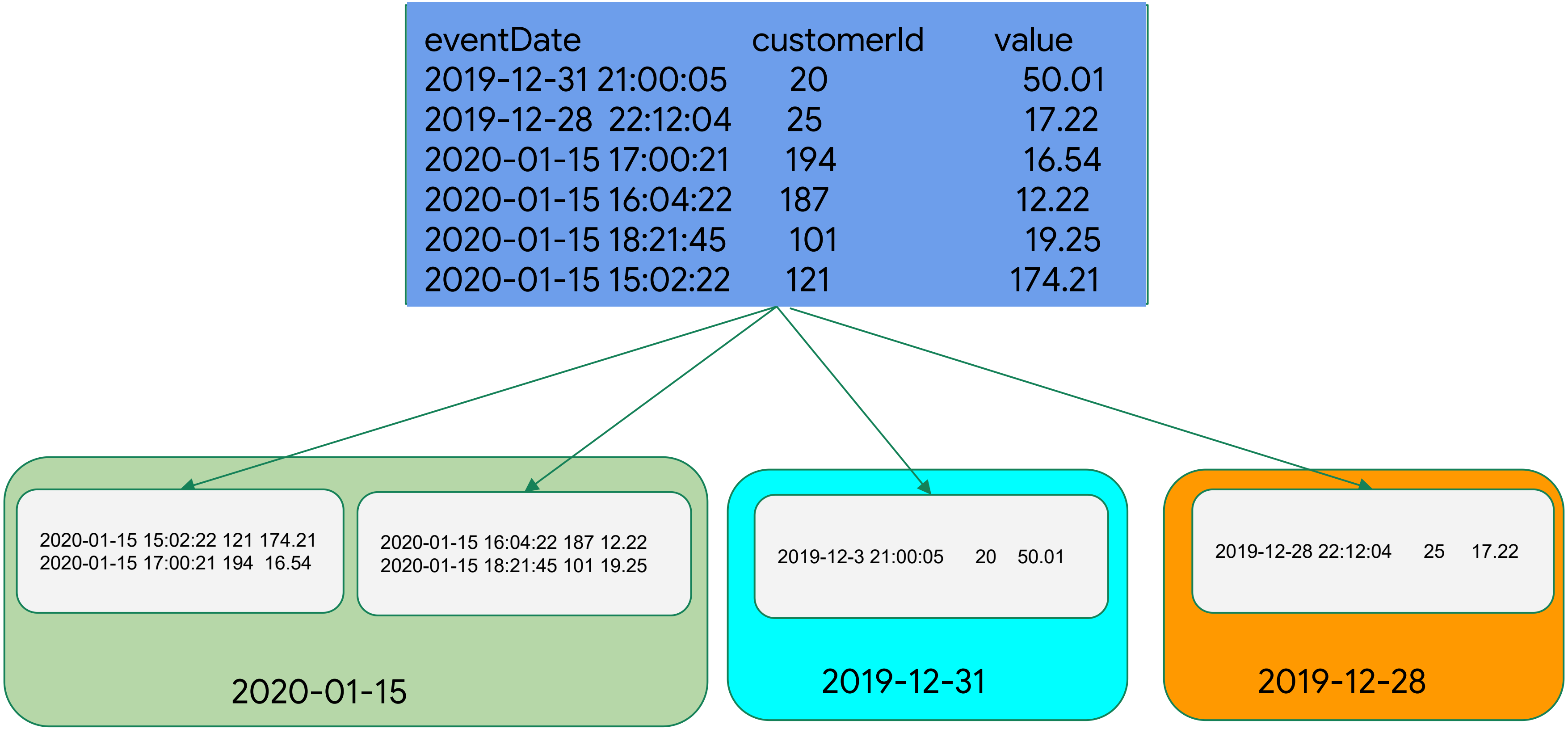


Partitioning and StorageSets




```
SELECT … WHERE eventDate >= "20200102"
```

Writing to a partitioned table



BigQuery supports three ways of partitioning tables

Ingestion time

```
bq query --destination_table mydataset.mytable
--time_partitioning_type=DAY
...
```

Any column that is of type DATETIME, DATE, or TIMESTAMP

```
bq mk --table --schema a:STRING,tm:TIMESTAMP --
time_partitioning_field tm
```

Integer-type column

```
bq mk --table --schema "customer_id:integer,value:integer"
--range_partitioning=customer_id,0,100,10
my_dataset.my_table
```

Partitioning types

- **Ingestion date/time partitioning**
  - Based on date/time that data is loaded
  - Filter using pseudo-columns: **_PARTITIONDATE, _PARTITIONTIME**
    - SELECT col FROM d.t WHERE **_PARTITIONDATE** > “2018-05-01”
    - Note: Streaming buffer has NULL values in pseudo-columns.

**Note: 4000-partition limit per table.
Can be increased to 10,000 on request.**

- **Column partitioning**
  - Supported column types
    - TIMESTAMP, DATE, DATETIME
    - INT64
  - Filter using column name
    - SELECT COUNT(*) FROM d.t WHERE datecol > “2018-05-01”

For tables partitioned by time-unit or ingestion time

Daily partitioning (default)

When your data is spread out over a wide range of dates, or if data is continuously added over time.

Hourly partitioning

If your tables have a high volume of data that spans a short date range (typically less than 6 months of timestamp values).

Monthly or yearly partitioning

If your tables have a relatively small amount of data for each day, but span a wide date range.

If your workflow requires frequently updating or adding rows that span a wide date range.

Demo

Querying a partitioned and clustered table

Spot the difference!

```
SELECT wiki, SUM(views) views
FROM `fh-bigquery.wikipedia_v2.pageviews_2017`
WHERE datehour >= '2017-01-01'
AND wiki IN('en', 'en.m')
AND title = 'Kubernetes'
GROUP BY wiki ORDER BY wiki
```

v2 is partitioned

Query results SAVE AS EXPLORE IN DATA STUDIO

Query complete (20.706 sec elapsed, 2.2 TB processed)

Job information Results JSON Execution details

Row	wiki	views
1	en	343047
2	en.m	90939

```
SELECT wiki, SUM(views) views
FROM `fh-bigquery.wikipedia_v3.pageviews_2017`
WHERE datehour >= '2017-01-01'
AND wiki IN('en', 'en.m')
AND title = 'Kubernetes'
GROUP BY wiki ORDER BY wiki
```

v3 is partitioned and clustered

Query results SAVE AS EXPLORE IN DATA STUDIO

Query complete (5.413 sec elapsed, 227.76 GB processed)

Job information Results JSON Execution details

Row	wiki	views
1	en	343047
2	en.m	90939

Auto-pruning with partitioning and clustering

Partitioned table

Table info

Table ID	fh-bigquery:wikipedia_v2.pageviews_2017
Table size	2.2 TB
Long-term storage size	2.2 TB
Number of rows	54,489,325,868
Created	Feb 27, 2018, 1:54:41 AM
Table expiration	Never
Last modified	Feb 27, 2018, 4:47:50 AM
Data location	US
Table type	Partitioned
Partitioned by	Day
Partitioned on field	datehour
Partition filter	Required

Partitioned and clustered table

Table info

Table ID	fh-bigquery:wikipedia_v3.pageviews_2017
Table size	2.2 TB
Long-term storage size	2.2 TB
Number of rows	54,489,325,868
Created	Aug 1, 2018, 1:24:57 AM
Table expiration	Never
Last modified	Aug 2, 2018, 8:50:32 PM
Data location	US
Table type	Partitioned
Partitioned by	Day
Partitioned on field	datehour
Partition filter	Required
Clustered by	wiki, title

Auto-pruning with partitioning and clustering

Partitioned table by datehour

```
SELECT *
FROM `fh-bigquery.wikipedia_v2.pageviews_2017`
WHERE DATE(datehour) BETWEEN '2017-06-01'
AND '2017-06-30'
LIMIT 1
```

1.7 sec elapsed, 180 GB processed

Partitioned table by datehour
Clustered table by wiki, title

```
SELECT *
FROM `fh-bigquery.wikipedia_v3.pageviews_2017`
WHERE DATE(datehour) BETWEEN '2017-06-01'
AND '2017-06-30'
LIMIT 1
```

1.8 sec elapsed, 16 MB processed

Clustered tables

What if your queries commonly include more than one column?

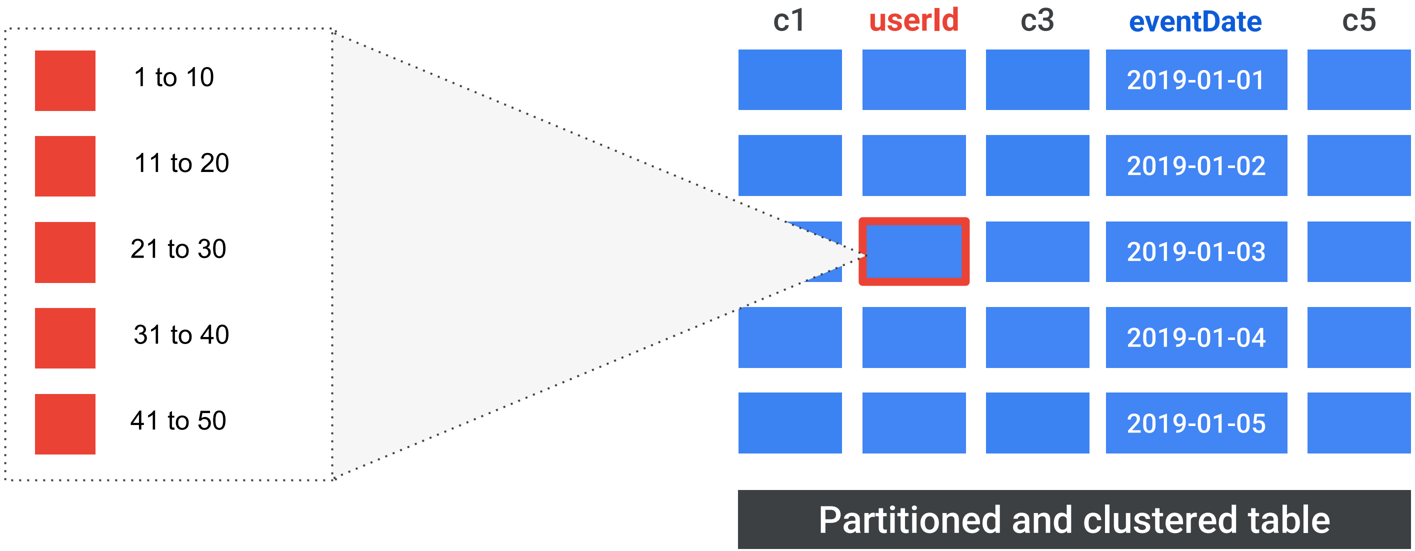
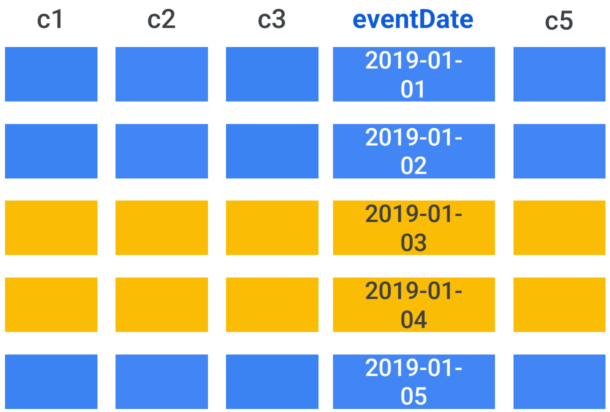


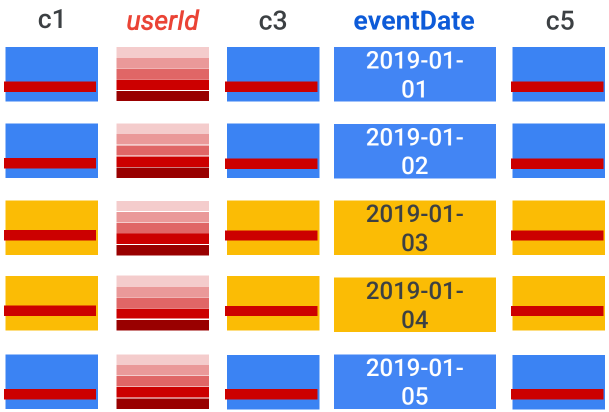
Table is partitioned by the **eventDate** and clustered by **userId**.

BigQuery automatically sorts the data based on values in the clustering columns



SELECT c1, c3, c5 FROM t1
WHERE eventDate BETWEEN “2019-01-03”
AND “2019-01-04”

Partitioned table



SELECT c1, c3, c5 FROM t1 WHERE *userId*
BETWEEN 52 and 63 AND eventDate
BETWEEN “2019-01-03” AND “2019-01-04”

Partitioned and clustered table

Partitioning and clustering

- User-provided directives influence the layout of data in a table.

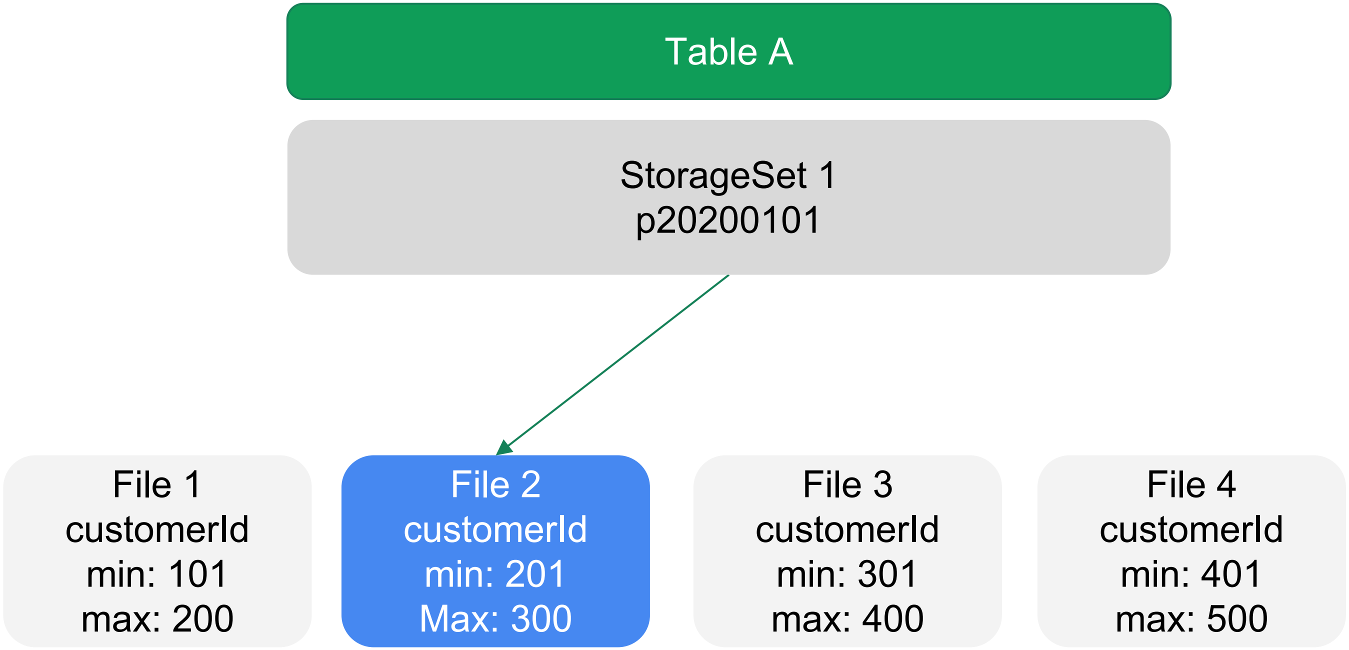
```
CREATE TABLE T1 (eventDate TIMESTAMP,
                 userId INTEGER,
                 itemId STRING,
                 ...,
                 ...)
PARTITION BY DATE(eventDate)
CLUSTER BY userId, itemId;
```

- Partitioning is available at table creation time only.
- Clustering is available for already existing tables.

When to use clustering

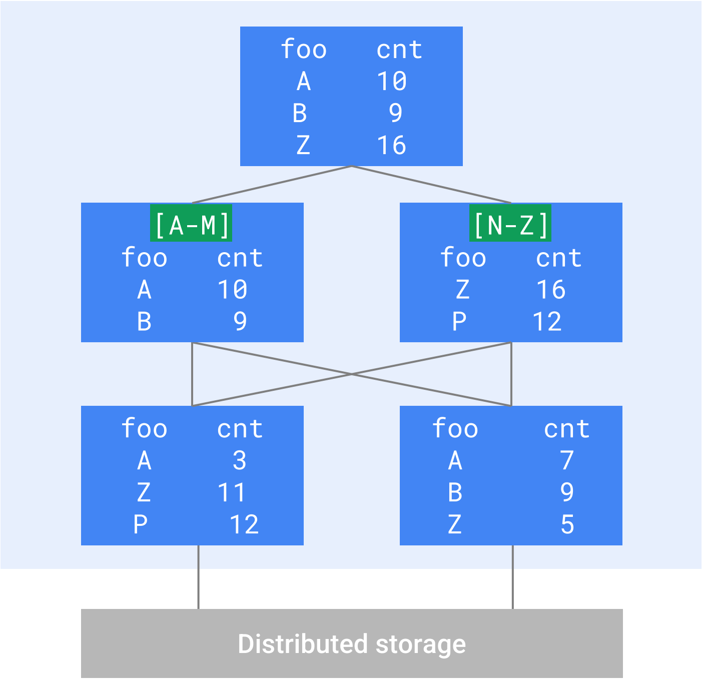
- Your data is already partitioned on a `DATE`, `DATETIME`, `TIMESTAMP` or Integer Range.
- You commonly use filters or aggregation against particular columns in your queries.

Clustering and StorageSets




```
SELECT … WHERE customerId = 275
```

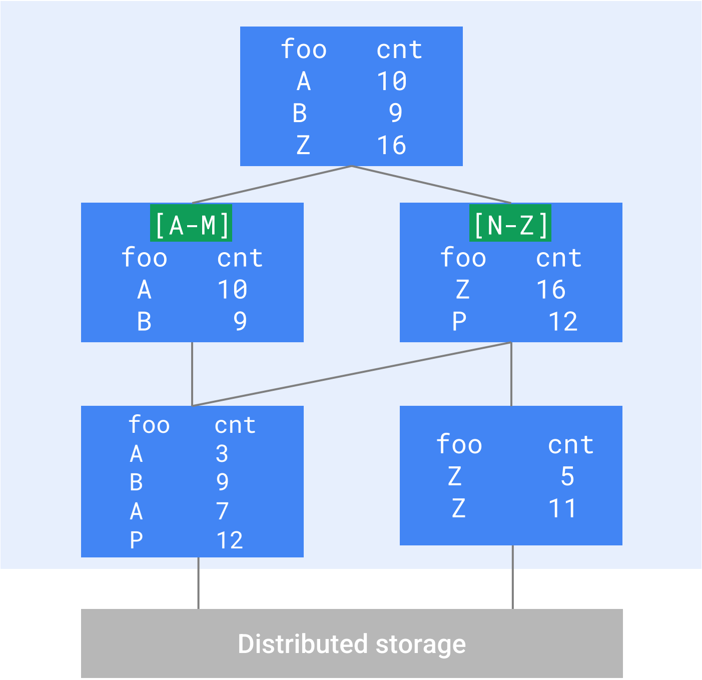
Clustering reduces the amount of data for aggregation




```
SELECT foo, COUNT(*) as cnt
FROM `...`
GROUP BY 1
```

Unclustered data

Clustering reduces the amount of data for aggregation



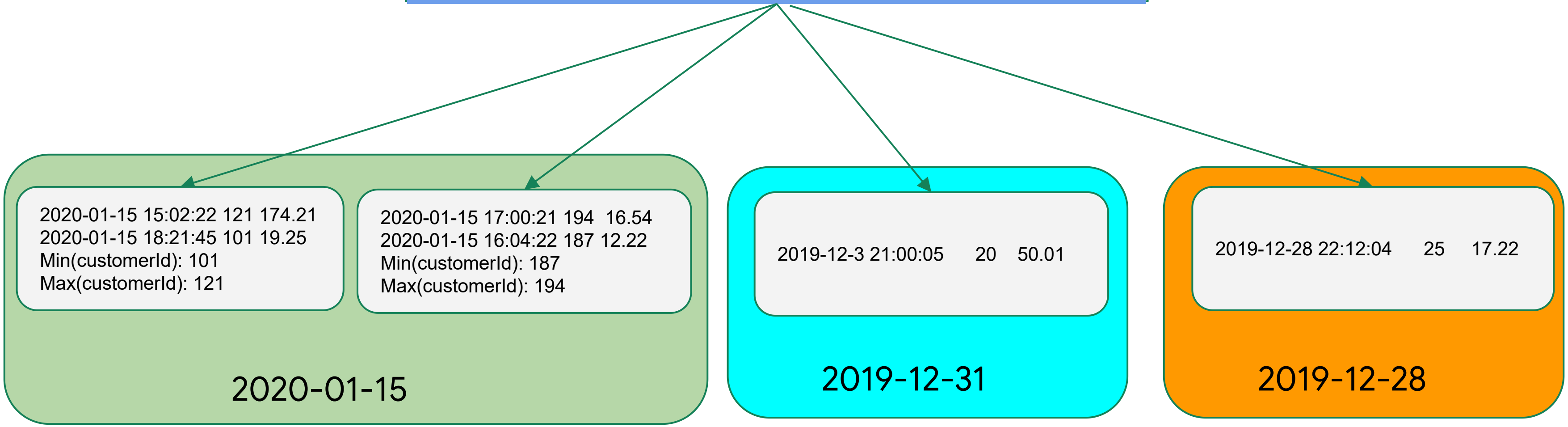

```
SELECT foo, COUNT(*) as cnt
FROM `...`
GROUP BY 1
```

Data is clustered by column foo.

LIMIT enhances performance on the amount of data to read in.

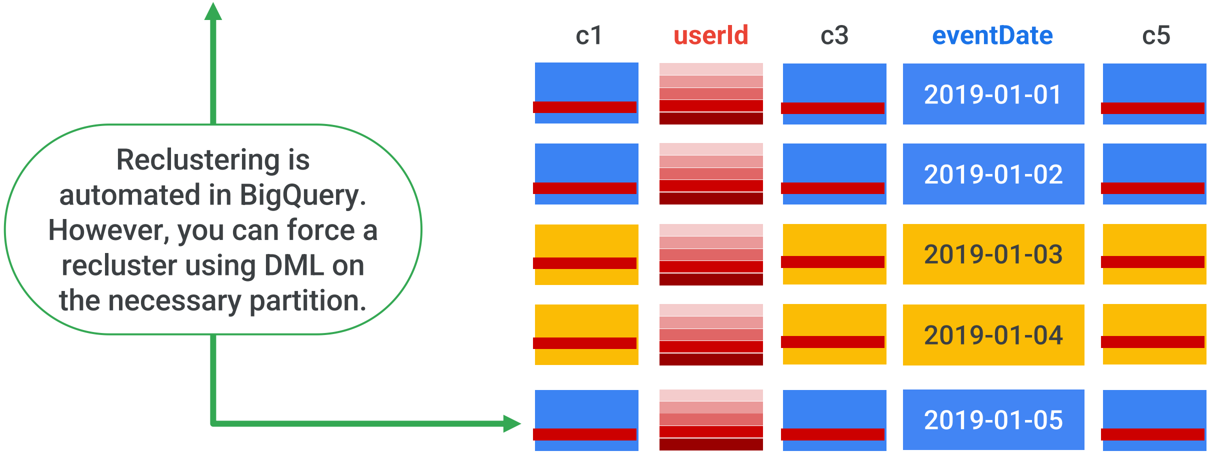
Writing to a partitioned and clustered table

eventDate	customerId	value
2019-12-31 21:00:05	20	50.01
2019-12-28 22:12:04	25	17.22
2020-01-15 17:00:21	194	16.54
2020-01-15 16:04:22	187	12.22
2020-01-15 18:21:45	101	19.25
2020-01-15 15:02:22	121	174.21

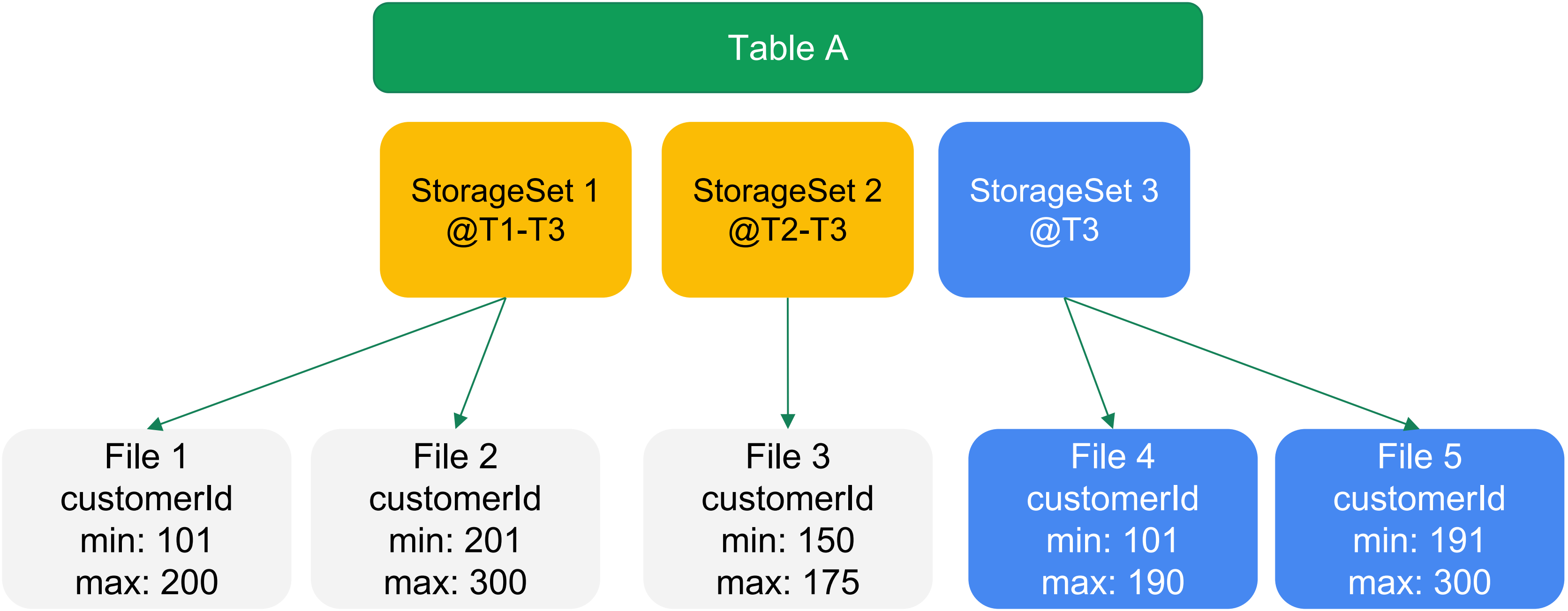


In streaming tables, the sorting fails over time, and so BigQuery has to recluster

```
UPDATE ds.table
SET c1 = 300
WHERE c1 = 300
AND eventDate > TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 DAY)
```



Reclustering



Organize data through managed tables

Partitioning

Filtering storage before query execution begins to reduce costs.

Reduces a full table scan to the partitions specified.

A single column results in lower cardinality (e.g., thousands of partitions).

- Time partitioning (Pseudocolumn)
- Time partitioning (User Date/Time column)
- Integer range partitioning

Clustering

Storage optimization within columnar segments to improve **filtering** and **record colocation**.

Clustering performance and cost savings can't be assessed before query begins.

Prioritized clustering of up to four columns on more diverse types (but no nested columns).

Partitioning and clustering caveat

Clustered table

```
SELECT name, state, ARRAY_AGG(STRUCT(date,temp) ORDER
BY temp DESC LIMIT 5) top_hot, MAX(date) active_until
FROM `fh-bigquery.weather_gsod.all`
WHERE name LIKE 'SAN FRANC%' AND fake_date > '1980-01-
01'
GROUP BY 1,2
ORDER BY active_until DESC
```

14.7 MB processed

Create Partitioned table

```
CREATE TABLE `YOUR-DATASET.gsod_partitioned`
PARTITION BY date_month
CLUSTER BY name
AS
SELECT *, DATE_TRUNC(date, MONTH) date_month
FROM `fh-bigquery.weather_gsod.all`
```

Partitioned and Clustered table

```
SELECT name, state, ARRAY_AGG(STRUCT(date,temp) ORDER
BY temp DESC LIMIT 5) top_hot, MAX(date) active_until
FROM `YOUR-PROJECT-ID.YOUR-DATASET.gsod_partitioned`
WHERE name LIKE 'SAN FRANC%'AND date > '1980-01-01'
GROUP BY 1,2
ORDER BY active_until DESC
```

1.53 GB processed

Clustering without partitions is much more efficient on tables that don't have many GB per partition

Questions?

Google Cloud

Lab (40 min)

Creating Partitioned and Clustered Tables in BigQuery

40:00

Google Cloud

03

Nested and Repeated Fields

Transactional databases often use normal form

Original data

Customer	OrderID	Date	Items	
Doug	1600p	8/20/19		
			Product	Quantity
			Caulk	3 boxes
			Soffit	34 meters
			Sealant	2 liters
Tom	221b	10/29/19		
			Product	Quantity
			Sealant	1 liter
			Soffit	17 meters
			Caulk	4 tubes

Normalized data

Orders

Date	OrderID
8/20/2019	1600p
10/29/2018	221b

Order_Items

OrderID	Product	Quantity
1600p	Caulk	3 boxes
221b	Sealant	1 liter
1600p	Soffit	34 meters
221b	Soffit	17 meters
221b	Caulk	4 tubes
1600p	Sealant	2 liters

Data warehouses often denormalize

Normalized data

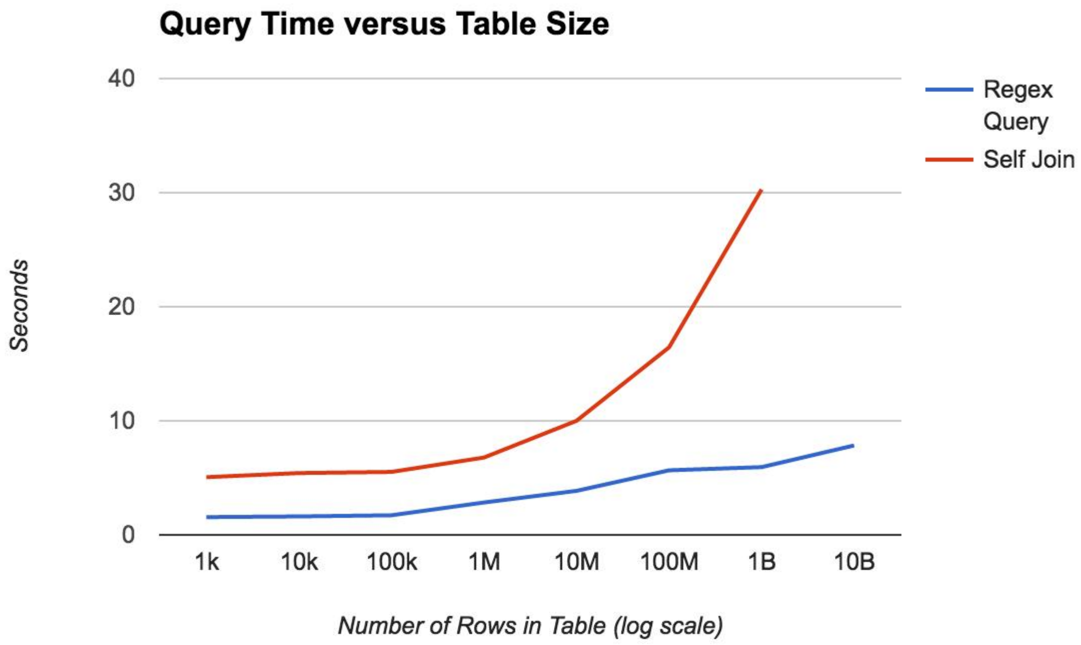
Orders	
Date	**OrderID**
08/20/2019	1600p
10/29/2018	221b

Order_Items		
OrderID	**Product**	**Quantity**
1600p	Caulk	3 boxes
221b	Sealant	1 liter
1600p	Soffit	34 meters
221b	Soffit	17 meters
221b	Caulk	4 tubes
1600p	Sealant	2 liters

Denormalized flattened data

Customer	OrderID	Date	Product	Quantity
Doug	1600p	08/20/2019	Siding	3 boxes
Doug	1600p	08/20/2019	Caulk	12 tubes
Tom	221b	10/29/2019	Soffit	17 meters
Tom	221b	10/29/2019	Sealant	1 liter
Doug	1600p	08/20/2019	Soffit	34 meters
Tom	221b	10/29/2019	Siding	2 boxes
Tom	221b	10/29/2019	Caulk	4 tubes
Doug	1600p	08/20/2019	Sealant	2 liters

Query performance on very large tables can be improved through denormalization



Disadvantages of denormalization

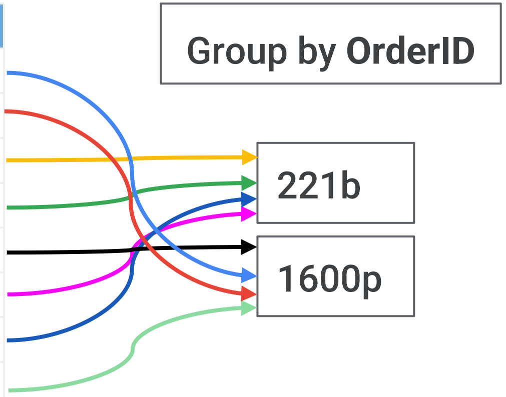
X Denormalized schemas aren't storage-optimal. (However, the low cost of BigQuery storage addresses concerns about storage inefficiency .)

X Maintaining data integrity can require increased machine time, and sometimes human time, for testing and verification.

Grouping on a 1-to-many field in flattened data can cause shuffling of data over the network

Denormalized flattened table

Customer	OrderID	Date	Product	Quantity
Doug	1600p	08/20/2019	Siding	3 boxes
Doug	1600p	08/20/2019	Caulk	12 tubes
Tom	221b	10/29/2019	Soffit	17 meters
Tom	221b	10/29/2019	Sealant	1 liter
Doug	1600p	09/09/2018	Soffit	34 meters
Tom	221b	10/29/2019	Siding	2 boxes
Tom	221b	10/29/2019	Caulk	4 tubes
Doug	1600p	08/20/2018	Sealant	2 liters



Nested and repeated columns improve the efficiency of BigQuery with relational source data

Denormalized flattened table

Customer	OrderID	Date	Product	Quantity
Doug	1600p	08/20/2019	Siding	3 boxes
Doug	1600p	08/20/2019	Caulk	12 tubes
Tom	221b	10/29/2019	Soffit	17 meters
Tom	221b	10/29/2019	Sealant	1 liter
Doug	1600p	8/20/2019	Soffit	34 meters
Tom	221b	10/29/2019	Siding	2 boxes
Tom	221b	10/29/2019	Caulk	4 tubes
Doug	1600p	08/20/2019	Sealant	2 liters

Denormalized with nested and repeated data

Order.ID	Order.Date	Order.Product	Order.Quantity
1600p	08/20/2019	Siding	3 boxes
		Caulk	12 tubes
		Soffit	34 meters
		Sealant	2 liters
221b	10/29/2019	Soffit	17 meters
		Sealant	1 liter
		Siding	2 boxes
		Caulk	4 tubes

GO-JEK is a ride booking service in Indonesia running on Google Cloud

GO-JEK has 13+ PB of data queried each month

Orders

Events

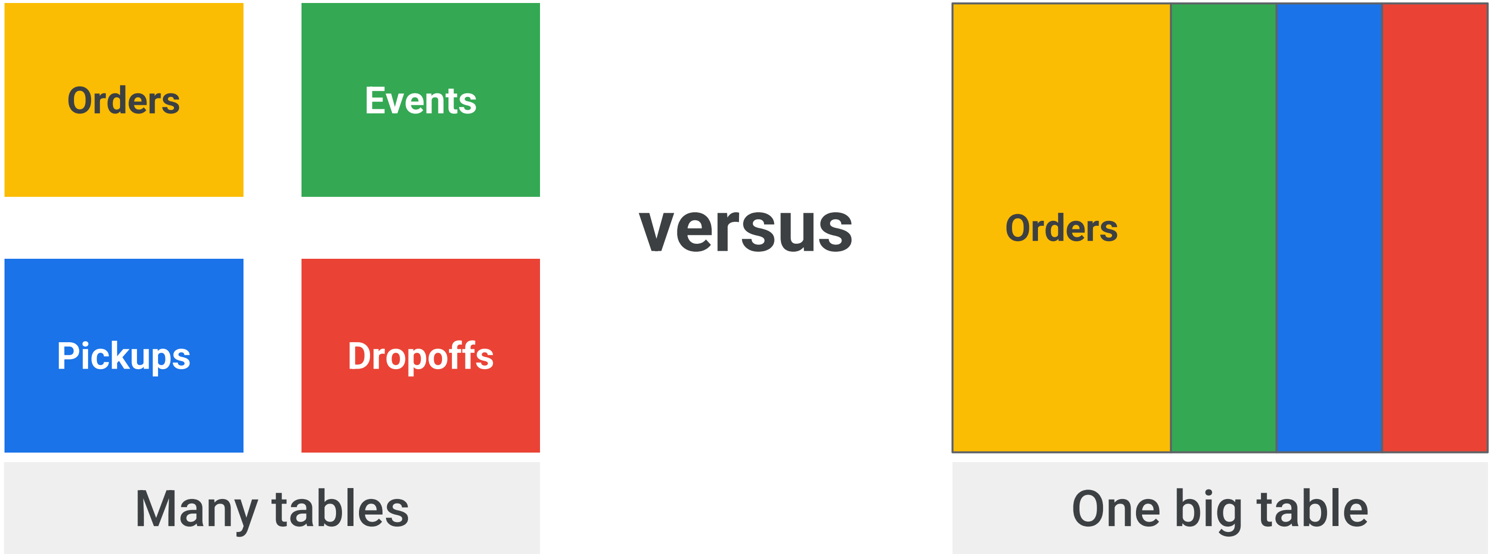
Pickups

Dropoffs

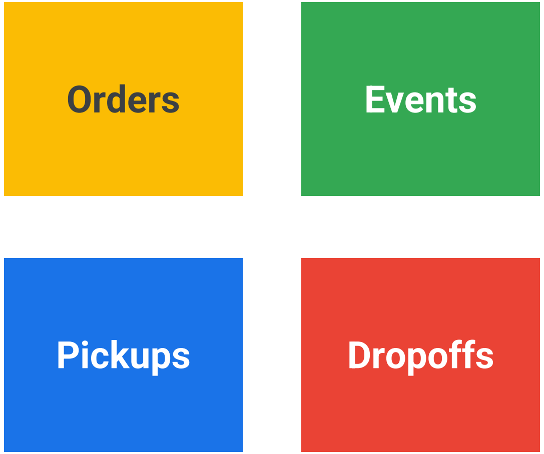
- Each ride is stored as an order.
- Each ride has a **single** pickup and dropoff.
- Each ride can have **one-to-many** events:
  - Ride confirmed
  - Driver en route
  - Pickup
  - Dropoff

How do you structure your data warehouse for scale?
Four separate and large tables that we join together?

Reporting approach: Should we normalize or denormalize?



Reporting approach: Should we normalize or denormalize?

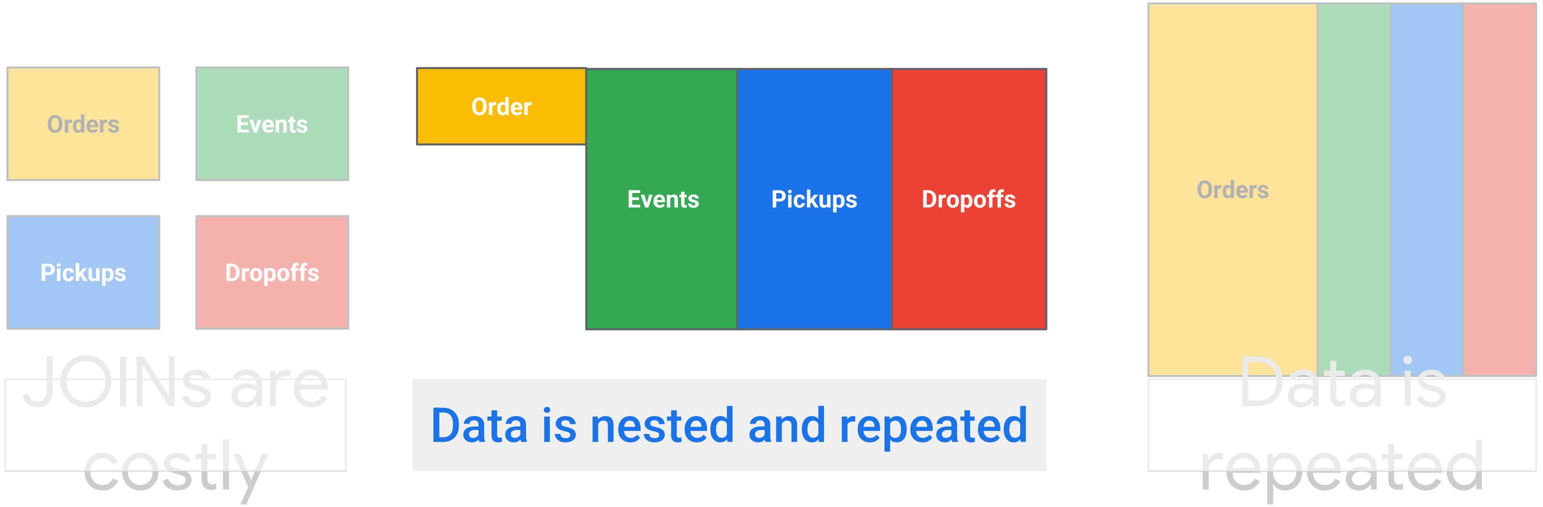


JOINs are costly

versus

Data is repeated

Nested and repeated fields allow you to have multiple levels of data granularity



Store complex data with nested fields (ARRAYS)

Row	order_id	service_type	payment_method	event.status	event.time	pickup.latitude
151	FD-5117	GO_FOOD	GOPAY	CREATED	2018-12-31 04:44:02.545210 UTC	-7.75105
				COMPLETED	2018-12-31 05:06:27.897769 UTC	
				PICKED_UP	2018-12-31 04:48:25.945331 UTC	
				DRIVER_FOUND	2018-12-31 04:44:06.869010 UTC	
152	FD-6834	GO_FOOD	CASH	PICKED_UP	2018-12-31 12:49:52.518880 UTC	1.121272
				DRIVER_FOUND	2018-12-31 12:40:14.214843 UTC	
				COMPLETED	2018-12-31 13:04:00.291780 UTC	
				CREATED	2018-12-31 12:40:13.431094 UTC	
153	FD-6293	GO_FOOD	PARTIAL_PAYMENT	PICKED_UP	2018-12-31 04:33:11.856445 UTC	-7.9657554

Report on all data in once place with STRUCTS

Row	order_id	service_type	payment_method	event.status	event.time	pickup.latitude	pickup.longitude	destination.latitude	destination.longitude
151	FD-5117	GO_FOOD	GOPAY	CREATED	2018-12-31 04:44:02.545210 UTC	-7.75105	110.410561	-7.7430367	110.4046433
				COMPLETED	2018-12-31 05:06:27.897769 UTC				
				PICKED_UP	2018-12-31 04:48:25.945331 UTC				
				DRIVER_FOUND	2018-12-31 04:44:06.869010 UTC				
152	FD-6834	GO_FOOD	CASH	PICKED_UP	2018-12-31 12:49:52.518880 UTC	1.121272	104.049739	1.1368655	104.03322
				DRIVER_FOUND	2018-12-31 12:40:14.214843 UTC				
				COMPLETED	2018-12-31 13:04:00.291780 UTC				
				CREATED	2018-12-31 12:40:13.431094 UTC				
153	FD-6293	GO_FOOD	PARTIAL_PAYMENT	PICKED_UP	2018-12-31 04:33:11.856445 UTC	-7.9657554	112.6247491	-7.9384084	112.6227862
				COMPLETED	2018-12-31 04:56:05.885521 UTC				
				CREATED	2018-12-31 04:16:24.356539 UTC				
				DRIVER_FOUND	2018-12-31 04:16:25.643766 UTC				
154	FD-7817	GO_FOOD	CASH	COMPLETED	2018-12-31 09:14:44.897136 UTC	-6.353915	106.247312	-6.368896	106.25787
				PICKED_UP	2018-12-31 09:01:11.471274 UTC				
				CREATED	2018-12-31 08:40:31.821796 UTC				
				DRIVER_FOUND	2018-12-31 08:40:32.910319 UTC				

Table JSON

First < Prev Rows 151 - 154 of 1137 Next > Last

Nested ARRAY fields and STRUCT fields allow for differing data granularity in the same table

Row	order_id	service_type	payment_method	event.status	event.time	pickup.latitude	pickup.longitude	destination.latitude	destination.longitude	total_distance_km	pricing_type
151	FD-5117	GO_FOOD	GOPAY	CREATED	2018-12-31 04:44:02.545210 UTC	-7.75105	110.410561	-7.7430367	110.4046433	1.56	regular
				COMPLETED	2018-12-31 05:06:27.897769 UTC						
				PICKED_UP	2018-12-31 04:48:25.945331 UTC						
				DRIVER_FOUND	2018-12-31 04:44:06.869010 UTC						
152	FD-6834	GO_FOOD	CASH	PICKED_UP	2018-12-31 12:49:52.518880 UTC	1.121272	104.049739	1.1368655	104.03322	4.84	surge
				DRIVER_FOUND	2018-12-31 12:40:14.214843 UTC						
				COMPLETED	2018-12-31 13:04:00.291780 UTC						
				CREATED	2018-12-31 12:40:13.431094 UTC						
153	FD-6293	GO_FOOD	PARTIAL_PAYMENT	PICKED_UP	2018-12-31 04:33:11.856445 UTC	-7.9657554	112.6247491	-7.9384084	112.6227862	4.68	regular
				COMPLETED	2018-12-31 04:56:05.885521 UTC						
				CREATED	2018-12-31 04:16:24.356539 UTC						
				DRIVER_FOUND	2018-12-31 04:16:25.643766 UTC						
154	FD-7817	GO_FOOD	CASH	COMPLETED	2018-12-31 09:14:44.897136 UTC	-6.353915	106.247312	-6.368896	106.25787	3.51	regular
				PICKED_UP	2018-12-31 09:01:11.471274 UTC						
				CREATED	2018-12-31 08:40:31.821796 UTC						
				DRIVER_FOUND	2018-12-31 08:40:32.910319 UTC						

Table JSON

First < Prev Rows 151 - 154 of 1137 Next > Last

Your turn

- Practice reading the new schema.
- Spot the STRUCTS.
- Type RECORD = STRUCTS.

booking

Schema | Details | Preview

Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event. status	STRING	NULLABLE
event. time	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE
pickup. latitude	FLOAT	NULLABLE
pickup. longitude	FLOAT	NULLABLE
destination	RECORD	NULLABLE
destination. latitude	FLOAT	NULLABLE
destination. longitude	FLOAT	NULLABLE
total_distance_km	FLOAT	NULLABLE
pricing_type	STRING	NULLABLE
duration	RECORD	NULLABLE
duration. booking_to_dispatch	FLOAT	NULLABLE
duration. booking_to_pickup	FLOAT	NULLABLE

Practice reading the new schema

- Practice reading the new schema.
- Spot the STRUCTS.
- Type RECORD = STRUCTS.

booking

Schema | Details | Preview

Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event. status	STRING	NULLABLE
event. time	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE
pickup. latitude	FLOAT	NULLABLE
pickup. longitude	FLOAT	NULLABLE
destination	RECORD	NULLABLE
destination. latitude	FLOAT	NULLABLE
destination. longitude	FLOAT	NULLABLE
total_distance_km	FLOAT	NULLABLE
pricing_type	STRING	NULLABLE
duration	RECORD	NULLABLE
duration. booking_to_dispatch	FLOAT	NULLABLE
duration. booking_to_pickup	FLOAT	NULLABLE

Events

Pickups

Destination

Duration

Your turn

- Practice reading the new schema.
- Spot the ARRAYS.
    - Hint: Look at Mode.

booking

Schema Details Preview

Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event. status	STRING	NULLABLE
event. time	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE

Practice reading the new schema

- Practice reading the new schema.
- Spot the ARRAYS.
- REPEATED = ARRAY

booking

Schema | Details | Preview

Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event. status	STRING	NULLABLE
event. time	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE

Events

Row	order_id	service_type	payment_method	event.status	event.time
151	FD-5117	GO_FOOD	GOPAY	CREATED	2018-12-31 04:44:02.545210 UTC
				COMPLETED	2018-12-31 05:06:27.897769 UTC

Status and Time in an ARRAY of event STRUCTs

Recap

- STRUCTS (RECORD)
- ARRAYS (REPEATED)
- ARRAYS can be part of regular fields or STRUCTS..
- A single table can have many STRUCTS.

booking

Schema Details Preview

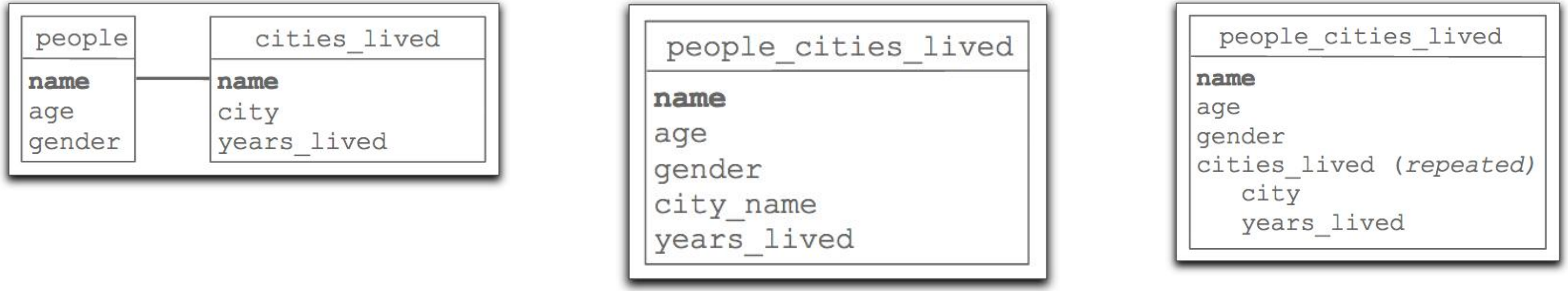
Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event. **status**	STRING	NULLABLE
event. **time**	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE
pickup. **latitude**	FLOAT	NULLABLE
pickup. **longitude**	FLOAT	NULLABLE
destination	RECORD	NULLABLE
destination. **latitude**	FLOAT	NULLABLE
destination. **longitude**	FLOAT	NULLABLE
total_distance_km	FLOAT	NULLABLE
pricing_type	STRING	NULLABLE
duration	RECORD	NULLABLE
duration. **booking_to_dispatch**	FLOAT	NULLABLE
duration. **booking_to_pickup**	FLOAT	NULLABLE

04

ARRAY and STRUCT Syntax in BigQuery

Recap: BigQuery architecture introduces repeated fields

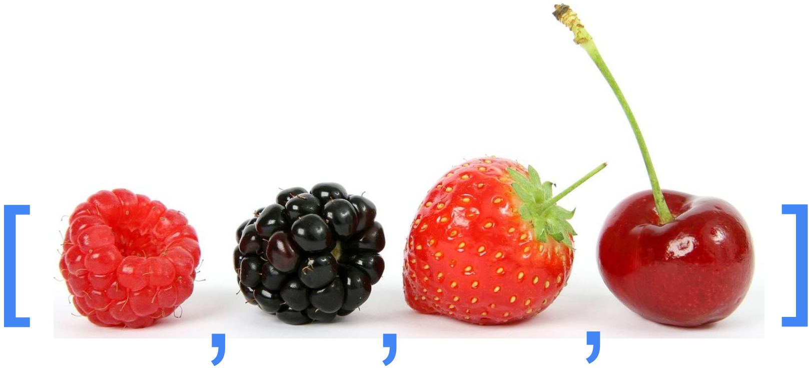
Normalized | Denormalized | Repeated



Less performant → High performing

Arrays are supported natively in BigQuery

Arrays are ordered lists of zero or more data values that **must have the same data type**.



Create an array with brackets [ ]

Working with SQL arrays in BigQuery

```
#standardSQL
SELECT ARRAY<STRING>
   ['raspberry', 'blackberry',
             'strawberry',
'cherry']
AS fruit_array
```

BigQuery
unflattened output

Row	fruit_array
1	raspberry
	blackberry
	strawberry
	cherry

BigQuery can infer data types for arrays

```
#standardSQL
SELECT ARRAY
   ['raspberry', 'blackberry',
            'strawberry',
'cherry']
AS fruit_array
```

BigQuery
unflattened output

Row	fruit_array
1	raspberry
	blackberry
	strawberry
	cherry

Index into the elements of an array

```
#standardSQL
WITH fruits AS (SELECT ['raspberry', 'blackberry', 'strawberry',
'cherry']
AS fruit_array)

SELECT fruit_array[OFFSET(2)]
AS zero_indexed
FROM fruits
```

What fruit name is returned?

Index into the elements of an array with **OFFSET**

```
#standardSQL                          0               1
WITH fruits AS (SELECT ['raspberry', 'blackberry',
 2             3
'strawberry', 'cherry'] AS fruit_array)
SELECT fruit_array[OFFSET(2)]
AS zero_indexed
FROM fruits
```

Row	zero_indexed
1	strawberry

Offset versus ordinal

```
#standardSQL
WITH fruits AS (SELECT ['raspberry', 'blackberry',
'strawberry', 'cherry'] AS fruit_array)
SELECT fruit_array[ORDINAL(2)]
AS one_indexed
FROM fruits
```

Index into the elements of an array

```
#standardSQL
WITH fruits AS (SELECT ['raspberry', 'blackberry', 'strawberry',
'cherry'] AS fruit_array)

SELECT fruit_array[OFFSET(999)]*
AS zero_indexed
FROM fruits
```

* Failed queries are at no charge.

Count the elements in an array

```
#standardSQL
WITH fruits AS (SELECT ['raspberry', 'blackberry', 'strawberry',
'cherry'] AS fruit_array)

SELECT ARRAY_LENGTH(fruit_array)
AS array_size
FROM fruits
```

Row	array_size
1	4

BigQuery uses unflattened arrays

```
#standardSQL
SELECT
  ['apple','pear', 'plum']
  AS item,
  'Jacob' AS customer
```

Row	item	customer
1	apple	Jacob
	pear	
	plum	

BigQuery output: Item ⟶ Unflattened array
Customer ⟶ Normal column

Flatten arrays with **UNNEST()**

```
#standardSQL
SELECT items, customer_name
FROM
    UNNEST(['apple', 'pear',

                  'peach']) AS
   items
   CROSS JOIN
    (SELECT 'Jacob' AS customer_name)
```

UNNEST flattens an array and returns a row for each element in the array, in random order.

Row	items	customer_name
1	apple	Jacob
2	pear	Jacob
3	peach	Jacob

Flattened output

Recover array order using **OFFSET**

```
#standardSQL
SELECT index, items
FROM
   UNNEST(['apple', 'pear',
    'peach']) AS items
WITH OFFSET AS index
ORDER BY index
```

OFFSET is a virtual column with 0-based index for the order used in the unflattened array.

Row	index	items
1	0	apple
2	1	pear
3	2	peach

Unflattened order

Aggregate into an array with **ARRAY_AGG**

```
#standardSQL
WITH fruits AS
  (SELECT 'apple' AS fruit UNION ALL
   SELECT 'pear' AS fruit UNION ALL
   SELECT 'banana' AS fruit)
SELECT ARRAY_AGG(fruit)*
AS fruit_basket
FROM fruits
```

* arrays inside arrays are not allowed.

Row	fruit
1	apple
2	pear
3	banana

Row	fruit_basket
1	apple
	pear
	banana

Aggregate into an array with ARRAY()

```
#standardSQL
SELECT ARRAY(
   SELECT 'apple' AS fruit UNION ALL
   SELECT 'pear' AS fruit UNION ALL
   SELECT 'banana' AS fruit
)
AS fruit_basket
```

* arrays inside arrays are not allowed.

Row	fruit_basket
1	apple
	pear
	banana

Aggregate into an array with **ORDER BY**

```
#standardSQL
SELECT ARRAY(
   SELECT 'apple' AS fruit UNION ALL
   SELECT 'pear' AS fruit UNION ALL
   SELECT 'banana' AS fruit
   ORDER BY fruit
)
AS fruit_basket
```

Row	fruit_basket
1	apple
	banana *
	pear

* Banana is now second.

Create sorted arrays with **ORDER BY**

```
#standardSQL
WITH fruits AS
  (SELECT 'apple' AS fruit UNION ALL
   SELECT 'pear' AS fruit UNION ALL
   SELECT 'banana' AS fruit)

SELECT ARRAY_AGG(fruit ORDER BY fruit)
AS fruit_basket
FROM fruits
```

Row	fruit_basket
1	apple
	banana *
	pear

* Banana is now second.

Or **ARRAY()** to build arrays from a subquery

```
#standardSQL
SELECT ARRAY(SELECT 'raspberry'
UNION ALL SELECT 'blackberry'
UNION ALL SELECT 'strawberry'
UNION ALL SELECT 'cherry'
) AS fruit_array
```

* a way to avoid [ ] for arrays

Row	fruit_array
1	raspberry
	blackberry
	strawberry
	cherry

Filter arrays using **WHERE IN**

```
#standardSQL
WITH groceries AS
(SELECT ['apple','pear','banana'] AS items
UNION ALL SELECT ['carrot','apple'] AS items
UNION ALL SELECT ['water','wine'] AS items)

SELECT
  items AS list
FROM groceries
WHERE 'apple' IN UNNEST(items)
```

Start with three arrays of groceries

Row	items
1	apple
	pear
	banana
2	carrot
	apple
3	water
	wine

Row	list
1	apple
	pear
	banana
2	carrot
	apple

STRUCTs are flexible containers

A **STRUCT** is a container of ordered fields, each with a type (required) and field name (optional).

STRUCTSs comply with the SQL 2011 standard.

You can store multiple data types in a **STRUCT** (even arrays!).

Working with STRUCTs

```
#standardSQL
SELECT STRUCT<INT64, STRING>(35,'Jacob')
```

What's with the result?

Store age as an integer; store name as a string.

Row	f0_._field_1	f0_._field_2
1	35	Jacob

A STRUCT and its elements can have names

```
#standardSQL
SELECT STRUCT(35 AS age,'Jacob' AS name)
AS customers
```

One STRUCT but many values. Like a table?

Also name the overall STRUCT container.

Row	customers.age	customers.name
1	35	Jacob

STRUCTs can even contain array values

```
#standardSQL
SELECT STRUCT(
   35 AS age,
   'Jacob' AS name,
   ['apple', 'pear','peach'] AS items)
AS customers
```

Row	customers.age	customers.name	customers.items
1	35	Jacob	apple
			pear
			peach

Arrays can contain STRUCTs as values

```
#standardSQL
SELECT ARRAY(

  SELECT AS STRUCT
   35 AS age,
  'Jacob' AS name,
  ['apple', 'pear', 'peach'] AS items

  UNION ALL

  SELECT AS STRUCT
   33 AS age,
  'Miranda' AS name,
  ['water', 'pineapple','ice cream'] AS items

) AS customers
```

Row	customers.age	customers.name	customers.items
1	35	Jacob	apple
			pear
			peach
	33	Miranda	water
			pineapple
			ice cream

Questions?

Google Cloud

05

Other Storage Best Practices

General guidelines to design the optimal schema for BigQuery

- Instead of joins, take advantage of nested and repeated fields in denormalized tables.
- Keep a dimension table smaller than 10 gigabytes normalized, unless the table rarely goes through `UPDATE` and `DELETE` operations.
- Denormalize a dimension table larger than 10 gigabytes, unless data manipulation or costs outweigh benefits of optimal queries.

Best practice for Surrogate keys

via UUID

```
SELECT
  GENERATE_UUID() AS SurrogateKey,
  *
 FROM
  `project.dataset.table`
```

via Hashing

```
SELECT
  (SHA256(bizKey)) AS SurrogateKey,
  *
FROM
  `project.dataset.table`
```

Surrogate keys substitute for natural keys and have no business meaning.

Avoid using ROW_NUMBER() to generate surrogate keys.

Prefer UUIDs in place of sequenced surrogate keys.

Prefer hashing for deterministic surrogate keys derived from the business key.

Use the expiration settings to remove unneeded tables and partitions

BigQuery supports lifecycle controls to age out data in accordance with user needs (regulatory, cost-driven, etc).

Audit events are generated as tables are removed, and recently removed tables can still be undeleted, if required.

Per table

- Delete table T at time X.
- Delete time partitions older than Y.

Per dataset

- Tables created in dataset D automatically expire M days after creation.
- Tables created with time-based partitions retain data for N days by default.

Configure the default table expiration for your datasets

```
bq update --default_table_expiration 7200 mydataset
```

Set the partition expiration for partitioned tables

```
bq update --time_partitioning_expiration 432000
mydataset.mytable
```

Configure expiration time for your tables

```
bq update --expiration 432000 mydataset.mytable
```

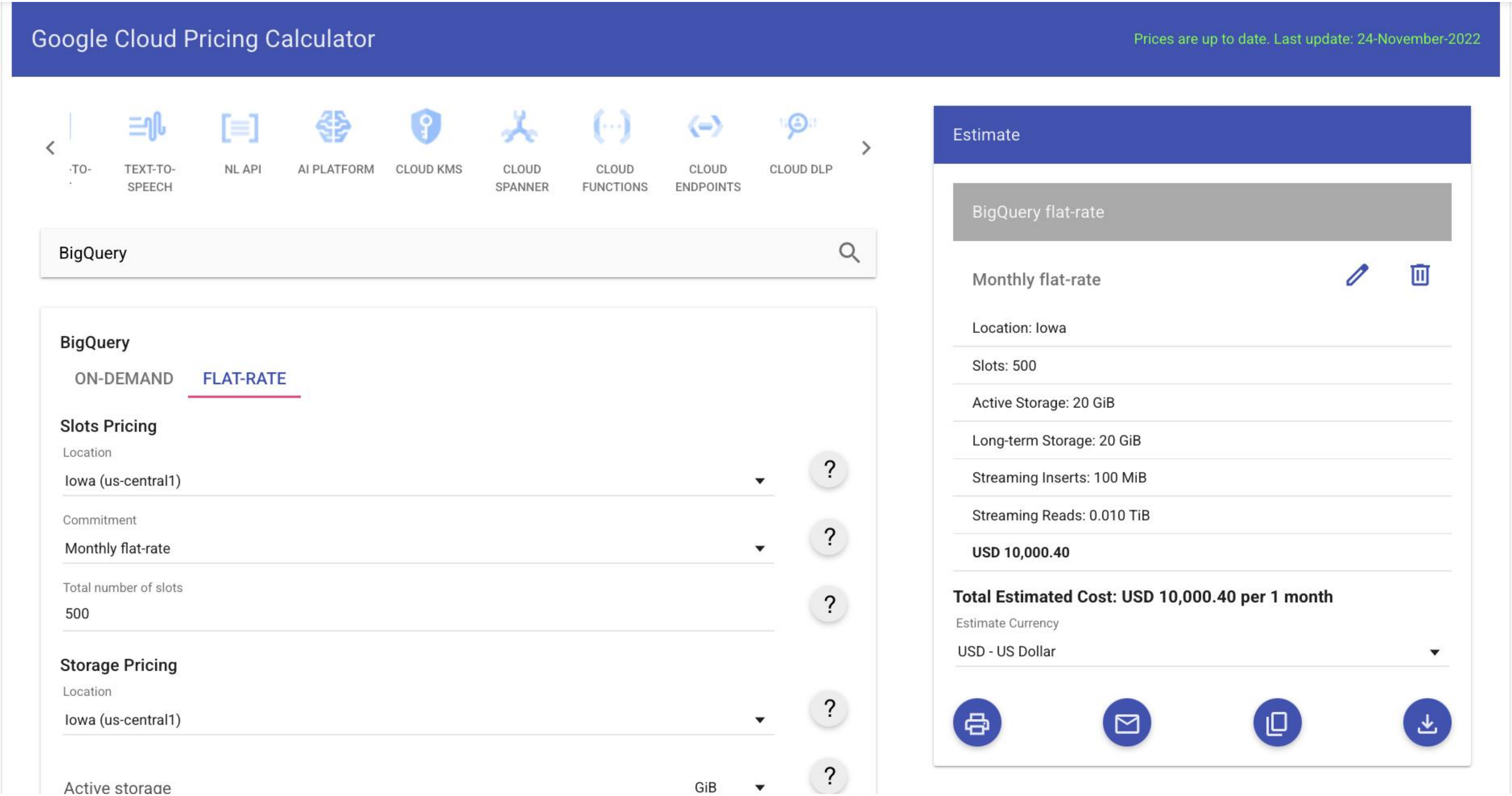
Benefit from BigQuery's long-term storage pricing

- For tables that are not edited for 90 consecutive days, the price per month of storage is much cheaper.
- For pricing purposes, each partition is considered separately.
  - Partitions that are not modified in the last 90 days are charged discounted pricing.

Use the same organization for managing BigQuery operations

- Latency of read and write operations is significantly improved by ensuring that both the source and destination tables are in the same organization.
- Therefore, check before running a BigQuery job.

Use the pricing calculator to estimate costs



Questions?

Google Cloud

Lab Intro

Working with JSON and Array Data in BigQuery

Objectives

- Loading semi-structured JSON into BigQuery.
- Creating and querying arrays.
- Creating and querying STRUCTs.
- Querying nested and repeated fields.