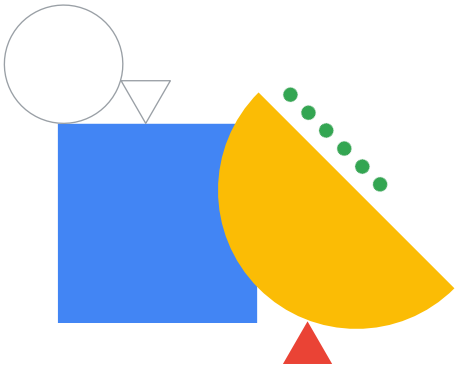
Google Cloud

Designing Schemas that Scale: Arrays and Structs in BigQuery



Agenda

01 BigQuery Versus Traditional Relational Data Architecture

Demo: Querying Nested and Repeated Fields in BigQuery

02 ARRAY and STRUCT Syntax

03 BigQuery Architecture

Lab: Schema Design for Performance: Arrays and Structs in BigQuery

This is one of the critical modules to pay attention to even if you're a SQL guru. Here we are going to look at the evolution of modern databases and end with how the technologies behind BigQuery addressed some of the limitations in architecture that prevented true petabyte-scale data analysis.

We'll discuss a core database concept called normalization and end with some pretty cool data structures, like having nested records in a table, that you may have not seen before.

Let's start the database evolution journey first.

BigQuery Versus Traditional Relational Data Architecture

Challenge: Data stored across multiple tables

Orders

Events

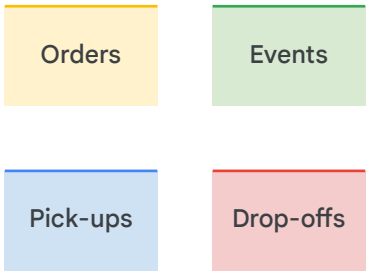
Pick-ups

Drop-offs

Relational model with many tables

1. We have built our data warehouse, but data problems are still the same. How can you create a single version of truth of data, for example if you want to create a single source table for booking that everybody can use? And we in GOJEK has booking data with many dimensions, such as location of pickup and drop off, booking status event, and many other dimensions.
2. How can you make sure that the table is rich in context and understandable from business perspective while still maintaining the fine granular details and quality of the data? For example maintaining the uniqueness of the data.
3. Luckily, BigQuery has the features that helps us with solving these kind of problem, and here, Evan will tells us how to leverage those BigQuery features.

Getting insights requires many **SQL JOINs**



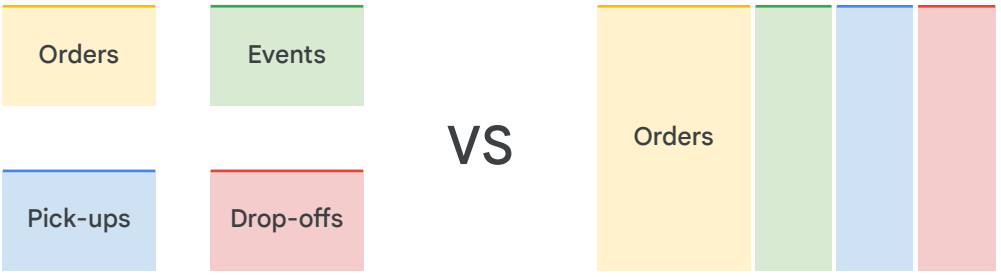
Relational model with many tables

```
SELECT * FROM
Orders LEFT JOIN Events USING(order_id)
LEFT JOIN Pick-ups USING(order_id)
LEFT JOIN Drop-offs USING(order_id)
```

Many table JOINs in SQL can be an expensive and time-consuming operation

Challenge: How can you best structure your data warehouse for insights?

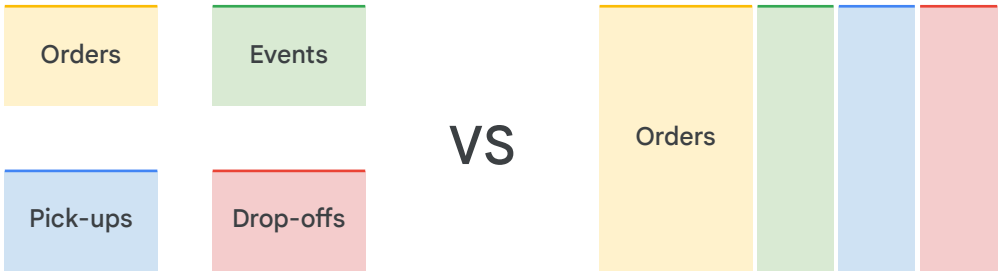
Reporting approach: Should we normalize or denormalize?



Many tables

One big table

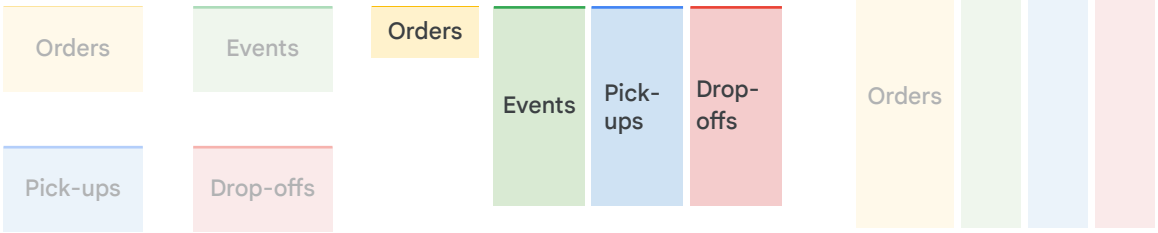
Reporting approach: Should we normalize or denormalize?



JOINs are costly

Data is repeated

Nested and repeated fields, the best of both worlds



JOINs are costly

Data is nested and repeated

Data is repeated

Store complex data with nested fields (ARRAYS)

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

GO JEK

Google Cloud

This is the example of a single version of truth of booking data in a table.

Report on all data in one place with **STRUCTS**

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

GO JEK

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

GO-JEK

Practice reading the new schema

Spot the STRUCTS

Type RECORD = STRUCT

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

GO JEK

Practice reading the new schema

Spot the STRUCTS

Type RECORD = STRUCT

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

Pick-ups

Destination

Duration

GO JEK

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
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

GO JEK

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

Status and Time are ARRAYs within the Event STRUCT

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
				PICKED_UP	2018-12-31 04:48:25.945331 UTC
				DRIVER_FOUND	2018-12-31 04:44:06.869010 UTC

GO JEK

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

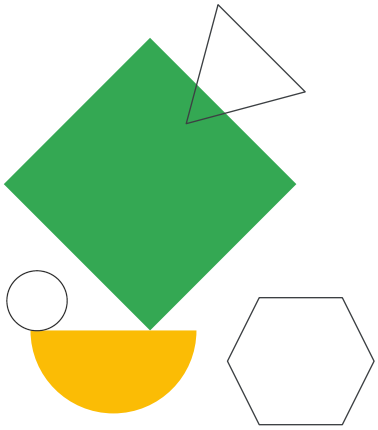
Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event. **status**	STRING	NULLABLE
event. **time**	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE
pickup. **latitude**	FLOAT	NULLABLE
pickup. **longitude**	FLOAT	NULLABLE
destination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

GO JEK

Demo

Querying Nested and Repeated Fields in BigQuery

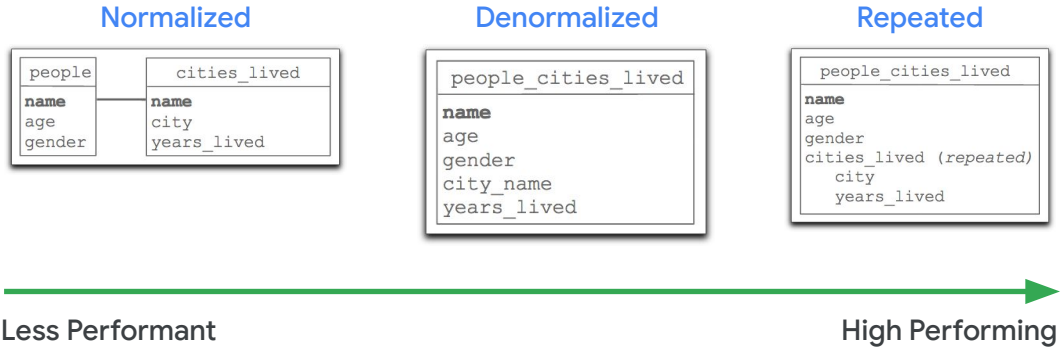
Practice creating and querying ARRAYs and STRUCTs



Refer to https://github.com/GoogleCloudPlatform/training-data-analyst/tree/master/courses/data-to-insights/demos/schemas-that-scale.sql

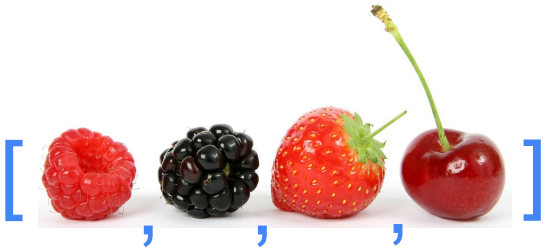
ARRAY and STRUCT Syntax

Recap: BigQuery architecture introduces repeated fields



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 [ ]

- SQL 2011 version (successor to 1999)
- can be declared without cardinality
- think indirection for arrays of arrays

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

Create an array with brackets… array elements must share the same datatype (e.g. string values for these fruits). You must specify the type

Flattened = BigQuery creates a row for each element in an array

See:
https://cloud.google.com/bigquery/docs/reference/standard-sql/arrays#flattening-arrays

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

Create an array with brackets… array elements must share the same datatype (e.g. string values for these fruits). You must specify the type

Flattened = BigQuery creates a row for each element in an array

See:
https://cloud.google.com/bigquery/docs/reference/standard-sql/arrays#flattening-arrays

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?

Create an array with brackets… array elements must share the same datatype (e.g. string values for these fruits)

Access the length of the array with functions like ARRAY_LENGTH. Access elements outside the array boundary (if you error you don't pay for the query)

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

Create an array with brackets… array elements must share the same datatype (e.g. string values for these fruits)

Access the length of the array with functions like ARRAY_LENGTH. Access elements outside the array boundary (if you error you don't pay for the query)

Offset vs Ordinal

```
#standardSQL                    1           2
WITH fruits AS (SELECT ['raspberry', 'blackberry',
      3           4
'strawberry', 'cherry'] AS fruit_array)

SELECT fruit_array[ORDINAL(2)]
AS one_indexed
FROM fruits
```

Create an array with brackets… array elements must share the same datatype (e.g. string values for these fruits)

Access the length of the array with functions like ARRAY_LENGTH. Access elements outside the array boundary (if you error you don't pay for the query)

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

Create an array with brackets… array elements must share the same datatype (e.g. string values for these fruits)

Access the length of the array with functions like ARRAY_LENGTH. Access elements outside the array boundary (if you error you don't pay for the query)

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

Create an array with brackets… array elements must share the same datatype (e.g. string values for these fruits)

Access the length of the array with functions like ARRAY_LENGTH

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

BigQuery uses **unflattens** Arrays with #standardSQL

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

To access array elements, we first need to unpack them before we can perform operations on them. To do this, we use the SQL syntax of UNNEST(array) and CROSS JOIN

Working with arrays:
https://cloud.google.com/bigquery/docs/reference/standard-sql/arrays

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

To access array elements, we first need to unpack them before we can perform operations on them. To do this, we use the SQL syntax of UNNEST(array) and CROSS JOIN

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

Packing items back into an array is done through ARRAY_AGG()

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

Packing items back into an array is done through ARRAY_AGG()

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
```

* Banana is now 2nd

Row	fruit_basket
1	apple
	banana *
	pear

Packing items back into an array is done through ARRAY_AGG()

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 2nd

You can sort items within array with ORDER BY

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 “desugar” [ ] for arrays

Row	fruit_array
1	raspberry
	blackberry
	strawberry
	cherry

Filter arrays using **WHERE IN**

```
#standardSQL
WITH groceries AS
(SELECT ['apple','pear','banana'] AS list
UNION ALL SELECT ['carrot','apple'] AS list
UNION ALL SELECT ['water','wine'] AS list)
SELECT
  SELECT list
  FROM groceries
  WHERE 'apple' IN UNNEST(list)
```

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)

SQL 2011 structured type

Can store multiple data types in a **STRUCT** (even arrays!)

Maximum row size ~100 MB. The maximum row size limit is approximate, as the limit is based on the internal representation of row data. The maximum row size limit is enforced during certain stages of query job execution.

Much like we saw an array of fruits which shared the same type, a struct can take multiple fields that have different data types. They can even have arrays within them!

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

Structs and its elements can have names

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

Also name the overall STRUCT container

One STRUCT but many values. Like a table?

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

BigQuery Architecture

Now let's talk about the positives and negatives of splitting apart your one massive table into smaller pieces through normalization. Then we'll introduce how the technologies behind BigQuery address these issues.

Challenge: Doesn't a fully denormalized schema lead to really wide and slow tables?

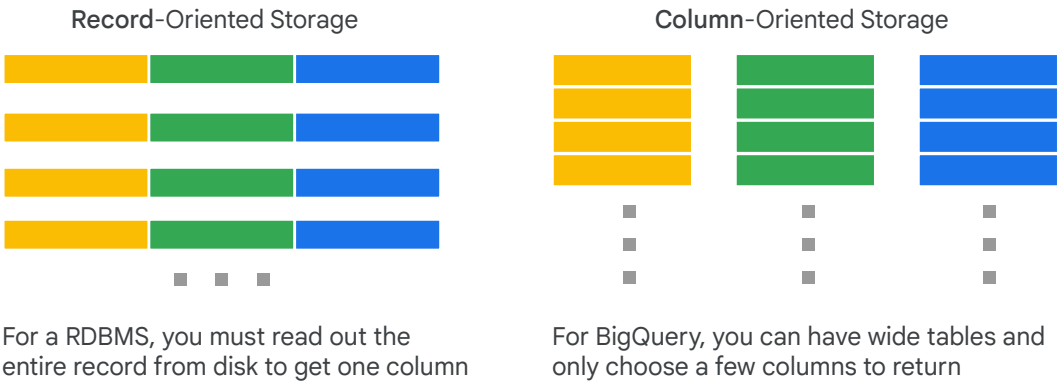
Row	order_id	service_type	payment_method	event.status	event.time	pickup.latitude	pickup.longitude	destination.latitude	destination.longitude	total_distance_km	pricing_type
151	FD-5117	GO_FOOD	GOPAY	CREATED	2018-12-31 04:44:02.545210 UTC	-7.75105	110.410561	-7.7430367	110.4046433	1.56	regular
				COMPLETED	2018-12-31 05:06:27.897769 UTC						
				PICKED_UP	2018-12-31 0[redacted] 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

GO JEK

BigQuery is column-oriented storage compared to traditional RDBMS' record-based storage



Technical deep dive into how the BigQuery column storage format works: https://cloud.google.com/blog/big-data/2016/04/inside-capacitor-bigquerys-next-generation-columnar-storage-format

Why analytics is much faster on column-oriented data: https://loonytek.com/2017/05/04/why-analytic-workloads-are-faster-on-columnar-databases/

"If the operation requires to access only a handful of columns from a very large number of rows in the table (the case of analytic or data warehouse queries), row stores tend to become less efficient since we have to walk the row, skip over the columns not needed, extract the value from relevant column of interest and move on to the next row.

In columnar databases, values of a particular column are stored separately or individually — contiguous layout for values of a given column in-memory or on-disk. If the query touches only few columns, it is relatively faster to load all values of a "particular column" into memory from disk in fewer I/Os and further into CPU cache in fewer instructions.

In other words, column stores are capable of accessing values of a particular column independently without bothering about the other columns in the table which may not even be needed by the query plan. Thus in columnar storage format, there is no need to read (and then skip/discard) columns that are not needed and this provides better

utilization of available I/O and CPU-memory bandwidth for analytical queries.”

BigQuery breaks apart all tables into smaller shards of data to enable massively parallel reads and operations

One Large Table

Company ID	Company Name
161218560	NY Association Inc.
...	...
...	...
...	...

10 Billion Row Table

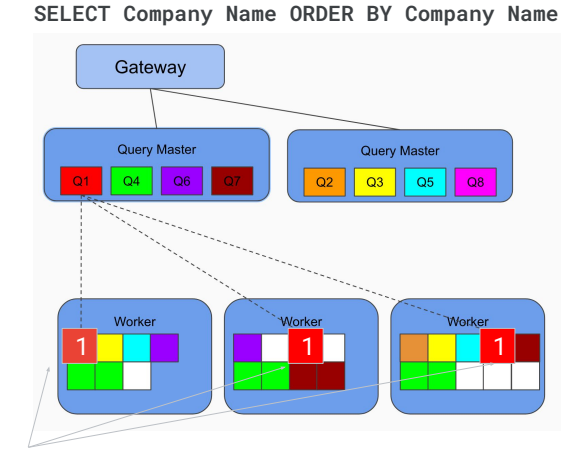
BigQuery breaks up the table (not seen to you) into many small shards

BigQuery stores data in a massively distributed format. This is the same for Google Cloud Storage and is what powers apps like Gmail, Ads, and many more Google products.

BigQuery auto-optimizes the balancing and partitioning of these data shards behind-the-scenes: https://cloud.google.com/blog/big-data/2016/05/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow

BigQuery manages the efficient allocation of workers and shards

- Up to 2,000 workers to process concurrent queries (on-demand tier)
- "Fairness model" for allocation
- You can setup custom quotas and limits for your teams



Shards of data are read and processed in parallel

https://cloud.google.com/blog/big-data/2016/01/bigquery-under-the-hood

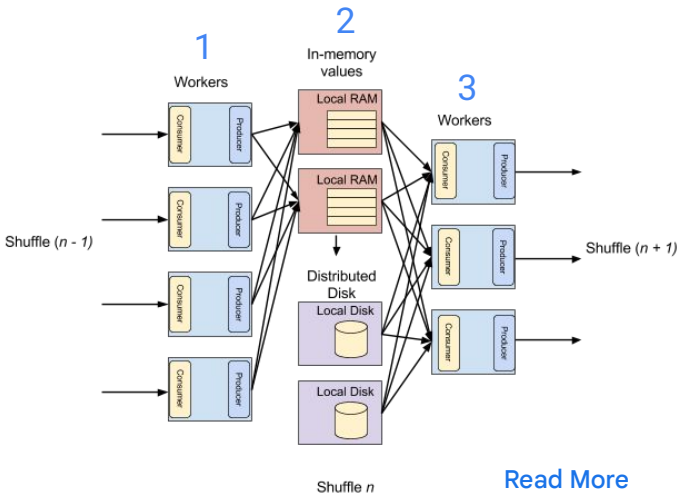
"Dremel (BigQuery query engine) dynamically apportions slots to queries on an as needed basis, maintaining fairness amongst multiple users who are all querying at once. A single user can get thousands of slots to run their queries.

Dremel is widely used at Google — from search to ads, from youtube to gmail — so there's great emphasis on continuously making Dremel better. BigQuery users get the benefit of continuous improvements in performance, durability, efficiency and scalability, without downtime and upgrades associated with traditional technologies."

BigQuery workers communicate by shuffling data in-memory

1. Workers Consume data values and perform operations in parallel
2. Workers Produce output to the In-Memory Shuffle Service
3. Workers Consume New Data and continue processing

   Workers (one or more slots) scale to meet the demand of the processing task.



Shuffle is a transient, transactional communication layer. Slots don't communicate directly with one another, they pass data via the use of shuffle.

Shuffle is partitioned, so that the outputs can be dealt with in parallel fashion. Hashing results into partitions allows us to align and localize data so that an individual worker can process it without needing global keyspace awareness.

Projects typically get some "fair share" budget of fast, memory-backed shuffle resources tracked across all their active queries. When this is exhausted, shuffle traffic may transition to slower disk-backed methods.

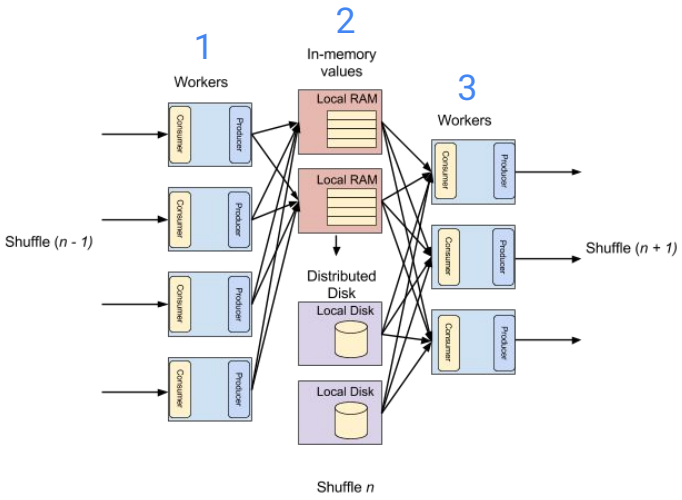
How BigQuery queries work:
https://cloud.google.com/blog/big-data/2016/01/anatomy-of-a-bigquery-query

In-memory execution:
https://cloud.google.com/blog/big-data/2016/08/in-memory-query-execution-in-google-bigquery

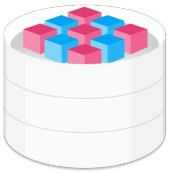
More on Shuffle:
https://medium.com/google-cloud/the-12-components-of-google-bigquery-c2b49829a7c7

BigQuery shuffling enables massive scale

- Shuffle allows BigQuery to **process massively parallel petabyte-scale data** jobs
- Everything after Query Execution is **automatically scaled and managed**
- All queries large and small use shuffle



Summary: BigQuery architecture is designed for petabyte-scale querying performance



Tables are broken into pieces, called shards, to allow for scalability

BigQuery uses compressed column-based storage for fast retrieval

Structs and arrays are data type containers that are foundational to repeated fields

Row	date	top_articles.title
1	2010-08-23	Why GNU grep is Fast
		Readme Driven Development
2	2010-04-26	Police raid Gizmodo editor's house
		Not even in South Park?
3	2009-09-15	Learning Advanced JavaScript
		Sub-pixel re-workings of YouTube and BBC favicons

Tables with repeated fields are conceptually like pre-joined tables

There was a lot to cover in this module and even you SQL gurus out there may have seen a lot of new architecture concepts.

To summarize the key points recall that BigQuery (and even Google Cloud Storage) rely on breaking apart datasets into smaller chunks called shards which are then stored on persistent disk. Working with data in smaller shards allows BigQuery to massively parallel process your query across many workers at the same time.

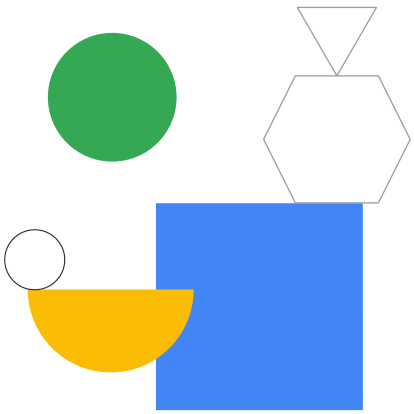
Another key difference for BigQuery is that your actual data table isn't stored as a table at all. Instead each individual column is broken off, compressed, and stored individually. This is why when you limit your SELECT queries to just the columns you need, BigQuery does not need to fetch the entire record from disk which saves you on bytes processed.

Lastly, BigQuery natively supports structs and arrays as part of repeated fields. Structs are your flexible data container and an array of multiple structs is how repeated fields are setup. Repeated fields offer the performance benefit of pre-joined tables (parent and child records stored in the same place) while avoiding the downfall of increased scanning time of a large detail record table when you don't need that level of detail. You access these nested child records by using the SQL UNNEST() syntax in tandem with a CROSS JOIN.

Phew! That was a lot of cover so it's time now to practice these concepts in our next lab.

Lab Intro

Schema Design for Performance:
Arrays and Structs in BigQuery



Lab objectives

01 Practice working with Arrays in SQL

02 Creating your own arrays with ARRAY_AGG()

03 Querying datasets that already have ARRAYs

04 Introduction to STRUCTs

05 Practice with STRUCTs and ARRAYs

06 Unpacking ARRAYs with UNNEST( )