Schema Design

Next, we will talk about efficient data warehouse schema design.

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

Take a look at the Original data table here and the Normalized data tables which contain the same data.

The data in the Original table is organized visually -- as you might have used merged cells or columns in a spreadsheet. But if you had to write an algorithm to process the data, how might you approach it? Access could be by rows, by columns, by rows-then-columns. And the different approaches would perform differently based on the query. Also, your method might not be parallelizable.

The original data can be interpreted and stored in many ways in a database. Normalizing the data means turning it into a relational system. This stores the data efficiently and makes query processing a clear and direct task. Normalizing increases the orderliness of the data. It is useful for saving space.

Many people with database experience will recognize this procedure. Normalizing data usually happens when a schema is designed for a database.

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

Denormalizing is the strategy of allowing duplicate field values for a column in a table in the data to gain processing performance.

Data is repeated rather than being relational. Flattened data takes more storage, but the flattened (non-relational) organization makes queries more efficient because they can be processed in parallel using columnar processing.

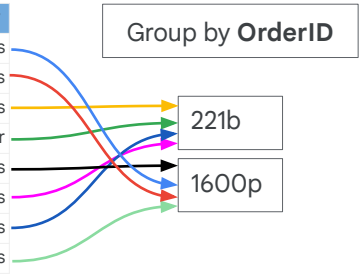
Specifically, denormalizing data enables BigQuery to more efficiently distribute processing among slots, resulting in more parallel processing and better query performance.

You would usually denormalize data before loading it into BigQuery.

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



However, there are cases where denormalizing data is bad for performance. Specifically, if you have to group by a column with a 1-to-many relationship. In the example shown, OrderID is such a column.

In this example, to group the data it must be shuffled. That often happens by transferring the data over a network between servers or systems. Shuffling is slow.

Fortunately, BigQuery supports a method to improve this situation.

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

BigQuery supports columns with nested and repeated data.

In this example, a denormalized flattened table is compared with one that has been denormalized and the schema takes advantage of nested and repeated fields. OrderID is a repeated field. Because this is declared in advance, BigQuery can store and process the data respecting some of the original organization in the data. Specifically, all order details for each order are colocated, which makes retrieval of the whole order more efficient.

For this reason, nested and repeated fields are useful for working with data that originates in relational databases.

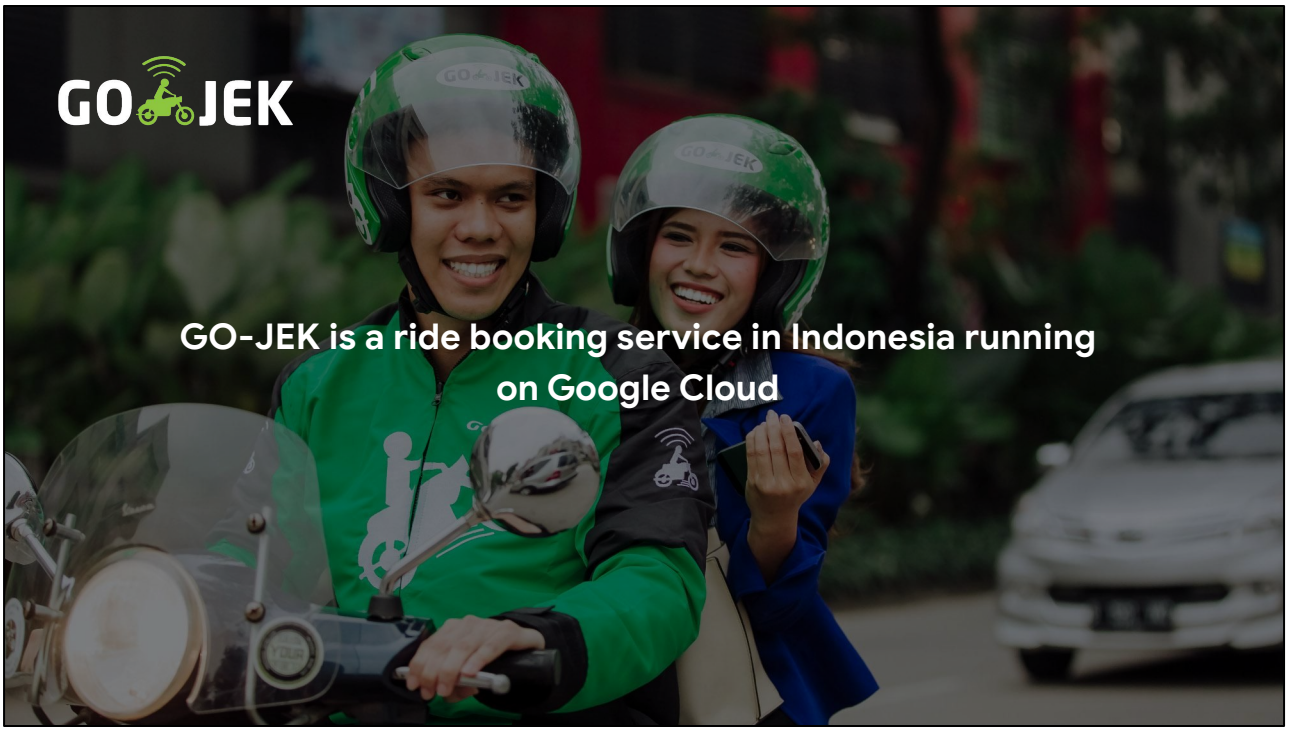
Nested columns can be understood as a form of repeated field. It preserves the relational qualities of the original data and schema while enabling columnar and parallel processing of the repeated nested fields. It is the best alternative for data that already has a relational pattern to it. Turning the relation into a nested or repeated field improves BigQuery performance.

Nested and repeated fields help BigQuery work with data sourced in relational databases.

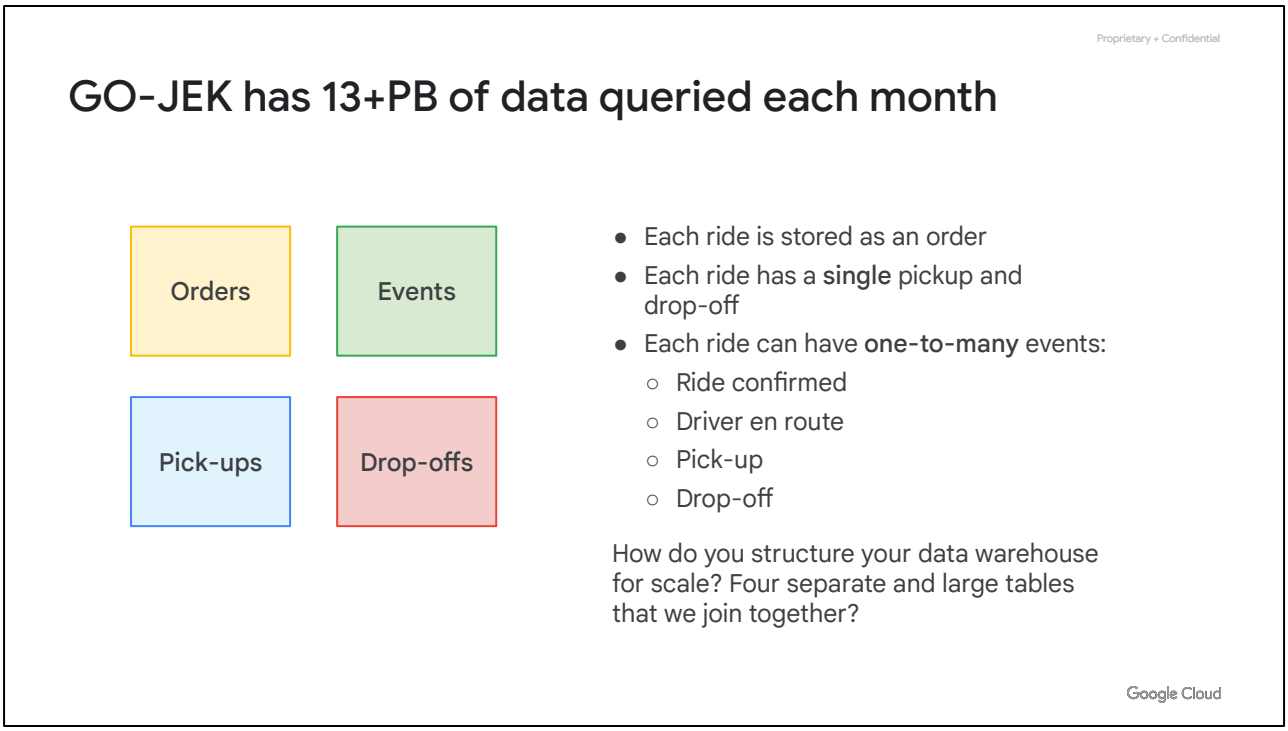
Look for nested and repeated fields whenever BigQuery is used in a hybrid solution in conjunction with traditional databases,

Nested and Repeated Fields

Let’s take a closer look at BigQuery’s support for nested and repeated fields and why this is such a popular schema design for enterprises.



I'll illustrate by using an example from a real business running on Google Cloud. GO-JEK is a company in Indonesia that is well known for its ride booking service…

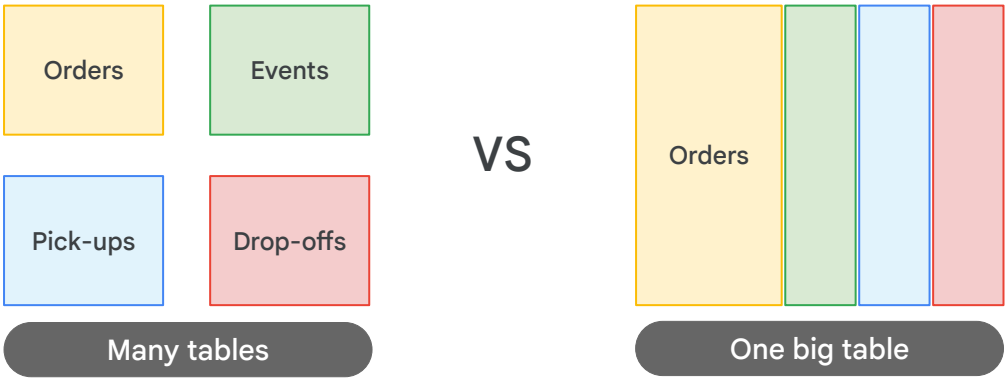


… and they process over a 13 petabyte of data on BigQuery per month from queries to support business decisions. What kind of decisions?

For GO-JEK, they track whenever a new customer places an order like hails a ride with their mobile app. That order is stored in an orders table. Each order has a single pick-up location and drop-off destination. For a single order, you could have one or many events like “Ride Ordered”, “Ride Confirmed”, “Drive En Route”, “Drop off Complete” etc.

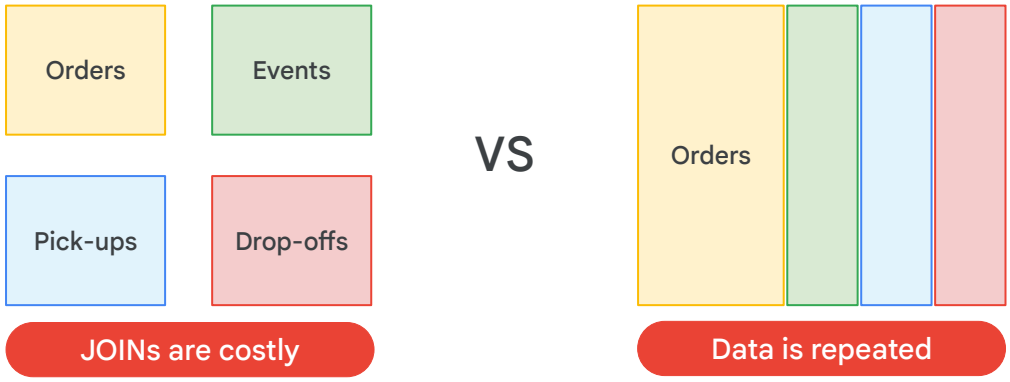
As a data engineer, how would you efficiently store these different pieces of data in your data warehouse? Keep in mind you need to support a large user base querying Petabytes per month.

Reporting approach: Should we normalize or denormalize?



Well as you saw earlier, we could store one fact in one place with the normalization route which is typical for relational systems. Or we could go the fully denormalized route and just store all levels of granularity in a single big table where you would have one OrderID like ‘123’ repeated in a row for each event that happens on that order. Faster for querying sure but what are the drawbacks?

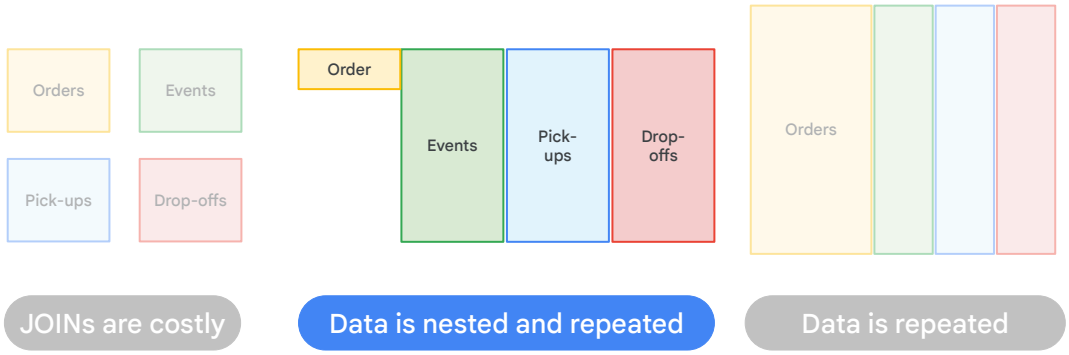
Reporting approach: Should we normalize or denormalize?



For relational schemas (normalized schemas) often the most intensive computational workloads are JOINS across very large tables. Remember RDBMS' are record based so they have to open each record entirely and pull out the join key from each table where a match exists. And that's assuming you know all the tables that need to be joined together! Imagine for each new piece of information about an order (like promotion codes, or user information) and you could be talking about a 10+ table join.

The alternative has different drawbacks. Pre-joining all your tables into one massive table makes reading data faster but you now have to be really careful if you have data at different levels of granularity. In our example, each row would be at the level of granularity of a specific event (like Driver Confirmed) for a given order. What does that mean for an OrderID like '123'? It is duplicated for each event on that order. Imagine if you're looking to join higher level information like the revenue per order and you now have to be exceedingly careful with aggregations to not double or triple count your duplicate OrderIDs. See the problem?

Nested and Repeated Fields allow you to have multiple levels of data granularity



One common solution in enterprise data warehouse schemas is to take advantage of nested and repeated data fields. You can have ONE ROW for each order and repeated values within that ONE ROW for data that is at a more granular level. For example, you could simply have an ARRAY of timestamps as your events. Let's see an example to illustrate this point.

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

Here you see it clearly. Shown here on screen are just 4 rows for 4 unique OrderIDs. Notice all that grey space in between the rows? That's because the event.status and event.time is at a deeper level of granularity. That means there are multiple repeated values for these events per each order. An ARRAY if a perfect data type to handle this repeated value and keep all the benefits of storing that data in a single row.

I mentioned the fields event.status and event.time. If this is one giant table, what is a dot doing in those column names? There are no other table aliases we've joined on … what's up with those fields?

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	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

Event, pickup, and destination are what are called STRUCT or structure data type fields in SQL. This isn't BigQuery specific, STRUCTS are standard SQL data types and BigQuery just supports them really well. STRUCTS you can think of as pre-joined tables within a table. So instead of having a separate table for EVENT and PICKUP and DESTINATION you simply NEST them within your main table.

So let's recap.

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

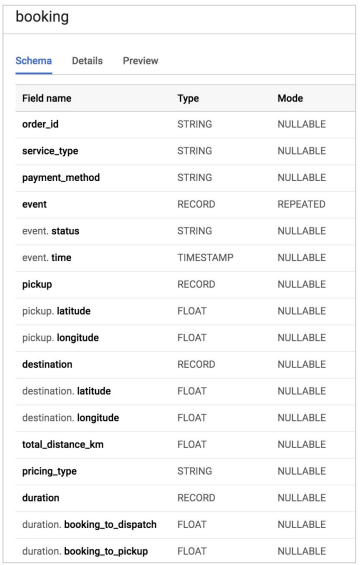
You can go deep into a single field and have it be more granular than the rest by using an ARRAY data type like you see here for STATUS and TIME.

And you can have really WIDE schemas by using STRUCTS which allow you to have multiple fields of the same or different data types within them (much like a separate table would). The major benefit of STRUCTs is that the data is conceptually pre-joined already so its much faster to query.

People often ask -- with really wide schemas (like a hundred columns) how is it still fast to query? Remember that BigQuery is column based storage not record based when storing data out on disk. If you did just a COUNT(order_id) here to get your total orders, BigQuery wouldn't even care that you have 99 other columns, some of which are more granular with ARRAY data types; it wouldn't even look at them. That gives you the best of both worlds if you're an analyst. Lots of data all in one place and no issues with multiple granularity pitfalls when doing aggregations.

Your turn

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



booking

Schema | Details | Preview

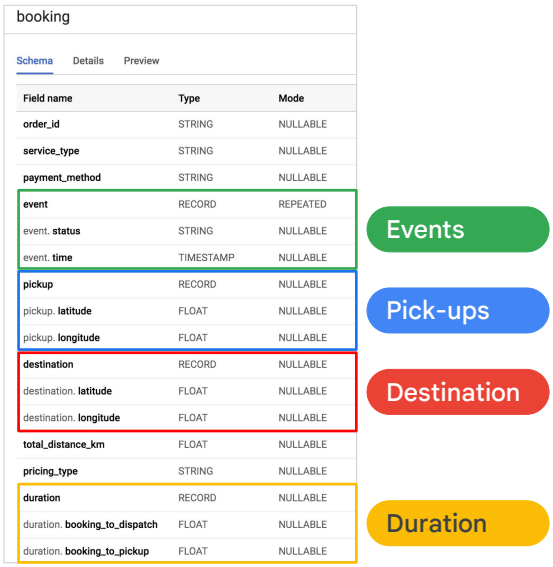
Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event. **status**	STRING	NULLABLE
event. **time**	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE
pickup. **latitude**	FLOAT	NULLABLE
pickup. **longitude**	FLOAT	NULLABLE
destination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

Now it’s your turn to practice reading one of these schemas that has nested and repeated fields. Take a moment and spot those STRUCTs. As a hint, you can look at the field name to see any field with a dot in the name OR you can look at the data type for any field values of the type RECORD (which means STRUCT).

Did you get them all?

Practice reading the new schema

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

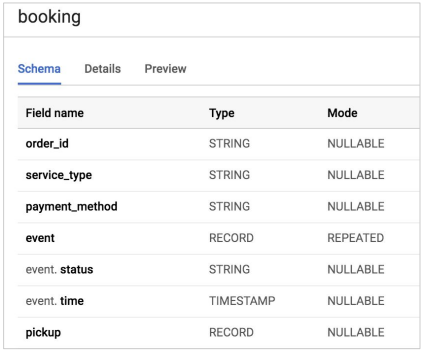


Here are the four STRUCTs in this dataset you saw earlier. Events, pickups, destination, and duration. Duration is a new one but we can simply keep adding more dimensions to our dataset by adding more STRUCTs.

Remember STRUCTs let you build really wide and informative schemas. Now it's time to go deep.

Your turn

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

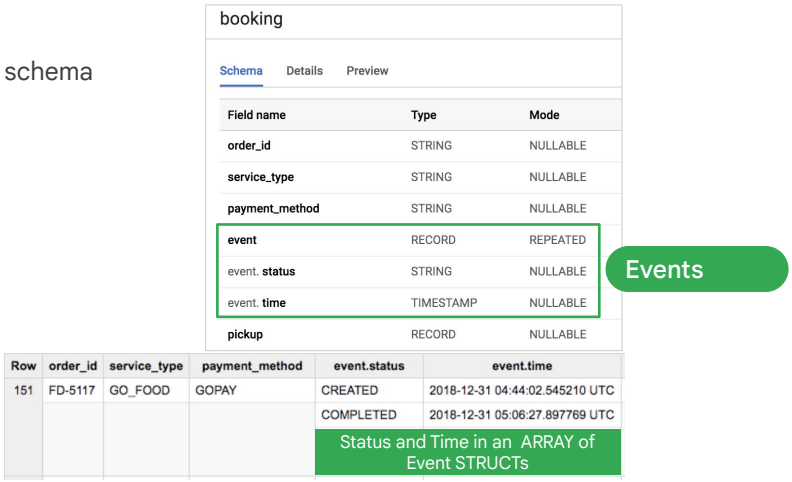


Find the ARRAY data types in this schema. As a hint, look at the Mode and find the REPEATED values.

Got them?

Practice reading the new schema

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



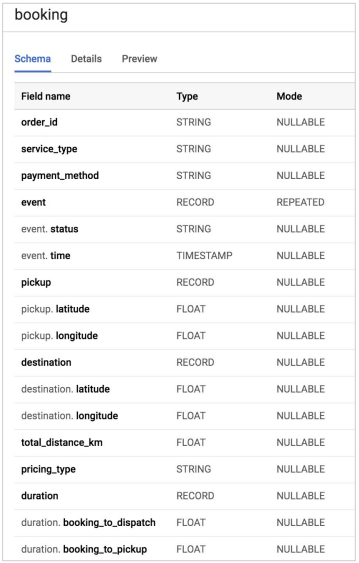
In this schema the repeated value is the EVENT STRUCT (which means here we have an ARRAY of EVENT STRUCTs with each having a status and time possibly)

A critical point I like to make here is that STRUCT and ARRAY data types in SQL can be absolutely independent of each other. You can have a regular column in SQL be an ARRAY column that has nothing to do with any STRUCT.

Likewise, you can have a STRUCT that has zero ARRAY field types in it's columns. The benefit of using them together is that ARRAYs allow a given field to go deep into granularity and STRUCTs allow you to organize all those useful fields into logical containers instead of separate tables.

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

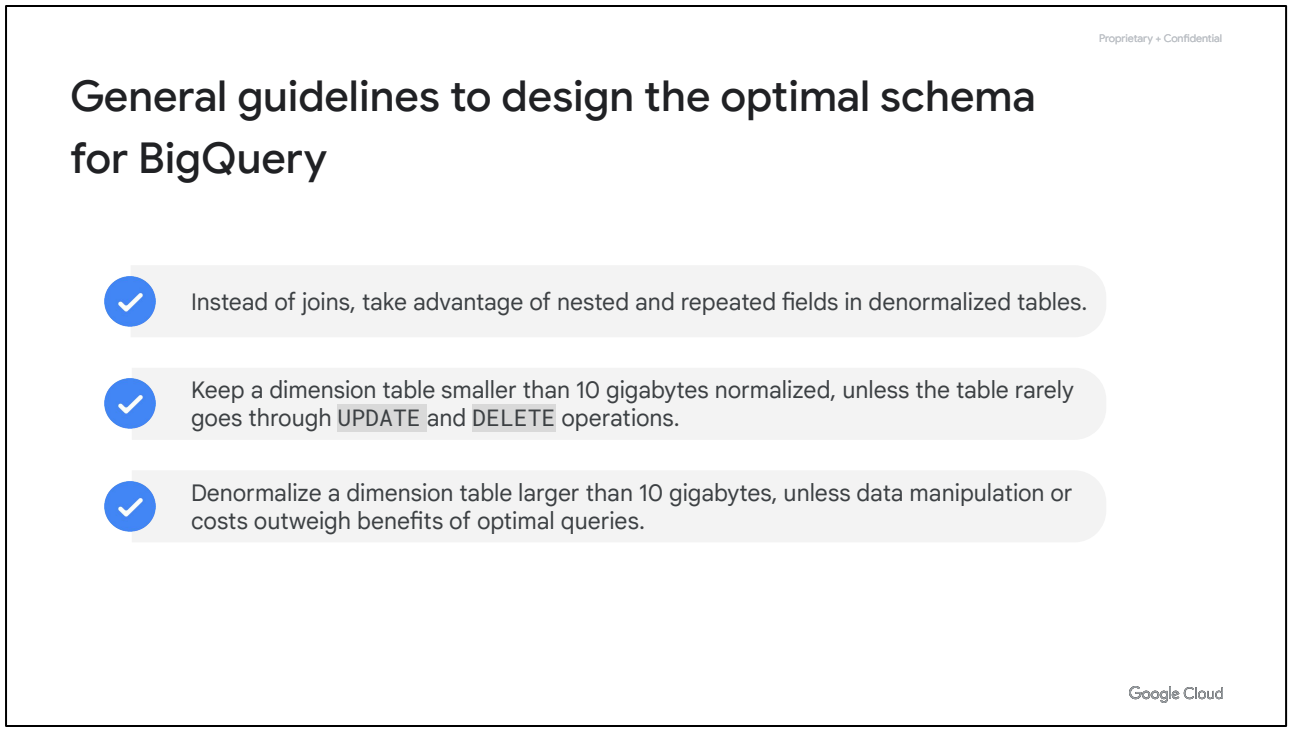
Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event. **status**	STRING	NULLABLE
event. **time**	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE
pickup. **latitude**	FLOAT	NULLABLE
pickup. **longitude**	FLOAT	NULLABLE
destination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

So here's the cheatsheet.

- STRUCTs are of type RECORD when looking at a schema , …
- …and ARRAYs are of MODE repeated. ARRAYs can be of any single type, like an ARRAY of floats or an ARRAY of strings, etc.
- ARRAYS can be part of a regular field or be part of a NESTED field nestled inside of a STRUCT.
- A single table can have zero to many STRUCTs and lastly, the real mind bending point, is that a STRUCT can have other STRUCTs nested inside of it as you will soon see in your upcoming lab which uses the real Google Analytics schema.

We've been talking a lot about nested and repeated fields so you're probably wondering what to do with your existing star-schema, snowflake and third normal form data. The great news is that BigQuery also works well with those schema types!

Use ARRAYS and STRUCTS when your data naturally arrives in that format and you'll benefit immediately from optimal performance. For the other schema types, bring them directly to BigQuery and you'll likely be pleased with the performance.



Let's recap some of the ways to design the schema of tables to improve query performance and lower query costs.

- It's much more efficient to define your schema to use nested, repeated fields instead of joins. Suppose you have orders and purchase items for each order. In a traditional relational database system, you'd have two tables: one table for purchase items, and another for orders, with a foreign key to connect the two tables. In BigQuery, it's much more efficient if you store each order in a row and have a nested, repeated column called purchase_item. ARRAYs are a native type in BigQuery. Learn to think in terms of ARRAYs.
- When you have dimension tables that are smaller than 10 gigabytes, keep them normalized. The exception to this is if the table rarely goes through UPDATE and DELETE operations.
- If you cannot define your schema in terms of nested, repeated fields, you have to make a decision on whether to keep the data in two tables or denormalize the tables into one big, flattened table. As a dataset's tables increase in size, the performance impact of a join increases. At some point, it can be better to denormalize your data. The crossover point is around 10 GB. If your tables are less than 10 GB, keep the tables separate and do a join.