Module 11

Schema Design and Nested Data Structures

In this module we will:

- **Compare Google BigQuery vs Traditional Relational Data Architecture**
- Normalization vs Denormalization: Performance Tradeoffs
- Working with Nested Data, Arrays, and Structs in Google BigQuery

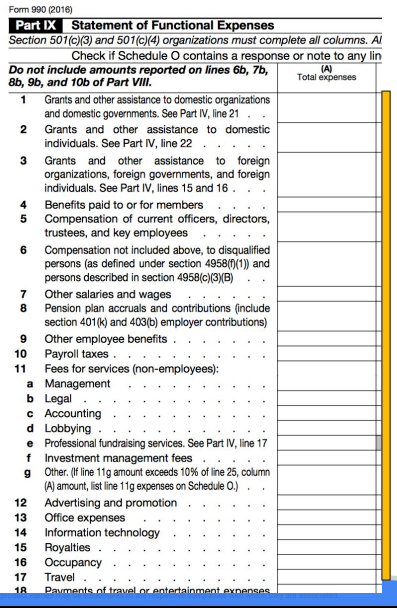
Google Cloud

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.

Let's Re-Examine our IRS Schema as an Architect



Form 990 (2016)

Part IX **Statement of Functional Expenses**

Section 501(c)(3) and 501(c)(4) organizations must complete all columns. Al

Check if Schedule O contains a response or note to any lin

	Do not include amounts reported on lines 6b, 7b, 8b, 9b, and 10b of Part VIII.	(A) Total expenses
1	Grants and other assistance to domestic organizations and domestic governments. See Part IV, line 21	
2	Grants and other assistance to domestic individuals. See Part IV, line 22	
3	Grants and other assistance to foreign organizations, foreign governments, and foreign individuals. See Part IV, lines 15 and 16	
4	Benefits paid to or for members	
5	Compensation of current officers, directors, trustees, and key employees	
6	Compensation not included above, to disqualified persons (as defined under section 4958(f)(1)) and persons described in section 4958(c)(3)(B)	
7	Other salaries and wages	
8	Pension plan accruals and contributions (include section 401(k) and 403(b) employer contributions)	
9	Other employee benefits	
10	Payroll taxes	
11	Fees for services (non-employees):	
a	Management	
b	Legal	
c	Accounting	
d	Lobbying	
e	Professional fundraising services. See Part IV, line 17	
f	Investment management fees	
g	Other. (If line 11g amount exceeds 10% of line 25, column (A) amount, list line 11g expenses on Schedule O.)	
12	Advertising and promotion	
13	Office expenses	
14	Information technology	
15	Royalties	
16	Occupancy	
17	Travel	
18	Payments of travel or entertainment expenses	

Each of these data fields needs to be stored in a structured way

Google Cloud

Page 10 of the IRS Form 990 lists 24 different expense types

What happens if I need to add another expense type? I need to change the schema and provide NULLs historically? Ugh..

Option 1: Add each Expense field as a New Column

Table Details: irs_990_2015

Schema | Details | Preview

yeebenef	payrolltx	feesforsrvcmgmt	legalfees	accntingfees	feesforsrvclobby	profndraising	feesforsrvcinvstmgmt	feesforsrvcothr	advrtpromo	officexpns	infotech	royaltsexpns	occupancy	travel
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
816623	847695	0	0	28654	0	0	0	27770	0	155715	43796	0	1156758	0
524396	539651	127071	34165	44264	0	0	0	0	567392	732920	875416	0	887599	33446
177305	209707	0	120	22000	0	0	0	11551	0	165306	11391	0	231092	0
1289799	543608	7415	14888	33514	0	0	0	0	1273856	2101383	217216	0	604569	40173
512540	170264	0	24000	64500	0	0	0	96660	344208	2128823	0	0	540746	0
217097	115324	0	0	0	0	0	0	0	0	0	0	0	0	0

Google Cloud

Page 10 of the IRS Form 990 lists 24 different expense types

What happens if I need to add another expense type? I need to change the schema and provide NULLs historically? Ugh..

Option 1: Add each Expense field as a New Column

Table Details: irs_990_2015

Schema | Details | Preview

yeebenef	payrolltx	feesforsrvcmgmt	legalfees	accntingfees	feesforsrvclobby	profndraising	feesforsrvcinvstmgmt	feesforsrvcothr	advrtpromo	officexpns	infotech	royaltsexpns	occupancy	travel
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
816623	847695	0	0	28654	0	0	0	27770	0	155715	43796	0	1156758	0
524396	539651	127071	34165	44264	0	0	0	0	567392	732920	875416	0	887599	33446
177305	209707	0	120	22000	0	0	0	11551	0	165306	11391	0	231092	0
1289799	543608	7415	14888	33514	0	0	0	0	1273856	2101383	217216	0	604569	40173
512540	170264	0	24000	64500	0	0	0	96660	344208	2128823	0	0	540746	0
217097	115324	0	0	0	0	0	0	0	0	0	0	0	0	0

… results in a really WIDE table that is **not scalable**...

Google Cloud

Page 10 of the IRS Form 990 lists 24 different expense types as of 2017

What happens if I need to add another expense type? I need to change the schema and provide NULLs historically? Ugh..

Option 2: Break Out Expenses into another Lookup Table

Organization Details

Company ID	Company Name
161218560	NY Association Inc.

Historical Transactions

Company ID	Expense Code	Amount
161218560	1	$10,000

Code Lookup Tables

Expense Code	Expense Type
1	Lobbying
2	Legal
3	Insurance

Google Cloud

Breaking apart one large table into multiple lookup tables is a common strategy in traditional database architecture.

Option 2: Break Out Expenses into another Lookup Table

Organization Details

Company ID	Company Name
161218560	NY Association Inc.

Code Lookup Tables

Expense Code	Expense Type
1	Lobbying
2	Legal
3	Insurance

Historical Transactions

Company ID	Expense Code	Amount
161218560	1	$10,000

… this breaking apart process is called **Normalization** ...

Google Cloud

This process is called normalization

Module 11

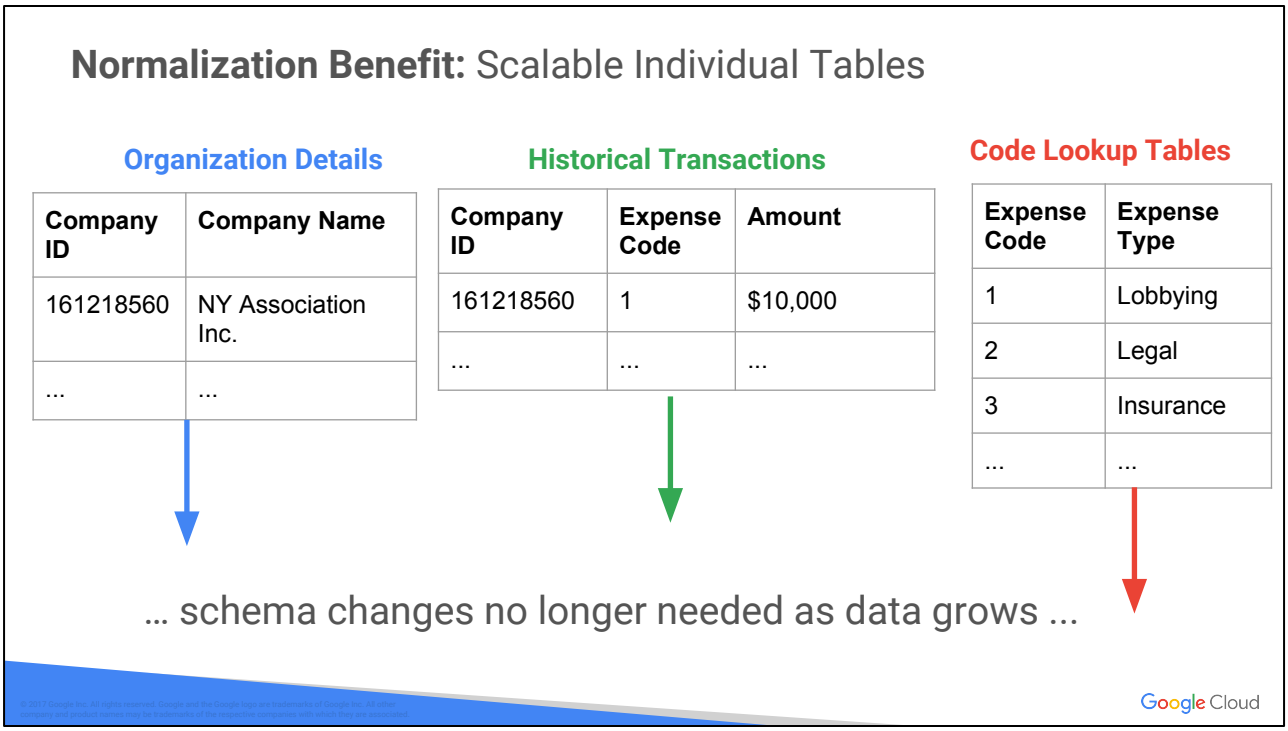
Schema Design and Nested Data Structures

In this module we will:

- Compare Google BigQuery vs Traditional Relational Data Architecture
- **Normalization vs Denormalization: Performance Tradeoffs**
- Working with Nested Data, Arrays, and Structs in Google BigQuery

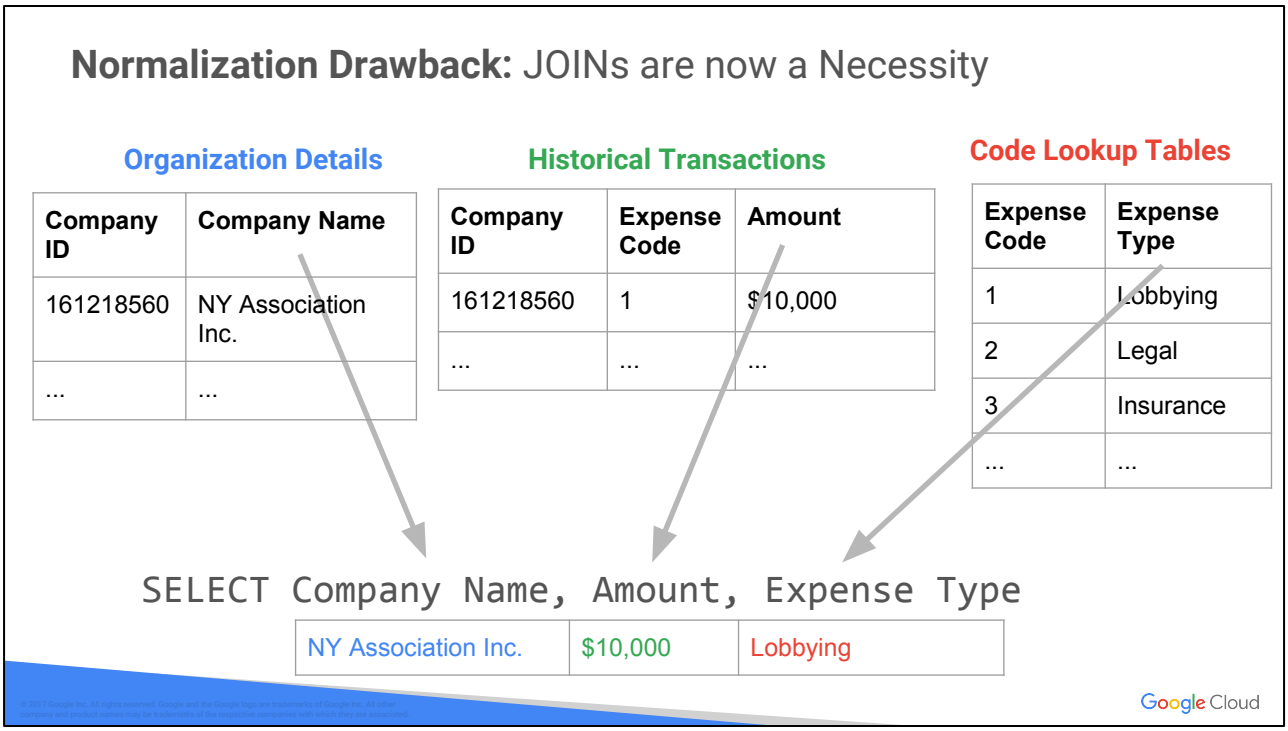
Google Cloud

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.

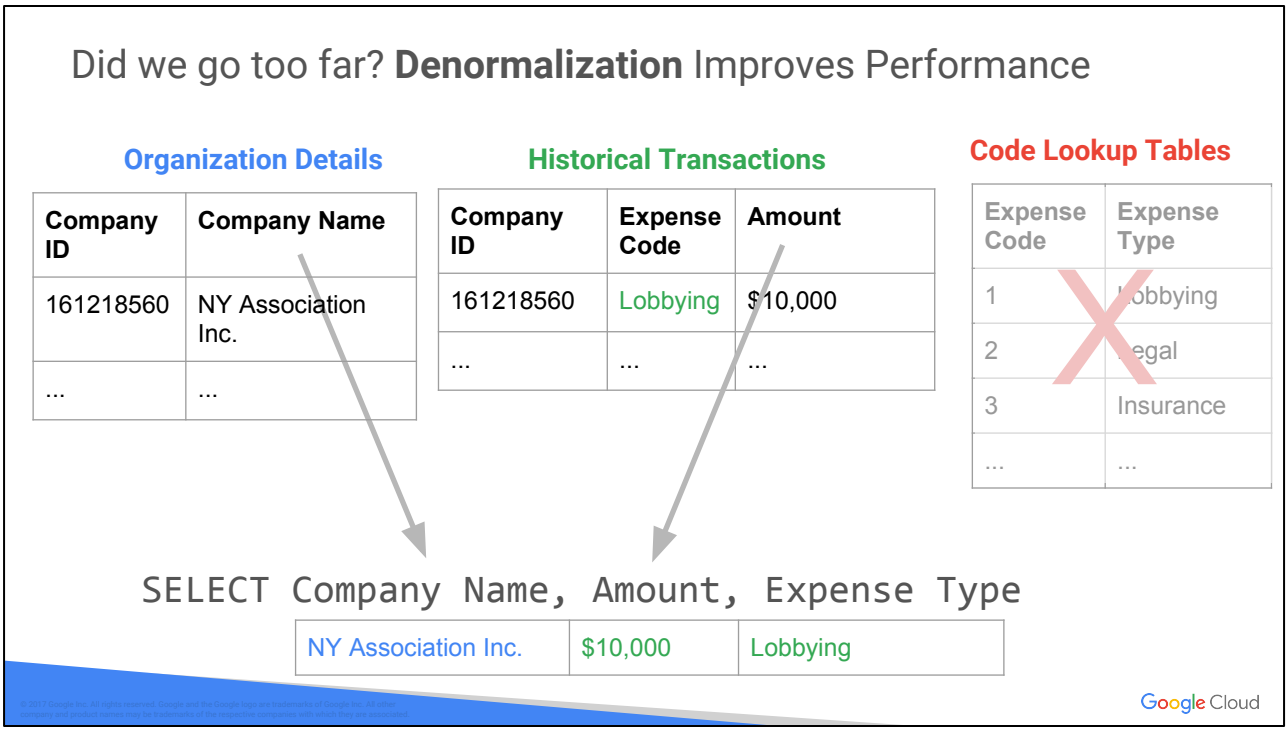


Page 10 of the IRS Form 990 lists 24 different expense types as of 2017

What happens if I need to add another expense type? I need to change the schema and provide NULLs historically? Ugh..



This is the birth of the Relational Model of Databases. Tables are now related to eachother through keys.



BigQuery prefers denormalized data (it handles data performance very differently as you will see soon when we discuss nested and repeated rows)
https://cloud.google.com/bigquery/docs/nested-repeated

Relational Databases at Scale?

How do traditional relational databases handle record growth at scale?

Google Cloud

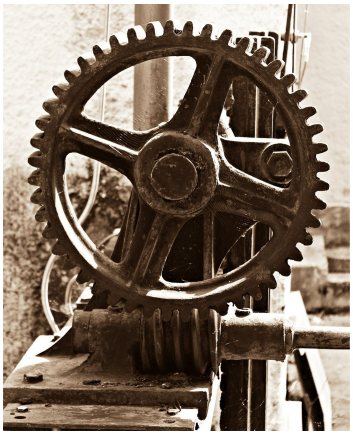
Image (tall buildings) cc0: https://pixabay.com/en/sky-building-tall-2097055/

Example: If you have a 100 floor building, are all floors equal? Is there more usage on one floor (lobby, reception, etc) that we could optimize for? This is similar for certain overused records in a database table.

Traditionally, Very Large Tables are **Hard to Scan and Compute**

Organization Details

Company ID	Company Name
161218560	NY Association Inc.
...	...
...	...
...	...
10 Billion Row Table	



```
SELECT Company Name ORDER BY Company Name
```


Google Cloud

Image (gears) cc0:
https://pixabay.com/en/gear-mechanics-wheels-transmission-408364/

Computer gears grinding through this one massive table to compute the sort over company name

Traditional: Pre-Sorted **Indexes** Introduced to Help Common Queries

Organization Details

Company ID	Company Name
161218560	NY Association Inc.
...	...
...	...
...	...
10 Billion Row Table	

Index

Company Name	Ranked Order
ACME Inc.	1
...	...
...	
NY Association Inc.	900,000
...	

Indexes do not exist in BigQuery because data is stored and handled in a fundamentally different way as you will see next...

```
SELECT Company Name ORDER BY Company Name
```

BigQuery Architecture Introduces Three Key Innovations

1. **Column-Based** Data Storage
2. **Break Apart Tables** into Pieces
3. Store **Nested Fields** within a Table

Google Cloud

BigQuery stores your table as small pieces (called shards) and stores each column individually. Also, it supports parent/child records nested within the very same table.

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

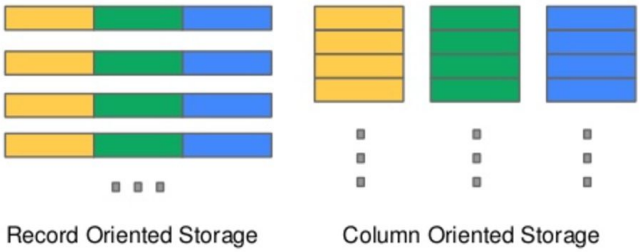
BigQuery Architecture Introduces Three Key Innovations

1. **Column-Based** Data Storage
2. **Break Apart Tables** into Pieces
3. Store **Nested Fields** within a Table

Google Cloud

Let’s first look at column-based storage

BigQuery **Column-Oriented Storage is Built for Speed**



- Storing related values (faster to loop through at execution time)
- Columns can be **individually** compressed
- Access values from a few columns without reading every one

Google Cloud

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 Architecture Introduces Three Key Innovations

1. **Column-Based** Data Storage
2. **Break Apart Tables** into Pieces
3. Store **Nested Fields** within a Table

Google Cloud

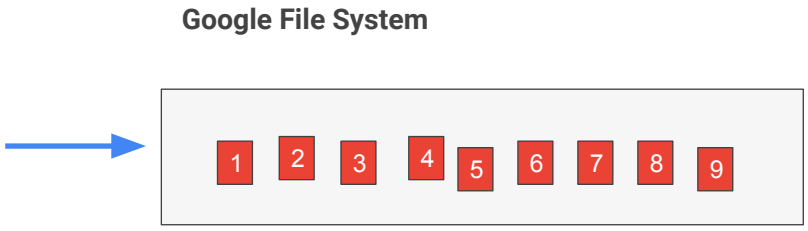
BigQuery data is stored in a massively distributed format. This is similar to MapReduce but faster due to the massive in-memory parallel execution and optimized shuffle steps.

Technical deep dive blog post:
https://cloud.google.com/blog/big-data/2016/08/in-memory-query-execution-in-google-bigquery

BigQuery Automatically **Breaks Apart Data into Smaller Shards**

Organization Details

Company ID	Company Name
161218560	NY Association Inc.
...	...
...	...
...	...
10 Billion Row Table	



Google Cloud

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.

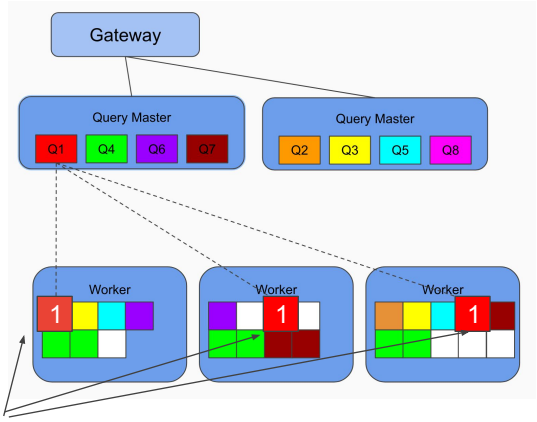
BigQuery auto-optimizes the balancing and partitioning of these data shards behind-the-scenes: https://cloud.google.com/blog/big-data/2016/05/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow

BigQuery Automatically **Pieces it All Back Together for Queries**

Organization Details

Company ID	Company Name
161218560	NY Association Inc.
...	...
...	...
...	...
10 Billion Row Table	

```
SELECT Company Name ORDER BY Company Name
```



Shards of data are read and Processed in Parallel

Google Cloud

Workers automatically scale up and scale down as needed based on query demand.

Gateway: Entry point for the query tree. BQ API servers communicate with the gateways. Gateways and query masters work to distribute query load.

Query Master: Responsible for building global query plan and individual work units. Workers execute individual units, but query master provides the steps to execute.

Scheduler: Responsible for fairness. Isn't directly part of query execution, but coordinates with query masters and workers to control what workers get assigned to which masters queries.

Workers: Tasks that run the individual units of query execution. Internally, these are also called shards. A worker may be responsible for a variable number of individual units of execution (slots).

How do workers talk to each other to share parts of data? (Answer: Shuffling)

BigQuery Automatically **Balances and Scales Workers**

- Up to 2,000 workers to process concurrent queries (on-demand tier)
- “Fairness model” for allocation

Google Cloud

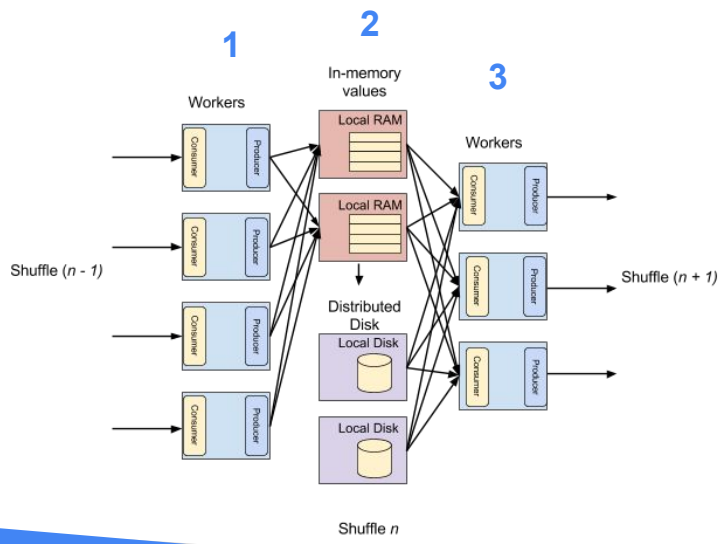
Image (cogs) cc0: https://pixabay.com/en/art-cogs-colorful-colourful-1866468/

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.

Read More

Google Cloud

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.

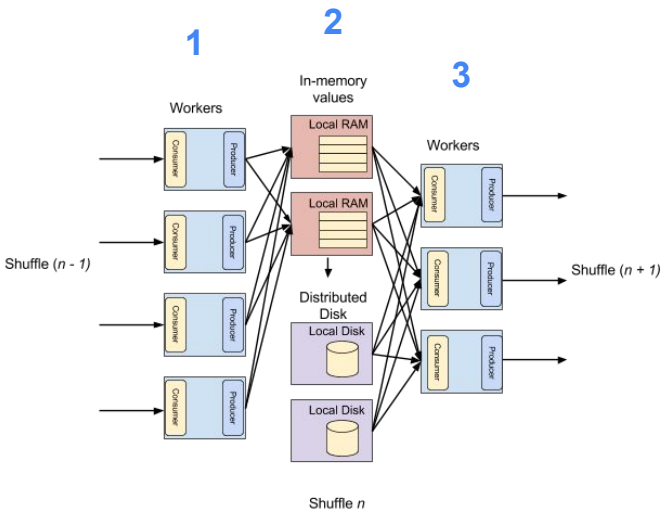
Projects typically get some "fair share" budget of fast, memory-backed shuffle resources tracked across all their active queries. When this is exhausted, shuffle traffic may transition to slower disk-backed methods.

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

Google Cloud

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

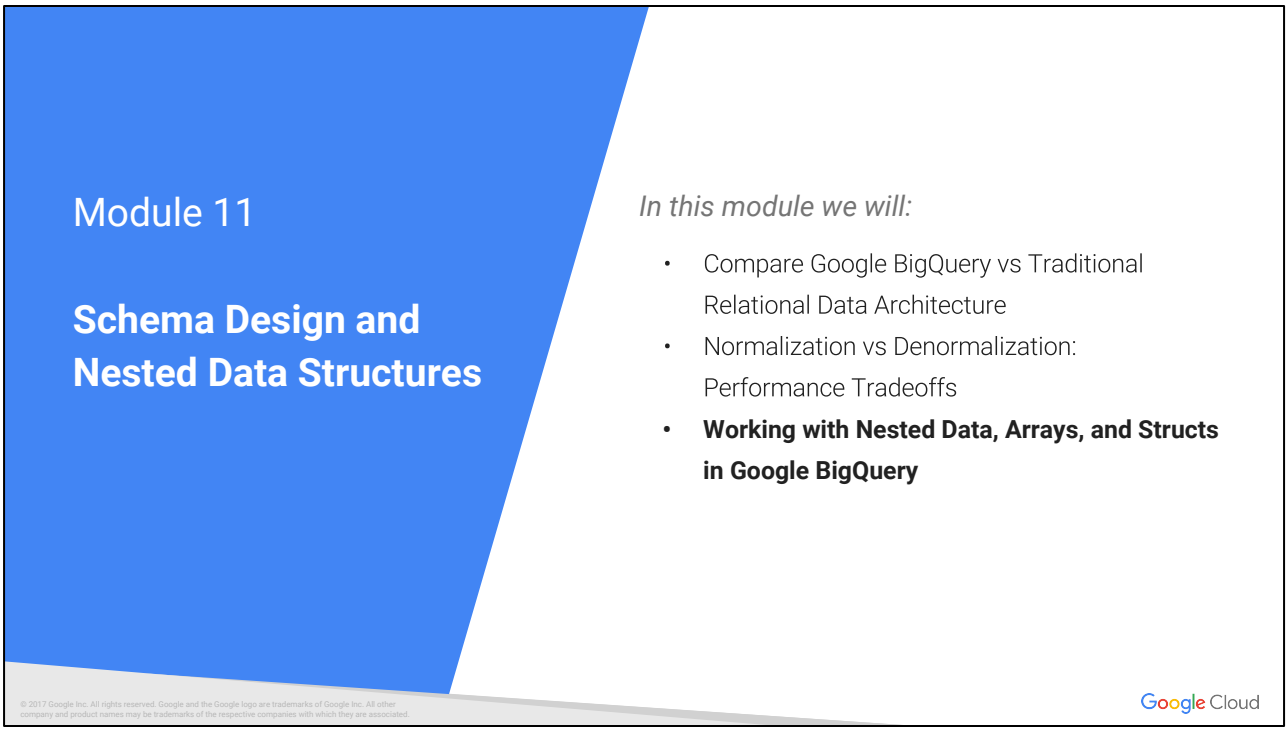
More on Shuffle:
https://medium.com/google-cloud/the-12-components-of-google-bigquery-c2b49829a7c7

BigQuery Architecture Introduces Three Key Innovations

1. **Column-Based** Data Storage
2. **Break Apart Tables** into Pieces
3. Store **Nested Fields** within a Table

Google Cloud

Lastly, we will discuss a table architecture that is likely new to a lot of data analysts: nested table rows.

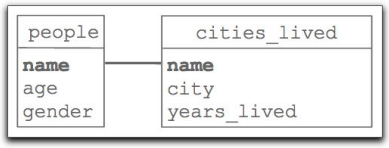


Wouldn't it be great if we can get all the performance advantages of having all our data in one single table and not have to do joins? And how about if we even could have lookup or drilldown details for records if we wanted it but could avoid the performance penalty of all those extra detail rows if we didn't need to query them? Well we're in luck. With the concept of nested and repeated records we can do exactly that.

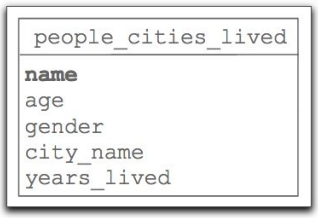
Let's see how it works conceptually and then practice what the SQL syntax looks like in BigQuery.

BigQuery Architecture Introduces **Repeated Fields**

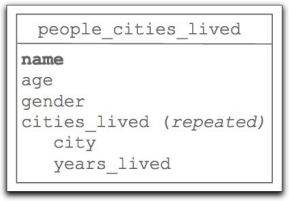
Normalized



Denormalized



Repeated



Less Performant → **High Performing**

Google Cloud

The Traditional Relational Model **Requires Expensive Joins**

Organization Details

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

Historical Transactions

Company ID	Expense Code	Amount
161218560	1	$10,000
...	...	...

Code Lookup Tables

Expense Code	Expense Type
1	Lobbying
2	Legal
3	Insurance
...	...

BigQuery Can Use **Nested Schemas** For Highly Scalable Queries

Organization Details with Nested Historical Transactions

NESTED

Company ID	Company Name	Transactions.Amount	Code.Expense
161218560	NY Association Inc.	$10.000	Lobbying
		$5,000	Legal
		$1,000	Insurance
123435560	ACME Co.	$7,000	Travel

Google Cloud

https://discourse.looker.com/t/why-nesting-is-so-cool/4182

- Avoids joins by already having the data matched in a single table, just UNNEST it
- Scans on Unnested data still fast COUNT(*) on Company ID

Nested Schemas Bring **Performance Benefits**

Organization Details with Nested Historical Transactions

Company ID	Company Name	Transactions.Amount	Code.Expense
161218560	NY Association Inc.	$10.000	Lobbying
		$5,000	Legal
		$1,000	Insurance
123435560	ACME Co.	$7,000	Travel

- Avoid costly joins
- No performance punishment for `SELECT(DISTINCT Company ID)`

Google Cloud

… but how do I query a nested data table?

Working with Repeated Fields

1. Introducing **Arrays and Structs**
2. **Flattening Arrays:** Legacy vs Standard
3. **Practicing SQL** with Repeated Fields

Google Cloud

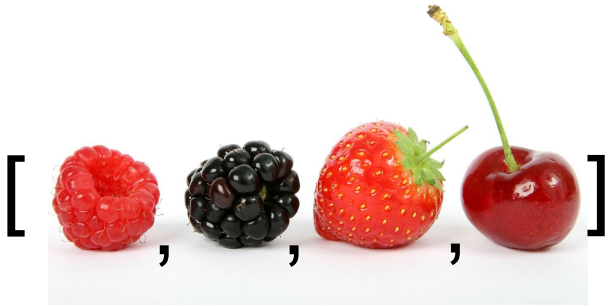
Working with Repeated Fields

1. Introducing **Arrays and Structs**
2. **Flattening Arrays:** Legacy vs Standard
3. **Practicing SQL** with Repeated Fields

Google Cloud

Arrays are Supported Natively in BigQuery

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



Google Cloud

Image (fruits) cc0: https://pixabay.com/en/berry-black-blackberry-blueberry-1238249/

An array of 4 fruits

Working with SQL Arrays

Create an array with brackets [ ]

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

BigQuery flattened output:

Row	fruit_array
1	raspberry
	blackberry
	strawberry
	cherry

Reminder: Use #standardSQL

Google Cloud

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

Working with SQL Arrays

```
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

Count the elements in an array with ARRAY_LENGTH

Google Cloud

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 Implicitly Flattens Arrays

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

Array = ['apple', 'pear', 'plum']

Flattened Array =
apple
pear
plum

BigQuery output:

- Item → Flattened array
- Customer → Normal field

Row	item	customer
1	apple	Jacob
	pear	
	plum	

Google Cloud

BigQuery Implicitly Flattens Arrays with #standardSQL

Explicitly Flatten Arrays with UNNEST()

```
SELECT
items,
customer_name
FROM
UNNEST(['apple', 'pear', 'peach']) AS items
CROSS JOIN
(SELECT 'Jacob' AS customer_name)
```

Associate all items in our array with the Customer

Flatten using a CROSS JOIN

BigQuery UNNESTED output:

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

UNNEST = A query that flattens an array and returns a row for each element in the array.

Google Cloud

To access array elements, we first need to unpack them before we can perfom 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

Aggregate into an Array with ARRAY_AGG

```
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;
```

Row	fruit
1	apple
2	pear
3	banana

← Subquery to create a table of fruits for us to aggregate later into an array

Row	fruit_basket
1	apple
	pear
	banana

Use ARRAY_AGG to aggregate values into an array

← These results are the same as saying: ["apple","pear","banana"]

Google Cloud

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

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

Sort Array Output with ORDER BY

```
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

← Notice how banana is now second

Google Cloud

You can sort items within array with ORDER BY

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

Filter Arrays using WHERE IN

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

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

← Start with a three arrays of shopping lists

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

Row	contains_apple
1	apple
	pear
	banana
2	carrot
	apple
3	

Use WHERE IN to filter an array. Note the empty third array returned back because 'apple' is not present in the original list

STRUCTs are Flexible Containers

STRUCT are a container of ordered fields each with a type (required) and field name (optional).

You can store multiple data types in a STRUCT (even Arrays!)

Google Cloud

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!

Image (cart) cc0:
https://pixabay.com/en/shopping-cart-healthy-shopping-fruit-2369146/

STRUCTs are Flexible Containers

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

Store age as an integer
Store name as a string

wait, what's wrong with the below result?

Row	f0_.age	f0_.name
1	35	Jacob

STRUCTs are Flexible Containers

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

Name the overall STRUCT container

Row	customers.age	customers.name
1	35	Jacob

One STRUCT can have many values. Looks and behaves similar to a table!

STRUCTs Can Even Contain ARRAY Values

STRUCTS can contain Arrays as 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

Google Cloud

Let's see how far we can take this ...

ARRAYS can Contain STRUCTs as Values

```
#standardSQL
SELECT
[
STRUCT(35 AS age, 'Jacob' AS name, ['apple', 'pear', 'peach'] AS items),
STRUCT(33 AS age, 'Miranda' AS name, ['water', 'pineapple', 'ice cream'] AS items)
] AS customers
```

ARRAYS can Contain STRUCTS as values

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

Filter for Customers who Bought Ice Cream

```
#standardSQL
WITH orders AS (
SELECT
[
STRUCT(35 AS age, 'Jacob' AS name, ['apple', 'pear', 'peach'] AS items),
STRUCT(33 AS age, 'Miranda' AS name, ['water', 'pineapple', 'ice cream'] AS items)
] AS customers
)

SELECT
  customers
FROM orders AS o
CROSS JOIN UNNEST(o.customers) AS customers
WHERE 'ice cream' IN UNNEST(customers.items)
```

CROSS JOIN and UNNEST Flattens arrays so we can access elements

Row	customers.age	customers.name	customers.items
1	33	Miranda	water
			pineapple
			ice cream

← Filter on items Array with UNNEST and using IN

Google Cloud

We have an array of STRUCT values ***called customers***

One of the values in the STRUCT is an array and is called ***items***

We flatten the array to using a CROSS JOIN and UNNEST()

The we filter for any array value in the items array that is ‘ice cream’

Nested (Repeated) Records are **Arrays of Structs**

- Nested records in BigQuery are Arrays of Structs.
- Instead of Joining with a sql_on: expression, **the join relationship is built into the table**.
- UNNESTing a ARRAY of STRUCTs is similar to joining a table.

Nested records in BigQuery are Arrays of Structs15. Instead of Joining with a sql_on: expression, the join relationship is built into the table. Other than that UNNESTing a ARRAY of STRUCTs is exactly like joining a table.

Working with Repeated Fields

1. Introducing **Arrays and Structs**
2. **Flattening Arrays:** Legacy vs Standard
3. **Practicing SQL** with Repeated Fields

Google Cloud

Legacy vs Standard SQL Repeated Record Differences

Legacy SQL Syntax
- Flattening happens explicitly with FLATTEN

Functions:
- WITHIN RECORD
- NEST

Standard SQL Syntax
- **Flattening happens implicitly** or explicitly with CROSS JOIN + UNNEST

Functions:
- ARRAY_LENGTH
- ARRAY_AGG

More Details

Google Cloud

Standard SQL Syntax
CROSS JOIN + UNNEST -- are actually optional, you can cross join with a comma and bigquery will implicitly unnest your array without UNNEST(). Good for readability however.
ARRAY_LENGTH -- use for counting child records instead of using deprecated WITHIN RECORD
ARRAY_AGG -- aggregates fields into an array instead of using deprecated NEST()

Legacy Syntax Notes:
FLATTEN -- you must explicitly flatten arrays in Legacy SQL, this is done implicitly in standard SQL
WITHIN RECORD -- Aggregates within repeated children. You would now use arr[SAFE_ORDINAL(index)] in Standard SQL (see: https://cloud.google.com/bigquery/docs/reference/standard-sql/migrating-from-legacy-sql#differences_in_repeated_field_handling)
NEST -- aggregates a field into an array. Replaced by ARRAY_AGG

https://cloud.google.com/bigquery/docs/legacy-nested-repeated#within
https://cloud.google.com/bigquery/docs/reference/standard-sql/migrating-from-legacy-sql#differences_in_repeated_field_handling

Working with Repeated Fields

1. Introducing **Arrays and Structs**
2. **Flattening Arrays:** Legacy vs Standard
3. **Practicing SQL** with Repeated Fields

Google Cloud

ARRAY/STRUCT example

```
# Top two Hacker News articles by day
WITH TitlesAndScores AS (
  SELECT
    ARRAY_AGG(STRUCT(title, score)) AS titles,
    EXTRACT(DATE FROM time_ts) AS date
  FROM `bigquery-public-data.hacker_news.stories`
  WHERE score IS NOT NULL AND title IS NOT NULL
  GROUP BY date)

SELECT date,
  ARRAY(SELECT AS STRUCT title, score
        FROM UNNEST(titles) ORDER BY score DESC
        LIMIT 2)
  AS top_articles
FROM TitlesAndScores;
```

WITH Clause:
- Make an array of (title, score) objects
- Extract the date from the timestamp
- Group by the date (which gives us the array contents)

ARRAY(SELECT AS STRUCT:
- Unnest the array from the WITH clause
- Order it and take the top 2
- Create a new array of (title, score) objects

Outer query:
- Project date from WITH clause
- Project Array

Google Cloud

With standard SQL, you can now store data in arrays natively. In Legacy SQL, you had to store a list of values in a record and then use string concatenation.

You can also store STRUCTs natively. Previously, you had to store two pieces of data as one record and then use string concatenation.

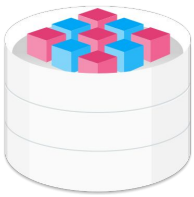
ARRAY/STRUCT example result

Row	date	top_articles.title	top_articles.score
1	2010-08-23	Why GNU grep is Fast	512
		Readme Driven Development	244
2	2010-04-26	Police raid Gizmodo editor's house	257
		Not even in South Park?	257
3	2009-09-15	Learning Advanced JavaScript	257
		Sub-pixel re-workings of YouTube and BBC favicons	154

Google Cloud

Results from previous query

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

Google Cloud

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.

Image (data blocks) cc0: https://cloud.google.com/data-transfer/
Image (blocks in a cylinder) cc0: https://cloud.google.com/products/storage/

Lab 10
Querying Nested and Repeated Data

Google Cloud

Lab 10 in Qwiklabs

Querying Nested and Repeated Data

In this lab, you will practice querying Nested and Repeated Fields using array manipulation and structs.

Results | Explanation | Job Information

Row	ein	name	expense_struct.type	expense_struct.amount	revenue_struct.type	revenue_struct.amount
1	510203813	IF	Lobbying	0	Contributions	110796
			Legal	0	Programs	0
			Insurance	250	Fundraising	0
			Travel	0		
			Ads Promotion	180		
			Office	9147		
2	364236669	ARF	Lobbying	0	Contributions	151818
			Legal	0	Programs	0
			Insurance	0	Fundraising	817
			Travel	0		
			Ads Promotion	5859		
			Office	16497		

Google Cloud