1. Google Cloud Platform . . . 2
1.1 Google SDK . . . 2
1.2 Billing . . . 2
1.3 Roles & Permissions . . . 2
1.4 Component Fundamentals . . . 2
1.5 Google Cloud Storage . . . 4
1.6 Design . . . 6
1.6.1 Case Study - Great Inside . . . 6
1.6.1.1 Compute . . . 7
1.6.1.2 Storage . . . 9
1.6.1.3 Database . . . 10
1.6.1.4 Auto Scaling Group . . . 11
1.6.1.5 Virtual Private Cloud Network . . . 12
1.6.1.6 Authentication . . . 14
1.6.1.7 Roles . . . 15
1.6.1.8 Service Accounts . . . 16
1.6.1.9 Data Protection and Encryption . . . 16
1.6.1.10 Legislation and Compliance . . . 17
1.6.1.11 Disaster Recovery . . . 18
1.7 Google Kubernetes Engine and App Engine . . . 19
1.7.1 Managing Kubernetes Engine Resources . . . 19
1.7.2 Demo Configuration: GKE . . . 20
1.7.3 Configuring Application Traffic . . . 20
1.8 Big Data . . . 21
1.8.1 Bigtable . . . 21
1.8.2 Big Query . . . 27
1.8.2.1 Introduction . . . 28
1.8.2.2 Core Concepts Overview . . . 29
1.8.2.3 Optimizing . . . 30
1.8.2.4 Access Control . . . 38
1.9 Creating Basic App Engine App . . . 40
1.10 Pub Sub Basics . . . 40
1.11 Data Flow . . . 41
2. Job Application CV . . . 42
2.1 September 2022 . . . 42
3. Migration . . . 42
3.1 On Prem Hadoop Cluster to GCP . . . 42
4. Effective Reading . . . 45
4.1 Books on Database Technologies . . . 45
4.2 Orchestration . . . 45
4.3 Data Engineering Design . . . 45
4.4 Advisory & Strategy . . . 46
5. Data Warehousing . . . 46
5.1 Design Patterns . . . 46
5.2 Data Modeling . . . 47
5.2.1 Modeling using Data Build Tool . . . 47
5.2.2 Data Vault 2.0 . . . 47
5.2.3 Slowly Changing Dimension . . . 47
5.2.4 Semi Structured & NoSQL Data Modeling . . . 47
5.2.5 Dimensional Modeling . . . 48
6. Open Source Tools . . . 48
6.1 Apache Beam . . . 48
6.2 Airflow . . . 48
7. Data Build Tool (DBT) . . . 51
7.1 DBT Certification Quick Reads . . . 51
8. Containerization . . . 51
9. Advisory [ Governance & Strategy ] . . . 51
10. Snowflake - Cloud DWH . . . 51
10.1 Good Reads . . . 51
11. Databricks . . . 51
12. SQL Practice . . . 51

# Google Cloud Platform

## Google SDK

- Allows interaction to GCP with commands
- Allow to create script for repeatable tasks
- ***glcoud, gsutil, bq***
- To set project specific configurations for multiple configurations, use *gcloud config set*

```
daniel_mease@cloudshell:~ (daniel-sandbox-305920)$ gcloud config configurations create project1-config
Created [project1-config].
Activated [project1-config].
daniel_mease@cloudshell:~$ gcloud config configurations create project2-config
Created [project2-config].
Activated [project2-config].
daniel_mease@cloudshell:~$ gcloud config configurations list
NAME             IS_ACTIVE  ACCOUNT  PROJECT                COMPUTE_DEFAULT_ZONE  COMPUTE_DEFAULT_REGION
cloudshell-11170  False               daniel-sandbox-305920  us-east1-b
project1-config   False
project2-config   True
daniel_mease@cloudshell:~$ gcloud config configurations activate project1-config
Activated [project1-config].
daniel_mease@cloudshell:~$ gcloud config configurations list
NAME             IS_ACTIVE  ACCOUNT  PROJECT                COMPUTE_DEFAULT_ZONE  COMPUTE_DEFAULT_REGION
cloudshell-11170  False               daniel-sandbox-305920  us-east1-b
project1-config   True
project2-config   False
daniel_mease@cloudshell:~$ gcloud config set project daniel-sandbox-305920
Updated property [core/project].
```

- **As a developer, I want to interact with GCP via gcloud**
  `gcloud auth login`
- **As a developer, I want my code to interact with GCP via SDK**
  `gcloud auth application-default login`
- **Initialize or reinitialize gcloud / authorize access to your account**
  `gcloud init`
- **Cloud Shell:** online terminal built right into the Google Cloud console

## Billing

- Free Tier available but eventually needs to be paid
- Project can be modified to use a different billing account anytime
- Billing can also be disabled for a project, but the amounts would be overdue
- To stop spending delete the project
- Alert for spending can be set up based on a budget
- Alerts can be sent based on different thresholds on the budget
- Alerts won't stop the resource from utilizing GCP resources
- Billing export are detailed reports of charges stored in Big Query
- Billing export may contain daily cost details or pricing on a project
- The export will be put into a big query table, this can be visualized into a reporting tool like Data Studio

## Roles & Permissions

## Component Fundamentals

-- serverless --
CLOUD FUNCTIONS - Events/Serverless
APP ENGINE - Container
CLOUD RUN - Container
GOOGLE KUBERNETES ENGINE - Many Containers

-- storage --
Cloud Storage - Standard(F), Near Line (1m), ColdLine(3m), Archive(1y) - Immidiate access irrespective
Cloud Filestore - Extra Volume Mounts (NFS)
CLoud SQL - RDBS : MYSQL, POSTGRES, SQL SERVER /OLTP
NOSQL : BigTable, FireStore, MemoryStore
CLOUD SPANNER : Scalable RDBMS (Expensive)
BIGQUERY : Data Warehouse/OLAP

-- network --
VPC:
- virtual network
- each resource has private IP
- Explicitly attach public IP to access from the internet

- Machines in same VPC can communicate each other
- Configure [VPC ROUTER] to see how everyonce can communicate each other via subnets
- Communication between multiple VPCs has to be done via [VPC NETWORK PEERING]
- Secure connection between VPC and on PREM network via [CLOUD VPN/ CLOUD INTERCONNECT/ PEERING]
- [CLOUD VPN] routes the traffic via public internet
- [INTERCONNECT and PEERING] are private connections between VPC & ON-PREM network
- [CLOUD INTERCONNECT] is very Expensive
- [PEERING] is not well integrated in GCP. Avoid it.

CDN:
- Caches content on Googles Global Network, No matter where you are located
- Fast content delivery, important for videos

CLOUD LOAD BALANCING:
- Distributes traffic into Compute engines located at different location
- Single IP address

CLOUD DNS : #TODO
GLOBAL LOAD BALANCER : #TODO
INTERNAL LOAD BALANCER : #TODO

-- security --
CLOUD ARMOR : Protects Cloud Load Balancer from DDOS attacks
IAM : Provides roles and Permissions
KMS : Store & encrypt your keys
SECRET MANAGER : Central Place to store passwords, certificates and API keys
DATA LOSS PREVENTION : Helps to mask and remove PII data such as credit card numbers
SECURITY COMMAND CENTRE:
- Gathers security and compliance information in one Place
- Overview Dashboard: Shows active threat and vulnerabilities
- Compliance Dashboard: Shows violation of compliances

-- operations suite --
CLOUD MONITORING : Monitor all the events and set up alerts
CLOUD LOGGING : Logs all events
EROOR reporting : #TODO
CLOUD TRACE : #TODO
CLOUD DEBUGGER : #TODO
CLOUD PROFILER : #TODO

-- compute-engine --
1 region : 3 zones ideally. Cant Change later
PRE-EMPTIBLE ENGINES:
- Can stop anytime , good for analytics payload which is not critical.
- Only Run for 24 hours
- costs 70% less

--ingest--
PUBSUB : Stream
BIG QUERY : Batch Analytics

-- process --
DATAPROC : SPARK
DATAFLOW : BEAM (Better than Hadoop)
DATAPREP : NoCode (DATA Flow underneath)

-- reporting --
LOOKER
DATA STUDIO

-- pipeline --
CLOUD COMPOSER : Managed AIRFLOW

-- IOT --
IOT Core - Ingest Data from IOT devices

-- DevOps --
CLOUD BUILD : CI-CD Pipeline + Github or BitBucket
ATIFCACT REGISTRY : Container registry inside google Cloud
CLOUD SOURCE REPOSITORY : Private GIT Repository inside GCP
GKE : Kubernetes on GCP using Containers from GCR.IO

-- Migration --
MIGRATE FOR ANTHOS : Migrate VMs connected in other cloud into GCP as containerized PODs on Kubernetes.
VMWARE ENGINE : #TODO

MIGRATE FOR COMPUTE ENGINE : #TODO
TRANSFER APPLIANCE : #TODO

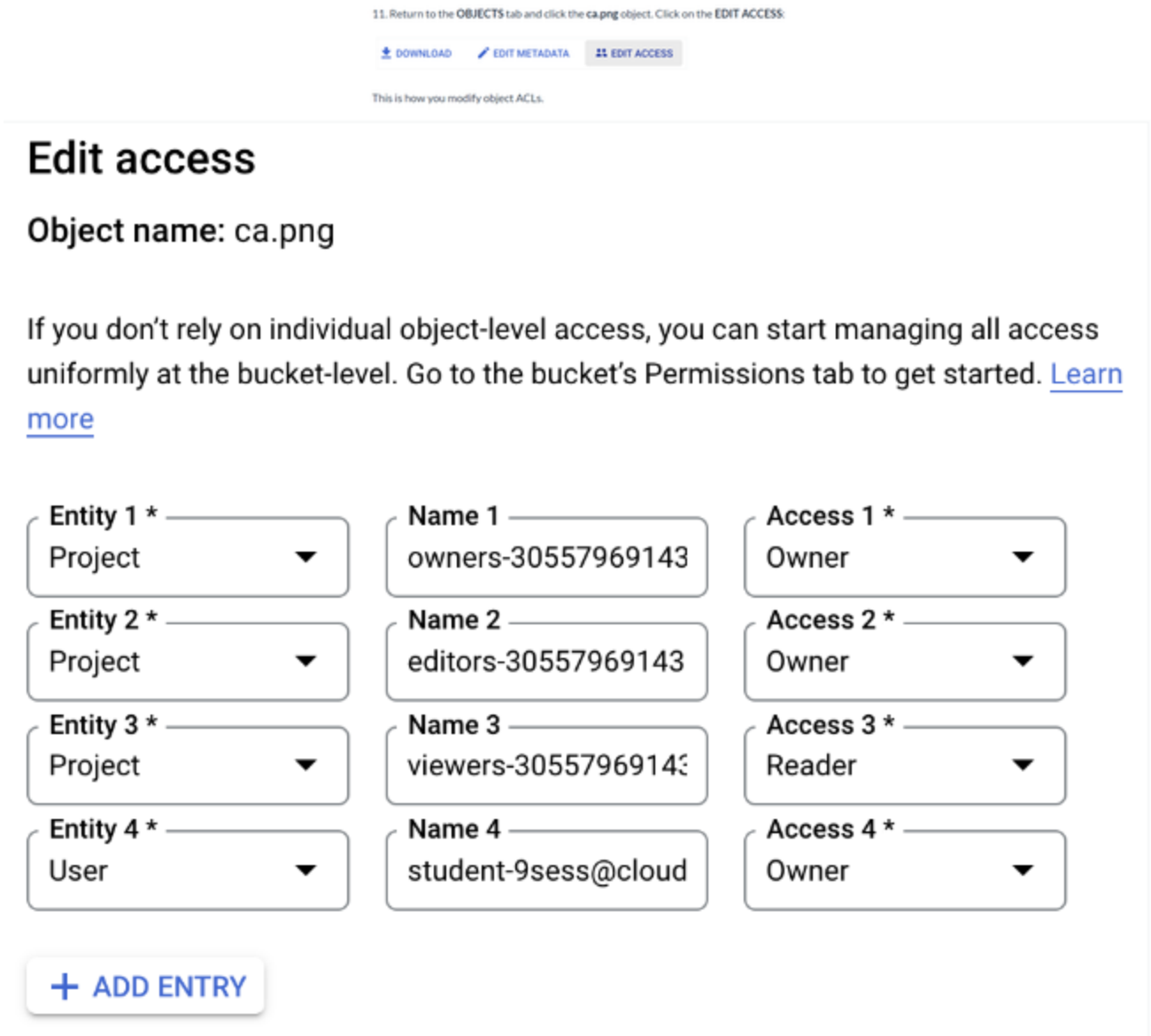
-- Deployment --
CLOUD DEPLOYMENT MANAGER : Similar to Terraform , Infra as Code

-- IAM --
GOOGLE CLOUD IDENTITY : Clones any existing LDAP or ACTIVE DIRECTORY from another setup
PERMISSION >> ROLES >> USERS
BASIC (PRIMITIVE) & PREDEFINED ROLES + CUSTOM ROLES
BASIC ROLES : Browser, View, Editor, Owner
Avoid BASIC roles! ( Specially in PROD Env )

## Google Cloud Storage

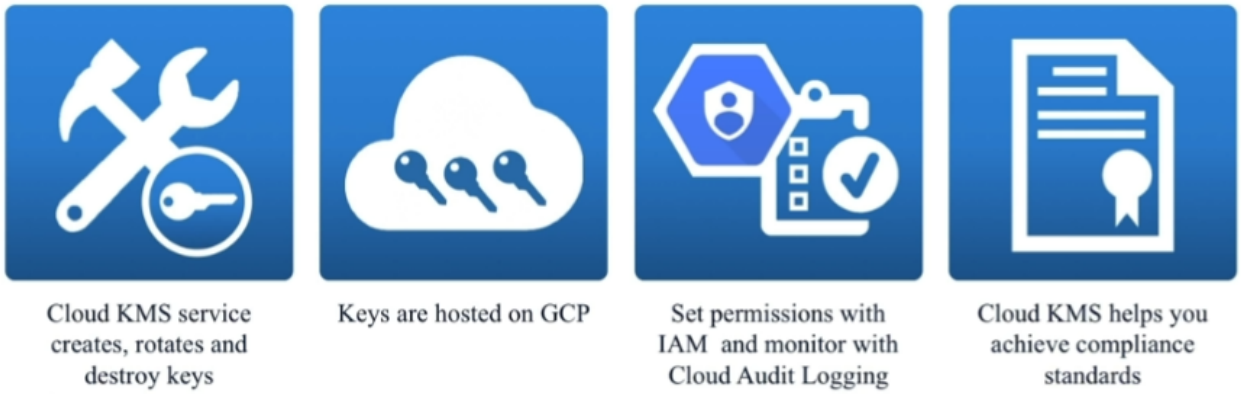
- Google Cloud's enterprise-ready storage service for unstructured data
- Buckets are not hierarchical, meaning that you can't store a bucket inside of another bucket as you can with folders on your local file system
- Labels can be used while resource creation (buckets in this case) to optimize selection
- **Location type**
  - **Region**: Data stored in a single region, but in multiple availability zones, for the second-best availability and lower storage cost than multi-regional
  - **Multi-Region**: Data is stored in multiple geographic regions giving the highest availability but also the highest storage cost
  - **Dual-Region**: Data stored in two different regions and in multiple availability zones
- For the best performance, you should choose a location that is near to the data's end users. Keep in mind different locations have varying costs, which should also be considered when storing very large amounts of data. When using the multi-regional storage class, there are only a few location choices because the data is replicated across several regions with the broader location you select
- **Storage Class**
  - **Nearline** and **Coldline** should be considered for infrequently accessed data
  - **Standard** (default) has the highest cost of storage but the lowest cost for operations and data retrieval is free which is not true for the other classes
  - **Archive** is a new storage class whose purpose is to maintain data for a long period (at least 365 days) and it is a good choice when you rarely access your data
- The default storage class can be overridden for individual objects when necessary
- **You can change a bucket's default storage class, but that does not modify the storage class of objects already in the bucket**
- **Object Lifecycle Management** provides a way to automatically change an object's storage class, such as for automatically *archiving* files by changing the storage class from multi-regional to *coldline* after three months. **Object Lifecycle Management applies to all objects in a bucket and cannot be targeted at a subset of objects in a bucket**
- **Access Control**
  - **Uniform (recommended)**: Uniform bucket-level access allows you to use Identity and Access Management (IAM) alone to manage permissions. IAM applies permissions to all the objects contained inside the bucket or groups of objects with common name prefixes. IAM also allows you to use features that are not available when working with ACLs, such as IAM Conditions and Cloud Audit Logs. This is the recommended way to control access to Cloud Storage, although it cannot handle all use cases, such as granting access to a single object in a bucket. Instead, it applies equally to all objects in a bucket
  - **Fine-grained**: The fine-grained option enables you to use IAM and Access Control Lists (ACLs) together to manage permissions. ACLs are a legacy access control system for Cloud Storage designed for interoperability with Amazon S3. You can specify access and apply permissions at both the bucket level and **per individual object.** Objects and buckets automatically have predefined ACLs applied to them, but you can adjust the ACLs if the default does not suit your requirements
  - **Public access prevention** protects Cloud Storage buckets and objects from being accidentally exposed to the public. When you enforce public access prevention, no one can make data in applicable buckets public through IAM policies or ACL. When public access prevention **no longer applies** for a resource, the following occurs:
    - Existing IAM policies and ACLs that grant access to `allUsers` and `allAuthenticatedUsers` take effect and make data accessible to the public
    - An object created under public access prevention without public ACLs may become accessible to the public if it was created in a publicly accessible bucket
    - You can disable public access prevention for a project, folder, or organization at any time**. Buckets with an** `enforced` **setting continue to have public access prevention enforced, even if you disable it for a project, folder, or organization that contains the bucket.**
    - When you enforce public access prevention on existing resources, all existing authorization and new additions of `allUsers` and `allAuthenticatedUsers` are blocked. If an application depends on `allUsers` and `allAuthenticatedUsers` to access your data or create public resources, enabling public access prevention can break the application.
  - Owners and editors of the GCP project containing the bucket are assigned the **Storage Legacy Bucket Owner** role
  - Viewers of the project are assigned the **Storage Legacy Bucket Reader** role
  - Two additional groups are
    - **All Users** - This grants public access
    - **All Authenticated Users** - This means all users logged in with Google account
  - The **Project** entries give object read and metadata write (**Owner**) access to owners and editors of the project, and object read-only (**Reader**) access to viewers of the project. *The last entry by default grants owner access to the user that created the object*

11. Return to the **OBJECTS** tab and click the **ca.png** object. Click on the **EDIT ACCESS**:




- **Metadata** - The pieces of data that are stored in buckets are called objects. Objects are similar to files, but they include both the data/file and metadata. The metadata is key-value pairs, similar to bucket labels. **Metadata** consists of
  - **Version/generation number** *when versioning is enabled*
  - Custom metadata that you can set for use in your applications
  - The **Content-Type** is automatically populated by Cloud Storage when you uploaded the file. The value is based on the extension of the file you upload
  - **Cache-Control** tells a browser if and how long to cache the object
- **Encryption -** Cloud Storage encrypts user data at rest using AES-256. There is no setup or configuration required, no need to modify the way you access the service, and no visible performance impact. Data is automatically decrypted when read by an authorized user.
  First, the data is broken into subfile chunks, and each chunk is encrypted with an individual data encryption key (or DEK). These keys are stored near the data to ensure low latency and high availability. The DEKs are then encrypted with a key encryption key (or KEK). The keys are AES-256 symmetric encryption keys.

  **Types of Encryptions:**
  - *Server-side encryption* - encryption that occurs after Cloud Storage receives your data, but before the data is written to disk and stored
    - **Google Managed Encryption Keys -** Cloud Storage always encrypts your data on the server side, before it is written to disk, at no additional charge
    - **Customer Managed Encryption Keys -** You can create and manage your encryption keys through **Cloud Key Management Service**




- **Customer Supplied Encryption Keys -** You can create and manage your own encryption keys. These keys act as an additional encryption layer on top of the standard Cloud Storage encryption. Google doesn't store your key

- *Client-side encryption* - encryption that occurs before data is sent to Cloud Storage. Such data arrives at Cloud Storage already encrypted **but also undergoes server-side encryption**
- *Encryption in flight* - All of the Cloud Platform services are accessible only by API (even when you're using other methods, such as the Cloud Console or the gcloud command, they're making API calls under the hood). And all API communication is encrypted using SSL/TLS channels. Furthermore, every request has to include a time-limited authentication token, so the token can't be used by an attacker after it expires

## Design

### Case Study - Great Inside

Suppose you've been hired to help a company called Great Inside, which offers interior design software as a service.

Great Inside makes its money by selling subscriptions to its web-based interior design application. It also has a free version that's supported by advertising. **Their customers are primarily in North America, but they hope to expand in Europe and Asia at some point in the future.**

The company has grown slowly for five years, but recently closed a venture capital round, brought in experienced executives, and is **now growing more quickly**. The company's existing infrastructure is not capable of scaling up quickly enough, so **they would like to move to the cloud**.

Great Inside started off with a Microsoft-centric infrastructure and then migrated to a LAMP stack. The only Microsoft infrastructure left is the payment processing system and an Active Directory server. **They would like to retire their Microsoft servers in the future, other than Active Directory**. But that isn't a priority right now, and the company would like to move both types of servers to the cloud. They've also started a pilot project using a NoSQL database.

Since they accept credit cards, they need to be PCI DSS compliant. Since their volume is increasing, they need to ensure that their payment processing environment meets a higher level of compliance. Note that Great Inside passes the validation and processing of credit card information to a certified payment processor.

They would like to improve their **disaster recovery solution**. At the moment, they're backing their data up to a cloud service, but it would take them a long time to recover from a disaster. Their existing technical environment is all in a single data center.

They have three types of databases. **MySQL for the interior design application, Microsoft SQL Server for payment processing, and a NoSQL database in the development environment**. They have two types of web and application servers. **Apache and Tomcat are running on six servers, each with 2 dual-core CPUs, 24GB of RAM, and two mirrored 200GB disks.** These servers are for their interior design

application. **IIS is running on four servers- two customer-facing and two internal,** each with a dual-core CPU, 16GB of RAM and two mirrored 250GB disks. These servers are for payment processing.

They have a variety of infrastructure servers, including **Active Directory and a file server for internal documents**, etc.

Here are their **business requirements**. Scale easily to handle rapid growth, move as much of the development, test, and production infrastructure as possible to the cloud, and increase performance, reliability, and security while reducing management overhead.

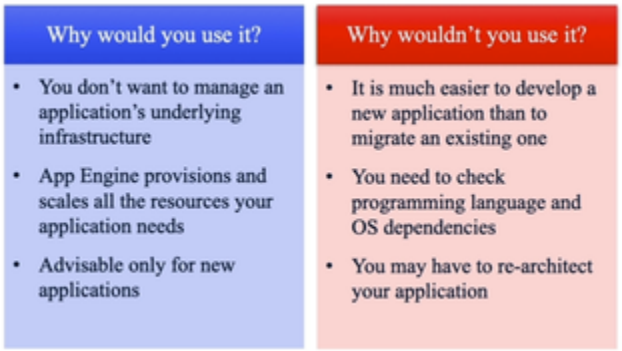
And their **technical requirements** are as follows: **connect the data center's network with the cloud environment's network, encrypt all data, design high availability into all tiers, and create a disaster recovery solution** that will reduce recovery time to a few hours, rather than a day.

---

**Compute**

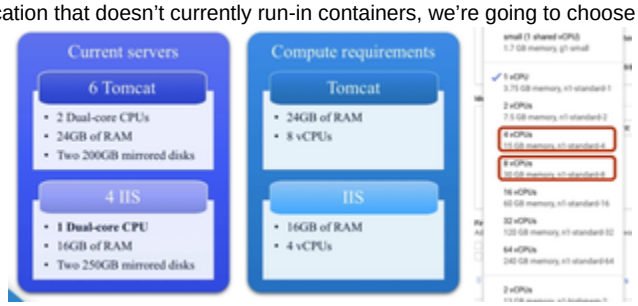
Although most Google Cloud designs include virtual machine instances, that doesn't mean VMs are your only option for compute resources. Before you start designing a solution using only Compute Engine instances, you should consider **App Engine** and **Kubernetes Engine**.

- **App Engine**




  App Engine provisions and scales all of the resources your application needs behind the scenes, without any human intervention required.
  App Engine doesn't support windows.
  App Engine is designed for microservices-based apps, so if your existing application has a monolithic architecture, then it might require some work to migrate it. For all of these reasons, it's usually advisable to use App Engine only for new applications rather than existing ones.
- **Kubernetes Engine**

- **Compute Engine** - Although Compute Engine requires more management than App Engine or Kubernetes Engine, it does give you ultimate flexibility. For example, you could run an application that requires Windows, a specific network driver, and high-performance GPUs. Since our case study involves an existing application that doesn't currently run-in containers, we're going to choose Compute Engine for our design




- **Discounts -**
  - **Committed-Use Discounts -** If you know that you're going to be running an instance continuously for a long period of time, then you can pay much less by purchasing either a one-year or three-year contract, which is called a committed-use contract. This will typically reduce the cost by up to 57%
  - **Sustained-Use Discounts -** However, that's a pretty big commitment, so Google provides a way to reduce costs without signing a long-term contract. You start getting an automatic discount after an instance runs for more than 25% of a month, and the discount increases the longer the instance runs during that month. For most machine types, you'll receive a sustained-use discount of 30% if you run the instance for the entire month
- **Windows Licensing -**
  - **Google's pay-as-you-go Windows licensing**
    - *Create a new instance with one of the pre-configured Windows Server boot disks*

| | | | |
|---|---|---|---|
| windows-server-2008-r2-dc-v20170417 | 50 GB | Microsoft | windows-2008-r2 |
| windows-server-2012-r2-dc-core-v20170417 | 50 GB | Microsoft | windows-2012-r2-core |
| windows-server-2012-r2-dc-v20170417 | 50 GB | Microsoft | windows-2012-r2 |
| windows-server-2016-dc-core-v20170417 | 50 GB | Microsoft | windows-2016-core |
| windows-server-2016-dc-v20170417 | 50 GB | Microsoft | windows-2016 |

    - *Import a Windows VM* - There are 2 options to import a VM
      - Import a virtual disk and turn it into an image that you can use to create a Compute Engine instance
      - **Cloud Migrate for Compute Engine** - This service makes replicas of existing VMs you have running on-premises or on another cloud platform. It will take care of the many steps that are needed to migrate important applications.
  - **Bring your own license -** you can run your Windows VMs on sole-tenant nodes, which are dedicated physical servers that are not shared with other customers. If you need to run any Microsoft applications, then you'll need licenses for those too, of course, but Microsoft is more flexible with its application licensing than with Windows licensing. If your organization has active Software Assurance contracts for its Microsoft applications, then you can move those licenses to either Compute Engine instances or sole-tenant nodes
- **SQL Server Licensing -**
  - Windows Licensing mentioned above (Bring your own license)
  - Create instances with pre-configured SQL Server boot disks
  - **Cloud SQL**
- **Service Account**

### Compute Engine default service account

New projects that have enabled the Compute Engine API have a Compute Engine default service account, which has the following email:

```
PROJECT_NUMBER-compute@developer.gserviceaccount.com
```

Google creates the Compute Engine default service account and adds it to your project automatically but you have full control over the account.

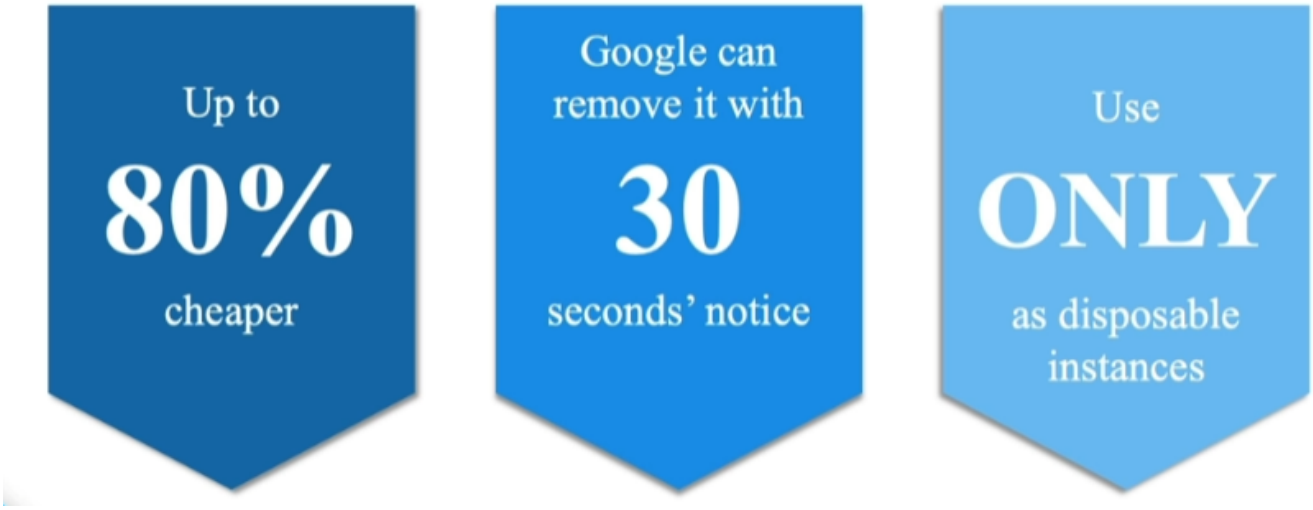
The Compute Engine default service account is created with the IAM basic Editor role, but you can modify your service account's roles to control the service account's access to Google APIs.

You can disable or delete this service account from your project, but doing so might cause any applications that depend on the service account's credentials to fail. If you accidentally delete the Compute Engine default service account, you can try to recover the account within 30 days. For more information, see Creating and managing service accounts.

In summary, the Compute Engine default service account has the following attributes:

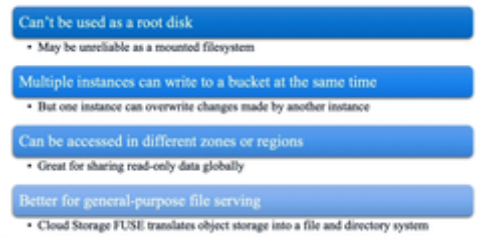
- Automatically created, with an autogenerated name and email address, and added to your project when you enable the Compute Engine API.
- Automatically granted the IAM basic Editor role, if you have not disabled this behavior.
- Attached by default to all instances created by the Google Cloud CLI and the Google Cloud console. You can override this behavior by specifying a different service account when you create the instance, or by explicitly specifying that no service account be attached to the instance.

- **Preemptible VMs** - You would usually only use them as disposable instances for things like big data batch jobs




**Storage**

- **Standard Persistent Disk -** Compute engines come with standard persistent disks by default. Standard Persistent Disks are magnetic drives. Their main advantage is low cost
- **SSD Persistent Disk -** 4 times the throughput and up to 40 times the I/O operations per second of a Standard Persistent Disk, so if you need high performance, SSDs are a must
- **Local SSD Persistent Disk -** Much faster than SSD Persistent Disks. Local SSDs are not redundant and are directly attached to an instance. That gives them major speed advantages, **but with high risk** because if they suffer a hardware failure, then your data will be gone**.** *Furthermore, Local SSDs disappear when you stop or delete an instance, so you should only use them for temporary data that you can afford to lose, such as a cache.* **They can't be used as boot disks.**
- **Ram Disk -** They are even faster than local SSD and are very expensive. They essentially make a chunk of memory look like a filesystem. Although RAM disks are the fastest option, **they're even less durable than Local SSDs**, so they're only suitable for temporary data
- **Cloud Storage -** Multiple instances can write to a bucket at the same time. **You can't do that with persistent disks, which can only be shared between instances in read-only mode**. The danger is that one instance could overwrite changes made by another instance, so your application would have to take that into account**.** *Instances can access a bucket in a different zone or region, which is great for sharing data globally, especially if it's read-only data, which would avoid the overwriting problem.*




**Cloud Storage usually isn't a good option for compute instance storage**, it is good for general-purpose file serving. *Use Cloud Storage FUSE, which is open-source software that translates object storage names into a file and directory system. Essentially, it makes Cloud Storage buckets look like network file systems. A better choice, though, would be* **Cloud File Store,** *which is a fully supported file sharing service that's designed specifically for this purpose. It's compatible with NFS version 3.*

**Cloud Storage is automatically replicated either across zones in a region (for the regional type) or across regions (for the multi-Region type)**

**Case Study Design**

Which instance storage option should we use for GreatInside?

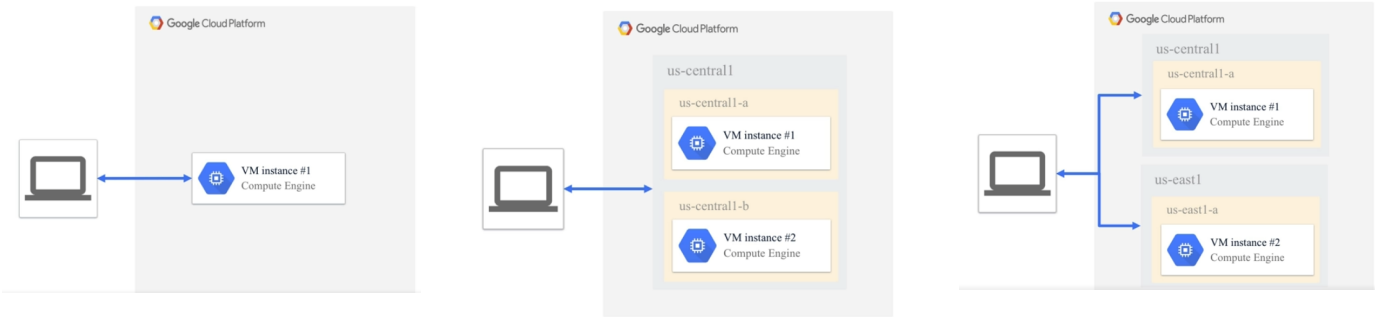
**Database**

- **Cloud SQL**
  - Cloud SQL is a relational database
  - It's a managed service for MySQL, PostgreSQL, or Microsoft SQL Server

    - It's suitable for everything from blogs to ERP and CRM to ecommerce
    - A relational database can scale vertically, meaning that you can run it on a more powerful VM to handle more transactions, but there are obviously limits to the size of a VM
    - You can also scale a relational database horizontally for reads by using read replicas, but most relational databases can't scale horizontally for writes
    - For complex queries, Cloud SQL is a better choice
    - **Multi-region READ from replicas**
- **Cloud Datastore**
    - Cloud Datastore is a NoSQL database service
    - It is horizontally scalable
    - Eventually consistent database
    - It's best suited to relatively simple data and queries, especially key-value pairs
    - Can do multiple write operations per entity group inside the same transaction
    - **Multi-region READ & WRITE**
- **Cloud Bigtable**
    - Cloud Bigtable is also a NoSQL database
    - It's designed to scale into the petabyte range with high throughput and low latency
    - It does not support ACID transactions, so it shouldn't be used for transaction processing
    - It's best suited for storing huge amounts of single-keyed data (>1TB)
    - It can handle big data in real-time or in batch processing
    - Typical examples are Internet of Things applications and product recommendations
- **Big Query**
    - Data warehouse service
    - It's best suited to aggregating data from many sources and letting you search it using SQL queries
    - OLAP based
    - Business Intelligence Reporting
- **Cloud Spanner**
    - It's a relational database that also scales horizontally
    - More expensive than the other options
    - Use Cloud Spanner when you need a relational database that is massively scalable
    - **Multi-region READ & WRITE**

**Case Study Design**

**Auto Scaling Group**




For a higher level of availability, you can use autoscaling instance groups. This was covered extensively in the **"Google Cloud Platform: Systems Operations"** (Cloud academy)

**Auto Scaling Instance Group**

- An Instance group consists of identical instances that perform processing for your application
- If one of the instances fails, then a health check will notice this and replace the instance with a new one
- If the load on the instance group gets too high, then the autoscaler will add more instances to maintain good application performance
- To ensure availability even if an entire zone fails, you should distribute the instances across multiple zones by selecting "Multizone" when you're creating the instance group

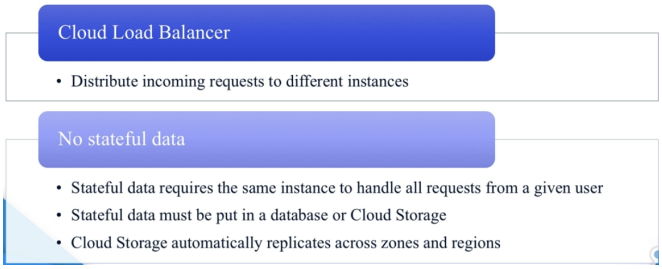
- If you want to make sure you'll still have enough instances to handle the load **if an entire zone goes down, then you should overprovision by 50%**. For example, if your instances are spread across 3 zones and you need 6 instances to handle your normal traffic load, then you should provision 9 instances. That way if one of the zones goes down (which would take out 3 of the instances), you'll still have 6 instances left in the two remaining zones
- You can either overprovision by 50% at all times or **save money by setting the upper limit on your autoscaler to at least 50% more than the normal number of instances.** If you decide to depend on the autoscaler during a zone failure, then the instances in your remaining two zones will be very heavily loaded until the autoscaler provisions additional instances, so only choose this option if you can tolerate this temporary performance degradation

---

**Use Case Design**

Since GreatInside has 6 web tier instances for its main application, this is how it should be set up. For the 2 customer-facing IIS instances in the payment processing system, you'd set an upper limit of 3 instances, which is 50% more than the 2 instances that it normally needs.

To make the instance group work as a high availability solution, you'll need a couple of other components.




**First**, the instance group has to be behind a **load balancer** that will distribute incoming requests to different instances. **Second**, the instances cannot have any stateful data.

Otherwise, the same instance would have to handle all requests from a given user. Although you can enable the "**session affinity**" option in this situation, **it will ruin your high availability since a failed instance will impact all of the users on it.**

Since most applications do have stateful data, you have to put it on other components, such as a **database** or **Cloud Storage.** Unfortunately, that just moves the availability issue to a different layer, but fortunately, Google Cloud has good ways to handle storage availability. **If Cloud Storage is sufficient for your stateful data needs, then you're covered because Cloud Storage is automatically replicated either across zones in a region (for the regional type) or across regions (for the multi-Region type)**. Since GreatInside is going to use Cloud SQL for both MySQL and SQL Server, then we just need to enable the high availability option when we create those databases.

**Virtual Private Cloud Network**

Resources can be isolated in 5 layers:

- Organizations
- Folders
- Projects
- Networks
- Sub Networks

*You aren't required to have organizations or folders, but they can be useful, especially for large companies. Projects are required, though. You use them to provide a level of separation between resources. Not only are resources in different projects unable to communicate with each other, but they're even in different billing accounts. Projects also have separate security controls, so for example, you could give Bob in QA the highest level of access in the Test Environment project, but a lower level of access in the Production Environment project.*

Each project has **one default network that comes with preset configurations and firewall rules** to make it easier to get started, but you can customize it, or you can create up to **4 additional networks (for a total of 5)**. If 5 networks per project isn't enough for you, then you can request a **quota increase to support up to 15 networks** in each project. A network belongs to only one project, a subnet belongs to only one network, and an instance belongs to only one subnet. **A network cannot host more than one project.**

**Subnets** - *they are Region Specific, has an IP address range and a default gateway*

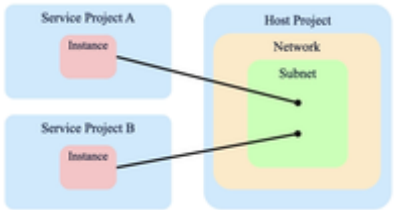
- Instances in the same subnet or even different subnets within the same network can communicate with each other
- Subnetworks are used to group and manage resources
- A network spans all regions, but each subnet can only be in one region

- A subnet allows you to define **an IP address range** and **a default gateway** for the instances you put in it
- The IP address ranges of the different subnets must be non-public (such as 10.0.0.0) and must not overlap
- They must be IPv4 addresses, though, because Compute Engine doesn't support IPv6 yet
- A network can have either **automatic** or **custom** subnets
- With automatic, the subnets are created for you, one in each region. With custom, you create them yourself
- If you discover that you need to customize a network with automatic subnets, then you can **convert it to custom mode**
- You **cannot convert custom mode back** to automatic mode
- On the **default** network, instances within the same subnet can communicate with each other over any TCP or UDP port, as well as with ICMP
- Instances in the same network can communicate with each other, regardless of which subnets they're in, because Google Cloud creates routes between all of the subnets in a network. However, the default network's firewall rules only allow **SSH**, **RDP**, **ICMP** traffic between subnets. If you don't want instances in different subnets to be able to reach each other, then you can change the firewall rules to deny traffic between them
- The default network comes with predefined firewall rules, when you create a new network, it doesn't have any firewall rule. However, the instances in that network will still be able to communicate with the Internet, assuming they have external IP addresses, because all outgoing traffic from instances is allowed. Only incoming traffic is blocked. And when an instance sends a request over the Internet, the incoming response is allowed, so two-way traffic is enabled at that point
- Each network includes a local DNS server so VM instances can refer to each other by name: **[HOSTNAME].c.[PROJECT_ID].internal.** This name is tied to the internal IP address of the instance. An Instance does not know its external IP address and name. That translation is handled elsewhere in the network
- To reach Internet resources, each VM needs an external IP address.
- An **ephemeral external IP** address is created for each VM by default, but an ephemeral address **gets replaced with another one if you stop and restart the instance**, so if you want an instance to always have the same IP address, then you need to **assign a static IP address** to it
- Since IPv4 addresses are a scarce resource, Google doesn't want customers to waste them. So, **you're not charged for having static IP addresses as long as you're using them**. But if a static IP address is not associated with a VM instance or if it's associated with an instance that's not running, then you'll be charged for it
- Normally, if a VM needs to send requests to other Google services, such as Cloud Storage, then by default, it has to do so using a public IP address rather than an internal one. enable the **Private Google Access option** in a subnet, then VMs in that subnet can connect to Google services using internal IP addresses, so their requests will go over Google's network rather than the Internet

### VPC Sharing

- If you want instances in different projects to communicate with each other, then you have three options
  - **The Internet**
  - **VPC Network Peering** - This allows two VPCs to connect over a private space, that is, using non-routable internal IP addresses. In other words, they don't need public IP addresses, and they communicate over Google's network. **Not only can you do this for VPCs in different projects, but you can even use it to connect VPCs in different organizations**.
    To make this work, both sides have to set up a peering association. If only one side sets up a peering association with the other VPC, then the networks won't be able to communicate with each other. Also bear in mind that **there can't be any overlapping IP ranges in the two networks**

  - **Shared VPC -** A more complicated option is to use a Shared VPC. The idea is that instances in different projects can share the same network. This is kind of a weird idea. It is to enforce security standards. For example, if you want to use the same firewall rules across all of your projects, then this is a good way to do that.



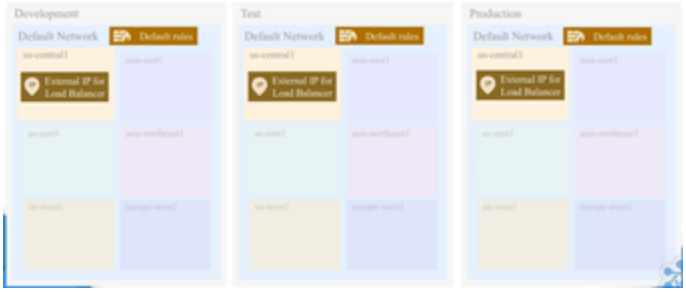

    - To set up a Shared VPC, you need to designate one of the projects as the host project and the others as service projects
    - The host project is the one that contains the Shared VPC
    - Instances in the service projects can use subnets in the Shared VPC
    - *This is made possible by giving Service Project Admins the authority to create and manage instances in the Shared VPC but nothing more*
    - The Shared VPC Admins have full control over the network
    - **All of the projects in this arrangement have to be part of the same organization**

### On-prem with VPC

There are three ways:

- **Cloud VPN** - Lets you set up a virtual private network connection between your own network and Google Cloud.
  - To do this, you need to have a peer VPN gateway in your own network, and it needs to use IPsec to connect to the **Cloud VPN Gateway** a nd encrypt traffic.
  - You can have multiple tunnels to a single Cloud VPN gateway
  - Cloud VPN requires you to make changes to static routes on your tunnels manually
  - But if you use **Google Cloud Router**, then the **routes will be updated dynamically using BGP**. Network topology changes are propagated automatically.
- **Cloud Interconnect**
  - ***Dedicated Interconnect:***
    - If your **internal network extends into a colocation facility** where Google has a point of presence, then you can *connect your network to Google's*
    - Provides higher bandwidth and lower latency than a connection over the public internet
    - It's a bit expensive, though, because the minimum bandwidth is 10 Gbps
  - ***Partner Interconnect:***
    - If you **don't have a presence in a supported colocation facility**
    - You *connect to a service provider* that has a presence in a supported colocation facility
    - You can purchase a monthly contract for connections as small as 50 Mbps and as large as 10 Gbps
- **Direct Peering** - This is similar to Cloud Interconnect because you connect your network to Google's network at a point of presence either directly (which is called **Direct Peering**) or through a service provider (which is called **Carrier Peering**). **One big difference with peering is that it doesn't cost anything**.
- So why would anyone pay for Cloud Interconnect when they could peer with Google for free? Well, because with Cloud Interconnect you get a direct connection between your on-premises network and one of your VPCs in Google Cloud. You have full control over the routing between your networks. If you want to change a route, you can change it on your on-premises router, and it will be picked up by BGP. ***Although the peering option uses BGP, too, it's done at the most basic level. It doesn't create any custom routes in your VPC network.***

### Use Case Design




### Authentication

By default, Google Cloud Platform requires users to have a Google account to access it. But if you have more than a handful of users, then you'll want to find a centralized way to manage your user accounts.

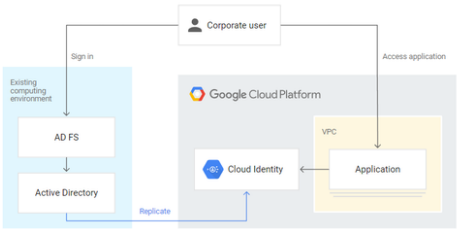
Most organizations already have a user directory, so the best policy is usually to manage users in your existing directory, and then synchronize the account information in G Suite.

There are three ways to do this:

- **Google Cloud Directory Sync or GCDS**
  - Synchronizes users, groups, and other data from your existing Active Directory or an LDAP server directory to your Google Cloud Domain Directory
  - GCDS runs inside your network on a machine that you control
  - One-way synchronization, so GCDS doesn't modify your existing directory
  - Synchronization has to happen on a regular basis to keep your Google Directory up to date
- **Google Apps Admin SDK**
- **Third party connector**
- **SSO (SAML 2.0)**
  - Once you've implemented SSO, then when a user would normally have to login, Google will redirect your authentication system
  - If the user is already authenticated in your system, then they don't have to login to Google Cloud separately
  - If they aren't already logged in, then they're prompted to login
  - Your users must have a matching account in Google's Directory - So you still need to use GCDS or one of the other synchronization options

Interesting Read:

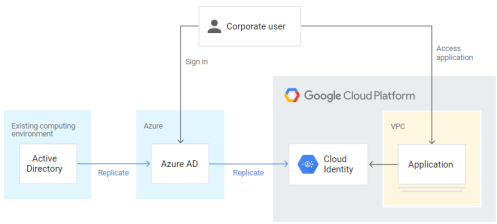
1. Patterns for authenticating workforce users in a hybrid environment | Cloud Architecture Center | Google Cloud
2. Single sign-on | Identity and access management | Google Cloud




For a variation of this pattern, you can also use Active Directory Lightweight Directory Services (AD LDS) or a different LDAP directory with either AD FS or another SAML-compliant IdP.

**User experience**

1. When you request the protected resource, you are redirected to the Google sign-on screen, which prompts you for your email address.
2. If the email address is known to be associated with an account that has been synchronized from Active Directory, you are redirected to AD FS.
3. Depending on the configuration of AD FS, you might see a sign-on screen prompting for your Active Directory username and password. Or AD FS might attempt to sign you in automatically based on your Windows login (IWA).
4. When AD FS has authenticated you, you are redirected back to the protected resource.




For more detailed information about this approach, see Federating GCP with Azure Active Directory.

**User experience**

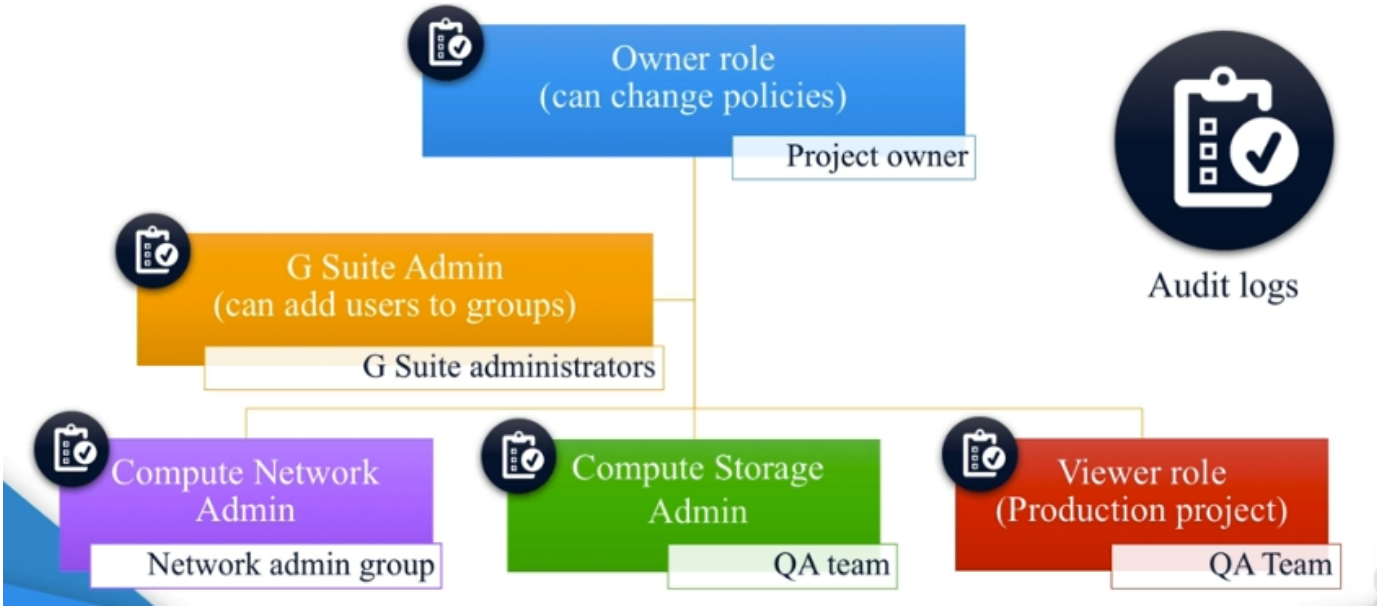
1. When you request the protected resource, you are redirected to the Google sign-on screen, which prompts you for your email address.
2. If the email address is associated with an account that has been synchronized from Azure AD, you are redirected to Azure AD.
3. Depending on how your on-premises Active Directory is connected to Azure AD, Azure AD might prompt you for a username and password. Or it might redirect you to an on-premises AD FS.
4. After successfully authenticating with Azure AD, you are redirected back to the protected resource.

youtu.be

## Roles

- There are 3 *basic roles* available: **Owner, Editor, and Viewer.**
- There are also fine-grained roles for individual resources. These are called *predefined roles*
- You can assign roles at different levels of the hierarchy, that is, at the organization, folder, project, and resource levels
- If you assign roles to the same user at different levels, then their **effective permissions are the union of the permissions** at the different levels
- Use the **principle of least privilege** when granting roles
- Whenever possible, **assign roles to groups** instead of to individuals
- Keep tight control of who can add members to groups and change policies
- Audit all policy changes by checking the **Cloud Audit Logs**
- **Always have more than one owner.** Otherwise, if that person is unavailable or leaves the organization, it would be difficult to for someone else to take their place as owner
- Avoid that situation by giving the owner role to several people, but choose wisely because owners can do just about anything
- Owners are the only ones who can change policies via **Organization Administrator role**
- The Admin Activity audit logs are viewable by all project members. Someone just needs to take on the responsibility of checking for policy changes
- You wouldn't need to explicitly grant access to the person who does the checking

## Case Study Design




**Service Accounts**

To avoid embedding credentials in an application, we need to use service accounts. For example, if an application uses Cloud Datastore as a database, then it needs to have authorization to use the Datastore API.

- By default, all **VM instances run as the Compute Engine default service account.** If you want something different, then you can create your own. Your instances use that identity when communicating with other Cloud Platform services. However, by default, an instance running as the Compute Engine default service account **has limited scope in how it can interact with other services**. By default, an instance can only read from Cloud Storage and can't write to it.
- To give an instance more permissions, you need to *set the scope when you're creating the VM*
- Application code has to obtain credentials from the service account whenever it uses the GCP APIs
- Google Cloud Platform uses OAuth 2.0 for API authentication and authorization. There are two ways to do it:
  - *Application Default Credentials (using Google Cloud Libraries)* - One great feature of ADC is that you can test your application locally and then deploy it to Google Cloud without changing the application code. Just set the environment variable GOOGLE_APPLICATION_CREDENTIALS to the path of the service account credentials file.
    `gcloud auth application-default login --client-id-file=clientid.json`
  - *Access tokens* - The second way is to use OAuth2 access tokens to directly connect to the API without going through a client library. One reason you'd have to use this method is if your application needs to request access to user data.
    Tokens are short-lived, so your application needs to request new ones regularly.
    `gcloud auth login`
- If you need to write shell scripts that access other Cloud Platform services, then you can use *gcloud* and *gsutil* commands to make API calls. These two tools **are included by default in most Compute Engine images** and they automatically use the instance's service account to authenticate with APIs.

**Use Case Design**

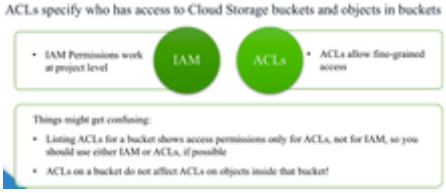
**Data Protection and Encryption**

There are actually five different mechanisms for controlling access to Cloud Storage:

*Also,* https://krishnendu-das.atlassian.net/l/cp/pRkYBatR *for more on ACLs and Encryption*

- **IAM permissions**
  - IAM permissions work at the project level
  - Specify that a user has full control of all the objects in all of the buckets in your project, but cannot create, modify, or delete the buckets themselves

- **ACLs**
  - Set fine-grained access, such as which buckets or objects a particular group can read
  - **Try to use either IAM or ACLs, but not both**
  - The ACLs on a bucket do not affect the ACLs on objects inside that bucket
  - **Bucket and object ACLs are independent of each other**

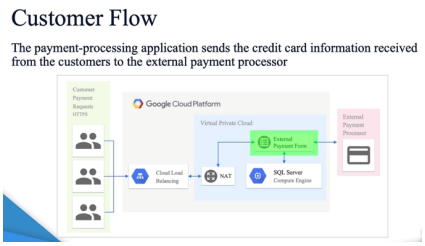


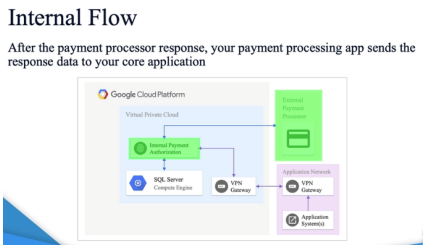

- Signed URLs
- Signed Policy Documents
- Firebase Security Rules

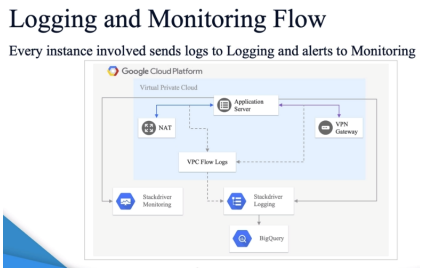
### Legislation and Compliance

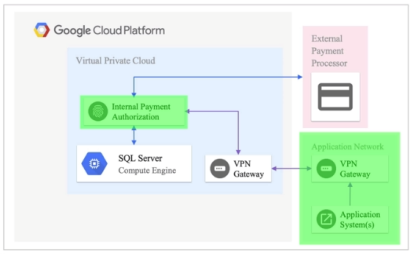
**PCI DSS Compliance** (particular for credit card processing) - If your organization accepts credit card payments, then you need to comply with this standard or you could be fined. More importantly, if you have security flaws that allow hackers to steal credit card information from your systems, then it would be very damaging to both your customers and your reputati on.

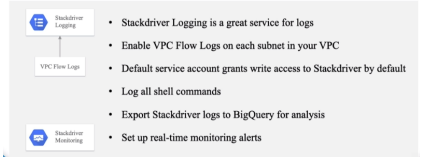
#### Architecture

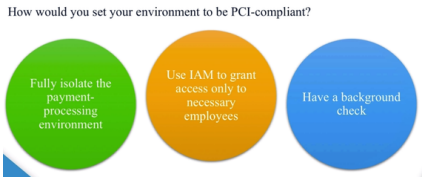


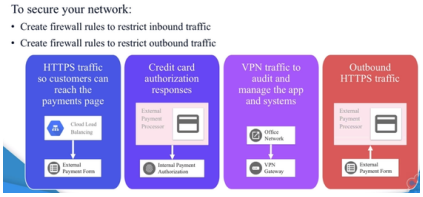





**Disaster Recovery**

Smaller incidents can be disastrous. For example, one of your databases could become corrupt. To recover from this sort of disaster, you need backups along with transactional log files from the corrupted database. That way you can roll back to a known-good state. Each type of database has its own method for doing this.

**Cloud SQL - MySQL**

- Enable **automated backups** and **point-in-time recovery**
- Restore from backups when corrupt

**Cloud SQL - SQL Server**

- Enable **automated backups**
- Does not support **point-in-time recovery**
- Cloud SQL retains up to **7 automated backups** for each instance of SQL Server

**Database on Compute Engine**

- Manually configure backups and transaction logging yourself
- First, set up an automated task that copies the SQL Server database backups to Google Cloud Storage. This is where we would finally need a service account because instances can't write to Cloud Storage by default. The SQL Server instances need to have a service account with the **Storage Object Creator role OR set a Cloud Storage access scope for the instance**
- Next, if disaster strikes, you will spin up a new SQL Server instance. Either use one of Google's preconfigured SQL Server images or your own custom disk image
- Then, use an open-source script to restore the database and re-execute the events in the log files up to the point in time desired

**RTO and RPO**

- Recovery Time Objective - This is the maximum length of time that your application can be offline and still meet the service levels your customers expect. *The advantage of having a custom disk image is that it helps you meet your recovery time objective because it reduces the amount of time it takes to get a new SQL Server instance running. If you have to configure SQL Server manually, that could significantly impact how long it takes to recover from a disaster*
- Recovery Point Objective - This is the maximum length of time when data can be lost. *It affects your backup and recovery strategy because, for example, if it's acceptable to lose an entire day's worth of work, then you can just recover using the previous night's backups. If you have a short RPO, which is usually the case, then you need to make sure you are constantly backing up your data, and when recovering from database corruption, you have to carefully consider which point in time to recover to*

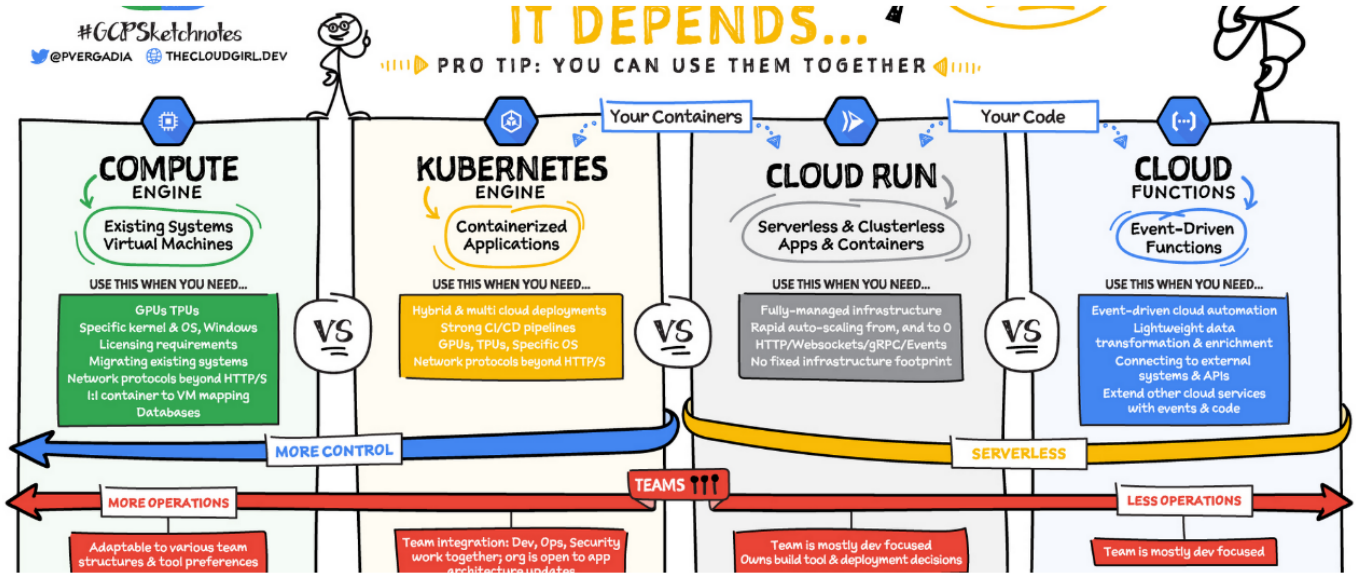
**Common Recovery Design pattern**

You may want to *run an application from your own data center* or from another cloud platform and *use Google Cloud as a disaster recovery solution*. There are many ways you could do this, but I'll go over a couple of common designs.

1. The first way is to continuously replicate your database to an instance on Google Cloud. Set up a monitoring service that would watch for failures. In the event of a disaster, the monitoring service would trigger a spin-up of an instance group and load balancer for the web tier of the application. The only part you would need to do manually is to change the DNS record to point to the load balancer's IP address. You could use **Cloud DNS** or another DNS service for this. This is already a low-cost solution because the only Google Cloud resource that needs to run all the time is the database instance. But you can reduce the costs even further by **running the database on the smallest machine type capable** of running the database service. Then if there's a disaster, you would delete the instance, but **with the option to keep the persistent disk**, and **spin up a bigger instance with the saved disk attached**. Of course, this solution would require more manual intervention and would lengthen your downtime, so you wouldn't want to do this if you have a short RTO.
2. If you want to reduce your downtime as much as possible, or even keep running in the event of hardware failures, you could **serve your application from both your on-premises environment and your Google Cloud environment at all times**. That way if you have an on-

premises failure, the Google Cloud environment would already be running and serving customers. It would just need to scale up to handle the extra load, which would be automatic if you use an autoscaling instance group. To make this hybrid solution work, you would need to use a DNS service that supports weighted routing, so it could split incoming traffic between the two environments. In the event of a failure, you would need to disable DNS routing to the failed environment.

## Google Kubernetes Engine and App Engine




> *Use Cloud Run if you just need to deploy a containerized application in a programming language of your choice with HTTP/s and websocket support. Examples: websites, APIs, data processing apps, webhooks.*
>
> *Use App Engine if you want to deploy and host a web-based application (HTTP/s) in a serverless platform. Examples: web applications, mobile app backends*

**READ**
**Where should I run my stuff? Choosing a Google Cloud compute option | Google Cloud Blog**
**When to use Google App Engine Flex vs Google Cloud Run - Stack Overflow**

### Managing Kubernetes Engine Resources

- Kubernetes is a system for orchestrating containerized applications
- Kubernetes gives you the tools to abstract away most hardware and networking considerations to make & manage y containerized systems
- A Kubernetes cluster is a complete set of hardware resources for an application environment
- This will be **confined to a single data center** and will comprise a number of servers and network interfaces. Hence Multi Cluster Kubernetes systems.
- Storage is also a possible resource here as Kubernetes can create *ephemeral and persistent volumes*
- The servers, whether physical machines, VMs running in EC2, or Google Cloud Compute are referred to as nodes. **Servers are nodes**
- **A Kubernetes cluster may, for example, run on three nodes,** three virtual servers, that will host your container-based applications
- Having multiple nodes grants redundancy in case of hardware failure and it also makes scaling up or down easier
- Each node will run a kubelet, a lightweight Kubernetes agent, that allows it to communicate with other nodes and stay in sync with cluster configuration and help
- In order to run a container, **the cluster must have access to an image repository, and it has to create pods**
- **A pod is a basic workload unit in Kubernetes**
- Often, a pod is an instance of a single container, however, it can also be comprised of multiple containers
- A pod will also have a unique IP address within the network as well as storage resources based on its config
- **Controllers:** You can think of a Kubernetes Controller as a control loop. This is a basic CS concept. It's a tool for maintaining a certain desired state. In Kubernetes, this refers to an API that manages a pod or set of pods by preserving a pre-set configuration. There are actually a few different types of Controllers.
  - *ReplicaSet* - which simply guarantees that a certain number of copies of a given pod are kept running
  - *DaemonSet* - which is meant to guarantee a specific distribution of pods on each node. This controller is useful for hardware monitoring, logging applications that need a one-to-one mapping to servers for whatever reason
  - *StatefulSet* - designed for stateful applications. This controller provides guarantees about pod ordering, uniqueness, and stable storage
  - *Deployment* - This is designed to work with other Controllers, such as ReplicaSets. It is designed for declarative updates to a set of pods - handles transitioning a set of pods from its current state to a defined desired state
- **Services:** Services are Kubernetes' way of exposing pods to external networks including the public internet. To make our application reachable from a browser, we're going to need a Service to set up the IP address and DNS name. There are a few different types of Services
  - *ClusterIP* - Exposes an internal IP address, suitable for apps that don't need to be accessed from the public internet

- *NodePort* - Exposes the node's IP address on a specific port
- LoadBalancer - Works with your provider's LoadBalancer resources to expose a set of pods
- ExternalName - Returns a CNAME record based a DNS name of your choosing

- **Controllers let us turn sets of running containers into resilient, updateable applications with predictable behavior. And Services let us control access to those applications by publishing them in various ways**

Demo Configuration: GKE

1. Create a basic application package using Docker and upload it to a registry

```
gcloud components install kubectl
docker build -t gcr.io/${PROJECT_ID}/hello-app:v1 .
gcloud auth configure-docker
docker push gcr.io/${PROJECT_ID}/hello-app:v1
```

2. Create a GKE cluster and

```
glcoud config set project $PROJECT_ID
glcoud config set compute/zone us-east1-b
gcloud container clusters create hello-cluster --num-nodes=2
gcloud compute instances list #instance in the cluster
gcloud container clusters list
```

3. Deploy the app into it

```
kubectl create deployment hello-web --image=gcr.io/${PROJECT_ID}
/hello-app:v1
kubectl get pods
```

4. Expose the app to the internet

```
kubectl expose deployment hello-web --type=LoadBalancer --port 80 --
target-port 8080
```

Configuring Application Traffic

Two types of app engine environment:

- **Flexible**
  - Flexible environment gives you direct control over your application runtime via docker files
  - You can enable root access to the underlying VM instances
  - Flexible environment is slower and it's less resilient
  - Instances take minutes to deploy and start up
  - Instances are automatically restarted more frequently by GCP
  - No access to all the same App Engine API, such as the users and images APIs, which are useful for automation scripting
  - If your runtime is not available in the provided runtimes, you can always **create your own Docker file** for the execution environment. The caveat with it is, it requires *having at least 1 instance running, even if no-one is using your app*, plus the scaling up and down requires a few minutes
- **Standard**
  - Standard environment has a bit more flexibility with its autoscaling options

    - Greater speed, reliability, and feature richness
    - Has dedicated Google services and libraries for caching, authentication
    - *The caveat with Standard environment is that it is very restrictive* ***since it runs in a sandbox***
- Configure your app with YAML files
- Only one YAML file is absolutely required, and that is the **app.yaml** file
- An application may be made up of multiple services
- They are configured in **service.yaml** files. You can have more than one **service.yaml** file in your applications root directory
- It is better practice to have separate subdirectories for each service
- **Example**
  So, here's a sample app.yaml file. This is the one for the Python app we launched in the last lesson. One line, "`runtime: python 37`". As you can see, our app.yaml file is pretty simple. **The only thing app.yaml really needs is runtime, is to specify a runtime and potentially a version parameter**. Now if we wanted to, we could break this out into a separate service by using a service.yaml file. And it would look pretty similar, with the only difference being that the service.yaml file generally starts with a field called "service". So, it would look something like this.

  ```
  service: python-service

  runtime: python37

  api_version: 1

  threadsafe: true
  ```

  We have a service name, runtime. And we can also put in an API version, and some other config. Now a more complex app might have several of these service.yaml files to create microservices, to create a microservices architecture within App Engine. Now aside from this, there are five other optional configuration files that can help you to extend your app's overall functionality.
- **Other Files**
    - *Dispatch.yaml* - For overriding routing rules. You set this file in your application root directory and you use it to route incoming requests to specific services based on specific paths or host names in the URL
    - *Queue.yaml* - For configuring push and pull task queues. This lets us define retry parameters, such as minimum back off time in seconds or, and a maximum task age, for example
    - *dos.yaml* - A security feature that lets you blacklist IP addresses or subnets to protect your application from DOS attacks
    - *cron.yaml* - Lets you define scheduled tasks. You set a schedule, such as every 24 hours, or every Monday at six p.m. and then give it a URL for a task definition, much like a typical cron job. The task definitions should live in the specified path. Now, this is very handy for maintenance, monitoring, and other standard automated tasks that you might want to configure and have in one easy to work with location.

## Big Data

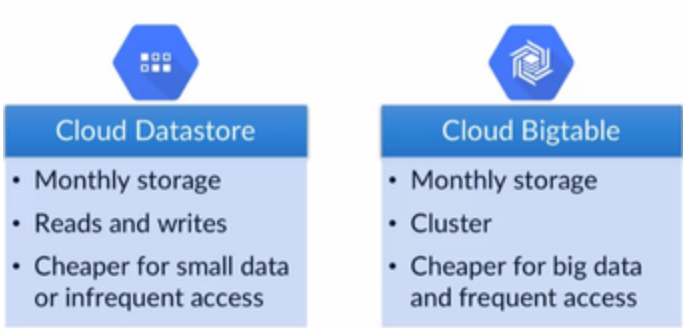
https://dzone.com/articles/hadoop-vs-database-vs-cloud-dwh

https://www.linkedin.com/pulse/big-data-analysis-architecture-considerations-mpp-dws-bryan-linton/

Read more: Design Patterns

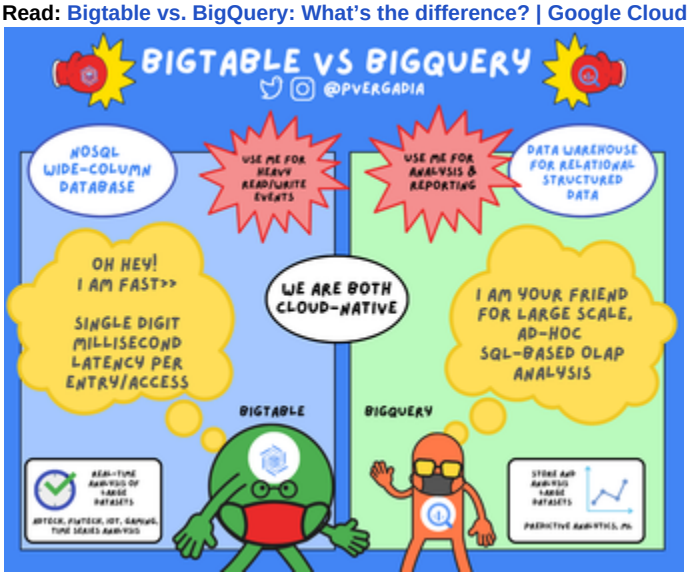
### Bigtable

- Kickstarted NoSQL industry
- Low latency access to Big Data
- Good for > 1TB data
- Good for element < 10 MB
- Stores Key Value pairs
- Perfect for Map Reduce as it has high throughput and scalability
- Reak time analytics with **Dataflow or DataProc**
- Fr interactive queries, **BigQuery** is better fit
- Pay for reads and writes along with the cluster maintenances
- **Cloud DataStore** is the better choice as it brings best of both worlds
    - Can handle more complex data than simple key value pairs
    - Secondary indexes are possible
    - Support ACID properties
    - SQL like Query language
    - Pay for reads and writes and not for the cluster
    - Cheaper for small amount of data




- **Cloud Spanner** is the most expensive database service
  - Supports relational schema
  - Strong consistency instead of eventual consistency
  - Multi Region Deployments

**Read:** **Bigtable vs. BigQuery: What's the difference? | Google Cloud Blog**




Bigtable use cases are of a certain scale or throughput **with strict latency requirements**, such as IoT, AdTech, FinTech, and so on.

**Architecture**

When a client sends a request to Bigtable, it first goes through the Front-end server pool, which is basically a security layer. Then the request gets sent to a node in a Bigtable cluster. The node then reads from or writes data to Colossus, which is Google's filesystem.

This architecture has a couple of interesting characteristics. First, you can easily scale the total throughput and number of simultaneous requests handled by adding more nodes to the cluster. Second, since the nodes don't hold any of the data, if a node goes down, there's no data loss.

OK, so what do the nodes actually do if they don't hold data? Well, they hold pointers to specific subsets of the table. Bigtable divides (or shards) a table into groups of rows. Each group of rows is called a tablet. Tablets are stored on Colossus in SSTable format. Each tablet is associated with a particular node.

The beauty of using pointers to the data is that Bigtable can quickly rebalance the cluster load by moving pointers from one node to another. For example, if a majority of the requests are going through one particular node, then Bigtable can move the pointers for some of the tablets from that node to the other nodes. The overall performance will improve because the nodes are now sharing the load fairly equally.

It can even do this if an individual tablet is getting a disproportionate number of requests. In this case, it will split the tablet in two, and then move the pointer for one of the new tablets to another node. There are limits to its ability to rebalance, though, based on how well you design your schema.

Importantly, data is never stored in Bigtable nodes themselves; each node has pointers to a set of tablets that are stored on Colossus

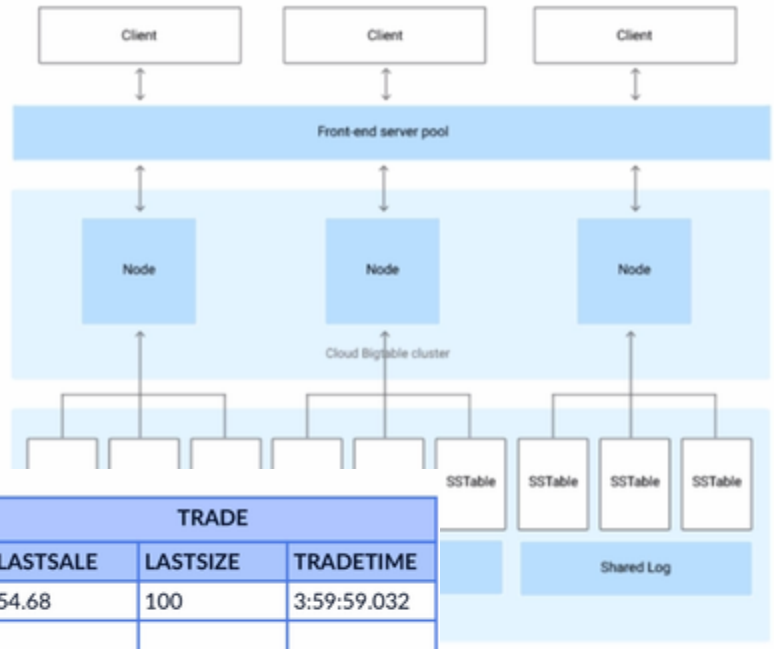
- Rebalancing tablets from one node to another happens quickly, because the actual data is not copied. Bigtable simply updates the pointers for each node
- Recovery from the failure of a Bigtable node is fast, because only metadata must be migrated to the replacement node
- When a Bigtable node fails, no data is lost

**Storage Model**

Suppose you have a table of stock data after a stock exchange has closed for the day. Each stock has a stock symbol, a last sale price, the number of shares sold in that transaction (called “last size”), and when the trade occurred.

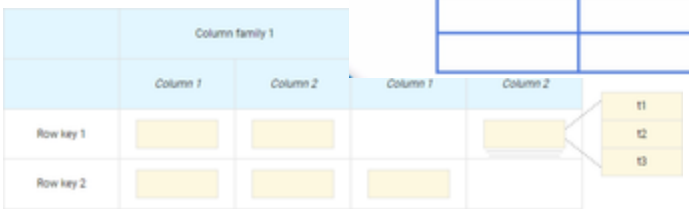
In Bigtable, you could represent this data with a structure that looks like this. Each record takes up a row in the table. The row key is the stock symbol for a company. Each record has 3 columns with data related to the row key: last sale, last size, and trade time. So far, this should look very familiar because relational database tables look like this too.

One difference is that Bigtable has what it calls “column families”. These are groups of related columns. This example only has one column family, called “TRADE”.




Column Family

| | TRADE | | |
|---|---|---|---|
| Row Key | LASTSALE | LASTSIZE | TRADETIME |
| ZIPP | 54.68 | 100 | 3:59:59.032 |
| | | | |
| | | | |
| | | | |




To illustrate some other differences, let’s look at another example. Suppose you have a table that tracks which users are following which other users on a social network like Twitter. This table structure is radically different from what you’d use in a relational database. There’s a row for each user and a column for each user. This is known as a wide table, because of the potentially large number of columns. In contrast, the stock market table was a tall table, because each row is quite narrow (with only 3 data columns in this case), but there are a large number of rows.

“follows” column family

| | Follows | | | |
|---|---|---|---|---|
| Row Key | gwashington | jadams | tjefferson | wmckinley |
| gwashington | | 1 | | |
| jadams | 1 | | 1 | |
| tjefferson | 1 | 1 | | 1 |
| wmckinley | | | 1 | |

Multiple versions

*If a cell is empty, then it doesn’t take up any space in the database. That’s because tables in Bigtable are sparse. Wide Table can have millions of columns*

| Row Key | LASTSALE | LASTSIZE | TRADETIME |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

OK, back to the wide table example. If a particular user, such as “gwashington”, follows another user, such as “jadams”, then there will be a 1 in that cell. If gwashington doesn’t follow a user, then the contents of the cell is empty.

Here’s one of the big differences compared to relational databases. If a cell is empty, then it doesn’t take up any space in the database. That’s because tables in Bigtable are sparse. This means, for example, that if a row with a thousand columns only has data in one column, then that row will only take up the space required for that one piece of data. That’s why you can get away with having seemingly crazy table structures like this one, where you could potentially have millions of columns.

Bigtable makes this work by storing both the column qualifier (that is, the column name) and the column value in a row. So each row is simply a list of key/value pairs, where a key is the column family plus the column qualifier. There's actually one other piece of information in the key as well, a timestamp.

| Row Key | Column Data |
|---|---|
| gwashington | Follows:jadams:1 |
| jadams | Follows:gwashington:1, Follows:tjefferson:1 |
| tjefferson | Follows:gwashington:1, Follows:jadams:1, Follows:wmckinley:1 |
| wmckinley | Follows:tjefferson:1 |

| Row Key | Column Data |
|---|---|
| gwashington | Follows:jadams:2017-11-04_090153.107:1 |

That might seem like a weird thing to include in a key, but this gives Bigtable the ability to store multiple copies of each cell. Every time the value in a cell changes, it stores the new value along with the timestamp of when it changed. The old versions don't stay around forever, but multiple versions do exist for a while.

A few things to notice in this illustration:

- Columns can be unused in a row.
- Each cell in a given row and column has a unique timestamp (t).
- **rowkey is the index for the table**

Read more: Bigtable overview | Cloud Bigtable Documentation | Google Cloud

**Schema Design**

- Choosing a row key is crucial because it is the only index in the table and rows are sorted by row key. Querying anything else would result in full table scan.
- Row Key determines the grouping for read and writes in tablets. Appropriate row key helps in evenly distribution of loads from the nodes and prevents swapping.
- Identical chunks of data in the adjacent rows result in compression
- Short row key takes up less storage and memory
- **Choosing a Row Key**

| Field | Example Data |
|---|---|
| Stock Symbol | ZIPP |
| Sale | 53.92 |
| Size | 200 |
| Trade Time | 2017-11-04_10:47:26.032 |

Let's see how this applies to the stock exchange example above. Suppose instead of just containing the last trade of the day for each stock, the table contains all of the trades throughout the day for the past year.

The first thing to consider is what queries will typically be used on this table. *Let's say that most of the queries ask for all of the trades for a given stock for the past day*. Since the row key is the only index, we should definitely use the stock symbol in the row key. But if we only include the stock symbol, then Bigtable would have to do a **full table scan to retrieve all of the stock's trades for the day because the dates wouldn't be sorted**. The simple solution is to **include the timestamp as part of the row key**. It's very common to include multiple pieces of data in a row key. But you'd almost never do this with a relational database.

Row key: Stock Symbol # Timestamp

ZIPP#2017-11-04_10:47:26.032

- **Checking Row Keys**

**Even Distribution Avoiding Hot Spotting**

If we were to put the timestamp first in the row key, then we'd have a hotspot, because as the trades came in from the stock exchange, all of the rows would be written sequentially through the same node, since the table would be sorted by date. We'd have the same problem for reads because most of our queries would be for the previous day, which again, would be linked to the same node.

**Similar Row Key Compression**

With this row key, we'd have identical patterns of data at the front of the key for lots of adjacent rows, so that's a plus.

Even distribution across nodes?

- Timestamp # Stock Symbol would create hotspot

```
2017-11-04_09:30:00.032#ZIPP
2017-11-04_09:30:00.104#ACME
2017-11-04_09:30:00.215#BOOK
```

- Stock Symbol # Timestamp would distribute across nodes

```
ACME#2017-11-04_09:30:00.014
ACME#2017-11-04_09:31:24.743
ACME#2017-11-04_09:32:44.512
```

```
ZIPP#2017-11-04_09:30:00.085
ZIPP#2017-11-04_09:31:42.463
ZIPP#2017-11-04_09:32:55.715
```

Putting the stock symbol first, on the other hand, would distribute reads and writes evenly between nodes. For example, the data for stock symbol ACME would be on the first node and the data for ZIPP would be on the last node. As the trades came in from the stock exchange, they would be written on the node where the stock symbol resides

- Efficient compression?
  - Stock Symbol # Timestamp

```
ACME#2017-11-04_09:30:00.014
ACME#2017-11-04_09:31:24.743
ACME#2017-11-04_09:32:44.512
```

- Small row key size?
  - Length is reasonable

How about the size of the row key? Well, it may look kind of long because we've concatenated two pieces of data together, but really, it's still relatively short, especially because the stock symbol is limited to four characters.

Generally speaking, the columns in a column family are stored together, accessed together, and cached together, so you can make queries more efficient by grouping related columns into column families. In this case, though, we'll usually be retrieving all of the columns when we perform a query, so we can put them all in the same column family: "TRADE"

- **A Complicated Example**
  Suppose you need to collect about 100 server metrics, such as CPU, memory, and disk usage, every minute, from a large number of servers in your datacenter. Also suppose that there will be two common types of queries on this data.

| Field | Example Data |
|---|---|
| Server Name | ursa1 |
| CPU0_SY | 1.6 |
| MEMFREE | 417408 |
| SWAPFREE | 5416960 |
| ... | ... |

- *The first is a request for the metrics for a given machine over a certain period of time* - The first type of query requests metrics from a particular server, the row key should start with the server's name. And since it requests metrics from a particular period of time, we should include the timestamp. Reads will be more or less evenly distributed.

  - Server Name # Timestamp in METRICS

```
bear1#2017-11-04_09:30:00.014
bear1#2017-11-04_09:31:24.743
bear1#2017-11-04_09:32:44.512
```

```
ursa1#2017-11-04_09:30:00.085
ursa1#2017-11-04_09:31:42.463
ursa1#2017-11-04_09:32:55.715
```

- *The second is a request for the latest metrics for all of the machines* - If we use those two pieces of data for the row key, then it will be difficult to run the second query efficiently. That's because it would need to retrieve one row of data (the latest one) for every single server, which would be spread out over the entire table. In situations like this, the best solution is usually to create a separate table with a different row key, and possibly only a subset of the data. In this case, we can create a new table called "CURRENT_METRICS" that contains a single row (the latest one) for each server. Then the row key could simply be server's name. Reads will be evenly distributed

  - Server Name in CURRENT_METRICS

```
bear1
```

```
ursa1
```

  - Since each row is supposed to hold the latest data for a server, you'd have to overwrite the data in every row every minute because that's how often we collect new metrics. *This isn't a real problem, though, because Bigtable* ***supports single-row transactions****. That is, if you* ***update multiple columns in a row at the same time****, Bigtable will guarantee that all of the column updates will either succeed or fail together.*
  - Since many of the queries are likely to request only a subset of the 100 or so metrics for each server, it would probably be a good idea to group them into column families. For example, the various CPU metrics should be in a **column family called "CPU"** and the various memory metrics should be in a column family called "MEMORY"

| Column | Column Family |
|---|---|
| CPU0_SY | CPU |
| CPU0_US | |
| ... | |
| MEMFREE | MEMORY |
| MEMUSED | |
| ... | |
| DISK0_FREE | DISK |
| DISK0_USED | |
| ... | |

- On the other hand, if your queries will almost always request all of the metrics instead of a subset of them, then you should use one column family for all of the columns, because that will improve performance

- **Avoid Hot Spotting**
  - **Field Promotion -** Move one of the data fields to the beginning of the row key, such as moving the stock symbol or the server name into the row key
  - **Salting -** Add a bit of salt to each key, which just means to add a number. For example, you could add either a 1, 2, or 3 to each key. By putting one of these numbers before the timestamp, you would randomly spread the data into three different piles. If you had 3 nodes in your Bigtable cluster, then this would work well. But if you had to scale up to more nodes, then this salting design might not work as well.
  - **Hashing -** Not good for time stamp. Randomize the keys by applying a hash function and using that as part of the key. This would distribute your data nicely across the cluster. For example, if you want to start a row key with a username, followed by another field, such as a follower of that user, then you may want to put a hash of the username into the key instead of the username itself, and also do the same with the username of the follower. That way, each portion of the key would always be the same length, which could be helpful. Hashing a timestamp is always problematic.
- **Domain Name Row Key example**

- Avoid domain names

  beam.lightning.com
  bear.example.com
  beast.lightning.com

- Reverse domain names

  com.example.bear
  com.lightning.beam
  com.lightning.beast

- **Limits**

Recommended Limits

- Row keys: 4 KB per key
- Column families: ~100 families per table
- Column qualifiers: 16 KB per qualifier
- Individual values: ~10 MB per cell
- All values in a single row: ~100 MB

Hard Limits

- Individual values: 100 MB per cell
- All values in a single row: 256 MB
- Tables: 1,000 per cluster

**Cluster Configuration**

| Storage Type | Reads | Writes | Scans |
|---|---|---|---|
| SSD | 10,000 QPS @ 6 ms | 10,000 QPS @ 6 ms | 220 MB/s |
| HDD | 500 QPS @ 200 ms | 10,000 QPS @ 50 ms | 180 MB/s |

These numbers are per node, and Bigtable scales linearly, so if you were running 10 nodes, then your performance would typically be 10 times these numbers, which is pretty incredible. But, overall, the HDD performance is really bad. So why would you ever use HDDs with Bigtable? - Well, in most cases, you shouldn't. Although they're less than one-sixth the cost of SSDs, you'd have to use way more nodes in your cluster to get even close to the performance of SSDs, and the extra cost for the nodes would far outweigh the cost savings from storage.

The only time you might consider using HDDs is if you're storing more than 10 terabytes of data, your application is not sensitive to response time, and you're mostly running batch scans and writes, rather than random reads of individual rows. *But frankly, if that's the type of application you're planning to build, then you'd probably be better off using something like BigQuery instead anyway.* ***Once you choose HDDs, you can't switch to SSDs later!! You'd have to export all of the data, create a new SSD-based instance, and then import all the data into the new one. And that wouldn't be much fun.***

**Two possible causes of poor performance in Bigtable. The first was choosing a poor row key and the second was using HDDs. There are lots of other potential sources of performance problems too:**

- Bigtable is designed for large workloads - you have a small amount of data or access it for a short period of time, then your performance will be detrimental
- Too much workload on few nodes can also bring the performance of the cluster down - To prevent this from happening, *you need to monitor Bigtable's performance and add more nodes when the existing ones are overloaded*. You can do that manually through the console or programmatically using **Stack Driver Monitoring**. After adding more nodes to the cluster, it will take up to 20 minutes under load before an improvement in performance.
- Another reason for having too few nodes is if you're running a development instance, which only has one node. *Fortunately, you can upgrade an existing instance from development to production*. You can't downgrade from production to development, though.
- Network Issues can weaken the performance - One reason for this would be if your clients are running in a different zone from your Bigtable cluster. *You should always have the big table clients in the same zone as the big table cluster*

**Access Controls**

Bigtable provides access control at the project, instance, and table levels. The most important roles for controlling access to Bigtable are reader, user, and admin. The reader role grants read access to data in tables and also to the performance monitoring graphs. The user role grants write access to tables. The admin role grants full access to everything in Bigtable, including the ability to create new instances.

- If you set these roles at the project level, the permissions apply to all of the tables in the projects. A reader can read all of the data in all of the tables in the project.
- If you don't want a user to have the same level of access to all of the tables in a project, then you can assign roles at the individual table level.
- There's no way to limit a user's access to specific rows or columns
- To assign one of the 3 roles to a GCP user, you need to go to the IAM page in the console

**Using Bigtable**

To use Cloud Bigtable, you will almost certainly need to connect to it from another product or service. Bigtable integrates with Cloud Dataflow (Google's big data processing system), Cloud Dataproc (Google's service for running Hadoop and Spark jobs), and BigQuery (Google's data warehouse).

**Example Code: cloud-bigtable-examples/java/dataproc-wordcount at master · cloudacademy/cloud-bigtable-examples (github.com)**

Steps to Run the Job

- Make sure all of the necessary APIs are enabled
- Create a Bigtable cluster
- Create a Cloud Storage bucket for Cloud Dataproc
- Create jar file for Hadoop MapReduce job
- Create the Dataproc cluster
- Run the job

Big Query

https://cloudacademy.com/course/introduction-to-bigquery/

https://www.theseattledataguy.com/snowflake-vs-bigquery%E2%80%8A-%E2%80%8Atwo-cloud-data-warehouses-of-many-setting-up-your-data-in-the-modern-data-stack/#page-content

## The limitations/disadvantages of BigQuery

As you may have heard, all tools are not created equal. With BigQuery its nothing different. BigQuery is amazing with huge datasets and workloads and not so good with smaller dataset. Meaning it might be like using '*sledgehammer to crack a nut*'. This might not be the case for every small medium dataset but hope you get the point. Let's see some use cases and some scenarios where BigQuery will perform really well and some limitations and cases where BigQuery might not be the best tool.

- BigQuery is not a replacement for a RDMS like SQL server, MySQL or MongoDB. You can definitely use it as one but it's not ideal for transactional or live production database. Its very good for storing historical data and running complex analytical queries.
- BigQuery likes Big Data. If you run queries on megabytes or couple gigabytes of data, then it won't give you optimal performance.
- BigQuery charges you for data storage and running queries. If your queries are not efficient and optimized, they run for too long. This will consume more resources which will increase your bill.
- CRUD operation limits. BigQuery is not a great when you are frequently inserting and updating data. There are various limits when it comes to data processing and queries and processes even fail because you have exceeded the limits.

**Introduction**

https://cloudacademy.com/course/introduction-to-bigquery/

- Ease of implementation
  - Building your own is expensive, time consuming, and difficult to scale
  - With BigQuery, just load data and pay only for what you use
- Speed
  - Processes billions of rows in seconds
  - Handles real-time analysis of streaming data

Storage

- $0.02 per GB per month
- After 90 days with no edits, price drops to $0.01 per GB per month
- No charge for reading data from storage

Queries

- First 1TB per month is free
- After first TB, it's $5 per TB ($0.005 per GB)
- Same cost regardless of where data is read from

Flat-rate pricing

- Only worthwhile for more than $2,000 per month
- Only applies to query costs, not storage

- If no destination table, results are cached
- Temporary table stays in cache for one day
- If you run a query again within 24 hours, there is no charge
- If you run a query again and specify a destination table, it won't read from cache, and you will be charged

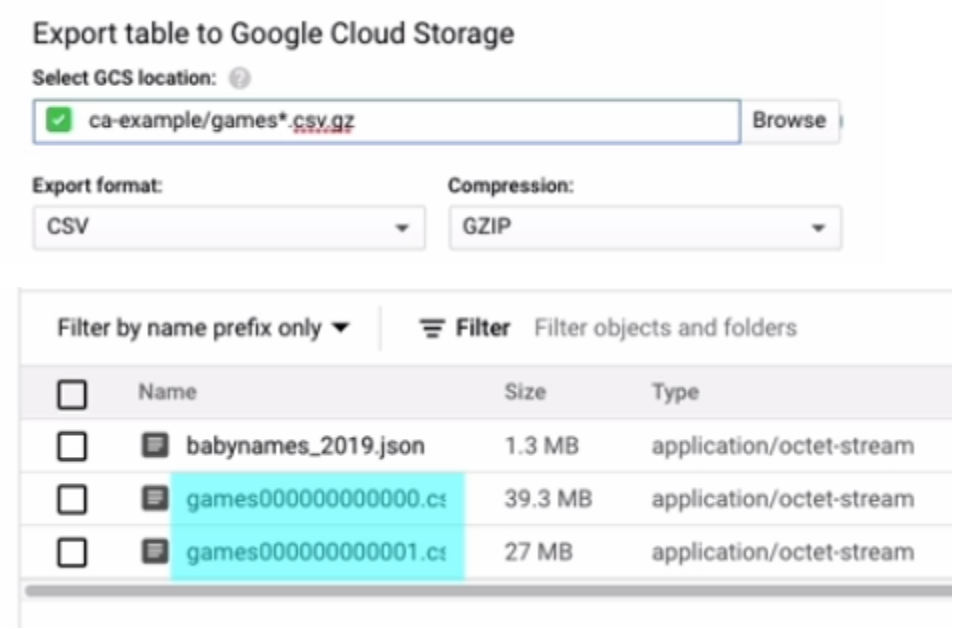
- Can't rename a dataset directly
- To do it
  - Create a new dataset
  - Copy all tables from the old dataset to the new dataset
  - Delete the old dataset and its tables

**Real Time Data**

- Streaming inserts: $0.05 per GB
- Loading data (non-streaming): Free
- Querying data: $0.005 per GB
- Storage: $0.02 per GB per month, at most

- You can only export to Cloud Storage
- To export anywhere else, export to Cloud Storage first, and then download

Large Files cannot be exported into one file. They are split into many files with a provided prefix.




**Google Data Transfer**

**Core Concepts Overview**

**Dataset**

A Dataset is a top-level item which can be created inside a GCP project. A Dataset is a kind of container where you can create **Tables** and **Views** . That means that if you want to create a certain number of tables, you can't create them and let them be isolated. They need to be part of a specific Dataset.

**Table**

A Table is an item which contains your data. Data inside a Table are represented as **records**, and each record identifies a **row**. A row is composed of columns (also called **fields**). Each field of a row contains only one datum and it has its own type.

**Schema**

Every BigQuery Table is defined by a **schema** which describes information related to the columns (names, type of the data and other info). You can specify the Table schema before populating it with data, or you can use the **schema auto-detection** feature offered by BigQuery which allows you to insert data and populate the Table and then it will be responsible for defining the schema by processing the data inside the Table.

## Using cached query results

bookmark_border

BigQuery writes all query results to a table. The table is either explicitly identified by the user (a destination table), or it is a temporary, cached results table. Temporary, cached results tables are maintained per-user, per-project. There are no storage costs for temporary tables, but if you write query results to a permanent table, you are charged for storing the data.

All query results, including both interactive and batch queries, are cached in temporary tables for approximately 24 hours with some exceptions.

#### Limitations

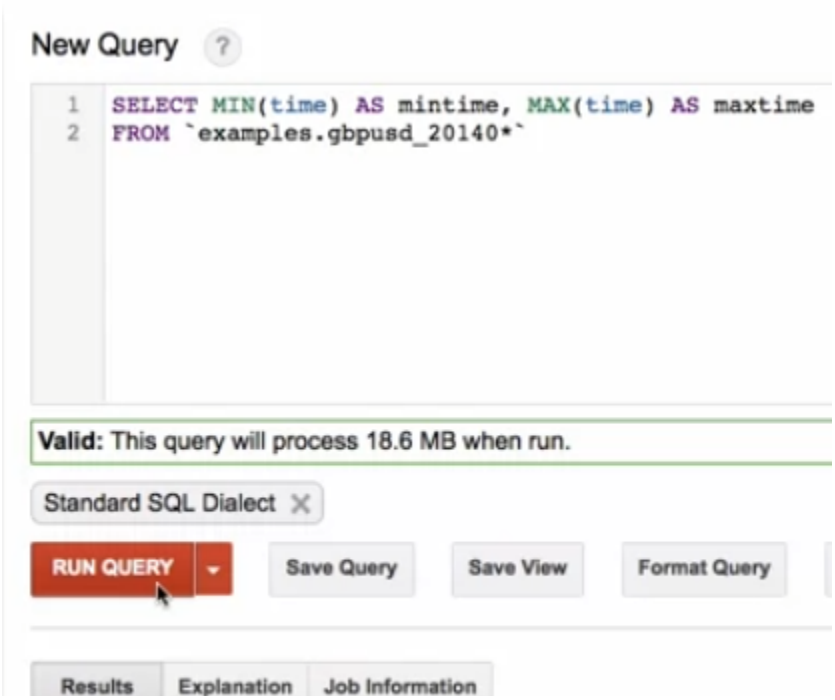
Using the query cache is subject to the following limitations:

- When you run a duplicate query, BigQuery attempts to reuse cached results. To retrieve data from the cache, the duplicate query text must be the same as the original query.
- For query results to persist in a cached results table, the result set must be smaller than the maximum response size. For more information about managing large result sets, see Returning large query results.
- You cannot target cached result tables with DML statements.
- Although current semantics allow it, the use of cached results as input for dependent jobs is discouraged. For example, you should not submit query jobs that retrieve results from the cache table. Instead, write your results to a named destination table. To enable easy cleanup, features such as the dataset level `defaultTableExpirationMs` property can expire the data automatically after a given duration.

**Optimizing**

1. Reduce the amount of Data being processed
2. Use SELECT [COLUMNS,] instead of SELECT *
3. Use Time range to limit the data using BETWEEN clauses. More efficient if the table is partitioned using the date column
4. Split the data in to separate tables to reduce table scan

5. Use wild cards to read from multiple tables



| Row | mintime | maxtime |
|---|---|---|
| 1 | 2014-01-01 00:00:00 UTC | 2014-02-28 23:59:57 UTC |

## Disadvantages of Separate Tables

- Have to split data somehow
- Query performance gets worse as number of tables increases
- Limit of 1,000 tables in a single query
  - 3 years takes up 1,095 daily tables

## Dividing Data into Daily Partitions

- Ingestion time
  - BigQuery puts new data in the partition for the current day
  - Creates a pseudo-column called _PARTITIONTIME
- Time-unit column
  - Tell BigQuery which column contains the date, and it will put data in the right partition

If you're starting fresh and want to stream new data into a table every day, then it's very easy. You can create an ingestion-time partitioned table. It wouldn't be very useful to upload our existing data to an ingestion-time partitioned table, though, because all of the data would go into today's partition regardless of what date was in each record. **The solution is to create a time-unit column-partitioned table. You just need to tell BigQuery which column contains the date, and it will put each data record into the right partition.**

You can partition BigQuery tables by:

- **Time-unit column**: Tables are partitioned based on a `TIMESTAMP`, `DATE`, or `DATETIME` column in the table.
- **Ingestion time**: Tables are partitioned based on the timestamp when BigQuery ingests the data.
- **Integer range**: Tables are partitioned based on an integer column.

**READ:** Introduction to partitioned tables | BigQuery | Google Cloud

https://cloud.google.com/bigquery

**Use clustering under the following circumstances:**

- You don't need strict cost guarantees before running the query.
- You need more granularity than partitioning alone allows. To get clustering benefits in addition to partitioning benefits, you can use the same column for both partitioning and clustering.
- Your queries commonly use filters or aggregation against multiple particular columns.
- The cardinality of the number of values in a column or group of columns is large.

**Use partitioning under the following circumstances:**

- You want to know query costs before a query runs. Partition pruning is done before the query runs, so you can get the query cost after partitioning pruning through a dry run. Cluster pruning is done when the query runs, so the cost is known only after the query finishes.
- You need partition-level management. For example, *you want to set a partition expiration time, load data to a specific partition, or delete partitions.*
- You want to specify how the data is partitioned and what data is in each partition. For example, you want to define time granularity or define the ranges used to partition the table for integer range partitioning.

**Prefer clustering over partitioning under the following circumstances:**

- Partitioning results in a small amount of data per partition (approximately less than 1 GB).
- Partitioning results in a large number of partitions beyond the limits on partitioned tables.
- Partitioning results in your mutation operations modifying most partitions in the table frequently (for example, every few minutes).

#### Partitioning versus sharding

Table sharding is the practice of storing data in multiple tables, using a naming prefix such as `[PREFIX]_YYYYMMDD`.

Partitioning is recommended over table sharding, because partitioned tables perform better. With sharded tables, BigQuery must maintain a copy of the schema and metadata for each table. BigQuery might also need to verify permissions for each queried table. This practice also adds to query overhead and affects query performance.

#### Limitations

You cannot use legacy SQL to query partitioned tables or to write query results to partitioned tables.

**BigQuery does not support partitioning by multiple columns. Only one column can be used to partition a table.**

*Time-unit column-partitioned tables are subject to the following limitations:*

- The partitioning column must be either a scalar `DATE`, `TIMESTAMP`, or `DATETIME` column.
- While the mode of the column can be `REQUIRED` or `NULLABLE`, it cannot be `REPEATED` (array-based).
- The partitioning column must be a top-level field. You cannot use a leaf field from a `RECORD` (`STRUCT`) as the partitioning column.

*Integer-range partitioned tables are subject to the following limitations:*

- The partitioning column must be an `INTEGER` column. While the mode of the column may be `REQUIRED` or `NULLABLE`, it cannot be`REPEATED` (array-based).
- The partitioning column must be a top-level field. You cannot use a leaf field from a `RECORD` (`STRUCT`) as the partitioning column.

#### Set the partition expiration

When you create a table partitioned by ingestion time or time-unit column, you can specify a partition expiration. This setting specifies how long BigQuery keeps the data in each partition. The setting applies to all partitions in the table but is calculated independently for each partition based on the partition time.

A partition's expiration time is calculated from the partition boundary in UTC. For example, with daily partitioning, the partition boundary is at midnight (00:00:00 UTC). If the table's partition expiration is 6 hours, then each partition expires at 06:00:00 UTC the following day. When a partition expires, BigQuery deletes the data in that partition. **If you set a table expiration, that value takes precedence over the partition expiration. For example, if the table expiration is set to 5 days, and the partition expiration is set to 7 days, then the table and all partitions in it are deleted after 5 days.**

#### Set partition filter requirements

When you create a partitioned table, you can require that all queries on the table must include a predicate filter (a `WHERE` clause) that filters on the partitioning column. This setting can improve performance and reduce costs, because BigQuery can use the filter to prune partitions that don't match the predicate.

For information on adding the **Require partition filter** option when you create a partitioned table, see Creating partitioned tables.

If a partitioned table has the **Require partition filter** setting, then every query on that table must include at least one predicate that only references the partitioning column. Queries without such a predicate return the following error:

`Cannot query over table 'project_id.dataset.table' without a filter that can be used for partition elimination`.

```
ALTER TABLE mydataset.mypartitionedtable
  SET OPTIONS (
    require_partition_filter = true);
```

#### Write data to a specific partition

You can load data to a specific partition by using the `bq load` command with a partition decorator. The following example writes data into the `20160501` (May 1, 2016) partition of an existing table, assuming the table is already partitioned by date:

```
bq query \
  --use_legacy_sql=false  \
  --destination_table='my_table$20160501' \
  --append_table=true \
  'SELECT * FROM my_dataset.another_table'
```

#### Query Partitioned Data

With ingestion-time partitioning, you can use this technique to load older data into the partition that corresponds to the time when the data was originally created.

For example, the following two queries are equivalent. Depending on the table size, the second query might perform better, because it places `_PARTITIONTIME` by itself on the left side of the > operator. Both queries process the same amount of data.

```
-- Might be slower.
SELECT
  field1
FROM
  dataset.table1
WHERE
  TIMESTAMP_ADD(_PARTITIONTIME, INTERVAL 5 DAY) > TIMESTAMP("2016-04-
15");

-- Often performs better.
SELECT
  field1
FROM
  dataset.table1
WHERE
  _PARTITIONTIME > TIMESTAMP_SUB(TIMESTAMP('2016-04-15'), INTERVAL 5
DAY);
```

To limit the partitions scanned, don't include any other columns in a `_PARTITIONTIME` filter. For example, the following query does not limit the scanned partitions, because `field1` is a column in the table.

```
-- Scans all partitions of table2. No pruning.SELECT
  field1FROM
  dataset.table2WHERE
  _PARTITIONTIME + field1 = TIMESTAMP('2016-03-28');
```

If you often query a particular range of times, consider creating a view that filters on the `_PARTITIONTIME` pseudo-column. For example, the following statement creates a view that includes only the most recent seven days of data from a table named `dataset.partitioned_table`:

```
-- This view provides pruning.CREATE VIEW dataset.past_week AS
  SELECT *
  FROM
    dataset.partitioned_table
  WHERE _PARTITIONTIME BETWEEN
    TIMESTAMP_TRUNC(TIMESTAMP_SUB(CURRENT_TIMESTAMP, INTERVAL 7 * 24
HOUR), DAY)
    AND TIMESTAMP_TRUNC(CURRENT_TIMESTAMP, DAY);
```

To query data in the `__UNPARTITIONED__` partition, use the `_PARTITIONTIME` pseudo-column with the `NULL` value. For example:

```
SELECT
  columnFROM dataset.tableWHERE
  _PARTITIONTIME IS NULL
```

#### Best practices for partition pruning

To limit the partitions that are scanned in a query, use a constant expression in your filter. If you use dynamic expressions in your query filter, BigQuery must scan all of the partitions. For example, the following query prunes partitions because the filter contains a constant expression:

```
SELECT
  t1.name,
  t2.categoryFROM
  table1 AS t1INNER JOIN
  table2 AS t2ON t1.id_field = t2.field2WHERE
  t1.ts = CURRENT_TIMESTAMP()
```

However, the following query doesn't prune partitions, because the filter, `WHERE t1.ts = (SELECT timestamp from table where key = 2)`, is not a constant expression; it depends on the dynamic values of the `timestamp` and `key` fields:

```
SELECT
  t1.name,
  t2.categoryFROM
  table1 AS t1INNER JOIN
  table2 AS t2ON
  t1.id_field = t2.field2WHERE
  t1.ts = (SELECT timestamp from table3 where key = 2)
```

#### Inserting data

When you use a DML statement to add rows to an ingestion-time partitioned table, you can specify the partition to which the rows should be added. You reference the partition using the `_PARTITIONTIME` pseudo column.

```
INSERT INTO
  project_id.dataset.mytable (_PARTITIONTIME,
    field1,
    field2)
SELECT
  TIMESTAMP("2017-05-01"),
  1,
  "one"
```

*Inserting data into partitioned tables*

Inserting data into a partitioned table using DML is the same as inserting data into a non-partitioned table.

For example, the following `INSERT` statement adds rows to partitioned table `mycolumntable` by selecting data from `mytable2` (a non-partitioned table).

```
INSERT INTO
  project_id.dataset.mycolumntable (ts,
    field1)
SELECT
```

```
  ts,
  id
FROM
  project_id.dataset.mytable2
```

#### Deleting data

*Deleting data in ingestion-time partitioned tables*

The following `DELETE` statement deletes all rows from the June 1, 2017, partition (`"2017-06-01"`) of `mytable` where `field1` is equal to `21`. You reference the partition using the `_PARTITIONTIME` pseudo column.

```
DELETE
  project_id.dataset.mytableWHERE
  field1 = 21
  AND _PARTITIONTIME = "2017-06-01"
```

*Deleting data in partitioned tables*

Deleting data in a partitioned table using DML is the same as deleting data from a non-partitioned table.

For example, the following `DELETE` statement deletes all rows from the June 1, 2017, partition (`"2017-06-01"`) of `mycolumntable` where `field1` is equal to `21`.

```
DELETE
  project_id.dataset.mycolumntableWHERE
  field1 = 21
  AND DATE(ts) = "2017-06-01"
```

#### Updating data

*Updating data in ingestion-time partitioned tables*

**The following `UPDATE` statement moves rows from one partition to another.** Rows in the May 1, 2017 partition (`“2017-05-01”`) of `mytable` where `field1` is equal to `21` are moved to the June 1, 2017 partition (`“2017-06-01”`).

```
UPDATE
  project_id.dataset.mytableSET
  _PARTITIONTIME = "2017-06-01"WHERE
  _PARTITIONTIME = "2017-05-01"
  AND field1 = 21
```

*Updating data in partitioned tables*

Updating data in a partitioned table using DML is the same as updating data from a non-partitioned table. For example, the following `UPDATE` statement moves rows from one partition to another. Rows in the May 1, 2017 partition (`“2017-05-01”`) of `mytable` where `field1` is equal to `2` are moved to the June 1, 2017 partition (`“2017-06-01”`).

```
UPDATE
  project_id.dataset.mycolumntableSET
  ts = "2017-06-01"WHERE
  DATE(ts) = "2017-05-01"
  AND field1 = 21
```

#### Clustering

When you create a clustered table in BigQuery, the table data is automatically organized based on the contents of one or more columns in the table's schema. The columns you specify are used to collocate related data. When you cluster a table using multiple columns, the order of columns you specify is important. The order of the specified columns determines the sort order of the data.

Clustering can improve the performance of certain types of queries such as queries that use filter clauses and queries that aggregate data. When data is written to a clustered table by a query job or a load job, BigQuery sorts the data using the values in the clustering columns. These values are used to organize the data into multiple blocks in BigQuery storage. When you submit a query that contains a clause that filters data based on the clustering columns, BigQuery uses the sorted blocks to eliminate scans of unnecessary data. You might not see a significant difference in query performance between a clustered and unclustered table if the table or partition is under 1 GB.

Similarly, when you submit a query that aggregates data based on the values in the clustering columns, performance is improved because the sorted blocks colocate rows with similar values

#### Clustering partitioned tables

In a partitioned table, data is stored in physical blocks, each of which holds one partition of data. A partitioned table maintains these properties across all operations that modify it: query jobs, data manipulation language (DML) statements, data definition language (DDL) statements, load jobs, and copy jobs. This requires BigQuery to maintain more metadata than a unpartitioned table. As the number of partitions increases, the number of metadata overhead increases.

Although more metadata must be maintained, by ensuring that data is partitioned globally, BigQuery can more accurately estimate the bytes processed by a query before you run it. This cost calculation provides an upper bound on the final cost of the query.

In a clustered table, BigQuery automatically sorts the data based on the values in the clustering columns and organizes them in optimally sized storage blocks. You can achieve more finely grained sorting by creating a table that is clustered and partitioned. A clustered table maintains the sort properties in the context of each operation that modifies it. As a result, BigQuery might not be able to accurately estimate the bytes processed by the query or the query costs. When blocks of data are eliminated during a query, BigQuery provides a best effort reduction of the query costs.

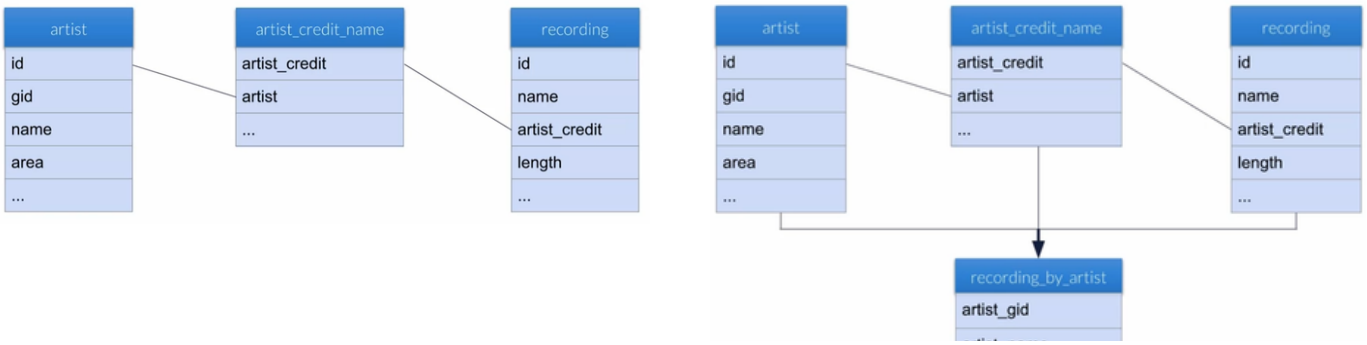
#### Denormalization

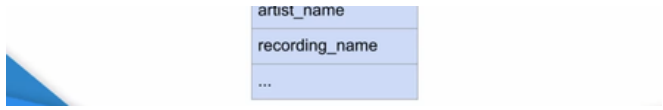
- Normalization
  - Removes redundancy from data
  - Improves data integrity
- Denormalization in BigQuery
  - Can make queries much faster
  - Data integrity is not an issue because data is not being modified

Denormalization can make your queries much faster, and because the downsides of having redundant data are not a problem for what we're doing. First, data integrity is not an issue because BigQuery isn't a transaction processing system. We're just using it for reporting, not for recording transactions, so we're not going to create inconsistencies in the data, since we're not going to modify the data at all.

Second, the extra cost associated with storing and querying extra copies of data is often outweighed by the much faster query times. Why does denormalization make queries so much faster? Because when you query a normalized database, you will often have to join multiple tables together, but joining tables is a very time-consuming operation. It's not very noticeable when the tables are small, but as the tables get larger, the time required to join them increases exponentially.

**Example**

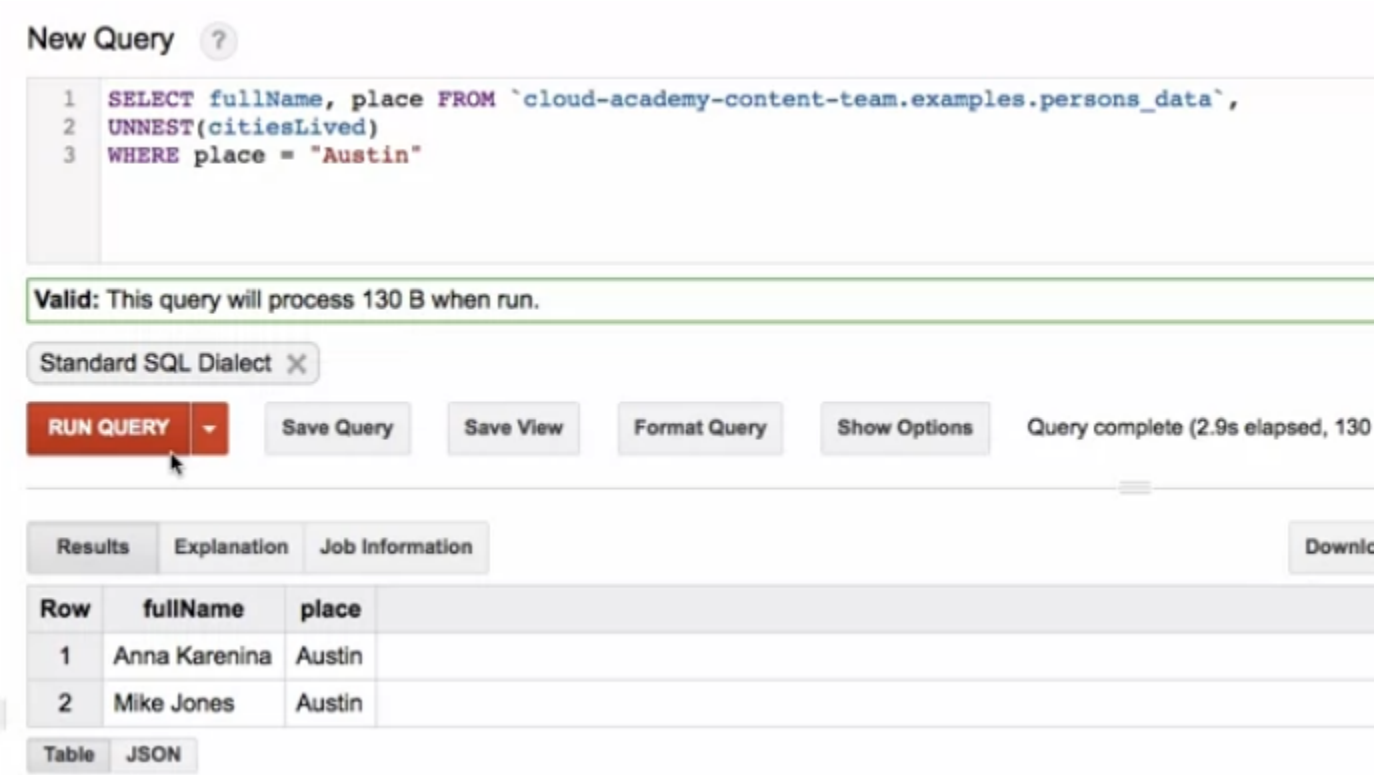





Nested Repeated Fields

| | | | |
|---|---|---|---|
| phoneNumber | RECORD | NULLABLE | Describe this field... |
| phoneNumber.areaCode | INTEGER | NULLABLE | Describe this field... |
| phoneNumber.number | INTEGER | NULLABLE | Describe this field... |
| children | RECORD | REPEATED | Describe this field... |
| children.name | STRING | NULLABLE | Describe this field... |
| children.gender | STRING | NULLABLE | Describe this field... |
| children.age | INTEGER | NULLABLE | Describe this field... |
| citiesLived | RECORD | REPEATED | Describe this field... |
| citiesLived.place | STRING | NULLABLE | Describe this field... |
| citiesLived.yearsLived | INTEGER | REPEATED | Describe this field... |

Add New Fields




- To refer to a **nested field**, you can use the **dot notation** that it shows in the schema. For example, to get a list of all of the people in the table and their phone numbers, you would run *phonenumber.number*
- To query a **nested repeated field,** so you have to use the **UNNEST** function. First, put in a comma after the table name. Then unnest citiesLived. Now, we can refer to the field as just “place” instead of citiesLived.place. **UNNEST** is actually doing a JOIN. If you find that you frequently need to unnest the same repeated fields in your queries, then you might want to create a view.

### *Nested and repeated fields*

BigQuery's internal storage format supports both nested fields and repeated fields.

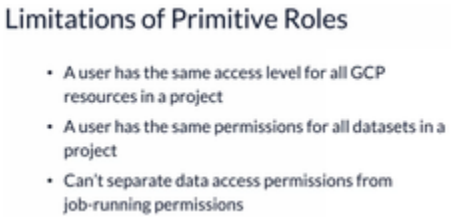
- A *nested field* is a complex data type that contains one or more strongly typed subfields. In Google Standard SQL, nested fields are represented by the STRUCT data type.
- A *repeated field* is an ordered list of zero or more values, all with the same data type. In Google Standard SQL, a repeated field is represented as an ARRAY data type.

For more information, see Specify nested and repeated columns in table schemas.

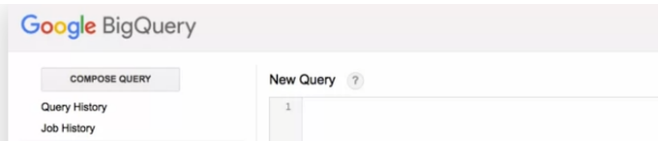
A repeated, nested field is an efficient way to store denormalized data in BigQuery. For example, suppose that a customer order can have multiple line items. The line items contain information such as product name, quantity, and total cost. In a traditional RDBMS, you might create an `Orders` table and an `OrderItems` table, and join them with a foreign key.

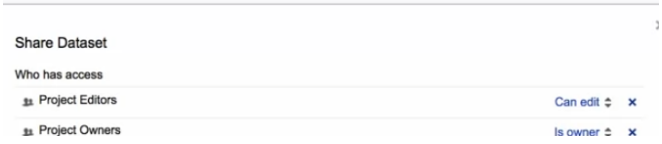
In BigQuery, you could represent the line items as an array of nested fields inside the `Orders` table. Running a query on a single table with repeated, nested fields is often more efficient than joining multiple tables. For historical data, such as customer transactions, you can denormalize the data as part of an ETL or ELT process when loading the data into BigQuery.

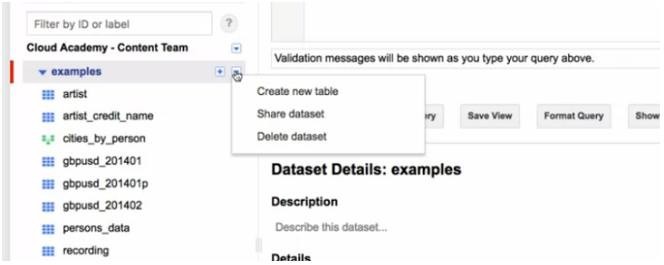
**Access Control**

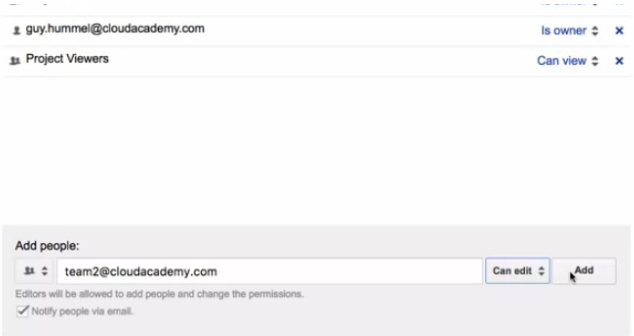


| Capability | dataViewer | dataEditor | dataOwner | user | jobUser | admin |
|---|---|---|---|---|---|---|
| List/get projects | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| List tables | ✓ | ✓ | ✓ | ✓ | ⊘ | ✓ |
| Get tables | ✓ | ✓ | ✓ | ⊘ | ⊘ | ✓ |
| Read tables/metadata | ✓ | ✓ | ✓ | ⊘ | ⊘ | ✓ |
| Modify/delete tables | ⊘ | ✓ | ✓ | ⊘ | ⊘ | ✓ |
| List/get datasets | ✓ | ✓ | ✓ | ✓ | ⊘ | ✓ |
| Create new datasets | ⊘ | ✓ | ✓ | ✓ | ⊘ | ✓ |
| Modify/delete datasets | ⊘ | ⊘ | ✓ | ⊘ | ⊘ | ✓ |
| Create jobs/queries | ⊘ | ⊘ | ⊘ | ✓ | ✓ | ✓ |
| Cancel jobs | ⊘ | ⊘ | ⊘ | Self-created jobs | Self-created jobs | Any jobs |
| Get/list saved queries | ⊘ | ⊘ | ⊘ | ✓ | ⊘ | ✓ |
| Create/update/delete saved queries | ⊘ | ⊘ | ⊘ | ⊘ | ⊘ | ✓ |





You might be wondering why you would want to separate data access permissions from job-running permissions because it would seem like most users would need both. That's true in many cases, so remember that you have to assign both types of roles to users in order for them to look at the data and run queries (unless you're using primitive roles). But there are situations when you might want to give only one or the other. For example, if you have an application that monitors the size of your tables, then you might want to assign only the BigQuery Data Viewer role to its service account. That way, even if the application developers accidentally make a change to the program that would cause a query to run, it will be disallowed. Why would that matter? Because queries incur a cost and a program could potentially run up some hefty charges if there's a bug.

## Authorized View

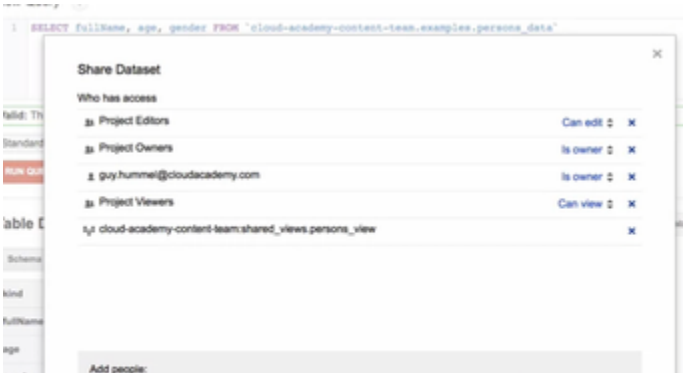
- Gives access to the results of a query without giving access to the tables that were queried
- Can be used to restrict access to particular columns or rows

## Creating an Authorized View

- Create a separate dataset to store the view
- Create the view in the new dataset
- Give the group read access to the dataset containing the view
- Authorize the view to access the source dataset

## Authorized View Example

- Give team1 access to only the name, age, and gender fields in the persons_data table
- Requires permission to assign roles to other users



**Now users in team1 will be able to run queries on this view, even though they don't themselves have access to the persons_data table.**

## Creating Basic App Engine App

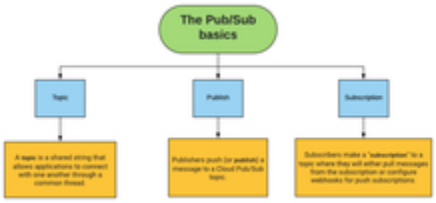
1. Create app engine

```
# create a basic app
gcloud app create --project=$PROJECT_ID
gcloud app describe
gcloud components install app-engine-python

# local run on port 8080 - localhost:8080 will show 'Hello World'
on the browser
dev_appserver.py app.yaml.

# to deploy the code and create google web service
glcoud app deploy
gcloud app browse
```

## Pub Sub Basics




While Dataflow pipelines (including ones built with Dataflow Templates) get the job done well, sometimes they can be more than what is needed for use cases that simply require raw data with no transformation to be exported to BigQuery.

Luckily, GCP announced a new Subscriber Type called **BigQuery Subscription** which writes data directly from Pub/Sub to BigQuery. This new extract, load, and transform (ELT) path will be able to simplify your event-driven architecture. For Pub/Sub messages where advanced preload transformations or **data processing before landing** data in BigQuery (such as masking PII) is necessary, you would still need Dataflow in the middle though

**Use cases**

- Simplify streaming ELT pipeline using BigQuery Subscriptions with Pub/Sub when no transformation is required prior loading
- GCP Logs Router: Route GCP logs to Pub/Sub
- Sending emails natively from Google Cloud Pub/Sub events. Google Cloud normally recommends using tools such as [SendGrid to send emails from Cloud Functions triggered by Cloud Pub/Sub](#).

Pub/Sub clients publish messages to **topics**.

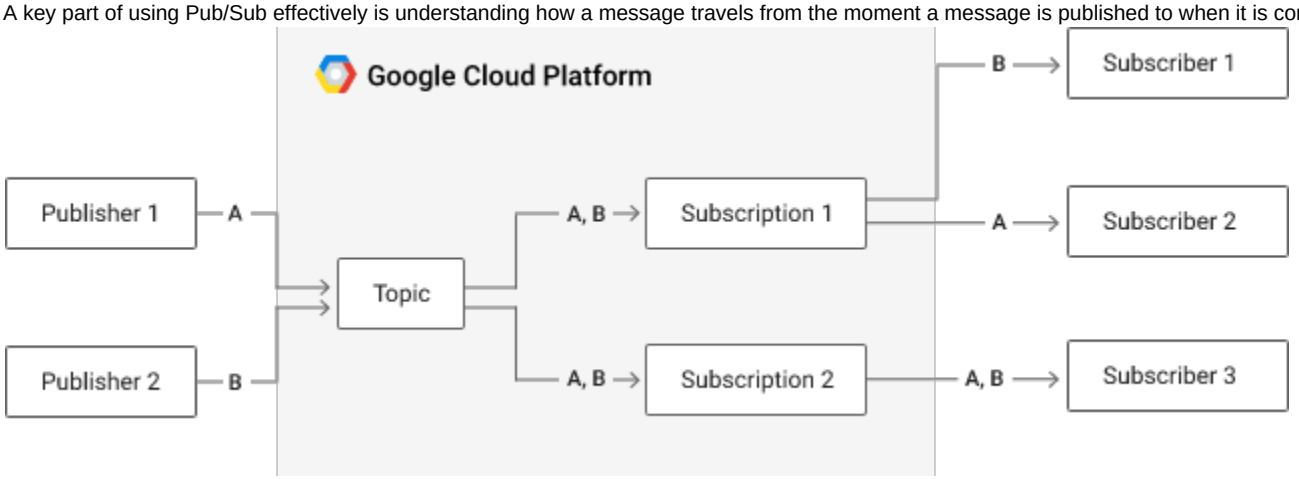
Pub/Sub clients receive messages from **subscriptions**. You can create multiple subscriptions for the same topic, in which case each subscription gets a copy of all of the messages in the topic. Some beginners assume that one would pull messages from a **topic**, which isn't possible — **topics** are for publishing only. The **subscription** is where you will receive all messages.

Your application can create multiple **Pub/Sub clients** to publish to a topic, across different processes. (With some caveats mentioned below.)

Your application can also create multiple **subscriber clients** for a single subscription, in which case message delivery is load balanced across all of them. Messages are delivered to one client on the **subscription**, not all of them.

### What is the lifecycle of a message in Pub/Sub?

A key part of using Pub/Sub effectively is understanding how a message travels from the moment a message is published to when it is consumed.




First, a publisher sends a message to the topic. Once the Pub/Sub server receives the message, it adds fields to the message such as publish timestamp and message ID. Pub/Sub delivers this message to all subscriptions. From each subscription, subscribers pull messages via one of two pull calls (more on this in a bit). Each subscription has an acknowledgement deadline, which determines how much time is allocated for the client to **ack**nowledge (**ack**) this message before the server redelivers it. If the client holds the message beyond the agreed deadline without acking, or if the client sends a **nack** (**n**egative **ack**nowledgement) request, the message will be redelivered (potentially to a different subscriber on the same subscription). Sometimes, the acknowledgement deadline is not enough time for a message to be acked by the client, so a **ModifyAckDeadline** request is used to extend the deadline

**More READ: Things I wish I knew about Google Cloud Pub/Sub: Part 2 | by Alex Hong | Google Cloud - Community | Medium**

### Ordering Basics

Ordering in Cloud Pub/Sub consists of two properties. The first is the ordering key set on a message when publishing. This string — which can be up to 1KB — represents the entity for which messages should be ordered. For example, it could be a user ID or the primary key of a row in a database. The second property is the enable_message_ordering property on a subscription. When this property is true, subscribers receive messages for an ordering key in the order in which they were received by the service.

### Ordering Properties

Ordered delivery has three main properties:

1. **Order**: When a subscription has message ordering enabled, subscribers receive messages published in the same region with the same ordering key in the order in which they were received by the service.
2. **Consistent redelivery**: If a message is redelivered, then all messages received after that message for the same ordering key will also be redelivered, whether or not they were already acknowledged.
3. **Affinity**: If there are messages with an ordering key outstanding to a streaming pull subscriber, then additional messages that are delivered are sent to that same subscriber. If no messages are currently outstanding for an ordering key, the service delivers messages to the last subscriber to receive messages for that key on a best-effort basis

**More READ: Google Cloud Pub/Sub Ordered Delivery | by Kamal Aboul-Hosn | Google Cloud - Community | Medium**

Following are the key features of Pub/Sub.

- Scalable and fully managed
- Global messaging
- pull and push modes.
- In-order delivery (Beta).
- At-least-once delivery
- Dead letter topic: Messages unable to be processed by subscriber applications to be put aside for offline debugging.
- Filtering: Pub/Sub can filter messages based upon attributes in order to reduce delivery volume to subscribers.

> *The core difference between Cloud Pub/Sub and Cloud Tasks is the notion of **implicit vs explicit** invocation*

**More READ: Cloud Tasks or Pub/Sub ?. Asynchronous communication in… | by Vikram Shinde | Google Cloud - Community | Medium**

## Data Flow

https://www.udemy.com/course/apache-beam-a-hands-on-course-to-build-big-data-pipelines/

https://beam.apache.org/releases/pydoc/current/

# Job Application CV

## September 2022

Krishnendu_Das...esume 2022.pdf

Krishnendu_Das_Resume.docx

# Migration

## On Prem Hadoop Cluster to GCP

***Parquet vs Avro***

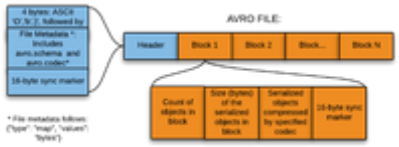
***Read: How to load data efficiently in BigQuery | by Jinho Seo | Medium***

*AVRO is a row-based storage format, whereas PARQUET is a columnar-based storage format. Avro is efficient and rich in visual expression, so it can be recommended in a convenient format. Avro uses self-describing binary files that can be split into blocks and compressed block-by-block. This format has the advantage of being able to parallelize loading data from Avro files and exporting data to Avro files as below figure. There are two downsides. One is that it is not human readable. If readability and expressiveness are important to you, you should use a newline-delimited JSON file to store your data. The second downside is that Avro files are stored line by line. This makes Avro files inefficient for federated queries.*

*Avro files are stored row by row, while Parquet files are stored column by column. Columnar files are optimized for reading subsets of columns. The column format is somewhat less efficient when loading data, as all columns must be read to load data. Therefore, if you are using columnar format and want to use federated queries in BigQuery, I recommend the Parquet file format over Avro.*

*If your network is slow, or if you have a lot of files or very large files, you can use **gsutil cp** to set up a multithreaded upload of your data. Also, if your data is all in Google Cloud Storage, you can invoke **bq load** on the Cloud Storage.*

***Conclusion:** Avro is recommended if you plan to load data into BigQuery and delete files when you select any file format. I also recommend using Parquet if you want to persist files for federated queries. JSON is recommended if human readability is important, and you want a small file size.*




**How To?**

**Read:**

1. **Hadoop to BigQuery Migration — New Edition | by Vladimir Elvov | Medium**

2. **Migrating HDFS Data from On-Premises to Google Cloud | Migrating Hadoop to GCP**

3. Introduction to loading data | BigQuery | Google Cloud

Migrations are made up of two parts: The initial load and the delta load.

1. The first part is where historical data is loaded into BigQuery
2. The second one is used for a continuous replication of data between the on-prem cluster and the GCP whilst data is still ingested from the sources to the on-prem cluster

There are two reasons for this two-phased approach.

- One is that doing both the initial load and then switching to the cloud-based data pipeline immediately is a Big Bang approach and this is very risky. Generally speaking, migrations are complicated and the more you can reduce the risk the better for your data, your data platform, your data consumers
- The second reason is that if you don't do a Big Bang approach, then you need to get continuous refreshes of data to your new cloud-based pipeline
- This delta load can be done by accessing the sources from the cloud right away, however, this would double the load to the source system (as the on-prem cluster also needs this data). If this source system is a relational database, this is a bad idea as its capacity is limited and you might break down the entire database. This is where the continuous replication from the on-prem cluster comes in handy
- ***If* you're ingesting data from a system that is designed to allow parallel consumption like Apache Kafka, you don't face such an issue and can consume the data directly from the Kafka topic through a cloud-based Kafka consumer**

### Phase 1 - Initial Load

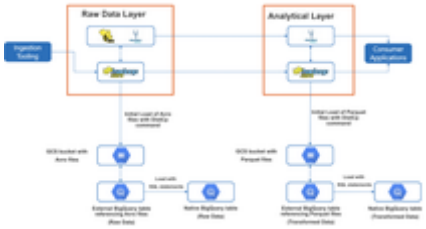
In the initial load data is first migrated and loaded from both the raw data layer and the analytical layer seated in the source data warehouse

The data is persisted on-prem in HDFS, so you will copy the data first from HDFS to Google Cloud Storage (GCS) in the form of files. This can be done using the HDFS connector for GCS that is installed on the on-prem cluster nodes and then allows to run a **distcp** command that will copy files to a GCS bucket by specifying the respective bucket URI. Data arriving in GCS will be available in the form of files and now needs to be loaded into BigQuery. For this the **GCP offers bq load jobs**. However, if you want to load to a BigQuery table, which is partitioned by a specific column and also encrypted, you will have to create this table first. You can leverage the schema auto-detection in BigQuery and save some of your migration budget. **Data in GCS is available in Avro and Parquet formats — this means the files contain schema information themselves**. BigQuery is able to identify the schema in these files and create external tables on top of the files.

Besides just being able to query the data in files on GCS, BigQuery also supports Hive partitioning schemas in GCS. This means if you've had a partitioning schema by e.g., year, month and day on HDFS, then copied over the data into GCS with that exact folder structure, then BigQuery can pick up this schema and create additional columns for the partitions. Query the external table and do partition pruning as well. Since BigQuery charges you for the amount of data queried, this is a significant mechanism to both improve performance and reduce cost.

For the analytical layer it makes more sense to load the data to BigQuery native tables as this offers the best performance for analytical queries as compared to queries on external tables.

Let's assume that you want to have your data natively in BigQuery. Once you have an external table, you can now use a CREATE TABLE AS SELECT FROM SQL query to create a new BQ native table and move this data to this native table from the external one. It's possible to specify partitioning for the BQ native table and an encryption key for encrypting the data with a Customer Managed Encryption Key (CMEK). Partitioning is important to configure at this stage, because in BigQuery the table partitioning can only be defined at table creation time. With this we now have a partitioned BigQuery table with all data from the Impala table.

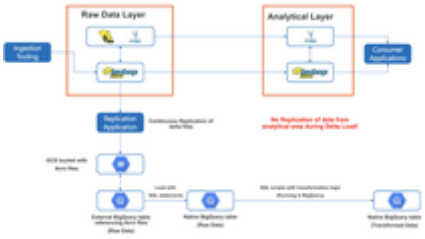



Use the `bq mkdef` command to create a table definition in JSON format for data stored in Cloud Storage or Drive.

Use the `bq mk` command to create a BigQuery resource.

Pay attention when using Avro files to ensure that using Avro logical types has been whitelisted for your project and the respective flag is set to true in the table definition file. Otherwise, you might face issues with your data, especially with timestamp columns. When using Parquet files, it's important to get the encoding of String fields right

### Phase 2 - Delta Load




Completing the initial load is a big step forward and will allow users to start working with the new cloud data platform. But they typically will want to have their data refreshed and this is where the Delta Load comes in. There are two main differences between the Initial Load and the Delta Load. The first is that data is now replicated only for the raw data layer, not for the analytical layer. This is done so that it is possible to build the actual data transformation pipeline directly on the cloud. In most cases these pipelines are SQL scripts in BigQuery, more complex transformations done on-prem in Spark can be migrated to Cloud Dataproc. Once it is clear that the output of the analytical layer both on-prem and in the cloud are the same, the new cloud-based pipeline is ready. The second difference is that since it has to be done in a continuous way instead of just once as with the initial load, using DistCp is sometimes not the best way ahead. Instead, it is better to use a solution that will be loading data through the GCS API. Typically, this is a simple replication app running either on-prem or in the cloud, depending on whether it's possible to push or pull data from on-prem. This will depend on your networking and security policies. Also, using DistCP might be required if the size of a single file is very large and a copy operation, which would require to assemble the file first from separate HDFS blocks before actually calling the GCS API to load the file, would take too long.

#### How do you identify the delta?

This heavily depends you're on-prem implementation, i.e., on how exactly data is ingested t the raw data layer. In some cases, the delta is made available in a staging directory in HDFS (often the case for truncate & load scenarios, see our previous story). In some, rather rare cases, a messaging mechanism is available that will publish the URI of a delta file on HDFS to e.g., a Kafka topic. In both these cases you can just fetch the file from HDFS and copy it to GCS through an API call. If the delta is not simply made available, then you need to calculate it. Typically, a table in a DWH-style Hadoop use case has a column with an insert-timestamp in it. Then it's required to run a query from the replication app to calculate the delta using the insert-timestamp column. The delta will be calculated by comparing the insert-timestamp to a timestamp value when the delta calculation was previously run. This timestamp value can be stored on a per-table basis in a log table. Once the delta is calculated, the replication app loads the delta through an API call to GCS. In the past we've typically used a simple Java app running on a Kubernetes cluster to do this replication.

#### How to load data from GCS to BigQuery?

Once the delta is available in GCS, the data needs to be loaded to BigQuery. This part depends on how you implement your use case. It depends on whether you need to transform the data before loading it to BigQuery and whether you have an append-only data model or want to run DML operations on it. The range of options goes from using external BigQuery tables (so no more loading of data to BigQuery native storage), overusing load jobs run through a Cloud Function, using an orchestration tool like Cloud Composer or leveraging heavy weight distributed compute frameworks like Spark or Beam/Dataflow. Generally speaking, it makes sense to **use the Spark or Beam/Dataflow only if** you need to perform transformations on data before loading it into BigQuery. Otherwise, using Cloud Functions or Cloud Composer to move data from GCS to native BigQuery tables is a viable approach as well. We will focus on the usage of Cloud Functions here.

#### Cloud Function Life Cycle

A Cloud Function is a serverless construct, i.e., you don't need to manage any infrastructure for it. Instead, when the triggering event is fired in our case a new file arrives in a GCS bucket it will run a piece of code. This piece of code can be a simple BigQuery load job that will load the contents of the replicated file from GCS to a BigQuery table in append only mode. If you want to replace truncate & load scripts because eventually you want to have a copy of a relational database table, then the DML functionality offered by BigQuery can be used out of the box. To do this, you would first load the delta file to a staging table in BigQuery and then run a Merge operation between the target table and the staging table. The lifecycle of such a Cloud Function would look as follows:

However, Cloud Functions can hit limits if you want to run DML operations and need to ensure ordering on data. The issue you could face is that data arrives in multiple files at the same time and these files contain data for the same id. Cloud Functions kick off immediately when their triggering event happens. If the file with a record for an id with a later point in time gets processed before a file with a record for the same id for an earlier point in time, you will end up with corrupted data which is out of order. If your ingestion mechanism to GCS can generate multiple files containing records for the same id at the same time, you should rather use sequential processing of the files on a time-based schedule, e.g. using Cloud Composer. For example, you could run the code shown below using the Python Operator and achieve DML functionality without data

inconsistencies using sequential processing. If on the other hand your ingestion mechanism can ensure that files will contain only one record per id in a batch load (e.g. when you have daily or hourly batch loads of deltas) then you can use Cloud Functions with DML safely without worrying about data inconsistencies.

Finally, it's important to point out that a Cloud Function triggered through a GCS event only guarantees at-least-once processing. This means, that in some rare cases, it is possible that a function will fire twice for the same file being written to GCS. If your data load process is idempotent (i.e. you use Upserts/DMLs) then this is not a problem. If you load data in append-only mode, then it is best to use a SQL query based transformation inside BigQuery in order to provide consistent data and remove potential duplicates.

## Effective Reading

Error rendering macro 'create-from-template' : Failed to find net.sf.hibernate.Session from the current thread

| Title | Creator | Modified |
|---|---|---|
| Advisory & Strategy | Krishnendu Das | about an hour ago |
| Data Engineering Design | Krishnendu Das | yesterday at 5:04 AM |
| Orchestration | Krishnendu Das | yesterday at 4:50 AM |

### Books on Database Technologies

| File | Modified |
|---|---|
| Learning PySpark.pdf | Oct 09, 2022 by Krishnendu Das |
| Building a Scalable Data Warehouse with Data Vault 2.0.pdf | Oct 09, 2022 by Krishnendu Das |

Drag and drop to upload or browse for files

Download All

### Orchestration

| File | Modified |
|---|---|
| Data Pipeline With Airflow.pdf<br>https://github.com/BasPH/data-pipelines-with-apache-airflow | yesterday at 4:52 AM by Krishnendu Das |

Drag and drop to upload or browse for files

### Data Engineering Design

| File | Modified |
|---|---|
| The-Delta-Lake-Series-Complete-Collection.pdf | yesterday at 5:04 AM by Krishnendu Das |

| | |
|---|---|
| Data Lake Architecture_ Designing the Data Lake and Avoiding the Garbage Dump.pdf | yesterday at 5:04 AM by Krishnendu Das |
| Build the Data Lake House Bill Inmon.pdf | yesterday at 5:05 AM by Krishnendu Das |
| Data Mesh_ Delivering Data-Driven Value at Scale.pdf | yesterday at 5:11 AM by Krishnendu Das |
| Lakehouse - A New Generation of Open Platforms that Unify.pdf | yesterday at 5:11 AM by Krishnendu Das |
| Data Engineering with Apache Spark, Delta Lake, and Lakehouse with Apache Spark.pdf | yesterday at 5:11 AM by Krishnendu Das |

Drag and drop to upload or browse for files

Download All

## Advisory & Strategy

| File | Modified |
|---|---|
| Data Management Body of Knowledge.pdf | about an hour ago by Krishnendu Das |
| WP_EN_DG_Talend_DefinitiveGuide_DataGovernance.pdf | about an hour ago by Krishnendu Das |
| Oreilly data-governance-the-definitive-guide.pdf | about an hour ago by Krishnendu Das |

Drag and drop to upload or browse for files

Download All

https://nicolejaneway.medium.com/15-data-strategy-books-6aef2a8b3472

# Data Warehousing

## Design Patterns

- ELT Design Patterns
- ETL Process Overview
- Modern Data Stack

https://dzone.com/articles/hadoop-vs-database-vs-cloud-dwh

https://www.firebolt.io/blog/the-real-meaning-of-a-data-lake

https://cloud.google.com/blog/products/data-analytics/unifying-data-lakes-and-data-warehouses-across-clouds-with-biglake

1. Data Mesh
   a. https://martinfowler.com/articles/data-monolith-to-mesh.html
   b. https://www.keboola.com/blog/data-mesh

   c. https://cloud.google.com/dataplex
   d. https://www.youtube.com/watch?v=1Kk7NE1RMVQ&ab_channel=nullQueries
2. Data Lake House
   a. https://www.striim.com/blog/a-brief-overview-of-the-data-lakehouse/
   b. https://cloud.google.com/biglake
   c. https://cloud.google.com/bigquery/docs/biglake-intro
   d. https://youtu.be/UdpNYk0EP0g
3. Data Lake
   a. http://cloudacademy.com
   b. https://cloud.google.com/learn/what-is-a-data-lake
4. Data Vault
   a. Overview
5. Data Mart
   a. Overview
6. *Delta Sharing*
   a. Overview
7. Delta Lake
   a. Overview

## Data Modeling

### Modeling using Data Build Tool

- https://www.getdbt.com/blog/conformed-dimensional-models/
- https://www.getdbt.com/coalesce-2020/kimball-in-the-context-of-the-modern-data-warehouse-whats-worth-keeping-and-whats-not/
- https://hub.getdbt.com/datavault-uk/dbtvault/latest/

### Data Vault 2.0

Read: resource

https://www.youtube.com/watch?v=PXJ0qhBis3c&t=6s&ab_channel=Datavault

https://www.youtube.com/watch?v=ElBMZHcy-w8&ab_channel=Datavault

https://www.youtube.com/watch?v=LwMtQluKSS0&ab_channel=Datavault

https://www.youtube.com/watch?v=MiU4Lz7Xoqo&ab_channel=Datavault

https://www.youtube.com/watch?v=l5UcUEt1IzM&ab_channel=ITkFunde

https://www.youtube.com/watch?v=KgW4YVZYpQk&ab_channel=ITandAnalytics

https://dbtvault.readthedocs.io/en/latest/tutorial/

### Slowly Changing Dimension

**What are the types of SCD?**
https://adatis.co.uk/introduction-to-slowly-changing-dimensions-scd-types/

Very simply, there are 6 types of Slowly Changing Dimension that are commonly used, they are as follows:

- Type 0 – Fixed Dimension
  - No changes allowed, dimension never changes
- Type 1 – No History
  - Update record directly, there is no record of historical values, only current state
- Type 2 – Row Versioning
  - Track changes as version records with current flag & active dates and other metadata
- Type 3 – Previous Value column
  - Track change to a specific attribute, add a column to show the previous value, which is updated as further changes occur
- Type 4 – History Table
  - Show current value in dimension table but track all changes in separate table
- Type 6 – Hybrid SCD
  - Utilize techniques from SCD Types 1, 2 and 3 to track change

Semi Structured & NoSQL Data Modeling

https://www.firebolt.io/blog/semi-structured-data-modeling

https://redis.com/blog/nosql-data-modeling/

https://phoenixnap.com/kb/nosql-data-modeling

Dimensional Modeling

# Open Source Tools

## Apache Beam

## Airflow

- Cross Downstream
- Wait For Downstream
- Wait For Completion
- Depend On Past
- Dag Run Timeout
- Dag Run Lockout
- Execution Timeout
- Airflow Pool
- Airflow Slots
- Weight Priority
- Weight Rule [*downstream, upstream]*
- Trigger Rules [ *all_success, all_failed, all_done, one_failed, one_success, none_failed, none_failed_or_skipped, none_skipped ]*
- Task SLA
- Email On Retry
- Email on Failure

**SAMPLE 1**

```
from airflow import DAG
from airflow.providers.postgres.operators.postgres import
PostgresOperator
from airflow.providers.http.sensors.http import HttpSensor
from airflow.providers.http.operators.http import SimpleHttpOperator
from airflow.operators.python import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook

import json
from pandas import json_normalize
from datetime import datetime

def _process_user(ti):
    user = ti.xcom_pull(task_ids="extract_user")
    user = user['results'][0]
    processed_user = json_normalize({
        'firstname': user['name']['first'],
        'lastname': user['name']['last'],
        'country': user['location']['country'],
        'username': user['login']['username'],
        'password': user['login']['password'],
        'email': user['email'] })
    processed_user.to_csv('/tmp/processed_user.csv', index=None,
header=False)
```

```
def _store_user():
    hook = PostgresHook(postgres_conn_id='postgres')
    hook.copy_expert(
        sql="COPY users FROM stdin WITH DELIMITER as ','",
        filename='/tmp/processed_user.csv'
    )

with DAG('user_processing', start_date=datetime(2022, 1, 1),
        schedule_interval='@daily', catchup=False) as dag:

    create_table = PostgresOperator(
        task_id='create_table',
        postgres_conn_id='postgres',
        sql='''
            CREATE TABLE IF NOT EXISTS users (
                firstname TEXT NOT NULL,
                lastname TEXT NOT NULL,
                country TEXT NOT NULL,
                username TEXT NOT NULL,
                password TEXT NOT NULL,
                email TEXT NOT NULL
            );
        '''
    )

    is_api_available = HttpSensor(
        task_id='is_api_available',
        http_conn_id='user_api',
        endpoint='api/'
    )

    extract_user = SimpleHttpOperator(
        task_id='extract_user',
        http_conn_id='user_api',
        endpoint='api/',
        method='GET',
        response_filter=lambda response: json.loads(response.text),
        log_response=True
    )

    process_user = PythonOperator(
        task_id='process_user',
        python_callable=_process_user
    )

    store_user = PythonOperator(
        task_id='store_user',
        python_callable=_store_user
    )
```

```
    create_table >> is_api_available >> extract_user >> process_user >>
store_user
```

**SAMPLE 2**

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.operators.subdag import SubDagOperator
from airflow.utils.task_group import TaskGroup

from random import uniform
from datetime import datetime

default_args = {
    'start_date': datetime(2020, 1, 1)
}

def _training_model():
    accuracy = uniform(0.1, 10.0)
    print(f'model\'s accuracy: {accuracy}')

def _choose_best_model():
    print('choose best model')

with DAG('xcom_dag', schedule_interval='@daily',
default_args=default_args, catchup=False) as dag:

    downloading_data = BashOperator(
        task_id='downloading_data',
        bash_command='sleep 3'
    )

    with TaskGroup('processing_tasks') as processing_tasks:
        training_model_a = PythonOperator(
            task_id='training_model_a',
            python_callable=_training_model
        )

        training_model_b = PythonOperator(
            task_id='training_model_b',
            python_callable=_training_model
        )

        training_model_c = PythonOperator(
            task_id='training_model_c',
            python_callable=_training_model
        )
```

```
choose_model = PythonOperator(
    task_id='task_4',
    python_callable=_choose_best_model
)

downloading_data >> processing_tasks >> choose_model
```

Common Operators:

- TriggerDagRunOperator
- https://github.com/tuanavu/airflow-tutorial/blob/v0.6/examples/gcloud-example/dags/bigquery_github/bigquery_github_trends.py

# Data Build Tool (DBT)

## DBT Certification Quick Reads

DBT - Notes.docx

# Containerization

# Advisory [ Governance & Strategy ]

# Snowflake - Cloud DWH

## Good Reads

11 Best Practices for Data Engineers (hushly.com)

Cloud Data Warehousing for Dummies (hushly.com)

# Databricks

# SQL Practice

1. https://stackoverflow.com/questions/2120544/how-to-get-cumulative-sum