**Google Cloud Data Engineer Certificate Study Guide**

**Exam Overview**

* Storage (20%)
  + GCS, Cloud SQL, DataStore, BigTable, BigQuery
* Big Data Processing (35%)
  + BigQuery, Dataflow, Dataproc, Datalab, Pub/Sub
* Machine Learning (18%)
  + ML APIs, TensorFlow
* Case Studies (15%)
* Others (Hadoop and Security about 12%)

What does a Data Engineer do?

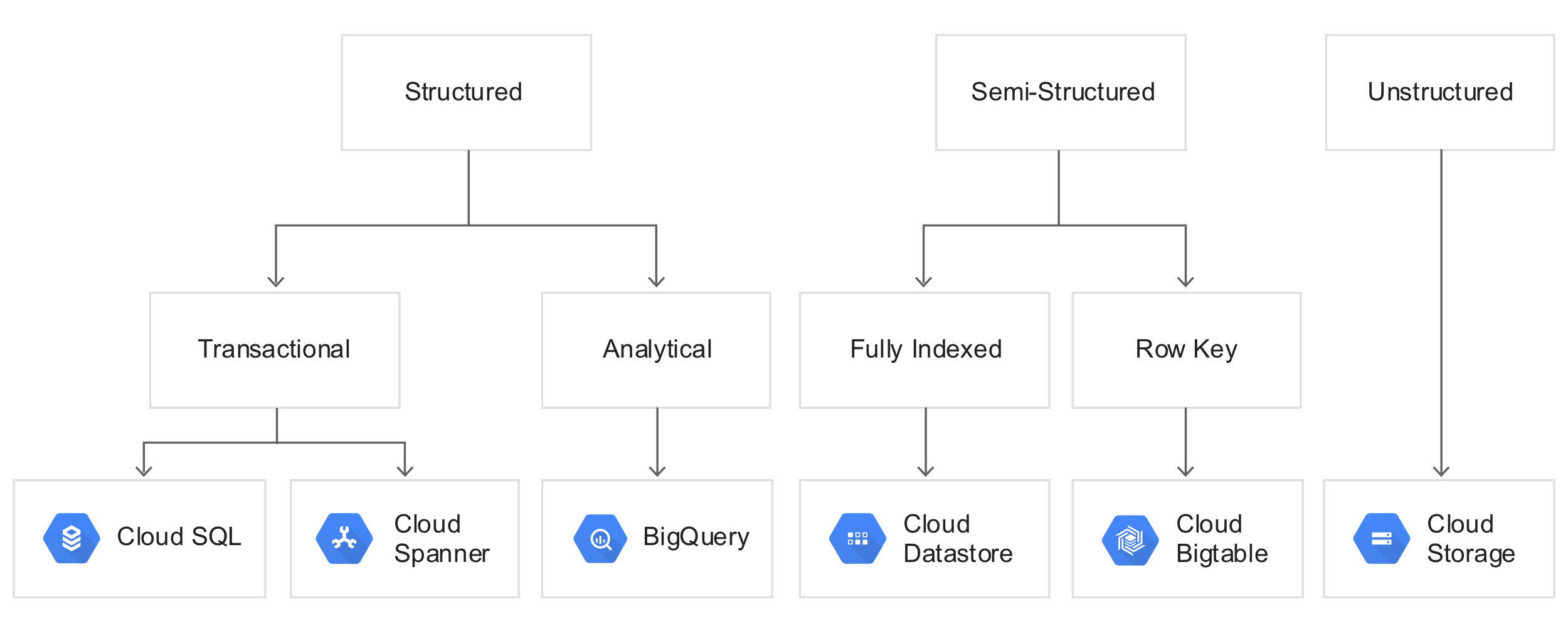
* Collect, store, manage, transform, and present data to make it useful

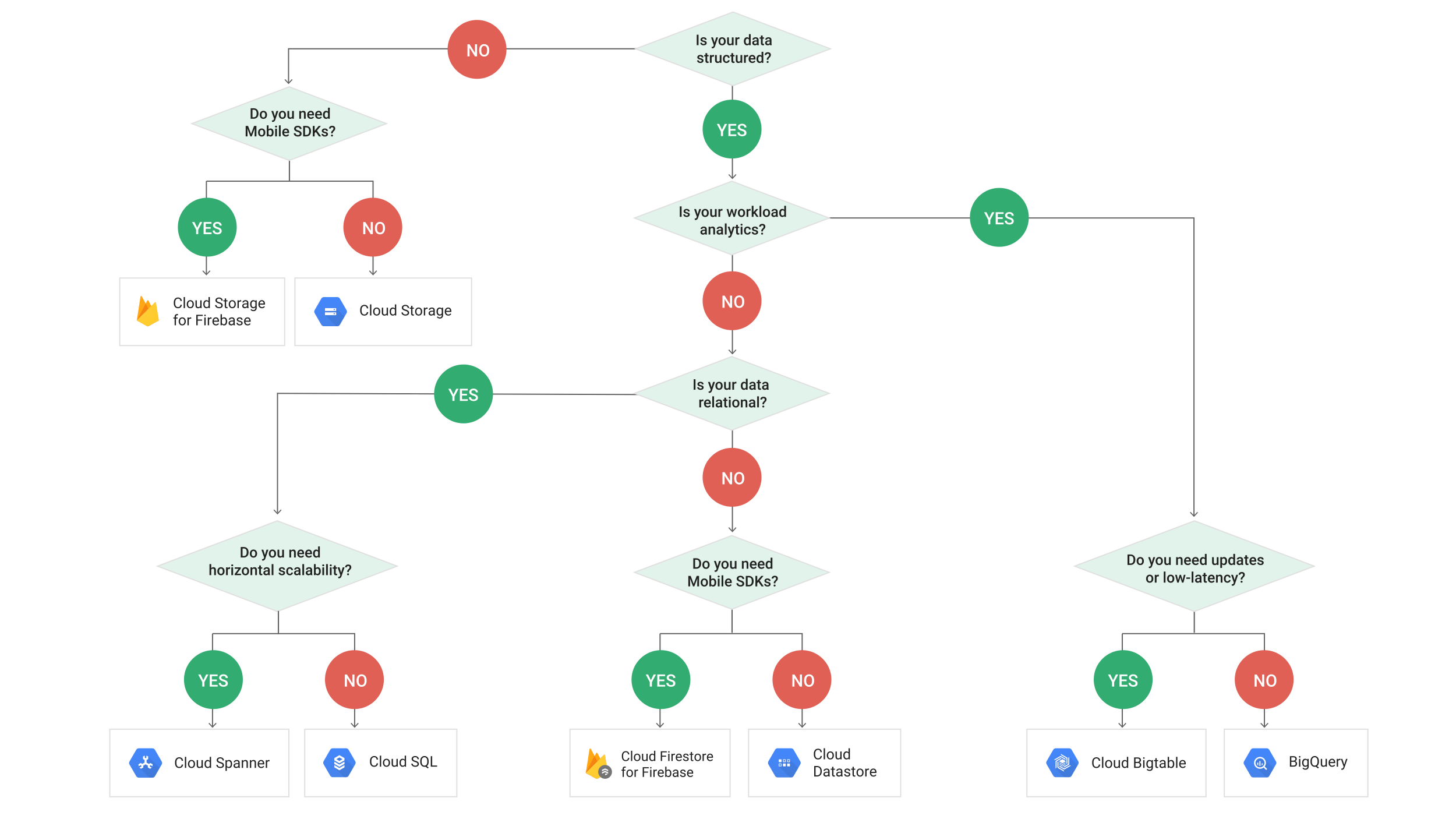
**Data Lifecycle**

**\*\*NOTE\*\* - The order of these do not matter exactly.**

* **Ingest**
  + Pull in raw data from a multitude of sources
    - Streaming data from devices
    - On-premise batch data
    - App logs
    - Mobile-app user events
    - Analytics
  + App
    - Stackdriver Logging
    - Cloud Pub/Sub
    - Cloud SQL
    - Cloud Datastore
    - Cloud Bigtable
    - Cloud Firestore
    - Cloud Spanner
  + Streaming
    - Cloud Pub/Sub
    - Common Uses:
      * Telemetry data
        + IoT devices gather data from surrounding environment through sensors.
      * User events and analytics
  + Batch
    - Cloud Storage
    - Cloud Transfer Service
    - Cloud Transfer Appliance
    - Common uses:
      * Scientific workloads
      * Migrating to the cloud
      * Backing up data
      * Importing legacy data
* **Store**
  + Format and destination
* **Process and analyze**
  + Transformed from raw form into actionable information
* **Explore and visualize**
  + Convert the results of the analysis into a format that is easy to draw insights from and to share with colleagues and peers.
* Cloud Storage Offline Media Import/Export
  + 3rd party solution to load data into GCS
  + Send physical media (hard disk drives, tapes, flash drives) to 3rd party service who uploads data on your behalf.
  + Helpful if limited to a slow, unreliable, or expensive internet connection.
* If you want to use R for data exploration, you can deploy [RStudio Server](https://www.rstudio.com/products/rstudio/" \t "r) or [Microsoft Machine Learning Server](https://docs.microsoft.com/machine-learning-server/what-is-machine-learning-server) on a Compute Engine instance

**Storage**

****



**Google Cloud Storage (GCS)**

* Blob storage. Content not indexed.
* Virtually unlimited storage.
* Can have domain name buckets
* Can make requesters pay (ex. requester in different project)
* Pub/Sub can have notifications based on operations to buckets/objects
* Objects are immutable
* Can set Cache-Control metadata for frequently accessed objects
* Keep in mind compliance requirements when storing data in certain regions.
* No native directory support
  + Forward slashes have no special meaning
  + Performance of a native filesystem is not present.
* Storage classes can change, but the objects (files) within them retain their storage class.
* Not ideal for high volume read/write
* A way to store data that can be commonly used by Dataproc and Bigquery
* **IAM vs ACLs**
  + TODO
* **Signed URL to give temporary access and users do not need to be GCP users**
  + TODO
* **Storage Classes**
  + **Multi-regional**
    - Serving website content, interactive workloads, mobile game/gaming applications
    - Highest availability
    - Geo-redundant: Stores data in at least 2 regions separated by at least 100 miles within the multi-regional location of the bucket.
  + **Regional**
    - Storing data used by Compute Engine
    - Better performance for data-intensive computation
  + **Nearline**
    - Accessed once a month max
    - 30 day min. storage duration
    - Ex. Data backup, disaster recovery, archival storage
  + **Coldline**
    - Accessed once a year max
    - 90 day min. storage duration
    - Ex. Data stored for legal or regulatory reasons
* **Versioning**
  + Needs to be enabled
  + Things this enables:
    - List archived versions of an object
    - Restore live version of an object from an older state
    - Permanently delete an archived version
  + Archived versions retain ACLs and does not necessarily have same permissions as live version of object.
* **Encryption**
  + **Encryption at rest (Google-Managed Encryption Keys)**
    - Default (AES-256)
    - Use TLS or HTTPS to protect data as it travels over Internet
  + **Server-side encryption:**
    - Layers on top of default encryption
    - Occurs after GCS receives data, but before written to disk
      * **Customer-supplied encryption keys**
        + Provide key for each GCS operation
        + Key purged from servers after operation is complete
        + Stores only a cryptographic hash of key for future requests
        + Transfer Service, Dataflow, and Dataproc do not support this currently
        + Key rotation

Edit .boto config file

Encryption\_key = [NEW\_KEY]

Decryption\_key1 = [OLD\_KEY]

gsutil rewrite -k gs:://[BUCKET]/[OBJECT]

* + - * **Customer-managed encryption keys**
        + Generate and manage keys using Cloud Key Management Service (KMS)
        + KMS can be independent from the project that contains buckets (separation of duties)
        + Uses service accounts to encrypt/decrypt
        + Cloud SQL exports to GCS and Dataflow do not support this currently
  + **Client-side encryption:**
    - Occurs before data sent to GCS
    - GCS performs default encryption on it as well.
* **Storage Transfer Service**
  + Transfers data from an online data source (Amazon S3, HTTP/HTTPS location, GCS bucket) to a data sink (always GCS bucket).
  + Use cases:
    - Backup data to GCS from other storage providers
    - Move data from one GCS bucket to another (enables availability to different groups of users or applications)
    - Periodically move data as part of a processing pipeline or analytical workflow
  + Schedule one-time transfer operations or recurring ones
  + Delete existing objects in the destination bucket if they don’t have a corresponding object in source
  + Delete source objects after transferring them
  + Schedule periodic synchronization from data source to data sink with advanced filters based on file creation data, file-name filters, and the times of day you prefer to import data.
  + **Transfer Service vs. Gsutil**
    - On premise data source : gsutil
    - Another cloud storage provider data source : Transfer Service

**Cloud SQL**

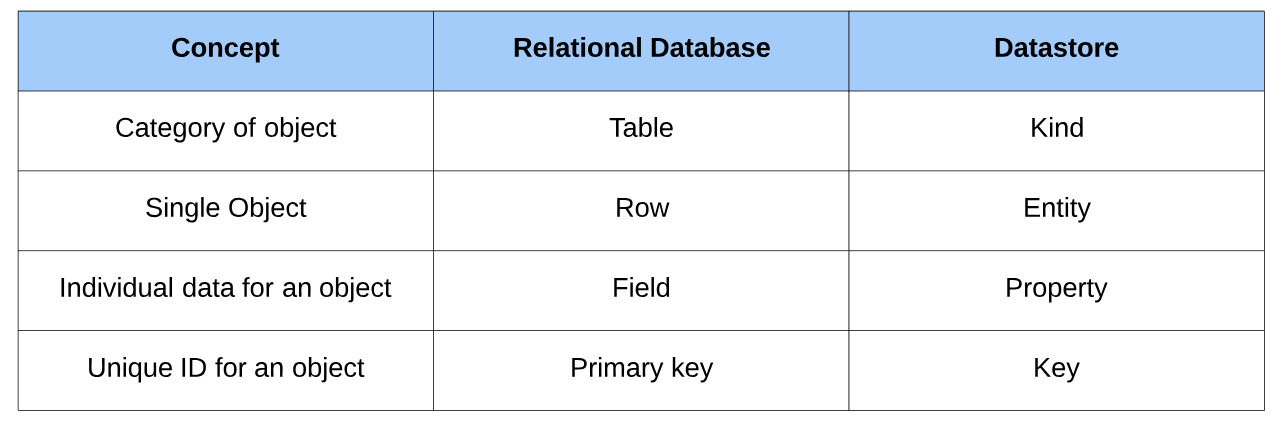
* Managed/No ops relational database (PostgreSQL, MySQL)
  + Complex queries perform better in postgresql
* Best for **gigabytes** of data with **transactional** nature
  + Low latency
  + Doesn’t scale well beyond GB’s
  + Data structures and underlying infrastructure required
* Too slow for analytics/BI/warehousing (OLAP)
* 2nd Generation Allow
  + Cloud Proxy Support
  + Higher availability configurations
  + Maintenance won’t take down the server
* Use SSD for production (instead of hard disk (persistent disk))
* Enable binary logging
  + For Point-in-time recovery and replication
* Bulk Loading Data
  + Copy data to GCS from SQL dump file or CSV files
    - SQL dump files cannot contain triggers, views, stored procedures
  + Import it into DB using copy from csv or something similar.
  + Use correct flags for dump file.
  + Compress data to reduce costs
    - Cloud SQL can import compressed .gz files
  + Use InnoDB for Second Generation instances
* Limited to 10 TB and is regional (not global)
  + Use Spanner if not good enough
* Use Case:
  + Medical Records
  + Blogs
* Read Replicas
  + In same region as master
  + Purpose to offload requests for analytics traffic from master.
* What to do when data size limits performance?
  + Many smaller tables perform better than one larger.

**Cloud Spanner**

* Distributed and scalable solution for RDBMS (more expensive)
* Horizontal scaling: Add more machines
* Use when:
  + Need high availability
  + Strong consistency
  + Transactional support for reads and writes (especially writes)
* Don’t use when:
  + Data is not relational, or not even structured
  + Want an open source RDBMS
  + Strong consistency/availability is overkill
* **IAM**
  + Project, instance, or database level
  + Roles/spanner.\_\_\_\_\_
    - Admin – Full access to Spanner resources
    - Database Admin – Create/edit/delete databases, grant access to databases
    - Database Reader – Read databases and execute schema
    - Viewer – View that instances and databases exist
      * Cannot modify or read from database.
* **Data Model**
  + Specifies a parent-child relationship for efficient storage
  + Interleaved representation (like **HBase**)
* **Parent Child Relationship**
  + Between tables
  + Cause physical location for fast access
    - i.e. query Students and Grades together, make Grades child of Student
  + Primary key of parent table **must** to be part of the key in the interleaved child table.
* **Interleaving**
  + Rows are stored in sorted order of primary key values
  + Child rows are inserted between parent rows with that key prefix
* **Hotspotting**
  + Need to choose primary keys carefully (like **HBase**)
  + Do not use monotonically increasing values, else writes will be on the same locations.
    - No timestamps (also sequential)
      * Use descending order if timestamps are required.
  + Use hash of key value if using naturally monotonically ordered keys (serial in postgres)
* **Splits**
  + Parent-child relationship can get complicated (i.e. 7 layers deep)
  + Spanner is distributed – uses “splits”
  + Split – Range of rows that can be moved around independent of other rows
  + Added to distribute high read-write data (to break up hotspots)
* **Secondary Indices**
  + Key-based storage ensures fast sequential scan of keys (like **HBase**)
  + Can also add secondary indices (**unlike HBase**)
    - Can cause data to be stored twice
      * i.e. Grades -> Course table | Grades -> Students table
  + Fine grained control on use of indices
    - Force query to use specific index: **Index Directives**
    - Force column to be copied into secondary index (use a STORING clause)
* Data Types
  + Non-normalized types such as ARRAY and STRUCT available too.
    - STRUCTs: NOT OK in tables, but can be returned in queries
    - ARRAYs: OK in tables, but ARRAYs of ARRAYs are not
* Transactions
  + Supports serializability
    - All transactions appear if they were executed in a serial order, even if some operations of distinct transactions actually occurred in parallel.
  + Stronger than traditional ACID
    - Transactions commit in an order that is reflected in their commit timestamps
    - Commit timestamps are “real time”
  + 2 Transaction Modes
    - Locking read-write
      * Slow
      * Only one that supports writing data
    - Read-only
      * Fast
      * Only requires read locking
  + If making a one-off read use “**Single Read Call**”
    - Fastest, no transaction checks needed!
* Staleness
  + Can set timestamp bounds
    - Strong: Read latest data
    - Bounded Staleness: Read version no later than …
      * Could be in past or future
* Production Environment
  + At least 3 nodes
  + Best performance when each CPU is under 75% utilization
* Multitenancy
  + Classic way is to create a separate database for each customer.
  + Recommended way for Spanner: Include a CustomerId key column in tables.
* Replicas
  + Paxos-based replication scheme in which voting replicas take a vote on every write request before it is committed.
  + Writes
    - Client write requests always go to leader replica first, even if a non-leader is closer geographically.
    - Leader logs incoming write, forwards it in parallel to other replicas that are eligible to vote.
    - Replicas complete its write and then responds back to leader with a vote on whether the write should be committed.
    - Write is committed when a quorum agrees.
  + Reads
    - Reads that are part of a read-write transaction are served from the leader replica, since the leader maintains the locks required to enforce serializability.
    - Single read and reads in a read-only transaction might require communication with leader, depending on concurrency mode.
  + Single-region instances can only use read-write replicas. (3 in prod)
  + Types
    - Read-write
      * Maintain a full copy of your data.
      * Can vote, can become leader, can serve reads
    - Read-only
      * Maintain a full copy of your data, which is replicated from read-write replicas.
      * Can serve reads
      * Do not participate in voting to commit writes -> location of read-only replicas never contribute to write latency.
      * Allow scaling of read capacity without increasing quorum size needed for writes (reduces total time of network latency for writes)
    - Witness
      * Can vote
      * Easier to achieve quorums for writes without the storage and compute resources required by read-write replicas to store a full copy of data and serve reads.
* **Architecture**
  + Nodes handle computation for queries, similar to that of BigTable.
    - Each node serves up to 2 TB of storage.
    - More nodes = more CPU/RAM = increased throughput.
  + Storage is replicated across zones (and regions, where applicable).
    - Like BigTable, storage is separate from computing nodes.
  + Whenever an update is made to a database in one zone/region, it is automatically replicated across zones/regions.
    - Automatic synchronous replications.
      * When data is written, you know it has been written.
      * Any reads guarantee data accuracy.

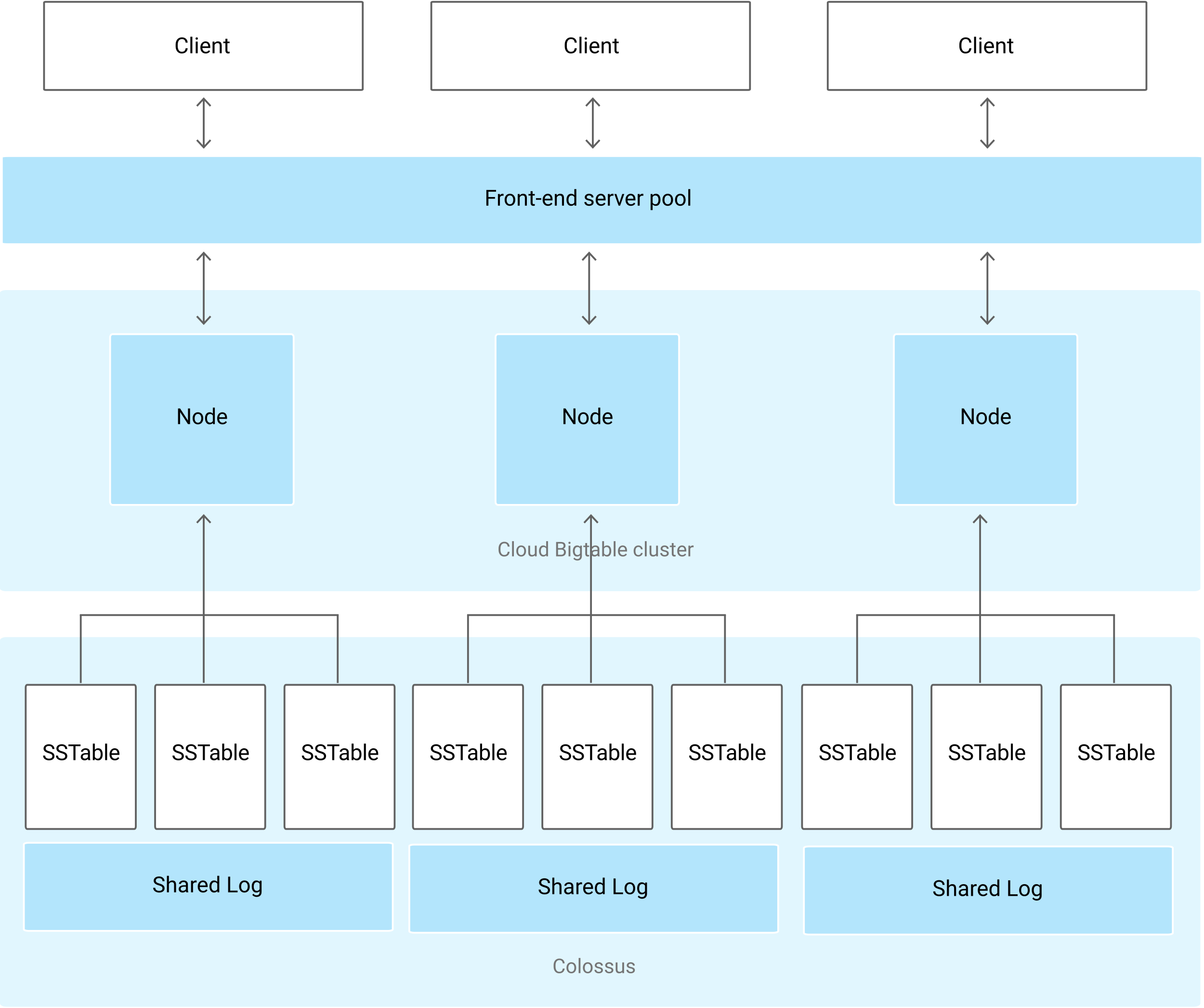
**DataStore**

* Typically, not used for either OLTP or OLAP
  + Fast lookup on keys is the most common use case.
* Specialty is that query execution depends on the size of the returned result and not the size of the data set.
  + Best for **lookup of non-sequential keys (needle in haystack)**
* Built on top of **BigTable**
  + Non-consistent for every row.
  + Document DB for **non-relational** data.
  + MongoDB equivalent (JSON-oriented NoSQL).
* Suitable for:
  + Atomic transactions
    - Can execute a set of operations where all succeed, or none occur.
  + ACID transactions, SQL-like queries.
  + Structured data.
  + Hierarchical document storage such as HTML and XML
* Query
  + Can search **by keys or properties** (if indexed)
  + Key lookups somewhat similar to Amazon DynamoDB
  + Allow for **SQL-like querying** down to property level.
  + Does not support:
    - Join operations
    - Inequality filtering on multiple properties.
      * Only 1 inequality filter per query is allowed.
    - Filtering on data based on a result of a subquery.
* Performance
  + Fast to **Terabyte** scale, low latency.
  + Quick read, **slow write** as it relies on indexing every property (default) and must **update indexes as updates/writes occur**.
* Comparison to RDBMS
  + All Datastore queries use indices.
  + Query time depends on size of result set alone in Datastore whereas RDBMS also depends on size of data set.
  + Entities (rows) of the same kind (table) can have different properties (fields).
  + Different entities can have properties with same name, but different value type.
* Properties can vary between entities.
  + Think optional tags in HTML.
* Avoid DataStore when:
  + If you need very strong transaction support.
  + Non-hierarchical or unstructured data (use BigTable instead)
  + Need extreme scale (10M+ read/writes per second) - BigTable
  + Analytics/BI/Data warehousing (BQ instead)
  + If application has a lot of writes and updates on key columns.
  + You need near zero latency (use in memory db Redis)
* Use DataStore when:
  + Scaling of read performance – to virtually any size.
  + Use for hierarchical documents with KV data.
    - Apps that need highly available structured data at scale.
    - Product catalogs, real time inventory
    - User profiles – mobile apps
    - Game save states
* Single Datastore database per project
* Where can you host?
  + Multi-regional for wide access
  + Single region for lower latency for single location
  + Cannot change after assignment… (have to delete project)
* **IAM Roles**
  + Primitive and predefined
  + Owner, user, viewer, import/export admin, index admin
* Entity Groups
  + Hierarchical relationship between entities.
  + Ancestor Paths and Child Entities.
* Index Types
  + Built in – default option
    - Allows single property queries
  + Composite – specified with index configuration file (index.yaml)
    - gcloud datastore create-indexes index.yaml
      * Creating/updating
* Deleting Index
  + datastore indexes cleanup
    - Deletes all indexes for the production Datastore mode instance that are not mentioned in the local version of index.yaml.
* Exploding Indexes
  + Default – create entry for every possible combination of property values
  + Results in higher storage and degraded performance
  + Solutions
    - Use custom index.yaml file to narrow index scope
    - Do not index properties that don’t need indexing
* Full Indexing
  + Built in indices on each property (~field) of each entity kind (~table row).
  + Composite indices on multiple property values.
  + Can exclude properties from indexing if certain it will never be queried.
  + Each query is evaluated using its “perfect index”
* Perfect Index
  + Given a query, which is the index that most optimally returns query results?
  + Depends on following (in order)
    - Equality filter
    - Inequality filter
    - Sort conditions if any specified.
* Implications of Full Indexing
  + Updates are really slow.
  + No joins possible.
  + Can’t filter results based on subquery results.
  + Can’t include more than one inequality filter.
* Multi-Tenancy
  + Separate data partitions for each client organizations.
  + Can use the same schema for all clients, but vary the values.
  + Specified via a namespace (inside which kinds and entities can exist)
* Transaction Support
  + Can optionally use transactions – not required
  + Stronger than BigQuery and BigTable
* Consistency
  + Strongly consistent
    - Return up to date result, however long it takes
    - Ancestor query
      * Those that execute against an entity group
      * Can set the read policy of a query to make this eventually consistent.
    - key-value operations
  + Eventually consistent
    - Faster, but might return stale data
    - Global queries/projections
* Deleting entities in bulk?
  + Use Dataflow
    - Datastore delete template that can be used to delete entities selected by a GQL query.
* Exporting Entities?
  + Deploy App Engine service that calls Datastore mode managed export feature.
  + Can run this service on a schedule with an App Engine Cron Service.
  + IAM
    - datastore.importExportAdmin
    - Storage Admin (for GCS)
* Serverless
* Cloud Firestore
  + Newest version of Datastore.
  + Native Mode
    - New strongly consistent storage layer.
    - New data model:
      * Kind => Collection Group
      * Entity => Document
      * Property => Field
      * Key => Document ID
    - Real-time updates
    - Mobile and Web client libraries
      * Scales to millions of concurrent clients.
  + Datastore Mode
    - Removes previous consistency limitations of Datastore.
    - Strongly consistent queries across the entire database.
    - Transactions can access any number of entity groups.
    - Scales to millions of writes per second.
* **Errors and Error Handling**
  + UNAVAILABLE, DEADLINE\_EXCEEDED
    - Retry using exponential backoff.
  + INTERNAL
    - Do not retry this request more than once.
  + Other
    - Do not retry without fixing the problem.



**BigTable**

* HBase equivalent
  + Work with it using HBase API
  + Advantages over HBase
    - Scalability (storage autoscales)
    - Low ops/admin burden
    - Cluster resizing without downtime
    - Many more column families before performance drops (~100K)
* Stored on Google’s internal store **Colossus**
* **Not transactional** (can handle **petabytes** of data)
* Fast scanning of sequential key values
* Column oriented NoSQL database
  + Good for sparse data
* Sensitive to hot spotting (like Spanner)
  + Data is sorted on key value and then sequential lexicographically similar values are stored next to each other.
  + Need to design key structure carefully.
* Designed for Sparse Tables
  + Traditional RDBMS issues with sparse data
    - Can’t ignore with petabytes of data.
    - Null cells still occupy space.
* Use BigTable When:
  + Very fast scanning and high throughput
    - Throughput has linear growth with node count if correctly balanced.
  + Non-structured key/value data
  + Each data item is < 10MB and total data > 1TB
  + Writes are infrequent/unimportant (no ACID) but fast scans crucial
  + Time Series data
* Avoid BigTable When:
  + Need transaction support
  + Less than 1TB data (can’t parallelize)
  + Analytics/BI/data warehousing
  + Documents or highly structured hierarchies
  + Immutable blobs > 10MB each
* **IAM**
  + Project wide or instance level
  + Read/Write/Manage
* 4-Dimensional Data Model
  + Row-Key
    - Uniquely identifies a row
    - Can be primitives, structures, arrays
    - Represents internally as a byte array
    - Sorted in ascending order
    - \*\*NOTE\*\* - Can only query against this key.
  + Column Family
    - Table name in RDBMS
    - All rows have the same set of column families
    - Each column family is stored in a separate data file
    - Set up at schema definition time
      * Columns can be added on the fly
    - Can have different columns for each row
  + Column
    - Columns are units within a column family.
  + Timestamp
    - Support for different versions based on timestamps of same data item. (like Spanner)
    - Omit timestamp gets you the latest data.
* Hotspotting
  + Overloading a node with requests.
  + Row keys to Use
    - Field Promotion
      * Move fields from column data that you need to search against should be included in a single row key.
      * Use in reverse URL order like Java package names
        + Keys have similar prefixes, but different endings
    - Salting
      * Hash the key value
    - Timestamps as suffix in key (reverse timestamp)
  + Row Keys to Avoid
    - Domain names (as opposed to field promotion)
      * Will cause common portion to be at end of row key leading to adjacent values to not be logically related.
    - Sequential numeric values.
    - Timestamps alone
    - Timestamps as prefix of row-key.
    - Mutable or repeatedly updated values.
* Schema Design
  + Each table has just 1 index – row key
  + Rows sorted lexicographically by row key
  + All operations are atomic at row level
  + Keep all entity info in a single row.
  + Related entities should be in adjacent rows
    - More efficient reads.
  + Tables are sparse: Empty columns don’t take up any space.
    - Create a very large number of columns even if most are empty in most rows.
* “Warming the Cache”
  + BigTable will improve performance over time.
  + Will observe read and write patterns and redistribute data so that shards are evenly hit.
  + Will try to store roughly same amount of data in different nodes.
  + Testing over hours is important to get true sense of performance.
* SSD or HDD Disks
  + Use SSD unless skimping costs.
    - Can be 20x faster on individual row reads.
      * Less important with batch reads or sequential scans.
    - More predictable throughput too (no disk seek variance)
  + When to use HDD?
    - > 10 TB storage
    - All batch queries
  + The more random access, the stronger case for SSD
    - Purely random -> maybe use DataStore
  + Impossible to switch between SSD and HDD
    - Export data from the existing instance and import data into a new instance.
    - OR write a cloud Dataflow or Hadoop MapReduce job that copies the data from one instance to another.
* Poor Performance Explained
  + Poor schema design
  + Inappropriate workloads
    - Too small (< 300 GB)
    - Used in short bursts
  + Cluster to small
  + Cluster just fired up or scaled up
  + HDD instead of SSD
  + Dev. Vs. Prod instance
* Data Update
  + Deleting/updating actually write a new row with the desired data.
  + Append only, cannot update a single field
  + Tables should be **tall and narrow**
    - Tall – Store changes by appending new rows
    - Narrow – Collapse flags into a single column
* Production & Development
  + Prod:
    - Standard instance with 1-2 clusters
    - 3 or more nodes in each cluster
      * Use replication to provide high availability
    - Replication available, throughput guarantee
  + Development:
    - Low cost instance with 1 node cluster
    - No replication
  + Create Compute Engine instance in same zone as Big Table instance
* Resizing
  + Add and remove nodes and clusters with no downtime.
* Tools for interacting with BigTable
  + **cbt (Command Line tool)**
    - Tool for doing basic interactions with BigTable
    - Use this if possible as it is simpler than HBase shell.
  + HBase Shell
    - Command-line tool performs admin tasks such as creating and deleting tables.
    - Can update the following without any downtime:
      * Number of clusters/replication settings
      * Upgrade a development instance to production (permanent)
* Architecture
  + Entire BigTable project is called an instance.
  + BigTable instance comprise of Clusters and Nodes
  + Tables belong to instances
    - If multiple clusters, you cannot assign a table to an individual cluster
* Structure (1 Cluster below)

****

* + Data is never stored in BigTable nodes; each node has pointers to a set of tables stored on Colossus (GCS is built on this as well).
    - Rebalancing tablets from one node to another is very fast because the data is not actually copied. Pointers are simply updated.
    - Recovery from the failure of a node is very fast, only metadata needs to be migrated to the replacement node.
    - When BigTable fails, no data is lost.
  + Single table is sharded across multiple tablets.
* Compared to DataStore
  + BigTable queries are on the Key rather than an Index
  + BigTable supports atomicity only on a single row – no transactions

**BigQuery**

* Hive equivalent
* No ACID properties
* Great for analytics/business intelligence/data warehouse (OLAP)
* Fully managed data warehouse
* Has connectors to BigTable, GCS, Google Drive, and can import from Datastore backups, CSV, JSON, and AVRO.
* **Performance**
  + **Petabyte** scale
  + **High latency**
    - Worse than BigTable and DataStore
* **Architecture**
  + Jobs (queries) can scale up to thousands of CPU’s across many nodes, but the process is completely invisible to end user.
  + Storage and compute are separated, connected by petabit network.
  + Columnar data store
    - Separates records into column values, stores each value on different storage volume.
    - Poor writes (BQ does not update existing records)
* **IAM**
  + Security can be applied at **project and dataset level**, but not at table or view level.
  + Predefined roles BQ
    - Admin – Full access
    - Data owner – Full dataset access
    - Data editor – edit dataset tables
    - Data viewer – view datasets and tables
    - Job User – run jobs
    - User – run queries and create datasets (but not tables)
  + **Authorized views** allow you to share query results with particular users/groups without giving them access to underlying data.
    - Restrict access to **particular columns or rows**
    - Create a **separate dataset** to store the view.
    - How:
      * Grant IAM role for data analysts (bigquery.user)
        + They won’t have access to query data, view table data, or view table schema details for datasets they did not create.
      * (In source dataset) Share the dataset, In permissions go to Authorized views tab.
        + View gets access to source data, not analyst group.
* **Pricing**
  + Based on:
    - **storage** (amount of data stored)
    - **querying** (amount of data/number of bytes processed by query)
    - **streaming** inserts.
  + **Storage options** are active and long term
    - Modified or not past 90 days
  + **Query options** are on-demand and flat-rate
* **Data Model**
  + Dataset = set of tables and views
  + Table must belong to dataset
  + Dataset must belong to a project
  + Tables contain records with rows and columns (fields)
    - Nested and repeatable fields are OK
* **Table Schema**
  + Can be specified at creation time
  + Can also specify schema during initial load
  + Can update schema later too
* **Query**
  + Standard SQL (preferred) or Legacy SQL (old)
    - Standard
      * Table names can be referenced with backticks
        + Needed for wildcards
  + Cannot use both Legacy and SQL2011 in same query.
  + Table partitioning
  + Distributed writing to file for output (i.e. file-0001-of-0002)
  + User defined functions in JS (**UDFJS**)
  + Query jobs are actions executed asynchronously to load, export, query, or copy data.
  + If you use the **LIMIT** clause, BigQuery will still process the **entire table**.
  + **Avoid SELECT \*** (full scan), select only columns needed (SELECT \* EXCEPT)
  + **Denormalized Data Benefits**
    - Increases query speed
    - Makes queries simpler
    - BUT: Normalization makes dataset better organized, but less performance optimized.
  + **Types**
    - **Interactive (default)**
      * Query executed immediately
      * Counts towards
        + Daily usage
        + Concurrent usage
    - **Batch**
      * Scheduled to run whenever possible (idle resources)
      * Don’t count towards limit on concurrent usage.
      * If not started within 24hr, BQ makes them interactive.
* **Data Import**
  + Data is converted into columnar format for Capacitor.
  + **Batch** (free)
    - web console (local files), GCS, GDS, Datastore backups (particularly logs)
    - Other Google services (i.e. Google Ad Manager, Google Ads)
  + **Stream** (costly)
    - Data with CDF, Cloud Logging, or POST calls
    - High volume event tracking logs
    - Realtime dashboards
    - When streaming to a partitioned table:
      * NULL value for \_PARTITIONTIME pseudo column
      * Check tables.get response for a section named streamingBuffer
        + If absent, data is available to copy/export and should have a non null value now for the pseudo column.
  + **Raw Files**
    - Federated data source, CSV/JSON/Avro on GCS, Google sheets
  + **Google Drive**
    - Loading is not currently supported.
    - Can query data in Drive using an external table.
  + Expects all source data to be **UTF-8** encoded.
  + To support (occasionally) **schema changing** you can use **automatically detect** (not default setting).
    - Available while:
      * Loading data
      * Querying external data
  + **Web UI**
    - Upload a file greater than 10MB in size
    - Upload multiple files at the same time
    - Upload a file in SQL format
    - Cannot load multiple files at once.
      * Can with CLI though.
* **Loading Compressed and Uncompressed Data**
  + Avro preferred for loading compressed data.
    - Faster to load since it can be read in parallel, even when data blocks are compressed.
  + Parquet Binary format also a good choice
    - Efficient per-column encoding typically results in better compression ratio and smaller files.
  + ORC Binary format offers benefits similar to Parquet
    - Fast to load because data stripes can be read in parallel.
    - Rows in each stripe are loaded sequentially.
    - To optimize load time: data stripe size of 256MB or less.
  + CSV and JSON
    - BQ load uncompressed files significantly faster than compressed.
    - Uncompressed can be read in parallel.
    - Uncompressed are larger => bandwidth limitations and higher GCS costs for data staged prior to being loaded into BQ.
    - Line ordering not guaranteed for compressed or uncompressed.
  + If bandwidth limited, compress with GCIP before uploading to GCS.
  + If speed is important and you have a lot of bandwidth, leave uncompressed.
* **Loading Denormalized, Nested, and Repeated Data**
  + BQ performs best with denormalized data.
  + Increases in storage costs worth the performance gains of denormalized data.
  + Joins require data coordination (communication bandwidth)
    - Denormalization localizes the data to individual slots so execution can be done in parallel.
  + If need to maintain data while denormalizing data
    - Use nested and repeated fields instead of completely flattening data.
    - When completely flattened, network communication (shuffling) can negatively impact query performance.
  + Avoid denormalization when:
    - Have a star schema with frequently changing dimensions.
    - BQ complements and OLTP system with row-level mutation, but can’t replace it.
* **BigQuery Transfer Service**
  + Automates loading data into BQ from Google Services:
    - Campaign Manager
    - Cloud Storage
    - Amazon S3
    - Google Ad Manager
    - Google Ads
    - Google Play
    - YouTube – Channel Reports
    - YouTube – Content Owner Reports
* **Partitions**
  + Improves query performance => reduces costs
  + **Cannot change an existing table into a partitioned table.**
  + **Types**
    - **Ingestion Time**
      * Partition based on data’s ingestion date or arrived date.
      * Pseudo column `\_PARTITIONTIME`
        + Reserved by BQ and can’t be used.
      * Need to update schema of table before loading data if loading into a partitions with a different schema.
    - **Partitioned Tables**
      * Tables that are partitioned based on a `TIMESTAMP` or `DATE` column.
      * 2 special partitions are created
        + \_\_NULL\_\_ paritition

Represents rows with NULL values in the partitioning column

* + - * + \_\_UNPARTITIONED\_\_ partition

Represents data that exists outside the allowed range of dates

* + - * All data in partitioning column matches the date of the partition identifier with the exception of those 2 special partitions.
        + Allows query to determine which partitions contain no data that satisfies the filter conditions.
        + Queries that filter data on the partitioning column can restrict values and completely prune unnecessary partitions.
  + **Wildcard tables**
    - Used if you want to union all similar tables with similar names. (i.e. project.dataset.Table\*)
    - Filter in WHERE clause
      * AND \_TABLE\_SUFFIX BETWEEN ‘table003’ and ‘table050’
* **Windowing**
  + Window functions increase the efficiency and reduce the complexity of queries that analyze partitions (windows) of a dataset by providing complex operations without the need for many intermediate calculations.
  + Reduce the need for intermediate tables to store temporary data.
* **Bucketing**
  + Like partitioning, but each split/partition should be the same size and is based on the hash function of a column.
  + Each bucket is a separate file, which makes for more efficient sampling and joining data.
* **Legacy vs. Standard SQL**
  + Standard: ‘project.dataset.tablename\*’
  + Legacy: [project.dataset.tablename]
  + It is set **each time you run a query**
  + Default query language is
    - Legacy SQL for classic UI
    - Standard SQL for Beta UI
* **Anti-Patterns**
  + Avoid self joins
  + Partition/Skew
    - Avoid unequally sized partitions
    - Values occurring more often than other values..
  + Cross-Join
    - Joins that generate more outputs than inputs
  + Update/Insert Single Row/Column
    - Avoid a specific DML, instead batch updates/inserts
  + Anti-Patterns: <https://cloud.google.com/bigtable/docs/schema-design>
* **Table Types**
  + **Native Tables**
    - Backed by native BQ storage
  + **External Tables**
    - Backed by storage external to BQ (**federated data source**)
    - BigTable, Cloud Storage, Google Drive
  + **Views**
    - Virtual tables defined by SQL query.
    - Logical – not materialized
    - Underlying query will execute each time the view is accessed.
    - Benefits:
      * Reduce query complexity
      * Restrict access to data
      * Construct different logical tables from same physical table
    - Cons:
      * Can’t export data from a view
      * Can’t use JSON API to retrieve data
      * Can’t mix standard and legacy SQL
        + E.g. standard sql cannot access legacy sql view
      * No user-defined functions allowed
      * No wildcard table references
        + Due to partitioning
      * Limit of 1000 authorized views per dataset
* **Caching**
  + No charge for a query that retrieves results from cache.
  + Results are cached for 24 hours.
  + Caching is per user only.
  + bq query –nouse\_cache ‘<QUERY>’
  + Cached by Default unless
    - A destination table is specified.
    - If any referenced tables or logical units have changed since results previously cached.
    - If any referenced tables have recently received streaming inserts even if no new rows have arrived.
    - If the query uses non-deterministic functions such as CURRENT\_TIMESTAMP(), NOW(), CURRENT\_USER()
    - Querying multiple tables using a **wildcard**
    - If the query runs against an external data source.
* **Export**
  + Destination has to be GCS.
    - Can copy table to another BigQuery dataset though.
  + Can be exported as JSON/CSV/Avro
    - Default is CSV
  + Only compression option: GZIP
    - Not supported for Avro
  + To export > 1 GB
    - Need to put a wildcard in destination filename
    - Up to 1 GB of table data in a single file
  + bq extract ‘project:dataset.table’ gs://bucket
* **Query Plan Explanation**
  + In web UI, click on “Explanation”
  + Good for debugging complex queries not running as fast as needed/expected.
  + Monitoring Query Performance (UI)
    - `Details` button after running query.
    - Colors
      * Yellow – Wait
      * Purple – Read
      * Orange – Compute
      * Blue – Write
    - Less parallel inputs => better performance => best cost
* **Slots**
  + Unit of computational capacity needed to run queries.
  + BQ calculates on basis of query size, complexity
  + Usually default slots are sufficient
  + Might need to be expanded over time, complex queries
  + Subject to quota policies ($$)
  + Can use StackDriver Monitoring to track slot usage.
* **Clustered Tables**
  + Order of columns determines sort order of data.
  + Think of Clustering Columns in Cassandra
  + When to use:
    - Data is already **partitioned** on date or timestamp column.
    - You commonly use **filters** or **aggregation** against **particular columns** in your queries.
  + Does not work if the clustered column is used in a complex filter (used in a function in the filter expression)
* **BigQuery ML**
  + Create and execute machine learning models in BQ using standard SQL
  + Supported models
    - Linear regression
    - Binary Logistic regression
    - Multiclass logistic regression for classification
  + Benefits from not having to export and re-format data
* **Best Practices**
  + **Costs**
    - Avoid SELECT \*
      * Query only columns you need.
    - Sample data using preview options
      * Don’t run queries to explore or preview table data.
    - Price your queries before running them.
      * Before running queries, preview them to estimate costs.
    - Limit query costs by restricting the number of bytes billed.
      * Use the maximum bytes billed setting to limit query costs.
    - LIMIT doesn’t affect cost
      * Do not use LIMIT clause as a method of cost control as it does not affect the amount of data that is read.
    - View costs using a dashboard and query your audit logs
      * Create a dashboard to view your billing data so you can make adjustments to your BigQuery usage. Also consider streaming audit logs to BigQuery to analyze usage patterns.
    - Partition data by date
    - Materialize query results in stages
      * Break large query into stages where each stage materializes the results by writing to a destination table.
      * Querying smaller destination table reduces amount of data that is read and lowers costs.
    - Consider cost of large result sets
      * Use default table expiration time to remove data when not needed.
      * Good for when writing large query results to a destination table.
    - Use streaming inserts with caution
      * Only use if data is needed immediately available.
  + **Query Performance**
    - Input data and data sources (I/O)
      * Control projection – Avoid SELECT \*
      * Prune partitioned queries
        + Use partition columns to filter
      * Denormalize data when possible
        + JSON, Parquet, or Avro
        + When creating, specify Type in the Schema as RECORD
      * Use external data sources appropriately
        + If performance is a top priority, do not use external source
      * Avoid excessive wildcard tables
        + Use most granular prefix possible
    - Communication between nodes (shuffling)
      * Reduce data before using a JOIN
      * Do not treat WITH clauses as prepared statements
      * Avoid tables sharded by date
        + Use time-based partitioned tables instead

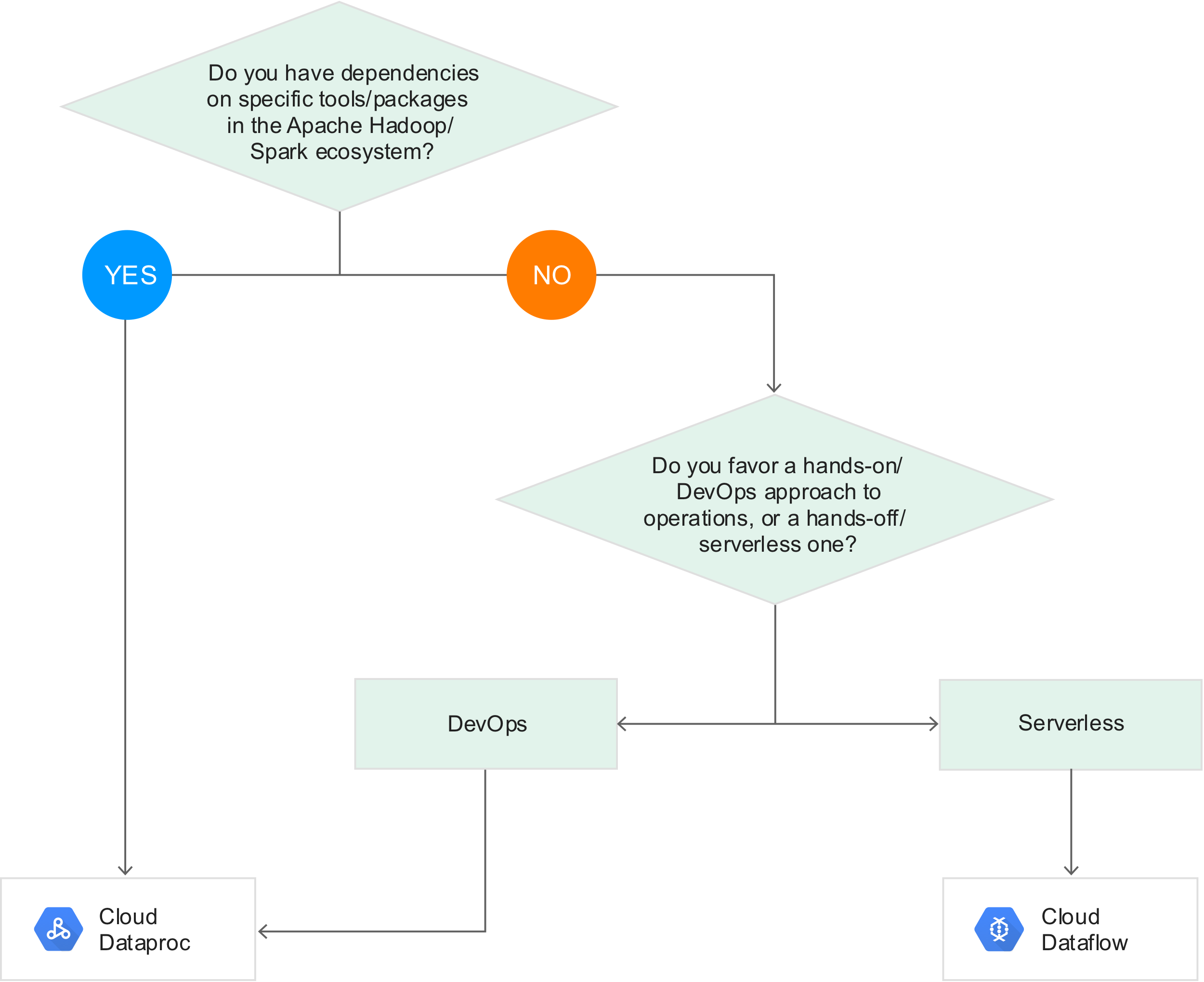
Copy of schema and metadata is maintained for each sharded table.

BQ might have to verify permissions for each queries table. (overhead)

* + - * Avoid oversharding tables
    - Computation
      * Avoid repeatedly transforming data via SQL queries
      * Avoid JavaScript user-defined functions.
        + Use native UDFs instead.
      * Use approximate aggregation functions
        + COUNT(DISTINCT) vs. APPROX\_COUNT\_DISTINCT()
      * Order query operations to maximize performance
        + Only use in the outermost query or within window clauses.
        + Push complex operations to the end of the query.
      * Optimize join patterns
        + Start with the largest table
      * Prune partitioned queries
    - Outputs (materialization)
      * Avoid repeated joins and subqueries
      * Carefully consider materializing large result sets
      * Use LIMIT clause with large sorts
    - Anti-patterns
      * Self-joins
        + Potentially doubles number of output rows
        + Use window function instead
      * Data skew
        + If query processes keys that are heavily skewed to a few values, filter your data as early as possible.
      * Cross joins (Cartesian product)
        + Avoid joins that generate more outputs than inputs.
        + Pre-aggregate data first if it is required.
      * DML statements that update or insert single rows
        + Use batch.
  + **Storage Optimization**
    - Use expiration settings to remove unneeded tables and partitions
      * Configure default table expiration for datasets
      * Configure expiration time for tables
      * Configure partition expiration for partitioned tables
    - Take advantage of long term storage
      * Untouched tables (90 days) are as cheap as GCS Nearline
      * Each partition is considered separately.
    - Use pricing calculator to estimate storage costs

**Dataflow**

* Executes **Apache Beam Pipelines**
* Can be used for **batch or stream** data
* **Scalable, fault-tolerant**, multi-step processing of data.
* Often used for data preparation/ETL for data sets.
* Integrates with other tools (GCP and external)
  + Natively – PubSub, BigQuery, Cloud AI Platform
  + Connectors – BigTable, Apache Kafka
* Pipelines are regional-based
* Follows the **Flink Programming Model**
  + Data Source -> Transformations -> Data Sink
* Use when:
  + No dependencies on Apache Hadoop/Spark
  + Favor hands-off/serverless
  + Preprocessing for machine learning with Cloud ML Engine
* **Apache Beam Architecture**
  + **Pipeline**
    - Entire set of computations
    - Not linear, it is a **DAG**
    - Beam programs start by constructing a Pipeline object.
  + A single, potentially repeatable job, from start to finish, in Dataflow.
  + Defined by driver program.
    - Actual computations run on a **backend**, abstracted in the **driver** by a **runner**.
      * Driver: Defines DAG
      * Runner: Executes DAG
  + Supports multiple backends
    - Spark
    - Flink
    - Dataflow
    - Beam Model
  + **Element**
    - A single entry of data (e.g. table row)
  + **PCollection**
    - Distributed data set in pipeline (immutable)
    - Specialized container classes that can represent data sets of virtually unlimited size.
      * Fixed size: Text file or BQ table
      * Unbounded: Pub/Sub subscription
    - **Side inputs**
      * Inject additional data into some PCollection
      * Can inject in **ParDo** transforms.
  + **PTransform**
    - Data processing operation (step) in pipeline
      * Input: 1 or more PCollection
      * Processing function on elements of PCcollection
      * Output: 1 or more PCollection
  + **ParDo**
    - Core of parallel processing in Beam SDKs
    - Collects the zero or more output elements into an output PCollection.
    - Useful for a variety of common data processing operations, including:
      * Filtering a data set.
        + Better than a Filter transform in the sense that a filter transform can only filter based on input element => no side input allowed.
      * Formatting or type-converting each element in a data set.
      * Extracting parts of each element in a data set.
      * Performing computations on each element in a data set.
* **Dealing with late/out of order data**
  + Latency is to be expected (network latency, processing time, etc.)
  + PubSub does not care about late data, that is resolved in Dataflow.
  + Resolved with Windows, Watermarks, and Triggers.
  + **Windows** = Logically divides element groups by time span.
  + **Watermarks** = Timestamp
    - Event time – When data was generated
    - Processing time – when data processed anywhere in pipeline
    - Can use Pub/Sub provided watermark or source generated.
  + **Trigger** = Determine when results in window are emitted.
    - (Submitted as complete)
    - Allow late-arriving data in allowed time window to re-aggregate previously submitted results.
    - Timestamps, element count, combinations of both.
* Templates
  + Google provided templates.
    - WordCount
    - Bulk compress/decompress GCS Files
    - PubSub to (Avro, PubSub, GCS Text, BigQuery)
    - Datastore to GCS Text
    - GCS Text to (BigQuery, PubSub, DataStore)
* Requires a **Staging Location** where intermediate files may be stored.
* **IAM**
  + Project-level only – all pipelines in the project (or none)
  + Pipeline data access separate from pipeline access.
  + Dataflow Admin
    - Full pipeline access
    - Machine type/storage bucket config access
  + **Dataflow.developer**
    - Full pipeline access
    - No machine type/storage bucket access (data privacy)
  + Dataflow Viewer
    - View permissions only.
  + **Dataflow.worker**
    - Enables **service account** to execute work units for a Dataflow pipeline in Compute Engine.
    - **Dataflow API** also needs to be enabled.
* **Stopping a Dataflow Jobs**
  + **Cancelling**
    - Immediately stop and abort all data ingestion and processing.
    - Buffered data may be lost.
  + **Draining**
    - Cease ingestion but will attempt to finish processing any remaining buffered data.
    - Pipeline resources will be maintained until buffered data has finished processing and any pending output has finished writing.
* **Pipeline Update**
  + Replace an existing pipeline in-place with a new one and preserve Dataflow’s exactly-once processing guarantee.
  + When updating pipeline manually, use **DRAIN** instead of CANCEL to maintain in flight data.
    - Drain command is supported for **streaming pipelines only**
  + Pipelines cannot share data or transforms.
* **Windowing**
  + Can apply **windowning** to streams for rolling average for the window, max in a window etc.
  + Types
    - **Fixed Time Windows (Tumbling Window)**
      * Fixed window size
      * Non-overlapping time
      * Number of entities differ within a window
    - **Sliding Time Windows (overlapped)**
      * Fixed window size
      * Overlapping time
      * Number of entities differ within a window
      * Window Interval: How large window is
      * Sliding Interval: How much window moves over
    - **Session Windows**
      * Changing window size based on session data
      * No overlapping time
      * Number of entities differ within a window
      * Session gap determines window size
      * Per-key basis
      * Useful for data that is irregularly distributed with respect to time.
    - **Single Global Window**
      * Late data is discarded
      * Okay for bounded size data
      * Can be used with unbounded but use with caution when applying transforms such as GroupByKey and Combine
  + **Default** windowing behavior is to assign all elements of a PCollection to a **single, global window** even for unbounded PCollections.
* **Triggers**
  + Determines when a Window’s contents should be output based on a certain being met.
    - Allows specifying a trigger to control when (in processing time) results for the given window can be produced.
    - If unspecified, the default behavior is to trigger first when the watermark passes the end of the window, and then trigger again every time there is late arriving data.
  + **Time-Based Trigger**
    - **Event Time Triggers**
      * Operate on event time, as indicated by timestamp on each data elements.
      * This is the **default trigger**.
    - **Processing Time Triggers**
      * Operate on the processing time – the time when the data element is processed at any given stage in the pipeline.
  + **Data-Driven Trigger**
    - Operate by examining the data as it arrives in each window, and firing when that data meets a certain property.
    - Currently, only support firing after a certain **number of data elements**.
  + **Composite Triggers**
    - Combine multiple triggers in various ways.
* **Watermarks**
  + System’s notion of when all data in a certain window can be expected to have arrived in the pipeline.
  + Tracks watermark because data is not guaranteed to arrive in a pipeline in order or at predictable intervals.
  + No guarantees about ordering.
  + Indicates all windows ending before or at this timestamp are closed.
  + No longer accept any streaming entities that are before this timestamp.
  + For unbounded data, results are emitted when the watermark passes the end of the window, indicating that the system believes all input data for that window has been processed.
  + Used with **Processing Time**
* **Tech**
  + **DirectPipelineRunner**
    - Allows you to execute operations in the pipeline directly and locally.
  + Create a cron job with App Engine Cron Service to run Dataflow job.
* System Lag
  + Max time an element has been waiting for processing in this stage of the pipeline.
* Wall Time
  + How long the processing takes.
* **Key Things** 
  + Constraints you may have.
  + Why you would use JSON or Java related to Pipelines
* **How to improve performance?**
  + TODO

****

**Dataproc**

* Managed Hadoop (Spark, SparkML, Hive, Pig, etc…)
* Automated cluster management, resizing
* Code/Query only
* Job management screen in the console
* Think in terms of a ‘job specific resource’ – for each job, create a cluster and then delete it.
* Used if **migrating existing on-premise Hadoop or Spark infrastructure** to GCP without redevelopment effort.
* Can sale even when jobs are running.
* Use Dataflow for streaming instead. **This is better for batch**.
* Connecting to Web Interface of Dataproc Cluster
  + Allow necessary web ports access via firewall rules, and limit access to your network.
    - Tcp:8088 (Cluster Manager)
      * <Master Node IP>:8088
    - Tcp:50070 (Connect to HDFS name node)
      * <Master Node IP>:50070
  + OR SOCKS proxy (routes through an SSH tunnel for secure access)
    - gcloud compute ssh [master\_node\_name]
* **Pricing**
  + Standard Compute Engine machine type pricing + managed Dataproc premium.
  + Premium = $0.01 per vCPU core/hour
  + Billed by the second, with a minimum of 1 minute.
* **IAM**
  + Project level only (primitive and predefined roles)
  + Cloud Dataproc Editor, Viewer, and Worker
    - Editor – Full access to create/edit/delete clusters/jobs/workflows
    - Viewer – View access only
    - Worker – Assigned to service accounts
      * Read/write GCS, write to Cloud Logging
* **Storage**
  + Can use on disk (HDFS) or GCS
  + HDFS
    - Split up on the cluster, but requires cluster to be up.
  + GCS
    - Allows for the use of preemptible machines that can reduce costs significantly.
      * DO not need to configure startup and shutdown scripts to gracefully handle shutdown, Dataproc already handles this.
      * Cluster MUST have at least 2 standard worker nodes however.
    - Separate cluster and storage.
* **Cluster Machine Types**
  + Build using Compute Engine VM instances
  + Cluster – need at least 1 master and 2 workers
* **High Availability Mode**
  + 3 masters rather than 1
  + 3 masters run in an Apache Zookeeper cluster for automatic failover.
* **Restartable Jobs**
  + Jobs do NOT restart on failure (default)
  + Can change this – useful for long running and streaming jobs (ex. Spark Streaming)
  + Mitigates out-of-memory errors, unscheduled reboots
* **Updating Clusters**
  + Can only change # workers/preemptible VM’s/labels/toggle graceful decommission.
  + Automatically reshards data for you.
* **Migrating and Optimizing for GCP Best Practices**
  + Move data first
    - Generally to GCS buckets.
    - Possible exceptions
      * Apache HBase data to BigTable
      * Apache Impala to BigQuery
      * Can still choose to move to GCS if BigTable/BQ feature not needed.
  + Small scale experimentation
    - Use a subset of data to test.
  + Think of it in terms of ephemeral clusters.
  + Use GCP tools to optimize and save costs.
* **Converting from HDFS to GCS**
  + Copy data to GCS
    - Install connector or copy manually
  + Update file prefix in scripts
    - Hdfs:// to gs://
  + Use Dataproc, and run against/output to GCS
* **Connectors**
  + BQ/BigTable (copies data to GCS) /CloudStorage
* **Optional Components**
  + Anaconda, Druid, Hive WebHCat, Jupyter, Kerberos, Presto, Zeppelin, Zookeeper
* **How to configure Hadoop to use all cores?**
  + Think spark executor cores
* **How to handle out of memory errors?**
  + Hint - Executor memory
* **How to install other components?**
  + Hint – Initialization actions

**Datalab**

* Managed Jupyter notebooks
* Great for use with a Dataproc cluster to write PySpark jobs
* Supports Python, SQL (BQ), and JavaScript (for BQ user-defined functions)
  + In Cell
    - %%bq query –name queryname
      * SQL underneath
    - %%bq execute -q queryname
* Runs on GCE instance, dedicated VPC and Cloud Source Repository
  + datalab create <instance name>
  + datalab-network (VPC) is created
  + datalab connect <instance name>
  + Cloud Source Repository
    - Used for sharing notebook between users
* 3 Ways to Run:
  + Locally
    - Good if only one person using
  + Docker on GCE
    - Better
    - Use by multiple people through SSH or CloudShell
    - Uses resources on GCE
  + Docker + Gateway
    - Best
    - Uses a gateway and proxy
    - Runs locally
* Powerful interactive tool to explore, analyze, transform and visualize data and build machine learning models on GCP.
* Notebooks
  + Can be in **Cloud Storage Repository** (git repo)
    - Use ungit to commit changes to notebooks
* Persistent Disk
  + Notebooks can be cloned from GCS to VM persistent disk.
  + This clone => workspace => add/remove/modify files
  + Notebooks autosave, but you need to commit.
* Kernel
  + Opening a notebook => Backend kernel process manages session and variables.
  + Each notebook has 1 python kernel
  + Kernels are single-threaded
  + Memory usage is heavy – execution is slow – pick machine type accordingly
* APIs and Services
  + Enable Compute Engine API
* Sharing Notebook Data:
  + GCE access based on GCE IAM roles:
    - Must have Compute Instance Admin and Service Account Actor roles
      * Service Account Actor role deprecated. Use Service Account Token Creator instead.
  + Notebook access per user only
  + Sharing data performed via shared Cloud Source Repository
  + Sharing is at the project level.
* Creating Team Notebooks
  + 2 Options
    - Team lead creates notebooks for users using –for user option:
      * datalab create [instance] –for-user [bob@blah.net](mailto:bob@blah.net)
    - Each user creates their own datalab instance/notebook
    - Everyone accesses same shared repository of datalab/notebooks
  + NO web console option
  + Machine Type
    - Standard n1 by default
    - Multi-threading does not work, but can use high memory
    - Custom machine types supported as well.
  + Can disable creating shared cloud repository
    - --no-create-repository
* Connecting
  + SSH tunnels to notebook on port 8081
  + datalab connect <instance name>
    - RSA key is passphrase
  + Can configure idle timeouts on the actual webpage
* Cost
  + Free
  + Only pay for GCE resources Datalab runs on and other GCP services you interact with.

**Pub/Sub**

* Server-less messaging “middleware”
* Many to many asynchronous messaging
* Decouples sender and receiver
* Attributes can be set by sender (KV pairs)
* Glue that connects all components
* Order not guaranteed
* Encoding as a Bytestring (utf-8) required for publishing.
* Publishers: Any app that can make HTTPS requests to googleapis.com
* **IAM**
  + Control access at project, topic, or subscription level
  + Admin, Editor, Publisher, Subscriber
  + Service accounts are best practice.
* **Pricing**
  + Data volume used per month (per GB)
* **Message Flow**
  + Publisher app creates a topic object and sends a message to the topic.
  + Messages persisted in message store until acknowledged by subscribers
  + Messages forwarded from topic to all subscriptions individually.
  + Subscriber receives pending messages from its subscription and acknowledges each one to the Cloud Pub/Sub service.
    - Push
      * WebHook endpoint (must accept POST HTTPS request)
      * Lower latency, more real time.
    - Pull
      * Subscriber explicitly calls pull method which requests messages for delivery.
      * More efficient message deliver/consume mechanism
      * Better for large volume of messages – batch delivery.
    - Acknowledgement Deadline
      * Per subscriber
      * Once a deadline has passed, an outstanding message becomes unacknowledged.
  + When acknowledged, it is removed from the subscriptions message queue.
* **Architecture**
  + Data Plane
    - Handles moving messages between publishers and subscribers
    - Forwarders
  + Control Plane
    - Handles assignment of publishers and subscribers to server on the data plane.
    - Routers
* **Use Cases**
  + Balancing workloads in a network cluster
  + Implementing async workflows
  + Distributing event notifications
  + Refreshing distributed caches
    - I.e. An app can publish invalidation events to update the IDs of objects that have changed
  + Logging to multiple systems
  + Data streaming from various processes or devices
  + Reliability improvement
    - I.e. a single-zone GCE service can operate in additional zones by subscribing to a common topic, to recover from failures in a zone or region.
* **Deduplicate**
  + Database table to store hash value and other metadata for each data entry.
  + **Message\_id** can be used to **detect duplicate messages**
* **Out of Order Messaging**
  + Messages may arrive from multiple sources out of order.
  + Pub/Sub does not care about message ordering
  + Dataflow is where out of order messages are processed/resolved.
    - Ingest – Pub/Sub
    - Process - Dataflow
  + Can add message attributes to help with ordering.

**Machine Learning**

* Types of Problems:
  + Classification
  + Regression
  + Clustering
  + Rule Extraction
* Supervised Learning
  + Labels associated with the training data are used to correct the algorithm.
* Unsupervised Learning
  + The model has to be set up right to learn the structure in the data.
* Representation Learning Algorithms
  + Feature learning. Algorithm identifies important features on its own.
* Deep Learning
  + Algorithms that learn what features matter.
  + Neural Networks
    - Most common class of deep learning algorithms.
    - Used to build representation learning systems.
    - Composed of neurons (binary classifiers)
    - Wide
      * Better for memorization
    - Deep
      * Better for generalization
  + Neurons
    - Apply 2 functions on inputs.
      * Linear (affine) transformation
        + Like linear regression.
        + X1 \* W1 + b

W = Weights

Shape of W

First dimension is equal to number of dimensions of feature vector.

Second dimension is equal to the number of params required to be tuned. (same goes for b)

B = Bias

Determined during training process.

* + - * Activation Function
        + Helps to model non linear functions. (Logistic regression)
        + Introduces non-linearity into the network.
        + Ex.

ReLu (Rectified Linear Unit)

Max(Wx + b, 0)

SoftMax (Logistic Regression)

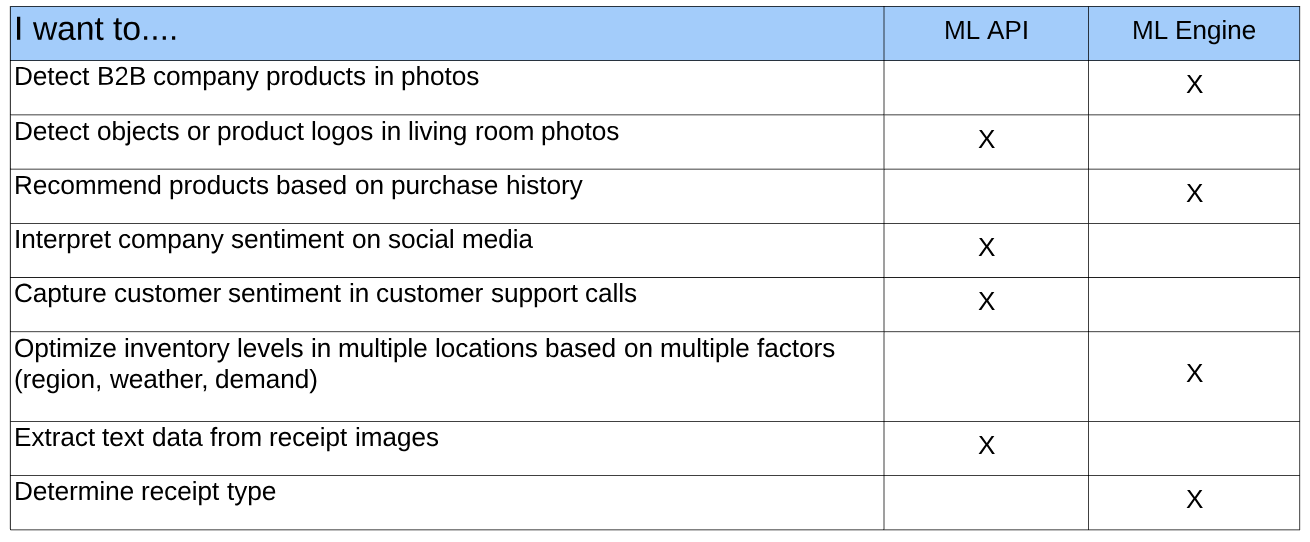
* + - * Best values of W and b found by using cost function, optimizer, and training data.
  + Back Propagation
* Modeling Linear Regression
  + 1 neuron with just an affine transformation.
  + Y = Ax + b
  + Minimize Least Square Error
* Optimizers for Best Fit
  + Method of Moments
  + Method of Least Squares
  + Maximum likelihood Estimator
* **Reducing Loss**
  + Hyperparameters are the configuration settings used to tune hot the model is trained.
    - Steps
      * Total number of training iterations. One step calculates the loss from one batch and uses that value to modify the model’s weights once.
    - Batch Size
      * Number of examples (chosen at random) for a single step.
      * Total # of trained examples = Batch Size \* Steps
    - Learning rate
  + Convergence: When loss stops changing or at least changes extremely slowly.
  + Gradient is a vector.
  + Learning rate is a scalar.
  + Gradient is multiplied by the learning rate.
  + Stochastic Gradient Descent
    - Random samples from data set to estimate.
    - Uses a batch size of 1 per iteration.
    - Works (given enough iterations), but noisy
  + Mini-Batch Stochastic Gradient Descent
    - Compromise between full batch and SGD
    - Typically between 10 – 10K examples chosen at random.
    - Reduce noise, but still more efficient than full-batch.
* Periods
  + # of training examples in each period = batch size \* steps / period
  + Controls granularity of reporting.
    - If periods = 7 and steps = 70, the loss value will be output every 10 steps.
  + Modifying period value does not alter what model learns.
* **Generalization**
  + The less complex an ML model, the more likely that a good empirical result is not just due to the peculiarities of the sample.
  + Overfitting occurs when a model tries to fit the training data so closely that it does not generalize well to new data.
  + Identify Overfitting
    - Loss for the validation set is significantly higher than for the training set. (look at loss curve (loss/iterations))
    - Validation loss eventually increases with iterations.
  + If the key assumptions of supervised ML are not met, then we lose important theoretical guarantees on our ability to predict new data.
  + 3 Basic Assumptions
    - We draw examples **independently and identically** at random from the distribution. I.e. examples don’t influence each other.
    - The distribution is **stationary**; that is it does not change within the data set.
    - We draw examples from partitions from the same distribution.
* **Training, Validation, and Test Sets**
  + Training set – a subset to train a model.
  + Test set – a subset to test the trained model.
    - Must be large enough to yield statistically meaningful results.
    - Is representative of the data set as a whole. i.e. don’t pick a test set with different characteristics than the training set.
  + Doing many rounds of just using a training and test set might cause implicit fitting to the peculiarities of the specific test set.
    - Use a validation set too!
    - Flow
      * Train model
      * Use model on validation set
      * Update hyperparams
      * Repeat
      * Finally test on test set
* **Representation**
  + Process of mapping data to useful features.
  + Discrete feature
    - A feature with a finite set of possible values.
    - Categorical feature are an example
  + One-Hot Encoding
    - A sparse vector in which:
      * One element is set to 1
      * All other elements are set to 0
    - Commonly used to represent strings or identifiers that have a finite set of possible values.
  + Feature Engineering
    - Process of determining which features might be useful in training a model, and then converting raw data from log files and other sources into said features.
    - Sometimes called feature extraction.
  + Qualities of Good Features
    - Avoid rarely used discrete feature values.
      * Should appear more than 5 or so times in a data set.
      * Having many examples with the same discrete value gives the model a chance to see the feature in different settings, and in turn, determine when it’s a good predictor for the label.
    - Prefer clear and obvious meanings
      * Ex. house\_age\_years vs. house\_age
      * Some cases, noisy data causes unclear values, such as data coming from sources that didn’t check for appropriate values.
        + Ex. user\_age\_years: 277
    - Don’t mix “magical” values with actual data
      * Ex. quality\_rating between 0 and 1.
        + If no value, it is set to -1
        + Create a Boolean feature to indicate if quality rating was defined.
      * Replace “magical” values as follows
        + For a variable that take a finite set of values (discrete variables), add a new value to the set and use it to signify that feature value is missing.
        + For continuous variables, ensure missing values do not affect the model by using the mean value of the feature’s data.
    - Account for upstream instability
      * Definition of a feature shouldn’t change over time.
  + Cleaning Data
    - Scaling feature vectors
      * Converting floating point feature values from their natural range (100 to 900) to a standard range (0 to 1 or -1 to 1)
      * Scaling ~= Normalization
      * If only 1 feature, little to no practical benefit.
      * Multiple features, great benefits
        + Helps gradient descent convere more quickly
        + Helps avoid NaN traps

One number in the model becomes a NaN (value exceeds floating point precision limit during training) and due to math operations, every other number in the model also eventually becomes NaN.

* + - * + Helps the model learn appropriate weights for each feature. Without scaling, the model pays too much attention to features having a wider range.
    - Handling extreme outliers
      * Log scaling
        + Still leaves a tail on distribution
      * Cap or Clipping
        + Reduce feature values that are greater than a set maximum value down to that maximum value.
        + Also, increasing feature values that are less than a specific minimum value up to that minimum value.
    - Binning (Bucketing)
      * Converting a (usually continuous) feature into multiple binary features called buckets or bins, typically based on a value range.
    - Scrubbing
      * Data can be unreliable due to:
        + Omitted values
        + Duplicate examples
        + Bad labels
        + Bad feature values
      * “Fix” by removing them from data set.
      * Omitted and duplicate easy to detect.
      * Detecting bad data in aggregate by using Histograms
      * Stats can also help identifying bad data:
        + Max and Min
        + Mean and Median
        + Standard Deviation
    - Follow These Rules:
      * Keep in mind what your data should look like
      * Verify that the data meets these expectations
        + Or that you can explain why it doesn’t
      * Double check that the training data agrees with other soruces
        + I.e. dashboards
* **Feature Crosses**
  + A synthetic feature formed by crossing (Cartesian product) individual binary features obtained from categorical data or from continuous features via bucketing.
  + Helps represent nonlinear relationships.
  + Encoding Nonlinearity
  + Crossing One-Hot Vectors
* **Regularization**
  + Minimize loss + complexity
    - Structural Risk Minimization
    - Penalizes complexity to prevent overfitting
  + 2 Common Ways to Think About Model Complexity
    - As a function of the weights of all the features in the model
      * **L2 Regularization**
      * A feature weight with a high absolute value is more complex than one with a low absolute value.
      * L2 = w1^2 + w2^2 + … + wn^2
      * Consequences of L2 Regularization
        + Encourages weight values toward 0 (but not exactly 0)
        + Encourages the mean of the weights toward 0, with a normal (bell shaped or Gaussian) distribution.
    - As a function of the total number of features with nonzero weights
  + Most developer tune the overall impact of the regularization term by multiplying it by a scalar known as **lambda (regularization rate)**
    - Minimize(loss function + lambda(regularization function))
    - When choosing a lambda value, the goal is to strike the right balance between simplicity and training-data fit
      * Lambda too high
        + Model will be simple but run the risk of underfitting data.
      * Lambda too low
        + Model will be more complex and run the risk of overfitting data.
  + Early Stopping
    - Ending training before the model reaches convergence (training loss finishes decreasing).
    - End model training when loss on a validation dataset starts to increase, that is, when generalization performance worsens.
* **Logistic Regression**
  + A model that generates a probability for each possible discrete label value in classification problems by applying a **sigmoid function** to a linear prediction.
  + Often used in binary classification problems, but can also be used in multi-class classification problems (multinomial regression)
  + Sigmoid Function
    - Maps logistic or multinomial regression output (log odds) to probabilities, returning a value between 0 and 1.
    - Can serve as an activation function in neural networks.
  + Loss and Regularization
    - Loss function is Log Loss
    - Regularization
      * L2 or Early Stopping
* **Classification**
  + Classification Threshold (Decision Threshold)
    - Determines what the probability output from logistic regression is classified as.
  + Accuracy
    - Number of correct predictions over total number of predictions
    - TP + TN / (TP + TN + FP + FN)
  + Class Imbalanced Dataset
    - Labels have significantly different frequencies in a classification problem.
    - Accuracy is not enough in this scenario.
  + Confusion Matrix
    - An NxN table that summarizes how successful a classification model’s predictions were.
    - Useful when calculating precision and recall
  + Precision
    - Identifies the frequency with which the model was correct when predicting the positive class.
    - TP/ (TP + FP)
    - i.e. how many predicted cats are actually cats
    - Raising classification threshold reduces FP, thus improving precision.
  + Recall
    - Out of all the possible positive labels, how many did the model correctly identify.
    - TP / (TP + FN)
    - i.e. number of predicted cats out of all cats
    - Raising classification threshold will cause # of TP to decrease or stay the same and will cause the # of FN to increase or stay the same. Thus recall will either stay constant or decrease.
  + Improving precision often reduces recall and vice versa.
  + ROC Curve
    - Receiver Operating Characteristic Curve
    - Shows performance of classification model at all classification thresholds.
    - TP rate (TP / TP + FN) vs. FP rate (FP / FP + TN)
    - Lowering classification threshold increase TP and FP.
  + AUC
    - Area Under the ROC Curve
    - Provides an aggregate measure of performance across all possible classification thresholds.
    - 0 – worst model
    - 1 – best model
    - Desirable Because:
      * Scale Invariant
        + Measures how well predictions are ranked, rather than their absolute values.
      * Classification Threshold Invariant
        + Measures the quality of the model’s predictions irrespective of what classification threshold is chosen.
    - Limitations
      * Scale invariance is not always desirable
        + We may need well calibrates probability outputs and AUC won’t tell us that.
      * Classification threshold invariance is not always desirable
        + In cases where there are wide disparities in the cost of false negatives vs. false positives, it may be critical to minimize one type of classification error.
  + Prediction Bias
    - = average of predictions – average of labels
    - Different than bias, b, in wx + b
    - Possible root causes of prediction bias:
      * Incomplete feature set
      * Noisy data set
      * Buggy pipeline
      * Biased training sample
      * Overly strong regularization
    - Avoid Calibration Layer as a fix
      * Fixing symptoms rather than cause.
      * Built a more brittle system that you must now keep up to date.
    - Examine prediction bias on a bucket of examples

**Pre-trained ML API’s**

* For App Developers
* Sight
  + **Vision AI**
    - Image Recognition/analysis
    - Label Detection
      * Extracts info in image across categories
    - Text Detection (OCR)
      * Detect and extract text from images
    - Safe Search
      * Recognize explicit content
    - Landmark Detection
    - Logo Detection
    - Image Properties
      * Dominant colors, pixel counts
    - Crop Hints
      * Crop coordinates of dominant object/face
    - Web Detection
      * Find matching web entries
  + **AutoML Vision**
    - Image Classification
    - Object Detection
    - **Edge**
  + Video AI
    - Video analysis
    - Labels, shot changes, explicit content
* Language
  + Natural Language
    - Text analysis
    - Extract information
    - Understand sentiment
  + Translation
    - Detect and translate languages
* Conversation
  + Cloud Speech-to-Text API
    - Convert audio to text
    - Multi-lingual support
    - Understand sentence structure
  + Cloud Text-to-Speech API
    - Convert text to audio
    - Multiple languages/voices
    - Natural sounding synthesis
  + **Dialogflow Enterprise Edition**
    - Conversational experiences
    - Virtual assistants
* Structured Data
  + AutoML Tables
  + Cloud Inference API
  + Recommendations AI (Beta)
  + BigQuery ML (beta)
* **Cloud AutoML**
* Cloud Job Discovery
  + More revelant job searches
  + Power recruitment, job boards
* Basic Steps for Most APIs
  + Enable API
  + Create API key
  + Authenticate with API key
  + Encode in base64 (optional)
  + Make an API request
  + Requests and outputs via JSON
* Pricing
  + Pay per API request per feature
  + Feature as in Landmark Detection
* How to convert images, video, etc for use with API?
  + Can use Cloud Storage URI for GCS stored objects
  + Encode in base64 format
* How to combine API’s for scenarios?
  + Search customer service calls and analyze sentiment
    - Speech to Text then Sentiment Analysis with Natural Language

****

* Vision API
* Speech API
* Natural Language API
* Translate API

**AI Platform**

* Can use multiple ML platforms such as **TensorFlow, scikit-learn** and **XGBoost**
* Workflow
  + **Source and prepare data**
    - Data analysis
      * Join data from multiple sources and rationalize it into one dataset.
      * Visualize and look for trends.
      * Use data centric languages and tools to find patterns in data.
      * Identify features in your data.
      * Clean the data to find any anomalous values caused by errors in data entry or measurement.
    - Data preprocessing
      * Transform valid, clean data into the format that best suits the needs of your model.
      * Examples
        + Normalizing numeric data to a common scale.
        + Applying formatting rules to data. Ex. removing HTML tagging from a text feature.
        + Reducing data redundancy through simplification. Ex. converting a text feature to a bag of words representation.
        + Representing text numerically. Ex. assigning values to each possible value in a categorical feature (or 1 hot).
        + Assigning key values to data instances.
  + **Develop model**
  + **Train an ML model on your data**
    - 3 subsets of data
      * Training
      * Validation
      * Testing
    - Benefits of Training Locally
      * Quick iteration
      * No charge for cloud resources
  + **Deploy trained model**
    - Upload to GCS bucket
    - Create a model resource in AI Platform specifying GCS path
  + **Send prediction requests to your model**
    - Online
    - Batch
  + **Monitor predictions on an ongoing basis**
    - APIs to examine running jobs.
    - Stackdriver
  + **Manage models and model versions**
    - gcloud ai-platform
* Preparing Data
  + Gather data
  + Clean data
    - Clean data by column (attribute)
    - Instances with missing features.
    - Multiple methods of representing a feature.
      * Length measurement in different scale/format
    - Features with values far out of the typical range (outliers)
    - Significant change in data over distances in time, geographic location, or other recognizable characteristics.
    - Incorrect labels or poorly defined labeling criteria.
  + Split data
    - Train, Validation, Test
    - Better to randomly sample the subsets from one big dataset than use pre-divided data. Otherwise could be non-uniform => overfitting.
    - Size of datasets: training > validation > test
  + Engineer data features
    - Can combine multiple attributes to make one generalizable feature.
      * Address and timestamp => position of sun
    - Can use feature engineering to simplify data.
    - Can get useful features and reduce number of instances in dataset by engineering across instances. I.e. calculate frequency of something.
  + Preprocess features
* Training Overview
  + Upload datasets already split (training, validation) into something AI Platform can read from.
  + Sets up resources for your job. One or more virtual machines (training instances)
    - Applying standard machine image for the version of AI Platform your job uses.
    - Loading application package and installing it with pip.
    - Installing any additional packages that you specify as dependencies.
  + Distributed Training Structure
    - Running job on a given node => **replica**
    - Each replica given a single role or task in distributed training:
      * **Master**
        + Exactly 1 replica
        + Manages others and reports status for the job as a whole.
        + Status of master signals overall job status.
        + Single process job => the sole replica is the master for the job
      * **Worker(s)**
        + 1 or more replica
        + Do work as designated in job configuration.
      * **Parameter Servers**
        + 1 or more replicas
        + Coordinate shared model state between the workers.
    - Tiers
      * **Scale tiers**
        + Number and types of machines you need.
      * **CUSTOM tier**
        + Allows you to specify the number of Workers and parameter servers.
      * Add these to TrainingInput object in job configuration.
    - **Exception**
      * The training service runs until your job succeeds or encounters an unrecoverable error.
      * Distributed Case – status of the master replica that signals the overall status.
      * Running a Cloud ML Engine training job locally (gcloud ml-engine local train) is especially useful in the case of testing distributed models.
  + Start training
    - Package application with any dependencies required
    - 2 ways
      * Submit by running `gcloud ai-platform jobs submit training`
      * Send a request to the API ar `projects.jobs.create`
        + Need `ml.jobs.create` permission.
    - Job ID
      * Define base name for all jobs associated with a given model and then append a data/time.
    - Job-Dir
      * Save model checkpoints to this GCS path.
      * Useful for VM restarts.
      * Used for job output.
    - GPUs
      * More effective at running certain operations on tensor data than adding another machine with one or more CPU cores.
      * Can specify GPU-enabled machines to run your job.
    - TPUs
      * Tensor Processing Units
      * Google’s custom developed ASICs used to accelerate machine learning workloads with TensorFlow.
      * Steps
        + Authorize Cloud TPU service account name associated with GCP project
        + Add service account as a member of your project with role **Cloud ML Service Agent**.
      * Only in us-central1 currently.
* Hyperparameter Tuning
  + –config hptuning\_config.yaml
  + Hyperparameter: Data that governs the training process itself.
    - DNN
      * Number of layers
      * Number of nodes for each layer
  + Usually constant during training.
  + How it works:
    - Running multiple trials in a single training job.
    - Each trail is a complete execution of your training application with values for chosen hyperparameters, set within limits specified.
  + Tuning optimizes a single target variable (hyperparameter metric)
    - Multiple params per metric.
  + Default name is `training/hptuning/metric`
    - Recommended to change to custom name.
      * Must set `hyperparameterMetricTag` value in `HyperparameterSpec` object in job request to match custom name.
  + How to actually tune?
    - Define a command line argument in main training module for each tuned hyperparameter.
    - Use value passed in those arguments to set the corresponding hyperparameter in application’s TensorFlow code.
  + Types
    - Double
    - Integer
    - Categorical
    - Discrete – List of values in ascending order.
  + Scaling
    - Recommended for Double and Integer types.
    - Linear, Log, or Reverse Log Scale
  + Search Algorithm
    - Unspecified
      * Same behavior as when you don’t specify a search algo.
      * Bayesian optimization
    - Grid Search
      * Useful when specifying a number of trials that is more than the number of points in feasible space.
        + In such cases AI Platform default may generate duplicate suggestions.
      * Can’t use with any params being Doubles
    - Random Search
* Online and Batch Prediction
  + Can process one or more instances per request.
  + Can serve predictions from a TensorFlow SavedModel.
  + Can make requests
    - Legacy Editor
    - Legacy Viewer (Online only)
    - AI Platform Admin or Developer
  + **Online**
    - Optimized to **minimize the latency** of serving predictions.
    - Predictions returned in the response message.
    - Input passed directly as a JSON string.
    - Returns as soon as possible.
    - Runs on runtime version and in region selected when deploying model.
    - Can serve predictions from a custom prediction routine.
    - Can generate logs if model is configured to do so. Must specify option when creating model resource.
      * onlinePredictionLogging or –enable-logging (gcloud)
    - Use when making requests in responses to application input or in other situations where timely inference is needed.
  + **Batch**
    - Optimized to **handle a high volume** of instances in a job and to run more complex models.
    - Predictions written to output files in **Cloud Storage location** that you specify.
    - Input data passed directly as one or more UIRs of files in Cloud Storage locations.
    - Asynchronous request.
    - Can run in any available region, using any runtime version.
      * Should run with defaults for deployed model versions.
    - Only Tensorflow supported. (Not XGBoost or scikit)
    - Ideal for processing accumulated data when you don’t need immediate results.
      * I.e. a periodic job that gets predictions for all data collected since the last job.
    - Generates logs that can be viewed on Stackdriver.
    - Slow because AI Platform allocates and initializes resources for a batch prediction job when the request is sent.
* Prediction Nodes and Resource Allocation
  + Think of a Node as a VM
  + Batch
    - Scales nodes to minimize elapsed time job takes.
    - Allocates some nodes to handle your job when you start it.
    - Scales the number of nodes during the job in an attempt to optimize efficiency.
    - Shuts down nodes as soon as job is done.
  + Online
    - Scales nodes to maximize number of requests it can handle without too much latency.
    - Allocates some nodes the first time you request predictions after a long pause in requests.
    - Scales number of nodes in response to request traffic, adding nodes when traffic increases, removing them when there are fewer requests.
    - Keeps at least 1 node ready over a period of several minutes, to handle requests even when there are none to handle.
    - Scales down to zero after model version goes several minutes without a prediction request.
* Predictions from Undeployed Models
  + Batch only
  + Specify URI of a GCS locations where the model is stored.
  + Explicitly set runtime version in request.

**TensorFlow**

* OS Machine learning/ Deep learning platform
* **Lazy evaluate** during build, full evaluate during execution.
* TensorFlow Estimator API
  + High level object oriented API
  + Makes it easy to build models.
  + Specifies predefined architectures, such as linear regressors or neural networks.
* Tf.layers, tf.losses, tf.metrics
  + Reusable libraries for common model components.
* Python TensorFlow
  + Provides Ops, which wrap C++ Kernels
* Can run on CPU, GPU, or TPU
  + Kernels work on more than one platform.
* Feature Engineering
  + Often means converting raw log file entries to tf.Example protocol buffers. See also tf.Transform

**Hadoop**

* Distributed
  + Lots of cheap hardware
    - HDFS
  + Replication and Fault Tolerance
    - YARN
  + Distributed Computing
    - MapReduce
* **HDFS**
  + GCS is used on GCP.
    - Don’t use HDFS as you would have to pay for a VM on Compute Engine.
  + Suited for batch processing.
    - Data access has high throughput rather than low latency.
  + **Architecture**
    - **Name Node**
      * 1 master node
      * Contains YARN resource manager
      * Manages overall file system
      * Stores
        + The directory structure
        + Metadata on the files
    - **Data Nodes**
      * Physically stores the data in the files.
  + Storing Data
    - Break data into blocks of equal size
      * Different length files are treated the same way
      * Storage is simplified
      * Unit for replication and fault tolerance
    - Blocks are of size 128 MB
      * Larger -> Reduces parallelism
      * Smaller -> Increases overhead (more metadata)
    - Stores the blocks across the data nodes
      * Each node contains a partition or a split of data
      * How do we know where the splits of a particular file are?
        + Name Node (File 1 | Block 1 | Data Node)
  + High Availability
    - Can have multiple name nodes.
    - Kept in sync with Zookeeper
  + Default Replication Strategy
    - Maximize Redundancy
      * 1st location chosen at random
      * 2nd has to be on a different rack (if possible)
      * 3rd will be on same rack as the second, but on a different node.
        + Reduces inter-rack traffic and improves write performance.
      * Read operations are sent to the rack closest to the client.
    - Minimize Write Bandwidth
      * Data is forwarded from first data node to the next replica location.
      * Forwarded further to the next replica location.
      * Forwarding requires a large amount of bandwidth.
        + Increases cost of writes.
* **MapReduce**
  + **Map**
    - An operation performed in parallel, on small portions of dataset.
    - Outputs KV pairs
  + **Reduce**
    - Mapper outputs become one final output.
  + SQL interface over MapReduce = Hive
    - Data analysts understand SQL but not Java code.
  + 1. What {key, value} pairs should be emitted in the map step?
  + 2. How should values with the same key be combined?
* **YARN (Yet Another Resource Negotiator)**
  + Coordinate tasks running on the cluster.
  + Assign new nodes in case of failure.
  + **Architecture**
    - **Resource Manager**
      * Runs on a single master node
      * Schedules tasks across nodes
      * Starts Application Master within containers.
    - **Node Manager**
      * Run on all other nodes
      * Manages tasks on the individual node.
      * Can have multiple containers.
      * Can request containers for mappers and reducers.
    - **Application Master**
      * If additional resources are required, Application Master makes the request.
      * 1 instance per application.
      * Client communicates directly to get status, progress updates via an application-specific protocol.
    - **Container**
      * All processes are run within a container in a Node Manager.
      * Package of resources including RAM, CPU, Network, HDD etc on a single node.
      * Executes the application code.
      * Can communicate with Application Master itself.
  + Location Constraint
    - Assign a process to the same node where the data to be processed lives.
    - If CPU/Memory not available, WAIT!
  + Scheduling Policies
    - FIFO Scheduler
      * Queue
    - Capacity Scheduler
      * Priority Queue
    - Fair Scheduler
      * Jobs assigned equal share of all resources
* **HBase**
  + Database management system on top of Hadoop.
  + Integrates with your application just like a traditional database.
  + **Columnar Store**
    - Advantages
      * Sparse Tables
        + No wastage of space when storing data.
      * Dynamic Attributes
        + Update attributes dynamically without changing storage structure.
        + Do not need to change schema.
  + **Denormalized Storage**
    - Column names repeat across rows.
    - Normalization Reduces data duplication => Optimizes storage.
      * Storage is cheap in a distributed file system.
      * Optimize number of disk seeks instead by denormalization.
        + Don’t have to join tables.
    - Read a single record to get all details about an employee in one read operation.
  + **Only CRUD operations**
    - No comparisons/sorting/inequality checks across multiple rows
      * No joins
      * No group by
      * No order by
    - No operations involving multiple tables
    - No indexes on tables
    - No constraints
  + **ACID at ROW level**
    - Updates to a single row are atomic
      * All columns are updated, or none are.
    - Updates to multiple rows are not atomic
      * Even if update is on the same column in multiple rows.
* **Hive**
  + Provides a SQL interface to Hadoop.
  + Bridge to Hadoop for people without OOP exposure.
  + Not suitable for very low latency apps due to HDFS.
  + HiveQL ~= SQL
  + Wrapper on top of MapReduce
  + Metastore
    - HCatalog
    - Bridge between HDFS and Hive
    - Stores metadata for all tables in Hive
    - Maps the files and directories in Hive to tables
    - Holds the definitions and the schema for each table
    - Any database with a JDBC driver can be used as a metastore.
    - Development
      * Use built-in Derby database
      * Embedded metastore
      * Only one session can connect.
    - Production
      * Local metastores
        + Allow multiple sessions to connect to Hive
        + DB is a separate process and can be on separate host.
      * Remote metastores
        + Separate processes for Hive and the metastore
        + Metastore runs in its own JVM process.
        + Processes communicate with Metastore using Thrift network API (hive.metastore.uris property)
        + Does not require admin to share JDBC login info for the metastore db with each Hive user.
    - Hive vs. RDBMS
      * Large vs. Small datasets
      * Parallel vs. serial computations
      * High vs. low latency
      * Read vs. Read/write operations
      * Not ACID compliant vs. ACID compliant
    - HiveQL vs. SQL
      * High latency
        + Records not indexed.
        + Fetching a row runs a MapReduce which may take minutes.
        + Not owner of the data.

It exists in HDFS

* + - * + Schema-on-read
      * Not ACID compliant
        + Data can be dumped into Hive tables from any source
      * Row level updates, deletes as a special case
      * Many more built in functions
      * Only equi-joins allowed
    - OLAP in Hive
      * Partitioning
        + State specific queries will run only on data in one directory.
        + Splits NOT of the same size.
      * Bucketing
        + Size of each split should be the same.

Hash of a column value

* + - * + Each bucket is a separate file
        + Makes sampling and joining data more efficient

Reduces search space

* + - * Join Optimizations
        + Join operations are Map Reduce jobs under the hood

Optimize joins by reducing the amount of data held in memory

* + - * + Reducing data held in memory

On a join, one table is held in memory while the other is read from disk

Hold smaller in memory

* + - * + Structuring Joins as Map-Only Operation

Filter queries (only these rows)

Mapper needs to use null as key

* + - * Windowing in Hive
        + A suite of functions which are syntactic sugar for complex queries.
        + Ex. What revenue percentile did this supplier fall into this quarter?

Window = 1 quarter

Operation = Percentile on revenue

* **Pig**
  + ETL
  + A data manipulation language
  + Transforms unstructured data into a structured format
  + Query this structured data using interfaces like Hive.
  + Raw Data -> Pig -> Warehouse -> HiveQL -> Analytics
  + Pig Latin
    - A procedural, data flow language to extract, transform and load.
      * Procedural
        + Uses a series of well-defined steps to perform operations.
        + No if statements or for loops.
        + Specifies exactly how data is to be modified at each step.
      * Data Flow
        + Focused on transformations applied to the data.
        + Written with a series of data operations in mind.
        + Nodes in a DAG
    - Data from one or more sources can be read, processed and stored in parallel.
    - Cleans data, precomputes common aggregates before storing in a data warehouse.
  + Pig on Hadoop
    - Optimizes operations before MapReduce jobs are run, to speed operations up.
  + Works better with Apache Tez and Spark.
* **Spark**
  + A distributed computing engine used along with Hadoop
  + Interactive shell to quickly process datasets
  + Has a bunch of built in libraries for machine learning, stream processing, graph processing, …, etc.
  + Dataflow
  + General purpose
    - Exploring
    - Cleaning and Preparing
    - Applying machine learning
    - Building data applications
  + Interactive
    - Provides a REPL environment
      * Read Evaluate Print Loop
  + Reduces boilerplate of standard MapReduce Java code.
  + **Resilient Distributed Datasets (RDDs)**
    - In memory collections of objects.
    - Can interact with billions of rows
    - Properties
      * Partitions
      * Read-only
        + Immutable
        + Operations allowed on RDD

Transformations

Transform into another RDD

Actions

Request a result

* + - * Aware of it’s Lineage
        + When created, RDD knows

A transformation

It’s parent RDD

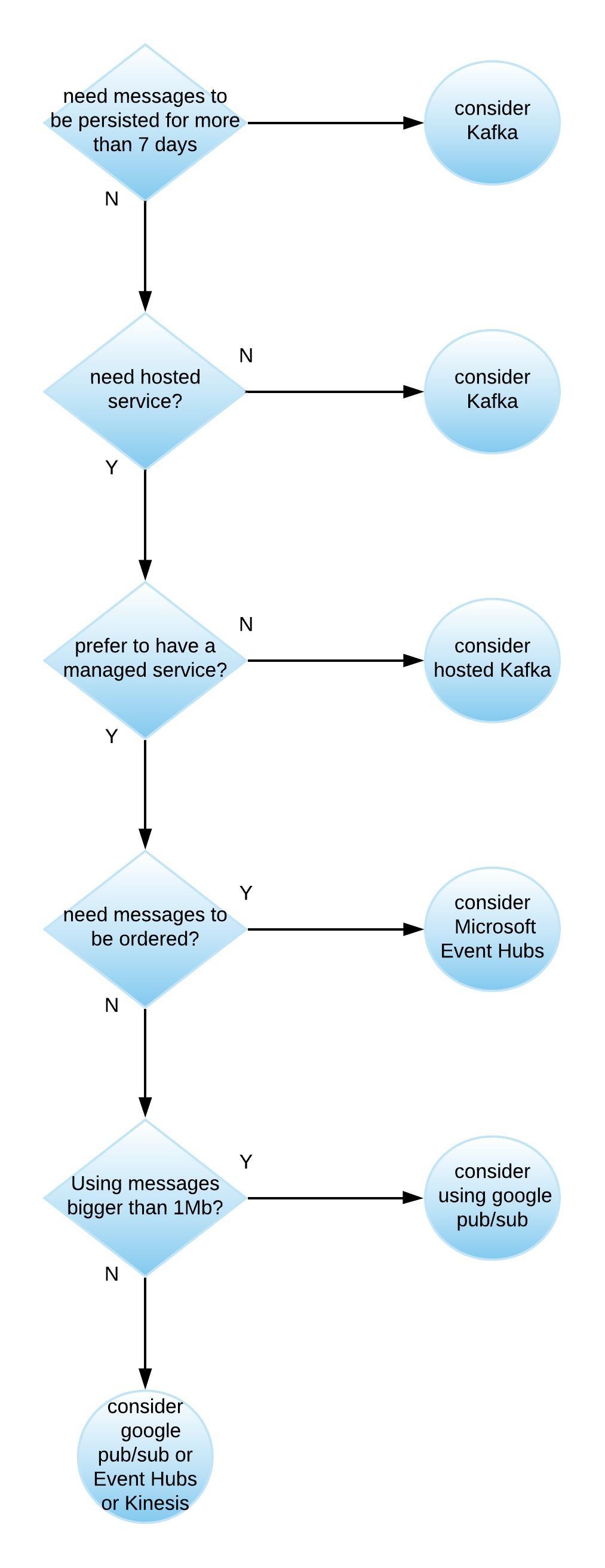
* + - * + Implications of Lineage

Built in fault tolerance

Reconstruct from source if something goes wrong

Lazy Evaluation

Materialize only when necessary

* + Spark Core
    - Basic functionality of Spark
    - Written in Scala
    - Runs on a Storage System and Cluster Manager
      * Plug and play components
      * Can be HDFS and YARN
  + Spark ML
    - **TODO**
* **Oozie**
  + A tool used to schedule workflows on all the Hadoop ecosystem technologies.
* **Kafka**
  + Stream processing for unbounded datasets.
  + Similar to PubSub
  + **Compared to PubSub**
    - Can have precisely once delivery with Spark direct Connector in addition to at least once.
      * Only at least once with PubSub
    - Guaranteed ordering within a partition.
      * No ordering guaranteed with PubSub
    - No max persistence period
      * 7 days or until acknowledged by all subscribers for PubSub
    - Partitioning under user control
      * Partitioning control abstracted away with PubSub
    - Cluster Mirroring for disaster recovery
      * Automated disaster recover for PubSub
    - 1MB max size for data blobs
      * 10 MB max size for PubSub
    - Can change partitioning after setup (does not repartition existing data)
      * Not under user control with PubSub
    - Pseudo push model supported using Spark.
    - ****
* **Streams Intro**
  + How can MapReduce be used to maintain a running summary of real-time data from sensors?
    - Send temp readings every 5 minutes
  + **Batches**
    - Bounded datasets
    - Slow pipeline from data ingestion to analysis
    - Periodic updates as jobs complete
    - Order of data received unimportant
    - Single global state of the world at any point in time
    - Typically small/singular source
    - Low latency not important
    - Often stored in storage services GCS, Cloud SQL, BigQuery
  + **Streams**
    - Unbounded datasets
    - Processing immediate, as data is received
    - Continuous updates as jobs run constantly
    - Order important, but out of order arrival tracked
    - No global state, only history of events received
    - Typically many sources sending tiny (KB) amounts of data
    - Requires low latency
    - Typically paired with Pub/Sub (ingest) and Dataflow (real-time processing)
  + Process data one entity at a time or a collection of entities as a batch
    - Filter error messages (logs)
    - Find a reference to latest movies (tweets)
    - Track weather patterns (sensor data)
  + Store, display, act on filtered messages
    - Trigger an alert
    - Show trending graphs
    - Warn of sudden squalls
  + **Stream-First Architecture**
    - Data items can come from multiple sources
      * Files, DBs, but at least one from a Stream
    - All files are aggregated and buffered in one way by a Message Transport (Queue)
      * i.e. Kafka, PubSub
    - Passed to Stream Processing system
      * Flink or Spark Streaming
  + **Micro-batches**
    - Message Transport
      * Buffer for event data
      * Performant and persistent
      * Decoupling multiple source from processing
    - Stream Processing
      * High throughput, low latency
      * Fault tolerant with low overhead
      * Manage out of order events
      * Easy to use, maintainable
      * Replay streams
    - A good approximation of stream processing is the use of micro-batches
      * Group data items (time they were received)
      * If small enough it approximates real-time stream processing
    - Advantages
      * Exactly once semantics, replay micro-batches
      * Latency-throughput trade off based on batch sizes
        + Can adjust to use case
        + Low latency better
        + High throughput better
    - Spark Streaming or Storm Trident

**Security**

* **Cloud IAM**
  + Provides administrators the ability to manage cloud resources centrally by controlling who can take what action on specific resources.
  + <https://cloud.google.com/docs/enterprise/best-practices-for-enterprise-organizations#identity-and-access-management>
  + <https://cloud.google.com/bigquery/docs/access-control>
* **Data Loss Prevention API**
  + Handle sensitive data (especially redaction of PII data)
  + Understand encryption techniques (in Cloud Storage Section)

**Cloud Composer**

* Fully managed workflow orchestration service based on Apache Airflow.
  + No need to provision resources.
* Pipelines are configured as DAGs
* Workflows live on-premises, in multiple clouds, or full within GCP
* Provides ability to author, schedule, and monitor your workflows in a unified manner.
* Multi-cloud
* Can use Python to dynamically author and schedule workflows.
* **Environments**
  + Airflow is a micro-service architected framework.
    - To deploy in a distributed setup, Cloud Composer provisions several GCP components, collectively known as an Environment.
  + Can create one or more inside a project.
  + Self contained Airflow deployments based on GKE.
  + Work with GCP services through connectors built into Airflow.
* **Architecture**
  + Distributes environment’s resource between a Google-managed tenant project and a customer project.
  + For unified Cloud IAM access control and an additional layer of data security, Cloud Composer deploys Cloud SQL and App Engine in the tenant project.
  + Tenant Project
    - Cloud SQL
      * Stores Airflow metadata.
      * Composer limits database access to the default or specified custom service account used to create the environment.
      * Metadata backed up daily.
      * Cloud SQL proxy in GKE cluster
        + Used to remotely authorize access to your Cloud SQL database from an application, client, or other GCP service.
    - App Engine
      * Hosts the Airflow web server.
      * Integrated with Cloud IAM by default.
      * Assign composer.user role to grant access only to Airflow web server.
      * Can deploy a self-managed Airflow web server in customer project (for orgs with additional access-control reqs)
  + Customer Project
    - Cloud Storage
      * Used for staging DAGs, plugins, data dependencies, and logs.
      * To deploy workflows (DAGs), copy files to the bucket for you environment.
      * Composer takes care of synchronizing DAGs among workers, schedulers, and the web server.
    - GKE
      * Scheduler, worker nodes, and CeleryExecutor here.
      * Redis
        + Message broker for the CeleryExecutor

Runs a StatefulSet application so that messages persist across container restarts.

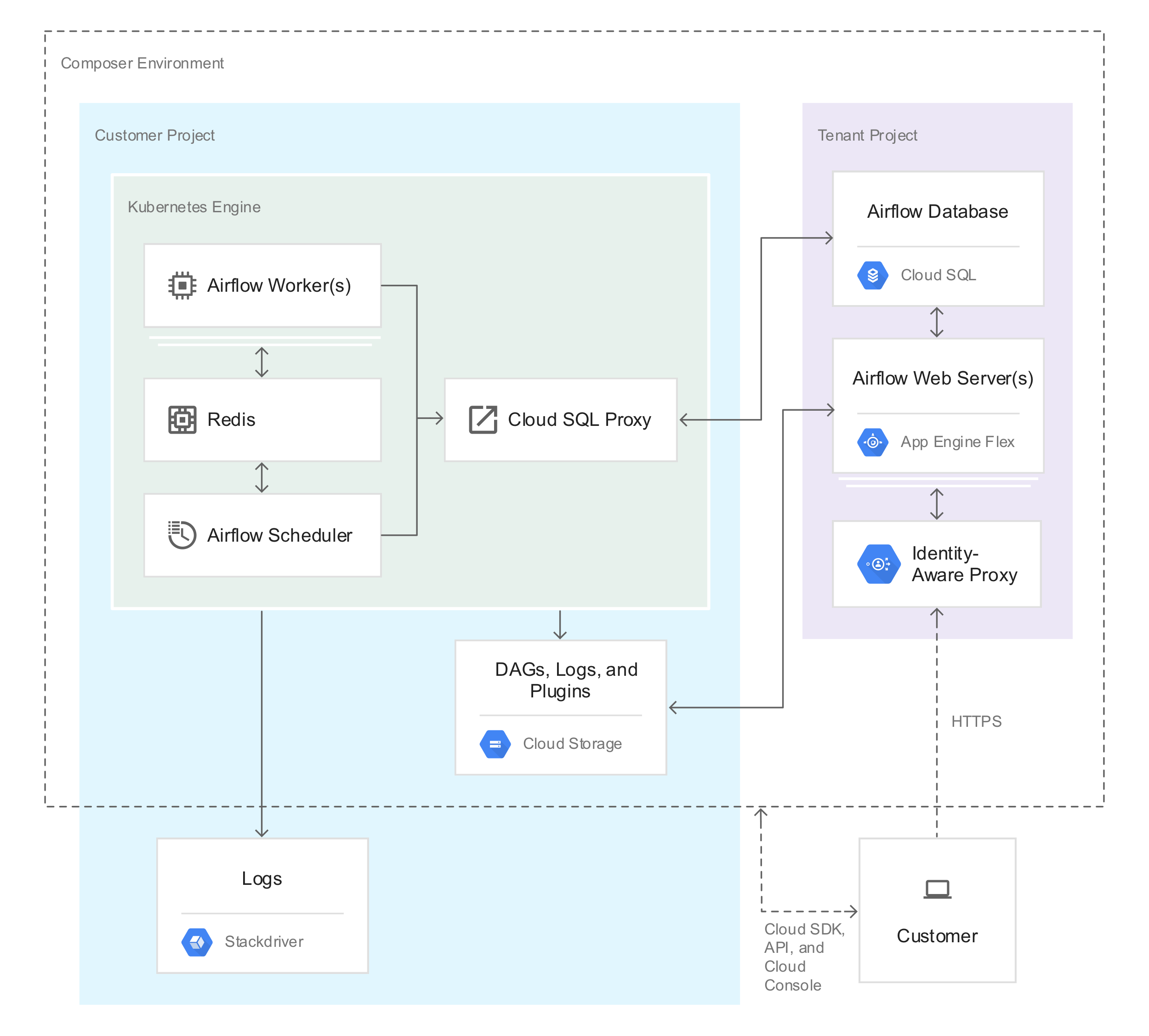
* + - * Allows use of KubernetesPodOperator to run any container workload.
      * Composer enables auto-upgrade and auto-repair to protect against security vulnerabilities.
        + Can perform manual upgrade too.
      * Service Accounts
        + Worker and scheduler nodes and the web server run on different service accounts.
        + Scheduler and workers

If service account is not specified during environment creation, default Compute Engine service account is used.

* + - * + Web Server

Auto generated during environment creation and derived from webserver domain.

* + Stackdriver
    - Integrates to have a central place to view all Airflow service and workflow logs.
    - Can view logs of scheduler and worker emit immediately instead of waiting for Airflow logging module synchronization (due to streaming nature of Stackdriver)
  + Airflow
    - DAGs
      * Composed of Tasks
      * Connects independent tasks and executes in specified sequence.
    - Tasks
      * Created by instantiating an Operator class.
      * Logical unit of code.
      * Link Tasks/Operators in your DAG python code.
    - Operators
      * Template to wrap and execute a task.
      * BashOperator is used to execute a bash script.
      * PythonOperator is used to execute python code.
      * Specify DAG when instantiating Operator.
      * Sensors
        + Special type of Operator that will keep running until a certain criterion is met.
    - Task Instance
      * Represents a specific run of a task is is characterized by a combination of a dag, a task, and a point in time.
      * Has a state (running, success, failed, skipped, up for retry, etc)
    - CeleryExecutor
      * Used to execute multiple DAGs in parallel
      * Requires a message broker.
      * SequentialExecutor is used for one DAG at a time.



**Cloud Dataprep**

* Intelligent data preparation
* Partnered with Trifecta for data cleaning/processing service
* Fully managed, serverless, and web based
* User friendly interface
  + Clean data by clicking on it
* Supported file types
  + Inputs
    - CSV, JSON, Plain Text, Excel, LOG, TSV, and Avro
  + Outputs
    - CSV, JSON, Avro, BQ Table
      * CSV/JSON can be compressed or uncompressed
* How it works
  + Backed by Cloud Dataflow
    - After preparing Dataflow processes via Apache Beam pipeline
    - “User-friendly Dataflow pipeline”
  + Dataprep Process
    - Import data
    - Transform sampled data with recipes
    - Run Dataflow job on transformed dataset
      * Batch Job
      * Every recipe step is its own transform
    - Export results (GCS, BigQuery)
* Intelligent Suggestions
  + Selecting data will often automatically give the best suggestion
  + **Can manually create recipes, however simple tasks (remove outliers, de-duplicate) should just use auto-suggestions.**
* **IAM**
  + Dataprep User
    - Run Dataprep in a project
  + Dataprep Service Agent
    - Gives Trifecta necessary access to project resources.
      * Access GCS buckets, Dataflow Developer, BQ user/data editor
      * Necessary for cross-project access + GCE service account
* Pricing
  + 1.16 \* cost of a Dataflow job
* Flows
  + Add or import datasets to process with recipes
  + Public Bucket for testing: gs://dataprep-samples
  + For large datasets:
    - UI only shows a sample to work with
    - Recipe created is then applied to entirety of dataset
* Jobs
  + Create a dataset in BQ first
  + Click on Run Job
    - Default option is CSV in GCS bucket
    - Choose BQ dataset instead
    - Name table
    - Run Job: Create Apache Beam pipeline with Dataflow

**Data Studio**

* Easy to use data visualization and dashboards.
  + Drag and drop report builder.
* Part of G Suite, not GCP:
  + Uses G Suite access/sharing permissions, not Google Cloud (no IAM)
  + Google account permissions in GCP will determine data source access.
  + Files saved in Google Drive.
* Connect to many Google, Google Cloud, and other services:
  + BQ, Cloud SQL, GCS, Spanner
  + YouTube Analytics, Sheets, AdWords, local upload
    - Local
      * Stored in managed GCS bucket
      * First 2GB free
  + Many third party integrations
* Price
  + Free
  + BQ access run normal query costs
* Data Lifecycle
  + Visualization
* Basic Process
  + Connect to data source
  + Visualize data
  + Share with others
* Creating Charts
  + Use combinations of dimensions and metrics
  + Create custom fields if needed
  + Add date range filters with ease
* Caching (most relevant for BQ)
  + Options for using cached data performance/costs
  + 2 choices
    - Query Cache
      * Remembers query issues by reports components (i.e. charts)
      * When performing same query, pulls from cache.
      * If query cache cannot help, goes to prefetch cache.
      * Cannot be turned off.
    - Prefetch Cache (exam material?)
      * “Smart Cache” – predicts what might be requested
      * If prefetch cache cannot serve data, pulls from live data set
      * Only active for data sources that use owner’s credentials for data access
        + If I create table that pulls from BQ table that does not use my credentials for data access, prefetch will be disabled.
      * Can be turned off.
  + When to turn caching off:
    - Need to view “fresh data” from rapidly changing data set.

**Transfer Appliance**

* Transfer large amounts of data quickly and cost-effectively to GCP.
* Transfers directly to GCS or BQ
* Data Size >= 20TB
* Offline Data Transfer
* Takes more than 1 week to upload data.
* Workflow
  + Receive Transfer Appliance and configure it and connect it to your network.
  + Before data is stored, it is deduplicated, compressed and encrypted with AES 256 algorithm using a password and passphrase specified by user.
  + Data integrity check is performed.
  + Transfer Appliance is shipped back to Google.
  + Encrypted data is copied to GCS staging bucket.
    - Still compressed, deduplicated, and encrypted.
  + Email will be sent to user notifying the rehydration process can start.
  + Transfer Appliance Rehydrator application is run specifying the GCS destination bucket.
    - This application is run on GCE.
    - Compared CRC32C hash value of each file being rehydrated.
    - If checksums don’t match, file is skipped and appears in skip file list with Data corruption detected.
  + Data integrity check performed again.
  + Appliance securely wiped and re-imaged.
* Use Cases:
  + Data Collection
    - Geographical, environmental, medical, or financial data for analysis.
    - Need to transfer data from researchers, vendors, or other sites to GCP.
  + Data Replication
    - Supporting current operations with existing on prem infrastructure but experimenting with cloud.
    - Allows decommissioning duplicate datasets, test cloud infrastructure, and expose data to machine learning analysis.
  + Data Migration
    - Offline data transfer is suited for moving large amounts of existing backup images and archives to ultra-low-cost, highly durable, and highly available archival storage (Nearline/Coldline).

**Kubeflow**

<https://developers.google.com/machine-learning/crash-course/>

**Practice Exam Missed Questions**

2 Benefits of Denormalized Data in BigQuery

* Increased query performance
* Decreased query complexity (no joins needed)

You are designing storage for CSV files and using an I/O-intensive custom Apache Spark transform as part of deploying a data pipeline on Google Cloud. You are using ANSI SQL to run queries for your analysts. You want to support complex aggregate queries and reuse existing code. How should you transform the input data?

Use BigQuery for storage. Use Cloud Dataproc to run the transformations.

As part of your backup plan, you create regular boot-disk snapshots of Compute Engine instances that are running. You want to be able to restore these snapshots using the fewest possible steps for replacement instances. What should you do?

Use the snapshots to create replacement instances needed.

You are developing an application that will only recognize and tag specific business to business product logos in images. What is the best method to accomplish this task?

Create a custom machine learning model to recognize specific logos in photos, then train it on Cloud AI Platform.

Your BigQuery dataset contains 1500 tables. When conducting a query, you are limited to a maximum of 1000 tables that you can query at once. You need to query data across all 1500 tables. What should you do?

If possible, merge the 1500 tables to bring the total number below 1000. You may still partition single tables to divide data for queries.

Creating multiple views of chunks of the 1500 tables will not work as it will still limit to 1000 tables per query, even behind views.

Flowlogistic needs to run analytics on their incoming inventory data. They need to use their existing Hadoop workloads to perform this task. What two steps must be performed to accomplish this? (Choose all that apply)

Stream from PubSub into Dataproc, which can then place relevant data in the appropriate storage location.

Connect Dataproc to BigTable and Cloud Storage, running analytics on the data in both services.

You created a job which runs daily to import highly sensitive data from an on-premises location to Cloud Storage. You also set up a streaming data insert into Cloud Storage via a Kafka node that is running on a Compute Engine instance. You need to encrypt the data at rest and supply your own encryption key. Your key should not be stored in the Google Cloud. What should you do?

Supply your own encryption key, and reference it as part of your API service calls to encrypt your data in GCS and your Kafka node hosted on Compute Engine.

* Requires you to use your own key and not store it on GCP.