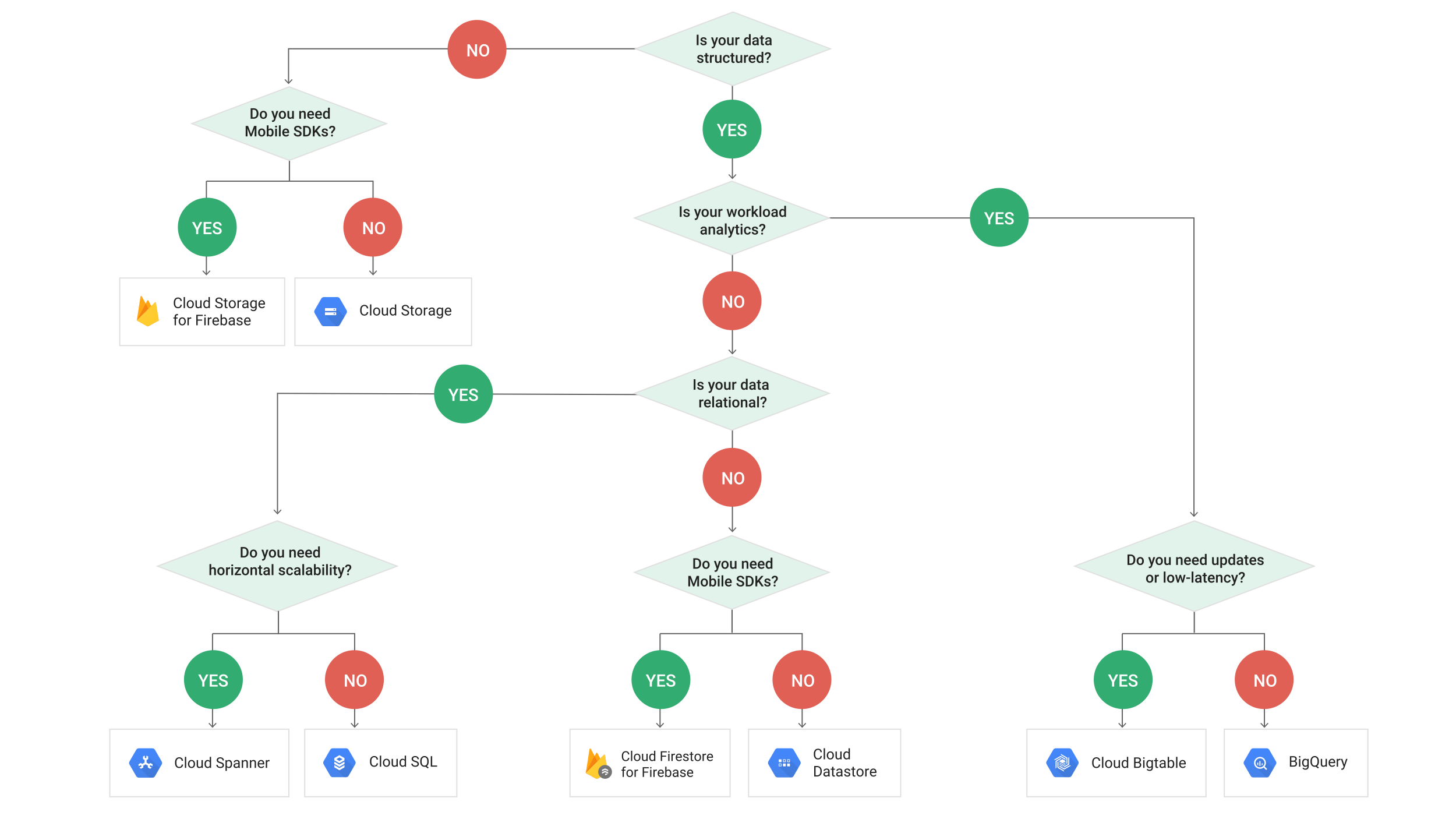
**Google Cloud Data Engineer Certificate Study Guide**

**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%)

**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
* **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    - Permanently delete an archived version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
  + Import it into DB using copy from csv or something similar.

**Cloud Spanner**

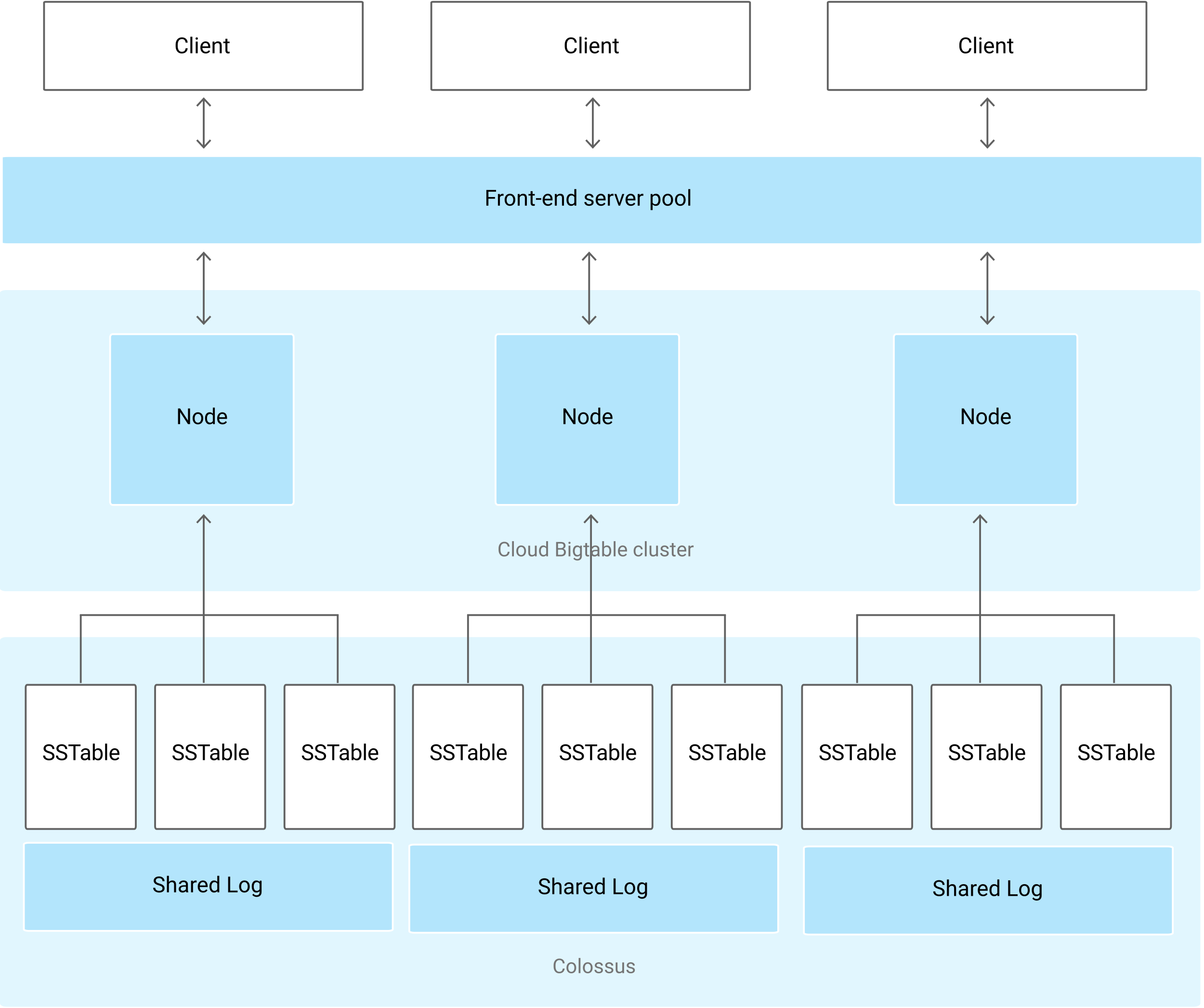
* Distributed and scalable solution for RDBMS (more expensive)
* Horizontal scaling: Add more machines
* Use when:
  + Need high availability
  + Strong consistency
  + Transactional support for reads and writes (especially writes)
* Don’t use when:
  + Data is not relational, or not even structured
  + Want an open source RDBMS
  + Strong consistency/availability is overkill
* **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.
  + 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.

**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.
* 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.
    - 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
  + Row => Entity
  + Tables => Kind
  + Fields => Property
  + Primary Key => Key
  + 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 of the same kind can have different properties.
  + 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)
  + Analytics/BI/Data warehousing (BQ instead)
  + If application has a lot of writes and updates on key columns.
* Use DataStore when:
  + Scaling of read performance – to virtually any size.
  + Use for hierarchical documents with KV data.
* 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
  + Eventually consistent
    - Faster, but might return stale data
* 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
  + Datastore Mode
    - Removes previous consistency limitations of Datastore.
    - Strongly consistent queries across the entire database.
    - Transactions can access any number of entity groups.
* **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
    - 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
* Columnar 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
* 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
  + 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.
* Avoid Hotspotting (Row keys to Use)
  + Field Promotion
    - 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
* 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.
* “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)
  + Don’t even think about HDD unless storing > 10 TB and 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
* Schema Design
  + Each table has just 1 index – row key
  + Rows sorted lexicographically by row key
  + All operations are atomic at row level
  + Related entities in adjacent rows
  + 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.
* 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
* BigTable instance comprise of Clusters and Nodes
* Tables belong to instances
  + If multiple clusters, you cannot assign a table to an individual cluster
* Production & Development
  + Prod:
    - Standard instance with 1-2 clusters
    - 3 or more nodes in each cluster
      * Use replication to provide high availability
  + Development:
    - Low cost instance with 1 node cluster
  + Create Compute Engine instance in same zone as Big Table instance
* **TOOLS**
  + **cbt**
    - Tool for doing basic interactions with BigTable
  + 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)
* Structure (1 Cluster below)

****

* + Data is never stored in BigTable nodes; each node has pointers to a set of tables stored on Colossus.
    - 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.
* 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
* **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
* **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\*)
* **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**
  + ‘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>
* **Access Control**
  + Security can be applied at **project and dataset level**, but not at table or view level.
  + **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.
* **Billing**
  + 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
* **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.
  + 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**
  + Can be exported as JSON/CSV/Avro
  + 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
* **Query Plan Explanation**
  + In web UI, click on “Explanation”
  + Good for debugging complex queries not running as fast as needed/expected.
* **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
      * 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
* Filter, group, transform
* Loosely semantically equivalent to Spark
* Follows the **Flink Programming Model**
  + Data Source -> Transformations -> Data Sink
* What are you computing (**operations**)
* Where in event time (**windows**)
* When in processing time?
* How (**triggers**)
* 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
  + **PCollection**
    - 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
  + **Transforms**
    - Data processing 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.
* Requires a **Staging Location** where intermediate files may be stored.
* **IAM**
  + **Dataflow.developer**
    - Enables the developer interacting with the Dataflow job with data privacy.
  + **Dataflow.worker**
    - Enables **service account** to execute work units for a Dataflow pipeline in Compute Engine.
    - **Dataflow API** also needs to be enabled.
* **DataSource**
  + Cloud Dataflow connector for Bigtable
    - Allows using Bigtable in a Dataflow pipeline
  + Can read data from **multiple sources** and can kick off multiple cloud functions in parallel writing to multiple sinks in a distributed fashion.
* **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.
* **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.
* **Key Things** 
  + Constraints you may have.
  + Why you would use JSON or Java related to Pipelines

**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.
* **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.
    - 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
* **Connectors**
  + BQ/BigTable (copies data to GCS) /CloudStorage
* **Networking**
  + Tcp:8088 (Cluster Manager)
    - <Master Node IP>:8088
  + Tcp:50070 (Connect to HDFS name node)
    - <Master Node IP>:50070
* **Optional Components**
  + Anaconda, Druid, Hive WebHCat, Jupyter, Kerberos, Presto, Zeppelin, Zookeeper
* Billed by the second, with a minimum of 1 minute.
* **IAM**
  + Service accounts must have **Dataproc.worker** role
    - Need permissions to read/write to GCS and to write to GC Logging

**Datalab**

* Managed Jupyter notebooks
* Great for use with a dataproc fluster to write pyspark jobs
* 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)
* 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

**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
* **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)
    - Pull
      * Subscriber explicitly calls pull method which requests messages for delivery.
      * More efficient message deliver/consume mechanism
    - 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**

**Machine Learning**

* Precision: Fraction of relevant instances among the retrieved instances
  + i.e. how many predicted cats are actually cats
* Recall: Fraction of relevant instances that have been retrieved over the total amount of relevant instances.
  + i.e. number of predicted cats out of all cats

**ML APIs**

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

**ML Engine**

* Train and predict ML models
* Can use multiple ML platforms such as **TensorFlow, scikit-learn** and **XGBoost**
* **Training Cluster**
  + Training service allocates the resources for the machine types you specify.
  + **Replica**
    - The running job on a given node.
    - Each replica in the training cluster is given a single role or task in distributed training.
    - **Master**
      * Only 1
      * This task manages the others and reports status for the job as a whole.
      * Single process job => the sole replica is the master for the job
    - **Workers**
      * 1 or more
      * Do the work as designated in job configuration.
    - **Parameter Servers**
      * 1 or more replicas
      * Coordinate shared model state between the workers
    - **CUSTOM tier**
      * Allows you to specify the number of Workers and parameter servers.
* Online and Batch Prediction
  + **Online**
    - Optimized to **minimize the latency** of serving predictions.
    - Predictions returned in the response message.
    - Returns as soon as possible.
  + **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.
    - Asynchronous request.
* **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.

**TensorFlow**

**Hadoop**

**Security**