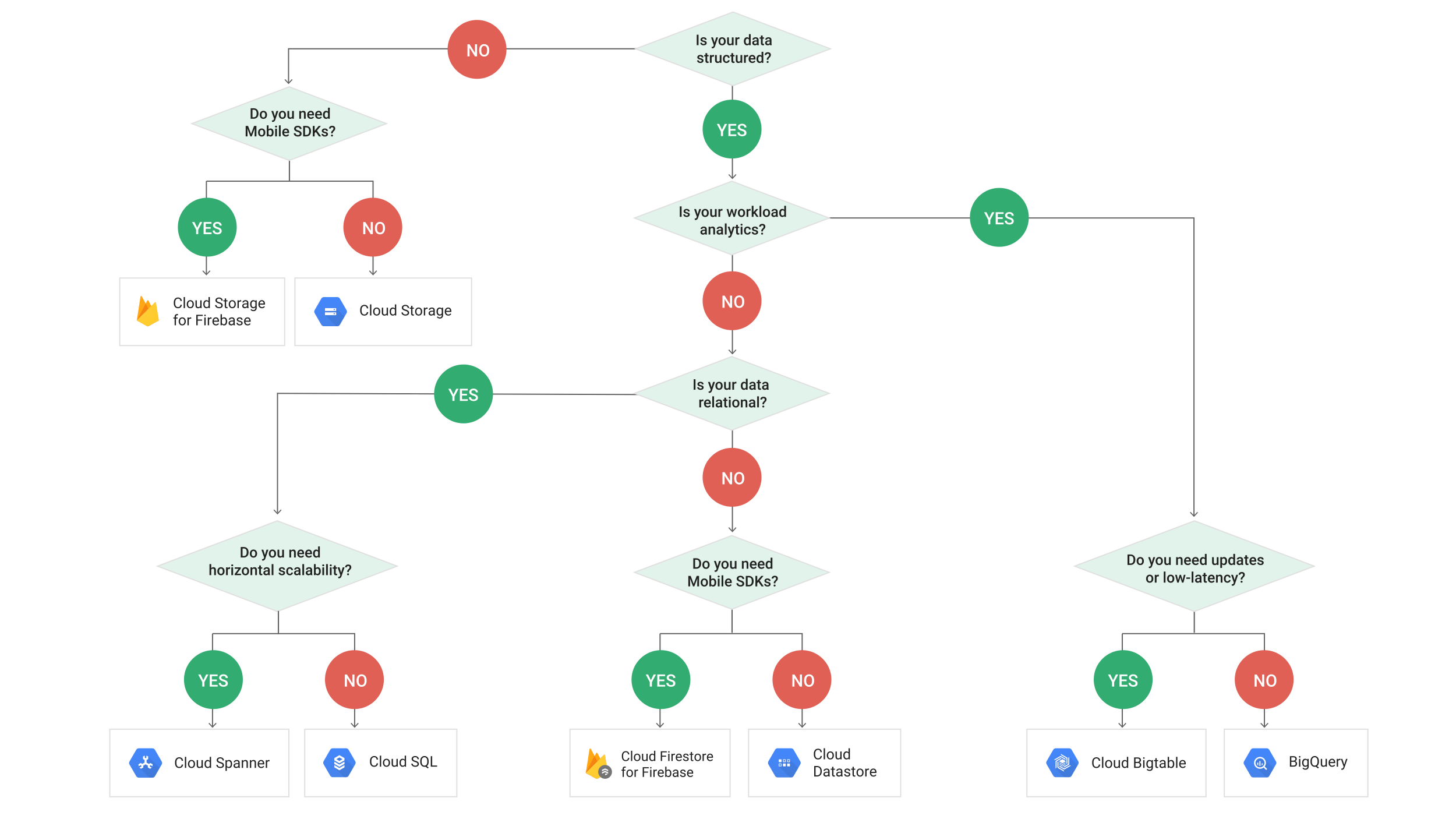
**Google Cloud Data Engineer Certificate Study Guide**

**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    - Permanently delete an archived version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**

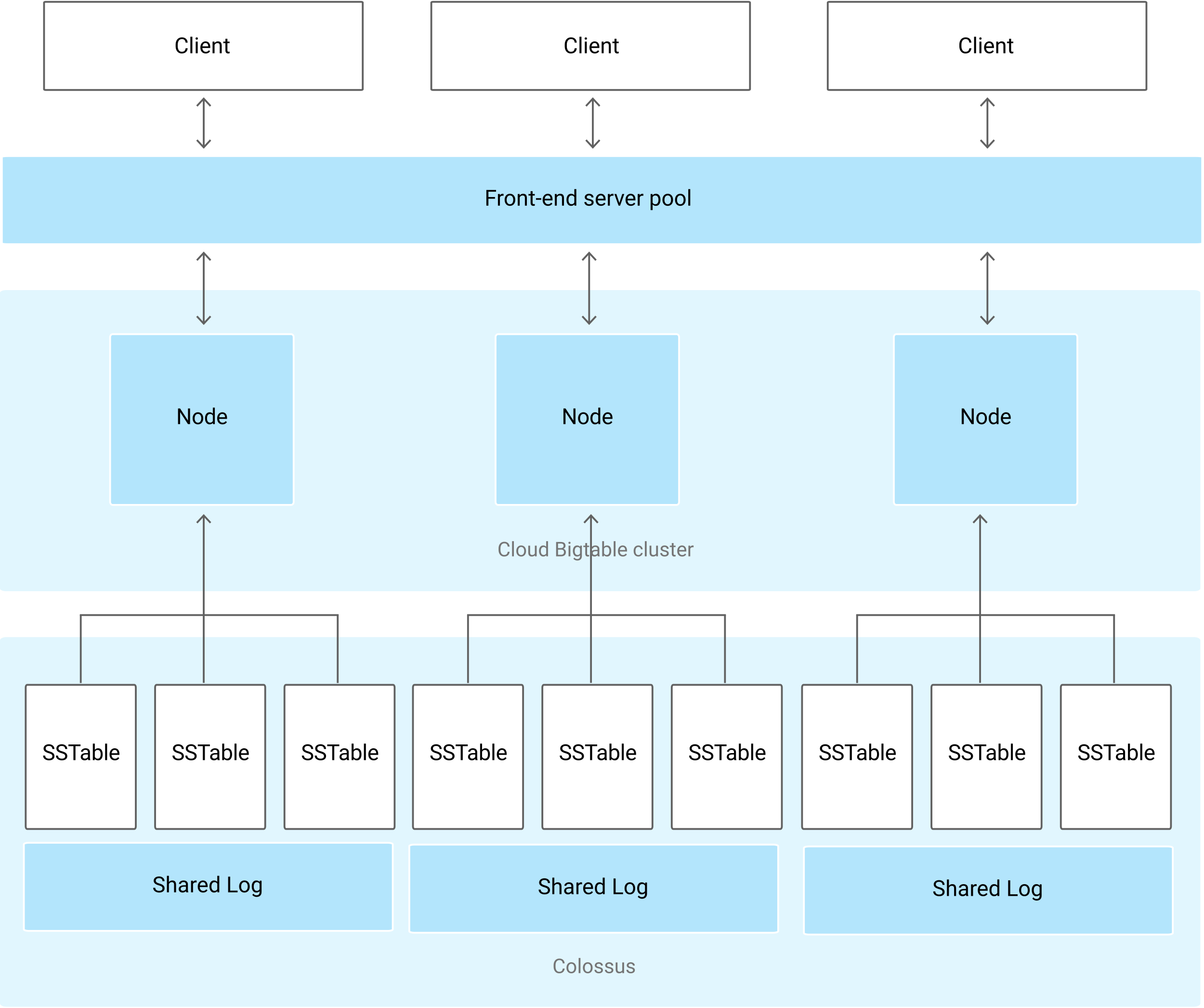
* Distributed and scalable solution for RDBMS (more expensive)
* Horizontal scaling: Add more machines
* Use when:
  + Need high availability
  + Strong consistency
  + Transactional support for reads and writes (especially writes)
* Don’t use when:
  + Data is not relational, or not even structured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

**Dataflow**

**Dataproc**

**Datalab**

**Pub/Sub**

**ML APIs**

**TensorFlow**

**Hadoop**

**Security**