

SQL vs. NoSQL Database: When to Use, How to Choose

 towardsdatascience.com/datastore-choices-sql-vs-nosql-database-ebec24d56106

Satish Chandra Gupta

September 3, 2021

Difference between SQL and NoSQL databases. Deep dive, decision tree, and cheatsheet to choose the best for your data type and use case from 12 database types.



How do you choose a database? Maybe, you assess whether the use case needs a Relational database. Depending on the answer, you pick your favorite SQL or NoSQL datastore, and make it work. It is a prudent tactic: a known devil is better than an unknown angel.

Picking the right datastore can simplify your application. A wrong choice can add friction. This article will help you expand your list of known devils with an in-depth overview of various datastores. It covers the following:

- that define a datastore's characteristics.
- categorized by : deep dive into databases for unstructured, structured (tabular), and semi-structured (NoSQL) data.
- between and databases.
- specialized for various .
- to navigate the landscape of on-prem and on-cloud datastore choices.

ToC:

Inside a Database

A high-level understanding of how databases work helps in evaluating alternatives. Databases have 5 components: interface, query processor, metadata, indexes, and storage:

1. **Interface Language or API:** Each database defines a language or API to interact with it. It covers definition, manipulation, query, and control of data and transactions.
2. **Query Processor:** The "CPU" of the database. Its job is to process incoming requests, perform needed actions, and return results.
3. **Storage:** The disk or memory where the data is stored.
4. **Indexes:** Data structures to quickly locate the queried data in the storage.
5. **Metadata:** Meta-information of data, storage. and indexes (e.g., catalog, schema, size).

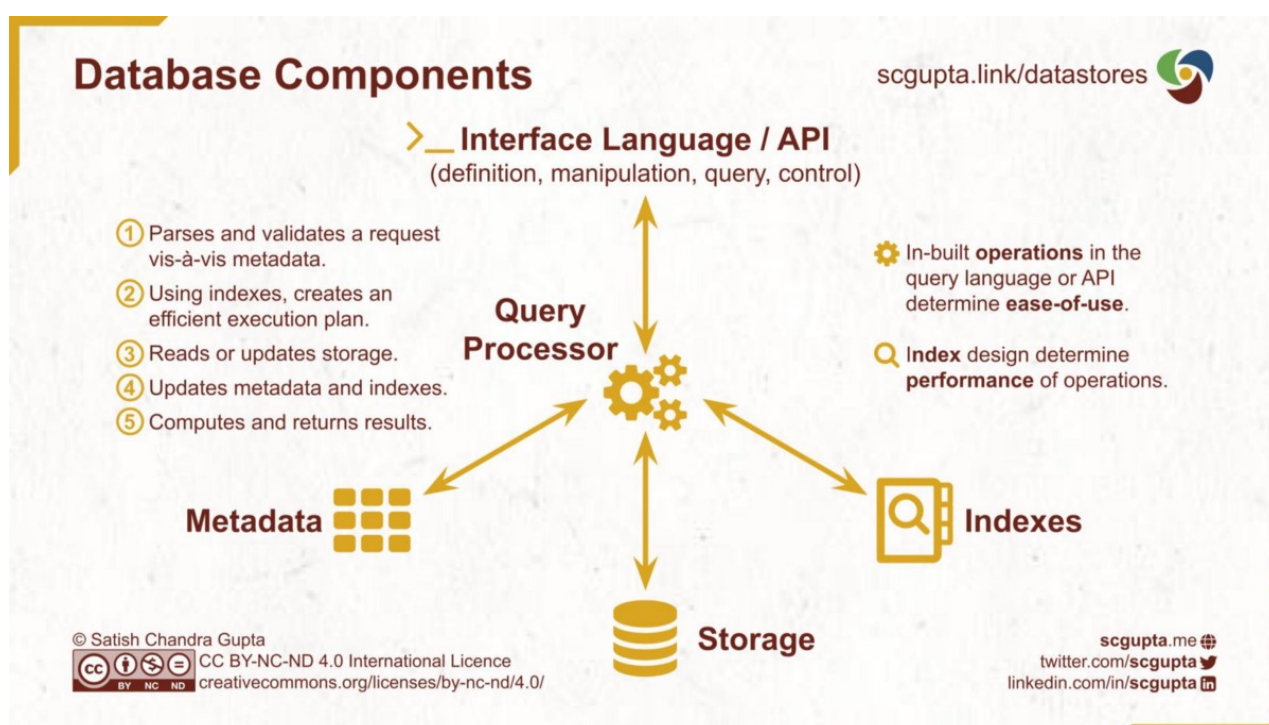
The **Query Processor** performs the following steps for each incoming request:

1. Parses the request and validates it against the metadata.
2. Creates an efficient execution plan that exploits the indexes.
3. Reads or updates the storage.
4. Updates metadata and indexes.
5. Computes and returns results.

To determine a datastore matches your application needs, you need carefully examine:

- **Operations** supported by the interface. If the computations you require are in-built, you will need to write less code.
- Available **indexes**. It will determine how fast your queries run.

In the next sections, let's examine operations and indexes in datastores for various data types.



Database Components: interface language, query processor, storage, indexes, and metadata; and the steps performed by the query processor. Image is by author and released under Creative Commons BY-NC-ND 4.0 International license.

Blob Storage for Unstructured Data

The file system is the simplest and oldest datastore. We use it every day to store all kinds of data. Blob Storage is a hyper-scale distributed version of the filesystem. It is used to store *unstructured data*.

Blob's backronym is Binary Large OBjects. You can store any kind of data. Therefore blob datastore has no role in interpreting the data:

- Blob supports CRUD (create, read, update, delete) **operations** at the file level.
- The directory or file *path* is the **index**.

So you can quickly locate and read the file. But locating something within a file requires a sequential scan. Documents, images, audio, and video files are stored in blobs.

Tabular Datastores for Structured Data

Tabular datastores are suitable for storing *structured data*. Each record (*row*) has the same number of attributes (*columns*) of the same type.

There are two kinds of applications:

- Online **Transaction** Processing (**OLTP**): Capture, store, and process data from transactions in real-time.
- Online **Analytical** Processing (**OLAP**): analyze aggregated historical data from OLTP applications.

OLTP applications need datastores that support *low latency* reads and writes of *individual* records. OLAP applications need datastores that support *high throughput* reads on a large number of (*read-only*) records.

OLTP: Relational or Row-Oriented Database

Relation Database Management Systems (RDBMS) are one of the earliest datastores. The data is organized in tables. Tables are normalized for reduced data redundancy and better data integrity.

Tables may have primary and foreign keys:

- **Primary Key** is a minimal set of attributes (columns) that uniquely identifies a record (row) in a table.
- **Foreign Key** establishes relationships between tables. It is a set of attributes in a table that refers to the primary key of another table.

Query and transactions are coded using Standard Query Language (SQL).

Relational databases are optimized for transaction operations. Transactions often update multiple records in multiple tables. Indexes are optimized for frequent low-latency writes of ACID Transactions:

- **Atomicity**: Any transaction that updates multiple rows is treated as a single *unit*. A successful transaction performs *all* updates. A failed transaction performs *none* of the updates, i.e., the database is left unchanged.
- **Consistency**: Every transaction brings the database from one valid state to another. It guarantees to maintain all database invariants and constraints.
- **Isolation**: Concurrent execution of multiple transactions leaves the database in the same state as if the transactions were executed sequentially.
- **Durability**: Committed transactions are permanent, and survive even a system crash.

There are plenty to choose from:

- Cloud Agnostic: Oracle, Microsoft SQL Server, IBM DB2, [PostgreSQL](#), and [MySQL](#)
- AWS: Hosted PostgreSQL and MySQL in [Relational Database Service \(RDS\)](#)
- Microsoft Azure: Hosted SQL Server as [Azure SQL Database](#)
- Google Cloud: Hosted PostgreSQL and MySQL in [Cloud SQL](#), and also horizontally scaling [Cloud Spanner](#)

OLAP: Columnar or Column-Oriented Database

While transactions are on rows (records), analytics properties are computed on columns (attributes). OLAP applications need an optimized column-read operation on a table.

One way to achieve it is by adding column-oriented indexes to Relational databases. For example:

- [Columnstore indexes in Microsoft SQL Server](#)
- [Columnstore indexing in PostgreSQL](#)

However, the primary RDBMS operation is low-latency high-frequency ACID transactions. That does not scale to the Big Data scale common in analytics applications.


For Big Data, storing in blob storage **Data Lakes** became popular. Partial analytics summarizations were computed and maintained in **OLAP Cubes**. Advances in the scale and performance of Columnar storage made OLAP Cubes obsolete. But the concepts are still relevant for designing the data pipelines.

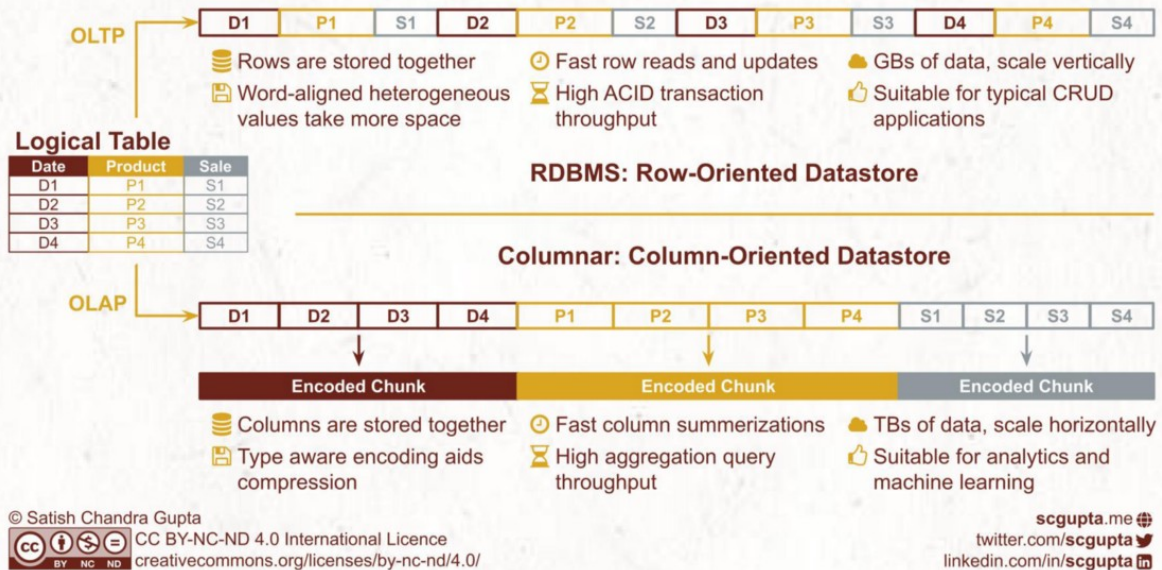
Modern **Data Warehouses** are built on **Columnar** databases. Data is stored by columns instead of by rows. Available choices are:

- AWS: [RedShift](#)
- Azure: [Synapse](#)
- Google Cloud: [BigQuery](#)
- Apache: [Druid](#), [Kudu](#), [Pinot](#)
- Others: [ClickHouse](#), [Snowflake](#)

[Databricks Delta Lake](#) offers columnar-like performance on data stored in data lakes.

RDBMS vs. Columnar: OLTP vs. OLAP

scgupta.link/datastores 



RDBMS vs. Columnar: Row-Oriented databases for OLTP and Column-Oriented databases for OLAP applications. Image is by author and released under [Creative Commons BY-NC-ND 4.0 International](https://creativecommons.org/licenses/by-nc-nd/4.0/) license.

NoSQL for Semi-structured Data

NoSQL datastores cater to semi-structured data types: key-value, wide column, document (tree), and graph.

Key-Value Datastore

A key-value store is a dictionary or hash table database. It is designed for CRUD operations with a unique key for each record:

- Create(key, value): Add a key-value pair to the datastore
- Read(key): Lookup the value associated with the key
- Update(key, value): Change the existing value for the key
- Delete(key): Delete the (key, value) record from the datastore

The values do not have a fixed schema and can be anything from primitive values to compound structures. Key-value stores are highly partitionable (thus scale horizontally). Redis is a popular key-value store.

Wide-column Datastore

A wide-column store has tables, rows, and columns. But the names of the columns and their types may be different for each row in the same table. Logically, It is a versioned sparse matrix with multi-dimensional mapping (row-value, column-value, timestamp). It is like a two-dimensional key-value store, with each cell value versioned with a timestamp.

Wide-column datastores are highly partitionable. It has a notion of column families that are stored together. The logical coordinates of a cell are: (Row Key, Column Name, Version). The physical lookup is as following: Region Dictionary \Rightarrow Column Family Directory \Rightarrow Row Key \Rightarrow Column Family Name \Rightarrow Column Qualifier \Rightarrow Version. So, wide-column stores are actually row-oriented databases.

Apache HBase was the first open-source wide-column datastore. Check out HBase in Practice, for core concepts of wide-column datastores.

Document Datastore

Document stores are for storing and retrieving a document consisting of nested objects. a tree structure such as XML, JSON, and YAML.

In a key-value store, the value is opaque. But the document stores exploit the tree structure of the value to offer richer operations. MongoDB is a popular example of a document store.

Graph Datastore

Graph databases are like document stores but are designed for graphs instead of document trees. For example, a graph database will suit to store and query a social connection network.

Neo4J is a prominent graph database. It is also common to use JanusGraph kind of index over a wide-column store.

SQL vs. NoSQL Database Comparision

Non-relational NoSQL datastores gained popularity for two reasons:

- RDBMS did not scale horizontally for Big Data
- Not all data fits into strict RDBMS schema

NoSQL datastores offer horizontal scale at various CAP Theorem tradeoffs. As per CAP Theorem, a distributed datastore can give at most 2 of the following 3 guarantees:

- **Consistency:** Every read receives the most recent write or an error.
- **Availability:** Every request gets a (non-error) response, regardless of the individual states of the nodes.
- **Partition tolerance:** The cluster does not fail despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

Note that the consistency definitions in CAP Theorem and ACID Transactions are different. ACID consistency is about data integrity (data is consistent w.r.t. relations and constraints after every transaction). CAP is about the state of all nodes being consistent with each other at any given time.

Only a few NoSQL datastores are ACID-complaint. Most NoSQL datastore support BASE model:

- **Basically Available:** Data is replicated on many storage systems and is available most of the time.
- **Soft-state:** Replicas are not consistent all the time; so the state may only be partially correct as it may not yet have converged.
- **Eventually consistent:** Data will become consistent at some point in the future, but no guarantee when.

Difference between SQL and NoSQL

Differences between RDBMS and NoSQL databases stem from their choices for:

- **Data Model:** RDBMS databases are used for normalized structured (tabular) data strictly adhering to a relational schema. NoSQL datastores are used for non-relational data, e.g. key-value, document tree, graph.
- **Transaction Guarantees:** All RDBMS databases support ACID transactions, but most NoSQL datastores offer BASE transactions.
- **CAP Tradeoffs:** RDBMS databases prioritize strong consistency over everything else. But NoSQL datastores typically prioritize availability and partition tolerance (horizontal scale) and offer only eventual consistency.

SQL vs. NoSQL Performance


RDBMS are designed for fast transactions updating multiple rows across tables with complex integrity constraints. SQL queries are expressive and declarative. You can focus on **what** a transaction should accomplish. RDBMS will figure out **how** to do it. It will optimize your query using relational algebra and find the best execution plan.


NoSQL datastores are designed for efficiently handling a lot more data than RDBMS. There are no relational constraints on the data, and it does not need to be even tabular. NoSQL offers performance at a higher scale by typically giving up strong consistency. Data access is mostly through REST APIs. NoSQL query languages (such as GraphQL) are not yet as mature as SQL in design and optimizations. So you need to take care of both *what* and *how* to do it efficiently.


RDBMS scale vertically. You need to upgrade hardware (more powerful CPU, higher storage capacity) to handle the increasing load.


NoSQL datastores scale horizontally. NoSQL is better in handling partitioned data, so you can scale by adding more machines.


SQL vs. NoSQL: Comparison


scgupta.link/datastores 

SQL		NoSQL	
	Relational	Model	Non-relational
Structured tables	Data		Semi-structured
Strict schema	Flexibility		Dynamic schema
ACID	Transactions		Mostly BASE, few ACID
Strong	Consistency		Eventual to Strong
Consistency prioritized	Availability		Basic Availability
Vertically by upgrading hardware	Scale		Horizontally by data partitioning

Key-Value
 Dictionary or Hash Table

Wide Column
 2-D Versioned Key-Value

Document
 Nested Objects (XML, JSON, YAML)

Graph
 Entity-Relationships

© Satish Chandra Gupta
CC BY-NC-ND 4.0 International Licence
creativecommons.org/licenses/by-nc-nd/4.0/

scgupta.me
twitter.com/scgupta
linkedin.com/in/scgupta

SQL vs. NoSQL: Difference between NoSQL and SQL databases. Image is by author and released under [Creative Commons BY-NC-ND 4.0 International](https://creativecommons.org/licenses/by-nc-nd/4.0/) license.

NoSQL Use Case Specializations

The line between various types of NoSQL datastore is blurry. On occasions, even the line between SQL and NoSQL is blurry ([PostgreSQL as a key-value store](#) and [PostgreSQL as JSON document DB](#)).

A datastore can be morphed to serve another similar data type by adding indexes and operations for that data type. Initial Columnar-like OLAP databases were RDBMS with column-store index. The same is happening to NoSQL stores for supporting multiple data types.

That's why it is better to think about the use case and pick the datastore suitable for your application. A datastore that serves multiple use cases may help reduce the overhead.

For analytics use cases, a tabular columnar database is often more suitable than a NoSQL database.

Datastores with in-built operations suitable for the use case are preferred (instead of implementing those operations in each application).

In-memory Key-Value Datastore

Same as a key-value store, but the data is in the memory instead of on the disk. It eliminates the disk IO overhead and serves as a fast cache.

Time Series Datastore

A time series is a series of data points, indexed and ordered by timestamp. The timestamp is the key in the Time Series datastores.

A time series can be modeled as:

- **Key-value:** associated pairs of timestamps and values
- **Wide column:** with the timestamp as the key for the table

A wide column store with date-time functions from the programming languages is often used as a time series database.

In analytics use cases, a columnar database can be used for time-series data as well.

Immutable Ledger Datastore

Immutable Ledger is for maintaining an immutable and (cryptographically) verifiable transaction log owned by a central trusted authority.

From the storage perspective, a wide column store suffices. But datastore operations must be **immutable** and **verifiable**. Very few datastores (e.g. [Amazon QLDB](#), [Azure SQL Ledger](#), and [Hyperledger Fabric](#)) fulfill those requirements at present.

Geospatial Datastore

A Geospatial database is a database to store geographic data (such as countries, cities, etc.). It is optimized for geospatial queries and geometric operations.

A wide column, key-value, document, or relational database with geospatial queries is commonly used for this purpose:

- [PostGIS](#) extension to PostgreSQL
- [GeoJSON](#) objects in MongoDB

In analytics use cases, a columnar database may suit better.

Text Search Datastore

Text search on unstructured (natural) or semi-structured text is a common operation in many applications. The text can either be plain or rich (e.g. PDF), stored in a document database, or stored in a blob store. [Elastic Search](#) has been a popular solution.

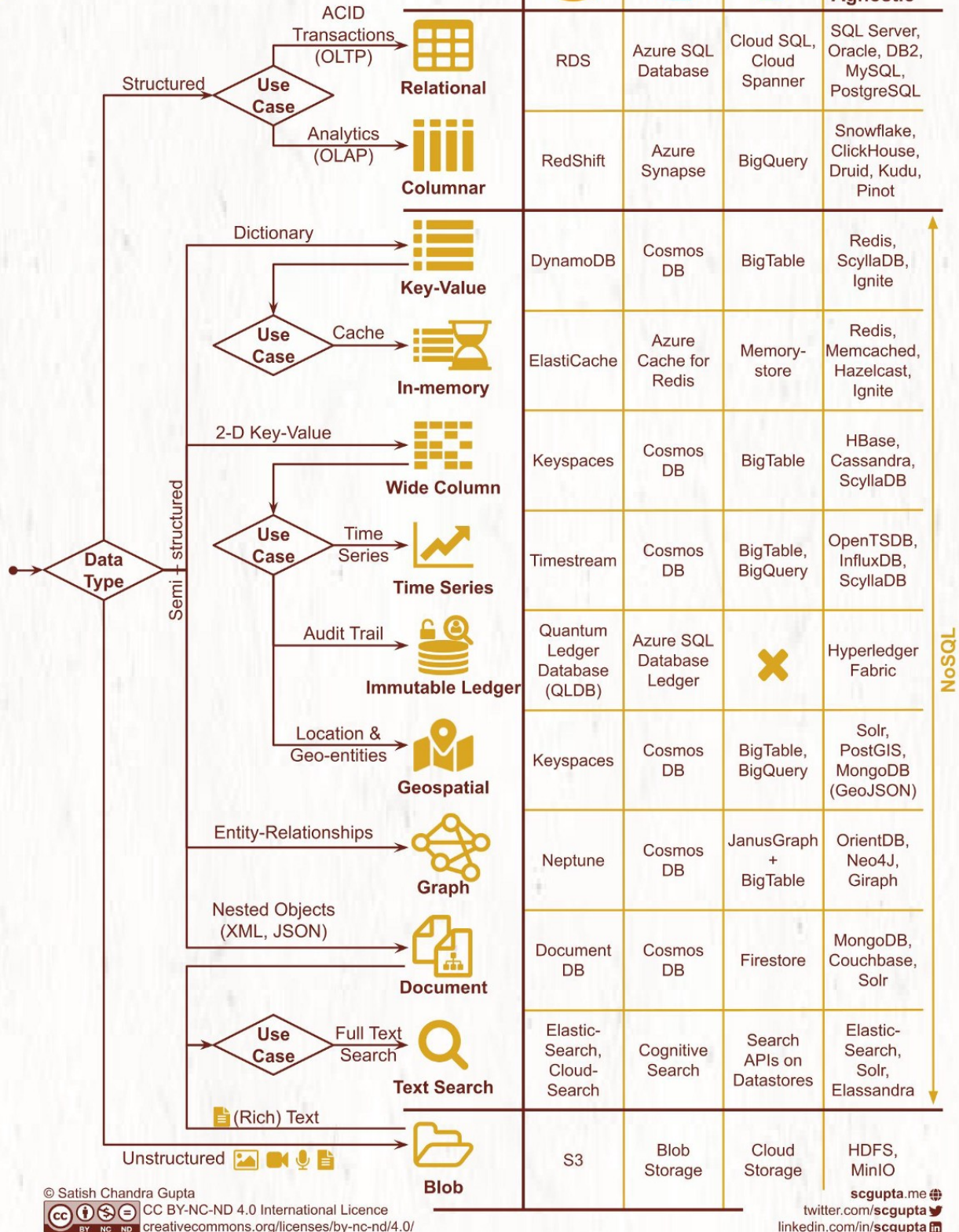
When to Use SQL vs. NoSQL: Decision Tree & Cloud Cheat Sheet

Given so many data types, uses cases, choices, application considerations, and cloud/on-prem constraints, it can be time-consuming to analyze all options. The cheat sheet below will help you quickly shortlist few candidates.

It is impractical to wait for learning everything needed to make a choice. This cheat sheet will get you few reasonable choices to start with. It is simplified by design, and some nuances and choices are absent. It is optimized for recall instead of precision.

SQL vs. NoSQL: Cheatsheet for AWS, Azure, and Google Cloud

scgupta.link/datastores



When to use SQL vs. NoSQL: Decision tree and cloud cheat sheet for database choices on AWS, Microsoft Azure, Google Cloud Platform, and cloud-agnostic/on-prem/open-source. Image is by author and released under [Creative Commons BY-NC-ND 4.0 International](https://creativecommons.org/licenses/by-nc-nd/4.0/) license.

Summary

This article walked you through various datastore choices and explained how to pick one based on:

- Application: transactions or analytics
- Data Type (SQL vs. NoSQL): Structured, Semi-structured, unstructured
- Use Case
- Deployment: major cloud provider, on-prem, vendor lock-in considerations

Resources

1. [Database Services on AWS](#)
2. AWS Whitepaper: [Overview of Amazon Web Services — Databases](#)
3. [How to choose the right database — AWS technical content series](#)
4. [Understanding Azure datastore models](#)
5. [Types of databases on Azure](#)
6. [Google Cloud database services](#)
7. [Data lifecycle and database choices on Google Cloud Platform](#)