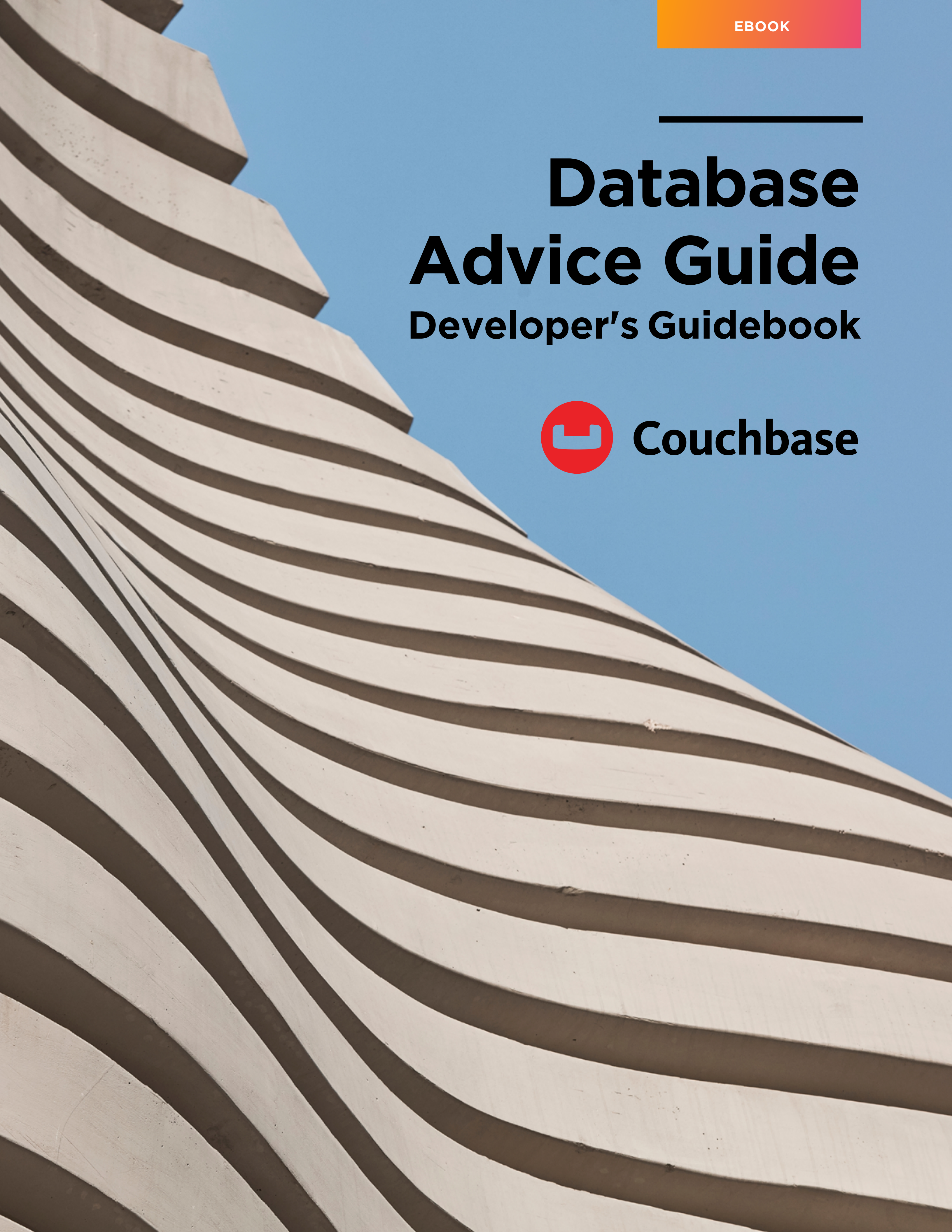
EBOOK

Database Advice Guide

Developer's Guidebook

Couchbase



Introduction

The database is one of the most important parts of a software system, whether for an application or a data warehouse project. As such, this document offers a virtual checklist that you can use when evaluating a database.

Don't underestimate the work that you need to spend in planning your data needs and scope. Picking a database is a long-term commitment. The database is the foundation of your application and provides secure, reliable storage and access to all of your information. And without trustworthy data, you don't have much of an application. This paper will focus on operational databases, those technologies designed as the data storage and data access support for application or microservices development. These databases have different attributes than pure analytical data warehouses, data marts, and data lakes, such as Snowflake or Databricks, which are used to aggregate data from a wider range of sources, requiring some extraction, transformation, and load (ETL) processes to bring data into the database. Additionally, the queries are often much more analysis-based than in an operational database.

This paper will cover:

- Areas to consider when choosing a database platform
- Setting up and configuring your database
- Ease of use
- Production performance, high availability, and scalability
- Security and data protection

Choosing a database platform

There are hundreds of databases and data-related platforms on the market. There are many ways to narrow down the scope and the following sections examine some of the most immediate choices.

The first question you will likely need to answer is, "Do I want to use a relational database?" And if so, what are the features I think I need in a relational database versus what I can have with a NoSQL database? Relational databases or relational database management systems (RDBMS) represent around 80% of the operational database market.

Relational databases

A RDBMS is structured on the relational model of data that organizes information into one or more tables of rows and columns, that are related to each other with a unique key for each row. Typically, different entity types (e.g., product or region) that are described in a database have their own table with the rows representing instances of that type of entity and the columns representing values attributed to that instance. Each row in a table has its own specific key and rows can be linked to rows in other tables by storing the unique key of the target row ("foreign key"). The linking of tables together allows for data to be set up in a way where data is not duplicated in the database, making storage very efficient. In order to get the best of a relational database, its schema (table and relational layout) is planned well in advance and tends to be difficult to change without significant changes to the application. This design rose to popularity from the 70s through the 90s when storage was very costly.

Nearly all RDBMSs use SQL (Structured Query Language) as the language for querying and updating the database. The SQL language had two big advantages over older read-write APIs. First, it gave rise to the idea of accessing many records with one single command, and second, it eliminated the need to specify how to reach a record. It is essentially a declarative language, but it also includes procedural elements. With the combination of organized table structure design and an easy-to-use query language, relational databases became quite popular due to their simplicity, robustness, transactional performance, and compatibility in managing data with other systems. For example, a fairly simple SQL SELECT statement could have many potential query execution paths. The RDBMS determines the best "execution plan" using features such as a cost-based optimizer to choose the correct indexes and paths.

The challenge for the relational database comes typically in two main areas: flexibility and scalability. As previously mentioned, the schema design of the database is often developed in the early stages of application development, with table design and key relationships intended to stay fairly static. Unfortunately, as the needs of the application change or as the desire to add new features evolves (often due to business changes) there is a need to redesign the schema. This often requires analysis to see the knock-on effects of the change and it may involve a database administrator (DBA) or other parts of an organization if they are using or contributing data to the same database. The other key challenge is scalability. Relational databases scale up well on a single machine but work less effectively when scaling out across multiple servers that can distribute the load. During attempts to scale to hundreds or thousands of servers, the complexities become overwhelming. The characteristics that make relational databases so appealing are the very same that also drastically reduce their viability as platforms for large distributed systems.

Relational Database Pros	Relational Database Cons
• Strong support for data integrity and transactions • Highly functional/popular query language including "joins" between tables • Powerful indexing capabilities, query planning, and cost-based optimization	• Rigid structure slows ongoing application evolution – less capable with semi-structured data • Scalability challenges and high costs • Overly efficient schema design can result in many tables and joins, which can impact read and write speeds

Examples of relational databases include: Amazon Aurora, IBM Db2, Microsoft SQL Server, MySQL, Oracle, and Postgres.

NoSQL databases

Non-relational databases, often referred to as NoSQL databases (short for "Not only SQL"), are designed to store and retrieve data that are not stored in relational table format. The most common types of NoSQL databases are key-value store, wide columnar, document, graph, and time-series.

- **Key-value store:** The simplest form of a NoSQL database, data elements are stored in key-value pairs that can be retrieved by using a unique key for each element. Values can be simple data types like strings/numbers or complex objects. Great speed can be achieved via key-value access.
- **Document databases:** Store data in JSON, BSON, or XML documents, in a form that is much closer to the data objects used in applications. This means less translation is required to use the data in the applications. Collections are a virtual grouping of documents used to help organize information. Documents provide great flexibility to change the database as the application evolves.
- **Graph databases:** Focusing on the relationship between the elements, data is stored in the form of nodes. Connections between nodes are called links or relationships. The goal is to be able to easily identify the relationship between the data by traversing the links.
- **Time-series databases:** Operate on data that is evaluated at regular intervals such as stock feeds or operating system activity logs. Here, the ability to zoom in and out at different granularity levels (from minutes to days, for example) may reveal a variety of trends from the data.
- **Wide columnar databases:** Stores the data in columns instead of rows, which is often helpful in running analytics across a small number of columns. This approach is both memory and processing efficient, even when large amounts of data are stored.

Overall, NoSQL databases are designed to give developers more flexibility in how they store and retrieve data. These newer databases are also architected to improve horizontal scaling by leveraging distributed architectures. Nodes can be added and changed without requiring changes to the application, with data automatically replicated to new nodes. This also often results in better uptime with less work required. These systems were designed for the world of big data and modern architectures like agile development, CI/CD, and serverless. Some NoSQL databases have proprietary query languages, while others have adopted SQL. Advanced databases offer sophisticated indexing technologies, ACID transaction support, and even cost-based optimizers. While most have been in development for far less time than relational databases, some NoSQL databases provide many of the same capabilities of a RDBMS.

NoSQL Database Pros	NoSQL Database Cons
• Flexible schemas/schemaless • High availability with less maintenance • Automated scaling • Versatile/specialized data modeling	• Potentially limited indexing (depends on type) • Less data integrity and consistency is possible • Less familiar to the industry • Data duplication

Examples of NoSQL databases include: Cassandra, Couchbase, Cosmos DB, DynamoDB, MongoDB™, Redis, and Neo4j.

Performance

Performance is another important area to consider when choosing your database. Almost all databases will work quickly at a small scale with limited numbers of nodes, data, and users. While you may not have full clarity on the expected growth of your application and database in the future, the more you know the better you can select the right database. Two typical measures of performance are throughput (operations per second) and latency, which measure the round trip time between a client and the database. For NoSQL databases, the industry standard for measuring performance is the Yahoo! Cloud Serving Benchmark (YCSB). A neutral third party, benchANT, publishes performance results from several products and vendors. The latest figures can be found on their site: https://benchant.com/ranking/database-ranking. Further information about performance is provided later in this paper.

Data access

Getting data out and into a database is critical. While SQL is the most common database language, it is not the only method to read and write data. Some databases are very limited in how one can access data, while others provide a variety of ways to get at the data. Key-value APIs use gets and puts for simpler and often faster data retrieval. Fuzzy search queries, also known as full-text search, can be a very useful database feature, providing search functionality for users within applications. Another option rising in popularity is GraphQL, which works with highly connected datasets. Depending on the combination of your present and future needs, ensure your database has your querying needs covered. It should also come with a software development kit (SDK) for different popular programming languages. Having many query options makes a database multi-model and lets developers do more in one database, which can reduce vendor and data sprawl. See this blog post on how the two are related.

Cloud strategy

As cloud services become more capable and help to reduce costs, moving your database to the cloud can likely offer flexibility, affordability, and scalability, but there are many ways to run in the cloud. Some organizations may prefer to self-manage in the cloud, giving them absolute control over their configuration and data, yet still getting the benefits of scalable infrastructure. Additionally, many look to automate self-management in some way using containers and Kubernetes. This makes it easier to administer a database once it is up and running and over time. For those looking for even less management, many database vendors now offer Database-as-a-Service (DBaaS) solutions. These let users provision clusters in a cloud provider of their choice and leave the majority of administration up to the database vendor. Alternatively, serverless databases further abstract the workings of the database and make it very simple to get data in and out. For all of these choices, a critical decision is the balance of control versus

Database-as-a-Service

A Database-as-a-Service typically runs on a cloud computing platform and is fully managed as-a-service by a database provider. Database services take care of replication, scalability, and high availability, making the underlying software more transparent to the user. Depending on the database you choose, it will either be easier or harder to build different functionality into the app and evolve it over time. Some databases provide very specific capabilities, while others are more broad and flexible. Cloud providers like AWS, Google Cloud, and Microsoft Azure offer a variety of DBaaS offerings, each focused on specific capabilities. Sometimes that selection can be more confusing than helpful. The good news is choice means more options for you to find a great DBaaS. Many cloud databases on the market offer common features that are all different in some way.

Mobile usage

Will your DBaaS need to sync data to a mobile or edge application? If so, you should consider the options for an embeddable database that can be synced with a central clustered cloud database. This brings the advantages of always-on apps regardless of web connectivity and speed. With a smart syncing and storage strategy, users will be able to quickly and easily search, update, and analyze data at the edge.

Analytics

As stated at the beginning, the focus of this paper is operational databases and not pure analytical databases, which doesn't mean that no analysis happens in an operational environment. Analysis is of course a very broad term. Analysis performed on the data tied to an operational dataset is often only used for one application. And by not having to move the data to a data warehouse, users can examine data in near-real time. The industry refers to this as hybrid transaction/analytical processing (HTAP) or hybrid operational and analytical processing (HOAP). It "breaks the wall" between transaction processing and analytics, and enables more informed and faster business decision-making. To help ensure that queries run quickly at scale, some databases accelerate queries by utilizing a massively parallel processing (MPP) engine.

Cost

Cost is always a factor when choosing a database. While most databases offer some sort of free tier or community edition, it is often not enough to support a production application. In planning for application deployment, you need to forecast your data volume needs, identify the services needed to meet your functional requirements, and understand the performance capabilities to meet your goals. Most vendors provide detailed pricing information on their websites. Conducting a proof of concept (PoC) is a good starting point to evaluate pricing. Understanding scalability also helps you identify the sizing needs for scale to determine the overall cost in production.

Setting up and configuring your database

Learning your database platform

Easy, quick setup

One of the keys to utilizing database management systems is a fast, simple setup. In today's dynamic work environments, users want data available on demand. The traditional steps of procuring hardware, installing software, testing that the setup was performed suitably, etc. are not feasible in the modern agile development life cycle. For many organizations the solution is to move to a DBaaS model.

Database-as-a-Service (DBaaS) is a managed service that lets users access database services without having to be concerned about managing infrastructure or software updates. It is the easiest and fastest way to quickly and effectively deploy a modern database. Basically, a fully managed cloud service eliminates your database management efforts. A database cluster can be deployed in just a few clicks. All that is required is to create an account and users are ready to quickly create clusters and databases, import data, and utilize the available database services of a fully managed DBaaS in a cloud environment.

Intuitive user interface

An intuitive interface works the way the user expects. Allowing users to navigate the UI instinctively while focusing on the tasks rather than struggling to navigate a complex interface. While you may want to script common tasks and/or use an API to manage your database, using a UI to explore and visualize data can be very helpful.

Familiar SQL syntax support

The most popular language for interacting with data is SQL. SQL is a language that has been around in relational databases since the 1970s, and has expanded to SQL++ for use with non-relational JSON document databases. Whether you're looking at relational or non-relational databases, robust SQL support means that many database and query skills will translate from database to database, and will help your team to ramp up on a new database faster without needing to hire experts in a proprietary query language.

Data modeling

Flexibility

Data flexibility is an important capability that allows teams to more quickly and efficiently respond to changing requirements. Document databases offer flexible JSON models with the schema enforced by the application instead of the database. Users can model data in a way that fits their application objects, nest documents, or even emulate RDBMS models. JSON provides a simple, lightweight, human-readable notation. It supports basic data types, such as numbers and strings; and complex types, such as embedded documents and arrays. JSON provides rapid serialization and deserialization and is a very popular REST API return data type.

Denormalizing RDBMS data
One of the many advantages of using a document-based database is the ability to use a flexible data model to store data without the constraints of a rigid, predetermined schema. When moving from an RDBMS data model to a NoSQL document model a common practice is to denormalize the data. Main RDBMS tables and auxiliary tables are often combined as nested parent and child JSON documents. However, it is also beneficial to consider how the data will be accessed since this greatly influences the data modeling strategy. For instance, smaller documents can be read/written faster. Please note that denormalizing data will increase the duplication of data and increase storage size. Storage costs have continued to decrease every year, but still should be a consideration.

Nested JSON documents
Nested JSON documents minimize the need for JOINs which makes data access faster and more efficient. It can also greatly simplify the data model. A typical RDBMS database architecture diagram has many more table objects than a NoSQL architecture diagram has corresponding collections. But it is important to note that with this approach, documents will be larger. If your model gravitates toward large documents, it's important to make use of sub-document APIs when possible (APIs that allow partial reading/writing of documents).

JOIN support
In the NoSQL document database world, SQL is uncommon, and JOINs are more so (but not completely absent anymore). Just as in a relational model, a JOIN clause can be used to create new objects by linking two or more source objects. Look for the following types of joins, and if they are supported:

- ANSI JOIN (joining between multiple documents on an arbitrary field)
- Lookup JOIN (joining between multiple documents based on their ID or document key joined with an arbitrary field)
- Index JOIN (the reverse of a lookup join, requiring an index)

When exploring a *distributed* document database, JOIN support across shards/partitions also needs to be considered, as they aren't always supported.

RDBMS concepts
Key concepts such as tables are widely understood. Many of these concepts can be mapped to a (roughly) equivalent feature in document databases.

RDBMS	Document DBMS
Schema	Scope
Table	Collection
Row	Document
Primary key	Document key
Index	Index

ACID transactions
Relational databases must support ACID transactions, due to the nature of their data modeling requirements. ACID transactions are required so that you are assured that the data you want to retrieve is what has been updated in a previous event and cascaded across tables throughout your database. Additionally, ACID guarantees are important for distributed activity and high availability requirements. JSON document databases have often initially skipped ACID transactions in favor of performance, but many of these NoSQL databases are now supporting ACID transactions. With the SQL++ query language, multi-document ACID transactions can be achieved and use the common syntax of BEGIN, COMMIT, and ROLLBACK. Again, there may be limitations based on sharding/partitioning, so if your application needs to perform transactions between shards, check to make sure what support is available (if any).

Ease of development

Familiarity, industry standards, and good documentation all help in ramping up development efforts to accelerate developer productivity.

CRUD
A common way to get familiar with a database and its SDK is by writing a CRUD application. CRUD stands for:

- Create
- Read
- Update
- Delete

These are key developer actions that also serve as very useful coding examples. The time it takes to get a CRUD example up and running can help to familiarize you with the basics, and give you a baseline to evaluate more complex operations.

DBaaS
A DBaaS solution means there is less maintenance required as compared to managing database software yourself. Users can sign up and start working without having to worry about software setup, installation, patches, or even system updates. This allows users to focus on utilizing the platform and not "yak shaving."

Supported access services
In a relational database, SQL is the only way to work with data. For NoSQL databases, there are many other services that could be available (including SQL++). Full-text search (browser-like), analytic queries, embedded mobile database syncing, database event triggers, key-value access, and in-memory cache performance enhancement are all examples of services that may be included in a database. You could take a "polyglot persistence" approach and use a different

database for each service, but this multiplies the maintenance and integration work (more "yak shaving"). Even if you don't plan to use all these services from the beginning, choosing a database that has these features ready to go can save you time, and provide a form of "future-proofing." Not to mention this approach can minimize "database sprawl" which can turn into an architectural nightmare.

Data access

Multi-model data access

A "multi-model" database is one that can support multiple access methods upon the same pool of data. The system provides unified data management, access and governance, among other key features. For example, Couchbase provides the following ways of interacting with data:

- **Key-value:** The ability to read/write data via a fast "key" lookup, given Couchbase's memory-first architecture, and is great for simple use cases and ultra-fast performance.
- **SQL++:** The world's most popular language for querying data; SQL syntax support is a database industry standard data access method.
- **Full-text search (FTS):** A text "search engine" for data that also supports geography-based searches, fuzzy search, faceting, etc.
- **Analytics:** Query data with complex, ad hoc SQL++ queries, in an isolated environment with multi-tenant support.
- **Eventing:** Developers can write JavaScript functions that respond to data change events.

These various data access methods help to minimize sprawl and include the efficiency benefit of users not having to learn multiple data management systems. This in turn reduces application development time and time to production.

SDKs, big data connectors, and related tools

Developers generally access data via specific language-based SDKs (or ODBC/JDBC connectors in some circumstances). While your team may be focused on one language, it's important to evaluate the available SDKs, should the need for other languages come up in the future. Here's a sample checklist of SDKs:

Server-Side SDKs	Mobile/Desktop SDKs
Java	Java (Android)
Scala	Java (Desktop)
.NET	Xamarin (.NET)
C/C++	MAUI (.NET)
Node.js	Swift
PHP	Objective-C
Python	C/C++ (embedded)
Ruby	
Go	
Kotlin	
Rust	
JDBC for Tableau	

Also check for available big data connectors and other tools for real-time analytics, streaming, data modeling, and search platforms. Some include:

- Kafka
- Spark
- Elasticsearch (ELK)
- CData
- Erwin
- Tableau

Development tools

Command line tools

CLI tools can be helpful for developers, DBAs, and DevOps alike. Some tools to look for include:

- Exporting data
- Importing data
- Creating backups
- REPL for SQL or SQL++
- Configuration and settings

While many of these tasks can be accomplished with a UI, having command line tools available is important and can provide efficiency and integration with other command line tools that you are already using.

Database IDEs/SQL editors

There are numerous editors or database IDEs available. You may already have licenses for them, or be familiar with them through your organization. Some popular choices include:

- ***JetBrains DataGrip*** – An IDE designed for SQL developers and database professionals. With it, you can write SQL in an intelligent query console with autocomplete, connect to multiple database servers/clusters, run queries, and much more. It's built on the IntelliJ IDEA platform, one of the top IDE platforms in the world.
- ***DbVisualizer*** – A popular tool utilized by developers and DBAs across platforms.
- ***SQL Server Management Studio (SSMS)*** – A tool primarily used by SQL Server Database developers but that can be connected to the Couchbase platform.

These tools may provide native support and/or generic ODBC/JDBC support.

JSON formatters/validators/parsers/helpers

JSON formatters and validators are helpful when working with JSON documents. Most of the IDEs listed above also feature JSON plugins.

There are also popular and easily accessible web-based JSON tools:

- JSON Parser
- JSON Lint
- JSON Editor Online

Production performance, high availability, and scalability

As enterprise projects move from development to production, the needs for stability and operational reliability become paramount. Developers needing these features of distributed systems look to NoSQL databases to deliver them. These enterprise production features provide the benefits of high availability, performance, and scalability.

High availability

High availability (HA) and disaster recovery (DR) are two of the driving reasons to move from a traditional RDBMS to a NoSQL-based system. High availability, as a concept, allows operations to guarantee certain levels of performance. This is not limited to system performance only, but also addresses recovery, failures, automated fixing of issues, backup, upgrades, and more.

NoSQL databases often support distributed database clustering, which helps deliver high availability. These databases also focus on the need for simple management and reducing interruptions to normal functioning. For example, operations can be done while the system remains online, without requiring modifications or interrupting running applications. All nodes do not need to be taken offline at the same time for routine maintenance such as software upgrades, index building, compaction, hardware refreshes, or other operations. Even provisioning new nodes or removing nodes can be done online without interruption to running applications, and without requiring developers to modify their applications.

Additionally, built-in fault tolerance mechanisms protect against downtime caused by arbitrary unplanned incidents, including server failures. Replication and failover are important mechanisms that increase system availability. For example, data can be replicated across multiple nodes to support failover scenarios. Ensuring that additional copies of the data are available is automated to deal with the inevitable failures that large distributed systems are designed to recover from. This functionality is provided automatically without the need for manual intervention or downtime.

To deliver increased high availability, disaster recovery, and geographic load balancing, entire clusters should also be replicated to one or more alternate geographical locations. Often this creates additional challenges in linking clusters that span across a wide-area network (WAN) rather than simply extending local-cluster replication, but the benefits are often extremely valuable to global products/companies.

For disaster recovery, customers can maintain a passive cluster (target) in addition to an active cluster (source) as part of their business continuity plan. This is extremely beneficial when the disasters occur at a data center or regional level. Backup and recovery processes are still essential components of disaster recovery and can also be distributed to reduce network overhead and related performance costs. The use of external blob or object storage services can help to store backup data close to the operational cluster for when it's needed.

Speed and performance

Clustered NoSQL environments can use commodity hardware, no supercomputer or massive scaled-up machines are necessary – this is by design. Anyone can have a superfast database if they spend millions on an extreme machine. But NoSQL users gain the benefits of scale-out – using many (virtual) machines to share the load – using normal or moderate server offerings or cloud instances.

The power of a cluster of nodes allows operations to be "smartly" distributed to handle workloads in an efficient manner. Additional nodes can be provisioned to help add specific capabilities to perform better (e.g., more data storage or increased memory allocations).

Distributed designs make performance at massive scale possible. As workloads increase, more nodes can be added without having to upgrade the existing nodes with more RAM, CPU, etc. Part of that distributed system design relies on a "shared nothing" architecture – each node manages its own resources including storage and processing. Additional nodes added for particular services, such as full-text search or big data analytics, can also use their own nodes without impacting performance of the others.

Replication and sharding are fundamental to automatically distributing data across nodes in a cluster. Thus, the database can grow horizontally to share load by adding more RAM, disk, and CPU capacity without increasing the burden on developers and administrators. This type of hardware efficiency is gained by employing asynchronous, non-blocking I/O for effective utilization of server resources. This helps increase both the storage I/O efficiency and the number of simultaneously connected clients per node.

NoSQL database architectures help workloads be evenly distributed across cluster nodes, to reduce bottlenecks and allow users to take full advantage of the available hardware. The end result is higher performance, better utilization, and improved TCO as over-provisioning can be avoided.

Scalability

There are several ways to scale a database platform. Scaling is not only about increasing RAM or storage, but also about adjusting specific services that are available. If a cluster has, for example, a JSON query service on one node but it is getting overutilized, then another similar node should be added or adjustment should be made to increase RAM and CPU settings. A system should allow users to optimize services alignment to hardware over time, as rarely does the needs of an application grow linearly for all of the data services that it is using. During these optimization changes, the team should not have to worry about how to replicate data or queries to those services. The system should just know that it has to adjust those services behind the scenes.

Clustered computing in itself is a very wide field of concepts and approaches that NoSQL databases adopt. Various cluster configurations are used depending on the need for performance versus high availability, etc. For example, NoSQL clusters can implement peer-to-peer architectures, support active-active configurations, flexible topology including multi-master bidirectional ring topology, simplified administration, and filtered replication (where only certain data is replicated).

Decentralization of resources and parallelized streams (e.g., for replication) are essential to having a fully scalable platform. But scalable clusters are also reliant on having easy management when needing to expand the cluster. Keeping the cluster online reduces any downtime or service disruptions.

When adding more nodes to a cluster, the database needs to be able to duplicate data and services from one node before taking it online. NoSQL databases that cover this well, also use concepts such as rolling upgrades where individual nodes are updated without being taken offline. This can help when replacing a node, adding another node, scaling up a node with more resources, etc.

Security and data protection

Security is an ongoing battle between organizations and hackers. In considering a database and its implementation, the one big question is often whether the database will be self-managed or fully managed by the database provider. Both will require many security considerations, but for Database-as-a-Service offerings, there are additional aspects that will need to be considered.

Core database security

Access control

Access control is the first core building block for securing data from a business and regulatory standpoint. The most basic level of security is who is allowed to access the data (authentication), which data (authorization), and how are we sure it is only those users. This applies to both users and applications seeking to access data. Best practices lay out two concepts: separation of duties and least privileged access.

- **Authentication:** Determines who is attempting to access the data. Users must be clearly and strongly authenticated. While this will depend on an organization's own standards and capabilities, certificate-based authentication is commonly the strongest form vendors provide. Database companies often use different models like:
    - **Password-based:** Built-in password authentication for both users and applications, where password strength policies are set based on complexity of the password, lifecycle, updating of the password, etc. Credentials transmission should be encrypted with transport-level security and/or hashed.
    - **Certificate-based:** Certificates like X.509 provide an additional layer of security where the certificate authority validates identities and issues certificates.
    - **Third-party/external authentication:** This allows you to plug into authentication platforms that organizations already have in place based on technologies like LDAP or Active Directory layer.
- **Authorization:** Once a user has been authenticated, authorization determines what a user or application is allowed to do with data. Sophisticated databases employ role-based access control (RBAC) where users are mapped to roles that determine the actions they are authorized to perform. Commonly the roles between administrators/users and applications/data access are separated. Each user may have zero or more roles which can be as broad or as restrictive as required, from a full administrator having access to all administrative functions and data, to an analyst with read-only access to a limited dataset.

Securing data means of course protecting data within your database and beyond. For on-premises, self-managed solutions the entire security environment must be managed. There are many areas to consider, but a few key areas include:

- **Physical security:** Buildings, data centers, servers
- **Network security:** Firewalls, IP tables, WAN encryption
- **Operating system:** User management, security patches, and updates
- **Application:** Credentials
- **Key management:** Rotation, revocation, remediation

Encryption

Another critical aspect of privacy and data protection is encryption. The goal is to make sure that sensitive information is not made available in the case of unauthorized access. To do so, unencrypted data (commonly referred to as plain text) is encrypted by using an algorithm and an encryption key. Doing so creates a ciphertext that can only be seen in its original form when decrypted with the correct encryption key. Encryption should be done for both data and metadata.

The following are important areas of data encryption:

- **Encryption at rest:** For data residing on physical media, encryption needs to protect against unauthorized access to the database files either from within the operating system or to the physical disks themselves. Often this is handled by the database working in conjunction with third-party software vendors which deny data access to anyone who does not possess an appropriate encryption key or is otherwise noncompliant with the configured security policy. Some examples of these vendors include Vormetric, Gemalto, Protegrity, and Amazon's encrypted EBS.
- **Encryption in transit:** Data often does not sit purely in a single database location. It is constantly on the move, being read, written, and replicated over networks. This movement requires an additional level of protection. Whether the data is moving between nodes in a single cluster within a data center, between data centers around the world, or out to mobile edge nodes, it is important to ensure that the database vendor provides robust encryption of data while in transit.
- **Data-level encryption:** This provides an extra layer of protection by encrypting user data within the database itself. Not only is it encrypted over the network and on disk, but requires a separate key from the application to decrypt.
- **Encrypted backups:** Backups contain a large volume of data and providers should provide options for encrypting this aspect of your data management.

Data retention, sovereignty, and auditing

In order to not just secure data, but to prove to others that data is being protected, various regulations have been enacted to set rules. Determining what data is retained, where, and for how long is becoming a critical function of data management. Often policies vary by continent, country, and even state/region. The handling of government data brings in other rules and regulations.

- **Retention:** This allows administrators to set policies about how long data is kept, in what manner, and the expiration processes. Retention rules can be applied to entire databases or data subsets.
- **Sovereignty:** Needing to filter out data subsets for different regions is a very common requirement, but not necessarily an easy one when working with large datasets on a global scale. Sophisticated replication technologies are built into some database platforms that allow fine-grained controls and automated replication based on those controls.
- **Auditing and reporting:** In alignment with regulatory needs and business needs, organizations need to be able to audit their data and report out the status of data, both current and historical.

Hosted data security

If choosing to utilize a Database-as-a-Service, ensure that your DBaaS vendor provides sophisticated, multilayer security technologies and 24/7 monitoring. Systems should include things like private networking and encryption while data is both at rest and in flight. Often DBaaS vendors will provide detailed whitepapers on their Trust Center webpages.

Here are some areas to think about as related to security for data hosted with a database vendor:

- **Governance, risk, and compliance:** It is important to understand how a vendor's Infosec team maintains information security policies, risk management processes, and compliance with regulatory and industry standards relevant to information security. Often organizations will publish detailed policy descriptions and controls.
- **Corporate operational security:** DBaaS providers should have established operational requirements to support the achievement of service commitments to customers, relevant laws, and regulations. Some of the important operational areas include:
  - ***Incident management:*** Having policies and procedures in place in case incidents occur
  - ***Vendor risk management:*** To minimize risk in working with outside vendors
  - ***Vulnerability management:*** Policies for protecting infrastructure
  - ***Endpoint security:*** For example, antivirus detection for all devices
  - ***Data classification and retention:*** For example, mandating specific protections for different information types
- **Corporate network protection:** Providers must ensure networks and entry points are protected against unauthorized access and have mechanisms in place to prevent efforts like distributed denial of service attack (DDoS) protection. A DevOps team should be involved to proactive monitor systems, networks, hardware, and software within the environment.
- **Compliances/regulations:** Your hosted data should be protected in many layers. Vendors have a variety of ways to demonstrate their commitment to that protection. One of these is through meeting compliance standards. There are many standards, most of which are industry-specific. The most common of which is SOC II compliance.

As every organization and application has different security requirements, vendors may be able to meet the requirements even if particular compliance is not complete. It is best to engage with a vendor regarding security to ensure all your needs are met.

Conclusion

There are many factors like speed, flexibility, time to market, and costs to consider when choosing a DBaaS to ensure it aligns with your application requirements. Choosing the right cloud database platform is not easy. To support important applications, many enterprises choose Couchbase to improve resiliency, performance, and stability, while reducing risk, data sprawl, and total cost of ownership. That's why 30% of the Fortune 100 manage critical data with the Couchbase database platform.

Getting started

Want to test drive Couchbase? Sign up to try Capella, our Database-as-a-Service, for a free trial. The service provides several sample datasets, data access tools, and tutorials to get you started. For those who are adventurous and want to jump right in, they can load their own data, connect an application, and start using features like SQL++ with various services such as Analytics, Full-Text Search, and Eventing. Mobile developers can explore the Couchbase mobile platform, with a robust embeddable database and sophisticated syncing technology.

Additional information

Here are several other ways to learn and try Couchbase:

- Couchbase Playground
- Tutorials and quickstarts
- Developer learning paths

Couchbase

About Couchbase

Couchbase provides an enterprise-class, multicloud to edge database that offers the robust capabilities required for business-critical applications on a highly scalable and available platform. Unlike other NoSQL databases, as a distributed cloud-native database, Couchbase Server runs in modern dynamic environments and on any cloud, either customer-managed or fully managed as-a-service. Couchbase is built on open standards, combining the best of NoSQL with the power and familiarity of SQL, to simplify the transition from mainframe and relational databases to distributed on-prem or cloud hosting environments.

Couchbase Capella™ is a Database-as-a-Service (DBaaS) that's fully hosted and managed by Couchbase to help reduce your database management burden and reduce your costs. Capella is the easiest and fastest way to begin with Couchbase NoSQL.

Couchbase has become pervasive in our everyday lives; our customers include industry leaders Amadeus, American Express, Carrefour, Cisco, Comcast/Sky, LinkedIn, Marriott, Tesco, Tommy Hilfiger, United Airlines, Verizon, as well as hundreds of other household names.

For more information, visit www.couchbase.com.