Canonical

How to choose a database management system

Hint: by asking the right questions

Executive Summary

Whether you are starting a new project or you are trying to solve an issue with your existing database management solution, you will eventually face the dilemma of choosing a database management system (DBMS). There are hundreds of available solutions and we can easily get lost in the maze of marketing materials and biased benchmarks.

Choosing the right database management system is often a critical decision for the success of your project. Once the product is actively used, changing the backing DBMS can be a gamble. According to Gartner, "more than 50% of data migration initiatives will exceed their budget and timeline—and potentially harm the business—because of flawed strategy and execution".[1]

Keeping an unfit DBMS is not a wise decision either as it might hinder the performance of your application, exponentially increase your costs when successfully scaling your user base or lead to costly outages.

In this guide, we help you make a considered decision on the right DBMS for your use case by going through the different criteria you need to review. We will go beyond the simplistic, and often misleading, classifications like SQL vs NoSQL to focus on the structural differences between DBMS solutions and their impacts from cost, performance, operability and convenience perspectives.

1 Make Data Migration Boring: 10 Steps to Ensure On-Time, High-Quality Delivery, Gartner, 2021

Contents

Executive summary 1
Table of contents 2
What criteria to consider when choosing your DBMS 3
- Human factors 3
- Cost factors 3
- Use case related factors 4

The right questions to answer 4
- Access patterns and used operators 4
- Availability guarantees 4
- Consistency and isolation guarantees 7
- Response times and throughput 7
  - Hardware characteristics 8
  - Data characteristics 9
    - Index related optimisations 10
  - Storage related optimisations 10
    - On-disk layout 10
    - Compression 10
    - Extent based space management 11
  - Database deployment models 11
  - Driver characteristics 12
- Scaling 13
  - Horizontal scaling 13
  - Vertical scaling 13
- Operability 14
  - Security 14
  - Compatibility/Integrability 14
  - Support 14

Run your own tests 15
- Conclusion 16

Further reading 16

What criteria to consider when choosing your DBMS

Choosing a database involves analysing different factors:

- Human factors
- Cost factors
- Use case related factors

Let's explore each of the factor types separately.

Human factors

Artificial intelligence is evolving quickly, but selecting and operating a DBMS remains the responsibility of humans. Therefore, we still need to care about:

- Getting **buy-in** from relevant stakeholders. First, we need to expose the rationale behind the introduction of the new DBMS. Then, we need to set clear KPIs to measure the success of the project (more satisfied customers, better performance or operability, talent attraction…). Finally, the execution plan should address the technical recommendations or concerns raised by the experts.
- **Knowledge** of the concerned team members about the involved technologies. This goes beyond the DBMS itself to cover all the tooling around it. Topics such as the testability of the set-up (in a dev laptop or in the CI/CD pipeline) and the quality of the available client drivers are as important as the DBMS server itself.

Therefore, ramping up your team is one of the areas you need to invest in as early as possible. That leads us directly to the next consideration – cost.

Cost factors

You need to factor several costs when introducing a new DBMS, including:

- As already mentioned, the **cost to strengthen or ramp-up your staff** on the DBMS technology.
- **Licence or subscription cost** to use the new software components or to get support for them.
- **Migration costs** to adapt your software stack and to migrate the data to the new solution.
- **Buying or upgrading hardware** to accommodate the new solution's needs.

Canonical helps you address all of the above points through our simple per-node Ubuntu Pro pricing and through our advisory services that provide expert insights during all the steps of your journey.

Caring about human and cost factors is essential when choosing a database. Yet, your use case and operability constraints should ultimately guide your choice.

Use case related factors

Your DBMS solution should ultimately help you achieve your business goal and it should fit into your operational constraints. Those goals and constraints are typically formalised through Service Level Agreements (SLAs) which you commit to your end users. SLAs can cover a wide variety of aspects like response time, data loss tolerance and uptime guarantees.

In the next section, we will go over most of the SLA related questions you need to answer when choosing your DBMS.

The right questions to answer

Let's lay out the questions to go through when choosing your DBMS. The questions are grouped in different categories and are ordered by priority.

Access patterns and used operators

Data is meant to be used, and how you use it should arguably be your most important criteria. Therefore, you need to answer the following:

- Do I always access my data with the same stable unique key?
- Do I need to express relations (e.g. joins, parent-child) between different data sets that cannot be merged into a single hierarchy?
- What kind of operations do I need to use on my data or its derivatives (equality, bigger/smaller, min/max, avg, count(*), pattern matching)?

You need then to check if the DBMS you are considering has built-in support for the access patterns you intend to use. For example, picking a pure key-value store when most of your queries involve joins is not optimal. Another example, would be picking a DBMS that does not support window functions while you often need to use them.

Now that we know we can access our data the way we want, we need to check whether the DBMS has the features to help us achieve the availability guarantees we are aiming for.

Availability guarantees

The availability of a given DBMS deployment is often expressed using the Recovery Time Objective (RTO) and the Recovery Point Objective (RPO) metrics. RTO measures service availability while RPO measures data availability. Therefore, our questions can be formulated as "what RTO and RPO values am I targeting in the event of one or several failures?" All hardware components will eventually fail so it is better to consider your target SLAs per failure event. For databases, you should at minimum consider:

- Local or remote storage failure
- Node failure (motherboard, cpus, power)
- Network failures (network partition, NIC failure)

All the components of a DBMS deployment contribute to the overall availability of the solution. For example, a well implemented database driver can contribute to a better perceived availability by implementing connection health checks and automatic retries. The following table list **some** of the important availability related features and how they contribute in honouring your availability claims:

Feature	Availability related benefits or impacts
Storage redundancy (RAID10, erasure coding …)	Sustain single drive failures.

Granular replication acknowledgement strategies	Gives you the flexibility to choose when a change is considered safely replicated by specifying:
	• The number of replicas acknowledging the reception of the change
	• The durability guarantee of the received change:
	  • Received in memory
	  • Persisted on disk
	  • Persisted and applied

Set maximum replication lag	Sets a maximum data loss target by ensuring that primary will **wait** for replicas to catch up when the replica lag exceeds a given threshold. You can think of it as a backpressure strategy.
Async replication	Provides a copy of your data that you can use as a fail-over target. Protects the primary copy from issues impacting the synchronously replicated instances at the expense of some potential data loss.
Synchronous replication	Provides an identical copy of your data that you can use as a fail-over target. Guarantees zero data loss if at least one of your synchronously replicated copies survives the failure event.
Delayed replication	Delays the propagation of logical corruptions (like a delete issued by mistake) to your copies.
Flashback	Allows you to rewind the database state to a previous point-in-time. Flashback can typically be performed at various levels: • Database level • Table level • Individual objects (e.g. rows)
Full backups	Store full copies of your data in another data store (typically an object store). So when a failure impacts your primary data store, you can still recover the data.
Incremental backups	Backup the delta between the current state and the most recent backup (full or incremental). This enables you to perform recovery with typically minute granularity while optimising the overall backup size (compared to performing full backups at the same frequency).

Feature	Availability related benefits or impacts
Backup on online instance	Keep the database instance usable when performing a backup. There are various usability levels. The database, while being backed up, can still be used: • As a failover target when primary is down • To serve read traffic • To serve write traffic
Consensus protocols	The cluster is able to automatically agree on a new topology in the event of a failure impacting a subset of its components. This is especially used to react to network failures.
Online changes of parameters	The more parameters you can change without restarting the database server, the less outages you will have.
Non blocking schema (a.k.a. DDL) changes	The more changes you can perform to your DBMS content without blocking concurrent user queries, the better perceived availability you will have from end user perspective.
Client or proxy based failovers	Makes server failovers transparent to connected clients.
Client/Server connection health checks	Prevents the application or the server from using a dead connection.
Client supplied timeouts	Prevents the application from hanging indefinitely – for example, when waiting for a lock release.
Client automatic retries	Overcome transient failures at the application layer.
Client-side circuit breaker	Implements fail-fast semantics to prevent propagation of failures to all your applications.

Generally speaking, the more availability features a DBMS supports, the easier it will be for you to achieve your availability targets. Yet, the relative importance of the features will depend on your use case and needs. Our Canonical experts can provide additional guidance through our advisory services.

Ensuring that your data is preserved against the adversity of failure events is one of the most desirable attributes of a DBMS deployment. Ensuring that the right data is returned to your clients is equally important. Let's explore why.

Consistency and isolation guarantees

In a simplistic world where all your clients are reading or modifying your data in a serial fashion (so one after the other), you do not need to care about consistency. Consistency and isolation concerns arise when you have multiple clients concurrently modifying and reading the same data sets. The ISO standard attempted to formalise some of those concerns. For example:

- **Dirty reads** happen when a client B reads in-flight changes (i.e. not yet committed) by another client A
- **Non repeatable reads** happen when client B reads the same object twice and the values returned differ because of the in-between committed modifications of client A.
- **Phantom reads** happen when client B reads a set of objects twice and their cardinalities differ because of the in-between committed modifications of client A.

Some use cases are more sensitive to the above issues than others. For example, the banking sector needs stronger consistency guarantees than the ones required by a social forum application. Allowing dirty reads in a system managing financial transactions would probably lead to the bankruptcy of the entity using the system.

There are dozens of consistency models to potentially know and care about. Yet, a pragmatic approach would be to stick to the SQL standard as most DBMS document their guarantees using the standard's four defined isolation levels:

- **READ UNcommitted** which is the lowest isolation level
- **READ COMMITTED** which prevents dirty reads
- **REPEATABLE READS** which prevents non repeatable reads
- **SERIALIZABLE** which prevents phantom reads

So the questions you might ask are the following:

- What consistency model is needed for my use cases ?
- What is my target consistency model ?
- Does my DBMS support my target consistency model ?

It is important to know that a DBMS can provide different consistency guarantees per type of operation (update of a single row, update of multiple rows …). So it is important to check the claimed consistency for the operations you intend to use.

You might wonder why you shouldn't simply aim for the strongest isolation and consistency model. It is because, according to the PACELEC theorem, distributed systems need to choose between latency and consistency.

So aiming for a given consistency might influence the latency of your transactions which in turn will influence the best achievable throughput and response times for your distributed database. Let's talk next about your target response times and target throughputs.

Response times and throughput

Having a better response time than competitors can be equivalent to having better market shares. This is especially true in e-commerce businesses like booking platforms.

The response time of a service depends on several factors including hardware and software, with hardware defining the limit. Let's start by looking at some hardware characteristics that are worth keeping in mind when thinking about response times.

Hardware characteristics

The below table summarises some of the fundamental latency ranges that you can expect from different hardware types and media. These ranges are a key consideration when targeting a given response time:

Hardware type	Media	Typical access latencies (in nanoseconds)
Volatile SRAM	CPU - L1 cache	1 ns
Volatile SRAM	CPU - L2 cache	4-10 ns
Volatile SRAM	CPU - L3 cache	10-20 ns
Volatile DRAM	Main memory (DRAM)	100 ns
Persistent flash	PCIe SSD	(10 us) 10 000 ns
Persistent flash	SAS SSD	(100 us) 100 000 ns
Persistent spinning disk	SAS HDD (15k rpm)	(2 ms) 2 000 000 ns
Persistent tape	SAS tape	(20 sec) 20 000 000 000 ns

The above table establishes a clear hierarchy between the different types of storage media. Yet, it only considers the access time (of the first fetched byte). The following table helps us understand the impact of the media type, interface and the I/O pattern on media performance, expressed in terms of throughput:

Protocol	Interface	Media type	I/O pattern	Throughput in MB/s
NVMe	PCIe 5	SSD	Random (4KB)	6 000
NVMe	PCIe 5	SSD	Sequential	14 000
NVMe	PCIe 4	SSD	Random (4KB)	4 000
NVMe	PCIe 4	SSD	Sequential	7 000
	SATA/SAS	HDD	Random (4KB)	3
	SATA/SAS	HDD	Sequential	200

The previous tables highlight several facts that are behind many of the optimisations implemented by mainstream DBMS:

- **Sequential access is faster than random access**. Thus, several DBMS (like PostgreSQL and MySQL) use Write Ahead Log to delay random IO in favour of sequential IO at commit time (and also to provide atomicity and durability). Many query optimisers will also favour sequential access over random access when choosing the optimal execution plan (check PostgreSQL's default values for random_page_cost and seq_page_cost for an example).
- **Flash storage is faster than spinning disks**. Hence, more and more DBMS are improving or redesigning their storage engines to benefit from modern storage solutions like NVMe SSDs. LSM Trees and RocksDB are examples of such efforts.
- **Volatile memory is faster than persistent memory**. This fact is, by far, the most impactful performance optimisation. Therefore all DBMS leverage memory as a:
  - Cache to accelerate their operations. Many DBMS will pick memory optimised structures when caching data (as with MySQL's adaptive hash index).
  - Staging area to compute changes before flushing them to persistent storage (instead of sending all intermediate results to disk).
  - Buffer changes before flushing them to persistent storage (check PostgreSQL's commit_delay for more practical details).
- **CPU caches are even faster than volatile memory**. CPU caches are managed solely by the CPU, so DBMS processes cannot access them directly. However, some DBMS try to favour cache locality by implementing a shard per core model (as with ScyllaDB).

So your target response time might significantly impact your choice of DBMS and the type of hardware you will allocate to it. For example, when targeting read times below the milliseconds range you need to ensure that your working set fits entirely in memory and that the chosen DBMS uses memory optimised structures. In such a case, choosing an in-memory DB or configuring your DB (check related settings in PostgreSQL, for an example) to act as one might be a pertinent choice when you can afford to lose some of your data.

In addition to choosing a DBMS that implements optimisations targeting the hardware you intend to use for your use case, it is also essential to use a DBMS that implements optimisations for the data types you need to manipulate and the associated access patterns.

Data characteristics

Your DBMS needs to have support (either built-in or through extensions) for the data types you need to manipulate. Data types can be categorised by the used exchange format (XML, JSON) or by their content (geo-spatial, graph, time-series, natural language).

You can store everything as data blobs and perform all decoding, sorting or analytical processing at the application layer. But this would lead to poor performance and poor developer experience, as it would be error prone and involve a lot of boilerplate code.

A DBMS with support for your data type can provide convenient functions to simplify your queries. Moreover, such a DBMS can implement various optimisations that are hard to mimic at the application layer. Those optimisations usually take the form of specialised index types or specialised storage formats that will significantly accelerate your queries.

Index related optimisations

The following table summarises the index types you need to look for based on the data you are manipulating and the operators you intend to use:

Index type	Targeted data types	Targeted patterns/operators
B+ tree	Generic or heterogeneous data. It should be your default choice.	Equality (=) Range comparison (>, <)
Bitmap	Low cardinality values (as with boolean or enumerations)	Analytical queries involving AND, OR or XOR.
Hash	Large or non naturally ordered data sets	Equality (specially when joining large data sets)
BRIN	Naturally sorted data sets (time series)	Equality (=) Range comparison (>, <)
R-Tree	Spatial data (geo-coordinates …)	Nearest neighbour, intersection, overlaps, contained in and more
[B]KD-Tree	Spatial data (geo-coordinates, geometric shapes)	Same as R-Tree
Inverted	Natural language (a.k.a. text search), arrays, semi-structured data (JSON, XML)	Inclusion of an item in a set
Bloom filter	Natural language (a.k.a. text search)	Inclusion of an item in a set Partial matching

Alongside indexing, another way to optimise lookup performance is to choose the right on-disk format for your specific use cases. Let's check some of the related optimisations.

Storage related optimisations

On-disk layout

All mainstream DBMS store data on-disk in pages. Every page is typically composed of a header and a payload. There are two main ideas on how to organise data within those payloads:

- Row oriented, where the payload contains a stack of rows.
- Column oriented, where the payload contains a stack of different values of the same column (but belonging to different rows).

Both layouts (or orientations) have pros and cons. Generally speaking, row-oriented storage engines will yield better performances for OLTP workloads, and column oriented will yield better results for analytical workloads. The on-disk layout can also influence the effectiveness of compression that you might want to enable for better performance.

Compression

Compressing the data on disk can yield better IO performances at the expense of some CPU overhead. The compression efficiency can be improved when your DBMS uses specific compression algorithms optimised for the used data types (check zson for an example).

Extent based space management

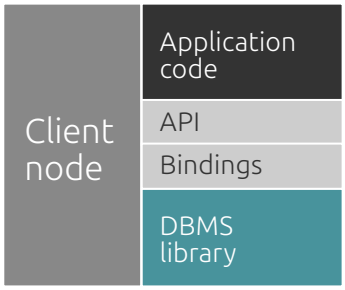
Some databases, like SQL Server, use contiguous pages (called extents) to manage on-disk space rather than individual pages (typically of 4KB size). Extents can significantly improve the performance of your queries when handling relatively large data sets.

We've covered the various database optimisations associated with the server side of your database (hardware and database engine). Next, we'll look at client side features. These features are often overlooked, but play a key role in DBMS performance (and reliability). Before diving into the client features let's clarify the client-server terminology.

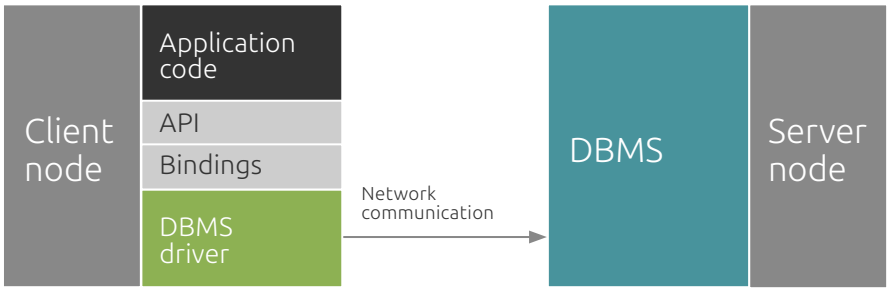
Database deployment models

When considering the deployment layout of a DBMS, we can distinguish the following categories:

Embedded database



Client-server database



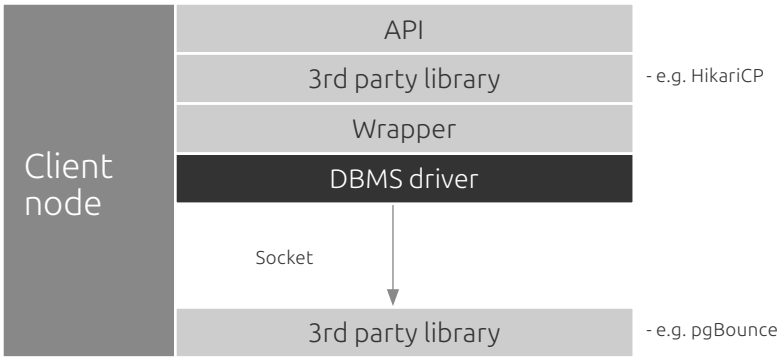
- **Client-server databases**, such as MySQL and PostgreSQL, have separate client and server units that you manage separately. The client components implement the network communication protocol with the database server. The client-server model tends to provide better scalability and easier centralised management.
- **Embedded databases**, such as Dqlite, are typically provided as a library that runs in the same node as your client application. The provided library contains all the functionalities needed to store and manage your data. Embedded databases tend to be lightweight and have better latencies.

The client features are more impactful in a client-server model than an embedded model. Let's check some of the client driver characteristics to look for when optimising the overall response time or throughput.

Driver characteristics

All DBMS provide one or more libraries that allow the applications to communicate with their database server. Those libraries can be written in the same language used by the application, in which case they are called native drivers. They can also just be wrappers over another driver (usually implemented in a low level language like C).

We can also complement those libraries with 3rd party tools (e.g. connection poolers, proxies) to provide additional features. The following diagram provides an overview on the possible combinations we might have:



When optimising for performance, it is better to reduce the number of abstractions by picking a native driver that has all the features you need. Here are the two most important features to look for from a performance perspective:

- **Connection pooling**. Acquiring a new connection to the database is often a costly operation as it involves several steps like creating a new network connection, spawning a dedicated server process or thread and establishing a secure (i.e. encrypted) communication channel. Therefore, it is a common best practice to maintain a small pool of cached connections and use them when an interaction with the database is needed.
- **Asynchronous (or reactive) programming**. This feature allows non-blocking calls to the database and might help achieve better throughput in highly concurrent workloads. Note that, for the effect to be visible, the driver itself should support non-blocking calls and not only provide an API that simulates such a behaviour.

We've explored the most important DBMS features to optimise the overall response time. It is also important to maintain the same level of responsiveness in the event of traffic peaks or changing workload patterns. For this, you need to be able to scale your DBMS.

Scaling

Scaling helps you accommodate growing workloads and can take several forms. We can scale:

- The number of DBMS deployments by adding new ones to typically accommodate new customers. Such scaling is useful when you are segregating your customers in different DBMS instances.
- Your operations to be able to manage additional deployments or growing usage of the existing ones. You can scale operations by ramping up your team or by using automation to help your existing staff manage a growing DBMS footprint.
- An existing DBMS deployment by allocating additional hardware resources to it. This might be the best solution to accommodate organic growth or seasonal peaks.

Canonical can help you address the first two points through our open source database operators. Our operators help you automate several DBMS operations within and across private and public clouds. So here we will rather focus on the DBMS features related to hardware scaling.

You can scale hardware resources vertically by adding more storage, cpu cores or ram to the same node where your DBMS is already running. You can also scale horizontally by allocating additional nodes to your DBMS. Let's check the features that can help you leverage both types of scaling:

Horizontal scaling

There are 2 main features that you need to check:

- **Read scaling** is a feature that allows you to perform your read queries not only on the primary but also on the replicas you might have.
- DBMS managed sharding is a feature that splits the data into different sets (called shards or slices) and then places those shards in different physical nodes. **Sharding** helps in scaling your write capacity.

Vertical scaling

There are different DBMS features that can help you scale your workloads by optimising existing resources usage or adding resources "in-place" to your existing infrastructure:

- **Partitioning** is similar to sharding but involves splitting the data into partitions that are placed on the same node. Using different partitions can help accelerate your queries by restricting the amount of data to be fetched to a single or few partitions.
- **Tablespaces** is a feature that helps organising data into different locations, which allows you to optimise the storage used per use case. For example, we can place index and wal files belonging to a particular schema into fast SSDs while placing the rest of the files in less performant storage. Thus, we can scale performance according to the workload.
- **Connection pooling** on server side. When you have a big number of clients (typically several hundreds) connected to your database, you might want to implement connection pooling at server side even when you have connection pooling at client side. Therefore, it is important to understand what connection poolers are available in your target DBMS ecosystem (built-in or provided by 3rd party).

Being able to scale your DBMS is one of the operability criteria to consider. Let's look at the other ones.

Operability

Your target total cost of ownership, availability, consistency, performance and scalability should all be part of the operability requirements you need to check against any technology you on-board in your team or organisation. Beyond these, further considerations include:

- Security
- Compatibility/Integrability
- Support

Security

We went through the common security controls we need to implement to secure a DBMS in this blog. To recap, here are the most important features you should check:

- Encryption on the wire
- Encryption at rest
- Grouping of DBMS permissions into roles
- Auditing
- LDAP/Kerberos integration

Compatibility/Integrability

We recommend checking how well your chosen DBMS can integrate with your existing ecosystem in terms of:

- **Identity and Access Management solution**, typically with existing LDAP/Kerberos infrastructure.
- **Programming languages**. It is important to choose a DBMS that supports the programming languages already used by your teams. It is even better when the DBMS has native drivers in your preferred programming languages.
- **CI/CD tools**. It is important to test your workflows involving a DBMS in your CI/CD pipeline. To do so, typically you need to use a containerised DBMS to ensure reproducibility and a lightweight footprint. Canonical provides a number of maintained container images that can fit your CI/CD needs.
- **Observability stack**. We need to monitor our DBMS fleet in a consistent manner. Again, our Canonical-supported operators ship with a standard observability stack that can help you easily achieve this target.

Support

Running a non supported combination of a DBMS and a set of 3rd party tools is a recipe for security and reliability issues. Therefore, we recommend picking a DBMS solution in which all the used components are supported and preferably validated together.

Through an Ubuntu Pro subscription, you can get support and security fixes for more than 25,000 open source packages for up to 10 years, including a large number of DBMS related components. Moreover, our charmed operators ship with a set of tightly integrated tools (e.g. DBMS server, proxy, monitoring tools) that are tested and released together.

Run your own tests

The next step is to put theory into practice and validate your choices with a series of tests and benchmarks in realistic conditions where you use production-like hardware and traffic. It is much better to run your own tests rather than just relying on public benchmarks.

Trying to figure out, for example, whether PostgreSQL is faster than MongoDB or the reverse will probably lead you to resources such as this Ongres benchmark where PostgreSQL is claimed to be up to 15x faster than MongoDB, or this MongoDB benchmark (that was written as a response to the first one) where MongoDB is claimed to be up to 240x than PostgreSQL. Trying to search for a 3rd party benchmark might lead you to this Arango benchmark where ArangoDB is shown to be faster in some scenarios.

There is no one size fits all, and the benchmarks may favour one scenario over another. So you need to run your own performance, availability and operability tests. Do not hesitate to contact our Canonical experts if you need help performing such tests or implementing a proof of concept of your new data stack.

Conclusion

This guide has covered a great deal of ground. To summarise, this is our recommended methodology for choosing a DBMS:

1. Document the KPIs you will use to measure the success of your journey (reduced cost, better performances, etc.).
2. Document the SLAs your DBMS needs to comply with in terms of availability, consistency, response time, scaling and operability.
3. Use this guide to rank available DBMS based on the features they offer in relation to your target SLAs. Use the ranking to keep the top 3 choices.
4. Iteratively run performance, availability and scaling tests on your chosen DBMS.
5. Create a report detailing the pros and cons of every tested DBMS and document the criteria used to choose the best option (e.g. using a weight system).
6. Run qualification tests on your chosen DBMS to ensure it integrates well in your ecosystem. Those tests need to emulate various flows including backup, restore and incident management.

Canonical can help you in all the steps of your journey so do not hesitate to contact us. We can speed up your decision making by sharing our expertise and insights from running hundreds of databases in all IT sectors.

Further reading

For more insights into database design, backup and migration you can read the following:

- https://ubuntu.com/engage/data-backups
- https://ubuntu.com/engage/database-cloud-migration
- https://ubuntu.com/blog/database-high-availability

Canonical