## Performance Metrics

- **memory usage** – this reading is critical, since running low on heap memory will cause your application to perform slower, and can even lead to *OutOfMemory* exceptions; on the other hand, using very little of the available memory could mean you could decrease your memory needs and therefore minimize costs
- **garbage collection** – since this is a resource-intensive process itself, you have to determine the right frequency at which it should be run, as well as if a sufficient amount of memory is freed up every time
- **thread usage** – if there are too many active threads at the same time, this can slow down the application, and even the whole server
- **request throughout** – this metric refers to the number of requests that the server can handle for a certain unit of time and can help determine your hardware needs
- **number of sessions** – a similar measure to the number of requests is the number of sessions that the server can support at a time
- **response time** – if your system takes too long to respond to requests, users are likely to quit so it's crucial to monitor this response time and investigate the potential causes
- **database connection pool** – monitoring this can help tune the number of connections in a pool that your application needs
- **error rates** – this metric is helpful in identifying any issues in your codebase
- **uptime** – this is a simple measure that shows how long your server has been running or down

## What to Do About Java Memory Leaks: Tools, Fixes, and More

Memory leaks often involve small amounts of memory resources, and you would probably not have problems with it. But when your application returns a java.lang.OutOfMemoryError, then your first and most likely suspect would be a memory leak.

Memory leaks are often an indicator of badly written programs, and if you are the type of programmer that wants everything to be perfect, then you should investigate any memory leak that occurs. As a Java programmer, there is no way for you to know when Java virtual machine would run the garbage collector. This is true, even if you specify System.gc(). The garbage collector will most probably run when memory runs low, or when the available memory is lesser than what your program needs. If the garbage collector does not free up enough memory resources, your program will get memory from your operating system.

## How to Avoid Java Memory Leaks

To avoid memory leaks, you need to pay attention to how you write your code. Here are specific methods to help you stamp out memory leaks.

### *1. Use reference objects to avoid memory leaks*

Using the java.lang.ref package, you can work with the garbage collector in your program. This allows you to avoid directly referencing objects, but use special reference objects that are easily cleared by the garbage collector. The special subclasses allow you to refer to objects indirectly. For instance, Reference has three subclasses: PhantomReference, SoftReference, and WeakReference.

- SoftReference object: Garbage collector is required to clear all SoftReference objects when memory runs low.
- WeakReference object: When garbage collector senses a weakly referenced object, all references to it are cleared and ultimately taken out of memory.
- PhantomReference object: Garbage collector would not be able to automatically clean up PhantomReference objects, you would need to clean it up manually by clearing all references to it.

WeakReference objects, especially with a cleanup thread, can help you avoid memory errors.

### *2. Avoid memory leaks related to a WebApp classloader*

If you are using Jetty 7.6.6. or higher, you can prevent WebApp classloader pinning. When your code keeps referring to a webapp classloader, memory leaks can easily happen. There are two types of leaks in this case: **daemon threads** and **static fields**.

- **Static fields** are started with the classloader's value. Even as Jetty stops deploying and then redeploys your webapp, the static reference persists and so the object cannot be cleared from memory.
- **Daemon threads** that are started outside the lifecycle of a Web application are prone to memory leaks because these threads have references to the classloader that started the threads.

### *3. Other specific steps*

- Release the session when it is no longer needed. Use the HttpSession.invalidate() to do this.
- Keep the time-out time low for each session.
- Store only the necessary data in your HttpSession.

- Avoid using string concatenation. Use StringBuffer's append() method because the string is an unchangeable object while string concatenation creates a lot of unnecessary objects. A large number of temporary objects will slow down performance.
- As much as possible, you **should not** create HttpSession in your jsp page. You can do this by using the page directive <%@page session="false"%>.
- If you are writing a query that is frequently executed, use PreparedStatement object rather than using Statement object. Why? PreparedStatement is precompiled while Statement is compiled every time your SQL statement is transmitted to the database.
- When using JDBC code, avoid using "*" when you write your query. Try to use the corresponding column name instead.
- If you are going to use stmt = con.prepareStatement(sql query) within a loop, then be sure to close it inside that particular loop.
- Be sure to close the Statement and ResultSet when you need to reuse these.
- Close the ResultSet, Connection, PreparedStatement, and Statement in the finally block.

**How do you know your program has a memory leak?** A very common telltale sign is the java.lang.OutOfMemoryError error. This error has several detail messages that would allow you to determine if there is a memory leak or not:

- Java heap space: Means that memory resources could not be allocated for a particular object in the Java heap. This can mean several things, including a memory leak, or the specified heap size is lower than what the application needs. It could also mean that your program is using a lot of finalizers.
- PermGen space: This means that the permanent generation area is already full. This area is where the method and class objects are stored. You can easily correct this by increasing the space via -XX:MaxPermSize.
- Requested array size exceeds VM limit: This means that the program is trying to assign an array that is > than the heap size.
- Request <size> bytes for <reason>. Out of swap space?: This means that an allocation using the local heap did not succeed, or the native heap is close to being all used up.
- <Reason> <stack trace> (Native method): This means that a native method was not allocated the required memory.

Now that you know your program has memory leaks, you can use these tools to help fix these leaks when they become a problem, or even before they become an issue.

*Using tools that can detect memory leaks*
*Using heap dumps*
*Using Eclipse memory leak warnings*

## Five Root-Cause Reasons Your Applications Are Slow

### #1. Sluggish client

**The problem:** Today's web-based applications tend to push user-interaction work — often accompanied by lots of data — to the client workstation. From there, JavaScript code processes hundreds or thousands of rows of data, which can cause multi-second pauses before the client display updates.
**The solution:** With a high-quality Application Performance Management (APM) system such as Riverbed SteelCentral™ AppResponse, you can easily identify clients with this type of internal-processing delay and differentiate application pauses from human "think time" delays.

### #2. Slow server

**The problem:** Server teams don't like to hear it, but the most common causes of slow application performance are the applications or servers themselves, not the network. Modern applications are typically deployed on a multi-tiered infrastructure:

- A front-end web server talks with an application server that talks with a middleware server that queries one or more database servers
- Then, all of those servers all might talk with DNS servers to look up IP addresses or map them back to server names

When that happens, just one weak link will slow the whole application down.
**The solution:** To identify the culprit, you must understand the interactions between multiple components in an application. This process, called Application Dependency Mapping (ADM), uses information from preexisting monitoring solutions as part of an integrated APM approach.
Fortunately, the network provides a perfect vantage point for ADM, meaning the network team can significantly help the application and server teams. Keep in mind, however, that using packet capture tools to discover whether the network or application is to blame could take many, many hours of work.
To save time, SteelCentral AppResponse lets you quickly and easily identify the cause of slow application performance. Once the proper monitoring points and basic configurations are set up, it provides immediate ROI and is a breeze to use. What's more, the information gathered provides Riverbed SteelCentral™ APM software with the input it needs to automatically draw dependency maps of critical applications.

**#3. Diminutive databases**

**The problem:** Applications developed on a fast LAN with a small data set seem to operate smoothly in the lab. But once rolled out into production, all bets are off. And, as the database continues to grow, so does your downtime.

**The solution:** In this case, a quick analysis with SteelCentral AppResponse might show that a key middleware server is making too many requests to a database server. (In fact, just one client request can result in many database requests or the transfer of a significant volume of data.) Simply making the database query more efficient typically solves the problem.

In another instance, a database server may take several seconds to return data to the middleware or application server. The application team can then use the SteelCentral AppResponse to identify the offending query.

**#4. Chatty conversations**

**The problem:** Another common cause of application slowness is chatty conversation: one application server, or perhaps the client itself, will make many small requests to execute a transaction on behalf of the person running the application.

However, with the advent of virtualization, the server team may have configured automatic migration of the server image to a lightly loaded host. This might move a server image to a location that puts it several milliseconds further away from other servers or from its disk storage system. And milliseconds can pile up fast.

**The solution:** To resolve this problem, you need visibility into the number of requests between systems, where the systems connect to the network, and into the delays between requests.

**#5. Slow network services**

**The problem:** Lastly, slow network services can slow application performance, which doesn't implicate the network itself, but the services that most network-based applications depend upon.

Consider an application that makes queries to a nonexistent primary DNS server. With no response, the application must time out the first request before attempting to query the second DNS server. In that situation the application periodically slows down, but it runs fine the rest of the time.

**The solution:** Intermittent problems like these are very challenging to diagnose. This is where SteelCentral AppResponse comes into play, as it watches and records all transactions all the time. Just identify the time of the slow performance and look for the root cause in the data.

## Why is My Database Application so Slow?

- **Network problems** – relating to the speed and capacity of the "pipe" connecting your SQL application client to the database
- **Slow processing times** – relating to the speed and efficiency with which requests are processed, at end side of the pipe.

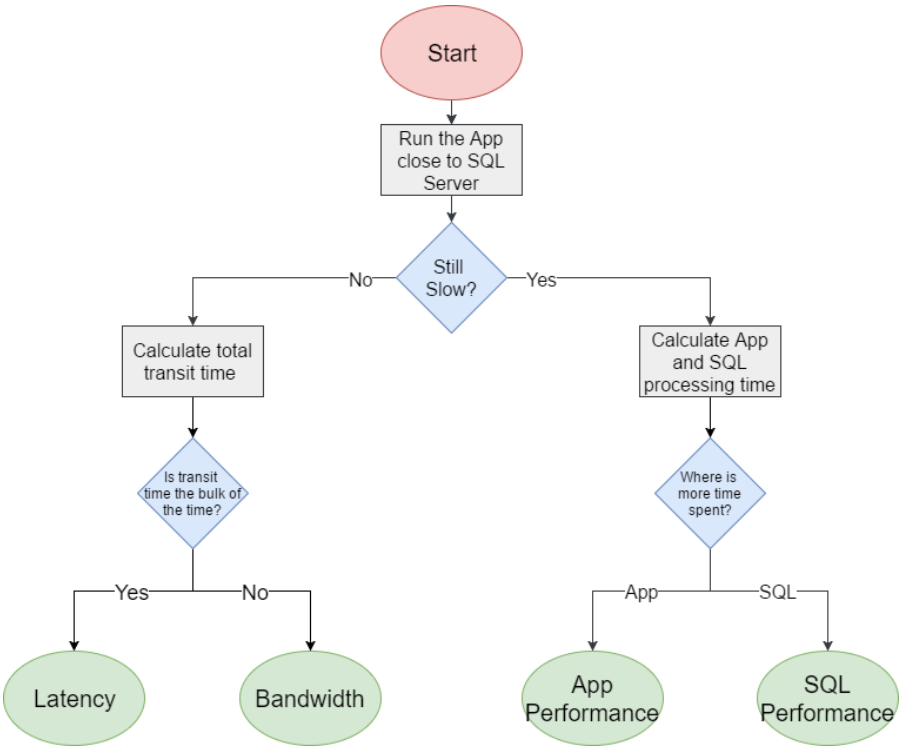
### Network problems

Latency
Latency is the time it takes to send a TCP packet between the app and the SQL Server.
Bandwidth
The amount of data that can be sent or received in an amount of time, normally measured in kb/s or Mb/s (megabits per second).

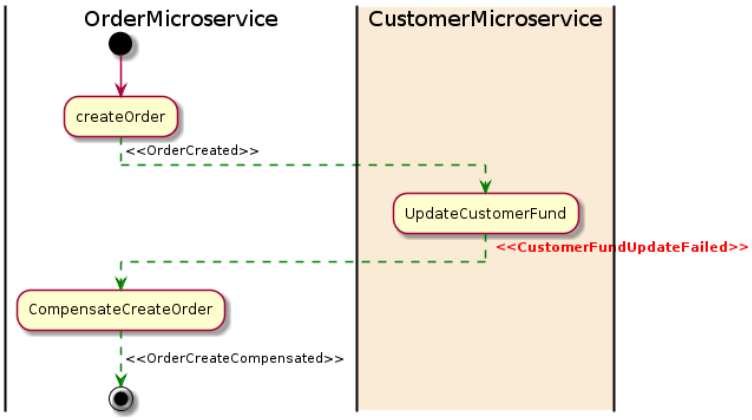
### Application problems: slow processing times

Whenever the client sends a request to SQL Server, to retrieve the required data set, the total processing time required to fulfill a request comprises both:

- **App processing time**: how long it takes for the app to process the data from the previous response, before sending the next request
- **SQL processing time**: how long SQL spends processing the request before sending the response



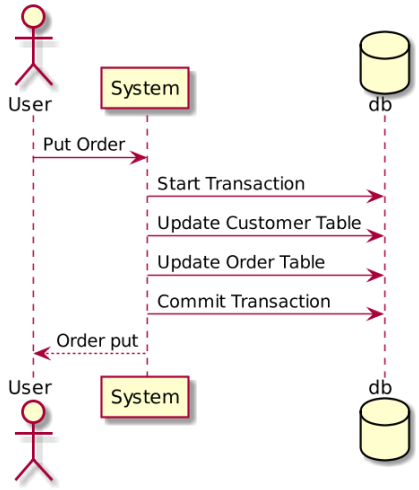

distributed transactions within a microservices architecture




What is a distributed transaction?

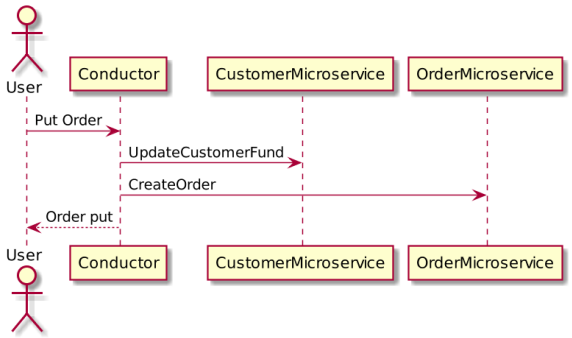
When a microservice architecture decomposes a monolithic system into self-encapsulated services, it can break transactions. This means a **local transaction** in the monolithic system is now **distributed** into multiple services that will be called in a sequence.

Here is a customer order example with a monolithic system using a local transaction:




In the customer order example above, if a user sends a **Put Order** action to a monolithic system, the system will create a  local database transaction that works over multiple database tables. If any step fails, the transaction can **roll back**. This is known as ACID (Atomicity, Consistency, Isolation, Durability), which is guaranteed by the database system.

When we decompose this system, we created both the CustomerMicroserviceand the OrderMicroservice**,** which have separate databases. Here is a customer order example with microservices:




When a **Put Order** request comes from the user, both microservices will be called to apply changes into their own database. Because the transaction is now across multiple databases, it is now considered a **distributed transaction**.

**What is the problem?**

In a monolithic system, we have a database system to ensure ACIDity. We now need to clarify the following key problems.

**How do we keep the transaction atomic?**

In a database system, atomicity means that in a transaction either **all steps complete** or **no steps complete.**

Possible solutions

The problems above are important for microservice-based systems. Otherwise, there is no way to tell if a transaction has completed successfully. The following two patterns can resolve the problem:

- 2pc (two-phase commit)
- Saga

**Two-phase commit (2pc) pattern**

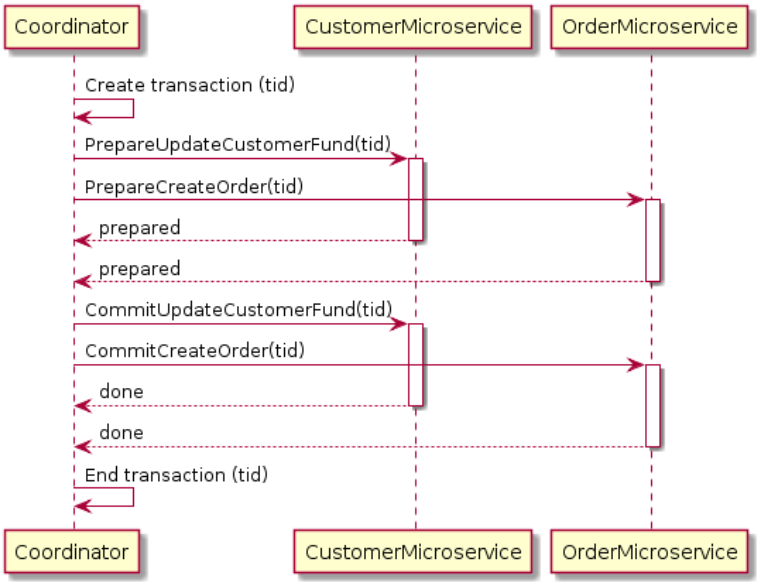
2pc is widely used in database systems. For some situations, you can use 2pc for microservices. Just be careful; not all situations suit 2pc and, in fact, 2pc is considered impractical within a microservice architecture (explained below).

**So what is a two-phase commit?**

As its name hints, 2pc has two phases: A *prepare phase* and a *commit phase*. In the prepare phase, all microservices will be asked to prepare for some data change that could be done atomically. Once all microservices are prepared, the commit phase will ask all the microservices to make the actual changes.

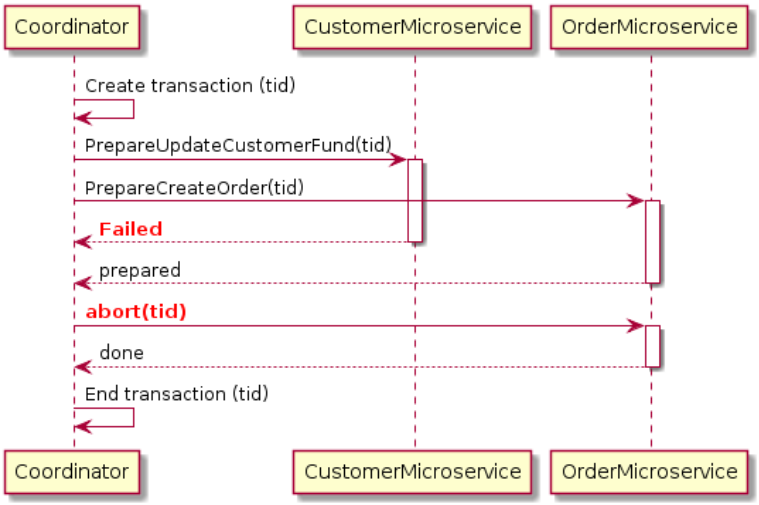
Normally, there needs to be a global coordinator to maintain the lifecycle of the transaction, and the coordinator will need to call the microservices in the prepare and commit phases.

Here is a 2pc implementation for the customer order example:




In the example above, when a user sends a put order request, the Coordinator will first create a global transaction with all the context information. It will then tell CustomerMicroservice to prepare for updating a customer fund with the created transaction. The CustomerMicroservice will then check, for example, if the customer has enough funds to proceed with the transaction. Once CustomerMicroservice is OK to perform the change, it will lock down the object from further changes and tell the Coordinator that it is prepared. The same thing happens while creating the order in the OrderMicroservice. Once the Coordinator has confirmed all microservices are ready to apply their changes, it will then ask them to apply their changes by requesting a commit with the transaction. At this point, all objects will be unlocked.

If at any point a single microservice fails to prepare, the Coordinator will abort the transaction and begin the rollback process. Here is a diagram of a 2pc rollback for the customer order example:




In the above example, the CustomerMicroservice failed to prepare for some reason, but the OrderMicroservice has replied that it is prepared to create the order.
The Coordinator will request an abort on the OrderMicroservice with the transaction and the OrderMicroservice will then roll back any changes made and unlock the database objects.

*Benefits of using 2pc*

2pc is a very strong consistency protocol. First, the prepare and commit phases guarantee that the transaction is atomic. The transaction will end with either all microservices returning successfully or all microservices have nothing changed.  Secondly, 2pc allows read-write isolation. This means the changes on a field are not visible until the coordinator commits the changes.

*Disadvantages of using 2pc*

While 2pc has solved the problem, it is not really recommended for many microservice-based systems because 2pc is synchronous (blocking). The protocol will need to lock the object that will be changed before the transaction completes. In the example above, if a customer places an order, the "fund" field will be locked for the customer. This prevents the customer from applying new orders. This makes sense because if a "prepared" object changed after it claims it is "prepared," then the commit phase could possibly not work.

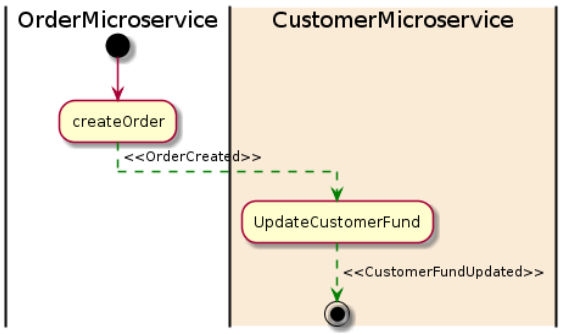
This is not good. In a database system, transactions tend to be fast—normally within 50 ms. However, microservices have long delays with RPC calls, especially when integrating with external services such as a payment service. The lock could become a system

performance bottleneck. Also, it is possible to have two transactions mutually lock each other (deadlock) when each transaction requests a lock on a resource the other requires.

### Saga pattern

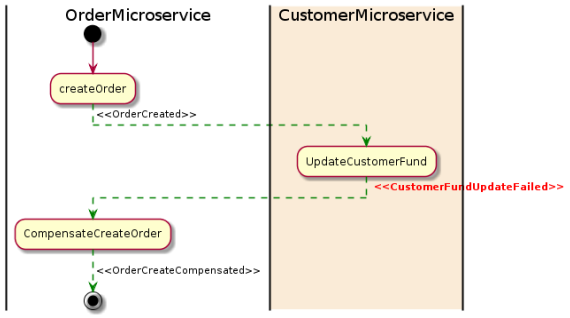
The Saga pattern is another widely used pattern for distributed transactions. It is different from 2pc, which is synchronous. The Saga pattern is asynchronous and reactive. In a Saga pattern, the distributed transaction is fulfilled by asynchronous local transactions on all related microservices. The microservices communicate with each other through an event bus.

Here is a diagram of the Saga pattern for the customer order example:




In the example above, the OrderMicroservice receives a request to place an order. It first starts a local transaction to create an order and then emits an OrderCreated event. The CustomerMicroservice listens for this event and updates a customer fund once the event is received. If a deduction is successfully made from a fund, a CustomerFundUpdated event will then be emitted, which in this example means the end of the transaction.

If any microservice fails to complete its local transaction, the other microservices will run compensation transactions to rollback the changes. Here is a diagram of the Saga pattern for a compensation transaction:



In the above example, the UpdateCustomerFund failed for some reason and it then emitted a CustomerFundUpdateFailed event. The OrderMicroservice listens for the event and start its compensation transaction to revert the order that was created.

*Advantages of the Saga pattern*

One big advantage of the Saga pattern is its support for long-lived transactions. Because each microservice focuses only on its own local atomic transaction, other microservices are not blocked if a microservice is running for a long time. This also allows transactions to continue waiting for user input. Also, because all local transactions are happening in parallel, there is no lock on any object.

*Disadvantages of the Saga pattern*

The Saga pattern is difficult to debug, especially when many microservices are involved. Also, the event messages could become difficult to maintain if the system gets complex. Another disadvantage of the Saga pattern is it does not have read isolation. For example, the customer could see the order being created, but in the next second, the order is removed due to a compensation transaction.

*Adding a process manager*

To address the complexity issue of the Saga pattern, it is quite normal to add a process manager as an orchestrator. The process manager is responsible for listening to events and triggering endpoints.

Conclusion

The Saga pattern is a preferable way of solving distributed transaction problems for a microservice-based architecture. However, it also introduces a new set of problems, such as how to atomically update the database and emit an event. Adoption of the Saga pattern requires a change in mindset for both development and testing. It could be a challenge for a team that is not familiar with this pattern. There are many variants that simplify its implementation. Therefore, it is important to choose the proper way to implement it for a project.

## ACID PROPERTIES

**Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either

before the execution of the transaction or after the execution/abortion/failure of the transaction.

- **Consistency** – The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

- **Durability** – The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

- **Isolation** – In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

## What Is Elasticsearch? (And Why You Need to Be Using It)

Elasticsearch is a database that stores, retrieves, and manages document-oriented and semi-structured data. Learn how you can use it to quickly resolve queries!

*Elasticsearch for faster queries.*

Products that involve e-commerce and search engines with huge databases are facing issues, including product information retrieval taking too long. This leads to poor user experience and in turn turns off potential customers.

Lag in search is attributed to the relational database used for the product design, where the data is scattered among multiple tables — and the successful retrieval of meaningful user information requires fetching the data from these tables.

The relational database works comparatively slowly when it comes to huge data and fetching search results through database queries. Understandably, businesses nowadays are looking for data storage alternatives in the hope of promoting quick retrieval.

This can be achieved by adopting NoSQL rather than RDBMS for storing data.

**Elasticsearch (ES) is one such NoSQL distributed database.**

Elasticsearch relies on flexible data models to build and update visitor profiles to meet the demanding workloads and low latency required for real-time engagement.

### What's So Significant About Elasticsearch?

ES is a document-oriented database designed to store, retrieve, and manage document-oriented or semi-structured data. When you use Elasticsearch, you store data in JSON document form. Then, you query them for retrieval.

It is schema-less, using some defaults to index the data unless you provide mapping as per your needs. Elasticsearch uses Lucene StandardAnalyzer for indexing for automatic type guessing and for high precision.

Every feature of Elasticsearch is exposed as a REST API:

1. **Index API**: Used to document the index.
2. **Get API**: Used to retrieve the document.
3. **Search API**: Used to submit your query and get a result.
4. **Put Mapping API**: Used to override default choices and define the mapping.

Elasticsearch has its own query domain-specific language in which you specify the query in JSON format. You can also nest other queries based on your needs. Real-world projects require search on different fields by applying some conditions, different weights, recent documents, values of some predefined fields, and so on.

All such complexity can be expressed through a single query. The query DSL is powerful and is designed to handle real-world query complexity through a single query. Elasticsearch APIs are directly related to Lucene and use the same name as Lucene operations. Query DSL also uses the Lucene TermQuery to execute it.