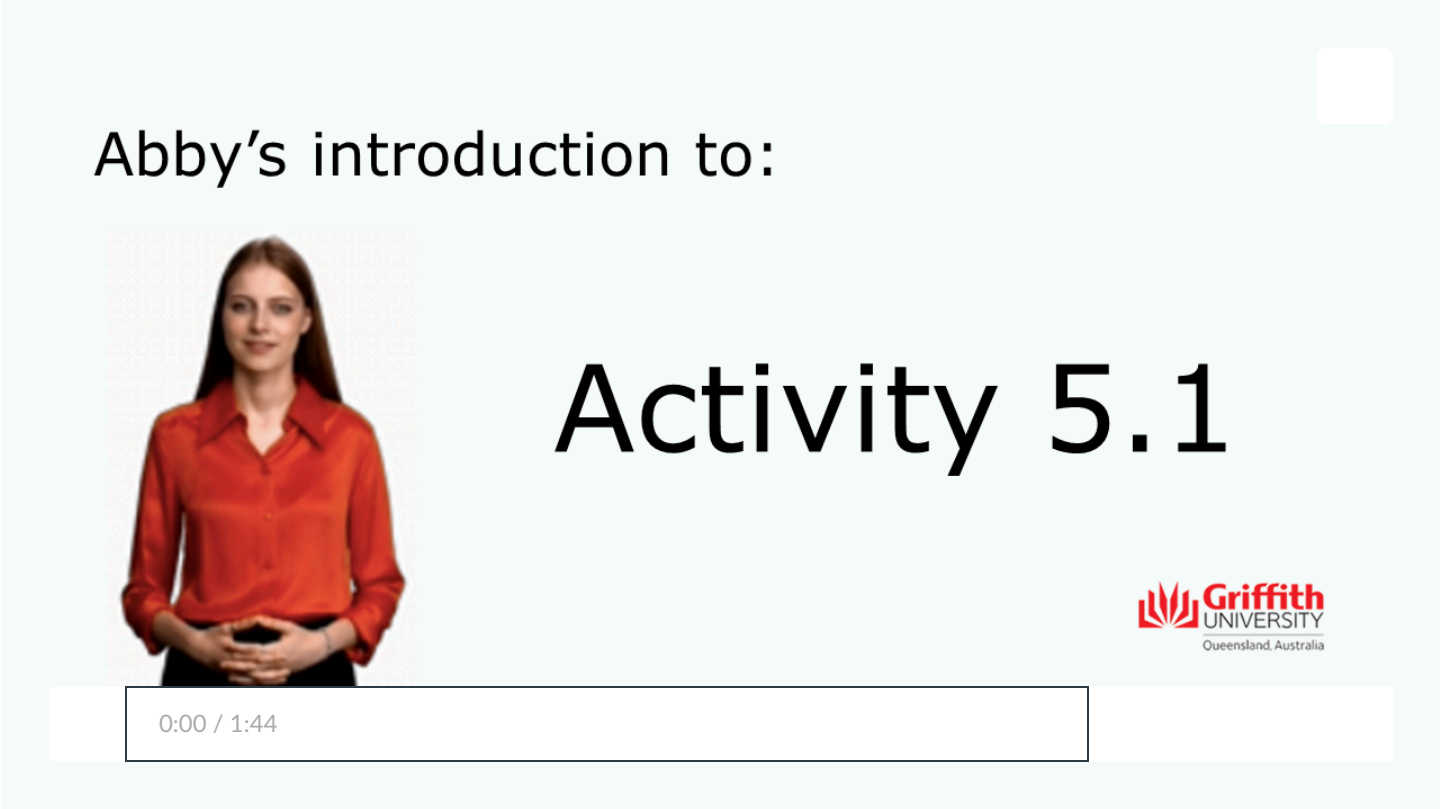
Activity 5.1 Identify performance bottlenecks and recommend evidence-based optimisation techniques for an application system scenario

Access course FAQ chatbot (https://lms.griffith.edu.au/courses/24045/pages/welcome-to-the-course-chatbot)

Module 5: Optimise performance, scalability, security, and privacy

---



What is this activity?

In Activity 5.1, you will identify performance bottlenecks and recommend evidence-based optimisation techniques for a complex application system scenario. This activity is designed to deepen your understanding of the factors that impact application system performance and scalability, and to develop your skills in analysing and optimising complex systems.

By examining real-world scenarios and proposing targeted improvements, you will gain practical experience in enhancing the efficiency and responsiveness of application systems.

Why is this activity important?

By engaging in this activity, you will learn to analyse complex systems, pinpoint areas of inefficiency, and propose evidence-based solutions to improve performance and scalability.

Some key benefits of this activity include:

Gaining a deep understanding of performance and scalability factors - Through analysing complex application system scenarios, you will develop a comprehensive understanding of the various factors that impact system performance and scalability, such as resource utilisation, data management, and algorithmic efficiency.

Developing practical skills in performance optimisation - By recommending evidence-based optimisation techniques, you will gain hands-on experience in applying best practices and strategies to improve application system performance and scalability.

Enhancing your problem-solving and analytical abilities - Identifying performance bottlenecks and proposing targeted improvements requires strong problem-solving and analytical skills, which you will develop and refine through this activity.

Preparing for real-world performance optimisation challenges - By working with complex, real-world scenarios, you will gain the skills and confidence needed to tackle performance optimisation challenges in your future career as an application system designer.

Case study

▶ Shopify - E-Commerce Application System

Supporting content for this activity

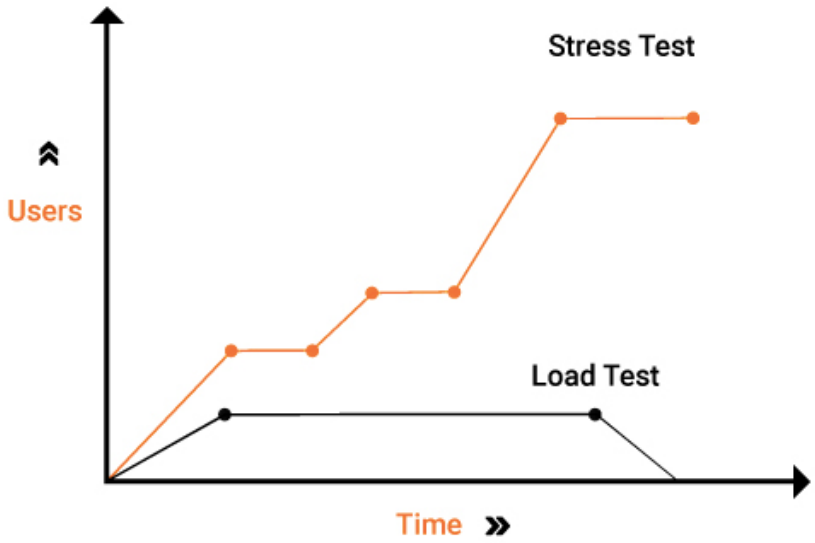
You should then work through the content elements below. These will reinforce the principles and elements from the shopify and will provide you with the knowledge and tools that you need to successfully complete this activity.

▼ Supporting content A - Understanding performance and scalability requirements

Techniques for analysing application system scenarios to identify performance and scalability goals

Analysing application system scenarios to identify performance and scalability goals involves a systematic approach that encompasses both qualitative and quantitative methods. One of the initial steps is to conduct a thorough **requirements analysis**, which includes gathering information on the expected user load, the types of operations to be performed, and the response times that are acceptable for the system. This analysis helps in setting clear and measurable performance goals, such as the number of transactions per second or the maximum latency tolerated for a specific operation. Additionally, it is crucial to understand the **scalability requirements**, which may involve determining how the system should behave under increased load and identifying the points at which additional resources need to be allocated to maintain performance levels.

To complement the requirements analysis, **performance monitoring and profiling** tools can provide invaluable insights into the runtime behaviour of the application. These tools can help identify bottlenecks by analysing resource usage patterns, such as CPU, memory, disk I/O, and network bandwidth. Profiling can reveal which components of the application are consuming the most resources and under what conditions, allowing for targeted optimisation efforts. For instance, if a particular database query is found to be the source of high latency, techniques such as indexing, query optimisation, or even denormalisation could be considered to improve its performance.



Load testing vs stress testing (Image source (https://www.testbytes.net/blog/difference-between-load-testing-and-stress-testing/))

Furthermore, **load testing** and **stress testing** are essential techniques for understanding the limits of the application system and its ability to scale. By simulating real-world usage patterns with increasing loads, these tests can help identify the breaking points of the system and provide data on

how performance degrades under stress. This information is critical for setting realistic scalability goals and for planning the necessary infrastructure improvements or algorithmic optimisations. It is also important to **continuously monitor** the system after deployment to ensure that performance and scalability goals are being met and to adjust strategies as needed based on actual usage patterns and emerging technologies.

Best practices for defining and documenting performance and scalability requirements

Defining and documenting performance and scalability requirements is a critical step in the design and development of application systems. Best practices in this area involve a **clear and collaborative** approach that ensures all stakeholders have a shared understanding of the system's expected behaviour under various conditions. One key practice is to involve end-users, system architects, and operations teams **early in the process** to gather comprehensive performance criteria. This includes identifying the expected number of concurrent users, the types of operations that will be performed, and the acceptable response times for different functions. Documenting these requirements in a way that is both detailed and accessible to all team members helps to set the stage for system design that meets these performance goals.

Another best practice is to use **specific and measurable metrics** when defining performance and scalability requirements. This means **avoiding vague terms** and instead providing concrete numbers that can be tested against. For example, instead of stating that the system should be "fast," specify that it must support 10,000 transactions per second with a maximum latency of 100 milliseconds. Similarly, **scalability requirements** should be documented with clear indicators of how the system should behave as load increases, such as the ability to linearly increase throughput by adding more servers or the maximum load a single server can handle before performance degrades.

Finally, it is important to keep performance and scalability requirements **living documents** that evolve with the application system. As the system is developed and tested, new insights into its performance characteristics will emerge. Regularly reviewing and updating these requirements based on feedback from load testing, profiling, and real-world usage ensures that the documentation remains relevant and that the system can continue to meet the demands placed upon it. Additionally, as technology advances and user expectations change, the requirements may need to be revised to reflect new benchmarks for performance and scalability.

Examples of performance and scalability requirements for various application system domains

Performance and scalability requirements can vary significantly across different application system domains, reflecting the unique demands and user expectations of each field. Here are examples of such requirements for several domains:

1. **E-commerce Platforms**:
    - Performance: The system must handle peak traffic during sales events, with a maximum response time of 2 seconds for page loads and 3 seconds for checkout processes.
    - Scalability: The platform should automatically scale to accommodate up to a 100% increase in user traffic during high-demand periods without manual intervention.
2. **Online Banking Systems**:
    - Performance: Transaction processing time must not exceed 5 seconds, with real-time balance updates and a 99.99% uptime guarantee.
    - Scalability: The system must support up to 50,000 concurrent users performing transactions without degradation in performance.
3. **Social Networking Applications**:
    - Performance: News feed generation and updates should occur within 1 second, with friend suggestions and notifications delivered within 3 seconds.
    - Scalability: The application must scale to support millions of daily active users, with the ability to handle sudden spikes in usage due to trending topics.
4. **High-Frequency Trading Systems**:
    - Performance: Trade execution and order processing must occur within milliseconds to capitalise on market opportunities.
    - Scalability: The system must handle thousands of trades per second during market volatility without latency increases.
5. **Healthcare Information Systems**:
    - Performance: Patient record retrieval and update operations should complete within 2 seconds to support fast-paced clinical environments.
    - Scalability: The system must accommodate the addition of new medical facilities and patient data without performance degradation, supporting up to a 30% annual growth in user base.
6. **Video Streaming Services**:

- Performance: Streaming startup time should not exceed 2 seconds, with seamless playback at 1080p resolution and minimal buffering.
- Scalability: The service must scale to support global audiences, delivering content to millions of concurrent viewers during live events.

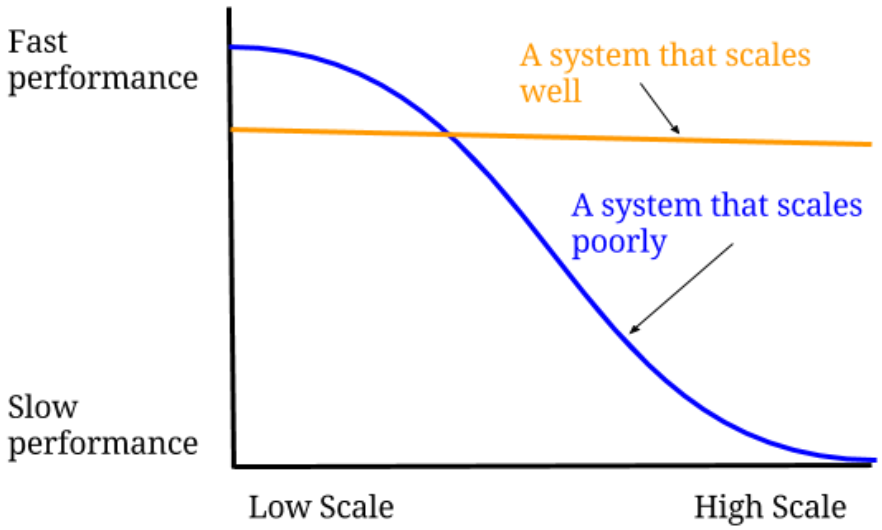
7. **Mobile Applications**:

- Performance: App launch time should be under 2 seconds, with smooth UI interactions and minimal delay for data-driven features.
- Scalability: The backend services supporting the mobile app must scale to handle an increasing number of devices, with the ability to support a global user base.

Each of these examples illustrates how performance and scalability requirements are tailored to the specific needs and use cases of different application system domains, ensuring that the systems can deliver a reliable and responsive user experience under various operational conditions.

Common pitfalls and challenges in identifying and addressing performance and scalability requirements

Identifying and addressing performance and scalability requirements is a complex task that often comes with several pitfalls and challenges. One common pitfall is the failure to accurately **predict future growth and load patterns**. Organisations may underestimate the potential user base or the intensity of peak usage periods, leading to systems that are inadequately prepared for real-world demands. This can result in costly last-minute upgrades or, worse, system failures under high load.



Performance vs Scalability (Image source (https://www.linkedin.com/pulse/performance-vs-scalability-rajaram-joshi/))

Another challenge lies in the **trade-offs between performance and scalability**. Improving one aspect can sometimes negatively impact the other. For example, optimising a system for maximum performance with the latest hardware might limit its scalability if the same level of hardware cannot

be easily replicated as the user base grows. Conversely, designing for infinite scalability might come at the cost of reduced performance due to the overhead of distributed systems. Striking the right balance requires a deep understanding of the application's usage patterns and a willingness to iterate on the design.

Moreover, the complexity of modern application systems, which often involve multiple layers of software and hardware, can make it difficult to pinpoint the source of **performance bottlenecks**. A seemingly straightforward scalability issue might have roots in various components, such as the database, the application server, the network, or even third-party services. Diagnosing these problems requires a holistic approach to monitoring and profiling, which can be resource-intensive and may not always yield clear answers.

Lastly, there is the challenge of **maintaining performance and scalability** as the application evolves. New features and changes to the codebase can introduce inefficiencies or disrupt the delicate balance of resources that was previously optimised. Continuous performance testing and a culture of performance awareness are necessary to ensure that improvements do not come at the cost of degraded performance or reduced scalability. However, this requires ongoing investment in tools, expertise, and time, which can be a significant challenge for organisations with limited resources.

▼ Supporting content B - Resource utilisation and contention

Overview of common resource utilisation and contention issues in application systems

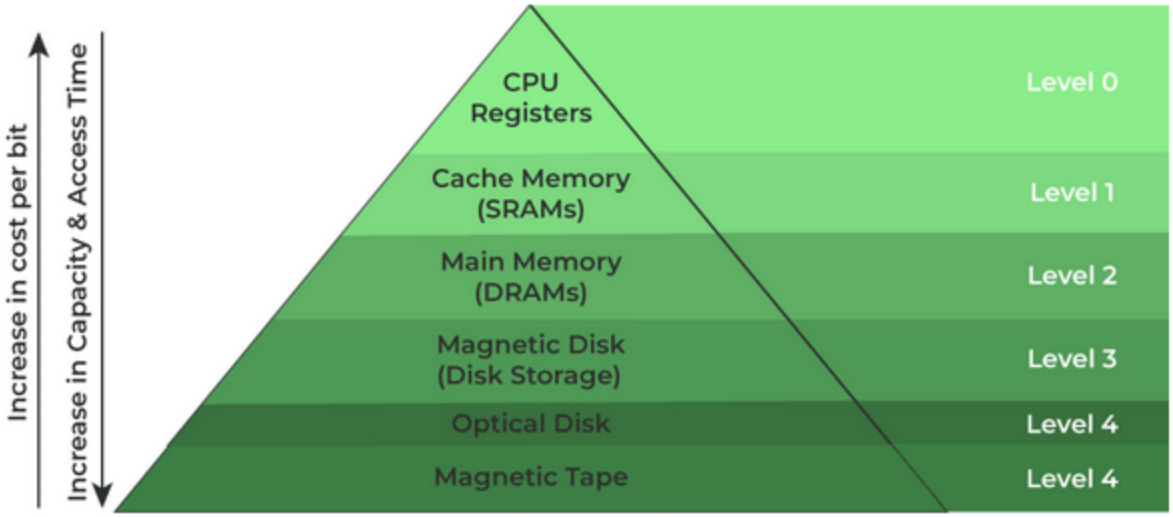
Resource utilisation and contention are critical aspects of application system performance, particularly in environments where multiple processes or users compete for limited resources. Common **resource utilisation** issues arise when system resources such as CPU, memory, disk I/O, and network bandwidth are not **efficiently managed**, leading to suboptimal performance. For instance, a poorly designed application may consume excessive CPU cycles due to inefficient algorithms or memory due to memory leaks, resulting in slower response times and reduced throughput.

Contention, on the other hand, occurs when two or more processes attempt to access the same resource simultaneously, leading to delays and increased wait times. This is particularly problematic in multi-user environments or applications with high concurrency requirements. For example, database locks can cause contention when multiple transactions try to access the same data, leading to deadlocks or increased latency. Similarly, network bandwidth contention can occur when multiple applications try to send data over the network at the same time, resulting in packet loss or reduced transmission speeds.

To address these issues, system administrators and developers must employ various strategies. These can include **optimising code** to reduce resource consumption, implementing **resource scheduling and allocation policies**, and using **locking mechanisms** that minimise contention. Additionally, monitoring tools and performance profiling can help identify bottlenecks and guide the implementation of targeted optimisations. By understanding and addressing resource utilisation and contention issues, it is possible to improve the overall performance and scalability of application systems.

Techniques for monitoring and analysing resource utilisation, such as CPU, memory, and I/O

Monitoring and analysing resource utilisation is essential for maintaining the performance and efficiency of application systems. This involves tracking the **consumption of critical resources** such as CPU, memory, and I/O to identify bottlenecks and areas for optimisation. One of the primary techniques for monitoring resource utilisation is the use of **system monitoring tools**. These tools can provide real-time data on resource usage patterns, allowing administrators to detect spikes or trends that may indicate underlying issues. For example, CPU usage can be monitored using tools like top, htop, or Windows Performance Monitor, which display the current load on the CPU and can help identify processes that are consuming excessive CPU cycles.



Memory Hierarchy (Image source (https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/))

Memory utilisation is another critical aspect to monitor, as insufficient memory can lead to increased use of swap space and a significant drop in performance. Tools such as free, vmstat, or the Windows Task Manager can be used to monitor memory usage, including the amount of free memory, cached memory, and swap space usage. By analysing memory usage patterns, it is possible to identify memory leaks or inefficient memory management within applications.

I/O utilisation is equally important, as slow disk or network I/O can severely impact application performance. Tools like iostat, sar, or Windows Resource Monitor can provide insights into disk

read/write speeds and network throughput. By monitoring I/O wait times and utilisation rates, administrators can identify I/O-bound processes and optimise file system layouts, caching mechanisms, or network configurations to improve performance.

In addition to these monitoring tools, **logging and tracing mechanisms** can provide detailed insights into resource utilisation. System logs can record resource usage over time, allowing for historical analysis and the identification of long-term trends. Application-level logging can also provide specific information about resource consumption within applications. Tracing tools, such as strace or DTrace, can be used to monitor system calls and application behaviour in real-time, offering a deep understanding of how resources are being used at the code level. Combining these monitoring and analysis techniques enables a comprehensive approach to resource utilisation management, leading to more efficient and responsive application systems.

Strategies for optimising resource allocation and minimising contention, such as vertical and horizontal scaling

Optimising resource allocation and minimising contention in application systems is crucial for ensuring that resources are used efficiently and that the system can handle the demands placed upon it. One strategy for achieving this is through **vertical scaling**, which involves increasing the capacity of existing servers by adding more resources such as CPU, memory, or storage. This approach can be effective for workloads that are centralised and have predictable growth patterns. By vertically scaling, it is possible to maximise the utilisation of existing hardware and delay the need for additional servers. However, vertical scaling has its limitations, as there is a physical cap to the amount of resources that can be added to a single server.

Horizontal scaling, on the other hand, involves adding more servers to a system to distribute the workload across multiple machines. This strategy is particularly useful for handling unpredictable or rapidly growing workloads, as it allows for more flexibility and redundancy. Horizontal scaling can help minimise contention by ensuring that no single resource becomes a bottleneck. Load balancing is a key component of horizontal scaling, as it distributes network traffic across multiple servers, preventing any one server from becoming overwhelmed. This approach also facilitates high availability, as the system can continue to operate even if one or more servers fail.

To effectively implement scaling strategies, it is important to **monitor resource utilisation and performance metrics**. This data can inform decisions about when and how to scale the system. Additionally, designing applications to be stateless or to use distributed caching and storage can make them more amenable to horizontal scaling. Containerisation and orchestration tools, such as Docker and Kubernetes, can further simplify the process of scaling by automating the deployment, scaling, and management of application containers across a cluster of servers. By combining these

strategies with right-sising of resources and intelligent scheduling, organisations can optimise resource allocation, minimise contention, and maintain high levels of performance and reliability in their application systems.

Case studies and real-world examples of resource utilisation optimisation in complex application systems

Case Study 1: Optimising CPU Utilisation in a High-Traffic Web Application

A large e-commerce company was experiencing performance issues during peak shopping seasons, leading to a significant increase in page load times and a degradation of the user experience. Upon analysing resource utilisation, it was discovered that the CPU was the bottleneck, with utilisation spiking to 100% during traffic peaks.

To address this, the company implemented several optimisation strategies. First, they refactored the application code to improve algorithm efficiency, reducing the CPU load by 30%. Additionally, they introduced caching mechanisms to store frequently accessed data, which reduced the number of CPU-intensive operations required per user request. Finally, they adopted horizontal scaling by adding more web servers and implementing a load balancer to distribute traffic evenly across the server fleet. These measures combined to optimise CPU utilisation, resulting in a smoother user experience even during high-traffic periods.

Case Study 2: Memory Management in a Big Data Processing System

A data analytics firm was struggling with memory contention in their big data processing system, which relied on in-memory computations for real-time analytics. As the volume of data grew, the system began to swap memory to disk, causing significant performance degradation.

The firm addressed the issue by analysing memory usage patterns and identifying memory leaks within their data processing framework. They also optimised data structures and reduced the memory footprint of their analytics algorithms. Furthermore, they implemented a more aggressive garbage collection policy to reclaim memory more frequently. To handle growth, they invested in servers with more RAM and used vertical scaling to increase memory capacity. These optimisations allowed the firm to continue providing real-time analytics without the previous performance bottlenecks.

Case Study 3: I/O Optimisation in a Financial Trading Platform

A financial trading platform was facing I/O contention due to the high volume of transaction logs written to disk, which was critical for regulatory compliance but was slowing down the system.

The platform's development team conducted an analysis of I/O operations and identified that the disk subsystem was the primary bottleneck. They optimised the disk I/O by implementing a more efficient logging mechanism that batch-wrote transactions to disk, reducing the number of I/O operations. Additionally, they upgraded to solid-state drives (SSDs) to increase I/O throughput and reduce latency. To further minimise contention, they introduced a tiered storage solution, keeping hot data on SSDs and archiving older data on slower, high-capacity drives. These changes significantly improved the platform's performance, allowing for faster trade executions and more responsive user interfaces.

Real-World Example: Cloud Service Providers and Auto-Scaling

Cloud service providers like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) offer auto-scaling features that dynamically adjust resource allocation based on demand. For example, a mobile gaming company using AWS can set up auto-scaling policies that automatically provision more servers during peak gaming hours and scale down during off-peak times. This approach ensures that the game servers have the necessary CPU, memory, and network resources to handle varying loads, optimising resource utilisation and minimising costs for the company.

These case studies and examples demonstrate the importance of resource utilisation optimisation in complex application systems and illustrate how different strategies can be applied to address specific bottlenecks in CPU, memory, and I/O resources.

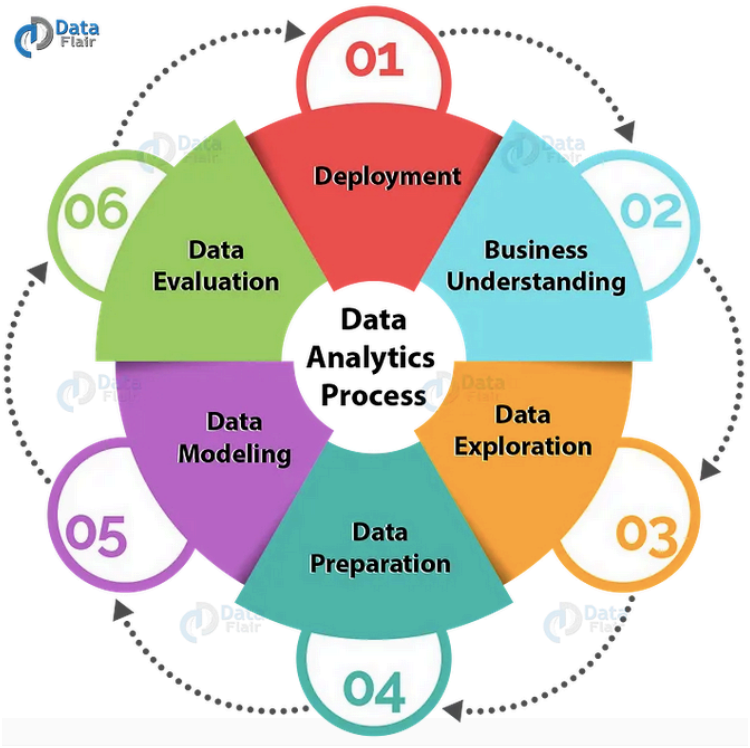
▼ Supporting content C - Data management and storage

Overview of data management and storage challenges in application system performance and scalability

Data management and storage are critical components of application system performance and scalability. As applications grow in complexity and user base, the volume, velocity, and variety of data they handle increase exponentially. This growth can lead to challenges such as **slow data retrieval times**, **inefficient data processing**, and difficulties in maintaining **data integrity** and **consistency** across various storage systems. Moreover, the distributed nature of modern applications, often spread across multiple data centers or cloud platforms, adds to the complexity of data management. Ensuring that data is synchronised and accessible in a timely manner across all nodes is a significant challenge that can impact application performance and user experience.

Scalability is another key challenge in data management and storage. As the number of users and the amount of data grow, the storage infrastructure must be able to scale out seamlessly to accommodate the increased load. This often requires a shift from traditional, vertically scaled systems to horizontally scalable architectures that can distribute data across numerous servers.

However, this shift also introduces complexities in data partitioning, replication, and load balancing, all of which must be managed to ensure that the application remains performant and responsive under varying levels of demand.



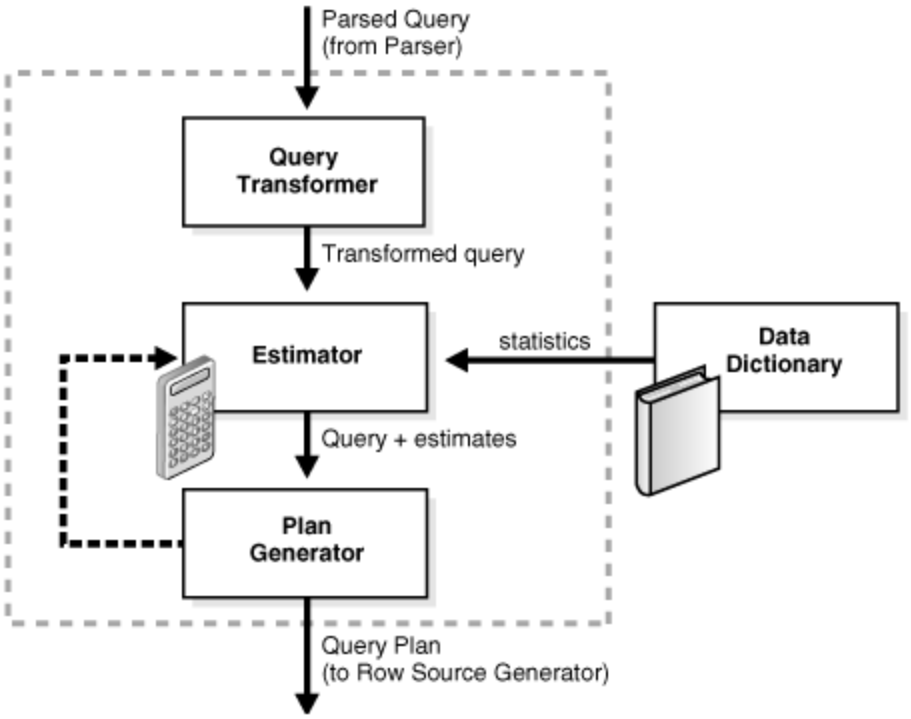
Big data analytics (Image source (https://medium.com/@patelharshali136/what-is-big-data-analytics-and-why-it-is-so-important-1de86fa37540))

Furthermore, the rise of **big data analytics** and the need for **real-time data processing** have placed additional burdens on data management and storage systems. Applications must now support complex queries and analytics workloads alongside their operational tasks, requiring storage solutions that can provide both high throughput and low latency. Additionally, the management of **unstructured data**, such as images, videos, and logs, presents unique challenges in terms of storage optimisation and retrieval mechanisms. Addressing these challenges requires a strategic approach to data management that includes the use of advanced technologies like in-memory databases, NoSQL databases, and distributed file systems, as well as robust data governance and lifecycle management policies.

Techniques for optimising database performance, such as indexing, query optimisation, and denormalisation

Optimising database performance is a multifaceted endeavor that involves various techniques aimed at improving the speed and efficiency of data retrieval and processing. One of the primary techniques is **indexing**, which involves creating pointers to data in the tables to facilitate quick access. By using indexes, databases can significantly reduce the amount of data that needs to be read from disk, thereby speeding up query execution times. However, it is important to carefully select which

columns to index, as excessive indexing can lead to increased overhead during data insertion, update, and deletion operations.



Query optimisation (**Image source** (https://docs.oracle.com/en/database/oracle/oracle-database/18/tgsql/query-optimizer-concepts.html#GUID-12C47112-B713-4658-89C2-DA756E4D29D3))

Query optimisation is another critical technique for enhancing database performance. This involves analysing and rewriting SQL queries to make them more efficient. Techniques include avoiding unnecessary joins, using subqueries judiciously, and leveraging database-specific features and functions that can process data more quickly. Database administrators and developers often use query analysis tools to identify slow-running queries and then apply optimisation strategies such as rewriting the query logic, adding missing indexes, or changing table structures to reduce complexity and improve execution time.

Denormalisation is a technique that involves deviating from strict normalisation rules to improve performance. In a normalised database, data is organised into multiple related tables to eliminate redundancy and ensure data integrity. However, this can lead to complex and slow queries due to the need for multiple joins. Denormalisation, on the other hand, involves combining related data into a single table, which can reduce the number of joins and simplify queries, leading to faster execution times. This approach must be balanced against the potential for data redundancy and the increased complexity of data updates and maintenance.

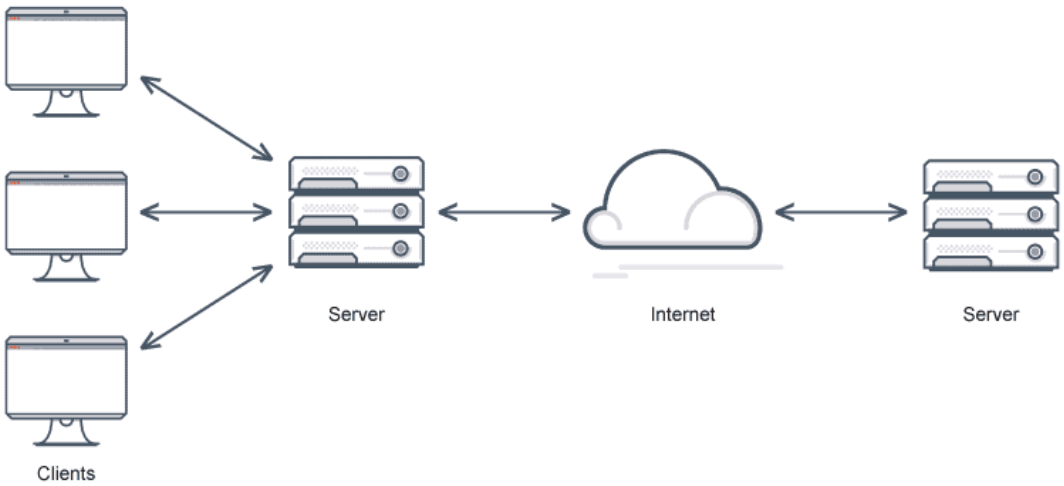
In addition to these techniques, other strategies for optimising database performance include **partitioning large tables** to distribute load, using **caching mechanisms** to store frequently accessed data in memory, and implementing **connection pooling** to reduce the overhead of establishing database connections. Each of these techniques must be considered in the context of the specific application and database workload to ensure that the optimisations are appropriate and

effective. Regular monitoring and performance tuning are also essential to adapt to changing data patterns and user behaviours over time.

Strategies for scaling data storage and management, such as sharding, replication, and caching

Scaling data storage and management is a critical aspect of maintaining the performance and availability of application systems as they grow in popularity and data volume. One of the primary strategies for scaling is **sharding**, which involves distributing data across multiple servers or databases. Each shard holds a portion of the entire dataset, and this distribution can be based on various criteria such as hashing, ranges, or geographical location. Sharding helps to balance the load and allows for horizontal scaling, where more servers can be added to the system to handle increased demand. It also enables parallel processing of queries, which can significantly improve response times.

Replication is another key strategy for scaling data storage and management. It involves creating multiple copies of data and storing them on different servers. Replication can be synchronous or asynchronous and is often used to ensure high availability, reduce latency, and provide redundancy in case of hardware failures or other issues. By having copies of data in different locations, read operations can be distributed across replicas, thereby reducing the load on any single server. Replication also plays a crucial role in disaster recovery, as it allows for quick failover to a replica in the event of a primary database outage.



Content caching (**Image source** (https://avinetworks.com/glossary/content-caching-2/))

Caching is a technique that can dramatically improve the performance of data-intensive applications by storing frequently accessed data in a fast-access storage layer. Caching can be implemented at various levels, including in-memory caches like Redis or Memcached, application-level caches, or even within the database itself. By storing data closer to the application or in a more accessible

location, caching reduces the need to fetch data from slower disk-based storage, thus speeding up data retrieval. Caching is particularly effective for data that does not change often or for read-heavy workloads. However, it requires careful management to ensure that the cached data remains consistent with the source data and to handle cache invalidation properly.

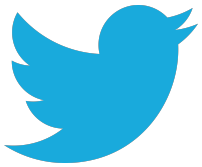
No SQL databases (**Image source (https://fauna.com/blog/nosql-databases-non-relational-databases-explained)**)

Finally, the **choice of storage technology** can also impact the scalability of data management. Traditional relational databases may not always be the best fit for large-scale applications due to their inherent limitations in horizontal scalability. NoSQL databases, such as document stores, key-value stores, and wide-column stores, offer alternative models that can scale out more easily. These systems often sacrifice some of the transactional guarantees of relational databases but provide the flexibility and scalability needed for modern web and mobile applications. Additionally, object storage solutions and distributed file systems can be used for handling large volumes of unstructured data, further expanding the scalability of the overall data storage infrastructure.

Case studies and real-world examples of data management optimisation in complex application systems

Case Study 1: Facebook's Use of TAO for Social Graph Management

Facebook's social graph is a complex application system that requires efficient data management to handle the vast amount of data generated by its user base. To optimise performance, Facebook developed TAO, a distributed database system built on top of MySQL. TAO is designed to handle complex queries that involve multiple joins across different types of entities, such as users, pages, photos, and events. By using a **combination (https://lms.griffith.edu.au/courses/24045/files/6203547?wrap=1) (https://lms.griffith.edu.au/courses/24045/files/6203547/download?download_frd=1)** of indexing, caching, and data denormalisation, TAO is able to provide sub-second read/write performance for complex queries that touch billions of edges.



Case Study 2: Twitter's Transition to a Denormalised Data Model

Twitter faced significant challenges in scaling its data management infrastructure to handle the rapid growth of its user base and the volume of tweets. Initially, Twitter used a normalised data model, which led to performance issues due to the complexity of queries and the number of joins required. To address this, Twitter transitioned to a denormalised data model using a custom data store called Manhattan. This allowed Twitter to simplify its queries and reduce the load on its databases, resulting in improved performance and scalability.

Case Study 3: Netflix's Use of Amazon DynamoDB

Netflix moved its recommendation and personalisation systems to Amazon DynamoDB, a NoSQL database service, to handle the scale and performance demands of its streaming service. By using DynamoDB's auto-scaling features, Netflix can automatically adjust capacity based on traffic, ensuring that the system can handle peak loads without manual intervention. Netflix also leverages DynamoDB Accelerator (DAX) for caching to reduce latency for read-heavy workloads. This combination of auto-scaling and caching has allowed Netflix to provide a seamless user experience even during peak viewing times.

Case Study 4: Uber's Migration to Microsoft Azure Cosmos DB

Uber needed a database that could handle its global operations and provide low-latency access to data for its real-time applications. To achieve this, Uber migrated its core trip data storage system to Microsoft Azure Cosmos DB, a globally distributed, multi-model database service. Cosmos DB's turnkey global distribution allowed Uber to replicate its data across multiple regions, ensuring low-latency access for users and drivers around the world. Additionally, Cosmos DB's ability to elastically scale throughput and storage on demand helped Uber to manage its unpredictable workloads efficiently.

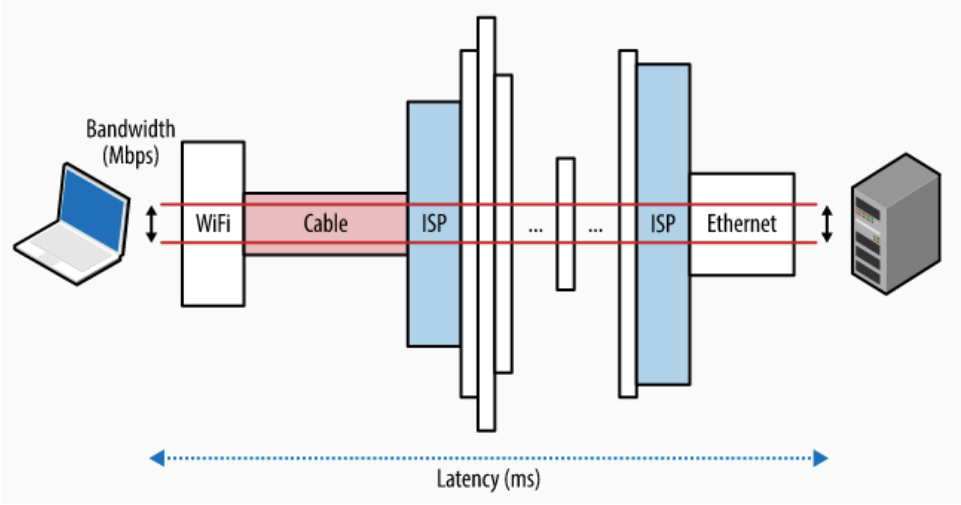
Case Study 5: Airbnb's Development of Aurora

Airbnb experienced rapid growth, which led to scaling challenges with its MySQL databases. To optimise performance, Airbnb worked with Amazon Web Services to develop Amazon Aurora, a MySQL and PostgreSQL-compatible relational database built for the cloud. Aurora automatically grows storage and compute resources as needed, and it offers up to five times the throughput of standard MySQL databases. Airbnb's use of Aurora has allowed it to scale its data management infrastructure while maintaining the transactional integrity and familiarity of relational databases.

These case studies demonstrate how large-scale application systems have successfully optimised their data management strategies to handle growth, improve performance, and ensure reliability. Each company has taken a different approach based on its specific needs and the nature of its data, highlighting the importance of choosing the right optimisation techniques for the task at hand.

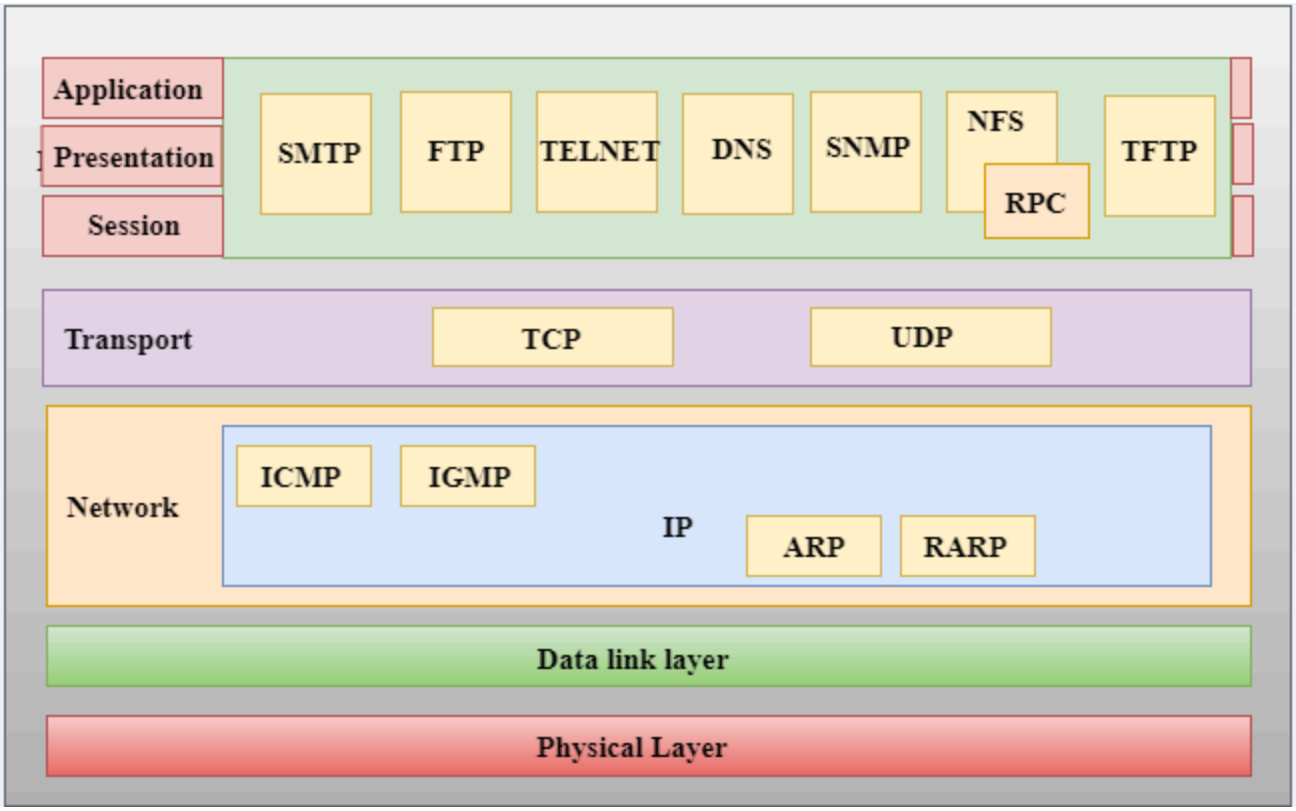
▼ Supporting content D - Network and communication overhead

Overview of network and communication overhead issues in application system performance and scalability



Bandwidth vs latency (Image source (https://hpbn.co/primer-on-latency-and-bandwidth/))

Network and communication overhead can significantly impact the performance and scalability of application systems. In the context of distributed applications, where components are spread across different networked environments, the latency and bandwidth associated with data transmission can become critical factors. **Communication overhead** refers to the additional resources and time required to facilitate the exchange of data between different parts of an application. This includes not only the data itself but also the protocols and mechanisms used to ensure reliable transmission, such as error checking and correction, handshaking, and encryption. High overhead can lead to delays in data processing, increased response times, and reduced throughput, which can degrade the user experience and limit the scalability of the application.



TCP/IP (Image source (https://www.javatpoint.com/computer-network-tcp-ip-model))

One of the primary sources of network overhead is the **protocol stack** used for communication. Protocols like TCP/IP, while reliable and widely supported, can introduce significant overhead due to their robust error correction and flow control mechanisms. Additionally, the use of **synchronous communication**, where processes wait for a response before proceeding, can exacerbate performance issues by causing threads or processes to block, leading to inefficient use of system resources. Furthermore, the **serialisation and deserialisation of data**, necessary for transmission over a network, can also contribute to overhead, particularly when dealing with complex data structures or large volumes of data.

To address these issues, several strategies can be employed to minimise network and communication overhead. **Optimising the data transmission** by reducing the size of payloads, compressing data, and minimising the frequency of updates can significantly reduce the burden on the network. Adopting **asynchronous communication patterns**, such as message queues or event-driven architectures, can improve responsiveness and throughput by allowing processes to continue without waiting for a response. Additionally, choosing the **right communication protocol** for the specific use case, such as UDP for real-time applications where some data loss is acceptable, can help reduce overhead. Lastly, implementing **caching strategies** and **edge computing** can bring data processing closer to the user, reducing the distance over which data must travel and thus minimising latency.

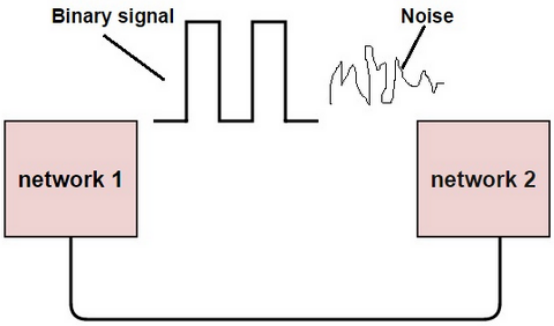
Techniques for monitoring and analysing network performance, such as latency, throughput, and error rates

Monitoring and analysing network performance is crucial for identifying bottlenecks and ensuring that application systems operate efficiently. Several techniques can be employed to measure key performance indicators such as latency, throughput, and error rates. One common method is the use of **network monitoring tools** that can continuously track these metrics in real-time. These tools can provide insights into the health of the network, helping administrators to quickly identify and address issues that may arise.

Latency, the time it takes for a packet of data to travel from one designated point to another, is a critical factor in network performance. Techniques for measuring latency include **ping tests**, which measure the round-trip time for messages sent from the origin to the destination and back, and **traceroute**, which identifies the path taken by packets and the delay at each hop along the way. By analysing latency, it is possible to determine if network delays are causing performance issues in application systems.

Throughput vs latency (Image source (https://www.homeowner.com/connectivity/latency-vs-throughput))

Throughput, the amount of data that can be transferred over a network in a given period, is another essential metric. Tools like **bandwidth monitors** can measure throughput by tracking the volume of data passing through network interfaces over time. Analysing throughput helps in understanding whether the network is capable of handling the required data load and if there are any bottlenecks that need to be addressed. For instance, if the throughput is consistently lower than the available bandwidth, it may indicate issues such as network congestion or inefficient data transfer protocols.

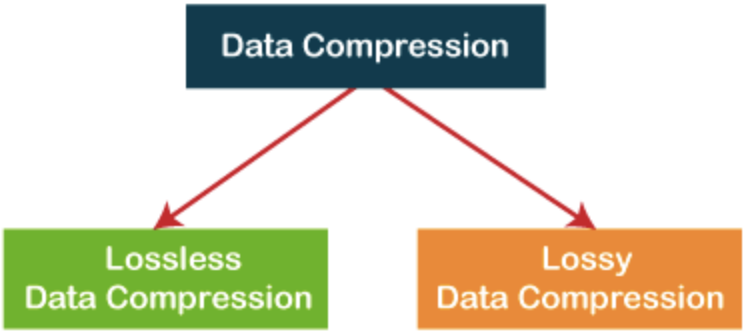


Error detection and correction codes (Image source (https://www.electronicshub.org/error-correction-and-detection-codes/))

Error rates, which reflect the frequency of data transmission failures, are also important for assessing network performance. Techniques for monitoring error rates include **checking the logs** of network devices for error messages and using specialised software that can **detect and report errors in real-time**. High error rates can significantly degrade application performance, leading to data corruption, increased retransmissions, and user dissatisfaction. By analysing error patterns, network administrators can pinpoint problematic areas, such as faulty hardware or poor signal quality, and take corrective actions to improve reliability.

In summary, a combination of network monitoring tools and specific techniques for measuring latency, throughput, and error rates is essential for effective performance analysis. These methods enable network administrators to gain a comprehensive understanding of network behaviour, identify potential issues, and implement targeted solutions to optimise application system performance.

Strategies for optimising network communication, such as compression, batching, and asynchronous processing



Data compression (Image source (https://codegyan.in/articles/what-is-data-compression-explain-its-types.htm))

Optimising network communication is essential for improving the performance and scalability of application systems. One strategy for achieving this is through **data compression**, which reduces the size of data packets before they are transmitted over the network. By compressing data, it is possible to lower the bandwidth requirements and minimise the time needed for transmission. Various compression algorithms can be used, ranging from lightweight techniques that have minimal impact on CPU usage to more complex algorithms that achieve higher compression ratios at the cost of additional processing power. The choice of compression strategy depends on the specific needs of the application and the available computational resources.

Another optimisation technique is **batching**, which involves grouping multiple small messages or requests into a single larger transmission. This approach can significantly reduce the number of round trips required for communication, thereby decreasing latency and improving throughput. Batching is particularly effective in scenarios where there are many small, independent operations that can be easily aggregated, such as database writes or sensor data uploads. However, it is important to balance the benefits of reduced network overhead with the potential drawbacks of increased latency for individual operations within a batch.

Asynchronous processing is a strategy that can be used to optimise network communication by allowing the application to continue executing other tasks while waiting for a response from the network. This is particularly useful in distributed systems where components may need to communicate with remote services or databases. By using asynchronous I/O operations, threads are not blocked waiting for network responses, which can lead to more efficient use of system resources and improved responsiveness. Implementing asynchronous communication often involves using callbacks, futures, promises, or reactive programming patterns that handle the complexities of managing concurrent operations and their results.

In addition to these strategies, it is important to consider the design of the **network communication protocols** themselves. Protocols that are lightweight and tailored to the specific needs of the application can minimise overhead. For example, using a binary protocol instead of a text-based one can reduce the size of the data being transmitted. Furthermore, choosing the right transport protocol (such as TCP for reliability or UDP for low latency) and optimising its parameters (such as window size and buffer settings) can also contribute to better network performance. By combining these

optimisation techniques, developers can create application systems that communicate efficiently over the network, leading to improved user experiences and reduced operational costs.

Case studies and real-world examples of network optimisation in complex application systems

Network optimisation in complex application systems is a critical practice that can lead to significant improvements in performance, scalability, and user experience. Here are several case studies and real-world examples that illustrate how network optimisation techniques have been applied in various contexts:

1. **High-Frequency Trading Systems**:
   In the financial industry, high-frequency trading (HFT) systems require extremely low latency for competitive advantage. One example is the use of microwave communication links between stock exchanges and trading firms. These links provide a straight-line path that reduces the distance data needs to travel, resulting in lower latency compared to fiber-optic cables. Additionally, HFT systems often employ compression algorithms to minimise the size of market data feeds, and they use asynchronous processing to handle the high volume of trade orders without blocking.

2. **Content Delivery Networks (CDNs)**:
   CDNs are a prime example of network optimisation on a global scale. By caching content at edge locations closer to users, CDNs reduce latency and improve the loading times of web pages and media content. Akamai, one of the largest CDN providers, optimises network communication by dynamically routing traffic around congestion and outages, ensuring high availability and performance. They also use compression and adaptive bitrate streaming to optimise the delivery of content based on the user's network conditions.

3. **Cloud Gaming Services**:
   Cloud gaming platforms like Google Stadia and NVIDIA GeForce Now face the challenge of streaming high-quality video games over the internet with minimal latency. These services use predictive models to anticipate user inputs and reduce the perceived latency. They also optimise network communication by using custom protocols that are more efficient than standard HTTP/HTTPS for real-time data transmission. Additionally, these platforms leverage Google's and NVIDIA's vast network infrastructure to ensure low latency and high throughput.

4. **Distributed Computing Platforms**:
   Apache Hadoop and Apache Spark are examples of distributed computing systems that handle large volumes of data across clusters of machines. Network optimisation is crucial for these platforms to ensure that data shuffling and communication between nodes do not become bottlenecks. Techniques such as data locality optimisation, where computation is brought to the data to minimise network transfer, and efficient serialisation formats, like Apache Avro or Protocol Buffers, are used to reduce overhead.

5. **Mobile Applications**:
   Mobile apps often need to optimise network communication due to the unreliable and limited bandwidth of mobile networks. Techniques such as delta encoding, where only changes in data are transmitted, and image compression are commonly used to reduce the amount of data sent over the network. For example, the Instagram app uses compression and progressive JPEG loading to optimise the display of images over mobile networks, improving the user experience.

6. **IoT (Internet of Things) Networks**:
   In IoT networks, devices often have limited processing power and bandwidth. Protocols like MQTT (Message Queuing Telemetry Transport) and CoAP (Constrained Application Protocol) are designed to be lightweight and efficient for these environments. For instance, smart home systems use MQTT to send sensor data and commands with minimal overhead, allowing for reliable communication even on low-bandwidth connections.

These case studies demonstrate the diverse applications of network optimisation techniques and the significant impact they can have on the performance of complex application systems. Whether it's through the use of specialised communication protocols, data compression, caching strategies, or infrastructure improvements, network optimisation remains a key area of focus for developers and organisations seeking to enhance their application systems.

▼ Supporting content E - Algorithmic and computational efficiency

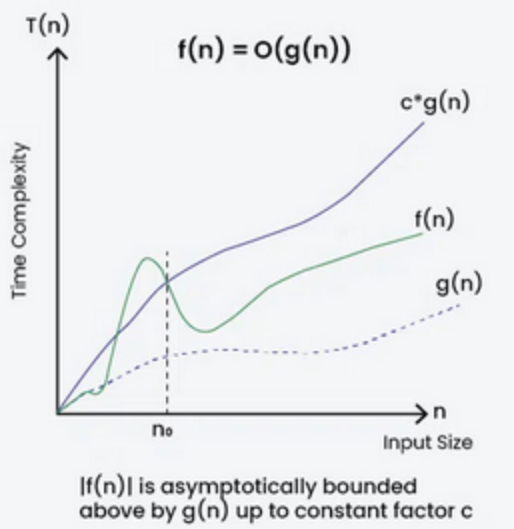
Overview of algorithmic and computational efficiency considerations in application system performance

Algorithmic and computational efficiency are critical considerations in the performance of application systems, as they directly impact the speed, scalability, and overall effectiveness of software applications. At its core, **algorithmic efficiency** refers to how well an algorithm performs in terms of time and space complexity, which are measures of the computational resources required to execute the algorithm. An efficient algorithm can process data more quickly, handle larger datasets, and operate with fewer hardware resources, leading to better application performance.

In the context of application systems, the **choice of algorithms** can significantly affect user experience and system responsiveness. For instance, a poorly optimised search algorithm can result in slow query responses, while an inefficient sorting algorithm can cause delays in data processing tasks. Moreover, as application systems grow in complexity and data volume, the importance of **computational efficiency** becomes even more pronounced. Developers and system architects must carefully analyse and select algorithms that not only solve the problem at hand but also do so in a manner that is efficient and scalable.

To achieve computational efficiency, developers often employ a variety of **optimisation techniques**. These can include algorithmic optimisations such as dynamic programming, divide and conquer, and greedy algorithms, which reduce the time complexity of operations. Additionally, computational efficiency can be enhanced through the use of appropriate data structures that facilitate faster access and manipulation of data. Memory management techniques, such as caching and garbage collection, also play a crucial role in optimising computational efficiency by ensuring that the application makes the best use of available memory resources.

Techniques for analysing and optimising algorithms, such as time complexity analysis and space complexity analysis



Big O analysis (**Image source** (https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/))

Analysing algorithms for their time and space complexity is a cornerstone of optimising application system performance. **Time complexity** analysis involves determining the amount of time an algorithm takes to process data as a function of the input size. This is commonly expressed using **Big O notation**, which provides a simplified representation of an algorithm's efficiency by focusing on the worst-case scenario. By understanding the time complexity, developers can predict how an algorithm will behave with large datasets and identify potential performance bottlenecks.

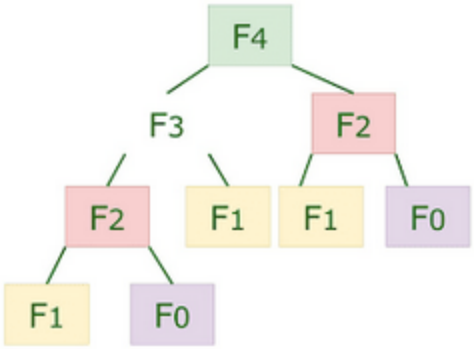
Space complexity analysis, on the other hand, evaluates the amount of memory an algorithm requires to execute. This is also expressed in Big O notation and is crucial for applications with limited memory resources or those that handle substantial amounts of data. Optimising space complexity involves minimising the memory footprint of an algorithm without compromising its functionality, which can be achieved through techniques such as using more compact data structures or avoiding unnecessary data duplication.

To optimise algorithms based on these analyses, developers can employ various strategies. One approach is to select more **efficient algorithms or data structures** that inherently have lower time

or space complexity for the required operations. Another is to **refactor existing algorithms** to reduce their complexity, for example, by eliminating redundant computations or improving loop efficiency. Additionally, **caching** and **lazy loading techniques** can be used to optimise both time and space complexity by storing frequently accessed data in a quickly retrievable form and loading data into memory only when needed. Profiling tools are essential in this process, as they help identify which parts of the algorithm are most in need of optimisation.

Strategies for improving computational efficiency, such as memoisation, dynamic programming, and parallelisation

Improving computational efficiency is a multifaceted endeavor that involves employing strategies to reduce the time and resources an algorithm consumes. One such strategy is **memoisation**, which is a technique where the results of expensive function calls are cached and returned when the same inputs occur again. This is particularly useful for recursive algorithms that solve overlapping subproblems, as it can drastically reduce the number of redundant calculations. By storing the results of subproblems in a lookup table, memoisation ensures that each unique problem is solved only once, leading to significant improvements in computational efficiency.



Dynamic programming (**Image source** (https://www.geeksforgeeks.org/dynamic-programming/))

Dynamic programming is another powerful strategy for enhancing computational efficiency. It is a method for solving complex problems by breaking them down into simpler subproblems, solving each subproblem only once, and storing the results to avoid redundant work. Dynamic programming is applicable when problems have the optimal substructure property (optimal solutions can be constructed from optimal solutions of its subproblems) and overlapping subproblems. It can lead to more efficient algorithms, often reducing the time complexity from exponential to polynomial.

Parallelisation is a strategy that leverages modern hardware architectures to improve computational efficiency. By executing multiple parts of an algorithm simultaneously on different processors or cores, parallelisation can greatly reduce the time it takes to complete tasks. This is particularly effective for algorithms that can be broken down into independent subtasks or that operate on large datasets that can be divided and processed in parallel. However, achieving efficient parallelisation requires careful design to minimise communication overhead and ensure that the workload is evenly distributed across processing units.

Another strategy for improving computational efficiency is the use of **efficient data structures and algorithms**. Choosing the right data structure can significantly impact the performance of an application. For instance, using a hash table instead of a linear search can reduce the time complexity of a search operation from O(n) to O(1). Similarly, selecting an appropriate algorithm for a task, such as using quicksort instead of bubblesort for sorting arrays, can lead to substantial efficiency gains. Understanding the characteristics of different data structures and algorithms and applying them correctly is key to achieving high computational efficiency.

Case studies and real-world examples of algorithmic optimisation in complex application systems

Algorithmic optimisation plays a critical role in the performance of complex application systems across various domains. Here are a few case studies and real-world examples that illustrate the impact of algorithmic optimisation:

Google Search:
Google's search algorithm, which processes billions of searches daily, is a prime example of algorithmic optimisation. Over the years, Google has optimised its search algorithms to return relevant results in a fraction of a second. One of the key optimisations was the implementation of the MapReduce programming model, which allowed for the parallel processing of large datasets across distributed servers. This optimisation enabled Google to scale its search capabilities efficiently and maintain low response times even with the exponential growth of the web.

Facebook's Graph API:
Facebook's Graph API is used to query data about users, photos, pages, and other content on the social network. To handle the massive scale and the complex relationships between different pieces of data, Facebook has optimised its algorithms for data storage and retrieval. They use a combination of custom data stores like Cassandra for structured data and Haystack for photos, along with advanced caching mechanisms to ensure quick response times for API queries.

Netflix Recommendation Algorithm:
Netflix's recommendation system is a complex application that uses machine learning algorithms to suggest movies and TV shows to users. Optimising these algorithms is crucial for providing a good user experience and increasing user engagement. Netflix employs a variety of optimisation techniques, including dimensionality reduction to handle the vast number of user and item features, and model ensembling to combine the predictions of multiple algorithms for better accuracy.

Uber's Route Optimisation:
Uber's ride-hailing service relies on efficient route optimisation algorithms to match drivers with riders and to calculate the fastest routes in real-time. Uber has developed its own routing algorithm that considers various factors like traffic conditions, road closures, and the user's preferred route. By continuously optimising these algorithms, Uber can reduce wait times for riders and increase the number of trips that drivers can complete.

Amazon's Product Recommendations:
Amazon's product recommendation system, which suggests items to customers based on their browsing and purchasing history, is another example of algorithmic optimisation in action. Amazon uses collaborative filtering algorithms, which scale to millions of customers and products. They also optimise their algorithms to reduce latency, ensuring that recommendations are generated quickly as users browse the site.

Airbnb's Search Ranking:
Airbnb's search ranking system uses machine learning algorithms to sort listings based on relevance to the user's search query. optimising these algorithms involves balancing multiple factors, such as user preferences, listing quality, and host response rates. Airbnb has worked on optimising these algorithms to improve the search experience and increase booking rates.

In each of these cases, algorithmic optimisation has been essential for improving the performance, scalability, and user experience of complex application systems. These optimisations often involve a combination of choosing the right data structures, designing efficient algorithms, leveraging parallel processing, and applying advanced machine learning techniques.

▼ Supporting content F - Evidence-based optimisation techniques and justifications

Overview of the importance of evidence-based optimisation in application system performance tuning

Evidence-based optimisation is a critical approach in application system performance tuning because it ensures that any adjustments or enhancements made to the system are grounded in empirical data and proven methodologies. This **evidence-based approach** helps to mitigate the risks associated with making changes to complex systems without a clear understanding of their potential impact. By relying on data, such as system logs, performance metrics, and user feedback, system administrators and developers can identify specific bottlenecks and tailor their optimisation strategies to address those issues directly. This targeted approach not only improves the efficiency of the system but also helps in avoiding unnecessary changes that could potentially introduce new problems.

Furthermore, evidence-based optimisation allows for the creation of a **performance baseline** against which future enhancements can be measured. This baseline serves as a benchmark for the system's performance before any optimisation efforts are undertaken. With this benchmark in place, any subsequent optimisations can be evaluated objectively to determine their effectiveness. This process of measurement and evaluation is iterative, allowing for continuous improvement of the system over time. The use of evidence also facilitates better decision-making by providing a clear rationale for choosing one optimisation technique over another, based on their respective track records and the specific context of the application system.

In addition to these benefits, evidence-based optimisation fosters a **culture of accountability and transparency** within development and operations teams. When decisions are backed by data and results are measurable, it becomes easier to communicate the value of performance tuning efforts to stakeholders. This can lead to better resource allocation and investment in performance optimisation, as the return on investment can be clearly demonstrated. Moreover, evidence-based practices encourage the sharing of knowledge and best practices within the industry, as successful optimisation techniques can be documented and replicated across different application systems, contributing to the collective expertise in application system performance tuning.

Techniques for researching and identifying relevant optimisation strategies, such as literature reviews and case study analysis

Researching and identifying relevant optimisation strategies is a multifaceted process that often begins with **comprehensive literature reviews**. This involves delving into existing academic papers, industry reports, and technical documentation to understand the state-of-the-art optimisation techniques. By examining the methodologies, results, and conclusions of previous studies, researchers can gain insights into what has been effective in similar application system scenarios. Literature reviews help in identifying trends, common practices, and potential gaps in the current knowledge that could be explored further. They provide a foundation of understanding upon which new optimisation strategies can be built, ensuring that efforts are not duplicative but rather additive to the existing body of knowledge.

In addition to literature reviews, **case study analysis** is another powerful technique for identifying optimisation strategies. Case studies offer in-depth examinations of specific instances where optimisation techniques were applied to real-world application systems. By analysing these cases, researchers can observe the strategies in action, understand the context in which they were successful, and identify the challenges that were overcome. Case studies often provide rich qualitative data, including the decision-making processes, the evolution of the optimisation approach, and the long-term impacts on system performance. This granular information can be invaluable for deriving actionable insights and for developing optimisation strategies that are tailored to the unique characteristics of a given application system.

Moreover, both literature reviews and case study analysis should be complemented by a **critical evaluation of the evidence** presented. It is important to assess the quality of the research, the validity of the results, and the applicability of the findings to the current application system. This involves considering the research design, the rigor of the analysis, and the relevance of the context. By critically appraising the literature and case studies, researchers can discern which optimisation strategies are most likely to be effective and which may require adaptation or further investigation. This critical approach ensures that the optimisation strategies identified are not only evidence-based but also practical and likely to yield positive outcomes when applied to the target application system.

Best practices for justifying optimisation recommendations, such as citing research, benchmarks, and industry standards

Justifying optimisation recommendations with robust evidence is essential for gaining stakeholder buy-in and ensuring that the proposed changes are both effective and efficient. One best practice is to **cite relevant research** from reputable sources, such as academic journals, industry whitepapers, and technical reports. By referencing studies that have investigated similar optimisation techniques in comparable application systems, practitioners can demonstrate that their recommendations are grounded in empirical evidence and theoretical underpinnings. This not only lends credibility to the proposed optimisations but also provides a clear rationale for why certain strategies are expected to be successful.

Another key practice is to **support recommendations with benchmarks**, which involve measuring the performance of the application system before and after the application of optimisation techniques. Benchmarks serve as a quantitative basis for assessing the impact of optimisations, allowing practitioners to present concrete metrics such as improvements in response times, throughput, or resource utilisation. These metrics are particularly persuasive when they align with the specific performance goals of the application system. Moreover, benchmarks can be used to compare the performance of different optimisation strategies, helping to identify the most effective approaches. By presenting benchmark results, practitioners can offer a data-driven justification for their optimisation recommendations.

Industry standards and **best practices** also play a crucial role in justifying optimisation recommendations. These standards are often developed by professional organisations, expert panels, or through widespread adoption in the industry. They represent a consensus on effective practices and can be used to support the validity of the proposed optimisations. Referencing industry standards not only helps to align the recommendations with widely accepted practices but also demonstrates an awareness of the broader context in which the application system operates. Furthermore, adhering to industry standards can facilitate interoperability, compliance with regulations, and the adoption of optimisations that have been proven to work at scale. By anchoring optimisation recommendations in these standards, practitioners can provide a comprehensive justification that resonates with both technical and non-technical stakeholders.

Examples of well-justified optimisation recommendations for various application system performance scenarios

1. **Database optimisation for Improved Response Times**:

- *Scenario*: A web application with a relational database experiences slow response times during peak user loads.
- *Recommendation*: Implement indexing on frequently queried columns to speed up data retrieval.
- *Justification*: Research from database management experts (citing specific papers or articles) shows that proper indexing can significantly reduce query execution times. Benchmarks from industry-standard tools like Apache JMeter demonstrate a 50% reduction in response times after indexing. Adherence to best practices from the database vendor's documentation further supports the recommendation.

2. **Caching Strategy for Reduced Server Load**:

- *Scenario*: An e-commerce platform suffers from high server load, leading to slow page loads and occasional downtime during sales events.
- *Recommendation*: Introduce a distributed caching system like Redis or Memcached to store session data and frequently accessed read-only content.
- *Justification*: Case studies from similar e-commerce platforms show that implementing caching strategies can reduce database load by up to 70%. Industry benchmarks using load testing tools indicate a significant decrease in server response time with caching in place. The recommendation aligns with industry standards for high-traffic web applications.

3. **Code Profiling and Refactoring for Efficiency**:

- *Scenario*: A legacy enterprise application written in Java has performance bottlenecks due to inefficient code.
- *Recommendation*: Use code profiling tools like VisualVM to identify hot spots and refactor those sections of code for better performance.
- *Justification*: Literature on software maintenance and performance tuning emphasises the importance of profiling before optimisation. Benchmarks with the application under load show specific methods taking up the majority of CPU time. Refactoring these methods based on coding best practices and Java performance guidelines leads to a measurable improvement in application efficiency.

4. **Content Delivery Network (CDN) for Faster Content Distribution**:

- *Scenario*: A global news website has a significant portion of its user base experiencing slow loading times due to the distance from the server.
- *Recommendation*: Implement a CDN to cache static content at edge locations closer to users.
- *Justification*: Industry reports and case studies from other global websites illustrate the effectiveness of CDNs in reducing latency for users across different geographical locations. Benchmarks using network performance tools show a 30% reduction in content load times after CDN integration. The recommendation follows web performance optimisation standards that prioritise the use of CDNs for distributing static assets.

5. **Concurrency optimisation in a High-Traffic Web Service**:
    - *Scenario*: A RESTful API service built on Node.js struggles with concurrency issues under heavy load.
    - *Recommendation*: Apply asynchronous programming patterns and use a message queue to handle requests.
    - *Justification*: Research on Node.js performance under high concurrency (citing specific Node.js performance articles) suggests that asynchronous coding and message queues can significantly improve throughput. Load testing with tools like Artillery.io demonstrates a 200% increase in requests served per second after implementing these changes. The recommendation is in line with Node.js best practices for building scalable network applications.

In each of these examples, the optimisation recommendations are supported by a combination of research, benchmarks, and adherence to industry standards, providing a well-rounded justification for the proposed changes.

This activity is complete when you have

- Engaged with the AI tutor in the Shopify case study and participated in class discussion to share your experiences and learn from others.
- Documented your analysis and recommendations for the Shopify case study in a short report (1-2 pages, or a copy of the chat transcript), which will form part of your **portfolio (https://lms.griffith.edu.au/courses/24045/pages/building-a-portfolio-for-assignment-2)** .
- Considered the type of UX experience for the scenario in your **application system design report (https://lms.griffith.edu.au/courses/24045/assignments/93487)** .