Mastering System Design

Section 8: Performance - Concepts, Tools & Techniques

Performance

Performance- Section Agenda

1. Introduction to System Performance
2. Caching for Speed Optimization
3. Messaging & Queues for Decoupling
4. Concurrency & Parallelism
5. Database Performance Optimization Techniques
6. Summary and Recap - Performance

Introduction to System Performance

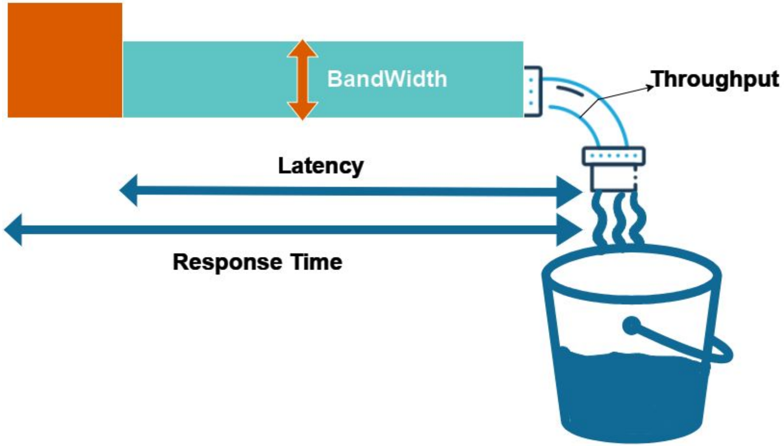
Performance

What is Performance in System Design?

- Performance = How efficiently a system meets its functional requirements under load
- Key dimensions:
  - Speed: Time to respond
  - Capacity: Amount of work handled
  - Efficiency: Resource usage under load
- Performance is not a single metric — it's a multi-dimensional goal

Latency vs. Throughput

- Latency
  - Time taken to process one request
  - Measured in ms or s
  - Affects responsiveness
- Throughput
  - Number of requests processed per second
  - Measured in RPS or TPS
  - Affects scalability
- Low latency ≠ High throughput and vice versa
- Both must be balanced based on use case



Scalability vs. Responsiveness

- Scalability: Ability to handle increased load without performance degradation
  - Horizontal vs. Vertical scaling
- Responsiveness: System's ability to respond quickly
  - Tightly linked to latency
- Good design should ensure responsiveness at scale

Measuring Performance

- Performance must be measurable & trackable:
    - SLA (Service Level Agreement): External contractual guarantee
    - SLO (Service Level Objective): Internal target
    - SLI (Service Level Indicator): Actual metric value
- Example:
    - SLA: 99.9% uptime
    - SLO: 95% of requests < 300ms
    - SLI: Actual measurement (e.g., 93% of requests < 300ms)

Understanding Percentiles

- Mean/average ≠ useful in tail-latency-sensitive systems
- Percentiles provide better insight:
  - P50: Median
  - P95: 95% of requests faster than this
  - P99: Tail latency — critical for user experience
- Track tail latencies for real-world performance insights

Why Performance Matters in Modern Applications

- Users expect instantaneous responses (esp. mobile/web)
- Poor performance leads to:
  - Drop-offs & bounce rates
  - Loss in revenue
  - System instability under load
- Performance is a feature, not an afterthought
- Impacts user experience, cost, and reputation

Performance Testing Overview

- Types of performance testing:
    - Load Testing: Normal load conditions
    - Stress Testing: Beyond normal limits
    - Spike Testing: Sudden large load
    - Endurance Testing: Over extended time
- Goals:
    - Identify bottlenecks
    - Ensure reliability under real-world scenarios

Introduction to Performance Monitoring

- Monitoring ≠ Testing — it's continuous
- Key tools:
  - APM (e.g., New Relic, Datadog)
  - Logs & Metrics (e.g., ELK, Prometheus + Grafana)
- Track:
  - Latency & throughput
  - Error rates
  - Resource usage (CPU, memory, DB queries)

Interview Questions – System Performance

1. What is the difference between latency and throughput?
2. How do SLAs, SLOs, and SLIs differ? Provide real-world examples.
3. Why are percentiles (like P95, P99) important in performance monitoring?
4. What strategies would you use to identify a system's performance bottleneck?
5. How would you ensure responsiveness in a highly scalable system?
6. What tools or techniques have you used for performance testing and monitoring?
7. How would you design a system to handle sudden traffic spikes?
8. Explain the trade-offs between performance and cost in cloud environments.

Summary and Key Takeaways

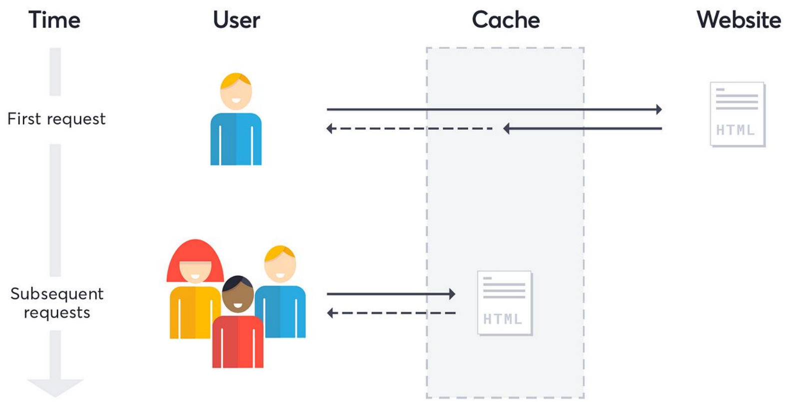
- Performance = speed, efficiency & scalability under load
- Understand latency vs. throughput
- Set & measure SLAs, SLOs, and percentiles
- Performance must be tested, monitored, and prioritized
- It's essential for modern, scalable, and user-friendly systems
- What’s next:
    - Caching for Speed Optimization

Caching for Speed Optimization

Performance

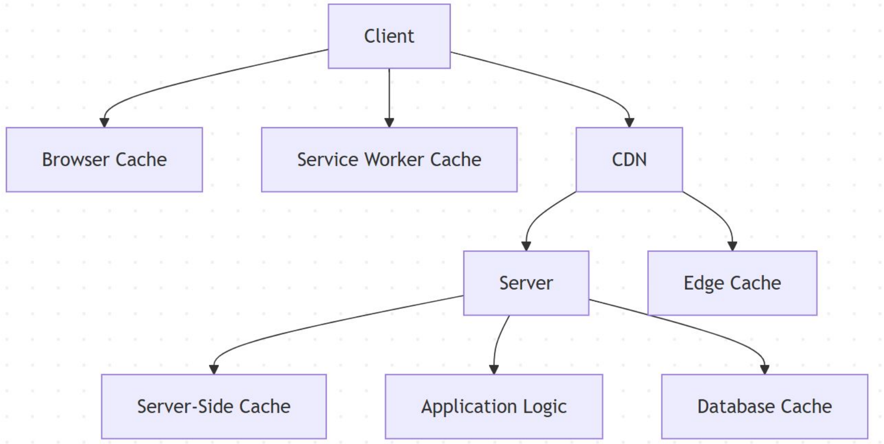
Why Caching Matters

- Reduces latency by avoiding expensive recomputation or data retrieval
- Eases load on backend services and databases
- Improves user experience and system scalability
- Critical in low-latency, high-throughput architectures



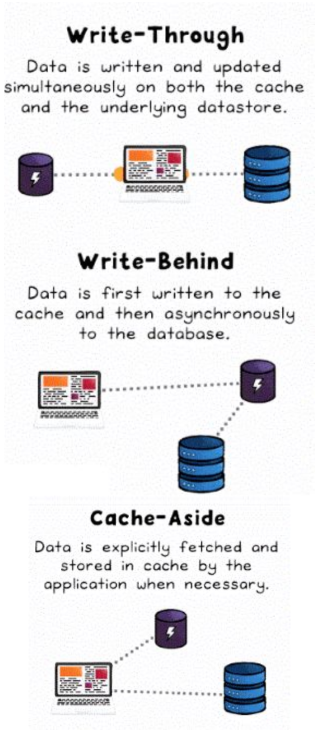
Types of Caching

- Client-side: Browser memory (e.g., localStorage, service workers)
- Server-side: Application-level memory or in-memory caches like Redis
- CDN caching: Static content cached close to users (e.g., Cloudflare, Akamai)
- Database caching: Result-set caching, materialized views



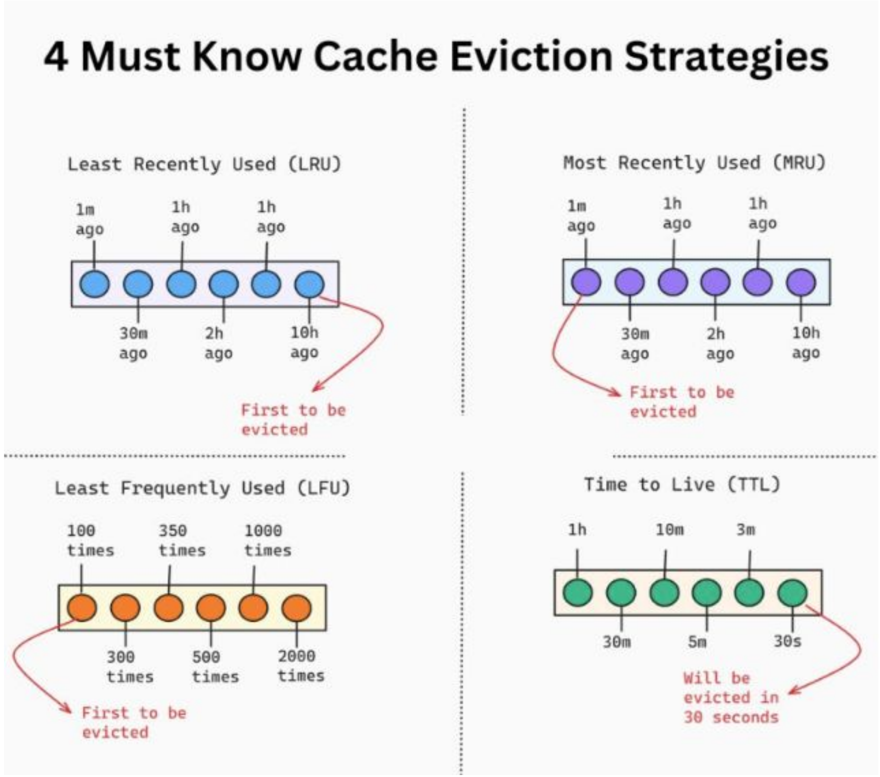
Caching Strategies

- Write-through: Write to cache and DB simultaneously
- Write-back (write-behind): Write to cache, DB is updated asynchronously
- Lazy loading (Cache-aside): Cache is populated only on demand
- Explicit/manual caching: Developer decides when to cache or evict



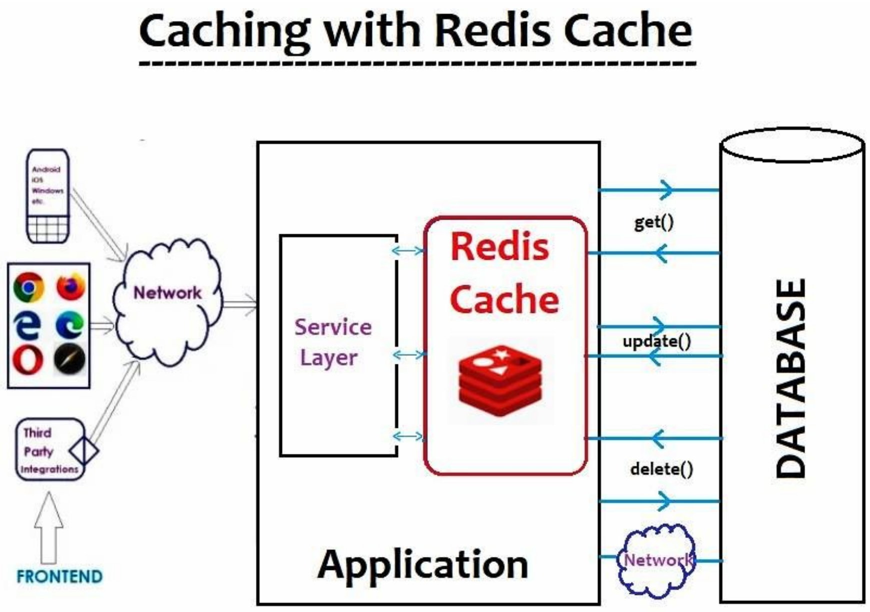
Cache Eviction Policies

- LRU (Least Recently Used): Remove least recently accessed item
- LFU (Least Frequently Used): Remove least used item by count
- FIFO (First In, First Out): Remove oldest item added
- TTL (Time To Live): Automatically expire items after a fixed time duration



Redis Overview

- In-memory key-value store for ultra-fast access
- Supports TTLs (time-to-live), pub/sub, persistence
- Used for caching, queues, sessions, leaderboards, etc.
- Open-source, widely adopted



Real-World Caching Examples

- CDN: Cache static assets (images, JS, CSS)
- Product page data: Cache popular catalog queries
- User sessions: Stored in Redis for fast access
- Search results: Frequently repeated queries cached
- API response caching: Microservices avoid recomputation

Interview Questions – Caching in System Design

- Core Conceptual Questions:
  - What is caching and why is it important in system design?
  - Explain different types of caching and where they are used.
  - What are write-through vs. write-back caching strategies?
  - What is lazy loading (cache-aside pattern) and when would you use it?
- Scenario-Based Questions:
  - How would you use caching to optimize a product details page?
  - What eviction strategy would you choose for a memory-limited system?
  - How would you keep cache and database in sync?
  - What are the potential downsides or risks of aggressive caching?
- Practical Implementation:
  - How would you implement Redis caching in a web application?
  - How can you prevent cache stampedes or thundering herd problems?
  - What tools can be used for distributed caching?

Summary & What’s Next

- Caching improves response time, reduces load, and scales better
- Choose cache type, strategy, and eviction wisely
- Redis is a powerful tool in the caching toolkit
- Caching is foundational in high-performance systems
- What's Next:
  - Messaging & Queues for Decoupling

Messaging & Queues for Decoupling

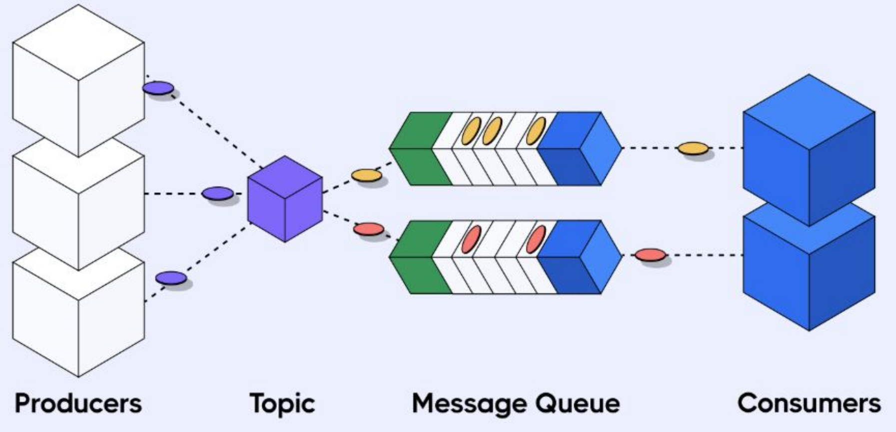
Performance

Why Use Asynchronous Messaging?

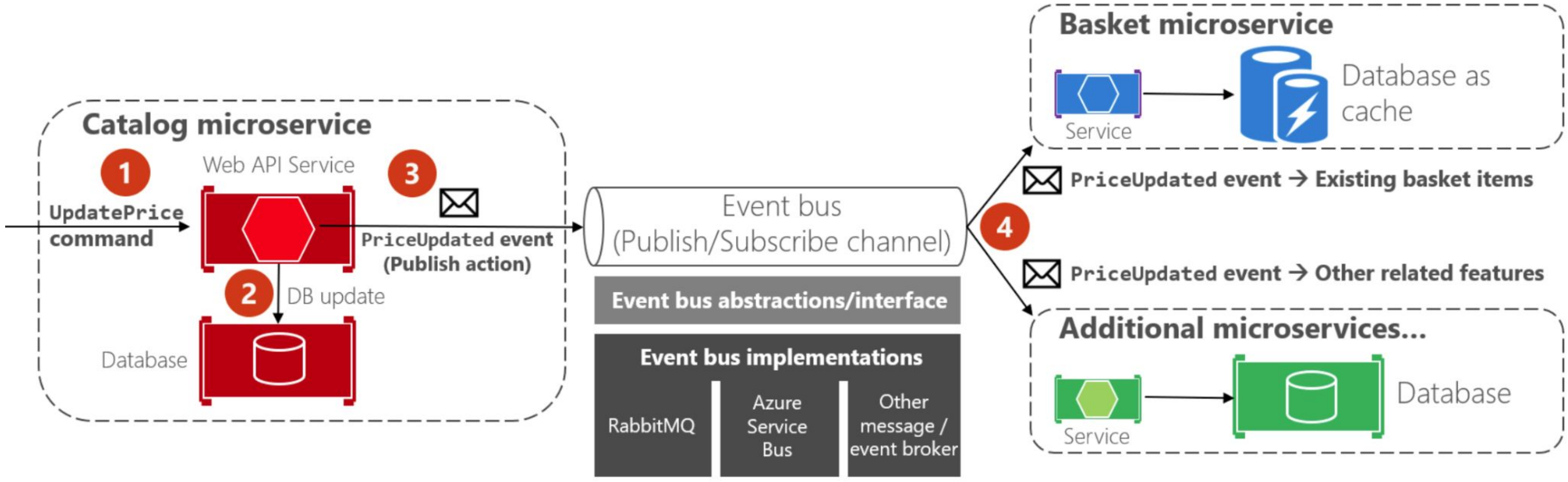
- Loose Coupling: Decouple producers from consumers
- Improved Performance: Producers don’t block waiting for consumers
- Scalability: Consumers can scale independently
- Resilience: Message durability adds fault tolerance
- Flexibility: Easily add new consumers without changing producers

Key Concepts of Messaging Systems

- Message: A data packet sent from producer to consumer
- Producer: Sends the message
- Consumer: Receives and processes the message
- Broker/Queue: Stores and delivers messages
- Topic/Queue: Logical channel for message delivery
- Ack: Acknowledgement of successful processing



Visualizing a Decoupled Architecture



When to Use Queues in Architecture

- When workloads are bursty
- When you need decoupling between services
- For background jobs (e.g. email, processing, exports)
- For rate-limited or expensive operations
- To buffer spikes in traffic

Popular Message Brokers: RabbitMQ vs Kafka

- 🐇 RabbitMQ – Traditional Message Broker
  - Built on AMQP, designed for reliable message delivery
  - Follows a push-based model: messages are pushed to consumers
  - Supports acknowledgements, retries, and dead-letter queues
  - Great for task distribution, background jobs, and real-time notifications
  - Focuses on routing flexibility (e.g., direct, topic, fanout exchanges)
  - Messages are removed after consumption
- ⚡ Kafka – Distributed Event Streaming Platform
  - Built for high-throughput, durable, distributed event logs
  - Uses a pull-based model: consumers read at their own pace
  - Stores messages in partitioned logs; supports message replay
  - Ideal for event sourcing, real-time analytics, and stream processing
  - Highly scalable and fault-tolerant
  - Messages are retained for configurable durations (even after consumption)

Delivery Guarantees

1. At-least-once (default in many systems)
    a. Message is retried until acknowledged
    b. May lead to duplicates
    c. Consumers must be idempotent
2. At-most-once
    a. Message sent only once
    b. No retries → may result in message loss
3. Exactly-once
    a. Guaranteed single delivery without duplicates
    b. Complex and more resource-heavy
    c. Kafka supports it under specific constraints

Common Use Cases of Messaging Queues

- Order Processing
  - Decouple frontend from inventory, payments, shipping
- Logging & Monitoring
  - Centralized log processing (e.g., ELK, Kafka)
- Rate Limiting / Traffic Shaping
  - Queue incoming requests and process at a safe pace
- Email / SMS Notification Systems
  - Queue user notifications for async sending
- ETL Pipelines / Stream Processing
  - Kafka used for real-time data transformation

Best Practices for Using Messaging Queues

- Use idempotent consumers
- Implement dead-letter queues (DLQs)
- Monitor queue length & processing time
- Handle retries & failures gracefully
- Choose delivery semantics based on need
- Secure your message brokers (auth, encryption)

Interview Questions – Messaging & Queues for Decoupling

- Why would you use asynchronous messaging in a system?
- What are the differences between RabbitMQ and Kafka? When would you use one over the other?
- Explain the different message delivery guarantees: At-least-once vs. Exactly-once vs. At-most-once – what are they and where do you apply them?
- How do queues help improve system scalability and fault tolerance?
- What problems might arise in a queue-based system under heavy load? How would you mitigate them?
- How would you design an order processing system using a message queue?
- How would you ensure idempotency in the consumers?

Summary and Key Takeaways

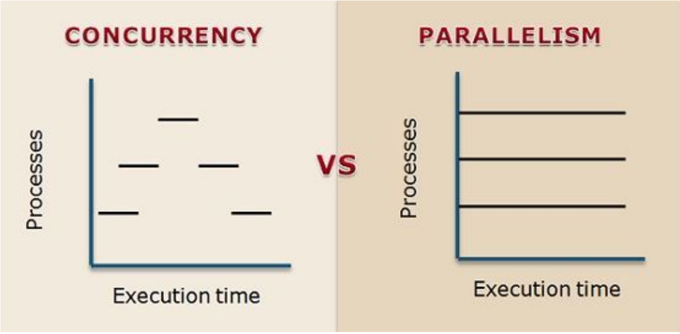
- Asynchronous messaging helps build scalable, decoupled systems
- Use queues when real-time response isn't necessary
- RabbitMQ is great for traditional queues
- Kafka excels at high-throughput event streaming
- Understand your delivery guarantees for each use case
- What’s next:
  - Concurrency & Parallelism

Concurrency & Parallelism in System Design

Performance

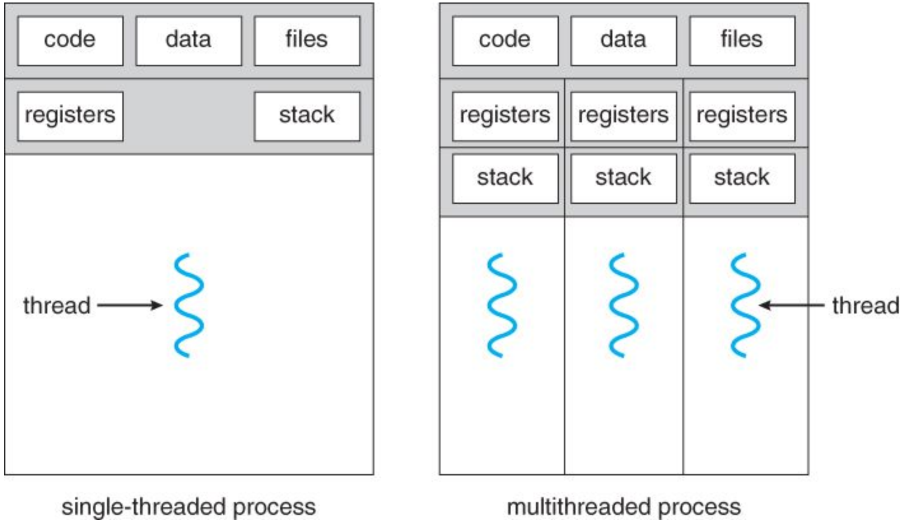
Concurrency and Parallelism

- What is Concurrency
  - **Definition**: Concurrency is when multiple tasks start, run, and complete in overlapping time periods – not necessarily simultaneously.
  - **Key Point**:
    - It's about managing multiple tasks efficiently, especially on a single CPU core.
    - It all about Task management
    - Can happen on single-core
    - Goal is Responsiveness
  - **Example**: Web server handling multiple incoming HTTP requests using asynchronous I/O.
- What is Parallelism?
  - **Definition**: Parallelism is when multiple tasks are executed simultaneously, typically on multiple CPU cores.
  - **Key Point**:
    - It's about actual simultaneous execution to speed up performance.
    - It's all about Task execution
    - Needs multi-core for true parallelism
    - Goal is Speed/throughput
  - Example: Matrix computation where different parts are calculated in parallel on multiple threads.



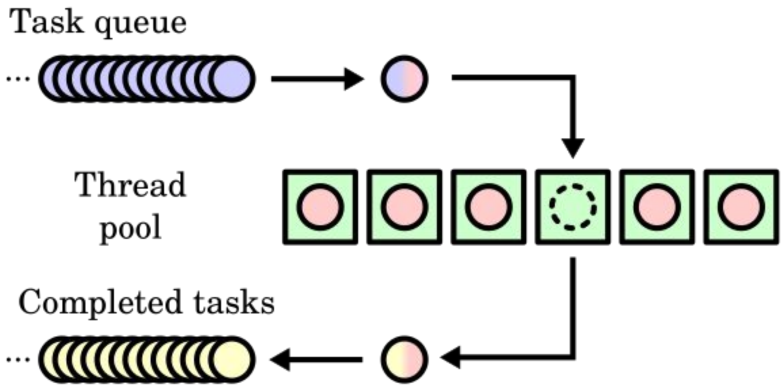
Processes vs. Threads

- Processes:
  - Have their own memory space
  - Heavier to create and switch
  - Isolated and safer
- Threads:
  - Share memory within a process
  - Lightweight and faster
  - Can lead to complex bugs (e.g., race conditions)



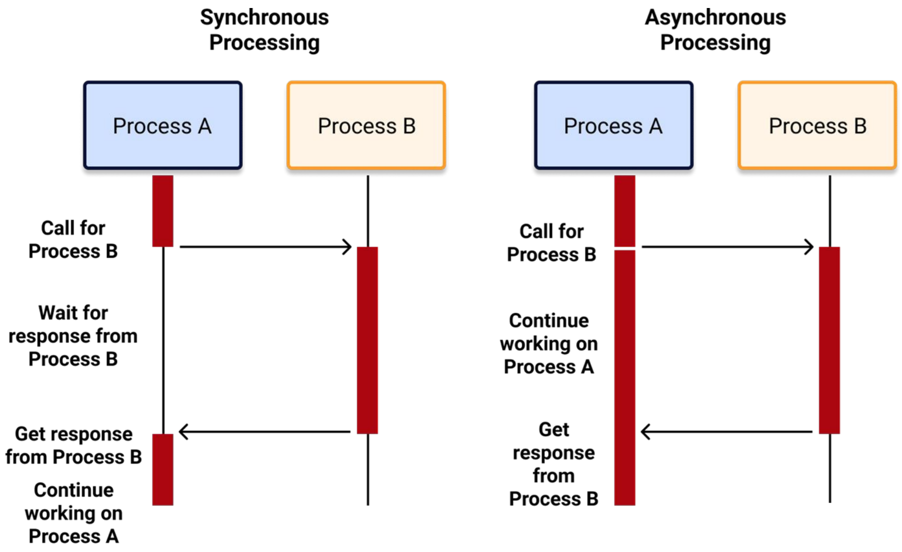
Thread Pools & Worker Models

- Thread Pools:
  - Pre-created threads reused for multiple tasks
  - Avoid overhead of creating/destroying threads
- Worker Models:
  - Tasks are distributed to idle workers from a shared queue
  - Improves scalability and CPU utilization
- Example:
  - ASP.NET Core uses thread pool to handle requests efficiently.



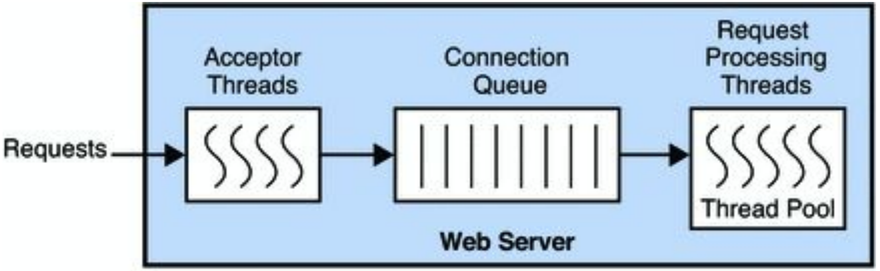
Asynchronous Processing

- Why Async?
  - Avoid blocking threads on I/O
  - Improve throughput
- Techniques:
  - Async/Await (C#, JS)
  - Promises, Futures
  - Message Queues (e.g., RabbitMQ, Kafka) for background work



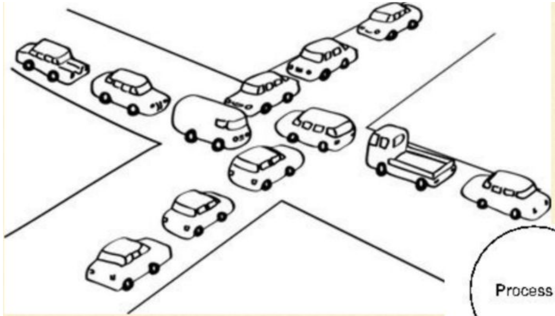
Concurrency in Web Servers

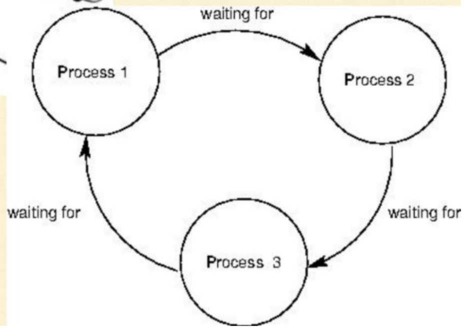
- Traditional Servers (e.g., Apache):
  - Spawn new thread/process per request
  - Not scalable for high traffic
- Modern Servers (e.g., Node.js, ASP.NET Core, Nginx):
  - Use async/non-blocking I/O
  - Event-loop or thread-pool models for scalability



Common Pitfalls

- Race Conditions:
    - Multiple threads access and modify shared data concurrently
    - Causes unexpected behavior or corruption
- Deadlocks:
    - Two or more threads wait for each other to release resources
    - System gets stuck indefinitely





Best Practices and Real World Examples

- Prefer async/non-blocking I/O for I/O-bound tasks
- Use thread pools instead of raw threads
- Always synchronize access to shared data (locks, mutexes, etc.)
- Detect and avoid deadlocks (lock ordering, timeout strategies)
- Real-World Examples
  - Web Server Handling 1000s of Requests: Uses thread pool + async I/O
  - Background Job Processing (e.g., Email Queue): Worker model with RabbitMQ
  - Parallel Image Rendering: Each frame rendered on a different core

Interview Questions – Concurrency & Parallelism

- Conceptual Questions:
  - What is the difference between concurrency and parallelism?
  - How do threads differ from processes?
  - What is a thread pool, and why is it preferred over creating new threads?
- Practical Scenarios:
  - How would you design a web server to handle thousands of concurrent requests?
  - Describe how you would implement background job processing in a scalable system.
  - How would you debug and resolve a deadlock in a multithreaded application?
- Pitfall Awareness:
  - What is a race condition, and how can you prevent it?
  - How do you ensure thread-safe operations in a shared-memory environment?
- Bonus (Advanced):
  - How does the event loop work in Node.js or similar environments?
  - What's the difference between parallelism using threads vs. async I/O?

Summary and Key Takeaways

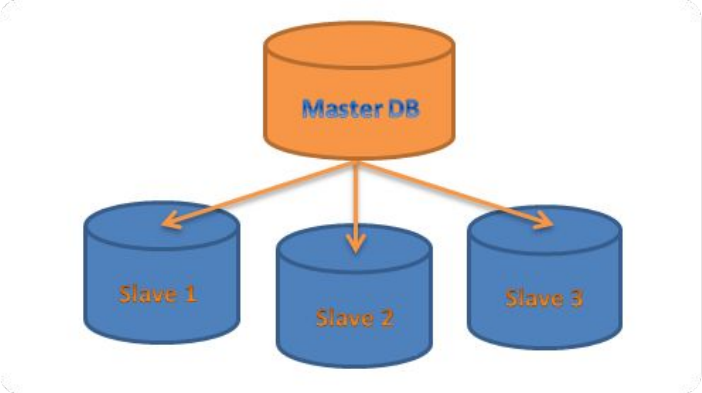
- Concurrency ≠ Parallelism – both are tools for performance and scalability
- Threads are lighter than processes but need careful handling
- Web servers rely on thread pools and async models
- Watch out for race conditions and deadlocks!
- What’s next:
    - Database Performance Optimization Techniques

Database Performance Optimization Techniques

Performance

Replication Recap

- What is Replication?
  - Definition: Replication is the process of copying and maintaining database objects in multiple databases.
- Why Replicate?
  - High Availability: Ensure data availability in case of failure.
  - Load Balancing: Distribute read traffic across multiple servers.
  - Disaster Recovery: Maintain copies of data across different locations.
- Types of Replication:
  - Master-Slave Replication: One primary database, multiple replicas for reads.
  - Master-Master Replication: Multiple primary databases, often used for writes and redundancy.



Sharding & Partitioning Strategies Recap

- Sharding:
  - Definition: Splitting large datasets into smaller, more manageable pieces (shards).
  - Goal: Distribute data across multiple servers for better performance and scalability.
  - Example: User data across multiple shards based on geographical region.
- Partitioning:
  - Definition: Dividing data within a single database into separate tables or partitions.
  - Types of Partitioning:
    - Range Partitioning: Split data by a range (e.g., by date).
    - Hash Partitioning: Split data by hashing on a key.

Vertical Partition

Original Table

CustomerID	Name	Email	Total Purchases
1	ABC	abc@gmail.com	2000
2	PQR	pqr@gmail.com	3000
3	XYZ	xyz@gmail.com	4000
4	LMN	lmn@gmail.com	5000
5	JKL	jkl@gmail.com	6000

Partition 1

CustomerID	Name	Email
1	ABC	abc@gmail.com
2	PQR	pqr@gmail.com
3	XYZ	xyz@gmail.com
4	LMN	lmn@gmail.com
5	JKL	jkl@gmail.com

Partition 2

CustomerID	Total Purchases
1	2000
2	3000
3	4000
4	5000
5	6000

Horizontal Partition / Database Sharding

Original Table

CustomerID	Name	Email	Total Purchases
1	ABC	abc@gmail.com	2000
2	PQR	pqr@gmail.com	3000
3	XYZ	xyz@gmail.com	4000
4	LMN	lmn@gmail.com	5000
5	JKL	jkl@gmail.com	6000

Shard 1

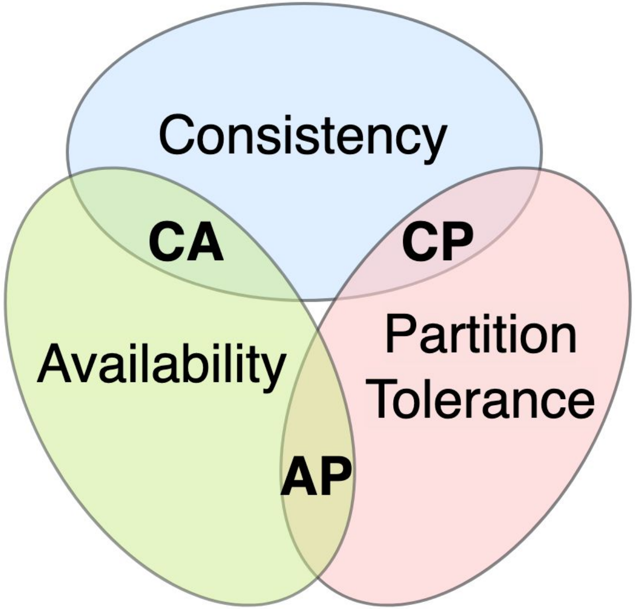
1	ABC	abc@gmail.com	2000
2	PQR	pqr@gmail.com	3000
3	XYZ	xyz@gmail.com	4000

Shard 2

CustomerID	Name	Email	Total Purchases
4	LMN	lmn@gmail.com	5000
5	JKL	jkl@gmail.com	6000

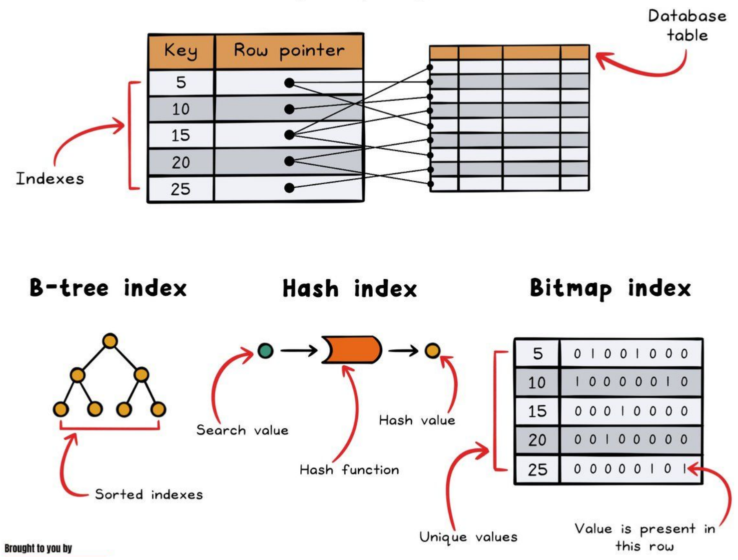
CAP Theorem Recap (Performance Focus)

- What is CAP Theorem?
  - Consistency: All nodes see the same data at the same time.
  - Availability: Every request gets a response, regardless of node state.
  - Partition Tolerance: System continues to operate despite network failures.
- Performance Focus/Trade-Offs for Performance:
  - In highly distributed systems, you may need to compromise on consistency or availability for performance.
  - Often systems focus on high availability and partition tolerance over consistency for better performance at scale.



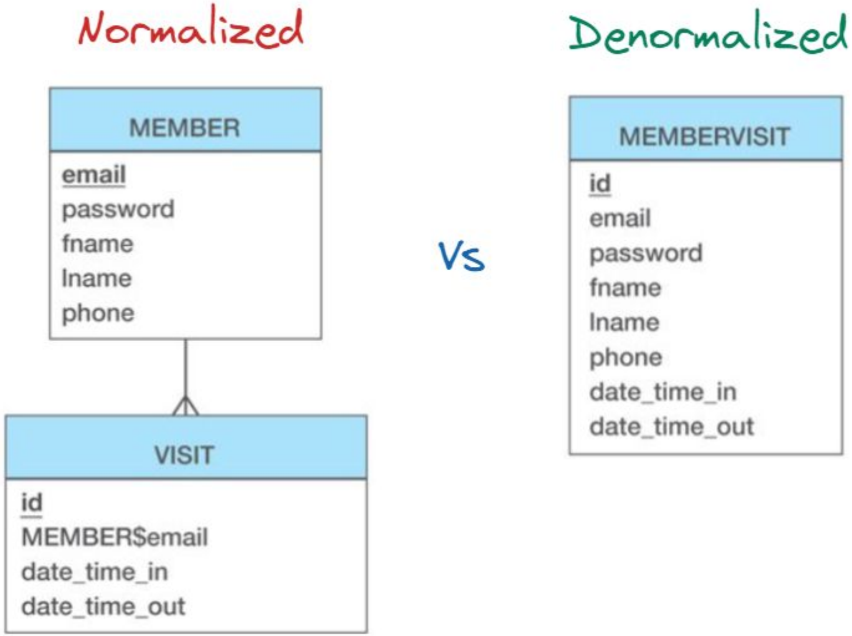
Indexes: Types & Use Cases

- What are Indexes?
  - Definition: Data structures that improve query performance by reducing the amount of data scanned.
- Types of Indexes:
  - B-Tree Indexes: Common for exact match and range queries.
  - Hash Indexes: Best for equality comparisons.
  - Full-Text Indexes: Used for searching large textual data.
  - Bitmap Indexes: Suitable for low cardinality columns (e.g., gender).
- When to Use Indexes:
  - Read-heavy Operations: Indexes speed up query performance.
  - Write-heavy Systems: Be cautious, as indexes can slow down inserts and updates.



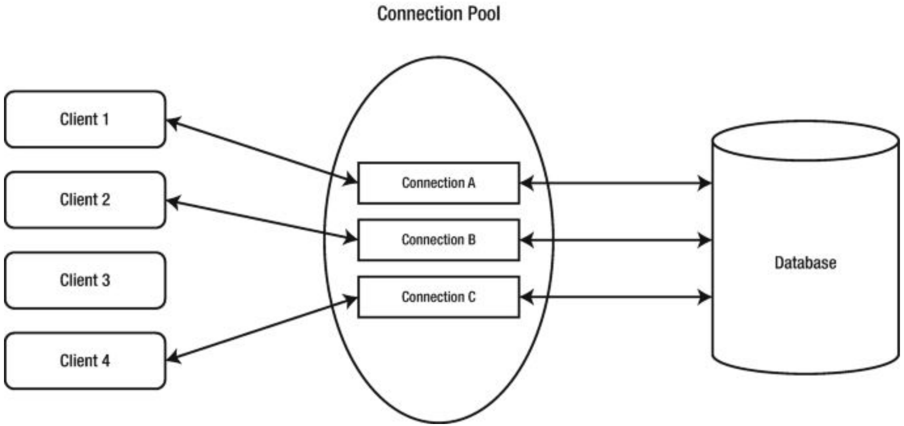
Normalization vs. Denormalization

- Normalization:
  - Goal: Reduce data redundancy by organizing data into tables.
  - Benefits: Minimizes storage costs, eliminates anomalies.
  - Drawback: Can lead to complex joins and slower read performance.
- Denormalization:
  - Goal: Introduce redundancy to reduce join operations and speed up reads.
  - Benefits: Faster read performance.
  - Drawback: Increased storage and potential data anomalies.
- When to Use Each:
  - Normalization: For transactional systems (OLTP).
  - Denormalization: For reporting systems or read-heavy workloads.



Additional Techniques - Connection Pooling

- What is Connection Pooling?
  - Definition: A technique used to manage database connections efficiently by reusing established connections instead of creating new ones.
- Why Use It?
  - Reduces overhead caused by frequent connection creation and teardown.
  - Helps handle a large number of concurrent connections effectively.

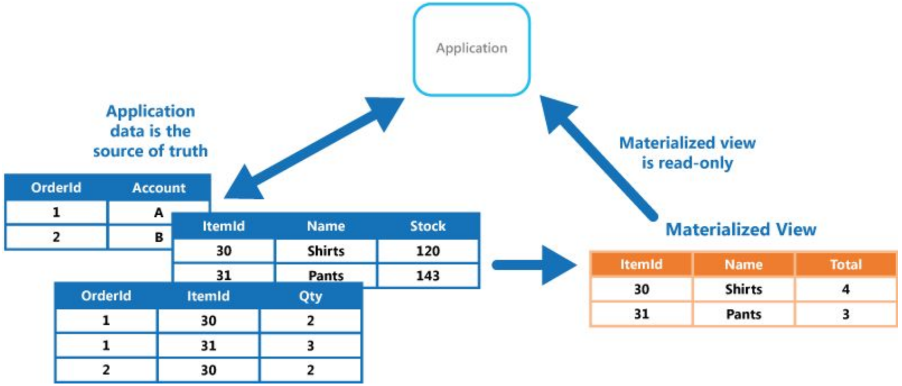


Additional Techniques - Query Optimization

- Definition: The process of improving the performance of SQL queries.
- Techniques:
    - Use of Indexes: Leverage indexes to speed up search operations.
    - Avoiding N+1 Queries: Reduce unnecessary database hits by using joins or batching queries.
    - Using Proper Joins: Minimize complex joins when possible.

Additional Techniques - Materialized Views

- Definition: A precomputed query result stored as a table.
- Benefits:
  - Speeds up query performance by avoiding real-time computation.
  - Useful in reporting and data warehousing.
- Use Cases:
  - Data aggregation or summary data that doesn't change frequently.
  - Reporting systems where fast retrieval is critical.



Additional Techniques - Batching & Pagination

- Batching:
  - Definition: Sending multiple operations in a single request or transaction to reduce overhead.
  - Use Case: Bulk inserts or updates.
- Pagination:
  - Definition: Breaking large sets of data into smaller chunks for efficient retrieval.
  - Prevents large queries that could lead to timeouts or memory issues.
  - Ensures responsive UI by fetching data incrementally.

Interview Questions - Database Performance Optimization Techniques

1. What is database replication, and why is it important for performance?
2. Can you explain the difference between sharding and partitioning in databases?
3. How does the CAP theorem impact database performance decisions?
4. What are some common types of indexes, and when would you use them?
5. What is the difference between normalization and denormalization, and how do they affect performance?
6. How does connection pooling improve database performance?
7. What is query optimization, and how would you approach optimizing a slow query?
8. What are materialized views, and how do they enhance performance?
9. How would you handle large datasets in your application?
10. What are the trade-offs between consistency and performance in distributed databases?

Summary and Key Takeaways

- Replication & Sharding: Key techniques for scaling databases across multiple servers.
- CAP Theorem: Understanding trade-offs between consistency, availability, and partition tolerance.
- Indexes: Crucial for improving query performance, but need to be managed properly.
- Normalization vs. Denormalization: Choose based on workload (transactions vs. reporting).
- Other Techniques: Connection pooling, query optimization, materialized views, and batching for efficient database operations.
- What's next:
  - Summary and Recap - Performance

Section Summary - Performance

- Introduction to System Performance
- Caching for Speed Optimization
- Messaging & Queues for Decoupling
- Concurrency & Parallelism
- Database Performance Optimization Techniques
- What’s in next Section:
  - Reliability – Availability, Failover & Recovery