Deep Dive into Concurrency for .NET Developers

This document provides a comprehensive overview of concurrency in both C# and SQL, designed to equip you with the knowledge needed for your presentation and to answer follow-up questions.

Part 1: C# Concurrency - Synchronization and Message Passing

Concurrency in C# allows multiple parts of your program to execute independently, often simultaneously. This is crucial for building responsive applications, utilizing modern multi-core processors, and handling multiple requests efficiently. However, it introduces challenges like race conditions, deadlocks, and data inconsistency, which require careful synchronization.

1. Signaling Mechanisms

Signaling mechanisms are fundamental tools to coordinate the execution of multiple threads, ensuring they access shared resources safely or proceed in a specific order.

a. `Monitor` (`Monitor.Wait`, `Monitor.Pulse`, `Monitor.PulseAll`)

The `Monitor` class (and the `lock` keyword, which is syntactic sugar for `Monitor.Enter` and `Monitor.Exit`) provides a powerful way to achieve mutual exclusion and thread synchronization.

- **`lock (object)`:** Ensures that only one thread can execute a block of code at a time. The `object` used for locking must be a reference type and should be a private, static, read-only field to prevent other parts of the code from locking on the same object.
- **`Monitor.Wait(object)`:** Releases the lock on the specified object and blocks the current thread until it's reacquired and the thread is notified by a `Pulse` or `PulseAll` call from another thread. A thread might also wake up spuriously, so `Wait` calls should always be inside a loop checking the condition.

- **`Monitor.Pulse(object)`:** Notifies a single waiting thread (if any) that the state of the locked object has changed. The notified thread will attempt to reacquire the lock and resume execution.
- **`Monitor.PulseAll(object)`:** Notifies all waiting threads on the specified object. All notified threads will compete to reacquire the lock.

Use Case: Simple Producer-Consumer with `Monitor`

The producer-consumer problem is a classic concurrency problem where one or more "producer" threads generate data and one or more "consumer" threads process that data. A shared buffer is used, and synchronization is needed to prevent producers from adding data to a full buffer and consumers from taking data from an empty buffer.

```
using System;
using System.Collections.Generic;
using System.Threading;

public class MonitorExample
{
    private static readonly Queue<int> buffer = new Queue<int>();
    private static readonly int capacity = 5;
    private static readonly object lockObject = new object();

    public static void Run()
    {
        Thread producerThread = new Thread(Producer);
        Thread consumerThread = new Thread(Consumer);

        producerThread.Start();
        consumerThread.Start();

        producerThread.Join();
        consumerThread.Join();

        Console.WriteLine("Producer and Consumer finished.");
    }

    private static void Producer()
    {
        for (int i = 0; i < 10; i++)
        {
            lock (lockObject)
            {
                // Wait if buffer is full
                while (buffer.Count == capacity)
                {
                    Console.WriteLine("Producer: Buffer full, 
waiting...");
                    Monitor.Wait(lockObject); // Releases lock, 
waits for Pulse
                }

                buffer.Enqueue(i);
                Console.WriteLine($"Producer: Produced {i}. Buffer 
size: {buffer.Count}");
                Monitor.Pulse(lockObject); // Notifies a waiting 
consumer
            }
            Thread.Sleep(100); // Simulate work
        }
    }

    private static void Consumer()
    {
        for (int i = 0; i < 10; i++)
        {
            lock (lockObject)
            {
                // Wait if buffer is empty
                while (buffer.Count == 0)
                {
                    Console.WriteLine("Consumer: Buffer empty, 
waiting...");
                    Monitor.Wait(lockObject); // Releases lock, 
waits for Pulse
                }

                int item = buffer.Dequeue();
                Console.WriteLine($"Consumer: Consumed {item}. 
Buffer size: {buffer.Count}");
                Monitor.Pulse(lockObject); // Notifies a waiting 
producer
            }
            Thread.Sleep(500); // Simulate work
        }
    }
}
```

b. AutoResetEvent

AutoResetEvent is a synchronization primitive that allows threads to signal each other. It automatically resets to a non-signaled state after releasing a single waiting thread.

- **Set():** Sets the event to a signaled state, releasing one waiting thread.
- **Reset():** Sets the event to a non-signaled state.
- **WaitOne():** Blocks the current thread until the event is in a signaled state. Once signaled, it automatically resets to non-signaled.

Use Case: Simple Handshake

```
using System;
using System.Threading;

public class AutoResetEventExample
{
    private static AutoResetEvent signal1 = new
AutoResetEvent(false); // Initially non-signaled
    private static AutoResetEvent signal2 = new
AutoResetEvent(false); // Initially non-signaled

    public static void Run()
    {
        Thread threadA = new Thread(ThreadA);
        Thread threadB = new Thread(ThreadB);

        threadA.Start();
        threadB.Start();

        threadA.Join();
        threadB.Join();

        Console.WriteLine("AutoResetEvent Example Finished.");
    }
    private static void ThreadA()
    {
        Console.WriteLine("Thread A: Waiting for signal from Thread
B...");
        signal2.WaitOne(); // Wait for signal from Thread B
        Console.WriteLine("Thread A: Received signal from Thread B.
Sending signal to Thread B...");
        signal1.Set(); // Signal Thread B
    }

    private static void ThreadB()
    {
        Console.WriteLine("Thread B: Sending signal to Thread
A...");
        signal2.Set(); // Signal Thread A
        Console.WriteLine("Thread B: Waiting for signal from Thread
A...");
        signal1.WaitOne(); // Wait for signal from Thread A
        Console.WriteLine("Thread B: Received signal from Thread
A.");
    }
}
```

c. *ManualResetEvent*

`ManualResetEvent` is similar to `AutoResetEvent`, but it requires manual resetting. Once signaled, it remains signaled until `Reset()` is explicitly called, allowing multiple waiting threads to be released.

- **`Set()`:** Sets the event to a signaled state, releasing all waiting threads.
- **`Reset()`:** Sets the event to a non-signaled state.
- **`WaitOne()`:** Blocks the current thread until the event is in a signaled state.

Use Case: Notifying Multiple Threads

```
using System;
using System.Threading;

public class ManualResetEventExample
{
    private static ManualResetEvent startSignal = new
ManualResetEvent(false); // Initially non-signaled

    public static void Run()
    {
        for (int i = 0; i < 5; i++)
        {
            Thread workerThread = new Thread(Worker);
            workerThread.Name = $"Worker {i}";
            workerThread.Start();
        }

        Console.WriteLine("Main: All worker threads created. Press
any key to start them...");
        Console.ReadKey();
        startSignal.Set(); // Signal all waiting worker threads

        // In a real app, you'd join or manage these threads.
        // For demonstration, we'll just let them finish.
        Thread.Sleep(1000);
        Console.WriteLine("ManualResetEvent Example Finished.");
    }

    private static void Worker()
    {
        Console.WriteLine($"{Thread.CurrentThread.Name}: Waiting for
start signal...");
        startSignal.WaitOne(); // Wait until startSignal is set
        Console.WriteLine($"{Thread.CurrentThread.Name}: Started!");
        // Do some work
    }
}
```

2. Message Passing with Channels (`System.Threading.Channels`)

While shared memory with locks and signaling is powerful, it can lead to complex code, especially in asynchronous scenarios. Message passing, where threads communicate by sending and receiving messages, often simplifies concurrency. `System.Threading.Channels` provides an asynchronous, thread-safe way to implement producer-consumer patterns and message passing.

- **Decoupling:** Producers and consumers don't directly interact with each other's state; they interact with the channel.
- **Asynchronous:** Designed to work seamlessly with async/await.
- **Bounded vs. Unbounded:**
    - **Unbounded Channels:** Have no limit on the number of items they can store. Producers will never block due to the channel being full.
    - **Bounded Channels:** Have a fixed capacity. Producers will block if the channel is full, and consumers will block if the channel is empty. This is useful for flow control.

```
using System;
using System.Threading.Channels;
using System.Threading.Tasks;

public class ChannelsExample
{
    public static async Task Run()
    {
        // Create an unbounded channel
        // var channel = Channel.CreateUnbounded<int>();

        // Create a bounded channel with a capacity of 5
        var channel = Channel.CreateBounded<int>(new
BoundedChannelOptions(5)
        {
            FullMode = BoundedChannelFullMode.Wait, // Producers
wait if full
            SingleReader = false, // Allow multiple readers
            SingleWriter = false // Allow multiple writers
        });

        // Start producer and consumer tasks
        Task producerTask = Producer(channel.Writer);
        Task consumerTask = Consumer(channel.Reader);

        await Task.WhenAll(producerTask, consumerTask);

        Console.WriteLine("Channels Example Finished.");
    }

    private static async Task Producer(ChannelWriter<int> writer)
    {
        for (int i = 0; i < 10; i++)
        {
            await writer.WriteAsync(i);
            Console.WriteLine($"Producer: Wrote {i}");
            await Task.Delay(100); // Simulate work
        }
        writer.Complete(); // Signal that no more items will be
written
    }

    private static async Task Consumer(ChannelReader<int> reader)
    {
        await Task.Delay(50); // Give producer a head start

        await foreach (var item in reader.ReadAllAsync()) //
Asynchronously read all items
        {
            Console.WriteLine($"Consumer: Read {item}");
            await Task.Delay(500); // Simulate work
        }
        Console.WriteLine("Consumer: No more items to read.");
    }
}
```

3. Semaphores (Semaphore and `SemaphoreSlim`)

A semaphore is a signaling mechanism that limits the number of threads that can concurrently access a shared resource or a section of code. It maintains a count of available "slots."

- **Semaphore:** A system-wide semaphore. It can be used to synchronize threads across different processes. It's heavier and less performant than `SemaphoreSlim`.
- **`SemaphoreSlim`:** An in-process semaphore. It's lighter and more performant, designed for synchronization within a single process. It supports `async/await`.

Key Methods:

- **`Wait()` / `WaitAsync()`:** Decrements the semaphore's count. If the count is zero, the calling thread blocks until a slot becomes available (i.e., another thread calls `Release()`).
- **`Release()`:** Increments the semaphore's count, potentially releasing a waiting thread.

Use Case: Limiting Concurrent Access

Imagine you have a resource that can only handle a limited number of concurrent requests, like a database connection pool, an external API, or a file handle.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class SemaphoreSlimExample
{
    // Allows a maximum of 3 concurrent accesses
    private static SemaphoreSlim semaphore = new SemaphoreSlim(3);

    public static async Task Run()
    {
        List<Task> tasks = new List<Task>();
        for (int i = 0; i < 10; i++)
        {
            int taskId = i;
            tasks.Add(Task.Run(async () => await
AccessResource(taskId)));
        }

        await Task.WhenAll(tasks);
        Console.WriteLine("SemaphoreSlim Example Finished.");
    }

    private static async Task AccessResource(int taskId)
    {
        Console.WriteLine($"Task {taskId}: Waiting to acquire
semaphore...");
        await semaphore.WaitAsync(); // Acquire a slot

        try
        {
            Console.WriteLine($"Task {taskId}: Acquired semaphore.
Accessing resource...");
            await Task.Delay(new Random().Next(500, 2000)); //
Simulate resource access
            Console.WriteLine($"Task {taskId}: Finished accessing
resource.");
        }
        finally
        {
            semaphore.Release(); // Release the slot
            Console.WriteLine($"Task {taskId}: Released
semaphore.");
        }
    }
}
```

4. Shared Memory Models vs. Message Passing (Channels)

These are two fundamental paradigms for inter-thread (or inter-process) communication and synchronization.

a. Shared Memory Model

- **Concept:** Multiple threads directly access and modify the same shared data in memory.
- **Pros:**
  - **Performance:** Can be very fast for large data sets as no copying is involved.
  - **Direct Access:** Threads work directly on the data.
- **Cons:**
  - **Complexity:** Requires careful use of synchronization primitives (locks, mutexes, semaphores) to prevent race conditions, deadlocks, and ensure data consistency.
  - **Debugging:** Race conditions can be notoriously hard to debug due to their non-deterministic nature.
  - **Scalability:** Can become a bottleneck if too many threads contend for the same lock.
- **When to Prefer:**
  - When performance is paramount and data sets are large.

    - When the shared data structure is complex and difficult to serialize/deserialize for message passing.
    - Within a single process where threads need fine-grained control over shared state.

b. Message Passing (Channels)

- **Concept:** Threads communicate by sending and receiving messages to each other, typically through a queue or channel. Data is often copied when a message is sent.
- **Pros:**
    - **Simplicity:** Often leads to simpler, easier-to-reason-about concurrent code. Threads don't directly share state, reducing race conditions.
    - **Decoupling:** Producers and consumers are loosely coupled.
    - **Safety:** Less prone to deadlocks and race conditions as data is typically immutable or transferred with ownership.
    - **Scalability:** Easier to scale to distributed systems (e.g., actors in Akka.NET).
- **Cons:**
    - **Overhead:** Copying data for each message can introduce overhead, especially for very large messages.
    - **Latency:** Messages introduce a small delay.
- **When to Prefer:**
    - When decoupling components is important.
    - When dealing with asynchronous operations (e.g., I/O-bound tasks).
    - When building robust, fault-tolerant systems where components might fail independently.
    - For producer-consumer patterns where data flow is sequential.
    - When the complexity of shared memory synchronization becomes too high.

In summary: Shared memory offers raw performance but demands meticulous synchronization. Message passing prioritizes safety, simplicity, and decoupling, often at the cost of some overhead. For modern asynchronous C# applications, `System.Threading.Channels` is often the preferred approach for producer-consumer scenarios due to its safety and ease of use with `async/await`.

Part 2: SQL Concurrency - Transactions, Isolation, and Locking

Database systems are inherently concurrent environments, handling numerous client requests simultaneously. SQL concurrency management ensures that multiple transactions can operate on the same data without compromising data integrity or system performance.

1. How Multiple Transactions Work Simultaneously

When multiple transactions execute concurrently, the database management system (DBMS) employs various mechanisms to ensure that the final state of the database is consistent, as if transactions were executed one after another (serially). This is achieved through a combination of:

- **Locking:** Preventing transactions from accessing data that another transaction is currently modifying.
- **Versioning (Multi-Version Concurrency Control - MVCC):** Allowing readers to see an older, consistent version of data while writers modify the latest version, reducing contention between readers and writers.
- **Isolation Levels:** Defining the degree to which one transaction's uncommitted changes are visible to other transactions.

2. Key Concepts: Consistency, Performance, Isolation

- **Consistency:** Ensures that a transaction brings the database from one valid state to another. Data must adhere to all defined rules (e.g., constraints, triggers) before and after a transaction. Concurrency control helps maintain this by preventing inconsistent intermediate states from being exposed.
- **Performance:** The ability of the database system to handle a high volume of concurrent transactions with minimal delay. High isolation levels and excessive locking can degrade performance.
- **Isolation:** The property that ensures concurrent execution of transactions results in a system state that would be achieved if transactions were executed serially. In other words, each transaction appears to execute in isolation from other concurrent transactions. This is the "I" in ACID.

3. ANSI SQL Isolation Levels and Read Phenomena

The ANSI SQL standard defines four isolation levels, each preventing specific "read phenomena" (anomalies that can occur during concurrent transactions).

a. Read Phenomena

- **Dirty Reads (Uncommitted Dependency):** A transaction reads data that has been modified by another transaction but not yet committed. If the modifying transaction then rolls back, the first transaction has read "dirty" or invalid data.
    - *Scenario:* Transaction A updates a row. Transaction B reads that row. Transaction A rolls back. Transaction B now has incorrect data.
- **Non-Repeatable Reads:** A transaction reads the same row twice, but between the two reads, another committed transaction modifies or deletes that row. The two reads yield different results.
    - *Scenario:* Transaction A reads a row. Transaction B updates/deletes that row and commits. Transaction A reads the same row again and gets a different (or missing) result.
- **Phantom Reads:** A transaction executes a query that returns a set of rows. If another committed transaction inserts new rows that satisfy the query's `WHERE` clause, a subsequent execution of the same query by the first transaction will return a different set of rows (phantoms).
    - *Scenario:* Transaction A queries for all employees in a department. Transaction B inserts a new employee into that department and commits. Transaction A re-queries and sees the new "phantom" employee.

b. ANSI SQL Isolation Levels

- **`READ UNCOMMITTED` (Lowest Isolation):**
    - **Prevents:** Nothing.
    - **Allows:** Dirty Reads, Non-Repeatable Reads, Phantom Reads.
    - **Use Case:** Rarely used for data integrity. Might be used for highly aggregated reports where slight inaccuracies are acceptable for maximum performance.
    - *Note: In SQL Server, this is often achieved with `WITH (NOLOCK)` hint.*
- **`READ COMMITTED` (Default for many DBMS, e.g., SQL Server):**
    - **Prevents:** Dirty Reads.
    - **Allows:** Non-Repeatable Reads, Phantom Reads.
    - **Mechanism:** Shared locks are held only while the data is being read, not for the entire transaction. Writers hold exclusive locks.

    - **Use Case:** Most common level. Provides a good balance between consistency and concurrency.
- **`REPEATABLE READ`:**
    - **Prevents:** Dirty Reads, Non-Repeatable Reads.
    - **Allows:** Phantom Reads.
    - **Mechanism:** Shared locks are held on all data read by the transaction until the transaction commits or rolls back. This prevents other transactions from modifying or deleting the read data.
    - **Use Case:** When a transaction needs to ensure that data it has read will not change during its lifetime.
- **`SERIALIZABLE` (Highest Isolation):**
    - **Prevents:** Dirty Reads, Non-Repeatable Reads, Phantom Reads.
    - **Allows:** Nothing.
    - **Mechanism:** Places range locks on data accessed by the transaction, preventing other transactions from inserting new rows that would satisfy the query (phantoms). All read and write operations acquire locks that are held until the end of the transaction.
    - **Use Case:** When absolute data consistency is critical, even at the cost of significant concurrency reduction. This level effectively makes concurrent transactions behave as if they were executed serially.

Setting Isolation Level in .NET (ADO.NET):

```
using System.Data;
using System.Data.SqlClient;

public class SqlIsolationExample
{
    public static void Run()
    {
        string connectionString = "YourConnectionString";

        using (SqlConnection connection = new
SqlConnection(connectionString))
        {
            connection.Open();

            // Begin a transaction with a specific isolation level
            SqlTransaction transaction =
connection.BeginTransaction(IsolationLevel.Serializable);

            try
            {
                // Perform database operations within the
transaction
                // Example:
                // SqlCommand command = new SqlCommand("SELECT *
FROM Products WHERE Category = 'Electronics'", connection,
transaction);
                // SqlDataReader reader = command.ExecuteReader();
                // ...

                transaction.Commit();
                Console.WriteLine("Transaction committed
successfully.");
            }
            catch (Exception ex)
            {
                transaction.Rollback();
                Console.WriteLine($"Transaction rolled back:
{ex.Message}");
            }
        }
    }
}
```

4. Locking Mechanisms

Locks are fundamental to concurrency control in databases. They prevent multiple transactions from accessing the same data in conflicting ways.

- **Granularity:** Refers to the size of the data unit being locked.
    - **Row-Level Locking:** Locks individual rows. Offers highest concurrency but incurs more overhead (more locks to manage).
    - **Page-Level Locking:** Locks entire data pages (a fixed-size block of data, typically 8KB in SQL Server). Less overhead than row-level, but reduces concurrency if multiple transactions need different rows on the same page.
    - **Table-Level Locking:** Locks the entire table. Lowest concurrency (only one transaction can access the table at a time for conflicting operations) but lowest overhead. Often used for schema changes or bulk operations.

5. Lock Types

SQL Server (and other DBMS) use various types of locks to manage concurrent access:

- **Shared (S) Locks:**
  - **Purpose:** Used for read operations (e.g., `SELECT`).
  - **Compatibility:** Multiple shared locks can exist on the same resource simultaneously.
  - **Conflict:** Conflicts with Exclusive (X) and Update (U) locks. If an X or U lock is present, an S lock request will wait.
- **Exclusive (X) Locks:**
  - **Purpose:** Used for write operations (e.g., `INSERT`, `UPDATE`, `DELETE`).
  - **Compatibility:** Only one exclusive lock can exist on a resource at a time. No other lock (S, U, X) can be held on the same resource concurrently.
  - **Conflict:** Conflicts with all other lock types.
- **Update (U) Locks:**
  - **Purpose:** Used during the initial phase of an `UPDATE` or `DELETE` operation. It's a hybrid lock.
  - **Compatibility:** Compatible with Shared (S) locks, but not with other Update (U) or Exclusive (X) locks.
  - **Benefit:** Prevents a common deadlock scenario. When a transaction needs to read data and then potentially update it, it first acquires a U lock. If it decides to update, the U lock is converted to an X lock. This prevents multiple transactions from acquiring S locks, then trying to upgrade to X locks simultaneously, which could lead to deadlocks.
- **Intent Locks (IS, IX, IU):**
  - **Purpose:** Placed on higher-level resources (e.g., tables, pages) to signal that a transaction intends to acquire locks on lower-level resources (e.g., rows within that table/page).
  - **IS (Intent Shared):** Indicates intent to place S locks on lower-level resources. Compatible with IS and IX locks.
  - **IX (Intent Exclusive):** Indicates intent to place X locks on lower-level resources. Compatible with IS locks, but not with other IX, S, or X locks.
  - **IU (Intent Update):** Indicates intent to place U locks on lower-level resources.
  - **Benefit:** Allows the DBMS to quickly determine if a higher-level lock (e.g., a table-level X lock) can be granted without checking every individual row lock.

6. Deadlocks

A deadlock occurs when two or more transactions are each waiting for a lock that the other transaction holds, creating a circular dependency where neither can proceed.

- **Definition:** Transaction A holds a lock on Resource 1 and requests a lock on Resource 2. Transaction B holds a lock on Resource 2 and requests a lock on Resource 1. Both are blocked indefinitely.
- **Detection:** DBMS systems have deadlock detection mechanisms (e.g., by building a wait-for graph).
- **Resolution:** When a deadlock is detected, the DBMS typically chooses one of the transactions (the "deadlock victim") to terminate (rollback) to break the cycle. The victim's transaction is rolled back, releasing its locks, allowing the other transaction(s) to proceed. The application then needs to handle this error and usually retry the rolled-back transaction.

Solutions and Best Practices (.NET & SQL)

1. **Consistent Order of Resource Access:** The most effective way to prevent deadlocks. If all transactions access resources (tables, rows) in the same predefined order, circular waits are less likely.
    a. *Example:* Always update `TableA` then `TableB`, never `TableB` then `TableA`.
2. **Keep Transactions Short and Atomic:** The shorter the transaction, the less time locks are held, reducing the window for deadlocks. Perform only necessary operations within a transaction.
3. **Use Appropriate Isolation Levels:**
    a. Higher isolation levels (e.g., `SERIALIZABLE`) acquire more locks and hold them longer, increasing the likelihood of deadlocks.
    b. Lower isolation levels (e.g., `READ COMMITTED`) reduce locking, thus reducing deadlocks, but at the cost of allowing more read phenomena. Choose the lowest isolation level that meets your application's data consistency requirements.
4. **Avoid User Interaction within Transactions:** Do not prompt users for input or perform lengthy client-side processing while a transaction is open. This holds locks unnecessarily.
5. **Implement Retry Logic in .NET:** Since the DBMS will choose a deadlock victim, your .NET application must be prepared to catch deadlock exceptions (e.g., `SqlException` with error code 1205 in SQL Server) and retry the entire transaction.

```
public void PerformTransactionWithRetry()
{
    int maxRetries = 3;
    int currentRetry = 0;
    bool success = false;

    while (currentRetry < maxRetries && !success)
    {
        try
        {
            // Your ADO.NET transaction code here
            // ...
            // If successful, set success = true;
            success = true;
        }
        catch (SqlException ex)
        {
            if (ex.Number == 1205) // Deadlock error code for SQL
Server
            {
                Console.WriteLine($"Deadlock detected. Retrying...
(Attempt {currentRetry + 1})");
                currentRetry++;
                Thread.Sleep(100 * currentRetry); // Exponential
back-off
            }
            else
            {
                throw; // Re-throw other SQL exceptions
            }
        }
        catch (Exception ex)
        {
            throw; // Re-throw other exceptions
        }
    }

    if (!success)
    {
        Console.WriteLine("Transaction failed after multiple retries
due to deadlock.");
    }
}
```

6. **Index Optimization:** Proper indexing can reduce the amount of data scanned and locked, improving performance and reducing lock contention.
7. **Consider `ROWLOCK`, `PAGLOCK`, `TABLOCK` Hints (Use with Caution):** While sometimes useful for specific scenarios, explicitly specifying lock hints can override default behavior and lead to more deadlocks if not fully understood. Generally, let the DBMS manage locking.
8. **`WITH (NOLOCK)` / `READ UNCOMMITTED` (Use with Extreme Caution):** Allows dirty reads. Only use for non-critical, highly concurrent reads where stale or inconsistent data is acceptable (e.g., reporting dashboards that are frequently refreshed). Never use for transactional data where consistency is paramount.

By understanding these concepts and applying the best practices, you can build robust and performant concurrent applications with .NET and SQL Server.

from0