

Optimize database performance

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/01-introduction>

Introduction

Completed

A slow query during peak hours can cascade into a service outage. A poorly chosen isolation level can cause users to see stale data or block each other for seconds at a time. Performance problems in Azure SQL Database rarely have a single cause. They emerge from the interaction between hardware configuration, concurrency behavior, query execution, and workload patterns. Diagnosing these problems requires understanding each layer and knowing which tools to use at each level.

Imagine a team managing an e-commerce application on Azure SQL Database. During a holiday sale, transaction throughput drops and customers report timeouts. Is the database under-provisioned? Are queries scanning entire tables instead of seeking through indexes? Are blocking chains forming between concurrent transactions? Is the query optimizer choosing a bad plan? Each possibility requires a different investigation tool and a different resolution strategy. This module gives you the skills to work through each of those questions systematically.

After completing this module, you're able to:

- Evaluate and recommend database configurations including service tiers, compute tiers, and resource limits.
- Choose transaction isolation levels and concurrency controls that balance consistency with throughput.
- Analyze query performance using execution plans and dynamic management views.
- Monitor and tune queries with Query Store and Query Performance Insight.
- Identify and resolve blocking and deadlocks using DMVs and Extended Events.

2. Recommend database configurations

Recommend database configurations

Completed

Before you can tune queries or troubleshoot concurrency, you need the right infrastructure underneath. Azure SQL Database gives you two resource models and multiple service tiers. The combination you choose sets the ceiling on compute, memory, I/O, and storage for your workload. Choose too little and performance suffers. Choose too much and you waste budget.

Compare resource models

Azure SQL Database supports two resource models: **vCore** and **DTU**. They measure and bill resources differently, so understanding the distinction helps you make the right choice from the start.

The **vCore model** gives you direct control over virtual cores, memory, and storage. You pick the hardware generation, service tier, and compute tier independently. If you're migrating from on-premises SQL Server, this model maps cleanly to physical CPU and memory, which makes capacity planning more straightforward. It also supports reserved instance pricing and the Azure Hybrid Benefit for cost savings.

The **DTU model** bundles CPU, memory, and I/O into a single unit called a **Database Transaction Unit** (DTU). DTU-based tiers (Basic, Standard, and Premium) offer preconfigured resource packages. This model works when you don't need fine-grained control over individual resource dimensions.

For most new deployments, Microsoft recommends the vCore model. It provides higher resource limits, greater scaling granularity, and more pricing flexibility.

Understand service tiers in the vCore model

The vCore model has three service tiers: General Purpose, Business Critical, and Hyperscale. Each tier uses a different architecture, which affects storage type, I/O performance, and availability.

General Purpose separates compute and storage. The database engine runs on a compute node while data files reside on Azure Blob Storage. Storage latency is typically between 5 milliseconds and 10 milliseconds. This architecture provides budget-friendly pricing and works well for most business workloads. If the compute node fails, Azure Service Fabric moves the process to a spare node and reattaches the remote storage files.

Consider the e-commerce application from the introduction. During normal business hours, General Purpose handles the order volume without issue. But during a holiday flash sale, I/O latency of 5

milliseconds to 10 milliseconds might not be fast enough for the checkout flow.

Business Critical integrates compute and storage on each node. The database engine and data files both use locally attached SSDs in an Always On availability group with three secondary replicas. This design gives you the lowest I/O latency (1 millisecond to 2 milliseconds on average), the highest IOPS (input/output operations per second), and a free read-only replica you can use for reporting queries. The trade-off is cost, roughly 2.7 times more than General Purpose for the same vCore count. For the e-commerce team, Business Critical makes sense if their checkout transactions need consistent sub-2-millisecond latency.

Hyperscale uses a decoupled storage architecture with independent page servers and a multi-tiered cache. It supports databases up to 128 TB, allows zero to four high-availability replicas, and scales compute up or down without copying data. You're billed only for allocated storage, not maximum storage. Hyperscale removes the practical storage and scaling limits of the other tiers and is suitable for the widest variety of workloads.

The following table summarizes the key differences:

Feature	General Purpose	Business Critical	Hyperscale
Storage type	Remote (Azure Blob Storage)	Local SSD	Decoupled with local SSD cache
Max storage	4 TB	4 TB	128 TB
Max IOPS per vCore	320	4,000	5,500 (local SSD)
Availability replicas	1 (no read replicas)	3 + 1 read replica	0 to 4 (configurable)
Best for	Budget-oriented workloads	Low latency, high I/O	Large databases, flexible scaling

Choose a compute tier

Within the vCore model, you also choose between two compute tiers: **provisioned** and **serverless**.

Provisioned compute allocates a fixed number of vCores that stay available regardless of workload activity. You pay a fixed hourly rate. This tier fits workloads with consistent or predictable resource consumption, like the e-commerce application that processes orders throughout the day.

Serverless compute automatically scales vCores based on demand and bills per second for the compute used. When the database is idle, it can autopause and eliminate compute costs entirely, though autopause is currently supported only in General Purpose. Serverless compute itself is available for both General Purpose and Hyperscale tiers. It works well for development environments, internal tools, or applications with intermittent traffic.

Match the configuration to your workload

Now that you understand the options, how do you decide? Evaluate your workload against these factors:

- **Latency requirements:** If your application needs I/O latency under 2 milliseconds consistently, choose Business Critical. For moderate latency tolerance, General Purpose is sufficient.
- **Storage size:** If your database exceeds 4 TB or you expect rapid growth, Hyperscale is the only option that accommodates up to 128 TB.
- **Read-heavy workloads:** Business Critical includes a free read-only replica. Hyperscale supports named replicas for flexible read scale-out.
- **Cost sensitivity:** General Purpose with provisioned compute offers predictable pricing. Serverless compute in General Purpose or Hyperscale reduces costs for intermittent workloads.
- **Availability requirements:** Business Critical provides the highest resiliency with three synchronous replicas and the fastest failover. Hyperscale lets you configure the number of replicas to balance resiliency with cost.

Tip

When migrating from on-premises SQL Server, use the vCore model because it maps directly to physical CPU and memory. The DTU model doesn't expose individual resource dimensions, which makes capacity planning harder for migrations.

Configure database-level settings

You picked your tier and compute model. Now look inside the database itself. Several settings affect how Azure SQL Database handles parallelism, query optimization, and recovery. You adjust these settings without changing your service tier.

Control parallelism with MAXDOP

Max degree of parallelism (MAXDOP) controls how many processor threads the engine assigns to a single query. Azure SQL Database defaults to **8**, which works for the widest variety of workloads. Before September 2020, new databases defaulted to 0, unlimited parallelism, and that caused problems. A single analytical query could consume every available thread, starving the checkout flow of CPU.

You set MAXDOP at the database level with `ALTER DATABASE SCOPED CONFIGURATION SET MAXDOP`. You can also set a different value for secondary replicas when your read-write and read-only workloads have different concurrency needs. For a specific query, use the `OPTION (MAXDOP)` hint. The one rule: avoid MAXDOP 0 in production. Unlimited parallelism leads to resource exhaustion, query timeouts, and application outages.

Let automatic tuning catch regressions

The query optimizer doesn't always pick the best plan. Statistics go stale, data distributions shift, and a plan that was fast yesterday becomes slow today. **Automatic tuning** monitors query performance

and applies corrections without waiting for you to notice.

Azure SQL Database supports three options:

- **FORCE_LAST_GOOD_PLAN** detects plan regressions and forces the previous fast plan. Enabled by default.
- **CREATE_INDEX** identifies missing indexes, creates them, and verifies the improvement. Disabled by default.
- **DROP_INDEX** removes unused and duplicate indexes. Disabled by default. Unique indexes, including indexes supporting primary key and unique constraints, are never dropped.

Every change goes through a validation window, 30 minutes to 72 hours depending on query frequency. If performance gets worse, the change is automatically reverted.

```
ALTER DATABASE CURRENT
SET AUTOMATIC_TUNING (FORCE_LAST_GOOD_PLAN = ON,
                      CREATE_INDEX = ON,
                      DROP_INDEX = OFF);
```

Think about the e-commerce application during a holiday sale. Query patterns shift as different product pages spike in popularity. **FORCE_LAST_GOOD_PLAN** catches those regressions automatically, so a bad plan change at 2 AM doesn't slow down checkout until someone notices Monday morning. You probably want to leave **CREATE_INDEX** and **DROP_INDEX** off until what they propose is reviewed.

Unlock optimizer features with compatibility level

Every database has a **compatibility level** that determines which query optimizer behaviors are available. New databases in Azure SQL Database default to level **170**, or the highest available level. Each level unlocks a set of **intelligent query processing (IQP)** features:

- **Level 150**: batch mode on rowstore, table variable deferred compilation, scalar user-defined function (UDF) inlining.
- **Level 160**: parameter sensitive plan optimization (PSP), cardinality estimation feedback.
- **Level 170**: optional parameter plan optimization.

Existing databases can run at a lower compatibility level because Microsoft never automatically upgrades this setting. A database created when a lower default was in effect keeps its original level. For example, if you created an Azure SQL Database in 2024, the database is still at level 160 if the level isn't manually updated. Likewise, if you imported a database through a BACPAC file, the imported database's compatibility level is based on the source database's compatibility level. To move up:

```
ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 170;
```

Don't change this setting blindly in production. Use Query Store to capture a performance baseline at the current level, upgrade in a test environment, and compare. If a query regresses, you can force

the old plan while you investigate.

Reduce plan cache bloat with `OPTIMIZE_FOR_AD_HOC_WORKLOADS`

The e-commerce application generates product search queries with dozens of filter combinations. Each unique query text gets its own compiled plan in the cache, even if that query never runs again. Over time, the plan cache fills with thousands of single-use plans, pushing out the frequently executed plans that actually matter.

`OPTIMIZE_FOR_AD_HOC_WORKLOADS` solves this issue. When enabled, the engine stores a tiny **compiled plan stub** on first execution instead of the full plan. Only when the same query runs a second time does the engine compile and cache the full plan.

```
ALTER DATABASE SCOPED CONFIGURATION  
SET OPTIMIZE_FOR_AD_HOC_WORKLOADS = ON;
```

This setting keeps the cache lean and ensures that plans for your most important queries stay resident in memory.

Understand accelerated database recovery

Accelerated database recovery is always enabled in Azure SQL Database. You can't turn it off, and you don't need to. Accelerated database recovery (ADR) redesigns the recovery process so that recovery time stays constant regardless of how many active transactions were running when a failure occurred. It also provides instantaneous transaction rollback and aggressive log truncation.

ADR stores row versions in a **persistent version store** (PVS) inside the database rather than in `tempdb`. Depending on the size of the row being modified, versions are stored either in-row on data pages or off-row in a separate internal table. Write-intensive workloads can see increased page splits and higher log generation because every row version is logged. To minimize that overhead, keep transactions short and reduce unnecessary aborted transactions.

The PVS shares your database's allocated storage, so a growing PVS reduces the space available for your data. To monitor off-row PVS overhead, query

`sys.dm_tran_persistent_version_store_stats` and check the `persistent_version_store_size_kb` column, which reports the size of off-row versions only and doesn't include in-row versions stored on data pages. To establish a baseline during typical workloads, compare that value to your total database size. If PVS grows significantly beyond that baseline, look for long-running transactions or high abort rates that delay version cleanup.

Key takeaways

Azure SQL Database gives you two resource models, three service tiers, and two compute tiers. The vCore model with General Purpose covers most workloads. Business Critical adds sub-2-millisecond latency. Hyperscale removes storage and scaling limits. Inside the database, MAXDOP 8 is the safe default, automatic tuning catches plan regressions, and upgrading your compatibility level unlocks

the latest intelligent query processing (IQP) features. Enable `OPTIMIZE_FOR_AD_HOC_WORKLOADS` to keep the plan cache clean, and monitor PVS storage usage from ADR using `sys.dm_tran_persistent_version_store_stats`, especially in write-heavy scenarios. Next, you explore how isolation levels and concurrency control affect the queries running inside this infrastructure.

3. Preserve data integrity with transaction isolation levels and concurrency controls

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/03-preserve-data-integrity-isolation-levels>

Preserve data integrity with transaction isolation levels and concurrency controls

Completed

Choosing the right hardware and configuration is only part of database performance. What happens when two customers try to buy the last item in stock at the exact same moment? Or when a report reads a price that another transaction is in the middle of changing? Transaction isolation levels control how the database handles these situations.

How isolation levels work

Isolation levels define how much one transaction can see of another transaction's uncommitted work. The SQL standard frames them around three **concurrency side effects**, each of which maps to a real problem in your applications. Let's use an e-commerce application to illustrate them:

- **Dirty reads:** A customer's order reads a discounted price that another transaction set but isn't committed yet. If the transaction that set the price rolls back, the order used a price that never existed. Dirty reads are defined as reading uncommitted data from another transaction, which can lead to incorrect behavior if that other transaction doesn't commit.
- **Nonrepeatable reads:** On the same transaction, your inventory service reads the stock count for an item. After the initial read, the transaction does some calculations, then reads the same row again. In between those steps, another transaction committed a change, so the count is now different. Nonrepeatable reads occur when a transaction reads the same row twice and gets different values each time.
- **Phantom reads:** A report queries all orders placed in the last hour, calculates a total, then runs the same query moments later within the same transaction. Another transaction inserted new

orders in between, so the totals don't match. Phantom reads happen when a transaction re-executes a query and sees new rows that weren't there before.

Consider our e-commerce application during a flash sale with hundreds of concurrent customers. All three side effects can show up at once. The isolation level you choose determines which ones your application tolerates and how much blocking it accepts in return.

Notice that each of these problems involves a transaction that performs multiple operations: read a row, do some work, then read again or make a decision. A single-statement transaction usually completes so quickly that these conflicts rarely surface. Isolation levels matter most when your transactions span multiple statements, which is common in business logic that reads data, applies rules, and then writes results.

Here's a typical pattern where isolation levels come into play. Imagine two customers both trying to buy the last unit of the same product:

```
-- Transaction A (Customer 1)          -- Transaction B (Customer 2)
BEGIN TRANSACTION;                      BEGIN TRANSACTION;

SELECT @Stock = StockCount
  FROM Products
 WHERE ProductID = 42;
-- @Stock = 1, proceed with order

UPDATE Products
  SET StockCount = StockCount - 1
 WHERE ProductID = 42;
INSERT INTO Orders (ProductID, Quantity)
  VALUES (42, 1);
COMMIT TRANSACTION;

-- StockCount is now -1. Two orders placed for one item.

SELECT @Stock = StockCount
  FROM Products
 WHERE ProductID = 42;
-- @Stock = 1, proceed with order

UPDATE Products
  SET StockCount = StockCount - 1
 WHERE ProductID = 42;
INSERT INTO Orders (ProductID, Quantity)
  VALUES (42, 1);
COMMIT TRANSACTION;
```

Both transactions read a stock count of 1 and both decided to proceed. The isolation level determines whether Transaction B sees the original value or waits for Transaction A to finish first. If the isolation level allows dirty reads, Transaction B reads the uncommitted value of 1 and proceeds, which leads to overselling. If the isolation level is more restrictive, Transaction B waits until Transaction A commits. If Transaction A commits successfully, Transaction B sees the updated stock count of 0 and can prevent the second order from going through. If Transaction A rolls back for some reason, Transaction B sees the original value of 1 and can still proceed without ever reading uncommitted data.

SQL Server and Azure SQL Database support six isolation levels. The first four use **pessimistic concurrency** (locking). The last two use **optimistic concurrency** (row versioning).

Lock-based isolation levels

With lock-based isolation, the database engine uses shared and exclusive locks to keep transactions from interfering with each other. The more restrictive the level, the longer those locks are held. Fewer side effects get through, but more transactions end up waiting.

READ UNCOMMITTED is the fastest and the riskiest isolation level. It skips shared locks on reads entirely, so there's no read-write blocking. But that also means a transaction can read another transaction's uncommitted changes. If that other transaction rolls back, your application just acted on data that never existed. This level makes sense only when approximate results are acceptable, like a dashboard showing rough order counts.

READ COMMITTED is the default isolation level in SQL Server and strikes a basic balance. Each read acquires a shared lock, grabs the data, and releases the lock right away. Releasing the lock quickly is enough to prevent dirty reads. However, nothing stops another transaction from changing the row between your two reads, so nonrepeatable reads and phantom reads can still happen.

REPEATABLE READ goes a step further than READ COMMITTED. It holds shared locks on every row you read until your transaction completes. Now those rows can't change while the transaction is active, which prevents both dirty reads and nonrepeatable reads. But other transactions can still insert new rows that match your query filter, so phantom reads remain a possibility.

SERIALIZABLE takes care of everything previous levels do and more. It takes range locks that cover not just the rows you read but also the gaps between key values, blocking inserts into those ranges. Dirty reads, nonrepeatable reads, and phantom reads are all prevented. The cost is real, though: range locks cause significantly more blocking and increase the chance of deadlocks. Reserve this level for scenarios like financial reconciliation or inventory reservation where phantom reads would cause calculation errors.

Row-versioning isolation levels

What if readers didn't have to wait for writers at all? That question is what row-versioning isolation addresses. Instead of blocking behind locks, the database engine keeps previous versions of rows in a **version store**. When a transaction needs to read data that another transaction is modifying, it reads from the version store instead of waiting. In Azure SQL Database, accelerated database recovery (ADR) is always enabled, so the version store resides in the database itself using the persistent version store (PVS).

Read Committed Snapshot Isolation changes the behavior of READ COMMITTED at the database level. With Read Committed Snapshot Isolation (RCSI) enabled, each read operation sees a snapshot of the data as it existed at the start of that *statement*. Writers still take locks on the rows they modify, but readers never block behind them. The important detail: RCSI is enabled by default in Azure SQL Database, so you get this behavior out of the box without any code changes.

SNAPSHOT isolation takes this solution a step further. Instead of a per-statement snapshot, each read sees the data as it existed at the start of the entire *transaction*. If your application needs to run multiple queries within a single transaction and get consistent results across all of them, SNAPSHOT isolation provides that guarantee. To use it, you must enable `ALLOW_SNAPSHOT_ISOLATION` on the database and set the isolation level explicitly in the session. One thing to watch for: if two transactions try to modify the same row, SNAPSHOT isolation raises an update conflict rather than letting one silently overwrite the other.

The following table summarizes the behavior of each isolation level:

Isolation level	Dirty reads	Nonrepeatable reads	Phantom reads	Blocking between readers and writers
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	Yes
REPEATABLE READ	No	No	Yes	Yes
SERIALIZABLE	No	No	No	Yes (range locks)
READ COMMITTED SNAPSHOT	No	Yes	Yes	No
SNAPSHOT	No	No	No	No (update conflicts possible)

Reduce blocking with optimized locking

RCSI eliminates read-write blocking. But what about write-write blocking? Traditional locking holds row and page locks for every modified row until the transaction commits. Under heavy write concurrency, those held locks add up fast.

Azure SQL Database addresses this issue with **optimized locking**, which is always enabled and works alongside RCSI. It uses two mechanisms:

- **Transaction ID (TID) locking:** Instead of holding individual key or row locks for every modified row, the engine takes a single exclusive lock on the transaction ID (TID). Other transactions that need to access the same row acquire a shared lock on the TID and wait for the modifying transaction to complete. The result is far fewer locks held in memory, which also makes lock escalation much less likely.
- **Lock after qualification:** Before a row is modified, the engine reads the latest committed version without acquiring a lock and checks whether the row matches the query predicate. Only rows that actually qualify get locked, and that exclusive lock is released as soon as the row update is complete, not at the end of the transaction. Lock after qualification (LAQ) requires RCSI to be enabled.

The practical effect is significant. Page and row locks for modifications are released as soon as each row is updated rather than held until the transaction commits. Concurrent writes that touch different

rows rarely block each other, even under peak load.

Choose the right isolation level

With six isolation levels available, how do you decide? Start with the defaults. For most Azure SQL Database workloads, the combination of RCSI and optimized locking already provides the best balance of consistency and performance. Readers don't block writers. Writers don't block readers. Lock memory stays low. You don't need to change anything.

Step up to SNAPSHOT isolation when your application runs multiple queries within a single transaction and needs consistent results across all of them. Think of a financial report that queries account balances and then totals them: you need both queries to see the same point-in-time data.

Reserve SERIALIZABLE for scenarios where phantom reads would cause real business errors, like double-booking inventory or miscalculating a reconciliation. Accept that throughput drops and deadlocks become more likely.

Important

Regardless of which level you choose, keep transactions as short as possible. Long-running transactions hold locks for extended periods, increase blocking, and consume version store space. A transaction that stays open while waiting for user input or an external API call can bring concurrency to a halt.

Key takeaways

Transaction isolation levels control the trade-off between data consistency and concurrency. Higher isolation prevents more side effects but increases blocking. Azure SQL Database enables RCSI by default, which eliminates read-write blocking by using row versioning instead of shared locks. Optimized locking further reduces blocking by releasing row and page locks as soon as each row is updated. For most workloads, the default combination of RCSI and optimized locking provides the best balance. Reserve SERIALIZABLE for scenarios where phantom reads would cause business logic errors.

4. Evaluate query performance with execution plans and DMVs

When a query runs slower than expected, the first step is understanding *how* the database engine executes it. Execution plans show you the exact operators, data access methods, and resource costs the optimizer chose for a query. Dynamic management views (DMVs) complement that by exposing runtime performance data across all queries in the database, so you can find the most expensive ones before diving into any single plan.

An **execution plan** is the set of instructions the query optimizer produces to retrieve and process data. It defines which tables to access first, whether to use indexes or scan tables, and how to join, filter, sort, and aggregate results. The optimizer evaluates multiple candidate plans and selects the one with the lowest estimated cost.



- 12 / 32

To display an estimated plan, run `SET SHOWPLAN_XML ON` before the query or select **Display Estimated Execution Plan** in SQL Server Management Studio (SSMS). To capture an actual plan, run `SET STATISTICS XML ON` or select **Include Actual Execution Plan** in SSMS before executing the query.

Even though the estimated and actual plans look similar, the actual plan's runtime metrics are crucial for diagnosing performance issues. For example, if the estimated row count for a table scan is 100 but the actual row count is 10,000, that could indicate outdated statistics leading to a bad plan choice. The optimizer compiles the plan based on statistics the first time it encounters a query. If those statistics don't reflect the current data distribution, the plan may perform poorly.

Identify common issues in execution plans

Execution plans are read left to right, top to bottom. The first operators access the base tables, and the final operator produces the query result. Look for these common issues:

Operator types tell you how the engine accesses data. There are many operator types, each representing a different method of retrieving or processing data. For example, an **Index Seek** operator represents a highly efficient method that targets specific rows using index keys. A **Table Scan** or **Index Scan** operator on the other hand, represents a less efficient method that reads every row. If you see a scan on a large table, you likely need an index. For example, if the e-commerce application queries orders by date and the plan shows a Clustered Index Scan on the `Orders` table, adding a nonclustered index on the `OrderDate` column could change that scan into a seek. Do note that not all scans are bad. If a table is small, or the search condition returns most rows in a table, a scan might be the most efficient access method. Always consider the context of the query and the size of the data. Know your data and use execution plans to confirm whether the access method makes sense.

Estimated versus actual row counts reveal whether the optimizer's assumptions match reality. The optimizer bases its plan on **statistics**, metadata that describes the distribution and density of data in your tables. If those statistics are stale, the estimated and actual row counts diverge. When the optimizer underestimates row counts, it might choose a **nested loop join** (which processes one row at a time from the inner table of a join) when a **hash join** (which builds a hash table in memory for fast lookups) would be faster, or allocate too little memory for a sort operation. Statistics can become stale after significant data changes, so updating statistics with `UPDATE STATISTICS` or enabling automatic statistics updates can help the optimizer make better decisions.

Key Lookup operators appear when the engine finds rows through a nonclustered index but needs extra columns from the clustered index. For every matching row, the engine performs an extra round trip to the clustered index to retrieve those columns. If the filter returns many rows, those extra lookups add up fast. For example, if the e-commerce application filters orders by `CustomerID` but also selects `OrderDate`, `TotalAmount`, and `ShippingAddress`, and the nonclustered index on `CustomerID` doesn't include those columns, the plan shows a Key Lookup for each matching order. You can eliminate Key Lookups by adding the missing columns as included columns in the index. Keep in mind that included columns increase index size, which can slow down writes, so weigh the read performance benefit against the write overhead.

Thick arrows between operators represent the number of rows flowing between them. An unexpectedly thick arrow early in the plan (reading from left to right, top to bottom) often means a missing filter or index is letting too many rows through.

Missing index suggestions appear as green highlighted text at the top of the graphical execution plan in SSMS. When the optimizer detects that an index could significantly reduce the cost of a query, it surfaces a recommendation directly in the plan. Right-click the suggestion and select **Missing Index Details** to generate a `CREATE INDEX` statement you can review and run. These suggestions are one of the easiest wins you can get from reading an execution plan.

Warnings appear as a yellow triangle with an exclamation mark (⚠) on the affected operator. Each warning points to an optimization opportunity. Common warnings include:

- **Missing statistics:** The optimizer couldn't find statistics for a column, so it guessed at row counts instead of using actual data distribution. To fix this issue, create statistics on the columns used in your queries or update existing statistics if they're stale.
- **Excessive memory grant:** The query requested more memory than it needed, wasting resources that other queries could use. This issue often happens when the optimizer overestimates row counts. Updating statistics or rewriting the query to filter rows earlier can help reduce memory grants.
- **No Join Predicate:** Two tables are joined without a proper condition, producing a Cartesian product that returns every possible row combination. Check your query for a missing or incorrect `ON` clause.
- **Implicit conversion:** A data type mismatch forces the engine to convert values at runtime, which can turn an index seek into a scan. For example, if a `WHERE` clause compares an `nvarchar` parameter to a `varchar` column, the engine converts every row in the column to `nvarchar` before comparing. To fix implicit conversions, match the data types in your query parameters to the column definitions.
- **Sort or Hash spill:** A sort or hash operation ran out of its granted memory and spilled intermediate results to tempdb. These operations are the second most common driver of high CPU after scans. If you see a spill warning, the optimizer likely underestimated row counts and requested too little memory. Running `UPDATE STATISTICS` to refresh the table's statistics or rewriting the query to reduce the number of rows before the sort can often eliminate the spill.

Execution plans are a powerful tool for understanding query performance. They show you exactly how the engine executes a query and where the bottlenecks are. By learning to read execution plans effectively, you can quickly identify and fix performance issues in your database queries.

Query DMVs for runtime performance data

DMVs expose real-time and accumulated performance data from the database engine. Azure SQL Database requires `VIEW DATABASE STATE` permission to query them. While execution plans show you how a single query runs, DMVs show you what's happening across all queries, which helps you find the most expensive ones first.

Find the most expensive queries

CPU time, logical reads, and execution count are the most common metrics to identify expensive queries. High CPU time or logical reads indicate a query is resource-intensive, while a high execution count means even a moderately expensive query can have a large impact on overall performance. Start by reviewing the top queries by average CPU time or logical reads to find candidates for optimization.

`sys.dm_exec_query_stats` returns aggregate performance statistics for cached query plans. Join it with `sys.dm_exec_sql_text` to see the query text and `sys.dm_exec_query_plan` to retrieve the execution plan.

The following query finds the top 10 queries by average CPU time:

```
SELECT TOP 10
    qs.total_worker_time / qs.execution_count AS avg_cpu_time,
    qs.execution_count,
    qs.total_logical_reads / qs.execution_count AS avg_logical_reads,
    SUBSTRING(st.text, (qs.statement_start_offset / 2) + 1,
        ((CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(st.text)
            ELSE qs.statement_end_offset
        END - qs.statement_start_offset) / 2) + 1) AS query_text
FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
ORDER BY avg_cpu_time DESC;
```

This script helps you identify which queries deserve your attention. High `avg_logical_reads` relative to the result set size often points to missing indexes or inefficient plans. However, be cautious when interpreting these results. A query with high average CPU time that only runs once a day might matter less than a moderate query that runs thousands of times per hour. Always consider both the average cost and the execution count when you prioritize. You can also order by `avg_logical_reads` to find queries that are heavy on I/O, which often indicates missing indexes or inefficient access methods.

Check currently executing queries

While the previous query shows you the most expensive historic queries in the plan cache, `sys.dm_exec_requests` gives you a snapshot of every request that is currently running. It includes columns for CPU time, reads, writes, wait type, wait time, and blocking session ID. Use this view to spot active queries that are consuming too many resources or stuck waiting on locks. This view is one of the most important DMVs for real-time performance monitoring and troubleshooting.

```
SELECT
    r.session_id,
    r.status,
    r.command,
    r.wait_type,
    r.wait_time,
    r.blocking_session_id,
    r.cpu_time,
```

```

        r.logical_reads,
        t.text AS query_text
FROM sys.dm_exec_requests AS r
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
WHERE r.session_id > 50
ORDER BY r.cpu_time DESC;

```

This query filters out system sessions (session IDs 1-50) and orders by CPU time. You can also order by `logical_reads` to find I/O-heavy queries. The `wait_type` and `wait_time` columns help you identify whether a query is waiting on locks, I/O, or other resources.

Discover missing indexes

Earlier, we saw how execution plans can show you missing index suggestions for a single query. The missing index DMVs give you a broader view of which indexes the optimizer would use across all queries if they existed. These views are a great way to find optimization opportunities that affect multiple queries. `sys.dm_db_missing_index_details` shows the table, equality and inequality columns, and included columns. `sys.dm_db_missing_index_group_stats` provides an improvement measure that estimates the cost reduction.

```

SELECT
    mid.statement AS table_name,
    mid.equality_columns,
    mid.inequality_columns,
    mid.included_columns,
    migs.avg_total_user_cost * migs.avg_user_impact *
        (migs.user_seeks + migs.user_scans) AS improvement_measure
FROM sys.dm_db_missing_index_groups AS mig
INNER JOIN sys.dm_db_missing_index_group_stats AS migs
    ON migs.group_handle = mig.index_group_handle
INNER JOIN sys.dm_db_missing_index_details AS mid
    ON mig.index_handle = mid.index_handle
ORDER BY improvement_measure DESC;

```

This query calculates an `improvement_measure` for each missing index recommendation, which is a product of the average cost of queries that would benefit from the index, the average percentage improvement, and the number of times those queries were executed. Sorting by this measure helps you prioritize which missing indexes to create first. However, remember that these results are just recommendations based on the queries currently in the plan cache. Always review the suggested index columns and test their impact on both query performance and write overhead before adding them to production.

Note

Missing index recommendations are suggestions, not directives. Always test the impact of a new index on both query performance and write overhead before adding it to production.

Monitor active sessions and waiting tasks

`sys.dm_exec_sessions` gives you information about all authenticated sessions, including sign in time, host name, program name, and cumulative CPU and reads. Combine it with `sys.dm_os_waiting_tasks` to see which tasks are waiting and what resources they're waiting on. These views become essential when you diagnose blocking and resource contention in a later unit.

Put it all together

Execution plans and DMVs give you a complete picture of query behavior. Start with DMVs to identify the most expensive queries. Then drill into their execution plans to understand *why* they're expensive. Is it a missing index causing a scan? Outdated statistics causing row estimate errors? A Key Lookup you can eliminate? This systematic approach, from system-wide view to individual query analysis, is the most efficient way to find and fix performance bottlenecks.

Key takeaways

Execution plans reveal the optimizer's strategy for a query, and actual plans include runtime metrics that expose discrepancies between estimated and actual row counts. When you read a plan, focus on operator types (seek vs. scan), row count estimates, warnings, and Key Lookup operators. DMVs provide system-wide performance data: use `sys.dm_exec_query_stats` to find the most expensive queries, `sys.dm_exec_requests` for currently running queries, and the missing index DMVs for optimization opportunities. Start broad with DMVs to identify where the biggest problems are, then drill into individual execution plans to understand why.

5. Monitor and tune queries with Query Store and Query Performance Insight

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/05-monitor-tune-queries-query-store>

Monitor and tune queries with Query Store and Query Performance Insight

Completed

Real-time monitoring tells you what's happening now, but what about yesterday? Last week? After a deployment? You need a historical perspective to diagnose performance regressions, because you

can only understand what changed by comparing current behavior against a baseline. **Query Store** gives you exactly this capability.

Understand Query Store architecture

Query Store captures query text, execution plans, and runtime statistics directly inside the database engine. It stores this data in three internal stores:

- **Plan store:** Persists execution plans for each query. A single query can have multiple plans over time due to statistics updates, schema changes, or index modifications.
- **Runtime stats store:** Records aggregate execution statistics (CPU time, duration, logical reads, row counts) for each plan, grouped into configurable time intervals.
- **Wait stats store:** Captures wait statistics per query and plan, categorized by wait type. This links query performance directly to the resources the query waits on.

Query Store is enabled by default in Azure SQL Database. Data is written asynchronously to minimize overhead on the production workload. You can configure the retention period, maximum storage size, and capture mode using `ALTER DATABASE SET QUERY_STORE` options.

Consider a common scenario: your team deploys a code update on Tuesday. By Thursday, users report that a key page is slow. Without Query Store, you'd need to reproduce the issue and compare current execution plans against plans you that were never saved. With Query Store, you open the Regressed Queries view and immediately see which queries switched to slower plans after Tuesday's deployment.

Detect regressed queries

One of the most valuable capabilities of Query Store is detecting queries whose performance degraded after a plan change. The **Regressed Queries** view in SQL Server Management Studio (SSMS) surfaces queries that recently switched to a slower execution plan.

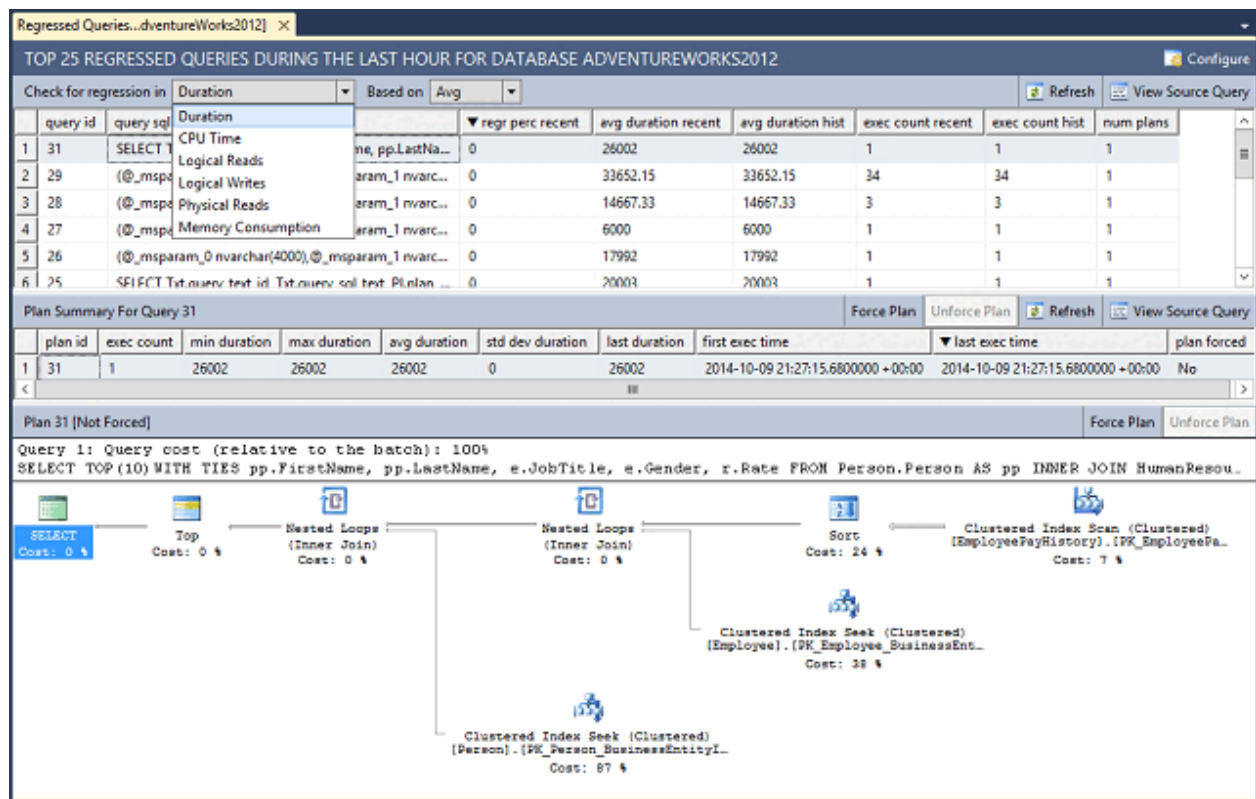
To use this view:

1. In SSMS, expand the database node in Object Explorer.
2. Expand the **Query Store** folder.
3. Select **Regressed Queries**.

This view displays queries where runtime metrics are worse than in a previous time interval. Use the dropdown lists at the top to filter by metric (CPU Time, Duration, Logical Reads, Memory Consumption, Execution Count, and more), time range, and aggregation method (Average, Maximum, Total).

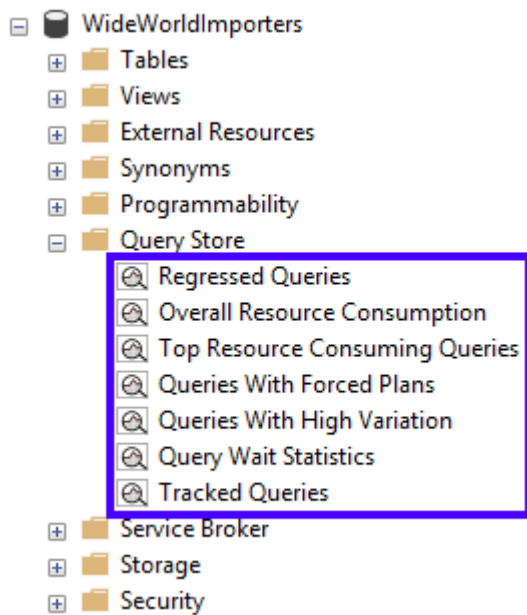
Selecting a query shows its plan history as a set of points in a chart. Each point represents a different execution plan. Select two plans to compare them side by side. This comparison is where the real investigation happens. Look for operators that changed between the fast plan and the slow plan.

Common patterns include a new plan switching from an index seek to a scan, losing a key lookup that was previously efficient, or introducing a sort operator that spills to disk.



The Query Store folder in Object Explorer contains several other views beyond Regressed Queries:

- **Top Resource Consuming Queries:** Shows the queries with the highest resource usage for a chosen metric and time range. This report is the most common starting point for day-to-day performance tuning.
- **Queries With High Variation:** Surfaces queries whose performance fluctuates significantly, which often indicates parameter sensitivity or varying data distributions.
- **Queries With Forced Plans:** Lists all currently forced plans so you can review and manage them.
- **Query Wait Statistics:** Groups *wait statistics* by category and show which queries contribute to each wait type.



Query the Query Store with T-SQL

You can also query Query Store data directly using its catalog views. The following query finds the top 10 queries by average duration in the last hour:

```
SELECT TOP 10
    qt.query_sql_text,
    q.query_id,
    p.plan_id,
    ROUND(CONVERT(FLOAT, SUM(rs.avg_duration * rs.count_executions))
        / NULLIF(SUM(rs.count_executions), 0), 2) AS avg_duration,
    SUM(rs.count_executions) AS total_executions
FROM sys.query_store_query_text AS qt
INNER JOIN sys.query_store_query AS q
    ON qt.query_text_id = q.query_text_id
INNER JOIN sys.query_store_plan AS p
    ON q.query_id = p.query_id
INNER JOIN sys.query_store_runtime_stats AS rs
    ON p.plan_id = rs.plan_id
WHERE rs.last_execution_time > DATEADD(HOUR, -1, GETUTCDATE())
GROUP BY qt.query_sql_text, q.query_id, p.plan_id
ORDER BY avg_duration DESC;
```

This query joins four catalog views: `sys.query_store_query_text` (query text), `sys.query_store_query` (query metadata), `sys.query_store_plan` (execution plans), and `sys.query_store_runtime_stats` (runtime statistics). You can adapt it to find queries by CPU time, logical reads, or execution count by changing the metric in the `SELECT` and `ORDER BY` clauses.

Force a plan

When you identify that a previous plan was better, you can tell the optimizer to use that specific plan for future executions. This approach gives you an immediate fix without modifying the query or adding hints to the application code.

To force a plan:

```
EXEC sp_query_store_force_plan
    @query_id = 42,
    @plan_id = 17;
```

To unforce a plan, and let the optimizer choose freely again:

```
EXEC sp_query_store_unforce_plan
    @query_id = 42,
    @plan_id = 17;
```

Plan forcing is useful during deployments. If a statistics update or schema change causes a plan regression, you can force the previously working plan immediately while you investigate the root cause. Think of it as a performance rollback that doesn't require an application rollback.

Apply Query Store hints

Sometimes you need to shape query execution without modifying application code. **Query Store hints** let you attach a query hint to a specific query through Query Store, and the optimizer applies it during compilation.

For example, to limit a query to a single thread:

```
EXEC sp_query_store_set_hints
    @query_id = 42,
    @query_hints = N'OPTION (MAXDOP 1)';
```

You can also force a recompile on every execution, which is useful for queries with parameter sensitivity issues:

```
EXEC sp_query_store_set_hints
    @query_id = 42,
    @query_hints = N'OPTION (RECOMPILE)';
```

You can even combine multiple hints in a single call:

```
EXEC sp_query_store_set_hints
    @query_id = 42,
    @query_hints = N'OPTION (MAXDOP 1, MAX_GRANT_PERCENT = 10)';
```

To remove a hint:

```
EXEC sp_query_store_clear_hints @query_id = 42;
```

Query Store hints override hard-coded statement-level hints and existing plan guides, so they give you full control without touching the application code.

Analyze wait statistics per query

Query Store captures **wait statistics** per query, not just at the server level. Enable wait stats capture with:

```
ALTER DATABASE CURRENT  
SET QUERY_STORE (WAIT_STATS_CAPTURE_MODE = ON);
```

The **Query Wait Statistics** view in SSMS groups waits into categories such as CPU, Lock, Buffer IO, and Memory. Selecting a wait category reveals which queries contribute the most to that wait type.

This connection between waits and queries is a significant advantage over server-level wait statistics, which can't attribute waits to specific queries. For example, if you see high Lock waits, you can drill into the specific queries causing them. You can then examine their execution plans, check their isolation levels, or evaluate whether missing indexes are causing long-held locks.

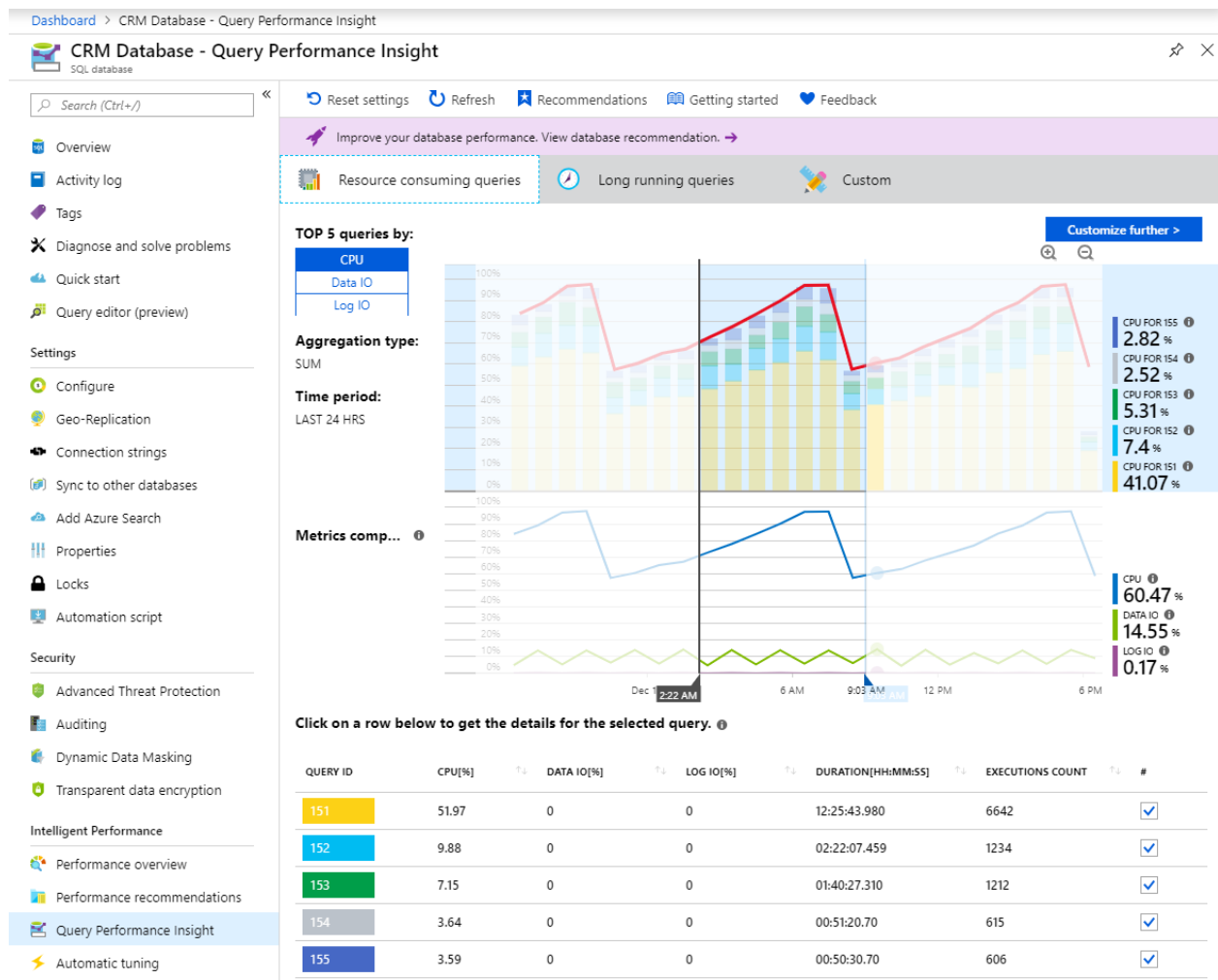
Monitor with Query Performance Insight

Query Performance Insight (QPI) is an Azure portal feature that visualizes Query Store data. It gives you a graphical view of the top resource-consuming queries without requiring SSMS.

To access QPI:

1. Open your database in the Azure portal.
2. In the left menu, select **Intelligent Performance** > **Query Performance Insight**.

By default, QPI shows the top five CPU-consuming queries. The top line in the chart represents overall Database Transaction Unit (DTU) percentage for the database. The following bars show CPU percentage consumed by each selected query during the selected interval. This layout lets you see at a glance whether a single query dominates resource usage or whether the load is spread across many queries.



Select the **Custom** tab to change the view. You can switch the metric to **Duration** or **Execution count**, adjust the time interval (last 24 hours, past week, or past month), change the number of queries displayed (5, 10, or 20), and choose an aggregation function (Sum, Max, Min, or Avg). This flexibility lets you investigate different dimensions of the same workload without leaving the portal.

Selecting an individual query opens a detail view with three charts: CPU consumption over time, duration over time, and execution count over time. This drill-down helps you distinguish between a query that's slow on every execution and one that's fast most of the time but spikes occasionally.

QPI also integrates with **Database Advisor** recommendations. Performance annotations appear on the chart as icons with a vertical line. Hover over an annotation to see a summary of the recommendation, such as creating an index or parameterizing queries. Select the icon to open the full recommendation detail and apply it directly from the portal.

Note

Query Performance Insight is limited to displaying the top 5 to 20 queries. If your workload includes many smaller queries that collectively consume significant resources, they don't appear in QPI. For deeper analysis, query the Query Store catalog views directly or use SSMS.

Note

Query Performance Insight requires Query Store to be active. If Query Store runs out of space and enters read-only mode, QPI can't display new data. Increase the Query Store maximum size or clear old data to restore normal operation.

Follow best practices

The default capture mode, **Auto**, is the right choice for most workloads. It filters out infrequent queries and queries with negligible resource consumption, which keeps storage usage manageable. Pair this setting with **size-based cleanup set to Auto**, so the database automatically removes the oldest data when Query Store approaches its maximum size.

Build a habit of checking the **Regressed Queries** view after every deployment, statistics update, or index change. Plan regressions often appear within hours and are easier to fix when you catch them early. When you do find a regression, use plan forcing as a short-term fix while you investigate the root cause. Long-term, address the underlying issue through index tuning, statistics updates, or query rewrites.

The most common operational problem with Query Store is running out of space. When that happens, Query Store silently switches to **read-only mode** and stops collecting new data. You can detect this issue by checking `sys.database_query_store_options`:

```
SELECT actual_state_desc, desired_state_desc,  
       current_storage_size_mb, max_storage_size_mb,  
       readonly_reason  
FROM sys.database_query_store_options;
```

If `actual_state_desc` shows `READ_ONLY` while `desired_state_desc` shows `READ_WRITE`, Query Store switched itself. The `readonly_reason` column tells you why. To restore normal operation, increase the maximum storage size or clear old data:

```
ALTER DATABASE CURRENT  
SET QUERY_STORE (MAX_STORAGE_SIZE_MB = 1024);
```

Key takeaways

Query Store captures query text, execution plans, and runtime statistics over time, and it's enabled by default in Azure SQL Database. The Regressed Queries view surfaces queries whose performance degraded after a plan change, making it essential to check after every deployment or schema change. When you identify a regression, plan forcing gives you an immediate fix without modifying application code, while Query Store hints let you attach query-level hints to shape execution behavior. For teams working in the Azure portal rather than SSMS, Query Performance Insight provides a graphical view of the same Query Store data.

6. Identify and resolve blocking and deadlocks

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/06-identify-resolve-blocking-deadlocks>

Identify and resolve blocking and deadlocks

Completed

Blocking is normal in a database that uses locking. One transaction holds a lock. Another transaction waits. Brief blocking that lasts a few milliseconds is expected. It becomes a problem when it lasts long enough to affect users. Deadlocks are the more severe form: two transactions permanently block each other, and the database engine has to terminate one to break the cycle.

Blocking

Blocking occurs when one session holds a lock on a resource and another session requests a conflicting lock on the same resource. The requesting session waits until the first session releases its lock.

Resources can be rows, pages, or even entire tables. Locks can be shared (for reads) or exclusive (for writes). When a session requests a lock that conflicts with an existing lock, it becomes blocked until the first session commits or rolls back its transaction and releases its locks.

Identify blocking chains

In Azure SQL Database, Read Committed Snapshot Isolation (RCSI) is enabled by default, so read operations use row versioning instead of shared locks. This setting significantly reduces blocking between readers and writers. However, blocking between two writers, or blocking caused by explicit transactions with higher isolation levels, still happens.

The session at the top of a blocking chain is called the **head blocker**. All other blocked sessions are waiting, directly or indirectly, for the head blocker to release its locks. To find the head blocker, query `sys.dm_exec_requests` and look for sessions where `blocking_session_id` is nonzero:

```
SELECT
    r.session_id,
    r.blocking_session_id,
    r.wait_type,
    r.wait_time,
    r.wait_resource,
    t.text AS query_text,
    r.status,
```

```

r.command
FROM sys.dm_exec_requests AS r
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
WHERE r.blocking_session_id <> 0
ORDER BY r.wait_time DESC;

```

To find the head blocker in those results, look for the session ID that blocks others but isn't blocked itself. For example, suppose the query returns these rows:

session_id	blocking_session_id
55	52
60	52
52	0

Sessions 55 and 60 are both blocked by session 52, and session 52 has a `blocking_session_id` of 0, which means nothing is blocking it. Session 52 is the head blocker. Once you identify it, query `sys.dm_exec_sessions` and `sys.dm_exec_requests` filtered to that session ID to see what it's running and why it's holding locks.

Keep in mind that RCSI eliminates blocking between readers and writers, but not between two writers. Consider a scenario where session 52 runs a batch update inside an explicit transaction:

```

-- Session 52
BEGIN TRANSACTION;
UPDATE Orders SET Status = 'Processing' WHERE Region = 'West';
-- Transaction stays open while application does other work

```

This update acquires exclusive locks on every matching row. Now session 55 tries to update one of those same rows:

```

-- Session 55
UPDATE Orders SET Priority = 1 WHERE OrderId = 4820;

```

Session 55 waits because session 52 already holds an exclusive lock on that row and hasn't committed. A `SELECT` query against those rows would still succeed under RCSI because it reads a row version instead of requesting a shared lock. With RCSI removing reader-writer blocking by default, the blocking you encounter in Azure SQL Database typically involves two sessions that both need to write to the same rows.

Recognize common blocking scenarios

Understanding *why* blocking happens helps you prevent it. The Azure SQL Database engine automatically manages locks, but certain patterns lead to longer blocking:

A long-running query that holds locks for an extended period. The query is actively executing and making progress, but it holds locks the entire time. Other sessions that need conflicting locks on

the same resources wait until the query finishes. To resolve this issue, look for ways to optimize the query, such as adding indexes or rewriting it to touch fewer rows.

A sleeping session with an uncommitted transaction. A session executes a statement within an explicit transaction, then stops executing but never commits or rolls back. The locks from the transaction remain held indefinitely. This issue often happens when an application experiences a query timeout or cancellation but doesn't issue a corresponding `ROLLBACK`. Use `SET XACT_ABORT ON` so that runtime errors automatically roll back the transaction.

A session that didn't fetch all result rows. An application sends a query but doesn't retrieve all the rows from the result set. Locks can remain held on rows that haven't been fetched yet. Make sure your application fetches all result rows to completion.

A session in a rollback state. A query that was terminated (with `KILL` or by a deadlock) is rolling back its changes. Rollback can take significant time for large modifications, and the session continues to hold locks during this process. Wait for the rollback to complete, and avoid large batch modifications during busy periods.

An orphaned connection. A client application crashes or a workstation restarts without cleanly closing the database connection. The server doesn't immediately detect the disconnection, so locks from that session remain held. Terminate the orphaned session with `KILL <session_id>;`.

Note

Two of these scenarios are mitigated in Azure SQL Database by default. RCSI reduces the impact of unfetched result rows because `SELECT` queries don't acquire shared locks under row versioning, so unfetched rows don't block writers. Accelerated Database Recovery (ADR) makes lengthy rollbacks rare because it can undo changes almost instantaneously regardless of transaction size. The remaining three scenarios (long-running queries, sleeping sessions with uncommitted transactions, and orphaned connections) remain fully relevant because they involve exclusive write locks that RCSI and ADR can't release early.

Resolve active blocking

When you find active blocking:

1. Identify the head blocker using the DMV query shown earlier.
2. Determine whether the blocking session's transaction can finish on its own or whether it's waiting on external input.
3. If the blocking session is an orphaned or abandoned connection, terminate it with `KILL <session_id>;`
4. Review the blocking query's execution plan for optimization opportunities such as missing indexes.

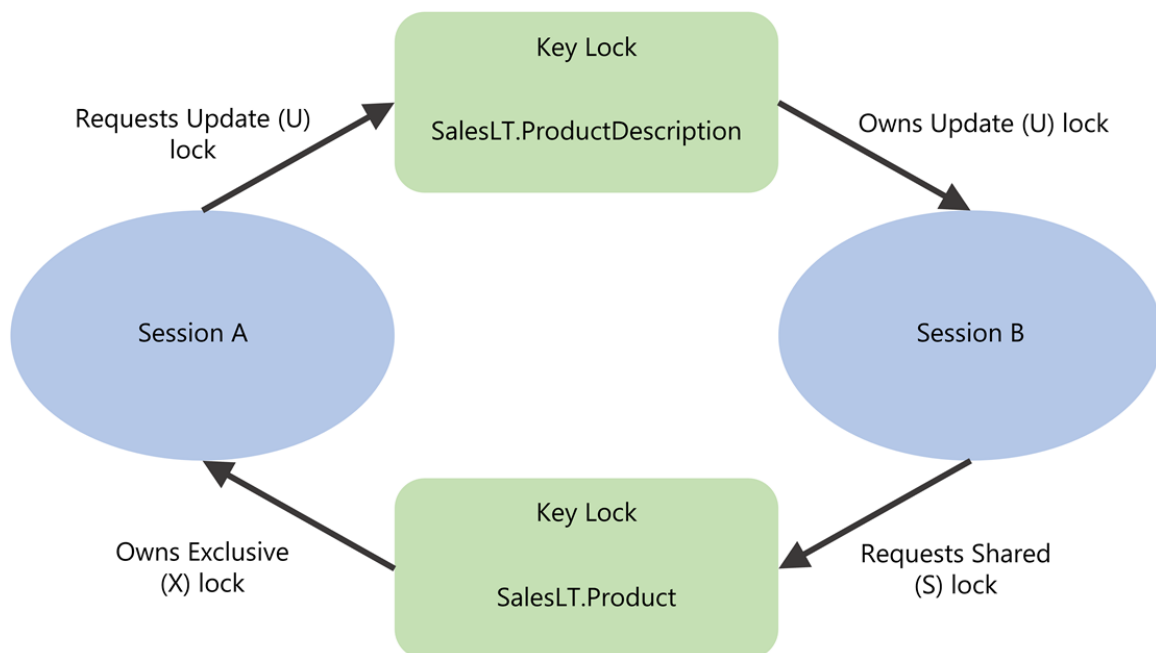
To prevent blocking from recurring, keep transactions short. Execute only the minimum required statements within a transaction and commit immediately. Use `SET XACT_ABORT ON` in your

application code so that any runtime error automatically rolls back the entire transaction, which prevents half-completed transactions from holding locks indefinitely. Move all user-facing logic outside transaction boundaries.

Deadlocks

A **deadlock** occurs when two or more transactions form a circular dependency. Each transaction holds a lock that the other needs, and neither can proceed. Here's a concrete example:

1. Transaction A updates row 1 and acquires an exclusive lock.
2. Transaction B updates row 2 and acquires an exclusive lock.
3. Transaction A tries to update row 2 and is blocked by Transaction B.
4. Transaction B tries to update row 1 and is blocked by Transaction A.



Neither transaction can finish. The database engine's **deadlock monitor** periodically checks for these cycles, with a default interval of five seconds that drops to as low as 100 milliseconds when deadlocks are frequent. When it detects a cycle, it chooses the transaction that's least expensive to roll back as the **victim**, rolls it back, and returns error 1205 to the application. This rollback allows the other transaction to complete.

Capture deadlock information

In SQL Server and Azure SQL Managed Instance, the `system_health` Extended Events session captures deadlock events by default. You can query the deadlock report from its ring buffer using `sys.dm_xe_session_targets` and `sys.dm_xe_sessions`.

In Azure SQL Database, the approach is different. You create a custom Extended Events session that captures the `sqlserver.database_xml_deadlock_report` event, and query it using the database-

scoped DMVs `sys.dm_xe_database_sessions` and `sys.dm_xe_database_session_targets`. The following example creates a deadlock capture session and queries its ring buffer:

```
-- Create and start the session
CREATE EVENT SESSION [deadlocks] ON DATABASE
ADD EVENT sqlserver.database_xml_deadlock_report
ADD TARGET package0.ring_buffer
WITH (STARTUP_STATE = ON, MAX_MEMORY = 4 MB);
GO

ALTER EVENT SESSION [deadlocks] ON DATABASE STATE = START;
GO

-- Query deadlock events from the ring buffer
DECLARE @tracename sysname = N'deadlocks';

SELECT
    d.value('/event/@timestamp')[1], 'datetime2') AS deadlock_time,
    d.query('/event/data[@name=' 'xml_report' ']/value/deadlock') AS deadlock_xml
FROM (
    SELECT CAST(target_data AS XML) AS rb
    FROM sys.dm_xe_database_sessions AS s
    INNER JOIN sys.dm_xe_database_session_targets AS t
        ON CAST(t.event_session_address AS BINARY(8)) = CAST(s.address AS BINARY(8))
    WHERE s.name = @tracename
        AND t.target_name = N'ring_buffer'
) AS ring_buffer
CROSS APPLY rb.nodes(
    '/RingBufferTarget/event[@name=' 'database_xml_deadlock_report' ']'
) AS xevent(d)
ORDER BY deadlock_time DESC;
```

The deadlock graph includes three sections. The **victim-list** identifies which transaction was terminated. The **process-list** shows each process involved, including the query text, isolation level, and lock mode. The **resource-list** shows the locked resources and which process owns and waits on each one.

In Azure SQL Database, you can also configure deadlock alerts through the Azure portal to receive notifications when deadlocks occur.

Prevent deadlocks

You can't eliminate all deadlocks, but you can significantly reduce how often they occur:

- **Access objects in a consistent order:** If all transactions modify Table A before Table B, circular dependencies can't form. Standardize access patterns through stored procedures.
- **Keep transactions short:** Shorter transactions hold locks for less time, which reduces the window for circular dependencies.
- **Use row-versioning isolation levels:** RCSI eliminates shared locks for read operations, which removes one common source of deadlock cycles. Optimized locking in Azure SQL Database further reduces deadlock likelihood.

- **Add appropriate indexes:** When queries scan many rows, they acquire locks across a wide range of data. Adding indexes that narrow the scan to fewer rows reduces lock conflicts.
- **Use plan forcing with Query Store:** If a plan change caused a query to scan more rows and acquire more locks, forcing the previous plan can reduce deadlocks while you investigate.

Handle deadlocks in application code

Applications should always include retry logic for deadlock errors. When a transaction is chosen as the deadlock victim, the database engine rolls it back and returns error 1205. Your application should catch this error, pause briefly, and resubmit the transaction.

```
BEGIN TRY
    BEGIN TRANSACTION;
    -- Your data modification statements
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF ERROR_NUMBER() = 1205
    BEGIN
        ROLLBACK TRANSACTION;
        WAITFOR DELAY '00:00:01'; -- Brief pause before retry
        -- Retry logic here
    END
    ELSE
    BEGIN
        ROLLBACK TRANSACTION;
        THROW;
    END
END CATCH;
```

Tip

Randomize the retry delay between attempts to prevent the same two transactions from deadlocking each other again immediately. A common pattern is to wait between one and three seconds with a random component.

Key takeaways

Blocking is normal, but extended blocking affects users, so you use `sys.dm_exec_requests` to find the head blocker and understand what it's doing. Common scenarios include long-running queries, sleeping sessions with uncommitted transactions, and orphaned connections, all of which you address by keeping transactions short, using `SET XACT_ABORT ON`, and ensuring applications properly manage connections and result sets. Deadlocks form when transactions create circular lock dependencies, and the database engine resolves them automatically by terminating the least expensive transaction and returning error 1205. You reduce deadlock frequency by accessing objects in a consistent order, keeping transactions short, using row-versioning isolation levels, and adding

appropriate indexes. Your application code should always include retry logic for error 1205 so it can recover automatically when chosen as a deadlock victim.

7. Exercise: Optimize query performance

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/07-exercise-optimize-query-performance>

Exercise: Optimize query performance

Completed

In this exercise, you investigate slow queries in Azure SQL Database using execution plans, dynamic management views (DMVs), and Query Store. You identify a missing index through the execution plan, simulate a parameter sniffing regression with a stored procedure, use the Query Store GUI to detect and fix the regression, apply a Query Store hint, and diagnose a blocking scenario.

Note

To complete this exercise, you need an [Azure subscription](#).

Launch the exercise and follow the instructions.

[Launch Exercise](#)

8. Knowledge check

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/08-knowledge-check>

Knowledge check

Completed

9. Summary

Summary

Completed

Performance optimization in Azure SQL Database is a systematic process. You start with infrastructure decisions, work through concurrency controls, and apply diagnostic tools to find and fix the problems that affect your users.

In this module, you learned how to:

- **Recommend database configurations:** Evaluate vCore versus DTU resource models. Choose among General Purpose, Business Critical, and Hyperscale service tiers based on I/O latency, storage, and availability needs. Select provisioned or serverless compute to match workload patterns.
- **Preserve data integrity with isolation levels:** Understand the trade-off between consistency and concurrency across six isolation levels. Use RCSI and optimized locking (both enabled by default in Azure SQL Database) to minimize blocking.
- **Evaluate query performance:** Read execution plans to identify scans, row estimate errors, Key Lookups, and warnings. Query DMVs to find the most expensive queries, currently running requests, and missing indexes.
- **Monitor and tune with Query Store:** Force previous plans for immediate fixes. Apply Query Store hints without modifying application code. Visualize performance in the Azure portal with Query Performance Insight.
- **Identify and resolve blocking and deadlocks:** Find head blockers with `sys.dm_exec_requests`. Capture deadlock graphs through Extended Events. Prevent concurrency issues by keeping transactions short, accessing objects in consistent order, and implementing retry logic for error 1205.

Learn more

- [vCore purchasing model - Azure SQL Database](#)
- [Transaction locking and row versioning guide](#)
- [Execution plan overview](#)
- [Monitor performance by using the Query Store](#)
- [Query Performance Insight for Azure SQL Database](#)
- [Understand and resolve blocking in Azure SQL Database](#)
- [Deadlocks guide](#)