20 Secrets to Optimizing SQL Queries

Anton Martyniuk

1. Use Indexes Wisely

Indexes drastically speed up queries by letting the database quickly find specific rows. Focus on indexing columns used frequently in WHERE, JOIN, ORDER BY, and GROUP BY clauses.

But don't over-index—too many indexes slow down insert, update, and delete operations. Regularly review your indexing strategy for maximum efficiency.

```sql
-- Create a composite index for frequent filters on Status and CreatedDate
CREATE INDEX IX_Orders_Status_CreatedDate
  ON dbo.Orders(Status, CreatedDate);
```

2. Avoid Select *

Always specify exactly the columns you need in your SELECT statement.
Retrieving unnecessary data wastes bandwidth and processing resources.
Fetching fewer columns results in faster queries and better server utilization.

```
-- Bad: SELECT *
SELECT * FROM dbo.Customers;

-- Good: only fetch needed columns
SELECT CustomerID, FirstName, LastName
  FROM dbo.Customers;
```

Anton Martyniuk

3. Implement Pagination Properly

Efficient pagination helps queries handle large datasets.

Instead of using OFFSET extensively, consider key-based pagination methods (if you don't need to access random page). Key-based pagination leverages indexed columns for quick navigation through records, significantly enhancing performance.

```
-- Key-based pagination on an indexed column
DECLARE @LastOrderID INT = 1000;
SELECT TOP(10) *
  FROM dbo.Orders
  WHERE OrderID > @LastOrderID
  ORDER BY OrderID;
```

4. Limit Rows Early

Filter your data as soon as possible in your queries. The earlier you apply filters (in WHERE clauses), the fewer rows your query processes. This significantly reduces query execution time and resource usage.

```sql
-- Apply filter before any joins or aggregations
SELECT o.OrderID, o.Total
  FROM dbo.Orders o
  WHERE o.OrderDate ⩾ '2025-01-01'
    AND o.OrderDate <  '2025-02-01';
```

5. Avoid Functions in WHERE

Using functions on columns in WHERE clauses prevents the use of indexes, causing full-table scans. Rewrite your conditions to avoid functions on columns, ensuring indexes remain effective. This small change can drastically boost query performance.

```sql
-- Bad: prevents index usage on OrderDate
SELECT * FROM dbo.Orders
 WHERE YEAR(OrderDate) = 2025;

-- Good: use sargable range
SELECT * FROM dbo.Orders
 WHERE OrderDate ⩾ '2025-01-01'
   AND OrderDate <  '2026-01-01';
```

Anton Martyniuk

6. Reduce Join Complexity

Keep JOIN operations simple and minimal. Excessive JOINs create complexity, causing slower performance and more resource consumption. Regularly audit your queries and eliminate unnecessary joins or redundant data lookups.

```sql
-- Simplify by removing an unnecessary lookup
-- Bad: joins Product, Category, Supplier but only needs CategoryName
SELECT p.ProductName, c.CategoryName
  FROM dbo.Products p
  JOIN dbo.Categories c
    ON p.CategoryID = c.CategoryID
  JOIN dbo.Suppliers s
    ON p.SupplierID = s.SupplierID;

-- Good: drop Suppliers join
SELECT p.ProductName, c.CategoryName
  FROM dbo.Products p
  JOIN dbo.Categories c
    ON p.CategoryID = c.CategoryID;
```

7. Choose Correct JOIN Types

Be selective with your JOIN types—INNER JOIN, LEFT JOIN, or EXISTS.
An INNER JOIN filters rows efficiently, while LEFT JOIN retrieves related data even without matches.
EXISTS clauses are highly performant for checking existence conditions.

```sql
-- Use EXISTS for simple existence check
SELECT c.CustomerID, c.Name
  FROM dbo.Customers c
 WHERE EXISTS (
     SELECT 1 FROM dbo.Orders o
      WHERE o.CustomerID = c.CustomerID
       AND o.Total > 100 );
```

Anton Martyniuk

8. Use Proper Data Types

Ensure data types match precisely when joining or filtering data. Mismatched data types force conversions, preventing indexes from working optimally. Matching exact data types enhances query efficiency significantly.

```
-- Ensure matching types: INT vs INT, not INT vs VARCHAR
CREATE TABLE dbo.Logs (
  LogID      INT        PRIMARY KEY,
  EventDate  DATETIME2  NOT NULL,
  UserID     INT        NOT NULL
);
```

9. Query Only What Changed

Instead of repeatedly querying the entire dataset, implement incremental data fetching. Track changes via timestamps or versioning columns. This method drastically reduces processing load and speeds up your queries.

```
-- Fetch only records updated since last run
DECLARE @LastRun DATETIME2 = '2025-05-20 00:00';
SELECT * FROM dbo.Records
 WHERE ModifiedDate > @LastRun;
```

10. Batch Operations

Group multiple insert, update, or delete operations into batches. Batch processing reduces overhead associated with individual transactions. This improves throughput and overall database responsiveness.

```sql
-- Batch 1000 inserts at a time
WHILE (1=1)
BEGIN
  INSERT INTO dbo.ArchiveTable (col1, col2)
  SELECT TOP(1000) col1, col2
    FROM dbo.SourceTable
   WHERE Processed = 0;

  IF @@ROWCOUNT = 0 BREAK;
END
```

11. Eliminate Redundant Subqueries

Repeated subqueries slow down queries significantly. Replace redundant subqueries with JOINs or Common Table Expressions (CTEs). This simplifies execution plans, boosting readability and query performance.

```sql
-- Bad: repeated subquery per row
SELECT o.OrderID,
       (SELECT Name FROM dbo.Customers c
         WHERE c.CustomerID = o.CustomerID) AS CustomerName
  FROM dbo.Orders o;

-- Good: single JOIN
SELECT o.OrderID, c.Name AS CustomerName
  FROM dbo.Orders o
  JOIN dbo.Customers c
    ON o.CustomerID = c.CustomerID;
```

Anton Martyniuk

12. Use EXISTS Instead of IN

The EXISTS clause typically outperforms IN, especially with large datasets. EXISTS returns immediately upon finding a match, saving processing time. Use EXISTS to quickly verify conditions without scanning entire datasets.

```sql
-- Better performance with EXISTS
SELECT p.ProductID, p.ProductName
  FROM dbo.Products p
 WHERE EXISTS (
     SELECT 1 FROM dbo.OrderDetails od
      WHERE od.ProductID = p.ProductID );
```

Anton Martyniuk

13. Normalize Wisely

While normalization reduces data duplication, too much can hurt query performance. Balance normalization with strategic denormalization to keep query complexity manageable. Finding the right balance ensures fast, efficient queries.

```
-- Denormalize heavy lookup for frequent reads
-- Create a summary table instead of always joining
SELECT ProductID, SUM(Quantity) AS TotalSold
  INTO dbo.SalesSummary
  FROM dbo.OrderDetails
 GROUP BY ProductID;
```

14. Use Materialized Views

Materialized views pre-calculate complex aggregations and joins, storing results persistently. This dramatically accelerates repeated read operations. Use them for expensive queries you run frequently.

```sql
-- SQL Server example using indexed view
CREATE VIEW dbo.Vw_TotalSales
WITH SCHEMABINDING
AS
  SELECT od.ProductID,
         SUM(od.Quantity) AS TotalQty
    FROM dbo.OrderDetails od
   GROUP BY od.ProductID;
GO
CREATE UNIQUE CLUSTERED INDEX IX_Vw_TotalSales_Product
  ON dbo.Vw_TotalSales(ProductID);
```

15. Analyze Execution Plans

Regularly examine execution plans to understand query performance. Execution plans reveal bottlenecks like missing indexes, table scans, or inefficient joins. Use this insight to continuously fine-tune your queries.

```
-- Show estimated plan
SET SHOWPLAN_XML ON;
GO
SELECT * FROM dbo.Orders WHERE Total > 100;
GO
SET SHOWPLAN_XML OFF;
```

Anton Martyniuk

16. Avoid Wildcards at Start

LIKE statements starting with wildcards (e.g., '%abc') cannot utilize indexes. Rewriting conditions to avoid leading wildcards allows efficient index usage. Small query adjustments here greatly improve performance.

```sql
-- Bad: cannot use index on LastName
SELECT * FROM dbo.Employees
 WHERE LastName LIKE '%son';

-- Good: leading text allows index seek
SELECT * FROM dbo.Employees
 WHERE LastName LIKE 'Anders%';
```

Anton Martyniuk

17. Keep Transactions Short

Long-running transactions cause locking, contention, and performance degradation. Shorten transactions by minimizing the number of operations in each transaction. Short transactions help maintain high database concurrency and responsiveness.

```
BEGIN TRAN;

UPDATE dbo.Accounts
   SET Balance = Balance - 100
 WHERE AccountID = 1;

INSERT INTO dbo.Transactions(AccountID, Amount)
VALUES (1, -100);

COMMIT;
```

18. Update Statistics Regularly

Database statistics guide query optimization by informing the optimizer about data distribution. Regularly updating statistics ensures the optimizer makes accurate decisions. This proactive step keeps query plans optimal.

```
-- Update stats on a big table
UPDATE STATISTICS dbo.Orders WITH FULLSCAN;
```

Anton Martyniuk

19. Use Query Hints Sparingly

Query hints force specific optimization behaviors but can hinder adaptability. Only apply hints after careful analysis and testing. Usually, proper indexing and schema design provide better long-term results than excessive hints.

```
-- Force recompile only for this execution
SELECT * FROM dbo.LargeTable
 OPTION (RECOMPILE);
```

20. Monitor and Tune Continuously

Consistently track query performance metrics like execution time and resource usage. Early identification of slow queries enables timely optimization actions. Continuous monitoring ensures high performance and scalability.

```sql
-- Find slow queries by average duration
SELECT TOP 10
       qs.query_hash,
       qs.execution_count,
       qs.total_elapsed_time / qs.execution_count AS AvgElapsed
  FROM sys.dm_exec_query_stats qs
 ORDER BY AvgElapsed DESC;
```

Anton Martyniuk

Next Steps

Hello there!
I'm Anton Martyniuk — a Microsoft MVP and Senior Tech Lead.

I have over 10 years of hands-on experience in .NET development and architecture. I've dedicated my career to empowering developers to excel in building robust, scalable systems.

Join my newsletter readers and let's build the future of .NET together!

antondevtips.com

01

Follow me on LinkedIn

I share amazing .NET and Software Development tips every day

02

Repost to your network

Share the knowledge with your network

03

Subscribe to my free newsletter

5 minutes every week to improve your .NET skills and learn how to craft better software

Anton Martyniuk

antondevtips.com