Achieving new SQL query performance levels through parallel execution in SQL Server

Marat Nuriev1*, Rimma Zaripova2, Andrey Potapov2, and Maxim Kuznetsov3

1Kazan National Research Technical University named after A. N. Tupolev – KAI, Kazan, Russia
2Kazan State Power Engineering University, Kazan, Russia
3Kazan State Agrarian University, Kazan, Russia

Abstract. This article provides an in-depth look at implementing parallel SQL query processing using the Microsoft SQL Server database management system. It examines how parallelism can significantly accelerate query execution by leveraging multi-core processors and clustered environments. The article explores SQL Server's sophisticated parallel processing capabilities including automatic query parallelization, intra-query parallelism techniques like parallel joins and parallel data aggregation, as well as inter-query parallelism for concurrent query execution. It covers key considerations around effective parallelization such as managing concurrency and locks, handling data skew, resource governance, and monitoring. Challenges like debugging parallel plans and potential bottlenecks from excessive parallelism are also discussed along with mitigation strategies. Real-world examples demonstrate how judicious application of parallel processing helps optimize complex analytics workloads involving massive datasets. The insights presented provide guidance to database developers and administrators looking to enable parallel SQL query execution in SQL Server environments for substantial performance gains and scalability.

1 Introduction

The advent of the digital era has brought about an explosion in data generation and collection across industries. As organizations aim to harness data to gain competitive advantages, the ability to efficiently process large volumes of data has become pivotal. However, traditional serial data processing approaches are proving inadequate in the face of modern big data challenges. This has led to growing adoption of parallel processing techniques that allow concurrent execution of operations and provide superior performance.

Parallel processing entails breaking up tasks into smaller units that can be simultaneously executed across multiple computational resources. This enables optimal utilization of available processors, leveraging combined computing power to accelerate operations. The key benefit is reduced processing time, with parallel execution enabling faster queries and more timely analytical insights.

* Corresponding author: marat_nu1@mail.ru

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).
However, implementing parallel data processing requires overcoming synchronization and coordination challenges. Effective task partitioning and scheduling algorithms are needed to distribute workloads optimally. Data dependencies must be managed to avoid conflicts between concurrent operations. Parallel programming frameworks and languages provide constructs to facilitate development, but architectural considerations remain vital.

Microsoft SQL Server incorporates robust capabilities to enable parallel querying and analytics for both transactional and data warehouse workloads. The SQL Server Query Processor utilizes sophisticated algorithms to determine optimal parallel execution plans. Techniques like parallel index scans, parallel data stream aggregates, and parallel joins allow leveraging multiple CPUs for faster data access and processing.

SQL Server also provides configuration options to control the degree of parallelism, balancing resource utilization against query concurrency. Beyond queries, technologies like parallel Data Warehouse load balancing and parallel index creation further enhance overall system throughput.

This article provides a comprehensive look at the organization and tuning of parallel SQL query processing in SQL Server environments. It examines the internal mechanisms powering parallel plan generation and execution. Key parallelism configuration parameters are analyzed, along with troubleshooting performance bottlenecks and optimizing workload distribution. Real-world examples demonstrate applied practices to tame big data through harnessing SQL Server's parallel processing capabilities.

The insights presented will empower data professionals to boost SQL query performance, enabling faster time-to-insight across massive datasets. With the right parallel processing strategies, SQL Server can serve as a high-throughput platform for tackling today’s data challenges.

2 Foundations of parallel data processing

Parallel processing techniques unlock substantial performance gains for data-intensive workloads by enabling concurrent execution across distributed resources. The key concept involves partitioning computation into discrete units that can be simultaneously processed in parallel and aggregating the results.

In database systems, query parallelization is enabled through splitting operations like large table scans, computation-heavy selections, and multi-table joins into smaller fragments that can leverage parallel hardware. Multi-core processors can assign these query subtasks to separate CPU cores, while clustered environments can distribute work across nodes [1,2].

Intra-query parallelism executes operations like scanning, filtering, aggregation, and joining in parallel as part of a single query execution plan. Inter-query parallelism allows concurrently running multiple queries by processing them simultaneously through the query scheduler.

SQL Server's sophisticated query optimizer determines when to introduce parallelism by estimating overall workload characteristics and available resources. It utilizes cost-based analysis to build optimal plans, leveraging various parallel execution techniques [3,4]:
- Parallel scanning partitioned data in sections across multiple threads.
- Performing aggregations simultaneously on subsets of data.
- Parallelizing join operations by dividing data and using hash/merge algorithms.
- Distributing sorting/grouping across resources for faster processing.

In clustered environments, SQL Server can direct parallel subtasks to distinct nodes for isolated execution before final result set assembly. This provides linear scalability to handle massive datasets spread across servers.
While parallel execution can significantly reduce query latency, excessive parallelism can sometimes have adverse effects like resource contention. Careful analysis and tuning is required to strike the right balance for a given workload mix and hardware configuration.

When applied judiciously, parallel SQL processing tackles big data bottlenecks through harnessing readily available but often underutilized multi-core processing capacity. As data volumes continue exponential growth, mastering parallel query optimization unlocks transformative performance for modern analytics.

### 3 Parallel processing architecture in MS SQL Server

SQL Server incorporates sophisticated parallel execution capabilities that can dramatically boost query performance by leveraging modern multi-core hardware. The Query Processor breaks down query plans into discrete tasks that can be simultaneously executed across multiple threads and CPU cores [5,6].

The degree of parallelism for a query is determined by SQL Server's cost-based optimizer after analyzing statement complexity, data distribution, and server resources. It builds an optimal plan splitting operations like scanning, filtering, aggregating, and joining into parallelizable subtasks [7,8].

For example, a query processing a large table can utilize parallelism by assigning different data partitions to separate threads running on individual cores. Each thread scans and filters its subset of data before the Query Processor aggregates results.

SQL Server implements various parallel execution techniques [9]:
- Parallel table and index scans – distributes data blocks across multiple threads.
- Parallel joins – uses hash or merge algorithms to match data subsets in parallel.
- Parallel aggregations – groups and calculates aggregates on partitions.
- Parallel sorts – each thread sorts a data portion, combined later.

In clustered environments, SQL Server can direct these parallel subtasks to different nodes for isolated execution before final result set assembly. This provides linear scalability to handle massive datasets distributed across servers.

Efficient resource management, scheduling, and synchronization are crucial for optimizing parallel performance [10]:
- SQL Server's thread pool handles provisioning and reuse of threads for parallel tasks.
- The scheduler balances workloads across available cores and CPUs.
- Lightweight synchronization objects like latches reduce contention between threads.
- Dynamic load balancing techniques adjust parallelism on the fly based on system conditions.

Database administrators can configure parameters like MAXDOP to control the maximum degree of parallelism permitted per query based on factors like hardware capacity [11]. Finding the ideal balance avoids unnecessary resource contention.

By leveraging parallel execution judiciously, SQL Server delivers tremendous performance gains for analytics and transaction processing workloads alike. As data volumes enter the petabyte scale, parallel query processing unlocks the scalability and speed needed for real-time insights.

### 4 Organizing parallel query processing

The Query Optimizer break down queries into independent parallel tasks during plan generation based on data flow and operation characteristics (see Fig.) [12]. Factors considered include [13]:
- Parallelizable operations - Scans, joins, aggregations etc.
- Subquery compatibility - Some subqueries may need sequential execution.
- Data segmentation - How to split data across threads.
- Resource utilization - Balance workload across available cores/nodes.

The SQL Server scheduler manages parallel task execution across threads pinned to CPU cores/nodes [14]. Lightweight synchronization objects like latches facilitate coordination. Dynamic load balancing detects and corrects skewed thread workload [15].

Partial results from subtasks are integrated based on data flow logic. The final result set is assembled by the parent query thread once child parallel tasks complete [16].

Data skew across threads can impede parallel performance. SQL Server has multiple skew handling strategies [17]:
- Dynamic redistribution of partitions to underutilized threads.
- Sampling data ahead of time to detect and correct skew.
- Repartitioning data during run time if high skew is detected.

Careful concurrency control mechanisms avoid blocking between simultaneous operations [18]:
- Partition level locks prevent inter-partition conflicts.
- Latches reduce contention for shared objects like memory caches.
- Lightweight spinlocks provide low-overhead mutual exclusion.

SQL Server monitors and governs resources used by parallel workloads [19]:
- Automatic memory management ensures sufficient available memory.
- The thread pool provides optimized thread reuse.
- Resource Governor can limit parallel resource consumption.

Tools like the query store, extended events, and DMVs enable deep visibility into parallel execution [20]. Performance counters and wait stats diagnose bottlenecks.

![Diagram of parallel query processing sequence in MS SQL Server.](image)

**Fig.** Diagram of parallel query processing sequence in MS SQL Server.

## 5 Navigating the intricacies of parallel execution

While parallel processing can deliver tremendous performance gains, it also introduces complexity that must be addressed:
- Resource governance - SQL Server's memory and thread management ensure queries get sufficient resources without excessive allocation [21]. Parallel workloads must be monitored to avoid resource exhaustion.
- Scalability – simply adding more parallelism can reach diminishing returns or even regress performance. Testing different parallelism levels is important [22].
- Blocking and deadlocks – parallel threads can block on the same resources causing stalls. Careful indexing, partitioning, and isolation level selection help avoid conflicts [23,24].
- Data movement – shuffling data between threads in a parallel plan adds CPU and memory overhead. Keeping parallel stages close together reduces data movement [25,26].
- Licensing costs – hardware utilization efficiency must be balanced against SQL Server edition licensing costs related to parallelism [27,28].
- Debugging and troubleshooting – tools like extended events and the query store provide visibility into parallel execution [29,30]. Special techniques are required for analyzing parallel plans.
- Query optimization – guidelines like simplifying complex queries, using plan guides, and redesigning algorithms facilitate tuning parallelism [31,32].
- Hardware limitations – available cores, NUMA architecture, memory bandwidth, and storage speed constrain possible parallelism speedups [33,34].

By understanding these intricacies, developers can optimize use of parallelism [35]:
- Start simple – introduce parallelism into one operation at a time.
- Benchmark and simulate queries under load to quantify gains.
- Analyze plans to model distributed data flow.
- Partition wisely to minimize data movement.
- Monitor resource usage and tune governors.
- Simplify queries and use indexes to improve parallel efficiency.

With careful testing and tuning, parallel execution can become a scalable and manageable technique for taming massive workloads.

6 Conclusion

In this extensive discussion, we took a deep dive into parallel data processing in SQL Server – a capability that is becoming indispensable for extracting value from massive datasets.

We covered how introducing intra-query and inter-query parallelism allows database engines like SQL Server to conquer long-running queries by leveraging all available CPU and storage resources. Techniques like parallel table scans, distributed aggregations, parallel joins, and indexed operations enable concurrent processing that simply cannot be achieved via traditional single-threaded execution.

However, as highlighted, effectively harnessing parallel power requires mastering the intricacies of modern multi-core hardware, SQL Server’s distributed architecture, and most importantly, the parallel execution plans generated by the Query Optimizer. Factors like proper indexing, statistics, and data distribution can radically impact possible performance gains.

While SQL Server handles much of the complexity behind the scenes with automatic parallelism adjustments, optimal configurations for parallel memory, tempdb, and resource governance are also critical for real-world workloads. Extensive testing and profiling is key to striking the right balance between parallel efficiency and oversaturation.

As data volumes continue exponential growth, driven by IoT sensors, social media, and other technologies, the need for parallel processing will only intensify. Future database trends like in-memory analytics, real-time streaming, and augmented intelligence will further raise the performance stakes.
In this landscape, the expertise to optimize parallel query plans, hardware, and infrastructure configurations will only become more crucial. Mastering parallel execution provides a key competitive edge in delivering game-changing speed and scalability for data-driven applications.

With this conclusion, we complete our journey through the world of parallel processing in SQL Server. The ground covered equips database developers and administrators with actionable insights for achieving substantial performance gains on everything from OLTP to advanced analytics workloads. Harnessing the parallel power lurking within server infrastructure unlocks a new era of possibilities for managing and deriving value from big data.

References

2. Z. M. Gizatullin, R. M. Gizatullin, M. G. Nuriev, 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), 120-123 (2020)
15. M. Shakirzyanov, R. Gibadullin, M. Nuriyev, E3S Web of Conferences 419, 02029 (2023)
16. K. Kulagin, M. Salikhov, R. Burnashev, 2023 International Russian Smart Industry Conference (SmartIndustryCon), 690-694 (2023)
17. J. Yoqubjonov, R. Gibadullin, M. Nuriev, E3S Web of Conferences 431, 07011 (2023)
18. I. Viktorov, R. Gibadullin, E3S Web of Conferences 431, 05012 (2023)
27. I. N. Madyshev, V. V. Kharkov, N. Z. Dubkova, M. G. Kuznetsov, AIP Conference Proceedings 2647, 1 (2022)
29. Z. Gizatullin, M. Shkinderov, 2019 International Russian Automation Conference, 8867761 (2022)