Memgraph:
Key Advantages over Neo4j

Memgraph is an in-memory graph database engineered for real-time analytics. It is designed to process data with minimal latency by using **in-memory architecture** for greater performance and efficiency.

Memgraph allows for the development of custom procedures to extend querying capabilities in cases where you feel limited by the Cypher query language. Users can expand these queries and manipulate data by using programming languages like Java, C++, and Python.

Additionally, Memgraph incorporates a library of pre-built procedures, known as MAGE (Memgraph Algorithm Graph Extensions), which includes various graph algorithms and utilities developed by Memgraph's team and external contributors. Through this, you can perform complex graph operations ideal for dynamic algorithms that demand real-time insights.

This document aims to highlight Memgraph's key differentiators compared to Neo4j and covers the following advantages:

- **Query execution speed** - The in-memory processing ensures queries run at lightning speeds with minimal latency. On the other hand, Neo4j relies more on disk-based storage which can lead to longer data retrieval times.
- **Deep-path traversals** -In contrast to Neo4j, Memgraph can traverse the graph using only Cypher queries, as these algorithms have been built into Memgraph's core. You can use advanced capabilities such as accumulators and path filtering without adding additional application logic.
- **Custom query modules** - Memgraph enables direct integration of custom logic within queries using familiar programming languages, a feature that offers more flexibility than Neo4j's more rigid structure.
- **MAGE vs. Neo4j Graph Data Science** - Memgraph's MAGE provides in-memory optimized algorithms and faster execution compared to Neo4j's Graph Data Science Library, which requires data caching from disk to RAM.
- **Dynamic algorithms** - Memgraph supports algorithms that adapt to real-time data changes, providing continuous, up-to-date analytics. Neo4j might not support real-time updates for large datasets so readily.

- **Analytical storage mode** - Memgraph allows for rapid, parallel data ingestion and analysis, optimizing for temporary, intensive computation workloads. This is particularly beneficial for workloads not requiring the strict consistency that Neo4j enforces through its ACID transaction model.

1. Query Execution Speed

Memgraph is an **in-memory graph database built in C++**. Such architecture, out of the box, is optimized for speed and can handle **high-throughput and low-latency operations**.

Memgraph optimizes the system to use less memory than is typically expected by making improvements to the query engine using fine-granular locking, lock-free data structures, and more. Because of this, Memgraph provides **high-speed data processing** while keeping the memory footprint low.

This is ideal for dynamic environments where real-time data analysis and quick decision-making are a priority. For example, in real-time fraud and anomaly detection systems, or network and IT ops.

Neo4j traditionally relies on a disk-based storage mechanism, augmented by in-memory caching to improve read performance. Although recent versions have focused on improving in-memory capabilities, its architecture inherently involves some level of disk I/O, which can introduce latency compared to a fully in-memory approach.

Read more about how you can improve query execution performance.

2. Deep-Path Traversals

The Memgraph deep-path traversals are an efficient way to handle complex graph queries and data mutations directly within the database core. This eliminates the need for the overhead of business logic on the application side.

The advanced Cypher syntax in Memgraph allows a more dynamic and accurate data analysis.

Here's an example:

```
MATCH path=( n  {id: 0} ) - [

*WSHORTEST
(r, n | n.total_USD) total_weight
(n, n | r.eu_border = false AND n.drinks_USD < 15)

] - (m { id: 46}]

RETURN path, total_weight;
```

The syntax inside the relationship `( -[*WSHORTEST (r, n | n.total_USD) total_weight (r, n | r.eu_border = false AND n.drinks_USD < 15)]- )` is specific to Memgraph and it's not part of openCypher. The advantage of deep-path traversals is that it allows the use of custom algorithms during graph traversal while filtering out nodes and relationships that don't meet specific criteria.

Memgraph also **supports accumulators**, such as `total_weight`, which aggregate values along a path for later use in the query. This feature significantly reduces the need for post-processing in applications, especially when dealing with large numbers of paths, ensuring efficient and accurate query results.

Additional Memgraph Cypher capabilities include features like **pattern matching**, **conditional statements**, and **filtering**, enhancing the flexibility and efficiency of data analysis and management. These features make Memgraph particularly useful for applications like cybersecurity, fraud detection, network resource optimization, identity and access management, data lineage, and recommendation engines.

Neo4j's approach to Cypher traversals, while powerful, might not offer the same level of customization and real-time data manipulation as Memgraph. In Neo4j, for complex queries you will need to call stored procedures in APOC (or Graph Data Science) which is less efficient than having it directly within the database core as Memgraph does.

Memgraph's deep-path traversals significantly improve execution speed, expressivity, and efficiency. This makes Memgraph particularly well-suited for applications where real-time analytics, custom query logic, and high-speed data processing are critical. In contrast, Neo4j offers a solid foundation for applications that require a balance of performance, reliability, and extensive graph database features. This makes Neo4j versatile across various use cases.

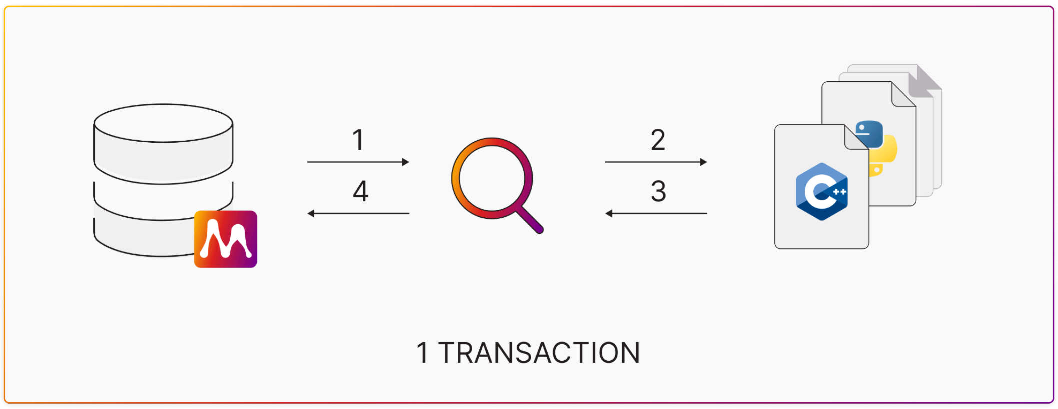
Read more about Writing Mutations and Complex Cypher Queries in Memgraph.

3. Custom Query Modules

You can create custom query modules in popular programming languages such as Python, C++, or Rust and execute this custom code during database transactions.

Neo4j supports custom procedures and functions through its Java API, offering customization within Cypher queries. However, Java's virtual machine reliance introduces higher memory overhead and slower performance compared to lower-level languages like C++. In contrast, Memgraph supports a wider array of languages, including C++, for writing custom queries. This approach not only reduces memory usage and boosts performance but also provides developers with greater flexibility.

Now, writing your custom query modules offers greater flexibility and performance of the Memgraph core functionalities which in turn allow for the integration of complex algorithms and data processing techniques. This approach significantly reduces the need for external data processing, leading to more efficient query execution and reduced latency.



The technical diagram shows data migration and here's how it works.

- **Step 1** - Start the transaction with a Cypher query in Memgraph. This is the first step to work with the database.

- **Step 2** - To make things faster and avoid unnecessary round trips, specific units of work can be executed inside the transaction using custom code that's connected to Memgraph via an API
- **Step 3** - After running the query module, results are returned from it using the YIELD keyword to be handled right within your Cypher query. This lets you work with the outcome directly.
- **Step 4** - Continue with your Cypher query to manipulate your data with the expressive processing you just brought in.

Now these are a few scenarios in which you would benefit from using custom query modules:

- **Data processing** - When standard query functions are insufficient, custom modules can perform specialized data processing like advanced mathematical calculations or text processing.
- **Data enrichment** - When you want to integrate external data sources into queries such as adding geo-location data or fetching real-time information from APIs.
- **Custom algorithms** - When you want to implement domain-specific algorithms that are not natively supported by the database like unique graph analytics or machine learning models.
- **Optimization routines** - When you want to run optimization algorithms for specific use cases like route planning or resource allocation.

Check out more details about Memgraph custom query modules.

4. Memgraph Advanced Graph Extensions (MAGE)

Memgraph Advanced Graph Extensions (MAGE) is an **open-source collection of algorithms** designed to enhance the Memgraph core functionalities. Specifically, to enhance the Memgraph in-memory data processing capabilities. This way, you have more efficient use of system memory and faster algorithm execution compared to disk-based solutions like the Neo4j GDS.

For example, when executing the PageRank algorithm, Neo4j may require more memory since it has to load data from the disk into RAM. Memgraph has an in-memory architecture and so has all the data already in RAM. No need for additional memory for data caching. The only memory used beyond what's already in RAM is for the execution of the algorithm itself.

Here's a code example that performs a match to find paths between two nodes identified by `id:1` and `id:2`. It uses a procedure from the module to calculate the maximum flow between these nodes.

```
MATCH p=(n:Generator)-[:CONNECTED_TO *BFS]->(m:Generator)
WITH project(p) AS graph
MATCH (n {id:1}), (m {id:2})
CALL max_flow.get_paths(graph, n, m, "flow")
YIELD flow, path
RETURN flow, path
```

The business advantage includes lower total cost of ownership due to reduced memory requirements, enhanced performance for real-time analytics, and the flexibility given by an open-source framework that you can customize for your specific need.

Other use cases include:

- **Recommendation systems** - To apply collaborative filtering methods to suggest new items to users based on graph patterns.
- **Fraud and anomaly detection** - To spot unusual patterns of behavior that may indicate fraudulent activity.
- **Community detection** - To discover groups or communities within larger networks, such as in social media data.

Read more about MAGE and Advanced algorithms.

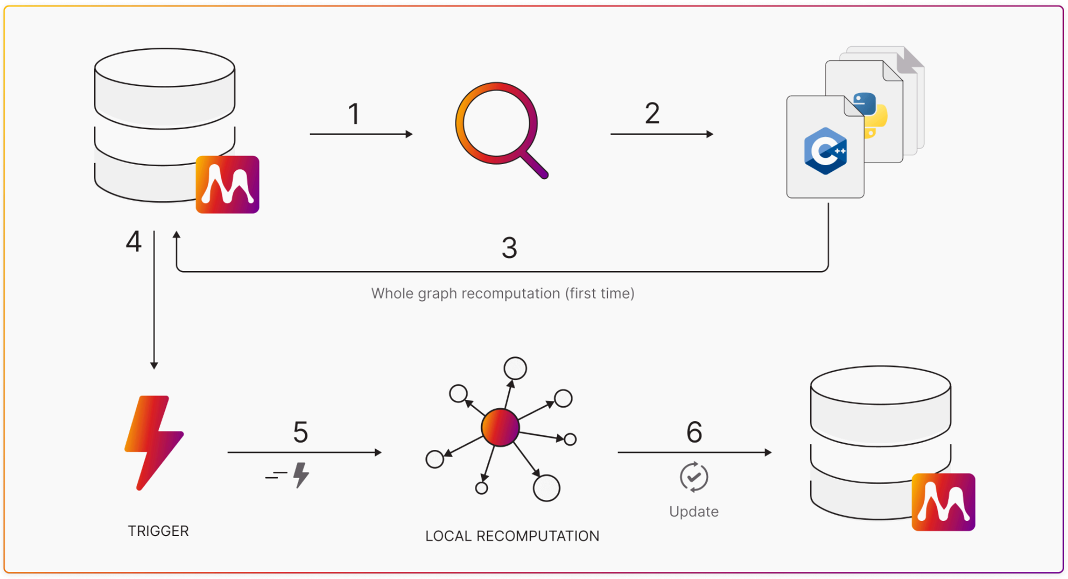
5. Dynamic Algorithms

In Memgraph, dynamic algorithms are a key feature for working with evolving graph data.

Dynamic algorithms are designed to update analytical results promptly as the graph changes, recalculating the necessary parts in a fraction of the time it would take to recompute the entire graph. This allows you to maintain real-time, up-to-date results essential for applications such as financial monitoring systems, social media platforms, and any other scenarios that require immediate response to data changes.

Both types of algorithms are essential for retrieving **immediate data insights**. Such features can significantly optimize performance and resource utilization in real-time applications.

The technical overview below, represents the process flow of dynamic algorithms in Memgraph.



- **Step 1** - Memgraph initiates a cypher query.
- **Step 2** - Inside the query, complex graph analytics are being executed in Memgraph as part of a query module
- **Step 3** - Memgraph is recording the algorithm's results for the entire graph.
- **Step 4 and 5** - Trigger and local recomputation represents the dynamic update. When a change occurs in the graph, instead of recomputing the entire graph, Memgraph only recomputes local or affected parts of the graph.
- **Step 6** - After local recomputation, the affected parts of the graph are updated with new values and structures.

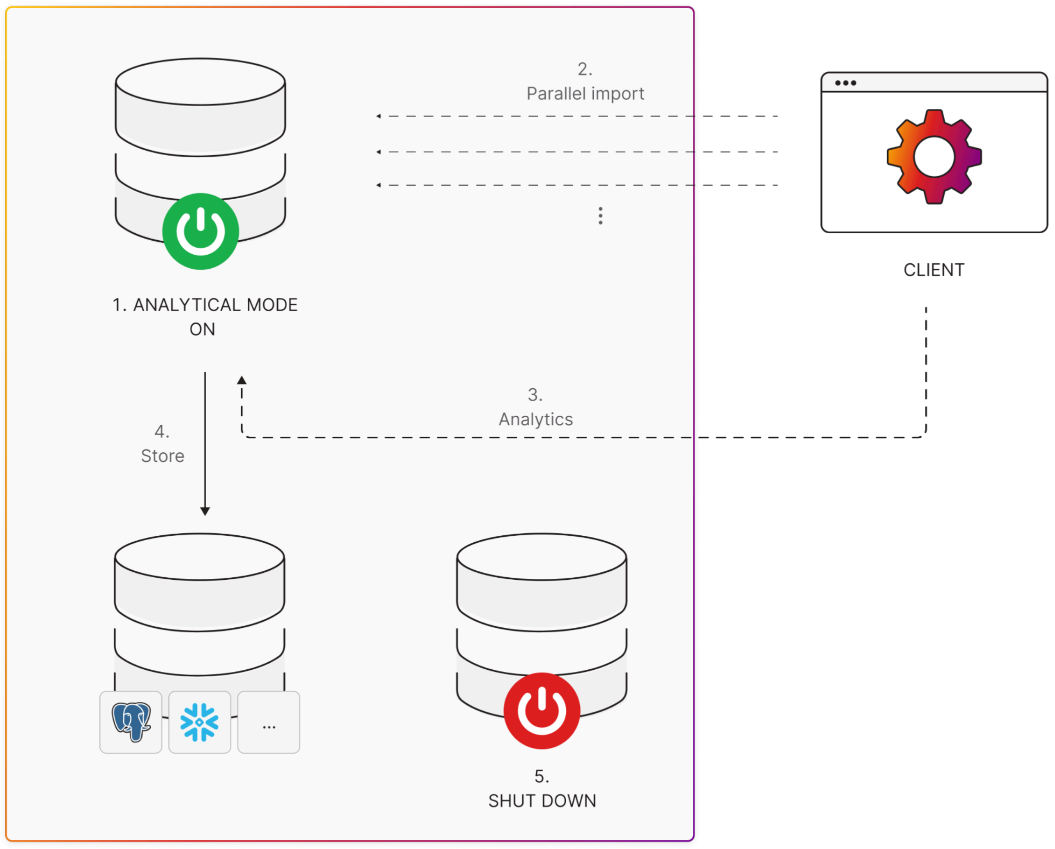
6. Analytical Storage Mode

Memgrap's analytical storage mode is a specialized operation mode that **optimizes the database for heavy analytical workloads**. It allows for rapid ingestion of large volumes of data and provides an optimized environment for temporary, complex computations.

This mode bypasses some of the usual transactional guarantees like ACID properties to maximize throughput for analytics.

It's particularly beneficial for workloads that don't require the strict consistency of traditional transactional operations. Also, it's ideal for scenarios where data can be pre-processed in bulk, analyzed quickly, and then the results moved elsewhere for persistence or further action.

Here's how it works:



- **Step 1 - Analytical mode activation** - Memgraph starts in the analytical storage mode designed for heavy computation.
- **Step 2 - Parallel data ingestion** - Data is imported simultaneously from various sources, enabling fast and efficient data loading.
- **Step 3 - Analytical processing** - Once data is ingested, Memgraph performs OLAP (Online Analytical Processing) to extract insights from the graph.
- **Step 4 - Data storage** - The results from Memgraph computations can be stored in other data systems, like S3, Snowflake, or Postgres.
- **Step 5 - Shutdown** - After the analytics are complete, Memgraph can be shut down, freeing up resources and emphasizing its role as a transient computational tool rather than a persistent storage system.

Read more about in-memory storage analytical mode in our docs. Additionally, check out the analytics mode blog post. It explains how disabling Multi-Version Concurrency Control (MVCC) during data import into an empty Memgraph instance can significantly speed up the process.

In summary, Memgraph distinguishes itself from Neo4j by offering better performance through in-memory data processing and advanced customization with support for multiple programming languages. It also has features like deep-path traversals and the MAGE library for enhanced graph operations. These capabilities not ensure rapid query execution, efficient data manipulation and provide a flexible environment for real-time analytics. That's why Memgraph is an optimal choice for applications requiring immediate insights and high-speed data processing.

Find out how Memgraph performs compared to Neo4j

View Benchgraph

Let's see how Memgraph fits into your environment

Contact us