Aurendil: Accelerating Graph Pattern Matching with Physical Optimizations

Guang Yang

Graphs are a fundamental data structure in mathematics and computer science as they model relationships between entities. Many core problems in areas such as social networks, recommendation systems, supply chains, and, more recently, indexing for LLMs are solved using graphs. In the era of big data and AI, the volume, velocity, and variety of data have grown exponentially, creating a critical need for a graph database management system to store, traverse, and analyze large-scale, complex networks with low latency and high scalability.

A fundamental operation in graph database systems is graph pattern matching (GPM), used to find specific structural patterns in graphs. Basically, it takes a query pattern (which includes node types, edge directions, and property conditions) and matches all subgraphs that conform to this topology. For example, we can use this cypher query `MATCH [m:Message]-(:HAS_CREATOR)->[p:Person] RETURN p.name, m.content` as the pattern. The database will then search the graph for every message and find its creator. Once a result is found, the database will output the message content and the creator's name for each message.

As the graph gets bigger, efficiently carrying out GPM is the main challenge for all graph databases. At the moment, the standard way of handling these queries uses joins, which creates large, memory-hogging intermediate tables. When you have queries involving large intermediate tables, resource utilization explodes, queries take a long time, and they may even get killed. We instead use a search-based approach, which is much more resource-efficient. The search technique uses a depth-first search. Think of GPM as navigating a gigantic maze. Instead of mapping out and memorizing every possibility before you find the actual path, we instead go down single paths and just go back if we reach a dead end. This avoids joins and minimizes database access. However, to make this approach efficient, we don't want to explore too many paths, and we want to make our access to the storage as quick as possible.

In this talk, we will show some ways to reduce the search space by exploiting the structure of the storage and query patterns. For query patterns that access multiple relationships for a single node (e.g., a star-shaped pattern), we minimize the relationship traversal to a single pass through the neighborhood of the node. To further reduce the search space, we can exploit the query's return type. For example, we can stop early and avoid going down paths if we just want to count the number of paths.