Is it worth it to use graph, if the relation is 1 on 1?

Relasi 1 ke 1 (1:1) pada grafik menunjukkan bahwa setiap node atau entitas hanya memiliki satu hubungan dengan node atau entitas lainnya.

Contoh Relasi 1 ke 1:

- Seorang karyawan hanya memiliki satu kartu identitas karyawan.
- Seorang siswa hanya memiliki satu induk atau wali.

Karakteristik Relasi 1 ke 1:

- Setiap node atau entitas hanya memiliki satu hubungan dengan node atau entitas lainnya.
- Tidak ada duplikasi hubungan antara node atau entitas.

Secara umum, graph database tidak ideal untuk relasi 1-on-1 (satu ke satu) karena:

Graph database unggul saat struktur data bersifat kompleks dan banyak hubungan (many-to-many). Contoh: social network, supply chain, fraud detection, dan recommendation engine.

Dalam relasi 1-on-1, data bisa ditangani lebih sederhana dan efisien di database relasional (seperti MySQL, PostgreSQL), karena cukup menggunakan foreign key.

Kapan boleh dipertimbangkan memakai graph untuk 1-on-1?

Jika relasi 1-on-1 itu hanya bagian kecil dari struktur data yang lebih besar dan kompleks (misalnya user juga terhubung ke group, location, device, dll), maka memakai graph bisa tetap berguna karena fleksibilitas dan skalabilitasnya.

Sumber Lain : https://www.quora.com/Does-a-graph-need-to-be-a-function-to-be-one-to-one

How to get data if it takes hop, efficiently?

Dalam graph database, "hop" berarti satu langkah perpindahan dari satu node ke node lain melalui relasi (edge). hop digunakan untuk menyatakan berapa banyak langkah atau kedalaman hubungan antar node yang ingin dijelajahi.

Kenapa *Hop* Penting?

- Untuk pembatasan ruang pencarian dalam query (supaya tidak terlalu luas).
- Untuk analisis hubungan tingkat lanjut seperti second-degree connection, shortest path, atau pattern discovery.
- Untuk mencegah kinerja lambat pada dataset besar (karena tanpa batas hop bisa sangat berat).

"Hop" dalam konteks graph database berarti berapa banyak langkah atau edge yang harus dilewati untuk sampai ke data tujuan. Traversal multi-hop bisa mahal jika tidak ditangani dengan baik.

Untuk efisiensi:

1. Gunakan batas traversal (depth limit):
   - Batasi kedalaman traversal agar tidak menjelajahi seluruh graph.
     Contoh di Neo4j (Cypher):
     `MATCH (a:User {name: 'Alice'})-[*1..3]->(b)`
     `RETURN b`
     Ini mencari node b yang terhubung ke Alice dalam 1 hingga 3 hop saja.
2. Gunakan indeks pada node awal:
   - Index di property (misal User.name) membuat pencarian node awal lebih cepat.
3. Gunakan pola relasi yang spesifik (typed relationship):
   - Jangan gunakan wildcard seperti [*] jika tidak perlu.
     Contoh:
     `MATCH (a:User)-[:FOLLOWS]->(:User)-[:LIKES]->(p:Post)`
     `RETURN p`
     Ini lebih efisien karena traversal diarahkan.
4. Filter node sebanyak mungkin di awal traversal:
   - Gunakan WHERE untuk membatasi ruang pencarian lebih awal.
5. Gunakan LIMIT untuk menghindari hasil berlebihan:
   - Traversal tanpa batas bisa sangat lambat jika data besar. Gunakan LIMIT saat query eksploratif.
6. Gunakan graph algorithm jika traversal kompleks:
   - Untuk analisis seperti shortest path, centrality, gunakan algoritma yang sudah dioptimasi seperti Dijkstra, PageRank, dll.