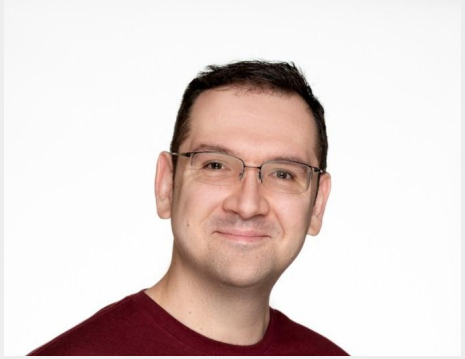
Graphs are Hard

The many pitfalls of working with graphs at scale

Who We Are

John Hopper (@zinic)
Rohan Vazarkar (@CptJesus)



What are we talking about?

- Problems in graph modelling
- Problems in graph scaling
- Lessons Learned from several years of development

SCALE

It’s always going to be bigger than what you think.

Graph Interconnectivity

N number of nodes can contain a maximum of N(N+1) / 2 edges (including loops):

- 10 Nodes - 55 Edges
- 100 Nodes - 5,050 Edges
- 1,000 Nodes - 500,500 Edges
- 10,000 Nodes - 50,005,000 Edges
- 100,000 Nodes - 5,000,050,000 Edges
- …

Interconnectivity and Size

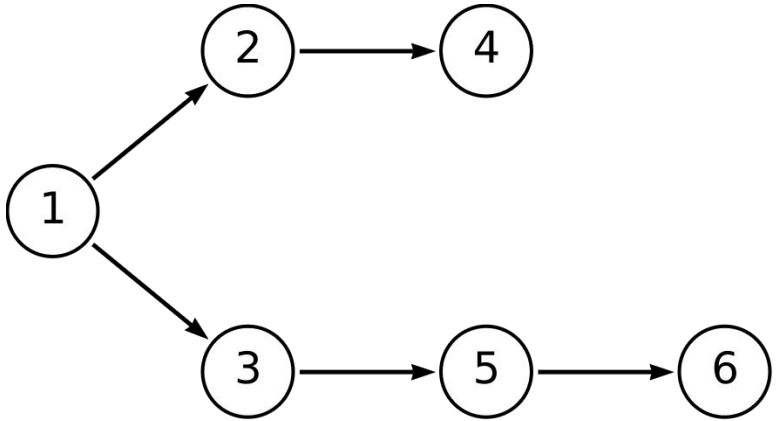
With Nodes and Edges represented by 64-bit Integers, how big is the graph?

- 10 Fully Connected Nodes - 0.001 Megabytes
- 100 Fully Connected Nodes - 0.122 Megabytes
- 1,000 Fully Connected Nodes - 12.020 Megabytes
- 10,000 Fully Connected Nodes - 1,200.000 Megabytes
- 100,000 Fully Connected Nodes - 120,002.000 Megabytes
- …

Interconnectivity and Latency

Naive traversals require as many round trips to your database as the depth of the traversed path space:

- Search path from Node 1 to Node 6
  - Fetch outbound from Node 1
  - Fetch outbound from Nodes 2 and 3
  - Fetch outbound from Nodes 4 and 5



Round Trips are Evil

Access times matter at scale:

- Random access time for Memory is between 50 ns to 150 ns.
- Random access time for SSDs is between 0.08 ms and 0.16 ms.
- Good networks will have a round trip time between 0.5 ms and 0.2 ms.
- Average networks will have a round trip time between 5 ms and 1 ms.

Each query to your database will **always incur** round trip time. On an average network, **1,000 queries in serial** will have a cost of up to **5 seconds** in just network time alone!

Enter the Graph Database

Round trips up and down the latency stack can compound and become expensive so to reduce the need for them, graph databases allow users to write expansion patterns in queries:

Intent: “Expand all outbound paths from from Node 1 fully”

Cypher: *match p = (n)-[*..]->() where id(n) = 1 return p*

The Revisiting Problem

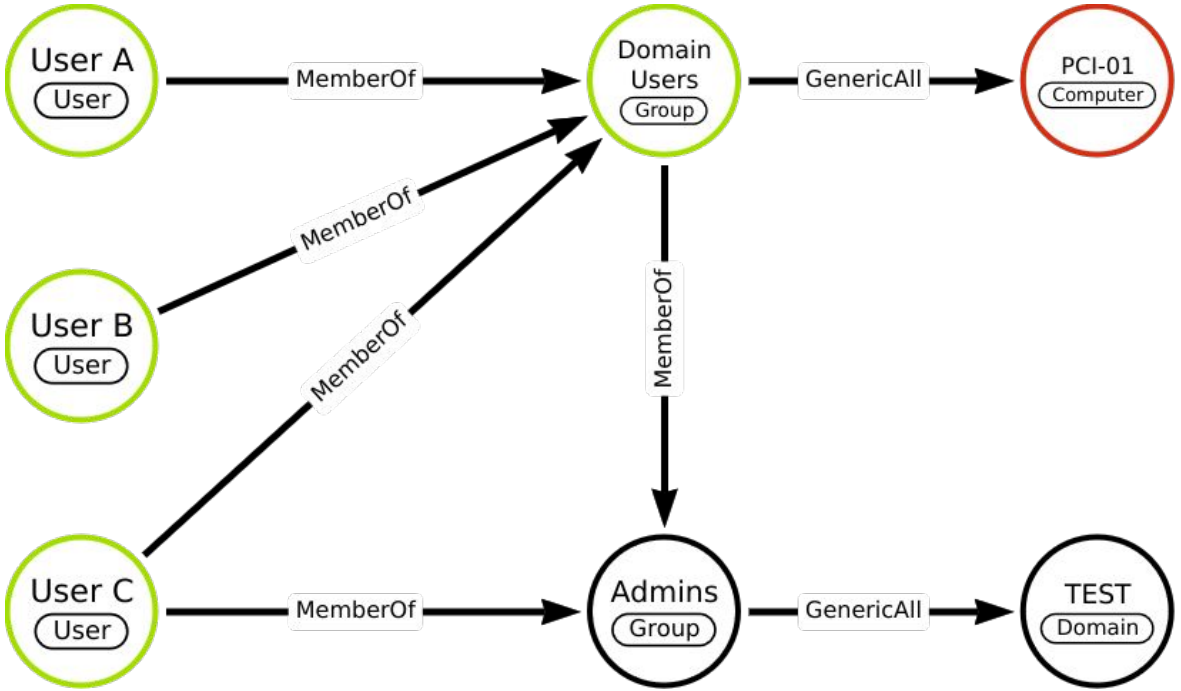
Interconnectivity results in sub-graphs that will often show up in the same traversals:

- Paths that hit common Active Directory groups like **Domain Users**
- Paths that involve an commonly used **RDP Host**
- Paths that explore **GPO Enforcement**
- **…**
- Any other expansion landmine just waiting for you in your graph

What can Control the PCI Database?

Assets that Control PCI-01

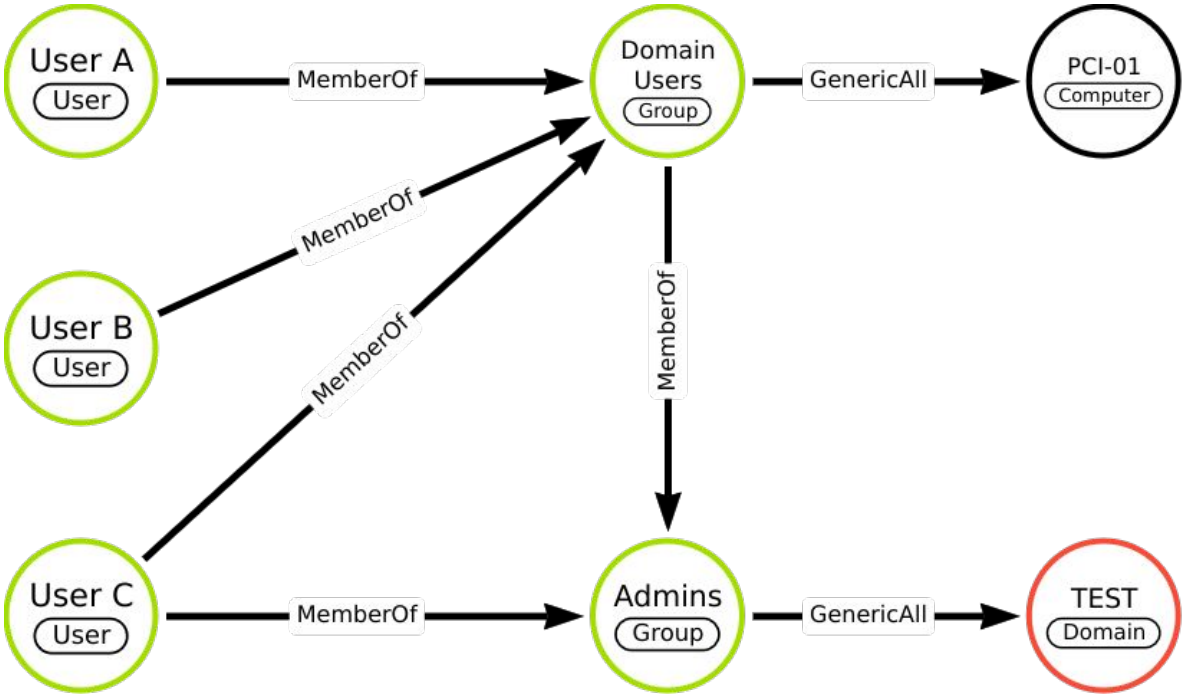
- Domain Users
- User A
- User B
- User C



What can Control the Domain?

Assets that Control TEST

- Domain Users
- Admins
- User A
- User B
- User C



Node Revisiting is a Complexity Problem

- Both traversal examples had to expand the same group: **Domain Users**.
- The database must expand all paths inbound to **Domain Users** for both queries.
- In a domain of **350,000 users** expanding **Domain Users** can become a significant and painful time complexity bottleneck.
- If expanding **Domain Users** takes the database **1 second** to complete, each revisit will incur a similar cost.
- Querying just **60 assets** that require expansion of **Domain Users** will take **1 minute** of query time.

Node Revisiting is also a Cardinality Problem

- Cardinality is the number of elements in a set or other grouping, as a property of that grouping.
- In the traversal example, what assets can control another results in querying the graph for the cardinality of nodes that have inbound paths to the asset being queried.
- In a domain of **350,000 users** expanding the cardinality of **Domain Users** and tracking it requires **significant memory space**.
- Storing **350,000 users** as 64-bit Integers requires **2.67 megabytes** of memory.
- Storing the control cardinality of just **60 assets** that involve **Domain Users** in the above domain requires **160.21 megabytes** of memory.

Tracking Inbound Impact for Common Nodes Part 1

- First pass at storing cardinality used bitmaps.
- For small domains, bitmaps resulted in expedient tracking of unique nodes that could control assets.
- Compressed bitmaps reduced overall memory consumption at minimal CPU cost.
- Strategy worked great for domains with **hundreds of thousands of users**.
- Did compressed bitmaps work for tracking inbound control cardinality for domains with **millions of users**?

NO

We ran out of memory.

Tracking Inbound Impact for Common Nodes Part 2

- Second pass at storing cardinality used hyperloglog.
- HyperLogLog is a probabilistic data structure that estimates the cardinality of a set. As a probabilistic data structure, HyperLogLog trades perfect accuracy for efficient space utilization.
- Solution is accurate to within ~0.5% for domains with greater than **3 million users**.
- Did hyperloglog work for tracking inbound control cardinality for domains with **tens of millions of users**?

YES!

But it still eats up to 62 GB of memory for the largest domains.

Common Strategies at Scale

- Dust off the computer science books, it’s time to get weird with data structures.
- Push as much criteria as you can down to the database to minimize round trips.
- Don’t store what you don’t need. Nodes are happy to be represented as integers. You can always fetch their details in-bulk later.
- Don’t be afraid to eat memory, it’s delicious.
- Choose your database wisely.
- Know your scale.

Real World Examples

ADCS Edges

- ADCS represents one of the most complex pieces of work we've put into the graph
- A very common scenario in ADCS is taking the cross product of 2 sets of node controllers and determining intersections

```
MATCH p1 =
(n:Base)-[:Enroll|GenericAll|AllExtendedRights|MemberOf*1..]->(ct:CertTemplate)-[:PublishedTo]-(ca:EnterpriseCA)-[:IssuedSignedBy*1..]-(rootca:RootCA)-[:RootCAFor]->(d:Domain)

MATCH p2 = (n)-[:Enroll|GenericAll|AllExtendedRights|MemberOf*1..]->(ca)-[:TrustedForNTAuth]->(nt:NTAuthStore)-[:NTAuthStoreFor]-(d)
```

The Easy Way

- Find all the first degree controllers of node A and node B
- Expand all group members recursively
- Find all the intersections between the two
- Make an edge from each principal found this way as an attack path

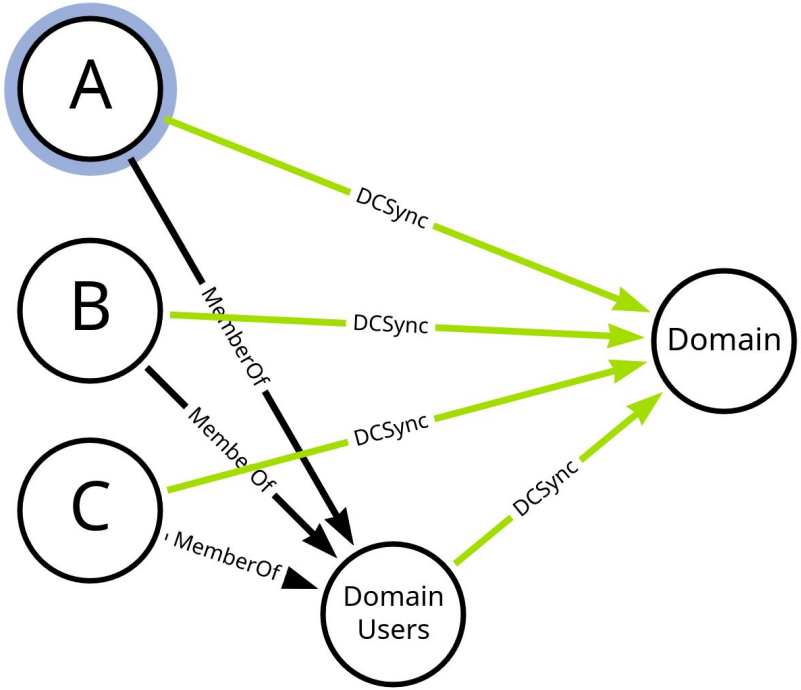
The fallout

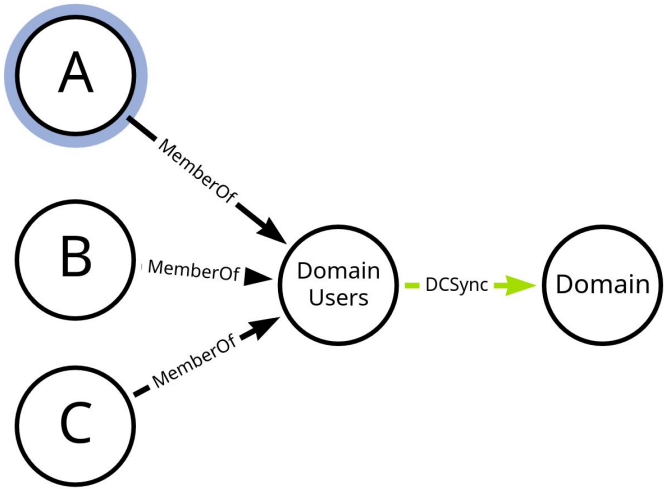
- Initial prototypes of ADCS post processing were blowing out RAM and crashing our instances
- Even successful post processing runs were taking exponentially more time than before and generating insane numbers of edges
- Group membership expansion continued to bite us

The hard, more right way

- Use the compressed bitmap to store all our group membership expansions to reduce expensive traversal time
- Implement shortcutting logic to opportunistically create edges at the highest possible level, minimizing the number of created edges and taking advantage of graph semantics
  - If the Domain Users group has the permission, all the members of the group have the permission via the MemberOf edge in the graph
  - Cost - some accuracy is lost as some ancillary, valid attack paths can be obscured by the shortcutting

Shortcutting - These graphs mean the same thing!





SyncLAPSPassword

- SyncLAPSPassword is a combination of 2 edges, and the previous implementation gave edges for all principals that have both permissions
- Didn’t expect to see all that many edges in a given environment as the permissions are relatively obscure

We were wrong

```
@neo4j> MATCH (:Base)-[r:SyncLAPSPassword]->() RETURN count(r);
+----------+
| count(r) |
+----------+
| 31013155 |
+----------+
```

The fallout

- Vast number of edges caused a huge slowdown in post processing both in creation and in deletion
- Caused massive performance issues on this node and insane memory use

AZResetPassword

- Azure is an absolute landmine for scale and complexity
- Currently creating a massive amount of AZResetPassword edges in environments, which leads to costly insertion and deletion
- Vast numbers of edges are not only expensive, they make understanding the problem for the user almost impossible

Software Stacks and You

Your software stack matters!

- Sometimes, what’s easy is not going to be scalable
- Libraries that start as prototyping convenience can eventually become lead anchors that stifle progress and require major refactors

Neo4j

- Initially chose Neo4j for ease of prototyping in early BloodHound days
- Worked great at smaller scales, for FOSS
- As we've continued to expand scale as well as complexity, we continue to run into issues with Neo4j that require more complex solutions
- Slowly migrating to Postgresql, but tech debt is staggering and requires us to be extremely careful with changes

Regraph

- Incredible graph visualization library for JS with tons of features and great performance
- Used in BHE for visualization
- During BHCE development, learned that we couldn’t use the license in an open source project
- Forced us to choose an entirely different graph viz library for open source and build on top of SigmaJS, consuming dev time to write abstractions

GORM

- Go ORM that promised easy sql use
- Has screwed us more ways than I can count with constant migration issues
- Took an entire week+ long just to move migrations to our own stack, with more work still necessary to detangle complexity

Conclusion

Graphs are Hard

- You’re going to run into unexpected problems…repeatedly
- Fun engineering challenges await you

Questions?