APIARY: A CASE FOR NEO4J?

Anuj Mathur and Dhaval Dalal
{amathur, ddalal}@equalexperts.com

What is Apiary?

The application aims at graphically representing an organization that contains different groups/units of employees. Metaphorically, a group/unit can be viewed as a beehive and an employee as a bee and hence the whole organization becomes an apiary - hence the application name.

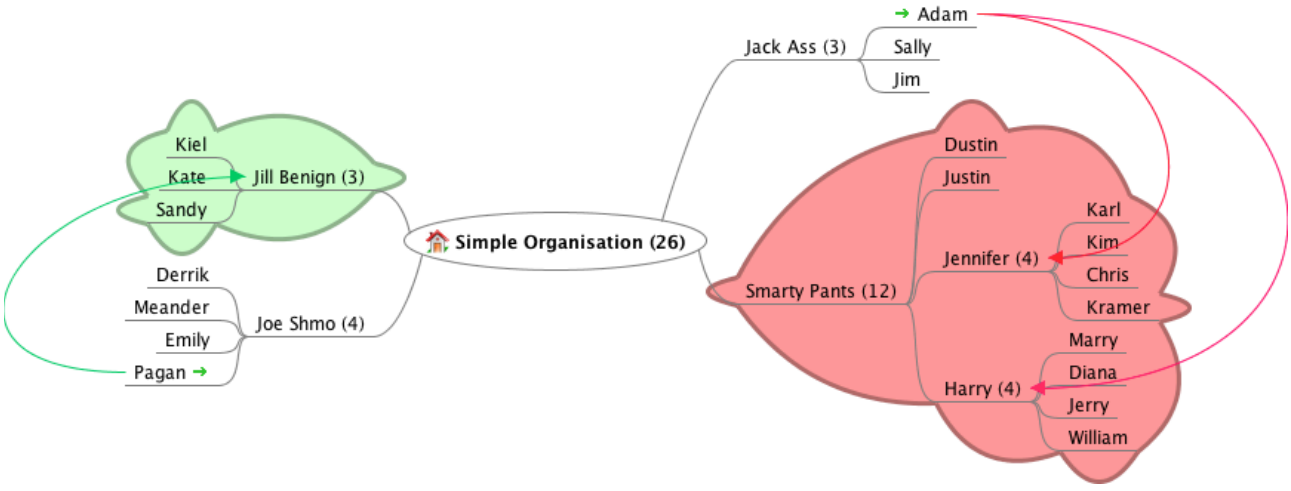
Typical Usecases

1) Modeling a Simple Organization (Phase 1)

Employees belong to single or multiple functional units lead by heads within an organization. An employee can have different types of reporting, direct reporting to his/her manager (can be conceptualized as a thick line relationship - in the figure below the direct lines coming out of a node are colored as grey) and multiple indirect reporting to different department heads/managers (can be conceptualized as a thin/dotted line relationship - in the figure below as green and red colored lines).

For example, in a IT Services company or a captive center, Adam may be reporting directly to Jack Ass, but for his project work, he would be functionally reporting to Jennifer and Harry as they are responsible for Cash Management and Corporate Banking.

As the number of employees increase, so would the relationships and cross-functional reporting, eventually developing into a complex model that is non-hierarchical.



2) Modeling a Group (Phase 2)

Business Groups have many companies serving many sectors. For instance, The Tata Group, BT Group, Reliance Group, BNP Paribas Group etc... would be serving many sectors and have multiple companies in each sector and its sub-sector.

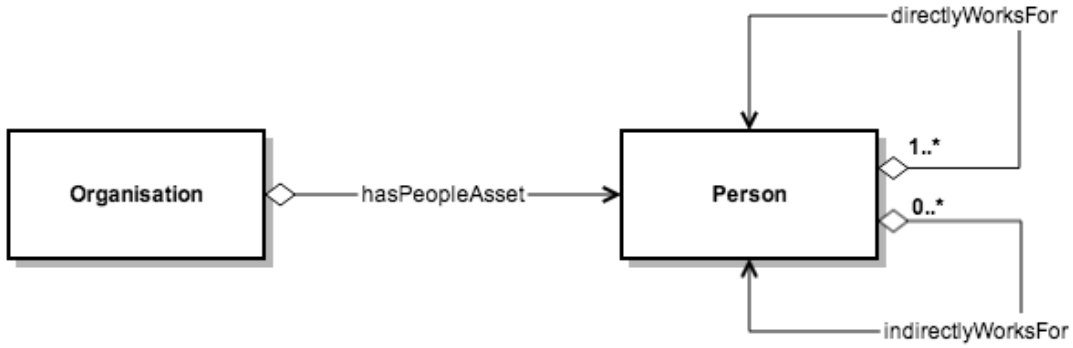
Once we do the above, we plan to add dynamic part to this graph, that is, the social relationships in an organization, to understand - who is talking to whom the most, are the top-level folks reaching out to the ones below and till what level is that effective, recommend people for a project that requires certain set of skills etc, etc... and visualize them for mining interesting organizational patterns.

Screen Renditions and Flows

We intend to render a UI that is similar to the Mindmap shown above, with ability to navigate, expand etc... from current node to other nodes.

1. Login causes user to be fetched (an index hit) from Neo4j and render his/her subordinates with count of people reporting to him/her until a pre-defined visibility level.
2. Ability to navigate up/down the levels.
3. Ability to expand subordinates.
4. Ability to see hierarchy lineage.
5. Dashboard shows count of people at all levels as a summary data.

Analysis Model (Phase 1)



1) Neo4J Domain Model

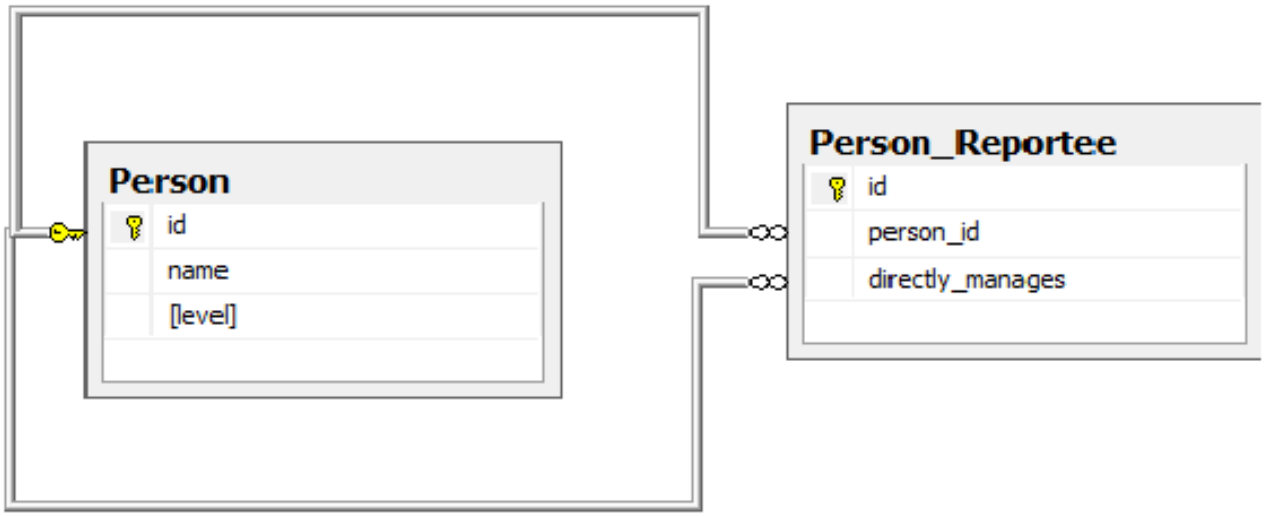
Node	Properties
Person	name
	type
	level

Relationships	Properties
DIRECTLY_MANAGES	N/A

Note: For the purpose of establishing the case, we have modeled minimal relationships and not all the relationships that would have been in the final application. Below is a list of relationships that are yet pending to be modeled, but are not relevant for the purposes of taking performance measurements.

Relationships	Properties
INDIRECTLY_MANAGES	N/A
DIRECTLY_REPORTS_TO	N/A
INDIRECTLY_REPORTS_TO	N/A
MEETS	frequency, where
PREFERS_COMPANY_OF	N/A

2) SQL Domain Model



Queries

Above screen-flow and modeling for organizations and groups use cases requires us to run queries that do the following:

1) Gather Subordinates names till a visibility level from current level.

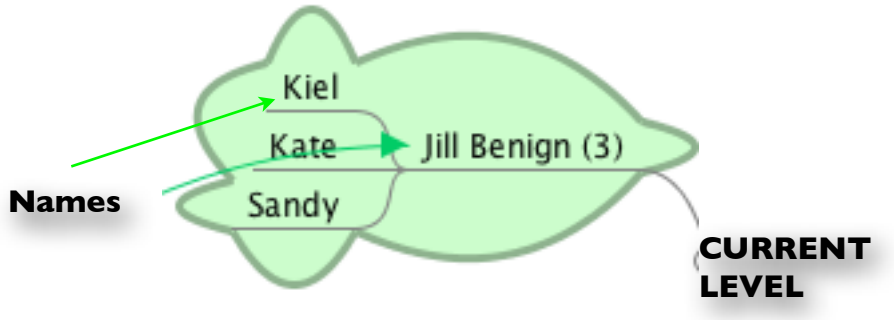
2) Gather Subordinates Aggregate Data from current level.

3) Gather Overall Aggregate Data for Dashboard

We have run these queries at different levels for various volumes of person-data ranging from 1000 people (needed by the organization use case) to 1 million people (needed by the Group use case) in steps of multiples of 10, i.e 1K, 10K, 100K, 1M. We have additionally taken readings for 2M and 3M organization at level 8.

Note: For Neo4j, we have restricted ourselves to Parametric Cypher queries (and not included Traversal API) by executing them and noting the execution time programmatically. This, we believe is a fair apple-to-apple comparison because just as what SQL is to RDBMS, Cypher is to Neo4j. Also, we understand that the Traversal API is likely the fastest at the moment; however, for the longer term as Neo4j intends to further the Cypher query planning and optimization, we think its prudent to stick to it.

1) Gather Subordinates names till a visibility level from a current level



We have varied total hierarchy levels in the organization from 3 to 8 for different volumes of people. Optimized generic Cypher query is:

```
start n = node:Person(name = "fName lName")
match p = n-[:DIRECTLY_MANAGES*1..visibilityLevel]->m
return nodes(p)
```

Where visibility level is a number that indicates the number of levels to show.

For SQL we have to recursively add joins for each level, a generic SQL can be written as:

```
SELECT manager.pid AS Bigboss,
manager.directly_manages AS Subordinate,
L1Reportees.directly_manages AS Reportee1,
L2Reportees.directly_manages AS Reportee2,
...
FROM person_reportee manager
LEFT JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
  LEFT JOIN person_reportee L2Reportees
  ON L1Reportees.directly_manages = L2Reportees.pid
    ...
      ...
WHERE manager.pid = (SELECT id
                     FROM person
                     WHERE name = "fName lName")
```

Visibility Level	Database	Queries

2	Neo4j	`start n = node:Person(name = "fName lName")`
		`match n-[:DIRECTLY_MANAGES*1..2]->m`
		`return nodes(p)`

	MySQL/ MSSQL	`SELECT manager.pid AS Bigboss, manager.directly_manages AS Subordinate, L1Reportees.directly_manages AS Reportee`
		`FROM person_reportee manager`
		`LEFT JOIN person_reportee L1Reportees`
		`ON manager.directly_manages = L1Reportees.pid`
		`WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")`

Visibility Level	Database	Queries

3	Neo4j	`start n = node:Person(name = "fName lName")`
		`match n-[:DIRECTLY_MANAGES*1..3]->m`
		`return nodes(p)`

	MySQL/ MSSQL	`SELECT manager.pid AS Bigboss,`
		`manager.directly_manages AS Subordinate,`
		`L1Reportees.directly_manages AS Reportee1,`
		`L2Reportees.directly_manages AS Reportee2`
		`FROM person_reportee manager`
		`LEFT JOIN person_reportee L1Reportees`
		`ON manager.directly_manages = L1Reportees.pid`
		`  LEFT JOIN person_reportee L2Reportees`
		`  ON L1Reportees.directly_manages = L2Reportees.pid`
		`WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")`

4	Neo4j	`start n = node:Person(name = "fName lName")`
		`match n-[:DIRECTLY_MANAGES*1..4]->m`
		`return nodes(p)`

	MySQL/ MSSQL	`SELECT manager.pid AS Bigboss,`
		`manager.directly_manages AS Subordinate,`
		`L1Reportees.directly_manages AS Reportee1,`
		`L2Reportees.directly_manages AS Reportee2,`
		`L3Reportees.directly_manages AS Reportee3`
		`FROM person_reportee manager`
		`LEFT JOIN person_reportee L1Reportees`
		`ON manager.directly_manages = L1Reportees.pid`
		`  LEFT JOIN person_reportee L2Reportees`
		`  ON L1Reportees.directly_manages = L2Reportees.pid`
		`  LEFT JOIN person_reportee L3Reportees`
		`  ON L2Reportees.directly_manages = L3Reportees.pid`
		`WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")`

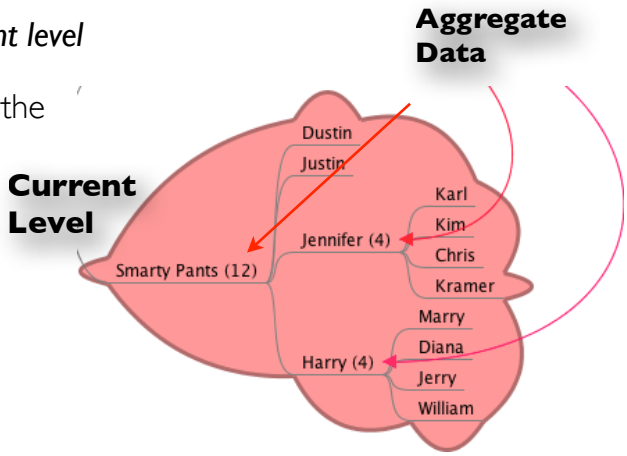
...	...	...

7	Neo4j	`start n = node:Person(name = "fName lName")`
		`match n-[:DIRECTLY_MANAGES*1..7]->m`
		`return nodes(p)`

	MySQL/ MSSQL	`SELECT manager.pid AS Bigboss,`
		`manager.directly_manages AS Subordinate,`
		`L1Reportees.directly_manages AS Reportee1,`
		`L2Reportees.directly_manages AS Reportee2,`
		`L3Reportees.directly_manages AS Reportee3,`
		`...,`
		`L6Reportees.directly_manages AS Reportee6,`
		`L7Reportees.directly_manages AS Reportee7`
		`FROM person_reportee manager`
		`LEFT JOIN person_reportee L1Reportees`
		`ON manager.directly_manages = L1Reportees.pid`
		`  LEFT JOIN person_reportee L2Reportees`
		`  ON L1Reportees.directly_manages = L2Reportees.pid`
		`  LEFT JOIN person_reportee L3Reportees`
		`  ON L2Reportees.directly_manages = L3Reportees.pid`
		`  ...`
		`  LEFT JOIN person_reportee L7Reportees`
		`  ON L6Reportees.directly_manages = L7Reportees.pid`
		`WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")`

2) *Gather Subordinates Aggregate data from current level*

We have varied total hierarchy levels in the organization from 3 to 8 for different volumes of people.



Further, the optimized Generic Cypher query is:

```
start n = node:Person(name = "fName lName")
match n-[:DIRECTLY_MANAGES*0..(totalLevels -
n.level)]->m-[:DIRECTLY_MANAGES*1..(totalLevels
- n.level)]->o
where n.level + visibilityLevel >= m.level
return m.name as Subordinate, count(o) as Total
```

For SQL aggregate query, we not only have to recursively add joins but also perform inner unions for each level till the last level to obtain the aggregate data for that level. Once we obtain the data for a particular level (per person), we perform outer unions to get the final result for all the levels. This results in a very big SQL query.

Here is a sample query that returns aggregate data for 3 levels below the current level. Say, we have current level as 5 with a total of 8 levels, this means I need to aggregate data for each person starting below level 5 until 8 as well as for the person in question (in the where clause).

```
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
  FROM (
    SELECT manager.pid AS directReportees, 0 AS count
    FROM person_reportee manager
    WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")

    UNION

    SELECT manager.pid AS directReportees, count(manager.directly_manages) AS count
    FROM person_reportee manager
    WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
    GROUP BY directReportees

    UNION

    SELECT manager.pid AS directReportees, count(reportee.directly_manages) AS count
    FROM person_reportee manager
    JOIN person_reportee reportee
    ON manager.directly_manages = reportee.pid
    WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
    GROUP BY directReportees

    UNION

    SELECT manager.pid AS directReportees, count(L2Reportees.directly_manages) AS count
    FROM person_reportee manager
    JOIN person_reportee L1Reportees
    ON manager.directly_manages = L1Reportees.pid
        JOIN person_reportee L2Reportees
        ON L1Reportees.directly_manages = L2Reportees.pid
```

INNER UNIONS

```
    WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
    GROUP BY directReportees
  ) AS T
GROUP BY directReportees)

UNION

(SELECT T.directReportees AS directReportees, sum(T.count) AS count
  FROM (
    SELECT manager.directly_manages AS directReportees, 0 AS count
    FROM person_reportee manager
    WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")

    UNION

    SELECT reportee.pid AS directReportees, count(reportee.directly_manages) AS count
    FROM person_reportee manager
    JOIN person_reportee reportee
    ON manager.directly_manages = reportee.pid
    WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
    GROUP BY directReportees

    UNION

    SELECT depth1Reportees.pid AS directReportees,
count(depth2Reportees.directly_manages) AS count
    FROM person_reportee manager
    JOIN person_reportee L1Reportees
    ON manager.directly_manages = L1Reportees.pid
        JOIN person_reportee L2Reportees
        ON L1Reportees.directly_manages = L2Reportees.pid
    WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
    GROUP BY directReportees
  ) AS T
 GROUP BY directReportees)

UNION

(SELECT T.directReportees AS directReportees, sum(T.count) AS count
  FROM(
    SELECT reportee.directly_manages AS directReportees, 0 AS count
    FROM person_reportee manager
    JOIN person_reportee reportee
    ON manager.directly_manages = reportee.pid
    WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
    GROUP BY directReportees

    UNION

    SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages) AS
count
    FROM person_reportee manager
    JOIN person_reportee L1Reportees
    ON manager.directly_manages = L1Reportees.pid
        JOIN person_reportee L2Reportees
        ON L1Reportees.directly_manages = L2Reportees.pid
    WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
    GROUP BY directReportees
) AS T
GROUP BY directReportees)

UNION
```

OUTER UNIONS

```
(SELECT L2Reportees.directly_manages AS directReportees, 0 AS count
 FROM person_reportee manager
 JOIN person_reportee L1Reportees
 ON manager.directly_manages = L1Reportees.pid
    JOIN person_reportee L2Reportees
    ON L1Reportees.directly_manages = L2Reportees.pid
 WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
)
```

If the current level is 1 and total levels are 8, we have these queries run into 500+ lines (and hence we have not shown all of them here). Also, hand-creating them and maintaining them is a painstaking task. Though programatic creation of these queries is possible using nested recursion, it would also be difficult to produce and verify their correctness, we simply have to trust the recursion. In all the above queries for both inner and outer union queries, the maximum number of joins vary as:

```
max no. of joins = total level - current level - 1
```

For Neo4j, it is worth noting that the queries do not exhibit such a bloat as levels increase, instead they remain constant for all levels and at the same time Cypher is more expressive and declarative in nature.

For taking measurements, we have assumed full visibility and the person in question is always at top level (`n.level = 1`) and viewing down the whole hierarchy. However, in the actual application, visibility level would change at runtime. We simply vary the visibility level in the where clause. Please note that Visibility level is to filter out the data to be shown to the user and has no bearing on the aggregate data being calculated.

Visibility Level	Total Levels	Queries

2	3	`start n = node:Person(name = "fName lName")`
		`match n-[:DIRECTLY_MANAGES*0..2]->m-[:DIRECTLY_MANAGES*1..2]->o`
		`where n.level + 2 >= m.level`
		`return m.name as Subordinate, count(o) as Total`

3	4	`start n = node:Person(name = "fName lName")`
		`match n-[:DIRECTLY_MANAGES*0..3]->m-[:DIRECTLY_MANAGES*1..3]->o`
		`where n.level + 3 >= m.level`
		`return m.name as Subordinate, count(o) as Total`

4	5	`start n = node:Person(name = "fName lName")`
		`match n-[:DIRECTLY_MANAGES*0..4]->m-[:DIRECTLY_MANAGES*1..4]->o`
		`where n.level + 4 >= m.level`
		`return m.name as Subordinate, count(o) as Total`

...		...

7	8	`start n = node:Person(name = "fName lName")`
		`match n-[:DIRECTLY_MANAGES*0..7]->m-[:DIRECTLY_MANAGES*1..7]->o`
		`where n.level + 7 >= m.level`
		`return m.name as Subordinate, count(o) as Total`

3) Gather Overall Aggregate Data for Dashboard

We need to show distribution of people at various levels and For this to work we need the following queries.

Neo4J	MySQL/MSSQL
`start n = node(*) return n.level as Level, count(n) as Total order by Level`	`SELECT level, count(id) FROM person GROUP BY level`

Indexes

All the above queries were run with indexing enabled on the join columns for the MySQL and MS-SQL databases and for Neo4j, a Person Index bearing the name field.

Databases Compared

All the Tests were performed on commodity grade laptop - Lenovo T430 with 4GB RAM, running 64-bit Win7 on Intel i5 2.5Ghz with Java 1.7 update 7 was used. Following databases were used to take measurements

- MySQL v5.6.12
- MS-SQL Server 2008 R2
- Neo4J v1.9 (Advanced)

Measurement Tools and Codebase

- For MS-SQL we have used the MS-SQL profiler.
- For MySQL we have noted the duration time (excluding the fetch time) from MySQL Workbench.
- For Neo4j, we ran the parametric queries programmatically (in embedded mode) and measured the time. We have not used Neo4j shell to take measurements as its intended to be an Ops tool (and not a transactional tool).

Our code and scaffolding programs that we used are available on - https://github.com/EqualExperts/Apiary-Neo4j-RDBMS-Comparison

Query Execution and Charts

Further, we have ensured the following across all the 3 databases:

1. Consistent data distribution for each level-cases for all the volumes across all the databases.

2. Functionally equivalent queries for the 3 databases returning same result set.

3. Cold-cache before each query execution for all the databases and re-executed the query to make the cache warm. We have measured execution times for both, Cold and Warm cache.

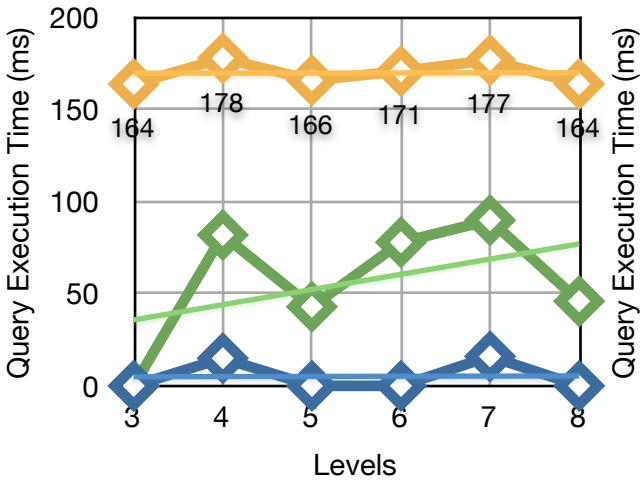
4. Databases and measurement tools running on same machine to avoid network transport times being factored in.

5. We have done an out-of-box comparison for the all the databases without tweaking any memory settings on any of them, apart from giving 3GB to the overall java process running embedded Neo4j.

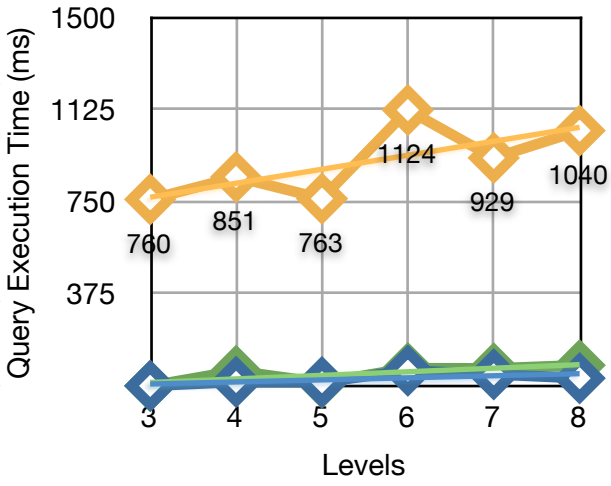
6. We have performed measurements against worst possible queries scenario being executed by the application, that is, say if the top-boss logs in and wants to see all the levels (max. visibility level), his/her query will take the most time.

Volume 1000 People Org																		
	MySQL						MS-SQL						Neo4j					
Lvls	Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
3	78	0	109	0	93	0	226	0	49	0	6	0	679	164	3329	760	1538	269
4	94	15	94	16	78	16	114	82	133	61	35	0	670	178	2878	851	892	183
5	109	0	62	15	47	0	212	43	138	13	59	0	813	166	2848	763	1533	221
6	125	0	156	62	63	0	118	78	278	72	26	0	699	171	3607	1124	1387	319
7	109	16	94	47	62	0	151	90	420	71	13	0	728	177	3123	929	1457	357
8	110	0	141	31	78	0	231	46	621	84	18	0	721	164	2879	1040	906	207

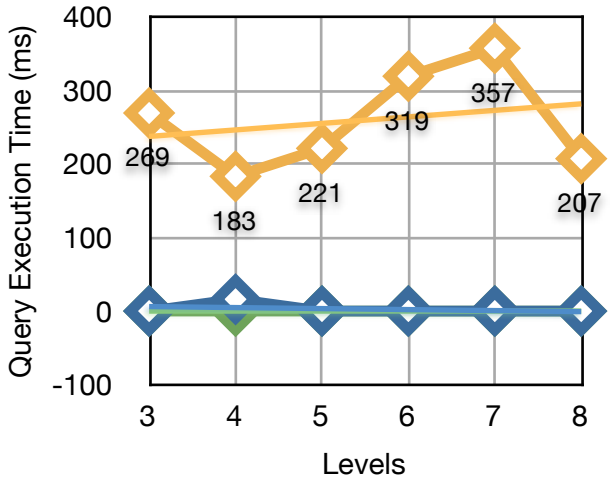
1000 People - Gather Subordinate Names



1000 People - Gather Subordinate Aggregate



1000 People - Overall Aggregate

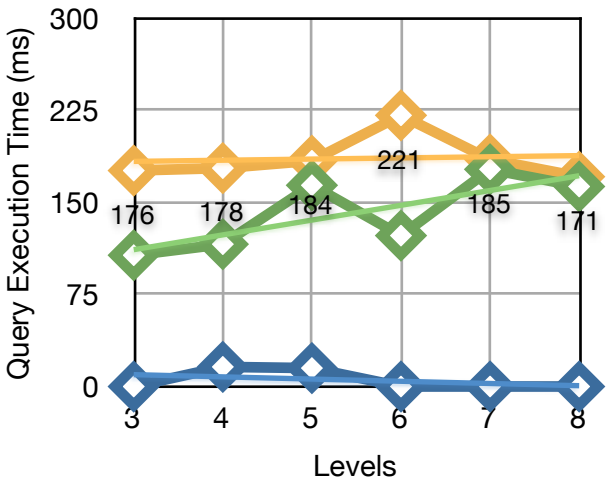


Warm Cache Plots

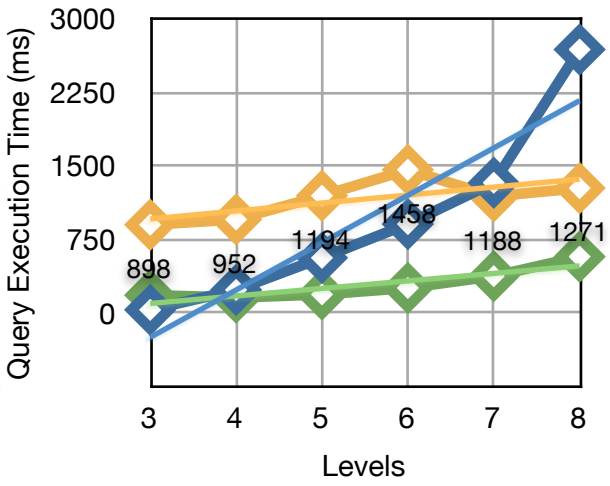
MySQL MSSQL Neo4j

Volume 10K People Org																		
	MySQL						MS-SQL						Neo4j					
Lvls	Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
3	109	0	140	31	109	15	172	107	294	181	103	1	730	176	6129	898	2292	613
4	141	16	343	234	109	0	229	116	1267	156	102	1	776	178	5865	952	2073	455
5	94	15	640	561	94	15	270	164	403	188	327	1	727	184	5718	1194	2119	328
6	94	0	998	905	78	0	401	123	650	254	38	3	742	221	5692	1458	3182	453
7	93	0	1482	1326	78	0	303	177	870	374	68	1	705	185	5710	1188	1990	456
8	93	0	2745	2683	78	0	387	163	1146	573	71	1	713	171	6002	1271	1976	36

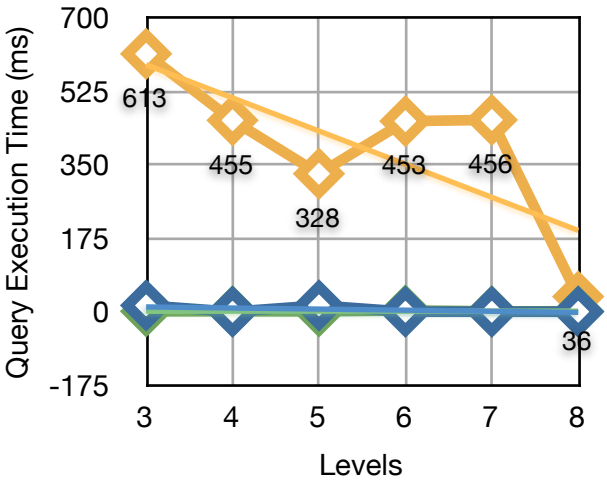
10K People - Gather Subordinate Names



10K People - Gather Subordinate Aggregate



10K People - Overall Aggregate

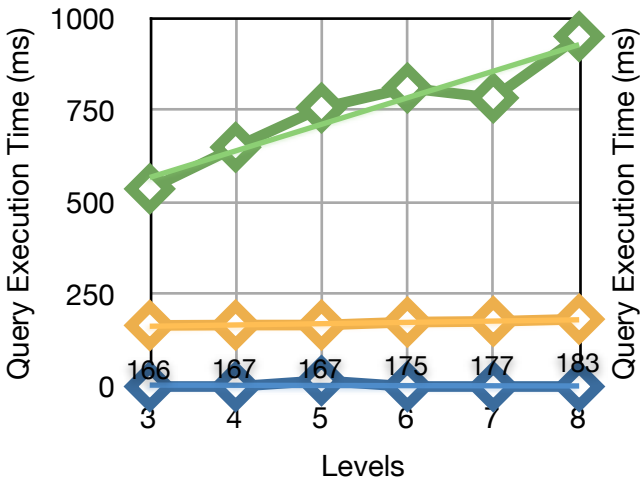


Warm Cache Plots

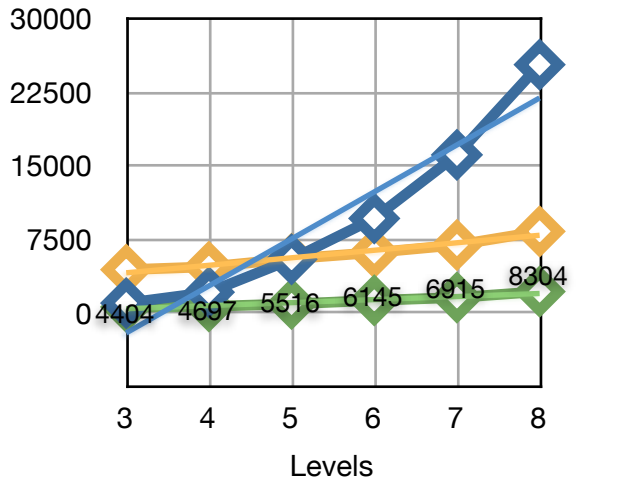
MySQL MSSQL Neo4j

Volume 100K People Org																		
	MySQL						MS-SQL						Neo4j					
Lvls	Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
3	124	0	1107	998	140	31	617	537	1201	608	168	12	736	166	13340	4404	5157	566
4	109	0	2231	2106	140	31	1224	650	1263	667	157	19	812	167	14686	4697	5396	594
5	124	16	5616	5460	125	31	1377	757	1830	913	194	12	713	167	15351	5516	5118	497
6	109	0	9734	9625	140	31	1006	811	3002	1274	159	12	756	175	17056	6145	4736	573
7	93	0	16271	16100	156	31	1125	784	2947	1566	158	14	700	177	17821	6915	4592	592
8	109	0	25334	25287	125	32	1285	951	4256	2171	510	14	691	183	19881	8304	4731	557

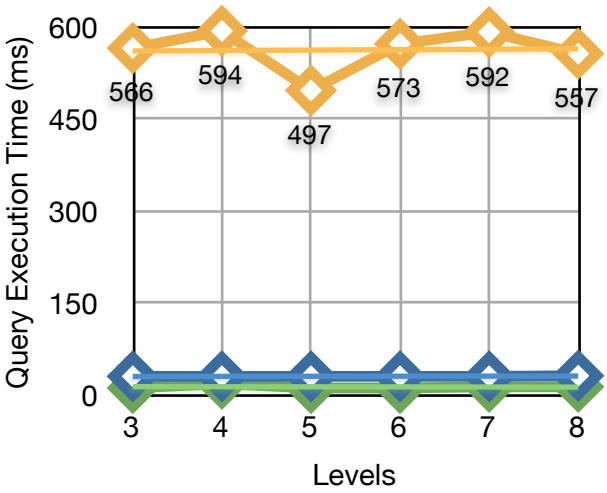
100K People - Gather Subordinate Names



100K People - Gather Subordinate Aggregate



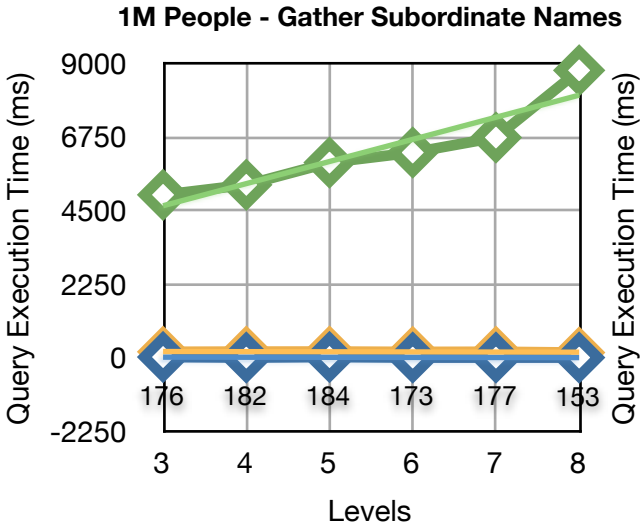
100K People - Overall Aggregate

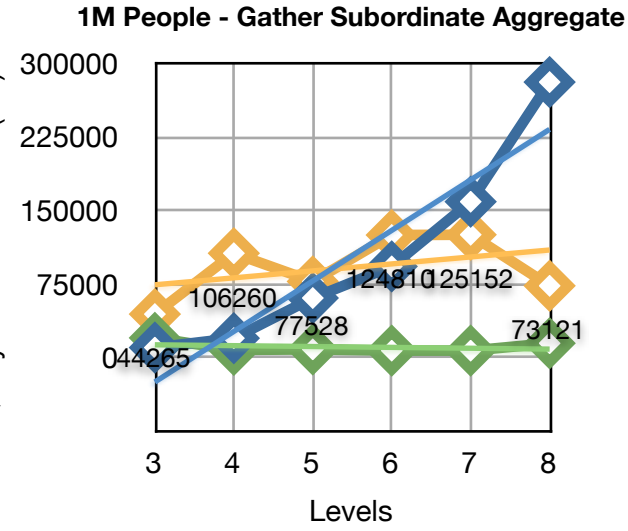


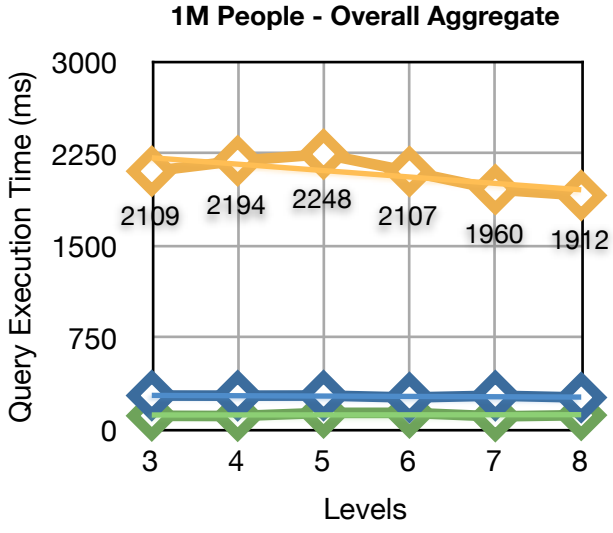
Warm Cache Plots

MySQL MSSQL Neo4J

Volume 1M People Org																		
	MySQL						MS-SQL						Neo4j					
Lvls	Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
3	109	16	10998	10171	452	281	6337	4973	22703	20037	798	116	718	176	109432	44265	3719	2109
4	172	0	20467	19734	515	280	5677	5280	14888	7813	1245	114	740	182	115011	106260	3603	2194
5	172	16	60606	60653	531	281	7111	5959	12047	8293	1211	140	721	184	169123	77528	3406	2248
6	141	0	92789	92867	483	265	7278	6266	15226	7076	1251	140	709	173	255813	124810	3301	2107
7	218	0	160151	158793	499	280	11890	6724	15226	7076	1330	111	700	177	171607	125152	3281	1960
8	265	0	281301	280630	577	265	11300	8777	20560	14564	976	121	835	153	136432	73121	2367	1912







Warm Cache Plots

MySQL MSSQL Neo4J

For 2M org and 3M org we have taken measurements just for Level 8.

Volume 2M People Org																		
	MySQL						MS-SQL						Neo4j					
Lvl	Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
8	281	0	683425	690664	1061	609	26349	21494	40561	30860	1676	203	822	149	437628	1423963	49019	4931

Volume 3M People Org																		
	MySQL						MS-SQL						Neo4j					
Lvl	Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate		Gather Subordinate Names		Gather Subordinate Aggregate		Overall Aggregate	
	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold	Warm
8	234	0	1089355	1093457	1779	1030	44495	31326	84092	75980	2387	282	744	148	1252647	578434	69751	4682

Our Observations

1. For Gather Subordinate Names query running in Neo4j, we have observed significant improvements in query execution times upon running different queries on the same dataset when the cache is warm and hot typically executes within 1-5 ms. However, MS-SQL does not seem to perform well (as we return large data-set), warm cache results are marginally better than cold cache results. Looks like, the join explosion seems to have out-grown the cache size, resulting in page faults. MySQL has out-performed here[1].

2. For Gather Subordinate Aggregate queries, Neo4j's performance is at par or in many cases even better than MySQL. For MySQL, data suggests an exponential increase in execution times with increasing levels, while MS-SQL seems to be still coping with this quite nicely and performing better than all. But all in all, the query execution times for all the databases are quite high as the volumes grow. This demands a different approach to gather such data.

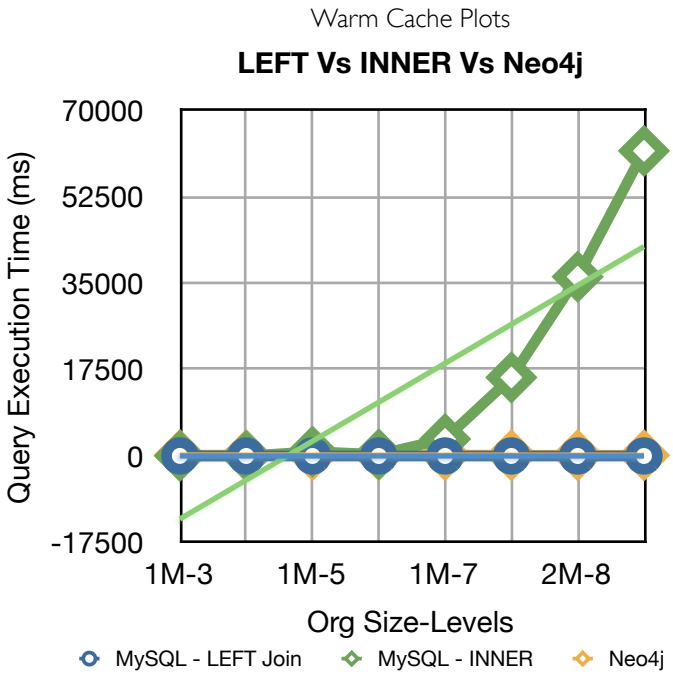
3. For Overall Aggregate Query, it is clear that Neo4j takes quite some time as global graph queries (`node(*)`) become expensive as the volume of data increases as they involve all the nodes, similar to a table scan in RDBMS'. However, RDMBS' are able to cope up quite nicely for such queries, esp. involving aggregates. This is inline with Neo4j recommendation that - it is not meant to do aggregation though it supports it, because its primarily meant for graph traversals.

[1] read this in light of point #4 of this section

4. One of the important observations that we have made is that LEFT join in MySQL is faster as compared to INNER join (contrary to our thinking - LEFT join would be more expensive as compared to INNER). As all the above queries are functionally equivalent, it would not be fair to use INNER join for taking measurements. However, we could not resist the curiosity to check out INNER join performance.

Executing the above queries using inner join on 1M (all levels), 2M and 3M (for level 8), we found a significant increase in query execution time for MySQL. Below are the results of removing the LEFT join from the Gather Subordinate Names query for 1M, 2M Level 8, and 3M Level 8.

	MySQL				Neo4j	
Org Size-Levels	*LEFT JOIN*		*INNER JOIN*			
	cold	*warm*	*cold*	*warm*	*cold*	*warm*
1M-3	109	16	156	16	718	176
1M-4	172	0	140	0	740	182
1M-5	172	0	1154	874	721	184
1M-6	141	0	531	312	709	173
1M-7	218	0	3760	3432	700	177
1M-8	265	0	17145	15896	835	153
2M-8	281	0	40061	36301	822	149
3M-8	234	0	60466	61776	744	148



What does this mean to us or any application? Say, if the situation demanded that we change the LEFT join to INNER join, suddenly we would find ourselves in a bad position performance wise. Whereas its important to note that Neo4j's performance is almost constant time execution for all data sizes.

For the above measurements, we did not factor-in INDIRECTLY_MANAGES relationship, however, taken into consideration, it would translate to additional join for SQL, thus bloating the already bloated query and increasing complexity. This will further degrade the performance as join selection would now take more time.

Few Concluding Thoughts

1. Performance of Neo4j queries is best when all the nodes and relationships are in object cache. For that we need to make sure that we can accommodate the complete graph in memory (esp. for the global graph queries) by throwing a good amount of memory to it. This characteristic without doubt, applies to all other databases equally, whether its RDBMS or NoSQL stores.

2. In order to take advantage of the cache, upon application start-up, we would make the cache hot by issuing queries that bring in almost all the graph into memory, so that the queries that return the sub-graph or the entire graph would be performant. This way the first request serviced by the application is also responsive. If the first request is not that critical, then one can fill the cache lazily.

3. It is better to run Neo4j in embedded mode to get max performance and use parameteric Cypher queries that can take advantage of paying the parsing cost once and caching the execution plan. Though in server mode the above is still valid, we would incur additional data transfer time. An important point that Jim brought to our notice is - in embedded mode the application itself can produce a lot of garbage thus keeping the GC busy (and making the app unresponsive), where as in the server mode, Neo4j takes care of not aggravating JVM in such a way.

4. We intend to model the application using Neo4j as not only the results for it look good when coping with scale, but the kind of flexibility that we would get when adding additional properties to a node/relationship or adding a different type of node/relationship all together, thus giving Apiary the ability to cope with the evolving needs of the organization at different times. With RDBMS', it would be a pain to manage these varietal data needs.

5. We do not rule out the possibility of using a non-graph database alongside Neo4j to model parts of the application best suited for each, esp. for Gather Subordinate Aggregate and Overall Aggregate queries as the readings suggest. Again, we need to carefully plan this part due to its OLAPish nature. One thought that crosses the mind is to use a separate thread that runs on the data, calculating the aggregates after CUD operations are performed. This can then be made available as a service for the application to consume. Second thought is to use a separate mirrored database from where this need can be fulfilled. In either cases, the aggregate data available to the application would be latent. As Jim points out Polyglot persistence is already a new normal, and many implementations use Neo4j alongside non-graph stores like RDBMS or NoSQL.

Asking Questions For a Graph Solution

Based on the experiments above, in order to determine whether Neo4j applies to a situation, we have come up with few asking questions as a generic guideline to ascertain whether graph solution would deem fit:

1. Is your data connected?

2. Or does your domain naturally gravitate towards lots of joins leading to explosion with large data?

    a. For example: Making Recommendations

3. Or are you finding yourself writing complex SQL queries?

    a. For example: recursive joins or large number of joins.

If the answers to any of the above questions is yes, then its potentially a case for a graph database like Neo4j.

Acknowledgements

We would like to thank Dr. Jim Webber from the Neo4j team for going through the report carefully and giving us important feedback along the way.