APIARY: A CASE FOR NEO4J?

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

What is Apiary? 3

Typical Usecases 3

1) Modeling a Simple Organization (Phase 1) 3

2) Modeling a Group (Phase 2) 3

Screen Renditions and Flows 4

Analysis Model (Phase 1) 5

1) Neo4J Domain Model 5

2) SQL Domain Model 6

Queries 6

1) Gather Subordinates names till a visibility level from a current level 7

2) Gather Subordinates Aggregate data from current level 9

3) Gather Overall Aggregate Data for Dashboard 12

Indexes 12

Query Execution Comparison Charts 12

Our Observations 17

Questions/Mullings... 17

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.

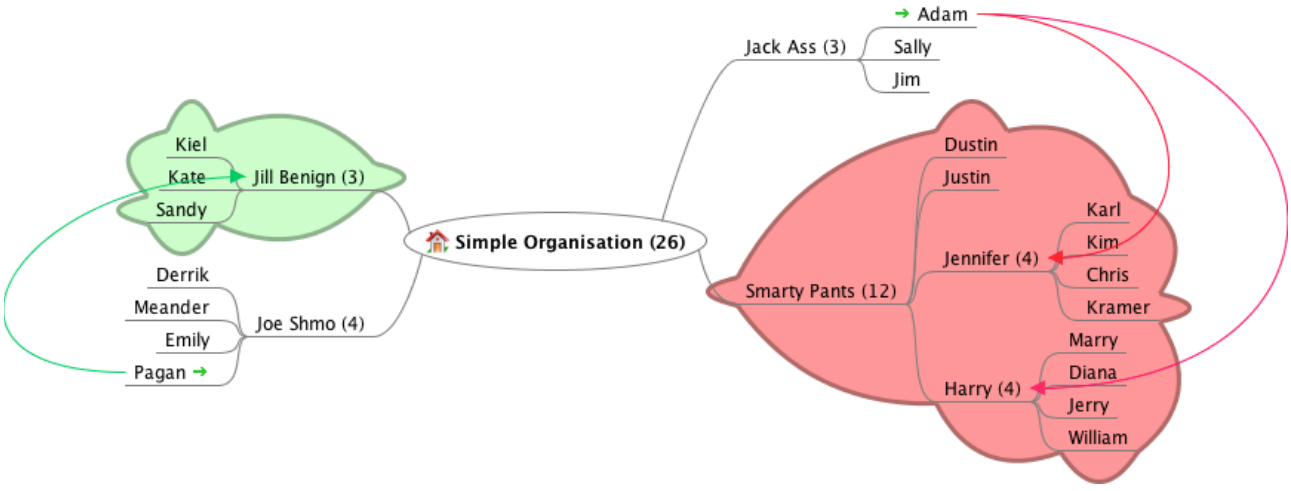
Typical Usecases

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.

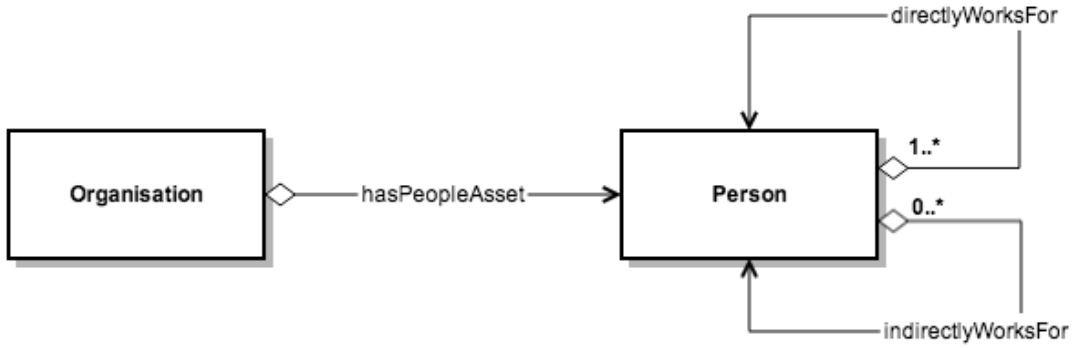
Once we do the above, we plan to add dynamic part to this graph, that is, the social relationships in an organization, to understand - who is talking to whom the most, are the top-level folks reaching out to the ones below and till what level is that effective, recommend people for a project that requires 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

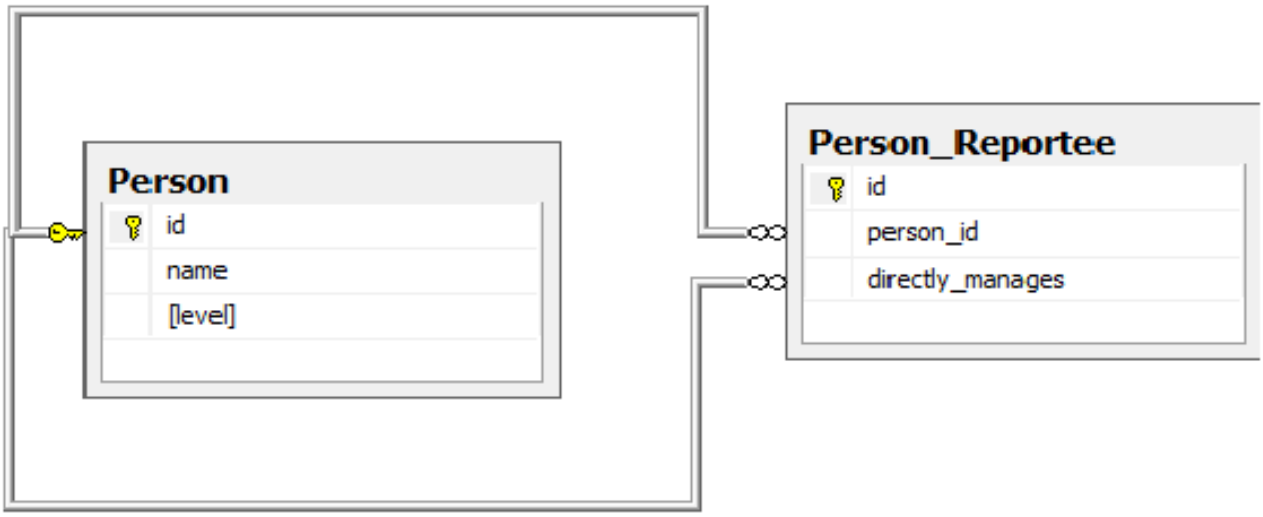
Node	Properties
Person	name
	type
	level

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.

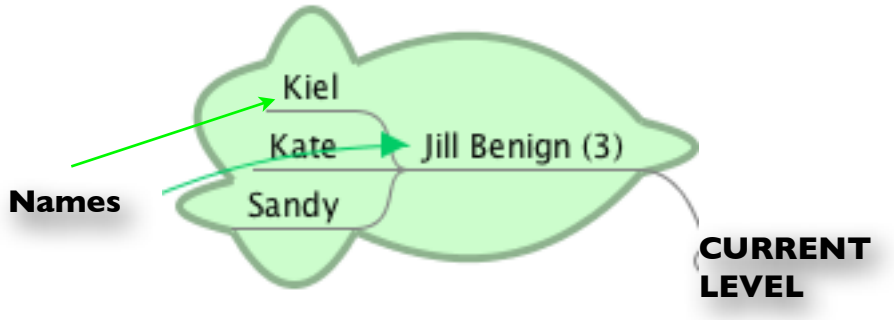
2) Gather Subordinates Aggregate Data from current level.

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.

Note: For Neo4J, we have restricted ourselves to Cypher queries (and not included Traversal using API) by executing them using the Neo-Shell and noting the execution time. Our understanding is that shell shows the actual query execution and does not take in to account the time taken for the records to display. We have not compared Cypher queries with API Traversal queries, as we intend to do that comparison after we establish a case for Neo4J.

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. Generic Cypher query is:

```
start n = node:Person(name = "fName lName")
match p = n-[?*1..visibilityLevel]->m
return nodes(p)
order by length(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-[?*1..2]->m`
		`return nodes(p)`
		`order by length(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-[?*1..3]->m`
		`return nodes(p)`
		`order by length(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-[?*1..4]->m`
		`return nodes(p)`
		`order by length(p)`

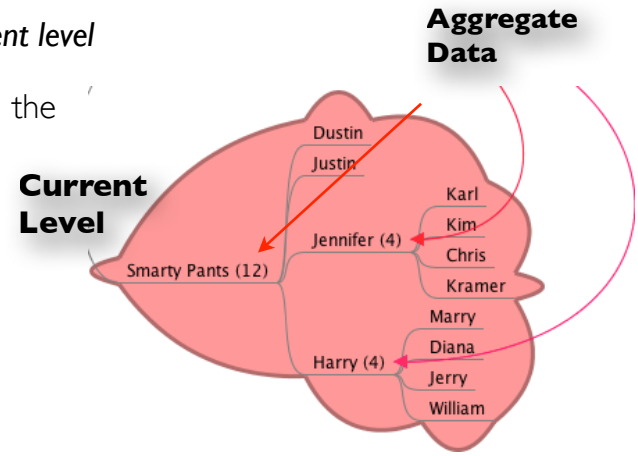
	MySQL/ MSSQL	`SELECT manager.pid AS Bigboss,`
		`manager.directly_manages AS Subordinate,`
		`L1Reportees.directly_manages AS Reportee1,`
		`L2Reportees.directly_manages AS Reportee2,`
		`L3Reportees.directly_manages AS Reportee3`
		`FROM person_reportee manager`
		`LEFT JOIN person_reportee L1Reportees`
		`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")`

...	...	...

8	Neo4J	`start n = node:Person(name = "fName lName")`
		`match n-[?*1..8]->m`
		`return nodes(p)`
		`order by length(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 and have assumed that the maximum levels are 8.

Generic Cypher query is:

```
start n = node:Person(name = "fName lName")
match n-[?*0..(totalLevels - 1)]->m-[?*1..
(totalLevels - 2)]->o
where n.level + visibilityLevel >= m.level
return n.name as BigBoss, m.name as Subordinate,
m.level as SubordinateLevel, count(o) as Total
order by SubordinateLevel
```

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).

INNER UNIONS

```
(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
```

```
    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.

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	**Queries**

2	`start n = node:Person(name = "fName lName")`
	`match n-[?*0..7]->m-[?*1..6]->o`
	`where n.level + 2 >= m.level`
	`return n.name as BigBoss, m.name as Subordinate, m.level as SubordinateLevel,`
	`count(o) as Total`
	`order by SubordinateLevel`

3	`start n = node:Person(name = "fName lName")`
	`match n-[?*0..7]->m-[?*1..6]->o`
	`where n.level + 3 >= m.level`
	`return n.name as BigBoss, m.name as Subordinate, m.level as SubordinateLevel,`
	`count(o) as Total`
	`order by SubordinateLevel`

4	`start n = node:Person(name = "fName lName")`
	`match n-[?*0..7]->m-[?*1..6]->o`
	`where n.level + 4 >= m.level`
	`return n.name as BigBoss, m.name as Subordinate, m.level as SubordinateLevel,`
	`count(o) as Total`
	`order by SubordinateLevel`

...	...

8	`start n = node:Person(name = "fName lName")`
	`match n-[?*0..7]->m-[?*1..6]->o`
	`where n.level + 8 >= m.level`
	`return n.name as BigBoss, m.name as Subordinate, m.level as SubordinateLevel,`
	`count(o) as Total`
	`order by SubordinateLevel`

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 in the SQL databases and in Neo4J, a Person Index bearing the name field.

Query Execution Comparison Charts

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)

For MS-SQL we have used the MS-SQL profiler and for MySQL we have noted the duration time (excluding the fetch time) from MySQL Workbench. For Neo4J, we ran the queries in Neo-Shell (localserver and 2GB heap for shell) and noted the time. 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.

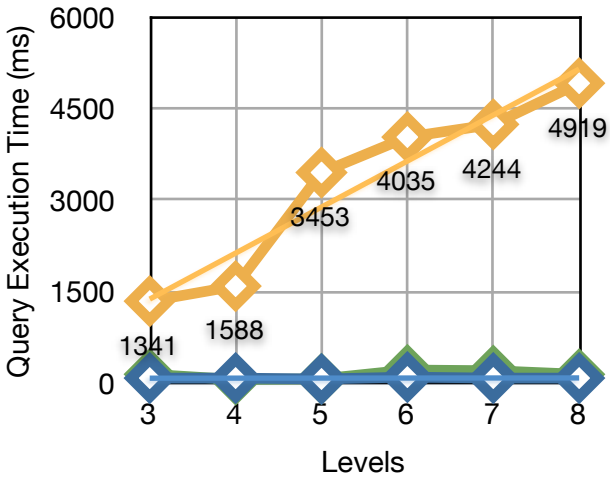
2. Cold-cache before each query execution for all the databases.

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

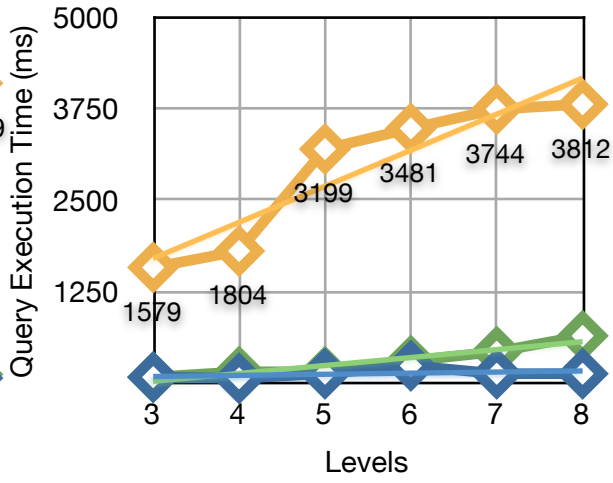
4. We have performed measurements against worst possible queries scenario being executed by the application, that is, say if the top-boss logs in and wants to see all the levels (max. visibility level), his/her query will take the most time.

Volume 1000 People Org									
Levels	MySQL			MS-SQL			Neo4J		
	Gather Subordinate Names	*Gather Subordinate Aggregate*	*Overall Aggregate*	*Gather Subordinate Names*	*Gather Subordinate Aggregate*	*Overall Aggregate*	*Gather Subordinate Names*	*Gather Subordinate Aggregate*	*Overall Aggregate*
3	78	78	31	141	78	62	1341	1579	1226
4	78	62	47	54	151	72	1588	1804	1214
5	63	125	46	65	151	51	3453	3199	1203
6	94	234	32	211	302	65	4035	3481	1221
7	78	125	47	199	426	61	4244	3744	1529
8	78	125	47	140	643	39	4919	3812	1559

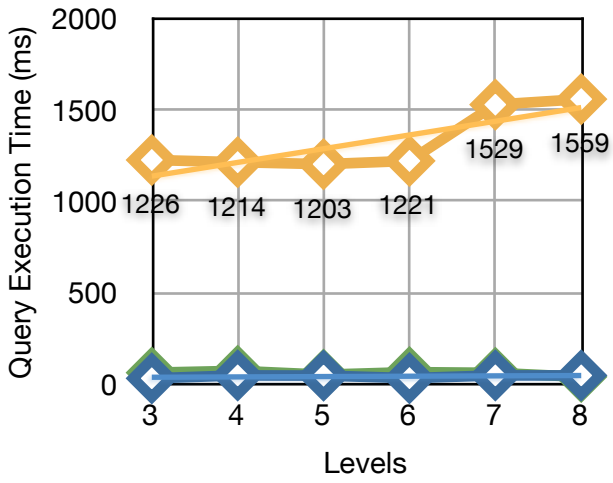
1000 People - Gather Subordinate Names



1000 People - Gather Subordinate Aggregate



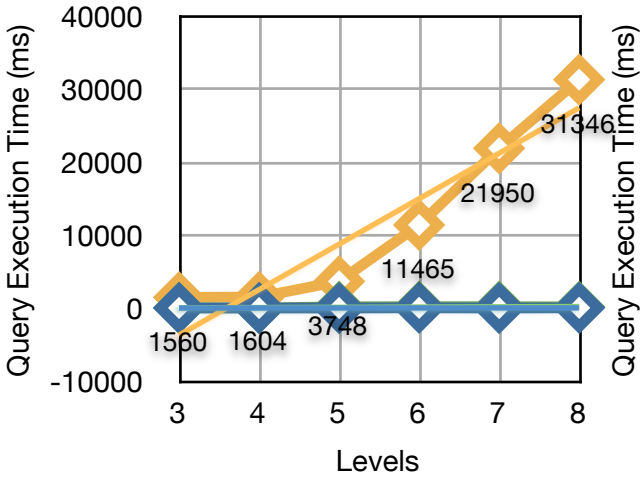
1000 People - Overall Aggregate



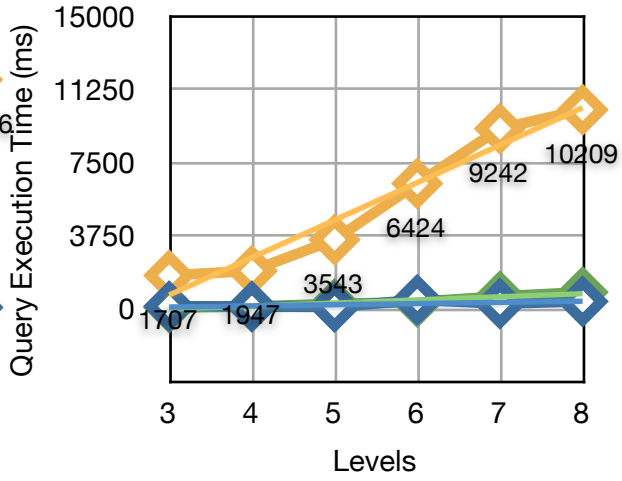
MySQL MSSQL Neo4J

Volume 10000 People Org									
Levels	MySQL			MS-SQL			Neo4J		
	Gather Subordinate Names	*Gather Subordinate Aggregate*	*Overall Aggregate*	*Gather Subordinate Names*	*Gather Subordinate Aggregate*	*Overall Aggregate*	*Gather Subordinate Names*	*Gather Subordinate Aggregate*	*Overall Aggregate*
3	109	125	78	119	65	46	1560	1707	2959
4	94	110	62	105	118	53	1604	1947	2299
5	93	93	78	267	239	79	3748	3543	2352
6	125	452	94	283	347	61	11465	6424	2328
7	94	265	94	278	633	67	21950	9242	2281
8	125	375	94	254	848	73	31346	10209	2358

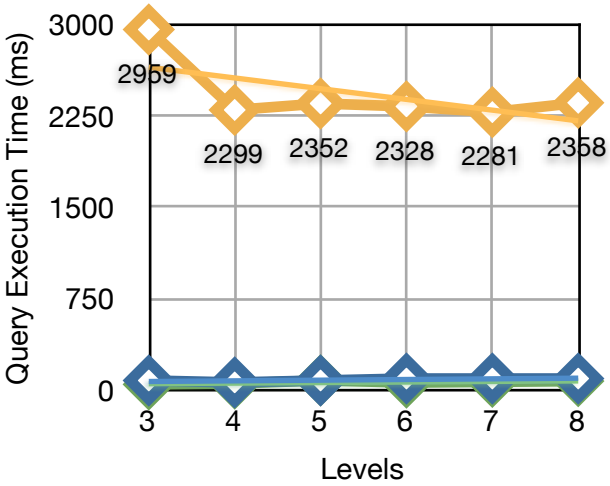
10K People - Gather Subordinate Names



10K People - Gather Subordinate Aggregate



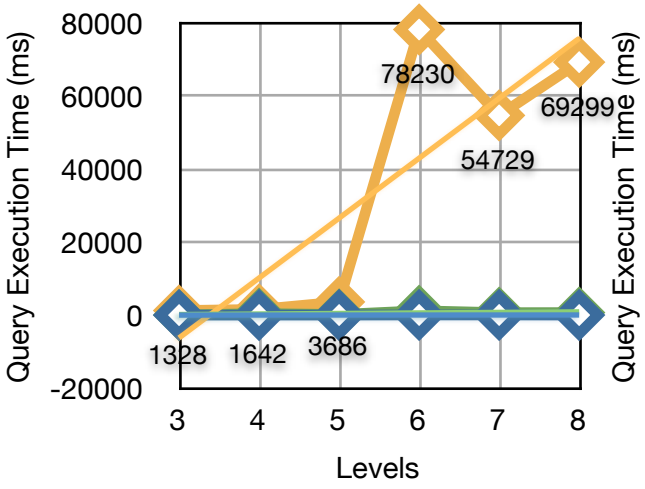
10K People - Overall Aggregate



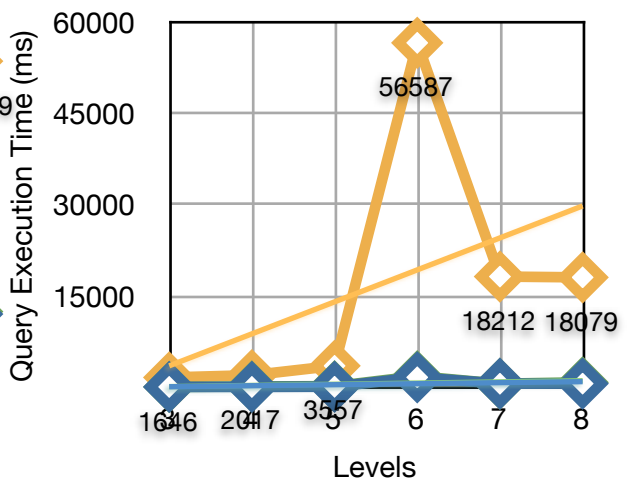
MySQL MSSQL Neo4J

Volume 100000 People Org									
Levels	MySQL			MS-SQL			Neo4J		
	Gather Subordinate Names	*Gather Subordinate Aggregate*	*Overall Aggregate*	*Gather Subordinate Names*	*Gather Subordinate Aggregate*	*Overall Aggregate*	*Gather Subordinate Names*	*Gather Subordinate Aggregate*	*Overall Aggregate*
3	78	93	172	112	89	136	1328	1646	4433
4	93	94	172	484	168	159	1642	2017	4409
5	93	125	140	253	213	182	3686	3557	4504
6	125	1669	219	1271	1849	127	78230	56587	4271
7	124	624	140	750	617	166	54729	18212	4269
8	94	640	219	773	848	175	69299	18079	4828

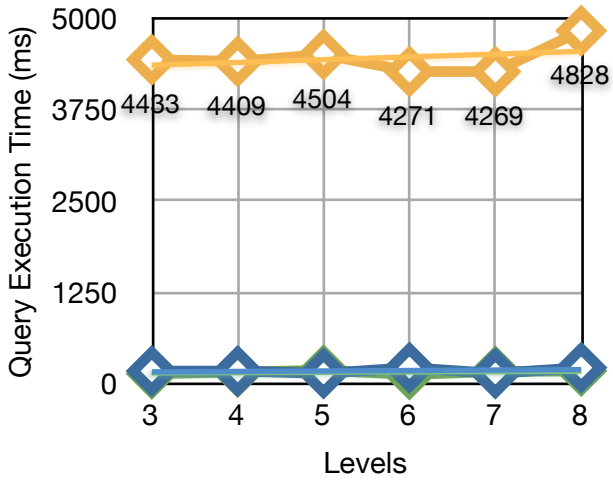
100K People - Gather Subordinate Names



100K People - Gather Subordinate Aggregate



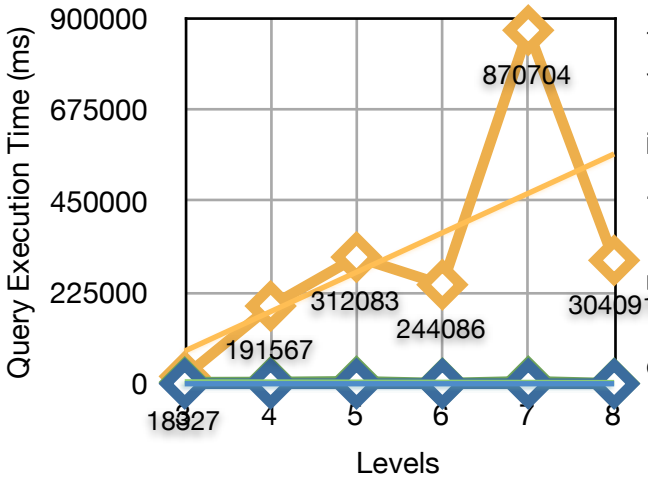
100K People - Overall Aggregate



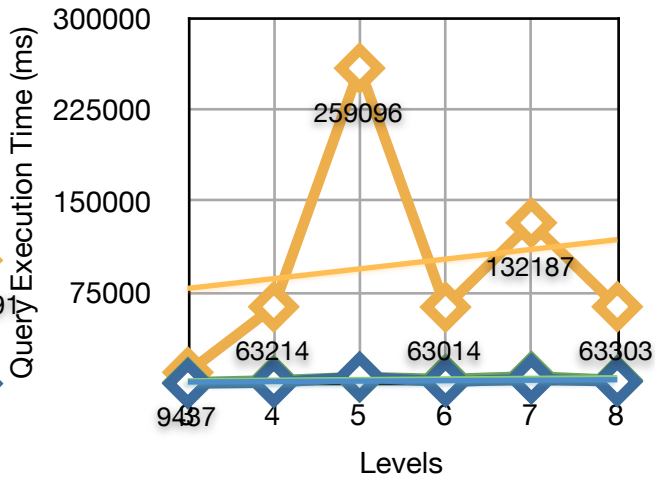
MySQL MSSQL Neo4J

Volume 1000000 People Org									
Levels	MySQL			MS-SQL			Neo4J		
	Gather Subordinate Names	*Gather Subordinate Aggregate*	*Overall Aggregate*	*Gather Subordinate Names*	*Gather Subordinate Aggregate*	*Overall Aggregate*	*Gather Subordinate Names*	*Gather Subordinate Aggregate*	*Overall Aggregate*
3	203	249	546	3961	1187	960	18327	9437	30342
4	156	1030	562	4387	2783	839	191567	63214	27765
5	156	5694	1092	6141	5563	973	312083	259096	26658
6	234	1529	749	1582	3816	1428	244086	63014	35285
7	203	4071	702	6309	6107	953	870704	132187	31170
8	140	1872	499	1204	3062	1018	304091	63303	32961

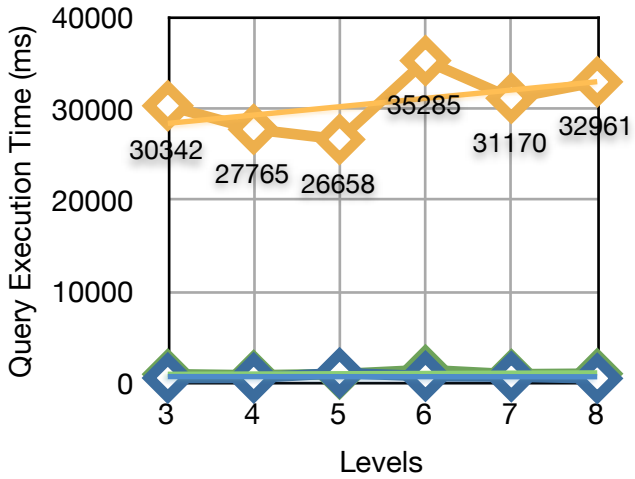
1M People - Gather Subordinate Names



1M People - Gather Subordinate Aggregate



1M People - Overall Aggregate



MySQL MSSQL Neo4J

Our Observations

1. For Gather Subordinate Names query, in almost all the cases after level 4 or 5, we have observed a significant rise in query execution times. Overall, Neo4J is taking longer times roughly by a factor of 10, regardless of the people volume. So, how do we justify these high query execution times?

2. From the above measurements on Overall Aggregate Queries, it is clear that Neo4J takes quite some time as `node(*)` queries in Neo4J 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.

3. For Gather Subordinate Aggregate query, though it does not involve `node(*)`, but still involves an aggregate function `count()`, it takes more time as compared to overall aggregate queries. The possible explanation for this could be the path traversals that the match clause performs. Number of paths traversed have a direct co-relation with time, in other words, wider hierarchies take larger time as compared to deeper hierarchies.

4. With Neo4J, we have observed significant improvements in query execution times upon running different queries on the same dataset when the cache is hot. This behavior is inline with RDBMS'.

Questions/Mullings...

1. We have done an out-of-box comparison for the all the databases, well almost, except for throwing 2G memory to Neo-shell. After looking at various performance tuning parameters, we can try to upgrade physical memory to 8G and have more available to Neo4J. Further, we can also enable memory-mapped settings.

2. Should we consider using two types of databases simultaneously, that is, use RDBMS and run Neo4J in embedded mode, to play on strengths of each? We can then have RDBMS to be the golden source of data, and as the application comes up, seed the embedded mode Neo4J from it. For queries that run faster in Neo4J send to it and the queries that are performant in RDBMS redirect them there. The cons with the embedded mode Neo4J could lead to non-scalability and SPOF. However, we think, these can be addressed by factoring out Neo4J as a separate service relying upon Neo4J clustering for HA.

3. In production, caches will not be cold except upon start-up, but can we expect similar query response times like the ones reported here when the cache gets invalidated as soon as data is either inserted, deleted or updated?

We think that Apiary could be a case for Neo4J and we seek your guidance and input to throw light on things that we may have misunderstood or did not get it right.