NODES 2019

Neo4j Online Developer Expo and Summit

neo4j.com/online-summit

Track #1, 4:00PM

NODES 2019

Best Practices to Make (Very) Large Updates in Neo4j

Fanghua(Joshua) Yu
Field Engineering Lead, APAC.

joshua.yu@neo4j.com

https://www.linkedin.com/in/joshuayu/

Introduction

Fanghua(Joshua) Yu

Pre-Sales & Field Engineering Lead,
Neo4j APAC

Joshua.yu@neo4j.com

Let's know each other …(later)

Ever complained, that why it is SO SO SO SLOW to update data in Neo4j?

And even worse, sometimes Neo4j database service just stopped responding?

Java OutOfMemory Error !!!!!!!

Agenda

- Understand How Neo4j Handles Updates
- Strategies to Optimize Updates
- A Case Study: Making Updates with Limited Memory
- More on Cypher Tuning
- Summary

How Neo4j Handles Updates

- (In most of the cases) Every Cypher statement runs within a thread.
- Database updates defined in one Cypher statement are executed as a Transaction.
- ACID: consistency is critical.
- Neo4j keeps all context of a Transaction in JVM Heap Memory.
- Large updates ➔ large memory

How Neo4j Handles Updates(cont.)

- Remember this?

```
USING PERIODIC COMMIT 1000
LOAD CSV FROM …
MATCH…
MERGE…
CREATE…
```

- When loading large amount of data, it is necessary to specify batch size to keep Transaction in a manageable size.

How Neo4j Handles Updates(cont.)

For any Cypher statement, we can use APOC procedures to achieve the same, i.e. limit the transaction size.

There are APOC procedures built for this purpose:

- apoc.periodic.commit()
- apoc.periodic.iterate(): see example below

APOC stands for 'Awesome Procedures of Cypher, or 'A Package of Components', or the name of a crew member on Nebuchadnezzar.

The first parameter is a Cypher query to return a collection of node ids.

```
CALL apoc.periodic.iterate(
"MATCH (p:Post) WHERE id(p) >=4000000 AND id(p) < 5000000 RETURN id(p) AS postId",
"MATCH (p:Post) <-[:POSTED]- (u:User) WHERE id(p) = postId SET p.postedBy = u.userId",
{batchSize:2000,parallel:false,iterateList:true });
```

The second parameter is the Cypher to update database based on results returned by the 1st query.

batchSize defines number of instances within a Transaction.

Whether to make updates in parallel?

Whether to have the whole list executed as one Transaction?

NODES 2019

Strategies to Optimize Database Updates

Let' have a look at all relevant aspects that can impact / improve the efficiency of database updates.

1) Hardware

2) Monitoring

3) Execution

4) Data volume

5) Parallel Processing

6) Query Tuning

7) Other

By Fanghua(Joshua) Yu, Oct. 2019

A Case Study

We will use the stackoverflow open dataset for the tests below.

- ✓ Contents：User, Post, Tag
- ✓ Data volume：~31 million nodes, 78 million relationships, 260 million properties

For detailed steps on how to download and import stackoverflow data into Neo4j, please check this page:

https://neo4j.com/blog/import-10m-stack-overflow-questions/

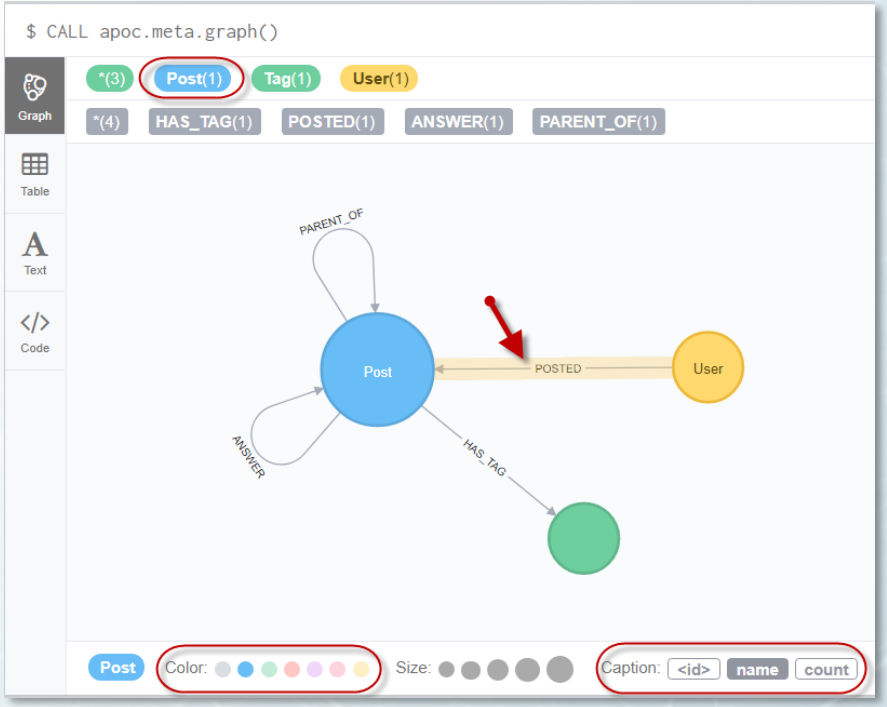
Test Case

The meta graph / meta model of Stackoverflow.

The Cypher statement to test:

```
MATCH (p:Post)
WITH id(p) AS postId
MATCH (p:Post) <-[:POSTED]- (u:User)
WHERE id(p) = postId SET p.postedBy = u.userId;
```

For each Post node, we find User nodes that are connected to it via POSTED relationship, and then save `name` of User node as property `postedBy` of Post node.



By Fanghua(Joshua) Yu, Oct. 2019

Test Environment

Hardware Specs :

- Lenovo Ideapad 510
- Intel i-7 CPU, 4 cores
- 12GB DDR4 RAM
- Seagate 2TB SATA 2 Mechanical
- Windows 10 Professional

To compare metrics, there is a Samsung 256GB SSD external HD connected via USB 3.0 port.

Neo4j :

- Neo4j Enterprise 3.3.1
- Database size : 16.5GB
- Java Page Cache : 2GB
- Java Heap : max 4GB

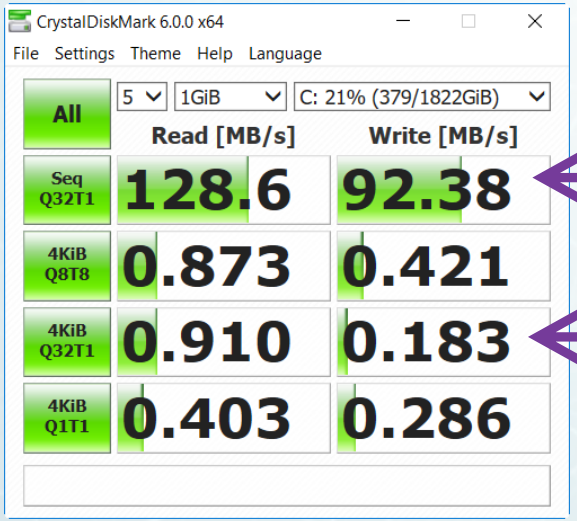
neo4j.conf

```
dbms.memory.heap.initial_size=2g

dbms.memory.heap.max_size=4g

dbms.memory.pagecache.size=2g
```

1 - Hardware

Firstly, let's run some tests on our hard drives. Data updates are mostly Random I/O operations so disk performance would make big differences.

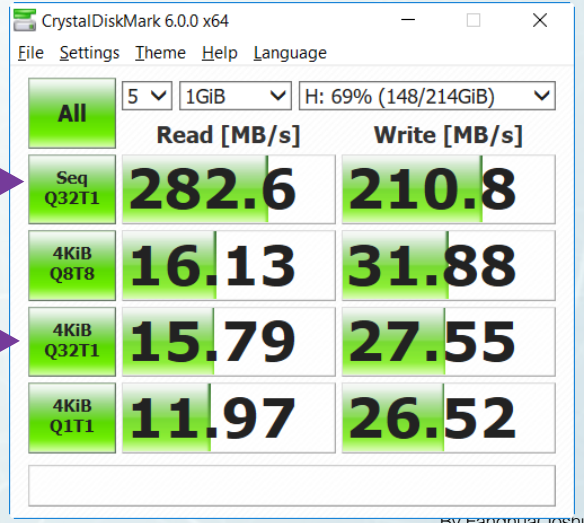
Local Mechanical Disk



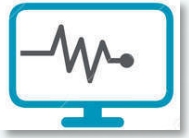
Sequential I/O: SSD is about 2 x local HD

Random I/O: SSD is about 15~150 x local HD!

External SSD via USB3.0



Tool used: CrystalDiskMark 64 v6



2 - Monitoring

During the tests, we monitor usage of CPU, RAM and disk, using Windows Task Manager, JConsole(the JMX client bundled with JDK).

To enable JMX metrics in Neo4j(Enterprise Edition ONLY) it involves these steps:

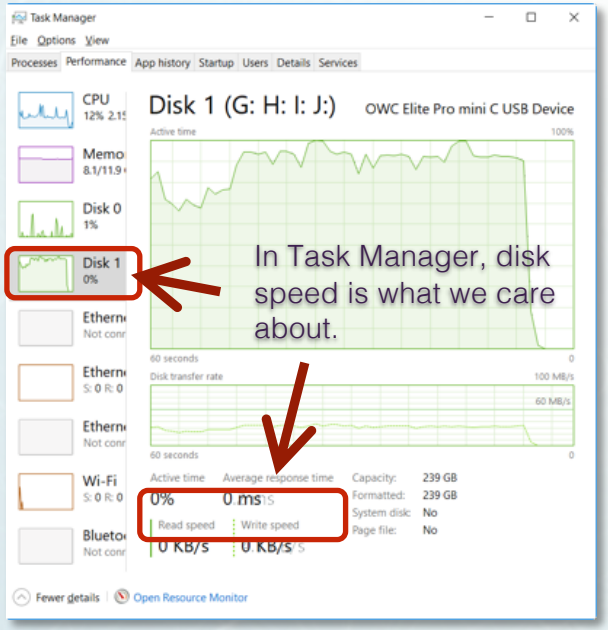
1) Neo4j Configuration

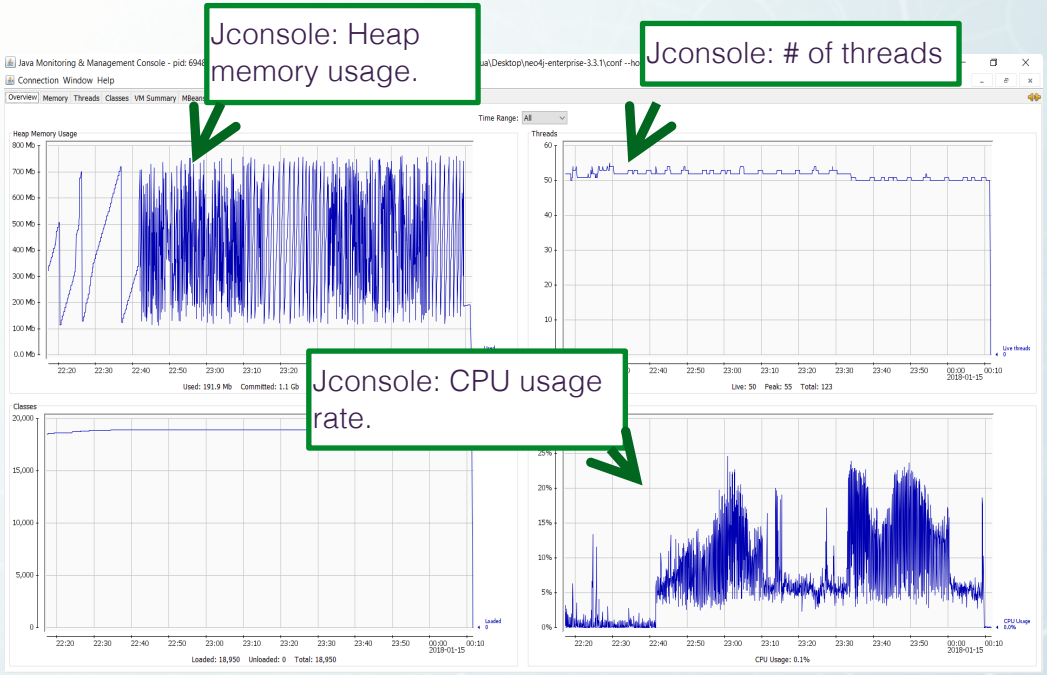
https://neo4j.com/docs/java-reference/current/jmx-metrics/

2) and set sole privilege to file jmx.passoword file:

https://docs.oracle.com/javase/8/docs/technotes/guides/management/security-windows.html

2 - Monitoring(cont.)





3 - Execution

Let's start with updating 1 million nodes:

```
MATCH (p:Post)
WHERE id(p) >=0 AND id(p) < 10000000
WITH id(p) AS postId
MATCH (p:Post) <-[:POSTED]- (u:User)
WHERE id(p) = postId SET p.postedBy = u.userId;
```

Filtering on id() to limit the number of nodes to update.

Accessing nodes and relationships via their ids is the <u>most efficient</u> method.

We record system metrics:

- CPU
- RAM
- Disk speed

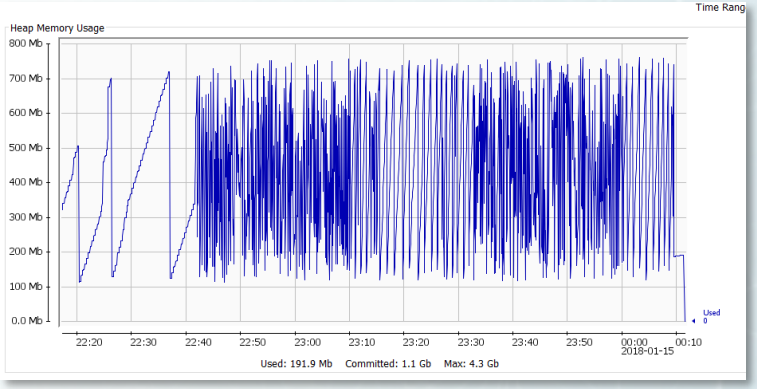
Execution in cypher-shell to avoid impact from browser.

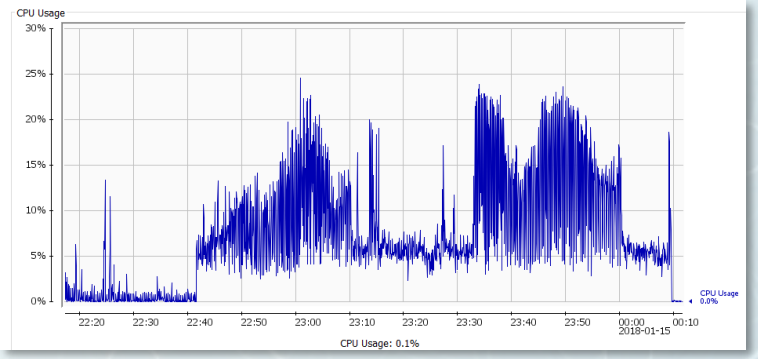
3 - Execution(cont.)

TC#2.1 Cypher-1M

Actual updates #	943K
Elapse(s)	46.5
Write speed(nodes/s)	20279
CPU usage	<25%
Java Heap (MB)	<750
System disk*	<30%
DB disk max/avg speed(MB/s)	25/10

* System disk is the local mechanic HD on which OS and Neo4j are installed.





3 - Execution(cont.)

TC#2.2 Cypher-1.5M

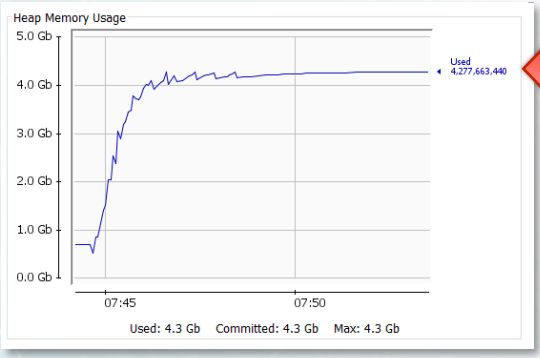
Actual updates #	1.49M
Elapse(s)	58
Write speed(nodes/s)	25657
CPU usage	<25%
Java Heap (MB)	3500
System disk*	<20%
DB disk max/avg speed(MB/s)	25/10

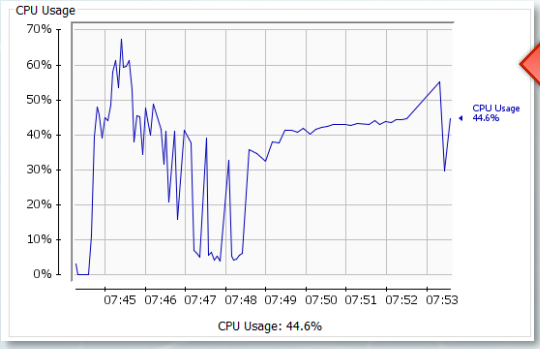
When we tried to update 1.5 million nodes in one Cypher statement, the Heap memory usage has reached 3.5GB which is close to the limit.

As all interim status of a Transaction are kept in Heap memory for the purpose of Roll-back, the more updates in a Transaction, the more Heap it would need.

3 - Execution(cont.)

TC#2.3 Cypher-2M, failed.

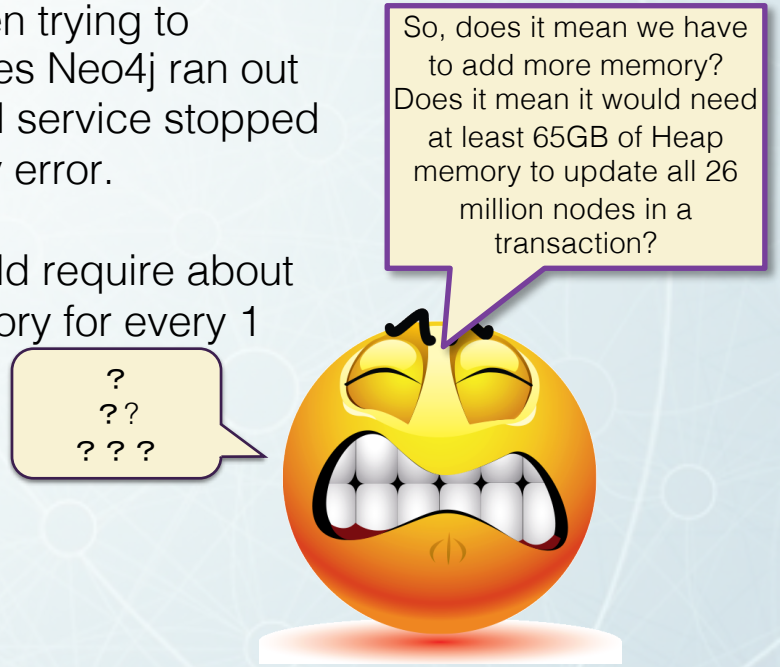




Not surprisingly, when trying to update 2 million nodes Neo4j ran out of Heap memory and service stopped due to OutOfMemory error.

In a summary, it would require about 2.5GB of Heap memory for every 1 million updates.

CPU usage rate



APOC to the rescue(again!)

For any Cypher statement, we can use APOC procedures to split large transaction into smaller batches, and each batch is executed as a transaction too.

There are APOC procedures built for this purpose:

- apoc.periodic.commit()
- apoc.periodic.iterate(): see example below

The first parameter is a Cypher query to return a collection of node ids.

```
CALL apoc.periodic.iterate(
"MATCH (p:Post) WHERE id(p) >=4000000 AND id(p) < 5000000 RETURN id(p) AS postId",
"MATCH (p:Post) <-[:POSTED]- (u:User) WHERE id(p) = postId SET p.postedBy = u.userId",
{batchSize:2000,parallel:false,iterateList:true });
```

The second parameter is the Cypher to update database based on results returned by the 1st query.

batchSize defines number of updates within a Transaction.

Whether to make updates in parallel?

Where to have the whole list executed as one Transaction?

APOC to the rescue(again!)

```
CALL apoc.periodic.iterate(
"MATCH (p:Post) WHERE id(p) >=4000000 AND id(p) < 5000000 RETURN id(p) AS postId",
"MATCH (p:Post) <-[:POSTED]- (u:User) WHERE id(p) = postId SET p.postedBy = u.userId",
{batchSize:2000,parallel:false,iterateList:true });
```

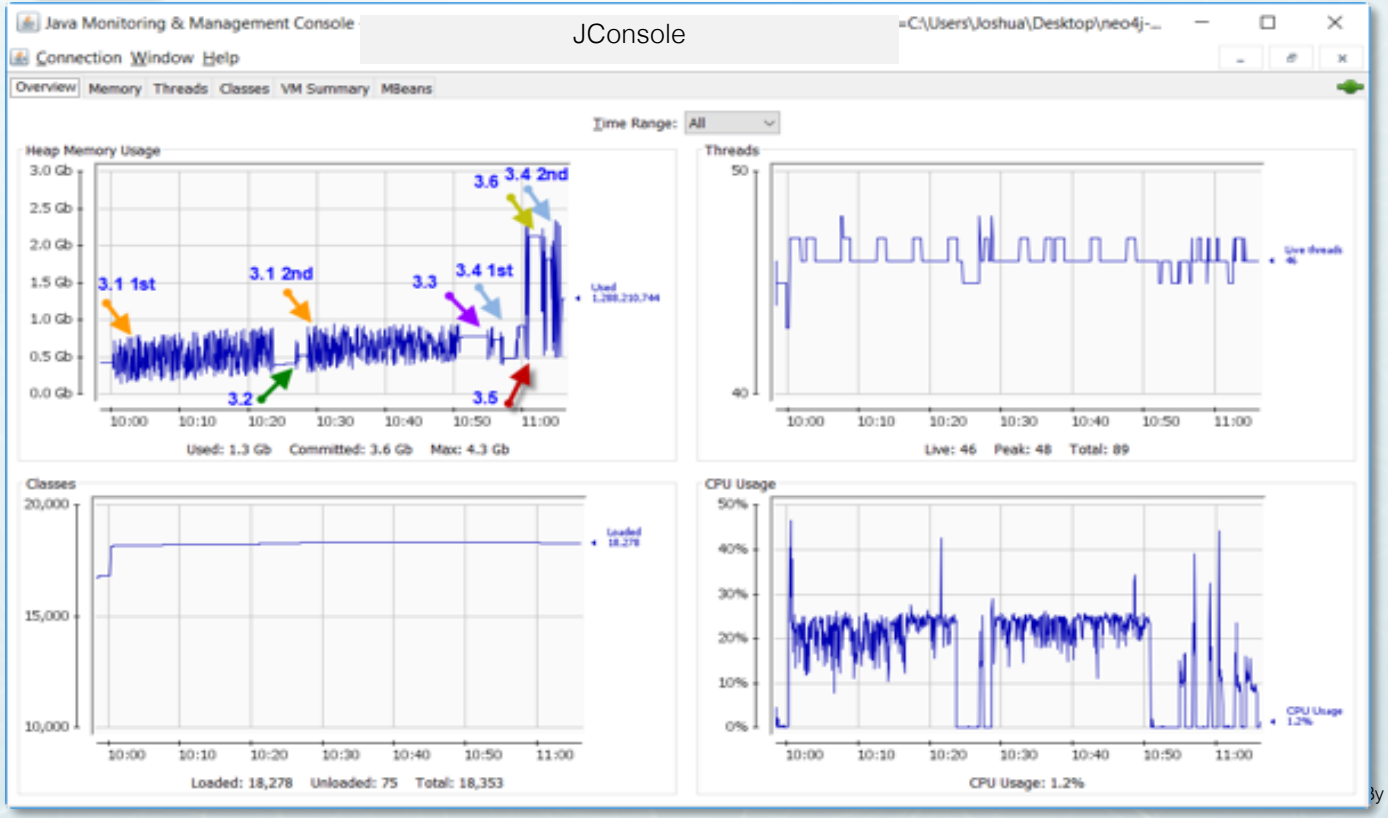
4 – Data Volume

TC#3.2 ~ 3.6 Find the optimized batchSize

With parallel = **false**, iterateList = **true**

Test Case #	3.2	3.3	3.4	3.5	3.6
batchSize	2000	200	10k	15k	20k
Actual updates #	1M	1M	1M	1M	1M
Elapse(s)	38	47	28	25	36
Write speed(nodes/s)	26315	21280	35714	40000	27778
CPU usage	<25%	<30%	<40%	<50%	<50%
Java Heap (MB)	<900	<900	<900	<2400	<2400
System disk*	<30%	<30%	<40%	<40%	<40%
DB disk max/avg speed(MB/s)	-/10~18	-/10	-/30	-/32	-/32

4 – Data Volume(cont.)



4 – Data Volume(cont.)

Based on previous tests, we figured out the I/O is about 26~30MB/s. batchSize defines how many statements to commit in each batch. For a total number of 1 million nodes to update, we can see:

- The larger batchSize, the less transactions to commit;
- By increasing batchSize from 2000 to 15k, the overall processing time has been reduced by 17%;
- When the batchSize is over 20k, the overall processing time actually increased by 19%, likely caused by the disk I/O capacity limit;
- Too small batchSize, say 200 in our test, has more batches and a longer overall processing time(+59%)

When batchSize is 2000, peak write has reached 18MB/s(60% of the max). In order to reserve some bandwidth to other thread, we will use it in the following test cases.

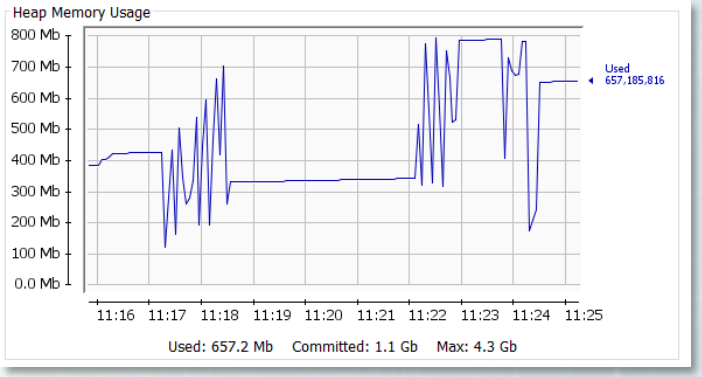
APOC to the rescue(again!)

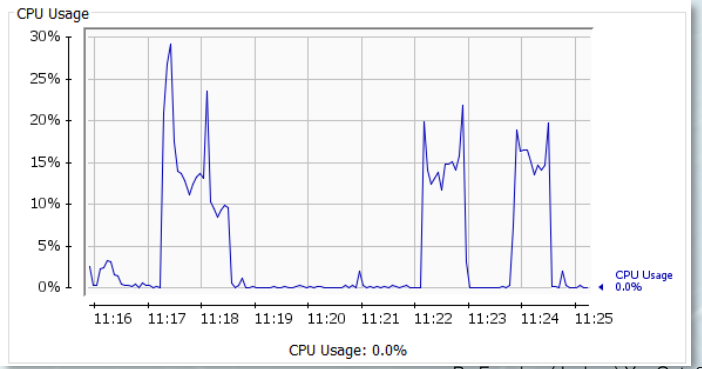
```
CALL apoc.periodic.iterate(
"MATCH (p:Post) WHERE id(p) >=4000000 AND id(p) < 5000000 RETURN id(p) AS postId",
"MATCH (p:Post) <-[:POSTED]- (u:User) WHERE id(p) = postId SET p.postedBy = u.userId",
{batchSize:2000,parallel:false,iterateList:true });
```

5 - Parallel

TC#4.1 With parallel = true

batchSize	500
parallel	true, 4 cores
iterateList	true
Actual updates #	1M
Elapse(s)	18.2
Write speed(nodes/s)	54945
CPU usage	<30%
Java Heap (MB)	<800
System disk*	<20%
DB disk max/avg speed(MB/s)	-/41





5 - Parallel(cont.)

TC#5 parallel=true, more updates

batchSize	500			
parallel	true, 4 cores			
iterateList	true			
Actual updates #	2M	4M	10M	15M
Elapse(s)	43	117	290	403
Write speed(nodes/s)	46511	188	34482	37220
CPU usage	<55%	<55%	<55%	<55%
Java Heap (MB)	<900	<900	<900	<900
System disk*	<20%	<20%	<20%	<20%
DB disk max/avg speed(MB/s)	-/45 max	-/46 max	-/60 max	-/90 max

Compared to TC#3.2: 26315

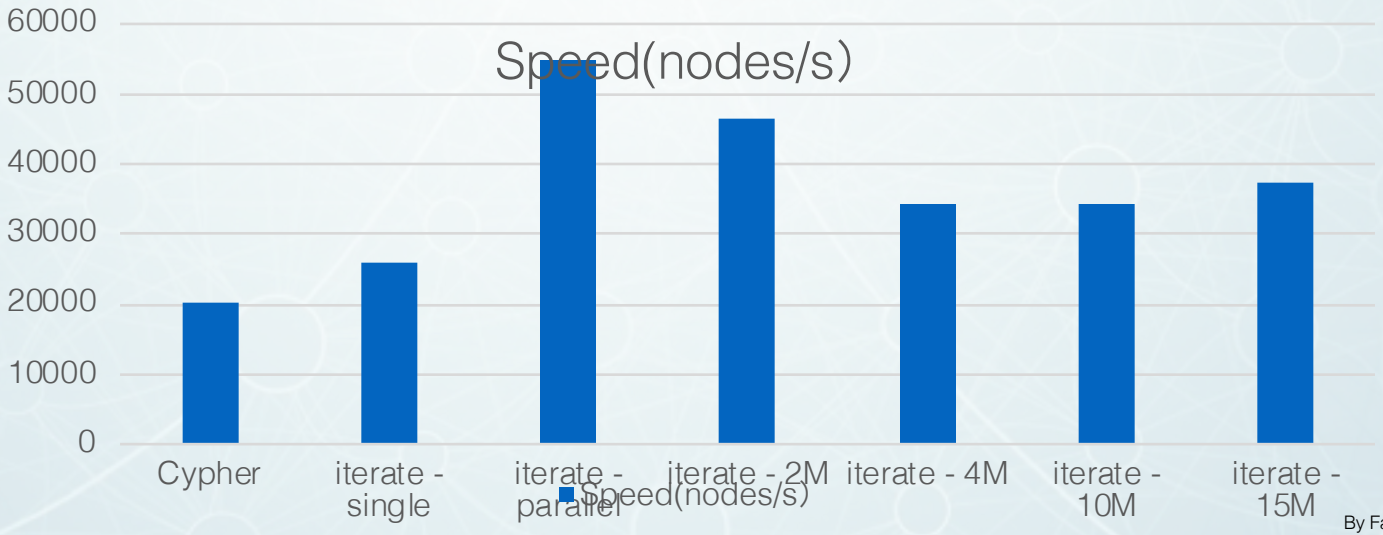
Compared to TC#3.2: <25%

Compared to TC#3.2: 18MB

5 - Parallel(cont.)

Some findings :

- Parallel processing is more efficient
- Be careful about **locking conflicts**



6 – Query Tuning

So far, in our Cypher statement, it returns node id:

```
MATCH (p:Post)
WHERE id(p) >=0 AND id(p) < 10000000
WITH id(p) AS postId
MATCH (p:Post) <-[:POSTED]- (u:User)
WHERE id(p) = postId SET p.postedBy = u.userId;
```

What if it returns node as object:

```
CALL apoc.periodic.iterate(
"MATCH (p:Post) <-[:POSTED]- (u:User) RETURN p, u",
"SET p.postedBy = u.userId",
{batchSize:500,parallel:true,iterateList:true })
```

What the results would look like？

6 – Query Tuning(cont.)

TC#6 Return node objects.

batchSize	500			
parallel	true, 4 cores			
iterateList	true			
Actual updates #	1M	2M		8M
Elapse(s)	34	67	121	219
Write speed(nodes/s)	29411	29850	33057	36529
CPU usage	<55%	<60%	<80%	<89%
Java Heap (MB)	<1400	<2200	<1800*	<2400
System disk*	<20%	20%	<20%	<20%
DB disk max/avg speed(MB/s)	-/-	-/-	-/-	-/-

Compared to TC#5: 43s

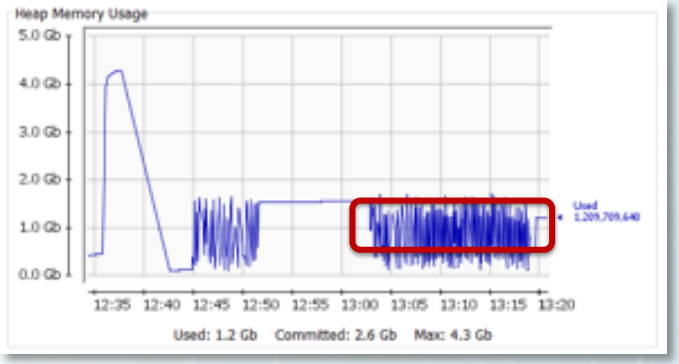
Compared to TC#5: 46511

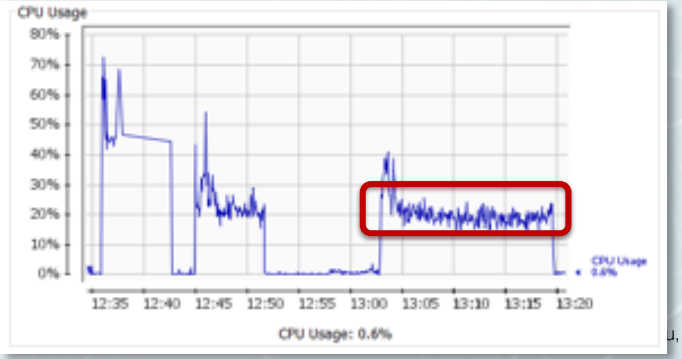
Compared to TC#5: 900MB

6 – Query Tuning(cont.)

TC#7.2 The full update.

batchSize	500
parallel	true, 4 cores
iterateList	true
Actual updates #	26,545,725
Batches#	1006
Elapse(s)	26387
CPU usage	<42%
Java Heap (MB)	<1800
System disk*	<20%





6 – Query Tuning(cont.)

This is no longer an issue in newer Neo4j versions.

*** Tested on Neo4j 3.5.5 with local SSD drive:
*** max heap: 3.5GB, CPU: 73%

```
CALL apoc.periodic.iterate(
  "MATCH (p:Post) RETURN id(p) AS postId",
  "MATCH (p:Post) <-[:POSTED]- (u:User) WHERE id(p) = postId SET
p.postedBy = u.userId",
  {batchSize:2000, parallel:true, iterateList:true}
);
```

PROBLEM SOLVED !

6 – Query Tuning(cont.)

More about query tuning.

With large query or update, it is ALWAYS recommended to EXPLAIN and / or PROFILE the query before it is sent to database.

Advanced Cypher Training modules can give you more details on those commands and how to analyse efficiency of *execution plan*.

6 – Query Tuning(cont.)

More about query tuning.

Sometimes, when importing data, even if USING PERIODIC COMMIT is used, it's still possible to get OutOfMemory error!

This can be caused by:
1) trying to do too many steps for each line read;
2) having ***eager*** operator that disables periodic commit.

6 – Query Tuning(cont.)

Plan evaluation are **Eager** or **Lazy**

- Most query evaluation is lazy:
  - Operators pipe their output rows to their parent operators as soon as they are produced.
  - Child operator may not be finished before the parent receives and processes rows.
- An Eager operation can take 2 forms:
  - An **EagerAggregation** step caused by any of the aggregation functions (e.g. count, sum). This is normal and of lesser concern.
  - An **Eager** step caused by a reference later in the query to an object modified earlier in the query.

https://neo4j.com/docs/cypher-manual/current/execution-plans/

6 – Query Tuning(cont.)

Eager operators disable PERIODIC COMMIT.

```
PROFILE USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/advanced-cypher/movies2.csv' AS row
MERGE (m:Movie {id:toInteger(row.movieId)})
   ON CREATE SET m.title=row.title, m.avgVote=toFloat(row.avgVote),
      m.releaseYear=toInteger(row.releaseYear),
      m.genres=split(row.genres,":")
WITH m, row
MERGE (p:Person {id: toInteger(row.personId)})
   ON CREATE SET p.name = row.name, p.born = toInteger(row
      p.died = toInteger(row.deathYear)
RETURN m.title ORDER BY m.title
```

Solutions:

i. Don't return anything

ii. Return no property.

6 – Query Tuning(cont.)

It is also possible to use APOC procedures:

```
CALL apoc.periodic.iterate(
"CALL apoc.load.csv('https://data.neo4j.com/advanced-cypher/movies2.csv' )
 YIELD map AS row RETURN row",
"MERGE (m:Movie {id:toInteger(row.movieId)})
   ON CREATE SET m.title=row.title, m.avgVote=toFloat(row.avgVote),
      m.releaseYear=toInteger(row.releaseYear),
      m.genres=split(row.genres,':')
 WITH m, row
 MERGE (p:Person {id: toInteger(row.personId)})
   ON CREATE SET p.name = row.name, p.born = toInteger(row.birthYear),
      p.died = toInteger(row.deathYear)",
{batchSize: 500}
)
```

6 – Query Tuning(cont.)

We've covered enough on adding or updating database, what about deleting data?

Let's have a look at the sample below:

We want to delete relationship (:Post) -[:PARENT_OF]-> (:Post), in total 16,502,856 / 16 millions.

And here is a simple and safe way to do so:

```
// method #1 use pattern matching

CALL apoc.periodic.commit(
  'MATCH (p) -[r:PARENT_OF]-> () WITH r LIMIT {limit} DELETE r RETURN
count(r)',
  {limit:5000}
)
```

6 – Query Tuning(cont.)

… but it is not fast enough.

To delete 16 million relationships, it took 8.5 hours, about 539 deletes / second.

Can we do better?

6 – Query Tuning(cont.)

Yes, of course!

Neo4j has invented a very unique storage structure for nodes and relationships, i.e. fixed width block. All nodes are stored in the Node Store with a fixed width of 15 bytes, so are relationships in the Relationship Store with a fixed width of 33 bytes.

Remember the internal id? It is actually the address / location of the node or relation in its store!

As a result, finding a node or relationship by its internal id is the most efficient way!

6 – Query Tuning(cont.)

Here is how we use the idea to do large deletion in a much faster way.

First, let's find out the low and high limits of internal id for PARENT_OF relationship:

```
// Method #2 use internal id
//
// Find out range of id

MATCH () -[r:PARENT_OF]-> () RETURN min(id(r)), max(id(r))
```

It returns 0 and 16502855. (Feel lucky right?)

6 – Query Tuning(cont.)

Second, we will construct id using nested loop and find relationship by its id before delete it.

```
// Method #2 use internal id
//
// Delete relationships in 1651 batches, and each has 10000 deletes.

WITH range(0,1650) AS highr
UNWIND highr AS i1
CALL apoc.periodic.commit(
  'WITH range(0,9999) AS lowr
   UNWIND lowr AS i2
   WITH '+i1+'*10000 + i2 AS id WHERE id < 16502856
   MATCH () -[r:PARENT_OF]-> () WHERE id(r) = id DELETE r RETURN 0',
  {batchSize:10000}
) YIELD updates
RETURN
```

6 – Query Tuning(cont.)

and this method finished in just **591s**, a 50 times of improvement!

Even if the relationship ids are not sequential like we have here, to access relationship(as well as node) via its internal id is still much more efficient than a pattern matching query over indexed property.

7 – Other

Things can be more complicated in a cluster environment.

Heavy update in a cluster may cause the Leader node too busy to respond. As a result, other Followers may think the Leader is offline and start a re-election.

Core: read + write, real time consistency

Replica: read only, eventual consistency

Solution: keep transaction size small enough.

Summary

- Disk Random I/O performance is critical:

  - SSD beats mechanical HD easily.

    For the tests we ran, if they were done over mechanical HD, the best ever achieved was about 7700 nodes/s, only 14% of SSD benchmark

  - SSD supports parallel processing much better.

- Neo4j uses JVM Heap memory to keep interim status of transactions. As a rough estimate, it requires 2.5 ~ 3GB RAM to update every 1 million nodes. As a result, transaction size matters a lot.
- When loading data from CSV files, remember to include USING PERIODIC COMMIT followed by a number to define batch size / bulk update size.

Summary(cont.)

- Use APOC procedures to control transaction size:
  - apoc.periodic.iterate()
  - apoc.periodic.commit()
- It is necessary to run some tests to reach a balance between total *number of transactions* and *batch size*, taking available memory into consideration.
- Use parallel processing whenever possible(but remember to avoid locking).
- There is always space to tune your Cypher further.
- Need more help?
  - community
  - training modules: Advanced Cypher, Modeling, APOC
  - talking to us

Hunger Games Questions

1. Which part of memory does Neo4j use to keep transaction status?
   A. Page cache
   B. Heap
   C. Stack

2. Which of the following statement doesn't allow control of transaction size?
   A. USING PERIODIC COMMIT 1000 LOAD CSV FROM url ...
   B. CALL apoc.periodic.commit(cypherToUpdate, {params})
   C. MATCH (n:Node) DETACH DELETE n

3. If you observe frequent Leader re-election in a Neo4j causal cluster, which item below is NOT the possible cause?
   A. Network communication is interrupted.
   B. A complex graph algorithm is running on a Read Replica.
   C. A heave update statement has been submitted.

Answer here: r.neo4j.com/hunger-games

THANK YOU!

Comment and feedback:
joshua.yu@neo4j.com