Final Bits:

- Why Graph Databases are a Growing Force.
- LRU? Not really,
  - In RDBMS its Clock replacement.

Graph Databases and Query Languages…

- Are better Why? How?

Ryan Wolter's Term Project Spring '19

- Implement an Application in Both

SQL and Cypher
And
Compare

The Application – Prologue

- *Very Sadly,* The Application is Proprietary
  - We can't publish.
  - Even in class, I can't reveal much about the application.

The Application

- Application area: Computer Security Forensics
- Practical nature:
  - An actual, labor intensive task
    - Done regularly
    - At the OS command line
    - By a team

The Application (cont'd)

- Application area: Computer Security Forensics
- Ryan's insight
  - If the system state(s) examined by the team were represented as data in a database.
  - Entire tasks could be accomplished using a query.
  - The data forms a graph!

The Experiment

- Developed a use-case document
  - That is: Identified actual tasks to be done.
  - 8 use cases (identifiable tasks).
- Defined schema, SQL and Cypher
- 8 use cases turned into 10 queries
  - 2 use cases took two different queries
- Wrote and r an the queries on Postgres and Neo4j
  - Compared results

Qualitative Assessment of the Optimizers

- Neo4j:
  - Has an optimizer that chooses good plans
  - Very primitive,
    - There are explicit instructions on how to write a Cypher query, so optimizer produces better plan.
    - Take the form of,
      - Certain operators, if lexically earlier in the query… will be evaluated earlier in the plan.
      - (developers get to *push down* operations)

Qualitative Assessment of the Optimizers

- Postgres:
  - wrt recursive queries
  - Also has, some, sensitivity, to developer syntax

Qualitative Assessment of the Optimizers

- Text (string) operations
  - Neo4j does well.
  - SQL uses the LIKE operator
    - Postgres, as is common among RDBMS, does a poor job with LIKE queries.
- With data scraped from non-traditional sources, this has become and important operation.

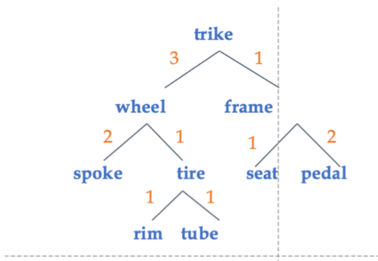
Performance (execution time)

Table 2: Performance Results

Query	Database	Speed
Find exact "all.txt"	Neo4j	1 ms
Find exact "all.txt"	Postgresql	1 ms
Find all *.txt (using `:is_root`)	Neo4j	4013 ms
Find all *.txt (using `WITH`)	Neo4j	777 ms
Find all *.txt (using `WHERE`)	Neo4j	947 ms
Find all *.txt	Postgresql	5949 ms
Find all *.txt ordered limit 10 (executing limit late)	Neo4j	833 ms
Find all *.txt ordered limit 10 (push limit early)	Neo4j	236 ms
Find all *.txt ordered limit 10 (executing limit late)	Postgresql	6008 ms
Find all *.txt ordered limit 10 (push limit early)	Postgresql	32 ms
Find all *.txt ordered limit 1000	Neo4j	636 ms
Find all *.txt ordered limit 1000	Postgresql	56 ms
Find all *.txt ordered limit 10000	Neo4j	636 ms
Find all *.txt ordered limit 10000	Postgresql	5720 ms
Find all *.txt ordered limit 10000 (no recursion)	Postgresql	183 ms

Graph Databases?
It's not about speed… of the computer

Recall Magic Sets Example



Datalog
Rule 1: Comp(Part, Subpt) :- Assembly(Part, Subpt, Qty).
Rule 2: Comp(Part, Subpt) :- Assembly(Part, Part2, Qty), Comp(Part2, Subpt).

Recall Magic Sets Example (2)

Datalog
Rule 1: Comp(Part, Subpt) :- Assembly(Part, Subpt, Qty).
Rule 2: Comp(Part, Subpt) :- Assembly(Part, Part2, Qty), Comp(Part2, Subpt).

SQL
```
WITH RECURSIVE Comp(Part, Subpt) AS      /* Define Comp */
(SELECT A1.Part, A1.Subpt FROM Assembly A1)
UNION
(SELECT A2.Part, C1.Subpt
 FROM Assembly A2, Comp C1
 WHERE A2.Subpt=C1.Part)

SELECT * FROM Comp C2   /* Returns all parts of a trike */
```

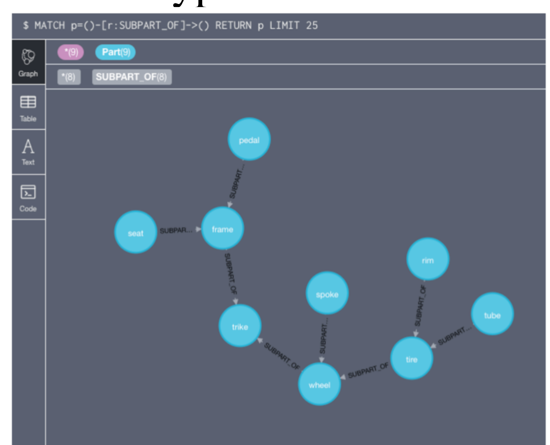

Magic Sets Example (3) - Cypher

Cypher
```
MATCH (n:Part)-[:SUBPART_OF*]->(p:Part {name:"trike"}) RETURN n.name
```

SQL
```
WITH RECURSIVE Comp(Part, Subpt) AS      /* Define Comp */
(SELECT A1.Part, A1.Subpt FROM Assembly A1)
UNION
(SELECT A2.Part, C1.Subpt
 FROM Assembly A2, Comp C1
 WHERE A2.Subpt=C1.Part)

SELECT * FROM Comp C2   /* Returns all parts of a trike */
```


Cypher - IDE



Real World Example (i.e. Ryan's)

Cypher

```
match(f:file) where f.name = "all.txt"
match(r:is_root)
match p = (r)-[:parent_of*]->(f)
// The reduce notation concatenates the parent nodes to reconstruct the path
return reduce(acc = "/", x IN nodes(p)[1..] | acc + "/" + x.name),
    reduce(acc = 0, x IN nodes(p)[1..] | acc + 1)
```

SQL

```
WITH RECURSIVE filetree AS (
        select file_id, filename, parent_file_id, host, path as path_org,
            filename as path, 0 as depth, parent_file_id as tpid from files
            where filename ='all.txt'
        UNION
        select ft.file_id, ft.filename, ft.parent_file_id, ft.host,
            ft.path as path_org, f.filename || '/' || ft.path as path,
            ft.depth + 1 as depth, f.parent_file_id as tpid
        from files f
        join filetree ft
        on ft.tpid = f.file_id
```


Graph DB: Developers are More Productive

- How important is that?

Graph DB: Developers are More Productive

- Assembler
- C
- C++
- Java
- Map Reduce (for parallel programming)

Clock Replacement

- A "lower-overhead" approximate implementation of LRU.
- Commonly argued to be good in the face of <u>sequential flooding</u> – (I've never bought it)
  - <u>Sequential flooding</u>: What happens, with LRU when repeated scanning R, and, e.g. B(R) = M(R) + 1

Buffer Replacement Policy slide thanks Joe Hellerstein

- Buffer is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU)
  - Most-recently-used (MRU)
    - Also called toss-immediate
  - Clock
- Policy can have big impact on # of I/O' s; depends on the *access pattern*.

LRU Replacement Policy

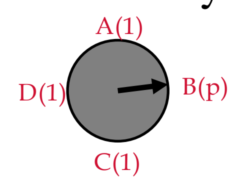
- *Least Recently Used (LRU)*
  - for each page in buffer pool, keep track of time when last *unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages
- Problems?
- *Problem: Sequential flooding*
  - LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O.
  - Idea: MRU better in this scenario?

LRU Replacement Policy

- *Least Recently Used (LRU)*
  - for each page in buffer pool, keep track of time when last *unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages
- Problems?
- *Problem: Sequential flooding*
  - LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O.
  - Idea: MRU better in this scenario?

"Clock" Replacement Policy

- Associate with the buffers, a cycle of frames
- Per frame, store
  - one *reference bit*
  - *pin count*
- On a buffer access,
  - **decrement pin count, until 0**
  - **when pin count = 0 set reference bit**



- replacement

do for each page in cycle {
if (pincount == 0 && ref bit is on)
turn off ref bit;
else if (pincount == 0 && ref bit is off)
choose this page for replacement;
} until a page is chosen;

Questions:
How like LRU?
Problems?

Facets of Clock Policy

- Pin buffers by initializing pin count to ∞
- Hack pin count for other purposes,
  - (nested loop join, anticipate how many times the page will be read).
- LRU is *the worst* policy for sequential flooding. Clock by itself isn't that much better, but you can hack pin count.