A Python Implementation of Schemaless Model on MySQL

Ryan (Jianye) Ye, Google

PyCon 2011, China

About Me

- 2 year Graph Architect in NVIDIA
- 1 year Lead Developer in Slide China (Acquired by Google)
- 1.5 year Technical Lead in Prizes.org Team, Google Shanghai

Slide uses Python to Build ...

- Web Servers
- Data Access Servers
- Background Processing Servers
- Application/Business Logics
- Infrastructure Tools

What is Schemaless ?

- No pre-defined columns and data types
- Each row in a table is a object with arbitrary data structure
- Example:

```
1: {'name' : 'John', 'phone' : '12345678'}
2: {'name' : 'Tom', 'address' : {
            'street' : 'Fifth Avenue',
            'no' : 321,
        }
    }
```

Why we went Schemaless data model?

- Same data representations in Python and database
- Fast development iteration

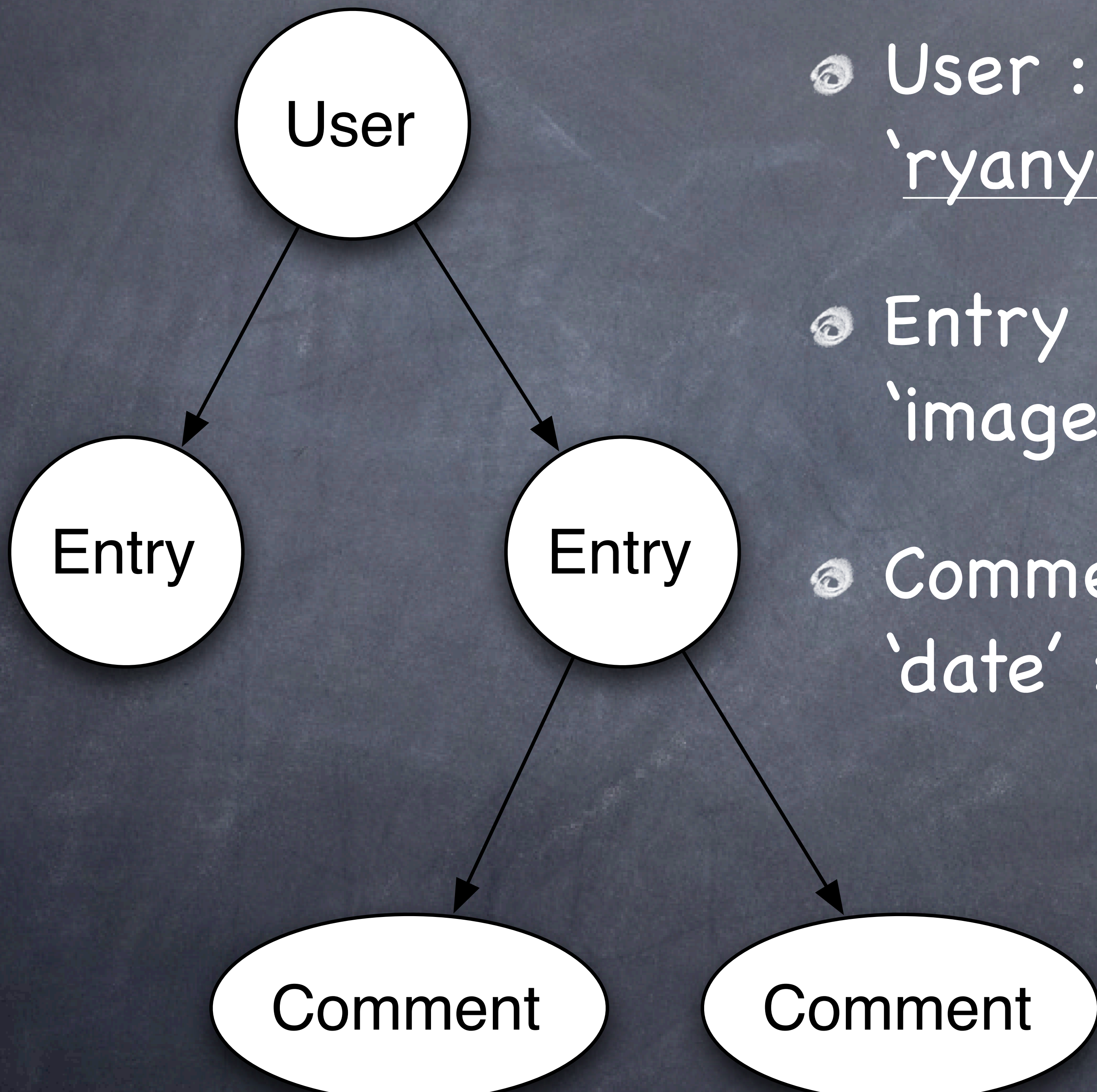
Why not using existing NoSQLs ?

- The short Answer is that we developed our solution before most of NoSQL solutions enter mainstream.
- Other reasons include
  - We want to store all our data in a centre place (MySQL)
  - It's fun to implement such software in Python

Basic structure of GRAPH

- Every object is in a node in a tree.
- Nodes are connected by edges
- Each node has its own properties

GRAPH Example



- User : {'name' : 'Ryan', 'email' : 'ryanye@google.com' }
- Entry : {'text' : 'PyCon Logo', 'image' : '/id345/logo.png'}
- Comment : {'text' : 'Awesome!', 'date' : 1322048950}

DB Schema - Node

- TABLE GraphNode
  - id: unique identity
  - type: long
  - properties: binary (max 64KB)
  - children_count: long
  - time_created: long
  - time_removed: long
- Serializer: wirebin - https://github.com/slideinc/wirebin

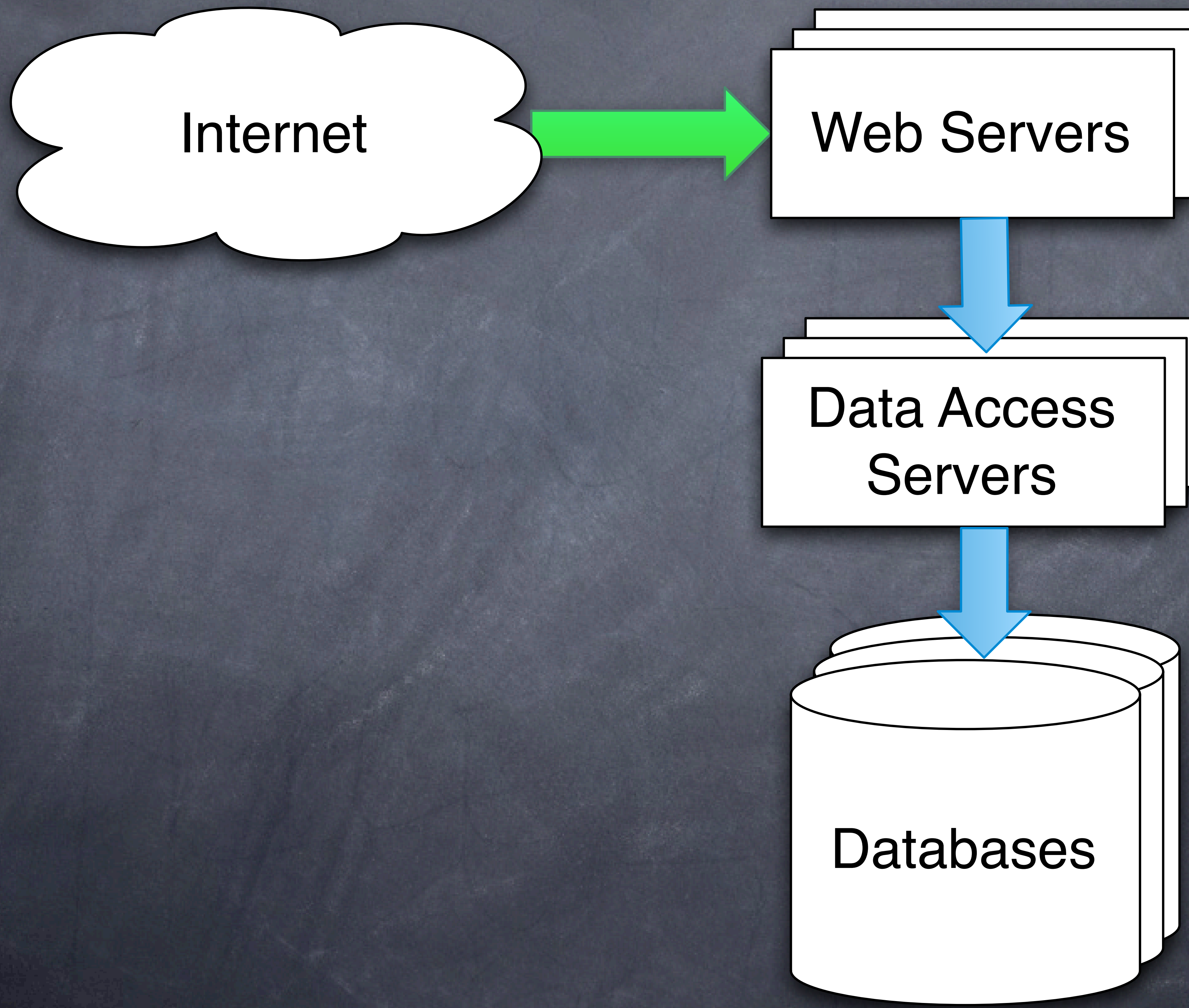
DB Schema - Edge

- TABLE GraphEdge
  - id: unique identity
  - parent_id: long
  - child_id: long
  - time_created: long
  - time_removed: long

Access GRAPH API

- graph.node(node_id)
- graph.children(parent_id, type = None)
- graph.create(parent_id, type, properties)
- graph.update(node_id, properties)
- graph.remove(node_id)
- graph.move(node_id, new_parent_id)

The Architecture



Scale with Multi-DB

- Sharding by high bits of node-id
  db-shard-id = (node-id >> 52) & 0xfff
  - Easy to implement - MySQL auto-incr-id + predefined-base-id
  - Easy to add new shards, maximum to 4096 db instances
  - No data relocation when adding shards

Scale with Multi-DB

- Edges and children nodes lives in the same db shard as their parents
  - Single SQL-statement on graph.children
  - Better use of locality
  - Not always true due to graph.move

Data Access Servers

- A Graph Access Server is a Python process with a dozens of coro-threads.
  - Dispatcher: A coro-thread listening to server port, dispatch access calls to workers
  - Workers: pre-allocated coro-threads performing cache lookup or SQL queries

    * coro-thread: coroutine thread, a lightweight user-space ‘thread’. https://github.com/slideinc/gogreen

Data Access Servers

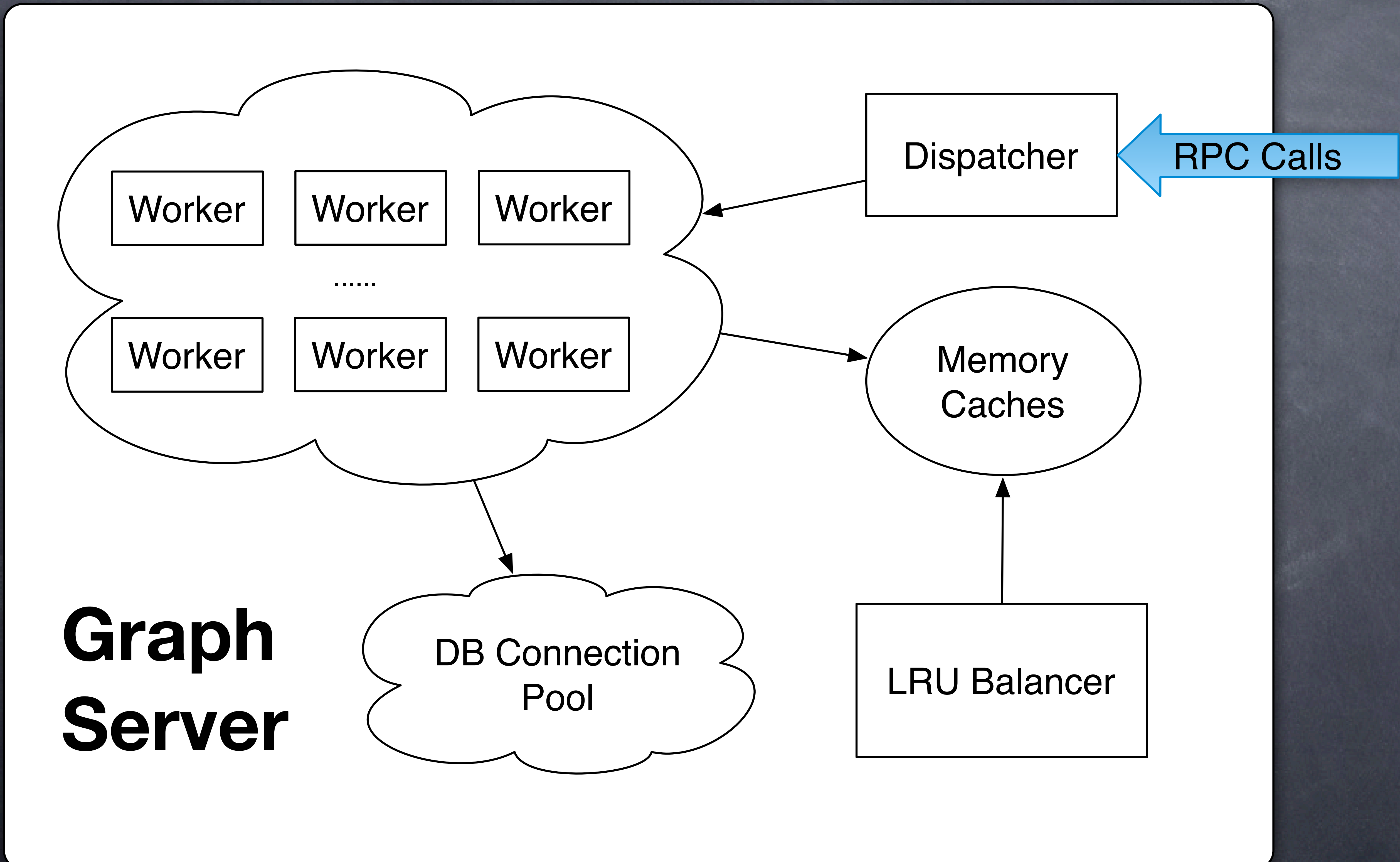
- What else in a Graph Access Server ?
  - A Pool of Connections to all DB shards
  - Cache LRU Balancer: a coro-thread periodically monitoring in-memory data cache, evicting least recently used items.

LRU Caches

- L1 Cache: nodes/edges, a big Python Dict using node-id as keys
- L2 Cache: Similar to L1, but all data are compressed via zlib + wirebin
- Cache data are persistent on disk when server exits. Serialize with wirebin
- Only read-cache, always write through

Cache Invalidation

- graph.update invalidates the cache of that node
- graph.create invalidates the cache of the parent
- graph.remove invalidates the cache of that node and its parent
- graph.move invalidates the cache of old parent and new parent


Worker
Worker
Worker
......
Worker
Worker
Worker
Dispatcher
RPC Calls
Memory Caches
Graph Server
DB Connection Pool
LRU Balancer

Server Configuration

- 32 graph-server instances on a physical server box (approximately to num-of-cores)
- In each graph-server
  - 128 workers
  - 16 Connections to each DB shards

Performance for Single Server

- Server 128 x 32 requests in parallel
- Average response times 1.38ms
- Average Cache-hit rate 99.72%
- Theoretically, MAX Request Per Sec on a box
  = 128 x 32 x (1000 / 1.38)
  = 2.73 Million

Scale with Multi Graph Access Servers

- Sharding by lower bits of node-id
  server-id = node-id & 0xff
  - Uniformly distribute traffic
  - A node only is cached on a single server.
    No cache-sync between servers.
    * Except for peers, see the next slide

Failover with a peer Graph Server

- If we have 32 servers with id 0...31, each server will subscribe requests for node-id meeting (node-id & 0xf) == (server-id & 0xf)
  i.e, server with id-N and id-(N+16) are peers.
- For cache invalidation, the server will broadcast to its peer.
- On pushing new server code, the peers always restarted sequentially

Summary

- A Graph Model for general data storage
- Leverage coroutine-threads to archive high performance
- A 2-level In-memory cache to minimize DB access
- Scale across multi servers with simple sharding function

Thanks, questions?