S2Graph: A large-scale graph database with Hbase

Doyoung Yoon x Taejin Chin
1. KakaoTalk
   a. Mobile Messenger replacing SMS
   b. ‘KaTalkHe’ is being used as a verb in Korea like ‘Googling’
   c. 96% of Korean smartphone users are using KakaoTalk
   d. 170M users worldwide
   e. 3B messages / day
DaumKakao
A Mobile Lifestyle Platform

Social Platform
- KakaoTalk
- KakaoStory
- KakaoGroup
- Daum Cafe
- Zap
- Sol Group

Contents Platform
- Media Daum
- KakaoGame
- Digital Item Store
- KakaoTopic
- KakaoPage
- KakaoMusic

Commerce Platform
- KakaoPick
- Gift Shop
- KakaoStyle

Marketing Platform
- Yellow ID
- Plus Friend
- Story Plus

Local Platform
- Daum Map
- KakaoPlace
- Daum Cloud

Personal Platform
- KakaoHome
- Sol calendar
- Sol Mail
- Daum Webtoon

96% of Korean smartphone users are using KakaoTalk messenger, 170 million users worldwide)

Biggest mobile SNS in Korea
Our Social Graph

- **Message length**: 9
- **Write length**: 3
- **Pick withFriend**: 3
- **Play level**: 6
- **Search keyword**: "HBase"
- **Ad ID**: 603
- **Advertise ctr**: 0.32
- **Listen count**: 6
- **Friend**: 
  - **Affinity 1**:
  - **Affinity 2**:
  - **Affinity 3**:
  - **Affinity 4**:
  - **Affinity 6**:
  - **Affinity 9**:
- **Post ID**: 97
- **Game ID**: 1984
- **Item ID**: 13
- **Music ID**: 603
- **Message ID**: 201
- **Comment length**: 15
- **Commenting count**: 6
- **Advertise ctr**: 0.32
- **Search keyword**: "HBase"
- **Listen count**: 6
- **Commenting count**: 6
- **Advertise ctr**: 0.32
Technical Challenges

1. Large social graph constantly changing

   a. Scale

   more than,
   social network: 10 billion edges, 200 million vertices, 50 million update on existing edges.
   user activities: 400 million new edges per day
Technical Challenges (cont)

2. Low latency for breadth first search traversal on connected data.

   a. performance requirement

   peak graph-traversing query per second: 20000
   response time: 100ms
3. Update should be applied to graph in real time for viral effect
Technical Challenges (cont)

4. Support for Dynamic Ranking logic

   a. push strategy: hard to change data ranking logic dynamically.

   b. pull strategy: can try various data ranking logic
Each app server should know each DB’s sharding logic.

Highly inter-connected architecture
After

S2Graph DB
stateless app servers

APACHE HBASE
S2Graph: Distributed Online GraphDB

1. Low-latency
2. Graph-traversable
3. Scalable
4. Eventually consistent
5. Asynchronous, non-blocking
Why We Choose HBase?

1. High Availability
2. Scalability
3. Low latency
4. High concurrency
5. Fault tolerant
6. Integration with HDFS
7. Distributed operation
The Data Model

1. Columns
2. Labels
3. Directions
4. Index Properties
5. Non-index Properties

```
name = "josh"
age = 32
date = 20150507
```
How to store the data - Edge
Logical View

1. Snapshot edges: Up-to-date status of edge

<table>
<thead>
<tr>
<th>row</th>
<th>column</th>
<th>Tgt Vertex ID1</th>
<th>Tgt Vertex ID2</th>
<th>Tgt Vertex ID3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tgt Vertex ID1</td>
<td>Properties</td>
<td>Properties</td>
<td>Properties</td>
</tr>
<tr>
<td></td>
<td>Tgt Vertex ID2</td>
<td>Properties</td>
<td>Properties</td>
<td>Properties</td>
</tr>
<tr>
<td></td>
<td>Tgt Vertex ID3</td>
<td>Properties</td>
<td>Properties</td>
<td>Properties</td>
</tr>
</tbody>
</table>

a. Fetching an edge between two specific vertex
b. Lookup Table to reach indexed edges for update, increment, delete operations
### How to store the data - Edge

**Logical View**

2. **Indexed edges**: Edges with index

<table>
<thead>
<tr>
<th>Index Values</th>
<th>Tgt Vertex ID1</th>
<th>Index Values</th>
<th>Tgt Vertex ID2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Vertex ID1</td>
<td>Non-index Properties</td>
<td>Source Vertex ID1</td>
<td>Non-index Properties</td>
</tr>
</tbody>
</table>

**Table:**

- **Indexed edges**
  - **Rows:** Index Values, Tgt Vertex ID1
  - **Columns:** Tgt Vertex ID2, Non-index Properties

**a. Fetches edges originating from a certain vertex in order of index**
How to store the data - Edge

Physical View - table schema

1. Snapshot Edge

   a. Rowkey

<table>
<thead>
<tr>
<th>Murmur Hash</th>
<th>Src Vertex ID</th>
<th>Label ID</th>
<th>Direction</th>
<th>Index Sequence</th>
<th>Is Inverted</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>variable length</td>
<td>30 bit</td>
<td>2 bit</td>
<td>7bit</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

Vertex IDs can be encoded with 8 bit header + byte array (long, integer, short, byte, string)
## How to store the data - Edge

### Physical View - table schema

1. **Snapshot Edge**

   **b. Qualifier**
   - Target Vertex ID
     - variable length

   **c. Value**
   - All Property Key Value Pairs
     - variable length
# How to store the data - Edge

## Physical View - table schema

### 2. Indexed Edge

#### a. Rowkey

<table>
<thead>
<tr>
<th>Murmur Hash</th>
<th>Src Vertex ID</th>
<th>Label ID</th>
<th>Direction</th>
<th>Index Sequence</th>
<th>Is Inverted</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>variable length</td>
<td>30 bit</td>
<td>2 bit</td>
<td>7 bit</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

Vertex IDs can be encoded with 8 bit header + byte array (long, integer, short, byte, string)
## How to store the data - Edge

**Physical View - table schema**

### 2. Indexed Edge

<table>
<thead>
<tr>
<th>b. Qualifier</th>
<th>c. Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Property Values</td>
<td>Tgt Vertex ID</td>
</tr>
<tr>
<td>variable length</td>
<td>variable length</td>
</tr>
</tbody>
</table>
# How to store the data - Vertex

**Logical View**

1. **Vertex**: Up-to-date status of Vertex

<table>
<thead>
<tr>
<th>column</th>
<th>Property Key1</th>
<th>Property Key2</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Src Vertex ID1</td>
<td>Value1</td>
<td>Value2</td>
</tr>
<tr>
<td>Vertex ID2</td>
<td>Value1</td>
<td>Value2</td>
</tr>
</tbody>
</table>
### How to store the data - Vertex

**Physical View - table schema**

**1. Vertex**: Up-to-date status of Vertex

**a. Rowkey**

<table>
<thead>
<tr>
<th>Murmur Hash</th>
<th>Column ID</th>
<th>Vertex ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>integer(32bit)</td>
<td>variable length</td>
</tr>
</tbody>
</table>

**b. Qualifier**

- **Property Key**: Byte(8 bit)

**c. Value**

- **Property Value**: variable length
How to read the data - GetEdges

Using a custom query DSL on top of HTTP

curl -XPOST localhost:9000/graphs/getEdges -H 'Content-Type: Application/json' -d ' 
{ 
   "srcVertices": [{"serviceName": "s2graph", "columnName": "account_id", "id":1}], 
   "steps": [ 
      [{"label": "friends", "direction": "out", "limit": 100}]} // step 
      [{"label": "hear", "direction": "out", "limit": 10}] 
   ] 
} 
',

Steps = a list of Step
Step = contains the labels to traverse and how to rank them in the result
How to read the data - GetEdges Example

Friend list

curl -XPOST localhost:9000/graphs/getEdges -H 'Content-Type: Application/json' -d '{
  "srcVertices": [{"serviceName": "s2graph", "columnName": "account_id", "id":1}],
  "steps": [
    [{"label": "friends", "direction": "out", "limit": 100}], // step
  ]
}'
How to read the data - GetEdges Example

Songs my friends have listened

curl -X POST localhost:9000/graphs/getEdges -H 'Content-Type: Application/json' -d ' {
    "srcVertices": [{"serviceName": "s2graph", "columnName": "account_id", "id":1}],
    "steps": [
        {"label": "friends", "direction": "out", "limit": 50, "scoring": {"score": 1.0}}],
        {"label": "listen", "direction": "out", "limit": 10}]
} '

Reference : https://github.com/daumkakao/s2graph#1-definition
How to read the data - GetEdges Example

Similar songs to songs that I have listened to.

curl -XPOST localhost:9000/graphs/getEdges -H 'Content-Type: Application/json' -d ' 
{
    "srcVertices": [{"serviceName": "s2graph", "columnName": "account_id", "id":1}],
    "steps": [
        {
            "label": "listen", "direction": "out", "limit": 50},
        {
            "label": "similar_song", "direction": "out", "limit": 10, "scoring": {"score": 1.0}
        }
    ]
}
How to read the data - GetVertices

curl -X POST localhost:9000/graphs/getVertices -H 'Content-Type: Application/json' -d '{
    "serviceName": "s2graph", "columnName": "account_id", "ids": [1, 2, 3]},
    "serviceName": "kakaomusic", "columnName": "user_id", "ids": [1, 2, 3]}'
How to write the data - Insert

curl -XPOST localhost:9000/graphs/edges/insert -H 'Content-Type: Application/json' -d ' 
[  
  {"from":1,"to":2,"label":"graph_test","props":{"time":-1, "weight":10},"timestamp":1417616431},  
] ,
'
How to write the data - Delete

curl -XPOST localhost:9000graphs/edges/delete -H 'Content-Type: Application/json' -d '{
    "from":1,"to":2,"label":"graph_test","timestamp":1417616431},
    "from":1,"to":3,"label":"graph_test","timestamp":1417616431},
}'}
How to write the data - Update

curl -XPOST localhost:9000/graphs/edges/update -H 'Content-Type: Application/json' -d '
[  
"from":1,"to":2,"label":"graph_test","timestamp":1417616431,  
"props": {"is_hidden": true, "status": 200},
  
"from":1,"to":3,"label":"graph_test","timestamp":1417616431,  
"props": {"status": -500}
]
<table>
<thead>
<tr>
<th>Read</th>
<th>Write</th>
<th>Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. getEdges</td>
<td>1. insert</td>
<td>1. create service (vertex type)</td>
</tr>
<tr>
<td>2. checkEdge</td>
<td>2. delete</td>
<td>2. create label (edge type)</td>
</tr>
<tr>
<td>3. getEdgesCount</td>
<td>3. update</td>
<td>3. add Index</td>
</tr>
<tr>
<td>4. getVertices</td>
<td>4. increment</td>
<td></td>
</tr>
</tbody>
</table>
HBase Table Configuration

1. setDurability(Durability.ASYNC_WAL)
2. setCompressionType(Compression.Algorithm.LZ4)
3. setBloomFilterType(BloomType.Row)
4. setDataBlockEncoding(DataBlockEncoding.FAST_DIFF)
5. setBlockSize(32768)
6. setBlockCacheEnabled(true)
7. pre-split by (Integer.MaxValue / regionCount). regionCount = 120 when create table(on 20 region server).
HBase Cluster Configuration

- each machine: 8core, 32G memory
- hfile.block.cache.size: 0.6
- hbase.hregion.memstore.flush.size: 128MB
- otherwise use default value from CDH 5.3.1
Overall Architecture

- kafka
- Spark Streaming
- Clients
- S2graph
- Apache HBase
Compare to other Online GraphDBs

Titan (v0.4.2)

a. Pros
   - Rich API and easy to setup
   - Relatively large community
   - Transaction handling

b. Cons
   - Using its own ID system; less efficient for graph traversal (details in next slide)
   - Index data stored on one region (hotspot) with strong consistency option
   - Not many references on Titan with HBase comparing to other storages
Titan is less efficient for graph traversal

- For following 1 normal graph traversal query,
  
  `Vertex("userID: 1").out("friends").limit(10).out("friends").limit(10)`
**Compare to Titan (cont)**

Vertex("userID:1").out("friend").limit(10).out("friend").limit(10)

<table>
<thead>
<tr>
<th># of read requests on HBase</th>
<th>Titan</th>
<th>S2graph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>112</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>1 (Vertex Lookup : a)</td>
<td>1 (1step edges : e)</td>
</tr>
<tr>
<td></td>
<td>+ 1 (1st step edges : b)</td>
<td>+ 10 (2nd step edges : f)</td>
</tr>
<tr>
<td></td>
<td>+ 10 (2nd step edges : c)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ 100 (Destination Vertices : d)</td>
<td></td>
</tr>
</tbody>
</table>
Performance

1. Test data
   a. **Total # of Edges**: 9,000,000,000
   b. **Average # of adjacent edges per vertex**: 500
   c. **Seed vertex**: vertices that has more than 100 adjacent edges.

2. Test environment
   a. **Zookeeper server**: 3
   b. **HBase Masterserver**: 2
   c. **HBase Regionserver**: 20
   d. **App server**: 8 core, 16GB Ram
2. Linear scalability

- Benchmark Query: `src.out("friend").limit(50).out("friend").limit(10)`
- Total concurrency: $20 \times \# \text{of app server}$
Performance

3. Varying width of traverse (tested with a single server)

- Benchmark Query: `src.out("friend").limit(x).out("friend").limit(10)`
- Total concurrency = 20 * 1(# of app server)
Performance

3. Varying width of traverse (tested with a single server)

- Benchmark Query: `src.out("friend").limit(x).out("friend").limit(10)`
- Total concurrency = 20 * 1(# of app server)
4. Different query path (different I/O pattern)

- All query touch 1000 edges.
- Each step’s limit is on x axis.
- Can expect performance with given query’s search space.
5. Write throughput per operation on single app server

![Graph showing latency vs request per second for insert operation]
6. write throughput per operation on single app server
Stats

1. HBase cluster per IDC (2 IDC)
   - 3 Zookeeper Server
   - 2 HBase Master
   - 20 HBase Slave

2. App server per IDC
   - 10 server for write-only
   - 20 server for query only

3. Real traffic
   - read: over 10K request per second
     - now mostly 2 step queries with limit 100 on first step.
   - write: over 5k request per second

   * Deep traversal queries are not counted since it is in test stage for production
Connect Everything
Connect Everything
Through HBase!
Now Available As an Open Source

- https://github.com/daumkakao/s2graph
- Finding a mentor

Contact

- Taejin Chin : taejin.chin@gmail.com
- Doyoung Yoon : shom83@gmail.com
Appendix

3.5x performance improvement using Asynchbase

- Test seed edges have adjacent edges more than 100: 30millions
- Total concurrency: 20 * # of app server

- Native Client QPS
- Native Client Latency (ms)
- Asynchbase QPS
- Asynchbase Latency (ms)

3.5x performance improvement using Asynchbase