Zen: A Graph Data Model on HBase

Raghavendra Prabhu (RVP)

Xun Liu

HBase @ Pinterest - 2012

- Original use case: materialized home feed
- Replaced Redis
- Need: elasticity, high write load, serve from disk/SSD
- Challenges:
  - Running on public cloud (AWS)
  - User facing use case (MTTR, latency, fault tolerance etc.)

HBase @ Pinterest - 2013

- Need: highly elastic key-value store
  - Access from Python
  - Support “move fast”
  - Low operational overhead

Enter UMegaStore

Storage-as-a-Service: Key-value thrift API on top of HBase

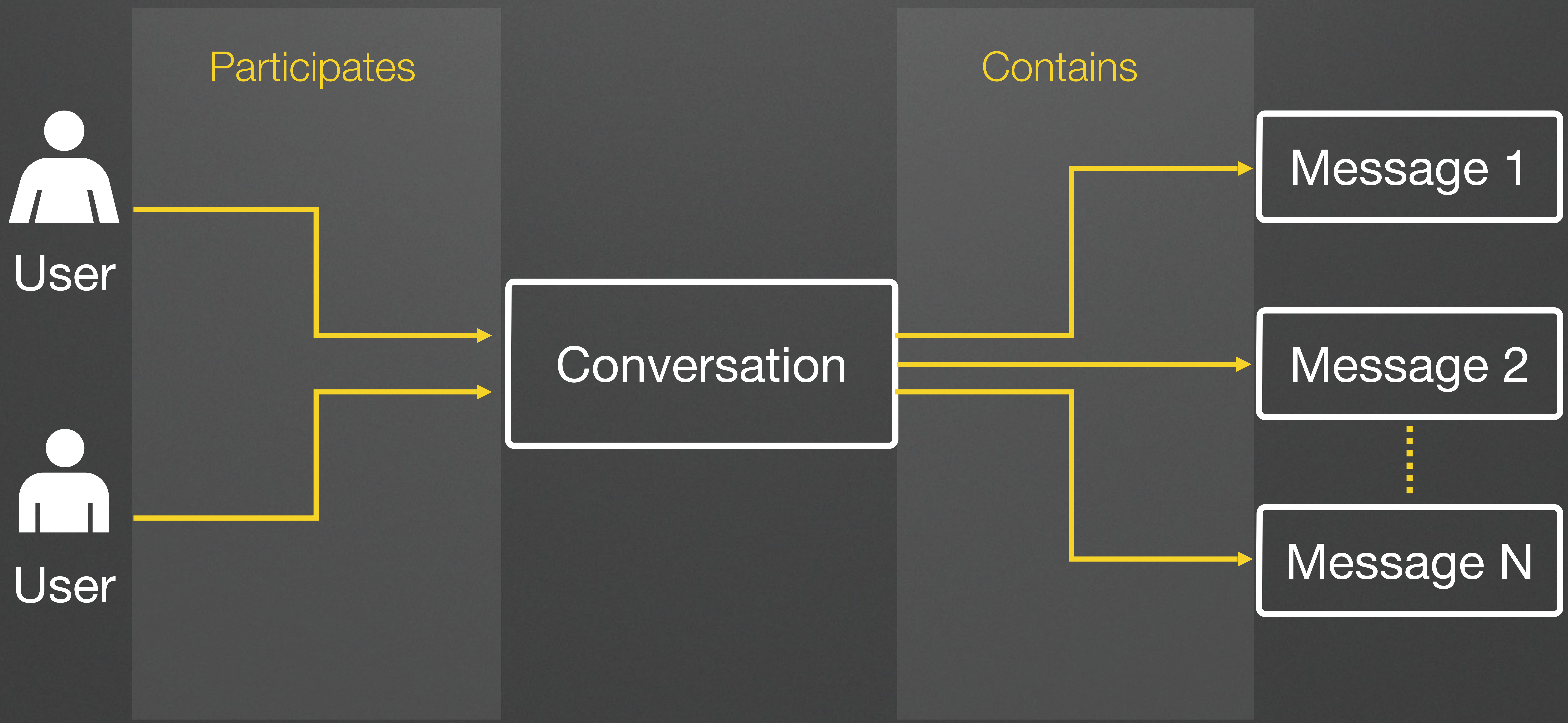
Features:

- Key partitioning to balance load
- Master-slave clusters, semi automatic failover
- Speculative execution
- Multi-tenancy with traffic isolation

Storage-as-a-service was a great step forward, but could we do better?

“Given how robust the messenger is on day one, it’s surprising to learn that Pinterest built the entire product in three months.” **— The Verge**

Example: Messages Data Model



Realization

- These object models closely resemble a graph
- Objects are nodes, edges represent relationships
- Typical needs:
  - retrieve data for a node or edge
  - get all *outgoing* edges from a node
  - get all *incoming* edges from a node
  - count *incoming* or *outgoing* edges for a node

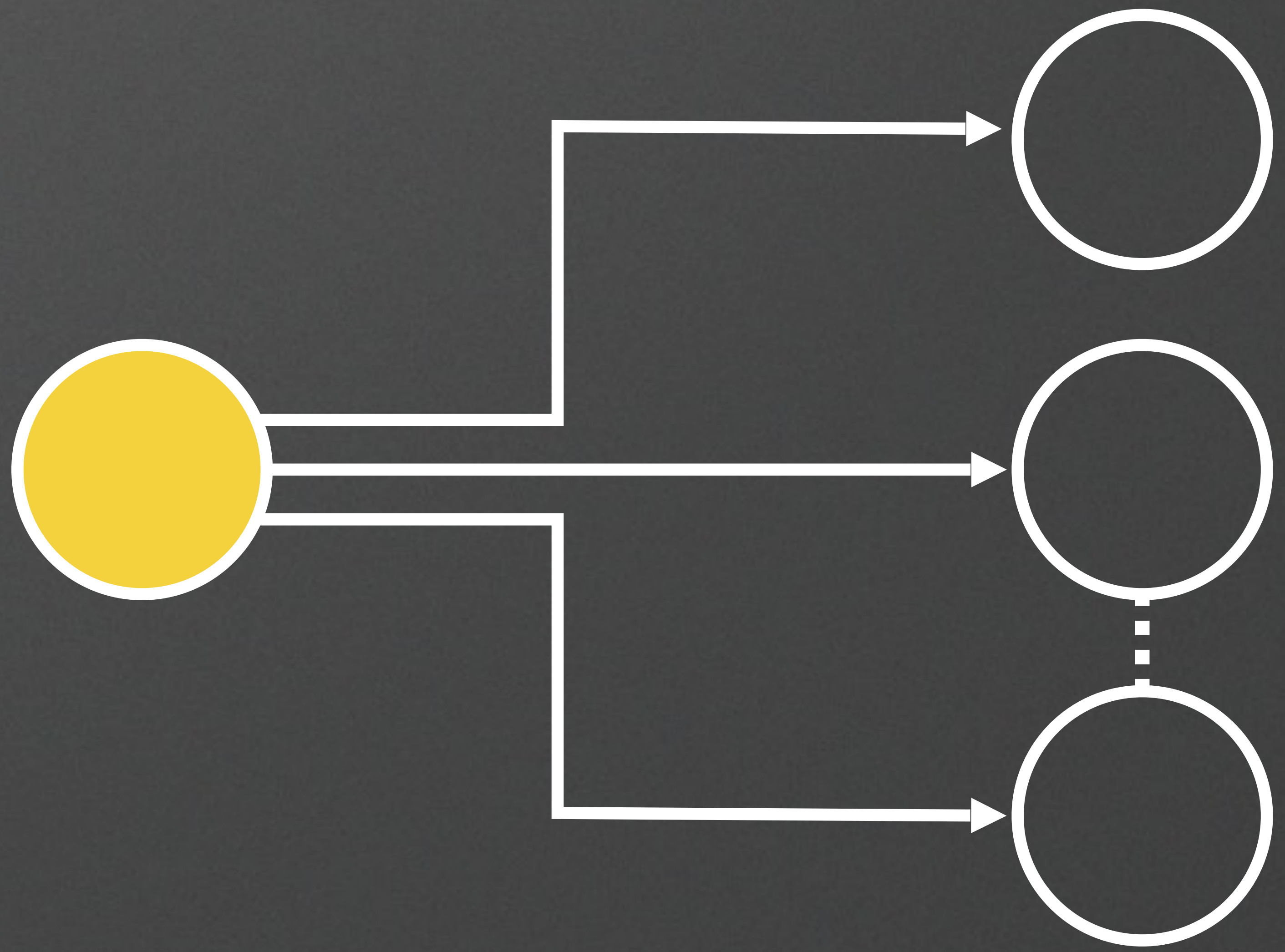
Enter Zen

- Provides a *graph* data model instead of key-value
- Automatically creates necessary indexes
- Materializes counts for efficient querying
- Implemented on top of HBase

Zen API

Nodes:

- addNode, removeNode, getNode
- Node id: globally unique 64-bit integer

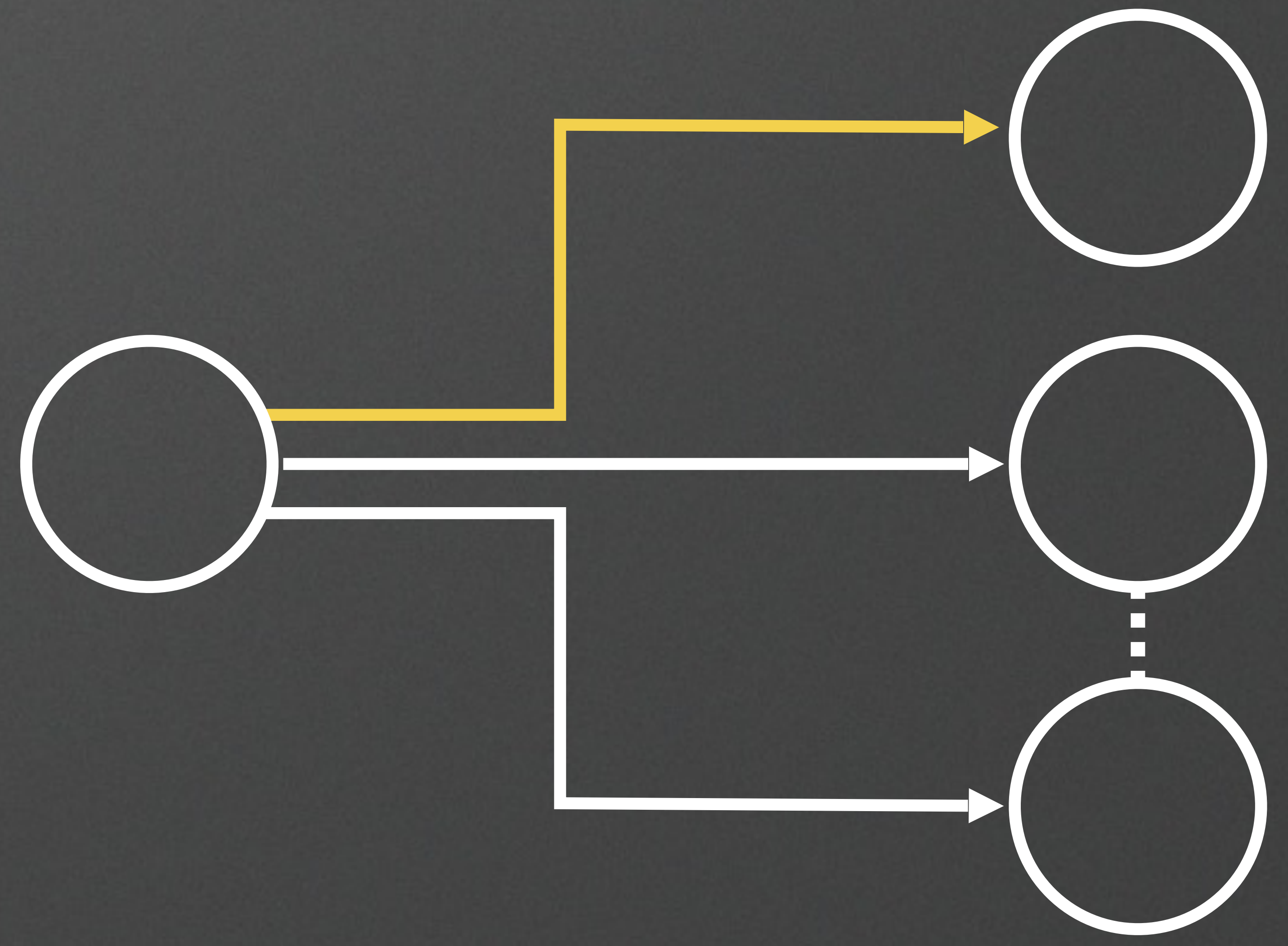


ID	123
Prop 1	Val 1
Prop 2	Val 2
⋮	⋮

Zen API

Edges:

- addEdge, removeEdge, getEdge
- Edge Ref: (edgeType, fromId, toId)
- Score for ordering



Edge Ref	**120, 123, 4567**
Prop 1	**Val 1**
Prop 2	**Val 2**
⋮	⋮

Zen API

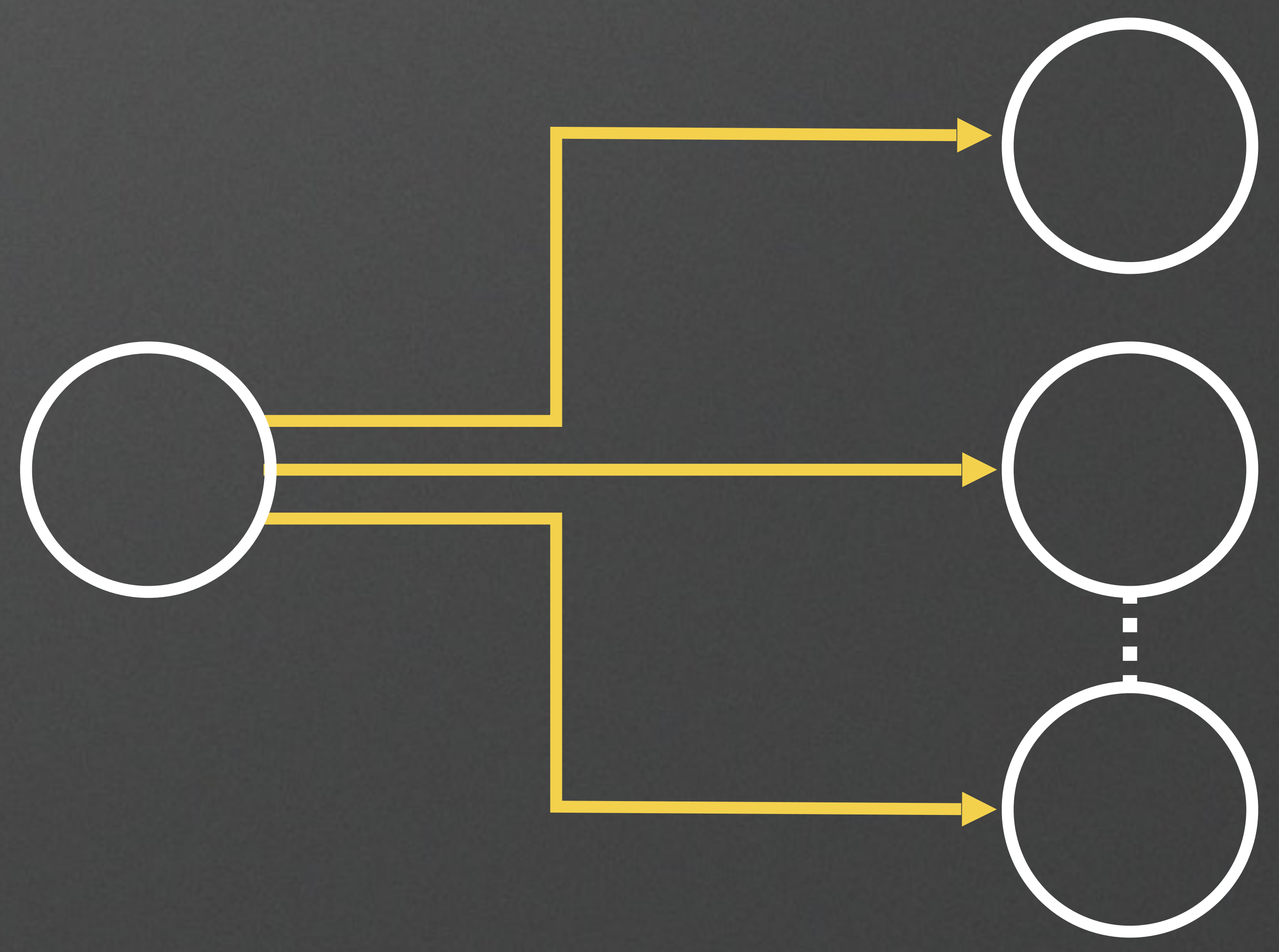
Edge Queries:

- getEdges, countEdges, removeEdges

```
struct EdgeQuery {
  1: required NodeId nodeId;

  2: required EdgeDirection direction;

  3: optional TypeId edgeType;
}
```



Zen API

Property Indexes

- Unique index
  - Ensures a property value is unique across all nodes of a type
- Non-unique index
  - Allows retrieval by property value
- Works for both nodes and edges

Illustration: Messages on Zen

Type: User
Name: “Ben Smith” **[unique]**
Status: Active

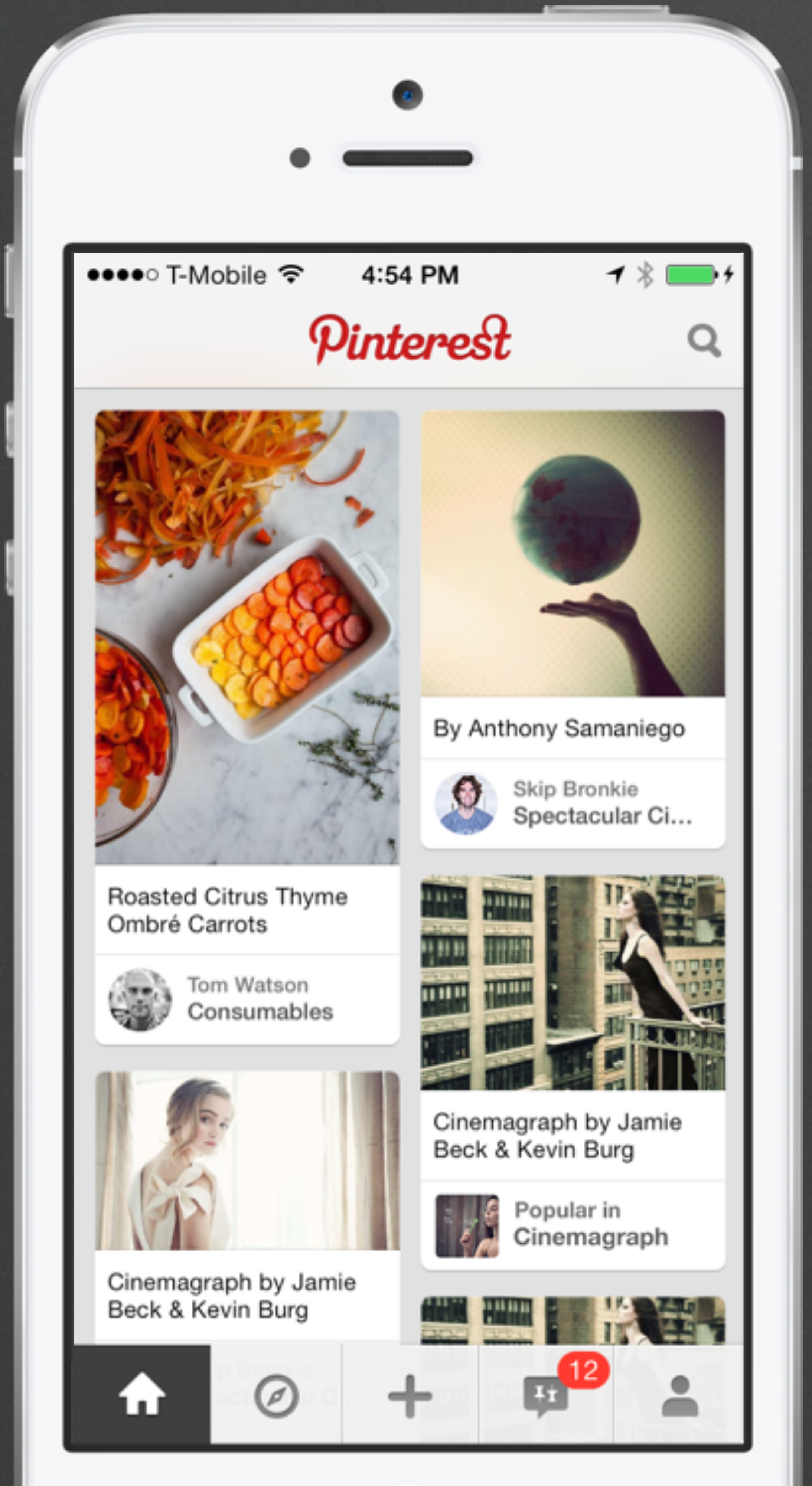
Type: Conversation
Started: 12 Aug 2014 08:00
Header: “Great pin!”
Pin Id: 10001 **[non-unique]**

Type: Message
Sent: 12 Aug 2014 08:00
Text: “Great pin!”

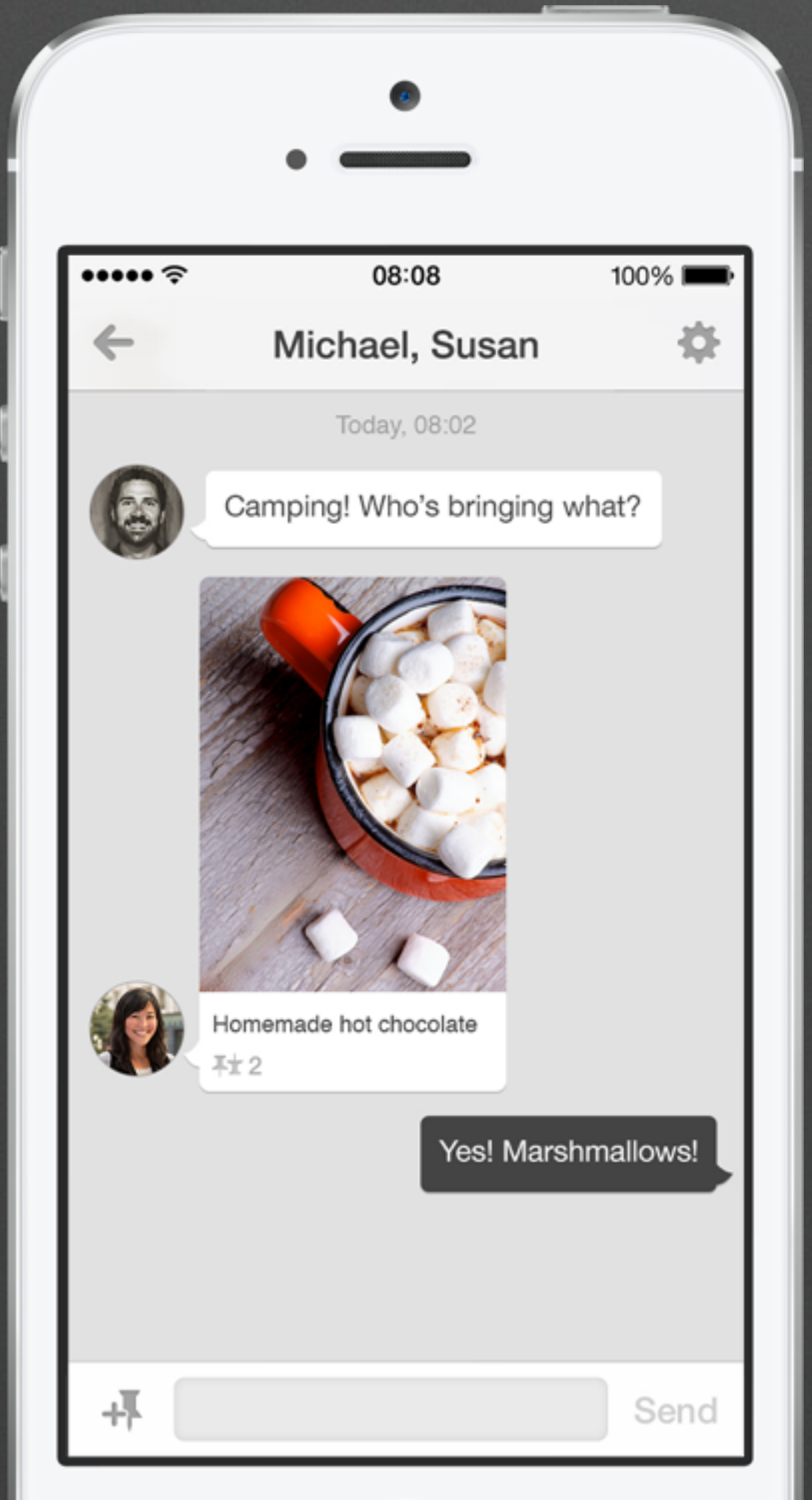
Zen @ Pinterest - 2014

Close to **10 million** operations every second

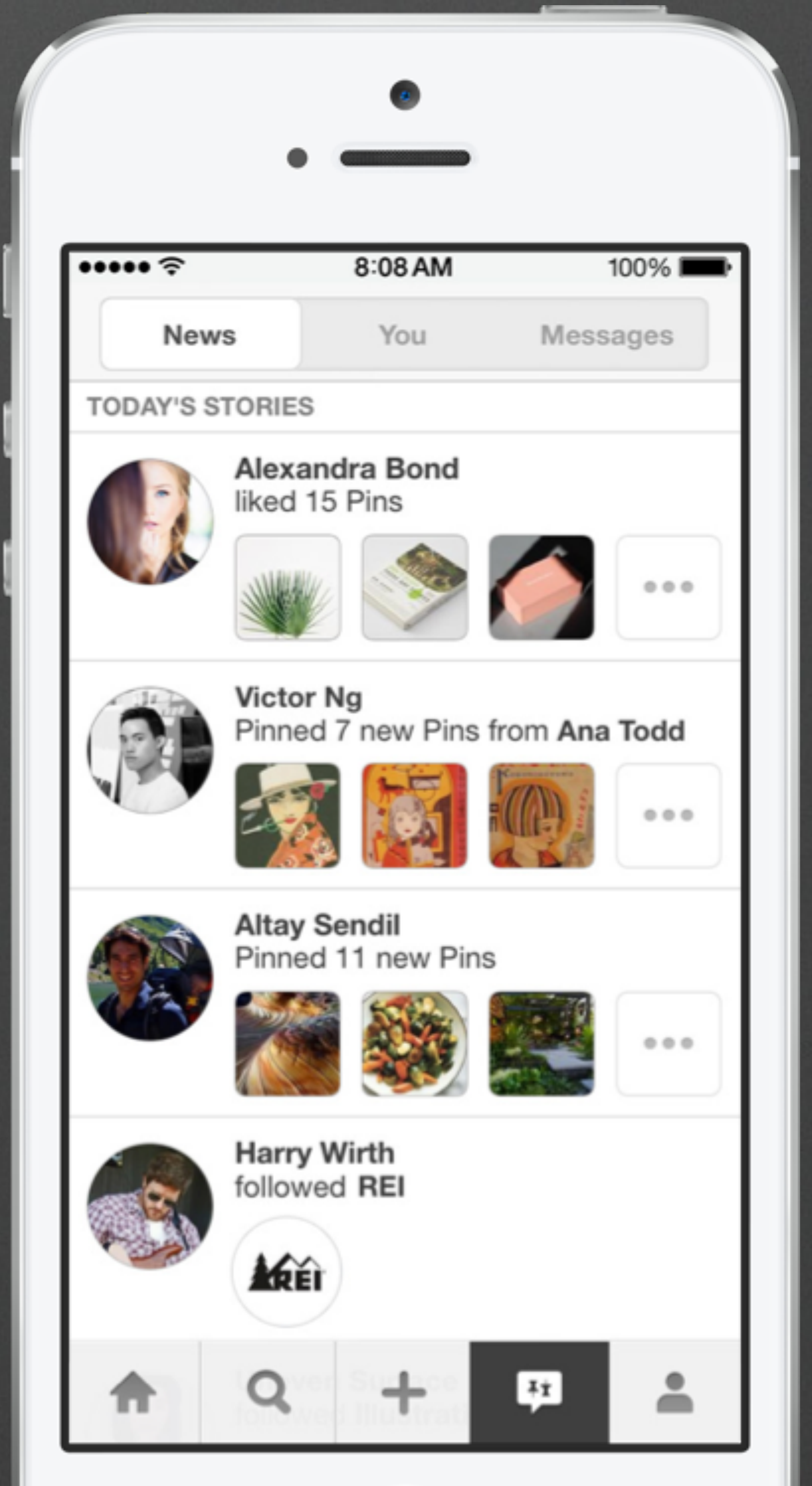
Smart Feed



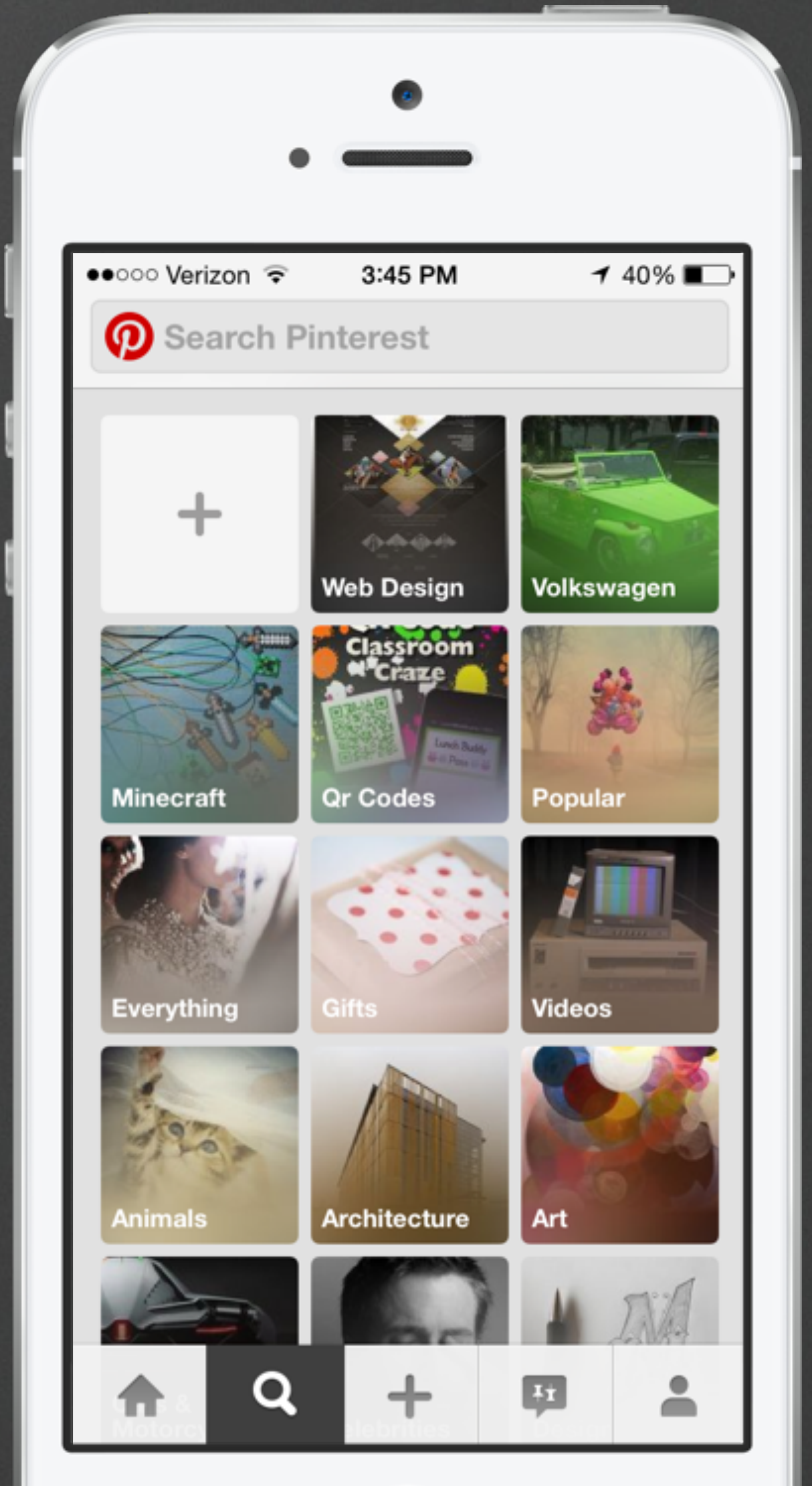
Messages



News



Interests



Zen Schema Design

Zen - Property

	Data			
	type	name	score	distance
12345 (node)	10	Ben Smith		
12345-20-67890 (edge)			1000	1 mile

Zen - Property Index

	Data
	ID
unique-10-name=ben smith	12345
nonuniq-10-lastname=smith-12345	
nonuniq-10-lastname=smith-67890	

Zen - Edge Score Index

	Data
12345-out-20-1000-67890	
12345-out-20-1001-67891	
12345-in-30-990-67892	
12345-in-30-991-67893	

Zen - Edge Count

	Data
	Count
12345-out-20	2
12345-in-30	4

Production Learnings

1. Avoid Hot Regions

- Salt row keys to achieve uniform distribution
  - Reverse bits of auto increment integers
  - Prepend hash to row keys
- Pre-split regions using uniform split
- Tall table

2. Batch For Throughput

- Bottleneck: HLog sync
- Deferred HLog sync can lose data
- Batching:
  - Client-side: batch APIs for clients to do bulk insertion
  - Server-side: Zen Server to buffer edits across clients and flush together

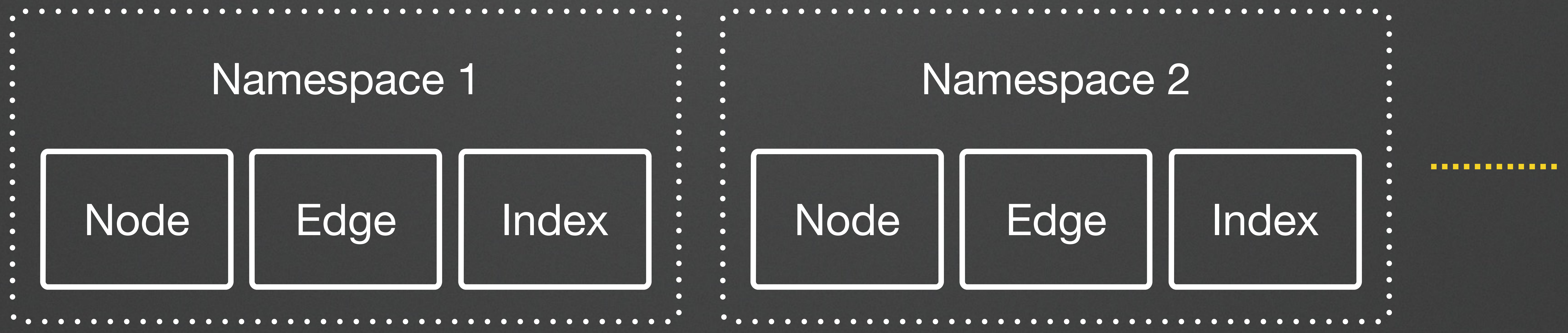
3. Tune For Performance

- Memory v.s. Latency
  - Default: BLOOMFILTER => ‘ROW’, BLOCKSIZE => '8192'
  - Special: BLOOMFILTER => ‘NONE’, BLOCKSIZE => ‘32768'

- CPU v.s. Data Size
  - Default: DATA_BLOCK_ENCODING => ‘FAST_DIFF’, COMPRESSION => ‘SNAPPY'
  - Special: DATA_BLOCK_ENCODING => ‘PREFIX’, COMPRESSION => ‘NONE’

4. Namespace for Isolation

- Dedicated HBase cluster for big applications
- Shared cluster with dedicated namespace for smaller ones



5. Coprocessor For Efficiency

- Use Case: remove a large number of edges (feeds)
- Usual Way:
  - Scan all edges of a node
  - Delete edges beyond a limit
  - Major compact to remove delete markers
- Coprocessor
  - Trim in a major compaction coprocessor

6. Best Effort Data Consistency

- No distributed transaction: keep things simple and fast
- Best effort to maintain data consistency:
    - Manual rollback in Zen server upon failures
    - Offline MapReduce jobs (Dr Zen) to scan and fix inconsistencies

Takeaways

- The graph data model can be a convenient abstraction to cut down product development time.
- HBase has worked very well as a storage backend for Zen for large scale user facing workloads.


thank you!