NoSQL Data Stores in Internet-scale Computing

Wei Tan

wtan@us.ibm.com
IBM T. J. Watson Research Center, New York, USA
Introduction + Aim of tutorial

- **About me**
  - Researcher in IBM T. J. Watson Research Center
  - 10+ years in distributed/grid/cloud computing
  - 3+ years in HBase, Hadoop and Spark; knowledge on other NoSQL stores
  - Contributed to IBM InfoSphere BigInsights (secondary index of HBase)

- **Aim of the tutorial**
  - **Why** NoSQL is so popular now?
  - **What** are the major features/products?
  - **How** they are implemented (aka., key technologies)
Introduction + Aim of tutorial

- After the tutorial, I hope you can:
  - Understand the NoSQL landscape
  - Know (some of) the essential aspects of NoSQL
    - User data model
    - Storage model
    - Data partition
    - Transaction semantics
  - Know where to find further information

- I try to make the tutorial both deep and interesting
  - Too much fluffy stuff on the Internet
  - NO exhausted visit of existing systems – we select representatives and focus on essential aspects
  - NO coverage of installation, APIs, “Me Too” features, … what you can easily find from a user manual

- Please raise question any time!
  - In some aspects you may have a better knowledge than me
  - And, three hours is long …
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

- HBase (table)
- Dynamo (K/V)
- MongoDB (json/doc)
- Neo4j (graph)

Analytics on NoSQL: Hadoop & Spark

Summary

Four features of NoSQL
Four categories of NoSQL
Four aspects to understand NoSQL
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

- HBase (table)
- Dynamo (K/V)
- MongoDB (json/doc)
- Neo4j (graph)

Four features of NoSQL
Four categories of NoSQL
Four aspects to understand NoSQL

Analytics on NoSQL: Hadoop & Spark

Summary
Why NoSQL?

- One size does not fit all
  - "traditional DBMS … originally designed and optimized for business data processing has been used to support many data-centric applications with widely varying characteristics and requirements" (M. Stonebraker, ICDE’05)

- Applications and requirements varies
  - Flexible and evolving schema
  - Store data in “native” format
    - JSON, XML, RDF, graph …
  - Simple applications demanding scalability and response
    - Game, social, location based apps …
  - Big data analytics demanding scale-out, affordable infrastructure
Why NoSQL (cont’d)?

- **Emerging of Big Data (from wikipedia)**
  - **Science**
    - Large Hadron Collider → 25 PB in 2012, 200 PB after replication
  - **Government**
    - Utah Data Center being constructed by NSA → (maybe) a few Exabytes
  - **Business**
    - eBay → 40PB Hadoop cluster for search and recommendation
    - Walmart: >1 million tranx per hour, DB > 2.5 petabytes
    - Facebook → 50 billion photos (in Haystack); messaging 25TB/month a while ago (in HBase)
Big Data Presents Big Opportunities

*Extract insight from a high volume, variety and velocity of data in a timely and cost-effective manner*

**Variety:** Manage and benefit from diverse data types and data structures

**Velocity:** Analyze streaming data and large volumes of persistent data

**Volume:** Scale from terabytes to zettabytes
Why NoSQL (cont’d)?

- RDBMS is not good at handling “big data”
  - Sharding data across many nodes
  - Run analytics in parallel across many nodes
  - Flexibility in schema
  - Flexibility in consistency
  - Cost
Why NoSQL: Hardware Evolution

- Inexpensive, commoditized hardware enables “scale-out” infrastructure
  - CPU: multi socket, core, thread …
  - Memory getting cheaper
  - Directly-attached storage, SSD

- Cloud is replacing a lot of in-house servers
  - Able to obtain a big infrastructure without upfront cost
  - Saves IT maintenance cost

- Software-defined everything (network, storage)
  - On commodity hardware, shared-nothing architecture
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

Four features of NoSQL
Four categories of NoSQL
Four aspects to understand NoSQL

HBase (table)
Dynamo (K/V)
MongoDB (json/doc)
Neo4j (graph)

Diff-Index

Analytics on NoSQL: Hadoop & Spark

Summary
NoSQL: What?

- NoSQL (Not Only SQL): non-RDBMS data stores usually with these features:
  - flexible schema
  - simple API
  - relaxed ACID
  - easy scale-out
  - on commodity hardware

- More than 150*, in these categories (see more on next page)
  - Document
    - Structured documents with variable fields (JSON, XML)
    - E.g., MongoDB, CouchDB
  - Key/Value
    - Value can be a complex object, but opaque to data store
    - Memcached, Berkeley DB, Dynamo, Amazon S3
  - Graph
    - Store nodes, edges and properties on them
    - Optimized for graph algorithms
    - Neo4J, Titan, DB2 RDF
  - Tabular: BigTable, HBase, Cassandra
    - Table with variable schema across rows
Four Features of NoSQL (the 1st “Four”)

Flexible schema

<table>
<thead>
<tr>
<th>rowA</th>
<th>col2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowB</td>
<td>col1</td>
<td>4</td>
</tr>
<tr>
<td>rowC</td>
<td>col1</td>
<td>1</td>
</tr>
<tr>
<td>rowC</td>
<td>col2</td>
<td>2</td>
</tr>
<tr>
<td>rowC</td>
<td>col3</td>
<td>6</td>
</tr>
<tr>
<td>rowF</td>
<td>col2</td>
<td>4</td>
</tr>
</tbody>
</table>

Simple API

<table>
<thead>
<tr>
<th>HTTP/REST</th>
<th>CRUD</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Create</td>
<td>INSERT</td>
</tr>
<tr>
<td>GET</td>
<td>Retrieve</td>
<td>SELECT</td>
</tr>
<tr>
<td>PUT</td>
<td>Update</td>
<td>UPDATE</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete</td>
<td>DELETE</td>
</tr>
</tbody>
</table>

Relaxed ACID

Scale-out on commodity HW

Availability

MySQL, Postgres, Vertica etc.

Cassandra, SimpleDB, Riak, Dynamo etc.

Pick Two

Consistency

MongoDB, HBase, BigTable, Redis etc.

Partition Tolerance
Four categories of NoSQL (the 2\textsuperscript{nd} “Four”)

### Tabular stores

- Google Bigtable
- HBase
- Cassandra
- Accumulo
- Hypertable

### Key/Value stores

- Dynamo
- Amazon S3
- Dropbox
- Memcached
- Riak

### Document stores

```json
{
  "firstName": "Wei",
  "lastName": "Tan",
  "age": 32,
  "phoneNumber": {
    "type": "office",
    "number": "914-784-7100"
  }
}
```

### Graph stores

- MongoDB
- Neo4j
- TITAN
- Lotus Notes
- CouchDB

- A Graph
- Nodes
- Relationships
- Properties
  - have
  - organize
  - have

Tutorial at IEEE ICWS, June 27, 2014, Alaska, USA
Market: NoSQL is still in adolescence

- **RDBMS:** $26 billion with about 9% annual growth
  – Gartner, 2013

- **NoSQL:** to reach **$3.4 billion** in 2020, representing a compound annual growth rate (CAGR) of **21%** for the period 2015 – 2020.
  – NoSQL Market Forecast 2015-2020, Market Research Media
Hadoop, Spark …

Figure from http://blogs.the451group.com/information_management/2014/03/18/updated-data-platforms-landscape-map-february-2014/
NoSQL on Cloud

- SimpleDB
- DynamoDB
- Couchbase
- MongoDB

Google Cloud Platform

Swift
- openstack
- Cloudant
- MongoDB
- Riak

IBM

- Datastore
- Megastore
- Bigtable
- Scalable and reliable storage

Block Blobs
- Microsoft Azure

Tables
Another Perspective: CAP Theorem (Eric Brewer, 2000)

- **Consistency**
  - all nodes see the same data at the same time

- **Availability**
  - every request to a non-failing node receives a response

- **Partition tolerance**
  - system allows arbitrary message loss

- All desirable but **impossible to achieve all three**

```
req1: a=2
    Node 1
   
 req2: a=?
    Node 2
```

```
msg

a=1

Node 1

msg

a=1

Node 2
```
Another Perspective: CAP Theorem (Eric Brewer, 2000)

- **Consistency**
  - all nodes see the same data at the same time

- **Availability**
  - every request to a non-failing node receives a response

- **Partition tolerance**
  - system allows arbitrary message loss

- All desirable but **impossible to achieve all three**
  - **AC \(\rightarrow\) !P**: A (req2 returns), C (a=2) \(\rightarrow\) cannot lose msg

```
req1: a=2

Node 1
a=1
msg

Node 2
a=1
```

Tutorial at IEEE ICWS, June 27, 2014, Alaska, USA
Another Perspective: CAP Theorem (Eric Brewer, 2000)

- **Consistency**
  - all nodes see the same data at the same time

- **Availability**
  - every request to a non-failing node receives a response

- **Partition tolerance**
  - system allows arbitrary message loss

- All desirable but **impossible to achieve all three**
  - **AC → !P:** A (req2 returns), C (a=2) → cannot lose msg
  - **CP → !A:** C(a=2), P (lose msg) → req2 need to stall until msg arrives
Another Perspective: CAP Theorem (Eric Brewer, 2000)

- **Consistency**
  - all nodes see the same data at the same time

- **Availability**
  - every request to a non-failing node receives a response

- **Partition tolerance**
  - system allows arbitrary message loss

- All desirable but **impossible to achieve all three**

  - **AC** → **!P**: A (req2 returns), C (a=2) → cannot lose msg
  - **CP** → **!A**: C(a=2), P (lose msg) → req2 need to stall until msg arrives
  - **AP** → **!C**: A(req2 returns), P (lose msg) → a=1 inconsistent

```
req1: a=2
  a=1
   msg
  a=1

Node 1  Node 2
```

Tutorial at IEEE ICWS, June 27, 2014, Alaska, USA
NoSQL Systems through the “CAP Glass”

*Figure from Nathan Hurst: http://blog.nahurst.com/visual-guide-to-nosql-systems
NoSQL Lineage

- Google Bigtable
- HBase
- Cassandra
- Amazon Dynamo
- Riak
- CouchDB
- MongoDB
- Neo4j

Tabular

K/V

Document

Graph

2005 2006 2007 2008 2009 2010 2011 2012
### RDBMS vs. NoSQL

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>NoSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ACID</strong>: Atomicity, Consistency, Isolation, Durability</td>
<td><strong>BASE</strong>: Basically Available, Soft State, Eventually consistent</td>
</tr>
<tr>
<td>Strong consistency (2 PC, Paxos)</td>
<td>Eventual consistency</td>
</tr>
<tr>
<td>Complex operations (joins, queries)</td>
<td>Simple operations (CRUD)</td>
</tr>
<tr>
<td>Availability through scale-up</td>
<td>Availability through scale-out</td>
</tr>
<tr>
<td>Scalability limited</td>
<td>Highly scalable</td>
</tr>
<tr>
<td>Mission-critical apps (e.g., core banking)</td>
<td>Web 2.0, Internet companies</td>
</tr>
</tbody>
</table>

No Free Lunch Theorem (in optimization): “any two optimization algorithms are equivalent when their performance is averaged across all possible problems”

→ RDBMS and NoSQL are suitable for different types of problems (applications)
From: http://blogs.the451group.com/information_management/?s=nosql+linkedin+skills
From: http://blogs.the451group.com/information_management/?s=nosql+linkedin+skills
Four aspects to understand NoSQL (the 3rd “Four”)

Data model (seen by end-users)

- table
- doc
- graph
- hashmap

Single node storage

- Sequence file: Key, Value, Key, Value, Key, Value
- B+ Tree
- Linked list
- LSM tree

Partitioning (sharding) scheme and metadata

- Range Partitioning
- Hash Partitioning

Transaction semantics

- Atomicity in what granularity
- Consistency level: strong, causal, session, eventual?
- Concurrency: locking, multi-version?
- Replication
- Availability
- Failover
- …
NoSQL Stores in Detail

Tabular stores

<table>
<thead>
<tr>
<th>rowA</th>
<th>col1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowB</td>
<td>col1</td>
<td>4</td>
</tr>
<tr>
<td>rowC</td>
<td>col1</td>
<td>1</td>
</tr>
<tr>
<td>rowC</td>
<td>col2</td>
<td>2</td>
</tr>
<tr>
<td>rowC</td>
<td>col3</td>
<td>6</td>
</tr>
<tr>
<td>rowF</td>
<td>col2</td>
<td>4</td>
</tr>
</tbody>
</table>

Key/Value stores

- Key: User1, Value: Mike
- Key: User2, Value: John
- Key: User3, Value: Mary

Document stores

```json
{
  "firstName": "Wei",
  "lastName": "Tan",
  "age": 32,
  "phoneNumber": {
    "type": "office",
    "number": "914-784-7100"
  }
}
```

Graph stores

Graph data relationships

Neo4j - the graph database

TITAN
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

HBase (table)

Dynamo (K/V)

MongoDB (json/doc)

Neo4j (graph)

Diff-Index

Analytics on NoSQL: Hadoop & Spark

Summary

Four features of NoSQL
Four categories of NoSQL
Four aspects to understand NoSQL
Apache HBase: Overview

- Modeled after Google BigTable (OSDI 06)
- Open source, written in Java
- Runs on top of Hadoop File System (HDFS); Hadoop MapReduce integration
- Strongly consistent (a “CP” system from CAP’s perspective)
- Range partitioned, auto-sharding, horizontally scalable
- Notable users
  - Facebook: message and operational data store
  - TaoBao: transaction history, browse history
  - Many others, see www.hbasecon.com/
Apache HBase: Overview

Data model (seen by end-users)

- Multi-versioned
- One column family can have variable number of columns

Partitioning (sharding) scheme and metadata

- Range partitioned into regions—good for range query
- Each region has a dedicated server
- Metadata stored in the same format as data tables

Transaction semantics

- Row level atomicity and strong consistency
- Concurrency: multi-version so read is not blocked
- Replication through HDFS
- Availability: interruption when region server fails

Single node storage

LSM (log structured merge) tree

- Put
- Get
- Memory
- Storage device
- WAL
- HFile
- Block Index

Cells within a column family are sorted physically

Very Sparse, most cell has NULL value
A table is a collection of rows
- SortedMap<RowKey, List<SortedMap<Column, List<Value, Timestamp>>>>
- All rows are unique, and sorted lexicographically by their row key

Columns and Column Families
- A row can have millions of columns
- Columns grouped into column families (CF); data in a column family are stored physically together in a storage file
- Reference to columns: family:qualifier

Column/Cell
- Every column value, or cell, is time-stamped (implicitly or explicitly)
HBase Client API

- **CRUD**
  - Put: add a new record; do NOT distinguish insert and update
  - Get: by row key; cannot get by column value
  - Delete: add a delete marker

- **Scan**
  - Iteration over ranges of rows
  - Can specify which rows/columns to return, and versions

- **Read-modify-write**
  - Atomic read-modify-write on data stored in a single row key

- **Counters**
  - Values can be interpreted as counters and updated atomically

- **Coprocessors**
  - Triggers and stored-procedures
  - Allow to push user code in the address space of the server
  - Usage: data pre-processing, filtering, summarization
Log Structure Merge (LSM) Trees

- LSM Tree = an in-memory store + several on-disk stores

[O'Neil, Acta Informatica'96]
Log Structure Merge (LSM) Trees

- LSM Tree = in-memory store + several on-disk stores
- Writes go to a commit log (seq. IO) and in-memory store – update not in-place, FAST

[O'Neil, Acta Informatica'96]
Log Structure Merge (LSM) Trees

- LSM Tree = a in-memory store + several on-disk stores
- Writes go to a commit log (seq. IO) and in-memory store – update not in-place, FAST
- Memstore periodically pushed to disk

[O'Neil, Acta Informatica'96]
Log Structure Merge (LSM) Trees

- LSM Tree = a in-memory store + several on-disk stores
- Writes go to a commit log (seq. IO) and in-memory store – update not in-place, FAST
- Memstore periodically pushed to disk
- Reads go to mem+disk stores (random IO) - SLOW

[O'Neil, Acta Informatica'96]
Log Structure Merge (LSM) Trees

- LSM Tree = a in-memory store + several on-disk stores
- Writes go to a commit log (seq. IO) and in-memory store – update not in-place, FAST
- Memstore periodically pushed to disk
- Reads go to mem+disk stores (random IO) - SLOW
- On-disk stores periodically compacted to save space & speedup read

[O'Neil, Acta Informatica’96]
Log Structure Merge (LSM) Trees

- Commit log
- Mem Store
- Disk: C1, C2

Flush:
(memstore → disk store)

Compact:
(merge disk stores)

Storage
Data model
Partition
ACID

Overview
Data model
Storage
Partition
ACID

B+tree (RDB)
LSM tree (HBase)
logging

Reads:
Slow → Fast
Writes/Inserts:
Fast → Slow

[O'Neil, Acta Informatica'96]
HBase data structure: Log Structure Merge (LSM) Trees

- 50/50 Read/update
  
  ![Workload A - Read latency](image)
  ![Workload A - Update latency](image)

- 95/5 Read/update
  
  ![Workload B - Read latency](image)
  ![Workload B - Update latency](image)

- Write much faster than MySQL – write to memory.
- Read latency is high – need to reconstruct data.

From: Brian F. Cooper. Yahoo! Cloud Serving Benchmark. Overview and results – March 31, 2010
HBase data structure: Log Structure Merge (LSM) Trees

Scans of 1-100 records of size 1KB

• HBase is good at scan due to the “sorted map” data structure

From: Brian F. Cooper. Yahoo! Cloud Serving Benchmark. Overview and results – March 31, 2010
HFile – on disk component
- The trailer the pointers to the other blocks
- Index blocks record the offsets of the data and meta blocks
- Block size: large → sequential access; small → random access
- **HBase**: range-based partitioning
  - Keys are sorted and each region serves a disjoint range of keys
  - Metadata: which servers contain which key ranges also stored as a table

- We will introduce hash-based partitioning in Dynamo
- **HBase: range-based partitioning**
  - Keys are sorted and each region serves a disjoint range of keys
  - Metadata: which servers contain which key ranges also stored as a table

BigTable-based Range Partitioning

To how store meta-data

- **Centralized:** HDFS
- **Partially dist:** BigTable, HBase, FDS,
- **Fully dist:** Cassandra, Dynamo
- **Atomicity**
  - All mutations are atomic within a row
  - No transaction across rows

- **Consistency and Isolation**
  - MVCC: read is never blocked and there is only write transaction
  - In one region,
    - write transactions commit strictly serially
    - scan exhibit snapshot isolation
  - Scans across multiple regions, do **not** exhibit *snapshot isolation*
    - Need a time/sequence oracle or clock synchronization

- **Durability**
  - Any operation that returns a "success" code will be made durable
  - Any operation that returns a "failure" code will not be made durable
HBase integration with Hadoop MR

Figure from: Lars George. HBase - The Definitive Guide. O'Reilly, 2011
HBase: Architectural Summary

Figure from: Lars George. HBase - The Definitive Guide. O'Reilly, 2011
Cons of HBase

- **Single point of failure**
  - on HDFS namenode and HBase master

- **Region unavailable in region server failure**
  - Cassandra, another BigTable clone with “Dynamo-like” behavior, is AP (available and partition tolerant) rather than CP (consistent and partition tolerant)
  - Facebook just announced HydraBase, HA version of HBase

- **Lack of features**, e.g., *secondary index*, transaction, SQL interface
Case study 1: Facebook Messages

- **Facebook Messages**
  - Combines chat, SMS, email, and Messages
  - 350 million users; 15+ billion messages per month
  - Chat: 300+ million users; 120+ billion messages per month
  - Used Cassandra, does not like eventual consistency; adapted HBase in 2010

- **Workload characteristics**
  - Temporal write surge: HBase is good at handling intensive write
  - Read-latest: improvements on compaction, adding Bloom Filter, timestamp aware HFile access

Example: Inbox Search

- Schema
  - Key: RowKey: userid, Column: word, Version: MessageID
  - Value: Auxillary info (like offset of word in message)

- Data is stored sorted by <userid, word, messageID>:

  User1:hi:17->offset1
  User1:hi:16->offset2
  User1:hello:16->offset3
  User1:hello:2->offset4
  ...
  User2:...
  User2:...
  ...

  Can efficiently handle queries like:

  - Get top N messageIDs for a specific user & word
  - Typeahead query: for a given user, get words that match a prefix

Nicolas Spiegelberg. Facebook Messages & HBase.
http://www.slideshare.net/brizzzdotcom/facebook-messages-hbase
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

- HBase (table)
- Dynamo (K/V)
- MongoDB (json/doc)
- Neo4j (graph)

Four features of NoSQL
Four categories of NoSQL
Four aspects to understand NoSQL

Analytics on NoSQL: Hadoop & Spark

Summary
Case Study 2: Diff-Index: Differentiated Index in HBase

• Purpose of this case study, to illustrate:
  – how to enhance a NoSQL store with “RDBMS features” – secondary index
  – how HBase’s data structure, CAP characteristics impact design and implementation

• More details:
  – Wei Tan, Sandeep Tata, Yuzhe Tang, Liana Fong. Diff-Index: Differentiated Index in Distributed Log-Structured Data Stores. (EDBT 2014)
Agenda

- Background: NoSQL, LSM and HBase
- Diff-Index schemes
- ACID properties
- System
- Performance
- Related work & Summary
Background: Apache HBase

- HBase is a widely used NoSQL store
  - Table-like and flexible schema, scale-out on commodity HW, integration with Hadoop
  - Use log-structure and good for high ingestion rate workload

- Gap: **HBase** is slow for *ad hoc* queries
  - Has no secondary index; query w/ table scan

- Two options for index in a partitioned data store like HBase
  - **Global**
    - index on entire table, partitioned on its own
    - no need to broadcast → good for selective queries
  - **Local**
    - one index co-located with each partition
    - need broadcast queries to each partition → costly; update fast, though
Log Structure Merge (LSM) Trees

- LSM Tree = an in-memory store + several on-disk stores

[O'Neil, Acta Informatica '96]
Log Structure Merge (LSM) Trees

- LSM Tree = a in-memory store + several on-disk stores
- Writes go to a commit log (seq. IO) and in-memory store – update not in-place, FAST

[O'Neil, Acta Informatica'96]
Log Structure Merge (LSM) Trees

- LSM Tree = an in-memory store + several on-disk stores
- Writes go to a commit log (seq. IO) and in-memory store – update not in-place, FAST
- Memstore periodically pushed to disk

[O'Neil, Acta Informatica'96]
Log Structure Merge (LSM) Trees

- LSM Tree = a in-memory store + several on-disk stores
- Writes go to a commit log (seq. IO) and in-memory store – update not in-place, FAST
- Memstore periodically pushed to disk
- Reads go to mem+disk stores (random IO) - SLOW

[O'Neil, Acta Informatica'96]
Log Structure Merge (LSM) Trees

- LSM Tree = an in-memory store + several on-disk stores
- Writes go to a commit log (seq. IO) and in-memory store – update not in-place, **FAST**
- Memstore periodically pushed to disk
- Reads go to mem+disk stores (random IO) - **SLOW**
- On-disk stores periodically compacted to save space & speedup read
Log Structure Merge (LSM) Trees

[O'Neil, Acta Informatica'96]

memory
Mem Store

disk
Commit log

C1
C2

memory
Mem Store'

disk
Commit log'

V1
V2
C2

V3
C3

disk
Commit log'

C1'

compact
merge disk stores

ACID
Schemes
Background
System
Performance

reads

B+tree (RDB)

LSM tree (HBase)

logging

write/inserts

Fast

Slow

Fast

Slow
HBase architecture

- Records range partitioned into regions
- Each region stores data in LSM
- Used in Facebook, Meetup, Splice Machine, …
Challenges of index maintenance in LSM

1. Log Structured Merge tree: a reviving interest in it
   a) Write workload 10~20% → > 50%: click streams, sensors, mobile…
   b) With non in-place update and slow read, index update can be slow

2. Distributed systems
   a) Distributed index maintenance needs coordination
   b) Performance/consistency tradeoff: CAP theorem

3. Failure recovery: adding another log?
Challenges of index maintenance in LSM

1. Log Structured Merge tree: a reviving interest in it
   a) Write workload 10~20% → > 50%: click streams, sensors, mobile…
   b) With non in-place update and slow read, index update can be slow

2. Distributed systems
   a) Distributed index maintenance needs coordination
   b) Performance/consistency tradeoff: CAP theorem

3. Failure recovery: adding another log?

**Diff-Index** -- differentiated secondary Index
a global index scheme on LSM-Tree with balanced performance and lightweight failure recovery cost
Index update operations in LSM

Task: index *review* by *stars*
Both data and index are stored as HBase tables

Key/Value

Data table (Review)

Index table (ReviewByStar)
Index update operations in LSM

Task: index *review* by *stars*
Both data and index are stored as HBase tables

Key/Value

Data table (Review)

- Rev1/3, t1
- Rev2/4, t1

Start from an empty table, at time *t1*

- Insert two new reviews -- Rev1 and Rev2

Index table (ReviewByStar)

<table>
<thead>
<tr>
<th>Stars</th>
<th>ReviewID</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Rev1</td>
</tr>
<tr>
<td>4</td>
<td>Rev2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReviewID</th>
<th>Text</th>
<th>Stars</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rev1</td>
<td>...</td>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>Rev2</td>
<td>...</td>
<td>4</td>
<td>...</td>
</tr>
</tbody>
</table>
Index update operations in LSM

Task: index \textit{review} by \textit{stars}
Both data and index are stored as HBase tables

Start from an empty table, at time \textbf{t1}

• Insert two new reviews -- Rev1 and Rev2
• Insert two index records for Rev1 and Rev2
Index update operations in LSM

Task: index *review* by *stars*
Both data and index are stored as HBase tables

Start from an empty table, at time *t1*

- Insert two new reviews -- Rev1 and Rev2
- Insert two index records for Rev1 and Rev2

At a later time *t2*: change the star of Rev1 to 5

- HBase has only one put() API for insert and update -- *different from RDBMS & SQL*
- Insert <Rev1/5> auto-invalidates <Rev1/3> -- *t1 < t2*
Index update operations in LSM

Task: index review by stars
Both data and index are stored as HBase tables

- Start from an empty table, at time \( t_1 \)
  - Insert two new reviews -- Rev1 and Rev2
  - Insert two index records for Rev1 and Rev2

- At a later time \( t_2 \): change the star of Rev1 to 5
  - HBase has only one put() API for insert and update -- different from RDBMS & SQL
  - Insert <Rev1/5> auto-invalidates <Rev1/3> -- \( t_1 < t_2 \)
Index update operations in LSM

Task: index review by stars
Both data and index are stored as HBase tables

- Start from an empty table, at time $t_1$
  - Insert two new reviews -- Rev1 and Rev2
  - Insert two index records for Rev1 and Rev2

- At a later time $t_2$: change the star of Rev1 to 5
  - HBase has only one put() API for insert and update -- different from RDBMS & SQL
  - Insert $<\text{Rev1/5}>$ auto-invalidates $<\text{Rev1/3}>$ -- $t_1 < t_2$
  - Index record insert: $<5/\text{Rev1}>$
  - $<5/\text{Rev1}>$ does NOT invalidate stale index $<3/\text{Rev1}>$
Index update operations in LSM

Task: index review by stars
Both data and index are stored as HBase tables

Key/Value

Rev1/5, t2

Data table (Review)
Rev1/3, t1
Rev2/4, t1

Index table (ReviewByStar)

5/Rev1, t2
3/Rev1, t1
4/Rev2, t1

Start from an empty table, at time t1
• Insert two new reviews -- Rev1 and Rev2
• Insert two index records for Rev1 and Rev2

At a later time t2: change the star of Rev1 to 5
• HBase has only one put() API for insert and update -- different from RDBMS & SQL
• Insert <Rev1/5> auto-invalidates <Rev1/3> -- t1 < t2
• Index record insert: <5/Rev1>
• <5/Rev1> does NOT invalidate stale index <3/Rev1>

Vanilla solution: read data table to get old value ("3") and delete index <3/Rev1>

index_update = insert + read + del (read is costly!)
Index update operations in LSM

Task: index review by stars
Both data and index are stored as HBase tables

Start from an empty table, at time $t_1$
- Insert two new reviews -- Rev1 and Rev2
- Insert two index records for Rev1 and Rev2

At a later time $t_2$: change the star of Rev1 to 5
- HBase has only one put() API for insert and update -- different from RDBMS & SQL
  - Insert <$Rev1/5$> auto-invalidate <$Rev1/3$> -- $t_1 < t_2$
  - Index record insert: <$5/Rev1$>
  - <$5/Rev1$> does NOT invalidate stale index <$3/Rev1$>

Vanilla solution: read data table to get old value (“3”) and delete index <$3/Rev1$>

$\text{index\_update} = \text{insert} + \text{read} + \text{del}$ (read is costly!)

Tune the above equation to shorten the latency (see next slides)?
Index schemes by revisiting the equation:
index_update = insert + read + del

Put a new <k/v'> when an old <k/v> exists (think of k as ReviewID, v as Star in Yelp)

<table>
<thead>
<tr>
<th>Stars</th>
<th>ReviewID</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReviewID</th>
<th>Text</th>
<th>Stars</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R00001</td>
<td>...</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Index schemes by revisiting the equation:
\[\text{index\_update} = \text{insert} + \text{read} + \text{del}\]

Put a new \(<k/v'>\) when an old \(<k/v>\) exists (think of \(k\) as ReviewID, \(v\) as Star in Yelp)

<table>
<thead>
<tr>
<th>Stars</th>
<th>ReviewID</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReviewID</th>
<th>Text</th>
<th>Stars</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00001</td>
<td>...</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. sync-full: read data tbl to get old value “v”; del index \(<v/k>\) using it

\[\text{sync\-full} = \text{insert} + \text{read} + \text{del}\]
Index schemes by revisiting the equation: \( \text{index\_update} = \text{insert} + \text{read} + \text{del} \)

Put a new \(<k/v'>\) when an old \(<k/v>\) exists (think of \(k\) as ReviewID, \(v\) as Star in Yelp)

<table>
<thead>
<tr>
<th>Stars</th>
<th>ReviewID</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReviewID</th>
<th>Text</th>
<th>Stars</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>R000001</td>
<td>...</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1. sync-full: read data tbl to get old value “\(v'\)”; del index \(<v/k>\) using it
   
   \[ \text{sync-full} = \text{insert} + \text{read} + \text{del} \]

2. sync-insert: let old \((<v/k>)\) in index;
   lazy clean: only when a query uses it
   
   \[ \text{sync-insert} = \text{insert} \]
Index schemes by revisiting the equation:
\[
\text{index\_update} = \text{insert} + \text{read} + \text{del}
\]

Put a new \(<k/v'>\) when an old \(<k/v>\) exists (think of \(k\) as ReviewID, \(v\) as Star in Yelp)

1. **sync-full**: read data tbl to get old value “\(v'\)”; del index \(<v/k>\) using it
   
   \[
   \text{sync-full} = \text{insert} + \text{read} + \text{del}
   \]

2. **sync-insert**: let old \(<v/k>\) in index; lazy clean: only when a query uses it
   
   \[
   \text{sync-insert} = \text{insert}
   \]

3. **async**: add (insert + read + del) into asynchronous update queue (AUQ)
   
   \[
   \text{async} = 0 \text{ (insert + read + del)}
   \]

<table>
<thead>
<tr>
<th>Stars</th>
<th>ReviewID</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReviewID</th>
<th>Text</th>
<th>Stars</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>R000001</td>
<td>...</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
## Operation complexity analysis of Diff-Index schemes

<table>
<thead>
<tr>
<th>method</th>
<th>index operation</th>
<th>data insert</th>
<th>data read</th>
<th>index insert</th>
<th>index read</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-index</td>
<td>update</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1 sync-full</td>
<td>update</td>
<td>1</td>
<td>1</td>
<td>1+1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2 sync-insert</td>
<td>update</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>M</td>
<td>(M)</td>
<td>1</td>
</tr>
<tr>
<td>3 async</td>
<td>update</td>
<td>1</td>
<td>1 (Def.)</td>
<td>1+1 (Def.)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**sync-full = insert + read + del**

**sync-insert = insert**

**async = 0 (insert + read + del)**

**Operation update**= put a record into HBase

**Operation read**= point query with index access only

**Def.**= deferred

**M**= #rows matching the search key
### Operation complexity analysis of Diff-Index schemes

<table>
<thead>
<tr>
<th>method</th>
<th>index operation</th>
<th>data insert</th>
<th>data read</th>
<th>index insert</th>
<th>index read</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-index</td>
<td>update</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1 sync-full</td>
<td>update</td>
<td>1</td>
<td>1</td>
<td>1+1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2 sync-insert</td>
<td>update</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>M</td>
<td>(M)</td>
<td>1</td>
</tr>
<tr>
<td>3 async</td>
<td>update</td>
<td>1</td>
<td>1 (Def.)</td>
<td>1+1 (Def.)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**sync-full = insert + read + del**

**sync-insert = insert**

**async = 0 (insert + read + del)**

---

Operation update = put a record into HBase
Operation read = point query with index access only
Def. = deferred
M = #rows matching the search key
### Operation complexity analysis of Diff-Index schemes

<table>
<thead>
<tr>
<th>method</th>
<th>index operation</th>
<th>data insert</th>
<th>data read</th>
<th>index insert</th>
<th>index read</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-index</td>
<td>update</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1 sync-full</td>
<td>update</td>
<td>1</td>
<td>1</td>
<td>1+1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2 sync-insert</td>
<td>update</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>M</td>
<td>(M)</td>
<td>1</td>
</tr>
<tr>
<td>3 async</td>
<td>update</td>
<td>1</td>
<td>1 (Def.)</td>
<td>1+1 (Def.)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**sync-full** = insert + read + del
**sync-insert** = insert
**async** = 0 (insert + read + del)

---

Operation update=put a record into HBase
Operation read=point query with index access only
Def.=deferred
M=#rows matching the search key
## Operation complexity analysis of Diff-Index schemes

<table>
<thead>
<tr>
<th>method</th>
<th>index operation</th>
<th>data insert</th>
<th>data read</th>
<th>index insert</th>
<th>index read</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-index</td>
<td>update</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1 sync-full</td>
<td>update</td>
<td>1</td>
<td>1</td>
<td>1+1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2 sync-insert</td>
<td>update</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>M</td>
<td>(M)</td>
<td>1</td>
</tr>
<tr>
<td>3 async</td>
<td>update</td>
<td>1</td>
<td>1 (Def.)</td>
<td>1+1 (Def.)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**sync-full** = insert + read + del  
**sync-insert** = insert  
**async** = 0 (insert + read + del)  

---

Operation update=put a record into HBase  
Operation read=point query with index access only  
Def.=deferred  
M=#rows matching the search key
Session consistency: *read-your-own-write*

- A fine tuning of $\text{async} = 0$ (insert + read + del)
- Recall the Yelp example
Session consistency: *read-your-own-write*

- A fine tuning of \( \text{async} = 0 \) (insert + read + del)
- Recall the Yelp example

**Timeline**

User 1

- \( t1 \). View reviews for business A

User 2

- \( t1 \). View reviews for business B

**Table:**

<table>
<thead>
<tr>
<th>BusinessID</th>
<th>ReviewID</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Table:**

<table>
<thead>
<tr>
<th>ReviewID</th>
<th>Text</th>
<th>Stars</th>
<th>UserID</th>
<th>BusinessID</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Session consistency: *read-your-own-write*

- A fine tuning of $\text{async} = 0$ (insert + read + del)
- Recall the Yelp example

User 1

- $t_1$. View reviews for business A
- $t_2$. Post review for business A

User 2

- $t_1$. View reviews for business B

<table>
<thead>
<tr>
<th>BusinessID</th>
<th>ReviewID</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReviewID</th>
<th>Text</th>
<th>Stars</th>
<th>UserID</th>
<th>BusinessID</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>...</td>
<td>5</td>
<td>U1</td>
<td>A</td>
</tr>
</tbody>
</table>

Timeline:

1. User 1
   - $t_1$. View reviews for business A
   - $t_2$. Post review for business A
2. User 2
   - $t_1$. View reviews for business B
Session consistency: *read-your-own-write*

- A fine tuning of $async = 0$ (insert + read + del)
- Recall the Yelp example
  
<table>
<thead>
<tr>
<th>ReviewID</th>
<th>Text</th>
<th>Stars</th>
<th>UserID</th>
<th>BusinessID</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>...</td>
<td>5</td>
<td>U1</td>
<td>A</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**User 1**
- t1. View reviews for business A
- t2. Post review for business A
- t3. View reviews for business A

**User 2**
- t1. View reviews for business B
- t3. View reviews for business A

**Cannot see R01**

**Cannot see R01**
Session consistency: *read-your-own-write*

- A fine tuning of \( \text{async} = 0 \) (insert + read + del)
- Recall the Yelp example

User 1
- t1. View reviews for business A
- t2. Post review for business A
- t3. View reviews for business A

User 2
- t1. View reviews for business B
- t3. View reviews for business A

<table>
<thead>
<tr>
<th>BusinessID</th>
<th>ReviewID</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>R01</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Session cache “local” to a user
Session consistency: *read-your-own-write*

- A fine tuning of \( \text{async} = 0 \) (insert + read + del) \( \rightarrow \) async-session

- Recall the Yelp example

  User 1
  - t1. View reviews for business A
  - t2. Post review for business A
  - t3. View reviews for business A

  Session cache
  "local" to a user

<table>
<thead>
<tr>
<th>BusinessID</th>
<th>ReviewID</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>R01</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

  User 2
  - t1. View reviews for business B
  - t3. View reviews for business A

  Can see R01
  Cannot see R01
ACID properties

- **Atomicity**
  - Individual operations $P_B$, $P_I$, $R_B$, $D_I$, are atomic

- **Consistency**
  - Sync-full and sync-insert: causal consistent
    - If any of $P_I$, $R_B$, $D_I$ fails, causal consistent $\rightarrow$ eventual consistent by AUQ

- **Isolation**: read committed

- **Durability**
  - Guaranteed by WAL and AUQ failure handling protocol
Failure recovery for Asynchronous Update Queue (AUQ)

- Index update fails in sync- or async- schemes
  - Append failed operation to AUQ: casual consistent → eventual consistent

- What if AUQ fails, e.g., during a server failure?
  - Option 1: add a log to AUQ
  - **Option 2**: leverage the Write-ahead-Log (WAL) of base table

HBase

P_i: insert index
R_B: read base
D_I: delete index
Failure recovery for Asynchronous Update Queue (AUQ)

- Index update fails in sync- or async- schemes
  - Append failed operation to AUQ: casual consistent → eventual consistent

- What if AUQ fails, e.g., during a server failure?
  - Option 1: add a log to AUQ
  - Option 2: leverage the Write-ahead-Log (WAL) of base table

**Diagram:**

- **HBase**
  - **MemTable**
    - **WAL**
      - 3.2 roll-forward
  - **HTable**
    - 3.1 flush

- **Diff-Index**
  - **AUQ**
    - 2.2
    - 4: \( P_I, R_B, D_I \)

**Note:**
- \( P_I \): insert index
- \( R_B \): read base
- \( D_I \): delete index
Failure recovery for Asynchronous Update Queue (AUQ)

- Index update fails in sync- or async- schemes
  - Append failed operation to AUQ: casual consistent → eventual consistent

- What if AUQ fails, e.g., during a server failure?
  - Option 1: add a log to AUQ
  - Option 2: leverage the Write-ahead-Log (WAL) of base table

HBase

- Diff-Index
  - 2.1 put 1
  - 2.2 AUQ X
  - 3.1 flush
  - 3.2 roll-forward
  - P_I: insert index
  - R_B: read base
  - D_I: delete index
Failure recovery for Asynchronous Update Queue (AUQ)

- Index update fails in sync- or async- schemes
  - Append failed operation to AUQ: casual consistent → eventual consistent

- What if AUQ fails, e.g., during a server failure?
  - Option 1: add a log to AUQ
  - **Option 2**: leverage the Write-ahead-Log (WAL) of base table

Durability is guaranteed if
- ✓ Enforce 3.0 before 3.1 flush
- ✓ Replay index update after WAL replay
- ✓ Index uses base’s timestamp

**Diagram**

- **HBase**
  - MemTable
    - 3.1 flush
    - 3.2 roll-forward
  - HTable

- **Diff-Index**
  - AUQ
    - 2.2
    - 4 P_I, R_B, D_I
  - Index tables

- **Put**
  - 1

- **ACID Schemes**
  - PI: insert index
  - RB: read base
  - DI: delete index
Diff-Index system: **global, server-managed index with configurable schemes**

**Function and performance testing**

**Client query API; index mgt**

**Client Library**
- Index Utility (create, destroy, bulk load, cleanse)
- Session cache
- getByIndex API

**BigSQL/BigInsights**
- DDL, Catalog, query engine …

**Regions**
- Coprocessors
  - AsyncObserver
  - SyncFullObserver
  - SyncInsertObserver

**Data Table**
- AUQ

**Index Table**

**Regions**

[In IBM InfoSphere BigInsights v2.1]
Performance of index update

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Data Update Only</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync-full = insert + read + del</td>
<td>1x</td>
<td>high &gt;5x</td>
</tr>
<tr>
<td>sync-insert = insert</td>
<td>2x</td>
<td>~2x</td>
</tr>
<tr>
<td>async = 0 (insert + read + del)</td>
<td>3x</td>
<td>low and grows with load</td>
</tr>
</tbody>
</table>
Performance of index read

You can trade read for update, or vice versa

Performance of index update

Fast

Slow due to double check

As fast as sync-full but inconsistent
Range query latency:
with different selectivity (i.e., how many records returned by it)

low latency

Double check can be costly
Consistency in async: index-after-data time lag

- Measure the distribution of the time-lag, under different transaction rate

Staleness of async index grows with tranx rate of the system

async = 0 (insert + read + del)
**Diff-Index schemes: experiments coherent with analysis**

<table>
<thead>
<tr>
<th>Scheme Feature</th>
<th>1 Sync-full</th>
<th>2 Sync-insert</th>
<th>3 Async-session</th>
<th>4 Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update latency</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Read latency</td>
<td>Low</td>
<td>High/Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Consistent read</td>
<td>Yes</td>
<td>Yes/No</td>
<td>No/Session</td>
<td>No</td>
</tr>
</tbody>
</table>

- Better consistency
- Better (update) performance
Related Work

- **HBase index**
  - Salesforce Phoenix: global index
    - Has only “sync-full” scheme
    - **Kill region server when index update fails (vs. drop to AUQ)**
  - Huawei: local index

- **Transactions in NoSQL**
  - Percolator: 2PC in BigTable
  - Spanner: 2PC + Paxos + GPS TrueTime
  - OMID: check R-W (instead of W-W) conflicts to ensure serializability

- **LSM**
  - bLSM: a better LSM compaction scheduler + BloomFilter
The basic idea behind Diff-Index: CAP Theorem

**CAP theorem**: achieve two out of three in a dist env

- **C**: consistency
- **A**: availability (latency)
- **P**: partition tolerance

Higher latency
Fewer inconsistencies

Lower latency
More inconsistencies

**Eventual**

**Diff-Index**: schemes to balance performance & consistency for LSM-Tree index, by tuning the equation:

\[ \text{index\_update} = \text{insert} + \text{read} + \text{del} \]
References for HBase

- **Papers**
  - Bigtable: A Distributed Storage System for Structured Data. OSDI 06
  - Large-scale Incremental Processing Using Distributed Transactions and Notifications. OSDI 2010
  - Megastore: Providing Scalable, Highly Available Storage for Interactive Services. CIDR 2011
  - Spanner: Google's Globally-Distributed Database. OSDI 2012
  - Omid: Lock-free transactional support for distributed data stores. ICDE 2014

- **Feature enhancements**
  - Index
    - InfoSphere BigInsights Diff-Index
    - Huawei: local secondary index
  - SQL layer
    - Intel Panthera
    - Salesforce Phoenix
  - SQL, transaction
    - Splice Machine

- **Conferences**
  - HBaseCon: [http://www.hbasecon.com/](http://www.hbasecon.com/)
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

HBase (table)

Dynamo (K/V)

MongoDB (json/doc)

Neo4j (graph)

Diff-Index

Analytics on NoSQL: Hadoop & Spark

Summary

Four features of NoSQL
Four categories of NoSQL
Four aspects to understand NoSQL
Dynamo: Overview

- Dynamo: Amazon’s Highly Available Key-Value Store (SOSP’ 07)

- Eventually consistent (a “CA” system from CAP’s perspective)

- Hash partitioned, auto-sharding, horizontally scalable

- Notable users
  - Amazon shopping cart, customer preferences, product catalog …
  - Influenced Cassandra, Riak, Voldemort (LinkedIn)
Dynamo: Overview

Data model (seen by end-users)

<table>
<thead>
<tr>
<th>Key 1</th>
<th>Blob 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key 2</td>
<td>Blob 2</td>
</tr>
<tr>
<td>Key 3</td>
<td>Blob 3</td>
</tr>
</tbody>
</table>

Blob: <1 MB. Shopping cart, session, preferences…

Partitioning scheme & metadata

Transaction semantics

- Single <key, blob> level atomicity
- Replication in the ring
- Availability: highly available
- Consistency: configurable
- Failover: hinted handoff

• Consistent hashing
• Fully distributed metadata (0-hop DHT)
- **Simple hashing with** \( n \) **machines: value based**
  - \( key \mod n \)
  - \( \text{hash}(key) \mod n \)

- **Problem: add or remove a machine**
  - \( (key \mod n) \neq (key \mod n+1) \neq (key \mod n-1) \)
  - All keys need to be moved!

- **Consistent hashing: region based**
  - Output of hash function forms a **ring** (e.g., largest hash value wraps to the smallest one)
  - Each node is assigned a value in this ring, and responsible for a **region** before it

- **Metadata**
  - P2P exchange, Gossip type of protocol

---

To how store meta-data

- **centralized:** HDFS
- **Partially dist:** BigTable, HBase, FDS,
- **Fully dist:** Cassandra, Dynamo
- **Replication**
  - Each item is replicated at $N$ hosts
  - One coordinator and $N-1$ clockwise successor nodes in the ring
  - When a node $D$ is not available, load to it can be handoff with hint to $E$
  - A key can have a preference list of nodes $> N$, choose $N$ healthy ones at run-time for high availability

- **Virtual nodes**
  - A physical node can be responsible for several virtual nodes in the ring
  - When a physical node becomes unavailable, its load is evenly distributed to many virtual nodes
  - When a physical node becomes available again, it accepts load from each of the many neighbor nodes
- Configurable consistency

- Dynamo writes to N replicas: quorum-like system
  - N: number of replicas
  - W: number of synchronous write acknowledgement
  - R: number of participate in a read
  - N < W + R: consistent
  - N ≥ W + R: not consistent

![Diagram with client a=2 and a=1 replicas]

N=3

Overview → Data model → Storage → Partition → ACID

Tutorial at IEEE ICWS, June 27, 2014, Alaska, USA
- **Configurable consistency**

- **Dynamo writes to N replicas: quorum-like system**
  - $N$: number of replicas
  - $W$: number of synchronous write acknowledgment
  - $R$: number of participate in a read
  - $N < W + R$: consistent
  - $N \geq W + R$: not consistent

```plaintext
W = 3
R = 1
```

![Diagram](image)

- Client sends a write request to 3 replicas.
- The replicas return acknowledgments.
- The client reads from the replicas.

- $N = 3$

---

**Tutorial at IEEE ICWS, June 27, 2014, Alaska, USA**
- **Configurable consistency**
- **Dynamo writes to N replicas: quorum-like system**
  - \( N \): number of replicas
  - \( W \): number of synchronous write acknowledgement
  - \( R \): number of participate in a read
  - \( N < W + R \): consistent
  - \( N \geq W + R \): not consistent

\[
\begin{align*}
W &= 3 \\
R &= 1
\end{align*}
\]

\[ N = 3 \]
- **Configurable consistency**

- **Dynamo writes to N replicas: quorum-like system**
  - $N$: number of replicas
  - $W$: number of synchronous write acknowledgement
  - $R$: number of participate in a read
  - $N < W + R$: consistent
  - $N \geq W + R$: not consistent

$$W = 3 \quad R = 1$$

![Diagram showing write and read operations with N=3 and a=2 for each replica](image-url)
- Configurable consistency

- Dynamo writes to N replicas: quorum-like system
  - \( N \): number of replicas
  - \( W \): number of synchronous write acknowledgement
  - \( R \): number of participate in a read
  - \( N < W + R \): consistent
  - \( N \geq W + R \): not consistent

\[
\begin{align*}
W &= 3 \\
R &= 1
\end{align*}
\]

Read from any node gives consistent result

\( N = 3 \)
- Configurable consistency

- Dynamo writes to N replicas: quorum-like system
  - N: number of replicas
  - W: number of nodes participate in write
  - R: number of participate in read
  - N< W + R: consistent
  - N ≥ W + R: not consistent

\[
W = 2 \\
R = 1
\]

\[
\text{client} \quad a = 2
\]

N=3

Overview → Data model → Storage → Partition → ACID

Tutorial at IEEE ICWS, June 27, 2014, Alaska, USA
- Configurable consistency

- Dynamo writes to N replicas: quorum-like system
  - N: number of replicas
  - W: number of nodes participate in write
  - R: number of participate in read
  - N < W + R: consistent
  - N ≥ W + R: not consistent

\[ W = 2 \]
\[ R = 1 \]

N = 3

Overview → Data model → Storage → Partition → ACID
- Configurable consistency
- Dynamo writes to N replicas: quorum-like system
  - $N$: number of replicas
  - $W$: number of nodes participate in write
  - $R$: number of participate in read
  - $N < W + R$: consistent
  - $N \geq W + R$: not consistent

W=2
R=1

N=3

Client

write
read

a=2

a=2

a=1
- Configurable consistency

- Dynamo writes to N replicas: quorum-like system
  - \( N \): number of replicas
  - \( W \): number of nodes participate in write
  - \( R \): number of participate in read
  - \( N < W + R \): consistent
  - \( N \geq W + R \): not consistent

\[
\begin{align*}
W &= 2 \\
R &= 1 \\
\end{align*}
\]

Read is inconsistent
- Configurable consistency

- **Dynamo writes to N replicas: quorum-like system**
  - N: number of replicas
  - W: number of nodes participate in write
  - R: number of participate in read
  - N < W + R: consistent
  - N >= W + R: not consistent

  
  
  \[
  \begin{align*}
  W &= 2 \\
  R &= 2
  \end{align*}
  \]

  \[N = 3\]
- Configurable consistency

- Dynamo writes to $N$ replicas: quorum-like system
  - $N$: number of replicas
  - $W$: number of nodes participate in write
  - $R$: number of participate in read
  - $N < W + R$: consistent
  - $N \geq W + R$: not consistent

$$\begin{align*}
W &= 2 \\
R &= 2
\end{align*}$$

$N = 3$

Diagram:
- Client
- $a = 2$
- $a = 1$
- $a = 1$

Write and read operations are shown from the client to the replicas.
- Configurable consistency

- Dynamo writes to N replicas: quorum-like system
  - \( N \): number of replicas
  - \( W \): number of nodes participate in write
  - \( R \): number of participate in read
  - \( N < W + R \): consistent
  - \( N > W + R \): not consistent

\[
\begin{align*}
W &= 2 \\
R &= 2
\end{align*}
\]

Storage

Data model

Partition

ACID

Overview

N=3

Client

\( a = 2 \)

write

read

\( a = 2 \)  \( a = 2 \)  \( a = 1 \)
- Configurable consistency

- Dynamo writes to N replicas: quorum-like system
  - \( N \): number of replicas
  - \( W \): number of nodes participate in write
  - \( R \): number of participate in read
  - \( N < W + R \): consistent
  - \( N >= W + R \): not consistent

\[
\begin{align*}
W &= 2 \\
R &= 2
\end{align*}
\]
- **Configurable consistency**

- **Dynamo writes to N replicas: quorum-like system**
  - $N$: number of replicas
  - $W$: number of nodes participate in write
  - $R$: number of participate in read
  - $N < W + R$: consistent
  - $N \geq W + R$: not consistent

![Diagram](image)
- Configurable consistency

- Dynamo writes to N replicas: quorum-like system
  - $N$: number of replicas
  - $W$: number of nodes participate in write
  - $R$: number of participate in read
  - $N < W + R$: consistent
  - $N \geq W + R$: not consistent

$W = 2$
$R = 2$

Read is consistent!
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

- HBase (table)
- Dynamo (K/V)
- MongoDB (json/doc)
- Neo4j (graph)

Diff-Index

Analytics on NoSQL: Hadoop & Spark

Summary

Four features of NoSQL
Four categories of NoSQL
Four aspects to understand NoSQL
MongoDB: Overview
Data model (seen by end-users)

{  
   _id: <ObjectId>,  
   username: "123xyz",  
   contact: {      
      phone: "123-456-7890",      
      email: "xyz@example.com"  
   },  
   access: {      
      level: 5,      
      group: "dev"  
   } 
}

- JSON: flexible schema
- Nested structure (vs. “flat” HBase, Dynamo)

Partitioning (sharding) scheme and metadata
- Range based
- Hash based
- Metadata stored in separate servers

Single node storage

Data: double linked BSON

Index: B-tree

Transaction semantics
- Single document atomicity
- Replication set
from: http://horicky.blogspot.com/2012/04/mongodb-architecture.html
**MongoDB replica set**
- The **primary** accepts all read/write from clients
- The **secondaries** replicate primary’s data asynchronously
- Replica set selects another member to become the new primary when primary fails
- Options for read/write quorum: $R + W ? N$

### Read Preference

<table>
<thead>
<tr>
<th>Read Preference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>Default. Read from primary.</td>
</tr>
<tr>
<td>primaryPreferred</td>
<td>Read from primary if possible; otherwise from secondaries.</td>
</tr>
<tr>
<td>secondary</td>
<td>Read from the secondaries.</td>
</tr>
<tr>
<td>secondaryPreferred</td>
<td>Read from secondaries if possible; otherwise from primary.</td>
</tr>
<tr>
<td>nearest</td>
<td>Read from node with the least network latency.</td>
</tr>
</tbody>
</table>

### Write Preference

<table>
<thead>
<tr>
<th>Write Preference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ack. from primary.</td>
</tr>
<tr>
<td>majority</td>
<td>Ack. from majority nodes.</td>
</tr>
<tr>
<td>n</td>
<td>Ack. from $n$ nodes.</td>
</tr>
</tbody>
</table>
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

HBase (table) 

Dynamo (K/V) 

MongoDB (json/doc) 

Neo4j (graph) 

Diff-Index 

Analytics on NoSQL: Hadoop & Spark 

Summary

Four features of NoSQL
Four categories of NoSQL
Four aspects to understand NoSQL
Neo4j: Overview

Data model (seen by end-users)

- Property graph

Partitioning (sharding) scheme and metadata
- No sharding
- HA optional

Single node storage

- Index free; pointer chasing

Transaction semantics
- Full ACID support
Why graph?

social network

PageRank

transportation

recommendation

• Collaborative filtering: people bought this also bought …
Neo4j Property Graph Model

- **Queries**
  - Nodes
  - Relations
  - Properties
  - Paths
  - Is there a friendship relation between two persons?
  - A person’s two-degree friends?
Neo4j Property Graph Model

- Queries
  - Nodes
  - Relations
  - Properties
  - Paths
    - Is there a friendship relation between two persons?
    - A person’s two-degree friends?
Relational

Find Alice’s friend

<table>
<thead>
<tr>
<th>ID</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>99</td>
<td>Zach</td>
</tr>
</tbody>
</table>

Person

<table>
<thead>
<tr>
<th>PersonFriend</th>
</tr>
</thead>
<tbody>
<tr>
<td>PersonID</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>99</td>
</tr>
</tbody>
</table>

Neo4j

(native format)

Node

Relationship (33 bytes)
Relational

Find Alice’s friend

Index lookup $O(\log n)$

Neo4j (native format)

Relationship (33 bytes)

Figure from: Ian Robinson, Jim Webber. Graph Databases. O'Reilly, 2013
### Relational

Find Alice’s friend

Index lookup $O(\log n)$

### Neo4j (native format)

Store pointers to relations: $O(1)$

Fix length record: use the pointer to calculate offset in the file

---

Figure from: Ian Robinson, Jim Webber. Graph Databases. O'Reilly, 2013
Comparison between relational and graph storage

Relational

<table>
<thead>
<tr>
<th>Person</th>
<th>PersonFriend</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>PersonID</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>99</td>
<td>99</td>
</tr>
</tbody>
</table>

- Entity and relations
- Use **index** to locate record

**Pros**
- Good at entity-based queries
- Good at range queries

Neo4j native storage

- Nodes with **pointers** to relations & properties

**Pros**
- Schema-less
- Good at relation-based queries
  - Find a path
  - Find neighbor nodes
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

- HBase (table)
- Dynamo (K/V)
- MongoDB (json/doc)
- Neo4j (graph)

Analytics on NoSQL: Hadoop & Spark

Summary
Analytics on NoSQL: Hadoop and Spark

- NoSQL stores do a good job to store and serve the data
- What about analytics?
  - Complex and ad hoc queries
  - Batch jobs: ETL, machine learning, etc
- Hadoop: the *de facto* standard
  - Initial version shortly after Google’s MapReduce paper in OSDI 2004
  - 2009, TeraSort on 1,400 nodes
  - 2013, Tencent (China) reports a 4000-node cluster
- Hadoop has interfaces to many NoSQL stores
  - HBase, Cassandra, MongoDB, S3 …
- Hadoop consists of
  - **Distributed File System (HDFS)**: distributed file system
  - **MapReduce**: parallel processing
  - **YARN**: cluster resource management and job scheduling
Hadoop Distributed File System

- Each file is split into **blocks** (e.g., 64MB), each block is replicated (e.g., 3 times)
- **Namenode** manages fs, maintains block → datanode mapping
- **Datanodes** serve data read/write

Figure from [http://hadoop.apache.org/docs/current2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html](http://hadoop.apache.org/docs/current2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html)
Dataflow of Hadoop MapReduce

- Map reads input split from HDFS
- Map outputs to memory, then spill to local disk
- Map outputs are shuffled, copied and merged to individual reduce tasks
- Reduce outputs to HDFS

Figure from “Tom White, Hadoop the definitive guide, 2nd edition, O'Reilly.”
Resource management and job scheduling

Hadoop 1

- **JobTracker**
  - Manage resources (worker nodes)
  - Tracking resource consumption
  - Manage job life-cycle

- **TaskTracker**
  - launch/teardown tasks
  - report task-status

- **Issues**
  - JobTrack too heavy duty – lack of scalability
  - JobTrack can schedule only MapReduce -- lack of versatility
Resource management and job scheduling

Hadoop 2 YARN

- Issues of Hadoop 1 scheduler
  - JobTrack too heavy duty – lack of scalability
  - JobTrack can schedule only MapReduce -- lack of versatility

- YARN
  - Separates resource management from job scheduling
  - **Resource Manager** (RM) allocate resources to applications, and launch **Application Master** (AM)
    - AM obtains resources, manages application logic and track status

- Different AMs can co-exist in one YARN cluster
  - MapReduce
  - MPI
  - Spark
  - …
The Overhaul of Hadoop MapReduce

- **Shortages of MapReduce**
  - Rigid Map→Reduce→Map→Reduce …, only two operators
  - Reduce output goes to HDFS -- slow
  - Cannot handle DAG jobs, pipeline, etc

- **What has changed since MapReduce invented?**
  - **Hardware**
    - “each machine had two … processors, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet …” (MapReduce paper, OSDI’ 04)
    - now: 32 cores, 512 GB RAM, SSDs, 10 gb Ethernet
  - **Application**
    - More diversified: stream, ad hoc query, batch …
    - More complex: flows instead of jobs, iterative processing
  - **Functional programming gets a reviving interest**
    - Lambda expression, rich operators on collection, side-effect free … make it suitable for parallel programming
    - Lambda expression, map, reduce, … are added into Python, Java 8

**Need a fast, functional style, easy-to-code & versatile “Hadoop”!**
Two Major MapReduce “Overhaul” projects

- Two major projects
  - Stratosphere from TU Berlin
  - Spark from UC Berkeley

- Common features
  - More operators: map, reduce, CoGroup, union ….
  - Start working in memory and gracefully go out-of-core
  - Support dataflow and iterative processing
  - Functional programming interface in Scala
  - Open source, and actively building an ecosystem (SQL, graph ….)

- We use Spark as an example

Dataflow in Hadoop

Input → HDFS read → iter. 1 → HDFS write → HDFS read → iter. 2 → HDFS write → ...
Dataflow in Hadoop

Input -> HDFS read -> iter. 1 -> HDFS write -> iter. 2 -> HDFS read -> iter. 3 -> HDFS write

Input -> HDFS read

query 1 -> result 1
query 2 -> result 2
query 3 -> result 3

Slow due to replication and disk I/O, but necessary for fault tolerance
In-Memory Data Sharing in Spark

Input

iter. 1

iter. 2

... 

one-time processing

query 1

query 2

query 3

... 

10-100× faster than network/disk, but how to get FT?
RDD Recovery: through lineage, not replication

Input

one-time processing

Input

query 1

query 2

query 3
RDD Recovery: through lineage, not replication

Input

iter. 1

iter. 2

...
RDD Recovery: through lineage, not replication
RDD Recovery: through lineage, not replication

Input

iter. 1

iter. 2

...
RDD Recovery: through lineage, not replication

Input → iter. 1 → iter. 2 → ... → one-time processing

Input → iter. 1 → query 1 → ... → query 2 → query 3 → ...
RDD Recovery: through lineage, not replication

Input

iter. 1

iter. 2

one-time processing

Input

query 1

query 2

query 3

. . .
RDD Recovery: through lineage, not replication

Input

iter. 1

iter. 2

. . .

one-time processing

Input

query 1

query 2

query 3

. . .
Spark: Programming and Run-time

WordCount in Spark (Scala)

```scala
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

In Hadoop (Java)

```java
public class WordCount {

    public static class Map extends Writable implements WritableValue, WritableComparable {
        private final static IntWritable one = new IntWritable();
        private Text word = new Text();
        public void map(Writable key, Text value, Context context) throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class Reduce extends Reducer {
        public void reduce(Text key, Iterator<IntWritable> values, Context context)
        throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        TextInputFormat.addInputPath(job, new Path(args[0]));
        TextOutputFormat.setOutputPath(job, new Path(args[1]));
        job.waitForCompletion(true);
    }
}
```
Spark: Programming and Run-time

WordCount in Spark (Scala)

```scala
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Spark run-time
# Spark Operators

## Spark operators (transformations and actions)

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T ⇒ U)</code></td>
<td>RDD[T] ⇒ RDD[U]</td>
<td></td>
</tr>
<tr>
<td><code>filter(f : T ⇒ Bool)</code></td>
<td>RDD[T] ⇒ RDD[T]</td>
<td></td>
</tr>
<tr>
<td><code>flatMap(f : T ⇒ Seq[U])</code></td>
<td>RDD[T] ⇒ RDD[U]</td>
<td></td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td>RDD[T] ⇒ RDD[T] (Deterministic sampling)</td>
<td></td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</td>
<td></td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) ⇒ V)</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
<td></td>
</tr>
<tr>
<td><code>union()</code></td>
<td>(RDD[T], RDD[T]) ⇒ RDD[T]</td>
<td></td>
</tr>
<tr>
<td><code>join()</code></td>
<td>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]</td>
<td></td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]</td>
<td></td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td>(RDD[T], RDD[U]) ⇒ RDD[(T, U)]</td>
<td></td>
</tr>
<tr>
<td><code>mapValues(f : V ⇒ W)</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)</td>
<td></td>
</tr>
<tr>
<td><code>sort(c : Comparator[K])</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
<td></td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[K])</code></td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>RDD[T] ⇒ Long</td>
<td></td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>RDD[T] ⇒ Seq[T]</td>
<td></td>
</tr>
<tr>
<td><code>reduce(f : (T, T) ⇒ T)</code></td>
<td>RDD[T] ⇒ T</td>
<td></td>
</tr>
<tr>
<td><code>lookup(k : K)</code></td>
<td>RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)</td>
<td></td>
</tr>
<tr>
<td><code>save(path : String)</code></td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
<td></td>
</tr>
</tbody>
</table>

Performance vs. Hadoop

Latency

Scalability

In case of failure

Speedup comes from

- Scheduling delay
- IO from/to HDFS
- De-serialization cost

BDAS, the Berkeley Data Analytics Stack
(https://amplab.cs.berkeley.edu/software/)
Agenda

Background: why NoSQL

Overview: “3 Fours” of NoSQL

- HBase (table)
- Dynamo (K/V)
- MongoDB (json/doc)
- Neo4j (graph)

Analytics on NoSQL: Hadoop & Spark

Summary

Four features of NoSQL
Four categories of NoSQL
Four aspects to understand NoSQL
Four Features of NoSQL

Flexible schema

Simple API

Relaxed ACID

Scale-out on commodity HW
Four categories of NoSQL

**Tabular stores**
- Google Bigtable
- HBase
- Cassandra
- Accumulo
- Hypertable

**Key/Value stores**
- Dynamo
- S3
- Dropbox

**Document stores**
- MongoDB
- CouchDB
- Lotus Notes

**Graph stores**
- Neo4j
- TITAN
Four aspects to understand NoSQL

Data model (seen by end-users)
- table
- doc
- graph
- hashmap

Partitioning (sharding) scheme and metadata
- Range Partitioning
- Hash Partitioning

Single node storage
- Sequence file
- B+ Tree
- Linked list
- LSM tree

Transaction semantics
- Atomicity in what granularity
- Consistency level: strong, causal, session, eventual?
- Concurrency: locking, multi-version?
- Replication
- Availability
- Failover
- …
Summary

- **Take away message: “three fours” (3x4)**
  - Four features of NoSQL
  - Four categories of NoSQL
  - Four aspects to understand NoSQL

- **NoSQL**
  - Incorporates a lot of recent (and less recent) advances in distributed systems, data management
    - CAP theorem
    - Multi-version concurrency control
    - Distributed hash table (DHT)
    - ...
  - Not to replace RDBMS, but at least an important complement
Thank You!