Lecture 8:
NoSQL Databases
Outline

- Introduction
- Transaction Consistency
- Column-Family Stores
- Key-Value Stores
  - Example: Redis
- Document Stores
  - Example: MongoDB
- Graph Databases
  - Example: neo4j
- Concluding Remarks
SQL Means More than SQL

• SQL stands for the query language
• But commonly refers to the traditional RDBMS:
  – Relational storage of data
    • Each tuple is stored consecutively
  – Joins as first-class citizens
    • In fact, normal forms prefer joins to maintenance
  – Strong guarantees on transaction management
    • No consistency worries when many transactions operate simultaneously on common data
• Focus on scaling up
  – That is, make a single machine do more, faster
Trends Drive Common Requirements

Social media + mobile computing

• Explosion in data, always available, constantly read and updated
• High load of simple requests of a common nature
• Some consistency can be compromised (e.g., 👍)

Cloud computing + open source

• Affordable resources for management / analysis of data
• People of various skills / budgets need software solutions for distributed analysis of massive data

Database solutions need to **scale out** (utilize distribution, “scale horizontally”)

Compromises Required

What is needed for effective distributed, data- and user-intensive applications?

1. Use data models and storage that allow to avoid joins of big objects
2. Relax the guarantees on consistency
NoSQL

• Not Only SQL
  – Not the other thing!
  – Term introduced by Carlo Strozzi in 1998 to describe an alternative database model
  – Became the name of a movement following Eric Evans’s reuse for a distributed-database event

• Seminal papers:
  – Google’s BigTable
  – Amazon’s DynamoDB
    • DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, Vogels: Dynamo: amazon's highly available key-value store. SOSP 2007: 205-220
NoSQL from nosql-database.org

“

• Next Generation Databases mostly addressing some of the points: being *non-relational*, *distributed*, *open-source* and *horizontally scalable*.

• The original intention has been modern web-scale databases. The movement began early 2009 and is growing rapidly. Often more characteristics apply such as: *schema-free*, *easy replication support*, *simple API*, *eventually consistent / BASE* (not ACID), a huge amount of data and more.

• So the misleading term “nosql” (the community now translates it mostly with “not only sql”) should be seen as an alias to something like the definition above.

”
Common NoSQL Features

• Non-relational data models
• Flexible structure
  – No need to fix a schema, attributes can be added and replaced on the fly
• Massive read/write performance; availability via horizontal scaling
  – Replication and sharding (data partitioning)
  – Potentially thousands of machines worldwide
• Open source (very often)
• APIs to impose locality
Database Replication

- Data replication: storing the same data on several machines ("nodes")
- Useful for:
  - **Availability** (parallel requests are made against replicas)
  - **Reliability** (data can survive hardware faults)
  - **Fault tolerance** (system stays alive when nodes/network fail)
- Typical architecture: master-slave

Replication example in MySQL (dev.mysql.com)
Database Sharding

• Simply partitioning data across multiple nodes
• Useful for
  – Scaling (more data)
  – Availability

Replication + sharding example in MongoDB
(mongodb-documentation.readthedocs.org)
Open Source

• Free software, source provided
  – Users have the right to use, modify and distribute the software
  – But restrictions may still apply, e.g., adaptations need to be opensource

• Idea: community development
  – Developers fix bugs, add features, ...

• How can that work?

• A major driver of opensource is Apache
Apache Software Foundation

• Non-profit organization
• Hosts communities of developers
  – Individuals and small/large companies
• Produces open-source software
• Funding from grants and contributions
• Hosts very significant projects
  – Apache Web Server, Hadoop, Zookeeper, Cassandra, Lucene, OpenOffice, Struts, Tomcat, Subversion, Tcl, UIMA, ...
We Will Look at 4 Data Models

- Key/Value Store
- Column-Family Store
- Document Store
- Graph Databases
Highlighted Database Features

• **Data model**
  – What data is being stored?

• **CRUD interface**
  – API for **Create**, **Read**, **Update**, **Delete**
  – Sometimes preceding S for Search

• **Transaction consistency guarantees**

• **Replication and sharding model**
  – What’s automated and what’s manual?
True and False Conceptions

• True:
  – SQL does not effectively handle common Web needs of massive (datacenter) data
  – SQL has guarantees that can sometimes be compromised for the sake of scaling
  – Joins are not for free, sometimes undoable

• False:
  – NoSQL says NO to SQL
  – Nowadays NoSQL is the only way to go
  – Joins can always be avoided by structure redesign
Outline

• Introduction
• Transaction Consistency
• Column-Family Stores
• Key-Value Stores
  ▪ Example: Redis
• Document Stores
  ▪ Example: MongoDB
• Graph Databases
  ▪ Example: neo4j
• Concluding Remarks
Transaction

• A sequence of operations (over data) viewed as a single higher-level operation
  – Transfer money from account 1 to account 2

• DBMSs execute transactions in parallel
  – No problem applying two “disjoint” transactions
  – But what if there are dependencies?

• Transactions can either commit (succeed) or abort (fail)
  – Failure due to violation of program logic, network failures, credit-card rejection, etc.

• DBMS should not expect transactions to succeed
Examples of Transactions

• Airline ticketing
  – Verify that the seat is vacant, with the price quoted, then charge credit card, then reserve

• Online purchasing
  – Similar

• “Transactional file systems” (MS NTFS)
  – Moving a file from one directory to another: verify file exists, copy, delete

• Textbook example: bank money transfer
  – Read from acct#1, verify funds, update acct#1, update acct#2
Transfer Example

### txn1
- Begin
- Read(A,v)
- \( v = v - 100 \)
- Write(A,v)
- Read(B,w)
- \( w = w + 100 \)
- Write(B,w)
- Commit

### txn2
- Begin
- Read(A,v)
- \( v = v - 100 \)
- Write(A,v)
- Read(B,w)
- \( w = w + 100 \)
- Write(B,w)
- Commit

### Scheduling

- **Scheduling** is the operation of interleaving transactions
  - *Why is it good?*

- A *serial scheduling* executes transactions one at a time, from beginning to end

- A *good* ("serializable") scheduling is one that *behaves like some serial scheduling* (typically by locking protocols)
Scheduling Example 1 (good)

txn₁

Begin
Read(A, v)
v = v-100
Write(A, v)
w=w+100
Write(B, w)
Commit

txn₂

Begin
Read(A, x)
x = x-100
Write(A, x)

Read(A, v)
v = v-100
Write(A, v)

Read(B, w)
w=w+100
Write(B, w)

Read(C, y)
y=y+100
Write(C, y)

Commit
Scheduling Example 2 (bad)

txn₁

Begin
Read(A,v)
v = v-100
Write(A,v)
w = w+100
Write(B,w)
Commit

txn₂

Begin
Read(A,x)
x = x-100
Write(A,x)
Read(C,y)
y = y+100
Write(C,y)
Commit
ACID

• **Atomicity**
  – Either all operations applied or none are (hence, we need not worry about the effect of incomplete / failed transactions)

• **Consistency**
  – Each transaction can start with a consistent database and is required to leave the database consistent

• **Isolation**
  – The effect of a transaction should be as if it is the only transaction in execution (in particular, changes made by other transactions are not visible until committed)

• **Durability**
  – Once the system informs a transaction success, the effect should hold without regret, even if the database crashes (before making all changes to disk)
ACID May Be Overly Expensive

• In quite a few modern applications:
  – ACID contrasts with key desiderata: high volume, high availability
  – We can live with some errors, to some extent
  – Or more accurately, we prefer to suffer errors than to be significantly less functional

• Can this point be made more “formal”? 

Simple Model of a Distributed Service

- **Context:** distributed service
  - e.g., social network

- **Clients make get / set requests**
  - e.g., setLike(user, post), getLikes(post)
  - Each client can talk to any server

- **Servers return responses**
  - e.g., ack, \{user_1, ..., user_k\}

- **Failure:** the network may occasionally disconnect due to failures (e.g., switch down)

- **Desiderata:** **Consistency, Availability, Partition tolerance**
CAP Service Properties

- **Consistency**: every read (to any node) gets a response that reflects the most recent version of the data
  - More accurately, a transaction should behave as if it changes the entire state correctly in an instant
  - Idea similar to serializability

- **Availability**: every request (to a living node) gets an answer: set succeeds, get returns a value

- **Partition tolerance**: service continues to function on network failures
  - As long as clients can reach servers
Simple Illustration

Consistency, Availability

Consistency, Partition tolerance

Availability, Partition tolerance
The CAP Theorem

Eric Brewer’s CAP Theorem:

*A distributed service can support at most two out of C, A and P*
• Brewer presented it as the **CAP principle** in a 1999 article
  – Then as an informal conjecture in his keynote at the PODC 2000 conference

• In 2002 a formal proof was given by Gilbert and Lynch, making CAP a **theorem**
  – It is mainly about making the statement formal; the proof is straightforward
Visual Guide to NoSQL Systems

Availability: Each client can always read and write.

Data Models
- Relational (comparison)
- Key-Value
- Column-Oriented/Tabular
- Document-Oriented

Pick Two

CA
- RDBMSs (MySQL, Postgres, etc)
- Aster Data
- Greenplum
- Vertica

AP
- Dynamo
- Voldemort
- Tokyo Cabinet
- KAI
- Cassandra
- SimpleDB
- CouchDB
- Riak

Consistency: All clients always have the same view of the data.

CP
- BigTable
- Hbase
- MongoDB
- Terrastore
- Scalaris
- Berkeley DB
- MemcacheDB
- Redis

Partition Tolerance: The system works well despite physical network partitions.

2010 visual by Nathan Hurst
http://blog.nahurst.com/visual-guide-to-nosql-systems
The BASE Model

- Applies to distributed systems of type AP
- **Basic Availability**
  - Provide high availability through distribution
- **Soft state**
  - Inconsistency (stale answers) allowed
- **Eventual consistency**
  - If updates stop, then after some time consistency will be achieved
    - Achieved by protocols to propagate updates and verify correctness of propagation (gossip protocols)
- Philosophy: best effort, optimistic, staleness and approximation allowed
More in Relevant CS Courses

- **236351**
  - Distributed Systems

- **234322**
  - Information Storage Systems
Outline

• Introduction
• Transaction Consistency
• Column-Family Stores
• Key-Value Stores
  ▪ Example: Redis
• Document Stores
  ▪ Example: MongoDB
• Graph Databases
  ▪ Example: neo4j
• Concluding Remarks
# 2 Types of Column Store

## Standard RDB

<table>
<thead>
<tr>
<th>id</th>
<th>sid</th>
<th>name</th>
<th>address</th>
<th>year</th>
<th>faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>861</td>
<td>Alma</td>
<td>Haifa</td>
<td>2</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>753</td>
<td>Amir</td>
<td>Jaffa</td>
<td>NULL</td>
<td>CS</td>
</tr>
<tr>
<td>3</td>
<td>955</td>
<td>Ahuva</td>
<td>NULL</td>
<td>2</td>
<td>IE</td>
</tr>
</tbody>
</table>

## Column-Family Store: NoSQL

**Column Store**: each column stored separately (still SQL)

*Why? Efficiency (fetch only required columns), compression, sparse data for free*

**Keyspace**

<table>
<thead>
<tr>
<th>column family</th>
<th>id</th>
<th>sid</th>
<th>name</th>
<th>address</th>
<th>year</th>
<th>faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>861</td>
<td>Alma</td>
<td>Haifa</td>
<td>2</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>753</td>
<td>Amir</td>
<td>Jaffa</td>
<td>NULL</td>
<td>CS</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>955</td>
<td>Ahuva</td>
<td>NULL</td>
<td>2</td>
<td>IE</td>
</tr>
</tbody>
</table>

**Column-Family Store**: Cassandra model

- **Timestamp for conflicts**
- **“Column”**
- **“Supercolumn”**

<table>
<thead>
<tr>
<th>column family</th>
<th>id</th>
<th>faculty</th>
<th>email</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>CS</td>
<td>prime:c@d ext:c@e</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>CS</td>
<td>email:{prime:a@b ext:a@c}</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>IE</td>
<td>email:{prime:a@b ext:a@c}</td>
</tr>
</tbody>
</table>
Column Stores

• The two often mixed as “column store” → confusion
  – See Daniel Abadi’s blog

• Common idea: don’t keep a row in a consecutive block, split via projection
  – Column store: each column is independent; column-family store: each column family is independent

• Both provide some major efficiency benefits in common read-mainly workloads
  – Given a query, load to memory only the relevant columns
  – Columns can often be highly compressed due to value similarity
  – Effective form for sparse information (no NULLs, no space)

• Column-family store is handled differently from RDBs, often requiring a designated query language
Examples Systems

• Column store (SQL):
  – MonetDB (started 2002, Univ. Amsterdam)
  – VectorWise (spawned from MonetDB)
  – Vertica (M. Stonebraker)
  – SAP Sybase IQ
  – Infobright

• Column-family store (NOSQL):
  – Google’s BigTable (main inspiration to column families)
  – Apache HBase (used by Facebook, LinkedIn, Netflix...)
  – Hypertable
  – Apache Cassandra
Example: Apache Cassandra

- Initially developed by Facebook
  - Open-sourced in 2008
- Used by 1500+ businesses, e.g., Comcast, eBay, GitHub, Hulu, Instagram, Netflix, Best Buy, ...
- Column-family store
  - Supports key-value interface
  - Provides a SQL-like CRUD interface: CQL
- Uses Bloom filters
  - An interesting membership test that can have false positives but never false negatives, well behaves statistically
- BASE consistency model (AP)
  - Gossip protocol (constant communication) to establish consistency
  - Ring-based replication model
Cassandra’s Ring Model

Replication Factor = 3

write($k, t$)

hash($k$) = 2

Advantage: Flexibility / ease of cluster redesign
Outline

- Introduction
- Transaction Consistency
- Column-Family Stores
- Key-Value Stores
  - Example: Redis
- Document Stores
  - Example: MongoDB
- Graph Databases
  - Example: neo4j
- Concluding Remarks
Key-Value Stores

• Essentially, big distributed hash maps
• Origin attributed to Dynamo – Amazon’s DB for world-scale catalog/cart collections
  – But Berkeley DB has been here for >20 years
• Store pairs \(\langle \text{key}, \text{opaque-value} \rangle\)
  – Opaque means that DB does not associate any structure/semantics with the value; *oblivious* to values
  – This may mean more work for the user: retrieving a large value and parsing to extract an item of interest
• Sharding via partitioning of the key space
  – Hashing, gossip and remapping protocols for load balancing and fault tolerance
Example Databases

- **Amazon’s DynamoDB**
  - Originally designed for Amazon’s workload at peaks
  - Offered as part of Amazon’s Web services

- **Redis**
  - Next slides

- **Riak**
  - Focuses on high availability, BASE
  - “As long as your Riak client can reach one Riak server, it should be able to write data.”

- **FoundationDB**
  - Focus on transactions, ACID

- **Berkeley DB (and Oracle NoSQL Database)**
  - First release 1994, by Berkeley, acquired by Oracle
  - ACID, replication
Outline

• Introduction
• Transaction Consistency
• Column-Family Stores
• Key-Value Stores
  ▪ Example: Redis
• Document Stores
  ▪ Example: MongoDB
• Graph Databases
  ▪ Example: neo4j
• Concluding Remarks
Redis

- Basically a data structure for strings, numbers, hashes, lists, sets

- Simplistic “transaction” management
  - Queuing of commands as blocks, really
  - Among ACID, only Isolation guaranteed
    - A block of commands that is executed sequentially; no transaction interleaving; no roll back on errors

- In-memory store
  - Persistence by periodical saves to disk

- Comes with
  - A command-line API
  - Clients for different programming languages
    - Perl, PHP, Rubi, Tcl, C, C++, C#, Java, R, ...
Example of Redis Commands

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>10</td>
</tr>
<tr>
<td>hset</td>
<td>5</td>
</tr>
<tr>
<td>hset</td>
<td>2</td>
</tr>
<tr>
<td>hset</td>
<td>Alma</td>
</tr>
<tr>
<td>sadd</td>
<td>20</td>
</tr>
<tr>
<td>sadd</td>
<td>Alma</td>
</tr>
<tr>
<td>rpush</td>
<td>a, b</td>
</tr>
<tr>
<td>lrange</td>
<td>a, b</td>
</tr>
<tr>
<td>lindex</td>
<td>b</td>
</tr>
<tr>
<td>lpop</td>
<td>c</td>
</tr>
<tr>
<td>rpop</td>
<td>b</td>
</tr>
</tbody>
</table>
Example of Redis Commands

- **Set**: `set x 10`
  - key: `x`
  - value: `10`

- **Hash Set**: `hset h y 5`
  - key: `h`
  - value: `{y: 5}`

- **Hash Set with Multiple Values**: `hset h1 name two`
  - key: `h1`
  - values: `{name: two}`

- **Hash Set with Multiple Values**: `hset h1 value 2`
  - key: `h1`
  - values: `{value: 2}`

- **Hash Set with Multiple Values**: `hmset p:22 name Alma age 25`
  - key: `p:22`
  - values: `{name: Alma, age: 25}`

- **Set Addition**: `sadd s 20`
  - key: `s`
  - values: `{20}`

- **Set Addition with Multiple Values**: `sadd s Alma`
  - key: `s`
  - values: `{Alma}`

- **List Push**: `rpush l a`
  - key: `l`
  - values: `([a])`

- **List Push**: `rpush l b`
  - key: `l`
  - values: `([a, b])`

- **List Push**: `lpush l c`
  - key: `l`
  - values: `([c, a, b])`

- **Get Simple Value**: `get x` => `10`
- **Get Hash Value**: `hget h y` => `5`
- **Get Hash Keys**: `hkeys p:22` => `{name, age}`
- **Get Hash Members**: `smembers s` => `{20, Alma}`
- **Get Hash Card**: `scard s` => `2`
- **List Length**: `llen l` => `3`
- **List Range**: `lrange l 1 2` => `[a, b]`
- **List Index**: `lindex l 2` => `b`
- **List Pop**: `lpop l` => `c`
- **List Pop**: `rpop l` => `b`
• A key can be any <256MB binary string
  – For example, JPEG image

• Some key operations:
  – List all keys: `keys *`
  – Remove all keys: `flushall`
  – Check if a key exists: `exists k`

• You can configure the persistency model
  – `save m k` means save every `m` seconds if at least `k` keys have changed
Redis Cluster

• Add-on module for managing multi-node applications over Redis
• Master-slave architecture for sharding + replication
  – Multiple masters holding pairwise disjoint sets of keys, every master has a set of slaves for replication and sharding

Outline

• Introduction
• Transaction Consistency
• Column-Family Stores
• Key-Value Stores
  ▪ Example: Redis
• Document Stores
  ▪ Example: MongoDB
• Graph Databases
  ▪ Example: neo4j
• Concluding Remarks
Document Stores

• Similar in nature to key-value store, but value is **tree structured** as a *document*

• Motivation: *avoid joins*; ideally, all relevant joins already encapsulated in the document structure

• A document is an atomic object that cannot be split across servers
  – But a document *collection* will be split

• Moreover, transaction atomicity is typically guaranteed within a single document

• Model generalizes column-family and key-value stores
Example Databases

- **MongoDB**
  - Next slides

- **Apache CouchDB**
  - Emphasizes Web access

- **RethinkDB**
  - Optimized for highly dynamic application data

- **RavenDB**
  - Deigned for .NET, ACID

- **Clusterpoint Server**
  - XML and JSON, a combined SQL/JavaScript QL
Outline

• Introduction
• Transaction Consistency
• Column-Family Stores
• Key-Value Stores
  ▪ Example: Redis
• Document Stores
  ▪ Example: MongoDB
• Graph Databases
  ▪ Example: neo4j
• Concluding Remarks
• Open source, 1\textsuperscript{st} release 2009, document store
  – Actually, an extended format called BSON (binary JSON) for typing and better compression
• Supports replication (master/slave), sharding
  – Developer provides the “shard key” – collection is partitioned by ranges of values of this key
• Consistency guarantees, CP of CAP
• Used by Adobe (experience tracking), Craigslist, eBay, FIFA (video game), LinkedIn, McAfee
• Provides connector to Hadoop
  – Cloudera provides the MongoDB connector in distributions
MongoDB Data Model

- JavaScript Object Notation (JSON) model
- **Database** = set of named **collections**
- **Collection** = sequence of **documents**
- **Document** = \{attribute_1:value_1,...,attribute_k:value_k\}
- **Attribute** = string (attribute_i≠attribute_j)
- **Value** = primitive value (string, number, date, ...), or a document, or an **array**
- **Array** = [value_1,...,value_n]

- Key properties: **hierarchical** (like XML), **no schema**
  - Collection docs may have different attributes
Data Example

Collection inventory

```json
{
  item: "ABC2",
  details: { model: "14Q3", manufacturer: "M1 Corporation" },
  stock: [ { size: "M", qty: 50 } ],
  category: "clothing"
}

{
  item: "MNO2",
  details: { model: "14Q3", manufacturer: "ABC Company" },
  category: "clothing"
}
```

(db.inventory.insert(  
  {  
    item: "ABC1",
    details: {model: "14Q3",manufacturer: "XYZ Company"},
    stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
    category: "clothing"
  }  
))

(docs.mongdb.org)
Example of a Simple Query

Collection orders

```json
{  
    _id: "a",
    cust_id: "abc123",
    status: "A",
    price: 25,
    items: [
        { sku: "mmm", qty: 5, price: 3 },
        { sku: "nnn", qty: 5, price: 2 }
    ]
}
{  
    _id: "b",
    cust_id: "abc124",
    status: "B",
    price: 12,
    items: [
        { sku: "nnn", qty: 2, price: 2 },
        { sku: "ppp", qty: 2, price: 4 }
    ]
}
```

In SQL it would look like this:

```sql
SELECT cust_id, price
FROM orders
WHERE status="A"
```
Map-Reduce in MongoDB

Collection orders

```json
{
    _id: "a",
    cust_id: "abc123",
    status: "A",
    price: 25
}
{
    _id: "b",
    cust_id: "abc124",
    status: "B",
    price: 12
}
{
    _id: "c",
    cust_id: "abc123",
    status: "A",
    price: 20
}
```

In SQL it would look like this:

```sql
SELECT cust_id, sum(price)
FROM orders
GROUP BY cust_id;
```

But orders are distributed all over...

Sum up the purchases per customer:

```json
{
    _id: "abc123",
    price: 45
}
{
    _id: "abc124",
    price: 12
}
```

2 options now:
(1) Built-in MongoDB aggregates
(2) MapReduce + custom JS code (more flexible, less smart)

Let’s MR it

Collection PurchasesPerCustomer
The Map-Reduce Programming Model

1. Map

(a,5) → (b,5) → (b,4) → (c,2) → (c,3)

(a,3) → (a,2) → (c,1) → (c,2) → (b,2) → (a,5)

(d,3) → (d,2) → (b,1) → (b,4) → (b,2) → (b,5) → (a,5)

(a,3) → (c,3) → (d,3) → (d,1)
The Map-Reduce Programming Model

1. Map
2. Shuffle
The Map-Reduce Programming Model

1. Map
2. Shuffle
3. Reduce
Map-Reduce in MongoDB

Collection orders

```javascript
{
  _id: "a",
  cust_id: "abc123",
  status: "A",
  price: 25
}
{
  _id: "b",
  cust_id: "abc124",
  status: "B",
  price: 12
}
{
  _id: "c",
  cust_id: "abc123",
  status: "A",
  price: 20
}
```

Sum up the purchases per customer:

```javascript
{
  _id: "abc123",
  price: 45
}
{
  _id: "abc124",
  price: 12
}
```

Collection PurchasesPerCustomers

```javascript
var emitCustPrice = function() {
  emit(this.cust_id, this.price);
};

var sumUp = function(custId, prices) {
  return Array.sum(prices);
};

db.orders.mapReduce(
  emitCustPrice, sumUp,
  {out: "PurchasesPerCustomer" }
)
```
Outline

• Introduction
• Transaction Consistency
• Column-Family Stores
• Key-Value Stores
  ▪ Example: Redis
• Document Stores
  ▪ Example: MongoDB
• Graph Databases
  ▪ Example: neo4j
• Concluding Remarks
Graph Databases

• Restricted case of a relational schema:
  – Nodes (+labels/properties)
  – Edges (+labels/properties)

• Motivated by the popularity of network/communication oriented applications

• Efficient support for graph-oriented queries
  – Reachability, graph patterns, path patterns
  – Ordinary RDBs either not support or inefficient for such queries
    • Path of length k is a k-wise self join; yet a very special one...

• Specialized languages for graph queries
  – For example, pattern language for paths

• Plus distributed, 2-of-CAP, etc.
  – Depending on the design choices of the vendor
Example Databases

• Graph with nodes/edges marked with labels and properties (labeled property graph)
  – Sparksee (DEX) (Java, 1st release 2008)
  – neo4j (Java, 1st release 2010)
  – InfiniteGraph (Java/C++, 1st release 2010)
  – OrientDB (Java, 1st release 2010)

• Triple stores: Support W3C RDF and SPARQL, also viewed as graph databases
  – MarkLogic, AllegroGraph, Blazegraph, IBM SystemG, Oracle Spatial & Graph, OpenLink Virtuoso, ontotext
Outline

• Introduction
• Transaction Consistency
• Column-Family Stores
• Key-Value Stores
  ▪ Example: Redis
• Document Stores
  ▪ Example: MongoDB
• Graph Databases
  ▪ Example: neo4j
• Concluding Remarks
• Open source, written in Java
  – First version released 2010
• Supports the Cyber query language
• Clustering support
  – Replication and sharding through master-slave architectures
• Used by ebay, Walmart, Cisco, National Geographic, TomTom, Lufthansa, ...
Examples taken from *Graph Databases* by Robinson, Webber, and Eifrem (O’Reilly) – free eBook
The Graph Data Model in Cypher

- **Labeled property graph model**
- **Node**
  - Has a set of *labels* (typically one label)
  - Has a set of *properties* key:value (where value is of a primitive type or an array of primitives)
- **Edge** (relationship)
  - Directed: node → node
  - Has a *name*
  - Has a set of *properties* (like nodes)
Example: Cypher Graph for Social Networks

- **User**
  - name: Billy
  - label: User
  - property: follows

- **User**
  - name: Ruth

- **User**
  - name: Harry

- Direction: follows
Another Example: Email Exchange
MATCH (bob:User{username:'Bob'})-[:SENT]->(email)-[:CC]->(alias), (alias)-[:ALIAS_OF]->(bob)
RETURN email
CREATE (alice:User {username:'Alice'}),
(bob:User {username:'Bob'}),
(charlie:User {username:'Charlie'}),
(davina:User {username:'Davina'}),
(Edward:User {username:'Edward'}),
 Alice)-[.:ALIAS_OF]->(bob)
MATCH (bob:User {username:'Bob'}),
(charlie:User {username:'Charlie'}),
(davina:User {username:'Davina'}),
(educ:User {username:'Edward'}),
(a:User)
CREATE (a)-[:ALIAS_OF]->(bob)

CREATE (alice:User {username:'Alice'}),
(bob:User {username:'Bob'}),
(charlie:User {username:'Charlie'}),
(davina:User {username:'Davina'}),
(educ:User {username:'Edward'}),
(a)-[:ALIAS_OF]->(bob)

MATCH (bob:User {username:'Bob'}),
(charlie:User {username:'Charlie'}),
(davina:User {username:'Davina'}),
(educ:User {username:'Edward'})
CREATE (bob)-[:EMAILED]->(charlie),
(bob)-[:CC]->(davina),
(bob)-[:BCC]->(educ)
MATCH p = (email:Email {id:'6'})
<-[:REPLY_TO]*1..4]-(:Reply)<-[[:SENT]]-(replier)
RETURN replier.username AS replier, length(p) - 1 AS depth
ORDER BY depth
Outline

• Introduction
• Transaction Consistency
• Column-Family Stores
• Key-Value Stores
  ▪ Example: Redis
• Document Stores
  ▪ Example: MongoDB
• Graph Databases
  ▪ Example: neo4j
• Concluding Remarks
Concluding Remarks on Common NoSQL

• Aim to avoid join & ACID overhead
  – Joined within, correctness compromised for quick answers; believe in best effort

• Avoids the idea of a schema

• Query languages are more imperative
  – And less declarative
  – Developer better knows what’s going on; less reliance on smart optimization plans
  – More responsibility on developers

• No standard well studied languages (yet)