Lecture 9:
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 solutions for distributed analysis of massive data

Database solutions need to \textit{scale out}
(utilize distribution, “scale horizontally”)
What should be done to allow for effective distributed, data 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
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 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 (systems 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
- Produced 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
• 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**

<table>
<thead>
<tr>
<th>txn₁</th>
<th>txn₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td>Read(A,v)</td>
<td>Read(A,v)</td>
</tr>
<tr>
<td>v = v-100</td>
<td>v = v-100</td>
</tr>
<tr>
<td>Write(A,v)</td>
<td>Write(A,v)</td>
</tr>
<tr>
<td>Read(B,w)</td>
<td>Read(B,w)</td>
</tr>
<tr>
<td>w=w+100</td>
<td>w=w+100</td>
</tr>
<tr>
<td>Write(B,w)</td>
<td>Write(B,w)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>

- **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

txn₁

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

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

txn₂

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

Read(A,x)
x = x-100
Write(A,x)
Read(C,y)
y=y+100
Write(C,y)
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 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 be disconnected 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 retunes 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
Eric Brewer’s CAP Theorem:

A distributed service can support at most **two** out of C, A and P
Historical Note

• 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
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

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

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 Store: each column stored separately (still SQL)

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

Column-Family Store: NoSQL

(Cassandra model)
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

write(k,t)

write(k,t)

write(k,t)

write(k,t)

write(k,t)

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 ⟨key, opaque-value⟩
  – 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
• 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>
</table>

- **get x** >> 10
- **hget h y** >> 5
- **hkeys p:22** >> name, age
- **smembers s** >> 20, Alma
- **scard s** >> 2

- **llen l** >> 3
- **lrange l 1 2** >> a, b
- **lindex l 2** >> b
- **lpop l** >> c
- **rpop l** >> b
• A key can be any <256MB binary string
  – For example, JPEG image
• Some key operations:
  – List all keys: \texttt{keys *}
  – Remove all keys: \texttt{flushall}
  – Check if a key exists: \texttt{exists k}
• You can configure the persistency model
  – \texttt{save m k} means save every \texttt{m} seconds if at least \texttt{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

• 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
  – Designed 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
MongoDB

• Open source, 1st 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 \( \textit{collections} \) \( \Rightarrow \text{generalizes relation} \)
- Collection = sequence of \( \textit{documents} \) \( \Rightarrow \text{generalizes tuple} \)
- Document = \{attribute\(_1\):value\(_1\),...,attribute\(_k\):value\(_k\}\}
- Attribute = string \( \text{attribute}_i \neq \text{attribute}_j \)
- Value = primitive value (string, number, date, ...), or a document, or an array \( \text{array} \)
- Array = \([\text{value}_1,...,\text{value}_n]\]

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

Collection inventory

```
{
    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"
}
```

(docs.mongodb.org)

Document insertion

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

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

Collection orders

```
{
    _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 } ]
}
```

db.users.find(
    { status: "A" },
    { cust_id: 1, price: 1, _id: 0 }
)

In SQL it would look like this:

```
SELECT cust_id, price
FROM orders
WHERE status="A"
```

```json
{
    cust_id: "abc123",
    price: 25
}
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:

```
{
    _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) → (d,3) → (d,2) → (a,2) → (b,1) → (b,2) → (a,3) → (a,2) → (c,1) → (c,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
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"}
)
```

Sum up the purchases per customer:

```javascript
{  
  _id: "abc123",
  cust_id: "abc123",
  status: "A",
  price: 20
}
```

```javascript
{  
  _id: "abc124",
  cust_id: "abc124",
  status: "B",
  price: 12
}
```

```javascript
{  
  _id: "a",
  cust_id: "abc123",
  status: "A",
  price: 25
}
```

```javascript
{  
  _id: "b",
  cust_id: "abc124",
  status: "B",
  price: 12
}
```

```javascript
{  
  _id: "c",
  cust_id: "abc123",
  status: "A",
  price: 20
}
```

Collection PurchasesPerCustomers
• 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
  – (Next lecture)
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 Cypher 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)
Graphs are extremely useful in understanding a wide diversity of datasets in fields such as science, government, and business. The real world—unlike the forms-based model behind the relational database—is rich and interrelated: uniform and rule-bound in parts, exceptional and irregular in others. Once we understand graphs, we begin to see them in all sorts of places.

Gartner, for example, identifies five graphs in the world of business—social, intent, consumption, interest, and mobile—and says that the ability to leverage these graphs provides a "sustainable competitive advantage." For example, Twitter's data is easily represented as a graph. In Figure 1-1 we see a small network of Twitter users. Each node is labeled User, indicating its role in the network. These nodes are then connected with relationships, which help further establish the semantic context: namely, that Billy follows Harry, and that Harry, in turn, follows Billy. Ruth and Harry likewise follow each other, but sadly, although Ruth follows Billy, Billy hasn't (yet) reciprocated.

Of course, Twitter's real graph is hundreds of millions of times larger than the example in Figure 1-1, but it works on precisely the same principles. In Figure 1-2 we've expanded the graph to include the messages published by Ruth.
Another Example: Email Exchange

This leads to the more complex, and interesting, graph we see in Figure 3-10.
Query Example

MATCH (bob:User{username: 'Bob'})-[[:SENT]->(email)]-[[:CC]->(alias)],
(alias)-[[:ALIAS_OF]->(bob)]
RETURN email
Creating Graph Data

This first modeling attempt results in a star-shaped graph with Bob at the center. His actions of emailing, copying, and blind-copying are represented by relationships that extend from Bob to the nodes representing the recipients of his mail. As we see in Figure 3-8, however, the most critical element of the data, the actual email, is missing.

Figure 3-8. Missing email node leads to lost information

This graph structure is lossy, a fact that becomes evident when we pose the following query:

MATCH (bob:User {username:'Bob'})-[e:EMAILED]->(charlie:User {username:'Charlie'})
RETURN e

This query returns the EMAILED relationships between Bob and Charlie (there will likely be one for each email that Bob has sent to Charlie). This tells us that emails have been exchanged, but it tells us nothing about the emails themselves:

```
+----------------+
| e              |
+----------------+
| :EMAILED[1] {} |
+----------------+
1 row
```

Creating Graph Data

This first modeling attempt results in a star-shaped graph with Bob at the center. His actions of emailing, copying, and blind-copying are represented by relationships that extend from Bob to the nodes representing the recipients of his mail. As we see in Figure 3-8, however, the most critical element of the data, the actual email, is missing.

Figure 3-8. Missing email node leads to lost information

This graph structure is lossy, a fact that becomes evident when we pose the following query:

```
MATCH (bob:User {username:'Bob'})-[e:EMAILED]->(charlie:User {username:'Charlie'})
RETURN e
```

This query returns the `EMAILED` relationships between Bob and Charlie (there will likely be one for each email that Bob has sent to Charlie). This tells us that emails have been exchanged, but it tells us nothing about the emails themselves:

```
+----------------+
| e              |
+----------------+
| :EMAILED[1] {} |
+----------------+
```

54 | Chapter 3: Data Modeling with Graphs

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

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'}),
(Edward:User {username:'Edward'})
CREATE (bob)-[:EMAILED]->(charlie),
(bob)-[:CC]->(davina),
(bob)-[:BCC]->(Edward)
Figure 3-12. Explicitly modeling replies in high fidelity
Here we capture each matched path, binding it to the identifier \( p \). In the \( \text{RETURN} \) clause we then calculate the length of the reply-to chain (subtracting 1 for the \( \text{SENT} \) relationship), and return the replier's name and the depth at which he or she replied.

This query returns the following results:

<table>
<thead>
<tr>
<th>replier</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Davina</td>
<td>1</td>
</tr>
<tr>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>Charlie</td>
<td>2</td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
</tr>
</tbody>
</table>

4 rows

We see that both Davina and Bob replied directly to Bob's original email; that Charlie replied to one of the replies; and that Bob then replied to one of the replies to a reply.

It's a similar pattern for a forwarded email, which can be regarded as a new email that simply happens to contain some of the text of the original email. As with the reply case, we model the new email explicitly. We also reference the original email from the

MATCH \( p = (\text{email:Email \{id:'6'\}}) \)  
<-[\text{:REPLY_TO}*1..4]-(:Reply)<-[\text{:SENT}]-(:replier) 
RETURN \( \text{replier.username} \text{ AS replier}, \text{length(p)} - 1 \text{ AS depth} \) 
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)