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
  - 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 scale out
  (utilize distribution, "scale horizontally")

Compromises Required

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
    - DeCandio, Westin, J., Jones, A., Kakapool, L., Lakshman, P., Sivasubramanian, S.
      - 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 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

Database Sharding

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

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

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

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

Scheduling Example

• 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)

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), getLike(post)
  – Each client can talk to any server
• Servers return responses
  – e.g., ack, {user1, ...., userk}
• 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 returns a value
• Partition tolerance: service continues to function on network failures
  – As long as clients can reach servers
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

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

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
  - InfiniDB
- 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
- 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

### 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

### 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 of Redis Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set x 10</td>
<td>Store the value 10 in key x</td>
</tr>
<tr>
<td>hget x</td>
<td>Get the value of key x</td>
</tr>
<tr>
<td>hkeys x</td>
<td>Get the keys of hash x</td>
</tr>
<tr>
<td>hvals x</td>
<td>Get the values of hash x</td>
</tr>
<tr>
<td>hset x y</td>
<td>Store the value y in key x</td>
</tr>
<tr>
<td>hdel x</td>
<td>Delete key x</td>
</tr>
<tr>
<td>hincr x</td>
<td>Increment the value of key x by 1</td>
</tr>
<tr>
<td>hdel x</td>
<td>Delete key x</td>
</tr>
<tr>
<td>hstrlen</td>
<td>Get the length of value of key x</td>
</tr>
<tr>
<td>hget x</td>
<td>Get the value of key x</td>
</tr>
<tr>
<td>hset x y</td>
<td>Store the value y in key x</td>
</tr>
<tr>
<td>hdel x</td>
<td>Delete key x</td>
</tr>
<tr>
<td>hstrlen</td>
<td>Get the length of value of key x</td>
</tr>
<tr>
<td>hget x</td>
<td>Get the value of key x</td>
</tr>
<tr>
<td>hset x y</td>
<td>Store the value y in key x</td>
</tr>
<tr>
<td>hdel x</td>
<td>Delete key x</td>
</tr>
<tr>
<td>hstrlen</td>
<td>Get the length of value of key x</td>
</tr>
<tr>
<td>hget x</td>
<td>Get the value of key x</td>
</tr>
</tbody>
</table>

### Example of Redis Commands

- `set x 10`: Store the value 10 in key x
- `hget x`: Get the value of key x
- `hkeys x`: Get the keys of hash x
- `hvals x`: Get the values of hash x
- `hset x y`: Store the value y in key x
- `hdel x`: Delete key x
- `hincr x`: Increment the value of key x by 1
- `hdel x`: Delete key x
- `hstrlen`: Get the length of value of key x
- `hget x`: Get the value of key x
- `hset x y`: Store the value y in key x
- `hdel x`: Delete key x
- `hstrlen`: Get the length of value of key x
- `hget x`: Get the value of key x

### 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, Ruby, Tcl, C, C++, C#, Java, R, …
Additional Notes

- 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

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

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
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 collections
- Collection = sequence of documents
- Document = (attribute1, value1, attribute2, value2)
- Attribute = string (attribute=value)
- Value = primitive value (string, number, date, …), or a document, or an array
- Array = [value1, value2]

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

Data Example

```
Collection inventory

{  
  item: "ABC1",  
  details: {  
    model: "14Q3", manufacturer: "M3 Corporation"  
  },  
  cat: "clothing"  
}
```

```
{  
  item: "ABC2",  
  details: {  
    model: "14Q3", manufacturer: "ABC Company"  
  },  
  cat: "clothing"  
}
```

```
{  
  item: "MNO2",  
  details: {  
    model: "14Q3", manufacturer: "M1 Corporation"  
  },  
  cat: "clothing"  
}
```

Example of a Simple Query

```
DB.inventory.insert
  
  {  
    item: "ABC1",  
    details: {  
      model: "14Q3", manufacturer: "XYZ Company"  
    },  
    cat: "clothing"  
  }
```

```
DB.users.update
  
  {  
    _id: "a",  
    cust_id: "abc123",  
    status: "A",  
    price: 25,  
    items: [{  
      sku: "mmm", qty: 5, price: 3 },  
      {  
        sku: "nmm", qty: 2, price: 2 }  
    ]  
  }
```

```
DB.users.find
  
  {  
    _id: "b",  
    cust_id: "abc124",  
    status: "B",  
    price: 12,  
    items: [{  
      sku: "nmm", qty: 2, price: 2 },  
      {  
        sku: "pmm", qty: 4, price: 4 }  
    ]  
  }
```

Map-Reduce in MongoDB

```
Collection orders

{  
  _id: "a",  
  cust_id: "abc123",  
  status: "A",  
  price: 25,  
  items: [{  
    sku: "mmm", qty: 5, price: 3 },  
    {  
      sku: "nmm", qty: 2, price: 2 }  
  ]  
}
```

```
Collection orders

{  
  _id: "b",  
  cust_id: "abc124",  
  status: "B",  
  price: 12,  
  items: [{  
    sku: "nmm", qty: 2, price: 2 },  
    {  
      sku: "pmm", qty: 4, price: 4 }  
  ]  
}
```

In SQL it would look like this:

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

Let’s MR it

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id</td>
<td>string</td>
</tr>
<tr>
<td>cust_id</td>
<td>string</td>
</tr>
<tr>
<td>status</td>
<td>string</td>
</tr>
<tr>
<td>price</td>
<td>number</td>
</tr>
</tbody>
</table>

Example of a Simple Query

```
db.users.update
  
  {  
    _id: "a",  
    cust_id: "abc123",  
    status: "A",  
    price: 25,  
    items: [{  
      sku: "mmm", qty: 5, price: 3 },  
      {  
        sku: "nmm", qty: 2, price: 2 }  
    ]  
  }
```

```
db.users.find
  
  {  
    _id: "b",  
    cust_id: "abc124",  
    status: "B",  
    price: 12,  
    items: [{  
      sku: "nmm", qty: 2, price: 2 },  
      {  
        sku: "pmm", qty: 4, price: 4 }  
    ]  
  }
```

Example of a Simple Query

```
Collection orders

{  
  _id: "a",  
  cust_id: "abc123",  
  status: "A",  
  price: 25,  
  items: [{  
    sku: "mmm", qty: 5, price: 3 },  
    {  
      sku: "nmm", qty: 2, price: 2 }  
    ]  
}
```

```
Collection orders

{  
  _id: "b",  
  cust_id: "abc124",  
  status: "B",  
  price: 12,  
  items: [{  
    sku: "nmm", qty: 2, price: 2 },  
    {  
      sku: "pmm", qty: 4, price: 4 }  
    ]  
}
```

In SQL it would look like this:

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

Let’s MR it

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id</td>
<td>string</td>
</tr>
<tr>
<td>cust_id</td>
<td>string</td>
</tr>
<tr>
<td>status</td>
<td>string</td>
</tr>
<tr>
<td>price</td>
<td>number</td>
</tr>
</tbody>
</table>

Example of a Simple Query

```
Collection orders

{  
  _id: "a",  
  cust_id: "abc123",  
  status: "A",  
  price: 25,  
  items: [{  
    sku: "mmm", qty: 5, price: 3 },  
    {  
      sku: "nmm", qty: 2, price: 2 }  
    ]  
}
```

```
Collection orders

{  
  _id: "b",  
  cust_id: "abc124",  
  status: "B",  
  price: 12,  
  items: [{  
    sku: "nmm", qty: 2, price: 2 },  
    {  
      sku: "pmm", qty: 4, price: 4 }  
    ]  
}
```

In SQL it would look like this:

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

Let’s MR it

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id</td>
<td>string</td>
</tr>
<tr>
<td>cust_id</td>
<td>string</td>
</tr>
<tr>
<td>status</td>
<td>string</td>
</tr>
<tr>
<td>price</td>
<td>number</td>
</tr>
</tbody>
</table>
The Map-Reduce Programming Model

1. Map
2. Shuffle

Map-Reduce in MongoDB

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

var sumPrice = function(cust, prices) {
  return Array.sum(prices);
};

db.orders.mapReduce(
  emitCustPrice,
  sumPrice,
  reduce
)
```

Collection
PurchasesPerCustomers

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
  - Relatively graph patterns, path patterns
  - Ordinary RDBs either not support or inefficient for such queries
  - Path of length k is a k-join and 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)
Graphs Are Everywhere

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.

Figure 1-1. A small social graph

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

(email_4)-[:TO]->(davina), (email_4)-[:TO]->(edward);
CREATE (email_5:Email {id: ‘5’, content: ‘email contents’}),
(email_5)-[:SENT]->(davina), (email_5)-[:TO]->(alice), (email_5)-[:BCC]->(bob), (email_5)-[:BCC]->(edward);

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

Figure 3-10. A graph of email interactions

Examples taken from Graph Databases by Robinson, Webber, and Ethim (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)
Query Example

```sql
MATCH (email:Email {
  username: 'Alice',
  from: 'Bob',
  to: 'Charlie',
  cc: 'Davina',
  bcc: 'Charlie'
}) -[:SENT]-> (email_1:Email {
  username: 'Charlie',
  from: 'Davina',
  to: 'Bob',
}) -[:BCC]-> (email_2:Email {
  username: 'Bob',
  from: 'Charlie',
  to: 'Davina',
})
RETURN email
```

Creating Graph Data

```sql
CREATE (email:Email {
  username: 'Alice',
  from: 'Bob',
  to: 'Charlie',
  cc: 'Davina',
  bcc: 'Charlie'
}) -[:SENT]-> (email_1:Email {
  username: 'Charlie',
  from: 'Davina',
  to: 'Bob',
}) -[:BCC]-> (email_2:Email {
  username: 'Bob',
  from: 'Charlie',
  to: 'Davina',
})
```

Another Example

```sql
MATCH p = (email:Email {
  username: 'Davina',
  from: 'Charlie',
  to: 'Bob',
}) -[:BCC]-> (email_3:Email {
  username: 'Bob',
  from: 'Davina',
  to: 'Charlie',
}) -[:SENT]-> (email_4:Email {
  username: 'Bob',
  from: 'Charlie',
  to: 'Davina',
})
RETURN COUNT(DISTINCT p)
```

Creating Graph Data

```sql
CREATE (email:Email {
  username: 'Alice',
  from: 'Bob',
  to: 'Charlie',
  cc: 'Davina',
  bcc: 'Charlie'
}) -[:SENT]-> (email_1:Email {
  username: 'Charlie',
  from: 'Davina',
  to: 'Bob',
}) -[:BCC]-> (email_2:Email {
  username: 'Bob',
  from: 'Charlie',
  to: 'Davina',
})
```

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)