COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 23
NOSQL DATABASES

NOSQL DATABASES: OUTLINE

The NOSQL Movement

Key-value Stores

Tuple and Document Stores

Column-oriented Databases

Graph Databases

Acknowledgment
These slides are adapted from the course materials provided for the textbook:
[Principles of Database Management]

THE NOSQL MOVEMENT

- RDBMSs put a lot of emphasis on keeping data consistent.
  - formal schema, data types, referential integrity, ACID properties, etc.
- The focus on consistency may hamper flexibility and scalability.
  - Vertical scaling: extend DBMS storage capacity and/or CPU power.
  - Horizontal scaling: add more DBMS servers arranged in a cluster.
- RDBMSs are *not good* at extensive horizontal scaling due to large coordination overhead because of the focus on consistency.
- Rigid database schemas and rich querying functionality are often overkill for many applications resulting in a need for DBMSs that can deal effectively with
  - massive data volumes
  - flexible data structures
  - scalability and availability

One size fits all is no longer appropriate.

THE NOSQL MOVEMENT (CONT'D)

- NoSQL DBMSs store and manipulate data in formats other than tabular relations (i.e., non-relational databases).
- NoSQL DBMSs aim at near-linear horizontal scalability.
  - Rather than placing data on a central server, it is distributed over a cluster of nodes to improve performance as well as availability.
- NoSQL DBMSs provide much simpler querying functionality and/or APIs (usually <u>not</u> SQL).
- NoSQL DBMSs introduce the concept of eventual consistency.
  - Data, and its replicas, will become *consistent at some point in time* after each transaction, but *continuous consistency is not guaranteed*.

KEY-VALUE STORES

- A key-value-based database stores data as (key, value) pairs.
  - Keys are unique and are the sole "search" criterion to retrieve the corresponding value.
  - Hash map, hash table or a dictionary are used to store key-value pairs (e.g., Java HashMap class).
- The hash table is distributed over different locations (to support horizontal scalability).

KEY-VALUE STORES (CONT'D)

- First, keys (e.g., “Dustin”, “Andy”) are hashed by a hash function.
  - Recall: a hash function takes an arbitrary value of arbitrary size and maps it to a key with a fixed size called the hash value.
  - Each hash value can then be mapped to a memory address.

Example: Map sailor name and rating where name is the key and rating is the value (i.e., hash function(name) = hash).

Key	Value
Dustin	7
Andy	8
Horatio	7
Zorba	10

hash function →

Hash	Key-Value Entries
1	(Dustin, 7)
3	(Andy, 8)
7	(Horatio, 7)
8	(Zorba, 10)

KEY-VALUE STORES (CONT'D)

Then, the data are partitioned and assigned to different nodes (servers).

❖ **This is called sharding; nodes are called shards.**

Example:

– Hash every key ("Dustin", "Andy", …) to a server identifier.

– index(hash) = mod(hash, number of servers) (i.e., hash of key hashes).

Key
Dustin
Andy
Horatio
Zorba

hash function

Hash	index (= mod 3)
1	1
3	0
7	1
8	2

For instance, hash 8 divided by 3 has a remainder of 2
⟹ assign (Zorba, 10) to server 2.

Server 0	Key-Value Entries
3	(Andy, 8)

Server 1	Key-Value Entries
1	(Dustin, 7)
7	(Horatio, 7)

Server 2	Key-Value Entries
8	(Zorba, 10)

REQUEST COORDINATION

- **Request coordination** routes a request for data to the appropriate node (shard) in the network.
- In many NoSQL implementations all nodes implement the same functionality (i.e., nodes are transparent to users) and, consequently, all nodes in the network can perform the role of request coordinator (RC).
- A membership protocol ensures that all nodes remain informed at all times of the other nodes in the network.
- Two basic components of a membership protocol in request coordination.
  – Dissemination: Membership list updates are based on periodic and pairwise (*gossip*) communications (to support node awareness).
  – Failure detection: Checking that nodes are not available in the network (to avoid routing to nodes that are down).

CONSISTENT HASHING

- **Consistent hashing** schemes avoid having to remap too many keys to a new server when servers are added or removed.

Example:

Key
Dustin
Andy
Horatio
Zorba

hash function

Hash	index (= mod 3)
1	1
3	0
7	1
8	2

Server 0	Entries
3	(Andy, 8)

Server 1	Entries
1	(Dustin, 7)
7	(Horatio, 7)

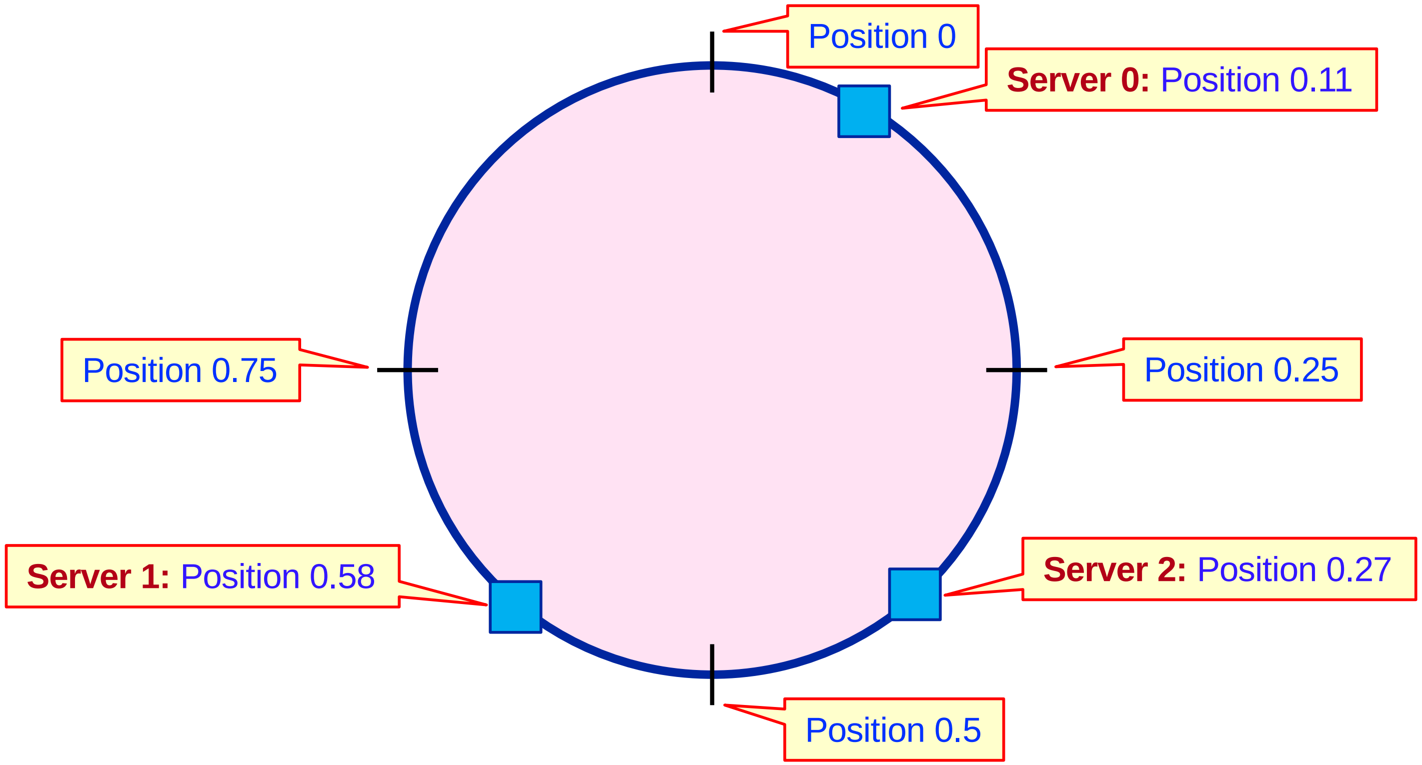
Server 2	Entries
8	(Zorba, 10)

Key hash	*n*		
	3	2	4
0	0	0	0
1	1	1	1
2	2	0	2
3	0	1	3
4	1	0	0
5	2	1	1
6	0	0	2
7	1	1	3
8	2	0	0
9	0	1	1

Change in the number of servers $n \Longrightarrow$ many items must be moved to a different server (highlighted in blue).
This is not a desirable outcome where servers are likely to be removed or added often.

CONSISTENT HASHING (cont'd)

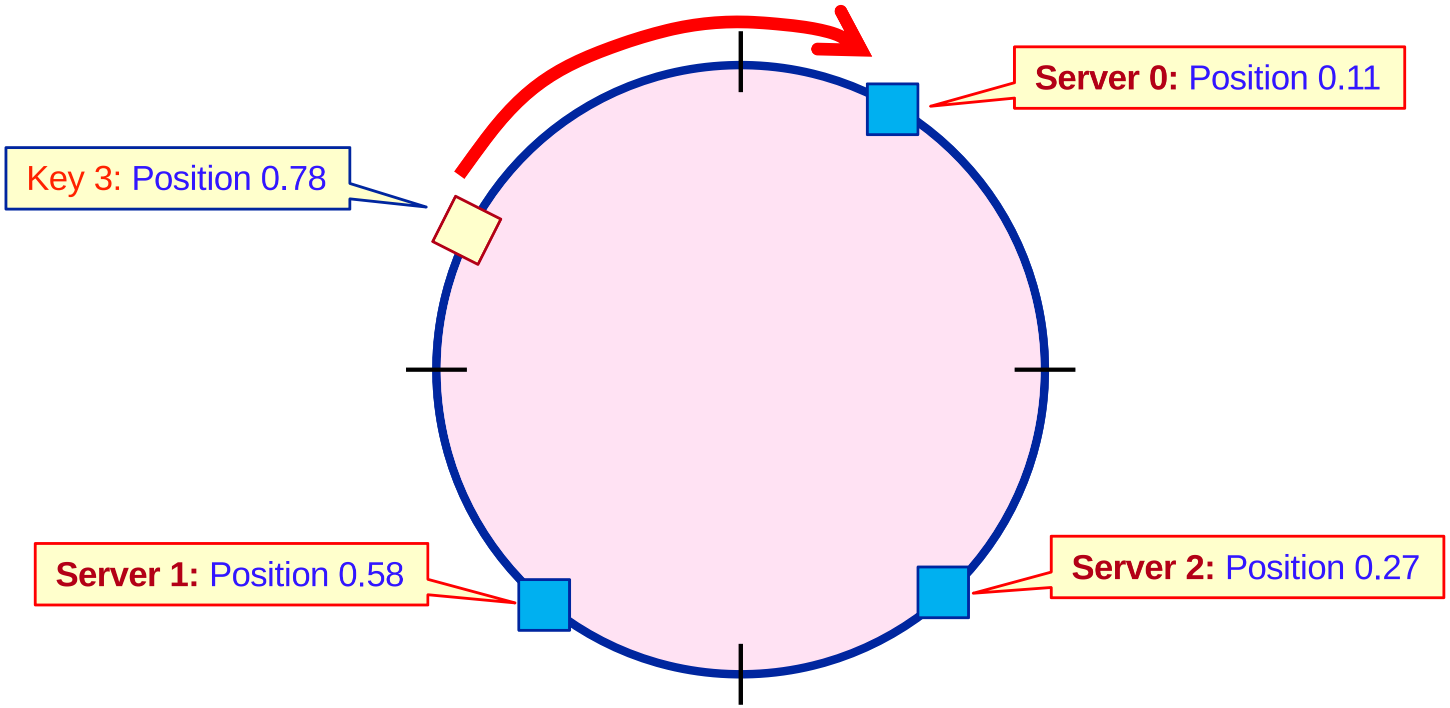
- At the core of a consistent hashing scheme is a so-called ring-topology (conceptual), which is basically a representation of the number range [0,1] as shown in the figure.



- First, all three servers with identifiers 0, 1, 2 are hashed to place them in a position on this ring.

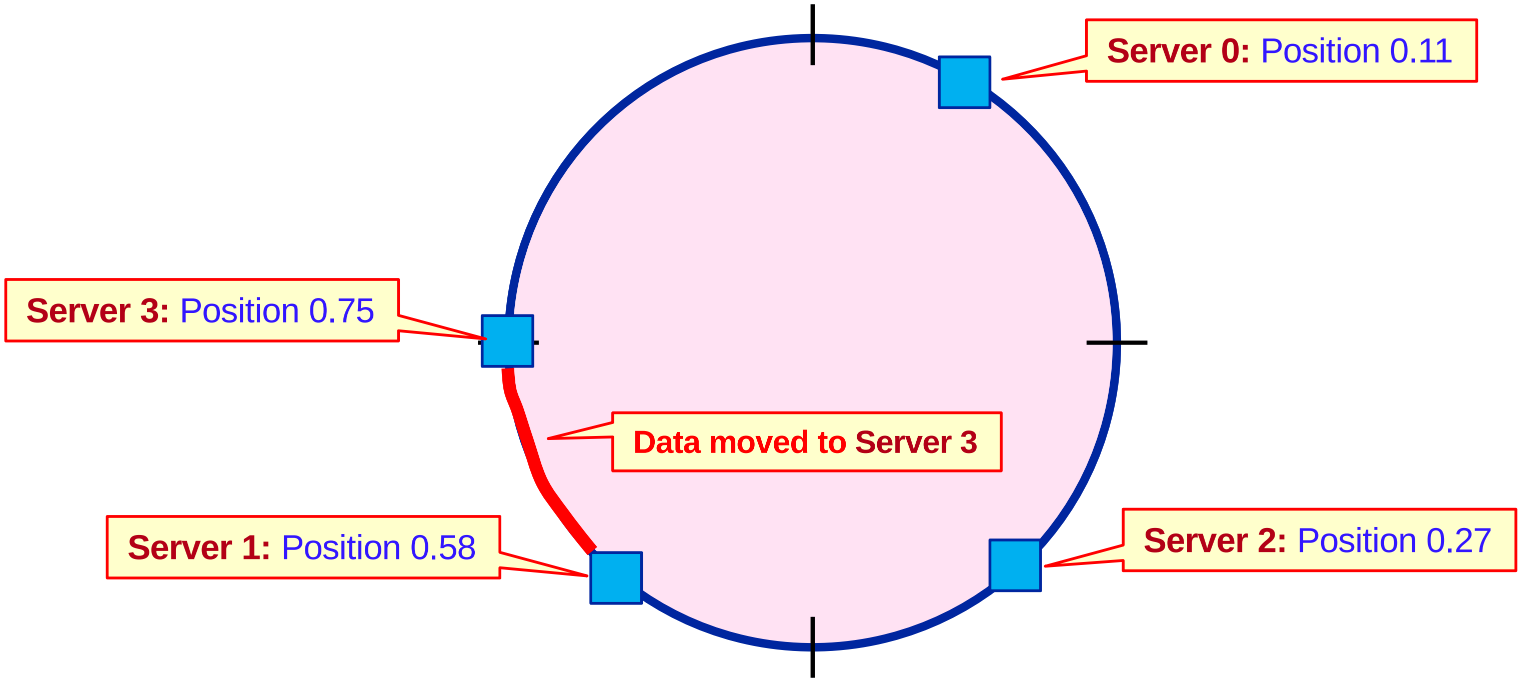
CONSISTENT HASHING (CONT'D)

- Then, each key is hashed to a position on the ring and the actual key-value pair is stored on the first server that appears clockwise from the hashed point on the ring.



CONSISTENT HASHING (CONT'D)

- If we add a new server, **Server 3**, to the ring, we only need to move the keys positioned on the red highlighted section of the ring to the new server.

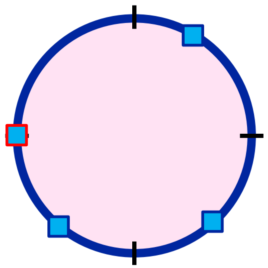


CONSISTENT HASHING (cont'd)

- Due to the uniformity property of a “good” hash function, roughly $1/n$ key-value pairs will end up being stored on each server.
- Most of the key-value pairs will remain unaffected if a server is added or removed.
- Typically, the fraction of keys that need to be moved when using this scheme is about $k/(n + 1)$ where k is the number of keys and n the number of servers.
- This is a much smaller fraction than is the case for modulo-based hashing.

REPLICATION

- Problems with consistent hashing:
  - If two servers are mapped too close to one another, one of these ends up with few keys to store.
  - When a server is added, all the keys moved to this new node originate from just one other server.



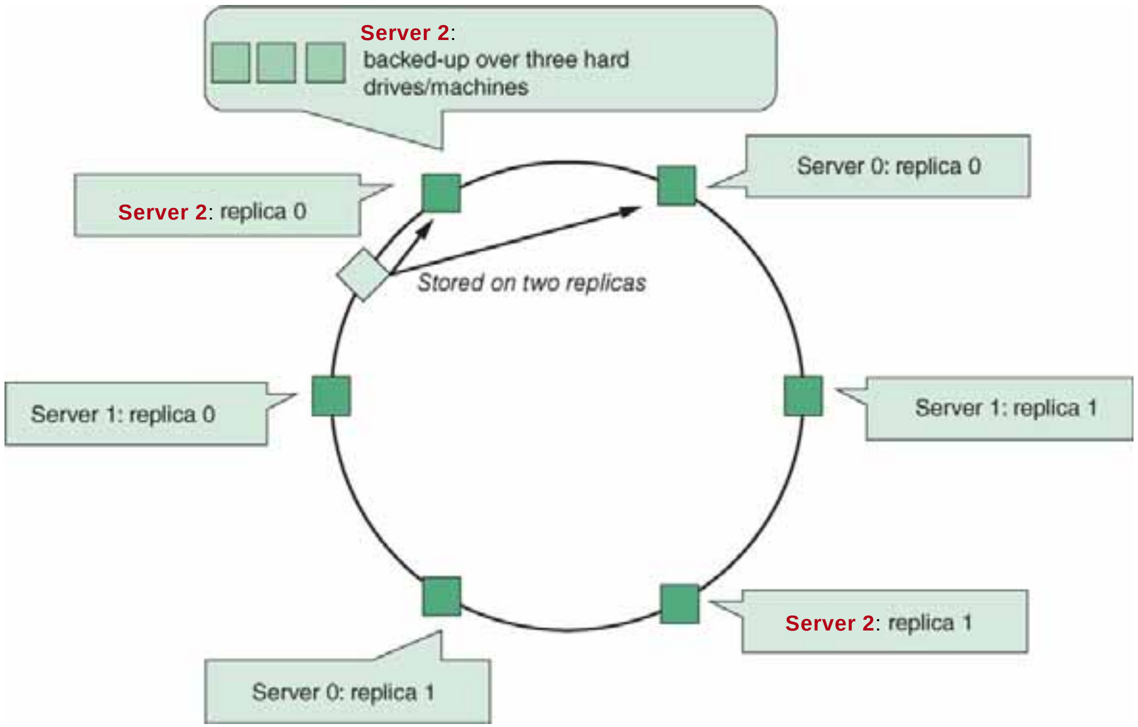
- Instead of mapping a server *S* to a single point on the ring, we map it to multiple positions, called replicas.
  - For each physical server *S*, we end up with *r* (the number of replicas) points on the ring, called virtual nodes.

  Note: Each replica (virtual node) still represents the same physical instance.

- The consistent hashing mechanism can be extended to duplicate key-value pairs across multiple nodes.
  - E.g., we can store a key-value pair on two or more nodes clockwise from the key's position on the ring.

REDUNDANCY

- As well as creating replicas, a full redundancy scheme can be set up in which each node itself corresponds to multiple physical machines, each storing a fully redundant copy of the data.
  - E.g. **Server 2** can be backed up (full redundant copy) over three hard drives or machines.



EVENTUAL CONSISTENCY

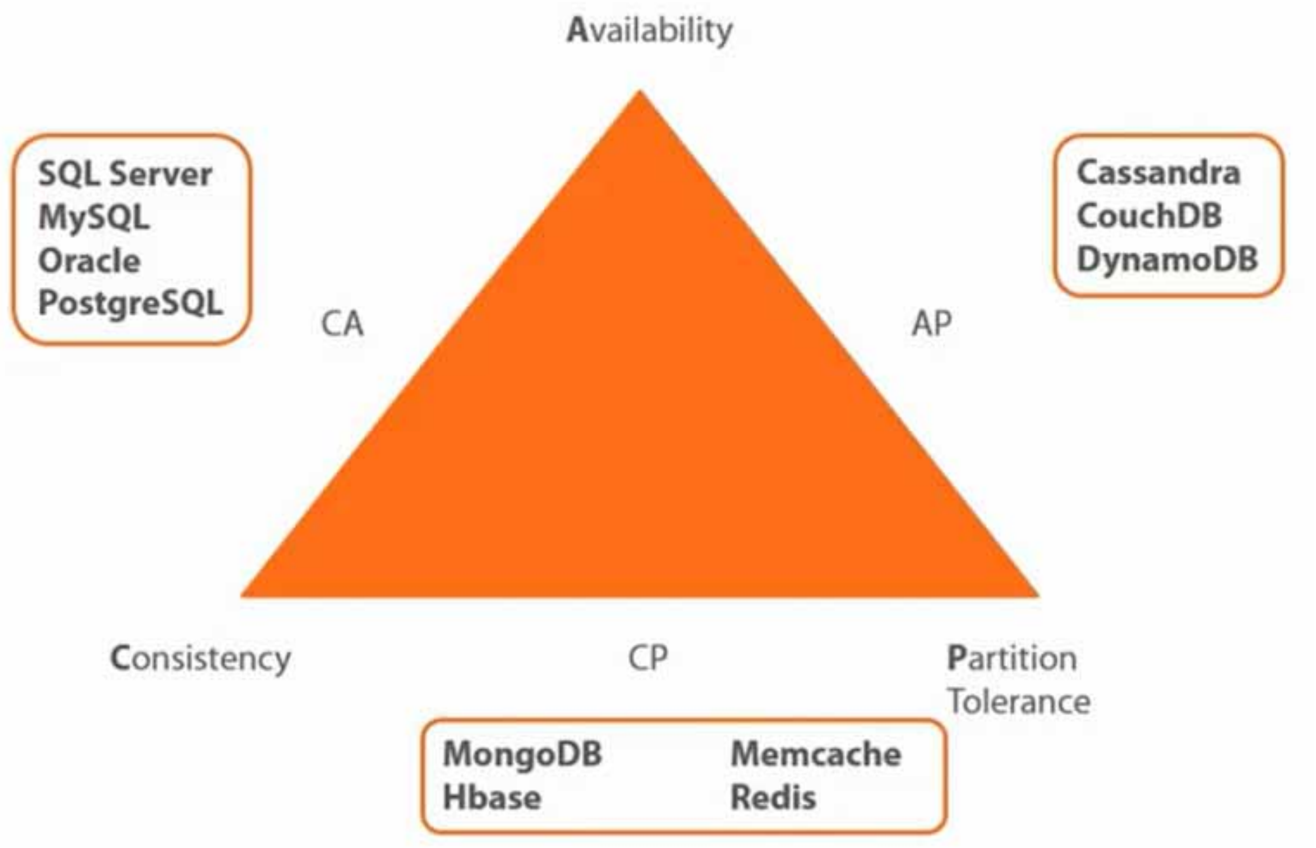
CAP theorem: A distributed computer system cannot guarantee the following three properties *at the same time*:

Consistency	all nodes see the same data at the same time.
Availability	every request receives a response indicating a success or failure result.
Partition tolerance	the system continues to work even if some nodes go down.

- In the presence of a network partition, one must choose between consistency and availability.
  - RDBMs choose consistency over availability
  - NoSQL databases choose availability over consistency.

EVENTUAL CONSISTENCY (cont'd)

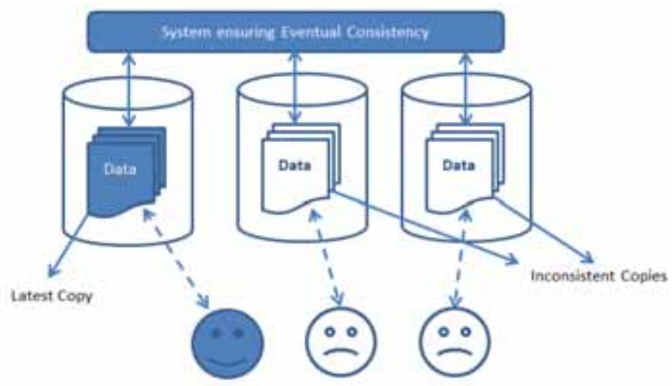
DBMSs and the CAP Theorem



EVENTUAL CONSISTENCY (cont'd)

- NoSQL DBMSs follow the BASE rather than the ACID principle.
  - Basically Available: NoSQL databases adhere to the availability guarantee of the CAP theorem.
  - Soft state: the system can change over time, even without receiving input.
  - Eventual consistency: the system will become consistent over time.

- The request coordination membership protocol does not guarantee that every node is always aware of every other node.
  - The database will reach a consistent state over time (e.g., via continuous pairwise communications).
  - The state of the database might not be perfectly consistent at any moment in time, though it will become eventually consistent at a future point in time.



KEY-VALUE STORE LIMITATIONS

- Key-value stores represent a very diverse range of systems:
  - Full-blown DBMSs versus cache Layered Relational DBMSs.
- Only limited query facilities are offered.
  - E.g., *put* and *get*.
- No means to enforce structural constraints.
  - The DBMS remains agnostic to the internal structure of the data.
- No relationships, referential integrity constraints or database schema can be defined over key-value stores.

TUPLE STORES

- A tuple store is similar to a key-value store, but rather than storing a key and a value, it stores a unique key together with a vector of data (i.e., a tuple).

Example: marc ⟶ ("Marc", "McLaine", 25, "Germany")

- There is no requirement to have the same length or semantic ordering for data.

☞ **The data is completely schema-less.**

- Various NoSQL implementations do permit organizing entries in semantical groups (similar to collections or tables).

Example:

Person: marc ⟶ ("Marc", "McLaine", 25, "Germany")
Person: harry ⟶ ("Harry", "Smith", 29, "Belgium")

Book: harry ⟶ ("Harry Potter", "J.K. Rowling")

DOCUMENT STORES

- Document stores store a collection of attributes that are labeled and unordered, representing items that are semi-structured (similar to XML data).

Example:

```
{
    Title = "Harry Potter"
    ISBN = "111-1111111111"
    Authors = [ "J.K. Rowling" ]
    Price = 32
    Dimensions = "8.5 x 11.0 x 0.5"
    PageCount = 234
    Genre = "Fantasy"
}
```

  – Documents are often represented using JSON (JavaScript Object Notation), BSON (Binary JSON), YAML (YAML Ain't Markup Language) or XML.

ITEMS WITH KEYS

- Most NoSQL document stores will allow items to be stored in tables (collections) in a schema-less manner but will enforce that a primary key be specified.
  - MongoDB uses _id as the primary key attribute name in an item.
  - The primary key value can be user-defined or auto-generated.
- A primary key will be used as a partitioning key to create a hash and determine where the data will be stored.

MongoDB Example:

```
{
    "_id": ObjectId("56349fh4ht49jv4j9jv4jvj94jv49"),
    "title": "Harry Potter",
    "price": 32
}
```

FILTERS AND QUERIES

- Just like key-value stores, the key of an item can be used to retrieve it.
- However, since items have multiple attributes, items can also be retrieved based on simple filters.
- MapReduce can be used for complex queries and aggregation queries.

EXAMPLE FILTERS AND QUERIES

This Java code shows how to connect to a MongoDB instance, insert some documents, query, and update them.

```
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import static com.mongodb.client.model.Filters.*;
import static java.util.Arrays.asList;

public class MongoDBExample {
   public static void main(String... args) {
      MongoClient mongoClient = new MongoClient();
      MongoDatabase db = mongoClient.getDatabase("test");

      // Delete all books first
      db.getCollection("books").deleteMany(new Document());

     // Add some books
      db.getCollection("books").insertMany(new ArrayList<Document>() {{
      add(getBookDocument("My First Book", "Wilfried", "Lemahieu", 12, new String[]{"drama"}));
      add(getBookDocument("My Second Book", "Seppe", “vanden Broucke", 437, new String[]{"fantasy", "thriller"}));
      add(getBookDocument("My Third Book", "Seppe", “vanden Broucke", 200
      add(getBookDocument("Java Programming", "Bart", "Baesens", 1
      }});
```

Set up a MongoClient and MongoDatabase

Use the deleteMany and InsertMany methods to delete and insert book entries

Use the getBookDocument helper method to get title, author, nrPages and genres

EXAMPLE FILTERS AND QUERIES (cont'd)

```java
// Perform query
FindIterable<Document> result = db.getCollection("books").find(
                              and( eq("author.last_name", “vanden Broucke"),
                                   eq("genres", "thriller"),
                                   gt("nrPages", 100)));
for (Document r : result) {
System.out.println(r.toString());
// Increase the number of pages:
db.getCollection("books").updateOne(
new Document("_id", r.get("_id")),
new Document("$set",
new Document("nrPages", r.getInteger("nrPages") + 100)));
}
 mongoClient.close();}
public static Document getBookDocument(String title,
    String authorFirst, String authorLast,
    int nrPages, String[] genres) {
return new Document("author", new Document()
                                   .append("first_name", authorFirst)
                                   .append("last_name", authorLast))
                                   .append("title", title)
                                   .append("nrPages", nrPages)
                                   .append("genres", asList(genres));}}
```

Use the “find” method and pass a conjunctive (and) condition

Update the pages in the result set

The helper method “getBookDocument”

FILTERS AND QUERIES (CONT'D)

- Queries with many criteria, sorting operations or aggregations can be slow because every filter (such as “author.last_name = Baesens”) requires a complete collection or table scan.

- Most document stores can define a variety of indexes (similar to relational indexes).
  - unique and non-unique indexes
  - compound indexes
  - geospatial indexes
  - text-based indexes

SQL AFTER ALL …

- **group by**-style SQL queries are convertible to an equivalent map-reduce pipeline.
- Consequently, many document store implementations express queries using an SQL interface.
- Many RDBMS vendors are implementing NoSQL databases by:
  - focusing on horizontal scalability and distributed querying.
  - dropping schema requirements.
  - supporting nested data types or allowing storing JSON directly in tables.
  - supporting map-reduce operations.
  - supporting special data types, such as geospatial data.

COLUMN-ORIENTED DATABASES

- A column-oriented DBMS stores data tables as sections of columns of data rather than as rows of data.

- Useful if:
  - Aggregates are regularly computed over large numbers of similar data items.
  - Data are sparse, i.e., columns with many null values.

- Can also be implemented by an RDBMS, key-value store or document store.

COLUMN-ORIENTED DATABASES

Id	Genre	Title	Price	Audiobook price
1	fantasy	My first book	20	30
2	education	Beginners guide	10	null
3	education	SQL strikes back	40	null
4	fantasy	The rise of SQL	10	null

Query: Find books with Price=10.

- Row-oriented databases need indexes to efficiently answer this query, which adds overhead when frequent updates are done.
- Column-oriented databases can directly process this query.
  - Null values do not take up storage space (e.g., Audiobook price has only one book id, 1).

Genre:	fantasy:1,4	education:2,3		
Title:	My first book:1	Beginners guide:2	SQL strikes back:3	The rise of SQL:4
Price:	20:1	10:2,4	40:3	
Audiobook price:	30:1			

COLUMN-ORIENTED DATABASES

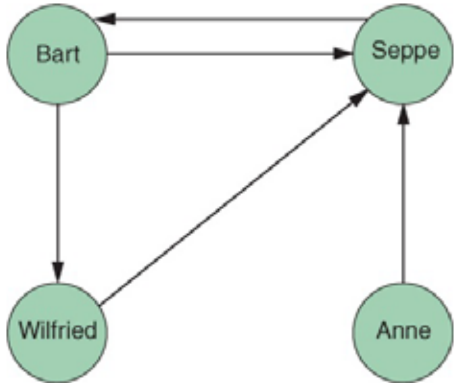
- Disadvantages
  - Retrieving all attributes pertaining to a single entity becomes less efficient.
  - Join operations will be slowed down.

Examples

- Google *BigTable*, *Cassandra*, *HBase* and *Parquet*.

Genre:	fantasy:1,4	education:2,3		
Title:	My first book:1	Beginners guide:2	SQL strikes back:3	The rise of SQL:4
Price:	20:1	10:2,4	40:3	
Audiobook price:	30:1			

GRAPH DATABASES



- A graph database stores information as nodes and edges.
- Relationship tables are replaced by more interesting and semantically meaningful relationships that can be navigated and/or queried using graph traversal based on graph pattern matching.
- One-to-one, one-to-many, and many-to-many structures can easily be modeled in a graph.
- Applications
  - Location-based services
  - Recommender systems
  - Social media (e.g., Twitter and FlockDB)
  - Knowledge-based systems

GRAPH DATABASES

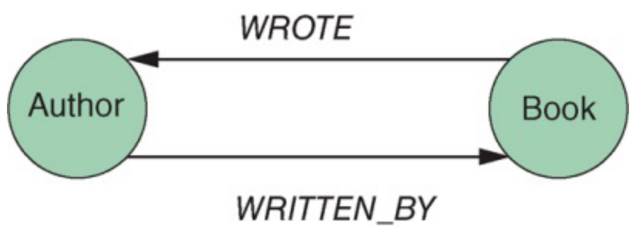
- For an N:M relationship between books and authors, an RDBMS needs three tables: Book, Author and BookAuthor to represent the relationship.

- An SQL query to return all book titles for books written by a particular author would look like:

```
select title
from Book, Author, BookAuthor
where Author.id=BookAuthor.author_id
      and Book.id=BookAuthor.book_id
      and Author.name='Bart Baesens';
```

- In a graph database, the schema would be the graph on the right and the query would look like (using the Cypher query language):

```
match (b:Book)←[:WRITTEN_BY]-(a:Author)
where a.name = "Bart Baesens"
return b.title
```



NOSQL DATABASES: SUMMARY

	Relational Databases	NoSQL Databases
Data paradigm	Relational tables	Key-value (tuple) based Document based Column based Graph based Others: XML, object based, time series, probabilistic, etc.
Distribution	Single-node and distributed	Mainly distributed
Scalability	Vertical scaling, harder to scale horizontally	Easy to scale horizontally, easy data replication
Openness	Closed and open source	Mainly open source
Schema role	Schema-driven	Mainly schema-free or flexible schema
Query language	SQL	No or simple querying facilities, or special-purpose languages
Transaction mechanism	ACID: Atomicity, Consistency, Isolation, Durability	BASE: Basically Available, Soft state, Eventual consistency
Feature set	Many features (triggers, views, stored procedures, etc.)	Simple API
Data volume	Capable of handling normal-sized datasets	Capable of handling huge amounts of data and/or very high frequencies of read/write requests