COMP207
Database Development

Lecture 24

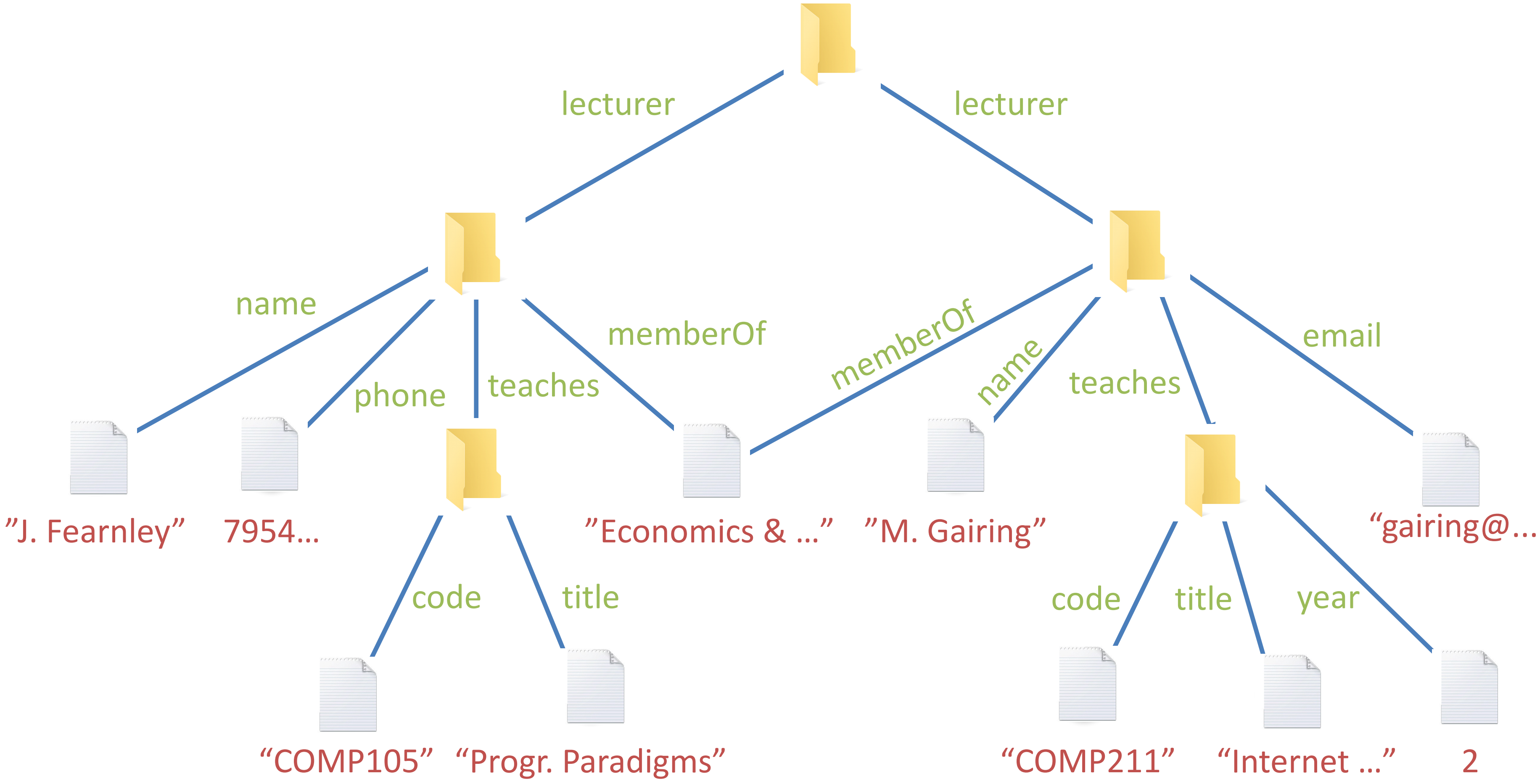
Beyond Relational Data:
Inputting XML into relation databases and
NoSQL Databases

What you should know at the end
(Learning Outcomes)

- Transaction management:
  - Identification & application of the principles underpinning transaction management within DBMS
- Advanced SQL:
  - SQL from COMP102 extended with indexes, transactions, query optimisation
  - Application in problem solving
- Object-relational models:
  - Identification of principles
- Web technologies:
  - Illustrate issues related to web technologies as a semi-structured data representation formalism
- Data warehouses and data mining:
  - Interpret the main concepts and security aspects in data warehousing
  - Interpret the main concepts of data mining

XML

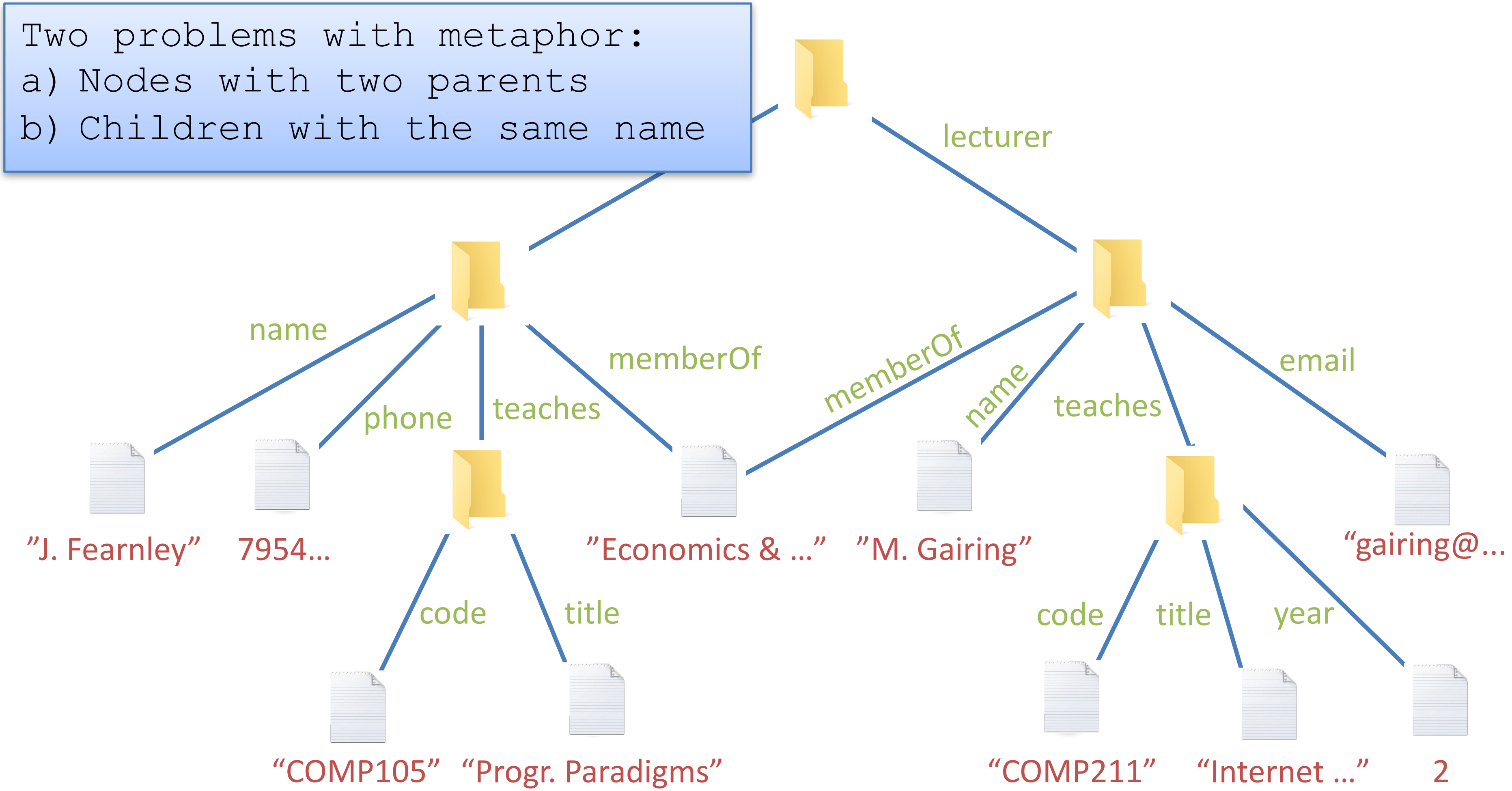
(from looooong ago)



XML

(from looooong ago)

```
Two problems with metaphor:
a) Nodes with two parents
b) Children with the same name
```



Relational database model

(from even longer ago)

- Data in the relation model is **structured**
  - Data **has to fit to schema**
  - Advantage: highly efficient query processing

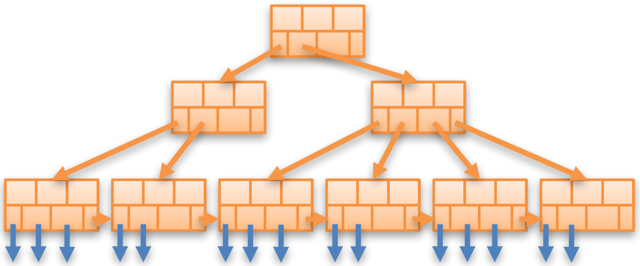
Student(name, number, programme)
Module(code, name)

Module

code	name
COMP207	Database Development
COMP219	Artificial Intelligence
…	…

Student

name	numb	
Anna	20171	
John	20174378	G702
…	…	…



XML and Relational Databases

- Four general approaches to storing XML documents in a relational database:
  - Store XML documents as entries of a table
  - Store XML documents in shredded form across a number of attributes and relations
  - Store XML documents in schema-independent form
  - Store XML documents in parsed form

Storing XML in an Attribute

- In past the XML would have been stored in an attribute whose data type was CLOB.
- More recently, some systems have a new native XML data type (e.g. XML or XMLType).
- Raw XML stored in serialised form, which makes it efficient to insert documents into database and retrieve them in their original form.
- Relatively easy to apply full-text indexing to documents for contextual and relevance retrieval. However, question about performance of general queries and indexing, which may require parsing on-the-fly.
- Also, updates usually require entire XML document to be replaced with a new document.

Creating Table using XML Type

```
CREATE TABLE XMLStaff (
        docNo CHAR(4), docDate DATE, staffData XML,
        PRIMARY KEY docNo);

INSERT INTO XMLStaff VALUES (‘D001’, DATE‘2004-12-01’,
        XML(‘<STAFF branchNo = "B005">
                <STAFFNO>SL21</STAFFNO>
                <POSITION>Manager</POSITION>
                <DOB>1945-10-01</DOB>
                <SALARY>30000</SALARY> </STAFF>’) );
```

Storing XML in Shredded Form

- XML decomposed (shredded) into its constituent elements and data distributed over number of attributes in one or more relations.
- Storing shredded documents may make it easier to index values of some elements, provided these elements are placed into their own attributes.
- Also possible to add some additional data relating to hierarchical nature of the XML, making it possible to recompose original structure and ordering, and to allow the XML to be updated.
- With this approach also have to create an appropriate database structure.

XML

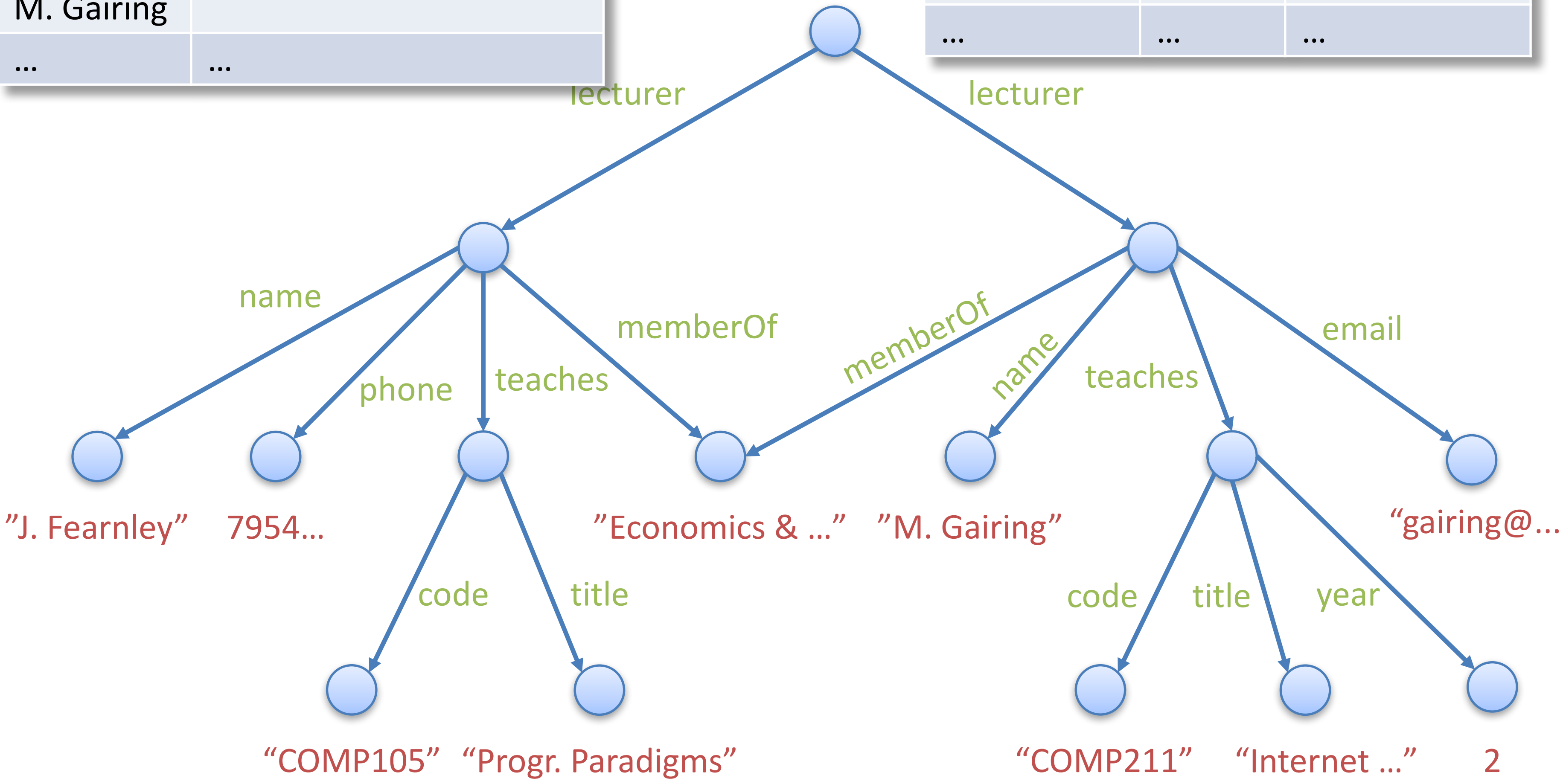
rom two times a

Lecturer

name	phone
J. Fearnley	7954...
M. Gairing	
…	…

teaches

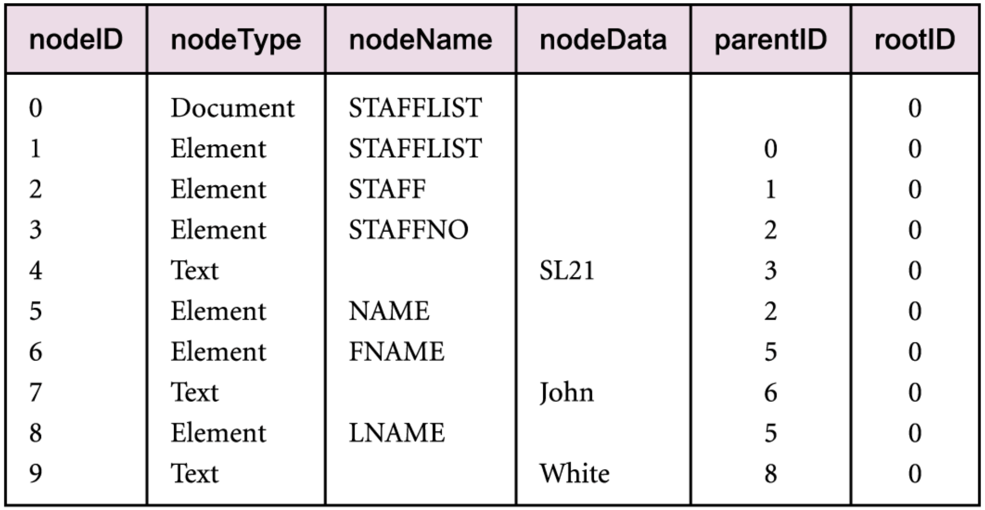
name	code	title
J. Fearnley	105	Progr. Para...
M. Gairing	211	Internet...
…	…	…



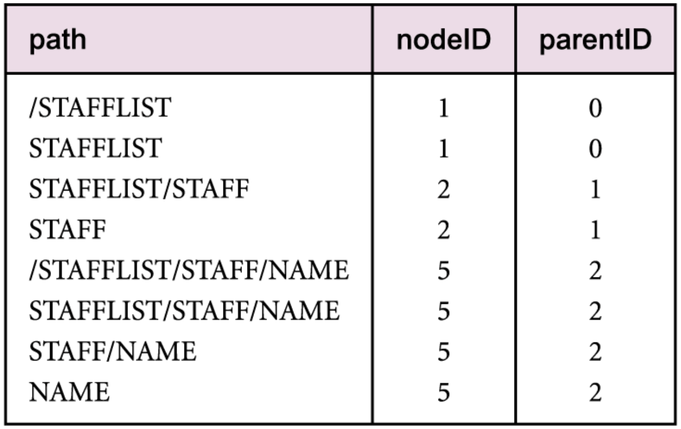
Schema-Independent Representation

- Could use DOM to represent structure of XML data.
- Since XML is a tree structure, each node may have only one parent. The rootID attribute allows a query on a particular node to be linked back to its document node.
- While this is schema independent, recursive nature of structure can cause performance problems when searching for specific paths.
- To overcome this, create denormalized index containing combinations of path expressions and a link to node and parent node.

Schema-Independent Representation



nodeID	nodeType	nodeName	nodeData	parentID	rootID
0	Document	STAFFLIST			0
1	Element	STAFFLIST		0	0
2	Element	STAFF		1	0
3	Element	STAFFNO		2	0
4	Text		SL21	3	0
5	Element	NAME		2	0
6	Element	FNAME		5	0
7	Text		John	6	0
8	Element	LNAME		5	0
9	Text		White	8	0



path	nodeID	parentID
/STAFFLIST	1	0
STAFFLIST	1	0
STAFFLIST/STAFF	2	1
STAFF	2	1
/STAFFLIST/STAFF/NAME	5	2
STAFFLIST/STAFF/NAME	5	2
STAFF/NAME	5	2
NAME	5	2

NoSQL Databases

NoSQL Databases

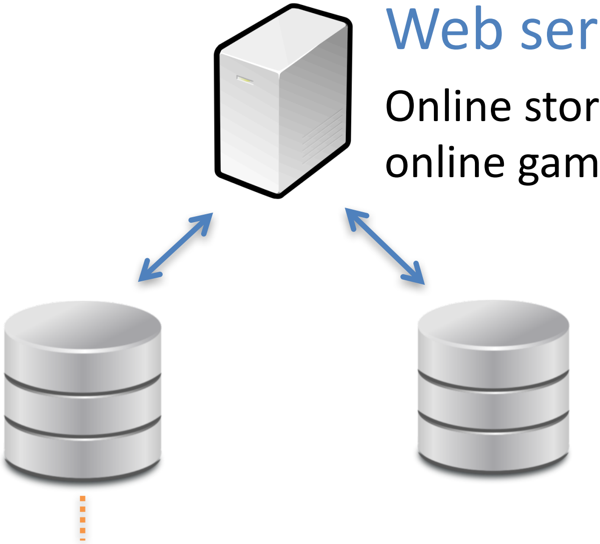
- NoSQL: “Not Only SQL” or “Not Relational”

- Cater to requirements for typical web applications
  - Very fast access to data, with millions of users in parallel
  - Fault-tolerance
  - Flexibility in the type of data stored (semi-structured)
  - Full ACID compliance can sometimes be relaxed

- Not restricted to such applications
  - E.g., popular for big data analytics projects

- Not a replacement for relational databases

Example: Managing User Data



Web service

Online store, massive multi-player online gaming platform, …

NoSQL Database

User profiles, shopping cart, messages, orders, etc.

Relational Database

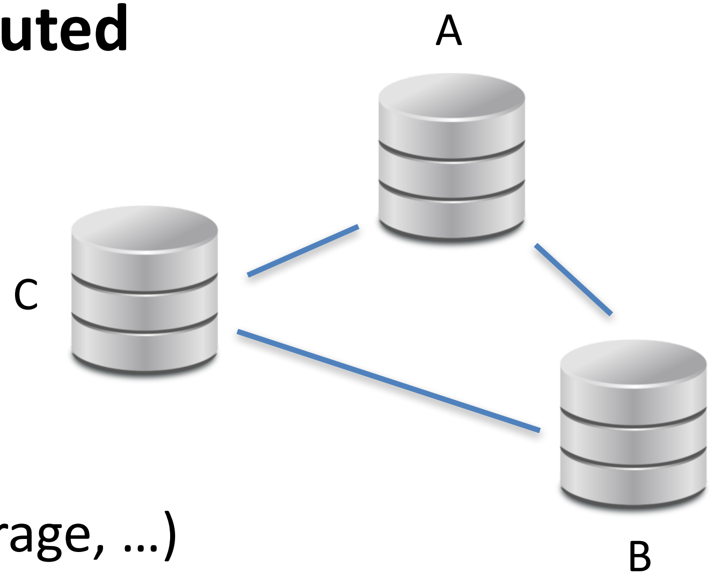
Credit card transactions, inventory, other data

Key	Value
User 1	{ “name”:“John Smith”, “items”:{ “1”:{“name”:“product 1”, “quantity”:3}, “2”:{“name”:“product 2”, “quantity”:1}, … … }
…	…

- Simple queries: return the value for key k
- Fast reads/writes via index on keys
- ACID can sometimes be relaxed

NoSQL Database Characteristics

- NoSQL databases are often **distributed**
  - Nodes run on commodity hardware
  - Standard network



- Designed to guarantee:
  - **Availability**: every non-failing node always executes queries
  - **Scalability**: more capacity (users, storage, …) by adding new nodes
  - **High performance**: often achieved by very simple interface (e.g., support lookups/inserts of keys, but do this fast)
  - **Fault-tolerance** (a.k.a. **partition-tolerance**): even if nodes fail, the remaining subnetworks can continue their work

- Often give up ACID to improve performance

NoSQL Database Classification

- Common Classification:
  - Key-value stores: ⬅ today
    - Efficient lookups and insertions of key-value pairs
  - Document stores:
    - Similar to key-value stores (values = semi-structured data, represented e.g. in XML/JSON)
    - Lookups can involve values from documents
  - Column stores
  - Graph databases
- Hybrids and other types possible
- No type fits all purposes

Key-Value Stores

Key-Value Stores

- Collection of table, tables = key-value pairs

Key	Value
User 1	{ “name”:“John Smith”, “items”:{ “1”:{“name”:“product 1”, “quantity”:3}, “2”:{“name”:“product 2”, “quantity”:1}, … … }
…	…

Essentially an index!

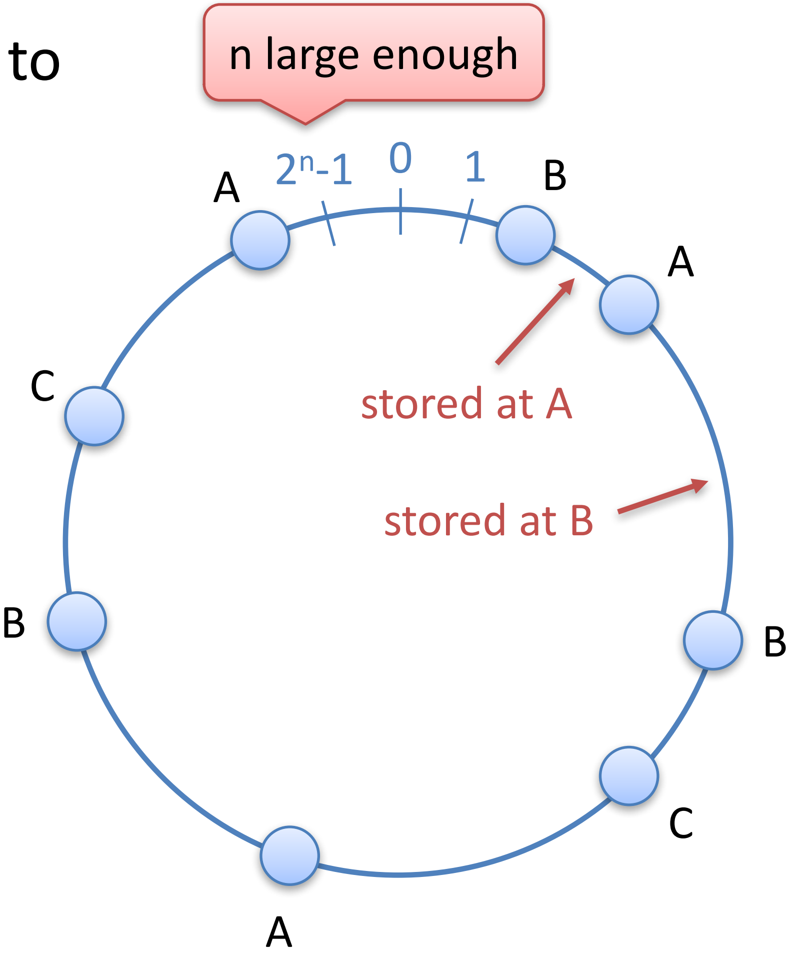
- Simple access mechanism:
  - **find(k)**: returns value for key **k**
  - **write(k, v)**: inserts value **v** under key **k**
- Fast due to index on key, no further indexes & no transactions

Available Systems

- Apache Cassandra
  https://cassandra.apache.org
- Amazon DynamoDB
  https://aws.amazon.com/dynamodb/
- Apache Voldemort
  http://www.project-voldemort.com/voldemort/
- Memcached
  http://memcached.org
- Redis
  https://redis.io
- Riak
  http://basho.com/products/#riak
- Many others…

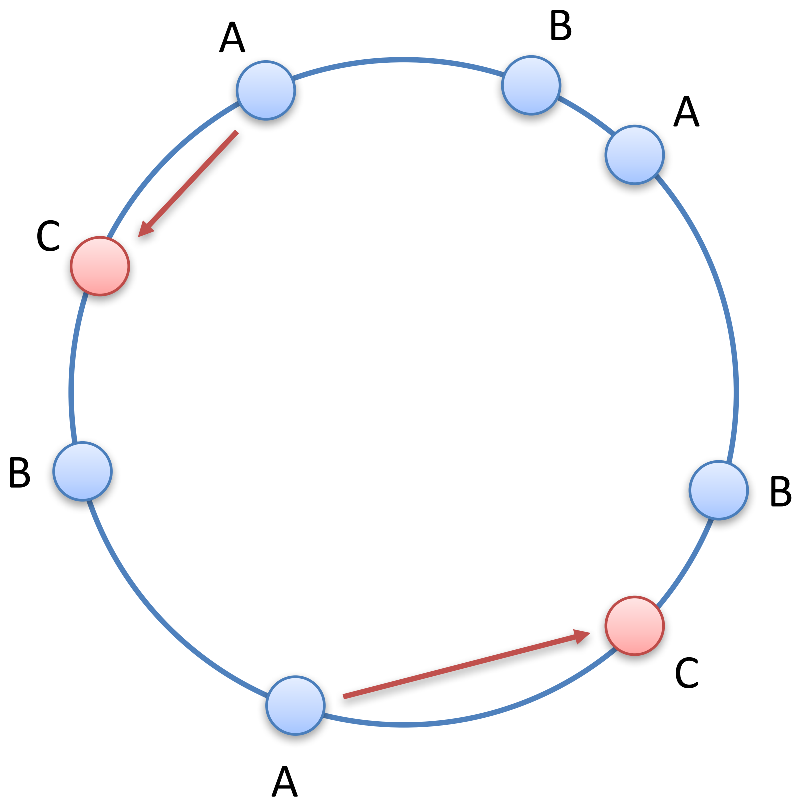
Distributed Storage

- Each key-value pair (**k**,**v**) is stored at some node
- **Step 1:** Assign values **v** for key **k** to integer between 0 and 2^n-1
  - Uses a hash function $h\colon \text{keys} \rightarrow \{0, \ldots, 2^n - 1\}$
- **Step 2:** Distribute nodes to some of the integers (typically randomly)
- If (**k**,**v**) is assigned to integer ***i***, it is stored at the node following ***i*** on the ring



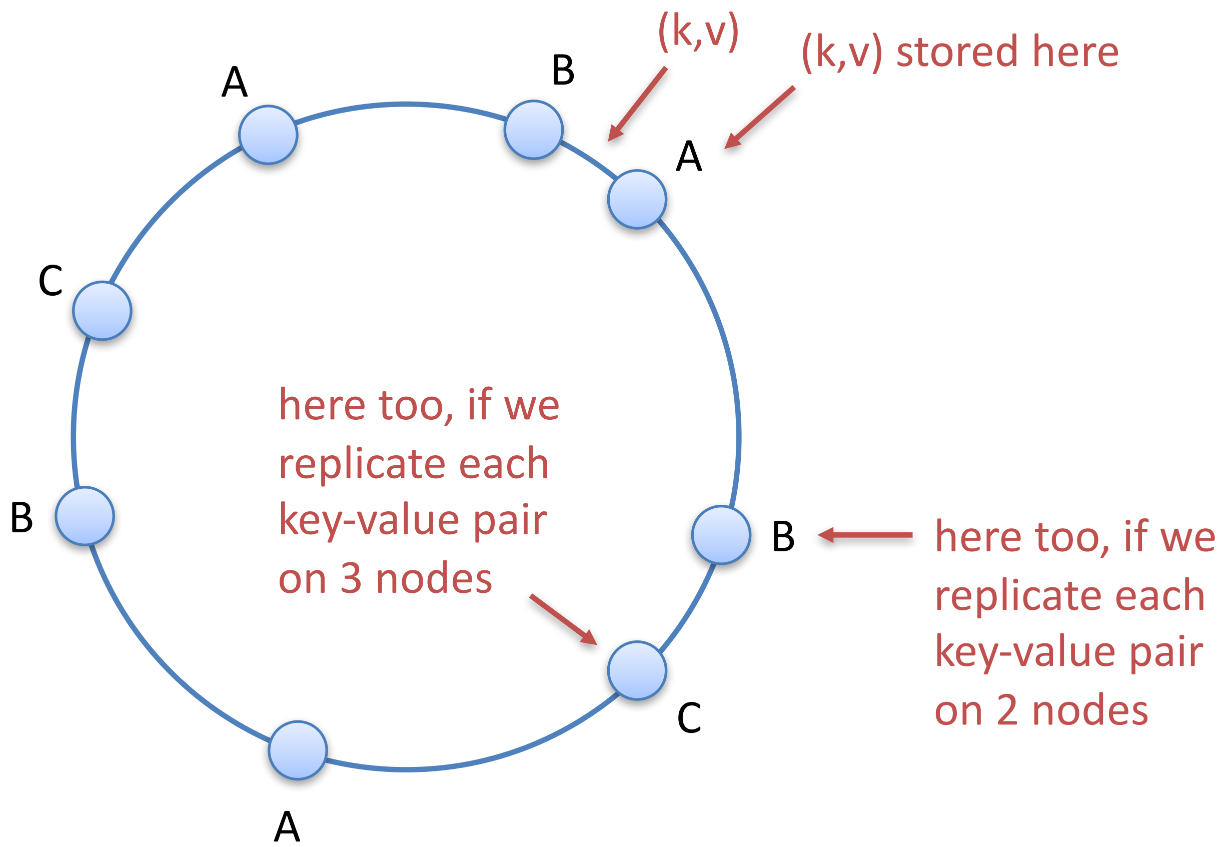
Scalability Via Horizontal Fragmentation

- New nodes can be added easily:
  - Add node to free range(s) and move key-value pairs appropriately
  - Automatic **horizontal fragmentation**



Replication

- Replication is used to ensure availability
- Replicas (copies of key-value pairs) are stored on consecutive nodes on the ring in clock-wise order


(k,v)
(k,v) stored here
A
B
A
C
B
B
C
A
here too, if we replicate each key-value pair on 3 nodes
here too, if we replicate each key-value pair on 2 nodes

Properties

- **Availability & fault-tolerance**:
  - Due to replication
  - Can retrieve value for a key, even if a few nodes storing values for that key fail
- **Scalability**:
  - Adding new nodes to increase capacity is easy
  - Automatic horizontal fragmentation
- **High performance**:
  - Retrieving the value for a key is fast: apply the hash function to determine the node, then ask the node
  - Writing, too

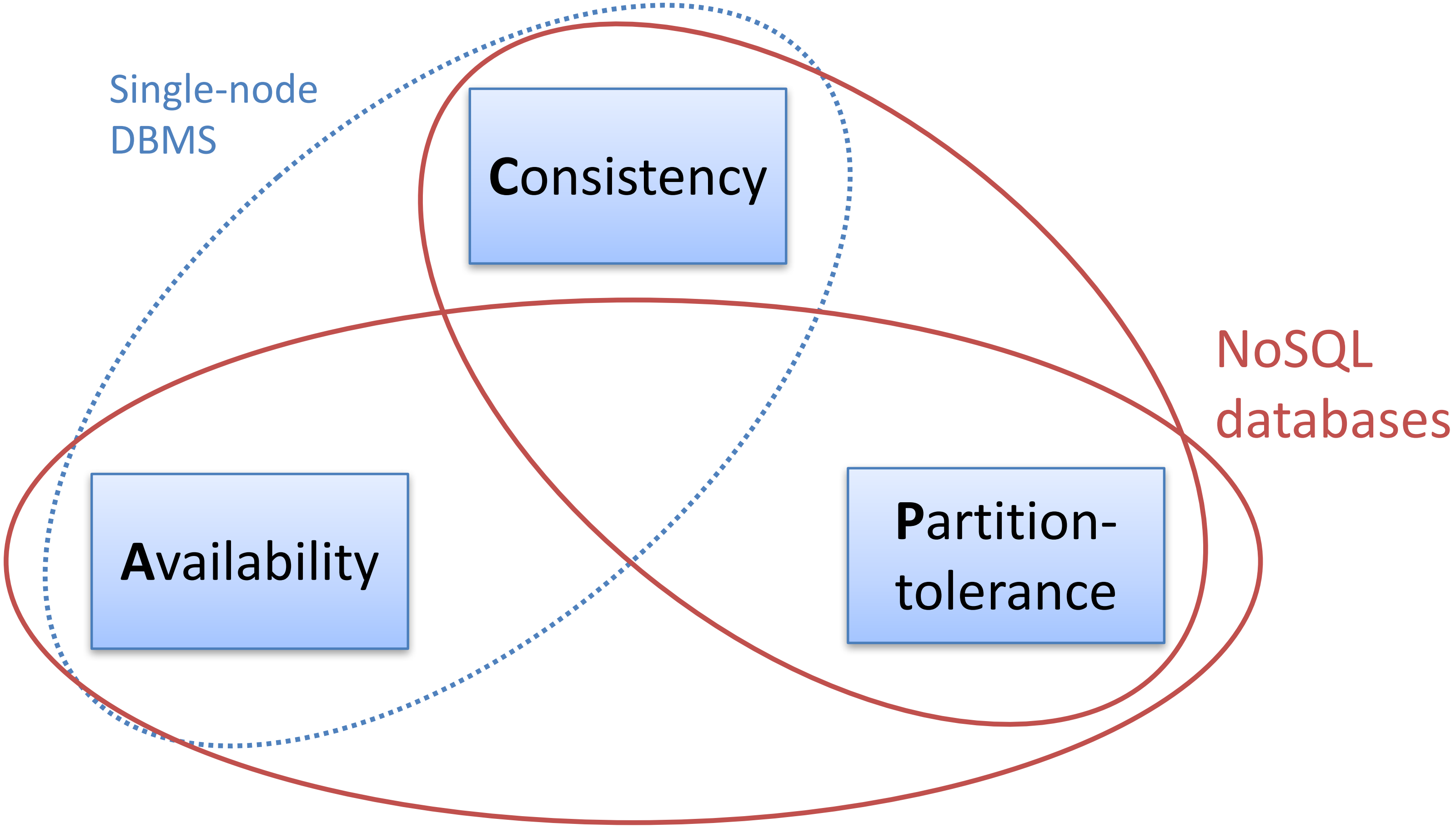
Every read returns the most recent value

- Problem: ensuring **consistency clashes with availability**

The CAP Theorem

Brewer (2000); Gilbert & Lynch (2002)

- We cannot achieve all three of these properties:



From ACID to BASE

- NoSQL databases drop ACID and often provide the weaker guarantees of BASE:

Basically **A**vailable,
Soft state,
Eventually consistent

A data item X has been updated, but the update has not yet been propagated to some of the replicas

- Soft state & eventual consistency:
  - the database state might occasionally be inconsistent…
  - but it will eventually be made consistent

Eventual Consistency

- DynamoDB & Voldemort allow **multiple versions of data item** to be present at the same time (”versioning”)
- If a newer version of a data item is not yet available at a node, the older version is used/updated
  - Fine for many applications (e.g., think of a shopping cart)
- Method: assign a **vector clock** to each version of an item X
  - a list/vector of pairs (node, timestamp)

Node that has written X

Local time on the node the item X was written

- Use clock to decide if version V_1 originated from version V_2
  - V_1 originated from V_2 if for all nodes in V_2’s clock the corresponding timestamp is less than or equal to the timestamp in V_1’s clock
- **Conflicts between versions are resolved at read-time**

Summary

- XML documents can be input into a relational database in four ways: As an attribute, shredded, schema independent or parsed
- NoSQL databases are the newest members to the data management family
- Main use cases:
  - Provide simple access to data in contexts where but speed, availability, and scalability is more important, and ACID can be traded for more performance
  - Management of large collections of semistructured data (JSON, XML, general graphs, key-value pairs)
- Not a replacement for relational databases, but a welcome complement