

Institute of
Data



2020



Data Science and AI

Module 2
Part 1:

SQL and Databases



Agenda: Module 2 Part 1

- Introduction to Databases
- The relational database paradigm
- Basic SQL
- RDBMS
- Advanced SQL
- SQL in Python
- NoSQL Databases



Introduction to Databases

- Databases: definition, usage, features, applications
- Database Elements
- Database Principles
- The relational database paradigm
- SQL
- RDBMS



Introduction to Databases

- What is a database?
 - a computer system that manages **storage** and **querying** of data
- How is a database used?
 - data **insertion** & **retrieval** are (typically) performed using a **query language**
 - a compact programming syntax
 - basic operators for data transformation
- What are the essential features of a database?
 - practical design for organising data
 - **efficient** methods to retrieve specific information
 - indexing, **performance optimisation**
 - **reliability, security, backup & replication**



Why Use a Database?

- the **standard** solution for data storage
- much more **robust** than text, CSV or JSON files
- most analyses involve pulling data to and from a resource; in most settings, this means using a **database**
- many types and variants to serve different use cases
- rules on structure make writing and retrieving data more **reliable and efficient**
- provide a **central source of “truth”**



Database Application Areas - Examples

- Operations
 - transaction systems
 - data capture
 - inventory management
- Data Warehouse
 - Reporting
 - Analytics
 - Data Science
- Master Data
 - products
 - customers
 - suppliers



Database Elements

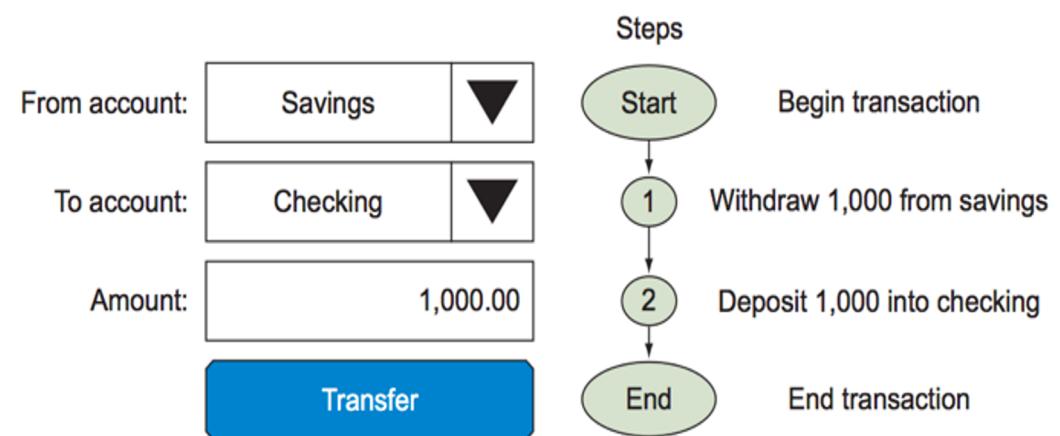
- tables
 - data storage by **columns** (attributes) and **rows** (records)
- keys
 - for matching **indexed** attributes across tables
- queries, views
 - for retrieving, subsetting, aggregating, joining data
- functions, procedures
 - **reusable** code units
- types
 - reusable data structures
- triggers
 - procedures that run automatically when a specific **event** occurs
- jobs
 - batches of procedures that run on a **schedule**



Database Principles: Transactional Integrity

def: Transaction

- a unit of work performed against a database
- this term generally represents any change in database
- involves **multiple steps**





Database Principles: ACID

def: ACID is a set of properties that guarantee that database transactions are processed reliably



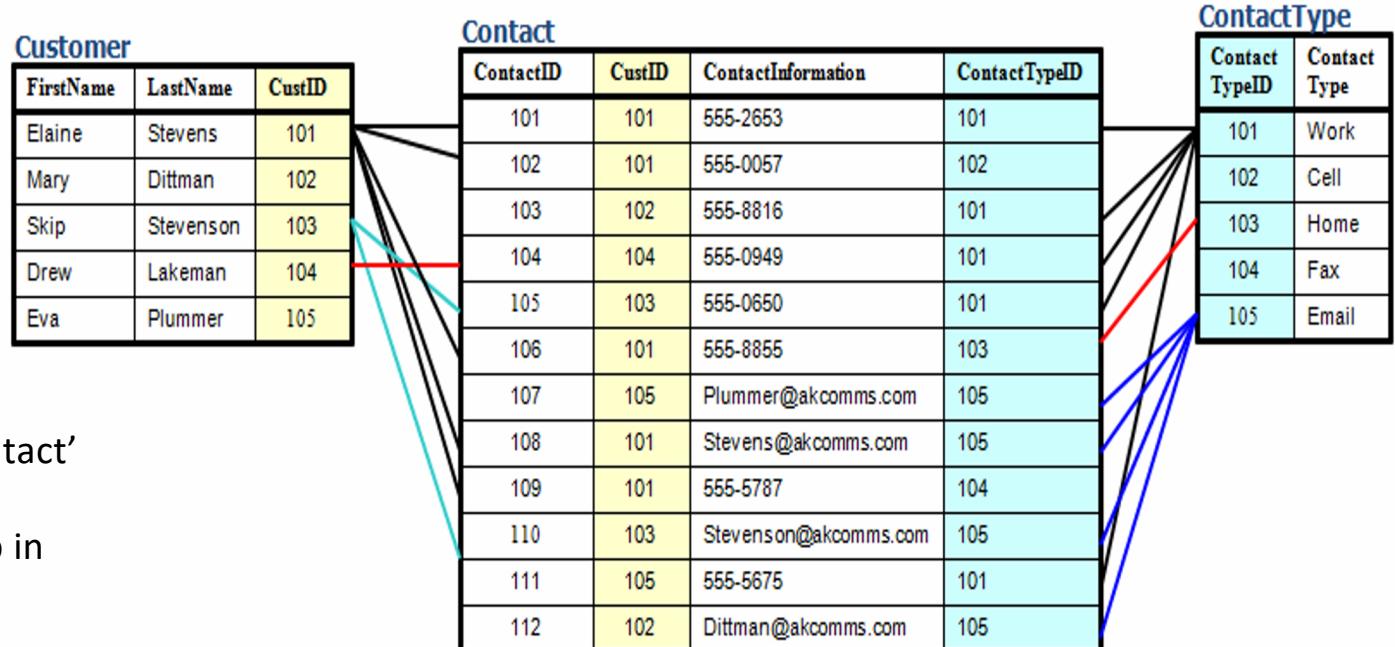
- **Atomicity:** if one part of the transaction fails, the entire transaction fails;
the database state is left unchanged ('all or nothing')
- **Consistency:** ensures that any transaction will bring the database from one **valid state** to another
- **Isolation:** ensures that the **concurrent** execution of transactions results in a system state that would be obtained if transactions were executed serially (one after the other)
- **Durability:** ensures that once a transaction has been committed it will be unaffected by **power loss, system crashes, or errors**



The relational Database Paradigm

- each table in a database is devoted to one **domain**
- keys are used to **connect** tables in a logical manner

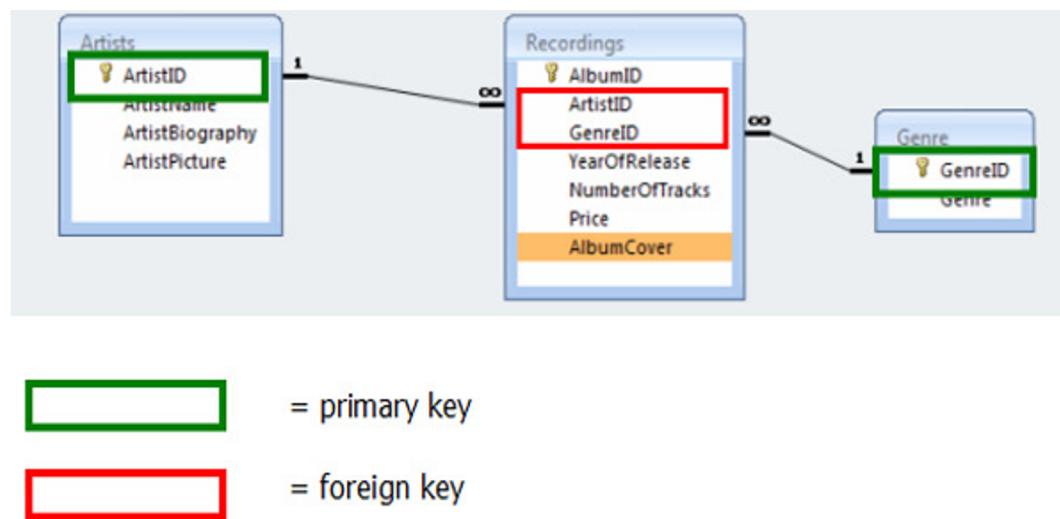
- customer info in ‘Customer’ table
- contacts info in ‘Contact’ table
- contact type lookup in ‘ContactType’ table





Relational Databases

- queries can pull together information from multiple tables by matching **foreign keys** to primary keys



- ‘ArtistID’ is:
 - a foreign key in the ‘Recordings’ table
 - the primary key in the ‘Artists’ table
- ‘GenreID’ is:
 - a foreign key in the ‘Recordings’ table
 - the primary key in the ‘Genre’ table



RDBMS

- RDB = ?
 - relational database
- RDBMS = ?
 - relational database management system





Database Schema

- table-level
 - columns and primary key
- database-level
 - overall design
 - data model (tables)
 - keys (PK, FK)
 - integrity constraints
- rationale
 - removes tight coupling of database objects and owners
 - improves security administration of database objects



SQL (Structured Query Language)

- essentially a **declarative** language
 - data are typically retrieved as **subsets** of the rows and columns in one or more tables: 'rowsets'
 - the user encodes the kind of rowset to be returned
 - the database engine produces a **plan** for how to execute it
 - **optimised** if possible
 - may accept **hints** from user (e.g. indexes to use)
- 3 functional divisions:
 - data definition language (**DDL**)
 - schema creation and modification
 - data manipulation language (**DML**)
 - insert, update, delete
 - data control language (**DCL**)
 - access, security



SQL – cont'd

simple rowset retrieval:

SELECT *columns* **FROM** *table*

conditional rowset retrieval:

SELECT *columns* **FROM** *table* **WHERE** *condition*

example (conditional rowset with ordering):

SELECT TOP 10 amount FROM sales WHERE amount > 99.99 ORDER BY amount



SQL – cont'd

Syntax

- identifiers with embedded **spaces or punctuation** require implementation-specific delimiters
 - e.g. SQL Server allows “my table” or [my table]
(delimiters are optional, otherwise)
- **namespace hierarchies** are usually delimited with periods (full stops)
 - e.g. (SQL Server): myschema.mytable.mycolumn



SQL – cont'd

Syntax

- SQL is **case-insensitive**
 - best to adopt a style, for readability
(e.g. uppercase for SQL keywords, lowercase for identifiers)
- Microsoft and others extend the ANSI standard
 - extensions add **power and convenience** to programming
 - many extensions do not readily **port** between versions of SQL



Indices (*aka* ‘Indexes’)

- An index is a column of unique numbers (one per row) used to speed up query performance
 - reduces number of database data pages that have to be **scanned**
 - can have **> 1** index per table
 - usually based on single columns or **tuples** of columns
- a **clustered index** (e.g. SQL Server) determines physical order of data in a table
 - can only have 1 clustered index per table
- Building / rebuilding an index is an **expensive operation**
 - for inserting a large number of rows, it is usually best to **drop the index**, do the insert, then rebuild the index



Primary and Foreign Keys

- ***primary key*** uniquely refers to each row in a table
 - can have only one per table
 - usually an integer
- ***foreign key*** refers to a row in a different table
 - is a primary key in the other table
 - can have any number per table
- Nb. the use of foreign keys reduces storage and makes database maintenance much easier



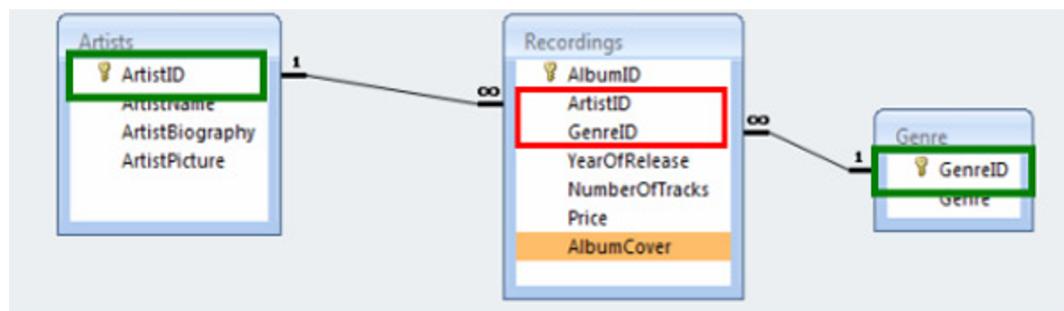
Joins

- used (in queries) for
 - combining columns from different tables
 - filtering columns from one table using criteria in a different table
- joins match a foreign key in one table to a primary key in another table
- Nb. simple joins are generally quite fast, but **compound joins** (involving many tables) can be very slow
- a database join is analogous to a set operation in mathematics



Joins – cont'd

- Compound inner join example



= primary key



= foreign key

Return artist name and album cover for every album of the 'rock' genre:

```
SELECT Artists.ArtistName,
Recordings.AlbumCover
FROM Artists
INNER JOIN Recordings
ON Artists.ArtistID = Recordings.ArtistID
INNER JOIN Genre
ON Recordings.GenreID = Genre.GenreID
WHERE Genre.Genre = 'rock'
```



Joins – cont'd

Left Join

- all rows from 1st table, plus matching rows from 2nd table:

SELECT cars.car, trucks.truck FROM cars

LEFT JOIN trucks ON cars.maker = trucks.maker

- unmatched rows will have NULL in place of truck

Right Join

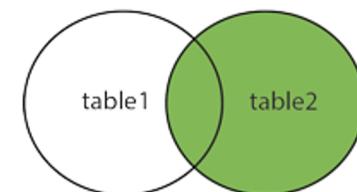
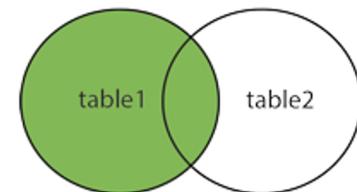
- all rows from 2nd table, plus matching rows from 1st table:

SELECT cars.car, trucks.truck FROM cars

RIGHT JOIN trucks ON cars.maker = trucks.maker

- unmatched rows will have NULL in place of car

Table 1	Table 2
aaa	aaa
bbb	xxx
ccc	ccc
ddd	yyy
eee	eee
fff	fff





Joins – cont'd

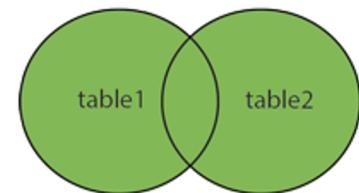
Outer Join

- all rows from 1st table, matched with all rows from 2nd table:

```
SELECT cars.car, trucks.truck FROM cars
```

```
OUTER JOIN trucks ON cars.maker = trucks.maker
```

- returns every possible pairing of car and truck



- uses:
 - creating contingency tables
 - creating dummy data for testing
 - other?



Database Scripting

- shell scripts / commands
 - for quick / convenient execution of routine tasks
 - **populating / updating a database** from a file
 - **backing up or restoring** a database
 - dumping a database object to a file
 - **executing queries** to deliver rowsets for subsequent analysis
 - **moving data** between databases, data lakes, etc.
- examples

```
psql -U postgres -d database_name -c "SELECT c_defaults FROM user_info  
WHERE c_uid = 'testuser'"
```

```
mysql -h "server-name" -u "root" "-pXXXXXXXXX" "database-name" < "filename.sql"
```



Database Administration

- granting **permissions**
 - users, roles
 - tables, views
- performing **backups & recoveries**
- creating & scheduling **jobs**
- Nb. It is not uncommon for a query to join tables from different databases
 - in SQL Server, the query must be created by the *single* owner of *both* databases
 - a database owner should be a virtual user -- not a particular person's login



Discussion

- Why is the RDBMS still in use?
- QUESTIONS?



SQL in Python

- Python/SQL integration paradigms
- Python with embedded SQL db (SQLite)
- Python with external SQL db (pyodbc)



Python / SQL Integration Paradigms

- **Embedded**
 - an SQL RDBMS is emulated by a library designed to work with Python
 - databases are not normally accessible outside of Python
 - ideal for a self-contained Python app with its own db
- **External**
 - a stand-alone SQL RDBMS resides on a database server that other applications can connect to
 - the db is made accessible to Python via an ODBC driver specific to the RDBMS in use
 - SQL Server, MySQL, DB2, etc.
 - used when the Python app is just a client of a more general-purpose db



Python with Embedded SQL Database

Example: SQLite

```
import sqlite3
connection = sqlite3.connect("company.db")

sql_command = """
CREATE TABLE employee (
staff_number INTEGER PRIMARY KEY,
fname VARCHAR(20),
lname VARCHAR(30),
date_joined DATE);"""
```



Python with External SQL Database

Example: pyodbc

```
import pyodbc
cnxn = pyodbc.connect("DSN=MSSQL-PYTHON")
cursor = cnxn.cursor()
cursor.tables()
rows = cursor.fetchall()
for row in rows:
    print row.table_name
```



Python with pyodbc

- Python uses a cursor to iterate through a returned rowset

```
import pyodbc
import pandas.io.sql as psql
import pandas as pd

cnxstr = "Server=myServerAddress;Database=myDB;User Id=myUsername;Password=myPass;"
cxn = pyodbc.connect(cnxstr)
cursor = cnxn.cursor()
cursor.execute("""SELECT ID, FirstName, LastName FROM mytable""")
rows = cursor.fetchone()
objects_list = []
for row in rows:
    d = collections.OrderedDict()
    d['UserID']= row.ID
    d['FirstName']= row.FirstName
    d['LastName']= row.LastName
cxn.close()
```



Lab 2.1.3: SQL in Python [Section 1 to 2]

- **Purpose:**
 - To investigate SQL implementation in Python
- **Tools and Resources:**
 - SQLite: Install on your laptop
<https://www.sqlite.org/download.html>
 - Jupyter Notebooks
 - Python package sqlite3
- **Materials:**
 - ‘Lab 2.1.3 – Databases.ipynb’
- **Data:**
 - ‘housing-data.csv’





Advanced SQL

- Aggregation functions
- Grouping
- Window functions



SQL Aggregation functions

- aggregate many rows into a single resultant row
- common aggregation functions:
 - COUNT counts all rows meeting criterion
 - SUM sums selected field(s) in all rows meeting criterion
 - AVG averages ...
 - MIN, MAX computes minimum/maximum of ...
- SQL engine may support user-defined aggregation functions
- example:

```
SELECT MIN(sale_date) AS firstdt, MAX(sale_date) AS lastdt,  
SUM(sales) AS netsales FROM sales
```



Aggregating by Groups in SQL

- grouping allows aggregation functions to be applied to subsets based on row-level criteria
- adds GROUP BY clause to select statement
- example:

```
SELECT agent_name, SUM(sales) AS netsales FROM sales  
GROUP BY agent_name
```



SQL Window Functions

- operates on a *set* of rows and return a value for *each* row
 - like aggregation, but performed in relation to current row
- example: running total

```
SELECT duration, SUM(duration)
    OVER (ORDER BY start_time) AS running_total
  FROM bikeshare
```

- OVER clause designates window
- ORDER BY sets sequence of rowset (oldest to newest values of start_time)
- each row shows current duration and sum of all previous values of duration



Lab 2.1.3: SQL in Python [Section 3 to 5]

- **Purpose:**
 - To investigate SQL implementation in Python
- **Tools and Resources:**
 - SQLite: Install on your laptop
<https://www.sqlite.org/download.html>
 - Jupyter Notebooks
 - Python package sqlite3
- **Materials:**
 - ‘Lab 2.1.3 – Databases.ipynb’
- **Data:**
 - ‘housing-data.csv’





Scalable SQL

- massively parallel processing relational database systems (**MPP RDBMS**)
 - (expensive) architecture supports scalability and high performance

TERADATA®

VERTICA
An HP Company

Greenplum

AMAZON
REDSHIFT

- NoSQL databases *with SQL*



NoSQL Databases

- What (and why) are NoSQL databases
- Characteristics of NoSQL databases
- NoSQL database types



What and Why are NoSQL Databases

What

- non-tabular
- non-relational

Why

- scalable
- flexible
- simple

- storage and retrieval of data that is modelled by means other than tabular relations
- not a new idea (1960+)
- supports **distributed** storage and processing
- term ‘NoSQL’ popularised by Facebook, Amazon, Google, etc.



Characteristics of NoSQL Databases

Advantages

- avoid administration & expense of RDBMS
- small footprint
- easy to modify schemas



Use Cases

- simple data requirements
- application-specific data
- start-ups





Characteristics of NoSQL Databases – cont'd

Disadvantages

- many don't support true ACID transactions
 - application code is obliged to try to manage concurrency issues
- easy to modify schemas
 - application developers need to collaborate closely to ensure schema development is under control
- much slower than RDBMS
- lack powerful management & development features of RDBMS



NoSQL Databases Types

- wide column store
 - > Hadoop / HBase, MapR, BigTable, Hortonworks, Cloudera, Cassandra, Informix
- document store
 - > MongoDB, CouchDB, Azure DocumentDB
- key value / tuple store
 - > DynamoDB, Azure Table Storage, Oracle NoSQL
- graph databases
 - > Neo4j
- multi-model databases
 - > ArangoDB, OrientDB
- object databases
 - > Versant, Objectivity VelocityDB



NoSQL Databases – cont'd

- Document databases
 - MongoDB



Document Databases

- semi-structured data
- records do not all need to have the same fields

XML:

a record is a block of XML tags and values

```
<contact>
  <firstname>Bob</firstname>
  <address>5 Oak St.</lastname>
  <hobby>saling</hobby>
</contact>
```

JSON:

a record is a list of key:value pairs

```
{
  "FirstName": "Bob",
  "Address": "5 Oak St.",
  "Hobby": "sailing"
}
```



MongoDB

- an **open-source** document database
 - no charge for *Community Server* version
- high performance
 - **embedded data models** reduce I/O activity
 - indexes
 - can include keys from embedded documents, arrays
- high availability
 - replication facility
 - automatic fail-over
 - data redundancy
- automatic scalability (horizontal)





MongoDB

- CRUD ✓
 - create, read, update, delete
- ACID ?
 - traditionally:
 - document databases are only ACID-compliant only at document level
 - no *transactions* for containing multiple I/O operations
 - application code is obliged to emulate transactions, if required
 - latency results in an **eventual consistency** model
 - MongoDB 4.0
 - introduced transactions



MongoDB: High-Level Objects

Document:

- a set of field:value pairs
 - values can be hierarchical
- analogous to RDB row

examples:

```
{ name: "sue", age: 26, status: "A", groups: [ "news", "sports" ] }
```

```
{ name: "fred", status: "A", groups: [ "sports", "hobbies", "cars" ] }
```

```
{ name: { first: "fred", last: "bloggs" }, status: "A", groups: [ "sports", "hobbies", "cars" ] }
```

Collection:

- a logical group of documents
- analogous to RDB table



MongoDB with Python

example:

```
import pymongo
connection =
pymongo.MongoClient("mongodb://localhost")
db = connection.school
students = db.students
cursor = students.find()

# find minimum homework score...
for doc in cursor:
    scores = doc["scores"]
    minhs = 101
    for entry in scores:
        if entry["type"] == "homework":
            if entry["score"] < minhs:
                minhs = entry["score"]
```

- create mongod client
- connect to database ‘school’
- create alias for table ‘students’
- fetch all rows into cursor
- loop through docs in cursor
- get value (doc) associated with key ‘scores’
- initialise min to impossibly large value (> 100%)
- loop through ‘scores’ docs, looking for ‘homework’ keys
- test each corresponding score to find new minimum



NoSQL Databases – cont'd

- Graph Databases
 - Neo4j



Graph Databases

- model members and relationships as a network
- high-level objects:
 - nodes
 - entities (e.g. people, accounts, organisations)
 - edges
 - connections between nodes
 - properties
 - node: differentiates types of nodes
 - roles, classifications, etc.
 - edge: describes the relationship
 - 2-way, 1-way, directionless
 - friend, follower, commenter, etc.



Neo4j

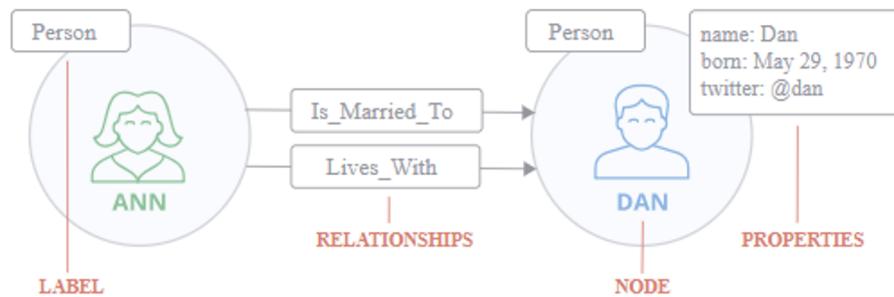
- open source
 - no charge for *Community Server* edition
- ACID-compliant, transactional database with native graph storage & processing
- online backup
- high availability
- most popular graph database





Neo4j: Basics

The Labeled Property Graph Model



Nodes

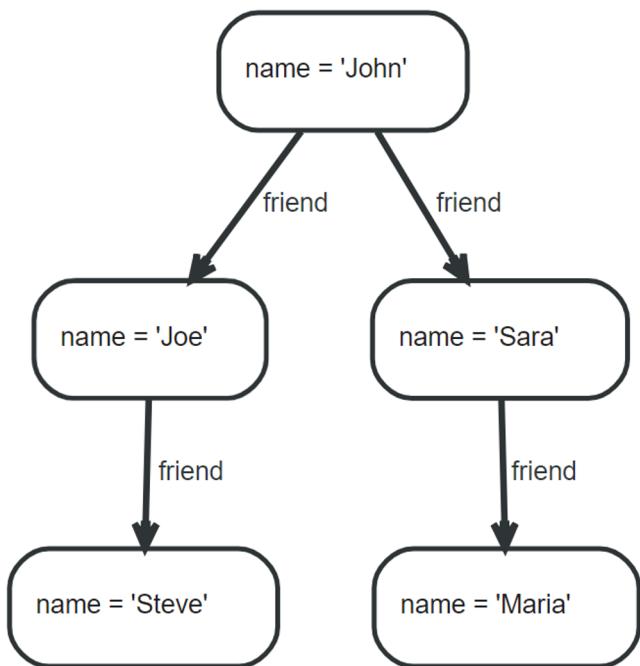
- Nodes are the main data elements
- Nodes are connected to other nodes via **relationships**
- Nodes can have one or more **properties** (i.e., attributes stored as key/value pairs)
- Nodes have one or more **labels** that describes its role in the graph

Relationships

- Relationships connect two nodes
- Relationships are directional
- **Nodes** can have multiple, even recursive relationships
- Relationships can have one or more **properties** (i.e., attributes stored as key/value pairs)



Cypher Query Language



- example: find friends of friends of John

```
MATCH (john {name: 'John'}) -[:friend]->()
      -[:friend]->(fof)
RETURN john.name, fof.name
```

output:

john.name	fof.name
"John"	"Maria"
"John"	"Steve"



Discussion: SQL vs NoSQL

SQL	NoSQL
Traditional rows and columns <ul style="list-style-type: none">• governed data model	No predefined data structure <ul style="list-style-type: none">• database at mercy of developers
Strict structure (incl. primary keys) <ul style="list-style-type: none">• schema changes difficult, risky	Ideal for unstructured data <ul style="list-style-type: none">• schema can change with application requirements
Entire column for each feature	Cheaper hardware
Industry standard	Supports design flexibility & growth <ul style="list-style-type: none">➢ popular among startups
ACID	Application code must manage transactions



Discussion: NoSQL with SQL ?!

- Why has SQL infiltrated the NoSQL paradigm?



Questions?



Appendices

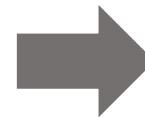


Relational Databases – Normalisation

Codd's 1st-normal form

- the domain of each attribute contains only atomic (indivisible) values
- the value of each attribute contains only a single value from that domain

Customer			
Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025, 192-122-1111
456	San	Zhang	(555) 403-1659 Ext. 53; 182-929-2929
789	John	Doe	555-808-9633



Customer			
Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025
123	Pooja	Singh	192-122-1111
456	San	Zhang	182-929-2929
456	San	Zhang	(555) 403-1659 Ext. 53
789	John	Doe	555-808-9633

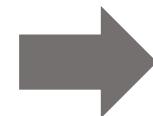


Relational Databases -- Normalisation - cont'd

Codd's 2nd-normal form

- in 1st-normal form
- no non-prime attribute is dependent on any proper subset of any candidate key of the relation
(an attribute that is not a part of any candidate key of the relation)

Electric Toothbrush Models			
Manufacturer	Model	Model Full Name	Manufacturer Country
Forte	X-Prime	Forte X-Prime	Italy
Forte	Ultraclean	Forte Ultraclean	Italy
Dent-o-Fresh	EZbrush	Dent-o-Fresh EZbrush	USA
Kobayashi	ST-60	Kobayashi ST-60	Japan
Hoch	Toothmaster	Hoch Toothmaster	Germany
Hoch	X-Prime	Hoch X-Prime	Germany



Electric Toothbrush Manufacturers	
Manufacturer	Manufacturer Country
Forte	Italy
Dent-o-Fresh	USA
Kobayashi	Japan
Hoch	Germany

Electric Toothbrush Models		
Manufacturer	Model	Model Full Name
Forte	X-Prime	Forte X-Prime
Forte	Ultraclean	Forte Ultraclean
Dent-o-Fresh	EZbrush	Dent-o-Fresh EZbrush
Kobayashi	ST-60	Kobayashi ST-60
Hoch	Toothmaster	Hoch Toothmaster
Hoch	X-Prime	Hoch X-Prime

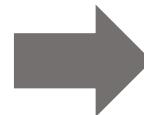


Relational Databases -- Normalisation - cont'd

Codd's 3rd-normal form

- in 2nd-normal form
- every non-prime attribute (of a table) is non-transitively dependent on every key (of a table)

Tournament Winners			
Tournament	Year	Winner	Winner Date of Birth
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977



Tournament Winners			Winner Dates of Birth	
Tournament	Year	Winner	Winner	Date of Birth
Indiana Invitational	1998	Al Fredrickson	Chip Masterson	14 March 1977
Cleveland Open	1999	Bob Albertson	Al Fredrickson	21 July 1975
Des Moines Masters	1999	Al Fredrickson	Bob Albertson	28 September 1968
Indiana Invitational	1999	Chip Masterson		



Which RDBMS Object to Use When

- queries
 - **ad hoc** queries
 - **stored** or **generated** in application code
- views
 - stored (**reusable**) queries
 - can incorporate **joins** with other views
 - **preferable** to queries in application code
- stored procedures
 - **more powerful than views** (can query and/or modify data)
 - preferred for delivering data to applications (**security, control, maintainability**)
- reports
 - **formatted output** containers based on tables, views
 - text & graphics
 - usually provided via a separate application (designed for but existing outside the RDBMS)
 - may have built-in subscription service



End of Presentation!