1. Searching in Databases

- Most common operation in a database.
- **SQL SELECT** is versatile but complex.
- **Linear Search**: Baseline efficiency.
    - Start at the beginning, check each element sequentially.
    - Worst case: O(n).

2. Lists of Records

- **Record**: A collection of attribute values (a row in a table).
- **Collection**: A set of records of the same entity type (a table).
- **Search Key**: A value used to locate records.

3. Data Structures for Storing Records

- **Contiguous Allocation (Arrays)**
    - Faster for **random access**.
    - Slower for **insertions** except at the end.
- **Linked Lists**
    - Faster **insertions** anywhere.
    - Slower **random access**.

4. Binary Search

- Works only on sorted data.
- **Time Complexity**:
    - Best case: O(1) (direct match).
    - Worst case: O(log n).

Implementation in Python:
python
CopyEdit

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
```

```
return -1
```

-

5. Database Indexing

- **Challenge**: Can't store data sorted by multiple columns.
- **Solutions**:
    1. **Sorted array of tuples** (Binary Search fast, insertions slow).
    2. **Linked list of tuples** (Fast insertions, slow searches).
    3. **Indexes** (Balanced trees, B+ Trees, Hash Indexing).

6. Binary Search Trees (BST)

- **Properties**:
    - Left subtree < Root < Right subtree.
- **Balanced BST**:
    - Avoids performance issues from skewed trees.
    - Operations: Insert, Search, Delete.
- **AVL Trees (Self-Balancing BSTs)**
    - Rebalancing with rotations (LL, RR, LR, RL cases).

7. Storage Hierarchy

- **CPU Registers** (fastest).
- **RAM** (fast, volatile).
- **Disk (SSD/HDD)** (slow, persistent).
- **Disk Block Size**: Must read full block (e.g., 2048 bytes) even for a single value.

8. B+Trees (Used in Databases)

- **Balanced, Multi-way Trees**.
- Handles large disk-based datasets efficiently.
- Search time remains O(log n).

Relational Databases & ACID Properties

9. Benefits of Relational Databases

- **Standard Model & SQL Querying**.
- **ACID Compliance**:
    - **Atomicity**: All or nothing.
    - **Consistency**: Maintains database integrity.

- **Isolation**: Transactions don't interfere.
- **Durability**: Committed changes persist.

10. Transaction Processing

- **Transaction**: A sequence of CRUD operations treated as a single unit.
  - **COMMIT**: All operations succeed.
  - **ROLLBACK**: Undo all changes if any operation fails.

11. Transaction Anomalies

1. **Dirty Read**: A transaction reads uncommitted data.
2. **Non-Repeatable Read**: Repeating a query gives different results.
3. **Phantom Reads**: A transaction sees newly inserted/deleted rows from another transaction.

12. Example - Bank Transfer Transaction

```
DELIMITER //

CREATE PROCEDURE transfer(sender_id INT, receiver_id INT, amount DECIMAL(10,2))
BEGIN
    START TRANSACTION;

    UPDATE accounts SET balance = balance - amount WHERE account_id = sender_id;
    UPDATE accounts SET balance = balance + amount WHERE account_id = receiver_id;

    IF (SELECT balance FROM accounts WHERE account_id = sender_id) < 0 THEN
        ROLLBACK;
    ELSE
        COMMIT;
    END IF;
END //

DELIMITER ;
```

Scaling and Distributed Systems

13. Issues with Relational Databases

- **Schema evolution** can be difficult.
- **Joins are expensive**.
- **Scaling challenges**:
  - Vertical Scaling (scale-up) → Expensive.
  - Horizontal Scaling (scale-out) → Requires **distributed data**.

14. Distributed Storage Models

- **Replication**: Data copied across multiple nodes.
- **Sharding**: Splitting data across multiple nodes.
- **Partitioning**: Dividing data to optimize performance.

15. CAP Theorem

- **Consistency** (Every read gets the latest write).
- **Availability** (Every request gets a response).
- **Partition Tolerance** (Works despite network failures).
- **Trade-offs**:
  - **CA** (Consistent & Available): Not Partition Tolerant.
  - **CP** (Consistent & Partition Tolerant): Not Always Available.
  - **AP** (Available & Partition Tolerant): Eventual Consistency.

16. NoSQL Databases

- **When to use?**
  - When schema flexibility is needed.
  - When scalability matters.
  - For real-time, high-performance applications.

Experimental Methodology

- **Measuring Database Performance**:
  - Memory/CPU efficiency.
  - Indexing techniques.
  - Search and retrieval speeds.
  - Comparing AVL Trees, B+Trees, and Hash Indexes.

This summary covers key concepts from database searching, indexing, ACID transactions, distributed systems, and the CAP theorem. Let me know if you need specific details

NoSQL & Key-Value Databases (KV DBs)

Distributed Databases and ACID - Pessimistic Concurrency

- **ACID Transactions** focus on *data safety* and follow a *pessimistic concurrency model*.
    - Assumes transactions must protect themselves from others (locks).
    - Uses **read and write locks** to prevent conflicts.
    - Example: Write Lock is like borrowing a book from a library—if you have it, no one else can.

Optimistic Concurrency

- Transactions do not lock data during reads or writes.
- Assumes conflicts are unlikely but handles them if they occur.
- Uses **timestamps and version numbers** to detect changes before committing transactions.
- Works well for **low-conflict systems** (e.g., backups, analytical DBs).
- **High-conflict systems** may require pessimistic locking.

NoSQL Overview

- Originally used in 1998 to describe a relational DB without SQL.
- Modern interpretation: **"Not Only SQL"**.
- Often associated with **non-relational databases**, designed to handle **unstructured, web-based data**.

CAP Theorem

You can have **2 out of 3**:

1. **Consistency (C)** – Every user has the same data at any time.
2. **Availability (A)** – System remains operational during failure.
3. **Partition Tolerance (P)** – Operates despite network failures.

Different combinations:

- **CA (Consistency + Availability):** Always provides latest data but cannot tolerate network partitions.
- **CP (Consistency + Partition Tolerance):** Data is always latest, but some requests may fail.
- **AP (Availability + Partition Tolerance):** Always available, but may return outdated data.

BASE Model (Alternative to ACID for Distributed Systems)

1. **Basically Available** – System is available, but responses may be unreliable.
2. **Soft State** – State can change over time, even without input (due to eventual consistency).
3. **Eventual Consistency** – System eventually reaches consistency once all writes complete.

NoSQL Database Categories

1. **Key-Value Stores**
2. **Document Databases**
3. **Columnar Databases**
4. **Graph Databases**
5. **Vector Databases**

Key-Value Databases (KV DBs)

Core Principles

- **Simplicity:** Only key-value pairs, no complex relations.
- **Speed:** O(1) lookup time (uses hash tables).
- **Scalability:** Easy horizontal scaling.

Use Cases

1. **EDA & Experimentation Results Store**
2. **Feature Store:** Fast feature retrieval for ML.
3. **Model Monitoring:** Real-time tracking of ML models.
4. **Session Storage:** Fast retrieval of user session data.
5. **User Profiles & Preferences**
6. **Shopping Cart Data**
7. **Caching Layer** (in front of disk-based DBs).

Redis - A Key-Value Store

- **Redis (Remote Dictionary Server)**
- Open-source, in-memory database.
- Supports:
  - **Key-Value** store.
  - **Graph, Spatial, Full-Text Search, Time-Series** data.
- **Fast performance** (>100,000 SET operations/sec).
- **Durability options:**
  - Snapshot to disk.
  - Append-only file (AOF) for logging changes.

Redis Data Types

1. **Strings**
2. **Lists** (linked lists)
3. **Sets** (unique values)
4. **Sorted Sets**
5. **Hashes** (key-value mapping)
6. **Geospatial Data**
7. **JSON (tree-structured for fast access)**

Setting Up Redis

1. **Docker Deployment**
    - Run Redis via Docker.
    - Port **6379** (default).
    - Security note: Avoid exposing Redis port in production.
2. **Connecting with DataGrip**
    - Create a new data source.
    - Use port **6379**.
    - Test connection.
3. **Redis Database Basics**
    - **16 default databases** (numbered 0–15).
    - No custom database names.

Redis Commands

Basic Key-Value Commands

sh
CopyEdit

```sh
SET user:1 "John Doe"
GET user:1
EXISTS user:1
DEL user:1
KEYS user*
```

Increment/Decrement

sh
CopyEdit

```sh
INCR counter
DECR counter
INCRBY counter 10
DECRBY counter 5
```

Hashes

sh

CopyEdit

```
HSET bike:1 model "Demios" brand "Ergonom" price 1971
HGET bike:1 model
HGETALL bike:1
```

Lists (Stacks & Queues)

sh
CopyEdit

```
LPUSH tasks "task1"
RPUSH tasks "task2"
LPOP tasks
RPOP tasks
```

Sets (Unique Elements & Operations)

sh
CopyEdit

```
SADD ds4300 "Mark"
SADD ds4300 "Sam"
SCARD ds4300
SINTER ds4300 cs3200  # Intersection
SDIFF ds4300 cs3200   # Difference
```

Redis with Python

Installation

sh
CopyEdit

```
pip install redis
```

Connecting to Redis

python
CopyEdit

```
import redis
```

```
redis_client = redis.Redis(host='localhost', port=6379, db=2,
decode_responses=True)
```

Working with Strings

python
CopyEdit

```
redis_client.set('clicks', 0)
redis_client.incr('clicks')
print(redis_client.get('clicks'))  # Output: 1
```

Using Hashes

python
CopyEdit

```
redis_client.hset('user:1001', mapping={'name': 'Alice', 'age': '30'})
print(redis_client.hgetall('user:1001'))
```

Using Pipelines (Batch Commands)

python
CopyEdit

```
pipe = redis_client.pipeline()
pipe.set("seat:1", "occupied").set("seat:2", "free")
pipe.execute()
```

Redis in Data Science & Machine Learning

- Used for **Feature Stores** (fast ML feature retrieval).
- **Session storage** for real-time inference.
- Supports **caching and model monitoring**.

1. Redis-py Overview

- **Redis-py** is the standard Python client for Redis.
- It is maintained by the Redis Company.

- GitHub repository: redis/redis-py.

Installation:
nginx
CopyEdit
```
pip install redis
```

-

2. Connecting to Redis Server

- For a **Docker deployment**, the host could be `localhost` or `127.0.0.1`.
- **Port** is likely `6379` (default).
- **Database index (db)** ranges from 0 to 15.
- **decode_responses=True** converts byte responses to strings.

Example connection code:

python
CopyEdit
```
import redis
redis_client = redis.Redis(host='localhost', port=6379, db=2,
decode_responses=True)
```

3. Redis Commands Overview

- A full list of Redis commands is available at:
  - Redis command documentation
  - Redis-py documentation
- Commands are categorized based on data structures: **Strings, Lists, Hashes, Pipelines**, etc.

---

4. String Commands

Basic Usage:
python
CopyEdit
```
r.set('clickCount:/abc', 0)
val = r.get('clickCount:/abc')
r.incr('clickCount:/abc')
print(f'click count = {r.get("clickCount:/abc")}')
```

-

Multiple Set/Get:
python
CopyEdit
```
redis_client.mset({'key1': 'val1', 'key2': 'val2', 'key3': 'val3'})
print(redis_client.mget('key1', 'key2', 'key3'))  # ['val1', 'val2', 'val3']
```

-
- Common string commands: `set()`, `mset()`, `get()`, `mget()`, `incr()`, `decr()`, `strlen()`, `append()`, etc.

---

5. List Commands

Creating a List:
python
CopyEdit
```
redis_client.rpush('names', 'mark', 'sam', 'nick')
print(redis_client.lrange('names', 0, -1))  # ['mark', 'sam', 'nick']
```

-
- Common list commands: `lpush()`, `rpush()`, `lpop()`, `rpop()`, `lrange()`, `llen()`, etc.

---

6. Hash Commands

Creating a Hash:
python
CopyEdit
```
redis_client.hset('user-session:123',
    mapping={'first': 'Sam', 'last': 'Uelle', 'company': 'Redis', 'age': 30})
print(redis_client.hgetall('user-session:123'))
```

-
- Common hash commands: `hset()`, `hget()`, `hgetall()`, `hkeys()`, `hdel()`, `hexists()`, etc.

---

7. Redis Pipelines

- **Pipelines reduce network overhead** by batching commands.

Example Usage:
python
CopyEdit

```
r = redis.Redis(decode_responses=True)
pipe = r.pipeline()

for i in range(5):
    pipe.set(f"seat:{i}", f"#{i}")

set_5_result = pipe.execute()
print(set_5_result)  # [True, True, True, True, True]

pipe = r.pipeline()
get_3_result =
pipe.get("seat:0").get("seat:3").get("seat:4").execute()
print(get_3_result)  # ['#0', '#3', '#4']
```

-

---

8. Redis in Data Science & Machine Learning

- Redis is used in ML applications, such as **feature stores**.
- Reference sources:
    - Feature Stores Explained
    - Redis in MLOps

Introduction to the Graph Data Model

What is a Graph Database?

- A data model based on the **graph data structure**
- Composed of **nodes** and **edges**
    - Edges connect nodes

- Each node and edge is uniquely identified
    - Both nodes and edges can contain properties (e.g., name, occupation)
- Supports queries based on **graph-oriented operations**, such as:
    - Traversals
    - Shortest path
    - Many others

Where do Graphs Show Up?

- **Social Networks**
    - Examples: Instagram, modeling social interactions in psychology and sociology
- **The Web**
    - A large graph of "pages" (nodes) connected by hyperlinks (edges)
- **Chemical and Biological Data**
    - Systems biology, genetics, chemistry (interaction relationships)

Basics of Graphs and Graph Theory

What is a Graph?

A **Labeled Property Graph** consists of:

- **Nodes (Vertices)**
- **Relationships (Edges)**
- **Labels** to classify nodes into groups
- **Properties** (key-value pairs) on nodes and relationships
- Nodes **can exist** without relationships, but edges **must connect** nodes

Example Graph Model

Node Labels:

- `Person`
- `Car`

Relationship Types:

- `Drives`
- `Owns`

- `Lives_with`
- `Married_to`

Properties:

Key-value pairs attached to nodes and edges

Paths in Graphs

- A **path** is an **ordered sequence of nodes** connected by edges, **without repetition**
- **Example of a valid path:**
  - `1 → 2 → 6 → 5`
- **Invalid path (repetition of nodes):**
  - `1 → 2 → 6 → 2 → 3`

Flavors of Graphs

1. **Connected vs. Disconnected**

   - **Connected:** There is a path between any two nodes
   - **Disconnected:** Some nodes cannot be reached from others
2. **Weighted vs. Unweighted**

   - **Weighted:** Edges have a weight property (useful for pathfinding)
   - **Unweighted:** No weight values on edges
3. **Directed vs. Undirected**

   - **Directed:** Relationships define a start and end node
   - **Undirected:** No direction in edges
4. **Cyclic vs. Acyclic**

   - **Cyclic:** Contains at least one cycle
   - **Acyclic:** No cycles exist
5. **Sparse vs. Dense**

   - **Sparse:** Few edges compared to the number of nodes
   - **Dense:** Many edges compared to the number of nodes
6. **Trees**

    - A special type of graph (acyclic and hierarchical)

Types of Graph Algorithms

Pathfinding Algorithms

- Finding the **shortest path** between two nodes
- "Shortest" can mean:
    - Fewest edges
    - Lowest total weight
- Examples:
    - **Minimum Spanning Tree**
    - **Cycle Detection**
    - **Max/Min Flow**

BFS vs. DFS

- **Breadth-First Search (BFS):** Explores level by level
- **Depth-First Search (DFS):** Explores as deep as possible before backtracking

Shortest Path Algorithms

- **Dijkstra's Algorithm** (for graphs with positive weights)
- *A Algorithm** (heuristic-based shortest path)

Centrality & Community Detection

- **Centrality:** Determines which nodes are "more important"
    - Example: Identifying **social media influencers**
- **Community Detection:**
    - Finds clusters or partitions within a graph

Famous Graph Algorithms

- **Dijkstra's Algorithm:** Finds the shortest path in a positively weighted graph
- *A Algorithm:** Uses heuristics for more efficient shortest path finding
- **PageRank:** Measures node importance based on incoming connections

---

Graph Databases - Neo4j

- A graph database system that supports **transactional and analytical processing**
- A type of **NoSQL database**
- **Schema-optional** (can impose one but not required)
- Supports **various indexing methods**
- **ACID-compliant**
- Can be **distributed** across multiple nodes
- Other graph databases:
    - **Microsoft CosmosDB**
    - **Amazon Neptune**

Neo4j

Introduction to Neo4j

- A **Graph Database System** that supports both transactional and analytical processing of graph-based data.
- A relatively new class of NoSQL databases.
- Considered **schema-optional** (a schema can be imposed).
- Supports various types of **indexing**.
- **ACID compliant**.
- Supports **distributed computing**.
- Similar databases: **Microsoft CosmosDB, Amazon Neptune**.

Neo4j Query Language and Plugins

Cypher

- Neo4j's graph query language, created in **2011**.
- Aims to be an **SQL-equivalent** language for graph databases.

Provides a visual way of **matching patterns and relationships**:
scss
CopyEdit

```
(nodes)-[:CONNECT_TO]->(otherNodes)
```

-

APOC Plugin (Awesome Procedures on Cypher)

- An add-on library that provides **hundreds of procedures and functions**.

Graph Data Science Plugin

- Provides **efficient implementations of common graph algorithms**.

Neo4j in Docker Compose

- **Docker Compose** allows multi-container management.
- The setup is **declarative**, using a `docker-compose.yaml` file.
- Defines:
  - **Services**
  - **Volumes**
  - **Networks**, etc.
- Provides a consistent method for producing **identical environments**.
- Uses **command-line interaction**.

Example docker-compose.yaml

yaml
CopyEdit

```yaml
services:
  neo4j:
    container_name: neo4j
    image: neo4j:latest
    ports:
      - 7474:7474
      - 7687:7687
    environment:
      - NEO4J_AUTH=neo4j/${NEO4J_PASSWORD}
      - NEO4J_apoc_export_file_enabled=true
      - NEO4J_apoc_import_file_enabled=true
      - NEO4J_apoc_import_file_use__neo4j__config=true
      - NEO4J_PLUGINS=["apoc", "graph-data-science"]
    volumes:
      - ./neo4j_db/data:/data
      - ./neo4j_db/logs:/logs
      - ./neo4j_db/import:/var/lib/neo4j/import
      - ./neo4j_db/plugins:/plugins
```

Important: Never store **secrets** in the Docker Compose file; use `.env` files.

Environment Variables with .env Files

- **.env files** store a collection of environment variables.
- Useful for managing **different environments** (`.env.local`, `.env.dev`, `.env.prod`).

Example:

ini
CopyEdit
```
NEO4J_PASSWORD=abc123!!!
```

Docker Compose Commands

Test if you have Docker CLI properly installed:

css
CopyEdit
```
docker --version
```

Major commands:

sql
CopyEdit
```
docker compose up
docker compose up -d
docker compose down
docker compose start
docker compose stop
docker compose build
docker compose build --no-cache
```

Accessing Neo4j

Open the **Neo4j Browser** at:
makefile

CopyEdit

```
localhost:7474
```

-
- Log in to manage the database.

Inserting Data by Creating Nodes

cypher
CopyEdit

```
CREATE (:User {name: "Alice", birthPlace: "Paris"})
CREATE (:User {name: "Bob", birthPlace: "London"})
CREATE (:User {name: "Carol", birthPlace: "London"})
CREATE (:User {name: "Dave", birthPlace: "London"})
CREATE (:User {name: "Eve", birthPlace: "Rome"})
```

Adding an Edge (Relationship)

cypher
CopyEdit

```
MATCH (alice:User {name:"Alice"})
MATCH (bob:User {name: "Bob"})
CREATE (alice)-[:KNOWS {since: "2022-12-01"}]->(bob)
```

Note: Relationships in Neo4j are **directed**.

Querying Data

Find all users born in London:

cypher
CopyEdit

```
MATCH (usr:User {birthPlace: "London"})
RETURN usr.name, usr.birthPlace
```

Importing Data from CSV

1. **Download Dataset**

Clone the repo:
arduino
CopyEdit
```
https://github.com/PacktPublishing/Graph-Data-Science-with-Neo4j
```

   ○

Navigate to `Chapter02/data`, unzip `netflix.zip`, and move `netflix_titles.csv` to:
bash
CopyEdit
```
neo4j_db/neo4j_db/import
```

   ○

2. **Basic Data Import**

cypher
CopyEdit
```
LOAD CSV WITH HEADERS
FROM 'file:///netflix_titles.csv' AS line
CREATE(:Movie {
    id: line.show_id,
    title: line.title,
    releaseYear: line.release_year
  }
)
```

General CSV Loading Syntax

cypher
CopyEdit
```
LOAD CSV
[WITH HEADERS]
FROM 'file:///file_in_import_folder.csv'
AS line
```

```
[FIELDTERMINATOR ',']
// Perform operations on 'line'
```

Handling Duplicate Nodes

Naïve Approach (Duplicates Present)

cypher
CopyEdit

```
LOAD CSV WITH HEADERS
FROM 'file:///netflix_titles.csv' AS line
WITH split(line.director, ",") AS directors_list
UNWIND directors_list AS director_name
CREATE (:Person {name: trim(director_name)})
```

Problem: Creates duplicate nodes when a director directs multiple movies.

Improved Approach (Avoiding Duplicates)

cypher
CopyEdit

```
MATCH (p:Person) DELETE p

LOAD CSV WITH HEADERS
FROM 'file:///netflix_titles.csv' AS line
WITH split(line.director, ",") AS directors_list
UNWIND directors_list AS director_name
MERGE (:Person {name: director_name})
```

Adding Relationships (Edges)

cypher
CopyEdit

```
LOAD CSV WITH HEADERS
FROM 'file:///netflix_titles.csv' AS line
MATCH (m:Movie {id: line.show_id})
WITH m, split(line.director, ",") AS directors_list
```

```
UNWIND directors_list AS director_name
MATCH (p:Person {name: director_name})
CREATE (p)-[:DIRECTED]->(m)
```

Verifying Data

To check if a specific movie exists in the database:

cypher
CopyEdit

```
MATCH (m:Movie {title: "Ray"})<-[:DIRECTED]-(p:Person)
RETURN m, p
```