Here is a structured **summary of the ACID Properties in DBMS** from the GeeksforGeeks article:

🔷 ACID Properties in DBMS

ACID stands for:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

These ensure that transactions are processed reliably in a database system.

1️⃣ Atomicity – *All or Nothing*

- **Definition**: A transaction either completes entirely or doesn't happen at all.
- **Key operations**:
  - **Commit** → Changes are permanently applied.
  - **Rollback** → All interim changes are undone.
- **Example**: Transferring $100 from X to Y: if debit from X happens but credit to Y fails, atomicity ensures the debit is rolled back.

2️⃣ Consistency – *Valid to Valid State*

- **Definition**: A transaction must take the database from one valid state to another.
- **Constraints**: Follows all rules like primary keys, foreign keys, triggers, etc.
- **Example**: Total balance before and after a transaction should remain the same (e.g., $700 → $700).

3️⃣Isolation – *No Interference*

- **Definition**: Concurrent transactions must not affect each other.
- **Prevents**:
    - **Dirty Reads**: Reading uncommitted changes.
    - **Non-repeatable Reads**: Different results for the same query within a transaction.
    - **Phantom Reads**: Rows appearing/disappearing during execution.
- **Example**: While T updates values, T′ must not read partial changes.

4️⃣Durability – *Permanent Changes*

- **Definition**: Once a transaction is committed, changes must survive system failures.
- **Example**: After transferring money and committing, even if a crash occurs, the update remains saved.

🧠 Why ACID Is Important

Area	Importance
Data Integrity	Prevents corruption by rolling back incomplete transactions
Concurrency Control	Ensures isolated execution of transactions
Fault Tolerance	Database can recover to the last committed state

---

🧰 Who Maintains What?

Property	Maintained By
Atomicity	Transaction Manager

Consistency	Application Programmer
Isolation	Concurrency Control Manager
Durability	Recovery Manager

🏥 Use Cases

- **Banking**: Money transfers must be accurate and consistent.
- **E-commerce**: Inventory and payment data must be synchronized.
- **Healthcare**: Patient records must remain reliable and consistent.

Would you like this summary in **PDF**, **Cheat Sheet format**, or **Flashcards** for revision?

Here is a **concise summary with main headings and one-line explanations** from the GeeksforGeeks article on **"Difference between Primary Key and Candidate Key"**:

🔹 What is Primary Key?

A **primary key** is a unique attribute (or set of attributes) in a table that **identifies each tuple (row) uniquely** and **cannot be NULL**.

✅ Advantages of Primary Key

- **Fast Searches**: Enables quick data access due to uniqueness and indexing.
- **Prevents Duplicates**: No two rows can have the same primary key.
- **Automatic Optimization**: Helps optimize query performance.

❌ Disadvantages of Primary Key

- **No NULL Allowed**: Reduces flexibility in certain scenarios.
- **Only One Allowed**: Limits use of other unique fields as primary.

🔹 What is Candidate Key?

A **candidate key** is any attribute (or group of attributes) that **can uniquely identify** a tuple. **One of them becomes the primary key**, others remain as alternate candidate keys.

✅ Advantages of Candidate Key

- **Flexibility**: You can choose the most suitable candidate as the primary key.
- **Backup Option**: Acts as fallback if the primary key is dropped or fails.
- **Ensures Uniqueness**: All candidate keys are unique.

❌ Disadvantages of Candidate Key

- **Complexity**: Having many candidates can complicate schema understanding.
- **Storage Overhead**: Indexing multiple candidate keys may consume more space.

🆚 Difference Between Primary and Candidate Key

Feature	Primary Key	Candidate Key
Count	Only one per table	Can be multiple
NULLs Allowed	❌ No	✅ Yes (possible)
Mandatory in a table?	Optional	Candidate key(s) must exist
Uniqueness	Must be unique	All are unique
Becomes Primary Key?	Already is	One of them may be chosen
Indexing	Indexed by default	Not always indexed
Storage	Less storage	May take more space
Composite Allowed?	❌ Not usually	✅ Allowed

🏁 Conclusion

Both primary and candidate keys help **maintain uniqueness** in a table. The **primary key is chosen** among the candidate keys and used to **enforce entity integrity**, while **candidate keys offer alternative unique identifiers** and flexibility in design.

Let me know if you'd like this turned into a **PDF, chart, or flashcard-style summary**!

Here's a clear **summary of all the methods** to find the **Nth highest salary** in SQL as explained in the GeeksforGeeks article you shared:

✅ Table Setup

```
CREATE TABLE emp (
  emp_name VARCHAR(50),
  emp_salary DECIMAL(10,2)
);

INSERT INTO emp (emp_name, emp_salary) VALUES
('Shubham Thakur', 50000.00),
('Aman Chopra', 60000.50),
('Naveen Tulasi', 75000.75),
('Bhavika Uppala', 45000.25),
('Nishant Jain', 80000.00);
```

🔢 Methods to Find Nth Highest Salary:

1. Using `DENSE_RANK()` (Best for Tied Salaries)

```
SELECT * FROM (
  SELECT emp_name, emp_salary, DENSE_RANK() OVER (ORDER BY emp_salary
DESC) AS r
  FROM emp
) AS subquery
WHERE r = N;
```

➡️ Replace `N` with desired rank (e.g. `r = 3` for 3rd highest).

---

2. Using `LIMIT` with Subqueries (MySQL / MariaDB)

```
SELECT * FROM (
  SELECT * FROM emp
  ORDER BY emp_salary DESC
  LIMIT N
) AS T
ORDER BY T.emp_salary ASC
LIMIT 1;
```

➡️ This returns the **Nth highest** by skipping first N-1 rows.

🟢 Alternate shortcut (MySQL):

```
SELECT * FROM emp
ORDER BY emp_salary DESC
LIMIT N-1, 1;
```

---

3. Using `MIN()` + `IN` Subquery

```
SELECT * FROM emp
WHERE emp_salary = (
  SELECT MIN(emp_salary) FROM (
    SELECT DISTINCT emp_salary FROM emp
    ORDER BY emp_salary DESC
    LIMIT N
  ) AS temp
);
```

➡️ Finds the **minimum among top N** distinct salaries, which is the Nth highest.

---

4. Using `COUNT(DISTINCT)`

```
SELECT emp_name, emp_salary FROM emp e1
WHERE (SELECT COUNT(DISTINCT emp_salary)
       FROM emp e2
       WHERE e2.emp_salary > e1.emp_salary) = N - 1;
```

➡️ For each employee, this checks if **exactly N-1 salaries are greater** than theirs.

---

5. Simple `LIMIT` Clause

```
SELECT emp_salary FROM emp
```

```
ORDER BY emp_salary DESC
LIMIT N-1, 1;
```

➡️ Skips N-1 salaries and picks the Nth highest.

🟢 To get the name too:

```
SELECT emp_name FROM emp
WHERE emp_salary = (
  SELECT emp_salary FROM emp
  ORDER BY emp_salary DESC
  LIMIT N-1, 1
);
```

---

6. To Find the Nth Lowest Salary

➡️ Just reverse the sort order (use `ASC` instead of `DESC`)

```
SELECT emp_salary FROM emp
ORDER BY emp_salary ASC
LIMIT N-1, 1;
```

✅ Which One Should You Use?

Method	Use Case
`DENSE_RANK()`	When salaries can be the same (ties)
`LIMIT`	Fastest in MySQL
`MIN()` with subquery	Good for distinct top salaries
`COUNT(DISTINCT)`	Useful when you want logic-based filtering

Would you like practice questions or code to test this on a local database or online SQL playground?

Here's a **clean summary** of the GeeksforGeeks article:
📌 **"5 Best Practices for Writing SQL JOIN Query"**
([Published: 23 July 2025])

🧠 What is SQL JOIN?

SQL `JOIN` allows combining rows from two or more tables based on related columns, helping maintain a normalized and efficient database.

🔗 Types of SQL JOINs

Join Type	Description	Result Example
INNER JOIN	Returns only matching records in both tables	Matching rows only
LEFT JOIN	All records from left, matched from right (NULL if none)	All left rows + right matches
RIGHT JOIN	All from right, matched from left (NULL if none)	All right rows + left matches
FULL JOIN	All records from both tables, NULLs where no match	All rows from both, with NULLs
CROSS JOIN	Returns Cartesian product (every combo of rows)	All combinations

---

✅ 5 Best Practices for Writing SQL JOINs

1. **Use Table Aliases**
    - Makes queries shorter, cleaner, and easier to read.

Example:

SELECT e.name, d.name

FROM employees e

JOIN departments d ON e.dept_id = d.id;

   ○

2. **Understand Join Types Before Using Them**
   - Know the **difference** between INNER, LEFT, RIGHT, FULL joins.
   - Helps avoid incorrect or unexpected results.
3. **Carefully Design the JOIN Condition**
   - Ensure the join condition uniquely identifies rows.
   - May involve multiple columns or non-equality conditions.

Example:

ON a.id = b.a_id AND a.status = b.status

   ○

4. **Use Explicit JOIN Syntax**

Prefer this:

SELECT * FROM A JOIN B ON A.id = B.id

   ○

Over this (implicit):

SELECT * FROM A, B WHERE A.id = B.id

   ○
   - Explicit syntax improves clarity and debugging.
5. **Learn and Use Multiple JOINS**
   - Joins aren’t limited to just two tables.
   - Write clear, ordered queries with multiple joins.

- Use indentation and place each `JOIN` on a new line for readability.

🏁 Conclusion

Efficient JOINs improve performance, reduce resource usage, and return accurate results — essential when working with large-scale databases.

Would you like this formatted into a **PDF**, **one-page cheat sheet**, or **flashcards**?

Here's a **summary of the GeeksforGeeks article** titled **"Introduction to Database Normalization"** using main headings with one-line explanations:

🔹 Introduction to Database Normalization

Normalization is the process of organizing database attributes to reduce redundancy and improve efficiency.

🔹 Why Do We Need Normalization?

To eliminate data anomalies (insertion, deletion, update) and maintain data integrity in relational databases.

🔹 Types of Anomalies

- **Insertion Anomaly**: Can't insert data due to missing required fields.
- **Deletion Anomaly**: Deleting one record unintentionally deletes important related data.
- **Update Anomaly**: Updating data in one place but not others causes inconsistency.

🔹 Before and After Normalization

Before: Redundant and inconsistent data.
After: Structured, logical tables with improved consistency and efficiency.

🔹 Features of Database Normalization

- Removes redundancy
- Ensures consistency
- Simplifies data management
- Improves design
- Avoids anomalies
- Promotes standardization

🔹 Normal Forms in DBMS

Each form defines a stricter set of rules for better database organization:

Normal Form	Description
1NF	All attributes must be atomic (no repeating values).
2NF	1NF + no partial dependency on primary key.
3NF	2NF + no transitive dependency on non-prime attributes.
BCNF	3NF + every determinant must be a super key.
4NF	BCNF + no multi-valued dependency.

5NF	4NF + no join dependency; cannot be further losslessly decomposed.

🔹 Advantages of Normalization

- Eliminates redundancy
- Increases consistency
- Simplifies updates
- Enhances query flexibility
- Supports integration across apps

🔹 Disadvantages of Normalization

- May slow performance due to complex joins
- Can reduce data context
- Requires strong DB design knowledge
- Can complicate the schema

Let me know if you'd like:

- A **PDF version** of this summary
- A **table format of normal forms with examples**
- Help creating **questions or notes for revision**

Here's a **summary** of the GeeksforGeeks article on **Multiple Access Protocols in Computer Networks** with main headings and brief explanations:

🔹 What Are Multiple Access Protocols?

Multiple Access Protocols are methods used in computer networks to manage how multiple devices share the same communication channel without collisions or data loss.

🔹 Who Handles Data Transmission?

The **Data Link Layer** is responsible for data transmission and includes:

- **Data Link Control**: Ensures reliable data delivery using framing, error control, and flow control.
- **Multiple Access Control**: Manages access to a shared medium when no dedicated link exists.

🔹 Classification of Multiple Access Protocols

1. **Random Access**
2. **Controlled Access**
3. **Channelization**

🔸 1. Random Access Protocols

Any station can transmit anytime, increasing chances of collision. Includes:

✔️ ALOHA

- **Pure ALOHA**: Transmit anytime; if no ACK, retry after random backoff.
  ⏱ Vulnerable time = 2 × frame time
  📈 Max throughput: 18.4%

- **Slotted ALOHA**: Time is divided into slots; transmission starts only at slot beginnings.
  ⏱ Vulnerable time = 1 × frame time
  📈 Max throughput: 36.8%

✔️ CSMA (Carrier Sense Multiple Access)

Before sending, a station checks if the channel is idle.

CSMA Access Modes:

- **1-Persistent**: Keep checking until idle, transmit immediately.
- **Non-Persistent**: Wait random time before rechecking.
- **P-Persistent**: Transmit with probability **p** when idle.
- **O-Persistent**: Stations have predefined priorities.

✔️ CSMA/CD (Collision Detection)

Abort transmission if collision is detected (used in wired LAN like Ethernet).

✔️ CSMA/CA (Collision Avoidance)

Avoid collisions by using:

- **Interframe Space (IFS)**: Delay before transmission.
- **Contention Window**: Random wait slots.
- **Acknowledgement**: Retransmit if no ACK received.

🔸 2. Controlled Access Protocols

Stations take turns to transmit, avoiding collisions:

- Examples: **Polling**, **Token Passing**

🔸 3. Channelization Protocols

Divide channel by time, frequency, or code for simultaneous access.

✔️ FDMA (Frequency Division Multiple Access)

Bandwidth is split into non-overlapping frequency bands.

✔️ TDMA (Time Division Multiple Access)

Time is divided into slots, and each station gets a turn. Needs synchronization.

✔️ CDMA (Code Division Multiple Access)

All stations transmit simultaneously using different codes.

✔️ OFDMA (Orthogonal Frequency Division Multiple Access)

Divides bandwidth into small sub-carriers; widely used in **5G**.

✔️ SDMA (Spatial Division Multiple Access)

Uses antennas and spatial separation (used in **MIMO** systems).

🔹 Features of Multiple Access Protocols

- **Contention-based access** (like ALOHA, CSMA)
- **Carrier sensing and collision detection** (CSMA/CD)
- **Collision avoidance** (CSMA/CA)
- **Token passing** (controlled access)
- **Bandwidth efficiency** depends on protocol type

✅ Summary Table

Type	Subtypes	Key Feature
Random Access	ALOHA, CSMA, CSMA/CD, CSMA/CA	Contention-based, chance of collisions
Controlled	Token Passing, Polling	Turns for access, avoids collisions

Channelization	FDMA, TDMA, CDMA, OFDMA, SDMA	Divides medium by freq/time/code/space

---

Would you like:

- A **PDF cheat sheet** of this summary?
- A **comparison table** of these protocols with advantages/disadvantages?
- **MCQs or flashcards** for revision?
  Here's a **clean and concise summary** of the **GeeksforGeeks article on B-Trees** that you just copied, focusing on key concepts and facts you should remember for interviews or exams:

🔷 B-TREE - OVERVIEW

A **B-Tree** is a self-balancing **m-way tree** designed to work efficiently on **disk-based storage systems**, ensuring minimal disk I/O.

📌 PROPERTIES OF B-TREE (Order = m)

1. Every node can have **at most m children** and **m-1 keys**.
2. **All leaves are at the same level.**
3. **Keys in a node** are stored in **ascending order**.
4. Every non-root, non-leaf node must have **at least ⌈m/2⌉ children**.
5. Every non-root node must have **at least ⌈m/2⌉ - 1 keys**.
6. Root node:
   - Can have minimum **1 key**
   - Must have at least **2 children** if it's not a leaf.
7. A node with `n-1` keys has `n` non-null children.

🔢 HEIGHT FORMULAS

- **Minimum height** when fully filled:
  $h_{min} = \lceil \log (n + 1) \rceil - 1$
- **Maximum height** when least filled (with $t = \lceil m/2 \rceil$):
  $h_{max} = \lfloor \log ((2n + 1)) \rfloor$

⚡ NEED FOR B-TREE

- **Efficient for disk storage systems**
- Better than Binary Trees or M-way trees for **large datasets**
- Ensures **logarithmic time complexity** for search, insert, delete
- Useful in **databases, file systems**, etc.

🛠️ OPERATIONS & TIME COMPLEXITY

Operation	Time Complexity
Search	O(log n)
Insert	O(log n)
Delete	O(log n)
Traverse	O(n)

🔍 SEARCH ALGORITHM (Pseudo C++)

- struct Node {
-     int n;
-     int key[MAX_KEYS];
-     Node* child[MAX_CHILDREN];
-     bool leaf;
- };
-
- Node* BtreeSearch(Node* x, int k) {
-     int i = 0;
-     while (i < x->n && k > x->key[i])
-         i++;
-     if (i < x->n && k == x->key[i])
-         return x;
-     if (x->leaf)
-         return nullptr;
-     return BtreeSearch(x->child[i], k);
- }

✅ APPLICATIONS

- **Database Indexing**
- **File Systems**
- **CAD systems** (for spatial data)
- **NLP, cryptography**, and **network routing tables**

👍 ADVANTAGES

- Always balanced → Consistent performance
- Good for **huge data** and **external memory**

- Efficient **multilevel indexing**
- Supports **high concurrency**

👎 DISADVANTAGES

- **More complex** to implement than binary trees
- For **small datasets**, slower than BST
- High **disk usage** possible

If you want notes, a short quiz, or code implementation for insert/delete, let me know!

Here's a **concise and clear summary** of the key points from the **GeeksforGeeks article on the difference between SQL and NoSQL**:

🔍 SQL vs NoSQL – Key Differences

Aspect	SQL (Relational)	NoSQL (Non-Relational)
Data Structure	Tables with rows and columns	Key-value, Document, Column-family, or Graph-based
Schema	Fixed schema (predefined)	Dynamic/flexible schema
Scalability	Vertically scalable (upgrade a single server)	Horizontally scalable (add more servers)

Data Integrity	ACID compliant (strong consistency, reliability)	BASE compliant (more availability, eventual consistency)
Query Language	SQL (Structured Query Language)	Varies by database (e.g., MongoDB uses its own syntax)
Performance	Great for complex queries, structured data	Great for high-volume, fast read/write operations
Use Cases	Banking, ERP, CRM, multi-row transactions	Big data, real-time web apps, flexible or changing data
Examples	MySQL, PostgreSQL, Oracle, MS SQL Server	MongoDB, Cassandra, CouchDB, Neo4j

---

✅ When to Choose SQL

- Data consistency and transaction safety are critical.
- Data is structured with clear relationships.
- Complex joins and queries are needed.
- Use case examples: **Banking**, **Inventory systems**, **Accounting**.

✅ When to Choose NoSQL

- You have **unstructured or semi-structured data**.
- Need for **scalability** across servers.

- Frequent **changes in data model**.
- Use case examples: **Social media**, **CMS**, **Real-time analytics**, **IoT data**.

🧠 Key Concepts

- **SQL databases follow ACID properties**:
    - **A**tomicity: All steps of a transaction are completed or none at all.
    - **C**onsistency: Data remains in a valid state.
    - **I**solation: Transactions don't interfere with each other.
    - **D**urability: Completed transactions remain even after system failure.
- **NoSQL databases follow the CAP theorem**:
    - **C**onsistency
    - **A**vailability
    - **P**artition Tolerance
      (You can only fully achieve **2 out of 3** at the same time.)

📝 Example Data Comparison

SQL Table Format:

- | id | name | category | qty |
- |-----|------------------|--------------|-----|
- | 101 | Apples | food | 150 |

NoSQL (JSON-like Format):

- {
- "id": "102",
- "category": "electronics",
- "name": "Apple MacBook Air",
- "qty": "10",
- "specifications": {

- "storage": "256GB SSD",
- "cpu": "8 Core",
- "camera": "1080p FaceTime HD"
- }
- }

If you'd like a **PDF summary** of this comparison or want help choosing between SQL and NoSQL for a specific project idea, just let me know!