## Binary Search Trees (BST)

- A BST is a binary tree where:
  - Left subtree nodes have keys less than the parent.
  - Right subtree nodes have keys greater than the parent.
- Keys are distinct.
- Each node has pointers to parent, left child, and right child.

### Traversal Methods

- **Inorder**: Left subtree → Node → Right subtree (Outputs sorted order)
- **Preorder**: Node → Left subtree → Right subtree
- **Postorder**: Left subtree → Right subtree → Node

### BST Operations

1. **Searching**
   - Compare the target key with the node's key and traverse left or right accordingly.
   - Time Complexity: O(h), where h is the tree height.
2. **Insertion**
   - Search for the correct position and insert while maintaining BST property.
3. **Deletion**
   - Three cases:
     - Node has no children → Remove it.
     - Node has one child → Replace it with the child.
     - Node has two children → Replace it with its in-order successor.

### BST Performance

- Worst-case height: O(n) (degenerated tree).
- Average height for random BST: O(log n).

## Balanced Trees

BSTs can become unbalanced, degrading search efficiency. Self-balancing trees maintain O(log n) height.

## AVL Trees

- Self-balancing BST with height-balance property:
  - For every node, height difference between left and right subtrees ≤ 1.
- Rotations:
  - **Single Rotation (LL, RR)**: Simple rotation when imbalance occurs on one side.

    - **Double Rotation (LR, RL)**: Two rotations required when imbalance occurs across different sides.
- Time Complexity:
    - Search: O(log n)
    - Insert/Delete: O(log n) (may require rebalancing)

## B-Trees and B+ Trees

- Designed for disk-based storage and indexing.
- Generalization of BST where each node can have multiple children.
- Each node has up to **m** children (order **m** tree).
- Ensures that all leaf nodes are at the same depth.

### B-Trees

- Internal nodes store keys and pointers to child nodes.
- Leaf nodes contain data records.
- Insertion and deletion maintain balance through **node splitting and merging**.

### B+ Trees

- Variation of B-Trees where:
    - **Internal nodes only store keys** (not data).
    - **Leaf nodes store all data records** and are linked together.
    - Enables fast range queries and sequential access.

## B+ Tree Insertion Process

1. Find the correct leaf node.
2. Insert the key.
3. If the leaf is full, **split** it and push the middle key to the parent.
4. If the parent is also full, **split** it and repeat up to the root.
5. If the root splits, a new level is created, making the tree deeper.

## Conclusion

- **BSTs** are simple but can become unbalanced.
- **AVL Trees** keep balance at the cost of rotations.
- **B-Trees and B+ Trees** optimize disk-based searches, with **B+ Trees** excelling in range queries.
- Choice of structure depends on access pattern and data size.

# Database Systems: Moving Beyond Relational Models & NoSQL

## 1. Relational Database Model Benefits

- **Standard model and language**: Widespread SQL adoption
- **ACID compliance**: Atomicity, Consistency, Isolation, Durability
- **Strengths**: Works well with structured data and can handle large data volumes
- **Established ecosystem**: Well-understood with extensive tooling and expertise

### Performance Optimization Techniques

- Indexing
- Direct storage control
- Column vs. row oriented storage
- Query optimization
- Caching/prefetching
- Materialized views
- Precompiled stored procedures
- Data replication and partitioning

## 2. Transaction Processing & ACID Properties

### Transaction Fundamentals

- **Definition**: Sequence of CRUD operations performed as a single logical unit
- **Outcomes**: Either fully succeeds (COMMIT) or entirely fails (ROLLBACK)
- **Benefits**: Data integrity, error recovery, concurrency control, reliable storage

### ACID Properties in Detail

- **Atomicity**: Transaction is an indivisible unit - fully executed or not at all
- **Consistency**: Database moves from one consistent state to another
- **Isolation**: Concurrent transactions don't interfere with each other
  - Potential issues: Dirty reads, non-repeatable reads, phantom reads
- **Durability**: Committed changes persist even during system failures

#### Isolation Issues

1. **Dirty Read**: Transaction reads uncommitted data from another transaction

2. **Non-repeatable Read**: Two queries in same transaction get different values due to another committed transaction
3. **Phantom Reads**: Records appear/disappear during a transaction due to another transaction's inserts/deletes

# 3. Limitations of Relational Databases

- Schema evolution challenges
- Full ACID compliance not always necessary
- Expensive join operations
- Limited handling of semi-structured/unstructured data (JSON, XML)
- Horizontal scaling difficulties
- Performance limitations for real-time/low-latency systems

# 4. Scaling Approaches

- **Vertical Scaling (Up)**: Using more powerful systems
    - Easier implementation, no architecture changes
    - Practical and financial limitations
- **Horizontal Scaling (Out)**: Distributed computing model
    - Modern systems making this more feasible
    - Requires handling distributed data storage

# 5. Distributed Data Systems

## Characteristics

- Computers operate concurrently
- Independent failure modes
- No shared global clock

## Data Storage Approaches

- Data stored on multiple nodes, typically replicated
- Can be relational or non-relational
- Network partitioning is inevitable
- System needs partition tolerance

# 6. CAP Theorem

## Core Principle

It's impossible for a distributed data store to simultaneously provide more than two of:

- **Consistency**: Every read receives most recent write or error
- **Availability**: Every request receives a non-error response
- **Partition Tolerance**: System operates despite network issues

### Practical Interpretations

- **C+A**: Latest data, guaranteed response, but vulnerable to network partitions
- **C+P**: Always latest data or no response, handles partitions
- **A+P**: Always responds but may not have latest data, handles partitions

### Reality

- Not about always giving up one property
- More about trade-offs when partitions occur

## 7. NoSQL Databases

### Concurrency Models

- **Pessimistic Concurrency (ACID)**: Assumes conflicts will occur, uses locks
- **Optimistic Concurrency**: No locks, assumes conflicts are rare
    - Uses timestamps and version numbers
    - Good for read-heavy systems
    - Less efficient for high-conflict systems

### BASE - Alternative to ACID for Distributed Systems

- **Basically Available**: System works most of the time
- **Soft State**: State can change without input
- **Eventual Consistency**: System will eventually reach consistency

## 8. Key-Value Databases

### Design Principles

- **Simplicity**: Extremely simple data model compared to RDBMS
- **Speed**: Often in-memory, O(1) retrieval operations
- **Scalability**: Easy horizontal scaling with eventual consistency

### Use Cases

- **Data Science**: Experiment results storage, feature storage, model monitoring
- **Software Engineering**: Session information, user profiles, shopping carts, caching

# 9. Redis - Key-Value Database

- Open-source, in-memory database
- High performance (>100,000 SET ops/second)
- Supports data durability through snapshots or append-only journals
- Simple lookup by key only (no secondary indexes)

## Data Types

- **Strings**: Basic text/binary data
- **Lists**: Linked lists for queues/stacks
- **Sets**: Unique unordered string collections
- **Hashes**: Field-value collections
- **Sorted Sets**: Sets with scores for ordering
- **Specialized Types**: Geospatial data, JSON, etc.

## Common Operations

- Basic CRUD: SET, GET, EXISTS, DEL
- Atomic counters: INCR, INCRBY, DECR, DECRBY
- List operations: LPUSH, RPUSH, LPOP, RPOP, LLEN, LRANGE
- Set operations: SADD, SCARD, SISMEMBER, SINTER, SDIFF
- Hash operations: HSET, HGET, HGETALL