DTS207TC Database Development and Design

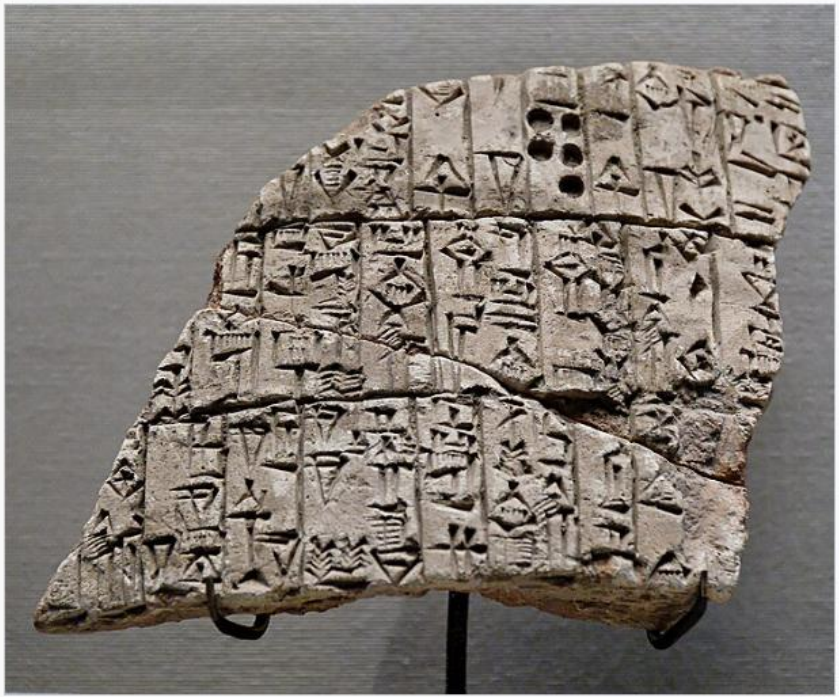
Lec 13 Review

Di Zhang, Autumn 2025

Outline

- Review of BPlus-Tree
- Review of Transaction

Clay



Fragment of an inscripted clay cone of Urukagina

File system

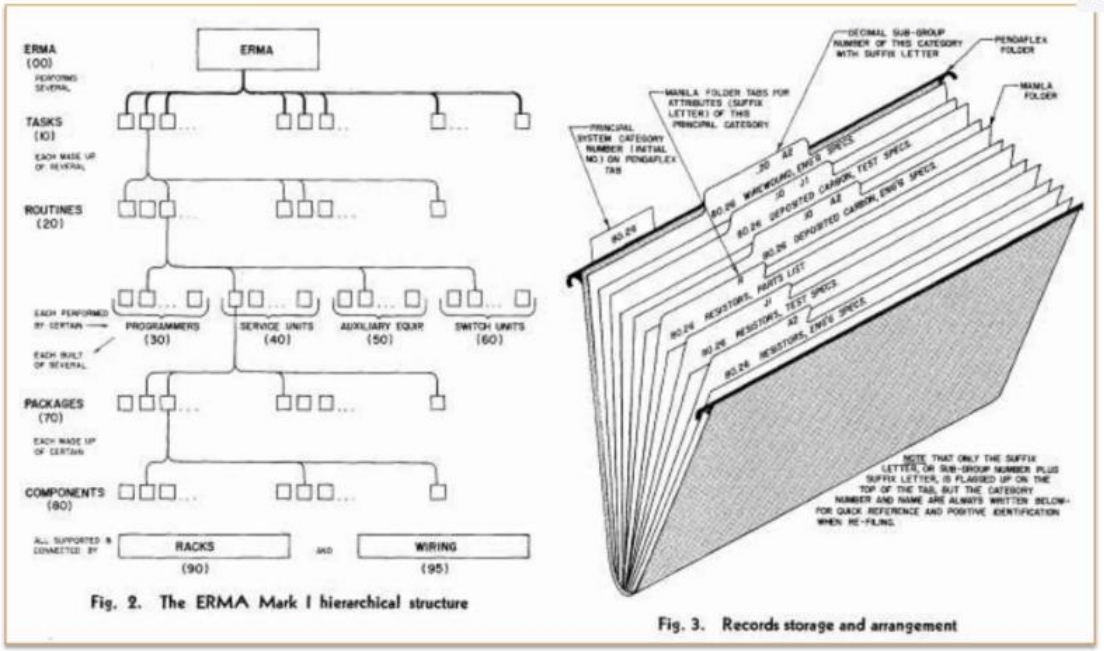
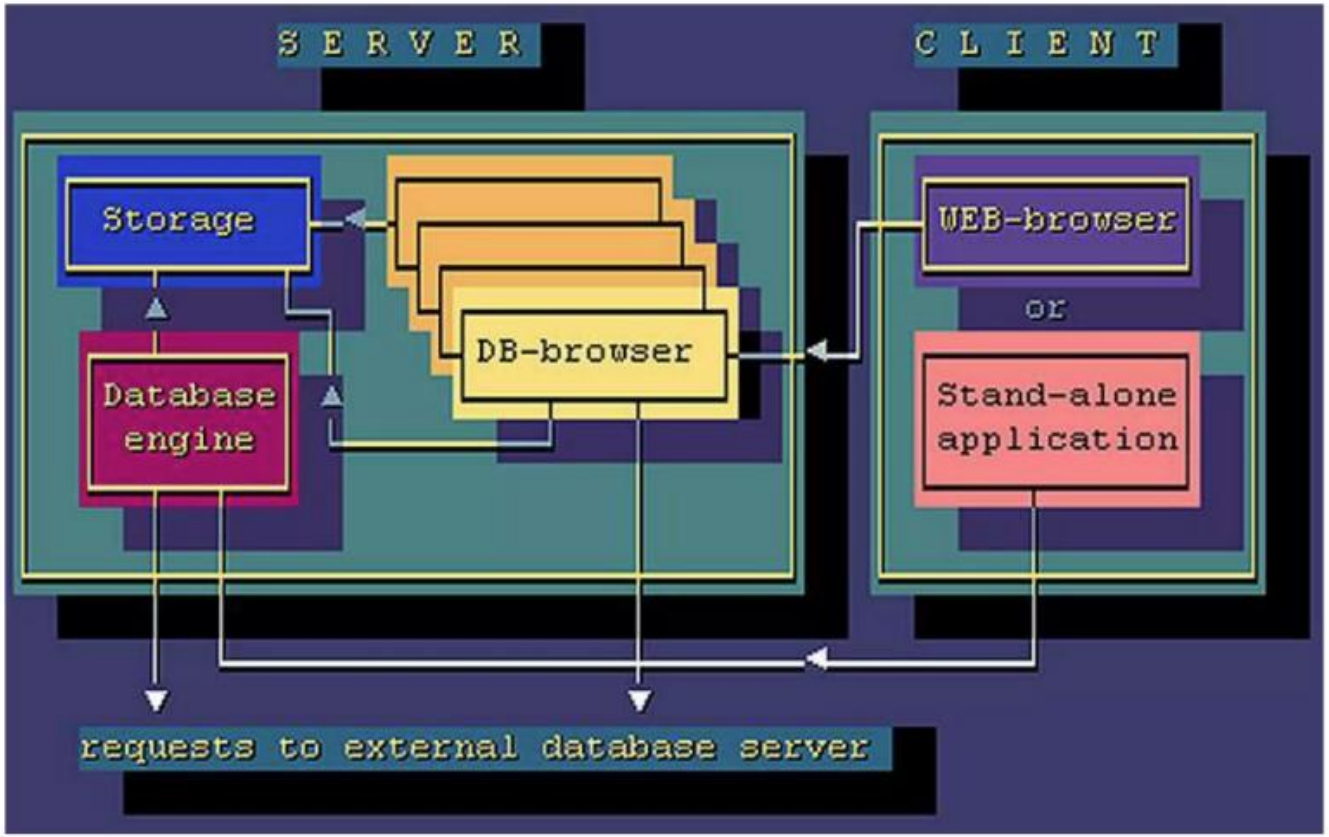


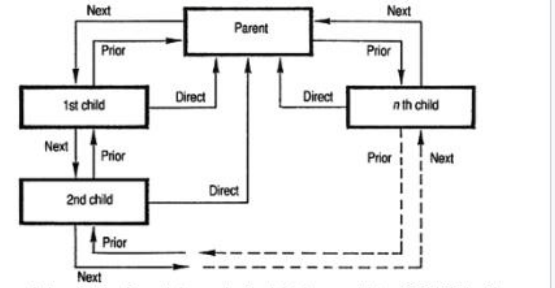
Fig. 2. The ERMA Mark I hierarchical structure

Fig. 3. Records storage and arrangement

DBMS


SERVER
CLIENT
Storage
DB-browser
Database engine
WEB-browser
or
Stand-alone application
requests to external database server

Navigational



A closed chain of records in a navigational database model (e.g. CODASYL), with ***next pointers***, ***prior pointers*** and ***direct pointers*** provided by keys in the various records.

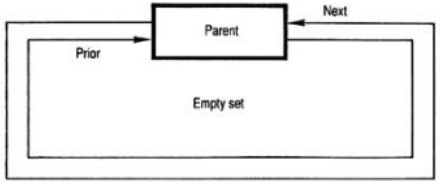


Illustration of an ***empty set***

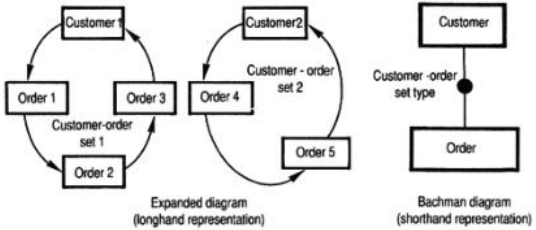
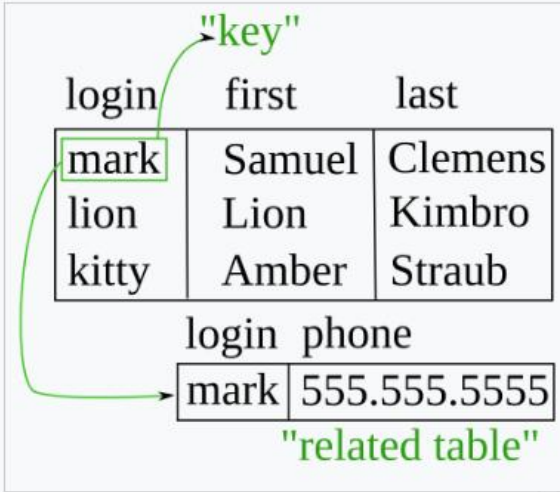


Illustration of a set type using a ***Bachman diagram***

The record set, basic structure of navigational (e.g. CODASYL) databse model. A set consists of one parent record (also called "the owner"), and n child records (also called members records)

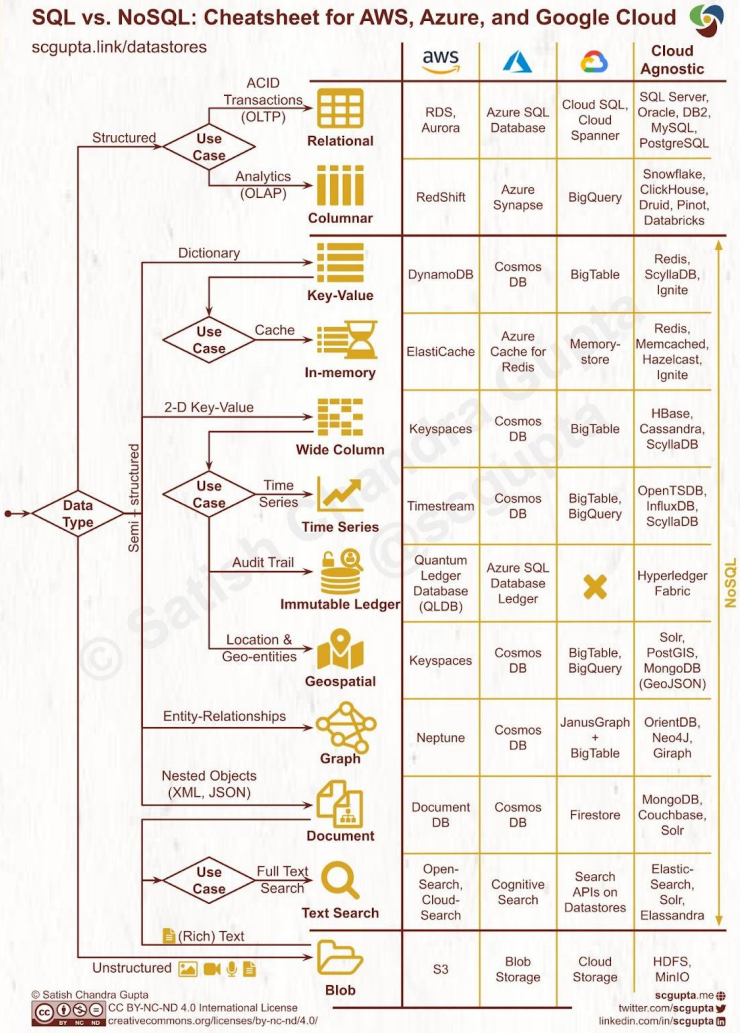
Basic structure of navigational CODASYL database model

Relational

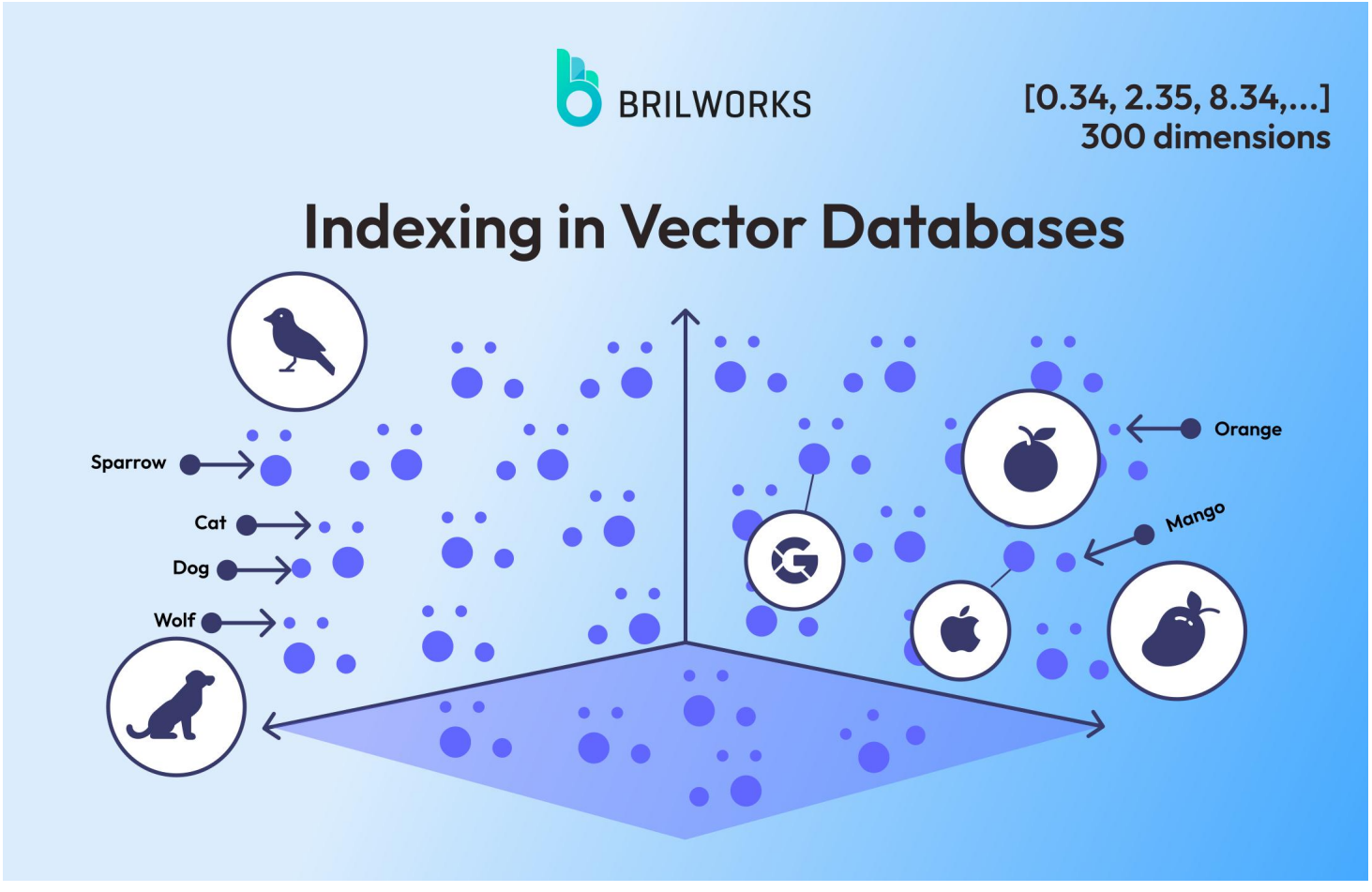


In the relational model, records are "linked" using virtual keys not stored in the database but defined as needed between the data contained in the records.

NoSQL


SQL vs. NoSQL: Cheatsheet for AWS, Azure, and Google Cloud
scgupta.link/datastores
aws
Cloud Agnostic
Data Type
Structured
Use Case
ACID Transactions (OLTP)
Relational
RDS, Aurora
Azure SQL Database
Cloud SQL, Cloud Spanner
SQL Server, Oracle, DB2, MySQL, PostgreSQL
Analytics (OLAP)
Columnar
RedShift
Azure Synapse
BigQuery
Snowflake, ClickHouse, Druid, Pinot, Databricks
Semi-structured
Dictionary
Key-Value
DynamoDB
Cosmos DB
BigTable
Redis, ScyllaDB, Ignite
Use Case
Cache
In-memory
ElastiCache
Azure Cache for Redis
Memory-store
Redis, Memcached, Hazelcast, Ignite
2-D Key-Value
Wide Column
Keyspaces
Cosmos DB
BigTable
HBase, Cassandra, ScyllaDB
Use Case
Time Series
Time Series
Timestream
Cosmos DB
BigTable, BigQuery
OpenTSDB, InfluxDB, ScyllaDB
Audit Trail
Immutable Ledger
Quantum Ledger Database (QLDB)
Azure SQL Database Ledger
Hyperledger Fabric
Location & Geo-entities
Geospatial
Keyspaces
Cosmos DB
BigTable, BigQuery
Solr, PostGIS, MongoDB (GeoJSON)
Entity-Relationships
Graph
Neptune
Cosmos DB
JanusGraph + BigTable
OrientDB, Neo4J, Giraph
Nested Objects (XML, JSON)
Document
Document DB
Cosmos DB
Firestore
MongoDB, Couchbase, Solr
Use Case
Full Text Search
Text Search
Open-Search, Cloud-Search
Cognitive Search
Search APIs on Datastores
Elastic-Search, Solr, Elassandra
NoSQL
Unstructured
(Rich) Text
Blob
S3
Blob Storage
Cloud Storage
HDFS, MinIO
© Satish Chandra Gupta
CC BY-NC-ND 4.0 International License
creativecommons.org/licenses/by-nc-nd/4.0/
scgupta.me
twitter.com/scgupta
linkedin.com/in/scgupta

Vector


BRILWORKS
[0.34, 2.35, 8.34,...]
300 dimensions
Indexing in Vector Databases
Sparrow
Cat
Dog
Wolf
Orange
Mango

Three Core Index Structures

- Sorted Array
- Tree-based Index
- Hash-based Index

Sorted Array Index

- Data Structure
  - Stores data ordered by key value (e.g., ascending)
  - Typically implemented using arrays
- Operation Complexity
  - Search: O(log n) — binary search
  - Insert/Delete: O(n) — requires shifting elements
- Advantages
  - Efficient for range queries
  - Compact storage, cache-friendly
- Disadvantages
  - High update cost
  - Static structure, requires pre-allocated space
- Use Cases
  - Read-only or rarely updated data
  - Frequent range queries required

Tree-based Index (e.g., B+ Tree)

- Data Structure
  - Multi-way balanced search tree
  - Leaf nodes form a sorted linked list
- Operation Complexity
  - Search/Insert/Delete: O(log n)
  - Tree height is usually small (3–4 levels)
- Advantages
  - Supports efficient range queries (via leaf node list)
  - Dynamically balanced, suitable for frequent updates
  - Supports partial match queries
- Disadvantages
  - Higher storage overhead (pointers, etc.)
  - May involve more random I/O
- Typical Applications
  - Default database index (e.g., MySQL InnoDB)
  - File system indexing

Hash-based Index

- Data Structure
  - Hash table + buckets
  - Hash function maps keys to buckets
- Operation Complexity
  - Average case: O(1) — direct access
  - Worst case: O(n) — all keys collide to one bucket
- Advantages
  - Extremely fast for equality queries
  - Simple and efficient
- Disadvantages
  - Does not support range queries
  - Hash function design is critical
  - Dynamic resizing can be costly
- Use Cases
  - Equality-only queries
  - In-memory databases or caches
  - Join operations (hash join)

Comparison Summary

Xi'an Jiaotong-Liverpool University 西交利物浦大學

Feature	Sorted Array	Tree (B+ Tree)	Hash
Search Complexity	O(log n)	O(log n)	O(1)
Insert/Delete Complexity	O(n)	O(log n)	O(1)
Range Queries	Excellent	Excellent	Not Supported
Equality Queries	Good	Good	Very Fast
Update Friendliness	Poor	Good	Good
Storage Overhead	Low	Medium	Variable
Typical Applications	Static datasets	General-purpose database indexing	In-memory tables, caches

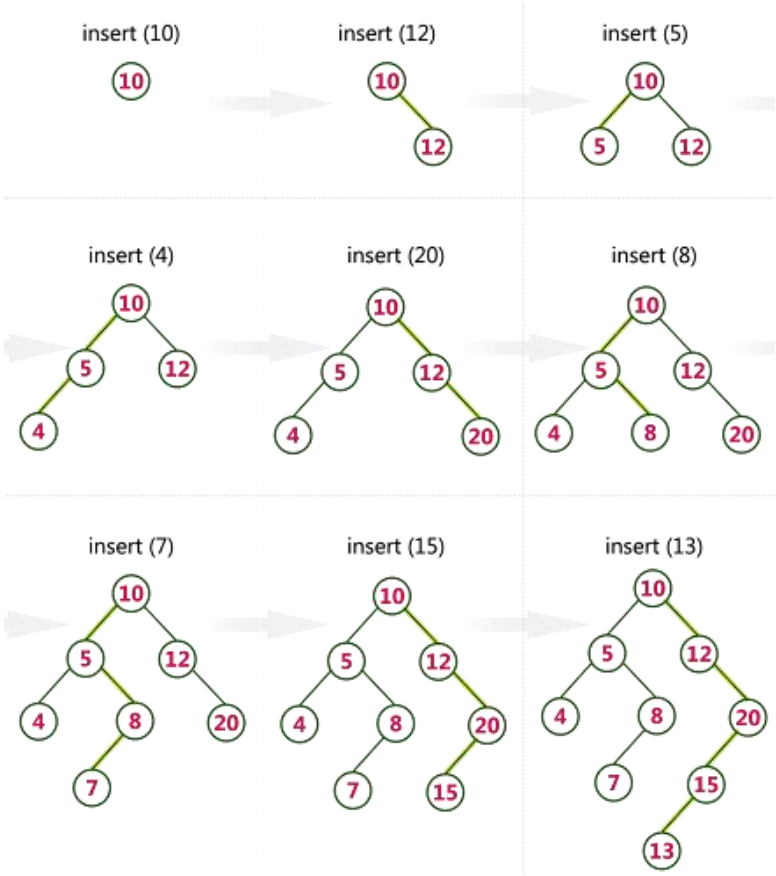
Problem Context

- Core Challenges:
  - Increasing data volume → Need for efficient data structures
  - High disk I/O cost → Reduce disk access frequency
  - Dynamic data operations → Maintain balance and performance

- Evolution Drivers:
  - Balance maintenance
  - Disk access optimization
  - Range query support

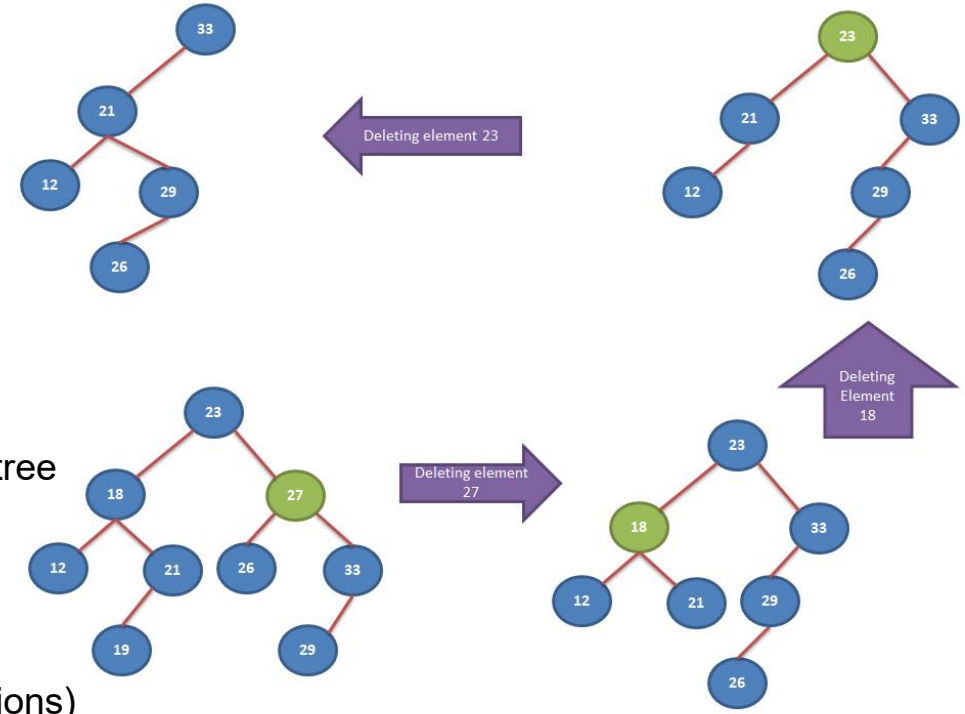
Binary Search Tree (BST)

- Basic Structure
- Property: Left subtree < Node < Right subtree
- Insert Operation
- Steps:
  - Start comparison from root node
  - If less than current node → Go to left subtree
  - If greater than current node → Go to right subtree
  - Find empty position → Insert new node
- Complexity:
  - Best: O(log n) (when balanced)
  - Worst: O(n) (degenerates to linked list)



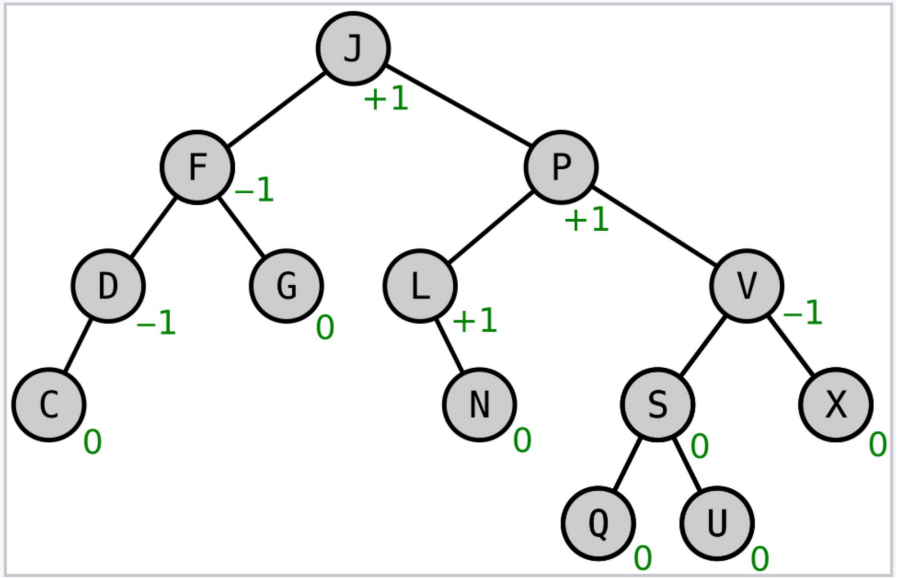
Delete Operation

- Three Cases:
  - Leaf node: Direct deletion
  - One child node: Replace with child node
  - Two child nodes:
    - Find minimum node in right subtree
    - Replace the node to be deleted
    - Delete the minimum node from right subtree
- Complexity: O(h), where h is tree height
- BST Problems
  - May degenerate into linked list (O(n) operations)
  - Imbalance leads to performance degradation



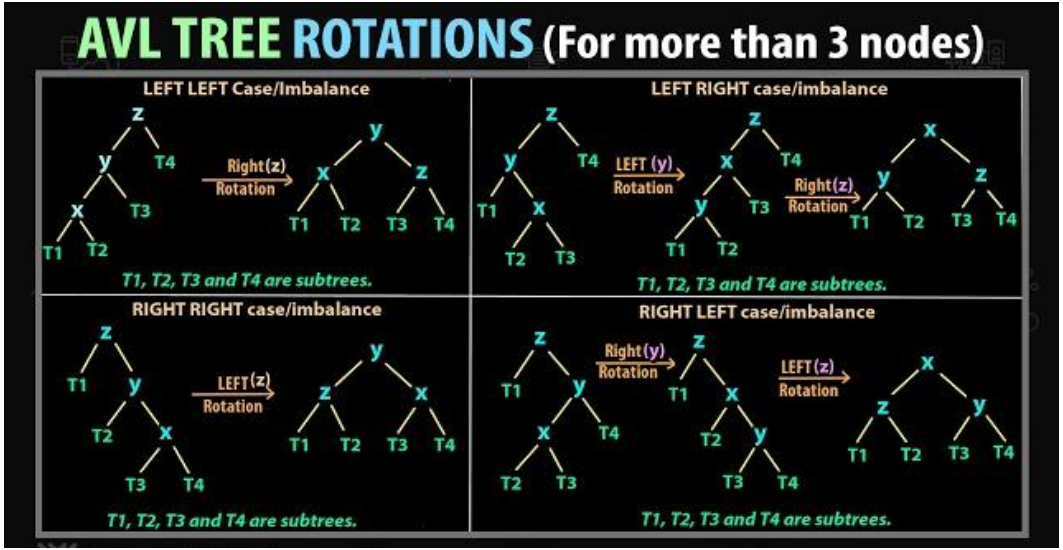
AVL Tree

- Self-Balancing BST
- Core: Height difference between left and right subtrees ≤ 1 for each node
- Balance Factor: height(left) - height(right) ∈ {-1, 0, 1}



Self-Balancing BST

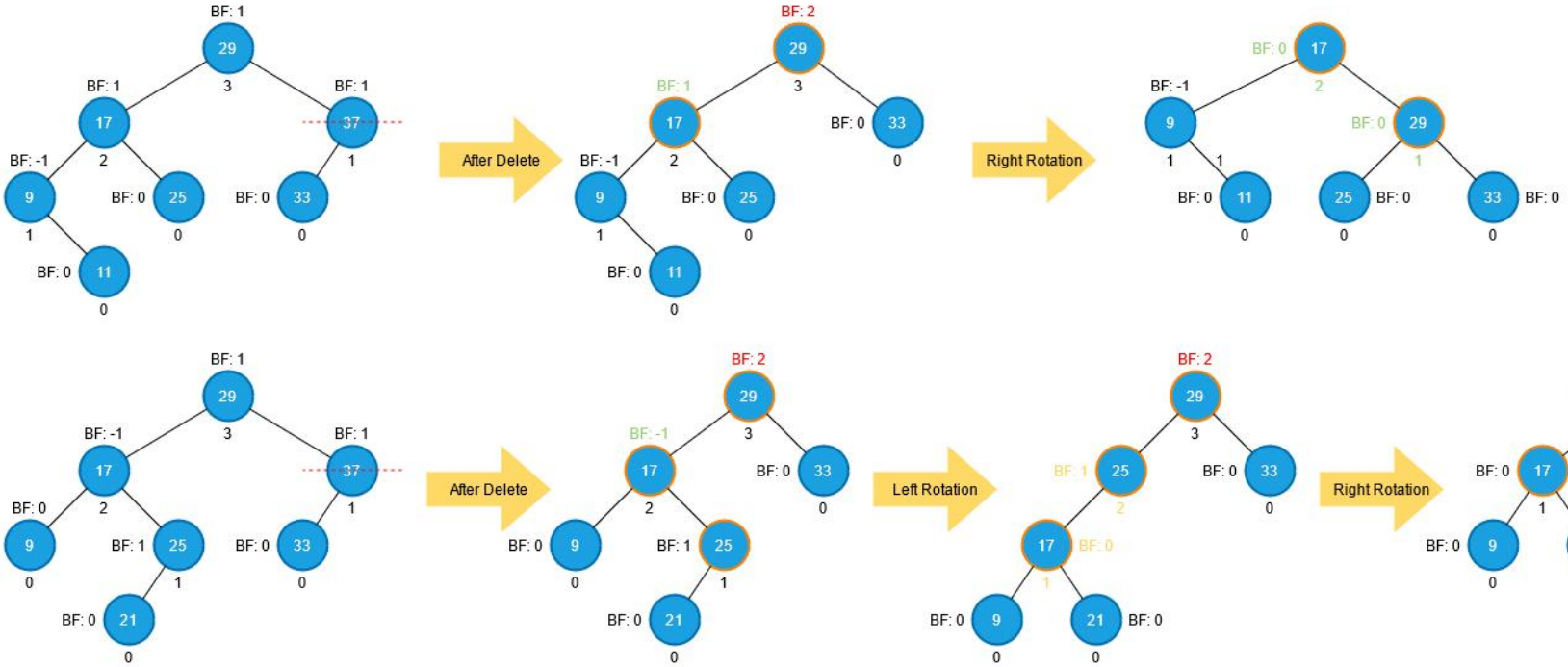
- Core: Height difference between left and right subtrees ≤ 1 for each node
- Balance Factor: height(left) - height(right) ∈ {-1, 0, 1}
- Insert Operation
  - Standard BST insertion
  - Update heights of ancestor nodes
  - Check balance factors
  - If unbalanced, perform rotations:
    - Left-Left: Right rotation
    - Right-Right: Left rotation
    - Left-Right: Left rotation + Right rotation
    - Right-Left: Right rotation + Left rotation
- Complexity: O(log n) (always balanced)



Delete Operation

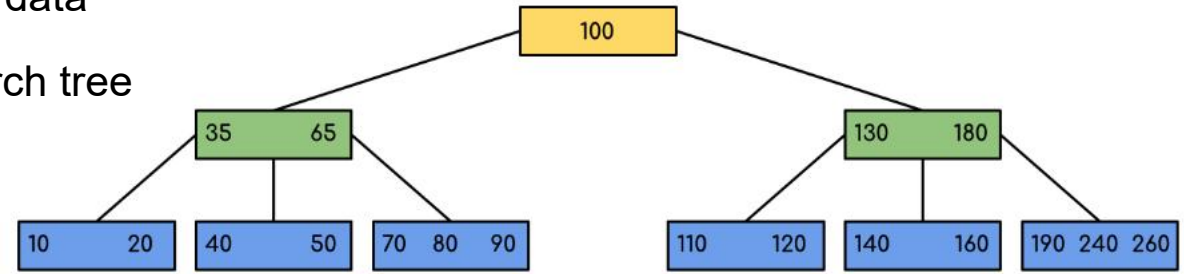
- Steps:
  - Standard BST deletion
  - Update heights upward from deletion point
  - Check balance factor for each ancestor
  - Perform rotations if needed (up to O(log n) times)
- Complexity: O(log n)

- AVL Advantages and Limitations
  - Advantages: Strict balance, excellent query performance
  - Limitations: Frequent rotations, high insertion/deletion cost


BF: 1
29
3
BF: 1
17
2
BF: 1
37
1
BF: -1
9
1
BF: 0
25
0
BF: 0
33
0
BF: 0
11
0
After Delete
BF: 2
29
3
BF: 1
17
2
BF: 0
33
0
BF: -1
9
1
BF: 0
25
0
BF: 0
11
0
Right Rotation
BF: 0
17
2
BF: -1
9
1
BF: 0
29
1
BF: 0
11
0
25
BF: 0
0
33
BF: 0
0
BF: 1
29
3
BF: -1
17
2
BF: 1
37
1
BF: 0
9
0
BF: 1
25
1
BF: 0
33
0
BF: 0
21
0
After Delete
BF: 2
29
3
BF: -1
17
2
BF: 0
33
0
BF: 0
9
0
BF: 1
25
1
BF: 0
21
0
Left Rotation
BF: 2
29
3
BF: 1
25
2
BF: 0
33
0
17
BF: 0
1
BF: 0
9
0
21
BF: 0
0
Right Rotation
BF: 0
25
2
BF: 0
17
1
BF: -1
29
1
BF: 0
9
0
21
BF: 0
0
BF: 0
33
0

B-tree

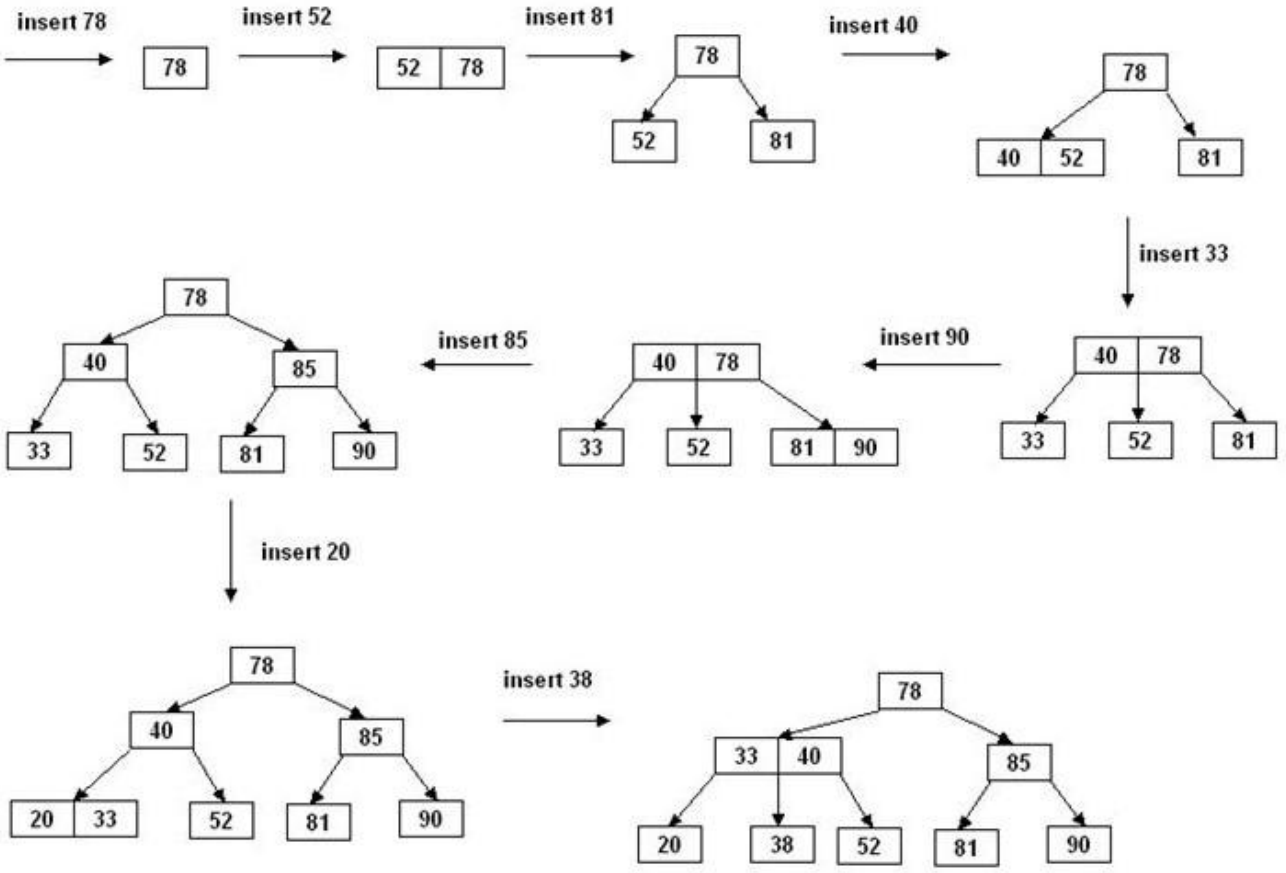
- Design Goals
  - Reduce disk I/O
  - Suitable for large-scale data
  - Multiway balanced search tree
- Structural Properties
- B-tree of order m:
  - Root node: 2 to m children
  - Internal nodes: ⌈ m/2⌉ to m children
  - All leaves at same level
  - Node contains k keys and k+1 child pointers



Insert Operation

- Steps:
  - Find insertion position (leaf node)
  - If leaf not full → Insert directly
  - If leaf is full:
    - Split node (middle key moves up)
    - Recursively split upward
    - May increase tree height
- Complexity: O(log_m n) disk accesses

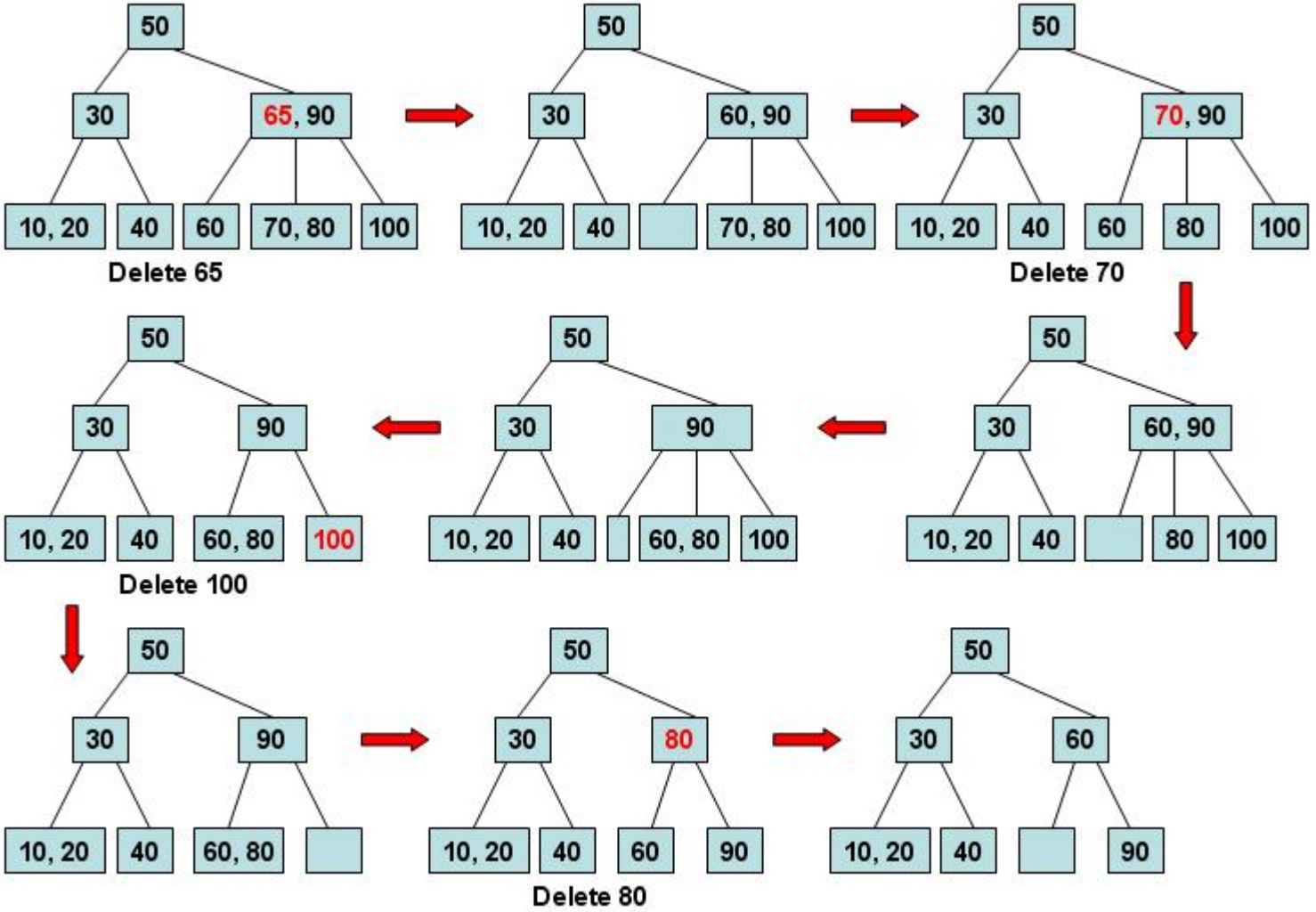
- Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



Delete Operation

- Complex Case Handling:
- Delete from leaf:
  - Direct deletion (if at least ⌈ m/2⌉ -1 keys remain)
  - Otherwise borrow key from sibling
  - Or merge with sibling
- Delete from internal node:
  - Replace with predecessor/successor
  - Recursively delete predecessor/successor

- Complexity: O(log_m n)


50
30
65, 90
10, 20
40
60
70, 80
100
Delete 65
50
30
60, 90
10, 20
40
70, 80
100
50
30
70, 90
10, 20
40
60
80
100
Delete 70
50
30
60, 90
10, 20
40
80
100
50
30
90
10, 20
40
60, 80
100
Delete 100
50
30
90
10, 20
40
60, 80
50
30
80
10, 20
40
60
90
Delete 80
50
30
60
10, 20
40
90

B+ Tree

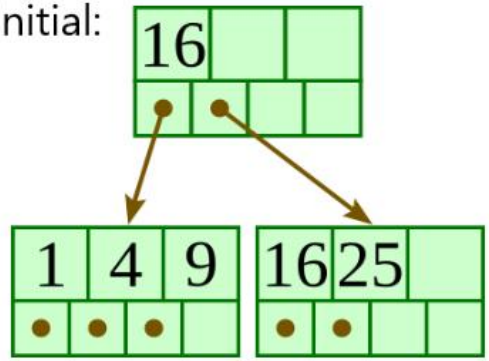
- Database Indexing Standard
- Key Differences from B-tree:
  - Data stored only in leaf nodes
  - Leaf nodes form linked list
  - Internal nodes store only keys (no data)
- Structural Advantages
  - Faster range queries
  - Higher fanout (more child nodes)
  - Optimized sequential access

Insert Operation

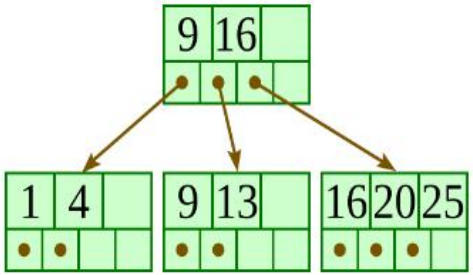
- Steps:
- Find corresponding leaf node
- If leaf not full → Insert in order
- If leaf is full:
  - Split leaf node
  - Copy middle key to parent
  - Update leaf linked list pointers

- Complexity: O(log_f n), where f is fanout

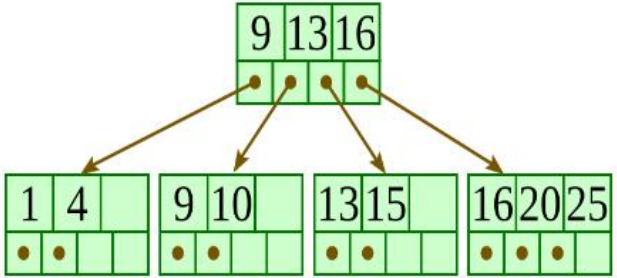
Initial:



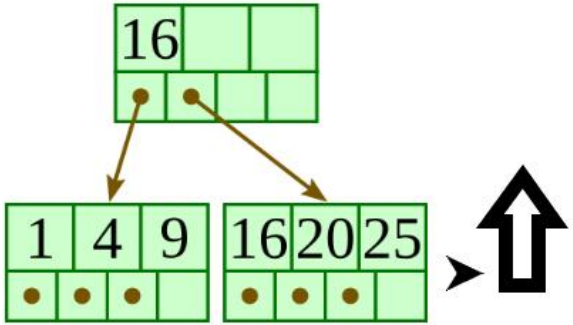
Insert 13:



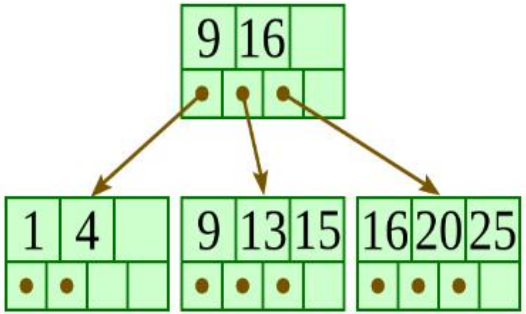
Insert 10:



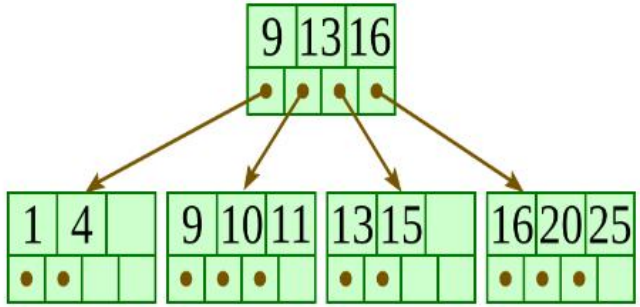
Insert 20:



Insert 15:



Insert 11:

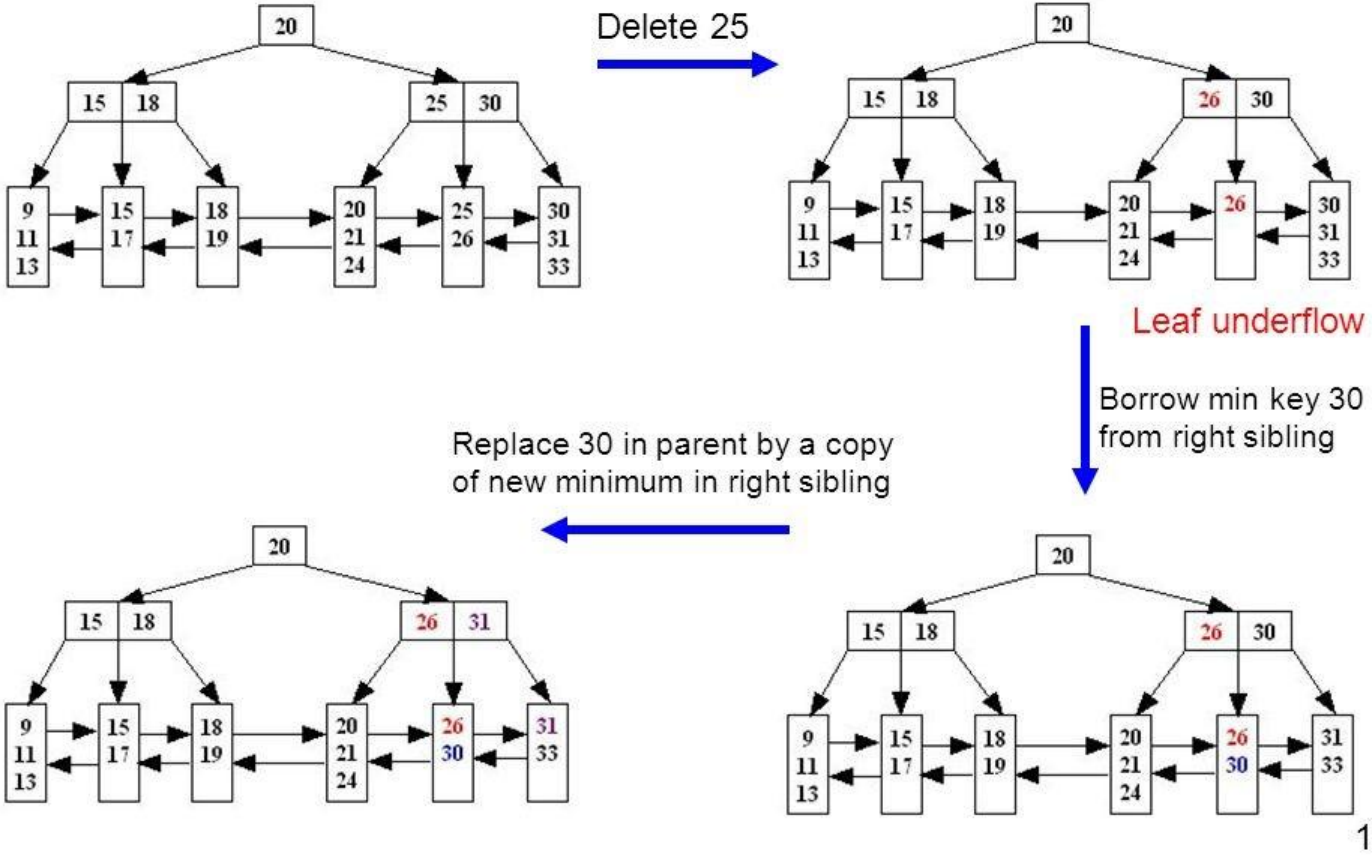


Delete Operation

- Steps:
  - Delete key from leaf node
  - If leaf node underflows:
    - Attempt redistribution from sibling
    - Or merge with sibling node
  - Update parent node keys
- Note: Deletion may propagate to root

Illustration when node has too few keys while sibling has extra keys

Example: Delete 25 from the following B+ tree of order M = 3 and L = 3



Insert/Delete Complexity Comparison

- Time Complexity

Structure	Insert	Delete	Balance Maintenance
BST	O(n)-O(log n)	O(n)-O(log n)	None
AVL	O(log n)	O(log n)	Strict, frequent
B-tree	O(log_m n)	O(log_m n)	Node split/merge
B+ Tree	O(log_f n)	O(log_f n)	Similar to B-tree

- Disk I/O Complexity

Structure	Insert	Delete	Practical Consideration
BST/AVL	High	High	Suitable for memory
B-tree	O(log_m n)	O(log_m n)	m typically 100-200
B+ Tree	O(log_f n)	O(log_f n)	f typically 200-500

- Key Insight: B-tree/B+ Tree have larger log base, significantly reducing disk access

Why Concurrency Control?

Xi'an Jiaotong-Liverpool University 西交利物浦大學

- Advantages of Concurrent Transactions
  - ☑Increased System Throughput: Full utilization of system resources
  - ☑Reduced Wait Times: Users don't wait sequentially
  - ☑Improved Resource Utilization: Parallel CPU and I/O operations
- Challenges Brought by Concurrency
  - ❌Risk of Data Inconsistency
  - ❌Violation of Transaction ACID Properties
  - ❌Unpredictable Results
- The Importance of Isolation
  - "Isolation ensures concurrently executing transactions appear as if they executed serially"

Problem One: Dirty Read

Xi'an Jiaotong-Liverpool University 西交利物浦大学

- What is a Dirty Read?
  - Definition: Transaction A reads data modified by Transaction B that is uncommitted
  - Key Characteristic: The read data may be rolled back, never truly existing
- Scenario Simulation

```
-- Transaction A (Transfer)          -- Transaction B (Query Balance)
BEGIN;                               BEGIN;
UPDATE accounts                      SELECT balance
SET balance = balance - 100          FROM accounts
WHERE id = 1;                        WHERE id = 1;
                                     -- Reads uncommitted balance here!
-- If ROLLBACK here...               COMMIT;
ROLLBACK;
```

- Result: Transaction B sees data that never truly happened
- Harm Analysis
  - Business decisions based on erroneous data
  - Complete breakdown of data consistency
  - Violates the basic atomicity of transactions

Problem Two: Non-repeatable Read

- What is a Non-repeatable Read?
  - Definition: Within the same transaction, two reads of the same data yield different results
  - Key Characteristic: Data is modified and committed by another transaction during the transaction's execution
- Scenario Simulation

```
-- Transaction A (Bank Reconciliation)  -- Transaction B (User Payment)
BEGIN;                                  BEGIN;
SELECT balance                          UPDATE accounts
FROM accounts                           SET balance = balance - 50
WHERE id = 1;                           WHERE id = 1;
-- Balance: 1000                        COMMIT;

-- Some time later...
SELECT balance                          -- Transaction B already committed!
FROM accounts
WHERE id = 1;
-- Balance: 950 (Different from first read!)
COMMIT;
```

- Result: Transaction A sees external changes during its own execution
- Harm Analysis
  - Logical judgments within the same transaction become invalid
  - Inaccurate statistical calculations
  - Violates business requirements for "repeatable reads"

Problem Three: Phantom Read

- What is a Phantom Read?
  - Definition: Within the same transaction, two identical range queries return a different number of rows
  - Key Characteristic: Other transactions insert or delete data within the range
- Scenario Simulation

```
-- Transaction A (Count Dept. Employees)
BEGIN;
SELECT COUNT(*)
FROM employees
WHERE dept_id = 10;
-- Result: 5 people

SELECT COUNT(*)
FROM employees
WHERE dept_id = 10;
-- Result: 6 people (One extra person!)
COMMIT;
```

```
-- Transaction B (HR Operation)
BEGIN;
INSERT INTO employees
(dept_id, name)
VALUES (10, 'Zhang San');
COMMIT;
```

- Result: Transaction A sees rows that "magically appear" or "mysteriously disappear"
- Difference from Non-repeatable Read

Aspect	Non-repeatable Read	Phantom Read
Target	Existing rows	New rows
Operation	UPDATE	INSERT/DELETE
Focus	Data content change	Data set change

Problem Four: Lost Update

Xi'an Jiaotong-Liverpool University 西交利物浦大學

- What is a Lost Update?
  - Definition: Two transactions modify the same data simultaneously, with the later commit overwriting the earlier one
  - Key Characteristic: The earlier commit's modifications are "silently" lost
- Scenario Simulation
- Final Result: quantity = 8 (Transaction A's deduction is completely o
- Two Types of Lost Update
  - First Lost Update (Rollback Overwrite)
    - Transaction A overwrites Transaction B's update upon rollba
    - Mostly solved in modern databases
  - Second Lost Update (Commit Overwrite)
    - Later-committing transaction overwrites earlier-committing transaction
    - Most common business problem

```
-- Transaction A (Inventory Deduction)  -- Transaction B (Inventory Deduction)
BEGIN;                                   BEGIN;
SELECT quantity                          SELECT quantity
FROM products                            FROM products
WHERE id = 100;                          WHERE id = 100;
-- quantity = 10                         -- quantity = 10

UPDATE products                          UPDATE products
SET quantity = 9                         SET quantity = 8
WHERE id = 100;                          WHERE id = 100;
COMMIT;                                  COMMIT;
```

Solution Framework: Isolation Levels

- Problems Solved at Each Level

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read	Performance
Read Uncommitted	✗ Allowed	✗ Allowed	✗ Allowed	★★★★★
Read Committed	☑ Prevented	✗ Allowed	✗ Allowed	★★★★
Repeatable Read	☑ Prevented	☑ Prevented	✗ Allowed*	★★★
Serializable	☑ Prevented	☑ Prevented	☑ Prevented	★

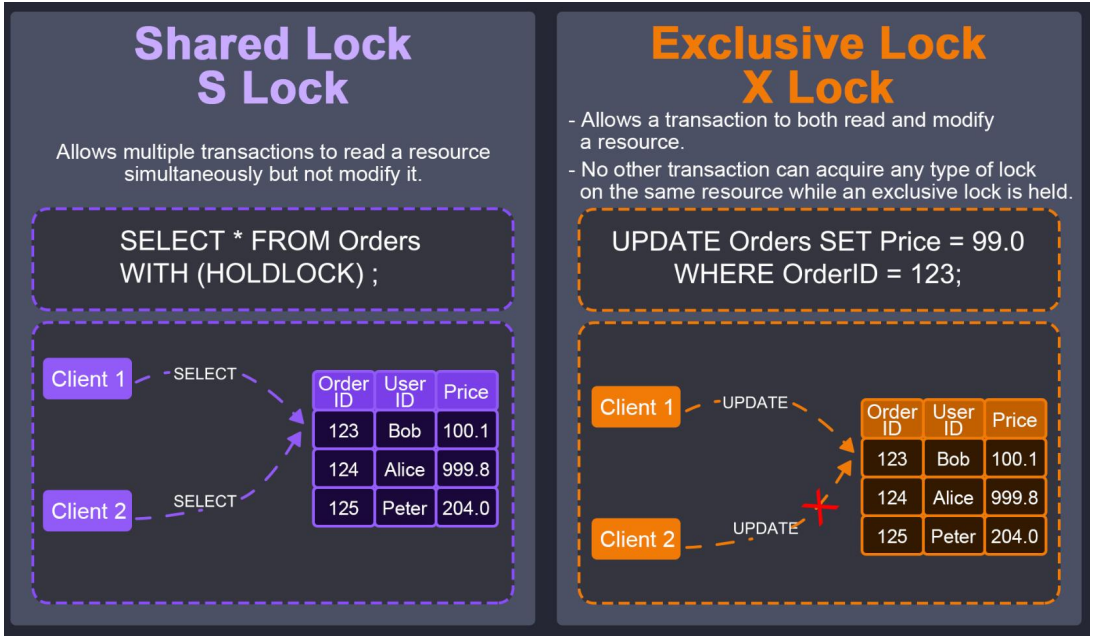
- *Note: Most databases also prevent phantom reads at Repeatable Read level (e.g., MySQL InnoDB)

Summary and Best Practices

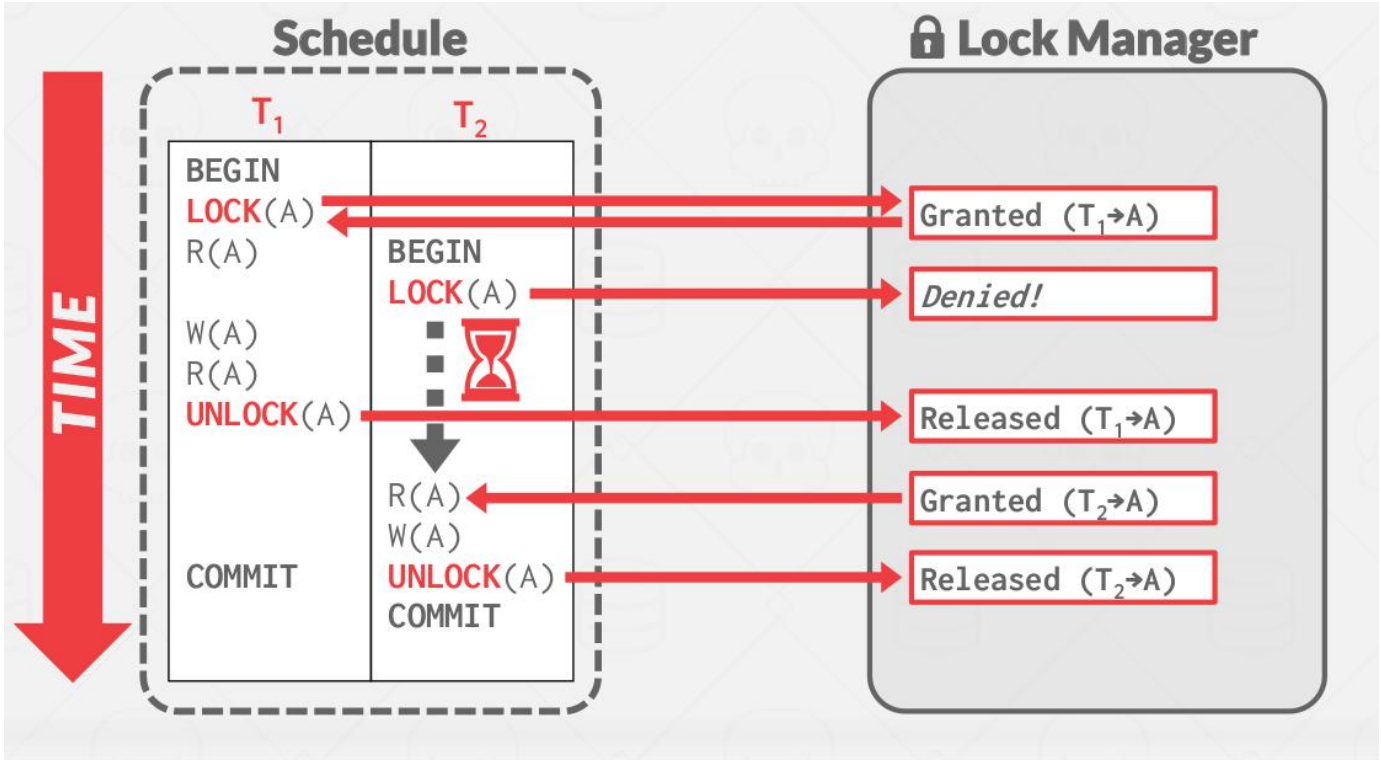
- Core Principles Review
  - No Perfect Solution: Trade-off between isolation and performance
  - Understand Business Requirements: Choose isolation level based on application scenario
  - Know Database Characteristics: Different databases have different implementations
- Isolation Level Selection Guide
  - Data Warehouse/Reporting Systems → READ UNCOMMITTED or READ COMMITTED
  - Most OLTP Applications → READ COMMITTED (balanced choice)
  - Financial/Accounting Systems → REPEATABLE READ or SERIALIZABLE
  - Requiring Highest Consistency → SERIALIZABLE (consider performance impact)

Core Mechanism: Locks*

- Isolation is primarily achieved through locking:
  - Shared Lock (S Lock): Read lock, allows concurrent reads
  - Exclusive Lock (X Lock): Write lock, blocks all other access



- Two-Phase Locking (2PL): Acquire phase → release phase



Core Mechanism: MVCC (Multi-Version Concurrency Control)*

- Each write creates a new row version
- Reads obtain a "snapshot," so they don't block writes
- Write-write conflicts still require locks
- Used in PostgreSQL, MySQL InnoDB, Oracle, etc.

- Pros: Read–write operations rarely block
- Cons: Old versions must be cleaned up (GC / Vacuum)

How Read Uncommitted Is Implemented*

- Characteristics: Allows reading uncommitted data
- Implementation:
  - Reads do not take S locks
  - Reads may view the latest version directly
  - Writes take X locks but do not block reads

How Read Committed Works*

- Goal: Prevent Dirty Reads
- Implementation:
  - Each SELECT reads the latest committed version (MVCC snapshot)
  - A new snapshot is taken for every statement
  - Writes hold X locks until commit
- Outcome:
  - No Dirty Reads
  - But Non-repeatable Reads can occur

Repeatable Read Implementation*

- Goal: Data read once remains stable
- Implementation varies by DBMS:
  - MySQL InnoDB (Important Case)
  - Snapshot created at transaction start
  - All SELECTs read from the same snapshot

How Serializable Is Implemented?

- Strictest level
- Common approaches:
- Strict Two-Phase Locking (2PL)
  - All reads/writes acquire locks
- Prevents all anomalies

- But significantly reduces concurrency

Bye~