Databases and Information Systems

Michael Hüppe

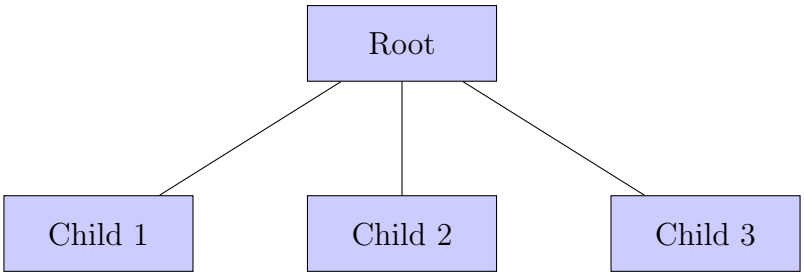
June 4, 2025

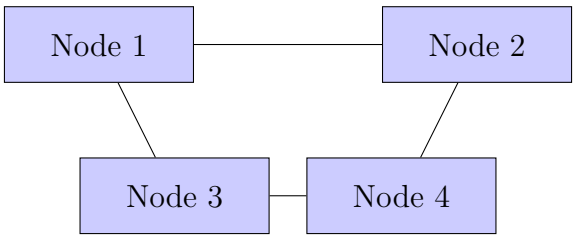
Contents

Chapter 1

Architectures of Database Systems

1.1 Model Types





1.1.1 Hierarchical Model

- Tree-like structure with one parent per child
- Good performance for one-to-many relationships, but poor flexibility
- each record has only one parent record
- a record is a collection of fields where each filed contains one value
- the fields of a record are defined by its type
- used only in few database systems today, e.g. in IBM Information management System

1.1.2 Network Model

- Allows many-to-many relationships with interconnected nodes
- More flexible than hierarchical models but more complex to manage

- most general form of a data structure → No limitation to the links between records
- enables complex data structures but only simple operations
- widely replaced by the relational model

1.1.3 Relational Model

- Data is stored in tables with rows (records) and columns (fields)
- Uses primary and foreign keys to establish relationships between tables
- created to be simple
- became more popular than the network model with increased computing power
- postulates independence of the language and implementation
- originally implementations of the relational model were too slow for productive use, but with growing computing power the relational model became the standard abstraction model for database systems
- declarative queries, the use of values, and set orientation make it easy to use compared to the network model which uses pointers and is record oriented

1.2 Building Relational Database Systems

1.2.1 Monolithic Approach

1. **Single Codebase**: All components-query processor, storage manager, transaction manager, etc. -reside in one codebase
2. **Tight coupling**: Components depend heavily on each other, making it hard to change one without affecting other
3. **Centralized control**: All functionality is in one process, internal calls are fast (no network latency)
4. **Difficult Maintenance & Scalability**: Updating or scaling individual components is harder than in modular systems

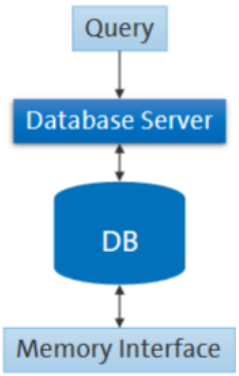


Figure 1.1: Monolithic Approach

1.2.2 The 5 Layer Model

1. **Data System Layer**
    - SQL parsing, query planning, optimization, execution
    - semantic understanding of data (tables, schemas)
    - **Output**: A logical query execution plan
    - **Manages**: Relations, views indexes
2. **Access System Layer**:
    - Logical access to data: retrieval and update
    - navigation, use of logical pointers and index structures
    - manages record format and system catalogs
3. **Storage System Layer**:
    - Physical data organization on disk
    - record management, free space access paths
    - manages tress, hash structures
4. **Buffer & File System Layer**:
    - Manages in memory-pages
    - page replacement strategies
    - interacts with file system and hardware
    - handles logging and recovery and durability

1.3 Record management

- Records stored on fixed-length pages (records can vary in length)
- Addressing via Tuple Identifier (TID) = Page ID, Index in Page
- Heap Management: Keeps track of free space for placing new records

1.4 Organization of Pages

- Pages linked in doubly-linked list
- Each page contains: Header info (page ID; previous/next page), Metadata (free space, record type)
- Pages support efficient access and stable record addressing even during data migration

1.5 Tuple Identifier (TIDs)

- stable address of a record
- record updates:
  - if too large: Moved to another page, old page stores pointer
  - if small: reorganized in place no change to TID
- ensures address consistency even after migration or deletion

1.6 Use of Indexes

- prevent full scans → use indexes for faster access
- Types of access:
  - Sequential scan
  - sorted access
  - direct (by key)
  - Navigational (related records)
- Requirements: Fast, direct, topology-preserving access

1.7 B-Tree and Index Classification

- Indexes are classified based on:
  - **Primary or Secondary**: based on uniqueness
  - **Simple or Multilevel**: one-level or hierarchical indexing
- Storage Structures:
  - Tree-structured: e.g., B-Tree
  - Sequential: e.g., sorted lists
  - Scattered: e.g., hash tables
- Access Methods:
  - Key comparison: e.g., B-Trees
  - Key transformation: e.g., hashing
- B-Tree features:
  - Reduces disk accesses by organizing keys hierarchically
  - Dynamic reorganization through page splits and merges
  - Supports direct access and range queries

1.8 Creating Indexes in SQL

- Syntax examples:
  - `CREATE INDEX my_index ON myTable(myAttr);`
  - `CREATE UNIQUE INDEX my_index ON myTable(myAttr);`
  - `CREATE INDEX my_index ON myTable(a + b * (c - 1), a, b);`
- Special options:
  - `INCLUDE`: add attributes to leaf nodes only
  - `[DIS]ALLOW REVERSE SCANS`: control scan direction
- Attribute order in composite indexes matters:
  - Index on (A, B) can be used for queries on A or A+B, not just B
- Not all features available in every DBMS → always check system documentation

Chapter 2

Exercise 2

2.1 Surrogate Keys

- artificial primary key that has no business meaning - it exists solely to uniquely identify records in a table
- **Example**: Auto-incrementing ID column (like SERIAL in PostgreSQL)
- they never change
- simpler joins (single-column integer keys are efficient)
- avoid exposing business data in URLs/APIs

2.2 Partitioning

When mapping object-oriented inheritance to relational tables, there are two main strategies:

2.2.1 Horizontal Partitioning (Table per Subclass)

- each subclass gets its own table containing all attributes (both inherited and specific)
- no parent table exits
- **Pros**:
  - No NULL values (houses don't need apartment fields)
  - clean separation of concerns
- **Cons**:
  - Duplicates shared attributes (e.g. city, postal code in both tables)
  - harder to query "all estates" → requires **UNION**

2.2.2 Vertical Partitioning (Table per Hierarchy)

- Single table for the entire hierarchy with:
  - a **type** column to distinguish subclasses (e.g. "House", "Apartment")

  - all possible attributes (many NULLs for subtype-specific fields)

- **Pros**:
  - Simple queries (no joins needed)
  - easy to get "all estates"
- **Cons**:
  - Many NULL values (wastes space)
  - No DB-level constraints to enforce subtype rules

Chapter 3

Exercise 5

3.1 Concepts

3.1.1 Deferred Writes (No-Force)

- When a client commits, you don't immediately write data to tdisk
- you defer it until the buffer is "too full"
- this mimics real-world behavior where disk writes are expensive

3.2 No Dirty Writes (No-Steal)

- You never write uncommited data to disk
- only data from committed transaction can be flushed
- this makes recovery simple (no undo needed)

3.3 Physical Logging

- You log the exact new state ofo page (called after-image)
- Every operation is logged before it's commited (Write Ahead Logging)

3.4 Crash Recovery (Redo Only)

- If the system crashes, you:
  - read the log
  - Find all commited transactions
  - Reapply their writes if needed

3.5 Why this matters

Real-world databases:

- Delay writes to optimize performance
- maintain logs for safety
- Use recovery algorithms (like ARIES) to bring the DB back to a consistent state after a crash

You're building a simplified version of that:

- No transaction rollback (no UNDO)
- just REDO commited changes
- Emphasizes correctness and robustness in concurrent enrionments