Functional Dependency and Closure

Functional Dependency (FD) is like a rule that says one piece of data (like a person's name) can tell you another piece of data (like their birthdate). If you know the rule applies consistently, then you know the birthdate for each name without fail.

Closure Set $F+$

The closure $F+$ is like knowing all the possible things you can find out if you know a starting piece of information under the rules of FD. For example, if knowing someone's name tells you their birthdate, and knowing their birthdate tells you their astrological sign, then knowing a name ultimately tells you both birthdate and astrological sign—that's the closure.
Example: If you have a functional dependency:

$$\text{Name} \rightarrow \text{Birthdate}$$

and another:

$$\text{Birthdate} \rightarrow \text{Astrological Sign}$$

Then the closure of Name would be:

$$\text{Name}+ = \{\text{Name, Birthdate, Astrological Sign}\}$$

Achieving Boyce-Codd Normal Form (BCNF)

BCNF is like organizing your closet so that every item has one and only one place where it belongs. A database is in BCNF if every piece of data can be uniquely identified by a 'key' (like a label for a shelf in your closet), and there's no confusion about where to find data or how it links to other data.

Relation Decomposition

To get a database into BCNF, you might need to 'decompose' it, which means breaking it into smaller, well-organized parts. It's like deciding to separate your socks, shirts, and pants into different drawers instead of having them all in one place.
Example: Suppose you have a table with courses and instructors, where the course code determines the instructor, and each instructor works in one department. This creates redundancy because the department can be determined by the course code.

$$\text{Course Code} \rightarrow \text{Instructor}$$

$$\text{Instructor} \rightarrow \text{Department}$$

To decompose into BCNF, you create separate tables:

- One for courses and instructors, where course code is a unique identifier.
- Another for instructors and departments, where instructor is a unique identifier.

This way, each piece of information is found in only one place, and your data is well-organized.

Relational Algebra Operations

Students Table: ID, Name, Age, Major.
Courses Table: CourseID, Name, Credits.

Basic Operations

Select (σ): $\sigma_{\text{Age}>21}(\text{Students})$.
Selects rows where age larger than 21 years.
Project (π): $\pi_{\text{Name, Major}}(\text{Students})$.
Selects the 'Name' and 'Major' columns from the 'Students' table.
Union ($\cup$): $\pi_{\text{Name}}(\text{Students}) \cup \pi_{\text{Name}}(\text{Courses})$.
Combines the 'Name' columns from 'Students' and 'Courses' tables.
Set Difference ($-$): $\pi_{\text{Name}}(\text{Students}) - \pi_{\text{Name}}(\text{Courses})$.
Rows in the first relation (Students) but not in the second (Courses).
Cartesian Product ($\times$): $\text{Students}_1 \times \text{Courses}_2$.
Creates pairs of every student with every course. Every $item_1$ goes with every $item_2$.
Rename (ρ): $\rho_{\text{Pupils(ID, StudentName, StudentAge, Major)}}(\text{Students})$.
Renames the 'Students' table to 'Pupils' and its attributes accordingly.
Intersection ($\cap$): $\pi_{\text{Name}}(\text{Students}) \cap \pi_{\text{Name}}(\text{Courses})$.
Finds names common to both 'Students' and 'Courses'.
Natural Join ($\bowtie$): Students $\bowtie$ Enrollments.
Joins 'Students' and 'Enrollments' on common attributes/columns.
Theta Join: Students $\bowtie_{\text{Students.Major = Courses.Name}}$ Courses.
Joins 'Students' and 'Courses' where the student's major matches the course name.
Division: Enrollments $\div \pi_{\text{CourseID}}(\sigma_{\text{Instructor = 'Dr. Smith'}}(\text{Courses}))$.
Finds students who are enrolled in all courses taught by 'Dr. Smith'.
Projection, Join, Selection:
$\pi_{\text{StudentName, CourseName}}(\sigma_{\text{Age}>21}(\text{Students} \bowtie \text{Enrollments} \bowtie \text{Courses}))$
Retrieves names of students older than 21 and their course names.
Union and Intersection:

$$\Big(\pi_{\text{Name}}(\sigma_{\text{Major}='CS'}(\text{Students})) \cup \pi_{\text{Name}}(\sigma_{\text{Age}>21}(\text{Students}))\Big) - \Big(\pi_{\text{Name}}(\sigma_{\text{Major}='CS'}(\text{Students})) \cap \pi_{\text{Name}}(\sigma_{\text{Age}>21}(\text{Students}))\Big)$$

Finds students either majoring in CS or older than 21, excluding those who meet both criteria.
Nested Operations:
$\pi_{\text{InstructorName}}(\sigma_{\text{CourseName}='Math'}(\text{Courses})) - \pi_{\text{InstructorName}}(\sigma_{\text{CourseName}='Physics'}(\text{Courses}))$
Identifies instructors teaching 'Math' but not 'Physics'.

Logical Model

Entity-Relationship (ER) Models visually represent relational databases, showing entities (real-world items), attributes (entity properties), and relationships (entity connections).

Components

Entity: Represents real-world items. E.g., 'Student', 'Course'.
Attribute: Entity properties. E.g., 'Student' has 'Name', 'Age', 'Major'.
Relationship: Connection between entities. E.g., 'Enrollment' links 'Student' and 'Course'.

Entity Types

Strong Entity: Exists independently. E.g., 'Student'.
Weak Entity: Depends on another entity. E.g., 'Classroom' depends on 'School'.

Attribute Types

Simple Attribute: Indivisible. E.g., 'Name'.
Composite Attribute: Divisible into sub-parts. E.g., 'Address' includes 'Street', 'City', 'Zip'.
Derived Attribute: Calculated from other attributes. E.g., 'Age' from 'Date of Birth'.

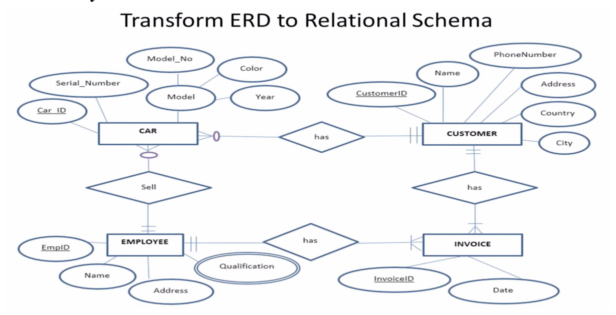
Relationship Types

Unary: Within a single entity type. E.g., 'Employee' supervises 'Employee'.
Binary: Between two entity types. E.g., 'Student' enrolls in 'Course'.
Ternary: Among three entities. E.g., 'Supplier' supplies 'Part' to 'Project'.

Cardinality Constraints

One-to-One: Single relation each way. E.g., 'Person' to 'Passport'.
One-to-Many: One to multiple relations. E.g., 'Instructor' teaches multiple 'Courses'.
Many-to-Many: Multiple relations each way. E.g., 'Students' enrolling in 'Courses'.

Tables and Keys



CAR table with primary key *Car_ID*.
CUSTOMER table with primary key *CustomerID*.
EMPLOYEE table with primary key *EmpID*.
INVOICE table with primary key *InvoiceID*, including foreign key *CustomerID*.
SELL join table to represent the many-to-many relationship between **CAR** and **EMPLOYEE**.

CAR	
Car_ID	Primary Key
Serial_Number	
Model	
Model_No	
Color	
Year	

CUSTOMER	
CustomerID	Primary Key
Name	
PhoneNumber	
Address	
Country	
City	

EMPLOYEE	
EmpID	Primary Key
Name	
Address	
Qualification	

INVOICE	
InvoiceID	Primary Key
Date	
CustomerID	Foreign Key

SELL	
Car_ID	Foreign Key (References CAR)
EmpID	Foreign Key (References EMPLOYEE)
Sell_Details	Additional information about the sale

SQL

SELECT: Retrieves data from a database.
FROM: Specifies the table from where data is retrieved.
WHERE: Filters records based on specified conditions.
JOIN: Combines rows from two or more tables based on a related column.
LEFT JOIN: Returns all records from the left table, and matched records from the right table.
RIGHT JOIN: Returns all records from the right table, and the matched records from the left table.
NATURAL JOIN: Performs a join between two tables using all columns with the same names.
INNER JOIN: Returns rows when there is at least one match in both tables.
GROUP BY: Arranges identical data into groups.
ORDER BY: Sorts the result-set in ascending or descending order.
HAVING: Filters records on aggregated results.
ROUND(): Rounds a number to a specified number of decimal places - similar to COUNT().
AVG(): Calculates the average value of a numeric column.
EXPLAIN: Provides information about how MySQL executes a query.
CREATE INDEX: Creates an index on a table to improve query performance.
LIKE: Operator used in a WHERE clause to search for a specified pattern in a column.
AND: Combines two or more conditions in a WHERE clause.
OR: Allows for multiple conditions in a WHERE clause, where if any condition is true, the row is included.
ASC: Specifies ascending order in an ORDER BY clause.
DESC: Specifies descending order in an ORDER BY clause.
IS NOT NULL: Operator used in a WHERE clause to filter out rows where specified column's value is not NULL.
%: Wildcard character used with LIKE operator to match any sequence of characters.
PRIMARY KEY: Constraint used to uniquely identify each row in a table.
FOREIGN KEY: Constraint used to link two tables together.

```sql
SELECT
    b.title AS BookTitle,
    CONCAT(a.first_name, ' ', a.last_name) AS AuthorName,
    ROUND(AVG(s.sale_amount)) AS AverageSales,
    COUNT(s.sale_id) AS TotalSales
FROM
    books b
INNER JOIN authors a ON b.author_id = a.author_id
LEFT JOIN sales s ON b.book_id = s.book_id
```

```sql
WHERE
    a.last_name LIKE 'R\%' AND
    b.publication_year > 2000 AND
    s.sale_id IS NOT NULL
GROUP BY
    b.book_id
HAVING
    COUNT(s.sale_id) > 50
ORDER BY
    TotalSales DESC,
    b.title ASC;
```

SELECT: Starts the selection of data.
CONCAT(): Concatenates the author's first and last name.
ROUND(): Rounds the average sales amount to the nearest integer.
FROM: Indicates the books table as the source.
INNER JOIN: Joins the books with the authors where the author_id matches.
LEFT JOIN: Joins the books with the sales to include all books, even if there are no sales records.
WHERE: Filters the result to authors with a last name starting with 'R' and books published after 2000.
LIKE 'R%': Searches for authors with last names beginning with 'R'.
AND: Combines multiple conditions in the WHERE clause.
GROUP BY: Groups the result by book ID to calculate averages and counts per book.
HAVING: Filters groups to include only those with more than 50 sales.
ORDER BY: Orders the results first by TotalSales in descending order, then by BookTitle in ascending order.
ASC/DESC: Specifies the order direction.
IS NOT NULL: Ensures that only sales with valid sale IDs are included.

Data Types

INT: A normal-size integer that can be signed or unsigned.
VARCHAR(n): A variable-length string with a maximum length of 'n' characters.
TEXT: A text column with a maximum length of 65,535 characters.
DATE: A date, formatted as YYYY-MM-DD.
TIMESTAMP: A timestamp, combining a date and a time.
FLOAT(p, d): A floating-point number with a precision 'p' and a scale 'd'.
DOUBLE(p, d): A double precision floating-point number.
DECIMAL(p, d): An exact fixed-point number.
BOOLEAN: A true or false value.
BLOB: A binary large object that can hold a variable amount of data.
CHAR(n): A fixed-length non-binary string.
BIGINT: A large integer that can be signed or unsigned.
ENUM(val1, val2, ...): A string object that can only have one value, chosen from a list of possible values.
SET(val1, val2, ...): A string object that can have zero or more values, chosen from a list of possible values.

Disk Storage, File Struct, Indexing, Transactions

Disk Storage

Physically stored on magnetic or solid-state disks.

Data Blocks: The smallest unit of data storage on a disk. Databases often read or write one block at a time.
Sectors and Tracks: Physical divisions of a disk; sectors are subdivisions of tracks.
File Organization: How records are physically arranged on a disk - sequential, direct, or indexed.
Partitions: Division of a database table into smaller parts for performance and manageability.
Physical Design: Involves the design of physical storage, file organization, and indexing for performance optimization.

File Structure

Sequential: Records are stored in sequential order, typically sorted by a key field.
Direct or Hashed: Records are stored in seemingly random order but according to a calculated position (hash).
Indexed: Uses indexes to quickly locate data without scanning the entire file.

Indexing

Used to speed up the retrieval of data.

Primary/Secondary Index: Primary index uses the unique table identifier, while secondary index uses non-primary fields for data access.
Index: A data structure that allows for faster data retrieval from a database.
Index File: A separate file containing indexes to improve data search efficiency in database tables.
Sparse Indices: Index type with entries for some records, requiring less space but potentially slower navigation.
Dense Indices: Index type with an entry for every record, providing faster lookup at the expense of more space.

Transactions

Deadlock: A situation in concurrent transactions where transactions wait indefinitely for each other's resources.
Atomicity: A property ensuring all operations in a transaction are completed successfully or not at all.
Consistency: Ensures a transaction transitions the database from one valid state to another, maintaining data integrity.
Isolation: The property that defines the visibility of transaction changes to other transactions.
Durability: Ensures that once a transaction is committed, it remains so even after a system failure.
ACID: Atomicity, Consistency, Isolation, Durability
Isolation Levels: Defines the degree of isolation between transactions, impacting concurrency and consistency.

B+ Tree - Overview

A type of self-balancing tree data structure that maintains sorted data and allows for searches, sequential access, insertions, and deletions in logarithmic time. The B+ tree represents an index structure that is highly optimized for systems that read and write large blocks of data.

Root Level

- The root node contains keys that act as separation values which divide its subtrees.
- For instance, the key '39' separates the range of keys that are less than '39' and those that are greater than or equal to '39' but less than '788'.

Intermediate Level

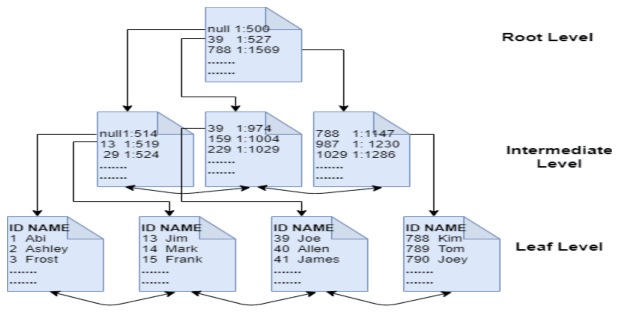
- These nodes (also known as internal nodes) contain keys that similarly divide the range of keys.
- Each key in the intermediate nodes corresponds to a pointer to a node in the level below.

Leaf Level

- The leaf nodes contain the actual records or pointers to the records. In your diagram, the leaf nodes contain a set of keys (ID) and their corresponding values (NAME).
- Leaf nodes are linked to each other in a linked-list fashion, which makes range queries efficient as you can traverse these nodes sequentially.

Characteristics of the B+ Tree in the Diagram

1) All keys are stored in the leaf nodes
2) The leaf nodes of the tree form a linked list - easy to move between leafs. This is useful for full table scans, range searches, or sequential access.
3) The intermediate nodes guide the search. They contain keys and pointers that direct the search to the correct leaf node.
4) The tree is balanced - all leaf nodes are at the same level, ensures that all search operations require the same number of disk accesses.
5) The internal nodes only contain keys and not the full records, they can hold more keys and thus reduce the height of the tree.



B+ Tree - Insert + Select

Insert

1) Start at the root. Traverse the tree to find the correct leaf node where the new key should be inserted.
2) Insert the new key in the leaf node in sorted order.
3) If the leaf node overflows (exceeds its maximum capacity), split it into two nodes and promote the middle key to the parent node.
4) If the parent node overflows, repeat the split process up the tree.

Select

1) Begin at the root and traverse the tree, using the key values in each node to direct the search to the correct leaf node.
2) Once the correct leaf is found, the key can be retrieved.

Block Access Policies

Block access policies are strategies used primarily in disk scheduling. They determine the order in which disk I/O requests are processed, which can significantly impact the overall performance of the system.
FIFO (First-In-First-Out): The oldest request in the queue is processed first. Simple to implement. Used in scenarios where fairness is important, and the load is evenly distributed. Can lead to suboptimal performance if older requests are not as critical as newer ones.
SJF (Shortest Job First): Requests are processed based on the length of the job, with shorter jobs being given priority. Minimizes the average waiting time and is efficient for systems with varying job sizes. Can lead to starvation of longer jobs and is difficult to implement, as predicting job length is challenging.
LOOK: The disk arm moves in one direction, fulfilling all requests until there are no more in that direction, then reverses its direction. Reduces the movement of the disk arm, lowering the seek time compared to FIFO. If requests are heavily skewed to one end, it can lead to longer waiting times for some requests.

Block Replacement Policies

Block replacement policies are techniques used in cache management. When the cache is full, and a new block needs to be loaded, these policies decide which existing block to replace.
FIFO (First-In-First-Out): The oldest block in the cache is replaced first. Simple to understand and implement. May not always remove the least useful block, leading to suboptimal cache performance.
LRU (Least Recently Used): Replaces the block that has been unused for the longest time. More effective than FIFO in many scenarios as it considers the recent usage pattern. Implementation can be more complex and resource-intensive than FIFO, as it requires tracking the usage history of each block.