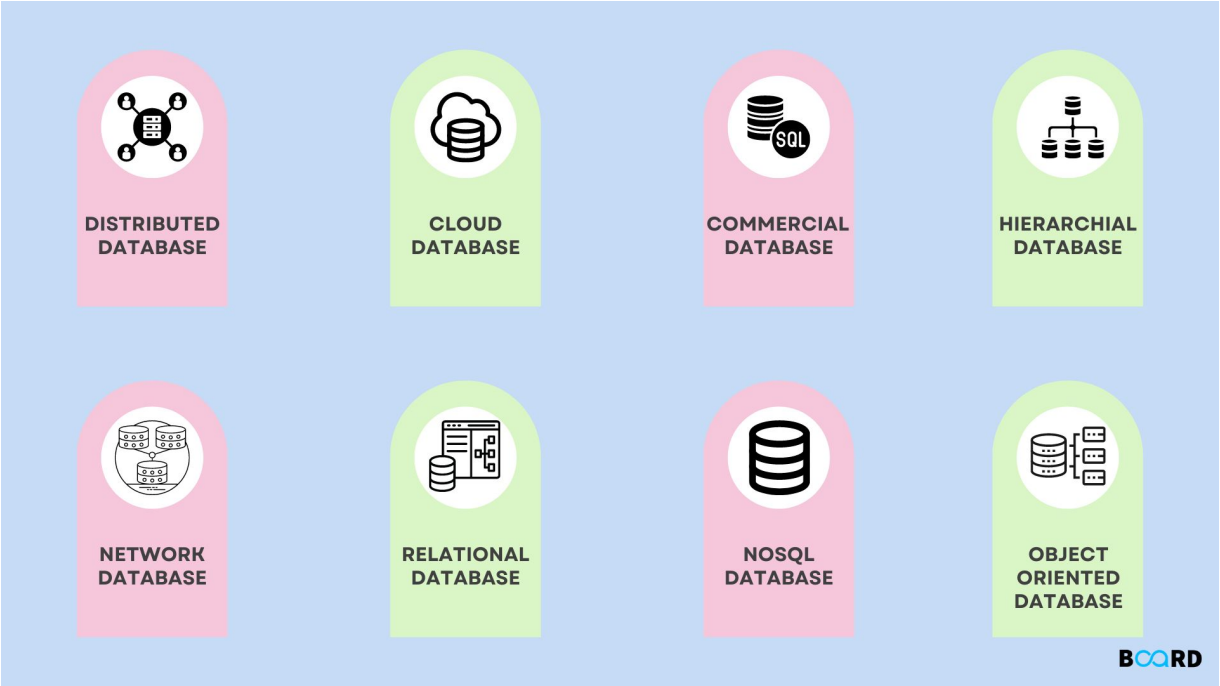
Databases

Relational Databases and RDBMS

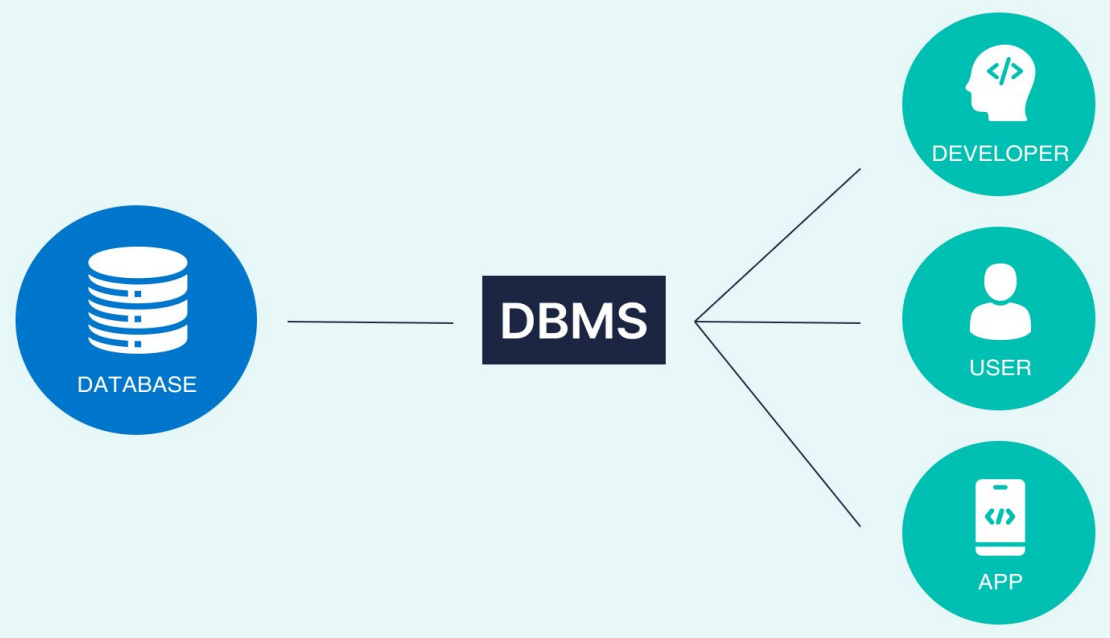
Database

- organized collection of data
- managed by DBMS
- stored on RAM, a file system, a computer cluster or a cloud storage



DBMS

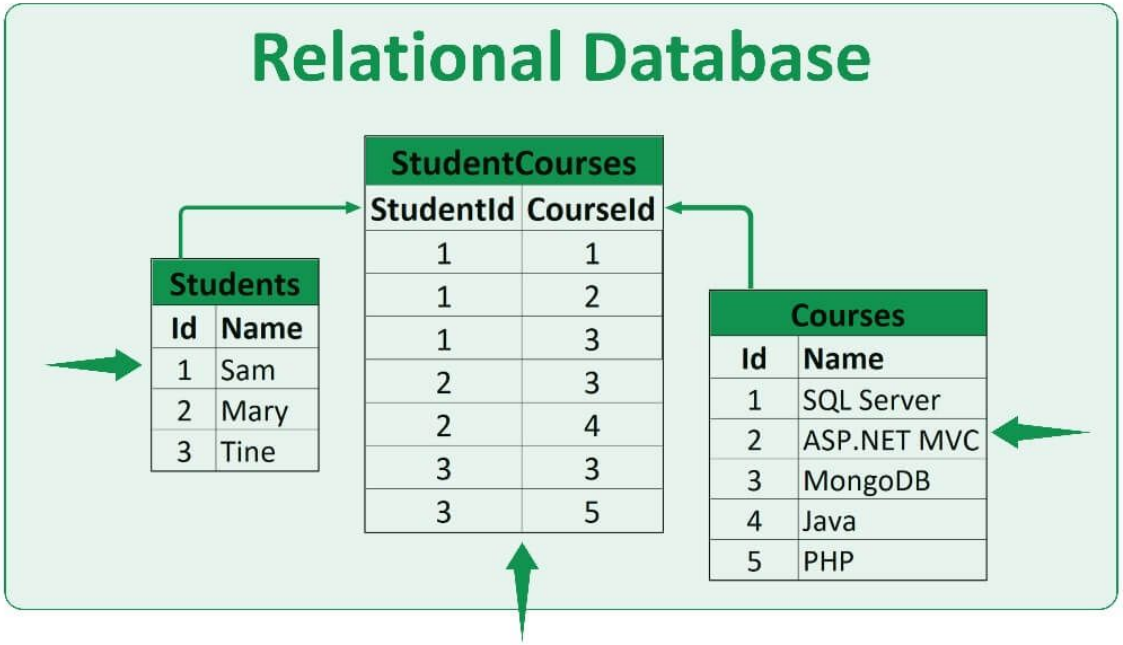
- Data storage, retrieval and update
- User accessible catalog or data dictionary describing the metadata
- Support for transactions and concurrency
- Facilities for recovering the database should it become damaged
- Support for authorization of access and update of data
- Access support from remote locations
- Enforcing constraints to ensure data in the database abides by certain rules



Relational Database (RDBMS)

Definition:

- Consists of rows and columns
- Implements relations between data
- Tables has primary and foreign keys



Relational Database (RDBMS)

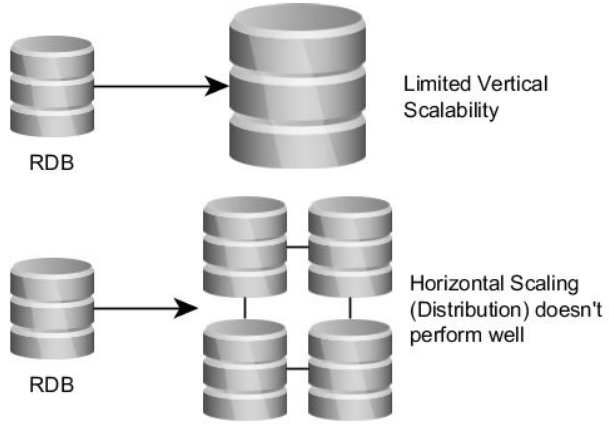
Advantages:

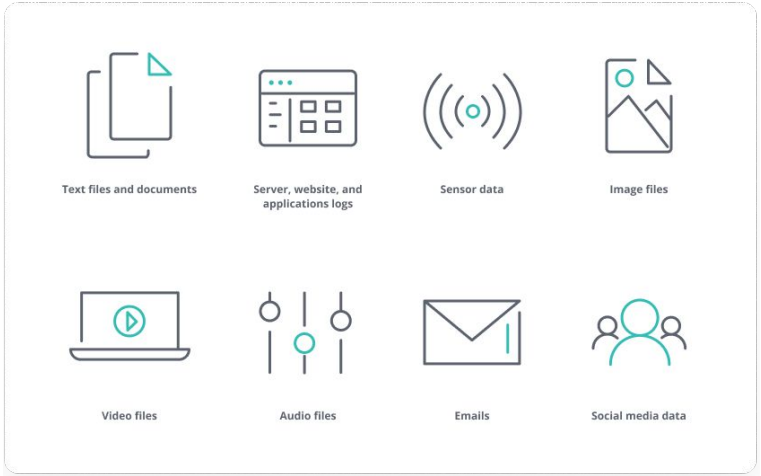
- Data Integrity
- Flexibility
- Data Security
- Scalability
- Data Querying

Relational Database (RDBMS)

Disadvantages:

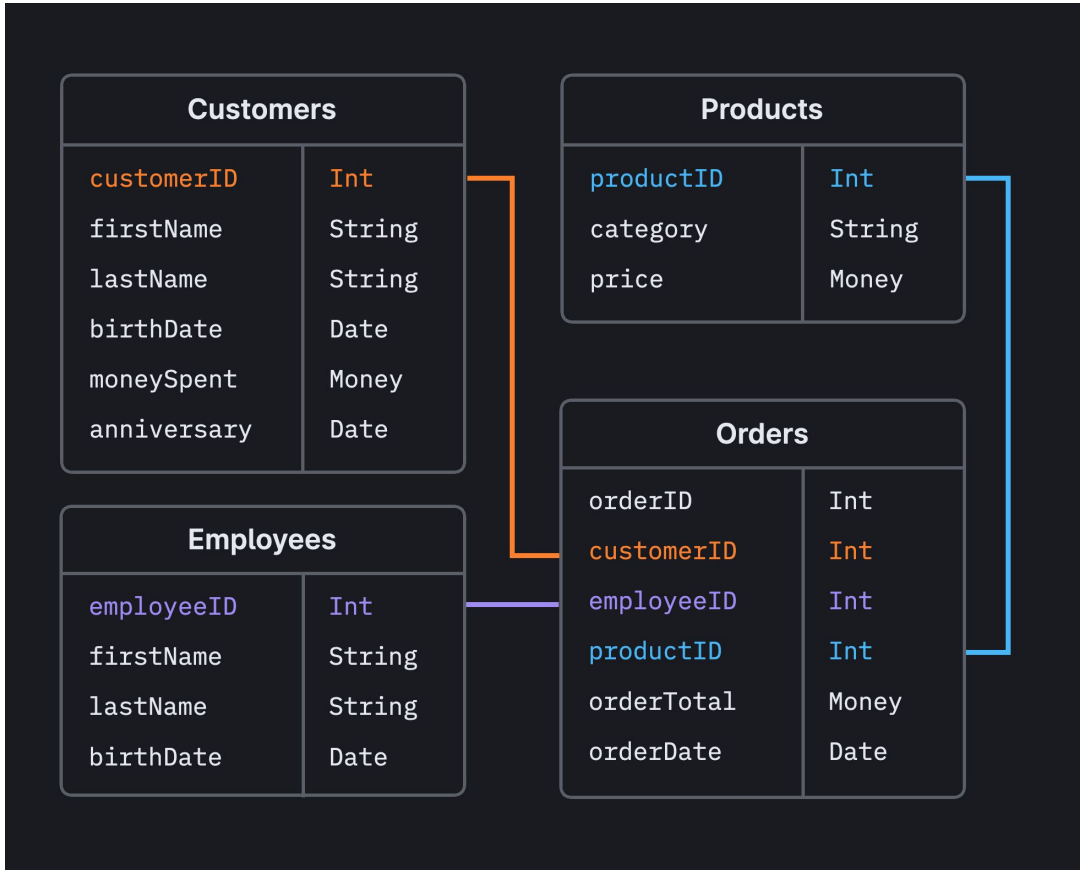
- Performance
- Schema Mutability
- Horizontal Scalability
- Not Suitable for Unstructured Data





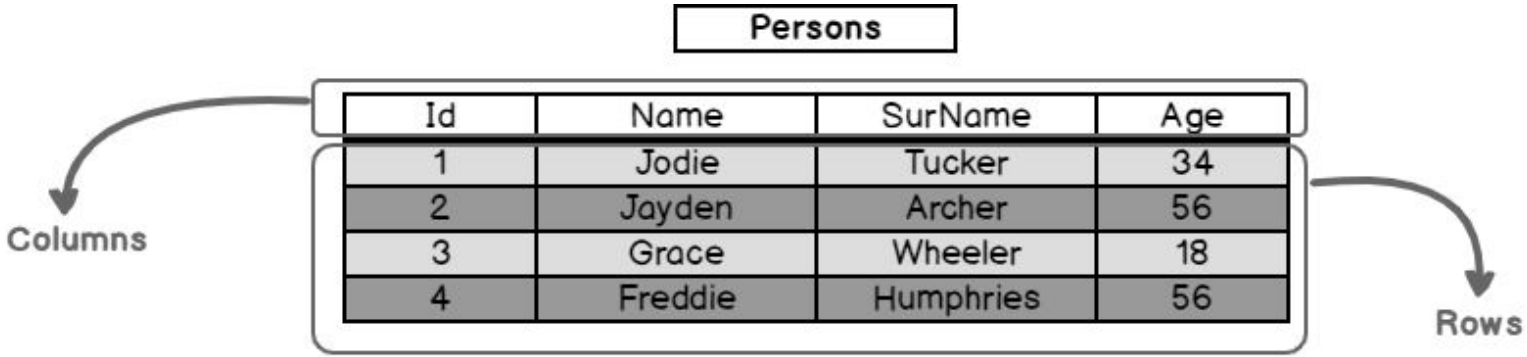
RDBMS Structure

- Schema
- Tables
- Rows and Columns
- Primary and Foreign Keys
- Relationships:
  - one-to-one
  - one-to-many
  - many-to-many
- Indices
- Views

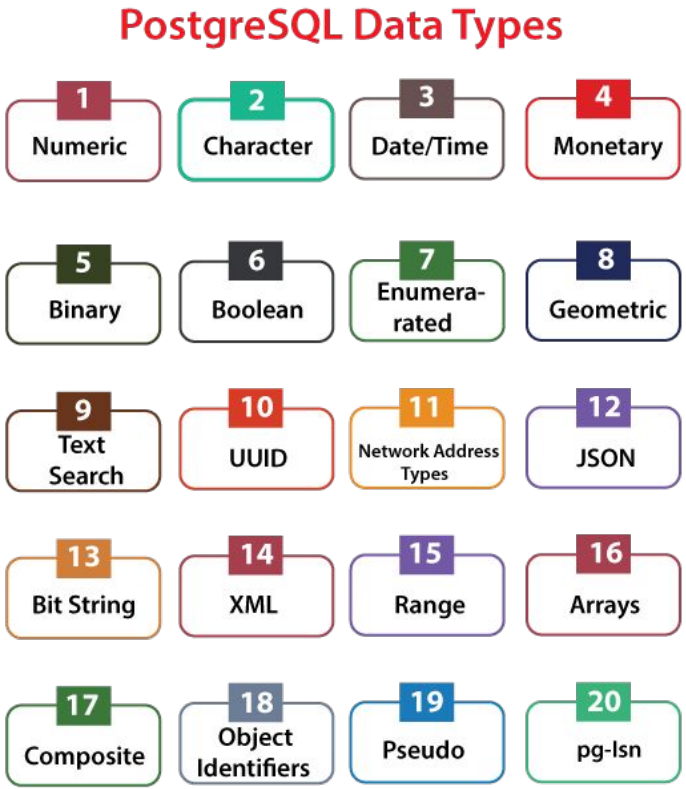


Relational Table

- Can be created, dropped, renamed
- Consists of rows, columns, keys and indices



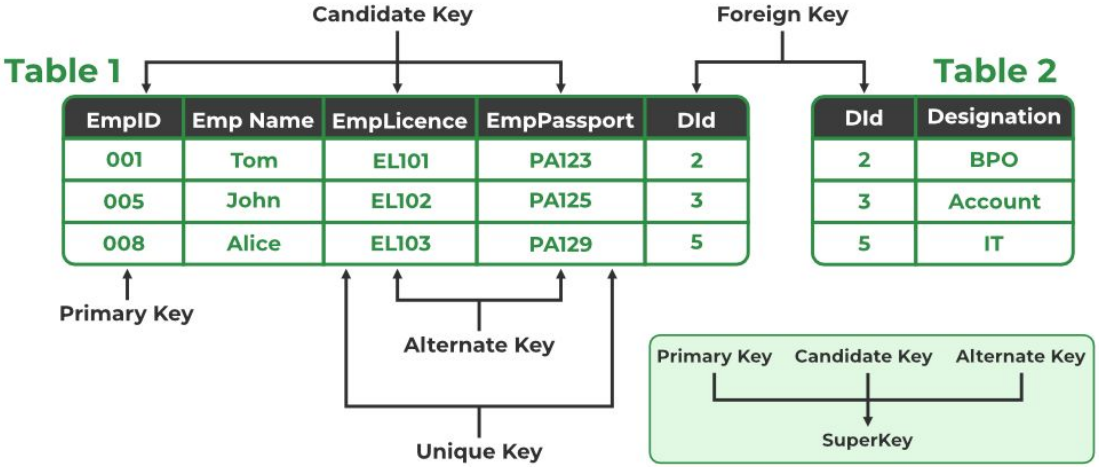
Data Types



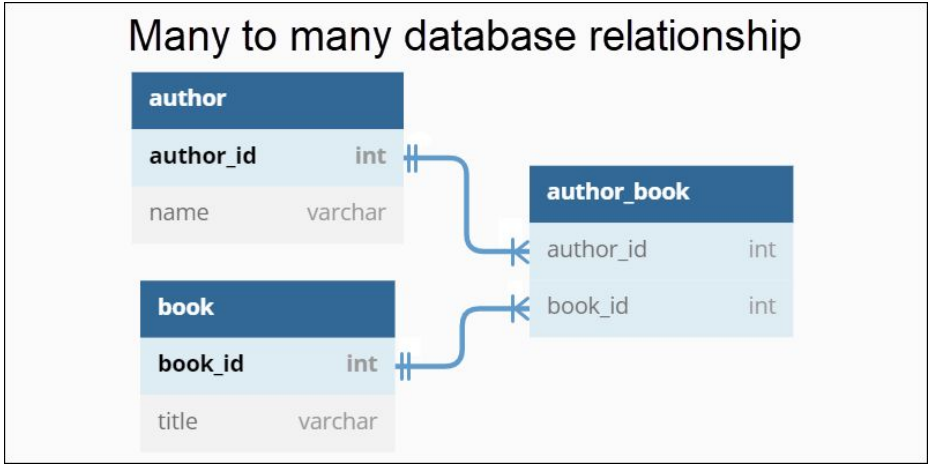
https://www.postgresql.org/docs/current/datatype.html

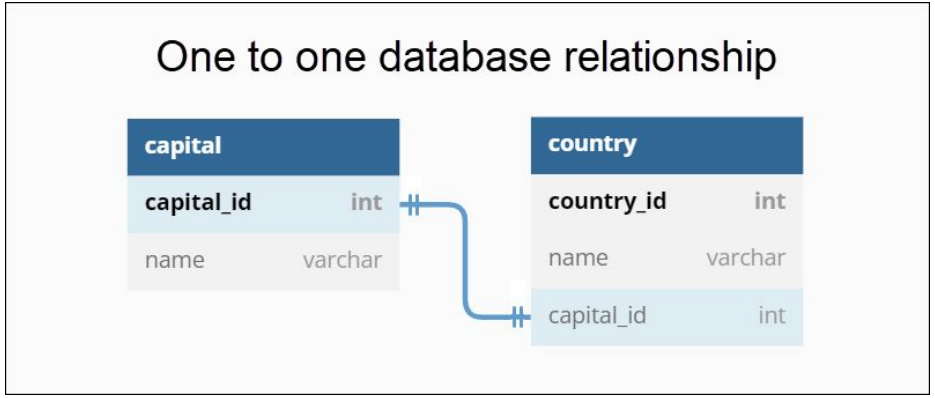
Relational Keys

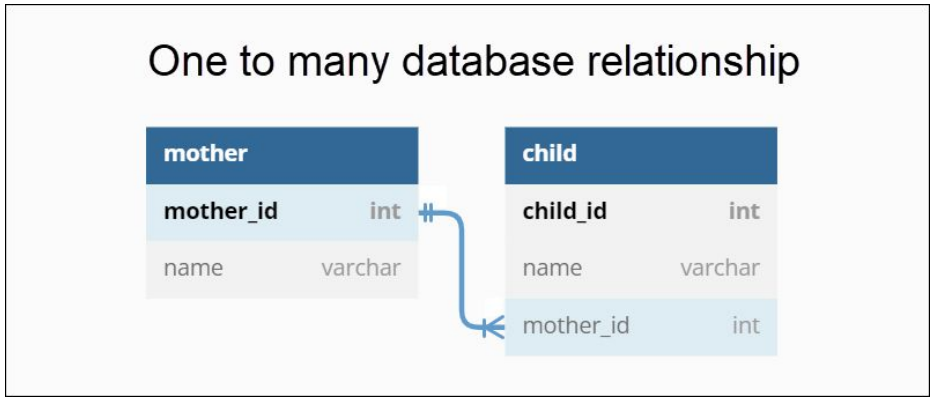
- Primary Key
- Foreign Key
- Unique Key
- Composite Key
- Candidate/Alternate Key



Relationships

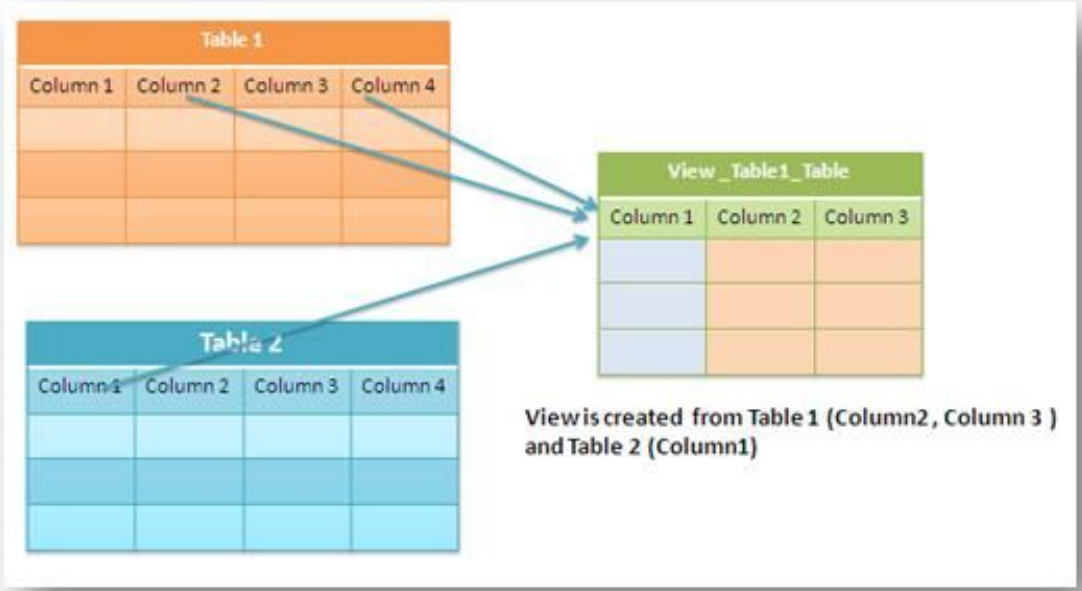






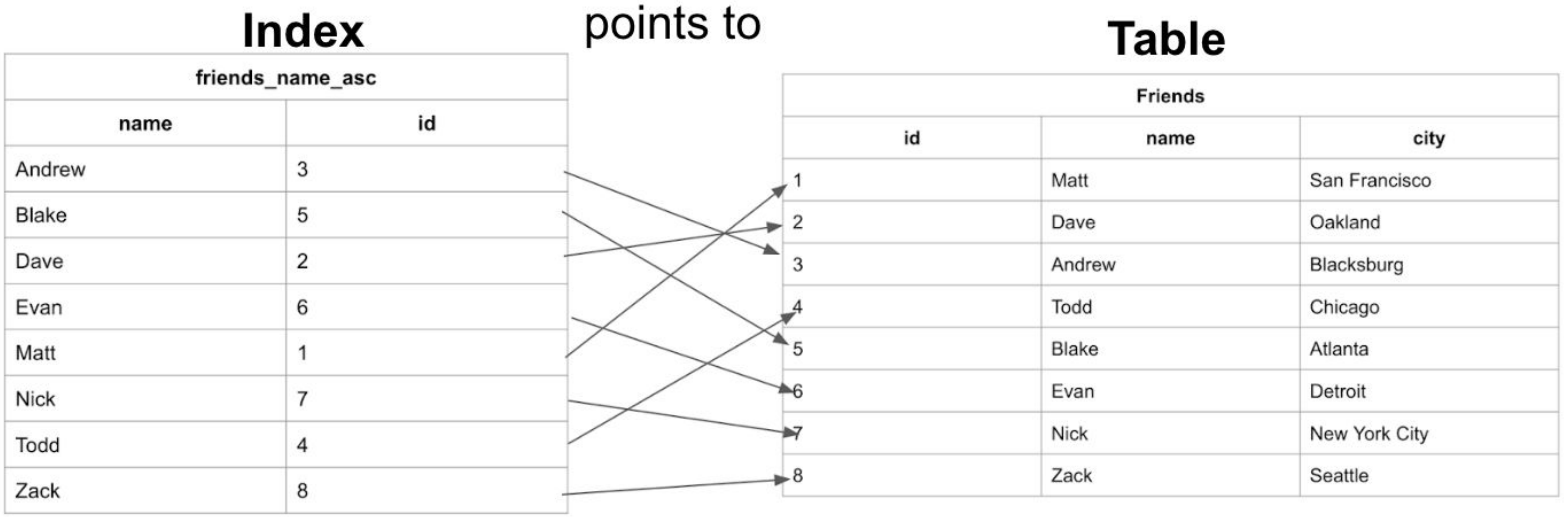
View

- Virtual Table
- Dynamic Representation
- Can be reused



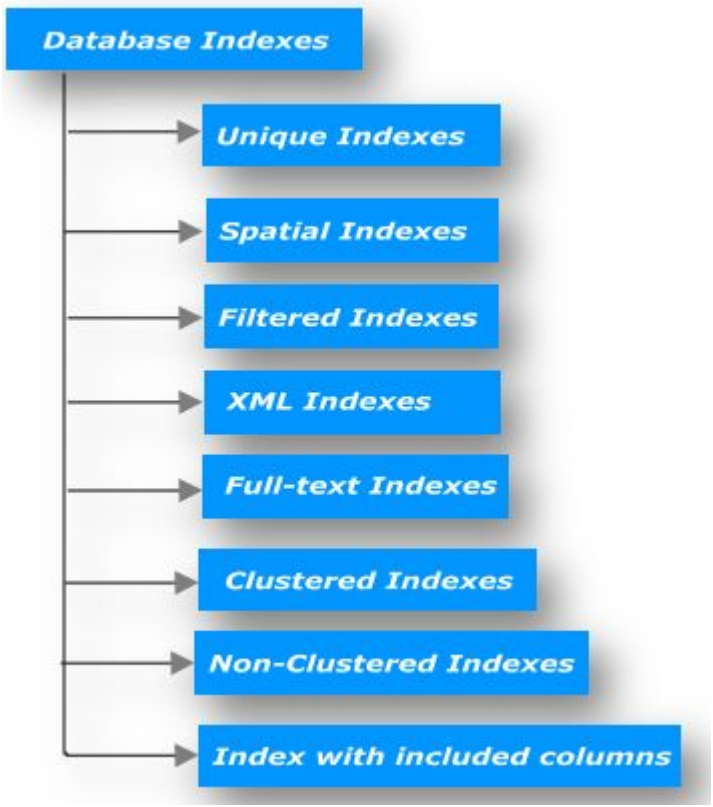
Indices

- Sorted structures
- Improve search, sorting and access



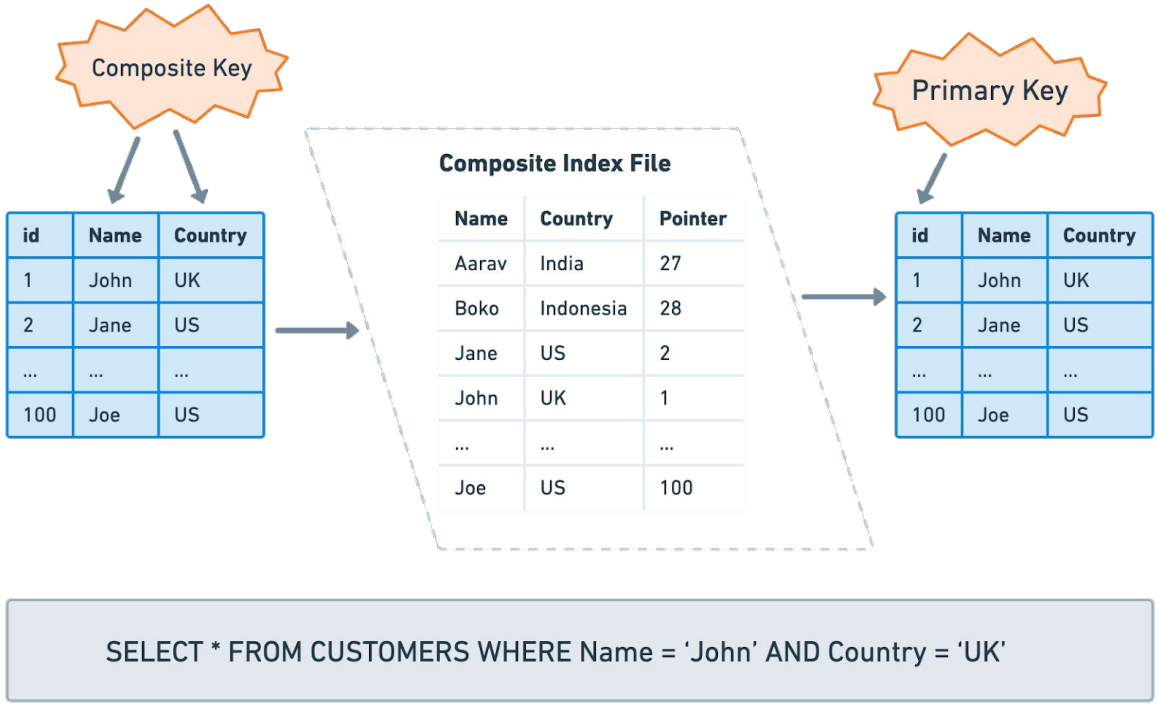
Index Types

- Primary Index
- Secondary Index
- Unique Index
- Composite Index
- Full-Text Index



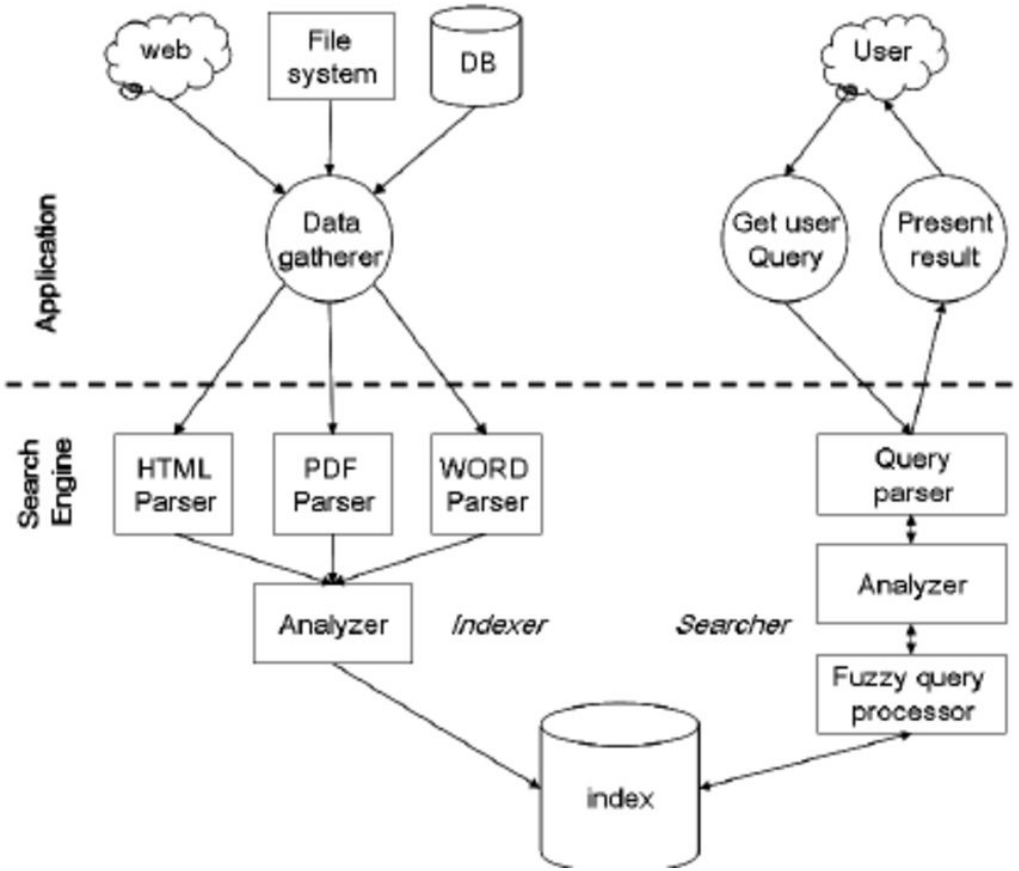
Composite Index

- Multi-column index
- Column order is significant
- Can be unique
- Query specific



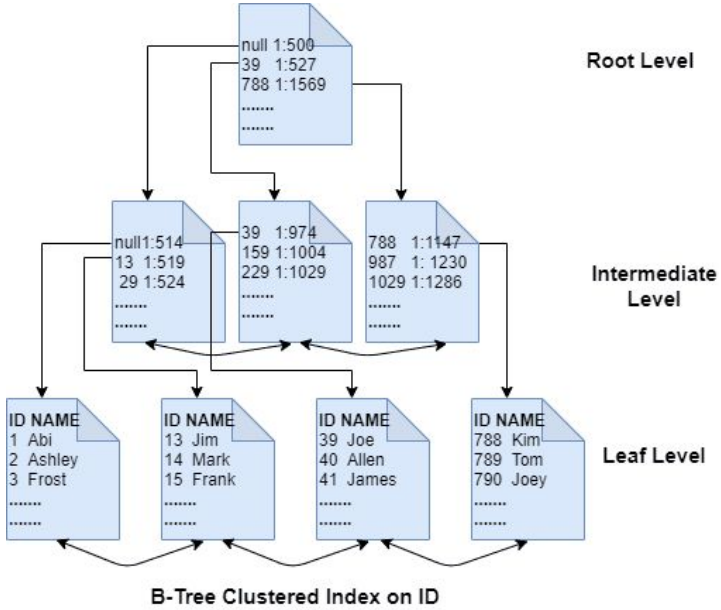
Full-Text Index

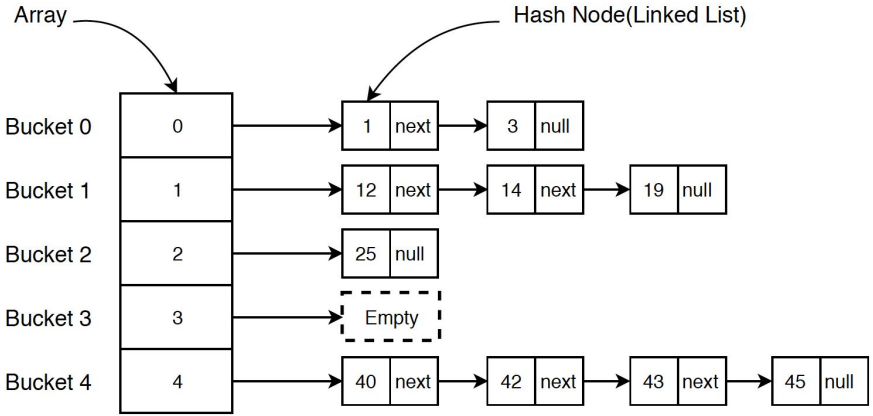
- Text-based data search
- Based on text splitting
- Search by keywords/phrases



Storage of Indices

- Stored separately from data
- Can be cached in RAM
- Common types:
  - B-tree
  - Hash
  - Bitmap





Bitmap Index

- Each value has its bit
- Efficient with low-cardinality data

Roll_no	Name	Gender	Result
01	Geeta Raj	F	Fail
02	Deep Singh	M	Fail
03	Ria Sharma	F	Pass
04	Ajit Singh	M	Fail
05	Jitu Bagga	M	Pass
06	Neha Kapoor	F	Pass

M	F
0	1
1	0
0	1
1	0
1	0
0	1

Bitmap Indices for Gender

Pass	Fail
0	1
0	1
1	0
0	1
1	0
1	0

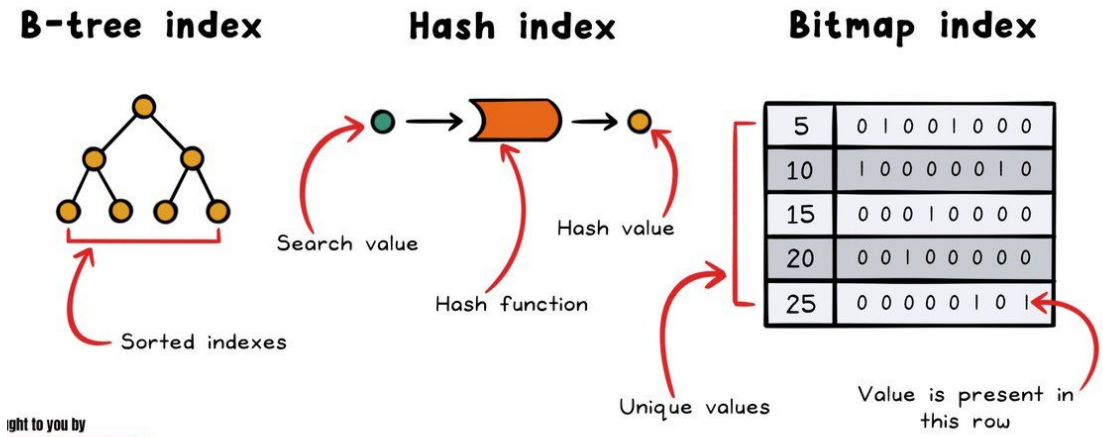
Bitmap Indices for Result

Row	Name	Age	0–10	10–20	20–30	30–40
1	Ali	28	0	0	1	0
2	Ayşe	33	0	0	0	1
3	Fatma	37	0	0	0	1
4	Alican	12	0	1	0	0
5	Fatmagül	6	1	0	0	0
6	Ayşegül	8	1	0	0	0

		Bitmaps			
ROW ID	Hair Colour	Brown	Black	Blonde	Red
1	Brown	1	0	0	0
2	Red	0	0	0	1
3	Brown	1	0	0	0
4	Black	0	1	0	0
5	Blonde	0	0	1	0
6	Brown	1	0	0	0
7	Blonde	0	0	1	0

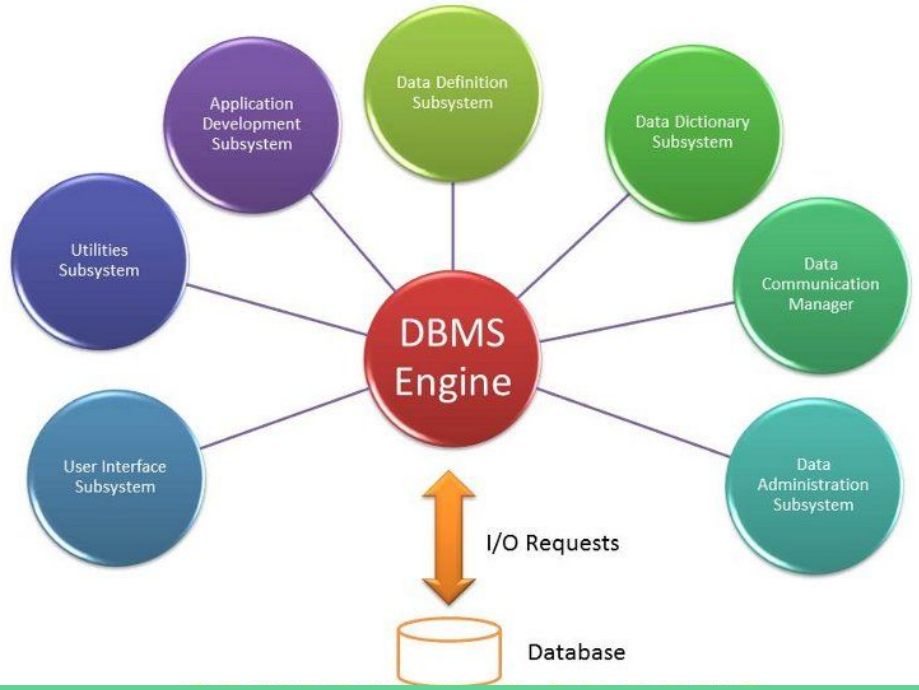
Hash vs B-tree vs Bitmap

- Hash Index:
  - Exact match
  - More I/O due to collisions
- B-tree:
  - Range query
  - Order
  - Minimal I/O due to balanced structure
- Bitmap:
  - Low-cardinality data
  - Boolean operations
  - Efficient I/O on low-cardinality



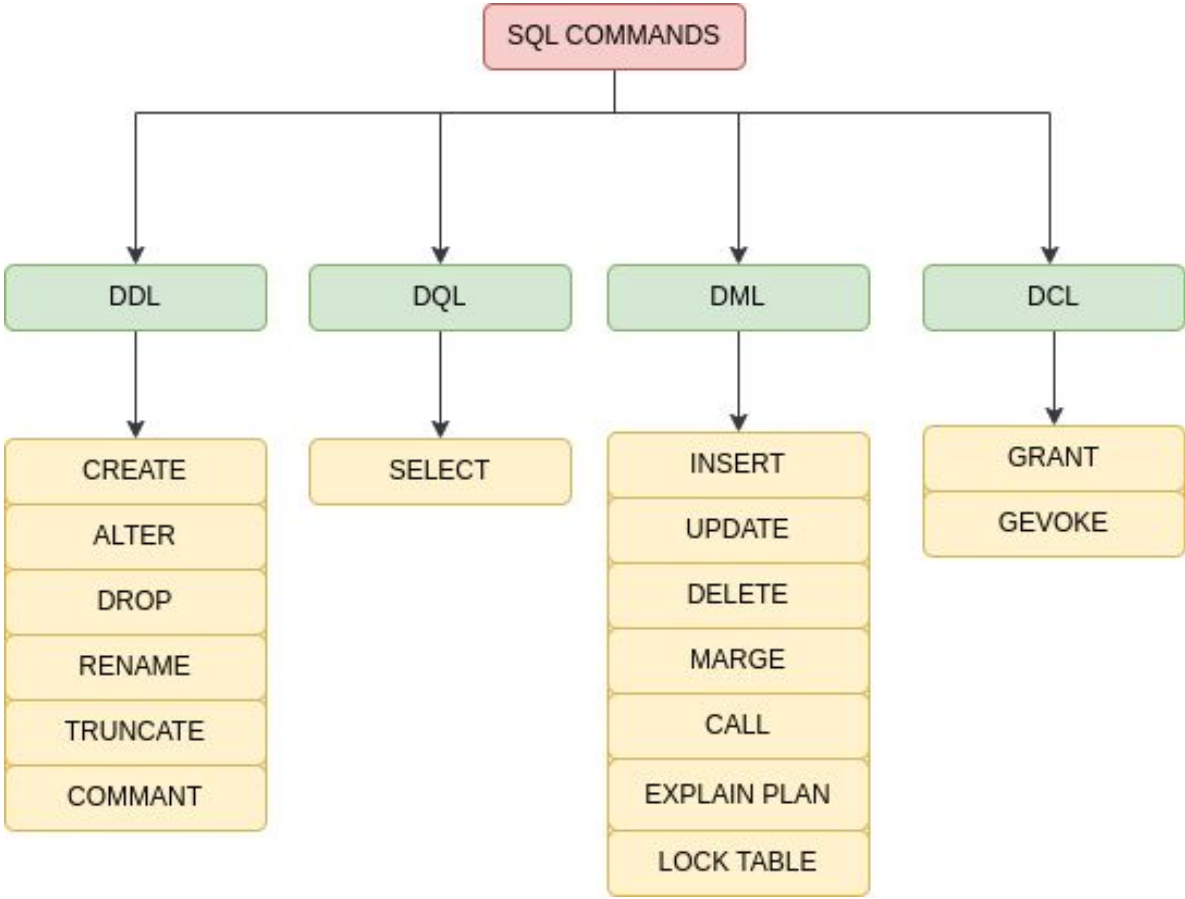
DSL (Domain Specific Language)

- SQL (Structured Query Language)
  - PostgreSQL SQL
  - MySQL SQL
- NoSQL Query Languages
  - MongoDB query language
  - Cypher for Neo4j
- ORM (Object-Relational Mapping) DSLs
  - SQLAlchemy for Python
  - ActiveRecord for Ruby
  - Sequel for Ruby
- Database Definition DSLs
  - PostgreSQL DDL
  - MySQL DDL
- Data Manipulation DSLs
  - MapReduce DSL
  - Hadoop DSL



SQL

- Data Query Language (DQL)
- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)



DDL

- Common operations:
  - CREATE
  - DROP
  - ALTER
- Used to create constraints
- Used in database schema management

```
CREATE TABLE clients (
  id SERIAL PRIMARY KEY,
  first_name VARCHAR(255),
  last_name VARCHAR(255),
  middle_name VARCHAR(255) DEFAULT '',
  full_name VARCHAR(255),
  code VARCHAR(255) NOT NULL,
  deleted_at TIMESTAMP,
  created_at TIMESTAMP NOT NULL,
  updated_at TIMESTAMP NOT NULL,
  tax_years INTEGER[] DEFAULT '{}',
  suffix VARCHAR(255),
  account_name VARCHAR(255),
  firm_id BIGINT,
  state VARCHAR(255),
  quickbooks_state INTEGER DEFAULT 0 NOT NULL,
  current_sign_in_at TIMESTAMP,
  custom_fields JSONB DEFAULT '{}' :: JSONB NOT NULL
);
CREATE UNIQUE INDEX index_clients_on_code_and_firm_id
  ON clients (code, firm_id);
CREATE INDEX index_clients_on_code ON clients (code);
CREATE INDEX index_clients_on_deleted_at ON clients (deleted_at);
CREATE INDEX index_clients_on_firm_id ON clients (firm_id);
ALTER TABLE
  clients
ADD
  CONSTRAINT accounts_account_type_null
    CHECK (account_type IS NOT NULL);
```

Queries

```sql
SELECT
  "clients".*
  "users".*
FROM
  "clients"
  INNER JOIN "client_users" ON
    "client_users"."client_id" = "clients"."id"
  INNER JOIN "users" ON
    "users"."id" = "client_users"."user_id"
WHERE
  "clients"."code" BETWEEN 'A'
  AND 'D'
  AND (
    users.last_sign_in_at > '2024-03-01'
  )
```

```sql
SELECT
  "clients"."id",
  "clients"."code",
  max(users.last_sign_in_at) as client_last_sign_in,
  count(users.id) as login_count
FROM
  "clients"
  INNER JOIN "client_users" ON
    "client_users"."client_id" = "clients"."id"
  INNER JOIN "users" ON
    "users"."id" = "client_users"."user_id"
WHERE
  "clients"."code" BETWEEN 'A'
  AND 'D'
  AND (
    users.last_sign_in_at > '2024-03-01'
  )
GROUP BY
  "clients"."id"
HAVING
  (
    count(users.id) > 10
  )
ORDER BY
  clients.code
```

Function

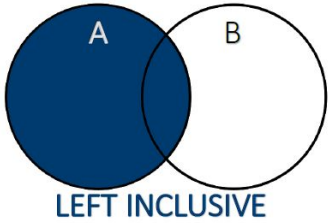
```sql
CREATE OR REPLACE FUNCTION multiply_three_numbers(a INT, b INT, c INT)
RETURNS INT AS $$
    SELECT a * b * c;
$$ LANGUAGE SQL;
```

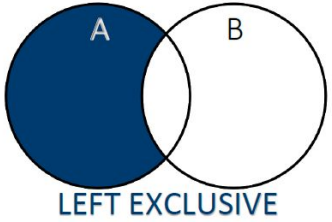
- Reusable block of code
- Accepts input parameters and returns results
- Types:
  - Scalar Functions
  - Set-Returning Functions
  - Aggregate Functions
- Languages:
  - SQL
  - PL/pgSQL

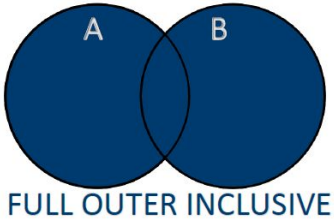
```sql
CREATE OR REPLACE FUNCTION triangle_area(a FLOAT, b FLOAT, c FLOAT)
RETURNS FLOAT AS
$$
DECLARE
    p FLOAT;
    area FLOAT;
BEGIN
    IF a + b > c AND a + c > b AND b + c > a THEN
      p := (a + b + c) / 2;
      area := sqrt(p * (p - a) * (p - b) * (p - c));
      RETURN area;
    ELSE
      RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Join

- Combines columns from one or more tables
- Based on related columns
- Types:
  - Inner Join
  - Right Join
  - Left Join
  - Full Join
  - Cross Join
  - Self Join

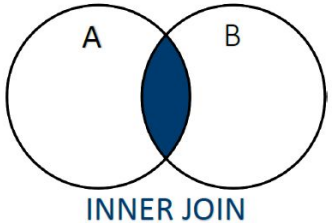




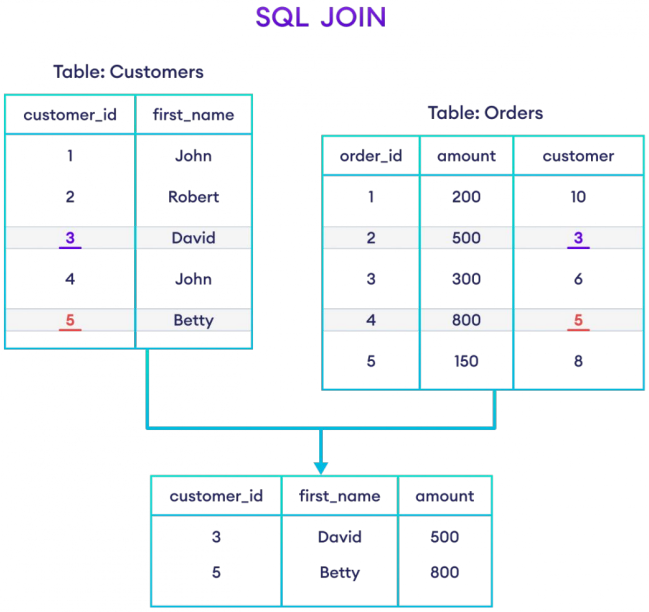


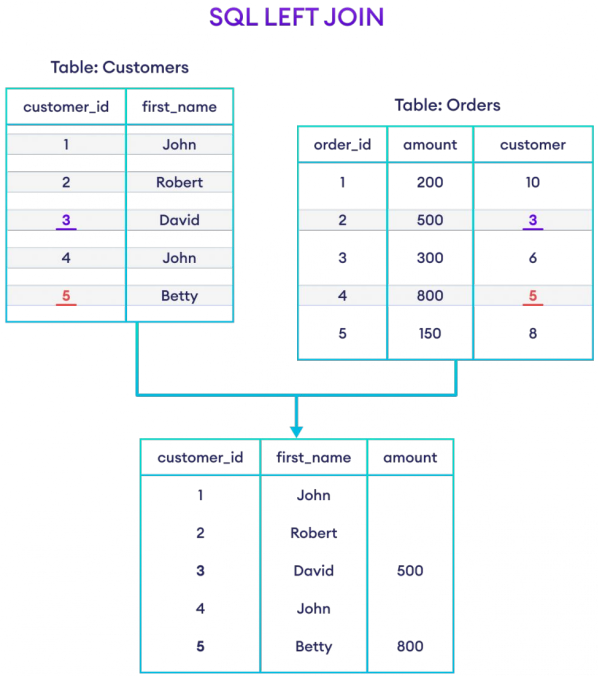
SQL JOINS

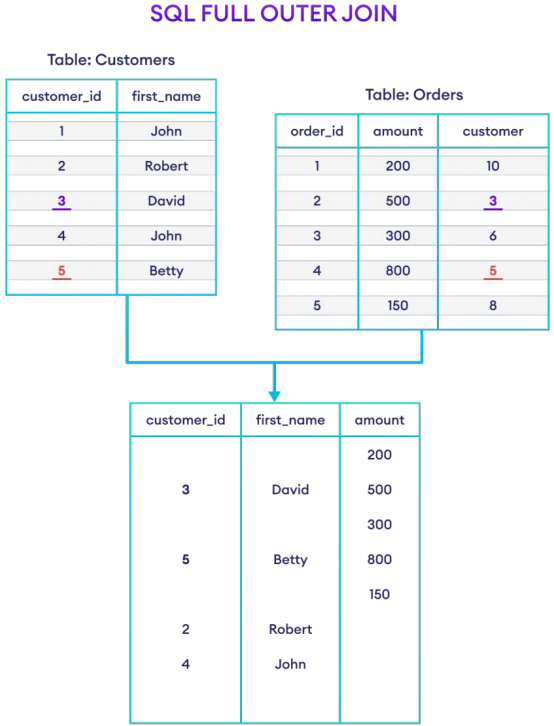
LEFT INCLUSIVE	RIGHT INCLUSIVE
SELECT *[Select List]* FROM TableA A LEFT OUTER JOIN TableB B ON A.Key= B.Key	SELECT *[Select List]* FROM TableA A RIGHT OUTER JOIN TableB B ON A.Key= B.Key
LEFT EXCLUSIVE	**RIGHT EXCLUSIVE**
SELECT *[Select List]* FROM TableA A LEFT OUTER JOIN TableB B ON A.Key= B.Key WHERE B.Key IS NULL	SELECT *[Select List]* FROM TableA A LEFT OUTER JOIN TableB B ON A.Key= B.Key WHERE A.Key IS NULL
FULL OUTER INCLUSIVE	**FULL OUTER EXCLUSIVE**
SELECT *[Select List]* FROM TableA A FULL OUTER JOIN TableB B ON A.Key = B.Key	SELECT *[Select List]* FROM TableA A FULL OUTER JOIN TableB B ON A.Key = B.Key WHERE A.Key IS NULL OR B.Key IS NULL
INNER JOIN	
SELECT *[Select List]* FROM TableA A INNER JOIN TableB B ON A.Key = B.Key	



Join

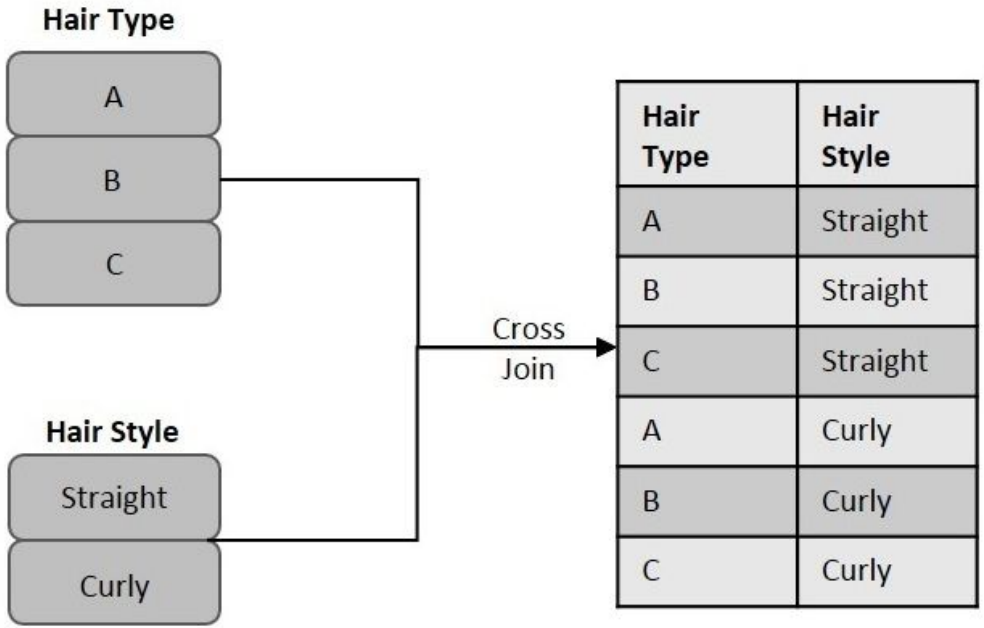

SQL JOIN
Table: Customers
customer_id
first_name
1
John
2
Robert
3
David
4
John
5
Betty
Table: Orders
order_id
amount
customer
1
200
10
2
500
3
3
300
6
4
800
5
5
150
8
customer_id
first_name
amount
3
David
500
5
Betty
800


SQL LEFT JOIN
Table: Customers
customer_id
first_name
1
John
2
Robert
3
David
4
John
5
Betty
Table: Orders
order_id
amount
customer
1
200
10
2
500
3
3
300
6
4
800
5
5
150
8
customer_id
first_name
amount
1
John
2
Robert
3
David
500
4
John
5
Betty
800


SQL FULL OUTER JOIN
Table: Customers
customer_id
first_name
1
John
2
Robert
3
David
4
John
5
Betty
Table: Orders
order_id
amount
customer
1
200
10
2
500
3
3
300
6
4
800
5
5
150
8
customer_id
first_name
amount
200
3
David
500
300
5
Betty
800
150
2
Robert
4
John

Cross Join

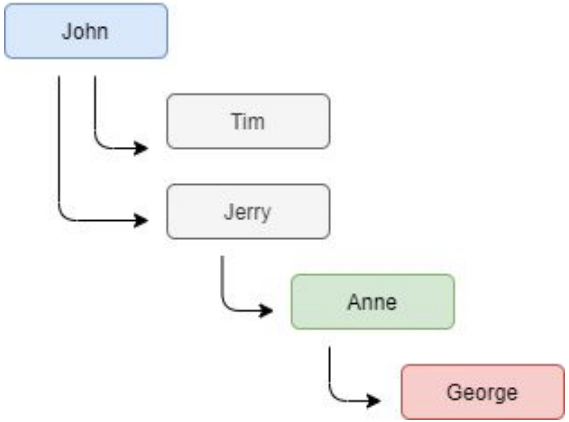
- Cartesian product
- Generates every possible combination of rows from the joined tables
- Can be replaced with an inner join with an always-true condition



Self Join

- Table is joined with itself
- Based on a related column
- Mostly used to represent hierarchies

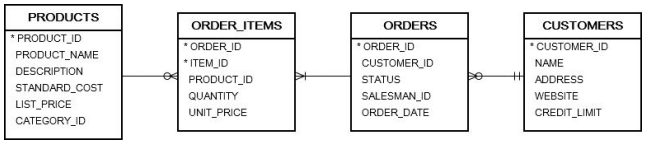
Employee ID	Employee Name	Manager ID
1	John	NULL
2	Tim	1
3	Jerry	1
4	Anne	3
5	George	4




```
SELECT emp.employee_id, emp.last_name, emp.manager_id, mng.last_name
FROM   employees emp, employees mng
WHERE  emp.manager_id = mng.employee_id
```

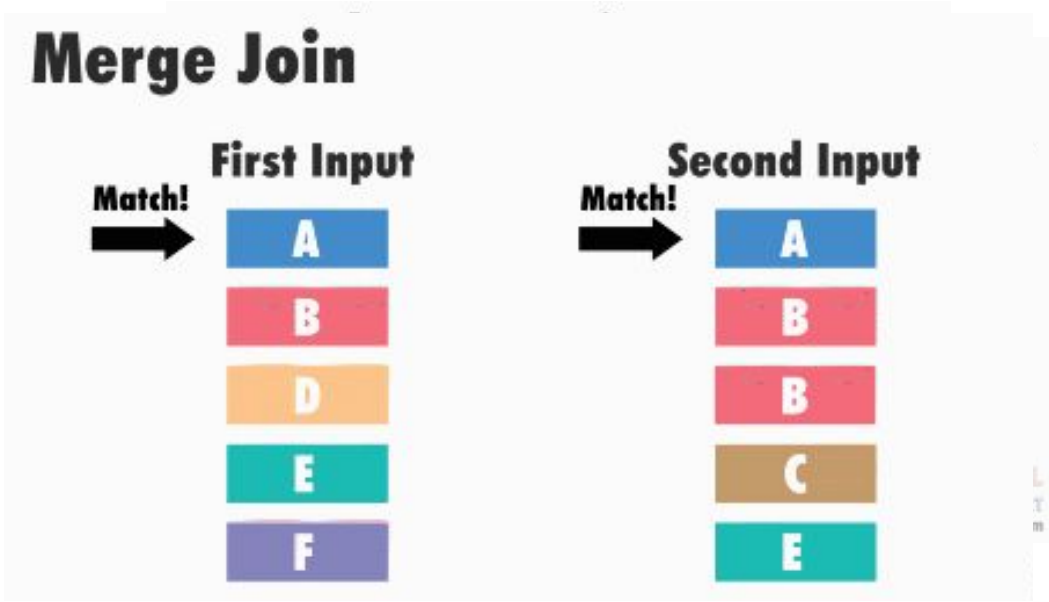
Join Order

- Sequence in which tables are joined together
- Impacts query performance
- Shown in query execution plan
- Database query optimizer chooses the most efficient join order
- To force the planner to follow the join order laid out by explicit JOINs, set the `join_collapse_limit` run-time parameter to 1
- Join priority:
  - Smaller tables
  - Tables with filters
  - Tables with selective join conditions
  - Tables with appropriate indexes on join columns



Join Algorithms

- Nested Loop Join
  - For each row scan the entire table
  - Small tables or on indexed columns
- Hash Join
  - Build hash tables based on the join key and find matches
  - Large tables and non-indexed join keys
- Sort-Merge Join
  - Sort both input tables based on the join key and merge them
  - Large tables with indexed join keys



Subqueries

- Nested queries
- Replaces DISTINCT

```
SELECT
  clients.code
FROM
  clients
WHERE
  id IN (
    SELECT
      "clients"."id"
    FROM
      "clients"
      INNER JOIN "client_users" ON "client_users"
          ."client_id" = "clients"."id"
      INNER JOIN "users" ON "users"."id" =
          "client_users"."user_id"
    WHERE
      users.last_sign_in_at > '2024-03-01'
    GROUP BY
      "clients"."id"
  )
```

CTE

- Improves readability and allows reusability
- Performance optimization

```sql
WITH last_clients AS (
  SELECT
    clients.id,
    clients.code
  FROM
    clients
    INNER JOIN client_users ON
        client_users.client_id = clients.id
    INNER JOIN users ON
        users.id = client_users.user_id
  WHERE
    users.last_sign_in_at > '2024-03-01'
  GROUP BY
    clients.id,
    clients.code
)
SELECT
  last_clients.id,
  last_clients.code,
  sum(bills.amount) as total_amount
FROM
  last_clients
  INNER JOIN bills ON last_clients.id = bills.client_id
WHERE
  bills.paid_at > '2024-03-01'
GROUP BY
  last_clients.id,
  last_clients.code
HAVING
  sum(bills.amount) > 10000;
```

Recursive Queries

- Used to access nested data
- Used to traverse trees

```
WITH RECURSIVE directory_tree AS (
  SELECT
    id,
    name,
    parent_id,
    1 AS level
  FROM
    doc_dirs
  WHERE
    parent_id IS NULL
  UNION ALL
  SELECT
    d.id,
    d.name,
    d.parent_id,
    dt.level + 1
  FROM
    doc_dirs d
    INNER JOIN directory_tree dt
        ON d.parent_id = dt.id
)
SELECT
  id,
  name,
  level
FROM
  directory_tree
WHERE
  level > 3;
```

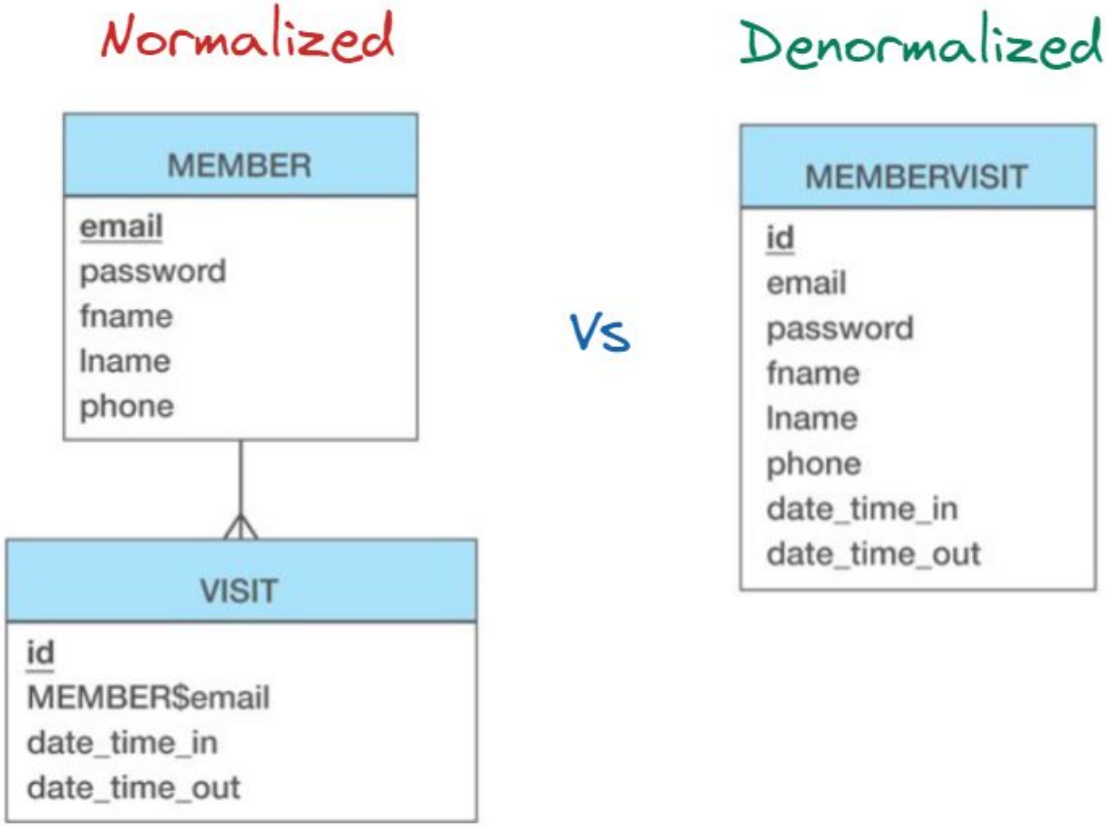
Normalization

Problems:

- Data redundancy
- Update anomalies
- Data inconsistency

Solution:

- Normal Forms



First Normal Form (1NF)

- Simple attributes
- No repeating groups

Students

FirstName	LastName	Knowledge
Thomas	Mueller	Java, C++, PHP
Ursula	Meier	PHP, Java
Igor	Mueller	C++, Java

Startsituation

Result after Normalisation

Students

FirstName	LastName	Knowledge
Thomas	Mueller	C++
Thomas	Mueller	PHP
Thomas	Mueller	Java
Ursula	Meier	Java
Ursula	Meier	PHP
Igor	Mueller	Java
Igor	Mueller	C++

Second Normal Form (2NF)

- 1NF
- Each non-key attribute is dependent on primary key

Students

IDSt	LastName	IDProf	Prof	Grade
1	Mueller	3	Schmid	5
2	Meier	2	Borner	4
3	Tobler	1	Bernasconi	6

Startsituation

Result after normalisation

Students

ID	LastName
1	Mueller
2	Meier
3	Tobler

Professors

IDProf	Professor
1	Bernasconi
2	Borner
3	Schmid

Grades

IDStIDProf	Grade	
1	3	5
2	2	4
3	1	6

Third Normal Form (3NF)

- 2NF
- Removes transitive dependencies
- No non-key attribute is dependent on another non-key attribute

Vendor

ID	Name	Account_No	Bank_Code_No	Bank

Startsituation

Result after normalisation

Vendor

ID	Name	Account_No	Bank_Code_No

Bank

Bank_Code_No	Bank

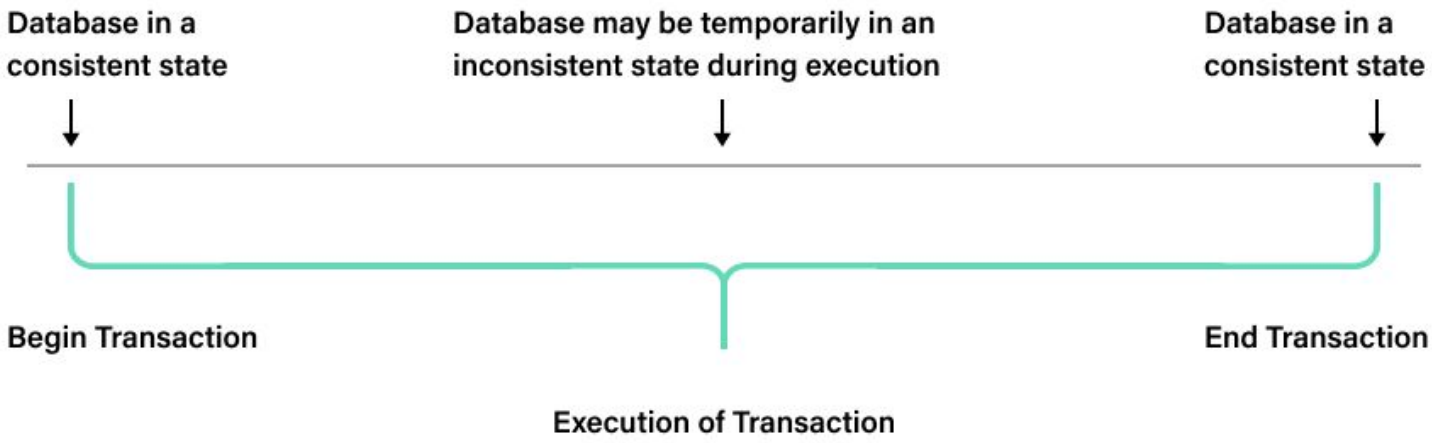
3 N F

Course code	Date	Seat	Remain	Room	Rcapa
C125	12-01-2014	12	5	101	15
DS144	12-01-2014	45	22	102	50
C125	21-07-2014	15	11	101	15
J678	08-11-2014	15	2	102	50

Course code	Cname
C125	C Programming
DS144	Data Structure
J678	Java Programming

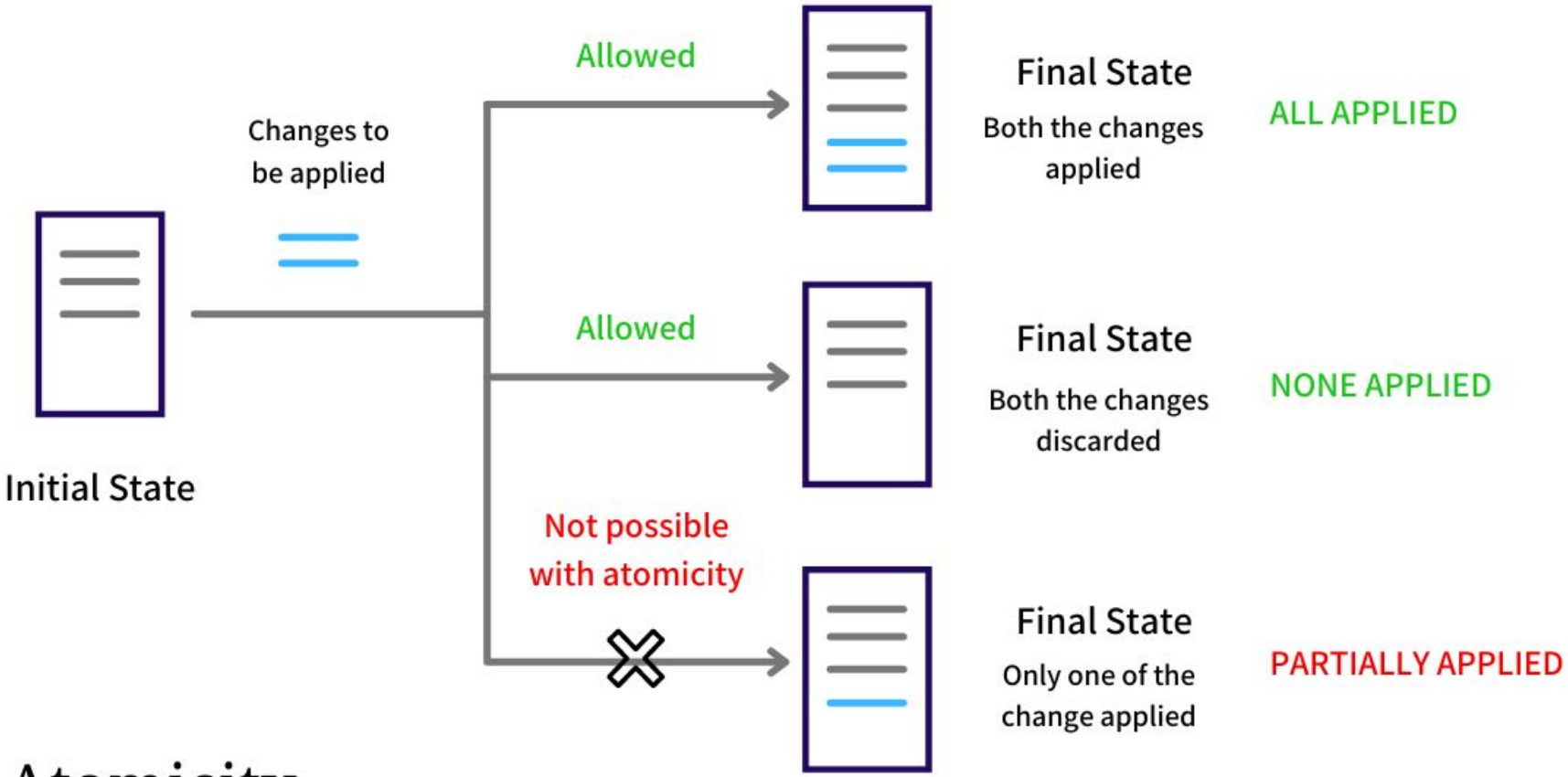
Transactions

- Must be completed entirely or not at all

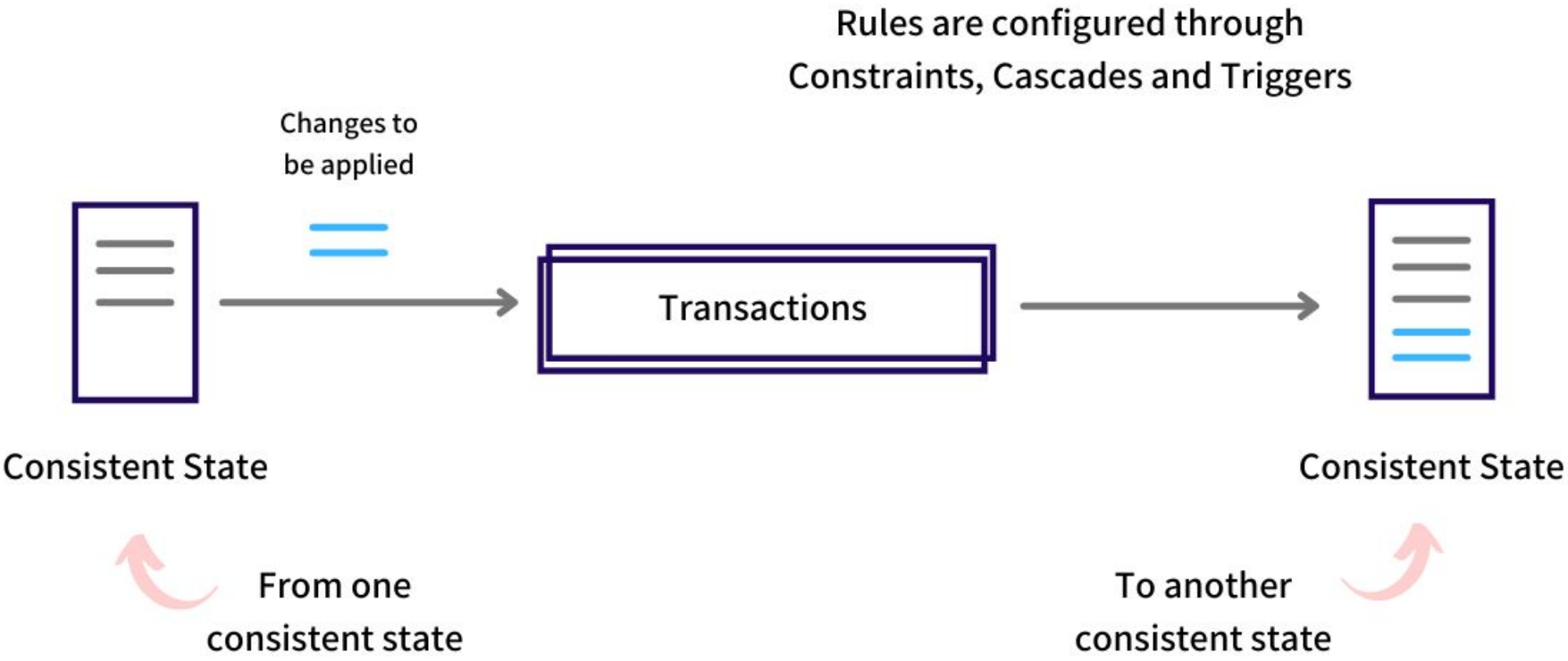


ACID

- Atomicity
  - complete successfully or fail and rollback
- Consistency
  - consistent state before and after execution
- Isolation
  - transactions are isolated from each other
  - changes are invisible until committed
- Durability
  - once commits its permanent even if system fails

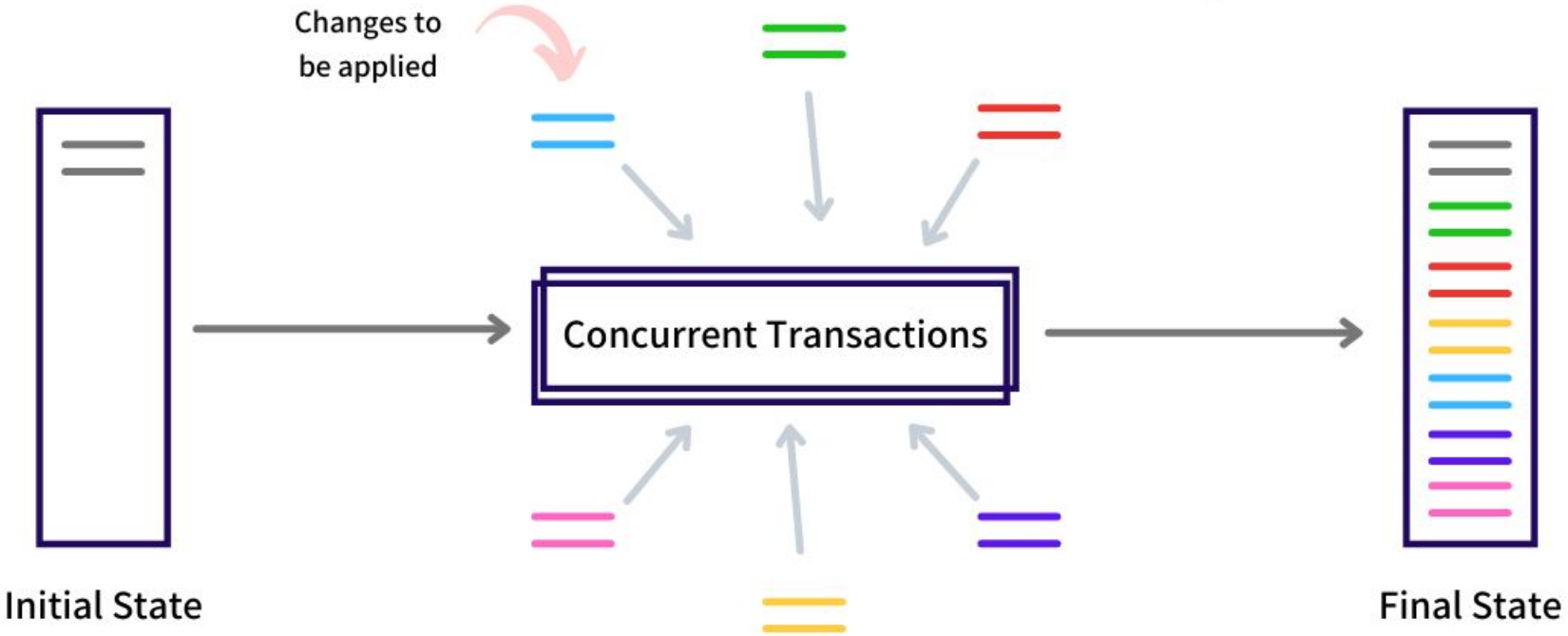

Changes to be applied
Initial State
Allowed
Final State
Both the changes applied
ALL APPLIED
Allowed
Final State
Both the changes discarded
NONE APPLIED
Not possible with atomicity
Final State
Only one of the change applied
PARTIALLY APPLIED

Atomicity

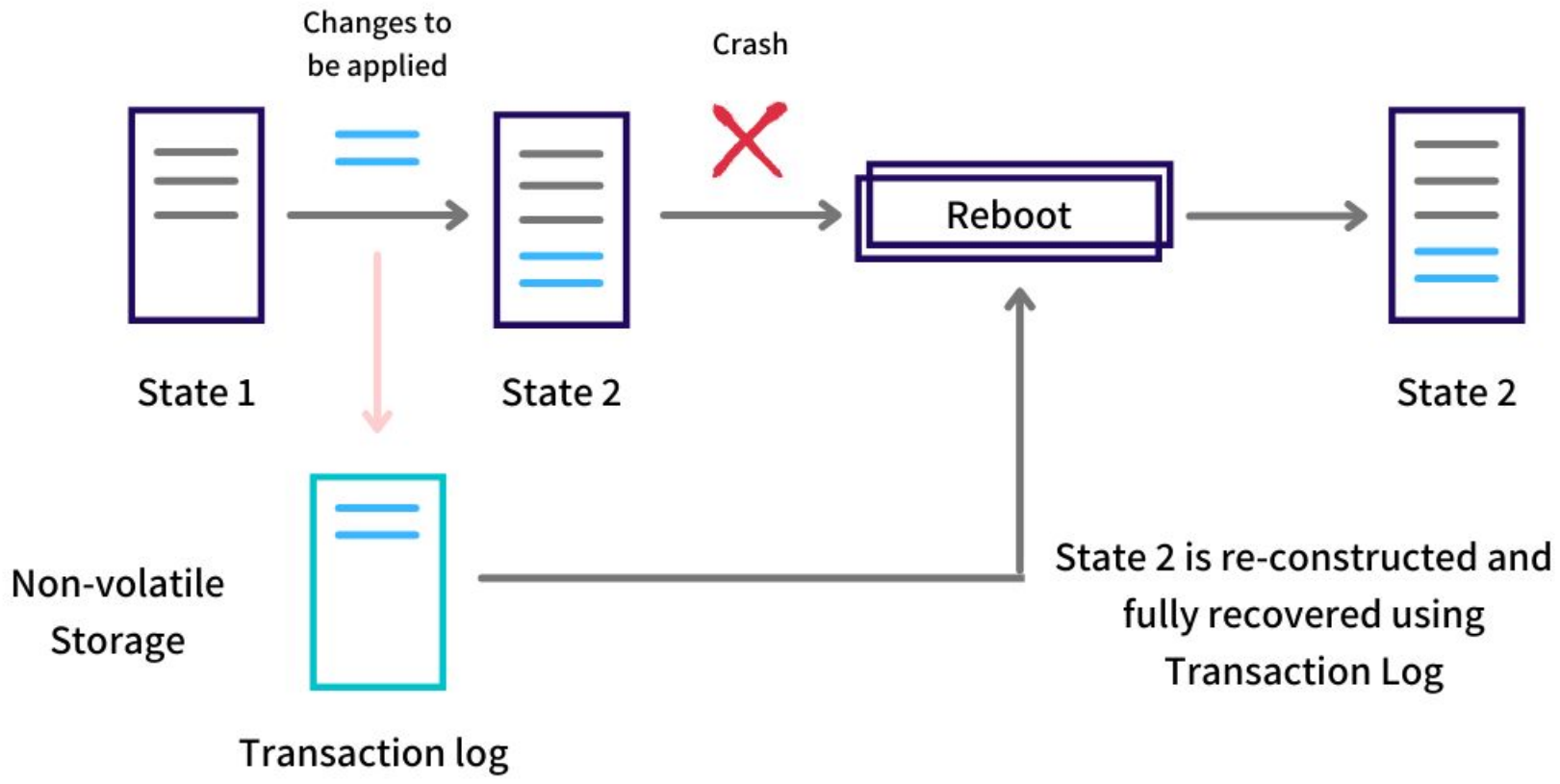


Consistency

Isolation is achieved through locking - exclusive and shared.



Isolation


Changes to
be applied
Crash
Reboot
State 1
State 2
State 2
Non-volatile
Storage
State 2 is re-constructed and
fully recovered using
Transaction Log
Transaction log

Durability

Read Errors and Isolation Levels

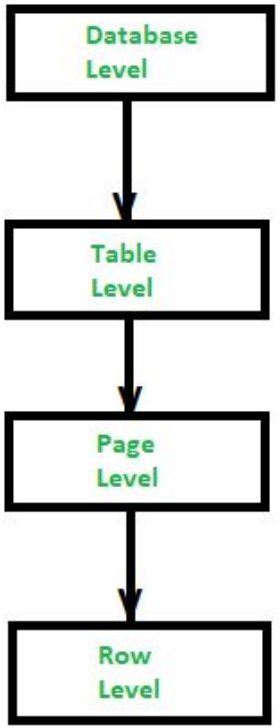
- Dirty Read
  - transaction reads uncommitted data
  - changes rollback
- Lost Update
  - two transactions concurrently read and update same data
  - one change overwrites another
- Non-Repeatable Read
  - read multiple times data that changes between reads
- Phantom Read
  - transaction reads a set of rows
  - other transaction inserts rows to same table

Isolation levels vs read phenomena [edit]

Isolation level \ Read phenomena	Dirty reads	Lost updates	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur	may occur
Read Committed	don't occur	may occur	may occur	may occur
Repeatable Read	don't occur	don't occur	don't occur	may occur
Serializable	don't occur	don't occur	don't occur	don't occur

Locks

- Control access to data during transactions
- Preventing conflicting operations
- Types:
  - Shared Lock (S) - blocks writing
  - Exclusive Lock (X) - blocks reading and writing
  - Update Lock (U) - blocks writing, can block reading
  - Schema Locks - access control



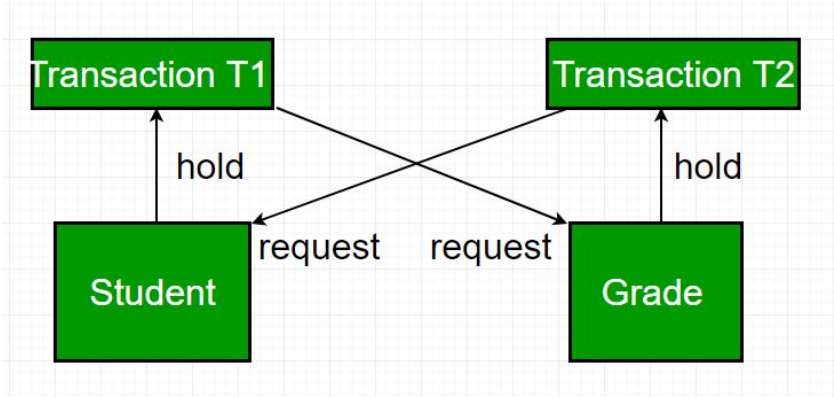
Deadlocks

Problem:

- Multiple transactions waiting to release a lock cause by
  - concurrent transactions
  - circular dependencies
  - inadequate locking strategies

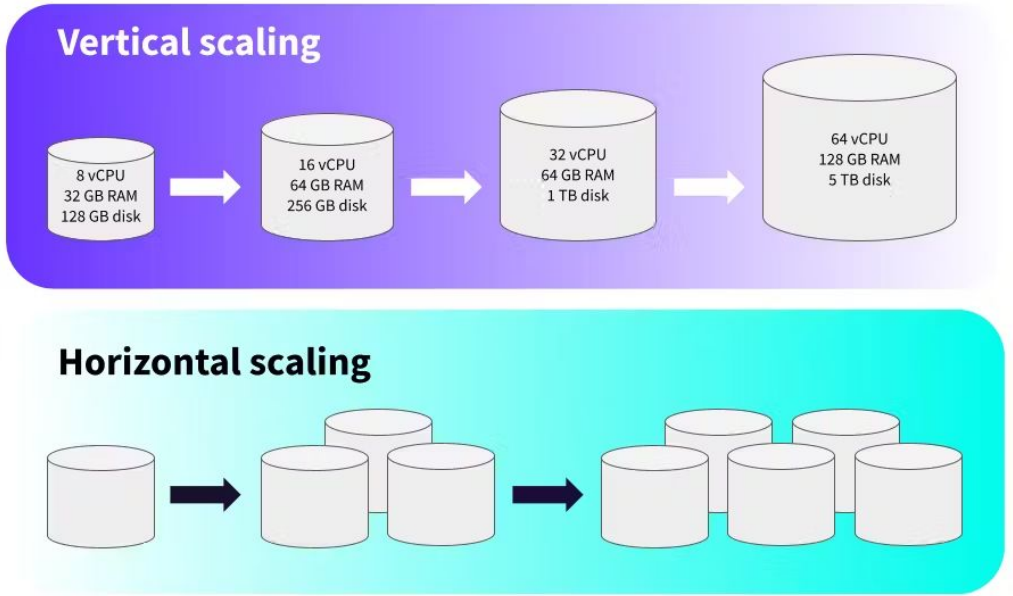
Solution:

- Detection:
  - Timeout Mechanisms
  - Deadlock Detection Algorithms
- Resolving:
  - Prevention
  - Avoidance
  - Resolution



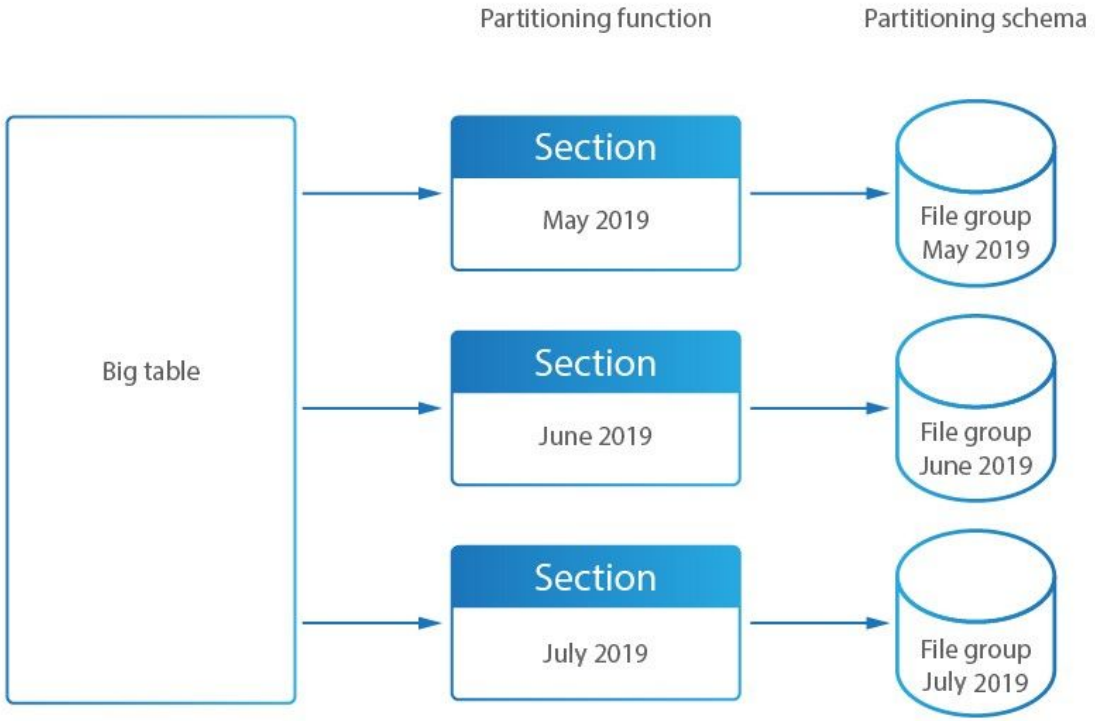
Scalability

- Vertical vs Horizontal
- Strategies:
  - Sharding
  - Replication
  - Partitioning
  - Caching
- Can improve performance
- Can lower consistency
- Harder maintenance



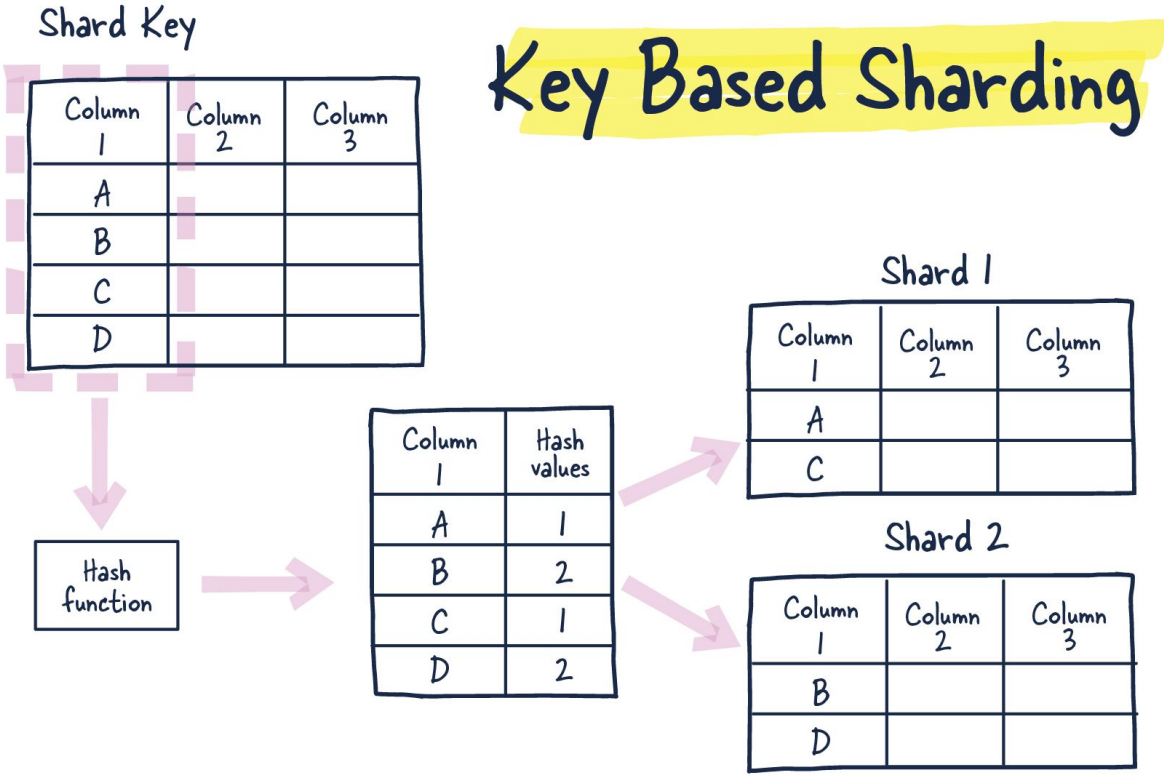
Partitioning

- Dividing a large database table into smaller partitions
- Division is done by keys
- Types:
  - Range Partitioning
  - List Partitioning
  - Hash Partitioning



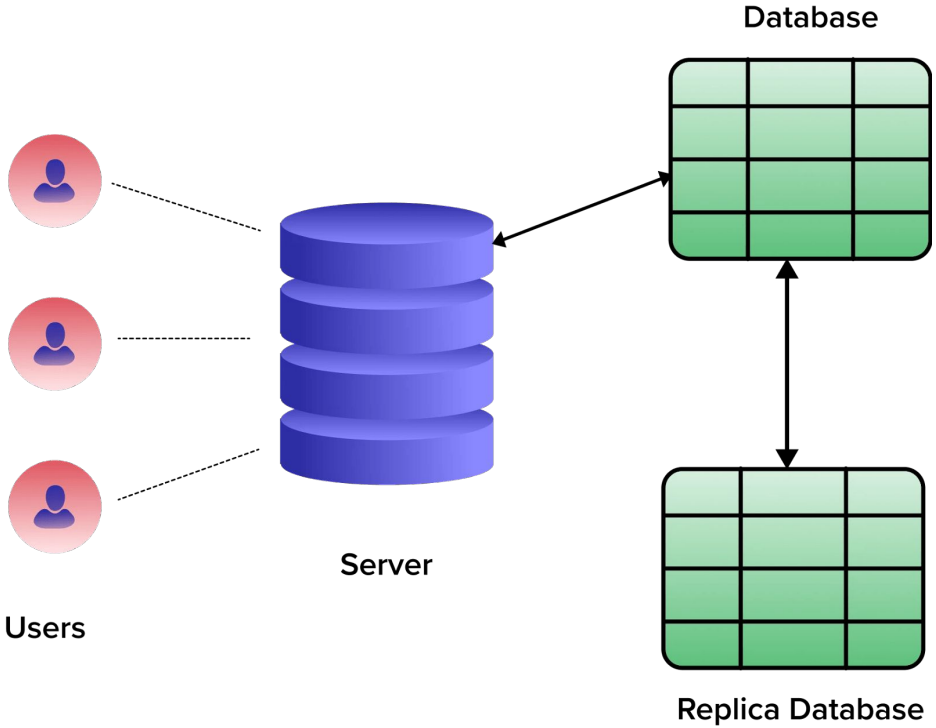
Sharding

- Dividing a large database into smaller
- Each shard is hosted on a separate database server
- Types:
  - Horizontal Sharding
  - Vertical Sharding
  - Key-Based Sharding



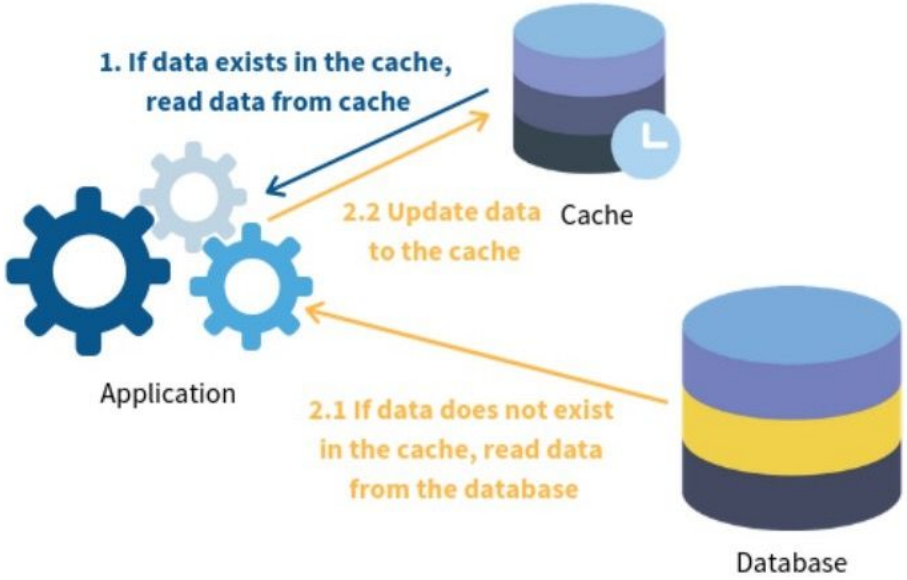
Replication

- Creating copies (replicas) of a database
- Master-Slave Replication
- Can be asynchronous or synchronous
- Can be full or partial



Caching

- Caching is the process of storing frequently accessed data in memory
- Types:
  - Query Result Caching
  - Object Caching



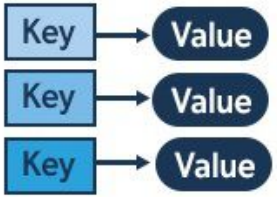
PostgreSQL

- ACID Compliance
- Extensibility
- Data Integrity
- Built-in Scalability
- Replication
- JSON and NoSQL Support
- Full Text Search
- Security
  - SSL/TLS encryption
  - Role-based access control
  - Row-level security

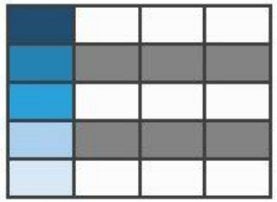
NoSQL

- NoSQL (Not Only SQL) - non-relational databases
- Designed to handle large volumes of unstructured or semi-structured data
- Not suited for complex queries
- Weak persistence
- Basic transactions

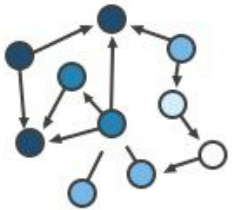
Key-Value



Column-Family



Graph



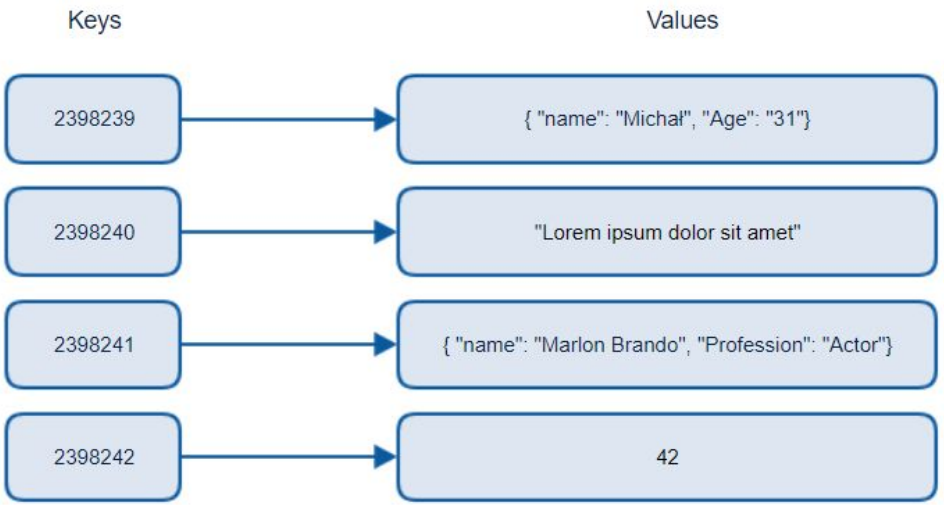
Document

SQL vs NoSQL

Feature	SQL Databases	NoSQL Databases
Key Focus	Reducing data duplication	Scaling and rapid application change
Data Storage Model	Tables with fixed rows and colums	Document: JSON documents; Key-value: key-value pairs; Wide-column: tables with rows and dynamic columns; Graph: nodes and edges
Schemas	Rigid	Flexible
Data to Object Mapping	Requires ORM (object-relational mapping)	Typically doesn't require ORMs. E.g. MongoDB documents map directly to data structures in popular programming languages.
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)

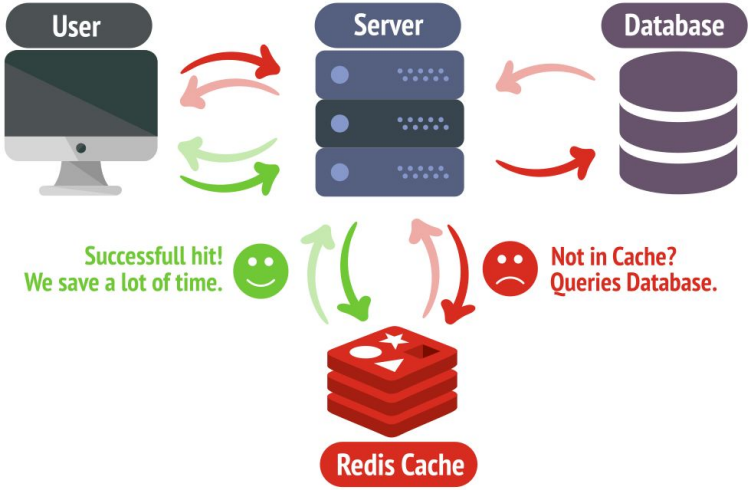
Key-Value Databases

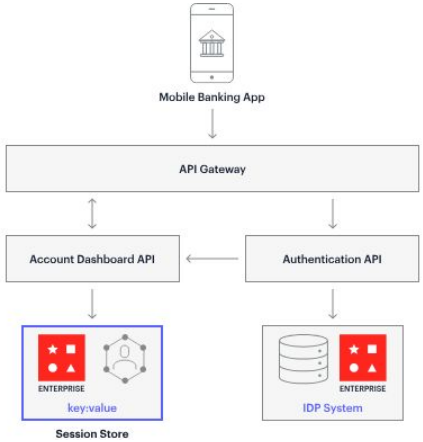
- Collection of key-value pairs
- Unique keys
- High read and write throughput
- Scales horizontally
- Use cases:
  - Caching
  - Session Management
  - Shared Memory
  - Real-time Analytics



Redis

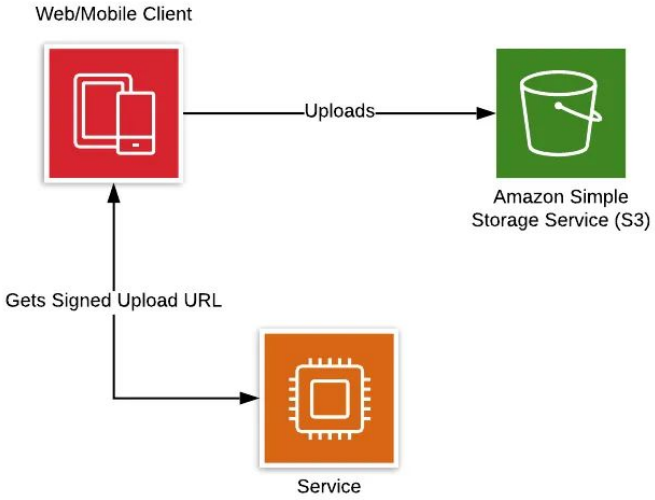
- Key-value database
- Stored in RAM
- Supports data structures
- Persistence strategies
- Supports clustering
- Supports pub/sub messaging
- Supports atomic operations with Lua





S3

- Amazon S3 (Simple Storage Service) - object storage service
- Key-value database
- Can store virtually unlimited data
- Designed for 99.999999999% (11 nines) durability
- Supports object versioning
- Clustered

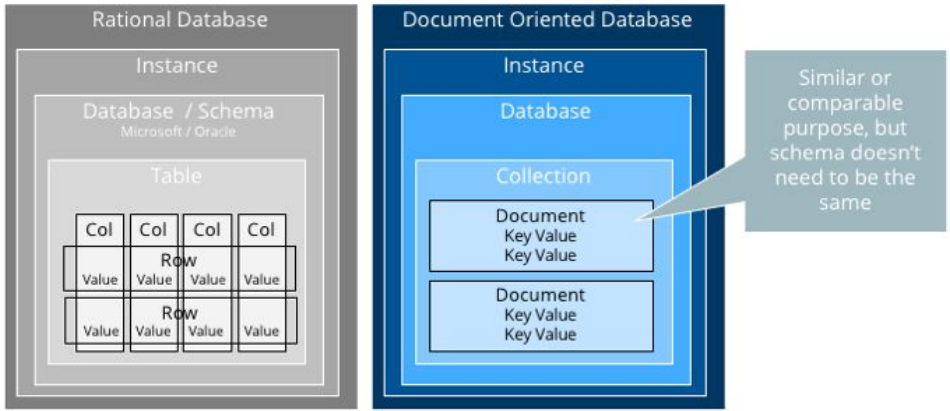


Column-Oriented Databases

- Store and query data by column rather than by row
- Best at aggregating operations
- Efficient compression and encoding
- Bulk loading data is efficient
- Single operations are slower
- Horizontal Scalability
- Suited for analytics

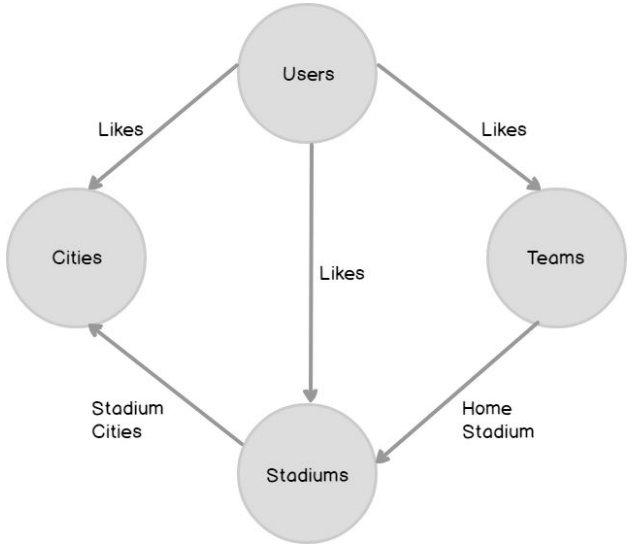
Document Databases

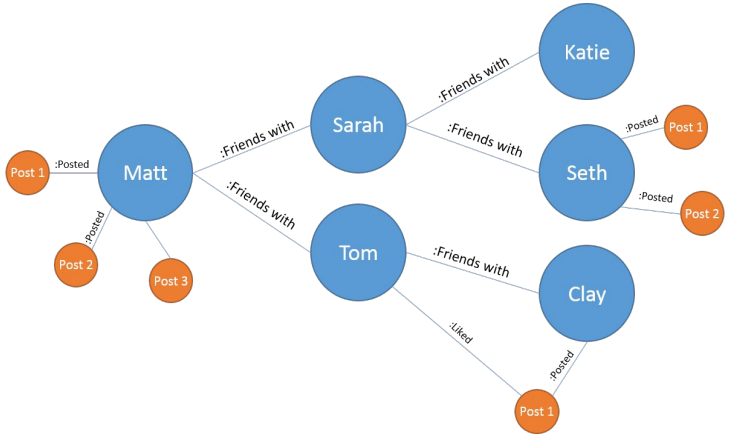
- Store data in semi-structured format known as documents
- Flexible evolving shemas
- JSON native, support nested structures
- Horizontal Scalability
- Support indexing
- Supports links (like relations)
- Can support complex queries
- Can support ACID transactions
- Suited for unstructured data



Graph Databases

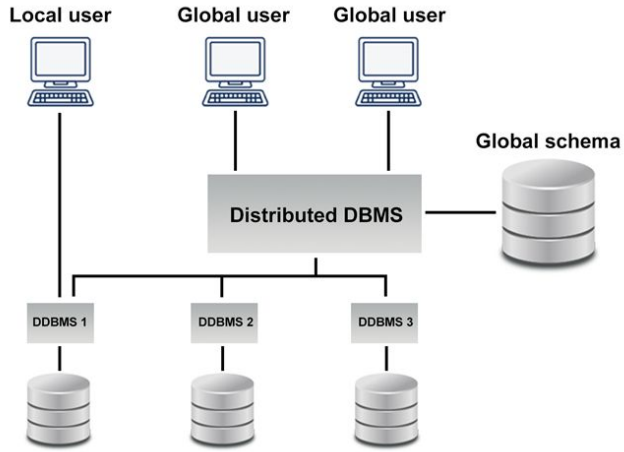
- Store data as graphs, consisting of nodes, edges, and properties
- Relationships between nodes are first-class citizens
- Properties add flexibility to the data model
- Horizontal Scalability
- Suited for highly interconnected data





Distributed Databases

- Store large volumes of data across multiple servers
- Scale horizontally by adding more nodes
- Replicate data to ensure fault tolerance and data durability
- Distribute data across nodes to achieve load balancing and efficient query processing
- Provide global reach
- Data partitioning can be problematic
- Consistency vs Availability
- Can lead to high latency
- Suited for the highest level scaling



Thank you for your attention!