Relational Database

SELECT
```
SELECT first_name, last_name FROM employees;
```
WHERE
```
SELECT * FROM employees WHERE department = 'Sales';
```
ORDER BY
```
SELECT * FROM employees ORDER BY hire_date DESC;
```
COUNT, SUM, AVG, MIN, MAX
```
SELECT COUNT(*), SUM(salary), AVG(salary),
MIN(salary), MAX(salary) FROM employees;
```
INNER JOIN
```
SELECT * FROM table1 INNER JOIN table2 ON table1.id =
table2.id;
SELECT employees.first_name,
departments.department_name
FROM employees
INNER JOIN departments ON employees.department_id
= departments.id;
```
LEFT JOIN
```
SELECT * FROM table1 LEFT JOIN table2 ON table1.id =
table2.id;
```
INSERT INTO
```
INSERT INTO table_name (column1, column2) VALUES
(value1, value2);
```
UPDATE
```
UPDATE table_name SET column1 = value1 WHERE
condition;
UPDATE employees SET salary = 55000 WHERE
first_name = 'John' AND last_name = 'Doe';
```
DELETE
```
DELETE FROM table_name WHERE condition;
```
CREATE TABLE,PRIMARY KEY, FOREIGN KEY
```
CREATE TABLE employees (
  id INT PRIMARY KEY,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
salary DECIMAL(10, 2),
department_id INT,
FOREIGN KEY (department_id) REFERENCES
departments(id));
```
ALTER TABLE
```
ALTER TABLE table_name ADD column_name datatype;
ALTER TABLE employees ADD email VARCHAR(100);
```
DROP TABLE
```
DROP TABLE table_name;
DROP TABLE employees;
```
UNIQUE
```
ALTER TABLE table_name ADD CONSTRAINT
constraint_name UNIQUE (column_name);
ALTER TABLE employees ADD CONSTRAINT
unique_email UNIQUE (email);
```
NOT NULL
```
ALTER TABLE table_name MODIFY column_name
datatype NOT NULL;
ALTER TABLE employees MODIFY email VARCHAR(100)
NOT NULL;
```
GROUP BY and HAVING
```
SELECT department, COUNT(*)
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```
BEGIN TRANSACTION, COMMIT, ROLLBACK
```
BEGIN TRANSACTION;
UPDATE employees SET salary = salary * 1.1 WHERE
department = 'Sales';
COMMIT; -- or use ROLLBACK; to undo changes
```
GRANT
```
GRANT SELECT, INSERT, UPDATE, DELETE ON employees
TO 'username' IDENTIFIED BY 'password';;
GRANT ALL PRIVILEGES ON employees TO 'username';
REVOKE SELECT, INSERT ON employees FROM
'username';
```
VIEW
```
CREATE VIEW sales_employees AS
SELECT first_name, last_name FROM employees WHERE
department = 'Sales';
```
CASE Statement
```
SELECT first_name,
  CASE
    WHEN dep = 'Sales' THEN 'Sales Team'
    WHEN dep = 'Acc' THEN 'Acc Team'
    ELSE 'Other'
  END as team
FROM employees;
```
LIMIT / OFFSET
```
SELECT * FROM staffs LIMIT 10 OFFSET 5;
```
DISTINCT
```
SELECT DISTINCT dep FROM employees;
```
INDEX
```
CREATE INDEX idx_employee_last_name ON employees
(last_name);
```

PROS

- Structured and organized data with tables, rows, and columns.
- Strong support for data integrity and consistency through constraints and transactions.
- Powerful querying capabilities with SQL.
- Well-established, widely used, and supported by many tools and applications.

CONS

- Limited flexibility with schema changes; altering the structure can be complex.
- Not well-suited for hierarchical or unstructured data.
- Scaling horizontally can be challenging compared to NoSQL databases.
- Requires predefined schemas, which may not accommodate all types of data efficiently.

OPTIMISATION & SECURITY

Optimization: Use indexing and query optimization to improve data retrieval speeds and performance.
Normalization: Reduce redundancy and maintain data integrity; use denormalization selectively for read-heavy applications.
Security: Implement strong authentication, authorization, and encryption for data protection.
Monitoring: Regularly audit, update, and patch systems to address vulnerabilities and detect security incidents.

NORMALISATION

First Normal Form (1NF): Ensures each column contains atomic (indivisible) values and each row is unique, eliminating repeating groups or arrays.
Second Normal Form (2NF): Builds on 1NF by ensuring that all non-key attributes are fully dependent on the entire primary key, eliminating partial dependencies.
Third Normal Form (3NF): Builds on 2NF by ensuring that all non-key attributes are only dependent on the primary key, eliminating transitive dependencies.
Boyce-Codd Normal Form (BCNF): A stricter version of 3NF where every determinant is a candidate key, addressing certain anomalies not covered by 3NF.

FUNTIONAL DEPENDENCIES

Full Dependencies - All Composite Key
an attribute is fully dependent on a composite primary key

Partial Dependencies - Part of Composite Key
a non-key attribute depends on only a part of a composite primary key

Transitive Dependencies (A→B, B→C, So A→C)
a non-key attribute depends on another non-key attribute indirectly through a third attribute, forming a chain of dependencies

Trivial Dependencies {Name, Email} → Name

Non-Trivial Dependencies - (ID → Name)
A specific type of functional dependency where the dependent attribute is not part of the determinant attribute.

Semantic Database

XML: eXtensible Markup Language, hierarchical tree data structure

XML Namespaces

```
<root xmlns:h="http://www.w3.org/TR/html4/">
 <h:table>
  <h:tr>
   <h:td>Apples</h:td>
   <h:td>Bananas</h:td>
  </h:tr>
 </h:table>
</root>
```

Special Characters in XML text

```
<Demo1>Bill &amp; Melinda</Demo1>
 # &lt;, &gt;, &quot; &apos;
<Demo4 test="&quot;Demo4A&quot;">Demo4
Value</Demo4>
<Demo6><![CDATA[
 You can put any junk like & or 1 < 2, also blank lines
]]></Demo6>
```

XPath

For text --->
```
/Reservation/Flight[1]/FlightLeg[1]/ArrivalDate[1]/text()
```
For "where" --> `/Reservation/Flight[@seq=2]`

For query element condition -->
```
//FlightLeg[FlightNo='2105' and DepartureAirport='STL']
```
Go up one level using .. -->
```
//Flight/FlightLeg[FlightNo=1849]/../FlightLeg[@seq=2]
```
Extract attribute --> `//Employee[@empID=189]/@dept`

Wildcard Search
```
//*[starts-with(./text(), 'OK')]
//*[starts-with(./text(), 'OK')]/../FlightNumber/text()
//*[substring-before(./text(), '-') = '2019']
```
Aggregate
```
sum(//Reservation/TotalPrice)
sum(//Item[@origin='MEXICO']/@price)
```
Local Name
```
<!--local-name() for <ns1:Airline> would return Airline-->
//*[contains(local-name(), 'Airline')]
```

XQuery

Selecting Elements
```
for $book in doc("books.xml")//book return $book/title
```
Filtering with conditions
```
for $book in doc("books.xml")//book where
$book/@category = "fiction" return $book/title
```
Sorting Results
```
for $book in doc("books.xml")//book
order by $book/title
return $book/title
```
Grouping Data
```
for $author in distinct-
values(doc("books.xml")//book/author)
return
  <author>
    <name>{$author}</name>
    <books>
      {for $book in doc("books.xml")//book[author =
$author]
       return <book>{$book/title}</book>}
    </books>
  </author>
```
Conditional Logic
```
for $book in doc("books.xml")//book
return
  if ($book/price > 50)
  then <expensive>{$book/title}</expensive>
  else <affordable>{$book/title}</affordable>
```
Using Let Clause for Reusability
```
let $books := doc("books.xml")//book
return
  for $book in $books
  where $book/price > 30
  return $book/title
```
Aggregating Data
```
let $total := sum(doc("books.xml")//book/price)
return <totalPrice>{$total}</totalPrice>
```
Constructing New XML Elements
```
for $book in doc("books.xml")//book
return
  <summary>
    <title>{$book/title}</title>
    <author>{$book/author}</author>
    <price>{$book/price}</price>
  </summary>
```
Extracting Substrings and Manipulating Text
```
for $title in doc("books.xml")//book/title
return fn:substring($title, 1, 10)
```
Joining Data from Multiple Documents
```
for $book in doc("books.xml")//book,
  $author in doc("authors.xml")//author
where $book/author = $author/name
return <book-author>
     <title>{$book/title}</title>
     <author>{$author/name}</author>
   </book-author>
```

XSLT

```
<body>
  <h2>Product Catalog</h2>
  <ul>
    <xsl:for-each select="catalog/product">
      <li>
        <xsl:value-of select="name"/> - $<xsl:value-of
select="price"/>
      </li>
    </xsl:for-each>
  </ul>
</body>
```

PROS

- Flexible and self-descriptive format, suitable for both structured and semi-structured data.
- Platform-independent and widely used for data interchange between systems.
- Supports hierarchical data representation, making it good for nested data structures.
- Can be validated against schemas (XSD) to ensure data correctness.

CONS

- Verbose and can lead to large file sizes, impacting performance and storage.
- Parsing XML can be slower compared to other formats like JSON.
- Does not support complex querying or indexing natively like relational databases.
- Lack of native support for relationships between data items, unlike relational databases.

Object Databases and JSON

JSON Data Structure

```
{"note": {
  "to": "Tove",
  "body": "Don't forget me this weekend!}}
```

JSON Query

```
db.collection.find({ "note.to": "Tove" });
```

JSON PROS

Alignment with Object-Oriented Models: Both support complex, nested, and hierarchical data structures, making them suitable for object-oriented applications.

Flexibility and Readability: JSON's human-readable format and the object database's schema flexibility allow for easy data representation and manipulation.

Performance: Object databases reduce impedance mismatch, and JSON's direct compatibility with APIs simplifies data exchange and integration with web technologies.

JSON CONS

Complexity and Standards: Object databases can be complex to manage, and both lack universal query standards, leading to potential compatibility and maintenance challenges.

Data Redundancy and Consistency Issues: Lack of strict schema enforcement in JSON can cause data inconsistencies, while object databases may struggle with data redundancy without proper management.

Performance and Size: JSON can be slower and larger in size compared to structured formats, which may impact performance in data-intensive applications.

MongoDB

Advantages: MongoDB's flexible schema would allow for more diverse data entries, such as varying fields for names or additional personal information, and it can efficiently store hierarchical data, like embedding test results within student records.
Disadvantages: The lack of enforced schemas could lead to inconsistencies in data entry, and querying relational data, like aggregating test results across students, can be more complex and slower compared to a relational database system like MySQL.

Linked Data

RDF (Resource Description Framework)

- RDF represents data as triples (subject, predicate, object) to describe relationships.
- Uses URIs to uniquely identify resources, with literals for values (e.g., strings, numbers).
- Common syntaxes include XML, Turtle, and JSON-LD for representing RDF data.
- Example: To describe "Alice knows Bob," use <http://example.org/Alice> <http://xmlns.com/foaf/0.1/knows> <http://example.org/Bob> .

RDF Schema (RDFS)

- A vocabulary extension of RDF providing basic elements for defining ontologies.
- Defines classes, properties, and relationships, allowing for inference and reasoning about data.
- Key elements: rdfs:Class (defines a class), rdfs:subClassOf (class hierarchy), rdfs:Property (defines a property).

Relax NG schema

Defines the structure of <text:list>, including its attributes, optional headers, and multiple <text:list-item> elements, validating the document's adherence to this structure.

```
<element name="book"
xmlns="http://relaxng.org/ns/structure/1.0">
  <element name="title">
    <text/>
  </element>
  <element name="price">
    <data type="decimal"/>
  </element>
</element>
```

SPARQL

- A query language for querying RDF data, similar to SQL for relational databases.
- Allows retrieval and manipulation of RDF data using SELECT, ASK, CONSTRUCT, and DESCRIBE queries.
- Uses pattern matching with triple patterns to extract information from RDF graphs.

```
SELECT ?subject ?predicate ?object
WHERE {  ?subject ?predicate ?object .}
```
```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person
WHERE {<http://example.org/Alice> foaf:knows ?person.}
```

TURTLE

Simplicity and readability compared to RDF/XML

- A compact and human-readable syntax for RDF data.
- Uses prefixes to shorten URIs and supports shorthand notations for repeated subjects and predicates.
- Commonly used due to its simplicity and readability compared to other RDF serializations like RDF/XML.

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
ex:Alice a foaf:Person ;
      foaf:name "Alice" ;
      foaf:knows ex:Bob .
ex:Bob a foaf:Person ;
     foaf:name "Bob" .
```

PROS

- Designed for representing information about resources in a graph structure, suitable for linked data.
- Supports semantic data and relationships, enabling complex queries using SPARQL.
- Facilitates data integration from multiple sources and is flexible in handling evolving schemas.
- Well-suited for web data and linked data applications, supporting interoperability.

CONS

- Can be complex and challenging to manage due to its graph-based nature.
- Requires understanding of RDF syntax, RDF Schema, and SPARQL, which have a steep learning curve.
- Performance issues can arise with large datasets due to the triple-store architecture.
- Not as widely adopted or supported as relational databases, limiting available tools and resources.