Document DBs and MongoDB

Document Database Fundamentals

- **Definition and Core Concepts**:
    - A non-relational database that stores data as structured documents, usually in JSON format
    - Designed to be simple, flexible, and scalable
    - Schema-less (or schema-flexible) approach allowing each document to have a different structure
    - Documents are self-describing, containing both data and structure in one unit
    - Optimized for developer productivity over strict data consistency
- **Advantages Over Relational Databases**:
    - Addresses the "impedance mismatch" between object-oriented programming and tabular data storage
    - Natural alignment with modern programming paradigms (objects map directly to documents)
    - Simplifies storage of complex, hierarchical data structures
    - Flexible schema allows for rapid application evolution and iteration
    - Typically scales horizontally more easily than traditional RDBMS
    - No need to deconstruct complex objects across multiple tables

JSON and BSON

JSON (JavaScript Object Notation)

- **Characteristics**:
    - Lightweight, human-readable data-interchange format
    - Built on two universal data structures:
        - Collections of name/value pairs (objects, dictionaries, hash tables, etc.)
        - Ordered lists of values (arrays, vectors, sequences, etc.)
    - Language-independent despite JavaScript origins
    - Widely supported across programming languages and platforms
    - Self-describing and easy to parse
- **Limitations for Database Use**:
    - Limited data type support (no native date type, binary data support)
    - Text-based format has parsing overhead
    - No schema validation capabilities built-in
    - Less efficient for storage and processing than binary formats

BSON (Binary JSON)

- **Advancements Over JSON**:
  - Binary-encoded serialization of JSON-like documents
  - Extends JSON with additional data types (Dates, Binary data, ObjectID, etc.)
  - Designed specifically for database needs with focus on:
    - Lightweight representation to minimize storage overhead
    - Traversable structure for efficient access patterns
    - Efficient encoding/decoding for performance
  - MongoDB's native storage format
- **Design Priorities**:
  - Performance optimization for database operations
  - Maintaining JSON-like document model while addressing technical limitations
  - Support for richer data types and structures
  - Efficient indexing and query execution

XML as a Predecessor

- **Historical Context**:
  - Precursor to JSON as a data exchange format
  - Widely used for configuration files and data interchange before JSON's rise
  - Combined with CSS to enable separation of content and presentation in web pages
- **Structural Characteristics**:
  - Tag-based markup similar to HTML but with extensible tag set
  - More verbose than JSON with opening/closing tags
  - Supports attributes and namespaces for additional metadata
- **Related Technologies**:
  - **XPath**: Query language for retrieving specific elements from XML documents
  - **XQuery**: SQL-equivalent language for interrogating XML documents
  - **DTD** (Document Type Definition): Schema language for XML structure validation
  - **XSLT**: Transformation language for converting XML to other formats
- **Legacy in Modern Systems**:
  - Still used in enterprise systems and specific domains
  - JSON has largely replaced XML for web APIs and data exchange
  - XML's verbosity made it less suitable for bandwidth-constrained environments

MongoDB Overview

- **Origins and Evolution**:
  - Started in 2007 by former DoubleClick engineers after Google acquisition
  - Created to address limitations of relational databases for high-volume ad serving (>400,000 ads/second)

    - Name derived from "humongous database" highlighting big data focus
    - MongoDB Atlas cloud service launched in 2016 (Database-as-a-Service)
- **Deployment Options**:
    - **MongoDB Atlas**: Fully managed cloud service (DBaaS)
    - **MongoDB Enterprise**: Subscription-based, self-managed with extended features
    - **MongoDB Community**: Source-available, free-to-use, self-managed edition
- **Core Architecture**:
    - Document-oriented storage model using BSON
    - Collections group related documents (similar to tables)
    - No enforced schema across documents in a collection
    - Rich query language with support for complex operations
    - Horizontal scaling through sharding

MongoDB Data Model

- **Hierarchical Structure**:
    - **Database**: Top-level container for collections
    - **Collections**: Groups of related documents (analogous to tables)
    - **Documents**: Individual records stored as BSON (analogous to rows)
    - **Fields**: Key-value pairs within documents (analogous to columns)
- **Document Characteristics**:
    - Self-contained units with all related data
    - No predefined schema requirements
    - Each document in a collection can have different fields/structure
    - Maximum document size of 16MB (larger data stored in GridFS)
    - Automatic generation of unique `_id` field if not specified
- **Relational vs. MongoDB Terminology**:

RDBMS	**MongoDB**
Database	Database
Table/View	Collection
Row	Document
Column	Field
Index	Index
Join	Embedded Document or Reference
Foreign Key	Reference

Data Modeling Approaches

- **Embedding vs. Referencing**:
  - **Embedding**: Nested document structure for related data
    - Advantages: Single query retrieval, atomic operations
    - Best for: one-to-few relationships, data accessed together frequently
  - **Referencing**: Storing IDs that refer to documents in other collections
    - Advantages: No duplication, easier updates to shared data
    - Best for: many-to-many relationships, frequently changing data
- **Schema Design Considerations**:
  - Design for query patterns rather than normalized data
  - Denormalization often preferred for performance
  - Balance embedding vs. referencing based on data size, update frequency
  - Consider document growth over time to avoid frequent reallocations
  - Implement application-level validation for consistency

MongoDB Features

- **Rich Query Support**:
  - Comprehensive CRUD (Create, Read, Update, Delete) operations
  - Complex query conditions including comparison, logical operators
  - Regular expression support for text searching
  - Aggregation framework for advanced data processing
- **Indexing Capabilities**:
  - Support for primary and secondary indices on document fields
  - Compound, multikey, text, and geospatial index types
  - Index creation and optimization tools
  - Index hints for query optimization
- **Scalability Features**:
  - Horizontal scaling through sharding
  - Automatic load balancing across shards
  - Zone sharding for data locality
- **High Availability**:
  - Replica sets with automatic failover
  - Self-healing recovery
  - Read preference options for distributing workload

MongoDB Query Operations

Basic CRUD Operations

- **Create (Insert)**:

    - `insert_one()`: Insert single document
    - `insert_many()`: Batch insert multiple documents
    - Each operation returns information about inserted IDs
- **Read (Find)**:
    - `find()`: Returns cursor to matching documents
    - `find_one()`: Returns single matching document
    - Query filters specified as first parameter
    - Field projections as second parameter
- **Update**:
    - `update_one()`, `update_many()`: Modify existing documents
    - Update operators: `$set`, `$unset`, `$inc`, etc.
    - Upsert option for insert-if-not-exists semantics
- **Delete**:
    - `delete_one()`, `delete_many()`: Remove documents
    - Empty filter `{}` matches all documents (use with caution)

Query Operators

- **Comparison Operators**:
    - `$eq`: Equal to
    - `$gt`, `$gte`: Greater than, greater than or equal
    - `$lt`, `$lte`: Less than, less than or equal
    - `$ne`: Not equal
    - `$in`, `$nin`: In array, not in array
- **Logical Operators**:
    - `$and`: Logical AND
    - `$or`: Logical OR
    - `$not`: Logical NOT
    - `$nor`: Logical NOR (neither condition true)
- **Element Operators**:
    - `$exists`: Field existence
    - `$type`: Field type checking
- **Array Operators**:
    - `$all`: All elements match
    - `$elemMatch`: Element matches criteria
    - `$size`: Array size matching

Advanced Query Features

- **Aggregation Framework**:
    - Pipeline-based data processing model

  - Stages include: `$match`, `$group`, `$project`, `$sort`, `$limit`
  - Performs complex analytics directly in database
  - More powerful alternative to simple group-by operations
- **Text Search**:
  - Full-text search capabilities
  - Language-specific stemming and stop word handling
  - Relevance scoring
- **Geospatial Queries**:
  - Support for GeoJSON data
  - Proximity searches
  - Intersection and containment operations

MongoDB Tools and Interfaces

- **mongosh (MongoDB Shell)**:
  - Interactive command-line interface
  - JavaScript execution environment
  - CRUD operations and administrative commands
  - Script execution for automation
- **MongoDB Compass**:
  - Official GUI for MongoDB
  - Visual query builder and data exploration
  - Schema visualization and analysis
  - Performance monitoring and index suggestions
- **Third-Party Tools**:
  - DataGrip and other database IDEs
  - Specialized visualization and management tools
  - Integration with monitoring systems
- **Language Drivers**:
  - Official drivers for major programming languages
  - PyMongo for Python
  - Mongoose for JavaScript/Node.js
  - Consistent API across languages

PyMongo

- **Core Functionality**:
  - Official Python driver for MongoDB
  - Pythonic interface to MongoDB operations
  - Maps Python dictionaries directly to MongoDB documents

Connection Setup:

```
from pymongo import MongoClient
client = MongoClient('mongodb://username:password@localhost:27017')
db = client['database_name']
collection = db['collection_name']
```

Basic Operations:

Document insertion:
```
post = {"author": "Mark", "text": "MongoDB is Cool!", "tags": ["mongodb", "python"]}
post_id = collection.insert_one(post).inserted_id
```

Document retrieval:
```
 # Find all documents matching criteria
results = collection.find({"year": 2010})

# Find single document
result = collection.find_one({"name": "Davos Seaworth"})

# Count documents
count = collection.count_documents({"rated": "PG-13"})
```

Query Examples:

Complex filtering:

```
 # Documents with specific rating from a country with minimum score
results = collection.find({
    "countries": "Mexico",
    "imdb.rating": {"$gte": 7}
})

# Logical OR condition
results = collection.find({
    "year": 2010,
    "$or": [
        {"awards.wins": {"$gte": 5}},
        {"genres": "Drama"}
    ]
})
```

Projections:

```
 # Select only certain fields
results = collection.find(
   {"year": 2010},
   {"title": 1, "director": 1, "_id": 0}
)
```

Best Practices and Performance Considerations

- **Indexing Strategy**:
  - Create indices for frequently queried fields
  - Consider compound indices for multi-field queries
  - Avoid over-indexing (impacts write performance)
  - Use `explain()` to analyze query execution plans
- **Document Design**:
  Keep documents reasonably sized (under 16MB limit)
  - Group frequently accessed data together
  - Consider read/write patterns when deciding on embedding vs. references
  - Design for growth in document size over time
- **Performance Optimization**:
  - Use projections to limit returned fields
  - Batch operations when possible
  - Leverage covered queries (queries satisfied by indices)
  - Consider read/write concerns based on use case
- **Security Considerations**:
  - Use role-based access control
  - Encrypt sensitive data
  - Implement network security (firewalls, VPN)
  - Regular security audits and updates

These comprehensive notes cover both the fundamental concepts and practical aspects of document databases and MongoDB, synthesizing the material presented in the slide deck while incorporating additional context and practical examples for deeper understanding.