NoSQL Databases

Where We've Been

- ✓ Challenges of Big Data for relational DBs
- ✓ Distributed file systems (e.g., GFS, HDFS)
- ✓ MapReduce for processing Big Data
  - Distributed full-scans via `map` and `reduce` phases

But we still miss database convenience and functionality!

The Gap

We want to:

- Maintain structured or semi-structured data
- Use in our applications
- **Without** the full overhead of relational DBs
- **Thereby** surmounting Big Data challenges

Is there any hope?

Enter NoSQL Databases

✓ **Yes!** Over the past 20 years, several Big Data databases have emerged from Big Tech (Google, Facebook, etc.)

✓ Called **"NoSQL"** because they're not relational

✓ Four main types:

1. **Key-Value** (e.g., Google BigTable)
2. **Document** (e.g., MongoDB)
3. **Column-Oriented** (e.g., Facebook's Cassandra)
4. **Graph** (e.g., Neo4j)

Type 1: Key-Value Databases

Key-Value: Definition

> A NoSQL database that stores data as a collection of **key-value pairs**

Key Characteristics:

- The **key** acts as an identifier for the value
- The **value** is opaque to the database
  - Can be anything
  - Database doesn't interpret, index, or analyze it

Key-Value: Constraints

What's NOT Supported:

- ❌ No relationships among keys
- ❌ No foreign keys
- ❌ No transactions across keys
  - BigTable supports transactions for a **single** key-value only

Simple Operation Set:

- `Put`
- `Get`
- `Delete`

Key-Value: Example Data

Key	Value
product:P001	`{"name": "Wireless Mouse", "price": 24.99, "stock": 150}`
product:P002	`{"name": "USB-C Cable", "price": 29.99, "stock": 200}`
product:P003	`{"name": "Laptop Stand", "price": 39.99, "stock": 75}`
product:P004	`{"name": "Mechanical Keyboard", "price": 89.99, "stock": 50}`

Key-Value: Adoption

Widely Used Despite Simplicity:

- BigTable was the **only database at Google** for many years
  - Used by Gmail, Calendar, Maps, and all Google apps
- Cloud storage services are key-value stores:
  - AWS S3
  - Google Cloud Storage (GCS)

Want to Learn More?

The 2006 Bigtable paper is an excellent read

Type 2: Document Databases

Document: Definition

> Similar to a key-value database, **except** the value is a **tagged document**

Key Difference:

- The database **understands tags**
- Queries can **reference tags**
- Document formats: XML, JSON, etc.

Document: Example Data (1/3)

```
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "username": "alice_smith",
  "email": "alice@example.com",
  "firstName": "Alice",
  "lastName": "Smith",
  "age": 28,
  "address": {
    "street": "123 Main St",
    "city": "Seattle",
    "state": "WA",
    "zipCode": "98101",
    "country": "USA"
  },
  "phone": "+1-206-555-0123",
  "accountType": "premium",
  "createdAt": ISODate("2024-01-15T08:30:00Z"),
  "lastLogin": ISODate("2025-10-30T09:45:00Z"),
  "preferences": {
    "notifications": true,
    "newsletter": false,
    "theme": "dark"
  },
  "tags": ["early-adopter", "active-user"]
}
```

Document: Example Data (2/3)

```
{
  "_id": ObjectId("507f1f77bcf86cd799439012"),
  "username": "bob_jones",
  "email": "bob@example.com",
  "firstName": "Bob",
  "lastName": "Jones",
  "age": 35,
  "address": {
    "street": "456 Oak Ave",
    "city": "Portland",
    "state": "OR",
    "zipCode": "97201",
    "country": "USA"
  },
  "phone": "+1-503-555-0456",
  "accountType": "free",
  "createdAt": ISODate("2024-03-22T14:20:00Z"),
  "lastLogin": ISODate("2025-10-29T16:30:00Z"),
  "preferences": {
    "notifications": false,
    "newsletter": true,
    "theme": "light"
  },
  "tags": ["occasional-user"]
}
```

Document: Example Data (3/3)

```
{
  "_id": ObjectId("507f1f77bcf86cd799439013"),
  "username": "carol_white",
  "email": "carol@example.com",
  "firstName": "Carol",
  "lastName": "White",
  "age": 42,
  "address": {
    "street": "789 Pine Rd",
    "city": "San Francisco",
    "state": "CA",
    "zipCode": "94102",
    "country": "USA"
  },
  "accountType": "enterprise",
  "createdAt": ISODate("2023-11-10T10:15:00Z"),
  "lastLogin": ISODate("2025-10-30T08:00:00Z"),
  "preferences": {
    "notifications": true,
    "newsletter": true,
    "theme": "dark"
  },
  "tags": ["power-user", "beta-tester"],
  "company": "Tech Corp Inc"
}
```

Document: MongoDB

MongoDB is a popular open-source document database

Key Resource:

SQL to MongoDB mapping

Comparison to SQL:

- SQL: Fixed schema, rows in tables
- MongoDB: Flexible schema, documents in collections

Type 3: Column-Oriented Databases

Column-Oriented: Definition

> Databases that use **column-centric storage**

Key Innovation:

- Google developed "ColumnIO" format
- Enables fast ad-hoc queries **without** a relational DB
- Basis of Google's BigQuery cloud service

Reference: Dremel paper

Understanding Storage: Sample Data

Employee Table (5 records):

EmployeeID	Name	Department	Salary	Age
101	Alice	Engineering	95000	28
102	Bob	Sales	75000	35
103	Carol	Engineering	105000	42
104	David	Marketing	68000	29
105	Eve	Sales	82000	31

Let's see how different storage approaches handle this data...

Row-Centric Storage (Traditional)

Conceptual Layout: Data stored **row by row**

```
Row 1: 101, Alice, Engineering, 95000, 28
Row 2: 102, Bob, Sales, 75000, 35
Row 3: 103, Carol, Engineering, 105000, 42
Row 4: 104, David, Marketing, 68000, 29
Row 5: 105, Eve, Sales, 82000, 31
```

Each row's **complete data** stored together sequentially

Row-Centric: Physical Disk Layout

```
Disk Blocks:
┌─────────────────────────────────────────────────────────────────┐
│ Block 1                                                         │
│ [101][Alice][Engineering][95000][28]                            │
│ [102][Bob][Sales][75000][35]                                    │
│ [103][Carol][Engineering][105000][42]                           │
├─────────────────────────────────────────────────────────────────┤
│ Block 2                                                         │
│ [104][David][Marketing][68000][29]                              │
│ [105][Eve][Sales][82000][31]                                    │
└─────────────────────────────────────────────────────────────────┘
```

Row-Centric: Query Example

Query: `SELECT Salary FROM Employees WHERE Department = 'Engineering'`

```
Read Path:
Block 1 → [101][Alice][Engineering][95000][28]  ← Read ALL, check dept
          [102][Bob][Sales][75000][35]           ← Read ALL, skip
          [103][Carol][Engineering][105000][42] ← Read ALL, check dept
Block 2 → [104][David][Marketing][68000][29]    ← Read ALL, skip
          [105][Eve][Sales][82000][31]           ← Read ALL, skip
```

Problem: Need to read **ALL columns** even though we only need Salary and Department!

Column-Centric Storage

Conceptual Layout: Data stored **column by column**

```
EmployeeID Column: 101, 102, 103, 104, 105
Name Column: Alice, Bob, Carol, David, Eve
Department Column: Engineering, Sales, Engineering,
                   Marketing, Sales
Salary Column: 95000, 75000, 105000, 68000, 82000
Age Column: 28, 35, 42, 29, 31
```

All values for each column stored together sequentially

Column-Centric: Physical Disk Layout

```
Disk Blocks:
┌────────────────────────────────────────────────┐
│ Block 1: EmployeeID Column                     │
│ [101][102][103][104][105]                      │
├────────────────────────────────────────────────┤
│ Block 2: Name Column                           │
│ [Alice][Bob][Carol][David][Eve]                │
├────────────────────────────────────────────────┤
│ Block 3: Department Column                     │
│ [Engineering][Sales][Engineering][Marketing][Sales]
├────────────────────────────────────────────────┤
│ Block 4: Salary Column                         │
│ [95000][75000][105000][68000][82000]           │
├────────────────────────────────────────────────┤
│ Block 5: Age Column                            │
│ [28][35][42][29][31]                           │
└────────────────────────────────────────────────┘
```

Column-Centric: Query Example

Query: `SELECT Salary FROM Employees WHERE Department = 'Engineering'`

```
Read Path:
Block 3 → [Engineering][Sales][Engineering][Marketing][Sales]
          → position 1  → skip  → position 3  → skip    → skip

Block 4 → [95000][75000][105000][68000][82000]
          → get position 1        → get position 3
```

Advantage: Only read **2 blocks**! (Department and Salary)

Skip EmployeeID, Name, and Age columns entirely

Side-by-Side: Query Comparison

Query: `SELECT AVG(Salary) FROM Employees`

Row-Centric Storage:

```
Must Read ALL columns for ALL rows
[101][Alice][Engineering][95000][28]
[102][Bob][Sales][75000][35]
[103][Carol][Engineering][105000][42]
[104][David][Marketing][68000][29]
[105][Eve][Sales][82000][31]

Bytes Read: ~500 bytes (ALL data)
```

Column-Centric Storage:

```
Must Read ONLY Salary column
[95000][75000][105000][68000][82000]

Bytes Read: ~25 bytes (ONLY Salary)
```

Result: Column-centric reads ~20× less data!

Type 4: Graph Databases

Graph: Definition

> Stores data about **relationship-rich environments** using graph theory

Motivation:

- Social networks have complex relationships
- Relational DBs support relationships... but only if you know them **upfront**
- Graph databases allow **adding new relationships** to existing databases
    - e.g., let customers "friend" one another
    - e.g., let users "like" an agent

Key Insight: The relationships are **as important** as the data itself

Graph: Core Concepts

Graph databases use three building blocks:

1. **Nodes** = Things (entities)
2. **Relationships** = Connections (arrows between nodes)
3. **Properties** = Details (key-value pairs on nodes/relationships)

Graph: Example Network

A Small Social Network:

```
┌──────────────┐
│    Alice     │
│   age: 28    │
└──────────────┘
       │
       │ FRIEND_OF
       │
       ↓
┌──────────────┐           ┌──────────────┐
│     Bob      │──────────→│   TechCorp   │
│   age: 35    │ WORKS_AT  │founded:2010  │
└──────────────┘           └──────────────┘
```

Three Nodes: Alice (Person), Bob (Person), TechCorp (Company)

Two Relationships: Alice → Bob (FRIEND_OF), Bob → TechCorp (WORKS_AT)

Graph: Node Structure

A node represents an entity:

```
Node: Bob
---
Label: Person

Properties:
  name: "Bob"
  age: 35
  email: "bob@ex.com"
```

Graph: Relationship Structure

A relationship connects two nodes:

```
(Alice) ─────────────────→ (Bob)
        FRIEND_OF
        since: 2020
```

More Detailed View:

```
┌─────────────────────────────────┐
│   Alice ──→ Bob                 │
├─────────────────────────────────┤
│ Type: FRIEND_OF                 │
│                                 │
│ Properties:                     │
│   since: 2020                   │
│   status: "close"               │
└─────────────────────────────────┘
```

Graph: Querying

Queries = Graph Traversals

Examples:

- Find shortest path between two people
- Calculate degree of connectedness
- "Friends of friends who work at Company X"
- "Who influenced this purchase decision?"

Popular Choice:

- **Neo4j** is a popular open-source graph database
- Short tutorial video

NoSQL: Summary

Four Main Types:

Type	Example	Best For
Key-Value	BigTable, S3	Simple lookups, caching, sessions
Document	MongoDB	Semi-structured data, flexible schemas
Column-Oriented	Cassandra, BigQuery	Analytics, aggregations, OLAP
Graph	Neo4j	Relationships, social networks, recommendations

NoSQL: Key Takeaways

1. **NoSQL emerged** to handle Big Data challenges that relational DBs couldn't solve efficiently
2. **Different types** for different use cases—no one-size-fits-all solution
3. **Trade-offs**: Simplicity and scalability vs. ACID guarantees and complex queries
4. **Not a replacement** for relational DBs, but a complementary tool
5. **Widely adopted** by Big Tech and now mainstream