Chapter 8: Storage Systems

Introduction: Why Storage Matters

Different types of data need different storage solutions.

```
The Storage Hierarchy:

┌─────────────────────────────────────────────────┐
│           Your Application Needs                 │
├─────────────────────────────────────────────────┤
│ Store user photos        → Object Storage        │
│ Host operating system    → Block Storage         │
│ Share files across servers → File Storage        │
│ Store structured data    → Database              │
│ Big data processing      → Distributed FS        │
└─────────────────────────────────────────────────┘

One size does NOT fit all!
```

---

1. Block Storage vs Object Storage vs File Storage

Block Storage

Concept: Raw storage volumes, like a hard drive. Data stored in fixed-size blocks.

```
Block Storage Structure:

Physical Storage:
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│Block│Block│Block│Block│Block│Block│Block│Block│
│  1  │  2  │  3  │  4  │  5  │  6  │  7  │  8  │
│ 4KB │ 4KB │ 4KB │ 4KB │ 4KB │ 4KB │ 4KB │ 4KB │
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘

File System Layer (on top of blocks):
- Organizes blocks into files and directories
- Maintains metadata (permissions, timestamps)
- Provides file operations (open, read, write, close)

To OS/Application:
Appears as: /dev/sda1 or C:\ drive
Can format with: ext4, NTFS, XFS, etc.
```

How It Works:

```
Writing a file:
1. OS requests: "Write file data.txt (10 KB)"
2. File system allocates blocks: Blocks 1, 2, 3
3. Writes data:
   Block 1: Bytes 0-4095
   Block 2: Bytes 4096-8191
   Block 3: Bytes 8192-10239
4. Updates metadata: data.txt → Blocks 1,2,3

Reading a file:
1. OS requests: "Read data.txt"
2. File system looks up: data.txt → Blocks 1,2,3
3. Reads blocks in sequence
4. Returns combined data to application
```

Use Cases:

```
✓ Operating system boot drives
✓ Database storage (MySQL, PostgreSQL)
✓ Virtual machine disks
✓ High-performance applications (low latency)
✓ Transactional workloads

Characteristics:
- Low latency (direct attached or over network)
- High IOPS (I/O operations per second)
- Can be formatted with file system
- Typically mounted to one server at a time
```

Examples:

```
AWS EBS (Elastic Block Store):
- Volumes attach to EC2 instances
- Types: gp3 (general), io2 (high IOPS), st1 (throughput)
- Snapshots for backup

Google Persistent Disk:
- Attached to Compute Engine VMs
- Can snapshot and clone

Azure Managed Disks:
- Premium SSD, Standard SSD, Standard HDD
- Attach to Azure VMs
```

Code Example (AWS EBS):

```python
```

```python
import boto3

ec2 = boto3.client('ec2')

# Create a block storage volume (EBS)
response = ec2.create_volume(
    AvailabilityZone='us-east-1a',
    Size=100, # 100 GB
    VolumeType='gp3', # General purpose SSD
    Iops=3000, # I/O operations per second
    Throughput=125, # MB/s
    TagSpecifications=[{
        'ResourceType': 'volume',
        'Tags': [{'Key': 'Name', 'Value': 'database-volume'}]
    }]
)

volume_id = response['VolumeId']
print(f"Created volume: {volume_id}")

# Attach to EC2 instance
ec2.attach_volume(
    VolumeId=volume_id,
    InstanceId='i-1234567890abcdef',
    Device='/dev/sdf'
)

# On the EC2 instance, you would then:
# 1. Format: mkfs.ext4 /dev/sdf
# 2. Mount: mount /dev/sdf /data
# 3. Use: Can now write files to /data
```

File Storage (Network File System)

Concept: Shared file system accessible over network. Multiple servers can access simultaneously.

```
File Storage Architecture:

┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│  Server 1    │  │  Server 2    │  │  Server 3    │
│              │  │              │  │              │
│  Mounts:     │  │  Mounts:     │  │  Mounts:     │
│  /shared     │  │  /shared     │  │  /shared     │
└──────┬───────┘  └──────┬───────┘  └──────┬───────┘
       │                 │                 │
       └─────────────────┼─────────────────┘
                         │ Network (NFS/SMB)
                         ↓
              ┌─────────────────────────┐
              │  File Storage Server    │
              │  (NAS or Cloud)         │
              │                         │
              │  /shared/               │
              │  ├── folder1/           │
              │  │   ├── file1.txt      │
              │  │   └── file2.txt      │
              │  └── folder2/           │
              │      └── file3.txt      │
              └─────────────────────────┘

All servers see the same files!
Changes are immediately visible to all.
```

Protocols:

```
NFS (Network File System) - Linux/Unix:
- Mounts remote directory as local
- POSIX compliant (standard file operations)
- Popular in enterprise environments

SMB/CIFS (Server Message Block) - Windows:
- Windows file sharing protocol
- Also works on Linux (via Samba)
- Used in Windows networks

Examples:
- AWS EFS (Elastic File System) - NFS
- Azure Files - SMB/NFS
- Google Filestore - NFS
```

Use Cases:

✓ Shared application data (logs, uploads)
✓ Web server content (HTML, images)
✓ Development environments (shared code)
✓ Content management systems
✓ Home directories for users
✓ Media files shared across servers

Characteristics:
- Hierarchical (folders and files)
- POSIX operations (open, read, write, close)
- Shared access (multiple servers simultaneously)
- Higher latency than block storage
- Good for sequential access

Code Example (AWS EFS):

```python
```

```python
import boto3

efs = boto3.client('efs')

# Create file system
response = efs.create_file_system(
    PerformanceMode='generalPurpose',  # or maxIO
    ThroughputMode='bursting',  # or provisioned
    Encrypted=True,
    Tags=[{'Key': 'Name', 'Value': 'shared-storage'}]
)

file_system_id = response['FileSystemId']
print(f"Created EFS: {file_system_id}")

# Create mount target (in each availability zone)
efs.create_mount_target(
    FileSystemId=file_system_id,
    SubnetId='subnet-12345',
    SecurityGroups=['sg-12345']
)

# On EC2 instances (all of them can mount this):
# sudo mount -t efs fs-12345:/ /mnt/efs
# Now all instances share /mnt/efs directory!

# Application code - just use normal file operations
with open('/mnt/efs/shared-data.txt', 'w') as f:
    f.write('This file is shared across all servers!')

# Other servers can immediately read this file
with open('/mnt/efs/shared-data.txt', 'r') as f:
    print(f.read())
```

Object Storage

Concept: Store data as objects (blobs) with metadata. Accessed via HTTP API, not file system.

```
Object Storage Structure:

Each object consists of:
┌─────────────────────────────────────┐
│         OBJECT                      │
├─────────────────────────────────────┤
│ Key (ID):                           │
│   photos/2024/user123/photo.jpg     │
├─────────────────────────────────────┤
│ Data (Binary):                      │
│   [JPEG image bytes...]             │
├─────────────────────────────────────┤
│ Metadata:                           │
│   Content-Type: image/jpeg          │
│   Content-Length: 2048576           │
│   X-User-ID: 123                    │
│   X-Upload-Date: 2024-01-15         │
│   Custom-Field: value               │
├─────────────────────────────────────┤
│ Version ID: v1234567                │
│ (if versioning enabled)             │
└─────────────────────────────────────┘

Bucket (Container):
┌──────────────────────────────────┐
│  Bucket: my-photos               │
├──────────────────────────────────┤
│ photos/2024/user123/photo1.jpg   │
│ photos/2024/user123/photo2.jpg   │
│ photos/2024/user456/photo1.jpg   │
│ documents/report.pdf             │
│ videos/intro.mp4                 │
└──────────────────────────────────┘

Note: "photos/2024/" looks like folders,
but it's actually part of the object key!
No true hierarchy - just naming convention.
```

Key Characteristics:

1. FLAT NAMESPACE
   - No true directories (just key names)
   - photos/2024/image.jpg is one key

2. HTTP API ACCESS
   - GET /bucket/key (retrieve)
   - PUT /bucket/key (upload)
   - DELETE /bucket/key (remove)
   - Not mounted like file system

3. METADATA
   - Key-value pairs attached to objects
   - Can query and filter by metadata

4. IMMUTABLE
   - Can't modify part of object
   - Must upload entire object again
   - Versioning available

5. MASSIVE SCALE
   - Store trillions of objects
   - Petabytes or exabytes of data
   - Distributed across many servers

6. EVENTUAL CONSISTENCY (sometimes)
   - Write might not be immediately visible
   - Eventually all replicas sync

Use Cases:

✓ User-uploaded content (photos, videos)
✓ Static website hosting
✓ Backup and archival
✓ Big data analytics (data lake)
✓ Media streaming (CDN origin)
✓ Application assets (CSS, JS, images)
✓ Log aggregation
✓ Machine learning datasets

Characteristics:
- Very cheap (cents per GB)
- Highly durable (11 nines: 99.999999999%)
- Infinitely scalable
- High latency (vs block storage)
- No random writes (object is atomic)

Code Example (AWS S3):

```python
```

```python
import boto3
import json

s3 = boto3.client('s3')

# 1. Create bucket
bucket_name = 'my-app-storage'
s3.create_bucket(Bucket=bucket_name)

# 2. Upload object
with open('photo.jpg', 'rb') as f:
    s3.put_object(
        Bucket=bucket_name,
        Key='photos/2024/user123/photo.jpg',
        Body=f,
        ContentType='image/jpeg',
        Metadata={
            'user-id': '123',
            'upload-date': '2024-01-15',
            'camera': 'iPhone 13'
        }
    )

# 3. Retrieve object
response = s3.get_object(
    Bucket=bucket_name,
    Key='photos/2024/user123/photo.jpg'
)

# Get object data
data = response['Body'].read()

# Get metadata
metadata = response['Metadata']
content_type = response['ContentType']
print(f"User ID: {metadata['user-id']}")

# 4. List objects (with prefix - like "folder")
response = s3.list_objects_v2(
    Bucket=bucket_name,
    Prefix='photos/2024/user123/'
)

for obj in response.get('Contents', []):
    print(f"Found: {obj['Key']} ({obj['Size']} bytes)")
```

```python
# 5. Delete object
s3.delete_object(
    Bucket=bucket_name,
    Key='photos/2024/user123/photo.jpg'
)

# 6. Generate pre-signed URL (temporary access)
url = s3.generate_presigned_url(
    'get_object',
    Params={'Bucket': bucket_name, 'Key': 'photos/2024/user123/photo.jpg'},
    ExpiresIn=3600  # Valid for 1 hour
)
print(f"Temporary URL: {url}")
# User can download directly from this URL without AWS credentials

# 7. Copy object (server-side, no download/upload)
s3.copy_object(
    Bucket=bucket_name,
    CopySource={'Bucket': bucket_name, 'Key': 'photo.jpg'},
    Key='photos/backup/photo.jpg'
)

# 8. Set object ACL (permissions)
s3.put_object_acl(
    Bucket=bucket_name,
    Key='photo.jpg',
    ACL='public-read'  # Anyone can read
)

# 9. Enable versioning (keep old versions)
s3.put_bucket_versioning(
    Bucket=bucket_name,
    VersioningConfiguration={'Status': 'Enabled'}
)

# Now every upload creates new version, old ones preserved!
```

Advanced S3 Features:

```python
```

```python
# Lifecycle policies - automatic archival/deletion
lifecycle_config = {
    'Rules': [
        {
            'Id': 'archive-old-logs',
            'Status': 'Enabled',
            'Filter': {'Prefix': 'logs/'},
            'Transitions': [
                {
                    'Days': 30,
                    'StorageClass': 'GLACIER'  # Move to cold storage
                }
            ],
            'Expiration': {'Days': 365}  # Delete after 1 year
        }
    ]
}

s3.put_bucket_lifecycle_configuration(
    Bucket=bucket_name,
    LifecycleConfiguration=lifecycle_config
)

# Event notifications - trigger Lambda on upload
notification_config = {
    'LambdaFunctionConfigurations': [
        {
            'LambdaFunctionArn': 'arn:aws:lambda:...',
            'Events': ['s3:ObjectCreated:*'],
            'Filter': {
                'Key': {
                    'FilterRules': [
                        {'Name': 'suffix', 'Value': '.jpg'}
                    ]
                }
            }
        }
    ]
}

s3.put_bucket_notification_configuration(
    Bucket=bucket_name,
    NotificationConfiguration=notification_config
)
# Now Lambda function runs automatically when .jpg uploaded!
```

```
# Cross-region replication - redundancy
replication_config = {
    'Role': 'arn:aws:iam::...',
    'Rules': [
        {
            'Status': 'Enabled',
            'Priority': 1,
            'Filter': {},
            'Destination': {
                'Bucket': 'arn:aws:s3:::backup-bucket',
                'StorageClass': 'STANDARD_IA'
            }
        }
    ]
}

s3.put_bucket_replication(
    Bucket=bucket_name,
    ReplicationConfiguration=replication_config
)
```

Comparison Table

Feature	Block	File	Object
Access	Block-level	File-level	HTTP API
Protocol	iSCSI, FC	NFS, SMB	REST/HTTPS
Mount	Yes (1 server	Yes (many)	No
Structure	Unformatted	Hierarchical	Flat
Modify	Random	Random	Full object
Metadata	Limited	File attrs	Rich
Performance			
Latency	<1 ms	1-10 ms	10-100 ms
IOPS	High (10K+)	Medium (1K)	Low (100)
Throughput	High	High	Very High
Scale	TB	PB	EB
Cost	$$$$	$$$	$
Durability	Single copy	RAID/backup	11 nines
Share	No (usually)	Yes	Yes (URL)
Use Cases	Databases	Shared files	Media
	VMs	Web content	Backups

```
│          │ Boot drives │ Home dirs   │ Archives    │
│          │ Apps        │ CMS         │ Big data    │
└──────────┴─────────────┴─────────────┴─────────────┘
```

Real-World Architecture Example:

```
E-commerce Application:

┌─────────────────────────────────────────────┐
│         Application Architecture            │
├─────────────────────────────────────────────┤
│                                             │
│  ┌─────────────────────────┐                │
│  │ Web Servers │ (EC2)     │                │
│  └───────────┬─────────────┘                │
│       │                         │           │
│   ┌───┴────┬──────────┬──────────────┐      │
│   ↓      ↓        ↓          ↓         │
│ ┌──────┐ ┌──────┐ ┌─────────┐ ┌─────────┐  │
│ │Block │ │ File │ │ Object  │ │Database │  │
│ │EBS   │ │EFS   │ │S3       │ │RDS      │  │
│ └──────┘ └──────┘ └─────────┘ └─────────┘  │
│   │      │        │          │         │
│   ↓      ↓        ↓          ↓         │
│ Boot   Shared   Product    Order       │
│ Drive  Logs     Images     Data        │
│ OS     Session  Videos     Inventory   │
│        Uploads  Static     Users       │
│                 Assets                 │
└─────────────────────────────────────────────┘

Each storage type serves specific purpose!
```

---

2. Distributed File Systems (HDFS, GFS)

Why Distributed File Systems?

Problem: Single file server can't handle massive data.

```
Traditional File Server:
┌─────────────────────────┐
│   File Server           │
│   - 10 TB storage       │
│   - 1 Gbps bandwidth    │
│   - Single point fail   │
└─────────────────────────┘

Problem with 1 PB dataset:
- Need 100 servers
- Manage 100 separate systems
- No unified view
- Complex coordination

Distributed File System:
┌─────────────────────────────────────┐
│    Unified File System              │
│    (Appears as one drive)           │
└─────────────────────────────────────┘
              │
   ┌──────────┼──────────┬──────────┐
   ↓          ↓          ↓          ↓
┌────────┐ ┌────────┐ ┌────────┐ ┌────────┐
│Node 1  │ │Node 2  │ │Node 3  │ │Node 4  │
│100 TB  │ │100 TB  │ │100 TB  │ │100 TB  │
└────────┘ └────────┘ └────────┘ └────────┘

Benefits:
- Single namespace
- Automatic replication
- Load balancing
- Fault tolerance
- Scalable to EB
```

HDFS (Hadoop Distributed File System)

Architecture:

```
HDFS Architecture:

┌─────────────────────────────────────────┐
│           NameNode (Master)             │
│  - Stores metadata (file → blocks mapping)  │
│  - Manages namespace (directory structure)  │
│  - Coordinates DataNode operations      │
└─────────────────────────────────────────┘
         │
    ┌────────┼────────┬────────┐
    ↓        ↓        ↓        ↓
┌────────┐ ┌────────┐ ┌────────┐ ┌────────┐
│DataNode│ │DataNode│ │DataNode│ │DataNode│
│   1    │ │   2    │ │   3    │ │   4    │
│        │ │        │ │        │ │        │
│ Stores │ │ Stores │ │ Stores │ │ Stores │
│ Blocks │ │ Blocks │ │ Blocks │ │ Blocks │
└────────┘ └────────┘ └────────┘ └────────┘

Block Size: 128 MB (default)
Replication Factor: 3 (default)
```

How File Storage Works:

```
Storing a 1 GB file:

1. File Split into Blocks:
   file.dat (1 GB) → 8 blocks of 128 MB each

2. NameNode Decides Placement:
   Block 1 → DataNodes 1, 2, 3 (3 replicas)
   Block 2 → DataNodes 2, 3, 4
   Block 3 → DataNodes 3, 4, 1
   Block 4 → DataNodes 4, 1, 2
   Block 5 → DataNodes 1, 2, 3
   Block 6 → DataNodes 2, 3, 4
   Block 7 → DataNodes 3, 4, 1
   Block 8 → DataNodes 4, 1, 2

3. Client Writes Blocks:
   Client → DataNode 1 (Block 1)
     DataNode 1 → DataNode 2 (replica)
       DataNode 2 → DataNode 3 (replica)
         All 3 acknowledge

4. NameNode Updates Metadata:
   /user/data/file.dat → [Block IDs and locations]

Visual:
┌─────────────────────────────────────┐
│ NameNode Metadata                   │
├─────────────────────────────────────┤
│ /user/data/file.dat:                │
│   Block 1: [DN1, DN2, DN3]          │
│   Block 2: [DN2, DN3, DN4]          │
│   Block 3: [DN3, DN4, DN1]          │
│   ...                               │
└─────────────────────────────────────┘
```

Reading a File:

```
1. Client Asks NameNode:
   "Where is /user/data/file.dat?"

2. NameNode Returns Block Locations:
   Block 1: DataNodes 1, 2, 3
   Block 2: DataNodes 2, 3, 4
   ...

3. Client Reads from Closest DataNode:
   Block 1: Read from DataNode 1 (closest)
   Block 2: Read from DataNode 2
   ...

4. Client Assembles Blocks:
   Block 1 + Block 2 + ... → Complete file

Benefits:
- Parallel reads (read multiple blocks simultaneously)
- Fault tolerance (if DN1 fails, read from DN2)
- Load balancing (pick least loaded DataNode)
```

Fault Tolerance:

```
Scenario: DataNode 2 Fails

Before:
Block 1: [DN1, DN2✓, DN3]
Block 2: [DN2✓, DN3, DN4]
Block 3: [DN3, DN4, DN1]

After Failure:
Block 1: [DN1, DN2✗, DN3]  (2 replicas left)
Block 2: [DN2✗, DN3, DN4]  (2 replicas left)

NameNode Detects:
- DN2 missed 3 heartbeats (30 seconds)
- Marks DN2 as dead
- Replication count below target (3)

NameNode Triggers Re-replication:
Block 1: DN1 → copy to DN4 → [DN1, DN3, DN4] ✓
Block 2: DN3 → copy to DN1 → [DN1, DN3, DN4] ✓

System self-heals automatically!
```

Python Example (using hdfs library):

```python
from hdfs import InsecureClient

# Connect to NameNode
client = InsecureClient('http://namenode:50070', user='hadoop')

# Upload file (automatically splits into blocks)
with open('large_file.dat', 'rb') as f:
    client.write('/user/data/large_file.dat', f, overwrite=True)

# List files
files = client.list('/user/data/')
for file in files:
    status = client.status(f'/user/data/{file}')
    print(f"{file}: {status['length']} bytes, "
          f"replication={status['replication']}, "
          f"blockSize={status['blockSize']}")

# Read file (automatically reads all blocks)
with client.read('/user/data/large_file.dat') as reader:
    content = reader.read()
    print(f"Read {len(content)} bytes")

# Delete file
client.delete('/user/data/large_file.dat')

# Get file block locations (for optimization)
status = client.content('/user/data/file.dat')
for block in status:
    print(f"Block {block['blockId']}:")
    print(f"  Locations: {block['locations']}")
```

HDFS Characteristics:

✓ ADVANTAGES:
- Massive scale (petabytes)
- Fault tolerant (automatic replication)
- High throughput (parallel I/O)
- Cost effective (commodity hardware)
- Write once, read many (optimized for this)

✗ DISADVANTAGES:
- High latency (not for low-latency apps)
- NameNode single point of failure (can be mitigated)
- No random writes (append-only)
- Not POSIX compliant (different semantics)
- Overkill for small data

When to use:
- Big data processing (MapReduce, Spark)
- Data lake storage
- Log aggregation
- Batch analytics
- Machine learning training data

Google File System (GFS)

Similar to HDFS but with some differences:

```
GFS Architecture:

┌─────────────────────────────────────┐
│   Master (like HDFS NameNode)   │
│   - Single master               │
│   - Stores metadata in memory   │
│   - Periodic checkpoints        │
└───────────────┬─────────────────────┘
        │
  ┌─────────┼─────────┬─────────┐
  ↓     ↓     ↓      ↓
┌─────────┐ ┌─────────┐ ┌─────────┐ ┌─────────┐
│Chunk    │ │Chunk    │ │Chunk    │ │Chunk    │
│Server 1 │ │Server 2 │ │Server 3 │ │Server 4 │
└─────────┘ └─────────┘ └─────────┘ └─────────┘

Differences from HDFS:
- Single master (no secondary)
- Chunk size: 64 MB (vs 128 MB in HDFS)
- Record append supported (concurrent appends)
- Designed for Google's workloads
```

Key Innovations:

```
1. Snapshot:
   - Create instant copy of file/directory
   - Copy-on-write (changes create new chunks)
   - Useful for backups, experimentation

2. Record Append:
   - Multiple clients can append concurrently
   - Atomic append operation
   - Great for log files

3. Lease Mechanism:
   - Master grants lease to primary replica
   - Primary coordinates writes
   - Reduces master load
```

---

3. Blob Storage (Deep Dive on S3)

Amazon S3 Architecture

```
S3 Internal Architecture (Simplified):
```

```
Client Request:
  PUT /bucket/photos/image.jpg
        ↓
┌──────────────────────────────────┐
│  API Frontend          │
│  (Load Balanced)       │
└──────────────┬───────────────────┘
        │
        ↓
┌──────────────────────────────────┐
│  Index Subsystem       │
│  (Metadata & Mapping)  │
│  - Object → Storage    │
│  - Permissions         │
│  - Versioning          │
└──────────────┬───────────────────┘
        │
        ↓
┌──────────────────────────────────┐
│  Storage Subsystem     │
│  (Distributed Storage) │
│                        │
│  ┌───────┐ ┌───────┐ ┌───────┐
│  │Disk │ │Disk │ │Disk │
│  │ 1 │ │ 2 │ │ 3 │
│  └───────┘ └───────┘ └───────┘
│ (Replicated across     │
│  multiple facilities)  │
└──────────────────────────────────┘

Data is:
- Encrypted at rest
- Replicated across availability zones
- Checksummed for integrity
```

S3 Storage Classes

```
┌─────────────────────────┬─────────────────┬───────────────────┬─────────────────┐
│ Storage Class    │ Access Time │ Cost/GB/mo  │ Use Case   │
├─────────────────────────┼─────────────────┼───────────────────┼─────────────────┤
│ S3 Standard    │ ms       │ $0.023     │ Frequent  │
│          │       │        │ access    │
├─────────────────────────┼─────────────────┼───────────────────┼─────────────────┤
│ S3 Intelligent- │ ms      │ $0.023     │ Auto-tier    │
│ Tiering        │       │ (monitored)  │ unknown    │
```

S3 Standard-IA (Infrequent)	ms	$0.0125	Infrequent monthly
S3 One Zone-IA	ms	$0.01	Infrequent 1 AZ ok
S3 Glacier Instant Retrieval	minutes-hours	$0.004	Archive quarterly
S3 Glacier Flexible	hours	$0.0036	Archive yearly
S3 Glacier Deep Archive	12 hours	$0.00099	Long-term 7-10 years

Cost Example:
100 TB in S3 Standard: $2,300/month
100 TB in Glacier: $400/month
100 TB in Deep Archive: $99/month

Savings: 95% by choosing right tier!

Lifecycle Policy Example:

python

```
lifecycle_rules = {
    'Rules': [
        {
            'Id': 'photos-lifecycle',
            'Status': 'Enabled',
            'Filter': {'Prefix': 'photos/'},
            'Transitions': [
                {
                    'Days': 30,
                    'StorageClass': 'INTELLIGENT_TIERING'
                },
                {
                    'Days': 90,
                    'StorageClass': 'GLACIER_IR'
                },
                {
                    'Days': 365,
                    'StorageClass': 'DEEP_ARCHIVE'
                }
            ],
            'Expiration': {
                'Days': 2555  # Delete after 7 years
            }
        },
        {
            'Id': 'logs-lifecycle',
            'Status': 'Enabled',
            'Filter': {'Prefix': 'logs/'},
            'Expiration': {
                'Days': 90  # Delete logs after 90 days
            }
        }
    ]
}

s3.put_bucket_lifecycle_configuration(
    Bucket='my-bucket',
    LifecycleConfiguration=lifecycle_rules
)

# Automatic cost optimization!
# Day 0: Photo in Standard ($0.023/GB)
# Day 30: Auto-moved to Intelligent Tiering
# Day 90: Auto-moved to Glacier ($0.004/GB)
```

```
# Day 365: Auto-moved to Deep Archive ($0.00099/GB)
# Day 2555: Auto-deleted
```

S3 Performance Optimization

Parallel Uploads (Multipart):

```python
import boto3
from boto3.s3.transfer import TransferConfig

# For files > 100 MB, use multipart upload
MB = 1024 * 1024
config = TransferConfig(
    multipart_threshold=100 * MB, # Use multipart for files > 100 MB
    multipart_chunksize=50 * MB,  # Upload in 50 MB chunks
    max_concurrency=10,           # 10 parallel threads
    use_threads=True
)

s3 = boto3.client('s3')

# Upload large file with optimal settings
s3.upload_file(
    'large_video.mp4',
    'my-bucket',
    'videos/large_video.mp4',
    Config=config
)

# How it works:
# 1. File split into 50 MB chunks
# 2. 10 chunks uploaded in parallel
# 3. S3 assembles chunks
# 4. Result: 10x faster for large files!
```

Transfer Acceleration:

```python
```

```python
# Enable transfer acceleration (uses CloudFront edge locations)
s3.put_bucket_accelerate_configuration(
    Bucket='my-bucket',
    AccelerateConfiguration={'Status': 'Enabled'}
)

# Use accelerated endpoint
s3_accelerated = boto3.client(
    's3',
    endpoint_url='https://my-bucket.s3-accelerate.amazonaws.com'
)

# Uploads route through nearest edge location
# Can be 50-500% faster for international uploads!
```

Request Rate Optimization:

```
S3 Performance Limits (per prefix):
- 3,500 PUT/COPY/POST/DELETE per second
- 5,500 GET/HEAD per second

Bad Design (one prefix):
/photos/image1.jpg
/photos/image2.jpg
...
All 1 million images in /photos/
→ Limited to 5,500 reads/sec

Good Design (many prefixes):
/photos/2024/01/15/image1.jpg
/photos/2024/01/16/image2.jpg
...
Each date is a prefix!
365 prefixes × 5,500 = 2 million reads/sec!

Python implementation:
from datetime import datetime

def get_s3_key(image_id):
    """Spread images across prefixes by date"""
    date = datetime.now()
    prefix = date.strftime('photos/%Y/%m/%d')
    return f"{prefix}/{image_id}.jpg"

# Now can handle 2M+ requests per second!
```

4. Database Types (Deep Dive)

We covered basics in Chapter 6. Here's deeper analysis.

Relational Databases (SQL)

When to Use:

```
✓ Complex queries with JOINs
✓ ACID transactions required
✓ Data has clear schema
✓ Strong consistency needed
✓ Referential integrity important

Examples:
- Banking (transactions, accounts)
- E-commerce orders
- Inventory management
- HR systems
```

Advanced Features:

```sql
```

```sql
-- Stored Procedures (reusable logic)
CREATE PROCEDURE transfer_money(
    from_account INT,
    to_account INT,
    amount DECIMAL
)
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        RESIGNAL;
    END;

    START TRANSACTION;

    UPDATE accounts
    SET balance = balance - amount
    WHERE id = from_account AND balance >= amount;

    IF ROW_COUNT() = 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Insufficient funds';
    END IF;

    UPDATE accounts
    SET balance = balance + amount
    WHERE id = to_account;

    COMMIT;
END;

-- Call: CALL transfer_money(123, 456, 100.00);

-- Triggers (automatic actions)
CREATE TRIGGER audit_balance_changes
AFTER UPDATE ON accounts
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (account_id, old_balance, new_balance, timestamp)
    VALUES (NEW.id, OLD.balance, NEW.balance, NOW());
END;

-- Views (virtual tables)
CREATE VIEW high_value_customers AS
SELECT u.id, u.name, SUM(o.total) as lifetime_value
FROM users u
JOIN orders o ON u.id = o.user_id
```

```sql
GROUP BY u.id, u.name
HAVING SUM(o.total) > 10000;

-- Query like a table: SELECT * FROM high_value_customers;

-- Materialized Views (cached query results)
CREATE MATERIALIZED VIEW daily_sales_summary AS
SELECT
  DATE(order_date) as date,
  COUNT(*) as order_count,
  SUM(total) as total_sales
FROM orders
GROUP BY DATE(order_date);

-- Refresh periodically
REFRESH MATERIALIZED VIEW daily_sales_summary;
```

Document Databases (MongoDB, CouchDB)

Data Model:

```json
```

```
// User document with embedded data
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "username": "john_doe",
  "email": "john@example.com",
  "profile": {
    "firstName": "John",
    "lastName": "Doe",
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "New York",
      "zip": "10001"
    }
  },
  "preferences": {
    "newsletter": true,
    "theme": "dark"
  },
  "posts": [
    {
      "postId": ObjectId("..."),
      "title": "My First Post",
      "content": "...",
      "tags": ["tech", "programming"],
      "likes": 42,
      "createdAt": ISODate("2024-01-15T10:00:00Z")
    }
  ],
  "followers": [
    ObjectId("507f1f77bcf86cd799439012"),
    ObjectId("507f1f77bcf86cd799439013")
  ],
  "stats": {
    "postCount": 150,
    "followerCount": 1234,
    "totalLikes": 5678
  },
  "createdAt": ISODate("2020-01-01T00:00:00Z"),
  "updatedAt": ISODate("2024-01-15T10:00:00Z")
}
```

Advanced Queries:

```javascript
```

```
const { MongoClient } = require('mongodb');

const client = new MongoClient('mongodb://localhost:27017');
const db = client.db('social_network');
const users = db.collection('users');

// 1. Find users by nested field
const nycUsers = await users.find({
  'profile.address.city': 'New York'
}).toArray();

// 2. Array operations
// Find users who have posts tagged with "tech"
const techUsers = await users.find({
  'posts.tags': 'tech'
}).toArray();

// 3. Aggregation pipeline (powerful!)
const topPosters = await users.aggregate([
  // Stage 1: Unwind posts array
  { $unwind: '$posts' },

  // Stage 2: Group by user
  {
    $group: {
      _id: '$_id',
      username: { $first: '$username' },
      postCount: { $sum: 1 },
      totalLikes: { $sum: '$posts.likes' }
    }
  },

  // Stage 3: Sort by total likes
  { $sort: { totalLikes: -1 } },

  // Stage 4: Limit to top 10
  { $limit: 10 }
]).toArray();

// 4. Text search
users.createIndex({ 'posts.content': 'text' });
const searchResults = await users.find({
  $text: { $search: 'mongodb database' }
}).toArray();

// 5. Geospatial query
```

```
users.createIndex({ 'profile.location': '2dsphere' });
const nearbyUsers = await users.find({
  'profile.location': {
    $near: {
      $geometry: {
        type: 'Point',
        coordinates: [-73.9857, 40.7484]  // NYC coordinates
      },
      $maxDistance: 5000  // 5 km
    }
  }
}).toArray();

// 6. Update nested document
await users.updateOne(
  { _id: ObjectId('...') },
  {
    $set: { 'profile.age': 31 },
    $inc: { 'stats.postCount': 1 },
    $push: {
      posts: {
        postId: ObjectId(),
        title: 'New Post',
        content: '...',
        createdAt: new Date()
      }
    }
  }
);

// 7. Transactions (MongoDB 4.0+)
const session = client.startSession();
session.startTransaction();

try {
  await users.updateOne(
    { _id: user1Id },
    { $inc: { balance: -100 } },
    { session }
  );

  await users.updateOne(
    { _id: user2Id },
    { $inc: { balance: 100 } },
    { session }
  );
```

```
    await session.commitTransaction();
  } catch (error) {
    await session.abortTransaction();
    throw error;
  } finally {
    session.endSession();
  }
```

When to Use Document DB:

```
✓ Flexible schema (fields vary by document)
✓ Hierarchical data (nested objects)
✓ Rapid development (schema changes frequently)
✓ Denormalized data preferred
✓ Document-oriented data model

Examples:
- Content management systems
- User profiles
- Product catalogs
- Real-time analytics
- Mobile app backends
```

Key-Value Databases (Redis, DynamoDB)

Redis Use Cases:

```python
```

```python
import redis

r = redis.Redis(host='localhost', port=6379, decode_responses=True)

# 1. Caching (most common)
def get_user_cached(user_id):
    # Check cache first
    cached = r.get(f'user:{user_id}')
    if cached:
        return json.loads(cached)

    # Cache miss - fetch from database
    user = db.query('SELECT * FROM users WHERE id = %s', user_id)

    # Store in cache (1 hour expiry)
    r.setex(f'user:{user_id}', 3600, json.dumps(user))

    return user

# 2. Session storage
def create_session(user_id):
    session_id = str(uuid.uuid4())
    session_data = {
        'user_id': user_id,
        'created_at': time.time()
    }

    # Store session (24 hour expiry)
    r.setex(f'session:{session_id}', 86400, json.dumps(session_data))

    return session_id

# 3. Rate limiting
def check_rate_limit(user_id, max_requests=100, window=3600):
    """Allow 100 requests per hour"""
    key = f'rate_limit:{user_id}'

    # Increment counter
    current = r.incr(key)

    # Set expiry on first request
    if current == 1:
        r.expire(key, window)

    # Check if exceeded
    if current > max_requests:
```

```python
        return False  # Rate limit exceeded

    return True  # Request allowed

# 4. Leaderboard (sorted set)
def update_leaderboard(player_id, score):
    r.zadd('game:leaderboard', {player_id: score})

def get_top_players(n=10):
    # Get top N players with scores
    return r.zrevrange('game:leaderboard', 0, n-1, withscores=True)

def get_player_rank(player_id):
    # Get player's rank (0-based)
    return r.zrevrank('game:leaderboard', player_id)

# 5. Pub/Sub (messaging)
def publish_notification(user_id, message):
    channel = f'notifications:{user_id}'
    r.publish(channel, json.dumps(message))

def subscribe_notifications(user_id):
    pubsub = r.pubsub()
    channel = f'notifications:{user_id}'
    pubsub.subscribe(channel)

    for message in pubsub.listen():
        if message['type'] == 'message':
            data = json.loads(message['data'])
            print(f"Notification: {data}")

# 6. Distributed lock
def acquire_lock(resource_id, timeout=10):
    """Acquire exclusive lock on resource"""
    lock_key = f'lock:{resource_id}'
    lock_id = str(uuid.uuid4())

    # Try to acquire lock
    acquired = r.set(lock_key, lock_id, nx=True, ex=timeout)

    return lock_id if acquired else None

def release_lock(resource_id, lock_id):
    """Release lock (only if we own it)"""
    lock_key = f'lock:{resource_id}'

    # Lua script for atomic check-and-delete
```

```
    lua_script = """
    if redis.call("get", KEYS[1]) == ARGV[1] then
        return redis.call("del", KEYS[1])
    else
        return 0
    end
    """

    return r.eval(lua_script, 1, lock_key, lock_id)

# 7. Bitmap (space-efficient)
def mark_user_active(user_id, date):
    """Track daily active users"""
    key = f'active_users:{date}'
    r.setbit(key, user_id, 1)

def count_active_users(date):
    key = f'active_users:{date}'
    return r.bitcount(key)

def was_user_active(user_id, date):
    key = f'active_users:{date}'
    return r.getbit(key, user_id) == 1

# For 100M users, only uses 12.5 MB per day!
```

DynamoDB Example:

```python
```

```python
import boto3

dynamodb = boto3.resource('dynamodb')

# Create table
table = dynamodb.create_table(
    TableName='Users',
    KeySchema=[
        {'AttributeName': 'user_id', 'KeyType': 'HASH'},  # Partition key
        {'AttributeName': 'timestamp', 'KeyType': 'RANGE'}  # Sort key
    ],
    AttributeDefinitions=[
        {'AttributeName': 'user_id', 'AttributeType': 'S'},
        {'AttributeName': 'timestamp', 'AttributeType': 'N'},
        {'AttributeName': 'email', 'AttributeType': 'S'}
    ],
    GlobalSecondaryIndexes=[
        {
            'IndexName': 'email-index',
            'KeySchema': [
                {'AttributeName': 'email', 'KeyType': 'HASH'}
            ],
            'Projection': {'ProjectionType': 'ALL'}
        }
    ],
    BillingMode='PAY_PER_REQUEST'  # or 'PROVISIONED'
)

# Put item
table.put_item(
    Item={
        'user_id': '123',
        'timestamp': 1234567890,
        'name': 'John Doe',
        'email': 'john@example.com',
        'age': 30
    }
)

# Get item (by primary key)
response = table.get_item(
    Key={
        'user_id': '123',
        'timestamp': 1234567890
    }
)
```

```
user = response['Item']

# Query (efficient - uses index)
response = table.query(
    KeyConditionExpression='user_id = :uid AND timestamp > :ts',
    ExpressionAttributeValues={
        ':uid': '123',
        ':ts': 1000000000
    }
)

# Scan (expensive - reads entire table)
response = table.scan(
    FilterExpression='age > :age',
    ExpressionAttributeValues={':age': 25}
)

# Batch operations
with table.batch_writer() as batch:
    for i in range(100):
        batch.put_item(Item={
            'user_id': str(i),
            'timestamp': int(time.time()),
            'name': f'User {i}'
        })
```

Graph Databases (Neo4j, ArangoDB)

When to Use:

```
✓ Relationship-heavy data
✓ Social networks
✓ Recommendation engines
✓ Fraud detection
✓ Network analysis
✓ Knowledge graphs

Graph databases excel at:
- Finding connections
- Shortest paths
- Pattern matching
- Traversals
```

Advanced Neo4j Examples:

cypher

```
// 1. Create social network
CREATE (john:Person {name: 'John', age: 30})
CREATE (jane:Person {name: 'Jane', age: 28})
CREATE (bob:Person {name: 'Bob', age: 32})
CREATE (alice:Person {name: 'Alice', age: 27})

CREATE (john)-[:FRIENDS_WITH {since: 2020}]->(jane)
CREATE (john)-[:FRIENDS_WITH {since: 2019}]->(bob)
CREATE (jane)-[:FRIENDS_WITH {since: 2021}]->(alice)
CREATE (bob)-[:FRIENDS_WITH {since: 2022}]->(alice)

CREATE (tech:Interest {name: 'Technology'})
CREATE (music:Interest {name: 'Music'})
CREATE (sports:Interest {name: 'Sports'})

CREATE (john)-[:INTERESTED_IN]->(tech)
CREATE (john)-[:INTERESTED_IN]->(music)
CREATE (jane)-[:INTERESTED_IN]->(tech)
CREATE (bob)-[:INTERESTED_IN]->(sports)
CREATE (alice)-[:INTERESTED_IN]->(music)

// 2. Friend recommendations (friends of friends)
MATCH (me:Person {name: 'John'})-[:FRIENDS_WITH]->(friend)-[:FRIENDS_WITH]->(fof)
WHERE NOT (me)-[:FRIENDS_WITH]->(fof) AND me <> fof
RETURN fof.name, COUNT(*) as mutual_friends
ORDER BY mutual_friends DESC

// Result: Alice (2 mutual friends: Jane and Bob)

// 3. Find path between people
MATCH path = shortestPath(
  (john:Person {name: 'John'})-[:FRIENDS_WITH*]-(alice:Person {name: 'Alice'})
)
RETURN path

// Result: John -> Jane -> Alice (or John -> Bob -> Alice)

// 4. Interest-based recommendations
MATCH (me:Person {name: 'John'})-[:INTERESTED_IN]->(interest)<-[:INTERESTED_IN]-(other)
WHERE NOT (me)-[:FRIENDS_WITH]->(other) AND me <> other
RETURN other.name, COLLECT(interest.name) as shared_interests
ORDER BY SIZE(shared_interests) DESC

// 5. Influence analysis (find most connected)
MATCH (p:Person)-[:FRIENDS_WITH]->(friend)
RETURN p.name, COUNT(friend) as connections
```

```
ORDER BY connections DESC
LIMIT 10

// 6. Community detection
CALL gds.louvain.stream({
  nodeProjection: 'Person',
  relationshipProjection: 'FRIENDS_WITH'
})
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name as name, communityId
ORDER BY communityId

// 7. Fraud detection pattern
MATCH (account1:Account)-[:TRANSFER]->(account2:Account)-[:TRANSFER]->(account3:Account)
WHERE account1 <> account3
  AND account1.created_at > datetime() - duration('P30D')
  AND account3.created_at > datetime() - duration('P30D')
RETURN account1, account2, account3
// Detects potential money laundering (quick in/out through intermediary)
```

Wide-Column Databases (Cassandra, HBase)

Data Model:

```
Column Family Structure:

Row Key: user_123
┌─────────────────────────────────────────┐
│ Column Family: profile                  │
├──────────────┬──────────────────────────┤
│ name         │ John Doe                 │
│ email        │ john@example.com         │
│ age          │ 30                       │
│ city         │ New York                 │
└──────────────┴──────────────────────────┘

┌─────────────────────────────────────────┐
│ Column Family: activity                 │
├──────────────┬──────────────────────────┤
│ 2024-01-15   │ login                    │
│ 2024-01-16   │ post_created             │
│ 2024-01-17   │ login                    │
└──────────────┴──────────────────────────┘

Benefits:
- Can add columns dynamically
- Sparse columns (not all rows have all columns)
- Time-series data (column = timestamp)
```

Cassandra Example:

```python
```

```python
from cassandra.cluster import Cluster
from cassandra.query import SimpleStatement

# Connect to cluster
cluster = Cluster(['127.0.0.1'])
session = cluster.connect()

# Create keyspace
session.execute("""
    CREATE KEYSPACE IF NOT EXISTS social_network
    WITH replication = {
        'class': 'SimpleStrategy',
        'replication_factor': 3
    }
""")

session.set_keyspace('social_network')

# Create table (wide-column model)
session.execute("""
    CREATE TABLE IF NOT EXISTS user_timeline (
        user_id UUID,
        post_time TIMESTAMP,
        post_id UUID,
        content TEXT,
        likes INT,
        PRIMARY KEY (user_id, post_time)
    ) WITH CLUSTERING ORDER BY (post_time DESC)
""")

# Insert data
session.execute("""
    INSERT INTO user_timeline (user_id, post_time, post_id, content, likes)
    VALUES (uuid(), toTimestamp(now()), uuid(), 'Hello world!', 0)
""")

# Query - efficient (uses partition key)
rows = session.execute("""
    SELECT * FROM user_timeline
    WHERE user_id = ?
    LIMIT 20
""", [user_id])

# Time-series query
rows = session.execute("""
    SELECT * FROM user_timeline
```

```
    WHERE user_id = ?
      AND post_time >= ?
      AND post_time < ?
""", [user_id, start_time, end_time])

# Counter column (for likes)
session.execute("""
    CREATE TABLE IF NOT EXISTS post_stats (
        post_id UUID PRIMARY KEY,
        like_count COUNTER,
        view_count COUNTER
    )
""")

# Increment counter
session.execute("""
    UPDATE post_stats
    SET like_count = like_count + 1
    WHERE post_id = ?
""", [post_id])
```

When to Use Wide-Column:

```
✓ Time-series data (logs, metrics, events)
✓ Write-heavy workloads
✓ Need horizontal scalability
✓ Can tolerate eventual consistency
✓ Simple query patterns

Examples:
- IoT sensor data
- Application logs
- User activity tracking
- Message history
- Metrics and monitoring
```

Key Takeaways

1. **Storage Types:**
    - Block: Low latency, databases, VMs
    - File: Shared access, content management
    - Object: Massive scale, cheap, media/backups

2. **Distributed File Systems:**
   - HDFS/GFS: Big data processing
   - Automatic replication and fault tolerance
   - Optimized for large sequential reads/writes
3. **Object Storage (S3):**
   - Unlimited scalability
   - Multiple storage tiers for cost optimization
   - Rich features (versioning, lifecycle, events)
4. **Database Types:**
   - Relational: ACID, complex queries
   - Document: Flexible schema, nested data
   - Key-Value: Simple, fast, caching
   - Graph: Relationships, recommendations
   - Wide-Column: Time-series, write-heavy

Practice Problems

1. Design a storage architecture for Netflix (storing and streaming petabytes of video)
2. Choose the right database type for: social network, time-series metrics, e-commerce transactions, recommendation engine
3. Calculate cost of storing 100TB: S3 Standard vs Glacier vs on-premises

Next Steps

You now understand the full spectrum of storage systems! In the next chapters, we'll explore how to use these systems in complete architectures.

Would you like me to:

1. Continue with more chapters?
2. Deep dive on specific storage system?
3. Practice designing complete systems?
4. Explore more code examples?