# Chapter 7: Database Scaling

## Introduction: Why Database Scaling Matters

When your application grows, the database becomes the bottleneck.

Growth Timeline:

Month 1: 1,000 users
- Single database handles everything fine ✓

Month 6: 100,000 users
- Database CPU at 80%
- Queries slowing down
- Backup takes hours

Year 1: 10,000,000 users
- Database overwhelmed!
- Writes queuing up
- Reads timing out
- Users experiencing errors
- Revenue lost!

Solution: Database Scaling

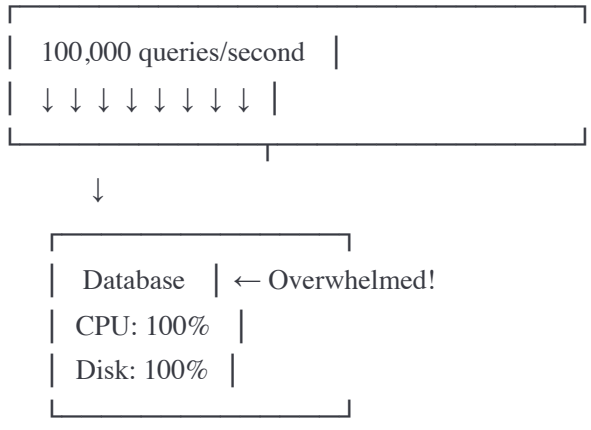
---

## 1. Read Replicas and Replication Lag

### The Read Scaling Problem

Single Database Problem:

All queries → One database



Symptoms:

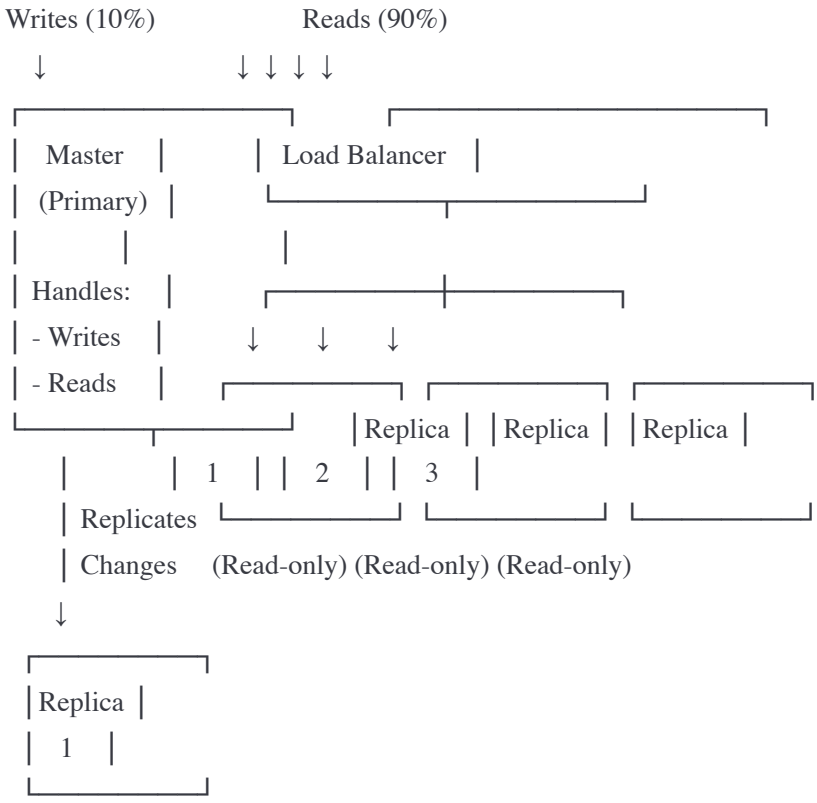
- Slow queries (500ms → 5 seconds)
- Timeouts
- Connection pool exhausted
- System crashes

---

### Read Replicas Solution

**Concept:** Copy data to multiple read-only databases.

With Read Replicas:




Benefits:

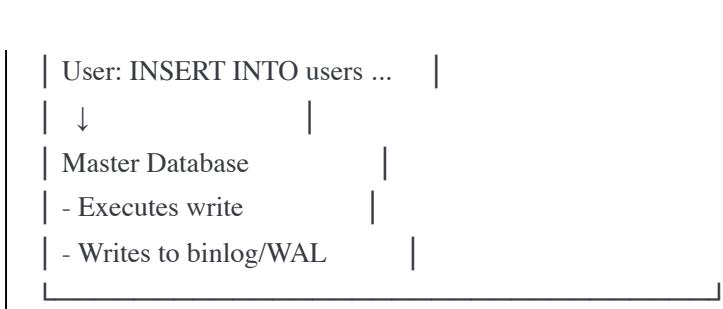
- Master: 10,000 writes/sec
- Each replica: 30,000 reads/sec
- Total read capacity: 90,000 reads/sec
- 9x read scaling!

---

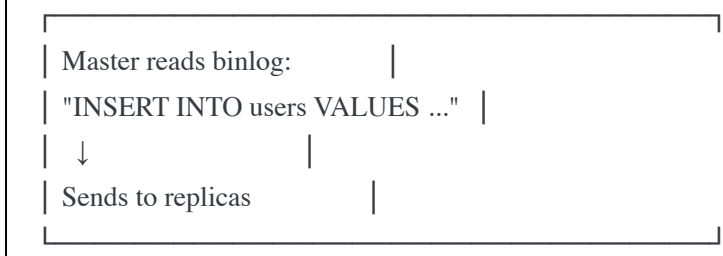
### How Replication Works

Step-by-Step:

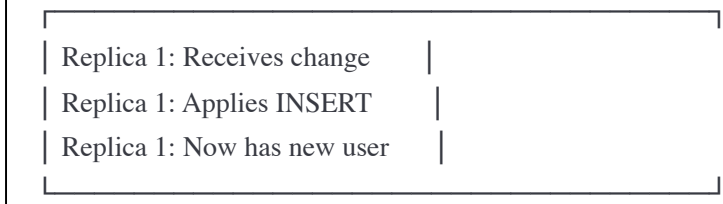
1. Write to Master




2. Master Ships Changes




3. Replicas Apply Changes




Timeline:
T=0ms: Write to master (complete)
T=5ms: Change shipped to replica
T=10ms: Replica applies change

---

**Replication Lag**

**The Problem:** Replicas are slightly behind master.

Replication Lag Example:

Time Master Replica

10:00 User creates post (no data yet)
POST #123 created

10:00 Response: "Post created!"
User redirected to view post

10:00 App reads from replica ← POST NOT THERE YET!
SELECT * FROM posts WHERE id=123
Result: Not found ✗

10:00.05 Replication completes POST #123 appears
(50ms lag)

Result: User sees "Post not found" error!
This is called "read-your-own-writes" problem.

### Measuring Replication Lag:

sql

```
-- PostgreSQL: Check replication lag
SELECT
    client_addr,
    state,
    sent_lsn,
    write_lsn,
    flush_lsn,
    replay_lsn,
    sync_state,
    -- Calculate lag in bytes
    pg_wal_lsn_diff(sent_lsn, replay_lsn) AS lag_bytes,
    -- Calculate lag in seconds
    EXTRACT(EPOCH FROM (now() - pg_last_xact_replay_timestamp())) AS lag_seconds
FROM pg_stat_replication;

-- Example output:
┌─────────────┬─────────┬───────────┬─────────────┐
│ client_addr │ state   │ lag_bytes │ lag_seconds │
├─────────────┼─────────┼───────────┼─────────────┤
│ 10.0.1.10   │ streaming │ 4096    │ 0.05        │
│ 10.0.1.11   │ streaming │ 16384   │ 0.15        │
│ 10.0.1.12   │ streaming │ 32768   │ 0.30        │
└─────────────┴─────────┴───────────┴─────────────┘

Interpretation:
- Replica 1: 50ms lag (excellent)
- Replica 2: 150ms lag (good)
- Replica 3: 300ms lag (acceptable)

Alert if lag > 1 second!
```

---

### Handling Replication Lag

### Strategy 1: Read from Master After Write

python

```python
class DatabaseRouter:
    def __init__(self):
        self.master = connect_to_master()
        self.replicas = [
            connect_to_replica('replica1'),
            connect_to_replica('replica2'),
            connect_to_replica('replica3')
        ]
        self.recent_writes = {}  # Track recent writes per user

    def write(self, user_id, query):
        """Execute write on master"""
        result = self.master.execute(query)

        # Remember this user just wrote
        self.recent_writes[user_id] = time.time()

        return result

    def read(self, user_id, query):
        """Smart read routing"""
        # Check if user wrote recently (last 1 second)
        if user_id in self.recent_writes:
            last_write = self.recent_writes[user_id]
            if time.time() - last_write < 1.0:
                # Read from master (has latest data)
                print(f"Reading from master for user {user_id} (recent write)")
                return self.master.execute(query)

        # Normal case: read from replica
        replica = random.choice(self.replicas)
        print(f"Reading from replica for user {user_id}")
        return replica.execute(query)

# Usage
router = DatabaseRouter()

# User creates post
router.write(user_id=123, query="INSERT INTO posts ...")

# User immediately views their post
# Reads from master (avoids replication lag!)
post = router.read(user_id=123, query="SELECT * FROM posts WHERE id=456")

# Another user views the post
```

```
# Can read from replica (doesn't need immediate consistency)
post = router.read(user_id=789, query="SELECT * FROM posts WHERE id=456")
```

---

### Strategy 2: Monotonic Reads (Session Consistency)

python

```python
class SessionConsistentRouter:
    """Ensure user always reads from same replica (or later)"""

    def __init__(self):
        self.master = connect_to_master()
        self.replicas = [
            connect_to_replica('replica1'),
            connect_to_replica('replica2'),
            connect_to_replica('replica3')
        ]
        self.user_replica_map = {}  # Pin users to replicas

    def get_replica_for_user(self, user_id):
        """Pin user to a specific replica"""
        if user_id not in self.user_replica_map:
            # Assign user to replica based on hash
            replica_idx = hash(user_id) % len(self.replicas)
            self.user_replica_map[user_id] = replica_idx

        return self.replicas[self.user_replica_map[user_id]]

    def read(self, user_id, query):
        """Always read from same replica for this user"""
        replica = self.get_replica_for_user(user_id)
        return replica.execute(query)

# Benefit: User never sees data "go backwards"
# - First read: sees posts 1, 2, 3
# - Second read: will see 1, 2, 3, 4, 5 (never less!)
```

---

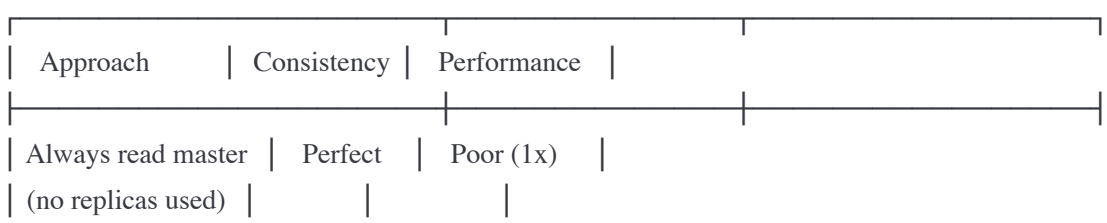
### Strategy 3: Causal Consistency (Version Tracking)

python

```python
class CausalConsistencyRouter:
    """Track data versions to ensure causality"""

    def write(self, query):
        """Write returns version number"""
        result = self.master.execute(query)
        version = self.master.get_current_log_position()
        return result, version

    def read(self, query, required_version=None):
        """Read from replica that has required version"""
        if required_version is None:
            # No version requirement, any replica is fine
            return random.choice(self.replicas).execute(query)

        # Find replica that has caught up to required version
        for replica in self.replicas:
            replica_version = replica.get_current_log_position()
            if replica_version >= required_version:
                return replica.execute(query)

        # No replica caught up yet, read from master
        return self.master.execute(query)

# Usage
router = CausalConsistencyRouter()

# User A creates post
result, version = router.write("INSERT INTO posts ...")

# User A shares link with User B
# Include version in URL: /posts/123?v=1234567

# User B views post
# Ensures they see the post (waits for replication if needed)
post = router.read("SELECT * FROM posts WHERE id=123", required_version=version)
```

---

### Replication Lag Trade-offs




Choose based on use case:

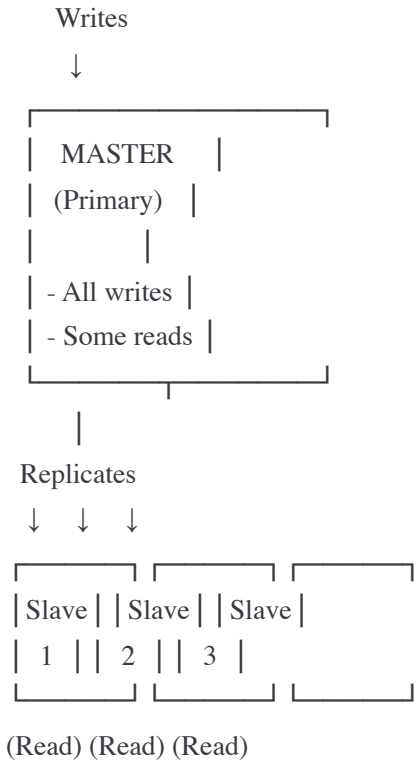
- Bank transactions: Read from master
- Social media: Eventual consistency fine
- User's own data: Read from master after write

---

## 2. Master-Slave vs Master-Master Replication

### Master-Slave (Primary-Replica)

**Architecture:** One master, multiple read-only replicas.




Characteristics:
- Single source of truth (master)
- Simple to reason about
- No write conflicts
- Scales reads only

**Configuration Example (MySQL):**

```sql
```

```
-- Master Configuration (my.cnf)
[mysqld]
server-id = 1
log-bin = /var/log/mysql/mysql-bin.log
binlog-format = ROW
binlog-do-db = myapp_db

-- Create replication user
CREATE USER 'replicator'@'%' IDENTIFIED BY 'password';
GRANT REPLICATION SLAVE ON *.* TO 'replicator'@'%';
FLUSH PRIVILEGES;

-- Get master status
SHOW MASTER STATUS;
-- Note: File and Position values needed for replica setup

-- Replica Configuration (my.cnf)
[mysqld]
server-id = 2
relay-log = /var/log/mysql/mysql-relay-bin
read-only = 1  -- Prevent writes to replica

-- Set up replication
CHANGE MASTER TO
  MASTER_HOST='master-db.example.com',
  MASTER_USER='replicator',
  MASTER_PASSWORD='password',
  MASTER_LOG_FILE='mysql-bin.000001',
  MASTER_LOG_POS=12345;

START SLAVE;

-- Check replication status
SHOW SLAVE STATUS\G
-- Look for: Slave_IO_Running: Yes
--           Slave_SQL_Running: Yes
--           Seconds_Behind_Master: 0
```

### Application Code:

```
python
```

```
import pymysql
from random import choice

class MasterSlaveDB:
    def __init__(self):
        self.master = pymysql.connect(
            host='master-db.example.com',
            user='app',
            password='password',
            database='myapp_db'
        )

        self.slaves = [
            pymysql.connect(host='slave1-db.example.com', ...),
            pymysql.connect(host='slave2-db.example.com', ...),
            pymysql.connect(host='slave3-db.example.com', ...)
        ]

    def execute_write(self, query, params):
        """All writes go to master"""
        cursor = self.master.cursor()
        cursor.execute(query, params)
        self.master.commit()
        return cursor.lastrowid

    def execute_read(self, query, params):
        """Reads go to random slave"""
        slave = choice(self.slaves)
        cursor = slave.cursor()
        cursor.execute(query, params)
        return cursor.fetchall()

# Usage
db = MasterSlaveDB()

# Write to master
user_id = db.execute_write(
    "INSERT INTO users (name, email) VALUES (%s, %s)",
    ("John", "john@example.com")
)

# Read from slave
users = db.execute_read(
    "SELECT * FROM users WHERE active = %s",
```

```
    (True,)
)
```

**Pros and Cons:**

✓ ADVANTAGES:
- Simple to set up and manage
- No write conflicts
- Clear data flow
- Easy to reason about consistency
- Most common architecture

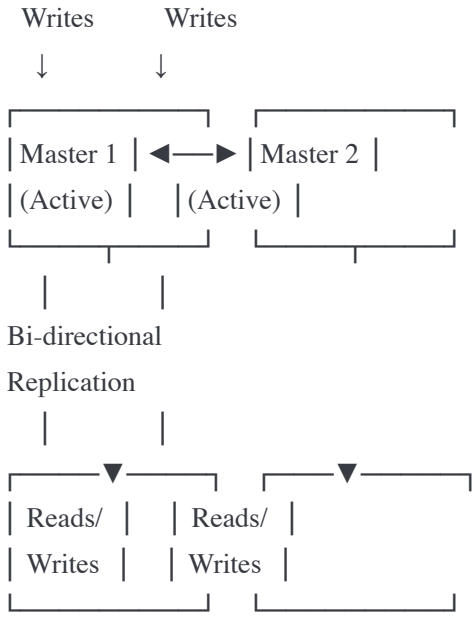
✗ DISADVANTAGES:
- Single point of failure (master)
- Master can become write bottleneck
- Doesn't scale writes
- Failover requires manual/automated promotion

When to use:
- Most applications (default choice)
- Read-heavy workloads (90% reads, 10% writes)
- When strong consistency for writes is needed

---

**Master-Master (Multi-Master)**

**Architecture:** Multiple masters, all accept writes.




Characteristics:

- Both accept writes
- Replicate to each other
- Complex conflict resolution
- Scales writes (to a point)
- High availability

**Conflict Example:**

The Write Conflict Problem:

Time Master 1 Master 2

10:00 User balance: $1000 User balance: $1000

10:00 User withdraws $500
Balance = $500

10:00 User withdraws $600
Balance = $400

10:01 Replication arrives Replication arrives
Sees Master 2's update Sees Master 1's update
Which is correct? Which is correct?

Options:
1. Last Write Wins (LWW): Use timestamp
- Problem: Both think they won!

2. Version Vectors: Track all changes
- Detect conflict, require manual resolution

3. Application Logic: Custom resolution
- Example: Sum all withdrawals, check if valid

**Conflict Resolution Strategies:**

sql

```sql
-- Strategy 1: Last Write Wins (LWW)
-- Use timestamp to decide winner

CREATE TABLE accounts (
    id INTEGER PRIMARY KEY,
    balance DECIMAL(10, 2),
    updated_at TIMESTAMP,
    updated_by VARCHAR(50)  -- Which master made the change
);

-- On conflict, keep the one with latest timestamp
-- Problem: Can lose data!

-- Strategy 2: Version Vectors
CREATE TABLE accounts (
    id INTEGER PRIMARY KEY,
    balance DECIMAL(10, 2),
    version_m1 INTEGER,  -- Version from master 1
    version_m2 INTEGER   -- Version from master 2
);

-- Master 1 write: version_m1++
-- Master 2 write: version_m2++
-- On replication, check if versions compatible
-- If conflict detected, flag for manual review

-- Strategy 3: CRDTs (Conflict-Free Replicated Data Types)
-- Use data structures that merge automatically
-- Example: Counter that only increments (never decreases)
-- Both masters can increment, sum is always correct

CREATE TABLE page_views (
    page_id INTEGER PRIMARY KEY,
    count_m1 INTEGER,  -- Increments from master 1
    count_m2 INTEGER   -- Increments from master 2
);

-- Total views = count_m1 + count_m2
-- No conflicts! Both can increment freely
```

**Configuration Example (PostgreSQL BDR):**

```
sql
```

```sql
-- Master 1 Configuration
CREATE EXTENSION pglogical;

-- Create node
SELECT pglogical.create_node(
    node_name := 'master1',
    dsn := 'host=master1.example.com dbname=myapp'
);

-- Create replication set
SELECT pglogical.create_replication_set('default');

-- Add all tables to replication
SELECT pglogical.replication_set_add_all_tables(
    'default',
    ARRAY['public']
);

-- Master 2 Configuration
CREATE EXTENSION pglogical;

SELECT pglogical.create_node(
    node_name := 'master2',
    dsn := 'host=master2.example.com dbname=myapp'
);

-- Subscribe to Master 1
SELECT pglogical.create_subscription(
    subscription_name := 'sub_from_master1',
    provider_dsn := 'host=master1.example.com dbname=myapp',
    replication_sets := ARRAY['default']
);

-- On Master 1, subscribe to Master 2
-- (Creates bi-directional replication)
SELECT pglogical.create_subscription(
    subscription_name := 'sub_from_master2',
    provider_dsn := 'host=master2.example.com dbname=myapp',
    replication_sets := ARRAY['default']
);
```

**Pros and Cons:**

✓ ADVANTAGES:
- High availability (both can handle writes)
- Scales writes across multiple masters
- Geographic distribution possible
- No single point of failure

✗ DISADVANTAGES:
- Complex conflict resolution
- Increased latency (cross-master replication)
- Can lose data on conflicts
- Harder to reason about
- More operational complexity

When to use:
- Multi-region deployments (master per region)
- High write throughput needed
- Can tolerate eventual consistency
- Application can handle conflicts

---

### Failover Strategies

### Automatic Failover (Master-Slave)

python

```
import time
import threading

class AutoFailover:
    def __init__(self):
        self.master = "master1.db.com"
        self.replicas = ["replica1.db.com", "replica2.db.com"]
        self.current_master = self.master

        # Start health check thread
        self.monitoring = True
        self.health_thread = threading.Thread(target=self._monitor_health)
        self.health_thread.start()

    def _monitor_health(self):
        """Continuously check master health"""
        failures = 0

        while self.monitoring:
            if not self._check_health(self.current_master):
                failures += 1
                print(f"Master health check failed ({failures}/3)")

                if failures >= 3:
                    # Master is down! Promote replica
                    self._failover()
                    failures = 0
            else:
                failures = 0

            time.sleep(5)  # Check every 5 seconds

    def _check_health(self, host):
        """Check if database is responsive"""
        try:
            conn = connect(host)
            conn.execute("SELECT 1")
            return True
        except:
            return False

    def _failover(self):
        """Promote replica to master"""
        print("🚨 Master failed! Initiating failover...")

        # Choose most up-to-date replica
```

```
        best_replica = self._choose_best_replica()

        print(f"Promoting {best_replica} to master")

        # 1. Stop replication on chosen replica
        self._stop_replication(best_replica)

        # 2. Make it writable
        self._make_writable(best_replica)

        # 3. Point other replicas to new master
        for replica in self.replicas:
            if replica != best_replica:
                self._repoint_replica(replica, best_replica)

        # 4. Update application connection
        self.current_master = best_replica

        print(f"✓ Failover complete. New master: {best_replica}")

        # 5. Alert operations team
        self._send_alert(f"Database failover: {best_replica} is now master")

    def _choose_best_replica(self):
        """Choose replica with least replication lag"""
        best_replica = None
        min_lag = float('inf')

        for replica in self.replicas:
            lag = self._get_replication_lag(replica)
            if lag < min_lag:
                min_lag = lag
                best_replica = replica

        return best_replica
```

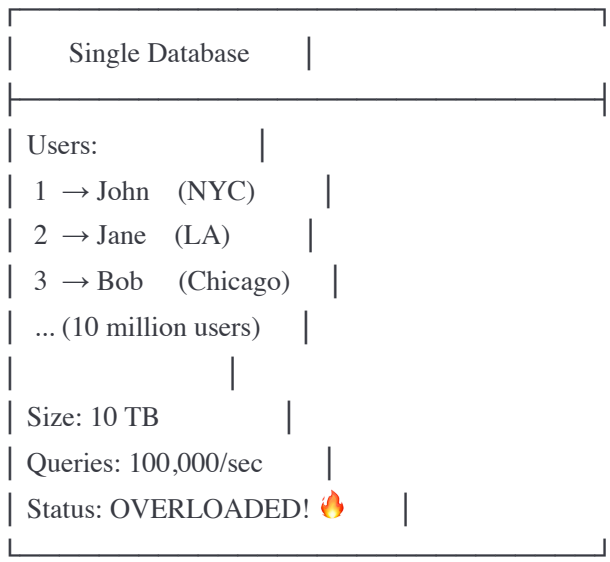
---

## 3. Database Sharding (Horizontal Partitioning)

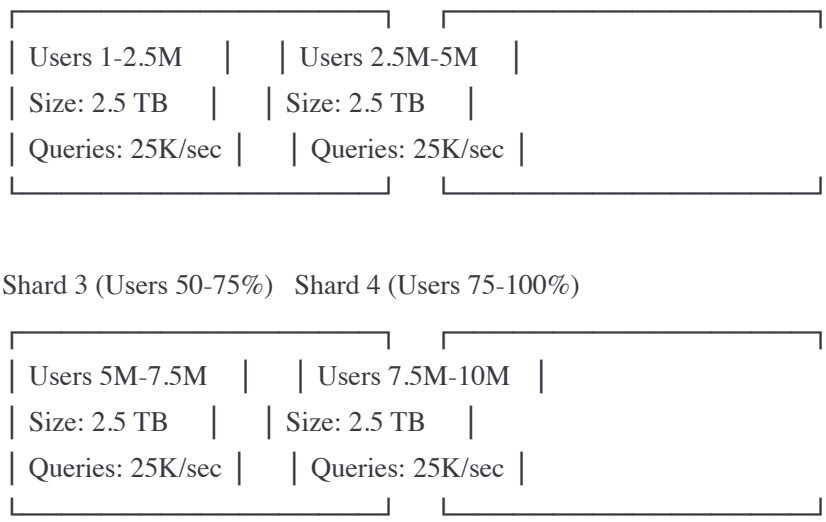
### What is Sharding?

**Sharding:** Split database into multiple independent databases (shards).

Before Sharding (Single Database):




After Sharding (4 Shards):




Benefits:

- 4x smaller databases (manageable)
- 4x write throughput
- 4x read throughput
- Can add more shards as you grow

---

## Why Shard?

Problem: Instagram's Growth

2010: 1 million users

- Single database: Works fine ✓

2012: 100 million users
- Single database: Struggling
- Queries slow (500ms)
- Backups take 12 hours
- Failover takes 1 hour

2015: 500 million users
- Single database: Impossible!
- Database size: 50+ TB
- Can't fit on one machine
- Can't handle write load

Solution: Shard by user_id
- Shard 1: Users 0-100M
- Shard 2: Users 100-200M
- Shard 3: Users 200-300M
- etc.

Result:
- Each shard: manageable size
- Linear scalability
- Can support billions of users

---

### Sharding Challenges

1. CROSS-SHARD QUERIES
   Problem: User A (Shard 1) follows User B (Shard 2)
   Query: "Get all posts from people I follow"
   Solution: Query both shards, merge results

2. CROSS-SHARD JOINS
   Problem: Can't JOIN across shards
   Solution: Denormalize data or query multiple shards

3. DISTRIBUTED TRANSACTIONS
   Problem: Transfer money between users on different shards
   Solution: Two-phase commit or eventual consistency

4. RESHARDING
   Problem: Adding new shards requires moving data
   Solution: Consistent hashing (covered later)

5. HOTSPOTS

Problem: One shard gets more traffic than others

Solution: Better shard key selection

---

## 4. Sharding Strategies

### Strategy 1: Hash-Based Sharding

**Concept:** Hash the shard key to determine shard.

Algorithm:

shard_number = hash(user_id) % number_of_shards

Example: 4 shards

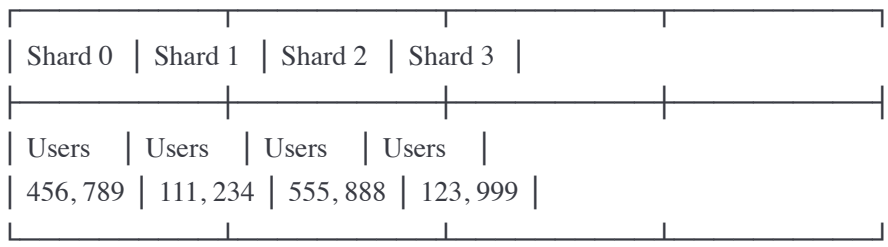
user_id = 123

shard = hash(123) % 4 = 3

user_id = 456

shard = hash(456) % 4 = 0

Distribution:




Hash function distributes users evenly

### Implementation:

python

```python
import hashlib

class HashBasedSharding:
    def __init__(self, shard_configs):
        """
        shard_configs: [
            {'host': 'shard0.db.com', 'db': 'shard_0'},
            {'host': 'shard1.db.com', 'db': 'shard_1'},
            {'host': 'shard2.db.com', 'db': 'shard_2'},
            {'host': 'shard3.db.com', 'db': 'shard_3'}
        ]
        """
        self.shards = [
            connect_to_db(config) for config in shard_configs
        ]
        self.num_shards = len(self.shards)

    def get_shard(self, user_id):
        """Determine which shard contains this user"""
        # Hash the user_id
        hash_value = int(hashlib.md5(str(user_id).encode()).hexdigest(), 16)

        # Modulo to get shard number
        shard_num = hash_value % self.num_shards

        return self.shards[shard_num]

    def insert_user(self, user_id, name, email):
        """Insert user into appropriate shard"""
        shard = self.get_shard(user_id)

        shard.execute(
            "INSERT INTO users (id, name, email) VALUES (%s, %s, %s)",
            (user_id, name, email)
        )

        print(f"Inserted user {user_id} into shard {shard_num}")

    def get_user(self, user_id):
        """Retrieve user from appropriate shard"""
        shard = self.get_shard(user_id)

        result = shard.execute(
            "SELECT * FROM users WHERE id = %s",
            (user_id,)
        )
```

```
        return result.fetchone()

    def get_user_posts(self, user_id):
        """Get posts for a user (same shard)"""
        shard = self.get_shard(user_id)

        result = shard.execute(
            "SELECT * FROM posts WHERE user_id = %s",
            (user_id,)
        )

        return result.fetchall()

    def get_feed(self, user_id, following_ids):
        """Get feed from multiple users (cross-shard query!)"""
        # Group users by shard
        shard_to_users = {}
        for followed_id in following_ids:
            shard_num = self.get_shard_number(followed_id)
            if shard_num not in shard_to_users:
                shard_to_users[shard_num] = []
            shard_to_users[shard_num].append(followed_id)

        # Query each shard
        all_posts = []
        for shard_num, user_ids in shard_to_users.items():
            shard = self.shards[shard_num]

            # Get posts from all users on this shard
            placeholders = ','.join(['%s'] * len(user_ids))
            result = shard.execute(
                f"SELECT * FROM posts WHERE user_id IN ({placeholders})",
                user_ids
            )

            all_posts.extend(result.fetchall())

        # Merge and sort
        all_posts.sort(key=lambda p: p['created_at'], reverse=True)

        return all_posts[:20]  # Top 20 posts

# Usage
sharding = HashBasedSharding([
    {'host': 'shard0.db.com'},
    {'host': 'shard1.db.com'},
```

```
    {'host': 'shard2.db.com'},
    {'host': 'shard3.db.com'}
])

# Insert user
sharding.insert_user(123, "John", "john@example.com")

# Get user (goes to correct shard automatically)
user = sharding.get_user(123)

# Get feed (queries multiple shards)
feed = sharding.get_feed(
    user_id=123,
    following_ids=[111, 222, 333, 444, 555]
)
```

**Pros and Cons:**

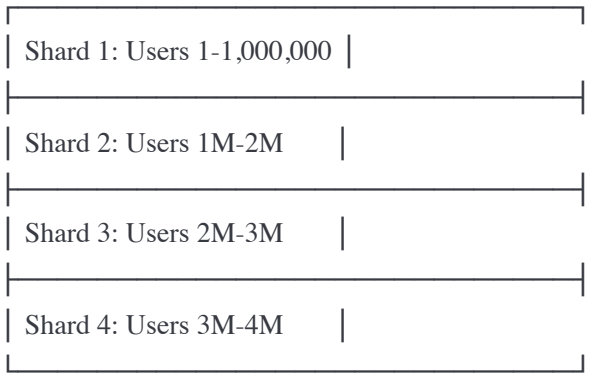
```
✓ ADVANTAGES:
  - Even distribution (if good hash function)
  - Simple to implement
  - No hotspots (assuming uniform access pattern)
  - Deterministic (same key always goes to same shard)

✗ DISADVANTAGES:
  - Adding shards requires rehashing ALL data!
  - Example: 4 shards → 5 shards
    - hash(123) % 4 = 3
    - hash(123) % 5 = 3 (lucky, same shard)
    - hash(456) % 4 = 0
    - hash(456) % 5 = 1 (different shard! Must move data)
  - Approximately 80% of data needs to move!
  - Can't do range queries

When to use:
- Stable number of shards
- Don't need range queries
- Access pattern is uniform
```

---

### Strategy 2: Range-Based Sharding

**Concept:** Shard based on ranges of values.

Shard by user_id ranges:




Or shard by date:


Or shard by geography:


**Implementation:**

python

```python
class RangeBasedSharding:
    def __init__(self):
        self.shard_map = [
            {'range': (0, 1_000_000), 'shard': connect_to_db('shard0.db.com')},
            {'range': (1_000_000, 2_000_000), 'shard': connect_to_db('shard1.db.com')},
            {'range': (2_000_000, 3_000_000), 'shard': connect_to_db('shard2.db.com')},
            {'range': (3_000_000, 4_000_000), 'shard': connect_to_db('shard3.db.com')}
        ]

    def get_shard(self, user_id):
        """Find shard containing this user_id"""
        for mapping in self.shard_map:
            start, end = mapping['range']
            if start <= user_id < end:
                return mapping['shard']

        raise ValueError(f"No shard found for user_id {user_id}")

    def get_users_in_range(self, start_id, end_id):
        """Efficiently query range (may span multiple shards)"""
        results = []

        for mapping in self.shard_map:
            shard_start, shard_end = mapping['range']

            # Check if this shard overlaps with query range
            if shard_start < end_id and shard_end > start_id:
                # Calculate overlap
                query_start = max(start_id, shard_start)
                query_end = min(end_id, shard_end)

                # Query this shard
                shard = mapping['shard']
                result = shard.execute(
                    "SELECT * FROM users WHERE id >= %s AND id < %s",
                    (query_start, query_end)
                )

                results.extend(result.fetchall())

        return results

# Example: Get users 1,500,000 to 2,500,000
# Queries Shard 2 and Shard 3 only!
```

```
sharding = RangeBasedSharding()
users = sharding.get_users_in_range(1_500_000, 2_500_000)
```

### Time-Based Sharding Example:

```
python
```

```python
from datetime import datetime, timedelta

class TimeBasedSharding:
    """Shard logs by date - useful for time-series data"""

    def __init__(self):
        self.shards = {
            '2024-Q1': connect_to_db('logs_2024_q1.db.com'),
            '2024-Q2': connect_to_db('logs_2024_q2.db.com'),
            '2024-Q3': connect_to_db('logs_2024_q3.db.com'),
            '2024-Q4': connect_to_db('logs_2024_q4.db.com')
        }

    def get_shard_for_date(self, date):
        """Determine shard based on date"""
        quarter = (date.month - 1) // 3 + 1
        shard_key = f"{date.year}-Q{quarter}"
        return self.shards[shard_key]

    def insert_log(self, timestamp, message):
        """Insert log into appropriate shard"""
        date = datetime.fromtimestamp(timestamp)
        shard = self.get_shard_for_date(date)

        shard.execute(
            "INSERT INTO logs (timestamp, message) VALUES (%s, %s)",
            (timestamp, message)
        )

    def query_logs(self, start_date, end_date):
        """Query logs across date range"""
        results = []

        # Determine which shards to query
        current = start_date
        while current <= end_date:
            shard = self.get_shard_for_date(current)

            result = shard.execute(
                "SELECT * FROM logs WHERE timestamp >= %s AND timestamp < %s",
                (current.timestamp(), (current + timedelta(days=90)).timestamp())
            )

            results.extend(result.fetchall())
            current += timedelta(days=90)
```

```
        return results

# Benefit: Old shards can be archived/deleted
# Query only relevant shards for date range
```

**Pros and Cons:**

✓ ADVANTAGES:
- Range queries are efficient
- Easy to add new shards (just add new range)
- Can archive old shards (for time-based)
- Intuitive and easy to understand

✗ DISADVANTAGES:
- Uneven distribution (hotspots!)
- Example: New users (high IDs) more active than old
- Recent data gets more traffic (for time-based)
- May need to rebalance ranges

When to use:
- Time-series data (logs, events)
- Need range queries
- Data has natural ranges (geography, dates)
- Can tolerate some imbalance

---

**Strategy 3: Directory-Based Sharding**

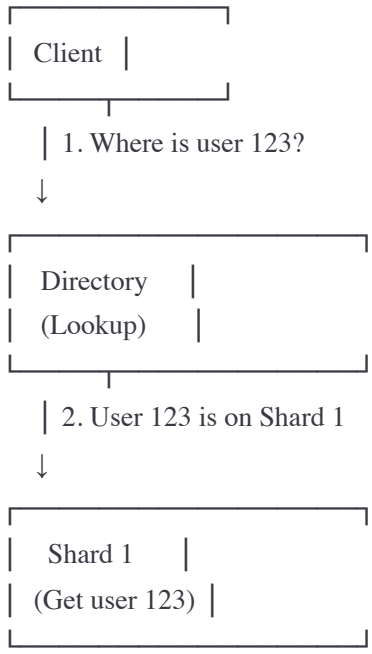
**Concept:** Maintain a lookup table (directory) that maps keys to shards.

Directory Table (Lookup Service):

| User_ID | Shard |
|---|---|
| 123 | Shard 1 |
| 456 | Shard 2 |
| 789 | Shard 1 |
| 111 | Shard 3 |
| 222 | Shard 2 |

Process:

1. Query directory: "Where is user 123?"
2. Directory returns: "Shard 1"
3. Query Shard 1 for user data




**Implementation:**

python

```python
class DirectoryBasedSharding:
    def __init__(self):
        # Directory: Maps user_id → shard_id
        self.directory = connect_to_db('directory.db.com')

        # Shards
        self.shards = {
            'shard_1': connect_to_db('shard1.db.com'),
            'shard_2': connect_to_db('shard2.db.com'),
            'shard_3': connect_to_db('shard3.db.com'),
            'shard_4': connect_to_db('shard4.db.com')
        }

    def get_shard_for_user(self, user_id):
        """Lookup which shard has this user"""
        result = self.directory.execute(
            "SELECT shard_id FROM user_shard_map WHERE user_id = %s",
            (user_id,)
        )

        row = result.fetchone()
        if not row:
            # User not found, assign to shard with most capacity
            shard_id = self._choose_shard_for_new_user()
            self._register_user(user_id, shard_id)
            return self.shards[shard_id]

        return self.shards[row['shard_id']]

    def _choose_shard_for_new_user(self):
        """Choose shard with most available capacity"""
        # Query each shard for size
        shard_sizes = {}
        for shard_id, shard in self.shards.items():
            result = shard.execute("SELECT COUNT(*) FROM users")
            shard_sizes[shard_id] = result.fetchone()[0]

        # Return shard with fewest users
        return min(shard_sizes, key=shard_sizes.get)

    def _register_user(self, user_id, shard_id):
        """Record user's shard in directory"""
        self.directory.execute(
            "INSERT INTO user_shard_map (user_id, shard_id) VALUES (%s, %s)",
            (user_id, shard_id)
        )
```

```
def insert_user(self, user_id, name, email):
    """Insert user"""
    shard = self.get_shard_for_user(user_id)

    shard.execute(
        "INSERT INTO users (id, name, email) VALUES (%s, %s, %s)",
        (user_id, name, email)
    )

def get_user(self, user_id):
    """Get user from appropriate shard"""
    shard = self.get_shard_for_user(user_id)

    result = shard.execute(
        "SELECT * FROM users WHERE id = %s",
        (user_id,)
    )

    return result.fetchone()

def move_user_to_shard(self, user_id, new_shard_id):
    """Rebalancing: Move user to different shard"""
    # Get user data from old shard
    old_shard = self.get_shard_for_user(user_id)
    user_data = old_shard.execute(
        "SELECT * FROM users WHERE id = %s",
        (user_id,)
    ).fetchone()

    # Insert into new shard
    new_shard = self.shards[new_shard_id]
    new_shard.execute(
        "INSERT INTO users (id, name, email) VALUES (%s, %s, %s)",
        (user_data['id'], user_data['name'], user_data['email'])
    )

    # Delete from old shard
    old_shard.execute("DELETE FROM users WHERE id = %s", (user_id,))

    # Update directory
    self.directory.execute(
        "UPDATE user_shard_map SET shard_id = %s WHERE user_id = %s",
        (new_shard_id, user_id)
    )

    print(f"Moved user {user_id} to {new_shard_id}")
```

```
# Usage
sharding = DirectoryBasedSharding()

# Insert user (automatically assigns to least-loaded shard)
sharding.insert_user(123, "John", "john@example.com")

# Later, can rebalance
sharding.move_user_to_shard(123, 'shard_2')
```

**Pros and Cons:**

✓ ADVANTAGES:
- Flexible (can place users anywhere)
- Easy to rebalance (just update directory)
- No rehashing when adding shards
- Can optimize placement (e.g., by geography, tenant size)
- Supports complex sharding logic

✗ DISADVANTAGES:
- Directory is single point of failure
- Directory can become bottleneck
- Extra lookup on every query (latency)
- Directory must be highly available

Solutions for directory scaling:
- Cache directory lookups
- Replicate directory
- Use distributed directory (e.g., ZooKeeper)

When to use:
- Need flexibility in shard assignment
- Plan to rebalance frequently
- Can tolerate extra lookup latency
- Complex sharding requirements

---

## 5. Consistent Hashing

Solves the "adding shards rehashes everything" problem.

**The Problem with Simple Hashing**

Simple Hash: shard = hash(key) % N

4 shards:

User 123 → hash(123) % 4 = 3 (Shard 3)
User 456 → hash(456) % 4 = 0 (Shard 0)
User 789 → hash(789) % 4 = 1 (Shard 1)

Add 1 shard (now 5 shards):
User 123 → hash(123) % 5 = 3 (Shard 3) ✓ Same!
User 456 → hash(456) % 5 = 1 (Shard 1) ✗ Moved!
User 789 → hash(789) % 5 = 4 (Shard 4) ✗ Moved!

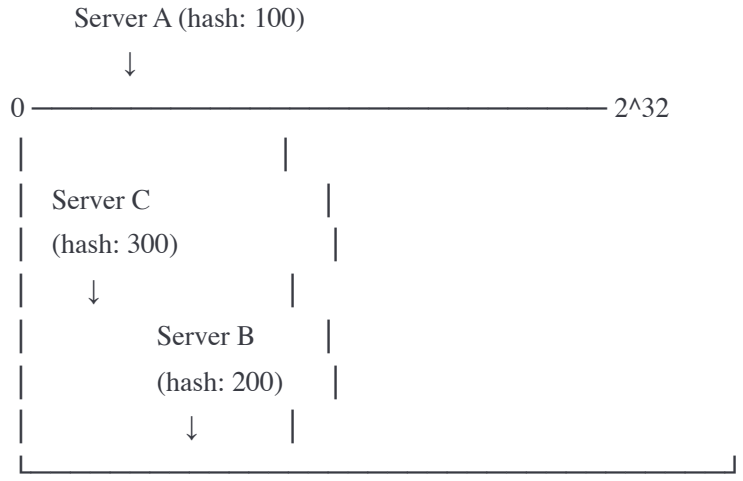
Result: ~80% of keys move to different shards!
Must migrate 80% of data!

---

**Consistent Hashing Solution**

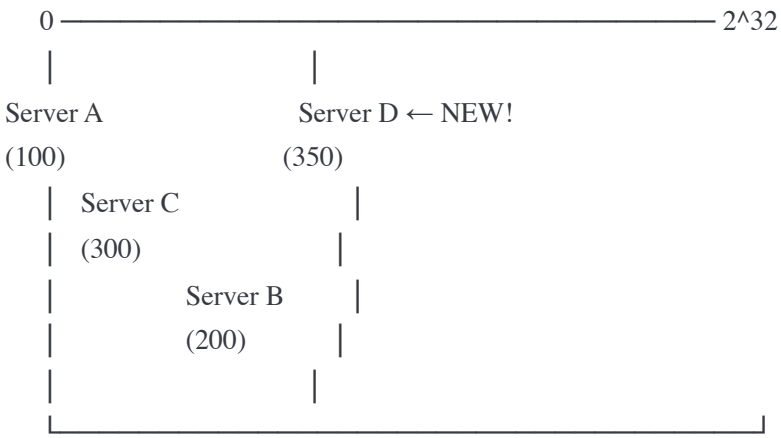
**Concept:** Hash keys and servers onto a ring. Key goes to next server clockwise.

Hash Ring (0 to 2^32-1):




Keys:
- Key X (hash: 50) → Server A (next clockwise)
- Key Y (hash: 150) → Server B (next clockwise)
- Key Z (hash: 250) → Server C (next clockwise)

Add Server D (hash: 350):




Keys affected:
- Key X (50) → Still Server A ✓
- Key Y (150) → Still Server B ✓
- Key Z (250) → Still Server C ✓
- NEW keys (325-350) → Server D

Only ~25% of keys moved! (1/4 servers)

---

## Implementation

python

```python
import hashlib
import bisect

class ConsistentHash:
    def __init__(self, nodes=None, virtual_nodes=150):
        """
        nodes: List of server addresses
        virtual_nodes: Number of virtual nodes per physical node
                       (helps with even distribution)
        """
        self.virtual_nodes = virtual_nodes
        self.ring = {}  # hash_value → node
        self.sorted_keys = []  # Sorted list of hash values
        self.nodes = set()

        if nodes:
            for node in nodes:
                self.add_node(node)

    def _hash(self, key):
        """Hash function: string → integer"""
        return int(hashlib.md5(key.encode()).hexdigest(), 16)

    def add_node(self, node):
        """Add a server to the ring"""
        self.nodes.add(node)

        # Add virtual nodes for better distribution
        for i in range(self.virtual_nodes):
            virtual_key = f"{node}:{i}"
            hash_value = self._hash(virtual_key)

            self.ring[hash_value] = node
            bisect.insort(self.sorted_keys, hash_value)

        print(f"Added node {node} with {self.virtual_nodes} virtual nodes")

    def remove_node(self, node):
        """Remove a server from the ring"""
        self.nodes.discard(node)

        # Remove all virtual nodes
        for i in range(self.virtual_nodes):
            virtual_key = f"{node}:{i}"
            hash_value = self._hash(virtual_key)
```

```
            del self.ring[hash_value]
            self.sorted_keys.remove(hash_value)

        print(f"Removed node {node}")

    def get_node(self, key):
        """Find which server should handle this key"""
        if not self.ring:
            return None

        hash_value = self._hash(str(key))

        # Find first node clockwise on ring
        # Binary search for first hash_value >= key's hash
        idx = bisect.bisect_right(self.sorted_keys, hash_value)

        # Wrap around if necessary
        if idx == len(self.sorted_keys):
            idx = 0

        return self.ring[self.sorted_keys[idx]]

    def get_distribution(self, num_keys=10000):
        """Analyze how evenly keys are distributed"""
        distribution = {node: 0 for node in self.nodes}

        for i in range(num_keys):
            node = self.get_node(f"key_{i}")
            distribution[node] += 1

        return distribution

# Usage
ch = ConsistentHash([
    'server1.db.com',
    'server2.db.com',
    'server3.db.com'
])

# Get node for keys
print(ch.get_node('user_123'))  # → server2.db.com
print(ch.get_node('user_456'))  # → server1.db.com
print(ch.get_node('user_789'))  # → server3.db.com

# Check distribution
print(ch.get_distribution())
# {
```

```
#   'server1.db.com': 3341,  (~33%)
#   'server2.db.com': 3329,  (~33%)
#   'server3.db.com': 3330   (~33%)
# }

# Add new server
ch.add_node('server4.db.com')

# Check distribution again
print(ch.get_distribution())
# {
#   'server1.db.com': 2501,  (~25%)
#   'server2.db.com': 2499,  (~25%)
#   'server3.db.com': 2500,  (~25%)
#   'server4.db.com': 2500   (~25%)
# }

# Only ~25% of keys moved to new server!
# Rest stayed on same servers
```

---

### Virtual Nodes

**Problem:** With few servers, distribution may be uneven.

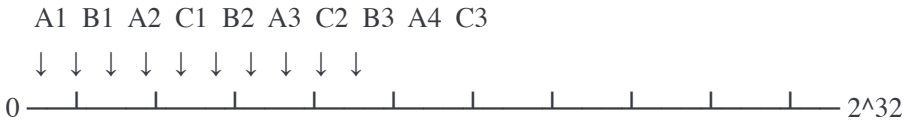
Without Virtual Nodes (3 servers):
Server A might get 40% of keys
Server B might get 35% of keys
Server C might get 25% of keys

With Virtual Nodes (3 servers × 150 vnodes = 450 points on ring):
Server A gets ~33.3% of keys
Server B gets ~33.3% of keys
Server C gets ~33.4% of keys

More points = better distribution!

**Visual:**

Ring with Virtual Nodes:




Each physical server has multiple positions on ring

Keys distributed more evenly

---

## 6. Database Partitioning Techniques

**Partitioning:** Splitting a single table into multiple physical tables.

### Vertical Partitioning

### Split table by columns

Original Users Table:

```
┌────┬──────┬─────────┬──────┬──────┬──────────┐
│ ID │ Name │  Email  │ City │ Bio  │ Settings │
├────┼──────┼─────────┼──────┼──────┼──────────┤
│ 1  │ John │ j@x.com │ NYC  │ Long... │ JSON... │
│ 2  │ Jane │ ja@x.com│ LA   │ Long... │ JSON... │
└────┴──────┴─────────┴──────┴──────┴──────────┘
```

After Vertical Partitioning:

Users_Core (Frequently accessed):

```
┌────┬──────┬─────────┬──────┐
│ ID │ Name │  Email  │ City │
├────┼──────┼─────────┼──────┤
│ 1  │ John │ j@x.com │ NYC  │
│ 2  │ Jane │ ja@x.com│ LA   │
└────┴──────┴─────────┴──────┘
```

Users_Extended (Rarely accessed):

```
┌────┬──────┬──────────┐
│ ID │ Bio  │ Settings │
├────┼──────┼──────────┤
│ 1  │ Long... │ JSON... │
│ 2  │ Long... │ JSON... │
└────┴──────┴──────────┘
```

Benefits:
- Smaller, faster queries on core data
- Can optimize storage differently
- Reduce I/O for common queries

**SQL Example:**

sql

```sql
-- Original query scans all columns
SELECT name, email FROM users WHERE city = 'NYC';
-- Scans: ID, name, email, city, bio, settings

-- After partitioning
SELECT name, email FROM users_core WHERE city = 'NYC';
-- Scans: ID, name, email, city (less I/O!)

-- Only fetch extended data when needed
SELECT bio FROM users_extended WHERE id = 123;
```

---

### Horizontal Partitioning (Same as Sharding)

### Split table by rows

Already covered in sharding section!

Users Table → Split into 4 partitions

Partition 1: Users 0-250K
Partition 2: Users 250K-500K
Partition 3: Users 500K-750K
Partition 4: Users 750K-1M

---

### List Partitioning

### Partition by discrete values

sql

```sql
-- Partition by country
CREATE TABLE users (
    id SERIAL,
    name VARCHAR(100),
    email VARCHAR(255),
    country VARCHAR(2)
) PARTITION BY LIST (country);

-- Create partitions
CREATE TABLE users_us PARTITION OF users FOR VALUES IN ('US');
CREATE TABLE users_uk PARTITION OF users FOR VALUES IN ('UK');
CREATE TABLE users_ca PARTITION OF users FOR VALUES IN ('CA');
CREATE TABLE users_other PARTITION OF users DEFAULT;

-- Queries automatically route to correct partition
SELECT * FROM users WHERE country = 'US';
-- Only scans users_us partition!

-- Benefit: Can optimize each partition differently
-- Example: US partition on fast SSD, others on slower storage
```

---

### Range Partitioning (PostgreSQL)

sql

```sql
-- Partition by date
CREATE TABLE logs (
    id SERIAL,
    timestamp TIMESTAMP,
    message TEXT
) PARTITION BY RANGE (timestamp);

-- Create partitions
CREATE TABLE logs_2024_q1 PARTITION OF logs
    FOR VALUES FROM ('2024-01-01') TO ('2024-04-01');

CREATE TABLE logs_2024_q2 PARTITION OF logs
    FOR VALUES FROM ('2024-04-01') TO ('2024-07-01');

CREATE TABLE logs_2024_q3 PARTITION OF logs
    FOR VALUES FROM ('2024-07-01') TO ('2024-10-01');

CREATE TABLE logs_2024_q4 PARTITION OF logs
    FOR VALUES FROM ('2024-10-01') TO ('2025-01-01');

-- Query automatically prunes partitions
SELECT * FROM logs
WHERE timestamp >= '2024-06-01'
  AND timestamp < '2024-07-01';
-- Only scans logs_2024_q2!

-- Easy to drop old data
DROP TABLE logs_2024_q1;  -- Removes old logs instantly
```

---

### Hash Partitioning (PostgreSQL)

sql

```sql
-- Partition by hash of user_id
CREATE TABLE users (
    id INTEGER,
    name VARCHAR(100),
    email VARCHAR(255)
) PARTITION BY HASH (id);

-- Create partitions
CREATE TABLE users_p0 PARTITION OF users
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);

CREATE TABLE users_p1 PARTITION OF users
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);

CREATE TABLE users_p2 PARTITION OF users
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);

CREATE TABLE users_p3 PARTITION OF users
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);

-- Distributes data evenly across 4 partitions
-- hash(id) % 4 determines partition
```

---

## Key Takeaways

1. **Read Replicas:**
   - Scale reads by adding replicas
   - Handle replication lag with smart routing
   - Read from master after write for consistency
2. **Replication Types:**
   - Master-Slave: Simple, one source of truth, scales reads
   - Master-Master: Complex, scales writes, conflict resolution needed
3. **Sharding:**
   - Split database horizontally for massive scale
   - Hash-based: Even distribution, hard to add shards
   - Range-based: Easy range queries, potential hotspots
   - Directory-based: Flexible, extra lookup overhead
4. **Consistent Hashing:**

- Solves resharding problem
- Only K/N keys move when adding/removing nodes
- Use virtual nodes for better distribution

5. **Partitioning:**
   - Vertical: Split by columns (hot/cold data)
   - Horizontal: Split by rows (sharding)
   - List/Range/Hash: Different access patterns

## Practice Problems

1. Design a sharding strategy for Instagram with 1 billion users and 500 billion photos.
2. Calculate how much data moves when adding a 5th shard to a 4-shard hash-based system with 1TB total data.
3. Design a multi-region database architecture for a global social network with users in US, Europe, and Asia.

## Next Chapter Preview

In Chapter 8, we'll explore **Storage Systems**:

- Block vs Object vs File storage
- Distributed file systems (HDFS, GFS)
- Blob storage (S3, Azure Blob)
- Different database types and when to use them

Ready to continue?