Database Sharding and Partitioning in Distributed Systems

Overview

Partitioning and **Sharding** are techniques used to distribute data across multiple storage units to improve performance, scalability, and manageability of large databases.

- **Partitioning**: Dividing data within a single database instance
- **Sharding**: Distributing data across multiple database instances/servers

Types of Partitioning

1. Horizontal Partitioning (Row-based)

Splits table rows across multiple partitions based on some criteria.

```sql
-- Example: Partition users by registration year
CREATE TABLE users_2023 (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100),
    created_at TIMESTAMP
) WHERE created_at >= '2023-01-01' AND created_at < '2024-01-01';

CREATE TABLE users_2024 (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100),
    created_at TIMESTAMP
) WHERE created_at >= '2024-01-01' AND created_at < '2025-01-01';
```

2. Vertical Partitioning (Column-based)

Splits table columns across multiple partitions.

```sql
-- Split user table into frequently and rarely accessed columns
CREATE TABLE users_core (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100)
);

CREATE TABLE users_extended (
    id INT PRIMARY KEY,
    bio TEXT,
    preferences JSON,
    last_login TIMESTAMP
);
```

3. Functional Partitioning

Splits data by feature or service boundaries.

```
User Service DB: users, profiles, authentication
Order Service DB: orders, payments, shipping
Product Service DB: products, inventory, categories
```

Sharding Strategies

1. Range-based Sharding

Distributes data based on value ranges.

```
Shard 1: user_id 1-1000000
Shard 2: user_id 1000001-2000000
Shard 3: user_id 2000001-3000000
```

Pros: Simple, range queries efficient **Cons**: Uneven distribution, hotspots possible

2. Hash-based Sharding

Uses hash function to determine shard placement.

```python
def get_shard(user_id, num_shards):
    return hash(user_id) % num_shards

# user_id 12345 → hash(12345) % 4 → Shard 2
```

Pros: Even distribution, simple implementation **Cons**: Range queries difficult, resharding complex

3. Directory-based Sharding

Maintains a lookup service to map data to shards.

```
Directory Service:
user_id 1-500000 → Shard A
user_id 500001-800000 → Shard B
user_id 800001-1000000 → Shard C
```

Pros: Flexible, supports complex sharding logic **Cons**: Additional complexity, single point of failure

4. Consistent Hashing

Distributes data using a hash ring to minimize resharding impact.

```
Hash Ring: [0 ──── Shard A ──── Shard B ──── Shard C ──── 2^32-1]
                     ↑              ↑              ↑
                Token 1M       Token 1.5B    Token 3B
```

Implementation Architecture

Master-Slave Sharding

```
                    Application Layer
                           |
                   Shard Router/Proxy
                    /       |       \
             Shard 1      Shard 2     Shard 3
            /       \    /       \   /       \
       Master    Slave Master   Slave Master  Slave
```

Federated Sharding

```
Application Layer
       |
Query Coordinator
   /       |       \
Service A   Service B   Service C
   |           |           |
Shard A1    Shard B1    Shard C1
Shard A2    Shard B2    Shard C2
```

Cross-Shard Operations

1. Cross-Shard Queries

sql

```sql
-- Challenge: Find all orders for users in different shards
-- Solution: Fan-out query with aggregation

-- Query Coordinator pseudocode:
results = []
for shard in shards:
    shard_result = shard.query("SELECT * FROM orders WHERE user_id IN (?)", user_ids)
    results.append(shard_result)
return merge_and_sort(results)
```

2. Distributed Transactions

python

```python
# Two-Phase Commit Protocol
def distributed_transaction():
    transaction_id = generate_id()

    # Phase 1: Prepare
    prepare_results = []
    for shard in involved_shards:
        result = shard.prepare(transaction_id, operations)
        prepare_results.append(result)

    # Phase 2: Commit or Abort
    if all(prepare_results):
        for shard in involved_shards:
            shard.commit(transaction_id)
    else:
        for shard in involved_shards:
            shard.abort(transaction_id)
```

3. Cross-Shard Joins

python

```python
# Hash Join across shards
def cross_shard_join(table_a_shards, table_b_shards, join_key):
    # Phase 1: Redistribute data by join key
    redistributed_a = redistribute_by_hash(table_a_shards, join_key)
    redistributed_b = redistribute_by_hash(table_b_shards, join_key)

    # Phase 2: Perform local joins
    results = []
    for partition in range(num_partitions):
        local_result = local_join(redistributed_a[partition],
                                  redistributed_b[partition])
        results.append(local_result)

    return merge_results(results)
```

Resharding Strategies

1. Live Migration

python

```python
def live_resharding():
    # 1. Start dual-write to old and new shards
    enable_dual_write()

    # 2. Migrate existing data
    for batch in get_data_batches():
        new_shard.write(batch)
        verify_consistency(batch)

    # 3. Switch reads to new shard
    switch_reads_to_new_shard()

    # 4. Stop dual-write, remove old shard
    disable_dual_write()
```

2. Consistent Hashing Resharding

python

```python
def add_new_shard():
    # Only affects adjacent ranges in hash ring
    # Minimal data movement required

    old_range = [token_start, token_end]
    new_ranges = split_range(old_range, 2)

    migrate_data(old_range, new_ranges)
    update_routing_table(new_ranges)
```

Challenges and Solutions

1. Hotspots

Problem: Uneven load distribution **Solutions**:

- Use composite sharding keys
- Implement automatic load balancing
- Split hot shards dynamically

2. Cross-Shard Consistency

Problem: ACID properties across shards **Solutions**:

- Saga Pattern for distributed transactions
- Event-driven eventual consistency
- Careful transaction boundary design

3. Operational Complexity

Problem: Monitoring, backup, maintenance across shards **Solutions**:

- Automated shard management tools
- Centralized monitoring and alerting
- Standardized deployment procedures

Best Practices

1. Shard Key Selection

```python
# Good shard keys:
user_id  # High cardinality, even distribution
tenant_id + timestamp  # Composite key for multi-tenant apps

# Poor shard keys:
status  # Low cardinality (only few values)
timestamp  # Creates hotspots for recent data
```

2. Query Optimization

```sql
-- Include shard key in WHERE clauses
SELECT * FROM orders
WHERE user_id = 12345 AND order_date > '2024-01-01';

-- Avoid cross-shard queries when possible
-- Bad: SELECT COUNT(*) FROM orders;
-- Good: SELECT COUNT(*) FROM orders WHERE user_id = 12345;
```

3. Application Design

```python
class ShardAwareService:
    def __init__(self, shard_router):
        self.router = shard_router

    def get_user_orders(self, user_id):
        # Single shard query - efficient
        shard = self.router.get_shard(user_id)
        return shard.query("SELECT * FROM orders WHERE user_id = ?", user_id)

    def get_global_stats(self):
        # Cross-shard aggregation - use caching/precomputation
        return self.get_cached_stats() or self.compute_stats()
```

Popular Sharding Solutions

1. Database-Native

- **MySQL Cluster**: Automatic sharding with MySQL
- **MongoDB**: Built-in sharding with shard keys
- **PostgreSQL**: pg_shard, Citus extensions

2. Middleware Solutions

- **Vitess**: YouTube's MySQL sharding solution
- **Apache ShardingSphere**: Database middleware
- **ProxySQL**: MySQL proxy with sharding capabilities

3. Application-Level

- **Custom sharding logic**: Full control, highest complexity
- **Framework support**: Django, Rails sharding gems
- **Microservices**: Service-based data partitioning

Monitoring and Metrics

Key Metrics to Track

python

```python
metrics = {
    'shard_distribution': 'Data size per shard',
    'query_latency': 'Response time per shard',
    'cross_shard_queries': 'Expensive operations count',
    'rebalancing_frequency': 'Resharding operations',
    'hotspot_detection': 'Uneven load patterns'
}
```

Health Checks

python

```python
def shard_health_check():
    for shard in shards:
        # Check connectivity
        assert shard.ping()

        # Check replication lag
        assert shard.replication_lag() < threshold

        # Check disk usage
        assert shard.disk_usage() < 80%

        # Check query performance
        assert shard.avg_query_time() < sla_limit
```

Conclusion

Database sharding and partitioning are essential techniques for scaling distributed systems. Success depends on:

1. **Careful planning**: Choose appropriate sharding strategy and keys
2. **Application design**: Build shard-aware applications from the start
3. **Operational excellence**: Implement proper monitoring and automation
4. **Gradual adoption**: Start simple, add complexity as needed

The key is balancing performance benefits with operational complexity while maintaining data consistency and system reliability.