Database Partitioning & Composite Key Design
A Comprehensive Guide to High-Performance Data Architecture

Table of Contents

Introduction

In today's data-driven world, efficiently managing large datasets is crucial for application performance and user experience. This guide explores advanced techniques for **partition filtering** and **composite key design** that can dramatically improve query performance, reduce resource consumption, and enhance scalability.

> **Key Benefit**: Proper partitioning can reduce query execution time by 90% or more on large datasets by scanning only relevant data portions.

Partition Filtering Best Practices

🎯 **Core Principles**

Partition filtering focuses on efficiently narrowing down data scanned during query execution, reducing latency and optimizing resource usage.

1. Choose Effective Partition Keys
- **Time-based partitioning**: Most common and effective for analytical workloads
- **Low to medium cardinality**: Avoid too many small partitions
- **Query alignment**: Match your most frequent query patterns

💡 Application Example: E-commerce order data partitioned by `order_date` allows fast retrieval of recent orders, monthly reports, and seasonal analysis.

2. Implement Time-Based Partitioning
```sql
-- Example: Daily partitioning for order data
CREATE TABLE orders_2025_01_15 PARTITION OF orders
FOR VALUES FROM ('2025-01-15') TO ('2025-01-16');
```

🚀 Real-World Impact: A retail company reduced their daily sales report generation from 45 minutes to 3 minutes by partitioning transaction data by date.

3. Balance Partition Size
- **Optimal range**: 100MB to few GB per partition
- **Avoid extremes**: Too small (high overhead) or too large (slow queries)
- **Memory consideration**: Partitions should fit comfortably in available memory

4. Enable Partition Pruning
```sql
-- Good: Explicitly filter on partition key
SELECT * FROM orders
WHERE order_date >= '2025-01-01' AND order_date < '2025-02-01';

-- Bad: Query without partition key filter
SELECT * FROM orders WHERE customer_id = 12345;
```

⚡ Performance Tip: Always include partition key predicates in WHERE clauses to benefit from automatic partition pruning.

Effective Partition Key Selection

🔑 **Strategic Considerations**

High Cardinality Keys
- **Best for**: User IDs, order IDs, session IDs
- **Benefit**: Even data distribution across partitions
- **Avoid**: Sequential timestamps alone (creates hotspots)

Composite Partition Keys
```sql
-- Example: Multi-level partitioning
CREATE TABLE user_events (
    user_id BIGINT,
```

```
    event_date DATE,
    event_type VARCHAR(50),
    event_data JSONB
) PARTITION BY RANGE (event_date, user_id);
```

🎯 Application Scenarios:
- **Multi-tenant SaaS**: `(tenant_id, created_date)`
- **IoT Data**: `(device_type, timestamp)`
- **Social Media**: `(user_id, post_date)`

Stability Requirement
> **Critical**: Partition key values should never change after creation, as this requires expensive data migrations.

Composite Key Design Strategies

🏗️ **Foundation Principles**

Composite keys use multiple columns together as primary keys, offering superior performance and data integrity in complex systems.

Key Design Patterns

Master-Detail Relationships
```sql
CREATE TABLE order_items (
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT,
    unit_price DECIMAL(10,2),
    PRIMARY KEY (order_id, product_id)
);
```

💼 Business Value: Ensures one product per order line while enabling fast retrieval of all items for a specific order.

Multi-Tenant Architecture
```sql
CREATE TABLE tenant_users (
    tenant_id UUID NOT NULL,
```

```
    user_id BIGINT NOT NULL,
    username VARCHAR(100),
    email VARCHAR(255),
    PRIMARY KEY (tenant_id, user_id)
);
```

🏢 Enterprise Application: Perfect for SaaS platforms serving multiple organizations with isolated data requirements.

Column Selection Criteria

✅ Ideal Characteristics:
- Frequently used in query filters
- Stable (values don't change)
- NOT NULL requirement
- High selectivity when combined

❌ Avoid:
- Columns with frequent updates
- Very large text fields
- Optional/nullable fields

Data Types in Composite Keys

📊 **Performance-Optimized Choices**

Recommended Data Types

Data Type	**Use Case**	**Performance**	**Storage**
`INT/BIGINT`	ID fields, counters	⭐⭐⭐⭐⭐	4-8 bytes
`CHAR(n)`	Fixed codes, statuses	⭐⭐⭐⭐	n bytes
`DATE/TIMESTAMP`	Time-based keys	⭐⭐⭐⭐	8-12 bytes
`VARCHAR(n)`	Variable text	⭐⭐⭐	Variable
`TEXT/BLOB`	Large content	⭐	Very large

Optimal Composite Key Example
```sql
CREATE TABLE product_reviews (
    product_id INT NOT NULL,          -- 4 bytes, high selectivity
    user_id BIGINT NOT NULL,          -- 8 bytes, ensures uniqueness
```

```
    review_date DATE NOT NULL,        -- 4 bytes, enables time-based queries
    rating SMALLINT,
    review_text TEXT,
    PRIMARY KEY (product_id, user_id, review_date)
);
```

🎯 Why This Works:
- Compact key size (16 bytes total)
- Supports common query patterns
- Efficient indexing and joins
- Natural data clustering

UUID vs Integer Considerations

⚖️ **Decision Matrix**

Aspect	**UUID**	**Integer**
Uniqueness	Global across systems	Local to database
Storage	16 bytes (larger indexes)	4-8 bytes (efficient)
Performance	Random inserts (can hurt B-tree)	Sequential (optimal for B-tree)
Distribution	Generated anywhere	Centralized generation
Readability	Complex (debugging harder)	Simple and readable

When to Use UUIDs
```sql
CREATE TABLE distributed_orders (
    order_id UUID DEFAULT gen_random_uuid(),
    customer_id UUID NOT NULL,
    order_date TIMESTAMP DEFAULT now(),
    PRIMARY KEY (customer_id, order_id)  -- Composite with UUID
);
```

🌐 Perfect For:
- Microservices architectures
- Data synchronization between systems
- Offline-capable applications
- Security-sensitive ID requirements

When to Use Integers

```sql
CREATE TABLE warehouse_inventory (
    warehouse_id INT NOT NULL,
    product_id BIGINT NOT NULL,
    quantity_available INT,
    last_updated TIMESTAMP,
    PRIMARY KEY (warehouse_id, product_id)
);
```

🏭 Ideal For:
- Single database systems
- High-performance requirements
- Storage-sensitive applications
- Human-readable references needed

Real-World Applications

🏆 **Success Stories**

E-Commerce Platform
Challenge: 100M+ orders, slow query performance
Solution: Partitioned by `order_date`, composite key `(customer_id, order_id)`
Result: 95% faster customer order history queries

IoT Data Processing
Challenge: Billions of sensor readings, expensive storage
Solution: Partitioned by `(device_type, reading_date)`, optimized data types
Result: 60% storage reduction, 10x faster analytics queries

Multi-Tenant SaaS
Challenge: Data isolation, scalability across thousands of tenants
Solution: Composite keys with `tenant_id` prefix, partition by tenant groups
Result: Perfect data isolation, linear scaling to 50,000+ tenants

---

Implementation Examples

🛠️ **Step-by-Step Implementations**

Example 1: Time-Series Data Partitioning

```sql
-- Create parent table
CREATE TABLE sensor_data (
    sensor_id INT NOT NULL,
    measurement_time TIMESTAMP NOT NULL,
    temperature DECIMAL(5,2),
    humidity DECIMAL(5,2),
    PRIMARY KEY (sensor_id, measurement_time)
) PARTITION BY RANGE (measurement_time);

-- Create monthly partitions
CREATE TABLE sensor_data_2025_01 PARTITION OF sensor_data
FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');

CREATE TABLE sensor_data_2025_02 PARTITION OF sensor_data
FOR VALUES FROM ('2025-02-01') TO ('2025-03-01');

-- Optimized query (partition pruning enabled)
SELECT AVG(temperature)
FROM sensor_data
WHERE measurement_time >= '2025-01-15'
  AND measurement_time < '2025-01-16'
  AND sensor_id IN (101, 102, 103);
```

Example 2: Multi-Tenant Application Schema
```sql
-- Tenant-aware composite keys
CREATE TABLE tenant_projects (
    tenant_id UUID NOT NULL,
    project_id BIGINT NOT NULL,
    project_name VARCHAR(200) NOT NULL,
    created_at TIMESTAMP DEFAULT now(),
    status VARCHAR(20) DEFAULT 'active',
    PRIMARY KEY (tenant_id, project_id)
);

CREATE TABLE project_tasks (
    tenant_id UUID NOT NULL,
    project_id BIGINT NOT NULL,
    task_id BIGINT NOT NULL,
    task_title VARCHAR(300) NOT NULL,
    assigned_to UUID,
    due_date DATE,
```

```
    PRIMARY KEY (tenant_id, project_id, task_id),
    FOREIGN KEY (tenant_id, project_id)
        REFERENCES tenant_projects(tenant_id, project_id)
);

-- Efficient tenant-scoped queries
SELECT p.project_name, COUNT(t.task_id) as task_count
FROM tenant_projects p
LEFT JOIN project_tasks t USING (tenant_id, project_id)
WHERE p.tenant_id = $1  -- Partition pruning
GROUP BY p.tenant_id, p.project_id, p.project_name;
```

🎯 **Key Takeaways**

1. **Start with Query Patterns**: Design partitions and keys based on how you'll actually query the data
2. **Balance is Critical**: Neither too many small partitions nor too few large ones
3. **Stability Matters**: Choose partition keys that won't change over time
4. **Test Performance**: Always benchmark with realistic data volumes and query loads
5. **Monitor and Adjust**: Use database metrics to optimize partition strategies over time

---

📚 **Additional Resources**

- **AWS Glue Data Catalog**: Partition indexing and filtering
- **PostgreSQL Documentation**: Native partitioning features
- **MongoDB Sharding**: Distributed partition strategies
- **Apache Kafka**: Message partitioning best practices

This guide provides foundational knowledge for implementing high-performance database partitioning strategies. Always test recommendations with your specific use case and data characteristics.