Architecture of Large-Scale Systems

By Dr. Taghinezhad

Mail:
a0taghinezhad@gmail.com

https://ataghinezhad.github.io/

CHAPTER 4 Services and Data

Introduction to Service-Based Architecture

- As applications transition to service-based architectures, the placement of data and state becomes critical.
- Services are categorized as **Stateless** or **Stateful** based on whether they maintain internal data or state.

Stateless/ Stateful Services

Stateless:

- Services that **do not maintain any internal data** or state (session information between requests).
- All required data is passed in through requests.
- Each **request** is **independent** and contains all **necessary information**

Stateful :

- Maintain **session/state** information
- Remember **information** from **previous** requests
- Require **careful deployment** planning
- Need **state synchronization** mechanisms

Stateful Services

Definition:

- Maintain session/state information
- Remember information from previous requests
- Require careful deployment planning
- Need state synchronization mechanisms

Stateful Service Example: Shopping Cart Service

Python

```
class ShoppingCartService:
    def __init__(self):
        self.active_carts = {}# In-memory state

    def add_item(self, cart_id, item):
        if cart_id not in self.active_carts:
        self.active_carts[cart_id] = []
        self.active_carts[cart_id].append(item)

    def get_cart(self, cart_id):
        return self.active_carts.get(cart_id, [])
```

Stateless Services

- **Benefits:**
  - **Scalability:** Stateless nature allows easy horizontal and vertical scaling by adding more instances.
  - **Caching Capabilities:** Since the service does not store state, frontend caches can handle requests more efficiently.
  - **Flexibility:** Easier to deploy, maintain, and adjust resources dynamically.
- **Limitations:**
- Not every service can be designed statelessly. Because:
  - Real-Time Collaboration Requirements:
    - Needs maintained:
      - Current document content
      - User presence information
      - Cursor positions
      - Edit history
      - Version control

Stateless Services

1. **Scalability**:

   Stateless Version:

   - Each request to /calculate-fare is completely independent
   - You can deploy unlimited instances since there's no shared state and use load balancers

   Stateful Problems**:** The StatefulAirlinePricing keeps user sessions, search history, and miles balances in memory

   - If you have multiple servers:
     - User searches on Server A, build up session data
     - Next request goes to Server B, which has no knowledge of their session
     - Would need complex session replication or sticky sessions

Stateless Services

2. Caching Capabilities:

- **Stateless Version:**
  - Identical requests always produce identical results
  - CDNs can cache common combinations
  - Multiple users can benefit from the same cached result
- **Stateful Problems:**
  - Can't effectively cache because:
    - Results depend on **session** state
    - Need to invalidate **cache** when session changes

Stateless Services

3. **Flexibility**:

- **Stateless** Version:
  - **Deploy new instances instantly**
  - **Update pricing logic** without worrying about existing sessions
  - **Scale down** instances during off-peak hours
  - Easy to test since inputs fully determine outputs
- **Stateful** Problems:
  - **Deployment** challenges:
    - Need to migrate existing sessions
    - Can't **easily shut down instances** with active sessions
    - **Memory leaks** from abandoned sessions
    - Complex testing due to state dependencies

Stateless Services

1. E-Commerce Platform

Stateless Service Example: Product Catalog Service

Python

```
@app.route('/products/<product_id>')
def get_product(product_id):
    # Each request is independent # No need to know about previous requests
    product = database.fetch_product(product_id)
    return product @app.route('/products/search')

def search_products(query):
    # Search request contains all needed parameters
    # No session or state needed
    results = database.search(query)
    return results
```

Stateful Services: Data localization

- A *stateful service* maintains internal state across requests: such as user session data, workflow progress, cached computations, or domain-specific records.
  - If this service must constantly fetch or update data stored far away (physically or across multiple services), it becomes slow and tightly coupled to other components.
- Data localization solves this by:
  - Keeping the relevant data inside the service boundary or in a data store optimized for that service.
  - Avoiding cross-service data ownership or “shared databases”.
  - Reducing round-trip latency and dependencies.

- **Without data localization (bad design)**
  - The **Order Service** stores no state. Every time an order is placed, it queries:
  - All (Inventory and Shipping and Customer) database directly

Stateful Services: Data localization

- A **stateful service** is a service that **remembers things** between requests. It keeps **internal data (state)** to know what's going on.
- So, every time a user or another service talks to it, it **knows the context** — it doesn't start from zero.
- If the stateful service needs to constantly **fetch or update its data from far away** (like another service's database), it becomes:
- **Slow** → because data must travel over the network each time.
- **Tightly coupled** → it depends directly on other services' data.
- **Hard to maintain** → if another service changes its database, this one can break.
- Data localization solves this by:
  - Keeping the relevant data inside the service boundary or in a data store optimized for that service.
  - Avoiding cross-service data ownership or "shared databases".
  - Reducing round-trip latency and dependencies.

Stateful Services: Data localization

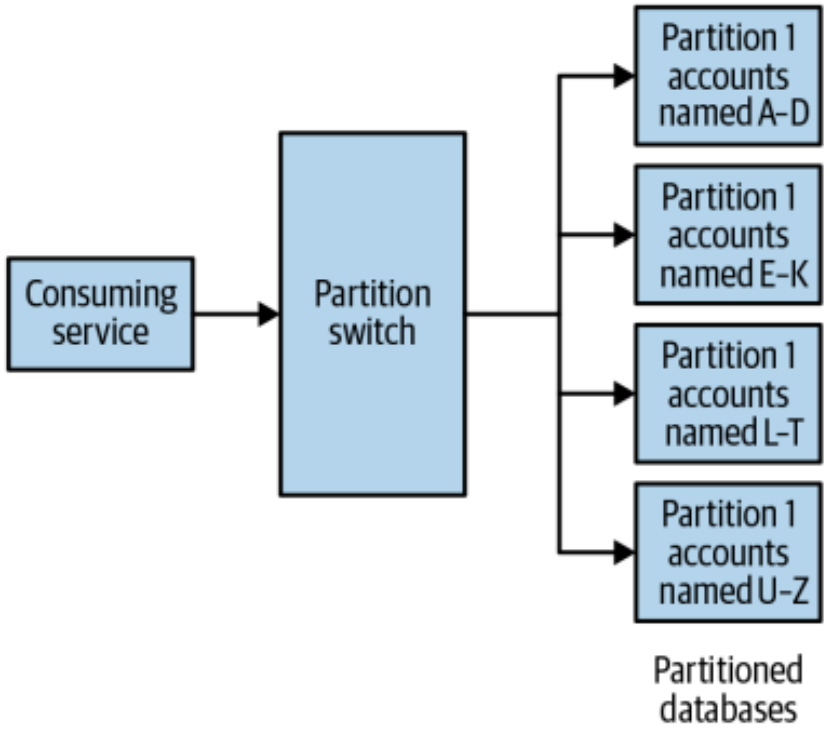
Localization

- When a service localizes its data (keeps the data inside its own boundary), it often becomes large enough that the service must **partition** that data to stay fast and scalable.
- But partitioning brings its own challenges.
  - These challenges matter because they can **cancel out** some of the performance benefits we gained from data localization if not handled correctly.

Data Partitioning

- **Definition:** It has other meanings, but in this context, partitioning divides large datasets into smaller segments to improve **access**, **scalability**, and **performance**.
- **Types of Data Partitioning:**
  - **Functional Partitioning:** Splits data by function, not size (e.g., orders, users, products).
  - **Key-Based Partitioning:** Uses a key (e.g., account ID) to distribute data across multiple partitions.

- **Example of Key-Based Partitioning:**
- Distributes data using a **key** (e.g., account ID).
  - Accounts A–D in one database, E–K in another, etc.
    - This approach allows better management of large datasets.



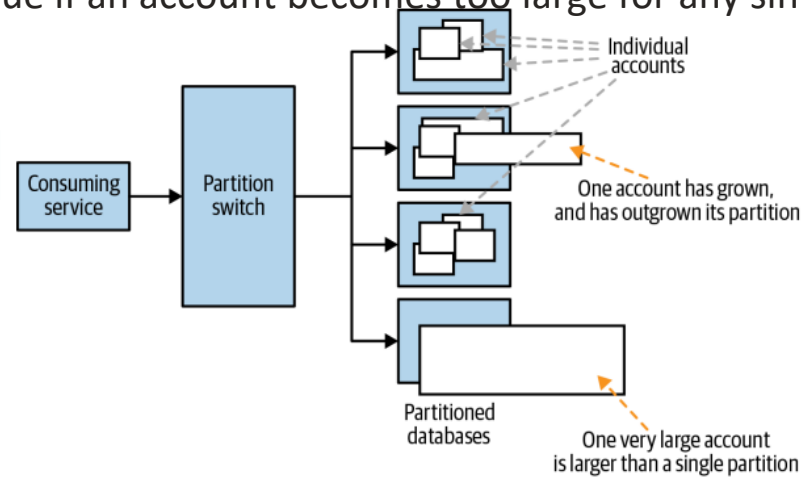
Challenges of Data Partitioning

1. **Increased Application Complexity:** Retrieving data becomes more complex, requiring knowledge of where the data resides.
2. **Cross-Partition Queries:** Analyzing data across multiple partitions becomes difficult.
3. **Skewed Partition Usage:** Poor partition key selection can lead to uneven load distribution.
    1. If you choose the wrong key, you can partitioning skew the usage of your database partitions, making some partitions run hotter and others colder, thus reducing the effectiveness of the partitioning while complicating your database management and maintenance.

Challenges of Data Partitioning

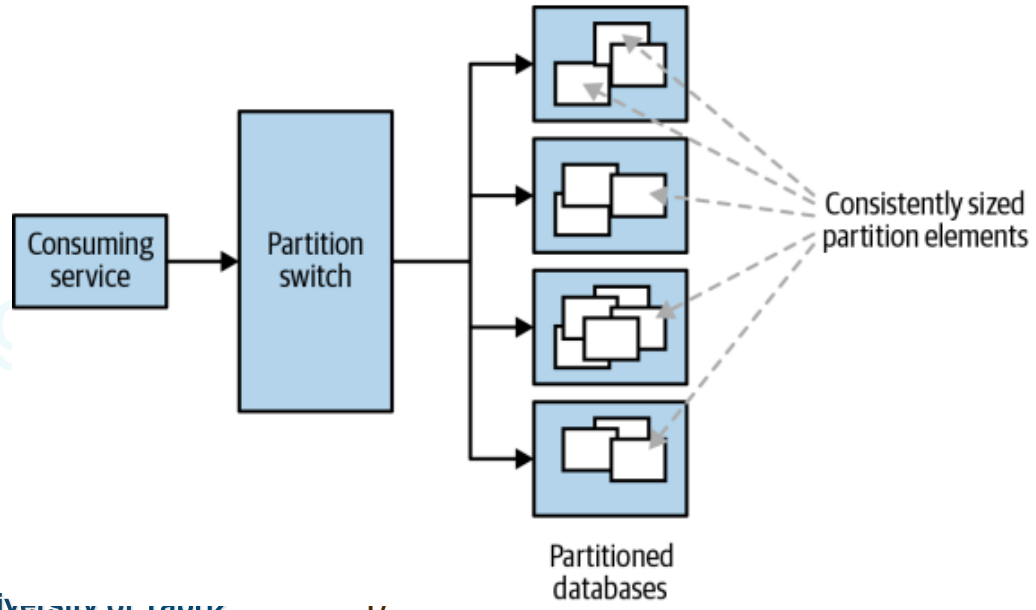
Avoid using account name or account ID as a partition key due to their potential to change i n size over time (Figure)

1. **Initial Size vs. Growth**:
    1. **Small Beginnings**: Accounts can start small and fit well within a partition.
    2. **Growth Over Time**: As accounts grow, they can overwhelm the partition they reside in.
    3. **Load Imbalance**: Growing accounts can disrupt the balance, requiring repartitioning
    4. **Size Limits**: A single large account can exceed the capacity of a single partition, causi ng partitioning schemes to fail.
2. **Repartitioning**:
    1. **Need for Adjustment**: Continuous growth necessitates repartitioning to maintain balance.
    2. **Limitation**: Even repartitioning can't solve the issue if an account becomes too large for any sin gle partition.



1. **Repartitioning Needs:**
   Rebalancing partitions is often complicated, especially when data grows unexpectedly.

- **Best Practice:**
  - Choose a partition key that ensures consistent partition size over time to minimize repartitioning.
  - Prefer many small elements mapped to larger partitions, allowing for easier adjustments.



Challenges of Data Partitioning

- **Best Practice: partition key that ensures consistent partition, Main Idea:**
  - Ensure consistent partition sizes over time to minimize the need for repartitioning by choosing appropriate partition keys and strategies.
  - **Scenario**: Managing user activity logs in a high-traffic web application.
  - **Better Partition Key Choice**: Instead of using user IDs as partition keys (which might vary greatly in activity volume over time), use timestamps to partition the logs.

1. **Consistent Partition Size**:
   1. **Example**: Partitioning logs by time intervals (e.g., daily or hourly) ensures each partition c overs a similar volume of data over time.
   2. **Result**: Balances load across partitions consistently, avoiding hotspots.

- -- SQL Table Definition for Partitioned Logs

```
CREATE TABLE user_activity_logs ( log_id SERIAL PRIMARY KEY, user_id INT, activity TEXT, log_time TIMESTAMP ) PARTITION BY RANGE (log_time); -

- Creating Partitions

CREATE TABLE user_activity_logs_2024_01 PARTITION OF user_activity_logs FOR VALUES FROM ('2024-01-01 00:00:00') TO ('2024-02-01 00:00:00');

CREATE TABLE user_activity_logs_2024_02 PARTITION OF user_activity_logs FOR VALUES FROM ('2024-02-01 00:00:00') TO ('2024-03-01 00:00:00');
```

Concept	Definition	Examples	Benefits
Stateless Services	No internal state	REST APIs, OAuth	Scalability, caching, flexibility
Stateful Services	Maintains internal state	Databases, shopping carts	Data localization, optimized access
Functional Partitioning	Split by function	Orders, users, products	Manageable datasets
Key-Based Partitioning	Split by key	Alphabetical partitioning	Improved scalability
Partitioning Challenges	Complexity, skewed usage	Social media data retrieval	Avoid overloading specific partitions

The Challenge of Growth in Modern Applications

- Growth is a natural aspect of any modern application.
- Areas of growth include:
  - **Traffic requirements** – Increased number of users or requests.
  - **Application size and complexity** – More features and interconnected modules.
  - **Team size** – More developers contributing to the codebase.

Ignoring Growing Pains: The Problem with Delayed Action

- Often, teams address scalability issues **only after problems become critical**. At this stage, solutions are limited, and easy remedies no longer work.
- Emergency responses may be more expensive, disruptive, or complex.
- **Consequences of ignoring scalability early:**
  - **Traffic spikes without preparation:**
    Example: Social media platforms crashing during viral events (e.g., a trending global news event).
  - **Increased technical debt:**
    Example: A platform forced to implement last-minute fixes, resulting in poorly designed code.
  - **Costly disruptions:**
    Example: A banking application failing due to unplanned growth, causing customer complaints and revenue loss.

Architectural Planning for Scalability

- **Architectural Lock-In:**
  - Without anticipating growth, teams make decisions that **constrain future scaling options**.
- **Proactive Planning:**
  - While **designing new applications or enhancing existing ones**, consider:
    - **How the application will grow** (users, features, data).
      - **Scalability walls:** What are the first barriers you'll encounter as the system grows?
      - **Mitigation strategies:** How will you address barriers without significant rearchitecting?

Key Considerations for Preemptive Planning

- **Capacity for Growth:**
  - Build room for **scaling** from the start to avoid being reactive later.
- **First Scalability Wall:**
  - Identify the **first constraint** (e.g., database capacity, server limits) the application will hit.
  - **Strategies to Overcome Barriers:**
    - Plan for **incremental improvements** rather than major architectural overhauls.

Benefits of Proactive Scalability Planning

- **Prevents costly disruptions:**
Anticipating growth allows smoother transitions during scaling phases.
- **Increases system flexibility:**
Applications designed with scalability in mind adapt more easily to change.
- **Supports business growth:**
Scaling bottlenecks can be removed **before** they become critical, enabling business continuity.

End Chapter 4