SQL Horizontal Autoscaler

Sunday, 5th October 2025

1. Introduction

1.1. Problem Statement

Traditional monolithic SQL databases are architecturally designed for vertical scaling (i.e., adding more CPU, RAM, or storage to a single server). This approach has physical limits and becomes prohibitively expensive. Horizontally scaling a traditional SQL database—distributing its load and data across multiple servers—is a complex challenge due to the relational nature of the data and the need to maintain transactional consistency.

1.2. Goals and Scope

This project aims to design and build a proxy-based system that enables horizontal scaling for a standard SQL database.

Goals:

- To create a database proxy layer that intelligently routes queries to the correct database shard.
- To distribute data across multiple database instances (shards) transparently to the client application.
- To implement a coordination service that monitors the health of the cluster and can trigger scaling operations.

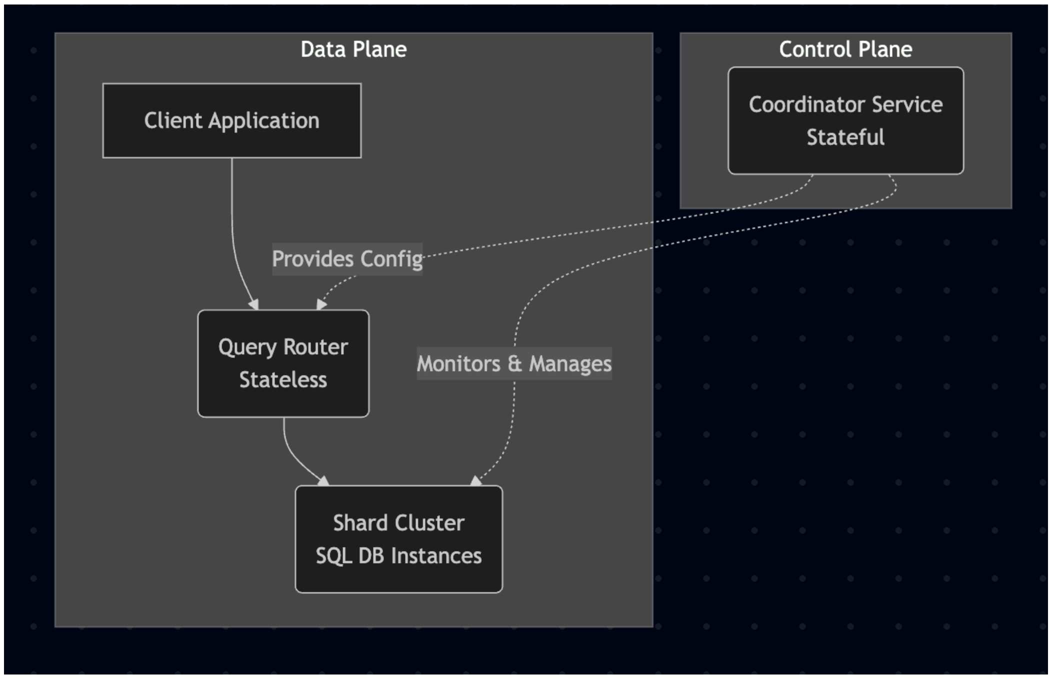
Out of Scope for v1.0:

- Support for distributed transactions that span across multiple shards.
- Automated failover and recovery of shards.
- Complex analytical queries that require joins across sharded tables.

2. High-Level Design (HLD)

The system consists of three core components: the **Query Router**, the **Coordinator Service**, and the **Shard Cluster**.

2.1. System Architecture Diagram



2.2. Component Responsibilities

- **Client Application:** The end-user service that needs to query the database. It interacts with the Query Router as if it were a single database instance.
- **Query Router (Proxy):** A stateless service that acts as the single entry point to the database cluster.
  - Receives SQL queries from the client.
  - Parses the query to identify the shard key.
  - Routes the query to the appropriate shard based on the sharding logic.
  - Performs scatter-gather operations for queries that do not contain a shard key.
- **Shard Cluster:** A collection of independent SQL database instances. Each instance, or "shard," holds a distinct subset of the total data.
- **Coordinator Service:** A stateful service that acts as the brain of the cluster.
  - Monitors the health, CPU, and disk utilization of each shard.
  - Maintains the authoritative map of the cluster's topology and sharding configuration.
  - Triggers and manages scaling operations based on pre-defined rules.

3. Low-Level Design (LLD)

3.1. Query Router API

The router exposes a single, simple endpoint for all database interactions.

- **Endpoint:** POST /query

- **Description:** Executes a SQL query against the distributed database.
- **Request Body:**
  ```
  {
   "query": "SELECT * FROM users WHERE user_id = 'user-123';"
  }
  ```
- **Success Response (200 OK):**
  ```
  {
   "status": "success",
   "data": [
    { "user_id": "user-123", "name": "Alice", "email": "alice@example.com" }
   ]
  }
  ```
- **Error Response (400 Bad Request / 500 Internal Server Error):**
  ```
  {
   "status": "error",
   "message": "Error details..."
  }
  ```

3.2. Coordinator Service API

The coordinator exposes administrative endpoints for monitoring and managing the cluster.

- **Endpoint:** GET /shards
- **Description:** Retrieves the status and metrics of all shards in the cluster.
- **Success Response (200 OK):**
  ```
  [
   {
    "id": "shard-1",
    "address": "10.0.1.10:3306",
    "status": "HEALTHY",
    "cpu_utilization_percent": 25,
    "disk_usage_gb": 500,
    "key_range_start": "0x0000",
    "key_range_end": "0x7FFF"
   },
   {
    "id": "shard-2",
    "address": "10.0.1.11:3306",
    "status": "HEALTHY",
    "cpu_utilization_percent": 30,
    "disk_usage_gb": 550,
  ```

```
    "key_range_start": "0x8000",
    "key_range_end": "0xFFFF"
  }
]
```

4. Scaling & Sharding Strategy

4.1. Sharding Method

The system will use **Hash-Based Sharding**. When a table is created, a specific column must be designated as the **shard key** (typically the primary key).

- **Smart Routing:** The Query Router will use a SQL parser library to analyze the WHERE clause of incoming queries.
    1. If the query contains the designated shard key, the router hashes the key's value to determine the correct shard. This is the most efficient query path.
    2. If the query does not contain the designated shard key, the router will perform a **scatter-gather** operation, sending the query to all shards and merging the results. This is an expensive operation and should be used sparingly.

4.2. Scaling Logic

The system will be configurable at startup to use one of two distinct scaling strategies, managed by the Coordinator.

- **Strategy 1: Ad-hoc Scaling (Hot Scaling)**
    - **Trigger:** The Coordinator monitors each shard's CPU and disk usage. If a shard exceeds a threshold (e.g., 80% CPU for 5 minutes), a scaling event is triggered.
    - **Action:** A single new shard is provisioned. A slice of the hash range from the overloaded shard is assigned to the new shard, and the corresponding data is migrated. This is a "live resharding" approach designed to minimize downtime.
- **Strategy 2: Doubling Strategy (Cold Scaling)**
    - **Trigger:** The Coordinator monitors the total number of entries across the *entire* cluster. When a threshold is met (e.g., num_shards * 1,000,000 entries), a scaling event is triggered.
    - **Action:** A new cluster with double the number of shards is provisioned. Data from the old cluster is migrated to the new one. This approach involves more downtime but results in a perfectly balanced cluster. It is suitable for applications that can tolerate maintenance windows.

5. Future Work

- **Distributed Transactions:** Implement a two-phase commit (2PC) protocol to support ACID transactions that span multiple shards.
- **Automated Failover:** Integrate the Coordinator with a consensus system like Raft to

automatically promote a replica if a primary shard fails.

- **Advanced Query Support:** Add support for cross-shard joins for analytical use cases.