Design Document: Distributed Task Scheduler

Objective

Define a mathematical formula to determine when the task scheduler application needs to scale up its instances to ensure tasks start within 10 seconds of their scheduled time, with a 10% buffer to avoid system overload.

Required Instances = ceiling(Total Required Compute Time / Buffer Percentage)

Front End

Task Management:

- **DataGrid:**
    - Displays a list of tasks with relevant details.
- **Task Creation and Editing:**
    - **Create Task:**
        - Users can create new tasks specifying if they are immediate tasks, recurring cron tasks, or scheduled for a specific time.
    - **Edit Task:**
        - Users can edit existing tasks, modifying details such as task type, schedule, and recurrence.

Monitoring:

- **Line Graph:**
    - Displays a real-time line graph to monitor system performance.
    - Indicates when it is necessary to scale based on task scheduling and system load.
    - Shows data for 3 hours into the future to predict required compute resources.

Metrics:

- **Task Metrics:**
    - Provides insights into task performance.

    - Displays the average execution time per task, helping in identifying performance bottlenecks and optimizing task execution.

Backend Systems

Task Scheduler System

Overview: Manages and executes tasks on a predefined schedule. Ensures tasks are executed on time and supports recurring tasks using cron expressions.

Functionality:

- **Database Connection and Transaction Management:**
    - Connects to the database using a query runner.
    - Starts a transaction to ensure atomic operations.
- **Time Calculation:**
    - Calculates the current time in UTC.
    - Determines the start and end of the current day in UTC.
- **Fetching Recurring Tasks:**
    - Retrieves active recurring tasks not yet scheduled for the current day.
- **Cron Expression Parsing:**
    - Determines next scheduled run times within the current day for each recurring task.
- **Task Scheduling:**
    - Inserts new scheduled times into the task_schedule table if they fall within the current day.
- **Transaction Commit and Rollback:**
    - Commits the transaction if all operations succeed.
    - Rolls back the transaction in case of errors to maintain data integrity.

Task Distributor System

Overview: Distributes scheduled tasks using RabbitMQ.

Components:

- **Task Distributor:** Distributes tasks based on their scheduled times.
- **RabbitMQ Integration:** Ensures reliable messaging between the Task Distributor and task processors.

Functionality:

- **Query Execution:** Fetches tasks due for execution with status 'Scheduled' and a scheduled_time <= current time.
- **Publishing to RabbitMQ:** Publishes tasks to RabbitMQ. Reverts task status to 'Scheduled' if publishing fails.
- **Transaction Management:** Uses transactions to ensure data consistency.

Task Processor System

Overview: Handles the execution of scheduled tasks by processing messages from a RabbitMQ queue.

Components:

- **RabbitMQ Integration:** Manages RabbitMQ connections and queue setup.
- **Processor:** Consumes messages from RabbitMQ and executes tasks.
- **Database Operations:** Updates task statuses in the database.
- **API Requests:** Makes external API requests to perform task-specific actions.

Functionality:

- **Message Consumption:** Consumes messages and processes tasks based on type (e.g., reminder, notification).
- **Task Execution:** Executes tasks by making necessary API requests and logs the output.
- **Error Handling:** Implements retry mechanisms for failed tasks, moving them to a dead-letter queue if retries fail.
- **Status Updates:** Updates task status in the database before and after execution.

API Endpoints

Create Task

- **Endpoint:** `/tasks`
- **Method:** POST
- **Description:** Allows the creation of new tasks, specifying details such as task type, schedule, and whether the task is recurring.
- **Details:** Accepts a JSON payload with task details including the type, schedule, and recurrence information. Returns the created task object.

Edit Task

- **Endpoint:** `/tasks/:taskId`
- **Method:** PUT
- **Description:** Enables editing of existing tasks. When a task is edited, all scheduled entries for that task are deleted. (separate scheduler service will repopulate the schedule within a min after deletion.
- **Details:** Accepts a JSON payload with updated task details. Deletes all scheduled tasks from the task_schedule table associated with the given task ID and repopulates the schedule with the updated task details.

Set Inactive

- **Endpoint:** `/tasks/:taskId/inactive`
- **Method:** PUT
- **Description:** Sets a task to inactive, stopping it from being scheduled.
- **Details:** Updates the task status to inactive in the database, preventing it from being included in future schedules.

Get Task Types

- **Endpoint:** `/task-types`
- **Method:** GET
- **Description:** Retrieves all task types.
- **Details:** Returns a list of available task types that can be used when creating or editing tasks.

Get Tasks

- **Endpoint:** `/tasks`
- **Method:** GET
- **Description:** Fetches all tasks with their schedules.

- **Details:** Returns a list of tasks, including details such as task type, schedule, and status.

Health Check

- **Endpoint:** `/health`
- **Method:** GET
- **Description:** Ensures the application is running correctly.
- **Details:** Returns a status indicating the health of the application.

Get Scheduled Tasks Summary

- **Endpoint:** `/scheduled-tasks-summary`
- **Method:** GET
- **Description:** Provides a summary of all scheduled tasks, including required compute time and instances.
- **Details:** Summarizes the number of tasks and required compute time for each scheduled time, then calculates the required instances based on the 10% buffer.

Get Task Summary

- **Endpoint:** `/task-summary`
- **Method:** GET
- **Description:** Retrieves the average elapsed time per task type for the past year.
- **Details:** Returns metrics on task performance, including average elapsed time for each task type.

Scaling Logic

Formula:

Required Instances = ceiling(Total Required Compute Time / (Max Compute Capacity per Instance × (1 - Buffer Percentage)))

Parameters:

- **Total Required Compute Time:** Sum of compute times for all scheduled tasks at a given time.
- **Max Compute Capacity per Instance:** 10,000 milliseconds.
- **Buffer Percentage:** 0.10.

Simplified Formula:

Required Instances = ceiling(Total Required Compute Time / 9000.0)

Data Views

- **average_elapsed_time_per_task_type:** Calculates the average elapsed time for each task type over the past year.
- **scheduled_tasks_summary:** Summarizes the number of tasks and required compute time for each scheduled time, then calculates the required instances based on the 10% buffer.

Monitoring and Metrics

- **Task Monitor:** Provides real-time visualization of scheduled tasks and instances using the LineGraph component, this component displays data for 3 hours into the future to predict required compute resources.
- **Metrics:** Offers insights into task performance, aiding in scaling and optimization decisions.

UI Considerations

- **Component Data Handling**: Each UI component is responsible for fetching its own data as needed, minimizing redundant API calls.
- **Delete Functionality**: When a task is deleted, it is marked as inactive, and any future schedules associated with it are deleted.
- **Edit Functionality**: When editing a task, future scheduled recurring jobs are deleted. However, the scheduler will resume new recurring jobs within 60 seconds or less.

Drawbacks and Bottlenecks

While scaling the processors is straightforward, potential bottlenecks may arise in the scheduling and distributing tasks. Scaling these components may require segmenting jobs to operate multiple instances effectively. Additionally, RabbitMQ, which handles messaging, could become a bottleneck. Replacing RabbitMQ with Kafka, known for its higher throughput, could mitigate this issue.

Potential Bottlenecks:

1. **Scheduling Tasks:**
    - **Challenges:** Distributing the scheduling load across multiple instances can be complex.
    - **Solutions:** Implementing job segmentation to ensure efficient operation and balance the load.
2. **Distributing Tasks:**
    - **Challenges:** Ensuring consistent task distribution without overloading a single instance and managing task dependencies and execution order when scaled.
    - **Solutions:** Design a mechanism for distributed task assignment that balances the load and respects task dependencies.
3. **RabbitMQ:**
    - **Challenges:** Limited throughput may hinder performance under heavy load.
    - **Solutions:** Potential replacement with Kafka for higher message processing speed and reliability.
4. **Database:**
    - **Challenges:** The database could become a bottleneck under heavy load, especially with high write and read operations.
    - **Solutions:** Implementing sharding to distribute the database load across multiple servers. Sharding involves partitioning the database into smaller, more manageable pieces, each hosted on a separate server to improve performance and scalability.
5. **UI Limitations**:

    - The UI currently lacks the ability to view task history and scheduled tasks effectively.
6. **Scheduling Challenges**:
    - **Predictive Scheduling**: There is a drawback in scheduling tasks for a single day into the future, scheduling further into the future would allow us to predict scaling needs further ahead.
    - The system needs improvements to handle immediate, recurring, and scheduled tasks efficiently.
    - Enhancing the ability to schedule tasks well in advance helps prepare for expected scaling requirements.
7. **Data Visualization**:
    - The line graph requires further development to tie in actual past data and provide users with the ability to select a date range.
    - Storing the number of current instances in the scheduling history would enable viewing past actual data in a chart, allowing comparisons with the estimated future task load.
8. **Configuration and Metrics**:
    - Current averages drop off if the containers are shut down, necessitating 24/7 operation for accurate metrics.
    - New UI features are needed to display additional metrics, such as average latency and long-running tasks.

Scaling Distributor/Scheduler

- **Workload Segmentation**:
    - One approach to scaling involves segmenting tasks so that specific workloads can be assigned to specific instances.
    - Alternatively, a granular approach could be used where tasks are processed carefully, one record at a time, ensuring transactional updates and avoiding duplication across concurrent instances.
    - Further development of these services will enable better concurrent operation and coordination of the scheduler and distributor.

Conclusion

This design ensures a robust, scalable, and efficient task scheduling and processing system. The independent and continuous repopulation of the schedule allows dynamic handling of task changes, while the scaling logic provides a buffer to ensure timely task execution and smooth operation. Despite potential bottlenecks in scheduling and distributing tasks, strategic segmentation and possible infrastructure upgrades (e.g., switching to Kafka) can mitigate these challenges.

By addressing these concerns, the system can maintain high performance and reliability, even as it scales to handle increased workloads.