Prompt to Recreate Task Queue Processing Solution (FIFO by Serial ID with Batches)

Objective

Create a complete solution for processing a task queue in PostgreSQL using Entity Framework Core (EF Core) in a C# application, deployed as an AWS Lambda function that calls an ASP.NET Core API. The solution must:

- Organize tasks into **defined batches** (e.g., A, B, C), with each Lambda instance (thread) exclusively owning one batch and processing all its tasks in **strict FIFO order** by a serial `Id` column (ascending, `ORDER BY "Id" ASC`).
- Ensure **exclusive batch ownership**: Multiple competing Lambda instances attempt to claim a batch, but only one can own it at a time.
- Process tasks within a batch one at a time using `SELECT FOR UPDATE SKIP LOCKED`, with **minimal reprocessing risk** (at most one task reprocessed on crash).
- Handle **short-lived tasks** (1–2 seconds processing time).
- **Delete completed tasks** instead of marking them as completed (existence in the `Tasks` table implies "Pending").
- Use **AWS Lambda** triggered every minute via CloudWatch Events, invoking an API (e.g., hosted on ECS or API Gateway) to process one task per invocation.
- Assume **idempotent tasks** or accept minimal reprocessing risk, logging completion to AWS CloudWatch instead of a `TaskLog` table.
- Follow C# best practices, including robust error handling, logging, and database connection management.
- Optimize for Lambda's stateless, serverless environment, handling concurrency, cold starts, and cost efficiency.

Requirements

1. **Database**:
   - A `Tasks` table with:
     - `Id` (SERIAL PRIMARY KEY, for FIFO ordering).
     - `BatchId` (VARCHAR, e.g., 'A', 'B', 'C').
   - A `BatchLocks` table to manage exclusive batch ownership.
2. **Batch Ownership**:
   - Use `SELECT FOR UPDATE SKIP LOCKED` on `BatchLocks` to ensure only one Lambda instance owns a batch.
   - If a batch is locked, try another batch or exit.
3. **Task Processing**:
   - Within a batch, dequeue one task using `SELECT FOR UPDATE SKIP LOCKED` with `ORDER BY "Id" ASC LIMIT 1`.
   - Process the task (1–2 seconds, simulated by `Task.Delay(1000)`).
   - Delete the task on completion.

   - On crash, rollback leaves the task in the table (implying "Pending") for reprocessing in FIFO order.
4. **Idempotency**: Log completion to CloudWatch; assume tasks are idempotent or accept minimal reprocessing risk (one task).
5. **Deployment**:
   - **Lambda**: Invokes every minute, sends a `POST` request to the API's `/process-task` endpoint with a preferred `BatchId` (e.g., 'A', 'B', 'C') and `WorkerId`.
   - **API**: ASP.NET Core API (e.g., on ECS) containing batch locking and task processing logic.
6. **Concurrency**: Handle multiple Lambda instances (e.g., 5–10) competing for batches, with `SKIP LOCKED` for both batch and task locking.
7. **Error Handling**: Retry transient errors (database, API) with exponential backoff, log to AWS CloudWatch.
8. **Performance**: Optimize with indexes and connection pooling.
9. **Cost**: Minimize Lambda costs for empty queues or locked batches.
10. **Batch Lock Cleanup**: Implement a timeout mechanism (e.g., 5 minutes) for stale locks.

Database Schema

```sql
CREATE TABLE "Tasks" (
    "Id" SERIAL PRIMARY KEY,
    "BatchId" VARCHAR(10) NOT NULL,
    CONSTRAINT check_batch_id CHECK ("BatchId" IN ('A', 'B', 'C'))
);

CREATE TABLE "BatchLocks" (
    "BatchId" VARCHAR(10) PRIMARY KEY,
    "LockedBy" VARCHAR(50),
    "LockedAt" TIMESTAMP WITH TIME ZONE NOT NULL,
    CONSTRAINT check_batch_id CHECK ("BatchId" IN ('A', 'B', 'C'))
);

CREATE INDEX idx_tasks_batchid_id ON "Tasks" ("BatchId", "Id" ASC);
CREATE INDEX idx_batchlocks_batchid ON "BatchLocks" ("BatchId");
```

C# Code

API Code (ASP.NET Core)

TaskEntity.cs

```csharp
public class TaskEntity
{
    public int Id { get; set; }
    public string BatchId { get; set; } // 'A', 'B', 'C'
}
```

BatchLockEntity.cs

```
public class BatchLockEntity
{
    public string BatchId { get; set; }
    public string LockedBy { get; set; }
    public DateTime LockedAt { get; set; }
}
```

MyDbContext.cs

```
using Microsoft.EntityFrameworkCore;

public class MyDbContext : DbContext
{
    public DbSet<TaskEntity> Tasks { get; set; }
    public DbSet<BatchLockEntity> BatchLocks { get; set; }

    public MyDbContext(DbContextOptions<MyDbContext> options) :
base(options) { }
}
```

TaskQueueProcessor.cs

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System;
using System.Threading.Tasks;

public class TaskQueueProcessor
{
    private readonly MyDbContext _context;
    private readonly ILogger<TaskQueueProcessor> _logger;

    public TaskQueueProcessor(MyDbContext context,
ILogger<TaskQueueProcessor> logger)
    {
        _context = context;
        _logger = logger;
    }

    public async Task ProcessQueueAsync(string preferredBatchId, string
workerId)
    {
        string acquiredBatchId = await
AcquireBatchLockAsync(preferredBatchId, workerId);
        if (acquiredBatchId == null)
        {
            _logger.LogInformation("No available batch for worker
{WorkerId}", workerId);
            return;
        }

        try
        {
            await ProcessSingleTaskAsync(acquiredBatchId, workerId);
        }
        finally
        {
            await ReleaseBatchLockAsync(acquiredBatchId, workerId);
        }
    }

    private async Task<string> AcquireBatchLockAsync(string
preferredBatchId, string workerId)
    {
        const int maxRetries = 3;
        int retryCount = 0;

        while (retryCount < maxRetries)
        {
            using var transaction = await
_context.Database.BeginTransactionAsync();
            try
            {
                var sql = @"
                    SELECT * FROM ""BatchLocks""
                    WHERE ""BatchId"" IN (@preferredBatchId, 'A', 'B',
'C')
                    ORDER BY CASE WHEN ""BatchId"" = @preferredBatchId
THEN 0 ELSE 1 END, ""BatchId""
                    LIMIT 1
                    FOR UPDATE SKIP LOCKED";
                var batchLock = await _context.BatchLocks
                    .FromSqlRaw(sql, new
Npgsql.NpgsqlParameter("@preferredBatchId", preferredBatchId))
                    .FirstOrDefaultAsync();

                if (batchLock != null)
                {
                    batchLock.LockedBy = workerId;
                    batchLock.LockedAt = DateTime.UtcNow;
                    await _context.SaveChangesAsync();
                    await transaction.CommitAsync();
                    _logger.LogInformation("Worker {WorkerId} acquired
batch {BatchId}", workerId, batchLock.BatchId);
                    return batchLock.BatchId;
                }

                retryCount++;
                if (retryCount < maxRetries)
                {
                    await Task.Delay(100 * (1 << retryCount));
                }
            }
            catch (Exception ex)
            {
                await transaction.RollbackAsync();
                retryCount++;
                _logger.LogWarning("Retry {RetryCount}/{MaxRetries} to
acquire batch {BatchId}: {Error}", retryCount, maxRetries,
preferredBatchId, ex.Message);
                if (retryCount >= maxRetries)
                {
                    _logger.LogError("Failed to acquire batch {BatchId}
after {MaxRetries} retries: {Error}", preferredBatch
```