ChatGPT

DynamoDB vs Amazon Aurora: comparação para consultas ordenáveis e paginação

1. Contexto

A captura de tela fornecida mostra uma lista de **boletos** com muitos campos (pagador, banco, CPF/CNPJ, beneficiário, número do documento, tipo de débito, situação, tipo de pagamento, data de vencimento, data limite para pagamento e valores). O usuário deseja permitir que **todas as colunas sejam ordenáveis** e que a API ofereça **paginação** eficiente para um grande volume de registros. A dúvida principal é:

- **DynamoDB** (NoSQL) seria uma boa opção para esse caso? Seria preciso criar um índice secundário (GSI) para cada coluna?
- **Amazon Aurora** (RDS relacional) seria melhor? Qual seria o impacto na performance e na paginação?

Para responder, precisamos entender como cada banco lida com ordenação, indexação, escalabilidade e paginação.

2. Características essenciais dos dois bancos

2.1 Amazon DynamoDB

- **Modelo chave-valor / documento** – DynamoDB armazena itens em tabelas sem esquema fixo. Cada item é identificado por uma chave primária que pode ser simples (partition key) ou composta (partition key + sort key). Os itens são distribuídos entre partições físicas de acordo com o valor da partition key.
- **Ordenação pela sort key** – dentro de cada partição, os itens são automaticamente ordenados pelo valor da sort key. O artigo sobre técnicas de ordenação em DynamoDB observa que o sort key é um atributo range que organiza os itens em cada partição; para obter ordem específica é necessário usar um **sort key bem definido** [1]. As consultas (`query`) retornam os itens na ordem ascendente da sort key [2]. Não é possível ordenar por outros atributos sem projetá-los na sort key ou usar um índice secundário.
- **Índices secundários** – para consultas por outros atributos, é possível criar **Global Secondary Indexes (GSI)** que definem uma nova combinação de chave (partition key e sort key) [3]. Cada GSI mantém uma cópia parcial da tabela em uma estrutura separada; as operações de escrita replicam os itens no índice de forma assíncrona [3]. No entanto, há limites (20 GSIs por tabela) e custos adicionais de provisionamento de capacidade e armazenamento. Além disso, um GSI só ordena itens por sua própria sort key, não por qualquer coluna arbitrária.
- **Consulta e paginação** – as operações `query` e `scan` podem limitar o número de itens retornados e fornecem um **cursor** (`LastEvaluatedKey`) para obter a próxima página de resultados. A paginação em DynamoDB é sempre baseada em cursor; não existe `offset` numérico. Para chegar à página N, é preciso percorrer as N-1 páginas anteriores sequencialmente. Esse método é eficiente para navegação sequencial, mas dificulta saltos arbitrários.

- **Escalabilidade e custo** – DynamoDB oferece escalabilidade automática com baixa latência para operações de leitura/escrita; armazena dados em múltiplas regiões e permite throughput muito alto [4]. Ele funciona em modo provisionado (capacidade fixa) ou on-demand (paga por uso) [4]. Porém, se mal modelado, pode tornar-se caro devido à necessidade de múltiplos GSIs e leituras/escaneamentos extras [5].

2.2 Amazon Aurora

- **Banco relacional compatível com MySQL/PostgreSQL** – Aurora é um mecanismo de banco de dados que roda sobre Amazon RDS e é compatível com MySQL e PostgreSQL [6]. Ele oferece desempenho maior que o MySQL/PostgreSQL padrão [7] e pode ser executado em modo provisionado ou serverless (com escalabilidade automática) [8]. O limite de armazenamento é 64 TB [9].
- **Ordenação e índices** – bancos relacionais usam **índices B-tree** que armazenam dados em ordem. O artigo *Use the Index Luke* explica que um índice fornece uma representação ordenada dos dados; quando a consulta contém `ORDER BY`, o otimizador pode usar o índice para evitar uma operação de ordenação cara [10]. Um `order by` utilizando índice permite retornar os primeiros resultados sem processar todo o conjunto [11]. Em bancos relacionais, é possível criar um índice para cada coluna de ordenação ou um índice composto para múltiplas colunas e utilizar `ORDER BY coluna1, coluna2` de forma eficiente.
- **Consulta e paginação** – o SQL oferece `ORDER BY`, `LIMIT` e `OFFSET` para ordenação e paginação. Com índices apropriados, a ordenação é feita de forma *pipeline* sem custo extra para as primeiras páginas [11]. No entanto, `OFFSET` grande pode degradar o desempenho pois o banco precisa descartar registros anteriores; boas práticas recomendam paginação do tipo *keyset* (usando o valor da última chave lida) para alto volume.
- **Escalabilidade e custo** – Aurora escala melhor que o RDS tradicional; pode ser configurada com réplicas de leitura. O modo serverless escala automaticamente e requer pouca manutenção [8] [12]. Comparado ao DynamoDB, Aurora oferece armazenamento limitado (64 TB) e custos associados à instância e ao volume; porém, oferece mais flexibilidade para consultas complexas.

3. Comparação para o caso dos boletos

3.1 Ordenação por múltiplas colunas

DynamoDB:

- A base da ordenação é a sort key. A técnica de ordenar somente via sort key significa que você deve incluir no valor da sort key todos os campos que deseja ordenar (ex.: `dataVencimento#valor`, `banco#cpf`) [1]. Dessa forma, consultas com `begins_with` ou `between` podem percorrer as dívidas ordenadas por data ou valor.
- Para ordenar pelos demais campos (beneficiário, situação, tipo de débito etc.), seria necessário **criar GSIs distintos** com um sort key específico para cada coluna. Conforme citado, cada GSI cria uma "tabela auxiliar" que replica dados e possui seu próprio custo e capacidade [3]. A AWS limita o número de GSIs por tabela (até 20). Com muitas colunas ordenáveis, a modelagem se torna complexa e cara. Além disso, cada GSI precisa ser criado no momento do projeto; não é possível ordenar por uma coluna arbitrária sem índice.
- Ordenar atributos que mudam de valor (por exemplo, "situação" ou "tipo de pagamento") exigiria manutenção de vários índices e pode comprometer a consistência temporal dos dados.

Aurora:

- Qualquer coluna pode ser ordenada com `ORDER BY`. Para garantir desempenho, basta criar um índice em cada coluna que frequentemente aparece em `ORDER BY` ou um **índice composto** (por exemplo, `(dataVencimento, valor)` para ordenar por data e, em seguida, valor). Bancos relacionais suportam vários índices por tabela; a maioria das engines (MySQL, PostgreSQL) permite dezenas de índices, facilitando a ordenação dinâmica.
- Em casos em que todas as colunas da tabela são ordenáveis, pode-se criar índices de acordo com o acesso mais comum (por exemplo, data de vencimento e valores). Outras ordenações podem ser atendidas sem índice, com custo maior em CPU, mas ainda viável para volumes moderados. O otimizador pode escolher varrer a tabela inteira se isso for mais eficiente [13].
- Assim, Aurora oferece **flexibilidade** para ordenar por qualquer campo com sintaxe SQL padrão; não é necessário projetar chaves previamente como no DynamoDB.

3.2 Paginação e navegação

DynamoDB:

- A paginação é baseada em **cursor**. O resultado de `query` ou `scan` inclui o atributo `LastEvaluatedKey`; para obter a próxima página, é preciso passar esse valor na próxima requisição. Isso funciona bem para navegação sequencial (próxima página/voltar), mas **não** oferece acesso direto à página N. Para saltar, seria necessário percorrer as páginas anteriores, o que pode tornar a resposta lenta.
- Além disso, o volume de dados retornado por `query` é limitado a 1 MB; se os itens forem grandes, pode ser necessário realizar várias iterações para montar uma página. Filtros e projeções podem ajudar a reduzir a quantidade de dados.

Aurora:

- O banco relacional permite `LIMIT` e `OFFSET` para implementar paginação tradicional. Com um índice ordenado, as primeiras páginas são obtidas rapidamente [11]. No entanto, valores grandes de `OFFSET` fazem com que o banco percorra todas as linhas anteriores; para volumes grandes recomenda-se **paginação baseada em cursor** (`WHERE id > :lastId ORDER BY id LIMIT n`) similar à de DynamoDB, mas a sintaxe SQL facilita a implementação.
- frameworks como **Spring Data JPA** oferecem suporte integrado a paginação e ordenação: basta adicionar um parâmetro `Pageable` ao método do repositório e o Spring gera a consulta SQL com `ORDER BY` e `LIMIT`. A mesma API permite especificar múltiplos campos de ordenação dinamicamente.

3.3 Custo, escalabilidade e manutenção

- **Desempenho:** DynamoDB oferece latência de milissegundos para acessos baseados em chave, ideal para sistemas de altíssima escala ou workloads imprevisíveis. Aurora é otimizada para consultas relacionais e pode ter latência levemente maior, mas ainda é adequada para a maioria das aplicações empresariais.
- **Escalabilidade:** DynamoDB escala de forma transparente e suporta tráfego praticamente ilimitado [4]. Aurora escala até 64 TB de dados [9] e tem opções serverless com escalabilidade automática [8]. Para cargas moderadas (como consultas de boletos), ambos atendem bem; para volumes massivos de leitura e escrita, DynamoDB pode levar vantagem.

- **Custo e manutenção:** DynamoDB segue modelo pay-per-request ou provisionado; mal dimensionado pode ficar caro [5]. Aurora tem custos de instância e armazenamento, mas oferece transações ACID completas, suporte a SQL e menos complexidade na modelagem de dados.

4. Spring Boot e facilidades

4.1 Utilizando DynamoDB

- No Spring Boot, é possível utilizar o **spring-cloud-aws-starter-dynamodb** ou o **DynamoDbEnhancedClient** do SDK v2. As operações de consulta podem receber parâmetros `scanIndexForward` para ordenar ascendente/descendente, mas apenas pela sort key. Para cada GSI criado, é necessário alterar a consulta para apontar o `IndexName` correspondente.
- Não existe recurso no Spring para ordenar dinamicamente por atributos arbitrários; a ordenação precisa ser planejada na modelagem (sort key composta ou GSIs). A paginação é gerenciada pelo cliente via `LastEvaluatedKey`.

4.2 Utilizando Aurora

- Spring Boot oferece **Spring Data JPA** para acesso relacional. Os repositórios do Spring aceitam um objeto `Pageable` que contém informações de paginação e ordenação. Por exemplo:

```
@GetMapping
public Page<Boleto> listarBoletos(Pageable pageable) {
    return boletoRepository.findAll(pageable);
}
```

O framework gera a consulta SQL com `ORDER BY` e `LIMIT` de acordo com os parâmetros de página e ordenação. Basta adicionar índices nas colunas com maior uso em ordenações para otimizar o desempenho. * Aurora suporta SQL avançado, transações ACID, integridade referencial e filtros complexos. Se a aplicação requer múltiplos filtros combinados (por exemplo, localizar boletos por CPF e ordenar por data de vencimento e valor), a abordagem relacional simplifica a implementação.

5. Conclusões e recomendações

Aspecto	DynamoDB	Aurora
Ordenação	Apenas por sort key do item; outras colunas exigem GSIs ou sort key composta [1].	Permite `ORDER BY` em qualquer coluna; índices B-tree armazenam dados em ordem e evitam operações de sort [10].
Número de colunas ordenáveis	Limitado pela quantidade de GSIs (máx. 20). Criar um GSI por coluna aumenta custo e complexidade [3].	Índices podem ser criados em várias colunas ou combinados; é possível ordenar por qualquer campo e combinar ordenação múltipla.

Aspecto	DynamoDB	Aurora
Paginação	Baseada em cursor; navegação sequencial eficiente, porém saltar para páginas distantes exige percorrer páginas anteriores.	`LIMIT` / `OFFSET` e paginação integrada via Spring Data; saltos podem degradar performance em offsets altos; keyset pagination é recomendável.
Escalabilidade	Muito alta, baixa latência para consultas por chave [4].	Alta para workloads relacionais; escalável com réplicas de leitura.
Modelo de dados	Requer design cuidadoso (single-table, sort key composta, GSIs). Pouco flexível para queries ad-hoc.	Flexível; suporte a SQL, joins, filtros complexos.
Custos e manutenção	Cobrança por requisição/ capacidade; múltiplos GSIs aumentam custo [5]. Baixa manutenção.	Custo de instância e armazenamento; manutenção mínima em modo serverless [12].

Quando escolher DynamoDB:

- A aplicação realiza leituras/escritas de alta taxa com padrões de acesso previsíveis e poucas necessidades de ordenação complexa.
- O conjunto de campos usados para ordenar é pequeno e pode ser modelado em sort keys ou poucos GSIs.
- É prioritário ter latência mínima e escalabilidade quase ilimitada.

Quando escolher Aurora:

- É necessário **ordenar e filtrar por várias colunas** de forma dinâmica, sem planejar todos os padrões antecipadamente.
- A aplicação usa **transações ACID**, consultas SQL complexas ou integrações com outras ferramentas que esperam SQL.
- A paginação deve permitir saltar entre páginas arbitrárias e aproveitar a facilidade de `ORDER BY` com índices.

No cenário dos boletos, onde todas as colunas podem ser ordenadas e os usuários podem aplicar diferentes critérios de ordenação e filtrar por diversos campos, **Aurora (ou outro banco relacional)** tende a ser mais apropriado. Ele permite criar índices para as colunas mais utilizadas e oferece flexibilidade na ordenação e paginação via SQL. O DynamoDB atende melhor a casos de acesso simples por chave; para ordenar por muitas colunas, seria necessário criar GSIs para cada atributo, o que aumenta custo e complexidade sem garantir a mesma flexibilidade.

6. Exemplos de implementação

Para ilustrar a diferença de complexidade entre DynamoDB e Aurora ao implementar ordenação e paginação, os exemplos a seguir demonstram consultas básicas em cada tecnologia utilizando Java e Spring.

6.1 DynamoDB: consulta e paginação por sort key

Suponha que temos uma tabela `Boletos` com `id` como partition key e um sort key composto `dataVencimento#valor`. O exemplo abaixo usa o **AWS SDK v2** para consultar todos os boletos de um pagador e ordená-los pela data de vencimento (ascendente) com limite de registros por página:

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;

public PageResult consultarBoletos(String pagadorId, String
lastEvaluatedKey) {
    DynamoDbClient client = DynamoDbClient.create();
    // Expression para chave: id = :id
    String keyCondition = "#pk = :id";
    Map<String,String> expressionNames = Map.of("#pk", "id");
    Map<String,AttributeValue> expressionValues = Map.of(":id",
AttributeValue.fromS(pagadorId));

    QueryRequest.Builder builder = QueryRequest.builder()
            .tableName("Boletos")
            .keyConditionExpression(keyCondition)
            .expressionAttributeNames(expressionNames)
            .expressionAttributeValues(expressionValues)
            .limit(20) // limita a 20 itens por página
            .scanIndexForward(true); // ordenação ascendente pela sort key

    // se houver cursor da página anterior, define o ExclusiveStartKey
    if (lastEvaluatedKey != null) {
        builder.exclusiveStartKey(Map.of(
                "id", AttributeValue.fromS(pagadorId),
                "sortKey", AttributeValue.fromS(lastEvaluatedKey)
        ));
    }

    QueryResponse response = client.query(builder.build());
    List<Map<String,AttributeValue>> items = response.items();
    String novoCursor = null;
    if (response.lastEvaluatedKey() != null && !
response.lastEvaluatedKey().isEmpty()) {
        novoCursor = response.lastEvaluatedKey().get("sortKey").s();
    }
    return new PageResult(items, novoCursor);
}

public record PageResult(List<Map<String,AttributeValue>> items, String
cursor) {}
```

Neste código:

- A consulta usa `keyConditionExpression` para selecionar todos os itens cujo `id` (partition key) corresponde ao pagador. Os itens são retornados na ordem do sort key (`dataVencimento#valor`) e limitados a 20 registros.
- O cursor `LastEvaluatedKey` é transformado em `cursor` (valor da sort key) para a próxima chamada. Para acessar a próxima página, basta passar esse valor como `lastEvaluatedKey`.
- Para ordenar por outro campo, seria necessário criar um GSI com esse campo como sort key e repetir a lógica com `indexName()` apontando para o novo índice.

6.2 DynamoDB: consulta usando GSI para ordenar por outra coluna

Se quisermos ordenar boletos por **beneficiário**, é preciso criar um **GSI** com `beneficiario` como partition key e, opcionalmente, `dataVencimento` como sort key. A consulta ficaria assim:

```
QueryRequest request = QueryRequest.builder()
        .tableName("Boletos")
        .indexName("BeneficiarioIndex") // nome do GSI criado para
beneficiário
        .keyConditionExpression("beneficiario = :nome")
        .expressionAttributeValues(Map.of(":nome",
AttributeValue.fromS("CHM SELVA DE FARIA")))
        .scanIndexForward(false) // ordena descendente pela sort key do
índice (dataVencimento)
        .limit(10)
        .exclusiveStartKey(cursorMap) // opcional
        .build();
QueryResponse response = client.query(request);
```

Cada coluna que necessitar ordenação exigirá outro índice, aumentando custo e manutenção. Além disso, a paginação continua sendo baseada no cursor retornado pela chamada.

6.3 Aurora: consulta e paginação via Spring Data JPA

Usando Aurora com Spring Data JPA, podemos aproveitar a funcionalidade nativa de ordenação e paginação. Primeiro, definimos a entidade `Boleto` e um repositório:

```
@Entity
public class Boleto {
    @Id
    private Long id;
    private String pagador;
    private String banco;
    private String beneficiario;
    private LocalDate dataVencimento;
    private BigDecimal valor;
    // demais campos
}
```

```
public interface BoletoRepository extends JpaRepository<Boleto, Long> {
    Page<Boleto> findByPagador(String pagador, Pageable pageable);
}
```

No controller, bastam poucos parâmetros para consultar e ordenar:

```
@GetMapping("/boletos")
public Page<Boleto> listarBoletos(
        @RequestParam String pagador,
        @RequestParam(defaultValue="0") int page,
        @RequestParam(defaultValue="20") int size,
        @RequestParam(defaultValue="dataVencimento,asc") String[] sort) {

    // converte parâmetros de sort em objeto Sort (pode ordenar por várias
colunas)
    List<Sort.Order> orders = new ArrayList<>();
    for (String ordem : sort) {
        String[] parts = ordem.split(",");
        orders.add(new Sort.Order(Sort.Direction.fromString(parts[1]),
parts[0]));
    }
    Pageable pageable = PageRequest.of(page, size, Sort.by(orders));
    return boletoRepository.findByPagador(pagador, pageable);
}
```

Vantagens dessa abordagem:

- O Spring cria automaticamente a consulta SQL com `WHERE pagador = ?` e adiciona `ORDER BY dataVencimento ASC, valor DESC` conforme o `Sort` informado.
- A paginação é implementada pelo banco com `LIMIT` / `OFFSET` e o objeto `Page<Boleto>` traz informações sobre total de registros, página atual e número de páginas.
- Para melhorar performance, basta criar índices nas colunas mais usadas (`pagador` , `dataVencimento` , `valor` , etc.). O próprio banco usa o índice ordenado para retornar os primeiros resultados sem precisar ordenar todo o conjunto [10] .

Em cenários mais complexos, podemos usar métodos derivados do Spring Data (`findByBeneficiarioContaining`) ou consultas com `@Query` para combinar filtros e ordenação. O código permanece conciso e a modelagem de dados não precisa ser ajustada para cada novo critério de ordenação.

---

[1] [2] Mastering Sorting Techniques in DynamoDB: From Chaos to Clarity - Best Software Development Team Extension Partner for Nordics
https://www.craftsmensoftware.com/mastering-sorting-techniques-in-dynamodb-from-chaos-to-clarity/

[3] DynamoDB Single Table Design - How to use a GSI? - DEV Community
https://dev.to/aws-builders/dynamodb-single-table-design-how-to-use-a-gsi-26eo

[4] [5] [6] [7] [8] [9] [12] RDS, Redshift, DynamoDB and Aurora Compared
https://www.justaftermidnight247.com/insights/rds-redshift-dynamodb-and-aurora-how-do-aws-managed-databases-compare/

[10] [11] [13] Effects of ORDER BY and GROUP BY on SQL performance
https://use-the-index-luke.com/sql/sorting-grouping