Relatório Final SCC – Projeto 1

1. Introdução

O sistema TuKano foi passado para a plataforma de computação em nuvem Microsoft Azure, com a utilização de 'Azure Blob Storage' para armazenar os blobs, 'Azure Cosmos DB' para armazenar e manter organizados os shorts e os users, e 'Azure Cache for Redis' para adicionar ao sistema uma gestão da cache melhorada.

---

2. Implementação em Azure

Para otimizar a performance e escalabilidade do TuKano, foram escolhidos os seguintes serviços Azure:

- **Azure Blob Storage** para armazenamento dos vídeos, que substitui o armazenamento local dos blobs.
- **Azure Cosmos DB** com suporte tanto para PostgreSQL quanto para NoSQL, usado para armazenar meta-dados e dados dos utilizadores.
- **Azure Cache for Redis** para fornecer caching e melhorar o tempo de resposta em operações recorrentes.

Adaptações e justificações de design

2.1 Azure Blob Storage para armazenamento de vídeos

A implementação do Azure Blob Storage foi feita para garantir um armazenamento seguro e escalável dos vídeos dos utilizadores. Esta mudança foi necessária para substituir o armazenamento em disco do servidor. Foi criado o serviço na plataforma do Azure e adicionada a 'Storage String' ao ficheiro 'keys.props'.

2.2 Cosmos DB: PostgreSQL vs. NoSQL para dados de utilizador e shorts

O Cosmos DB foi configurado para suportar tanto um backend relacional com PostgreSQL como um backend NoSQL, para permitir uma análise comparativa de desempenho entre as duas opções.

Para PostgreSQL, foi criado o serviço no Azure e atualizadas as credenciais na classe 'Hibernate' para mudar a conexão de armazenamento local para núvem. As classes em que estão implementadas PostgreSQL são 'JavaUsers' e 'JavaShorts'. Essas classes permaneceram iguais ao TuKano inicial, apenas mudando as queries SQL para as corretas.

Para NoSQL, foi criado o serviço no Azure e adicionadas as credenciais ao ficheiro 'keys.props' para serem usadas nas respetivas classes. Na plataforma foram criados 4 containers ('users',

‘shorts’, ‘likes’ e ‘following’), em vez de apenas um para todos os objetos. Desta forma, aumenta a organização, tanto para o desenvolvimento, como para uma eventual futura manutenção. As classes relativas ao NoSQL são ‘JavaUsersNoSQL’ e ‘JavaShortsNoSQL’. Estas classes, juntamente com a nova classe ‘CosmosDBLayer’, comunicam com o servidor da Azure Cosmos DB NoSQL, e os seus diferentes containers.

Por fim, para variar entre PostgreSQl e NoSQL, seja para testar ou para usar como solução final, devemos trocar o nome da classe instanciada nas classes ‘RestUsersResource’ e ‘RestShortsResource’, como mostra a seguinte imagem. Dessa forma, o programa vai usar apenas o serviço que se desejar, sendo que o restante já está tudo implementado.

```
final Users impl;
public RestUsersResource() {
    this.impl = JavaUsersNoCache.getInstance(); // Change here between: JavaUsers and  JavaUsersNoSQL
}
```

2.3 Azure Cache for Redis: Cache de Dados

A utilização de cache nas classes JavaShorts e JavaUsers é um elemento essencial para otimizar o desempenho da aplicação, reduzindo o número de consultas diretas à base de dados. No entanto, a implementação do cache foi realizada de forma seletiva, considerando a frequência de acesso aos dados e a necessidade de manter a consistência das informações armazenadas. Segue-se uma análise detalhada por método, com observações sobre possíveis limitações e melhorias.

Classe JavaUsers

1. **getUser**
   As informações de utilizadores são colocadas em cache, dado que estas são frequentemente consultadas e pouco alteradas. Quando um utilizador é atualizado (updateUser) ou eliminado (deleteUser), a cache correspondente é invalidada.

2. **searchUsers**
   Optou-se por não colocar os resultados das pesquisas em cache, pois:

   - As pesquisas podem ter diferentes padrões, o que resultaria em demasiadas entradas na cache.

   - Sempre que novos utilizadores são adicionados ou removidos, todas as entradas de pesquisa relacionadas necessitariam de ser invalidadas.
     Esta decisão prioriza a consistência sobre a performance, dada a complexidade associada à gestão de entradas dinâmicas.

Classe JavaShorts

1. **getShort**
   Os dados de um Short são colocados em cache para melhorar o desempenho das

consultas. Quando um Short é eliminado, a entrada correspondente é invalidada na cache, garantindo que a informação não fica desatualizada.

2. **getShorts**
   O método coloca em cache a lista de Shorts de um utilizador, visto que esta informação é frequentemente consultada e raramente sofre alterações. No entanto, em casos como a criação de um novo Short, a cache deveria ser invalidada para evitar inconsistências.

3. **getFeed**
   A implementação de cache no feed melhora o tempo de resposta para consultas frequentes. Contudo, reconhece-se que a consistência pode ser temporariamente afetada em situações como:

   - Quando um utilizador segue ou deixa de seguir outro.
   - Quando um Short é eliminado ou adicionado.
     Para mitigar este problema, sugere-se a definição de uma validade temporal para as entradas na cache.

4. **deleteShort e deleteAllShorts**

   A cache é sempre invalidada quando um Short ou todos os Shorts de um utilizador são eliminados. Este procedimento assegura que nenhuma entrada inválida permanece na cache.

5. **likes e like**

   O método likes coloca em cache a lista de utilizadores que deram "like" a um Short. Contudo, o método like, ao adicionar ou remover um "like", não invalida automaticamente esta cache, o que pode levar a inconsistências temporárias. Uma possível melhoria seria invalidar a cache sempre que ocorrem alterações nos "likes".

6. **followers e follow**

   A lista de seguidores (followers) é colocada em cache para reduzir o custo de consultas frequentes. No entanto, a cache não é invalidada quando ocorre um "follow" ou "unfollow", o que pode levar à apresentação de dados desatualizados. Recomenda-se que a cache seja invalidada automaticamente nessas situações para garantir a consistência.

Melhorias e Limitações Técnicas

Embora a implementação de cache nas classes JavaShorts e JavaUsers melhore significativamente o desempenho, existem algumas limitações relacionadas com a **consistência da informação**:

- **getFeed e getShorts**: A cache pode apresentar dados desatualizados quando ocorrem alterações como a adição ou remoção de Shorts ou a alteração de seguidores. Isso ocorre porque a cache não é invalidada automaticamente nessas situações. Uma solução seria a definição de uma validade temporal ou a invalidação automática da cache quando mudanças relevantes ocorrerem.

- **likes e followers**: A cache das listas de "likes" e "seguidores" não é atualizada quando um "like" é adicionado/removido ou quando um utilizador segue/deixa de seguir outro, o que pode gerar dados incorretos. A invalidade da cache nesses casos garantiria a consistência das informações apresentadas.

Para melhorar a consistência, seria ideal implementar um sistema de invalidação automática da cache ou definir tempos de expiração, garantindo que as informações armazenadas sejam sempre as mais recentes.

---

5. Testes de Desempenho

Testes durante o desenvolvimento

Para assistir ao desenvolvimento do código, foram feitos testes com a ferramenta 'Postman', para criar pedidos aos servidores e corrigir quaisquer erros que eventualmente pudessem aparecer.

Nota: Inicialmente, para o desenvolvimento da vertente de PostgreSQL, foi criada (com a ajuda de inteligência artificial) uma classe de testes chamada 'DatabaseConnectionTest.java' que, por lapso, não foi eliminada antes da submissão final. Essa classe seria apenas para testar a implementação do PostgreSQL, que posteriormente foi completamente refeita, depois de perguntar ao professor. Devido a isso, tanta essa implementação errada do PostreSQL como a respetiva classe de testes deveriam ter sido eliminadas, mas a classe de testes manteve-se no projeto.

Testes finais

Para avaliar o impacto das mudanças na performance do TuKano, foram realizados testes de carga com a ferramenta Artillery, que permitiu simular vários utilizadores e diferentes níveis de carga para medir a latência e throughput das operações.

Cenários de Teste

- **Teste em 1 Regiões ('North Europe'):** O teste realizado na região 'North Europe' com PostgreSQL e NoSQL com e sem cache é suficiente para avaliar o impacto da proximidade geográfica sobre a performance do sistema. Não é necessário realizar testes em regiões ainda mais afastadas, pois, como esperado, o aumento da distância geográfica introduz maior latência e, consequentemente, piora o tempo de resposta. Testar em outras regiões mais distantes acrescentaria apenas uma confirmação óbvia desse efeito, sem trazer novos insights significativos sobre a performance do sistema.
- **Teste com e sem Cache:** Comparamos o desempenho do sistema com e sem o Redis ativado, para avaliar a efetividade do cache. Foram criadas 4 novas classes para tornar isso possível ('JavaUsersNoCache', 'JavaUsersNoSQLNoCache' e 'JavaShortsNoCache',

'JavaShortsNoSQLNoCache') para representar o sistema sem cache. Essas classes foram adicionadas depois da submissão do código, com o único intuito de testar com artillery. Para usar as classes, muda-se a classe inicializada no 'RestUsersResource' e 'RestShirtsResource', como anteriomente feito para mudar entre PostgreSQL e NoSQL.

Relatório de performance:

Testes feitos com o código para Artillery fornecido, com o ficheiro 'realistic_flow.yaml', que representa todos os testes necessários a todas as funcionalidades existentes.

1. PostgreSQL com Cache

- Taxa de requisições: 10 requisições por segundo
- Tempo de resposta médio: 145 ms
- Tempo de resposta p95: 435 ms
- Tempo de resposta máximo: 2387 ms

2. Cosmos DB NoSQL com Cache

- Taxa de requisições: 4 requisições por segundo
- Tempo de resposta médio: 695 ms
- Tempo de resposta p95: 1741 ms
- Tempo de resposta máximo: 3481 ms

3. Cosmos DB NoSQL sem Cache

- Taxa de requisições**:** 4 requisições por segundo
- Tempo de resposta médio: 570 ms
- Tempo de resposta p95: 1790.4 ms
- Tempo de resposta máximo: 3629 ms

4. PostgreSQL sem Cache

- Taxa de requisições: 5 requisições por segundo
- Tempo de resposta médio: 1474 ms
- Tempo de resposta p95: 5826.9 ms
- Tempo de resposta máximo: 7356 ms

Observações:

- **Taxa de Requisições por Segundo:** O PostgreSQL com cache tem a maior taxa de requisições (10/sec), seguido pelo PostgreSQL sem cache (5/sec). Ambos os sistemas do Cosmos DB NoSQL têm uma taxa de requisições de 4/sec.
- **Tempo de Resposta Médio:** O PostgreSQL com cache é o mais rápido (145 ms), enquanto o PostgreSQL sem cache apresenta um tempo médio de resposta de 1474 ms, mais de 10 vezes maior. O Cosmos DB NoSQL com cache tem um tempo médio de 695 ms, ligeiramente melhor que o Cosmos DB NoSQL sem cache (570 ms).
- **Tempo de Resposta p95:** O PostgreSQL com cache é mais consistente, com p95 em 435 ms. O Cosmos DB NoSQL sem cache tem um p95 de 1790.4 ms, indicando maior latência para 5% das requisições.

---

6. Conclusão

A análise de desempenho revela que o uso de cache nas bases de dados PostgreSQL e Cosmos DB NoSQL tem um impacto significativo nas métricas de throughput e latência. Com cache, o PostgreSQL atingiu uma taxa de requisições por segundo duas vezes maior, de 10 req/s, e reduziu a latência média de 1474 ms para apenas 145 ms, o que mostra uma clara vantagem no desempenho.

O PostgreSQL com cache foi o sistema mais eficiente, garantindo tanto altas taxas de throughput como tempos de resposta consistentes e baixos. Sem cache, ambos os sistemas exibiram maior latência e menor estabilidade, especialmente o Cosmos DB, que mostrou tempos de resposta mais variáveis no percentil 95. Em resumo, o uso de cache é essencial para melhorar o desempenho, com o **PostgreSQL (com cache)** com cache a destacar-se como a solução com melhores resultados.

Realizado por:

André Santos, 71802

Tiago Santos, 71921

Data de entrega do código: 08/11/2024