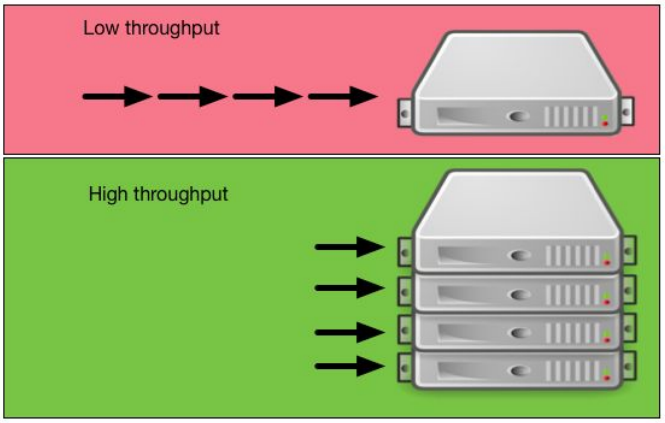
Giovani Nascimento Pereira - RA: 168609
Matheus Campanha Ferreira - RA: 174435
Vitor Kaoru Aoki - RA: 178474

Neo4j

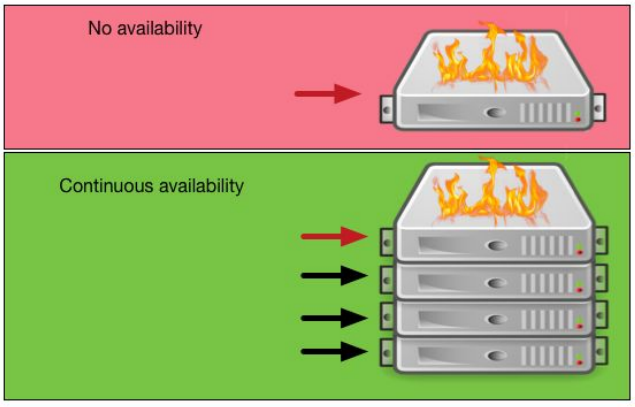
- Arquiteturas Possíveis:

O neo4j trabalha com uma arquitetura distribuída baseada em Clusters. Com essa abordagem, diversos problemas podem ser contornados, como por exemplo, uma taxa de transferência maior de dados, já que, com um sistema distribuído, os dados não vêm de uma única fonte, o que faz com que o congestionamento de dados seja menor e também a quantidade de fontes para se obter os dados seja maior, fazendo com que eles sejam obtidos com uma taxa de transferência maior.

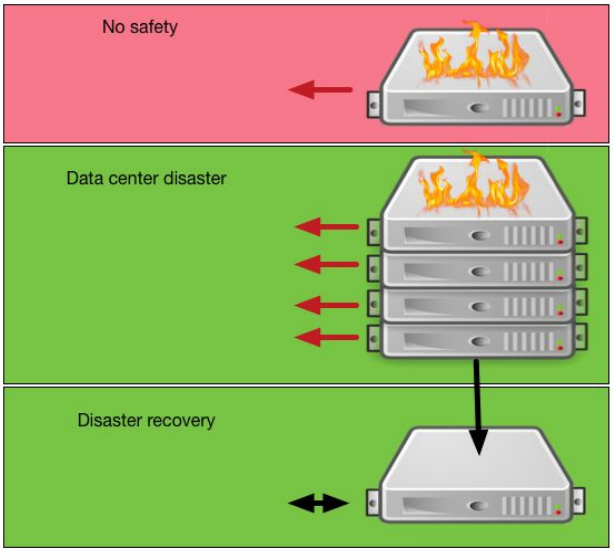
Além disso, por trabalhar com uma arquitetura distribuída e com redundância de dados, em caso de haver algum acidente com um dos clusters que ocasione o seu não funcionamento, o sistema distribuído permite que o banco de dados continue funcionando. Isso ocorre, pois, ao ser detectado o erro, o sistema se encarrega de buscar em outro cluster os dados requeridos, continuando assim, normalmente o funcionamento do banco. Ainda sim, o neo4j possui um sistema de recuperação dos dados do cluster perdido. Esse sistema de segurança se encarrega de recuperar os dados do cluster original, para que a manutenção possa ser realizada e ele não perca seus dados. Essas ferramentas trazem uma maior segurança para o banco, pois os dados estarão seguros e o usuário não perderá acesso ao banco em caso de falhas. Em contrapartida, para que esse sistema funcione bem, é preciso uma quantidade maior de memória para armazenamento dos dados, acarretando em um maior gasto com memória física e também um ótimo controle de sincronização dos clusters, para que funcionem corretamente como uma única máquina, o que demanda um bom sistema de controle e profissionais sempre atentos para qualquer eventualidade.



Comparação entre um sistema com um único servidor e um sistema distribuído baseado em clusters, onde é indicado a diferença na taxa de transferência de dados



Comparação entre um sistema com um único servidor e um sistema distribuído baseado em clusters, onde é mostrado que em caso de alguma falha, o sistema distribuído continua seu funcionamento

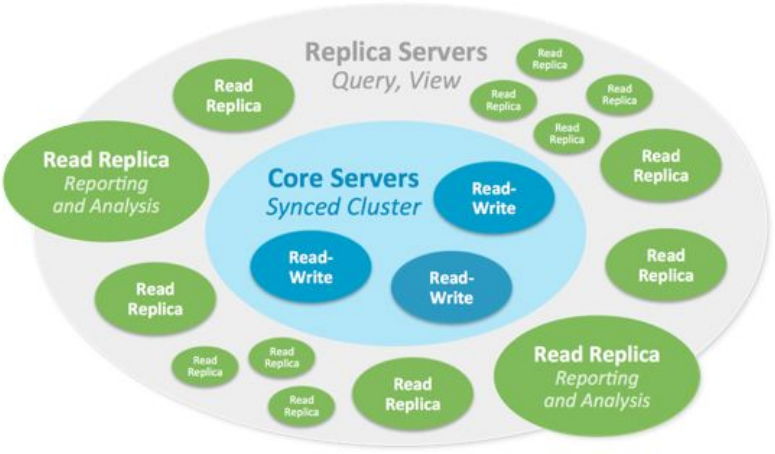


Comparação entre um sistema com um único servidor e um sistema distribuído baseado em clusters, onde é mostrado o sistema de recuperação de dados em caso de falha nos clusters

O neo4j trabalha com dois tipos diferentes de abordagens diferentes para a mesma arquitetura distribuída baseada em clusters:

- Causal Clusters:

Os Causal Clusters são baseados em um core, onde se encontram servidores com os dados armazenados e servidores periféricos, chamados de *Read Replicas* que funcionam basicamente como caches dos servidores do core.



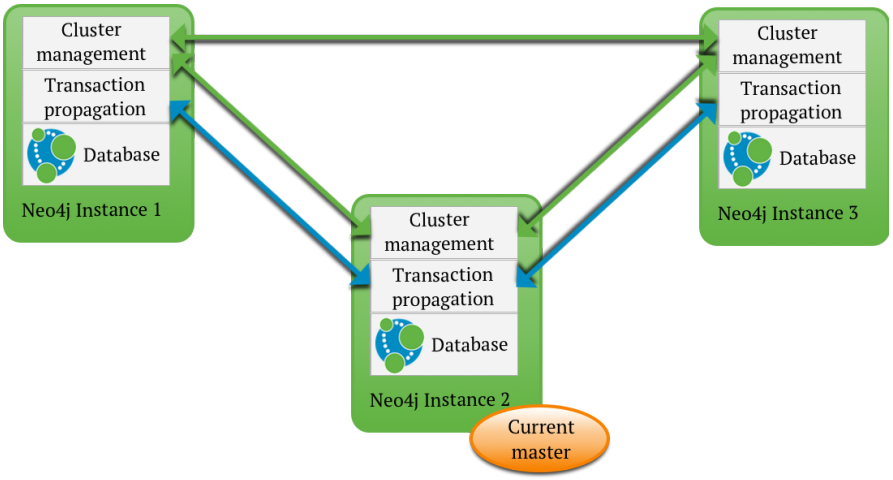
Uma representação de como funciona o Causal Cluster

Os servidores do core trabalham em um protocolo Raft, que garante a segurança de que as transações serão armazenadas em um número suficiente de servidores, a fim de que ao ser realizada uma nova transação, os dados estejam consistentes. Esses servidores do core armazenam os dados do banco e, como apresentam duplicata de dados, permitem uma maior segurança quanto a perda e inconsistência de dados, já que os dados possuem cópias. Porém, como mencionado acima, para isso, é necessário uma maior quantidade de memória física.

Os *Read Replicas* funcionam como caches dos servidores do core. Elas são também parte do banco de dados, mas podem realizar somente ações de leitura. Elas, ao receberem a transação, enviam-na para os servidores do core, para que o dado seja buscado, e retornam-o para o cliente que fez a requisição. Armazenando-o ao final, para caso seja feita uma nova busca por ele. Caso o dado buscado já esteja armazenado, ele é retornado diretamente sem ter que ser acessado no servidores do core. Isso facilita a busca por dados no grafo, de forma a aumentar a velocidade das transações e não sobrecarregarem os servidores do core, o que poderia causar problemas de mau funcionamento e lentidão.

- Highly Available Clusters:

Essa outra abordagem é representada por um sistema de mestre-escravo.



Representação do sistema mestre-escravo utilizado nos clusters

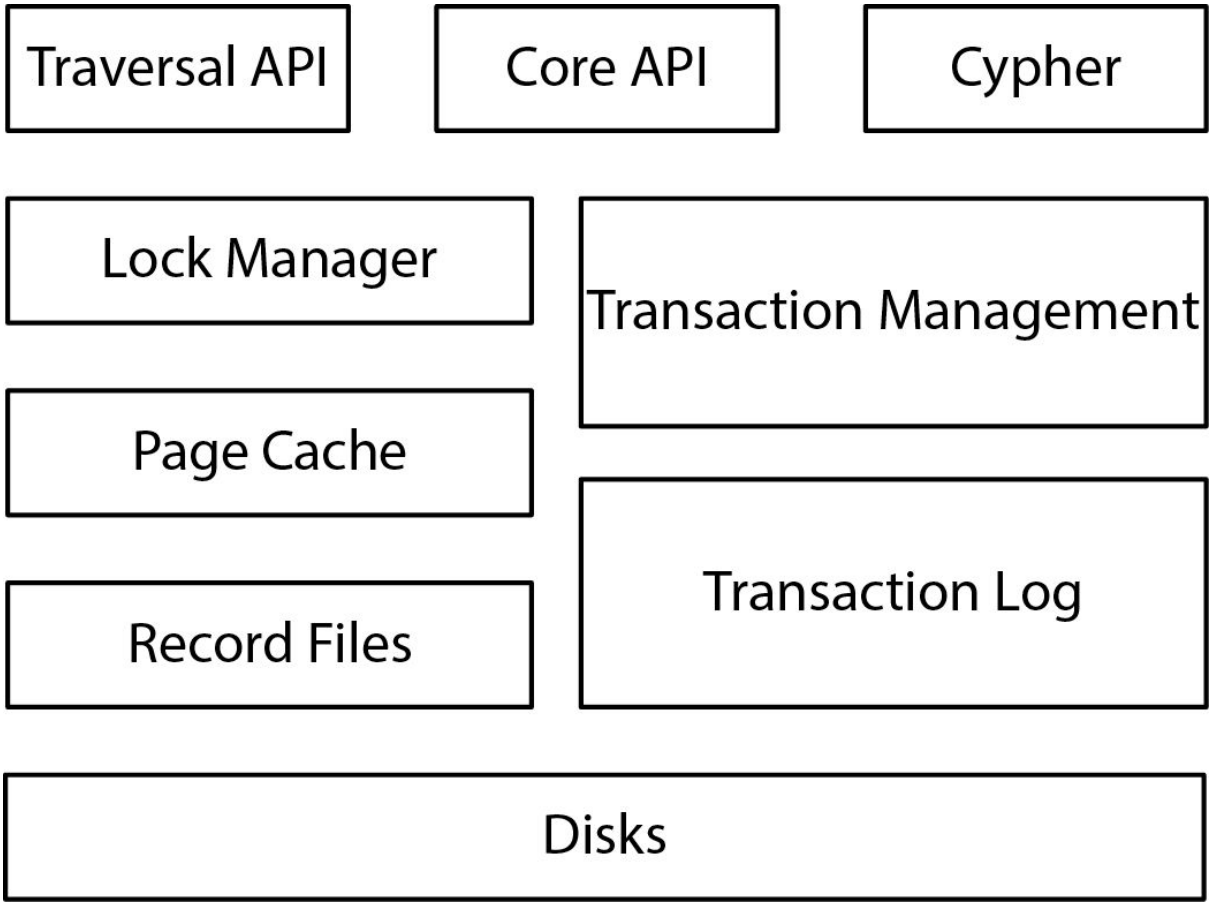
Nesse sistema, todos as instâncias do cluster possuem uma cópia dos dados. Quando uma transação de escrita é realizada na instância mestre e *commitada* com sucesso, as instâncias escravas são atualizadas, a fim de ficarem sincronizadas com a mestre. Quando uma transação de escrita é requerida em uma instância escrava, ela é sincronizada com a mestre e, caso seja *commitada* com sucesso, as outras instâncias escravas são atualizadas.

Esse sistema, de mestre-escravo com redundância de dados em todas as instâncias traz uma maior segurança dos dados. Com ele, os dados não correm o risco de serem perdidos e também, caso alguma falha ocorra como interrupção na conexão ou falha de hardware, o banco continua conectado, já que outras instâncias podem suprir a que deu defeito. Caso a instância defeituosa seja a mestre, o sistema possui um sistema inteligente de escolha de qual instância escrava se tornará a nova mestre, fazendo assim, com que o sistema não perca consistência e seja tolerante a falhas, já que a instância principal, que controla as demais, continuará operando para realizar as transações.

Esses mecanismos de segurança e otimização das consultas apresentam um contraponto, que assim com a outra abordagem (Causal Clusters) ele necessita de um espaço de armazenamento físico maior, para que todas as instâncias possuam uma cópia dos dados. Além disso, um bom controle entre as instâncias têm que ser implementado a fim de garantir a perfeita sincronia do cluster e fazer com o sistema continue consistente, além de necessitar para isso, um contínuo trabalho de monitoramento do banco.

- Native Graph Storage:

O neo4j é um banco de dados que além de usar grafos como armazenamento, apresenta sua arquitetura toda projetada para esse armazenamento em grafos. Essa característica vai desde seu armazenamento em disco, até suas camadas mais alto nível que contém suas API's e Cypher.

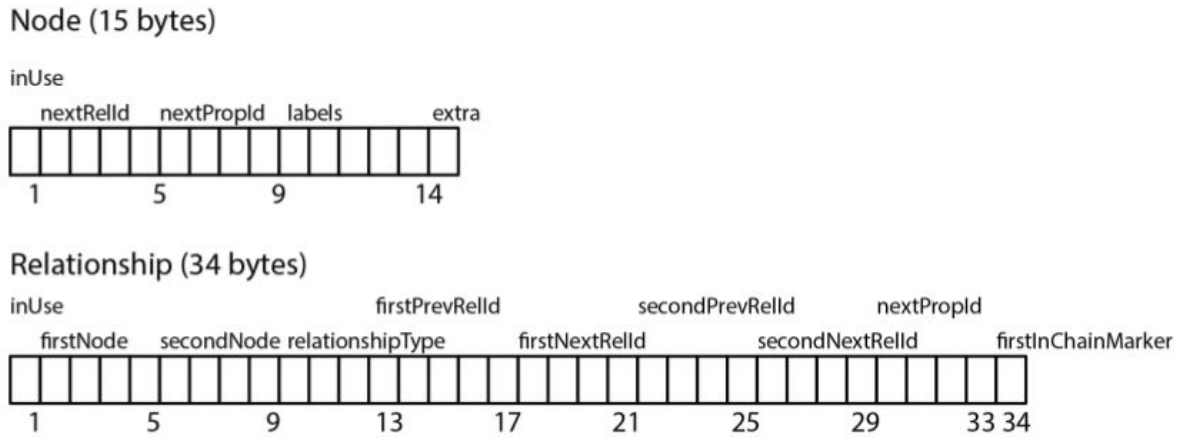


Arquitetura interna do neo4j

Por possuir sua arquitetura toda nativa em grafos, o neo4j apresenta uma vantagem em relação a bancos que são *non-native graph*, pois os bancos *non-native graph* apresentam problemas de desempenho, já que, como sua estrutura não é projetada para armazenar dados em grafos, ele precisa guardar todos os nós e arestas do grafo a todo momento e para uma busca simples ele precisa remontar todo o grafo. Assim, como o neo4j apresenta uma arquitetura voltada para armazenamento de grafos, os seus conjuntos da arquitetura, que são mostrados na imagem acima, são altamente interconectados, o que facilita todo o processo de busca e escrita. Porém, como essa arquitetura é toda voltada a grafos,

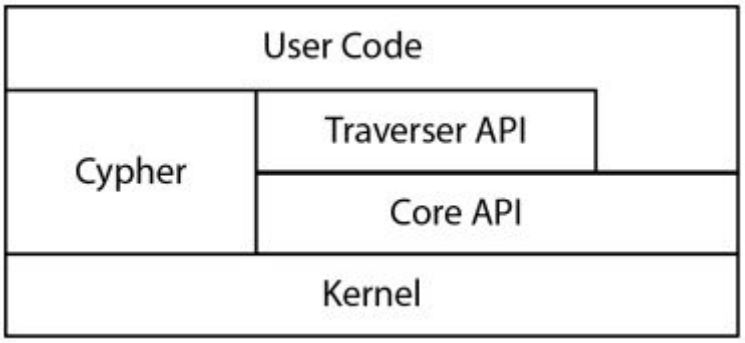
uma mudança no banco de dados, para um armazenamento que não é em grafos, se mostra muito difícil, já que toda a estrutura do banco deverá ser modificada.

Os dados em disco (*Disk*) são armazenados em diferentes arquivos de armazenamento. Cada arquivo de armazenamento guarda informações de diferentes partes do grafo, como nós, arestas, propriedades, etc.

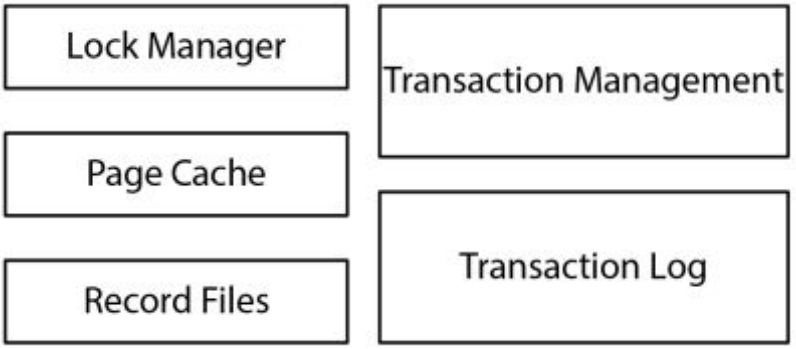


Estrutura dos arquivos de armazenamento de nós e arestas

A arquitetura, envolve ainda, a parte mais alto nível, onde se encontram as API's do sistema. Elas permitem ao usuário um fácil uso do bando e é nela que se encontra a linguagem de consulta Cypher. Além disso, essa camada mais alto nível possui um kernel que faz a tradução dela, com a camada que é responsável pelas transações, logs, controle de concorrência, todos os processamentos do sistema.



Camada onde se encontram as API's do sistema e o kernel

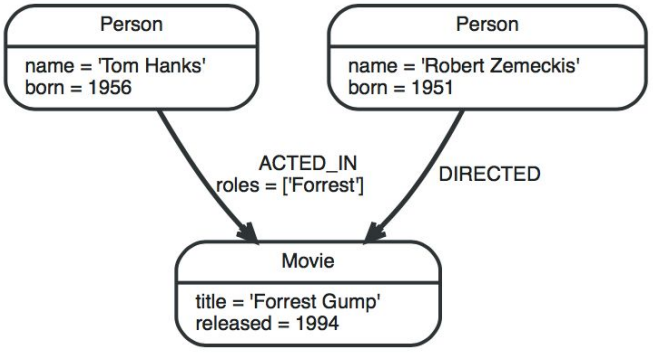


Camada onde ocorrem os processamentos do banco de dados

A disponibilidade dessas API's facilitam o uso do banco, já que é baseada em java, que é uma linguagem muito conhecida. Além disso, a presença de um kernel faz com que o usuário não tenha que se preocupar com a *linkagem* de seu programa em alto nível com o banco de dados diretamente. Por outro lado, isso faz com que o usuário não tenha um controle completo sobre o banco, já que está trabalhando sobre um banco que possui uma API criada pelo desenvolvedor do neo4j, que dá acesso ao usuário, somente as funcionalidades que este desenvolvedor deseja.

- Estruturas de Armazenamento:

O Neo4J é um banco de dados baseado em grafos, portanto ele armazena as informações na forma de grafos usando um modelo do *property graphs,* com *properties* e *labels*. Um *property graph* pode ser definido, da teoria de grafos, como um multigrafo direcionado com auto-arestas onde cada aresta e vértice tem seu próprio índice, um *label* é um token que identifica um nó único, um *property* é um par de *property key* (token que identifica unicamente uma propriedade) e *property value.*



Exemplo de um grafo com os atributos de cada nó

No exemplo acima, podemos notar que para o modelo, entidades são geralmente mapeadas para nós do grafo, e relações como as arestas (Dependendo das condições é comum relações possuírem nós) e os atributos são as propriedades (indicados como os "itens" dentro do grafo).

Outra parte importante são os labels (indicados na parte de cima de cada nó do grafo). Eles permitem agrupar os nós em um mesmo *set* (todos os nós com o mesmo label, pertencem ao mesmo set). Labels são criados para adicionar *constraints* ao modelo, ou definir melhor indexação para os dados (labels podem ser definidas e removidas em runtime) - por exemplo, se você quiser encontrar todos os "Person", você pode fazer essa consulta ao banco.

A utilização de grafos para a modelagem de dados fornece ao banco uma maior eficiência principalmente para dados que precisam ou contenham muitas relações. Hoje em dia, uma das coisas que mais vale para as grandes companhias é conseguir extrair *insights* baseados na forma como seus usuários, ou dados se comportam e relacionam. Muitas vezes, os relacionamentos (arestas) são mais importantes que os dados (nós) em si.

Com a arquitetura baseada em grafos garantem 3 principais benefícios:

- Performance:

Ao contrário de outras arquiteturas, que conforme a profundidade das relações o tamanho e complexidade das buscas aumenta num ritmo absurdo, num banco com modelo de grafos, essas buscas são feitas sempre em tempos constantes - mesmo com o crescimento dos dados ao longo dos anos.

- Flexibilidade:

Não tendo um modelo rígido, o grafo pode ser alterado assim como seus usuários precisarem, permitindo rápidas alterações em partes essenciais do modelo, e da organização dos dados, bem como adicionar novas relações, grupos, ou informações ao próprio modelo (ou remover).

- Agilidade:

Agilidade não só no uso do banco, mas também em relação a desenvolvimento ágil: Se for necessário fazer alterações durante o desenvolvimento, é muito fácil que isso seja feito, sem parar ou comprometer o desenvolvimento do sistema como um todo.

Mas existem alguns **problemas** que surgem quando usamos bancos de dados baseados em grafos:

- Não são apropriados para *informações transacionais* (como transações bancárias), onde os dados guardados são bem mais importantes que as relações em si
- Não é bom para consultas de agregação
- Não é bom se você tiver que fazer versionamento dos dados (*"apesar de ser possível, é meio estranho"*)

(Se você estiver considerando o uso no mundo real do Neo4j ou de um banco de dados baseado em grafos, muita referências também apontam como problema a imaturidade da tecnologia, a falta de suporte e de comunidade de uso e a dificuldade em ter uma forma padrão de se fazer consultas (como o SQL faz para o modelo relacional)).

- Indexação:

O neo4j utiliza um sistema de indexação baseado em índices que são cópias das informações do banco de dados. Esse sistema é adotado de forma a aumentar a eficiência do banco. Os índices do banco podem que podem ser criados, podem ter propriedades únicas, ou múltiplas propriedades. Em ambos os casos, o índice é criado para uma dada label, e a diferença entre ambos é que um pode possuir cópias para uma única propriedade do nó e o outro pode possuir mais do que uma propriedade.

Essa abordagem de cópias de dados para a indexação permite uma maior eficiência na busca de dados, já que para uma determinada query, como o índice já possui uma cópia do dado, não é necessária uma busca direta no banco de dados para a sua recuperação. Entretanto, essa abordagem faz com que seja necessário espaço adicional de armazenamento, pois muitas informações podem ser guardadas, e também faz com que o banco perca um pouco de eficiência na escrita, já que pode ser que muitas propriedades sejam selecionadas no índice, fazendo com que muitos dados sejam armazenados. Assim, para um bom funcionamento e eficiência do banco, é necessário uma atenção maior na escolha do que deve ou não ser indexado, o que é uma tarefa que, em bancos de dados mais complexos e com grande quantidade de dados, não é trivial.

O neo4j apresenta algumas funções para indexação, que tentam facilitar o trabalho com índices:

- Obter uma lista com todos os índices do banco:

Comando:

```
CALL db.indexes
```

Resultado:

```
+-----------------------------------------------------------------------+
| description                 | state    | type                         |
+-----------------------------------------------------------------------+
| "INDEX ON :Person(firstname)" | "ONLINE" | "node_label_property" |
+-----------------------------------------------------------------------+
1 row
```

- Criar um índice para uma única propriedade:

Comando:

```
CREATE INDEX ON :Person(firstname)
```

Resultado:
Nesse caso o retorno é nulo, pois o índice apenas foi criado.

```
+-------------------------------------------+
| No data returned, and nothing was changed. |
+-------------------------------------------+
```

- Uso de um índice de uma única propriedade com condicional *where* de igualdade:

Comando:

```
MATCH (person:Person)
WHERE person.firstname = 'Andres'
RETURN person
```

- Linguagem de Consulta e mecanismos de Processamento e Otimização de Consultas:

A linguagem de consultas usada pelo Neo4j é o **Cypher*,*** uma linguagem declarativa textual, que usa uma forma de *ASCII art* (olha só que legal) para representar os padrões de ligações dos grafos***.***

A ideia do Cypher, é permitir encontrar informações "navegando" mais facilmente através das relações através do que ele chama de *padrões*, e assim facilitar consultas que sejam muito complexas.

A linguagem usa parênteses para representar nós, e dentro dos parênteses, um nó pode ser identificado por um label, ou por uma variável do contexto. Um nó pode ainda ser mais especificado adicionando-se propriedades com o uso de chaves e uma lista de pares chave:valor.

```
(matrix:Movie {title: "The Matrix", released: 1997})
```

Para representar as relações, são usados um par de hífens (dashes) --.

Eles podem ser descritos colocando-se a direção da relação (-->) ou (<--), e colchetes para adicionar informações sobre a própria relação, como outras váriveis, e com o uso de chaves com pares de chave valor para descrever atributos dessa relação.

```
-[role:ACTED_IN {roles: ["Neo"]}]->
```

E os padrões para consulta são uma combinação dessas duas partes, da relação entre os nós e dos nós em que ela incide.

```
(keanu:Person:Actor {name:"Keanu
Reeves"})-[role:ACTED_IN{roles:["Neo"]}]->(matrix:Movie {title:"The Matrix"})
```

Cypher contém uma grande variedade de cláusulas, mas as mais comuns são MATCH e WHERE e RETURN - apesar de parecidas a ideia com SQL elas tem um funcionamento diferente:

MATCH - usado para descrever a estrutura de um padrão que estamos querendo encontrar, baseado primariamente nas relações.

WHERE - usado para adicionar constraints adicionais ao padrão.

RETURN - qual informação será retornada da consulta.

No caso da consulta em exemplo abaixo, ela deve retornar todos os filmes em que a atriz "Nicole Kidman" atuou (ACTED_IN), e, por conta da cláusula WHERE, filmes que vieram antes de uma determinada data passada como parâmetro.

```
MATCH (nicole:Actor {name: 'Nicole Kidman'})-[:ACTED_IN]->(movie:Movie)
WHERE movie.year < $yearParameter
RETURN movie
```

Exemplo de consulta feita usando Cypher

(Cypher não é uma linguagem padrão para bancos de dados baseados em grafos, mas isso vem se tornando verdade depois do projeto *OpenCypher* iniciado pela própria equipe do Neo4j em 2015.)

- Otimização de Consulta:

O Neo4j não tem muita coisa de otimização, ele é bem mais baseado que o programador seja responsável em implementar de maneira otimizada, e a documentação disponibiliza bastante coisa voltada para isso (dicas e boas práticas).

No release 2.2, eles introduziram um novo *Query Optimizer*, chamado **Ronja**. Ele atua otimizando queries de leitura apenas. A ordem de execução de uma query Cypher passa pelas seguintes etapas:

1. Converter o query string em um modelo árvore abstrato (AST - abstract syntax tree)
2. Otimizar e normalizar a AST
3. Criar um grafo de query da AST normalizada
4. Criar um plano lógico de X
5. Reescrever o plano lógico
6. Criar um plano de execução do plano lógico
7. Executar a query usando o plano de execução

O Ronja atua nas etapas de 2 a 5 desse processo.

2 - Otimizar e normalizar a AST:

Começa fazendo otimizações simples e normalizações na AST

- Mover labels de MATCH para WHERE
- Suprimir WITHs redundantes
- …

3 - Criar um grafo de query da AST normalizada

We use the normalized AST to create a query graph, which is a more abstract, high level representation of the query. Using the query graph instead of

operating directly on the AST allows us to compute costs and perform optimizations far more effectively, as we detail in the step below.

4 - Criar um plano lógico de X

Um plano lógico é uma árvore com no máximo 2 filhos (semelhantes a os usados por bancos de dados relacionais).

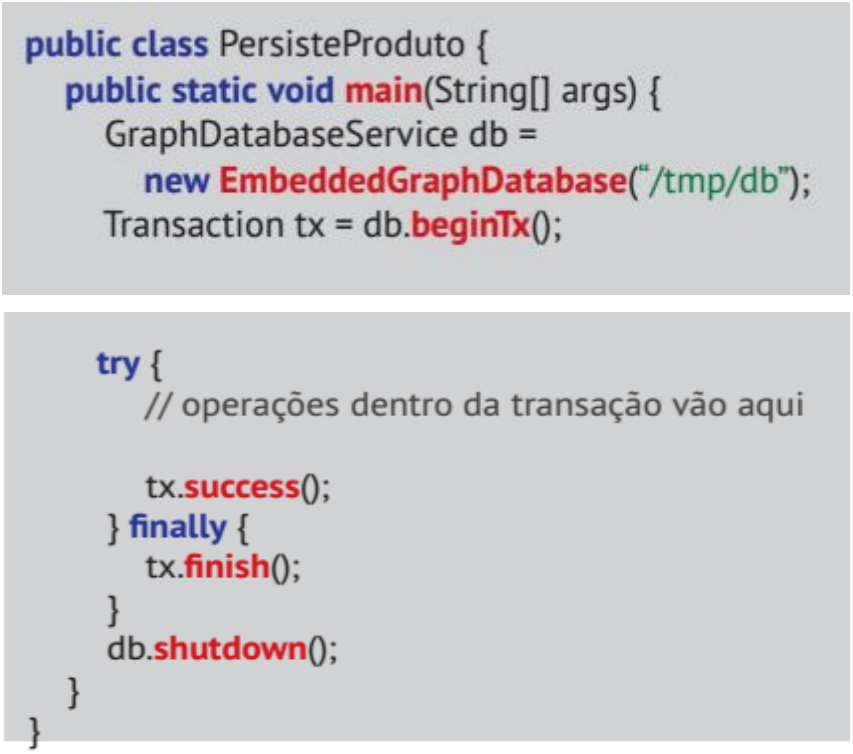
Em cada etapa, primeiro obtemos dados como seletividade de índice e rótulos dos nossos dados de estatísticas. Estes dados são então utilizados para estimar a cardinalidade - este é o número de linhas correspondentes - usando informações do gráfico de consulta. Com isso, podemos estimar um custo, que é usado para construir um plano lógico candidato.

5 - Reescrever o plano lógico

O plano lógico agora é reescrito usando não-aninhamento, fusão e simplificação de vários componentes; por exemplo. qualquer operador de igualdade transformado em 'IN's no Passo 2 é transformado de novo em operadores de igualdade. No final deste estágio, **Ronja completou seu objetivo**.

- Mecanismos de Processamento de Transações e Controle de Concorrência:

O SGBD neo4j possui o controle de concorrência baseado nas propriedades ACID. Com isso, ele permite a atomicidade do banco, a consistência, o isolamento e a durabilidade do mesmo. Para o controle de transações, é verificado dentro do Cypher se uma transação já existe, ou se uma nova deve ser criada. Caso uma transação já exista, a nova query (seja ela uma busca, ou uma escrita) será rodada dentro do dessa transação já existente e não será interrompida caso a transação não tenha sido *commitada*. Caso uma transação não exista, ela é criada, e a query roda dentro dessa transação, que é *commitada* apenas quando todas as queries forem finalizadas. Esse sistema permite que várias queries sejam *commitadas* como uma única transação.



Exemplo de criação de uma transação e de sua execução

Por seguir as propriedades ACID, o neo4j apresenta uma maior confiabilidade. Por conta delas é possível realizar transações de forma que seja possível manter a consistência dos dados, sem que uma interfira nas ações da outra. A sua ferramenta que permite que várias queries sejam executadas em uma mesma transação, permite uma execução mais rápida das queries, já que elas executam em uma mesma transação, sem que sejam criadas diversas transações, uma para cada query. Isso faz com que não seja perdido tempo por conta da atomicidade de transações. Dentro das transações, é permitido que várias queries sejam executadas, pois o sistema Cypher funciona de forma a realizar as queries

em partes. Na execução das queries as ações são realizadas de forma a ou buscar um elemento, ou executar uma escrita no banco.

Com isso, temos um ganho em relação ao tempo de execução de transações. Porém, para que se tenha uma segurança em relação a modificação do banco, para cada transação, as queries executadas, antes de serem *commitadas*, são armazenadas em memória e todas as modificações ficam armazenadas. Isso necessita de um espaço maior de memória, o que pode ser um problema para sistemas com pouca capacidade de memória.

Além disso, o neo4j apresenta outras 3 formas de transação, sendo elas:

- Auto-commit Transactions:

Essa forma de transação é a mais simples apresentada pelo neo4j e consiste apenas em uma transição que será *commitada* automaticamente assim que for finalizada com sucesso. Porém, caso ocorra um erro, ela não reinicia automaticamente, o que se apresenta como um problema, já que uma transação com alto grau de necessidade pode não ser realizada.

```
var records = new List<string>();
var session = Driver.Session();

try
{
    // Send cypher statement to the database.
    // The existing IStatementResult interface implements IEnumerable
    // and does not play well with asynchronous use cases. The replacement
    // IStatementResultCursor interface is returned from the RunAsync
    // family of methods instead and provides async capable methods.
    var reader = await session.RunAsync(
        "MATCH (p:Product) WHERE p.id = $id RETURN p.title", // Cypher statement
        new { id = 0 } // Parameters in the statement, if any
    );

    // Loop through the records asynchronously
    while (await reader.FetchAsync())
    {
        // Each current read in buffer can be reached via Current
        records.Add(reader.Current[0].ToString());
    }
}
finally
{
    // asynchronously close session
    await session.CloseAsync();
}
```

Exemplo de Auto-Commit Transaction

- Transaction Functions:

Diferentemente das Auto-commit Transactions, as Transactions Functions, possuem o mecanismo de reinicialização da transação caso algum erro seja detectado. Esse mecanismo dá mais segurança ao sistema no sentido de que, caso

alguma transação com uma alta importância tenha algum erro durante sua execução, o sistema tentará refazê-la.

```
var session = Driver.Session();

try
{
    // Wrap whole operation into an implicit transaction and
    // get the results back.
    result = await session.ReadTransactionAsync(async tx =>
    {
        var records = new List<string>();

        // Send cypher statement to the database
        var reader = await tx.RunAsync(
            "MATCH (p:Product) WHERE p.id = $id RETURN p.title", // Cypher statement
            new { id = 0 } // Parameters in the statement, if any
        );

        // Loop through the records asynchronously
        while (await reader.FetchAsync())
        {
            // Each current read in buffer can be reached via Current
            records.Add(reader.Current[0].ToString());
        }

        return records;
    });
}
finally
{
    // asynchronously close session
    await session.CloseAsync();
}
```

Exemplo de Transaction Functions

- Explicit Transactions:

As Explicits Transactions são as funções de transações básicas existentes. São elas os métodos *Begin*, *Commit*, *Rollback*, que devem ser utilizados pelo programador a fim de controlar a atomicidade e controle de concorrência das transações. Elas dão ao programador o poder de controle das transações, porém, por outro lado, devem ser utilizadas com cuidado, pois um erro em sua utilização pode fazer com que os programas não funcionem corretamente, ocasionando erros como *Deadlocks*, por exemplo.

```
var records = new List<string>();
var session = Driver.Session();

try
{
    // Start an explicit transaction
    var tx = await session.BeginTransactionAsync();

    // Send cypher statement to the database through the explicit
    // transaction acquired
    var reader = await tx.RunAsync(
        "MATCH (p:Product) WHERE p.id = $id RETURN p.title", // Cypher statement
        new { id = 0 } // Parameters in the statement, if any
    );

    // Loop through the records asynchronously
    while (await reader.FetchAsync())
    {
        // Each current read in buffer can be reached via Current
        records.Add(reader.Current[0].ToString());
    }

    // Commit the transaction
    await tx.CommitAsync();
}
finally
{
    // asynchronously close session
    await session.CloseAsync();
}
```

Exemplo de Explicit Transactions

- Segurança:

Para controlar a segurança do Neo4j, são implementados métodos de autenticação e autorização. O primeiro para verificar se o usuário é quem ele está falando e o segundo para verificar se determinado usuário tem permissão para fazer certas ações no banco de dados.

A autorização é gerenciada por meio de RBAC (*role-based access control*) e consiste em 4 autenticadores: *reader*, *publisher*, *architect* e *admin.* Sendo que cada um dos autenticadores possui determinadas permissões.

Além disso, o banco estudado possui, principalmente, 4 formas de autenticação e autorização: uma nativa do banco de dados, que salva o usuário e suas funções no disco local; LDAP e Kerberos, que também são outras maneiras de implementar o sistema de segurança, e plugin personalizado, ou seja, existe um plugin com a opção de implementar integrações personalizadas.

Action	reader	editor	publisher	architect	admin	(no role)
Change own password	X	X	X	X	X	X
View own details	X	X	X	X	X	X
Read data	X	X	X	X	X	
View own queries	X	X	X	X	X	
Terminate own queries	X	X	X	X	X	
Write/update/delete data		X	X	X	X	
Create new types of properties key			X	X	X	
Create new types of nodes labels			X	X	X	
Create new types of relationship types			X	X	X	
Create/drop index/constraint				X	X	
Create/delete user					X	
Change another user's password					X	
Assign/remove role to/from user					X	
Suspend/activate user					X	
View all users/roles					X	
View all roles for a user					X	
View all users for a role					X	
View all queries					X	
Terminate all queries					X	

Exemplo de autorizações no modo de segurança nativo do Neo4j

Atribuir funções e limitar autorizações para diferentes usuários faz com que o banco de dados seja facilmente administrado pela pessoa que gerencia o banco de uma determinada empresa. Sendo assim, alterações nos acessos dos usuários de cada função podem ser feitas sem problemas, o que contribui muito para o RBAC ser um sistema eficiente e de baixo custo de manutenção.

No entanto, o problema de um sistema RBAC é a complexidade inicial de implementação e configuração. O RBAC é melhor implementado aplicando uma estrutura detalhada e estruturada que divide cada tarefa em suas partes componentes. Dessa forma, para instalar um sistema RBAC em um sistema, é preciso muita experiência e cautela. Além disso, normalmente, nesses sistemas de segurança, é comum um problema de “explosão de funções”, já que no mundo real há cada vez mais funções aparecendo, porém isso não é um problema do Neo4j, visto que suas funções já foram previamente estabelecidas e criadas.

- Mecanismos de Recuperação:

A estratégia de backup deve ser projetada enquanto leva em consideração seus requisitos específicos. Esses requisitos podem incluir demandas de desempenho e tolerância para perda de tempo e perda de dados em caso de falha na rede ou no hardware. A estratégia de de backup utilizada pelo Neo4j baseia-se, novamente, em Causal Clusters.

- Read Replica Backups:

As *Read Replicas*, integrantes do Causal Clusters e explicadas com mais detalhes na parte de arquiteturas, ficam com a função de Backup do banco, visto que as *Replicas* são muito mais numerosas que os Cores. No entanto, isso faz com que mais memória física seja requisitada por parte do Neo4j.

No entanto, as *Read Replicas* são replicadas dos *Core Servers* assincronamente, o que é desvantajoso, pois dessa forma é possível que elas estejam atrasadas na aplicação de transações em relação ao cluster Core. Pode até ser possível que uma réplica de leitura se torne órfão de um Servidor Core, de modo que seus conteúdos sejam bastante obsoletos, ocupando memória que poderia estar livre.

Por outro lado, é possível verificar a última ID de transação processada em qualquer servidor e verificar que está suficientemente próximo da última ID de transação processada pelo Core Server. O que é muito vantajoso, pois em caso de alguma pane no sistema ou alguma fatalidade, podemos efetuar o backup seguro de nossa Read Replica com confiança de que esta está atualizada em relação aos Servidores Core.

- **Core Server Backups:**

Em um Core-Only server, não temos o privilégio de ter um grande número de réplicas de memória, sendo assim, não há *Read Replicas* para fazer este trabalho, desta forma, o cluster funcionará normalmente, mesmo quando forem realizados grandes backups. Por conta disso, o backup poderá sobrecarregar o servidor, o que pode afetar sua performance. Nesta linha, pode-se ver que um backup de *Read Replicas* é uma opção que, mesmo que envolva mais espaço físico de memória, fará com que o banco de dados seja mais eficiente.

Bibliografia:

- Mecanismos de Processamento de Transações e Controle de Concorrência: https://neo4j.com/docs/developer-manual/current/drivers/sessions-transactions/#driver-transactions-transaction-functions

https://neo4j.com/docs/developer-manual/current/cypher/introduction/transactions/

http://www.univale.com.br/unisite/mundo-j/artigos/51Neo4j.pdf

- Arquiteturas Possíveis:

http://neo4j.com/docs/operations-manual/current/clustering/

http://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/introduction/

http://neo4j.com/docs/operations-manual/current/clustering/high-availability/architecture/

https://neo4j.com/blog/native-vs-non-native-graph-technology/

Ian Robinson, Jim Webber e Emil Eifrem - Graph Databases

- Estruturas de Armazenamento

https://neo4j.com/why-graph-databases/

- Indexação:

https://neo4j.com/docs/developer-manual/current/cypher/schema/index/

- Ronja, otimização de consultas:

https://neo4j.com/blog/introducing-new-cypher-query-optimizer/

- Mecanismos de Recuperação e Segurança:

https://neo4j.com/blog/role-based-access-control-neo4j-enterprise/

https://www.computerworld.com/article/2573892/security0/how-role-based-access-control-can-provide-security-and-business-benefits.html

https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/backup-planning/