NoSQL

O que garantiu a hegemonia dos bancos de dados relacionais?

- **Armazenamento persistente de dados:** armazenar uma grande quantidade de dados em disco enquanto aplicações acessam estes dados e realizam buscas;
- **Controle de concorrência:** vários usuários podem ter acesso aos dados ao mesmo tempo graças ao controle de transações;
- **Padronização através do SQL**: utilização de uma linguagem padrão largamente conhecida que garante uma interface uniforme e independência de plataforma (até certo ponto);
- **Integração entre aplicações:** permite que aplicações distintas compartilhem informações de forma consistente e atualizada;
- **Relacionamento dos dados:** criação de forte relacionamentos entre os dados permitindo criar esquemas complexos.

Introdução

NoSQL é um recente paradigma de banco de dados que rompe com o tradicional modelo de banco de dados relacional. Ele visa dar mais flexibilidade a organização dos dados e, com isso, quebra com a rigidez de definição de dados que existe nos bancos de dados relacionais. Em outras palavras, a estruturação baseada em tabelas previamente definidas, simplesmente não existe. Além disso, devido a essa contraposição ao modelo relacional, esses bancos de dados NoSQL não possuem, como o próprio nome já diz, uma interface SQL. Nesses bancos, a interface para consultas varia de implementação para implementação.

O nome NoSQL surgiu pela primeira vez em 1988 e era o nome de um banco de dados relacional, porém que não possuía um interface SQL. O termo NoSQL foi adotado como nome para o movimento de banco de dados não relacionais em 2009 em um *workshop*. Entretanto existem discussões a respeito desse nome, já que o enfoque principal de bancos de dados NoSQL é a ausência do modelo relacional e não o fato de não existir uma interface SQL. Bancos de dados NoSQL não possuem uma interface SQL por não serem BDs relacionais, e não o contrário. Devido a isso, alguns defendem que o nome mais apropriado seria algo como NoREL.

É importante que se perceba que os BDs NoSQL não tem o objetivo de substituir o tradicional modelo relacional. Seu foco é prover uma alternativa para aplicações que naturalmente demandem um diferente modo de organização e distribuição dos dados. Existem aplicações em que essa natureza livre de esquema não funciona bem. Para esses e outros tipos de aplicações, os bancos de dados relacionais sempre serão uma boa opção.

Principais características

Não existem definições formais para BDs NoSQL, porém eles possuem um conjunto de características em comum:

- Não utilizam o modelo relacional e a linguagem SQL;
- Tendem a ser projetados para rodar em clusters;
- Tendem a ser open soucers;
- Não possuem esquemas fixos, permitindo que qualquer dado possa ser guardado em qualquer registro.

Outras características

- Baixa latência;
- Tolerância a falhas;
- Escalabilidade horizontal e dinâmica;
- Alta disponibilidade;
- Esquemas flexíveis;
- Suporte a distribuição geográfica;
- Baixo custo;
- Busca de dados através de heurística;
- Em geral abrem mão de: transações, *locks*, consistência, integridade e buscas complexas.

Aplicações

- Sistemas com grandes quantidades de dados;
- Arquitetura essencialmente distribuída;
- Aplicações em que buscas feitas com *joins* seriam inviáveis;
- Aplicações que possuem mudanças constante no esquema.

Prós e contras

Prós

- Alta disponibilidade;
- Escalabilidade horizontal;
- Baixo custo;
- Esquemas flexíveis;
- Dados distribuídos.

Contras

- Buscas dinâmicas limitadas;
- Consistência deve ser garantida pelo programador;
- Sem padrão (cada BD possui uma API diferente);
- Sem controle de acesso;
- Em geral não possuem um busca textual tão boa quanto bancos SQL. Não possuem um operador LIKE que permite essas buscas.

NoSQL e banco de dados relacionais

Os bancos de dados NoSQL surgiram principalmente para melhorar questões de escalabilidade que não são tão bem suportadas por BDs relacionais. Bancos de dados relacionais conseguem resolver de certa forma os problemas de escalabilidade, porém, em geral, a solução pode ser muito cara e complexa. A Web é uma das grandes motivadoras do movimento NoSQL. Isso porque as aplicações, além de passarem a ser distribuídas, também permitem um grande número de acessos simultâneos e isso faz com que o volume de dados trafegado seja muito grande.

Em troca de uma melhor escalabilidade e uma maior flexibilidade na organização dos dados, os bancos de dados NoSQL muitas vezes abrem mão de certas propriedades típicas de BDs relacionais, como propriedades ACID, controle de transações, controle de conflitos. A vantagem dessa flexibilidade é que mudanças nos requisitos das aplicações não fazem com que o banco precise ser (pelo menos parcialmente) travado para que mudanças no esquema possam ser realizadas.

Para que seja possível utilizar um BD NoSQL é necessário considerar a natureza dos dados. Se os dados forem naturalmente livres de esquema e não exigirem uma forte rigidez entre os relacionamentos, então a utilização de um banco NoSQL poderá ser uma boa opção. Além disso, apesar dos dados poderem ser estruturados com NoSQL é importante que o foco seja a inserção e recuperação de grandes quantidades de dados e não a relação entre os elementos.

Outros usos do NoSQL são relacionados com a flexibilidade do seu modelo de dados; muitas aplicações podem ganhar com este modelo desestruturado dos dados: ferramentas como CRM, ERP, BPM poderiam utilizar esta flexibilidade para armazenar seus dados sem modificar tabelas ou criar colunas genéricas no banco de dados. NoSQL também é bom para criar protótipos ou *fast applications* já que sua flexibilidade providência uma ferramenta para desenvolver facilmente novas funcionalidades.

Tipos de bancos de dados NoSQL

Um fator que possui lados positivos e negativos nos bancos de dados NoSQL é a não existência nem de uma interface padrão para acesso aos dados (como o SQL, por exemplo) e nem de uma forma padrão de organização dos dados. O fator negativo é que a padronização garante uma interface uniforme e uma independência de plataforma e isso não existe entre os BDs NoSQL. O fator positivo é a existência de uma grande gama de possibilidades que permitem atender a um maior número de requisitos das diferentes aplicações. Por exemplo, enquanto existem aplicações em que é mais interessante utilizar arquitetura baseada em documentos, existem outras aplicações em que a arquitetura baseada em chave/valor se encaixa melhor. Os bancos NoSQL são subdivididos pelo seu núcleo, ou seja, como ele trabalha com os dados. Alguns destes núcleos são:

- ***Key Value*.** Tipo mais simples de NoSQL, onde os dados são armazenados em registros desestruturados consistindo de uma chave com os valores associados com este registro. Estes tipos de bancos de dados são os que tem maior escalabilidade. Utilizam esta tecnologia: Berkeley DB, Tokyo Cabinet, Project Voldermort, MemcacheDB, SimpleBD.
- ***Wide Column Store*.** Também chamados de bancos de dados orientados a registros, são similares aos bancos de dados relacionais. É constituído por várias tabelas, cada uma contendo um conjunto de linhas endereçáveis. Cada linha consiste de um conjunto de valores que são considerados colunas. Fortemente inspirado pelo BigTable do Google, suporta várias linhas e colunas, e permite subcolunas. Utilizam esta tecnologia: HBase (Apache), HiperTable, Cassandra (Apache).
- ***Document Store*.** Documentos encapsulam e codificam dados (ou informações) em algum formato ou codificação padrão. Codificações em uso incluem XML,

YAML, JSON, BSON, PDF e Microsoft Office documents. Suas implementações podem organizar e/ou agrupar os documentos por coleções, tags, *non-visible metadata*, hierarquias de diretórios. Documentos podem ser endereçados tanto pelo seu valor de chave único quanto pelo seu conteúdo. Utilizam esta tecnologia: CouchDB (Apache), MongoDB, Riak, RabenDB.

- ***Graph Store*.** Com uma complexidade maior, guardam objetos e não registros como os outros tipos de NoSQL. A busca destes itens é feita pela navegação desses objetos. Além disso, Graph Store enfatiza o alto desempenho para acesso a dados associativos, evitando a necessidade de junção. Utilizam esta tecnologia: Neo4J, InfoGrid, HyperGraphDB, BigData.
- ***Column Oriented Store*.** Bancos de dados relacionais com características de NoSQL. A principal diferença deles é que os dados são armazenados em colunas, ajudando na escalabilidade. Utilizam esta tecnologia: Vertica, MonetDB, LucidDB, Infobright, Ingres/Vectorwise.

Outra forma de dar alguma estrutura aos dados ficou famosa por causa do Google Bigtable é a abordagem *column-oriented*. A idéia é salvar os dados através de colunas, e não em linhas (*row-oriented*) como nos bancos relacionais. A adoção desta abordagem fica transparente para o desenvolvedor que vai utilizar o banco de dados, mas para quem desenvolveu o banco, há enormes melhorias. Isto não é muito vantajoso quando for salvo apenas um registro, como cada coluna tem que ser acessada separadamente. Também complica mais na recuperação de um registro específico (*random-access*) pelo mesmo motivo. Nestes casos a abordagem row-oriented tem vantagens. Por outro lado, usando colunas, pode-se empacotar os dados melhor, já que os dados semelhantes, de mesmo formato, estão próximos um do outro. Gravando dados empacotados em BDs traz grandes vantagens, porque podemos recuperar e armazenar mais informações em menos tempo. Outra vantagem é a facilidade em aplicar projeção, a qual é importante principalmente para sistemas OLAP (*online analytic process*) que usam este tipo de pesquisa pesadamente.

Registros em NoSQL

No caso dos bancos NoSQL, toda a a informação necessária está normalmente agrupada no mesmo registro, ou seja, em vez de haver o relacionamento entre várias tabelas para formar uma informação, ela está - em sua totalidade - no mesmo registro sem utilizar tipagem de dados. Entretanto isto não é uma regra, afinal a diferença entre os bancos de dados NoSQL dos relacionais é que estes primeiros não possuem esquemas padrões de relacionamento, o que significa que relacionamentos “não oficiais” podem ser criados em bancos NoSQL. O BD não se encarrega de cuidar dos relacionamentos, mas sim o programador.

Em geral, a ideia de registros NoSQL se encaixa melhor com os objetos do mundo real. Considere um exemplo de um banco de dados para cartões de visita. Em um banco de dados relacional esse cartão poderia ser modelado como uma tabela e suas colunas seriam nome, telefone e outras informações que possam existir em um cartão de visita. Repare que apesar dos cartões de visita possuírem algumas informações em comum, nem sempre eles possuem as mesmas informações. No caso do fax, por exemplo, alguns cartões de visita apresentam essa informação e outros não. Com um banco de dados relacional é necessário a definição de todos possíveis atributos da tabela. Assim, em nosso caso obrigatoriamente definiríamos uma coluna para fax, porém como nem todos cartões de visita possuem essa informação, a coluna fax muitas vezes terá valor nulo. Em contrapartida, com bancos de dados NoSQL cada cartão de visita pode ter a informação que precisar sem a necessidade de definir no banco de dados que tal informação poderá existir. Por exemplo, é possível ter um cartão de visita de João armazenado apenas com o nome e telefone, e é possível ter um cartão de visita de Maria contendo, nome, telefone, fax, endereço e quantas outras informações que forem necessárias.

Outra vantagem que a flexibilização existente nos registros de BDs NoSQL trás é nos casos onde os requisitos da aplicação mudam. Considere ainda o exemplo dos cartões de visita e considere que no banco de dados relacional eles possuem as seguintes colunas: nome, telefone, fax, endereço. Agora imagine que seja necessário

incluir, em alguns cartões de visita, a informação do endereço de correio eletrônico. Nesse caso a estrutura da tabela terá que ser refeita para atender ao novo requisito da aplicação. Já em um banco NoSQL basta que a informação do endereço de correio eletrônico seja adicionada aos registros necessários.

Escalabilidade

Escalabilidade é a característica que avalia a habilidade de um sistema em aumentar sua capacidade de trabalho. Se um sistema possuí uma boa escalabilidade, está preparado para que o seu fluxo de trabalho aumente. **Escalabilidade vertical** é escalar adicionando novos componentes na máquina (aumentar a memória, por exemplo). A escalabilidade vertical, em geral, tem limite físico rapidamente atingível. Já a **escalabilidade horizontal** visa aumentar o número de máquinas em paralelo, fator que gera sistemas e arquiteturas mais complexas.

Bancos de dados relacionais são muito bem desenhados para rodar em uma uma única máquina. Eles funcionam muito bem desta forma, porém quando o volume de dados e de acessos aumentam a solução inicial de melhorar a máquina, buscando aumentar a capacidade de armazenamento e processamento é limitada. Se o volume de dados e acessos continuar crescendo a solução passa a ser buscar pela escalabilidade horizontal, onde se tenta distribuir o banco em diversas máquinas e não apenas em uma. O problema dessa abordagem é que pode ser muito caro e complexo manter um banco de dados distribuído. Além disso, o *overhead* para manter um BD distribuído pode, muitas vezes, degradar o desempenho drasticamente.

O NoSQL surgiu da necessidade de uma performance superior e de uma alta escalabilidade. Os atuais bancos de dados relacionais são muito restritos a isso, sendo necessária a distribuição vertical de servidores, ou seja, quanto mais dados, mais memória e mais disco um servidor precisa. O NoSQL tem uma grande facilidade na distribuição horizontal: mais dados, mais servidores, não necessariamente de alta performance. Um grande utilizador desse conceito é o Google, que usa computadores de pequeno e médio porte para a distribuição dos dados; essa forma de utilização é muito mais eficiente e econômica.

Bancos de dados NoSQL permitem uma melhor escalabilidade pois não possuem uma estrutura rígida e porque possuem, devido a sua natureza, um controle de consistência relativamente fácil de ser realizado. Dessa forma, o *overhead* para manter

uma BD NoSQL distribuído é relativamente baixo. Isso permite que a escalabilidade horizontal possa ser feita sem as mesmas preocupações que existem com um banco relacional. Além disso, os bancos de dados NoSQL são muito tolerantes a erros.

Teorema CAP

- **Consistence**: após a operação o dado já está pronto para consulta.
- **Availability**: sempre disponível. Tolerante a falhas de máquinas individuais e expansão sem *downtime*.
- ***Partition Tolerance***: os dados estão espalhados em várias máquinas tanto para leitura quanto para escrita.

CouchDB

O CouchDB é um banco de dados NoSQL que se baseia no formato de *document store,* ou seja, cada registro é um documento que é armazenado em algum formato específico. No caso do CouchDB esse formato é o JSON (*JavaScript Object Notation*) que possui um processamento relativamente leve se comparado com outros formatos como XML. Ele é escrito em Erlang que é uma linguagem tolerante a falhas, com paradigmas concorrente e funcional. Erlang foi desenvolvida para suportar aplicações distribuídas que executam em tempo real ininterruptamente.

Entre as principais características do CouchDB estão a utilização de *map-reduces*, *views* escritas em JavaScript, API REST para acesso e manipulação dos dados, possibilidade de validação dos dados antes de armazená-los e até mesmo a construção de aplicações Web inteiramente embutidas no banco de dados.

Couch é um acrônimo para “Cluster of Unreliable Commodity Hardware” (Conjunto de Hardware não confiáveis). A ideia por trás do nome é refletir o objetivo do banco de dados de ser extremamente escalável, oferecendo alta disponibilidade e confiabilidade, mesmo quando executando em um hardware que é tipicamente suscetível a falhas.

Organização física

API REST

Uma de suas principais ideias é a utilização da Web como canal direto para a inserção, busca, remoção e atualização dos dados. Mesmo a definição de parâmetros de configuração e a definição das *views* (que serão tratadas mais adiante) são feitas via Web. O CouchDB apresenta portanto uma Arquitetura Orientada a Recursos (ROA, da sigla em Inglês) e como utiliza a Web para fornecer seus serviços, a implementação dessa arquitetura é REST (Representational State Transfer). Uma aplicação REST consiste da utilização de um conjunto de tecnologias da Web para garantir os princípios definidos por ROA, princípios como identificação única de recursos, interface uniforme,

múltipla representação dos recursos e outros.

A ideia do CouchDB é justamente servir a estas aplicações onde o modelo de dados NoSQL se encaixa muito bem. Em geral, estas aplicações também são Web e utilizam REST, sendo assim a escolha do CouchDB por utilizar este tipo de arquitetura para a disponibilidade de seus serviços faz com que as aplicações naturalmente Web tenham uma maior facilidade de integração com o banco de dados.

- `/`
- `/{nomeDoBanco}`
- `/{nomeDoBanco}/{identificadorDoDocumento}`
- `/{nomeDoBanco}/{identificadorDoDocumento}/{nomeDoAnexo}`
- `/{nomeDoBanco}/_design/{nomeDoDesing}`
- `/{nomeDoBanco}/_design/{nomeDoDesing}/_view/{nomeDaView}`
- `/{nomeDoBanco}/_design/{nomeDoDesing}/_show/{nomeDaShowFunction}/{identificadorDoDocumento}`
- `/{nomeDoBanco}/_design/{nomeDoDesing}/_list/{nomeDaListFunction}/{nomeDaView}`

São suportados os seguintes métodos HTTP:

- **GET**: requisita um item específico, podendo ser uma base de dados, um documento, informações de configuração, anexos e outros.
- **HEAD**: tem o mesmo efeito que um GET, porém retorna apenas o cabeçalho da resposta.
- **POST**: permite o envio de novos dados. Ele é usado para fixar dados de documentos, anexos e alterar configurações administrativas.
- **PUT**: usado para colocar um recurso específico. Permite a criação de base de dados, documentos, *views* e *design documents*.
- **DELETE**: remove um recurso específico, incluindo documentos, *views* e *design documents*.
- **COPY**: é um método especial do não padrão do HTTP utilizado para copiar documentos e objetos.

JSON

Futon

O CouchDB disponibiliza ainda um cliente Web que fornece, através de uma interface gráfica de fácil utilização, funcionalidades para manipulação das diversas bases de dados. Com o Futon é possível realizar de forma transparente inserção, edição, remoção e atualização de dados, definição de *views*, configuração de parâmetros da base de dados, criação de bases de dados e outros. Em suma, o Futon permite que se realize, através de uma interface gráfica, o acesso a API HTTP REST disponibilizada pelo CouchDB.

Para utilizar o Futon basta acessar através de um navegador Web, o local onde o CouchDB está rodando. Por padrão o Couch roda na porta 5984, então se você estiver rodando localmente, basta acessar através do endereço `localhost:5984/_utils`.

Na apresentação deverá ser mostrado através do Futon:

- Criar um banco de dados;
- Criar um documento;
- Criar outro documento;
- Visualizar os documentos inseridos;
- Editar um documento e mostrar a revisão nova;
- Anexar um arquivo;
- Remover um documento;
- Criar uma *view* com *map*;
- Rodar uma o *map*;
- Adicionar na *view* criada um *reduce*;
- Rodar o *reduce*;

Documento

Um documento básico do CouchDB possui pelo menos dois atributos, `_id` e `_rev`. O primeiro se refere ao identificador único do documento e pode ser tanto fornecido na criação do documento, quanto pode ser deixado para ser criado pelo

CouchDB. Caso ele seja gerado pelo CouchDB, ele será uma String JSON aleatória contento 32 caracteres.

Já `_rev` indica a revisão do documento. Esse valor é gerado e mantido automaticamente pelo Couch. Após ser criado, toda vez que um determinado documento é modificado ele tem o seu atributo `_rev` atualizado. Um fator importante é que o Couch mantém todas as revisões dos documentos, sendo assim é possível obter todas as modificações feitas em um documento ao longo do tempo. Manter as revisões é particularmente útil em aplicações em que isso seja naturalmente um requisito, como no caso das *wikis*, por exemplo. Um outro motivo para a existência da revisão é que o Couch não realiza controle de conflitos. Ou seja, se dois usuários atualizarem um documento ao mesmo tempo, ambas as atualizações serão guardadas e poderão ser acessadas. A atualização que chegar por último será a revisão atual. Existe também a opção de eliminar as revisões antigas, para isso deve ser utilizada a opção compactar.

Um documento além de ser composto por `_id`, `_rev` e seus atributos, também pode conter anexos que são mantidos no atributo `_attachments`. Os anexos nada mais são que arquivos em qualquer formato. Isso possibilita que arquivos binários sejam guardados no banco. Assim, para guardar um arquivo binário é necessário criar um documento e anexar este arquivo ao documento criado. Um fator interessante é que um documento pode ter anexado a si diversos arquivos. Considere que se deseja guardar álbuns musicais. Cada álbum poderia ser um documento e seria possível anexar a esse documento todas as músicas do álbum.

Designs documents

Design documents são um tipo especial de documento do CouchDB que contem código de aplicação. Como falado anteriormente o Couch permite que aplicações inteiras sejam construídas e embutidas no próprio banco de dados. O que realiza essa mágica são os *design documents*. Com eles é possível criar *templates*, realizar validações, fornecer documentos com outros formatos (que não o JSON), criar *views* que são responsáveis pelas buscas e outros.

Ao contrário dos documentos normais que possuem a URI

`/{nomeDoBanco}/{identificadorDoDocumento}`, os *design documents* são mantidos na URI `/{nomeDoBanco}/_design/{nomeDoDesing}`. Com exceção da URI e dos atributos especiais, os *design documents* funcionam como um outro documento qualquer, sendo assim, eles podem ser criados, removidos, replicados, alterados e atualizados da mesma forma como ocorre com outros documentos. Além disso, eles também são guardados no formato JSON.

Na prática é possível ter quantos *design documents* quanto for desejado. O que se faz em geral é modularizar os *design documents* e dividi-los por entidade da aplicação. Por exemplo, em um sistema para um livro de receitas, podería-se ter um *design documents* para as receitas, outro para ingredientes, outro para os cozinheiros e assim por diante. Entretanto o esquema é livre, então se o programador desejar é possível colocar toda aplicação dentro de apenas um *design document*.

Nos *designs documents* o atributo `validate_doc_update` é uma função escrita em JavaScript que é responsável por realizar a validação de documentos que são atualizados ou criados. Caso o novo documento não seja validado pela função, então um erro HTTP 40X será retornando indicando se tratar de uma má requisição e o documento não será atualizado. Também é possível personalizar as mensagens de erro. Toda vez que um documento é inserido, o CouchDB roda a função existente em `validate_doc_update` (caso exista) e passa como parâmetro, o documento antigo, o novo documento e o usuário que fez a modificação. A função de validação receberá três parâmetros, o documento antigo, o novo documento e o contexto do usuário. Para invalidar a inserção do documento basta utilizar a função `throw`, passando como parâmetro o erro em questão. Por exemplo, `throw({forbidden : 'no way'});`.

Nos *design documents* também é possível ter as chamadas *show functions* que provêem os documentos em formatos alternativos. Por exemplo, é possível ter uma *show function* que fornecer o documento em formato HTML. Um *design document* pode conter várias *shows functions* que são guardadas no atributos `shows`. Cada uma delas será uma função JavaScript que receberá como parâmetro o documento que se deseja formatar e os detalhes da requisição HTTP. Para obter um documento formatado por uma

determinada *show function*, bastar enviar uma requisição GET na URI `/{nomeDoBanco}/_design/{nomeDoDesing}/_show/{nomeDaShowFunction}/{identificadorDoDocumento}`.

Existem também as *lists functions*. Elas funcionam da mesma forma que uma *show function*, porém ao invés de um documento, elas são aplicadas a uma *view*. O acesso a uma *show function* se dá através da seguinte URI `/{nomeDoBanco}/_design/{nomeDoDesing}/_list/{nomeDaListFunction}/{nomeDaView}`.

Views são a forma primária para a realização de buscas no CouchDB. Elas são armazenadas em algum *design document* no atributo `views` que pode conter várias delas. As views utilizam o conceito de *MapReduce* e no CouchDB elas são funções JavaScript que realizam o *map* e o *reduce*. Através do *map* os dados são filtrados e ordenados, enquanto que através do *reduce*, os dados provenientes do map são agregados com o objetivo, em geral, de fornecer uma resposta única.

Ao criar uma *view*, ela é armazenada em um *design document*, porém não é executada de imediato. A *view* somente será executada quando for requisitada pela primeira vez. Nesse processo de execução o CouchDB rodará para cada documento no banco de dados a view, passando como parâmetro o documento em questão. Entretanto o Couch apenas realizará esse processo uma vez após a criação ou atualização das views e guardará o resultado em disco. Se um documento é criado, removido ou atualizado entretanto, o Couch rodará a view apenas para o documento em questão e atualizará ela em disco. Devido a isso, acessar uma view pela primeira vez após a sua criação ou atualização pode demorar já que ela terá que ser rodada para todos documentos do banco. Porém os próximos acessos serão extremamente rápidos já que o resultado da *view* será guardado.

Para acessar uma view, basta enviar uma requisição HTTP GET na URI `/{nomeDoBanco}/_design/{nomeDoDesing}/_view/{nomeDaView}`.

No CouchDB, dentro de uma função map é possível chamar o método `emit`. O `emit` recebe dois parâmetros, o primeiro é a chave que será utilizada para ordenar os

resultados do *map*, e o segundo é o próprio resultado. Ou seja, o `emit` tem o objetivo de incluir incluir registros no resultado da *view*. A chave utilizada para ordenar os resultados pode ser composta e o valor emitido pode ser composto apenas de alguns atributos do documento e não de todos. É interessante perceber também que a função `emit` pode ser chamadas dentro de um *map*, tantas vezes quanto for desejado.

Os resultados de uma view são armazenados em forma de uma árvore B, da mesma forma como ocorre com os documentos do Couch. Os resultados são mantidos em um arquivo separado, por isso, é possível que uma *view* seja armazenada no seu próprio disco.

Ao requisitar uma view é possível utilizar alguns parâmetros para escolher apenas um *range* de resultados, limitar a quantidade de resultados, ou até mesmo selecionar apenas um resultado. Com o parâmetro `?key={chave}` apenas um resultado da view é selecionado. O resultado selecionado será aquele que tenha sido emitido com a chave em questão. Para selecionar um range de resultados utiliza-se os atributos `?startkey={chave}` e `?endkey={chave}`, podendo utilizar apenas um desses atributos ou os dois em conjunto. É importante lembrar que os resultados da view são ordenados de acordo com a chave emitida. É possível também obter os resultados em ordem contrária. Para isso, basta apenas utilizar o parâmetro `?descending=true` na requisição. Também é possível, no caso de chaves compostas, indicar quantas das chaves serão utilizadas nos parâmetros `?key={chave}`, `?startkey={chave}` e `?endkey={chave}`. Para isso, utiliza-se o atributo `?group_level={numero}`. Por exemplo, se `?group_level=2`, então será considerado apenas as duas primeiras chaves da chave composta. Se for `?group_level=1`, então será considerada apenas a primeira chave da chave composta. Já o parâmetro `?limit={limite}` é utilizado para limitar a quantidade de resultados selecionados em uma *view*, por exemplo *`?limit=10`* seleciona apenas 10 resultados da *view*;

A função *reduce* é responsável por agregar os resultados provenientes de um *view* em um único resultado. Por exemplo, uma função *reduce* pode contar o número de registros, a soma de valores de registros, o valor máximo, o valor mínimo e outros. A

função *reduce* pode ser chamada em dois casos diferentes. No primeiro caso em que o parâmetro *rereduce* tem o valor falso, ela recebe como parâmetros as chaves e os valores emitidos no *map*. No segundo caso em que o parâmetro *rereduce* tem o valor verdadeiro, ela recebe como parâmetro os valores resultantes dos *reduces* anteriores. O que ocorre é que a função *reduce* é chamada várias vezes. Primeiramente ela será chamada recebendo como parâmetro os diversos agrupamentos de nodos folhas da árvore B resultante do *map*. Após isso, ela será chama novamente, porém, os valores ao invés de serem provenientes do *map*, serão provenientes dos *reduces* anteriores. Isso permite um maior paralelismo já que vários *reduces* podem ser executados ao mesmo tempo e o resultado final será o junção de todos os *reduces*.

Replicação

Consiste na sincronização de dados de duas cópias da mesma base de dados. Pode ser feito com as cópias em um mesmo servidor ou em servidores diferentes, sendo que em ambos os casos a operação é executada da mesma forma.

O procedimento de replicação é feito a partir da comparação dos dados da base fonte e da base destino. O objetivo é definir quais documentos da base fonte diferem da base destino. Em seguida as mudanças detectadas são subtidas a base destino. Nesse tipo de replicação os dados sempre seguem o fluxo fonte-destino. Porém, outro tipo de replicação é a *master-master*, onde o fluxo de dados segue em ambas as direções. Nesta replicação há duas tarefas, cada qual monitorando um banco de dados. Um exemplo de replicação *master-master* seria: suponha que a tarefa A monitora a base de dados fonte e a tarefa B monitora a base destino. Caso ocorra uma mudança na fonte a tarefa A deve submeter a mudança à base destino. Quando ocorrer a mudança na base destino a tarefa B irá tentar submeter a mudança a base fonte, porem como o dado alterado vai estar sincronizado, nenhuma mudança ocorre na base fonte.

É possível também definir filtros de replicação. Assim é possível definir situações onde as mudanças não devem ser submetidas. A tarefa de atualização deve primeiramente avaliar os filtros antes de confirmar se uma mudança deve ser submetida a

base destino.

Exemplo mostrando comparativamente SQL com CouchDB

- SELECT * FROM times
- SELECT nome, dataDeFundacao FROM times
- SELECT * FROM bandas WHERE _id = "..."
- SELECT nome FROM times WHERE estado = "SC"
- SELECT nome FROM times WHERE dataDeFundacao BETWEEN 1910 AND 1920
- SELECT nome FROM times WHERE estado = "SC" AND dataDeFundacao BETWEEN 1990 AND 1999
- SELECT nome FROM times WHERE estado = "SC" OR estado = "SP"
- SELECT nome, capacidadeDoEstadio FROM time ORDER BY capacidadeDoEstadio
- SELECT nome, capacidadeDoEstadio FROM time ORDER BY capacidadeDoEstadio DESC

- SELECT nome FROM jogadores WHERE identificadorDoTime = "..."

- SELECT t.nome, j.nome FROM jogadores j JOIN times t ON j.identificadorDoTime = b._id
- SELECT t.nome, c.nome FROM participa p JOIN campeonatos c ON c._id = p.identificadorDoCampeonato JOIN times t ON t._id = p.identificadorDoTime WHERE t._id = "..."
- SELECT t.nome, c.nome FROM participa p JOIN campeonatos c ON c._id = p.identificadorDoCampeonato JOIN times t ON t._id = p.identificadorDoTime WHERE c._id = "..."
-
- SELECT nome FROM times WHERE nome LIKE "%Atlético%"
- SELECT COUNT(*) FROM times
- SELECT COUNT(estado), estado FROM times GROUP BY estado
- SELECT COUNT(estado), estado FROM times GROUP BY estado HAVING COUNT(*) > 10
- SELECT MAX(capacidadeDoEstadio) FROM times

Conclusão

Os bancos de dados relacionais são a solução que melhor se enquadra para armazenamento de dados em aplicações com especificações bem definidas. Podem ser especificações como: controle de transações, integração de aplicações e relacionamento entre dados. Por exemplo uma aplicação bancária, as transações devem ser rigidamente controladas e a atualização dos dados deve ser imediata para evitar inconsistências ou operações inválidas (por exemplo um saque que reduz o saldo a zero seguido de uma transferência de um valor qualquer, neste caso a transferência deveria ser bloqueado por saldo insuficiente).

Bancos de dados NoSQL são uma alternativa para armazenamento a baixo custo garantindo escalabilidade horizontal. Promovem soluções simplificadas e com boa flexibilidade. Tais características despontam os bancos de dados NoSQL como a solução ideal para bancos de dados distribuídos, uma tendência para aplicações com grande volume de dados (big data).

Referências sobre NoSQL

- http://martinfowler.com/nosql.html
- http://martinfowler.com/articles/nosql-intro.pdf
- http://martinfowler.com/articles/nosqlKeyPoints.html
- http://martinfowler.com/bliki/NosqlDefinition.html
- http://martinfowler.com/bliki/PolyglotPersistence.html
- http://martinfowler.com/bliki/AggregateOrientedDatabase.html
- http://nosql.mypopescu.com/
- http://nosql.mypopescu.com/kb/nosql
- http://nosql.mypopescu.com/kb/nosql-getting-started
- http://nosql.mypopescu.com/post/1016366403/nosql-guide-for-beginners
- http://imasters.com.br/artigo/17043/banco-de-dados/nosql-voce-realmente-sabe-do-que-estamos-falando
- http://blog.caelum.com.br/bancos-de-dados-nao-relacionais-e-o-movimento-nosql
- http://nosql-database.org
- http://oracle.com/technetwork/products/nosqldb/overview/index.html
- https://en.wikipedia.org/wiki/NoSQL
- http://pt.wikipedia.org/wiki/NoSQL
- http://fxplabs.com.br/blog/nosql-conceito-de-banco-de-dados-nao-relacional
- http://slideshare.net/jornaljava/banco-de-dados-nosql-jornaljava
- http://www.aerospike.com/what-is-a-nosql-key-value-store/

Referências sobre CouchDB

- http://couchdb.apache.org
- http://docs.couchdb.org
- http://guide.couchdb.org
- http://guide.couchdb.org/draft/
- http://wiki.apache.org/couchdb/
- http://pt.wikipedia.org/wiki/CouchDB

- http://ibm.com/developerworks/br/library/os-couchdb

Outras referências

- http://www.json.org/