Complementos de Bases de Dados

José Mendes 107188

2023

1 Evolução dos Sistemas de Base de Dados

Sistemas de Dados - Cada vez mais as aplicações de hoje em dia são Data-Intensive, em vez de Compute-Intensive.
Para Data-Intensive, o poder bruto da CPU deixa de ser um fator limitante quando comparado com a **quantidade**, **complexidade** e **velocidade de atualização** dos dados.

De forma a otimizar a sua performance, um sistema de dados tipicamente oferece as seguintes funcionalidades:

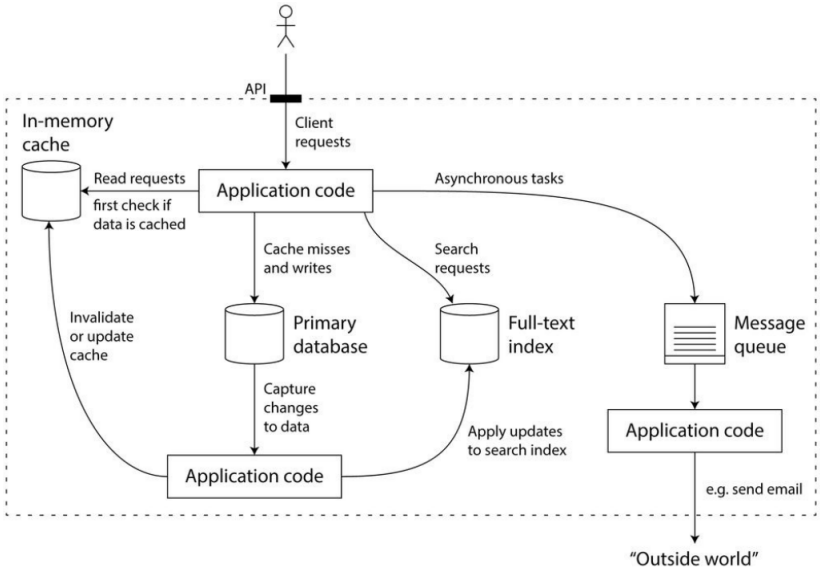
1. **Bases de Dados -** armazenam os dados para utilização futura;
2. **Caches -** guardam os resultados de operações dispendiosas, de forma a tornar a leitura mais rápida;
3. **Search Indexes -** permitem aos utilizadores procurarem por palavras-chave ou filtrar os dados;
4. **Message Queues -** permitem a comunicação assíncrona entre processos;
5. **Stream Processing -** permite o processamento de dados em tempo real;
6. **Batch Processing -** permite o processamento de dados acumulados, periodicamente;

Exemplo: Um exemplo de **stream processing** ocorre na banca. Sempre que é realizada uma transação, os dados da mesma são imediatamente processados de forma a que o saldo esteja sempre atualizado.
O **batch processing** é visível na faturação dos serviços pós-pagos pelas operadoras de telecomunicações. No final de cada mês, é feita uma consulta às suas bases de dados de forma a identificar todos os consumos do cliente, que são somados e depois gerada a fatura.
No **stream** os dados são processados antes de armazenados, enquanto que no **batch** são processados depois de armazenados.

Cada vez mais as aplicações requerem um **maior wide-range de requisitos**. Muitas das vezes, uma única ferramenta já não consegue satisfazer todas as necessidades de **data processing** e **storage**.

Em vez disso, o trabalho é partido em tasks que possam ser realizadas de forma eficiente por uma única ferramenta. As ferramentas individuais utilizadas são depois juntas utilizando código de aplicação.

Exemplo: Podemos ter uma aplicação que utiliza uma Catching Layer (**memcached**), um Full-Text Search (**Elasticsearch**) e uma Base de Dados principal separada (**MySQL**).



1.1 Desafios que os Sistemas de Dados enfrentam

Como garantir que todos os dados se mantêm corretos e consistentes, mesmo quando, internamente, ocorreu algum erro? (ex: persistência de dados)

Como fornecer boa performance para os clientes, mesmo quando partes do sistema estão degredadas?

Como escalar o sistema para ser capaz de aguentar uma load intensiva de trabalho?

Qual a aparência de uma boa API para o serviço?

1.2 Alguns Requisitos

Fiabilidade - O Sistema deve continuar a funcionar corretamente em caso de adversidades (ex: falhas de hardware, software ou mesmo humanas).

Escalabilidade - O Sistema deve ser capaz de responder ao crescimento seja do volume de dados, do tráfego, ou mesmo da complexidade.

Manutenibilidade - Deve ser possível que o Sistema sofra alterações ao longo do tempo por várias pessoas diferentes de forma produtiva.

1.3 Bases de Dados

São definidas como um conjunto de dados relacionados entre si e a sua organização. Dividem-se em vários tipos, sendo atualmente os mais comuns: **Relacionais**, seguidas por **Documentais**, **Motores de busca**, **Chave-Valor**, entre outras. O controlo às bases de dados é realizado por **Sistemas de Gestão de Base de Dados** (**SGBD** ou DBMS em inglês). Estes fornecem funções que permitem a manipulação de grandes quantidades de informação.

2 NoSQL Databases - Key-Value Databases

1. É o mais simples dos tipos NoSQL;

   Consiste apenas em chaves únicas e a um "bucket" que contêm qualquer tipo de dados que se pretenda;

2. Pares chave-valor:

   Chave: (id, identificador, chave primária) Normalmente é uma String;

   Valor: Pode ser qualquer tipo de dados, texto, estrutura, imagem ...;

3. O conteúdo do valor ("bucket") pode ser, literalmente, qualquer coisa (mais comum é não estruturado ou semi-estruturado);

4. Os "buckets" podem armazenar entradas pesadas, incluindo BLOBs (Basic Large Objects);

5. Row based systems, utilizados para eficiência;

2.1 Vantagens

1. Tolerância a falhas elevada - sempre disponível;

2. Schemaless, logo, muito flexível. oferece uma grande escalabilidade para mudar os requisitos dos dados;

3. Eficiente a devolver dados de um objeto, com operações de disco minimas;

4. Muito simples, rápido e fácil de dar deploy;

5. Ótimo para escalabilidade horizontal (muitos servidores);

6. Não necessita de queries SQL, indexes, triggers, sp's, views, ...;

7. Data ingest rates muito elevadas (muitos dados a entrar);

   Favorece: escreve uma vez, lê muitas vezes;

8. Potente no "offline reporting" com data sets muito grandes;

9. Existem formas avançadas de KVs que apresentam capacidades de document ou column oriented stores;

2.2 Desvantagens

1. Não é apropriado para aplicação complexas;
2. Não é eficiente a ataualizar records em que apenas parte do "bucket" é alterado;
3. Não é eficiente em devolver informação limitada de records específicos (ex: returning only records of employees making between $40K and $60K);
4. Não é apropriado para queries complexas;
5. Com o aumento do volume de dados, manter chaves únicas pode tornar-se um problema;
6. Geralmente precisa de ler todos os records de um "bucket" ou talvez precise de contruir índices secundários;

2.3 Use Cases

1. Session data, user profiles, user preferences, shopping carts, . . . ;
2. Criar datasets que são raramente acessados mas crescem ao logo do tempo (Caching);
3. Onde a performance de escrite é a prioridade;

2.4 Quando NÃO usar

1. Quando precisamos de ter relações entre entidades;
2. Queries requerem acesso a conteúdos da parte dos valores;
3. Set operations que envolvem múltiplos pares chave-valor;

2.5 Key Management

Como devem as chaves serem produzidas?

Manualy assigned keys - Identificadores do mundo real (ex: e-mail, login names, . . .);

Automatically generated keys - Auto-incremente integers ou chaves mais complexas geradas por algotitmos;

2.6 Query Patterns

1. Basic **CRUD** operations;

   Apenas quando a chave for dada;

   O conhecimento da chave é essencial;

   Ás vezes, pode ser até dificil para uma base de dados dar uma lista com todas as chaves;

2. **No searching by value**;

   Mas pode-se instruir à base de dados como dar parse aos valores, para fazer operações;

3. **Batch / sequential processing**

   MapReduce;

2.7 Outras funcionalidades

1. Expiração de pares chave-valor;
2. Coleções de valores (We can store not only ordinary values, but also their collections such as ordered lists, unordered sets etc.);
3. Links entre pares chave-valor (podem ser usados quando se usa queries);

2.8 Exemplos

1. **RiakKV**
2. **Redis**

(Ver slides 12-40)

3 NoSQL Databases - Document Databases

As bases de dados de documentos são bastante eficientes em cenários **one-to-many**. Oferecem um esquema flexível (mesmo dentro das mesmas coleções (informação heterogénia)) e melhor performance (devido ao armazenamento da informação junto à entidade a que esta se refere) que são manipulados através de código simples.

Nota: A flexibilidade permite que existam objetos na mesma coleção com atributos ligeiramente diferentes, sem necessidade de criarmos uma tabela para cada tipo de objeto.
A localidade pode levar à duplicação de dados entre documentos.

Um **documento** caracteriza-se por uma **string continua** codificada em JSON, XML, ou outro formato binário estruturado. É self-described (atributos são claros) e apresentam uma **estrutura em árvore**. São identificados por um **id único**.

Geralmente, para manipular, é necessário carregá-lo por completo e para guardar as alterações reescreve-lo na totalidade.

Nota: A localidade só se torna uma vantagem se manipularmos porções grandes do documento.

Esta topologia não aplica a modelos **many-to-many**, pois não existem operações de **join** de documentos. Deve também ser evitada quando a estrutura do documento é demasiado instável (sempre a mudar).

Nota: Não é desejável demasiada granularidade entre os documentos porque se todos apresentarem características diferentes não são relacionáveis e assim não fará sentido estarem na mesma coleção.

No entanto, é a ideal para logging de eventos, sistemas de gestão de conteúdos, blogues, web analytics, aplicações e-commerce ... Por ser a solução para alguns, mas não para todos os problemas, as bases de dados relacionais começaram a incluir funcionalidades documentais e vice-versa.

3.1 Exemplos

1. **MongoDB**
2. **CouchDB**
3. **Couchbase**

(Ver slides 11-51)

4 Modelos de bases de dados

Os modelos de dados com os quais os programas vão trabalhar têm um papel fundamental na sua programação. Têm um efeito enorme na forma como os programas são escritos e na forma como nós pensamos sobre os problems que estamos a resolver. Todos os modelos de dados têm formas diferentes de representar os dados e de os manipular.

4.1 Bases de Dados Relacionais

Este tipo de base de dados oferece vários benefícios (resolvendo a maior parte dos problemas com dados), entre os quais a **persistência** dos dados (ao guardar dados eles mantêm-se guardados), a **integração** com várias aplicações (com a mesma DB) e a **atomicidade**, **consistência**, **isolamento** e **duranilidade** oferecidas pelas transações (ACID).

4.1.1 Transactions - ACID Properties

- **Atomicidade** - Numa transação, ou todas as operações são executadas, ou nenhuma é;
- **Consistência** - É garantido que as restrições de integridade antes da transação se mantêm após esta;
- **Isolamento** - As alterações feitas na BD só são visíveis quando a transação termina.
- **Durabilidade** - Assim que commited, as alterações de uma transação persistem mesmo em caso de falhas.

  A durabilidade é garantida através das transactions log, que permite a recunstrução das transações perdidas em caso de falhas.

(Ver slides 7-8, história das bases de dados relacionais)

4.2 Current Trends and Issues

Algumas trends e issues motivaram à mudança nas tecnologias de armazenamento de dados relacionais (em use cases e na tecnologia).
Key Trends include: Aumentar o volume de dados e tráfego. Ligação entre dados mais complexo. **Key Issues include:** O problema → **impedance mismatch**.

4.3 Impedance Mismatch

Nos últimos tempos, tem-se assistindo a um **aumento do volume de dados e tráfego**, a par da redução do relacionamento entre eles, ou seja, **cada vez há mais dados não relacionados**.

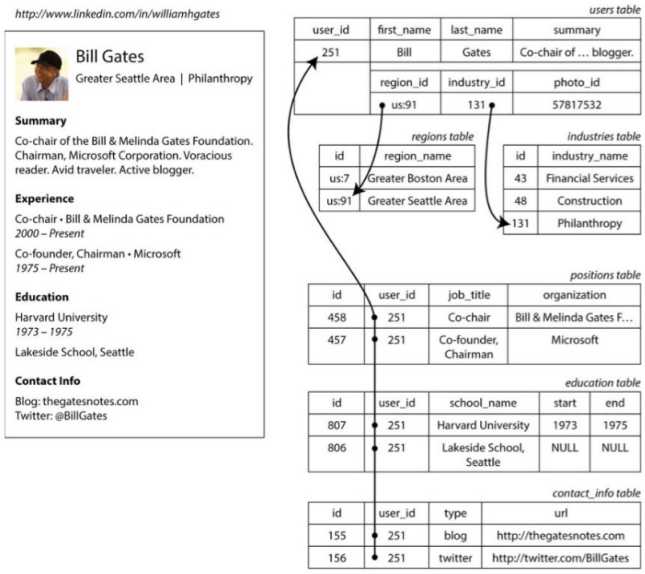
Tem-se também verificado um conflito entre os princípios de engenharia de software, onde o paradigma é orientado a objetos e os princípios relacionais baseados em modelos matemáticos. Este problema é designado por **Impedance** (oposição que um circuito elétrico faz à passagem de corrente elétrica quando é submetido a tensão) **Mismatch** (Disparidade, incompatibilidade).

Atualmente, este verifica-se nas estruturas isoladas, que violam os princípios da **normalização**. Para armazenar informação persistentemente em programas modernos, uma única estrutura lógica tem de ser separada (**normalização**).

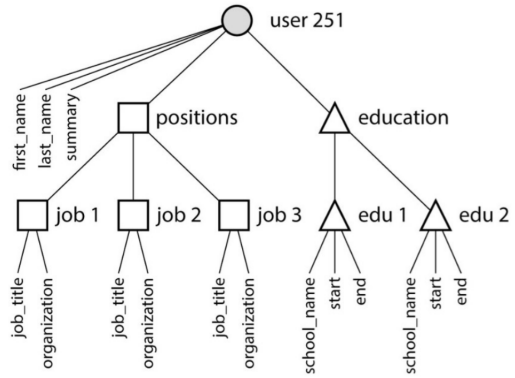
4.3.1 Exemplo

Vários objetos que representem funcionários numa empresa. Cada funcionário terá o seu departamento, mas vários funcionários podem trabalhar no mesmo departamento. Se a base de dados refletir o paradigma orientado a objetos, iremos ter uma repetição dos departamentos nos vários funcionários e base de dados não estará normalizada!

No entanto, fazer múltiplos selects e joins para construir uma entidade às vezes não é a melhor opção.



4.3.2 One-to-Many relations

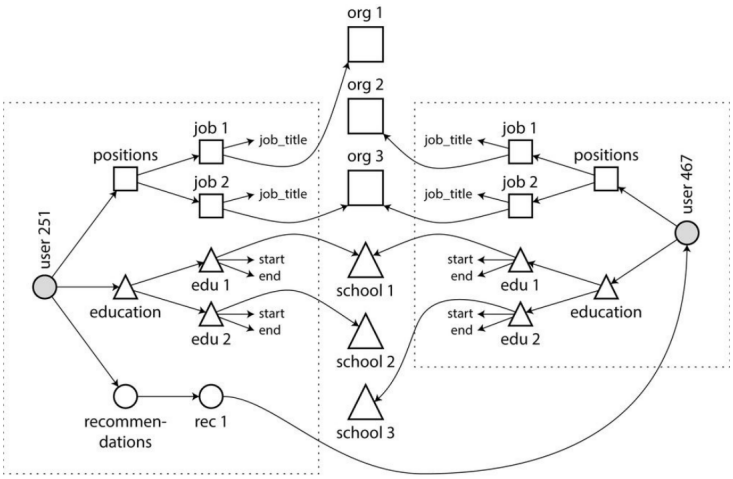


4.4 Normalização

Tem o objetivo de **reduzir a redundância dos dados**. Em DB's este reflete-se na utilização de IDs para identificar entidades, oferecendo uma **consistência de utilização**, ao mesmo tempo que **previne ambiguidades**, caso hajam entidades semelhantes (ex: com o mesmo nome), **facilita alterações** das entidades, uma vez que a sua informação está armazenada numa única tabela, motivo pelo qual também **facilita a tradução**.

Nota: Uma base de dados que verifica estas características diz-se **normalizada**. Uma base de dados na qual as entidades como região e industria estão referidas por ID diz-se **normalizada**. No entanto, uma basde de dados que duplica nomes e propriedades de entidades em cada documento diz-se **desnormalizada**.

4.4.1 Many-to-Many relationships



4.5 Responder ao aumento do volume de dados

We are creating, storing, processing more data than ever before!

Existem duas abordagens possíveis:

- Contruir bases de dados mauores;
- Criar um grupo de máquinas mais pequenas que se complementam;

1. A primeira abordagem tem alguns problemas, uma vez que o **custo** de duplicar a capacidade de uma DB é geralmente mais do dobro do custo de uma DB "normal" e mesmo com recursos financeiros, há **limitações físicas** e de engenharia à sua capacidade;
2. A segunda, apesar de mais exequível, tem também alguns defeitos, uma vez que por ser uma solução **barata**, pode refletir-se em **menos fiabilidade**. É ainda nexessário a integração com um DBMS compatível com a tipologia.

Os Sistemas de Gestão de Bases de Dados (DBMS) têm alguma **dificuldade em gerir a escalabilidade horizontal** (distribuição da BD).

4.6 O movimento NoSQL

Este movimento, cujo acrónimo significa **Not only SQL**, pretende promover a utilização de bases de dados não relacionais (SQL). Tem por base vários princípios.

1. **Não é relacional** - Podem ser mas não são boas nisso;
2. **API simples** - Sem necessidade de realizar join;
3. **Teorema BASE & CAP** - Viola os princípios ACID;
4. **Schema-free** - Esquema implícito e gerido pela aplicação (sem verificações do lado da DB);
5. **Distribuídas** - Algumas mais do que outras;
6. **Open-source** - Mostly;

4.6.1 Transações BASE

Este acrónimo nasceu em oposição aos princípios ACID, principalmente em resposta às limitações de consistência que um cenário de um sistema distribuído impõe.

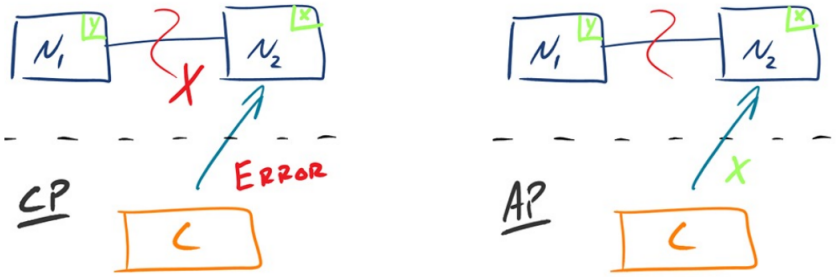
- **Basic Availability** - A DB funciona a maior parte do tempo;
- **Soft-state** - As manipulações dos dados não têm de ser write-consistent, nem diferentes réplicas têm de ser mutualmente consistentes o tempo todo. Escritas num nó da base de dados não têm de ser escritas garantidamente em simultâneo nos restantes nós;
- **Eventual consistency** - O armazenamento de dados eventualmente torna-se consistente, em algum ponto (e.g. lazily at read time);

As bases de dados NoSQL caracterizam-se então por ser **otimistas** e **simples**, o que torna a base de dados mais **rápida**, **disponibilidade em primeiro lugar**, **best effort** e **appoximate answers OK**.

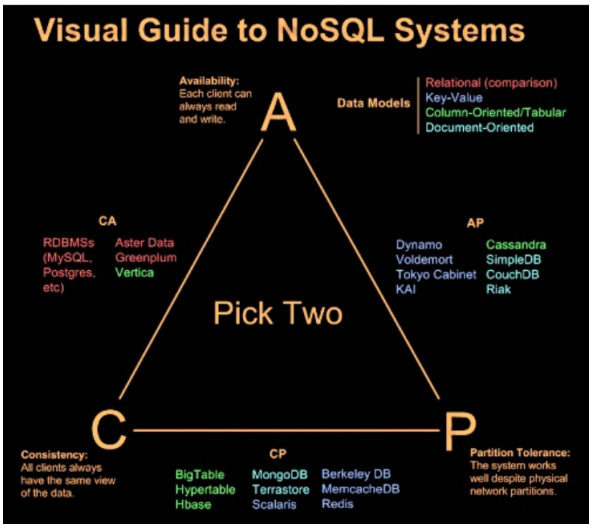
4.6.2 Teorema CAP (Brewer's)

Este teorema diz que um sistema distribuído só pode apresentar duas de três características:

- **Consistent** - Escritas atómicas em toda a DB em simultâneo;
- **Available** - A DB responde sempre a pedidos;
- **Partition Tolerant** - O sistema consegue funcionar mesmo que um nó deixe de responder;



1. No primeiro temos uma base de dados que implementa a consistência e a tolerância a falhas. Quando um cliente faz um pedido de consulta de um valor, caso o nó não consiga contactar os restantes de forma a confirmar que todos têm o mesmo valor, retorna uma mensagem de erro.
2. No segundo temos a disponibilidade e tolerância a falhas. Neste caso, mesmo com uma falha de comunicação entre os nós, o nó contactado pelo cliente vai responder com o valor pedido, mesmo que este não seja o mai atual.



4.7 Tipos de Bases de Dados NoSQL