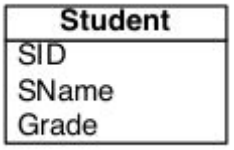
Resumos Teoria BDAD

UML

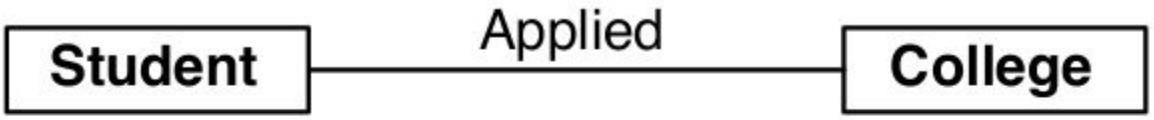
Classe

Representa um conjunto de objetos que partilham propriedades (atributos, relações, etc).
Caracterizados por nome (singular, primeira letra maiúscula) e atributos.



Associação

Relação entre objetos de duas classes.
Pode haver mais do que uma associação entre as mesmas classes, com outro nome.

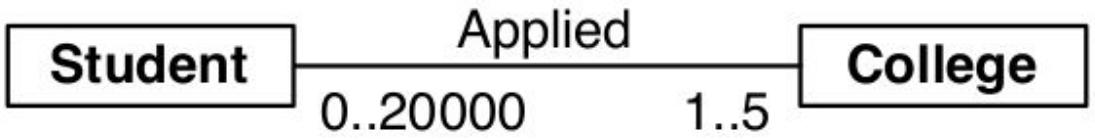


Multiplicidade

Cada objeto da classe 1 está relacionado com pelo menos m e no máximo n objetos da classe 2. Um * significa que não há limite superior

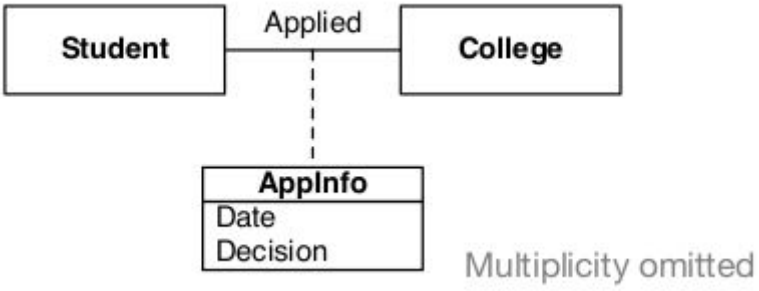
Multiplicidades comuns

- Muitos para um: * - 0..1
- Um para um: 0..1 - 0..1
- Muitos para muitos: * - *



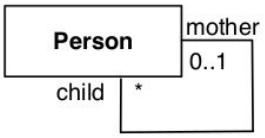
Classes associação

Relação entre duas classes, mas contendo também atributos sobre essa relação. Não são necessárias em multiplicidades de 0..1 ou 1..1 -> passar atributos para uma das classes (a que tiver maior multiplicidade)



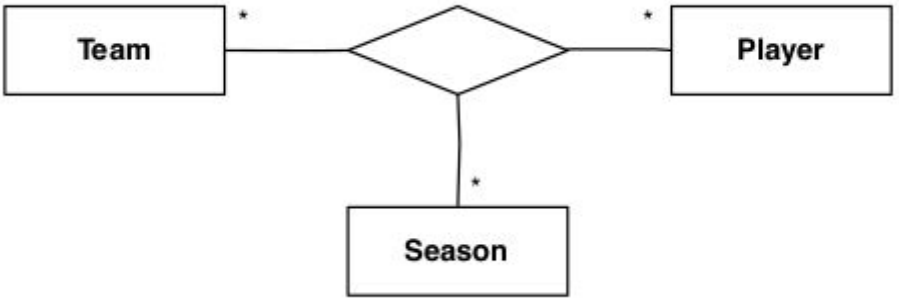
Associação Reflexivas

Associação entre dois objetos da mesma classe

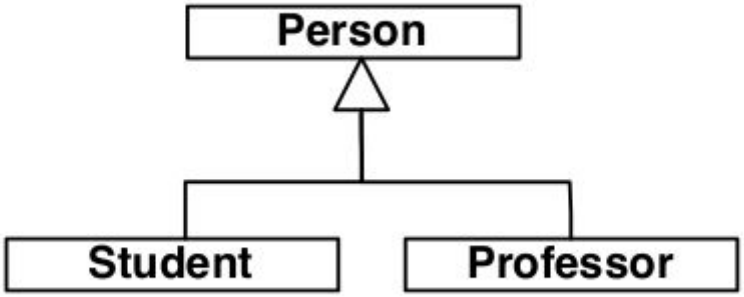


Associações n-árias

Representam associações entre mais que duas classes



Generalizações

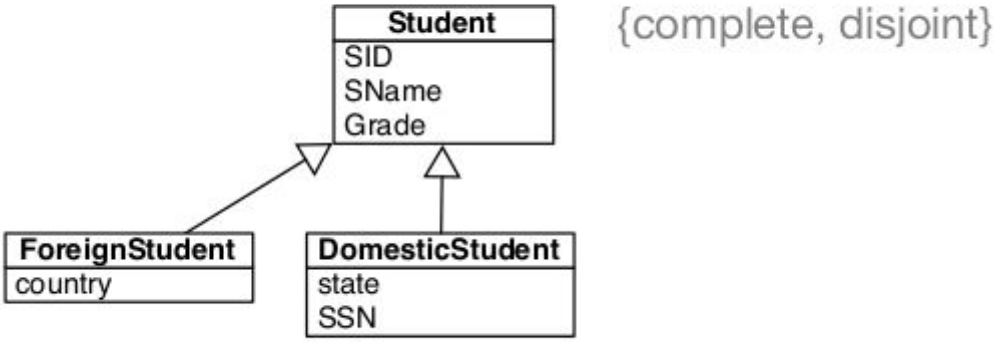


Quando classes herdam propriedades de outras, e podem também adicionar mais.

Propriedades

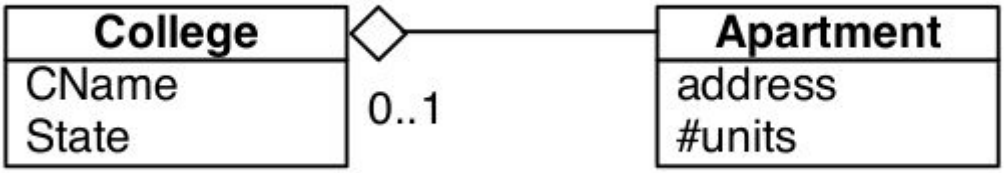
Completa -> se todos os objetos da superclasse estão em pelo menos uma sub classe
Incompleta -> !Completa

Exclusiva -> se todos os objetos da superclasse estão no máximo numa sub classe
Sobreposta -> !Exclusiva



Agregação

Objetos de uma classe pertencem a objetos de outra



Composição

Como agregação, mas multiplicidade é 1..1

Restrições

Representam condições que devem estar presentes no sistema, indicadas por uma nota perto dos elementos ao qual se refere

Elementos Derivados

Elemento que é calculado usando outros elementos no modelo.

Modelo Relacional

Relações -> Tabelas
Atributos -> Colunas

Tuplos: Linha de uma tabela, que representa um objeto da relação.

Chave

Conjunto de atributos cujos valores combinados são únicos numa relação.
Chave estrangeira
Conjunto de atributos que refere uma chave de outra relação.

Exemplo:
Student (ID, name, GPA, country->Country)
Classroom (building, number, capacity)
Country (ID, name)

Chaves compostas
Chave primária ou estrangeira com mais do que um atributo

Passar UML para Relações

Cada classe torna-se numa relação, onde os atributos são as colunas. A chave primária deve ser um atributo (ou conjunto de atributos) único para cada objeto.

Associações

Muitos para muitos
Criar relação com uma chave primária composta por chaves para cada classe da associação

Muitos para um
Colocar uma chave estrangeira na classe com maior multiplicidade, para a outra

Um para um
Colocar uma chave estrangeira de uma classe para a outra, e colocá-la na classe que se espera ter menos instâncias.

Classes associação

Criar relação com chave para cada classe associada, adicionando os restantes atributos da associação)

Associações Reflexivas

Tratar da mesma forma que uma associação normal, tendo em conta a multiplicidade da associação.

Associações n-árias

Criar relação com chave de cada classe.

Generalizações

Depende do tipo:

Sobrepostas com grande número de subclasses
Subclasses contêm chave para a superclasse (primária), e os seus restantes atributos

Exclusivas, onde superclasse tem poucos atributos e subclasses têm muitos
As subclasses contêm todos os atributos

Sobrepostas com menor número de subclasses
Ter uma única relação com todos os atributos, sendo que nas suas instâncias, alguns atributos ficam com valor NULL.

Composição e Agregação

Tratar como associação normal.

Restrições

NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK, DEFAULT

Teoria do Modelo Relacional

Anomalias
Quando modelos não são normalizados...
Redundância -> guardar mesma informação várias vezes
Atualizar -> factos são atualizados em certos locais, mas não em todos…
Apagar -> apagar tuplos sem querer

Dependências funcionais (DF)

Permitem desenhar melhor modelo, minimizando anomalias;
Usar ferramentas para comprimir espaço usado;
Otimizar *queries*.

DF trivial
Se lado direito da DF está contido na primeira.

Encontrar DFs

Regra da separação
Podemos separar uma DF em várias, se lado direito contém mais que um atributo (lado esquerdo nunca pode ser separado…)

Combinação
Inverso da anterior: se várias DF partilharem o lado esquerdo, podemos combinar numa única DF, juntando todos os atributos do lado direito.

Trivial
Podemos acrescentar ao lado direito o que já está no lado esquerdo.
Por exemplo: se A -> B, então A -> AB.

Transitiva

$$\bar{A} \rightarrow \bar{B} \ \wedge \bar{B} \rightarrow \bar{C} \Rightarrow \bar{A} \rightarrow \bar{C}$$

Fecho
Encontrar o conjunto de atributos funcionalmente dependentes do conjunto A.
Algoritmo:
Começar com o conjunto A. Usando dependências funcionais existentes, se o lado esquerdo já pertence ao conjunto, adicionar o lado direito também. Parar quando não houver mais alterações.

Fecho e Chaves
Se fecho de A inclui todos os atributos da relação, então A é uma chave.

Chave (ou chave candidata) se for super chave, mas nenhum subconjunto também for.

Formas normais

1NF
Cada célula na relação tem que ser atómica i.e não conter mais que um valor.

2NF
Atributo que faz parte de uma chave é primo.
1NF e nenhum atributo não primo é funcionalmente dependente de um subconjunto de uma chave candidata.

BCNF
Se para cada DF não trivial, lado esquerdo é uma chave

3NF
Se para cada DF não trivial, lado esquerdo é uma chave OU se lado direito têm apenas atributos primos

Chave e Super Chave
Super Chave se todos os atributos da relação forem funcionalmente dependentes dela.

Decompor em BCNF

1. Pegando nas DFs, calcular as chaves da relação

2. Repetir até todas as relações resultantes estarem em BCNF
   2.1 Pegar em qualquer relação R' com DF $\overline{A}$ -> $\overline{B}$ que viole BCNF
   2.2 Decompor em duas relações -> R1($\overline{A}$, $\overline{B}$) e R2($\overline{A}$, resto)
   2.3 Calcular DF para R1 e R2, usando as DF originais
   2.4 Calcular chaves para R1 e R2

Exemplo
Student(SSN, sName, address, HScode, HSname,HScity, GPA, priority)
DFs: SSN -> sName, address, GPA
GPA -> priority
HScode -> HSname, HScity
Chave: {SSN, HScode}

1. Escolher violação
   HSCode -> HSname, HScity (HSCode não é chave)

2. Decompor
   S1 (HScode, HSname, HScity)
   S2 (HScode, SSN, sName, address, GPA, priority)

3. FDs e chaves
   S1: FD: HScode -> HSname, HScity
   Chave: HScode

   S2: FD: SSN -> sName, address, GPA
   GPA -> priority
   Chave: {SSN, HScode}

   S2 ainda não está em BCNF -> **repetir processo**

1. Escolher violação
   GPA -> priority

2. Decompor
   S3 (GPA, priority)
   S4 (GPA, HScode, SSN, sName, address)

3. FDs e chaves
   S3: FD: GPA -> priority
   Chave: GPA

   S4: FD: SSN -> sName, address, GPA
   Chave: {SSN, HScode}

   S4 ainda não está em BCNF -> **repetir processo**

1. Escolher violação
   SSN -> sName, address, GPA

2. Decompor
   S5 (SSN, sName, address, GPA)
   S6 (SSN, HScode)

3. FDs e chaves

S5: FDs: SSN -> sName, address, GPA
Chave: {SSN}

S6: Chave: {SSN}

Todas as relações estão em BCNF:
S1 (HScode, HSname, HScity)
S3 (GPA, priority)
S5 (SSN, sName, address, GPA)
S6 (SSN, HScode)

Teste da caça

Testar se uma decomposição não tem perdas de informação.

Construir uma tabela, onde as colunas representam os atributos da relação original, e as linhas representam as relações decompostas.

Preencher a tabela:

- Colocar uma letra em cada célula: Se atributo não existir na relação correspondente, atribui um índice correspondente à linha (linhas vão de 1 até ao número de relações).
- De seguida, pegar nas dependências funcionais e verificar se há linhas que devem 'concordar'. Objetivo é diminuir índices.

Se no final, a tabela tiver uma linha sem índices, então a decomposição é sem perdas.

EXEMPLO
S (A, B, C, D)
Decomposições:
S1 (A, D), S2 (A,C) e S3 (B, C, D)
DFs:
A->B; A->C; CD->A

Iteração 1	A	B	C	D
S1	a	b1	c1	d
S2	a	b2	c	d2

S3	a3	b	c	d

Como A-> B, então se as linhas S1 e S2 têm o mesmo A, devem ter o mesmo B:

Iteração 2	A	B	C	D
S1	a	b1	c1	d
S2	a	b1	c	d2
S3	a3	b	c	d

Como B->C, então se as linhas S1 e S2 têm o mesmo B, devem ter o mesmo c:

Iteração 3	A	B	C	D
S1	a	b1	c	d
S2	a	b1	c	d2
S3	a3	b	c	d

Como CD->A então se as linhas S1 e S3 têm o mesmo C e mesmo D, então devem ter o mesmo A:

Iteração 4	A	B	C	D
S1	a	b1	c	d
S2	a	b1	c	d2
S3	a	b	c	d

Última linha não tem índices, logo decomposição é sem perdas.

Decompor em 3NF

- Alterar o conjunto das DFs, de forma a que os lados direitos tenham apenas um atributo (usar regra da separação)

- Para cada DF $\overline{X}$ -> $\overline{A}$, calcular o fecho de $\overline{X}$ usando as outras dependências. Se $\overline{A}$ estiver contido no fecho, então esta dependência é redundante: deve ser retirada.

- Para cada DF, remover um atributo do lado esquerdo, e calcular o fecho dos restantes atributos com as DFs originais. Se o fecho incluir o lado direito, o atributo pode ser removido.

- Voltar a combinar DFs com lados esquerdos iguais.
  Para cada DF $\overline{X}$ -> $\overline{A}$, criar relação R' ($\overline{X}$, $\overline{A}$)
  Se nenhuma das relações for super chave de R, adicionar relação com chave de R.

EXEMPLO

R (A, B, C, D, E)
DFs: AB->C, C->B e A->D

1. Encontrar base mínima
   Em todas as DFs, o lado direito só tem um elemento: já está em base mínima.

2. Verificar DFs redundantes:
   AB: A,B,D -> não contém C, logo é essencial
   C: C -> não contém B, logo é essencial
   A: A -> não contém D, logo é essencial

3. Verificar atributos redundantes:

   Apenas testar AB->C, porque restantes DFs apenas têm um elemento do lado esquerdo.
   Remover A: B->C. Fecho de B é B, como não contém C, A é essencial.
   Remover B: A->C. Fecho de A é A, como não contém C, B é essencial.

4. Criar relações:
   R1(A,B,C)
   R2(C,B)
   R3(A,D)

   Super-chaves: A,B,E e A,C,E (tem que conter E, porque não aparece no lado direito de nenhuma DF)

Por isso, criar R4(A,B,E) ou R4(A,C,E).

BCNF vs 3NF
Ambos garantem que não há perdas de informação, mas só 3NF preserva dependências.

Álgebra Relacional

Opera com relações, e retorna relações.

Operadores

Select
Retorna todos os tuplos que satisfazem uma dada condição.
Condição pode envolver =, <, <=, >, >=, <>

$\sigma_{condition} Relation$

Students with GPA>3.7
$\sigma_{GPA>3.7}\ Student$

Project
Seleciona certas colunas.

$\pi_{A_1,\ldots,A_n}\ Relation$

ID and decision of all applications
$\pi_{sID,dec}\ Apply$

Operadores de Álgebra relacional eliminam duplicados, SQL não (sem DISTINCT).
Pode ser combinado com o select:

ID and name of students with GPA>3.7
$\pi_{sID,sName}\ (\sigma_{GPA>3.7}\ Student)$

Natural Join
Junta duas tabelas, tendo em conta atributos em comum.
Tabela resultante tem atributos das duas tabelas, mas sem repetir informação.

$Student \bowtie Apply$

Theta Join
Como anterior, mas envolve uma condição.

$$Exp_1 \bowtie_\theta Exp_2 \equiv \sigma_\theta(Exp_1 \; x \; Exp_2)$$

Semi Join
Retorna tuplos de R1 que tenham um par correspondente em R2 (por exemplo, mesmo ID)

$$Exp_1 \ltimes Exp_2 \equiv \pi_{A1,\ldots,An}\,(Exp_1 \bowtie Exp_2)$$

Where $A_1, \ldots, A_n$ are atributes in Exp_1

União
Une duas tabelas, verticalmente. Ou seja, ao aplicar este operador, tabela resultante não tem mais colunas, mas mais linhas.

$$\pi_{cName} College \cup \pi_{sName} Student$$

Interseção
Como a anterior, mas tabela resultante no geral tem menos linhas.

$$\pi_{cName} College \cap \pi_{sName} Student$$

Operadores de agregação
cnt, sum, avg, max, min
Combinados com operador project.

$$\pi_{cnt(*)} \; R \quad \pi_{A,\max(B)} \; R$$

2ª imagem retorna 1 tuplo para cada valor de A, com dois atributos: o valor de A e o máximo valor de B para esse valor. Equivalente ao GROUP BY.

SQL

Restrições

NOT NULL -> coluna não tem valores NULL
DEFAULT -> quando não presente num INSERT, atributo fica com valor especificado

PRIMARY KEY -> chave primária da relação, não pode ser NULL e não pode ter valores repetidos. em SQLITE, se PK for inteiro e num insert não for especificado, então esse valor é incrementado do último existente na tabela.

Se PK tem mais que um atributo, definir no final do CREATE TABLE statement, com colunas entre parêntesis.

UNIQUE -> representa uma chave candidata, por isso não pode haver mais linhas com o mesmo valor.
Tal como PK, podemos definir um UNIQUE com mais que uma coluna.

CHECK
usado para verificar valor do atributo aquando do INSERT.

REFERENCES
declara uma chave estrangeira
Se chave tiver mais que um atributo, usar FOREIGN KEY (...) REFERENCES Table()
Pode-se omitir os nomes das colunas se forem PK.

Operadores Importantes

LIKE - para comparar conteúdo de uma string.
Exemplo: nome LIKE 'Gabriel%', onde % representa uma sequência qualquer de carateres.
DISTINCT - para eliminar resultados repetidos
INTERSECT - Interseção de SELECTs com mesmo número de colunas
UNION - união de SELECTs com mesmo número de colunas
AS - para renomear coluna do resultado da query
EXISTS / NOT EXISTS (SELECT ...) - verifica se query tem resultado ou é vazia.
IN / NOT IN (SELECT …) - verifica se algo especificado está presente na query (que deve ter o mesmo nº de colunas)
ORDER BY - ordena query pelo atributo ou conjunto de atributos especificado.
GROUP BY - usado em combinação com agregadores, agrupa resultados pelo atributo especificado.
HAVING - usado **apenas** com GROUP BY, permitindo restringir resultados através de agregadores.
… JOIN … using(...) - para juntar duas tabelas de acordo com o atributo especificado.

Agregadores

MAX(atributo) - retorna o máximo do atributo especificado
COUNT(* / atributo) - retorna o nº de elementos
AVG(atributo) - média dos valores do atributo
SUM(atributo) - soma dos valores do atributo

Gatilhos

Usados, por exemplo, para manter restrições.
Ativados quando ocorre determinado evento estabelecido na criação do gatilho.

Em SQLITE

Usar New para referir a nova linha, Old para referir antiga linha.

Estrutura de um trigger
CRIAR
QUANDO APLICAR
EVENTO
CONDIÇÃO (opcional)
BEGIN
Instruções SQL
END;

Criar: CREATE TRIGGER IF NOT EXISTS Nome
Quando aplicar: AFTER | BEFORE | INSTEAD OF
Evento: UPDATE OF, INSERT ON, DELETE FROM
Instruções SQL: Normalmente, INSERT INTO Table Values(...), DELETE FROM, UPDATE, ou quando se quer ignorar: SELECT RAISE(IGNORE).
Condição: WHEN …

Exemplo do exame 2013:

```
CREATE TRIGGER IF NOT EXISTS Vulneravel
AFTER INSERT ON AplicacaoServidor
WHEN EXISTS (SELECT * FROM Bug JOIN AplicacaoServidor USING(idAplicacao)
        WHERE idAplicacao = New.idAplicacao)
BEGIN
UPDATE Servidor
```

```
SET vulneravel = 'sim'
WHERE Servidor.idServidor = New.servidor;
END;
```

Vistas

Usar para:

- esconder informação dos utilizadores;
- tornar queries mais fáceis
- modularizar acesso à base de dados

```
CREATE VIEW Vname AS
<QUERY>
```

Uma vez criada, pode ser usada como uma tabela normal.
Vista não é guardada, quando uma query a envolve, esta é reescrita tendo em conta a sua definição: é criada tabela temporária, a query usa-a, e no final tabela é apagada.

Não se podem modificar Views como as tabelas normais (porque view não é guardada). É necessário criar um trigger do género Instead-of.

Índices

Objetivo é melhorar desempenho de uma base de dados, mais precisamente as consultas. Com índices, uma query deixa de ter que percorrer uma tabela inteira.
Índices são escondidos do utilizador.

Estruturas usadas:
B trees, hash tables

Problemas
Gastar espaço adicional
Criação
Manutenção

Transações

Motivação
- Acesso à base de dados concorrente
- Resistência a falhas de sistema

Sequência de operações tratadas como uma unidade
Objetivo é que, se ocorrer uma falha de sistema durante uma transação, as alterações implicadas pela transação são feitas ou no seu total, ou são ignoradas.

Em *'commit'*, termina uma transação e começa outra.

Propriedades ACID

Atomicidade
Cada transação nunca é deixada a meio. Se o sistema crashar e a transação não tiver sido terminada, usa-se logging para desfazer efeitos da transação – Rollback (ou abort)

Consistência
Cada transação pode assumir que as restrições são válidas quando começa, e tem que garantir que também o são quando termina.

Isolação
Operações podem ser interpoladas, mas a execução deve ser equivalente a uma ordem sequencial de transações.

Durabilidade
Se o sistema crashar após o commit de uma transação, os efeitos da transação mantêm-se na base de dados.

NoSQL

Escalabilidade vertical: estender capacidade de armazenamento e/ou potência do CPU do servidor da base de dados.
Escalabilidade horizontal: vários servidores de base de dados arranjados num conjunto.

NoSQL é preciso quando escalabilidade e disponibilidade, bem como um maior volume de informação e estruturas de dados flexíveis, são importantes.
Focam-se na escalabilidade horizontal. Distribuindo dados sobre um conjunto de *nodes*, para aumentar o desempenho e disponibilidade.

Em contraste, as bases de dados relacionais usam escalabilidade vertical.

Consistência eventual: os dados tornar-se-ão consistentes em alguma altura após cada transação.

Key-Value Store

Guardar dados como um par (Key, Value)
Chaves são únicas
Implementado com hash map, hash table ou dicionário.

Uma chave é identificada usando uma hash-function
Sharding → distribuir chaves por servidores diferentes, sendo que a hash table guarda a hash, e o servidor correspondente.

Tuple Store

Semelhante ao anterior, mas guarda uma chave e um vetor de dados.

Document Store

Guardam uma coleção de atributos categorizados, mas sem ordem necessária.
Exemplo: JSON.

Teorema CAP

Um sistema distribuído não pode garantir estas 3 propriedades ao mesmo tempo:
Consistência (todos os nodes vêm os mesmos dados ao mesmo tempo)
Disponibilidade (todos os pedidos têm uma resposta, seja de sucesso ou falha)
Tolerância a particionamento (o sistema continua a dar mesmo se *nodes* falharem ou forem adicionados outros)

Princípio BASE

Basically Available
Bases de dados NoSQL aderem ao conceito de disponibilidade do CAP

Soft state
O sistema pode mudar ao longo do tempo, mesmo sem receber input

Eventual consistency
O sistema ficará consistente ao longo do tempo