Banco de dados NoSQL - Cassandra (Modelagem)

Prof. Gustavo Leitão

- O grande segredo para um bom desempenho está na modelagem adequada dos dados

- Cassandra propõe que os dados sejam modelados de maneira não normalizada onde o modelo dos dados seja definido de acordo com as consultas necessárias

- Com os dados bem modelados tiraremos proveito das vantagens do cassandra: alta escalabilidade, replicação dos dados e velocidade.

- Premissas a **não** serem seguidas

  - **Minimizar número de escritas:** Cassandra é otimizado para alta vazão de escrita. As escritas no calandra tem basicamente o mesmo custo. Par melhorar a leitura é normal criar várias escritas

  - **Minimizar Duplicação:** Denormalização e duplicação é um fato para o cassandra. Não tenha medo disso! Espaço em disco normalmente é o recurso mais barato entre CPU, memória e rede. Cassandra é projetado em torno desse fato. Para obter leituras eficientes muitas vezes você terá que duplicar!

- Premissas a serem seguidas
    - **Regra 1 - Espalhe dados ao longo do cluster**
        - Os dados são distribuídos no cluster através da chave de partição (PartitionKey). Escolhe bem a chave de partição para possibilitar boa distribuição dos dados.
- **Regra 2 - Minimize o número de partições a serem lidas:**
    - Partições são grupos que compartilham a mesma chave de partição. Otimize o modelo para que as consultas consulte o menor número de nós no cluster.

MÉTODO

- **Passo 1:** Determine as consultas que serão necessárias.

- **Passo 2:** Tente criar uma tabela que satisfaça sua consulta lendo o mínimo de partições. Em geral isso significa utilizar o padrão uma tabela por consulta!

Exemplo 1

Busca de usuários

Localizar usuários por nome ou e-mail

Modelo proposto 1:

```
CREATE TABLE users (
    id uuid PRIMARY KEY,
    username text,
    email text,
    age int
)

CREATE TABLE users_by_username (
    username text PRIMARY KEY,
    id uuid
)

CREATE TABLE users_by_email (
    email text PRIMARY KEY,
    id uuid
)
```

Dados espalhados? SIM
Leitura em única partição? NÃO

Busca de usuários

Localizar usuários por nome ou e-mail

Modelo proposto 2:

```
CREATE TABLE users_by_username (
    username text PRIMARY KEY,
    email text,
    age int
)

CREATE TABLE users_by_email (
    email text PRIMARY KEY,
    username text,
    age int
)
```

Dados espalhados? SIM
Leitura em única partição? SIM

Busca de usuários

Username: gustavo, email: gustavo@im

Localizar usuários por nome ou e-mail

P1 P3

Modelo proposto 2:

```
1   CREATE TABLE users_by_username (
2       username text PRIMARY KEY,
3       email text,
4       age int
5   )
6
7   CREATE TABLE users_by_email (
8       email text PRIMARY KEY,
9       username text,
10      age int
11  )
```

P2

Dados espalhados? SIM

Leitura em única partição? SIM

Exemplo 2

Busca de usuários por grupo

Usuários são agora agrupados em grupos e a consulta deve retornar todos usuários de um grupo

Modelo proposto 1:

```
CREATE TABLE users (
    id uuid PRIMARY KEY,
    username text,
    email text,
    age int
)

CREATE TABLE groups (
    groupname text,
    user_id uuid,
    PRIMARY KEY (groupname, user_id)
)
```

Modelo diminui duplicação. Porém a um alto preço: se tivermos 100 usuários em um grupo teremos que ler 101 partições!

Dados espalhados? Depende
Leitura em única partição? Não

Busca de usuários por grupo

Usuários são agora agrupados em grupos e a consulta deve retornar todos usuários de um grupo

Modelo proposto 2:

```
CREATE TABLE groups (
    groupname text,
    username text,
    email text,
    age int,
    PRIMARY KEY (groupname, username)
)
```

Se tivermos poucos grupos com muitos usuários esse modelo pode se tornar ineficiente

Dados espalhados? Depende.
Leitura em única partição? SIM

Busca de usuários por grupo

Usuários são agora agrupados em grupos e a consulta deve retornar todos usuários de um grupo

Modelo proposto 3:

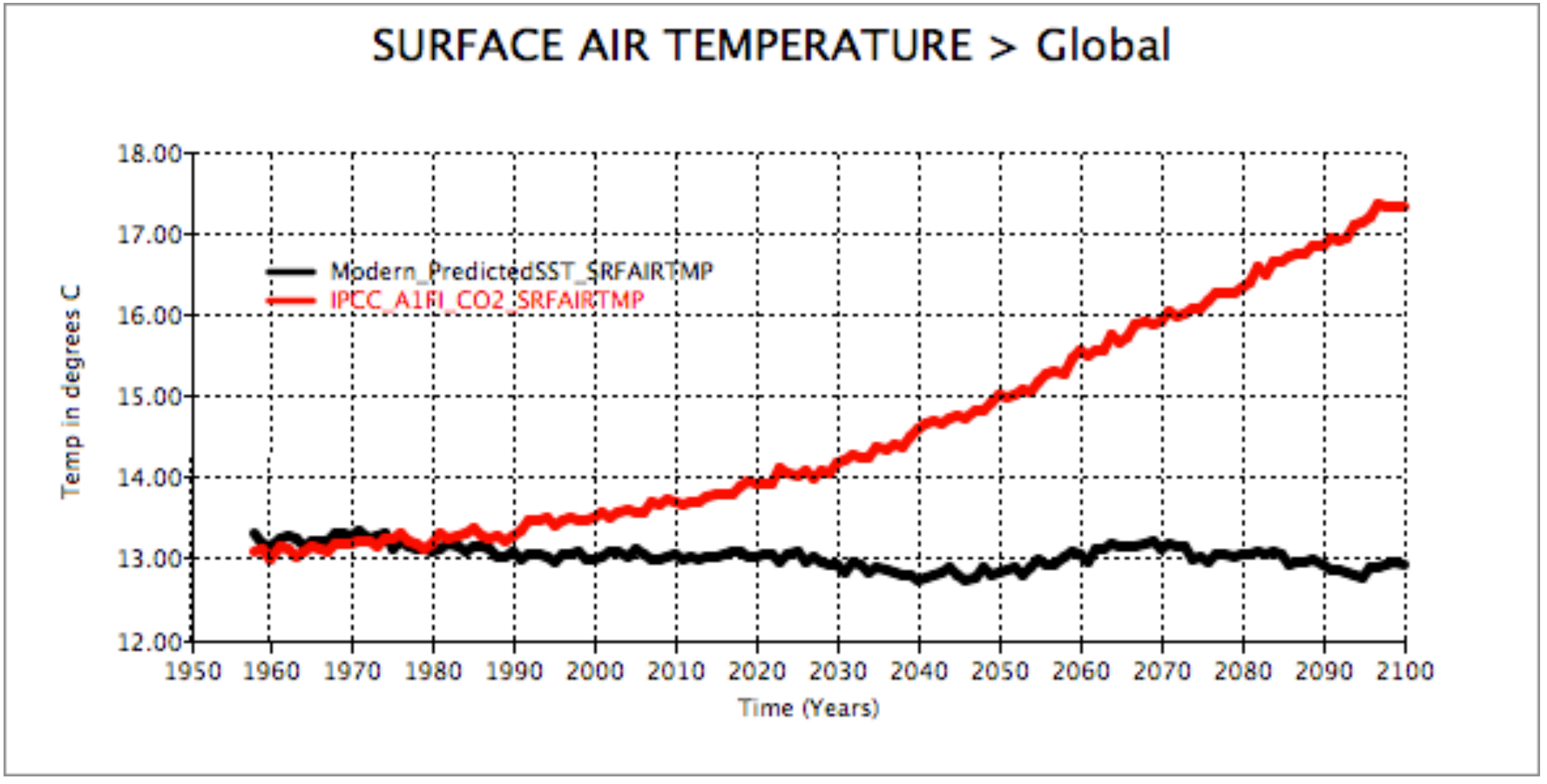
```
CREATE TABLE groups (
    groupname text,
    username text,
    email text,
    age int,
    hash_prefix int,
    PRIMARY KEY ((groupname, hash_prefix), username)
)
```

hash_prefix tem função de diminuir a concentração dos dados

Dados espalhados? SIM
Leitura em única partição? SIM

Exemplo 3

Séries Temporais



Armazenam o comportamento de variáveis ao longo do tempo

Modelo básico

```
CREATE KEYSPACE bigdata
WITH durable_writes = true
AND replication = {
   'class' : 'SimpleStrategy',
   'replication_factor' : 1
};
```

Modelo básico

```
create table bigdata.timeseries (
   tag text,
   descricao text,
   tipo int,
   data timestamp,
   valor double,
   PRIMARY KEY (tag, data)
)
WITH CLUSTERING ORDER BY (data ASC);
```

Chave primária: tag e data

Chave partição: tag

Clustering Column: data

Inserindo Dados

```
INSERT INTO bigdata.timeseries(tag,descricao,tipo,data, valor)
values ('temp-002', 'Temperatura forno 2', 1, '2017-02-01 00:00:01', 11.4);

INSERT INTO bigdata.timeseries(tag,descricao,tipo,data, valor)
values ('temp-002', 'Temperatura forno 2', 1, '2017-02-01 00:00:02', 12.0);

INSERT INTO bigdata.timeseries(tag,descricao,tipo,data, valor)
values ('temp-002', 'Temperatura forno 2', 1, '2017-02-01 00:00:03', 11.3);

INSERT INTO bigdata.timeseries(tag,descricao,tipo,data, valor)
values ('temp-002', 'Temperatura forno 2', 1, '2017-02-01 00:00:04', 10.7);

INSERT INTO bigdata.timeseries(tag,descricao,tipo,data, valor)
values ('temp-002', 'Temperatura forno 2', 1, '2017-02-01 00:00:05', 8.0);

INSERT INTO bigdata.timeseries(tag,descricao,tipo,data, valor)
values ('temp-002', 'Temperatura forno 2', 1, '2017-02-01 00:00:06', 72.0);
```

Consultando

```
//consultando todos
select * from bigdata.timeseries;

//contando elementos
select count(*) from bigdata.timeseries;
```

```
//consultando dados de uma tag
select * from bigdata.timeseries where tag =
'temp-001';

//ordenando resultado
select * from bigdata.timeseries where tag = 'temp-001'
order by data desc;
```

Consultando

Limitações do modelo

```
//Ordenar por tag sem definição de partkey
select * from bigdata.timeseries order by data desc;
```

ORDER BY só é permitido em uma clustering column e quando a partkey está definida no WHERE com operadores IGUAL

Solução

```
select * from bigdata.timeseries where tag = 'temp-001'
order by data desc;
```

Consultando

Limitações do modelo

```
//filtrando por campo que não estejam na primary/partkey
select * from bigdata.timeseries where tipo >= 1;
```

Só permitido por cláusula WHERE em campos da chave primária.

Solução 1 (Adicionar ALLOW FILTERING)

```
//filtrando por campo que não estejam na primary/partkey
select * from bigdata.timeseries where tipo >= 1 ALLOW
FILTERING;
```

Solução 2 (Criar índice secundário)

```
CREATE INDEX ON bigdata.timeseries(tipo)
```

Melhorando o modelo

```
create table bigdata.timeseries (
   tag text,
   descricao text,
   ano int,
   tipo int,
   data timestamp,
   valor double,
   PRIMARY KEY ((tag, ano) data)
)
WITH CLUSTERING ORDER BY (data ASC);
```

Supera o problema do número máximo de colunas por partição: 2 bilhões

Materialized Views

(Visões Materializadas)

VISÕES MATERIALIZADAS

- Criada a partir do cassandra 3.0 para facilitar processe de denormalização dos dados

- Cria uma nova estrutura de armazenamento para os dados a partir de uma tabela de fonte de dados.

- As views só ficam disponíveis para leitura

- Alternativa aos índices secundários para atributos de alta cardinalidade, evitando assim, acesso a múltiplas partições.

VISÕES MATERIALIZADAS

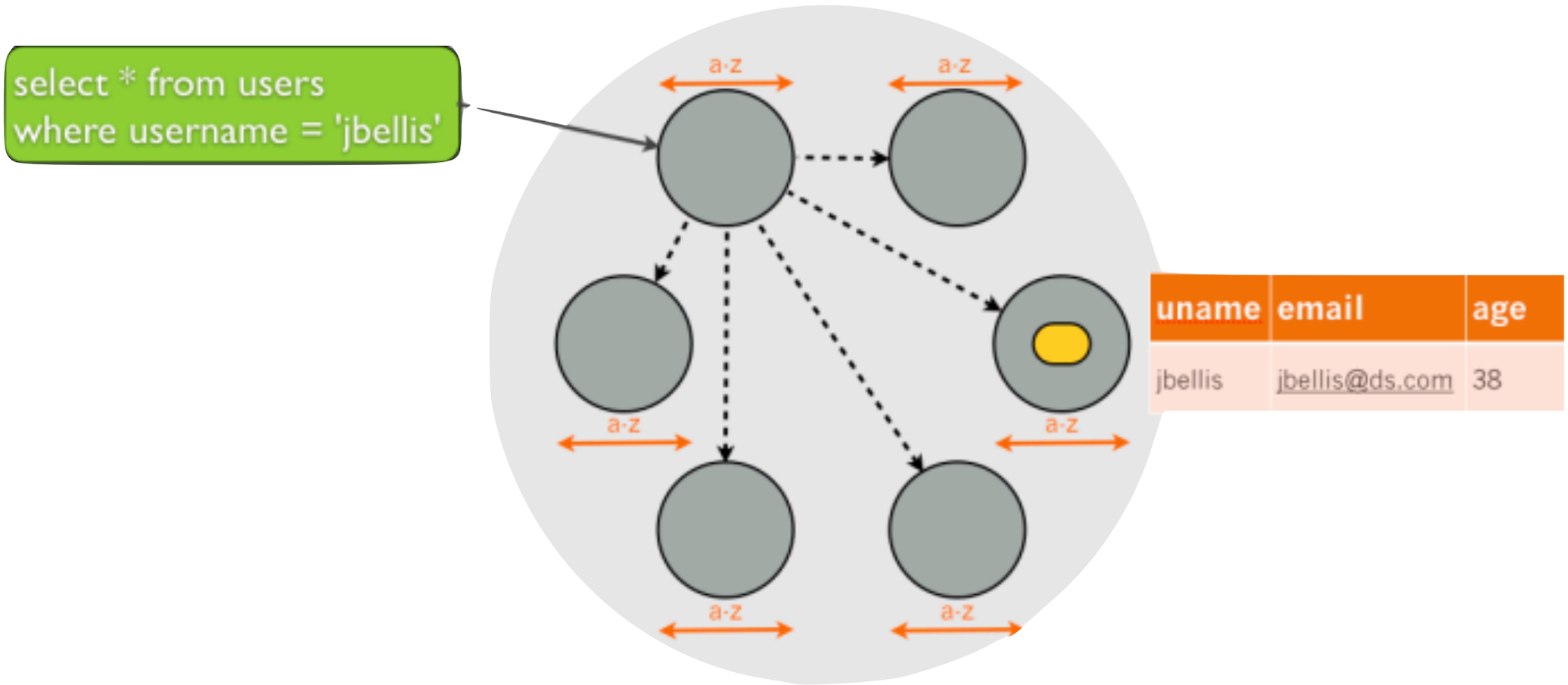
Dado o seguinte modelo

```
CREATE TABLE users (
    id uuid PRIMARY KEY,
    username text,
    email text,
    age int
);
```

Caso seja necessário acessar um dado usuário pelo username teremos que criar um índice

```
CREATE INDEX users_by_name ON users (username);
```

VISÕES MATERIALIZADAS



O índice reside localmente em cada nó. Assim, ainda será necessário realizar acesso a vários nós para responder a essa consulta

VISÕES MATERIALIZADAS

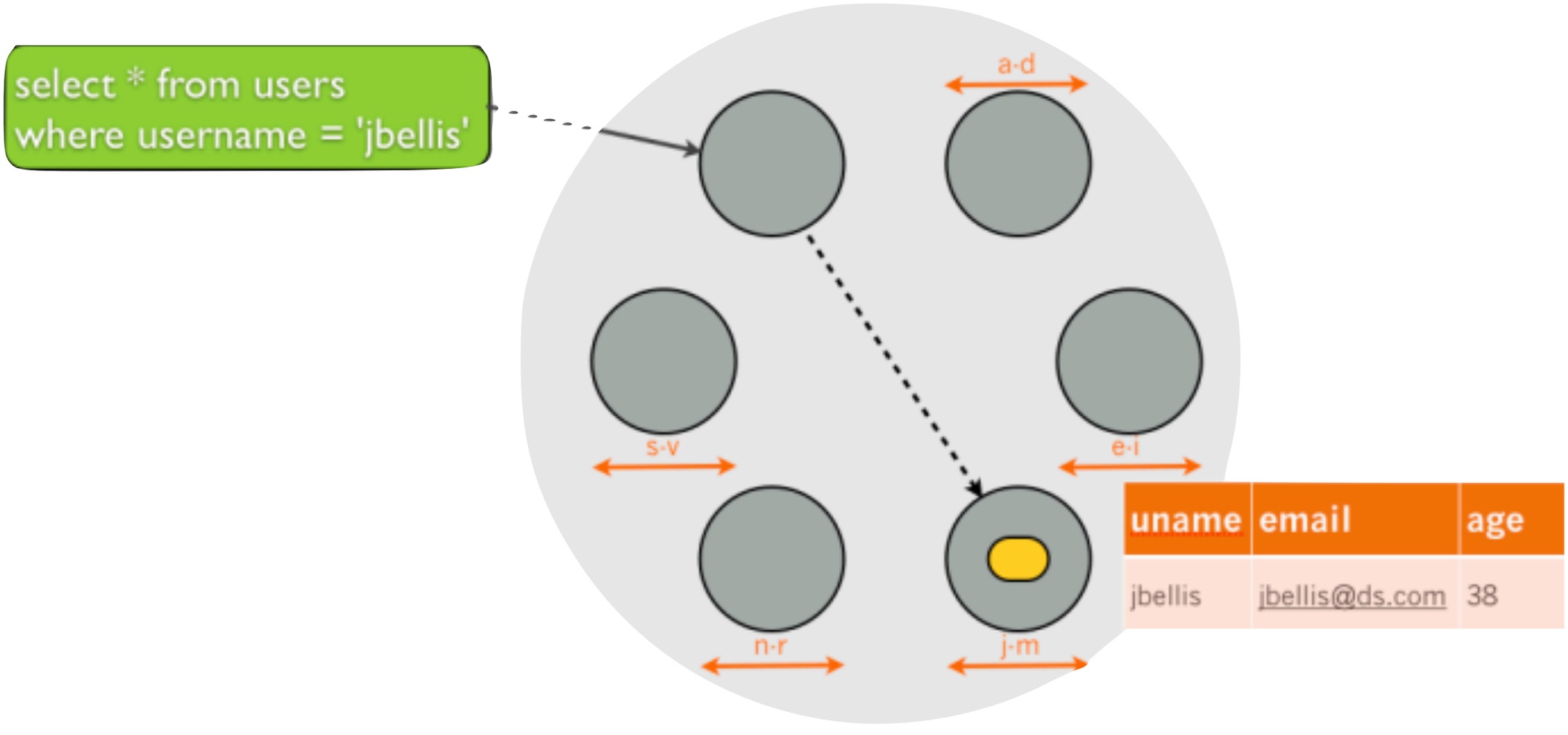
Alternativa é criar uma tabela onde o usuário é chave

```
CREATE TABLE users_by_name (
    username text PRIMARY KEY,
    id uuid
);
```

Porém será de responsabilidade da aplicação manter essa tabela sincronizada com a tabela de usuários.

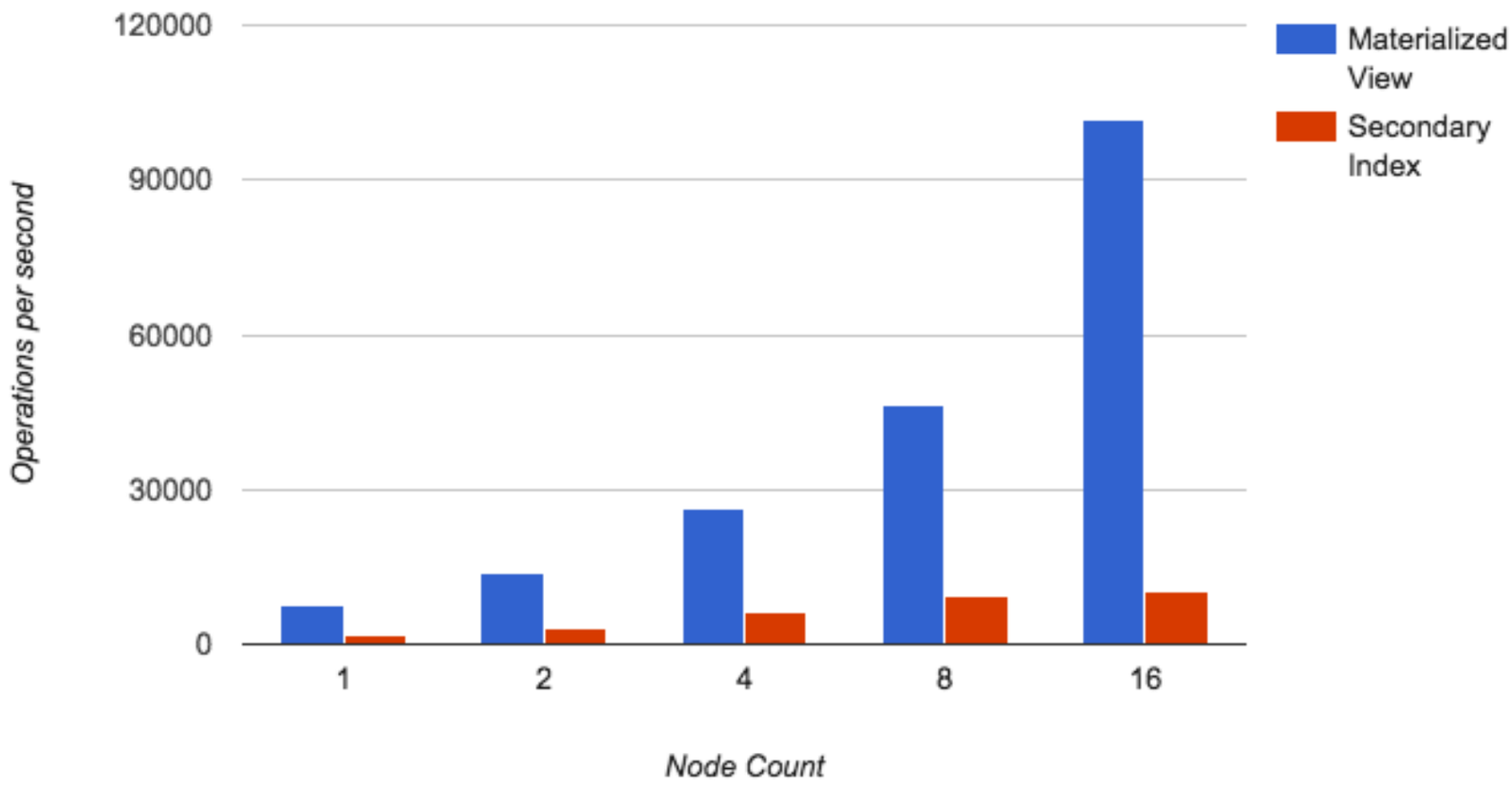
Alternativa:

```
CREATE MATERIALIZED VIEW users_by_name AS
SELECT * FROM users
WHERE username IS NOT NULL
PRIMARY KEY (username, id);
```



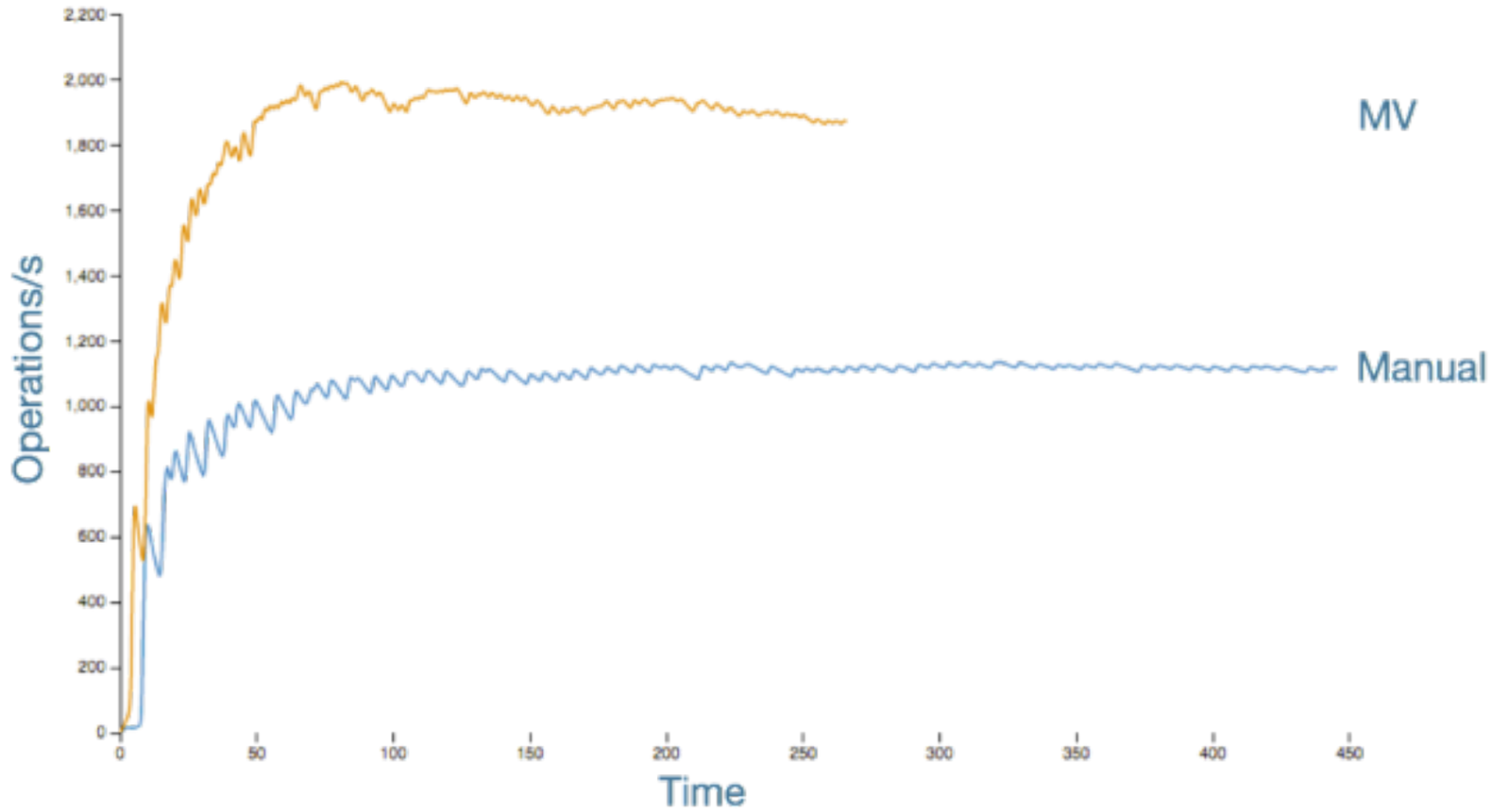
VISÕES MATERIALIZADAS

Desempenho comparativo entre índices secundários e views materializadas



VISÕES MATERIALIZADAS

Denomarlização manual vs Views Materializadas



Cenário: 3 nós, RF=3. Carga de escrita de dados

Outros tipos de dados

Tipos de dados

Tipo	Descrição
ascii	Representa caracteres no formato ASCII
bigint	Inteiro de 64 bits
blob	Sequencia de bytes
Boolean	representa true ou false
counter	Representa um número que só pode ser alterado
decimal	Representa números de ponto flutuante de precisão variável
double	Ponto flutuante de 64 bits
float	Ponto flutuante de 32 bits
inet	Representa IP (ipv4 ou ipv6)
int	Inteiro de 32 bits
text	Texto em formato UTF-8
timestamp	Representa um instante no tempo
timeuuid	Representa um instante no tempo livre de colisão
uuid	Identificador universal único
varchar	Mesmo que text
varint	Inteiro de precisão arbitrária

Coleções

- Além dos tipos básicos apresentados, o Cassandra possui tipos de dados de coleção:

- **LIST** - Lista de algum tipo básico. Permite armazenar listas de elementos ordenados por inserção

- **SET** - Lista única de tipo básico. Permite armazenar elementos sem repetição.

- **MAP** - Coleção de objetos chave/valor

Coleções

LIST

```
select * from bigdata.list_data;
```

```
UPDATE  bigdata.list_data
   SET email = email +['gustavo9@logiquesistemas.com']
   WHERE name = 'gustavo';
```

```
UPDATE  bigdata.list_data
   SET email = email -['guga@dca.ufrn.br']
   WHERE name = 'gustavo';
```

Coleções

LIST

```
CREATE TABLE bigdata.list_data(
   name text PRIMARY KEY,
    email list<text>
);

INSERT INTO bigdata.list_data(name, email) VALUES
('gustavo',
['gustavo@dca.ufrn.br','guga@dca.ufrn.br']);

select * from bigdata.list_data;
```

Coleções

LIST

```
SELECT * from bigdata.list_data where email contains
'lala@dca.ufrn.br' ALLOW FILTERING;
```

```
CREATE index on bigdata.list_data(email);
```

```
SELECT * from bigdata.list_data where email contains
'lala@dca.ufrn.br';
```

Coleções

SET

```
CREATE TABLE  bigdata.set_data (
   name text PRIMARY KEY,
   phone set<varint>
);

INSERT INTO bigdata.set_data(name, phone) VALUES
('gustavo',{8499988745,8496852314});

select * from bigdata.set_data;
```

Coleções

SET

```
UPDATE bigdata.set_data
    SET phone = phone + {8399562514}
    where name = 'gustavo';
```

```
UPDATE bigdata.set_data
    SET phone = phone - {84999988745}
    where name = 'gustavo';


select * from bigdata.set_data;
```

Coleções

SET

```
SELECT * FROM bigdata.set_data where phone contains
8496852314 ALLOW FILTERING;
```

```
CREATE INDEX ON bigdata.set_data(phone);
```

```
SELECT * FROM bigdata.set_data where phone contains
8496852314;
```

Coleções

MAP

```
CREATE TABLE bigdata.map_data (
   name text PRIMARY KEY,
   address map<text, text>
);

INSERT INTO bigdata.map_data (name, address)
   VALUES ('gustavo', {'home' : 'Av. Dos Anzóis, 1457' ,
'office' : 'Rua Dr. Barata, 2541' } );


select * from bigdata.map_data;
```

Coleções

MAP

```
UPDATE bigdata.map_data
  SET address = address + {'office':'Rua Dr. Barata, 111'}
  WHERE name = 'gustavo';
```

```
 UPDATE bigdata.map_data SET address['office'] = ‘Av.Xavantes,
1010’ WHERE name = ‘gustavo';
```

```
 UPDATE bigdata.map_data
   SET address = address - {'office'}
   WHERE name = 'gustavo';
```

Coleções

MAP

```
SELECT * FROM bigdata.map_data WHERE address CONTAINS
KEY 'office' ALLOW FILTERING;
```

```
CREATE INDEX ON bigdata.map_data (KEYS(address));
```

```
SELECT * FROM bigdata.map_data WHERE address CONTAINS
KEY 'office';
```

Exercício

Exercício 1

- Crie um modelo de dados seguindo os princípios do Cassandra para armazenar músicas.
    - Cada música deve armazenar: um título, o album e o artista.
    - As seguintes consultas serão necessárias:
        - Lista de musica por artista
        - Lista de música por título
        - Lista de musica por album
- Ao final deve ser feito o código de criação das tabelas/views, inserção de dados e consultas.

Exercício 2

- Adicione o modelo anterior a possibilidade de criar playlists
    - Cada playlist deve armazenar: titulo, ordem da musica, lista de tags e a musica.
- As seguintes consultas serão necessárias:
    - Lista de musicas ordenadas de uma dada playlist
    - Lista de playlists com uma determinada tag

Banco de dados NoSQL - Cassandra (Modelagem)

Prof. Gustavo Leitão