UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

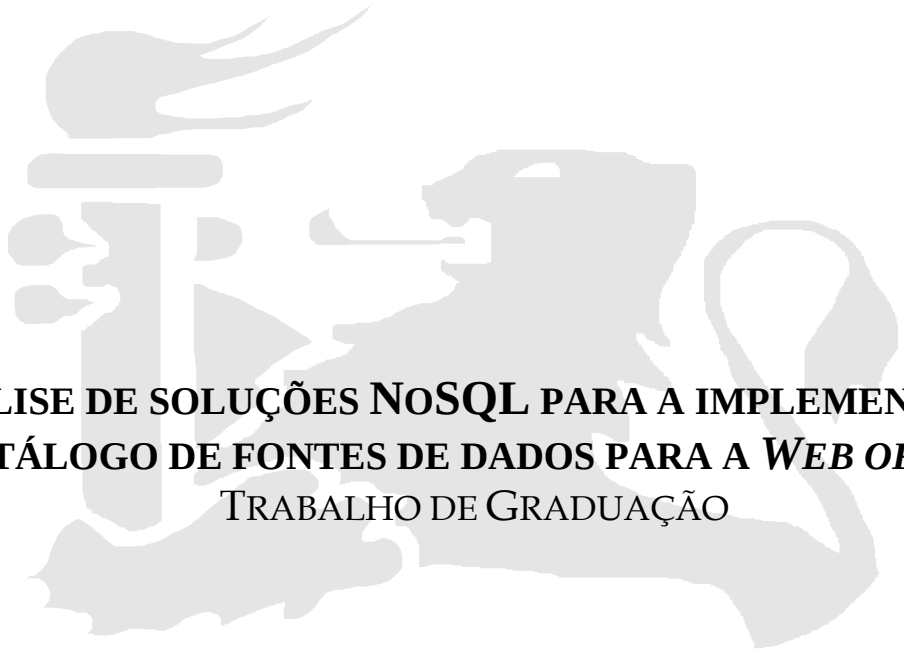
ALBERTO TRINDADE TAVARES

ANÁLISE DE SOLUÇÕES NOSQL PARA A IMPLEMENTAÇÃO DE CATÁLOGO DE FONTES DE DADOS PARA A *WEB OF THINGS*

TRABALHO DE GRADUAÇÃO

RECIFE
2015

ALBERTO TRINDADE TAVARES



ANÁLISE DE SOLUÇÕES NOSQL PARA A IMPLEMENTAÇÃO DE CATÁLOGO DE FONTES DE DADOS PARA A *WEB OF THINGS*

TRABALHO DE GRADUAÇÃO

Trabalho de Graduação apresentado à graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Bernadette Farias Lóscio (bfl@cin.ufpe.br)

Co-orientador: Marcelo Iury S. Oliveira (miso@cin.ufpe.br)

RECIFE
2015

ALBERTO TRINDADE TAVARES

ANÁLISE DE SOLUÇÕES NOSQL PARA A IMPLEMENTAÇÃO DE CATÁLOGO DE FONTES DE DADOS PARA A *WEB OF THINGS*

TRABALHO DE GRADUAÇÃO

Trabalho de Graduação apresentado à graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco para obtenção do grau de Bacharel em Ciência da Computação.

Recife, ____ de julho de 2015.

BANCA EXAMINADORA

_____

Prof.ª Bernadette Farias Lóscio
(Orientadora)

_____

Prof.º Kiev Santos da Gama
(Avaliador)

AGRADECIMENTOS

Primeiramente, eu gostaria de agradecer à minha família por todo o apoio incondicional que me foi dado em todos os meus passos da vida acadêmica. Todo o incentivo que recebi foi fundamental para eu ter alcançado vários dos meus objetivos, incluindo a realização deste trabalho.

Também agradeço à minha namorada e aos meus amigos – do colégio, da universidade e de tantos outros lugares – pelo companheirismo e momentos de descontração que foram muito importantes durante a minha trajetória na universidade.

Agradeço a Marcelo, o meu co-orientador, pela valiosa ajuda no desenvolvimento dessa pesquisa, me guiando por todo o processo, desde a concepção do projeto até a execução de experimentos.

Por último, mas não menos importante, gostaria de agradecer a Bernadette, a minha orientadora tanto nesse trabalho quanto em outros que eu desenvolvi durante os meus anos de iniciação científica. A sua orientação em todos esses projetos me ensinou muitas coisas que vou levar para o resto da minha carreira.

RESUMO

O advento da *Web of Things* (WoT) trouxe novos desafios para a coleta, integração e processamento de dados. Entre esses desafios, destacam-se o volume massivo, heterogeneidade e autonomia das fontes de dados. Dessa forma, torna-se necessária a existência de mecanismos eficientes para a publicação e descoberta de produtores de dados da WoT. Uma das soluções para o desenvolvimento de um serviço de descoberta é o registro de metadados de descrição de produtores de dados em um catálogo, que deve ser flexível, para lidar com a heterogeneidade das fontes, e escalável, para lidar com o grande volume de produtores e consumidores de dados. Este trabalho tem o objetivo de implementar e avaliar diferentes soluções de catálogo de fontes de dados para a WoT que faz uso de bancos de dados NoSQL.

ABSTRACT

The rise of Web of Things (WoT) has created new challenges for data collection, integration and processing. The main challenges are the large volume, heterogeneity and autonomy of data sources. Therefore, we need an efficient mechanism for publishing and discovering data producers on the WoT. One possible solution for the development of a discovery service is the storage of metadata for data sources into a catalog. That catalog must be flexible, to deal with heterogeneity, and scalable, to deal with massive volume of data producers and consumers. In this work, we implement and evaluate four solutions of data source catalog for the WoT, which are built on top of NoSQL databases.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Este capítulo fornecerá uma contextualização sobre os conceitos abordados neste trabalho e a motivação para o desenvolvimento do projeto. Além disso, os principais objetivos deste trabalho serão apresentados, assim como a estrutura do restante do documento.

1.1. Contextualização e Motivação

O volume de dados produzidos e compartilhados através da *Web* está crescendo rapidamente. Esses dados incluem desde informações textuais até conteúdo multimídia, em uma grande variedade de plataformas. Um dos tipos de dados que vem se destacando são os relacionados a observações físicas do mundo real. O aumento da coleta de dados do mundo físico se deve aos avanços recentes das tecnologias de comunicação com o objetivo de fornecer acesso ubíquo à Internet, assim como a redução do custo de sensores e dispositivos móveis. A extensão da Internet atual, permitindo a comunicação entre objetos e dispositivos físicos, é descrita pelo termo *Internet of Things* (IoT) [1, 2].

A *Internet of Things* consiste em uma rede de objetos interconectados que não apenas consomem informações do ambiente e interagem com o mundo físico, mas também utilizam os padrões de comunicação da Internet para fornecer serviços [3]. Esses objetos podem ser dispositivos equipados com sensores e etiquetas de identificação por radiofrequência (RFID), permitindo a monitoração do ambiente. Tais dispositivos podem atuar tanto como produtores de dados, assim como consumidores de dados [4].

Nesse contexto, surge o conceito de *Web of Thing*s (WoT) que, além de fazer uso da Internet como veículo de troca de dados, permite a interação entre dispositivos, dados e pessoas na *Web* [5]. O principal objetivo da WoT é aplicar a arquitetura da *Web* para conectar recursos físicos, tais como sensores, com outros recursos da *Web*. Dessa forma, objetos do mundo real passarão a ser tratados da mesma forma que qualquer outro recurso Web, sendo identificados por uma URI (*Universal Resource Identifier*) e acessíveis através de protocolos de comunicação, como o HTTP (*Hypertext Transfer Protocol*) [6].

O advento da WoT trouxe novos desafios para a coleta, integração e processamento de dados do mundo físico na *Web*. Entre os desafios, destaca-se o volume massivo de dispositivos conectados à Internet. Como a quantidade de recursos que podem atuar como produtores de dados é crescente, escolher o produtor mais adequado para o fornecimento de uma determinada informação se torna uma tarefa complexa [4]. Além disso, a heterogeneidade, autonomia e ausência de esquemas globais nas fontes de dados trazem dificuldades adicionais. Os recursos

da WoT podem ser ubíquos e geralmente são limitados em termos de capacidade de armazenamento, processamento e comunicação [1]. Diante desses problemas, torna-se necessária a existência de mecanismos eficientes para a publicação e descoberta de produtores de dados da WoT.

Uma das soluções para o desenvolvimento de um serviço de descoberta é o registro de metadados de descrição dos produtores em um catálogo [7]. Este catálogo pode incluir informações de acesso e estrutura sobre qualquer tipo de fonte de dados da WoT, desde dispositivos físicos até *datasets* da *Linked Open Data*. O catálogo de fontes deve ser flexível, para lidar com a heterogeneidade das fontes, e escalável, para lidar com o potencial volume massivo de fontes e consumidores de dados.

Usualmente, os catálogos são desenvolvidos através de Sistemas de Gerenciamento de Banco de Dados (SGBD) para a manipulação e persistência de seus dados. Desta forma, as características e configurações de SGBDs podem impactar diretamente o desempenho geral do catálogo. Em especial, o uso de mecanismos de indexação, particionamento, visões e *cache* podem ser fundamentais para o atendimento de níveis ideais de desempenho. Devido aos requisitos de escalabilidade e a não obrigatoriedade de utilização de um esquema comum para todos os registros, os bancos de dados NoSQL se adequam melhor para a o desenvolvimento de catálogos de fontes de dados para a WoT.

1.2. Objetivos

Este trabalho de graduação tem como objetivo geral a análise de desempenho de diferentes soluções de banco de dados NoSQL para o armazenamento de metadados sobre fontes da WoT, implementado por meio de um catálogo. Como objetivos específicos deste trabalho, destacamos as seguintes atividades:

i. Implementação de catálogos de produtores de dados a partir de:
    a. Dois SGBDs NoSQL: OrientDB[1], utilizando o modelo orientado a grafos, e MongoDB[2], utilizando o modelo orientado a documentos;
    b. Dois esquemas de representação de fontes de dados: SensorML[3] e WoTDataSchema, desenvolvido de forma paralela a este trabalho como uma alternativa para a especificação de produtores de dados da WoT.

[1] http://orientdb.com/
[2] https://www.mongodb.org/
[3] http://www.opengeospatial.org/standards/sensorml

ii. Desenvolvimento de ambiente de simulação e execução de experimentos com as diferentes soluções de catálogo e configurações do SGBD, tais como o uso de índices e *cache*;

iii. Avaliação dos resultados obtidos nos experimentos.

Em suma, esse trabalho busca, através da avaliação do desempenho e robustez de diferentes implementações de catálogos, responder a seguinte pergunta de pesquisa:

Qual é a melhor solução para o desenvolvimento de catálogos para a WoT?

1.3. Estrutura do Documento

O restante deste trabalho será organizado da seguinte forma:

- Capítulo 2: Apresenta um breve histórico sobre os sistemas de gerenciamento de dados, até chegarmos aos bancos de dados NoSQL, além de explicar as características do NoSQL, que justificam o seu uso no contexto da *Web of Things*, e os diferentes modelos de dados que podem ser implementados.
- Capítulo 3: Descreve as soluções de desenvolvimento de catálogos que são avaliadas neste trabalho, abordando esquemas de dados para representar recursos da WoT, bem como os diferentes bancos de dados NoSQL utilizados.
- Capítulo 4: Descreve os experimentos realizados com os catálogos desenvolvidos a partir de um ambiente de simulação e analisa os resultados obtidos.
- Capítulo 5: Dá uma visão geral das principais conclusões deste trabalho, bem como fornece indicações sobre possíveis contribuições futuras.

2. BANCOS DE DADOS NOSQL

Neste capítulo, iremos apresentar os fundamentos da tecnologia de banco de dados NoSQL. O capítulo inicia com um breve histórico sobre as soluções para armazenamento de dados, trazendo a motivação para o surgimento do NoSQL. A seguir, serão descritas as características de bancos de dados NoSQL, que os tornam apropriados para o desenvolvimento de catálogos no contexto da WoT. Por fim, iremos apresentar os principais modelos de dados NoSQL, dos quais dois são implementados nas soluções de catálogos avaliadas neste trabalho.

2.1. Histórico dos Sistemas de Gerenciamento de Bancos de Dados

A partir de meados da década de 1960, foram criados os primeiros Sistemas de Gerenciamento Banco de Dados (SGBD), ainda de forma primitiva, misturando relacionamentos conceituais com o armazenamento de registros em discos [8]. Os principais problemas desses sistemas eram a incapacidade para abstração de dados, independência entre dados e programa e flexibilidade para acesso de registros. Dentre esses primeiros sistemas, se destacam três principais paradigmas: sistemas hierárquicos, sistemas baseados em modelos de rede e sistemas de arquivos invertidos [8].

No início dos anos 70, surgiram os bancos de dados relacionais, propostos originalmente para separar o armazenamento físico dos dados da sua representação conceitual, que se estabeleceram como solução comercial para o gerenciamento de dados que possuem uma estrutura fixa e bem definida [8, 9]. No modelo relacional, os dados são armazenados em tabelas, onde cada linha corresponde a um registro, também chamado de tupla, e cada coluna define uma propriedade do registro, chamada de atributo. Um dos motivos do grande sucesso dos Sistemas de Gerenciamento de Banco de Dados Relacional (SGBDR) foi a linguagem de consulta desenvolvida para o modelo relacional, o SQL (*Structured Query Language*), que permite a realização de consultas e manipulação de dados a partir de múltiplas tabelas [9].

No final da década de 1980, o desenvolvimento de linguagens de programação orientadas a objetos e a necessidade de armazenar objetos complexos levaram a criação de Bancos de Dados Orientados a Objetos (BDOO). Embora esses bancos de dados incorporassem conceitos úteis do paradigma de orientação a objetos, tais como tipos de dados abstratos e herança, o seu uso foi bastante limitado, devido a alguns fatores, como a complexidade do modelo. Também incorporando conceitos orientados a objetos, surgiram os Sistemas de Gerenciamento de Bando de Dados Objeto-Relacionais (SGBDOR), como extensão dos bancos de dados puramente relacionais [8].

Atualmente, os sistemas de banco de dados convencionais, em especial o relacional, são baseados em um conjunto de princípios que define como as transações devem ser executadas a fim de garantir a consistência dos dados. Esses princípios são denominados ACID, propriedades aos quais as transações dos SGBDs devem obedecem e que são definidas a seguir [9, 10]:

- ***A**tomicity* (atomicidade): uma transação é concluída somente quando todas as operações da mesma são executadas com sucesso, ou seja, se uma das operações falhar, nada é executado;
- ***C**onsistency* (consistência): uma transação não pode executar uma operação que torne o banco de dados inconsistente, ou seja, após a conclusão de uma transação, todas as condições de consistência e restrições de integridades existentes são satisfeitas;
- ***I**solation* (isolamento): todas as transações são independentes e uma não pode interferir em outra que esteja sendo executada concorrentemente;
- ***D**urability* (durabilidade): quando uma transação é concluída sem falhas, os seus efeitos não podem ser desfeitos.

Com o surgimento e crescimento da *Web*, foram estabelecidos novos requisitos de bancos de dados, como a necessidade de gerenciar grandes volumes de dados semiestruturados e não estruturados, além de novas exigências de disponibilidade e escalabilidade, tornando os princípios ACID difíceis de se alcançar [9, 10]. Dessa forma, com o objetivo de satisfazer esses novos requisitos, novas soluções para o gerenciamento de dados foram desenvolvidas, dentre essas, os bancos de dados NoSQL.

NoSQL (*Not Only SQL*) é usado atualmente como um termo que faz referência a todos os sistemas de bancos de dados que não seguem o modelo relacional e são mais flexíveis quanto aos princípios ACID [9, 11]. Essa flexibilidade se deve aos requisitos de alta disponibilidade e escalabilidade para manipular grandes volumes de dados, características essenciais em diversas aplicações da *Web* 2.0 [9]. Em contraste ao ACID, os bancos de dados NoSQL têm o foco nos princípios BASE [10]:

- ***B**asically **A**vailable* (basicamente disponível): os dados são distribuídos de tal forma que o sistema continue funcionando, mesmo que haja uma falha;
- ***S**oft state* (estado flexível): não há garantia de consistência dos dados;
- ***E**ventually consistent* (eventualmente consistente): o sistema garante que caso os dados não estejam consistentes, eventualmente o banco de dados passará para um estado consistente.

2.2. Características dos Bancos de Dados NoSQL

À medida que o volume do conteúdo da *Web* cresce e as fontes de dados se tornam cada vez mais heterogêneas, os seguintes desafios para os sistemas de bancos de dados são potencializados [11]:

i. Armazenar e acessar de forma eficiente grandes volumes de dados, tendo que lidar com demandas adicionais de tolerância a falhas;
ii. Recuperar-se de falhas durante a execução de um processo, entre vários que estejam rodando de forma paralela, e fornecer resultados rapidamente;
iii. Gerenciar a contínua evolução de esquemas e metadados para dados semiestruturados e não estruturados gerados por diversas fontes.

Os bancos de dados NoSQL apresentam importantes características que os SGBDs relacionais não possuem, tornando-os mais adequados para enfrentar esses desafios, em especial, o armazenamento de grandes quantidades de dados não estruturados ou semiestruturados, um cenário comum quando lidamos com fontes da *Web of Things*. Descreveremos, a seguir, tais características [4, 9, 11, 12]:

a. Escalabilidade horizontal

Com o cenário de crescimento da quantidade de dados, a escalabilidade e melhoria de desempenho do sistema se tornam mais relevantes. Escalabilidade diz respeito à capacidade de um sistema de aumentar a vazão de dados com a inserção de novos recursos para lidar com o aumento de carga. A escalabilidade pode ser obtida de duas formas: i) Escalabilidade vertical e ii) Escalabilidade horizontal.

A escalabilidade vertical consiste em aumentar o poder de processamento e armazenamento das máquinas. As opções de escalabilidade vertical geralmente são caras e fazem uso de soluções proprietárias. Por outro lado, a escalabilidade horizontal envolve o uso de múltiplos nós que são acomodados em *clusters*, ocorrendo o aumento no número de servidores disponíveis para o processamento e o armazenamento de dados. Dessa maneira, o *cluster* escala na medida em que a carga aumenta.

A escalabilidade horizontal tende a ser uma solução mais viável para gerenciar grandes volumes de dados. No entanto, é necessário o uso intenso de *threads* e processos para lidar com a criação e distribuição de tarefas. Em contraste com os SGBDs relacionais, uma das características de bancos de dados NoSQL é a ausência de bloqueios no acesso aos dados, que permite a escalabilidade horizontal e torna esta tecnologia adequada para manipular volumes de dados crescentes, tais como os gerados por sensores e dispositivos móveis na *Web of Things*.

b. Ausência de esquema ou esquema flexível

Uma das características mais importantes dos bancos NoSQL é a ausência completa ou parcial de um esquema que define rigorosamente a estrutura dos dados a serem armazenados. Essa flexibilidade contribui para um aumento na disponibilidade dos dados, assim como torna mais fácil a escalabilidade. Em contrapartida, diferente do que ocorre em sistemas relacionais, a manutenção de regras de integridade dos dados é mais fraca ou inexistente.

O nível de flexibilidade dos esquemas varia entre as diversas soluções NoSQL. Enquanto algumas tecnologias são totalmente livres de esquema (*schema-free*), outras permitem a criação de um esquema na qual novos campos podem ser adicionados. Essa flexibilidade tem um forte impacto no desenvolvimento de catálogos para a WoT. O catálogo deve permitir o registro de produtores heterogêneos de dados, logo, é necessária a utilização de um esquema que permite a extensão e atualização dos metadados sobre os produtores, bem como o suporte a registros com alguns campos não preenchidos.

c. Replicação

Os bancos de dados NoSQL oferecem suporte nativo a replicação, sendo mais uma forma de prover escalabilidade, pois a replicação nativa reduz o tempo gasto no acesso aos dados. Há duas principais formas para implementar essa replicação: i) *Master-slave* (mestre-escravo) e ii) *Multi-master*.

No *master-slave*, temos uma arquitetura baseada em um nó mestre e diversos nós escravos, onde a escrita é feita no nó mestre e, em seguida, essa escrita é refeita em cada nó escravo. Esta abordagem oferece uma leitura mais rápida, mas há um gargalo de desempenho na escrita. Deste modo, o mestre-escravo não é recomendado para manipular um grande volume de dados. O *multi-master* propõe uma arquitetura com vários nós mestres, diminuindo assim o gargalo na escrita que o mestre-escravo apresenta. Um problema dessa abordagem é a ocorrência de possíveis inconsistências, devido à existência de múltiplos nós mestres.

d. Consistência eventual

O teorema CAP (*Consistency*, *Availability* e *Partition tolerance*) diz que, em um dado momento, só é possível garantir duas entre essas três propriedades: consistência, disponibilidade e tolerância à partição. Como já vimos anteriormente, os bancos de dados NoSQL seguem os princípios BASE, tolerando, temporariamente, inconsistências em prol da disponibilidade. No contexto da *Web of Things*, os princípios mais relevantes são os de alta disponibilidade e a tolerância à partição, ou seja, as propriedades BASE, adotadas pelos bancos NoSQL, são mais apropriadas do que as ACID, por sua vez, adotadas nos SGBDs relacionais.

2.3.Modelos de Dados NoSQL

Com a crescente adoção de bancos de dados NoSQL, diversos sistemas foram desenvolvidos e, atualmente, há mais de 150 diferentes bancos NoSQL [10]. Todas essas soluções NoSQL são baseadas nos mesmos princípios, como o suporte à esquemas flexíveis e escalabilidade horizontal, porém elas possuem algumas características diferentes. Dessa forma, os bancos NoSQL podem ser agrupados em categorias, de acordo com o modelo de dados adotado. Os principais modelos de dados NoSQL são [9, 13]: i) Chave-valor; ii) Orientado a documentos; iii) Orientado a colunas e iv) Orientado a grafos. A Figura 1 ilustra exemplos de dados representados nesses modelos.

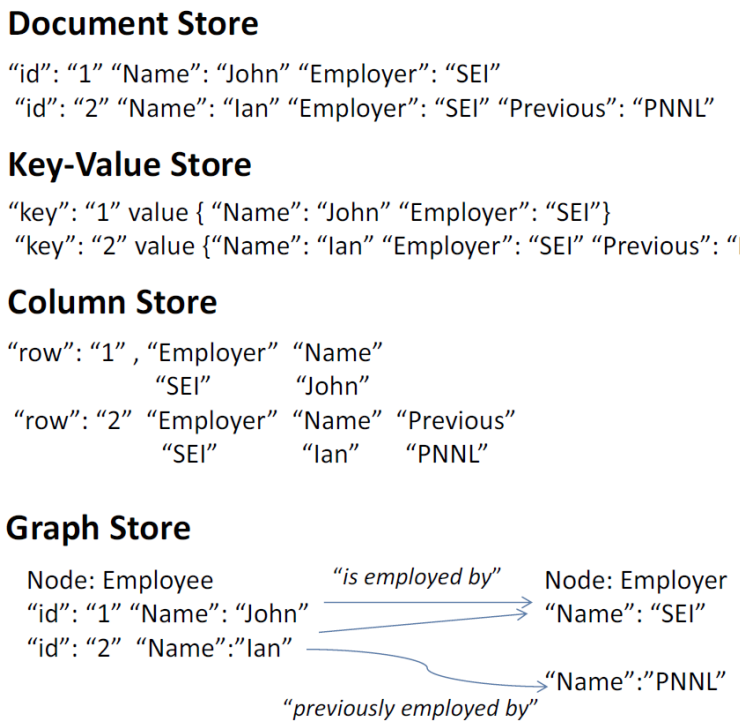


Figura 1. Exemplos de dados representados nos principais modelos NoSQL: Orientado a documentos (*Document*), Chave-valor (*Key-value*), Orientado a colunas (*Column*) e Orientado a grafos (*Graph*) [13]

Nós vamos, a seguir, descrever cada um desses modelos, apresentando as suas diferenças, assim como vantagens e desvantagens.

a. Chave-valor

Esse é o mais simples entre os modelos apresentados aqui. Banco de dados que adotam esse modelo são similares a uma estrutura de *hash map*, onde valores são endereçados por meio de uma chave única [14]. Os valores associados a cada chave são tratados pelo sistema de forma opaca, pois eles são apenas uma sequência de *bytes* (*blob*) não interpretados. Deste modo, os registros podem ser acessados somente através de buscas pela chave [13, 14].

A Figura 2 apresenta a estrutura conceitual de um banco NoSQL chave-valor. Como os valores são isolados e independentes, os relacionamentos devem ser tratados na lógica da aplicação [14]. Ao fazer uma inserção, a aplicação define a chave e o valor *blob* correspondente. O banco de dados, por sua vez, utiliza a chave para determinar onde armazenar o valor, de acordo com uma função *hash* [15]. Devido a opacidade dos valores armazenados, a maioria dos bancos de dados chave-valor suportam apenas operações de consulta, inserção e deleção [15]. Além disso, não é possível recuperar objetos através de consultas mais complexas [9].

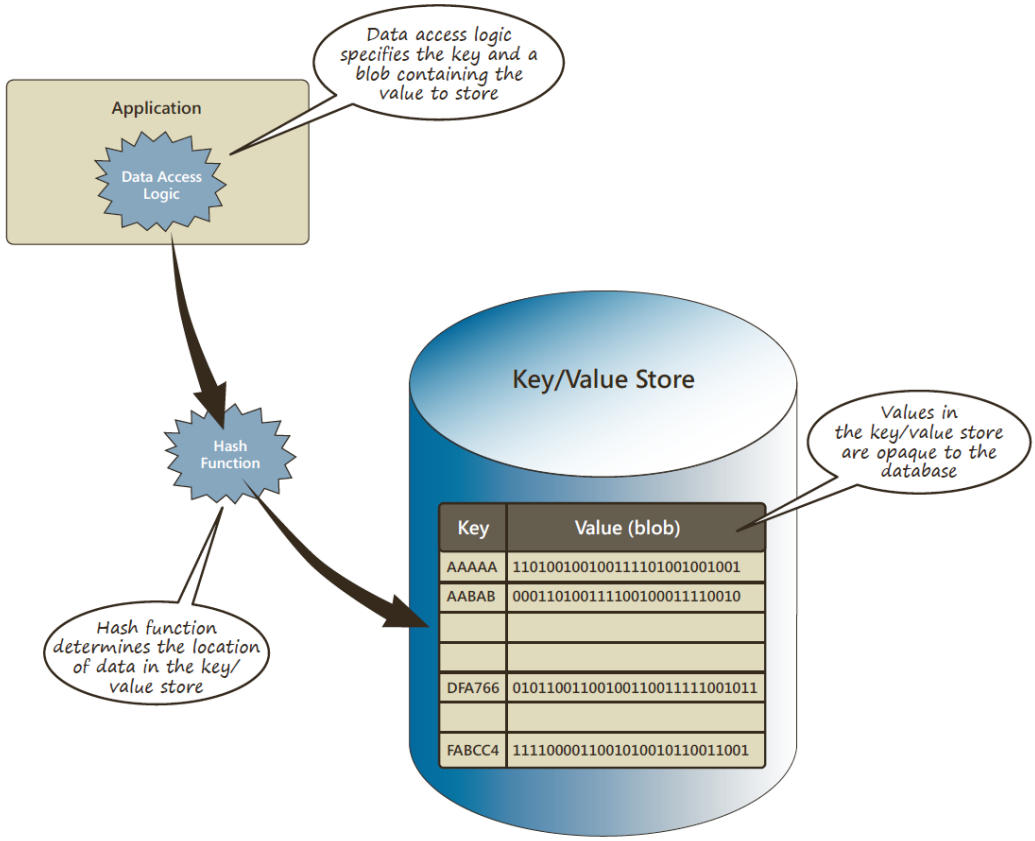


Figura 2. Estrutura conceitual de um banco de dados NoSQL chave-valor [15]

Como consequência da estrutura simples de armazenamento, os bancos de dados chave-valor são completamente livres de esquema, ou seja, novos valores de qualquer tipo podem ser adicionados [14]. A simplicidade deste modelo também permite que os dados sejam rapidamente acessados pela chave, especialmente em sistemas que oferecem uma alta escalabilidade, possibilitando, assim, o aumento da disponibilidade de acesso [9].

Como exemplos de bancos de dados NoSQL que utilizam o modelo chave-valor, destacamos o DynamoDB[4], desenvolvido pela Amazon, o Redis[5] e o Riak[6].

[4] http://aws.amazon.com/dynamodb/
[5] http://redis.io/
[6] http://basho.com/products/#riak

b. Orientado a documentos

O modelo orientado a documentos é similar ao chave-valor, com a diferença que, no primeiro, os valores são objetos complexos: documentos [12, 15]. Um banco de dados orientado a documentos armazena coleções de documentos, que, basicamente, são objetos com um identificador único (*row key*) e um conjunto de campos [9]. Cada um desses campos consiste em um par chave-valor, onde cada valor pode ser um item escalar ou composto, como listas e documentos filhos [15].

Na Figura 3, temos um exemplo de dois documentos que possuem tanto campos escalares, como *OrderData*, como campos compostos. Em *OrdemItems*, por exemplo, o valor é uma lista de documentos filhos, com os campos *ProductId*, *Quantity* e *Cost*. Uma característica importante desse modelo é que ele não depende de um esquema rígido, contrastando com os SGBDs relacionais, sendo possível adicionar novos campos [9].

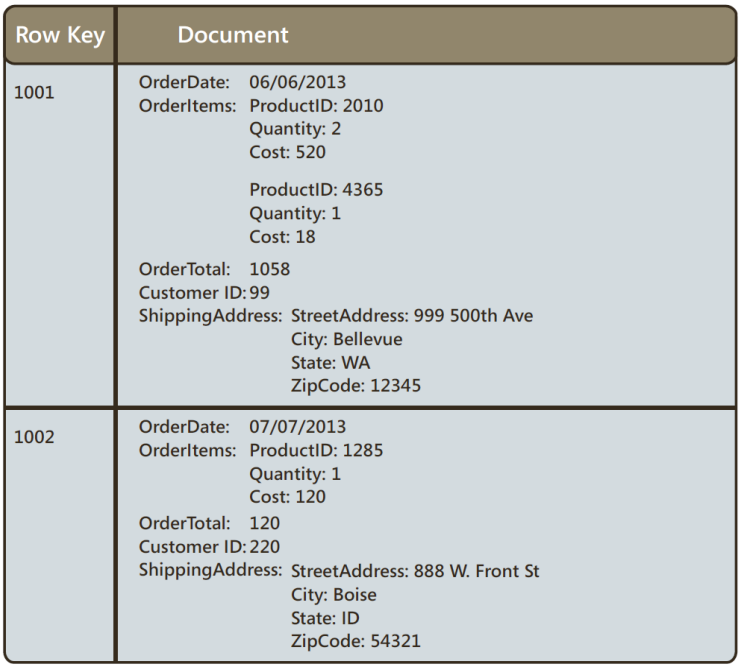


Figura 3. Exemplo de uma coleção de documentos em um banco de dados NoSQL orientado a documentos [15]

Duas das principais soluções NoSQL que adotam o modelo orientado a colunas são o MongoDB e o CouchDB[7]. O MongoDB utiliza o formato BSON para representar os documentos. Ele possui replicação nativa, implementando a estratégia *master-slave*, e permite concorrência, possibilitando a escalabilidade horizontal. Devido a essas características, nós escolhemos o MongoDB, como o representante do modelo orientado a documentos, para ser avaliado neste trabalho, na implementação de soluções de catálogos para a *Web of Things*.

[7] http://couchdb.apache.org/

c. Orientado a colunas

No modelo orientado a colunas, nós temos o conceito de famílias de coluna (*column family*), que é usado para agrupar colunas que armazenam o mesmo tipo de informação [9]. Este modelo pode ser considerado como uma extensão do chave-valor, mapeando uma chave para uma coleção de colunas, onde cada coluna é um par chave-valor [13]. Também há uma semelhança com o modelo relacional, ao menos conceitualmente, porém o poder dos bancos orientados a coluna está na abordagem desnormalizada para estruturar dados esparsos [15].

A Figura 4 apresenta um exemplo de banco de dados orientado a colunas. Nós temos duas famílias de colunas, *CustomerInfo* e *AddressInfo*, e podemos observar que os clientes de *ID* 1 e 3 possuem o mesmo endereço. Em um SGBD relacional, devidamente normalizado, não haveria a duplicação desses endereços. Ao invés disso, teríamos duas tabelas separadas. O problema da abordagem relacional é que há uma sobrecarga na realização de operações de junções, podendo se tornar significante se houver um número grande de requisições. O objetivo do modelo orientado a colunas é lidar com essas situações de forma mais eficiente [15].

Row Key	Column Families	
CustomerID	CustomerInfo	AddressInfo
1	CustomerInfo:Title Mr CustomerInfo:FirstName Mark CustomerInfo:LastName Hanson	AddressInfo:StreetAddress 999 500th Ave AddressInfo:City Bellevue AddressInfo:State WA AddressInfo:ZipCode 12345
2	CustomerInfo:Title Ms CustomerInfo:FirstName Lisa CustomerInfo:LastName Andrews	AddressInfo:StreetAddress 888 W. Front St AddressInfo:City Boise AddressInfo:State ID AddressInfo:ZipCode 54321
3	CustomerInfo:Title Mr CustomerInfo:FirstName Walter CustomerInfo:LastName Harp	AddressInfo:StreetAddress 999 500th Ave AddressInfo:City Bellevue AddressInfo:State WA AddressInfo:ZipCode 12345

Figura 4. Estrutura de dados em um banco de dados orientado a colunas [15]

A solução NoSQL que introduziu esse modelo foi o BigTable[8], desenvolvido pelo Google, que dá suporte ao particionamento de dados e oferece consistência forte, porém não há garantia de alta escalabilidade [9]. A partir do BigTable, surgiu o HBase[9], que utiliza o sistema de distribuição de arquivos do Hadoop[10]. Outro banco orientado a colunas é o Cassandra[11]. O Cassandra também permite particionamento, mas ele tem um modelo de concorrência mais fraco, sem nenhum mecanismo de bloqueio [12].

[8] https://cloud.google.com/bigtable/
[9] http://hbase.apache.org/
[10] http://hadoop.apache.org/
[11] https://cassandra.apache.org/

d. Orientado a grafos

Os bancos de dados orientados a grafos, assim como em qualquer outro modelo NoSQL, permitem o armazenamento de informações sobre entidades, porém o foco principal desse modelo são os relacionamentos que essas entidades podem ter [15]. Os bancos orientados a grafos são especializados no gerenciamento eficiente de dados fortemente conectados [14]. Este modelo possui dois componentes básicos: os nós (ou vértices), que podem ser vistos como instâncias de entidades, e as arestas, que especificam relacionamentos entre os nós. Nós e arestas podem ter propriedades que fornecem informações sobre o respectivo elemento e, adicionalmente, arestas podem ser direcionadas [9, 15].

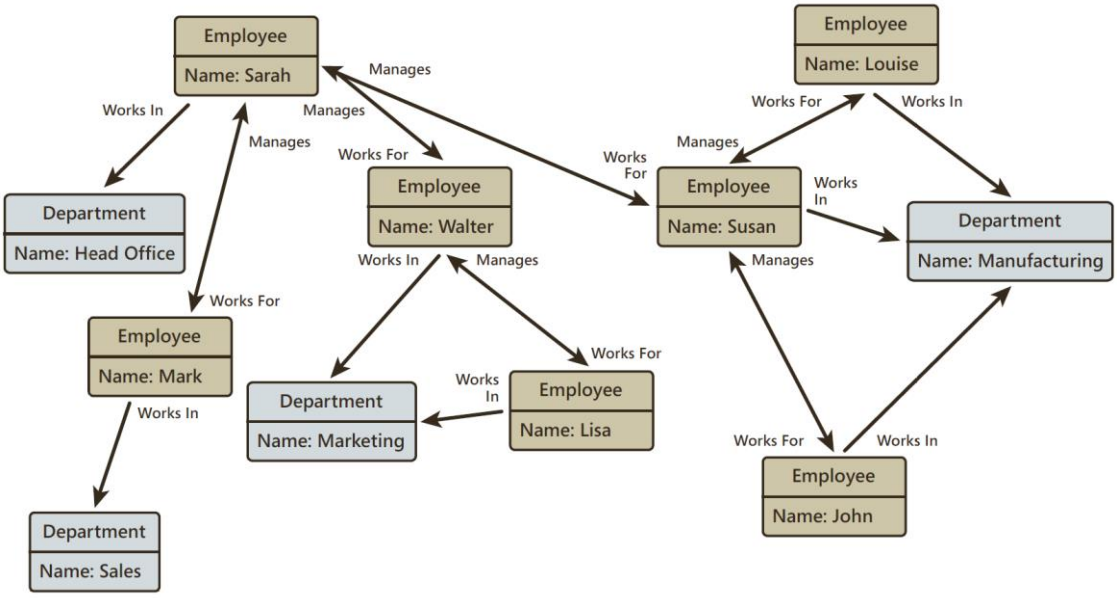


Figura 5. Exemplo de informações estruturadas como um grafo [15]

Na Figura 5, temos um exemplo de grafo que estrutura informações sobre os recursos humanos de uma organização. Temos, como entidades, funcionários (*Employee*) e departamentos (*Department*) de uma empresa. As arestas indicam hierarquias entre funcionários e os departamentos onde cada um trabalha. Nesse exemplo, todos os nós possuem a propriedade *Name*, porém, em vários bancos baseados em grafos, as arestas poderiam também ter propriedades, indicando uma informação específica sobre o relacionamento.

O principal objetivo de um banco de dados orientado a grafos é permitir uma aplicação executar, eficientemente, consultas que realizam travessias pela rede de nós e arestas. Portanto, aplicações baseadas em dados com muitos relacionamentos são mais adequadas para o uso do modelo orientado a grafos [14, 15]. Por exemplo, vamos supor que queremos descobrir quais empregados são gerenciados por um certo funcionário, de acordo com o exemplo acima. No modelo relacional, uma consulta para responder a essa pergunta poderia ser complexa, devido

ao uso de operações de junções entre tabelas, acarretando uma possível diminuição no desempenho. Por outro lado, essas consultas se tornam bastante simples em um banco orientado a grafos [9].

Como exemplos de bancos de dados baseados em grafos, temos o Neo4j[12], o OrientDB, o AllegroGraph[13] e o Virtuoso[14]. O Neo4j e o OrientDB são soluções *open-source*, implementadas em Java, que utilizam um modelo de grafo nativo. O AllegroGraph e o Virtuoso utilizam o modelo RDF para representar os grafos [9]. O OrientDB possui algumas funcionalidades que os outros bancos não oferecem, entre outros, o suporte a outros modelos de dados NoSQL. Além do modelo orientado a grafos, ele implementa os modelos chave-valor e o orientado a documentos. Uma das principais vantagens em utilizar esse SGBD é o fato que ele combina características de todos esses modelos através do conceito de multi-modelo [16].

2.4. Considerações Finais

Este capítulo introduziu importantes conceitos sobre os bancos de dados NoSQL, fornecendo o contexto histórico em que eles surgiram. Foram apresentadas as principais características presentes nas soluções NoSQL, que se adequam melhor, do que os tradicionais bancos de dados relacionais, aos requisitos exigidos para o desenvolvimento de catálogos para a *Web of Things*. Por fim, descrevemos diversos modelos de dados NoSQL, incluindo os baseados em documentos e em grafos, que serão avaliados neste trabalho por meio do uso do MongoDB e OrientDB.

[12] http://neo4j.com/
[13] http://franz.com/agraph/
[14] http://www.openlinksw.com/

3. SOLUÇÕES NOSQL PARA CATÁLOGOS DE FONTES DE DADOS

Este trabalho tem o objetivo de avaliar diferentes soluções para a implementação de catálogos de fontes da *Web of Things*. Tais soluções se diferem principalmente em dois aspectos: i) esquema de representação das fontes e ii) banco de dados NoSQL. Neste capítulo, serão apresentados os dois esquemas avaliados: SensorML e WoTDataSchema. Em seguida, nós iremos detalhar a implementação dos catálogos, para cada um desses esquemas, utilizando duas soluções NoSQL: OrientDB, orientado a grafos, e o MongoDB, orientado a documentos.

3.1. Sensor Web Enablement e SensorML

O SensorML (*Sensor Model Language*) é uma especificação, proposta pelo *Open Geospatial Consortium*, que oferece um modelo, que pode ser codificado em XML, para a descrição de metadados de sensores e sistemas de sensores [17, 18]. O principal objetivo dessa especificação é o suporte à descoberta de sensores, ao oferecer uma forma de descrever metadados dos mesmos, e o fornecimento de informações que podem ser usadas para a interpretação e análise de dados produzidos por sensores [18].

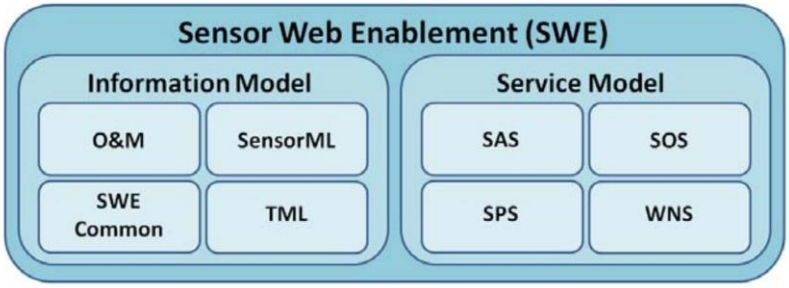


Figura 6. Visão geral da arquitetura do Sensor Web Enablement [17]

O SensorML faz parte da arquitetura SWE (*Sensor Web Enablement*), representada na Figura 6, que define padrões para modelos de dados e interfaces de serviços [4]. Além do SensorML, o SWE especifica os seguintes padrões [4, 17]:

- *Observations and Measurements* (O&M): Define um modelo para representar dados observados e medidos por sensores.
- *SWE Common*: Fornece um método para codificar informações que são usadas entre diferentes formatos do SWE.
- *Transducer Markup Language (TML)*: Suporta a codificação de dados e metadados de sensores, sendo otimizado para *streaming*.
- *Sensor Observations Service (SOS)*: Oferece operações para acessar dados medidos por sensores, sejam elas em tempo real ou não.

- *Sensor Alert Service (SAS)*: Usado para notificação de alertas de eventos relacionados a medições de sensores.
- *Sensor Planning Service (SPS)*: Define interface para configurar e manipular sensores.
- *Web Notification Service (WNS)*: Fornece notificação assíncrona entre componentes do SWE.

No SensorML, diversas entidades são especificadas e podem ser usadas para representar sensores e dispositivos da *Web of Things*. Os componentes do SensorML são modelados como processos que recebem entradas e geram saídas. Entre esses elementos, temos os seguintes: *PhysicalComponent*, *PhysicalSystem*, *ProcessChain*, *Detector* e *Sensor* [4, 20].

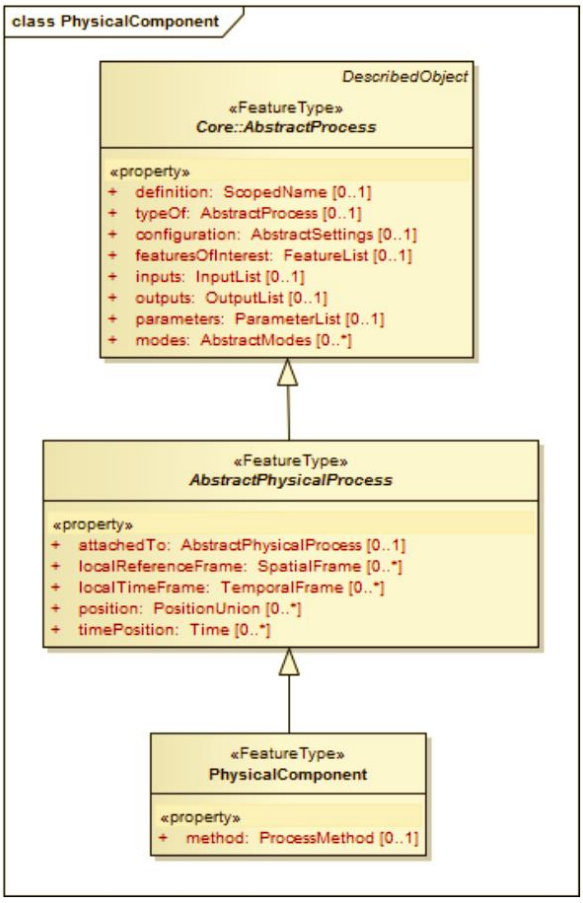


Figura 7. Modelo de dados do PhysicalComponent [20]

Um dos elementos mais abrangentes é o *PhysicalComponent*, que permite a descrição de atributos de identificação, funcionalidades, caracterização, saídas, parâmetros e serviços de sensores [4]. Na Figura 7, temos o modelo de dados que representa a estrutura dessa entidade. Neste trabalho, nós usamos somente o *PhysicalComponent* na implementação dos catálogos de fontes, pois ele oferece atributos que são suficientes para permitir a descoberta de produtores de dados [19].

3.2. WoTDataSchema

Como alternativa ao SensorML e os outros padrões do *Sensor Web Enablement*, diversas soluções foram propostas com a ideia de aplicar tecnologias semânticas para diferentes tarefas relacionadas à descoberta e integração de dados vindo de fontes heterogêneas e autônomas da *Web* [21]. Entre essas soluções, temos o WoTDataSchema, um modelo de dados, conectado a diversas ontologias externas, que está sendo desenvolvido de forma paralela a este trabalho.

O WoTDataSchema tem o objetivo de ser uma alternativa de esquema para a representação de recursos da *Web of Things*, não se restringindo a tipo de fonte (sensor, banco relacional, etc.) e nem a representação de dados (relacional, XML, RDF, etc.). A Figura 8 apresenta o modelo de dados da versão atual desse esquema.

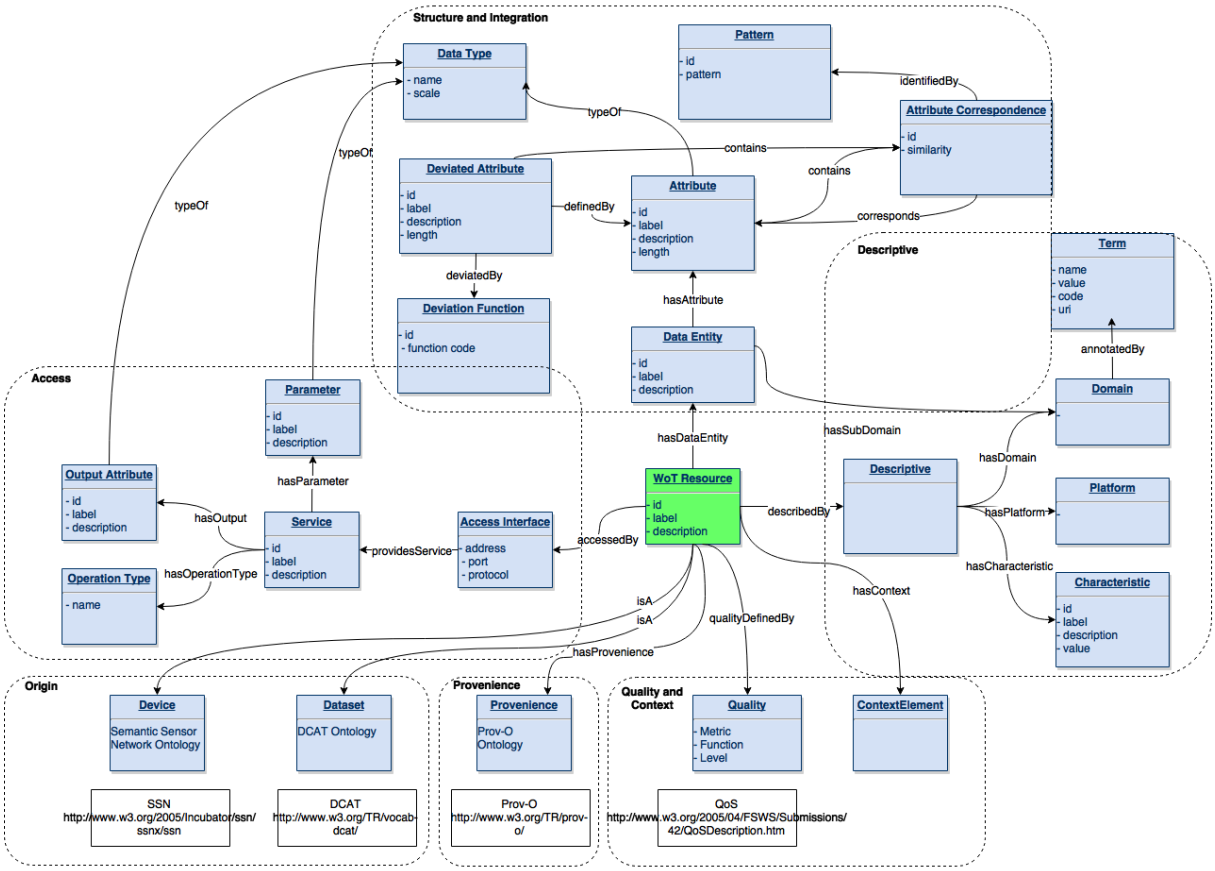


Figura 8. Modelo de dados da versão atual do WoTDataSchema

A principal classe do WoTDataSchema é o *WoT Resource*, que pode ser usado para modelar um produtor de dados da WoT. Complementarmente, temos diversas outras classes que se relacionam com o *WoT Resource*. Essas classes, e os seus respectivos relacionamentos, podem ser agrupadas em módulos que representam aspectos particulares de um produtor e os dados que ele provê. Como podemos ver na Figura 8, temos os seguintes módulos: i) Estrutura

e Integração (*Structure and Integration*); ii) Acesso (*Access*); iii) Origem (*Origin*); iv) Proveniência (*Provenience*); v) Qualidade e Contexto (*Quality and Context*) e vi) Descritivo (*Descriptive*);

O módulo de Estrutura e Integração tem o propósito de descrever a estrutura dos dados providos pelo respectivo *WoT Resource*, representando agrupamentos de atributos como *data entities*. A modelagem de esquemas e atributos foi baseado em outros trabalhos, tais como a ontologia *Schema* desenvolvida em [21, 22] e o modelo de dados do catálogo de fontes apresentado em [23]. Além de classes para representar a estrutura dos dados, esse módulo inclui elementos que permitem a integração entre esquemas de diferentes fontes da WoT.

No módulo de Acesso, temos classes que modelam interfaces de acesso para um produtor de dados, listando endereços e protocolos que permitem a comunicação com o mesmo, bem como parâmetros, saídas e operações disponíveis. O módulo de Origem oferece informações sobre o *WoT Resource* de acordo com a sua natureza: i) *Device*, como sensores e *smartphones*, ou ii) *Dataset*, como um SGBD relacional. Essas duas classes são definidas por ontologias externas.

A classe *Device* é especificada pelo *Semantic Sensor Network* (SSN), desenvolvido pelo *Semantic Sensor Networks Incubator Group* (SSNXG), um grupo criado pela W3C (*World Wide Web Consortium*) [24]. O SSN é uma ontologia que pode descrever sensores, sistemas, processos, observações e métodos usados para a coleta de dados. A Figura 9 mostra o modelo de dados do SSN.

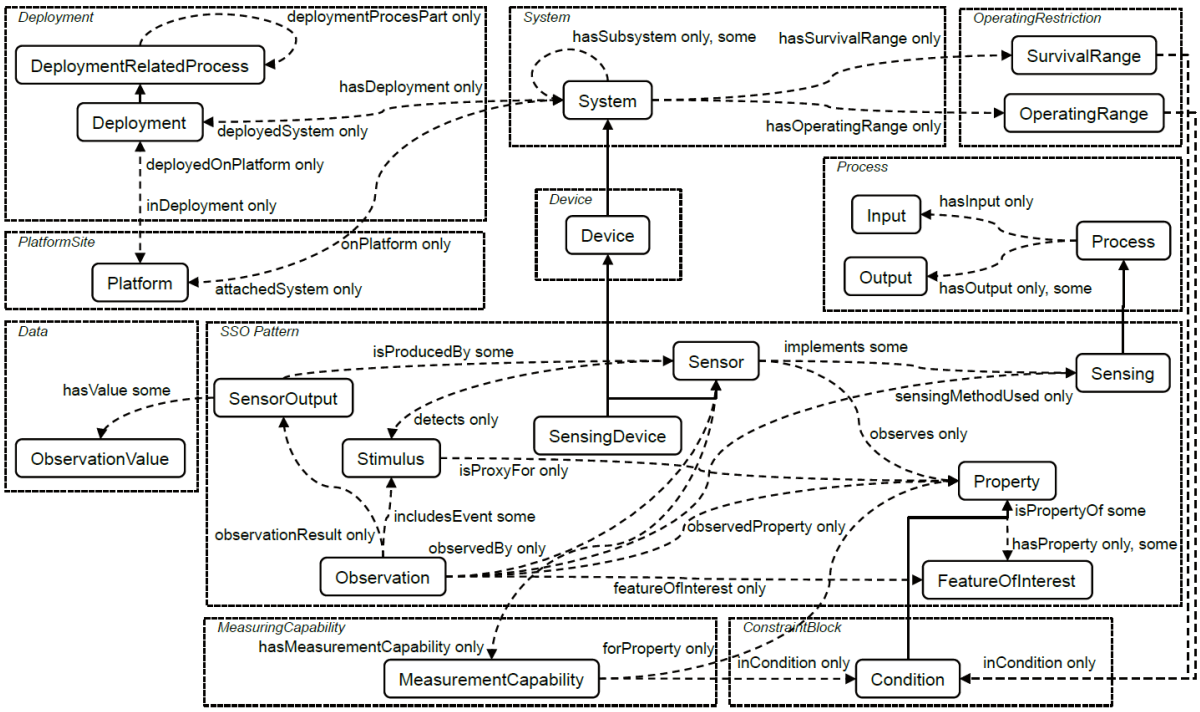


Figura 9. Modelo de dados da ontologia SSN [24]

Por outro lado, a classe *Dataset* é definida pela ontologia *Data Catalog Vocabulary* (DCAT), também desenvolvida pela W3C. O DCAT é um vocabulário RDF projetado para facilitar a interoperabilidade entre catálogos de fontes de dados publicados na *Web* [25]. A Figura 10 apresenta as classes especificadas no DCAT. Uma instância de dcat:*Dataset* é definida como uma coleção de dados publicados por um agente (*foaf:Agent*) e disponíveis para *download* em um ou mais formatos [25].

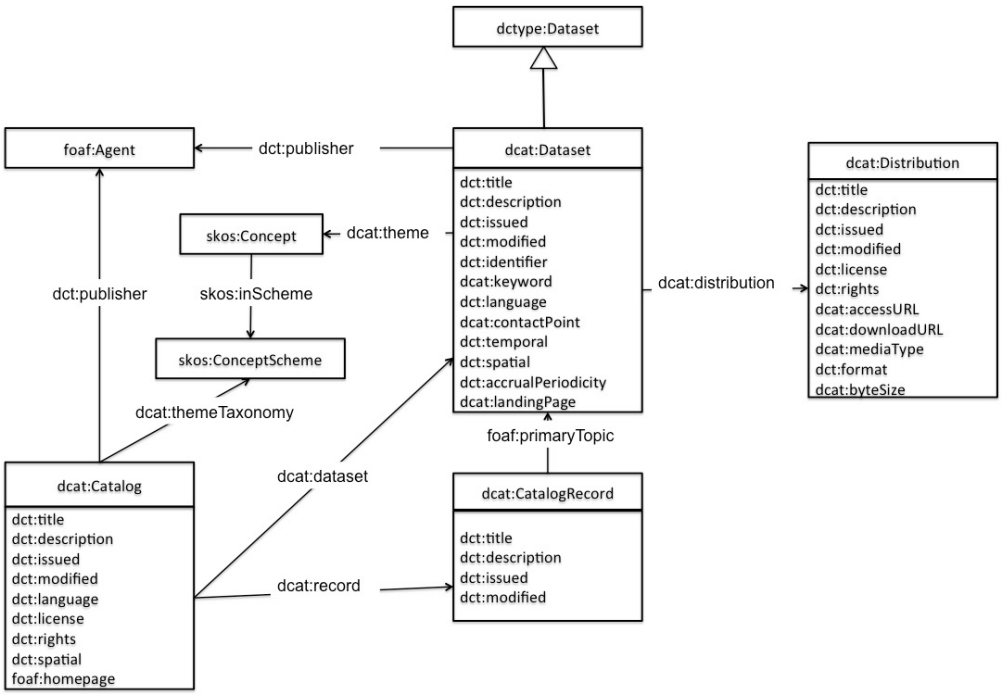


Figura 10. Modelo de dados da ontologia DCAT [25]

O módulo de Proveniência do WoTDataSchema também é representado por uma ontologia externa, o *PROV Ontology* (PROV-O). Essa ontologia fornece um conjunto de classes, propriedades e restrições que podem ser usadas para representar informações de proveniência geradas por diferentes sistemas. O PROV-O também pode ser especializado para criar novas classes e propriedades para modelar informações de proveniência para diferentes aplicações e domínios [26].

No módulo de Qualidade e Contexto, temos as classes *ContextElement*, que fornece informações contextuais sobre o produtor de dados, e *Quality*, mais uma classe especificada por uma ontologia externa. *Quality* é definida pela ontologia *Quality of Service* (QoS), que expressa a qualidade de um serviço em termos de um conjunto de dimensões (*Quality Dimension*). Associadas a uma certa dimensão, temos: i) *Function*, que descreve como a dimensão é calculada; ii) *Metric*, que expressa a unidade de medida da dimensão e, por fim, iii) *Level*, que indica o nível ao qual a dimensão pertence [27]. A Figura 11 ilustra o modelo de dados do QoS.

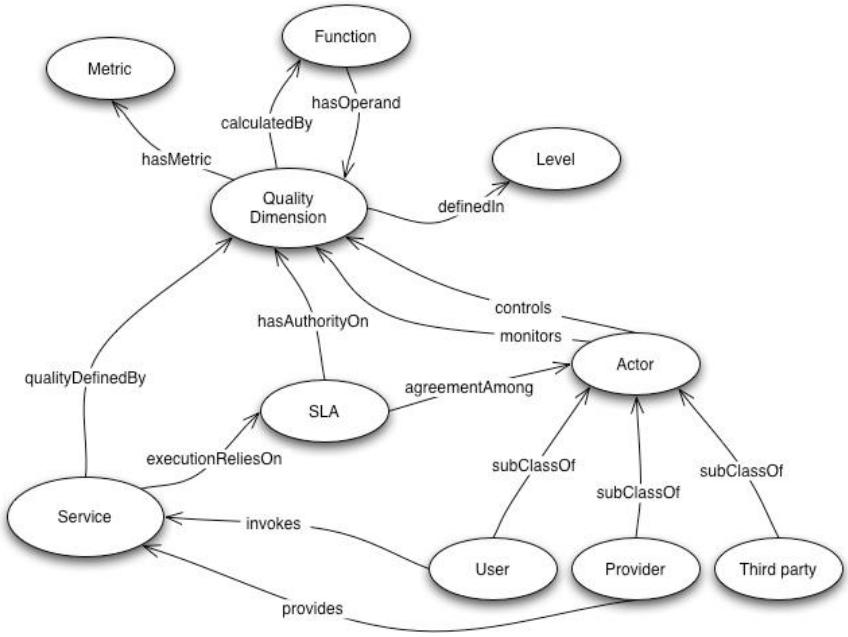


Figura 11. Modelo de dados da ontologia QoS [27]

Por fim, temos o módulo Descritivo, que disponibiliza classes para descrever características, plataformas e domínios de recursos da *Web of Things*. Por exemplo, a classe *Characteristic* pode ser usada para indicar uma categoria a qual um certo produtor de dados pertence, fornecendo o nome dessa categoria e uma descrição textual da mesma.

Como vimos, o SensorML é limitado à descrição de metadados de sensores. Já o WoTDataSchema é um esquema mais amplo, que permite a modelagem de outros tipos produtores de dados da WoT, como SGBDs, além de fornecer elementos para prover dados sobre acesso, proveniência, contexto, etc. Outra vantagem do WoTDataSchema em relação ao SensorML diz respeito à representação mais compacta de informação. O SensorML, baseado no formato XML, é bastante verboso, apresentando longos encadeamentos de elementos.

3.3. Implementação com OrientDB

O OrientDB foi uma das soluções NoSQL escolhidas para a implementação de catálogos a serem avaliados. Esse banco de dados adota o modelo orientado a grafos, como vimos no Capítulo 2, permitindo a representação de dados através de vértices e arestas. Nós consideramos o OrientDB por ele ser um dos principais bancos de dados orientados a grafos e possuir características desejadas para o contexto da *Web of Things*, tais como facilidade em prover escalabilidade horizontal, suporte nativo a replicação e flexibilidade de esquema, além de ser *open source* [16]. A versão do OrientDB usada neste trabalho foi a 2.0.8 para Windows 64 bits.

Outro aspecto interessante do OrientDB é que ele combina propriedades de diferentes modelos, principalmente dos orientados a grafo e documentos, oferecendo as funcionalidades

de um banco de dados baseado a grafos junto a características encontradas somente em bancos orientado a documentos [16]. Nesse projeto, nós trabalhamos com o OrientDB utilizando somente o modelo de grafos, pois um dos objetivos deste trabalho é a análise comparativa entre diferentes modelos NoSQL, nesse caso, os modelos de grafo, representado pelo OrientDB, e o de documentos, representado pelo MongoDB.

Foram desenvolvidos dois protótipos de catálogos para o OrientDB, um para cada esquema utilizado para modelar produtores de dados da WoT: SensorML e WoTSchemaData. Cada protótipo é composto por dois componentes: o catálogo propriamente dito e o módulo de acesso ao catálogo. O catálogo em si é responsável pelo armazenamento dos metadados de descrição das fontes, sendo, nesse caso, implementado através do OrientDB. Por sua vez, o módulo de acesso fornece operações de registro, atualização, remoção e consulta de dados armazenados no catálogo [4]. Para cada modelo de dados considerado neste trabalho, SensorML e WoTSchemaData, foi criado um banco de dados OrientDB, assim como foram desenvolvidos diferentes módulos de acesso.

O modelo de grafos do OrientDB é representado pelo conceito de *property graph*, que define vértice como uma entidade que pode ser conectada com outros vértices e arestas como uma entidade que conecta dois vértices, onde cada vértice ou aresta pode ter um conjunto de propriedades [16]. No OrientDB, nós temos também o conceito de classe, incorporando-o do paradigma de orientação a objetos, que define um tipo de registro. Cada classe pode ter um ou mais *clusters*, que determinam onde as instâncias das respectivas classes são armazenadas fisicamente. A estrutura de grafo do OrientDB é baseada em duas classes: V, para vértices (*vertices*), e E, para arestas (*edges*). Essas classes são criadas automaticamente ao construir um novo banco de dados do OrientDB usando o modelo de grafos [16].

É possível construir um grafo no OrientDB usando somente instâncias de V e E, porém, é extremamente recomendável usar tipos customizados para vértices e arestas, isto é, classes que estendem V e E. O conceito de herança de orientação a objetos também pode ser aplicado na criação de classes do OrientDB. Na definição das propriedades de cada classe, o OrientDB dá suporte a três modos de operação [16]:

- *Schema-Full*: modo mais restritivo que estabelece todas as propriedades definidas, para uma classe, como obrigatórias.
- *Schema-Less*: classes não tem propriedades definidas, ou seja, os registros podem ter campos arbitrários.
- *Schema-Hybrid*: o modo mais usado, onde temos classes criadas com alguns campos definidos, mas os registros podem ter campos customizados.

Nesse trabalho, foi adotado o modo *Schema-Hybrid*. Para cada banco de dados OrientDB criado, um por esquema, nós definimos um conjunto de classes de vértices e arestas, de acordo com o respectivo modelo de dados. Em cada classe, foram definidas propriedades, quando existentes no esquema. No entanto, como é usado o modo *Schema-Hybrid*, o catálogo permite o preenchimento de novos campos e os campos previamente criados não são obrigatórios, sendo, assim, flexível o suficiente para o contexto da WoT.

a. Implementação do catálogo a partir do SensorML

Para a implementação do catálogo baseado no SensorML, nós não utilizamos a especificação completa do *PhysicalComponent*. Entre os atributos disponíveis, foi selecionado um subconjunto, os mais relevantes para a identificação e caracterização das fontes, de acordo com o esquema apresentado na Figura 12. Nesse esquema, um subconjunto da especificação do SensorML, nós temos uma lista de características (*characteristics*) e saídas (*outputs*) para cada produtor de dados da WoT, modelado como um componente físico (*physical component*).

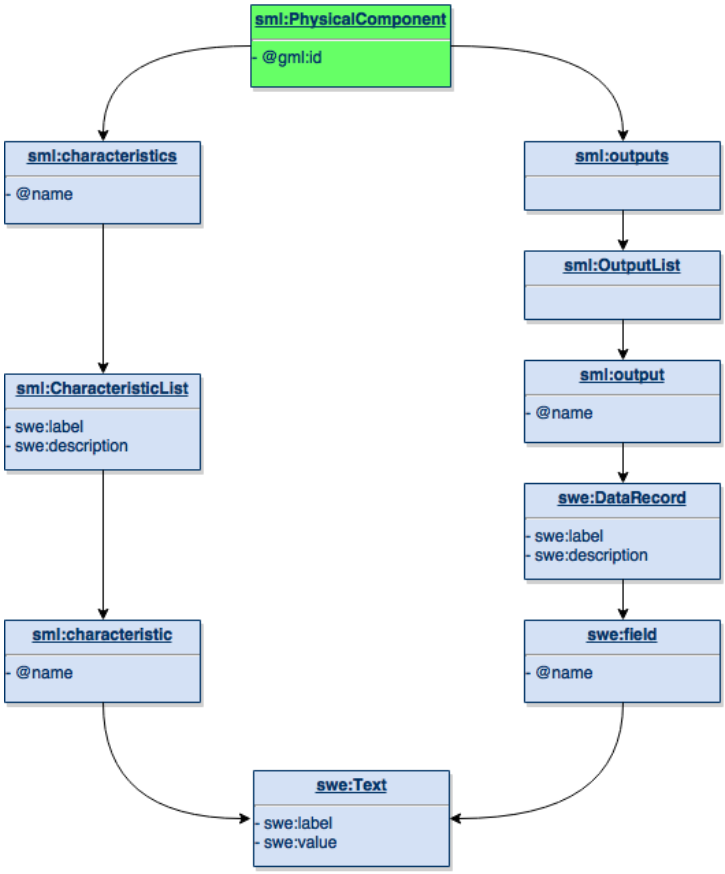


Figura 12. Subconjunto da especificação do SensorML de PhysicalComponent

Na Tabela 1, temos as classes de vértice e aresta criadas na implementação do catálogo em OrientDB para o SensorML, refletindo o esquema ilustrado na Figura 12.

Tabela 1. Classes criadas no banco de dados OrientDB para SensorML

Classe	Superclasse	Propriedades
PhysicalComponent	V	id
Characteristics	V	name
CharacteristicList	V	label, description
Characteristic	V	name
Text	V	label, value
Outputs	V	-
OutputList	V	-
Output	V	name
DataRecord	V	label, description
Field	V	name
hasCharacteristics	E	-
hasCharacteristicList	E	-
hasCharateristic	E	-
hasText	E	-
hasOutputs	E	-
hasOutputList	E	-
hasOutput	E	-
hasDataRecord	E	-
hasField	E	-

O OrientDB fornece uma interface para a administração do banco de dados, o *Studio*, onde é possível visualizar as instâncias de vértices e arestas. Na Figura 13, temos um exemplo de grafo, visualizado pelo *Studio*, armazenado nesse banco de dados. Esse grafo representa uma fonte de dados da WoT de acordo com a especificação do *PhysicalComponent* do SensorML.

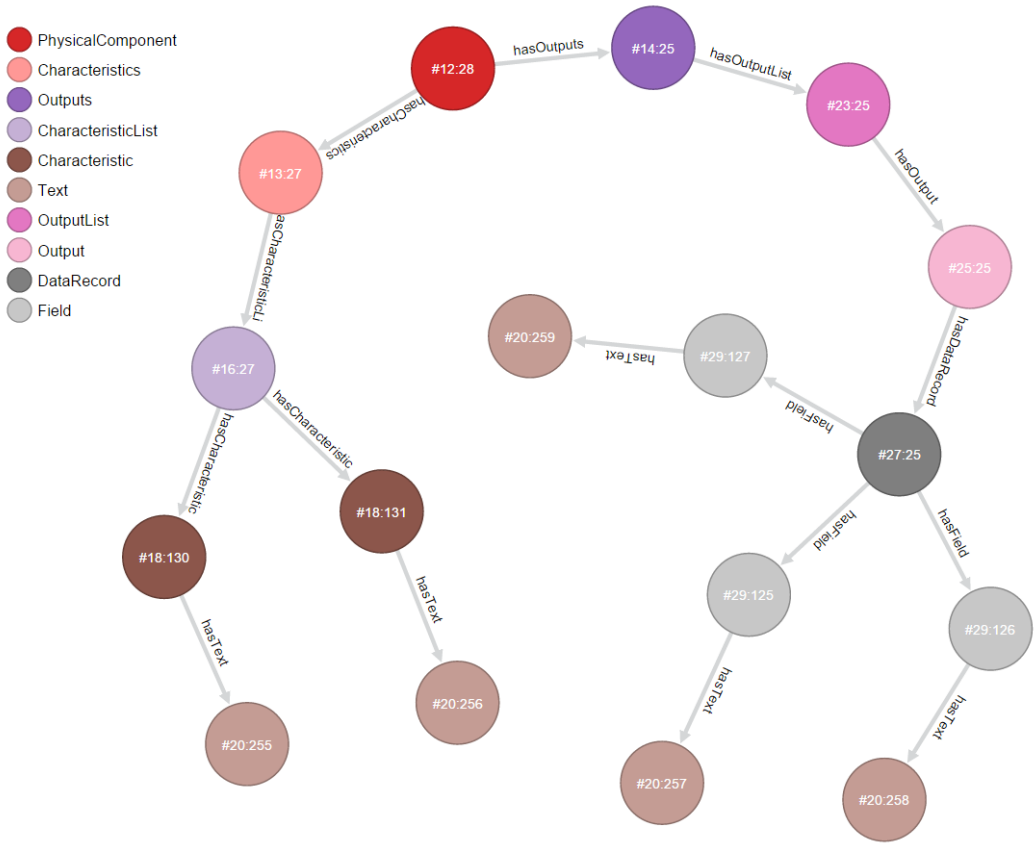


Figura 13. Exemplo de grafo criado no OrientDB baseado no SensorML

b. Implementação do catálogo a partir do WoTSchemaData

Assim como há um uso restrito da especificação do SensorML na implementação do catálogo, nós também usamos somente uma parte do WoTDataSchema nesse trabalho. Da mesma forma como selecionamos classes do SensorML para descrever características e saídas de um produtor de dados, vamos fazer uma seleção análoga do WoTDataSchema, utilizando somente essas classes: *WoT Resource*, *Data Entity*, *Attribute*, *Descriptive* e *Characteristic*.

Tabela 2. Classes criadas no banco de dados OrientDB para WoTDataSchema

Classe	Superclasse	Propriedades
WoTResource	V	id, label, description
DataEntity	V	id, label, description
Attribute	V	id, label, description, length
Descriptive	V	-
Characteristic	V	id, label, description, value
hasDataEntity	V	-
hasAttribute	V	-
describedBy	V	-
hasCharacteristic	V	-

Os atributos (*Attribute*), associados a um esquema (*Data Entity*), do WoTDataSchema correspondem às saídas (*Outputs*) do SensorML, assim como o conteúdo de características (*Characteristic*). A diferença é que o SensorML requer mais classes para fornecer, basicamente, a mesma informação. O catálogo em OrientDB para o WoTSchemaData foi criado de acordo com a Tabela 2, onde podemos ver as classes de vértice e aresta baseadas na Figura 8. Na Figura 14, podemos ver um exemplo de grafo cadastrado nesse catálogo, visualizado pelo *Studio*, no qual temos uma instância de *WoTResource* com treze atributos e cinco características.

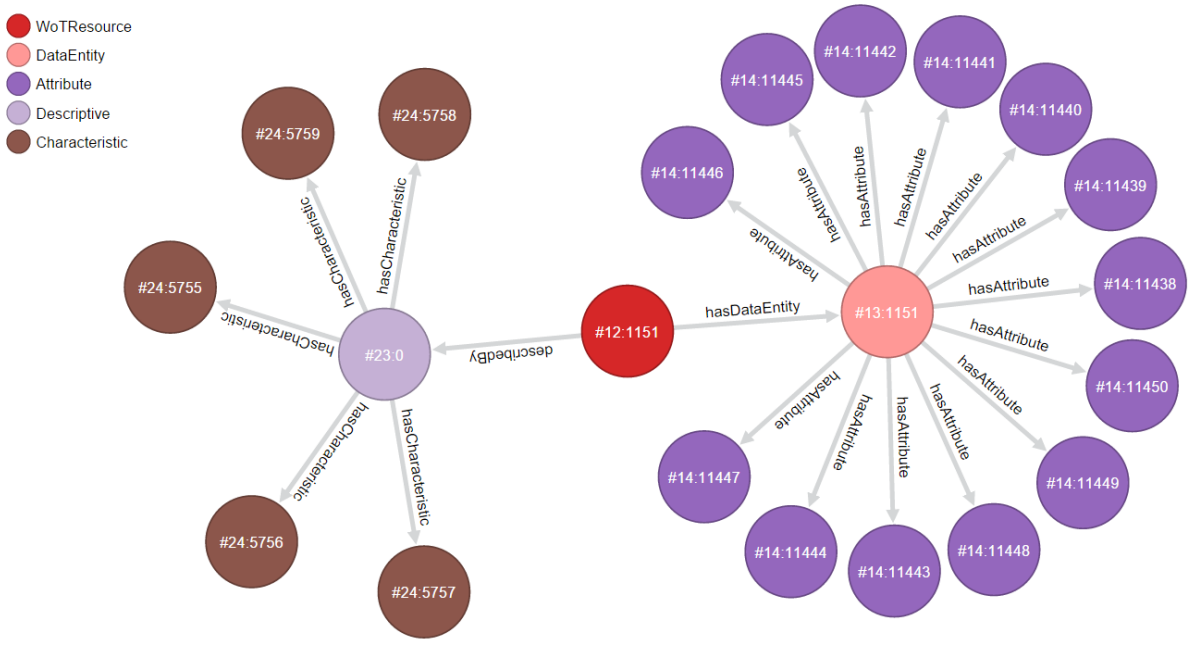


Figura 14. Exemplo de grafo criado no OrientDB baseado no WoTDataSchema

c. Implementação dos módulos de acesso

Os módulos de acesso, para cada catálogo implementado, foram desenvolvidos através da linguagem de programação Java[15]. Java foi escolhido por diversos fatores, tais como: i) alta portabilidade do código fonte; ii) suporte à programação concorrente e iii) bibliotecas disponíveis para acesso aos bancos de dados OrientDB e MongoDB. Por meio do módulo de acesso, uma aplicação pode efetuar ações sobre o respectivo catálogo, como o registro de uma nova fonte de dados ou a busca das fontes que possuem uma certa característica.

Na implementação do módulo de acesso ao catálogo baseado no SensorML, foi utilizada a biblioteca *Common Library*[16], do SWE, na codificação, decodificação e validação de registros de fontes da WoT. As operações de inserção e atualização desse módulo recebem, como parâmetro, um arquivo JSON, descrevendo um produtor de dado em SensorML. Havia outras alternativas de representação, mas decidimos adotar o JSON porque o módulo de acesso do SensorML para o MongoDB, o qual descreveremos mais tarde, necessita desse formato, pois o MongoDB armazena documentos em uma versão binária do JSON [28]. Dessa forma, o módulo de acesso do SensorML para o OrientDB utiliza o JSON por questões de compatibilidade e reuso de componentes desenvolvidos. No entanto, como o SensorML é especificado em XML, foi necessária a implementação de *parsers* para a tradução entre os dois formatos. Essa conversão é realizada com base no *Badgerfish*[17], uma convenção que propõe regras que lidam com a tradução de atributos, elementos vazios e a ordem do conteúdo [4].

Já na implementação do módulo de acesso ao catálogo baseado no WoTDataSchema, foram criadas classes POJO (*Plain Old Java Object*) que correspondem às classes definidas no próprio esquema, ilustrado na Figura 8, sendo *WoTResource* a classe principal. O módulo de acesso do WoTDataSchema manipula instâncias de *WoTResource* nas operações sobre o banco de dados.

O acesso ao banco de dados do OrientDB, para ambos os módulos, é feito, principalmente, por meio do *Blueprints*[18]. O *Blueprints* é uma API que suporta operações básicas em um banco de dados orientado a grafos [16]. Uma das vantagens de utilizá-lo é a sua compatibilidade com outros bancos baseados em grafos, tornando a implementação do módulo mais portátil. A classe *OCommandSQL*, da própria API do OrientDB, também é utilizada na execução de consultas e na criação de índices.

[15] https://www.java.com
[16] https://github.com/sensiasoft/lib-sensorml
[17] http://badgerfish.ning.com
[18] https://github.com/tinkerpop/blueprints/

Além das operações de inserção, atualização, remoção e consulta, cada módulo é responsável pela criação de índices e pela implementação de um sistema de *cache* de consultas. O uso de índices e *cache* são parâmetros usados nos experimentos. Em ambos os módulos, há um método que executa a criação de um índice, no banco de dados do OrientDB, sobre *Characteristic*, que é utilizado nas consultas realizadas nos experimentos. No OrientDB, esse índice é baseado em uma árvore B, mas com diversas otimizações relacionadas à inserção de dados [16].

O sistema de *cache* de consultas foi implementado através do *framework* Apache JCS[19]. O JCS é um sistema de *cache* distribuído, escrito em Java, que tem o objetivo de acelerar aplicações. Esse sistema é utilizado para armazenar os resultados de operações de consulta que o módulo oferece, adotando a política LRU (*Least Recently Used*) de *cache*.

3.4. Implementação com MongoDB

Entre as diversas soluções NoSQL que adotam o modelo orientado a documentos, o MongoDB foi escolhido por ser um dos bancos de dados mais usados [4] e, assim como o OrientDB, oferecer características que são propícias para o desenvolvimento de catálogos para a *Web of Things*. Entre essas características, destacam-se [4, 28]:

- Alta performance: o suporte a documentos embutidos reduz a atividade de entrada e saída no sistema de armazenamento.
- Alta disponibilidade: MongoDB oferece replicação automática através de *replica sets*, grupos de servidores que mantém o mesmo conjunto de dados, provendo redundância.
- Escalamento automático: MongoDB fornece escalabilidade horizontal através do uso de *sharding*, um sistema de particionamento automático.

Um registro em MongoDB é um documento que é, basicamente, composto por um conjunto de pares chave/valor. Esses documentos são armazenados na forma BSON, uma codificação binária do JSON. Na Figura 15, temos um exemplo de documento do MongoDB. Os valores podem incluir outros documentos, *arrays* e *arrays* de documentos [28].

Figura 15. Exemplo de documento do MongoDB [28]

[19] https://commons.apache.org/proper/commons-jcs/

No MongoDB, temos o conceito de *collection*, um grupo de documentos que possuem uma estrutura similar. Assim como foi feito para o OrientDB, foram implementados dois catálogos para o MongoDB, um para o SensorML e outro para o WoTSchemaData, e os seus respectivos módulos de acesso. Para cada catálogo, temos uma *collection* que armazena os documentos que descrevem produtores de dados conforme o modelo de dados correspondente. As *collections* no MongoDB possuem um esquema flexível, no qual é possível ter documentos dentro de uma mesmo *collection* com campos diferentes.

A versão do MongoDB usada no desenvolvimento dos catálogos, tanto para o SensorML quanto para o WoTSchemaData, foi a 3.0.4 para Windows 64 bits.

a. Implementação do catálogo a partir do SensorML

O catálogo no MongoDB para o SensorML é uma *collection* na qual os documentos JSON armazenados seguem a estrutura definida pelo subconjunto da especificação de *PhysicalComponent* ilustrada na Figura 12. A Figura 16 apresenta um exemplo de documento que descreve um produtor de dados de acordo com essa estrutura. As listas de características (*sml:chacteristic*) e de saídas (*sml:output*) são armazenadas de forma embutida, como *arrays*.


```
{ ⊟
   "sml:PhysicalComponent":{ ⊟
      "@xmlns":{ ⊞ },
      "@gml:id":"1",
      "sml:characteristics":{ ⊟
         "@name":"AlertSystems Attributes ",
         "sml:CharacteristicList":{ ⊟
            "swe:label":{ ⊞ },
            "swe:description":{ ⊞ },
            "sml:characteristic":[ ⊟
               { ⊟
                  "@name":"CAT1",
                  "swe:Text":{ ⊞ }
               }
            ]
         }
      },
      "sml:outputs":{ ⊟
         "sml:OutputList":{ ⊟
            "sml:output":{ ⊟
               "@name":"Description",
               "swe:DataRecord":{ ⊟
                  "swe:label":{ ⊞ },
                  "swe:description":{ ⊞ },
                  "swe:field":[ ⊟
                     { ⊟
                        "@name":"ATT1",
                        "swe:Text":{ ⊞ }
                     }
                  ]
               }
            }
         }
      }
   }
}
```


Figura 16. Exemplo de documento JSON, baseado no SensorML, a ser armazenado no catálogo em MongoDB

b. Implementação do catálogo a partir do WoTDataSchema

De forma semelhante ao catálogo para o SensorML, o catálogo para o WoTDataSchema também é uma *collection* de documentos JSON, porém, esses documentos são baseados na classe *WoT Resource*, do modelo de dados definido pelo WoTDataSchema. Nesse catálogo, assim como no correspondente desenvolvido para o OrientDB, também não há a utilização da especificação completa desse esquema.


```
{
  "_id":{
    "$oid":"559c17106295669aa2240505"
  },
  "id":1,
  "label":"wotResource1",
  "description":"wotResource1 description",
  "dataEntities":[
    {
      "id":1,
      "label":"schema1",
      "description":"schema1 description",
      "attributes":[
        {
          "id":1,
          "label":"attribute1",
          "description":"attribute1 description",
          "length":10
        },
        {
          "id":2,
          "label":"attribute2",
          "description":"attribute2 description",
          "length":10
        }
      ]
    }
  ],
  "descriptive":{
    "characteristics":[
      {
        "id":1,
        "label":"CAT0",
        "description":"characteristic1 description",
        "value":"VALUE8"
      }
    ]
  }
}
```


Figura 17. Exemplo de documento JSON, baseado no WoTDataSchema, a ser armazenado no catálogo em MongoDB

A Figura 17 apresenta um exemplo de documento JSON que representa um produtor de dados da WoT. O conteúdo desse documento é gerado a partir da serialização da classe Java *WoTResource*, desenvolvida nos módulos de acesso para o WoTDataSchema, de tal forma que os seus atributos são embutidos no mesmo documento. Nesse exemplo, temos um produtor de dados com dois atributos, agrupados em um mesmo esquema, e somente uma característica.

Como podemos observar, para ambos os modelos de dados, o MongoDB permite a representação de uma fonte de dados da *Web of Things* através de um único documento JSON, enquanto nos catálogos para o OrientDB, cada fonte é armazenada sob a forma de um grafo.

c. Implementação dos módulos de acesso

Os módulos de acesso para o MongoDB também foram desenvolvidos em Java, utilizando a versão 2.13.0 do driver JDBC para esse banco de dados. De forma análoga ao que já vimos para o OrientDB, foram desenvolvidos dois módulos de acesso, um para o SensorML e outro para o WoTDataSchema. Nos dois módulos, as operações de inserção, remoção, atualização e consulta de registros trabalham com o formato JSON.

No módulo para o SensorML, a biblioteca SWE *Common Library* foi novamente utilizada para lidar com a descrição dos registros em SensorML em XML e a essa descrição, por sua vez, é convertida para JSON de acordo com o *Badgerfish*. Por outro lado, no módulo para o WoTDataSchema, as classes POJO – criadas para representar, em Java, a estrutura desse esquema – são serializadas para JSON.

O processo de inserção e recuperação de dados no MongoDB requer o preenchimento de um objeto apropriado para o armazenamento pelo SGDB. Esse objeto é uma instância da interface *DBObject*, que é, basicamente, um *hash map* que possui funções de serialização e deserialização para JSON. Desse modo, nas operações de inserção, uma descrição em JSON é serializada para *DBObject* para, enfim, ser adicionada a uma *collection*, enquanto nas operações de consulta, uma instância *DBObject*, recuperada de uma *collection*, é convertida para JSON.

Para ambos os módulos de acesso, também foram desenvolvidos métodos para a criação de índices e sistema de *cache* para o resultado de consultas. Os índices são criados para a respectiva *collection* sobre o campo *swe:Text*, que contém o valor de uma certa característica. Nos experimentos, são executadas consultas sobre esse campo. Já o sistema de cache é implementado por meio do Apache JCS, assim como nos módulos dos catálogos do OrientDB.

3.5. Considerações Finais

Neste capítulo, foram apresentados dois modelos de dados para a representação de recursos da *Web of Things*: o SensorML e o WoTDataSchema, explicando as suas características e diferenças. Em seguida, este capítulo descreveu a implementação de diferentes soluções de catálogos para a WoT a partir desses dois esquemas e dois bancos de dados NoSQL: o OrientDB e o MongoDB. Foram detalhados tanto o desenvolvimento dos catálogos em si, que provê o armazenamento de metadados, quanto os módulos de acesso a esses catálogos. O próximo capítulo irá descrever os experimentos realizados com essas soluções de catálogo e discutir os seus resultados.

4. EXPERIMENTOS E ANÁLISE DOS RESULTADOS

Nesse capítulo, serão apresentados os experimentos realizados nesse trabalho com o objetivo de avaliar o desempenho de diferentes soluções de catálogos para fontes de dados da *Web of Things*. Primeiramente, será descrito o ambiente desenvolvido que permite a execução desses experimentos, simulando interações entre produtores e consumidores de dados com os catálogos. A seguir, os experimentos, e os respectivos cenários simulados, serão detalhados e, por fim, os resultados obtidos serão analisados.

4.1. Ambiente de Simulação

A execução de experimentos envolvendo soluções para a WoT usando sistemas reais pode ser inviável devido aos altos custos de componentes físicos e a complexidade ao controlar variáveis [4]. Como alternativa ao uso de componentes reais, esse trabalho propõe o uso de simulações, que permitem a modelagem de um sistema próximo do real, aumentando, assim, a confiabilidade nos resultados e na tomada de decisões [29].

O ambiente de simulação desenvolvido nesse trabalho, com base nos simuladores propostos em [4], faz uso de *mocks* para produtores de dados (PDs) e consumidores de dados (CDs). Os *mocks* são objetos pré-programados que simulam o comportamento de um sistema real de uma forma controlada [4]. O modelo de simulação criado permite os *mocks* de PDs e CDs submeter requisições ao protótipo de catálogo. Os *mocks* PDs enviam requisições de registro, atualização e remoção de fontes de dados, enquanto os *mocks* CDs enviam requisições de consulta.

No lado do servidor do ambiente, recebendo as requisições dos *mocks*, o simulador faz uso de um conjunto de *threads* para lidar com tais requisições, de tal forma que cada *thread* tem acesso a uma instância do módulo de acesso a um catálogo. O objetivo principal dos experimentos executados é o de verificar como o catálogo se comporta diante de um envio intermitente de requisições, sem considerar o tempo de transmissão das mesmas. Dessa forma, nós abstraímos aspectos relacionados à infraestrutura de comunicação, como o congestionamento e a topologia da rede, permitindo, assim, a execução da simulação em uma única máquina.

Em todos os experimentos, o ambiente de simulação é responsável pela criação de um *testbed*. Esse *testbed* é constituído por um conjunto de instâncias que representam produtores de dados. Com os experimentos envolvendo o WoTDataSchema, essas instâncias são da classe *WoTResource*. Já nos experimentos com o SensorML, as instâncias são de uma interface

chamada *DataProducer*, que estende a interface *PhysicalComponent*, disponibilizada pela biblioteca SWE *Common Library*.

O resultado dos experimentos é medido através de dois valores: a latência total de execução e a vazão de atendimento de requisições. A primeira é a duração total do experimento, sendo medida somente após a criação do *testbed*, até o momento do atendimento de todas as requisições. As requisições simuladas são atendidas em uma ordem aleatória, tornando difícil o cálculo do tempo exato de execução de cada requisição. Portanto, o modelo de simulação faz um cálculo aproximado da latência considerando o tempo de relógio, ao invés do tempo de execução para cada requisição [4, 29]. Já a medida de vazão de requisições é a taxa de atendimento de requisições, sendo calculada pela divisão entre a quantidade de requisições atendidas e a latência total.

4.2. Experimentos e Cenários Simulados

Nesse trabalho, foram realizados dois tipos de experimentos: i) experimentos de vazão e ii) experimentos de resiliência, ambos com o objetivo principal de avaliar o desempenho das soluções de catálogos desenvolvidas. Para cada um desses tipos, foram executados quatro experimentos utilizando as mesmas configurações de cenários simulados, variando somente a solução de catálogo avaliada. As quatro soluções de catálogo analisadas nesses experimentos são a combinação dos modelos de dados e bancos de dados NoSQL apresentados no capítulo anterior: i) SensorML e OrientDB; ii) WoTDataSchema e OrientDB; iii) SensorML e MongoDB e iv) WoTDataSchema e MongoDB. A Tabela 3 apresenta os experimentos realizados e as respectivas soluções avaliadas. Os experimentos de vazão são os A, B, C e D, enquanto os de resiliência são os E, F, G e H.

Tabela 3. Experimentos realizados

Id. Experimentos	Banco de Dados NoSQL	Modelo de Dados
Experimentos A/E	OrientDB	SensorML
Experimentos B/F	OrientDB	WoTDataSchema
Experimentos C/G	MongoDB	SensorML
Experimentos D/H	MongoDB	WoTDataSchema

Os experimentos de vazão têm como principal foco a avaliação do desempenho do catálogo através das medidas de latência e vazão de atendimento de requisições de inserção, atualização, remoção e busca. Por sua vez, os experimentos de resiliência fazem uso de uma carga maior de requisições, permitindo, assim, a avaliação não somente do desempenho, mas também da robustez do catálogo. Essa robustez diz respeito à capacidade do catálogo de manipular frequentes requisições.

Nos experimentos de vazão, foram realizadas simulações nas quais todos os produtores de dados pertencentes ao *testbed* tiveram os seus registros inseridos, atualizados e removidos do catálogo, executando cada uma dessas operações uma única vez por PD. Além de tais operações, esses experimentos incluíram requisições de consulta por PDs cadastrados nesse *testbed* por característica, através de um par *label*/*value*. O *testbed* para cada uma das quatro operações foi constituído pelo mesmo conjunto de 10000 PDs. Desse modo, para cada cenário, foram executadas 10000 requisições de inserção, atualização e remoção. Por outro lado, para as operações de busca, foram enviadas 1000 requisições de consulta em cada cenário simulado.

Para todos os experimentos realizados nesse trabalho, tanto de vazão quanto de resiliência, a geração das instâncias para o *testbed* é baseada em dados sintéticos, representando PDs genéricos, com uma lista de atributos (ou *outputs* no SensorML) e características. No entanto, o desempenho dos catálogos ao lidar com esses dados seria semelhante caso usássemos dados reais, representando diferentes tipos de PDs (como semáforos, bases de dados abertas e táxis), pois a quantidade de atributos e características para cada PD, bem como o conteúdo dos mesmos, foram gerados de forma a simular informações reais. As consultas de buscas são geradas, de forma aleatória, a partir de cinco valores de *label* e dez valores de *value* para características, ou seja, um total de 50 consultas distintas possíveis.

Tabela 4. Cenários simulados nos experimentos de vazão

Id. Cenários	Política de SGBD
Cenários A1/B1/C1/D1	Sem índice
Cenários A2/B2/C2/D2	Com índice

Em cada um dos quatro experimentos de vazão, foram executados dois cenários, um sem o uso de índice e outro com o uso de índice sobre característica. Na Tabela 4, temos os cenários desses experimentos. O índice tem o objetivo de acelerar a busca, porém ele pode gerar uma sobrecarga nas inserções e atualizações. Dessa forma, é interessante avaliar o impacto do uso de índices não somente na busca, mas também nas outras operações. No OrientDB, esse índice é criado de forma composta sobre o par *label*/*value* da classe *Text*, no SensorML, e *Characteristic*, no WoTDataSchema. No MongoDB, o índice é criado somente no campo *value*, pois o uso do índice composto apresenta uma pior performance.

Para todos os cenários simulados nos experimentos de vazão, foi também avaliado o impacto da quantidade de *threads* usadas para o atendimento das requisições. Cada cenário foi executado para um dado número de *threads*. Nós utilizamos dez valores para essa quantidade, variando entre 4 a 40, de quatro em quatro. Em contrapartida, nos experimentos de resiliência, foram utilizadas, de forma fixa, 50 *threads*.

Nos experimentos de resiliência, a simulação de cada cenário foi executada em ciclos, nos quais todos os PDs e CDs participantes enviam uma requisição ao catálogo. A quantidade total de requisições enviadas é determinada pela quantidade de ciclos, PDs e CDs. Nesse experimento, os PDs submetem somente requisições de inserção, enquanto os CDs, da mesma forma que nos experimentos de vazão, submetem requisições de consulta de PDs por característica. No *testbed* desses experimentos, são registrados 1000 PDs.

Os cenários simulados nos experimentos de resiliência são apresentados na Tabela 5. Assim como nos experimentos de vazão, foram avaliadas diferentes configurações de banco de dados, porém, além do uso de índices, foi também investigado o impacto do uso de *cache* de resultados de consultas. Outras variáveis utilizadas foram a quantidade de CDs, de ciclos e o percentual de PDs previamente cadastrados no *testbed*. Ao longo da simulação, os PDs não cadastrados solicitam o registro dos seus metadados. Dessa forma, quanto maior esse percentual, menor é a carga de inserção. Em todos os cenários executados nos experimentos de resiliência, foi utilizado um limite de 100 na quantidade de registros retornados por uma consulta. Tal limitação é uma prática comum em sistemas *Web* [30].

Tabela 5. Cenários simulados nos experimentos de resiliência

Id. Cenários	Quantidade de CDs	Quantidade de Ciclos	Percentual de PDs Previamente Cadastrados	Política de SGBD
Cenários E1/F1/G1/H1	10000	50	50%	Sem índice
Cenários E2/F2/G2/H2	10000	50	50%	Com índice
Cenários E3/F3/G3/H3	10000	50	75%	Com índice
Cenários E4/F4/G4/H4	10000	50	100%	Com índice
Cenários E5/F5/G5/H5	10000	100	50%	Com índice
Cenários E6/F6/G6/H6	10000	100	75%	Com índice
Cenários E7/F7/G7/H7	10000	100	100%	Com índice
Cenários E8/F8/G8/H8	10000	50	50%	Com índice e *cache*
Cenários E9/F9/G9/H9	10000	50	75%	Com índice e *cache*
Cenários E10/F10/G10/H10	10000	50	100%	Com índice e *cache*
Cenários E11/F11/G11/H11	10000	100	50%	Com índice e *cache*
Cenários E12/F12/G12/H12	10000	100	75%	Com índice e *cache*
Cenários E13/F13/G13/H13	10000	100	100%	Com índice e *cache*
Cenários E14/F14/G14/H14	25000	50	50%	Com índice e *cache*
Cenários E15/F15/G15/H15	25000	50	75%	Com índice e *cache*
Cenários E16/F16/G16/H16	25000	50	100%	Com índice e *cache*
Cenários E17/F17/G17/H17	25000	100	50%	Com índice e *cache*
Cenários E18/F18/G18/H18	25000	100	75%	Com índice e *cache*
Cenários E19/F19/G19/H19	25000	100	100%	Com índice e *cache*

Todos os experimentos foram realizados em um computador portátil HP ProBook 6360b. Esse computador tem o processador Intel Core i5-2520M 2.5 GHz, memória RAM de 4 GB e sistema operacional Windows 7 Professional. Para cada experimento, cada cenário foi executado por, no mínimo, três rodadas. A latência e a vazão para cada cenário foram calculadas a partir do tempo médio de execução das rodadas.

4.3. Resultados

A seguir, os resultados dos experimentos de vazão e de carga serão apresentados e analisados. Essa análise se dá tanto pela avaliação dos resultados de experimentos individualmente, bem como na comparação entre os resultados obtidos a partir de diferentes experimentos que envolvem cenários de mesma configuração, onde temos diferentes soluções de catálogo utilizadas, permitindo, portanto, a avaliação comparativa entre essas soluções.

a. Experimentos de vazão

Foram realizados quatro experimentos de vazão: i) Experimento A; ii) Experimento B; iii) Experimento C e iv) Experimento D. Cada experimento avaliando, com os mesmos cenários, uma das quatro soluções de catálogo desenvolvidas nesse trabalho. O Experimento A avalia a latência e vazão de operações sobre o protótipo de catálogo implementado a partir do SensorML e OrientDB. A Figura 18 mostra os resultados relacionados às operações de inserção, atualização e remoção obtidos no Experimento A. Na Figura 19, temos os resultados referentes à operação de busca do mesmo experimento.

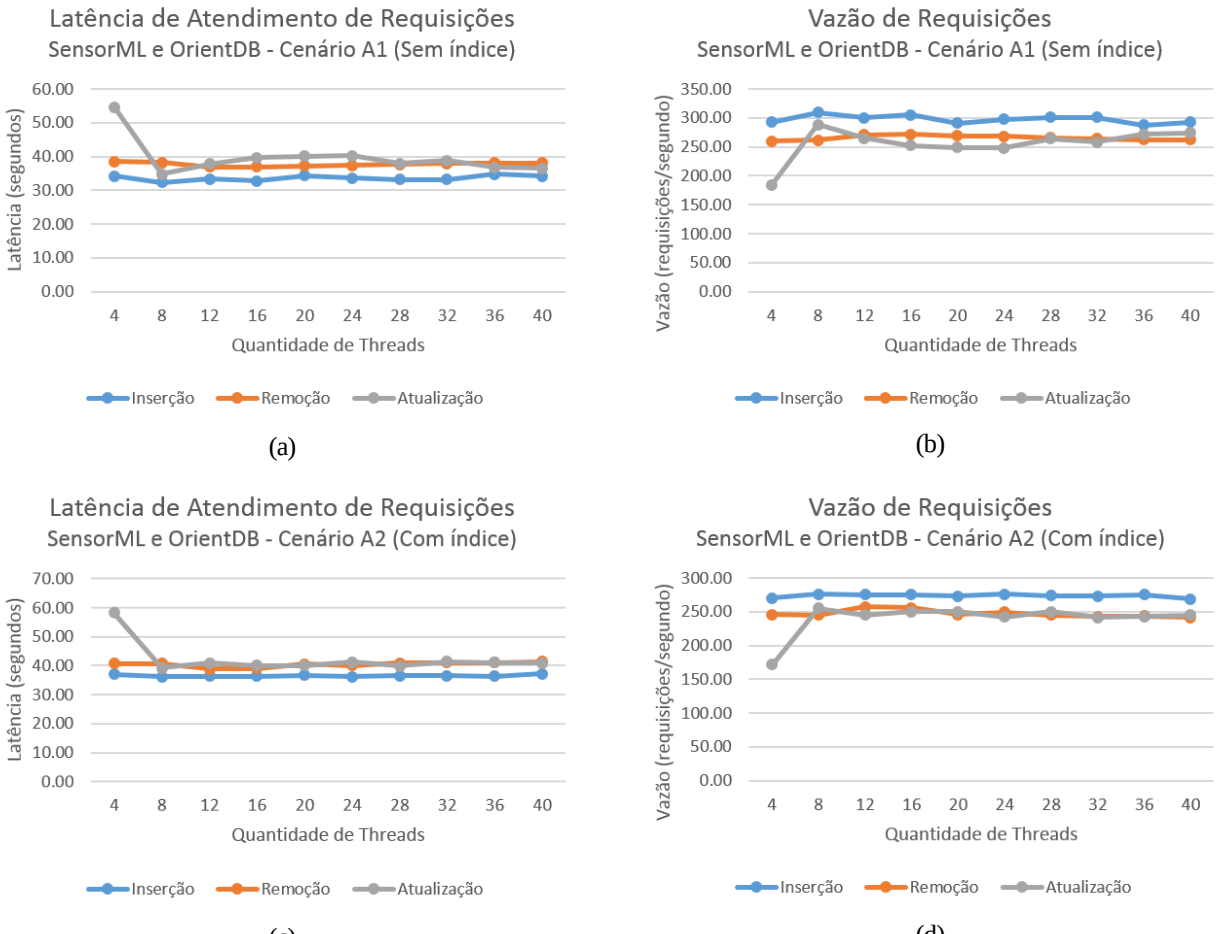
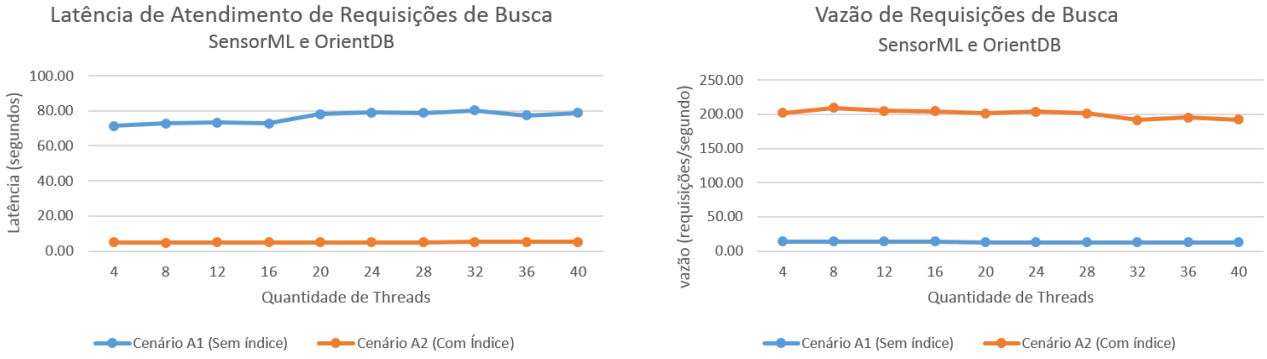


Figura 18. Resultados das operações de inserção, atualização e remoção do Experimento A. (a) Latência no Cenário A1, (b) Vazão no Cenário A1, (c) Latência no Cenário A2 e (d) Vazão no Cenário A2

Em todos os experimentos de vazão, esses dois conjuntos de operações são analisados separadamente. O primeiro conjunto (inserção, atualização e remoção) são as operações realizadas por um produtor de dados, enquanto a operação de busca, do segundo conjunto, é submetida por um consumidor de dados. Em ambas as figuras, o eixo x representa a variação da quantidade de *threads* utilizadas no atendimento de requisições e o eixo y representa as latências, em segundos, e as vazões, em requisições/segundo, obtidas.

Analisando os resultados apresentados na Figura 18, pode-se observar que as operações de inserção, atualização e busca possuem performances semelhantes, com a atualização sendo, em grande parte, a operação mais lenta e, por outro lado, a inserção a mais rápida. Esse comportamento acontece devido à forma como essas operações são implementadas para o OrientDB. A inserção, entre essas três operações, é a mais simples porque ela envolve somente a criação de vértices e arestas. A remoção requer uma travessia no grafo e a remoção de vértices, e respectivas arestas, visitadas nessa busca. Por sua vez, a atualização envolve tanto a remoção de vértices quanto a criação de novos.



(a) (b)

Figura 19. Resultados das operações de busca do Experimento A. (a) Latência e (b) Vazão de atendimento de requisições de busca

Como ainda pode-se observar na Figura 18, o uso de índice possui um impacto negativo nas operações de inserção, busca e remoção. Quando o índice é utilizado, uma vazão menor é apresentada para cada operação. No entanto, esse impacto é consideravelmente pequeno, visto que a vazão de cada operação diminui, em média, 8% com o uso de índice. Pode-se ainda perceber que o aumento na quantidade de *threads* para o atendimento de requisições melhora a performance das operações, mas somente até um certo limitar. A partir desse limiar, inclusive, o desempenho pode voltar a piorar.

Na Figura 19, temos os resultados do Experimento A para a operação de busca. Podemos perceber que o uso de índice sobre característica possui um grande impacto positivo na consulta. O uso de índice resulta em uma vazão de atendimento de requisições de busca por PDs até 16

vezes maior do que o cenário onde não há o seu uso. O impacto positivo proporcionado pelo índice na busca é bem maior que o impacto negativo que o mesmo apresenta nas operações de inserção, atualização e remoção. O aumento na quantidade de *threads*, para a busca, apresenta praticamente nenhuma melhoria e, até mesmo, uma queda no desempenho. Esse comportamento pode ser explicado pelo forte controle de concorrência, com o uso de bloqueios, do OrientDB.

O Experimento B avalia o desempenho do catálogo desenvolvido também com o OrientDB, mas implementando o WoTDataSchema. A Figura 20 mostra os resultados obtidos para a inserção, atualização e remoção de PDs. Percebe-se que a atualização continua sendo a operação mais lenta, enquanto a inserção é a mais rápida. No entanto, diferentemente do experimento anterior, a atualização é, em média, duas vezes mais lenta que a inserção e remoção. Da mesma forma que vimos no Experimento A, o uso de índice degrada ligeiramente o desempenho das inserções, atualizações e remoções. O aumento na quantidade de *threads* para o atendimento de requisições mantém o seu comportamento, melhorando o desempenho até um certo limiar.

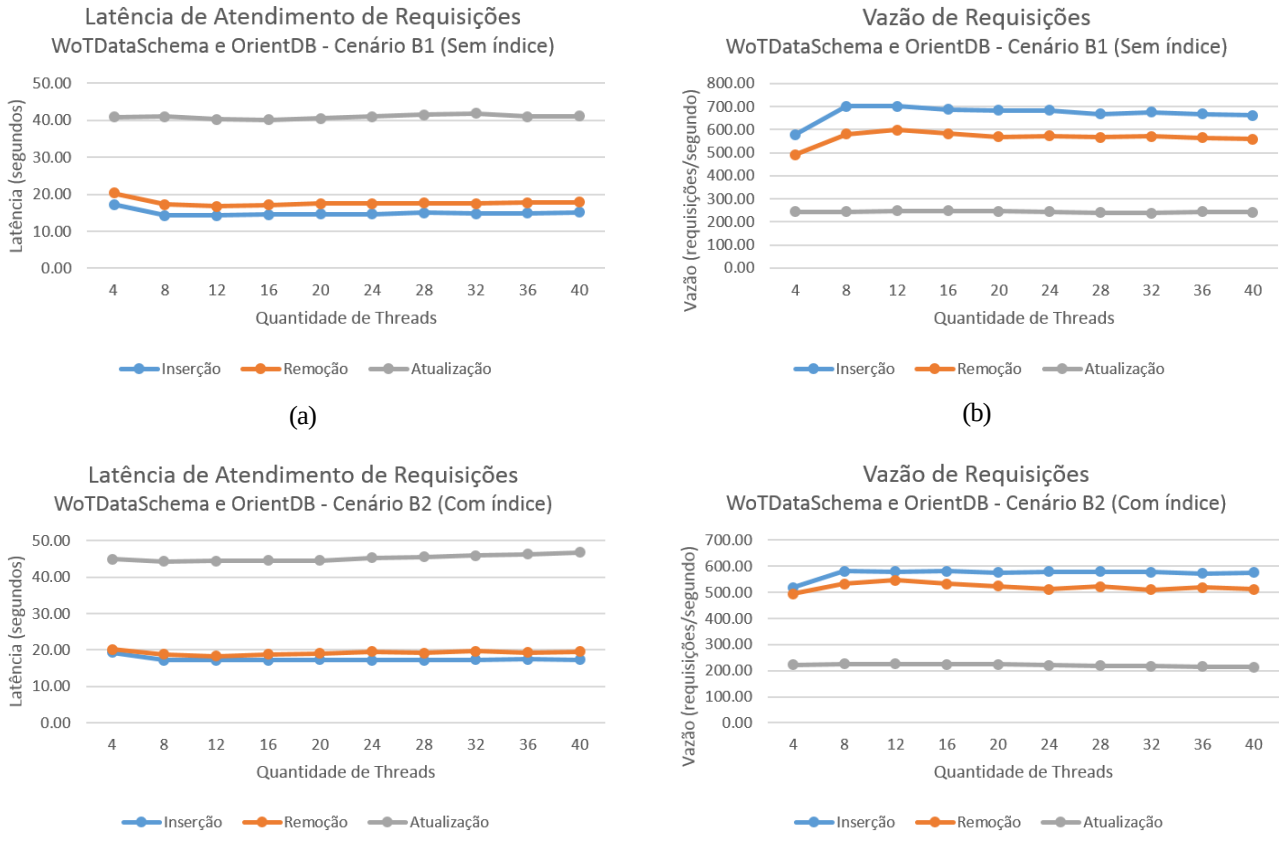
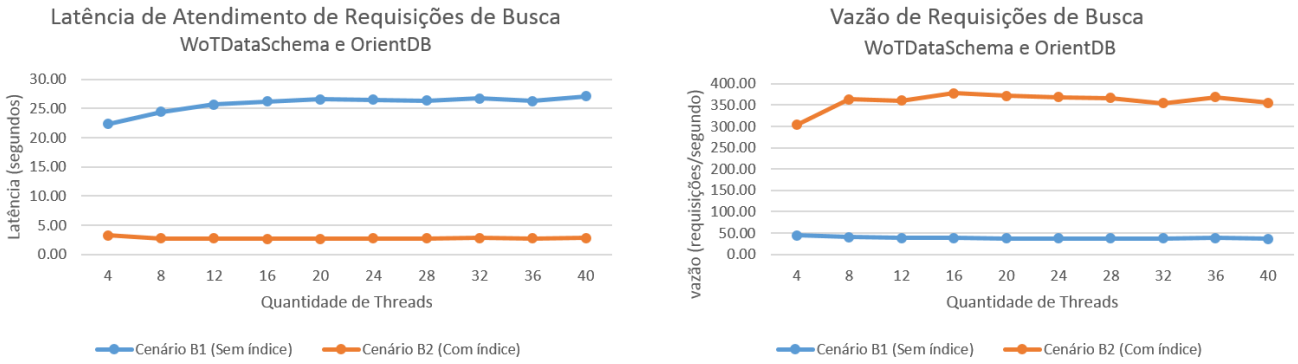


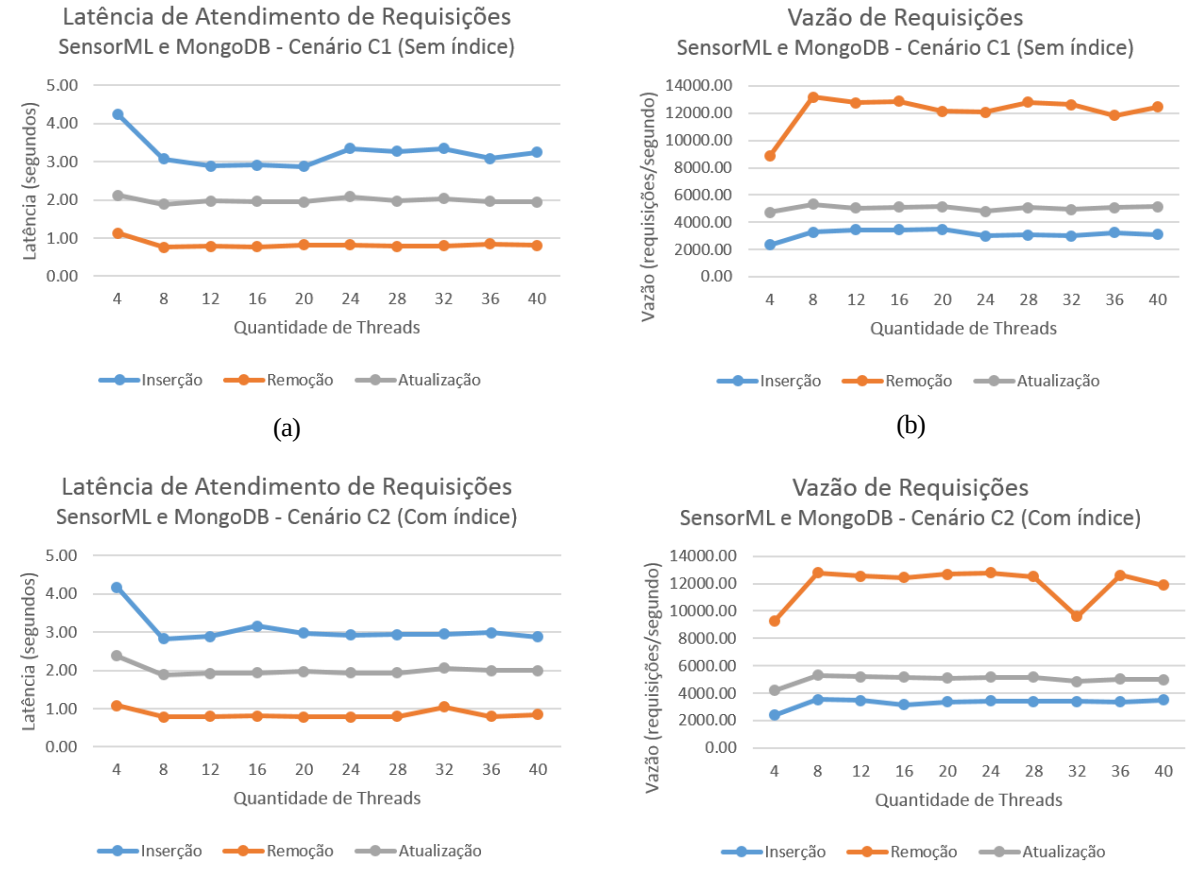
Figura 20. Resultados das operações de inserção, atualização e remoção do Experimento B. (a) Latência no Cenário B1, (b) Vazão no Cenário B1, (c) Latência no Cenário B2 e (d) Vazão no Cenário B2



(a) (b)

Figura 21. Resultados das operações de busca do Experimento B. (a) Latência e (b) Vazão de atendimento de requisições de busca

Na Figura 21, temos os resultados do Experimento B para a operação de busca. Mais uma vez, o uso de índice possui um impacto relevante na performance do catálogo na consulta de busca por PDs. A vazão aumenta em até dez vezes ao usar o índice em característica. Ao aumentar a quantidade de *threads*, o desempenho apresenta uma pequena piora, mostrando, mais uma vez, que o controle de concorrência do OrientDB torna mais lento o atendimento a um grande número de requisições de busca que sejam submetidas de forma concorrente.



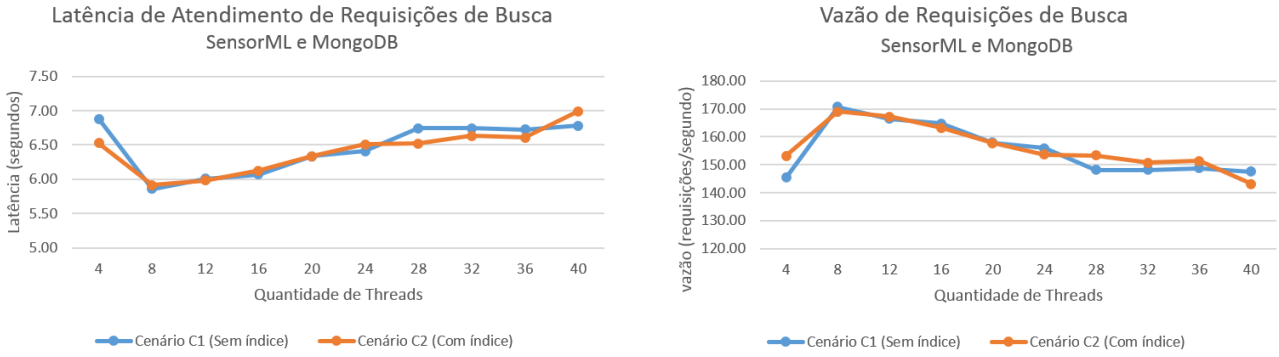
(a) (b)

(c) (d)

Figura 22. Resultados das operações de inserção, atualização e remoção do Experimento C. (a) Latência no Cenário C1, (b) Vazão no Cenário C1, (c) Latência no Cenário C2 e (d) Vazão no Cenário C2

No Experimento C, avaliamos o SensorML e MongoDB. A Figura 22 apresenta os resultados do experimento para a inserção, atualização e remoção. Podemos observar que as três operações possuem performances bem distintas. Em contraste ao que acontece com o OrientDB, a inserção nessa solução é a operação mais lenta entre as três, enquanto a remoção é a mais rápida. Isso ocorre porque a inserção no MongoDB realiza a escrita completa do documento. A atualização apresenta uma performance semelhante, pois também requer a escrita de todos os campos, porém, ela é mais rápida por não envolver a criação de um novo documento em si, reutilizando a chave primária já atribuída. Por fim, a remoção é mais rápida que as demais operações por ela consistir somente de uma marcação dos registros a serem removidos, delegando ao gerenciador do MongoDB a exclusão efetiva dos mesmos [4].

Pode-se também observar, na Figura 22, que o uso de índice praticamente não influencia a vazão das inserções, atualizações e remoções. O desempenho é reduzido, mas em uma escala bem menor do que vimos nos experimentos com o OrientDB. Por fim, podemos perceber que a adição de novas *threads* resulta em uma melhoria mais efetiva no desempenho das operações do catálogo, mas, assim como nos outros experimentos, há um limiar no qual o aumento da quantidade de *threads* não proporciona uma melhoria significativa de performance.



(a) (b)

Figura 23. Resultados das operações de busca do Experimento C. (a) Latência e (b) Vazão de atendimento de requisições de busca

A Figura 23 ilustra os resultados obtidos no Experimento C para a operação de busca. Analisando esses resultados, vemos que o impacto do uso de índice nas consultas é pequeno. Para algumas quantidades de *threads*, inclusive, o desempenho chega a ser pior. Também podemos observar que o uso de mais *threads* para atender requisições oferece uma melhoria no desempenho até um certo ponto, mas, após esse ponto, há uma degradação no mesmo.

Por último, entre os experimentos de vazão, foi executado o Experimento D, avaliando o WoTDataSchema e o MongoDB. Na Figura 24, temos os resultados obtidos para as operações de inserção, atualização e remoção. O comportamento dos mesmos é semelhante ao apresentado no experimento anterior, no qual temos a remoção como a operação mais lenta e a inserção a

mais rápida. Além disso, a influência do uso de índice nessas operações, da mesma forma, também é mínima. Por último, o aumento na quantidade de *threads* possui um impacto também semelhante ao que já vimos, com uma melhoria no desempenho até um certo valor.

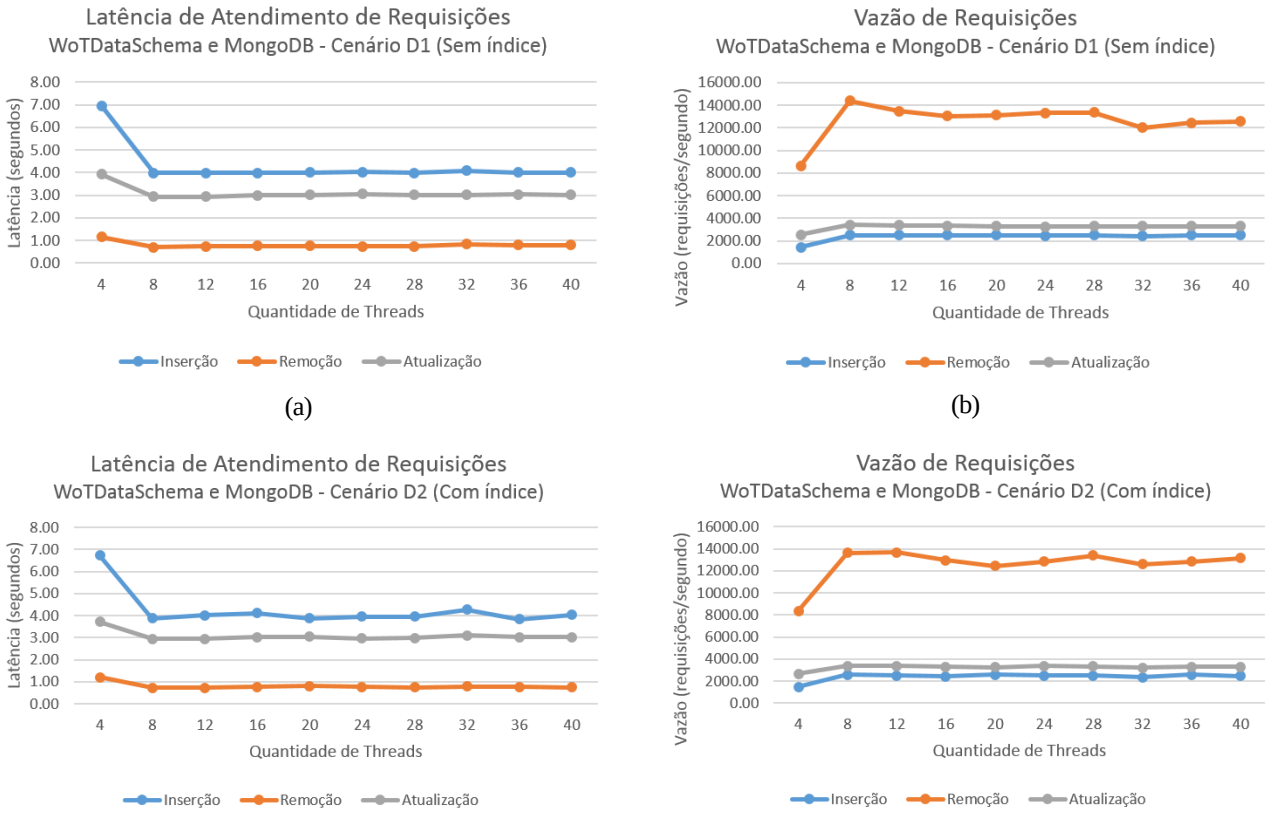


Figura 24. Resultados das operações de inserção, atualização e remoção do Experimento D. (a) Latência no Cenário D1, (b) Vazão no Cenário D1, (c) Latência no Cenário D2 e (d) Vazão no Cenário D2

Na Figura 25, temos os resultados relacionados à operação de busca. O impacto dos índices é ainda mais insignificativo na vazão das consultas, o que mostra limitações na implementação dessas estruturas no MongoDB. Um maior número de *threads* para lidar com as requisições melhora ligeiramente o desempenho, porém, também só até um certo limiar. Esse limiar é relacionado com a quantidade de núcleos de processamento disponíveis.

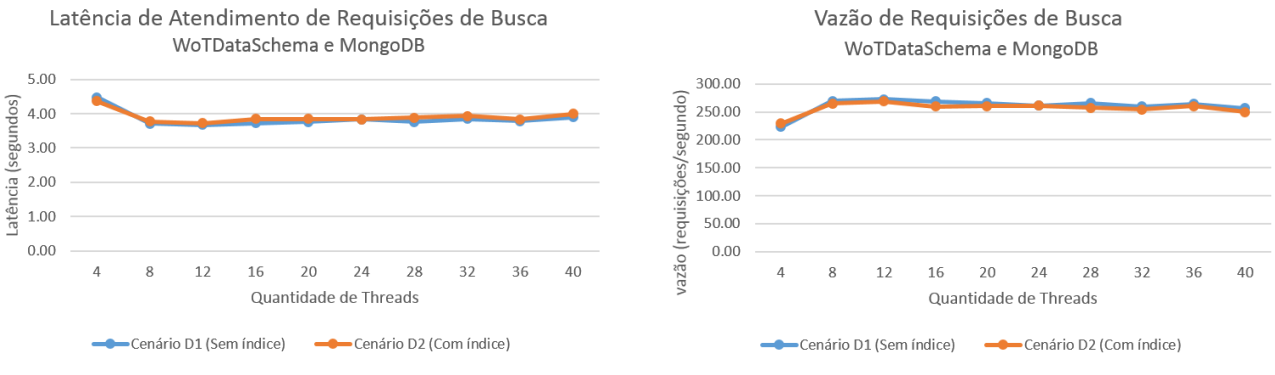
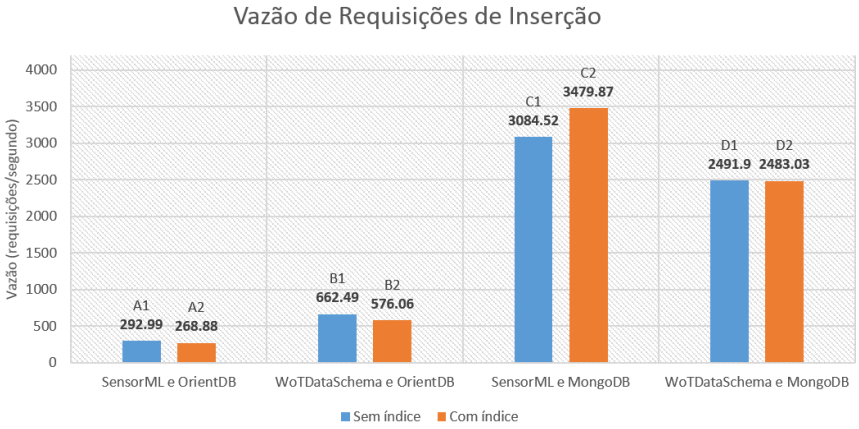
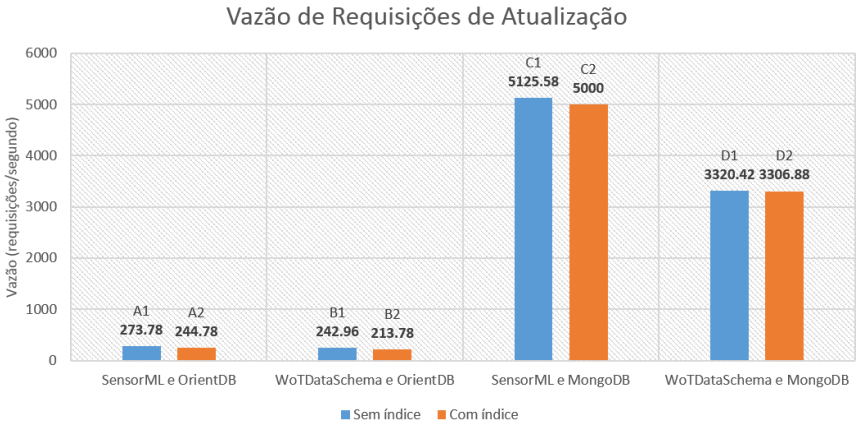


Figura 25. Resultados das operações de busca do Experimento D. (a) Latência e (b) Vazão de atendimento de requisições de busca

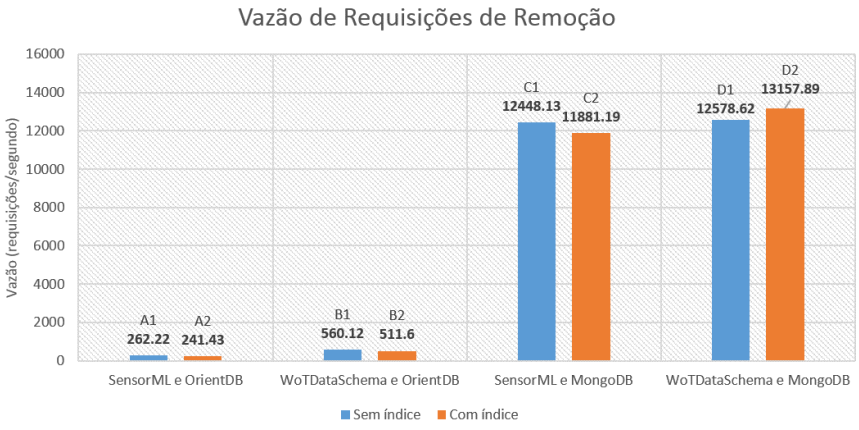
(a)



(b)



(c)



(d)

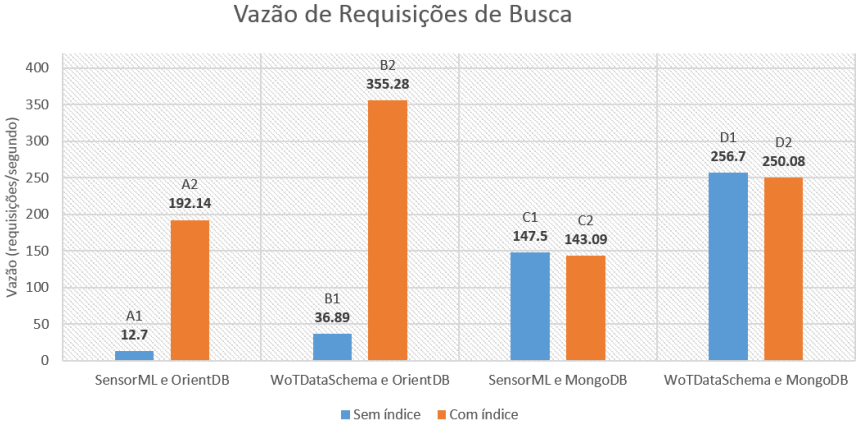


Figura 26. Comparação entre os resultados dos Experimentos A, B, C e D com 40 *threads*. (a) Vazão de inserções, (b) Vazão de atualizações, (c) Vazão de remoções e (d) Vazão de buscas

Com todos os resultados dos experimentos de vazão disponíveis, é possível fazer análises comparativas entre as diferentes soluções de catálogo avaliadas. A Figura 26 apresenta a comparação entre essas soluções, considerando os resultados obtidos ao usar 40 *threads* para o atendimento de requisições. A seguir, discutiremos algumas análises baseadas nesses resultados.

Entre as soluções que fizeram uso do OrientDB, todas as operações do catálogo que usa o WoTDataSchema são bem mais rápidas que as mesmas do catálogo baseado em SensorML. A única exceção é a operação de atualização, onde as duas soluções apresentam uma performance semelhante. Para as demais operações, em todos os cenários avaliados, a vazão é, em média, duas vezes maior quando o WoTDataSchema é implementado. Essa diferença acontece devido a uma menor complexidade da estrutura do grafo usado para modelar um PD no WoTDataSchema. Como temos mais classes de vértice e aresta no SensorML, há um maior custo na criação e remoção de vértices, bem como na travessia pelo grafo.

Entre as soluções baseadas no MongoDB, as operações de inserção e atualização no catálogo que implementa o SensorML são mais rápidas que as correspondentes no catálogo que usa o WoTDataSchema. Essa vantagem do SensorML frente ao WoTDataSchema ocorre porque a serialização da classe *WoTResource*, que representa um PD de acordo com o WoTDataSchema, para JSON requer um custo computacional maior do que a conversão direta a partir da XML que descreve um PD em SensorML. Em contrapartida, a solução de catálogo que usa o WoTDataSchema apresenta uma maior vazão nas operações de busca, devido a estrutura mais simples e menos verbosa do documento representado nesse esquema.

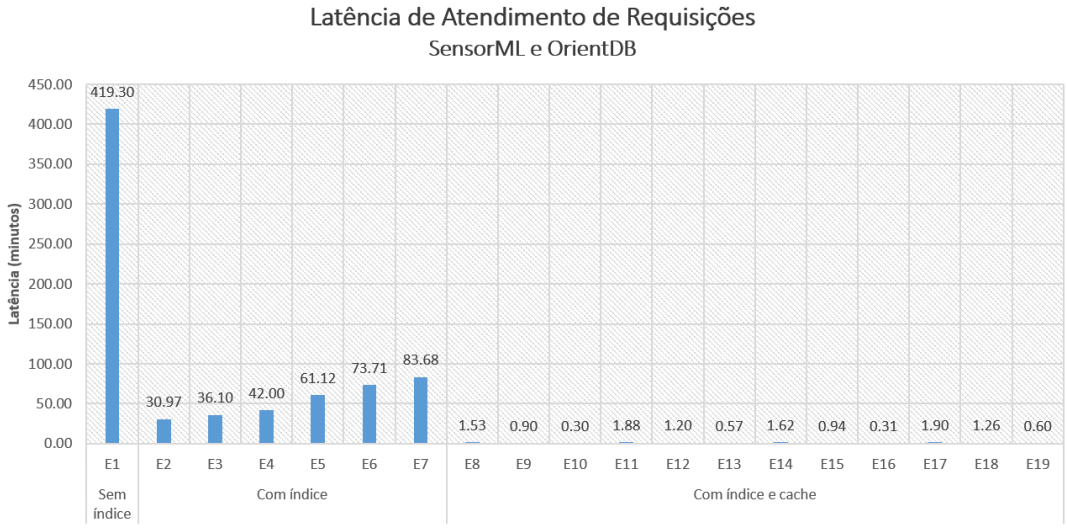
Comparando as soluções de catálogo que usam o OrientDB com as do MongoDB, vemos que as operações de inserção, atualização e remoção são consideravelmente mais rápidas nas soluções baseadas na última, em ambos os modelos de dados. Por exemplo, ao comparar a vazão de atendimento das requisições de inserção dos cenários A1 e C1, apresentada na Figura 26a, vemos que a solução que faz uso do MongoDB, o C1, oferece uma vazão cerca de dez vezes maior. A manipulação de documentos ao invés de grafos para essas operações se mostrou muito mais eficiente.

Contudo, considerando as operações de busca, temos um panorama diferente. Quando não há o uso de índice, o MongoDB ainda apresenta um desempenho melhor, tanto no WoTDataSchema quanto no SensorML. No entanto, quando o índice sobre característica é usado, o OrientDB oferece uma vazão bem maior nos dois esquemas, como podemos ver na Figura 26d. Esse cenário específico, quando há busca com a utilização de índice, é o único em que o OrientDB apresenta uma vantagem considerável em relação ao MongoDB.

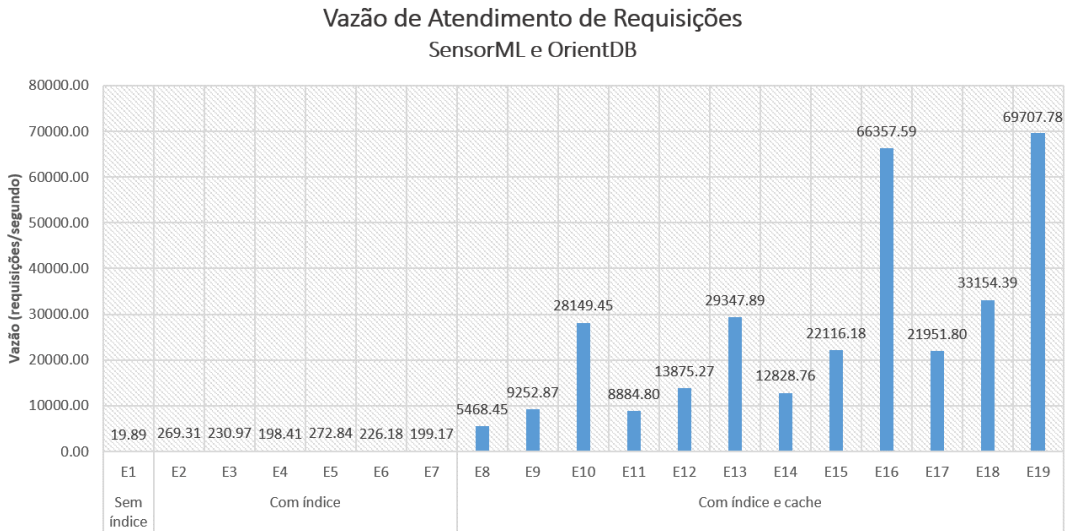
b. Experimentos de resiliência

Com o objetivo de avaliar a robustez das soluções de catálogos desenvolvidas, foram realizados quatro experimentos de resiliência: i) Experimento E; ii) Experimento F; iii) Experimento G e iv) Experimento H. Os experimentos simularam os mesmos cenários, mas cada um avaliando um protótipo diferente de catálogo.

Os cenários simulados em todos os experimentos de resiliência são apresentados na Tabela 5. Em alguns cenários, temos 10000 CDs submetendo requisições e, em outros, 25000 CDs. Paralelamente às submissões de busca de PDs, há também submissões de inserção a partir de PDs que não são previamente cadastrados no *testbed*. Dessa forma, com uma carga alta de requisições concorrentes sendo atendidas em ciclos, não só o desempenho dos catálogos é avaliado, como também o comportamento dos mesmos em condição de estresse.



(a)



(b)

Figura 27. Resultados do Experimento E. (a) Latência e (b) Vazão de Atendimento de Requisições

A Figura 27 mostra os resultados obtidos no Experimento E, que avalia o catálogo baseado no SensorML e OrientDB. Uma das primeiras observações que podemos fazer é o impacto que o uso de índice tem na vazão de atendimento das requisições, algo já constatado nos experimentos anteriores. Pode-se perceber isso a partir dos cenários E1 e E2. O cenário E2, com o uso de índice, apresenta uma vazão 14 vezes maior do que a de E1. Analisando os resultados obtidos nos cenários E2 ao E7, onde há o uso de índice, vemos que aumento da quantidade de ciclos resulta em um aumento na duração do experimento, mas ele não altera significativamente a vazão de requisições atendidas.

Ainda analisando os cenários E2 ao E7, vemos que, na medida em que uma maior porcentagem de PDs já estão previamente cadastrados, a latência de atendimento de requisições cresce. O aumento dessa porcentagem implica em um número menor de requisições de inserção, porém, as requisições de busca passam a manipular um número maior de PDs. Nesse caso, com o uso apenas de índice, o aumento da quantidade de PDs consultados teve um impacto maior no desempenho do catálogo do que a diminuição da quantidade de inserções.

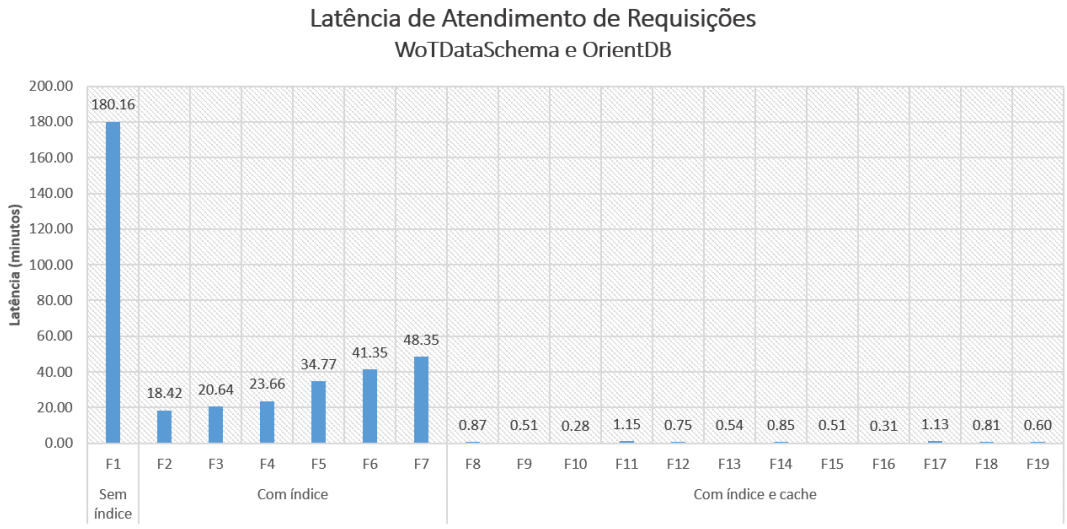
Nos experimentos de resiliência, além do uso de índice, o uso de *cache* também é avaliado. Observando os resultados dos cenários E8 ao E19, podemos ver o impressionante impacto que o *cache* proporciona no desempenho do catálogo. Por exemplo, a vazão obtida no cenário E8 é 20 vezes maior do que em E2, no qual somente o índice é usado. Nesses cenários, o aumento do número de ciclos resulta em uma melhoria na vazão. Esse resultado ocorre porque, com o aumento da quantidade de ciclos, mais consultas são submetidas ao catálogo, mas como o conjunto de consultas distintas é pequeno (50 consultas), a probabilidade de *cache hit* cresce, reduzindo a atividade de leitura em disco rígido, diminuindo, assim, a latência. A mesma melhoria foi também observada com o aumento do número de CDs.

Pode-se ainda perceber, avaliando os resultados dos cenários E8 ao E19, que o aumento da carga de inserção, com uma maior porcentagem de PDs previamente cadastrados, resulta em um aumento da vazão, ao contrário do que acontece nos cenários sem o *cache*. Essa melhoria da vazão acontece porque, com uma menor carga de inserção, o *cache* mantém o seu conteúdo por uma maior duração, pois quando um novo PD é inserido, o *cache* é esvaziado, para evitar inconsistências. Consequentemente, com menos inserções, um maior número de consultas é respondido pelo *cache*, evitando leituras em disco.

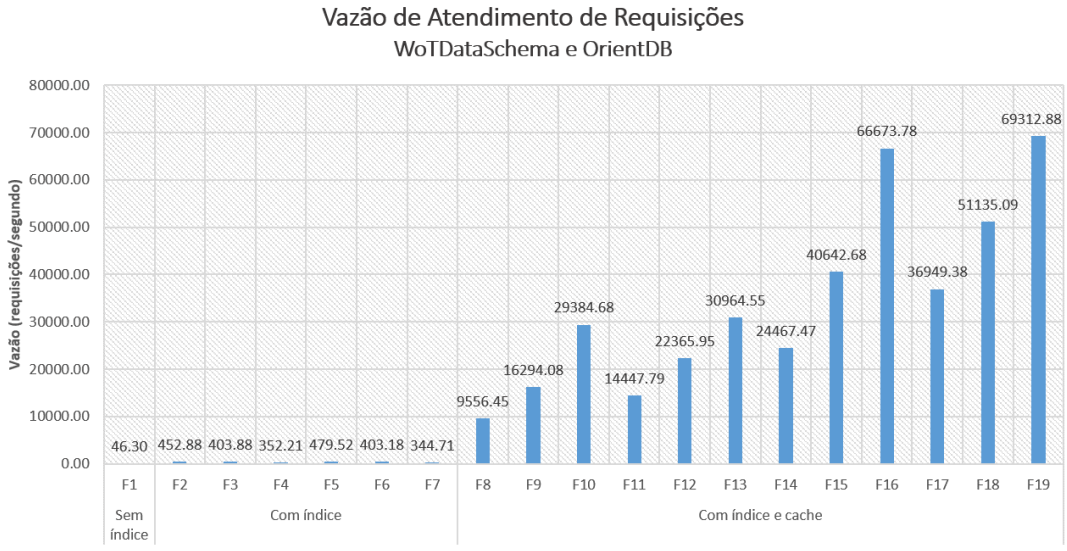
Durante a execução dos Experimentos E e F, ambos usando o OrientDB, pode-se perceber que o consumo de memória se manteve estável, em torno de 2,8 GB. Além disso, o experimento consumiu cerca de 95% da capacidade de processamento da máquina, mas não houve problemas na execução.

O Experimento F avalia o catálogo implementado em OrientDB com o WoTDataSchema. Os resultados desse experimento são apresentados na Figura 28. As observações que podemos fazer ao analisar esses resultados são semelhantes ao que vimos no Experimento E. O uso de índice também oferece uma melhoria significante na vazão de requisições. Podemos constatar isso ao comparar os cenários F1 e F2. A vazão obtida ao simular F1 foi de 46.3 requisições por segundo, enquanto a de F2 foi 452.88 requisições por segundo.

Também se percebe que o uso de *cache*, aliado à criação de índice, proporciona um aumento ainda maior na vazão. Comparando o cenário F8 com o F2, observamos uma melhoria na vazão em, aproximadamente, 2100%. O comportamento da variação no número de ciclos, de CDs e na porcentagem de PDs já cadastrados no testbed foi exatamente o mesmo observado no experimento anterior, em todos os cenários analisados.



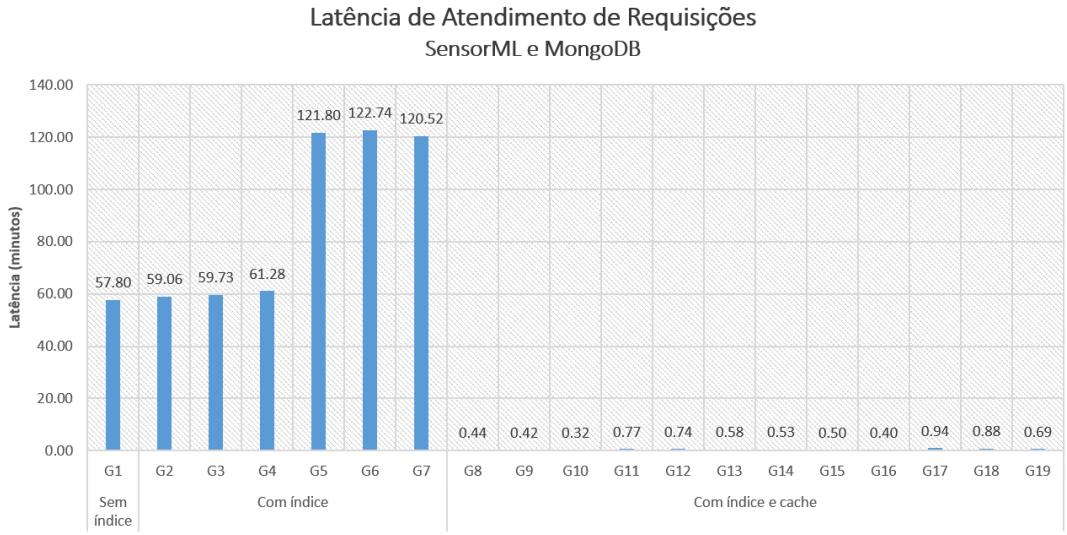
(a)



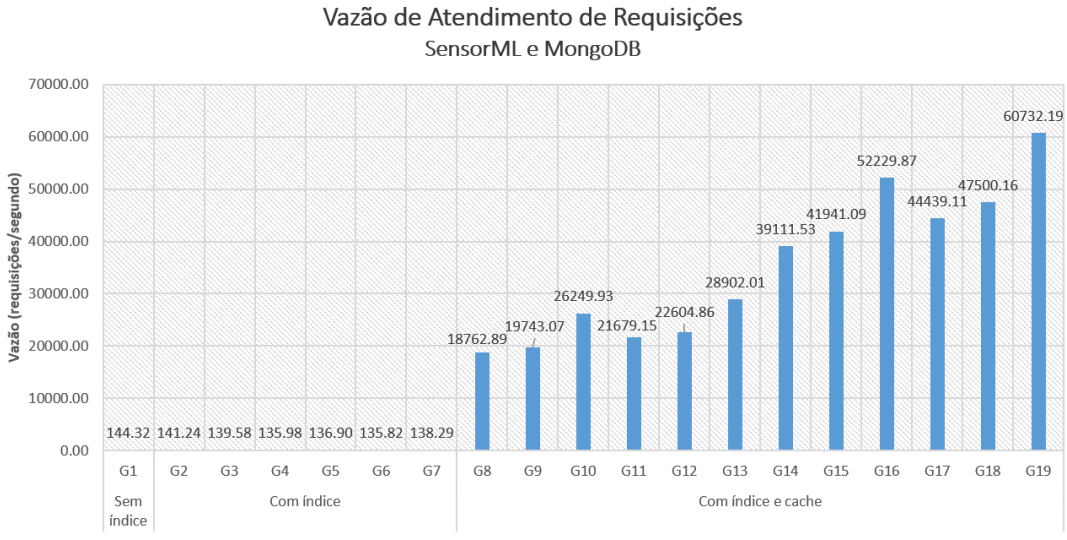
(b)

Figura 28. Resultados do Experimento F. (a) Latência e (b) Vazão de Atendimento de Requisições

No Experimento G, os cenários simulados foram executados com o catálogo baseado em SensorML e MongoDB. A Figura 29 ilustra os resultados desse experimento. Um comportamento que já foi verificado nos experimentos de vazão, em relação ao MongoDB, é a ineficiência dos índices ao lidar com consultas em característica, tanto ao usar SensorML bem como o WoTDataSchema. Nos experimentos de resiliência, mais uma vez, essa ineficiência é demonstrada. Comparando os cenários G1 e G2, percebe-se que há uma pequena degradação no desempenho do catálogo, havendo uma diminuição da vazão em torno de 2%.



(a)



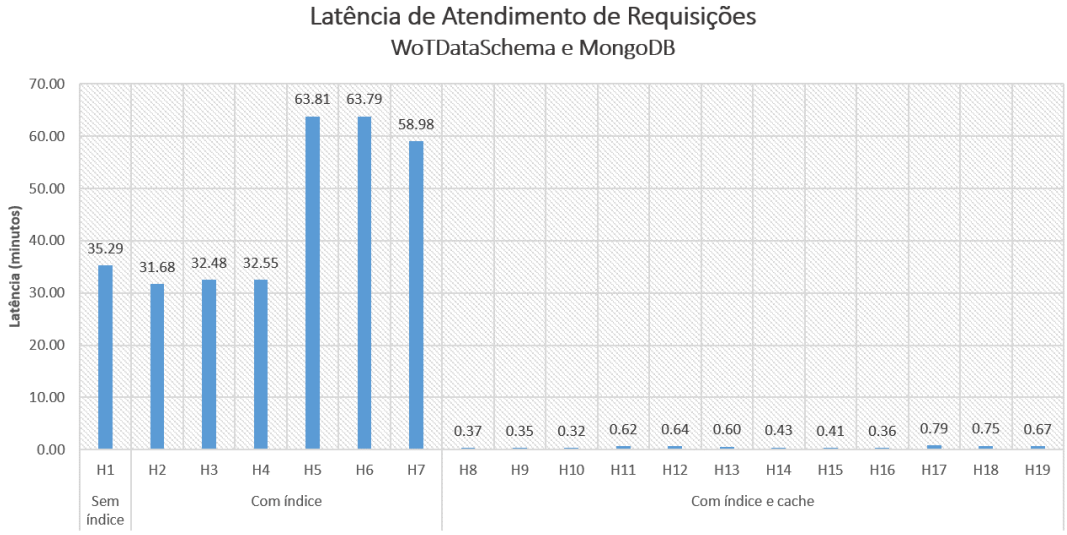
(b)

Figura 29. Resultados do Experimento G. (a) Latência e (b) Vazão de Atendimento de Requisições

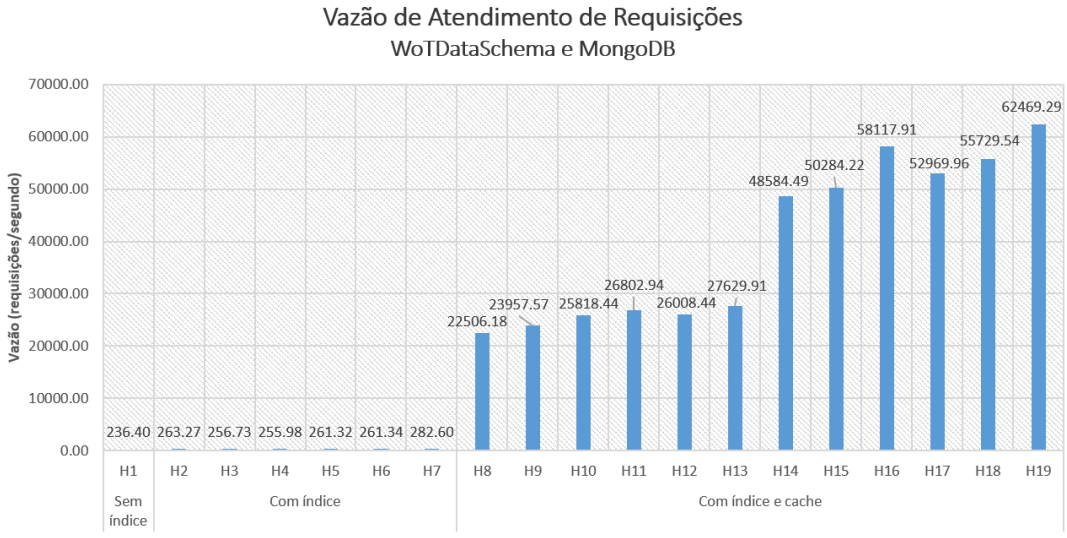
Analisando os cenários G2 ao G7, nos quais há o uso de índice, temos que, além do impacto negativo que os índices possuem, o aumento no número de ciclos de execução mantém, aproximadamente, constante a vazão de atendimento de requisições. Do mesmo modo,

podemos observar que o aumento da quantidade de PDs cadastrados previamente no *testbed* possui um impacto com pouca relevância no desempenho do catálogo.

Nos cenários G8 ao G19, temos o uso de *cache* que, mais uma vez, mostrou um grande ganho de eficiência. A vazão apresentada em G8 é cerca de 130 vezes maior do que a que temos em G2, mas, vale salientar que o uso de índice em G2 não contribuiu em nada com a melhoria de desempenho. Tanto o aumento da quantidade de ciclos de execução quanto de CDs resulta em uma quantidade maior de requisições submetidas ao catálogo. Por sua vez, o maior número de requisições proporciona uma melhoria na vazão de atendimento de requisições, devido ao *caching* de resultados de consultas. Um percentual maior de PDs cadastrados também melhora a vazão do catálogo, por aumentar a taxa de *cache hit*.



(a)



(b)

Figura 30. Resultados do Experimento H. (a) Latência e (b) Vazão de Atendimento de Requisições

Por último, temos o Experimento H, avaliando o catálogo implementado a partir do WoTDataSchema e MongoDB. Os resultados do experimento são apresentados na Figura 30. Observando os cenários H1 e H2, percebemos que, diferentemente do que vimos no experimento anterior, o uso de índice melhorou o desempenho, porém o ganho não é muito significante. Nos cenários H2 ao H8, em todos os valores para quantidade de ciclos e PDs cadastrados, a vazão se mantém constante, entre 260 e 280 requisições/segundo.

O uso do *cache* também apresenta um grande ganho de eficiência, como podemos observar nos resultados dos cenários H9 ao H19. Da mesma forma como foi constatado nos experimentos anteriores, o desempenho do *cache* melhora com o aumento do número de ciclos, CDs e PDs previamente cadastrados.

Durante a execução dos experimentos que usaram o MongoDB, os experimentos G e H, o consumo da memória foi maior do que o observado nos experimentos anteriores, em torno de 3,5 GB. A capacidade de processamento também foi mais utilizada, sendo totalmente consumida na maior parte dos experimentos. No entanto, não houve problemas de concorrência ao longo da execução.

Após analisar os resultados dos experimentos de resiliência individualmente, serão comparados, agora, os resultados entre diferentes experimentos, com o objetivo de investigar quais soluções de catálogo se mostraram mais eficientes ao lidar com um grande número de requisições enviadas simultaneamente, um cenário comum ao se trabalhar com a *Web of Things*.

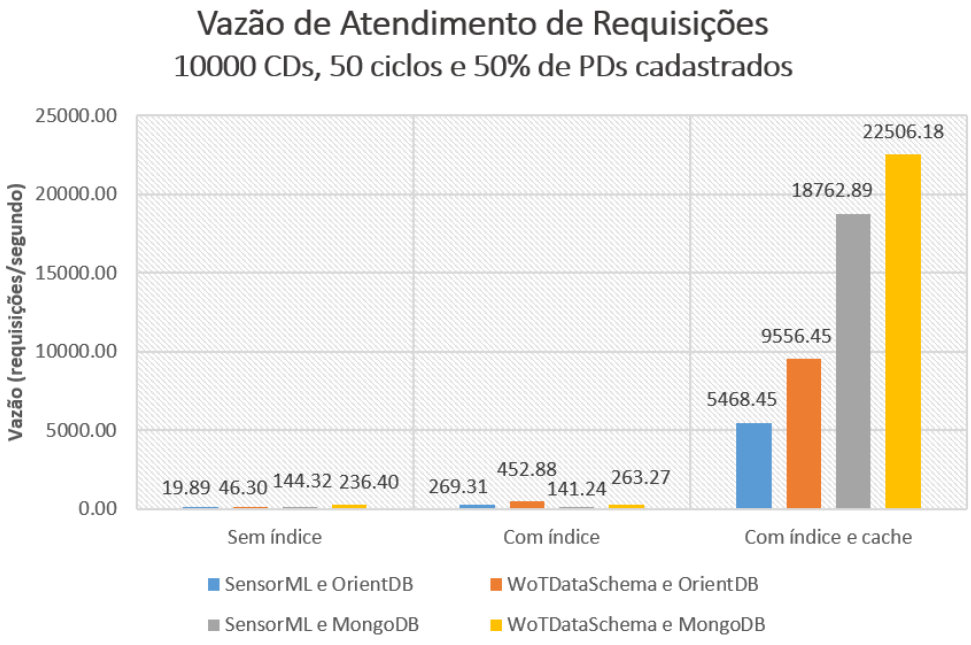


Figura 31. Comparação entre os resultados dos Experimentos E, F, G e H, com 10000 CDs, 50 ciclos e 50% de PDs cadastrados

A Figura 31 apresenta uma comparação entre os resultados dos experimentos de resiliência, com 10000 CDs, 50 ciclos e 50% de PDs previamente cadastrados. Esses valores são utilizados nessa comparação porque todas as soluções (sem índice, com índice e com índice e *cache*) foram avaliadas nesse cenário.

Considerando primeiramente as soluções sem o uso de índice, pode-se observar que os catálogos implementados com o MongoDB apresentam um melhor desempenho em relação ao OrientDB. Esse resultado reflete a análise comparativa que fizemos a partir dos experimentos de vazão, na qual vimos que as operações de inserção e busca no MongoDB, quando não há índice, são mais rápidas que no OrientDB. Entre as soluções desenvolvidas com o OrientDB, o WoTDataSchema oferece uma melhor performance, do mesmo modo que acontece com o MongoDB. Nos experimentos de vazão vimos que, para o MongoDB, a inserção é mais rápida com o SensorML, mas a busca, por sua vez, é melhor com o WoTDataSchema. Como esses experimentos de resiliência possuem uma carga maior de consulta do que de inserção, o WoTDataSchema se sobressaiu.

Ao analisar as soluções de catálogo que fazem uso de índice, temos a situação contrária. Como a implementação do MongoDB não é eficiente considerando os esquemas de dados e cenários avaliados, o OrientDB apresenta uma melhor performance. Uma das possíveis razões que podem explicar o mau desempenho da criação de índice no OrientDB, para os experimentos realizados nesse trabalho, está relacionado ao tamanho do índice. Como as estruturas dos documentos, para ambos os modelos, são relativamente complexas, o tamanho do índice pode ser grande o suficiente para não poder ser mantido na memória RAM, o que requer leituras em disco rígido, aumentando, assim, a latência das operações. Em ambos os bancos de dados NoSQL, o WoTDataSchema oferece uma melhor vazão que o SensorML.

Por fim, entre as soluções que usam tanto índice como *cache*, os catálogos baseados no MongoDB apresentam uma melhor performance. No entanto, essa observação só é verdadeira nos cenários com somente 50% dos PDs cadastrados previamente. Quanto menor essa porcentagem, maior é a carga da inserção do experimento. Como os experimentos de vazão demonstraram, o MongoDB, em ambos os modelos de dados, é bem mais eficiente que o OrientDB na inserção de novos dados. Porém, o OrientDB é mais eficiente na busca quando há o uso de índices.

Desse modo, é presumível que em um cenário no qual a carga de inserção é consideravelmente menor que a carga de busca – sendo, portanto, predominante as requisições de buscas –, as soluções implementadas com o OrientDB possam apresentar um melhor desempenho.

Com a finalidade de verificar essa hipótese, fizemos a análise comparativa ilustrada na Figura 32. Nessa figura, temos a comparação entre os resultados dos experimentos de resiliência entre todos os cenários com 100% de PDs cadastrados, com índice e *cache*. Essa análise inclui somente as soluções baseadas em WoTDataSchema, pois, para cada banco de dados NoSQL, esse modelo foi o mais eficiente nesses experimentos. Com todos os PDs já cadastrados, não há requisições de inserção, ou seja, a carga de requisições é exclusiva de consultas de busca.

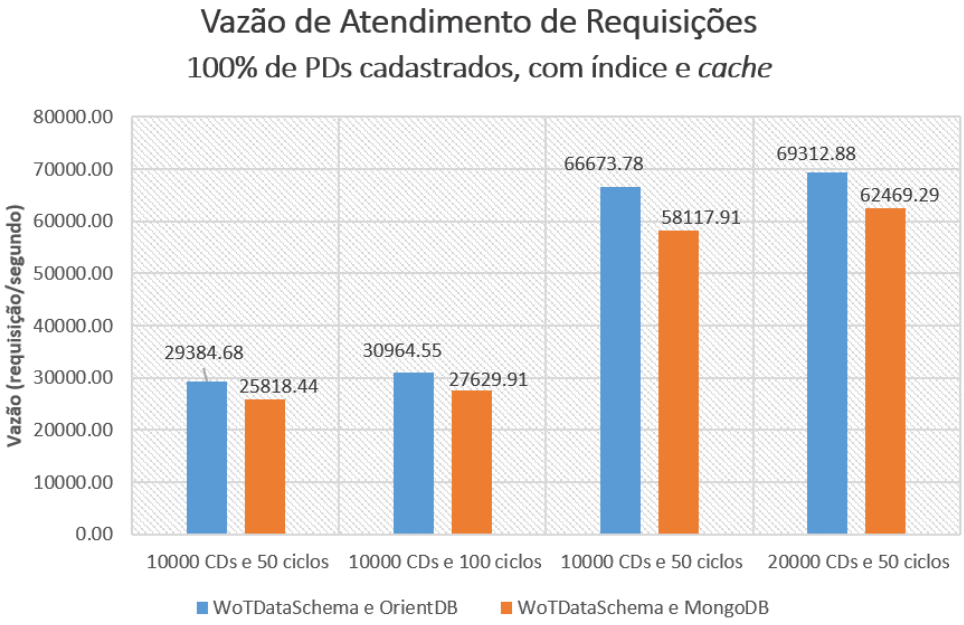


Figura 32. Comparação entre os resultados dos Experimentos F e H, com 100% de PDs cadastrados, índice e *cache*

Nesse cenário específico, com uma carga de busca bem maior que a de inserção, pode-se observar que a solução baseada em WoTDataSchema e OrientDB apresenta o melhor desempenho. Temos esse resultado por dois fatores:

i. O WoTDataSchema é um modelo mais simples que o SensorML, possuindo uma estrutura mais compacta que torna menor o custo computacional para as buscas;
ii. O OrientDB oferece uma implementação mais eficiente de indexação para os modelos de dados e cenários avaliados.

4.4. Considerações Finais

Esse capítulo apresentou e descreveu os experimentos realizados nesse trabalho, com o objetivo de avaliar diferentes soluções de catálogos de fontes de dados para a *Web of Things*. Em seguida, os resultados desses experimentos foram analisados. No próximo capítulo, teremos uma discussão geral sobre os tópicos abordados nesse trabalho e os resultados obtidos nos experimentos.

5. CONCLUSÃO

Este trabalho apresentou uma análise comparativa entre soluções de catálogo de produtores de dados para a *Web of Things*, implementados a partir de dois modelos de dados, o SensorML e o WoTDataSchema, e dois bancos de dados NoSQL, o OrientDB e o MongoDB. Inicialmente, foi introduzido o contexto do processamento de dados na WoT, que envolve diversos desafios devido ao grande volume, heterogeneidade e autonomia das fontes de dados. Diante desses desafios, foi proposto o armazenamento de metadados de fontes, por meio de catálogos, como uma solução para o desenvolvimento de um mecanismo eficiente para a publicação e descoberta de produtores de dados na WoT.

Vimos como os bancos de dados NoSQL possuem características que atendem os requisitos de escalabilidade, alta disponibilidade e flexibilidade para a implementação de catálogos para a *Web of Things*. Foram descritos diversos modelos NoSQL, entre os quais, dois foram selecionados para o desenvolvimento de catálogos avaliados nesse trabalho: o orientado a grafos, representado pelo OrientDB, e o orientado a documentos, representado pelo MongoDB. Em seguida, detalhamos o SensorML e o WoTDataSchema, esquemas de dados para a representação de dispositivos e fontes de dados da WoT. Neste trabalho, foram desenvolvidos quatro protótipos de catálogos com base nessas soluções NoSQL e esquemas de dados.

Finalmente, foram apresentados os experimentos propostos nessa pesquisa, com o objetivo de avaliar a performance dos protótipos de catálogo desenvolvidos, e uma análise dos resultados obtidos a partir da simulação e execução desses experimentos.

5.1. Contribuições

A principal contribuição deste trabalho foi a execução de experimentos que permitiram avaliar o desempenho e robustez tanto de bancos de dados NoSQL bem como de esquemas de dados na implementação de catálogos para a *Web of Things*. Foram realizados dois tipos de experimentos: i) experimentos de vazão e ii) experimentos de resiliência.

Ao final desses experimentos, foi possível chegar a conclusões acerca das soluções de catálogo desenvolvidas. Primeiramente, foi demonstrado o relevante impacto que a criação de índices possui no OrientDB, otimizando bastante as consultas de busca. Por outro lado, ficou evidente que o uso de índices no MongoDB, para os cenários simulados, oferece um ganho pouco significativo de desempenho. Já o uso de *cache* se mostrou muito importante para alcançar um bom desempenho para todos os catálogos avaliados, aumentado consideravelmente a vazão de atendimento de requisições de consulta.

Ao comparar os resultados dos experimentos para cada catálogo, nós percebemos que há dois casos principais, considerando os cenários analisados, nos quais há um melhor candidato entre as soluções avaliadas:

- Carga de trabalho com mais requisições de inserção, atualização e remoção: os experimentos de vazão mostraram que o MongoDB possui um melhor desempenho que o OrientDB ao lidar com essas requisições. O modelo de dados em que os melhores resultados foram obtidos, para o MongoDB, foi o SensorML. Dessa forma, a solução de catálogos mais adequada para uma carga de requisições que envolve, de forma majoritária, inserções, atualizações e remoções é o SensorML e o MongoDB;
- Carga de trabalho com mais requisições de consulta: quando há o uso de índice, o OrientDB apresentou um desempenho bem superior ao MongoDB no atendimento de requisições de busca. O WoTDataSchema foi o modelo em que os melhores resultados foram alcançados. Portanto, para uma aplicação na qual há uma carga maior de consultas, o catálogo implementado em OrientDB usando o WoTDataSchema para modelar os PDs se mostrou a solução mais apropriada.

É importante notar que os resultados obtidos nos experimentos executados também dependem da capacidade de processamento e de armazenamento da máquina utilizada. Nesse trabalho, foi utilizado um computador de pequeno porte. Melhores resultados certamente podem ser gerados ao usar servidores ou *clusters*.

5.2.Dificuldades Encontradas

Uma das maiores dificuldades encontradas durante o desenvolvimento desse trabalho foi a de lidar com uma grande quantidade de cenários simulados, resultando em longos experimentos. Alguns cenários tiveram que ser reformulados devido a limitação de tempo disponível para a simulação dos mesmos. Outra dificuldade enfrentada foi a necessidade de otimizar as consultas submetidas e os índices criados a fim de obter os melhores resultados possíveis para cada solução de catálogo, permitindo, assim, uma avaliação justa.

5.3. Trabalhos Futuros

Dentre as possíveis contribuições futuras, pretende-se realizar mais experimentos envolvendo as soluções de catálogo implementadas, com uma maior carga de requisições e um uso mais abrangente do SensorML e o WoTDataSchema. Outro desafio é o de avaliar o desempenho de outros bancos de dados NoSQL, tais como o Cassandra, que adota o modelo orientado a colunas.

REFERÊNCIA BIBLIOGRÁFICA

[1] BARNAGHI, P.; SHETH, A.; HENSON, C. **From data to actionable knowledge: Big data challenges in the web of things**. Intelligent Systems, IEEE, vol. 28, no. 6, p. 6–11, 2013.

[2] BISCHOF, S.; KARAPANTELAKIS, A.; SHETH, A.; MILEO, A.; NECHIFOR, S.; BARNAGHI, P. **Semantic Modelling of Smart City Data**. W3C Workshop on the Web of Things, Berlim, Alemanha, 2014.

[3] GUBBI, J. et al. **Internet of Things (IoT): A vision, architectural elements, and future directions**. Future Generation Computer Systems, v. 29, no. 7, p. 1645-1660, 2013.

[4] OLIVEIRA, M. I. S., GAMA, K. S., LOSCIO, B. F. **Análise de Desempenho de Catálogo de Produtores de Dados para Internet das Coisas baseado em SensorML e NoSQL**. XIV Workshop em Desempenho de Sistemas Computacionais e de Comunicação, 2015.

[5] GUSTAFSON, S.; SETH, A. **The Web of Things**. Computing Now, v. 7, no. 3, 2014. Disponível em: http://www.computer.org/portal/web/computingnow/archive/march2014. Acesso em: 25 de Abril de 2015.

[6] WILDE, E.; MICHAHELLES, F.; LÜDER, S. **Leveraging the Web Platform for the Web of Things.** W3C's Web of Things Workshop, Berlim, Alemanha, 2014.

[7] MESHKOVA, E. et al. **A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks**. Computer networks, v. 52, n. 11, p. 2097-2128, 2008.

[8] ELMASRI, R.; NAVATHE, S. **Fundamentals of Database Systems**. Addison Wesley, 6th ed., 2011.

[9] LOSCIO, B. F.; OLIVEIRA, H. R.; PONTES, J. C. S. **NoSQL no desenvolvimento de aplicações Web colaborativas**. VIII Simpósio Brasileiro de Sistemas Colaborativos, 2011.

[10] ABRAMOVA, V.; BERNARDINO, J. **NoSQL databases: MongoDB vs Cassandra**. International C* Conference on Computer Science and Software Engineering, p. 14–22, 2013.

[11] TIWARI, S. **Professional NoSQL**. Wrox, 2011.

[12] CATTELL, R. **Scalable SQL and NoSQL data stores**. ACM SIGMOD Record, v. 39, n. 4, 2010.

[13] KLEIN, J. et al. **NoSQL Data Store Technologies**, Software Engineering Institute, Carnagie Mellon University, 2014.

[14] HECHT, R.; JABLONSKI, S. **NoSQL evaluation: A use case oriented survey**. Cloud and Service Computing, 2011 International Conference, p. 336-341, 2011.

[15] MCMURTRY, D. et al. **Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence**. Microsoft Press, Redmond, WA, 2013.

[16] **OrientDB Manual**. Orient Technologies, version 2.0. Disponível em: http://orientdb.com/docs/last/. Acesso em: 29 de Junho de 2015.

[17] JIRKA, S.; BRÖRING, A.; STASCH, C. **Discovery Mechanisms for the Sensor Web**. Sensors, v. 9, no. 4, 2009.

[18] BRÖRING, A. et al. **New generation sensor web enablement**. Sensors, v. 11, no. 3, p. 2652-2699, 2011.

[19] OLIVEIRA, M. I. S; GAMA, K. S.; LOSCIO, B. F. **Waldo: Serviço para Publicação e Descoberta de Produtores de Dados para Middleware de Cidades Inteligentes**. XI Simpósio Brasileiro de Sistemas de Informação, 2015.

[20] BOTTS, M.; ROBIN, A. **OGC SensorML: Model and XML Encoding Standard**. Open Geospatial Consortium, 2013.

[21] GRAY, A. et al. **A semantic sensor web for environmental decision support applications**. Sensors, v. 11, no. 9, p. 8855-8887, 2011.

[22] GARCÍA-CASTRO, R.; CORCHO, O.; HILL, C. **A Core Ontological Model for Semantic Sensor Web Infrastructures**. International Journal on Semantic Web and Information Systems (IJSWIS), 8(1), p. 22-42, 2012.

[23] HUSEMANN, M.; RITTER, N. **Data Source Management and Selection for Dynamic Data Integration**. Second International Workshop on Resource Discovery (RED 2009), p. 49-65, 2009.

[24] COMPTON, M. et al. **The SSN Ontology of the W3C Semantic Sensor Network Incubator Group**. Web Semantics: Science, Services and Agents on the World Wide Web, North America, 2012.

[25] W3C. **Data Catalog Vocabulary (DCAT)**. W3C Recommendation, 2014. Disponível em: http://www.w3.org/TR/vocab-dcat/. Acesso em: 10 de Julho de 2015.

[26] W3C. **PROV-O: The PROV Ontology**. W3C Recommendation, 2013. Disponível em: http://www.w3.org/TR/prov-o/. Acesso em: 10 de Julho de 2015.

[27] CAPPIELLO, C.; PERNICI, B.; PLEBANI P. **Quality-agnostic or quality-aware semantic service descriptions?**. W3C Workshop on Frameworks for Semantics in Web Services, 2005.

[28] **The MongoDB 3.0 Manual**. MongoDB, Inc. Disponível em: https://docs.mongodb.org/manual. Acesso em: 11 de Julho de 2015.

[29] CHWIF, L.; MEDINA, A. C. **Modelagem e Simulação de Sistemas a Eventos Discretos**. Prentice Hall, 2007.

[30] VESDAPUNT, N.; GARCIA-MOLINA, H. **Identifying Users in Social Networks with Limited Information**. Stanford University, 2014.