Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Auto-Tuning de banco de dados NoSQL com dados de Internet das Coisas: um estudo de caso com o Cassandra

Lucas Benevides Dias

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientadora
Profa. Dra. Maristela Terto de Holanda

Brasília
2018

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Auto-Tuning de banco de dados NoSQL com dados de Internet das Coisas: um estudo de caso com o Cassandra

Lucas Benevides Dias

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Profa. Dra. Maristela Terto de Holanda (Orientadora)
CIC/UnB

Profa. Dra. Aletéia Patrícia Favacho de Araújo
CIC/UnB

Prof. Dr. Angelo Roncalli Alencar Brayner
UFC

Prof. Dr. Bruno Luiggi Macchiavello Espinoz
Coordenador do Programa de Pós-graduação em Informática

Brasília, 21 de junho de 2018

Dedicatória

Dedico este trabalho a meus filhos Sofia e Bento.

Que eles tenham amor à Ciência!

Agradecimentos

Ao Instituto de Pesquisa Econômica Aplicada (Ipea), instituição altaneira da qual faço parte, que me concedeu licença para cursar o mestrado que resultou neste trabalho.

A toda a equipe do Laboratório de Tecnologias da Tomada de Decisão (Latitude), da Faculdade de Tecnologia da Universidade de Brasília, na pessoa de seu coordenador, Prof. Dr. Rafael Timóteo de Sousa Júnior, que me permitiu, mesmo não sendo aluno daquele departamento, utilizar máquinas do laboratório para esta pesquisa.

Aos amigos do Centro de Estudos e Atividades Culturais (CEAC) de Brasília, por terem me acolhido nas milhares de horas em que lá estive estudando.

Aos membros da lista de e-mails da comunidade Cassandra, que por vezes sanaram minhas dúvidas sobre detalhes do funcionamento desse SGBD NoSQL.

Ao colega Jefferson Chaves Gomes pela ajuda na programação, em linguagem Java, da customização do Cassandra Stress Tool.

À minha orientadora Maristela Holanda pela paciência e tempo dedicados ao longo de dois anos de trabalho.

À minha esposa Lourdiane por ter cuidado de nossos filhos nos momentos em que estive ausente devido à pesquisa.

Resumo

Os dados provenientes de um ambiente de Internet das Coisas (IoT - *Internet of Things*) podem atingir um volume muito grande, proporcional à quantidade de dados gerados pelos sensores, à sua periodicidade de envio e ao número de dispositivos conectados. Estes dados são séries temporais e possuem características específicas que podem ser exploradas para facilitar seu armazenamento. Há sistemas gerenciadores de bancos de dados que possuem funcionalidades específicas para armazenar estes dados, entre eles está o banco NoSQL Cassandra, o qual provê duas estratégias de compactação, que organizam as páginas de dados de maneira otimizada para dados de séries temporais, como os de IoT. Este trabalho compara as duas estratégias e encontra a mais eficiente quanto ao tempo de resposta e *throughput*. A estratégia de compactação possui parâmetros de configuração, cuja definição fica a cargo do usuário. O efeito destes parâmetros no desempenho do sistema é estudado e pontos ótimos de configuração são definidos, por meio de testes e análises de resultados. Um mecanismo de *auto-tuning* chamado C*DynaConf foi desenvolvido, baseado nos pontos ótimos de configuração preestabelecidos. Os resultados apontaram que seu uso trouxe melhoria média de 4,52% no número de operações realizadas, quando comparado a um cenário de IoT que se inicia com configuração ótima, mas passa a ter suas características alteradas.

Palavras-chave: Banco de dados, *NoSQL*, Internet das Coisas, séries temporais, auto-tuning, Cassandra, estratégias de compactação

Abstract

Data provided by Internet of Things (IoT) may achieve very great volumes, proportional to the amount of data generated by sensors, to its periodicity and to the number of devices connected. This data, that is a case of time series, has some specific characteristics that can be used to favour its storage. There are some database management systems that provide mechanisms that are proper to time series data storage, between those is the NoSQL Cassandra database. These mechanisms in Cassandra are two compaction strategies, that organize data pages in an optimized way, in order to take advantage of IoT Data peculiarities. This work compares the two compaction strategies and finds the mos efficient in terms of response time and throughput. The compaction strategy has parameters that supply the configuration to the users. The effect of these parameters in performance is studied and optimal configuration points are defined, through tests and results analysis. An auto-tuning engine, called C*DynaConf was developed, based on the optimal configuration points obtained. The results show that the use of the engine has brought a 4.52% mean gain in terms of operations performed, when compared to an IoT test case where the initial configuration is optimal but the scenario's characteristics are variated.

Keywords: Database, Internet of Things, time series, NoSQL, auto-tuning, self-tuning, Cassandra, compaction strategies

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

C*DynaConf Cassandra Dynamic Configurator.

CQL *Cassandra Query Language.*

DTCS *Date Tiered Compaction Strategy.*

IoT *Internet of Things.*

IoTBDS *Internet of Things, Big Data and Security.*

JMX *Java Management Extensions.*

JSON *JavaScript Object Notation.*

LCS *Leveled Compaction Strategy.*

NoSQL *Not only SQL.*

SGBD Sistema Gerenciador de Banco de Dados.

SGBDR Sistema Gerenciador de Bancos de Dados Relacional.

SQL *Structured Query Language.*

SSTables *Sorted String Tables.*

STCS *Size Tiered Compaction Strategy.*

TTL *Time to Live.*

TWCS *Time Window Compaction Strategy.*

XML *Extensible Markup Language.*

YAML *YAML Ain't Markup Language.*

Capítulo 1

Introdução

Com o avanço das tecnologias de sistemas embarcados, microeletrônica, telecomunicações e sensoriamento, alguns dispositivos eletrônicos deixaram de ser apenas meios para que as pessoas acessem a Internet, e passaram a ser agentes ativos, que conectam-se à Internet de forma autônoma, emitindo sinais de sensoriamento e recebendo sinais de controle, interagindo com outros dipositivos, com ou sem intervenção humana (Santos *et al.* , 2016). A infraestrutura de rede que provê esse paradigma de interação entre dispositivos é chamada Internet das Coisas, em inglês, *Internet of Things (IoT)* (Xu *et al.* , 2014).

O número de dispositivos conectados à Internet superou o número de pessoas conectadas à rede mundial em 2011, quando já havia 9 bilhões de dispositivos, e esse número deve chegar a 24 bilhões em 2020 (Gubbi *et al.* , 2013). O número aumenta porque cresce também a gama de dispositivos que se conecta à Internet, como por exemplo telefones celulares, veículos de transporte (carros, ônibus, trens), maquinário industrial e até mesmo roupas e acessórios como os relógios e pulseiras inteligentes (Atzori *et al.* , 2010). Dentro das casas também cresce o número de eletrodomésticos e eletrônicos que transmitem dados à Internet. Todos estes sensores, em última análise, proverão dados com características de *Big Data*, tais como: grandes Volume, Velocidade e Variedade (Laney, 2001). Esta relação entre dados IoT e *Big Data* foi apresentada por Ramaswamy *et al.* (2013), Zaslavsky *et al.* (2013), Zhang *et al.* (2013). Esta grande massa de dados gerada em um ambiente IoT precisa ser analisada, compreendida e interpretada para que tenha valor (Perera *et al.* , 2014).

Os dados precisam ser armazenados para que possam ser analisados. Não é possível nem desejável armazenar localmente, nos sensores, grandes quantidades de dados, porque os sensores geralmente são dispositivos leves, com pouco poder de processamento e pouco espaço de armazenamento (Ma *et al.* , 2013). Além disso, manter os dados nos próprios dispositivos seria inadequado, pois cada vez que um usuário precisasse de dados de uma área, teria que percorrer diversos sensores para coletar os dados (Abu-Elkheir *et al.* ,

2013).

Sistemas de Bancos de dados *Not only SQL* (NoSQL) são adequados para armazenar dados com características *Big Data*, pois oferecem uma solução flexível, com escalabilidade horizontal, para armazenar dados estruturados, semiestruturados e não estruturados (Bhogal & Choksi, 2015, Srivastava *et al.* , 2015). Com o aumento do número de fontes de dados com essas características, sistemas NoSQL seguem evoluindo com o intuito de prover mecanismos mais eficientes e eficazes para armazenar e gerenciar esses dados. Por isso, são muitas vezes escolhidos para armazenar dados de IoT (Oussous *et al.* , 2015).

O sistema de banco de dados é algo crítico em uma arquitetura IoT, pois, caso não acompanhe a vazão aos dados recebidos, tornar-se-á um gargalo (Cecchinel *et al.* , 2014). Além disso, se o banco de dados ou componente de armazenamento estiver indisponível, os dados poderão ser perdidos, pois nem chegarão a ser armazenados. Isso ressalta a importância de serem usados sistemas de bancos de dados com tolerância a falhas.

O uso de sistemas gerenciadores de bancos de dados distribuídos pode ajudar nessa tarefa, haja vista que eles têm subsistemas replicados que eliminam, ou pelo menos diminuem, os pontos únicos de falha (Öszu & Valduriez, 2001). Além disso, podem prover balanceamento de carga entre seus nós com o intuito de prover melhor desempenho.

Dentre os vários bancos NoSQL existentes, foi escolhido o Apache Cassandra para este trabalho, motivado por algumas de suas características propícias ao armazenamento de dados IoT, dentre elas:

(i) É um sistema projetado para suportar uma grande taxa de inserção de dados e prover alta disponibilidade (Lakshman & Malik, 2010).

(ii) "O Apache Cassandra é um notável banco de dados NoSQL e seu modelo de dados é muito adequado para lidar com dados sequenciais, não importando o tipo e o tamanho dos dados" (tradução livre) (Lu & Xiaohui, 2016).

(iii) Possui duas estratégias de compactação de dados que buscam tirar proveito das características de dados como os de IoT, para prover armazenamento e gerenciamento mais eficiente (Datastax, 2018).

(iv) Possui configuração que limita o tempo de vida do dado, em inglês, o *Time to Live* (TTL) (Carpenter & Hewitt, 2016).

Além das características listadas, o sistema Cassandra é um dos mais populares bancos NoSQL (DB-Engines, 2018b), o que aumenta a aplicabilidade da pesquisa, e possui código-fonte aberto, o que permite customizações para a realização deste trabalho.

Os bancos NoSQL, derivados do seminal SGBD Google BigTable, devem ter suas páginas de dados periodicamente compactadas, para o funcionamento eficiente do sistema

(Chang *et al.* , 2008). O Cassandra é um SGBD que tem essa funcionalidade de compactação derivada do BigTable (Lakshman & Malik, 2010). A operação chamada, originalmente, de *major compaction*, não envolve algoritmos de compactação[1] ou compressão de dados propriamente ditas. Consiste em um processo que lê as páginas de dados em disco e realiza uma fusão, por meio de algoritmo *merge-sort*, das páginas de dados, resultando em uma nova página (Ghosh *et al.* , 2015). Há diferentes estratégias de compactação, que definem quais páginas devem ser fundidas e em quais momentos (Lu & Xiaohui, 2016).

Neste contexto, a correta aplicação das estratégias de compactação tem como objetivo melhorar o desempenho do banco de dados, diminuindo o tempo de resposta das operações de leitura e de escrita. Em suma, este objetivo será atingido por uma melhor organização dos dados em disco, fazendo com que estes sejam escritos e lidos de maneira sequencial, o que é mais eficiente do que a leitura e a escrita de maneira fragmentada (Kona, 2016). Outra função das estratégias de compactação é a de desalocar o espaço reservado a dados excluídos, uma vez que a deleção de células possui suas especificidades em bancos NoSQL como o Cassandra (Hegerfors, 2014).

A configuração dos parâmetros que definem o comportamento das estratégias de compactação do Cassandra é feita manualmente. Assim, imagina-se que uma configuração automática pode trazer melhorias de desempenho e de facilidade de uso, fatores que motivaram este trabalho.

1.1 Definição do Problema

O Sistema Gerenciador de Banco de Dados (SGBD) NoSQL Cassandra tem configurações estáticas e definidas por usuário, das estratégias de compactação para as suas tabelas. Em um ambiente IoT, que usualmente é dinâmico, isto é, em que as características de inserção e de leitura variam, uma configuração inadequada pode levar a um mau desempenho em relação ao tempo de resposta ao usuário e ao *throughput* (número de operações por segundo) do banco NoSQL.

É muito trabalhoso para o usuário administrar as estratégias de compactação do Cassandra. Além disso, é muito arriscado, pois não se sabe de antemão quais valores de parâmetros trarão uma configuração mais eficiente.

O que se pretende neste trabalho é automatizar a configuração – o que se chama de *auto-tuning* – da estratégia de compactação do Cassandra mais propícia para armazenamento de dados de IoT, tarefa hoje a cargo dos usuários.

[1]Este termo será usado, embora não se refira à compactação de dados, pois é utilizado na literatura.

1.1.1 Hipótese

É possível criar um mecanismo de *auto-tuning* de estratégia de compactação do Cassandra, que funcione de maneira dinâmica e obtenha desempenho mais eficiente do que uma configuração estática, ainda que para um cenário inicial, a configuração estática esteja em seu ponto ótimo de configuração.

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho objetiva desenvolver um mecanismo, chamado Cassandra Dynamic Configurator (C*DynaConf)[2], que faça a configuração dinâmica (em tempo de execução) e autônoma (sem intervenção manual) dos parâmetros da estratégia de compactação do SGBD NoSQL Cassandra, aplicado a dados de Internet das Coisas, com o intuito de maximizar o *throughput*, minimizando o tempo de resposta. O C*DynaConf será capaz de alterar as configurações do sistema quando houver mudanças na proporção entre operações de leitura e operações de escrita, e quando houver mudanças no tempo de vida do dado.

1.2.2 Objetivos Específicos

Para alcançar o objetivo geral desta proposta, os seguintes objetivos específicos foram definidos:

(i) criar mecanismo que extraia informações acerca da razão entre operações de leitura e escrita de dados de *Internet of Things*, para subsidiar a estratégia de *auto-tuning* do SGBD Cassandra;

(ii) avaliar os efeitos da variação dos parâmetros de configuração da estratégia de compactação no tempo de resposta e no *throughput*, decorrentes de alterações no tempo de vida do dado, isto é, após quanto tempo eles se expiram;

(iii) avaliar o espaço de armazenamento em disco utilizado pelo banco de dados, de acordo com a variação dos parâmetros de configuração da estratégia de compactação;

(iv) analisar resultados de execuções de carga e consulta de dados IoT, com diferentes configurações de estratégias de compactação, para obter pontos ótimos que serão usados no mecanismo de configuração dinâmica;

[2]Os caracteres C* são uma abreviatura de Cassandra, adotada por sua comunidade.

(v) criar mecanismo que configure de maneira autônoma, baseado em regras previamente estabelecidas, os parâmetros de configuração da estratégia de compactação do Cassandra mais adequada aos dados de IoT.

1.3 Estrutura do Documento

Este documento está dividido nos seguintes capítulos:

- o Capítulo 2 traz o referencial teórico necessário para o desenvolvimento desta pesquisa, tais como características dos dados IoT, de séries temporais, de bancos de dados NoSQL, entre outros;
- o Capítulo 3 exibe o ambiente de execução, suas configurações, ferramentas utilizadas e detalhamento dos casos de teste;
- o Capítulo 4 apresenta a fase de aprendizagem do processo de compactação, na qual foram obtidos pontos ótimos de configuração. Os resultados alcançados são analisados e servem como base para a construção do mecanismo de *auto-tuning*;
- o Capítulo 5 apresenta o funcionamento do mecanismo de *auto-tuning* C*DynaConf e analisa seus resultados;
- o Capítulo 6 traz a conclusão e os trabalhos futuros.

Capítulo 2

Referencial Teórico

Este capítulo apresenta o conteúdo teórico necessário para o desenvolvimento deste trabalho. A Seção 2.1 apresenta histórico e definição sobre a Internet das Coisas. Na Seção 2.2 são listadas características dos dados provenientes de IoT. A Seção 2.3 traz atributos dos bancos de dados NoSQL e uma classificação quanto ao seu tipo de armazenamento. A Seção 2.4 apresenta características do Cassandra importantes para o entendimento do trabalho, como o funcionamento das operações de leitura e escrita, e as estratégias de compactação. A Seção 2.5 mostra conceitos teóricos sobre configuração dinâmica, ou *auto-tuning*. Por fim, a Seção 2.6 traz trabalhos relacionados ao problema de pesquisa.

2.1 Internet das Coisas

O termo *Internet of Things* (IoT) foi cunhado pela primeira vez por Kevin Ashton em 1999 no contexto de gerenciamento de cadeia de suprimentos, entretanto, na década de 2000 o termo tornou-se mais inclusivo e passou a abranger uma gama maior de aplicações, tais quais: cuidados com a saúde, utilitários do lar, acessórios, meios de transporte, dentre outros (Perera *et al.* , 2014).

Assim, Internet das Coisas foi definida, em tradução livre, como:

> Interconexão de dispositivos de sensoriamento e de acionamento, que provê a habilidade de compartilhar informações sobre plataformas através de um *framework* unificado, desenvolvendo uma imagem de operação comum para possibilitar aplicações inovadoras. Isto é alcançado pelo sensoriamento ubíquo e contínuo, análise de dados e representação de informações uniformizada [...] (Gubbi *et al.* , 2013).

O objetivo principal de IoT é prover interconexão entre objetos e pessoas, permitindo respostas inteligentes, autônomas a determinadas situações, ou produzindo informações para auxiliar na tomada de decisões por parte do usuário, baseada em dados precisos e em processamento inteligente de dados. Por isso, IoT é um domínio emergente da Computação, na qual os dados estão se acumulando em alta velocidade e em grande volume (Ma *et al.* , 2013).

2.2 Características de Dados em Internet das Coisas

Os dados provenientes de sensores podem ser classificados como séries temporais, pois são "medidas tomadas ao longo do tempo, de forma sequencial" (Ramesh *et al.* , 2016). Os dados podem ser enviados em intervalos pré-determinados ou eventualmente. No primeiro caso, a cada determinado intervalo temporal o sensor realiza uma medida e envia os dados. Por outro lado, os sensores podem enviar dados somente quando algum evento ocorre, como por exemplo, quando o usuário ligar ou desligar a televisão. Em ambos os casos o sinal enviado geralmente vem com um selo temporal[1], informando o momento da aferição. Ainda que esse selo temporal não seja enviado, o SGBD pode criar esse selo no momento de sua inserção e armazená-lo junto aos dados do sinal, o que só deve ocorrer quando realmente não estiver disponível o dado gerado pelo próprio sensor, que é mais confiável.

Dados IoT, como um caso peculiar de séries temporais, possuem características próprias que podem ser aproveitadas pelos SGBD para aperfeiçoar seu gerenciamento. Com o uso de mecanismos e funcionalidades específicas, pode-se obter maior eficiência nas operações de leitura e escrita, bem como reduzir o volume de espaço necessário. As seguintes características são as mais relevantes para este trabalho (Dias *et al.* , 2018):

(i) escala massiva – Ocorre quando há grande número de dispositivos e quando há existência de dispositivos que, em uma única observação, podem enviar pacotes de dados com volumes grandes, tais quais *scanners*, câmeras e microfones (Li *et al.* , 2012);

(ii) dados ordenados - Dados gerados por sensores geralmente são enviados aos sistemas que os armazenam, na ordem em que foram feitas as observações, e são inseridos nos bancos de dados na mesma ordem em que foram gerados. Pode haver chegada de dados desordenados, por problemas de roteamento de rede ou problemas de sincronização dos relógios dos sensores, mas via de regra esses casos são pontuais e limitados a uma janela de tempo específica (Waddington & Lin, 2016);

[1]Tradução livre do termo *timestamp*.

(iii) recuperação de dados por chave temporal - É muito comum que o usuário, ao recuperar o dado, busque os dados por algum campo relacionado ao momento de sua observação, seja por intervalos ou por valores exatos (Abu-Elkheir *et al.* , 2013);

(iv) dados raramente mudam - Na maioria dos casos, dados de IoT não mudam. Contudo, os dados podem ser sobrescritos ou apagados se for detectado algum erro de inacurácia em um conjunto de sensores (Ma *et al.* , 2013). O controle de concorrência na escrita dos dados não é algo crítico, pois somente em ocasiões excepcionais haverá mudança em algum dado específico;

(v) dados expiram - O ciclo de vida de dados de IoT depende de cada aplicação, mas em alguns casos, dados antigos podem perder sua utilidade e podem ser arquivados, agregados ou até mesmo excluídos, pois sua consulta é algo pouco provável. Até mesmo em aplicações mais persistentes, que armazenam dados por longos períodos, os dados novos são mais frequentemente consultados do que dados antigos (Abu-Elkheir *et al.* , 2013).

A utilidade principal de um ambiente IoT é subsidiar o processo de resposta e tomada de decisão baseadas no sensoriamento de um ambiente, de maneira inteligente (Ma *et al.* , 2013). Portanto, o uso de sistemas de bancos de dados com alta disponibilidade, como o Apache Cassandra, usado neste trabalho, é recomendado.

2.3 Bancos de Dados NoSQL

O termo *Not only SQL (NoSQL)* foi criado em 1998 para descrever um banco de dados relacional que não possuía suporte à linguagem SQL. Tempos depois, o termo foi reaproveitado e usado em apresentações de pesquisadores de bancos de dados não relacionais, que organizaram um encontro informal em San Francisco, Califórnia, no ano de 2009, com apresentações dos fabricantes dos bancos Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, e MongoDB (Fowler, 2012, Strauch & Kriha, 2011).

Os bancos de dados relacionais, em geral, não são os melhores para armazenar os dados chamados *Big Data*, em especial os gerados por aplicações IoT (Vongsingthong & Smanchat, 2015, Zhu, 2015). Alguns estudos apontam que a solução para armazenamento de dados nas características explicitadas na Seção 2.2 estão em bancos de dados *NoSQL* (Ma *et al.* , 2013, Zhu, 2015).

Sistemas de bancos de dados *NoSQL* podem ser definidos como "uma classe de banco de dados não relacionais que possuem como características alta disponibilidade, menor tempo de resposta, suporte a paralelismo e flexibilidade de esquema, sendo propícias

ao contexto de IoT" (Oliveira *et al.* , 2015). São mais adequados a armazenarem dados semiestruturados ou não estruturados, embora permitam também armazenar dados estruturados. Geralmente, se valem da escalabilidade horizontal, isto é, permitem a criação de *clusters* com múltiplos nós de *hardware* de baixo custo, aumentando assim a disponibilidade e a vazão de armazenamento e recuperação de dados, sem aumentar, exponencialmente, o custo do hardware (Srivastava *et al.* , 2015). Bancos de dados *NoSQL* não existem para substituir os Sistemas Gerenciadores de Bancos de Dados Relacionais SGBDR, e sim para complementar seu uso, ou seja, as duas tecnologias podem coexistir (Bhogal & Choksi, 2015).

Para descrever como os bancos *NoSQL* funcionam, existe uma categorização de tipos de bancos, classificando-os quanto ao modelo de dados. Neste trabalho é adotada a classificação feita por (Bhogal & Choksi, 2015, Srivastava *et al.* , 2015, Strauch & Kriha, 2011), que subdivide estes bancos em 4 classes: armazenamento chave-valor, orientados a coluna, orientados a documento e bancos de dados de grafos, os quais serão descritos nas próximas seções.

2.3.1 Armazenamento Chave-valor

Estes bancos mapeiam uma chave a um conjunto de valores. As chaves são únicas e atômicas e, usualmente possuem limitação quanto ao tamanho, permitindo-se apenas uma pequena quantidade de *bytes*. Por outro lado, aos valores permite-se tamanhos muito grandes. Para se obter boa performance, comumente estes sistemas só permitem consultas pela chave e não pelo valor.

Bancos chave-valor oferecem o esquema menos rígido, sendo apropriados para dados não estruturados. Geralmente, estes bancos privilegiam alta escalabilidade, pois possuem uma implementação de tabela *hash* com pares chave-valor espalhados ao longo de múltiplos nós em um *cluster* distribuído, permitindo alta performance em leitura e escrita de dados, em detrimento da consistência. Por isso, muitos deles não possuem funcionalidades de consultas *ad-hoc* e de combinação de dados, tais quais cláusulas de junção, chaves estrangeiras e cláusulas de agregação.

Sistemas de armazenamento chave-valor são utilizados por empresas que precisam de grande performance, como Amazon, Twitter e Facebook. Diversos bancos *NoSQL* desta classe foram influenciados pelo DynamoDB, da Amazon, que pode ser considerado o pioneiro no uso dessa classe em *Big Data* (Strauch & Kriha, 2011). Além deste, alguns dos exemplos de bancos *NoSQL* desta classe são: Redis, Scalaris, Azure CosmosDB e Memcached (DB-Engines, 2018a).

2.3.2 Orientados a Coluna

A abordagem de se armazenar e processar dados por colunas ao invés de por linhas (como nos SGBDR) tem sua origem em sistemas de *Data Warehouse* e de *Business Intelligence*, nos quais a consulta é otimizada em detrimento das escritas (inserção, atualização e exclusão) (Stonebraker *et al.* , 2005). Nos referidos sistemas, *containers* logicamente estruturados por colunas (*column-stores*) operam em uma arquitetura de processamento paralelo usada para criar aplicações de alta performance.

Há vários SGBD de mercado que possuem o mecanismo *column-store*, entre eles o Sybase IQ, Vertica, Microsoft SQL Server e Oracle (Sheldon, 2013, Strauch & Kriha, 2011). O armazenamento orientado a colunas torna mais fáceis as operações de agregação e de ordenação por colunas, muito comuns em análise de dados.

Os bancos *NoSQL* orientados a coluna não utilizam somente índices baseados em colunas, o que comprometeria a performance em operações de escrita. Tais bancos armazenam grupos de dados relacionados em conjuntos de colunas, chamadas famílias de colunas, nos quais cada família é associada com uma chave da linha. Os bancos procuram armazenar dados de colunas em espaços contíguos de disco, ao invés de armazená-los por linha como nos bancos relacionais. Os registros podem ter número de colunas diferentes, sem que o banco use espaço vazio para preencher colunas que existem em outros registros, ou seja, valores nulos não desperdiçam espaço. É também permitido que uma coluna tenha colunas aninhadas dentro dela, quando recebem o nome de supercolunas (Bhogal & Choksi, 2015).

Estes bancos permitem uma eficiente taxa de compressão de dados e permitem a recuperação de apenas parte de uma entidade com rapidez. Os dados são armazenados na ordem lexicográfica de suas chaves, para que seus dados sejam armazenados em espaços próximos, em um ou poucos nós de um *cluster*. Os nós do *cluster* são chamados servidores *tablet* e os espaços de alocamento de disco contíguos são chamados *tablets* (Gessert, 2016).

O produto considerado pioneiro no uso de armazenamento orientado a colunas em *Big Data* é o BigTable, da Google. Outros produtos da mesma classe são o Hypertable, Hbase e o Cassandra.

2.3.3 Orientados a Documento

Esta classe de bancos é um tipo especial do banco chave-valor na qual os valores são restritos a documentos, que são dados semiestruturados em formatos *Extensible Markup Language* (XML), *JavaScript Object Notation* (JSON), *Binary JSON* (BSON) ou outros formatos similares. Para ilustrar seu funcionamento, pode-se comparar seu funcionamento com sistemas relacionais, nos quais um documento equivaleria a uma tupla. Cada

documento possui todas as informações acerca daquela tupla. Além dos documentos, há também as coleções, que são agrupamentos lógicos de múltiplos documentos, nos quais cada documento pode ter esquemas diferentes, isto é, número de colunas e tipos de dados diferentes.

Consultas *ad hoc* são suportadas nos bancos desta classe, já que há uma estrutura mínima. Isto significa que os valores podem ser recuperados tanto pela chave quanto pelos valores. Operações de junção, geralmente, não são necessárias já que cada documento possui todas as informações acerca de um determinado registro.

Os bancos orientados a documento são especialmente otimizados para armazenar dados textuais, como e-mails e documentos. Também, por sua natureza, são próprios para serem utilizados quando os dados já estão organizados em alguns dos formatos suportados pelo banco. Exemplos de sistemas gerenciadores de bancos de dados dessa classe são: CouchDB, MongoDB e RavenDB.

2.3.4 Bancos de Dados de Grafos

A teoria dos grafos é um ramo da matemática estudado há mais de 300 anos e formalizada por Leonhard Euler em 1736, embora o termo "Grafo" só tenha surgido no final do século XIX (Wakabayashi, 2007). A teoria dos grafos também é muito estudada pela Ciência da Computação, que analisa inúmeros algoritmos desta área e projeta aplicações para diferentes ramos da ciência.

Os bancos de dados de grafos permitem armazenar entidades, também chamadas nós, que se relacionam entre si por meio de arestas. Eles são mais utilizados em bancos de dados nos quais os relacionamentos tenham uma importância tão grande ou maior que a dos nós e onde os relacionamentos sejam caracterizados por atributos.

Esses bancos são adequados para aplicações que buscam os dados por meio de relacionamentos, como por exemplo, achar amigos em comum em redes sociais ou achar trabalhos científicos que citaram um determinado conjunto de artigos. Esses bancos se utilizam de algoritmos de teoria dos grafos, com soluções ótimas para recuperação de dados conectados mais rápida do que em bancos relacionais.

Embora os bancos relacionais também permitam implementar relacionamentos usando chaves estrangeiras, as operações de junção necessárias para se percorrer um modelo cheio de relacionamentos são custosas, o que impacta negativamente no desempenho de consultas que envolvam muitos relacionamentos. Ao contrário, os bancos de dados de grafos fazem com que a navegação pelos relacionamentos seja de baixo custo. Grande parte disso é possível porque o trabalho de navegação pelos relacionamentos é facilitado pelas estruturas de dados, que já prevêem os relacionamentos desde o momento da inserção (Sa-

dalage & Fowler, 2013). Exemplos de bancos de dados de grafos são o Neo4J, OrientDB, Infogrid, InfiniteGraph e FlockDB.

2.4 Cassandra

Este Sistema Gerenciador de Banco de Dados NoSQL recebeu esse nome devido a um episódio da mitologia grega. A linda humana Cassandra, filha de Príamos de Tróia, um dia foi brincar com seu irmão Heleno, no templo de Apolo, o deus grego da luz e do sol, que acompanhou seu crescimento e apaixonou-se por Cassandra. Em busca do seu amor, Apolo lhe deu o dom de prever o futuro. Daí a razão do nome, os idealizadores do banco Cassandra quiseram associar a propriedade da clarividência ao uso do banco NoSQL de mesmo nome (Carpenter & Hewitt, 2016). Na mitologia, a bela Cassandra não teve muita sorte e, após ser amaldiçoada e perseguida, acabou decapitada.

Por outro lado, o SGBD homônimo, que surgiu em 2007 no Facebook, para resolver problemas internos de mensageria entre os colaboradores (Carpenter & Hewitt, 2016), tornou-se um dos mais populares bancos NoSQL (DB-Engines, 2018b, Philip Chen & Zhang, 2014). Seu mecanismo de banco de dados é baseado no NoSQL Google BigTable e suas características de replicação são inspiradas no sistema de armazenamento distribuído Amazon Dynamo (Krishnan, 2013). Atualmente, o Cassandra é mantido como um projeto da fundação Apache, por sua comunidade, e possui licença Apache de software livre.

Dentre as características do Cassandra, destacam-se as seguintes (Carpenter & Hewitt, 2016):

- Distribuído – Isso significa que o Cassandra é executado em vários computadores diferentes – chamados nós – mas sob o ponto de vista do usuário, aparenta ser um único sistema, independentemente de qual nó o usuário acessa. O conjunto de nós que funciona como um sistema é chamado de *cluster*. Em um ambiente de produção, na maioria dos casos não há vantagem em se usar o Cassandra em um nó único, pois um grande diferencial do Cassandra é a facilidade de se adicionar ou remover nós, de maneira transparente ao usuário. Além disso, na definição de sua estratégia de replicação, o Cassandra aceita dois níveis lógicos de subdivisão, que são *datacenter* e *rack*. Um *datacenter* tem que possuir pelo menos um *rack* e deve ser definido quando o *cluster* for executado sobre múltiplos centros de computação, separados geograficamente. O Cassandra permite também agrupar os nós que estejam fisicamente dispostos em um mesmo armário em *racks*, conforme pode ser visto na Figura 2.1. Desta forma, a configuração dos nós deve refletir a disposição física dos mesmos, para que os mecanismos de replicação atuem de maneira mais customizada. Pode-se, por exemplo, definir que cada célula de uma família de colunas

esteja hospedada em pelo menos dois *datacenters* e, em cada *datacenter*, haja uma cópia do dado em pelo menos um nó de cada *rack*.

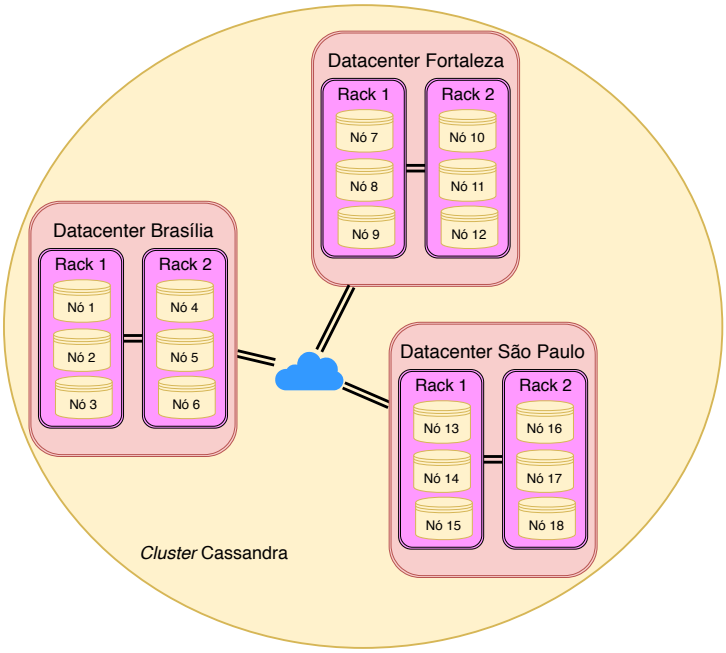

Figura 2.1: Exemplo de Configuração de Topologia em Rede da Estratégia de Replicação.

- Descentralizado –Todos os nós do *cluster* funcionam de maneira igual, ressalvando as diferenças de capacidade de *hardware* como processador e discos, que podem ser diferentes. Ao contrário de outros SGBD distribuídos, não há uma relação de mestre-escravo, portanto, não há ponto único de falha. A cada conexão, um nó do *cluster* serve de coordenador e vai receber e tratar as requisições do cliente. Contudo, dependendo das configurações do *driver* do cliente, a próxima conexão deste poderá ser coordenada por outro nó do mesmo *cluster*, evitando assim a sobrecarga de alguns nós específicos. Por isso, todos os nós atuam como coordenadores em algum momento ao longo do tempo.

- Escalabilidade horizontal – Refere-se à capacidade que um *cluster* Cassandra tem de poder adicionar ou remover nós, de maneira transparente, à medida que a carga de trabalho aumenta ou diminui. A carga de armazenamento (e consequentemente de trabalho) no Cassandra é distribuída pelos *tokens*, uma espécie de vaga para o dado, calculado por algoritmo de *hash* sobre a chave primária. Cada nó recebe um intervalo de *tokens*, que indica quais dados poderão ser armazenados em si. Se os nós possuem o mesmo poder de processamento e armazenamento, é recomendado que todos recebam um intervalo de *tokens* de mesmo tamanho, por outro lado se há algum nó com maior capacidade, ele deve receber mais *tokens*, na proporção

de sua capacidade. Quando implementado com a funcionalidade de virtualização de nós (*vnodes*) este processo de balanceamento de carga é automático e se dá quando um novo nó passa a integrar o *cluster* – um processo chamado *bootstrap* (DataStax, 2018b). A replicação com uso de *vnodes* está ilustrada na Figura 2.2, que representa um Cluster Cassandra com seis nós e um fator de replicaçao igual a três, o que implica que cada dado seja armazenado em três nós diferentes. Nesta ilustração, o *Token* é uma letra do alfabeto, ou seja, sobre cada chave primária é aplicado um algoritmo de *hash* que resulta em uma letra.

- Alta disponibilidade e Tolerância a Falhas – Uma das maneiras mais comuns de se manter a disponibilidade é ter redundância. Este quesito no Cassandra é configurável por meio de um parâmetro, no nível do *keyspace* (o equivalente ao database nos SGBDR), chamado fator de replicação. Este parâmetro diz quantas cópias do dado devem ser armazenadas por nós do *cluster* e funciona em conjunto com o parâmetro de estratégia de replicação. Caso um nó fique indisponível, por meio do protocolo *Gossip*, os demais nós ficam rapidamente cientes da indisponibilidade e passam a responder as requisições feitas referentes ao intervalo de *tokens* do nó indisponível. Neste intervalo, é possível incluir outro nó com o mesmo intervalo de *tokens*, sem causar indisponibilidade. O nó substituto é paulatinamente carregado com os dados que estavam no antigo nó, indisponível. A carga de dados é feita pela rede, recebendo cópias dos dados em que havia réplicas dos dados. Como o parâmetro é configurável, caso o usuário opte por um fator de replicação igual a 1, não há replicação, e caso um nó tenha uma falha, a disponibilidade do *cluster* será comprometida.

- Consistência ajustável – O termo consistência, neste contexto, não refere-se ao mesmo conceito das propriedades ACID de transações em SGBD. Consistência neste contexto diz respeito a garantir que um valor escrito em um nó seja persistido entre seus nós replicados e, do mesmo modo, assegurar que uma operação de leitura em um nó, recupere a versão mais atualizada entre os valores replicados em todos os nós. Essa consistência no Cassandra é configurável, no nível da sessão, seja pelo *prompt* de linha de comando, seja pelo *driver* de conexão. Existem vários níveis de consistência de leitura e de escrita. No menor nível – ONE – o usuário define que a requisição será considerada concluída tão logo um nó encerre a operação. Por outro lado, no nível mais alto – ALL – a requisição só será dada como completa quando todos os nós que possuem réplicas dos dados concluírem suas operações. Desta forma, operações com menor nível de consistência terão menores latências, enquanto que operações com níveis maiores de consistência levarão mais tempo.

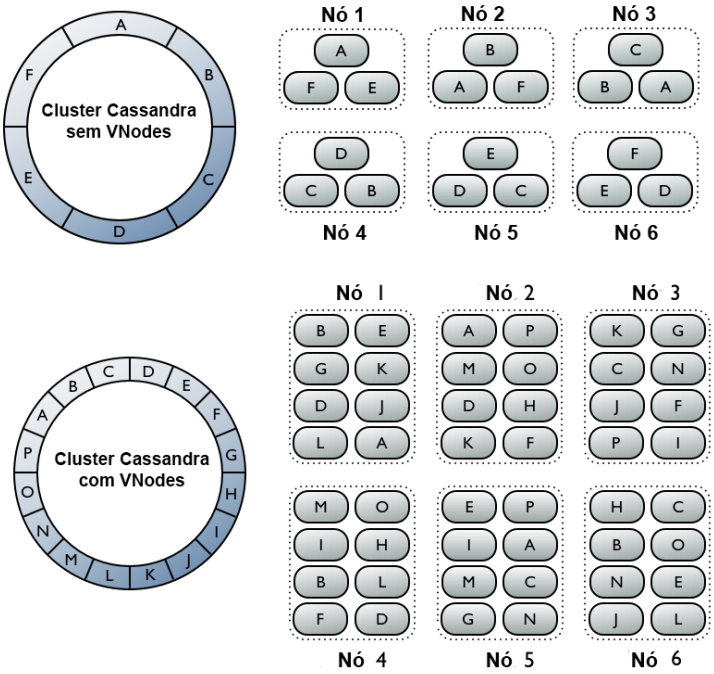

Figura 2.2: *Cluster* Cassandra com Fator de Replicação 3. Fonte: (DataStax, 2018b).

Cabe ao administrador priorizar o nível de consistência de sua aplicação ou o tempo de resposta.

- Esquema livre – O Cassandra indexa todos seus dados pelas linhas, que podem ser comparadas às tuplas no modelo relacional. Mas, diferentemente das tuplas, cada linha no Cassandra pode ter um número variável de colunas. Isto significa que, ao definir uma família de colunas – conceito análogo ao de relação – o usuário não precisa definir toda a estrutura que esta entidade receberá. O modelo é livre para que sejam armazenadas novas colunas, que não foram definidas na criação da tabela. Entretanto, há certas regras, como a definição de uma chave primária, composta pela chave de partição e chave de *clustering*, as quais tem que ser respeitadas após definidas. Uma linha não pode ser recuperada por uma coluna que não esteja na chave primária ou na chave de *clustering*, exceto se um índice for criado sobre aquela coluna.

- Alta Performance – "O Cassandra foi projetado para rodar em *hardware* barato e com alta taxa de escrita sem sacrificar a eficiência da leitura", tradução livre (Lakshman & Malik, 2010). Este SGBD tira proveito do paralelismo e de ambientes com multiprocessadores e com *multicore*, e pode ser escalonado para milhares de nós, o que pode prover alta performance que dificilmente seria atingida em um sistema não distribuído.

Este trabalho trata do *auto-tuning* de estratégias de compactação, logo é importante detalhar os passos dessa funcionalidade do Cassandra. Para entender seu funcionamento, deve-se entender como o Cassandra escreve os dados e como ele efetiva as operações de leitura.

2.4.1 Operações de Escrita, Leitura e Deleção

Muitos bancos de dados NoSQL, a exemplo do Google BigTable, Cassandra, HBase, RocksDB e CouchDB, armazenam seus dados temporariamente na memória (Ghosh *et al.* , 2015), uma vez que o acesso randômico de dados é de 50.000 a 200.000 vezes mais rápido que em disco, e de 250 a 1.000 vezes em relação à memória *flash* (Patterson & Hennessy, 2013). Por outro lado, para acesso sequencial de dados, isto é, aquele em que os dados são lidos de maneira contígua pelo *hardware*, a diferença entre memória e disco é bem menor. Para leitura sequencial de uma grande quantidade de dados – 4GB – a memória RAM é de 7 a 8,5 vezes mais rápida que os discos (Jacobs, 2009). Por isso, os bancos utilizados em *Big Data*, geralmente, privilegiam o acesso em memória e, quando precisam acessar o disco, privilegiam o acesso sequencial de grandes porções de dados.

Os sistemas de bancos de dados NoSQL derivados do BigTable (Cassandra, HBase e RocksDB, entre outros) utilizam uma estrutura de armazenamento na qual os dados ficam armazenados em memória, em estruturas chamadas *Memtables*, e são descarregados[2] para o disco quando (Ghosh *et al.* , 2015):

(i) as *Memtables* atingem seu tamanho máximo;

(ii) as *Memtables* atingem sua idade máxima (diferença entre o tempo atual e tempo em que foram criadas); ou

(iii) quando o usuário comanda, em cada nó do *cluster*, por meio do comando `nodetool flush`.

Não se pode dizer que o Cassandra é um *In-Memory Database*, pois ao serem inseridos, os dados vão também para um espaço no disco chamado *Commit Log*, que funciona apenas para recuperação de falha do sistema. A operação de escrita pode ser vista na Figura 2.3, que mostra os dados oriundos do sensor sendo inseridos na Memtable e no Commit Log.

Uma vez que esses dados vão para o disco, as estruturas que os recebem, de maneira persistente, são chamadas *Sorted String Tables* (SSTables)(Chang *et al.* , 2008). No Cassandra, cada *SSTable* é materializada como um arquivo, no Sistema Operacional. Elas não aceitam alterações nem deleções e sua estrutura consiste em *arrays* ordenados

[2]Tradução nossa do termo *flushed*.

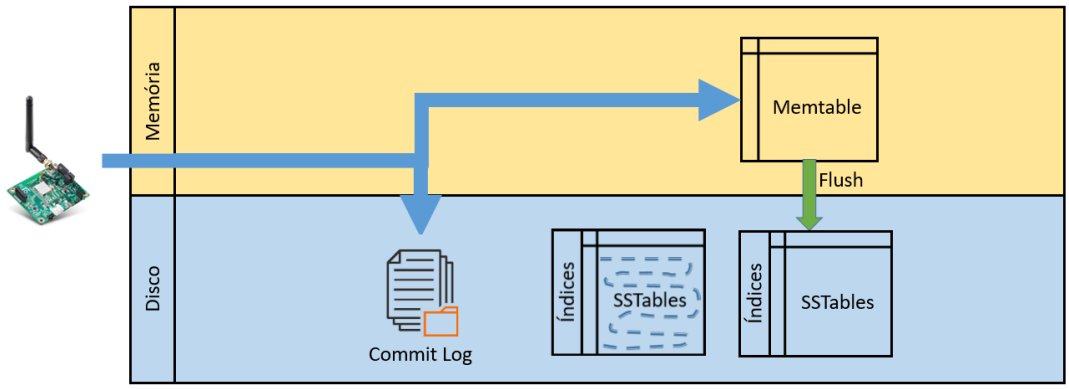

Figura 2.3: Funcionamento da operação de escrita no Cassandra.

contendo chave e valores (Hegerfors, 2014). A chave de busca e de ordenação dos *arrays* das *SSTables* é chamada *clustering key*.

Para facilitar o acesso aos dados, guarda-se um índice para cada *SSTable*, os quais são chamados *per-SSTable*, que indexa os dados dentro de cada tabela. Desta forma, o problema não é saber onde os dados estão dentro de cada *SSTable*, e sim saber em qual *SSTable* o dado está, uma vez que pode haver diversas *SSTables* para uma mesma tabela lógica, se várias *Memtables* foram descarregadas (Eriksson, 2014). Para ajudar a resolver esse problema, guardam-se metadados sobre as *SSTables*, tais quais: valores máximo e mínimo de *clustering keys*, e valores mínimo e máximo de selo temporal, de cada *SSTable*. O uso desses metadados tem o intuito de filtrar quais *SSTables* serão pesquisadas em um momento de busca dos dados, de modo que somente sejam lidas *SSTables* cujos intervalos de *clustering keys* e selo temporal sejam compatíveis com os valores procurados.

Se eventualmente um registro de uma *SSTable* recebe alteração, o novo valor é armazenado em uma *Memtable* e o valor antigo, gravado na *SSTable*, permanece como estava, pois ela é imutável. Se ocorrer um comando de leitura sobre dados alterados, o banco faz a fusão dos dados da *Memtable* com os dados da *SSTable* antes de exibi-los ao usuário (Eriksson, 2014). No caso da *Memtable* com os dados de um determinado registro serem descarregados em disco, haverá dados de um mesmo registro (valores associados a uma mesma chave) em duas *SSTables*, o que gerará um custo de acesso a duas *SSTables* no momento da leitura. O cenário piora se forem múltiplas alterações em um mesmo registro (Chang *et al.* , 2008).

Do mesmo modo da alteração, a deleção também não altera as *Memtables* ou *SSTables* já gravadas. Ocorre uma deleção lógica, isto é, é acrescido um marcador nulo junto à célula excluída, em uma nova *Memtable*. A célula ou linha excluída recebe o nome de *tombstone*. Quando houver uma operação de leitura, o sistema terá que recuperar todos os valores

gravados, identificar qual foi o mais recente, e exibir os valores recentes, deixando de exibi-los caso tenha sido deletado. Se houver várias operações de deleção e de inserção intercaladas sobre a mesma linha, da mesma família de colunas, a leitura desses dados será muito custosa (Carpenter & Hewitt, 2016).

Todos os dados que são deletados no Cassandra – e em muitos sistemas distribuídos – recebem um parâmetro chamado *grace period*. Este período é o tempo mínimo necessário para que o dado seja de fato excluído.

O *grace period* é necessário porque o Cassandra é um banco de dados distribuído e com tolerância a partição. Imagine-se uma situação em que os dados estão replicados em três nós (fator de replicação igual a 3). Se, no momento da exclusão, um desses nós estiver indisponível, a deleção será efetivada em dois nós. Quando o nó se puser disponível novamente, os outros dois poderiam considerar que o dado que fora excluído foi fruto de uma nova inserção, o que faria com que o dado já deletado voltasse a ser válido. Para evitar essa situação, os dados são apenas marcados como apagados e, após o *grace period*, estão aptos a serem desalocados. Se um nó ficar indisponível por um período maior que o *grace period*, esse nó deve passar por um procedimento de sincronização de dados – chamado *repair* – antes de voltar ao *cluster* – procedimento chamado *bootstrap*. O valor padrão do parâmetro `gc grace seconds` é de 864.000 segundos, ou seja, 10 dias, mas esse não foi o valor utilizado nas execuções deste trabalho, porque senão cada exclusão teria que esperar, no mínimo, 10 dias para ser efetivada.

Entretanto, a exclusão definitiva de um dado deletado não se dá automaticamente quando o *tombstone* atinge seu *grace period*. A desalocação do dado e recuperação do espaço em disco só se dá quando ocorre a compactação das *SSTables*.

Pensando em um ambiente em que não está configurada uma estratégia de compactação, o Cassandra possui um parâmetro em nível de tabela, chamado *tombstone threshold*, cujo valor padrão é de 0,2. Ele indica o percentual máximo de registros de uma SSTable que pode ser alocada para *tombstones* já expirados. Se, no caso do valor padrão, a quantidade de *tombstones* superar 20%, o Cassandra vai automaticamente disparar uma operação especial de compactação de tabela única, que servirá para remover os *tombstones* já expirados (Ellis & Morishita, 2012).

2.4.2 Estratégias de Compactação

Cada alteração em uma célula é tratada como uma inserção de nova versão de um dado, pois as SSTables são imutáveis. Após muitas alterações em uma célula, o banco terá muitas páginas em disco com dados do mesmo registro. Para consolidar essas páginas, o banco periodicamente realiza uma compactação, que é o processo que lê as páginas de dados em disco e realiza uma fusão, por meio de algoritmo *merge-sort*, das páginas de

dados, resultando em uma nova página (Ghosh *et al.* , 2015). Há diferentes estratégias de compactação que definem quais páginas devem ser fundidas e em quais momentos (Lu & Xiaohui, 2016).

Para mitigar o problema de várias versões de um mesmo registro fragmentado por múltiplas *SSTables*, foram criadas estratégias de compactação nos bancos de dados NoSQL. Os objetivos dessas estratégias no Cassandra são (Eriksson, 2014, Ghosh *et al.* , 2015, Hegerfors, 2014):

- Permitir que os bancos NoSQL acessem a mínima quantidade de *SSTables* possível no momento de leitura de dados;
- Ocupar menos espaço em disco, eliminando os dados deletados logicamente (*tombstones*) cujo *grace period* já expirou. Embora o intuito seja ocupar menos espaço, para que ocorra a compactação, pode ser necessário alocar mais espaço temporariamente, uma vez que a operação gerará uma nova tabela e deixará as antigas serem eliminadas pelo *garbage collector*.

O processo de compactação de *SSTables* não envolve a aplicação de algoritmos de compactação de dados, que está fora do escopo deste trabalho, mas envolve a eliminação de *tombstones* e a fusão de *SSTables*, por meio de algoritmos de fusão derivados do *merge-sort*, o que é eficiente porque as *SSTables* são ordenadas pela *clustering key* (Ghosh *et al.* , 2015), conforme pode ser visto na Figura 2.4. A operação de Merge compara os pares chave-valor que possuem a mesma chave e mantém somente os valores mais recentes. Células deletadas são expurgadas.

Por outro lado, as operações de compactação envolvem operações de leitura e de escrita em disco, portanto, não deveriam ser feitas a menos que seu benefício seja maior do que o custo da compactação.

2.4.3 Size Tiered Compaction Strategy (STCS)

A abordagem *Size Tiered Compaction Strategy* (STCS) é o método padrão de compactação de tabelas no Cassandra e existe desde sua criação. A estratégia consiste em agrupar as *SSTables* que ocupem aproximadamente o mesmo espaço em disco, desde que tenha sido atingida a quantidade mínima de tabelas para que haja a operação de compactação (Datastax, 2018).

Na versão 3.0 do Cassandra, o STCS possui 10 parâmetros de configuração, dentre os quais, serão destacados os mais relevantes quanto ao desempenho:

(i) `min_sstable_size` - define o tamanho da *SSTable* no qual ela deva sofrer compactação, juntamente com outras tabelas. O valor padrão é de 50MB. Há margens de tolerância que são definidos nos parâmetros (ii) e (iii);

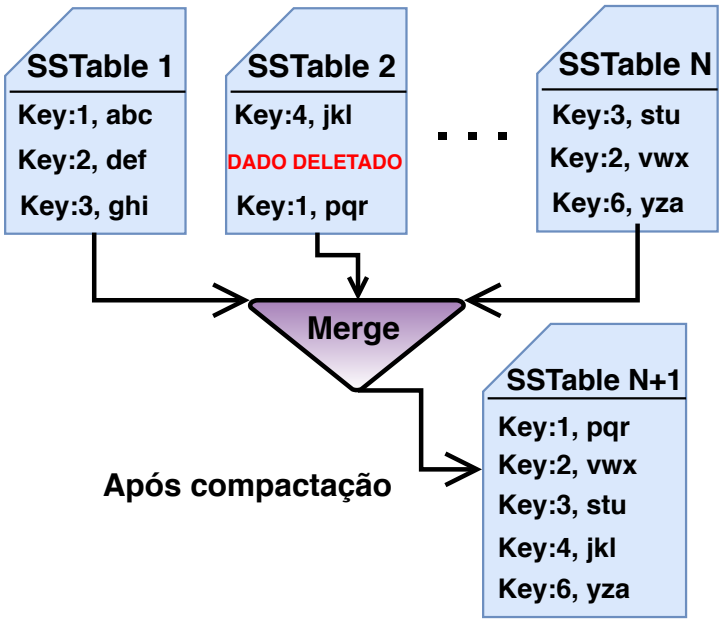

Figura 2.4: Exemplo de Compactação de *SSTables* no Cassandra.

(ii) `bucket_high` - indica o quão maiores que o `min sstable size` as *SSTables* podem ser para que sejam agrupadas em uma mesma *SSTable*. O valor padrão é 1,5;

(iii) `bucket_low` - indica o quão menores que o `min sstable size` as *SSTables* podem ser para que sejam agrupadas em uma mesma *SSTable*. O valor padrão é 0,5;

(iv) `min_threshold` - define o número mínimo de *SSTables* disponíveis em disco, para que possa ser disparada uma operação de compactação. O valor padrão é 4. Este parâmetro é utilizado no componente de *auto-tuning* C*DynaConf;

(v) `max_threshold` - define o número máximo de *SSTables* que podem coexistir em disco sem que haja uma operação de compactação. O valor padrão é 32.

A STCS é recomendada como um caso geral, de escrita intensiva de dados (Lu & Xiaohui, 2016). Entretanto, ela não tira proveito das características específicas de IoT porque ele não agrupa as *SSTables* de acordo com a data em que foram inseridas, ou seja, não há nenhum critério temporal nas operações de compactação. Isto pode acarretar que dados com selos temporais muito diferentes, passem a coexistir dentro de uma mesma *SSTable*. Isto pode misturar dados já expirados, os *tombstones* com dados válidos, o que vai impedir que o sistema faça a desalocação de dados excluídos ou expirados (Hegerfors, 2014).

A Figura 2.5 mostra como as *SSTables* ficam após algumas compactações. Cada linha horizontal representa uma *SSTable*, sua margem esquerda representa o selo temporal em que foi criada, e a margem direita o selo temporal mais avançado no tempo que existe na

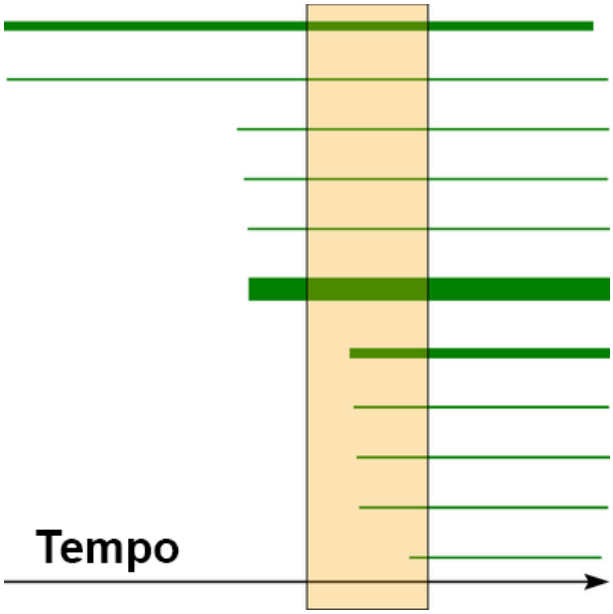

Figura 2.5: Efeito do STCS nas *SSTables*. (Fonte: Hegerfors (2014)).

SSTable. A área do retângulo representa os dados recuperados em uma consulta feita por intervalo de tempo. Nesse caso, a consulta teria que consultar todas as 11 *SSTables*, o que demonstra que não é a melhor escolha para um ambiente que tenha muitas consultas por intervalo de tempo, assim como em um ambiente IoT.

2.4.4 Leveled Compaction Strategy (LCS)

Conforme visto na Subseção 2.4.1, um dos objetivos das operações de compactação é minimizar o número de *SSTables* que são consultadas no momento de recuperar dados. Esta estratégia vai privilegiar este aspecto desejado.

A *Leveled Compaction Strategy* (LCS) possui uma compactação multinível, isto é, as *SSTables* são agrupadas em níveis, com tamanhos 10 vezes maiores que os níveis inferiores. Isto vai fazer com que as *SSTables* mais antigas sejam agrupadas em arquivos grandes. Deste modo somente os dados mais recentes ficam em *SSTables* nos níveis mais baixos (Datastax, 2018).

Com essa estratégia, considerando uma situação em que todas as linhas possuem o mesmo tamanho, há uma probabilidade de que 90% das consultas a uma família de colunas seja resolvida consultando apenas uma *SSTable* (Lu & Xiaohui, 2016). Por outro lado, há mais ocorrência de operações de compactação e estas são mais lentas pois lidam com grandes conjuntos de dados, por isso, esta desvantagem desfavorece o uso dessa estratégia em um ambiente de escrita intensiva.

A estratégia LCS é recomendada, de maneira geral, quando se quer privilegiar operações de leitura em detrimento de operações de escrita. Contudo, ela não possui funcionalidades específicas que tirem proveito das características de dados IoT (Hegerfors, 2014).

2.4.5 Date Tiered Compaction Strategy (DTCS)

Em dados de séries temporais, como os originários de sensores IoT, muito provavelmente não há alteração de dados e os dados são inseridos no banco de dados de maneira sequencial e ordenados pelo selo temporal. Deste modo, cada *SSTable*, geralmente, armazena um espaço contíguo no tempo, diferentemente do que ocorreria se dados fossem inseridos com selos temporais fora de ordem. Hegerfors (2014) propôs a estratégia *Date Tiered Compaction Strategy* (DTCS) e a implementou no Cassandra, pois as outras duas estratégias implementadas até então (STCS e LCS) não tiravam proveito das características dos dados de séries temporais.

A *Date Tiered Compaction Strategy* (DTCS) tem o intuito de agrupar *SSTables* em janelas temporais, baseadas em quanto tempo os dados estão armazenados na *SSTable*, isto é, a idade do dado. A compactação é feita dentro das janelas temporais, para assegurar que *SSTables* de idades contíguas sejam agrupadas, evitando agrupar dados de *SSTables* que foram descarregadas fora de ordem (Eriksson, 2014).

A aplicação da estratégia de compactação envolve a passagem de dez parâmetros ao banco Cassandra durante o comando de criação ou alteração da tabela que armazenará os dados de séries temporais. No entanto, os parâmetros que mais podem alterar o desempenho, são três (Eriksson, 2014, Hegerfors, 2014):

(i) `base_time_seconds`: define o tamanho inicial da janela de tempo, relativa ao instante de inserção dos dados, na qual os dados devem ser agrupados e compactados. Determina a idade mínima na qual uma *SSTable* pode sofrer compactação. Os dados mais novos que `base time seconds` não são compactados mas, segundo seus projetista, isso não deteriora o desempenho pois estes dados serão mais facilmente achados em memória. Após o primeiro agrupamento de dados, o tamanho da janela de tempo cresce exponencialmente em relação ao parâmetro `min threshold`. O valor padrão é uma hora (3600 segundos);

(ii) `min_threshold`: define a quantidade de janelas de tempo de mesmo tamanho (que no primeiro caso são de `base time seconds`) que devem ser agrupadas. O agrupamento ocorre assim que aparecer a janela `min threshold` + 1 de mesmo tamanho. O resultado do agrupamento gera janelas de tempo cada vez maiores, equivalentes ao tamanho anterior, multiplicado pelo `min threshold`. O valor padrão é 4;

(iii) `max_sstable_age_days`: idade na qual os dados não serão mais compactados. Depois de muito tempo, as janelas estarão grandes o suficiente e a compactação deixa de trazer benefícios. Como a operação de compactação é custosa, em especial para grandes janelas de tempo, chegará um ponto em que não valerá mais à pena compactar. O valor padrão é 356 dias.

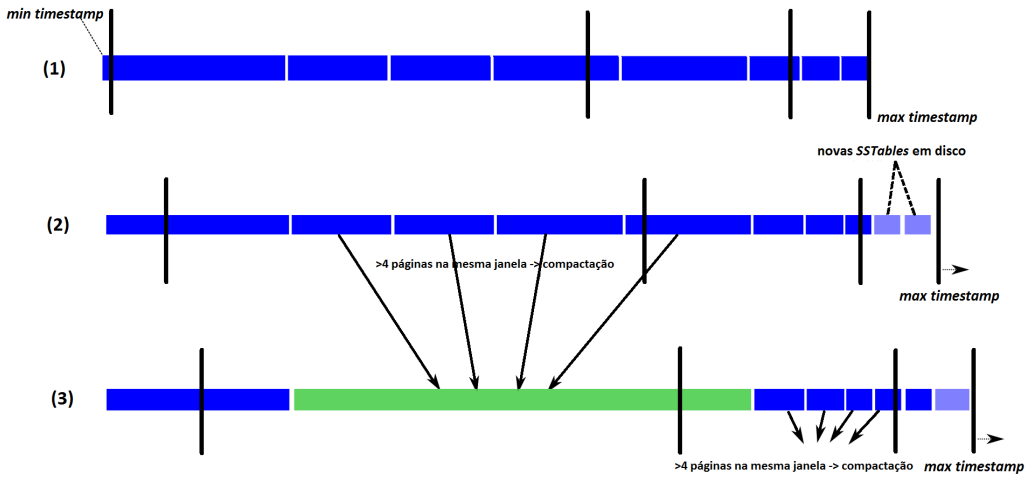

Figura 2.6: Funcionamento do DTCS (Fonte: Eriksson (2014)).

A Figura 2.6 ilustra o funcionamento do DTCS. O eixo horizontal representa o tempo e as três linhas representam a alocação de *SSTables* em disco, em três instantes consecutivos. As linhas verticais delimitam a janela de tempo, cujo intervalo mais à direita corresponde ao valor inicial do parâmetro `base time seconds`. No caso ilustrado, o `min threshold` está definido com o valor padrão 4. Os retângulos azuis representam *SSTables* em disco. Em (1) não há nenhuma janela de tempo com mais de `min threshold` SSTables. No instante (2), há cinco *SSTables* na primeira janela temporal, o que resulta em uma operação de compactação. Na janela mais recente, duas novas *SSTables* são escritas em disco. Em (3) as *SSTables* compactadas em (2) aparecem como um único *SSTable*. Além disso, na segunda janela temporal aparecem mais de 4 *SStables* em um único intervalo, o que implicará em mais uma operação de compactação. À medida que o tempo passa, as *SSTables* mais antigas ficam maiores e as operações de compactação nessas *SSTables* ocorrem menos vezes (Eriksson, 2014).

Por outro lado, a Figura 2.7 ilustra como ficam as *SSTables* após o processo de compactação. O eixo horizontal é o tempo, os retângulos verdes são as *SSTables* e o retângulo longitudinal amarelo é um espaço no tempo consultado pelo usuário. Neste caso, todos os dados estão organizados por seu selo temporal, isto é, não existem duas *SSTables* distintas que contêm dados com o mesmo selo temporal e cada registro tem seus registros

adjacentes quanto ao tempo em sua mesma *SSTable* ou em *SSTable* adjacente (Hegerfors, 2014). Isto vai implicar em uma única operação de leitura, sequencial, em disco, o que diminui o tempo para recuperação de dados.

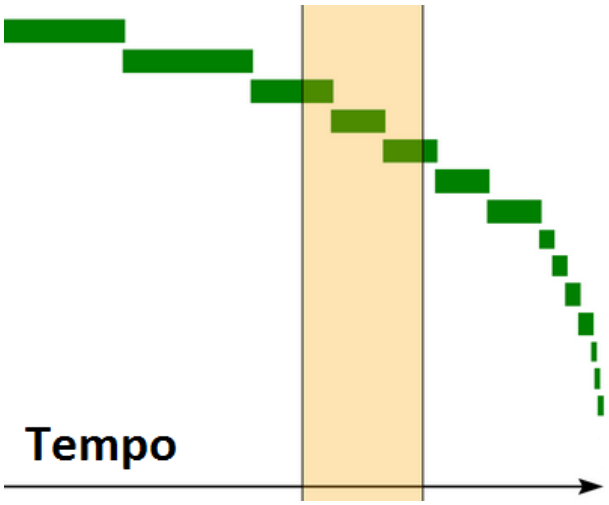

Figura 2.7: Efeito do DTCS nas *SSTables*. (Fonte: Hegerfors (2014)).

A estratégia DTCS é recomendada por Hegerfors (2014) nos casos em que as consultas aos dados sejam feitas de acordo com o selo temporal que marca sua inserção. Além disso, o autor ainda argumenta que a estratégia é vantajosa em conjuntos de dados que são escritos de forma cadenciada, em períodos regulares.

2.4.6 Time Window Compaction Strategy (TWCS)

Até o ano de 2016, o DTCS era a única estratégia de compactação recomendada para armazenamento de séries temporais, assim como num ambiente IoT. Isso mudou quando, em Setembro de 2016, foi lançada uma nova estratégia de compactação, a TWCS na versão 3.8 do Cassandra.

A intenção do TWCS era mitigar alguns pontos fracos do DTCS já anteriormente apontados em Eriksson (2014). Em suma, os problemas identificados no DTCS foram os seguintes (Jirsa & Eriksson, 2016):

(i) a primeira compactação de *SSTables* ocorre sem uma seleção criteriosa quanto ao tamanho. Agrupa as *SSTables* de tamanhos diferentes, apresentando fracos resultados de *throughput* de compactação[3] nas primeiras operações;

(ii) após algum tempo, as tabelas param de ser compactadas, quando atingem o valor definido pelo parâmetro `max_sstable_age_days`. Isso é muito negativo quando

[3]Número de linhas compactadas por segundo

ocorre alguma recuperação ou restauração e dados antigos são inseridos. Nesse caso, dados antigos podem entrar em uma *SSTable* que já atingiu o valor definido e esses dados podem possuir muitas operações de alteração ou deleção, o que precisaria de compactação, que nesse caso nunca ocorrerá;

(iii) o DTCS possui muitos parâmetros, dez no total. Isto torna a estratégia muito complexa para uso e muito difícil de ajustar;

(iv) alguns parâmetros são definidos em dias, outros em segundos, o que pode confundir os usuários.

Para evitar os mesmo problemas, a nova estratégia TWCS tem algumas características diferentes, mas mantém a compactação baseada em uma janela de tempo. O intuito é que os dados que foram inseridos em uma mesma janela de tempo, mantenham-se juntos, em um espaço contíguo, pois geralmente serão recuperados em conjunto. Seguem as diferenças em relação ao DTCS e peculiaridades acerca do TWCS (Jirsa & Eriksson, 2016):

(i) Possui apenas dois parâmetros e os dois referem-se à mesma grandeza. Estes parâmetros são: `compaction window size`, que define o tamanho da janela de tempo em que as *SSTables* serão compactadas e `compaction window unit`, que é a unidade de tempo, podendo ser minutos, horas ou dias.

(ii) A definição da janela de tempo é feita diretamente em unidades de tempo. No caso do DTCS é muito difícil prever em qual tamanho a janela temporal ficará, pois isso dependerá do número de *SSTables* geradas e das configurações de tamanho mínimo da janela, e de números máximo e mínimo de *SSTables* possíveis para compactação. Ademais, na estratégia anterior, pode haver diversos níveis de compactação, em que cada nível possui um tempo maior de janela temporal, que pode crescer de acordo com o crescimento dos dados.

(iii) Uso da *Size Tiered Compaction Strategy* (STCS), presente no Cassandra desde os primórdios, para a primeira compactação. Enquanto os dados não atingem a idade equivalente à janela de tempo, os dados podem ser dispostos em diversas *SSTables* e podem ser compactados. Neste caso, entra em ação a estratégia já consolidada STCS, que basicamente agrupa as *SSTables* de tamanhos equivalentes, o que resulta em melhores *throughputs*. Como é usado o STCS, seus parâmetros são aceitos na configuração do TWCS, mas são aplicados somente à primeira compactação.

Embora o TWCS tenha surgido para substituir o DTCS, este último continua existindo. Um estudo comparativo quanto ao desempenho destas duas abordagens faz parte deste trabalho e está descrito na Seção 4.1.

O TWCS faz a compactação levando em conta as páginas adjacentes. Por isso, é adequado para o armazenamento de dados IoT, em que os dados comumente são inseridos em intervalos contíguos de tempo. Desta forma, se não houver inserção fora de ordem, somente uma única página de dados em disco terá dados de um determinado instante. Isto implica que o banco de dados consulte o mínimo de páginas de dados em disco para buscas por intervalos temporais. O desempenho do banco de dados está relacionado, de maneira inversamente proporcional, ao número de acessos a disco (Chang *et al.* , 2008), isto é, quanto menos o banco acessar o disco, melhor desempenho ele terá.

2.5 *Auto-Tuning*

Em um contexto de *software*, diz-se que um sistema possui funcionalidade *Auto-tuning* ou *self-tuning* quando o programa em execução pode determinar sua própria configuração ótima para uma determinada carga de trabalho e pode ser capaz de se reconfigurar, quando necessário, em resposta às mudanças de condições (Sullivan *et al.* , 2004).

À medida que a complexidade dos sistemas de informação cresce, aumenta também o trabalho dos programadores (Filieri *et al.* , 2014). Sistemas de bancos de dados distribuídos, como o Cassandra, têm maior complexidade que sistemas monolíticos e sua natureza é mais dinâmica, pois estão sujeitos a mudanças repentinas em relação aos seus nós de processamento, que podem tornar-se indisponíveis de maneira inadvertida (Alves & Freitas, 2015).

Pesquisas na área de configuração automática de *software* têm aumentado em número e variedade, com o objetivo de auxiliar os desenvolvedores e usuários a avaliar um conjunto de parâmetros de *software* e escolher aquele que tiver o melhor desempenho (Tiwari *et al.* , 2009).

Dentre as abordagens de configuração autônoma, há uma tipificação que as subdivide em duas (Alves & Freitas, 2015):

(i) configuração Automática Estática é feita de modo autônomo antes da execução do programa. Ocorre, por exemplo, quando um programa executa ações de leitura em arquivos para decidir, sem intervenção manual, quais são os melhores parâmetros para inicialização de outro programa;

(ii) configuração Dinâmica ocorre quando o programa altera, em tempo de execução, os parâmetros que definem sua execução.

Como apresentado anteriormente, o objetivo desta dissertação é gerar um programa de configuração dinâmica, que configure os parâmetros da estratégia de compactação mais propícia ao gerenciamento e armazenamento de dados IoT.

2.6 Trabalhos Relacionados

Na área de banco de dados, o *auto-tuning* vem sendo estudado, pelo menos, desde (Hammer & Chan, 1976), sob o prisma de se obter uma configuração ótima para os índices de um banco de dados. Posteriormente, foi utilizado *auto-tuning* também por meio do uso de particionamento de tabelas e uso de visões materializadas (Oliveira & Lifschitz, 2014).

Desde então houve muitos estudos conduzidos nesta linha, mas poucos se dedicaram à configuração de estratégias de compactação de bancos de dados, pois este é um problema específico de alguns bancos NoSQL.

Dentre as quatro estratégias de compactação existentes no Cassandra, há duas típicas para o armazenamento de dados como os de IoT. A primeira é o *Date Tiered Compaction Strategy* (DTCS), proposta e implementada por (Hegerfors, 2014), cujos detalhes do funcionamento e efeitos do uso da estratégia são mostrados em (Eriksson, 2014). A segunda, mais recente é a *Time Window Compaction Strategy* (TWCS), que foi criada com o intuito de evitar alguns problemas conhecidos no DTCS (Jirsa & Eriksson, 2016).

No trabalho (Sathvik, 2016) a performance do Cassandra foi analisada, mudando-se parâmetros de configuração das *Memtables* e de uma tabela chamada *Key-Cache*, que reside na memória e salva ponteiros para recuperar mais rapidamente os dados nas *SSTables*. Entretanto, o referido estudo não analisa o impacto no desempenho após a alteração dos parâmetros de configuração das estratégias de compactação tampouco estuda cenários de dados de IoT.

No curso da pesquisa de Lu & Xiaohui (2016), foram feitas execuções de cenários de estresse e métricas foram analisadas. O estudo afirma que "O Apache Cassandra é um notável banco de dados NoSQL e seu modelo de dados é muito adequado para lidar com dados sequenciais, não importando o tipo e o tamanho dos dados" (tradução nossa). O mesmo trabalho, conclui que a estratégia DTCS é adequada no uso em séries temporais, resultando em um melhor desempenho nesses casos em comparação a outras estratégias de compactação disponíveis no Cassandra.

Kona (2016) investiga, com o uso do Cassandra, as métricas apropriadas para se avaliar o desempenho das estratégias de compactação. Este trabalho compara – em um *cluster* com um apenas um nó – as estratégias DTCS, STCS e LCS em um ambiente de escrita intensiva, onde 90% das operações são de escrita e 10% de leitura. Ele conclui que a estratégia DTCS não é a mais adequada para o cenário simulado, perdendo em performance para a LCS. Entretanto, no caso analisado os dados não possuem características de séries temporais.

Na mesma linha de pesquisa e ainda trabalhando com um *cluster* de somente um nó, (Ravu, 2016) simulou o comportamento do Cassandra em três situações de carga de trabalho, uma de escrita intensiva, uma de leitura intensiva e uma balanceada, na

qual havia o mesmo número de operações de leitura e de escrita. A estratégia DTCS apresentou métricas mais favoráveis para o cenário de leitura intensiva, ainda que os dados não tenham características de séries temporais. Para o cenário de escrita intensiva, por outro lado, a estratégia mais performática foi a LCS. Na época da publicação desses dois últimos estudos, não havia sido disponibilizada para o Cassandra a estratégia TWCS, objeto de estudo deste trabalho.

Saindo do âmbito do Cassandra, em (Xiong *et al.* , 2017), obteve-se resultados muito positivos com o *auto-tuning* do banco NoSQL HBase, que faz parte da suíte Hadoop. Nesse trabalho foram utilizadas técnicas de inteligência artificial, mais especificamente algoritmos de *ensemble learning*, para obter uma configuração ótima. Foram analisados 23 parâmetros de configuração em cinco cenários distintos nos quais se identificaram os que têm maior impacto na performance. O componente de *auto-tuning* consegue elevar o *throughput* em 41% em média e reduzir a latência em 11%. No entanto, não foi encontrado trabalho semelhante a esse no Cassandra. E nenhum dos cinco cenários analisados possuíam características específicas de IoT.

A Tabela 2.1 apresenta as características relevantes dos trabalhos relacionados. Como pode ser visto, não foi encontrada nenhuma publicação acadêmica que tenha analisado o desempenho da nova estratégia TWCS, comparando-a com o anterior DTCS. Nem tampouco foi encontrada publicação que busque pontos ótimos de configuração dentro da estratégia TWCS. Ademais, embora testes tenham sido feitos para se obter qual a estratégia de compactação mais adequada para diferentes cargas de trabalho usando o Cassandra, não foi feita nenhuma configuração nos parâmetros da estratégia de compactação, nem iniciativa de *auto-tuning* desses parâmetros no Cassandra. O único trabalho que implementa um *auto-tuning* em parâmetros de configuração, o faz em outro banco NoSQL, o Hbase.

Esta dissertação é o único trabalho que aborda o *auto-tuning* do Cassandra, usando as estratégias DTCS e TWCS, com dados que apresentem características IoT.

Tabela 2.1: Trabalhos Relacionados.

Autores	**Estratégias**	**IoT**	**Auto-Tuning**
Hegerfors (2014)	DTCS	X	
Eriksson (2014)	DTCS	X	
Jirsa & Eriksson (2016)	DTCS e TWCS	X	
Sathvik (2016)	STCS		
Lu & Xiaohui (2016)	STCS, LCS e DTCS		
Kona (2016)	STCS, LCS e DTCS		
Ravu (2016)	STCS, LCS e DTCS		
Xiong *et al.* (2017)	N/A		X
Esta dissertação	DTCS e TWCS	X	X

Capítulo 3

Ambiente de Execução

Este capítulo detalha o ambiente de execução que foi montado para o desenvolvimento desta dissertação. Ferramentas e técnicas de teste foram buscadas na literatura bem como foram desenvolvidas em laboratório. O capítulo divide-se em quatro seções: na Seção 3.1 é apresentado o hardware e o software utilizados; na Seção 3.2 é descrito como as características de dados IoT foram simuladas em laboratório; na Seção 3.3 é apresentada a ferramenta de estresse utilizada nos testes e, por fim, na Seção 3.4 são listadas e qualificadas as métricas utilizadas para avaliação da performance do banco de dados.

3.1 Hardware e Software Utilizados

Conforme já detalhado na Seção 2.4, o Cassandra não possui grande vantagem ante outros sistemas se não for usado em um ambiente distribuído. Por isso, foi montado um minilaboratório para realizar toda a pesquisa.

Todos os testes foram executados em um *cluster* com 10 nós, todas máquinas virtuais de igual capacidade. Cada nó integrante do *cluster* possui:

- um core do processador Intel Xeon®;
- discos rígidos magnéticos de 7200 rpm, com 50GB de espaço;
- 3.2GB de memória RAM DDR3;
- interface Ethernet Gigabit.

Além dos 10 nós do *cluster*, foi utilizado ainda um computador extra como ferramenta de estresse, cujo papel era geração de massa de dados e carga desses dados no *cluster* Cassandra. Este nó possui 40GB de espaço em disco, 1 core do Intel Xeon® e 2GB de memória RAM. Essa arquitetura está ilustrada na Figura 3.1.

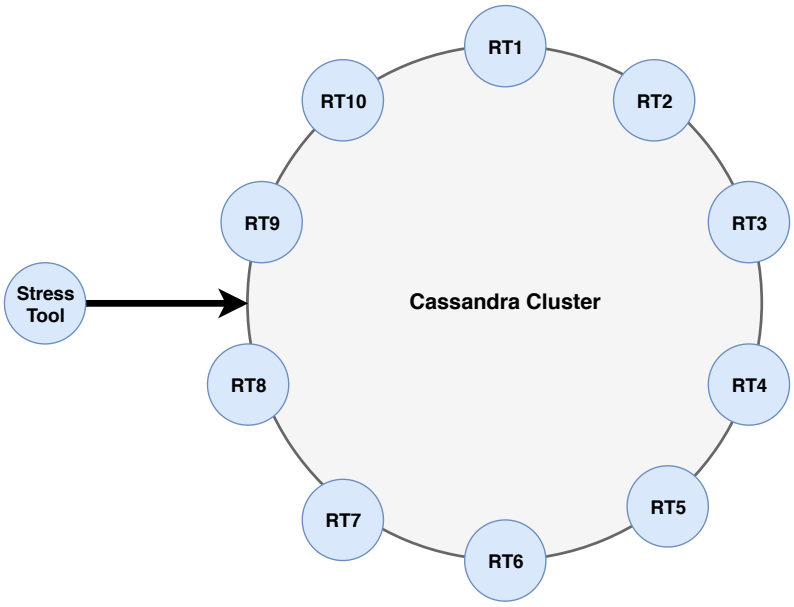

Figura 3.1: Arquitetura Usada para o Desenvolvimento da Pesquisa.

Todos os nós têm como sistema operacional o Linux, distribuição Ubuntu, versão 16.04. A versão do sistema de banco de dados Cassandra instalada no *cluster* foi a 3.11.1, enquanto que a versão do *Cassandra Stress Tool* foi a 4.0, ambos com customizações, que são apresentadas na Seção 3.3.

Quanto à configuração do Cassandra, apenas alguns itens foram alterados em relação ao arquivo de configuração original. Os valores de *timeout* de escrita foram duplicados e o *timeout* de leitura foram quintuplicados. Isto foi necessário pelo alto grau de trabalho e baixo poder de processamento e de I/O do hardware utilizado. Além disso, foram alteradas configurações de parâmetros para permitir a conexão do cliente com os diferentes nós via *Java Management Extensions* (JMX) para monitoramento de métricas, o que não vem originalmente habilitado.

Outra configuração relevante foi a do *keyspace*, que no âmbito do Cassandra é uma divisão lógica dentro do *cluster*, análoga ao conceito de *database*, em bancos de dados relacionais. Um *keyspace* pode ter um conjunto de famílias de colunas, conceito análogo ao de tabelas. Dentro do *keyspace* configura-se a replicação dos dados entre os nós do *cluster*, que envolve a estratégia de replicação e o fator de replicação. A estratégia de replicação usada foi a *SimpleStrategy*, que replica os dados inseridos em cada nó, nos nós subsequentes em sentido horário. A outra estratégia disponível, *Network Topology Strategy*, é indicada para casos em que há mais de um *data center* ou mais de um *rack*, o que não ocorreu em nossos testes.

O fator de replicação usado foi 3, o que significa que todos os dados inseridos em um nó, têm pelo menos duas réplicas dentro do *cluster*. Isto acarreta nas seguintes consequências:

- este fator aumenta o espaço em disco utilizado, multiplicando por 3 o espaço usado pelos dados nos discos do *cluster*;
- aumenta a quantidade de operações de I/O em todo o *cluster*, pois cada dado inserido deverá ser replicado em dois nós;
- pode aumentar diretamente o tempo de resposta de operações de leitura e escrita, a depender do *Consistency Level*, explicado na Figura 2.2, no entanto, o valor padrão de *Consistency Level* utilizado pelo Cassandra Stress Tool é LOCAL_ONE e foi mantida essa configuração, razão pela qual o tempo de resposta não foi tão impactado diretamente, apenas indiretamente pela maior carga de I/O.

Todas essas configurações são definidas no arquivo `cassandra.yaml`. Este arquivo e todas as outras ferramentas estão no respositório *github* desta dissertação que encontra-se em Dias (2018).

É muito importante ressaltar que, durante todas as execuções, não havia um ambiente exclusivo de testes, isto é, os servidores do *cluster* e de estresse estavam em máquinas físicas que executam outros serviços e, portanto, não houve o devido e desejado isolamento. Este problema foi mitigado pois todas as execuções que analisavam um mesmo cenário, foram rodadas em períodos próximos, por um longo período. Alguns cenários foram repetidos em diferentes dias da semana e uma média foi tomada.

3.2 Simulação do Ambiente IoT

Para realizar os testes com dados de Internet das Coisas, optou-se por utilizar dados gerados artificialmente, com características de IoT, os quais chamamos de dados simulados.

Um ambiente IoT é tipicamente um ambiente de escrita intensiva, com menos operações de leitura do que de escrita e possui características específicas, já detalhadas na Seção 2.2.

Uma característica muito comum a dados de IoT é que eles expiram. Para ser fiel a essa peculiaridade, em todos os cenários as execuções contaram com o parâmetro *Time to Live* (TTL) habilitado na família de colunas. Isto significa que todos os dados inseridos na tabela em algum momento expiram. O TTL variou de acordo com os cenários escolhidos, entre uma e três horas. O parâmetro *grace period*, que foi detalhado na Seção 2.2, foi definido para todas as execuções como 1800 segundos, isto é, meia-hora. O valor *default* equivale a 24 horas, mas isso tornaria todos os testes muito demorados, razão pela qual foi reduzido.

Uma tarefa muito importante de qualquer trabalho de ajuste de desempenho é a análise e definição do modelo de dados (Gurry & Corrigan, 1996). Por isso, a modelagem foi feita segundo boas práticas de modelagem de dados presentes em (King & Berglund, 2015).

Por questões de performance, o tipo de modelagem recomendada para o Cassandra é a modelagem orientada a consulta, na qual as famílias de colunas armazenam os dados da maneira que eles serão consultados (Chebotko *et al.* , 2015). Este tipo de modelagem pode gerar alguma redundância, mas é essencial para o bom desempenho de bancos NoSQL, em especial aqueles orientados a coluna, como o Cassandra.

O modelo pode ser visto na Figura 3.2 na notação Chebotko, desenvolvida especificamente para o Cassandra (Chebotko *et al.* , 2015). Nos diagramas, a chave de partição é exibida com a letra K, as *clustering keys* são indicadas com a letra C, seguida de uma seta que exibe a ordem ascendente ou descendente de indexação. As colunas `Static`, que são aquelas armazenadas uma única vez por chave de partição, são indicadas pela letra S.

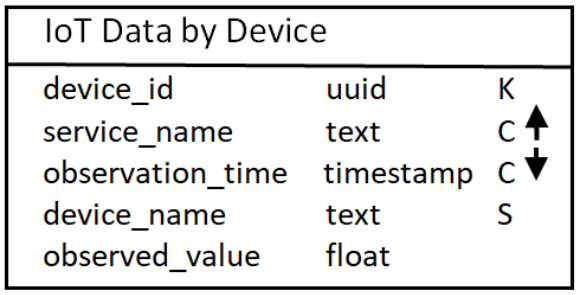

Figura 3.2: Esquema da Família de Colunas.

Para chave de partição foi escolhido um tipo `universal unique identifier`, que identifica univocamente um dispositivo sensor. Para cada dispositivo pode haver diversos serviços de medição, como por exemplo, temperatura, umidade, pressão atmosférica etc. representados no campo `service name`. Em todos os cenários testados neste trabalho, para cada sensor existe um número fixo de cinco serviços, o que implica em cinco séries temporais por cada dispositivo. O `observation time` recebe uma indexação decrescente porque dados mais recentes são mais usualmente consultados e recuperados. Os demais campos recebem o nome do dispositivo e o valor, que é do tipo `float`.

Um ambiente IoT, geralmente, possui mais escrita do que leitura e durantes os testes variamos o percentual de leitura de 1% a 30% do total de operações. Foram definidas três consultas, apresentadas no Código 3.1 na linguagem *Cassandra Query Language* (CQL), que é muito semelhante à linguagem SQL:

- `devdat` – recupera todas os dados emitidos por um determinado dispositivo, passando seu identificador. É responsável por 40% das operações de leitura enviadas

ao banco durante as execuções. Essa consulta possui um limitador para recuperar, no máximo, 500 linhas. Antes de se colocar esse limite, a consulta estava gerando muitos erros de *timeout*;

- `lesrow` – Recupera os dados de uma série temporal específica, passando-se como parâmetro o identificador do dispositivo e o nome da série. É responsável por 30% das consultas e também possui limitador de no máximo 500 linhas;

- `avgdat` – Faz uma consulta muito comum em um ambiente IoT que é a média dos valores observados por uma medida do dispositivo. Retorna apenas uma linha, embora tenha que varrer todos os dados no banco. Esta é responsável por 30% das consultas enviadas ao banco pela ferramenta de estresse.

```
devdat:
        SELECT * FROM iot_data
        WHERE device_id = ? LIMIT 500
lesrow:
        SELECT * FROM iot_data
        WHERE device_id = ? AND service_name = ? LIMIT 500
avgdat:
        SELECT device_id, AVG(observed_value) FROM iot_data
        WHERE device_id = ? AND service_name = ?
```

Código 3.1: Consultas Utilizadas nas Execuções dos Testes.

As operações de INSERT terão seus valores gerados aleatoriamente, o que significa que, se a chave primária se repetir, o que é improvável de acontecer, haverá a geração de um UPSERT, que é como o Cassandra agrupa as operações de INSERT e UPDATE. No Cassandra, se houver uma inserção com os mesmos valores de chave, estes são sobrescritos por quaisquer outros valores que estejam associados a uma mesma chave. Tal comportamento é bem diferente do comportamento em um SGBD relacional, no qual a inserção resultaria em erro, causado por restrição de chave primária. Portanto, é possível que haja, durante uma execução, algumas operações de UPDATE, ainda que sua probabilidade seja muito remota. Não foram definidas operações de DELETE, embora haja exclusão de dados, pois foi definido que os dados terão TTL.

Os conteúdos desta seção, da Seção 4.1 e da Seção 4.2 já foram publicados, de forma resumida, no evento *Internet of Things, Big Data and Security* (IoTBDS) no artigo (Dias *et al.* , 2018).

3.3 Cassandra-Stress Tool

O Cassandra possui uma ferramenta de estresse, mantida pela comunidade, também escrito em Java, chamado *Cassandra-Stress tool*, que possui código-livre. A versão da ferramenta utilizada foi a 4.0, uma versão *alpha*, ainda em desenvolvimento, diferentemente da versão do servidor Cassandra, que foi a versão estável 3.11.1.

Esta ferramenta é capaz de gerar uma massa de testes, com funcionalidades de escolha de distribuição estatística dos dados criados. Ainda pode-se escolher a média e desvio padrão das medidas, de acordo com a distribuição escolhida.

A ferramenta possui diversos modos de uso e o escolhido foi o modo *user*, na qual o usuário entra com as configurações por meio de um arquivo do tipo *YAML Ain't Markup Language* (YAML). Tal arquivo contém o esquema da família de colunas, a distribuição de valores de cada coluna. É nesse ponto que se define que cada dispositivo possui 5 séries temporais e que, a cada operação são inseridos 60 valores para cada série temporal. As consultas explicitadas na Seção 3.2 também são definidas dentro do arquivo. Todos os arquivos de configuração YAML estão presentes em (Dias, 2018).

Para os testes realizados na Seção 4.1, a ferramenta foi usada sem customizações, do jeito que é distribuída pela comunidade. Entretanto, notou-se uma peculiaridade nos dados inseridos. Da maneira como os dados estavam sendo gerados, valores repetidos ocorriam frequentemente para uma mesma chave de partição. Isso significa que dentro de um mesmo dispositivo, valores iguais de tempo de observação, o *timestamp* do dados, estavam sendo inseridos de forma repetida. Isto ocorreu porque os dados aleatórios de `observation time` eram gerados uma única vez e reutilizados a cada iteração no nível da chave de partição.

Conforme já explicitado na Seção 3.2, quando um dado com a mesma chave primária é inserido, o Cassandra não impede a inserção, pelo contrário, ele sobrescreve o valor antigo. Este comportamento foi percebido durante as execuções dos testes descritos na Seção 4.1, mas isso não invalida os resultados pois, como já visto na Subseção 2.4.1, as operações de atualização de dados (UPDATE) são internamente tratados, como operações de inserção e possuem desempenhos semelhantes.

Entretanto, para ser mais fiel a um ambiente de IoT, procurou-se evitar a ocorrência frequente de operações de UPDATE. Para fazê-lo, uma versão diferente do *Cassandra-Stress tool* foi codificada e implementada.

A nova versão alterou a forma como é gerada a massa de dados aleatória. O comportamento indesejado de se repetir o mesmo `observation time` para todas as chaves de partição foi alterado, com uma nova geração de `observation time` aleatórios a cada vez que o programa percorre uma chave de partição, no caso do nosso modelo, o `device id`.

Como tratou-se de uma customização, as alterações não foram submetidas ainda, à comunidade, uma vez que a versão customizada só funciona com modelo de dados semelhante ao usado neste trabalho. O objetivo da customização foi atingido e os dados gerados mantinham a aleatoriedade em todos os níveis. O código-fonte da customização pode ser acessado em (Dias, 2018).

A ferramenta *Cassandra-stress* é *multithread*, isto é, ela permite que diversas *threads* de execução atuem na carga de dados simultaneamente, o que eleva a capacidade de popular dados da ferramenta (Apache, 2016b). O número de *threads* é algo configurável. No funcionamento inicial, o *cassandra-stress* executa a mesma tarefa com um número crescente de *threads* até que haja duas simulações com número maior de *threads*, em que tenha havido perda de desempenho. Na fase inicial dos testes, foram testados alguns valores de *threads*, para operações de leitura e escrita de dados IoT. Em todas as execuções preliminares o valor ótimo em relação ao *throughput* ficou em 24 *threads* e esse foi o valor utilizado ao longo da pesquisa em todos os testes.

Outra característica diz respeito a como a execução termina. A execução da ferramenta de estresse pode ser definida por um número fixo de operações, nesse caso ela termina quando todas forem realizadas. Este método foi o utilizado na simulação para se comparar as estratégias de compactação, presentes na Seção 4.1. Nesses casos, as execuções têm durações diferentes, de acordo com sua eficiência, o que prejudica a comparação entre diferentes execuções por meio de gráficos, pois uma linha pode terminar antes da outra. Portanto, nas demais execuções optou-se por executar o processo de estresse limitado por tempo, usando o parâmetro `duration`, expresso em minutos. A operação termina exatamente quando o tempo definido é atingido. Desta forma, uma execução com melhor configuração, apresentará maior número de operações realizadas. Esta opção torna a análise dos resultados por meio dos gráficos mais ilustrativa.

A ferramenta *cassandra-stress* gera um arquivo de *log* com métricas, que foram analisadas ao longo deste trabalho. O arquivo exibe métricas a cada 30 segundos e, no final de cada execução, apresenta uma média consolidada de *throughput*, de latência e outras métricas de execução do *Garbage Collector* da máquina virtual Java.

3.4 Métricas

Para atingir o objetivo desta pesquisa, é necessário avaliar a performance e, para fazê-lo, é preciso definir indicadores, que por sua vez são baseados em métricas.

Uma das funções desejáveis de um SGBD é gerar métricas que permitam ao administrador avaliar seu desempenho e, nesse quesito, o Cassandra não é diferente e provê um

amplo leque de métricas. Cada nó do *cluster* gera suas métricas, o que permite a avaliação individual, mas precisam ser consolidadas para uma avaliação integral do sistema.

Durante a pesquisa foram utilizados dois métodos para recuperar as métricas, um usado durante o período de observação do funcionamento, para realizar o ajuste do banco e obter pontos ótimos de configuração das estratégias de compactação analisadas. E outro método durante o funcionamento do programa de *auto-tuning* C*DynaConf.

Durante o período de execuções, foi configurado no Cassandra um pacote de exportação de métricas para o formato CSV[1], chamado *Core Metrics*. Este pacote recebe, via arquivo de configuração YAML, quais métricas devem ser exportadas e a periodicidade com que o sistema deve gravá-las em disco, que foi definida em 30 segundos.

A funcionalidade de exportar dados para CSV tem um problema, que é ao se reiniciar o Cassandra, se o diretório de métricas já estiver criado, o Cassandra não se inicializa, apresentando erro. Neste ponto foi feita uma contribuição ao sistema Cassandra, pois foi acrescentada a funcionalidade de, quando se reinicia o sistema Cassandra, ele automaticamente cria um novo diretório para armazenar as métricas, adicionando ao nome do diretório um *timestamp* do momento da inicialização. Essa contribuição foi submetida à comunidade, mas ainda não foi avaliada nem incorporada.

Foi criado um *script*, na linguagem Perl, que lê os arquivos CSV e armazena as métricas em um SGBD MySQL, em outro servidor na rede. Este *script* e os arquivos de configuração das métricas estão disponíveis em (Dias, 2018). Os dados não foram armazenados no próprio Cassandra porque se fossem consultados durante a execução dos testes, poderia causar alterações no desempenho. Além disso, como se tratam de dados estruturados e sem criticidade quanto ao seu desempenho, um SGBDR mostra-se mais adequado. O MySQL era a opção já disponível no laboratório.

No entanto, não convém que o *software* de *auto-tuning* C*DynaConf leia arquivos em discos de diferentes nós na rede. Além de muito custoso, isso ensejaria configuração de compartilhamento de arquivos em disco por todos os nós, o que em um ambiente de produção, demandaria custo extra de configuração. Ao invés de ler em disco, o C*DynaConf captura as métricas, por meio do próprio *driver* de conexão, via protocolo *Java Management Extensions* (JMX). As métricas são as mesmas, somente a maneira de recuperá-las é diferente. A periodicidade com que o C*DynaConf funciona é também de 30 segundos. Assim, as métricas mais utilizadas durante o processo foram:

- *throughput* – a quantidade de operações por segundo. O *throughput* analisado nesta dissertação refere-se à medida gerada pelo *log* do Cassandra Stress Tool. No caso das operações de leitura, é definido pela quantidade de consultas por segundo, não

[1]Inicialmente foi usado o software Graphite mas não se mostrou adequado pois ele não provê persistência de dados.

importando quantas linhas cada consulta recuperou. No caso do INSERT, cada operação é definida como uma inserção de partição (`device id`), com 5 séries temporais (`service name`) e 60 tempos de observação, o que resulta num total de 300 linhas inseridas por operação. Nos cenários em que se gera a mesma quantidade de dados, o sistema com o maior *throughput* termina a execução primeiro. Nos cenários em que se limita a execução por tempo, a configuração com maior *throughput* realiza mais operações;

- latência – o tempo necessário para o sistema de banco de dados responder a uma requisição. Ela começa a contar o quando a requisição é recebida e cessa a contagem quando a mensagem de confirmação ou de erro é entregue ao cliente. Neste trabalho avaliamos tanto a latência de escrita quanto a de leitura. A latência pode ser medida em cada nó, quando é medida somente a partir do momento em que o nó recebeu a requisição, seja do usuário, seja de outro nó coordenador ou pode ser medida também em relação ao cliente. Neste caso é medida no nó que faz a interface com o usuário, que é chamado coordenador.

- espaço em disco – a quantidade total de *bytes* necessários para armazenar os dados. Esta métrica, diferentemente das anteriores, deve receber a soma entre os nós;

- tempo de execução – esta métrica foi usada no teste detalhado na Seção 4.1, que foi simulada com número fixo de operações. Definida uma quantidade de inserções e consultas a serem feitas, a configuração mais eficiente é aquela que finaliza primeiro;

- número de partições tocadas – esta métrica foi usada nos casos em que a execução foi limitada pelo parâmetro do *Cassandra-stress tool* `duration`. Ela é diretamente proporcional ao número de operações de leitura e de escrita realizadas. Usa-se o termo "tocada" pois, se a partição não existia antes, ela é inserida. Caso a partição já exista, novos dados são inseridos naquela partição. Cada partição, quando tocada por operação de escrita, recebe 300 linhas (5 séries com 60 *timestamps* cada). Já em uma operação de leitura, cada consulta realizada toca apenas uma partição. Isto ocorre porque uma consulta sempre tem em seu critério a chave de partição, o que faz com que dados de apenas uma partição sejam retornados. O número de partições tocadas é apresentado pelo Cassandra Stress Tool em seus *logs*, e já representa o total do *cluster*. A configuração mais performática é a que consegue tocar mais partições em um determinado período de tempo.

Além dessas métricas, há duas métricas usadas pelo C*DynaConf, as médias móveis exponencialmente ponderadas, chamadas de *ClientRequest.Read.Latency.FiveMinuteRate* e *ClientRequest.Write.Latency.FiveMinuteRate*, que serão detalhadas na Seção 5.2.

Capítulo 4

Aprendizagem da Compactação

Este capítulo trata da obtenção de pontos ótimos de configuração para subsidiar a construção do C*DynaConf. Na Seção 4.1 a estratégia de compactação mais adequada ao armazenamento de dados de IoT é escolhida, testes são realizados e os resultados são analisados. Uma vez escolhida a estratégia, a Seção 4.2 estuda quais parâmetros possuem maior impacto no desempenho e, por isso, devem constar no mecanismo de *auto-tuning*. A Seção 4.2 apresenta ainda os resultados de diversas execuções com valores distintos dos referidos parâmetros, analisa quais foram os valores que resultaram nos melhores indicadores de desempenho e consolida os pontos ótimos, apresentando gráficos e tabelas.

4.1 Escolha da Estratégia de Compactação

A comunidade do Cassandra fez alguns testes de performance (Jirsa & Eriksson, 2016) e chegou à conclusão de que o TWCS poderia substituir o DTCS, no entanto, nenhuma publicação acadêmica foi encontrada com resultados e análises de testes de comparação entre as duas estratégias de compactação. Esta seção descreve como foram projetados e executados os testes de comparação entre as duas estratégias de compactação citadas e analisa seus resultados.

A criação do componente de *auto-tuning* já foi bastante dispendiosa considerando apenas uma estratégia de compactação, no entanto, o Cassandra possui duas estratégias de compactação típicas de dados como os de IoT: DTCS e TWCS. Coube a este trabalho escolher qual seria a mais adequada.

O DTCS possui 10 parâmetros, conforme explicitado na Subseção 2.4.5. No entanto, os testes foram feitos equiparando-se os valores de `base time seconds` com o parâmetro `compact window size` da estratégia TWCS, para se obter resultados comparáveis, uma vez que ambos são baseados em janelas temporais. Quem define o tamanho da janela tem-

poral nas estratégias DTCS e TWCS são os parâmetros `base time seconds` e `compact window size`, respectivamente.

Para os testes de comparação entre as estratégias, o volume de dados gerado foi sempre o mesmo, e o número de linhas inseridas LI é dado pela Equação 4.1.

$$LI = OP * TXI * ST * VO \tag{4.1}$$

Onde:

LI é o número de linhas inseridas;

OP é o número de operações;

TXI é a taxa percentual de inserção em relação ao total de operações;

ST é o número de séries temporais por sensor;

VO é o número de valores observados por cada série temporal.

A ferramenta de estresse foi configurada para gerar 2.000.000 (dois milhões) de operações, com 90% de operações de escrita e 10% de operações de leitura, o que significa que a taxa de inserção foi de 0,9. Cada operação inseriu dados referentes a exatamente um dispositivo. Conforme referenciado na Seção 3.2, para cada dispositivo, em todos os testes, sempre são geradas 5 séries temporais, que populam a coluna `service name`. Cada série temporal recebe, em uma operação, 60 valores observados. Aplicando os números à Equação 4.1 tem-se à igualdade 4.2.

$$LI = 2,000,000 * 0.9 * 5 * 60 = 540,000,000 \tag{4.2}$$

Em cada execução foram geradas e inseridas no banco Cassandra 540 milhões de linhas de dados – também chamadas de registros – de séries temporais, distribuídos entre os dez nós do *cluster* e 60 milhões de operações de leitura foram executadas, as quais representam 10% do total. O número de linhas não é contado, pois cada operação de leitura pode trazer um conjunto de resultados diferente, dependendo dos dados inseridos. Mas há um limite de, no máximo, 500 linhas para as consultas `devdat` e `lesrow` e sempre uma única linha na consulta `avgdat`.

O TTL foi definido em 60 minutos, que adicionados ao *grace period* definido em 30 minutos, significa que os dados ficariam expirados e prontos para deleção após 90 minutos. Cada execução durou, no mínimo, 120 minutos, de modo que o comportamento da expiração dos dados e sua desalocação pudesse ser percebido. Vale ressaltar que a partir dos 90 minutos de execução, à medida que novos dados eram inseridos, dados antigos eram desalocados, o que significa que as 540 milhões de linhas não coexistiram no mesmo instante no banco de dados.

Tabela 4.1: Tempo médio para as 600M de operações usando DTCS and TWCS.

Estratégia	**Tempo médio**	**Desvio** (%)	**Speedup**
DTCS	02h41m48s	0,71%	N/A
TWCS	02h14m57s	0,18%	1,20

Para comparar as duas estratégias, o mesmo volume de dados simulados foi inserido e consultado em duas famílias de colunas, uma de cada vez. O esquema das duas tabelas, que pode ser visto na Figura 3.2, eram iguais exceto pela estratégia de compactação, na qual uma usava o DTCS e a outra usava o TWCS. A tabela do primeiro usava os parâmetros *default*, com exceção do `base time seconds` que foi definido em 10 minutos. Do mesmo modo, a família de colunas que teve a TWCS aplicada recebeu o parâmetro `compaction window size` definido em 10 minutos e os demais parâmetros seguiram os valores padrão.

4.1.1 Resultados da comparação entre DTCS e TWCS

O primeiro indicador de performance exibido é o tempo decorrido necessário para realizar todas as operações de leitura e escrita. No total, seis operações de estresse foram feitas em cada estratégia de compactação e, como o desvio padrão foi baixo, serão apresentados a média e o desvio padrão, este último em termos percentuais da média. Os resultados são exibidos na Tabela 4.1. O *speedup* é o tempo do DTCS dividido pelo tempo do TWCS.

A tabela configurada com TWCS foi 20% mais rápida para tratar a mesma quantidade de dados que a família de colunas configurada com DTCS. Esta vantagem também pode ser observada no *throughput* de operações de inserção, expressada em linhas inseridas por segundo, na Figura 4.1. Mais uma vez, como foram seis operações, uma média foi feita. O desvio padrão entre todas as simulações foi de 6,5%, valor este calculado sobre as médias de *throughput* de cada execução e não individualmente a cada intervalo de 30 segundos.

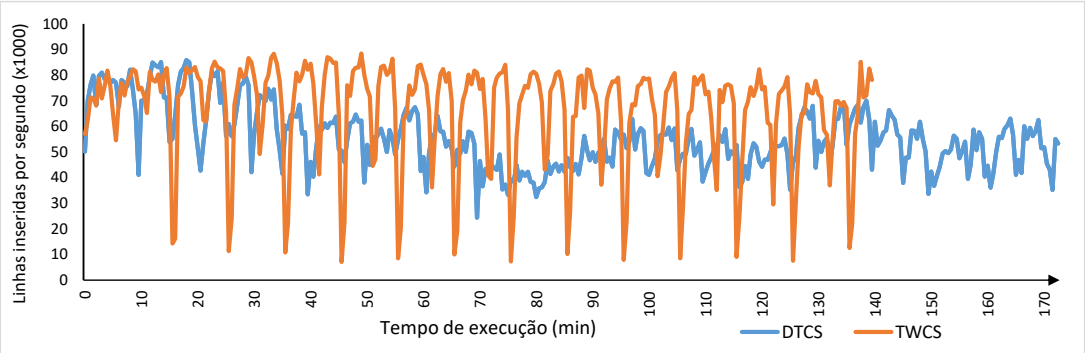

Figura 4.1: Número de linhas inseridas por segundo: DTCS e TWCS.

Na Figura 4.1 cada teste de estresse foi executado em tempos diferentes mas exibidos da mesma linha do tempo para comparação. O eixo `y` representa milhares de linhas inseridas por segundo enquanto que o eixo `x` expressa o tempo de execução, em minutos. Diante disso, notou-se que a execução da configuração TWCS termina um pouco antes. Ela possui ainda um período de baixa mais definido, que ocorre a cada 10 minutos, que é justamente o tempo da janela de execução. Já no DTCS esse período de baixa não é regular. Isto indica que a compactação no DTCS não ocorre exatamente a cada 10 minutos em todos os nós. É importante lembrar que ambas as curvas representam uma média de *throughput* entre seis execuções, o que suaviza eventuais picos e pontos de baixa.

Já foi mostrado que a escrita foi mais eficiente com a estratégia mais recentemente desenvolvida, a TWCS. Quanto às operações de leitura, que são mais suscetíveis à organização interna das *SSTables* realizada pelas compactações, o TWCS também leva vantagem, analisando-se a latência. Diferentemente do *throughput*, em que foi analisada a média de seis simulações, neste caso considerou-se uma simulação de cada ambiente, sendo a escolhida aquela mais próxima à média. Assim sendo, é possível ver as latências de ambas as estratégias na Tabela 4.2. O grupo de colunas Total representam a latência considerando-se todas as operações, de leitura e de escrita. Como há 90% de operações de escrita e 10% de operações de leitura, a primeira tem um peso maior no Total. A coluna *Speedup* representa a latência do DTCS, dividida pela latência do TWCS, o que demonstra a vantagem do último em relação ao primeiro.

Tabela 4.2: Estatísticas de Latência com Percentis: DTCS x TWCS.

Medidas	**Read (ms)**		**Insert (ms)**		**Total (ms)**		
	DTCS	**TWCS**	**DTCS**	**TWCS**	**DTCS**	**TWCS**	***Speedup***
média	272,4	181,8	102,6	88,6	119,5	97,9	1,22
mediana	63,5	62,0	37,3	40,9	38,5	41,5	0,93
0,95	1057,5	595,9	278,9	231,6	363,6	260,0	1,40
0,99	3253,4	1683,7	1252,0	426,2	1618,0	677,4	2,39
0,999	7152,7	8018,1	5431,6	4945,1	5779,8	5326,8	1,09
máxima	17753,1	27263,0	24142,4	46271,6	24142,4	46271,6	0,52

Cada operação gera um resultado de latência. Como são dois milhões de operações realizadas, não é possível mostrar os valores de cada uma. Na coluna da esquerda da Tabela 4.2 são mostradas as medidas tomadas sobre o universo de operações.

Os valores numéricos representam os percentis superiores. Os percentis são medidas que dividem a amostra, ordenadas da menor latência para a maior latência em 100 partes, cada uma com uma porcentagem de dados aproximadamente igual (Pagano & Gauvreau, 2004). Por exemplo, o valor 0,95 indica o valor que separa os 95% de operações com menor latência dos 5% de operações com maior latência.

Assim como o percebido no tempo de execução, o TWCS mostrou uma menor latência, com decréscimo de 22% em relação ao DTCS, tanto nas operações de leitura quanto nas operações de escrita. Este número pode ser visto no *Speedup* da média do total de operações. Pode-se perceber que a vantagem é mais acentuada nas operações de leitura.

Dessa forma, embora a mediana da latência total tenha sido ligeiramente mais rápida no DTCS, isso não representa uma vantagem relevante, porque a média do TWCS e outros dois percentis são menores. O percentil 0,99 revela que o Cassandra pôde responder a 99% das requisições em 677,4 milissegundos usando TWCS, enquanto que usando o DTCS o Cassandra levou até 1618,9 milissegundos. Os percentis superiores da tabela mostram que o banco de dados respondeu à maioria das requisições em um tempo aceitável.

Em muitos ambientes NoSQL, os percentis superiores são relevantes indicadores de performance. A latência máxima mostra números altos, de até 46 segundos, mas há que se considerar que foram executadas 600 milhões de operações, e garantir uma latência baixa para todas as requisições teria um custo muito alto (DeCandia *et al.* , 2007). Contudo, é importante lembrar que o hardware utilizado neste caso (e em muitos ambientes NoSQL) é comum, sem componentes especializados para alto desempenho.

A máquina virtual Java possui um componente de *Garbage Collection*, que é o responsável por desalocar da memória objetos e variáveis não mais usados. Seu funcionamento é composto de várias etapas e uma delas, a mais crítica para o Cassandra, chama-se "parada do mundo"[1], que leva esse nome porque implica em uma pausa de todas as *threads* em execução da máquina virtual, para limpeza da memória (Carpen-Amarie *et al.* , 2015). Estas pausas geram um aumento das latências máximas e dos percentis superiores .

Para analisar o volume em disco ocupado, diferentemente da latência, há que se somar os valores individuais dos 10 nós do *cluster*. As operações de compactação se dão nos nós e não ocorrem exatamente no mesmo período, no entanto, como os 10 nós possuem o mesmo tipo de hardware e a mesma capacidade, a distribuição entre os nós pode ser considerada uniforme, pois cada nó é responsável por aproximadamente 10% do espaço utilizado, conforme pode ser visto nas Figuras 4.2 a 4.3.

Os valores exibidos nas Figuras 4.2 a 4.3 referem-se a uma das seis execuções. Foi escolhida aquela mais próxima à média, nas duas estratégias de compactação utilizadas. O espaço em disco está exibido de forma empilhada, onde o volume usado por cada nó corresponde a uma camada. Percebe-se que o TWCS mostra mais operações de desalocação, que ocorrem a partir do minuto 110, em intervalos de 10 minutos. Já o DTCS mostra queda de espaço alocado em dois momentos. A primeira ocorre próximo ao minuto 120, de menor amplitude. A segunda, de maior magnitude, ocorre aproximadamente no

[1]Tradução nossa de "Stop the world".

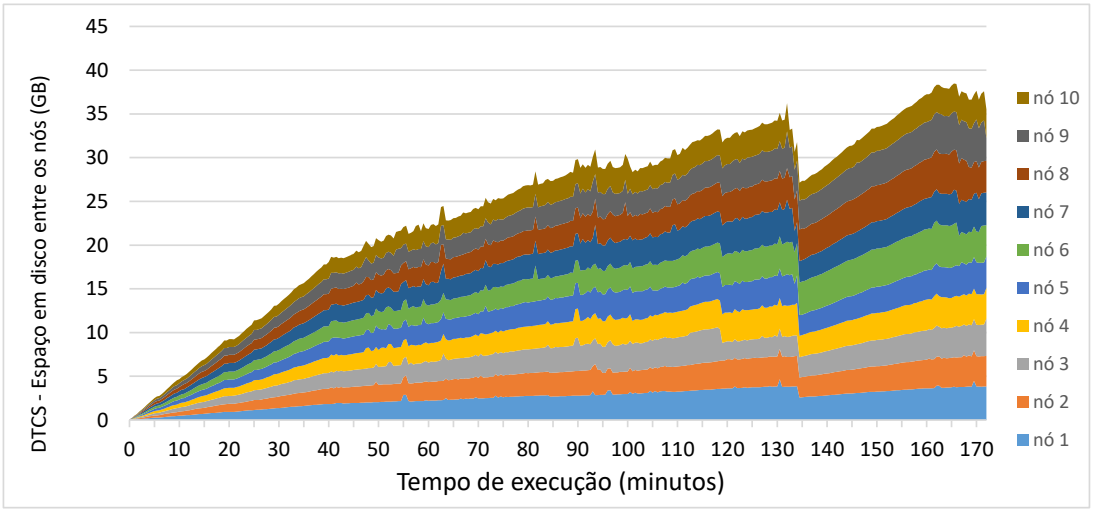

Figura 4.2: Espaço em disco usado pelos nós - DTCS.

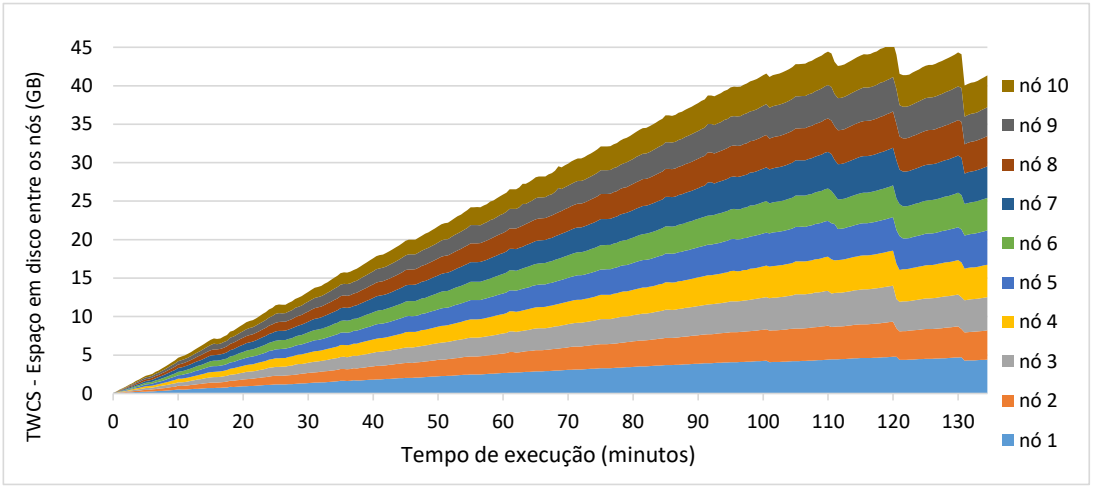

Figura 4.3: Espaço em disco usado pelos nós - TWCS.

minuto 135. Esta foi uma compactação de segundo nível, ou seja, que agrupou *SSTables* já previamente compactadas.

O DTCS demonstrou uma vantagem sobre o TWCS em termos de espaço em disco usado. A Figura 4.4 mostra, na mesma linha do tempo, a média, entre as seis execuções, da soma de espaço em disco entre os nós.

É importante lembrar que os dados expiram após 90 minutos (60 do TTL mais 30 do *grace period*) e, embora novas linhas sejam constantemente inseridas, o volume se torna estável após 110 minutos de execução. Percebe-se na curva do TWCS que há três compactações, nos minutos 110, 120 e 130, com um intervalo de aproximadamente 10 minutos entre elas, que é a janela de compactação definida. Por outro lado, no DTCS as compactações não estão bem definidas, porque não ocorreram exatamente no mesmo

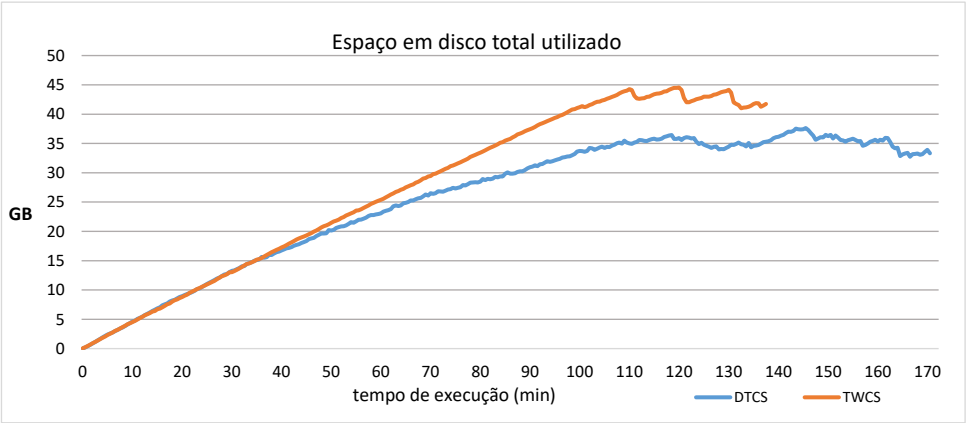

Figura 4.4: Espaço em Disco ao Longo da Execução: DTCS e TWCS.

Tabela 4.3: Espaço em Disco Total Usado por DTCS e TWCS.

	DTCS(GB)	**TWCS(GB)**	**Acréscimo TWCS**
Média	25,64	27,05	5,5%
Máximo	37,63	44,55	18,4%

intervalo entre as seis execuções tomadas.

Graficamente fica evidente que o DTCS utiliza menos espaço em disco do que o TWCS, o que pode ser notado em valores numéricos na Tabela 4.3. Em valores totais, o espaço máximo necessário foi 18,4% mais alto no TWCS e, em média, 5,5% a mais. Os testes foram executados com a compressão desabilitada, para não degradar a performance.

O motivo principal pelo qual a estratégia DTCS é mais eficiente quanto ao espaço, é que ela faz uma compactação em vários níveis, diferentemente do TWCS. Este último faz apenas uma compactação de agrupamento de *SSTables*, quando a quantidade de tabelas atinge o parâmetro `min threshold`, e estão dentro da janela de tempo, após isso, o TWCS não mais compactará aquelas *SSTables*, a menos que elas estejam expiradas.

Por outro lado, o DTCS faz diversas compactações com *SSTables* mesmo após passado o tempo da janela temporal, podendo agrupar inclusive as já compactadas antes, criando *SSTables* cada vez maiores ao longo do tempo. Todavia, as operações de compactação multinível são mais demoradas e comprometem o *throughput* de inserção, bem como as latências, pois operações de compactação são de I/O intensivo.

Em diversos casos na Ciência da Computação existe uma permutação entre tempo e espaço, na qual se prioriza um em detrimento do outro. Este caso não foge à regra e a estratégia de menor tempo de resposta TWCS utiliza um pouco mais de espaço em disco que a estratégia com pior tempo de resposta. No entanto, melhorias podem ser feitas para aproveitar as características de compactação multinível do DTCS no TWCS, pois há indícios que isso trará mais eficiência de espaço. Contudo, essas melhorias na estratégias

de compactação estão fora do escopo deste trabalho.

Nos testes apresentados, o TWCS foi mais eficiente que o DTCS, executando a mesma tarefa 20% mais rápido. Métricas de *throughput* e latência também evidenciam essa vantagem. Além disso, a comunidade do Cassandra já marcou como deprecado o DTCS e recomenda o uso da TWCS, decisão que é considerada acertada por este trabalho. A estratégia mais eficiente, também possui menos parâmetros, o que diminui a complexidade de um componente de *auto-tuning*. Por isso, a estratégia TWCS foi escolhida para este trabalho e será alvo do mecanismo C*DynaConf.

4.2 Tuning da Estratégia TWCS

Escolhida a estratégia TWCS para o C*DynaConf, a mesma foi ajustada para obter pontos ótimos de desempenho para subsidiar a construção do componente de *auto-tuning*. Esta seção descreve como foi feita a escolha dos parâmetros de configuração do TWCS e as execuções dos testes variando-se tais parâmetros. Os resultados são analisados e são indicados para uso no mecanismo de *auto-tuning*.

4.2.1 Escolha dos Parâmetros do TWCS

Conforme visto na Subseção 2.4.6, a TWCS possui apenas dois parâmetros, o tamanho da janela temporal (`compaction window size`) e a unidade de medida dessa janela (`compaction window unit`), que só pode ser minuto, dia ou hora. Este último parâmetro foi definido, em todas as execuções como minuto, pois adotando-se essa unidade pode-se chegar a qualquer intervalo maior, como horas ou dias. Quanto ao primeiro parâmetro, foram feitas diversas execuções e os resultados foram analisados. Portanto, o parâmetro `compaction window size` é um dos parâmetros considerados pelo *auto-tuning*.

Além de seus próprios parâmetros, a TWCS também aceita os parâmetros da STCS, pois esta é a estratégia usada nas primeiras compactações, enquanto ainda não se atingiu o intervalo de janela temporal. Dentre seus dez parâmetros, quatro são relativos às compactações feitas para expurgar os *tombstones* (Alapati, 2018), que já são tratados adequadamente pela TWCS e por isso, em teoria, não teriam impacto no desempenho desta estratégia, se o uso for para dados de IoT. Estes parâmetros tiveram seus valores padrão mantidos, os quais são: `tombstone threshold`, `tombstone compaction interval`, `unchecked tombstone compaction` e `only purge repaired tombstone` (Cassandra, 2018).

O parâmetro `log all`, comum a todas as estratégias de compactação, indica que todas as operações de compactação devem estar presentes em arquivo de *log* próprio, chamado *log* de compactação. Esta opção foi definida como `true` em todas as execuções, pois assim se obteriam mais dados para analisar os resultados. Entretanto, após análise detalhada

Tabela 4.4: Quantidade Média de *SSTables* em cada Compactação STCS.

min threshold	**max threshold**	**Média SSTables**
2	32	3,17
4	32	4,85
5	32	5,67
6	32	6,27
8	32	8,08
10	32	10,00
12	32	12,00

desses *logs*, percebeu-se que as informações presentes no arquivo de *log* gerados estavam todas contidas no *log* de *debug*, cuja criação é configurada em cada nó do Cassandra. Além disso, este *log* de *debug* possui mais dados referentes à compactação do que os dados do próprio *log* de compactação, o que o torna desnecessário. Já existe uma iniciativa para melhorar o *log* de compactação do Cassandra, proposto por (Yeksigian & Luciani, 2016), que traria mais informações, tornando seu uso mais proveitoso. Ainda que os *logs* não tenham sido úteis, para efeitos de comparação com as primeiras execuções, adotou-se sempre o valor `true`, uma vez que a geração de *log* exige o uso de recursos computacionais e a mudança do parâmetro poderia ser motivo de alteração nos resultados.

Os parâmetros do STCS que foram analisados quanto à performance foram o `min threshold` e `max threshold`. Eles definem, respectivamente, os valores mínimo e máximo da quantidade de *SSTables* que deve estar em disco para haver uma compactação. O valor padrão no `min threshold` é 4 e do `max threshold` é 32.

Para proceder a análise foi desenvolvido um programa que lê e sumariza os *logs* de *debug* do Cassandra. Além disso, foram analisadas 7 execuções, com valores de `min threshold` variando de 2 a 14, e mantendo-se sempre o `max threshold` em 32. Percebeu-se que, em um ambiente de alta carga de escrita como o simulado pela ferramenta de estresse, o parâmetro `max threshold` nunca é atingido, conforme pode ser visto na Tabela 4.4. O valor da coluna da direita representa a média de número de *SSTables* em cada compactação, pois durante cada execução ocorrem diversas operações de compactação. O programa foi escrito na linguagem Perl e está disponível em (Dias, 2018).

Em um ambiente de escrita intensiva, observa-se que assim que o número mínimo de *SSTables* é alcançado, a compactação ocorre. Apenas quando o parâmetro `min threshold` foi configurado com valor muito baixo, de 2, é que o número de *SSTables* médio presente em cada operação de compactação supera significativamente o número mínimo definido. Nos valores mais altos, a partir de 10, nenhuma compactação ocorre com valor acima do limite mínimo. Logo, os resultados da análise de *logs*, sumarizados na Tabela 4.4 mostram que o parâmetro `max threshold` não foi atingido e, portanto, sua alteração não traria

impacto nenhum nas execuções. Por isso, definiu-se que o parâmetro `max threshold` não deve fazer parte dos parâmetros alterados pelo C*DynaConf, por outro lado, o parâmetro `min threshold` é considerado porque sua alteração traz algum efeito sobre o desempenho do *cluster*.

Os parâmetros `average table size`, `bucket low` e `bucket high` (cuja definição está na Subseção 2.4.3) não foram considerados para simplificação da quantidade de variáveis a serem consideradas pelo *auto-tuning*. Mas, como seus valores impactam na ocorrência de compactações, é provável que sua variação tenha repercussão no desempenho. Trabalhos futuros deverão avaliar esses parâmetros e, caso seja mostrada influência destes com o desempenho, incluí-los no mecanismo de *auto-tuning*.

Assim sendo, os parâmetros considerados para o *performance tuning* são o `compaction window size` e o `min threshold`.

4.2.2 Parâmetro Compaction Window Size

Este é o principal parâmetro da TWCS e define o tamanho da janela temporal dentro da qual as *SSTables* sofrerão compactação. Após passar pelo período de compactação, as *SSTables* não sofrerão mais compactação oriunda da TWCS.

Seu valor mínimo é de 1 minuto e pode ir até milhares de anos. Neste trabalho, adotou-se intervalos de 1 minuto até o limite de 80 minutos, pois nos testes realizados observou-se que após esse limite não houve melhoria. O uso de intervalos muito longos também dificultaria o trabalho de simulação de ambientes IoT pois, além de ter que executar mais cenários, o tempo de execução passa a ter que ser maior, o que impacta diretamente nos prazos da pesquisa, que tem tempo limitado.

Conforme visto na Seção 3.3, a ferramenta Cassandra-Stress foi configurada para executar os testes de carga por uma duração fixa. Neste caso, as execuções que realizaram mais operações são as mais eficientes. Portanto, um indicador chave de perfomance é o número total de partições tocadas, que agrega partições inseridas e partições consultadas. Cada partição inserida, no modelo de dados usado, tem 300 linhas (5 séries temporais com 60 observações cada). Há que se ressaltar que cada partição pode ser visitada várias vezes por consultas, e a cada leitura esse indicador é incrementado. Esta grandeza é diretamente proporcional ao *throughput*, que mede a quantidade de operações por segundo. Por fim, a latência também está relacionada diretamente à performance, sendo outro indicador chave e por vezes são apresentadas as latências de leitura, de escrita e total.

Nos primeiros testes fixou-se a proporção de leitura em 10% das operações e variou-se o TTL de 1 a 3 horas. O tempo de execução dos testes variou de acordo com o TTL, pois para valores maiores, o tempo de execução também deve ser maior, para poder observar

Tabela 4.5: Tempo de Execução - Leitura 10%.

TTL	Duração	window size
1h	150min	1 a 80
2h	200min	1 a 80
3h	300min	1 a 50

dados expirando. Os valores de tempos de duração e os intervalos do tamanho da janela temporal estão na Tabela 4.5.

Os resultados da execução estão na Figura 4.5. Cada ponto do gráfico representa uma execução, com os valores de duração da Tabela 4.5. Os pontos máximos tiveram sua cor destacada. Como as três séries representadas apresentam execuções com diferentes durações, as séries que levam mais tempo possuem mais partições tocadas.

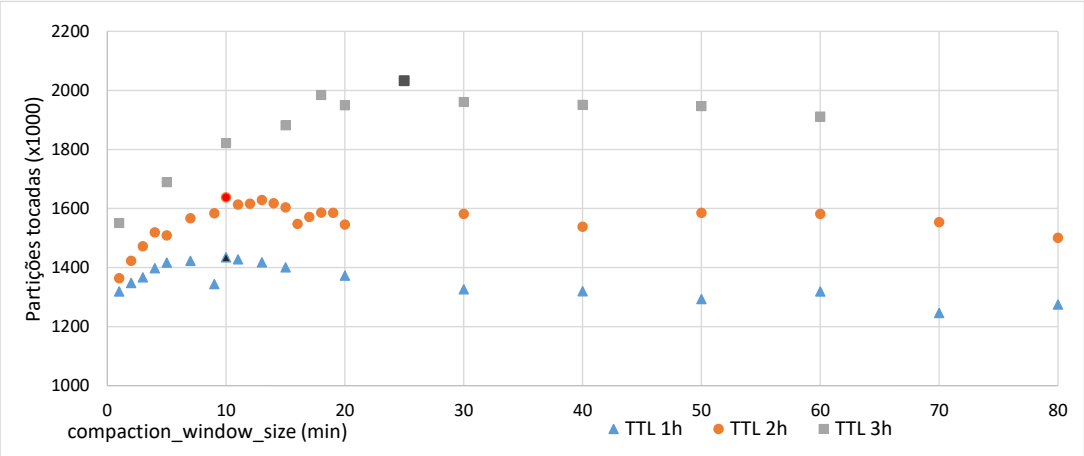

Figura 4.5: Número Total de Partições Tocadas - Leitura 10%.

Os pontos ótimos obtidos nas séries com TTL de 1h e 2h coincidem no mesmo tamanho de janela, de 10 minutos. Já na execução com TTL definido em 3h, o melhor desempenho ficou com a `compaction window size` definido em 25 minutos.

Em geral, as execuções que duram menos possuem maior número de partições tocadas por minuto, o que pode ser visto na Figura 4.6, que mostra o *throughput* de cada execução. Os pontos máximos tiveram sua cor destacada. Há uma relação direta entre o número de partições tocadas por segundo e o *throughput*, pois o Cassandra-Stress tool executa cada operação tocando uma única partição, por isso, os pontos ótimos quanto ao *throughput* foram os mesmos que os relativos às partições tocadas.

Nas execuções mais demoradas, há mais tempo para operações de inserção, o que resulta em um banco de dados mais carregado, isto é, com maior número de registros. Com um maior volume de dados, as operações tendem a ser mais lentas, pois no caso da leitura, há um universo maior para se procurar os dados e, no caso da escrita, as

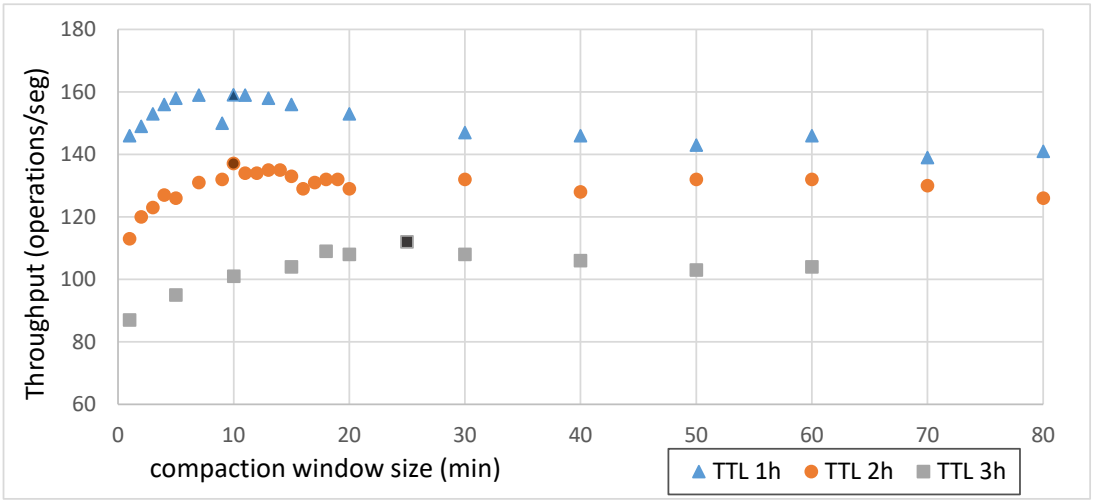

Figura 4.6: *Throughput* - Leitura 10%.

operações têm que disputar recursos com as compactações de *SSTables*, que são de uso intensivo do sistema de I/O. Isso resulta num *throughput* menor nas execuções com maior duração. Para ilustrar, a Figura 4.7 exibe dados de *throughput* de dois casos de teste, das execuções que obtiveram os melhores desempenhos no cenário analisado, de 10% de proporção para operações de leitura. A série com TTL definido em 1h teve duração de 150 minutos, enquanto a série de TTL de 2h teve 200 minutos. O comportamento de ambas é semelhante até que os dados da primeira comecem a expirar, o que ocorre após 90 minutos (o *grace period* se soma ao TTL).

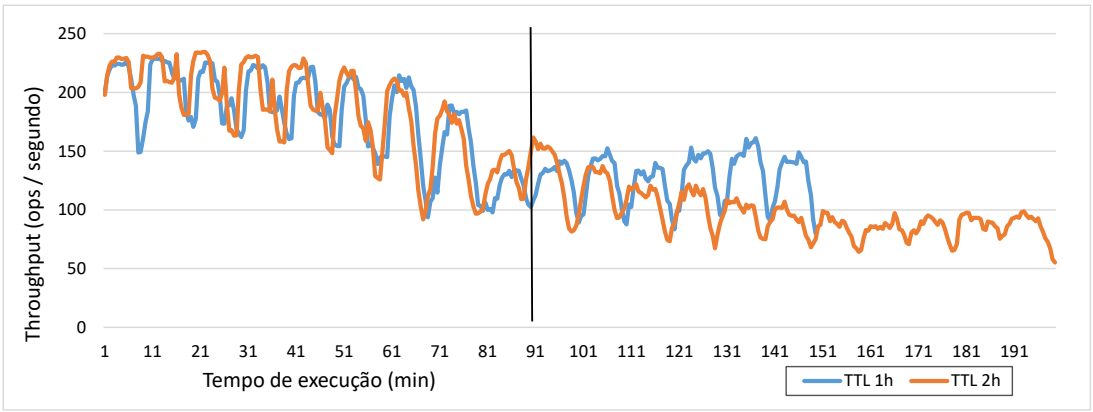

Figura 4.7: Médias Móveis de 5 Períodos de *Throughput* - Duas execuções.

Os dados de *throughput* oscilam muito ponto a ponto, a cada intervalo de 30 segundos. Para suavizar as curvas e tornar o gráfico mais ilustrativo, foi aplicado às séries de dados de *throughput*, a média móvel de 5 períodos, mostrados na Figura 4.7. O uso de médias móveis é uma das maneiras mais simples de suavizar gráficos (NIST, 2003). Sua aplicação

é fácil, a cada ponto deve-se computar a média dos dois pontos anteriores, do ponto atual e de dois pontos posteriores.

A Figura 4.7 mostra que até os dados começarem a ser expurgados, o que acontece após 90 minutos, os valores de *throughput* são similares. A partir do minuto 101 o *throughput* da execução com TTL de 2h fica abaixo dos valores da execução com TTL de 1h. Isto ocorre porque a execução de TTL 2h passa a ter mais dados em relação à de TTL 1h. Com maior volume de dados, há maior carga das operações de compactação, bem como um maior universo para se buscar dados, para as operações de leitura, o que impacta negativamente no desempenho.

Outro indicador chave que mostra quais casos de testes são os mais eficientes é a latência média. Ela indica qual foi o tempo de resposta médio entre todas as operações de leitura e de escrita e é registrada em milissegundos. Quanto menor a latência, mais rápidas são as operações. A latência das execuções por cada `compaction window size` é apresentada na Figura 4.8. Igualmente aos outros gráficos de partições tocadas e de *throughput*, os pontos ótimos das séries com TTL 1h, 2h e 3h foram, respectivamente, com tamanhos de janela de 10, 10 e 25 minutos.

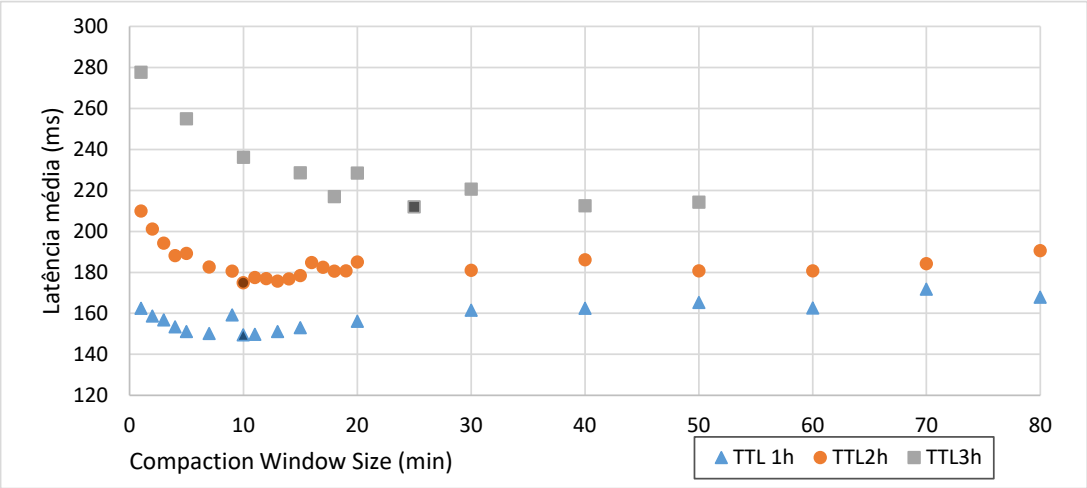

Figura 4.8: Latência Média com Proporção de Leitura em 10%.

A Figura 4.8 exibe a latência média total, de operações de leitura e de escrita, considerando que a escrita tem um peso bem maior nessa média, pois são responsáveis por 90% das operações nesse cenário. Entretanto, as latências de escrita e de leitura possuem valores bem distintos, com o tempo de resposta das operações de leitura sendo bem maior do que as operações de escrita. Isso se justifica porque cada operação de leitura pode recuperar até 500 linhas. Além disso, o Cassandra é otimizado para altas taxas de inserção (Lakshman & Malik, 2010).

A latência de escrita inicia com seu valor mínimo no menor intervalo de `compaction window size`, em 1 minuto e cresce até a última execução. A latência de leitura, por sua vez, tem seu valor mínimo no `compaction window size` de valor 9 minutos, que é um ponto fora da curva em todos os dados, pois houve um desempenho muito abaixo de seus pontos vizinhos. Se desconsiderarmos esse ponto, o valor mínimo de latência de leitura é em 15 minutos. Já o valor mínimo de latência total, considerando escrita e leitura é em 10 minutos, também ponto ótimo de *throughput* e de partições tocadas.

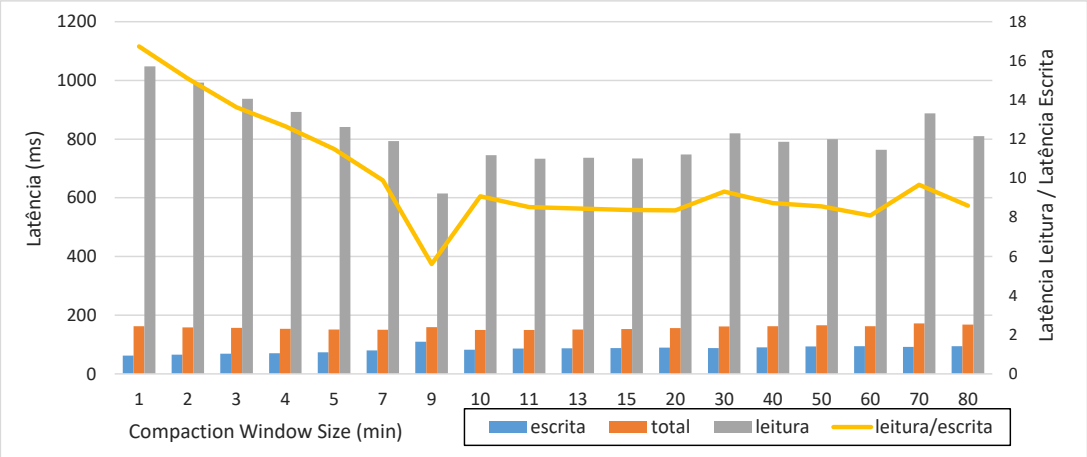

Figura 4.9: Latência Média com Proporção de Leitura em 10% e TTL 1h.

A Figura 4.9 mostra as latências de leitura, de escrita e o total das execuções com dados com TTL de 1h. A latência de escrita é a menor medida dentre as representadas. A latência total, considerando escrita e leitura é mais próxima da latência de escrita, pois esta representa 90% das operações. As latências de leitura, calculada como a média entre as três consultas listadas na Seção 3.2, tem seus valores maiores, aproximadamente 10 vezes o tempo da latência de escrita. Uma razão entre a leitura e a escrita é exibida em uma linha para ressaltar que a proporção entre essas operações é algo crítico para o correto ajuste dos parâmetros do Cassandra, uma vez que os pontos ótimos de latências de escrita, leitura e total são em diferentes tamanhos de `compaction window size`. O eixo y da esquerda representa as latências das colunas, enquanto que o eixo y da direita mede o quociente entre as latências de leitura e escrita.

O ambiente de IoT é de escrita intensiva, com poucas ou nenhuma alteração de dado ou exclusão. No entanto, não existe uma definição entre a proporção de leitura e de escrita, pois isso varia de acordo com as aplicações. Foi visto, na Figura 4.9 que `compaction window size` diferentes geram pontos ótimos diferentes quanto à leitura e à escrita. Por isso, outros dois cenários de proporções foram testados: com 30% de operações de leitura e com 3% de operações de leitura.

O cenário já analisado teve a proporção de operações de leitura configuradas em 10%. A seguir, é analisado o cenário com 70% de escrita e 30% de leitura. Como as operações de leitura levam mais tempo, já era esperado que o número de partições tocadas fosse menor do que o cenário com 10% de operações de leitura.

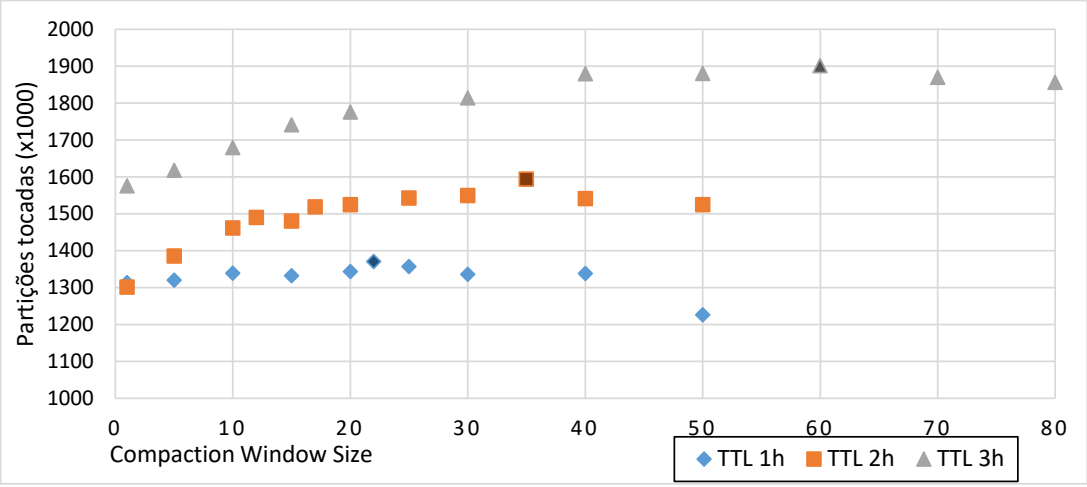

Figura 4.10: Número Total de Partições Tocadas - Leitura 30%.

A Figura 4.10, de maneira semelhante à Figura 4.5, exibe o número de partições tocadas durante os testes com TTL de 1 a 3h. Cada ponto do gráfico representa uma execução e os pontos máximos de cada série tiveram sua cor alterada, pois representam os valores de janela temporal ótimos. Comparando as execuções com proporção de leitura em 30%, em relação à leitura 10%, tem-se que:

(i) O número de partições tocadas é menor. A execução mais eficiente do cenário 30% toca 4,54% menos partições, em média, do que as execuções mais eficientes do cenário de leitura em 10%;

(ii) Os valores de `compaction window size` que obtém os pontos mais eficientes são maiores no cenário de Leitura 30%. Para os TTL de 1h, 2h e 3h os valores foram respectivamente 22, 35 e 60 minutos.

Tamanhos de janela maiores significam que a compactação oriunda do TWCS vai ocorrer menos vezes. A compactação é uma operação de I/O intensivo, mas torna mais rápidas principalmente as operações de leitura. Portanto, era de se esperar que quanto mais compactações ocorressem, melhor seria o resultado com uma proporção maior de operações de leitura. Contudo, o que aconteceu foi o contrário, os tamanhos de janela ótimos foram maiores do que no cenário com leitura em 10%.

Para investigar o motivo de (ii), foi utilizado o *script* de leitura de *logs* de *debug* para observar qual foi o comportamento das operações de compactação. O *log* é gerado em

Tabela 4.6: Quantidade de Operações de Compactação em um Nó e sua Duração, com TTL 1h.

Leitura	**Janela**	**STCS**	**Duração média**	**TWCS**	**Duração média**
10%	10min	69	15,95 ms	14	52,5 ms
30%	10min	71	16,11 ms	14	56,9 ms
30%	22min	156	18,02 ms	6	87,8 ms

cada nó e para essa análise foi utilizado o arquivo de somente um nó, o RT1, no entanto todos os nós apresentam números semelhantes, pois têm as mesmas configurações.

Dessa forma, foram comparadas duas execuções com TTL de 1h e com o `compaction window size` definido em 10 minutos, que é a configuração ótima do cenário de Leitura em 10%, além de comparar com a janela de compactação ótima do cenário de Leitura 30%, que foi com a janela definida em 22 minutos. Os números estão na Tabela 4.6, na qual as colunas STCS e TWCS representam a quantidade de compactações, oriundas dessas estratégias, que ocorreram durante toda a execução.

A Tabela 4.6 nos ajuda a entender por que no cenário com mais leitura, os tamanhos ótimos ficaram maiores. Enquanto a compactação TWCS não ocorre em seu intervalo de janela, as *SSTables* são compactadas pela STCS. Até o `compaction window size` com 22 minutos, a maior proporção entre compactações STCS em relação às compactações do TWCS torna a execução mais eficiente. Portanto, aumentar o tamanho do `compaction window size` diminui apenas o número de compactações do TWCS mas aumenta o número de compactações do STCS.

O outro cenário executado foi com 97% de operações de escrita e 3% de operações de leitura. O número de partições tocadas para execuções com TTL de 1 a 3h é maior do que os cenários com proporção de leitura em 10% e 30% e pode ser visto na Figura 4.11. Os pontos máximos tiveram suas cores destacadas.

No cenário de leitura 3% o número de partições tocadas supera os cenários de leitura 10% e de leitura 30%. No primeiro caso, o incremento é de 12% de partições tocadas e no último é de 18%. Isto ocorreu porque as operações de INSERT são mais rápidas do que as operações de leitura, e este cenário possui menos operações de leitura.

Os valores de `compaction window size` para os cenários de TTL de 1h, 2h e 3h foram em 2 minutos, 12 minutos e 35 minutos, respectivamente. Em todos os cenários se observou que, quanto maior o TTL, maior ou igual deverá ser o `compaction window size` para se obter uma configuração próxima à ótima.

Os pontos ótimos de `compaction window size` foram sumarizados na Tabela 4.7. Esses resultados são a base do desenvolvimento do mecanismo C*DynaConf.

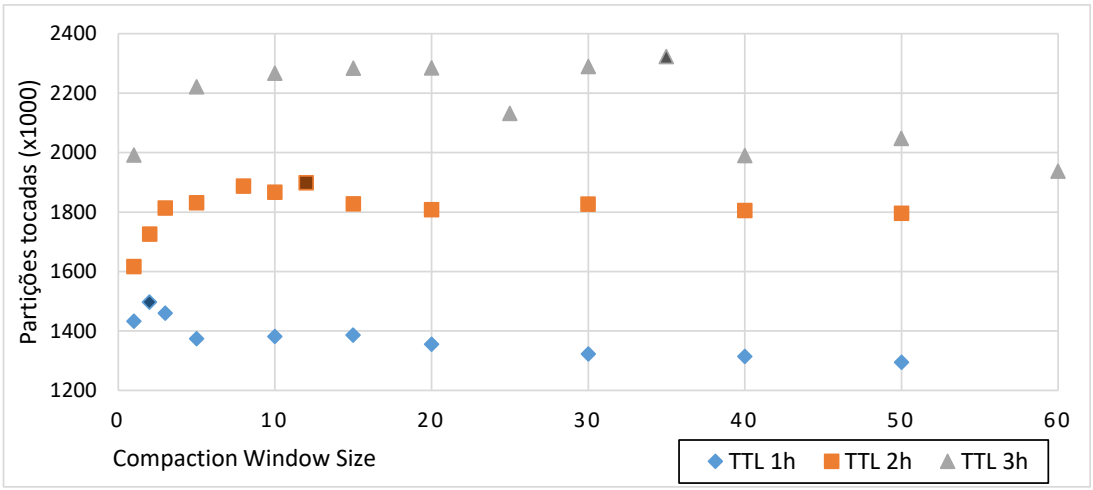

Figura 4.11: Número Total de Partições Tocadas - Leitura 3%.

Tabela 4.7: Compaction Window Size - Pontos Ótimos (minutos).

Cenários	TTL 1h	TTL 2h	TTL 3h
Leitura 3%	2	12	35
Leitura 10%	10	10	25
Leitura 30%	22	35	60

4.2.3 Parâmetro Min Threshold

O parâmetro de compactação `min threshold` define um número mínimo de *SSTables* que deve existir em disco para que a compactação oriunda da estratégia STCS ocorra. Importante lembrar que na estratégia TWCS, enquanto não se atinge o tamanho da janela temporal, em minutos, compactações são feitas pelas regras da STCS.

O parâmetro `min threshold` tem como valor padrão 4. Para testar quais são os valores de configuração ótima para cada TTL foram feitas simulações, que variaram os valores do parâmetro de 2 a 14.

Nos testes para encontrar os pontos ótimos de `compaction window size` já tinham sido executados testes com o valor de `min threshold` 4, que é o padrão. Mesmo assim foram executados novos testes com esse valor de parâmetro, pois as condições do laboratório variam de acordo com eventos sazonais. Os tempos de execução de cada cenário são os mesmos já listados na Tabela 4.2.

Seria inviável, por motivos de tempo, testar diferentes valores de `min threshold` para cada valor de `compaction window size` testado, em cada cenário, com três valores diferentes de TTL. Por isso, foram testados apenas os pontos ótimos de tamanho de janela temporal, variando o TTL. O cenário escolhido foi o de Leitura em 10%. Os valores utilizados como tamanho da janela temporal, são os listados na segunda linha da Tabela 4.7.

Tabela 4.8: Minimum Threshold - Pontos Ótimos para Leitura 10%.

TTL 1h	TTL 2h	TTL 3h
6	8	10

Embora tenham sido executados testes apenas no cenário de Leitura 10%, os valores de `min threshold` são adotados em todas as simulações que tiverem o mesmo TTL definido. Trabalhos futuros poderão detalhar o comportamento do parâmetro em cada cenário de proporção de leitura.

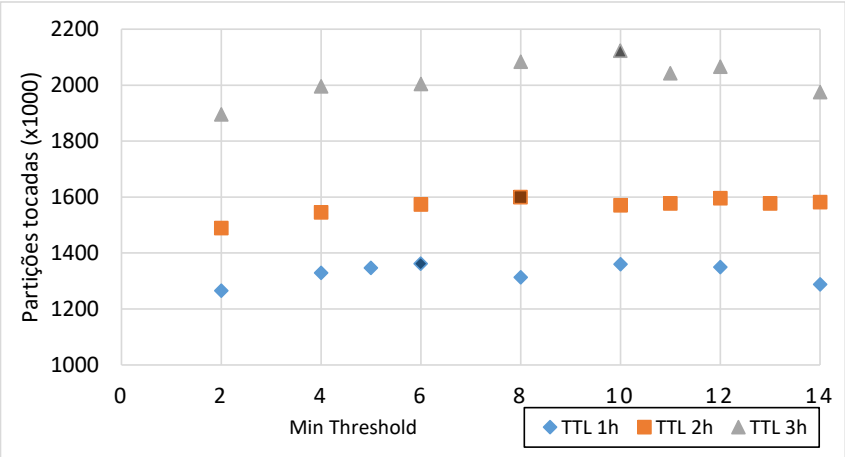

Figura 4.12: Número total de Partições Tocadas, por Min Threshold.

A Figura 4.12 mostra os resultados de números de partições tocadas, variando-se os valores de `min threshold`. Cada ponto no gráfico é o resultado de uma execução. Os pontos máximos tiveram sua cor destacada. Percebe-se que o valor padrão de `min threshold` não é o valor ótimo em nenhum dos valores de TTL.

As operações de compactação oriundas da STCS são intensivas quanto ao I/O, mas em compensação elas reorganizam as páginas de dados do Cassandra. Portanto, seu valor não pode ser muito pequeno, senão o funcionamento do sistema será interrompido, com muita frequência, por uma operação de compactação. Por outro lado, também não pode demorar muito para ocorrer, senão há muita fragmentação dos dados, o que prejudica as operações de leitura.

Portanto, há valores para o parâmetro `min threshold`, que resultam no melhor *throughput* do ambiente. Estes valores são chamados pontos ótimos. Os pontos ótimos de `min threshold` foram sumarizados na Tabela 4.8 e foram utilizados na construção do C*DynaConf.

Capítulo 5

C*DynaConf

Este capítulo trata do mecanismo de *auto-tuning* C*DynaConf, construído com base nos pontos ótimos obtidos. Ele está organizado em três seções. A Seção 5.1 traz uma arquitetura abstrata do C*DynaConf, trazendo seus componentes lógicos e sua interação como Cassandra. A Seção 5.2 trata do funcionamento do programa, apresenta a métrica escolhida para identificar a proporção de operações de leitura e de escrita e ressalta limitações e possíveis melhorias do programa. A Seção 5.3 apresenta o cenário modelado para a avaliação do C*DynaConf. Por fim, a Seção 5.4 apresenta os resultados das execuções e analisa seus principais indicadores.

5.1 Arquitetura do C*DynaConf

O C*DynaConf é um software de *auto-tuning* baseado em regras preestabelecidas. Ele se utiliza de uma tabela que guarda os pontos ótimos de configuração já conhecidos previamente. Baseado nesses pontos e nos metadados obtidos do banco de dados, ele computa valores próximos do ótimo de parâmetros de configurações de compactação de dados no Cassandra e aplica estes parâmetros.

A arquitetura do C*DynaConf pode ser vista na Figura 5.1. O componente Buscador de Metadados é responsável por ler as configurações das famílias de colunas que estejam no Cassandra. Ele vai filtrar somente as que estão configuradas com a estratégia TWCS e repassar para o calculador de parâmetros. Os metadados recuperados são: `o compaction window size`, o `min threshold` e o TTL da família de colunas.

O componente Buscador de Métricas requisita ao Cassandra as métricas que permitem ao C*DynaConf identificar a proporção entre leitura e escrita para a família de colunas que está recebendo o *auto-tuning*.

O C*DynaConf possui um temporizador, que itera uma execução da rotina principal do programa a cada 30 segundos. Este valor foi escolhido porque não onera significativamente

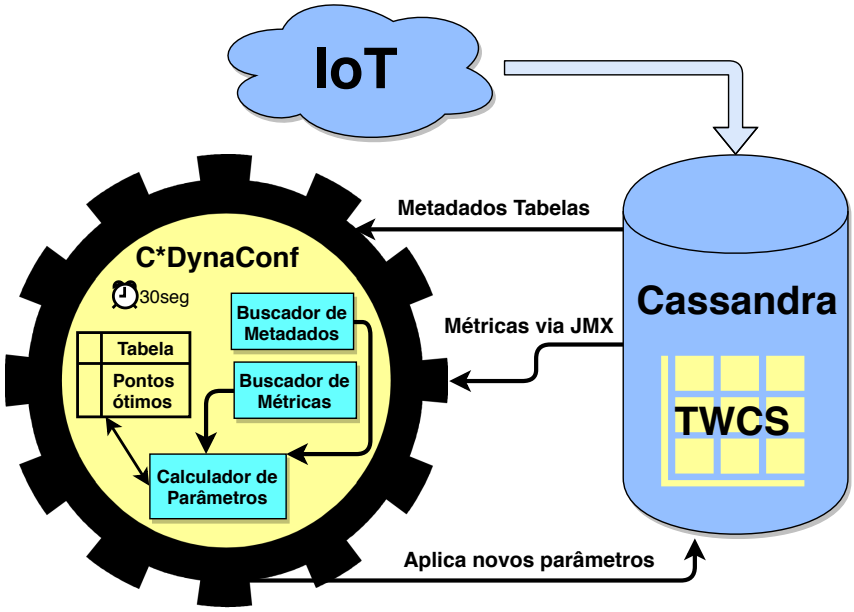

Figura 5.1: Diagrama da Arquitetura do C*DynaConf.

a performance do banco Cassandra que está recebendo o *auto-tuning* e recebe métricas em frequência suficiente para subsidiar uma melhor configuração.

O calculador de Parâmetros leva em consideração os metadados recebidos e as métricas, calcula os pontos ótimos de acordo com uma tabela de pontos ótimos de configuração preestabelecidos e os compara com os parâmetros configurados na tabela. Se, dado o cenário, a configuração já estiver ajustada com os pontos ótimos conhecidos, o programa não executa nenhum ajuste. Por outro lado, se o cenário não estiver com a sua respectiva configuração ótima, o programa envia uma alteração de tabela ao Cassandra, com os parâmetros novos.

5.2 Funcionamento do C*DynaConf

O C*DynaConf funciona somente sob a estratégia de compactação TWCS. Seu uso é indicado para otimizar o armazenamento de dados de IoT. O programa é recomendado para um ambiente Cassandra que:

- não receba grande número de operações de DELETE;
- os dados sejam inseridos com *Time to Live*, e esse TTL não varie muito entre os diferentes dados;
- não tenha grande frequência de operações de INSERT WITH TIMESTAMP, ou seja, na qual haja inserção de dados fora de ordem;

- não tenha a proporção de operações de leitura maior do que 30% nem menor do que 3% em relação ao total de operações, pois os pontos ótimos obtidos estão nesse intervalo;

- não tenha grande número de variações abruptas entre suas proporções de leitura e de escrita, ou seja, não altere muitas vezes em um curto período a quantidade média de consultas em relação à quantidade de INSERT.

O programa C*DynaConf foi desenvolvido em Java e utiliza alguns componentes que realizam tarefas que possibilitaram a construção do mecanismo de forma simplificada. O *driver* de conexão Java com o Cassandra fabricado pela DataStax®, que gerencia a conexão com os nós do *cluster* (DataStax, 2018a). Outro pacote usado é o Criteo Cassandra Exporter (Criteo, 2018), o qual gerencia o recebimento das métricas por meio de *Java Management Extensions* (JMX) e faz pré-computações para popular objetos com as métricas, que são definidas em arquivo de configuração.

O programa se inicia recebendo como parâmetro, via linha de comando, o *Keyspace* `k` que deve ser monitorado e o nome de algum nó do *cluster* `c`. Posteriormente, o C*DynaConf entra em um *loop* (que só é interrompido pelo usuário) no qual ele verifica se há famílias de colunas com a estratégia TWCS configurada e, caso existam, busca seus metadados, dos quais são usados o `compaction window size`, o `min threshold` e o TTL da tabela. Posteriormente, o mecanismo vai buscar as métricas de número de operações de leitura e de escrita, para definir a proporção entre elas. Em seguida, de posse do percentual de operações de leitura, o programa calcula de qual cenário de pontos ótimos, o ambiente em execução está mais próximo. Os valores ótimos de configuração são recuperados e comparados com os valores de metadados da tabela. Se estiverem diferentes, é aplicada uma alteração de tabela, por meio do comando ALTER TABLE, por meio do *driver* de conexão, feito durante a execução da massa de testes. Haveria alguma interrupção na carga se fosse alterada a estratégia de compactação mas como apenas os parâmetros são alterados, a alteração é processada rapidamente e as mudanças curtem efeito somente na próxima operação de compactação. Depois há uma pausa de 30 segundos e em seguida, todo o *loop* se repete até que o usuário interrompa seu funcionamento. Um fluxo do funcionamento pode ser visto na Figura 5.2.

O cálculo dos parâmetros é feito pela similaridade do cenário em execução em relação aos cenários previamente simulados. Se houver algum cenário muito diferente dos já simulados, a tendência é que não haja ganho relevante de desempenho. Em cenários muito distantes dos já simulados, pode haver perda de performance em relação a uma configuração manual.

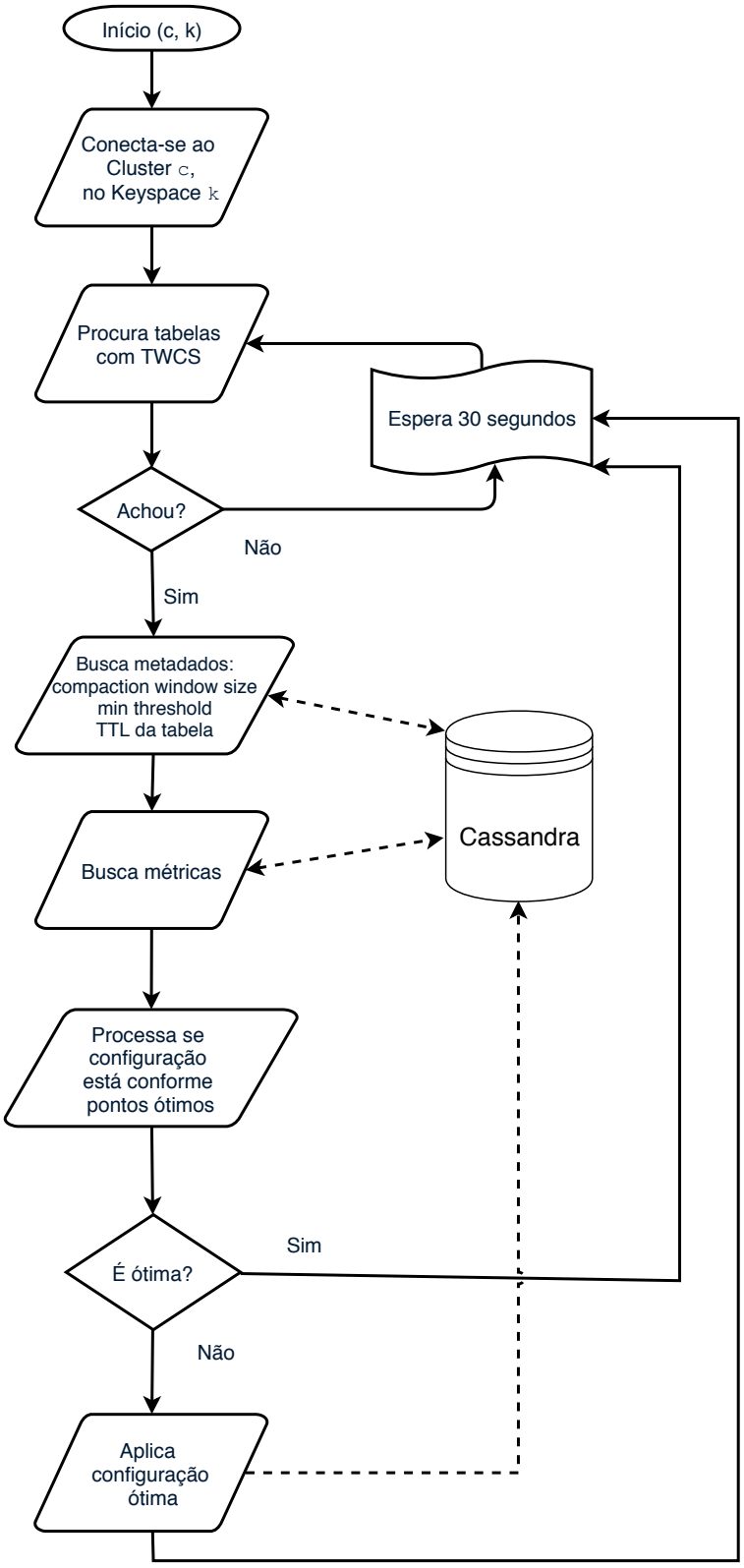

Figura 5.2: Diagrama de Fluxo de Dados do C*DynaConf.

5.2.1 Métrica de Proporção entre Leitura e Escrita

O C*DynaConf altera os parâmetros da TWCS de acordo com variações de TTL e de proporção entre operações de leitura e de escrita. O primeiro é facilmente recuperado nos metadados da tabela. Já a última não é definida em nenhuma estrutura do Cassandra e deve ser calculado baseado em métricas, obtidas dos nós do *cluster*.

Dentre as várias métricas que o Cassandra oferece, primeiro procurou-se a métrica que mostra o número de operações realizadas sobre cada tabela. Todavia, como o ambiente configurado possui fator de replicação 3, cada operação de escrita implica na escrita em três diferentes nós. Após análise dessa métrica, ela não se mostrou confiável, pois mesmo em um ambiente simulado, em que há garantias da proporção de operações de leitura e escrita, os números oscilaram bastante dentro de um mesmo nó e também havia valores díspares entre os nós. Os motivos para essas oscilações são os mecanismos de replicação das operações de escrita e de leitura. As leituras também podem ocorrer mais de uma vez para cada requisição, pois, quando um nó não responde em tempo hábil (*timeout*) o nó coordenador repassa a requisição de leitura para outro nó que possua a réplica do dado consultado.

Existe um conjunto de métricas que mede as requisições do cliente chamado *ClientRequest*. Dentro deste conjunto, há um objeto que guarda as métricas de leitura e de escrita. Trata-se de um objeto Java, de um tipo chamado *Meter*, que armazena o *throughput* instantâneo, do segundo em execução e também uma média móvel exponencialmente ponderada, que representa os valores médios dos últimos 1, 5 e 15 minutos (Apache, 2016a).

A média móvel exponencialmente ponderada é um indicador, originalmente criado na área de econometria, e posteriormente utilizado na indústria para controle de qualidade. Ela é calculada dando um peso maior às medidas mais recentemente observadas e peso menor às medidas tomadas há mais tempo (Hunter, 1986).

Estas métricas se adequaram perfeitamente ao desejado para o funcionamento do C*DynaConf, pois trazem um histórico dos últimos minutos, já sumarizado. Se o mecanismo fosse considerar o valor do *throughput* do instante em que ele vai buscar o metadado, o programa estaria sujeito a oscilações advindas das mais variadas fontes, como por exemplo, pausa da máquina virtual Java para *Garbage Collection*. Haveria mudanças muito frequentes de parâmetros de compactação, o que teria um custo alto.

Dentre os valores possíveis de médias móveis, de 1, 5 e 15 minutos, optou-se pela média de 5 minutos, por considerar esse valor suficiente para perceber mudanças nas características de um ambiente IoT. Entretanto, se o usuário optar por outro valor, estes valores são facilmente alteráveis no código-fonte da aplicação, desde que seja um dos três valores citados.

As métricas de leitura ClientRequest.Read.Latency.FiveMinuteRate[1] e de escrita ClientRequest.Write.Latency.FiveMinuteRate se mostraram confiáveis durante os experimentos, representando os valores de proporção conforme o definido na ferramenta Cassandra Stress Tool. Essas métricas estão presentes em todos os nós. Como elas são recuperadas somente a cada 30 segundos, o programa consulta todos os nós e soma as taxas de leitura e taxas de escrita e depois calcula o indicador de proporção de leitura. Se for um ambiente com número muito grande de nós, o programa pode ser alterado para consultar apenas alguns deles.

Há, entretanto, uma restrição causada pelo uso desta métrica. Ela é definida no nível do servidor, isto é, não é específica de uma determinada tabela ou *keyspace*. Isto significa que ela só poderá ser utilizada como indicador confiável se o banco Cassandra receber requisições somente para uso da tabela monitorada. Durante todos os testes realizados nesta pesquisa, o Cassandra Stress Tool só gerou carga de trabalho para uma única família de colunas por vez.

5.3 Cenário de IoT simulado

Para avaliar os efeitos do uso do C*DynaConf, foi modelado um cenário no qual as variáveis observadas pelo programa se alteram. Essas variáveis são o TTL e a proporção entre leitura e escrita.

A hipótese de trabalho (Subseção 1.1.1) supõe que o mecanismo de *auto-tuning* pode gerar um cenário com melhor desempenho em relação a uma configuração manual, ainda que esta configuração manual esteja em uma configuração ótima antes do ambiente sofrer mudanças. Portanto, a etapa inicial é configurada com os parâmetros ótimos. Em seguida, o ambiente varia suas condições de TTL e de proporção entre escrita e leitura. Ao final, as condições iniciais são retomadas. Assim, simula-se uma situação em que um usuário deixa o ambiente configurado otimamente, esse ambiente varia e, ao final, o ambiente retorna à condição inicial.

Este cenário cíclico ilustra que, mesmo que um sistema seja otimamente configurado, as condições podem variar e durante essas variações, a configuração deve variar também, em busca de otimizar o desempenho. O cenário ilustra uma situação em que um usuário administrador do Cassandra fez diversos testes para o cenário padrão de seu ambiente e configura o sistema com os pontos ótimos obtidos durante os testes. Contudo, ele não está presente durante todo o tempo, com disponibilidade para fazer ajustes no sistema assim que o cenário muda.

[1] Apesar de ter latência no nome, esta métrica mede o *throughput*.

No total, o cenário simulado possui quatro etapas, sendo que a primeira e a última etapa são iguais. Elas possuem TTL definido em 1h, e a proporção é de 90% escrita e 10% leitura. A segunda etapa possui TTL definido em 2h, e 30% das operações são de leitura. A terceira etapa possui TTL de 3h e 3% de leitura. Este cenário está ilustrado na Figura 5.3.

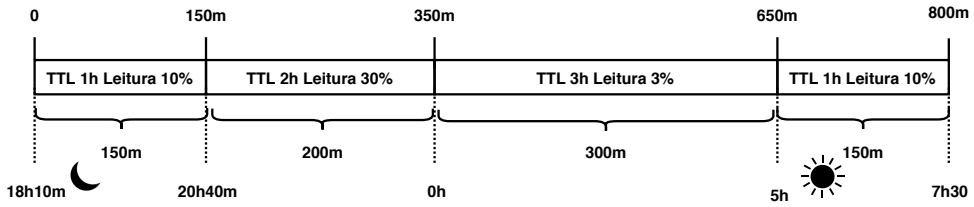

Figura 5.3: Cenário de teste do C*DynaConf.

Cada teste leva 800 minutos, ou 13 horas e 20 minutos para ser executado. Trabalhos futuros deverão modelar outros cenários, com diferentes valores de TTL e proporção entre leitura e escrita, para testar a eficiência do C*DynaConf.

Um exemplo que ilustra o cenário escolhido é o de uma cidade inteligente, com número fixo de sensores, à noite. A central de controle teria por configuração padrão o TTL de 1 hora e na primeira etapa as condições são ainda as mesmas do dia: as operações de leitura por parte das ferramentas de monitoramento e análises de dados feita por pessoas são responsáveis por 10% das operações.

Assim, é possível supor que o cenário modelado se inicie na central de controle da cidade inteligente às 19h10m. Na primeira etapa o TTL não precisa ser muito grande pois os operadores estão atentos, fazendo o monitoramento ativo dos dados, *in loco*, na central. A primeira etapa terminaria às 20h40m.

A segunda etapa necessitaria de um TTL um pouco maior, pois a equipe de operadores estaria reduzida e portanto os dados precisariam de um tempo de vida maior, para serem consultados pela equipe de reação a incidentes, antes de expirar. Além disso, nesta etapa, que se iniciaria às 21h30, os sistemas de suporte a decisão fariam diversas consultas ao banco de dados, para popular seus *Data Warehouses*, aumentando a proporção de leitura para 30%.

Na terceira etapa, que se iniciaria à 1h da madrugada, o monitoramento seria feito somente por computadores e as pessoas seriam acionadas somente em caso de algum incidente, em um regime de sobreaviso. Por isso, o TTL seria de 3h, para dar tempo da equipe de reação a incidentes ser avisada e se dirigir ao local antes dos dados expirarem. No período da madrugada as consultas a dados seriam exclusivamente feitas por ferramentas de monitoramento e por isso a proporção de leitura seria reduzida a 3%.

A quarta e última etapa se iniciaria às 6h da manhã e já teria as mesmas configurações dos períodos matutino e vespertino. O TTL seria de 1h e a leitura voltaria a ser de 10% pois os operadores voltariam a fazer consultas.

5.4 Análise dos Resultados

Os testes foram executados 3 vezes com a configuração estática, sem o uso do C*DynaConf, para mitigar oscilações advindas do ambiente computacional, uma vez que o mesmo não é isolado. A família de colunas foi criada com a configuração inicial definida como ótima, encontrados na Subseção 4.2.2, ou seja, o `compaction window size` foi definido em 10 minutos e o `min threshold` definido em 6. Posteriormente, os cenários dos testes foram alterados mas a configuração não foi alterada.

Os testes usando o C*DynaConf foram executados 5 vezes. Do mesmo modo que a execução estática, a tabela foi criada com os valores ótimos, levantados durante as execuções descritas no Capítulo 4.

A média foi tomada entre as 3 execuções do cenário estático, e outra média foi tomada entre as 5 execuções do cenário com o C*DynaConf sendo executado. A média entre as execuções é significativa, pois o desvio padrão do número de partições tocadas foi de menos de 2% da média, em ambos os casos, sendo de 1,58% com a configuração estática e 1,92% com a configuração dinâmica.

Ao longo de toda a execução, o cenário com o C*DynaConf tocou 4,52% mais partições do que o cenário estático, o que comprova a hipótese e vai ao encontro do objetivo de pesquisa.

Há que se considerar que, dos 800 minutos do cenário de execução, 300 são executados com a configuração ótima pela configuração estática. Isto porque, das quatro etapas, a primeira e a última, ambas com duração de 150m, são iguais e o cenário inicial está configurado de acordo com os pontos ótimos previamente simulados. A Tabela 5.1 mostra o número médio de partições tocadas entre as execuções com Configuração Estática e Dinâmica.

Se forem considerados apenas os cenários em que houve mudança de configuração, a melhoria chega a 9,12% em número de partições tocadas.

Num hipotético cenário de uma cidade inteligente sugerido na Seção 5.3, o atual ambiente de banco de dados poderia suportar uma quantidade de 403.046[2] sensores emitindo 5 observações por minuto, usando o C*DynaConf. Isso representaria um acréscimo de 17.443 na capacidade de sensores, utilizando um ambiente computacional igual ao dos testes.

[2] 5.971.059 / 800 * 60 * 0,9

Tabela 5.1: Média de Partições Tocadas por Etapa.

Etapa	Configuração	Estática	Dinâmica	Melhoria
1	TTL 1h, Leitura 10	1.247.113	1.230.392	-1,34%
2	TTL 2h, Leitura 30	1.278.766	1.386.875	8,45%
3	TTL 3h, Leitura 3	2.022.161	2.215.174	9,54%
4	TTL 1h, Leitura 10	1.164.596	1.138.618	-2,23%
	Total	5.712.636	5.971.059	4,52%

Como o número de partições tocadas foi maior utilizando o C*DynaConf, é de se esperar que o *throughput* também seja melhor com o uso do mecanismo. Para ilustrar como se comportou o *throughput* escolhemos uma execução de cada grupo, com o *auto-tuning* e sem ele. Escolheu-se a execução cujo *throughput* foi mais próximo da média. Como o *throughput* oscila muito abruptamente, utilizamos as médias móveis com período 5 para suavizar as curvas e exibimos na Figura 5.4.

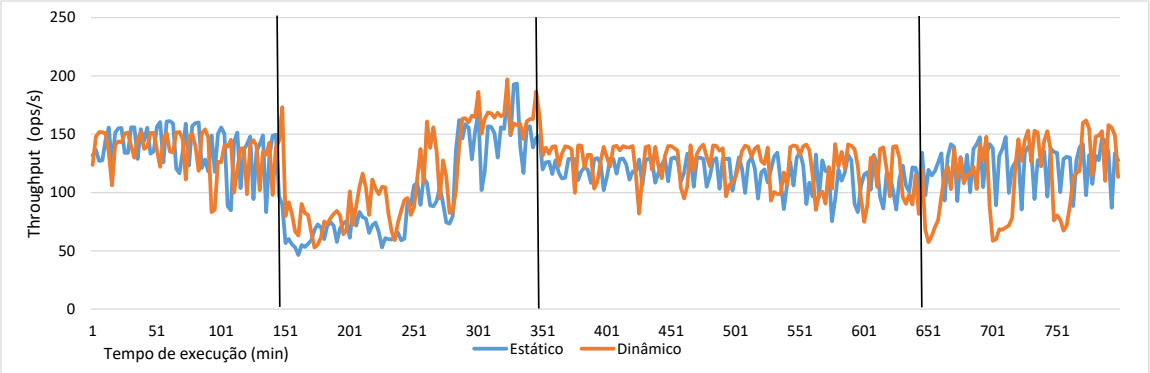

Figura 5.4: Médias Móveis com 5 Períodos do *Throughput*.

Na Figura 5.4 as etapas do cenário estão separadas por uma linha preta. Percebe-se que a execução com a configuração dinâmica tem *throughput* superior nas etapas 2 e 3, onde a diferença de configuração com os pontos ótimos de `compaction window size` e `min threshold` implicaram em melhor desempenho.

Para avaliar melhor as etapas do cenário que possuem configuração distinta de parâmetros, é apresentada uma seção das curvas da Figura 5.4 em outro gráfico na Figura 5.5. Nestas duas etapas percebeu-se uma melhoria de 9,3% no *throughput* da execução com uso do C*DynaConf.

Quanto à latência, a execução que contou com o C*DynaConf também apresentou melhora em relação à latência da execução com configuração estática. As latências de uma execução de cada configuração são apresentadas na Tabela 5.2. Nas execuções que têm o número de partições tocadas mais próximo da média, a latência da configuração dinâmica teve uma latência 4,09% menor.

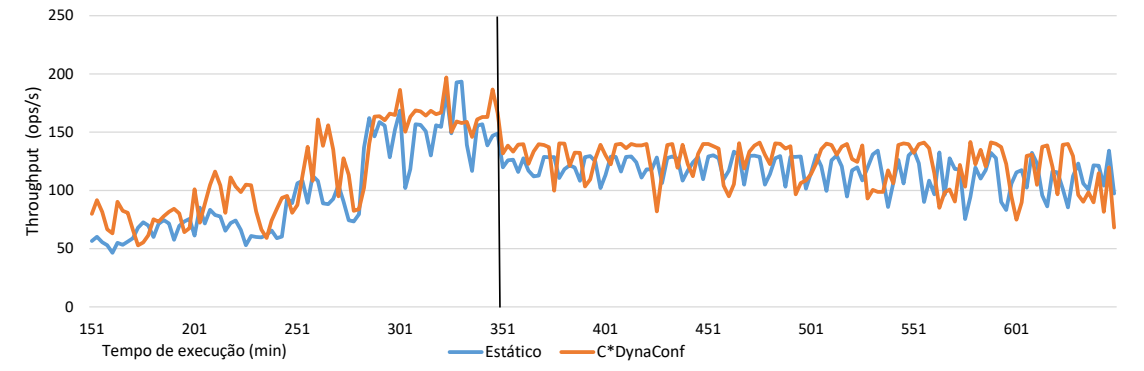

Figura 5.5: Médias Móveis do *Throughput* das Etapas com Parâmetros Diferentes.

Tabela 5.2: Latência Média (ms).

Etapa	Estática	Dinâmica	Melhoria
1	183,4	185,4	-1,09%
2	238,0	220,1	7,10%
3	189,4	172,1	9,13%
4	185,6	188,6	-1,62%
Total	199,7	191,6	4,09%

A latência também indica vantagem do uso do ambiente de configuração dinâmica, com a execução do C*DynaConf. Por outro lado, o mesmo não pode se dizer do espaço em disco, apresentado na Figura 5.6. O gráfico representa uma execução com configuração estática e uma execução com configuração dinâmica. As escolhidas foram as que apresentaram o número de partições tocadas mais próximo da média.

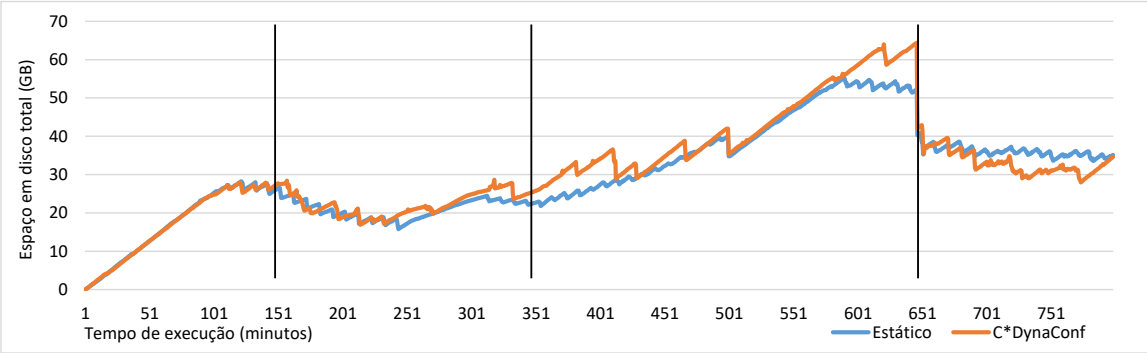

Figura 5.6: Total de Espaço em Disco Utilizado nos 10 Nós, em GigaBytes.

Na primeira etapa da execução, na qual as configurações estão iguais, o espaço cresce de maneira bem similar entre os dois cenários. Posteriormente, na etapa em que as operações de leitura passam a ser 30%, o espaço armazenado cai, à medida que os dados já inseridos vão se expirando e o *throughput* de inserção não consegue inserir dados na

mesma taxa do que os que já foram inseridos anteriormente e se expiram. Na terceira etapa, em que o TTL aumenta para 3h o volume em disco aumenta e atinge seu pico pouco antes da mudança de etapa. É também na terceira etapa que a configuração dinâmica utiliza significativamente mais espaço do que a configuração estática.

Tomando a média do espaço usado ao longo dos 800 minutos de execução, o teste com o C*DynaConf utilizou 3,6% a mais de espaço em disco do que o teste com a configuração estática. Entretanto, o pico de espaço usado, que pode ser considerado indicador mais relevante de espaço pois determina o espaço necessário, no ambiente dinâmico chegou a 16,7% a mais em relação ao ambiente estático.

Assim como na comparação entre as estratégias DTCS e TWCS (Seção 4.1.1), a configuração com melhor performance em relação ao *throughput* e tempo de resposta utilizou mais espaço em disco. Este tipo de permutação entre eficiência de espaço de armazenamento e desempenho de tempo de resposta é comum a diversos problemas na Ciência da Computação.

O C*DynaConf pode receber muitas melhorias, mas, para seu bom funcionamento no universo ao qual ele se propôs, destacamos duas.

O TTL atualmente é coletado somente nos metadados da tabela. Entretanto, o TTL pode ser definido também nas colunas e no momento da inserção dos dados, por meio do comando INSERT WITH TTL. Se os valores entre as colunas forem os mesmos, o programa pode ser facilmente adaptado para receber estes metadados. Contudo, se forem colocados valores distintos de TTL em diferentes colunas, a estratégia de compactação TWCS provavelmente não seria a mais adequada e o desempenho seria melhor utilizando outra estratégia, que não é suportada pelo programa.

Outra melhoria diz respeito aos cenários que não correspondem exatamente a nenhum cenário já previamente simulado mas que ficam entre os cenários, como por exemplo, um hipotético cenário que tenha o TTL de 1h30m. Atualmente o C*DynaConf o aproximaria para os dados ótimos de cenário com TTL de 1h (no caso de 1h31m seria aproximado para 2h). No entanto, se houver um maior número de simulações, pode-se utilizar algum método de regressão matemática nas curvas dos pontos ótimos, que permita aproximar os parâmetros ótimos.

Capítulo 6

Conclusão

Os ambientes de IoT podem gerar uma grande quantidade de dados e seu armazenamento é crítico para se obter algum valor desses dados. A taxa de criação dos dados pode variar ao longo do tempo, pois os ambientes de IoT, geralmente, são dinâmicos.

Bancos NoSQL são frequentemente usados para armazenar dados de IoT e o Cassandra é um dos mais indicados. Esta dissertação coaduna com esta indicação, apontando e testando a eficiência de estratégias de compactação de dados, próprias para os dados com características como os de IoT. Além disso, neste trabalho foi desenvolvido o C*DynaConf – um software que permite que o armazenamento de dados de IoT, no Cassandra, seja dinâmico e tenha suas configurações ajustadas automaticamente de acordo com certos atributos dos dados recebidos.

Para gerar o novo *software*, foi preciso analisar as estratégias de compactação existentes no Cassandra. A nova estratégia de compactação específica para o uso de séries temporais, TWCS, mostrou-se mais eficiente em termos de número de operações realizadas por segundo e tempo de resposta do que sua predecessora, a DTCS. Não obstante, esta última ainda deve ser usada em ambiente de IoT no qual o espaço em disco seja mais crítico do que o tempo de resposta.

Foram apresentados pontos ótimos de configuração da estratégia de compactação TWCS, obtidos durante a pesquisa. Os pontos ótimos podem ser utilizados por usuários como pontos de referência para fazer o *tuning* do SGBD NoSQL Cassandra. Além disso, foi apontado um conjunto de métricas que se mostraram indicadores chave de performance.

O C*DynaConf configura, em tempo de execução e sem intervenção manual, os parâmetros da TWCS. Em um cenário modelado, que já havia sido estudado previamente, obteve-se ganho de número de operações realizadas de 4,52%. Para cenários diferentes, o uso do mecanismo de *auto-tuning* deve ser monitorado, pois o comportamento não foi estudado.

O uso deve ser evitado também quando o limite de espaço em disco seja mais crítico do que o tempo de resposta, pois o uso do C*DynaConf aumentou a necessidade de espaço em disco, por 16,7% em relação ao cenário de configuração manual, estática.

Além do software objeto do trabalho, foram feitas outras contribuições às comunidades acadêmica e do Cassandra. Logo, foi implementada customização na ferramenta Cassandra Stress Tool que corrige o comportamento da geração de dados randômicos como massa de dados de teste. O comportamento não esperado no Cassandra insere dados repetidos, dependendo do tamanho e da composição da chave primária. A customização deve ser transformada em melhoria, pois o Cassandra é muito utilizado para armazenamento de séries temporais e muitos usuários passam por situações semelhantes com modelos de dados similares ao criado neste trabalho. Como trata-se de um problema que não gera erros acusados pelo sistema, é provável que muitos usuários nem saibam que o Cassandra Stress Tool tem um comportamento repetitivo na geração de dados dependendo das características de sua chave primária.

Durante este trabalho, foi alterado o *script* de inicialização do Cassandra para criar um novo diretório a cada vez que o SGBD se reiniciar. Para evitar criação com nomes repetidos, é acrescido ao nome padrão do diretório, um sufixo contendo o *timestamp* da inicialização. Esta contribuição já foi registrada na ferramenta gerenciadora de melhorias para o Cassandra.

Neste trabalho também foram desenvolvidos *scripts* de leitura de *logs*, que sumarizam métricas que não podem ser obtidas de outra forma no Cassandra. Além disso, já foi registrada sugestão de melhoria no *log* de compactação do Cassandra, que deve trazer mais informações.

Este trabalho teve o artigo "NoSQL Performance Tuning For IoT Data" pubicado nos anais do evento *3rd International Conference on Internet of Things, Big Data and Security*, que ocorreu em Funchal, Portugal (Dias *et al.* , 2018).

Como trabalhos futuros, os testes de *tuning* devem ser realizados em outro ambiente computacional, que esteja isolado de outras aplicações e tenha o hardware exclusivamente dedicado aos testes. Estes novos testes podem confirmar, total ou parcialmente, os resultados obtidos. Caso haja algum resultado diferente do obtido, o programa C*DynaConf deve ser reajustado com os novos pontos ótimos obtidos.

O Cassandra possui funcionalidades de compressão de dados, que envolvem algoritmos de compressão para economizar o espaço em disco. Todos os testes feitos neste trabalho foram executados com a compressão desabilitada, para não degradar o desempenho. Testes complementares devem ser feitos usando algumas opções de algoritmos de compressão para verificar o impacto disso no espaço em disco utilizado e no tempo de resposta.

Outras configurações de hardware também devem ser testadas para que sejam validados os resultados. Além de obter um ambiente físico diferente, é importante que haja a simulação em outras configurações do Cassandra, em especial estudando outros valores do fator de replicação, que neste trabalho ficou fixo em 3.

Outra possível melhoria seria o uso de um conjunto de dados reais de IoT. Os dados simulados utilizados neste trabalho procuraram refletir um ambiente real, contudo, o uso de dados gerados em um ambiente de produção deve ser feito para que se possa validar a eficácia e eficiência do C*DynaConf.

Outra melhoria relevante é estender a abrangência do C*DynaConf para as outras estratégias de compactação não deprecadas: STCS e LCS, avaliando seus parâmetros que possam influenciar no desempenho.

Por fim, as operações de compactação e geração de novas *SSTables* envolve muitas operações de disco, o que pode ser prejudicado pela fragmentação da *SSTable* – que é refletida no disco como um arquivo. Como trabalho futuro, o C*DynaConf poderia verificar junto aos nós, o grau de fragmentação, isto é, o número de fragmentos de arquivo e, caso haja alta fragmentação, emitir chamadas do sistema operacional para tentar desfragmentar os arquivos.

Referências

Abu-Elkheir, Mervat, Hayajneh, Mohammad, & Ali, Najah. 2013. Data Management for the Internet of Things: Design Primitives and Solution. *Sensors*, **13**(11), 15582–15612. 1, 8

Alapati, Sam R. 2018. Tuning Cassandra Performance. *Pages 371–422 of: Expert Apache Cassandra Administration*. Apress, Berkeley, CA. 45

Alves, Nathália de Meneses, & Freitas, André Lage. 2015 (Feb.). *Comparação entre Abordagens para Configuração Automática do Hadoop*. Mestrado, UFAL, Maceió. 26

Apache. 2016a. *Apache Cassandra Monitoring*. Disponível em `http://cassandra.apache.org/doc/latest/operating/metrics.html`. Acessado em 11 jun 2018. 60

Apache, Software Foundation. 2016b. *Cassandra-stress*. Disponível em `http://cassandra.apache.org/doc/latest/tools/cassandra_stress.html`. Acessado em 11 jun 2018. 35

Atzori, Luigi, Iera, Antonio, & Morabito, Giacomo. 2010. The Internet of Things: A survey. *Computer Networks*, **54**(15), 2787–2805. 1

Bhogal, J., & Choksi, I. 2015 (Mar.). Handling Big Data Using NoSQL. *Pages 393–398 of: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 2, 9, 10

Carpen-Amarie, Maria, Marlier, Patrick, Felber, Pascal, & Thomas, Gaël. 2015. A Performance Study of Java Garbage Collectors on Multicore Architectures. *Pages 20–29 of: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM '15. New York, NY, USA: ACM. 42

Carpenter, Jeff, & Hewitt, Eben. 2016. *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. "O'Reilly Media, Inc.". 2, 12, 18

Cassandra, Apache. 2018. *Apache Cassandra Compaction Documentation*. Disponível em `http://cassandra.apache.org/doc/latest/operating/compaction.html`. Acessado em 11 jun 2018. 45

Cecchinel, C., Jimenez, M., Mosser, S., & Riveill, M. 2014 (June). An Architecture to Support the Collection of Big Data in the Internet of Things. *Pages 442–449 of: 2014 IEEE World Congress on Services*. 2

Chang, Fay, Dean, Jeffrey, Ghemawat, Sanjay, Hsieh, Wilson C, Wallach, Deborah A, Burrows, Mike, Chandra, Tushar, Fikes, Andrew, & Gruber, Robert E. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, **26**(2), 4. 3, 16, 17, 26

Chebotko, Artem, Kashlev, Andrey, & Lu, Shiyong. 2015. A Big Data Modeling Methodology for Apache Cassandra. *Pages 238–245 of: 2015 IEEE International Congress on Big Data*. IEEE. 32

Criteo. 2018 (June). *cassandra_exporter: Apache Cassandra® metrics exporter for Prometheus*. Disponível em `https://github.com/criteo/cassandra_exporter`. Acessado em 11 jun 2018. 58

Datastax. 2018 (Apr.). *Configuring compaction | Apache Cassandra 3.0 | Apache Cassandra 3.0*. Disponível em `https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsConfigureCompaction.html`. Acessado em 11 jun 2018. 2, 19, 21

DataStax. 2018a. *Java Driver for Apache Cassandra - Home*. Disponível em `https://docs.datastax.com/en/developer/java-driver/3.5/`. Acessado em 11 jun 2018. 58

DataStax. 2018b. *Virtual nodes | Apache Cassandra 3.0 | Apache Cassandra 3.0*. Disponível em `https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeVnodesUsing.html`. Acessado em 11 jun 2018. 14, 15

DB-Engines. 2018a. *DB-Engines Ranking - popularity ranking of key-value stores*. Disponível em `https://db-engines.com/en/ranking/key-value+store`. Acessado em 11 jun 2018. 9

DB-Engines. 2018b. *DB-Engines Ranking - popularity ranking of wide column stores*. Disponível em `https://db-engines.com/en/ranking/wide+column+store`. Acessado em 11 jun 2018. 2, 12

DeCandia, Giuseppe, Hastorun, Deniz, Jampani, Madan, Kakulapati, Gunavardhan, Lakshman, Avinash, Pilchin, Alex, Sivasubramanian, Swaminathan, Vosshall, Peter, & Vogels, Werner. 2007. Dynamo: Amazon's Highly Available Key-value Store. *Pages 205–220 of: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. New York, NY, USA: ACM. 42

Dias, Lucas B. 2018 (Apr.). *Repositório GitHub mestrado: Dados experimentais gerados durante o Mestrado*. Disponível em `https://github.com/lucasbenevides/mestrado`. Acessado em 11 jun 2018. 31, 34, 35, 36, 46

Dias, Lucas B., Holanda, Maristela, Huacarpuma, Ruben C., & Jr, Rafael T. de Sousa. 2018 (May). NoSQL Database Performance Tuning for IoT Data - Cassandra Case Study. *Pages 277–284 of: Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security*. 7, 33, 68

Ellis, Jonathan, & Morishita, Yuki. 2012 (Nov.). *[CASSANDRA-3442] TTL histogram for sstable metadata*. Disponível em `https://issues.apache.org/jira/browse/CASSANDRA-3442`. Acessado em 11 jun 2018. 18

Eriksson, Marcus. 2014 (Nov.). *DateTieredCompactionStrategy: Compaction for Time Series Data.* Disponível em `http://www.datastax.com/dev/blog/datetieredcompactionstrategy`. Acessado em: 2017-02-10. 17, 19, 22, 23, 24, 27, 28

Filieri, Antonio, Hoffmann, Henry, & Maggio, Martina. 2014. Automated design of self-adaptive software with control-theoretical formal guarantees. *Pages 299–310 of: Proceedings of the 36th International Conference on Software Engineering.* ACM. 26

Fowler, Martin. 2012 (Jan.). *bliki: NosqlDefinition.* 8

Gessert, Felix. 2016 (Aug.). *NoSQL Databases: a Survey and Decision Guidance.* Disponível em `https://medium.baqend.com/nosql-databases-a-survey-and-decision-guidance-ea7823a822d`. Acessado em 11 jun 2018. 10

Ghosh, M., Gupta, I., Gupta, S., & Kumar, N. 2015 (June). Fast Compaction Algorithms for NoSQL Databases. *Pages 452–461 of: 2015 IEEE 35th International Conference on Distributed Computing Systems.* 3, 16, 19

Gubbi, Jayavardhana, Buyya, Rajkumar, Marusic, Slaven, & Palaniswami, Marimuthu. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, **29**(7), 1645–1660. 1, 6

Gurry, Mark, & Corrigan, Peter. 1996. *ORACLE performance tuning.* "O'Reilly Media, Inc.". 32

Hammer, Michael, & Chan, Arvola. 1976. Index Selection in a Self-adaptive Data Base Management System. *Pages 1–8 of: Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data.* SIGMOD '76. New York, NY, USA: ACM. 27

Hegerfors, Björn. 2014 (Dec.). *Date-Tiered Compaction in Apache Cassandra.* Disponível em `https://labs.spotify.com/2014/12/18/date-tiered-compaction/`. Acessado em 11 jun 2018. 3, 17, 19, 20, 21, 22, 24, 27, 28

Hunter, J. Stuart. 1986. The Exponentially Weighted Moving Average. *Journal of Quality Technology*, **18**(4), 203–210. 60

Jacobs, Adam. 2009. The pathologies of big data. *Communications of the ACM*, **52**(8), 36. 16

Jirsa, Jeff, & Eriksson, Marcus. 2016 (June). *Provide an alternative to DTCS - ASF JIRA. [CASSANDRA-9666].* Disponível em `https://issues.apache.org/jira/browse/CASSANDRA-9666`. Acessado em 11 jun 2018. 24, 25, 27, 28, 38

King, Jamie, & Berglund, Tim. 2015 (Oct.). *DataStax Academy: Data Modeling.* Disponível em `https://academy.datastax.com/resources/ds220-data-modeling`. Acessado em 11 jun 2018. 32

Kona, Srinand. 2016. *Compactions in Apache Cassandra : Performance Analysis of Compaction Strategies in Apache Cassandra.* Masters, Blekinge Institute of Technology, Karlskrona, Sweden. 3, 27, 28

Krishnan, Krish. 2013. *Data warehousing in the age of big data.* Newnes. 12

Lakshman, Avinash, & Malik, Prashant. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, **44**(2), 35–40. 2, 3, 15, 50

Laney, Douglas. 2001 (February). *3D Data Management: Controlling Data Volume, Velocity, and Variety.* Tech. rept. META Group. 1

Li, T., Liu, Y., Tian, Y., Shen, S., & Mao, W. 2012 (Nov.). A Storage Solution for Massive IoT Data Based on NoSQL. *Pages 50–57 of: 2012 IEEE International Conference on Green Computing and Communications (GreenCom).* 7

Lu, Bai, & Xiaohui, Yang. 2016. Research on Cassandra data compaction strategies for time-series data. *Journal of Computers*, **11**(6), 504–513. 2, 3, 19, 20, 21, 27, 28

Ma, M., Wang, P., & Chu, C. H. 2013. Data Management for Internet of Things: Challenges, Approaches and Opportunities. *Pages 1144–1151 of: Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing.* 1, 7, 8

NIST. 2003. *NIST/SEMATECH e-Handbook of Statistical Methods.* NIST. 49

Oliveira, Marcelo Iury S, Lóscio, Bernadette Farias, da Gama, Kiev Santos, & Saco, Fazenda. 2015. Análise de Desempenho de Catálogo de Produtores de Dados para Internet das Coisas baseado em SensorML e NoSQL. *In: XIV Workshop em Desempenho de Sistemas Computacionais e de Comunicação.* 9

Oliveira, Rafael, & Lifschitz, Sérgio. 2014. Sintonia fina baseada em ontologia: o caso de visoes materializadas. *In: Workshop de Teses e Dissertações em Banco de Dados (WTDBD).* Curitiba, PR, Brazil: PUC-Rio. 27

Oussous, Ahmed, Benjelloun, Fatima-Zahra, Lahcen, Ayoub Ait, & Belfkih, Samir. 2015. Comparison and classification of nosql databases for big data. *In: Proceedings of International Conference on Big Data, Cloud and Applications.* 2

Pagano, Marcello, & Gauvreau, Kimberlee. 2004. Princípios de bioestatística. *In: Princípios de bioestatística.* 41

Patterson, David A, & Hennessy, John L. 2013. *Computer organization and design: the hardware/software interface.* Newnes. 16

Perera, C., Zaslavsky, A., Christen, P., & Georgakopoulos, D. 2014. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys Tutorials*, **16**(1), 414–454. 1, 6

Philip Chen, C. L., & Zhang, Chun-Yang. 2014. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences*, **275**(Aug.), 314–347. 12

Ramaswamy, Lakshmish, Lawson, Victor, & Gogineni, Siva Venkat. 2013. Towards a quality-centric big data architecture for federated sensor services. *Pages 86–93 of: Big Data (BigData Congress), 2013 IEEE International Congress on*. IEEE. 1

Ramesh, D., Sinha, A., & Singh, S. 2016 (Mar.). Data modelling for discrete time series data using Cassandra and MongoDB. *Pages 598–601 of: 2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*. 7

Ravu, Venkata Sathya Sita J. S. 2016. *Compaction Strategies in Apache Cassandra : Analysis of Default Cassandra stress model*. Masters, Blekinge Institute of Technology, Karlskrona, Sweden. 27, 28

Sadalage, Pramod J, & Fowler, Martin. 2013. *NoSQL Essencial: Um guia conciso para o Mundo emergente da persistência poliglota*. Novatec Editora. 11

Santos, Bruno P., Silva, Lucas A. M., Celes, Clayson S. F. S., Borges Neto, João B., Vieira, Augusto M., Vieira, Luiz Filipe M., Goussevskaia, Olga N., & Loureiro, Antonio A. F. 2016. Internet das coisas: da teoria à prática. *In: Minicursos / XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. Salvador, BA: Sociedade Brasileira de Computacao. 1

Sathvik, Katam. 2016. *Performance Tuning of Big Data Platform : Cassandra Case Study*. Ph.D. thesis, Blekinge Institute of Technology, Faculty of Computing, Department of Communication Systems., Sweden. 27, 28

Sheldon, Robert. 2013 (July). *Columnstore Indexes in SQL Server 2012*. Disponível em `https://www.simple-talk.com/sql/database-administration/columnstore-indexes-in-sql-server-2012/`. Acessado em 11 jun 2017. 10

Srivastava, P. P., Goyal, S., & Kumar, A. 2015 (Oct.). Analysis of various NoSql database. *Pages 539–544 of: 2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*. 2, 9

Stonebraker, Mike, Abadi, Daniel J, Batkin, Adam, Chen, Xuedong, Cherniack, Mitch, Ferreira, Miguel, Lau, Edmond, Lin, Amerson, Madden, Sam, O'Neil, Elizabeth, *et al.* . 2005. C-store: a column-oriented DBMS. *Pages 553–564 of: Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 10

Strauch, Christof, & Kriha, Walter. 2011. NoSQL Databases. *Lecture Notes, Stuttgart Media University*. 8, 9, 10

Sullivan, David G., Seltzer, Margo I., & Pfeffer, Avi. 2004. Using Probabilistic Reasoning to Automate Software Tuning. *Pages 404–405 of: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '04/Performance '04. New York, NY, USA: ACM. 26

Tiwari, A., Chen, C., Chame, J., Hall, M., & Hollingsworth, J. K. 2009 (May). A scalable auto-tuning framework for compiler optimization. *Pages 1–12 of: 2009 IEEE International Symposium on Parallel Distributed Processing*. 26

Vongsingthong, Suwimon, & Smanchat, Sucha. 2015. A Review of Data Management in Internet of Things. *KKU Research Journal*, 215–240. 8

Waddington, Daniel G., & Lin, Changhui. 2016. A Fast Lightweight Time-Series Store for IoT Data. *arXiv:1605.01435 [cs]*, May. arXiv: 1605.01435. 7

Wakabayashi, Yoshiko. 2007. *Euler e as Origens da Teoria dos Grafos*. Disponível em `https://www.ime.usp.br/~yw/2016/grafinhos/aulas/Euler-yw-usp-2007.pdf`. Acessado em 11 jun 2018. 11

Xiong, W., Bei, Z., Xu, C., & Yu, Z. 2017. ATH: Auto-Tuning HBase's Configuration via Ensemble Learning. *IEEE Access*, **5**, 13157–13170. 28

Xu, L. D., He, W., & Li, S. 2014. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics*, **10**(4), 2233–2243. 1

Yeksigian, Carl, & Luciani, T Jake. 2016 (May). *[CASSANDRA-11865] Improve compaction logging details - ASF JIRA*. Disponível em `https://issues.apache.org/jira/browse/CASSANDRA-11865`. Acessado em 11 jun 2018. 46

Zaslavsky, Arkady, Perera, Charith, & Georgakopoulos, Dimitrios. 2013. Sensing as a service and big data. *arXiv preprint arXiv:1301.0159*. 1

Zhang, J., Iannucci, B., Hennessy, M., Gopal, K., Xiao, S., Kumar, S., Pfeffer, D., Aljedia, B., Ren, Y., Griss, M., Rosenberg, S., Cao, J., & Rowe, A. 2013 (June). Sensor Data as a Service – A Federated Platform for Mobile Data-centric Service Development and Sharing. *Pages 446–453 of: 2013 IEEE International Conference on Services Computing*. 1

Zhu, Sainan. 2015. *Creating a NoSQL database for the Internet of Things : Creating a key-value store on the SensibleThings platform*. Ph.D. thesis, Mid Sweden University, Faculty of Science, Technology and Media, Department of Information and Communication systems, Sundsvall, Sweden. 8

Öszu, M. Tamer, & Valduriez, Patrick. 2001. *Princípios de Sistemas de Bancos de Dados Distribuídos*. 2ª edn. Campus. 2