Aurélio Carlos Prado Silva

SQLStreamify - Middleware baseado em microsserviços fornecendo transações de leitura de fluxo de dados em SGBDs

Presidente Prudente - SP

Novembro/2021

Aurélio Carlos Prado Silva

SQLStreamify - Middleware baseado em microsserviços fornecendo transações de leitura de fluxo de dados em SGBDs

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, da Faculdade de Ciências e Tecnologia da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de Presidente Prudente.

Universidade Estadual Paulista - UNESP

Faculdade de Ciências e Tecnologia

Programa de Pós-Graduação em Ciência da Computação

Orientador: Ronaldo Celso Messias Correia

Presidente Prudente - SP

Novembro/2021

S586s

Silva, Aurélio Carlos Prado
SQLStreamify - Middleware baseado em microsserviços fornecendo transações de leitura de fluxo de dados em SGBDs / Aurélio Carlos Prado Silva. -- Presidente Prudente, 2022
96 p.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Faculdade de Ciências e Tecnologia, Presidente Prudente
Orientador: Ronaldo Celso Messias Correia

1. Banco de dados. 2. Fluxo de dados (Computadores). 3. Middleware. 4. Microsserviços. 5. Consultas ativas. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Faculdade de Ciências e Tecnologia, Presidente Prudente. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

UNIVERSIDADE ESTADUAL PAULISTA

Câmpus de Presidente Prudente

CERTIFICADO DE APROVAÇÃO

TÍTULO DA DISSERTAÇÃO: SQLStreamify - Middleware baseado em microsserviços fornecendo transações de leitura de fluxo de dados em SGBDs

AUTOR: AURÉLIO CARLOS PRADO SILVA

ORIENTADOR: RONALDO CELSO MESSIAS CORREIA

Aprovado como parte das exigências para obtenção do Título de Mestre em Ciência da Computação, área: Computação Aplicada pela Comissão Examinadora:

Prof. Dr. RONALDO CELSO MESSIAS CORREIA (Participaçao Virtual)
Matemática e Computação / FCT/UNESP - Câmpus Presidente Prudente

Prof. Dr. DANILO MEDEIROS ELER (Participaçao Virtual)
Departamento de Matemática e Computação / Faculdade de Ciências e Tecnologia de Presidente Prudente

Prof. Dr. ROBSON LEONARDO FERREIRA CORDEIRO (Participaçao Virtual)
Ciências de Computação / Universidade de São Paulo

Presidente Prudente, 20 de dezembro de 2021

À minha família que é a base de tudo.

Agradecimentos

Agradeço primeiramente à Deus pelos planos e caminhos que me foram dados.

Aos meus pais Aurélio e Nair, por todo apoio, educação e amor que sempre me deram.

Aos meus filhos Felipe e Henrique que, se não conseguiram entender o porquê dessa fase de muitos estudos, um dia irão entender. Meu esforço é sempre pensando em vocês. Ter que fazer silêncio em época de um "novo normal", dentro de casa e sem poderem sair. Eu sei que não foi fácil. Agradeço demais a ajuda de vocês. Obrigado pelo abraço apertado que enche de energia nos períodos mais difíceis, quando parecia que não daria certo.

Em especial à minha esposa Renata, que me ajudou a segurar a barra, cuidar de tudo, tudo mesmo. As horas e horas de revisão que me davam segurança e tranquilidade nas sugestões, mesmo não sendo da área. Pelas palavras de apoio, pelo carinho e incentivo sempre. Com toda certeza, sem você esse trabalho nunca seria possível. Te amo!

Ao professor Ronaldo pela oportunidade, pela orientação, paciência e liberdade durante o desenvolvimento desse trabalho.

À UNESP não só por toda minha formação acadêmica, mas também por grande parte da minha formação profissional. Universidade onde estudo, trabalho e onde meus filhos estudaram, tenho uma gratidão imensa por essa instituição.

Aos amigos de trabalho da Diretoria Técnica de Informática da UNESP.

E a todos que de alguma forma contribuíram nessa jornada.

Epígrafe

"A good algorithm is like a sharp knife: it does what it is supposed to do with a minimum amount of applied effort. Using the wrong algorithm to solve a problem is like trying to cut a steak with a screwdriver: you may eventually get a digestible result, but you will expend considerably more effort than necessary, and the result is unlikely to be aesthetically pleasing."

(Th. Cormen, Ch. Leiserson, R. Rivest, Introduction to Algorithm)

Resumo

O uso de informações como postagens em redes sociais, cotações de ativos financeiros, consumo de combustível do automóvel, informações de sensores de casas inteligentes, equipamentos de UTI, entre tantas outras geradas continuamente, leva a mudanças no armazenamento e processamento de dados. Técnicas de programação reativas, nas quais algoritmos reagem a eventos para disparar ações, têm sido cada vez mais utilizadas e aprimoradas por diversas linguagens e *frameworks* de desenvolvimento. Essa abordagem altera a maneira como os dados devem ser consultados; deve se monitorar constantemente alterações em bases de dados para disparar eventos e executar ações. Na busca por trabalhos relacionados ao tema, é possível verificar as demandas na área, buscando validar a utilidade do trabalho proposto, com trabalhos que resolvem o problema do consumo de fluxo de dados apenas para suas aplicações. São demonstrados também trabalhos que apresentam técnicas de buscas em fluxo de dados e formas de publicação dos mesmos. Apesar da existência de vários trabalhos foi constatada a demanda por uma solução completa para integração de funções de leitura de fluxo de dados em Tempo Real em sistemas com bancos existentes e em produção. O objetivo do trabalho é projetar e implementar um *middleware*, com arquitetura baseada em microsserviços, capaz de disponibilizar fluxos de dados em tempo real obtidos pela conversão de consultas em consultas ativas, sem a complexidade de um SGDF (Sistema Gerenciador de Dados em Fluxos) e com a possibilidade de execução sem mudanças em bancos de dados já em uso. A escolha por uma arquitetura baseada em microsserviços possibilitou que o trabalho fosse dividido e resolvido em partes que se comunicam para entregar a solução final. O método criado por contêineres atuando como servidores de replicação para detecção de mudanças nos dados se provou eficiente nos testes comparado com as soluções existentes, uma vez que não prejudica as operações no SGBD (Sistema Gerenciador de Banco de Dados) e apresentou tempos de inserções paralelas a execução de em média 2.10 vezes mais rápido do que técnicas existentes para a solução do problema. Além do que, o uso do MQTT (*Message Queue Telemetry Transport* - Transporte de Telemetria de Fila de Mensagens) se adequou perfeitamente como protocolo de entrega dos fluxos de dados.

Palavras-chave: banco de dados. fluxo de dados. *stream* de dados. tempo real. *middleware*. microsserviços. buscas contínuas. consultas ativas.

Abstract

Use of information such as social media posts, financial stock quotations, car fuel consumption, smart home sensors, ICU equipment, among many others, leads to changes in data storage and processing. Reactive programming techniques, in which algorithms react to triggering actions, have been increasingly used and improved by several programming languages and development frameworks. This approach changes the way data should be consulted, changes in databases must be constantly monitored to trigger events and perform actions. In the literature, it is possible to verify demands in this area, seeking to validate the usefulness of the proposed work. Once this demand for integrating real-time data stream reading functions in systems in production using existing DBMSs (Database Management System), the objective of this work is to design and implement a middleware, with microservice-based architecture, capable of providing real-time data streams obtained by converting queries into continuous queries, without all the DSMS (Data Stream Management System) tool complexity and with the possibility of execution without changes in legacy systems databases. The choice for a microservice-based architecture enabled the work to be divided and solved in parts that communicate in order to deliver the final solution. The method created by containers that act as replication servers for detecting changes in data proved to be efficient in tests when compared to existing solutions. Moreover, employment of MQTT (Message Queue Telemetry Transport) protocol fit perfectly in the delivery of data streams.

Keywords: databases. data stream. real time. middleware. microservices. continuous query.

Lista de ilustrações

Lista de quadros

Lista de códigos

Lista de tabelas

Lista de abreviaturas e siglas

BD	Banco de Dados
BDTR	Banco de Dados de Tempo Real
CQL	*Continuous Query Language* - Linguagem de Consulta Contínua
DBMS	*Database Management System* - Sistema Gerenciador de Bando de Dados
DSMS	*Data Stream Management System* - Sistema Gerenciador de Fluxo de Dados
HTTP	*Hypertext Transfer Protocol* - Protocolo de Transferência de Hipertexto
IoT	*Internet of Things* - Internet das coisas
JSON	*Javascript Object Notation* - Notação de objetos Javascript
M2M	*Machine-to-machine* - Máquina a máquina
Mb	megabyte
MQTT	*Message Queue Telemetry Transport* - Transporte de Telemetria de Fila de Mensagens
SGBD	Sistema Gerenciador de Banco de Dados
SGBD-TR	Sistema Gerenciador de Banco de Dados em Tempo Real
SGDF	Sistema Gerenciador de Dados em Fluxos
SNMP	*Simple Network Management Protocol* - Protocolo Simples de Gerência de Rede
SPE	*Stream Processing Engine* - Motores de Processamento de Fluxo
SQL	Structured Query Language
STR	Sistema de Tempo Real
XML	*Extensible Markup Language* - Linguagem de marcação extensível
YAML	*YAML Ain't Markup Language* - YAML não é uma linguagem de marcação

Sumário

1 Introdução

Têm sido cada vez mais comuns situações nas quais se faz o uso de informações no mesmo instante em que são geradas, dentre elas uma postagem em rede social, checagem de cotações de ativos financeiros, consumo de combustível do automóvel, sensores de monitoramento de casas inteligentes, equipamentos de UTI, entre tantas outras nas quais, na maioria dos casos, só tem valor naquela ocasião. Em todos esses casos, as aplicações envolvidas com essas informações são consideradas Sistemas em Tempo Real (STRs), que Kopetz (2011) define como "um sistema que deve informar o estado atual do objeto controlado e deve apoiar seu operador no controle desse objeto" (Feng et al., 2018; PARK et al., 2019; ISAH; ZULKERNINE, 2018).

Evoluções no modo como determinadas aplicações manipulam os dados criaram essa demanda por consumo de informações em tempo real - os *streams* de dados (fluxo de dados) - que de forma contínua devem ser processados. Técnicas de programação reativas têm sido cada vez mais utilizadas e aprimoradas por diversas linguagens e *frameworks* de desenvolvimento, nas quais algoritmos processam um fluxo contínuo de dados e reagem a eventos - alterações nos estados dos objetos controlados (ISAH; ZULKERNINE, 2018).

Um Sistema Gerenciador de Banco de Dados em Tempo Real (SGBD-TR) é definido por Bestavros, Lin e Son (2012) como uma fusão de um SGBD (Sistema Gerenciador de Banco de Dados) e um STR. Sendo assim, qualquer SGBD que armazene dados de um STR é classificado como Banco de Dados em Tempo Real. Além disso, existem diversos SGBDs criados exclusivamente para essa finalidade, com funções específicas para armazenamento e recuperação de fluxo de dados, consultas temporais, entre outras, gerenciando para os dados serem entregues ou armazenados em seus prazos corretos. Entre eles estão os SGDFs (Sistemas Gerenciadores de Dados em Fluxo), criados especificamente para o tratamento de fluxos de dados em Tempo Real, que utilizam o conceito de consultas ativas, isto é, consultas que permanecem produzindo resultados em tempo real conforme os dados são incluídos ou alterados (BAPTISTA, 2014).

Entretanto, existem situações nas quais projetos de sistemas de informação, já em produção, são ampliados e passam a demandar funções reativas com leitura de dados em fluxos resultantes de consultas ativas. Sistemas legados que em sua concepção não contemplavam tais funções, projetados utilizando SGBDs convencionais e não SGDFs. Alterações em sistemas já em execução ou em fase avançada de projeto, dificilmente são viáveis devido ao custo de migração do SGBD por outro que atenda aos novos requisitos por se tratarem de funções pontuais, por questões de licenciamento, de treinamento, de tecnologias ou infraestrutura. Há também casos de sistemas que necessitam apenas de

algumas funções de um SGDF, com poucas consultas ativas e não todas suas funcionalidades, sendo assim, nestes casos não se justifica a troca dos seus SGBDs (PARK et al., 2019; Feng et al., 2018; FERNANDES et al., 2014; HELMY; ZANFALY; OTHMAN, 2009; CALBIMONTE et al., 2012).

Nesse contexto, soluções que em suas configurações mais básicas exigem muitos recursos por entregarem além das funções de buscas, as de escrita, manipulação de prazos, entre outras, tornando o sistema complexo, podendo ter um custo maior que os recursos computacionais que a organização disponha para atender seus sistemas. Os SGDFs mais conhecidos tratam operações em fluxos de dados e se mostram eficazes nessa tarefa, porém um dos requisitos principais que é a necessidade da geração de fluxos de dados a partir de dados armazenados em SGBDs dificilmente são encontrados (HIRAMAN; M.; C., 2018; BEZ, 2015; ALBUQUERQUE, 2019; XIN, 2018; ÇETINTEMEL et al., 2016).

Considerando a inexistência de uma solução completa para integração de funções de manipulação de dados em fluxos em sistemas com bancos já existentes, constatou-se a necessidade de especificar, projetar e implementar o *SQLStreamify*, um *middleware* com arquitetura baseada em microsserviços, com o objetivo de disponibilizar fluxos de dados contínuos em tempo real obtidos por consultas em bases de dados existentes. O *SQLStreamify* tem como proposta ser um middleware que integra, ao SGBD, fluxos de dados gerados por consultas SQL em dados constantemente inseridos.

A arquitetura de microsserviços em contêineres, que podem ser executados tanto em servidores locais como em *clouds* e em número variado de instâncias, resulta em um baixo custo computacional para entregar a solução completa para um baixo número de buscas e altamente escalável com a possibilidade de aumentar recursos conforme a demanda. É importante ressaltar que estes estudos integrados não são encontrados na literatura e, portanto, configuram uma contribuição original.

A criação de um método que utiliza contêineres como servidores de replicação, realizando a leitura do log binário para detecção de alterações nos dados, se mostrou extremamente eficiente no consumo de recursos e no tempo de execução. Esse fator possibilita a geração de fluxos de dados a partir das alterações pelas consultas do *SQLStreamify*.

1.1 Objetivo Geral

O objetivo deste trabalho é especificar, projetar e implementar o *SQLStreamify*, um *middleware* com arquitetura baseada em microsserviços, capaz de disponibilizar consultas ativas em SGBDs existentes (que não contemplam o tratamento de fluxos de dados) obtidas pela conversão de consultas tradicionais, gerenciando para enviar aos clientes todas as atualizações que houverem nos dados que correspondam a consulta, gerando assim fluxos de dados contínuos em tempo real.

1.2 Objetivos Específicos

- Apresentar e avaliar por meio de testes as melhores alternativas para obtenção de alterações de dados em SGBDs existentes e implementar alternativas que não bloqueiem o SGBD ao realizar a busca;
- Permitir por meio da modularidade da arquitetura baseada em microsserviços que o *SQLStreamify* suporte futuros módulos com conexão para outros SGBDs ou com outras formas de publicação de fluxo de dados;
- Projetar e implementar o *SQLStreamify* utilizando uma arquitetura baseada em microsserviços em contêineres;
- Realizar testes com medições de recursos e desempenho do *SQLStreamify* em situações diversas e desenvolver aplicações modelo que demonstrem o funcionamento com uso de dados simulados;
- Disponibilizar o projeto por meio de repositório público com licença de código aberto.

1.3 Organização do Texto

Este trabalho está organizado da seguinte maneira: no Capítulo 2 são apresentados conceitos fundamentais e trabalhos relacionados, que servem como base para compreensão do funcionamento do *SQLStreamify*. No Capítulo 3 o desenvolvimento do trabalho, com escolha das tecnologias, motivações, tomadas de decisões e soluções encontradas para entrega do *middleware*. No Capítulo 4 são realizados testes e avaliações de desempenho. No Capítulo 5 são apresentadas duas aplicações práticas desenvolvidas utilizando o *SQLStreamify* na solução. Resultados e discussões sobre o desenvolvimento são comentados no Capítulo 6.

2 Revisão Bibliográfica

Este capítulo trata em detalhes a base de temas e conceitos sobre a qual a pesquisa desta dissertação foi realizada. Conceitos básicos utilizados na definição e desenvolvimento do projeto e as tecnologias utilizadas, possibilitando o entendimento de trabalhos relacionados apresentados. São introduzidos conceitos de Fluxos de Dados, Sistemas em Tempo Real, Consultas Ativas, Processamento de Eventos, Programação Reativa, Sistemas Gerenciadores de Fluxos de Dados, Arquitetura baseada em microsserviços e protocolo MQTT (*Message Queue Telemetry Transport*).

Além dos conceitos básicos, são apresentados trabalhos relacionados ao tema de Sistemas Gerenciadores de Dados em Fluxo. Desde Terry et al. (1992) que define o conceito de Consultas Ativas, trabalhos que demonstram a necessidade de funções de tratamento de dados em fluxos em sistemas legados que utilizam SGBDs, até pesquisas com os principais SGDFs, uma comparação entre eles, detalhando seus objetivos, arquiteturas e principais funcionalidades.

São apresentados também trabalhos com a utilização de logs binários de SGBDs tanto para aprimoramento de consultas, como para análise forense e entre outros usos.

2.1 Fluxo ou *Stream* de dados

Dados podem ser gerados em grandes volumes e continuamente por meio de várias atividades, como por exemplo as curtidas, postagens e visualizações em uma rede social, logs de servidores, monitoramento de tráfego de rede. Esse tipo de dados que são gerados continuamente são considerados Fluxos ou *Streams* de Dados. Muitos desses dados perdem sua relevância se não forem processados imediatamente, surgindo assim aplicações com funções de análises de dados em tempo real (HIRAMAN; M.; C., 2018; SILBERSCHATZ et al., 2010).

A análise de fluxo de dados é um paradigma de programação desenvolvido para incorporar dados continuamente em algum processo de decisão. É útil na identificação de *insights* perecíveis que requeiram ações imediatas ou com certas restrições de tempo (ISAH; ZULKERNINE, 2018).

Alguns exemplos de fluxos de dados e suas aplicações consumindo os dados em tempo real são apresentados por Silberschatz et al. (2010):

- Mercado financeiro: no mercado financeiro cada negociação é representada por uma tupla, que são enviadas como *stream* para os sistemas de processamento. Sistemas

de análise buscam por padrões para tomadas de decisão de compra ou venda com requisitos de tempo real na casa de segundos, sendo que sistemas mais potentes não aceitam atrasos maiores que microssegundos para terem um tempo de resposta melhor que outros que busquem pelos mesmos padrões.

- Comércio eletrônico: em sistemas de comércio eletrônicos muitos fluxos de dados podem ser gerados, como buscas realizadas no site e compras efetuadas, entre outras. A análise desses dados em tempo real auxilia em controle dos estoques, controle de performance das campanhas de publicidade, picos de vendas de produtos específicos e monitoramento de atividade dos clientes.

- Sensores: são utilizados nos mais diversos setores, desde veículos, construções, indústrias, cidades inteligentes e em muitos dispositivos conectados. Esses sensores enviam seus dados periodicamente em forma de fluxo e a leitura desses é usada para monitorar a saúde do sistema. Leituras com valores fora de um padrão específico do sistema merecem ações que devem ser tomadas, como alarmes que devem ser acionados com o menor atraso possível. Dependendo da complexidade do sistema, a frequência da leitura dos dados é alterada e o fluxo pode chegar a ter um grande volume de dados, com dados de inúmeros sensores chegando normalmente a uma central que processa essas informações.

- Dados de redes de computadores: organizações com uma grande rede de computadores devem ser capazes de monitorar as atividades em rede a fim de detectar problemas, bem como ataques em sua rede. Os dados trocados, ou uma agregação desses dados podem ser representados como um fluxo de dados. Esse deve ser processado para detectar problemas, por exemplo, um tráfego excessivo a determinada porta de um computador pode significar um ataque de negação de serviço entre vários outros problemas que podem ser detectados por meio de padrões encontrados nestes fluxos.

- Redes sociais: redes sociais como *Facebook* e *Twitter* têm um fluxo contínuo de dados com mensagens em posts ou *tweets* de seus usuários. Cada mensagem deve ser roteada para seus amigos ou seguidores. Esses fluxos podem ser consumidos não só por humanos, mas também por *software*, por exemplo, empresas que monitoram esses dados para publicidade, campanhas ou estudos que detectam os mais diversos padrões dentro desses dados.

2.1.1 Consumo de Fluxo de Dados

Em sistemas que utilizam o consumo de *stream* (fluxo) de dados, não é viável aplicar o método tradicional, de armazenar e analisar posteriormente, pois neste cenário tem um conjunto de dados em constante mudança e em uma análise tardia esses dados poderiam ter perdido sua relevância (ROSA, 2017).

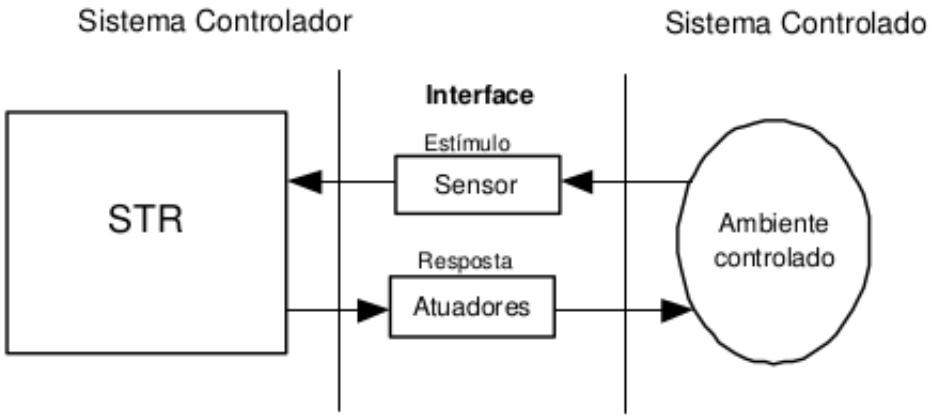
A pesquisa na área de processamento de fluxo de dados pode ser dividida em três áreas (ZHAO et al., 2017):

- Sistemas de gestão de fluxo de dados, nos quais as linguagens de consultas são exploradas (SGDF - descrito na seção 2.6);
- Algoritmos de processamento de dados online cujo objetivo é processar os dados numa única passagem;
- Plataformas de *stream processing* (processamento de fluxo de dados) que permitem a implementação e escalabilidade de aplicativos baseados em processamento de fluxo de dados.

2.2 Sistemas em Tempo Real

FERNANDES et al. (2005) ilustram o cenário de um STR, em dois sistemas: um controlador e um controlado, Figura 1. O Sistema Controlado é o ambiente que disponibiliza dados para o Sistema Controlador, no caso o STR. Os dados podem ser obtidos por sensores ou uma rede de sensores.

Figura 1 – STR: Sistema Controlador e Sistema Controlado



Fonte: FERNANDES et al. (2005)

Buttazzo (2011) define Sistemas em Tempo Real (STR) como sistemas computacionais que precisam reagir com precisão de tempo a eventos do ambiente. Como consequência, o comportamento correto do sistema depende não só do resultado computacional, mas também do tempo para produzir tais resultados.

> O termo "real" em STR indica que a reação do sistema a eventos externos deve ocorrer durante a sua evolução, assim o "tempo" do sis-

> tema deve ser medido usando a mesma escala de tempo do ambiente externo(BUTTAZZO, 2011).

O conceito de Tempo Real, em STR, frequentemente é usado de maneira incorreta, consideram STR um sistema capaz de reagir rapidamente a eventos externos. Rapidez porém, é muito relativa e não traduz corretamente as principais características desse tipo de sistemas(BUTTAZZO, 2011). Ou seja, um STR deve ser previsível ao invés de ser rápido. A diferença de um STR para um sistema não tempo real é caracterizada por um prazo, que é o tempo máximo com que suas tarefas devem ser executadas.

Em aplicações críticas, um resultado produzido depois do seu prazo não apenas é atrasado como também é errado. Dependendo das consequências que podem ocorrer por conta de um prazo expirado, Buttazzo (2011) classifica STRs em três categorias:

- *Hard*: uma tarefa é considerada *hard* se os resultados produzidos depois do prazo podem causar consequências catastróficas no sistema sob controle, como por exemplo em um sistema de monitoramento de equipamentos hospitalares;
- *Firm*: uma tarefa em tempo real é dita *firm* se os resultados produzidos depois de expirados o prazo são inúteis para o sistema, mas não causam nenhum dano. Um exemplo são sistemas de automação de linhas de montagens, qua um atraso pode resultar na montagem inadequada de uma peça, sendo que esse resultado pode ser previamente esperado;
- *Soft*: uma tarefa em tempo real é *soft*, quando os resultados produzidos depois do prazo ainda têm alguma utilidade para o sistema apesar de causarem degradação na performance do sistema, como exemplo podemos ter sistemas de som de computadores.

2.3 Consultas Ativas

As transações de busca que têm como resultado um fluxo de dados contínuo são conhecidas como Consultas Ativas (*Continuous Query*), são persistentes não tendo um resultado final, mas um retorno contínuo. Os dados inseridos ou atualizados na base de dados e que coincidam com a busca são entregues. Diferente da execução das buscas estáticas, retornando um *set* de dados de uma só vez de acordo com os dados já armazenados no Banco (HELMY; ZANFALY; OTHMAN, 2009; SILBERSCHATZ et al., 2010).

Dados armazenados em bancos são comumente chamados de *data-at-rest* (dados em descanso), em contraste aos *streams* de dados que conceitualmente nunca terminam. Buscas que só podem mostrar seu resultado depois de visualizar todas as tuplas de um

stream nunca seriam capazes de resultar uma saída. Por exemplo, uma busca pelo número de tuplas em um *stream* nunca teria um resultado final (SILBERSCHATZ et al., 2010).

Uma opção é retornar os resultados que satisfaçam a busca até um determinado ponto no *stream* e atualizar sempre que mais tuplas sejam atualizadas e satisfaçam a busca. No mesmo exemplo da busca pelo número de tuplas em um *stream*, a resposta seria dada até um instante em particular e atualizada conforme esse resultado fosse alterado.

As operações de Consultas Ativas devem ser implementadas usando técnicas de *pipeline*, que consistem em entregar os resultados em fluxos sem bloquear as operações de escrita nestes dados (SILBERSCHATZ et al., 2010).

Existem duas abordagens mais utilizadas para fornecer ao utilizador a capacidade de realizar consultas ativas. A primeira é a definição de uma extensão da linguagem SQL(*Structured Query Language*) como a CQL - *Continuous Query Language* e a segunda é a definição de operadores aplicáveis aos fluxos de dados de forma a produzir o resultado esperado. No *SQLStreamify* é utilizada a linguagem SQL para a criação das Consultas Ativas, sem a necessidade de uma curva de aprendizado de nova linguagem, juntamente com a seleção do modo de publicação da consulta que são apresentados na subseção 3.4.1 (HEBRAIL, 2008).

2.3.1 Consultas Ativas em Banco de Dados apenas com inserções

> Em um banco de dados no qual os dados são continuamente inseridos, usuários podem desejar realizar uma busca permanente e serem notificados sempre que os dados combinarem com essa busca (TERRY et al., 1992).

Com esse conceito Terry et al. (1992) apresentaram o termo *Continuous Query* em um artigo assinado pelo Centro de Pesquisa da Xerox Corporation, no qual descrevem métodos para a criação de Consultas Ativas em bancos de dados apenas com inserções de dados. Foi desenvolvido e incorporado a um sistema chamado *Tapestry* para filtros de fluxos de documentos eletrônicos como mensagens de email ou artigos de notícias.

O sistema *Tapestry* utiliza o *Sybase*, um banco de dados relacional comercial com suporte a linguagem SQL. E como uma primeira abordagem para a implementação de Consultas Ativas neste banco de dados consta a execução periódica de uma *query* SQL em intervalos determinados de tempo (Código 2.1).

Código 2.1 – Execução periódica de *query*

```
FOREVER DO
        Execute Query Q
        Return results to user
        Sleep for some period of time
ENDLOOP
```

Essa abordagem, apesar da sua simples implementação, tem três principais deficiências:

- Resultado não-determinísticos: os registros selecionados pela query dependem de quando a query é executada. Uma query executada de hora em hora pode produzir um resultado diferente de uma mesma query executada a cada dia, ou mesmo de hora em hora, mas disparadas em diferentes momentos;
- Duplicações: a cada execução da query, o usuário terá como retorno todos os registros, inclusive os antigos;
- Ineficiência: executar a mesma query diversas vezes é extremamente custoso e o tamanho do retorno tende a ser cada vez maior (lembrando que trata-se de um artigo de 1992), uma vez que o banco é de escrita apenas (append-only). Idealmente, o custo de execução de uma *Continuous Query* deve ser em função da quantidade de novos dados e não em função do tamanho de banco de dados inteiro.

O foco de Terry et al. (1992) passa então a solução do problema de resultados não-determinísticos. Considerando a consulta: "selecionar as mensagens que não obtiveram respostas", quando uma mensagem é adicionada ao banco de dados, ela corresponde a consulta. Contudo, uma vez que a mensagem é respondida, ela deixa de corresponder a mesma consulta. Se uma mensagem, em particular, chega ao banco às 8:15h e é respondida às 8:45h, ela pode não retornar por um sistema que execute o algoritmo do Código 2.1, sendo executado de hora em hora, mas pode retornar a um sistema que execute o mesmo algoritmo mas que começa a ser executado na metade de uma hora (considerando que a mensagem iria corresponder a consulta as 8:30h).

> Isso levanta a questão geral: Qual a semântica de uma consulta executada continuamente? Em outras palavras, quais garantias podem ser dadas aos usuários sobre o conjunto de resultados retornados por uma Consulta Ativa? (TERRY et al., 1992)

Terry et al. (1992) definem o resultado de uma Consulta Ativa como um conjunto de dados que seriam retornados se uma consulta fosse executada a todo instante.

Para ser preciso, considerando $Q(t)$ como o conjunto de resultados retornados pela execução da query Q em um banco de dados no momento de tempo t. Isto é $Q(t)$ é o resultado de executar Q no momento t e considerando $Q_m(t)$ como o total de resultados retornados até o tempo t executando a consulta Q como uma Consulta Ativa:

$$Q_m(t) = \bigcup_{s \leq t} Q(s)$$

Quando a consulta Q é executada, ela retorna $Q_m(t)$ e não $Q(t)$.

As Consultas Ativas são qualitativamente diferentes de consultas tradicionais. Considerando o exemplo dado da consulta sobre mensagens não respondidas, em sua formulação obvia todas as mensagens ao serem adicionadas ao banco de dados ainda não estariam respondidas, consequentemente todas as mensagens adicionadas ao banco de dados deveriam retornar a consulta contínua. O "não respondidas" para a Consulta Ativa está semanticamente desnecessário em um sistema que as mensagens são adicionadas sempre sem respostas. Isso ilustra o ponto de que nem todas as consultas a banco de dados são possíveis semanticamente de serem convertidas em Consultas Ativas.

Certamente executar uma *query* em todo instante de tempo não é possível, e se fosse possível, não seria prático, e o artigo propõe técnicas para prover Continuidade Semântica de maneira eficiente, melhorando o algoritmo inicial para algo como o Código 2.2.

Código 2.2 – Execução de Consulta Ativa

```
Set π = −∞
FOREVER DO
        set t:= current_time
        Execute Query Q^i(π, t)
        Return results to user
        set π := t
        Sleep for some period of time
ENDLOOP
```

No Código 2.2 tem-se a execução de $Q^i(\pi, t)$ que representa uma consulta incremental, parametrizadas por dois instantes de tempo, um inicial e um final, retornando apenas os dados que combinem com a consulta neste intervalo de tempo.

O artigo apresenta uma base teórica interessante para a implementação de Consulta Ativa, uma vez que demonstra a concepção da ideia e consegue explanar com detalhes e com exemplos os conceitos e definição de semânticas, envolvendo uma questão até então inédita. No desenvolvimento prático da solução não obtiveram resultados esperados de performance na execução de consultas incrementais. Na medição de tempo da consulta tradicional com relação à incremental os tempos obtidos foram relativamente semelhantes e em algumas ocasiões mais custosos. Tendo em vista os bancos disponíveis na época, sem suporte a *triggers* (gatilhos, nos quais se executada alguma ação leva a execução de uma ação ou um conjunto de ações), e com inúmeras limitações de recursos computacionais, resultou em um trabalho interessante com definições de conceitos utilizados até os dias atuais. Terry et al. (1992) definem como trabalhos interessantes para o futuro, a criação de continuous query em consultas com relacionamento entre tabelas, uma vez que o desenvolvimento só foi de consultas sem relacionamento, mas que atendeu as necessidades propostas e desejadas para o sistema *Tapestry* e a utilização de *triggers* na implementação de Consulta Ativa.

2.3.2 Necessidade de Consultas Ativas em aplicações com SGBDs convencionais

Diversos trabalhos apresentam necessidade de consumo de fluxos de dados obtidos de um BDTR por consultas ativas. Nesta seção são apresentados alguns trabalhos com o desenvolvimento das soluções utilizadas, dos mais variados segmentos, desde controle de dispositivos IoT até monitoramento de sondas de produção de petróleo e gerenciamento de UTIs hospitalares que utilizam SGBDs convencionais e desenvolvem soluções pontuais para a geração de consultas ativas.

Costa, Leite e Neto (2008) apresentam o desenvolvimento de um sistema autônomo que utiliza fluxos de dados obtidos de uma rede de sensores para o monitoramento de sondas de perfuração de poços de petróleo. São relatadas dificuldades de soluções que atendam de forma eficiente restrições temporais do sistema.

A aplicação utiliza aspectos de Sistemas em Tempo Real e manipula os dados armazenando-os em um SGBD convencional (*Postgresql* - SGBD relacional de código aberto), gerenciando e obedecendo às restrições de tempo tudo pela própria aplicação. Os dados armazenados podem ser analisados de forma gráfica, além de definir possíveis status das sondas.

Os autores consideram três pontos críticos do sistema, nos quais funções de Tempo Real devem ter seus prazos respeitados:

1. Porta serial para a captura de dados: não deve existir atraso na captura, uma vez que atrasos neste ponto ocasionam perda de dados;

2. Análise gráfica em Tempo Real: nesta análise são utilizados os últimos dados como principal alvo para detecção de problemas e foram utilizadas transações com prazos para a geração dos gráficos, tratando-os como um fluxo de dados obtidos por consultas no SGBD;

3. Status das sondas: restrições temporais periódicas são dadas como forma de entender os estados atuais das sondas, podendo ser utilizado não só para identificação de problemas, mas também para detecção de operações que estão ocorrendo nos poços.

Nos resultados do trabalho foram citados fatores como custo, eficiência, desempenho e taxa de perda de dados como satisfatórias, comparado com aplicações anteriores que eram utilizadas para essa função. Um dos aspectos que o autor identifica que poderia ser melhorado é a qualidade nos sistemas de controle, garantindo maior desempenho em pontos críticos do sistema e suas restrições de prazos (COSTA; LEITE; NETO, 2008).

Se utilizada uma solução que, ao determinar as restrições de tempo, retornasse os dados em fluxos, o desenvolvedor poderia focar na solução do seu problema e no

desenvolvimento de sua aplicação, sem a preocupação da geração e tratamento de fluxos de dados. Sistemas nos quais SGBDs atendam necessidades do desenvolvedor e que tenham, entre as suas funções, o consumo de um fluxo de dados armazenado neste banco, são exatamente o foco do *SQLStreamify*.

Outro estudo de caso consiste em um sistema que alerte os profissionais da saúde quando um paciente se encontra em um estado crítico e que forneça imediatamente estatísticas completas e atuais sobre o estado do paciente. É composto por diversos equipamentos, com sensores gerando dados em fluxo continuamente, que devem ser armazenados e manipulados para geração dos alertas.

Nos aspectos de um Sistema em Tempo Real, há o Sistema Controlador composto por um Computador/Servidor e pelas interfaces humanas (Profissionais da Saúde), enquanto o Sistema Controlado representa o ambiente, no caso a UTI hospitalar (FERNANDES et al., 2014; LINDSTRÖM, 2007).

Não são apresentados quais bancos e tecnologias foram utilizados, mas levam a entender que técnicas de Consultas Ativas foram utilizadas em um SGBD convencional, tratando dos requisitos de prazos das transações utilizando um Gerenciador de Transações, com escalonamento das transações concorrentes e um Gerenciador de Cache com a finalidade de tratar fluxos de dados, movendo os dados para a memória principal para agilizar as consultas e posteriormente armazená-los em memória secundária (FERNANDES et al., 2014).

Teve como resultado positivo o uso das transações com fluxos de dados, garantindo a comunicação eficiente entre o sistema com monitoramento e alertas em tempo real e os profissionais de saúde. Mais um trabalho em que o autor se preocupa com a implementação de funções de tempo real em bancos de dados convencionais, no qual o foco da aplicação é preterido para garantir o funcionamento de transações com fluxos de dados.

Foram verificados também, trabalhos (HUACARPUMA et al., 2016; FREMANTLE, 2015) que apresentam soluções para o gerenciamento de dados em fluxos gerados por dispositivos IoT (*Internet of Things* - Internet das coisas). Uma característica comum é que nestes casos os dados são gerados continuamente e devem ser processados e armazenados.

Huacarpuma et al. (2016) propõem um sistema de controle e integração dos mais variados tipos de dispositivos IoT conectados. Também citam o grande aumento do número e a variedade de dispositivos conectados à Internet, tendo como resultado um aumento no volume de dados gerados. Em seu trabalho, Huacarpuma et al. (2016) especificam uma proposta para o gerenciamento de dados em um ambiente de IoT, contribuindo com a especificação de funcionalidades e a concepção de técnicas para coletar, filtrar e armazenar dados em fluxos de forma eficiente.

Fremantle (2015) propõe o WSO2, um *framework* completo de código aberto para

o gerenciamento de dispositivos IoT conectados. Nesta proposta está incluído um módulo para a coleta de fluxos de dados por lotes e em tempo real.

2.4 Processamento de Eventos

O modelo de processamento em uma plataforma que utiliza dados em fluxos é uma das mais importantes características e vai determinar as possíveis operações e limitações. Em SGDFs são possíveis dois modelos distintos, o Processamento em Fluxo (*Stream Processing*) e o Processamento em Lotes (*Batch Processing*) cada um com vantagens e desvantagens (SHAHRIVARI, 2014).

No Processamento em Fluxo, também conhecido por Processamento Nativo, é aplicado o processamento uma tupla por vez. O dado é processado imediatamente ao ser inserido, o que permite uma maior fluidez, com menor latência que o modelo de Processamento em Lotes. Por outro lado a implementação desse modelo é mais trabalhosa e computacionalmente mais custosa (FERREIRA, 2016).

O Processamento em Lotes é feito por conjunto de tuplas aumentando a latência e reduzindo a fluidez. Por outro lado a complexidade é reduzida. Esse modelo se aplica em cenários nos quais uma grande coleção de dados é processada de uma vez (FERREIRA, 2016).

Há ainda o Processamento em Micro-lotes (*Micro-Batching*), no qual o fluxo de dados é processado em lotes com pequenas quantidade de dados. Tem-se assim uma latência relativamente reduzida em relação ao Processamento em Lotes, dependendo do tamanho dos lotes a serem processados (SHAHRIVARI, 2014).

O modelo de Processamento em Fluxo é também denominado de Processamento de Eventos uma vez que os dados conforme são recebidos “um a um” podem ser considerados eventos de um ambiente controlado. Esse modelo permite que as aplicações interajam com os dados de forma assíncrona diferentemente de aplicações que utilizam bancos de dados tradicionais de uma forma síncrona (os bancos de dados são entidades passivas que recebem requisições síncronas das aplicações) (BAPTISTA, 2014).

O Processamento de Eventos surgiu a partir de requisitos de domínios de aplicações que necessitam de um sistema que seja uma entidade ativa que dê suporte às suas atividades, como por exemplo sistemas de monitoramento, detecção de falhas, entre outros. Os sistemas de processamento de eventos recebem continuamente fluxos de dados, processam de forma contínua estes dados e enviam de forma assíncrona notificações para as partes interessadas.

Stonebraker, Çetintemel e Zdonik (2005) propuseram um conjunto de requisitos que devem estar presentes em Sistemas de Processamento de Eventos que se baseiam fundamentalmente em latência e escalabilidade:

- Manter os dados em movimento: manter a latência no mínimo possível, processando os dados assim que capturados;
- Capacidade para lidar com imperfeições nos streams;
- Utilizar SQL(*Structured Query Language*) em consultas de fluxos de dados;
- Capacidade de fornecer resultados garantidos e previsíveis;
- Capacidade de combinar diferentes fluxos ou fluxos de diferentes fontes externas com taxas de inserção variáveis;
- Recursos de tolerância a falhas de forma a garantir o processamento contínuo dos dados com o mínimo de perda possível;
- Possibilidade de distribuir o processamento de dados entre vários processadores/nós;
- Capacidade para obtenção de respostas em tempo real com sobrecarga mínima para um fluxo de dados de alto volume.

2.5 Programação Reativa

O termo Programação Reativa foi citado pela primeira vez por Berry (1989) no artigo *"Real Time Programming: special purpose or general purpose languages"*. É comumente definido como um paradigma de programação orientado a fluxo de dados e propagação de mudanças.

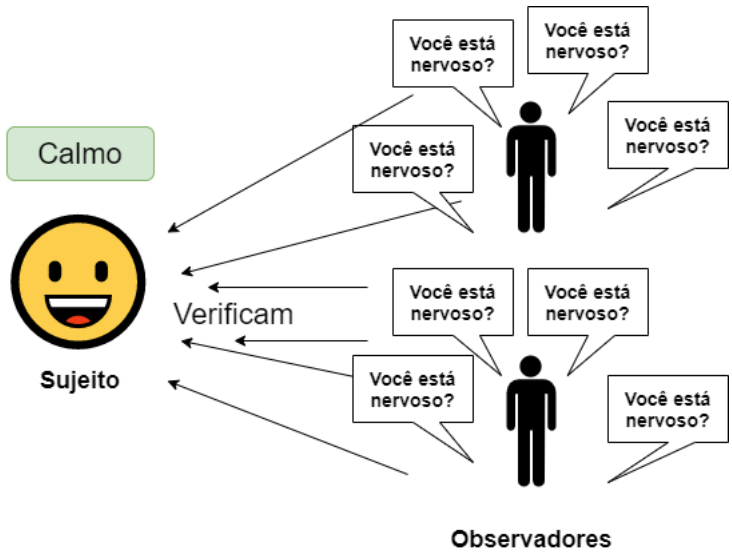
Propagação de mudanças remete a um padrão muito conhecido, o padrão do "observador". Esse padrão define uma dependência de um para muitos, onde se tem um "sujeito" e uma interface "observador", a qual pode ser implementada por muitos (WAN; HUDAK, 2000; MICHELSON, 2006; ROMANOV; TROSHINA, 2017).

Supomos uma situação, apresentada na Figura 2, na qual temos diferentes "observadores" consultando o estado de um "sujeito". Para resolver o problema sem o paradigma de Programação Reativa, usando uma linguagem puramente funcional, seria necessário realizar essa consulta constantemente, numa frequência que se julgue suficiente para que o atraso entre uma consulta e outra não prejudique a reação a tal mudança.

Explicando de forma ilustrativa a utilização de Programação Reativa, no exemplo apresentado na Figura 3 temos um "sujeito", o qual possui um estado ou informações que os "observadores" estão interessados. Os "observadores" por sua vez se registram para ouvir mudanças de estado ou notificações do "sujeito".

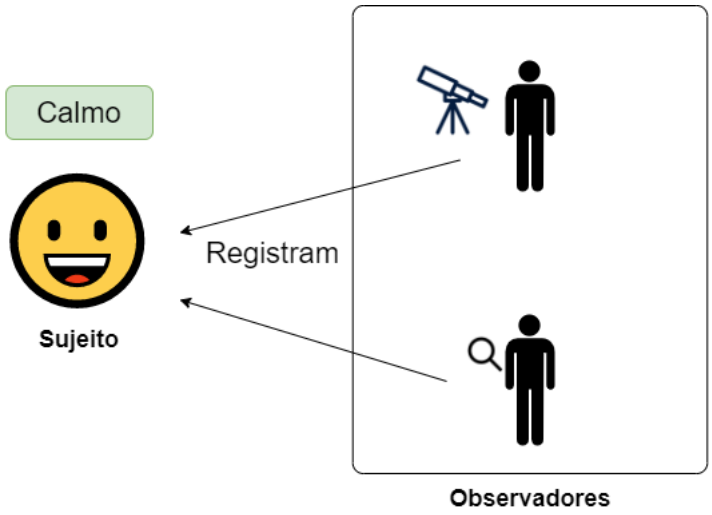
Assim que ocorre a mudança de estado do "sujeito", os "observadores" que se registraram para escutar essas alterações são notificados (Figura 4).

Figura 2 – Situação com Sujeito e Observadores, resolvendo utilizando programação Funcional



Fonte: Elaborado pelo autor

Figura 3 – Situação com Sujeito e Observadores, utilizando o paradigma de Programação Reativa. Observadores se registram

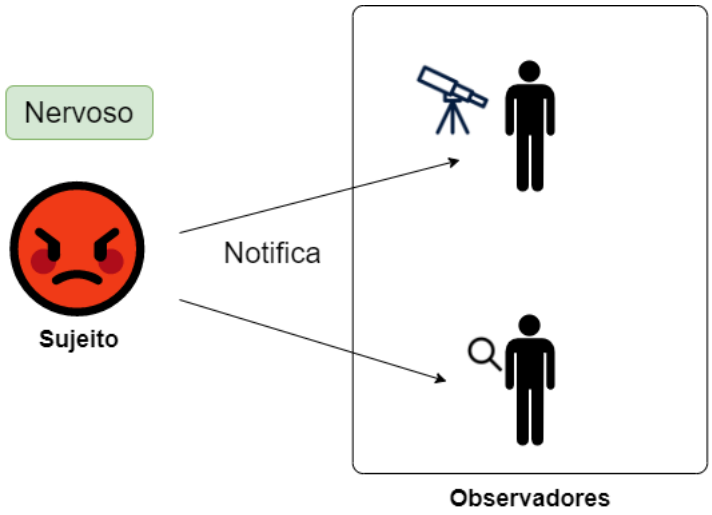


Fonte: Elaborado pelo autor

Quando notificados, os "observadores" atuam da forma que foram programados, realizando suas ações que dependem da alteração de estado do "sujeito" (Figura 5).

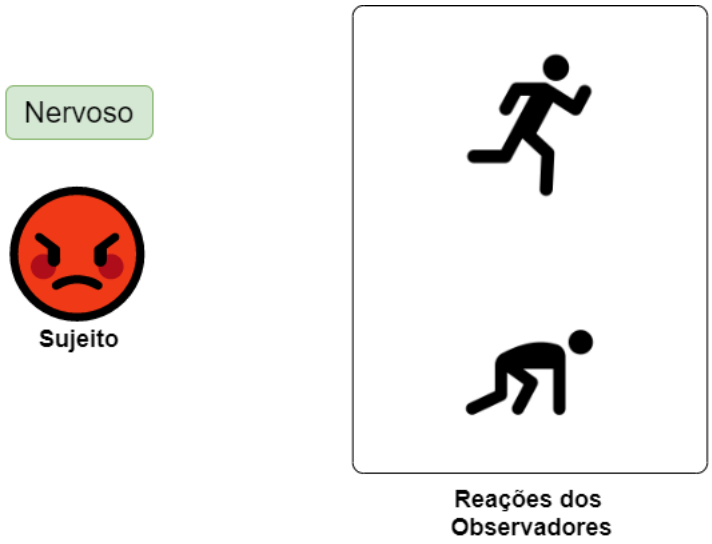
Em Programação Reativa existe ainda o termo "Observável" (*Observable*) que é quando o "sujeito" é definido como um recipiente que emite sinais para "observadores" ao longo do tempo. Em muitas definições e documentações de linguagens esses sinais emitidos pelos "observáveis" são considerados fluxos de dados, que são grande objeto de

Figura 4 – Situação com Sujeito e Observadores, utilizando o paradigma de Programação Reativa. Observadores são notificados da alteração do estado do Sujeito



Fonte: Elaborado pelo autor

Figura 5 – Situação com Sujeito e Observadores, utilizando o paradigma de Programação Reativa. Observadores reagem a alteração do estado assim que notificados



Fonte: Elaborado pelo autor

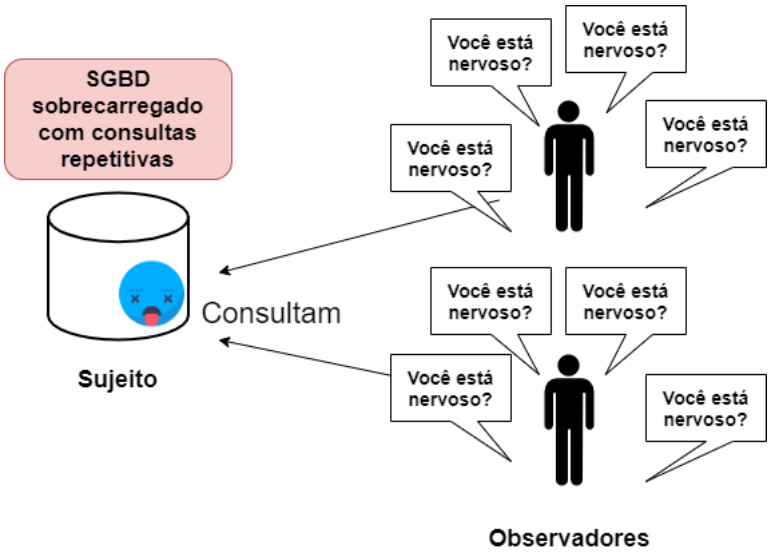
nosso pesquisa.

2.5.1 Programação Reativa em SGBDs

Na Figura 6 é exposta uma situação na qual o "sujeito" que é observado depende de consultas a um SGBD para verificação do seu estado. Isso é relativamente comum

de se deparar e muitas vezes é resolvido com constantes consultas que sobrecarregam o SGBD, tornando sistemas extremamente lentos e não responsivos. Baseado exatamente nessa situação que o *SQLStreamify* é proposto, criando "observáveis" através de consultas SQL em SGBDs evitando essa sobrecarga de constantes consultas, obtendo um fluxo de dados com as alterações de estado.

Figura 6 – Sujeito e Observadores - Banco de Dados utilizando programação funcional para identificação de alteração de estado



Fonte: Elaborado pelo autor

Zimmerle e Gama (2018) apresentam o crescimento do uso do conceito de Processamento de Eventos em paralelo ao de Programação Reativa como uma resposta ao crescimento de dados de diferentes tipos e a necessidade de um processamento em tempo (quase) real desses dados. Em seu trabalho, Zimmerle e Gama (2018) propõe operadores de SGDFs provendo extensões a uma plataforma *IoT* existente utilizando Programação Reativa no desenvolvimento.

O conceito de programação reativa deve ser entendido para possibilitar o entendimento das necessidades de diferentes tipos de algoritmos e situações que aplicativos passam a se deparar devido a constante atualização de dados. Esse é justamente um dos objetivos do trabalho, disponibilizar um meio de transformar uma consulta SQL, que tem origem em um modelo de programação síncrono em algo passível de observação, entregando as alterações de estado dos dados armazenados em um banco de dados existente.

Com o crescimento da demanda por sistemas orientados a eventos, surgiu a necessidade de adaptações em sistemas já existentes para que fossem implementadas neles funcionalidades reativas, conforme apresentado na subseção 2.3.2.

Grande parte de sistemas já existentes utilizam banco de dados relacionais para

armazenamento. Incluir funções reativas neles, como consultas ativas (seção 2.3) depende de ferramentas externas. Eyers et al. (2010) propõem em seu trabalho, um *middleware* que acrescenta funções de orientação a eventos no *PostgreSQL*, o *PostgreSQL-PS*. Em seu trabalho são apresentadas questões de entrega utilizando o paradigma *publisher/subscriber* (seção 2.9) desenvolvido por completo pelo autor. Os testes apresentados se baseiam em comparações da entrega das informações aos clientes, comparando a sistemas que utilizam o protocolo MQTT (seção 2.10).

Na pesquisa de Eyers et al. (2010) não foi deixado claro o método utilizado para a detecção de alterações dos dados. É apresentada uma linguagem extra utilizada como API para a criação de consultas aos eventos por aplicações que desejem consumir esses dados.

2.6 SGDF - Sistema Gerenciador de Dados em Fluxo

Os SGDFs - Sistemas Gerenciadores de Dados em Fluxo (*DSMS - Data Stream Management Systems*) têm foco no gerenciamento de fluxo de dados, diferentemente dos SGDBs - Sistemas Gerenciadores de Banco de Dados (*DBMS - Database Management Systems*) nos quais o objetivo é a gestão de base de dados. Os SGDFs são capazes de conectar-se a uma ou mais fontes de *streams* e são capazes de processar *queries* (consultas) contínuas aos dados, ou seja, que permanecem em execução, enquanto os dados em fluxo contínuo são transitórios. Em Çetintemel et al. (2016), Abadi et al. (2005) os SGDFs são denominados SPEs (Stream Processing Engines - Motores de Processamento de Fluxo).

A principal característica é que os dados produzidos por *streams* não são armazenados, mas sim processados *"on the fly"*. Esta última definição é exatamente o oposto das base de dados padrão (SGBDs), nos quais os dados são permanentes e as consultas transitórias (ROSA, 2017). No Quadro 1 são resumidas as principais diferenças entre os dos tipos de sistemas de gestão de dados.

Quadro 1 – Quadro comparativo SGBD e SGDF

SGBD	**SGDF**
Persistência de dados	Dados transitórios
Streams Finitos	Streams Infinitos
Acesso aleatório	Acesso sequencial
Queries transitórias	*Queries* permanentes
Plano de *queries* fixo	Plano de *queries* adaptável

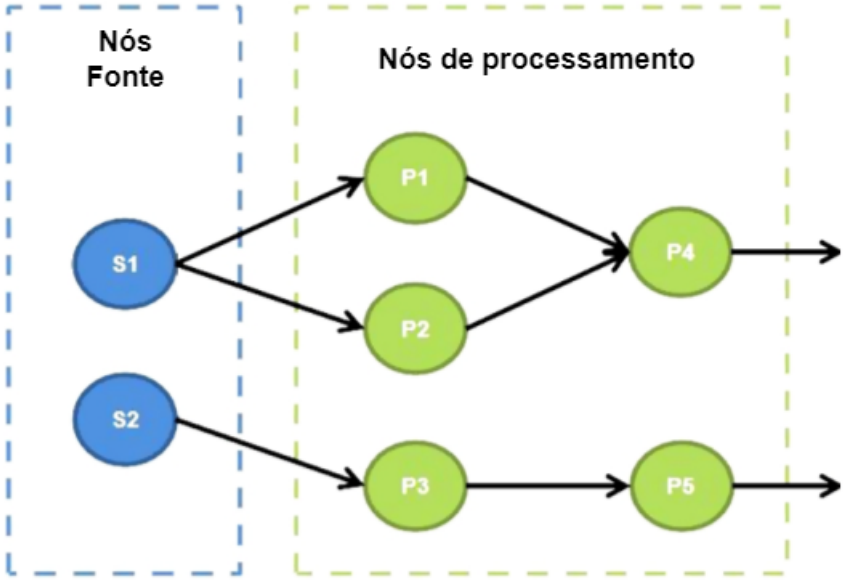
Fonte: Rosa (2017)

No geral, plataformas de *stream processing* são projetadas para funcionar em tecnologias de computação distribuídas e paralelas, como num *cluster*, de forma a alcançarem o processamento em tempo real de um grande volume de dados. O fluxo de dados é o

elemento central destas plataformas e refere-se ao fluxo de dados infinito que existe dentro de todo o sistema.

Tipicamente sistemas que trabalham com processamento de stream recebem esses dados de fontes externas, por exemplo, de sensores, de redes sociais e de base de dados, e entregam os resultados do processamento de volta para serem armazenados em uma base de dados ou para publicação dos mesmos num sistema online. Assim, os sistemas de processamento de *stream* baseiam-se em dois conceitos fundamentais, a fonte de dados externa e os nós de processamento, nos quais atividades de transformação aos dados são realizadas. Um modelo geral destes sistemas é apresentado na Figura 7 (ZHAO et al., 2017).

Figura 7 – Modelo geral de fluxo de dados de uma plataforma de *stream processing*



Fonte: Adaptado de Zhao et al. (2017)

Os nós geradores (S1 e S2) fornecem as tuplas (que tratando-se de fluxo de dados, são partições do mesmo) de um *stream* para as unidades de processamento (P1, P2, ...), que por sua vez os consomem, aplicam alguma transformação e emitem as tuplas para as fases seguintes. Cada unidade de processamento pode decidir não emitir nada, emitir tuplas de forma periódica, ou até emitir as tuplas após um certo número dos mesmos recebidos (ROSA, 2017).

Quando estes nós conectados na forma de um grafo são executadas em um *cluster*, cada unidade é distribuída paralelamente entre os nós de computação de forma a alcançar alto rendimento computacional. Existem duas propostas para essa distribuição: na primeira, existe um servidor central que orquestra a distribuição das unidades de

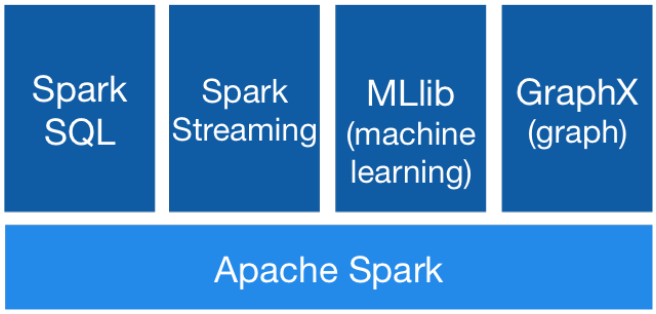
processamento para os diferentes nós físicos do *cluster*, sendo também o responsável pelo balanceamento das tarefas dos mesmos. Na outra forma, não existe nenhum nó responsável pela distribuição, funcionando como sistemas *peer-to-peer*, dividindo o trabalho igualmente (KAMBURUGAMUVE et al., 2013).

Nas subseções a seguir são abordados e comparados os principais SGDFs utilizados e os trabalhos acadêmicos a respeito. Estes são considerados o estado da arte no tratamento e gerenciamento de fluxos de dados. Por mais que o *SQLStreamify* não é considerado um SGDF, podemos comparar com outros SGDFs uma vez que realiza o tratamento, geração e distribuição de fluxos de dados a partir de dados inseridos continuamente em SGBDs.

2.6.1 Apache Spark Streaming

Apache Spark[1] é um framework idealizado por um grupo de pesquisadores da Universidade de Berkley com o objetivo de ser um modelo de uso geral de computação para Big Data em clusters. Ele fornece um conjunto de bibliotecas de alto nível em Java, Scala, Python e R. Conforme apresentado na Figura 8, o Apache Stark é dividido em módulos, tais quais o Spark SQL para trabalhar com dados estruturados, Mlib destinado aos algoritmos de Machine Learning, GraphX voltado ao processamento de grafos e Spark Streaming que facilita a construção de aplicações com dados em streaming. Todos esses módulos são otimizados para processamento paralelo, escalabilidade e tolerância a falhas. O modelo de computação do Spark se baseia no processamento de dados em memória principal. (BEZ, 2015; ALBUQUERQUE, 2019).

Figura 8 – Pilha de módulos do Apache Spark



Fonte: Bez (2015)

Dois módulos do Spark que buscamos para comparação ao trabalho são a habilidade de trabalhar com SQL para manipulação de dados pelo Spark SQL e o Spark Streaming para manipulação de fluxo de dados em tempo real.

[1] Apache Spark - <https://spark.apache.org/>

Em razão da popularidade do SQL, existe uma forte demanda pela incorporação de interfaces relacionais em frameworks para processamento de Big Data. Neste sentido, o Spark disponibiliza por meio da biblioteca Spark SQL a comunicação das aplicações com bases de dados utilizando SQL (XIN, 2018).

De acordo com Xin (2018) o Spark SQL foi projetado para atender os seguintes requisitos:

- API amigável para o programador com suporte ao processamento relacional;
- Alto desempenho utilizando técnicas já empregadas e bem estabelecidas em SGBDs;
- Suporte fácil a novas fontes de dados, incluindo dados semi-estruturados e bancos de dados externos;
- Extensão com algoritmos analíticos avançados tais como processamento de grafos e aprendizado de máquina.

Como fonte de dados, o Apache Spark aceita uma grande variedade de dados como Kafka, Kinesis e Sockets TCP que podem ser processados utilizando algoritmos complexos com funções de alto nível, com aprendizado de máquina ou processamento de grafos (módulos existentes no Spark). Finalmente os dados devem ser enviados a um sistema de arquivos, um banco de dados ou um dashboard em tempo real. O esquema do processamento dos dados em fluxos é apresentado na Figura 9.

Figura 9 – Esquema de processamento de Fluxo de Dados do Spark Streaming

Fonte: Spark Streaming Programming Guide - <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

Recebendo uma fonte de fluxo de dados contínuos externa como entrada, o Spark Streaming internamente divide esse fluxo de dados em lotes, que são processados pelo Spark gerando um fluxo final em lotes dos dados já processados (Figura 10).

Figura 10 – Esquema de processamento de Fluxo de Dados em batch do Spark Streaming

Fonte: Adaptado de *Spark Streaming Programming Guide* - <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

2.6.2 Apache Flink

O Apache Flink[2] é um framework para computação distribuída sobre fluxo de dados limitados e ilimitados. Foi desenvolvido para ser executado em ambientes de clusters e realizar o processamento dos fluxos fazendo uso da memória principal.

Fluxos de dados limitados e ilimitados são definidos pela documentação[3] do Flink da seguinte maneira:

- Fluxos de dados ilimitados: tem um começo, mas não é definido um fim do fluxo de dados. Esse tipo de fluxo não termina e os dados são providos conforme são gerados. Eles devem ser continuamente processados, isto é, "os eventos devem ser manuseados após sua ingestão". Não é possível aguardar todos os dados serem recebidos para depois serem processados.
- Fluxos de dados limitados: tem um começo e final definidos. Neste tipo, o processamento pode ser realizado após o recebimento de todos os dados. São também conhecidos como processamento em lotes.

2.6.3 Apache Storm

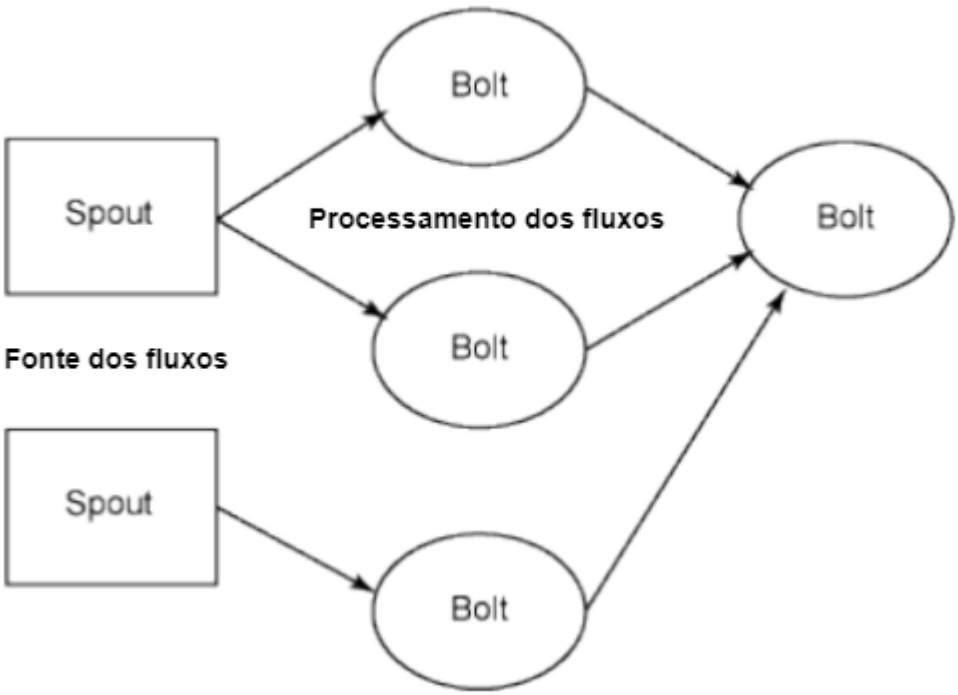
O Apache Storm é uma plataforma distribuída para o processamento de fluxo de dados em tempo real que tem como objetivo tornar fácil e confiável o processo de processamento de fluxos de dados ilimitados (BEZ, 2015).

Concebido com o conceito de "integrar qualquer sistema de enfileiramento de mensagens com qualquer sistema de banco de dados"[4].

[2] Apache Flink - <https://flink.apache.org/>
[3] Apache Flink Docs - <https://ci.apache.org/projects/flink/flink-docs-release-1.11>
[4] Apache Storm - <https://storm.apache.org>

Conforme apresentado por Toshniwal et al. (2014), o modelo de processamento do Storm consiste em fluxos de tuplas percorrendo uma topologia. A topologia é um grafo direcionado no qual os vértices representam computação e as arestas representam o fluxo de dados entre os componentes. Os vértices podem ser divididos em duas categorias: spouts e bolts. A Figura 11 ilustra uma topologia no Apache Storm.

Figura 11 – Exemplo de topologia no Apache Storm



Fonte: Adaptado de *Apache Storm Docs*

Os *spouts* são a origem dos fluxos de dados em uma topologia. São responsáveis por ler tuplas de uma fonte externa e inseri-las na topologia. O processamento é feito nos *bolts*, elementos que podem ter diversas funções, desde aplicação de filtros, funções, agregações, associações, até comunicação com banco de dados. Cada *bolt* pode realizar apenas uma das diversas tarefas, por isso são necessários vários *bolts* em uma topologia.

Um cluster Storm executa topologias, que em seu processo não tem um fim definido. Ela é executada até que se force o seu encerramento.

Existe um modo de execução do Apache Storm denominado "Modo Local", executado em processos que simulam um cluster localmente. Pela sua documentação, esse modo foi desenvolvido pensando no processo de desenvolvimento e testes, nos quais não seria necessário um cluster para sua execução.

2.6.4 Aurora e Borealis

O sistema Aurora é um Gerenciador de Fluxo de Dados de proposta geral desenvolvido em conjunto pela Brandeis University, Brown University e o M.I.T de modo que suportasse eficientemente uma variedade de aplicações de monitoramento de dados em tempo real (ABADI et al., 2003).

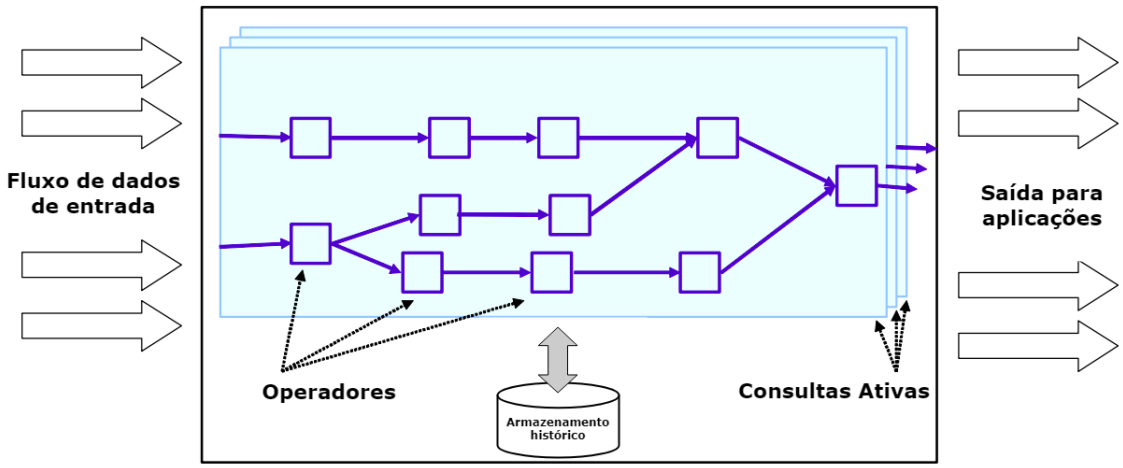
Uma rede de processamento Aurora (Figura 12) representa um conjunto de consultas ativas (seção 2.3) como um grafo cíclico direcionado com operadores baseados em operações sobre fluxo de dados. As tuplas são processadas conforme chegam em fluxos pela rede de processamento e são distribuídas as suas respectivas aplicações.

Os principais componentes do Aurora são:

- *scheduler*: tem a função de decidir quais operadores executar e em que ordem;
- *storage manager*: projetado para armazenar filas ordenadas de tuplas;
- *load shedder*: lida com situações de sobrecarga.

Cada aplicação Aurora define uma ou mais funções de qualidade de serviço, que corresponde a prioridade de cada consulta em relação as outras. Essa definição é o que fará a divisão de recursos computacionais e resultará no trabalho do *Load Shedder* em caso de sobrecarga, que decidirá por perder dados das aplicações que têm menor prioridade (ÇETINTEMEL et al., 2016).

Figura 12 – Exemplo da arquitetura de uma rede de processamento Aurora



Fonte: Adaptado de Abadi et al. (2003)

O sistema Aurora tem uma interface gráfica que permite construir redes de processamento Aurora, especificar a qualidade de serviço, além de permitir o monitoramento da execução das consultas ativas.

O projeto Borealis surge como uma segunda geração do projeto Aurora, herdando todo o motor de processamento de dados em fluxos, interface gráfica e tendo como o objetivo a migração para uma arquitetura distribuída. Essa mudança se deveu ao aumento de aplicações utilizando dados em fluxos e pelo aumento considerável na quantidade de dados (ABADI et al., 2005).

Para o Borealis foi definido um módulo administrador, responsável pela distribuição das consultas entre os nós que compõe um cluster. Cada nó executa uma instância do servidor Borealis cujo componente principal é o QP (*Query Processor*). Nele é executada cada consulta, cada uma em um diferente nó e posteriormente as respostas são recompostas para a distribuição aos clientes do serviço (ÇETINTEMEL et al., 2016).

2.6.5 STREAM

Acrônimo de *The STanford stREam datA Manager*, o projeto STREAM [5] criado na Universidade de Stanford tem como objetivo desenvolver um SGDF para processamento de consultas ativas sobre múltiplas fontes de fluxos de dados. Foi projetado para lidar com fluxos com grande volume de dados e um grande número de consultas ativas complexas (ARASU et al., 2016).

Duas motivações são citadas como as principais para concepção do projeto:

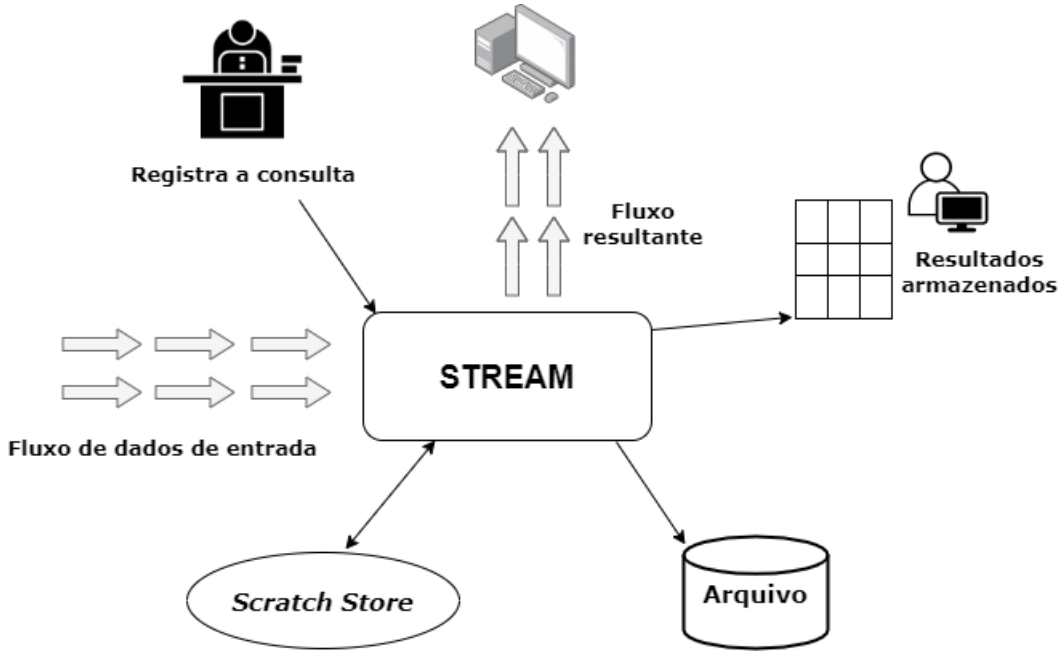
- Um SGDF deve lidar com múltiplas conexões e grande volume de dados que demandem respostas imediatas;
- Devido a natureza contínua dos fluxos de dados, um SGDF deve suportar consultas ativas que são continuamente executadas e que produzam resultados evitando atrasos;

Uma visão geral em alto nível é apresentada na Figura 13. Na esquerda estão representadas as entradas de dados que são fluxos de dados externos obtidos indefinidamente que são levados ao processamento das consultas. Para o processamento de consultas ativas tipicamente é necessário um estado intermediário dos dados que devem ser armazenados em memória ou disco, no STREAM esse estado é denominado *Scratch Store*. Além desse armazenamento, o STREAM faz um *Arquivo* dos fluxos de dados para posterior processamento ou análise histórica dos dados. No topo da figura estão representados os usuários ou aplicações que registram as consultas ativas que são executadas por tempo indeterminado ou até que sejam explicitamente canceladas. O resultados das consultas

[5] STREAM - The Stanford Stream Data Manager - <http://infolab.stanford.edu/stream/>

ativas são transmitidos como fluxos de dados, mas também podem ser resultados relacionais que são atualizados durante a execução.

Figura 13 – Visão Geral do Projeto STREAM



Fonte: Adaptado de Arasu et al. (2016)

Para o cadastro das consultas ativas foi definida uma linguagem própria, a *Continuous Query Language* (CQL) baseada em SQL e inclui operadores para tratamento de dados em fluxos (ARASU et al., 2016).

O projeto STREAM fornece uma interface Web para o registro das consultas ativas. As respostas são fornecidas em fluxos com formato XML pelo protocolo HTTP.

2.6.6 TelegraphCQ

Desenvolvido na Universidade de Berkley, é um sistema para processamento de consultas ativas sobre dados em fluxo. Baseado em uma nova arquitetura altamente adaptável que suporta consultas com cargas dinâmicas em ambientes com dados voláteis.

O *TelegraphCQ* é implementado aproveitando grande parte do código fonte do SGBD de código aberto, *PostgreSQL*. Apesar das diferenças entre um SGBD e um SGDF, pôde ser reutilizada uma grande parte incluindo recursos de extensão do *PostgreSQL* como tipos de dados e a capacidade de carregar funções desenvolvidas pelo usuário. Para criação das consultas ativas é utilizada uma linguagem derivada do CQL, do Projeto STREAM (subseção 2.6.5).

Os desafios encontrados no desenvolvimento usando como base o código de um SGBD convencional foram:

- Adaptatividade: a execução das consultas tem uma arquitetura bem diferente das consultas tradicionais;
- Processamento compartilhado: o *TelegraphCQ* compartilha a execução de múltiplas consultas, causando uma mudança significativa sobre o modelo de um processo por conexão do *PostgreSQL*;
- Ingestão dos dados: fluxos de dados resultam em diferentes métodos para o bloqueio de janelas de dados compartilhados;

O *TelegraphCQ* se aproveita do bom uso que o *PostgreSQL* faz da arquitetura de processos do *Unix* e utiliza o esquema de memória compartilhada em memória principal para a troca de informações entre os componentes de sua estrutura (MEEHAN et al., 2016).

2.6.7 Comparação entre trabalhos

Com a pesquisa sobre os principais SGDFs do mercado e os trabalhos científicos sobre o tema, foram montados quadros comparativos para elucidar suas principais características e comparar com o *SQLStreamify*, ferramenta produto deste trabalho.

As seguintes ferramentas foram comparadas:

- Apache Spark (Streaming)
- Apache Flink
- Apache Storm
- Aurora
- Borealis
- STREAM
- TelegraphCQ
- *SQLStreamify*

No Quadro 2 foram apresentados os objetivos gerais de cada um e o formato de distribuição das aplicações. O Apache Spark Streaming e o Apache Flink são distribuídos como um framework, com bibliotecas para manipulação de fluxos de dados para construção

de aplicações. O Apache Storm, Aurora, Borealis, STREAM e TelegraphCQ são distribuídos como SGDFs de uso geral. O STREAM tem como objetivo integrar múltiplas fontes de fluxos de dados permitindo realizar operações de manipulação sobre os mesmos. O TelegraphCQ tem como objetivo utilizar como base o código-fonte aberto do SGBD PostgreSQL para o desenvolvimento de um SGDF.

O *SQLStreamify* tem como proposta ser um *middleware* que integra, ao SGBD, fluxos de dados gerados por consultas SQL em dados constantemente inseridos. A capacidade de geração de fluxos de dados utilizando como fonte consultas a um SGBD é um dos diferenciais da proposta, entregando uma solução que se adapta a casos de usos nos quais isso é necessário, um pipeline diferente dos que são entregues pelos SGDFs disponíveis.

Quadro 2 – Quadro comparativo - Objetivos e Modelo da Aplicação

Trabalho	**Objetivo**	**Formato**
Apache Spark (Streaming)	- Construção de aplicações utilizando dados em fluxos	Módulo de Framework
Apache Flink	- Computação distribuída sobre dados em fluxos	Framework
Apache Storm	- Processamento de fluxos de dados em tempo real	SGDF de uso geral
Aurora	- Monitoramento de dados em tempo real	SGDF de uso geral
Borealis	- Monitoramento de dados em tempo real	SGDF de uso geral
STREAM	- Consultas ativas sobre multiplas fontes de dados	SGDF de uso geral
TelegraphCQ	- Processamento de consultas ativas sobre dados em fluxos - Utilizar como base o código-fonte do PostgreSQL (SGBD)	SGDF de uso geral
SQLStreamify	- Utilizando o SGBD realizar consultas em dados inseridos constantemente para a geração e distribuição de fluxos de dados	Middleware

No Quadro 3 são apresentadas as possíveis fontes de dados de cada um dos trabalhos bem como seu suporte a linguagem de manipulação de dados. A grande maioria deles trabalham tendo como fonte fluxos de dados já existentes e fazem a manipulação e operações sobre eles. A proposta do SQLStreamify é gerar esse fluxo de dados com base em consultas em uma base de dados estruturada existente que tenha inserções constantes. O Apache Spark tem função semelhante em outro módulo que faz parte do framework. A integração do SQLStreamify se dá nos SGBDs MySQL[6] e MariaDB[7].

Vários trabalhos usam variações do SQL para a manipulação dos fluxos de dados como o CQL no STREAM e no TelegraphCQ (variação do CQL). O Apache Storm criou o

[6] MySQL - SGBD de código aberto mais utilizado <https://mysql.com/>

[7] MariaDB - SGBD de código aberto desenvolvido pelo time dos primeiros desenvolvedores do MySQL com o propósito de permanecer com seu código-fonte aberto <https://mariadb.org/>

StormSQL e no Apache Spark tem o SparkSQL. O Aurora e o Borealis usam representações gráficas para a manipulação dos fluxos de dados.

Quadro 3 – Quadro comparativo - Fonte de Dados e Suporte à SQL

Trabalho	**Fonte de dados**	**Suporte à SQL**
Apache Spark (Streaming)	Suporte a diferentes fontes de dados, incluindo dados estruturados desde que em fluxos.	Spark SQL (manipulações sobre fluxos de dados)
Apache Flink	Fluxo de dados	não
Apache Storm	Fluxo de dados	StormSQL (manipulações sobre fluxos de dados)
Aurora	Fluxos de dados	não
Borealis	Fluxos de dados	não
STREAM	Fluxos de dados	não (CQL - baseada em SQL)
TelegraphCQ	Fluxos de dados	CQL (do Projeto STREAM, com modificações)
SQLStreamify	MySQL, MariaDB	SQL para a geração dos fluxos de dados

Considerando a arquitetura e execução dos SGDFs foi montado o Quadro 4. A grande maioria utiliza arquitetura de computação distribuída para o desenvolvimento. Isso se deve ao fato do aproveitamento de recursos para tratamento de grandes fluxos de dados. São executados em clusters o Apache Spark, Apache Flink, Apache Storm e Borealis. O Apache Storm disponibiliza um modo de execução local, que não é apropriado para a execução em produção, mas sim para testes e desenvolvimento. O Borealis tem em seu precursor Aurora o modelo monolítico. O STREAM e o TelegraphCQ frutos de desenvolvimento acadêmicos tem sua arquitetura para produção pouco desenvolvida pensada em casos de testes de desempenho. O SQLStreamify foi projetado em microsserviços que podem ser executados em uma grande variedade de ambientes, tanto local como em clusters, podendo escalar para atender aos requisitos de cada caso de uso.

O Apache Spark, Flink e Storm por serem frameworks ou bibliotecas para a manipulação de fluxos de dados, não possuem uma interface de gerenciamento. O Aurora e Borealis têm uma interface para a criação dos manipuladores sobre os fluxos de dados. O STREAM tem em seu projeto o plano de desenvolvimento de uma interface web para o gerenciamento. O SQLStreamify propõe uma interface para monitoramento das consultas ativas em execução.

2.7 Arquitetura baseadas em microsserviços

O número de aplicações com arquiteturas baseadas em microsserviços tem crescido rapidamente. Muitas aplicações distribuídas e baseadas em nuvens evoluíram de arquiteturas monolíticas para microsserviços. A arquitetura se baseia em dividir o domínio da

Quadro 4 – Quadro comparativo - Arquitetura

Trabalho	Arquitetura	Execução	Interface de Gerenciamento
Apache Spark (Streaming)	computação distribuída		
Apache Flink	computação distribuída	clusters	
Apache Storm	- computação distribuída - modo local para desenvolvimento e testes	- clusters - local*	
Aurora	monolitica		GUI
Borealis	distribuída	cluster	GUI
STREAM			interface web
TelegraphCQ			
SQLStreamify	distribuída, microsserviços		WebUI (administração)

aplicação em problemas menores que consequentemente têm uma complexidade menor. Os microsserviços devem ser independentes e trocam informações com outros microsserviços para se complementarem formando a aplicação. Exemplos de aplicações de larga escala que adotaram microsserviços incluem *Netflix*, *Amazon*, *Uber* (SOUSA, 2018).

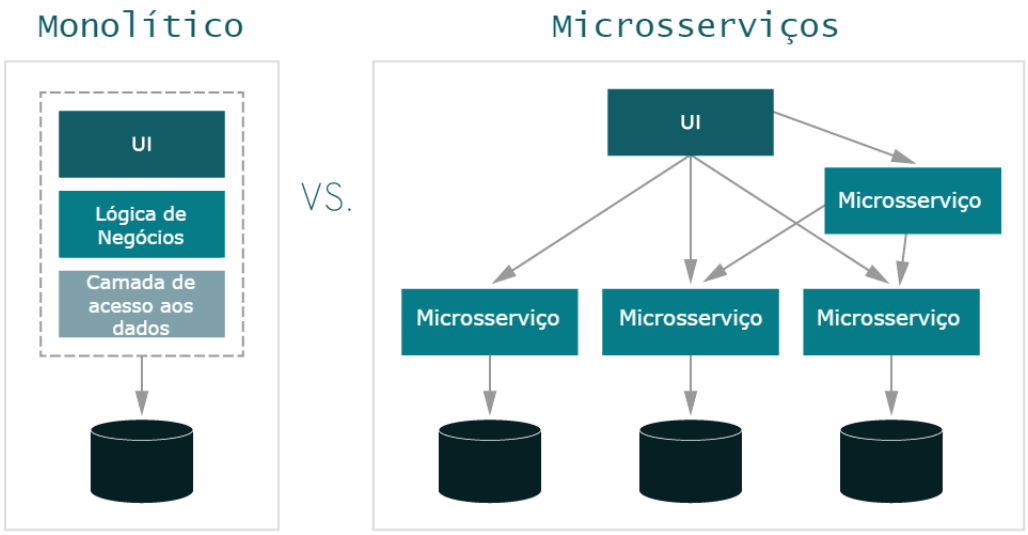
Essa abordagem para a criação de aplicações se diferencia das abordagens monolíticas pelo modo como decompõe a aplicação em funções básicas(Figura 14). Cada função denomina um serviço que pode ser criado e implantado de maneira independente. Isso significa que cada serviço individual pode funcionar ou falhar sem comprometer os demais (REDHAT, 2020). Quebrar o domínio do problema em problemas menores é uma das vantagens obtidas pela escolha pela arquitetura baseada em microsserviços, diminuindo a complexidade de cada função do *SQLStreamify*.

Microsserviços normalmente por sua característica de isolamento e independência são executados em contêineres. Diferente de máquinas virtuais, contêineres não criam uma cópia virtual do hardware do host (máquina que provê o serviço de gerenciamento de contêiner) e nem necessitam um SO completo instalado neles (Figura 15). Funcionam diretamente sobre o hardware físico e compartilham o kernel (núcleo) do SO utilizado pelo host, exigindo apenas as bibliotecas e softwares desejados isoladamente em cada contêiner. Vários contêineres podem ser executados num mesmo host, inclusive várias instâncias de uma mesma imagem de contêiner pode ser executadas num mesmo momento. Dessa forma ocupam muito menos espaço e recursos e tem sua execução com desempenho muito semelhante a serviços que rodam diretamente no host.

Um contêiner pode ser entendido como um compartimento dentro do SO host, com acesso a uma parte dos recursos computacionais do hospedeiro, limitados durante o provisionamento do contêiner pelo seu gerenciador.

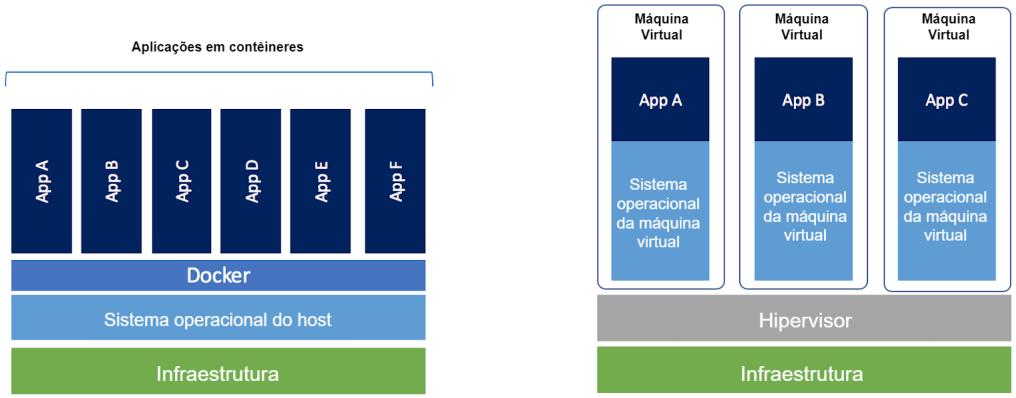
Contêineres são cada vez mais populares por serem (DOCKER, 2019):

Figura 14 – Comparação entre arquitetura monolítica e arquitetura baseada em microsserviços



Fonte: Adaptado de RedHat (2020)

Figura 15 – Diferenças entre Contêineres e Máquinas Virtuais



Fonte: Adaptado de Docker (2019)

- Flexíveis: até as aplicações mais complexas podem ser convertidas para contêineres;
- Leves: por compartilharem o kernel e recursos com o host;
- Intercambiáveis: por permitirem atualizações durante a execução;
- Portáveis: por poderem ser executadas em máquinas de desenvolvimento, rodarem em nuvens ou qualquer servidor;

- Escaláveis: permitem aumentar e distribuir automaticamente réplicas dos contêineres;
- Empilháveis: permitem empilhar serviços verticalmente em execução.

2.8 Utilização do log binário de SGBDs

Trabalhos científicos apresentam diversos usos para o log de SGBDs, colocando-o como uma poderosa fonte de recursos para aplicações práticas, desde aprimoramento de consultas, análises forenses, entre outras (SELLAM; KERSTEN, 2017; GHOSH et al., 2002).

Nesta seção são apresentados alguns trabalhos que utilizam o log binário para aplicações práticas.

2.8.1 Aprimoramento de consultas

Uma vez que os logs descrevem se as consultas são realizadas com sucesso ou falha, quanto tempo levam para serem executadas e quantas tuplas retornam. Combinado a algoritmos preditivos, podem ajudar emitindo alertas, na escolha de um plano de consultas mais efetivo e construir motores de buscas mais robustos para SGBDs (SELLAM; KERSTEN, 2017).

Ghosh et al. (2002) propõem um sistema que sugere um plano de consultas baseado em tempo de resposta obtidos do log binário do SGBD. Para isso utilizam dados como número de tabelas mencionadas, relacionamentos entre tabelas e presença ou não de índices.

2.8.2 Análise Forense

O trabalho de análise forense consiste em coletar, preservar e analisar evidências encontradas em computadores e mídias de armazenamento digital.

Neste cenário o uso de logs para reconstituir informações é amplamente utilizado. Frühwirt et al. (2013), em sua pesquisa, apresenta uma aplicação que consiste em reconstruir manipulações de dados em SGBDs através do log binário.

2.8.3 Fluxo com atividades de Banco de dados com *Amazon Aurora*

Desde meados de 2020, a *Amazon* passou a oferecer stream de atividades do banco de dados dos serviços *Amazon Aurora* (distribuições dos banco de dados relacionais *MySQL* e *PostgreSQL* de dentro do *Amazon Web Services*). Segundo informações da própria *Amazon*, esses dados são fornecidos em tempo real. Essas informações são fornecidas para serem integradas a ferramentas de monitoramento de atividade do banco de dados de forma

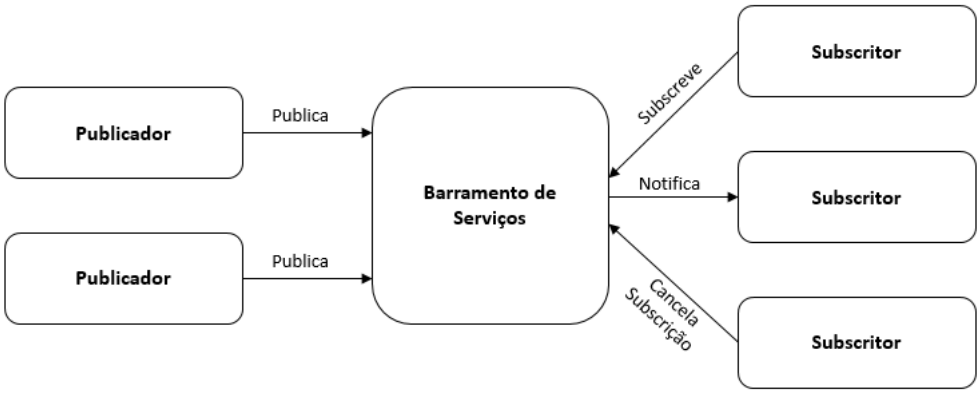
assíncrona em um stream de dados criptografados do *Amazon Kinesis* (serviço de stream de dados da Amazon).

2.9 Paradigma *Publish-Subscribe*

É um paradigma de comunicação que apoia a construção de sistemas distribuídos assíncronos, capaz de prover uma forma de comunicação desacoplada, na qual produtores de informações enviam mensagens sem a necessidade de saber quem são os consumidores. Essa capacidade aumenta a escalabilidade em sistemas de grande porte, pois os elos entre os produtores e os consumidores não são estaticamente definidos (EUGSTER et al., 2003; CARVALHO, 2017).

No método *Publish-Subscribe* existe o conceito de um *Broker*, barramento de serviços, no qual entidades denominadas de publicadores (produtores) publicam seus eventos e subscritores (consumidores) se subscrevem para receber eventos de interesse. Quando um evento ocorre no barramento e é do tipo que o subscritor registrou seu interesse, ele o recebe por uma notificação. O barramento é o canal de comunicação no qual há a interação entre produtores e consumidores. Normalmente implementado por *middleware*, é responsável por lidar com o processo de troca de mensagens, além de requisitos relacionados à rede, como tolerância a falhas, garantia de entrega, entre outros. Na Figura 16 é possível observar a interação entre os elementos de um sistema *Publish-Subscribe* (CARVALHO, 2017).

Figura 16 – Sistema Publish-Subscribe simples.



Fonte: Carvalho (2017)

2.10 MQTT

MQTT (Message Queue Telemetry Transport) é um protocolo de comunicação voltado para aplicações M2M (*Machine-to-Machine*) que utiliza o paradigma *publish-subscribe* para troca de mensagens. Foi desenvolvido pensando em atender taxas de conexões de rede e hardwares embarcados (com potência reduzida, muito utilizados em dispositivos conectados) e que possibilitem em certo grau, garantia de entrega das mensagens com confiabilidade. Essas características fazem do MQTT um dos principais candidatos a protocolo de comunicação voltado para aplicações IoT (*Internet of Things* - Internet das Coisas) ou M2M e também desponta quando se trata de aplicações destinadas a dispositivos móveis, nos quais economia de energia e gerenciamento da rede são quesitos vitais (LIGHT et al., 2017; MELO et al., 2018; MQTT, 2019).

> Sua aplicação original era vincular sensores em pipelines de petróleo a satélites. Como seu nome sugere, ele é um protocolo de mensagem com suporte para a comunicação assíncrona entre as partes. Um protocolo de sistema de mensagens assíncrono desacopla o emissor e o receptor da mensagem, tanto no espaço quanto no tempo e, portanto, é escalável em ambientes de rede que não são confiáveis. Apesar de seu nome, ele não tem nada a ver com filas de mensagens, na verdade, ele usa um modelo publish/subscriber. No final de 2014, ele se tornou oficialmente um padrão aberto OASIS, com suporte em linguagens de programação populares, usando diversas implementações de software livre (YUAN, 2017).

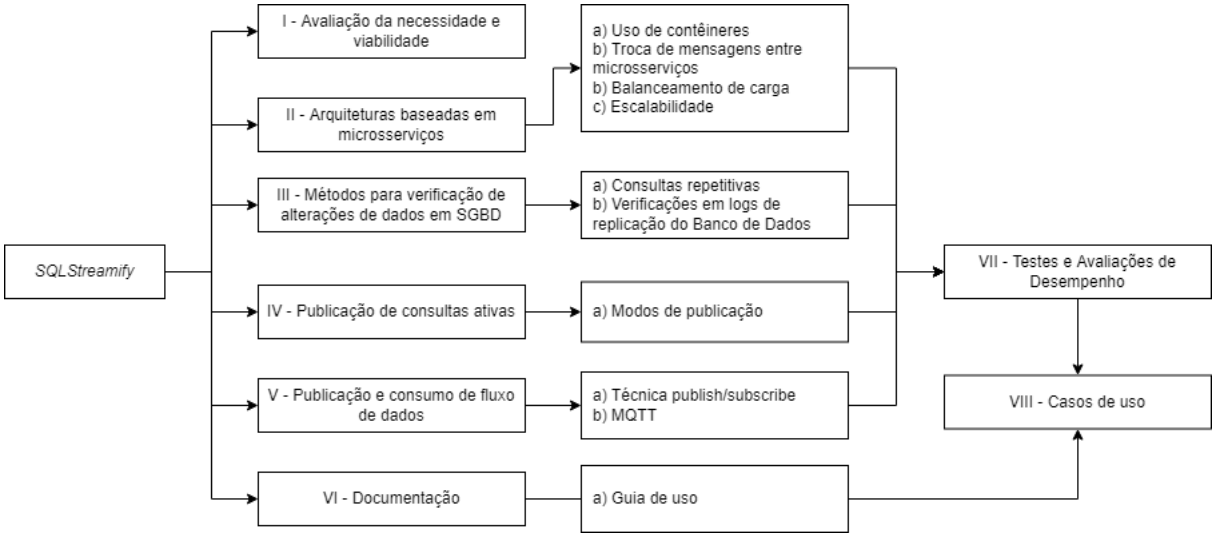
O protocolo MQTT proporciona desacoplamento entre os elementos da comunicação nas dimensões espacial (*publisher* e *subscriber* não precisam se conhecer) e temporal (não é necessário que o *publisher* e o *subscriber* estejam em execução simultaneamente). Devido a esse desacoplamento a conexão sempre acontece entre cliente e *broker*. São considerados clientes MQTT, os elementos finais do processo de comunicação, ou seja, qualquer dispositivo com características de hardware suficientes para executar uma biblioteca MQTT e que esteja conectado a um *broker* MQTT. Este, por sua vez, é responsável por receber todas as mensagens e filtrá-las, para que sejam enviadas aos respectivos interessados (HIVEMQENTERPRISE, 2016).

Alguns exemplos de aplicações conhecidas e com grande número de usuários que utilizam MQTT são *Facebook Messenger, Amazon Web Services, Openstack, Microsoft Azure*, entre outros.

3 Métodos e técnicas

Os métodos, técnicas e procedimentos empregados no desenvolvimento do projeto do *SQLStreamify* são apresentados neste capítulo e estão esquematizadas na Figura 17.

Figura 17 – Metodologia do trabalho



Fonte: Elaborado pelo autor

3.1 Visão geral da solução

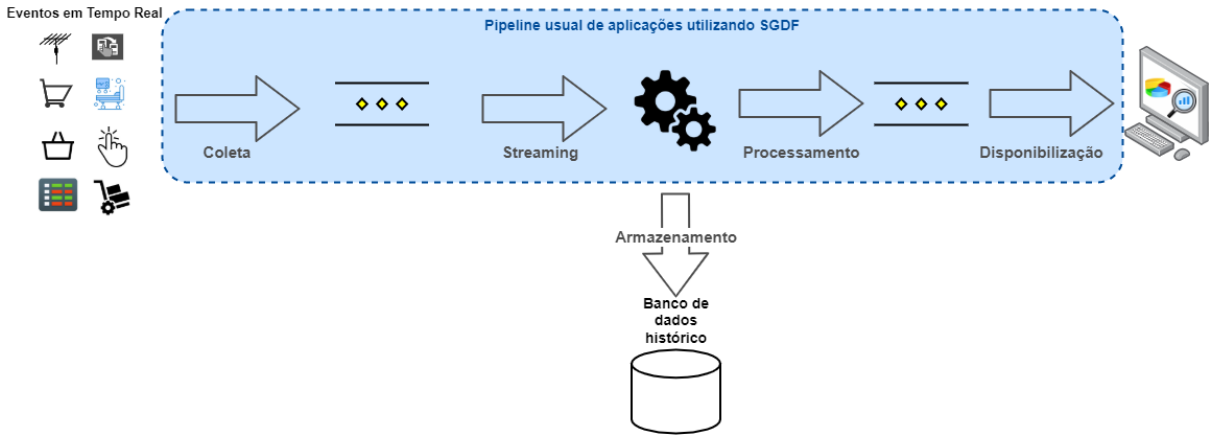
Como demonstrado na Revisão Bibliográfica (Capítulo 2), desenvolvedores se deparam com situações nas quais necessitam de funções reativas em suas aplicações que podem já estar em produção, utilizando SGBDs convencionais, sem suporte à consultas ativas (dados são retornados em fluxos contínuos, conforme são atualizados) (COSTA; LEITE; NETO, 2008; FERNANDES et al., 2014). Além disso, diversos fatores podem inviabilizar a troca de um SGBD por um SGDF, como a substituição da estrutura física necessária para executá-lo, na maioria das vezes, em *clouds* e servidores mais robustos, assim como o custo do aprendizado de uma nova tecnologia, a substituição de sistemas já em produção, troca de linguagem ou *framework* de programação entre outros.

Em sistemas de informação, frequentemente há a necessidade de apenas algumas funções básicas de leitura de fluxo de dados em tempo real (COSTA; LEITE; NETO, 2008), com poucas consultas ativas e não todas as características e funcionalidades de um SGDF (FERNANDES et al., 2014). Tal consideração, constitui um desafio deste trabalho disponibilizar, por meio de um *middleware*, transações de consultas ativas que gerem como

resultado fluxos de dados em Tempo Real obtidos de SGBDs existentes, utilizando uma arquitetura baseada em microsserviços, executada em contêineres, que possibilitem alta escalabilidade.

O *pipeline* comum de aplicações que fazem uso de fluxo de dados é apresentado na Figura 18, na qual os dados são armazenados depois da sua coleta e com processamento básico. Esse banco é constantemente definido como Banco de Dados Histórico.

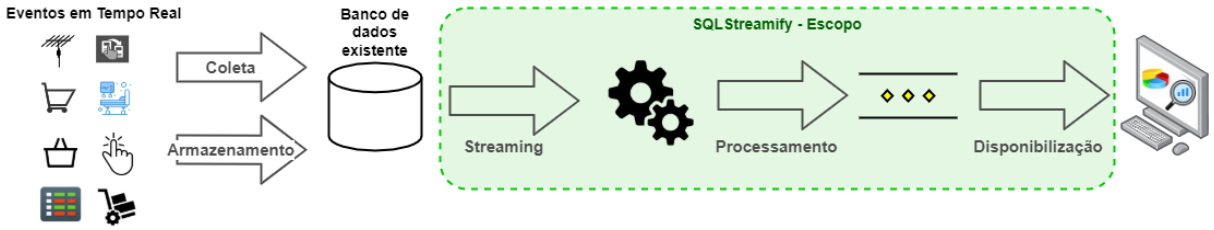
Figura 18 – Visão em alto nível de um pipeline de SGDF



Fonte: Elaborado pelo autor

No entanto, existem casos nos quais aplicações legadas fazem a captura e armazenamento de dados em bancos de dados existentes. As atualizações nos dados destes banco de dados são os eventos desejados em novas aplicações que fazem uso de dados em fluxos. Esse é precisamente o escopo deste trabalho. O *SQLStreamify* transforma esses eventos em fluxos de dados pela consulta dos dados coletados e armazenados em Bancos de Dados relacionais por aplicações legadas já existentes. (veja Figura 19).

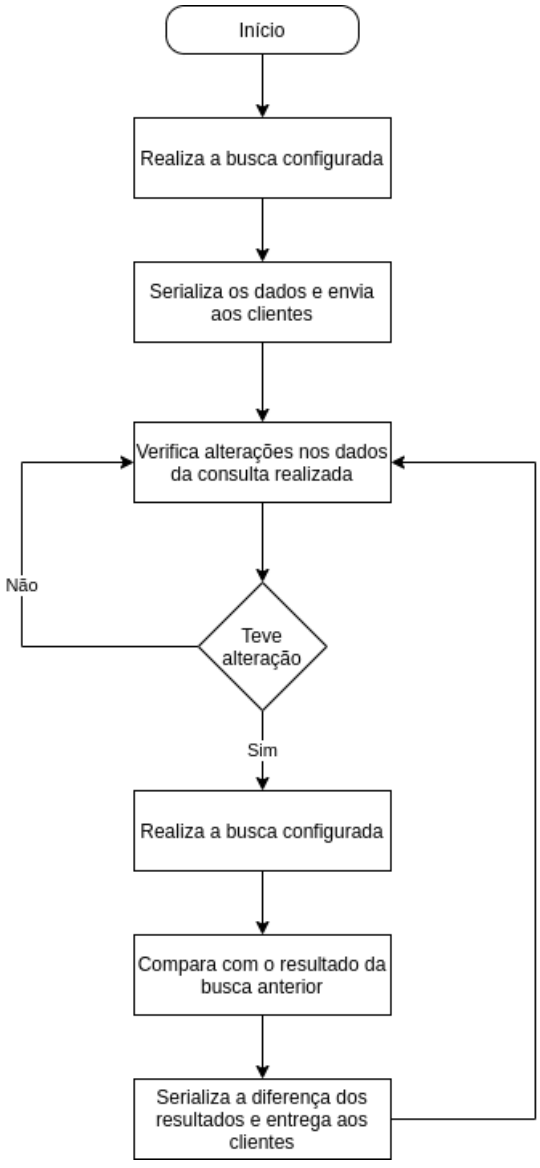
Figura 19 – Situação na qual a coleta e armazenamento são feitos por uma aplicação já existente e existe a necessidade de geração de um fluxo de dados com os eventos de inserções e atualizações de dados em SGDBs



Fonte: Elaborado pelo autor

Um fluxograma com a sequência de instruções do *SQLStreamify* é demonstrado na Figura 20 com alto grau de abstração que serve para elucidar seu funcionamento.

Figura 20 – Fluxograma com alto grau de abstração com o funcionamento do *SQLStreamify*



Fonte: Elaborado pelo autor

O funcionamento do *SQLStreamify* se baseia em: cadastradas as consultas em linguagem SQL, realizar as buscas, verificar quando houverem alterações nos dados referentes a estas buscas e entregar, no instante das alterações, os dados do que foram alterados.

Um dos objetivos ao desenvolver o *SQLStreamify* utilizando uma arquitetura baseada em microsserviços (seção 2.7) é a flexibilidade obtida por sua modularidade. Por exemplo, tendo a conexão sido desenvolvida em módulo para um modelo específico de SGBD, podem ser criados módulos que se conectem a outros modelos de SGBD aumentando

os casos em que o *SQLStreamify* pode ser usado. Além disso, tem-se a vantagem da divisão do domínio da aplicação em problemas menores com consequente complexidade reduzida e a possibilidade de execução de contêineres em diferentes sistemas operacionais, sem necessidade de adaptações em sua estrutura.

SQLStreamify foi proposto considerando que desenvolvedores constantemente necessitam de funções de leitura de fluxos de dados em tempo real em aplicações que usam SGBDs relacionais existentes (COSTA; LEITE; NETO, 2008; FERNANDES et al., 2014). Ele entrega consultas ativas baseadas nas alterações de dados e notifica aos subscritores quando os dados correspondentes as consultas cadastradas são alterados.

3.1.1 Estrutura do serviço

A estrutura do *SQLStreamify* consiste em cinco microsserviços - Gerenciador, Eventos, Consultas, WebUI e Broker-MQTT (Figura 21). Além deles é utilizado um Banco de Dados não relacional com armazenamento em memória principal. Sua função é servir como armazenamento rápido de alto desempenho que todos os microsserviços possam ter acesso direto e rápido aos dados que necessitem ser trocados entre eles.

O banco de dados não relacional utilizado para armazenamento dos dados é o Redis[1] por ser de código aberto, altamente escalável e por utilizar a memória principal para armazenamento de suas estruturas no formato chave/valor, o que o torna extremamente eficaz. Outra vantagem é ser distribuído em imagens prontas para serem executadas em contêineres, se encaixando perfeitamente na arquitetura escolhida para o desenvolvimento (Alami; Bahaj; Khourdifi, 2018; Zhang et al., 2018).

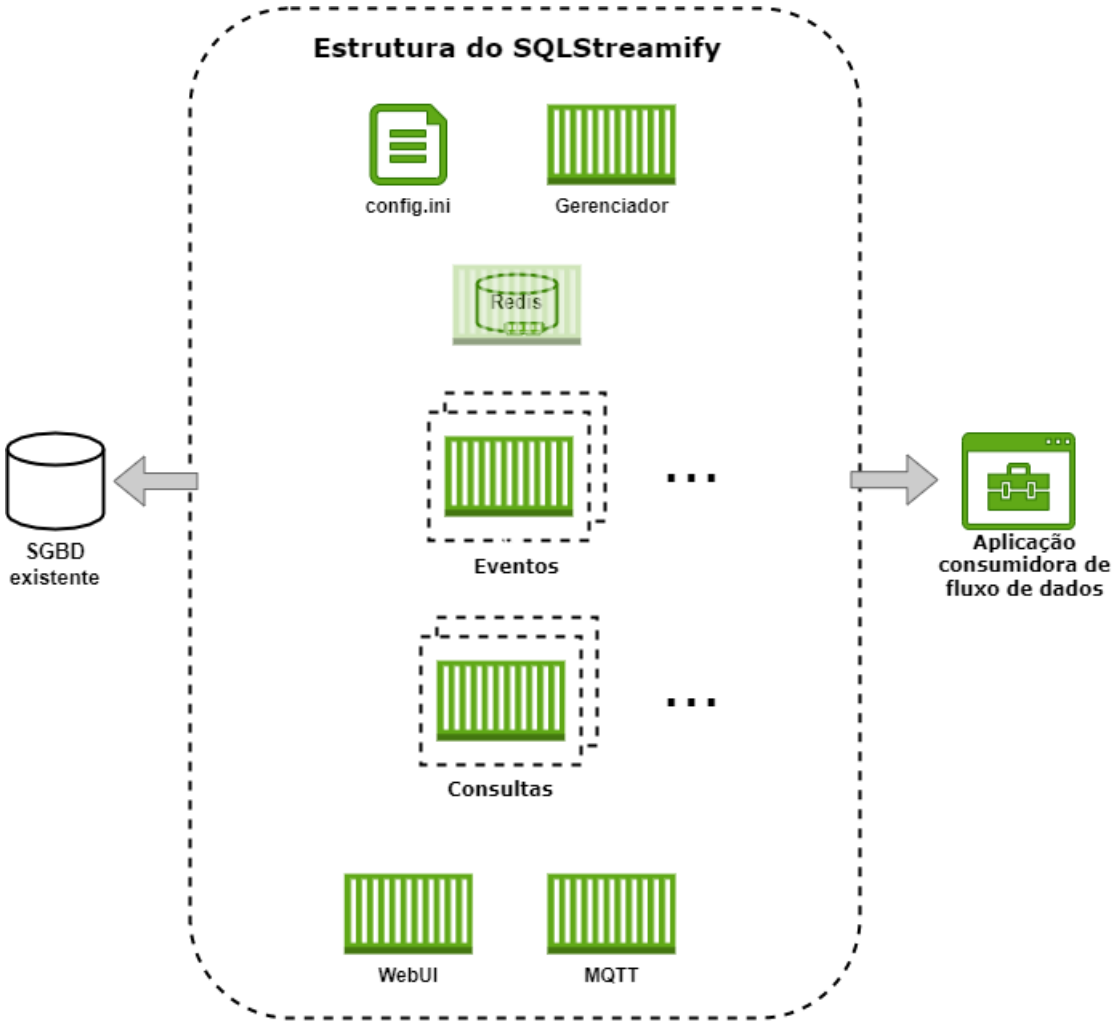
O primeiro passo para o funcionamento do *SQLStreamify* é criar um arquivo de configuração com os parâmetros de acesso ao banco de dados, de exposição para a publicação dos fluxos de dados e configurações das consultas em linguagem SQL, escolhendo um nome e o modo da consulta que são detalhados na subseção 3.4.1. O modelo para o arquivo de configuração é mostrado em um exemplo no Código 3.1.

Código 3.1 – Exemplo do arquivo de configuração

```
[DB]
type = mysql
host = 172.17.0.1
port = 3306
db = banco_de_dados
user = usuario
password = senha

[EXPOSICAO]
ip = 200.200.200.1

[nome da consulta]
query = SELECT * FROM compras
```

[1] Redis - <https://redis.io/>

Figura 21 – Estrutura do *SQLStreamify* utilizando microsserviços executados em contêineres



Fonte: Elaborado pelo autor

```
modo = full_dataset
```

Um script de inicialização do *SQLStreamify*, desenvolvido em Python[2], faz a leitura do arquivo de configuração e um balanceamento inicial dos microsserviços baseado no número de consultas cadastradas (mais detalhes sobre o balanceamento são apresentados na seção 3.3). O script então cria o arquivo necessário para inicializar os contêineres e o executa.

O arquivo para a criação do ambiente com todos os microsserviços e toda a comunicação possível entre eles é especificada em formato YAML[3] de nome docker-

[2] Linguagem de Programação - Python - <https://python.org/>
[3] YAML - formato para serialização de dados - <https://yaml.org/>

compose.yml. Neste arquivo são definidos quais microsserviços têm acesso para se comunicar com os outros pelo parâmetro “links”, além de serem definidas as portas que serão abertas para responder algum servidor que for executado no contêiner pelo parâmetro “ports”. Com esses parâmetros, toda a configuração de rede é criada, com os microsserviços respondendo pelo nome dado a cada um. O parâmetro “scale” tem o número de instâncias que serão executadas de cada microsserviço que serão definidos pelo script de inicialização que cria o docker-compose. No “build” é definido o nome da imagem que será utilizada.

Cada imagem é criada de acordo com a necessidade dos microsserviços e são geradas utilizando um arquivo Dockerfile. Neste são definidos o sistema operacional, os serviços, bibliotecas da linguagem e tudo o que for necessário para a execução. Uma vantagem de se utilizar contêineres é que sempre que a imagem for construída serão instaladas as últimas versões ou as versões definidas para cada imagem, o que torna o processo de atualização mais prático em futuras instalações e extremamente estável quando definidas as versões utilizadas (XIE; Govardhan, 2020).

Quadro 5 – Tecnologias utilizadas em cada microsserviço

Microsserviço	**Linguagem**	**Framework**	**Banco de Dados**	**Múltiplas Instâncias**
Gerenciador	Python		Redis	Não
Eventos	Python	Flask	Redis SGBD	Sim (1 por consulta)
Consulta	Python	Flask	Redis SGBD	Sim (2 por consulta)
WebUI	Python Javascript	Flask JQuery		Não

O serviço *Gerenciador* tem a função de criar no *Redis* um set com as consultas criadas, seus modos de publicação e contadores de eventos para o monitoramento das atividades. Esse serviço será o responsável por solicitar a execução dos outros microsserviços e controlar a performance das atividades de cada consulta (Quadro 5).

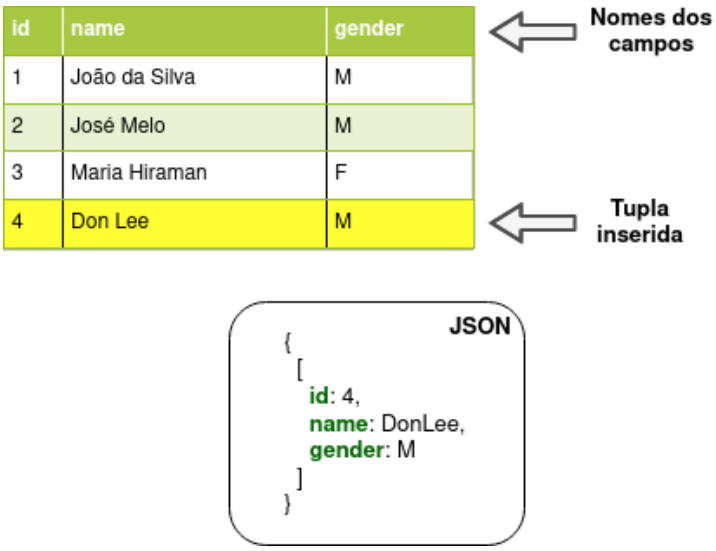
O microsserviço de Controle de Eventos é executado em contêineres e em mais de uma instância se necessário. O balanceamento inicial é de uma instância para cada consulta cadastrada no arquivo de configuração do SQLStreamify. Esse microsserviço executa um servidor web que aguarda por requisições *HTTP* que serão feitas pelo microsserviço *Gerenciador* com os parâmetros das consultas. Sua primeira função é, utilizando a consulta cadastrada em SQL, identificar as tabelas e os dados por ela afetados.

O funcionamento desse microsserviço é descrito detalhadamente na subseção 3.2.3 na qual é apresentada a técnica para a verificação das alterações nos dados do SGBD utilizando contêineres atuando como servidores de replicação.

Assim que um evento for identificado nos dados verificados é feita uma chamada para o próximo microsserviço, o de Consulta. Este faz uma busca do último resultado

armazenado em memória, depois disso realiza a consulta no SGBD e compara os resultados. Com a diferença obtida, são extraídos os nomes dos campos e os dados para a publicação (Figura 22). O último resultado é então armazenado na memória.

Figura 22 – Captura dos nomes dos campos da consulta e serialização dos dados em formato JSON



Fonte: Elaborado pelo autor

Com o nome dos campos e com os dados que foram alterados, é gerado um JSON[4] com o resultado para ser distribuído pelo MQTT, de acordo com o modo de publicação escolhido para a consulta. O processo de distribuição e os modos dessa distribuição são detalhados na seção 3.4.

Por fim, existe o microsserviço WebUI que provê uma interface de monitoramento do *SQLStreamify*. Foi desenvolvido em Python utilizando o framework web *Flask*[5] e Javascript para mostrar em tempo real as atualizações com dados de número de eventos por consulta e os dados de cada mensagem enviada aos clientes.

3.1.2 Troca de mensagens entre microsserviços

Em contraste com o modelo monolítico, na arquitetura de microsserviços, cada componente de uma aplicação, chamado de microsserviço, é tratado como parte independente mas que opera junto a outros na execução de tarefas e se comunicam utilizando mecanismos leves.

No arquivo de definição da estrutura dos microsserviços são definidos quais podem comunicar-se entre si. Essa comunicação deve ser implementada pela aplicação. Para o

[4] *Javascript Object Notation* - Notação de objetos Javascript - formatação para troca de dados
[5] Flask - <https://palletsprojects.com/p/flask/>

SQLStreamify, em cada microsserviço a aplicação foi desenvolvida utilizando Python - uma das linguagens de programação mais populares para o desenvolvimento de back-end de aplicações web - e o popular framework web Flask, que provê um simples e flexível servidor Web para atender requisições (Vogel et al., 2017).

Utilizando o Flask, são criadas rotas HTTP que serão expostas por cada microsserviço como mostradas no Quadro 6. Por essas rotas é possível também enviar parâmetros, funcionando de forma semelhante à chamadas de funções em linguagens de programação funcionais.

Quadro 6 – Quadro com as rotas para comunicação entre microsserviços

Microsserviço	**Rota**	**Função**
Consulta	/	- status do serviço
	/<string:consulta>	- publica o resultado no broker MQTT - retorna o resultado da busca em JSON
Eventos	/	- status do serviço
	/<string:consulta>/<int:server_id>	- inicia um servidor de replicação para a consulta com o server_id indicado

3.2 Métodos para verificação de alterações de dados em SGBD

O primeiro grande desafio para o desenvolvimento deste trabalho foi implementar e testar métodos para detecção de alterações em dados, baseados em consultas em SQL, de SGBDs em produção. Isso se faz necessário, uma vez que as atualizações nestes dados constituirão fluxos que serão transmitidos no caso de coincidirem com as consultas cadastradas.

Dos três métodos apresentados nas subseções a seguir, foi escolhido o da análise dos logs de replicação do SGBD, devido as suas vantagens em relação aos outros métodos.

3.2.1 Criação de triggers para notificação de alterações

Um dos recursos e alternativas cogitados no desenvolvimento do detecção de alterações é o uso de triggers. Gatilhos que são disparados quando alcançadas algumas condições (ELMASRI et al., 2005).

Esta solução foi dispensada uma vez que por tratar de Bancos de Dados existentes e utilizados por outras aplicações, alterar estruturalmente o Banco de Dados por vezes não é permitido.

Um dos objetivos é fazer a detecção sem realizar alterações no Banco de Dados. Por ser um *middleware* e trabalhar na intermediação para a entrega de suas funções, quanto menos alterações nas aplicações e estruturas já existentes e em funcionamento, mais interessante o funcionamento do *middleware*.

3.2.2 Consultas repetitivas

Baseado na pesquisa de Terry et al. (1992), um primeiro cenário foi desenvolvido utilizando o método de repetição de consultas em janelas de tempo. O desenvolvimento dessa abordagem é interessante para verificar o desenvolvimento da função de identificar alterações em sua forma mais primitiva e geradora do conceito de consultas ativas. Comumente é esse método que desenvolvedores fazem uso ao implementarem funções reativas em SGBDs sem funções de consultas ativas (COSTA; LEITE; NETO, 2008; FERNANDES et al., 2014). Resultados de testes de performance dessa abordagem servem de base em comparações com abordagens mais práticas e eficientes e são abordados no Capítulo 4.

Código 3.2 – Abordagem de consultas repetitivas

```
set t an amount of time
r1 = execute_query()
publish(r1)
FOREVER DO
        r2 = execute_query()
        diff_ = diff(r2,r1)
        if diff_ != NULL
                publish(diff_)
                r1 = r2
        Sleep for t
ENDLOOP
```

O método consiste em definir uma janela de tempo t em que a consulta é realizada repetidamente (Código 3.2). Os resultados obtidos serão armazenados em um dataset. Após o período t de tempo executa-se novamente a consulta e o resultado também é armazenado. Os resultados são comparados e as diferenças entre eles indica se houveram ou não alterações nos dados. Caso a diferença não for nula indica que houveram alterações. Esse mesmo procedimento é repetido de modo a obter os dados modificados enquanto se deseja o monitoramento da consulta. Para esse primeiro método, foi desenvolvida a aplicação utilizando contêiner (Figura 23), na qual é feita a consulta repetidas vezes ao banco de dados.

3.2.3 Verificação de alterações de dados pela análise logs de replicação de SGBDs

Em Bancos de Dados relacionais são implementados mecanismos de controle de concorrência, utilizando travas para impedir que os dados em suas tabelas sejam atualizados por mais de um usuário ao mesmo tempo, o que poderia provocar inconsistências no Banco

Figura 23 – Consultas repetitivas ao Banco de Dados em busca de alterações nos dados

Fonte: Elaborado pelo autor

de Dados. Essas travas impedem que alguém altere um dado enquanto outro estiver utilizando. Existem diversos tipos de travas, sendo que algumas delas impedem até mesmo que alguém atualize um dado enquanto este estiver sendo consultado, mesmo sem a realização de alterações (Eisa; Salem, 2017).

O processo de consultas repetitivas além de custoso devido ao alto número de comparações de datasets também bloqueiam o acesso a tabelas ou janelas de dados que estão sendo consultados, impedindo novos acessos para leitura ou escrita, travando o banco de dados enquanto estiverem sendo realizadas as consultas. Se a janela de tempo entre as consultas for pequena ou as consultas levarem muito tempo para serem executadas esse bloqueio é praticamente permanente nos dados em questão.

Para resolver o problema de bloqueio dos dados e diminuir o custo computacional foi desenvolvida uma alternativa que busque essas alterações em dados do SGBD sem a necessidade de realizar consultas.

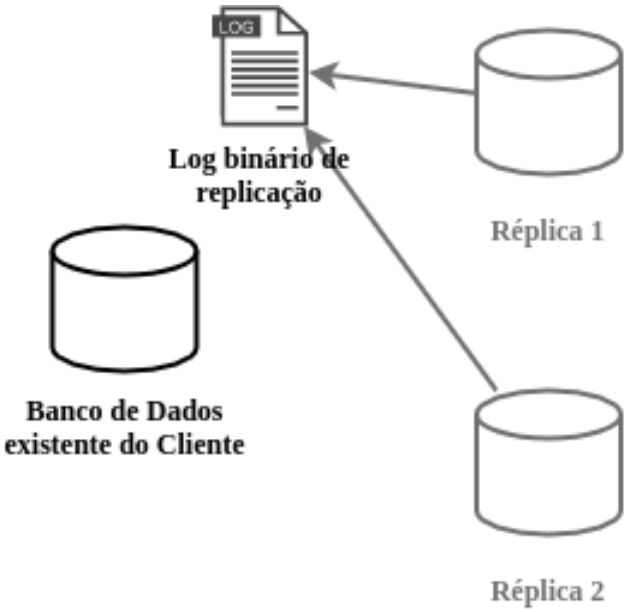
Em servidores MySQL e MariaDB, a função de replicação gera logs binários (Figura 24) que contêm arquivos que armazenam todas as operações realizadas no banco de dados (operações de *delete, insert, replace, create table, drop table, grant e revoke*). O conteúdo dos logs é escrito em SQL em formato binário e serve para replicar todo o SGBD por questões de disponibilidade e de back-up sincronizando tudo que ocorre no banco de dados com suas réplicas (ROKHMAH, 2020).

Para ativar o log de replicação de dados MySQL e MariaDB é necessário alterar seu arquivo de configuração, em alguns casos essa opção já vem habilitada por padrão, dependendo do canal de distribuição da versão do banco de dados.

Código 3.3 – Alterações no arquivo de configuração de SGBDs MySQL e MariaDB para ativação do log binário de replicação de dados

```
log_bin = /var/log/mysql/mysql-bin.log
expire_logs_days = 1
max_binlog_size = 10M
binlog_format = row
```

Figura 24 – Replicação de Banco de Dados por meio de logs



Fonte: Elaborado pelo autor

A configuração apresentada no Código 3.3 habilita o log binário no formato utilizado pelo *SQLStreamify*. Além disso são configurados os tamanhos e prazos máximos para expiração dos dados e dessa forma o log é rotacionado, isto é, são excluídos os mais antigos para que não ocupe toda a capacidade de armazenamento.

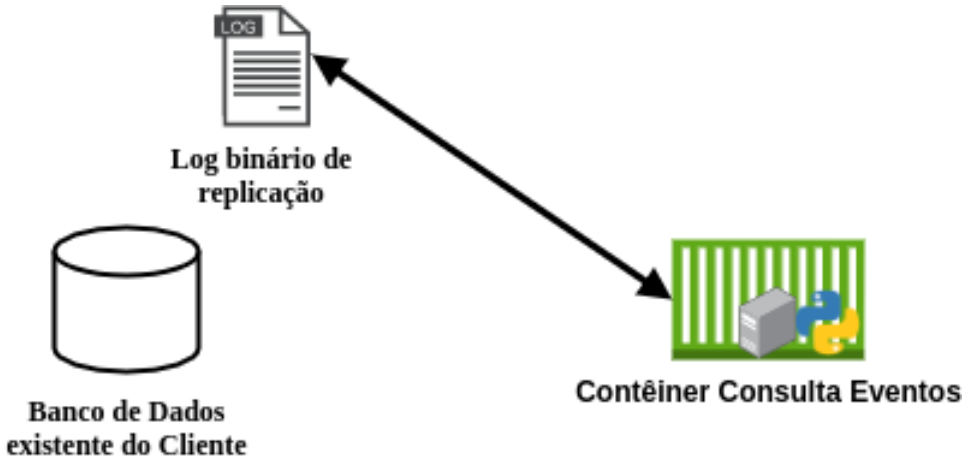
Com acesso a esses logs é possível verificar alterações de forma mais eficaz e sem bloqueio de nenhuma janela de dados do SGBD. É realizada uma checagem se um conjunto específico de dados, que coincidem com as consultas, foram alterados (Figura 25).

Como as consultas cadastradas no *SQLStreamify* são em linguagem SQL, é necessária a identificação das tabelas e condições afetadas pela consulta. Foram utilizadas funções da biblioteca *Moz SQL Parser* (MOZILLA, 2020) da fundação Mozilla, cujo objetivo é converter de SQL para uma estrutura de árvore em formato JSON identificando e separando as tabelas e condições da consulta. Essa biblioteca tem se mostrado eficiente em comparação a outros parsers (ZENG et al., 2020; LEE, 2019). Além dessa tarefa, a biblioteca foi utilizada para validação sintática das consultas em linguagem SQL.

Código 3.4 – Exemplo da conversão de uma consulta SQL para formato JSON utilizando a biblioteca Moz SQL Parser

```
# SELECT * FROM dual WHERE a>b ORDER BY a+b
{
    "select": "*",
    "from": "dual",
    "where": {"gt": ["a", "b"]},
    "orderby": {"value": {"add": ["a", "b"]}}
```

Figura 25 – Verificação de log de replicação do Banco de Dados para verificação de alterações nos dados



Fonte: Elaborado pelo autor

```
}
```

No Código 3.4 é exibida a conversão realizada pela ferramenta Moz SQL Parser. Cada parte da sentença é atribuída a um objeto com o mesmo nome. As expressões também são objetos mas apenas com uma propriedade, o nome da operação e os valores num array como parâmetros da operação (no exemplo `{"gt": ["a", "b"] }`).

De posse dos nomes das tabelas e das condições da consulta é o momento de analisar os eventos conforme ocorrem no banco de dados por meio do log para checar se ocorrem eventos que coincidam com a consulta. Essa ação é realizada por um contêiner rodando um código em Python para cada consulta cadastrada no sistema. A propriedade de isolamento de contêineres possibilita que cada consulta tenha um contêiner agindo como um servidor de replicação diferente, analisando os logs em busca do que coincida com a consulta atribuída a ele, já filtrando as tabelas que fazem parte da consulta no log binário.

3.2.3.1 Contêineres atuando como servidores de replicação de dados

Pela capacidade de no *SQLStreamify* poderem ser incluídas inúmeras consultas, cada uma gerando um fluxo de dados diferentes e como cada consulta necessita de dados de diferentes tabelas, a busca das alterações pelo log de replicação requer que para cada consulta se tenham diferentes parâmetros para as verificações. No log binário de alguns bancos é possível realizar uma pré-filtragem selecionando as tabelas que serão retornadas pelo log, uma vez que é possível replicar apenas algumas tabelas pela função de replicação dos bancos.

Tendo essa capacidade de filtragem de tabelas no log, para cada consulta é instan-

ciado um contêiner que atua como um servidor de replicação, utilizando como parâmetro de filtragem apenas os eventos das tabelas afetadas pela consulta. O processamento do log fica dividido, aumentando o desempenho de cada consulta ativa cadastrada, diminuindo a sobrecarga de uma única análise do log para todas as consultas.

3.2.4 Comparação entre os métodos

No Quadro 7 são apresentadas características importantes para escolha e desenvolvimento das técnicas de detecção de alterações nos dados.

Quadro 7 – Comparativo entre os métodos de detecção de alteração no banco de dados

Método	Bloqueio da janela de dados	Necessidade de alterações na estrutura da base de dados
Consultas Repetitivas Terry et al. (1992)	Sim	Não
Detecção utilizando triggers (subseção 3.2.1)	Não	Sim
Verificação log binário utilizando contêineres (SQLStreamify)	Não	Não

As propriedades desejáveis para o método de detecção de alterações foram definidas na concepção do *SQLStreamify*. Uma delas é o não bloqueio da janela de dados ao se realizar a detecção, visto que isso tem um grande impacto na performance das aplicações (apresentado na seção 4.1). Outra propriedade desejada é que não fossem necessárias alterações na estrutura da base de dados, uma vez que um *middleware* deve realizar suas operações sem grandes alterações nas soluções já existentes.

Como observado o método desenvolvido pela detecção de alterações pelo log binário com uso de contêineres atuando como servidores de replicação (subseção 3.2.3.1) se mostrou a melhor opção por ter as qualidades desejadas e que são objetivos deste trabalho.

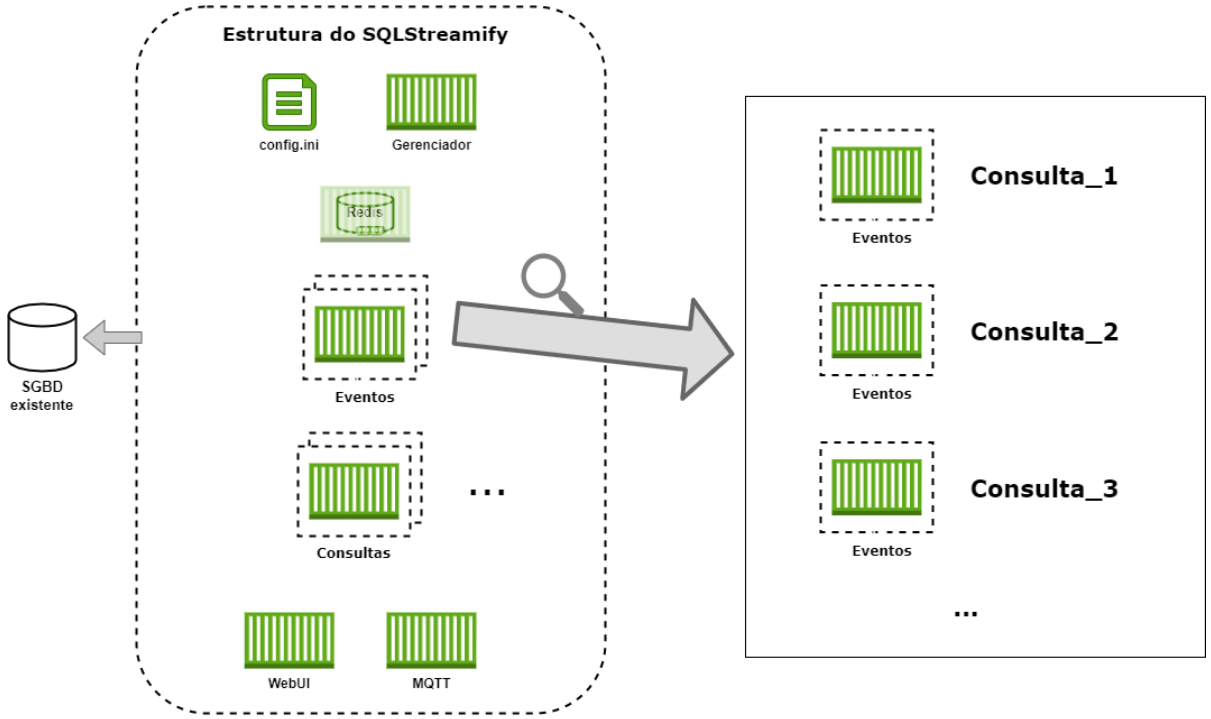
3.3 Escalabilidade

Por utilizar contêineres, é possível replicar a quantidade de instâncias necessárias para atender requisições feitas ao middleware sem sobrecarga ou atraso nas respostas, em uma estrutura que pode ser montada localmente, em *clusters* em nuvens ou até mesmo de forma mista. Essa escalabilidade atende a requisitos das mais variadas fases de projetos, que em suas fases iniciais não despendem de grandes orçamentos, nem têm grandes demandas e conforme aumentam os acessos, a estrutura pode ser elevada tornando-a mais robusta e capaz de atender um maior número de requisições.

Por separar as funções em microsserviços bem definidos que trabalham de forma isolada mantendo a comunicação com os outros microsserviços tem-se a possibilidade de conseguir aumentar ou diminuir o número de instâncias que fazem a mesma função.

Por exemplo, no caso da verificação das alterações nos dados afetados pela consulta, é possível aumentar o número de instâncias do microsserviço de modo que tenha um para cada consulta ativa cadastrada. Dessa forma, os requisitos de hardware para execução se moldam de acordo com as necessidades do usuário (Figura 26).

Figura 26 – Escalabilidade do microsserviço de Verificação de Eventos de acordo com o número de consultas cadastradas, um microsserviço atuando como um servidor de log para cada consulta



Fonte: Elaborado pelo autor

Não só o microsserviço de verificação, como também os de consulta aos dados e os de publicação podem ser escalados caso se faça necessário. É possível que seja executada mais instâncias de um microsserviço, aumentando o desempenho de acordo com as possibilidades de processamento e memória.

A escalabilidade na execução do *SQLStreamify* é mais um diferencial da proposta, uma vez que é possível sua execução com poucas consultas e baixo consumo de recursos, assim como a execução de várias consultas ativas, caso seja necessário, com consequente aumento de consumo.

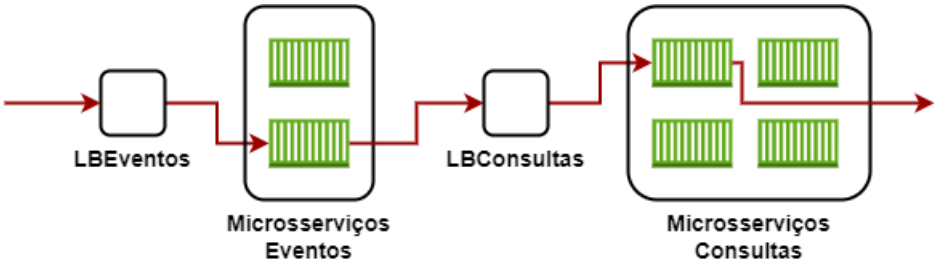
3.3.1 Balanceamento de carga

Em consonância com o fator de escalabilidade está a necessidade de um balanceamento de carga, isto é, que sejam alternadas as tarefas entre as instâncias disponíveis de cada microsserviço.

O problema do balanceamento de carga é estudado nos mais variados contextos, desde aplicações web ou em cloud, até computação paralela e balanceamento de carga em redes de computadores, entre microsserviços entre outros (Yu et al., 2019; Niu; Liu; Li, 2018).

No Quadro 8 são apresentados os microsserviços e as estratégias utilizadas para controlar a carga recebida por cada microsserviço. São utilizados balanceadores de carga HTTP que alternam o fluxo de requisições entre as instâncias dos microsserviços (Figura 27).

Figura 27 – Visão geral da estratégia de balanceamento de carga em arquiteturas baseadas em microsserviços



Fonte: Elaborado pelo autor

No código as requisições HTTP são feitas aos *Load Balancers* referentes a cada microsserviço e esse é responsável por repassar a solicitação a uma das instâncias ativas para que seja realizada a tarefa.

Quadro 8 – Estratégias de Escalabilidade utilizadas

Microsserviço	HTTP Load Balancer	Background Workers
Gerenciador		X
Eventos	X	X
Consulta	X	X
WebUI		X

3.4 Publicação e consumo de fluxo de dados

O modelo que melhor atende a disponibilização dos dados por meio de fluxos em Tempo Real são técnicas que usam o paradigma *publish/subscribe*. Nelas, clientes assinam um determinado tópico e recebem de imediato as mensagens na forma de notificações

sempre que acontecem eventos relacionados ao tópico. Os eventos neste caso seriam as alterações ou inclusões de dados no SGBD.

Para o formato das mensagens é utilizado o JSON que é amplamente difundido em variadas linguagens de programação. É derivado dos literais da linguagem de programação *JavaScript*, e embora seja um subconjunto da linguagem, não é uma linguagem de programação, mas um formato para troca de dados. Surgiu como uma alternativa ao XML e ganhou popularidade pela sua facilidade de implementação (SMITH, 2020).

3.4.1 Modos de publicação de consultas ativas

No estudo dos modos de disponibilização dos fluxos de dados foram encontradas duas situações. Na primeira, todo o dataset resultante da consulta SQL seria de interesse das aplicações quando houvessem atualizações ou inserções nos dados. Na segunda, apenas os novos dados inseridos ou atualizados são interessantes para a aplicação. São duas situações específicas que devem ser definidas de acordo com a lógica encontrada para a solução dos problemas pelos desenvolvedores. Conhecendo as situações, foram desenvolvidas duas alternativas com a possibilidade da escolha pela configuração do *SQLStreamify* na criação da consulta ativa, o modo *One-at-time* e o modo *Full-dataset*.

3.4.1.1 Modo *One-at-time*

O modo *One-at-time* (um por vez) é a forma mais lógica de envio de um fluxo de dados. Sempre que houver alterações, são enviadas uma a uma (Figura 28).

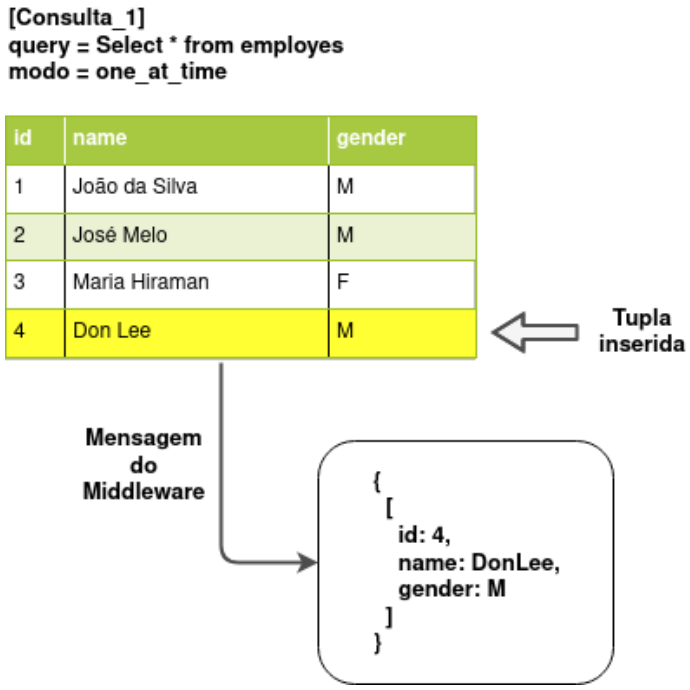
3.4.1.2 Modo *Full-dataset*

De acordo com a semântica da situação, o desenvolvedor deve optar pelo modo como os dados serão enviados. O Modo *Full-dataset* recebeu esse nome pelo fato da necessidade de enviar todo o resultado da consulta cadastrada pelo usuário sempre que houverem alterações (Figura 29). Alguns casos que podem necessitar desse modo são consultas nas quais há a sumarização de dados, como um contador, agrupamentos, cálculo de dados. Outro caso que requer esse modo são listagens ordenadas e que limitam a busca, como por exemplo, as dez últimas vendas para determinado estado. Neste exemplo de listagem sempre que houver alteração serão alterados todos os dados da consulta e dependendo do cenário da aplicação é necessário todo o dataset.

3.5 Método de distribuição dos fluxos dos dados

Utilizando o protocolo MQTT que permite a publicação de documentos ou outro formato de dados associados por tópicos, nos quais os subscritores se inscrevem e sempre

Figura 28 – Modo de publicação One-at-time: apenas os dados alterados correspondentes à consulta serão enviados



Fonte: Elaborado pelo autor

que um documento é publicado associado a esse tópico, seus subscritores recebem uma cópia desse documento (SILBERSCHATZ et al., 2010). Considerando fluxo de dados como foco, em um sistema *Publish-Subscribe* usamos as tuplas como documentos e cada busca classificamos como tópicos. Sistemas como o *Apache Kafka* usam esse modelo para o envio de atualizações de tuplas em *streams* de dados (HIRAMAN; M.; C., 2018).

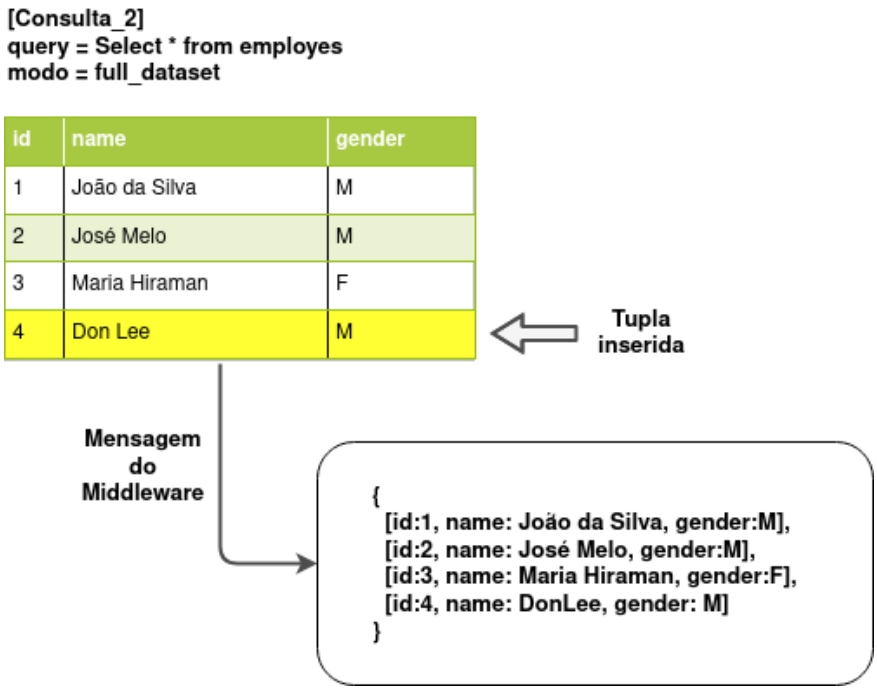
Para o *SQLStreamify* foi utilizado o RabbitMQ[6], um broker MQTT de código-aberto que possui uma distribuição específica para ser executada em contêineres. Esse microsserviço tem como função tornar público os fluxos de dados. Por ele, será publicada a resposta da consulta ativa por notificações aos assinantes por tópicos, um para cada busca, para que as aplicações clientes façam uso desses dados. Utilizando MQTT, o custo computacional para atender poucos ou um número grande de assinantes é praticamente o mesmo. Os eventos são disparados para quem estiver "assinando" cada consulta em forma de tópicos, nomeados com o mesmo nome dado no arquivo de configuração inicial.

3.6 WebUI - Interface de gerência

O microsserviço webUI disponibiliza na porta 8080 a interface de gerenciamento do sistema. A interface é construída em HTML5, Flask, Javascript e utilizando o framework

[6] RabbitMQ - <https://www.rabbitmq.com/>

Figura 29 – Modo de publicação Full-Dataset: todo o dataset correspondente a consulta ativa é retornado quando houver alteração nos dados



Fonte: Elaborado pelo autor

Bootstrap, e é responsável por gerenciar as consultas e os dados. Nela são apresentadas as consultas cadastradas e as atualizações dos dados, conforme ocorrem.

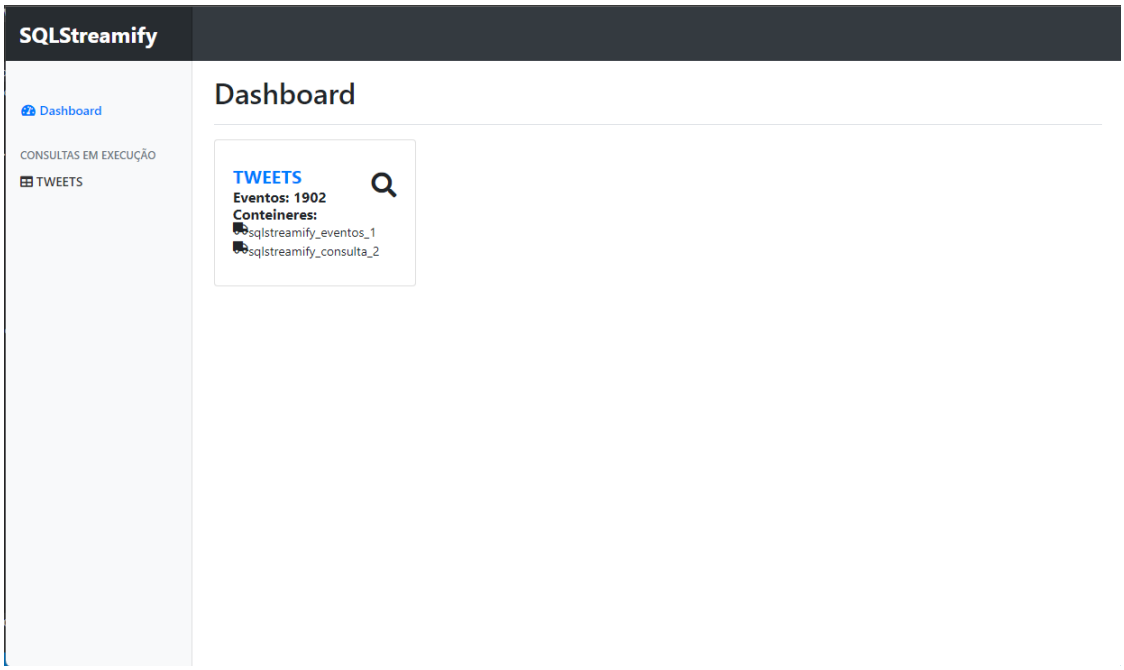
Na página inicial da interface de gerenciamento (Figura 30) é apresentada uma visão geral das consultas cadastradas. É apresentado um resumo com o número de eventos que já foram entregues desde que a consulta foi criada. Ainda é exibido em qual instância do microsserviço de consulta e de verificação de eventos que a busca foi executada.

Clicando na consulta, é direcionado para uma visão mais específica (Figura 31). Nela são informados o modo da consulta, se *One-at-time* ou *full-dataset*. O SQL que está configurado para realizar a consulta. Em performance é exibido uma métrica de quantos eventos são entregues por minuto.

Abaixo ainda é possível verificar uma tabela com os dados conforme são entregues, ela é montada em tempo real. No caso de consultas *one-at-time* os dados são inseridos na tabela a medida que são disponibilizados. Já no *full-dataset* a tabela é atualizada sempre que um evento é entregue, pois todos os dados são entregues a cada atualização.

Na tabela de mensagens é possível verificar as mensagens que são enviadas aos subscritores pelo protocolo MQTT em formato JSON.

Figura 30 – Dashboard do SQLStreamify - visão geral

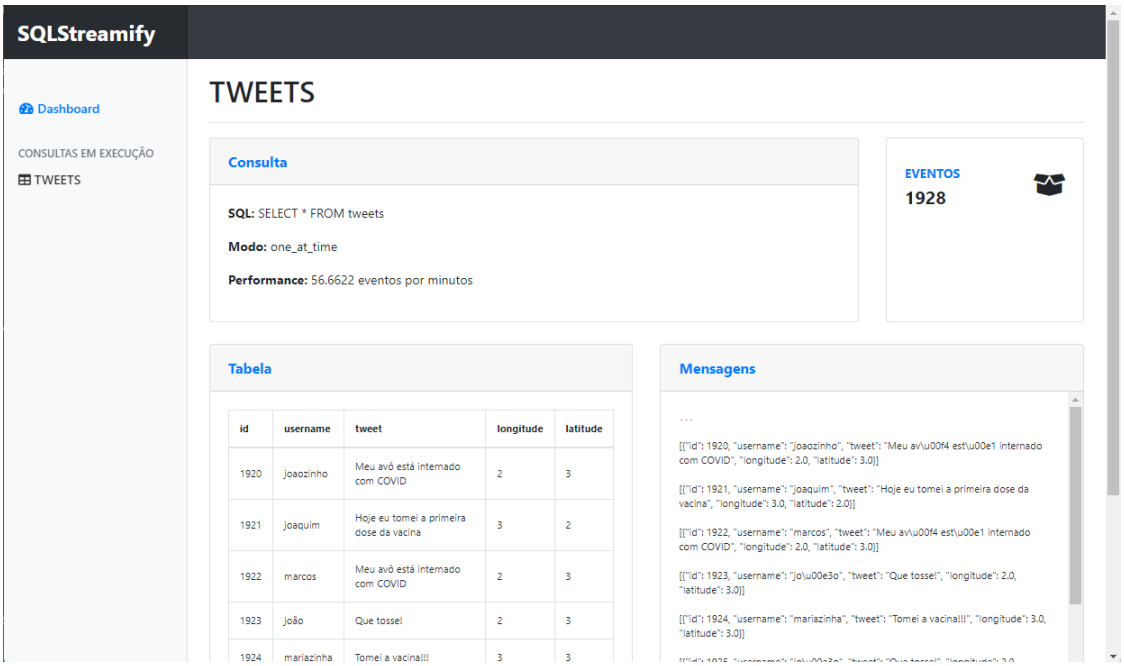


Fonte: Elaborado pelo autor

3.7 Documentação e guia de uso

O manual do *SQLStreamify* está disponibilizado com os instaladores e instruções básicas de uso. Estão em um repositório do *Github* sob uma licença de código-aberto para a disponibilização aos interessados (Apêndice A). O código está detalhadamente documentado para auxiliar na alteração e manutenção do código por colaboradores.

Figura 31 – Dashboard do SQLStreamify - parte da visualização de consulta



Fonte: Elaborado pelo autor

4 Testes e avaliações de desempenho

Neste capítulo são apresentados testes de *benchmark* realizados. *Benchmark* são ferramentas utilizadas para auxiliar na escolha de melhores métodos e/ou sistemas para determinadas ações. Seu objetivo é reportar comparativamente a qualidade e o desempenho em diferentes cenários, levando em consideração suas características e limitações.

Segundo Bog (2015), um *benchmark* deve cumprir quatro requisitos:

- relevância: as métricas coletadas devem auxiliar na comparação entre diferentes situações dentro de um domínio específico;
- portabilidade: deve ser facilmente adaptável a diferentes sistemas e arquiteturas;
- escalabilidade: deve ser capaz de medir o comportamento do sistema considerando diferentes volumes de dados;
- simplicidade: os resultados devem ser facilmente interpretáveis.

São reproduzidos testes para simular diferentes situações para auxílio no desenvolvimento do *SQLStreamify*, na criação dos métodos utilizados e nos ajustes para um melhor desempenho e escalabilidade da ferramenta. Foram realizados testes para verificar o desempenho do método de busca por alterações nos SGBDs utilizando o log binário e a execução do *SQLStreamify* em diferentes situações.

4.1 Comparação entre métodos de verificação de alteração de dados

O primeiro teste tem como objetivo comparar a carga sobre o SGBD gerada pelos métodos para a verificação de alterações nos dados. Além disso, busca-se verificar a ocorrência de bloqueios de janelas de dados durante a execução das soluções.

Foi criada uma base de dados de simulação onde são inseridos dados criados aleatoriamente de forma contínua, em um intervalo que varia de 0,1s a 0,9s entre cada inserção. Neste teste é medido o tempo de execução de cada uma das inserções como parâmetro de desempenho do SGBD. O esquema da base de dados para simulação é extremamente simples com uma tabela com duas colunas, sendo uma com um identificador e outra com valores gerados aleatoriamente.

O banco de dados utilizado foi o MariaDB sendo executado em contêiner. A escolha pela execução do SGBD em contêiner se deve a possibilidade de limitar recursos de processamento, memória, disco, de modo a tornar os testes de execução iguais para todos

os diferentes cenários. A montagem em contêineres possibilita a alteração dos arquivos de configuração e realização dos testes sem a preocupação de "quebrar" a instalação do SGBD da máquina do usuário. Além da vantagem de não ser necessária a instalação de um SGBD localmente para execução dos testes. Esse isolamento permite posterior exclusão sem resíduos da instalação no sistema operacional. Além disso, existe a possibilidade de portar e remontar o ambiente de testes completo em diferentes máquinas.

São realizadas as comparações entre três diferentes cenários:

- Apenas benchmark: nele são inseridos os dados no SGBD sem nenhuma consulta sendo realizada concomitantemente às inserções;
- *SQLStreamify*: com inserção dos dados e execução do *SQLStreamify* em paralelo;
- Consultas repetitivas: com inserção dos dados e execução do método de consultas repetitivas (apresentado na subseção 3.2.2).

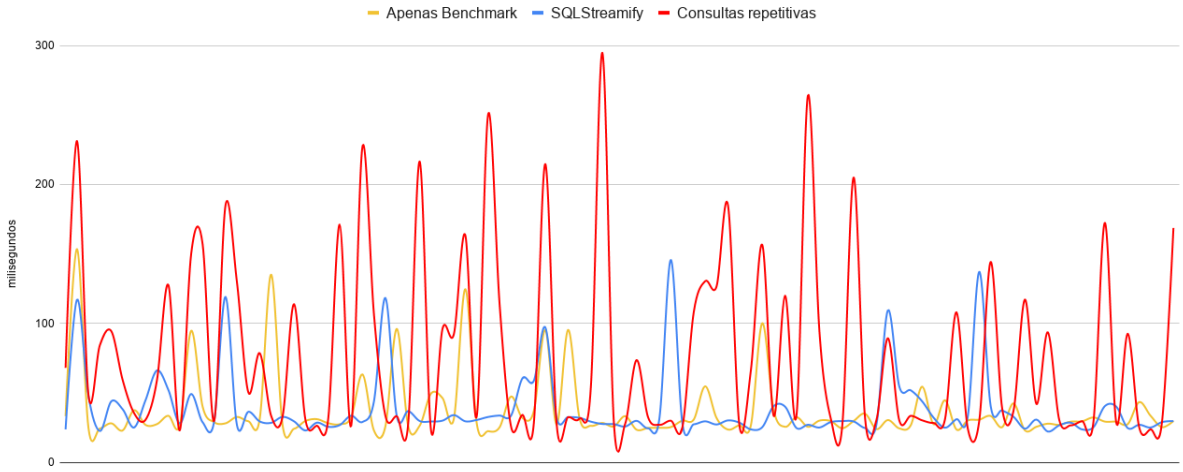
Neste teste é simulada uma situação de inserção contínua de dados com intervalo de tempo baixo e variado, com a execução de apenas uma consulta ativa extremamente simples. Para estressar o ambiente de testes foi reduzido o poder de processamento do contêiner que executa o SGBD, de modo a alcançar alguma sobrecarga do sistema. Além disso, é necessário que as inserções levem um tempo, em milissegundos, considerável para estudos e comparações.

O tempo de execução das inserções de registros no SGBD serve como parâmetro de desempenho uma vez que o SGBD é sobrecarregado com as consultas, geração de logs binários e com as próprias inserções. O tempo de execução das inserções aumenta a medida que o SGBD é sobrecarregado. Foram tomadas 100 medições de tempo em cada um dos cenários.

Podem ser observados no gráfico da Figura 32 os picos nos tempos de execução das inserções, sendo mais frequentes e com valores mais altos no cenário das Consultas Repetitivas. Isto é resultado do consumo de recursos ser maior neste método, uma vez que as buscas devem ser frequentemente executadas sobrecarregando o SGBD. A simulação foi em um banco da dados simples com apenas uma tabela e sem consultas complexas, isto é, sem junção de dados e com tempo de resposta baixo. Um ponto importante a ressaltar é que os experimentos foram realizados separadamente e as tomadas de tempo foram levadas em consideração para a montagem do gráfico.

Na Tabela 1 são apresentados os tempos mínimos, máximos e a média dos tempos de cada um dos cenários dos testes e foi plotado um gráfico na Figura 33 para melhor visualização dos dados. A diferença entre as médias reforça o tempo de execução maior no método de Consultas Repetitivas, mais que o dobro (2,10 vezes maior) da situação

Figura 32 – Gráfico do tempo de inserção (em milissegundos) em SGBD executando os métodos de verificação de alteração dos dados



Fonte: Elaborado pelo autor

Tabela 1 – Sumarização dos tempo de execução, em milissegundos, (mínimo, máximo e média) das inserções nos testes realizados em diferentes cenários

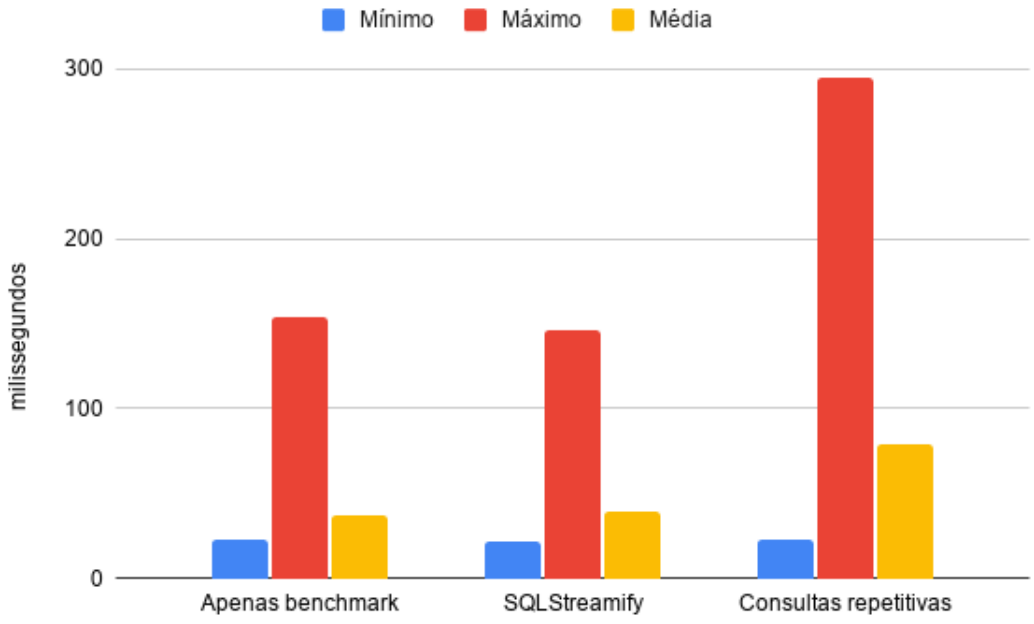
	Tempo de execução (ms)		
Cenário	**Mínimo**	**Máximo**	**Média**
Apenas benchmark	22,66144753	153,5997391	37,44094187
SQLStreamify	22,35460281	145,7798481	38,96534706
Consultas repetitivas	23,10538292	294,7573662	78,94847831

onde é executado apenas o benchmark. Já os tempos do *SQLStreamify* e do benchmark ficaram relativamente próximos, mesmo em uma simulação na qual os dados eram inseridos continuamente em intervalos de tempos baixos, fazendo com que o *SQLStreamify* fosse constantemente acionado.

Outro fato que pôde ser verificado durante o desenvolvimento e execução dos testes foi a elevada carga de processamento que o método das consultas repetitivas utiliza. Neste método é utilizado todo o processamento da máquina dependendo da janela de tempo entre cada uma das consultas, independente de alterações nos dados. Quanto menor a janela de tempo entre cada consulta repetitiva, maior a carga de processamento.

O método de buscas pela consulta do log binário por contêineres atuando como servidores de replicação de dados, utilizado pelo *SQLStreamify* se mostrou mais eficiente e utilizou menos processamento para identificação das alterações dos dados, além de não sobrecarregar o SGBD com consultas.

Figura 33 – Gráfico do tempo de inserção (em milissegundos) em SGBD executando paralelamente métodos de verificação de alteração dos dados



Fonte: Elaborado pelo autor

4.2 Consumo de recursos pela ativação do log binário

Os testes desta seção são realizados com o objetivo de verificar o consumo de recursos com a ativação do log binário pelo SGBD. Esse dado é importante, uma vez que o *SQLStreamify* atua como um servidor de replicação utilizando o log binário como método para checar quando os dados consultados forem modificados.

Para os testes é utilizado o MariaDB sendo executado em contêiner, isso possibilita a medição dos recursos utilizados pelos processos do SGBD de forma isolada e permitem a remontagem para cada teste sem que nenhuma execução anterior vicie ou destoe as medições seguintes.

Para esses testes foram criados quatro diferentes situações:

- Log binário ativado - gravando dados continuamente e lendo o log binário;
- Log binário ativado - gravando dados continuamente sem nenhuma leitura e nenhum servidor de replicação lendo os logs binários;
- Log binário desativado - gravando e lendo dados continuamente;
- Log binário desativado - apenas gravando dados sem nenhuma leitura.

Para gravação e leitura dos dados aleatórios foi utilizado o *sysbench*[1] como ferramenta de *benchmark*. Para a preparação foram passados os parâmetros para a criação de apenas uma tabela contendo um milhão de registros, essa geração de registros foi realizada em cada um dos cenários do teste.

Depois de preparado o ambiente de testes foi executado o *benchmark oltp-read-write* do *sysbench*. Este teste realiza operações de leitura e escrita no SGBD para comparação de desempenho.

Foi configurado para gerar os dados aleatoriamente e realizar operações de escrita e leitura em apenas uma tabela na maior taxa de operações possível, isto é, não foi limitado o número de operações. O número de registros da tabela ficou limitado em um milhão de registros, o mesmo utilizado na preparação da tabela. Isto impede que a tabela aumente de forma aleatória com a execução do teste e varie a utilização de memória em cada um dos cenários.

No gráfico da Figura 34 são apresentados os dados do consumo de memória, em MB, pelo SGBD. Observando o gráfico, é possível identificar três fases distintas. A fase com a preparação dos dados, na qual o consumo de memória aumenta devido ao volume de dados armazenados. Depois disso uma fase de estabilização, nela não são realizadas nenhuma operação no SGBD e o consumo é mantido onde parou depois da inserção dos dados. A última fase é quando o *benchmark* é executado com as operações de leitura e escrita.

A primeira conclusão que pode ser tirada é que nos cenários em que o log binário está ativado o consumo de memória é maior do que quando está desativado. Conclui-se também que as operações de leitura concomitantes à execução do *benchmark* levam a um consumo maior de memória pelo SGBD.

Uma última observação é a manutenção e estabilização do consumo de memória. Isto se deve ao fato de que o número máximo de registros para a tabela é mantido sempre em um milhão, passado por parâmetro na execução do *benchmark*.

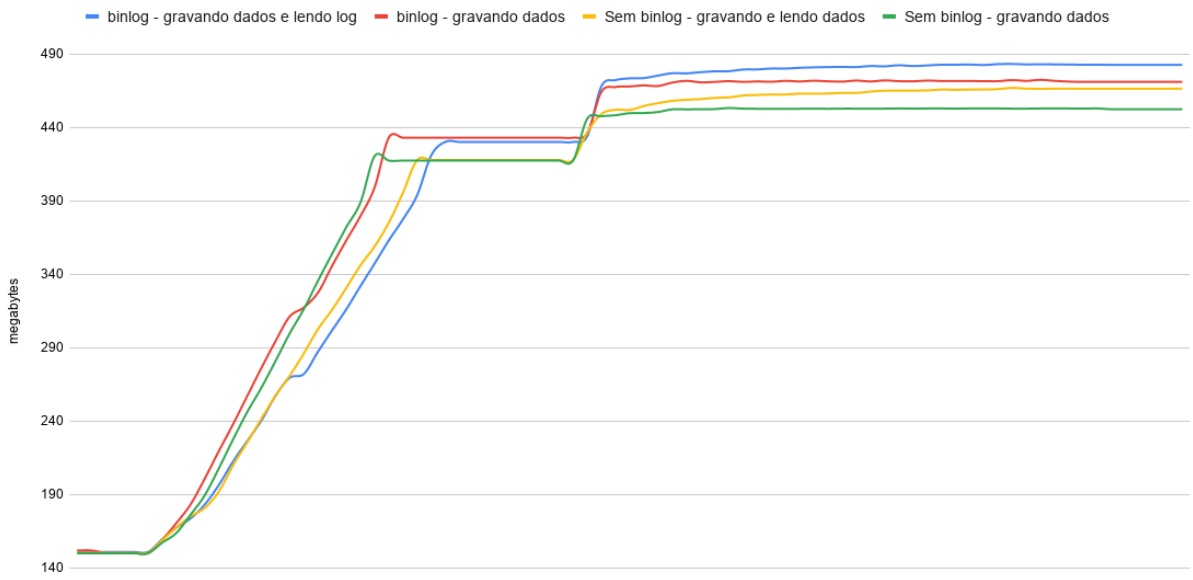
4.3 Escalabilidade na execução do log binário

Foi realizado mais um teste com o objetivo de mostrar o quanto o aumento do número de tabelas e consequente aumento de dados no SGBD influencia no consumo com a ativação do log binário.

A diferença entre a ativação ou não do log binário no consumo pôde ser notada no teste anterior (seção 4.2), mas é necessário verificar o quão grande é o aumento do

[1] sysbench - um dos mais populares benchmark de código aberto para testes em SGBDs - <https://github.com/akopytov/sysbench>

Figura 34 – Gráfico do consumo de memória(Mb) pelo SGBD em diferentes situações



Fonte: Elaborado pelo autor

consumo de memória dependendo do número de tabelas e dados armazenados.

No teste foi executado o *sysbench* aumentando os parâmetros para o número de tabelas e o número de registros. O SGDB utilizado foi o MariaDB sendo executado em contêiner pelos motivos citados nos testes anteriores.

Depois da estabilização que pôde ser observada no teste anterior (seção 4.2), foram registrados os consumos de memória nas seguintes situações: o número de tabelas variando de 1 a 4, cada uma com um milhão de registros, com o log binário ativado e com o log binário desativado.

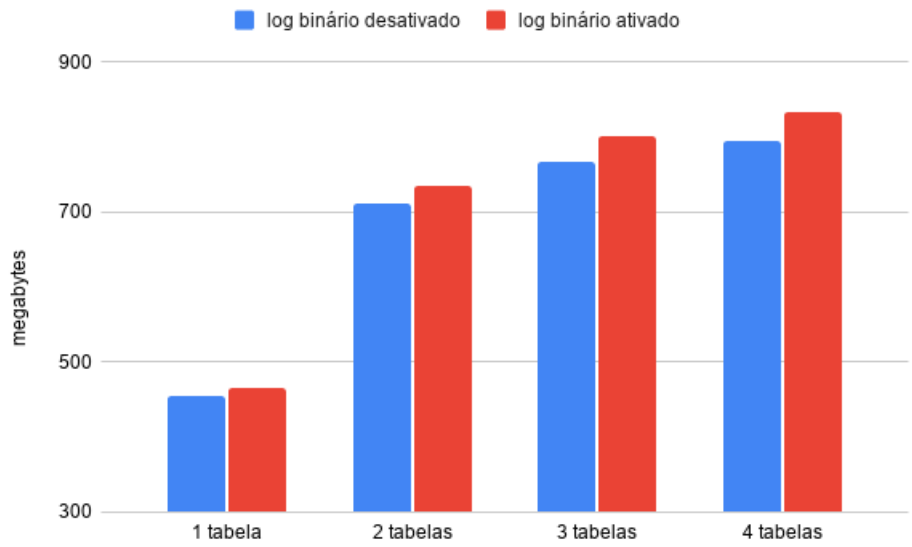
Os dados são apresentados na Tabela 2 e em forma de gráfico na Figura 35.

Tabela 2 – Consumo log binário

	log binário desativado	log binário ativado
1 tabela	454 Mb	465 Mb
2 tabelas	711 Mb	735 Mb
3 tabelas	767 Mb	802 Mb
4 tabelas	794 Mb	833 Mb

Pode ser notado que o aumento no consumo de memória pelo SGBD não é proporcional ao número de dados armazenados. Outra conclusão que pode ser observada é que a diferença entre a ativação ou não do log binário é relativamente pequena, em um cenário com um grande número de operações sendo executadas no SGBD. Para o cenário

Figura 35 – Gráfico do consumo de memória(Mb) pelo SGBD com a ativação do log binário com diferentes números de tabelas



Fonte: Elaborado pelo autor

Tabela 3 – Porcentagem do consumo com a ativação do log binário

	% do consumo de memória com a ativação do log binário
1 tabela	2,365591398
2 tabelas	3,265306122
3 tabelas	4,364089776
4 tabelas	4,681872749

de 1 tabela com um milhão de registros é de apenas 11Mb, no cenário com 2 tabelas e 2 milhões de registros é de 24Mb, no de 3 tabelas é de 35Mb e no de 4 tabelas é de 39Mb. Esses dados levam a montagem da Tabela 3 na qual são apresentadas as porcentagens do consumo de memória RAM com a ativação do log binário em cada um dos cenários.

Pode ser notado que o consumo do log binário variou de 2,3% a 4,6% de memória em cada cenário do teste.

5 Estudos de caso

Esta seção tem como objetivo apresentar aplicações práticas desenvolvidas utilizando a ferramenta como provas de conceito das funcionalidades e funcionamento do *SQLStreamify*. Foram desenvolvidas duas ferramentas, uma para monitoramento de ativos de redes em tempo real e outra para monitoramento de reações em redes sociais em tempo real.

5.1 Monitoramento em tempo real de ativos de redes

Uma tarefa muito importante de administradores de redes é o monitoramento dos equipamentos que compõem a rede. É importante ter informações como tráfego, disponibilidade, temperatura, processamento, utilização de discos, entre outras tantas informações que podem ser obtidas.

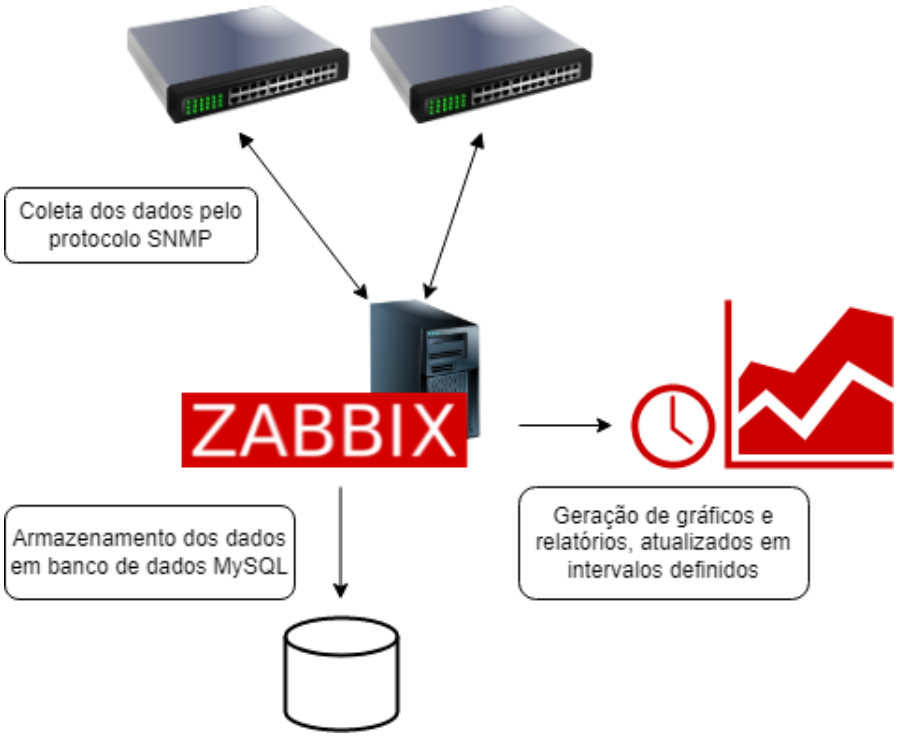
Com o uso desses dados pode ser realizado o diagnóstico de problemas, consumo abusivo de recursos, identificar pontos de gargalos na troca de tráfego entre links, tentativas de invasão, uso indevido da rede e outras situações. Podem ser configurados alertas quando determinada situação for identificada, como um aumento no tráfego fora dos padrões considerados usuais para a rede. Outro uso bem comum é o de gráficos com essas informações para *insights* visuais e montagem de relatórios.

Devido a composição das redes de computadores com os mais diversos tipos de equipamentos, de diferentes fabricantes foi criado o protocolo SNMP (*Simple Network Management Protocol* - Protocolo Simples de Gerência de Rede) que permite a gerência facilitada, incluindo padronização na disponibilização de métricas dos equipamentos, próprio de troca de mensagens de status entre dispositivos de rede.

Para o desenvolvimento foi utilizado o banco de dados de uma das mais utilizadas ferramenta de monitoramento de ativos de redes, o *Zabbix*, que tem código aberto. Esta base de dados é constantemente atualizada e são armazenados os dados conforme capturados pelo protocolo SNMP. Esses dados são rotacionados, isto é, os mais antigos são excluídos, dependendo da estrutura da rede pois podem chegar a um volume relativamente grande de informações.

O *Zabbix* faz o armazenamento e disponibiliza gráficos que são gerados em um intervalo de tempo determinado. Normalmente, é configurado um intervalo baixo entre as atualizações para a geração dos gráficos e disponibilização das informações. Uma grande solicitação da comunidade de usuários do *Zabbix* é a disponibilização de gráficos em tempo real com atualização instantânea dos dados recebidos dos equipamentos (Figura 36).

Figura 36 – Modo de funcionamento do *Zabbix* com os gráficos sendo atualizados em intervalos definidos de tempo



Fonte: Elaborado pelo autor

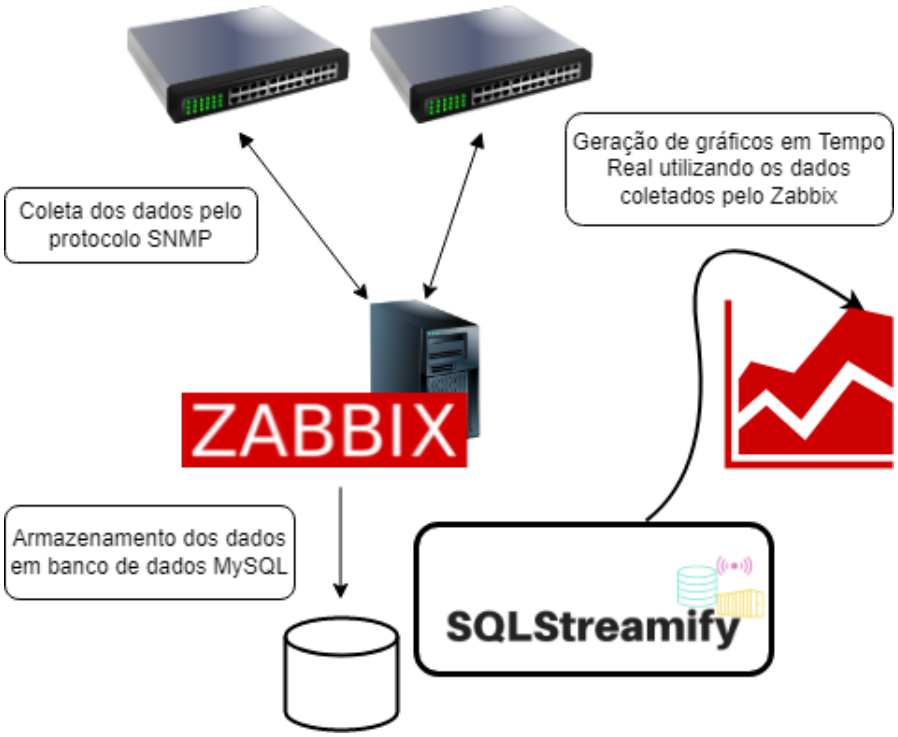
Pensando nisso foi desenvolvido como prova de conceito, uma aplicação adicional para o monitoramento em tempo real de ativos de redes, utilizando o banco de dados *MySQL* alimentado pelo *Zabbix*, utilizando o *SQLStreamify* para geração de fluxos de dados dos dados adicionados ao banco de dados gerando gráficos em tempo real (Figura 37).

5.1.1 Desenvolvimento da aplicação

Se pensar no modelo usual para o desenvolvimento da aplicação, gerando gráficos da forma que o *Zabbix* o faz, teríamos que determinar um intervalo de tempo entre as atualizações dos dados. Quanto mais baixo esse intervalo mais rápido será a disponibilização dos dados para a geração do gráfico e mais próximo de um gráfico em tempo real se teria. Porém um efeito em cascata se dá pela sobrecarga no SGBD com consultas constantes, maior o consumo de recursos e maior atraso na inclusão de novos dados, uma vez que as consultas bloqueiam janelas de dados, consequentemente tornando o "tempo real" desejado mais distante.

Utilizando o *SQLStreamify*, é inicializado o serviço com o cadastro das consultas com as informações desejadas para os gráficos a serem desenvolvidos. Tendo assim, fluxos de dados em tempo real. Essas informações podem ser utilizadas em *frameworks* e bibliotecas

Figura 37 – *SQLStreamify* atuando como fornecedor dos fluxos de dados inseridos em tempo real pelo *Zabbix* para geração de gráficos com atualizações instantâneas



Fonte: Elaborado pelo autor

de visualização de dados, como o *Highcharts* ou *D3.js* para gerar gráficos. Muitas dessas ferramentas possuem módulos desenvolvidos para consumo de fluxos de dados em tempo real. Nessa forma de consumo a aplicação funciona como uma subscritora do fluxo e sempre que um novo dado é recebido, o gráfico é atualizado em tempo real, sem necessidade de recarregar ou remontar o gráfico.

O primeiro passo foi identificar o *itemid* do item desejado para a geração do gráfico. Foi realizada uma busca pelo tráfego específico de uma porta de um dos switches da rede. Cada porta possui além do tráfego inúmeras outras informações, como taxa de erros, velocidade configurada, descrição da porta, entre outros. Essas informações são encontradas mais facilmente na tabela de itens cadastrados. Localizado o *itemid* a próxima ação foi identificar em qual tabela são gravadas as informações coletadas, o Zabbix armazena o tráfego na tabela *history-uint*, pois essa é uma informação numérica armazenada em formato inteiro.

Com essas informações foi configurada a consulta ativa nomeada *link-barrafunda-download*, o modo utilizado foi o *Full-dataset* já que a consulta busca apenas o último registro armazenado.

Código 5.1 – Arquivo de configuração do SQLStreamify para a consulta de geração de

gráficos em tempo real para o Zabbix

```
[DB]
type = mysql
host = 172.17.0.1
port = 3306
db = zabbix
user = zabbix
password = z@bb1x

[EXPOSICAO]
ip = 200.145.181.62

[link-barrafunda-download]
query = SELECT * FROM history_uint WHERE itemid=61375 ORDER BY clock DESC LIMIT 1
modo = full_dataset
```

Para o consumo do fluxo de dados gerado pelo *SQLStreamify* foi utilizada a biblioteca STOMP para Javascript. A conexão é feita ao servidor que executa o *SQLStreamify* e foi configurado em **EXPOSICAO**, a fila de mensagens tem o nome configurado na consulta ativa **link-barrafunda-download**. Sempre que um novo registro satisfaz a consulta é recebido pela aplicação e inserido no gráfico.

Para o gráfico foi utilizada a biblioteca *Chart.Js* com o *plugin RealTime*. Ele suporta geração de gráficos a partir de fluxos de dados em tempo real. A integração é bem simples e tem algumas opções de configuração como a duração das informações que são exibidas, um delay para a atualização do gráfico, e uma taxa de atualização entre os intervalos de tempo.

Código 5.2 – Código para geração de gráficos em tempo real utilizando fluxos gerados pelo *SQLStreamify* utilizando a biblioteca *STOMP* e o *Chart.js* para a geração de gráficos

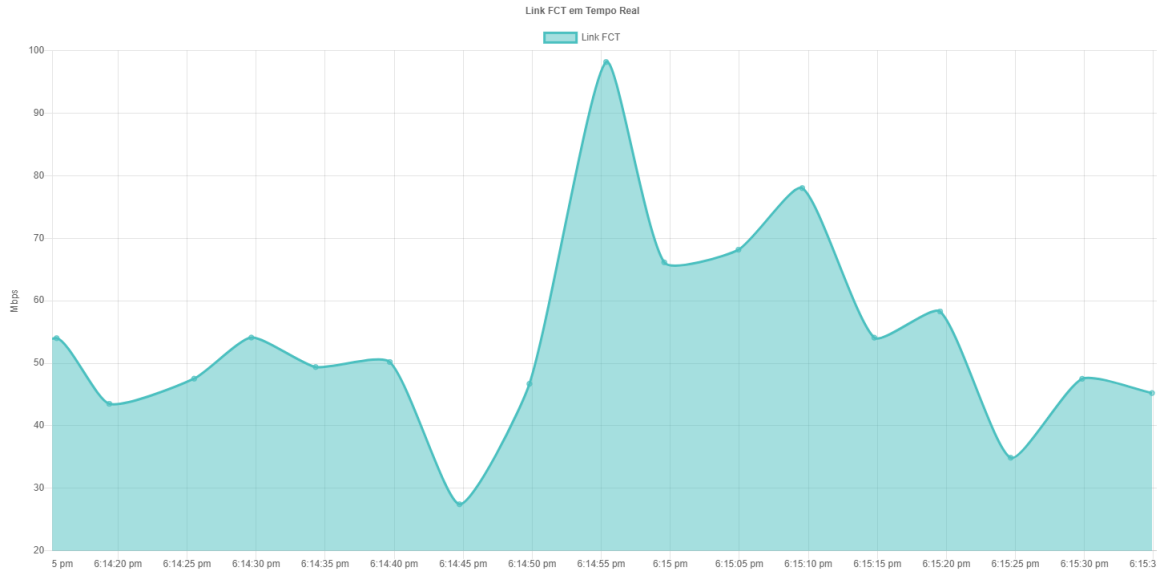
```
var client_zabbix = Stomp.client('ws://200.145.181.62:15674/ws');

var on_connect_zabbix = function(x) {
id = client_zabbix.subscribe("/exchange/link_barrafunda_download", function(d) {
        data = JSON.parse(d.body);

        myChart.data.datasets[0].data.push({
        x: Date.now(),
        y: data[0]["value"] / 1000000
        });
        myChart.update();

});
};
var on_error_zabbix =  function() {
console.log('error');
};
client_zabbix.connect(on_connect_zabbix, on_error_zabbix, '/');
```

Um código extremamente simples e muito mais eficiente que buscas constantes em intervalos de tempo para a obtenção dos dados e geração do gráfico.

5.1.2 Resultado

Como resultado final, tem-se uma aplicação sem sobrecarga no SGBD, sem atraso na inclusão de novos dados e sem bloqueio de janelas de dados. Podendo ser executada no mesmo servidor do *Zabbix* e com a possibilidade de escalar a aplicação para a quantidade de gráficos e ativos ao qual se deseja monitorar (Figura 38).

Figura 38 – Gráfico atualizado em tempo real com informações capturadas e armazenadas pelo *Zabbix* em Banco de Dados *MariaDB*



Fonte: Elaborado pelo autor

Essa visualização é utilizada frequentemente para *troubleshooting* da rede, isto é, auxilia na detecção de problemas. Quando algum cliente aumenta o tráfego consumido na rede, ao desconectarmos clientes suspeitos, instantaneamente é refletido no tráfego e rapidamente observado no gráfico. Outro uso também é na alteração da porta verificada apenas localizando o *itemid* e alterando de qual porta temos o gráfico em tempo real. Com outras portas podemos verificar gargalos na rede, abuso no uso de banda por algum cliente, portas problemáticas em switches e loops na rede. Por ser em tempo real não há necessidade de aguardar um tempo para as informações refletirem no gráfico e ações para resolver os problemas podem ser testadas e checar a solução imediatamente.

5.2 Aplicação em pesquisa sobre o monitoramento em tempo real de reações em redes sociais sobre COVID-19

Outra aplicação desenvolvida é de visualização em tempo real de reações em redes sociais, mais especificamente do Twitter sobre COVID-19.

Foi utilizado o SGDB de uma pesquisa que realiza a coleta e o armazenamento destes dados e faz uma sumarização e estudos utilizando-os.

Os dados são coletados em tempo real e armazenados em banco de dados. Utilizando esse cenário, foi desenvolvida uma ferramenta que utiliza o *SQLStreamify* disponibilizando sobre esses dados uma camada de consultas ativas e montagem de visualizações em tempo real, que podem disponibilizar *insights* interessantes e pontuais.

Figura 39 – Reações COVID19 em redes sociais

Fonte: Elaborado pelo autor

5.2.1 Desenvolvimento da aplicação

Para o desenvolvimento das visualizações em tempo real, foi utilizado o SDK (*Software Development Kit* - Kit de Desenvolvimento de Software) do Google Maps.

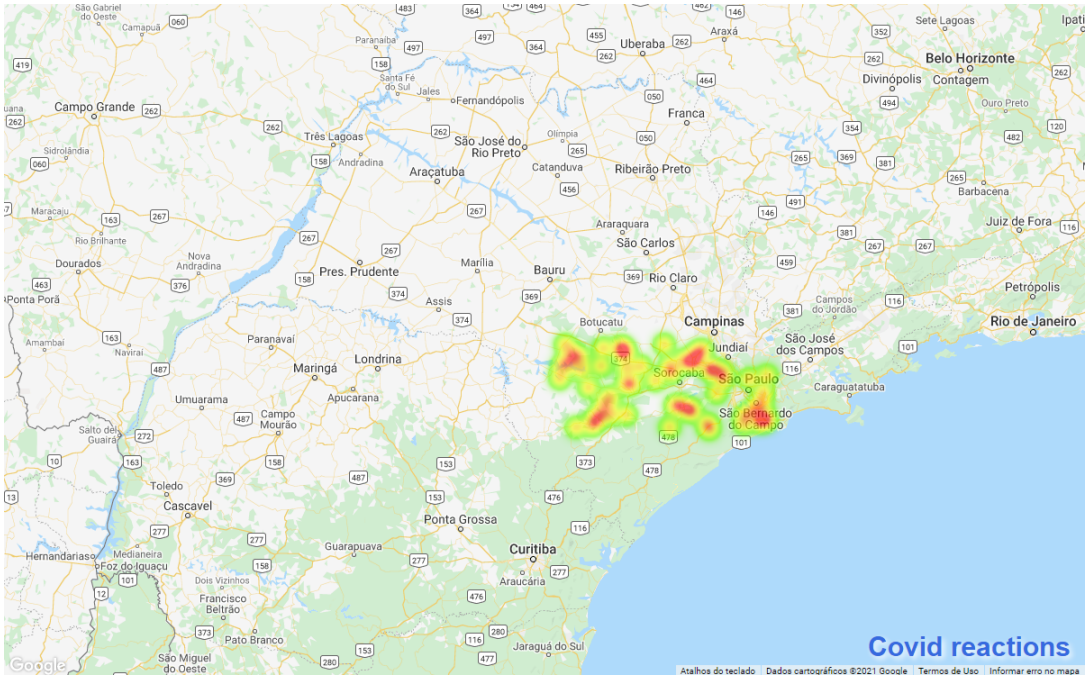
Utilizando a função de mapa de calor (*HeatMap*) são plotadas as reações com a localização geográfica dos usuários. A ferramenta da pesquisa que faz a coleta de dados busca reações buscando por palavras-chave, como: *coronavirus*, *covid-19*, *tosse*, *covid*, *vírus*, *sintomas*, *corona*, e outras relacionadas ao tema.

Para consumo do fluxo de dados é utilizada a biblioteca *STOMP* em *Javascript* preenchendo em tempo real no SDK do Google Maps. As atualizações são feitas sem a necessidade de recarregar a página e são atualizadas conforme ocorrem os eventos.

5.2.2 Resultado

A visualização gerada é útil para acompanhar reações em redes sociais sobre COVID-19, ao surgimento de notícias ou fatos que influenciem a situação. Outro fator interessante é a possibilidade de focar em diferentes regiões, aumentando ou reduzindo o zoom no mapa para uma melhor visualização ou estudo de uma região específica (Figura 40).

Figura 40 – Reações COVID-19 em redes sociais com zoom em uma região específica



Fonte: Elaborado pelo autor

Visualizações em tempo real podem ser muito interessantes para serem usadas como termômetro de reações instantâneas, como por exemplo, reações sobre determinado programa ou atividade ao vivo, como debates eleitorais, eventos esportivos e outros.

6 Considerações finais

Este trabalho foi desenvolvido a partir da constatação da necessidade de utilização de fluxo de dados em tempo real por sistemas já existentes que utilizam SGBDs tradicionais. Trabalhos apresentam o desenvolvimento de soluções pontuais para a solução desse problema, porém todas direcionadas para uma aplicação específica, outros fazem o tratamento de fluxo de dados porém sem utilizar dados inseridos em SGBDs como fonte. Por esse motivo, desenvolver um *middleware* que entregue uma solução geral para essa necessidade é o objetivo geral deste trabalho. Sua principal contribuição é a ferramenta *SQLStreamify*, *middleware* que entrega as funções de consultas ativas à SGBDs em produção, disponibilizando a solução para um *pipeline* diferente dos SGDFs à disposição.

Como um importante resultado temos o desenvolvimento do método de detecção de alterações em dados do SGBD, que foi utilizado para a construção do *SQLStreamify*. Baseado na utilização de contêineres atuando como servidores de replicação realizando a busca nos logs binários sem bloqueio da janela de dados e com baixo consumo na carga de processamento do SGBD. Os testes realizados (Capítulo 4) mostram a eficiência do método comparado com técnicas normalmente utilizadas. O método desenvolvido não prejudica as operações já existentes no SGBD e apresentou tempos de inserção paralelo a execução de em média 2.10 vezes mais rápido do que técnicas existentes para a solução do problema.

A escalabilidade na execução do *SQLStreamify* é mais um diferencial do trabalho, uma vez que é possível sua execução com poucas consultas e baixo consumo de recursos, assim como a execução de várias consultas ativas, caso seja necessário, com consequente aumento de consumo. Isso foi possível pela utilização da arquitetura baseada em microsserviços, na qual o número de instâncias de cada microsserviço escalam de acordo com o número de consultas ativas cadastradas.

Foram verificados o consumo da ativação do log binário em bancos de dados MySQL e MariaDB, sendo essa função fundamental para o funcionamento do *SQLStreamify*. Os resultados foram bem satisfatórios, com baixo consumo de memória com a ativação do log binário (entre 2% e 5% de aumento no uso de memória). Um resultado paralelo interessante obtido pela realização dos testes foi a execução dos SGBDs de experimento em contêineres, resultando em testes replicáveis, pelo isolamento e definição de recursos possibilitado pela arquitetura.

Como forma de publicação do fluxo de dados para assinatura dos clientes, foi escolhida uma distribuição do MQTT. Novamente, o fato do modelo de desenvolvimento em microsserviços mostrou-se benéfico, possibilitando a substituição dessa distribuição ou

troca para outro protocolo que melhor se adeque em diferentes situações ou para possíveis expansões futuras.

O *SQLStreamify* está disponível em formato de código aberto sob licença *Apache License 2.0* em repositório *GitHub*[1], juntamente com um guia de uso e exemplos de utilização. Por sua característica modular, usuários podem realizar alterações para novos formatos de publicação dos fluxos de dados e integração com outros modelos de SGBDs.

[1] *SQLStreamify* - <https://github.com/lelinho/SQLStreamify>

Referências

ABADI, D. et al. Aurora: a data stream management system. In: CITESEER. *SIGMOD Conference*. [S.l.], 2003. p. 666. Citado na página 40.

ABADI, D. J. et al. The design of the borealis stream processing engine. In: *Cidr*. [S.l.: s.n.], 2005. v. 5, n. 2005, p. 277–289. Citado 2 vezes nas páginas 34 e 41.

Alami, A. E.; Bahaj, M.; Khourdifi, Y. Supply of a key value database redis in-memory by data from a relational database. In: *2018 19th IEEE Mediterranean Electrotechnical Conference (MELECON)*. [S.l.: s.n.], 2018. p. 46–51. Citado na página 54.

ALBUQUERQUE, D. J. S. d. *Identificação de dificuldades e questões de interesse de desenvolvedores de aplicações para Big Data com o framework Apache Spark*. Dissertação (Mestrado) — Brasil, 2019. Citado 2 vezes nas páginas 18 e 36.

ARASU, A. et al. Stream: The stanford data stream management system. In: *Data Stream Management*. [S.l.]: Springer, 2016. p. 317–336. Citado 2 vezes nas páginas 41 e 42.

BAPTISTA, G. Processamento distribuído de eventos complexos (cep). 2014. Citado 2 vezes nas páginas 17 e 29.

BERRY, G. *Real time programming: Special purpose or general purpose languages*. Tese (Doutorado) — INRIA, 1989. Citado na página 30.

BESTAVROS, A.; LIN, K.-J.; SON, S. H. *Real-time database systems: Issues and applications*. [S.l.]: Springer Science & Business Media, 2012. v. 396. Citado na página 17.

BEZ, J. Plataformas de big data: Spark, storm e flink. *no. July*, 2015. Citado 3 vezes nas páginas 18, 36 e 38.

BOG, A. *Benchmarking transaction and analytical processing systems*. [S.l.]: Springer, 2015. Citado na página 71.

BUTTAZZO, G. C. *Hard real-time computing systems: predictable scheduling algorithms and applications*. [S.l.]: Springer Science & Business Media, 2011. v. 24. Citado 2 vezes nas páginas 22 e 23.

CALBIMONTE, J.-P. et al. Enabling query technologies for the semantic sensor web. *International Journal On Semantic Web and Information Systems (IJSWIS)*, IGI Global, v. 8, n. 1, p. 43–63, 2012. Citado na página 18.

CARVALHO, F. O. *Descoberta Contínua de Serviços em IoT*. Tese (Doutorado) — PUC-Rio, 2017. Citado na página 49.

ÇETINTEMEL, U. et al. The aurora and borealis stream processing engines. In: *Data Stream Management*. [S.l.]: Springer, 2016. p. 337–359. Citado 4 vezes nas páginas 18, 34, 40 e 41.

COSTA, C. M.; LEITE, C. R.; NETO, P. R. Desenvolvimento de um componente de software-sams: Sistema automático de monitoramento em tempo-real de sondas de produção de petróleo. In: *VIII Conferência Internacional de Aplicações Industriais-INDUSCON, Poços de Caldas, Minas Gerais*. [S.l.: s.n.], 2008. Citado 4 vezes nas páginas 27, 51, 54 e 59.

DOCKER, I. *Docker Documentation*. 2019. Disponível em: <https://docs.docker.com>. Citado 2 vezes nas páginas 46 e 47.

Eisa, I.; Salem, R. Concurrency control algorithm for shared disk cloud dbms. In: *2017 13th International Computer Engineering Conference (ICENCO)*. [S.l.: s.n.], 2017. p. 202–207. Citado na página 60.

ELMASRI, R. et al. Sistemas de banco de dados. Pearson Addison Wesley São Paulo, 2005. Citado na página 58.

EUGSTER, P. T. et al. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, ACM, v. 35, n. 2, p. 114–131, 2003. Citado na página 49.

EYERS, D. M. et al. Relational database support for event-based middleware functionality. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. [S.l.: s.n.], 2010. p. 160–171. Citado na página 34.

Feng, Y. et al. Design and implementation of real time data center access interface based on big data technology. In: *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*. [S.l.: s.n.], 2018. p. 235–239. Citado 2 vezes nas páginas 17 e 18.

FERNANDES, Y. Y. M. P. et al. Sistemas de gerenciamento de banco de dados em tempo-real na área hospitalar: Um estudo de caso para unidade de terapia intensiva. *Revista Brasileira de Inovação Tecnológica em Saúde-ISSN: 2236-1103*, 2014. Citado 5 vezes nas páginas 18, 28, 51, 54 e 59.

FERNANDES, Y. Y. M. P. et al. Técnica de controle de concorrência semântico para sistemas de gerenciamento de bancos de dados em tempo-real. Universidade Federal de Campina Grande, 2005. Citado na página 22.

FERREIRA, P. E. S. *AIoTA: an IoT Platform on MonetDB*. Tese (Doutorado) — Universidade do Minho, 2016. Citado na página 29.

FREMANTLE, P. A reference architecture for the internet of things. *WSO2 White paper*, 2015. Citado na página 28.

FRÜHWIRT, P. et al. Innodb database forensics: Enhanced reconstruction of data manipulation queries from redo logs. *Information Security Technical Report*, Elsevier, v. 17, n. 4, p. 227–238, 2013. Citado na página 48.

GHOSH, A. et al. Plan selection based on query clustering. In: ELSEVIER. *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. [S.l.], 2002. p. 179–190. Citado na página 48.

HEBRAIL, G. Data stream management and mining. *Mining massive data sets for security*, p. 89–102, 2008. Citado na página 24.

HELMY, Y. M.; ZANFALY, D. S. E.; OTHMAN, N. A. Prioritized query shedding technique for continuous queries over data streams. *International Conference on Computer Engineering and Systems*, 2009. Citado 2 vezes nas páginas 18 e 23.

HIRAMAN, B. R.; M., C. V.; C., K. A. A study of apache kafka in big data stream processing. *International Conference on Information, Communication, Engineering and Technology (ICICET)*, 2018. Citado 3 vezes nas páginas 18, 20 e 67.

HIVEMQENTERPRISE, M. *Broker. MQTT Essentials Part2: Publish & Subscribe*. 2016. Citado na página 50.

HUACARPUMA, R. C. et al. Concepção e desenvolvimento de um serviço distribuído de coleta e tratamento de dados para ambientes de internet das coisas. In: *SBBD*. [S.l.: s.n.], 2016. p. 28–39. Citado na página 28.

ISAH, H.; ZULKERNINE, F. A scalable and robust framework for data stream ingestion. *IEEE International Conference on Big Data (Big Data)*, 2018. Citado 2 vezes nas páginas 17 e 20.

KAMBURUGAMUVE, S. et al. Survey of distributed stream processing for large stream sources. *Grids Ucs Indiana Edu*, v. 2, p. 1–16, 2013. Citado na página 36.

KOPETZ, H. *Real-time systems: design principles for distributed embedded applications*. [S.l.]: Springer Science & Business Media, 2011. Citado na página 17.

LEE, D. Clause-wise and recursive decoding for complex and cross-domain text-to-sql generation. *arXiv preprint arXiv:1904.08835*, 2019. Citado na página 61.

LIGHT, R. A. et al. Mosquitto: server and client implementation of the mqtt protocol. *J. Open Source Software*, v. 2, n. 13, p. 265, 2017. Citado na página 50.

LINDSTRÖM, J. Real time database systems. *Wiley Encyclopedia of Computer Science and Engineering*, Wiley Online Library, p. 1–13, 2007. Citado na página 28.

MEEHAN, J. et al. Integrating real-time and batch processing in a polystore. In: IEEE. *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. [S.l.], 2016. p. 1–7. Citado na página 43.

MELO, R. C. d. S. et al. Sistema de monitoramento de consumo de água utilizando o protocolo de comunicação mqtt. Universidade Federal de Uberlândia, 2018. Citado na página 50.

MICHELSON, B. M. Event-driven architecture overview. *Patricia Seybold Group*, v. 2, n. 12, p. 10–1571, 2006. Citado na página 30.

MOZILLA, O. *Moz SQL Parser*. [S.l.]: GitHub, 2020. <https://github.com/mozilla/moz-sql-parser>. Citado na página 61.

MQTT. 2019. Disponível em: <http://mqtt.org/>. Citado na página 50.

Niu, Y.; Liu, F.; Li, Z. Load balancing across microservices. In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. [S.l.: s.n.], 2018. p. 198–206. Citado na página 65.

PARK, H. et al. *Event batching, output sequencing, and log based state storage in continuous query processing*. [S.l.]: Google Patents, 2019. US Patent App. 16/279,855. Citado 2 vezes nas páginas 17 e 18.

REDHAT. *O que são os microsserviços?* RedHat, 2020. Disponível em: <https://www.redhat.com/pt-br/topics/microservices/what-are-microservices>. Citado 2 vezes nas páginas 46 e 47.

ROKHMAH, S. Binary log analysis on mysql to help investigation process against database privillege attacks. *International Journal of Computer and Information System (IJCIS)*, v. 1, n. 1, 2020. Citado na página 60.

ROMANOV, E. L.; TROSHINA, G. V. The iot-architecture on the principles of reactive programming. In: IEEE. *2017 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*. [S.l.], 2017. p. 317–322. Citado na página 30.

ROSA, A. M. *Análise de Fluxos em Tempo Real para Gestão de Dados de Tráfego*. Tese (Doutorado), 2017. Citado 3 vezes nas páginas 21, 34 e 35.

SELLAM, T.; KERSTEN, M. 80 new packages to mine database query logs. *arXiv preprint arXiv:1703.08732*, 2017. Citado na página 48.

SHAHRIVARI, S. Beyond batch processing: towards real-time and streaming big data. *Computers*, Multidisciplinary Digital Publishing Institute, v. 3, n. 4, p. 117–129, 2014. Citado na página 29.

SILBERSCHATZ, A. et al. *Database system concepts*. [S.l.]: McGraw-Hill New York, 2010. v. 6. Citado 4 vezes nas páginas 20, 23, 24 e 67.

SMITH, B. *JSON básico: conheça o formato de dados preferido da web*. [S.l.]: Novatec Editora, 2020. Citado na página 66.

SOUSA, J. V. de. Computação em nuvem no contexto das smart grids: Uma aplicação para auxílio à localização de faltas em sistemas de distribuição. 2018. Citado na página 46.

STONEBRAKER, M.; ÇETINTEMEL, U.; ZDONIK, S. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, ACM, v. 34, n. 4, p. 42–47, 2005. Citado na página 29.

TERRY, D. et al. *Continuous queries over append-only databases*. [S.l.]: ACM, 1992. v. 21. Citado 6 vezes nas páginas 20, 24, 25, 26, 59 e 63.

TOSHNIWAL, A. et al. Storm@ twitter. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. [S.l.: s.n.], 2014. p. 147–156. Citado na página 39.

Vogel, P. et al. A low-effort analytics platform for visualizing evolving flask-based python web services. In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. [S.l.: s.n.], 2017. p. 109–113. Citado na página 58.

WAN, Z.; HUDAK, P. Functional reactive programming from first principles. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2000. (PLDI '00), p. 242–252. ISBN 1581131992. Disponível em: <https://doi.org/10.1145/349299.349331>. Citado na página 30.

XIE, X.; Govardhan, S. S. A service mesh-based load balancing and task scheduling system for deep learning applications. In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. [S.l.: s.n.], 2020. p. 843–849. Citado na página 56.

XIN, R. S. *Go with the Flow: Graphs, Streaming and Relational Computations over Distributed Dataflow*. Tese (Doutorado) — UC Berkeley, 2018. Citado 2 vezes nas páginas 18 e 37.

Yu, R. et al. Load balancing for interdependent iot microservices. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. [S.l.: s.n.], 2019. p. 298–306. Citado na página 65.

YUAN, M. Getting to know mqtt. why mqtt is one of the best network protocols for the internet of things. 2017. *Dostopno na: https://developer. ibm. com/articles/iot-mqtt-why-good-for-iot/[7.11. 2018]*, 2017. Citado na página 50.

ZENG, J. et al. Photon: A robust cross-domain text-to-sql system. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. [S.l.: s.n.], 2020. p. 204–214. Citado na página 61.

Zhang, P. et al. Redis++: A high performance in-memory database based on segmented memory management and two-level hash index. In: *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*. [S.l.: s.n.], 2018. p. 840–847. Citado na página 54.

ZHAO, X. et al. A taxonomy and survey of stream processing systems. In: *Software Architecture for Big Data and the Cloud*. [S.l.]: Elsevier, 2017. p. 183–206. Citado 2 vezes nas páginas 22 e 35.

ZIMMERLE, C.; GAMA, K. A web-based approach using reactive programming for complex event processing in internet of things applications. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. [S.l.: s.n.], 2018. p. 2167–2174. Citado na página 33.

Apêndices

APÊNDICE A – Documentação do Projeto

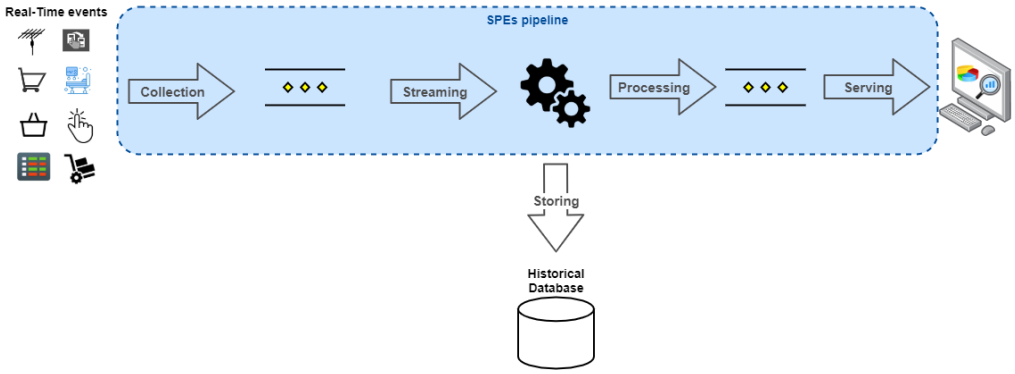
===

SQLStreamify is a simple and flexible middleware, with microservice-based architecture, capable of providing real-time data streams obtained by converting queries into continuous queries, without all the CEP tool complexity and with the possibility of execution without changes in legacy systems databases.

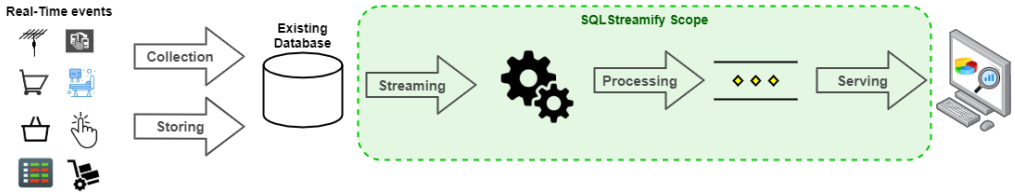
Is supported on Linux, Windows, and macOS. Running on Docker containers.

Pipeline

The usual pipeline for apps using data stream is presented in figure below, which shows storing data after collection and a basic processing. It is constantly named as Historical Database.



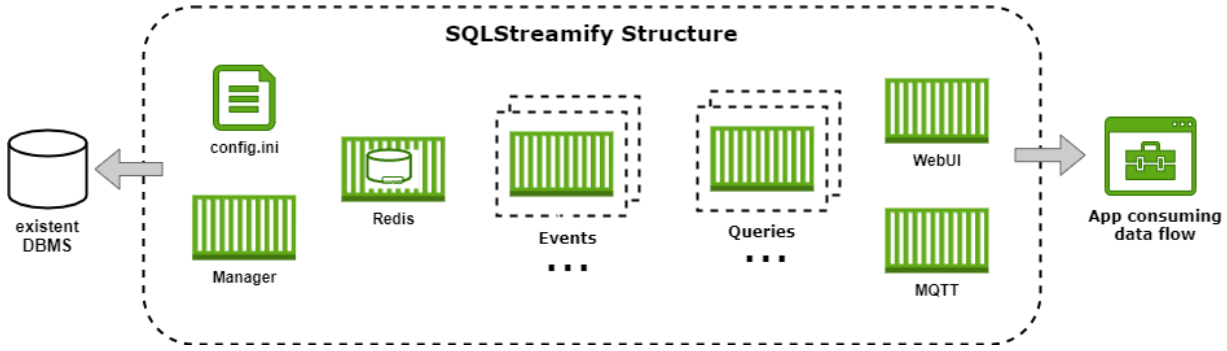
Nevertheless, there are cases where legacy applications record events collected in existing DBMSs. These data being updated in the database are the desired events for new applications that makes use of data streams from there. This is precisely the scope of the work. SQLStreamify transforms these events into flows by searching the data collected and stored in DBMSs.



SQLStreamify was proposed considering that developers constantly need real-time stream data reading functions in apps that use DBMSs. It delivers CQs based on data changes and notifies listeners when any change is made.

Structure

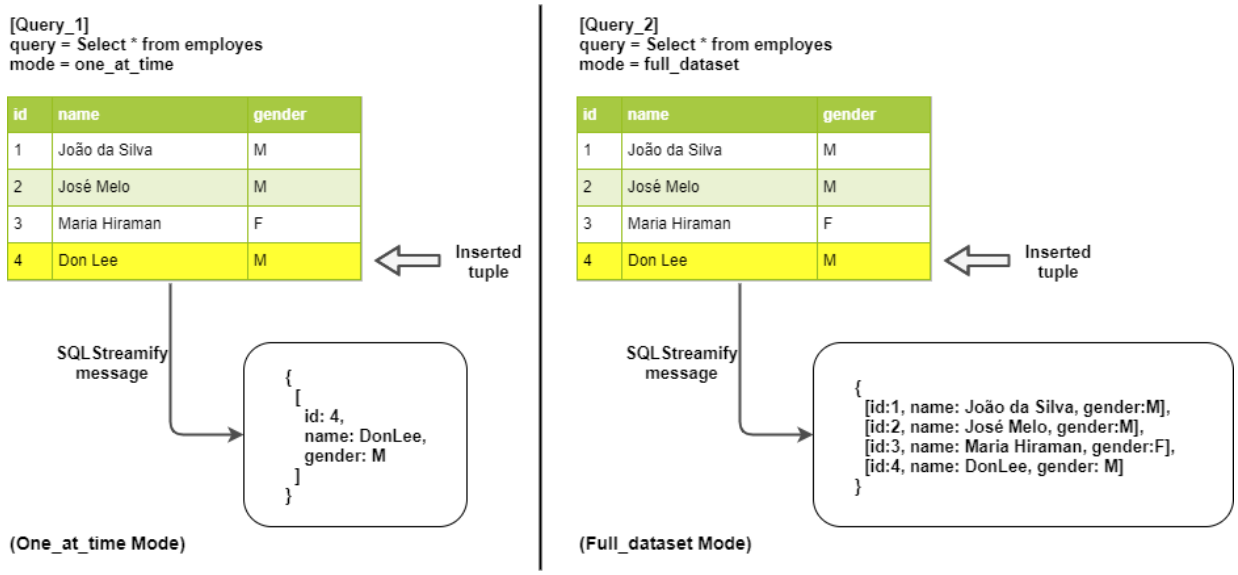
SQLStreamify structure consists of five microservices - Manager, Events, Queries, WebUI and MQTT-Broker. Besides these services, a Redis NoSQL database is used as storage. Its function is to provide a high performance and fast storage, once the main memory is used as storage, where all microservices can access to exchange data among them.



Delivery Modes

Modos de entrega.

- **One-at-time Mode**: One-at-time mode is the most logical alternative to send data stream. Whenever any change is observed, they are sent one by one.
- **Full-dataset Mode**: According to the semantics of the situation, the developer must choose how the data will be sent. Full-dataset received this name because of the need to send the entire result of the query registered by the user when the changes occur. Some cases that may need it are queries in which there is a data summary, such as a counter, groupings or data calculation. Another case that requires this mode is the ordered listings that limit the search, for example, the last ten sales for a given state. In this example listing, when any change is noticed, all query data are change and depending on the application scenario, the entire dataset is required.



Quick Start

Pre-requisites

- Install Python
- Install Docker
- Enable MySQL binary log

Installing

- Clone this repo
- Adjust **config.ini**
  - Database access parameters
  - Data stream exposing configs
  - Create queries, name, SQL and type

Running

- Execute start_service.py

Monitoring

- Dashboard will be avaiable on 8080 port.

Consuming

- Developers can use MQTT or WebSTOMP methods to get data streams from broker, using CQ giving name on config.ini.

Examples

- Some use case examples are available on repository.