FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Monitorização em Tempo Real de Indicadores de Performance do Retalho

Ricardo Jorge de Sousa Teixeira

DISSERTAÇÃO

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Henriqueta Sampaio da Nóvoa, PhD.

10 de Fevereiro de 2014

Monitorização em Tempo Real de Indicadores de Performance do Retalho

Ricardo Jorge de Sousa Teixeira

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: A. Augusto de Sousa

Arguente: Pedro Abreu

Vogal: Henriqueta Sampaio da Nóvoa

10 de Fevereiro de 2014

Resumo

No retalho, o acompanhamento da performance das vendas faz parte do dia a dia das equipas de gestão: a Sonae não é exceção, tendo implementado todo um sistema informático de forma a fornecer esta informação, crítica para o negócio, aos diretores, gestores e administração. Esse sistema permite visualizar, para o dia anterior, a performance das vendas de uma superfície comercial. Existe assim um desfasamento entre o momento em que as vendas ocorrem e o momento em que a informação fica disponível para consulta. Este desfasamento condiciona o tempo de reação das equipas que necessitam desta informação. Com o objetivo de eliminar este desfasamento e melhorar o tempo de reação das equipas de gestão, a Sonae pretende implementar um sistema capaz de processar o fluxo de dados proveniente das lojas de forma a conseguir apresentar ao utilizador, através de uma aplicação web, os indicadores de performance das lojas em tempo real. A arquitetura e desenvolvimento desse sistema é o objetivo desta dissertação.

A solução proposta consiste num sistema composto por vários componentes que capturam o fluxo de transações das lojas, processam esse fluxo gerando os indicadores de performance das vendas e, de seguida, armazenam esses resultados. A captura e processamento dos dados acontece em tempo real, de forma a que possam ser de imediato apresentados às equipas de gestão através de *dashboards web*. Os componentes que disponibilizam estes *dashboards* também fazem parte do sistema.

Esta solução utiliza tecnologias do domínio do "Big Data" como bases de dados NoSQL (Apache Cassandra e Redis) e sistemas de processamentos de fluxos de eventos (Apache Storm) para conseguir processar e armazenar o fluxo de dados de elevado volume gerado pelas lojas. A solução desenvolvida é distribuída, escalável e resistente a falhas, sejam estas de hardware ou de software.

Agradecimentos

A todos os que acreditaram que seria capaz de terminar esta longa viagem.

Ricardo Teixeira

"Always make new mistakes."

Esther Dyson

Conteúdo

Lista de Figuras

LISTA DE FIGURAS

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contexto/Enquadramento

Esta dissertação descreve o trabalho efetuado na ISI – direção de sistemas de informação e inovação da Sonae. Este departamento faz parte do grupo Sonae, um dos maiores retalhistas portugueses e é responsável pelos sistemas de informação ligados à venda a retalho - hipermercados Modelo e Continente, lojas Worten, lojas SportZone, entre outros.

Uma das funções da ISI é disponibilizar sistemas informáticos - ERP, bases de dados, e outros sistemas, que permitam aos gestores de lojas visualizar e analisar as vendas de produtos.

É a ISI que está encarregue dos sistemas que fornecem aos diretores e gestores as informações que lhes permitem analisar a performance das vendas das superfícies comerciais, informação de gestão essencial ao negócio. Esses sistemas têm como base um armazém de dados que é carregado durante a madrugada com os dados do dia anterior - os dados das vendas de um dia só ficam disponíveis para análise na manhã do dia seguinte. Devido a este desfasamento na disponibilização dos dados, existe espaço para a evolução destes sistemas, no sentido de os tornar mais ágeis e versáteis, de forma a possibilitar a monitorização de indicadores da performance das vendas em tempo real. É nesta área que a ISI pretende começar a atuar, visto que a monitorização de indicadores em tempo real é uma mais valia identificada ao nível da venda a retalho [MT06].

1.2 Motivação e Objetivos

O principal motivador desta dissertação é a resolução de uma necessidade identificada na Sonae - a da visualização da performance das vendas das lojas em tempo real.

Atualmente, a performance das vendas de uma loja é visualizada nos relatórios disponibilizados no dia seguinte. Por vezes, esse desfasamento entre o momento em que ocorrem as vendas e a sua análise prejudica a tomada de decisões e a reação a situações que acontecem durante o normal período de funcionamento das superfícies comerciais.

Se pelo menos uma parte dos dados que agora demoram um dia a chegar aos sistemas consultados pelos gestores e analistas, ficasse disponível em tempo real, o tempo de resposta a ocorrências identificáveis através desses dados, diminuiria drasticamente. Como a reação estaria mais próxima do acontecimento, seria mais fácil corrigir de imediato situações que, com o sistema atual, só seriam identificadas e corrigidas no dia seguinte.

Como exemplo, podemos descrever uma ocorrência comum: um artigo que está em campanha promocional esgota, ficando indisponível para venda (poderá haver um erro de reaprovisionamento). No sistema atual, esta situação é identificada de forma implícita pela direção de loja. Caso esta indisponibilidade não seja detetada, porque os clientes não perguntaram pelo artigo ou os colaboradores encarregados do reaprovisionamento não detetaram a falta deste, as perdas de vendas podem ser significativas e só serão detetadas no dia seguinte através da consulta dos relatórios das vendas. Existindo um sistema que permita acompanhar o volume de vendas de uma promoção em tempo real, uma quebra de vendas de uma campanha promocional seria imediatamente detetada através de um volume de vendas da campanha diferente do previsto. Ao verificar essa quebra, a direção de loja poderia agir imediatamente e corrigir a situação.

O exemplo dado, que acontece no dia a dia da venda a retalho, permite ver que, em algumas situações, disponibilizar em tempo real os indicadores de vendas se traduz imediatamente em benefícios financeiros.

Assim, o presente projeto pretende eliminar o intervalo acima descrito para alguns indicadores estratégicos que beneficiem de uma análise imediata ao longo do período de vendas. A visualização em tempo real desses indicadores estratégicos relacionados com a performance das vendas vai assim facilitar o dia a dia de equipas como as de direção de loja e direção comercial da Sonae, encarregadas de gerir lojas, produtos e campanhas promocionais. Estes indicadores de performance em tempo real também são úteis à administração (*upper management*) porque ter acesso a dados instantâneos é um fator de diferenciação em alguns processos de decisão de alto nível.

É de ressalvar que as necessidades de agregação dos dados consultados pelas várias direções são diferentes, sendo que as direções comerciais necessitam de uma agregação ao produto e à campanha promocional e a direção de loja tem interesse nos resultados ao nível das vendas da loja. Já a administração, para além dos resultados ao nível das lojas, está também interessada nos resultados ao nível das insígnias (marcas de cadeias de lojas como o Continente ou a Worten) e das zonas (conjuntos de lojas da mesma insígnia numa determinada área geográfica).

O sistema a implementar deve monitorizar e processar, de forma contínua, as transações das caixas registadoras das superfícies comerciais, agregando os dados de forma a fornecer indicadores de performance das vendas dessas superfícies. O sistema será visível para o utilizador final na forma de *dashboards* consultáveis numa aplicação web [Mal05]. Tendo em conta que os utilizadores alvo deste sistema são os gestores e a administração, o sistema deve ser acessível e a informação deve ser transmitida de forma clara [ACNM06]. Para facilitar a consulta de informação, os vários indicadores de performance das vendas presentes nestes *dashboards* serão apresentados na forma de gráficos e tabelas [WWH+06], devendo estar disponíveis dados agregados de forma adequada aos vários grupos de gestão da Sonae mencionamos acima.

O principal problema tecnológico a resolver na implementação do sistema proposto consiste no tratamento do volume de dados em questão - num dia, podem ser gerados pelas caixas registadoras perto de nove milhões e meio de transações, que poderão ou não ter que ser cruzados com cerca de dois milhões de produtos e milhares de promoções em vigor. O processamento e agregação de volumes de dados desta magnitude pode acarretar problemas que potenciem a utilização de soluções específicas para o tratamento de grandes volumes de dados. Entra-se assim no domínio dos problemas vulgarmente conhecidos como "Big Data" Os sistemas analisados no capítulo 2, referente ao estado da arte, pertencem a esse domínio.

1.3 Estrutura do documento

O presente documento tem mais cinco capítulos para além desta introdução.

No Capítulo 2, faz-se uma revisão do estado da arte das tecnologias necessárias para resolver o problema proposto, analisando-se as *data stores* escaláveis e também os sistemas de processamento de fluxos e de eventos. No Capítulo 3, detalha-se qual o estado atual da visualização de indicadores de performance da Sonae, apresentando-se as estruturas da organização e os sistemas atualmente utilizados. Os requisitos, funcionais e não funcionais, que foram definidos para a solução a implementar são descritos no Capítulo 4. No Capítulo 5 descreve-se a arquitetura e detalha-se a implementação da solução proposta. Por fim, no Capítulo 6 são apresentadas as conclusões desta dissertação e as possibilidades de evolução da solução proposta.

Capítulo 2

Revisão Bibliográfica

2.1 Introdução

Neste capítulo, será analisado o estado da arte das principais tecnologias de relevo para o objetivo deste projeto. Podem-se dividir essas tecnologias em dois grandes grupos: tecnologias de armazenamento de dados e tecnologias de processamento de eventos. No que se refere ao armazenamento de dados, estudam-se na Secção 2.2 *data stores* capazes de escalar horizontalmente e lidar com vastas quantidades de dados simples. Na Secção 2.3 analisam-se sistemas de processamento de eventos complexos e de fluxos de eventos capazes de processar rapidamente uma elevada quantidade de dados.

2.2 Análise de data stores escaláveis

Atualmente, os Sistemas de Gestão de Bases de Dados Relacionais (*Relational Database Management Systems* - RDBMS) são a tecnologia predominantemente escolhida para guardar dados estruturados [Cat11]. Estes sistemas utilizam a linguagem SQL (*Strucuted Query Language*) como forma de operar sobre os dados aí guardados. Desde a sua introdução, nos anos 70, que as alternativas que surgiram aos sistemas RDBMS foram incorporadas nos próprios RDBMS ou relegadas para produtos de nicho, nunca ganhando quota de mercado em relação aos RDBMS [SSK11]. Recentemente, a utilização dos RDBMS como solução para todos os tipos de necessidades de armazenamento de dados tem vindo a ser questionada [Gho10], o que levou ao surgimento de uma grande variedade de bases de dados alternativas. Estas bases de dados alternativas e o próprio movimento têm sido chamados de NoSQL. Este termo é utilizado para descrever o aumento da utilização de bases de dados não relacionais por programadores web [FTD+12].

O conceito de *data store*, sem tradução óbvia para português, é utilizado neste documento para descrever um sistema que consegue persistir dados. É um conceito mais abrangente que o conceito

de base de dados visto que não só abrange bases de dados como também outro tipo de estruturas que podem armazenar dados, tais como ficheiros simples.

Nesta secção, analisam-se as propriedades de várias *data stores*, quer relacionais quer não relacionais, quando utilizadas para operações simples - pesquisas através de chaves e leituras ou escritas de um ou poucos registos [Cat11].

Uma das características analisada, essencial para o objetivo desta dissertação, é a escalabilidade horizontal. Esta consiste no aumento de capacidade dos sistemas através do aumento do número de servidores pelos quais os dados e carga são distribuídos. Existe também a escalabilidade vertical: aumento da capacidade do sistema alterando a configuração do hardware - adicionando mais núcleos e/ou processadores que partilham RAM e discos. Devido ao elevado volume de dados que o sistema proposto por esta dissertação terá que processar e armazenar, a escalabilidade horizontal é mais importante que a vertical, porque esta permite escalar além dos limites (verticais) permitidos pela quantidade de RAM, disco e processadores que um único sistema pode ter. Quando se trata de elevados volumes de dados é muito fácil ultrapassar os limites de processamento de uma única máquina, por muito poderosa que esta seja. Esta forma de escalar pode também levar a uma economia de custos, através da utilização de hardware de consumo, normalmente mais económico que as soluções de gama empresarial [AADE11].

Agrupam-se os projetos analisados em três categorias de data stores:

- *Key-Value Stores*
- *Document Stores*
- *Extensible Record Stores*

As *key-value stores* guardam os dados em pares chave valor: pares que associam a uma chave (*key*) um valor (*value*). Uma chave é um identificador único que determina um valor. Um valor pode ser qualquer tipo de dados tal como cadeias de caracteres (*strings*), contadores, imagens ou vídeos. Como exemplo de um par chave valor temos **17 = "Adelaide"**. Neste caso, o número 17 consiste na chave única que identifica uma pessoa e o nome "Adelaide" é o valor - nome dessa pessoa. As *key-value stores* podem guardar uma imensa quantidade destes pares, por vezes ultrapassando os *terabytes* [See09].

As *document stores* guardam dados mais complexos do que os guardados pelas *key-value stores*. O termo documento, neste caso, refere-se a qualquer tipo de objeto complexo. Este tipo de sistemas tem, normalmente, capacidades que permitem gerir mais facilmente documentos: índices secundários; suporte para múltiplos tipos de documentos; documentos dentro de outros documentos (documentos aninhados); listas de documentos.

As *extensible record stores* permitem guardar dados organizados através de linhas e colunas. A técnica de escalabilidade comum para este tipo de *data stores* é a de distribuir quer as linhas, quer as colunas, através de múltiplos nós. As linhas são distribuídas através de *sharding* (partições) da chave primária. As colunas são distribuídas através de grupos de colunas. Estes dois tipos de partições (horizontais e verticais) podem ser implementados simultaneamente. As linhas são

análogas a documentos e podem ter um número arbitrário de atributos (colunas). Ao contrário das *document stores*, as *extensible record stores* não costumam suportar objetos aninhados (*nested objects*).

Apesar das *extensible record stores* serem inspiradas no projeto BigTable da Google, não chegam ao nível de escalabilidade deste [Cat11].

De seguida, são analisados vários projetos em cada uma destas categorias. Tendo em conta a quantidade de projetos de *data stores* existentes, foi necessário efetuar uma filtragem prévia em que se tentou selecionar apenas projetos com relevância para o objetivo desta dissertação. Por exemplo, foram descartados projetos que não apresentavam funcionalidades de resistência à perda de dados no caso de uma falha de hardware. Para além disso, a análise privilegia *data stores* de alta performance que permitam escalar horizontalmente e que recuperem automaticamente de falhas de nós.

2.2.1 Project Voldemort

Project Voldemort[1] é uma *data store open source* escrita em Java. Proporciona controlo de concorrência multiversão para *updates*. Como esta *data store* atualiza as réplicas de forma assíncrona não garante a consistência de dados. No entanto, pode garantir vistas atualizadas se a maioria das réplicas for lida.

Esta *data store* suporta *optimistic locking* - uma forma de *locking* que utiliza o *roll back* como forma de gestão de transações. Este método permite maior rapidez quando os conflitos são raros [KR81]. Utiliza também um sistema de relógios vetoriais para ordenar as versões, semelhante à *Amazon Dynamo* [DHJ+07] [SK92].

Tal como muitas outras *data stores NoSQL*, este projeto suporta o *sharding* dos dados, que são distribuídos por um anel de nós.

O sistema adapta-se automaticamente à adição e remoção de nós e também deteta e recupera nós com falhas. Os dados podem ser guardados apenas em RAM ou então, caso seja desejado, a persistência pode ficar a cabo de um motor de armazenamento (*storage engine*). Para além de valores escalares, são suportadas listas e registos (*records*)[Cat11].

2.2.2 Riak

Riak[2] é uma *data store* escrita pela Basho. É descrita pelos seus criadores como uma *key-value store* e também uma *document store*. Pode ser caracterizada como uma *key-value store* avançada, visto que não disponibiliza funcionalidades importantes que, normalmente, estão presentes nas *document stores*[Cat11].

Os objetos podem ser guardados e obtidos no formato JSON e como tal podem ter múltiplos campos. Os objetos podem ser agrupados em baldes (*buckets*) e os campos permitidos/necessários podem ser definidos para cada um. Apenas as chaves primárias são indexáveis. A Riak não tem

[1] *Site* do Project Voldemort: `http://www.project-voldemort.com/voldemort/`
[2] *Site da Riak*: `http://basho.com/riak/`

os mecanismos de interrogação das *document stores* - a pesquisa apenas pode ser feita através das chaves primárias [Muh11].

A Riak suporta replicação de objetos e *sharding* através de *hashing* na chave primária. As réplicas podem ser temporariamente inconsistentes. É possível definir, por cada leitura/escrita, quantas réplicas em nós diferentes irão responder. Através da utilização de *vector clocks*, é facilitada a reparação automática dos dados quando estes estão fora de sincronia. O acesso aos dados é feito através de pedidos REST em HTTP. Também existem interfaces em Erlang, Java e outras linguagens. Tal como o Project Voldemort, a persistência dos dados é conseguida através de motores de armazenamento que podem ser trocados. Os dados podem ser persistidos através de tabelas ETS, tabelas DETS ou tabelas Osmos. Todos estes *plugins* de persistência são escritos em Erlang. Uma funcionalidade que merece ser mencionada é o facto de se poder guardar links entre documentos.

2.2.3 Redis

A Redis[3] é uma *data store in memory* semelhante à Memcached desenvolvida por Salvatore Sanfilippo e Pieter Noordhuis [SN]. Os dados são guardados em memória e, apesar de ser possível guardar uma cópia de segurança em disco [DB12], é possível haver perda de dados. Este facto torna esta *data store* desadequada para a utilização em situações que tal não possa acontecer. No que diz respeito à resistência a falhas, desde a versão 2.4.16 que é possível utilizar o Redis Sentinel para gerir automaticamente os nós.

2.2.4 Tokyo/Kyoto Cabinet Cabinet

Tokyo Cabinet[4] é uma implementação do motor de base de dados *DBM* originalmente escrito por Ken Thompson. Os dados podem ser guardados em memória ou no disco e consistem numa tabela de *key-value pairs* [See09]. Esta *data store* não dispõe de funcionalidades que facilitem a sua distribuição por vários nós, nem a recuperação automática de nós falhados, o que a torna menos adequada para situações de alta disponibilidade. A empresa que desenvolve a Tokyo Cabinet recomenda a utilização da mais recente Kyoto Cabinet[5]. No entanto, esta solução apresenta as mesmas lacunas no que respeita à recuperação de falhas e distribuição.

2.2.5 Couchbase Server

A Couchbase Server[6] é uma base de dados NoSQL desenvolvida pela Couchbase. É distribuída segundo a licença Apache Public Licence.

[3] *Site* da Redis: `http://redis.io/`
[4] *Site* da Tokyo Cabinet: `http://fallabs.com/tokyocabinet/`
[5] *Site* da Kyoto Cabinet: `http://fallabs.com/kyotocabinet/`
[6] *Site* da Couchbase Server: `http://www.couchbase.com/couchbase-server`

Esta solução é baseada na conhecida Memcached, uma base de dados *in memory*, *open source*, que é utilizada como *cache* em muitos websites atuais. A Couchbase adiciona à Memcached funcionalidades como persistência, replicação, alta disponibilidade e crescimento dinâmico - funcionalidade importante para o objetivo desta dissertação. A Couchbase Server suporta *sharding* e distribui os dados de forma uniforme entre os nós. A persistência é assíncrona, sendo apenas utilizada para salvaguardar os dados e não como forma de melhorar o desempenho.

Desde a versão 2.0 que deve ser considerada como um híbrido entre *data store* e *document store*, visto que foi adicionada a funcionalidade que permite guardar documentos em JSON.

2.2.6 CouchDB

Este projeto da Apache Foundation é um dos projetos NoSQL mais antigos. A CouchDB[7] é escrita em Erlang e permite armazenar documentos que consistem em campos escalares (cadeias de carateres, dados numéricos ou variáveis booleanas) ou compostos (documentos ou listas). Podem-se agrupar os documentos em coleções (*collections*) e criar índices secundários, desde que se explicite os campos das coleções que se quer indexar.

As interrogações são feitas através de *views* definidas em Javascript. Os índices são *b-trees*, permitindo ordenar os resultados das interrogações ou definir intervalos para os mesmos. As interrogações podem ser distribuídas em paralelo por múltiplos nós. O mecanismo de *views* desta *data store* faz com que seja mais trabalhoso para o programador fazer interrogações do que utilizar uma linguagem semelhante ao SQL. A replicação é síncrona, não existindo lugar a *sharding* [PPS11]. Este facto leva a que seja difícil escalar esta *data store* horizontalmente sem recorrer a outras soluções. Para isso, existe atualmente o projeto BigCouch que permite distribuir uma base de dados CouchDB através de múltiplos servidores.

A consistência não é garantia, mas cada cliente vê uma réplica consistente da base de dados. Os clientes são notificados se o documento foi atualizado desde que foi obtido, cabendo ao cliente decidir o que fazer depois. A durabilidade dos dados é garantida, visto que todos os *inserts* ou *updates* são persistidos no disco imediatamente.

A interface com esta *data store* é feita através de REST. Existem, no entanto, bibliotecas para várias linguagens nativas (Java, C, Python, PHP, entre outras) que convertem chamadas nativas para chamadas REST.

2.2.7 MongoDB

A MongoDB[8] é uma *document store open source* escrita em C++. Utiliza índices, *locks* apenas para escrita e tem um mecanismo de interrogações a documentos. Por suportar particionamento de dados, é uma boa escolha para aplicações em que a escalabilidade horizontal seja importante.

A MongoDB suporta *sharding* automático, distribuindo os documentos através de vários servidores e utilizando um dos nós como *master*. É possível configurar a base de dados para que

[7] *Site* da CouchDB: `http://couchdb.apache.org/`
[8] *Site* da MongoDB: `http://www.mongodb.org/`

seja possível ler de qualquer um dos nós, no entanto, apenas é possível escrever na base de dados através do *master*. A replicação de dados é suportada, sendo utilizada apenas para resistência a falhas e não como um método de escalabilidade.

Não é garantida consistência global quando se lê de qualquer nó, no entanto, é possível ter consistência local obtendo uma cópia local atualizada de um documento.

As interrogações são dinâmicas e, tal como as bases de dados relacionais, utilizam automaticamente os índices presentes. A concorrência é gerida através de operações atómicas ao nível dos campos de um documento. O comando de *update* suporta modificadores que facilitam alterações a valores individuais: **$set** define um valor; **$inc** incrementa um valor; **$push** adiciona um valor a um *array*; **$pushAll** adiciona vários valores a um *array*; **$pull** remove um valor de um *array*; **$pullAll** remove vários valores de um *array*. Estas alterações são efetuadas, na maioria dos casos, *in place*: não é necessário obter primeiro os dados e só depois alterar o valor no servidor.

Os dados são guardados num formato binário, semelhante a JSON, chamado BSON. A codificação dos dados é feita no cliente, antes dos dados serem enviados para a *data store*.

2.2.8 Cassandra

O Apache Cassandra é um sistema de gestão de bases de dados distribuído, não relacional (NoSQL), desenvolvido inicialmente pelo Facebook. Em 2010, passou a ser um projeto de topo da Apache Software Foundation. É *open source*, utilizando a Apache Software License 2.0. Esta *data store* foi desenhada para lidar com grandes quantidades de dados distribuídos através de servidores de consumo. Um dos objetivos é providenciar alta disponibilidade sem haver um ponto de falha único [Sut10].

O Apache Cassandra permite guardar pares chave valor com consistência configurável [TB11]. Os dados são agrupados em famílias de colunas, que são definidas quando uma base de dados é criada. Estas são equiparáveis a tabelas de uma base de dados relacional. Os valores dos pares chave valor são compostos por várias colunas, as quais podem ser definidas linha a linha, apesar de ser possível também definir colunas base por cada família de colunas. Na figura 2.1 podemos ver um esquema de uma destas famílias. As colunas base são "name"e "email", que são definidas através da *column family*. Para além destas, algumas das linhas têm também outras colunas como "address"e "state". Esta propriedade permite que, para uma chave, se utilizem as colunas como se tratassem de um array de objetos.

As principais funcionalidades do Cassandra são:

Descentralização Todos os nós num *cluster* Cassandra têm o mesmo papel e, como tal, não existe um ponto único de falha. Os dados são distribuídos através do *cluster* de forma a que cada nó contenha dados diferentes. Apesar dos dados serem distribuídos, o acesso ao *cluster* pode ser feito através de qualquer um dos nós deste, que depois se encarrega de obter os dados dos outros nós.

Funcionalidades de replicação avançada Como o Cassandra é desenhado com um sistema distribuído, uma das suas funcionalidades principais é o facto de se poder distribuir o sistema

row key	columns...			
jbellis	name	email	address	state
	jonathan	jb@ds.com	123 main	TX
dhutch	name	email	address	state
	daria	dh@ds.com	42 2nd St.	CA
egilmore	name	email		
	eric	eg@ds.com		

Figura 2.1: Cassandra - Exemplo de esquema de colunas.

através de um grande número de nós de vários *data centers*. Existem funcionalidades específicas para efetuar replicações entre *data centers*.

Elasticidade A taxa de transferência aumenta linearmente à medida que são adicionados nós, os quais podem ser adicionados com o sistema *on-line* [KAB+11].

Tolerância a falhas Os dados são replicados automaticamente através de vários nós. Os nós podem ser locais ou estar distribuídos por múltiplos *data centers*. Os nós falhados podem ser recuperados com o sistema *on-line*.

Consistência ajustável É possível ajustar o nível de consistência das leituras e escritas. A consistência pode ser definida por cada interrogação. Ao nível da escrita, pode-se definir que uma operação nunca falhe, que seja difundida para pelo menos um número mínimo de nós ou, então, que tenha que ser replicada para todos os nós. Ao nível da leitura, existem definições semelhantes: pode-se obter uma resposta do nó mais próximo, da alteração mais recente de um número mínimo de nós ou então da alteração mais recente de todos os nós.

MapReduce Existe integração com MapReduce e também com outras tecnologias complementares como Apache Pig e Apache Hive.

Linguagem de interrogação proprietária O Cassandra utiliza CQL (Cassandra Query Language), uma linguagem semelhante ao SQL que é uma alternativa à utilização de RPC (Remote Procedure Calls). Existem adaptadores para CQL em várias linguagens, incluindo Java (JDBC).

2.3 Processamento de eventos

As tecnologias de processamento de dados analisadas enquadram-se na categoria de processamento de eventos complexos (CEP) e também na categoria de processamento de fluxos de eventos (ESP). Existem várias semelhanças entre CEP e ESP. No entanto, o processamento de fluxos de eventos foca-se em analisar rapidamente vários eventos que chegam através de um ou mais fluxos ordenados. Quanto ao processamento de eventos complexos, este lida com regras de reação a eventos, assim como a relação causal entre esses eventos, permitindo a extração de informação de

nuvens de eventos - conjuntos de eventos não ordenados. Ressalva-se que a distinção entre CEP e ESP é alvo de alguma controvérsia[9].

O processamento de eventos complexos (tradução do inglês *complex event processing* - CEP) consiste em fazer a correspondência entre eventos ou séries de eventos e padrões que descrevam as características dos eventos. Estes padrões funcionam como filtros, assinalando os eventos que se enquadram nas definições presentes no padrão [KM12]. Este pode ser, por exemplo, a deteção do acontecimento de um evento B após um outro evento A.

O processamento de eventos complexos está mais orientado para a deteção de padrões de eventos. No entanto, os motores de processamento de eventos complexos não costumam ter em atenção as questões temporais associadas a um fluxo de eventos. Por outro lado, o processamento de fluxos de eventos (*event stream processing*) consiste em escutar uma quantidade infinita de eventos e reagir a situações que ocorrem nesse fluxo. As considerações a fazer ocorrem normalmente sobre uma janela deslizante (*sliding window*), que pode ser definida temporalmente ou em número de eventos [GZK05]. Um exemplo de um caso de uso pode ser o de detetar se a temperatura de um sensor aumentou mais de 20% nos últimos cinco minutos. O processamento de eventos complexos também considera a correlação entre fluxos ou entre fluxos e dados históricos [MBM09].

De seguida, são analisados o Storm e o S4, dois sistemas de processamento de fluxos de eventos e também o Esper, um motor de processamento de eventos que suporta eventos complexos e fluxos de eventos.

2.3.1 Storm

O Apache Storm[10] [11] é uma plataforma para processamento de fluxos de eventos em tempo real. O Storm pode ser utilizado para reunir e executar elementos de processamento de fluxos de eventos.

As propriedades principais do Storm são:

Conjunto alargado de casos de uso O Storm pode ser utilizado para processar mensagens e atualizar bases de dados, executar uma interrogação contínua em fluxos de dados e direcionar os resultados para clientes, paralelizar uma interrogação complexa como uma pesquisa em tempo real, entre outros. O Storm consegue esta adaptabilidade com um pequeno conjunto de primitivas.

Escalabilidade Um *cluster* Storm pode escalar até conseguir processar uma imensa quantidade de mensagens por segundo [Hen12]. Devido à utilização de ZooKeper para coordenação do *cluster*, o Storm consegue escalar até tamanhos maiores [HKJR10].

Sem perdas de dados O Storm garante que todos os dados são processados, não sendo assim necessário implementar métodos externos para garantir esse processamento. Isto faz com

[9] Diferença entre CEP e ESP: http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/

[10] Storm no GitHub: https://github.com/nathanmarz/storm/

[11] *Site* do projecto Storm: http://storm-project.net

que seja mais versátil do que sistemas como o S4 [NRNK10], que não dão essa garantia e, como tal, têm casos de uso mais limitados. Para garantir o processamento de todos os dados, o Storm vigia todos os tuplos de uma topologia através de tarefas especiais encarregues da confirmação e do reenvio de tuplos que não foram entregues ao destino [Nag12].

Facilidade de gestão É um objetivo explícito do projeto Storm que os clusters Storm sejam fáceis de criar e gerir. Para facilitar essa experiência existe o Storm UI, um interface gráfico que permite visualizar o estado do *cluster*. Na Figura 2.2 pode-se ver um exemplo desse interface.

Tolerância a falhas Todas as topologias correm para sempre. Se houver falhas de um ou mais nós durante a execução de uma topologia, as tarefas desses nós serão automaticamente atribuídas a outros nós.

Múltiplas linguagens de programação Apesar de existir uma interface nativa para JAVA, as topologias e os componentes de processamento podem ser definidos noutras linguagens. Isto é possível porque a definição das topologias é efetuada através da linguagem Thrift. Como tal basta definir a topologia em Thrift[12] a partir de qualquer linguagem [ASK07].

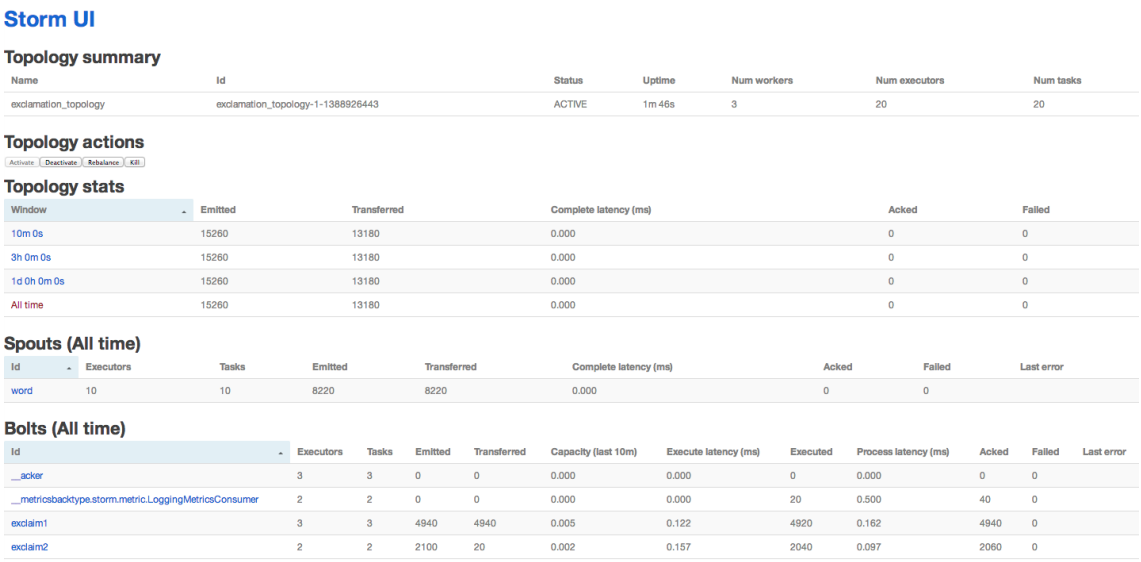

Figura 2.2: Interface gráfico do Twitter Storm.

Os componentes principais do Twitter Storm são Spouts, Bolts e topologias (Topologies).

Os Spouts geram os dados a serem processados. Estes dados podem ser externos ou então gerados pelos próprios Spouts.

O segundo tipo de elementos do Storm são os Bolts. Os Bolts são elementos de processamento de dados. Os dados a processar são originados pelos Spouts ou por outros Bolts. Isto acontece porque é possível, opcionalmente, ligar a saída de um Bolt a outro Bolt. O processamento que é

[12]Thrift: `http://thrift.apache.org/`

feito nos Bolts pode ser de vários tipos, desde filtragem ou agregação a guardar tuplos numa base de dados.

Um *cluster* Storm permite executar topologias. Uma topologia é um grafo composto por Spouts e Bolts ligados através de fluxos de dados internos. A cada um destes fluxos de dados internos chama-se Stream. Um Stream é composto por tuplos do mesmo tipo, conhecido de antemão.

Aquando da sua criação, os Spouts declaram qual o tipo de tuplo que produzem. Os Bolts que produzem dados que serão posteriormente processados (por outros outros Bolts) também declaram qual o tipo de tuplo que emitem.

Pode-se ver um exemplo de uma arquitetura Storm na Figura 2.3.

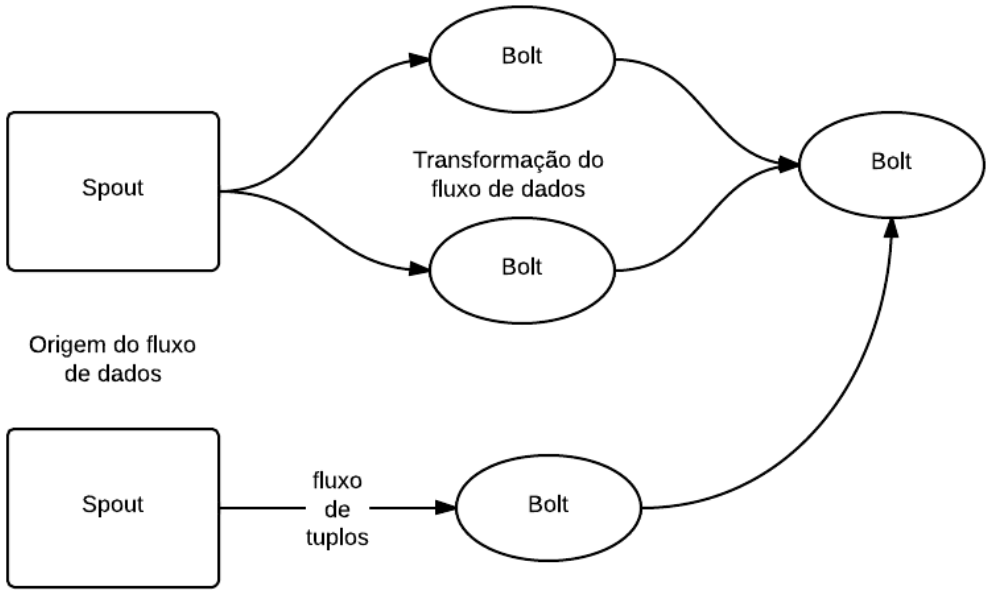

Figura 2.3: Exemplo de uma topologia de Twitter Storm [13].

Uma característica que está ausente no Storm é o conceito de janela temporal deslizante. Este conceito pode, no entanto, ser implementado no Storm diretamente em código recorrendo às primitivas disponibilizadas [14].

2.3.2 S4

O Apache S4[15] (*Simple Scalable Streaming Simnstem*) é uma plataforma de *streaming*, inicialmente desenvolvida pela Yahoo. O seu objetivo é permitir o processamento paralelo de fluxos de informação de forma simples e distribuída, escondendo a complexidade da computação paralela do programador.

[14] Discussão sobre janelas temporais em Storm: https://groups.google.com/forum/#!msg/storm-user/-r9HOIdYMGI/ZgTLSzlAjXIJ

[15] *Site* do Apache S4: http://incubator.apache.org/s4/

O processamento é conseguido recorrendo a pequenos programas: os elementos de processamento (*Processing Elements* - PE). Estes PE consomem eventos de um determinado fluxo e tanto podem publicar os resultados do processamento efetuado como podem gerar novos fluxos para serem consumidos por outros PE. A distribuição dos fluxos pelos eventos é feita de acordo com chaves - todos os eventos com a mesma chave de um determinado fluxo são consumidos pelos mesmo PE. Isto implica que os PE podem ser distribuídos de forma a processar informação em paralelo. Isto pode levar a um sistema altamente escalável, principalmente se houver um cuidado ao atribuir as chaves aos PE de forma a particionar a carga de uma forma equilibrada.

O S4 é completamente descentralizado, todos os nós são simétricos e não existe um único ponto de falha. Tal como no Twitter Storm, o ZooKeeper é utilizado para gerir o *cluster*. Quando um dos servidores do *cluster* falha, um servidor em *stand-by* é automaticamente ativado. Durante esta passagem, o estado dos PE envolvidos é perdido.

Para manter a simplicidade, não é possível modificar o tamanho de um *cluster* em *runtime*. O número de nós, incluindo os de *stand-by* é definido à partida. Pelo mesmo motivo também não é possível rebalancear a carga dos nós com o sistema *online*.

Está a ser desenvolvida uma integração com Apache Helix para permitir um número variável de nós e o rebalanceamento das partições [16].

O S4 implementa um conceito de degradação graciosa: o sistema de processamento deve ser capaz de acompanhar a taxa de eventos, caso isso não aconteça, são descartados eventos de forma propositada [NRNK10].

O S4 suporta o conceito de janelas temporais deslizantes, mas ressalva na sua documentação que, por uma questão de gestão de memória, não é aconselhável utilizar este conceito para se lidar com eventos esparsos [17].

2.3.3 Esper

O Esper é um motor de processamento de fluxos de eventos e também de eventos complexos. É produzido pela EsperTech[18] e disponibilizado numa licença *open source*. Existe uma versão com suporte comercial chamada Esper Enterprise Edition e também uma versão comercial especializada em fornecer alta disponibilidade - EsperHA. Esta análise será focada na versão *open source* (Gnu Public Licence v2) do Esper.

O Esper é desenvolvido em Java. Existe, no entanto, uma versão em C# denominada NEsper.

O Esper permite a deteção de situações complexas em tempo real de forma a despoletar ações quando um determinado conjunto de condições ocorre em um ou mais fluxos de eventos. O Esper é utilizado em campos tão diversos como a negociação algorítmica, deteção de fraudes, *business intelligence* em tempo real e aplicações CRM.

[16] Discussão sobre integração com Helix: http://mail-archives.apache.org/mod_mbox/incubator-s4-user/201211.mbox/%3C0AA96966-4013-4687-B11D-8B9F0F4B9CD9@apache.org%3E

[17] Janelas temporais em S4: http://people.apache.org/~mmorel/apache-s4-javadoc/0.5.0/org/apache/s4/core/window/AbstractSlidingWindowPE.html

[18] *Site* do Esper: http://www.espertech.com/products/esper.php

O Esper funciona, como muitos outros motores de processamento de eventos complexos, através de uma arquitetura Publish/Subscribe. As subscrições são efetuadas através de *listeners* que são anexados ao motor de Esper, no qual cada *listener* contém uma interrogação que define a subscrição. Estes subscritores contêm métodos que são chamados quando a interrogação corresponde a um evento de entrada. Todos os outros eventos que não correspondam a uma interrogação são descartados. Na Figura 2.4 pode-se ver um exemplo de uma interrogação em Esper. A subscrição "Alert User" é despoletada quando a chamada é de um cliente "Vito" ou de um cliente "Tom".

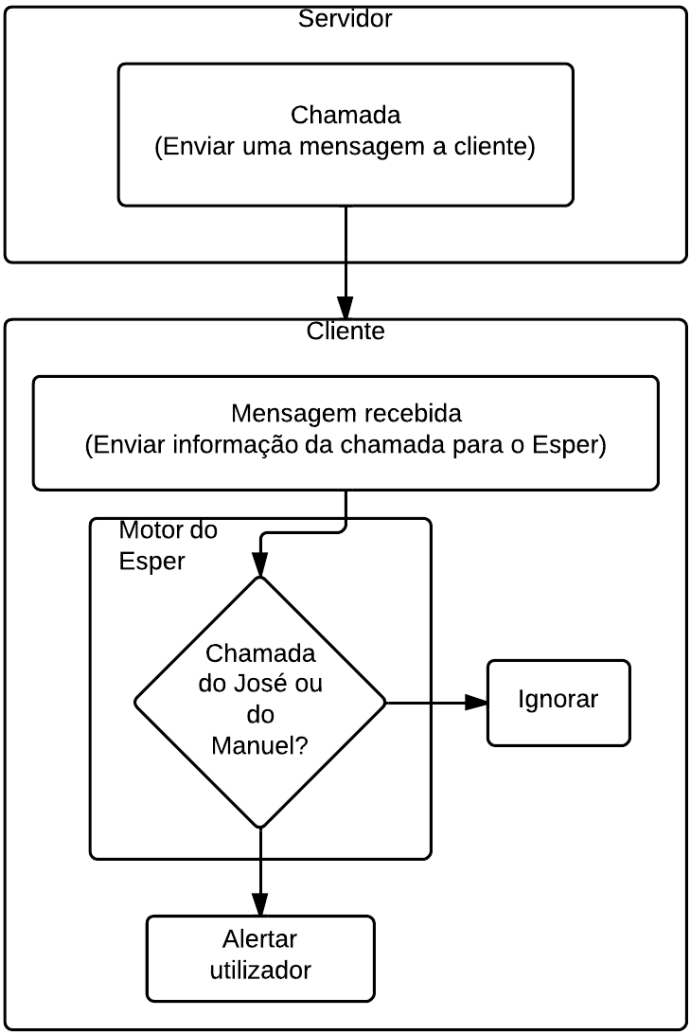

Figura 2.4: Esper - Exemplo de funcionamento[19].

Para fazer a correspondência entre eventos, o Esper providencia uma linguagem de processamento de eventos (EPL) para filtrar e agregar e fazer *joins* em janelas temporais de múltiplos

[19]Figura adaptada de http://www.debugrelease.com/2012/06/14/complex-event-processing-with-esper-tutorial-fo

fluxos de eventos. Também é possível exprimir causalidade temporal complexa entre eventos. Na Figura 2.5 mostra-se um esquema de funcionamento do Esper.

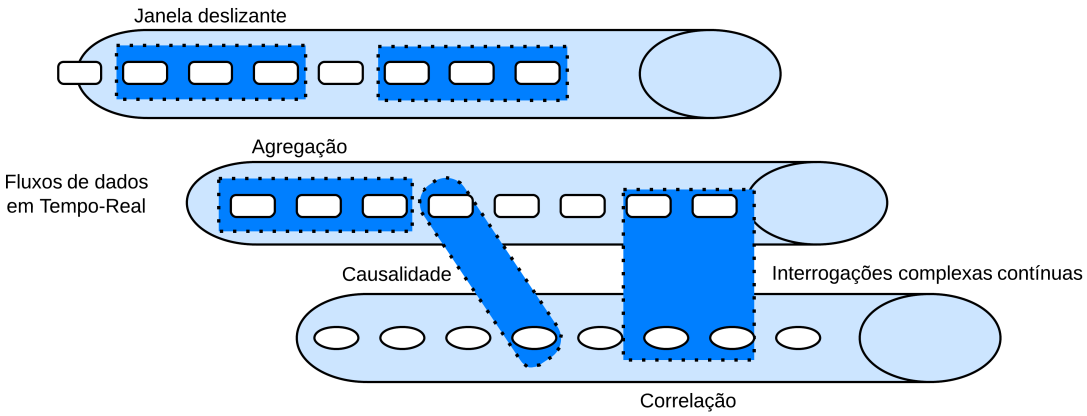

Figura 2.5: Esper - Processamento de fluxos de eventos [20].

A EPL é um mecanismo poderoso para trabalhar com janelas temporais deslizantes [HBG10]. Esta permite que estas janelas temporais sejam integradas em interrogações e relacionadas com outro tipo de constrangimentos. Por exemplo, se se quiser saber quando é que, nas últimas seis horas, a média do tempo de resposta de um processo ultrapassou os 300ms, pode-se ter uma interrogação como a seguinte (simplificada):

```
1 select * from ProcessEvent win:time(6 hours). stat:uni('ResponseTime')
2 where average > 300
```

A EPL também suporta sub-interrogações e análise de frequência de saída. São também suportados padrões de eventos para que seja possível o processamento de eventos complexos. Estes padrões permitem definir relações entre eventos subsequentes - relações "A seguido de B". Outra das possibilidades suportadas pela EPL é a definição de padrões para não eventos. Isto implica que é possível definir, por exemplo, uma interrogação que é correspondida quando não existe um evento A nos últimos dez minutos.

A opção de utilização de uma janela temporal deslizante ao verificar a ausência de eventos é uma funcionalidade que não está disponível em nenhum dos outros motores de processamento analisados.

As interrogações contínuas com janelas temporais implicam guardar em memória os dados necessários para conseguir processar a interrogação, o que implica que quanto maior for a janela temporal, maior é a quantidade de memória necessária.

[20] Figura adaptada de http://www.espertech.com/products/esper.php

O facto da versão *open source* do Esper não possuir suporte de alta disponibilidade implica que, se um processo terminar inesperadamente, os dados em memória serão perdidos.

O Esper suporta uma grande variedade de representação de eventos - Java, XML, mapas ou pares de chave valor. Qualquer objeto Java imutável pode ser utilizado como um evento desde que providencie métodos do tipo *getter* para aceder às propriedades do evento. Os eventos também podem ser representados através da classe java.util.Map. Finalmente, também se podem representar eventos através da classe org.w3c.dom.Node em XML utilizando propriedades em XPath.

Uma das funcionalidades interessantes do Esper, é o facto de poder ser embebido em aplicações Java [MC11]. A versão em C# NEsper é embebível em .Net. A versão comercial do Esper pode funcionar de forma independente.

2.4 Conclusões

2.4.1 Análise de data stores escaláveis

Como o objetivo desta dissertação está diretamente relacionado com o tratamento e armazenamento de grandes quantidades de dados de uma forma contínua a escalabilidade e disponibilidade das bases de dados de suporte são essenciais.

A maioria das bases de dados analisadas são desenhadas de forma a permitir que os dados sejam distribuídos por várias máquinas (nós). À medida que o número de nós aumenta, também aumenta a capacidade da base de dados e o número de operações de leitura e escrita que se conseguem realizar por segundo.

Devido à distribuição dos dados por vários nós, estas *data stores* são altamente disponíveis - se um ou mais nós deixarem de responder, o sistema como um todo mantém-se disponível. Um conceito que está diretamente relacionado com a disponibilidade da *data store* é o de resistência à perda de dados - os dados podem ser replicados através dos nós de forma a assegurar que não se perdem aquando de uma falha de um ou mais nós. Esta resistência à perda de dados é normalmente configurável.

Na tabela 2.1 apresentam-se algumas propriedades das *data stores* analisadas para que possam ser comparadas entre si.

Todas as *data stores* analisadas são capazes de providenciar o armazenamento de dados da solução proposta nesta dissertação. No entanto, existem algumas que melhor preenchem os requisitos necessários em termos de escalabilidade, recuperação automática de falhas, persistência e interligação com outras plataformas.

Devido ao facto de não ser orientada para a persistência em disco, obrigando a configurações algo complexas para poder garantir a persistência dos dados em caso de uma falha de energia, a Redis poderá não ser adequada para ser utilizada como suporte principal dos dados. No entanto, pelo facto de residir em memória, apresenta uma performance elevada ao nível da leitura/escrita que poderá servir para guardar dados de suporte que tenham que ser consultados muitas vezes

Tabela 2.1: Data Stores

Data Store	**Tipo**	**Persistência em disco**	**Recuperação de falhas**	**Escalabilidade**
Project Voldemort	Key-Value Store	Sim	Automática	Sim
Riak	Key-Value Store	Sim	Automática	Sim
Redis	Key-Value Store	In Memory	Através de Redis Sentinel	Sim
Tokyo/Kyoto Cabinet	Key-Value Store	Sim	Não	Não
Couchbase Server	Key-Value e Document Store	Sim	Sim	Sim
CouchDB	Document Store	Sim	Sim	Complexa
MongoDB	Document Store	Sim	Sim	Sim
Cassandra	Extensible Record Store	Sim	Automática	Sim

e que caibam na memória. Para guardar mais dados do que os que cabem em memória seria necessário recorrer a *sharding* ao nível da aplicação.

A Tokyo Cabinet e a Kyoto Cabinet não permitem recuperação automática de falhas de nós, o que implica uma monitorização constante do sistema. Estas também não são facilmente escaláveis, o que as torna desaconselhadas para aplicações que necessitem de escalar facilmente. A escalabilidade da CouchDB é complexa, visto não ser essa uma das suas premissas iniciais.

Pela performance de escrita, pela escalabilidade e pela resistência a falhas, a Cassandra será uma boa solução para a persistência de dados em aplicações semelhantes à que se propõe. Fazendo uso da elevada performance de escrita, é possível escrever os dados processados já no formato necessário para a leitura, de forma a que, na altura em que esta seja feita, não sejam necessárias interrogações complexas. Para além disso, a funcionalidade providenciada pelos contadores é interessante para quando se utilizam métricas cumulativas. O facto de ser escalável horizontalmente permite aumentar o poder de processamento e o espaço disponível, de acordo as necessidades do negócio. É preciso ter, no entanto, em atenção as especificidades desta base de dados no que respeita à utilização do padrão "leitura antes de escrita". É de todo desaconselhado esse formato de acesso aos dados. Em síntese, da análise extensiva das funcionalidades, benefícios e inconvenientes das *data stores* analisadas, a Cassandra mostrou ser a mais adequada ao problema, visto implementar conceitos de escalabilidade e resistência a falhas associados a uma elevada performance de escrita.

2.4.2 Processamento de eventos

As três plataformas de processamento de eventos analisadas são bastante diferentes entre si. De facto, as aplicações comerciais do processamento de eventos (sejam estes complexos ou em fluxo) são tão variadas que criam um alargado ecossistema de soluções[21]. No entanto, este ecos-

[21]Lista de soluções para processamento de eventos: http://epthinking.blogspot.co.uk/2011/12/genealogy-of-event-processing-players.html

sistema é dominado pelos produtos comerciais, principalmente os dos gigantes da indústria como são os sistemas da IBM, da Oracle e da SAP, havendo poucas alternativas *open source* [MV11]. Este facto, aliado ao âmbito deste trabalho, limitou a escolha dos sistemas a analisar.

Duas das soluções analisadas, o S4 e o Storm, partilham grandes semelhanças entre si, principalmente no que respeita ao facto de serem orientados para o processamento de fluxos e de serem extremamente distribuídas e descentralizadas. O S4 é considerado um percursor do Storm e ambos têm um funcionamento semelhante [Hen12].

O S4 é programado em Java e permite definir os filtros de forma extremamente flexível. Esse benefício é contrabalançado pelo facto de não garantir o processamento de todos os eventos. Esta é uma decisão que foi tomada para garantir o desenho mas que tem sérias consequências em sistemas que necessitem de garantias de processamento.

O Storm dá garantia de processamento de todos os eventos, mesmo em caso de falha de nós, o que é uma mais valia quando se fala na deteção de eventos para gerar alarmes ou notificações críticas, ou quando se tratam de sistemas que precisam que os resultados do processamento sejam fiéis aos dados de entrada.

No que diz respeito ao balanceamento de carga de forma dinâmica, este apenas é suportado pelo Storm.

Quer o Storm quer o S4 têm dificuldades em lidar com janelas temporais deslizantes - o Storm obriga a fazê-lo de forma programática e o S4 tem dificuldades a lidar com eventos esparsos.

O Esper é completamente diferente do Storm e do S4 visto que não é um sistema distribuído. A versão *open source* analisada é sim uma biblioteca Java ou C# desenhada para ser integrada noutros sistemas. A mais valia do Esper é o seu motor de processamento de eventos, sendo não só capaz de processar eventos complexos mas também fluxos de eventos através de interrogações contínuas. Aliado a este facto, utiliza uma linguagem de interrogação semelhante ao SQL que o torna extremamente versátil [MV11]. O Esper faz o processamento de todos os dados em memória. Caso a interrogação envolva uma janela temporal deslizante, os dados dos eventos que correspondam à interrogação são guardados na RAM e descartados quando já não pertencerem ao intervalo temporal.

Em síntese, das três soluções analisadas o Storm mostrou-se o mais adequado ao problema visto que, apesar da dificuldade em lidar com janelas temporais, garante o processamento de todos os dados e pode ser expandido de acordo com as necessidades do sistema.

Capítulo 3

Características do sistema atual

Este capítulo descreve como funciona, atualmente, a visualização da performance das vendas na Sonae. Começa-se por descrever qual é a estrutura utilizada na organização dos dados, detalhando-se de seguida quais as métricas mais importantes e os sistemas utilizados para as apresentar. Por fim, é feito um resumo do capítulo.

3.1 Organização

No que respeita à visualização da performance do retalho, a Sonae organiza os elementos analisados de acordo com duas estruturas: a estrutura comercial e a estrutura operacional. A estrutura comercial categoriza os vários produtos vendidos. A estrutura operacional refere-se à organização das diversas superfícies comerciais. De seguida, descreve-se com mais pormenor cada uma destas estruturas.

3.1.1 Estrutura comercial

Na estrutura comercial estão presentes os vários produtos, organizados através de uma estrutura hierárquica. Esta estrutura é, do geral para o particular, composta por departamento, unidade de negócio, categoria, sub-categoria, unidade base e, por fim, produto. Cada um destes elementos da hierarquia é definido de forma a facilitar a pesquisa, visualização e outras tarefas de gestão, principalmente quando se trata de campanhas ou promoções. Na figural 3.1 pode-se ver como se enquadram nesta estrutura três produtos distintos: leite meio gordo; leite magro e ovos. Todos eles pertencem ao mesmo departamento - *Alimentar* - e à mesma unidade de negócio - *Laticínios / Congelados*. Os ovos, no entanto, pertencem a uma categoria diferente da dos leites. Estes, por sua vez, pertencem a unidades base diferentes. Pode-se também verificar na figura que esta estrutura está orientada para ajudar a encontrar produtos semelhantes. É simples listar, por exemplo, todos os diferentes leites UHT magros, visto que estes pertencem à mesma unidade de base.

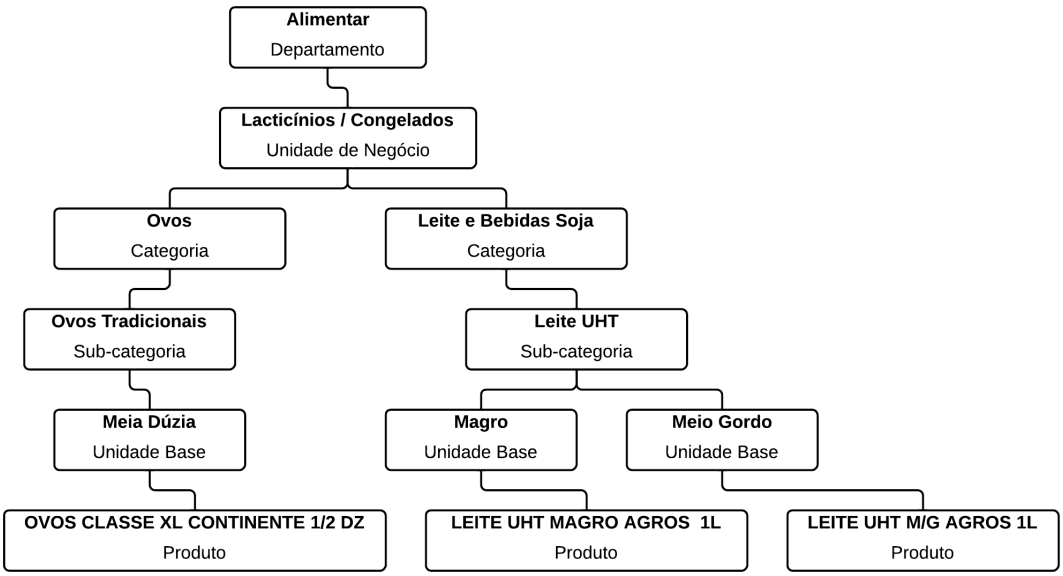

Figura 3.1: Excerto da estrutura comercial.

3.1.2 Estrutura operacional

A outra estrutura utilizada - estrutura operacional - tem como elemento mais baixo a loja. As lojas são organizadas em zonas, que por sua vez são organizadas em insígnias. Uma insígnia consiste numa das várias marcas da Sonae como, por exemplo, o Continente (retalho alimentar), a Zippy (retalho especializado em roupa de criança) ou a Worten (retalho especializado de eletrodomésticos e eletrónica). Uma zona define um conjunto de lojas geograficamente próximas umas das outras. Esta estrutura tem como particularidade o facto das zonas não serem partilhadas por lojas de diferentes insígnias. Mesmo que, por exemplo, uma Zippy e um Continente estejam geograficamente no mesmo local, estes não pertencem à mesma zona. Na figura 3.2 está representada uma parte da estrutura operacional que permite visualizar o lugar nessa estrutura do Continente de Matosinhos e também da Worten das Antas.

3.1.3 Promoções

As promoções são um fator importante na estratégia comercial da Sonae, tendo uma influência significativa no volume de vendas.

As promoções são transversais a ambas as estruturas da organização (operacional e comercial) e dependem sempre destas para serem definidas.

Exemplos de como pode estar configurada uma promoção:

- promoção de um determinado produto,transversal a todas as lojas;

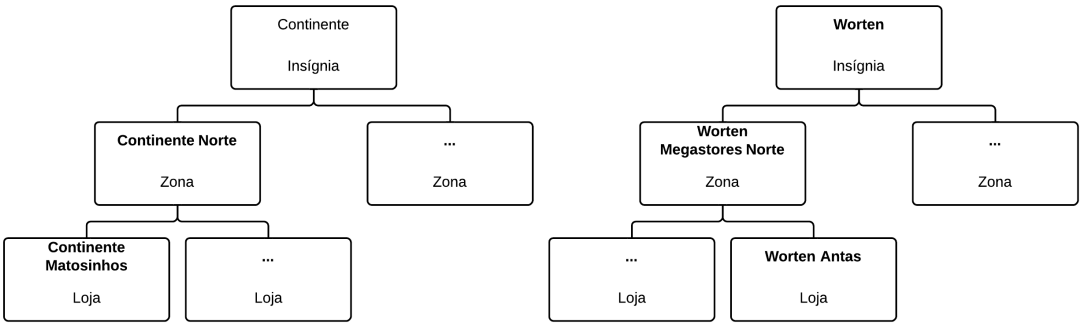

Figura 3.2: Excerto da estrutura operacional.

- promoção que se aplica a todos os produtos de uma determinada loja, relacionando-se assim com a estrutura operacional;

- promoção que se aplica apenas a uma determinada categoria ou a outro nível da estrutura comercial;

- promoção semelhante ao exemplo anterior, englobando apenas uma categoria da estrutura comercial, mas existindo apenas numa loja específica;

Expandindo a definição de uma promoção, obter-se-á um conjunto de pares produto/superfície comercial específicos para os quais a promoção se irá aplicar. Desta forma, é possível ter uma tabela que relaciona os pares produto/superfície com as promoções. Sabendo para cada par quais as promoções a que estes estão sujeitos.

As promoções são definidas com alguma antecedência, fazendo parte da sua definição a especificação do início e do fim das mesmas. No início do dia, é sempre possível saber quais as promoções que irão vigorar nesse dia.

3.2 Indicadores de performance

Como acontece na maioria das áreas de negócio, no retalho são utilizados indicadores para acompanhar a performance das vendas das superfícies comerciais. Estes indicadores de performance são consultados através dos sistemas que se menciona neste capítulo (3.3). Os gestores, diretores e outras entidades utilizam assim estes indicadores quer no planeamento quer no processo de análise e na tomada de decisões.

Descreve-se, de seguida, um subconjunto dos indicadores utilizados atualmente na Sonae que é relevante para a solução proposta.

3.2.1 Volume de vendas

O mais óbvio indicador de performance de uma superfície comercial é o seu volume de vendas. O volume de vendas permite saber qual foi o total de vendas de um determinado elemento num período de tempo definido. É uma métrica de elevada importância visto que está diretamente relacionada com o lucro (mesmo considerando a complexidade do cálculo do lucro na venda a retalho). Olhando para a evolução do volume de vendas de uma loja, produto ou promoção pode-se verificar se este volume está de acordo com a previsão para o período em questão. Esta informação permite ao gestor tomar medidas no caso de uma situação anómala.

3.2.2 Previsão de vendas

Um importante facilitador da análise do volume de vendas é a previsão de vendas que, no caso da Sonae, é efetuada pela equipa de Marketing. Recorrendo à previsão de vendas para um dado período e comparando esta previsão com o valor real é possível identificar situações anómalas em que o desempenho do elemento em questão, seja este uma loja, uma categoria de produtos ou uma promoção, não está de acordo com o esperado. A previsão de vendas é efetuada não para produtos específicos, mas sim para sub-categorias de produtos. Agregando estas previsões é possível saber a previsão de vendas de uma loja ou até de uma insígnia.

3.2.3 Visitantes

O número de visitantes é um dado importante na análise da performance de uma superfície comercial. É definido como visitante todo aquele indivíduo que entra numa superfície comercial, independentemente de, à saída, adquirir ou não um produto. Um visitante passa a ser considerado um cliente quando adquire um bem ou serviço. O número de visitantes é quase sempre superior ao número de clientes, visto que nem todos os indivíduos que entram numa loja adquirem um produto: muitos dos visitantes são apenas acompanhantes de clientes e outros, apesar de possíveis clientes, acabam por sair da loja sem adquirir nada.

O número de visitantes permite compreender se existem de facto possíveis clientes dentro da loja ou se, por algum motivo, estes não estão sequer a entrar na loja. Permite também perceber em que períodos é que uma superfície comercial tem mais ou menos possíveis clientes. No entanto, nem todas as superfícies têm nas entradas os contadores necessários para proceder à contagem dos visitantes: os hipermercados Continente são um exemplo dessa exceção.

3.2.4 Transações Comerciais ou Faturas

Uma transação comercial acontece sempre que um cliente efetua um pagamento de um conjunto de produtos na caixa. Cada transação comercial corresponde a uma fatura e ambas são identificadas através de um número único em todo o sistema. Este é um indicador importante porque, pela quantidade de transações comerciais num dado intervalo de tempo consegue-se perceber quantos visitantes estão, efetivamente, a adquirir produtos.

É importante deixar claro que, no resto deste documento, quando se utiliza o termo transação, não se está a falar de uma transação comercial, mas sim das linhas de uma fatura. Tal acontece porque cada linha de fatura equivale a uma transação de dados. Só se utiliza o termo "transação" nos indicadores de performance porque é este que é utilizado pelas equipas de gestão.

Assim, para evitar ambiguidades, de agora em diante falar-se-á apenas em "faturas", sendo que "transações" referir-se-ão sempre a linhas de uma fatura.

3.2.5 Taxa de conversão

A quantidade de visitantes (*qtdVisitantes*) identifica o número de pessoas que entram na superfície comercial e a quantidade de faturas (*qtdFaturas*) identifica o número de clientes que adquirem produtos na dita superfície. Com estes dois dados é possível obter um rácio que permite saber se uma superfície está a converter muitos ou poucos visitantes em clientes/faturas. Esse rácio é chamado de taxa de conversão (*tConvers*) e a sua fórmula é apresentada na equação 3.1 .

$$tConvers = \frac{qtdFaturas}{qtdVisitantes} \tag{3.1}$$

Atentando a esta equação, compreende-se facilmente que uma taxa de conversão mais alta significa que mais visitantes adquiriram produtos ao invés de saírem sem efetuar nenhuma compra. Esta taxa é sempre calculada para um dado intervalo. É importante referir que a taxa de conversão de uma superfície comercial que tenha 1000 visitantes e 500 faturas seria igual à taxa de conversão de uma outra superfície comercial que tenha 10 visitantes e 5 faturas. Como tal, a taxa de conversão deve ser analisada tendo em conta o número de visitantes e/ou o número de faturas.

3.3 Sistemas

A visualização da performance das vendas no retalho faz parte do dia a dia das equipas de gestão da Sonae que utilizam diversas ferramentas ao seu dispor para consultar detalhes e explorar dados relacionados com as vendas das várias superfícies comerciais.

Na sua maioria, os dados de suporte à informação sobre as vendas estão alojados num armazém de dados (denominado EDW - *Enterprise Data Warehouse*) que é alimentado a partir dos vários sistemas operacionais. Os dados presentes no EDW são a base de várias aplicações que os tratam e apresentam em diversos formatos, de acordo com as necessidades diversas dos vários tipos de utilizadores.

É relevante mencionar que, pelas caraterísticas do EDW e dos processos que o alimentam, a maioria dos dados presentes no EDW são atualizados na madrugada do dia seguinte a serem gerados. Isto significa que os dados consultados referem-se ao dia anterior, o que implica que o tempo de reação a qualquer situação está sempre condicionado a esse desfasamento.

De seguida, apresenta-se uma descrição dos sistemas que utilizam os dados do EDW. Destes, destacam-se dois - o ZOOM e o OBIEE. O sistema proposto nesta dissertação pretende complementar estes dois sistemas. Para além destes, existem outros, que se descrevem brevemente.

3.3.1 ZOOM

Uma das aplicações que se alimenta dos dados do EDW é o ZOOM. Este consiste numa plataforma para exploração analítica da informação desenvolvida sobre Microsoft SQL Server Analysis Services. A exploração da informação OLAP é efetuada sobretudo através do Microsoft Excel. A figura 3.3 apresenta um ecrã típico de Zoom dentro do Excel. A grande vantagem desta componente de exploração é possibilitar a construção de diversas agregações que permitem oferecer uma resposta com performance às diversas interrogações *ad hoc* efetuadas pelos utilizadores finais.

O ZOOM é principalmente utilizado pelos diretores comerciais e analistas de negócio que gerem as diversas gamas de produtos, e pelos diretores de loja que gerem uma ou mais superfícies comerciais. Os principais domínios de informação presentes em ZOOM são: Vendas, Stocks, Transferências, Ruturas, Quebras e Análises de Lucros e Perdas. No ZOOM, os utilizadores têm a possibilidade de analisar a informação sobre diversas perspetivas, sendo que as principais vertentes de análise são Temporal, Comercial (produtos) e Operacional (lojas).

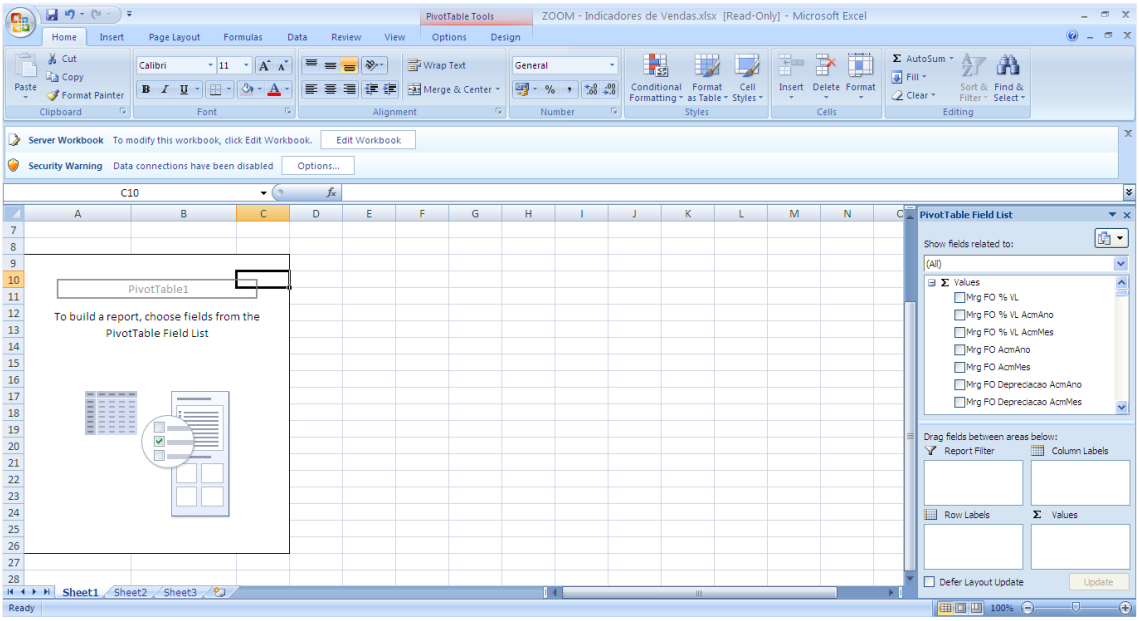

Figura 3.3: ZOOM - Exploração de dados.

3.3.2 OBIEE

Outra aplicação relevante é o Oracle Business Intelligence Enterprise Edition (OBIEE).

O OBIEE é uma plataforma de *business intelligence* da Oracle que permite a construção de *dashboards*/listagens sobre os dados presentes no EDW. Como se pode ver no exemplo da figura 3.4, o interface com o OBIEE é efectuado através de uma página *web*. A utilização do OBIEE é transversal a toda a companhia. É utilizado desde a administração – que consulta *dashboards* executivos e resumidos da atividade da companhia – até aos funcionários das lojas – que efetuam

o controlo dos seus departamentos através dos *dashboards* construídos e disponibilizados para o efeito.

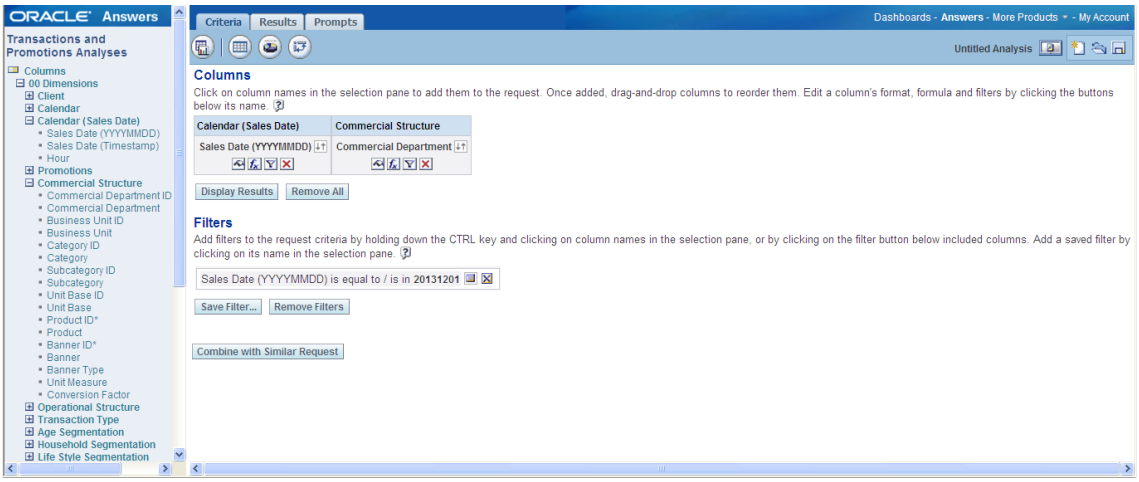

Figura 3.4: OBIEE - Página principal.

3.3.3 Outros sistemas

Para além dos dois sistemas mencionados acima (ZOOM e OBIEE) existe um conjunto considerável de aplicações de nicho que utiliza o EDW e que serve as necessidades específicas de vários departamentos dentro da Sonae. Nesse conjunto estão aplicações de *data mining*, outros *dashboards*, que não os disponibilizados pelo OBIEE, e aplicações estratégicas e de exploração de dados.

É de mencionar, também, o facto do departamento de Marketing interrogar diretamente o EDW (através de SQL) visto que integra quadros técnicos com a capacidade de o fazer. Tal acontece devido às necessidades específicas desse departamento no que respeita à agilidade e à rapidez na obtenção dos dados.

3.4 Resumo

Neste capítulo, apresentou-se o que acontece na atualidade em relação à visualização da performance das vendas das lojas.

Ao analisar os indicadores de performance que são atualmente utilizados, verifica-se que o mais importante é o volume de vendas - valor total de vendas de um produto, loja ou promoção. Outros indicadores como a taxa de conversão e o cumprimento da previsão das vendas também são importantes.

A organização dos dados relativos à performance distribui-se segundo duas estruturas: a estrutura comercial com a hierarquia de produtos e a estrutura operacional com a hierarquia de superfícies comerciais. Esta organização está também relacionada com as promoções, sendo que

estas últimas se aplicam sempre a um subconjunto de produtos que é definido recorrendo a uma ou ambas as estruturas.

Identificaram-se alguns dos principais sistemas em utilização na Sonae para a visualização da performance das vendas e outras métricas associadas. Estes, de entre outras funcionalidades disponíveis, permitem a exploração de cubos OLAP e geram *dashboards* com informação consultada pelas várias áreas, desde a administração aos funcionários das lojas. Existem também outros sistemas, orientados para servir necessidades específicas de algumas áreas. Salvo algumas exceções, todos os sistemas de visualização de performance de vendas funcionam com base num armazém de dados denominado EDW.

Capítulo 4

Requisitos

Neste capítulo, dão-se a conhecer os requisitos, quer funcionais quer não funcionais, do sistema que se pretende ver implementado. Também é efetuada uma análise ao volume dos dados que se pretende processar, devido às repercussões que este volume tem na implementação da solução proposta.

4.1 Introdução

Na fase inicial deste projeto apenas havia uma necessidade identificada: a visualização de dados em tempo real. Com o passar do tempo, esta necessidade foi evoluindo. Esta evolução aconteceu, principalmente, devido ao refinamento das especificações que adveio das várias interações com os diferentes elementos chave e potenciais utilizadores do sistema. Essas interações aconteceram ao longo das várias reuniões de troca de ideias que foram efetuadas durante o decorrer do projeto. Houve também a troca de impressões à medida que iam sendo apresentadas versões preliminares da solução proposta. Também contribuiu para a evolução dos requisitos a investigação efetuada ao estado da arte que permitiu compreender a capacidade das aplicações atualmente a serem desenvolvidas no domínio do *software* "Big Data".

De seguida, apresenta-se a versão atual dos requisitos do sistema a implementar. Estes requisitos estão agrupados em dois tópicos: requisitos funcionais e requisitos não funcionais.

Os requisitos aqui apresentados são requisitos de alto nível, de forma acomodar a evolução da conceção do sistema.

Considera-se necessário fazer uma análise ao volume de dados a processar: o volume de dados atualmente gerado pelos sistemas de faturação das lojas é de uma dimensão considerável, o que terá implicações no desenho da solução proposta.

4.2 Requisitos funcionais

4.2.1 Volume de vendas de uma loja

O utilizador seleciona uma loja e consulta através de um gráfico e de uma tabela o volume de vendas dessa loja. Os dados são agrupados em intervalos de uma hora, permitindo ao utilizador visualizar qual foi o total de vendas da loja para uma determinada hora do dia. A seleção da loja é feita através de uma lista com todas as lojas a que o utilizador têm acesso.

4.2.2 Volume de vendas de um produto de uma loja

O utilizador seleciona um produto de uma determinada loja e consulta, através de um gráfico e de uma tabela, o volume de vendas desse produto. Os dados são agrupados em intervalos de uma hora, permitindo ao utilizador visualizar qual foi o total de vendas do produto para uma determinada hora do dia. A escolha do produto a analisar é conseguida primeiro selecionando a loja onde o produto é vendido e de seguida introduzindo o SKU (número único que identifica um produto) a analisar.

4.2.3 Volume de vendas de uma campanha de loja

O utilizador seleciona uma campanha de uma determinada loja e visualiza, através de um gráfico e de uma tabela, o volume de vendas dessa campanha. Os dados são agrupados em intervalos de uma hora, permitindo ao utilizador visualizar qual foi a performance da campanha para uma determinada hora do dia.

4.2.4 Apresentação em *dashboards*

A informação é apresentada em *dashboards* que permitem visualizar todos os gráficos e tabelas de uma dimensão (loja, produto ou campanha) na mesma página.

4.2.5 Total dos contadores das entradas em loja

Na página onde é possível acompanhar o volume de vendas da loja, é também possível consultar os dados fornecidos pelos contadores de entradas das lojas. O total de entradas por hora está disponível no mesmo gráfico que o volume de vendas de maneira a que se possam comparar os dois. Os valores estão também em formato de tabela. Nas lojas onde não existem dados de contadores, o gráfico e a tabela respetiva não terão informação referente a contadores.

4.2.6 Total de faturas emitidas numa loja

Na página onde é possível acompanhar o volume de vendas de uma loja, é também possível consultar a quantidade de faturas emitidas por essa loja, agrupadas por hora. A informação do número de faturas por hora está disponível no gráfico do volume de vendas e também numa tabela,

lado a lado com o total dos contadores de entradas, de forma a que seja possível fazer comparações entre ambos os valores.

4.2.7 Taxa de conversão de uma loja

Nas lojas em que existam contadores de entradas é possível visualizar, para cada hora, a taxa de conversão dessa loja. Esta informação está disponível numa tabela e também num gráfico à parte. A taxa de conversão é dada pela fórmula em 3.1.

4.2.8 Acumulado de vendas diário

No *dashboard* das lojas, produtos e campanhas é visível um gráfico e uma tabela com a soma, hora a hora, do valor total de vendas diário até essa hora.

4.2.9 Percentagem da previsão de vendas cumprida

Para as lojas é possível verificar qual a percentagem da previsão de vendas que já foi cumprida. Esta informação está disponível através de uma tabela com a evolução do cumprimento da previsão de vendas a cada hora. A previsão de vendas para o dia também é visível no gráfico de vendas acumuladas.

4.3 Requisitos Não Funcionais

4.3.1 Aplicação *web*

A aplicação de visualização de dados é uma aplicação web, está disponível online através da intranet, consultável através de um *browser* de internet.

4.3.2 Escalabilidade

As tecnologias utilizadas no processamento e armazenamento de dados permitem, sempre que possível, escalar a capacidade desse processamento de forma horizontal, ou seja, acrescentar novas instâncias que irão funcionar em paralelo com as atuais e aumentar a capacidade de processamento ou armazenamento.

4.3.3 Tecnologia *Open Source*

As tecnologias a utilizar são *open source*, com o código fonte disponibilizado ao abrigo de uma das várias licenças *open source*. É dada preferência a projetos da fundação Apache pelas garantias que estes apresentam.

4.3.4 Interface com o sistema

A fronteira do sistema é implementada através de filas de mensagens. É através destas que o fluxo das transações é inserido no sistema. Tal faz com que exista uma dissociação entre o sistema a implementar e os sistemas das lojas onde são geradas as transações. Também implementa um paradigma de "*push*"ao invés de "*pull*", de forma a que o sistema a implementar não tenha que estar constantemente a fazer interrogações a um sistema de produção.

4.3.5 Tempo de atualização de dados

O sistema atualiza a informação no mais curto espaço de tempo. Os dados são processados e ficam disponíveis para apresentação logo que são inseridos no sistema através de filas de mensagens. O tempo entre a inserção na fila de mensagens e a apresentação deve ser de tal forma baixo que o utilizador entenda estar a receber a informação em tempo real.

Este requisito não é definido de forma precisa porque um dos objetivos da investigação subjacente a esta dissertação é encontrar formas de conseguir processar e apresentar os dados no mais curto espaço de tempo. Como tal, considera-se que não é ideal estar a definir um valor para o processamento em tempo real, mas sim definir que o sistema deve processar os dados de modo a que o intervalo de processamento seja instantâneo para um utilizador comum.

4.4 Volume de dados a processar

Esta secção não faz parte dos requisitos funcionais ou não funcionais, mas está no entanto inserida na secção de requisitos e constrangimentos do sistema porque é necessário compreender qual será o volume de dados que o sistema terá que processar. Este volume de dados influenciará as várias escolhas efetuadas ao nível da arquitetura, das tecnologias utilizadas e terá repercussões no desenvolvimento do sistema.

O volume de dados a ser tratado é considerável, visto que para a informação fornecida aos utilizadores ter relevância e os requisitos serem cumpridos, o sistema em produção terá de processar continuamente todas as transações geradas nas lojas num intervalo de tempo que permita apresentar a informação ao utilizador sem grande desfasamento.

O fluxo de transações das lojas é, então, a base de todo o sistema e processando este fluxo é possível calcular qual o volume de vendas ao nível comercial, ao nível operacional e ao nível promocional.

Cada elemento do fluxo de transações é uma linha numa fatura, que corresponde a uma leitura do código de barras pelo leitor da caixa registadora. Se o produto não for previamente pesado ou for um produto vendido à unidade, é pesado na balança dessa mesma caixa. Esta transação de caixa não deve ser confundida com uma fatura: após terem sido lidos e processados todos os produtos que o cliente quer comprar, estes são agrupados numa fatura que é emitida e paga pelo cliente. Como tal, todas as transações correspondem a uma fatura.

Para o sistema implementado, os principais dados que caracterizam uma transação são:

Data e hora Estes dois campos definem exatamente quando foi registado o produto na caixa. A granularidade mínima do intervalo é de um segundo.

Produto Este campo contém o SKU do produto vendido. O SKU é um número único que identifica um produto específico.

Local O local onde o produto foi vendido é definido pelo código do local, um campo numérico que identifica uma superfície comercial.

Fatura Uma fatura agrupa um conjunto de produtos que foram vendidos a um determinado cliente. É a fatura que é entregue ao cliente no final das compras. Cada produto vendido está inserido numa fatura. Ao contar quantas faturas existem durante um determinado intervalo de tempo consegue-se saber quantos clientes teve a superfície comercial nesse período.

Valor O valor total que o cliente adquiriu, calculado multiplicando o preço unitário do produto pela quantidade adquirida.

Quantidade A quantidade de produto adquirida. Em unidades ou Kg.

Sinal Na compra normal de um produto, a transação é marcada com sinal positivo, contando para o total de vendas. No entanto, existem transações que são créditos (devido a devoluções), sendo essas transações marcadas com sinal negativo. Visto que estes créditos também fazem parte do fluxo de transações processadas pelo sistema, é necessário ter em atenção qual o sinal de cada transação para que o valor dos créditos seja subtraído ao valor total de vendas.

Como já foi dito acima, o principal volume de dados a processar advém das transações de caixa de todas as superfícies. Estas transações variam bastante entre as várias superfícies comerciais. De seguida, analisa-se o volume diário de transações de forma a perceber qual a quantidade de dados que o sistema terá de processar.

Na tabela 4.1 pode-se ver a quantidade total aproximada de transações de todas as superfícies comerciais para alguns dias dos últimos anos[1]. Dados do departamento de marketing identificam o dia com mais afluência como sendo o dia anterior à véspera de Natal. Assim, a tabela apresentada permite comparar o total de transações do dia 23 de dezembro de 2013 com os totais de outros dias. Visto que 23 de dezembro de 2013 é o dia com maior número de transações, faz sentido escolher essa data para a análise. Nesse sentido, de seguida são analisados os dados desse dia.

A tabela 4.2 apresenta o total de transações, hora a hora, para o dia 23 de dezembro de 2013[1]. Constata-se que o intervalo com maior número de transações é o das 16 às 17 horas (assinalado a negrito na tabela).

Estas 888 000 transações por hora resultam, em média, em 247 transações por segundo. Tal implica que, ignorando os picos que podem acontecer dentro de uma hora e que são compensados por momentos de menor carga dentro desse mesmo intervalo, o sistema deverá conseguir processar, no mínimo, essas mesmas 247 transações por segundo para garantir que consegue lidar com o volume de dados gerados no dia do ano com maior número de transações.

[1] Por serem confidenciais, estes valores são apenas aproximados.

Tabela 4.1: Quantidade de transações diárias em vários dias (todas as superfícies comerciais)

Data	**Quantidade**
2011/12/23	9100000
2012/12/23	8700000
2013/10/11	5700000
2013/12/18	4900000
2013/12/22	8300000
2013/12/23	**9400000**
2013/12/26	3700000
2013/12/30	8800000

No total e à data deste documento, existem 1013 superfícies comerciais. Estas, no dia mais movimentado de 2013, geraram cerca de nove milhões e quatrocentas mil transações. Este valor traduz-se numa média de nove mil transações por superfície (aproximadamente 0.1% do total) , caso as transações acontecessem uniformemente em todas as superfícies. No entanto, tal não acontece visto que há bastantes diferenças entre o volume de vendas/cliente entre as diversas superfícies comerciais, havendo algumas que originam muitas mais transações que outras. Este facto pode ser comprovado através da tabela 4.3 que apresenta as lojas com mais transações no dia 23 de dezembro de 2013 [2]. Verifica-se que apenas dez lojas processam mais de 16% de todas as transações para esse dia, valor que é exponencialmente superior ao esperado (10 x 0.1 = 1%). Esta distribuição não uniforme do número de transações deve ser tida em conta caso seja necessário recorrer a algum tipo de particionamento do sistema, devendo-se ter em conta que existem lojas que irão originar maior volume de dados.

4.5 Resumo

A solução proposta deverá consistir num sistema integrado que capture, processe e armazene os dados necessários de forma a conseguir apresentar em tempo próximo do real, os indicadores de performance das vendas das lojas. Os indicadores devem ser apresentados através um ou mais *dashboards* presentes numa aplicação *web*

Pretende-se um sistema escalável e baseado em tecnologia *open source*, que receba os dados dos atuais sistemas através de filas de mensagens, no sentido de facilitar a interação com diversas tecnologias. O sistema deverá ter em conta o elevado volume de dados a processar. No capítulo seguinte será detalhada a arquitetura e a implementação dos vários componentes deste sistema.

[2]Não se apresentam os nomes das lojas por motivos de confidencialidade.

Tabela 4.2: Transações por hora no dia 23/12/2013 (todas as superfícies comerciais)

Intervalo	**Transações/hora**	**Transações/minuto no intervalo**	**Transações/segundo no intervalo**
00:00 - 01:00	30 000	500	8
01:00 - 02:00	600	10	0
02:00 - 07:00	0	0	0
07:00 - 08:00	200	3	0
08:00 - 09:00	28 000	467	8
09:00 - 10:00	227 000	3783	63
10:00 - 11:00	557 000	9283	155
11:00 - 12:00	799 000	13317	222
12:00 - 13:00	826 000	13767	229
13:00 - 14:00	672 000	11200	187
14:00 - 15:00	607 000	10117	169
15:00 - 16:00	779 000	12983	216
16:00 - 17:00	888 000	14800	247
17:00 - 18:00	**884 000**	**14733**	**246**
18:00 - 19:00	837 000	13950	233
19:00 - 20:00	774 000	12900	215
20:00 - 21:00	609 000	10150	169
21:00 - 22:00	455 000	7583	126
22:00 - 23:00	309 000	5150	86
23:00 - 00:00	136 000	2267	38

Tabela 4.3: Superfícies comerciais com maior número de transações em 23/12/2013

#	**Superfície**	**Transações**	**% de Total de Transações**
1	Loja A	178640	1,89%
2	Loja B	167763	1,78%
3	Loja C	160755	1,71%
4	Loja D	160399	1,70%
5	Loja E	156986	1,67%
6	Loja F	151214	1,60%
7	Loja G	149919	1,59%
8	Loja H	138988	1,47%
9	Loja I	132772	1,41%
10	Loja J	132077	1,40%
	Total	***1529513***	***16,22%***

Capítulo 5

Solução proposta

Neste capítulo, detalha-se a arquitetura e a implementação da solução proposta. Para além de se apresentar o sistema e de se descrever a arquitetura do mesmo, é detalhado cada um dos seus três módulos e descrita a implementação dos componentes que o constituem.

5.1 Visão geral

A solução proposta consiste num conjunto de componentes interligados que interagem para cumprir com todos os requisitos apresentados.

O elemento central do sistema consiste num processador de fluxos de mensagens encarregado de processar, linha a linha, as faturas das lojas à medida que estas vão sendo emitidas. Este processador de fluxos obtém os dados de uma fila de mensagens que é alimentada em tempo real com as linhas das faturas emitidos pelos sistemas das lojas. Os dados, depois de processados, são armazenados numa base de dados não relacional, ficando disponíveis para consulta.

Uma base de dados *in memory* guarda dados estáticos de suporte ao processamento, carregados durante a madrugada por um sistema integrador.

O utilizador consulta os *dashboards* com os indicadores de performance das vendas das lojas através de uma aplicação web. Um servidor REST suporta essa aplicação web, disponibilizando os dados previamente processados e armazenados.

5.2 Arquitetura

Os elementos do sistema são chamados de componentes, sendo que cada componente é desenvolvido numa determinada tecnologia e tem uma função específica, nomeadamente armazenar dados, processar dados, obter dados ou apresentar os dados ao utilizador. No total, existem sete componentes, que são descritos na tabela 5.1.

Nomearam-se os componentes de acordo com a tecnologia ou *framework* utilizada para os desenvolver. Exceção a esta regra são os componentes de *backend* e de *frontend* que utilizam múltiplas tecnologias e foram nomeados de acordo com a funcionalidade que apresentam.

Tabela 5.1: Componentes do sistema

Módulo	Componente	Descrição
Processamento	Cassandra	Base de dados onde são guardados os dados apresentados.
	Storm	Processa o fluxo de transações a partir do RabbitMQ, persistindo os resultados em Cassandra.
	Redis	Base de dados em memória com os dados auxiliares que são utilizados para processamento do fluxo de transações
Visualização	Backend	Backend REST que serve de interface aos dados presentes em Cassandra e também serve como interface para informação presente no EDW.
	Frontend	Aplicação Web que obtém os dados a partir do Backend e os apresenta aos utilizadores através de *dashboards*.
Interface	Camel	Em desenvolvimento, carrega as transações, minuto a minuto, para a RabbitMQ, emulando a chegada de dados reais das lojas. Carrega os dados das promoções para o Redis.
	RabbitMQ	Fila de mensagens que recebe as transações dos sistemas de faturação das lojas, servindo de interface com o exterior.

Dependendo da função que desempenham, os componentes pertencem a um de três módulos: o módulo de processamento que agrupa os componentes responsáveis pelo processamento e armazenamento dos resultados; o módulo de visualização que contém os componentes responsáveis pela aplicação *web*; o módulo de interface onde estão inseridos os componentes que suportam a interligação com os sistemas exteriores. Na tabela 5.1 é também identificado o módulo a que pertence cada componente.

Na figura 5.1 apresenta-se um diagrama com a arquitetura do sistema. No diagrama podem-se ver os módulos do sistema, cada um dos seus componentes e também os elementos exteriores que interagem com o sistema. Também são visíveis os fluxos de dados, internos e externos, que existem entre os vários elementos. Cada fluxo de dados está legendado junto à sua origem e representado com uma cor diferente. A espessura da linha de cada fluxo representa o volume de dados que o constitui.

De seguida, descreve-se cada um dos módulos que compõem o sistema, detalhando-se os componentes de cada um.

5.3 Módulo de Processamento

O módulo de processamento contém não só o componente encarregue de processar as transações das caixas registadoras das lojas como também os componentes que suportam esse processamento. Integram este módulo os seguintes componentes: o sistema de processamento Storm (componente Storm), a base de dados em Cassandra (componente Cassandra) e a base de dados Redis (componente Redis), que a seguir se apresentam.

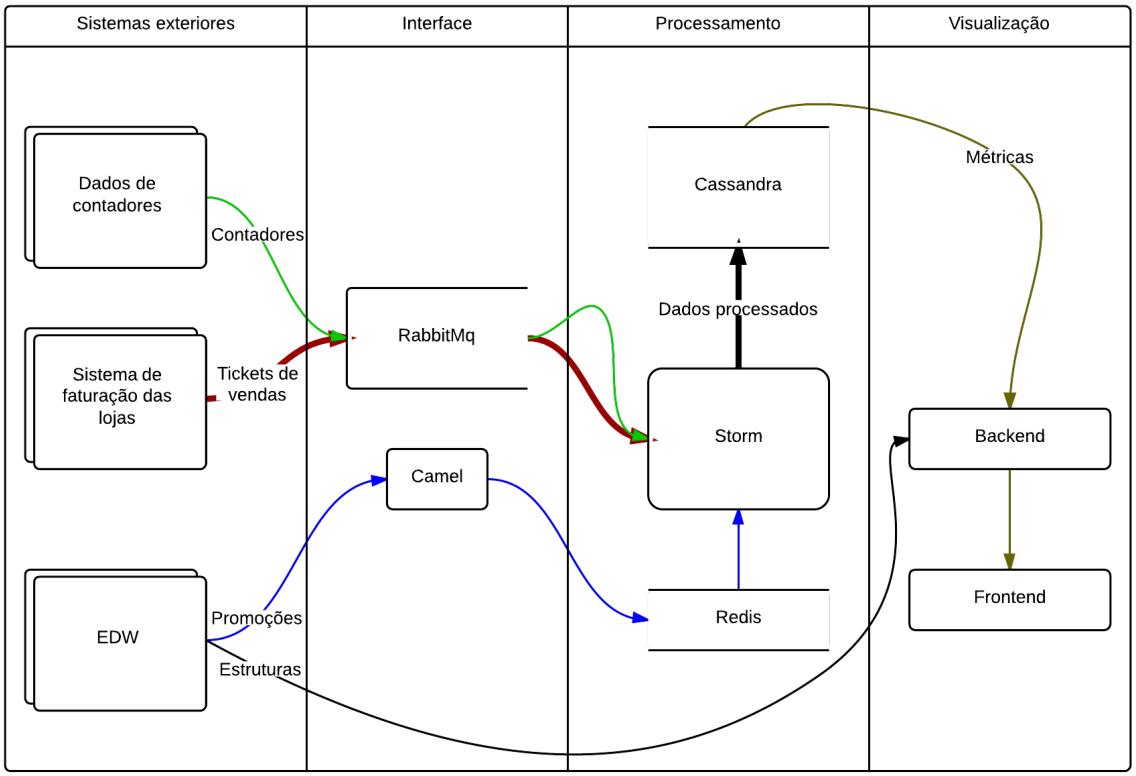

Figura 5.1: Diagrama do sistema.

5.3.1 Storm

Para o processamento de dados em tempo real era necessário uma tecnologia que, para além de processar dados rapidamente, fosse capaz de processar continuamente um fluxo de dados. Com base no estudo feito na revisão bibliográfica, o sistema que melhor preenche estes requisitos é o Apache Storm: assim, na solução proposta optou-se por um *cluster* Storm para efetuar o processamento dos dados. A versão de Storm utilizada é a 0.8.2. Esta era a versão estável aquando do início do projeto.

Como anteriormente referido (2.3.1) o Storm é composto por Bolts (processadores de mensagens) e Spouts (origens de dados) organizados em topologias. Existem várias implementações de Bolts e Spouts que permitem que o Storm se ligue a outras tecnologias.

No sistema implementado, foram necessárias duas topologias, uma para o tratamento do fluxo de dados dos contadores de visitantes, outra para o tratamento dos dados das vendas. Esta última é a topologia que origina maior carga no *cluster*, sendo que a utilização de recursos da topologia que processa os visitantes é negligenciável.

Na figura 5.2 pode-se ver um diagrama com a topologia que foi implementada para tratar os dados das vendas.

As transações de cada loja chegam ao Storm a partir do RabbitMQ, através de uma implementação específica de um Spout: `storm-rabbitmq` [Pat]. Cada tuplo de dados que vem do RabbitMQ consiste numa *string* de XML com os dados de uma venda de um produto específico.

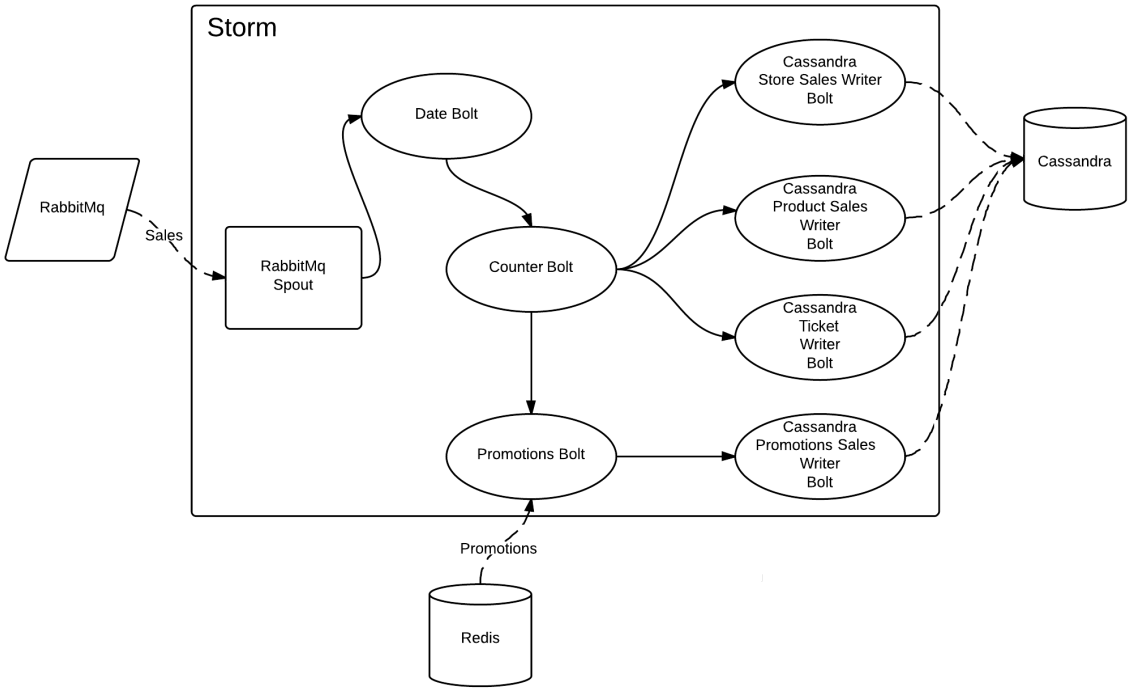

Figura 5.2: Storm - topologia que processa vendas.

Cada tuplo corresponde assim a uma linha na fatura do cliente. Em cada tuplo estã presentes a hora e data a que foi vendido o produto, o identificador do produto vendido (SKU), o sinal (compra ou devolução) e valor da venda bem como a loja e o número de fatura correspondente.

O primeiro Bolt que processa os dados, Date Bolt, identifica a data e hora a que o produto foi vendido e prepara esses dados para serem escritos para o Cassandra. Este processamento é necessário porque os dados, no Cassandra, são agregados em intervalos de hora em hora através de uma chave "data_hora".

Depois de ser gerada a chave temporal a escrever no Cassandra, os dados são de seguida passados ao Counter Bolt sendo que este tem como objetivo preparar os dados para a posterior escrita no Cassandra. Uma das transformações que acontece no Counter Bolt é verificar o sinal das transações e gerar um valor negativo ou positivo de acordo com o facto da transação ser uma compra ou uma devolução. Outra transformação que ocorre é a multiplicação do valor da venda para obter um número inteiro, necessário para a inserção em Cassandra onde os contadores apenas funcionam com números inteiros. Posteriormente, quando são lidos os dados a partir do Cassandra, estes são novamente transformados em números reais.

A partir do Counter Bolt os dados são consumidos em quadriplicado: vão diretamente para um dos três Cassandra Bolts (que os escrevem de imediato para Cassandra) e também são processados pelo Promotions Bolt. Este último acede ao Redis, utilizando a biblioteca Jedis [Lei], e consulta quais as promoções para o produto vendido. Para cada uma das promoções a que um produto pertence, o Promotions Bolt emite um tuplo que é posteriormente guardado pelo Cassandra

Promotions Sales Writer Bolt.

Os Bolts que escrevem os dados em Cassandra são uma implementação específica da biblioteca `cassandra-bolt` [Sci]. Estes Bolts escrevem em Cassandra os resultados do processamento num dos seguintes formatos:

Contadores O valor de uma venda é incrementado nos contadores de volume de vendas correspondentes à loja, ao produto e às promoções do respetivo produto vendido. São estes dados que depois permitem apresentar os indicadores de vendas dos produtos, lojas e promoções.

Conjunto de elementos não repetidos Para cada intervalo temporal são guardados, para cada loja, os números de todas as faturas emitidas. Contando os elementos destes conjunto é possível depois quantas faturas foram emitidas para uma determinada loja. Apesar de o número de uma fatura ser escrito tantas vezes quantas os produtos vendidos nessa fatura, os dados não são repetidos: no Cassandra, caso o número já exista, este é simplesmente re-escrito.

As especificidades relacionadas com o modelo de dados do Cassandra são desenvolvidas na secção dedicada a este componente.

Na figura 5.3 apresenta-se o diagrama da segunda topologia, que processa os dados dos contadores de visitantes presentes nas lojas. Esta é uma versão reduzida da topologia que processa as vendas, partilhando com esta a implementação do Spout de RabbitMQ e do Date Bolt. Assim, para estes elementos não há diferenças do que foi dito na topologia das vendas. Em relação ao Visitors Counter Bolt, este processa os dados das entradas para depois o Cassandra Visitors Writer Bolt incrementar os contadores referentes ao número de entradas por hora de cada loja.

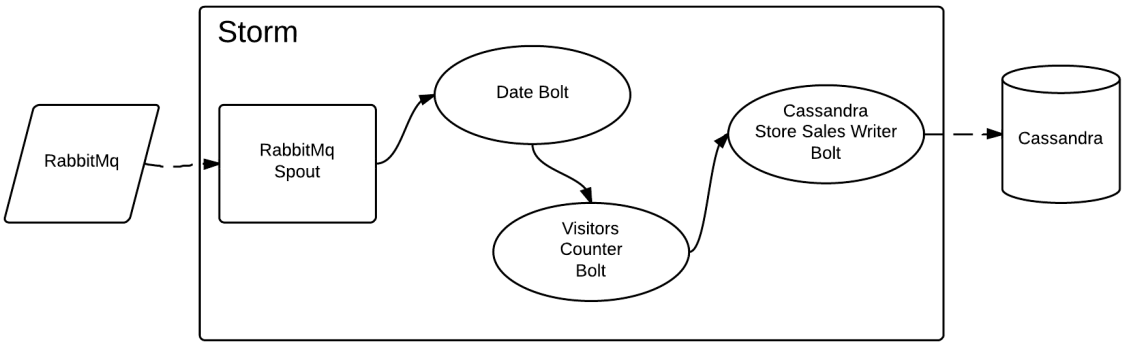

Figura 5.3: Storm - topologia que processa os contadores dos visitantes.

Nos testes efetuados, o Storm consegue processar cerca de 1250 entradas por segundo. Este valor foi alcançado com apenas um só nó num sistema virtual com 4Gb de RAM e um processador QuadCore Intel Xeon E7-4860 a 2.27GHz. Apesar deste valor ser muito superior à carga máxima de menos de 300 entradas por segundo, num sistema em produção é necessário adicionar pelo menos mais um nó para garantir que, mesmo que exista uma falha de hardware, o sistema continua a processar dados.

Em resumo, a solução proposta prova que um *cluster* com apenas duas instâncias de Storm a funcionar em máquinas com especificações modestas é suficiente para garantir o processamento do volume de dados necessário para apresentar os indicadores de performance dos requisitos do sistema.

5.3.2 Cassandra

O sistema proposto utiliza uma base de dados Apache Cassandra para guardar os resultados do processamento efetuado pelo Storm. Nesta base de dados, os dados são guardados num formato não normalizado, orientado para responder às interrogações efetuadas pelo sistema de visualização no momento de apresentação do *dashboard*. São guardados na base de dados contadores, um para cada associação chave/métrica/intervalo e quando existe uma ocorrência de uma destas associações é incrementado o respetivo contador. Os contadores estão sempre prontos a ser lidos sem serem efetuadas agregações/cálculos.

A arquitetura de base de dados utilizada, comum nos sistemas "Big Data", obriga a uma duplicação exponencial dos dados escritos, mas permite depois apresentar os resultados ao utilizador de forma imediata. Pelo contrário, numa base de dados relacional, teriam que ser lidos todos os dados de uma tabela apenas para dar uma contagem de um valor total.

Utilizando como exemplo desta mudança de paradigma o cálculo do total de vendas de uma loja numa determinada hora, um dos mais importantes indicadores de performance que o sistema desenvolvido apresenta, é fácil de compreender a diferença entre a solução apresentada e uma solução baseada numa base de dados tradicional.

No sistema proposto, para cada venda de um produto o Storm incrementa o valor total de vendas da loja nessa hora. Assim, o volume de vendas do Continente de Matosinhos para a décima sexta hora do dia atual está disponível de forma direta, sem ser necessário fazer qualquer cálculo. De notar que os totais são agregados hora a hora, por isso a décima sexta hora começa as 16:00:00 e acaba às 16:59:00.

Num sistema tradicional, em que os dados estivessem guardados numa base de dados relacional, para obter o volume total das vendas do Continente de Matosinhos para a décima sexta hora, ter-se-ia que percorrer toda a tabela das vendas, filtrando os resultados que ocorressem entre as 16:00:00 e as 16:59:59 e que pertencessem à loja de Matosinhos. Desta forma o sistema teria que pelo menos todos os registos vendidos nesse dia (chegam a ser gerados 9 milhões de registos por dia) para obter um conjunto de resultados filtrado e somar o valor de cada um destes para calcular o total.

Os benefícios da solução proposta em relação à utilização de uma base de dados tradicional traduzem-se principalmente na rapidez com que se conseguem disponibilizar os dados ao utilizador. Perde-se, no entanto, flexibilidade nas interrogações que podem ser feitas ao sistema, visto que é necessário que estas tenham sido calculadas de antemão. Num sistema como o que se pretende, este facto não é importante, visto que as interrogações necessárias à apresentação dos *dashboards* não mudam.

A avaliação que foi feita na revisão bibliográfica de bases de dados NoSQL, elegeu o Apache Cassandra como a base de dados mais indicada para o problema proposto. Para tal contribuíram especificações desta base de dados, tais como o suporte a contadores e o suporte à escrita concorrente de uma quantidade elevada de dados. É também uma mais valia para o sistema proposto o facto de ser uma base de dados distribuída e de alta disponibilidade permitindo aumentar linearmente a performance do *cluster* através da adição de mais nós.

Para guardar os dados que são depois apresentados na aplicação *web*, criaram-se no Cassandra várias tabelas (denominadas *column families*). Estas *column families* são de dois tipos:

Dynamic column families Tabelas em que cada linha pode ter um número variável de colunas. Numa linha, todas as colunas têm que ter nomes diferentes, apesar dos dados que contêm poderem ser iguais.

Counter column families Uma especialização das *dynamic column families* em que cada coluna contém um par nome/contador. Incrementar um contador é uma operação atómica e apenas de escrita. Este é o tipo de *column family* mais utilizado, porque os dados que são guardados são, na sua maioria, contagens.

Os contadores atómicos das *counter column families* são imprescindíveis ao sistema. Com contadores atómicos é possível atualizar um contador sem ser necessário ler o valor que este tinha anteriormente. Como o Cassandra não implementa o conceito de transações para garantir que um valor não é alterado entre a leitura e a escrita, esta é a única forma de fazer contagens sem ficar suscetível a condições de corrida (*race conditions*). De outra forma, o valor teria que ser lido, somado em memória e ser novamente escrito. Ao executar esta sequência de operações, num sistema com múltiplos clientes concorrentes, não há garantia que não exista uma alteração ao valor entre a leitura e escrita, originando uma perda de informação.

Ao guardar os dados em Cassandra é utilizado um sistema de segmentação dos dados por períodos de tempo que permite depois saber qual o total de vendas por intervalo com uma única leitura. De acordo com os requisitos, o intervalo utilizado em todas as *column families* é a hora, o que implica que é esta a granularidade dos dados que se conseguem obter nos Dashboards. Esta granularidade não deve ser confundida com taxa de atualização - apesar de os indicadores serem apresentados em intervalos de hora a hora, os dados são atualizados em tempo real. As chaves das colunas do tipo contador são assim uma concatenação da data e da hora. Por exemplo, para uma ocorrência às 18 horas do dia 23 de dezembro de 2013, a chave correspondente seria "2013122318".

Como as colunas em Cassandra são naturalmente ordenadas, é possível obter todas as colunas de um dia com uma única operação, através da especificação de um intervalo. Utilizando o mesmo dia como exemplo, obtém-se todas as vendas até as 18 horas com a interrogação:

```
SELECT 2013122300..2013122318 FROM store_interval_totals WHERE store=1
```

Esta facilidade é amplamente utilizada no sistema para diminuir o número de interrogações necessárias para apresentar um *dashboard*.

Para a persistência dos dados, utilizam-se cinco *column families* de um dos dois tipos já apresentados. Quatro das *column families* são *counter column families* e uma é apenas *dynamic column family*. Na tabela 5.2 apresenta-se cada um das *column families*. De seguida descreve-se em pormenor o propósito de cada uma dessas *column families*.

Tabela 5.2: *Column Families* em Cassandra

Nome	**Tipo**	**Chave Linha**	**Chave Coluna**	**Valor Coluna**
product_store_hourly	Contadores	<Produto>_<Loja>	<Dia><Hora>	Total de vendas para o dia/hora
store_interval_totals	Contadores	<Loja>	<Dia><Hora>	Total de vendas para o dia/hora
tickets_store_hourly	Colunas Dinâmicas	<Loja>_<DataHora>	<Nº de Fatura>	Nenhum (não é necessário)
promotion_store_hourly	Contadores	<Promoção>_<Loja>	<Dia><Hora>	Total de vendas para o dia/hora
store_counters	Contadores	<Loja>	<Dia><Hora>	Total de entradas para o dia/hora

`product_store_hourly` Na *column family* `product_store_hourly` são guardados, para cada hora, os volumes totais de vendas de cada produto vendido em cada loja. Esta coluna é composta por contadores em que a chave de cada linha é a concatenação do código do produto e do código de loja onde esse produto pertence. Para a chave de cada uma das colunas utiliza-se o dia e a hora em que a venda ocorreu.

`store_interval_totals` A *counter column family* `store_interval_totals` armazena, para cada loja, o volume total de vendas a cada hora. Cada linha corresponde a uma loja e cada coluna utiliza como chave o dia e hora da venda, permitindo assim saber para cada hora do dia qual foi o volume de vendas de uma loja.

`tickets_store_hourly` Relembra-se que cada linha processada contém o número da fatura correspondente. Como uma fatura contém muitas linhas, uma linha não equivale a uma nova fatura. Para manter atualizado o número de faturas emitidas numa determinada hora para uma loja é necessário determinar a cardinalidade do conjunto de números de fatura diferentes emitidos nessa hora. A solução mais simples para este problema seria incrementar o contador se o número da fatura processada não estivesse numa tabela de referência com o número de todas as faturas emitidas até então. No entanto, esta solução obrigaria a uma leitura seguida de uma escrita no contador e outra na tabela de referência para inserir o número da fatura. Este formato não é aconselhado no Cassandra devido, por um lado, a este ser otimizado para escrita e, por outro, a não suportar transações.

Assim, para contornar este problema, para cada uma das linhas recebidas o sistema escreve o número da fatura respetiva na *dynamic column familiy* `tickets_store_hourly`. Para

chave de coluna é utilizado o número da fatura. Para chave de linha é utilizada a concatenação do número da loja com a data e hora da venda. Como em Cassandra a chave das colunas é única, os números de faturas repetidas são simplesmente re-escritos, não dando origem a uma nova coluna. Desta forma, existe para cada loja uma linha com todos os números de faturas diferentes emitidas para cada hora. Esta solução faz com que ao escrever os dados não seja necessário ler a base de dados para confirmar se um número de fatura já existe, sendo apenas necessário escrever o número da fatura. No momento em que se quiser saber quantas faturas diferentes existem para uma loja numa determinada hora, basta contar quantas colunas existem na linha correspondente. Esta funcionalidade é fornecida pelo Cassandra e apresenta uma performance suficiente para conseguir ler *dashboard* as contagens de todos os intervalos de um dia para uma determinada loja sem uma demora que seja percetível pelo utilizador.

`promotions_store_hourly` Esta *column family* armazena, para cada promoção presente em cada loja, o volume de vendas de cada hora. Para a chave de cada linha é utilizada a concatenação da chave do produto com a chave da loja. Para a chave das colunas, tal como nas outras *counter column families* é utilizada a concatenação do dia e da hora para identificar o intervalo temporal correspondente.

`store_counters` Os valores dos contadores das lojas são persistidos nesta *counter column family*. É utilizada a chave da loja como identificador das linhas e cada uma destas contém para cada hora uma coluna com o total de visitantes nesse intervalo.

Para o desenvolvimento da solução proposta, foi instalado um *cluster* de Cassandra composto por um só nó instalado numa máquina virtual nos sistemas da Sonae. Esse nó começou por ter um único *core* de um processador QuadCore Intel Xeon E7-4860 a 2.27GHz e 4GB de RAM.

O Storm, em momentos de maior fluxo de dados, processa perto de 300 tuplos por segundo, o que origina, devido à expansão que ocorre no processamento, cerca de 2000 escritas por segundo no Cassandra. Constatou-se que a configuração inicial não era suficiente para conseguir suster a persistência do volume de dados do Storm nas alturas de maior tráfego, sendo necessário aumentar o número de *cores* para quatro. Nesta segunda configuração já foi possível conseguir velocidades de escrita acima das 2500 inserções por segundo, como se pode ver na figura 5.4 que apresenta os dados do programa `cassandra-stress`. Este valor é acima do necessário para suportar a carga do sistema em alturas de maior tráfego.

Tal como acontece no Storm, apesar de um nó ser suficiente para processar os dados, não se consegue a redundância necessária para resistir a falhas de *hardware*,razão porque será necessário adicionar ao *cluster* mais um nó. Essa operação é facilitada pelo facto do Cassandra permitir adicionar e remover nós de forma simples e com o sistema em funcionamento.

Em resumo, o Cassandra provou ser uma boa solução para a persistência dos dados, com uma performance suficiente para armazenar todos os dados provenientes do Storm e fornecer esses dados em tempo real ao Backend para posterior apresentação nos *dashboards*.

```
Averages from the middle 80% of values:
interval_op_rate          : 2512
interval_key_rate         : 2512
latency median            : 7.1
latency 95th percentile   : 39.8
latency 99.9th percentile : 313.9
Total operation time      : 00:07:19
```

Figura 5.4: Cassandra - resultado do teste de stress.

5.3.3 Redis

Sempre que uma linha é processada pelo Storm, este precisa de saber quais as promoções que existem para o par produto/loja presente nessa linha. Os dados referentes às promoções em vigor, necessários para obter essa informação, não mudam durante o dia e como tal são carregados no sistema durante a madrugada. Para guardar esses dados foi necessário utilizar uma base de dados em que a leitura fosse rápida o suficiente para acompanhar a velocidade de chegada dos dados ao Storm.

Considerando a análise efetuada na revisão bibliográfica, decidiu-se utilizar o Redis para guardar os dados das promoções, apesar de este não dar garantias em caso de falha de *hardware*. Pesou nesta decisão o facto de o Redis, por ser uma base de dados *in memory*, ter uma performance de leitura bastante elevada, ideal para guardar dados de referência consultados até cerca de trezentas vezes por segundo. Outro fator a favor do Redis é que existem bibliotecas estáveis de Redis para Camel e para Java (utilizada no Storm). A primeira é utilizada em Camel para carregar, todas as madrugadas, os dados das promoções a partir do EDW. Antes do carregamento dos dados, o Redis é esvaziado através de uma entrada presente em Cron.

Para cada par promoção/loja é guardada no Redis uma entrada com todas as promoções correspondentes. Os dados são armazenados através de *Sets*, para que não existam, na mesma entrada, promoções repetidas. As chaves das entradas são caracterizadas pelo prefixo "promo" concatenado com a data, o código do produto e o código da loja. Cada entrada contém os códigos das promoções a que o produto está sujeito na loja em questão. Um exemplo de uma entrada para o dia 23 de dezembro de 2013, produto com código 1111 e loja 2 pode ser o seguinte:

```
 smembers promo:20131223:1111:2
1) "12345"
2) "13579"
3) "67890"
```

Neste caso, o comando `SMEMBERS` foi utilizado para obter todos os membros de uma entrada. Esta entrada contém três promoções: 12345, 13579 e 67890.

Os dados das promoções ocupam em memória menos de 100 MB, por isso os recursos de *hardware* necessários para o Redis são bastante reduzidos - na implementação da solução proposta foi

utilizada uma máquina virtual com apenas um *core* de um processador QuadCore Intel Xeon E7-4860 a 2.27GHz e 4GB de RAM. Através de testes efetuados verificou-se que, nesta configuração, o Redis consegue processar cerca de cem mil leituras por segundo, o que significa que este sistema será suficiente mesmo que o volume de dados a processar aumente consideravelmente. Não obstante este facto, sugere-se que, em produção, seja utilizada uma arquitetura redundante, utilizando um segundo nó de Redis a replicar os dados do primeiro, para garantir a disponibilidade do Redis em caso de falha de *hardware* [SN].

O Redis é assim uma peça fundamental para conseguir efetuar, sem perdas de velocidade de processamento, o cruzamento dos dados que entram em Storm com dados de referência como as promoções. A sua performance é elevada, havendo margem para suportar o cruzamento com outros conjuntos de dados.

5.4 Módulo de Visualização

O módulo de visualização é responsável pela apresentação dos dados aos utilizadores. É composto por dois componentes: o Frontend e o Backend. O Frontend apresenta aos utilizadores a aplicação *web* com os *dashboards*, enquanto o Backend suporta, através de REST, a interligação do Frontend com o EDW e o Cassandra.

5.4.1 Frontend

O Frontend consiste numa aplicação web que apresenta ao utilizador os *dashboards* com os indicadores de performance das vendas. É com este componente que os utilizadores interagem, podendo ser considerado a camada de apresentação de todo o sistema. O Frontend não se liga diretamente às bases de dados, utilizando sempre o Backend (através de um interface REST) para obter dados.

O Frontend foi inteiramente desenvolvido em HTML e JavaScript e utiliza para a criação dos gráficos a biblioteca Google Charts [1].

Os *dashboards* estão divididos em cartões, cada cartão contendo um gráfico ou uma tabela. No topo de cada *dashboard* há um cartão especial com a informação do elemento que esse *dashboard* foca - produto, promoção ou loja.

O *dashboard* de cada uma das lojas é acessível através de uma página com a listagem de todas as lojas. Nessa página, o utilizador pode escolher visualizar a listagem de acordo com a estrutura operacional ou, então, através de uma tabela com possibilidade de filtragem. Para cada loja da listagem é também apresentado o valor total de vendas atual e a previsão de vendas para esse dia. Na figura 5.5 apresenta-se um exemplo desta página.

A figura 5.6 contém um exemplo de um *dashboard* de uma loja. No topo do deste pode-se ver o cartão com a informação da loja: nome, código, data, local e telefone. Também no cartão de topo está a previsão de vendas para esse dia, o total de vendas acumuladas, o valor e intervalo do

[1] *Site* da biblioteca Google Charts: `http://developers.google.com/chart/`

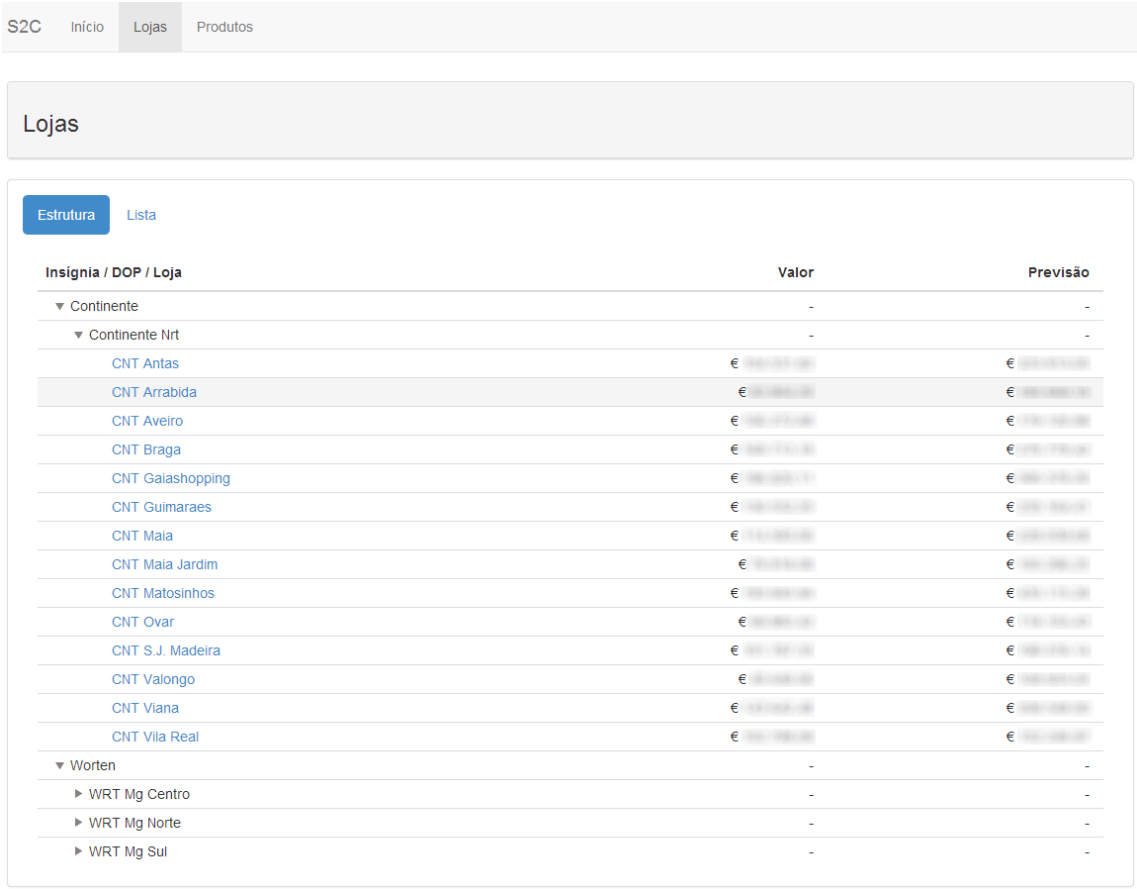

Figura 5.5: Frontend - Listagem de lojas.

maior volume de vendas e o total de visitantes e de transações comerciais (faturas). O primeiro gráfico disponível contém a informação hora a hora de três indicadores: o volume de vendas, o total de visitantes o total de transações comerciais. Selecionando uma coluna é possível visualizar a taxa de conversão dessa hora. No cartão ao lado, os dados presentes neste gráfico estão disponíveis também no formato de tabela. O segundo gráfico disponível permite acompanhar a evolução do total de vendas diário, relacionando-a com a previsão de vendas para esse dia. À direita deste encontra-se a tabela com as vendas acumuladas para cada hora e a respetiva percentagem da previsão de vendas já alcançada. Acima desta tabela encontra-se um manómetro com uma indicação da percentagem da previsão de vendas que já foi alcançada.

A partir deste *dashboard* é possível aceder ao *dashboard* de um produto dessa loja inserindo um código de produto no menu que é apresentado depois de premir o botão "Produto" no cartão do topo. Este *dashboard* tem o mesmo formato que o *dashboard* de uma loja, mas não apresenta nem os dados da previsão das vendas nem dos visitantes, visto que estas informações não estão disponíveis para os produtos.

Premindo o botão "Promoções" no *dashboard* de uma loja é apresentada uma lista com todas as promoções dessa loja, tal como se pode ver na figura 5.7. Esta listagem pode ser filtrada através

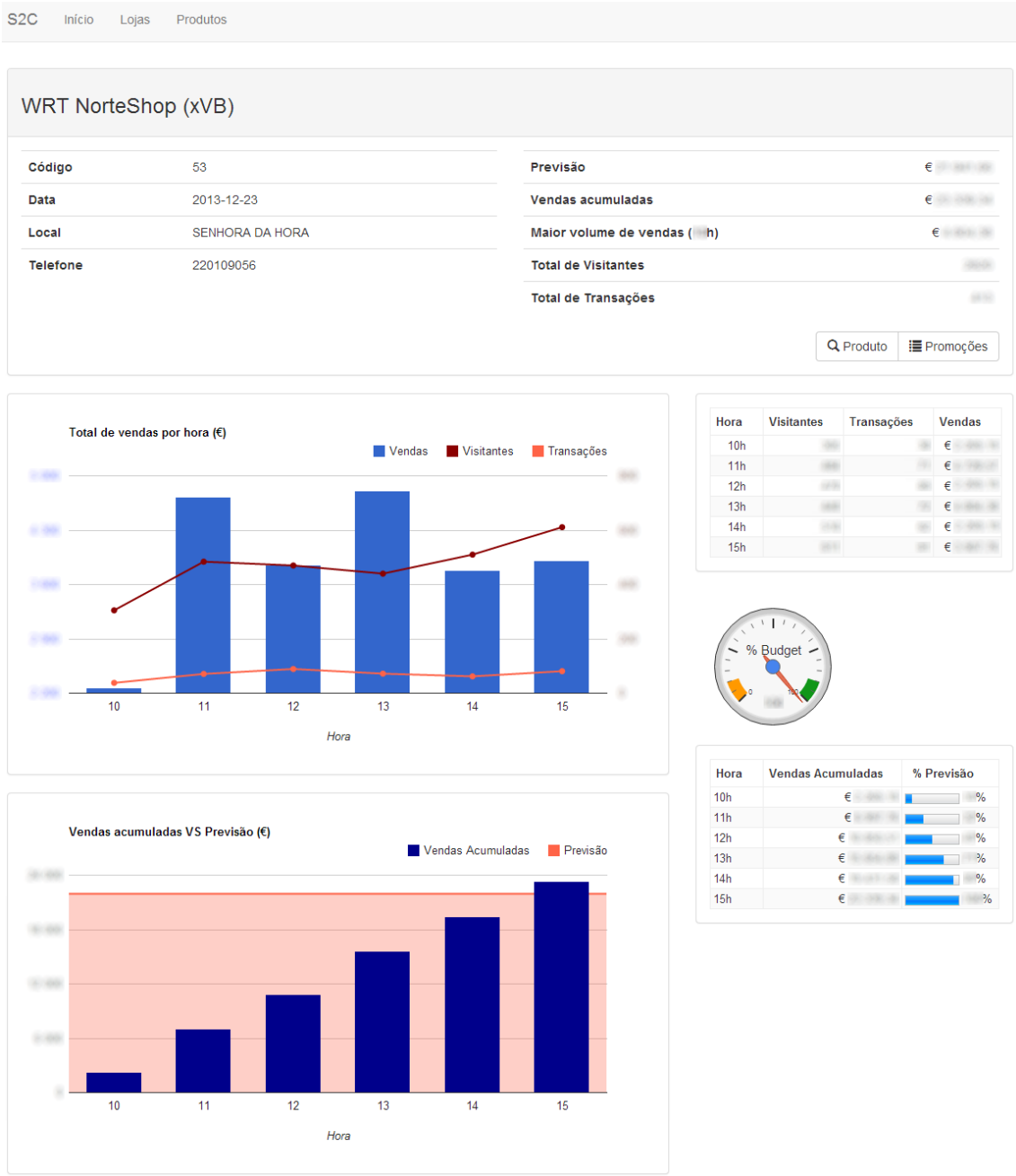

Figura 5.6: Frontend - *Dashboard* de uma loja.

da caixa de texto para o efeito disponível no canto superior direito.

Escolhendo uma promoção da listagem de promoções, acede-se ao *dashboard* com a performance dessa promoção para essa loja. Tal como para os produtos, este *dashboard* contém as vendas por hora e os respetivos totalizadores, não apresentando nem os dados referentes à previsão das vendas nem os dados dos visitantes.

Os dados necessários para apresentar os gráficos e tabelas são obtidos de forma assíncrona utilizando JavaScript (mais especificamente a biblioteca jQuery) através de um interface REST

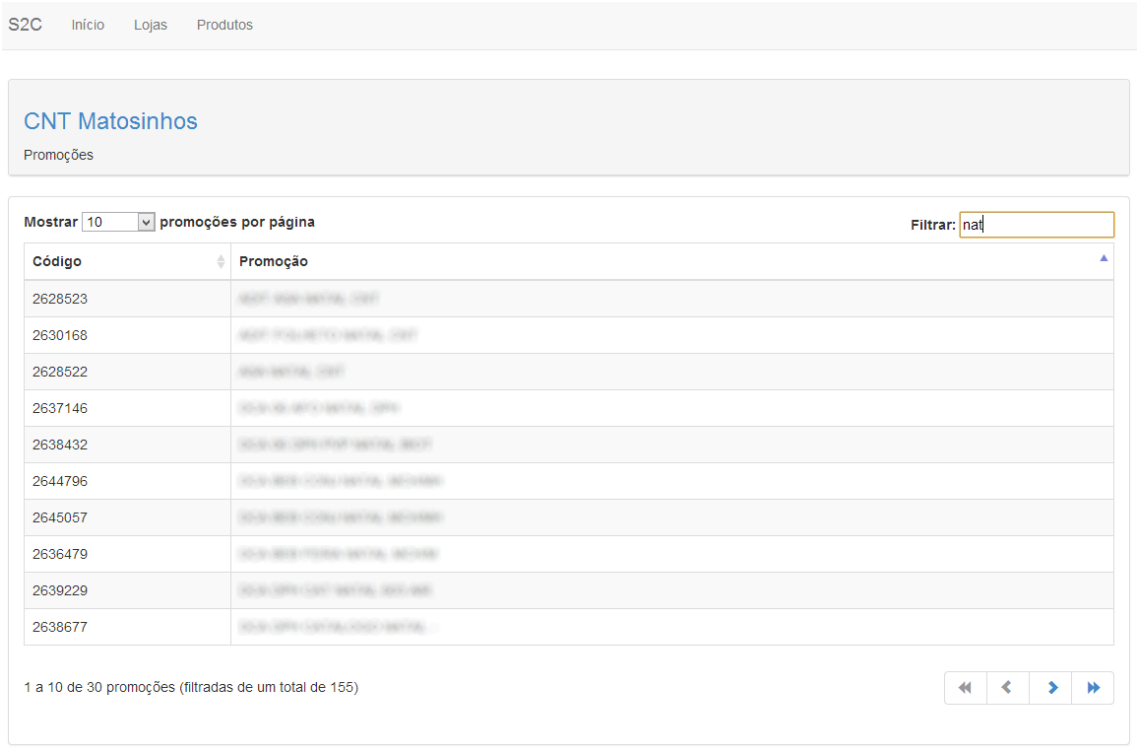

Figura 5.7: Frontend - Listagem de promoções de uma loja.

providenciado pelo servidor de Backend.

O Frontend está encapsulado num arquivo WAR para que possa ser disponibilizado através de servidores aplicacionais como o Glassfish ou *web servers* como o Tomcat. Isto torna-o adequado para ser disponibilizado em redes empresariais e faz com que não seja necessária a instalação de um servidor *web*, podendo utilizar o mesmo servidor que o Backend.

Em resumo, o Frontend disponibiliza aos utilizadores, de uma forma simples e intuitiva, informação crítica para a gestão do negócio das lojas Sonae.

5.4.2 Backend

O Backend é uma aplicação que implementa o conceito REST (Representational State Transfer) [Fie00] e contém toda a lógica de negócio do módulo de visualização, servindo de interface entre o Frontend e os dados presentes no Cassandra e no EDW da Sonae. Assim, o Backend liga-se ao Cassandra para obter os dados relativos à performance das vendas e também ao EDW para obter dados relacionados com o negócio (características e listagens de produtos, lojas e promoções) e serve esses dados através de JSON ao Frontend.

O Backend é desenvolvido utilizando a *framework* Spring[2]. A aplicação tem três tipos de componentes:

[2] *Site* da Spring Framework: `http://projects.spring.io/spring-framework/`

Controllers Tratam da interação com o Frontend, mapeando os vários *endpoints* REST, invocando os métodos apropriados dos Services e apresentando os resultados em JSON.

Services Onde reside a lógica de negócio. Estes componentes adquirem dados através dos DAO, tratando-os e retornando-os aos Controllers.

Data Access Object (DAO) Responsáveis pelo acesso aos dados. Estes componentes interagem com o EDW e com o Cassandra para obter os dados necessários à aplicação.

Um diagrama da arquitetura desta aplicação pode ser visualizado na figura 5.8.

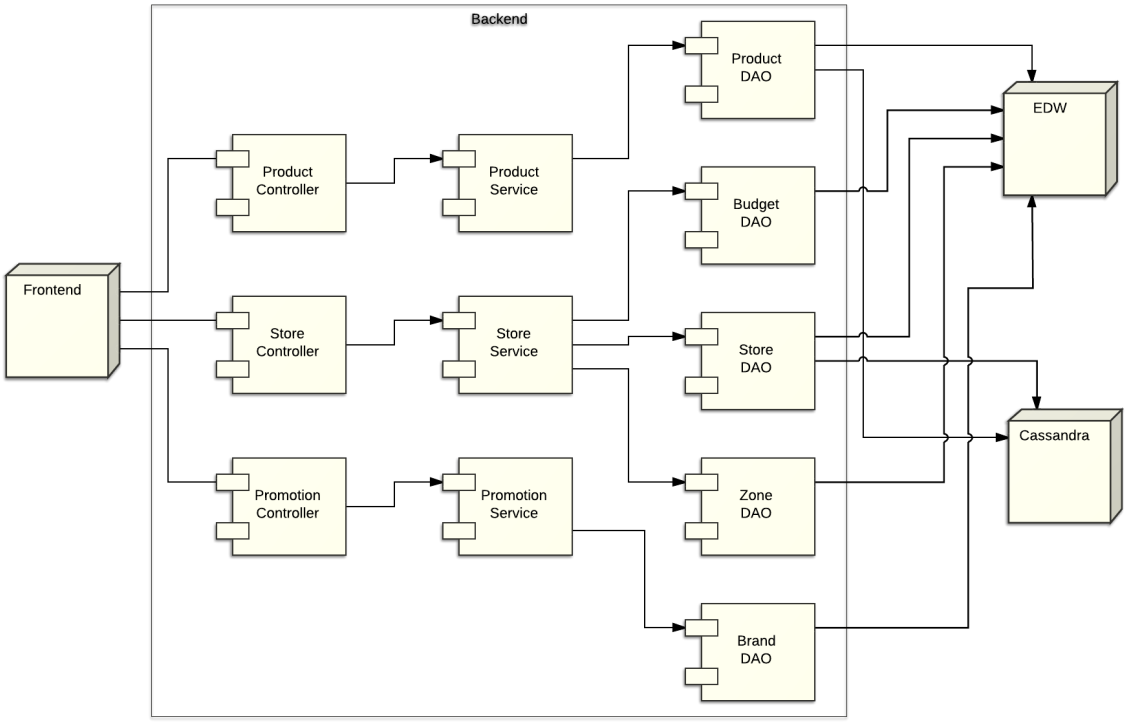

Figura 5.8: Backend - Diagrama.

O Frontend liga-se a três Controllers: o Product Controller, o Store Controller e o Promotion Controller. Cada um destes chama os métodos do Service respetivo.

O Product Controller é responsável pelos *endpoints* dos produtos. Estes endpoints permitem saber se um produto existe, quais os detalhes de um produto e qual o volume de vendas de um produto numa loja. O Product Controller invoca os métodos presentes no Product Service que, por sua vez, obtém os dados do EDW e do Cassandra através do Product DAO.

O Store Controller é responsável pelos *endpoints* das lojas. Estes *endpoints* permitem ao Frontend obter, através dos métodos presentes no Store Service, o volume de vendas de uma loja, as listagens das lojas existentes, a hierarquia operacional das lojas (Zona e Insígnia), os dados dos contadores das lojas e os dados das faturas das lojas. O Store Service liga-se ao EDW e ao Cassandra através de quatro DAO, o Store DAO que obtém dados do EDW e do Cassandra e o Budget DAO, o Zone DAO e o Brand DAO que obtêm dados apenas do EDW.

O Controller responsável pelos *endpoints* referentes às promoções é o Promotions Controller. Este invoca os métodos disponibilizados pelo Promotions Service que comunica com o EDW através do Promotions DAO. Com este controller consegue-se obter as promoções em vigor.

Os DAO utilizam, para comunicar com as bases de dados, duas bibliotecas: a Netflix Astyanax[3] para comunicar com o Cassandra e a Spring-Data-JPA[4] (a funcionar sobre o driver JDBC da Oracle) para comunicar com o EDW (uma base de dados Oracle).

Em resumo, o Backend utiliza o potencial da framework Spring para estabelecer uma camada lógica que serve de ponte entre o Frontend e os dados. Ao disponibilizar os dados através de um interface REST torna-se possível reutilizar este componente para que, no futuro, se possam desenvolver outros componentes de visualização como aplicações móveis e de *desktop*.

5.5 Módulo de Interface

O módulo de interface é responsável pela comunicação com os sistemas exteriores. Neste módulo existem dois componentes: o RabbitMQ, uma fila de mensagens que recebe o fluxo de transações das lojas e o Camel, um sistema de transporte de dados que efetua a importação de dados para o sistema.

5.5.1 RabbitMQ

De forma a que o carregamento de dados no sistema fosse modular foi implementado um componente para servir de fronteira entre os sistemas das lojas que geram o fluxo de transações e o Storm. O sistema escolhido foi o RabbitMQ, um mediador de mensagens que implementa filas de mensagens de acordo com o protocolo AMQP - *Advanced Message Queuing Protocol*[5]. A tecnologia AMQP é largamente utilizada no mercado empresarial [Vin06] e existem vários adaptadores para este tipo de filas de mensagens, na maioria dos Enterprise Service Bus, como acontece no JBoss Fuse[6] (anteriormente FuseESB) e no MuleESB[7]. O facto de existirem também bibliotecas de interligação com AMPQ para a maioria das linguagens faz com que seja simples implementar sistemas que se interliguem ao RabbitMQ.

A cada venda efetuada pelo sistema de faturação das lojas, o sistema de gestão das caixas envia as linhas da fatura resultante - em tempo real - para um *exchange* (uma espécie de endereço) do RabbitMQ. Esse *exchange* entrega as mensagens numa fila de mensagens para depois serem consumidas pelo Storm. O mesmo processo acontece para os dados dos contadores de entradas das lojas sendo que estes também são carregados para um *exchange* e disponibilizados ao Storm através de uma fila. Na figura 5.9 pode-se ver um diagrama com o funcionamento do sistema.

O RabbitMQ tem uma performance bastante alta: uma instância consegue processar mais de 6000 mensagens por segundo. Este valor é mais do que o suficiente para o sistema proposto,

[3] *Site* da Astyanax: https://github.com/Netflix/astyanax
[4] *Site* da Spring-Data-JPA: http://projects.spring.io/spring-data-jpa/
[5] *Site* do RabbitMQ: http://www.rabbitmq.com
[6] *Site* do JBoss Fuse: http://www.redhat.com/products/jbossenterprisemiddleware/fuse/
[7] *Site* do MuleESB: http://www.mulesoft.org/

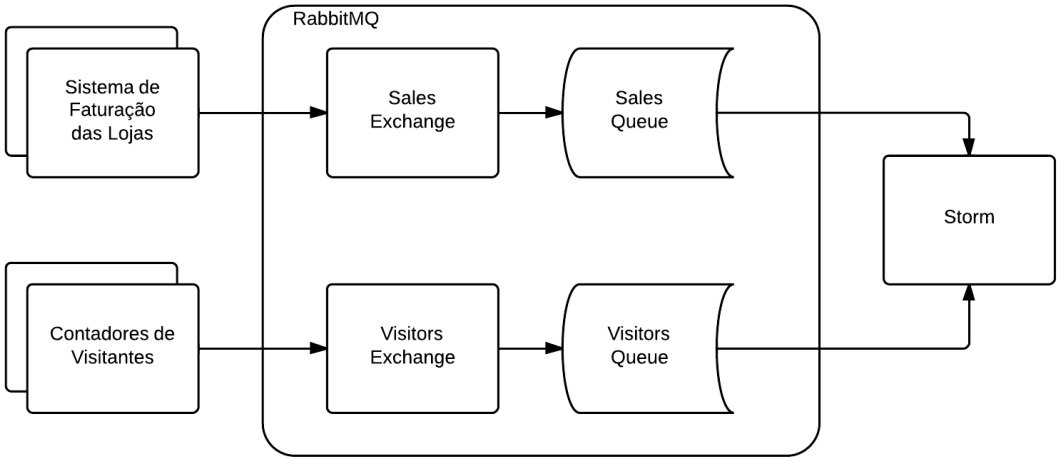

Figura 5.9: RabbitMQ - Diagrama.

deixando bastante espaço para a evolução do sistema. De forma a implementar sistemas de alta disponibilidade, é possível adicionar nós ao sistema, sendo até possível adicionar estes nós enquanto o sistema está em funcionamento.

O RabbitMQ estabelece uma fronteira do sistema, permitindo uma flexibilidade na origem dos fluxos a processar e fazendo com que a inserção de dados no Storm seja efetuada através de um interface baseado numa tecnologia estável e com grande difusão como é o AMQP.

5.5.2 Camel

O Apache Camel[8] é um sistema de transporte e tradução de mensagens que efetua a importação para o sistema, a partir do EDW, dos dados estáticos referentes aos produtos em promoção. A importação destes dados acontece todas as madrugadas, de forma a que os dados estejam disponíveis quando as lojas entram em funcionamento. Os dados importados são armazenados no Redis e posteriormente utilizados pelo Storm.

O Camel é também o responsável por carregar dados de teste para o RabbitMQ, simulando um fluxo de dados reais a partir de dados históricos presentes no EDW.

A arquitetura do Camel baseia-se em dois conceitos: rotas e componentes. As rotas definem qual a origem dos dados, quais as transformações a efetuar nesses dados e quais os destinos dos dados. Estas rotas foram programadas em Java, utilizando uma *Domain Specific Language* específica de Camel. Os componentes permitem interligar o Camel a origens e destinos de dados ou adicionar funcionalidades como processadores ou tradutores às funcionalidades base.

As rotas em Camel são bastante compactas e as poucas linhas de código apresentadas em seguida são responsável pela rota de importação dos dados das promoções do EDW para o Redis..

[8] *Site* do Apache Camel: `http://camel.apache.org/`

```
from("quartz2://myGroup/myTimerName?cron=0+0+5+1/1+*+?+*")
    .to("sql:{{sql.selectOrder}}")
     .split(simple("${body}")).streaming().parallelProcessing()
     .to("redisParamCreator")
.to("spring-redis://10.126.96.211:6379?serializer=#redisserializer");
```

Esta rota utiliza um componente `sql` que permite ler os dados a partir de uma base de dados SQL[9], neste caso, o EDW e outro componente de Camel, o `spring-redis`, que permite escrever em Redis[10] os dados das promoções. Para que o processamento seja mais rapido é utilizado o processamento paralelo através do comando `.parallelProcessing()`. Utilizou-se o componente de agendamento `quartz2`[11] para executar esta rota todos os dias durante a madrugada.

Por serem sistemas de produção, os sistemas de faturação das lojas têm regras bastante estritas e assim não foi possível ligar diretamente este sistemas à solução proposta, como se espera que aconteça quando esta entrar em produção. O Camel foi assim uma peça fundamental no desenvolvimento da solução porque permitiu simular o fluxo de dados das lojas. Preparou-se uma rota em Camel que insere os dados históricos das transações efetuadas (disponíveis no EDW), minuto a minuto, no RabbitMQ como se fossem dados reais de produção. Dessa forma, foi possível testar a performance do sistema com cargas iguais às reais, utilizando-se os dados do dia 23 de dezembro de 2013 de forma a testar o sistema para o dia com maior maior carga. Desta forma foi possível demonstrar e utilizar o sistema como se este estivesse a funcionar em produção, com dados reais, o que permitiu averiguar especificidades em relação à visualização e ao fluxo de dados que de outra forma não seriam possíveis de compreender.

Em resumo, o Apache Camel contribui de duas formas para a solução final que se propõe: (1) importa os dados das promoções de forma a estarem disponíveis no Redis para serem consultados pelo Storm e (2) simula a chegada de dados das lojas para testar o sistema com um fluxo de dados semelhante ao de produção.

5.6 Resumo

O sistema proposto é um sistema heterogéneo caraterizado por vários componentes. Para processar dados é utilizado o Apache Storm, que atualiza os totalizadores presentes na base de dados NoSQL Apache Cassandra. Para auxiliar o Storm no processamento dos totalizadores das promoções é utilizada uma base de dados *in memory*, o Redis. Esta contém todas as promoções a que um produto está sujeito. Os dados presentes em Redis são carregados para o sistema todas as madrugadas utilizando o Apache Camel, um integrador que obtém as promoções a partir do EDW. É também o Camel que é utilizado para simular a chegada das vendas e dos visitantes das lojas

[9] Camel - componente SQL: http://camel.apache.org/sql-componente.html
[10] Camel - componente Redis: http://camel.apache.org/redis.html
[11] Camel - componente Quartz2: http://camel.apache.org/quartz2.html

ao sistema, visto que não foi possível importar os dados diretamente a partir destes sistemas. A apresentação dos totalizadores ao utilizador final é feita por um Frontend em HTML5 e AJAX, que desenha os gráficos e tabelas dinâmicas com recurso a Google Charts. Os dados para o Frontend proveem de um serviço REST implementado pelo Backend, que faz a ponte entre o Frontend e os dados necessários: sejam estes os dados atualizados em tempo real presentes no Cassandra ou os dados estruturais disponíveis no EDW.

Mesmo utilizando *hardware* com pouca capacidade, o sistema foi capaz de cumprir todos os requisitos e processar o fluxo de dados necessário para apresentar ao utilizador, em tempo real, os indicadores da performance das vendas para todas as superfícies comerciais do universo Sonae.

Capítulo 6

Conclusões

Na Sonae, a consulta dos indicadores de performance das vendas das superfícies comerciais faz parte integrante da gestão dessas superfícies. Diariamente, as equipas de gestão analisam o desempenho das lojas, produtos e promoções através de métricas como o volume de vendas (seja este de uma loja, dos seus produtos ou das promoções a que estão sujeitos), a taxa de conversão (rácio ente clientes que entram numa loja e faturas emitidas) e a percentagem de cumprimento da previsão de vendas para uma loja. A análise destes indicadores é de uma importância fulcral para perceber não só o desempenho das superfícies comerciais como um todo, mas também o desempenho dos produtos e respetivas promoções que nelas são comercializados. Atualmente, os indicadores mencionados são disponibilizados através de dois sistemas, o ZOOM e o OBIEE. A informação apresentada nestes dois sistemas provém de um armazém de dados, o EDW. Os dados presentes no EDW proveem de vários sistemas que, na sua maioria, apenas transferem novos dados para o EDW durante a madrugada. Este desfasamento entre a altura em que os dados são criados e a sua transferência para o EDW, onde podem ser consultados, condiciona o tempo de reação das equipas que dependem destes dados.

Sendo a visualização dos indicadores crítica para a gestão eficiente das superfícies comerciais, a Sonae entende que existem benefícios em eliminar o desfasamento que existe atualmente entre o momento das vendas e a disponibilização dos indicadores aos utilizadores. O sistema a desenvolver deverá, assim, capturar o fluxo de dados das transações das lojas, processar esses mesmos dados e armazenar os resultados desse processamento, para que depois possam ser apresentados aos utilizadores, em tempo real, através de uma aplicação *web* com *dashboards* simples e fáceis de consultar. Os dados chegarão das lojas como um fluxo de linhas de fatura, cada linha correspondendo à venda de uma quantidade de um determinado produto. O sistema deve ser capaz de processar estes dados e apresentar, ao utilizador, os indicadores que mais se adequem a uma consulta em tempo real. São estes indicadores: o volume de vendas de uma loja, produto ou promoção; a percentagem da previsão de vendas de uma loja que já foi cumprida; a taxa de conversão (rácio de entradas *versus* faturas) de uma loja. Os dados devem ser apresentados através

de gráficos e tabelas, quer agrupados em intervalos de uma hora quer totalizados até ao momento da consulta.

Para cumprir estes objetivos foi necessário estudar e selecionar sistemas e tecnologias *open source*, um requisito da Sonae, que permitissem processar e armazenar o elevado volume de dados em questão (cerca de 9 milhões de transações diárias). As tecnologias analisadas pertencem ao domínio do "Big Data" - tecnologias orientadas para o tratamento de grandes quantidades de informação. Foram analisados sistemas capazes de processar elevados volumes de mensagens em tempo real e bases de dados NoSQL capazes de armazenar os resultados desse processamento. Dessa análise surgiram as várias tecnologias e sistemas que foram utilizados na implementação de cada um dos componentes da solução proposta.

Para processar o fluxo de transações das lojas, foi escolhido o Apache Storm, um sistema de computação distribuída em tempo real. Este sistema é capaz de processar fluxos de dados de grande dimensão, em tempo real e com garantia de processamento de todos os elementos. O Storm funciona em modo *cluster*, permitindo, através da adição de nós, aumentar a capacidade de processamento ou obter maior redundância.

Os dados processados pelo Storm têm que ser persistidos para que possam ser consultados, sendo para tal necessário uma base de dados capaz de suportar o elevado fluxo de dados a que o sistema estará sujeito. A base de dados escolhida para o efeito foi a Apache Cassandra, uma *data store* NoSQL com uma performance de escrita elevada. Outras características desta base de dados são a sua tolerância a falhas de *hardware* e a possibilidade de aumentar a sua performance através da adição de instâncias ao *cluster*.

Para processar os dados, o Storm necessita de uma base de dados onde guardar dados de referência a ser consultados durante o processamento. A escolha desta base de dados recaiu sobre a Redis - uma base de dados *in memory* caracterizada pela sua elevada performance. Esta característica é essencial para que a consulta dos dados não atrase o processamento do Storm.

Para ligar o sistema a implementar e os sistemas de faturação das lojas era necessário implementar uma fila de mensagens, de forma a conseguir uma comunicação assíncrona entre ambos e permitir um sistema mais modular e flexível. Para este propósito foi selecionado o RabbitMQ, um mediador de mensagens de alta performance que implementa filas de mensagens acessíveis através do protocolo AMQP. Para além de conseguir lidar com um fluxo de mensagens com uma cadência elevada, este sistema é também fiável, sendo inclusive utilizado na indústria bancária e na de telecomunicações.

Como integrador foi utilizado o Apache Camel. A flexibilidade desta tecnologia permite que seja utilizada quer para integrar os dados de referência como também para simular a chegada de dados ao efetuar testes ao sistema.

Escolheu-se implementar os *dashboards* de visualização dos dados em HTML5 e JavaScript assíncrono, utilizando-se a biblioteca Google Charts para facilitar o desenho dos gráficos e tabelas dinâmicas. Os dados necessários aos *dashboards* são disponibilizados através de um servidor REST, encarregado de comunicar com as fontes de dados necessárias.

Com todas as tecnologias descritas foi desenvolvido um sistema com uma arquitetura modular. Os três módulos que compõem esta arquitetura são: o módulo de processamento, que processa o fluxo de dados e armazena os resultados desse processamento, o módulo de visualização que apresenta os indicadores aos utilizadores e o módulo de interface que permite que o sistema interaja com outros sistemas da Sonae.

O módulo de processamento de dados configura-se em três componentes. O principal é um *cluster* Storm, que processa o fluxo de linhas de faturas das caixas e também o fluxo das leituras dos contadores de visitantes das lojas. Para auxiliar este processamento, os dados das promoções a que os produtos estão sujeitos são disponibilizados através de uma instância de Redis, que é consultada sempre que o Storm processa uma linha de fatura. A cada elemento processado, os dados resultantes são armazenados na base de dados escolhida para o efeito, a Apache Cassandra. Nesta base de dados utilizam-se colunas de contadores que permitem que a atualização dos dados numéricos (como os visitantes e os volumes de vendas) seja efetuada em apenas uma operação, eliminando o risco de erros devido a acessos concorrentes. Uma particularidade dos dados presentes em Cassandra é serem guardados de forma não normalizada, já no formato necessário às interrogações efetuadas pelo módulo de visualização. Isto implica uma elevada duplicação de dados mas torna desnecessária as agregações e contagens durante a consulta dos mesmos, diminuindo o tempo de resposta de cada consulta e tornando imediata a consequente apresentação desses dados aos utilizadores.

O módulo de interface é composto por dois componentes: uma instância de Apache Camel e uma fila de mensagens RabbitMQ. O Camel trata do carregamento, para o Redis, dos dados das promoções a partir do EDW, o armazém de dados da Sonae. O RabbitMQ, uma fila de mensagens de alta performance, permite que os sistemas de faturação das lojas introduzam o fluxo de dados das vendas no sistema para depois serem processadas pelo Storm. Durante a fase de desenvolvimento e testes, o Camel foi também utilizado para simular a entrada de dados no sistema. Tal foi necessário porque, devido a preocupações com a estabilidade e segurança dos dados, não foi possível ligar o fluxo de dados dos sistemas em produção das lojas à solução desenvolvida.

No que respeita à disponibilização dos indicadores, em tempo real, aos utilizadores finais, esta está a cargo do módulo de visualização, que é composto por dois componentes: o Backend e o Frontend. O Frontend é uma aplicação *web* desenvolvida em HTML5 e JavaScript que permite aos utilizadores visualizar os dados dos indicadores através de cartões contendo gráficos e tabelas implementados recorrendo à biblioteca Google Charts. O Frontend obtém os dados necessários através de um interface REST disponibilizado pelo Backend. Este obtém, por sua vez, os dados a partir do Cassandra (indicadores) e do EDW (dados de estrutura).

Em resumo, as linhas de fatura e as contagens dos visitantes entram no sistema através do RabbitMQ, são processadas pelo Storm e cruzadas com as promoções presentes em Redis, carregadas diariamente pelo Camel. Os resultados do processamento são armazenados em Cassandra num formato não normalizado para que a consulta dos mesmos seja rápida. Estes dados são disponibilizados pelo Backend através de um interface REST para depois serem apresentados ao utilizador pelo Frontend, uma aplicação web com *dashboards* compostos por cartões de gráficos e tabelas

com os indicadores das vendas.

As principais dificuldades encontradas no desenvolvimento e implementação da solução proposta ocorreram em duas frentes: na compreensão dos modelos de dados e na dificuldade em testar a interação da solução com aplicações de produção. No que diz respeito à dificuldade em compreender o modelo de dados, esta foi ultrapassada com a ajuda dos vários membros das equipas de Databases e de Knowledge. O apoio destas equipas foi indispensável para ultrapassar muitas das dificuldades que surgiram na compreensão do funcionamento complexo dos fluxos de dados da Sonae, em particular dos sistemas de faturação das lojas e do EDW. Em relação à dificuldade em interagir com os sistemas de produção, recorreu-se ao Apache Camel para fazer a emulação do sistema das lojas com dados históricos presentes no EDW. Sempre que necessário foram também utilizados ambientes de pré-produção.

O sistema proposto cumpriu todos os requisitos, funcionais e não funcionais, propostos pela Sonae. Os testes ao sistema provaram que este é capaz de processar os dados para o dia com maior volume de transações diárias (perto de trezentas transações por segundo) e apresentar em tempo real todos os indicadores definidos nos requisitos da Sonae. Apesar de ser um ponto subjetivo, considera-se importante o facto de todos os elementos da Sonae que testaram o sistema, realçarem a facilidade com que os *dashboards* permitiam a consulta dos indicadores. Por tudo isto considera-se que, após a implementação de um sistema de controlo de acessos, a solução desenvolvida pode ser utilizada em ambiente de produção sem modificações de maior, fornecendo aos administradores, diretores e analistas de negócio a capacidade de acompanhar em tempo real o desempenho das vendas.

Desenvolvimentos Futuros

Um sistema como o proposto está sempre em constante evolução devido às alterações das necessidades de negócio e dos respetivos indicadores a visualizar. Por esse motivo, os melhoramentos ao sistema passarão, numa primeira fase, pelo acrescentar de outros indicadores que os utilizadores entendam necessários e críticos para uma melhor gestão de negócio.

Numa segunda fase, entende-se que o sistema pode evoluir de um sistema de visualização para um sistema de monitorização, gerando alertas quando uma situação anómala é detetada. Por exemplo, se o volume de vendas de uma promoção for abaixo do esperado, o sistema pode alertar o diretor comercial da loja onde está a acontecer essa anomalia.

Para além dos desenvolvimentos específicos às vendas, a arquitetura utilizada neste sistema pode ser utilizada para acompanhar outros indicadores de performance. Como exemplo, pode-se utilizar o sistema para o acompanhamento do estado dos stocks e das encomendas das mercadorias aos fornecedores.

As hipóteses de evolução do sistema proposto no contexto atual da Sonae são imensas, sendo o aproveitamento e expansão das suas potencialidades no apoio à estratégia de negócio fatores decisivos para a competitividade, tanto atual como futura.

Referências

[AADE11] Divyakant Agrawal, Amr El Abbadi, Sudipto Das e AJ Elmore. Database scalability, elasticity, and autonomy in the cloud. *Database Systems for Advanced Applications*, 2011.

[ACNM06] Ben Azvine, Zhan Cui, DD Nauck e B Majeed. Real time business intelligence for the adaptive enterprise. In *E-Commerce Technology, 2006. The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, The 3rd IEEE International Conference on*, page 29. IEEE, 2006.

[ASK07] Aditya Agarwal, Mark Slee e Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007.

[Cat11] R Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 2011.

[DB12] Chad Deloatch e Scott Blindt. NoSQL Databases: Scalable Cloud and Enterprise Solutions. Technical report, University of Illinois at Urbana Champaign, 2012.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall e Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, dec 2007.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[FTD+12] Avrilia Floratou, Nikhil Teletia, David J DeWitt, Jignesh M Patel e Donghui Zhang. Can the elephants handle the nosql onslaught? *Proceedings of the VLDB Endowment*, 5(12):1712–1723, 2012.

[Gho10] Debasish Ghosh. Multiparadigm Data Storage for Enterprise Applications. *IEEE Software*, 27(October):57–60, 2010.

[GZK05] Mohamed Medhat Gaber, Arkady Zaslavsky e Shonali Krishnaswamy. Mining data streams: a review. *ACM Sigmod Record*, 34(2):18–26, 2005.

[HBG10] Annika Hinze, Alejandro P Buchmann e IGI Global. *Principles and applications of distributed event-based systems*. Information Science Reference, 2010.

[Hen12] Martin Hentschel. *Scalable systems for data analytics and integration*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 20295, 2012, 2012.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P Junqueira e Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, page 11, 2010.

[KAB+11] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos e Nectarios Koziris. On the elasticity of nosql databases over cloud management platforms. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2385–2388. ACM, 2011.

[KM12] Ivo Koga e Claudia Bauzer Medeiros. Integrating and processing events from heterogeneous data sources. In *Proceedings VI eScience Workshop-XXXII Brazilian Computer Society Conference*, 2012.

[KR81] Hsiang-Tsung Kung e John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[Lei] Jonathan Leibiusky. Jedis: a blazingly small and sane redis java client. https://code.google.com/p/jedis/. Acedido em 2013-12-11.

[Mal05] Shadan Malik. *Enterprise dashboards: design and best practices for IT*. John Wiley & Sons, 2005.

[MBM09] Marcelo RN Mendes, Pedro Bizarro e Paulo Marques. A performance study of event processing systems. In *Performance Evaluation and Benchmarking*, pages 221–236. Springer, 2009.

[MC11] Alessandro Margara e Gianpaolo Cugola. Processing flows of information: from data stream to complex event processing. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 359–360. ACM, 2011.

[MT06] Gerasimos Marketos e Yannis Theodoridis. Measuring performance in the retail industry (position paper). In *Business Process Management Workshops*, pages 129–140. Springer, 2006.

[Muh11] Yousaf Muhammad. Evaluation and Implementation of Distributed NoSQL Database for MMO Gaming Environment. Master's thesis, Department of Information Technology, Uppsala University, 2011.

[MV11] Vuk Mijović e Sanja Vraneš. A survey and evaluation of cep tools. In *Proceedings of the 2011 YU INFO Conference*, 2011.

[Nag12] Kristian A. Nagy. Distributing complex event detection. Master's thesis, Imperial College of Science, Technology and Medicine, June 2012.

[NRNK10] Leonardo Neumeyer, Bruce Robbins, Anish Nair e Anand Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.

[Pat] Peter Pathirana. Storm-rabbitmq: a library of tools for interacting with rabbitmq from storm. https://github.com/ppat/storm-rabbitmq. Acedido em 2014-01-17.

[PPS11] Rabi Prasad Padhy, Manas Ranjan Patra e Suresh Chandra Satapathy. Rdbms to nosql: reviewing some next-generation non-relational databases. *International Journal of Advanced Engineering Science and Technologies*, 11(1):15–30, 2011.

[Sci] Health Market Science. Storm cassandra integration. https://github.com/hmsonline/storm-cassandra. Acedido em 2013-11-23.

[See09] Marc Seeger. Key-Value stores: a practical overview. *Computer Science and Media*, pages 1–21, 2009.

[SK92] Mukesh Singhal e Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47–52, 1992.

[SN] Salvatore Sanfilippo e Pieter Noordhuis. Redis. http://redis.io. Acedido em 2013-10-15.

[SSK11] Christof Strauch, Ultra-Large Scale Sites e Walter Kriha. Nosql databases. http://www.christof-strauch.de/nosqldbs.pdf, 2011. Acedido em 2013-11-15.

[Sut10] Olli Sutinen. *NoSQ - Factors Supporting the Adoption of Non-Relational Databases*. PhD thesis, University of Tampere, 2010.

[TB11] Bogdan George Tudorica e Cristian Bucur. A comparison between several NoSQL databases with comments and notes. *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, pages 1–5, June 2011.

[Vin06] Steve Vinoski. Advanced message queuing protocol. *Internet Computing, IEEE*, 10(6):87–89, 2006.

[WWH+06] Hugh J. Watson, Barbara H. Wixom, Jeffrey A. Hoffer, Ron Anderson-Lehman e Anne Marie Reynolds. Real-time business intelligence: Best practices at continental airlines. *Information Systems Management*, 23(1):7–18, 2006.