Universidade do Minho

Pedro Filipe Linhares Gomes

Migração de aplicações legadas para bases de dados NOSQL

Tese de Mestrado
Mestrado em Informática
Trabalho efectuado sob a orientação de
Doutor José Orlando Roque Nascimento Pereira

Setembro 2011

Este trabalho foi parcialmente financiado pela PT Inovação e pelo projecto ReD– Resilient Database Clusters (PDTC / EIA-EIA / 109044 / 2008).

Declaração

Nome: Pedro Filipe Linhares Gomes

Endereço Electrónico: pedrolinharesgomes@gmail.com

Telefone: 914864464

Bilhete de Identidade: 13196146

Título da Tese: Migração de aplicações legadas para bases de dados NOSQL

Orientador: Professor Doutor José Orlando Pereira

Ano de conclusão: 2011

Designação do Mestrado: Mestrado em Engenharia Informática

Universidade do Minho, 5 de Junho de 2011

Pedro Filipe Linhares Gomes

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.

Rudyhard Kipling

Agradecimentos

Primeiramente o meu orientador, Professor José Orlando Pereira, pela disponibilidade demonstrada, bem como por todas a ajuda prestada. Agradeço igualmente ao professor Rui Oliveira pela ajuda e aconselhamento fornecidos durante a minha estadia no grupo de Sistema Distribuídos.

Um agradecimento à PT Inovação e em particular ao Engenheiro José Bonnet pela sua compreensão e ajuda durante o projecto. Do mesmo modo gostaria de agradecer ao Engenheiro Marco Martins pelo seu acolhimento na minha curta estadia nos escritórios da empresa, e também ao restante pessoal pela ajuda prestada.

A nível pessoal agradeço também aos amigos que comigo embarcaram nesta etapa de formação. Em especial ao Emanuel Braga e Matheus Almeida, amigos e modelos de dedicação e aos meus eternos colegas de laboratório e de amizade Ricardo Gonçalves, Miguel Araújo e Nelson Gonçalves pela ajuda e pelos bons momentos proporcionados neste percurso. Não esquecendo também todas as outras pessoas que neste processo estiveram presentes, o meu obrigado para o Tiago Oliveira, Mario S. João, Rui Quelhas, Luís Zamith, João Bordalo, Ricardo Xing, Rui Castiço, Carlos Silva, André Carvalho e Waldir.

Agradecimentos ao alunos de doutoramento e mestrado do laboratório de sistemas distribuídos pela ajuda fundamental em muitas áreas a mim desconhecidas e interessantes discussões proporcionadas. Um especial obrigado ao Nuno Carvalho, uma ajuda sempre presente nos mais diversos assuntos e companhia vital em parte do trabalho realizado. Do mesmo modo gostaria de agradecer ao Francisco Cruz e Ricardo Vilaça pela sua troca de conhecimento em grande parte deste caminho. Um obrigado final à Idoia Ruiz-Fuertes pela partilha de informação que muito me ajudou nesta dissertação.

Um agradecimento especial aos meus pais e irmãos pelo suporte e encorajamento neste longo processo e por sempre me incentivarem nos meus estudos mesmo durante os maus momentos.

Resumo

Enfrentando o actual crescimento exponencial do volume de dados originados pelos serviços Web, assiste-se hoje a uma revolução no mundo das bases de dados. De facto, a procura por soluções que permitam de forma escalável a persistência de grandes volumes de dados, levou ao recente aparecimento de soluções como o Dynamo ou a Cassandra caracterizados pelos seus modelos de coerência e de uso. Com arquitecturas desenhadas para enfrentar cenários de falha, tal como novos modelos de dados construídos para abrigar a actual natureza dinâmica da informação, estas são hoje vistas como uma alternativa viável às tradicionais bases de dados relacionais.

No entanto, a mudança para este novo paradigma é hoje um desafio. Os seus novos modelos de dados e interfaces de utilização obrigam uma mudança radical de mentalidade dos programadores quando oriundos do modelo relacional. Tais soluções delegam também para o lado do programador novas responsabilidades no seu uso com a ausência de garantias transaccionais, a perda das relações explicitamente expressas nos dados e o controlo de parâmetros como as definições de coerência por operação.

Nesta dissertação pretendemos assim avaliar e tentar resolver esta separação entre os paradigmas, relacional e não relacional, observando através de um caso concreto quais as alterações exigidas no modelo e operações. Partindo do modelo tradicional, observa-se o modo como a mudança para uma solução não relacional afecta o desenvolvimento do caso de estudo ao nível do modelo, complexidade de implementação e performance.

Com base nesta avaliação propomos assim o desenvolvimento de uma solução de mapeamento de objectos. Esta fornecerá uma abstracção da camada de dados subjacente permitindo ao programador uma mais fácil construção de aplicações escaláveis. Através do desenvolvimento deste componente, pretende-se assim a criação de uma solução que una a escalabilidade de uma base de dados não relacional e a interface de programação característica das soluções de mapeamento de objectos.

Abstract

As a result of the current exponential growth of the Web and associated data and services, we assist today to a profound revolution in database management systems. New database systems like Cassandra or Dynamo are emerging as a response to the need of large data storage systems. Based on architectures that embrace eventual failure scenarios and novel data models built to deal with the dynamic nature of Web data, these new systems represent today a viable alternative to relational databases

Nonetheless, the change to these new systems doesn't come without a cost. To the developer, these systems with their novel models and API represent a necessary change of mindset when departing from traditional databases. In fact, they imply new responsibilities for him, as he now faces the maintenance of data relations on the client side, lower transactional guaranties and the new complexity associated with factors such as the consistency definitions.

The basis to this dissertation is then to evaluate and propose a solution to the gap between the new and the old paradigms. Departing from a relational solution we assess how the change to a non-relational product affects the development of an actual use case in terms of the used model, programming complexity and performance.

Based on this assessment, we then present an object mapping solution, that while abstracting the underlying data layer, offers the developer a method for the construction of scalable systems. Through its development we expect to combine the scalability of a non relational database and the simple programming interface of object-relational mapping solutions.

x

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Numa época de crescimento da área Web, as empresas deste sector são obrigadas a lidar com quantidades cada vez maiores de dados em rápido crescimento. Tendo afectado inicialmente gigantes como Google e Facebook, esta é actualmente uma preocupação para muitas empresas que enfrentam novos problemas e desafios derivados dos milhões de utilizadores que acedem aos seus serviços. [1] Não sendo os tradicionais sistemas de gestão de base de dados relacionais (RDBMS na sigla anglófona) uma solução viável para este problema, novos sistemas surgiram assim num mercado que durante 40 anos permaneceu inalterado. Mas esta é apenas a face mediática do problema, uma vez que empresas da área bancária ou das telecomunicações tem muitas vezes também elas de lidar com informação referente a milhões de utilizadores interligados entre si. Estas são assim empresas que se encontram actualmente atentas a este movimento.

Desde da publicação do artigo "A Relational Model of Data for Large Shared Data Banks" [7] publicado por Edgar Codd, passando por sistemas como o System R [1] berço da linguagem SQL (*Structured Query Language*) ou o INGRES [22], várias soluções no armazenamento de dados surgiram até aos nossos tempos. Ainda que outras correntes existam como as bases de dados orientadas a objectos ou baseadas em XML, o paradigma relacional sempre foi dominante nesta área. Construído sobre o modelo relacional, baseado num sistema de tabelas e actualmente portador de funcionalidades como B-trees para indexação e transacções ACID, as bases de dados relacionais sempre provaram ser uma alternativa de fácil utilização mas fiável e eficiente para clientes empresariais. São hoje do conhecimento comum grandes nomes da área como a Oracle, SQL Server, DB2 e também as soluções de código aberto MySQL e PostgreSQL [21].

Mas se o modelo relacional é hoje a solução maioritariamente presente no mercado, não significa no entanto que ela represente a melhor abordagem para a resolução de muitos dos novos desafios surgidos neste novo contexto. Novos requisitos em escalabilidade, disponibilidade e redução de custos criaram no mercado a oportunidade para o nascimento de novas soluções em diferentes áreas. Foram assim criadas, sob o crescimento de novas correntes de desenvolvimento, dezenas de novos motores de base dados não relacionais, agrupados hoje sobre o termo NOSQL.

Mas ainda que actualmente popular, este termo é actualmente contestado devido a sua pouca clareza e má descrição sobre a base deste movimento. Ausente na maioria, presente em alguns, a linguagem SQL pouco tem a ver na verdade com o mesmo. Segundo o seu criador, este é um termo que significará sim "Not Only SQL", uma alternativa ao tradicional modelo relacional, à

[1] http://www.busmanagement.com/article/Managing-the-Data-Explosion/

visão onde apenas uma solução existe para todos os problemas.[2] Este é um termo abrangente que engloba diferentes tipos de base de dados focadas em características como modelos de dados dinâmicos, escalabilidade horizontal e modelos alternativos de coerência que oferecem novas soluções numa natural evolução do mercado.[3]

Nesta corrente várias grandes empresas da área Web criaram as suas próprias base de dados respondendo a requisitos de escalabilidade, performance e disponibilidade em situações limite, tanto para uso interno ou para venda como serviço. Entre os pioneiros encontram-se a Google, Amazon e Yahoo!, sendo que mais tarde outras empresas como o Facebook e LinkedIn também se juntaram ao movimento com soluções agora de código aberto. Entretanto outras opções surgiram no mundo do software aberto, na forma de base de dados orientadas a documentos ou mesmo baseadas em grafos.

Envolvido em grande discussão, este é um movimento ainda em amadurecimento, onde elementos como um conjunto de boas práticas, mecanismos de gestão e restantes ferramentas ainda se encontram em desenvolvimento. Mas independentemente do seu grau de maturação, estes novos modelos de gestão de dados constituem uma nova oportunidade para empresas de menores dimensões mas com problemas semelhantes apostarem em soluções que complementem os seus sistemas e os tornem mais ágeis, escaláveis e mais baratos. Representa no entanto um corte radical com o paradigma relacional como o conhecemos e por isso a sua adopção deverá ser uma decisão bem estudada. Elementos como os a seguir apresentados descrevem esta realidade e devem ser amplamente compreendidos aquando da decisão sobre qual o melhor motor de base de dados para um determinado problema.

Interfaces de utilização Em termos de API e linguagens associadas, existe em geral no movimento NOSQL um suporte para um conjunto variado de linguagens mas não para SQL. Para o desenvolvimento de certas aplicações, o uso de bibliotecas de acesso que partilham a mesma linguagem do código base, em conjunto com um bom modelo de dados pode no entanto ser benéfica. De facto, esta unificação das linguagens usadas pode levar a uma redução da impedância na escrita e mesmo à redução do tempo despendido neste processo, tornando assim a integração da base de dados mais natural. Para além disto o SQL como linguagem, sempre foi considerada por alguns como limitada e não natural para alguns tipos de dados, razão pela qual foi desenvolvida na Yahoo! a linguagem Pig Latin [18], orientada para processos de *map reduce*. Também a Microsoft criou integrada na plataforma .NET o LINQ [5], linguagem que pode ser usado para pesquisa de dados em diversas fontes tal como em SQL Server ou na plataforma Windows Azure, e com suporte também ela para *map reduce*.

No entanto não se pode descartar o SQL, com todos os seus anos de optimização, e ao qual existem associados toda uma miríade de ferramentas de desenvolvimento e *reporting*, standards e conhecimento humano. Ela é em si uma linguagem extremamente poderosa que permite ao utilizador a rápida escrita de operações para o modelo relacional, sendo que por implementação esta previne erros de baixo nível [24]. Por essa mesma razão existe hoje um movimento para a criação de um modelo standard para a interrogação e descrição destes novos produtos. Usualmente com sintaxes inspiradas no SQL, estas linguagens fornecem assim ao cliente uma interface mais intuitiva e de fácil uso. São exemplos o GQL (Google Query Language)[4] e o CQL (Cassandra

[2] http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html
[3] http://lwn.net/Articles/376626/
[4] http://code.google.com/appengine/docs/python/datastore/gqlreference.html

Query Language) implementado no Cassandra[5].

Mas este problema não reside somente na linguagem de interface usada, mas também nas operações pelo motor de dados suportadas que obrigam o programador a migrar muito do código de tratamento dos dados para o lado do cliente.

Modelo de dados O modelo de dados é também um ponto de divergência entre o antigo e o novo paradigma, não só em termos da estrutura dos esquemas mas também no correspondente processo de modelação. Na visão relacional, o esquema de dados é encarado como uma entidade relativamente estática presente na base de dados, onde permanece isolado da camada computacional sobre ele construída. Tais modelos permitem estabelecer garantias de integridade e assim manter os dados coerentes e organizados. O sucesso do paradigma relacional baseia-se na modelação problemas bem definidos, fornecendo sobre este uma poderosa linguagem de alto nível para o desenvolvimento de programas sobre os dados persistidos.[6]

No entanto, o paradigma de desenvolvimento de software mudou e de problemas definidos e com soluções estáveis passámos para problemas baseados em dados não estruturados que variam rapidamente ao longo do tempo. Com vista a um maior dinamismo e desenhados para larga escala, motores como os originados na Amazon e Google apresentam assim modelos alternativos baseados em pares chave valor, ou no particionamento de famílias de colunas. Estes modelos apresentam-se assim adaptados a uma maior dinâmica dos dados, suportando também a persistência e leitura de múltiplas versões temporais de um mesmo objecto. Estas características inviabilizam no entanto a expressão explícita de relações entre os dados em grande parte destas soluções. Outros modelos baseados em documentos podem também ser vistos em bases de dados como o CouchDB[7], ou na área dos grafos podemos encontrar soluções como o Neo4j[8], motores que se provaram eficientes nas suas áreas de uso [24].

Quanto à modelação de dados, estabelece-se também aí uma mudança no paradigma. No modelo tradicional, um esquema estudado e cuidadosamente modelado dava posteriormente origem a uma solução. Nestes novos sistemas, com vista a um aumento da performance e para evitar operações como os *Joins* que se provaram não escaláveis em muitos cenários [14], o desenho de esquemas centra-se na análise dos requisitos da aplicação em termo de operações, optimizando-se posteriormente o modelo sobre estes. Baseados em modelos dinâmicos, o esquema implementado pode depois ser alterado, sendo que o gestor da base de dados deverá contudo ser cuidadoso com o mesmo, ou submete-se ao risco de comprometer a usabilidade dos dados.

Coerência e garantias transaccionais Com o crescimento dos dados e a generalização do uso da replicação para atingir melhor performance e garantir a redundância em caso de falha torna-se cada vez mais difícil garantir coerência forte e garantias transaccionais. Apesar de representaram um modelo simples de programação estas garantias limitam a escalabilidade e disponibilidade dos sistemas de base de dados [14].

Em particular, o teorema CAP demonstra que num serviço distribuído apenas duas das seguintes características podem ser obtidas em simultâneo: Coerência, Disponibilidade e Tolerância de Partições [3, 13]. Quebrando com a convenção, sistemas como o Dynamo da Amazon ou o

[5] http://www.datastax.com/docs/0.8/api/cql_ref#cql-reference

[6] http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/

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

[8] http://neo4j.org/

Apache Cassandra fornecem aos seus clientes a opção de sacrificar a coerência dos dados entre réplicas aquando de uma partição na rede, permitindo assim aos sistemas manterem-se disponíveis. Quando a rede é reparada, as actualizações dos dados são então transmitidas entre os nós e a coerência volta a ser restabelecida. Assim surge o conceito de *eventual consistency*.[9] Este relaxamento da coerência, que é uma escolha no sistema e não um traço marcado deste, não pode no entanto ser ignorado pois apenas alguns serviços podem operar nestas condições [10, 25].

Mesmos os sistemas que exibem um modelo de coerência forte, como o BigTable, apenas suportam transacções ao nível da linha. Contudo, tem havido um interesse crescente em mecanismos externos para o suporte de transacções entre várias linhas, tal como o Percolator [19], o Megastore [11], o suporte para transacções em HBase com *Snapshot isolation* [27] e o sistema CloudTPS, que permite com o uso de uma camada suplementar de *middleware* o suporte de transacções em Amazon SimpleDB e Hbase [26].

1.1 Definição do problema

Muitas empresas olham actualmente para o movimento NOSQL em busca de vantagens para os seus sistemas, quer sejam estas financeiras ou tecnológicas. Partindo de soluções já estabelecidas, estas empresas enfrentam no entanto um vazio de conhecimento sobre qual o real impacto desta soluções, não só em termos de desempenho, mas também na forma como o sistema é afectado pelos paradigmas de desenvolvimento a elas associadas.

Existem na verdade alguns trabalhos direccionados para a avaliação do desempenho de tais sistemas como a plataforma extensível de nome Yahoo! Cloud Serving Benchmark (YCSB) que permite comparações simples de latência e taxas de transferência entre as soluções relacionais e não relacionais [8]. Estudos existem também sobre o relaxamento de coerência, os seus casos de uso e influências sobre os utilizadores de um sistema [10,25]. Mas este é apenas um dos pontos de divergência neste género de soluções, não existindo um estudo completo que avalie as implicações e os compromissos na passagem de uma implementação assente sobre o modelo relacional para uma solução com um diferente modelo de dados, API e modelo de coerência.

Partindo da adaptação de um sistema de teste de bases de dados amplamente conhecido e passando posteriormente por um caso de estudo realista tirado da industria avalia-se esta mudança, abordando temáticas como a adaptação do modelo e a complexidade do código. Este processo de migração torna-se também importante pois só com ele obteríamos a experiência necessária que posteriormente generalizamos sob a forma de uma ferramenta de *middleware*.

1.2 Contribuições

Esta dissertação aborda a migração de aplicações existentes para bases de dados não relacionais. No seu total pretende-se que esta forneça uma visão de quais as limitações e problemas que caracterizam uma mudança de uma aplicação já existente para um destes produtos, tal como uma possível solução. Em concreto são feitas as seguintes contribuições:

[9] Na verdade, muitos engenheiros concordam que a configuração *master slave*, presente por omissão em bases de dados como o MySQL garante apenas *eventual consistency*. - http://theryanking.com/entries/2010/04/29/potential-consistency/

Adaptação da plataforma de benchmarks TPC-W Após a pesquisa sobre várias destas soluções e a escolha do Cassandra como objecto de estudo no projecto, esta plataforma é aqui usada para testar esta base de dados. Com uma breve descrição da sua migração, a plataforma é depois utilizada com módulos para MySQL e Cassandra, permitindo estabelecer uma comparação entre estes dois paradigmas. Neste processo é avaliada também como a ausência de transacções afecta a coerência dos dados na base de dados, um dos desafios na mudança para uma solução não relacional do tipo Cassandra.

Adaptação de um sistema de chamadas Na fase seguinte apresenta-se o trabalho realizado em conjunto com a Portugal Telecom Inovação (PT Inovação). Com um problema real de escala que o seu sistema de validação de chamadas representa, este projecto tem como objectivo a avaliação de quais as vantagens e requisitos de uma migração do mesmo para uma solução como Cassandra. Com uma descrição da implementação tanto no paradigma relacional como não relacional, demonstramos as diferenças em termos de modelo, acesso à base de dados e garantias dadas por ambos.

Componente de mapeamento de objectos Numa fase final desenvolve-se um componente de mapeamento de objectos (OM na sigla anglófona, onde associada ao modelo relacional se designa ORM) para Cassandra. Esta solução pretende fornecer uma abstracção da camada de dados subjacente e assim criar uma plataforma para um mais fácil desenvolvimento de novas aplicações de forma escalável ou para a migração de sistemas já existentes como o apresentado. Na sua criação, avaliamos como pode o mapeamento de entidades e relações ser feito em Cassandra.

Na totalidade do trabalho são assim abordados os temas relacionados com estruturas de dados e interfaces de utilização numa comparação entre os paradigmas relacional e não relacional, fornecendo no final uma possível solução para reduzir o fosso entre eles.

1.3 Estrutura da dissertação

Esta dissertação está organizada da seguinte forma. No Capítulo 2, apresenta-se um estudo do estado da arte sobre a área das base de dados não relacionais, com as suas características e casos de uso. Inclui-se também uma pesquisa sobre mapeamento de objectos em geral e soluções já existentes no mercado para bases de dados não relacionais. No Capítulo 3 apresentam-se os benchmarks realizados com o TPC-W sobre Cassandra, descrevendo-se a plataforma usada, a sua adaptação e os resultados recolhidos. No Capítulo 4 apresenta-se um caso de estudo realista e a migração do mesmo para Cassandra com uma avaliação do processo. De seguida, no Capítulo 5 apresenta-se a solução de OM desenvolvida, os desafios encontrados e as soluções escolhidas. Finalmente no Capítulo 6 temos as conclusões tiradas e trabalho futuro.

Capítulo 2

Estado da Arte

Nesta secção do documento temos numa primeira fase a descrição das várias bases de dados investigadas no contexto desta dissertação. Começando por uma pequena introdução que pretende fazer luz sobre a área, apresentam-se vários novos motores de base de dados surgidos nos últimos anos, sendo alguns deles avaliados resumidamente em termos de estrutura de dados e interfaces de utilização. No seguimento do trabalho realizado existe também uma introdução sobre interfaces para o mapeamento de objectos. Esta abordará os conceitos gerais e algumas das conhecidas soluções na área, tanto para o lado relacional como para o movimento NOSQL.

2.1 NOSQL

Os anos recentes foram marcados pelo aparecimento de dezenas de novas soluções na área das bases de dados não relacionais. Este movimento, ainda hoje em marcha, dividiu a comunidade em discussões sobre o potencial valor destas novas soluções, ou sobre o futuro da base de dados relacionais. Mas a verdade é que as base de dados não relacionais já existem há tantos ou mais anos que as suas parceiras relacionais. Mesmo o nome já foi usado por Carlo Strozzi para em 1998 dar nome à sua base de dados relacional sem interface SQL.[1]

Do ponto de vista histórico e ainda antes do desenho do modelo relacional, sistemas como o MultiValue e o IBM IMS foram criados nos anos 60, em conjunto com o M[umps], linguagem que incorpora um sistema de armazenamento hierárquico com árvores B+. Nas duas décadas seguintes encontramos o DBM desenvolvido pela AT&T tal como sistemas que ainda hoje são mantidos como o BerkeleyDB e o Lotus Notes. Nos anos seguintes e até ao final do milénio, outros nomes de base de dados não relacionais surgiram como o GDBM e o Menésia desenvolvido na Ericsson. LDAP é outro dos nomes sonantes da área, onde outras categorias também existem como as bases de dados orientadas a objectos que tem as suas raízes nos anos 70 [24].[2]

Mas o ano que marca o nascimento do movimento NOSQL é o ano 2000 com o aparecimento do Neo4j, uma base de dados orientada a grafos. A esta muitos outros nomes se seguiram em diferentes modelos do tipo tuplos chave-valor, base de dados orientadas a documentos ou por famílias de colunas. Incorporadas no movimento, surgiram também bases de dados baseadas em XML respondendo ao crescimento no sector Web.

[1] http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page
[2] http://blog.knuthaugen.no/2010/03/a-brief-history-of-nosql.html

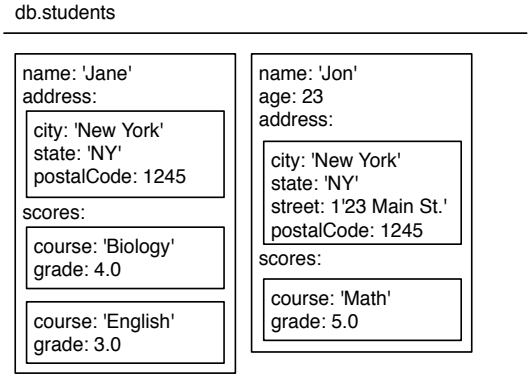


Figura 2.1: Modelo de dados por documentos

Na miríade de soluções abordadas, uma linha pode no entanto ser estabelecida entre os diferentes produtos estudados de acordo com os seus modelos de dados e escalabilidade. Ao contrário das soluções desenvolvidas pelos grandes nomes da área Web como o Cassandra ou Voldemort, construídos de raiz para escalar horizontalmente, existem soluções mais inclinadas a nichos específicos do mercado como as bases dados orientadas a grafos. Já para um uso mais comum, quando um esquema dinâmico é uma mais-valia, temos também as bases de dados orientadas a documentos.

São assim analisados alguns dos nomes nas diferentes áreas, apresentando a forma como estes motores abordam o tema da escalabilidade. Estes são divididos em três secções, sendo a última referente aos sistemas que foram construídos de raiz para o armazenamento de grandes volumes de dados.

2.1.1 Base de dados orientadas a documentos

As bases de dados orientadas a documentos (*Document Stores*) baseiam-se em modelos de dados semi-estruturados onde a informação é codificada em elementos a que chamamos documentos. Estes documentos são constituídos por um número virtualmente ilimitado de campos onde a informação é guardada[3], permitindo também o aninhamento de outros documentos, sendo um exemplo visível na Figura 2.1. Este é um modelo interessante pois a informação armazenada está até certo ponto não estruturada, permitindo lidar com dados de natureza dinâmica. Embora similares às bases de dados por objectos, estas evitam as relações entre os elementos armazenados.

Entre os nomes actualmente presentes na área encontramos o MongoDB e o Apache CouchDB. O MongoDB é actualmente usado em produção para serviços como a gestão de informação sobre os utilizadores em sites como SourceForge, Foursquare, The New York Times, Electronic Arts entre muitos outros nomes conhecidos da área Web. De certo modo criticada pela sua decisão de por omissão não confirmar o sucesso das escritas, focando as garantias de durabilidade dos dados na sua replicação, esta é ainda assim amplamente usada.

Como abordagem ao problema da escalabilidade, esta solução possui em desenvolvimento uma plataforma automática de particionamento. Na verdade como neste tipo de base de dados, os documentos têm normalmente outros aninhados dentro de si mesmos e não relações, a escalabili-

[3]Os documentos são geralmente limitados em tamanho.

dade horizontal da base de dados é assim natural. Para eliminar pontos de falha no sistema, cada uma destas partições será constituída por uma combinação de um mestre e várias réplicas ou por uma estrutura apelidada de *Replica Set* onde com base na nesta mesma arquitectura, os processos de delegação em caso de falha e recuperação de nós são automáticos. Neste mesmo propósito existem também operações que asseguram que N réplicas para além do mestre contêm os dados, garantido a durabilidade dos mesmos, em caso de falha. Para a implementação deste sistema o MongoDB baseia-se em três tipos distintos de elementos: os *shards* que contêm os dados, os servidores de configuração que contêm informação sobre a localização dos dados, e os *mongos* que funcionam como uma interface para o utilizador fornecendo a visão de uma base de dados única.

O CouchDB tem como cliente de maior dimensão a BBC, sendo também usado em outros sites e aplicações do Facebook. Este produto exibe uma arquitectura baseada no controlo de concorrência multi-versão (normalmente designado por MVCC de multiversion concurrency control) e um sistema de replicação bidireccional adaptada ao funcionamento em *offline* da uma das partes. Esta característica faz como esta seja apontada como uma solução para armazenamento dos dados em dispositivos móveis. Neste contexto o controlo de versões na base de dados resolverá de forma automática parte dos conflitos que poderão surgir, sendo que a escolha entre documentos com versões concorrentes terá de ser feita pela aplicação que faz uso da mesma. Em termos de escala, não sendo esta uma solução direccionada para a área do suporte de grandes conjuntos de dados através do particionamento horizontal dos dados, tal é possível através do projecto CouchDB-lounge[4].

Outras soluções existem, como as base de dados XML que são um derivado deste modelo, área onde podemos encontrar várias soluções maduras como a proposta pela MarkLogic[5].

2.1.2 Bases de dados orientadas a grafos

Tem-se assistido de forma discreta nos últimos anos a um interesse crescente em teoria de grafos: sejam pelos grafos sociais derivados da crescente importância das redes sociais, ou de topologia Web como os manipulados nos gigantes da pesquisa de informação. Como os investigadores da Google apontam aquando da apresentação da sua estrutura para extracção de informação de grafos de nome Pregel [6], os grafos estão ao nosso redor e a sua manipulação é hoje um desafio em curso.

Com esta tendência surgiram assim vários motores para manipular este tipo de dados, criando API especializadas e optimizando os motores de interrogação para operações como a realização de transversais. Com um modelo que consiste em nós, relações e propriedades, do qual vemos um exemplo na Figura 2.2[7], é possível de uma forma mais fácil e eficiente lidar com este tipo de estruturas. Existem actualmente no mercado várias soluções como o Neo4j, InfiniteGraph ou o, recentemente desenvolvido por engenheiros da Twitter, flockDB.

Em termos de escalabilidade, sendo este um modelo de dados focado nas relações entre os nós presentes na base de dados, esta é um ponto particularmente difícil, pois uma solução baseada na partição dos dados irá inevitavelmente levar à separação de vértices ligados entre máquinas. Existe no entanto, na maioria das soluções mais maduras, um esforço da base de dados para minimizar este efeito aquando do particionamento horizontal, desenhando para isso um linha de separação nos dados que reduza o número de arestas partilhadas pelas máquinas.

[4] http://tilgovi.github.com/couchdb-lounge/
[5] http://www.marklogic.com/
[6] http://googleresearch.blogspot.com/2009/06/large-scale-graph-computing-at-google.html
[7] Imagem retirada de http://blog.neo4j.org/

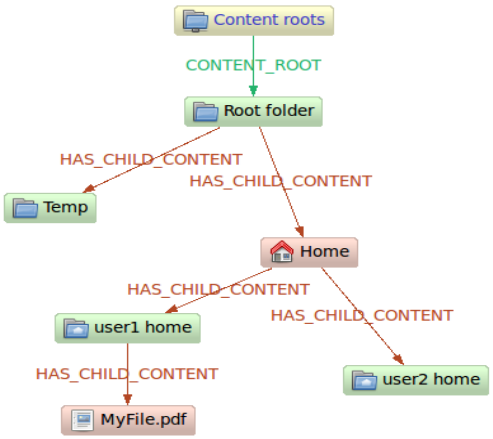


Figura 2.2: Modelo de dados por grafos

2.1.3 Big Data

Big data é um termo usado com alguma frequência nos nossos dias para descrever o extraordinário crescimento do volume de dados associados a algumas áreas de negócio e os problemas a ele associado. Estes problemas, tanto na área empresarial como científica, referem-se não só à manipulação destes mesmos dados para extracção de informação mas também ao problema base de os como armazenar de forma sustentável. Este é um dos principais problemas que as bases de dados a seguir apresentadas se propõem de alguma forma mitigar.

Para o típico cenário de desenvolvimento de uma aplicação Web, a escolha de uma base de dados relacional como o MySQL ou PostgreSQL é pelas suas características uma escolha segura. As vantagens inerentes ao modelo relacional, tal como o conhecimento e ferramentas associados ao mesmo são factores que tornam esta a melhor escolha para o utilizador comum. No entanto, em situações de aumento exponencial dos dados existe uma necessidade de reavaliar qual a melhor opção para garantir a escalabilidade da estrutura de suporte de dados de forma a garantir disponibilidade e latência aceitável. Como a hipótese de escalar na vertical é aqui uma escolha difícil, pois evolve normalmente custos avultados e dependência em relação ao fornecedor, muitas empresas optam assim pela abordagem horizontal.

Optar por uma solução de escalabilidade horizontal numa base de dados standard com o MySQL é no entanto na situação actual um desafio. Adoptando um esquema de replicação mestre-réplica existe sempre o risco inerente de o mestre poder falhar e perder dados num pico tráfego, existindo depois os atrasos correspondentes à eleição de um novo. Esta é também uma alternativa que permite apenas escalar em termos de leitura pois as escritas recaem todas no mestre. Passando depois por esquemas de replicação mestre-mestre de difícil configuração, a solução acaba por recair num mecanismo de particionamento. O particionamento consiste, nesta área, na divisão dos dados existentes entre várias máquinas, permitindo uma divisão da carga em termo do seu tamanho e das instruções recebidas. No entanto esta não é uma solução natural no mundo relacional, pois requer a criação cuidadosa de uma função de distribuição e o redesenhar do modelo para evitar custosos *Joins* entre máquinas. Para além da difícil configuração e da necessidade de constante monitorização do cluster, esta solução acaba em muitos casos por retirar muitas das vantagens associadas as bases de dados relacionais. Com a particionamento, perdem-se as cha-

ves estrangeiras, sendo que as transacções ACID e *Joins* acabam também por ser proibitivas em termos de desempenho quando envolvendo dois ou mais nós. Deste modo resta no sistema pouco mais que a capacidade de manipulação dos dados através da linguagem SQL. Outras opções como os produtos fornecidos pela Oracle, para além custos de produção exorbitantes podem também na opinião de alguns engenheiros de sistemas fornecer menor fiabilidade que uma base de dados distribuída sobre hardware de uso genérico.[8] [9]

Deste modo conclui-se que escalar uma base de dados relacional não é barato ou fácil, uma vez que as soluções verticais são sinónimo de pouca fiabilidade e as horizontais não são naturais neste modelo. Neste contexto surgiram assim novas soluções no mercado da larga escala, com novas arquitecturas e modelos de coerência, possibilitando uma melhor abordagem das situações de falha e altas latências associadas às comunicações entre centros de dados.

Em 2004 a Google inicia um novo projecto de nome BigTable [6], desenhado para escalar ao nível do petabyte. Esta base de dados é baseada num vector associativo multidimensional esparso e ordenado que assim vai de encontro aos requisitos aplicacionais e natureza dinâmica dos dados associados à Google. Em 2007 a Amazon apresenta o seu sistema de dados, Dynamo, usado em várias partes da sua estrutura de negócio [9]. Baseado na agregação de diferentes tópicos de investigação na área dos sistemas distribuídos, esta é uma solução conhecida por permitir ao utilizador o relaxamento da coerência dos dados de forma a garantir a disponibilidade do sistema em caso de falha. Estes foram sistemas pioneiros nas suas áreas e em conjunto com outras soluções inspiradas por eles, representam hoje um conjunto de bases de dados que pretendem de forma mais sustentável garantir a escalabilidade dos dados, oferecendo modelos alternativos de dados e coerência. A alteração na natureza dos dados levou também a uma mudança no desenho dos modelos, que vão hoje mais de encontro às necessidades aplicacionais. Com preocupações de eficiência a desnormalização da informação é hoje uma realidade comum tal como a utilização da ordem natural dos dados e o uso extensivo de índices.

2.2 Análise de bases de dados em grande escala

No contexto do trabalho desenvolvido, apresenta-se agora um estado da arte sobre os sistemas de armazenamento dirigidos a grandes volumes de dados. Com este objectivo em mente, não se inclui apenas uma simples descrição dos mesmos, mas também uma visão do processo de desenvolvimento à luz dos diferentes modelos de dados, API, modelos de coerência, etc.

Tendo como origem o trabalho desenvolvido tanto pela Google como pela Amazon, as diversas soluções existentes nesta área são hoje divididas em duas grandes famílias consoante o seu modelo de dados: os projectos com um modelo chave-valor inspirado no Dynamo e o conjuntos de produtos que exibem uma estrutura por famílias de colunas à semelhança do BigTable. Sobre esta divisão, que se estabelece ao nível do modelo de dados, organizam-se assim as diversas soluções representativas da sua área, cada uma delas caracterizada por diferentes opções ao nível da API ou modelo de coerência.

[8] http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king

[9] http://www.25hoursaday.com/weblog/2010/03/10/BuildingScalableDatabases AreRelationalDatabasesCompatibleWithLargeScaleWebsites.aspx

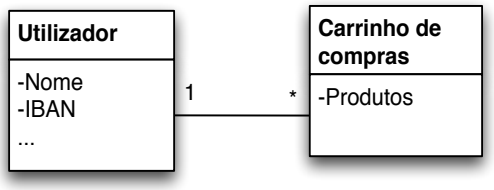


Figura 2.3: Modelo exemplo

2.2.1 Bases de dados por chave-valor

Baseadas no modelo simples de chave-valor (Key-Value), existem actualmente no mercado vários produtos em diferentes estados de maturação. Com várias soluções que têm como base este modelo, as bases de dados aqui escolhidas para representação desta área são o Riak e o Voldemort. De facto tanto o Riak com o seu modelo híbrido e suporte fornecido pela Basho ou o Voldemort desenvolvido pela LinkedIn são os nomes mais conhecidos. Para cada uma delas é apresentado conforme o caso de estudo abaixo apresentado, quais as suas características base e oportunidades fornecidas em termos de modelo e de implementação [10] .

Outras soluções existem que não foram mencionadas ou exploradas. Em parte porque as suas arquitecturas e modelos de replicação não são escaláveis, soluções maduras e interessantes como o Redis (mencionado pelo seu modelo de dados) ou Berkeley DB não são assim analisadas. Destacam-se ainda soluções mais académicas como o projecto Scalaris com o seu modelo de coerência forte e transacções ACID. O sistema S3 é também não avaliado, pois para além do seu modelo base e API base, não existe muita informação sobre o sistema.

Um caso de estudo

Com o intuito de melhor demonstrar o funcionamento base das soluções propostas, tal como algumas das características que as distinguem, apresenta-se um problema base ao qual esta abordagem se adequa. Sendo geralmente dirigidas a problemas com modelos de dados simples onde os objectos não estejam directamente relacionados entre si, optou-se pela escolha do modelo de carrinhos de compras, tal como foi apresentado no artigo de apresentação do Dynamo [9].

Este é um modelo constituído por duas entidades: o carrinho de compras e o cliente. Cada um dos carrinhos é constituído por diversos produtos e respectivas quantidades adquiridas estando este associado ao cliente que efectuou as operações de compra. A entidade cliente é onde se encontra a informação sobre os utilizadores do sistema. A chave do carrinho será incremental e supõe-se que cada cliente saberá a chave do seu, não se especificando como esta relação é mantida no modelo. O modelo pode ser visto na Figura 2.3.

Em termo de operações, as operações base de escrita e leitura são as únicas que se assume existirem em todas as soluções, sendo depois exploradas outras oportunidades que eventualmente sejam fornecidas pelas plataformas estudadas.

[10]Note-se que o Voldemort é aqui usado em parte como representante do Dynamo.

Modelo de dados

O modelo comum a estas bases de dados, minimalista por desenho, é caracterizado por simples relações entre uma determinada chave e um objecto que será em grande parte opaco ao sistema. Existem depois variações em termos de ordenação dos objectos, nos meta-dados neles incluídos, opções de armazenamento, etc.

Apresentam-se agora alguns dos produtos estudados e soluções com base no modelo estabelecido.

Soluções com valores opacos: Voldemort Em soluções como o Voldemort, o modelo base desenhado é unicamente constituído por tuplos chave-valor. Como tal, os valores inseridos no sistema são tratados à partida como caixas negras, podendo no entanto existir o suporte para vários protocolos de serialização como implementado pelo Voldemort, permitindo armazenar num formato acessível, estruturas ricas em conteúdo.

Modelando o exemplo, as entidades cliente e carrinho serão totalmente opacas ao sistema, sendo da total responsabilidade do cliente a distinção entre os vários tipos de valores. Do mesmo modo, não existe qualquer tipo de relação explícita no modelo entre as diferentes entidades, podendo depois o cliente contar uma lista dos carrinhos que lhe pertencem ou vice-versa consoante a implementação escolhida e gerida pelo programador. Tal é visível na Figura 2.4.

Soluções com valores complexos: Riak e Redis Indo para além do modelo base, estas duas soluções enriquecem-no com novos tipos de dados, meta dados ou ligações para outros elementos.

Explorando o Redis, esta é uma base de dados em memória construída para funcionar como um dicionário de acesso concorrente, actualmente baseada num só servidor. No seu interior, no entanto, não são admitidos apenas valores arbitrários, mas também estruturas como listas, conjuntos ordenados e vectores de associação que em conjunto com uma interface rica, melhoram assim a experiência do utilizador. De facto o Redis não é uma solução escalável neste ponto, estando no entanto em desenvolvimento uma versão distribuída desta base de dados. Como se trata de uma interessante solução baseado em chave valor e actualmente usada por diversas empresas, esta é aqui referida em termos de modelo.

No caso do Riak, embora este seja na sua base um clone do Dynamo, o seu modelo evoluiu e possui hoje a opção do tratamento dos seus valores como sendo documentos ainda que na base de dados estes permaneçam opacos. Assim muitas vezes tratada como uma base de dados orientada a documentos, esta não permite no entanto alterações de valores dos mesmos. Ainda assim, esta base de dado apresenta outras vantagens como o facto de que cada valor poder ter associado meta dados que acerca deste fornecem informação, podendo também conter referências para outros registos (*links*). Com esta funcionalidade podemos então ter, por exemplo, entidades cliente que contém uma referência para um carrinho de compras ou vice-versa como visível na Figura 2.4.

Características e implementação

Voldemort Como a maioria dos sistemas que apresentam este modelo, o Voldemort apresenta uma API simples constituída por métodos simples de armazenamento e leitura. Estas operações são garantidamente atómicas, mas para além disso, não existem quais quer garantias transaccio-

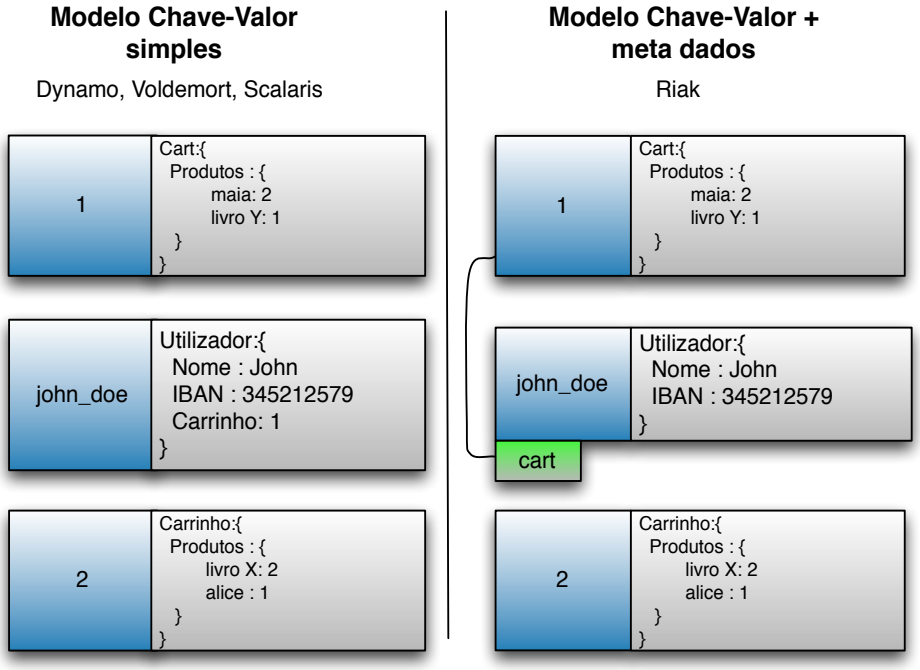


Figura 2.4: Modelo numa solução por chave-valor

nais ou mecanismos semelhantes. Ao nível de coerência, esta solução funciona num mecanismo de Quóruns, onde é definível o número de réplicas onde cada elemento dos dados estará. Com base neste parâmetro cliente estabelecerá então, aquando do arranque da base de dados, quantos desses nós serão necessários para validar cada escrita e quantas nós servirão de fontes para cada leitura. Deste modo o cliente tem controlo sobre os compromissos entre a coerência pretendida e a disponibilidade que assegurará em caso de falha. Originalmente definido na Amazon para acomodar os diferentes casos de uso dentro da organização, este sistema permite também estabelecer compromissos entre sistemas de mais direccionados a escritas ou leituras, estabelecendo por exemplo que apenas um só nó é necessário para leitura, mas todas as réplicas terão de confirmar aquando de uma escrita.

Partindo deste sistema de Quórum, a noção de *eventual consistency* nasce do cenário onde operações concorrentes escrevem diferentes valores dos mesmos dados em duas réplicas diferentes, criando assim uma situação de incoerência temporária. No caso destes sistemas, os conflitos são depois detectados com a ajuda de relógios vectoriais. Quando depois se efectua uma leitura de dois ou mais nós que contem dados incoerentes, e se se verificar um conflito nas versões dos relógios então o utilizador terá de escolher qual a versão que considera correcta, caso contrário o sistema escolherá a versão mais recente. Este mecanismo é apelidado de reparação na leitura ou *read repair*. Aplicando ao exemplo proposto, obtemos o caso exposto no artigo do Dynamo [9] onde para a resolução de conflitos é automática não necessitando da intervenção do utilizador final. Para tal, quando duas versões concorrentes de um carrinho existem, a solução proposta é fazer a união das duas resultando numa versão final que contém a soma de todos os produtos adquiridos, podendo conter elementos que já tinham sido removidos pelo cliente. Assegurar a melhor resposta no caso de falha é vital a muitos negócios pois pode significar a fidelidade de um cliente, que numa situação de não disponibilidade do sistema poderia nunca mais retornar.

Outra característica marcante destas soluções é o facto de todos os nós serem iguais entre si, não existindo assim pontos de falha. Quanto à distribuição dos dados entre estes, esta é assegurada com recurso a técnicas de *consistent hashing*. Desta forma e utilizando um boa função de dispersão assegura-se que os dados irão ser distribuídos por todos os nós equitativamente, facilitando também a redistribuição automática aquando da entrada de um nó no *cluster*. Isto tem também influência na API fornecida, uma vez que não existindo ordem natural dos dados, impossibilita-se assim a implementação de uma operação de pesquisa e leitura eficiente de informação de forma sequencial.

Para a implementação do exemplo, existem em primeiro lugar algumas definições a configurar. O gestor da base e dados, terá em primeiro lugar de escolher qual a ferramenta de persistência sobre a qual a base de dados irá operar, sendo que soluções como o Voldemort permitem a escolha de opções como MySQL, BDB ou simplesmente a persistência em memória. Outro elemento configurável da base de dados é também o modo como a serialização da informação será feita: String, serialização Java ou JSON são algumas das opções. Tais formatos textuais como o Json permitem por exemplo ao gestor da base de dados a consulta de informação de forma simples.

De seguida é necessário observar os dados e estatísticas de uso se existentes, para decidir qual as configurações para os factores de replicação, leitura e escrita. Supondo que os carrinhos de compras terão um tráfego não muito díspar em termos de leitura e escrita, e assumindo que a coerência total é algo a preservar em todos os momentos, uma possível escolha passaria pelo uso de 3 réplicas para cada elemento dos dados, usando depois 2 desses nós para escrita e leitura (quórum). De tal forma, assegura-se que cada para cada escrita executada, aquando da leitura no mínimo um dos nós irá conter a versão mais recente dos dado, tal como se permite deste modo a falha de uma das máquinas sem risco associado.

Quanto à interface, na sua base o Voldemort suporta protocolos HTTP ou TCP/IP simples existindo por cima destes implementadas bibliotecas em diversas linguagens. Já em termos de API, são apenas fornecidas as operações base restando ao programador manter de alguma forma associada ao cliente qual a chave do seu carrinho de compras actual, informação esta que permite assim a leitura e escrita do mesmo.

Riak Também assumidamente inspirado no Dynamo, o Riak é uma solução de código aberto que oferece modelos de dados mais ricos e um modelo de coerência ligeiramente diferente das outras soluções do género. Como foi referido no modelo de dados, esta solução oferece aos seus utilizadores a hipótese de utilizar um modelo mais rico através da utilização de documentos em cada um dos valores (de nome *bucket*). Cada um destes valores poderá também conter meta-dados que indicam por exemplo o tipo do valor, tal como *links* para outras entidades.

A nível de coerência embora esta seja também baseada em Quóruns e factores de replicação, no Riak a definição sobre qual o número de nós a usar para cada escrita e leitura, não é definida no arranque do sistema, mas sim por operação. Deste modo o cliente tem mais controlo sobre os compromissos entre coerência e disponibilidade, sendo que o próprio factor de replicação pode não ser universal e ser sim definido por tuplo. No entanto, como o Riak funciona baseado na noção de nós virtuais que são depois espalhados pelos nós reais do *cluster*, existe um risco inerente a este mecanismo. De facto quando os dados são replicados, não existem garantias que o sistema de distribuição de nós virtuais consiga espalhar estas mesmas réplicas por mais que uma máquina, embora esses sejam casos excepcionais.

Para além disto, é introduzida no Riak a opção de escolha sobre qual o mecanismo de resolução a usar. Deste modo o utilizador da base de dados escolhe entre a resolução de conflitos através de relógios vectoriais à imagem do Dynamo ou baseada num mecanismo automático do lado servidor sobre uma estratégia onde o valor mais recente ganha. A estratégia de resolução de conflitos pode também ela ser definida por tuplo.

Em termos de implementação o Riak oferece uma API mais rica tal como um novo conjunto de funcionalidades. Numa primeira fase e à semelhança do Dynamo/Voldemort o cliente terá também de escolher qual o motor de dados usado para garantir a persistência dos dados, podendo optar por soluções como o Bitcask ou InnoDB. Já em termos de modelo de coerência dos dados, as definições sobre qual o número de réplicas será talvez o único parâmetro a definir de raiz, mas também ele poderá ser alterado e definido para cada tuplo.

Em termos de interface todos os pedidos são efectuados com base no protocolo HTTP por servidores Web *RESTful*, existindo clientes em diversas linguagens de programação. Com as operações base de *put* e *get*, obtemos uma funcionalidade semelhante à apresentada na solução anterior. O Riak permite depois com o uso extensivo de headers HTTP, a configuração das propriedades dos *bucket*, a inclusão de meta-dados, e a utilização de links. Esta última funcionalidade permite estabelecer relações entre as entidades e o utilizador pode assim realizar instruções onde se retornam todos os carrinhos de compras de um determinado cliente ou todas as pessoas a este associadas, por exemplo.

Mas uma das características que distingue o Riak de outras soluções é a sua implementação do paradigma *Map Reduce*, permitindo assim criar instruções que irão ser distribuídos pelos diversos nós aproveitando o paralelismo impresso nos dados. Tal ferramenta permite a análise de informação sobre os dados contidos no *cluster* permitindo no âmbito do exemplo apresentado, fazer questões como: "Qual o produto mais vendido?"ou "Qual a média de produtos num carrinho de compras".

2.2.2 Bases de dados por famílias de colunas

Também elas direccionadas para a escalabilidade dos dados, apresenta-se agora última categoria do movimento NOSQL: as soluções baseadas em famílias de colunas, ou no seu nome original *Wide Column stores*.

Categoria que tem as suas bases no projecto BigTable [6] desenvolvido na Google, esta é constituído por versões deste sistema que outras empresas adaptaram. Em muito semelhantes à solução apresentadas pela Google, estes produtos são hoje o Apache Hbase utilizado pela Adobe e o Hypertable patrocinado pelo Baidu o maior motor de pesquisa chinês. Como último da categoria, temos o Apache Cassandra [16]. Desenvolvido inicialmente no Facebook para resolver um problema de pesquisa na caixa *Inbox* presente na sua página, esta empresa abriu o seu código fonte para a comunidade em 2008. Esta solução reúne em si a arquitectura do Dynamo e o modelo de dados do BigTable, sendo actualmente usado por empresas com o Twitter, Netflix e Reddit.

Com o intuito de avaliar estas mesmas soluções, é apresentado a seguir um modelo de dados através através do qual depois se compara as diferenças entre as várias abordagens (sendo o BigTable e clones agrupados numa só secção pela sua similitude).

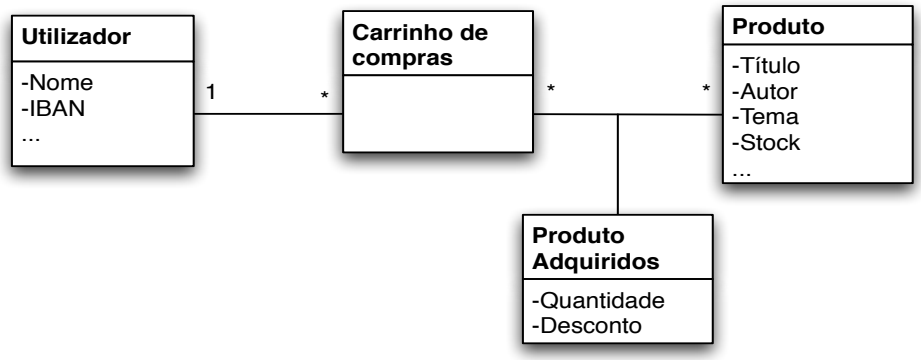


Figura 2.5: Modelo de dados exemplo

Um caso de estudo

Como modelo e caso de estudo exemplo optou-se nesta secção por uma extensão ao modelo apresentado na secção anterior. Deste modo existem as entidades cliente e carrinho de compras, as quais agora adicionamos a entidade produto que representa os artigos para venda no sistema. Estes produtos, aqui materializados na forma de livros, contêm em si informação sobre os mesmos, tal como o título, autor, comentários, etc. Uma representação do modelo é visível na Figura 2.5.

Em relação às operações realizadas sobre o sistema, para além das operações base de escrita e de leitura, novamente nada mais se assume.

Modelo de dados

O modelo original, tal como as variações presentes nas diversas bases de dados deste tipo, consistem num vector associativo multidimensional esparso e ordenado. Este é um modelo que pela sua estrutura e nomenclatura requer dos modeladores um processo prévio de aprendizagem e uma mudança radical de mentalidade quando vindo de um passado relacional.

Big Table, HyperTable e HBase Os modelos base, tal como os implementados no Big Table e restantes derivações são constituídos por um vector associativo multidimensional onde para cada chave existente será mapeado um conjunto de colunas que se encontram agrupadas em famílias (*Column Family*). Cada uma destas colunas irá conter um nome (totalmente independente de qualquer esquema) que a identifica na linha, o respectivo valor e uma marca temporal (timestamp na versão anglófona). Note-se também que diferentes versões de uma mesma coluna podem coexistir na base de dados ordenadas pela respectiva marca temporal.

Assim na inserção de uma coluna, o cliente tem de especificar qual o nome da coluna a inserir, o seu valor e marca temporal, fornecendo também a chave que identifica a linha onde ocorre a inserção. O outro parâmetro que o cliente terá também de indicar é o nome da família de colunas, estruturas que agrupam, para o conjunto total dos registos presentes na base de dados, as colunas que se consideram estarem entre si relacionadas. Estas estruturas ocupam no sistema um papel em muitos pontos semelhante às tabelas relacionais. No seu total, cada linha poderá armazenar um número virtualmente ilimitado de colunas sendo que nenhuma restrição existe em termos de modelo sobre quais os nomes das colunas ou o seu número. Quanto ao número total de linhas, que

se encontram no sistema ordenadas lexicograficamente e onde todas as operações são atómicas, é dado o nome de Tabela.

Sobre o modelo existem depois opções como os grupos de localidade. Esta que codifica as preferências de localização dos dados, de tal forma que dados normalmente acedidos em conjunto ficam fisicamente mais próximos, aumentando a performance do sistema[11].

Desenhando o modelo para a implementação do exemplo, o responsável primeiro terá de avaliar todas as oportunidades fornecidas por características como os grupos de localização (se existentes) ou a ordem lexicográfica das chaves, etc. Neste caso em específico, opta-se pela criação de três diferentes famílias de colunas correspondentes às entidades cliente, carrinho de compras e produto. Como optimizações, temos por exemplo o uso da ordem das chaves com a inclusão do nome ou tema do livro nesta tornando assim mais eficiente o uso de operações de leitura sequencial de informação com base nestes campos.

Para cada linha e família, existirão assim colunas que codificam a informação sobre cada uma das entidades, como o título no caso dos livros ou nome no caso dos clientes. Já no caso do carrinho de compras, assumindo que o nível de produtos no seu interior nunca será elevado, podemos criara uma coluna com informação para cada item aquando da sua adição ao carrinho. Isto implica no entanto que os dados relativos ao nome, quantidade e potenciais descontos sejam mantidos numa única coluna. Este facto impossibilita alterações de um só parâmetro do produto, obrigando à rescrita completa da coluna.

O sistema de versões, pelo qual as ultimas N versões de cada coluna são armazenadas (onde N é definido pelo utilizador), pode também ser usado para guardar alterações ao perfil ou informação antiga dos produtos existentes. Esta funcionalidade pode se revelar útil na visualização da evolução dos dados (a classificação dos livros por exemplo) ou recuperação de versões antigas dos mesmos.

Podemos agora observar o aspecto do modelo na Figura 2.6.

Cassandra Na sua essência o modelo do Cassandra é em muitos pontos igual ao modelo acima descrito, possuindo no entanto duas diferenças fulcrais: a perda da dimensão temporal nas colunas e introdução de super famílias de colunas. As super famílias de colunas são famílias onde associada a cada chave não se encontra uma, mas sim várias listas de colunas. Cada um destes elementos será então mapeado por uma chave interna, numa estrutura denominada super coluna.

Quanto às chaves que mapeia todas as linhas da base de dados, que se passa a chamar *keyspace*, estas podem ser ordenadas lexicograficamente, pelos seus bytes[12] ou distribuídas aleatoriamente. Mas para além da ordenação das chaves existe também para cada família de colunas ao nível da linha, e no caso das super famílias para as suas super colunas, um campo de configuração que estabelece como as colunas irão ser ordenadas. Esta opção permite, por exemplo, a leitura das últimas dez colunas de uma linha usando como nome destas uma marca temporal e escolhendo como elemento de ordenação o *TimeUUIDType*.

No desenho do modelo, o responsável terá de avaliar a natureza dos dados para fazer uma escolha ponderada acerca do esquema de partição de dados e também sobre a natureza das famílias de colunas a usar. De facto as super colunas são úteis em situações de relações de um para muitos;

[11] Não implementado no HBase

[12] Apenas a partir da versão 0.7

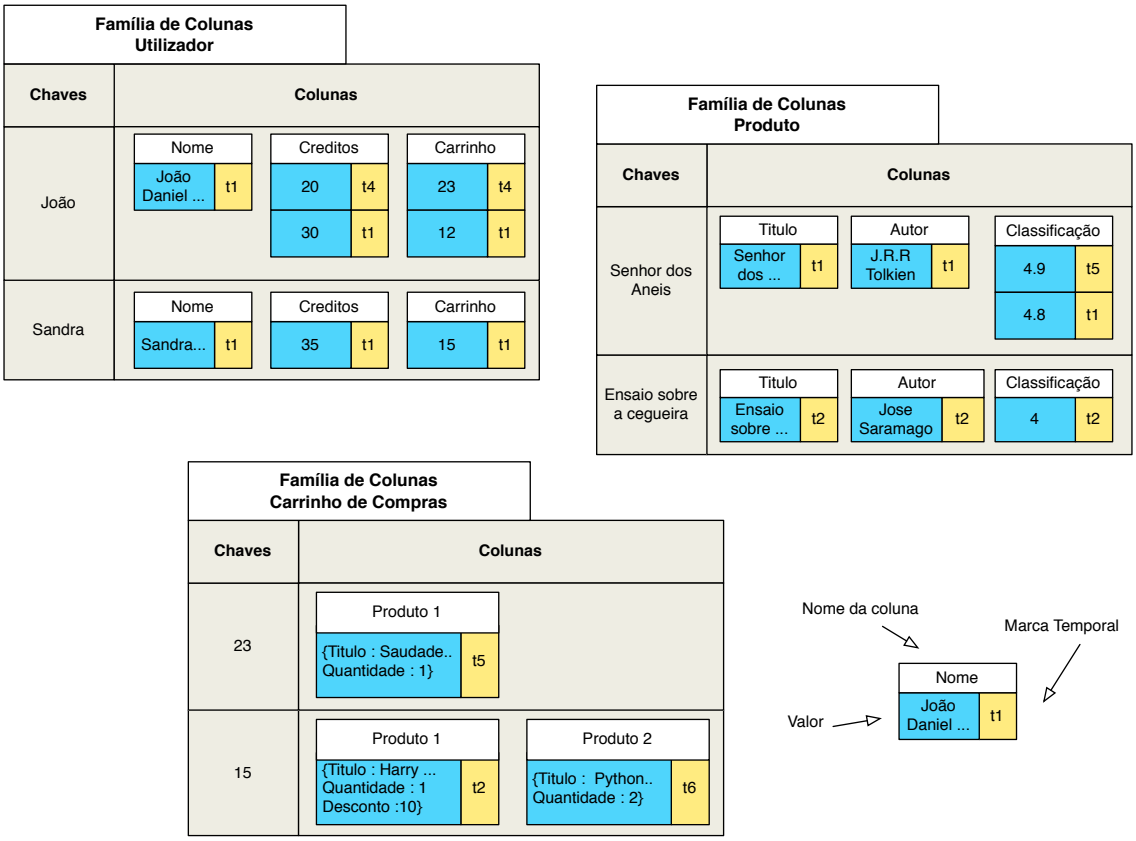


Figura 2.6: Modelo de dados - BigTable e derivados

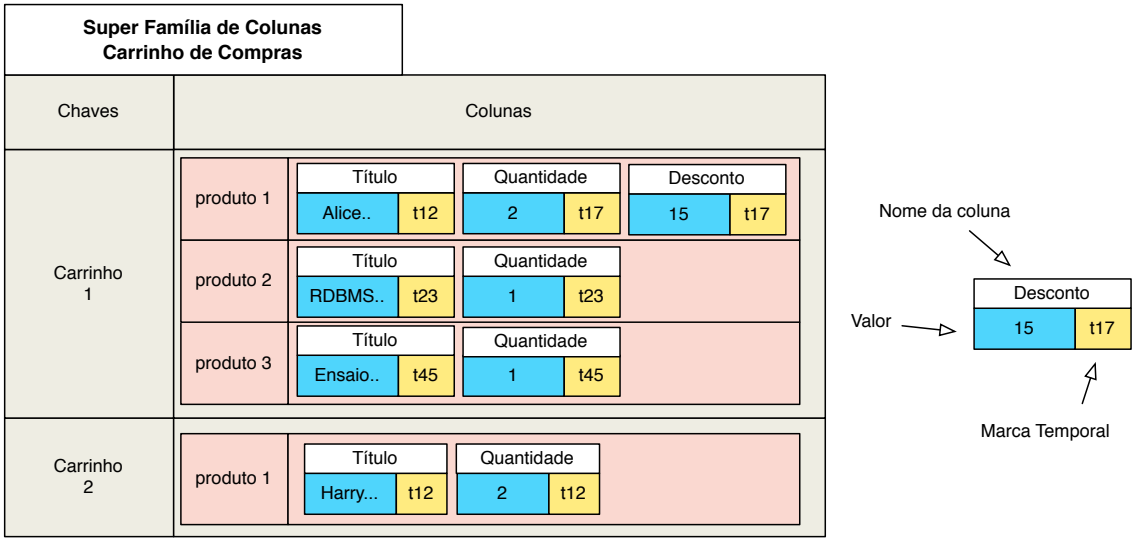


Figura 2.7: Super família de colunas

situações onde o cliente pretende que a base de dados retorne todos os resultados associados a um determinado valor.

Passando à implementação do exemplo, o desenho base do modelo mantém-se, com diferentes famílias de colunas paras as diferentes entidades. Muda no entanto, a natureza da família de colunas correspondente aos carrinhos de compras. Uma vez que são constituídos por vários produtos e correspondente informação, pode-se agora organizar estes dados em várias colunas e agrupá-las sobre super colunas como pode ser visualizado na Figura 2.7. Assim para cada chave e corresponde carrinho de compras, existe uma super coluna para cada produto adicionado. Pode-se ainda aqui usar à imagem do exemplo dado acima, um nome com conotação temporal que permite a leitura dos últimos N elementos adicionados ao carrinho.

Em relação ao esquema de partição podemos argumentar que talvez seja interessante manter as chaves ordenadas para pesquisa de informação sobre os produtos. No entanto sem o necessário cuidado, esta escolha pode levar por exemplo à concentração dos carrinhos de compras mais recentes e por isso mais usados numa só máquina. Alternativamente o arquitecto do sistema pode também optar por uma distribuição aleatória das chaves, recorrendo ao uso de índices secundários para suporte das pesquisas.

Características e implementação

Big Table, HyperTable e HBase Num contexto diferente de soluções como o Dynamo, os sistemas aqui abordadas foram desenhadas para a manipulação de grandes quantidades de dados, mantendo no entanto um modelo coerência forte. Para tal os três sistemas dependem primeiramente de um sistema de ficheiros distribuído subjacente que será responsável pelo armazenamento dos dados e *logs*. Sobre esta estrutura existem depois dois tipos de servidor: os servidores de dados e os servidores de controlo que tipicamente irão eleger um mestre entre si por um mecanismo de *locking*. Os servidores de dados serão responsáveis por uma fatia da informação armazenada, sendo que o mestre irá monitorizar estes mesmos servidores e controlar a distribuição dos dados

entre eles. Esta arquitectura é suportada no BigTable através do uso do GFS [12] em conjugação de um grupo de vários nós a correr o serviço Chubby [4] (responsáveis pela eleição do líder entre eles). Num mecanismo análogo o HBase baseia-se em sistema de ficheiros Hadoop [2] e Apache Zookeeper[13], e o Hypertable usa o Hyperspace (clone do Chubby) permitindo ao utilizador o uso de diferentes sistemas de ficheiros.

Em termos de interfaces de acesso, para além de bibliotecas em diferentes linguagens, tanto no Hypertable como no BigTable existe uma linguagem de consulta semelhante ao SQL de nome HQL (*HyperTable Query Language*) e GQL respectivamente. Estas linguagens contêm no entanto algumas limitações quando em comparação, não permitindo, por exemplo, operações de pesquisa por colunas não indexadas. Transacções nativas são também possíveis ao nível da linha no Bigtable, existindo sobre esta solução dois sistemas de transacções desenvolvidos também pela Google numa fase posterior. Tais sistemas têm o nome de Megastore [11] e Percolator [19]. Entretanto com base em HBase, existe também vários trabalhos de investigação sobre como poderão ser alcançadas transacções neste sistema. [26, 27].

No contexto da implementação do exemplo, avalia-se agora as garantias e funcionalidades oferecidas por estas soluções. Exibindo um modelo mais rico que as soluções baseadas em tuplos de chave-valor, o armazenamento de informação pode agora ser mais estruturado. Deste modo, as leituras e escritas podem assim ser feitas de forma mais granular. Em termos de leitura existem também nestes sistemas multi-versão a possibilidade da extracção das últimas N versões de um elemento, tal como a ordem lexicográfica das chaves permite a realização de operações de leitura por intervalos. Estas funcionalidades permitem ao programador implementar métodos que possibilitem a visualização do histórico do cliente ao longo do tempo, tal como a pesquisa eficiente de informação sobre livros fazendo uso de uma possível ordem natural impressa nos dados. Existem no entanto limitações na leitura dos dados, uma vez que os valores são armazenados de forma opaca. De facto, qualquer procura baseado num título de um livro ou tema não pode ser implementado de forma eficiente, restando como solução o uso de índices.

Para além das operações base, e descartando as transacções que são para já apenas utilizáveis no BigTable, estas bases de dados oferecem também alguns mecanismo que permitem a implementação de operações com necessidades de coerência como é o caso da actualização do stock nos produtos. Tomando como exemplo o HBase, em casos onde a concorrência entre operações seja diminuta, podemos optar por uma estratégia optimista com base em operações de escrita condicional nesta existentes. Neste género de operação, uma escrita apenas será bem sucedida se o valor a substituir for igual ao valor de testa nela incluso. Podem-se assim efectuar alterações ao stock sem qualquer tipo de monitor, enfrentando no entanto o cenário de inanição, onde vários pedidos concorrentes impedem repetidamente a execução desta mesma operação. Na verdade, existem também no HBase operações de *lock*, mas estas podem efeitos nefastos, pois, embora possuam uma API pública, estas foram criadas para uso interno e a sua utilização pode rapidamente causar situações de *deadlock*[14].

Cassandra Numa abordagem em muitos pontos oposta às restantes alternativas da área, o Cassandra baseia a sua arquitectura subjacente no Dynamo. Construída para correr sobre vários nós iguais entre si, esta base de dados não apresenta por desenho pontos de falha. Como tal e à se-

[13] http://hadoop.apache.org/zookeeper/

[14] http://jerryjcw.blogspot.com/2009/10/hbase-notes-casual-remark-about-row.html

melhança do Riak, o seu modelo de coerência baseia-se em factores de replicação e mecanismos de Quórum que permitem ao cliente definir a coerência pretendida em cada operação. Nesta linha, o Cassandra suporta também *eventual consistency*, permitindo assim ao utilizador continuar a usufruir do serviço em caso de uma partição na rede, pondo no entanto em causa a coerência dos dados durante esse intervalo de tempo. Se tal se verificar, a base de dados aplicará mecanismos de resolução do lado do servidor, sendo para isso usado um mecanismo de marcas temporais onde em caso de conflito a actualização mais recente ganha.

Ao abordarmos a implementação do exemplo proposto, os primeiros passos a seguir consistem na definição sobre qual o esquema de famílias a usar no *cluster* e qual o seu factor de replicação entre outras opções. Em termos de interface, os acessos à base de dados são feitos através de Apache Thrift, abrindo assim portas as bibliotecas para inúmeras linguagens. Neste contexto e em termos de API temos acesso às operações base de leitura e escrita, em muito similares às presentes nas soluções anteriores. Entres as oportunidades oferecidas temos também a possibilidade de agrupar operações em *batchs* que permite a execução de múltiplas alterações num só pedido. Sobre esta interface disponível em várias linguagens surgiram posteriormente vários clientes de alto nível. Foi também recentemente adicionada à base de dados uma linguagem de interface com inspiração no SQL de nome CQL. Esta fornece uma interface de programação mais amigável e limpa mantendo muitas das capacidades de interacção das API de baixo nível .

Existe no entanto um problema de implementação, sendo que se nenhuma garantia transaccional é fornecida pelo motor de dados a actualização concorrente do stock pode assim causar rupturas do mesmo. Este pode no entanto ser colmatado com o uso de uma base de dados externa com garantias transaccionais para assim lidar com o stock dos produtos, ou o uso de um mecanismo de locking externo como ZooKeeper[15]. Existem ainda assim algumas garantias de atomicidade ao nível da linha. De facto, garanta-se que um conjunto de campos que constituem um produto adicionado à base de dados será sempre escrito na totalidade ou rejeitado por completo. Esta garantia prende-se apenas com o estado final dos dados, sendo que não existem quaisquer garantias de isolamento.

Existem depois as diversas oportunidades fornecidas pelo modelo, tal como o uso das super famílias de colunas para a materialização de resultados, podendo-se por exemplo agrupar todos os livros consoante os seus autores para tornar a pesquisa por este campo mais eficiente. O mesmo se aplica ao uso das opções de ordenação quer entre linhas ou nas colunas.

2.3 Interfaces baseadas em objectos

Num mercado ainda hoje dominado pelo paradigma relacional e pelo paradigma de programação orientada a objectos, o mapeamento de objectos tem a sua base no esforço dos criadores de software para união destes dois mundos. Conjugados com bases de dados relacionais baseadas em valores escalares, os objectos presentes nestas linguagens têm assim de ser simplificados e desconstruídos para o seu armazenamento. Num contexto Java e de forma similar noutras linguagens, este trabalho pode ser realizado através de interfaces de programação de baixo nível como o JDBC (*Java Database Connectivity*). Estas levam no entanto a soluções verbosas e de demorada codificação. Abordando este problema e tentando automatizar este processo surgiram assim soluções

[15]http://ria101.wordpress.com/2010/05/12/locking-and-transactions-over-cassandra-using-cages/

ORM - Hibernate

```
Session s = HibernateUtil.getSessionFactory().getCurrentSession();
Livro livro = new Livro("Alice","Fantasy",20);
s.save(livro);
```

JDBC

```
//Connect

Class.forName("com.mysql.jdbc.Driver").newInstance();
String url = "jdbc:mysql://localhost/database";
conn = DriverManager.getConnection(url, "user", "pass");

//Save

Livro livro = new Livro("Alice","Fantasy",20);
Statement st = conn.prepareStatement("INSERT INTO LIVROS" +
  "VALUES (?,?,?)");

st.setString(1,livro.getNome());
st.setString(2,livro.getTipo());
st.setInt(3,livro.getStock());

st.execute();
```

Figura 2.8: ORM versus JDBC

de mapeamento objecto-relacional (ORM na sigla anglófona). Estas permitem, através de API padrão como o JDO (*Java Data Objects*) ou JPA (*Java Persistence API*), garantir a persistência e leitura de objectos de forma transparente sendo o código SQL responsabilidade da camada de mapeamento.

Estas soluções baseiam-se em primeiro lugar num mapeamento previamente elaborado pelo programador, podendo este ser externo à aplicação (ficheiro de XML) ou em soluções actuais embebido no código fonte através de anotações. Estas são ferramentas flexíveis que permitem ao programador definir como a aplicação e os dados se conjugam. Este pode assim escolher entre o quanto a aplicação se deverá adaptar aos dados que possivelmente são de maior importância do que esta ou se pelo contrário deverá ser o modelo de dados que deverá integrar da melhor forma o modelo de negócio que a aplicação representa [20].

As vantagens encontram-se na redução tanto do tempo de desenvolvimento aplicacional através de uma abstracção do modelo relacional, como dos testes associados que são assim desnecessários. As ferramentas de ORM têm também outras vantagens, uma vez que implementam os melhores padrões de desenho de software e assim promovem a uniformização do código gerado por diferentes elementos de uma mesma equipa. Na Figura 2.8 pode-se compara a diferença entre um cliente baseado numa API como JDBC e uma opção de ORM tal como o Hibernate[16].

2.3.1 Desafios principais

Apresentam-se agora alguns dos conceitos presente nas várias soluções de ORM que hoje se encontram no mercado, tal como noções de mapeamento, identidade entre outras. Embora muitos

[16]http://www.hibernate.org/

destes conceitos se apliquem também a outras línguas e soluções, como o SQLalchemy[17] para Python, este estado da arte tem como alvo soluções com base em Java, linguagem usada no contexto da dissertação.

Tipicamente uma solução de ORM será uma camada de abstracção entra a aplicação criada e a base de dados, fornecendo uma interface de programação mais simples ao seu utilizador. Como primeiro passo, o programador irá estabelecer quais os objectos a persistir e como serão mapeadas na base de dados as relações de herança e entre instâncias. Este mapeamento, tal como a configuração de opções de cache, pode ser feito em ficheiros de XML ou alternativamente por anotações JDO ou JPA consoante a especificação implementada [17].

A plataforma irá então ser responsável pela manutenção das conexões e por mecanismos de alteração de bytecode, geração de código ou reflexão, irá automatizar as operações base de persistência e leitura de objectos. No seu funcionamento a plataforma irá tipicamente depender de contextos de persistência, que podem em diferentes soluções ter nomes como *PersistenceManager* ou *Session*. Estes representam um objecto temporário onde numa conversação entre a aplicação e a base de dados se estabelece a conexão e se criam transacções [20].

Relações e mapeamento As relações entre entidades caracterizam-se neste contexto em termos de cardinalidade e direcção. Em termos de direcção as relações podem ser unidireccionais caso apenas uma das entidades envolvidas tenha conhecimento da relação. Temos como um exemplo uma promoção que referencia um livro. Se pelo contrário ambas as entidades tiverem conhecimento da ligação, esta será assim apelidada de bidireccional.

Já em termos cardinalidade as relações caracterizam-se como sendo de um-para-um, um-para-muitos ou muitos-para-muitos, de acordo com o número de ligações que as entidades possuam entre si. Como exemplo podemos dar a ligação entre um livro e o seu autor, onde se se considerar que cada livro terá apenas um autor então a sua relação é de um para um. Se por outro lado cada autor tiver mais de um livro esta ligação será então um para muitos e finalmente se um livro puder ter também vários autores então esta será de muitos para muitos.

As relações são um dos pontos importantes no mapeamento das aplicações para a camada de persistência. Cabe ao programador a tarefa de sinalizar estas mesmas relações no ficheiro XML de mapeamento ou por anotações, configurando o modo como elas serão feitas. Abordando o modo como este mapeamento é depois realizado no modelo relacional, observámos agora as várias estratégias para cada um dos tipos de relação. As mais simples, de um para um, são geralmente feitas através da partilha da chave de uma das entidades, o que em linguagem relacional se denomina chave estrangeira. Pode no entanto existir, em raras situações, uma terceira entidade no motor de base de dados que estabeleça a relação entre as duas. Tal solução é sim a geralmente usada para o mapeamento das relações de muitos para muitos, onde existe uma entidade extra na base de dados que codifica a relação entra as duas entidades. Quanto às relações de um para muitos, estas podem ser usualmente codificadas das duas maneiras, quer pelo uso de uma mecanismo do tipo chave estrangeira, quer pelo uso de uma tabela de união.

Mas o mapeamento não se resume somente a relações entre as entidades mas também às suas relações de hierarquia. Tomemos como exemplo uma classe produto e duas classes livro e filme que descende desta, analisando as várias estratégias de mapeamento. Uma opção é a criação de uma única tabela na base de dados, onde todas as entidades que descendem de produto são

[17] http://www.sqlalchemy.org/

persistidas, existindo depois um campo de discriminação. Outra estratégia passa pela criação de uma representação na base de dados para cada uma das entidades filho com replicação de possíveis atributos herdados. E como método alternativo aos anteriores, temos a como criação de uma tabela para todas entidades do modelo, sendo que os campos de um livro serão assim persistidos separadamente na tabela da entidade produto e livro. Outras opções poderão também definidas no mapeamento das classes como a possibilidade de um dos seus campos ser escrito na base de dados de forma serializada, ou simplesmente não persistido sendo caracterizado como transiente.

A manutenção da integridade na base de dados é igualmente um ponto importante, sendo útil ao programador ter noções de *Cascading*. *Cascading* representa o mecanismo pelo qual uma acção numa entidade da base de dados afecta todas as outras a si relacionadas em termos de herança ou por outra qualquer ligação, de forma a garantir a coerência dos dados [15].

Identidade Nos diferentes motores de bases de dados a unicidade das chaves que identificam a informação armazenada é também um ponto vital que não pode ser ignorado nas plataformas de OM. Assim as plataformas geralmente oferecem mecanismos para a geração automática de chaves. Estes mecanismos tomam partido de operações fornecidas pela base de dados como o incremento atómico, ou então baseiam-se em geração de uuids com base em tempo ou IP. Numa opção alternativa o programador pode simplesmente indicar quais campos a ser usados como chaves das entidades.

Objectos em memória ou persistidos. Existe normalmente no contexto das plataformas de ORM uma distinção entre objectos em memória e objectos persistidos. De facto, soluções como Hibernate ou DataNucleus[18] possuem, por motivos de desempenho, mecanismo de cache em memória. Divididos muitas vezes em vários níveis, estes mecanismos possuem uma primeira cache normalmente ao nível do contexto de persistência, existindo depois caches de nível dois para manter em memória objectos raramente actualizados.

Existe também nestas soluções a noção de diferentes estados pelos quais os objectos podem passar. Existe assim o estado transiente aplicado se o objecto acabou de ser criado e não se encontra associado a nenhum contexto de persistência. Quando o objecto se encontra associado a este contexto então ele é considerado persistente e acredita-se que o seu valor é igual ao existente na base de dados. Existe ainda o estado separado (*detached*) que aplica-se quando o objecto já não se encontra associado a um contexto de persistência, estando sujeito a alterações que não serão repercutidas na base de dados, até que o programador assim o diga.

São também comuns nestas soluções conceitos como *lazy fetching*. Estes mecanismos implementados em diferentes soluções, permitem optimizar as leituras de objectos definindo quais os dados a carregar, no que se usualmente se chama de grupos de busca (*fetch groups*).

2.3.2 Opções actuais

No contexto de soluções de ORM para Java apresentam-se agora alguns dos produtos observados. Numa secção final, é também apresentado como alguns projectos investigadas no contexto

[18] http://www.datanucleus.org/

desta dissertação se propõem também elas fornecer uma solução de mapeamento de objectos para Cassandra.

Hibernate Hibernate é uma das mais conhecidas e usadas soluções de ORM para Java. Plataforma gratuita e de licença LGPL, o Hibernate consiste numa implementação da interface JPA com uso de *proxies* baseados nas capacidades de reflexão do Java.

Suportando a persistência semitransparente de objectos, esta solução permite também a criação e uso de instruções mais ricas que podem ser usados nos dados existentes. Para tal o Hibernate fornece três opções: o mapeamento de instruções em SQL nativo, o uso de HQL (*Hibernate Query Language*) uma linguagem de inspiração SQL ou numa linguagem por alguns considerada mais amigável, a utilização das *Criteria Queries*.

Através de ficheiros XML de configuração ou por anotações JPA, esta plataforma permite um mapeamento flexível das entidades, tal como a configuração dos mecanismos de cache entre outras opções. Na plataforma em si, o programador tem depois controlo sobre transacções, grupos de busca para optimização das leituras entre outras características.

Datanucleus O Datanucleus é uma plataforma actualmente conhecida pela sua modularidade e adaptabilidade a vários motores de dados quer sejam eles relacionais, orientados a objectos ou do novo movimento NOSQL. Tal é possível pois esta é uma das únicas soluções que implementa não só a especificação JPA, mas também a especificação JDO originalmente idealizada para o uso de diferentes tipos de base de dados.

Em termos de suporte a instruções de alto nível, a plataforma tem também implementadas linguagens de consulta orientadas a objectos. De sintaxe semelhante ao SQL estas têm o nome de JPQL e JDOQL, definidas na especificação tanto do JPA como do JDO respectivamente. Existe também um suporte para a introdução de instruções em SQL nativo à semelhança do Hibernate.

Baseado num mecanismo de *plug-ins* de fácil extensão e implementação, este é uma plataforma de facto interessante para criar suporte para novos motores de base de dados, ou simplesmente para adicionar um novo mecanismo de cache ou de geração de chaves aos já existentes. Assim esta solução para além de várias bases de dados relacionais possui também suporte para outros produtos como HBase, MongoDB ou LDAP.

Outras abordagens

Saindo das soluções mais maduras e originalmente orientadas ao modelo relacional, existiram também com aparecimento do movimento NOSQL alguns projectos que tentaram criar uma ponte entre estas bases de dados e o paradigma da programação orientada aos objectos. Entre as diversas soluções encontradas existem algumas que são aqui avaliadas pois têm um propósito similar ao trabalho que é descrito neste documento.

HelenaORM Baseada em objectos de acesso a dados (DAO na sigla anglófona), padrão de desenho de aplicações direccionadas à persistência de dados o HelenaORM[19] é uma solução simples de OM baseada em Hector[20] um cliente de alto nível para Cassandra.

[19] http://github.com/marcust/HelenaORM
[20] http://github.com/rantav/hector

Nesta ferramenta, o mapeamento das entidades é feito através de anotações inclusas nas classes. Esta anotações explicitamente demarcam qual o tipo de família de colunas a usar e quais os campos que serão usados como chave ou que mapeiam uma possível super coluna. Depois de mapear as classes, dentro de aplicação o cliente terá apenas de criar um objecto DAO através do qual persiste os seus dados e realiza leituras sobre os mesmos. Não existem no entanto no contexto desta ferramenta quaisquer mecanismos de relação entre entidades.

OCM Esta é outra das primeiras tentativas de mapeamento de objectos para Cassandra. Também ela baseada em Hector, o OCM baseia-se na geração de código a partir de uma descrição do modelo de dados pelo utilizador dada[21]. Embora sem suporte para relações entre objectos esta solução introduzia algumas vantagens como a manutenção de índices secundários numa versão do Cassandra que ainda não suportava estas estruturas.

Datanucleus - HBase Implementada numa outra base de dados NOSQL, este plug-in é uma das primeiras tentativas de mapeamento de objectos neste género de soluções com suporte a uma interface de persistência como é o JDO. Este *plug-in*, que também não suporta a persistência de relações entre objectos, é uma prova de conceito em como se pode adaptar uma plataforma madura como o Datanucleus a uma opção não relacional ainda que não fornecendo as mesma garantias que os *plug-ins* relacionais[22].

2.4 Sumário

Neste capítulo percorremos a história das bases de dados, focando nesta nova era que agora vivemos, marcada pelo aparecimento de dezenas de novas soluções não relacionais. Entre os sistemas analisados, estudamos a fundo aqueles que se focam nos desafios actuais de grande escala descrevendo algumas das suas características mais marcantes através de casos exemplo.

Numa primeira fase do projecto existiu a necessidade de escolher uma solução para realizar este estudo. De entre estas opções estudadas, o lote de escolha resumia-se ao Riak, Cassandra ou HBase. Por não garantir a duplicação dos dados por mais de um nó descartamos à partida o Riak. O mesmo acabou por acontecer o HBase não só pela complexidade da sua arquitectura, mas também por possuir um nível de maturidade inferior e uma comunidade reduzida aquando do inicio deste trabalho.

Escolhemos assim o Cassandra pela sua arquitectura sem pontos de falha, modelo intrincado de colunas e super colunas e também pelas seus mecanismos de replicação e afins que a tornam uma base de dados que facilitam a integração a cenários com múltiplos centros de processamento de dados. Esta solução introduz no entanto um nível de complexidade superior a uma base de dados relacional e a implementação de um sistema sobre tal sistema pode assim exigir uma curva de aprendizagem demasiado acentuada para ser viável.

Numa fase posterior criamos uma camada de mapeamento de objectos sobre esta solução para facilitar a transição ou construção de sistemas sobre a mesma, razão pela qual aqui introduzimos o tema do mapeamento de objectos sobre bases de dados. Com o objectivo de produzir uma solução mais madura e que ao mesmo tempo permitisse a abstracção dos pormenores base de

[21] https://github.com/charliem/OCM
[22] http://www.datanucleus.org/products/accessplatform_3_0/hbase/support.html

implementação, optou-se numa fase prévia pela extensão de uma plataforma já existente. Tal escolha permite uma focagem em problemas como o mapeamento de entidades e relações, sendo as operações de leitura de anotações, manutenção dos meta-dados e mecanismos de cache geridos pela plataforma.

Com este intuito foi assim escolhida a plataforma DataNucleus, conhecida pelo seu mecanismo de plug-ins e extensões. Esta solução permite deste modo uma fácil adaptação de um novo motor de dados através do desenvolvimento de um novo *plug-in*, e com o mecanismo de extensões possibilita também um desenvolvimento faseado de novas características sobre o mesmo. Esta é já uma plataforma com conhecidas adaptações para soluções NOSQL como HBase e MongoDB, sendo também a solução usada pela Google para fornecer uma interface Java para o App Engine[23].

[23]http://code.google.com/appengine/

Capítulo 3

Migração do TPC-W

Com o objectivo de avaliar o impacto do uso de uma solução não relacional como o Cassandra, foi numa primeira fase deste trabalho estabelecida como meta a construção de uma plataforma de testes para diferentes motores de dados. Com o objectivo definido de avaliar as diferenças na passagem do modelo relacional para motores de dados NOSQL, criou-se assim uma implementação parcial da especificação fornecida pelo TPC-W [23]. Na plataforma criada sobre este standard, são testados soluções como o Cassandra e MySQL, sendo posteriormente utilizada também para o teste do componente de OM desenvolvido. Esta plataforma é descrita em mais pormenor no Anexo A.

3.1 Descrição geral

O TPC-W [23] é um *benchmark* que retrata uma situação de comércio Web onde uma loja de venda online é simulada. A sua escolha neste contexto prende-se em várias razões:

É de raiz desenhada para o modelo relacional De facto o *benchmark* TPC-W é na sua essência direccionado ao modelo relacional, quer no seu modelo de dados quer na descrição das operações nele executadas (algumas com exigências transaccionais ou de atomicidade). Este representa um ponto de partida bem definido através do qual podemos avaliar uma transição para uma base de dados não relacional.

Representa um caso real Sendo baseada num cenário de venda online que hoje se tornou comum, o TPC-W fornece um caso de estudo realista. Deste modo podemos avaliar quais as dificuldades do comum programador quando se move entre os dois mundos.

A diversidade de operações Existem no TPC-W operações de diferentes tipos que depois agrupadas em diferentes perfis de teste, podem ser usadas para avaliar as soluções propostas sob vários ângulos.

Numa implementação completa o cliente terá acesso a um conjunto de páginas Web onde se representa a loja virtual. Nestas, o utilizador pode pesquisar produtos, consultar a sua informação, adicioná-los e ver o seu estado num carrinho de compras virtual, e por fim comprá-los. Entre as 14 especificadas, existem operações de escrita como o adicionar de um produto à lista de compras, ou

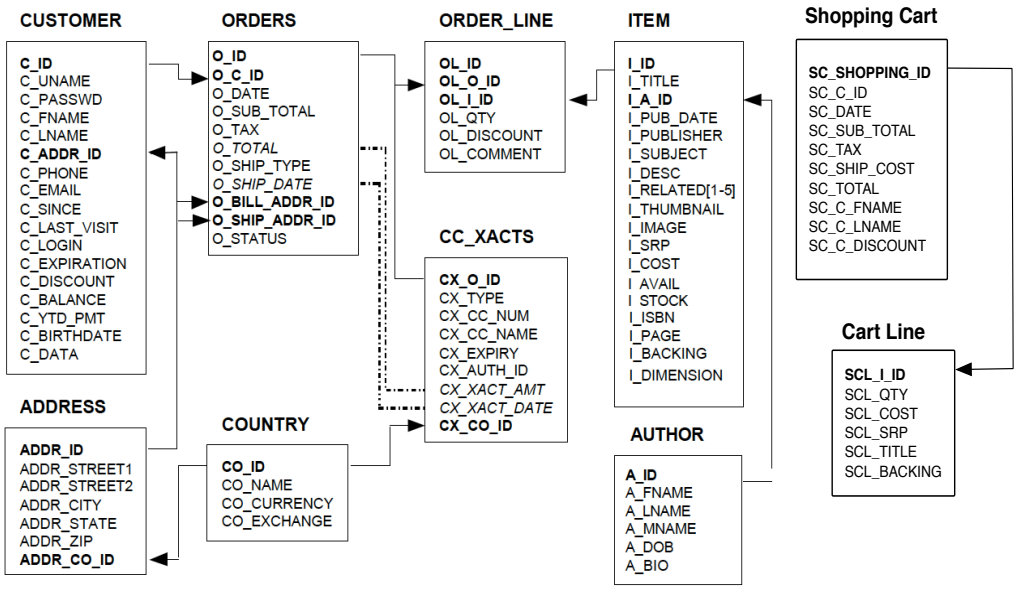


Figura 3.1: Modelo de entidades no *benchmark* TPC-W [23]

a realização de encomendas. Em termos de operações de leitura existem a simulação da procura de produtos por título, autor ou tema, e também a determinação de quais destes são os mais recentes ou os mais vendidos. Ao conjunto constituído maioritariamente pelo primeiro tipo de operações damos o nome de compra (*order*), sendo as operações de leitura e pesquisa são agrupadas sobre o conjunto de consulta (*browse*).

A variação na razão entre as operações de cada tipo define depois os três tipos de perfis de benchmark existentes:

- Perfil de consulta (*browsing mix*), com 95% de interacções de consulta e 5% de compra;
- Perfil de compra (*shopping mix*), com 80% de interacções de consulta e 20% de compra;
- Perfil de encomenda (*ordering mix*), com 50% de interacções de consulta e 50% de compra.

Baseado num modelo base com 8 entidades, apresentado na Figura 3.1 o TPC-W inclui também na sua especificação a definição de *shopping cart* e *cart line* que representam o carrinho de compras de um normal cliente. No modelo descreve-se igualmente quais as relações entre as diversas entidades tal como publicado no documento inicial. Deste modo as linhas picotadas representam as relações de um para um, sendo que as restantes representam relações de um para muitos, indicando a seta o sentido da ligação. Da mesma forma o itálico representa os campos com o primeiro tipo de relação, sendo o negrito usado para os afectados pelo segundo tipo.

A loja Web simulada funcionará sobre este modelo, existindo uma posterior descrição de como se deverão processar as transições entre as várias páginas web e respectivas operações nestas executadas. As métricas do sistemas como definidas na especificação são as taxa de transferência e também uma razão entre estas e o custo (representando os custos de compra, hardware e software).

3.2 Implementação não relacional

Numa primeira fase deste trabalho, procedemos á implementação da especificação TPC-W sobre a base de dados não relacional Cassandra. Várias são os passos de adaptação a que foi sujeita a especificação do TPC-W no decorrer da implementação, sendo estes aqui divididos por tópicos para uma melhor compreensão. Embora muitas das características apresentadas sejam exclusivas ao Cassandra, existem aquelas que são comuns a outros sistemas similares em termos de modelo de dados ou coerência, sendo este assim um processo pedagógico que vai além desta solução.

Partindo da implementação relacional em grande parte baseada na especificação TPC-W e no modelo apresentado na Figura 3.1, observámos os passos de transição para Cassandra. Como primeiro passo, olhamos para o modelo de dados original e descrevemos alguns dos padrões de modelação e como estes se aplicam a este caso em específico.

Abordando depois as questões de interface com o Cassandra, resumimos igualmente algumas das actuais opções de interface existentes com base em algumas das operações implementadas. Estas são descritas em termos da sequência de instruções que executam, tal como das restrições a elas associadas. Entre os desafios encontrados, estão os requisitos de atomicidade na escrita de muitos dos dados ou uso de transacções para garantir a coerência dos mesmos.

3.2.1 Modelo de dados

Quando observando os estruturas de dados oferecidas, não só pelo Cassandra como também por outras soluções como o HBase e restantes clones do BigTable, nota-se a similaridade entre tabelas e famílias de colunas. Tais estruturas não são no entanto equivalentes, de modo que as tabelas do modelo relacional não podem simplesmente ser transformadas em famílias de colunas.

Na verdade, com uma interface de acesso simplificada, realidade comum nas soluções NOSQL, existe nas bases de dados como o Cassandra uma maior interligação entre o modelo e as operações sobre os dados executadas. Como tal, esta transição entre paradigmas implica muitas vezes uma desnormalização dos dados e uma reorganização do modelo para a sua natural indexação como única solução viável para a implementação de algumas das operações pretendidas. Partindo de um modelo relacional onde operações de alto nível são construídas para se adaptarem a um modelo de dados definido, uma das primeiras e mais importantes lições na transição para o universo NOSQL é que aqui estes papéis são invertidos. Aquando da modelação do esquema de dados em tais soluções, o programador terá primeiro de ter em mente quais os dados que pretende aceder na base de dados e só então construir o modelo de forma a optimizar tais operações. Um exemplo de tal fenómeno, quando migramos entre paradigmas é a transformação de instruções de selecção e de agregação como os *Selects* e *Joins* que dão lugar à desnormalização dos dados e à construção de vistas na base de dados.

Seguindo este padrão analisamos as diferentes operações a implementar tal como descritas na especificação do TPC-W em termos de argumentos de entrada e acesso aos dados. Com este conhecimento, apresentam-se agora os diferentes factores de modelação e alguns dos padrões usualmente presentes em soluções deste tipo.

Famílias de colunas / Super famílias de colunas Distinguido-se dos modelos de dados similares, existe no Cassandra a opção de armazenar os dados quer em simples famílias de colunas ou então em super famílias de colunas. A escolha entre estas estruturas prende-se maioritariamente

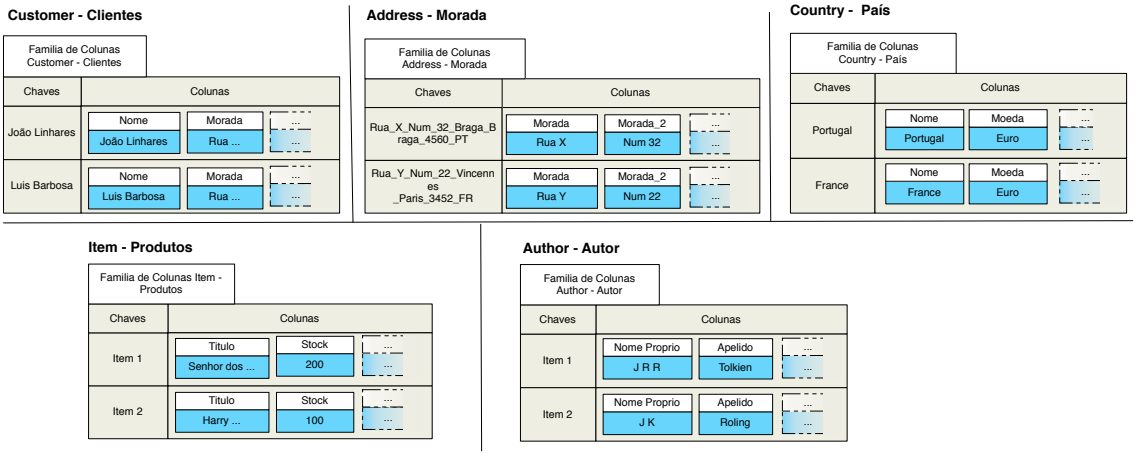


Figura 3.2: Entidade TPC-W representadas em famílias de colunas

com o acesso aos dados que tal família irá sofrer no futuro, evidenciado-se aqui que realmente são as operações que definem o modelo.

As famílias de super colunas servem como estrutura para a indexação dos dados, permitindo saber por exemplo obter informação sobre os diferentes produtos que se encontram actualmente no carrinho, sendo cada um destes uma super coluna sobre a chave que identifica a entidade no seu todo. Estas estruturas possuem no entanto limitações, pois não sendo este terceiro nível de endereçamento indexado, o acesso a um só campo de uma super coluna leva à desserialização de todas as super colunas. Tais estruturas são apenas apropriadas para acessos que usem de toda a informação nelas contida.

Tabelas como os clientes, endereços, países, produtos e autores são armazenados em simples famílias de colunas, por vezes complementadas com índices secundários. Existem no entanto casos onde o uso de uma super família de colunas se justifica. É o caso do carrinho de compras já usado como exemplo e onde se agrupa a informação sobre os produtos e o carrinho em si pois esta é usualmente acedida em conjunto. Outras tabelas foram também convertidas em famílias de super colunas, como é o caso das duas tabelas que guardam informação sobre as encomendas e que fundidas numa só destes estruturas. Este tipo de família de colunas é também usado para guardar a informação de facturação por cliente.

Podemos ver este mesmo modelo nas Figuras 3.2 e 3.3 onde, agrupadas segundo o tipo de família usada, podemos ter uma melhor imagem de como as diferentes entidades são persistidas em Cassandra.

Chaves Sendo o Cassandra, tal como muitas outras bases de dados NOSQL, um sistema onde o principal meio de acesso aos dados é o uso das chaves, estas entidades devem ser cuidadosamente pensadas pelo programador. De facto, se excluirmos o uso de índices secundários, as API base baseiam-se na identificação dos dados pela sua chave e como tal esta deverá conter um significado implícito que permita um posterior acesso.

Tomando por exemplo o caso da entidade cliente, para que a informação de cada utilizador seja prontamente acedida no processo de login, a sua chave terá de ser o nome de registo do mesmo. Por uma questão de utilidade esta chave será também utilizada na informação de facturação permitindo

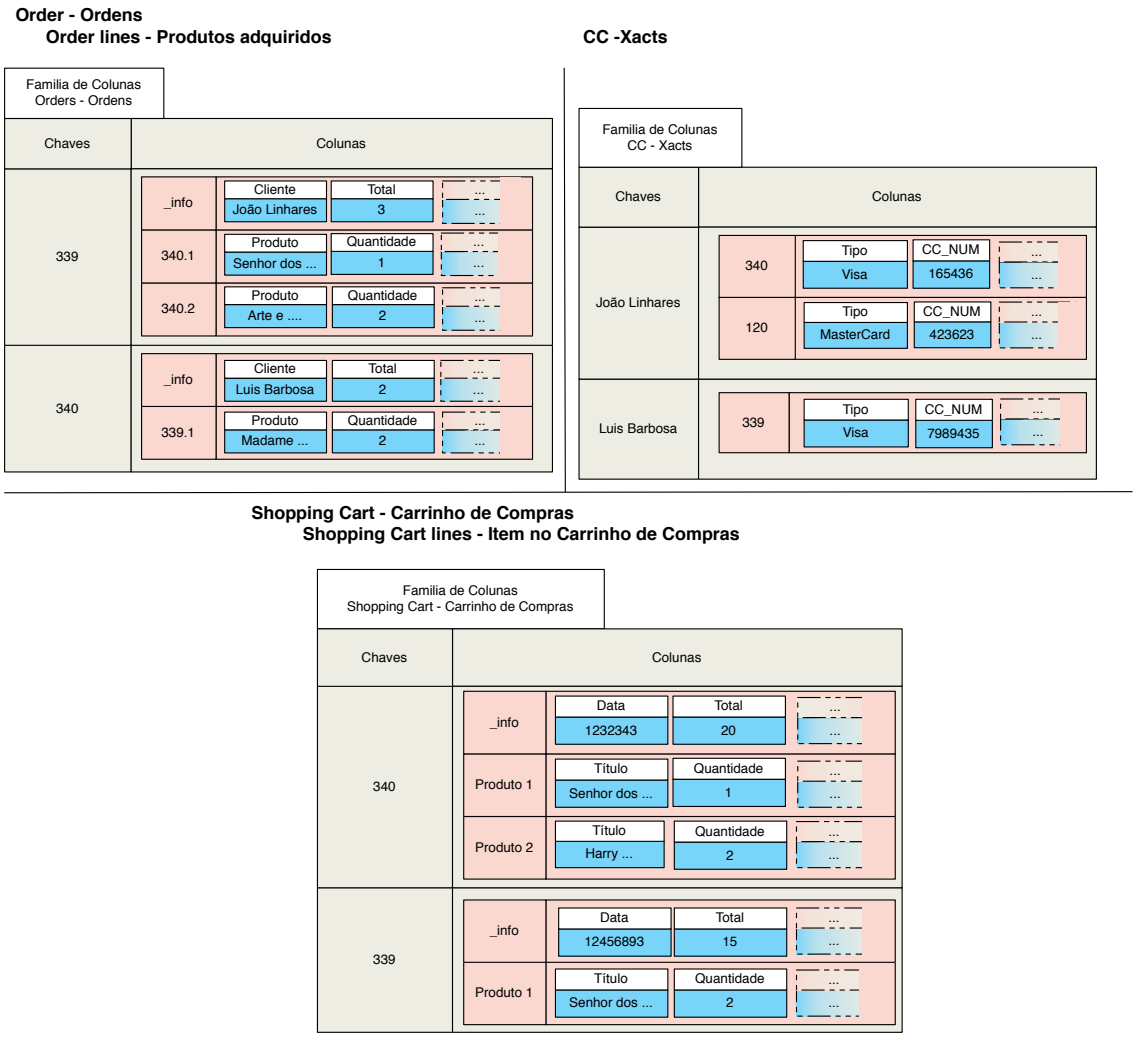


Figura 3.3: Entidade TPC-W representadas em famílias de super colunas

assim um rápido acesso a estes dados. Para além destas estratégias as chaves podem também ser usadas para evitar a inserção de informação duplicada. Esta questão surge do requisito descrito na especificação onde se diz que uma nova morada será apenas inserida se não existir um registo na base de dados que possua o mesmo nome de rua, cidade, etc. Para efectuar este teste de forma eficiente em Cassandra (solução extensível a outras base de dados), usamos como chave uma agregação dos campos a testar. Desta forma facilmente se verifica se o registo existe ou não na base de dados. Esta solução leva no entanto a um aumento no consumo de espaço em disco, problema normalmente associado a estas base de dados devido aos seus esquemas dinâmicos e frequente desnormalização dos dados.

Mas não só o desenho das chaves deve ser aqui um factor de ponderação. De facto, quando inseridos no Cassandra, os dados irão ser distribuídos pelos diferentes nós com base nestas mesmas chaves. Este mecanismo de distribuição não é no entanto fixo, podendo o administrador da base de dados definir, numa fase anterior à inserção dos dados, se deseja que estes seja distribuídos de forma aleatória ou ordenados segundo um factor lexicográfico (em versões inferiores a 0.6) ou binário (em versões superiores a 0.7). Esta decisão tem um papel fulcral no sistema em desenho uma vez que uma distribuição ordenada dos dados permite a pesquisa sequencial de informação mas pode levar à criação de uma má distribuição da carga sobre o sistema. A outra opção é então um modelo de distribuição mais eficiente, mas que não permite tais operações (pelo menos de forma eficiente). Como solução híbrida o programador pode optar por um esquema de particionamento que ordene os dados entre os nós, mas ao inserir dados que não necessitem de tal garantia, a sua chave é sujeita a um algoritmo de dispersão e assim garante-se a sua uniforme distribuição.[1]

Índices A simples modelação de famílias de colunas e das correspondentes chaves pode no entanto não ser suficiente para a implementação das operações pretendidas. Tomemos o caso da procura de produtos por tema ou autor, tal operação necessita que exista na base de dados uma estrutura que permita a pesquisa de instâncias desta entidade por campos que não a sua chave primária.

Para este efeito podem ser adicionados à base de dados índices de pesquisa que podem por exemplo mapear quais os produtos que tem um determinado tema ou autor. A partir da versão 0.7 do Cassandra, este género de estrutura é suportada pela base de dados tornando mais fácil o papel do programador uma vez que a actualização dos mesmos teria de ser inteiramente gerida por ele em versões anteriores. Ainda assim, índices baseados em simples famílias de colunas são ainda hoje aconselhados para situações onde os campos indexados possuem uma elevada cardinalidade (os índices nativos são mantidos localmente e podem prejudicar o desempenho de outras operações).

Mas indo além de simples índices, com o uso de super colunas é possível no Cassandra a materialização de vistas na base de dados. Esta é de facto a solução usada para a pesquisa de informação sobre os diferentes produtos na base de dados, uma vez que podemos a cada nome de autor não só associar a identificação dos produtos mas também outra informação como o seu nome, tema e outros campos tal como especificado no TPC-W, permitindo assim a sua leitura directa. Outra das vantagens, é que podemos aqui mapear campos de mais de uma entidade, tendo assim por tema, informação não só sobres os livros associados mas também sobre os seus autores. A grande desvantagem desta abordagem é que sendo a informação destas entidades espalhada por várias famílias de colunas, maior é a carga posta sobre o programador e maior o tempo de escrita

[1] Para mais informação sobre o assunto:
http://ria101.wordpress.com/2010/02/22/cassandra-randompartitioner-vs-orderpreservingpartitioner/

dos dados para que não se perca a coerência entres estas.

Ordenação de colunas Da mesma forma que operações leitura e agregação de informação levam à criação de novas famílias de colunas, também operadores como o *Order by* dão aqui lugar à ordenação natural dos dados. De facto para além da possível ordenação das chaves entre os nós, também as colunas podem ser ordenadas de diferentes formas em cada uma das famílias de colunas.

Várias são as estratégias que aqui usamos baseadas nestas ordenações. Em primeiro lugar todas as encomendas registadas tem como chave um identificador decrescente que permite com uma simples pesquisa pelas primeiras linhas recolher as mais recentes. Na família de colunas que armazena a informação bancária guarda-se também estes dados de forma ordenada para cada um dos clientes para garantir que sabemos sempre qual a transacção mais recente. No caso dos índices secundários esta ordenação pode também ter interesse pois permite que o utilizador obtenha apenas os últimos dados indexados.

3.2.2 Operações de leitura

Nesta descrição que aqui se inicia e que se prolonga pela Secção 3.2.3 e Secção 3.2.4, propõe-se uma avaliação das API de acesso ao Cassandra agrupadas segundo os diversos tipos de operação: escrita, leitura de uma chave e leitura sequencial. Entres os clientes analisados temos em primeiro lugar, servindo como elemento de comparação, o SQL desenvolvido para MySQL. Temos depois três das várias opções de acesso a Cassandra, começando pelo Thrift como cliente de baixo nível e passando depois ao cliente de alto nível Hector[2], sendo as versões usadas destes clientes, as correspondentes à versão 0.7 do Cassandra. Por fim terminamos com a nova linguagem de interrogação CQL que corresponde à versão 0.8 do Cassandra.

Focando nas operações de leitura, como exemplo para esta demonstração optamos pelo uso da operação associada à página inicial do site simulado no TPC-W (originalmente *Home*). Em termos de instruções executadas sobre os dados, esta irá requerer o nome do cliente e os identificadores e imagens em miniatura de cinco produtos. Estes cinco são seleccionados pela escolha de um produto através de uma distribuição uniforme e posterior leitura dos seus relacionados.

Tomando esta operação como exemplo para demonstrar as diferenças entre as interfaces que marcam cada um dos paradigmas numa simples leitura, apresentamos agora respectivo código.

SQL Tomando como base a implementação relacional, demonstramos agora o usual processo de desenvolvimento neste paradigma que passa em primeiro lugar pelo desenvolvimento das instruções SQL e depois pela sua execução na plataforma através de JDBC. Entre os diversos usados, apresenta-se aqui uma das instruções SQL desta operação:

```
SELECT Relacionado.id, Relacionado.thumbnail
    FROM Produto Seleccionado, Produto Relacionado
    WHERE
      (Seleccionado.relacionado1 = Relacionado.id OR
```

[2]https://github.com/rantav/hector

```
    Seleccionado.relacionado2 = Relacionado.id OR
    Seleccionado.relacionado3 = Relacionado.id OR
    Seleccionado.relacionado4 = Relacionado.id OR
    Seleccionado.relacionado5 = Relacionado.id)
  AND  Seleccionado.id = ?
```

Como podemos ver, com uma simples instrução é possível num só contacto com a base de dados extrair os cinco produtos relacionados, ilustrando-se deste modo a capacidade de síntese desta linguagem. Na sua integração com a linguagem Java perde-se no entanto algumas destas vantagens uma vez que o código de preparação e extracção dos dados pode por vezes ser verboso.

```
PreparedStatement relacionados = null;
ResultSet resultados = null;

try {
    relacionados =
        con.prepareStatement("SELECT ...");
    relacionados.setInt(1, produto);

    resultados = relacionados.executeQuery();

    i = 0;
    while(resultados.next()){
        thumbnail_produtos[i] = resultados.getObject("thumbnail");
        i++;
    }

} catch (SQLException e) {
        ...
}
finally {
    if(resultados!=null){
        resultados.close();
    }
    if(relacionados!=null){
        relacionados.close();
    }
}
```

Thrift Partindo da implementação relacional e adaptando o funcionamento da operação ao novo paradigma, apresenta-se agora o código para a interface de invocação remota Thrift. Um dos pormenores observáveis é que com o uso da interface de utilização nativa do Cassandra somos obrigados a executar dois passos de leitura devido às limitações impostas pela simplicidade da mesma.

Outra das questões pertinentes é a serialização dos diferentes tipos de objectos que são lidos e persistidos na base de dados. De facto, todo o objecto tem de ser convertido num vector de

bytes, e quando lido tem de ser reconvertido para o formato original levando a elevados tempos de programação. Por essa razão, e por uma questão de simplicidade, no seio da plataforma todos os dados são interpretados como objectos Java, sendo serializados como tais. Aquando da leitura, o mesmo processo é utilizado mas de forma inversa e com um simples *cast* recuperamos o valor. Já as chaves são sempre interpretadas como vectores de caracteres.

```
List<String> colunas_a_extrair = new ArrayList<String>();
colunas_a_extrair.add("relacionado1");
colunas_a_extrair.add("relacionado2");
colunas_a_extrair.add("relacionado3");
colunas_a_extrair.add("relacionado4");
colunas_a_extrair.add("relacionado5");

List<ColumnOrSuperColumn> produtos
    = getListColumns(produto_id, "Produto", null ,colunas_a_extrai);

i = 0;

for (ColumnOrSuperColumn colunas : produtos) {
    int produto_relacionado_id  =
        (Integer) Utils.toObject(colunas.getColumn().getValue());
    thumbnail_produtos[i] =
        readfromColumn(Integer.toString(produto_relacionado_id),
                "Produto", "thumbnail");
    i++;
}
```

Embora o código da operação por si não exiba uma grande complexidade, é de notar o uso de métodos de leitura por nós desenvolvidos. Entre estes usa-se o *getListColumns* que retorna uma lista de colunas consoante a lista de nomes dada como argumento. Este método é bastante mais verboso e complexo, pois sendo baseado num cliente de baixo nível é da responsabilidade do programador estabelecer o grau de coerência, o tratamentos das excepções, a desserialização dos dados, etc.

```
private List<ColumnOrSuperColumn> getListColumns( String chave,
                String familia_de_colunas, String super_coluna,
                List<String> colunas_a_extrair) throws Exception {

    ColumnParent parent = new ColumnParent(columnFamily);
    if (super_coluna != null) {
        parent.setSuper_column(super_coluna.getBytes());
    }

    SlicePredicate colunas = new SlicePredicate();
    List<ByteBuffer> id_colunas = new ArrayList<ByteBuffer>();
    for (String coluna_nome : colunas_a_extrair) {
        id_colunas.add(ByteBuffer.wrap(coluna_nome.getBytes());
    }
```

```
    colunas.setColumn_names(id_colunas);

    ByteBuffer chave_bytes = ByteBuffer.wrap(chave.getBytes());

    int tentativas = 0;
    while (tentativas != tentativas_timout) {
        try {
            List<ColumnOrSuperColumn> colunas_extraidas =
                getCassandraClient().get_slice(keyspace,
                    chave_bytes, parent, colunas, READ_CONSISTENCY_LEVEL);
            tentativas = tentativas_timout;
        } catch (TimedOutException e) {
            tentativas++;
            if (tentativas == tentativas_timout) {
                throw e;
            } else {
                Thread.sleep(1000);
            }
        }
    }

    return colunas_extraidas;
}
```

Hector Em seguida mostra-se o código correspondente ao cliente de alto nível Hector. Este cliente oferece já várias vantagens como a gestão do conjunto das conexões à base de dados. Mas a sua principal vantagem é permitir a redução do código necessário para a realização de diversas operações com o Cassandra associadas. O utilizador terá ainda assim de sempre definir os mecanismos de serialização que pretende usar tanto para as chaves como para os nomes de colunas e valores, estando diversas opções disponíveis no cliente. A criação de templates, objectos que descrevem de certo modo cada uma das famílias de colunas seguido o padrão DAO, é também possível para a simplificação do código.

```
    StringSerializer serializador_string = new StringSerializer();

    SliceQuery<String, String, String> instrucao_pesquisa =
        HFactory.createSliceQuery(keyspace, serializador_string,
            serializador_string, serializador_string);
    instrucao_pesquisa.setColumnFamily("Produto")
    .setKey(id_produto)
    .setColumnNames("relacionado1", "relacionado2",
                    "relacionado3", "relacionado4", "relacionado4");

    QueryResult<ColumnSlice<String, Integer>> resultado
        = instrucao_pesquisa.execute();

    int i = 0;

    List<HColumn<String, Integer>> produtos= resultado.get().getColumns();
```

```
for (HColumn<String, Integer> produto : produtos) {
    ColumnQuery<String, String, String> get_coluna =
        HFactory.createColumnQuery(keyspace, serializador_string,
            serializador_string, serializador_string);

    get_coluna.setColumnFamily("Produto")
    .setKey(produto.getValue())
    .setName("thumbnail");

    QueryResult<HColumn<String, String>> coluna_thumbnail =
         get_coluna.execute();

    thumbnail_produtos[i] = coluna_thumbnail.get().getValue();
    i++;
}
```

CQL Como última interface de acesso demonstra-se aqui como podemos usar o CQL, linguagem de consulta recentemente introduzida. De modo similar ao exemplo SQL mostrado, também podemos distinguir aqui uma fase de modelação e outra de adaptação à linguagem Java através do JDBC. Note-se no entanto que esta não possui o mesmo poder de síntese do SQL, pois associado às simplificadas capacidades de pesquisa do Cassandra, também aqui a operação tem de ser partida em dois passos. O CQL permite ainda assim uma mais clara visão e fácil manipulação das operações que estamos a efectuar necessitando porém de uma maior especificação das estruturas de dados na fase de modelação.

Demonstra-se agora a instruções de leitura de todos os produtos associados:

```
SELECT relacionado1, relacionado2, relacionado3,
    relacionado4,relacionado5 FROM Produto WHERE key=?
```

E a leitura da informação sobre os thumbnails

```
SELECT thumbnail FROM Produto WHERE key = ?
```

Após a construção da família de colunas definindo o tipo dos campos utilizados, procedemos à adaptação das instruções modelados através de JDBC. Sendo baseado num driver genérico, este é um processo em tudo similar ao uso de uma solução relacional. Note-se porém que por se encontrar ainda em processo de desenvolvimento, este driver não suporta a substituição de variáveis em instruções genéricas.

```
try{

    PreparedStatement relacionados =
        conn.prepareStatement("SELECT relacionado1,relacionado2,
            relacionado3,relacionado4,relacionado5
            FROM Produto WHERE key="+id_produto+";");
    ResultSet resultado = relacionados.executeQuery();
```

```
        if (resultado.next()) {
            for (int i = 1; i < 6; i++) {

                int key = resultado.getInt(i)

                PreparedStatement ler_thumbnail =
                    conn.prepareStatement("SELECT thumbnail ....);
                ResultSet t_set = ler_thumbnail.executeQuery();
                if (t_set.next()) {
                   thumbnail_produtos[i] = t_set.getString("thumbnail"));
                }
            }

    } catch (Exception ex) {
            ...
    }
```

3.2.3 Operações de escrita

Abordando as operações de escrita, como caso de estudo usa-se aqui a operação de manuseamento do carrinho de compras (*Shopping Cart*). Conforme as variáveis de input, esta operação irá executar operações de adição / remoção de produtos do carrinho, sendo que na sua execução se exige que todas as alterações sejam atómicas. É também da responsabilidade da operação a criação de um novo carrinho se tal não existir. Note-se que sendo a plataforma mais focada nas operações a executar e não tanto no fluxo do *benchmark*, os produtos a adicionar são sempre escolhidos conforme uma distribuição definida e não como resultado das operações de pesquisa como na especificação original. Para a sua implementação em ambos os sistemas, assume-se o uso de instruções/operações base de persistência.

SQL Ignorando o primeiro passo de criação do carrinho de compras, foca-se aqui e nas restante linguagens o segmento que terá de adicionar um novo elemento ou incrementar um já existente. Para tal, a plataforma tentará ler o produto da base de dados e posteriormente executar uma destas operações num bloco de código que será supostamente atómico. Quando modelando esta operação como base uma solução relacional com suporte ACID, podemos de facto responder a esta necessidade de atomicidade das operações agrupando-as numa só transacção.

Representando as instruções modeladas, demonstra-se a seguir como este conjunto de operações seria idealizado numa solução relacional e transaccional.

```
BEGIN TRANSACTION

    SELECT quantidade
        FROM Produtos_no_carrinho_compras
        WHERE id_carrinho = ? AND id_produto = ?

    -- Se retornar um valor, altera-se a quantidade.
```

```
    UPDATE Produtos_no_carrinho_compras
        SET quantidade = ?
        WHERE id_carrinho = ? AND id_produto = ?

    -- Se não retornar um valor, insere-se o produto.

    INSERT into Produtos_no_carrinho_compras
        (id_carrinho, quantidade, id_produto, custo, titulo)
        VALUES (?,?,?,?,?)

END TRANSACTION
```

Esta lógica tem de ser adaptada com o uso de JDBC. Aqui podemos observar como, com a desactivação do mecanismo automático de *commit*, criámos uma transacção onde realizamos as operações de escrita e leitura de forma atómica.

```
con = novaConexao();
con.setAutoCommit(false);

...

ResultSet quantidade_lida = ler_quantidade.executeQuery();

if (quantidade_lida.next()) {
    //O produto já existe no carrinho

    quantidade_a_adicionar +=
        quantidade_lida.getInt("quantidade");

    PreparedStatement  actualizar_produto
      = con.prepareStatement("UPDATE Pro...");
    actualizar_produto.setInt(1, quantidade_a_adicionar);
    actualizar_produto.setInt(2, id_carrinho);
    actualizar_produto.setInt(3, id_produto);
    actualizar_produto.executeUpdate();

}
else{
    //O produto tem de ser adicionado

    //recolha de informação sobre o produto....

    PreparedStatement adicionar_produto
      = con.prepareStatement("INSERT...");

    adicionar_produto.setInt(1, id_carrinho);
    adicionar_produto.setInt(2, quantidade_a_adicionar);
    adicionar_produto.setInt(3, id_produto);
    adicionar_produto.setFloat(4, custo);
    adicionar_produto.setString(6, titulo);
```

```
    adicionar_produto.executeUpdate();
    adicionar_produto.close();
}
con.commit(); // Commit da transacção

//Tratamento de excepções
//Fecho de conexões e instruções.
```

Thrift Entrando agora no domínio do Cassandra, e ainda antes da análise da interface Thrift em si, temos de primeiro observar os requisitos impostos pela especificação. Como constatado acima, quando um produto é adicionado ao carrinho, para evitar adições concorrentes tal operação (leitura e escrita do carrinho) deveria ser atómica. A necessidade de tal garantia é no entanto discutível. Sendo esta uma operação que se destina ao cliente e sendo ele o único que sobre o carrinho pode efectuar operações, é altamente improvável que exista algum tipo de concorrência que a justifique.

Esta operação é assim implementada em Cassandra sem esta garantia. Veja-se que mesmo a nível individual, sobre nenhum dos pedidos enviados à base de dados existem garantias que este será executado em todos os nós, sendo que se não se atingir o critério de coerência, o novo valor permanecerá como válido nos nós que o escreveram. No entanto dado à natureza idempotente das escritas, estas podem ser repetidas em caso de falha. Estratégias como as usadas pela Amazon no seu sistema Dynamo são também aqui aplicáveis [9].

Em termos de isolamento, também nenhuma garantia é dada em termos globais ou ao nível do nó. As falhas deste tipo terão de ser colmatadas do lado do cliente, filtrando casos como carrinhos incompletos ou produtos que ainda não contêm todos os seus campos preenchidos, caso exista alguma situação que o justifique.

Passando agora ao código Thrift, temos mais uma vez o código da operação que em si não apresenta grande complexidade, sendo até mais simples do que a versão relacional. Uma vez que não existe diferença no Cassandra entre inserir ou actualizar um produto na base de dados é usado o mesmo método em ambos os casos, variando somente a colunas inseridas.

```
Object o =
    readfromSuperColumn(id_carrinho, "Carrinho_Compras",
         "quantidade", id_produto);

Map<String,Object> colunas_a_inserir = new TreeMap<String,Object>();

if (o != null) {
    quantidade_lida = (Integer) o;
    quantidade_a_inserir += quantidade_lida;

}else{

    //recolha de informação sobre o produto....

    colunas_a_inserir.add("custo",custo)
    colunas_a_inserir.add("titulo",titulo)
}
```

```
    colunas_a_inserir.add("quantidade",quantidade_a_inserir)

    super_batch_mutate(id_carrinho, "Carrinho_Compras", colunas_a_inserir);
```

Novamente, note-se que a complexidade da operação é na verdade reduzida graças ao uso de métodos genéricos como o último dos métodos usados. Este, que a seguir apresentamos, trata-se de um método de envio de conjuntos de operações de escritas ("batchs") para a inserção de uma só super coluna. Na sua totalidade, podemos ver que várias das características da base de dados transparecem no código, como a marca temporal que segue com os pedidos, característica do mecanismo de resolução do Cassandra, ou tratamento das excepções para a repetição da operação em caso de falha. Como é visível, cada um destes métodos, embora genérico e de fácil reutilização, provêm de um longo processo de programação e teste.

```
public void super_batch_mutate(String chave, String chave_super_coluna,
    String familia_colunas, Map<String,Object> colunas_a_inserir)
    throws Exception {

    List<Column> sub_colunas_a_inserir = new ArrayList<Column>();

    long marca_temporal = System.currentTimeMillis();

    for (Map.Entry<String, Object> coluna_a_inserir
         : colunas_a_inserir.entrySet()) {

        ByteBuffer nome
            = ByteBuffer.wrap(coluna_a_inserir.getKey().getBytes());
        ByteBuffer valor
            = ByteBuffer.wrap(Utils.getBytes(coluna_a_inserir.getValue()));

        Column coluna = new Column(nome, valor, marca_temporal);

        sub_colunas_a_inserir.add(coluna);
    }

    ByteBuffer chave_super_coluna_bytes
        = ByteBuffer.wrap(chave_super_coluna.getBytes());

    SuperColumn super_coluna
      = new SuperColumn(chave_super_coluna_bytes, sub_colunas_a_inserir);

    List<Mutation> lista_mutacoes = new ArrayList<Mutation>();

    ColumnOrSuperColumn produto = new ColumnOrSuperColumn();
    produto.setSuper_column(super_coluna);
    Mutation mutacao = new Mutation();
    mutacao.setColumn_or_supercolumn(produto);
    lista_mutacoes.add(mutacao);

    Map<String, List<Mutation>> mutacoes
        = new TreeMap<String, List<Mutation>>();
    mutacoes.put(familia_colunas, lista_mutacoes);
```

```
    Map<ByteBuffer, Map<String, List<Mutation>>> mapa_de_mutacoes
        = new TreeMap<ByteBuffer, Map<String, List<Mutation>>>();
    ByteBuffer chave_bytes = ByteBuffer.wrap(chave.getBytes());

    mapa_de_mutacoes.put(chave_bytes, mutacoes);

    Cassandra.Client cliente = getClient();

    int tentativas = 0;
    while (tentativas != tentativas_erro) {

        try {
            cliente.batch_mutate(mapa_de_mutacoes,WRITE_CONSISTENCY_LEVEL);
            tentativas = tentativas_erro;
        } catch (TimedOutException e) {
            tentativas_erro++;
            if (tentativas == tentativas_erro) {
                throw e;
            } else {
                Thread.sleep(1000);
            }
        } catch( UnavailableException e){
            tentativas_erro++;
            if (tentativas == tentativas_erro) {
                throw e;
            } else {
                 cliente = getOtherClient();
            }
        }
    }

}
```

Hector Passando agora para o Hector, de modo similar ao observado nas operações de leitura, esta biblioteca permite ao programador uma diminuição do código pois efectua de forma automática muitas das acções subjacentes ao processo de escrita na base de dados. Tomando como exemplo o funcionamento em caso de falha de uma operação, este cliente permite especificar se o erro será retornado de imediato ou se a operação deverá ser tentada em um ou mais nós.

No extracto de código a seguir demonstra-se como com a criação de um *template*, se torna fácil ao programador a extracção e escrita de informação na super coluna correspondente ao produto. Sobressai ainda assim a constante definição operadores de serialização tanto para as chaves como para nomes de colunas e nomes de super colunas.

```
    SuperCfTemplate<String, String,String> template =
        new ThriftSuperCfTemplate<String,String,String>(keyspace,
            "Carrinho_Compras",
            StringSerializer.get(),
            StringSerializer.get(),
            StringSerializer.get());
```

```
SuperCfUpdater<String, String,String> updater
    = template.createUpdater(id_carrinho,id_produto);

try {
    SuperCfResult<String, String,String> leitor_quantidade
        = template.querySuperColumn(id_carrinho,id_produto);

    Integer quantidade_lida = leitor_quantidade.getInteger("quantidade");

    if(quantidade_lida!=null){

        quantidade_a_inserir += quantidade_lida;

    }
    else{

        //recolha de informação sobre o produto....

        updater.setString("titulo","titulo");
        updater.setDouble("custo",2.3);

    }

    updater.setInteger("quantidade",quantidade_a_inserir);

    template.update(updater);

} catch (HectorException e) {
        ...
}
```

CQL Por último temos a linguagem de interrogação CQL. Neste ponto, como podemos ver pelo código apresentado, as abordagens relacional e não relacional acabam por se confundir devido similaridade entre as instruções dos dois lados usados. Existe no entanto uma diferença entre estas e as outras abordagens em Cassandra. Devido à tendência actual de afastamento das famílias de super colunas, não existe na especificação do CQL suporte para este género de estruturas.

Para a implementação com base em CQL supomos, por esta razão, a existência de uma família de colunas onde os produtos são armazenados com uma chave que contém o identificador do carrinho mais o do produto. Esta operação seria codificada por:

```
SELECT quantidade
    FROM Carrinho_Compras
    WHERE key= ?

-- Se retornar um valor, altera-se a quantidade.

UPDATE Carrinho_Compras
    SET quantidade = ?
```

```
        WHERE key= ?

    -- Se não retornar um valor, insere-se o produto.

    INSERT INTO Carrinho_Compras (key, quantidade, custo, titulo)
        VALUES (?,?,?,?)
```

Em termos de adaptação com base em JDBC, comprova-se outra vez a similaridade entre as soluções à excepção do algoritmo associado à chave, pois não podemos em Cassandra identificar uma linha por uma coluna sem a indexarmos.

```
...

ResultSet quantidade_lida = ler_quantidade.executeQuery();

if (quantidade_lida.next()) {
    //O produto já existe no carrinho

    int quantidade_existente =
        quantidade_lida.getInt("quantidade");
    quantidade_existente += quantidade_a_adicionar;

    String chave = id_carrinho+"."+id_produto;

    PreparedStatement  actualizar_produto
      = con.prepareStatement(benchmark_queries.get(
      "UPDATE Carrinho_Compras
        SET quantidade = "+quantidade_a_adicionar+"
        WHERE key= "+ chave +")
        );

    actualizar_produto.executeUpdate();

}
else{
    //O produto tem de ser adicionado

    //recolha de informação sobre o produto....

    String chave = id_carrinho+"."+id_produto;

    PreparedStatement adicionar_produto
      = con.prepareStatement(benchmark_queries.get("INSERT..."));

    adicionar_produto.close();
}

//Tratamento de excepções
//Fecho de conexões e instruções.
```

3.2.4 Operações de leitura sequencial

Como último caso de estudo sobre as diversas interfaces de acesso e as suas diferenças focamos agora a nossa atenção sobre as operações de leitura sequencial. Usando como exemplo a operação de administração do stock (*Admin comfirm*) esta é uma das operações onde o poder de síntese do SQL mais se destaca em comparação com as interfaces do Cassandra.

Nesta operação o administrador do sistema irá sobre um determinado produto recolher as últimas encomendas onde este figura. Com esta informação, ele pode determinar quais os produtos que foram mais vezes com este adquirido, criando uma lista de 5 produtos relacionados. Esta é na verdade uma operação extremamente custosa pois requer a recolha de informação sobre as 10 000 encomendas executadas. Se numa base de dados relacional tal operação pode ser expressa com uma só instrução de SQL ficando ao cargo do motor da base de dados a sua execução de forma eficiente, tal não é possível numa base de dados não relacional como o Cassandra. Assim, como veremos pelas diversas interfaces expostas, as dez mil encomendas terão de ser na sua totalidade transferidas para o lado do cliente e sua análise terá de ser feita pelo programador.

Este procedimento fará esta operação ter um desempenho claramente inferior quando comparado ao modelo relacional. Este não se trata no entanto do tipo de operação que esperaríamos encontrar implementada sobre Cassandra. De facto, esta é uma base de dados mais direccionada a manipulações simples sobre grandes quantidades de dados, isto quando falamos em aplicações com necessidade de operações em tempo real. Para operações de gestão como esta, existe a possibilidade de usarmos a interface Hadoop do Cassandra[3] sobre a qual foi recentemente desenvolvida a plataforma Brisk[4].

SQL Em primeiro lugar a instrução de leitura percorre entre todas as encomendas as 10 000 mais recentes e identifica, usando a tabela que os codifica, os produtos a que elas estão associadas. Sobre estes são identificados aqueles que foram comprados em conjunto com o produto em causa, retornando-se os cinco elementos de maior ocorrência. As restantes instruções utilizadas na operação são simples escritas para actualizar o produto com a nova informação, que aqui ignoramos.

```
SELECT Produtos_encomendados.id_produto
 FROM Encomendas, Produtos_encomendados
  WHERE Encomendas.id = Produtos_encomendados.id_encomenda
  AND NOT (Produtos_encomendados.id_produto = ? )
  AND Encomendas.id_cliente IN (
   SELECT id_cliente
    FROM Encomendas, Produtos_encomendados
     WHERE Encomendas.id = Produtos_encomendados.id_encomenda
     AND Encomendas.id > (SELECT MAX(id)-10000 FROM Encomendas)
     AND Produtos_encomendados.id_produto = ?
  )
 GROUP BY id_produto
```

[3] http://wiki.apache.org/cassandra/HadoopSupport
[4] http://www.datastax.com/docs/0.8/brisk/index

```
ORDER BY SUM(quantidade) DESC
LIMIT 5
```

Este é de facto uma instrução complexa que para além da simples pesquisa de informação realiza a filtragem dos resultados deixando pouco mais para fazer do lado do cliente.

```
...

PreparedStatement relacionados
    = connection.prepareStatement("SELECT...");

// Set parameter
relacionados.setInt(1, id_produto);
relacionados.setInt(2, id_produto);
ResultSet resultados = relacionados.executeQuery();

int[] produtos_relacionados = new int[5];

int i =0;
while (resultados.next()) {
    produtos_relacionados[i] = resultados.getInt(1);
    i++;
}

\\ Alteração da informação do produto
```

Thrift Codificando esta operação com a biblioteca de acesso Thrift, temos de igual modo de pesquisar as 10 000 últimas encomendas, sendo estas retornadas no seu total para o cliente. Tirando partido da forma como as encomendas são persistidas na sua família de colunas, onde as mais recentes estão sempre no inicio, esta operação é aqui realizada através de uma leitura sequencial sobre esta estrutura. No método de pesquisa o programador terá depois de codificar a selecção e ordenação dos pedidos para determinar os cinco produtos relacionados.

Esta migração de código para o lado do cliente é um dos pontos marcantes de muitas das soluções não relacionais que nos últimos anos surgiram. Devido à simples API, estas soluções implicam que toda a lógica aos dados associados seja transcrita na estrutura dos dados ou no cliente de acesso à base de dados.

```
Map<String, Map<String, Map<String, Object>>> encomendas =
  super_rangeQuery("Encomendas", null, 10000);

Map<Integer, Integer> informacao_produtos
  = new TreeMap<Integer, Integer>();

for (Map<String, Map<String, Object>> info_encomendas
  : encomendas.values()) {
```

```
  boolean comprado_em_conjunto = false;
  TreeMap<Integer, Integer> produtos_adquiridos
    = new TreeMap<Integer, Integer>();
  for (Map.Entry<String, Map<String, Object>> encomenda
    : info_encomendas.entrySet()) {

    String nome_super_coluna = encomenda.getKey();

//Se a super coluna não for a que contém a informação sobre a encomenda
    if (!nome_super_coluna.equals("informacao")) {
      Map<String, Object> colunas = encomenda.getValue();
      int id_produto = (Integer) colunas.get("id_produto");
      if (id_produto == produto_a_testar) {

        //nesta encomenda o cliente comprou o produto
        comprado_em_conjunto = true;

      } else {

        //guardar a quantidade adquirida
        int quantidade = (Integer) colunas.get("quantidade");
        produtos_adquiridos.put(id_produto, quantidade);

      }
    }
  }

  // se comprado em conjunto guarda-se a informação,
  // caso contrario descarta-se
  if (comprado_em_conjunto == true) {
    for (Integer id_produto : produtos_adquiridos.keySet()) {
      if (informacao_produtos.containsKey(id_produto)) {
        int quantidade = informacao_produtos.get(id_produto);
        informacao_produtos.put(id_produto,
          (produtos_adquiridos.get(id_produto) + quantidade));
      } else {
        informacao_produtos.put(id_produto,
          produtos_adquiridos.get(id_produto));
      }
    }
  }
}

//Ordenação para obter os mais vendidos
Map mais_vendidos = reverseSortByValue(informacao_produtos);

int i = 0;
for (Iterator<Integer> produtos = mais_vendidos.keySet().iterator()
    ; produtos.hasNext(); )
{
  produtos_relacionados[i] = produtos.next();
  if (i == 4)
    break;
```

```
    i++;
  }
```

Sendo já relativamente complexo, este método é simplificado por um método de pesquisa sequencial dos dados. Este é um método complexo onde se destaca a forma repartida como faz a pesquisa dos dados, por uma questão de preservação de memória, e também o modo como as operações são repetidas em caso de erro por espera demorada.

```
public Map<String, Map<String, Map<String, Object>>> super_rangeQuery(
        String familia_colunas, List<String> campos, int limite)
        throws Exception {

  Map<String, Map<String, Map<String, Object>>> resultados
    = new TreeMap<String, Map<String, Map<String, Object>>>();

  SlicePredicate predicate = new SlicePredicate();

  if (campos == null) {

    SliceRange range =
      new SliceRange(ByteBuffer.wrap("".getBytes()),
        ByteBuffer.wrap("".getBytes()), false, 2000);
    predicate.setSlice_range(range);

  } else {

    List<ByteBuffer> campos_a_extrair = new ArrayList<ByteBuffer>();
    for (String campo : campos) {
      campos_a_extrair.add(ByteBuffer.wrap(campo.getBytes()));
    }
    predicate.setColumn_names(campos_a_extrair);
  }

  KeyRange range = new KeyRange();
  range.setStart_key("".getBytes());
  range.setEnd_key("".getBytes());
  //partiçao_de_pesquisa = número de linhas a pesquisar em cada iteração
  range.setCount(partiçao_de_pesquisa);

  ColumnParent parent = new ColumnParent();
  parent.setColumn_family(familia_colunas);

  boolean terminado = false;
  limite = (limite < 0) ? -1 : limite;

  int numero_chaves = 0;

  while (!terminado) {

      Cassandra.Client cliente = getClient();
```

```
  int tentativas = 0;

List<KeySlice> linhas  = null;

while (tentativas != tentativas_erro) {

  try {
    linhas =
      cliente.get_range_slices(parent
        , predicate, range, RANGE_CONSISTENCY_LEVEL);
    tentativas = tentativas_erro;
  } catch (TimedOutException e) {
    tentativas_erro++;
    if (tentativas == tentativas_erro) {
      throw e;
    } else {
      Thread.sleep(1000);
    }
  }
}

if (!linhas.isEmpty()) {
  range.setStart_key(linhas.get(linhas.size()-1).getKey());
}

for (KeySlice linha : linhas) {
  if (!linha.columns.isEmpty()) {
    Map<String, Map<String, Object>> campos_retornados =
      new TreeMap<String, Map<String, Object>>();
    resultados.put(new String(linha.getKey()), campos_retornados);
    for (ColumnOrSuperColumn c : linha.getColumns()) {
      if (c.isSetSuper_column()) {
        if (!c.getSuper_column().columns.isEmpty()) {

          Map<String, Object> colunas =
            new TreeMap<String, Object>();

          for (Column coluna : c.getSuper_column().columns) {
            String nome_coluna = new String(coluna.getName());
            Object valor = Utils.toObject(coluna.getValue());
            colunas.put(nome_coluna, valor);
          }
          String nome_super_coluna =
            new String(c.getSuper_column().getName());
          campos_retornados.put(nome_super_coluna, colunas);
        }
      }
    }
    numero_chaves++;
  }
  if (numero_chaves >= limite && limite != -1) {
    terminado = true;
```

```
        break;
      }
    }
    if (linhas.size() < partiçao_de_pesquisa) {
      terminado = true;
    }
  }

  return resultados;
}
```

Hector Abordando agora o cliente de alto nível Hector, este fornece neste caso o método de pesquisa sobre Cassandra, permitindo assim uma diminuição do código base. O mesmo não se aplica no entanto ao código de análise de resultados, pois este advém das características subjacentes ao Cassandra.

```
RangeSuperSlicesQuery<String,String,String,Integer> pesquisa_encomendas
  = HFactory.createRangeSuperSlicesQuery(keyspace,
    StringSerializer.get(),
    StringSerializer.get(),
    StringSerializer.get(),
    IntegerSerializer.get());

pesquisa_encomendas.setColumnFamily("Encomendas");
pesquisa_encomendas.setRange("", "", false, 10000);
QueryResult<OrderedSuperRows<String, String, String, Integer>> resultado
  = pesquisa_encomendas.execute();

List<SuperRow<String,String,String,Integer>> encomendas
  = resultado.get().getList();

Map<Integer, Integer> informacao_produtos
      = new TreeMap<Integer, Integer>();

for (SuperRow<String,String,String,Integer> info_encomendas : encomendas){

  boolean comprado_em_conjunto = false;
  TreeMap<Integer, Integer> produtos_adquiridos
    = new TreeMap<Integer, Integer>();

  for (HSuperColumn<String,String,Integer> encomenda
    : info_encomendas.getSuperSlice().getSuperColumns()) {

    String nome_super_coluna = encomenda.getName();

//Se a super coluna não for a que contém a informação sobre a encomenda
    if (!nome_super_coluna.equals("informacao")) {
      List<HColumn<String,Integer>> colunas = encomenda.getColumns();

      int id_produto = -1;
      int quantidade = -1;
```

```
        for (HColumn<String, Integer> coluna : colunas) {
          if(coluna.getName().equals("id_produto")){
            id_produto = coluna.getValue();
          }
          if(coluna.getName().equals("quantidade")){
             quantidade = coluna.getValue();
          }
        }

        if (id_produto == 90) {

          comprado_em_conjunto = true;

        } else {

          produtos_adquiridos.put(id_produto, quantidade);

        }
      }
    }

    // se comprado em conjunto guarda-se a informação,
    // caso contrario descarta-se
    if (comprado_em_conjunto == true) {
      ... //Restante código de análise
```

CQL Como exemplo final, temos o modo como em CQL podemos fazer uma leitura sequencial da informação. Assumindo novamente uma família de colunas normal que permita a execução nesta linguagem, supõem-se que esta terá a cada chave associadas vários produtos encomendados. Para a extracção das dez mil unidades temos a instrução a seguir mostrada.

```
SELECT * FROM Encomendas LIMIT 10000
```

Na verdade, para além da maior clareza que a linguagem introduz nas operações executadas e o facto de permitir a abstracção dos mecanismos de serialização, poucas mais são aqui as vantagens inerentes à linguagem. Como vemos abaixo o código de análise permanece similar aos casos anteriores.

```
    PreparedStatement instrucao_leitura = conn.prepareStatement("SELECT...");
    ResultSet set = instrucao_leitura.executeQuery();

    Map<String,Map<String,Map<String,Integer>>> encomendas
        = new TreeMap<String, Map<String, Map<String, Integer>>>();

    while (set.next()) {
```

```
        String encomenda = set.getString("KEY");

        List<String> nomes_colunas  =getColumnNames(set);
            for (String nome_coluna : nomes_colunas) {
              //analisar os nomes e construir o mapa de produtos comprados
            }
    }


    Map<Integer, Integer> informacao_produtos
        = new TreeMap<Integer, Integer>();

    for (Map<String, Map<String, Object>> info_encomendas
        : encomendas.values()) {
        boolean comprado_em_conjunto = false;
        TreeMap<Integer, Integer> produtos_adquiridos
            = new TreeMap<Integer, Integer>();
        for (Map.Entry<String, Map<String, Object>> encomenda
            : info_encomendas.entrySet()) {
        ...
```

Em resumo das três secções de análise de operações, conclui-se que mesmo com o acréscimo do código JDBC, a versão relacional é a no geral a mais concisa. Isto deve-se às características do SQL que permite, na extracção de informação, exprimir relações entre diferentes elementos de uma mesma tabela ou mesmo entra várias destas estruturas. Como uma interface mais simplista, o código sobre Cassandra acaba por se tornar mais complexo pois uma parte da lógica transita para o lado do cliente. Ainda assim demonstra-se que vários são os níveis de abstracção que o programador tem para escolha quando desenvolvendo uma aplicação sobre Cassandra

3.2.5 Outras operações

Indo além da simples interfaces de acesso, é também visível nas operações anteriores os vários passos de adaptação que as operações tiveram de sofrer para serem implementadas em Cassandra. Estas não serão as únicas naturalmente, uma vez que descartando operações simples, como a leitura de informação sobre um produto (*Product Detail*), todas elas tiveram de ser submetidas a algum tipo mudança. Além da já referida ordenação das encomendas para uma melhor pesquisa e do uso de chaves complexas na persistência de endereços para evitar duplicação de informação, outros passos de transição foram tomados.

Para a implementação da operação que determina quais os produtos mais recentes (*New Products*), e de modo similar nas pedidos de pesquisa por tema ou autor (*Search*), foram introduzidas na base de dados novas famílias de colunas onde para cada um deste campos armazenamos a informação que o cliente pretende ver. A operação de obtenção dos produtos mais recentes tem no entanto uma particularidade, pois as colunas que correspondem aos novos produtos são aqui persistidas com uma marca temporal no seu nome garantido assim que os primeiros elementos são os mais recentes. Mas não só em termos de implementação esta operação foi estudada, pois existe entre ela a operação de administração de produtos, que acima apresentamos, um possível

problema de concorrência. De facto cada vez que um produto é alterado, este deve ser também modificado na família que os ordena para a pesquisa. Existindo aqui uma especificada necessidade de atomicidade entre estas operações, tal não é possível garantir em Cassandra. É no entanto discutível se a experiência do cliente fica prejudicada por não ver na lista dos últimos adicionados, um livro que foi há momentos alterado.

O mesmo não se pode afirmar sobre as operações de compra (*Buy Confirm*), pois estas implicam alterações concorrentes ao stock dos diversos produtos tornando difícil a sua conversão para Cassandra. Ainda assim, em situações de stock relativamente abundante, um controlo rígido do mesmo pode não ser necessário na venda de produtos ao público. Garantido a persistência de cada uma das encomendas, o conhecimento exacto de qual o stock de um produto pode ser calculado em todos os momentos, sendo os valores presentes nestas entidades usados para que nas operações de venda exista uma ideia geral de qual o valor do mesmo.

3.3 Avaliação da coerência

O Cassandra baseia-se num modelo de coerência dinâmico baseado na ordenação temporal dos pedidos submetidos e no número de nós que cada operação recorre para confirmar os seus resultados. Assim dois ou mais nós podem de facto divergir entre si sendo que inevitavelmente os dados serão reconciliados com o uso das marcas temporais que a cada um dos tuplos estão associadas.

Este factor adicionado à ausência de garantias transaccionais coloca um entrave à implementação de operações como a compra de um produto que se encontram especificadas no TPC-W. Levanta-se no entanto a questão sobre a real necessidade de tais garantias. De facto, não são estranhas ao cliente situações, onde grandes revendedores como a Amazon permitem a encomenda de produtos sobre stock que entretanto esgotou.

Assim, mesmo sobre uma base de dados onde se aplica a noção de *eventual consistency*, tais operações podem ser implementadas se o número de casos de incoerência se provar reduzido e controlável. Na prática, em casos como o apresentado, pequenas incoerências no controlo do stock têm pouco significado quando o seu valor é diminuto, podendo ser depois facilmente detectadas através de um registo das vendas efectuadas. Casos que acabem por passar para o lado do cliente do sistema, poderão depois ser tratados com mecanismos de desculpa e/ou compensação [10]. Este mecanismo baseia-se no entanto na suposição que as incoerências no stock nunca serão elevados e por isso realiza-se aqui uma avaliação quantitativa da mesma.

Descreve-se agora o primeiro dos casos de uso implementados. Executado maioritariamente sobre Cassandra, este tem como objectivo a avaliação da ausência de garantias transaccionais nesta solução. Como fruto desta ausência analisa-se assim a consequente incoerência que pode surgir nos dados derivada de conjuntos de escritas e leituras concorrentes, sendo testados também como os mecanismos de coerência afectam tais resultados.

Com este intuito específico introduzem-se alterações nas operações utilizadas, onde algumas são assim simplificadas e outras alteradas. Para tal é formado um novo *workload* centrado na execução de ciclos de instruções a seguir e aos quais damos o nome de ciclos de compra. Estes ciclos centrados nas operações de observação de um produto, na sua adição a um carrinho e na sua compra são deste modo criados:

1. Determina-se o número de produtos a adicionar ao carrinho de compras, sendo este um

número aleatório entre 1 e 10.

2. Para cada produto a adicionar:
   (a) Entre os produtos disponíveis, é escolhido um através de uma distribuição estatística definida pelo utilizador da plataforma.
   (b) Para este item é consultada a sua informação onde se inclui o stock
   (c) É gerada aleatoriamente a quantidade que irá ser adicionada ao carrinho. Este número será maior que 1 e menor que o mínimo entre 5 e o stock lido.
   (d) O produto é adicionado ao carrinho, que irá apenas conter a informação base do mesmo descartando-se os restantes dados.
3. O carrinho é sujeito a uma compra onde para cada item:
   (a) É analisado novamente o stock do produto.
   (b) Caso o stock lido seja menor que a quantidade a comprar, então esta operação falha.
   (c) Caso o stock lido seja maior que a quantidade a comprar, então este produto é adquirido.

Com estes ciclos pretende-se medir até que ponto a escrita concorrente das actualizações do stock provocam incoerência nos dados. Tal situação acontece sempre que duas operações gravam o decremento do stock de um determinado produto de forma concorrente, pois ambas acabaram de o comprar. Quando tal acontece, existe assim o registo de duas compras efectuadas com sucesso, mas apenas uma delas se reflecte no novo stock.

Para a avaliação deste fenómeno precisamos no entanto de obter quais as quantidades vendidas, tal como uma medida real do stock gasto durante a execução. Para este efeito existem estruturas auxiliares que irão armazenar qual o valor inicial (SI) e final do stock (SF) de cada produto permitindo assim obter o valor do stock gasto (SG) definido como:

```
SG = SI - SF
```

São também armazenadas as unidades compradas (C), sendo que a diferença entre este valor e o stock gasto nos permite visualizar qual a quantidade de comprada cujo o valor não foi descontado na base de dados (I) através de:

```
I = SG - C
```

Na sua execução o utilizador terá numa primeira fase de estipular qual a distribuição estatística a utilizar para a escolha dos produtos a comprar. Esta distribuição é responsável pelo padrão de acesso aos produtos, podendo gerar cenários onde todos têm igual probabilidade de serem escolhidos ou onde uns serão muito mais requisitados que outros. Encontram-se deste modo implementadas no sistema uma distribuição uniforme e uma *power-law*, fornecendo-se também uma interface genérica que permite a adição de outras opções.

Outro facto importante é o stock inicial dos produtos, que terá de ser o suficiente para não acabar a meio do *benchmark*. Poderiam ser realizadas adições ao stock dos produtos durante a execução do sistema, mas isso levaria ao aumento das escritas concorrentes, tornando a análise dos resultados mais complexa.

Ambiente de execução

Para a execução deste benchmark aqui definido são usadas duas máquinas de teste sobre um *cluster* de cinco máquinas. O hardware consiste em sete máquinas HP com um processador Intel(R) Core(TM)2 CPU 6400 - 2.13GHz, dois Gbyte de RAM e um disco rígido de uso comum SATA (7200 RPM). Todas as máquinas estão ligadas por LAN conectadas a um *switch* de 1GB/s. O sistema operativo é Linux, Ubuntu Server com kernel 2.6.31-1 e o sistema de ficheiros encontra-se em ext4. Para estes testes, realizados numa primeira fase deste trabalho, a versão de Cassandra usada é a 0.6.6 significando assim que não existem um suporte a índices por parte da base de dados. Para o teste comparativo em MySQL, o testes são executados numa só maquina com a versão 5.1.54 desta base de dados, sendo o nível de isolamento transaccional usado o *Read Uncommitted*.

Em termos de parâmetros de execução, existem 100 (50x2) clientes concorrentes em duas máquinas individuais. Cada um deste efectua 500 pedidos à base de dados de 10 000 produtos. Para estes testes, os factores que variamos na sua execução prendem-se com o nível de coerência usado nas instruções do Cassandra, o factor de escolha dos produtos e o tempo de espera entre operações. No decorrer dos testes varia-se o nível de coerência entre um nó e quórum destes, sendo que nestes dois cenários são depois utilizados diferentes tempos de espera e padrões de acesso aos produtos. Em termos de combinações temos assim 8 cenários de execução sendo que os 4 base que são depois executados com diferentes níveis de coerência resumem-se a:

1. Tempo de espera entre operações segundo a especificação do TPCW (0-70s) e escolha uniforme dos produtos para compra.
2. Tempo de espera entre operações segundo a especificação do TPCW (0-70s) e compra dos produtos existentes segundo uma distribuição *power-law*.
3. Tempo de espera entre operações fixo em 1 segundo e escolha uniforme dos produtos para compra.
4. Tempo de espera entre operações fixo em 1 segundo e compra dos produtos existentes segundo uma distribuição *power-law*.

Em Cassandra cada um destes cenários é executado 5 vezes com uma sincronização dos relógios entre as máquinas aquando do inicio de cada ronda.

Análise dos resultados

Numa fase anterior à execução dos testes descritos sobre Cassandra, e como comparação, fizemos também esta mesma experiência num nó de MySQL. Escolhendo o mais agressivo dos cenários de teste apresentado, onde a escolha dos produtos segue uma distribuição *power-law* e número de operações médias é de cerca de 20 por segundo através de um tempo de espera entre pedidos de

Tempo de espera	Factor de selecção	Total comprado (média)	Incoerência no stock (média)	Produtos afectados (média)
TPC-W	Uniforme	67596.2	12.8	4.2
TPC-W	Power Law	67482.8	56.6	17.8
1s	Uniforme	67570.6	568.6	187.0
1s	Power Law	67440.8	1183.2	351.6

Tabela 3.1: Vendas de produtos e incoerências no stock - nível de coerência baseado num só nó

1 segundo[5], as incoerências de stock não afectaram mais do que seis produtos numa soma total que não foi além das 20 unidades. Este é um valor claramente reduzido e que funciona como base de comparação, pois num só nó e com uma ordenação não temporal dos pedidos (ao contrário do Cassandra), aqui a incoerência gerada resulta unicamente da concorrência entre os pedidos efectuados.

Analisando agora os resultados em Cassandra, temos numa primeira fase de execução onde para cada pedido apenas se requisita a resposta de um nó, os resultados que se encontram resumidos na Tabela 3.1, onde vemos as vendas e as respectivas incoerências no stock. Temos também na Figura 3.4 uma das rondas de execução para cada um dos cenários propostos, onde se mostra esta mesma informação para os diversos produtos em conjuntos de 100.

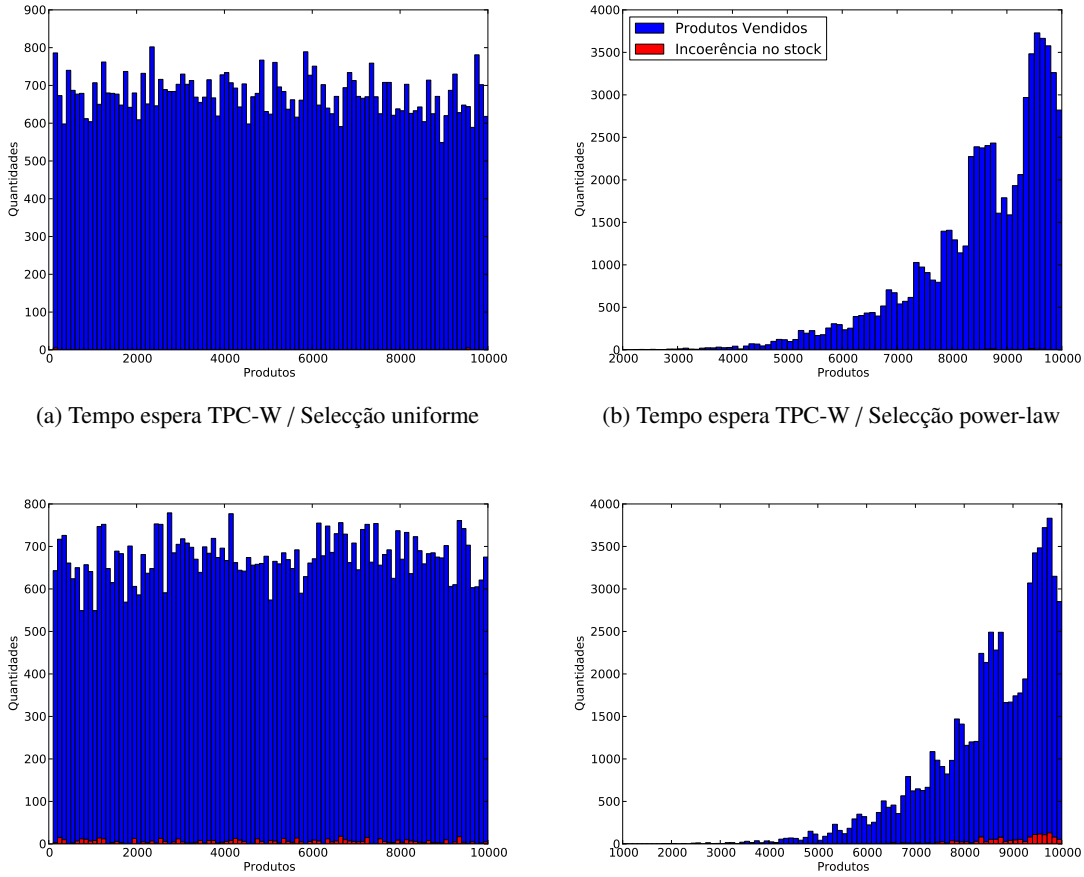
Inicialmente com um tempo de espera que segue a especificação dada pelo TPC-W e que se traduz numa média de 2.5 operações de compra por segundo, vemos em 3.4a e 3.4b que as incoerências no stock são diminutas sendo de difícil detecção. O cenário piora ligeiramente quando aumentamos a concorrência através da diminuição do tempo de espera para 1 segundo. De facto mesmo quando acedemos aos objectos de forma uniforme (Figura 3.4c) existe dezenas de produtos incoerentes, sendo que alterando o padrão de acesso (Figura 3.4d), este valor aumenta.

Analisando agora os resultados com um nível de coerência maior, vemos na Tabela 3.2 que este factor acaba não ter impacto no stock. De facto este factor não tem durante o normal funcionamento de todos os nós um papel significativo. Este nível refere-se nas escritas não ao número de nós que realizam a operação, mas sim aos que a confirmam não afectando assim os resultados. Como aqui se mede o efeito de concorrência entres os diferentes clientes, este factor poderia ainda assim aumentar o tempo de execução das operações e aumentar também a janela de concorrência entre compras de um mesmo produto. Tal não se verifica no entanto nos testes realizados. Estes níveis desempenham sim um importante papel no fornecimento de garantias aquando da execução das operações durante situações de falha do sistema.

Nos gráficos que retratam uma das rondas exemplo, quer para os cenários com tempo de espera baseado no TPC-W (Figura 3.5a e 3.5b) quer nos cenários de maior concorrência (Figura 3.5c e 3.5d) é assim visível a similaridade dos resultados com o caso anterior.

Note-se no entanto a discrepância nos resultados quando comparados com os primeiramente obtidos com base em MySQL. Tal cenário não pode ser simplesmente explicado com o número de nós que constituem o Cassandra ou o sistema de gestão transaccional do MySQL. Tal discrepância é sim resultante da dessincronização dos relógios entre as máquinas de testes. De facto como posteriormente constatamos, as máquinas de teste chegavam ter uma variação nos relógio de um milissegundo a cada 3 segundos.

[5]Como comparação, o produto mais vendido de sempre na Amazon atingiu sozinho o valor de 158 vendas por segundo (http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=1510745).



(a) Tempo espera TPC-W / Selecção uniforme

(b) Tempo espera TPC-W / Selecção power-law

(c) Tempo espera 1s / Selecção uniforme

(d) Tempo espera 1s/ Selecção power-law

Figura 3.4: Vendas de produtos e incoerências no stock - nível de coerência baseado num só nó

Demonstra-se assim que a ordenação temporal dos pedidos pode ser problemática em Cassandra. Por esta razão repetimos os testes para os cenários de maior concorrência em paralelo com um processo de sincronização baseado em pedidos *ntpdate* a um servidor de *ntp* local. Tal medida atenua a divergência temporal das máquinas e assim evita-se que escritas feitas por uma máquina com uma marca temporal já vários milissegundos avançada no tempo impeçam que posteriores escritas realizadas por outros clientes sejam persistidas até que o relógio destes atinja tal tempo.

Como é visível pelos resultados presentes na Tabela 3.3, o número de incoerências no stock, tal como número de produtos por ela afectada é reduzido drasticamente. Existem em média 27 produtos em falha num universo de 10 000 não se tratando esta incoerência de necessárias situações de ruptura de stock, mas apenas falsas actualizações do mesmo. Sobre estes valores podemos argumentar que este género de operações pode ser implementado em Cassandra. Mantendo um registo das compras efectuadas por cada cliente para um controlo rigoroso das unidades a cada momento existentes e um mantendo o actual sistema de actualizações de stock por produto para fornecer uma não tão rigorosa informação sobre este medida, um sistema similar ao TPC-W pode ser implementado.

Tempo de espera	Factor de selecção	Total comprado (média)	Incoerência no stock (média)	Produtos afectados (média)
TPC-W	Uniforme	67327.6	8.4	3.2
TPC-W	Power Law	67449.0	46.0	15.6
1s	Uniforme	67402.4	635.0	210.4
1s	Power Law	67455.6	861.8	265.0

Tabela 3.2: Vendas de produtos e incoerências no stock - nível de coerência baseado em quoruns de nós

Critério de coerência	Factor de selecção	Total comprado (média)	Incoerência no stock (média)	Produtos afectados (média)
Um nó	Uniforme	67403.4	81.4	24.4
Um nó	Power Law	67545.2	94.6	27.6
Quorum	Uniforme	67579.4	51.4	15.4
Quorum	Power Law	67353.2	137.0	41.0

Tabela 3.3: Vendas de produtos e incoerências no stock - testes com sincronização dos relógios

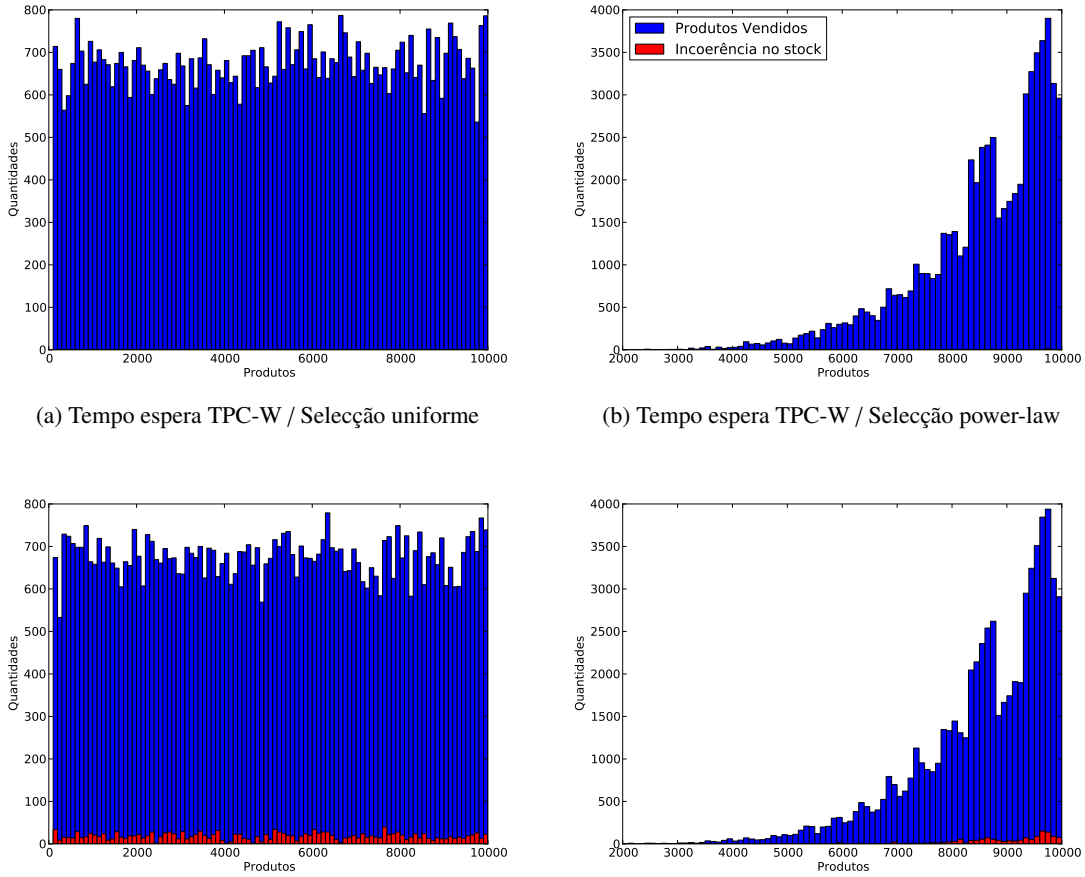
3.4 Avaliação de desempenho

Após o desenvolvimento e adaptação da plataforma de *benchmark*, pretende-se agora obter uma imagem do desempenho obtido pelo Cassandra e MySQL, estabelecendo-se diferentes perfis de *benchmark*. À imagem dos originalmente definidos no TPC-W, este perfis pretendem representar cargas maioritariamente de leitura e pesquisa de produtos ou que tenham também uma componente de compra e registo de novos clientes.

Sendo este caso de estudo focado na avaliação das bases de dados utilizadas, todas as operações relacionadas com a componente Web do TPC-W foram removidas. Com esta remoção perde-se contudo o conceito de navegação implícito nas páginas e correspondes factores de decisão que definiam os diferentes perfis de *benchmark* originais. Um esforço é no entanto desenvolvido para a recriação em parte destes perfis, de modo a que as razões entre as operações executadas sejam similares.

No processo de desenvolvimento foram assim construídos *workloads* focados nas operações a executar e não no fluxo entre elas. Deste modo, cada operação é manipulada de tal forma a não depender de qualquer output da plataforma, sendo os seus parâmetros gerados na fase anterior à execução. Como excepção encontramos as operações de compra que dependerão de um carrinho de compras já gerado, que se não existir será criado com um produto aleatório.

Neste contexto, também algumas das operações originais acabam por desaparecer por não serem relevantes à plataforma. É o caso das operações responsáveis pela simples recolha de informação tais como a exibição da última encomenda (*Order Display*), a submissão de pedidos de procura de produtos (*Search Request*) e de pedidos de administração dos mesmos (*Admin Request*). Toda a informação que era nestas operações gerada, continua a o ser de um modo similar, sendo apenas estas instruções integradas com as restantes na plataforma. No caso da operação de registo de um cliente (*Customer Registration*), esta passará a ter um significado diferente, executando agora a inserção de um novo cliente na base de dados.



(a) Tempo espera TPC-W / Selecção uniforme

(b) Tempo espera TPC-W / Selecção power-law

(c) Tempo espera 1s / Selecção uniforme

(d) Tempo espera 1s / Selecção power-law

Figura 3.5: Vendas de produtos e incoerências no stock - nível de coerência baseado em quoruns de nós

Ambiente de execução

Para a execução deste benchmark aqui definido são usadas duas máquinas de teste sobre um *cluster* de cinco máquinas. O hardware consiste em sete máquinas HP com um processador Intel(R) Core(TM)2 CPU 6400 - 2.13GHz, dois Gbyte de RAM e um disco rígido de uso comum SATA (7200 RPM). Todas as máquinas estão ligadas por LAN conectadas a um *switch* de 1GB/s. O sistema operativo é Linux, Ubuntu Server com kernel 2.6.31-1 e o sistema de ficheiros encontra-se em ext4. São assim utilizadas as mesmas sete máquinas onde duas funcionam como clientes de execução sendo as restantes usadas para a base de dados. Os motores de dados usados são o Cassandra versão 0.6.6 e o MySQL 5.1.54 sendo que nesta última existe um nó responsável pelas escritas, existindo depois 4 réplicas de leitura. O nível de isolamento usado no MySQL é *Read Repeatable* e o nível de coerência no Cassandra para a maioria das operações é o Quórum.

Em termos de parâmetros de execução, existem 60 (30x2) clientes concorrentes em duas máquinas individuais. Cada um deste efectua 200 operações com um tempo de espera como o definido na especificação original. Durante os testes, o factor de carga do TPC-W é 10 e o número de

Operações	Workload de Pesquisa (%)	Workload de Encomenda (%)
Operações de pesquisa	95	50
Home	29.00	9.12
New Products	11.00	0.46
Best Sellers	11.00	0.46
Product Detail	21.00	12.35
Search	23.00	27.61
Operações de Encomenda	5	50
Shopping Cart	2.00	13.53
Customer Registration	0.82	12.86
Buy Request	0.75	12.73
Buy Confirm	0.69	10.18
Order Inquiry	0.55	0.47
Admin Confirm	0.19	0.23

Tabela 3.4: Distribuição das operações num cenário de pesquisa e encomenda.

produtos na base de dados é 10.000. Cada teste é executado cinco vezes sendo os resultados finais uma média destes.

Como factores de variação usamos aqui dois dos cenários base de execução do TPC-W: o *workload* de pesquisa (*browsing mix*) e o de encomenda (*ordering mix*) cuja a distribuição de operações podemos ver na Tabela 3.4.

Análise dos resultados

Os resultados para os dois *workloads* testados encontram-se nas Tabelas 3.5 e 3.6 onde são usados os nomes das operações tal como presentes na especificação original. Numa análise preliminar dos resultados vemos que o benchmark apresenta resultados superiores em MySQL para todas as operações, sendo que em muitos dos casos a diferença atinge várias ordens de grandeza.

Note-se também que, os resultados para Cassandra são na sua maioria três, quatro, ou mais vezes maiores no cenário de leitura, como visível nas Figuras 3.6, 3.7 e 3.8 provenientes de uma das rondas de execução. De facto existem nestes testes grandes limitações de hardware e sendo as leituras mais prejudiciais, pois exigem no Cassandra um maior uso de recursos, as máquinas encontram-se muitas vezes em contenção de CPU. Assim os valores de latência são não só superiores mas também mais irregulares neste cenário.

Se o Cassandra é de facto uma alternativa viável para o armazenamento de dados em grande escala, esta base de dados ainda hoje se encontra em mutação num tentativa de optimizar muitas das suas operações, mecanismos de compactação e os efeitos nefastos do *garbage collection* do Java. Outras causas também se podem indicar para os maus resultados como o uso de super colunas em estruturas que são depois sujeitas a pesquisas sequenciais. Assim se cortarmos as instruções de pesquisa sequencial, temos os resultados apresentam na Tabela 3.7 que em termos de latência são menores e de menor variabilidade.

	MySQL			Cassandra		
Débito médio (operações/min)	417.4			198.8		
	Latência (ms)			**Latência (ms)**		
Operações	Média	10º percentil	90º percentil	Média	10º percentil	90º percentil
Home	1.61	1	2	854.58	136	1824
New Products	3.29	3	4	132.78	6	319
Best Sellers	1.71	1	2	8001.03	5198	11050
Product Detail	1.24	1	2	278.09	19	664
Search	1.74	1	3	147.26	2	374
Shopping Cart	3.19	2	3	410.82	51	920
Customer Registration	4.38	3	6	292.67	19	720
Buy Request	1.82	1	2	199.71	4	501
Buy Confirm	8.75	6	11	901.49	174	1850
Order Inquiry	2.65	1	4	508.55	36	1129
Admin Confirm	138.79	113	164	13395.06	8696	18582

Tabela 3.5: Resultados para um cenário de pesquisa

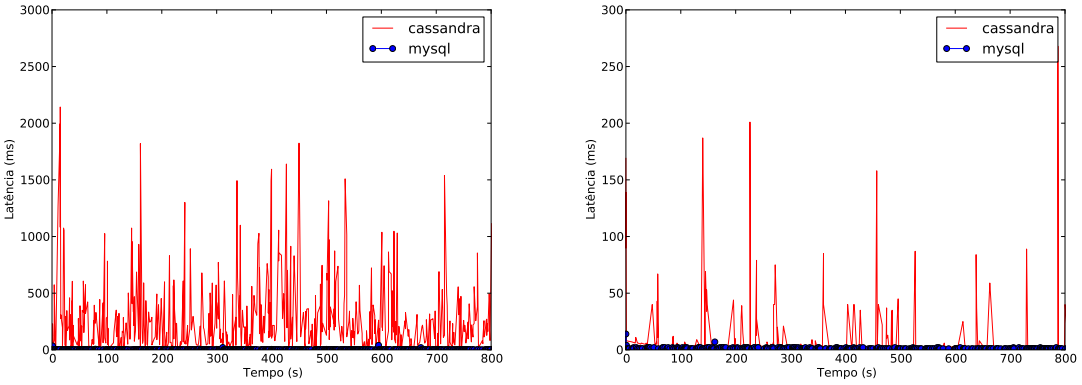
	MySQL			Cassandra		
Débito médio (operações/min)	466.2			442.6		
	Latência (ms)			**Latência (ms)**		
Operações	Média	90º percentil	10º percentil	Média	90º percentil	10º percentil
Home	1.64	1	2	31.98	7	83
New Products	3.47	3	4	9.12	4	12
Best Sellers	1.93	1	2	4218.53	3461	5079
Product Detail	1.22	1	2	10.43	2	18
Search	1.74	0	3	15.24	1	30
Shopping Cart	4.16	2	3	18.15	3	46
Customer Registration	4.69	2	4	13.41	4	27
Buy Request	2.24	1	2	8.70	2	14
Buy Confirm	8.16	5	8	33.66	9	85
Order Inquiry	2.45	1	4	17.17	2	38
Admin Confirm	172.83	135	208	6446.18	5306	781

Tabela 3.6: Resultados para um cenário de encomenda

3.5 Sumário

Com este capítulo pretende-se descrever as várias vertentes de migração da plataforma tal como as rotinas associadas. No seu total, este contém uma descrição do processo de modelação dos dados, a apresentação das diferentes API tal como uma medição de qual o efeito dos mecanismo de quóruns e resolução de conflitos na coerência dos dados. Com esta informação pretende-se aqui demonstrar as opções e decisões que o programador terá de tomar quando abordando este género de sistemas. No final realizamos também uma análise do desempenho da plataforma numa comparação entre MySQL e Cassandra.

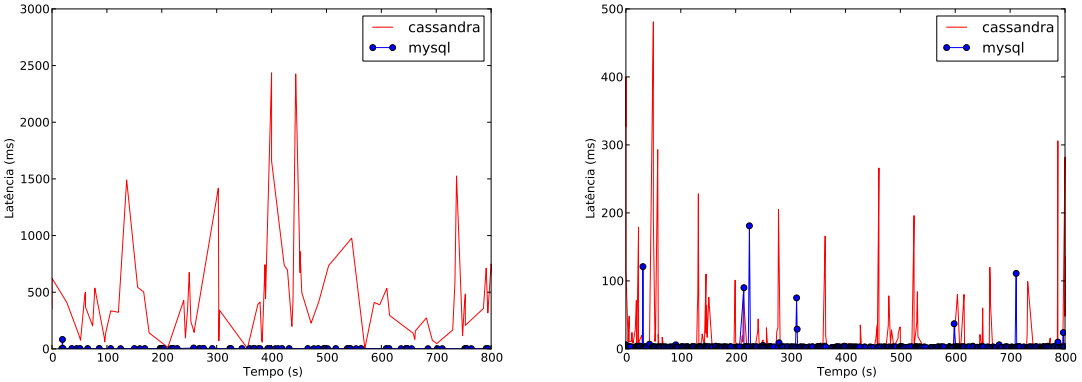
Podemos concluir que embora seja possível converter todas as operações para Cassandra, muitas delas tornam-se inutilizáveis devido às necessidades de tempo real que estão associadas a sistema como o especificado pelo TPC-W. Mesmo a operação de compra, embora implementável representa um esforço extra do programador e da plataforma para evitar as rupturas de saldo. Ainda assim, se consideráramos a escala para a qual o Cassandra foi desenhado, algumas destas



(a) Workload de pesquisa

(b) Workload de encomenda

Figura 3.6: Latência de numa operação de leitura base (*Product Detail*)



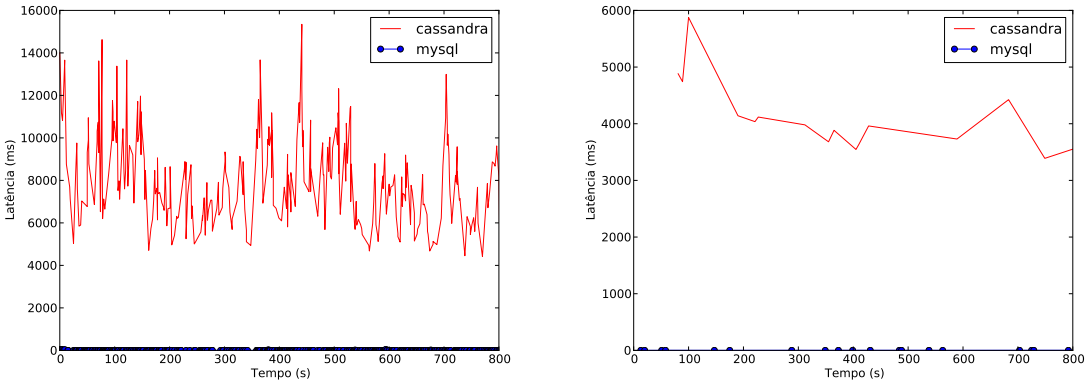
(a) Workload de pesquisa

(b) Workload de encomenda

Figura 3.7: Latência de numa operação de escrita base (*Shopping Cart*)

operações implicariam índices gigantescos e partição dos produtos, podendo esta por essa razão pode ser uma implementação viável.

Apesar destes primeiros maus resultados, esta solução apresentou ainda assim resultados razoáveis para padrões de acesso simples aos dados. Como tal, decidimos continuar a explorar esta solução de armazenamento de dados e num projecto em parceria com a Portugal Telecom partimos para um caso realista que a seguir apresentamos.



(a) Workload de pesquisa

(b) Workload de encomenda

Figura 3.8: Latência de numa operação com leitura sequencial (*Best Sellers*)

	Latência (ms)		
Operações	Média	90º percentil	10º percentil
Home	10.36	7	13
New Products	5.56	4	6
Product Detail	3.78	2	4
Search	10.28	1	30
Shopping Cart	6.61	4	7
Customer Registration	6.97	5	10
Buy Request	4.12	3	6
Buy Confirm	16.65	11	22
Order Inquiry	10.22	2	21

Tabela 3.7: Resultados sem operações sequenciais

Capítulo 4

Migração de uma aplicação realista

No seguimento do trabalho realizado na primeira fase atrás descrita, e com base nos conhecimentos adquiridos na mesma, entrámos agora na segunda parte deste trabalho. Do processo de estudo sobre a plataforma TPC-W desenvolvido partimos para a análise de um caso realista que representa um desafio de escala e adaptabilidade. Num contexto de indústria, apresenta-se a base da nossa colaboração com PT Inovação. Neste projecto analisa-se a fundo o sistema de atendimento de chamadas existente e qual a sua implementação no modelo relacional. Desta base, redesenha-se o sistema e efectua-se a sua implementação em Cassandra com vista a avaliar uma solução que permita uma mais fácil e barata adaptação do sistema a vários cenários de utilização que variam dos milhares de utilizadores à dezena de milhão.

No decorrer deste capítulo descrevemos o sistema em estudo, com os seus componentes e desafios associados. Iniciando pela modelação do problema num solução relacional, paradigma sobre o qual a solução se encontra actualmente implementada, passamos posteriormente para a sua adaptação para uma solução não relacional. Este processo encarado de forma diferente aqui, baseia-se numa cuidadosa análise das entradas e saídas de cada uma das operações efectuadas sobre o sistema, tentando assim maximizar o desempenho do mesmo. Estas duas alternativas são depois comparadas lado a lado.

4.1 Sistema de chamadas

No seio dos sistemas de telecomunicações da Portugal Telecom, e de forma similar nas outras empresas, existe um componente responsável pela gestão dos dados dos clientes e a sua consulta aquando do estabelecimento de uma chamada ou envio de um SMS. De facto, cada vez que um utilizador realiza uma chamada, todo um conjunto de procedimentos têm de ser efectuados para averiguar quais os serviços telefónicos que este usufrui, quais destes podem ou não ser usados e se existe ou não saldo disponível. Este género de dados abrange toda a população de utilizadores, sendo assim armazenada informação sobre os serviços que cada cliente possui, a sua conta associada, unidades de saldo, planos tarifários e outra informação aos serviços referente.

São identificáveis no sistema as seguintes entidades:

Cliente: Representação do cliente: nome, morada, etc. Cada cliente pode ter a si associado uma ou mais contas.

Conta: Entidade central do sistema, esta contém associados vários serviços subscritos de acordo com um tipo definido. Contém também uma associação a uma ou mais unidades de saldo caracterizadas também por um tipo definido. Para além disto, no seio do sistema, cada conta poderá ter a si associada uma outra conta numa ligação hierárquica que serve como mecanismo de delegação dos custos sempre que o cliente usufrui de um serviço onde existe um saldo partilhado com mais pessoas. Este tipo de conta apelida-se de conta pai.

Tipo de saldo: Entidade que codifica os diferentes tipos de unidades de saldo que existem no sistema (unidades de dinheiro, SMS, etc.). Na sua relação com cada conta nasce a entidade *Saldo* que codifica, para cada conta e tipo de unidade associado a esta, qual o saldo do cliente (e também qual a data de validade, etc).

Saldo: Representa qual o saldo do cliente segundo o tipo de unidade em causa. Esta entidade desempenha também um papel central na plataforma, pois cada chamada ou semelhante terá sempre de verificar este elemento para ser validada.

Serviço subscrito: Entidade que representa quais os serviços subscritos pelo cliente. Para além de um número (número de telemóvel ou outro tipo de identificador) que o liga ao cliente, esta entidade contém também uma prioridade. Este campo, mais à frente discutido, define aquando da chegada de um estimulo qual o serviço subscrito que irá lidar com essa chamada (e se este falhar qual o seguinte, etc.).

Tipo de serviço: Codifica os diferentes tipos de serviços que existem no sistema: serviços base de voz e SMS ou serviços de família, etc. A cada tipo de serviço existe também associado uma lista de quais os tipos de estímulo a que este suporta, isto é, se se trata de um serviço de voz, de voz e SMS ou de dados por exemplo. Quando um novo serviço é subscrito ele irá sempre ser definido segundo um destes tipos de serviço.

Plano Tarifário: Esta entidade é no sistema responsável pelo mapeamento entre serviços pelo cliente subscritos e os unidades de saldo que este pode consumir. Sendo que mais do que um tipo de saldo pode por vezes ser gasto pelo mesmo serviço, existe nesta entidade uma prioridade que traduz qual deles será o que vai ser consumido. Nesta relação existem também regras de uso e qual a tarifa associada a cada um dos casos.

Tal modelo pode ser visualizado na Figura 4.1 onde as linhas a cheio representam as ligações entre as entidades do sistema estando também representadas a multiplicidade das mesmas. As linhas a tracejado marcam as entidades que nascem nestas associações, como por exemplo, a ligação entre uma conta e um tipo de serviço que corresponde a um serviço subscrito pelo cliente.

4.1.1 Funcionamento

Estando definida a base do modelo de dados que suporta o sistema, apresenta-se agora os diversos passos de execução que um normal contacto efectua no sistema. Dividido em fases demarcadas, cada um destes passos é aqui examinado de acordo com os seus normais parâmetros de entrada, funcionamento geral e resultados. Nesta primeira fase, descrevemos assim os elementos do sistema de forma genérica sendo estes materializados, numa fase posterior, à luz dos diferentes paradigmas.

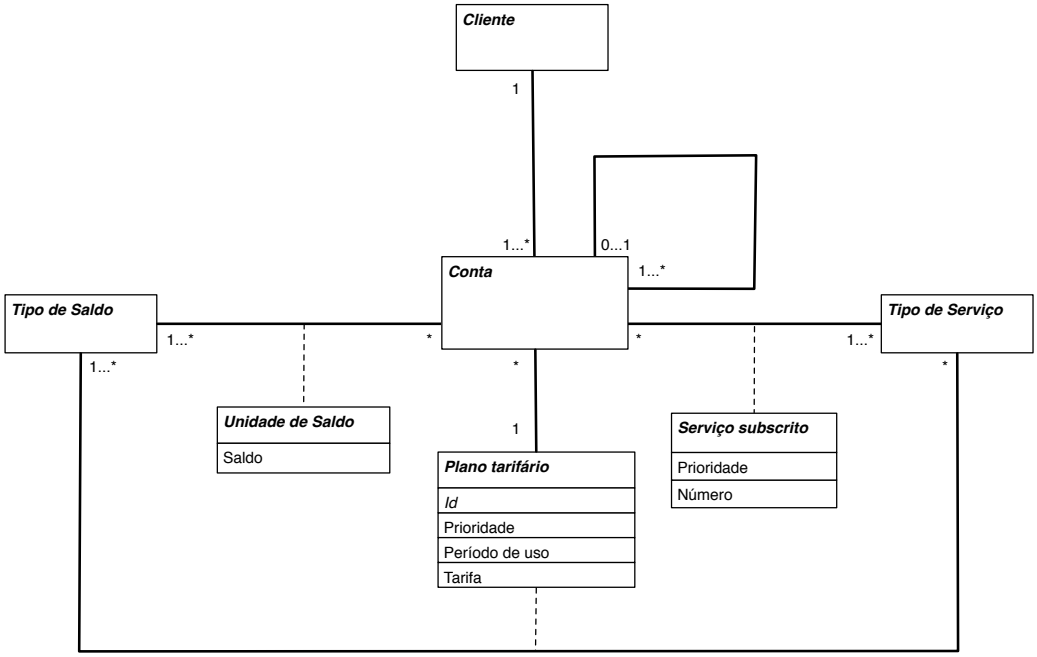


Figura 4.1: Modelo de entidades

Ignorando todo o processo previamente percorrido pelos estímulos de entrada da plataforma antes da chegada a esta, este processo inicia-se como o par (*Id*,*Tipo*). Neste inclui-se um identificador do cliente, como o seu número de telemóvel, tal como o tipo do estímulo (voz, SMS ou dados). Partindo deste par de entrada o restante processo segue-se como descrito.

Escolha dos serviços

A primeira fase pela qual cada estímulo passa aquando da entrada no sistema é a selecção dos serviços passíveis de serem utilizados. Esta pesquisa é realizada com base nos parâmetros de entrada da plataforma, nomeadamente o seu identificador e o tipo do evento recebido. Na pesquisa a plataforma terá de consultar os diferentes serviços associados ao identificador, filtrando-os para que restem apenas aqueles que correspondem ao tipo de entrada. Assim aquando de uma chamada de voz, a plataforma irá seleccionar, por exemplo, um serviço que dá desconto em chamadas para amigos ou outro semelhante, deixando de parte serviços de SMS e dados. Depois de filtrados pelo seu tipo, muitos destes serviços tem também condições de uso que têm de ser testadas. De facto, serviços que dão por exemplo descontos para um grupo de amigos têm de assim verificar se o número de destino pertence a tal categoria.

Note-se que pode haver mais do que um serviço valido, sendo depois escolhido um de acordo com um procedimento aqui simulado. Num caso real esta escolha será feita num critério sobre qual o serviço mais vantajoso, sendo normalmente o serviço que nas condições apresentadas é o mais barato. No entanto, por uma questão de simplificação, opta-se por incluir na entidade correspondente ao serviço subscrito um campo de prioridade que irá funcionar como elemento de

decisão[1]. Nesta fase extraímos também o identificador da conta a que estes serviços e o cliente se encontram associados.

Escolha dos tipo de saldo a utilizar

Tal como cada cliente pode possuir vários tipos de serviços associados ao seu dispositivo móvel ou fixo, também cada serviço pode ter diferentes fontes onde pode ir descontar a tarifa que corresponde ao contacto realizado. Esta informação que se encontra nos planos tarifários associados à conta do cliente, têm assim de ser recolhida com base no nome do serviço escolhido. Nestes planos diz-se para cada serviço que o cliente subscreveu quais os tipos de saldo que ele pode consumir, qual a ordem de consumo e a tarifa a cobrar. Este saldos podem ter igualmente restrições temporais de uso. Por exemplo, se o serviço seleccionado permite o envio de 30 SMS, das nove às nove, então o plano tarifário apontará para uma entidade saldo constituída por mensagens durante esse período, sendo que no restante tempo aponta para uma entidade do tipo monetário.

Resumindo, recebendo o tipo de serviço e a conta a que este está associado, neste passo será retornado um tipo de saldo a gastar tal como a tarifa a cobrar. Se no passo seguinte não existir saldo suficiente para a tarifa a cobrar então outro tipo de saldo será seleccionado.

Localização das unidades de saldo

Existe uma razão para o sistema ter retornado no passo anterior um tipo e não a localização das unidades de saldo a consumir, prendendo-se esta com o conceito de conta pai. Na verdade quando um cliente tem um serviço do tipo empresarial, este implica muitas vezes que seja a empresa a pagar parte dos custos das chamadas que este efectua. Como tal, este serviços apontam para um tipo de saldo que não existe na conta do cliente, e por isso o sistema terá de procurar uma conta associada onde encontrará as unidades de saldo correspondentes. Esta conta, usualmente ao encargo de uma empresa ou organização, é responsável por parte dos gastos do cliente. Este processo é no entanto evitado em muitos dos contactos realizados pois o tipo seleccionado está associado directamente ao cliente. Deste passo resulta assim a conta onde o saldo será descontado.

Consumo do saldo

No último passo ocorre o consumo de saldo de acordo com o tipo seleccionado e a tarifa a cobrar. Esta que é a mais simples das operações no sistema levanta, no entanto, questões de coerência quando em situações de partilha de saldo num empresa. Ao contrário de outras situações, como a venda de produtos, aqui o acesso não pode ser relaxado pois uma falha no saldo pode custar um cliente à empresa.

4.1.2 Serviços

Seguindo as direcções dos departamentos de marketing e afins, os serviços oferecidos pelas empresas de telecomunicações normalmente variam ao longo do tempo. Ainda assim, definimos aqui três dos exemplos usados na plataforma de simulação aquando do seu desenho e posterior teste.

[1]Considera-se que a prioridade quando referida no documento, segue uma ordem ascendente, isto é, um número mais alto corresponde a uma prioridade mais alta

Cada um destes serviços contém condições de uso que foram já referidas no fluxo do sistema e que são específicas ao serviço.

Família e amigos

No serviço de Família e amigos o cliente irá usufruir de descontos em todos os contactos estabelecidos com um grupo restrito de pessoas. Existindo assim uma lista de pessoas pelo utilizador definida, o sistema terá de manter tal lista para posterior verificação aquando do uso do serviço.

Comunidade

O serviço Comunidade funciona de certa maneira como uma extensão do serviço Família e amigos, onde ao invés de famílias singulares existe um grupo global de pessoas que o subscrevem. Este baseia-se no simples conceito de um grande grupo ao qual uma parte dos clientes pertence e assim poupam dinheiro entre si.

Golo

O serviço Golo não é mais que uma especialização do serviço comunidade, sendo que aqui os clientes se agrupam em termos de preferências desportivas. No entanto, neste serviço os clientes têm apenas direito a descontos aquando de um jogo da sua equipa. Devido a este factor de limitação no tempo, este serviço é designado como sendo do tipo dinâmico, pois possui limitações que variam ao longo do tempo.

4.2 Implementação relacional

Como primeiro passo neste projecto procedemos à implementação do sistema sobre uma base relacional, processo de certo modo ainda envolvido na troca de ideias entre ambas as partes para a obtenção de uma imagem final do mesmo. Este processo que aqui tem como suporte a base de dados PostgreSQL, seguiu as normais fases do paradigma relacional que aqui delineamos. Da representação dos dados e das relações entre estes passando pelas instruções SQL para o seu acesso e pela sua optimização numa fase posterior, documentamos aqui os vários passos de desenvolvimento.

4.2.1 Modelo de dados

Numa primeira fase, debruçamo-nos sobre o desenho do modelo de dados onde se representa cada uma das entidades do sistema e as suas relações. Este processo, que segue o trabalho de modelação já realizado e que se encontram representado na Figura 4.1, é então aqui descrito em termos de tabelas e relações com exemplos inclusos.

Iniciando pelas entidades que estão directamente ligadas ao cliente, temos em primeiro lugar a tabela responsável pela persistência dos seus dados. Esta entidade cliente acaba por não desempenhar um papel de maior na parte da plataforma que aqui simulamos, mas inclui-se ainda assim por representar um elemento importante do sistema global. Passando para a entidade conta,

Cliente

***Cliente* Id**	**Nome**
C1	João
C2	Rui
C3	Miguel
C4	Lab

Conta

Id	**Cliente Id**	**Plano Tarifário**
A1	C1	T1
A2	C2	T2
A3	C3	T3
A4	C4	---

Relações Contas

Conta pai	**Conta Id**
A4	A2
A4	A3

Figura 4.2: Tabelas relativas a contas e clientes.

Serviços Subscritos

Nome Serviço	**Conta Id**	**Numero**	**Prioridade**
Voz	A1	960000001	0
SMS	A1	960000001	0
Voz	A2	960000002	0
Voz	A3	960000003	0
Familia_voz	A2	960000002	1
Familia_sms	A2	960000002	0
Partilhado	A2	960000002	2
Partilhado	A3	960000003	1

Tipo Serviço

Nome Serviço	**Tipo**
Voz	Voz
SMS	SMS
Partilhado	Voz
Familia_voz	Voz
Familia_sms	SMS

Figura 4.3: Tabelas relativas aos serviços.

esta desempenha aqui um papel de ligação entre algumas das outras entidades. Ligada à entidade cliente e sendo que este pode possuir mais que uma conta, esta tabela é responsável por manter a chave que codifica a relação. Nesta tabela encontramos também a chave do plano tarifário que à conta em questão corresponde. Por fim temos a entidade que codifica as relações entre contas indicando para cada uma destas qual a sua conta pai se tal existir. Tais tabelas são criadas no sistema sendo os exemplos visíveis na Figura 4.2 onde, aqui e nas seguintes, se indicam também as chaves primárias utilizadas.

Passando para o domínio dos serviços, temos a tabela que codifica os tipos de serviço e quais os tipos de entradas que estes suportam (voz, SMS, etc.). Numa ligação entre esta e a entidade que codifica a conta do cliente temos então a tabela dos serviços subscritos. Nesta tabela encontramos associado a cada nome de serviço o identificador do dispositivo[2], a prioridade que este possui para o cliente e qual a conta associada. Um exemplo destas tabelas pode ser visto na Figura 4.3.

Igualmente presente na base de dados, podemos também encontrar a tabela responsável pelos dados que aos planos tarifários se referem. Nestas estruturas podemos encontrar, agrupada por um identificador que delimita os diferentes tarifários, informação sobre que tipos de saldo que cada serviço pode usar. Em cada linha encontram-se também as colunas que codificam quando e com que prioridade estes podem ser usados tal como o preço associado. Esta tabela é visível na Figura 4.4.

[2]Por uma questão de simplificação este identificador será no decorrer do documento identificando como sendo o número do cliente, tornando assim as imagens e exemplos mais legíveis

Plano Tarifário					
***Plano* Id**	**Nome Serviço**	***Tipo Saldo***	**Prioridade**	**Periodo de uso**	**Tarifa por minuto**
T1	Voz	Dinheiro	0	24:00- 24:00	5
T1	SMS	SMS	1	8:00 - 9:00	5
T1	SMS	Dinheiro	0	24:00 - 24:00	5
T2	Partilhado	Dinheiro Partilhado	1	8:00 - 19:00	5
T2	Partilhado	Dinheiro	0	24:00- 24:00	5
T2	Voz	Dinheiro	0	24:00- 24:00	5
T2	Familia_voz	Dinheiro	0	8:00 - 9:00	5
T2	Familia_sms	SMS	1	8:00 - 9:00	5
T2	Familia_sms	Dinheiro	0	8:00 - 9:00	5
T3	Voz	Dinheiro	0	24:00 - 24:00	5
T3	Partilhado	Dinheiro Partilhado	1	8:00 - 19:00	5
T3	Partilhado	Dinheiro	0	24:00- 24:00	5

Figura 4.4: Tabela relativa aos planos tarifários.

Tipo Saldo
Tipo
Dinheiro
SMS
Dinheiro Partilhado

Unidades de Saldo		
Conta Id	***Tipo Saldo***	**Saldo**
A1	Dinheiro	10
A1	SMS	50
A2	Dinheiro	15
A2	SMS	12
A3	Dinheiro	20
A4	Dinheiro Partilhado	50

Conta		
Id	**Cliente Id**	**Plano Tarifário**
A1	C1	T1
A2	C2	T2
A3	C3	T3
A4	C4	---

Figura 4.5: Tabelas relativas aos saldos.

Por último temos as estruturas onde é persistida a informação sobre os saldos dos clientes. Iniciando pela tabela que codifica quais os tipos de saldo existentes no sistema, esta em relação como a entidade conta dá lugar à tabela dos saldos. As unidades de saldo que é uma das principais tabelas no sistema, contém para cada conta e tipo qual o valor do saldo. Terminando deste modo a modelação das entidades base do sistema, podemos ver estas tabelas na Figura 4.5.

Serviços

Indo além do modelo base, alguns serviços exigem também a construção de estruturas auxiliares na base de dados para o seu correcto funcionamento. Assim dos três exemplos dados, o serviço Família e amigos necessita de uma tabela que registe os elementos que pertencem à lista de contactos definida pelo utilizador . Do mesmo modo, para o serviço Golo terá de ser gerada uma tabela que descreva o calendário de jogos que irão ocorrer. Exemplificando a informação persistida, estas

Relações Família	
Número de Cliente	**Número Família**
960000002	960000003
960000002	960000005
960000005	960000002
960000005	910000007

Jogos Futebol		
Equipa	**Início do Jogo**	**Fim do Jogo**
Benfica	31/12/2011 16:00	31/12/2011 18:00
Porto	31/12/2011 18:00	31/12/2011 20:00
Sporting	31/12/2011 18:00	31/12/2011 20:00

Figura 4.6: Tabelas relativas aos serviços implementados.

tabelas estão representadas na Figura 4.6. Já no caso de serviços como o Comunidade, estes não necessitam de estruturas secundárias pois a sua validação pode ser realizada sobre o modelo base.

4.2.2 Operações

Sobre o desenho do modelo de dados entramos agora na fase de desenvolvimento do sistema. Nesta, optamos por percorrer cada umas das fases de execução que atrás descrevemos e para cada uma das quatro estabelecemos como pode esta em SQL ser transcritas. Mas além do simples desenvolvimento destas instruções, documentamos também aqui aquele que é usualmente o primeiro passo de optimização nas soluções relacionais: a criação de índices secundários. Analisamos assim estas operações e pelos seus padrões de acesso avaliamos quais o campos de risco sobre os quais devemos então criar índices secundários.

Escolha dos Serviços Nesta primeira fase, e como já descrito, são escolhidos os serviços com base no número do utilizador e o tipo de estímulo que chegou à plataforma. Tomando como parâmetro de entrada o par (*Número*,*Tipo*), para a pesquisa dos serviços e contas associadas é então usada a seguinte instrução:

```
SELECT ServicosSubscritos.conta_id,ServicosSubscritos.nome_servico
 FROM ServicosSubscritos
  JOIN TipoServicos
   ON  TipoServicos.nome_servico = ServicosSubscritos.nome_servico
   AND TipoServicos.tipo_servico = Tipo
 WHERE ServicosSubscritos.numero = Número
 ORDER BY ServicosSubscritos.prioridade DESC
```

Nesta instrução existe a selecção dos serviços que possuem o número de entrada, sendo a filtragem para a obtenção dos que realmente suportam o tipo de entrada realizada com a ajuda de um *Join* à tabela dos tipos de serviços. Deste modo, a tabela dos serviços subscritos é acedida com base nas colunas nome do serviço e número, sendo que apenas um dos campos faz parte da chave primária. De forma a optimizar a performance desta instrução, é criado um índice secundário com base nestes dois campos. O mesmo se aplica à tabela dos serviços, que sendo acedida pelo campo tipo de serviço, é também complementada com um índice sobre esse campo.

Escolha do tipo de saldo Após a escolha de um dos serviços entre os vários retornados, a fase seguinte compreende a escolha dos tipos de saldo passíveis de serem utilizados. Tal operação é codificada pela seguinte instrução SQL que tem como parâmetros de entrada o par (*Serviço*, *Conta Id*), resultante do passo anterior.

```
SELECT PlanoTarifario.tipo_saldo FROM PlanoTarifario
 WHERE PlanoTarifario.plano_id = (
   SELECT Conta.plano_tarifario FROM Conta
      WHERE Conta.id = Conta Id
 )
 AND PlanoTarifario.nome_servico  = Serviço
 AND Hora_actual BETWEEN PlanoTarifario.periodo_de_fim_uso
   AND PlanoTarifario.periodo_de_inicio_uso
 ORDER BY PlanoTarifario.prioridade DESC
```

Analisando a instrução, pode-se observar um acesso à tabela, que contêm as contas dos clientes, com base na sua chave primária. Quanto ao plano tarifário, este é acedido pelo seu identificador e nome de serviço, ambos campos da chave primária. É de seguida efectuado, um teste com base nas horas de uso não sendo, no entanto, importante na recolha primária de informação. Assim os acessos efectuados são sempre efectuados por chave primária não sendo por isso necessário adicionar mais nenhum índice. É importante salientar a ordem das chaves na criação da tabela que contêm os planos tarifários, da tal forma que os campos acedidos têm de estar à cabeça dos campos que constituem a chave primária.

Localização das unidades de saldo Após a escolha do tipo de saldo a utilizar, é necessário descobrir qual a origem do mesmo, pois este pode estar contido na conta base do cliente ou numa conta pai. Assim, com base no par (*Conta id*,*Tipo Saldo*), para verificar se esse tipo existe na conta base usa-se a seguinte instrução:

```
SELECT EXISTS(SELECT 1 FROM UnidadesSaldo
 WHEREUnidadesSaldo.conta_id = Conta Id
 AND UnidadesSaldo.tipo_saldo = Tipo Saldo
```

Se esta falhar, é então preciso verificar a conta pai através da instrução:

```
SELECT UnidadesSaldo.conta_id  FROM UnidadesSaldo
 WHERE UnidadesSaldo.conta_id = (
   SELECT RelacaoContas.conta_pai
     FROM RelacaoContas
   WHERE RelacaoContas.conta_id = Conta Id
 )
 AND UnidadesSaldo.tipo_saldo = Tipo Saldo
```

Fazendo uso das tabelas que codificam os saldos, e no segundo caso as relações entre contas base e pais, os campos aqui acedidos são também chaves primárias. Deste modo, não são adicionados índices secundários.

Consumo do saldo O último passo é o consumo do saldo contido na unidade de saldo seleccionada. Este contendo parâmetros de entrada similares à operação anterior, realiza-se pela seguinte instrução:

```
UPDATE UnidadesSaldo SET UnidadesSaldo.saldo = ?
  WHERE UnidadesSaldo.conta_id = Conta Id
  AND UnidadesSaldo.tipo_saldo = Tipo Saldo
```

Sendo esta similar às anteriores, no sentido em que os campos acedidos aqui são os mesmos, nada assim existe a optimizar.

4.2.3 Serviços

De modo similar ao processo de construção do modelo de dados, também em termos de implementação descrevemos aqui o modo como as validações dos vários serviços são executadas. Assim para cada um dos três exemplos descrevemos como estes são transcrito para SQL e quais os índices criados para a sua optimização. Nas suas descrições assume-se como parâmetro de entrada a informação de descrição da chamada, ou seja, o tuplo (*Número de origem,Número de destino,Tipo*)

Familia Com base numa lista, que contêm, para cada utilizador os seus amigos e familiares, a validação do serviço Família e amigos baseia-se na seguinte instrução:

```
SELECT EXISTS(
  SELECT 1 FROM RelacoesFamilia
    WHERE RelacoesFamilia.num_cliente = Número de origem
       AND RelacoesFamilia.num_familia = Número de destino
)
```

Nesta os campos acedidos são as chaves primárias da tabela, e nenhum índice é assim necessário.

Comunidade A validação do serviço Comunidade tem por base a tabela dos serviços subscritos, onde é verificado se o cliente de destino possui este mesmo serviço. Para atingir esse objectivo, e adicionando aqui como parâmetro o serviço em causa que identificamos abaixo por *Serviço*, é usada a seguinte instrução:

```
SELECT EXISTS(
  SELECT 1 FROM ServicoSubscrito
   WHERE ServicoSubscrito.nome_servico  = Serviço
    AND ServicoSubscrito.numero = Número de destino
)
```

Sendo esta instrução é similar à pesquisa de serviços no sistema, também aqui é utilizado o índice criado sobre os números e nomes dos clientes.

Golo Adicionando informação ao serviço Comunidade, o serviço Golo valida não só se os clientes partilham o serviço, mas também se está um jogo a decorrer. Nesta operação temos como parâmetros de entrada adicionais o serviço em causa e a equipa da qual o cliente é adepto, formando assim o par (*Serviço*,*Equipa*). Quanto à sua implementação, esta validação é efectuada pela seguinte instrução:

```
SELECT EXISTS( SELECT 1 FROM ServicoSubscrito
   WHERE ServicoSubscrito.nome_servico  = Serviço
      AND ServicoSubscrito.numero = Número de destino
      AND EXISTS( SELECT 1 FROM JogosFutebol
    WHERE JogosFutebol = Equipa
      AND CURRENT_DATE
    BETWEEN JogosFutebol.início_jogo
     AND JogosFutebol.fim_jogo
       )
)
```

Mais uma vez, o índice duplo sobre a tabela dos serviços subscrito pode ser utilizado. Para o resto da instrução, como o campo de pesquisa é a equipa de futebol, então o índice de chave primária desta tabela, leva a que não seja preciso mais optimizações.

4.3 Implementação não relacional

No desenvolvimento da solução não relacional em Cassandra parte-se não dos dados como na anterior implementação, mas sim das funcionalidades que são pretendidos para o sistema. Esta característica da modelação de soluções em bases de dados não relacionais que foi já demonstrada no capítulo anterior, é aqui ainda mais evidente. De facto, sem a necessidade de descrever os detalhes de modelação e da base de dados, o processo aqui descrito não se separa em secções de modelo de dados ou operações uma vez que estas duas realidades acabam por não ser divisíveis.

Neste paradigma, os dados são, para optimização do pedidos, armazenados e acedidos com base num estrutura que acaba por transparecer de forma mais explicita a lógica de negócio, numa clara separação do tradicional paradigma relacional. Levando a uma optimização da actual implementação, esta escolha tem no entanto custos em futuras alterações do sistema que necessitem de novas ou diferentes operações sobre os dados. Este problema é apesar de tudo atenuado pela flexibilidade dos esquemas de dados que caracterizam estas soluções, e que permitem em tempo real actualizar as estruturas de armazenamento, dando ao cliente a possibilidade de adoptar uma postura mais dinâmica na forma como encara os dados.

4.3.1 Operações e modelo de dados

Abordando o desenvolvimento da parte não relacional da plataforma, apresenta-se agora este processo através do fluxo do sistema. Sendo as estruturas de dados e operações derivadas do funcionamento do mesmo, apresentamos assim como as suas 4 fases distintas marcam a implementação.

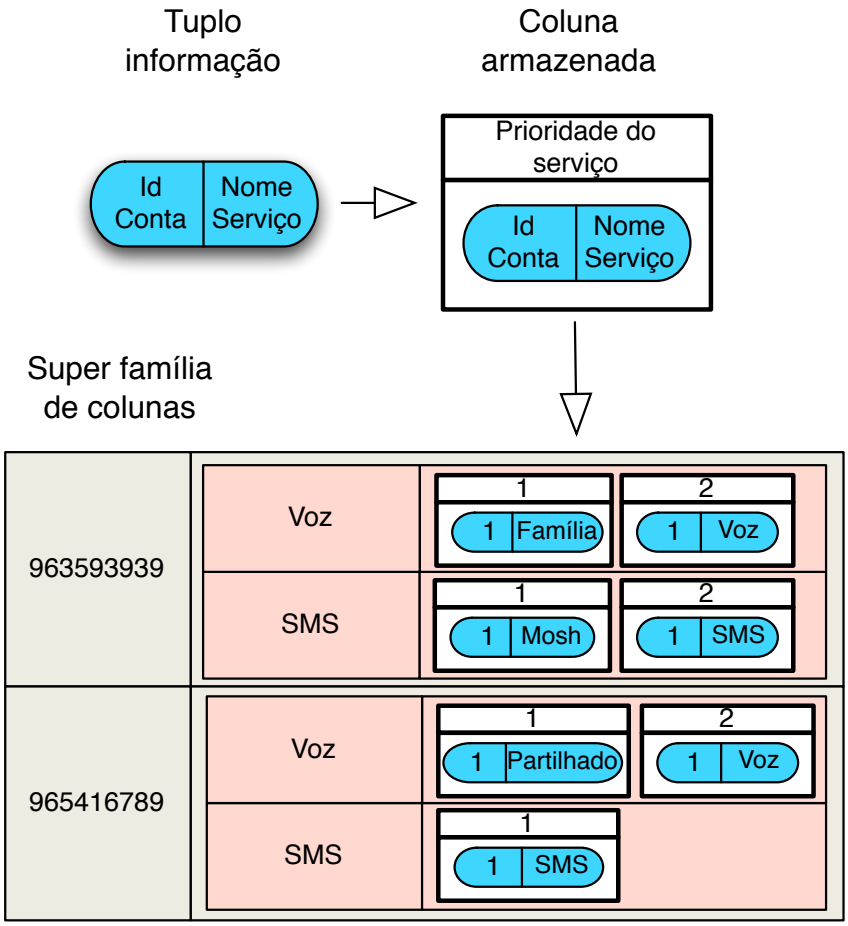


Figura 4.7: Super família de colunas - Serviços.

Para cada uma destas indicamos os parâmetros de entrada e qual o resultado pretendido para uma melhor compreensão do processo.

Escolha dos Serviços Nesta fase, partindo do estímulo de entrada, que contém o número do cliente e o tipo de chamada, temos de identificar quais os serviços que podem ser usados.

Parâmetros de entrada O número e o tipo de chamada recebidos pelo sistema.

Resultado pretendido A lista dos serviços passíveis de serem utilizados, e a conta à qual estes estão associados, ordenados por uma prioridade presente nos dados.

Sendo a informação devolvida o número de conta e os nomes dos serviços, começamos pela criação de um tuplo que contenha tal informação. Este é depois inserido na base de dados em colunas, onde o nome da coluna é a sua prioridade, deste modo é garantido que os dados são lidos ordenadamente. No último passo, estas colunas são colocadas numa família de super colunas, onde serão mapeadas pelo número e tipo de chamada. Uma representação desta família é visível na Figura 4.7.

Com uma simples operação de procura na base de dados, obtemos assim a informação pretendida, pois esta encontra-se associada aos parâmetros que dão entrada na plataforma.

Após a recolha dos serviços, estes estão já ordenados segundo a sua prioridade, sendo verificadas uma a uma as restrições a estas associadas (por exemplo se o destinatário pertence aos contactos do serviço Família). Quando encontrado um serviço que cumpra as suas restrições, passa-se à fase seguinte.

Escolha dos tipos de saldo Após a escolha do serviço a utilizar, o passo seguinte passa pela consulta do plano tarifário, à conta associado para saber qual o tipo de saldo a utilizar. À semelhança dos serviços, os tipos de saldo seguem também uma prioridade e possuem condições de utilização baseadas na hora de consulta. Focando nas restrições temporais, estas podiam ser avaliadas no processo de leitura, mas como isso implicaria a construção de índices sobre parâmetros de elevada cardinalidade optou-se aqui pelo seu retorno para o lado do cliente onde são avaliadas.

Parâmetros de entrada O serviço a usar e a conta associada

Resultado pretendido A lista dos tipos de saldo que podem ser utilizados para o serviço em causa, as suas tarifas e as condições temporais associadas.

À semelhança da escolha dos serviços, também aqui a estratégia passa pela criação de um tuplo que contêm a informação que pretendemos retornar. Este tuplo é constituído pelo tipo de saldo, a tarifa a consumir e as data de inicio e fim, sendo depois armazenado em colunas, onde a nome da coluna é novamente a sua prioridade. Estas colunas são depois mapeadas pela informação de procura, ou seja, pelo número de conta e posteriormente pelo nome de serviço. A Figura 4.8 representa esta estrutura de dados.

Após o retorno dos vários tipos de saldo, procede-se à verificação da hora de uso e, se positiva, passamos ao passo seguinte.

Localização das unidades de saldo Após a escolha do tipo de saldo a utilizar, o próximo passo implica pesquisar a sua localização, pois este pode estar numa conta local ou numa conta pai.

Parâmetros de entrada O tipo de saldo que pretendemos usar e a conta base onde pesquisar.

Resultado pretendido O identificador da conta onde se encontra a unidade de saldo a utilizar

Este passo podia, à semelhança da solução relacional, ser resolvido com uma tabela que codifica as relações entre contas. Mas tendo em conta o passo seguinte, onde para cada conta e tipo de saldo é necessário ler e alterar o seu valor, opta-se por construir uma estrutura de dados que abranja ambas as funcionalidades.

Resumindo-se a informação sobre a unidade de saldo ao seu valor, esta estará normalmente mapeada pelo seu identificador de conta e tipo de saldo. Esta estrutura é depois também armazenada numa família de super colunas. No entanto, também existe a necessidade de expressar a localização da unidade de saldo (o saldo pode estar noutra conta que não a base), sendo que tal tarefa é simplificada por não haver um esquema fixo de coluna, como se demonstra na Figura 4.9.

Por último, resta apenas a actualização do saldo.

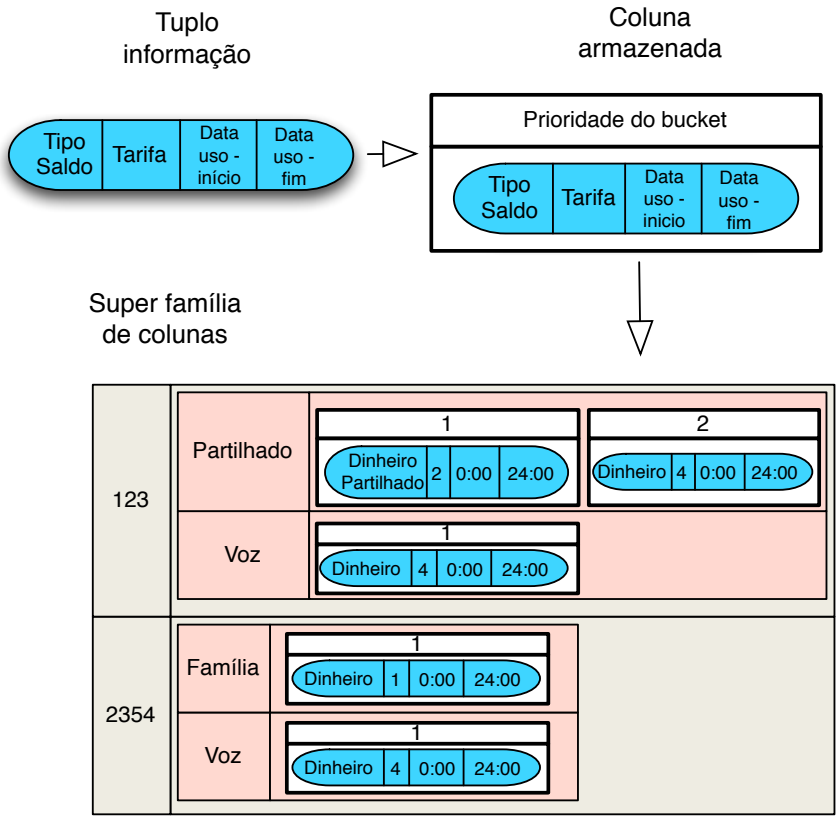


Figura 4.8: Super família de colunas - Plano tarifário.

Consumo do saldo Após os passos descritos anteriormente, resta apenas um último passo de actualização do saldo. Realçando que nenhum mecanismo foi utilizado para garantir a coerência, o saldo é modificado na estrutura descrita no passo anterior (Figura 4.9).

Finalmente, é de notar que a informação sobre o cliente não se encontra em nenhuma das estruturas já mencionadas. Esta é armazenada numa estrutura idêntica à da Figura 4.10.

4.3.2 Serviços

Referidas no primeiro passo de execução, as condições de validação dos serviços foram também implementadas em Cassandra. Mostra-se, de seguida como cada um dos três serviços descritos foi portado para esta base de dados e as estruturas que para tal foram criadas.

Serviço Família Por definição, o serviço Família oferece preços especiais em chamadas para os números que pertencem ao grupo de familiares definido pelo cliente. À semelhança da implementação relacional onde esta relação era mapeada numa tabela, também aqui opta-se pela construção de uma família de colunas à parte. Esta pode ser visualizada na Figura 4.11.

Serviço Comunidade Num serviço do tipo Comunidade, o desafio consiste em determinar se ambos os intervenientes possuem o serviço, ou seja, se ambos pertencem à mesma comunidade.

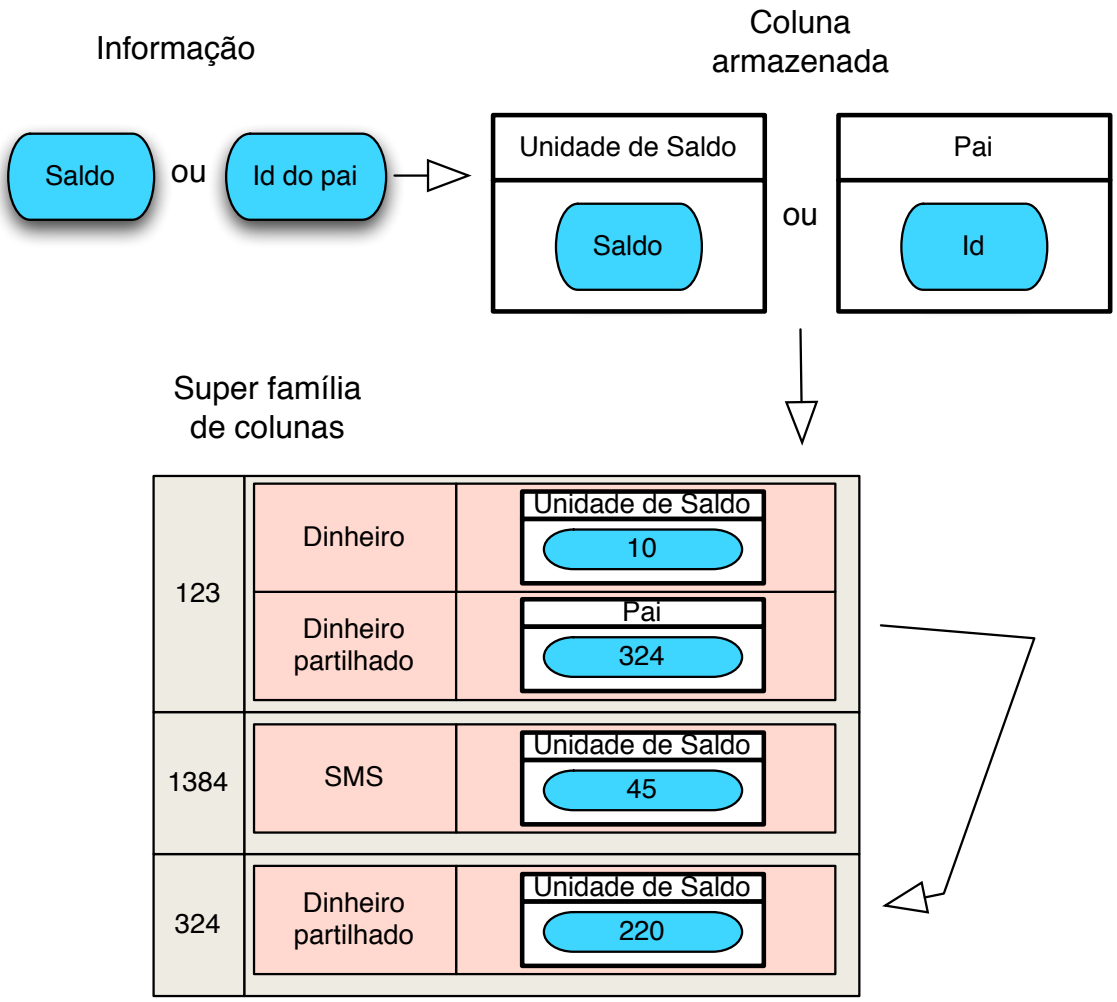


Figura 4.9: Super família de colunas - Saldo.

Embora, seja possível no modelo actual obter essa informação através da família dos serviços, tal seria ineficiente pois requereria a leitura de todos os serviços atribuídos ao utilizador. Assim, foi criada uma família de colunas adicional, que contém todos os serviços de cada cliente, permitindo realizar a validação com uma simples operação de leitura. Esta solução, como em qualquer acto de desnormalização da informação, implica no entanto que na alteração dos dados seja feita uma dupla escrita ou remoção do mesmo; mas sendo este um cenário onde perduram as leituras esta é uma opção válida. Tal família de colunas é apresentada na Figura 4.12

Embora esta seja a opção mais eficiente, assumindo que o cliente não subscreva muitos serviços, a solução usando a leitura da família dos serviços pode tornar-se mais vantajosa pois permite poupar espaço de armazenamento. Se essa solução for usada, pode-se sacrificar o ordenamento pela prioridade, e colocando o nome do serviço como nome das colunas, permitindo realizar a validação sem extrair mais do que um serviço.

Serviço Golo O serviço Golo, como já descrito, tem como condições de uso que para além dos clientes partilharem o mesmo serviço, tem de estar também a ocorrer nesse momento um jogo da

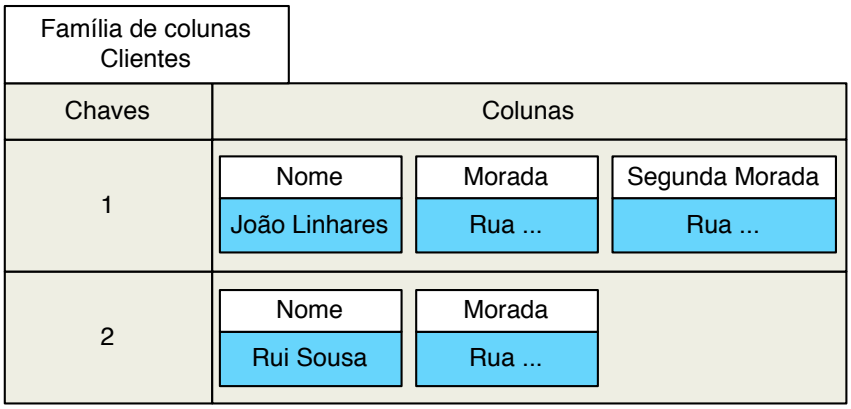


Figura 4.10: Família de colunas - Clientes.

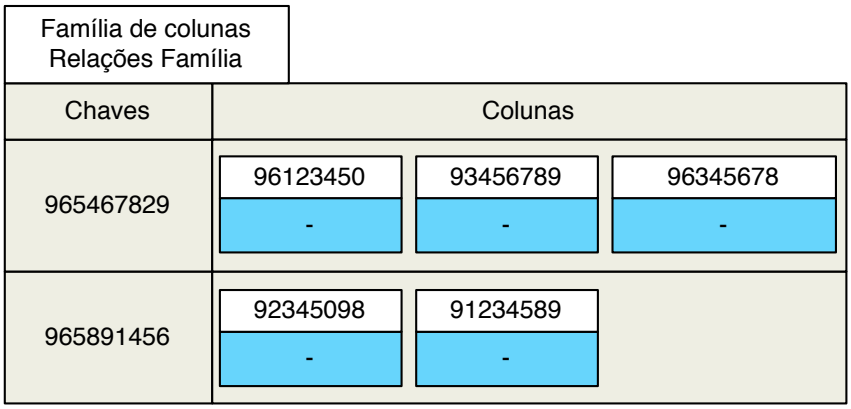


Figura 4.11: Família de colunas - Serviço Família e Amigos.

equipa da qual estes são adeptos.

Com a estrutura da Figura 4.12 pode-se facilmente responder à primeira parte da questão, sendo no entanto necessário criar uma nova família de colunas para consultar a informação sobre as datas de realização dos jogos. Tal estrutura pode ser visualizada na Figura 4.13.

4.4 Avaliação do sistema

4.4.1 Plataforma de simulação

No contexto do projecto, é de vital importância o desenvolvimento de uma plataforma de simulação e *benchmarking* do sistema, que permita a geração de valores para a inserção na base dados e de operações que simulem diferentes cenários. Existe no entanto um desafio, uma vez que os dados do sistema e as suas relações tal como o processo de geração de chamadas envolvem muitas vezes factores externos, que sendo de natureza humana podem seguir vários padrões ao longo do tempo e do espaço. Tal cenário leva à tentativa de modelar a informação apresentada, de uma forma mais genérica possível, de tal modo que o sistema se adapte facilmente a vários cenários e requisitos. Tal simplificação baseia-se no seguinte pressuposto: Um qualquer cliente, sempre

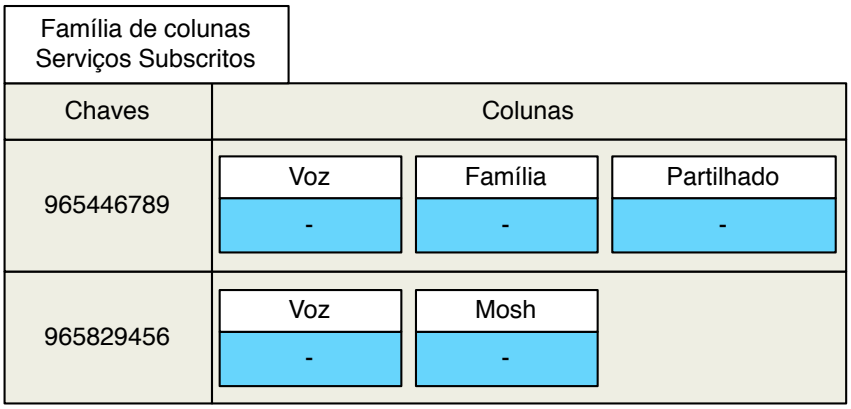


Figura 4.12: Família de colunas - Serviços subscritos.

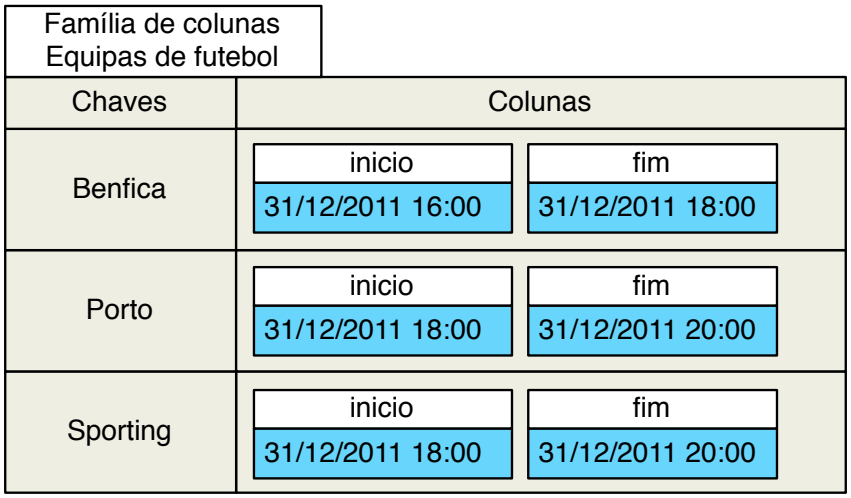


Figura 4.13: Família de colunas - Equipas e jogos.

que realiza uma chamada ou subscreve um serviço, fá-lo normalmente baseado em factores de localidade ou tempo.

Os critérios temporais referidos, resultam de uma grande parte das chamadas ser efectuada para números com os quais o utilizador interagiu recentemente, ou seja, se o utilizador faz uma chamada para um determinado número, ou se recebe uma chamada de um determinado número, então existe uma grande probabilidade de o utilizador o voltar a usar. Já os critérios de localidade, referem-se às pessoas mais próximas do cliente e que pertencem ao mesmo grupo que este. Introduz-se assim a noção de grupo. O grupo é um conjunto de clientes ligados entre si por laços de parentesco ou profissionais, afectando a escolha de quais os clientes que partilham o serviço A ou B por exemplo, mas tem também um papel vital na geração de chamadas, naquilo a que chamamos de proximidade local.

A plataforma evolui a partir deste núcleo, constituído pelos clientes e os seus grupos de relacionamentos. Assente numa base configurável, onde outros parâmetros como o número de grupos ou a relevância de cada um deste pode ser alterado, esta permite modelar vários cenários de uso do sistema.

Cliente, grupos e serviços

Na geração de informação para inserção na base dados, e a sua posterior interrogação, os dados dos clientes e dos seus grupos de relacionamento são os mais importantes. Em seguida, são apresentados, os desafios e respostas na criação destas entidades.

Definição dos grupos e suas características A medida de a quantos e quais os grupos a que um cliente pertence depende dos diferentes grupos que se admite existir, e qual a sua representação na população. Por exemplo, os grupos do tipo família ao qual, naturalmente, a maior parte dos clientes pertencerá, ao contrário dos grupos do tipo trabalho, que apenas afectarão uma menor parte da população. Deste modo, este número resultará de parâmetros que indiquem a distribuição dos grupos pela população de clientes.

Atribuição dos clientes a grupos Após a atribuição de um tipo de grupo a um cliente, existe ainda a questão a qual grupo em especifico ele irá pertencer. Numa primeira fase, o número de grupos é determinado pela razão entre o tamanho da população de teste e o número médio de pessoas que se admite existirem em cada um deles. A escolha de um destes grupos, será depois realizada através de uma distribuição uniforme ou *power law* (entre outras). Convém especificar, que a distribuição *power law* permite retratar as disparidades entre os clientes, que possuem uma larga rede de contactos, e o utilizador comum que possui uma pequena rede de conhecidos.

Relação entre serviços/grupos Existe normalmente uma relação de um-para-um entre grupos e serviços, de tal modo que se o cliente pertence a um grupo Família então ele subscreve o subscreve o serviço Família e Amigos. Esta atribuição depende no entanto da implementação de cada um dos serviços. Se tomarmos como exemplo o serviço Golo, ele partirá os grupos em clubes segundo a distribuição dos mesmos, ou seja, dos grupos globais 30% vão para a equipa *P* e 70% vão para a equipa *B*. Do mesmo modo, um grupo em específico pode atribuir vários serviços sobre um mesmo nome, isto é, pode criar serviços de voz, SMS e dados.

A geração de informação sobre contas e unidade de saldo não foi abordada, pois deriva directamente dos dados sobre os clientes e os seus serviços, o mesmo acontecendo para as restantes entidades do sistema.

Chamadas

Outra parte vital do sistema, é a geração de chamadas que, é influenciada no contexto de cada cliente tanto por factores temporais como por factores locais. Neste contexto, o modelo de dados do *benchmark* é o apresentado na Figura 4.14.

Deste modo, cada cliente possui um conjunto de grupos e, por conseguinte, cada grupo tem um conjunto de clientes. Por outro lado, cada cliente possui uma lista de clientes que lhe ligaram recentemente e clientes para os quais ele ligou. Este modelo contém a informação base do cliente, mas mais do que isso, desempenha o papel central na geração de chamadas, afectando de diferentes formas, o modo como os números são seleccionados. De seguida, é abordado o modo como os números são seleccionados, bem como diversas questões que poderão surgir.

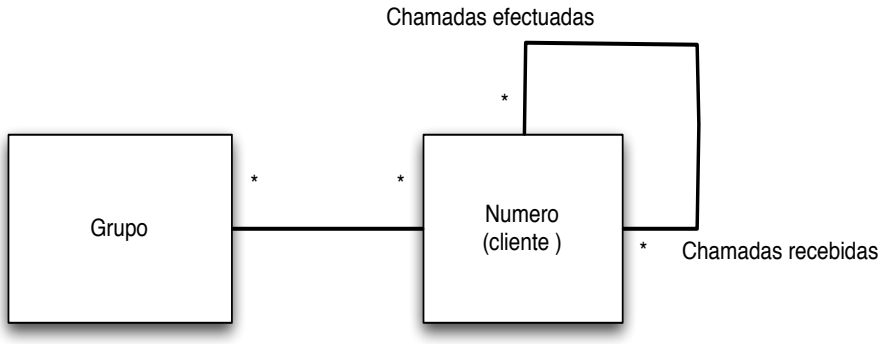


Figura 4.14: Modelo de dados

Este último problema é agora abordado, sendo para isso consideradas várias das questões que podem surgir no contexto do mesmo.

Distribuição das chamadas pelo clientes A geração de chamadas é regida, por omissão, por uma distribuição *power law*, sendo este factor parametrizável. Significa isto, que existe um pequeno grupo de clientes que efectua um grande número de chamadas enquanto a maioria dos clientes as efectua em pequeno número. Qualquer correlação entre o número médio de chamadas que um cliente faz, e os serviços normalmente associados a esse número é, para já, ignorado.

Factores de escolha do destino de uma chamada Para cada chamada gerada no sistema, o número de destino será seleccionado segundo, uma de três opções: 1) um número para o qual o cliente ligou recentemente ou então um número do qual ele recebeu uma chamada; 2) um número escolhido entre os vários grupos a que o cliente pertence; 3) um número escolhido de forma aleatória. O método de selecção é parametrizado pelo utilizador da plataforma de teste.

Abordando os factores temporais, quando o número de destino é seleccionado com base no histórico, opta-se, nesta primeira fase, por uma escolha aleatória entre os números das chamadas efectuadas e recebidas. Características extra, como a selecção com base na posição do número da lista são para já ignoradas.

Em relação ao efeito dos grupos estes são aqui usados para a modelação de diferentes períodos de uso. Durante o período diurno, um cliente irá, com maior probabilidade, efectuar chamadas por motivos profissionais, enquanto no período nocturno, efectua chamadas por motivos familiares/pessoais. Mesmo que esta suposição não seja verdadeira, representa o cenário comum onde um grupo influência a escolha do número de destino, de diferente forma ao longo do tempo. Assim, cada grupo terá, em diferentes rodas de execução, aquilo a que podemos chamar intensidade ou relevância, que varia ao longo de um intervalo de tempo, como pode ser observado na Figura 4.15.

Optando por realizar o *benchmark*, de modo a acomodar cenários de execução como o *workload* Dia ou o *workload* Noite, para cada um deles calculamos o conjunto de números para os quais o cliente terá maior probabilidade de ligar durante esse teste. Por outras palavras, se o cliente pertencer a dois grupos A e B, e se o grupo A tiver maior relevância que o grupo B, então a lista de números para os quais o cliente poderá ligar durante o teste irá conter mais números do grupo

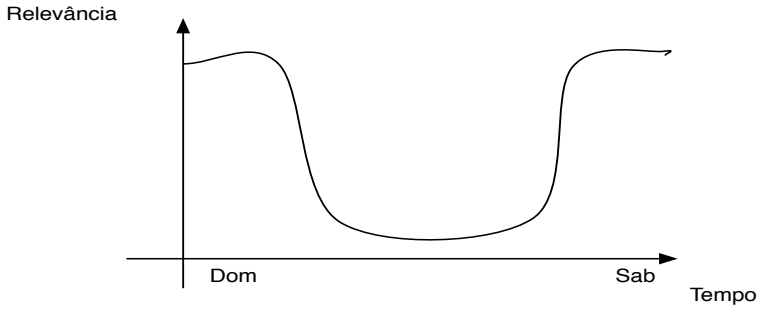


Figura 4.15: Relevância do grupo

A em detrimento do *B*.

Também são gerados grupos virtuais para retratar um elevado número de chamadas para um número em particular, por exemplo um programa de televisão tipo Ídolos ou festival da canção.

4.4.2 Ambiente de execução

Após a descrição das diferentes implementações e um descrição genérica da plataforma, segue-se agora a caracterização do ambiente de execução. Desse modo, descreve-se em seguida qual a população utilizada, *workloads* executados e alguns dos pormenores de configuração que ajudam a compreender os resultados.

Tamanho da População

O primeiro dos parâmetros que caracterizam os cenários de teste, é o tamanho das populações de clientes. Existem 3 cenários distintos: um pequeno com 70 000, um médio com 7 000 000 e um grande com 70 000 000 clientes, que doravante serão designados por população pequena, média e grande, respectivamente. Estes 3 cenários distintos pretendem simular os diferentes mercados, em que a empresa PT se encontra envolvida. No entanto, na execução são no entanto apenas utilizados os cenários pequeno e médio, devido a limitações de hardware.

Serviços existentes

Outro parâmetro que caracteriza os cenários de teste são os serviços que existem e qual a sua abrangência sobre a população. Para tal, foram reunidos todos os serviços exemplo da descrição do projecto, sendo que são apresentados dois serviços comunidade distintos e adiciona-se um serviço de voz base que cobre toda a população. Temos assim o serviço Comunidade, um serviço simples que simula produtos como o tarifário TMN Moche[3] e outro que possui propriedades dinâmicas e ao qual chamamos de serviço Golo. O serviço com saldo partilhado é aqui apelidado de serviço Trabalho. Segue-se a lista dos serviços e a sua cobertura sobre a população, em parte baseados em estatísticas fornecidas pela operadora.

- Voz: 100% da população

[3] http://www.tmn.pt/moche.html

- Comunidade: 10% da população
- Família: 50% da população
- Trabalho: 5% da população
- Golo: 1% da população

Sem a implementação de lógicas de exclusividade ou incompatibilidades entre serviços, note-se que na população final nada se conhece sobre quantos ou quais os serviços que cada cliente possui.

Workloads

No contexto da plataforma foram criados diferentes cenários de execução para a simulação de períodos de uso distintos, onde carga e tipo de pedidos variam. As características que definem os diferentes cenários de execução são:

Tipo de estímulos Define-se para cada *workload* quais os tipos de contactos que se consideram existir. Este pode ser um parâmetro global, definindo um tipo único, como o usado nos cenários testados, onde apenas se consideram chamadas de voz. Se pelo contrário o utilizador da plataforma optar assim por vários tipos de estímulos, então este terá de definir quais as suas distribuições sempre que um cliente escolhe comunicar com uma pessoas aleatória ou para um elemento do grupo.

Escolha da origem do estímulo Como descrito anteriormente neste capítulo, o destino das chamadas ou mensagens SMS pode ser seleccionado de uma de três opções: o resultado de uma interacção com histórico de chamadas do cliente; do seu grupo de influência; ou simplesmente escolhido aleatoriamente. Cada uma destas opções tem um peso e opções associadas, que nos testes realizados foram previamente definidos e mantidos para os diferentes cenários:

Grupos Peso: 40%.

Aleatório Peso: 20%. O utilizador terá aqui de definir qual o probabilidade de fazer uma chamada de voz ou enviar uma SMS quando um tipo global não for definido.

Histórico Peso: 40%. Existe ainda uma diferença no histórico entre a caixa de entrada e saída, que embora seja definível, nos testes realizados é ignorada tendo assim cada uma delas uma probabilidade de escolha de 50%

Relevância dos grupos Como explicado na secção sobre a plataforma, cada utilizador possui aquilo a que se chama um grupo de influência, grupos estes constituídos por clientes com os quais o utilizador terá maior probabilidade de comunicar no intervalo de tempo que o cenário de execução pretende simular. Cada um dos utilizadores está inserido em vários grupos distintos (trabalho, família, etc). Esta medida de quais os utilizadores que pertencem ao seu grupo de influência é determinada pela relevância destes mesmo grupos. Em cada grupo é definida a sua importância, sendo que o valor atribuído é um peso genérico, pois não se sabe à partida quais os grupos a que o cliente pertence. Isto é, se se define que o

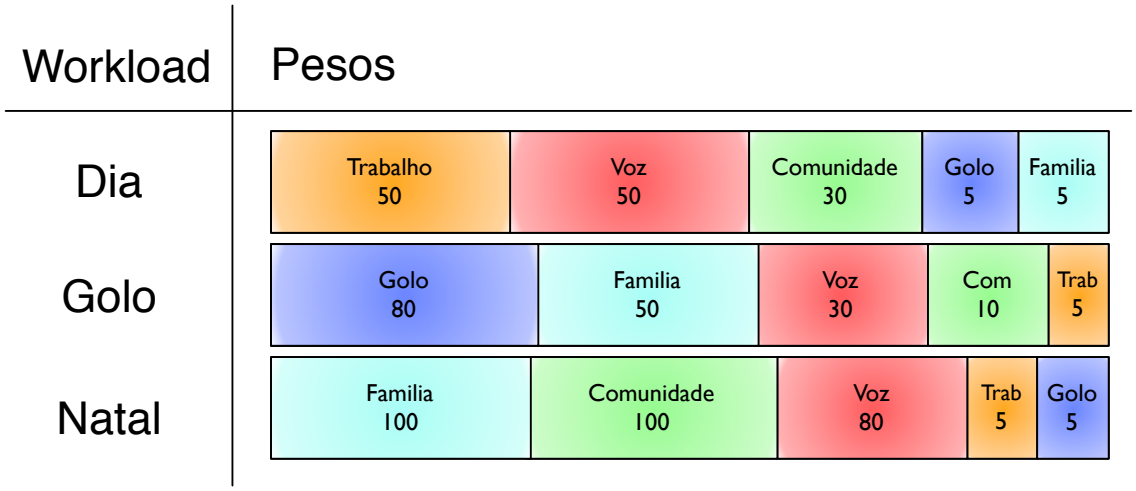


Figura 4.16: Relevância dos vários grupos em cada workload.

grupo voz tem uma importância de 50% e o família 50%, então pertencendo aos dois, o cliente será influenciado por números de ambos os grupos com igual probabilidade, mas se pertencer apenas ao primeiro, todos os seus contactos de influência virão deste.

Assim, foram criados 3 *workloads* distintos para o teste na plataforma de simulação, diferindo entre si na relevância de cada um dos grupos. A distribuição dos pesos de cada serviço pelos diferentes *workloads* pode ser visualizado na Figura 4.16

Dia Este cenário de execução representa o tráfego normal durante o dia, dando uma baixa importância à família em detrimento do grupo trabalho, que se assume estar mais activo. As chamadas simples de voz e comunidade permanecem também relevantes.

Golo Este cenário representa a altura do dia em que um jogo ocorre, e embora a sua cobertura seja pequena, o serviço Golo ganha aqui relevância. Tratando-se, sobretudo, de fins de semana ou horas da noite, o serviço família ganha também aqui relevância, ao contrário do serviço trabalho.

Mas não só este aumento da relevância é utilizado. Para aumentar o uso do serviço e assim simular aquele espaço do tempo onde muitos dos utilizadores acedem à plataforma devido a ele (ainda que a sua cobertura na população seja baixa), é forçado no sistema que cerca de 50% das chamadas recaiam no mesmo.

Natal Este cenário representa a altura do Natal, onde não só as relevâncias dos serviços se alteram mas também outro parâmetros são ajustados na plataforma para aumentar a carga no sistema. Neste *workload* ganham importância os serviços de comunidade e família, sendo que o serviço voz normal permanece também ele relevante, pois assume-se serem estes os serviços mais activos nesta altura

Parâmetros globais de execução

Também existem outros parâmetros que desempenham um papel importante aquando da execução do plataforma de simulação. Entre os bases podemos encontrar:

Número de chamadas O número de chamadas realizadas;

Número de clientes de execução O número de clientes simultâneos, que permitem definir, junto com outros parâmetros, a carga sobre a base de dados;

Tempo de espera O tempo de espera entre a realização de chamadas em cada um dos clientes de execução.

Tempo de chamada O tempo de uma chamada;

O número de chamadas simultâneas Parâmetro de ajuste do número de chamadas que um cliente pode estar a executar ao mesmo.

Nesta secção são apresentados os resultados dos testes efectuados a ambas as plataformas. Estes cobrem os três *workloads* definidos, sendo executados sobre as populações média e pequena, com a recolha de medidas de latência e débito tanto das chamadas em geral como das operações que as constituem. A população grande de 70 milhões de clientes não foi testada devido a limitações de hardware.

Para a execução de tais cenários, é utilizada uma máquina de 8 cores de 16 GB de RAM para a execução da plataforma de testes, sendo a base de dados colocada numa máquina com 24 cores, 128 GB de RAM e armazenamento baseado numa HP HSV300 Eva Storage com RAID1 e FC (FiberChannel) HBA 8Gbps. Em termos de rede as máquinas encontram-se ligadas por uma rede Gigabit. Quanto ao sistema operativo, trata de Linux, Ubuntu Server com kernel 2.6.24-28-server e versão do Cassandra é a 0.7.4. Esta bases de dados seriam idealmente executada num cluster de máquinas relativamente idênticas entre si, mas novamente, por falta de recursos tal não foi possível.

4.4.3 Resultados

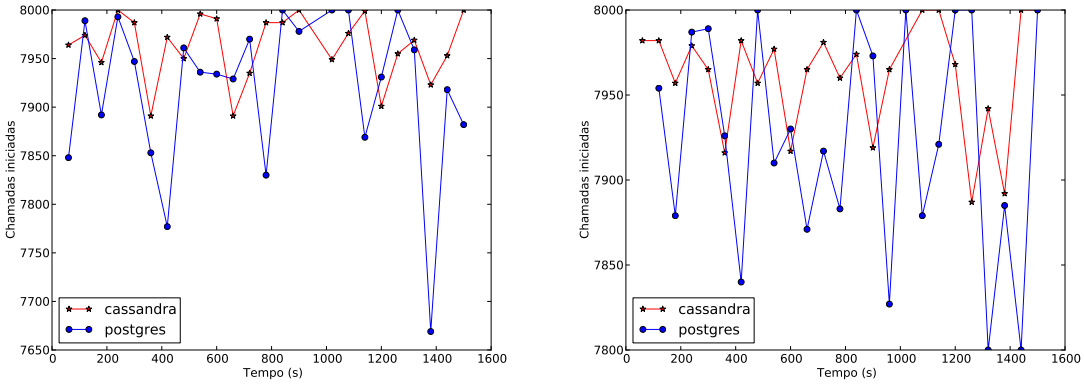
População pequena - 70 000 clientes

No cenário pequeno, construído para a simulação de um pequeno operador de telecomunicações, foram executados dos três *workloads* predefinidos o Dia e o Golo, tentando-se o mais possível simular o número de chamadas concorrentes que se espera ver nestes. O *workload* Natal acabou por não ser testado pois sendo o a população pequena, a elevada concorrência para a qual este foi desenhado faz com que as chamadas a realizar fosse esgotadas em meros segundos não sendo os resultados válidos para análise. Assim, para este dois *workloads* assume-se uma utilização de 60% ajustando-se para tal o número de clientes, o tempo de chamada e o intervalo entre entre a iniciação das mesmas. Os valores definidos estão presentes na Tabela 4.1.

Com estes valores, o número máximo teórico de chamadas iniciadas pelos clientes em um 1 minutos é de 8 mil chamadas, sendo o número máximo de chamadas concorrentes no sistema igual a 40 mil, perfazendo uma taxa de utilização de 60%.

Parâmetros	Workload Dia / Workload Golo
Número de clientes de execução	200
Número de chamadas concorrentes por cliente de execução	200
Tempo de chamada (min)	5
Intervalo entre chamadas (s)	1.5

Tabela 4.1: Parâmetros de execução numa população pequena



(a) (b)

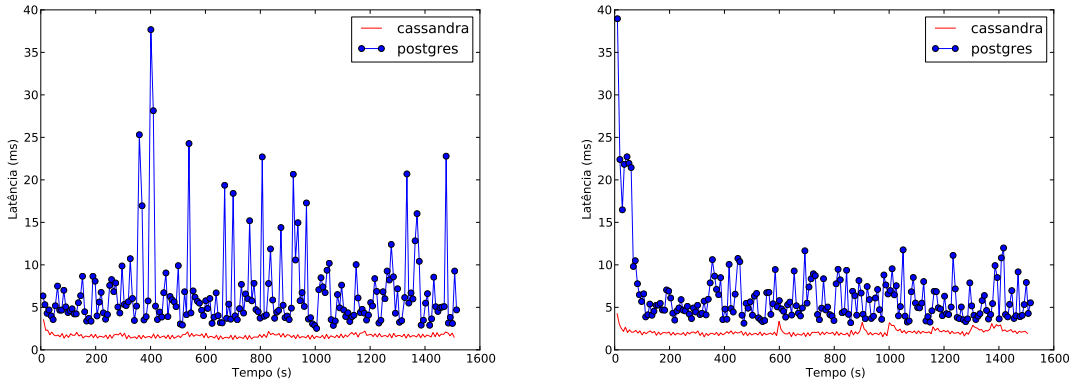
Figura 4.17: Chamadas iniciadas por minuto numa população pequena num workload a) Dia b) Golo

Na Figura 4.17 podemos ver o número de chamadas efectuadas na plataforma, sendo os valores geralmente limitados não pela base de dados, mas sim pelos parâmetros que caracterizam este cenário. De facto, em alguns pontos o número de chamadas está muito perto do limite teórico demonstrando que a latência de estabelecimento de uma chamada é extremamente reduzida.

Olhando para o tempo de estabelecimento de chamadas, podem-se observar algumas diferenças, como se pode constatar na Figura 4.18 e 4.19. Aqui nota-se uma diferença, embora mínima entre as soluções nos dois cenários. De facto, nestes cenários onde o tráfego é menor o Cassandra consegue obter melhores valores.

Entrando também para as medidas base do funcionamento do sistema, mostra-se agora as latências na leitura e actualizações do saldo apresentadas na Figura 4.20 onde se nota uma clara diferença entre as soluções. Note-se no entanto, que neste caso em especial, os valores não são directamente comparáveis, uma vez que as garantias dadas pelo PostgresSQL em termos transaccionais, são no Cassandra inexistentes.

Neste gráfico também se evidencia uma característica da plataforma, que é a inexistência de alterações ao saldo durante o decorrer das chamadas, sendo que de facto existe apenas uma destas operações no início de cada chamada. Tal, resulta do facto que nesta fase do projecto o foco encontra-se no estabelecimento das chamadas, que é o onde o modelo não relacional pode realmente ser aplicado, passando as actualizações de saldo para segundo plano.



(a) (b)

Figura 4.18: Tempo de estabelecimento das chamadas ao longo do tempo para uma população pequena num workload a) Dia b) Golo

Parâmetros	Workload Dia / Workload Golo	Workload Natal
Número de clientes de execução	160	160
Número de chamadas concorrentes por cliente de execução	20000	20000
Tempo de chamada (min)	5	5
Intervalo entre chamadas (s)	0.01	0

Tabela 4.2: Parâmetros de execução numa população média

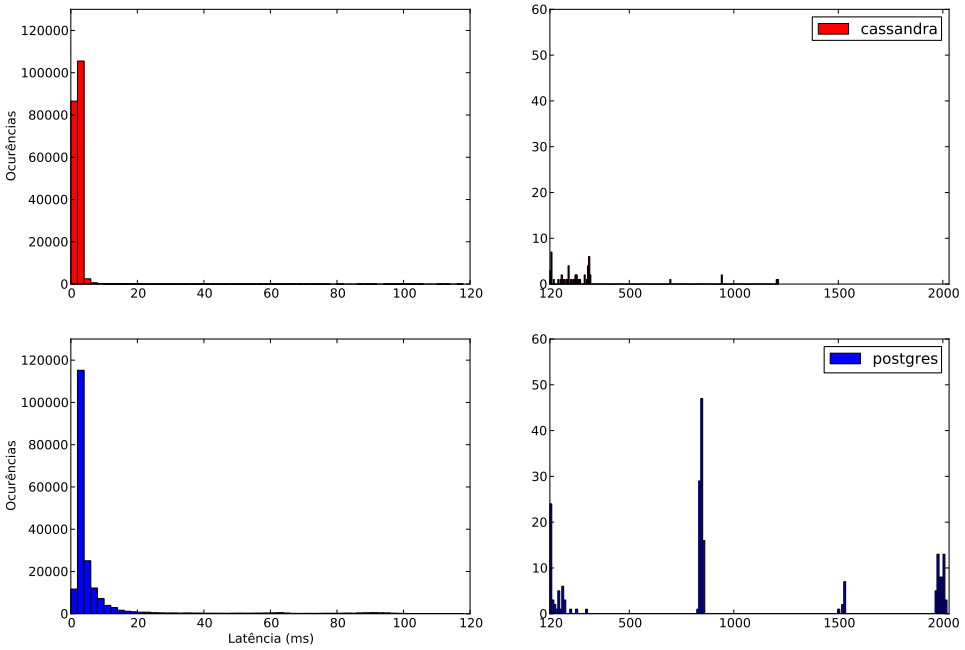
População média - 7 000 000 clientes

No cenário médio, utilizado para simular um cenário equivalente ao nosso país, o número de utilizadores aumenta significativamente. De facto, existe alguma dificuldade na simulação do número de chamadas que se supõem existir nos diferentes *workloads*, mas não obstante, esta é uma população que permite ver como reagem as soluções a um aumento de carga.

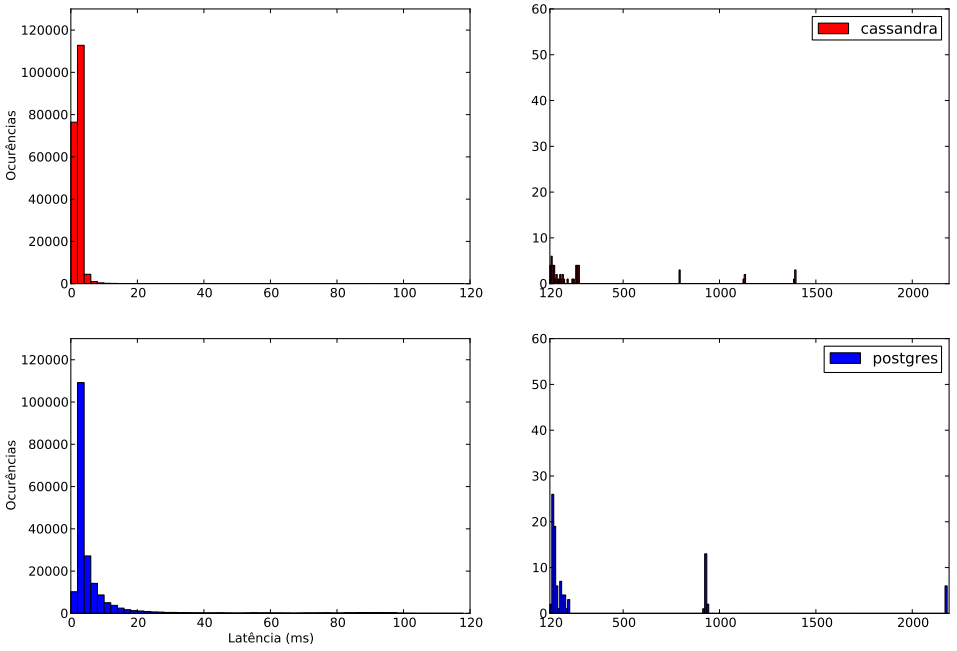
Novamente com diferenças entre os diferentes *workloads*, o objectivo no cenário médio é atingir uma taxa de utilização de 70%, valor no entanto nunca alcançado. Para este efeito os valores de execução utilizados são os representados na Tabela 4.2.

Para estes valores, o máximo teórico de chamadas iniciadas a cada minuto é de 960 mil, e ao fim dos 5 min atingiria-se um máximo de 4,8 milhões de chamadas concorrentes ($\tilde{7}0\%$ de utilização). No entanto, como podemos ver pela figura 4.21 tal valor não é alcançando devido à latência que está associada a cada estabelecimento de chamada. Neste caso, nota-se novamente uma clara superioridade do Cassandra, onde o número de chamadas iniciadas a mais que a versão relacional é de dezenas de milhar. No cenário Natal onde os parâmetros são ainda menos limitativos o Cassandra é a única das bases de dados que consegue aumentar o débito de chamadas.

Em termos de latência, num cenário normal Dia, embora os resultados do Cassandra sejam relativamente mais baixos, os tempos de estabelecimento das chamadas são equivalentes. O mesmo não se passa nos cenários de maior carga como o Natal onde se nota um agravamento nas latências

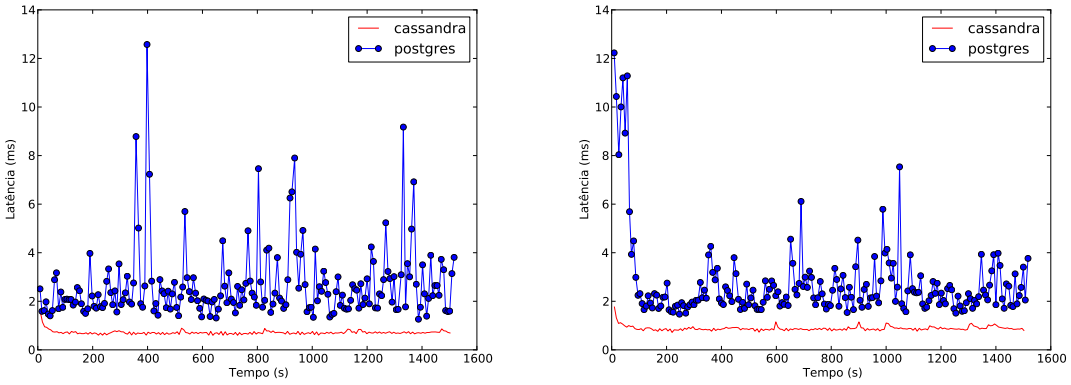


(a)



(b)

Figura 4.19: Distribuição do tempo de estabelecimento de chamada numa população pequena num workload a) Dia b) Golo



(a) (b)

Figura 4.20: Tempo de leitura e actualização do saldo para uma população pequena num workload a) Dia b) Golo

em PostgreSQL enquanto as do Cassandra se mantêm mais baixas e constantes como se vê na Figura 4.22 e 4.23.

Finalmente em termos de latências nas actualizações de saldo os resultados são equivalentes aos obtidos na população pequena. Tal pode ser visto na Figura 4.24.

4.5 Sumário

Neste capítulo descreve-se um caso realista de migração. Descreve-mos assim como um sistema de chamadas que foi originalmente implementado numa solução relacional foi portado para Cassandra e quais as alterações que este teve de sofrer. No final apresentamos os resultados de execução que permitem comparar ambas as soluções sobre um sistema que tenta simular padrões reais de chamadas.

Numa análise resumida dos resultados, podemos de facto afirmar que o Cassandra obteve melhores valores, levantando-se no entanto a questão sobre quais os motivos para tal. A resposta está na base de dados claro, mas não tanto na sua implementação, ou protocolo de comunicação, esta está sobretudo no paradigma de desenvolvimento que a ela está associada. Um dos elementos que mais afectou os resultados foi a ausência de garantias transaccionais no Cassandra. De facto mesmo nos casos onde a latência em Cassandra não é significativamente menor nas operações de estabelecimento, existem ainda assim um maior débito de chamadas nesta solução. Tal se deve á operação de actualização de saldo que acaba por funcionar como ponto de contenção na execução. Esta é também ao mesmo tempo a razão pela qual este motor de dados nunca poderia representar só por si uma solução.

Outro dos factores é que bases de dados como o Cassandra, com a sua base em motores de base de dados simples, obrigam a migrar o código aplicacional de volta à camada de cliente. Por este motivo, tal paradigma obriga assim uma optimização das estruturas de dados com a desnormalização dos dados e pré materialização dos resultados a extrair levando no caso de estudo

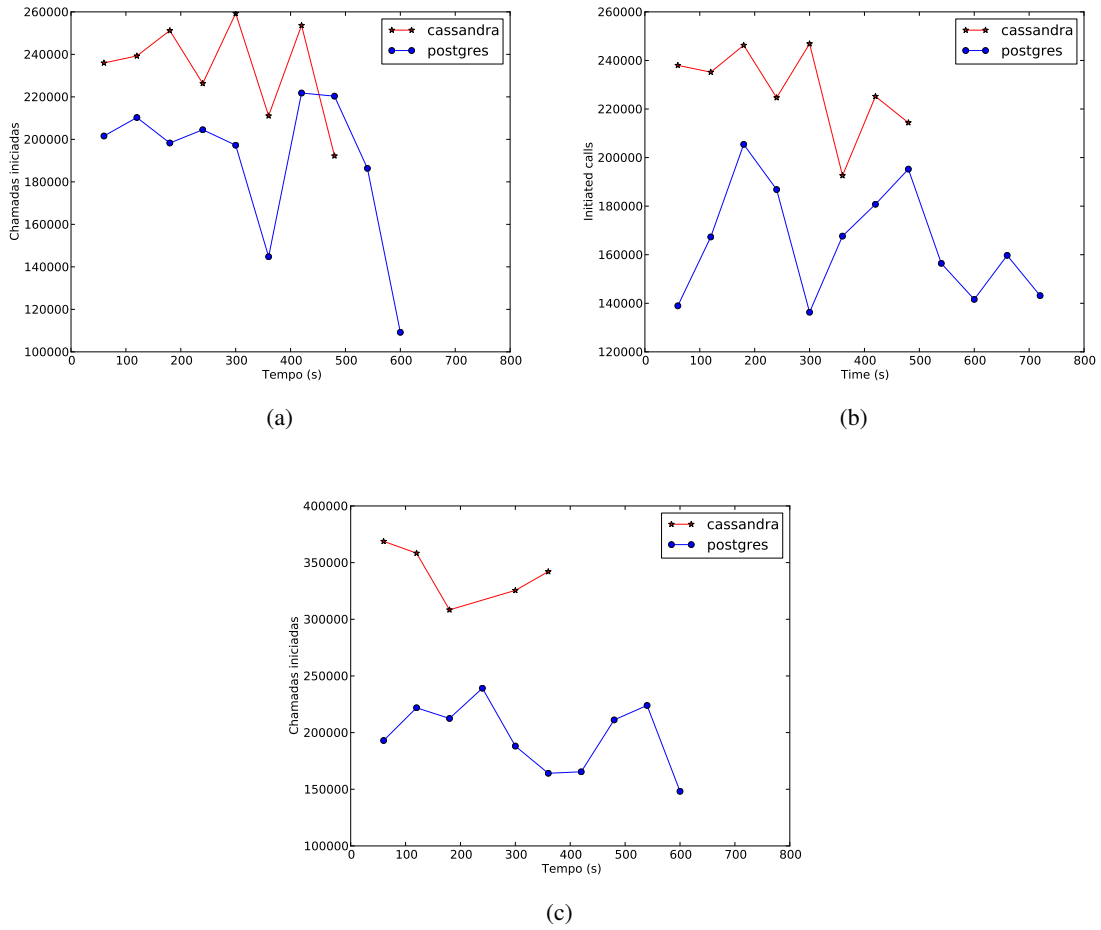


Figura 4.21: Número de chamadas iniciadas por minuto numa população média num workload a) Dia b) Golo c) Natal

a menores latências e um maior número de chamadas realizadas por minuto.

Poderia-se argumentar que o mesmo poderia ser realizado na implementação relacional. Embora possível, outras soluções existem, tal como a replicação da base de dados, uma vez que se trata de um cenário onde predominam as leituras, ou a adição de *caches* externas para as operações mais simples. Para realmente entrar na zona da desnormalização dos dados e o abandono de *Joins* e chaves estrangeiras é imperativo analisar o que se perde neste modelo, não correndo assim o risco de estar a recriar mais uma solução com a marca NOSQL mas sem as suas vantagens, como natural escalabilidade horizontal ou as suas arquitecturas resilientes.

Mas se esta análise é neste ponto mais favorável ao paradigma não relacional, a plataforma precisa de ser executada num ambiente verdadeiramente distribuído para testar a capacidade do sistema para escalar progressivamente. De facto o verdadeiro valor de uma implementação baseada em Cassandra nunca seria unicamente o seu desempenho mas sim as vantagens que este traria ao sistema em termos de adaptabilidade a vários cenários e a redução dos custos associados.

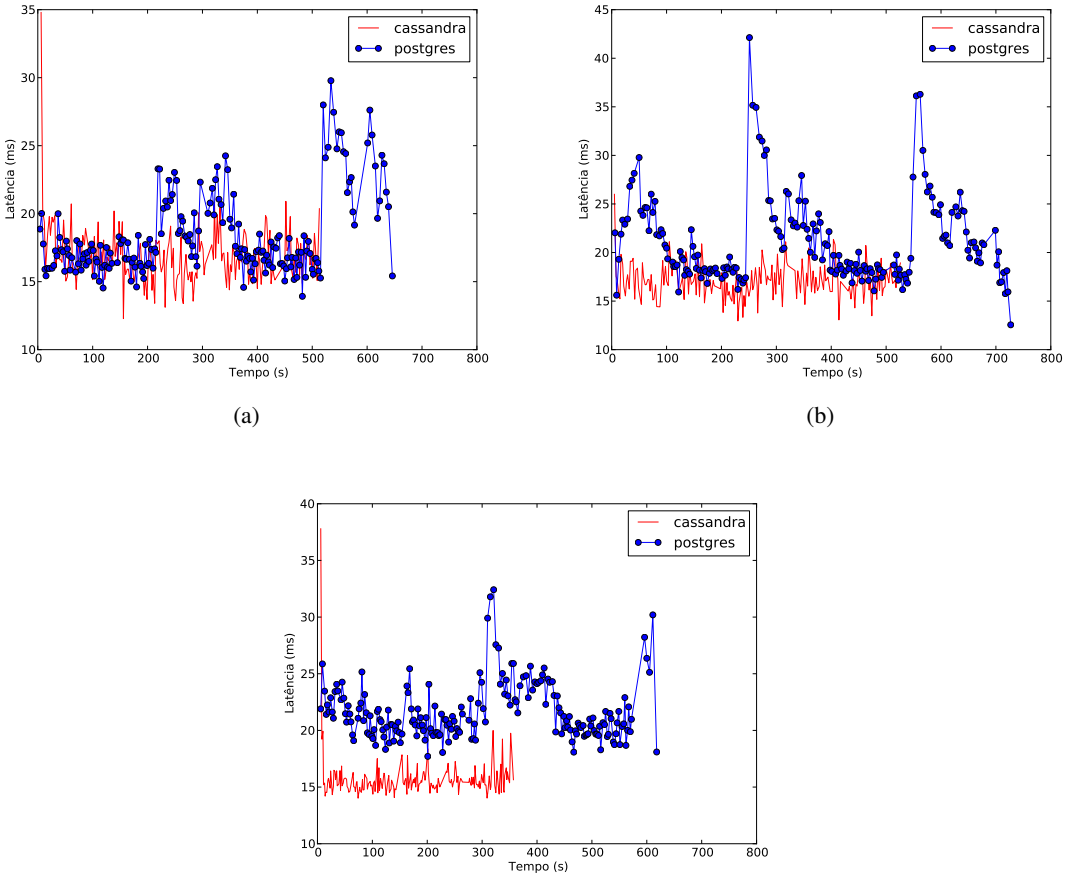
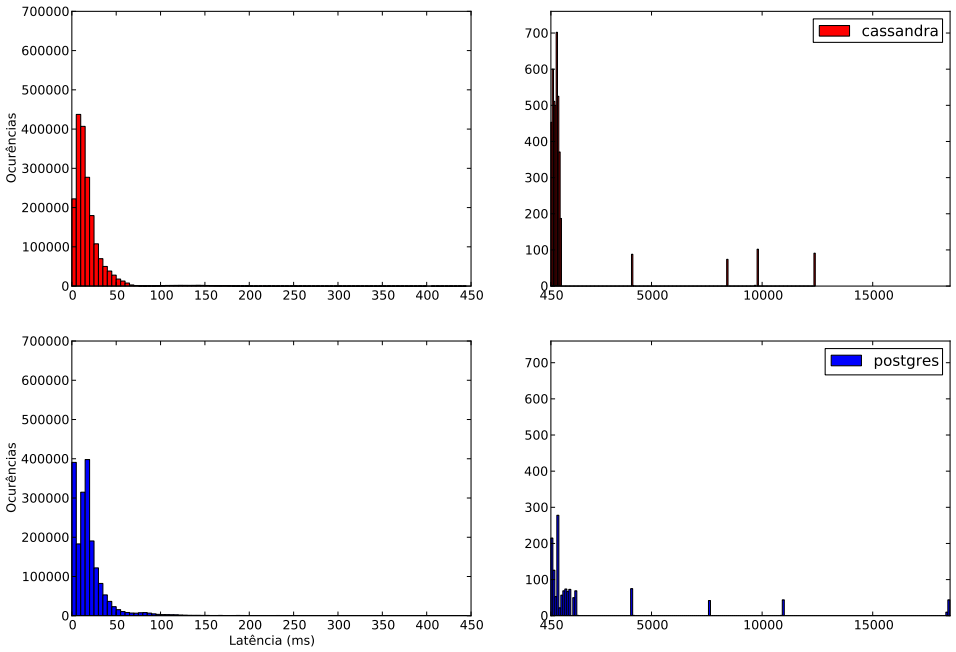
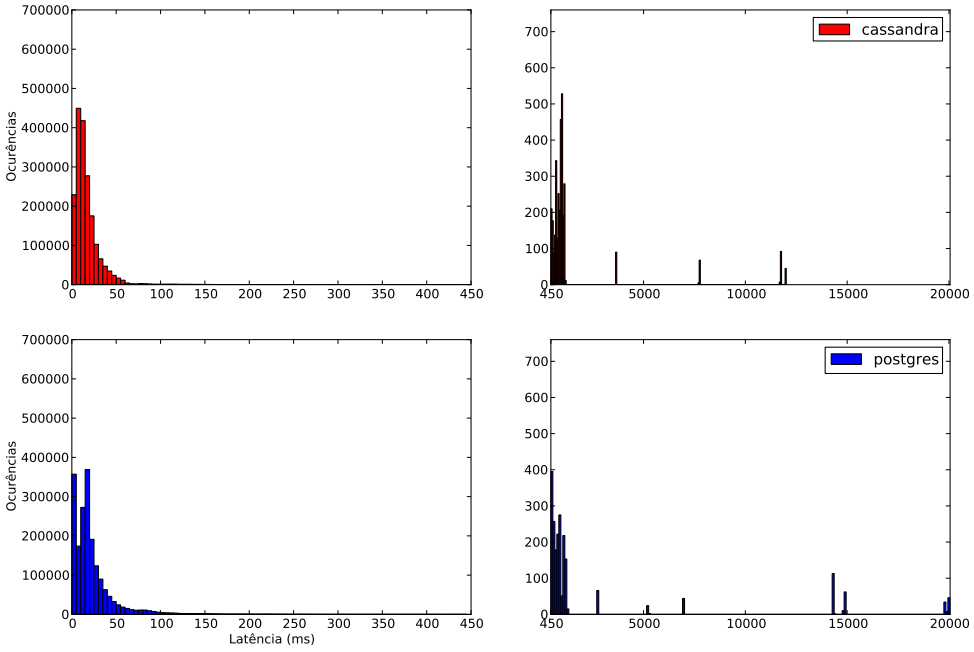


Figura 4.22: Tempo de estabelecimento das chamadas ao longo do tempo para uma população media num workload a) Dia b) Golo c) Natal

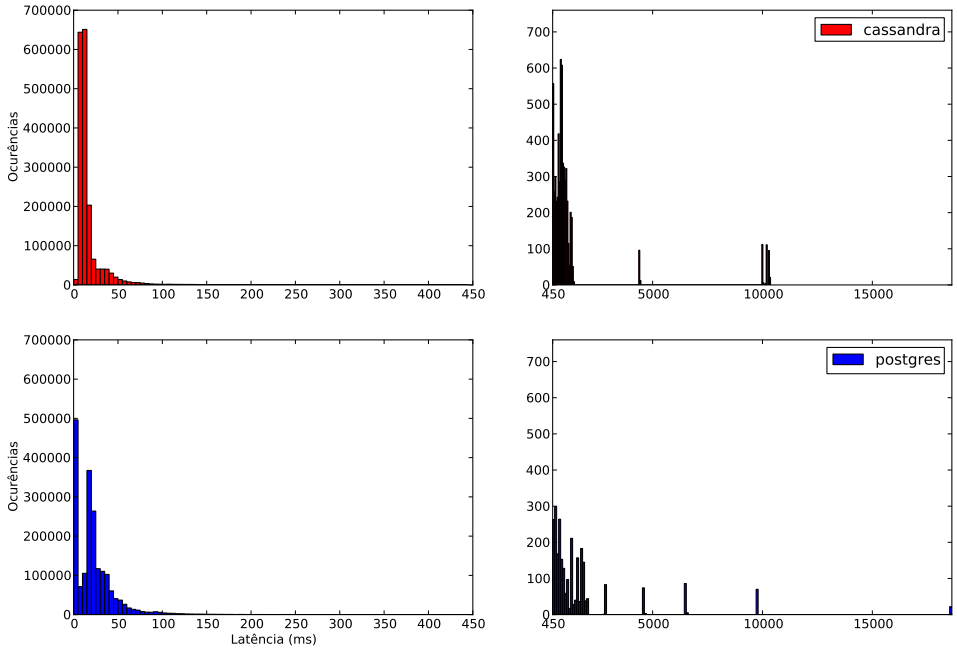


(a)



(b)

Figura 4.23: Distribuição do tempo de estabelecimento de chamada numa população media num workload a) Dia b) Golo c) Natal



(c)

Figura 4.23: Distribuição do tempo de estabelecimento de chamada numa população media num workload a) Dia b) Golo c) Natal

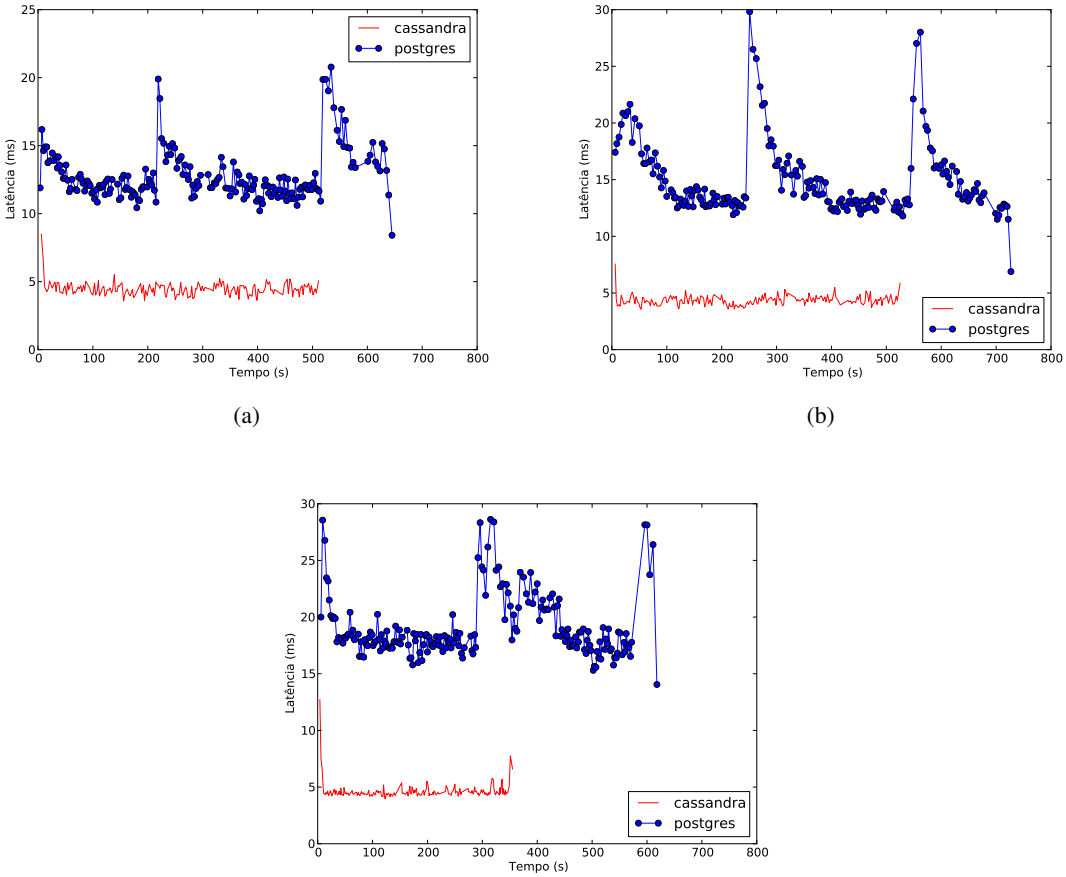


Figura 4.24: Tempo de leitura e actualização do saldo para uma população media num workload a) Dia b) Golo c) Natal

Capítulo 5

Interface baseada em objectos

No decorrer do trabalho desenvolvido, muitos são os factores que nesta mudança de paradigma afectam negativamente o trabalho do programador. Nesta nova realidade, elementos como as estruturas de dados com múltiplos níveis podem na verdade ser um verdadeiro desafio para muitos programadores com uma mentalidade relacional. Do mesmo modo, as interfaces de programação, como retratado neste documento, podem ser limitadas e de difícil manipulação quando comparadas com linguagens de pesquisa como o SQL.

Como resposta a este desafio, desenha-se neste capítulo uma possível solução. Neste contexto apresenta-se uma nova solução de mapeamento de objectos para Cassandra em Java, criada para a diminuição do impacto desta mudança de paradigma e redução dos tempos de desenvolvimento. Esta dará assim aos programadores uma ferramenta que lhes permitirá o uso da base de dados de uma forma similar ao modo como outras soluções como o Hibernate ou Datanucleus o fazem para as bases de dados relacionais. Criada com base no conhecimento que foi sendo recolhido nos dois processos de migração que nesta dissertação foram realizados, surgiu assim o COM (*Cassandra Object Mapper*).Tal solução permite deste modo construir uma ponte entre o paradigma de desenvolvimento em objectos e modelo de dados do Cassandra

5.1 Motivação

O desenvolvimento desta interface baseada em objectos endereça os seguintes desafios:

Novos conceitos: São vários os novos conceitos que são apresentados aos utilizadores quando estes adoptam o Cassandra como solução. Entender o modelo, as suas características e o modo como este se relacionam como o paradigma de programação pode de facto se problemático numa primeira fase. Este é um dos problemas que abordamos, pois com esta solução o programador poderá focar-se apenas no modelo de objectos.

O mesmo não se pode no entanto dizer sobre os modelos de coerência característicos destas bases de dados. Embora sejam simplificados os mecanismos de atribuições de marcas temporais e a definição de níveis de coerência, não é aqui apresentado qualquer mecanismo de suporte a transacções ou semelhante. De facto, a este nível o programador deverá tentar compreender e ter em mente as características e limitações deste género de sistemas.

Código: O uso de bibliotecas de baixo nível como o Thrift permite ao programador uma plena integração do seu código com a sua linguagem de preferência tal como o controlo das diversas características que marcam o Cassandra. Estas levam no entanto a uma elevada verbosidade e complexidade do código, fenómeno agravado pelas operações que são passadas para o lado do cliente devido à interface simples do Cassandra.

Como seria espectável, com o tempo novas bibliotecas surgiram em diversas linguagens que assim permitem uma melhor integração. Contudo, estas falham por vezes em abstrair muitas das características inerentes à base de dados como é o caso do mecanismo de serialização do Hector. De um modo diferente, o CQL, linguagem de pesquisa com base no SQL apresenta também algumas limitações impostas pelo seu cliente de JDBC. De facto, quando adaptada a uma linguagem por objectos, acabamos por enfrentar os mesmos desafios de impedância entre modelos que as soluções de ORM procuram combater, sem usufruir de todas as capacidades de pesquisa que estão presentes no SQL mas não nesta. Propomos assim uma abordagem que permita facilitar o processo de persistência e recuperação dos dados de uma forma completamente transparente.

Integração: O último desafio que aqui abordado advém de casos como o apresentado no capítulo anterior. De facto, quando perante um cenário onde se pretende migrar aplicações já existentes e todo o conhecimento a elas associado para este novo paradigma, qualquer organização ou simples programador dificilmente tomará tal decisão. De facto a rescrita do código da aplicação e todo o processo associado tornariam os custos inaceitáveis.

Com uma interface padrão de persistência, a nossa proposta permite que este processo seja feito de forma mais fácil em soluções já baseadas em mecanismos de ORM. Mesmo que este não seja o caso, esta solução facilita aqui uma migração progressiva de apenas parte do código. Quando sobre uma aplicação que necessite de garantias de coerência em parte dos seus dados, o programador pode facilmente na mesma base de código redireccionar parte dos objectos para uma solução com garantias transaccionais, permanecendo os restantes numa solução escalável como o Cassandra.

5.2 Mapeamento de objectos em Cassandra

Apresenta-se agora a solução desenvolvida para o mapeamento de objectos sobre Cassandra. Esta foi desenvolvida como uma ferramenta para a construção de aplicações de uma forma simples mas escalável, fornecendo uma abstracção do modelo de dados e API subjacentes.

5.2.1 Mapeamento de objectos

No desenvolvimento de uma plataforma de mapeamento por objectos, o principal desafio passa pela construção de métodos eficientes de leitura e persistência que mantenha na base de dados as relações entre as diversas entidades. Para tal propósito existem hoje duas especificações padrão que definem para a linguagem Java o modo como estas operações deverão ser processadas: a especificação Java Data Objects (JDO) e a Java Persistence API (JPA). Estas duas especificações, ambas suportadas pela plataforma DataNucleus, definem uma série de regras sobre o mapeamento de classes, persistência e leitura de informação. Nas camadas de baixo nível da plataforma, as diferentes implementações, como a aqui apresentada, são responsáveis pelo modo como estas

operações são executadas na base de dados.

Este processo não consiste somente na implementação de simples operações de leitura ou escritas, uma vez que é da responsabilidade das plataforma efectuar o mapeamento das diversas entidades no sistema e das suas relações. Estas que podem variar em direcção e cardinalidade (um-para-um, um-para-muitos, etc.) têm de ser na plataforma tratadas com base em diferentes estratégias. Descrevendo estes diversos mecanismos, apresentamos agora alguns dos passos do processo de mapeamento.

Mapeamento de classes De modo similar ao que acontece nas soluções para o modelo relacional, a base do processo de mapeamento passa também pela aqui pela persistência de cada uma das diferentes entidades num família de colunas, sendo os seus campos guardados em colunas individuais. Esta estratégia, para além da sua simplicidade, permite um claro reconhecimento da estrutura dos dados na base de dados, facilitando a sua posterior extracção. Em termos de hierarquia, apenas se suporta a solução onde cada uma das classes é persistida na sua própria família não existindo ainda uma forma de pesquisar os dados pela classe pai.

Relações um-para-um/muitos-para-um Em relações de um-para-um, as instâncias de uma classe possuem um objecto de uma outra classe (mas apenas um). Este é o caso mais simples, sendo o objecto associado persistido na família de colunas e a sua chave guardada na coluna do campo correspondente. Existem no entanto aqui duas diferenças quando em comparação com o modelo relacional. A primeira, que é comum aos restantes casos, prende-se com a perda das garantias de integridade dos dados impostas pela base de dados. Estas fariam, se uma chave estrangeira fosse usada, que um erro fosse retornado nas remoções de entidade associadas ao objecto. Suporta-se no entanto a noção de objecto dependente, onde aquando da remoção do objecto pai, também o relacionado será eliminado se tiver sido marcado como tal. A outra diferença prende-se com a impossibilidade de suportar relações bidireccionais com uma só das entidades contendo a chave, devendo-se tal limitação à interface de acesso do Cassandra. Assim este tipo de relação só poderá ser estabelecido se ambos os lados tiveram uma referência.

Esta estratégia é usada também em relações de muitos-para-um, onde uma instância de uma classe contém um objecto associado (e apenas um) de uma outra classe como um livro que contém um autor. A diferença aqui é que um autor pode ter mais que um livro e todos eles têm uma referência ao autor. Neste cenário similar ao anterior, fazem-se as mesmas considerações sobre a remoção de objectos. Em termos de ligações bidireccionais, estas têm aqui de ser implementadas do mesmo modo que as relações de um-para-muitos descritas a seguir.

Relações um-para-muitos/muitos-para-muitos Em relações de um-para-muitos e muito-para-muitos um objecto da uma classe A tem varias objectos associados da classe B, sendo que na última, cada instância de B contêm também diversos objectos de A. De forma a mapear tais relações temos assim de formular uma maneira de persistir várias conexões em cada uma destas entidades. Para tal estruturamos três estratégias diferentes que agora delineamos e que podemos também ver na Figura 5.1.

- Utilizar uma família de colunas extra para o mapeamento externo das relações, onde para cada instância, à qual corresponde uma linha, existe colunas que contêm as chaves das entidades associadas.

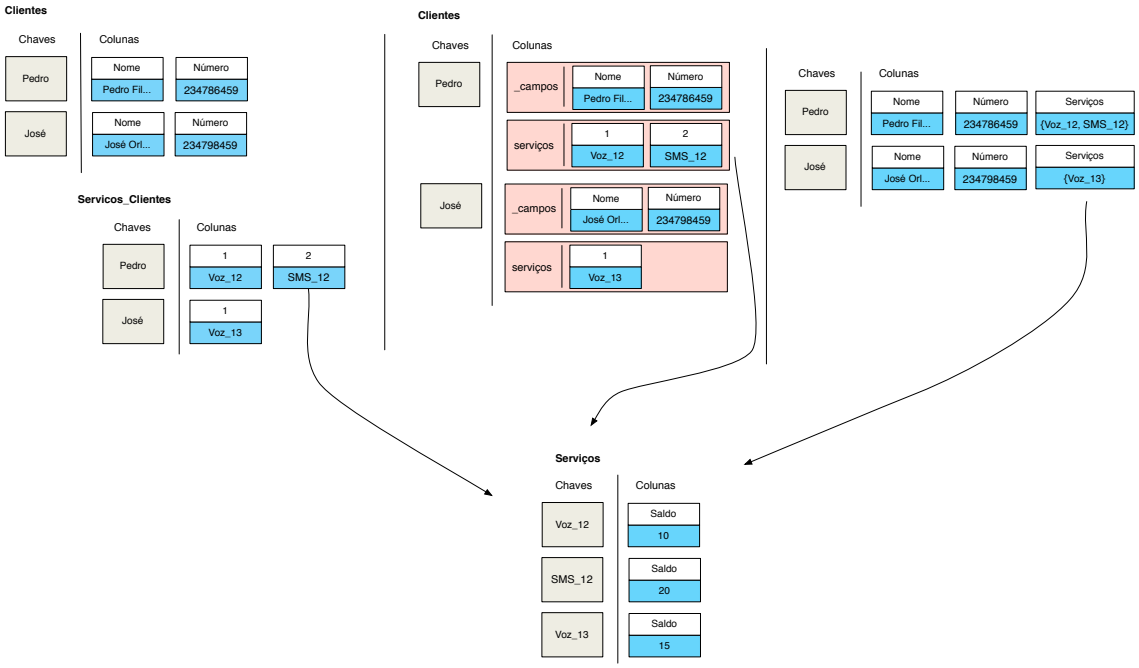


Figura 5.1: Estratégias de mapeamento.

Vantagens: Simples e de fácil implementação sendo que a estrutura é consultável na base de dados.

Desvantagens: São necessárias duas operações de leitura para a materialização das entidades codificadas na relação.

- Persistir toda a informação que se refere à classe em causa numa super família de colunas. Nesta, para cada instância da classe todos os campos base são persistidos numa única super coluna com um nome padrão e distinto dos restantes. As restantes super colunas serão depois usadas para a persistir as chaves dos objectos associados, existindo assim uma destas estrutura por associação.

  Vantagens: Menos ligações à base de dados para a leitura das entidades codificadas nas relações, e a estrutura é também ela consultável na base de dados.

  Desvantagens: Traz complexidade à plataforma e é baseada em super famílias de colunas que implicam por vezes problemas de desempenho.

- Persistir a informação relativa às entidades em causa num campo da família de origem. Não existindo a possibilidade de utilizar uma super colunas de forma isolada numa normal família de colunas, esta informação terá de ser serializada para ser persistida numa só coluna.

  Vantagens: À semelhança da opção anterior esta obriga também um menor número de leituras para a recuperar os objectos associados.

  Desvantagens: A des/serialização dos objectos implica um custo e ainda que este seja mínimo, as relações acabam por ser obscurecidas e não são directamente consultáveis.

Por uma questão de desempenho e simplicidade optamos assim pela terceira opção.

5.2.2 Outros elementos da implementação

No processo de desenvolvimento do *plug-in* de mapeamento, para além das questões de mapeamento, outras se levantam ao baixo nível. A primeira desta é a gestão de conexões, parte importante da plataforma devido à natureza distribuída da base de dados. De facto um só *cluster* de Cassandra pode conter dezenas de nós e por esse motivo incluímos um mecanismo de descoberta automática de nós permitindo á plataforma balancear a carga entre eles.

Outro dos pontos abordados foi a definição dos parâmetros que são característicos do modelo de coerência do Cassandra. Tomando o nível de coerência que tem de ser passado em todas as operações executadas, é aqui utilizado um valor de coerência padrão, sendo o escolhido o critério de quórum. Este parâmetro pode no entanto ser alterado na plataforma mas sempre de forma global, esperando-se que no futuro a interface do JDO permita a sua alteração através do mecanismo destinado aos níveis de isolamento das soluções relacionais. De modo semelhante, as marcas temporais a ser incluídas nas escritas e remoções são também adicionadas de forma automática pela nossa solução.

Por último implementamos também uma parte da linguagem de pesquisa JDOQL. Esta é uma linguagem que tenta capturar o poder de expressão do SQL e aplica-lo à linguagem Java. Como esperado, devido às limitações pelo Cassandra só algumas das operações foram implementadas. Entre estas temos a implementação das leituras sequenciais de todas as chaves de uma classe, de um número definido destas ou então de um conjunto com início e fim. Naturalmente esta opção será apenas útil se as chaves estiverem ordenadas na base de dados porque caso contrário retornará um conjunto aleatório de chaves segundo o limite pelo utilizador definido.

Relacionado também com este ponto adicionamos também à plataforma o suporte para índices secundários que assim permitiram aumentar as capacidades de pesquisa. Numa primeira fase, quando estes não eram ainda suportados no Cassandra, a nossa implementação passou pela criação de colunas extra onde todas as chaves indexadas eram armazenadas. Esta implementação apenas permite no entanto que o utilizador pesquise por uma única chave primária. Na versão 0.7 com a implementação nativa dos índices em Cassandra, este componente foi passado para a base de dados e é agora permitido a pesquisa por mais de um campo.

5.3 Avaliação da solução

De forma a avaliar o desempenho e impacto da nossa implementação, apresentámos dois casos de estudo. Em primeiro lugar abordamos a plataforma de testes TPC-W que é apresentada no Capitulo 3. Esta permite-nos obter medidas de desempenho base e comparar os dois processos de implementação em termos de operações usadas. Numa segunda fase, fazemos o mesmo para a plataforma de chamadas estudada no Capítulo 4. Neste processo demonstramos como o modelo é convertido numa relação entre objectos e damos uma ideia de como poderá ser esta usada numa aplicação real, como elemento de ajuda à migração para Cassandra. Com esta avaliamos também os resultados em termos de latência e débito de chamadas num cenário que pretende simular um caso real de uso

5.3.1 TPC-W

De forma a avaliar as vantagens e desvantagens da solução implementada repete-se agora de forma similar o processo efectuado no capítulo 3. Deste modo é agora apresentado em parte o modo como podemos migrar a implementação do TPC-W proposta para esta nova solução, avaliando o modo como esta nos permite abstrair da camada de dados e assim simplificar o processo de desenvolvimento. São assim abordados os passos de mapeamento da entidades e das relações entre si, tal como o desenvolvimento de algumas das operações. No final é apresentado uma comparação entre os resultados em ambos os modelos.

Implementação

Quando programando com base em Thrift, o programador têm acesso a interfaces em múltiplas linguagens, nas quais se permite a manipulação de muitos dos mecanismo inerentes à base de dados. No entanto, esta implica o domínio de inúmeros conceitos e estruturas que tornam longa a curva de aprendizagem. Embora o mesmo se aplique às soluções de mapeamento de objectos, estas foram desenhadas para a simplificação do código e o conhecimento a elas associado encontra-se largamente difundido.

Ao desenvolver uma aplicação com base numa solução de mapeamento de objectos como a apresentada, o primeiro passo consiste na definição de quais as entidades existentes no sistema. De igual modo é também necessário delinear quais as chaves primárias e relações entre entidades. Com o uso de anotações JDO ou pelo uso de um ficheiro de mapeamento XML, a codificação destes meta dados é relativamente fácil uma vez que os parâmetros base são de rápida aprendizagem. Esta é uma fase em tudo diferente do processo de desenvolvimento associado a soluções não relacionais como o Cassandra. Se neste género de soluções existe uma adaptação do modelo para uma optimização das operações esperadas, é uma decisão arriscada optar por uma plataforma de programação que necessita de uma prévia definição do mesmo.

No entanto, do mesmo modo que famílias de colunas são criadas na base de dados para a optimização de uma operação em específico, também estas podem ser criadas do lado do cliente sob a forma de novas classes. Sendo implementada num plataforma madura a nossa solução permite também o uso de optimizações como os grupos de procura (*fetching groups*) que permite a leitura de apenas alguns campos de um objecto.

Entrando depois no código da aplicação, a simplicidade pela plataforma introduzida começa aqui a ser visível, dado que o cliente apenas tem de invocar um único método para a persistência de um objecto.

```
PersistenceManager interface_persistencia = pmf.getPersistenceManager();

CarrinhoCompras carrinho = new CarrinhoCompras(Id);
carrinho.setData(new Timestamp(tempo));
interface_persistencia.makePersistent(carrinho);
```

O mesmo se aplica aos métodos de leitura que necessitam apenas do identificador do objecto. Se fosse codificada com base em Thrift, tal operação necessitaria de amplo código de serialização, tratamento de excepções, níveis de coerência, etc. Mesmo numa linguagem como o CQL, a

adaptação desta e outras operações similares necessitaria da escrita de uma instrução própria que teria depois de ser adaptada com o uso de JDBC.

```
PersistenceManager interface_persistencia = pmf.getPersistenceManager();

Produto item = interface_persistencia.getObjectById(Produto.class, id);
String nome = item.getAutor().getNome();
```

Operações pequenas como a extracção da informação de um produto torna-se deste modo de fácil desenvolvimento e mesmo as mais complexas como a pesquisa pelos melhores autores ou baseada num índice acabam por ser simplificadas. Para a pesquisa das diversas encomendas usadas para encontrar os autores com mais vendas pode ser codificada por:

```
PersistenceManager interface_persistencia = pmf.getPersistenceManager();

Extent e = pm.getExtent(Encomenda.class, true);
Query instrução_pesquisa = pm.newQuery(e, null);
instrução_pesquisa.setRange(0, 3333);

Object resultado = instrução_pesquisa.execute();

if (resultado == null)
    return;

Collection<Encomenda> encomendas = (Collection) resultado;

//Código de análise
```

Do mesmo modo, a pesquisa por um campo indexado, como o tema de um livro, passa a ser codificado por:

```
PersistenceManager interface_persistencia = pmf.getPersistenceManager();

List<Producto> items = new ArrayList<Producto>();

Extent e = pm.getExtent(Producto.class, true);
Query instrução_pesquisa = pm.newQuery(e, "Tema == \"" + tema + "\"");
Object resultado = instrução_pesquisa.execute();
if (resultado == null)
    return;

Collection<Producto> produtos = (Collection) resultado;

//Código de análise
```

Esta migração exige, no entanto, mais do que a simples mudança do código de persistência ou leitura pois algumas das estruturas de dados tiveram de ser migradas para famílias de colunas normais. Surgem deste modo no código do cliente classes como o carrinho de compras que contêm agora uma lista de produtos, sendo esta informação persistida de acordo com as estratégias acima delineadas. Do mesmo modo alguns dos campos destas entidades contêm agora índices para a sua pesquisa. Esta vertente da migração é objecto de maior detalhe no caso de estudo seguinte.

Ambiente de trabalho

Para a execução destes testes aqui definidos são usadas duas máquinas de teste sobre um *cluster* de cinco máquinas. O hardware consiste em sete máquinas HP com um processador Intel(R) Core(TM)2 CPU 6400 - 2.13GHz, dois Gbyte de RAM e um disco rígido de uso comum SATA (7200 RPM). Todas as máquinas estão ligadas por LAN conectadas a um *switch* de 1GB/s. O sistema operativo é Linux, Ubuntu Server com kernel 2.6.31-1 e o sistema de ficheiros encontra-se em ext4. Para estes testes, realizados numa primeira fase deste trabalho, a versão de Cassandra usada é a 0.6.6 significando assim que não existem um suporte a índices por parte da base de dados e como tal estes são geridos pelo solução de mapeamento.

Em termos de parâmetros de execução, existem 60 (30x2) clientes concorrentes em duas máquinas individuais. Cada um deste efectua 200 pedidos à base de dados de 10.000 produtos com o tempo de espera entre pedidos padrão e um *workload* de encomenda. Cada teste é realizado 5 vezes sendo os resultados uma média destas e as figuras provenientes de uma das rondas.

Resultados

Analisando os resultados presentes na Tabela 5.1 podemos concluir que embora o desempenho piore em algumas das operações mais pesadas, outras aparentam melhorar. Analisando estas duas vertentes, temos na Figura 5.2 uma representação das latências obtidas para operações simples de leitura (*Product Detail*) e escrita (*Buy confirm*). Sobre estas podemos concluir que os resultados obtidos são, para este género de operações, comparáveis. Na verdade os resultados do COM são de facto melhores pois apresentam-se mais estáveis que os obtidos com a interface nativa. Este fenómeno deve-se ao uso de super famílias de colunas e às operações de pesquisa sobre elas efectuadas na versão directamente implementada sobre Cassandra.

Na Figura 5.3 temos a medição de latências com base em operações de leitura de mais que um elemento. Na caso da procura de produtos por campos indexados (Figura5.3a) vemos no entanto que os resultados provenientes da implementação baseada no COM podem ser divididos. Esta diferença que faz com que os resultados desta operação sejam por vezes piores, está ligado aos vários tipos de pesquisas existentes, sendo o caso problemático a pesquisa por tema que retorna muitos mais valores que qualquer uma das outras. Com base nesta operação de pesquisa por tema e também na operação de procura dos produtos mais vendidos (Figura 5.3b) podemos ver as limitações inerentes ao COM. Quando pesquisa como estas retornam um elevado número de elementos a plataforma irá tentar reconstruir cada um destes causando um expectável aumento na latência.

	Cassandra			OCM		
Débito médio (operações/min)	442.6			398.8		
	Latência (ms)			**Latência (ms)**		
Operações	Média	90º percentil	10º percentil	Média	90º percentil	10º percentil
Home	31.98	7	83	21.02	25	15
New Products	9.12	4	12	1348.15	1705	1100
Best Sellers	4218.53	3461	5079	30399.32	35860	24950
Product Detail	10.43	2	18	4.51	4	2
Search	15.24	1	30	425.74	1363	5
Shopping Cart	18.15	3	46	16.33	21	10
Customer Registration	13.41	4	27	10.38	12	6
Buy Request	8.70	2	14	11.01	15	6
Buy Confirm	33.66	9	85	24.65	34	16
Order Inquiry	17.17	2	38	15.94	29	1
Admin Confirm	6446.18	5306	781	84162.50	95584	73636

Tabela 5.1: Resultados para um cenário de encomenda

5.3.2 Sistema de chamadas

Neste segundo caso de estudo, abordamos o sistema de chamadas que delineamos na Capítulo 4. No entanto, como aqui demonstramos, a implementação do sistema com base no mapeamento de objectos seguirá um rumo diferente que podemos até equiparar à implementação relacional. De facto, a fase inicial de descrição do modelo e posterior implementação das operações é mais próxima versão relacional que da versão sobre Cassandra. Esta implementação é também testada utilizando a plataforma de simulação num cenário similar ao apresentado para implementação nativa.

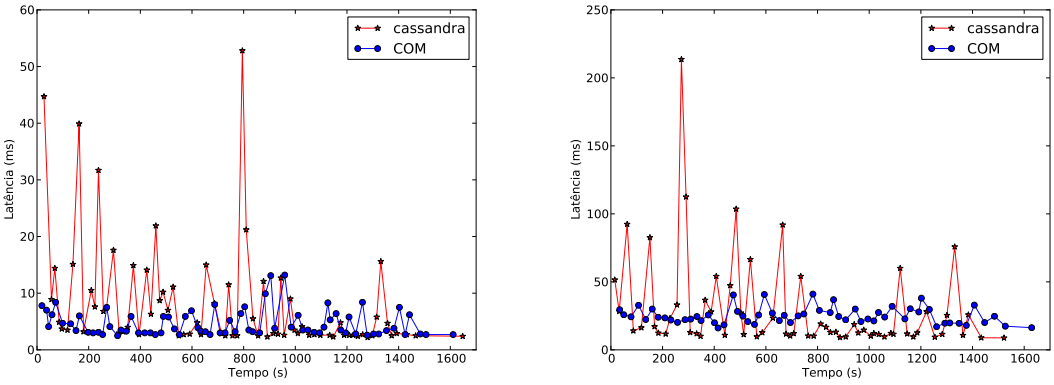
Implementação

Contrastando com a versão da plataforma de atendimento de chamadas implementada directamente sobre Cassandra, o processo de modelação teve aqui de ser invertido. Sendo o primeiros dos passos numa solução deste género a criação do mapa de entidades, de facto o processo de desenvolvimento inicia-se de forma similar à implementação relacional.

Afastado dos conceitos de famílias ou super famílias, o programador tem agora de se focar no modelo de objectos. Tal mudança leva a uma afastamento de uma estrutura baseada na indexação natural da informação como apresentada anteriormente para uma estrutura mais genérica baseada em entidades e relações. A indexação de diversos dos campos continuará sempre a existir, mas esta será agora mantida à parte da informação global. Este processo não é no entanto directamente controlado pelo utilizador, mas sim uma consequência do processo de modelação.

Assim para implementação da plataforma são geradas 5 classes principais que são a Conta, o Cliente, o Serviço, o Saldo, e o Plano tarifário. Este último tem no entanto de se dividir em 2 classes pois cada plano tarifário não é mais que um mapa de informação tarifária por serviço subscrito. Este modelo pode ser observado na Figura 5.4.

Analisando as fases de execução, terão depois de ser adicionados índices secundários para tornar possível a implementação de algumas das operações. Novamente percorrem-se as 4 fases que caracterizam o sistema e analisam-se as adaptações realizadas.



(a) (b)

Figura 5.2: Latências para as operações de (a) Informação de um produto (b) Compra de um produto no TPC-W.

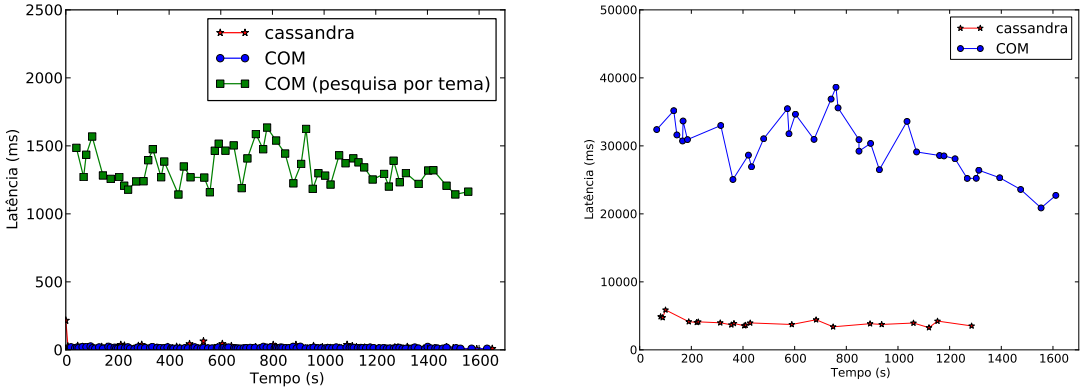
Selecção dos Serviços Tendo como parâmetro de entrada o número do cliente e o tipo de contacto recebido, nesta operação temos assim de pesquisar pelos serviços que a estes dados correspondem. Para tal é necessário a inserção de índices nestes campos permitindo assim o uso de operações em JDOQL para retorno das entidades. Adicionando as anotações na classe original, o processo de extracção é marcado por instruções como:

```
Extent e = interface_persistencia.getExtent(Servicos.class, true);
Query q = interface_persistencia
    .newQuery(e, "numero == " + (chamada.getNumeroOrigem())
    + " && " + "tipo_servico == \"" + chamada.getTipo() + "\"");
Object res = q.execute();
```

Note-se que, aquando do uso de operações de pesquisa baseadas em índices, o utilizador de Cassandra deverá ter o cuidado de colocar o critério de procura que retorne menos resultados sempre em primeiro lugar.

Escolha dos tipo de saldo a utilizar Para a pesquisa dos tipos de saldo, usamos também aqui o JDOQL com base em índices secundários. Esta operação acaba assim por se tornar em muito semelhante à implementação relacional. Na tabela que contém a informação para as várias tarifas procuramos então pelo nome do serviço e pela conta associada. Poderiam também ser incluídos na instrução de pesquisa predicados para o teste das limitações temporais associadas às tarifas. No entanto como esta decisão implicaria a adição de mais índices à base de dados, tal não foi concretizado.

```
Extent e = interface_persistencia
    .getExtent(InformacaoTarifaria.class, true);
Query q = interface_persistencia
```



(a) (b)

Figura 5.3: Latências para as operações de (a) Procura de produtos (b) Mais vendidos no TPC-W.

```
        .newQuery(e, "conta_id == " + (servico.getIdConta())
            + " && " + "nome_servico == \"" + servico.getNome() + "\"");

    Object res = q.execute();
```

Pesquisa do tipo de saldo Na fase seguinte, que passa pela descoberta de qual a conta onde se encontra o saldo a utilizar, as operações executadas baseiam-se no retorno de uma simples entidade e o percorrer das suas relações. Assim para saber-se se a conta base contém o tipo de saldo pretendido, lê-se o objecto e neste se verifica se a sua lista de saldos contém este mesmo tipo.

```
Conta conta = interface_persistencia
    .getObjectById(Conta.class, servico.getIdConta());
Map<String, Saldo> saldos = conta.getSaldos();

if (saldos.containsKey(tipo_saldo)) {
    ...
}
```

Se tal não se verificar, então obtém-se a entidade pai a partir da conta base e repete-se o teste. Não estando no grupo de campos que por omissão são carregados (opção por nós tomada), esta informação será apenas lida nesta fase. Este é provavelmente aquele passo onde melhor se vê a simplicidade desta alternativa.

```
Conta conta_pai = conta.getPai();
saldos = conta_pai.getSaldos();
```

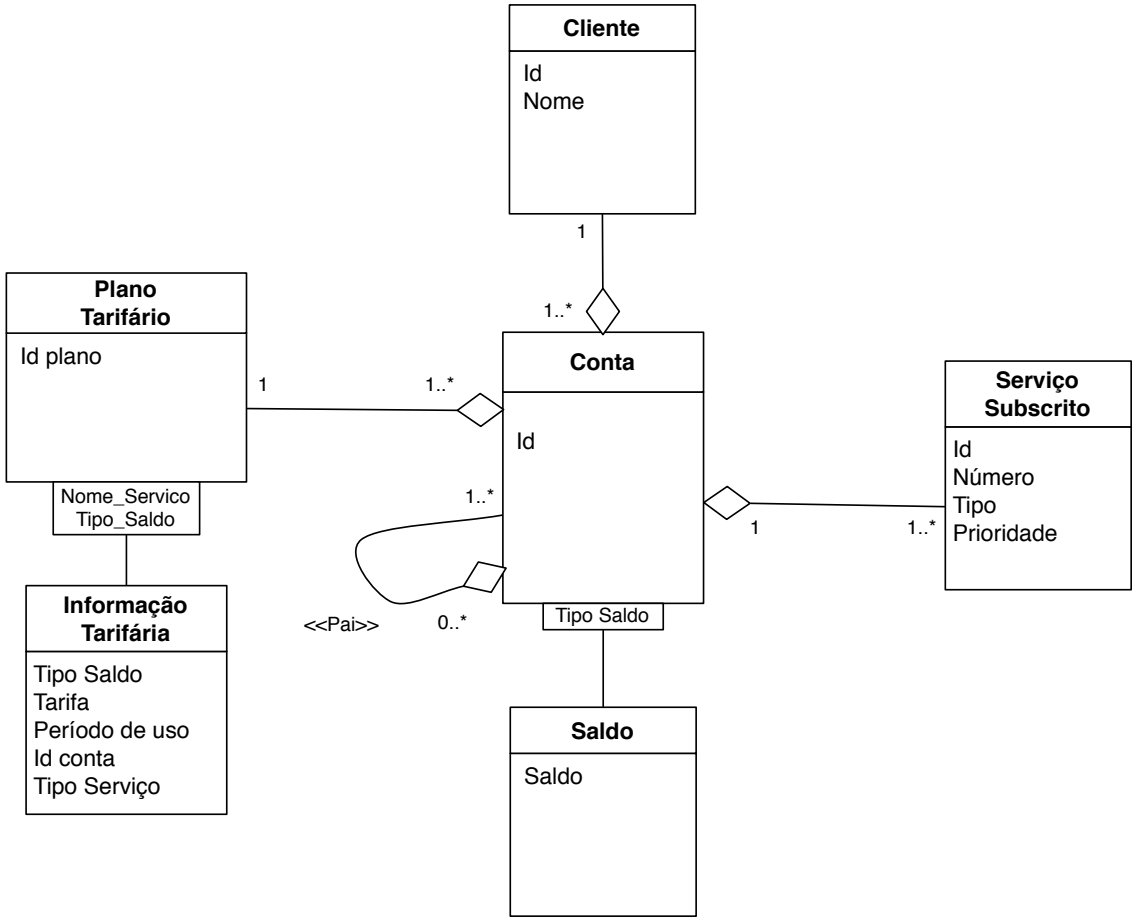


Figura 5.4: Modelo de classes implementado.

```
if (saldos.containsKey(tipo_saldo)) {
    ...
}
```

Consumo do saldo Neste último passo, o processo passa somente pela leitura da conta em causa onde se pesquisa a instância que contém o saldo e se proceda à sua actualização.

```
Conta conta = interface_persistencia
    .getObjectById(Conta.class, conta_id);
Saldo unidade_saldo = conta.getSaldos().get(tipo_saldo);
int saldo = unidade_saldo.getSaldo();
if ( saldo > custo) {
    saldo -= custo;
    unidade_saldo.setSaldo(saldo);
}
```

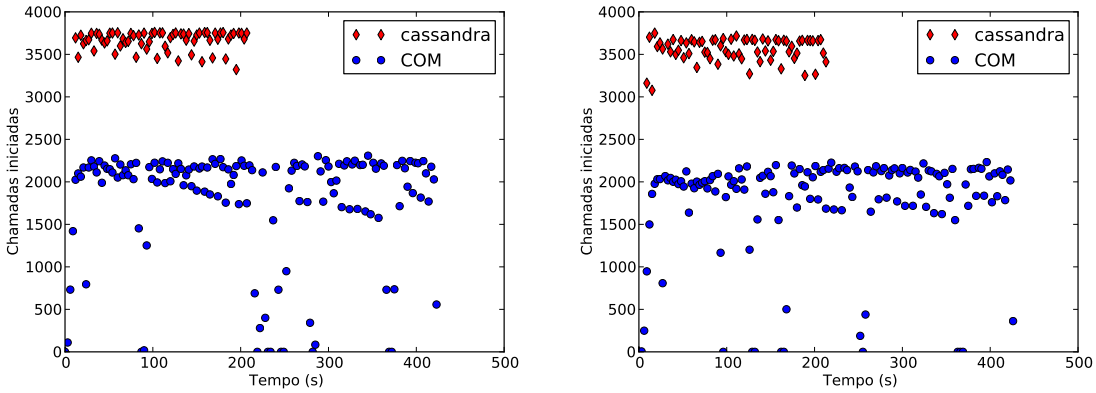
Embora não tenha sido implementada em Cassandra para teste da solução de mapeamento,

esta operação poderia também ser redireccionada para uma base de dados transaccional. Com os dados de pesquisa do saldo, podemos facilmente introduzir no código base algo como:

```
PersistenceManagerFactory interface_relacional = JDOHelper
    .getPersistenceManagerFactory("datanucleus_relacional.properties");

....

PersistenceManager interface_persistencia
    = interface_relacional.getPersistenceManager();
Transaction transaccao=pm.currentTransaction();
try
 {
    transaccao.begin();

    Saldo unidade_saldo = interface_persistencia
        .getObjectById(Saldo.class, conta_id+"_"+tipo_saldo);
    int saldo = unidade_saldo.getSaldo();
    if ( saldo > custo) {
        saldo -= custo;
        unidade_saldo.setSaldo(saldo);
    }

    transaccao.commit();
}
finally
 {
    if (transaccao.isActive())
    {
         transaccao.rollback();
    }
    interface_persistenciaclose();
}
```

Ambiente de trabalho

Para a o teste da solução neste caso de estudo usamos dois dos três *workloads* definidos: Dia e Golo, sendo executados sobre uma populações média, com a recolha de medidas de latência e débito tanto das chamadas em geral como das operações que as constituem. Para a execução de tais cenários, é utilizada uma máquina de 8 cores de 16 GB de RAM para a execução da plataforma de testes, sendo a base de dados colocada numa máquina com 24 cores, 128 GB de RAM e armazenamento baseado numa HP HSV300 Eva Storage com RAID1 e FC (FiberChannel) HBA 8Gbps. Em termos de rede as máquinas encontram-se ligadas por uma rede Gigabit. Quanto ao sistema operativo, trata de Linux, Ubuntu Server com kernel 2.6.24-28-server e versão do Cassandra é a 0.7.4. Esta versão do Cassandra significa o uso de uma nova versão do COM que delega na base de dados a gestão do índices.



(a) (b)

Figura 5.5: Chamadas estabelecidas num cenário de a) dia b) golo.

Resultados

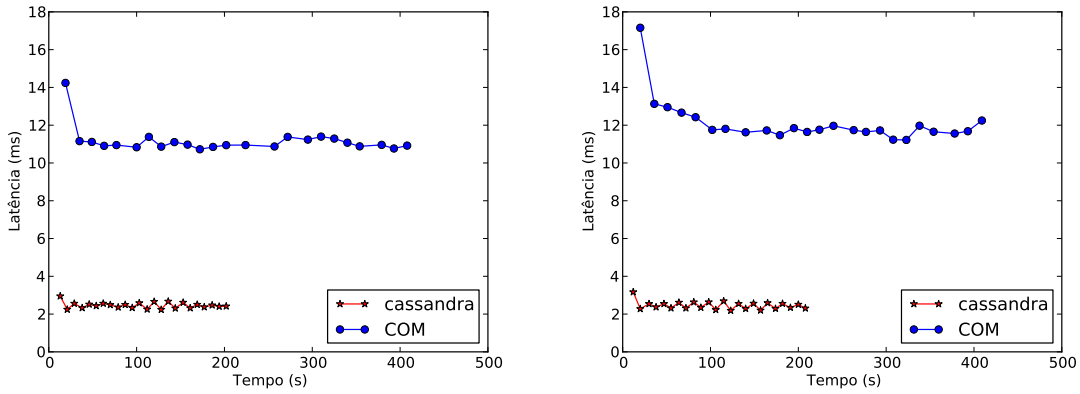
Analisando agora os resultados obtidos são visíveis os custos associados à simplicidade ao modelo genérico no Cassandra implementado. Em termos de chamadas realizadas, em ambos os cenários, o COM apresenta em média uma diminuição de 1500 por cada período de 30 segundos como visível na Figura 5.5. Da mesma maneira, observando os valores de latência visíveis na Figura 5.6 vemos que estes são como esperado superiores aos obtidos na implementação nativa. Ainda assim, com uma latência média de 12 milissegundos este é um valor perfeitamente aceitável.

Na verdade existem aqui várias causas secundárias para o decréscimo do número de chamadas por minuto. A primeira prende-se com um problema na versão actual do componente de mapeamento ainda não localizado que causa picos de processamento anormais. Com este problema resolvido espera-se obter resultados mais estáveis. A outra das causas relaciona-se com as limitações do índices secundários do Cassandra quando sujeitos a elementos de grande cardinalidade como por exemplo o número de telefone. Poderíamos aqui oferecer ao utilizador a opção de optar por índices secundários não nativos, mas tal opção levaria a que todo o mecanismo de pesquisa tivesse de ser implementado do lado do cliente aquando de operações com mais de um campo de pesquisa.

5.4 Sumário

Neste capítulo apresentamos a solução de mapeamento de objectos por nós implementada com o nome de COM. Neste contexto abordamos o seu processo de desenvolvimento, onde se mostram alguns dos pormenores de implementação. Na fase final mostramos como foram os já conhecidos casos de estudo migrados para esta solução e qual o seu desempenho.

Aqui conclui-se que soluções como o COM são uma grande valia no desenvolvimento de aplicações sobre Cassandra, permitindo uma redução do tamanho e complexidade do código. Esta simplicidade tem no entanto um custo na medida em que muitas operações têm aqui um pior desempenho. Assim no caso do TPC-W destacam-se as várias operações mais simples que tem



(a) (b)

Figura 5.6: Latência no estabelecimento de chamadas num cenário de a) dia b) golo.

resultados comparáveis com a implementação nativa. Já as operações com muitas leituras que tem um pior desempenho. Abordando os resultados do sistema de chamadas vemos aqui uma grande quebra no número de chamadas por minuto, mas por outro lado o tempo médio de estabelecimento de chamada é ainda assim aceitável.

Assim podemos concluir que dependendo dos requisitos de desempenho pelo utilizador impostos, esta solução pode ser uma opção viável quando se pretende uma interface de programação mais amigável e um modelo mais genérico que permita uma mais fácil implementação de futuras operações.

Capítulo 6

Conclusão

Perante novos desafios no armazenamento e manipulação de grandes quantidades de dados existe, no passado recente, uma crítica crescente às limitações impostas pelas tradicionais soluções relacionais. Surgiram então sob a designação NOSQL diversos motores de dados, propondo diferentes interfaces de acesso, modelos dinâmicos de dados ou uma melhor adaptação à actual natureza distribuída dos dados. Sistemas como o Cassandra ou Hbase são, com as suas arquitecturas distribuídas, soluções reconhecidas para os problemas de escalabilidade e elasticidade que afectam muitas empresas. Baseadas em hardware de uso comum, estas bases de dados de larga escala apresentam também mecanismos automáticos de replicação, fail-over e balanceamento de carga.

Contudo, quando comparadas com as consolidadas bases de dados relacionais, este género de soluções oferece uma série de desvantagens. Com interfaces limitadas que implicam clientes de comunicação mais complexos e sujeitos a erros, estruturas de dados intrincadas e modelos de alternativos de coerência de difícil aprendizagem estas soluções requerem uma longa curva de aprendizagem. Todos estes factores tornam assim a adaptação a estes novos modelos mais difícil principalmente para utilizadores que provêm de um ambiente relacional.

Neste contexto, com recurso a uma plataforma de testes como o TPC-W, analisamos as diferenças no paradigma de desenvolvimento, as estruturas de dados alteradas, as interfaces de acesso e muitos outros elementos de mudança que destas migrações são característicos. Utilizando o Cassandra como objecto de estudo, obtêm-se as mais importantes características desta base de dados que afectam o processo de desenvolvimento da aplicação. Abordando as questões de modelação observamos o modo como o uso de super colunas é uma solução para a materialização de informação a pesquisar, substituindo assim o uso de *joins*. Neste mesmo tópico, apresenta-se também como a ordenação de colunas e linhas acabam por ser soluções para algumas das operações de pesquisa como os mais vendidos ou a determinação dos produtos mais recentes. Em termos de interface são analisados diferentes tipos de clientes, comprovando que apesar de não se poder evitar parte da migração do código para o lado do cliente, o programador possui contudo diversas opções com diferentes níveis de complexidade. Analisam-se no final as questões de coerência e associados mecanismos de resolução de conflitos comprovando que é possível migrar operações como a gestão de um stock, que eram anteriormente baseadas em mecanismos transaccionais.

Com a implementação final conclui-se que de facto esta aplicação pode transitar para tal paradigma necessitando no entanto de inúmeras adaptações ao seu modelo e o relaxamento de algumas das garantias que originalmente eram dadas. No entanto comprova-se também que algumas das operações, como os mais vendidos ou a operação de administração sobre um produto, acabam

por ser tornar inutilizáveis se delas esperarmos uma resposta em tempo real, devido às interfaces simplificadas impostas por bases de dados.

Com um caso real de migração de uma empresa que pretende medir qual o impacto do uso de uma solução como Cassandra no seu sistema, realçamos o modo como esta transição leva a uma afastamento das estruturas genéricas de dados levando a uma solução altamente optimizada em modelo e funcionamento. Ainda que com este custo da perda de adaptabilidade do sistema a futuras alterações comprova-se, com os testes realizados, que ambos os paradigmas podem ter um desempenho similar. Em concreto, o desempenho nos cenários pequenos é equiparável em termos de chamadas realizadas por minuto, sendo que o mesmo não acontece em cenários de maior dimensão. Nos cenários médios, o Cassandra acaba por apresentar uma diferença de desempenho de 18% a 36% a mais que a versão relacional, devendo-se este facto à ausência de garantias transaccionais nesta base de dados, tal como ao modelo optimizado sobre ela implementado.

Estas conclusões e resultados contribuíram para o projecto *Cloud Computing* realizado em parceria pela U.Minho e pela PT Inovação, tendo como objectivo a avaliação de quais as vantagens e perigos na migração do seu sistema chamadas para uma base de dados não relacional.

As conclusões obtidas são então generalizadas através do desenvolvimento de um componente de mapeamento de objectos em Cassandra. Designado por COM, este componente permite a persistência de objectos e das suas relações de uma forma simples e rápida. Na sua implementação definem-se as estratégias de persistência para cada um dos tipos de relações existentes, baseadas sempre em famílias de colunas normais e no seu armazenamento na entidade que as possui. Neste componente são também incluídos mecanismos de pesquisa com base em JDOQL que baseando-se em índices secundários permitem um mais fácil desenvolvimento de operações de leitura sobre Cassandra.

Com as adaptações dos casos de estudo já apresentados demonstram-se as vantagens desta abordagem na fase de desenvolvimento, mas também os seus efeitos negativos em termos de desempenho. De facto, existe com o uso deste componente um corte no número de chamadas por minuto de aproximadamente 50%. Em termos de latência o valor médio é também ele várias vezes maior com o uso do COM, sendo que este é ainda assim aceitável pois situa-se na casa dos 10 ms. Ainda que apresente problemas de desempenho, o modelo mais genérico e adaptável, tal como o processo de programação simplificado tornam esta uma solução útil para a migração de soluções já existentes.

O COM foi o primeiro OM para Cassandra a suportar uma das interfaces padrão de persistência. Por esta razão ele foi aquando do sua publicação aceite como um cliente para a versão 0.6.x do Cassandra pela sua comunidade[1] e na plataforma DataNucleus[2]. As contribuições nele implícitas foram entretanto utilizadas já por outros projectos de OM de âmbito mais vasto[3]. Este e outros projectos que surgiram no decorrer deste trabalho encontram-se no Anexo B onde também se refere algumas das evoluções do Cassandra neste mesmo período.

6.1 Publicações/Apresentações

Partes deste trabalho foram publicados e apresentados da seguinte forma:

[1] http://wiki.apache.org/cassandra/ClientOptions06

[2] http://www.datanucleus.org/products/accessplatform/datastores_thirdparty.html

[3] https://github.com/tnine/Datanucleus-Cassandra-Plugin

- P. Gomes, J. Pereira and R. Oliveira. An Object Mapping for the Cassandra Distributed Database. In INForum. 2011.
- F. Cruz, P. Gomes, J. Pereira and R. Oliveira. Migrating a Telecom Application to a NoSQL Database. In International Workshop on Clouds for Enterprises (C4E). 2011.
- Apresentação "Cassandra - Bridging the gap between SQL and NOSQL"nas jornadas de informática do Instituto Superior de Engenharia de Lisboa em 2011 (ISEL Tech 2011) em colaboração com Francisco Cruz.
- Apresentações "An introduction to Cassandra"e "SQL vs NOSQL - friends or foes"nas Geek Nights, evento realizado todos os meses em Braga.

6.2 Questões em aberto

Apresenta-se neste trabalho um caso de estudo real de migração de uma aplicação para uma base de dados distribuída como o Cassandra. No entanto, como referido nas conclusões do capítulo 4 não pudemos por limitações de hardware executar os testes num ambiente distribuído. Para uma maior rigor, e para de facto avaliar as vantagens de escalabilidade desta base de dados, este testes devem no futuro ser executados em mais de um nó e com o maior dos cenários que não foi testado.

Abordando o caso do componente de mapeamento de objectos, para além da sua actualização para a nova versão do Cassandra, seria interessante no futuro abordar a questão transaccional do problema. Com o uso de uma mecanismo de controlo de acesso externo como o Zookeeper seria assim mais útil. Este permitiria a adaptação em situações onde este género de garantias, embora não frequentes ao ponto de ser necessário o uso de uma outra solução, não podem ser relaxadas para implementação directa.

Apêndice A

Plataforma de testes TPC-W

Com o desenvolvimento e execução desta plataforma pretende-se fornecer uma visão de como a performance das operações é afectada pelas decisões de implementação, tal como avaliar de algumas das características das soluções estudadas. De facto o valor retirado da plataforma encontra-se não só nos resultados desta obtidos mas também da experiência retirada da sua migração uma vez que ela foi originalmente desenhada para o modelo relacional.

Com o intuito de testar e comparar as implementações do TPC-W, descrevemos aqui as bases desta mesma plataforma de *benchmark*. Com a necessidade de suporte para vários perfis de *benchmark* para a observação das diferentes soluções sob as várias implementações subjacentes, esta terá de ser modular. Na seu conjunto esta é assim constituída por quatro elementos que podemos identificar na Figura A.1 e que são agora descritos.

Gerador da carga de trabalho (*Workload*) Elemento responsável pela geração da sequência de operações a executar tal como dos parâmetros nelas inclusas. Este foi desenhado para fornecer vários clientes de geração que executarão em paralelo na plataforma, fornecendo instruções aos elementos de execução.

Pretende-se que estes componentes sejam o mais genéricos possível de modo a serem transversais ao motor de dados usado. Como tal nenhuma lógica de execução deverá ser incluída nestes. Na sua implementação cabe ao programador codificar qual o funcionamento da plataforma em três fases distintas: inicialização, execução e consolidação dos resultados. Este última fase é útil para avaliar possíveis dados resultantes da execução.

Povoador da base de dados Usado como o nome indica para a povoação da base de dados, este elemento possibilita definir o modo como a base de dados é povoada ou limpa. Também construído de de forma o mais genérica possível para a execução sobre diferentes motores de dados, tal não é por vezes possível devido à heterogeneidade das APIs e esquemas de dados subjacentes.

Executor Esta é a interface para execução na base de dados. Nela se implementam algumas operações base de leitura e escrita fornecendo uma interface com a base de dados. Este componente terá também de ser codificado de forma a interpretar todas as operações que provêem da carga de trabalho executando-as na base de dados subjacente. Note-se que, sem qualquer tipo de lógica associada às operações, pretende-se que estas sejam assim implementadas com a noção de quais as vantagens e limitações das soluções usadas.

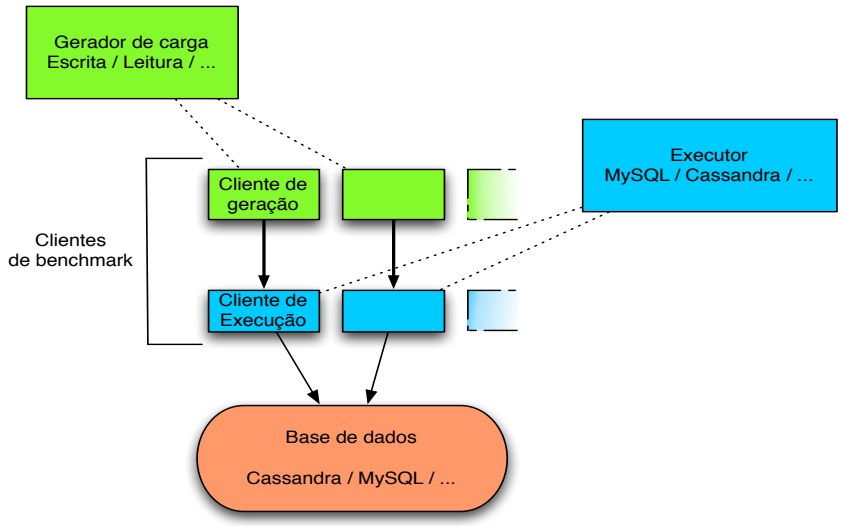


Figura A.1: Funcionamento da plataforma

Controlador No contexto da plataforma, existe também numa camada superior um controlador destes diferentes componentes. A sua tarefa é instanciar os diferentes elementos segundo a carga e base de dados que se pretende usar. Esta terá depois de gerar o número pretendido de clientes e executá-los, sendo também responsável pela coordenação dos vários nós de benchmark. De facto, foi adicionada à plataforma a capacidade de ser executada em diferentes máquinas sendo a coordenação entre estas feita através de um mestre. Esta camada de complexidade foi adicionada ao sistema para permitir um aumento da concorrência real de pedidos sobre as bases de dados e para o teste da influencia dos mecanismo de resolução de conflitos ao Cassandra inerentes.

Com esta arquitectura base, tal plataforma permite a execução de diferentes implementações e posterior comparação dos resultados obtidos. Sob esta base analisamos agora o processo de transição entre os modelos que resultou da construção dos módulos de execução quer na versão relacional como não relacional.

Apêndice B

Novas tecnologias e clientes

No contexto deste trabalho abordamos múltiplas ferramentas e tecnologias que dada a sua natureza recente estão sujeitas a constantes mudanças. Por esta razão, apresenta-se algumas tecnologias e soluções que apareceram no decorrer ou já numa fase posterior à conclusão deste trabalho. Aborda-se deste modo a evolução do Cassandra tal como outros clientes de OM que surgiram depois deste trabalho.

B.1 Cassandra

Aquando do início deste trabalho, para o desenvolvimento da plataforma TPC-W foram na altura avaliadas as diferentes opções baseadas em Java, resumindo-se as viáveis à interface nativa baseada em Thrift e o na altura ainda imaturo cliente de alto nível Hector. Desde então o ecossistema que a rodeia a base de dados cresceu e novas opções sugiram como o cliente Pelops[1] ou a nova linguagem de interrogação, CQL, que foi introduzida com a versão 0.8 da base de dados e presume-se ser no futuro a interface de acesso por omissão. Devido à imaturidade do cliente Hector aquando do inicio do projecto e posteriormente devido à experiência e código acumulados, a interface de acesso usada no decorrer deste foi sempre o Thrift.

Do mesmo modo, são nas soluções implementadas usadas famílias de super colunas para muitas vezes persistir a informação indexados pelos campos pelos quais são realizadas pesquisas. Note-se no entanto que o uso de super colunas é hoje cada vez mais desaconselhado devido aos seus limitados casos de uso e á frequente penalização que provoca no desempenho. É visível nos resultados que foram obtidos com a plataforma TPC-W que de facto estas estruturas causam um agravamento da latência, não só nas operações que sobre elas incidem, mas em todas as outras que em paralelo são executadas. Um alternativa futura a este género de colunas é o uso de colunas compostas que foram apresentadas na versão 0.8[2].

[1] https://github.com/s7/scale7-pelops
[2] https://github.com/edanuff/CassandraCompositeType

B.2 Interfaces baseadas em objectos

Posteriores à publicação do nosso componente, outra soluções emergiram no seio da indústria. Apresentamos agora algumas dessas soluções.

Kundera Kundera[3] é um OM com uma implementação compatível com JPA. Divulgado publicamente em Agosto de 2010, este tira partido de cliente Java Pelops para interagir com o Cassandra. Ela possui suporte para mecanismos de cache, relações complexas entre entidades e possui também uma linguagem de consulta própria. Para a execução do mapeamento, o cliente faz uso de uma extensão às anotações definidas pelo JPA 2.0. Esta suporta para além de Cassandra, as bases de dados HBase e MongoDB.

Neste cliente, o utilizador terá de definir para cada entidade qual o tipo de família a usar para além de quais os campos a persistir. Existe assim por parte da plataforma um suporte a famílias de super colunas. Esta ferramenta inclui igualmente suporte para índices que sendo implementados em Solandra[4] permitem assim efectuar pesquisas textuais. Esta plataforma possui também uma linguagem de consulta onde utilizador pode utilizar operadores como *WHERE* ou *LIKE*.

Datanucleus-Cassandra-Plugin Emergindo poucas semanas a seguir à nossa solução, o projecto Datanucleus-Cassandra-Plugin [5] tem ainda como base algum do código por nós implementado. Sendo actualmente usado em produção, esta solução de mapeamento é também ela baseada na plataforma Datanucleus. Com base no cliente Pelops, esta plataforma segue muitas das estratégias definidas para o COM, não suportando assim a persistência em famílias de super colunas. Ela permite também o uso de JDOQL sendo que possui implementado um suporte para diversos operadores.

Este cliente foi recentemente abandonado estando uma nova versão baseada em JPA a ser desenvolvida[6].

Outras opções Numa normal evolução do mercado outras operações foram surgindo com diferentes níveis de maturidade. Entre elas encontramos projectos como o Apache Gora[7] que se apresenta como um interface unificada para muitos dos projectos NOSQL. Este não suporta interfaces como JDO ou JPA pois estas acabam por limitar a sua capacidade de modelação de problemas sob as bases de dados subjacentes. Esta solução possui também suporte para Hadoop.

Outra das soluções que se encontra entre os cliente para Cassandra é o Demoiselle Cassandra[8]. Também ele sem suporte a interfaces base de persistência como o JPA ou JDO, este é um componente da plataforma brasileira Demoiselle. Esta é uma solução baseada na padrão de desenvolvimento DAO que não suporta índices ou operações de maior complexidade.

A um nível mais simples existe actualmente uma tentativa de integração de Cassandra com Hibernate[9] estando este ainda em processo de desenvolvimento.

[3] https://github.com/impetus-opensource/Kundera
[4] https://github.com/tjake/Solandra
[5] https://github.com/tnine/Datanucleus-Cassandra-Plugin
[6] https://github.com/riptano/hector-jpa
[7] http://incubator.apache.org/gora/
[8] http://demoiselle.sourceforge.net/component/demoiselle-cassandra/1.0.0/index.html
[9] https://github.com/emmanuelbernard/hibernate-core-ogm

Bibliografia

[1] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: a relational approach to database management. pages 16–36, 1988.

[2] D. Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[3] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM.

[4] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[5] CCA08. *LINQ-to-Datacenter*, 2008.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[7] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, 1983.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. page 12. ACM, 2010.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[10] S. Finkelstein, D. Jacobs, and R. Brendle. Principles for inconsistency. In *CIDR*, 2009.

[11] J. Furman, J. Karlsson, J. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A scalable data system for user facing applications. *SIGMOD 2008*, 2008.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[13] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[14] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[15] A. Hemrajani. *Agile Java Development with Spring, Hibernate and Eclipse (Developer's Library)*. Sams, Indianapolis, IN, USA, 2006.

[16] A. Lakshman and P. Malik. Cassandra: a structured storage system on a p2p network. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 47–47, New York, NY, USA, 2009. ACM.

[17] J. Linwood and D. Minter. *Beginning Hibernate, Second Edition*. Apress, Berkely, CA, USA, 2010.

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[19] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[20] B. Sam-Bodden. *Beginning POJOs: From Novice to Professional (Beginning from Novice to Professional)*. Apress, Berkely, CA, USA, 2006.

[21] M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[22] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of ingres. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.

[23] T. P. P. C. (TPC). Tpc benchmark w(web commerce) specification version 1.8, 2002.

[24] I. T. Varley. No relation: The mixed blessings of non-relational databases. Master's thesis, The University of Texas at Austin, 2009.

[25] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.

[26] Z. Wei, G. Pierre, and C.-H. Chi. CloudTPS: Scalable transactions for Web applications in the cloud. Technical Report IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, Feb. 2010. http://www.globule.org/publi/CSTWAC_ircs53.html.

[27] C. Zhang and H. D. Sterck. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase. In *GCA*, 2010.