FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Framework de Cache Pesquisável e de Alta Performance

João Filipe Meneses Henriques

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Rui Filipe Lima Maranhão de Abreu

6 de Março de 2015

Framework de Cache Pesquisável e de Alta Performance

João Filipe Meneses Henriques

Mestrado Integrado em Engenharia Informática e Computação

Presidente: Raul Fernando de Almeida Moreira Vidal
Arguente: João Saraiva
Vogal: Rui Filipe Lima Maranhão de Abreu
6 de Março de 2015

Resumo

Nos dias de hoje, todas as organizações necessitam de partilhar e guardar informação para consultas futuras, sendo que quanto maior for a organização, maior será o fluxo da circulação de dados, e maior serão o número de pessoas a aceder concorrentemente a toda a informação. É através das bases de dados centrais que é possível manter esta informação coerente e acessível a todos sempre que necessária. Quando uma organização cresce ao ponto de saturar a largura de banda ou a capacidade de processamento da base de dados, é necessário arranjar formas inovadoras para se conseguir expandir e manter a qualidade dos serviços. Normalmente é nos acessos às bases de dados onde se encontra o ponto de estrangulamento do sistema, tornado-se numa área onde existe uma forte necessidade de tornar o mais eficiente possível. Uma abordagem que nos últimos tempos tem ganho algum *momentum* e que na maior parte das situações é suficiente, é através da utilização de sistemas de cache *in-memory* para guardar temporariamente os dados mais frequentemente utilizados. No entanto no caso de existir a necessidade de colocar em memória um *data set* (ou parte dele), não existe forma de serem efectuadas pesquisas directamente nestas tecnologias por terem normalmente uma arquitectura do tipo *key-value*, semelhante a um dicionário. As alternativas actuais passam pela utilização bases de dados híbridas, que colocam o *data set* em memória mas que precisam de persistir os dados num meio de armazenamento mais lento.

O trabalho efectuando nesta tese teve o objectivo de pesquisar e determinar se seria possível implementar uma framework de cache *in-memory* pesquisável e de alta performance, continuando a manter as características de um sistema de cache *in-memory*.

Palavras-chave: bases de dados, cache, optimização, redução de tráfego

Abstract

Nowadays, all organizations need to share and store information for future reference, and the larger the organization is the greater will be the flow of data and the number of persons accessing it concurrently. It is through a central databases that is possible to keep all this information consistent and accessible whenever needed. When a organization grows to the point of saturating the bandwidth or processing power of their database, it is necessary to think of innovative ways to keep their quality of service. Usually most system bottlenecks are located in the database access, being this one of the most important places to keep as efficient as possible. One approach that recently is gaining some momentum and that in most situations its enough, is by using in-memory caching systems to temporarily store the most frequently used data. However if there is the need to store a data set (or part of it), there is no way to make a search in this data because this systems typically have a key value architecture, similar to a dictionary. The current alternatives are the use of hybrid databases, which place in memory the current data set, but still need a slower storage medium.

The work done in this thesis had the objective to research and determine whether it would be possible to implement a high efficiency and searchable in-memory caching framework while maintaining the characteristics of an in-memory caching system.

Keywords: database, cache, optimization, bandwidth reduction

"Information is not knowledge."

Albert Einstein

Conteúdo

Lista de Figuras

LISTA DE FIGURAS

Lista de Tabelas

Abreviaturas e Símbolos

CPU	Unidade de Processamento Central
DDD	Domínio do Discurso
BD	Base de Dados
SSD	Solid State Drive
SGBD	Sistema de Gestão de Bases de Dados
Redis	REmote DIctionary Server
LRU	Least Recently Used
SQL	Structured Query Language (Linguagem de Consulta Estruturada)
NoSQL	Not SQL / Not Only SQL
JSON	JavaScript Object Notation
BSON	Binary JSON
XML	Extensible Markup Language
BLOB	Binary Large OBject
KB	Kilobyte
WCF	Windows Communication Foundation
SOAP	Simple Object Access Protocol
N1QL	Non-first Normal Form Query Language (pronuncioado "Nickel")
BLOB	Binary Large OBject

Capítulo 1

Introdução

1.1 Contexto/Enquadramento

Este trabalho insere-se na área de Sistemas de Informação, tendo sido desenvolvido com o apoio da Glintt Healthcare (Glintt HS), sediada no Porto.

A Glintt HS é uma empresa com mais de 20 anos de experiência no desenvolvimento de soluções informáticas para a área de saúde e está presente em mais 200 hospitais e clínicas espalhados por Portugal, também com clientes no Brasil e na Polónia.

1.2 Motivação e Objetivos

É pressuposto que quando necessitamos de consultar alguma informação que se encontra armazenada numa base de dados, que esta nos seja retornada o mais rapidamente possível para que a possamos processar da forma pretendida.

Com a evolução da tecnologia, e numa sociedade cada vez mais ligada em rede, existem muitos mais computadores a aceder às mesmas bases de dados concorrentemente. O pedido de um utilizador pode causar dezenas ou mesmo centenas de pesquisas a uma base de dados, causando um grande impacto na utilização da BD. Facilmente se deduz que o aceso às BDs pode ser o ponto de estrangulamento da uma vasta quantidade de sistemas. [LGZ03]

Qualquer sistema tem como ponto chave a performance, e muitas vezes, os atrasos não são tolerados, como por exemplo em situações que podem colocar em risco vidas humanas, sendo o caso da área da saúde.

É possível escalar verticalmente uma base de dados, acrescentando mais recursos à mesma, tal como actualizar o CPU, acrescentar memória ou colocar discos mais rápidos mas infelizmente existe sempre um limite para este escalonamento, pelo que é necessário pensar no escalonamento horizontal, seja através da delegação de tarefas para outras máquinas, ou simplesmente abordando os problemas de uma forma diferente na altura de desenvolvimento dos sistemas.

É dentro deste contexto que a *Glintt HS* pretendia descobrir se seria possível tornar mais eficientes os acessos a bases de dados que concentram grandes volumes de informação, principalmente quando existe a necessidade de fazer acessos constantes.

A forma mais usual de optimizar sistemas com este tipo de características é através da utilização de caches temporárias *in-memory*. No entanto com a utilização destas caches surgem algumas questões:

- Quais os dados a colocar em cache?
- Quais os tipos correctos de cache a utilizar dependendo da situação?
- Ou até que ponto é possível escalar um sistema de cache?

Uma abordagem que pode pode ser tomada para optimizar uma aplicação através da utilização de cache é através da colocação em memória dos dados mais utilizados enquanto houver memória disponível, e despejando da memória os menos frequentemente utilizados quando esta se encontrar cheia (política *LRU*).

Uma outra abordagem que nalgumas situações pode ser útil, e desprezando por completo a quantidade de memória disponível, passa por colocar em memória o *data set* inteiro pretendido, ou uma parte do mesmo. Esta abordagem permite que qualquer elemento deste *data set* esteja sempre acessível.

Com esta tese pretendeu-se explorar esta última abordagem.

Como referido, um dos maiores problemas aqui presentes trata-se da quantidade de memória disponível, mas que foi levemente desprezado durante o desenvolvimento, sendo um problema delegado para o *developer* no momento da implementação, que deve fazer uma gestão cuidada do *data set* que pretende tratar.

Um outro problema trata-se de não existir uma forma eficaz de efectuar pesquisas sobre os dados que se encontram nos sistemas de cache *in-memory*, muito em parte pela arquitectura que estes utilizam. O sistema de cache *in-memory* tradicional normalmente é do tipo *key-value*, onde é possível associar um valor a uma chave, tal e qual como num array associativo, ou num dicionário. Existem no entanto sistemas híbridos *in-memory* que permitem efectuar pesquisas, embora não de forma totalmente eficiente, mas que continuam a ter sempre alguma persistência em disco que é completamente desnecessária no caso de uma cache.

Os objectivos desta dissertação passaram assim por:

- Produzir uma framework de cache *in-memory* que permita efectuar pesquisas eficientes e independentes do *data set*.
- Modularidade da interface de comunicação com a framework, que permita a qualquer momento desenvolver e/ou trocar o sistema de cache em utilização de forma transparente para o sistema que a utilize.

A metodologia seguida passou primeiro por adquirir o conhecimento sobre as diferentes tecnologias candidatas a serem implementas como um serviço de cache, conhecer as suas vantagens e limitações. Através de uma série de testes comparar as suas performances e tentar desenvolver uma solução que permitisse a sua utilização como um serviço de cache *in-memory*, modular, pesquisável e de alta performance. O objectivo final foi desenvolver um conector para o cliente e um serviço de cache em rede, que pudesse ser partilhado por diferentes clientes, utilizando a tecnologia candidata que mais se adequou às necessidades, com base na sua performance e suas características. Após o desenvolvimento ter terminado foi novamente comparado a performance das várias tecnologias.

1.3 Estrutura da Dissertação

Para além da introdução, esta dissertação contém mais quatro capítulos e dois anexos.

No capítulo 2, é descrito o estado da arte e são apresentados trabalhos relacionados.

No capítulo 3, é apresentada a descrição do trabalho realizado.

No capítulo 4, é feita uma análise da eficiência e medição dos tempos da framework produzida.

No capítulo 5, é apresentada uma conclusão do trabalho realizado.

No anexo A, é apresentada o protocolo utilizado pela framework para a comunicação entre o conector e o serviço de cache.

No anexo B, é apresentado um exemplo prático de utilização da framework.

Capítulo 2

Revisão Bibliográfica

2.1 Bases de dados

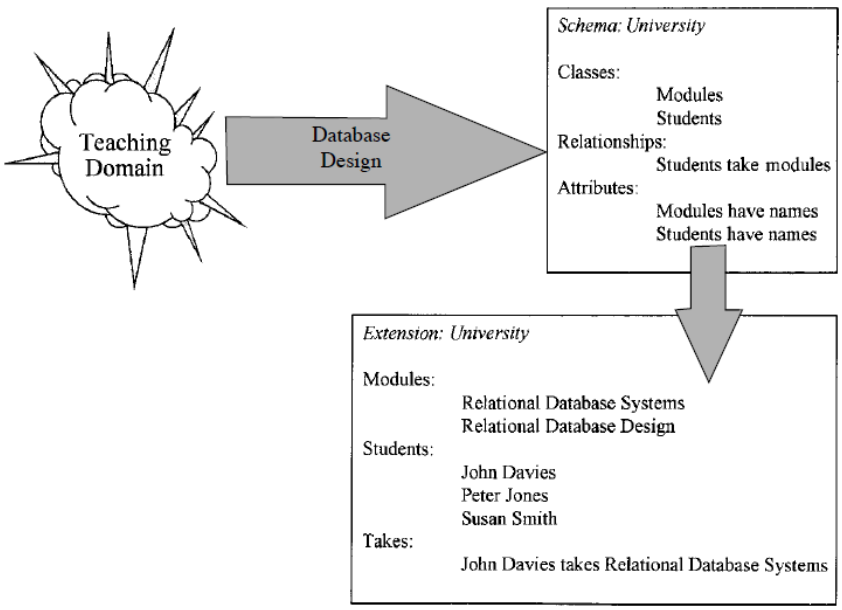

Figura 2.1: Parte Intensional e Extensional de uma base de dados

Uma base de dados pode ser considerada como o modelo de uma realidade de uma organização, sendo convencionalmente tratada como o *domínio do discurso* (DDD). O DDD pode ser representado por classes com relações entre si, sendo estas classes definidas por atributos e propriedades. [BD04, Capítulo 1.2]

Uma base de dados consiste em duas partes, uma parte intencional e outra extensional. A parte intencional trata de estruturar a representação do DDD, normalmente referida como o esquema,

sendo que a parte extensional trata do conjunto total dos dados presentes na BD. [BD04, Capítulo 1.2.3] Esta representação pode ser consultada na Figura 2.1.

2.1.1 Persistência

Para que estes dados possam ser consultados no futuro, eles precisam de persistir em algum meio.

No meio informática, as BD podem tomar grandes dimensões e são armazenadas nos discos rígidos de uma máquina/servidor. As escritas em disco normalmente são dispendiosas, uma vez que os tempos de acesso e de escrita costumam ser relativamente lentos. Os discos SSD podem ser uma alternativa para acelerar este processo já que estes que têm tempos de escrita e de acesso muito mais rápidos em relação aos discos mecânicos. Em bases de dados de grandes volumes, os discos SSD podem não ser, por vezes, viáveis para algumas organizações dados os custos demasiado altos, e capacidades de espaço mais reduzidas.

2.2 Sistemas de Gestão de Bases de Dados Relacionais

Um Sistema de Gestão de Bases de Dados (SGBD, ou DBMS em Inglês) é um software que permite através da utilização de um método standard, a recolha, catalogação e consulta de dados presentes na BD. O standard comum mais utilizado nas bases de dados relacionais é o SQL, tratando-se de uma linguagem de programação estruturada. Esta informação é armazenada em diferentes tabelas, e idealmente normalizada. No momento da consulta, esta informação é unida (*joined*) de forma a reconstruir os objectos que a representam.

Cada SGBD pode acrescentar extensões a esta linguagem, expondo características que as distingam de todos os outros.

Alguns exemplos dos SGBD mais utilizados são: *MySQL/MariaDB*, *PostegreSQL*, *SQLite*, *Microsoft SQL Server*, *Oracle*, entre muitos outros.

2.2.1 Propriedades ACID

Estas são quatro propriedades fundamentais que os SGBD relacionais devem implementar:

- **Atomicidade**: As bases de dados devem oferecer um mecanismo de transações que garanta que um conjunto de operações sejam todas executadas em caso de sucesso, ou nenhuma executada no caso de algum erro ter ocurrido, ou seja, deve tratar as operações como um trabalho indivisível que nunca pode ser terminado parcialmente.

- **Consistência**: As operações efectuadas numa base de dados têm de manter a consistência dos dados nela presentes, ou seja, respeitar regras de chaves primárias, ou de valores malformados, entre outros.

- **Isolamento**: As operações devem ser isoladas entre os vários utilizadores que estejam a aceder concorrentemente a uma base de dados. Por exemplo, se um utilizador estiver a escrever dados que outro pretenda ler, estas operações devem ser executadas sequencialmente, ou seja, o utilizador que pretende ler os dados só o deve poder fazer após o utilizador que esteja a escrever tiver concluído a sua operação, ou vice-versa, caso a operação de leitura tenha começado primeiro que a de escrita.

- **Durabilidade**: Os dados devem resistir a falhas como de energia, sendo sempre recuperáveis numa situação deste género. No caso de um erro de escrita, a operação deve falhar por completo, informando ao utilizador da falha.

Estas quatro propriedades oferecem uma garantia a qualquer organização que os seus dados estarão *sempre* seguros e insusceptíveis de erro. O único problema que pode ocorrer é a de corrupção dos dados por falhas de hardware, mas que podem ser facilmente contrariadas através da criação de cópias de segurança frequentes, ou de replicação *master/slave* (ver capítulo 2.4).

2.3 Bases de Dados não relacionais (NoSQL)

As bases de dados não relacionais, conhecidas como *NoSQL*, têm vindo a ganhar alguma popularidade nos últimos tempos devido à sua facilidade de integração, utilização e manutenção. Estas bases de dados quebram o legado das propriedades ACID e não costumam utilizar a linguagem SQL como linguagem primária. Algumas destas BDs também oferecem o SQL como uma alternativa mas normalmente de forma mais limitada que o SQL dos SGBD relacionais — por exemplo, operações *join* podem não ser possíveis.

Existem várias arquitecturas que podemos utilizar para classificar estes tipo de BD, sendo referidas 3 delas de seguida:

2.3.1 Baseada em Documentos ou Objectos

Trata-se a uma das principais arquitecturas dos sistemas BD NoSQL. Um documento normalmente refere-se a um conjunto estruturado de informação, ou objectos, que podem ser armazenados directamente na BD.

Alguns dos principais tipos de BD NoSQL baseados em documentos são: *Couchbase* (2.8.4), *MongoDB* [kn:f], *Apache CouchDB*, *Elastic Search*, *eXist*, *Sedna*, entre outros.

```
1 <Article>
2   <Author>
3       <FirstName>Bob</FirstName>
4       <Surname>Smith</Surname>
5   </Author>
6   <Abstract>This paper concerns....</Abstract>
7   <Section n="1">
```

```
8       <Title>Introduction</Title>
9       <Param..></Param..>
10    </Section>
11  </Article>
```

Listing 2.1: Exemplo de um documento NoSQL em XML

Algumas formas de representação dos objectos podem ser através de:

- **XML**: Exemplos são a *eXist* e *Sedna*.
- **JSON**: Como o *Couchbase* e *Apache CouchDB*.
- **BSON**: JSON em modo binário, utilizando por exemplo pelo *MongoDB*.

É fácil perceber a razão que tem levado a um aumento de popularidade das bases de dados NoSQL, dado que uma vasta quantidade delas utilizam o JSON como forma de armazenamento, no que toca a desenvolvimento web, armazenar estes dados directamente e consultar uma base de dados deste género revela um esforço quase inexistente para o *developer*.

2.3.2 Chave-valor

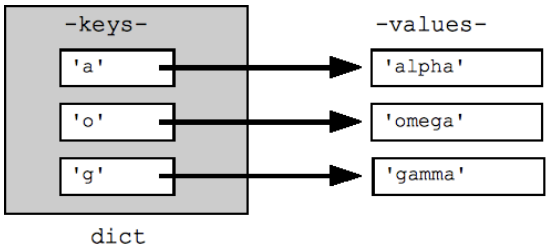

Figura 2.2: Atribuição de valor a uma chave

Outra das principais arquitecturas NoSQL é a de chave-valor, onde é possível associar um valor a uma chave, tal como num array associativo, ou um dicionário.

Alguns dos principais exemplos de BD NoSQL deste tipo são: *Memcache* (2.8.1), *Redis* (2.8.2), *Aerospike*, *FoundationDB*, entre outros.

Estas bases de dados normalmente não oferecem uma forma eficiente, ou nenhuma forma de todo, de efectuar pesquisas ao conteúdo dos valores armazenados uma vez que apenas tratam de associar um conjunto de bytes a uma chave.

2.3.3 Grafos

A arquitectura baseada em grafos permite armazenar dados através de nós e ligações, não havendo necessidade de indexação dos dados porque cada elemento tem referências para os todos os seus nós adjecentes.

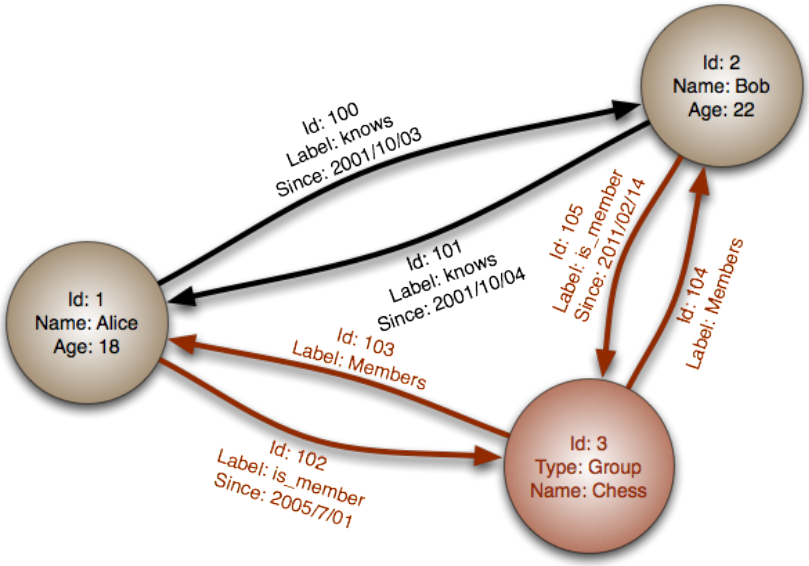

Figura 2.3: Exemplo de um grafo

Alguns exemplos de bases de dados baseadas em grafos são: *GraphDB*, *Neo4j*, *OrientDB*, entre outros.

2.4 Master/slave

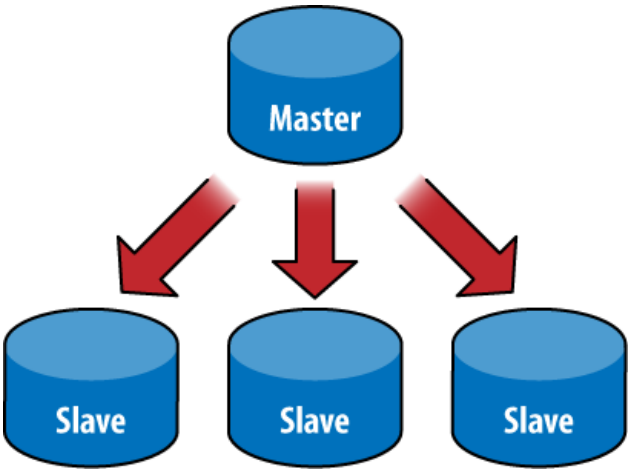

Figura 2.4: Master/Slave Replication

Uma forma de distribuir os pedidos de uma base de dados passa por criar cópias através de uma configuração *master/slave*, permitindo que a base de dados principal (*master*) seja replicada por várias nós secundários *slave*. [Rou08] Esta é também uma forma prática de criar redundâncias/backups da base de dados.

Uma vez que o master trata de sincronizar os dados entre os vários nós secundários, uma aplicação pode ser desenvolvido para efectuar as pesquisas e escritas directamente a um destes nós. Algumas SGBD permitem que sejam efectuadas escritas também nos nós slave, outros obrigam que sejam feitas apenas na base de dados principal.

Esta abordagem consegue resolver problemas de distribuição da carga de acesso e de integridade dos dados, mas pode também não ser perfeita.Todos os nós secundários necessitam de ter a mesma capacidade de armazenamento que o master. Manter réplicas implica também custos para a organização, isto porque será necessário haver mais máquinas dedicadas ao mesmo objectivo que o master. A largura de banda entre o master e o slave também pode se tornar num problema, caso algum dos slaves tenha um baixo débito os dados demorarão mais tempo a serem sincronizados com esse nó.

2.5 Sistemas de cache

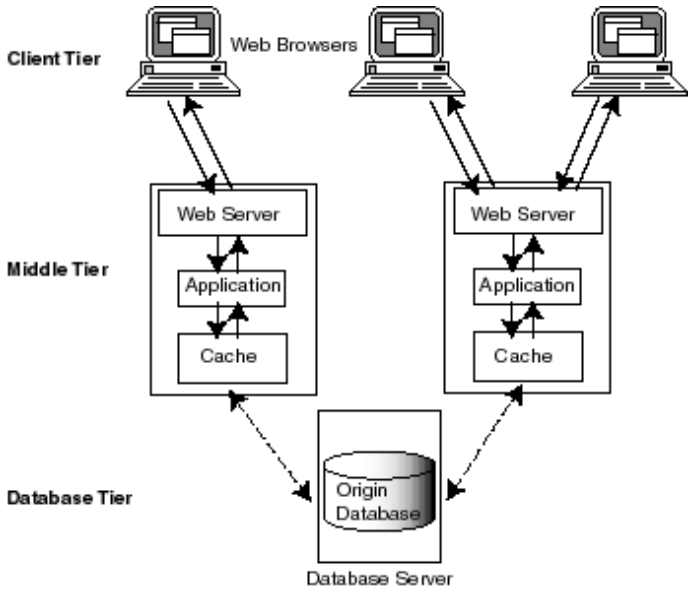

Figura 2.5: Caching de pedidos de páginas web

Hoje em dias cada vez mais são utilizados sistemas de cache para acelerar pesquisas a bases de dados, pesquisas de documentos ou até pedidos de páginas web. [ALK+02, LKM+02] Na figura 2.5 podemos observar a arquitectura de um sistema de cache de páginas Web.

Este tipo de sistemas estão limitados à quantidade de memória disponível na máquina em que se encontram a correr, pelo que necessitam de descartar alguma informação quando a memória disponível se esgota. Estes sistemas normalmente regem-se pelo mesmo princípio que as memórias cache do processador, defendendo que se um certo dado foi utilizado recentemente, provavelmente voltará a ser utilizado num futuro próximo. Desta forma, quando começa a esgotar a memória disponível os dados menos recentemente utilizados são descartados. [kn:d]

É comum estes sistemas oferecer também a possibilidade de definir o tempo de expiração de uma chave, tornando os dados guardados válidos apenas durante um determinado período de tempo, sendo apagados assim que expirados.

2.6 Caching de Bases de Dados

Este tipo de ferramentas oferecem uma solução pronta a funcionar que permite colocar, de forma transparentes, alguns conteúdos de uma BD em cache.

2.6.1 CSQL Cache

Esta ferramenta pode ser colocada em frente a alguns SGBD compatíveis, permitindo colocar em cache as tabelas de pesquisas efectuadas mais recentemente, aclamando conseguir um aumento de performance cerca de 20 vezes superior do que usando directamente os SGBD líderes. [CSQ, Capítulo 1.1] Algumas das interfaces compatíveis são o SQL, ODBC and JDBC tornando a sua utilização transparente para a aplicação, tal como se estivesse ligado directamente ao SGBD de eleição.

Algumas das BD que podem ser utilizadas são MySQL, Oracle, Sybase, DB2, etc.. [CSQ, Capítulo 1.1]

Todas as pesquisas efectuadas acerca desta ferramente resultaram em pouco ou nada, não existindo muitos exemplos ou ajuda de como a utilizar e a última versão deste software foi actualizada em 2011.

2.6.2 Oracle Database In-Memory

O *Oracle Database In-Memory* é um produto proprietário da Oracle, que oferece à sua família de bases de dados uma aceleração das pesquisas aproximadamente 1000 superiores e sem qualquer alteração na aplicação. [Eri]

Tal como qualquer outro sistema de cache, coloca em memória os dados usados mais frequentemente. [Ora]

2.6.3 MTcache

O *MTcache* trata-se de um protótipo de uma solução de cache para *Microsft SQL*, colocada numa camada intermédia entre a aplicação e a base de dados. [LGZ03]

Nesta camada é criada uma base de dados sombra, contendo as mesmas tabelas que a base de dados principal, incluído as restrições e índices mas estando as tabelas vazias. Os dados são acrescentados à cache através da criação de uma *vista* para a base de dados. Estas *vistas* são populadas e actualizadas pela BD através de subscrições. Os *updates*, *inserts* e *deletes* são reencaminhados transparentemente para a base de dados principal. [LGZ03, Capítulo 2]

2.7 Bases de Dados Híbridas

Uma base de dados híbrida trata-se de uma base de dados que suporta tanto armazenamento *in-memory* como armazenamento em disco. [kn:c] Existem várias bases de dados deste género como o caso do *Couchbase* (2.8.4) ou o *MongoDB* [kn:f].

Uma desvantagem deste tipo de bases de dados trata-se de como a memória de armazenamento é superior à RAM disponível, poderá haver sempre atrasos quando é necessário carregar dados que ainda não estejam em memória e descartar outros para disponibilizar espaço.

2.8 Algumas tecnologias actuais

2.8.1 Memcache

O *Memcache*, ou *Memcached* (como passou a se chamar nas suas últimas versões), é uma tecnologia com arquitectura *key-value* desenvolvida de raíz para funcionar apenas como cache, não permitindo qualquer persistência de dados.

Na sua inicialização é definida a quantidade de memória que poderá utilizar, descartando as chaves usadas menos recentemente quando a memória disponível se esgota. [kn:d] Também é possível definir um tempo de expiração para as chaves.

Existem outras BD compatíveis com o protocolo do *memcache* que podem ou não oferecer persistência, como o caso do *Couchbase* (2.8.4), *MemcacheDB* [kn:e], entre outros.

2.8.2 REDIS

O REDIS é uma bases de dados *in-memory* não relacional muito popular, do tipo *key-value*. [kn:b] Como qualquer outra BD esta armazena os dados em disco, mas também pode ser utilizada como um sistema de cache, desactivando a persistência e definindo uma política para descartar as chaves quando a memória disponível se esgota. [kn:k]

Existem muitas outras funcionalidades que o REDIS também oferece, sendo algumas delas:

- tempo de expiração de chaves
- criação de listas (que podem ser ordenadas).
- criação de *hashsets*, onde é possível associados várias propriedades a uma chave, podendo aceder ou alterar apenas as propriedades desejadas.
- utilização de *LUA* scripts no lado do servidor.
- mecanismos de comunicação distribuída de publicação/subscrição.
- Redis Clusters.

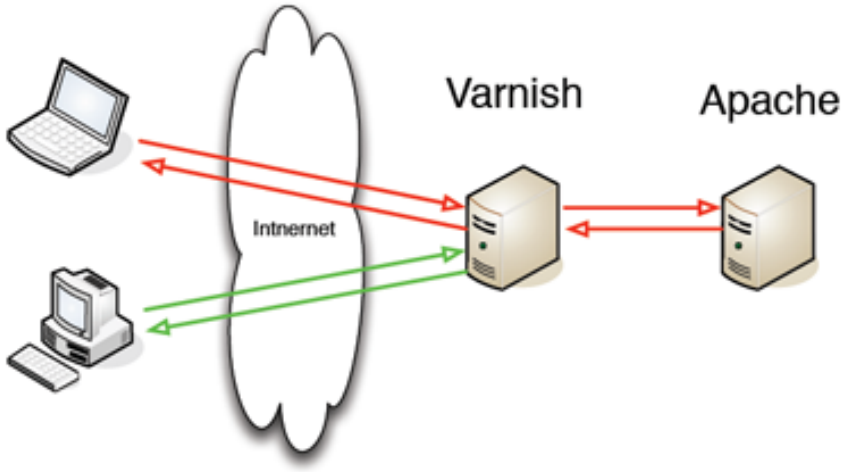

Figura 2.6: Utilização do Varnish como serviço de cache

2.8.3 Varnish

O Varnish é um acelerador de páginas Web. Ao contrário dos sistemas anteriores, este dedica-se exclusivamente a conteúdo Web e aclama conseguir acelerar entre 300 a 1000 vezes a entrega deste conteúdo comparado com um servidor http normal. [kn:l]

Este é um serviço colocado em frente do servidor http (ver figura 2.6), que coloca em cache conteúdos estáticos como páginas html, imagens, documentos, ou qualquer outro tipo de conteúdo, e pode funcionar também como um load balancer.

2.8.4 Couchbase

O *Couchbase* é uma BD NoSQL orientada a documentos JSON. Os documentos são mantidos em *buckets*, sendo comum criar um *bucket* para cada tipo de documento. [kn:a]

Existem no entanto dois tipos de *buckets* com características bem distintas.

- **couchbase buckets**: permite persistência e replicação, limitado a documentos com um tamanho máximo de 20 Megabytes.
- **memcached buckets**: não permitem persistência nem replicação e estão limitados a documentos com o tamanho máximo de 1 Megabyte.

O *Couchbase* pode ser configurando em modo individual ou em modo *cluster*. Este último modo permite *sharding* dos documentos entre os vários nós (escalonamento horizontal).

2.8.5 SQLite

O *SQLite* é uma base de dados relacional criado com intuito de ser embutido em aplicações através de uma biblioteca. Este tipo de base de dados é muito útil em projectos que não necessitam

de um SGBD completo, apenas necessitam de guardar dados de uma forma simples e continuando a tirar proveito das capacidades da linguagem SQL.

2.9 Resumo ou Conclusões

Uma base de dados trata de representar o *domínio do discurso* de uma organização, através de classes com relações entre si, compostas por atributos e propriedades.

Esta informação deve de alguma forma persistir para poder ser consultada no futuro.

Os SGBDs oferecem um standard que possibilita a escrita e leitura dos dados presentes numa base de dados, sendo o SQL o mais comum nas BD relacionais.

O ponto de estrangulamento de muitos sistemas trata-se da comunicação com a BD, uma vez que quanto maior for a organização e mais acesos concorrentes estiverem a ser feitos, mais lentas serão as respostas dos SGBDs. Uma forma de acelerar o processo e reduzir o número de acessos concorrentes a uma BD, passa por utilizar sistemas de cache numa camada intermédia.

Como referido nas várias secções do capítulo 2.6, os sistemas de cache oferecem um aumento de performance de 300 a 1000 vezes superiores quando presentes, facilmente é possível identificar os benefícios da sua utilização em grandes organizações. A principal desvantagem é que estes estão limitados à quantidade de memória disponível na máquina, pelo que é necessário identificar quais os dados que deverão permanecer e quais deverão ser descartados quando a memória disponível se esgotar. Normalmente permanece em memória as entradas mais recentemente utilizados.

Capítulo 3

Framework de Cache Pesquisável e de Alta Performance

3.1 Problema

As tecnologias de cache *in-memory* actuais mais usadas são normalmente baseadas numa arquitecura *key-value* (ver 2.3.2). Estas tecnologias apenas tratam de associar a uma chave um conjunto de bytes, e como não têm conhecimento sobre o seu conteúdo é impossível efectuar quaisquer pesquisas. Uma possível solução para pesquisar sobre este conteúdo seria retirar todos os dados da cache para a aplicação e efectuar a pesquisa localmente, mas com grandes *data sets* a performance seria inaceitável. Outra solução seria aproveitar tecnologias que suportam *scripting*, como o caso do *REDIS*, e efectuar esta pesquisa no lado do servidor, mas mais uma vez, esta pesquisa só seria possível com alguma manipulação dos conteúdos associados a cada chave e percorrendo o *data set* inteiro, trazendo na mesma problemas de performance.

3.2 Solução

A abordagem tomada para solucionar este problema passou por criar uma framework composta por um conector a ser utilizado pelo cliente e um serviço de cache a funcionar em rede, com suporte a pesquisas.

O serviço de cache não implementa qualquer persistência de dados, e permite associar a uma chave um conjunto de bytes. Não existe descriminação do conteúdo destes bytes, podendo tanto representar um objecto ou um ficheiro.

Esta framework foi implementada com a linguagem de programação C# com a biblioteca *.NET 4.5*. A tecnologia de cache candidata escolhida para o ser utilizada no serviço de cache foi o *SQLite* utilizando bases de dados *in-memory*.

As outras tecnologias candidatas eram o *REDIS*, *Memcache*, *MySQL* com as tabelas do tipo *MEMORY*, *Couchbase* e *MySQL Cluster*. Na secção 3.4 podem ser lidas algumas razões pelas

quais algumas destas tecnologias não foram escolhidas e não passaram para a fase de desenvolvimento.

Para o conector comunicar com o serviço de rede foi criado um protocolo de comunicação específico que funciona com *sockets TCP/IP assíncronas*, podendo este ser consultado extensivamente e em detalhe no Anexo A.

Um exemplo prático de implementação e utilização da framework, cobrindo todas as as secções deste capítulo, pode ser também consultado no Anexo B.

3.2.1 SQLite

Com o desenvolvimento do projecto, e após testadas diferentes tecnologias, o SQLite foi escolhido entre as várias candidatas devido à possibilidade de criação de tabelas *in-memory*, que nunca chegam a tocar o meio de armazenamento, e pelo suporte a colunas binárias (*BLOB*) de tamanho variável.

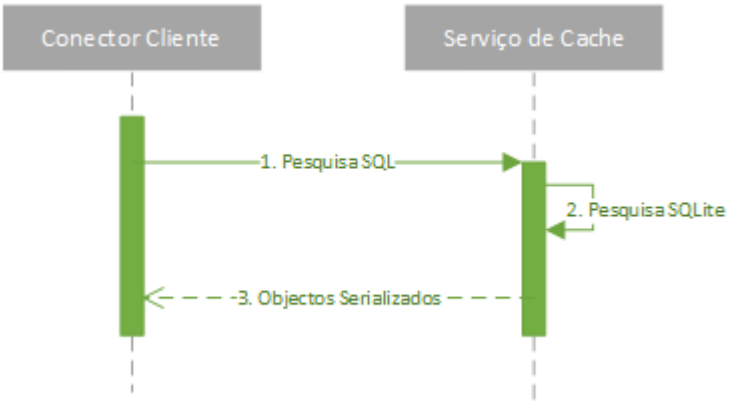

Figura 3.1: Diagrama de Sequência de uma pesquisa em cache

3.2.2 Serialização

Por motivos de generalização e eficiência, o serviço de cache não necessita de conhecer o conteúdo dos objectos que estão a ser guardados. Este apenas recebe um conjunto de bytes que podem ser de um objecto serializado, como podem ser de um ficheiro. Desta forma, os objectos necessitam de ser serializados em bytes no lado de cliente antes de ser enviados para o serviço de cache.

Felizmente o C# suporta serialização nativa, e fornece uma API simples que efectua serialização e desserialização dos objectos. Infelizmente a serialização nativa não é suficientemente rápida e eficiente em termos de espaço ocupado em bytes devido a um acréscimo de *metadados*, nem em termos de velocidade. Estes *metadados* são úteis em termos genéricos porque permitem efectuar diversas verificações no momento da deserialização, um exemplo é o versionamento da classe que permite verificar se a sua implementação não foi modificada após a instância ter sido serializada. Embora seja informação útil é desnecessária num ambiente que é à partida é controlado

pelo programador, e torna-se ineficiente ocupar dados extra no serviço de cache. Como método de serialização alternativo foi escolhido o *Protocol Buffers* (*protbuf*).

O *protbuf* é um protocolo criado pela Google e é utilizado em quase toda a comunicação de mensagens interna entre os seus próprios serviços. Este protocolo permite uma serialização estruturada de mensagens numa linguagem neutra de forma extremamente rápida e eficiente em termos de espaço ocupado. [kn:i]

Para C# existe a biblioteca *protobuf-net*, disponível no sistema de pacotes *Nuget* da Microsoft. Para uma comparação detalhada entre diferentes formas de serialização em C# , a seguinte referência pode ser consultada [kn:h]

Este protocolo é também utilizado internamente para a troca de mensagens entre o conector e o serviço de cache implementado.

Para efectuar a serialização e por consequente a desserialização é necessário especificar o protocolo da mensagem, através de um ficheiro *.proto*. A biblioteca *protobuf-net* efectua a criação deste protocolo de forma transparente para o programador.

```
1 message Person {
2   required int32 id = 1;
3   required string name = 2;
4   optional string email = 3;
5 }
```

Listing 3.1: Exemplo da especificação de uma mensagem em protobuf (.proto)

O *protobuf-net* permite a inclusão de instâncias de subclasses dependentes, bem como elementos de listas e dicionários no momento da serialização, tornando possível de uma só vez serializar um objecto acompanhado de todas as suas dependências.

3.2.3 Parâmetros Pesquisáveis e Estruturação

Nem todos os parâmetros dos objectos podem necessitar de ser pesquisáveis, e visto que o serviço não tem conhecimento do conteúdos dos objectos que são guardados não é possível efectuar pesquisas nos conteúdos deste objecto. Desta forma, o conteúdo dos parâmetros pretendidos de serem utilizados em pesquisas necessitam de ser enviados juntamente com o obejecto serializado, como *metadados*. Se todos os campos forem marcados como pesquisáveis haverá duplo armazenamento, não sendo eficiente em termos de espaço ocupado embora a nível de performance não existe quaisquer penalizações. Isto porque na transferência para o cliente continua apenas a ser enviado um único objecto serializado, e não vários parâmetros individuais.

No lado do serviço, tanto os parâmetros como o objecto serializado são guardados numa tabela in-memory do *SQLite*. Os campos pesquisáveis podem ser de texto, inteiros, longos ou reais, à excepção do objecto que é enviado em bytes e guardado numa coluna de tipo binária (*BLOB*). A estruturação de cada objecto/entidade necessita de ser enviada pelo cliente na inicialização da aplicação, podendo ser automatizada, como é possível ver na secção 3.2.5.

É também possível definir se estes campos irão originar a criação de índices, e se podem ser nulos.

3.2.4 Modularidade

A framework foi desenvolvida tendo em conta a modularidade, permitindo que a qualquer momento o serviço que fornece a cache seja modificado de forma completamente transparente para o utilizador. Como é possível ver no diagrama da figura (3.2) todos os serviços de cache implementam a mesma interface *IQueryCache*.

No serviço desenvolvido, o próprio sistema de cache pode ser modificado directamente no ficheiro de configuração XML da aplicação, uma vez que a instanciação do tipo de cache é feita por injecção de dependências através do Unity da Microsoft. [kn:j]

O conector utilizado pelo cliente também implementa esta interface, tornando possível utilizar a cache directamente na aplicação sem a utilização da rede, por exemplo com o *LocalSQLiteCache*, em vez do conector que comunica com o serviço de cache através da rede, o *RemoteServiceCache*.

3.2.5 Cache Helper

De forma a facilitar o esforço necessário para configurar e utilizar o serviço de cache foi desenvolvido uma classe de ajuda que utiliza *reflexão* para automatizar este processo. Para o programador apenas é necessário inserir algumas anotações nas classes que pretendam ser utilizadas pelo serviço (anotação *[CacheEntity]*). Para definir o campo primário apenas é necessário inserir sobre o parâmetro pretendido a anotação *[PkCacheParameter]*. A anotação *[CacheParameter]* permite informar ao helper quais os campos da classe que podem ser pesquisados, e por consequente serão enviada juntamente com o objecto serializado no momento da inserção.

Este helper trata também de criar automaticamente o *schema* das classes quer será enviado na inicialização para o serviço de cache. Este *schema* permite que o serviço crie todas as estrutura que irão manter os objectos desta classe quando estes forem inseridos. No caso do SQLite, estas estruturas são tabelas numa base de dados in-memory, e os parâmetros pesquisáveis bem como o objecto serializado colunas desta tabela.

Se o serviço de cache for reinicalizado e a aplicação do cliente tentar inserir ou pesquisar dados na cache, este helper irá apanhar uma excepção indicando que a estrutura de dados não se encontra inicializada, fazendo também uma reinicialização automática e transparente para o cliente.

3.2.6 Connection Pooling

Um aspecto muitas vezes desprezado por completo pelos programadores é o da criação de ligações *TCP/IP* que são dispendiosas para o sistema operativo, por isso é uma boa prática partilhar uma ligação assim que esta já não seja precisa. O método mais comum, e mais prático, é através da utilização *Connection Pools*. Sempre que seja necessário uma ligação, esta é pedida a uma *pool* (piscina), e quando tivermos terminado a sua utilização, é retornada para a *pool*. Um outro benefício é que no momento de o cliente pedir uma ligação à *pool*, podemos verificar se a ligação

ainda se encontra activa e criar uma nova em caso de ter sido terminada por alguma razão. São controlado também alguns aspectos como o número máximo de ligações activas e/ou ligações em *stand-by*.

Esta *Connection Pool* implementa a interface *IQueryCache*, significando que ela própria efectua toda a gestão de pedidos e retorno de ligações de forma transparente, o programador apenas necessita de especificar o delegate que irá criar instanciar as novas ligações. Ao implementar a interface utilizada que todos os serviços de cache implementam também pode ser utilizada directamente pelo *Cache Helper*.

Um exemplo da utilização deste conceito pode ser consultado no Anexo B.

3.2.7 Expiração de Dados

Ao serem inseridos dados no serviço, pode ser definido opcionalmente uma data de expiração para os objectos. Durante as pesquisas, caso a data tenha expirado, o objecto não será retornado.

Existe uma rotina em *background* no serviço que corre periodicamente num tempo configurável. O objectivo é apagar todos as entradas em que a data tenha expirado.

3.2.8 Modos de leitura

O conector do cliente, oferece dois modos de leitura, um modo que envia os dados todos de uma vez sequencialmente, e outro que envia os dados à medida que são pedidos, modo *reader*.

O modo *reader* é um modo eficiente em termos de utilização e foi desenvolvido como um melhoramento para um serviço de cache. Inicialmente foi verificado que o serviço consumia largas quantidades de memória quando eram pedidos uma lista extensa de objectos. Isto acontecia porque o serviço alocava toda a quantidade de memória necessária para alojar os objectos pedidos que de seguida eram enviados um a um por socket para o cliente. Sendo o envio por rede uma operação lenta, não havia necessidade de existir dados parados no serviço a ocupar memória, à expera que chegasse a sua vez de serem entregues. Este modo solucionou a utilização excessiva e desnecessária de memória no lado do serviço, porque o próprio SQLite já retorna cada objecto um a um de cada vez. Neste caso, só é pedido ao SQLite o próximo objecto assim que o anterior tenha sido enviado por rede. No caso da aplicação do cliente, permite receber e processar individualmente cada objecto à medida que chega pela rede, sendo útil quando apenas é necessário consultar alguns valores do objecto e/ou não é necessário mantê-los guardados numa lista.

3.2.9 Outras optimizações

Na comunicação entre o conector e o serviço de cache, foram utilizados alguns *buffers* para a comunicação que se revelaram num grande aumento da performance inicial do serviço. Por defeito, as classes nativas de comunicação por sockets da biblioteca *.NET* utilizam o algoritmo de *Naggle* que já inclui um buffer interno. Este algoritmo permite enviar os dados apenas quando se justificam, por exemplo, quando o *buffer* tiver cheio ou existir uma quantidade de dados considerável à espera de serem enviados. Em contraste, este algorítmo poderia ser completamente

desactivado, mas a performance também não seria a ideal, causando que todos os dados enviados em rede ficassem bloqueadas à espera que os anteriores fossem entregues (*ACK*). Aumentar o tamanho do buffer deste algorítmo também só piorou a performance, e embora já existisse um ganho ao utilizar este algoritmo, não era o suficiente. Nos testes iniciais, numa rede 100 Mbits, foi notado que o serviço não estava a atingir as velocidades de transferência desejadas. Para solucionar este problema foram utilizadas *buffers* de mais alto nível para leitura (*BufferedReader*) e escrita em *streams* (*BufferedWritter*). Ao utilizar estes *buffers* com tamanhos de 64 *KB*, tanto no cliente como no serviço, e na rede em questão, passou logo a ser possível consumir 100% da largura de banda da rede. Noutros testes efectuados numa rede *Gigabit*, embora com alguma utilização de outros computadores, 64 KB foram suficientes para atingir velocidades de transferência superiores a 450 Mbits/s.

3.2.10 Desqualificação de índices nas pesquisas

O *SQLite* (bem como outras bases de dados), só consegue utilizar um índice para optimizar cada pesquisa mesmo que exista mais que um campo na mesma pesquisa com índices associados. A escolha do índice é feita automaticamente pelo *SQLite*, embora às vezes possa não ser a escolha acertada.

É possível, deixar o programador escolher quais os índices que não queria que sejam utilizados, colocando o operador **+** antes do nome do campo.

Por exemplo, nesta query "*WHERE ID BETWEEN 1 AND 50000 ORDER BY NAME DESC*", o *SQLite* escolhe utilizar o índice do campo *ID* por ser a chave primária. Infelizmente esta pesquisa tem uma performance muito inferior do que se fosse utilizado o índice do campo NAME na ordenação dos resultados. A query final ficaria: "*WHERE +ID BETWEEN 1 AND 50000 ORDER BY NAME DESC*".

A desvantagem de utilizar este operador antes de um campo, é a perda do tipo de afinidade associado a este campo, que pode mudar completamente o sentido da pesquisa. Por exemplo, se o campo *ID* fosse do tipo texto, ao utilizar o operador **+** na cláusula *+ID=5*, a comparação seria efectuada em modo numérico, retornando sempre falso. Esta e muitas outras optimizações podem ser consultadas em [kn:g].

3.2.11 Operações do Serviço de Cache

De seguida encontra-se uma lista com todas as operações disponibilizadas pelo serviço de cache, que são também encapsuladas e em grande parte automatizadas pelo *Cache Helper*.

3.2.11.1 *Prepare*

Esta operação tem de ser chamada pelo conector, idealmente no arranque do programa e uma vez por cada classe que será colocada ou pesquisada da cache. O seu objectivo é permitir enviar um *PrepareDocument* indicando o nome da entidade, o nome e tipo do campo primário, bem como os nomes e tipos dos campos pesquisáveis. No serviço, é criada uma tabela com esta informação.

O servidor gera uma hash do pedido que é associada ao nome desta entidade, para o caso de outro cliente efectuar um pedido de preparação de uma entidade diferente, mas com o mesmo nome. Neste caso o servidor irá informarar que a entidade já existe, ou pelo menos tem uma estrutura diferente.

3.2.11.2 *Request*

Esta é a operação utilizada para efectuar pedidos/pesquisas ao serviço de cache, sendo retornados numa lista. Estes pedidos são feitos com o formato SQL, e são anexados após uma cláusula *WHERE* implícita, sendo possível acrescentar cláusulas *SORT* se necessário.

3.2.11.3 *RequestReader*

Esta operação é semelhante à anterior e é utilizada para retornar os dados através de um reader *IQueryCacheReader*, em vez de ser retornados em lista (3.2.8).

3.2.11.4 *Insert*

Esta operação permite inserir um ou mais objectos no serviço de cache. Esta inserção é feita com um *InsertDocument* que além de conter a instância do objecto a ser inserido, permite definir o conteúdo dos campos pesquisáveis. O *Cache Helper* desenvolvido automatiza completamente este processo, apenas é necessário passar uma ou mais instâncias a serem enviadas para a cache.

É possível passar um campo opcional que define o tempo de expiração dos dados, caso não seja passado os dados inseridos ficam com tempo de expiração indefinido.

Se for enviada uma lista de objectos, estes são inseridos numa única transação no *SQLite*.

3.2.11.5 *Delete*

Esta operação permite definir uma cláusula *WHERE* para apagar dados de uma certa entidade no serviço de cache.

3.2.11.6 *Flush*

A operação *Flush* permite apagar por completo todo o conteúdo da cache, simulando um restart no serviço.

3.2.11.7 *IsConnected*

Esta é uma propriedade que deve reflectir o estado da ligação ao serviço, retornando *true* quando a ligação se encontra activa e *false* em caso contrário.

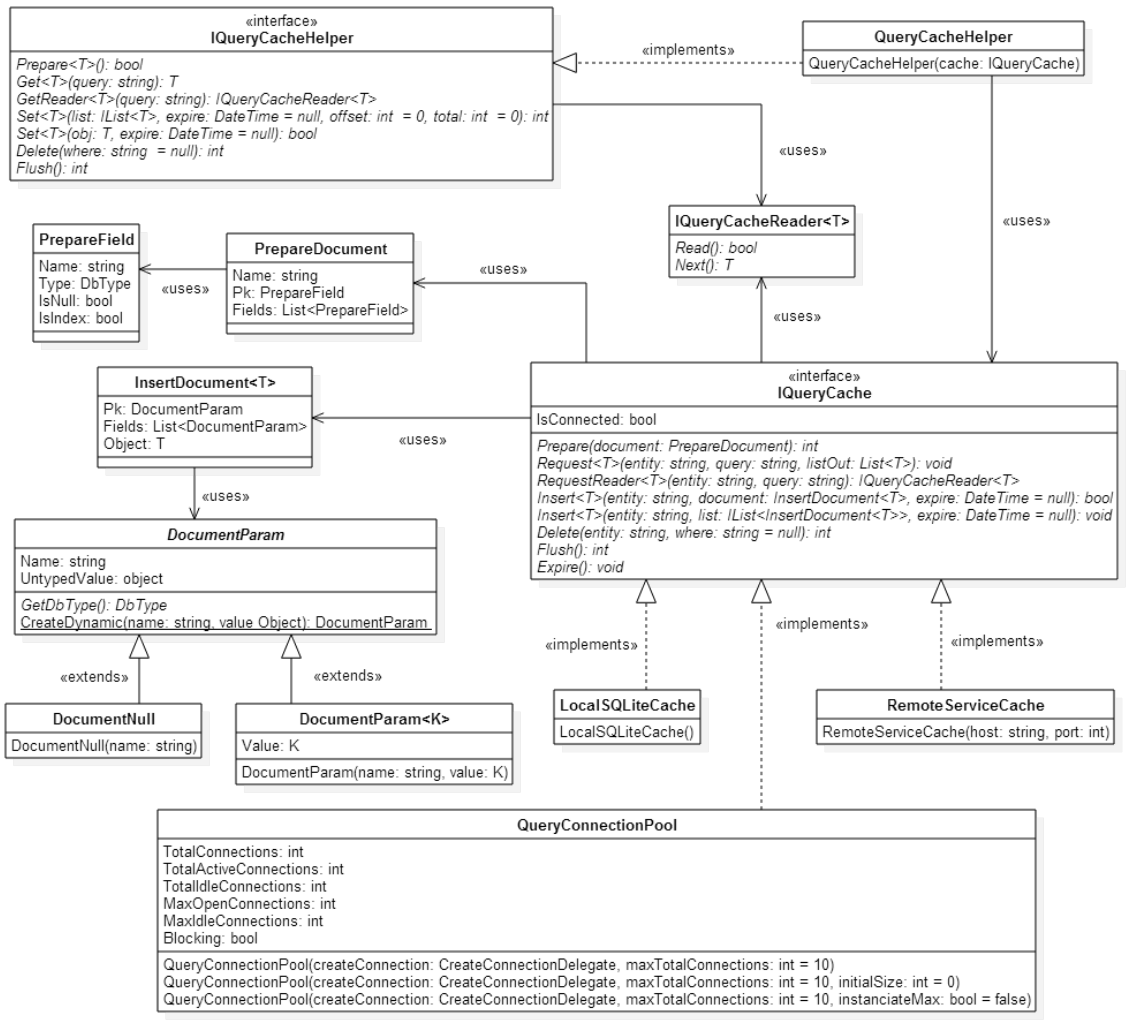

Figura 3.2: Diagrama de Classes da Framework de Cache

3.3 Projectos da Solução

Os projectos presentes nesta solução, com fins diferentes, encontram-se listados abaixo:

- **glintt-cache-lib**: dll com todo o código necessário para utilizar o serviço de cache.
- **glintt-cache-service**: Serviço de Cache em *Windows Service*, para ambiente de produção.
- **glintt-async-cache-server**: Serviço de Cache em modo consola, ideal para desenvolvimento.
- **Example**: Aplicação com exemplos de utilização do código.
- **TestProfiler**: Aplicação com testes de medição de performance das diferentes tecnologias de Cache.

3.4 Outras tecnologias de Cache

Nesta secção são apresentadas algumas outras tecnologias, e as suas razões pelas quais não foram escolhidas ou consideradas.

3.4.1 Memcache/Memcached

O *REDIS* ultrapassou fortemente o *Memcache* nas últimas versões, muito por continuar a ser activamente desenvolvido, acompanhando o desenvolvimento da tecnologia e as necessidades dos programadores. Em termos de performance ambos funcionam de forma muito semelhante para consultas de chaves individuais, mas o *REDIS* oferece uma operação de consulta de várias chaves (*MGET*) em apenas um único pedido, tornando extremamente eficiente quando necessário consultar várias chaves. Neste momento existem apenas duas razões que justificam a escolha do *Memcache* sobre o *Redis*:

- Simplicidade de configuração do *Memcache*, já que todas as configuração possíveis são feitas através de flags ao executar a aplicação, não existindo quaisquer ficheiros de configuração.
- Necessidade de manter compatibilidade com projectos antigos (*legacy*).

Com a operação *MGET* do *REDIS* e visto este ser a escolha óbvia para novos projectos, o *Memcache* não passou para a fase de desenvolvimento. Foram feitos alguns testes iniciais mas rapidamente não se justificou continuar a ser investido tempo com a implementação ao ponto de ser testado com o *data-set* utilizado para os teste de avaliação, quando podiam ser testadas outras tecnologias.

3.4.2 REDIS

Para o caso do *REDIS* (2.8.2), tecnologias baseada em aquitectura *key-value*, uma vez que não existe suporte a pesquisas, a abordagem implementada passou por efectuar as pesquisas SQL directamente na base de dados, sendo retornando apenas os identificadores dos resultados obtidos. Estes identificadores são de seguida pedidos ao *REDIS*, que retorna os objectos serializados com o *protobuf*.

No diagrama de sequências 3.3 é possível verificar como é feita esta comunicação.

Esta tecnologia encontra-se funcional e é compatível com o projecto na fase final.

3.4.3 MySQL

O *MySQL* suporta tabelas em memória (*MEMORY*) semelhante ao *SQLite*, mas as colunas *BLOB*/*BINARY* necessitam da definição de um tamanho fixo, tornando impraticável a sua utilização para guardar os objectos serializados.

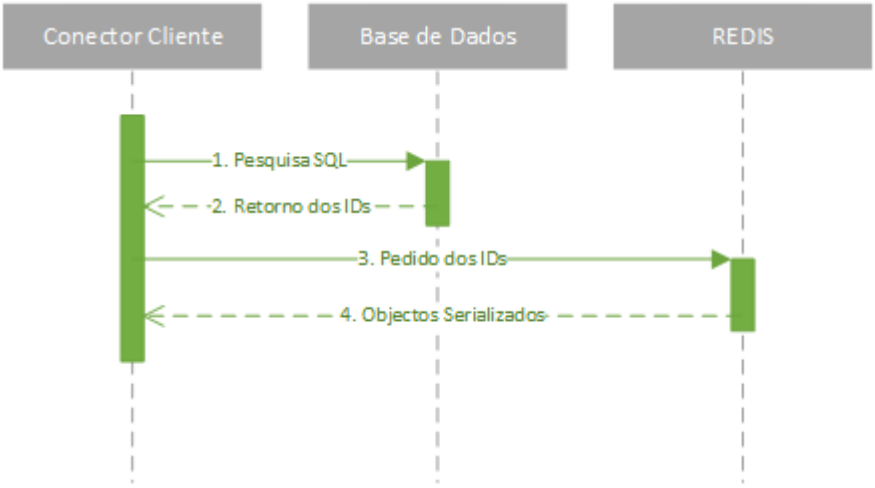

Figura 3.3: Diagrama de Sequências utilizando REDIS

3.4.4 MySQL Cluster

Como alternativa ao *MySQL*, o *MySQL Cluster* oferece um tipo de tabelas in-memory chamado *NDBENGINE*, que suporta colunas *BLOB* com tamanho variável. Para instalar esta tecnologia, é obrigatório haver no mínimo duas instâncias do *mysql server*, e que podem ser instaladas na mesma máquina. A grande desvantagem desta tecnologia é necessitar de grandes quantidades de memória e grandes capacidades de processamento.

A abordagem tomado para utilizar esta tecnologia é a mesma que o *SQLite*, embora não tenha sido desenvolvida até a fase final.

3.4.5 Couchbase

O *Couchbase*, é uma base de dados híbrida, que pode funcionar em cluster e permite manter em memória todos os conteúdos desde que haja memória disponível. Quando o *data set* é superior à memoria disponível, apenas são carregados da memória de armazenamento os dados mais recentemente pedidos.

Para efectuar as pesquisas, foi utilizado o *N1QL DP4 (developer preview 4)*. Embora não se encontre ainda preparado para ser utilizado em produção, já dá para testar e efectuar pesquisas com uma sintaxe semelhante e compatível com o SQL. Por enquanto, o *N1QL* é executado como uma aplicação independente do *couchbase server*, e o conector usado no C# trata-se também de uma versão beta.

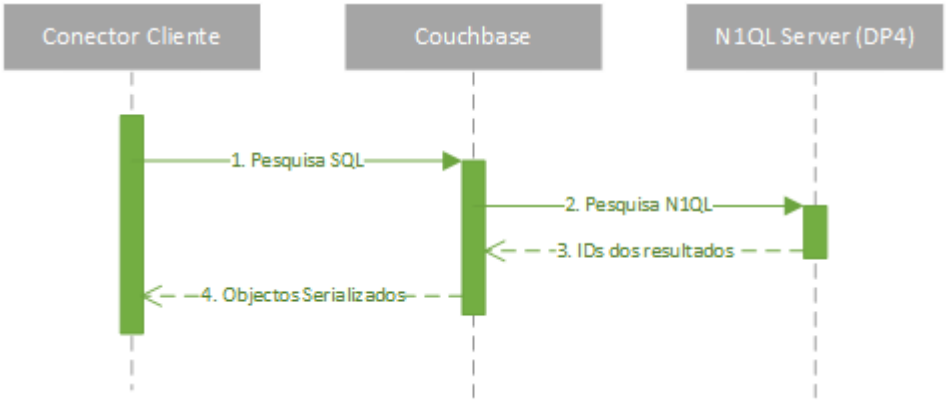

Figura 3.4: Diagrama de Sequências utilizando Couchbase

Capítulo 4

Avaliação Empírica

Nesta capítulo são apresentados alguns resultados dos testes efectuados, sendo possível analisar os tempos e avaliar a performance das abordagens implementadas com as diferentes tecnologias.

4.1 Data set

O *data set* utilizado em todos os testes consiste numa classe *Pessoa*, composto pelas propriedades que podem ser consultados no diagrama 4.1.

Cada objecto foi gerado com dados aleatórios, sendo os identificadores numerados incrementalmente. A propriedade *BinData* contém *2KB* de dados, somando em aproximadamente 2090 bytes o tamanho final de cada objecto.

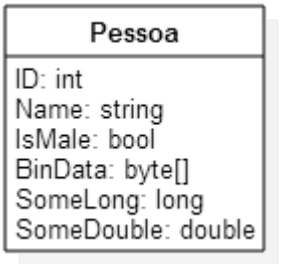

Figura 4.1: Diagrama da classe Pessoa

4.2 Ambientes de Testes

Foram utilizados dois ambientes distintos com os testes efectuados, que são descrito de seguida:

4.2.1 Ambiente 1 - VM

Neste ambiente, o serviço de rede e as tecnologias de cache foram instaladas numa máquina virtual, ficando a aplicação de testes na máquina física.

Este máquina física era composta por:

- Windows 8.1 64 bits
- 8 Gigabytes de RAM DDR3
- *Intel i7-3630QM*, 4 cores e 8 threads

A máquina virtual foi criada com as seguintes características:

- Windows 8.1 64 bits
- 3.4 Gigabytes de RAM
- 4 cores virtuais
- Interface de Rede virtual com 1 Gigabit de largura de banda

A utilização de uma máquina virtual não influencia quaisquer conclusões que possam ser tiradas, isto porque todas as tecnologias testadas estão sujeitas às mesmas condições.

4.2.2 Ambiente 2 - Local

Neste ambiente, o serviço de rede, as tecnologias de cache e a aplicação de testes encontravam-se a funcionar todos na mesma máquina física do ambiente 1 (4.2.1).

4.3 Terminologia utilizada

A terminologia utilizada para descrever as tecnologia utilizadas em cada teste, foi a seguinte:
Nome[-r][-b]
Tendo o seguinte significado:

- **Nome**: Nome da tecnologia utilizada.
- **-r**: Foi utilizado o modo de leitura *reader*.
- **-b**: Os dados não foram desserializados após serem pedidos à cache, ficando em bytes.

Os valores entre parêntesis rectos **[...]** são opcionais.

4.4 Definição de um teste

Existem 3 características que definem um teste.

1. Tecnologias utilizadas.
2. Lista de objectos a pedir (*queries*).
3. Número de vezes a repetir cada teste.

Por exemplo, num teste com a tecnologia **A** e **B**, a serem pedidos 1 objecto, 10 objectos e 100 objectos de cada vez, com cada teste repetido 30 vezes, a ordem de execução seria a seguinte:

1. **30** vezes pedido **1** objecto à tecnologia **A**.
2. **30** vezes pedido **1** objecto à tecnologia **B**.
3. **30** vezes pedidos **10** objecto à tecnologia **A**.
4. **30** vezes pedidos **10** objecto à tecnologia **B**.
5. **30** vezes pedidos **100** objecto à tecnologia **A**.
6. **30** vezes pedidos **100** objecto à tecnologia **B**.

A *query* para pedir o número de objectos pretendidos foi criada da seguinte forma:

- **1 objecto:** "*ID = 1*"
- **10 objectos:** "*ID BETWEEN 1 AND 10*"
- **100 objectos:** "*ID BETWEEN 1 AND 100*"
- ...
- **50000 objectos:** "*ID BETWEEN 1 AND 50000*"

4.5 Testes Efectuados

Vários testes foram desenvolvidos para avaliar diferentes situações, em diferentes condições, utilizando o *data set* descrito anteriormente (4.1) e os ambientes listados em 4.2.

No fim de cada teste foi calculada a média e o desvio padrão dos tempos de resposta.

Em todos os testes, a base de dados utilizada foi o *SQL Server Express 64bit 12.0.2*, e o *REDIS 64bit 2.8.17*.

Todos os gráficos utilizados nesta secção dependem da cor para sua interpretação.

4.5.1 Teste 1 - Todas as tecnologias

Este teste foi desenvolvido para comparar a performance entre as tecnologias: *REDIS*, *SQLite*, *MySQL Cluster* e *Couchbase*, referidas na secção 3.4. O ambiente de testes utilizado foi o 1 (4.2.1).

Também foram medidos os tempos de acesso à base de dados, que se encontrava a funcionar na mesma máquina que os restantes serviços.

Cada tecnologia foi individualmente testada 10 vezes, e a lista de objectos pedidos foi: 1, 100, 1000 e 5000.

4.5.1.1 Descrição das tecnologias

Neste teste foram testadas cinco abordagens com diferentes tecnologias.

- **couchbase**: Pedidos efectuados a uma base de dados Couchbase com N1QL e *couchbase buckets*.

- **mysqlcluster**: Pedidos efectuados directamente a um mysql cluster com tabelas em *NDBE-GINE*. Não existe um serviço de cache.

- **redis**: Teste efectuado com o conector do redis. Não existe um serviço de cache, o conector do cliente efectua as pesquisas ligando-se a uma base de dados que retorna os identificadores dos resultados, de seguida o conector do cliente pede estes identificadores à instância do redis.

- **sqlite**: Utilização de um serviço de cache em rede com sqlite.

- **sqlserver**: Base de dados principal com os dados normalizados.

4.5.1.2 Resultados

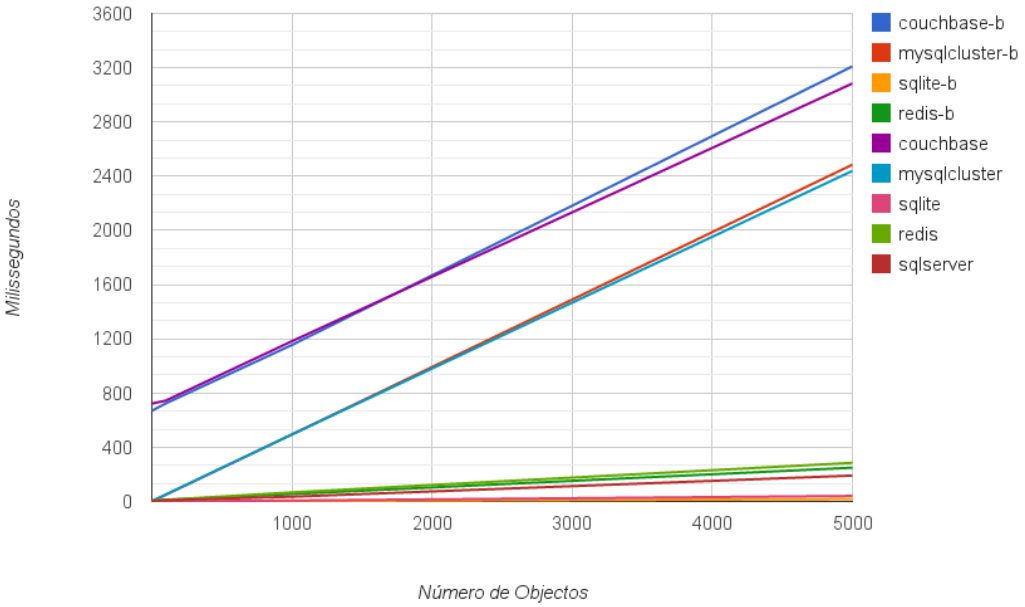

Figura 4.2: Gráfico da Performance do teste 1

Teste / Objectos	**1**	**100**	**1000**	**5000**
couchbase-b	669	723	1154	3210
mysqlcluster-b	1	49	494	2485
sqlite-b	<1	<1	3	18
redis-b	6	11	55	249
couchbase	722	744	1182	3084
mysqlcluster	2	51	494	2439
sqlite	<1	<1	7	41
redis	5	12	67	285
sqlserver	5	9	35	191

Tabela 4.1: Média de tempos do teste 1 (em milissegundos)

Teste / Objectos	1	100	1000	5000
couchbase-b	13.77	21.86	27.81	125.83
mysqlcluster-b	1	6.77	17.75	87.5
sqlite-b	<1	<1	<1	3.41
redis-b	1.66	2.82	7.4	16.23
couchbase	43.56	35.47	61.35	86.17
mysqlcluster	1	4.93	23.52	41.82
sqlite	<1	<1	<1	3.51
redis	<1	2.4	10.04	33.75
sqlserver	<1	3.16	5.07	30.86

Tabela 4.2: Desvio padrão do teste 1 (em milissegundos)

4.5.1.3 Análise dos Resultados

Como já era de esperar, o *Couchbase* não iria ter uma boa performance, devido ao *N1QL* ser ainda apenas um *developer preview* que é totalmente desaconselhado de ser usado em produção.

O *MySQL Cluster* também não teve uma muito boa performance quando comparado com as restantes tecnologias. Embora as tabelas criadas sejam *in-memory*, a sua arquitectura, composta por redudâncias, nós de controlo e instâncias de base de dados, impedem que a performance atingida seja a ideal.

Os testes ao *REDIS* e *SQLite* foram feitos com os conectores desenvolvidos na fase final, enquanto os restantes foram utilizados da fase inicial de pesquisa, dando para perceber que devido à sua má performance o porquê de terem sido abandonados e não terem passado para a fase de desenvolvimento.

4.5.2 Teste 2 - Redis e SQLite

Este teste foi desenvolvido para avaliar a performance na fase final de desenvolvimento, entre as tecnologias *REDIS* e o *SQLite*, no ambiente de testes 2 (4.2.2).

Também foram medidos os tempos de acesso à base de dados, que se encontrava a funcionar na mesma máquina que os restantes serviços.

Cada tecnologia foi individualmente testada 100 vezes, e a lista de objectos pedidos foi: 1, 100, 1000 e 5000, 10000, 25000 e 50000.

4.5.2.1 Descrição das tecnologias

Os quatro tipos de tecnologias testadas foram:

- **sqlite-in**: Este tipo de testes é feito com o sqlite a funcionar no próprio contexto da aplicação de testes, ou seja, não existe qualquer serviço de rede porque o serviço de cache funciona embutido na própria aplicação.

- **sqlite**: Utilização de um serviço de cache em rede com sqlite.

- **redis**: Teste efectuado com o conector do redis. Não existe um serviço de cache, o conector do cliente efectua as pesquisas ligando-se a uma base de dados que retorna os identificadores dos resultados, de seguida o conector do cliente pede estes identificadores à instância do redis.

- **sqlserver**: Base de dados principal com os dados normalizados.

4.5.2.2 Resultados

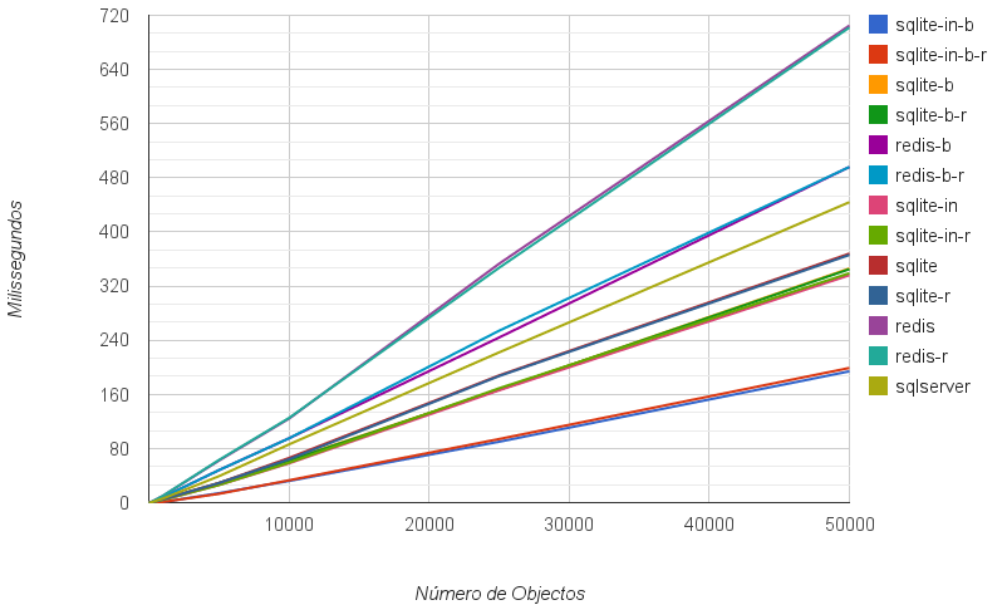

Figura 4.3: Gráfico da Performance do teste 2

Teste / Objectos	1	100	1000	5000	10000	25000	50000
sqlite-in-b	<1	<1	1	14	32	90	194
sqlite-in-r-b	<1	<1	1	13	33	94	199
sqlite-b	<1	<1	5	29	63	167	346
sqlite-r-b	<1	<1	5	28	62	167	345
redis-b	<1	<1	8	48	95	244	496
redis-r-b	<1	<1	8	48	95	254	496
sqlite-in	<1	<1	4	26	58	166	336
sqlite-in-r	<1	<1	4	26	59	169	339
sqlite	<1	<1	5	29	66	188	368
sqlite-r	<1	<1	5	29	64	187	366
redis	<1	<1	10	62	124	353	705
redis-r	<1	<1	10	63	125	347	702
sqlserver	<1	<1	6	39	86	222	444

Tabela 4.3: Média de tempos do teste 2 (em milissegundos)

Teste / Objectos	1	100	1000	5000	10000	25000	50000
sqlite-in-b	<1	<1	<1	7.64	3.34	5.36	10.57
sqlite-in-r-b	<1	<1	<1	2.97	2.56	5.8	10.49
sqlite-b	<1	<1	<1	3.01	2.58	6.41	11.41
sqlite-r-b	<1	<1	<1	2.73	2.96	5.97	9.33
redis-b	<1	<1	<1	3.31	7	8.83	12
redis-r-b	<1	<1	<1	3.74	6.36	11.72	9.21
sqlite-in	<1	<1	<1	2.76	2.59	6.69	10.99
sqlite-in-r	<1	<1	<1	2.36	2.88	7.71	9.73
sqlite	<1	<1	<1	2.14	3.55	6.86	9.79
sqlite-r	<1	<1	<1	2.31	3.19	8.11	7.76
redis	<1	<1	1.11	4.15	5.38	13.21	23.32
redis-r	<1	<1	<1	4.29	5.07	17.01	20.36
sqlserver	<1	<1	<1	3.46	6.39	9.07	14.82

Tabela 4.4: Desvio padrão do teste 2 (em milissegundos)

4.5.2.3 Análise dos Resultados

Com os resultados obtidos neste teste podemos verificar que em qualquer que seja o teste, a melhor performance que se consegue obter é quando os objectos são pedido em bytes, sem ser efectuada desserialização.

No caso do teste do pedido de 50000 objectos ao sqlite com os resultados em bytes, *sqlite-b (346 ms)*, e desserializados, *sqlite (368 ms)*, tem apenas uma diferença de aproximadamente 22 segundos (6 %).

O serviço desenvolvido, *sqlite (368 ms)*, quando comparado com o pedido dos objectos directamente da base de dados, *sqlserver (444 ms)*, têm um aumento de performance de aproximadamente 17%. Comparado também com o *redis (705 ms)*, teve uma peformance de mais 47%.

Como era de esperar os tempos de acesso utilizando o serviço de cache embutido na própria aplicação são mais baixos por não haver transferência em rede.

4.5.3 Teste 3 - Multi-threading

Este teste foi desenvolvido para comparar a performance entre as tecnologias: *REDIS* e *SQLite* utilizando o ambiente de testes 2 (4.2.2) em ambiente *multi-threading*, simulando vários acessos em simultâneo. Ao contrário dos outros testes, não foram pedidos várias quantidades de objectos à cache, mas sim repetidos os testes com diferentes números de *threads* em utilização.

Também foram medidos os tempos de acesso à base de dados, que se encontrava a funcionar na mesma máquina que os restantes serviços.

Cada tecnologia foi individualmente testada 100 vezes, e foram sempre pedidos 50000 objectos. Foram feitos testes para 1, 2 e 4 *threads*, sendo que em 2 *threads*, cada uma ficou com uma carga de 50 testes, e em 4 *threads*, cada uma ficou com 25, totalizando sempre os 100 testes.

Para efectuar este teste, foram instanciadas até 4 threads a executar o mesmo teste em simultâneo, forçando que existissem até 4 ligações ao serviço de cache activas no mesmo momento. Cada ligação activa no serviço tem uma thread associada, fazendo com que este teste usasse até um máximo de 8 threads (4 da aplicação do cliente e 4 do serviço). Por esta razão o número máximo de threads utilizadas neste teste foi 4 devido às característica da máquina (8 threads no máximo).

4.5.3.1 Descrição das tecnologias

As tecnologias testadas foram as mesmas que as utilizadas no teste 2, 4.5.2.1.

4.5.3.2 Resultados

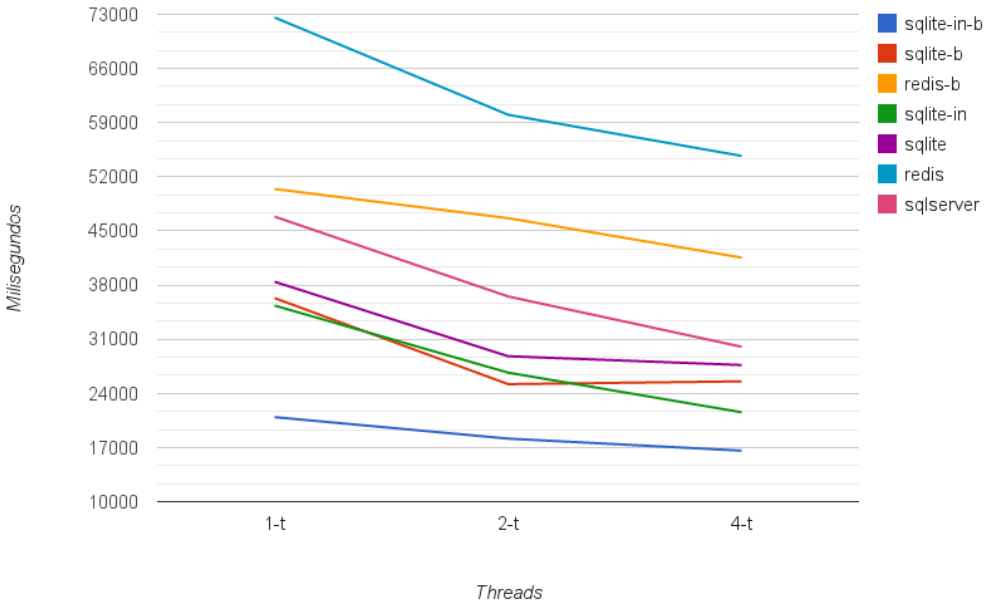

Figura 4.4: Gráfico do tempo total do teste 3

Teste / Threads	1	2	4
sqlite-in-b	20926	18150	16594
sqlite-b	36320	25188	25556
redis-b	50422	46649	41549
sqlite-in	35350	26681	21555
sqlite	38431	28798	27661
redis	72595	60017	54696
sqlserver	46852	36527	30020

Tabela 4.5: Tempo total do teste 3 (em milissegundos)

Teste / Threads	1	2	4
sqlite-in-b	208	361	654
sqlite-b	362	502	1019
redis-b	503	932	1638
sqlite-in	352	531	858
sqlite	383	572	1105
redis	725	1199	2116
sqlserver	467	728	1198

Tabela 4.6: Média de tempos do teste 3 (em milissegundos)

Teste / Threads	1	2	4
sqlite-in-b	19.4	30.01	74.55
sqlite-b	14.88	40.05	85.21
redis-b	56.61	536.36	1292.41
sqlite-in	12.36	37.53	78.29
sqlite	15.58	33.83	57.56
redis	34.71	76.88	400.63
sqlserver	20.13	57.36	90.65

Tabela 4.7: Desvio padrão do teste 3 (em milissegundos)

4.5.3.3 Análise dos Resultados

Como é possível analisar pelos tempos totais de cada teste, quanto mais threads utilizadas, mais rápido se consegue transferir e processar o mesmo número de objectos.

Por exemplo, com o serviço desenvolvido do *sqlite*, é possível efectuar o mesmo trabalho com 4 threads (aprox. 27.7 segundos), que com uma única thread (aprox. 38.4 segundos), com um ganho de cerca de 28%. Em contrapartida os tempos médios de acesso aumentaram de 383ms para 1105ms, um aumento de 187%, muito além dos 400% por estarem 4 *threads* a serem usadas em simultâneo.

4.5.4 Teste 4 - Optimização SQLite

Este teste é semelhante ao teste anterior (teste 3), mas foi desenvolvido para comparar a performance do *SQLite* em *multi-threading* utilizando a desqualificação de índices, no ambiente de testes 2 (4.2.2).

Também foram medidos os tempos de acesso à base de dados, que se encontrava a funcionar na mesma máquina que os restantes serviços.

Cada tecnologia foi individualmente testada 40 vezes, e foram sempre pedidos 25000 objectos. Foram feitos testes para 1, 2 e 4 *threads*, sendo que em 2 *threads*, cada uma ficou com uma carga de 20 testes, e em 4 *threads*, cada uma ficou com 10, totalizando sempre os 40 testes.

A cláusula utilizada nesta query foi semelhante às anteriores mas com ordenação descendente pela propriedade pesquisável *Name*: "*ID BETWEEN 1 AND 25000 ORDER BY NAME DESC*". A query optimizada com a desqualificação do índice *ID*, foi: "*+ID BETWEEN 1 AND 25000 ORDER BY NAME DESC*".

4.5.4.1 Descrição das tecnologias

Este teste foi composto por:

- **sqlite-in**: Neste teste foi utilizado o sqlite no contexto da aplicação de texto, sem um serviço de rede.

- **sqlite-in-opt**: Este teste é como o anterior mas utilizando a query optimizada.

- **sqlite**: Foi utilizado o serviço de rede com o sqlite, mas com a query normal.

- **sqlite-opt**: Este teste é como o anterior mas utilizando a query optimizada.

- **sqlserver**: Base de dados principal com os dados normalizados.

4.5.4.2 Resultados

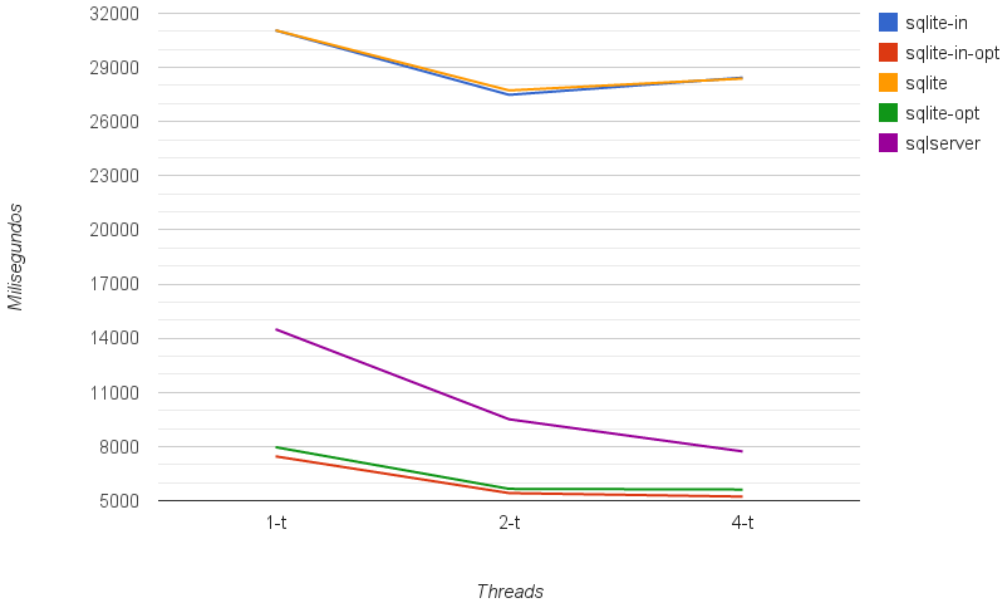

Figura 4.5: Gráfico do tempo total do teste 4

Teste / Threads	**1**	**2**	**4**
sqlite-in	31056	27485	28436
sqlite-in-opt	7453	5418	5225
sqlite	31060	27729	28387
sqlite-opt	7962	5653	5616
sqlserver	14497	9508	7727

Tabela 4.8: Tempo total do teste 4 (em milissegundos)

Teste / Threads	1	2	4
sqlite-in	775	1373	2828
sqlite-in-opt	185	269	513
sqlite	775	1384	2830
sqlite-opt	198	281	556
sqlserver	361	474	746

Tabela 4.9: Média de tempos do teste 4 (em milissegundos)

Teste / Threads	1	2	4
sqlite-in	24.58	209.04	478.33
sqlite-in-opt	6.57	24.46	61.24
sqlite	23.78	128.45	479.7
sqlite-opt	11.69	20.75	59.61
sqlserver	11.79	30.7	118.09

Tabela 4.10: Desvio padrão do teste 4 (em milissegundos)

4.5.4.3 Análise dos Resultados

Com este teste é possível analisar o ganho de performance do tempo total de cada teste, utilizando a desqualificação de índices numa query ao *SQLite*. No caso do serviço de cache em rede (testes *sqlite* e *sqlite-opt*), o ganho foi de 74% para um thread, 79% para duas threads e de 80% para 4 threads.

Comparando com a pesquisa directa ao *SQL Server* os ganhos também foram significativos: 45% para uma thread, 41% para 2 threads e 27% para 4 threads.

Como no teste anterior, os tempos médios de acesso aumentaram, mas nunca chegaram a 200% ou 400% ao utilizar 2 e 4 threads em simultâneo.

Capítulo 5

Conclusões

Com a elaboração deste projecto, foi possível adquirir um vasto conhecimento sobre várias tecnologias, desde bases de dados relacionas a não relacionais, dentro destas últimas, bases de dados NoSQL *in-memory* e híbridas. Com a investigação efectuada foi possível conhecer as suas características bem com as suas limitações, e com base nos testes realizados, comparar as suas performances. No final, o *SQLite* foi selecionada por entre as tecnologias candidatas: *REDIS*, *SQLite*, *MySQL Cluster* e *Couchbase*.

Foi produzido um serviço de cache *in-memory* que permite pesquisas eficientes sobre os seus seus conteúdos. Este serviço utiliza um protocolo próprio escrito utilizando sockets *TCP/IP* para a comunicação e foi desenvolvido desde o início com o objectivo de ser modular, significando que poderia ser implementado um novo serviço de cache com a mesma interface, e que pudesse a qualquer momento ser trocado de forma completamente transparente para o utilizador. O próprio conector do lado do cliente também implementa esta interface, o que permite a utilização da cache no próprio processo da aplicação, sem um serviço de rede.

Com este projecto também foram adquiridos vastos conhecimento sobre C# e a biblioteca *.NET 4.5*, bem como a influência do tamanho dos pacotes na velocidade de transferência através de sockets *TCP/IP*.

Para trabalhos futuros, seria um bom começo começar por implementar um método de pesquisa independente da linguagem *SQL*, por exemplo, através do *LINQ* do C# , ou mesmo através de uma classe genérica que iria gerar a pesquisa conforme a tecnologia utilizada.

Poderia ser também desenvolvido um módulo que com base num conjunto de regras pré-determinadas poderia carregar automaticamente para a cache os conteúdos antes da sua utilização, por exemplo, no caso de um consultório médico poderiam ser carregados os dados dos pacientes com base na agenda do médico. O serviço como está, actualmente necessita da intervenção do programador para implementar estas regras de carregamento dos dados.

Referências

[ALK+02] Mehmet Altinel, Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Bruce G. Lindsay, Honguk Woo e Larry Brown. Dbcache: Database caching for web application servers. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 612–612, New York, NY, USA, 2002. ACM. URL: http://doi.acm.org/10.1145/564691.564765, doi:10.1145/564691.564765.

[BD04] Paul Beynon-Davies. *Database Systems*. PALGRAVE MACMILLAN, Third edition, 2004.

[CSQ] CSQL Cache. *CSQL Main Memory Database Cache User Manual*. A versão 2.1 do manual pode ser obtido através do download do projecto no seguinte http://sourceforge.net/projects/csql/files/csql/csql3.3/, consultado em Julho de 2014.

[Eri] Jeff Erickson. In-memory acceleration for the real-time enterprise. Disponível em http://www.oracle.com/us/corporate/features/database-in-memory-option/index.html, consultado em Julho de 2014.

[kn:a] Couchbase. Disponível em http://www.couchbase.com/memcached, consultado em Julho de 2014.

[kn:b] Db-engines ranking of key-value stores. Disponível em http://db-engines.com/en/ranking/key-value+store, consultado em Julho de 2014.

[kn:c] Definição de base de dados híbrida. Disponível em http://www.webopedia.com/TERM/H/hybrid_database.html, consultado em Janeiro de 2015.

[kn:d] Forgetting data is a feature. Disponível em https://code.google.com/p/memcached/wiki/NewOverview#Forgetting_Data_is_a_Feature, consultado em Julho de 2014.

[kn:e] Memcachedb. Disponível em http://memcachedb.org/, consultado em Julho de 2014.

[kn:f] Mongodb. Disponível em http://www.mongodb.org/, consultado em Janeiro de 2015.

[kn:g] Optimizações de pesquisa em sqlite. Disponível em https://www.sqlite.org/optoverview.html#uplus, consultado em Janeiro de 2015.

[kn:h] Performance do protobuf-net. Disponível em https://code.google.com/p/protobuf-net/wiki/Performance, consultado em Dezembro de 2014.

REFERÊNCIAS

[kn:i] Protocol buffers. Disponível em https://developers.google.com/protocol-buffers/, consultado em Dezembro de 2014.

[kn:j] Unity 3 dependency injection. Disponível em https://msdn.microsoft.com/en-us/library/dn223671(v=pandp.30).aspx, consultado em Dezembro de 2014.

[kn:k] Using redis as an lru cache. Disponível em http://redis.io/topics/lru-cache, consultado em Julho de 2014.

[kn:l] Varnish administrator documentation. Disponível em https://www.varnish-cache.org/docs/4.0/, consultado em Julho de 2014.

[LGZ03] Per-Ake Larson, Jonathan Goldstein e Jingren Zhou. Transparent mid-tier database caching in sql server. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 661–661, New York, NY, USA, 2003. ACM. URL: http://doi.acm.org/10.1145/872757.872848, doi:10.1145/872757.872848.

[LKM+02] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay e Jeffrey F. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 600–611, New York, NY, USA, 2002. ACM. URL: http://doi.acm.org/10.1145/564691.564763, doi:10.1145/564691.564763.

[Ora] Oracle. *Oracle Database In-Memory*. Oracle Data Sheet, acessível em http://sourceforge.net/projects/csql/files/csql/csql3.3/, consultado em Julho de 2014.

[Rou08] Margaret Rouse. Definição de master/slave. Disponível em http://searchnetworking.techtarget.com/definition/master-slave, consultado em Julho de 2014, October 2008.

Anexo A

Protocolo de Comunicação

A troca de mensagens entre o serviço de cache e o conector do cliente é feita seguindo um protocolo definido com esta finalidade. Cada tipo de operação (ver 3.2.11) tem o fluxo de mensagens estruturada de forma diferente, sendo todas tecnicamente especificadas neste capítulo.

A finalidade de ter sido desenvolvido um protocolo próprio em vez de utilizar um outro protocolo de mais alto nível que o C# pudesse oferecer, foi devido à performance. Os protocolos de mais alto nível, embora que escritos de forma eficiente, são demasiado genéricos e complexos. Por exemplo, poderia ter sido utilizado o *WCF*, que é independente de plataforma, mas utiliza envelopes *SOAP* escritos em *XML* que necessitam de várias conversões nos dados tanto no lado do cliente como no serviço, completamente desnecessárias e ineficientes.

A.1 Estruturas de dados

Existem três tipos de dados que podem ser enviados pelo protocolo.

A.1.1 Inteiros e longos

Os inteiros são enviados como 4 bytes em *little-endian*, e os longos em 8 bytes *little-endian*.

A.1.2 Strings

As string necessitam de ser representadas em bytes antes de serem enviadas, segundo os seguintes passos:

1. A string é convertida para bytes e são calculados o número de bytes utilizados.
2. É enviado o número de bytes ocupados pela string.
3. São enviados os bytes da string.

Número de bytes utilizados	inteiro 4 bytes (little-endian)
String codificada em bytes	bytes da string

Tabela A.1: Envio de strings

A.1.3 Mensagens protobuf

Algumas operações necessitam de enviar mensagens estruturadas, que são serializadas com o protobuf. Estas mensagens serão referidas como *mensagem protobuf*, e são enviadas da seguinte forma:

1. A mensagem é serializada com protobuf e é calculado os número de bytes utilizados.
2. É enviado o número de bytes utilizados pela mensagem.
3. São enviados os bytes da mensagem.

Número de bytes utilizados	inteiro 4 bytes (little-endian)
Mensagem protobuf serializada	bytes da string

Tabela A.2: Envio de Mensagens protobuf

A.1.3.1 PrepareDocument

A mensagem *PrepareDocument* é enviada para o serviço de cache no momento da inicialização do conector do cliente, contendo toda a informação necessária para descrever as entidades que o conector poderá colocar e/ou pesquisar da cache. O diagrama desta mensagem pode ser consultado na figura 3.2.

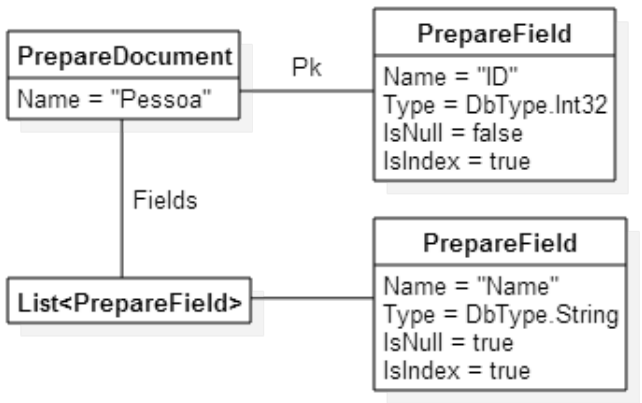

Figura A.1: Exemplo de um PrepareDocument

A.1.3.2 InsertDocument

A mensagem *InsertDocument* é enviada para o serviço de cache quando é pretendido inserir dados no serviço de cache. Cada documento descreve apenas um objecto, mas podem ser enviados uma lista destes objectos em um único pedido. O diagrama desta mensagem pode ser consultado na figura 3.2.

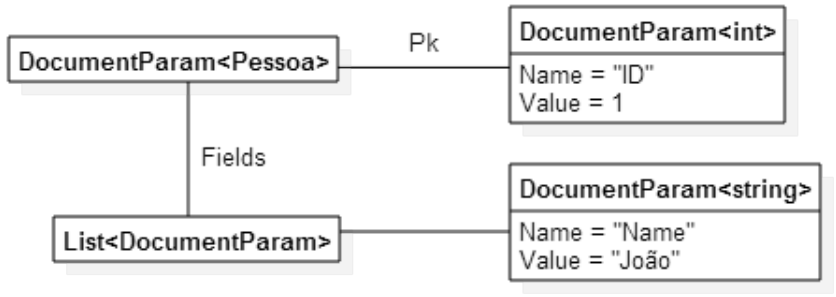

Figura A.2: Exemplo de um InsertDocument

A mensagem *InsertDocument* tem um campo que contem a instância do objecto correspondente, mas não é serializado junto com a mensagem, isto porque para o serviço desserializar uma mensagem *InsertDocument<Pessoa>* teria de conhecer o conceito de *Pessoa*.

Quando existe um campo que seja pretendido ir a nulo, pode ser enviado como caso especial, uma mensagem do tipo *DocumentNull* no lugar do *DocumentParam*.

A.1.4 Códigos de estado

Existem vários códigos de estados, que podem ser enviados a informar o resultado da operação pedida, indicando ao cliente como deverá proceder posteriormente.

A.1.4.1 Prepare

- **OK** (0): Pedido de criação efectuado com sucesso.
- **ERROR** (1): Ocorreu um erro no pedido de criação. Para mais detalhes consultar os logs do serviço.

Este código ocupa um byte.

A.1.4.2 Query

- **OK** (0): Query efectuada com sucesso.
- **NOT_PREPARED** (1): A entidade em questão não existe, e tem de ser preparada antes de ser utilizada.

- **BAD_QUERY** (2): A query SQL efectuada não validou correctamente.
- **ERROR** (3): Ocorreu um indefinido. Para mais detalhes consultar os logs do serviço.

Este código ocupa um byte.

A.1.4.3 Insert

- **ZERO_SUCCESS** (0): Zero objectos inseridos com sucesso.
- **NOT_PREPARED** (-1): A entidade em questão não existe, e tem de ser preparada antes de ser utilizada.
- **ERROR** (-2): Ocorreu um indefinido. Para mais detalhes consultar os logs do serviço.

Este código é enviado em formato de inteiro, ocupando 4 bytes.

A.2 Operações

De seguida é descrito o fluxo tomado pelo protocolo, nas diferentes operações.

A.2.1 Prepare

Byte de comando: 0x1

1. O cliente envia uma *mensagem protobuf* com um *PrepareDocument* (A.1.3.1).
2. O serviço responde com um código de estado (A.1.4.1), indicando o resultado da operação.

A.2.2 Query

Byte de comando: 0x2

1. O cliente envia uma string (A.1.2) com o nome da entidade.
2. O cliente envia uma string (A.1.2) com a cláusula SQL a ser utilizada para pesquisar os dados.
3. O serviço responde com um código de estado (A.1.4.2), indicando o resultado da operação, e em caso de erro, o fluxo termina.
4. O serviço envia um inteiro com o número de bytes do próximo objecto, se for negativo, significa que não existem mais objectos, e o fluxo termina.
5. O serviço envia os bytes do objecto actual.
6. Repete os últimos dois passos.

A.2.3 Insert

Byte de comando: 0x3

1. O cliente envia um inteiro, referindo quantos objectos serão enviadas.
2. O cliente envia um longo, com a representação da data de expiração do objecto (*DateTime.Ticks* do *.NET*). Este valor pode vir a zeros, significando que não existe expiração.
3. O cliente envia uma string (A.1.2) com o nome da entidade.
4. O cliente envia a *mensagens protobuf* com o *InsertDocument* (A.1.3.2) do primeiro objecto a ser inserido.
5. O cliente envia um inteiro com o número de bytes do objecto serializado a ser inserido
6. O cliente envia os bytes do objecto.
7. Os últimos 3 passos são repetidos para todos os objectos a serem enviados.
8. O serviço responde com um inteiro: se for negativo ou igual a zero, corresponde a uma mensagem de erro (A.1.4.3), se for positivo, corresponde ao número de objectos inseridos com sucesso.

A.2.4 Delete

Byte de comando: 0x4

1. O cliente envia uma string (A.1.2) com o nome da entidade.
2. O cliente envia uma string (A.1.2) com a cláusula SQL a ser utilizada para apagar os dados.
3. O serviço responde com um inteiro: se for negativo, significa que ocorreu um erro e devem ser consultados os logs do serviço para mais detalhes, se for zero ou positivo, indica o número de objectos apagados com sucesso.

A.2.5 Flush

Byte de comando: 0x5

1. O serviço responde com um inteiro: se for negativo, significa que ocorreu um erro e devem ser consultados os logs do serviço para mais detalhes, se for zero ou positivo, em caso contrário, a operação foi concluída com sucesso.

Anexo B

Exemplo Prático de Utilização da Framework

Este anexo descreve um exemplo prático de utilização do serviço de cache desenvolvido, em C# , com uma entidade *Pessoa*.

Para informações de como instalar o serviço de cache *glintt-cache-service* como um *Windows Service*, consultar: http://msdn.microsoft.com/en-us/library/zt39148a(v=vs.110).aspx#BK_Install

```
1  // Indica que o protobuf-net poder ser utilizado para serializar esta classe,
2  // e que deve serializar todos os campos públicos;
3  [ProtoContract(ImplicitFields = ImplicitFields.AllPublic)]
4  // Indica que esta entidade pode ser processada pelo Cache Helper
5  // Pode ser defenido outro nome para esta entidade, p.e.: [CacheEntity("TESTE_Pessoa")]
6  [CacheEntity]
7  public class Pessoa
8  {
9      // Indica que a Chave primária da entidade é o campo ID
10     // Opcionalmente pode ser defenido outro nome, p.e.: [PkCacheParameter("PK_id")]
11     [PkCacheParameter]
12     public int ID { get; set; }
13
14     // Indica que o campo Name é pesquisável
15     [CacheParameter(Index = true, Null = true)]
16     public String Name { get; set; }
17
18     // Este atributo será serializado mas não será pesquisável em cache
19     public bool IsMale { get; set; }
20
21     // Este atributo será serializado mas não será pesquisável em cache
22     public byte[] BinData { get; set; }
23
24     // Este atributo será serializado mas não será pesquisável em cache
25     public long SomeLong { get; set; }
```

```
26
27      // Este atributo será serializado mas não será pesquisável em cache
28      public double SomeDouble { get; set; }
29  }
```

Listing B.1: Declaração da entidade Pessoa

As seguintes classes podem ser consultadas na documentação do projecto para mais informações da sua utilização:

- glintt_cache_lib.Cache.CacheEntityAttribute
- glintt_cache_lib.Cache.PkCacheParameterAttribute
- glintt_cache_lib.Cache.CacheParameterAttribute

Para exemplos de utilização do protobuf-net os seguintes links fornecem a documentação e exemplos necessários:

- https://code.google.com/p/protobuf-net/wiki/GettingStarted
- https://code.google.com/p/protobuf-net/wiki/Attributes
- http://www.codeproject.com/Articles/642677/Protobuf-net-the-unofficial-manual

Caso seja necessário incorporar instâncias de classes dependentes na serialização do protobuf-net, o seguinte link pode ser consultado: https://code.google.com/p/protobuf-net/wiki/GettingStarted#Inheritance

Pequeno exemplo extraído do link acima:

```
1  [ProtoContract]
2  [ProtoInclude(7, typeof(SomeDerivedType))]
3  class SomeBaseType {...}
4  [ProtoContract]
```

Listing B.2: Exemplo de classes dependentes com protobuf-net

Para facilitar a utilização do serviço de cache, a classe *QueryCacheHelper* automatiza toda a lógica necessária, pelo que deve ser utilizada sempre que possível.

```
1      // Instância uma connection pool com um máximo de 5 ligações, sendo apenas duas ↩
           inicializadas préviamente
2      // O delegate que instância cada ligação retorna um RemoteServiceCache que se liga ↩
           a 'localhost' na porta '8889' e com 64k de buffer interno
3      QueryConnectionPool pool = new QueryConnectionPool(() => { return new ↩
           RemoteServiceCache("localhost", 8889, 64 * 1024); }, 5, 2);
4
```

```
5       // Indica que só podem haver 3 ligações em Idle num dado momento
6       pool.MaxIdleConnections = 3;
7
8       // QueryCacheHelper aceita qualquer classe que implemente a inteface IQueryCache, n↩
            ão sendo obrigatório utilzar uma QueryConnectionPool
9       QueryCacheHelper cacheHelper = new QueryCacheHelper(pool);
10
11      // Em caso de outra ligação (outro cliente) efectuar flush à base de dados a cache ↩
            retornará uma excepção com esta indicação
12      // Por defeito a cache irá tentar voltar a efectuar o prepare
13      cacheHelper.ReprepareOnError = true;
14
15      // É possível indicar ao helper que torne a enviar os dados caso seja necessário ↩
            voltar a fazer prepare da entidade e ReprepareOnError seja true
16      cacheHelper.ResendOnReprepare = true;
17      Console.WriteLine("Preparando a classe Pessoa");
18
19      // Prepara a Entidade do tipo Pessoa em cache
20      cacheHelper.Prepare<Pessoa>();
21      var pex1 = new Pessoa();
22      var pex2 = new Pessoa();
23      pex1.ID = 1;
24      pex1.Name = "Pessoa 1";
25      pex1.IsMale = true;
26      pex1.BinData = new byte[1024];
27      pex1.SomeLong = 123L;
28      pex1.SomeDouble = 0.0123D;
29      pex2.ID = 12;
30      pex2.Name = "Pessoa 12";
31      pex2.IsMale = false;
32      pex2.BinData = new byte[2048];
33      pex2.SomeLong = 321123L;
34      pex2.SomeDouble = 0.0321123D;
35      var list = new List<Pessoa>();
36      list.Add(pex1);
37      list.Add(pex2);
38
39      // adiciona duas entidades do tipo Pessoa à cache com expiração de uma hora a ↩
            partir deste momento
40      int set = cacheHelper.Set<Pessoa>(list, DateTime.Now.AddHours(1));
41      Console.WriteLine("Resultados inseridos: {0}", set); list.Clear();
42
43      // seleciona todas as entidades do tipo pessoa em que o nome comecem por 'Pessoa'
44      cacheHelper.Get<Pessoa>("Name LIKE 'Pessoa%'", list);
45      Console.WriteLine("Resultados obtidos com Get<Pessoa>: {0}", list.Count); list.↩
            Clear();
46
47      // em alternativa, a leitura pode ser feita através de um IQueryCacheReader<Pessoa↩
            >,
48      // sendo possível processar os dados à medida que são recebidos
49      using (var reader = cacheHelper.GetReader<Pessoa>("Name LIKE 'Pessoa%'"))
50      {
51          while (reader.Read())
52          {
53              list.Add(reader.Next());
54          }
55      }
```

```
56
57      // Um reader necessita SEMPRE de ser usado dentro de um bloco using(...) { }
58      // ou após a sua utilização forçar a chamada ao método reader.Dispose() dentro de ↩
            um bloco finally { ... }
59      Console.WriteLine("Resultados obtidos com GetReader<Pessoa>: {0}", list.Count); ↩
            list.Clear();
60
61      // Apaga a pessoa com ID = 12 da cache
62      int deleted = cacheHelper.Delete<Pessoa>("ID = 12");
63      Console.WriteLine("Resultados apagados: {0}", deleted);
64
65      // reinicializa toda a cache, sendo tudo apagado, e necessário chamar novamente os ↩
            respectivos Prepare<...>()
66      cacheHelper.Flush();
67      Console.WriteLine("Toda a cache foi reinicializada");
```

Listing B.3: Exemplo de utilização do API da framework

Para uma informação mais detalhada da API, devem ser consultadas as seguintes classes na documentação do projecto:

- glintt_cache_lib.Cache.IQueryCache
- glintt_cache_lib.Cache.IQueryCacheReader<T>
- glintt_cache_lib.Cache.QueryConnectionPool
- glintt_cache_lib.Cache.IQueryCacheHelper

O conector da cache pode ser utilizado directamente (sem o helper), sendo possível adquirir uma Stream, ou o byte array (byte[]) em vez da instância da classe, tornando-se útil para por exemplo, guardar e retornar ficheiros em cache.