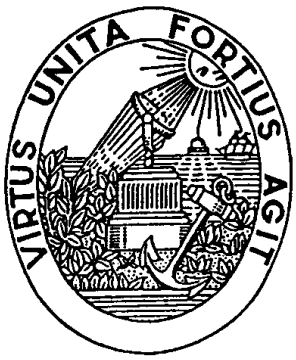
Optimização das comunicações entre Aplicações e Bases de Dados

Nuno Alberto Paulino da Fonseca

Outubro 2000

Nuno Alberto Paulino da Fonseca

Optimização das comunicações entre Aplicações e Bases de Dados



Tese submetida à Faculdade de Engenharia da Universidade do Porto para obtenção do grau de Mestre em Inteligência Artificial e Computação

Tese realizada sob orientação do Prof. Doutor José Paulo Leal

Em memória ao meu avô paterno.

Agradecimentos

Quero começar por agradecer de uma forma muito especial ao Prof. Doutor Fernando Silva por todo o apoio que sempre me concedeu. Gostaria ainda de agradecer à Prof. doutora Inês Dutra pelos esclarecimementos e conversas que tivemos sobre *caches*.

Não quero deixar de reconhecer o apoio e compreensão demonstrados pela minha família e amigos que me aturaram durante o período de realização da tese. Não poderia deixar de realçar todo o apoio e ajuda dados pela minha namorada na realização desta tese.

Para terminar, gostaria ainda de agradecer a todos os que contribuiram, de alguma forma, para a realização deste trabalho.

Resumo

Na última década assistiu-se a um quase meteórico crescimento dos sistemas de informação e da sua interligação em rede, nomeadamente através da Internet. A estrutura destes sistemas, cuja fonte de dados é normalmente um sistema de base de dados, assenta sobretudo no modelo cliente/servidor. Como consequência deste desenvolvimento, as comunicações, naturalmente existentes entre as fontes dados e as aplicações de interface, bem como o volume de dados transferidos são cada vez maiores e resultam num aumento significativo da latência no acesso aos dados.

Esta tese propõe um modelo, e respectiva implementação, que visa reduzir a latência no acesso aos dados por parte de aplicações. O modelo baseia-se no uso de caches para guardar os dados mais frequentemente usados e que resultam de consultas efectuadas à fonte de dados. A coerência dos dados em cache é garantida por recurso aos mecanismos de invalidação ou de actualização imediata. A implementação deste modelo resultou no sistema Polites.

O sistema Polites foi desenvolvido de forma modular para que fosse possível, testar e validar não só o modelo proposto, mas também estratégias alternativas que venham a ser pensadas futuramente. O sistema define uma interface de comunicação com as aplicações de forma a uniformizar e tornar transparente o acesso a diferentes fontes de dados por parte das aplicações.

No âmbito deste trabalho foi ainda desenvolvido uma API de acesso a base de dados, designada por DBase, que mapeia os resultados das consultas a bases de dados em estruturas de dados (na linguagem TCL) e as mantém coerentes.

O modelo proposto foi testado através da realização de experiências com o Polites. Os resultados obtidos indicam que, em certas condições, é possível reduzir a latência e o tráfego de dados para menos de metade. No entanto, em alguns casos a utilização

do sistema tem efeitos contrários aos desejados, aumentando a latência e o número de bytes transferidos.

Abstract

Nowadays, databases play a key role on the dissemination and sharing of information among users throughout client-applications residing elsewhere on the network. Recently, given the success of the Internet, the number of data-centric applications and the data/traffic on the network has increased enormously resulting in a high latency in accessing data sources.

This work proposes a model that tries to minimise the data communications and latency between applications and data sources, such as databases. The model is based on the idea of employing a cache to keep results of previous queries to data sources in order to speedup future data accesses. The cache is kept within the application itself or/and within a proxy. The coherency of the data stored in the cache is maintained by the use of update or invalidate protocols. This model has been implemented as the system Polites.

The Polites is a middleware that interfaces the user applications with data sources. It has been structured in such a way that allows applications to access several types of data sources in a transparently.

Another component of this work consisted in the development of an API, named DBase, that maps the results of database queries in data structures in the TCL language. It is also responsible for keeping the content of these structures synchronized with the corresponding data whithin the mapped databases.

Polites was throughly tested and the results obtained indicate that, in certain conditions, it is able to reduce the latency and network traffic (due to the communications between applications and databases) to less than half of the initial value. However, the Polites was not found very helpful when the frequency of changes in the data being cached is too high.

Résumé

Sur de nos jours, les bases de données jouent un rôle principal la diffusion et partage d'information parmi des utilisateurs dans toutes des client-applications résidant ailleurs sur le réseau. Récemment, donné le succès de l'Internet, le nombre d'applications donnée-centrales et le data/traffic sur le réseau a augmenté énormément ayant pour résultat un temps d'attente élevé dans des points d'émission de données d'accès.

Ce travail propose un modèle qui essaye entre de réduire au minimum les communications de données et temps d'attente les applications et les points d'émission de données, tels que des bases de données. Le modèle est basé sur l'idée d'utiliser une *cache* pour garder des résultats des requêtes précédentes accès de données de *speedup* de points d'émission de données à de futurs. Le *cache* est gardée dans l'application elle-même ou/et dans une procuration. La concordance des données enregistrées dans le *cache* est mise à jour par l'utilisation de la mise à jour ou infirme des protocoles. Ce modèle a été mis en application comme système Polites.

Le Polites est un logiciel personnalisé qui connecte les applications d'utilisateur avec des points d'émission de données. Il a été structuré de telle manière que permette à des applications d'accéder à plusieurs types de points d'émission de données dans a d'une manière transparente. Un autre composant de ce travail a consisté en développement d'une API, nommé DBase, qui trace les résultats des requêtes de base de données en structures de données dans le langage de TCL. Il est également responsable de garder la teneur de ces structures synchronisées avec le correspondant de base de données.

Polites a été testé et les résultats obtenus indiquent que, en certaines conditions, il peut ramener le traffic (à cause de temps d'attente et de réseau aux transmissions entre les applications et les bases de données) à moins que la moitié de la valeur initiale. Cependant, le Polites n'a pas été trouvé très utile quand la fréquence des changements

des données étant cachés est trop haute.

Conteúdo

Índice de Tabelas **14**

Índice de Figuras **16**

Abreviaturas **17**

1 Introdução **18**

1.1 Motivação 18

1.2 Objectivos 19

1.3 Organização da tese 20

2 Aplicações e Bases de Dados **22**

2.1 Introdução 23

2.1.1 SQL 23

2.1.2 As linguagens de programação e o SQL 24

2.1.3 Modelo Cliente/Servidor 26

2.1.4 Acesso a bases de dados 27

2.2 *Middleware* de acesso a bases de dados 29

2.2.1 X/Open CLI 30

2.2.2 ODBC 32

2.2.3 JDBC 34
2.2.4 SQLJ 36
2.2.5 OLE DB e ADO 38
2.3 Middleware de acesso a bases de dados 40
2.3.1 Java Blend 40
2.3.2 dbKona 41

3 *Sistemas de Cache* 43
3.1 Introdução 44
3.1.1 Implementação 47
3.1.2 Falhas na *cache* 47
3.1.3 Coerência da *cache* 48
3.1.4 Políticas de Troca 51
3.1.5 Políticas de Escrita 53
3.1.6 Localização da *cache* 54
3.1.7 Métricas de avaliação 55
3.2 Optimização das comunicações com BDs 57
3.2.1 *Cache* de resultados de consultas a bases de dados 57
3.2.2 *Cache* na *WWW*: semelhanças e diferenças 59
3.2.3 Outras formas de optimização das comunicações 59

4 Modelo Proposto 61
4.1 Introdução 62
4.2 Modelo Básico 63
4.3 *Gateways* 64

4.4 Pontes (*Bridges*) 66
4.5 *Proxies* 66

5 Polites **68**

5.1 Introdução 69
5.2 Arquitectura 69
5.3 Núcleo 73
5.3.1 Núcleo Cliente 73
5.3.1.1 *Cache* 74
5.3.1.2 Configurações 75
5.3.1.3 Funcionamento 76
5.3.2 Núcleo servidor 76
5.3.2.1 Tabela de coerência de dados (DCT) 77
5.3.2.2 Configurações 78
5.3.2.3 Funcionamento 78
5.4 Módulos de dados 79
5.4.1 SQL 80
5.4.1.1 Normalização de Comandos SQL 81
5.5 API 84
5.5.1 Mysql 84
5.5.2 Oracle 84
5.5.3 DBase 85
5.6 Comunicação 87
5.7 Implementação 88

6 Experiências e Resultados **89**

6.1 Experiências 90

6.2 Análise de resultados 91

6.2.1 *Speedup* 91

6.2.2 *Ganho obtido na transferência de bytes* 94

6.2.3 *Considerações Finais* 94

7 Conclusões **97**

Referências **100**

A Resultados **104**

B Parâmetros da Simulação **111**

Lista de Tabelas

5.1 API Mysql 85

5.2 API DBase 86

B.1 Operações SQL 112

B.2 Estatísticas das operações SQL 113

B.3 Parâmetros da aplicação C1 113

B.4 Parâmetros da aplicação C2 113

B.5 Parâmetros da aplicação C3 114

B.6 Velocidade da rede 114

Lista de Figuras

2.1 CLI 26

2.2 Acesso a bases de dados 28

2.3 Evolução do modelo cliente/servidor 29

2.4 Arquitectura ODBC 33

2.5 Arquitectura JDBC 35

2.6 Conversão de SQLJ para Java 37

2.7 OLE DB e ADO 39

3.1 Acesso a dados sem utilizar uma *cache* 45

3.2 Acesso a dados através de uma *cache* 46

3.3 Acesso a dados através de várias *caches* 46

4.1 Modelo tradicional 62

4.2 Modelo básico 63

4.3 Acesso à fonte de dados através de uma *gateway* 65

4.4 Acesso à fonte de dados através de uma "ponte" 66

4.5 Acesso à fonte de dados através de um *proxy* 67

5.1 Arquitectura do Polites 70

5.2 Representação de um comando SQL 82

6.1 Speedup obtido com o protocolo de invalidação 92

6.2 Speedup obtido com o protocolo de actualização 93

6.3 Ganho obtido na transf. de bytes com a utilização do protocolo de invalidação 95

6.4 Ganho obtido na transf. de bytes com a utilização do protocolo de actualização 96

A.1 Resultados obtidos pela aplicação C1 105

A.2 Resultados obtidos pela aplicação C1 106

A.3 Resultados obtidos pela aplicação C2 107

A.4 Resultados obtidos pela aplicação C2 108

A.5 Resultados obtidos pela aplicação C3 109

A.6 Resultados obtidos pela aplicação C3 110

Abreviaturas

ADO ***A**ctiveX **D**ata **O**bject*

ANSI ***A**merican **N**ational **S**tandard **I**nstitute*

API ***A**pplication **P**rogram **I**nterface*

CCE ***C**lient **C**ursor **E**ngine*

CLI ***C**all **L**evel **I**nterface*

COM ***C**omponent **O**bject **M**odel*

DCT ***D**ata **C**oerency **T**able*

DML ***D**ata **M**anipulation **L**anguage*

ESQL ***E**mbebed **SQL***

HTTP ***H**ypertext **T**ransfer **P**rotocol*

ISO ***I**nternational **S**tandards **O**rganization*

JDBC ***J**ava **D**ata**B**ase **C**onnectivity*

OLE ***O**bject **L**inking and **E**mbedding*

RDS ***R**emote **D**ata **S**ervice*

SGBD ***S**istema de **G**estão de **B**ases **D**ados*

SQL ***S**tructured **Q**uery **L**anguage*

UDA ***U**niversal **D**ata **A**ccess*

Capítulo 1

Introdução

1.1 Motivação

Hoje em dia, na chamada era das tecnologias de informação, as bases de dados desempenham um papel importante na manutenção de informação. A centralização da informação em bases de dados resultou na utilização crescente do modelo cliente/servidor como forma de estruturar as aplicações baseadas em, ou que utilizem, bases de dados.

A centralização e a utilização do modelo cliente/servidor originou um crescimento no volume de dados transferidos nas redes, como resultado das comunicações entre as aplicações e as bases de dados. A comunicação com bases de dados é muitas vezes a grande fonte de estrangulamento na performance das aplicações, principalmente se a operação envolver a transferência de muitos dados. Este estrangulamento depende principalmente de dois factores: o tráfego na rede; e a velocidade de resposta do Sistema de Gestão de Base de Dados(SGBD).

As aplicações executam um grande número de operações de consulta ("select") sobre as bases de dados. A execução de consultas satisfaz, num grande número de aplicações, uma propriedade importante - localidade temporal. Isto significa que as consultas realizadas durante a execução de uma aplicação tendem a ser repetidas individualmente (por cada aplicação) ou de forma distribuída (por um conjunto de aplicações) à base de dados. No entanto, as aplicações não exploram esta propriedade. Por vezes, as aplicações repetem a execução de consultas porque não sabem se os resultados obtidos previamente estão actualizados, uma vez que não são informadas das alterações

ocorridas na base de dados entre as consultas.

A localidade temporal é uma propriedade explorada pelas *caches*. Neste sentido, a *cache* dos resultados das consultas às bases de dados, pelas aplicações, poderá ser uma solução atractiva para optimizar as comunicações. Esta solução potencia a redução: da latência, uma vez que o resultado de uma consulta poderá estar disponível imediatamente a partir da *cache*; da carga de trabalho no SGBD, uma vez que terá de responder a um número menor de pedidos; e do tráfego na rede resultante dos pedidos e respostas.

A utilização da *cache* poderá ser particularmente atractiva quando os resultados são originados de consultas complexas, pois a latência nestes casos tenderá a ser maior. No entanto, a utilização de *caches* requer uma solução para o problema de garantir a manutenção da coerência da *cache*. As aplicações ao utilizarem os resultados de uma consulta esperam que os resultados estejam coerentes com os dados na base de dados.

1.2 Objectivos

Um dos objectivos desta tese consistiu na optimização das comunicações entre aplicações e bases de dados. Para atingir este objectivo explorou-se a ideia de manter em *cache* os resultados das consultas a bases de dados. Tendo como base esta ideia, foi proposto um modelo que permite às aplicações reduzir a latência no acesso aos dados, através da utilização de uma *cache*, e que assegura uma consistência forte ou fraca da mesma, consoante o mecanismo de manutenção da coerência utilizado.

Para validar o modelo foi implementado um sistema baptizado com o nome de Polites. O Polites foi desenvolvido de forma a ser modular, transparente e a garantir a coerência dos dados mantidos em *cache*. O sistema foi testado com o intuito de avaliar os efeitos da sua utilização: nas aplicações, ao nível da diminuição da latência; e na rede, ao nível do número de bytes transferidos.

Existem alguns sistemas [37, 4, 7, 1], descritos no capítulo 3, que tentam optimizar as comunicações entre aplicações e bases de dados através da manutenção em *cache* dos resultados das consultas realizadas. No entanto, estes sistemas asseguram apenas uma consistência fraca. O modelo proposto e o sistema implementado asseguram uma

consistência forte através da utilização do protocolo de invalidação ou do protocolo de actualização.

Outro objectivo desta tese consistiu na implementação de uma API de acesso a base de dados, baptizada com o nome de DBase, que: reflicta automaticamente os resultados das consultas a base de dados em estruturas de dados; permita a modificação dos dados na estrutura de dados criada (sem a utilização de SQL) e propague à base de dados as alterações realizadas na estrutura; mantenha as estruturas de dados coerentes com a base de dados.

Foram encontrados dois produtos comerciais, o DBKona [6] e o JavaBlend [2], que reflectem os resultados das consultas a base de dados em estruturas de dados. Estes produtos não mantêm a coerência da informação nas estruturas de dados, resultando numa diminuição da utilidade dos produtos. O DBase diferencia-se destes produtos por assegurar a coerência da informação mantida nas estruturas de dados. O DBase assegura a coerência da informação através da utilização do Polites para realizar as operações sobre as bases de dados, usufruindo ainda dos mecanismos de manutenção de coerência disponíveis.

1.3 Organização da tese

Esta tese está organizada em 7 capítulos.

O capítulo 2 descreve a forma das aplicações acederem a bases de dados e apresenta o trabalho relacionado com a reflexão dos resultados das consultas a bases de dados.

O terceiro capítulo oferece uma introdução a *caches* e apresenta os trabalhos relacionados com a utilização de *caches*.

O capítulo 4 descreve o modelo proposto para a optimização das comunicações entre aplicações e bases de dados.

No capítulo 5 é descrito o sistema Polites, que se baseia no modelo proposto no capítulo 4.

No capítulo 6 são apresentados os resultados obtidos com a utilização do Polites .

Por último, são apresentadas as conclusões, onde é feito um breve resumo do trabalho realizado, dos resultados obtidos e do trabalho futuro.

Capítulo 2

Aplicações e Bases de Dados

As bases de dados são, cada vez mais, o meio utilizado para manter e gerir informação. A consulta ou manipulação da informação é habitualmente realizada através da utilização de aplicações de interface (software). Estas aplicações oferecem, aos utilizadores, uma interface simples para aceder e manipular os dados mantidos nas bases de dados.

Este capítulo começa por dar uma breve introdução à forma como as aplicações interagem com as bases de dados. De seguida, são apresentadas as principais formas de *middleware* de acesso a bases de dados. E para finalizar o capítulo são apresentadas as soluções encontradas que simplificam o desenvolvimento de aplicações que acedam a bases de dados.

2.1 Introdução

Em 1997, o mercado de software de bases de dados valia 14 biliões de dólares (2.8 biliões de contos), estando à frente de todos os outros tipos de software. Os sistemas de bases de dados e serviços, que incluem o hardware e serviços vendidos em associação com o software de base de dados, era de 43 biliões de dólares[1].

A centralização da informação em bases de dados resultou na utilização crescente do modelo cliente/servidor como forma de estruturar as aplicações baseadas em, ou que utilizem, bases de dados. A centralização e a utilização do modelo cliente/servidor originou um crescimento na transferência de dados nas redes, como resultado das comunicações entre as aplicações e as bases de dados.

Os Sistemas de Gestão de Bases de Dados (SGBD), e em particular os Sistemas de Gestão de Bases de Dados Relacionais (SGBDR), são a principal forma utilizadada na gestão de dados.

2.1.1 SQL

A manipulação e consulta dos dados nos SGBD é feita utilizando a linguagem SQL (*Structured Query Language*). Apesar de existirem muitas implementações de SQL desde a sua primeira implementação em 1979, só em 1986 é que foi publicado uma especificação standard pela ANSI e ISO. Em 1989 a especificação foi revista para introduzir integridade referencial, e ficou conhecida como SQL-89 ou ANSI SQL. Esta primeira versão do SQL foi uma tentativa de uniformizar as diversas implementações de SQL existentes na altura nos vários SGBD. O standard SQL foi alterado em 1992 dando origem ao SQL-92 (também designado por SQL2). Em 1999 foi publicada a nova especificação da linguagem SQL - o SQL3. As principais novidades prendem-se com a possibilidade de definição de tipos de dados (UDT - User data type); *triggers*; *stored procedures*; SQL/OLB (Object Language Bindings) que define a forma de ligação do SQL a linguagens orientadas a objectos, dando origem ao SQLJ.

A linguagem SQL baseia-se no modelo relacional proposto por Codd em 1970 [21]. Este modelo separa a representação lógica dos dados da sua representação física. Os

[1] Dados obtidos do Database Demographics Report, de 1998 da Strategic Research Corp.

dados são apresentados como tabelas bidimensionais que escondem os mecanismos complexos de armazenamento dos dados. O modelo permite que os programadores não se preocupem com os detalhes de como é que os dados são guardados. Permite ainda que o acesso aos dados seja feito de uma forma lógica através da utilização de declarações SQL, sendo apenas necessário especificar as tabelas, colunas e linhas (registos) dos dados pretendidos.

2.1.2 As linguagens de programação e o SQL

O SQL é uma linguagem declarativa, especializada na interrogação e manipulação dos dados, e que não tem algumas das capacidades de uma linguagem de programação imperativa. Na prática encontramos situações em que precisamos da flexibilidade de uma linguagem de programação, como o C, e das capacidades de manipulação de dados do SQL. Um exemplo é a necessidade de implementar aplicações gráficas que "manipulem" as bases de dados de uma forma transparente (sem que os utilizadores usem o SQL para manipular os dados).

A solução encontrada foi a utilização do SQL no contexto de outras linguagens de programação, **linguagens hospedeiras**. A relação entre a linguagem hospedeira e o SQL é designada por **estilo de ligação** (*binding style*). As duas principais formas de ligação definidas nos vários standards do SQL são o ESQL (SQL Embebido) e CLI (***C**all **L**evel **I**nterface*).

SQL embebido

O SQL Embebido [9] (ESQL) é uma especificação para a inclusão de declarações SQL no código de programas numa linguagem hospedeira. A especificação do SQL-92 definiu o estilo de ligação de sete linguagens (Ada, C, COBOL, FORTRAN, MUMPS, Pascal, e PL/I), e na especificação do SQL3 foi acrescentada mais uma linguagem à lista, o Java.

A sintaxe do ESQL depende da linguagem hospedeira, mas as funcionalidades são semelhantes nas diversas linguagens. Cada declaração SQL é anotada com identificadores que marcam o início e fim da declaração, de forma a distingui-la do código da linguagem hospedeira. Esta abordagem requer a utilização de um pré-compilador

(ou tradutor), providenciado pelo SGBD, que traduza as declarações SQL anotadas no código do programa.

Um dos grandes inconvenientes da utilização de ESQL resulta da necessidade da base de dados ser conhecida e estar disponível durante o desenvolvimento das aplicações. Outro inconveniente é a dependência dos tradutores em relação aos SGBD - como cada tradutor funciona para um dado SGBD é necessário recompilar o código das aplicações com ESQL sempre que se mude de SGBD. A principal vantagem do ESQL reside no facto de ser potencialmente mais rápido que os outros estilos de SQL, uma vez que o tradutor pode realizar optimizações no código gerado. Outra vantagem da utilização do ESQL resulta da realização, pelos tradutores, de uma verificação sintáctica e semântica das declarações SQL encontradas no programa, evitando a ocorrência de erros sintáticos e semânticos nas declarações SQL durante a execução.

SQL/CLI

Nos finais dos anos 80, vários fabricantes de SGBD e de aplicações começaram a reconhecer a necessidade de se poder aceder às fontes de dados[2] de uma forma uniforme. Muitos fabricantes de software tinham aplicações genéricas que precisavam de aceder a várias fontes de dados. Embora a linguagem SQL fosse relativamente portável tornou-se evidente que os programadores precisavam de um modelo alternativo ao ESQL em que não fosse necessário recompilar a aplicação sempre que se mudasse de sistema de gestão de base de dados.

A alternativa encontrada foi a utilização de APIs SQL para aceder a bases de dados. A utilização de uma API não requer a utilização de um tradutor que converta as declarações SQL em código da linguagem hospedeira, e que é depois compilado e ligado à base de dados. Em teoria, a utilização de uma API standard ajuda a escrever aplicações que utilizam bases de dados e que são independentes do SGBD.

O estilo de ligação CLI é simplesmente uma interface para o SQL. As aplicações utilizam CLI para submeter declarações SQL para SGBD, e para executar operações nas bases de dados. O CLI pode ser visto como um encapsulador (*wrapper*) de SQL,

[2] O termo *fonte de dados* (ou "data source") é utilizado por alguns autores para referenciar bases de dados, ou outras formas de manutenção de dados, que suportem a linguagem SQL como linguagem de consulta e manipulação de dados.

Figura 2.1: CLI

que não aumenta nem diminui as capacidades do SQL.

Foi definida uma especificação de uma API CLI standard - X/Open CLI. Foram ainda implementadas outras formas de CLI (ex: ODBC da Microsoft) que são basicamente mutações do X/Open CLI.

2.1.3 Modelo Cliente/Servidor

A divisão das aplicações em clientes e servidores [31] tem sido muito explorada na última década na criação de várias soluções de software que acedem a bases de dados.

O modelo cliente/servidor pode ser descrito como um relacionamento entre processos que correm (geralmente) em máquinas diferentes. Basicamente, o modelo cliente/servidor providencia uma separação clara de funções. O processo servidor (ou simplesmente servidor) é um fornecedor de serviços, e o processo cliente o consumidor de serviços.

A ligação entre o cliente e o servidor é realizada utilizando alguma forma de *middleware*. O *middleware* é um termo vago que cobre todo o software distribuído necessário para suportar interacções entre clientes e servidores. É visto, muitas vezes, como o software que está entre o cliente e o servidor, e que permite a um cliente obter um serviço de um servidor. Alguns autores descrevem o *middleware* como a '/' do modelo cliente/servidor.

2.1.4 Acesso a bases de dados

Os primeiros SGBD, desenvolvidos nos inícios da década de 80, providenciavam um esquema de acesso simples e ineficiente: cada aplicação que quisesse aceder aos dados deveria invocar as bibliotecas do SGBD ou, em alternativa, lançar o programa do SGBD e interagir com ele.

Em meados da década de 80, a indústria de software começou a mudar a forma de desenvolvimento das aplicações com acessos a bases de dados e passou a utilizar uma arquitectura cliente/servidor (descrita em 2.1.3). As aplicações são os clientes, que com uma única declaração SQL, podem ir buscar ou modificar um conjunto de registos na base de dados, através do servidor da base de dados. Os servidores de bases de dados são também designados por servidores SQL, SGBD ou, simplesmente, bases de dados.

Os servidores SQL controlam e executam comandos SQL recebidos dos clientes. Um servidor SQL permite que várias aplicações acedam, em simultâneo, aos dados na base de dados. O servidor SQL, para além de processar os pedidos e retornar os resultados, deve gerir a concorrência, recuperação, segurança, e os aspectos de consistência da base de dados. Quase todos os servidores SQL disponibilizam, no mínimo, funcionalidades ao nível do SQL-89. A maioria dos servidores também inclui algumas características do SQL-92. Alguns servidores oferecem versões proprietárias do SQL3, como funções (*stored procedures*) e *triggers*.

As aplicações acedem aos servidores de bases de dados para obterem serviços relacionados com dados (como a ordenação, selecção, modificação de dados, etc). Para obterem os serviços pretendidos, as aplicações enviam pedidos SQL, como mensagens, para o servidor de bases de dados (ver figura 2.2). Os resultados de cada comando SQL são retornados pelo servidor à aplicação através da rede. A execução do código SQL recebido pelo servidor é feita na mesma máquina onde os dados residem, isto é, o

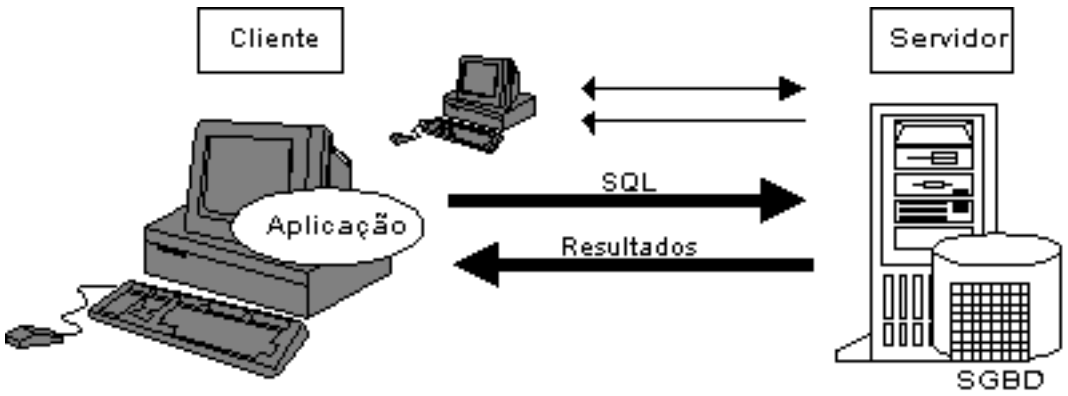


Figura 2.2: Acesso a bases de dados

servidor e os dados residem na mesma máquina. O servidor utiliza a sua capacidade de processamento para encontrar os dados pretendidos, em vez de enviar todos os dados ao cliente para que este efectue a pesquisa nos dados.

A utilização do modelo cliente/servidor na arquitectura das aplicações não garante a interoperabilidade das aplicações. Apesar de poder existir uma API comum a todos os SGBD, continua a ser necessário existir *drivers* para cada SGBD, para além de haver problemas com a utilização das extensões proprietárias ao SQL de cada SGBD. Assim, para garantir a interoperabilidade é necessário que as aplicações não utilizem extensões do SQL proprietárias e que sejam instaladas nas máquinas onde são executados os *drivers* necessários para aceder ao SGBD.

Em meados da década de 90, o modelo cliente/servidor tradicional, também conhecido por duas camadas (*2-tier*), evoluiu para um modelo com três camadas (*3-tier*) permitindo resolver, em certa medida, o problema de interoperabilidade existente no modelo cliente/servidor tradicional.

O modelo com três camadas adiciona um **servidor de aplicações** (ou *gateway*) entre o cliente e o SGBD. Os clientes passaram a comunicar com o SGBD através do servidor de aplicações, utilizando um protocolo de comunicação independente do SGBD. O servidor de aplicações é responsável por executar esses pedidos traduzindo-os para um protocolo que o SGBD entenda.

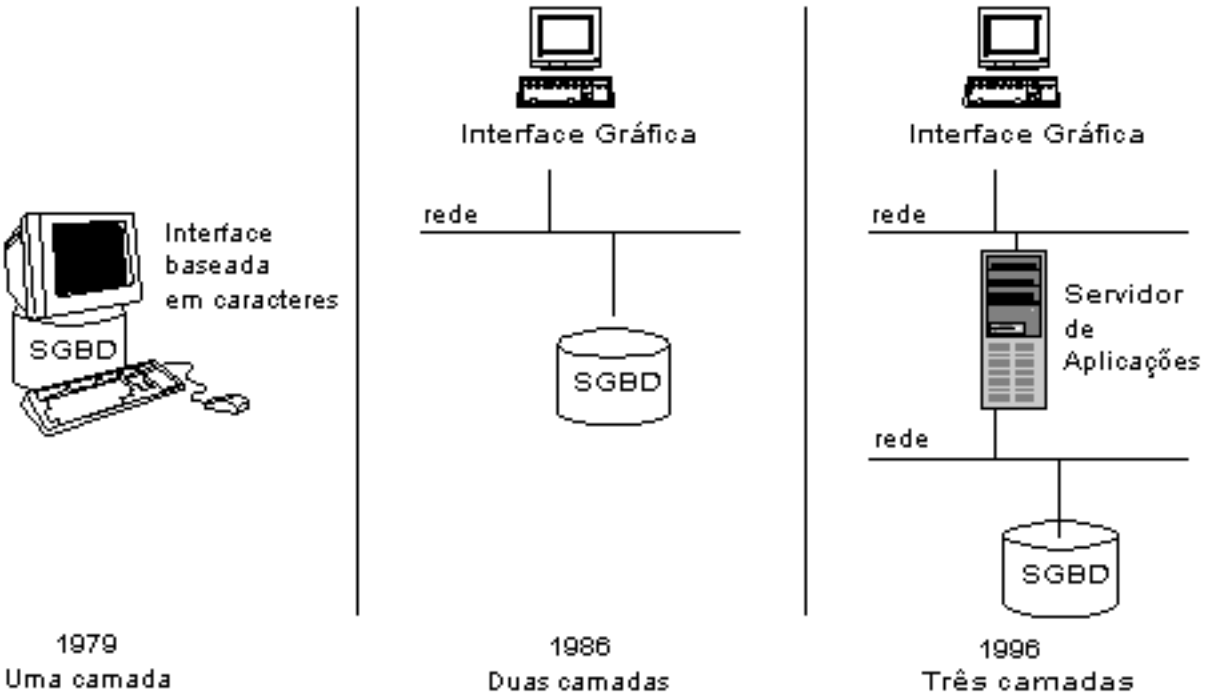


Figura 2.3: Evolução do modelo cliente/servidor

2.2 *Middleware* de acesso a bases de dados

Middleware é um termo vago que cobre todo o software distribuído necessário para suportar interacções entre clientes e servidores. É visto muitas vezes como o software que está entre o cliente e o servidor, e que permite ao cliente obter um serviço de um servidor. O *midleware* fornece uma API, ao cliente, que é usada para invocar um serviço num servidor, e encarrega-se da transmissão do pedido e da recepção da resposta resultante.

Quando se fala de *middleware* de acesso a bases de dados, o serviço providenciado pelos servidores é a execução de declarações SQL enviadas pelos clientes. O *midleware* não inclui o software que providencia o serviço (que é da responsabilidade do servidor da aplicação), não inclui a base de dados, e também não inclui a interface gráfica (que é do domínio da aplicação).

O *middleware* de acesso a bases de dados é o elo de ligação entre as aplicações e as bases de dados. O standard SQL especifica as declarações que as aplicações podem usar, para guardar ou aceder a dados no SGBD, não especificando porém, os passos envolvidos na ligação às bases de dados e os mecanismos de transferência de dados

para as bases de dados, e vice-versa. Normalmente, cada SGBD tem associado um mecanismo de acesso e de transferência de dados específico.

Através de uma interface standard (ex: X/Open CLI) ou proprietária (ex:ODBC), o *middleware* de acesso a bases de dados tenta providenciar às aplicações um mecanismo de entrega de dados independente do SGBD. A adição de uma camada intermédia de *middleware*, que isola a aplicação do SGBD, permite que as várias formas de middleware de acesso a bases de dados apresentem uma interface de acesso comum, independentemente do tipo de acesso fornecido pelo fabricante. Por outro lado, a adição de uma camada suplementar pode aumentar o consumo de memória e tornar mais lento o acesso aos dados.

Quase todas as formas de *middlleware* interoperáveis baseiam-se no ODBC, ou no caso de aplicações Java em JDBC, como forma de acesso às bases de dados. Existem soluções de *middleware* de acesso a bases de dados que disponibilizam outros serviços para além das ligações com bases de dados. Os serviços "extra"disponibilizados vão desde a realização de cache de ligações ODBC e JDBC, ao mapeamento de tabelas em objectos e de objectos em tabelas.

A seguir apresentam-se as principais soluções de *middleware* existentes para aceder às bases de dados, como é o caso do X/Open CLI, ODBC, JDBC, SQLJ, OLE DB e ADO.

2.2.1 X/Open CLI

Nos finais de 1989, as empresas a Microsoft, Lotus, SysBase e DEC desenvolveram em conjunto uma especificação chamada de *SQL Connectivity*. Esta especificação definia um conjunto de funções para aceder às bases de dados. Por essa altura, um consórcio fundado por empresas de bases de dados formou o *SQL Access Group* (SAG), com o objectivo de promover a interoperabilidade do SQL. Em Junho de 1990, o consórcio SAG aceitou a especificação *SQL Connectivity* como documento base para a CLI (***C****all* ***L****evel* ***I****nterface*). Em Dezembro de 1994, o consórcio entregou a versão final da CLI à X/Open[3]. Esta organização é desde então a "guardiã" oficial da SAG CLI, que é agora chamada de X/Open CLI, para a diferenciar do ODBC da Microsoft e das

[3]http://www.xopen.org

outras variantes existentes. No início de 1996, X/Open CLI [12] foi tornado standard internacional.

O standard **X/Open CLI** (ou apenas SQL/CLI) define um conjunto de funções (API) que uma aplicação pode utilizar para aceder a bases de dados SQL. Contém funções para codificar os tipos de dados SQL na linguagem hospedeira e de tratamento de erros. A API SQL/CLI, definida na especificação, permite ainda a ligação a bases de dados através de um *driver*, preparar as declarações SQL, executar as declarações, aceder aos resultados, terminar uma declaração, e terminar a ligação.

Uma aplicação, que utiliza a forma de acesso SQL/CLI, interage com as bases de dados através da invocação de funções definidas na API SQL/CLI. A especificação SQL/CLI requer (ver figura 2.1) o uso de *drivers* que aceitem a invocação das funções da API SQL/CLI e as traduzam para a API de acesso nativa do SGBD. Desta forma é necessária a existência de um *driver* para cada SGBD que uma aplicação pretenda aceder. Cada *driver* é, na maioria dos casos, desenvolvido para um determinado SGBD, usando os métodos de acesso e os protocolos de comunicação específicos do SGBD.

A utilização do SQL/CLI não se resume apenas ao acesso a dados em SGBD. Com o *driver* apropriado, qualquer aplicação que disponibilize dados pode funcionar como um servidor SQL/CLI e permitir que outros programas, que utilizem uma API SQL/CLI, acedam aos seus dados.

A independência do SGBD e a possibilidade de acessos concorrentes por uma aplicação são as principais vantagens que as aplicações podem usufuir por utilizarem uma forma SQL/CLI para aceder a bases de dados.

A especificação SQL/CLI permite que as aplicações sejam independentes do SGBD que utilizam. A independência é obtida através da utilização de uma API standard de acesso às bases de dados, e dos *drivers* que implementam o acesso. Deste modo, não é necessário executar transformações no código de uma aplicação para aceder a um SGBD diferente, basta utilizar um *driver* diferente e apropriado ao SGBD.

Por vezes, as aplicações têm de executar operações concorrentes sobre as bases de dados. A existência de áreas de dados globais no ESQL levanta o problema da visibilidade e do âmbito das alterações feitas através de operações concorrentes nos dados. O SQL/CLI elimina as áreas de dados globais através da introdução do conceito

de *handle* [42]. A utilização de *handles* permite que uma aplicação possa ter várias ligações a uma base de dados e execute comandos SQL concorrentemente, através de cada uma das ligações.

Embora uma das vantagens da forma SQL/CLI, como meio de acesso às bases de dados, seja a independência das aplicações face às bases de dados, isso nem sempre acontece. Cada SGBD oferece, quase sempre, um conjunto único de extensões ao SQL. Por vezes, estas extensões requerem a definição, e implementação, de um conjunto de funções adicionais à API SQL/CLI de forma a ser possível o acesso aos serviços providenciados pelas extensões. Qualquer aplicação que aproveite as capacidades mais avançadas de um SGBD deixa de ser independente do SGBD. Assim, API SQL/CLI acaba por ser o "menor denominador comum" dos métodos de acesso, pois permite o acesso aos serviços que todos os SGBD fornecem.

2.2.2 ODBC

O ODBC (*Open Database Connectivity*) [10] é a implementação de uma versão extendida da especificação X/Open CLI realizada pela Microsoft.

O ODBC é uma API que contém funções que permitem a uma aplicação ligar-se a um SGBD, executar declarações SQL, e receber os resultados. Define ainda códigos de erro e tipos de dados standard.

Os principais objectivos do ODBC foram a definição de uma API standard de acesso a bases de dados, a disponibilização de todas as funcionalidades dos SGBD, a obtenção de uma performance equivalente às APIs nativas[4] do SGBD e inclusão do acesso a outras fontes de dados (por exemplo: folhas Excel).

O ODBC permite o acesso local e remoto a fonte de dados. O acesso é local quando a aplicação e a fonte de dados estão na mesma máquina, caso contrário o acesso é remoto. Os dois componentes principais da arquitectura ODBC (ver figura 2.4) são o gestor de *drivers* e o *driver* propriamente dito. A comunicação entre o *driver* e o SGBD é realizada através da utilização de software e protocolos fornecidos pela empresa que implementou o SGBD. O ODBC define o lado cliente da ligação às bases de dados, isto é, a API disponível para as aplicações, e a API que os *drivers* devem implementar

[4] API nativa de um SGBD - API disponível para as aplicações e que utiliza bibliotecas do SGBD.

para serem acedidas pelos gestores de *drivers* .

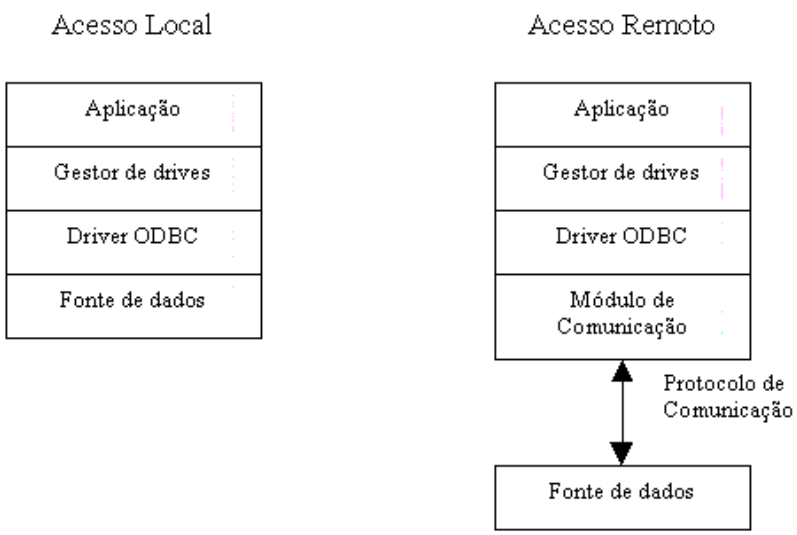


Figura 2.4: Arquitectura ODBC

O gestor de *drivers* (*Driver Manager*) providencia uma API standard às aplicações e permite que a aplicação procure e escolha a fonte de dados que pretende, e questione o *driver* para saber se suporta uma dada funcionalidade. O gestor de *drivers* tem como função carregar, libertar e invocar os diversos *drivers*.

O *driver* ODBC processa as funções ODBC, submete os comandos SQL para a fonte de dados, e retorna os resultados para a aplicação. Os *drivers* dependem do SGBD pois necessitam de aceder à API nativa do SGBD e, no acesso remoto, ao protocolo de comunicações do SGBD.

O protocolo de comunicações pode ser definido pela empresa que implementou o SGBD, ou por terceiros. Neste último caso é necessário que exista um *driver* ODBC para o SGBD.

A adição de uma camada intermédia de *middleware*, que isola a aplicação do SGBD, permite ao ODBC apresentar uma interface de acesso comum, independentemente do tipo de acesso fornecido pelo fabricante, mas por outro lado, consome mais memória e torna mais lento o acesso aos dados.

Uma das principais desvantagens do ODBC resulta do facto de ser uma especificação controlada pela Microsoft. Embora esteja em constante desenvolvimento o seu futuro é incerto, pois a Microsoft tem-se empenhado no OLE DB (descrito em 2.2.5) como uma forma universal de acesso a dados, que introduz um paradigma de programação diferente - é orientado a objectos em vez de ser funcional. Outro dos problemas do ODBC resulta do facto de ser visto como uma interface alternativa, quer para as aplicações, quer para os SGBD. Por exemplo, a Oracle suporta ODBC como uma API opcional e a sua API CLI nativa é a *Oracle Call-level Interface* (OCI). O SGBD DB2 da IBM suporta ODBC e X/Open CLI, mas o seu protocolo nativo é ESQL/DRDA. A única excepção é o SQL Server da Microsoft que usa o ODBC como protocolo nativo.

2.2.3 JDBC

A linguagem *Java*^{tm} proíbe a execução de código nativo durante a execução dos programas(*applets*) num *browser*. Este facto implica que as applets não podem invocar ODBC ou outras interfaces que não estejam implementadas em Java. Transformando os *drivers* em classes Java, as *applets* podem carregar, durante a execução, os *drivers* que necessitam para aceder aos SGBD.

O JDBC [43, 40] (***J**ava **D**atabase **C**onnectivity*), da Javasoft, é uma SQL/CLI implementada inteiramente em Java. O JDBC é uma colecção de *drivers* de acesso a base de dados que fornecem aos programas Java uma CLI API. As applets e aplicações podem usar a API para estabelecer ligações com os SGBD, guardar e aceder a dados no SGBD, e executar operações na base de dados.

O JDBC tem como objectivos a disponibilização de uma API JAVA de acesso a SGBD, o respeito pela especificação SQL, e que a API seja rápida e fácil de usar.

Tal como o ODBC e SQL/CLI, o JDBC providencia dois grandes conjuntos de interfaces: uma interface para a aplicação que permite o acesso aos serviços SQL de uma forma independente do SGBD, e uma interface para os *drivers* que os fornecedores de SGBD têm de adaptar aos seus SGBD. O JDBC usa o gestor de *drivers* para carregar automaticamente o *driver* necessário para aceder à base de dados pretendida.

O JDBC, assim como o ODBC, baseia-se no conceito de DSN (Data Source Name) para identificar o nome de uma ligação a uma base de dados. Para além do DSN,

o JDBC usa uma forma de URL (Uniform Resource Locator) para descrever o tipo de ligação (JDBC), o *driver*, o nome da máquina, o número da porta e o DSN (ex. jdbc:openlink://khampa:6000/DSN=Teste).

A arquitectura do JDBC (ver figura 2.5) baseia-se em quatro tipos de *drivers*. Os *drivers* diferem na quantidade de código Java que contêm e no protocolo que usam para comunicar com o servidor de bases de dados.

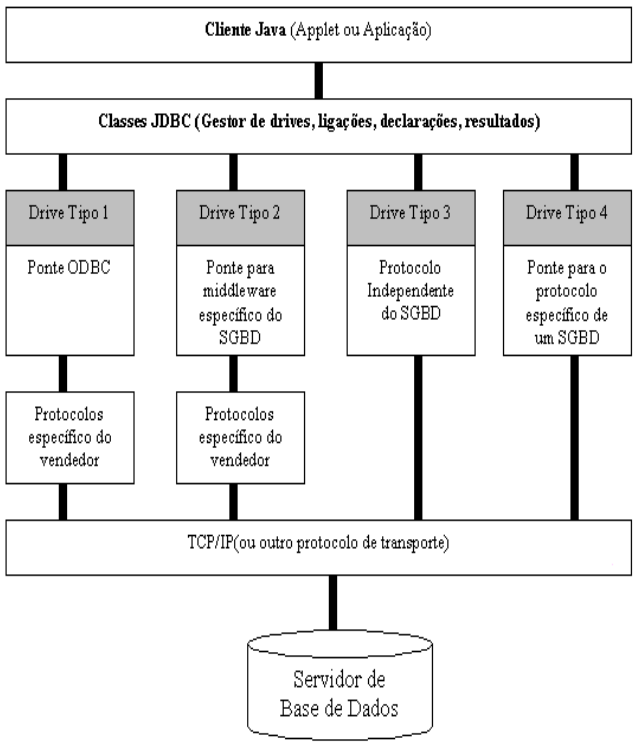


Figura 2.5: Arquitectura JDBC

Um *driver* do tipo 1 é uma ponte JDBC-ODBC que providencia acessos a SGBD via *drivers* ODBC. A máquina que faz a ponte JDBC-ODBC necessita de ter o *driver* ODBC e o *driver* proprietária do SGBD instalada e configurada. Este tipo de *driver* foi definido para funcionar com os SGBD que não têm *drivers* JDBC.

O *driver* tipo 2 consiste numa interface, escrita parcialmente em Java, entre programas Java e o *middleware* de acesso ao SGBD (ex: Oracle SQL*Net). Este tipo de *driver* JDBC é praticamente uma ponte directa para a CLI do SGBD (ex: Oracle OCI). Não necessita da presença do driver ODBC, mas requer a instalação e configuração do *middleware*.

O *driver* tipo 3 é escrito inteiramente em Java. Transforma as invocações de funções no cliente em pedidos ao SGBD, utilizando um protocolo independente do SGBD. No lado do servidor, uma componente traduz os pedidos em chamadas nativas ao SGBD. Os *drivers* tipo 3 são capazes de ligar clientes Java a quaisquer SGBD que tenham a parte servidor do *driver*. Este tipo de *drivers* não necessitam da presença de *middleware* ou ODBC, necessitam apenas de poderem efectuar simples ligações TCP/IP. Os programas Java que utilizam este tipo de *drivers* podem ser considerados altamente genéricos, pois correm em qualquer plataforma com Java e TCP/IP.

O *driver* tipo 4 é 100% Java e converte os pedidos JDBC num protocolo específico do SGBD. Como o *driver* depende dos protocolos de acesso ao SGBD, este tipo de *driver* deverá ser criado e disponibilizado pela empresas que implementaram os SGBD.

2.2.4 SQLJ

Em Abril de 1997 a Oracle, IBM, e a divisão Tandem da Compaq formaram um consórcio para definir e avançar com uma especificação de uma implementação de ESQL para a linguagem $Java^{tm}$. Pouco tempo depois, outras empresas juntaram-se ao consórcio - SyBase, Informix e a JavaSoft (criada a partir da Divisão de Software Java da Sun). Em Dezembro de 1998 a especificação, chamada de SQLJ, foi aceite como standard ANSI X3.135.10-1998 e inserida posteriormente no SQL3.

O SQLJ é uma forma de ESQL, que permite inserir declarações SQL no código de programas Java, e ser usado para implementar *stored procedures* em Java.

As principais empresas fornecedoras de SGBD, como a Oracle, IBM e Informix, já providenciam tradutores de SQLJ ou estão em vias de providenciar, como é o caso da SyBase. A maioria destes tradutores geram código Java com chamadas JDBC a partir das declarações SQL.

Sendo uma forma de ESQL, o SQLJ necessita de um tradutor que converta as declarações SQL em chamadas a funções Java antes de ser compilado pelo compilador de Java. A forma como a tradução é feita é dependente da implementação e, por vezes, do SGBD. Os próximos parágrafos descrevem como funciona a conversão do SQLJ para Java, realizada pelo tradutor da Oracle [35].

Ao escrever uma aplicação utilizando o SQLJ, o programador escreve um programa

em Java com declarações SQL embebidas utilizando a sintaxe do SQLJ. Após a escrita do programa, o tradutor é executado sobre o ficheiro Java com o programa SQLJ. O tradutor converte o programa SQLJ num programa Java eliminando o SQL embebido, através da troca das declarações SQL por chamadas a funções JDBC. O código Java gerado pelo tradutor pode ser, de seguida, compilado pelo compilador de Java.

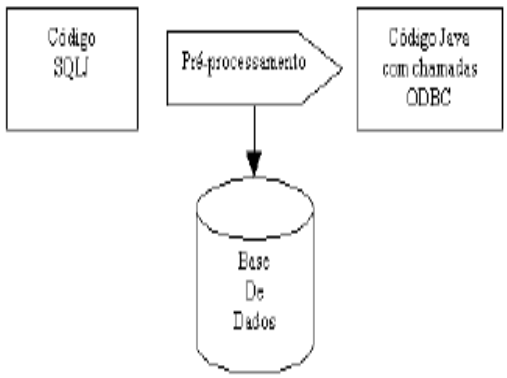


Figura 2.6: Conversão de SQLJ para Java

O tradutor, numa primeira fase, faz a análise sintáctica do código SQLJ, invocando o analisador sintáctico do Java e dos construtores SQLJ. O analisador sintáctico do Java é invocado para verificar a sintaxe das expressões e variáveis Java dentro das declarações SQLJ. Se o tradutor descobrir erros sintáticos durante esta fase, a tradução é abortada e os erros são reportados ao programador.

Na segunda fase, o tradutor entra na fase de verificação semântica. Nesta fase são executadas basicamente três tarefas:

1. Análise dos tipos Java das expressões Java dentro das declarações SQLJ.
2. Verificação da sintaxe de operações de SQL DML nas declarações SQLJ.
3. Comparação dos tipos das expressões Java, dentro do SQLJ, com os tipos SQL das respectivas colunas na base de dados, argumentos de funções ou de retorno. É ainda verificado se os objectos do esquema da base de dados (com tabelas, vistas e *stored procedures*) existem realmente na base de dados.

Por fim, se não houver erros sintáticos e semânticos, o código é traduzido para Java puro. As declarações SQLJ são convertidas em invocações de funções JDBC.

As principais vantagens, face ao JDBC, do SQLJ são o aumento da portabilidade das aplicações e da produtividade dos programadores.

As aplicações SQLJ obtêm a portabilidade de três formas: a sintaxe do SQLJ foi desenhada como sendo independente do SGBD, e o tradutor SQLJ faz suposições mínimas sobre o SQL convertido; os elementos do consórcio partilham um tradutor comum; o código Java gerado a partir do SQLJ é standard.

O programador consegue aumentar a produtividade uma vez que o código escrito utilizando SQLJ é mais compacto que o JDBC, e porque durante a compilação é feita uma verificação sintáctica, semântica, de tipos e de esquema das declarações SQL do programa, e os programas são mais fáceis de ler e escrever.

2.2.5 OLE DB e ADO

A Microsoft disponibilizou as primeiras componentes da infraestructura OLE DB em Outubro de 1996. Nos dias de hoje, a Microsoft explora OLE DB e ADO em quase todos os seus produtos, incluindo o Internet Explorer, Windows 98, NT, e Windows 2000.

OLE DB define um conjunto de objectos COM (***C****omponent* ***O****bject* ***M****odel*) e interfaces para esses objectos. ADO (***A****ctiveX* ***D****ata* ***O****bject*) providencia uma API de acesso a objectos dados sobre o OLE DB - é uma CLI para este novo tipo de sistema orientado a objectos. O ADO e OLE DB fazem parte da estratégia de acesso a dados da Microsoft, chamada Universal Data Access (UDA), cujo objectivo consiste em providenciar uma forma genérica de acesso a dados, independentemente da sua estrutura e localização.

OLE DB define um conjunto de interfaces (e classes) que medeiam a comunicação entre os fornecedores de dados (*data providers*) e os consumidores de dados (*data consumers*).

Os fornecedores de dados são componentes OLE DB que representam uma fonte de dados específica - dados SQL, folhas de cálculo, sistemas de ficheiros, páginas HTML, etc. Os fornecedores retornam os dados usando uma abstração OLE DB com o nome de *rowset* - é uma tabela que pode conter tabelas inseridas nas suas colunas, e pode ser vista como uma tabela multi-dimensional. O *result set* do ODBC, pelo contrario, retorna tabelas bi-dimensionais. Os consumidores de dados são aplicações

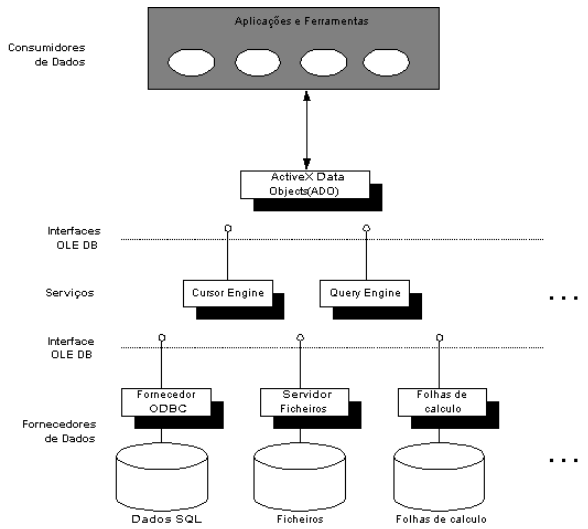


Figura 2.7: OLE DB e ADO

que acedem aos dados; OLE DB providencia um conjunto de classes que podem ser usadas para consumir os dados. Os componentes de serviços (*service*) funcionam como consumidores e fornecedores. Fornecem funções adicionais sobre os dados, como junções e manipulações de cursores.

ADO é um modelo de programação de mais alto nível que o OLE DB. É um substituto de dois protocolos de acesso a dados da Microsoft - Data Access Object (DAO) e Remote Data Objects (RDO). ADO suporta uma grande variedade de ferramentas e linguagens de programação, incluindo Visual Basic, Delphi, Java, e Javascript. Também providencia uma componente que fornece um serviço remoto de dados (***R****emote* ***D****ata* ***S****ervice* - RDS) que suporta a *cache* no lado do cliente (este serviço é descrito na secção 3.2.1).

O modelo orientado a objectos da ADO consiste em: 1) objectos de ligação que representam uma ligação com uma fonte de dados; 2) objectos de comandos que representam uma consulta (ou comando) para ser executada na fonte de dados; e 3) um objecto que representa os resultados de uma consulta (*recordset*). Este modelo é bastante semelhante ao encontrado no JDBC e ODBC.

2.3 Middleware de acesso a bases de dados

Na secção anterior foram descritas algumas formas de *middleware* que providenciam o acesso a SGBD. O *middleware* pode fornecer outros serviços para além da comunicação, nomeadamente providenciar mecanismos que simplifiquem a implementação de aplicações que utilizem bases de dados.

A simplificação da implementação de aplicações que acedam a bases de dados pode ser obtida através da utilização de *middleware* que: reflicta automaticamente os resultados das consultas às bases de dados em estruturas de dados; permita a modificação dos dados na estrutura de dados criada (sem a utilização de SQL); e propague à base de dados as alterações realizadas na estrutura.

A reflexão dos resultados das consultas aumenta a produtividade dos programadores uma vez que estes dispendem menos tempo quer na escrita de código para processar os resultados de consultas, quer na construção das declarações SQL para efectuar as alterações nas bases de dados.

Nas subsecções seguintes são descritos dois produtos de *middleware* que reflectem os resultados das consultas em estruturas de dados, o Java Blend e dbKona.

2.3.1 Java Blend

O $JavaBlend^{tm}$ [2] é um produto da Sun^{tm} que permite o acesso transparente a bases de dados relacionais.

O principal ponto forte do JavaBlend é o mecanismo que faz o mapeamento relacional-objecto e objecto-relacional.

Ao realizar o mapeamento relacional-objecto, o JavaBlend gera automaticamente classes Java para um esquema completo de uma base de dados. As consultas à base de dados retornam objectos, o que significa é desnecessário traduzir o conjunto dos resultados. Ao fazer o mapeamento objecto-relacional, o Javablend gera o SQL e o código Java necessário para manter os dados da aplicação de forma persistente na base de dados. O programador pode definir as regras de tradução, assim como renomear os campos.

O JavaBlend para além de oferecer um mecanismo de persistência de dados transpar-

ente, também permite aceder a bases de dados de uma forma transparente.

2.3.2 dbKona

O dbKona [6] da WebLogic providencia uma forma de acesso a bases de dados para a linguagem Java.

O dbKona é constituído por um conjunto de objectos Java que simplificam o trabalho com os dados de SGBD. Pode ser utilizado com qualquer driver JDBC para comunicar com multiplos e heterógeneos SGBD de diferentes fabricantes.

O dbKona providencia um nível de abstração superior ao JDBC, que lida com os detalhes de baixo nível da gestão dos dados. Com os objectos dbKona, um programador pode utilizar os objectos de uma base de dados sem conhecimentos especiais sobre a metodologia de cada SGBD para guardar, actualizar e consultar os dados, ou sobre os tipos de dados específicos de cada SGBD. O dbKona providencia métodos para a geração automática de SQL: com um conhecimento geral de SQL, mas sem conhecimentos específicos do SQL do SGBD, um programador pode escrever consultas, actualizações e apagar registos num SGBD. Claro que o dbKona também permite o envio de SQL "puro"para o SGBD (por exemplo, para a criação tabelas).

O dbKona lida com os dados do SGBD através de *DataSets*. Um *DataSet* é uma representação de alto nível dos dados do SGBD que contém objectos chamados *Records*, que por sua vez contêm objectos chamados *Values*. Um objecto *Value* encapsula os dados (a sua estrutura e tipo). O *DataSet* permite uma grande liberdade na navegação dos registos que o constituem, ao contrário do *ResultSet* do JDBC, que está limitado a iterar sempre para o registo seguinte através do método next(). O objecto *DataSet* é uma abstração usada pelo dbKona para fazer *cache* dos registos retornados pelo SGBD. É utilizado para ir buscar os registos, enviar registos, e modificar os registos no SGBD. O *DataSet* é, em certa medida, equivalente a uma tabela SQL.

O dbKona disponibiliza métodos e objectos que providenciam a geração automática de SQL. Por exemplo, em vez de providenciar uma declaração SQL como "`select * from Empregados where salario` $<$ `100000`", o programador pode construir um objecto TableDataSet com apenas dois argumentos - um objecto que representa a ligação ao SGBD, e o nome da tabela da base de dados - e de seguida pode utilizar

o método where() do objecto para definir os parâmetros da consulta. O SQL para ir buscar os registos é gerado automaticamente quando o primeiro registo é acedido.

Capítulo 3

Sistemas de Cache

A utilização de *caches* é um tópico bem estudado em diversas áreas [23], nomeadamente no desenho de sistemas de memória de computadores, na gestão de memória virtual, em sistemas de ficheiros, em bases de dados e por diversos serviços na Internet como DNS (*Domain Name Service*), FTP (*File Transfer Protocol*) e WWW (*World Wide Web*).

Este capítulo começa por introduzir os conceitos relacionados com sistemas de *cache*, seguindo-se uma descrição do trabalho relacionado com a *cache* de resultados de consultas a base de dados.

É importante realçar que apesar da vasta utilização, os termos e informação encontrada sobre *caches* são muito específicos da área da sua aplicação. A aproximação seguida foi a de apresentar os conceitos relacionados com as *caches* de uma forma genérica, isto é, de uma forma independente da área da sua aplicação.

3.1 Introdução

"Cache: um lugar seguro para esconder ou guardar coisas"

in Webster's new World Dictionary of the American Language

A *cache* foi implementada pela primeira vez nos anos 60, quando a velocidade dos processadores começou a ultrapassar a velocidade das memórias. Para optimizar a performance dos processadores, foram introduzidas pequenas, e rápidas, memórias auxiliares (ou *cache*) entre o processador e a memória central para diminuir o tempo de acesso à memória central.

A *cache* pode ser implementada ao nível do hardware ou do software. Em qualquer dos casos, a implementação de uma *cache* é guiada por dois objectivos. Por um lado, deve tentar reduzir o tempo de acesso aos dados, isto é, reduzir a latência. Por outro lado, a *cache* deve ser transparente para o consumidor, isto é, o consumidor ao aceder aos dados não se deve aperceber que existe um nível intermédio entre ele e a fonte de dados.

Os sistemas de *cache* baseiam-se no **princípio de localidade de referência** (*principle of locality of reference*) que assume que os dados acedidos mais recentemente têm uma grande probabilidade, num futuro próximo, de serem acedidos novamente. Em [17] são apresentadas três propriedades da localidade de referência: a localidade temporal; a localidade espacial; e a localidade geográfica. A localidade temporal determina que os dados acedidos recentemente serão provavelmente acedidos novamente. A localidade espacial determina que os dados "próximos"[1] dos dados recentemente acedidos serão provavelmente acedidos novamente. A localidade geográfica determina que os dados acedidos por um consumidor serão provavelmente acedidos outra vez por consumidores "próximos"[2].

O acesso a dados pode ser visto, de uma forma geral, como tendo dois intervenientes (figura 3.1): a fonte de dados e o consumidor. A **fonte de dados** é vista como o local que contém os dados (ex: memória de um computador, bases de dados, servidores de HTTP). O **consumidor** consulta e/ou modifica os dados na fonte de dados. Por exemplo, no caso da fonte de dados ser um servidor HTTP o consumidor pode ser

[1] A noção de proximidade depende do tipo de dados.

[2] A noção de proximidade neste caso corresponde à proximidade física.

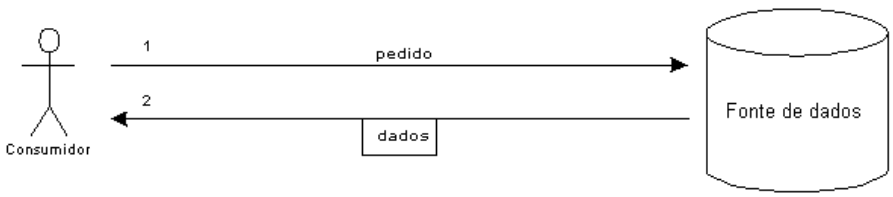


Figura 3.1: Acesso a dados sem utilizar uma *cache*

um *browser* que pede páginas e as apresenta ao utilizador. Outro exemplo de um consumidor é um processador, pois acede e modifica valores na fonte de dados - a memória. O consumidor para aceder aos dados formula pedidos à fonte de dados, que de seguida retorna os dados pretendidos. O canal de comunicação, os dados enviados no pedido e os dados recebidos na resposta dependem da fonte de dados e do consumidor. O **canal de comunicação** é o meio utilizado para transmitir e receber dados (ex: *bus*, rede).

A *cache* é colocada habitualmente entre o consumidor e a fonte de dados (figura 3.2) e mantém uma pequena parte dos dados disponíveis na fonte de dados. Quando o consumidor necessita de determinados dados efectua o pedido correspondente à *cache*. A *cache* verifica se os dados resultantes do pedido estão em *cache*. Se os dados acedidos estão em *cache* diz-se que houve um **acerto** (*cache hit*) na *cache*. Se os dados não estiverem em *cache* diz-se que houve uma **falha** (*miss*) na *cache*.

Quando ocorre um *hit* os dados são retornados pela *cache* ao consumidor como se tivessem sido retornados pela fonte de dados. Se ocorre uma falha, o pedido é passado para a fonte de dados. De seguida, a fonte de dados envia à *cache* a resposta, com os dados resultantes do pedido, que depois os envia para o consumidor. Os novos dados enviados pela fonte de dados à *cache*, depois de uma falha, poderão ser guardados na *cache* de forma a tentar aproveitar a localidade temporal que o consumidor possa exibir em acessos futuros. A decisão de guardar os novos dados em *cache* é tomada pela política de troca utilizada pela cache.

As *caches* não se limitam a manter cópias de dados. Precisam de assegurar que as cópias em *cache* são actualizadas após modificação dos dados originais na fonte de dados.

A manutenção em *cache* de alguns dos dados poderá proporcionar uma redução no tempo de acesso aos dados pelo consumidor. Uma vez que a distância entre o

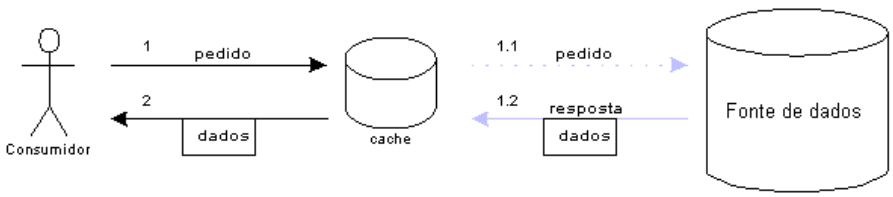


Figura 3.2: Acesso a dados através de uma *cache*

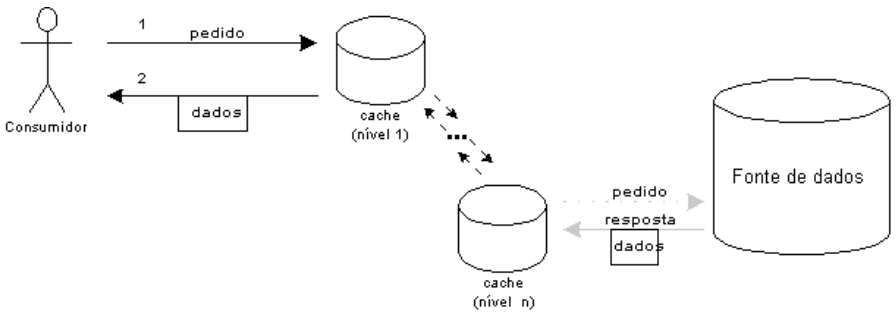


Figura 3.3: Acesso a dados através de várias *caches*

consumidor e a *cache* é mais pequena que a distância entre o consumidor e a fonte de dados, o tempo dispendido na comunicação é reduzido sempre que os dados pretendidos pelo consumidor estejam na *cache*. Desta forma, a *cache* é frequentemente colocada o mais próxima possível do consumidor, de forma a minimizar o tempo dispendido na comunicação.

Podem ainda existir várias *caches* entre um consumidor (ver figura 3.3) e uma fonte de dados, isto é, diversos **níveis de *cache***. A *cache* de nível i (i=1,2,3,..) é a i-ésima *cache* mais próxima do consumidor. Portanto, a *cache* mais próxima do consumidor designa-se por *cache* de nível 1. O consumidor envia o pedido para a *cache* de nível 1, que caso ocorra uma falha, passa-o para o nível seguinte. Sempre que ocorra uma falha num nível da *cache*, o pedido é passado à *cache* do nível seguinte, até atingir o último nível. Quando atinge o último nível o pedido é feito à fonte de dados. Uma *cache*, em qualquer dos níveis, pode ser partilhada por vários consumidores, ou por várias *caches*, de forma a explorar a localidade geográfica.

Neste capítulo o cenário utilizado, na descrição dos vários tópicos, é o de um consumidor, uma cache e uma fonte de dados (figura 3.2). A escolha deste cenário, em detrimento do cenário com vários consumidores e vários níveis de *cache*, permite uma descrição mais simples do funcionamento dos vários tópicos relacionados.

3.1.1 Implementação

Ao nível da implementação, uma *cache* poder ser vista como uma tabela indexada. Cada linha da tabela, ou entrada na *cache* (*cache entry*), contém os dados resultantes de um acesso à fonte de dados, um identificador e, possivelmente, mais alguma informação adicional.

Os dados mantidos pela *cache* resultantes de um acesso à fonte de dados são designados por (**objecto**). Como cada objecto na *cache* é resultante de um acesso, é necessário associar o objecto ao acesso que o originou. Assim, a cada objecto é associado um **identificador** (*tag*). Este identificador é gerado a partir do pedido. A forma de geração do identificador depende da forma como a tabela é implementada. Os identificadores são utilizados para pesquisar objectos na *cache*. Associada a cada entrada da *cache* podemos ainda ter: o tempo em que a entrada foi criada; o tempo da última actualização do objecto; e a indicação se o objecto é válido, isto é, se o seu valor pode ser utilizado pelo consumidor.

As *caches*, após a recepção de um pedido, devem pesquisar rapidamente as cópias dos dados mantidas em *cache*. Por isso, as técnicas de pesquisa utilizadas nas *caches* são fundamentais para o seu desempenho. Técnicas como a pesquisa linear não são apropriadas pois são muito lentas. As técnicas baseadas em tabelas de hashing [25] são muito utilizadas por permitirem pesquisar rapidamente.

3.1.2 Falhas na *cache*

As falhas na *cache* podem ser categorizadas [22] em falhas compulsórias (*compulsory misses*), falhas de capacidade (*capacity misses*) e falhas de colisão (*collision or conflict misses*). Estes três tipos de falhas são designadas por 3 C's (**C**ompulsórias, **C**apacidade e **C**olisão).

As **falhas compulsórias** são as falhas resultantes do primeiro acesso feito aos dados. Este tipo de falhas é inevitável, uma vez que os dados só poderão estar em *cache* após o primeiro acesso. O peso desta falha inicial na performance diminui à medida que vão sendo feitos mais pedidos à *cache*.

As **falhas de capacidade** ocorrem quando não é possível guardar um objecto em

cache por falta de espaço. Neste caso, a *cache* tem duas possibilidades: ou remove alguns objectos em *cache* para dar espaço ao novo; ou não insere o objecto na *cache*. O número de falhas de capacidade tende a diminuir à medida que se aumenta o tamanho da *cache*.

As **falhas de colisão** ocorrem quando a *cache* remove um objecto da *cache* para inserir um objecto novo, mesmo havendo espaço para os dois. Isto ocorre quando os dois objectos são guardados na mesma entrada da cache. Para evitar este tipo de falhas as *caches* devem ser implementadas de modo que a localização dos objectos não resulte em conflitos.

3.1.3 Coerência da *cache*

A utilidade das *caches* baixa consideravelmente se a propriedade de coerência [27] não for respeitada. Segundo esta propriedade os valores das cópias na *cache* devem ser actualizados ou invalidados sempre que os respectivos dados na fonte de dados são alterados. As cópias em *cache* cujos valores na fonte de dados sejam diferentes dizem-se **desactualizadas** (*stale*).

Os mecanismos de manutenção da coerência garantem que as cópias na *cache* são eventualmente actualizadas de modo a reflectir as alterações nos dados originais. A altura em que uma alteração na fonte de dados é visível para o consumidor é definida pelo modelo de consistência. Uma alteração é visível para o consumidor quando este efectua uma operação de leitura e obtém o valor modificado. Ao falarmos do modelo de consistência da *cache* estamos a descrever a ordem de execução das operações de escrita e leitura na fonte de dados, e da altura em que estas se tornam visíveis para o consumidor. Isto é, estamos a definir que dados é que um consumidor espera obter quando executa um acesso à fonte de dados. Uma *cache* é consistente se respeitar o modelo de consistência existente originalmente entre o consumidor e a fonte de dados.

A utilização de *caches* pode alterar o modelo de consistência existente entre o consumidor e a fonte de dados. Se as *caches* não forem mantidas coerentes, as cópias dos dados retornados ao consumidor podem estar desactualizadas. Neste caso, o consumidor obtém um resultado diferente do que obteria se estivesse a pedir os dados directamente à fonte de dados. A falta de coerência das *caches* resulta, assim, numa alteração do modelo de consistência. Por outras palavras, a manutenção da coerência

da *cache* implica a consistência da *cache*.

Em [19] são apresentados dois tipos de consistência, a consistência forte (*strong consistency*) e a consistência fraca (*weak consistency*). A **consistência forte** é definida como o modelo de consistência em que a *cache* não retorna nenhum valor desactualizado ao consumidor, e a **consistência fraca** é definida como o modelo de consistência que permite o retorno de dados desactualizados ao consumidor.

A manutenção da coerência da *cache* requer a utilização de mecanismos de manutenção da coerência localizados nas *caches* e/ou nas fontes de dados. Podem-se distinguir os mecanismos de manutenção da coerência quanto ao tipo de consistência que asseguram, consistência forte ou consistência fraca. Os principais mecanismos de manutenção da coerência de *cache* que asseguram uma consistência fraca são: tempo de vida (***T****ime* ***T****o* ***L****ive* ou TTL), e *Client-Polling*. A consistência forte pode ser obtida através da utilização da invalidação, actualização e *Polling every-time*.

Os mecanismos de manutenção de coerência existentes também podem ser classificados de forma a indicar quem é que detém a tarefa de assegurar a coerência, se o consumidor ou se a fonte de dados. O *Client-Polling* pode ser classificado como pertencendo ao primeiro caso. No segundo caso são englobados os mecanismos baseados no protocolo de invalidação e no protocolo de actualização.

O protocolo de invalidação necessita que a fonte de dados mantenha um registo dos objectos copiados pelas várias *caches*. De cada vez que um objecto é alterado, o servidor verifica nos registos se alguma *cache* tem uma cópia do objecto. As *caches* que tiverem cópias são notificadas que a cópia que detêm é inválida.

O protocolo de actualização é semelhante ao protocolo de invalidação. A principal diferença reside no envio dos dados actualizados à *cache* em vez de uma simples mensagem de invalidação, ficando ao critério da *cache* a actualização dos dados incoerentes.

A escolha entre o protocolo de invalidação ou actualização, com o objectivo de obter uma melhor performance, deve basear-se nos padrões de acesso aos dados. A avaliação de cada um dos protocolos é feita através do número de falhas e do tráfego gerado. No caso do consumidor voltar a aceder a um objecto, após uma modificação, a actualização imediata permitirá obter melhores resultados que a invalidação. No caso de os objectos actualizados não voltarem a ser acedidos o tráfego resultante das actualização torna-se

inútil, e neste caso será preferível utilizar o protocolo de invalidação.

Em [22] são apresentados casos, baseados na *cache* de memória, em que cada opção é substancialmente melhor que a outra. Uma alternativa à escolha entre invalidação e actualização é a utilização de uma solução híbrida que combina ambos os protocolos. Nesta solução híbrida, a escolha entre entre o protocolo de actualização e invalidação é feita durante a execução para cada objecto invalidado. Idealmente, deveria ser possível prever durante a execução as referências futuras a um objecto e decidir, com base na previsão, entre invalidação ou actualização do objecto. No entanto, a informação necessária para realizar as previsões nem sempre está disponível. Uma solução encontrada para ajudar nas previsões passa pela análise histórica dos acessos como forma de prever os acessos futuros [33].

O principal problema com os protocolos de invalidação ou actualização está na necessidade da manutenção de um registo dos objectos mantidos nas caches pela fonte de dados. Esta necessidade pode introduzir problemas de escalamento ou a necessidade de utilização de diversos níveis de cache. Outro problema com os protocolos de invalidação resulta da necessidade de serem feitas alterações na fonte de dados para manter o registos dos objectos pedidos. Em alguns casos poderá não ser possível fazer as alterações uma vez que pode ser impossível modificar a fonte de dados[3].

O mecanismo baseado em tempo de vida consiste na associação a cada objecto de uma estimativa do seu tempo de vida (TTL). O TTL é usado para determinar o tempo que o objecto vai ser considerado válido na *cache*, isto é, quando o TTL termina o objecto em *cache* é considerado inválido. O desafio no uso de tempo de vida está em seleccionar o valor correcto para o tempo. Frequentemente atribui-se um valor baixo para que os dados sejam recarregados mais vezes, com o objectivo de diminuir o número de dados incoerentes retornados pela *cache*.

Client Polling é uma técnica em que a *cache* verifica periodicamente na fonte de dados se as cópias dos dados que detém ainda estão válidas. O uso desta técnica não garante que os dados retornados pela *cache* estejam coerentes. Durante o período de tempo entre duas validações, os dados em *cache* são considerados válidos mesmo sendo alterados na fonte de dados durante esse período. Esta técnica pode ser alterada de forma a assegurar uma consistência forte. Para isso, sempre que algum pedido é feito à

[3]No caso de a fonte de dados ser um software comercial é muito provável que o seu código fonte não esteja disponível ao público, o que impede que terceiros possam realizar as modificações.

cache e os dados resultantes desse pedido estão em *cache*, a *cache* contacta a fonte de dados para validar a cópia dos dados. Esta técnica é referida como *Polling-every-time*.

3.1.4 Políticas de Troca

A capacidade da *cache*[4] é limitada e, normalmente, bastante menor que a capacidade da fonte dos dados. Quando toda a capacidade da *cache* está ocupada por objectos diz-se que a *cache* está cheia. Se ocorrer uma falha, e a *cache* estiver cheia, é necessário que um (ou vários objectos) sejam removidos da *cache* de forma a obter espaço para a inserção do novo objecto. A política de troca (*Replacement Policy*), também designada por Política de Remoção, define a forma de selecção dos objectos a remover.

Existe uma grande variedade de propostas de políticas de troca. Iremos descrever apenas cinco das políticas de troca: remoção aleatória, *First In First Out*, *Least Recentlty Used*, *Least Frequently Used* e *Size policy*.

O algoritmo de remoção aleatória, como o nome indica, selecciona os objectos a serem removidos da *cache* de forma aleatória. Como a remoção é aleatória, não são considerados factores como o tempo de estadia em *cache* do objecto, nem a frequência dos acessos. Assim, a performance obtida da utilização deste algoritmo é fraca, pois permite a remoção, da *cache*, de objectos inseridos recentemente ou acedidos com muita frequência, diminuindo assim a probabilidade de ocorrência de acertos. A principal vantagem deste algoritmo reside na sua fácil implementação.

O algoritmo ***F**irst **I**n **F**irst **O**ut* [41, 39] (FIFO) selecciona os objectos da *cache* que foram inseridos em primeiro lugar (os mais antigos). Uma possível implementação deste algoritmo consistiria em associar a cada objecto na *cache* um valor, como a hora em que o objecto foi colocado na *cache*, que indicasse a ordem de chegada (por ordem ascendente) do objecto à *cache*. Este valor seria utilizado para indicar a idade dos objectos (quanto menor o valor, mais antigo o objecto). Apesar de não permitir a obtenção de boas performances, este algoritmo é ligeiramente superior ao algoritmo de remoção aleatória, pois explora o tempo de estadia do objecto na *cache*.

O algoritmo *Least Recently Used* [13, 41] (LRU) selecciona o objecto que não é usado à mais tempo. A cada objecto na *cache* é associado um valor que representa o tempo

[4]Espaço em disco ou memória onde são guardados os dados em *cache*

em que o objecto foi acedido pela última vez. Este valor é actualizado, sempre que o objecto é acedido, com o tempo do acesso. A implementação deste algoritmo é ligeiramente mais difícil que os anteriores. Contudo, permite a obtenção de melhores resultados, especialmente se os acessos apresentarem uma forte localidade temporal. A lacuna deste algoritmo reside na não utilização, como critério de selecção, do tamanho dos objectos ou da latência na obtenção do objecto.

Com o algoritmo *Least Frequently Used* [14, 39] (LFU) é seleccionado o objecto em *cache* com menor número de acessos. Ao nível da implementação, este algoritmo é semelhante ao LRU. A principal diferença reside no valor associado a cada objecto. No caso do LFU é associado um contador com o número de acessos ao objecto pelo consumidor, enquanto que no LRU é mantido o tempo do último acesso. Uma desvantagem potencial do LFU reside na possibilidade de os objectos em *cache* acumularem um grande número de referências numa fase inicial e não voltarem a ser utilizados. Uma vez que o objecto tenha sido muito acedido, o valor do seu contador será grande e o objecto poderá manter-se em *cache*, mesmo que não volte a ser utilizado. Para atenuar este problema, pode ser utilizada uma política de envelhecimento para os objectos em *cache*, como a proposta em [34]. A política de envelhecimento requer dois parâmetros: A_{max}, que define um limite para o número de referências médio de todos os objectos em *cache*; e M_{refs}, que impõe um limite máximo para o valor que os contadores de referências podem conter. Quando o número médio de referências dos objectos na *cache* ultrapassa A_{max}, o contador de referências de cada objecto é reduzido por um factor de dois.

A política de troca baseada no tamanho (*Size*) dos objectos [44] selecciona os maiores objectos da *cache* para remoção. Ao nível da implementação, este algoritmo é parecido com LRU e LFU. A principal diferença reside no valor associado a cada objecto, que no caso deste algoritmo é o tamanho do objecto.

As outras políticas de troca existentes são variações ou combinações das cinco políticas apresentadas. Por exemplo, a *GreedyDual-Size* [18] que combina o tamanho com a idade de um objecto em *cache*. Cao e Irani propuseram diversas variações desta política [18]. A política LRU-K [29, 30] também considera a frequência de acessos e a idade do objecto para decidir o(s) objecto(s) a remover. Este algoritmo mantém um registo com as últimas K referências para os objectos, e usa esta informação para determinar estatisticamente os intervalos de referência aos objectos. A política

LRV [28] (*Lowest Relative Value*) selecciona o objecto para troca com o menor valor relativo entre os objectos em *cache*. O valor do objecto é calculado baseado em informações históricas como o número de acessos realizados, o tempo do último acesso e o tamanho do objecto.

3.1.5 Políticas de Escrita

Os consumidores para além de executarem as operações de leitura, também podem realizar operações de escrita através de uma *cache*. Neste caso, em vez de realizar a operação de escrita directamente na fonte de dados, o consumidor executa a operação de escrita na *cache*.

A **política de escrita** (*write policy*) da *cache*, também conhecida como política de actualização (*update policy*), define a forma da *cache* executar as operações de escrita. Isto é, define **onde** são escritos os dados, se na *cache* e/ou na fonte de dados, e **quando** é que são escritos. As políticas de escrita mais conhecidas são o *write-trough* e *write-back* (também conhecido por "copy-back").

Se uma *cache* utiliza a política de escrita ***write-through*** [38], as operações de escrita do consumidor são realizadas na *cache* e na fonte de dados. A utilização desta política garante a coerência dos dados na fonte de dados, uma vez que esta contém sempre os valores actualizados

Quando o consumidor executa uma operação de escrita sobre a *cache*, esta necessita de verificar se a operação de escrita é feita sobre algum objecto em *cache*, isto é, se ocorre um **acerto de escrita** (*write hit*). Para isso, é usada a mesma técnica de pesquisa utilizada quando o consumidor executa uma operação de leitura. Se acontecer um acerto de escrita, a operação de escrita é realizada sobre o objecto na *cache*. Caso contrário, é utilizada a política de troca para decidir se os dados resultantes da operação de escrita são mantidos em *cache*. Para terminar, a operação de escrita pode ser executada na fonte de dados. A ordem de execução da operação de escrita pode ser trocada, isto é, executando primeiro na fonte de dados e depois na *cache*.

Uma variante desta política consiste na execução, pela *cache*, das operações de escrita apenas na fonte de dados. A *cache* apenas recebe os pedidos de execução das operações de escrita, passando-os de seguida à fonte de dados.

Usando a política de ***write-back*** [38], as operações de escrita são realizadas na *cache*, mas não são propagadas à fonte de dados imediatamente. A operação de escrita só é realizada na fonte de dados quando os dados em *cache* resultantes da escrita forem removidos pela política de troca. A utilização desta política permite eliminar o *overhead* de ter de escrever várias vezes os mesmos dados na fonte de dados. Contudo, tem a desvantagem da fonte de dados poder ficar incoerente em relação à *cache*.

3.1.6 Localização da *cache*

Em sistemas distribuidos, as *caches* podem estar localizadas nos clientes (consumidores) , e/ou em *proxies*, e/ou em servidores (fontes de dados).

Como já foi referido, um dos objectivos principais da utilização de uma *cache* é a possibilidade de reduzir os acessos aos servidores permitindo aos clientes reduzir a latência no acesso aos dados.

A colocação da ***cache* no cliente** permite, sempre que ocorra um acerto na *cache*, reduzir o número de pedidos ao servidor, e consequentemente diminuir a comunicação com o servidor. A redução do número de pedidos feitos ao servidor, resultante dos acertos na *cache*, tem como benefício a diminuição da carga de trabalho dos servidores, permitindo-lhes responder mais rapidamente aos pedidos recebidos. A diminuição da comunicação, entre o cliente e o servidor, permite diminuir o consumo de largura de banda da rede.

Através dos acertos na *cache* e da consequente redução do número de pedidos feitos ao servidor, a *cache* no cliente permite reduzir bastante a latência nos acessos a dados. No entanto, não permite a exploração da propriedade de localidade geográfica (não permite a partilha das *caches* por vários clientes) o que limita os ganhos obtidos.

O ganhos de performance resultante da utilização de *caches* dependem do número de acertos. Se a *cache* for partilhada por vários clientes, o número de acertos tenderá a aumentar, através da exploração da localidade geográfica. A utilização de ***cache* em *proxies***, ou *caches* intermédias, surgiu como uma forma de explorar a localidade geográfica.

Um *proxy* funciona como mediador entre o cliente e terceiros (outros *proxies* ou

servidores). Do ponto de vista do cliente, o *proxy* funciona como um servidor: recebe pedidos de clientes e depois envia-lhes uma resposta. Do ponto de vista do servidor, o *proxy* funciona como um cliente: faz os pedidos (recebidos dos clientes), e recebe as respostas (retornadas pelo servidores) que de seguida envia ao respectivo cliente.

O *proxy* é assim o local ideal para colocar uma *cache*. Como é partilhada por vários clientes, a localidade geográfica é explorada nos acessos. A partilha da *cache* permite aumentar, comparativamente a utilização de *cache* nos clientes, a probabilidade de ocorrer um acerto.

A implementação de *cache* no servidor tem como objectivo a redução do tempo de resposta, pelo servidor, aos pedidos mais frequentes. Para isso, o servidor mantém as respostas dos pedidos mais frequentes em *cache*. Quando um pedido é recebido, a *cache* é pesquisada na procura da resposta a esse pedido. Se a resposta estiver em *cache* é enviada logo para o cliente. Se não estiver em *cache*, a resposta é calculada e, dependendo da política de troca, guardada na *cache*.

A implementação da *cache* na fonte de dados torna mais fácil a manutenção da consistência dos dados. Permite ainda implementar uma política de troca mais eficiente, pois a fonte de dados tem uma melhor ideia da frequência da alteração dos dados do que os consumidores.

3.1.7 Métricas de avaliação

A análise do desempenho do ganho na utilização da *cache* pode ser feita segundo três perspectivas: do consumidor; do canal de comunicação; e da fonte de dados. Para o consumidor, a *cache* é avaliada pelo tempo que demora a retornar um objecto. Isto depende do tempo que demora a: retornar um objecto da cache; a realizar o pedido à fonte de dados no caso de ocorrer uma falha. Do ponto de vista do canal de comunicação, o principal objectivo é a redução de tráfego. Para a fonte de dados a utilização de *caches* pelos consumidores potencia uma redução no número de pedidos recebidos.

Algumas das métricas utilizadas para avaliar o sucesso do uso de uma *cache* são: taxa de acertos (*hit ratio*); taxa de falhas (*miss rate*); e a taxa de transferência de bytes.

A **taxa de acertos** (***H**it **R**ate* - HR) expressa a percentagem dos acessos que resultaram em acertos. Por outras palavras, a HR define a probabilidade de ocorrer um acerto. Quanto maior for a taxa de acertos, maior será o ganho de performance resultante da utilização da *cache* em termos de redução da latência [18].

$$\text{Taxa de acertos(HR)} = \frac{\#acertos}{\#acessos} \tag{3.1}$$

A **taxa de falhas** (***M**iss **R**ate*) exprime a percentagem de acessos à *cache* que resultaram no acesso à fonte de dados. Por outras palavras, define a probabilidade de ocorrer uma falha.

$$\text{Taxa de falhas(MR)} = \frac{\#falhas}{\#acessos} \tag{3.2}$$

A **taxa transferência de bytes** (***B**yte **H**it **R**ate* - BHR) indica a fracção do número de bytes resultantes de acertos na *cache*. A BHR utiliza as seguintes estatísticas:

- s_i : o tamanho do objecto i
- p_i : a probabilidade de encontrar o objecto i na *cache*

A taxa de transferência de bytes é definida como:

$$BHR = \frac{\sum s_i p_i}{\sum s_i} \tag{3.3}$$

A partir do BHR [28] podemos obter o ganho médio (***s**peedup* - s) obtido com a utilização da *cache* como a razão dos tempos de acesso com e sem a *cache*.

$$s \approx \frac{1}{1 - BHR} \tag{3.4}$$

Esta aproximação baseia-se no pressuposto que o tempo de acesso à cache é desprezível quando comparado com o tempo de acesso à fonte de dados.

3.2 Optimização das comunicações com BDs

Um grande número de aplicações acede remotamente (de computadores diferentes) a servidores de base de dados. A comunicação com as bases de dados é, muitas vezes, a grande fonte de estrangulamento da performance dessas aplicações, principalmente se a operação envolver a transferência de muitos dados. Este estrangulamento depende principalmente de dois factores, o tráfego na rede e a velocidade de resposta do Sistema de Gestão de Base de Dados (SGBD, ou simplesmente, base de dados). Deste modo, a optimização das comunicações entre as aplicações e as bases de dados permitirá melhorar a performance das aplicações.

A *cache* dos resultados das consultas pelas aplicações poderá ser uma solução atractiva para optimizar as comunicações. Esta solução potencia a redução: da latência, uma vez que o resultado de uma consulta poderá estar disponível imediatamente a partir da *cache*; da carga de trabalho no SGBD, uma vez que terá de responder a um número menor de pedidos; e do tráfego na rede resultante dos pedidos e respostas. A utilização da *cache* poderá ser particularmente atractiva quando os resultados são originados de consultas complexas, pois a latência nestes casos tenderá a ser maior.

Relacionado com a utilização de *caches* está o problema da manutenção da coerência da *cache*. As aplicações ao utilizarem os resultados de uma consulta esperam que esses resultados estejam coerentes com os dados na base de dados. Assim, a *cache* de resultados de consultas requer a utilização de protocolos de coerência que garantam uma consistência forte.

3.2.1 *Cache* de resultados de consultas a bases de dados

Tem sido efectuada bastante investigação na *cache* de resultados de consultas no servidor, como a descrita em [20], principalmente na *cache* de resultados de consultas intermédias. São ainda propostas algumas soluções baseadas na *cache* de resultados de consultas nos clientes, como as propostas em [24] e [36]. Ambas as propostas não asseguram uma consistência forte, e baseiam-se na ideia de transferir parte do trabalho de processamento das consultas para o cliente.

Tanto quanto é do nosso conhecimento, só existem quatro sistemas que utilizam a

cache de resultados de consultas como forma de optimização das comunicações com bases de dados: o WATCHMAN [37], *data caching* da NetDynamics [4], *Remote Data System* [7, 11], e TimesTen [1].

O WATCHMAN é um sistema de gestão de *cache* para conjuntos de dados, resultantes de consultas a bases de dados. Este sistema foi implementado de forma a ser utilizado como uma biblioteca de funções por uma aplicação, mais especificamente, por um gestor de *Data Warehouses* [5]. Este sistema não garante a coerência dos dados mantidos na *cache*. Assume que a aplicação detecta as alterações, realizadas na base de dados, que afectam a coerência da *cache*.

O servidor de aplicações NetDynamics dispõe de um sistema de *cache* que permite realizar a *cache* de resultados de consultas. Este sistema mantém a coerência dos dados em *cache* através da utilização de um mecanismo baseado em TTL que garante uma consistência fraca.

O RDS (***R**emote **D**ata **S**ervice*) é uma das componentes do ADO, da Microsoft. O RDS permite a *cache* dos resultados de consultas a base de dados. A *cache*, no RDS, é implementada por um motor de cursores (***C**lient **C**ursor **E**ngine*) que residem no lado cliente. O CCE é uma colecção de estruturas de dados que implementa a *cache*, na memória ou no disco, dos resultados de consultas realizados a bases de dados, e que propaga as modificações realizadas pelo cliente às base de dados. O CCE, e por conseguinte o RDS, não asseguram uma consistência forte, uma vez que não é descrita a utilização de nenhum mecanismo de manutenção da coerência.

O *Times-Ten Front Tier* é um sistema comercial da TimesTen Performance Software. Este sistema permite manter no cliente uma *cache* com os resultados de consultas a bases de dados, e executar operações SQL sobre os dados em *cache*. Este sistema para além de não garantir a consistência dos dados, também não é dinâmico. Não é dinâmico porque só são mantidos em *cache* os dados quentes (*hot data*), que são definidos durante a fase de desenvolvimento da aplicação.

[5] *Data warehouse* é um repositório integrado de várias, possivelmente heterogéneas, bases de dados e é normalmente utilizada no processamento de análise de dados e sistemas de apoio à decisão (SAD).

3.2.2 *Cache* na *WWW*: semelhanças e diferenças

Os mecanismos baseados em *caches* são bastante utilizados na *World Wide Web* (WWW). A utilização da *cache* na WWW é feita a diversos níveis: pelos clientes, que está embutido no navegador (*browser*); pelos *proxies* [13]; e pelos servidores [17].

Existem diversos pontos em comum entre a *cache* de documentos na WWW e a *cache* de resultados de acessos a bases de dados: a velocidade de transmissão de dados (uma vez que partilham as mesmas redes); a variabilidade do tamanho das respostas (o tamanho da resposta recebida pelo cliente varia bastante em ambos os casos); e a arquitectura (seguem um modelo cliente-servidor [31]).

Existem também algumas diferenças. A *cache* de resultados de acessos a bases de dados requer uma consistência forte, enquanto que a *cache* na WWW utiliza mecanismos de manutenção da coerência de *cache* que asseguram uma consistência fraca. A frequência com que os documentos na WWW são modificados é geralmente muito menor do que a frequência com que os dados nas bases de dados são modificados.

Apesar das diferenças, os estudos realizados sobre *cache* na WWW como forma de redução da latência são uma boa base de partida para resolver o mesmo problema nos acessos a bases de dados. A implementação de *caches* de resultados de consultas a bases de dados deverá poder ser feita pelos clientes e *proxies*, como acontece na WWW. A principal diferença reside na necessidade de se manter uma consistência forte dos dados mantidos nos diversos níveis da *cache*.

3.2.3 Outras formas de optimização das comunicações

As soluções encontradas para optimizar as comunicações entre aplicações e bases de dados não se resumem à *cache* de resultados de consultas.

Uma das opções existentes para reduzir a latência no acesso a bases de dados passa pela *cache* de ligações [15, 16]. Quando uma aplicação necessita de uma ligação, a *cache* com as ligações é pesquisada. Se a ligação estiver em *cache* é imediatamente retornada para a aplicação, e esta não precisa de passar pelo processo completo de estabelecer uma ligação. Se a ligação não estiver em *cache*, a ligação é estabelecida com a base de dados.

Esta solução é vantajosa porque a abertura de uma ligação com uma base de dados pode ser lenta. Embora permita a criação mais rápida das ligações, não oferece mais nenhuma melhoria na redução da latência.

Capítulo 4

Modelo Proposto

Neste capítulo apresenta-se um modelo para optimizar as comunicações entre as aplicações e fontes de dados.

O modelo baseia-se na utilização pelas aplicações de mecanismos de *cache* que permitam reduzir a latência no acesso aos dados e a comunicação entre a aplicação e a fonte de dados. Este modelo tem como grande objectivo assegurar a coerência dos dados mantidos nas *caches* através da utilização de mecanismos de manutenção da coerência.

4.1 Introdução

Muitas aplicações necessitam de aceder a dados localizados noutros programas que partilham informação (**fontes de dados**) como os servidores de Bases de Dados, Servidores HTTP, só para citarmos alguns. A figura 4.2 descreve, de uma forma simplificada, a forma das aplicações acederem aos dados nas fontes de dados. As aplicações acedem aos dados através da invocação de funções disponibilizadas por uma API. As funções da API encarregam-se do envio de mensagens à fonte de dados, contendo a operação (acesso) de leitura ou escrita, e da leitura da respectiva resposta. A comunicação entre a API utilizada e a fonte de dados foi designada, para abreviar a sua referência, como **comunicação nativa**.

Figura 4.1: Modelo tradicional

A comunicação entre a aplicação e a fonte de dados pode ser optimizada através da utilização de uma *cache* que mantenha os resultados da execução das operações na fonte de dados. Como foi referido no capítulo 3, associado à utilização da *cache* está o problema da manutenção da sua coerência. Para implementar um mecanismo de manutenção da coerência baseado, por exemplo no mecanismo de invalidação ou actualização, torna-se necessário alterar o código da fonte de dados e da aplicação (o que nem sempre é possível).

Nas próximas secções apresenta-se um modelo que potencia a redução das comunicações entre aplicações e fontes de dados com base na *cache* dos resultados da execução de operações (nas fontes de dados), e que garante a coerência da *cache* através da utilização de mecanismos como a invalidação ou actualização. O modelo contempla

uma solução, baseada na utilização de *bridges* (pontes) e *gateways*, para o caso de não ser possível modificar o código da(s) aplicação(ões) e/ou das fontes de dados. É ainda considerada a possibilidade da utilização de *proxies* que mantêm uma *cache* com os resultados da execução de operações.

A secção 4.2 descreve o modelo básico, onde as fonte de dados e as aplicações podem ser modificadas de forma a implementarem o mecanismo de manutenção da coerência da *cache*. A secção 4.3 descreve a utilização de *gateways* para os casos em que as fontes de dados não podem ser modificadas. A secção 4.4 apresenta as "pontes" (*bridges*) como a solução para integrar aplicações que não podem ser modificadas. A secção 4.5 extende o modelo de forma a permitir a *cache* em *proxies*, tentando tirar partido da localidade geográfica nos acessos das várias aplicações.

4.2 Modelo Básico

A versão mais simples do modelo envolve a alteração das aplicações e da fonte de dados o que implica, como é obvio, o acesso ao código fonte da aplicação e fonte de dados.

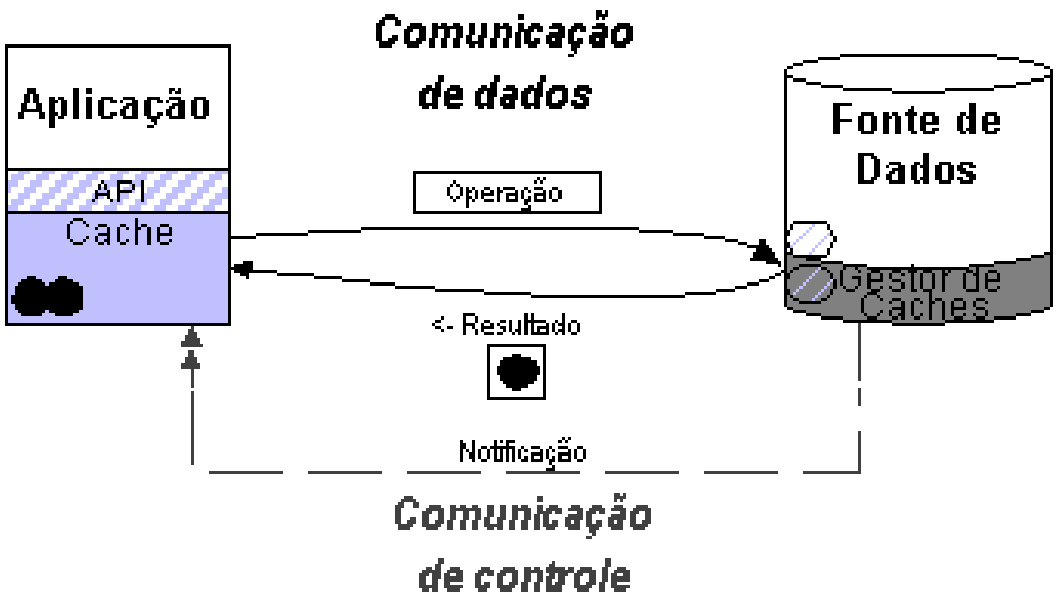


Figura 4.2: Modelo básico

Como se explicou anteriormente, as aplicações acedem às fontes de dados através da invocação de métodos de uma API. No modelo proposto (ver figura 4.2), os métodos

ao serem invocados não enviam a operação para ser executada na fonte de dados, mas sim sobre a componente ***Cache***. A *cache* mantém os resultados das operações já executadas sobre a fonte de dados. A execução de uma operação envolve a consulta prévia da *cache* para esta verificar se detém o resultado da operação. Em caso afirmativo, o resultado é retornado imediatamente à aplicação, evitando-se deste modo o acesso à fonte de dados. Caso contrário a operação é enviada para o **Gestor de *Caches*** (GC) na fonte de dados que executa a operação e retorna o resultado. Ao receber o resultado, a componente *cache* guarda-o (sempre que possível), de forma a poder utiliza-lo no futuro quando a aplicação tornar a executar uma operação idêntica.

A componente *cache* na aplicação permite a execução de operações nas fontes de dados, usando sempre que possível os resultados mantidos na *cache*. Na fonte de dados, o GC (Gestor de *Caches*) recebe e executa as operações enviadas pelas aplicações. Cabe-lhe fazer o registo de todos os pedidos recebidos e a sua origem, de forma a poder notificar as *caches* que os resultados das operações que poderão deter já não estão coerentes. A **notificação** consiste no envio de mensagens de invalidação ou actualização.

Subjacnte a este modelo está a preocupação de reduzir o impacto da sua implementação nas aplicações. Neste sentido, a nova API utilizada pela aplicação (para aceder aos dados através da componente *cache*) deverá apresentar uma interface idêntica à API original (API nativa), embora a sua implementação seja obviamente diferente. A utilização de uma interface idêntica permite reduzir as alterações a efectuar, no código da aplicação, à mudança do nome da API. A outra alternativa passa pela criação de novas API (com novas interfaces) que utilizam a *cache*, o que acarretará a realização de um maior número de alterações no código fonte das aplicações.

4.3 *Gateways*

O modelo básico, proposto na secção anterior, necessita que sejam realizadas alterações na fonte de dados e nas aplicações. No entanto, embora a modificação das fontes de dados seja essencial para manter os dados coerentes nas diversas *caches* nem sempre é possível realizar essas alterações. Por exemplo, os sistemas de gestão de bases de dados são um tipo de fonte de dados em que não é possível alterar o código fonte. Isto resulta do facto da maioria dos SGBD serem proprietários e de, naturalmente, o seu

código fonte não estar disponível.

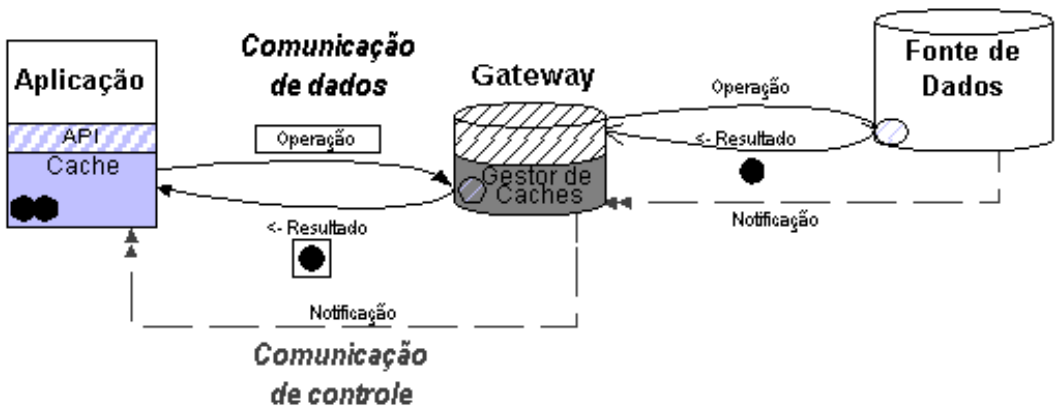


Figura 4.3: Acesso à fonte de dados através de uma *gateway*

Para os casos em que não é possível alterar o código da fonte de dados a solução passa pela utilização de *gateways*. A cada fonte de dados é associada uma *gateway*, que funciona para as aplicações como a fonte de dados. Cada *gateway*, é assim, a porta de acesso à fonte de dados, isto é, todos os acessos a uma fonte de dados passam pela *gateway*. Na figura 4.3 podemos observar como a *gateway* se integra no sistema proposto.

O GC que no modelo básico era adicionado à fonte de dados passa para a *gateway*. A única diferença no GC reside na forma de execução das operações na fonte de dados. Ao ser colocada na *gateway*, o GC necessita de utilizar uma API (talvez a mesma que era utilizada no modelo básico) para executar as operações na fonte de dados.

Para que a *cache* seja mantida coerente é necessário que **todos** os acessos à fonte de dados passem pela *gateway* respectiva. No entanto poderão surgir situações em que os acessos não passam pela *gateway*, minando assim os esforços de manter as *caches* coerentes. Tendo em vista contornar este problema as fontes de dados que normalmente incluem mecanismos semelhantes aos *triggers* existentes nas bases de dados, podem ser configuradas de forma a enviar uma notificação à *gateway* sempre que ocorra uma alteração nos seus dados.

4.4 Pontes (*Bridges*)

Nem sempre é possível alterar a fonte de dados de forma a incluir o GC. Também pode acontecer não ser possível modificar algumas das aplicações que acedem à fonte de dados. É essencial que estas aplicações acedam aos dados na fonte de dados, através do respectivo GC (localizado na fonte de dados ou numa *gateway*) para garantir a coerência das *caches*.

A integração das aplicações que não contêm a componente *cache* passa pela utilização de pontes (*bridges*). As pontes fazem a tradução do protocolo nativo (utilizado entre a API nativa e a fonte de dados) para o protocolo de dados (utilizado entre a *cache* e o Gestor de Caches).

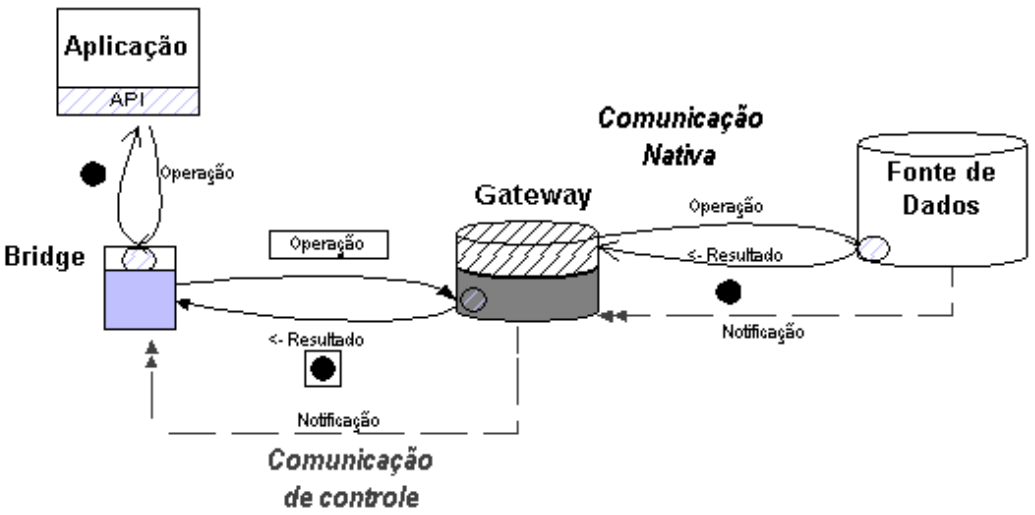


Figura 4.4: Acesso à fonte de dados através de uma "ponte"

A figura 4.4 mostra como é que as pontes se integram no modelo. As aplicações sem a componente *cache* enviam as operações para a fonte de dados através de uma ponte. Esta última, dependendo da situação, envia-as à fonte de dados directamente ou através de uma *gateway*.

4.5 *Proxies*

Como foi referido em 3.1.6 uma *cache* pode estar localizada nas aplicações, nas fontes de dados e em *proxies*. Já foi descrita a possibilidade das aplicações poderem conter

a componente *cache*. Nesta secção iremos abordar a *cache* em *proxies*.

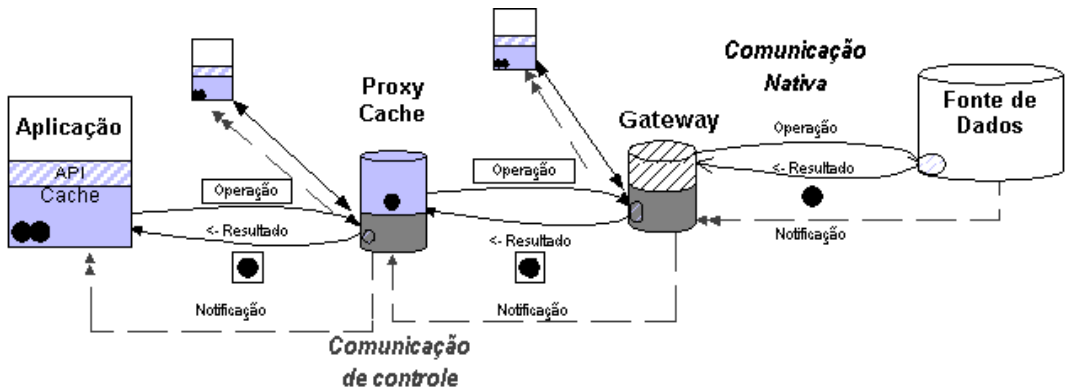


Figura 4.5: Acesso à fonte de dados através de um *proxy*

A principal vantagem de se ter uma *cache* em *proxies* reside na exploração da localidade geográfica. Na figura 4.5 podemos observar a forma de integração desta nova componente no modelo.

Para as *gateways* um *proxy* é uma aplicação. Cabe a cada *proxy*, sempre que notificado de uma alteração na fonte de dados, propagar a notícia para as aplicações suas clientes. Para as aplicações um *proxy* é a fonte de dados (ou a *gateway*). Um proxy é, basicamente, a junção das duas componentes do modelo: a *Cache* e o Gestor de *Caches*.

O modelo para optimizar as comunicações entre aplicações e bases de dados descrito neste capítulo foi implementado no sistema Polites, que é descrito no capítulo seguinte.

Capítulo 5

Polites

Este capítulo descreve o desenho e implementação do sistema Polites. O Polites é um sistema intermédio (*middleware*) de acesso a bases de dados que implementa o modelo apresentado no capítulo 4, que visa optimizar as comunicações entre aplicações e fontes de dados.

5.1 Introdução

Um dos objectivos desta tese foi o desenvolvimento de um sistema, baseado na utilização de mecanismos de *cache*, que optimizasse a comunicação entre aplicações e bases de dados. Contudo, tentou-se ir mais longe e abstrair o problema para o de optimizar as comunicações entre aplicações e fontes de dados.

Para atingir o objectivo proposto foi implementado um sistema baseado na utilização de uma *cache* que mantém os resultados das operações efectuadas na fonte de dados pelo cliente (aplicação), de forma a aumentar a velocidade de reexecução de operações, pelos clientes. A este sistema foi dado o nome de **Polites**[1].

A concepção do sistema Polites teve por base os seguintes objectivos:

- **modularidade** - no sentido de permitir lidar com outros tipos de dados para além dos resultados de consultas a bases de dados;
- **transparência** - permitir que as aplicações, ao acederem às fontes de dados através do Polites, não se apercebam da sua existência;
- **assegurar a coerência dos dados mantidos na *cache*** - através da utilização de mecanismos como a invalidação ou actualização.

As secções seguintes descrevem a arquitectura e modo de funcionamento, as principais componentes e a implementação do Polites.

5.2 Arquitectura

O Polites é um *middleware* de acesso a dados que funciona entre as aplicações e fontes de dados (SGBD em particular). A arquitectura do Polites, esquematizada na figura 5.1, segue o modelo cliente/servidor e baseia-se na ideia descrita no capítulo anterior.

O Polites pode ser utilizado pelas **aplicações** para acederem às fontes de dados (**aplicações Polites cliente**), ou para acederem e partilharem dados (**aplicações**

[1] Polites é o nome de uma personagem da mitologia grega célebre pela sua velocidade.

Polites). A ***gateway* Polites**, ou simplesmente, *gateway* funciona como a porta de acesso à fonte de dados. Assume-se que: 1) todos os acessos à fonte de dados passam pela *gateway*; ou 2) a fonte de dados dispõe de algum mecanismo semelhante aos *triggers* existentes em SGBD, que permitirá enviar uma **notificação** à *gateway* indicando que os dados foram alterados.

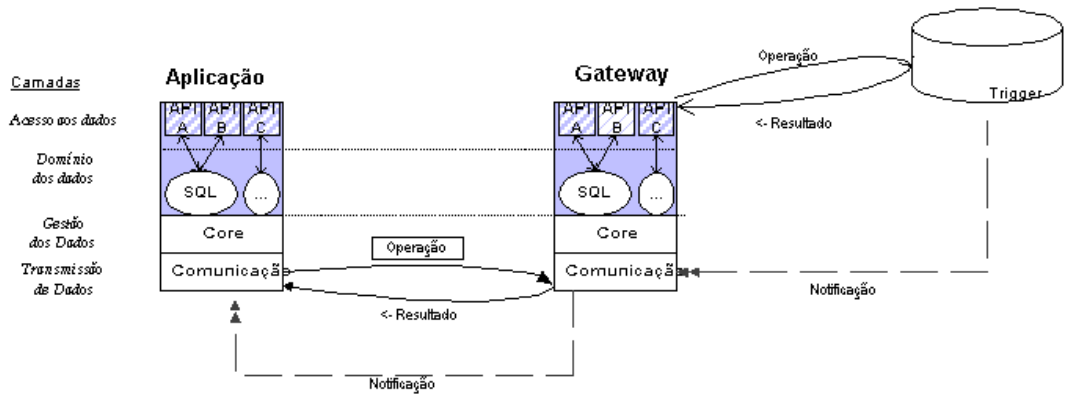


Figura 5.1: Arquitectura do Polites

Como já foi referido, um dos objectivos do Polites era a realização de *cache* de diversos tipos de dados, para além de resultados de consultas a bases de dados. De modo a atingir esse objectivo, o sistema (ver figura 5.1) foi dividido em quatro camadas (*layers*):

- transmissão de dados
- gestão de dados
- domínio de dados
- acesso aos dados

A camada de acesso providencia as API de acesso aos dados. As API são divididas em duas partes, a parte cliente e a parte servidor, que se complementam. No lado do cliente é disponibilizada a interface de acesso aos dados, enquanto que no lado do servidor são implementados os métodos da interface.

A divisão das API, na parte cliente e na parte servidor, pode ser vista de uma forma análoga à separação, na programação orientada a objectos, da interface de

um objecto (cliente) da sua implementação (servidor). Deste modo, a invocação de um método na API cliente pode ser vista como a invocação remota de uma função. A execução remota é feita através da camada do domínio de dados .

A camada do domínio de dados constitui o elo de ligação entre a camada de acesso e a camada de gestão de dados, e tem como função providenciar os mecanismos necessários de manipulação de cada tipo de dados (resultados de operações sobre bases de dados, documentos Web, etc). Para cada tipo de dados é necessária a implementação do módulo de dados correspondente. Tal como a camada de acesso, também esta camada está divida em duas partes, a parte cliente e a parte servidor.

A camada de acesso a dados fornece, no lado cliente, os mecanismos para executar funções remotamente: faz a codificação das operações e parâmetros (*marshalling*), e a descodificação dos resultados. No lado do servidor, implementa os mecanismos de execução de funções: descodifica as operações e parâmetros (*unmarshalling*); executa a função; e codifica o resultado. Para cada domínio de dados que se pretenda suportar no sistema, deverá existir um módulo que implemente um conjunto de funções específicas para esse domínio. Assim, desenvolveu-se o módulo SQL (descrito em 5.4.1) para permitir o uso do sistema no acesso a bases de dados.

A camada de gestão de dados fornece serviços relacionados com o tipo de dados, de forma a permitir a gestão da *cache* e a execução das funções Esta camada providencia ainda os mecanismos necessários para executar operações remotas e manter, se possível, os resultados dessas operações em *cache*, assim como garantir a manutenção da coerência da *cache*.

A camada de comunicação fornece os mecanismos de envio e recepção de mensagens, incluindo a comunicação de controle. Esta camada, tal como as outras, também contém uma parte cliente e uma parte servidor.

As camadas de Gestão e Transmissão de dados asseguram a comunicação entre as componentes cliente e servidor do Polites, a manutenção em *cache* das respostas e a manutenção da coerência da *cache*. A terceira camada, domínio de dados, assegura a independência do sistema em relação ao domínio de dados. A quarta camada, providencia as interfaces de acesso aos dados.

Funcionamento do Polites

Com o objectivo de facilitar a compreensão do funcionamento do sistema descreve-se, nos próximos parágrafos, a sua utilização por uma aplicação e uma *gateway* Polites, em que a aplicação acede, através da *gateway*, a uma base de dados.

Suponhamos que uma aplicação já existente utiliza a **API B** para aceder à base de dados para executar operações (de leitura ou escrita). Assumamos ainda, para simplificar, que a API B fornece apenas um método (``execute()''), para executar operações na base de dados.

Para poder utilizar o sistema, sem modificar código, é disponibilizada a **API B cliente**. Esta API apresenta a mesma interface que a API utilizada originalmente, mas no lugar de aceder à base de dados directamente, envia o pedido para a *gateway*, que utilizando a API B original faz o pedido à base de dados.

Vejamos então em pormenor como se processa a execução, pela aplicação, de uma operação de leitura (por exemplo,"`select codigo, nome from Alunos`") na base de dados. No lado da aplicação processa-se da seguinte forma:

1. a aplicação invoca o método `execute(``select codigo, nome from Alunos)`;
2. o método `execute()` invoca um método no módulo de dados SQL (onde passa como argumentos o nome da função a executar remotamente, os parâmetros e a máquina e porta onde está a *gateway*) que constrói o pedido que é passado ao Núcleo (*Core*), indicando que é uma operação de leitura, para ser enviado à *gateway*;
3. o Núcleo verifica se a resposta do pedido (contendo o resultado da operação) está em *cache*, e se estiver retorna-o imediatamente ao módulo de dados (indo para o passo 7), caso contrário passa o pedido ao módulo de comunicação;
4. o módulo de comunicação envia o pedido à *gateway* e aguarda pela resposta;
5. ao receber a resposta passa o pedido e a resposta ao Núcleo;
6. o Núcleo passa o pedido e a resposta à *cache*, para esta decidir se mantém a resposta para um possível uso futuro, e de seguida retorna a resposta ao módulo de dados

7. finalmente, o módulo de dados retorna o resultado da operação (contido na resposta) para a aplicação

Entretanto, na *gateway*:

1. o módulo de comunicação da *gateway* recebe o pedido e passa-o ao núcleo;
2. o núcleo passa o pedido ao módulo de dados correspondente, neste caso, o módulo SQL;
3. o módulo SQL descodifica os dados passados no pedido, nomeadamente a função e os parâmetros, e de seguida executa `execute(‘‘select codigo, nome from Alunos)` utilizando a API B, sendo os resultados da execução da função codificados e colocados na mensagem de resposta e retornados ao Núcleo;
4. o Núcleo verifica se a resposta pode ser mantida em *cache* pelo cliente e, em caso afirmativo, regista o pedido realizado de forma a poder notificar o cliente de futuras alterações na resposta;
5. por fim, o Núcleo retorna a mensagem ao módulo de comunicações que a envia ao cliente.

5.3 Núcleo

O **Núcleo** (*Core*) é o módulo que compõe a camada de gestão de dados. Fornece, aos módulos de dados, os serviços de comunicação de dados e de execução remota, e faz a gestão da *cache* e a manutenção da sua coerência.

O Núcleo está dividido (ao nível das funcionalidades) em duas partes, a parte cliente e a parte servidor. A parte cliente é utilizada pelas aplicações, enquanto que a parte servidor pelas *gateways*. Com a combinação das duas partes, juntamente com o módulo de dados, permite obter um *proxy* que realiza a *cache* de dados.

5.3.1 Núcleo Cliente

O Núcleo Cliente tem como função principal a gestão e manutenção dos dados em *cache*. É através do Núcleo cliente que os módulos de dados executam as operações,

através da *gateway*, na fonte de dados. Esta componente é utilizada, indirectamente, pelas aplicações para acederem às fontes de dados.

5.3.1.1 *Cache*

A principal componente do Núcleo Cliente é a *cache*. A *cache* mantém objectos, com as respostas dos pedidos enviados, em memória.

A *cache* mantém os objectos numa tabela indexada. Cada objecto contém a resposta, assim como o pedido que a originou. A forma como a indexação é feita varia consoante o tipo de dados. Cada módulo de dados define quais são os índices, e implementa uma função que dado um objecto retorna uma lista com os vários índices (chaves) desse objecto. Cada entrada na *cache* contém:

- o objecto
- o tamanho do objecto
- uma *flag* indicando a validade do objecto
- o tempo do último acesso
- o tempo da criação da entrada
- o número de acertos e de falhas
- o número de invalidações e de operações de escrita
- o tempo de obtenção do objecto

A *cache* foi desenhada e implementada de forma a eliminar as falhas de colisão. Assim, as únicas falhas possíveis de ocorrer são as falhas compulsórias e as falhas de capacidade.

A coerência da *cache* é assegurada através da invalidação ou actualização dos objectos. Cabe ao Núcleo servidor a tarefa de assegurar a notificação dos clientes das modificações ocorridas na fonte de dados. Cada cliente só deverá ser notificado das alterações que possam afectar os dados que possam estar em *cache*. A notificação dos clientes

é feita através do envio de mensagens do Núcleo servidor para o Núcleo cliente. As mensagens são recebidas pelo Núcleo cliente na porta de controle.

A política de escrita utilizada é uma variante da política *write-through* em que os dados são escritos apenas na fonte de dados. Assim, a utilização de uma *cache* intermédia não altera a forma de execução das operações de escrita.

O Polites só dispõe de uma política de troca, LRU, que funciona da seguinte forma: ao inserir um objecto na *cache*, é chamada uma função que remove objectos menos acedidos da *cache* até que exista espaço suficiente para o novo objecto. A política de gestão da cache tentará optimizar a cache de modo a obter-se a melhor performance possível.

Com a informação disponível em cada entrada na *cache* é possível implementar, praticamente, todas as políticas de troca apresentadas no capítulo 2. Um dos desenvolvimentos a realizar no sistema seria a implementação das várias políticas de troca apresentadas e o estudo dos resultados obtidos em cada uma delas. Também seria interessante implementar a política de troca proposta por Reddy e Fletcher [33], que utiliza a história da vida dos dados como forma de optimizar a gestão da cache. Esta política assenta, essencialmente, no tamanho do objecto, o tempo de obtenção do objecto, o tempo do último acesso, a frequência de modificações, e o número de acertos.

A forma como a *cache* é implementada, mais precisamente a forma como são implementadas as operações de pesquisa e de inserção, afecta a performance do sistema, pelo que houve a preocupação de a tornar bastante eficiente.

A *cache* utiliza uma estrutura de dados que baseada nos conceitos de chave primária, chaves secundárias e índices das bases de dados. A cada entrada na *cache* é associada uma chave primária, e possivelmente chaves secundárias e índices. A chave primária de cada entrada corresponde ao identificador associado a cada operação. A definição das chaves secundárias e índices são da responsabilidade de cada módulo de dados, assim como a computação dos seus valores para cada objecto.

5.3.1.2 Configurações

O Núcleo cliente pode ser configurado em três modos:

- Ponte (*bridge*) - Neste modo o Núcleo cliente não mantém as respostas dos pedidos em *cache*. O sistema é utilizado para efectuar operações na fonte de dados através de um *proxy Polites*, *gateway Polites*, ou numa aplicação que utilize o sistema Polites para partilhar dados.

- *Proxy*[2] - Este modo é utilizado quando o Núcleo cliente faz parte de um *proxy*. Neste caso, a *cache* é mantida em disco com o objectivo de permitir o armazenamento de uma quantidade maior de objectos.

- Normal - Com este modo, o Núcleo cliente permite efectuar operações sobre as fontes de dados e, simultaneamente, manter em *cache* os resultados dessas operações.

5.3.1.3 Funcionamento

O Núcleo cliente faz a gestão da *cache* e providencia os mecanismos necessários para que os módulos de dados possam executar operações num Núcleo servidor. Cada operação é "empacotada", pelo módulo de dados, numa mensagem e enviada, através do módulo de comunicação, para um Núcleo Servidor (que faz parte de uma aplicação, *gateway* ou *proxy*), que se encarregará de a executar.

Antes de enviar a mensagem, o módulo verifica se a resposta da mensagem (contendo o resultado da operação) está na *cache*. Em caso afirmativo, retorna imediatamente a resposta ao módulo de dados. Caso não esteja em *cache*, envia a mensagem e espera pela resposta. Ao receber a resposta decide se a coloca em *cache* ou não, consoante a configuração e o tipo de operação que originou a resposta.

5.3.2 Núcleo servidor

A principal função do Núcleo Servidor é servir os pedidos recebidos dos Núcleo cliente, através do módulo de comunicação. Cabe, também, ao Núcleo Servidor assegurar que os dados mantidos em *cache* pelos Núcleo cliente são mantidos coerentes.

Para assegurar a coerência da *cache* no Núcleo Cliente, o Núcleo Servidor mantém

[2]Por altura da escrita desta tese, este modo de configuração ainda não se encontra completamente implementado.

os pedidos feitos por cada cliente, cujas respostas poderão ser mantidas em *cache*, na tabela DCT. Sempre que ocorre uma operação de escrita, o Core servidor pesquisa a tabela para saber quais são os clientes presumivelmente afectados pela operação, isto é, os clientes que realizaram pedidos de execução de operações cujas respostas recebidas (presumivelmente mantidas em *cache* no cliente) já não são válidas. Cada cliente afectado é notificado com uma mensagem, de invalidação ou actualização, indicando o pedido com a operação que originou a resposta que já não é válida.

5.3.2.1 Tabela de coerência de dados (DCT)

A tabela de coerência de dados - DCT (***Data Coerency Table***) - gere as informações sobre os clientes que realizaram operações cujos resultados possam ser mantidos em *cache*.

Cada cliente é identificado pela sua porta de controle, pelo número IP da máquina onde está a correr, e pelo módulo de dados que utiliza. Para cada cliente são mantidos os pedidos (que as respostas possam ser mantidas em *cache* no cliente), e o tipo de notificação que o cliente deseja que seja utilizada em caso de alteração dos dados - invalidação ou actualização.

É a DCT que, dada uma operação de escrita, indica quais os clientes afectados por uma operação de escrita. Para isso, a DCT é informada pelo Núcleo Servidor de todas as operações realizadas por este, sejam de escrita ou de leitura. As operações de leitura, cujas respostas são susceptíveis de ser mantidas em *cache* pelo cliente, são registadas. Assim, quando ocorre uma operação de escrita os pedidos registados são pesquisados de forma a saber quais as operações de leitura que foram afectadas pela operação de escrita.

A tabela DCT desempenha um papel importante no sistema especialmente ao nível da performance. Sendo extremanente utilizada, uma tabela DCT lenta iria afectar negativamente a performance de todo o sistema. Com o intuito de diminuir os efeitos na performance, a tabela foi implementada de forma permitir a inserção, remoção e pesquisa rápidas de informação, em particular, quais os clientes afectados por uma operação de escrita.

Tal como a componente *Cache*, a DCT utiliza uma estrutura de dados que importa

os conceitos de chave primária, chaves secundárias e índices das bases de dados. Cada registo na DCT contém uma chave primária, e possivelmente chaves secundárias e índices. A chave primária de cada registo é definida pela DCT, e é composta pelo identificador da mensagem que continha a operação, o nome da máquina (ou número IP) de onde a mensagem foi emitida, a porta de notificação do cliente, e o nome do módulo de dados. A definição das chaves secundárias e índices são da responsabilidade de cada módulo de dados, assim como a computação dos seus valores.

Por mais rápida que seja a DCT, a sua utilização aumenta a latência quando uma operação é executada. Através da utilização de *cache* nos clientes, serão feitos menos pedidos de leitura, e por conseguinte, o aumento da latência na DCT poderá ser compensada pela diminuição do tempo obtenção dos dados na fonte de dados.

5.3.2.2 Configurações

O Núcleo Servidor permite dois modos de configuração: modo *proxy* ou modo servidor.

No modo servidor, o Núcleo servidor funciona como a porta de acesso à(s) fonte(s) de dados (locais ou remotas), para que seja possível monitorizar todas as alterações efectuadas sobre os dados, e assim notificar os clientes das alterações.

No modo *proxy*, o Núcleo servidor não acede directamente às fontes de dados para executar as operações. Ao receber uma operação, o Núcleo servidor passa os pedidos a outro *proxy* ou *gateway*, através do Núcleo cliente.

5.3.2.3 Funcionamento

O Núcleo servidor utiliza o módulo de comunicações para ficar à escuta de mensagens numa porta, a porta da comunicação de dados. O módulo de comunicações ao receber uma mensagem, passa-a ao Núcleo servidor que processa a operação contida na mensagem consoante o seu modo de configuração.

Se o Núcleo servidor estiver no modo servidor, a operação recebida é passada ao respectivo módulo de dados, para que a execute e retorne o resultado ao Núcleo servidor. Se não ocorrer nenhum erro na execução da operação, o Núcleo servidor invoca uma função no módulo de dados que retorna as chaves secundárias e índices

da operação executada. No caso de a operação executada ter modificado os dados, as chaves e índices da operação, assim como a mensagem que continha a operação são passados à DCT para que esta retorne todos os clientes, e respectivas operações, afectadas. Os clientes afectados são notificados das operações cujos resultados possam ter sido afectados. No caso da operação ser de leitura (não escrever na fonte de dados) a operação é registada na DCT.

O Núcleo servidor poderá ter para além da porta de comunicação de dados, uma porta controlo. Esta porta de controlo é utilizada pelas fontes de dados (no caso de não se poder assegurar a exclusividade do servidor no acesso à fonte dados) para comunicar alterações dos dados.

No caso de estar configurado no modo *proxy*, a mensagem que contém a operação é passada ao Núcleo cliente para que a envie para o *proxy* seguinte, ou *gateway*, e de seguida retorne a resposta. Uma vez retornada a resposta, através do Núcleo cliente, o Núcleo servidor consoante o tipo de operação e de resposta, regista ou não a operação na DCT.

5.4 Módulos de dados

Para cada tipo de dados (ou fonte de dados), que se pretenda partilhar e/ou aceder através do Polites, deverá existir um módulo de dados. O módulo de dados tem a tarefa de executar as operações de escrita e leitura, remota e localmente, fazer a manipulação dos dados, definir quais as operações que podem ser executadas e quais são os resultados que podem ser mantidos na *cache*.

Para o sistema permitir o acesso a bases de dados, e consequentemente, a *cache* dos resultados das consultas, foi implementado o módulo de dados SQL. Podem ser implementados módulos para outros tipos de dados como variáveis de programas[3], documentos WWW, só para citar alguns. Os módulos de dados não devem ser manipulados directamente pelas aplicações. As aplicações acedem e manipulam os dados através da utilização de API de acesso a dados, que por sua vez utilizam os

[3]Foi implementado um pequeno módulo de dados designado de VSMTCL(*Virtual Shared Memory in TCL*) que permite a aplicações TCL partilhar variáveis e utilizar variáveis partilhadas por outras aplicações.

módulos de dados.

Cada módulo de dados deve definir as chaves primárias, secundárias e índices utilizados pela *cache* e DCT. Cabe ainda aos módulos de dados implementar os métodos que calculem as chaves e os indíces de uma operação. Cada módulo de dados deve providenciar uma interface às API de acesso a dados.

5.4.1 SQL

Para permitir que o Polites seja utilizado para aceder a bases de dados foi implementado o módulo de dados SQL. O módulo de dados SQL permite que as aplicações, através de uma API, executem operações sobre bases de dados. A execução das operações é realizada através da utilização do Núcleo que mantém em *cache* os resultados das operações de consulta.

O módulo de dados SQL está dividido em duas partes: o módulo SQL cliente e o módulo SQL servidor.

Normalmente, as aplicações acedem às bases de dados através da utilização de uma API. A parte cliente do módulo SQL permite que as aplicações executem operações sobre as bases de dados através de uma "cópia" da API nativa ou de uma nova API de acesso às bases de dados.

Os métodos da API de acesso a base de dados são construídos sobre os módulos de dados. Cada método ao ser invocado, passa os seus argumentos e nome a um método do módulo de dados SQL ("sql_execute"). O método "sql_execute" constrói uma mensagem contendo, entre outras informações: o nome da função a invocar, os argumentos e a API nativa a utilizar para aceder à base de dados. A mensagem é enviada ao *proxy* ou *gateway* para que a operação seja executada. O envio da mensagem é feito através da invocação de um método no Núcleo cliente, "Polites_write"ou "Polites_read", consoante a operação é de escrita ou de leitura. São consideradas operações de leitura os "select" e os "show", sendo todas as outras operações consideradas de escrita (como as operações de "update","delete","create", entre outras).

A parte servidor do módulo SQL encarrega-se de executar as operações nas bases de dados. Quando o Núcleo servidor recebe uma mensagem para executar uma

operação do tipo SQL, a mensagem é passada ao módulo SQL para que a operação seja executada. O módulo analisa a mensagem de forma a identificar qual é a API a utilizar para aceder à base de dados. Depois de identificada a API, o módulo utiliza essa API para executar a operação sobre a base de dados. O resultado da execução da operação é encapsulado numa mensagem que é retornada ao Núcleo Servidor.

5.4.1.1 Normalização de Comandos SQL

As operações que se podem efectuar nas bases de dados podem ser escritas de várias formas em SQL. Assim, para se poder verificar se o resultado de uma operação SQL está em *cache* não é suficiente fazer uma comparação das *strings* dos comandos. Por exemplo, a operação de selecção do código e do nome de todos os alunos pode ser representada pelo comando SQL ‘‘`select codigo, nome FROM alunos`’’ ou ‘‘`select nome, codigo from alunos`’’. Uma simples comparação de *strings* indicaria que os comandos são diferentes, apesar dos resultados obtidos serem idênticos.

A normalização consiste em reduzir os comandos SQL a uma forma normal, isto é, reescrever os comandos SQL de forma a permitir a sua comparação. Uma vez normalizado um comando SQL torna-se possível compará-lo com outros comandos SQL normalizados. A comparação permite indicar se o resultado da execução de um comando SQL está contido, contém, é igual ou diferente de outro comando SQL.

A normalização de um comando SQL passa por três fases: 1) Análise léxica; 2) Análise sintáctica; 3) reescrita. Se ocorrer algum erro durante alguma destas fases a versão normalizada do comando SQL é o próprio comando SQL. Existe a possibilidade de ocorrer erro por diversas razões, nomeadamente pela utilização de uma versão do SQL diferente da suportada pelo analisador sintáctico.

Na primeira fase da normalização realiza-se a análise léxica que consiste na transformação do comando SQL numa lista de palavras. Durante a criação da lista, as palavras chave (*keywords*) do SQL encontradas são convertidas para maiúsculas. Se for encontrada alguma palavra que não faça parte da linguagem o processo é abortado.

Numa segunda fase é feita uma análise sintáctica. Durante esta fase é construída uma árvore sintáctica que representa o comando SQL. Na figura 5.2 apresenta como um comando SQL é representado.

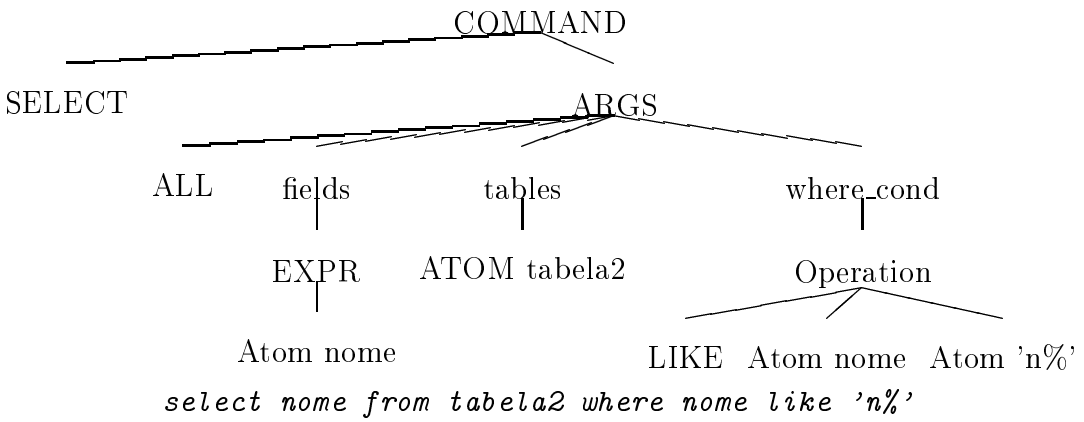


Figura 5.2: Representação de um comando SQL

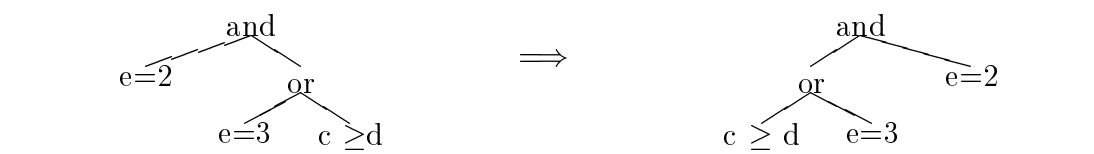
A última fase da normalização consiste na reescrita da árvore obtida na fase anterior. A reescrita da árvore consiste:

- na ordenação lexicográfica dos nomes das colunas e tabelas.

  Exemplo: O comando "select a,d,c from tab2, tab3, tab1" é reescrito em "select a,b,c from tab1, tab2, tab3".

- na ordenação lexicográfica da expressão na condição *where* (apenas no caso da operação ser um *select*). A ordenação é feita através da ordenação lexicográfica do valor das expressões.

  Exemplo:



- eliminar a renomeação de colunas

  Exemplo: O comando SQL ``select codigo as cod, nome from alunos'' é reescrito em ``select codigo, nome from alunos''.

Como já foi referido, podem ocorrer erros durante a normalização de comandos SQL. Estes erros resultam do facto da gramática do SQL implementada ser baseada no SQL-

89. Por exemplo, o caso de uma tabela no "from" ser o resultado de uma consulta (ex: `select a from ( select d,c from d) ...`) não foi implementado.

Foi ainda assumido, de forma a simplificar a implementação, que não são realizadas modificações ou consultas sobre *views*. Isto não impede a realização de consultas ou actualizações sobre uma *view*. No caso de ocorrer, o sistema não garante a coerência dos dados, uma vez que o módulo de dados não utiliza a informação sobre a base de dados (esquema) para saber quais as tabelas realmente manipuladas.

A comparação de comandos SQL, em especial de consultas, é particularmente importante quando se mantém em *cache* os resultados de consultas. A normalização, por si só, não permite a comparação de comandos SQL, sendo por isso foi implementada uma função que compara dois comandos SQL. No caso de ambos os comandos serem consultas, a função pode retornar quatro valores possíveis: "$>$" o resultado do primeiro comando contém o resultado do segundo; "$<$"o resultado do primeiro comando se está contido no segundo; "$=$" os resultados dos dois comandos são idênticos; ou se os resultados são diferentes "$\neq$". Para testar a igualdade de duas consultas, a função começa por verificar se as tabelas utilizadas e as condições são iguais. Em caso negativo, a função retorna que são diferentes. Caso contrário realiza uma comparação sobre o conjunto dos campos seleccionados, retornando se os conjuntos, e logo os resultados dos comandos, são iguais, diferentes, ou que o primeiro conjunto contém ou é contido pelo segundo.

Como o processo de normalização é bastante utilizado, a função de normalização mantém uma pequena *cache* com as últimas normalizações realizadas. Antes de um comando SQL ser normalizado, a *cache* de comandos SQL normalizados é pesquisada. Se ocorrer um acerto é retornada a versão normalizada que estava na *cache*. Caso contrário, o comando é normalizado e a versão normalizada guardada na *cache* dos comandos normalizados.

O trabalho de pesquisa realizado resumiu-se à procura de artigos na Internet, não tendo sido encontrado nada de relevante. No entanto, é de realçar que existe uma tese de mestrado em curso subordinada ao tema "Análise de frases SQL", orientada pelo Professor Doutor José Paulo Leal, que se debruça sobre a normalização de comandos SQL.

Existe ainda muito trabalho a realizar de forma a melhorar o sistema. Na análise

sintáctica o principal trabalho a realizar seria a extenção da gramática, de forma a suportar sub-consultas. Na reescrita devia ser considerada a possibilidade de as tabelas poderem ser vistas (*views*).

5.5 API

Foram implementadas três API de acesso a base de dados. Uma permite o acesso a bases de dados MySQL, outra a bases de dados Oracle, e a outra a ambos os SGBD.

5.5.1 Mysql

A API Mysql oferece, às aplicações, um conjunto de funções para aceder a SGBD Mysql [3].

Existem dois tipos de API: a API Mysql disponível para os programadores, que permite às aplicações aceder a base de dados Mysql através do Polites; e a API nativa, que é a API utilizada pela *gateway* para aceder à base de dados Mysql. Para a comunicação nativa com o SGBD Mysql utilizou-se um *package* TCL - tclsql. Este *package* foi encapsulado num novo *package* TCL designado por p_mysql e cuja API é descrita na tabela 5.1. A aplicação utiliza uma "cópia" desta API para aceder à base de dados Mysql através do Polites.

5.5.2 Oracle

Esta API oferece um conjunto de funções que permitem às aplicações interagir com SGBD Oracle através do Polites.

A API Oracle é idêntica à API Mysql apresentada na subsecção anterior. Para realizar a comunicação nativa com o SGBD utilizou-se um *package* TCL que permite a interacção com bases de dados Oracle - oratcl [5].

Método	***Argumentos***	***Descrição***
init	-	Inicializa a API.
connect	user password database host [port]	Cria uma ligação a uma base de dados. Retorna um *handle*.
close	handle	Fecha a ligação à base de dados associada ao *handle*.
execute	handle sql	Executa uma declaração SQL.
select	handle query	Executa uma consulta à base de dados.
describe	handle table	Retorna informação sobre a tabela.
show_tables	handle	Retorna o nome de todas as tabelas existentes na base de dados.

Tabela 5.1: API Mysql

5.5.3 DBase

O DBase é uma API genérica de acesso a bases de dados optimizada para ser utilizada no Polites, que reflecte os resultados de consultas a bases de dados em estruturas de dados.

O DBase é uma API genérica, no sentido de permitir o acesso a vários tipos de SGBD. O acesso às bases de dados é realizado através das API descritas anteriormente, Oracle e Mysql, que permitem o acesso a SGBD Oracle e MYSQL, respectivamente. O aumento do número de SGBD suportados pelo DBase pode ser realizado, de uma forma relativamente simples, através da utilização de uma API nativa ODBC.

A principal característica do DBase consiste na reflexão dos resultados das consultas em estruturas de dados, e na manutenção da coerência através da utilização do Polites. O DBase permite modificar, apagar e criar dados na base de dados sem a utilização de SQL, através da realização das operações sobre as estruturas de dados.

A API providencia diversos métodos, sendo os principais descritos na tabela 5.2

Método	***Argumentos***	***Descrição***
inicia	-	Inicia a ligação à base de dados
reflecte	consulta [campo1 campo2 ...]	Reflecte o resultado de uma consulta numa estrutura de dados, usando como chaves o campo1, campo2, ... Retorna uma referência (ref_estr) para a estrutura criada.
apaga_refl	ref_estr	Liberta uma reflexão existente.
chaves	ref_estr	Enumera as chaves da estrutura.
valor	registo campo [valor]	Retorna o valor de um campo de um registo. Se o 'valor' estiver definido, atribui o valor ao campo no registo, e propaga a alteração à base de dados.
mapeia	ref_estr campos fun	Executa a função 'fun' para cada registo da estrutura 'ref_estr'. Ao invocar a função 'fun' são criadas variáveis globais com os valores dos 'campos'.
enumera	ref_estr	Retorna uma lista com as referências de todos os registos da estrutura 'ref_str'.
cria	ref_estr campos valores	Cria um novo registo na estrutura de dados, e propaga-o à base de dados. Retorna o resultado da execução da operação na base de dados (sucesso ou falha).

Tabela 5.2: API DBase

5.6 Comunicação

O módulo de comunicação providencia os mecanismos de envio e recepção de mensagens, e gere a porta de comunicação de dados e a porta de controle.

Podemos classificar a comunicação do sistema (ver figura 5.1) em comunicação de dados, comunicação de controle e comunicação nativa.

A *comunicação de dados* realiza-se entre o Núcleo cliente e o Núcleo servidor. O Núcleo cliente faz um pedido e o Núcleo servidor executa-o, retornando o estado da sua execução e se houver, o resultado da execução.

A *comunicação de controle* é usada para fazer invalidação dos dados ou dar informação extra. O núcleo servidor notifica os núcleos clientes que fizeram pedidos, que os dados de um determinado resultado já não são válidos. Em particular, se a fonte de dados for uma base de dados, e esta não tiver uma *gateway* Polites que funcione como única porta de acesso, torna-se necessário implementar no SGBD o mecanismo de controle. Isto pode ser feito através da criação de *triggers* que comunicam à *gateway* as alterações realizadas na base de dados.

A *comunicação nativa* é a comunicação usada pelo módulo de dados à fonte de dados. No caso do módulo de dados SQL, a comunicação poderá ser feita utilizando um dos protocolos genéricos para aceder a bases de dados, como o ODBC ou JDBC, ou protocolos específicos do SGBD.

O protocolo utilizado pelo Polites é uma extensão do protocolo HTTP [26](***H****ypertext* ***T****ransfer* ***P****rotocol*). O HTTP é um protocolo genérico e sem estado. É um protocolo bastante utilizado nomeadamente na WWW, onde é o protocolo de comunicação standard utilizado pelos *browsers* para comunicarem com os servidores de HTTP. A principal motivação da utilização do protocolo HTTP foi a possibilidade do Polites poder ser utilizado atrás de *firewalls* usando os túneis e *proxies* já existentes para o protocolo HTTP.

O Polites extendeu o protocolo HTTP através da adição de seis novos métodos: INIT, END, WRITE, READ, UPDATE e INVALIDATE. Os primeiros quatro são utilizados na comunicação de dados, enquanto que a últimos dois são utilizados na comunicação de controle.

Os métodos INIT e END são utilizados pelos núcleos cliente para iniciar e terminar sessões em núcleos servidores.

O método **WRITE** é utilizado nas mensagens com operações que os resultados não serão mantidos em *cache* pelo núcleo cliente.

O método **READ** é utilizado nas mensagens que contêm operações cujos resultados podem ser mantidos em *cache* pelo núcleo cliente.

O método **INVALIDATE** é utilizado pelos núcleo servidores para notificar os clientes que os resultados de uma dada operação, possivelmente mantidos em *cache* pelo cliente, já não são válidos.

O método **UPDATE** é em tudo idêntico ao método INVALIDATE, excepto na diferença da mensagem do UPDATE conter o resultado actualizado da operação executada pelo cliente.

5.7 Implementação

A implementação do Polites foi feita utilizando a linguagem TCL [32, 8]. Esta é uma linguagem interpretada de alto nível, relativamente rápida e de aprendizagem fácil.

Todo o código do Polites foi implementado de raiz e ronda actualmente as 10000 linhas de código. Foram utilizadas dois *packages tcl*, tclsql e Oratcl [5], para realizar o acesso nativo às bases de dados MySQL e Oracle,respectivamente.

Capítulo 6

Experiências e Resultados

No capítulo anterior foi descrito o sistema Polites, cujo principal objectivo consiste na optimização das comunicações entre aplicações e fontes de dados (bases de dados em particular).

Neste capítulo faz-se uma avaliação dos resultados obtidos na utilização do Polites, no sentido de aferir os efeitos da sua utilização pelas aplicações. A avaliação é através da análise de dois parâmetros: *speedup* e o ganho obtido na transferência de bytes (*transf*).

6.1 Experiências

As experiências realizadas testaram o modelo com uma *gateway*, isto é, as aplicações acedem à base de dados através de uma *gateway*. Na tentativa de testar um leque mais variado de situações (velocidade de rede diferentes, tipos de aplicação, protocolo de manutenção de coerência utilizado) a execução das aplicações foi feita através de simulação, tendo sido executadas 198 experiências (66 para cada aplicação). Também a execução das operações na base de dados foi simulada na *gateway*, de forma a obter resultados independentes do SGBD.

Simolou-se a execução de três aplicações (C1, C2, C3) que se distinguem pelas operações SQL realizadas e pela frequência com que cada uma delas é executada. A simulação de cada aplicação consistia na execução de 100 operações SQL. A aplicação C1 executa operações (de escrita e de leitura) sobre uma única tabela. A aplicação C2 executa um conjunto de operações sobre duas tabelas, em que uma das tabelas é muito pouco modificada. A aplicação C3 executa um número mais vasto de operações de consulta.

Cada aplicação foi simulada variando três parâmetros:

- **Velocidade da rede**
  Este parâmetro define a velocidade da rede por onde é realizada a comunicação entre a aplicação e a fonte de dados. Foram definidos três tipos de rede: rede lenta (2 KB/s); rede média (500 KB/s); e rede rápida (4 MB/s).

- **Distribuição dos tipos de operações executadas**
  Este parâmetro define qual a distribuição dos tipos de operação, isto é, qual é a percentagem das operações SQL a executar por cada aplicação que são de leitura e de escrita. De certa forma, este parâmetro indica qual é a probabilidade de os dados serem alterados na fonte de dados. Cada uma das aplicações é executada com diversas distribuições para a percentagem do número de operações de escrita e leitura. Mais precisamente, cada aplicação executa x por cento (x=0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100) de operações de leitura. Isto é equivalente a dizer que os dados têm $(1 - x)\%$ de probabilidade de serem alterados.

- **Protocolo de manutenção de coerência**
  Este parâmetro indica qual é o protocolo de manutenção da coerência utilizado

pelo Polites. Foram considerados os dois protocolos implementados no Polites - invalidação e actualização.

É ainda importante realçar que o tamanho da *cache* estava configurado para 5MB. Isto implica, que dado o tamanho das operações, a *cache* não encheu, não sendo assim necessário avaliar a política de troca implementada.

O anexo B contém uma descrição das operações SQL e os valores dos parâmetros utilizados na simulação para cada aplicação.

6.2 Análise de resultados

Foram utilizadas duas métricas (descritas em 3.1.7) para avaliar o Polites: o *speedup*; e o ganho obtido na transferência de bytes (*transf*).

6.2.1 *Speedup*

Na figura 6.1 é apresentado o *speedup* das três aplicações ao utilizar o protocolo de invalidação nos três tipos de rede - lenta, média, rápida. Podemos observar que à medida que a velocidade da rede aumenta o *speedup* obtido pelas aplicações diminui, uma vez que o peso da comunicação diminuiu. Mesmo assim, sempre que a percentagem de operações de leitura ultrapassa os 50%, obtêm-se ganhos na utilização do Polites. É de realçar ainda que as aplicações ficam pelo menos 2x mais rápidas no acesso aos dados sempre que a percentagem de operações de leitura é superior aos 70%.

A figura 6.2 apresenta o *speedup* obtido pelas aplicações ao utilizarem o protocolo de actualização para manter a coerência da *cache*. Os resultados obtidos na rede lenta são bastante bons, com as aplicações a atingirem um *speedup* superior a 60% sempre que a taxa de alterações dos resultados é inferior a 80%, isto é, sempre que a percentagem de operações de leitura é superior a 20%. No entanto, com redes mais rápidas, é necessário que a taxa de alteração dos dados seja inferior a 20%.

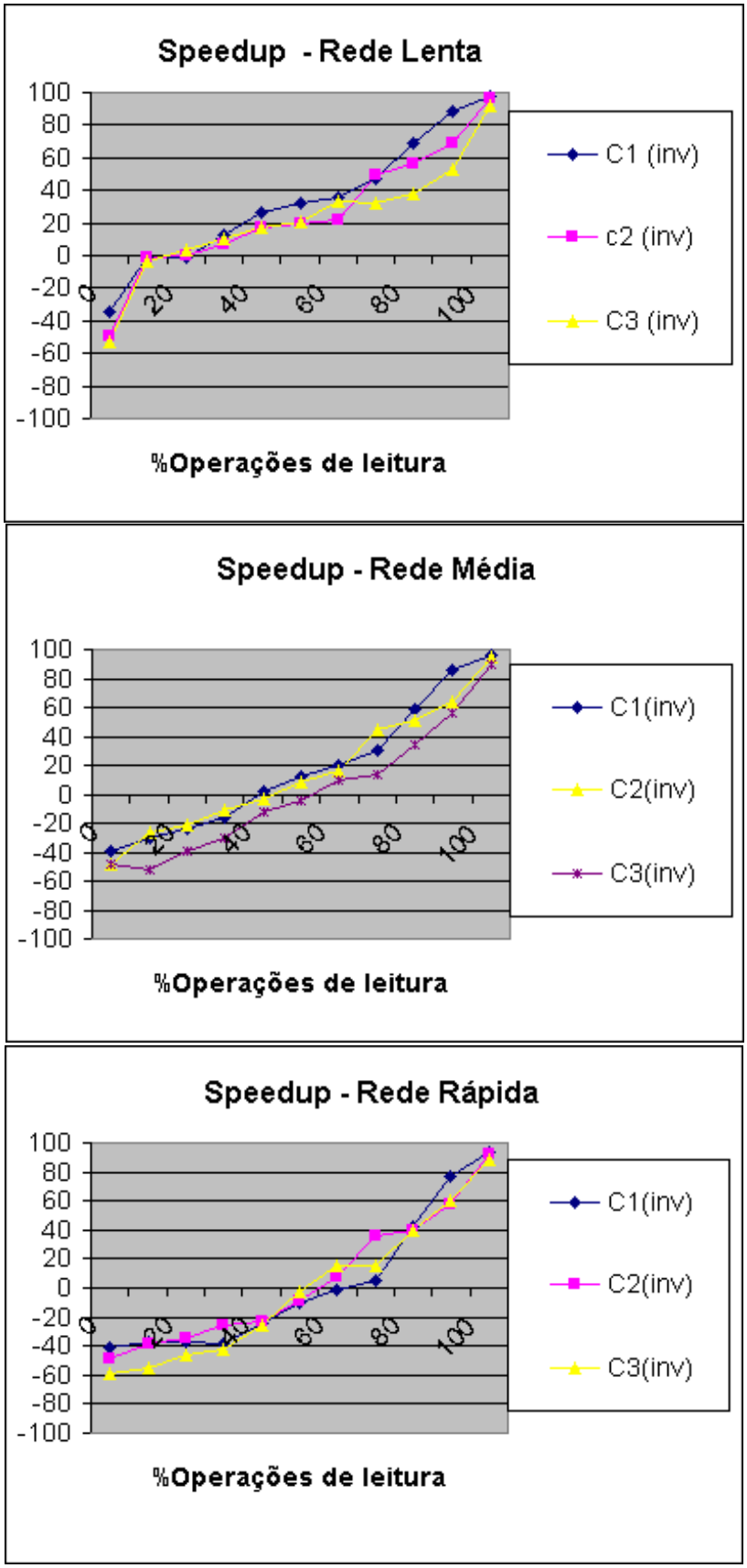


Figura 6.1: Speedup obtido com o protocolo de invalidação

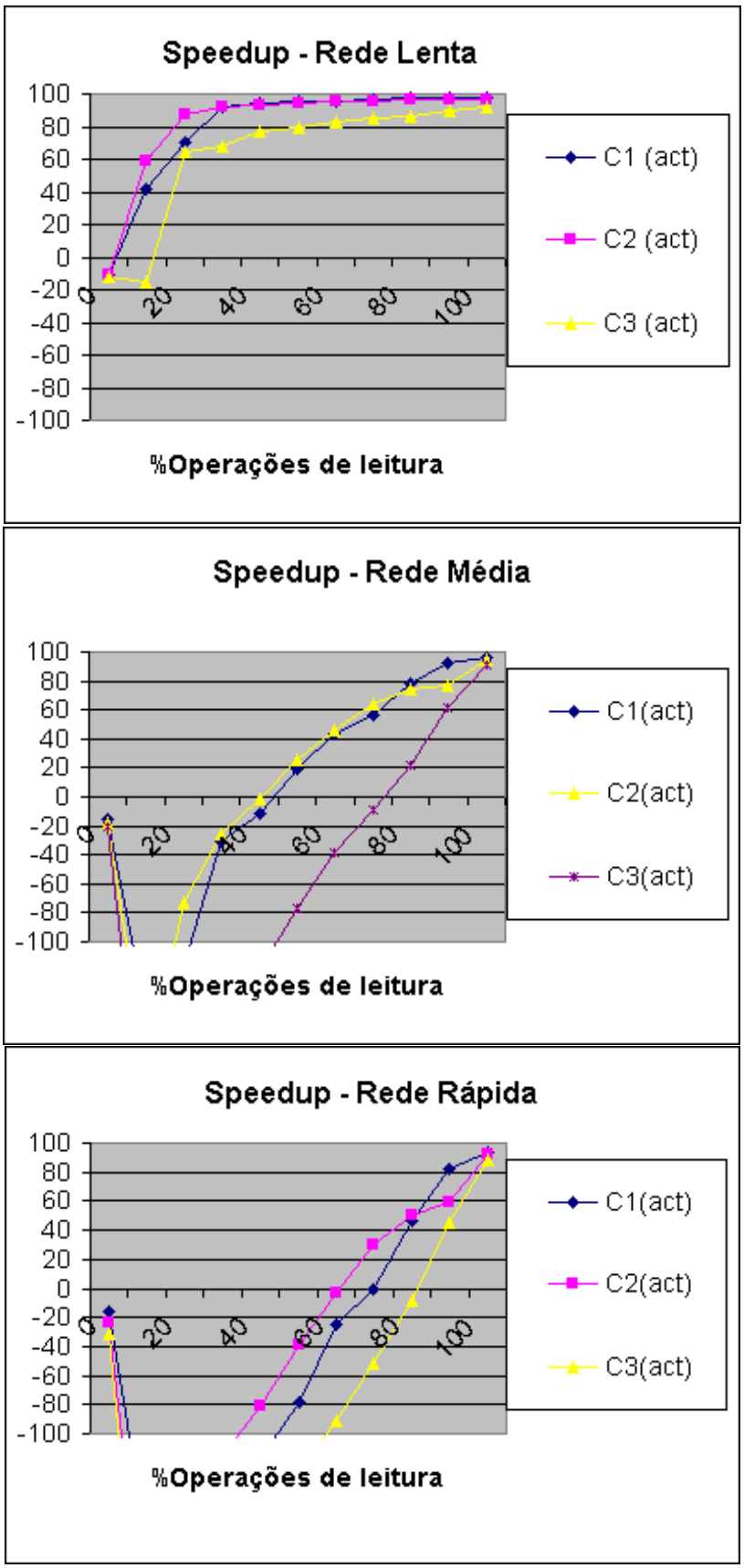


Figura 6.2: Speedup obtido com o protocolo de actualização

Pelos resultados obtidos, a utilização do protocolo de actualização permite atingir bons *speedups* sempre que a rede é lenta, mesmo que a percentagem de operações de escrita seja elevado. No entanto, o protocolo de invalidação oferece um comportamento mais regular nos vários tipos de rede.

6.2.2 *Ganho obtido na transferência de bytes*

Na figura 6.3 apresenta-se o ganho obtido na transferência de bytes, para cada uma das aplicações, com a utilização do protocolo de invalidação. Como seria de esperar, o ganho obtido na transferência de bytes não é afectado pela velocidade da rede. No entanto, observa-se através da comparação da 6.3 com 6.4 que a utilização do protocolo de actualização obtém resultados muito negativos quando a percentagem de operações de leitura é inferior a 60%.

O protocolo de invalidação permite melhores resultados, conseguindo uma diminuição do número de bytes transferidos quando a percentagem de operações de leitura é superior a 30%. É ainda de realçar que quando o número de operações de leitura é superior a 40% são transferidos menos de metade dos bytes do que os transferidos sem a utilização do Polites.

6.2.3 *Considerações Finais*

A utilização do Polites garante a diminuição do tempo de latência das aplicações para metade e do número de bytes transferidos sempre que a percentagem de operações de leitura é superior a 70%, independentemente da velocidade da rede e do protocolo de manutenção da coerência de dados escolhido.

O protocolo de invalidação apresenta bons resultados, quer ao nível do *speedup*, que ao nível do ganho obtido na transferência de bytes, mesmo com percentagens elevadas (70%) de alteração dos dados.

O protocolo de actualização permite a obtenção de *speedups* excelentes em redes lentas. Por outro lado, para redes mais rápidas e do ponto de vista do ganho obtido na transferência de bytes, o protocolo de actualização não deve ser utilizado excepto se o número de operações de leitura for superior a 70%.

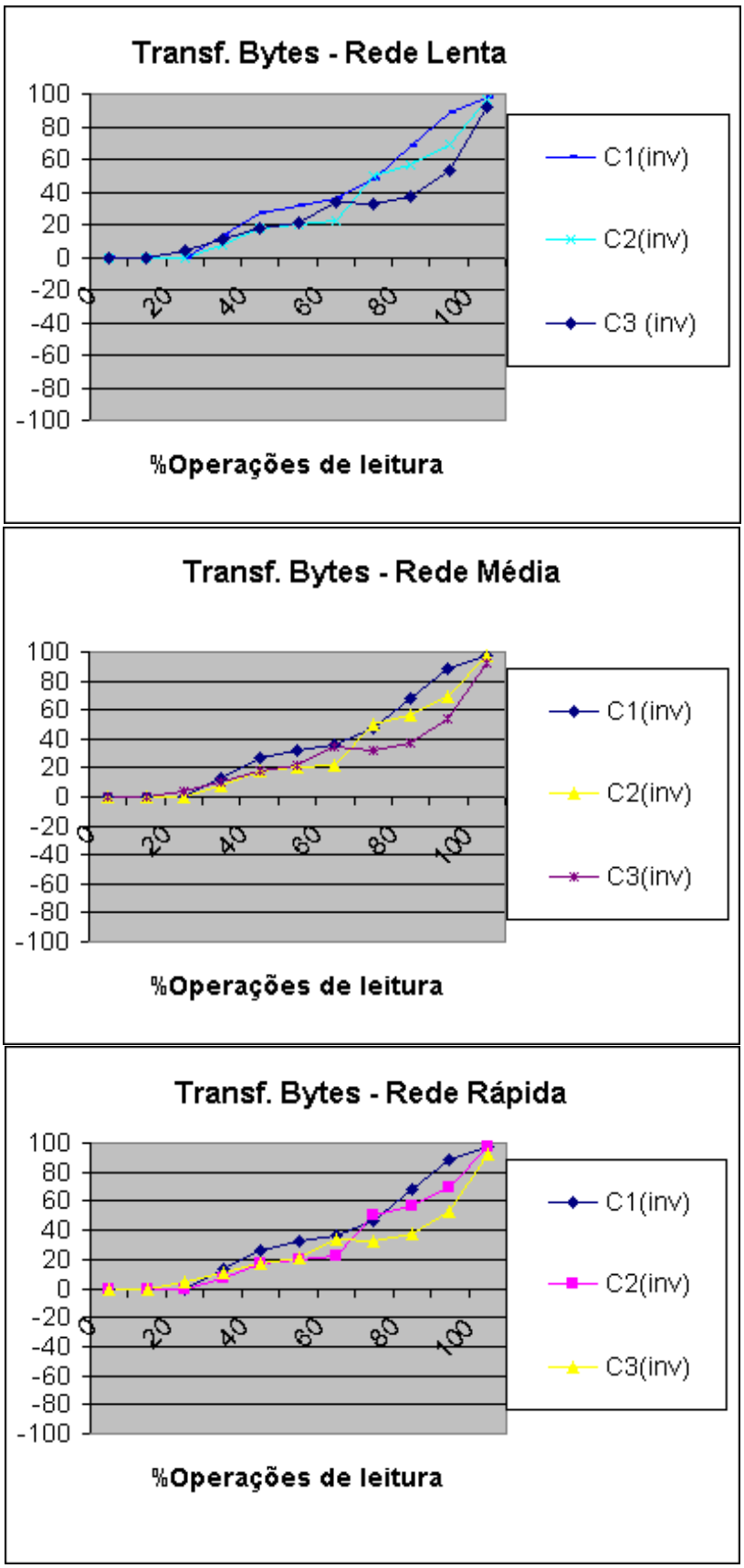


Figura 6.3: Ganho obtido na transf. de bytes com a utilização do protocolo de invalidação

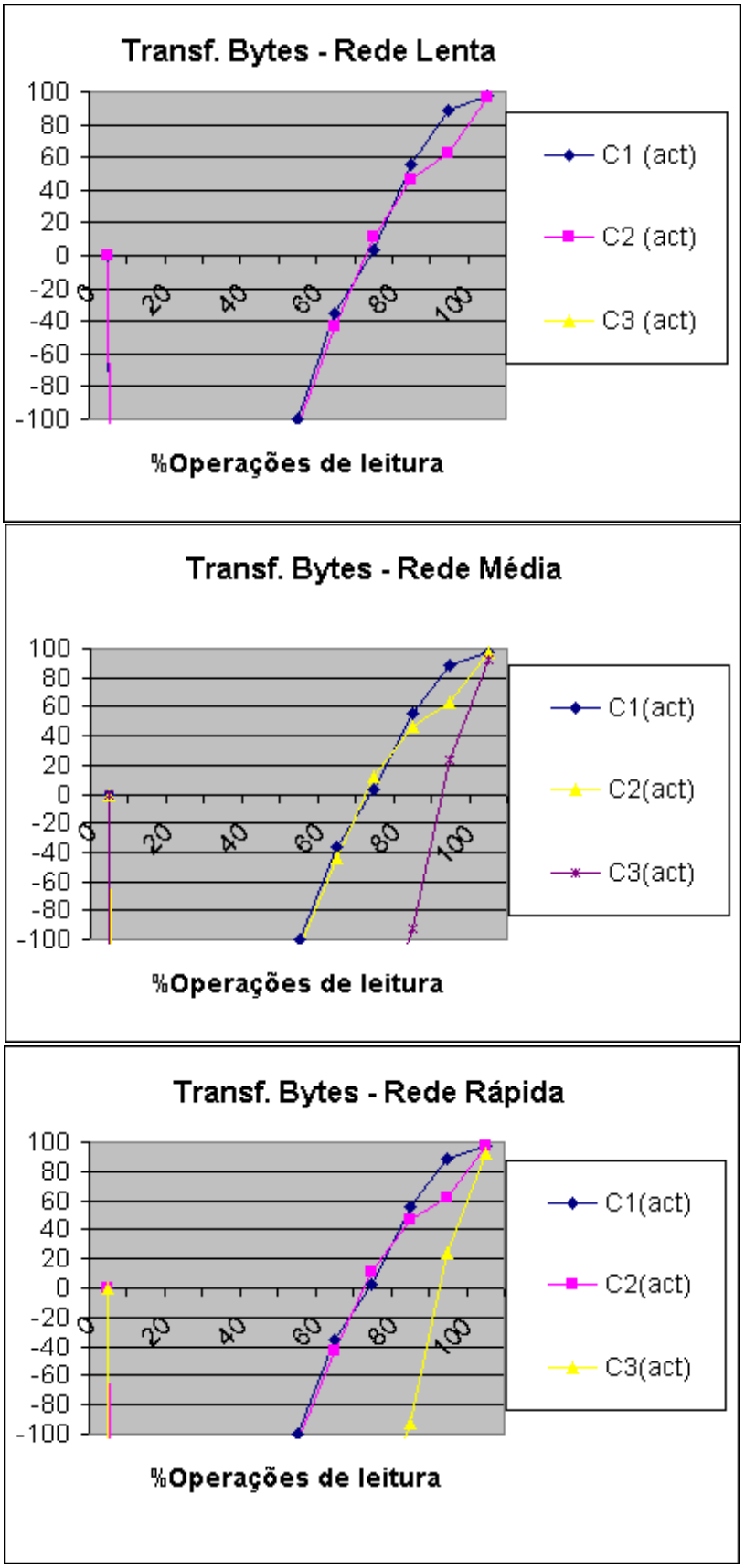


Figura 6.4: Ganho obtido na transf. de bytes com a utilização do protocolo de actualização

Capítulo 7

Conclusões

Na proposta da tese estavam dois objectivos principais:

1. o desenvolvimento de uma API de acesso a bases de dados que mapeie a informação da base de dados em estruturas de dados na linguagem TCL
2. a optimização das comunicações entre aplicações e bases de dados

Para realizar o primeiro objectivo desenvolveu-se uma API de acesso a base de dados, designada por DBase, que: reflecte automaticamente os resultados das consultas a bases de dados em estruturas de dados; permite a modificação dos dados na estrutura de dados criada e propaga as alterações feitas localmente na aplicação na base de dados; e assegura que os dados acedidos na estrutura pelas aplicações estão coerente com os dados na base de dados.

O segundo objectivo foi realizado através do desenvolvimento do sistema Polites que explora a ideia de manter em *cache* os resultados das consultas efectuadas a bases de dados. Com esta ideia pretendeu-se optimizar as comunicações entre aplicações e bases de dados através da diminuição do número de bytes transferidos e da redução da latência.

Nesta tese foi proposto um modelo que permite optimizar as comunicações entre aplicações e fontes de dados (bases de dados em particular) que se baseia na *cache* dos resultados das operações executadas na fonte de dados. Este modelo distingue-se dos existentes por assegurar a coerência da *cache* utilizando mecanismos como o

protocolo de invalidação ou actualização imediata.

Tendo como base o modelo proposto, foi implementado o sistema Polites. O sistema foi testado de forma a avaliar o seu efeito da sua utilização: nas aplicações, ao nível da diminuição da latência; e na rede, ao nível do número de bytes transferidos.

Fez-se uma avaliação exaustiva do sistema através de simulação e os testes realizados indicam que as aplicações obtêm um *speedup* superior a 50% quando a frequência de alteração dos dados na fonte de dados é inferior a 30%. Também o tráfego entre as aplicações é reduzido para metade nestas condições.

O sistema apresenta ainda ganhos significativos, ao nível da redução do do número de bytes transferidos, quando utiliza o protocolo de invalidação e a frequência de alteração dos dados é inferior a 70%.

Em situações em que a frequência de alteração dos dados é superior a 70% a utilização do sistema não traz vantagens, originando até uma maior transferência de tráfego e um aumento no tempo de latência.

Trabalho Futuro

O Polites não é ainda um sistema completo, pelo que existe ainda algum trabalho a realizar na finalização da implementação do modelo, adição de novas funcionalidades e melhoria das funcionalidades já existentes.

Do ponto de vista do modelo, falta terminar o desenvolvimento do *proxy* e iniciar o desenvolvimento das *bridges*. Estas tarefas são relativamente simples de completar e realizar dada à forma como o Polites foi estruturado e implementado.

Um outro trabalho interessante a realizar seria a extensão das capacidades de normalização para permitir a normalização de:

- consultas com sub-consultas
- *views*
- expressões aritméticas.

A avaliação realizada no sistema incidiu sobre o *speedup*, obtido pelas aplicações, e sobre os efeitos na rede, ao nível do número de bytes transferidos. Assim, uma vez que não foram analisados os efeitos da política de troca seria interessante implementar, analizar e comparar as várias políticas de troca apresentadas em 3.1.4.

Referências

[1] Dynamic Data Caching. http://www.timesten.com/.

[2] Java blend^{tm} White Paper. http://www.sun.com/software/javablend/.

[3] Mysql. http://web.mysql.com/.

[4] Netdynamics application server platform. http://www.netdynamics.com.

[5] Oratcl. http://www.nyx.net/~tpoindex/tcl.html.

[6] An overview of dbkona architecture. WebLogic documentation. http://www.ewblogic.com/docs/techoverview/db.html.

[7] Remote Data Service. Microsoft Documentation. http://www.microsoft.com/data/ado/rds/.

[8] TCL/TK Documentation. http://dev.scriptics.com/doc/.

[9] Database Language - Embedded SQL. (X3.168-1989). American National Standards Institute, 1994. Technical Committee X3H2.

[10] Microsoft open database connectivity software development kit version 2.0. Microsoft Press, 1994. ISBN 1-55615-658-8.

[11] Understanding Remote Data Service Applications. Microsoft Documentation, 1998. http://www.microsoft.com/data/ado/adords15/docs/adcdg01_1.htm.

[12] IS 9073. International standard for database language SQL - part 3: Call level interface. Document ISO/IEC 9075-3:1995. equivalent to the American National Standard ANSI/ISO/IEC 9075-3:1995.

[13] Marc Abrams, Charles R.Standridge, and et al. Caching Proxies: Limitions and Potentials. In *Proc. Fourth International Conference WWW*, pages 312–319. O'Reilly & Associates, 1995.

[14] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of web cache replacement policies. Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, 1999.

[15] Calvin Austin and Monica Pawlan. *Writing Advanced Applications for the Java^{tm} Platform*, chapter 8 - Connection Pooling. Java Documentation, 2000. http://developer.java.sun/developer/onlineTraining/Programming/JDCBook/.

[16] Hans Bergsten. Improved performance with connection pool. http://webdevelopersjournal.com/columns/connection_pool.html.

[17] Azer Bestavros. WWW Traffic Reduction and Load Balancing through Server-Based Caching. *IEEE Concurrency*, pages 56–67, January-March 1997.

[18] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 193–206, December 1997.

[19] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the World-Wide Web. In *Workshop on Internet Server Performance(WISP'98)*, June 1998.

[20] Chungmin M. Chen and Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proceedings of the 4th International Conference on Extending Database Technology*, pages 323–336, 1994.

[21] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[22] David E Culler, Jaswinder Pal Singh, and Anoop Gupta, editors. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1998.

[23] Brian D. Davison. A survey of proxy cache evaluation techniques. In *Proceedings of the 4th International Web Caching Workshop*, April 1999. http://www.ircache.net/Cache/Workshop99/Papers/davison2-final.ps.gz.

[24] A. Delis and N. Roussopoulos. Evaluation of an enchanced workstation-server dbms architecture. In *Proceedings of the 18th VLDB*, 1992.

[25] A.K. Dewdney. *The Turing Omnibus*, chapter 40 - Storage by hashing, pages 264–270. Computer Science Press, 1989.

[26] R. Fielding, J. Gettys, J. Mogul, H Frystyk, and T. Berners-Lee. RFC 2068 - Hypertext Transfer Protocol – HTTP1.1. http://www.w3c.org/Protocols, January 1997.

[27] Kai Hwang. *Advanced computer architecture: parallelism, scalability, programmability*, chapter 4, page 192. McGraw-Hill, 1993.

[28] P. Lorenzetti and L. Rizzo. Replacement policies for a proxy cache. Technical report, Universita di Pisa, December 1996.

[29] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data SIGMOD'93*, pages 297–306, Washington, DC, 1993.

[30] Elizabeth J. O'Neil and Patrick E. O'Neil. An optimality proof of the LRU-K page replacement algorithm. *Journal of the ACM*, January 1999.

[31] Robert Orfali, Dan Harkey, and Jeri Edwards. *Client/Server Survival Guide*. John Wiley & Sons, Inc, third edition, 1999.

[32] John Ousterhout. *TCL and TK Toolkit*. Addison-Wesley, 1994.

[33] Mike Reddy and Graham P. Fletcher. An adaptative mechanism for web browser cache management. *IEEE Internet Computing*, pages 78–81, January-February 1998.

[34] J. Robinson and M. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS Confererence on the Measurement and Modeling of Computer Systems*, pages 134–142, Bouler, CO, May 1990.

[35] Dave Rosenberg. Bringing java to the enterprise: Oracle on its java server strategy. *IEEE Internet Computing*, March-April 1998.

[36] Prasan Roy, Jinesh Vora, Krithi Ramamritham, S. Seshadri, and S.Sudarshan. Query Result Caching in Data Warehouses and Data Marts. Technical report, Indian Institute of Technology, Bombay, April 1999.

[37] Peter Scheuermann, Junho Shim, and Radek Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of the 22nd VLDB Conference*, Mumbay(Bombay), India, 1996.

[38] Curt Schimmel. *Unix Systems for Modern Architectures: symmetric multiprocessing and caching for kernel programmers.* Addison-Wesley, 1994.

[39] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*, chapter 9, pages 312–325. Addison-Wesley, fourth edition, 1994.

[40] Sun Microsystems Inc. JDBC - U*niversal* D*atabase* C*onnectivity For the Enterprise*. The Source For JAVA Technology(java.sun.com).

[41] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, 1987.

[42] M. Venkatrao and M. Pizzo. SQL/CLI - An new binding style for SQL. *SIGMOD Record*, 24(4), December 1995.

[43] Seth White and Mark Hapner. JDBC 2.0 API. Sun Microsystems Inc., May 1998.

[44] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox. Removal policies in network caches for worl-wide web documents. In *Proceedings on ACM SIGCOMM '96*, pages 293–305, August 1996.

Apêndice A

Resultados

Neste apêndice são apresentados os gráficos com resultados obtidos por cada aplicação utilizando o protocolo de actualização ou de invalidação, nos diversos tipos de rede.

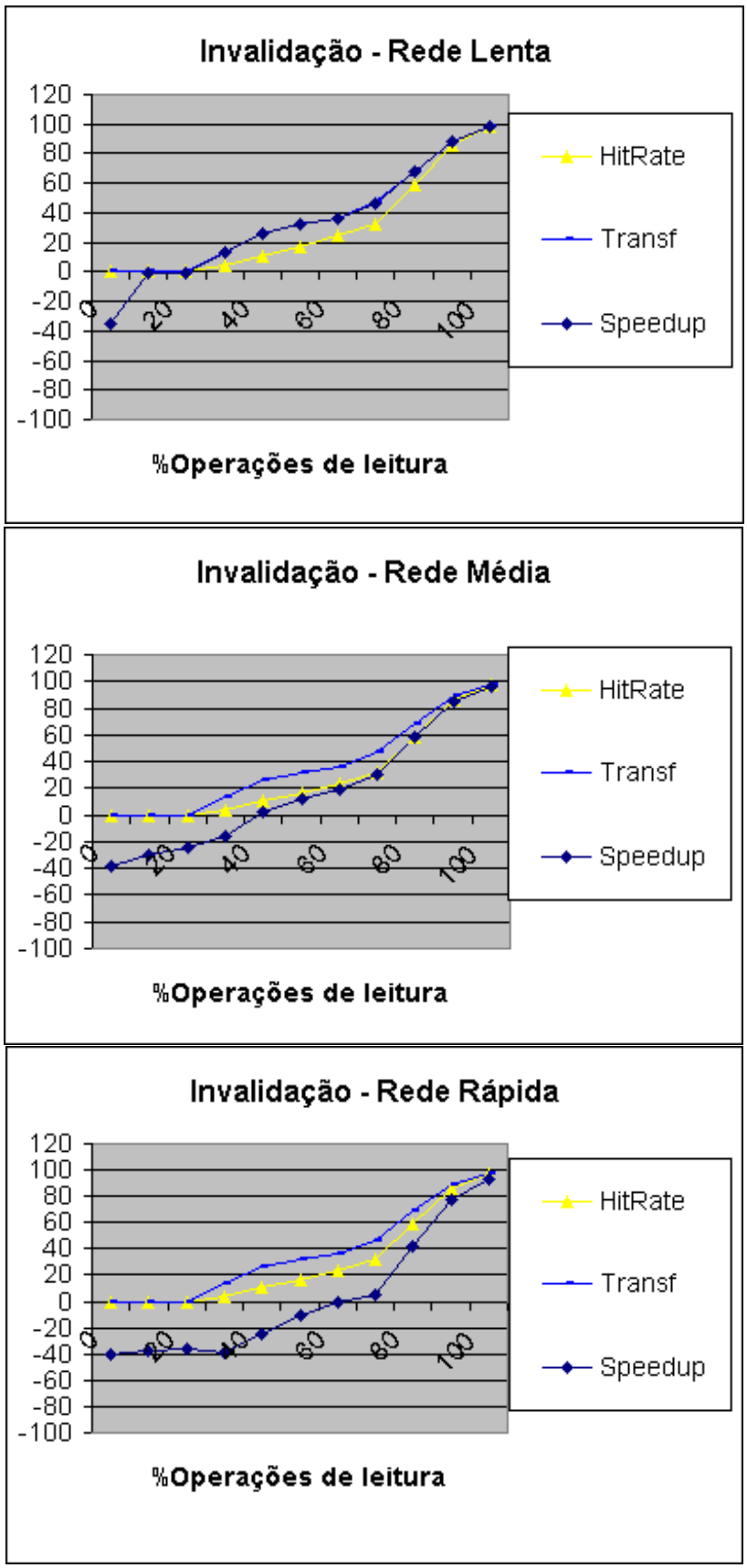


Figura A.1: Resultados obtidos pela aplicação C1

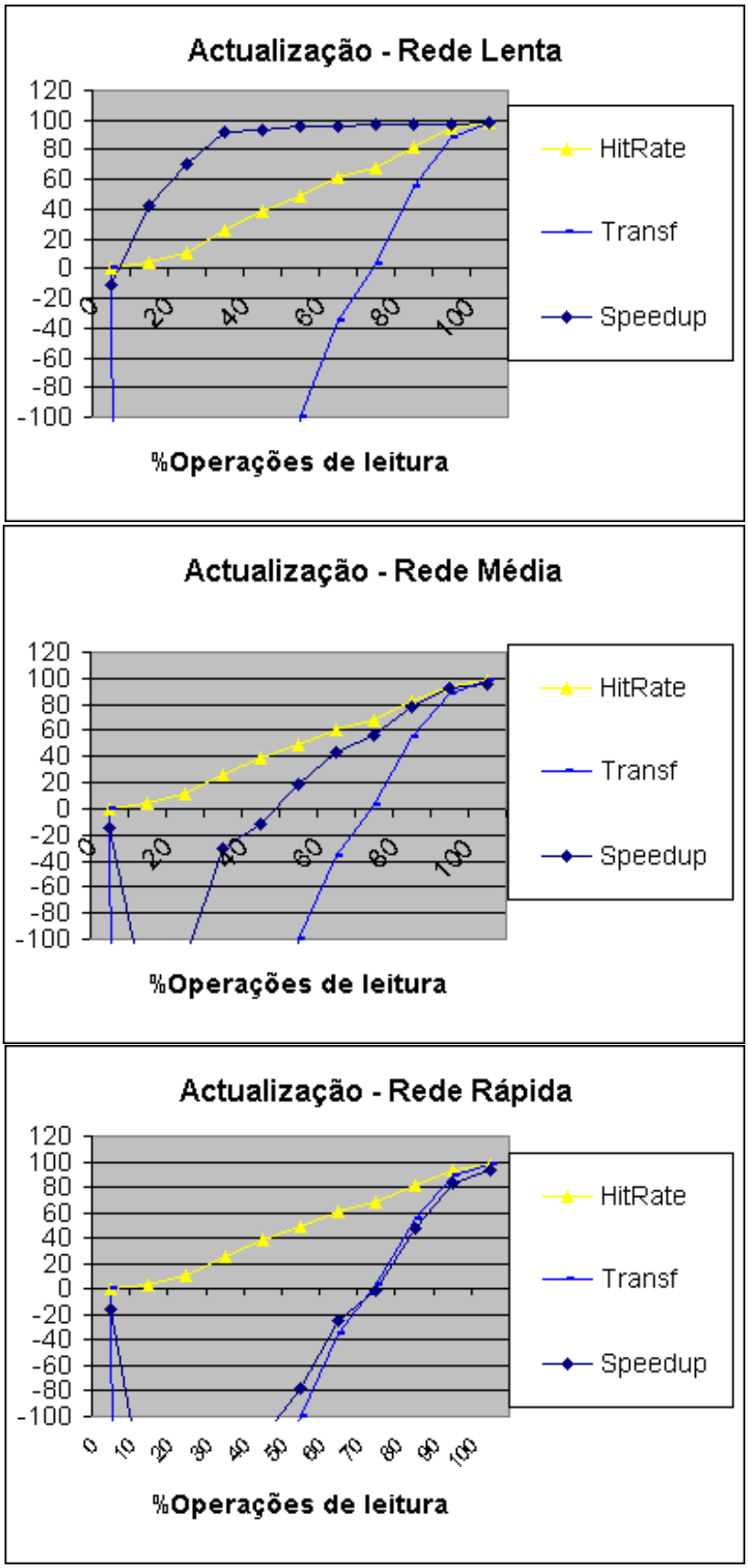


Figura A.2: Resultados obtidos pela aplicação C1

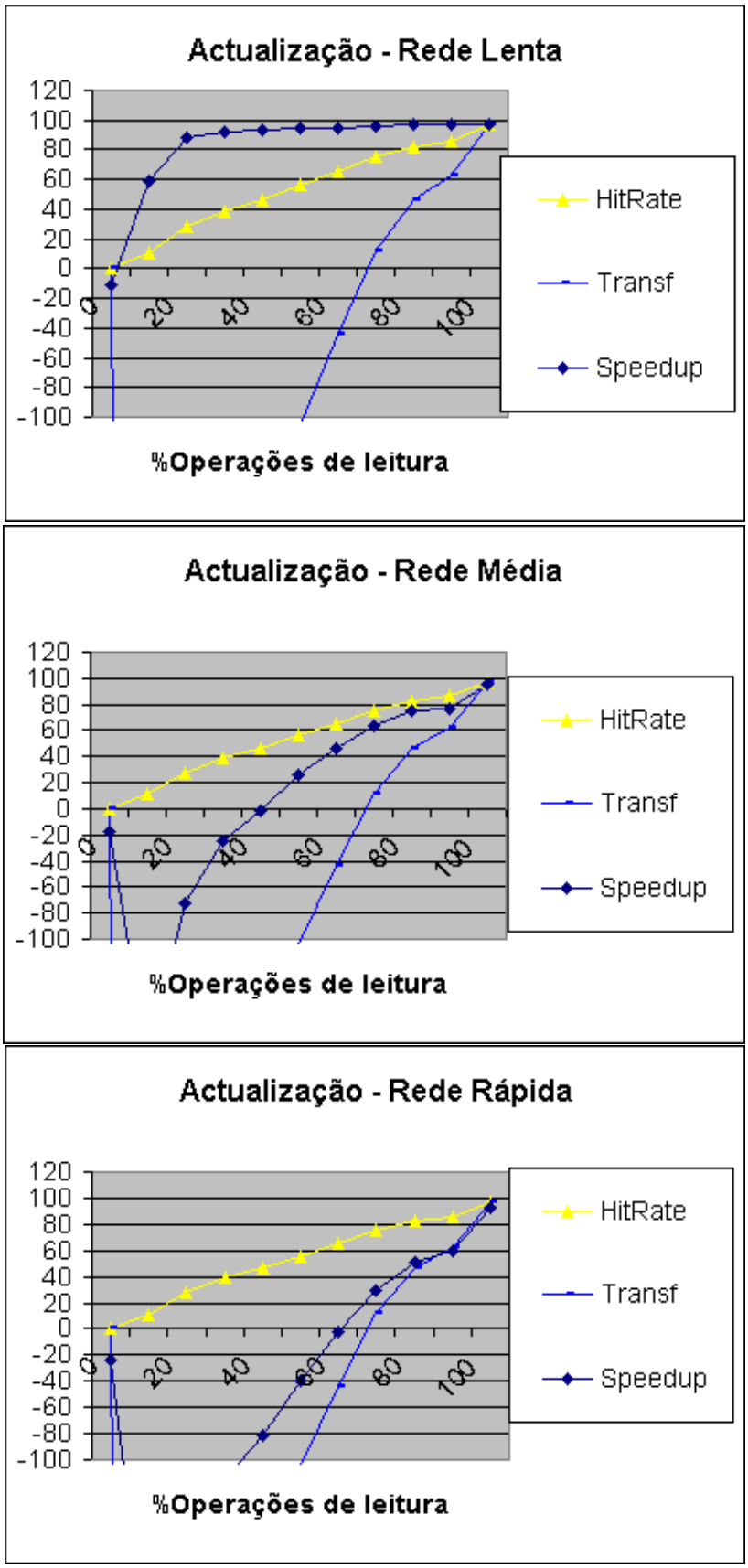


Figura A.3: Resultados obtidos pela aplicação C2

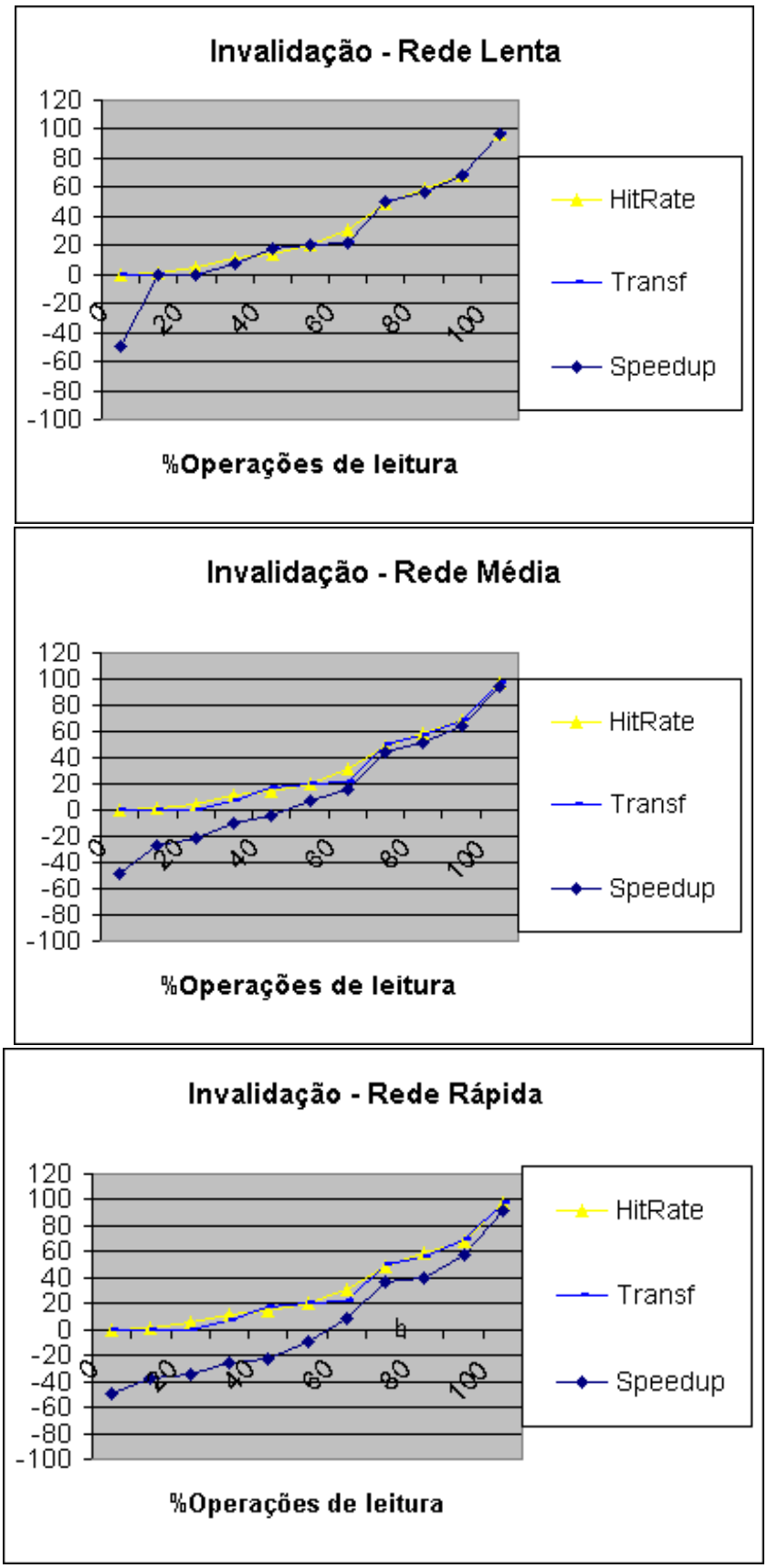


Figura A.4: Resultados obtidos pela aplicação C2

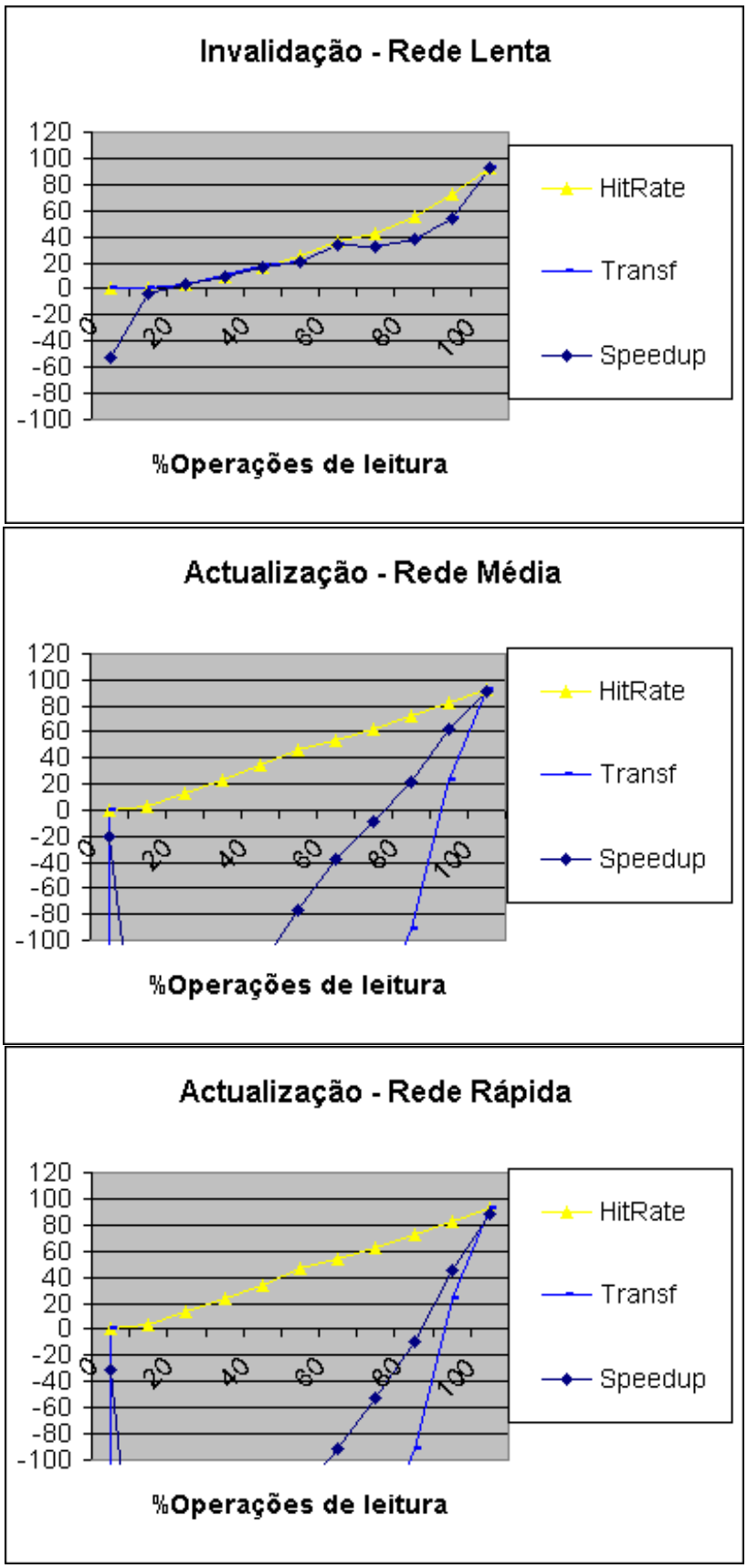


Figura A.5: Resultados obtidos pela aplicação C3

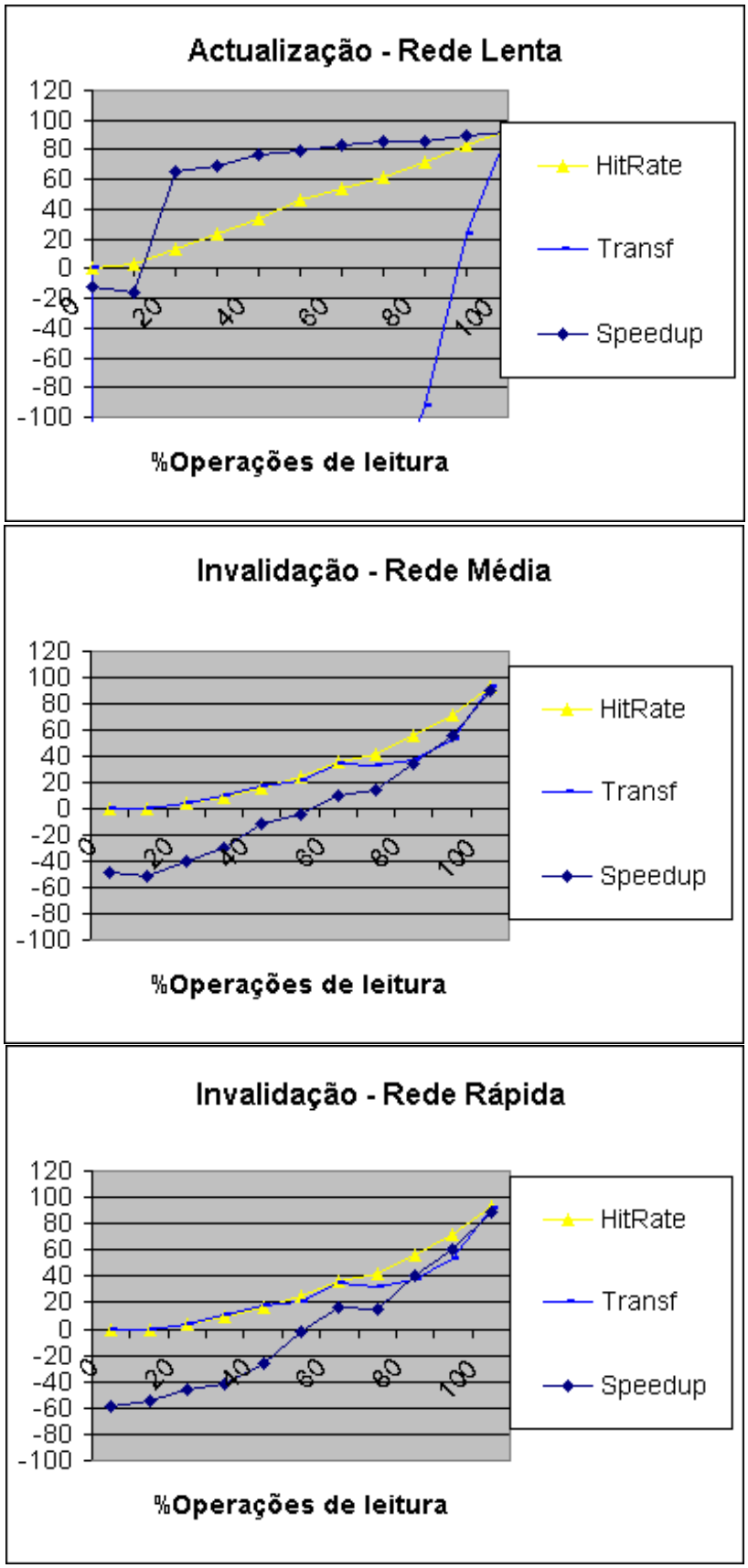


Figura A.6: Resultados obtidos pela aplicação C3

Apêndice B

Parâmetros da Simulação

Código	***Comando SQL***
q1	select nome,bi,ncontrib,morada,datanasc, fotografia from Alunos
q2	select ncd, nome, bi, morada, datanasc from Alunos where nome like 'N%'
q3	select ncd, nome, bi from Alunos where nome like 'N%'
q4	select codigo, titulo, enunciado, solucao, tests, dominio, cod_estado from exercicios
q5	select ncd,bi_no,nome from alunos
q6	select alunos.ncd,alunos.nome from insc,Alunos where insc.cadeira=1999071107 and insc.ncd=alunos.ncd order by alunos.nome
q7	select ano_lect,ncd, cadeira, Max(nota) nota FROM exames, cadeiras where exames.cadeira=cadeira.cad AND nota NOT IN ('E','e','F','f','R','r') group by ano_lect,ncd,cadeira
q8	select n_fase AS FASE, nota_entrada AS NE, nota_especif as PE, nota_12ano as ANO12 FROM entradas where ano = 1997 order by n_fase asc, nota_entrada desc, nota_especif desc, nota_12ano desc
q9	select ncd,avg(nota) FROM exames where nota NOT IN ('E','e','F','f','R','r') group by ncd having avg(nota)¿10
w1	update Alunos set nome='jjjjj' where codigo='23131'
w2	insert into Alunos(codigo,nome,bi) values ('2313264314891','jjjjj','232324')
w3	insert into cadeiras(cad,nome) values ('002321230','Lógica')

Tabela B.1: Operações SQL

Código	***Tamanho da resposta (em bytes)***	***Tempo de calculo no servidor (segundos)***
q1	3000000	1
q2	2000	2
q3	300	3
q4	100594	1
q5	434576	1
q6	5021	2
q7	250736	2
q8	73345	1
q9	2038	3
w1	10	1
w2	10	1
w3	10	1

Tabela B.2: Estatísticas das operações SQL

Operação sql	***Tipo***	***Frequência de execução***
q1	leitura	40%
q2	leitura	30%
q3	leitura	30%
w1	escrita	100%

Tabela B.3: Parâmetros da aplicação C1

Operação sql	***Tipo***	***Frequência de execução***
q1	leitura	40%
q2	leitura	20%
q3	leitura	20%
q9	leitura	20%
w1	escrita	100%

Tabela B.4: Parâmetros da aplicação C2

Operação sql	***Tipo***	***Frequência de execução***
q1	leitura	20%
q3	leitura	30%
q4	leitura	10%
q5	leitura	10%
q6	leitura	10%
q7	leitura	10%
q8	leitura	10%
w1	escrita	70%
w3	escrita	30%

Tabela B.5: Parâmetros da aplicação C3

Rede	***Transferência de bytes por segundo***
Lenta	2048 (2 KB/s)
Média	500000 (500 KB/s)
Rápida	4000000 (4 MB/s)

Tabela B.6: Velocidade da rede