



Wilson Bertino
Lopes dos Santos

JDBC (Java DB Connectivity) Concorrente



**Wilson Bertino
Lopes dos Santos**

JDBC (Java DB Connectivity) Concorrente

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Dr. Diogo Nuno Pereira Gomes, Assistente Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e do Mestre Óscar Narciso Mortágua Pereira, Assistente Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president

Prof. Dr. Rui Luís Andrade Aguiar

Professor Associado da Universidade de Aveiro

vogais / examiners committee

Prof. Dra. Maribel Yasmina Campos Alves Santos

Professora Auxiliar do Dep. de Sistemas de Informação da Universidade do Minho
(arguente principal)

Dr. Diogo Nuno Pereira Gomes

Assistente Convidado da Universidade de Aveiro (orientador)

Mestre Óscar Narciso Mortágua Pereira

Assistente Convidado da Universidade de Aveiro (co-orientador)

agradecimentos /
acknowledgements

Os meus agradecimentos vão para os meus pais, cujo esforço e dedicação permitiu-me atingir e superar esta etapa.

Resumo

A API JDBC permite aos programas Java manipularem dados de uma base de dados. No entanto, a definição da API não prevê uma utilização concorrente dos seus serviços, não é por isso possível partilhar objectos JDBC em segurança entre threads.

Neste documento é descrita uma implementação concorrente da interface ResultSet. Esta interface é utilizada para ler ou modificar linhas do resultado da execução de uma instrução SQL. O driver JDBC foi criado para SQL Server 2008.

De modo a avaliar o desempenho da solução desenvolvida foram realizados testes de desempenho comparando-a com a implementação do driver da Microsoft, em que se criou um ResultSet por thread. Os resultados mostraram que a ideia desenvolvida produz um aumento de desempenho em ambientes multithreaded.

Abstract

The JDBC API allows Java programs to access data stored on a data base. However, the API specification doesn't provide a solution for concurrent access to its interfaces, so it isn't safe to shared the same JDBC object between threads.

This document describes the concurrent implementation of the Result Set interface. This interface is used to read or modify lines in the result of executing a SQL statement. The JDBC driver was created for SQL Server 2008.

In order to assess its performance, the developed solution was benchmarked against the situation where it is created one ResultSet per thread using Microsoft's implementation of the JDBC driver. Results show that the solution increases performance on a multithreaded environment.

Conteúdo

Conteúdo	iii
Lista de tabelas	v
Lista de figuras	viii
1 Introdução	1
1.1 Integração de Linguagens de programação e Bases de dados	2
1.2 O que é JDBC?	3
1.3 O que é um JDBC driver?	4
1.4 Breve tutorial JDBC	5
1.4.1 Instalação	5
1.4.2 Criar uma ligação	6
1.4.3 Executar uma query	6
1.5 Motivação	10
2 Implementação de um driver JDBC	13
2.1 Camada TDS	14
2.1.1 TDSMessage	14
2.1.2 ITDSResultSet	16
2.2 Camada JDBC	17
3 Arquitectura do ResultSet Concorrente	19
3.1 Introdução	19
3.1.1 Problema	20
3.1.2 Ideia base da solução	20
3.2 ResultSet Wrapper	21
3.2.1 Utilização	22
3.3 Cursor Concorrente	23
3.3.1 Anatomia de um <i>ResultSet</i>	24
3.3.2 Cache individual	25
3.3.3 Cache partilhado	27
3.3.4 Utilização	29

4	Benchmark	31
4.1	Benchmark principal	31
4.2	Benchmark com atrasos	35
4.3	Cache individual vs Cache partilhado	38
5	Resultados	39
5.1	Plataforma de teste	39
5.2	Benchmark principal	40
5.2.1	Comparação com MSJDBC	40
5.2.2	Comparação com WJDBC	48
5.2.3	Resumo	54
5.3	Benchmark com atrasos	54
5.3.1	Atraso entre colunas	54
5.3.2	Atraso entre linhas	62
5.3.3	Resumo	69
5.4	Cache individual vs Cache partilhado	69
5.4.1	Fetch size 10%	70
5.4.2	Fetch size 20%	70
5.4.3	Fetch size 50%	71
5.4.4	Fetch size 75%	72
5.4.5	Fetch size 100%	72
5.4.6	Resumo	74
6	Discussão	75
6.1	Análise de resultados	75
6.1.1	Comparação com JDBC	75
6.1.2	Comparação com WJDBC	78
6.1.3	Comparação com atrasos	79
6.1.4	Cache individual vs Cache partilhado	80
6.2	Conclusão	80
6.3	Trabalho relacionado	81
6.4	Trabalho Futuro	81
	Glossário	84
	Acrónimos	85
	Bibliografia	87
A	Estudo do SQLServerResultSet	93
A.1	Cursors no SQL Server	93
A.1.1	Fetching e Scrolling	93
A.1.2	Concorrência	94

A.1.3	Tipos de cursor	95
A.2	Tipos de cursor por result set	95
A.3	Adaptive Buffering	98
B	Tabular Data Stream	101
B.1	Mensagens	101
B.2	Pacotes	102
B.2.1	Cabeçalho	102
B.2.2	Zona de dados	103
B.3	<i>Tokenless Streams</i>	104
B.3.1	Pre-Login	104
B.3.2	Login	105
B.3.3	SQLBatch	105
B.4	<i>Token Streams</i>	105
C	Cursor Stored Procedures	107
C.1	sp_cursor	107
C.1.1	Sintaxe	108
C.1.2	Argumentos	108
C.2	sp_cursoropen	108
C.2.1	Sintaxe	109
C.2.2	Argumentos	109
C.3	sp_cursorfetch	110
C.3.1	Sintaxe	110
C.3.2	Argumentos	110
C.4	sp_cursorclose	111
C.4.1	Sintaxe	111
C.4.2	Argumentos	111
D	Funcionalidade implementada	113
D.1	Implementação JDBC	113
D.2	Implementação TDS	115

Lista de Tabelas

1.1	Serviços do ResultSet que permitem mover o cursor.	8
1.2	Serviços do ResultSet que permitem modificar os dados do dataset.	9
2.1	Descrição dos serviços da classe TDSMessage	15
2.2	Descrição dos serviços da interface ITDSResultSet	16
2.3	Interfaces da API JDBC implementadas.	17
3.1	Descrição dos atributos da implementação do <i>ResultSet</i>	24
3.2	Métodos reimplementados pela classe CursorIndividualCache.	27
3.3	Métodos reimplementados pela classe CursorSharedCache	28
A.1	Tipos de cursor suportados pelo driver	96
A.1	Tipos de cursor suportados pelo driver	97
A.1	Tipos de cursor suportados pelo driver	98
B.1	Campos do cabeçalho TDS	103
B.2	Indicação das mensagens que usam <i>tokens</i>	104
B.3	Opções da mensagem de Pre-Login	105
B.4	Packet Data Token Streams	106
C.1	<i>Stored prodecures</i> do sistema relevantes para a implementação do driver JDBC. 107	
D.1	Métodos implementados da interface <i>Driver</i>	113
D.2	Métodos implementados da interface <i>Statement</i>	113
D.3	Métodos implementados da interface <i>ResultSet</i>	114
D.4	Tipos SQL suportados pelo driver.	115

Lista de Figuras

1.1	Arquitetura JDBC	4
2.1	Arquitetura de um driver JDBC do tipo 4 para SQL Server 2008.	14
2.2	Classe TDSMessage.	15
2.3	Interface ITDSResultSet.	16
3.1	Representação da relação entre o <i>ResultSet</i> , o cursor servidor e o <i>dataset</i> . . .	19
3.2	Relação de muitos para um entre o <i>ResultSet</i> e o cursor do servidor.	20
3.3	Diagrama de classes da solução ResultSet Wrapper.	21
3.4	Diagrama das classes envolvidas na utilização das solução Cursor Wrapper. .	23
3.5	Visão geral da implementação do <i>ResultSet</i>	24
3.6	Diagrama de classes do cursor com cache individual.	26
3.7	Diagrama de classes da implementação do cursor com cache partilhado. . . .	28
3.8	Classes principais na utilização da solução Cursor.	30
4.1	Tabela utilizada no benchmark.	34
4.2	Medição do tempo de preparação	34
4.3	Medição do tempo de execução	35
5.1	<i>MSJDBC/CJDBC_I</i> , no contexto Actualização	40
5.2	<i>MSJDBC/CJDBC_S</i> , no contexto Actualização	41
5.3	<i>MSJDBC/WJDBC</i> , no contexto Actualização	42
5.4	<i>MSJDBC/CJDBC_I</i> , no contexto Leitura	42
5.5	<i>MSJDBC/CJDBC_S</i> , no contexto Leitura	43
5.6	<i>MSJDBC/WJDBC</i> , no contexto Leitura	44
5.7	<i>MSJDBC/CJDBC_I</i> , no contexto Inserção	44
5.8	<i>MSJDBC/CJDBC_S</i> , no contexto Inserção	45
5.9	<i>MSJDBC/WJDBC</i> , no contexto Inserção	46
5.10	<i>MSJDBC/CJDBC_I</i> , no contexto Remoção	46
5.11	<i>MSJDBC/CJDBC_S</i> , no contexto Remoção	47
5.12	<i>MSJDBC/WJDBC</i> , no contexto Remoção	48
5.13	<i>WJDBC/CJDBC_I</i> , no contexto Actualização	49
5.14	<i>WJDBC/CJDBC_S</i> , no contexto Actualização	50
5.15	<i>WJDBC/CJDBC_I</i> , no contexto Leitura	50

5.16	<i>WJDBC/CJDBC_S</i> , no contexto Leitura .	51
5.17	<i>WJDBC/CJDBC_I</i> , no contexto Inserção .	51
5.18	<i>WJDBC/CJDBC_S</i> , no contexto Inserção .	52
5.19	<i>WJDBC/CJDBC_I</i> , no contexto Remoção .	53
5.20	<i>WJDBC/CJDBC_S</i> , no contexto Remoção .	53
5.21	<i>MSJDBC/CJDBC_I</i> , efeito do atraso no contexto Atualização .	55
5.22	<i>MSJDBC/CJDBC_S</i> , efeito do atraso no contexto Atualização .	56
5.23	<i>MSJDBC/WJDBC</i> , efeito do atraso no contexto Atualização .	56
5.24	<i>MSJDBC/CJDBC_I</i> , efeito do atraso no contexto Leitura .	57
5.25	<i>MSJDBC/CJDBC_S</i> , efeito do atraso no contexto Leitura .	58
5.26	<i>MSJDBC/WJDBC</i> , efeito do atraso no contexto Leitura .	58
5.27	<i>WJDBC/CJDBC_I</i> , efeito do atraso no contexto Atualização .	59
5.28	<i>WJDBC/CJDBC_S</i> , efeito do atraso no contexto Atualização .	60
5.29	<i>WJDBC/CJDBC_I</i> , efeito do atraso no contexto Leitura .	61
5.30	<i>WJDBC/CJDBC_S</i> , efeito do atraso no contexto Leitura .	61
5.31	<i>MSJDBC/CJDBC_I</i> , efeito do atraso no contexto Atualização .	62
5.32	<i>MSJDBC/CJDBC_S</i> , efeito do atraso no contexto Atualização .	63
5.33	<i>MSJDBC/WJDBC</i> , efeito do atraso no contexto Atualização .	64
5.34	<i>MSJDBC/CJDBC_I</i> , efeito do atraso no contexto Leitura .	64
5.35	<i>MSJDBC/CJDBC_S</i> , efeito do atraso no contexto Leitura .	65
5.36	<i>MSJDBC/WJDBC</i> , efeito do atraso no contexto Leitura .	66
5.37	<i>WJDBC/CJDBC_I</i> , efeito do atraso no contexto Atualização .	67
5.38	<i>WJDBC/CJDBC_S</i> , efeito do atraso no contexto Atualização .	67
5.39	<i>WJDBC/CJDBC_I</i> , efeito do atraso no contexto Leitura .	68
5.40	<i>WJDBC/CJDBC_S</i> , efeito do atraso no contexto Leitura .	69
5.41	<i>CJDBC_I/CJDBC_SM</i> , no contexto 10 .	70
5.42	<i>CJDBC_I/CJDBC_SM</i> , no contexto 20 .	71
5.43	<i>CJDBC_I/CJDBC_SM</i> , no contexto 50 .	71
5.44	<i>CJDBC_I/CJDBC_SM</i> , no contexto 75 .	72
5.45	<i>CJDBC_I/CJDBC_SM</i> , no contexto 100 .	73
5.46	<i>CJDBC_I/CJDBC_S</i> , no contexto 100 .	73
6.1	Comparação entre <i>CJDBC</i> e <i>MSJDBC</i> , no contexto Leitura .	77
6.2	Comparação entre <i>CJDBC</i> e <i>MSJDBC</i> , no contexto Atualização .	77

Capítulo 1

Introdução

O objectivo deste trabalho é realizar uma implementação concorrente de um driver JDBC para SQL Server 2008.

Este primeiro capítulo começa por referir o problema da integração de linguagens de programação e bases de dados, explicando a razão da escolha do JDBC como solução. É fornecida uma descrição mais detalhada do que consiste o JDBC, incluindo um pequeno tutorial exemplificando como se pode utilizar a API JDBC para aceder a dados numa base de dados. Este capítulo explica também o que é um driver JDBC. Por fim são identificados os problemas que a API JDBC apresenta no âmbito da execução de código concorrente, e que estão na base da motivação para a realização deste trabalho.

No capítulo 2 é explicada a arquitectura da implementação do driver JDBC para SQL Server 2008.

No capítulo 3 são apresentadas as soluções para o problema da partilha de objectos JDBC e é explorado em maior profundidade o desenvolvimento do driver JDBC, no que se refere à implementação do ResultSet.

No capítulo 4 é descrito o método experimental utilizado para avaliar o desempenho das soluções desenvolvidas.

No capítulo 5 apresentam-se os resultados obtidos.

No capítulo 6 analisam-se e discutem-se os resultados. São apresentadas as conclusões, é analisado o trabalho relacionado e são indicadas sugestões para trabalhos futuros.

No apêndice A é apresentado um estudo da implementação de um ResultSet para SQL Server. Este apêndice revela conceitos importantes relacionados com o que envolve implementar um ResultSet, e reunindo o que se aprendeu através do estudo do driver da Microsoft.

No apêndice B é descrito o protocolo de comunicação entre aplicações cliente e o SQL Server, conhecido como Tabular Data Stream.

No apêndice C são apresentados os stored procedures que existem no SQL Server e que são utilizados pelo ResultSet para manipular os dados do dataset.

No apêndice D é listada a funcionalidade da API JDBC que foi implementada neste trabalho.

1.1 Integração de Linguagens de programação e Bases de dados

A integração de bases de dados e linguagens de programação é um problema que existe quase há 50 anos [8] e que é conhecido por *impedance mismatch* [8, 62, 2]. Uma das principais razões da existência deste problema é o facto de estas duas entidades terem sido desenvolvidas independentemente, durante muitos anos [3]. E como consequência surgiram incompatibilidades na interoperabilidade entre a interface das linguagens procedimentais e a interface das linguagens *query* das bases de dados. Exemplos dessas incompatibilidades são programas imperativos *versus* queries declarativas, optimização ao nível da compilação *versus* optimização ao nível da query, algoritmos e estruturas de dados *versus* relações e índices, threads *versus* transações, ponteiros nulos *versus* nulo como ausência de dados [8].

Desde sempre, a generalidade dos programas necessitou de alguma forma de dados permanentes. Alguns programas implementam sistemas de armazenamento específicos, mas a verdade é que existem diversos sistemas cuja principal função é a gestão eficiente dos dados. Os Relational Database Management Systems (RBMS) constituem soluções aceites e bem estabelecidas para essa função, e que apesar de surgirem soluções no âmbito de bases de dados orientadas a objectos, crê-se que os sistemas de base de dados relacionais continuem a existir por muitos mais anos [7]. Dois exemplos de popularidade são a Oracle Database e o SQL Server da Microsoft [74]. Existe assim, a necessidade de encontrar uma solução para a integração de linguagens de programação e bases de dados.

Têm sido realizados vários esforços para integrar linguagens de programação e bases de dados. Exemplos são a exploração de linguagens de programação especializadas em base de dados, persistência ortogonal, bases de dados orientadas a objectos, modelos de transação, bibliotecas de acesso a dados, *embedded queries*, e mapeamento objecto-relacional [8].

A persistência ortogonal consiste em estender a existência de um objecto para além do tempo de duração da execução de um programa [5]. Um dos problemas da persistência ortogonal é que não dá espaço a optimizações [8]. PJama [45] e OPJ [4, 70] são exemplos de persistência ortogonal.

Como alternativa à persistência ortogonal existe a execução explícita de queries. A Call Level Interface (CLI) [79] é um mecanismo predominante na execução explícita de queries, e permite à linguagem de programação o acesso ao database engine a partir de uma API estandardizada. Um dos principais problemas da CLI é a ausência de tipagem estática, o que causa a detecção de erros apenas em *runtime*. No entanto, permite melhorar o desempenho geral através da redução da latência na comunicação [8]. ODBC e JDBC são dois exemplos. As *embedded queries* constituem outro mecanismo de execução explícita de queries. Neste mecanismo as instruções (statements SQL) são escritas directamente no código fonte da linguagem de programação. Este mecanismo está a deixar de ser suportado por diversos sistemas, como por exemplo o Microsoft SQL Server [22] e Sybase [78].

Apesar das inúmeras soluções que surgiram e continuam a surgir, ainda não se chegou a um consenso na escolha de uma solução definitiva para o problema da integração. Na

escolha da utilização de uma das soluções, existem alguns factores a considerar tais como a portabilidade e o desempenho.

A linguagem Java permite escrever aplicações independentes da plataforma, para sistemas computacionais fixos ou móveis, tendo por isso um elevado nível de portabilidade. A optimização ao nível da Java Virtual Machine, torna a linguagem uma solução forte também no âmbito do desempenho [1]. Para além disso, sendo o JDBC uma CLI, permite o acesso directo ao database engine, dando flexibilidade para outras opções de desempenho.

A generalização da utilização da linguagem Java e a existência de vários Relational Database Management Systems com raízes profundas, faz do JDBC um objecto de alvo estudo.

1.2 O que é JDBC?

A JDBC (*Java Database Connectivity API*) é uma Application Programming Interface (API) para acesso a bases de dados. A JDBC é uma *SQL-level API* [77] o que significa que é possível construir statements SQL e introduzi-las em chamadas a código Java. Isso faz com que se esteja praticamente a utilizar SQL e ao mesmo tempo tem-se acesso ao mundo orientado a objectos em que os resultados dos pedidos à base de dados são objectos Java e os problemas de acesso são resolvidos com a gestão de excepções. O objectivo principal do JDBC é funcionar de modo simples e flexível.

A JDBC não foi a primeira tentativa de uma solução para o acesso universal a bases de dados. Uma das outras soluções que se destacam é a Open DataBase Connectivity (ODBC) [69, 30], que também tem como principal objectivo fornecer uma interface uniforme de acesso a bases de dados. No entanto sofre de um pouco de excesso de complexidade [77].

O desenvolvimento da JDBC foi influenciado pelas APIs já existentes, tais como a ODBC e a X/Open SQL Call Level Interface (CLI), e foi tido o cuidado de reutilizar as principais abstracções existentes nessas APIs, com o intuito de melhorar a aceitação por parte dos fabricantes de base de dados, e aproveitar o conhecimento já existente dos utilizadores de ODBC e SQL CLI [77].

A API JDBC é definida em dois pacotes Java [53, 75]:

- `java.sql`[54] fornece a API de acesso aos dados (normalmente guardados numa base de dados relacional). É neste pacote que se encontram as classes mais usadas: *Connection*, *ResultSet*, *Statement* e *PreparedStatement*.
- `javax.sql`[55] fornece a API de acesso aos serviços do servidor. Este pacote fornece serviços para J2EE, tais como *DataSouce* e *RowSet*.

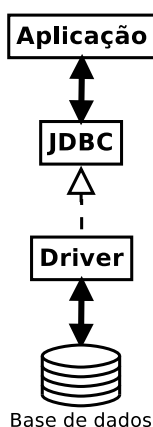
Entre as vantagens da JDBC destacam-se a possibilidade de utilizar os sistemas de base de dados já existentes, a facilidade de aprendizagem, simplicidade na instalação e de manutenção barata. Não há a necessidade de configurações de rede. O URL JDBC contém toda a informação necessária para estabelecer a ligação [52].

1.3 O que é um JDBC driver?

Um JDBC driver é um componente de software que implementa a API JDBC, permitindo assim às aplicações Java interagirem com uma base de dados.

A aplicação que utiliza JDBC para aceder a uma base de dados é alheia ao modo como são implementadas as interfaces que utiliza, isso é inteira responsabilidade do driver. A Figura 1.1 mostra a arquitectura típica do JDBC, e vem reforçar o último ponto: uma aplicação Java utiliza o conjunto de interfaces disponibilizadas pelo JDBC, essas interfaces são implementadas pelo driver que também se encarrega de comunicar com o sistema da base de dados, seja ele qual for (Oracle, SQL Sever, MySQL, etc.).

Figura 1.1: *Arquitectura JDBC*



Cada driver JDBC é construído para um Database Management System (DBMS) específico implementando o protocolo de comunicação de queries e resultados entre o cliente e a base de dados.

Existem quatro categorias de driver[48, 65]:

1. JDBC-ODBC

Utiliza um driver Open Database Connectivity fazendo uma ponte para estabelecer a comunicação. O driver JDBC converte as invocações da API JDBC em invocações a funções ODBC. Uma vez que o ODBC depende de bibliotecas nativas do sistema operativo em que a JVM está a correr, este tipo de driver é dependente da plataforma.

2. Native-API

Converte os pedidos em chamadas a uma biblioteca cliente. O driver JDBC converte as invocações a métodos da API JDBC em invocações nativas da API da base de dados. Este tipo de driver é dependente da plataforma, e necessita que as bibliotecas estejam instaladas no cliente.

3. Network-Protocol

Utiliza uma camada intermédia (middleware) cuja função é converter os pedidos na linguagem (protocolo) da base de dados. A camada intermédia converte as invocações

a métodos da API JDBC em mensagem do protocolo da base de dados. Este tipo de driver é escrito totalmente em código Java e é independente da plataforma porque a camada intermédia encarrega-se das especificidades do sistema.

4. Native-Protocol

Converte os pedidos directamente no protocolo utilizado pelo Database Management System. O seu desempenho é melhor do que o desempenho dos drivers do tipo 1 e 2 porque não existe a conversão em chamadas ODBC ou em chamadas da API da base de dados. Este tipo de driver é escrito totalmente em código Java e é independente da plataforma. A diferença deste tipo para o tipo 3 é que a conversão no protocolo da base de dados é realizada no cliente, enquanto que no driver do tipo 3 a conversão é realizada na camada intermédia.

Um driver da categoria JDBC-ODBC é conhecido como driver JDBC do Tipo 1. Um driver da categoria Native-API é conhecido como driver JDBC do Tipo 2. Um driver da categoria Network-Protocol é conhecido como driver JDBC do Tipo 3. Um driver da categoria Native-Protocol é conhecido como driver JDBC do Tipo 4.

1.4 Breve tutorial JDBC

Nesta secção serão apresentados e explicados alguns exemplos de utilização do JDBC para aceder aos conteúdos de uma base de dados relacional.

1.4.1 Instalação

A instalação de um driver JDBC é tão simples como incluir um ficheiro jar no classpath:

```
java -cp driver.jar:... programa
```

Em que *driver.jar* é o ficheiro jar do driver que se pretende utilizar, e *programa* é o nome da classe Java principal.

Depois no programa, antes do driver poder ser utilizado é necessário criar uma nova instância do driver utilizando:

```
Class.forName(nome);
```

Em que *nome* é uma String que possui o nome da classe que implementa a interface `java.sql.Driver`. Este nome pode ser obtido consultando a documentação que acompanha o driver.

Se o driver implementar a versão 4.0 da API JDBC, este último passo pode ser omitido, porque a versão 4.0 introduz o carregamento automático das classes que implementam `java.sql.Driver` e que estão presentes no *classpath*, através do mecanismo Java SE Service Provider [46].

1.4.2 Criar uma ligação

Para criar uma ligação ao servidor apenas utiliza-se o método:

```
Driver.getConnection(url);
```

O parâmetro *url* é uma string no formato (note-se que o url está no formato para SQL Server 2008)[14]:

```
jdbc:sqlserver://[serverName\[instanceName][:portNumber]]  
[;property=value[;property=value]]
```

Em que:

- `jdbc:sqlserver://` é o sub-protocolo, é constante e obrigatório.
- `serverName` é o endereço do servidor.
- `instanceName` é uma instância no servidor.
- `portNumber` é o número da porta do serviço no servidor.
- `property` é uma propriedade da ligação. As propriedades para SQL Server 2008 podem ser consultadas em [34], e dois exemplos são o nome de utilizador e a password.

A Listagem 1.1 apresenta o código Java para a criação de uma ligação à base de dados utilizando JDBC.

Listagem 1.1: Criação de um objecto da ligação à base de dados.

```
1 // O valor de url deve ser alterado de acordo com a configuração do sistema.  
String url = "jdbc:sqlserver://localhost:1433;database=AdventureWorks "  
3         + ";username=admin;password=admin";  
Connection con = java.sql.DriverManager.getConnection(url);
```

1.4.3 Executar uma query

O resultado da execução de uma statement SQL denomina-se por result set ou dataset e é obtido invocando o método `executeQuery` da interface *Statement*. O resultado desse método é um objecto *ResultSet* que fornece serviços para operar sobre as linhas do dataset, linha a linha. Para criar um result set, temos portanto, de estabelecer uma ligação com o servidor, criar uma statement¹ e invocar o método `executeQuery` da statement, tal como demonstrado na Listagem 1.2.

Listagem 1.2: Criação de um objecto result set.

```
String sql = "SELECT _column1, _column2 FROM _mytable";  
2 Statement stmt = con.createStatement(); // con é o objecto da ligação ao servidor  
ResultSet rs = stmt.executeQuery(sql);
```

¹Instância de uma classe que implementa a interface *Statement*.

As operações disponíveis pelo objecto result set² dependem do seu tipo. A criação da statement (linha 2 da listagem anterior) determina o tipo de result set que é criado. Existem três versões do método createStatement[51]:

- createStatement()
- createStatement(int resultSetType, int resultSetConcurrency)
- createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)

Os parâmetros válidos para estes métodos pertencem à interface *ResultSet* e são descritos a seguir.

resultSetType pode ter um dos seguintes valores:

- TYPE_FORWARD_ONLY O result set só pode ser percorrido numa direcção, da primeira para a última linha, e uma linha de cada vez.
- TYPE_SCROLLABLE_SENSITIVE O result set pode ser percorrido em qualquer direcção, e pode-se aceder a qualquer linha; as modificações externas são visíveis.
- TYPE_SCROLLABLE_INSENSITIVE O result set pode ser percorrido em qualquer direcção, e pode-se aceder a qualquer linha; as modificações externas não são visíveis.

resultSetConcurrency pode ter um dos seguintes valores:

- CONCUR_READ_ONLY O result set só suporta operações de leitura.
- CONCUR_UPDATABLE O result set também suporta operações de modificação.

resultSetHoldability pode ter um dos seguintes valores:

- HOLD_CURSORS_OVER_COMMIT Os cursores continuam abertos após a instrução de commit.
- CLOSE_CURSORS_AT_COMMIT Os cursores são fechados após a instrução de commit.

O método createStatement sem argumentos cria por pré-definição um result set do tipo TYPE_FORWARD_ONLY/CONCUR_UPDATABLE.

Por exemplo para criar um result set scrollable³ actualizável e sensível a actualizações externas criamos uma statement como demonstrado na Listagem 1.3.

Listagem 1.3: Criação de um result set scrollable e actualizável.

```
1 Statement stmt = con.createStatement(ResultSet.TYPE_SCROLLABLE_SENSITIVE ,
                                     ResultSet.CONCUR_UPDATABLE);
```

²Instância de uma classe que implementa a interface *ResultSet*.

³Result set do tipo TYPE_SCROLLABLE_SENSITIVE ou TYPE_SCROLLABLE_INSENSITIVE, permitindo o acesso aleatório às suas linhas.

1.4.3.1 Ler os valores do result set

O processamento do resultado é realizado linha a linha, e acede-se a uma linha movendo o cursor⁴ para essa linha.

A Tabela 1.1 descreve os serviços da interface `ResultSet` que permitem mover o cursor.

Tabela 1.1: *Serviços do `ResultSet` que permitem mover o cursor.*

Método	Descrição
<code>next</code>	move para a próxima linha.
<code>previous</code>	move para a linha anterior.
<code>absolute(n)</code>	move para a linha n.
<code>relative(n)</code>	move n linhas a partir da linha actual.
<code>first</code>	move para a primeira linha.
<code>last</code>	move para a última linha.
<code>beforeFirst</code>	move para a posição anterior à primeira linha.
<code>afterLast</code>	move para a posição posterior à última linha.

Quando o result set é criado o cursor encontra-se antes da primeira linha. A Listagem 1.4 exemplifica a leitura dos dados das colunas da primeira linha do result set.

Listagem 1.4: *Ler a primeira linha do result set.*

```
rs.next(); // mover o cursor para a próxima linha
2 rs.getInt(1); // ler o inteiro da primeira coluna
rs.getString(2); // ler a string da segunda coluna
```

Se o result set for do tipo `TYPE_FORWARD_ONLY` apenas o serviço `next` se encontra disponível, se um dos outros for invocado será apresentado um erro.

1.4.3.2 Modificar os valores de um result set

A interface `ResultSet` também permite actualizar os dados de um result set, mas a `Statement` que lhe dá origem tem que ser criada com o parâmetro `resultSetConcurrency` igual a `CONCUR_UPDATABLE`.

A Tabela 1.2 descreve os serviços da interface `ResultSet` que permitem modificar os dados do dataset.

⁴O interface `ResultSet` fornece o mesmo tipo de funcionalidade que um cursor [19, 20], por isso diz-se que se está a mover o cursor quando se invoca uma operação que altera a linha em que o `ResultSet` se encontra.

Tabela 1.2: *Serviços do ResultSet que permitem modificar os dados do dataset.*

Método	Descrição
updateXXX(n, val)	Actualiza o valor da coluna n com o valor val. Existe um método update para cada tipo de dados, por isso XXX deve ser substituído por Int, String, Date, etc.
updateRow	Envia para o servidor as modificações realizadas à linha.
moveToInsertRow	Move o cursor para uma linha especial que depois pode ser enviada ao servidor para ser inserida no dataset.
insertRow	Envia para o servidor uma linha que deve ser acrescentada ao dataset.
moveToCurrentRow	Cancela a inserção da linha e move o cursor para a linha actual.
deleteRow	Remove uma linha do dataset.

As listagens seguintes exemplificam uma operação de actualização, inserção e remoção, respectivamente.

Listagem 1.5: *Actualização de uma linha do result set.*

```

1 // Criação do result set.
  Statement stmt = con.createStatement(TYPE_SCROLL_SENSITIVE,
3                                     CONCUR_UPDATABLE);
  ResultSet rs = stmt.executeQuery(sql);
5
  // Actualização do result set.
7 rs.absolute(5); // mover para a linha 5, será actualizada.
  rs.updateInt(1, val1); // actualizar a primeira coluna com o valor "val1"
9 rs.updateString(2, val2); // actualizar a segunda coluna com o valor "val2"
  rs.updateRow(); // enviar as modificações para o servidor

```

Listagem 1.6: *Inserção de uma linha no result set.*

```

// Criação igual ao exemplo anterior (...)
2
  // Inserção de uma nova linha.
4 rs.moveToInsertRow(); // iniciar uma inserção.
  rs.updateInt(1, val1);
6 rs.updateString(2, val2);
  rs.insertRow(); // enviar as modificações para o servidor.
8 rs.moveToCurrentRow(); // mover o cursor para linha actual.

```

Listagem 1.7: *Remoção de uma linha do result set.*

```

// Criação igual ao exemplo anterior (...)
2
  // Remoção de uma linha.

```

```
4 rs.last(); // mover para a última linha, que será removida.
rs.deleteRow(); // enviar as modificações para o servidor
```

1.5 Motivação

A motivação para o desenvolvimento deste trabalho surge do facto que o JDBC não inclui mecanismos para tirar partido de um ambiente multithreaded.

As operações dos objectos dos pacotes `java.sql` e `javax.sql` devem ser *thread-safe* [58], isto é, devem operar correctamente numa situação em que existem diversos threads a aceder a um objecto. No entanto, apenas garantir o invariante de um objecto não é tirar partido de um ambiente multithreaded, e há mesmo situações em que um objecto não deve ser partilhado entre threads.

Vejamos a situação representada na Listagem 1.8, que mostra o fio de execução de dois threads que se encontram a trabalhar em paralelo sobre o mesmo objecto (`rs` é uma referência para uma instância de um `ResultSet`).

Listagem 1.8: *Dois threads a operar em simultâneo sobre o mesmo result set.*

	<code>// Thread A</code>	<code>// Thread B</code>
1	<code>rs.absolute(4);</code>	<code>rs.next();</code>
3	<code>int id = rs.getInt(1);</code>	<code>int id = rs.getInt(1);</code>
		<code>String name = rs.getString(2);</code>

Como não há certezas em relação à ordem em que as operações serão executadas, os resultados são imprevisíveis. Por exemplo, a linha 2 do Thread A pode ser executada, e antes que a sua linha 3 seja executada, a linha 2 do Thread B é executada, e a seguir o Thread A lê o valor da linha a seguir à pretendida. Neste exemplo existe também o problema que quando a linha 2 do Thread B foi executada pretendia-se mover para a próxima linha (a linha anterior à chamada `absolute(4)`), e afinal moveu-se para a linha 5.

Vejamos só mais um problema, representado na Listagem 1.9.

Listagem 1.9: *Dois threads a operar em simultâneo sobre o mesmo result set.*

	<code>// Thread A</code>	<code>// Thread B</code>
2	<code>rs.updateInt(1, ival);</code>	<code>rs.relative(5);</code>
	<code>rs.updateString(2, sval);</code>	
4	<code>rs.updateRow();</code>	

O Thread A está a tentar actualizar os valores de uma linha. O Thread B quer mover o result set para uma linha que se encontra 5 linhas à frente da linha actual. O código do Thread B pode executar antes que a chamada `updateRow()` seja realizada, tendo como consequência a perda dos novos valores do Thread A.

Estas são apenas duas situações problemáticas de várias que podem surgir quando se partilha o mesmo objecto de um `ResultSet` entre threads.

Como se resolvem estes problemas? Bem, uma solução poderia passar por criar um result set para cada thread. Mas isso implica a criação e execução de uma statement para cada thread. É facilmente perceptível que esta situação cria desperdício de recursos. Outras soluções envolvem a criação de mecanismos especiais ao nível da aplicação, de modo a providenciar um ambiente realmente concorrente, ou então ser o próprio driver fornecer tais mecanismos. Estas soluções serão discutidas no próximo capítulo.

Capítulo 2

Implementação de um driver JDBC

Este capítulo apresenta as linhas gerais da implementação do driver JDBC do tipo 4 para o Database Management System Microsoft SQL Server 2008 [9]. A informação aqui contida refere-se particularmente ao driver desenvolvido neste trabalho.

Apesar do JDBC ser uma Client Level Interface, e dar por isso a possibilidade de se interacção directa com o database engine, este trabalho não contempla optimizações que têm como origem a utilização de funcionalidades próprias do DBMS utilizado. Pretende-se antes encontrar um modelo de optimização no que se refere ao acesso concorrente de objectos JDBC partilhados. Neste caso em particular, o objectivo centra-se em partilhar um objecto ResultSet. Encontrando um modelo de acesso que prova ser eficiente, independentemente do database engine sobre o qual se está a trabalhar, criam-se condições para que esse modelo seja mais largamente aceite, e possa ser utilizado na implementação de drivers JDBC para os diversos sistemas de base de dados existentes.

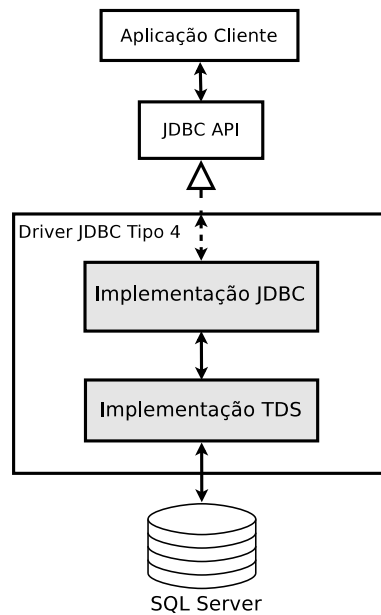
O desenho e implementação deste driver é focalizado na componente cliente que o constitui, deixando a componente servidor o mais abstracta possível. A única componente específica referente à componente servidor, consiste na interface de comunicação com este. Mais propriamente utiliza-se o protocolo de comunicação com o SQL Server (protocolo Tabular Data Stream (TDS)), e utiliza-se a interface programática existente para efectuar diversas operações com um cursor do servidor, tais como a sua declaração ou a obtenção de uma linha específica do result set (conjunto de linhas seleccionadas pela statement SQL). Esta especialização da componente servidor era inevitável para se conseguir criar um driver JDBC funcional, que possibilitasse testar o seu desempenho.

O driver implementa duas camadas: JDBC e TDS. A camada JDBC refere-se ao conjunto de classes que implementam as interfaces definidas na API JDBC [46]. A camada TDS refere-se ao conjunto de classes e interfaces que implementam o protocolo de comunicação com SQL Server, o Tabular Data Stream [11, 63].

A arquitectura geral de um driver JDBC do tipo 4 para SQL Server 2008 é apresentada na Figura 2.1. O driver JDBC converte os pedidos de acesso à API JDBC em invocações à camada TDS, que conhece o modo como os pedidos devem ser realizados ao SQL Server e

converte esses pedidos em mensagens do protocolo TDS.

Figura 2.1: *Arquitetura de um driver JDBC do tipo 4 para SQL Server 2008.*



2.1 Camada TDS

A camada TDS do driver JDBC constitui a implementação do protocolo Tabular Data Stream, que consiste no protocolo de transporte de informação entre uma aplicação cliente e o SQL Server.

Esta secção apenas descreve a interface que esta camada disponibiliza. No apêndice B é fornecida uma resumida descrição do protocolo. Informação mais detalhada sobre o protocolo pode ser obtida nas páginas MSDN dedicadas ao TDS [11] ou na especificação da Sybase [63].

A camada TDS fornece duas entidades que permitem à camada superior interagir com o SQL Server sem conhecer as especificidades do protocolo: *TDSMessage* e *ITDSResultSet*.

2.1.1 TDSMessage

A classe *TDSMessage* implementa toda a comunicação com o servidor. A comunicação é realizada utilizando um socket TCP.

Os serviços disponibilizados pela classe *TDSMessage* consistem em invocações de serviços do SQL Server. A classe converte os pedidos da camada superior na linguagem do SQL Server. A Figura 2.2 apresenta os serviços disponibilizados pela classe.

Figura 2.2: Classe *TDSMessage*.

TDSMessage
<pre> +close() +login(hostName:String,userName:String,password:String, serverName:String,database:String) +executeSQLBatch(sqlText:String): ITDSResultSet +cursorClose(cursor:int) +cursorOpen(String stmt,scrollopt:int,ccopt:int): ITDSResultSet +cursorUpdate(cursor:int,rownum:int,updateValues:UpdateValue[]) +cursorUpdateAbsolute(cursor:int,rownum:int,updateValues:UpdateValue[]) +cursorDelete(cursor:int,rownum:int) +cursorDeleteAbsolute(cursor:int,rownum:int) +cursorInsert(cursor:int,updateValues:UpdateValue[]) +cursorFetch(cursor:int,fetchType:int,rownum:int,nrows:int): ITDSResultSet +cursorRefresh(cursor:int,rownum:int,nrows:int): ITDSResultSet +cursorRows(): int </pre>

A classe *UpdateValue* que é utilizada nos métodos *cursorUpdate*, *cursorUpdateAbsolute*, *cursorDelete*, *cursorDeleteAbsolute* e *cursorInsert*, tem como objectivo converter os valores das colunas de uma linha no formato do protocolo. Converte o valor de um tipo de dados Java num conjunto de bytes formatados de acordo com o TDS, o resultado é depois transferido para o SQL Server na mensagem (actualização, remoção ou inserção).

A Tabela 2.1 apresenta uma sucinta descrição dos serviços da classe *TDSMessage*.

Tabela 2.1: Descrição dos serviços da classe *TDSMessage*

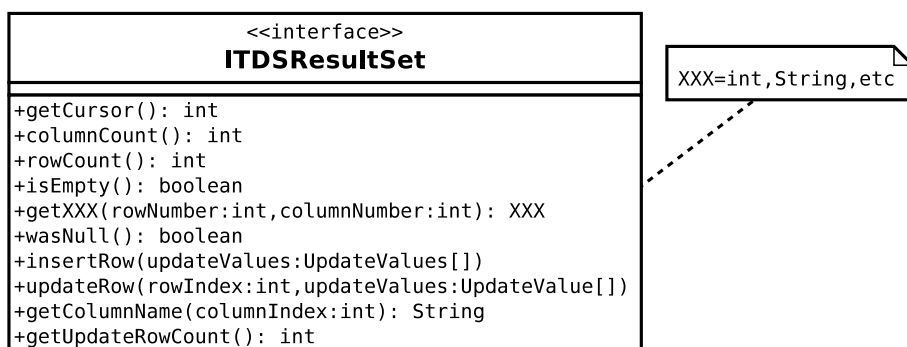
Nome	Descrição
<code>close</code>	Fecha os streams de comunicação com o servidor.
<code>login</code>	Envia um pedido de login ao servidor.
<code>executeSQLBatch</code>	Envia um pedido de execução de um batch de comandos SQL ao servidor.
<code>cursorClose</code>	Envia um pedido ao servidor para fechar e desalocar um cursor.
<code>cursorOpen</code>	Envia um pedido ao servidor para alocar e abrir um cursor.
<code>cursorUpdate</code>	Envia um pedido ao servidor de actualização de valores das colunas de uma linha, considerando uma posição relativa.
<code>cursorUpdateAbsolute</code>	Envia um pedido ao servidor de actualização de valores das colunas de uma linha, considerando uma posição absoluta.
<code>cursorDelete</code>	Envia um pedido ao servidor de remoção de uma linha, considerando uma posição relativa.
<code>cursorDeleteAbsolute</code>	Envia um pedido ao servidor de remoção de uma linha, considerando uma posição absoluta.
<code>cursorInsert</code>	Envia um pedido ao servidor de inserção de uma linha.
<code>cursorFetch</code>	Envia um pedido ao servidor de carregamento de linhas do result set.
<code>cursorRefresh</code>	Envia um pedido ao servidor de recarregamento de linhas do result set.
<code>cursorRows</code>	Envia um pedido ao servidor de informação relativamente ao número total de linhas do result set do cursor.

2.1.2 ITDSResultSet

Quando o servidor envia ao cliente um result set¹, este vem formatado de acordo com o protocolo. A interface ITDSResultSet fornece serviços para extrair a informação do result set, garantindo acesso aos dados sem a necessidade do conhecimento dos detalhes.

A Figura 2.3 apresenta os serviços fornecidos pela interface ITDSResultSet.

Figura 2.3: Interface ITDSResultSet.



A descrição dos serviços da interface ITDSResultSet é apresentada na Tabela 2.2.

Tabela 2.2: Descrição dos serviços da interface ITDSResultSet

Nome	Descrição
<code>getCursor</code>	Obtém o identificador do cursor do servidor.
<code>columnCount</code>	Obtém a quantidade de colunas que cada linha do result set possui.
<code>rowCount</code>	Obtém a quantidade de linhas que foram enviadas para o cliente.
<code>isEmpty</code>	Verifica se a quantidade de linhas é zero.
<code>getXXX</code>	Obtém o valor de uma coluna, dado o seu índice. Existe um método get para cada um dos tipos de dados Java suportados: int, double, String e Date.
<code>wasNull</code>	Verifica se o valor da coluna lido é SQL NULL.
<code>insertRow</code>	Insere uma linha na cópia do result set que existe no cliente.
<code>updateRow</code>	Actualiza uma linha da cópia do result set que existe no cliente.
<code>getColumnName</code>	Obtém o nome de uma coluna, dado o seu índice.
<code>getUpdateRowCount</code>	Obtém o número de linhas que foram alteradas na base de dados como resultado da execução de um comando DML.

¹Resultado da execução de uma statement SQL

2.2 Camada JDBC

Esta camada é constituída por um conjunto de classes que implementam as interfaces definidas na API JDBC.

Neste trabalho procedeu-se apenas a uma implementação parcelar da API. Um dos objectivos principais para o driver JDBC era suportar a execução de statements SQL e obter acesso a um result set. A Tabela 2.3 lista as interfaces que foram implementadas e identifica as classes que as implementam. As interfaces implementadas pertencem todas ao pacote `java.sql`.

No capítulo 3 é apresentada uma explicação mais detalhada da implementação da interface `ResultSet`.

Tabela 2.3: *Interfaces da API JDBC implementadas.*

Interface	Classe	Descrição
Driver	CDriver	Esta interface tem que ser obrigatoriamente implementada por um driver. A classe <code>java.sql.DriverManager</code> encarrega-se de carregar os drivers, que mais tarde serão usados para criar uma ligação à base de dados.
Statement	CStatement	Esta interface permite executar statements SQL e obter o respectivo result set.
ResultSet	CResultSet	Esta interface permite obter acesso ao resultado da execução de uma statement SQL. Existem serviços para leitura e modificação do result set.

Capítulo 3

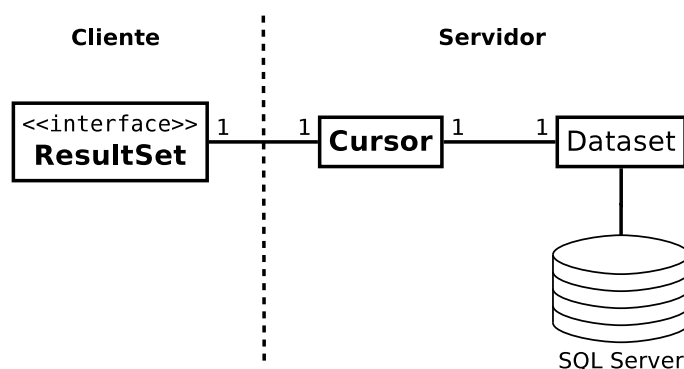
Arquitectura do ResultSet Concorrente

3.1 Introdução

O *ResultSet* é uma interface que fornece uma abstracção ao paradigma Orientado a Objectos para um conceito do paradigma Relacional: o **cursor**. O comportamento e terminologia envolvidos na criação do objecto result set, a sua manipulação, até à sua remoção são os mesmos que se encontram associados a um cursor do servidor (também conhecido como cursor de base de dados).

A implementação da interface *ResultSet* cria um cursor no SQL Server¹, e na sua operação interage com o cursor do servidor de modo a obter ou actualizar os dados do *dataset*² associado ao cursor. Este *dataset* corresponde ao conjunto de linhas que satisfazem a condição da statement SQL executada usando o método `executeQuery` da interface *Statement*. A relação entre estas entidades encontra-se representada na Figura 3.1.

Figura 3.1: Representação da relação entre o *ResultSet*, o cursor servidor e o dataset



¹Normalmente é feita uma excepção para o *ResultSet* do tipo FORWARD-ONLY/READ-ONLY em que o *ResultSet* é obtido através da execução de um batch.

²Os termos *dataset* e *result set* são equivalentes quando se referem aos dados seleccionados por uma statement SQL. O termo result set também é usado no documento para se referir a uma instância da interface *ResultSet*.

Devido a esta relação *ResultSet*/cursor podemos dizer que criar uma instância do *ResultSet* corresponde a criar um cursor cliente.

3.1.1 Problema

A relação de 1 para 1 entre *ResultSet* e cursor, implica que para cada *ResultSet* criado num programa Java, irá ser alocado e aberto um novo cursor no servidor. Como a API JDBC não define um mecanismo que permita tirar partido de um ambiente multithreaded, para se poder trabalhar concorrentemente sobre um *dataset* é necessário criar um *ResultSet*/cursor para cada entidade concorrente, o que desde logo implica uma maior alocação de recursos. A maior alocação de recursos traduz-se em desperdício, porque está claro que vai haver replicação várias vezes da mesma informação. E este é um problema que se verifica tanto no lado do cliente como do servidor. Do lado do cliente existe a instanciação de mais objectos. Do lado do servidor existem mais cursores alocados e abertos.

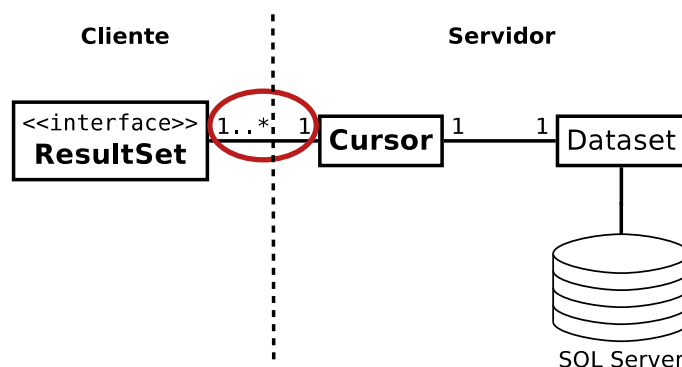
3.1.2 Ideia base da solução

As boas práticas do acesso a informação em base de dados ensinam que se deve evitar a utilização de cursores pois normalmente eles utilizam muitos recursos e reduzem o desempenho e a escalabilidade das aplicações [71], e que se devem utilizar alternativas, como por exemplo:

- Ciclos `while`
- Tabelas temporárias
- Tabelas derivadas
- Múltiplas *queries*

Uma vez que o modo de operação de um *ResultSet* é linha a linha, temos de continuar a utilizar cursores do servidor. Mas o que se pode fazer é diminuir o número de cursores utilizados. Por isso a ideia base consiste em transformar a relação de 1 para 1 numa relação de muitos para um, isto é, permitir que várias instâncias do *ResultSet* utilizem o mesmo cursor do servidor. Esta alteração é assinalada na Figura 3.2.

Figura 3.2: Relação de muitos para um entre o *ResultSet* e o cursor do servidor.



A implementação encontra-se, deste modo, preparada para operar num ambiente concorrente, garantindo uma cooperação correcta entre as entidades concorrentes. A cada entidade é atribuída uma referência para uma instância do *ResultSet* que por sua vez opera concorrentemente com as outras instâncias sobre o mesmo cursor, permitindo o desejado acesso concorrente ao dataset.

Agora que já se sabe qual é a ideia base da solução, nas duas próximas secções serão descritas duas possíveis soluções concretas; uma implementando a concorrência ao alto nível e outra ao baixo nível.

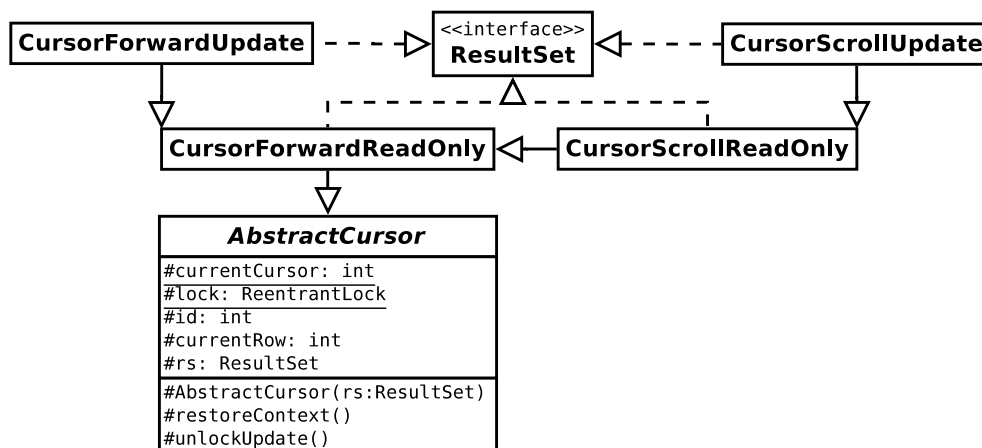
3.2 ResultSet Wrapper

Esta primeira solução é independente da implementação do driver, isto é, funciona para qualquer driver que existe. A ideia base consiste em criar um mecanismo que controle os acessos concorrentes a um objecto result set partilhado, e providencie uma implementação dos serviços garantindo o acesso correcto ao result set, resolvendo assim os problemas identificados anteriormente (secção 1.5).

O acesso correcto ao result set implica que para cada entidade que lhe acede seja guardado o número da linha do result set em que ele se encontra a trabalhar, e quando voltar a ser a entidade activa o result set volta a encontrar-se nessa linha. Para tal é necessário que a linha do result set em que cada entidade se encontra a trabalhar seja guardada quando existe troca de entidade activa, e seja restaurada a linha quando voltar a ser a entidade activa. A operação que guarda o número da linha do result set denomina-se por **salvaguarda de contexto** e a operação que move o result set para a linha guardada denomina-se por **restauração de contexto**.

A Figura 3.3 apresenta o diagrama de classes da solução ResultSet Wrapper.

Figura 3.3: Diagrama de classes da solução ResultSet Wrapper.



A solução consiste em criar cursores cliente que encapsulam (wrap) e partilham a mesma instância de um *ResultSet*. A cada thread que pretende trabalhar sobre o result set é lhe atribuído um cursor cliente. O processo de salvaguarda/restauro de contexto é realizado pelo

cursor cliente, e é totalmente transparente para o utilizador. Isto significa que o modo de utilização desta solução é rigorosamente igual à utilização da interface *ResultSet*.

Existem quatro tipos de cursores:

CursorForwardReadOnly

Wrapper para um result set FORWARD_ONLY/READ_ONLY.

CursorForwardUpdate

Wrapper para um result set FORWARD_ONLY/UPDATABLE.

CursorScrollReadOnly

Wrapper para um result set SCROLLABLE_SENSITIVE/READ_ONLY.

CursorScrollUpdate

Wrapper para um result set SCROLLABLE_SENSITIVE/UPDATABLE.

Cada tipo de cursor implementa os métodos da interface *ResultSet* que fazem sentido, lançando uma excepção dizendo que a operação não é suportada no caso contrário. Por exemplo, o *CursorScrollReadOnly* implementa os métodos relacionados com o movimento do cursor, e lança excepção se se tentar invocar um método de update, enquanto que o *CursorForwardUpdate* suporta os métodos de update e lança uma excepção se se tentar mover o cursor para uma linha específica (método absolute).

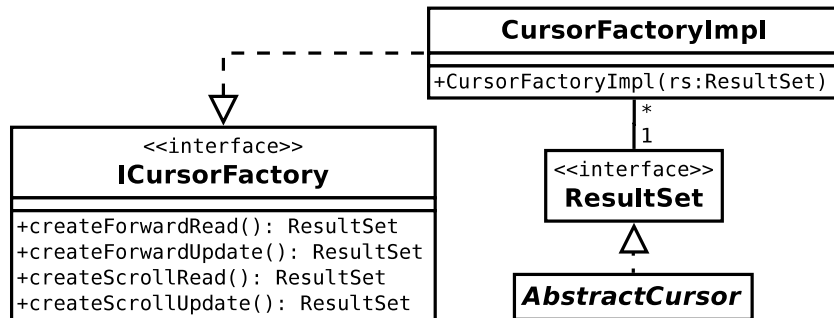
A gestão dos contextos dos cursores cliente é realizada pelos próprios cursores. Cada um tem a noção da linha do result set onde se encontra acedendo à variável `currentRow` da própria classe. Assim, quando é o cursor cliente activo, se for necessário recupera o seu contexto, movendo o result set para a linha onde se encontrava a trabalhar. A detecção da mudança do cursor activo é realizada com o auxílio da variável `currentCursor` que tem visibilidade ao nível da classe *AbstractCursor* e que guarda o identificador do cursor activo. Quando um cursor entra em execução, o primeiro procedimento é verificar se o valor de `currentCursor` é igual ao seu identificador (`id`), se for igual então não é necessário mover o result set, porque ele foi o último cursor a trabalhar com o result set, caso contrário invoca o método do result set que o posiciona na linha onde se encontrava a trabalhar anteriormente. O trabalho sobre o result set tem que ser realizado num regime de acesso exclusivo, para o obter esse acesso exclusivo (lock) é utilizada a variável `lock` que é uma instância da classe *ReentrantLock*. Esta classe providencia o mesmo comportamento e semântica que o monitor explícito acedido usando métodos e blocos `synchronized`, mas com funcionalidades extra [50].

3.2.1 Utilização

A criação de um result set wrapper é realizada por intermédio da interface *ICursorFactory*. Esta interface fornece um serviço para construir cada um dos tipos de cursor (*CursorForwardReadOnly*, *CursorForwardUpdate*, *CursorScrollReadOnly* e *CursorScrollUpdate*). A interface da fábrica de cursores é implementada pela classe *CursorFactoryImpl*, que recebe

no seu construtor o objecto result set que será mais tarde partilhado pelos cursores. No acto de construção de um cursor é verificado se o result set encapsulado é compatível com o tipo de cursor que se está a pedir, lançando-se uma excepção numa situação de incompatibilidade. Por exemplo, se o método `createForwardRead` é invocado e o result set é do tipo `SCROLLABLE_SENSITIVE/READ_ONLY` uma excepção a explicar a situação é criada e lançada. A Figura 3.4 apresenta o diagrama de classes com a fábrica de cursores.

Figura 3.4: Diagrama das classes envolvidas na utilização da solução *Cursor Wrapper*.



Para utilizar esta solução primeiro cria-se um objecto `ResultSet`, como se criaria habitualmente utilizando o método `executeQuery` da interface `Statement`. Em seguida cria-se uma instância da fábrica de cursores passando como argumento o result set ao seu construtor. O último passo consiste em utilizar um dos serviços disponibilizados pela interface da fábrica para criar o wrapper do result set desejado.

A Listagem 3.1 exemplifica a utilização da solução `ResultSet Wrapper` para um result set do tipo `TYPE_FORWARD_ONLY/CONCUR_UPDATABLE`.

Listagem 3.1: Criação de um *ResultSet Wrapper*.

```

1 // Criação do result set.
  ResultSet rs = stmt.executeQuery(sql);
3 // Criação da fábrica de cursores.
  ICursorFactory factory = new CursorFactoryImpl(rs);
5 // Criação do wrapper (cursor).
  ResultSet cursor = factory.createForwardUpdate();
  
```

O objecto `stmt` é uma instância de `Statement` e foi criado com o tipo adequado ao tipo de wrapper criado na listagem. O argumento `sql` é uma `String` que contém a statement SQL que dá origem ao result set.

3.3 Cursor Concorrente

O `ResultSet Wrapper` parece ser uma solução válida, no entanto, ao colocar o controlo de acesso à região crítica num nível tão alto, diminui-se a concorrência porque existe mais código que poderia ser executado em paralelo e que passa a ser executado sequencialmente. Como consequência perde-se a oportunidade de explorar sistemas com multiprocessador e diminui-se também o *throughput*[61, 73].

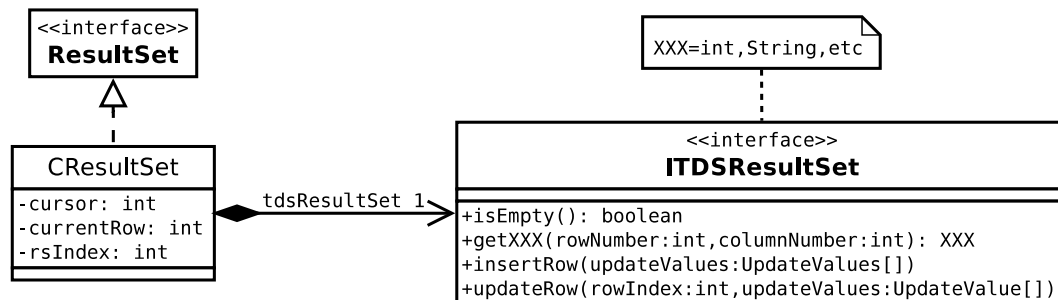
Usando um driver JDBC existente implica que o melhor que se consegue é mesmo ficar nesse nível alto de concorrência. Mas implementando um driver próprio tem-se acesso ao funcionamento interno, e por isso consegue-se analisar e descobrir os pontos que são totalmente concorrentes (isto é, que podem ser executados em paralelo), e também descobrir os pontos em que se acede a recursos partilhados e tem que haver por isso acesso sequencial e exclusivo. Nesta secção é apresentada uma solução que começou pela implementação de um driver JDBC do tipo 4 (Native-Protocol Driver). A seguir à implementação do driver procedeu-se à inclusão do mecanismo apresentado na secção 3.1.2 na implementação do driver.

3.3.1 Anatomia de um *ResultSet*

Nesta secção pretende-se, sem entrar em grandes detalhes, dar a entender a estrutura geral de uma implementação do *ResultSet*, fazendo assim a cobertura de alguns conceitos e terminologias que ajudam a perceber as subseqüentes secções.

A classe *CResultSet* implementa a interface *ResultSet*, comunicando com camada que implementa o protocolo Tabular Data Stream (TDS) para executar as operações sobre o result set. A Figura 3.5 apresenta o diagrama de classes simplificado da implementação do *ResultSet*.

Figura 3.5: Visão geral da implementação do *ResultSet*



A Tabela 3.1 possui uma descrição dos atributos da classe *CResultSet*.

Tabela 3.1: Descrição dos atributos da implementação do *ResultSet*.

Atributo	Descrição
cursor	Identificador do cursor do servidor. É obtido na resposta do servidor ao pedido de criação do cursor (ver secção C.2).
currentRow	Número da linha actual do cursor cliente.
rsIndex	Índice para aceder ao cache.
tdsResultSet	Cache do result set.

Apesar de o *ResultSet* operar linha a linha interagindo com o cursor do servidor, cada instância de uma implementação do *ResultSet* possui em memória um conjunto de linhas, isto é, possui um *cache*, de modo a diminuir o número de mensagens trocadas entre o cliente e o servidor. Tal como na arquitectura de computadores em que os processadores usam o

princípio da localidade de referência[76, 72] e transferem para a memória de mais alto nível não só os dados pedidos, mas um bloco de dados adjacentes; no acesso a uma linha de um result set existe uma grande probabilidade de uma linha próxima ser também utilizada (pelo princípio), é por isso uma boa estratégia, sempre que existe o pedido de uma linha que não se encontra na cache, transferir um bloco de linhas adjacentes.

A classe *CResultSet* possui um objecto que implementa a interface *ITDSResultSet*. Esse objecto corresponde ao cache referido anteriormente e possui o result set formatado de acordo com o protocolo TDS. O cache essencialmente é constituído por uma lista com as linhas do result set e possui ainda a descrição das colunas que formam uma linha, nessa descrição está presente, por exemplo, o tipo de dados da coluna e o respectivo nome.

O cache é indexado usando a variável **rsIndex**, cuja gama de valores vai desde zero até ao tamanho da cache. O tamanho da cache é definido pelo método `setFetchSize` da interface *ResultSet*. A variável **rsIndex** para além de ser utilizada para ler o conteúdo de uma linha do result set, pode também ser utilizada nas operações de actualização e remoção (se o tipo de result set as suportar). O cursor servidor actualiza o result set utilizando um índice relativo ao seu buffer (ver apêndice C, particularmente a secção C.1), esse índice corresponde exactamente ao valor de **rsIndex**. A operação de inserção ignora este valor, porque as inserções são realizadas após a última linha do result set.

O número da linha actual do result set é guardado pela variável **currentRow**, cujo valor varia na gama 1 até ao número total de linhas seleccionadas pela execução da statement SQL. O seu valor pode ser obtido a partir do método `getRow` da interface *ResultSet*, e tem particular importância para o funcionamento interno de *CResultSet* na verificação da necessidade de efectuar um fetch de um novo conjunto de linhas. Se o valor de **currentRow** não estiver contido na gama de linhas que o cache possui, então é preciso pedir novas linhas ao cursor do servidor.

3.3.2 Cache individual

Uma vez que diminuindo a disputa pelo acesso a uma memória partilhada melhora-se a eficiência de uma aplicação [61], a ideia desta solução é tentar diminuir ao máximo os pontos em que é necessário proceder ao lock para a obter acesso exclusivo. Daí surgiu a implementação de um *ResultSet* que para cada instância criasse uma cópia (normalmente parcial) do result set. O que isto significa é que cada cursor cliente possui cache próprio.

O cache individual possui as seguintes vantagens:

1. O acesso ao cache, tanto para leitura como para escrita, é realizado sem restrições, isto é, não há a necessidade obter lock sobre o objecto do cache.
2. Numa situação de cópia parcial do result set diminui-se o número de vezes que se interage com o servidor.

O segundo ponto carece de uma explicação. Imaginemos um cache com capacidade de uma linha, partilhado por alguns threads. Um thread T1 está a trabalhar na linha 5 do

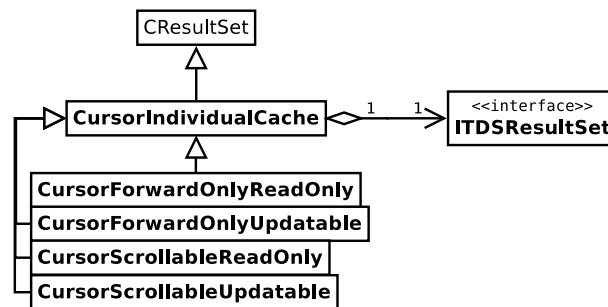
result set, entretanto T1 deixa de ser o thread em execução, que passa a ser o thread T2 cuja linha do result set é a 10 e que agora terá de ser pedida ao servidor. A troca de thread em execução provoca uma grande quantidade de mensagens trocadas entre o cliente e o servidor, pois cada thread precisa de pedir as linhas em que está a trabalhar. E apesar de se ter utilizado um exemplo simples em que o cache tem capacidade de apenas uma linha, caches com capacidades maiores sofrem ainda mais de perda de desempenho (a demonstração e explicação deste fenómeno encontram-se nas secções 5.4 e 6.1.4, respectivamente). Com um cache individual o carregamento do conteúdo do cache é realizado consoante as necessidades individuais do cursor cliente, pelo que se resolve o problema do maior volume de mensagens trocadas entre o cliente e o servidor, porque o cursor cliente tem em cache as linhas em que estava a trabalhar.

O cache individual possui as seguintes desvantagens:

1. Replicação da informação; podem existir caches cujo conteúdo seja o mesmo.
2. A actualização de um dos caches não reflete as alterações nos restantes caches, sendo necessário os cursores pedirem ao servidor os novos dados.

A Figura 3.6 apresenta o diagrama de classes da solução com cache individual. Note-se a relação de *1 para 1* entre CursorIndividualCache e ITDSResultSet, que determina o cache como único para cada cursor cliente.

Figura 3.6: *Diagrama de classes do cursor com cache individual.*



As classes CursorForwardOnlyReadOnly, CursorForwardOnlyUpdatable, CursorScrollableReadOnly e CursorScrollableUpdatable implementam um result set do tipo FORWARD_ONLY/READ_ONLY, FORWARD_ONLY/UPDATABLE, SCROLLABLE_SENSITIVE/READ_ONLY e SCROLLABLE_UPDATABLE, respectivamente. Estas classes reimplementam os métodos que o respectivo result set não suporta, indicando um erro se esses métodos forem invocados.

A classe CursorIndividualCache tem como função adequar a implementação do ResultSet realizada pelo CResultSet à situação de ter um cache individual. A Tabela 3.2 descreve os métodos que têm de ser reimplementados (override) pela classe CursorIndividualCache.

Tabela 3.2: *Métodos reimplementados pela classe CursorIndividualCache.*

Método	Explicação
next	Cada cursor cliente pode mover o cursor do servidor, assim quando um cursor cliente pede ao servidor que mova o seu cursor para o próxima linha ou próximo buffer, a posição inicial pode não ser a correcta para o cursor cliente que faz o pedido. Por isso, o método next é reimplementado para executar um FETCH ABSOLUTE em vez de um FETCH NEXT (a secção A.1.1 apresenta uma explicação destes termos).
update	Aqui também existe o problema do cursor do servidor poder estar noutra linha, por isso o RPC sp_cursor tem que ser invocado com um <i>optype</i> igual a ABSOLUTE UPDATE (conjunção de duas opções usando o operador OR) em vez de UPDATE (ver secção C.1). Ao efectuar esta alteração do valor de optype o cursor do servidor passa a considerar o argumento <i>rownum</i> como sendo o número da linha contando desde o início do result set em vez de contar desde o início do buffer.
delete	A situação é semelhante ao update, mas o valor DELETE do argumento <i>optype</i> é substituído por ABSOLUTE DELETE.

3.3.3 Cache partilhado

Como foi visto na secção anterior o cursor com cache individual será vantajoso numa situação em os threads trabalham sobre gamas de linhas díspares e distantes, no entanto provoca a replicação de informação transmitida e guardada no cliente.

Esta secção apresenta a implementação do cursor cliente, em que as várias instâncias do cursor partilham o mesmo cache.

O cache partilhado tem as seguintes vantagens:

1. Só existe uma cópia dos dados do result set.
2. A actualização do cache realizada por um cursor cliente é visível pelos restantes cursores cliente.

O cache partilhado tem as seguintes desvantagens:

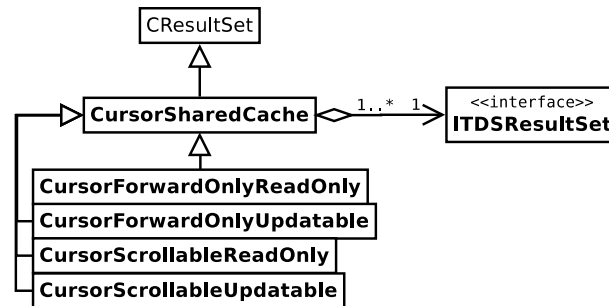
1. Todo o acesso ao cache, seja para leitura ou escrita, tem que ser precedido de um lock no objecto que representa o cache.
2. O número de mensagens trocadas com o servidor pode aumentar, diminuindo o desempenho da aplicação, principalmente numa situação em que os cursores cliente trabalhem em gamas de linhas distantes entre si.

Se o cache for partilhado nas operações de leitura (`getInt`, `getDate`, ...) é necessário recuperar o contexto, isto é, verificar se a linha para a qual se moveu o cursor cliente ainda está presente no cache, e efectuar um fetch caso não esteja em cache. Num ambiente multithreaded é muito frequente outro thread entrar em execução e alterar as linhas que estão presentes no cache. Isto provoca uma maior troca de mensagens entre o cliente e o servidor porque é necessário alterar várias vezes o conteúdo do cache. Tentando resolver este problema, na

implementação do cursor com cache partilhado são seleccionadas e guardadas em cache todas as linhas do dataset. Isto transforma o cache hit ratio em 100%.

A Figura 3.7 apresenta o diagrama de classes da solução com cache partilhado. Podemos ver que a relação entre os cursores cliente e o cache é de muitos para 1; a mesma memória cache (*ITDSResultSet*) é utilizada por vários cursores cliente.

Figura 3.7: Diagrama de classes da implementação do cursor com cache partilhado.



Os atributos e os métodos de *CursorSharedCache* não são apresentados, porque essencialmente o que a classe faz é reimplementar (override) os métodos da interface *ResultSet*, adequando-os ao facto de agora estar a operar com um cache partilhado. A Tabela 3.3 apresenta os métodos que têm de ser reimplementados pela classe *CursorSharedCache*.

Tabela 3.3: Métodos reimplementados pela classe *CursorSharedCache*

Método	Explicação
afterLast	A implementação normal deste método invoca o RPC <code>sp_cursorfetch</code> com o optype <code>FETCH AFTER</code> (A.1.1), o que faz com que o cursor do servidor seja movido para uma posição a seguir à última linha e o buffer fique vazio. Com cache partilhado não há a necessidade de mover o cursor do servidor, apenas o cursor cliente. Também não é desejável que o cache seja esvaziado.
beforeFirst	A situação é semelhante ao <code>afterLast</code> , mas o optype é <code>FETCH BEFORE</code> (A.1.1), e o cursor do servidor é movido para uma posição anterior à primeira linha do result set.
getInt	O acesso ao cache para leitura só pode ocorrer depois da obtenção do lock.
getString	O acesso ao cache para leitura só pode ocorrer depois da obtenção do lock.
getDouble	O acesso ao cache para leitura só pode ocorrer depois da obtenção do lock.
getDate	O acesso ao cache para leitura só pode ocorrer depois da obtenção do lock.
updateRow	O acesso ao cache para actualizar uma linha só pode ocorrer depois da obtenção do lock.
insertRow	O acesso ao cache para inserir uma linha só pode ocorrer depois da obtenção do lock.
scroll	O método <code>scroll</code> serve de boilerplate para os métodos de navegação no result set. Como no cache partilhado todas as linhas do result set já se encontram no cache, não há a necessidade de verificar se uma linha se encontra em cache e em caso negativo pedi-la ao servidor.

A partilha do cache implica a utilização de um mecanismo de sincronização. O cursor cliente com cache partilhado não só tem de garantir acesso exclusivo ao canal de comunicação com o servidor, como tem também de garantir o acesso exclusivo ao cache. Este aspecto parece constituir uma desvantagem em relação à implementação com cache individual, sendo um potencial factor de perda desempenho. No entanto, com o cache partilhado não existe o overhead da construção de vários caches. Esta discussão ficará para a secção de análise dos resultados (secção 6.1) apresentados no capítulo 5.

As classes `CursorForwardOnlyReadOnly`, `CursorForwardOnlyUpdatable`, `CursorScrollableReadOnly` e `CursorScrollableUpdatable` implementam um result set do tipo `FORWARD_ONLY/READ_ONLY`, `FORWARD_ONLY/UPDATABLE`, `SCROLLABLE_SENSITIVE/READ_ONLY` e `SCROLLABLE_UPDATABLE`, respectivamente. Estas classes reimplementam os métodos que o respectivo result set não suporta, indicando um erro se esses métodos forem invocados.

3.3.4 Utilização

O driver implementado é utilizado tal como outro driver JDBC. A versão 4.0 da API JDBC introduziu a funcionalidade de carregamento de um `java.sql.Driver` [46], através da utilização do mecanismo *Java SE Service Provider*. Isto significa que já não é necessário invocar `Class.forName`. O driver suporta esta funcionalidade, por isso para obter uma ligação à base de dados basta utilizar o código apresentado na Listagem 3.2. O jar do driver tem que se encontrar no classpath.

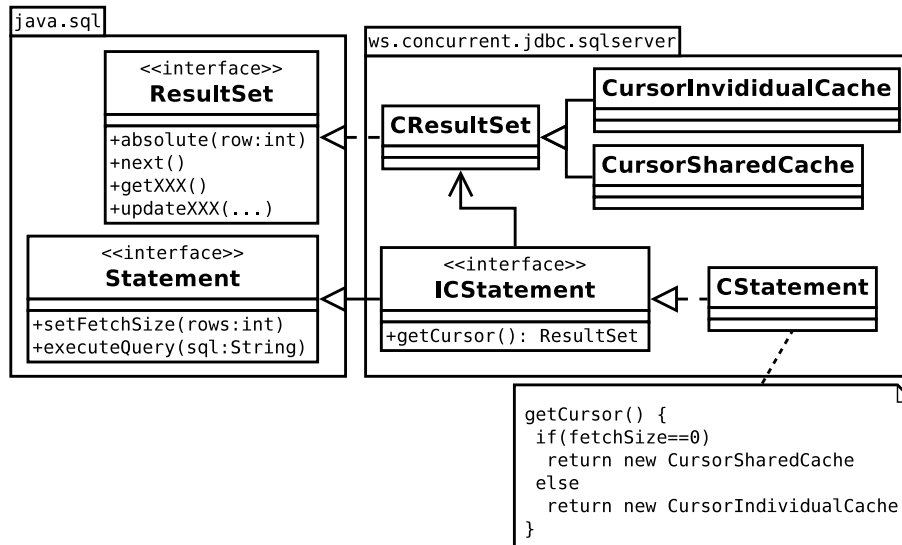
Listagem 3.2: *Obtenção de uma ligação.*

```
String url = ...  
2 Connection con = DriverManager.getConnection(url);
```

Em que `url` é uma `String` no formato descrito na secção 1.4.2.

O que é novo é a possibilidade de criar vários objectos result set (cursos cliente) que utilizam o mesmo cursor do servidor, operando assim sobre o mesmo dataset. Para tal é necessário obter uma instância da interface *ICStatement*. Esta interface fornece o serviço `getCursor()` que devolve um cursor cliente. A interface *ICStatement* estende a interface *Statement*. A classe que as implementa devolve um cursor com cache partilhado (3.3.3) quando o valor de *fetch size* é igual a zero, e devolve um cursor com cache individual (3.3.2) quando o valor de *fetch size* é superior a zero (um valor inferior a zero é inválido). Uma vez que por pré-definição o valor de *fetch size* é superior a zero, se o método `setFetchSize` da interface `ResultSet` não for explicitamente invocado então será criado um cursor com cache individual. Será então boa prática utilizar o método `setFetchSize` explicitamente, para evitar confusões.

Figura 3.8: *Classes principais na utilização da solução Cursor.*



As listagens 3.3 e 3.4 exemplificam a criação de um cursor cliente com cache partilhado e de um cursor cliente com cache individual, respectivamente.

Listagem 3.3: *Criação de um cursor cliente com cache partilhado.*

```

stmt = con.createStatement();
2 // Com fetchSize igual a zero todas as linhas do result set irão ser colocadas em cache.
  stmt.setFetchSize(0);
4 stmt.executeQuery(sqlQuery);

6 ResultSet cursor = ((ICStatement) stmt).getCursor();
  
```

Listagem 3.4: *Criação de um cursor cliente com cache individual.*

```

1 stmt = con.createStatement();
  // Cache individual com capacidade igual a 25 linhas.
3 stmt.setFetchSize(25);
  stmt.executeQuery(sqlQuery);
5

ResultSet cursor = ((ICStatement) stmt).getCursor();
  
```

Com a exceção da última linha de cada listagem, a criação é igual para ambos e deve ser bastante familiar e intuitiva para quem já utiliza a API JDBC. A partir da criação, cada cursor cliente pode ser utilizado como um ResultSet, tal como está definido na documentação da API [47].

Capítulo 4

Benchmark

Uma vez implementadas as soluções, é necessário realizar uma avaliação do seu desempenho. Para tal foi criado um *Domain-Specific Benchmark*[59], que para ser útil tem que ser:

Relevante: Tem que medir o desempenho máximo que um sistema apresenta durante uma operação típica.

Portável: Deve ser fácil de realizar em diferentes sistemas e arquitecturas.

Escalável: Deve ser aplicável a pequenos e a grandes sistemas.

Simples: Tem que ser facilmente compreensível.

Foi com estas características em mente que o benchmark foi construído.

4.1 Benchmark principal

O benchmark consiste em criar um determinado número de threads e medir o tempo total que eles levam a executar a sua tarefa. A tarefa é atribuída no momento da criação dos threads e corresponde à execução de um contexto.

Foram realizados testes nos contextos: **leitura**, **actualização**, **inserção** e **remoção**. O código executado para cada contexto encontra-se nas listagens 4.1 até 4.4.

Estes contextos foram testados num cenário que tenta representar uma situação de utilização de um result set por vários threads. O cenário consiste em dividir logicamente um result set e atribuir uma gama de linhas a cada thread. Na sua execução, cada thread realiza a sua tarefa sobre as linhas lhe são destinadas. De notar que este é um cenário de vários possíveis.

Listagem 4.1: *Contexto de leitura.*

```
1 int firstLine = counter.getAndAdd(linesPerThread);  
  int lastLine = firstLine + linesPerThread - 1;
```

3

```

    rs.absolute(firstLine - 1);
5 for (int i = firstLine; i <= lastLine; ++i) {
    rs.next();
7    val1 = rs.getInt(1);
    val2 = rs.getString(2);
9    val3 = rs.getString(3);
    val4 = rs.getDate(4);
11   val5 = rs.getDouble(5);
    val6 = rs.getInt(6);
13   val7 = rs.getDate(7);
    val8 = rs.getString(8);
15 }

```

Listagem 4.2: *Contexto de actualização.*

```

initUpdateValues();
2 int firstLine = counter.getAndAdd(linesPerThread);
  int lastLine = firstLine + linesPerThread - 1;
4
  rs.absolute(firstLine - 1);
6 for (int i = firstLine; i <= lastLine; ++i) {
    rs.next();
8    rs.updateInt(1, val1);
    rs.updateString(2, val2);
10   rs.updateString(3, val3);
    rs.updateDate(4, val4);
12   rs.updateDouble(5, val5);
    rs.updateInt(6, val6);
14   rs.updateDate(7, val7);
    rs.updateString(8, val8);
16   rs.updateRow();
  }

```

Listagem 4.3: *Contexto de inserção.*

```

1 initUpdateValues();
  for (int i = 0; i < linesPerThread; ++i) {
3    rs.moveToInsertRow();
    rs.updateInt(1, val1);
5    rs.updateString(2, val2);
    rs.updateString(3, val3);
7    rs.updateDate(4, val4);
    rs.updateDouble(5, val5);
9    rs.updateInt(6, val6);
    rs.updateDate(7, val7);
11   rs.updateString(8, val8);
    rs.insertRow();
13   rs.moveToCurrentRow();
  }

```

```

1 int firstLine = counter.getAndAdd(linesPerThread);
  int lastLine = firstLine + linesPerThread - 1;
3
  rs.absolute(firstLine - 1);
5 for (int i = firstLine; i <= lastLine; ++i) {
    rs.next();
7   rs.deleteRow();
  }

```

A variável inteira `linesPerThread` possui o número de linhas atribuídas a cada thread, e o seu valor é atribuído directamente como parâmetro de teste. O número de linhas da tabela é igual ao número de linhas por thread multiplicado pelo número de threads. A variável `counter` é uma instância da classe *AtomicInteger* [49], e é-lhe atribuído o valor 1 no início de cada teste. A variável `counter` auxilia o processo de divisão do result set em gamas. Cada thread obtém o valor da variável, que corresponde ao número da primeira linha da gama, e calcula o valor do início da próxima gama. O valor do número da última linha da gama é calculado adicionando o número de linhas por thread ao valor da primeira linha da gama.

Foram realizados benchmarks com o driver da Microsoft (disponível em [10]), com a solução *ResultSet Wrapper* e a com solução *Cursor*.

De modo a apresentar e discutir os resultados de modo mais sintético, utilizaram-se os seguintes nomes para identificar cada solução:

- *MSJDBC*: Teste usando o driver da Microsoft;
- *CJDBC_I*: Teste usando o driver implementado, na versão com cache individual;
- *CJDBC_S*: Teste usando o driver implementado, na versão com cache partilhado;
- *WJDBC*: Teste usando o driver da Microsoft, na versão *Wrapper*;

O benchmark do *MSJDBC* cria um result set para cada thread, porque o result set não pode ser correctamente partilhado pelas entidades concorrentes. Para o benchmark do *WJDBC* é criado um result set do driver da Microsoft que depois é partilhado entre os threads, e cada thread acede ao result set através de um *wrapper*. Para os benchmarks do *CJDBC_I* e *CJDBC_S* é criado um cursor cliente do result set para cada thread.

O tipo de result set dos testes será sempre `TYPE_SCROLLABLE_SENSITIVE`. Uma vez que um cursor *forward-only* só permite percorrer o dataset numa direcção, não é possível recuperar o contexto, nem existe a possibilidade de atribuir uma porção do dataset a cada thread, pois este não pode mover o result set para uma linha específica (a operação `absolute` não está disponível). A solução *ResultSet Wrapper* tem como pedra basilar o restauro do contexto, e como já foi dito não é possível restaurar o contexto a não ser que o result set seja *scrollable*; por estes motivos não são realizados testes com o tipo de result set `TYPE_FORWARD_ONLY`. No contexto da Leitura o tipo de concorrência do result set é `CONCUR_READ_ONLY` e nos contextos da Actualização, Inserção e Remoção a concorrência é do tipo `CONCUR_UPDATABLE`.

Antes da execução propriamente dita do benchmark é necessário proceder à preparação da base de dados, pois alguns testes de desempenho necessitam que existam dados na tabela de testes.

Todas as statement SQL utilizadas nos testes de desempenho para criar os result set foram realizadas sobre a tabela representada na Figura 4.1.

Figura 4.1: Tabela utilizada no benchmark.

Std_Student
Std_id: INT PRIMARY KEY
Std_firstName: NVARCHAR(25)
Std_lastName: NVARCHAR(50)
Std_RegDate: DATETIME
Std_applGrade: FLOAT
StdCrs_id: INT
Std_birthDate: DATETIME
Std_eMail: NVARCHAR(25)

As listagens 4.5 e 4.6 apresentam os passos que são efectuados na preparação da base de dados dos testes de desempenho para os diferentes contextos.

Os contextos de Leitura, Actualização e Remoção limpam os dados da tabela, e introduzem novos dados antes de executar o benchmark. O contexto de Inserção só limpa os dados da tabela antes de executar o benchmark, já que a própria execução consiste em introduzir dados na tabela.

Listagem 4.5: Preparação para executar os contextos de Leitura, Actualização e Remoção.

```

1 Limpar a tabela de testes
  Introduzir novos valores na tabela de testes
3 Executar o benchmark

```

Listagem 4.6: Preparação para executar o contexto de Inserção.

```

1 Limpar a tabela de testes
  Executar o benchmark

```

Na execução do benchmark são medidos dois tempos: **tempo de preparação** (fig. 4.2), que corresponde ao tempo necessário para executar as *queries* e criar os threads, e **tempo de execução** (fig. 4.3), que corresponde ao tempo total que os threads criados levam a terminar a sua tarefa.

Figura 4.2: Medição do tempo de preparação

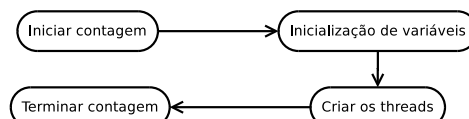
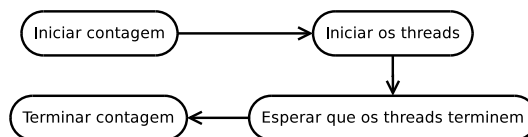


Figura 4.3: *Medição do tempo de execução*



Na inicialização de variáveis são criados os result set apropriados a cada situação e definidos parâmetros tais como o número de linhas por thread. Para o *MSJDBC* neste passo não é criado algum result set, pois cada thread é que se encarrega de criar o seu próprio result set. Para o *WJDBC* este passo também incluir criar a fábrica de cursores.

Na criação dos threads é atribuída uma tarefa (Runnable) a cada thread. A tarefa está relacionada com o contexto. Os threads no *MSJDBC* criam agora o seu result set e os threads das restantes soluções obtêm o cursor cliente para o result set criado no passo anterior.

Os resultados deste benchmark são apresentados nas secções 5.2 e 5.2.2, e discutidos nas secções 6.1.1 e 6.1.2.

Para além do benchmark descrito até aqui, foram ainda criados dois benchmarks adicionais que testam situações particulares, e que serão descritos nas secções seguintes.

4.2 Benchmark com atrasos

A motivação para realizar este benchmark é que o cenário do benchmark principal pode ser considerado irrealista; normalmente existe algum processamento associado à leitura e escrita de linhas, e admite-se que a introdução desse processamento possa vir a alterar os resultados. Por isso resolveu-se introduzir a realização de uma tarefa entre operações de leitura/escrita, que provocará um atraso entre as invocações sucessivas de operações do ResultSet. A introdução do atraso é realizada em dois locais: entre colunas e entre linhas; ver-se-á assim se algum dos dois tem algum efeito nos resultados finais do benchmark. O resultados para os dois locais foram obtidos em dois benchmarks separados.

Neste benchmark apenas o contexto de leitura e de actualização foram testados. O contexto de inserção do ponto de vista operacional é semelhante ao contexto de actualização, pelo que os seus resultados seriam redundantes. O contexto de remoção não realiza processamento dos valores das colunas, e é por isso excluído. As listagens 4.7, 4.8, 4.9 e 4.10 apresentam o código executado por cada thread para os diferentes contextos.

Listagem 4.7: *Contexto de leitura com atraso entre colunas.*

```
int firstLine = counter.getAndAdd(linesPerThread);
2 int lastLine = firstLine + linesPerThread - 1;
  rs.absolute(firstLine - 1);
4 for (int i = firstLine; i <= lastLine; ++i) {
    rs.next();
6    val1 = rs.getInt(1);
```

```

        ResultSetRunnable.delayfunc();
8    val2 = rs.getString(2);
        ResultSetRunnable.delayfunc();
10   val3 = rs.getString(3);
        ResultSetRunnable.delayfunc();
12   val4 = rs.getDate(4);
        ResultSetRunnable.delayfunc();
14   val5 = rs.getDouble(5);
        ResultSetRunnable.delayfunc();
16   val6 = rs.getInt(6);
        ResultSetRunnable.delayfunc();
18   val7 = rs.getDate(7);
        ResultSetRunnable.delayfunc();
20   val8 = rs.getString(8);
        ResultSetRunnable.delayfunc();
22 }

```

Listagem 4.8: *Contexto de leitura com atraso entre linhas.*

```

1  int firstLine = counter.getAndAdd(linesPerThread);
   int lastLine = firstLine + linesPerThread - 1;
3  rs.absolute(firstLine - 1);
   for (int i = firstLine; i <= lastLine; ++i) {
5      rs.next();
       val1 = rs.getInt(1);
7      val2 = rs.getString(2);
       val3 = rs.getString(3);
9      val4 = rs.getDate(4);
       val5 = rs.getDouble(5);
11     val6 = rs.getInt(6);
       val7 = rs.getDate(7);
13     val8 = rs.getString(8);

15     ResultSetRunnable.delayfunc();
   }

```

Listagem 4.9: *Contexto de actualização com atraso entre colunas.*

```

1  initUpdateValues();
   int firstLine = counter.getAndAdd(linesPerThread);
3  int lastLine = firstLine + linesPerThread - 1;
   rs.absolute(firstLine - 1);
5  for (int i = firstLine; i <= lastLine; ++i) {
       rs.next();
7       ResultSetRunnable.delayfunc();
       rs.updateString(2, val2);
9       ResultSetRunnable.delayfunc();
       rs.updateString(3, val3);
11      ResultSetRunnable.delayfunc();
       rs.updateDate(4, val4);
13      ResultSetRunnable.delayfunc();

```



```

        rs.updateDouble(5, val5);
15    ResultSetRunnable.delayfunc();
        rs.updateInt(6, val6);
17    ResultSetRunnable.delayfunc();
        rs.updateDate(7, val7);
19    ResultSetRunnable.delayfunc();
        rs.updateString(8, val8);
21    ResultSetRunnable.delayfunc();
        rs.updateRow();
23 }

```

Listagem 4.10: *Contexto de actualização com a atraso entre linhas.*

```

    initUpdateValues();
2  int firstLine = counter.getAndAdd(linesPerThread);
    int lastLine = firstLine + linesPerThread - 1;
4  rs.absolute(firstLine - 1);
    for (int i = firstLine; i <= lastLine; ++i) {
6      rs.next();

8      ResultSetRunnable.delayfunc();

10     rs.updateString(2, val2);
        rs.updateString(3, val3);
12     rs.updateDate(4, val4);
        rs.updateDouble(5, val5);
14     rs.updateInt(6, val6);
        rs.updateDate(7, val7);
16     rs.updateString(8, val8);

18     rs.updateRow();
    }

```

O código dos contextos deste benchmark é muito semelhante ao código do benchmark principal, a diferença fundamental está na invocação do método `delayfunc()` da classe `ResultSetRunnable` entre cada operação sobre uma coluna para introduzir atraso entre colunas, ou entre cada linha lida para introduzir atraso entre linhas. O código executado por este método encontra-se na listagem 4.11. Realizando diversos testes, verificou-se que o método introduz o atraso de aproximadamente `delayms`. A variável `delayms` pertence à classe `ResultSetRunnable`, e o seu valor pode ser alterado usando o método `setDelay` dessa mesma classe.

Listagem 4.11: *Método de atraso.*

```

1  final static Object = new Object();
    public static void delayfunc() {
3      synchronized(obj) {
        obj.wait(delayms);
5      }
    }

```

Os resultados deste benchmark são apresentados na secção 5.3, e discutidos na secção 6.1.3.

4.3 Cache individual vs Cache partilhado

Foi realizado um benchmark suplementar comparando a implementação do cursor cliente com cache individual com a implementação do cursor cliente com cache partilhado. Este benchmark tem o intuito de justificar a existência de ambos como solução ao problema, e ainda justificar a decisão de utilizar um *fetch size* de 100% para o cache partilhado.

O benchmark contemplou apenas o contexto de leitura, pelo que o código executado pelos threads no processo é o mesmo que foi apresentado na listagem 4.1. Decidiu-se assim porque o contexto de leitura é o que mais directamente depende da implementação do cache, e por isso é também o mais relevante para testar.

A ideia principal deste benchmark é verificar o efeito que vários valores de *fetch size* têm em cada implementação do cache. Na medida em que a implementação do cache partilhado por pré-definição carrega todas as linhas da tabela para o cache, isto é, tem um *fetch size* de 100%, foi necessário realizar uma modificação à implementação, de modo a ser possível trabalhar com um cache partilhado que não escolhe todas as linhas; esta implementação é denominada por *CJDBC_SM*, a implementação do cache individual é denominada por *CJDBC_I* e a implementação normal do cache partilhado é denominada por *CJDBC_S*.

O procedimento passou por calcular o valor de linhas para o *fetch size* aplicando uma percentagem ao número total de linhas presentes na tabela. As várias percentagens definem os contextos, e são as seguintes: **fetch size 10%**, **fetch size 20%**, **fetch size 50%**, **fetch size 75%** e **fetch size 100%**. No entanto, nos resultados e nas discussões utiliza-se a notação mais compacta: **10**, **20**, **50**, **75** e **100** para designar os contextos.

O cenário de teste é o mesmo que foi utilizado no benchmark principal: a cada thread é atribuída uma gama de linhas sobre a qual ele deve realizar a tarefa que lhe foi atribuída.

Os resultados deste benchmark são apresentados na secção 5.4, e discutidos na secção 6.1.4.

Capítulo 5

Resultados

Neste capítulo, primeiro são apresentadas as características do sistema computacional utilizado para realizar os benchmarks, e a seguir são apresentados os resultados para os benchmarks descritos no capítulo 4.

Os gráficos apresentados refletem os resultados comparando as soluções aos pares efectuando um rácio. Isto permite perceber qual a solução que melhor responde à variação dos parâmetros do benchmark. A variável independente é a quantidade de threads, a variável dependente é o rácio do tempos obtidos para cada solução e cada série representa o número de linhas que é atribuído a cada thread. Cada thread executou a tarefa associada a um determinado contexto sobre esse número de linhas.

São apresentados os resultados obtidos para três tempos: execução (ExecT), preparação (SetupT) e total (Total).

5.1 Plataforma de teste

Os resultados apresentados foram obtidos usando a seguinte plataforma:

- **Cliente:** Intel® Core™ 2 Duo P8600 @ 2.40GHz, 4GB DDR2
- **Servidor:** Inter® Pentium™ SU4100 @ 1.30GHz, 4GB DDR3, Disco Rígido SATA II 7200 RPM
- SQL Server 2008 versão 10.0.1600
- Java 1.6.0.17
- Microsoft SQL Server JDBC Driver 3.0 (sqljdbc4 - versão 3.0.1301.101, Abril de 2010)

Foram registadas **30 amostras** para cada teste. Para dar maior valor estatístico ao resultados as 10 melhores amostras e as 10 piores foram ignoradas, e os resultados reflectem portanto a média das restantes amostras. A razão para não ter em conta algumas amostras deve-se ao facto que podem acontecerem condições favoráveis ou desfavoráveis, externas ao

ambiente de teste, que alteram o comportamento habitual. Assim, descartando estes resultados anómalos, obtém-se uma melhor estimativa. De notar que a opção por este processo surge de observação empírica.

Entre o registo de cada amostra foi executado o garbage collector com o intuito de reciclar os objectos anteriores que já não estejam a ser utilizados, libertando memória [56, 60].

Foi criada uma base de dados no SQL Server para registar os resultados e fornecer dados para a execução dos cenários de teste, com um *modelo de recuperação simples* e com a opção de *estatísticas automáticas* desactivada. Estas opções servem para diminuir o impacto que o SQL Server possa ter nos resultados.

5.2 Benchmark principal

5.2.1 Comparação com MSJDBC

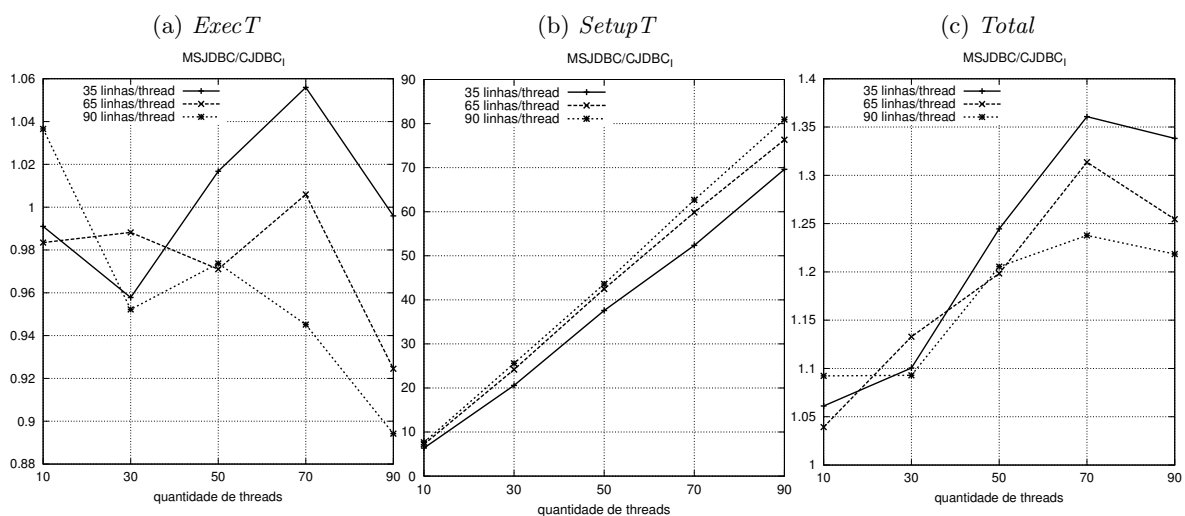
Esta secção apresenta os resultados do benchmark principal (secção 4.1). Os resultados são apresentados sob a forma de uma comparação entre o desempenho registado para o *MSJDBC* e as restantes soluções. A comparação é realizada através de um rácio dos tempos registados.

5.2.1.1 Actualização

5.2.1.1.1 $CJDBC_I$

A Figura 5.1 apresenta os resultados da comparação do desempenho de *MSJDBC* e de *CJDBC_I*, no contexto Actualização.

Figura 5.1: Comparação entre *MSJDBC* e *CJDBC_I*, no contexto **Actualização**.



Em relação ao tempo de execução, o *CJDBC_I* apresenta um desempenho um pouco inferior ao de *JDBC*, principalmente para os valores do número de linhas por thread maiores.

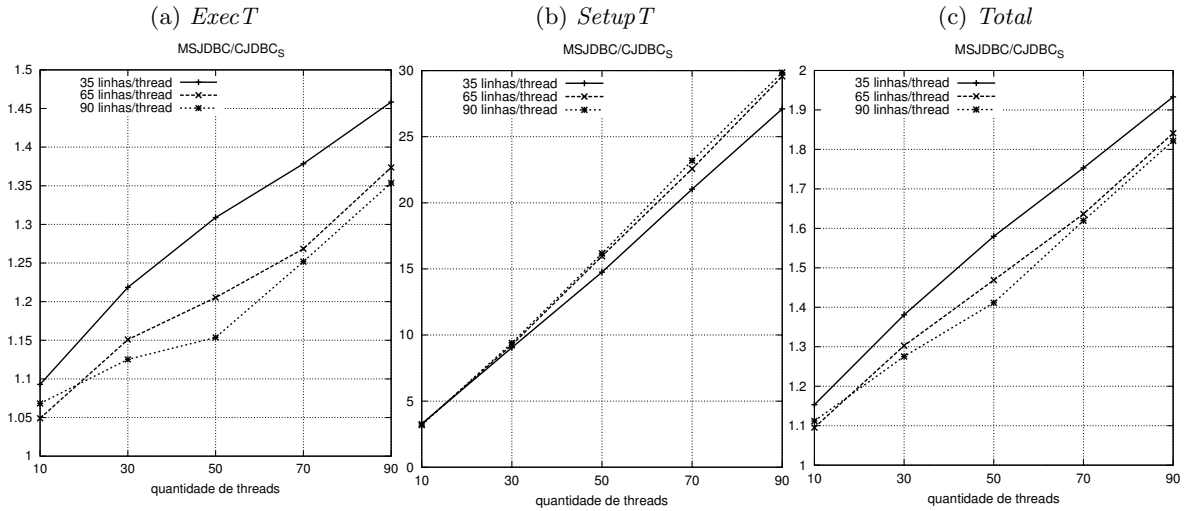
O aumento da quantidade de threads penaliza mais o desempenho da solução $CJDBC_I$.

Com a ajuda do tempo de preparação, o $CJDBC_I$ consegue ser sempre mais rápido do que $JDBC$ no que se refere ao tempo total. Neste caso o aumento da quantidade de threads beneficia o desempenho do $CJDBC_I$, e para quantidades de threads baixas os resultados dos valores do número de linhas por thread são aproximados, diferenciando-se um pouco para as quantidades de threads mais altas, em que os valores de linhas mais altos penalizam mais o desempenho do $CJDBC_I$.

5.2.1.1.2 $CJDBC_S$

A Figura 5.2 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_S$, no contexto Actualização.

Figura 5.2: Comparação entre $MSJDBC$ e $CJDBC_S$, no contexto Actualização.



Aqui o desempenho do $CJDBC_S$ é sempre superior, conseguindo bons resultados totais, pois para as quantidades de threads maiores consegue ser quase duas vezes melhor do que $JDBC$.

Quanto maior é a quantidade de threads melhor é o desempenho de $CJDBC_S$ em relação a $JDBC$.

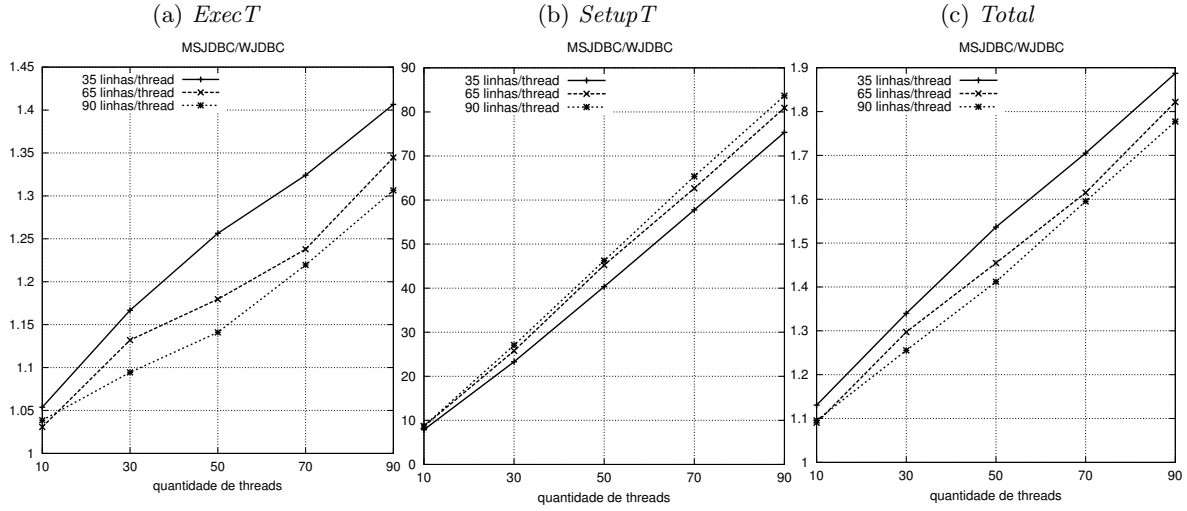
Diferentes números de linhas por thread não implicam alterações significativas ao comportamento geral.

O tempo de preparação tem alguma importância, pois vem aumentar a superioridade do $CJDBC_S$ em relação ao $JDBC$ para os tempos totais.

5.2.1.1.3 $WJDBC$

A Figura 5.3 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $WJDBC$, no contexto Actualização.

Figura 5.3: Comparação entre *MSJDBC* e *WJDBC*, no contexto **Actualização**.



O desempenho do *WJDBC* é sempre melhor, destacando-se para quantidades de threads maiores. A alteração do valor de linhas por thread não provoca diferenças dignas de registo.

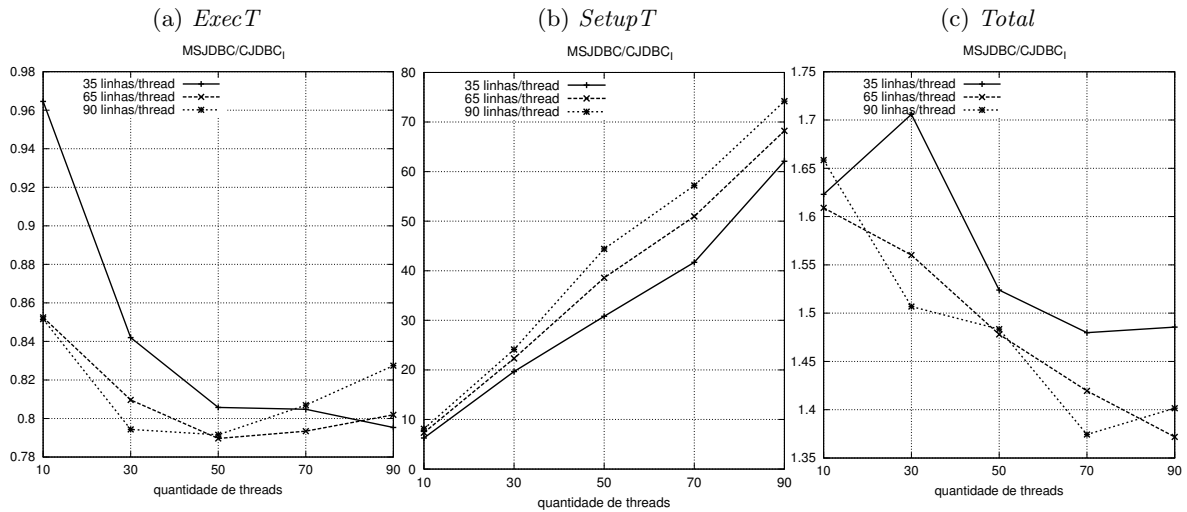
O tempo de preparação acentua a vantagem que o *WJDBC* tem sobre o *JDBC* no tempo total.

5.2.1.2 Leitura

5.2.1.2.1 *CJDBC_I*

A Figura 5.4 apresenta os resultados da comparação do desempenho de *MSJDBC* e de *CJDBC_I*, no contexto Leitura.

Figura 5.4: Comparação entre *MSJDBC* e *CJDBC_I*, no contexto **Leitura**.



O desempenho do *CJDBC_I* é superior ao de *JDBC* para o tempo total, já no tempo de

execução verifica-se o contrário, embora a desvantagem não seja grande.

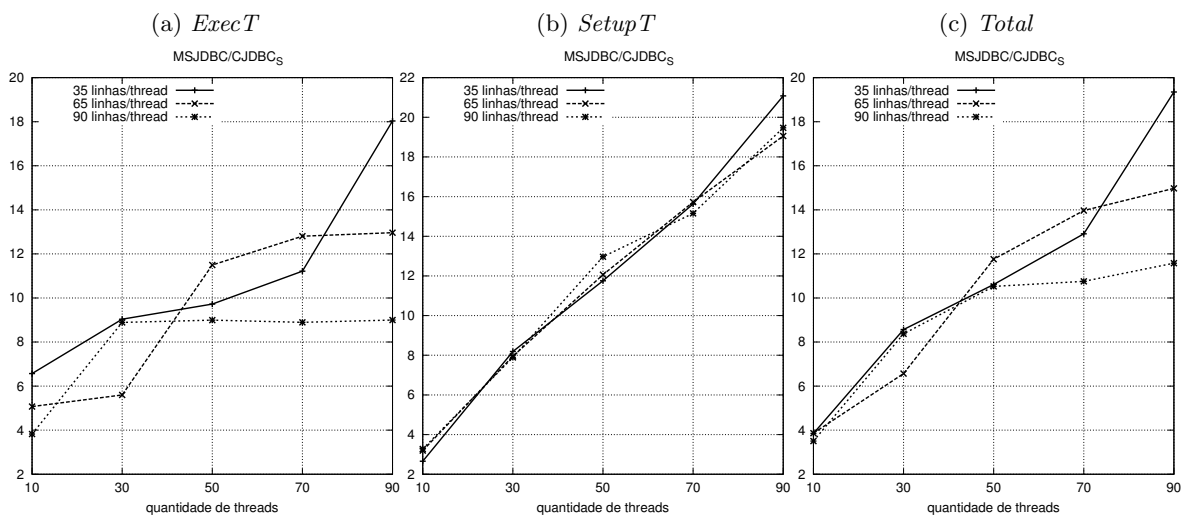
Nota-se uma descida do desempenho do $CJDBC_I$ em relação ao $JDBC$ para quantidades de threads maiores.

Os valores de número de linhas por thread maiores prejudicam um pouco o desempenho do $CJDBC_I$ em relação ao desempenho do $JDBC$.

5.2.1.2.2 $CJDBC_S$

A Figura 5.5 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_S$, no contexto Leitura.

Figura 5.5: Comparação entre $MSJDBC$ e $CJDBC_S$, no contexto Leitura.



Basicamente, o desempenho do $CJDBC_S$ é muito bom comparativamente ao de $JDBC$ sendo, para o tempo total, no mínimo 4 vezes melhor e no máximo consegue ser quase 20 vezes melhor.

Os valores do número de linhas por thread maiores penalizam mais o desempenho do $CJDBC_S$, embora não gravosamente.

O aumento da quantidade threads beneficia o desempenho do $CJDBC_S$.

5.2.1.2.3 $WJDBC$

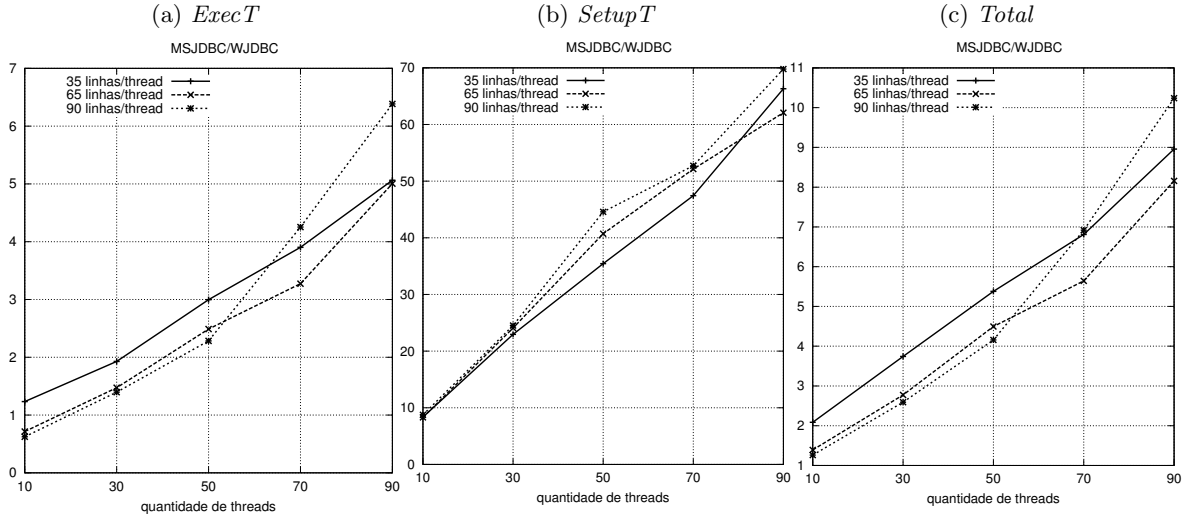
A Figura 5.6 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $WJDBC$, no contexto Leitura.

O desempenho do $WJDBC$ é sempre muito melhor do que o desempenho do $JDBC$, chegando a ser 10 vezes melhor do tempo total.

As quantidades de threads maiores beneficiam o desempenho do $WJDBC$.

O aumento do número de linhas por thread começa a dar vantagem ao desempenho de $WJDBC$ à medida que o número de threads também aumenta.

Figura 5.6: Comparação entre *MSJDBC* e *WJDBC*, no contexto **Leitura**.



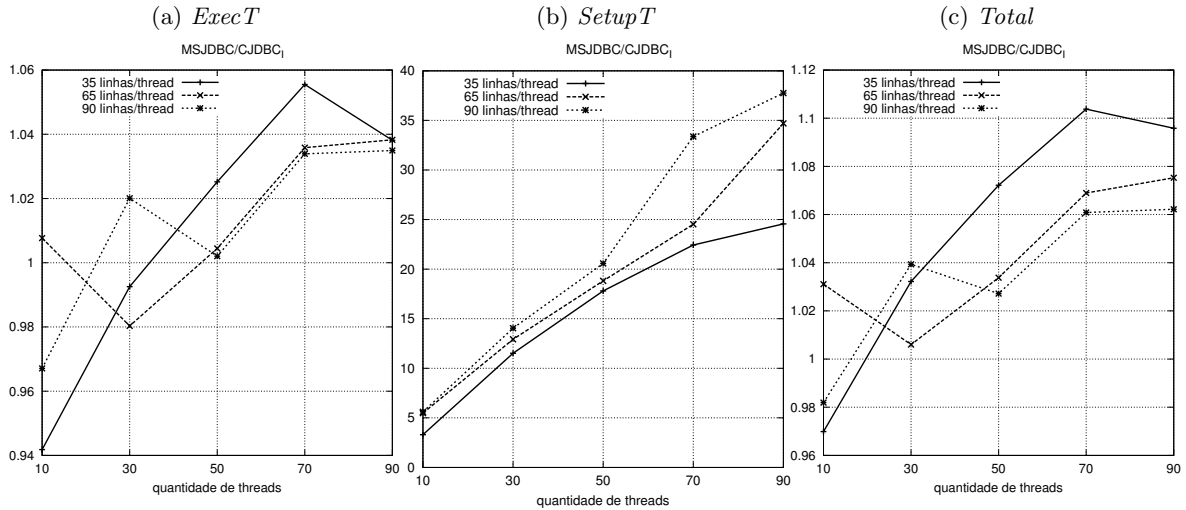
O tempo de preparação realça mais a vantagem do *WJDBC* no tempo total.

5.2.1.3 Inserção

5.2.1.3.1 *CJDBC_I*

A Figura 5.7 apresenta os resultados da comparação do desempenho de *MSJDBC* e de *CJDBC_I*, no contexto Inserção.

Figura 5.7: Comparação entre *MSJDBC* e *CJDBC_I*, no contexto **Inserção**.



Praticamente sempre, o desempenho do *CJDBC_I* é ligeiramente superior ao de *JDBC*.

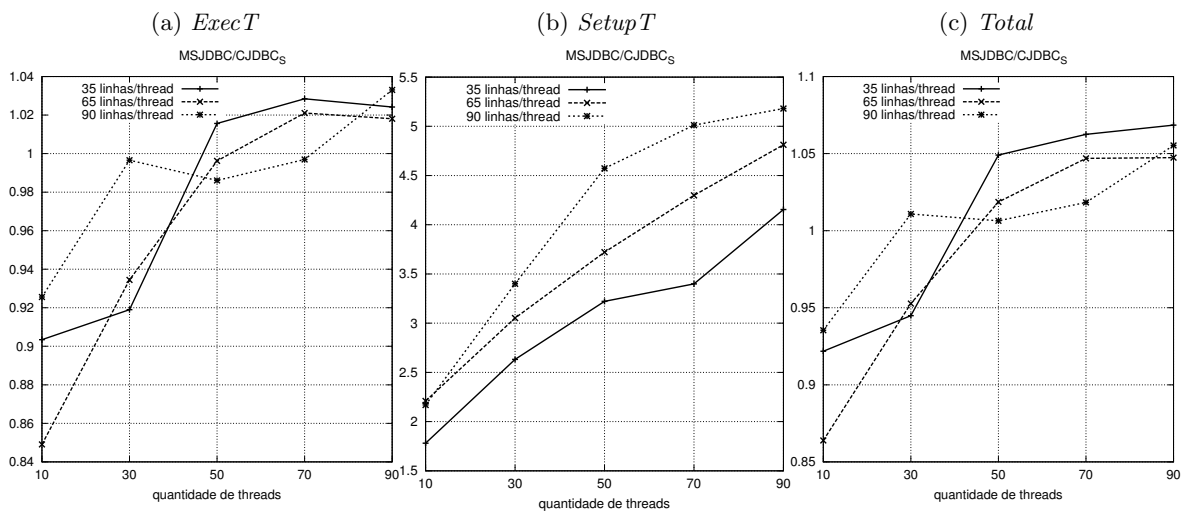
A atribuição de um maior número de linhas por thread, não tem grande peso para o desempenho, e o facto da quantidade de threads aumentar dá uma pequena vantagem ao *CJDBC_I*.

Os resultados dos tempos de execução e total são semelhantes, pelo que se pode dizer que o tempo de preparação tem pouca influência para o resultado final.

5.2.1.3.2 $CJDBC_S$

A Figura 5.8 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_S$, no contexto Inserção.

Figura 5.8: Comparação entre $MSJDBC$ e $CJDBC_S$, no contexto **Inserção**.



No geral o desempenho do $CJDBC_S$ é marginalmente melhor do que o desempenho do $JDBC$.

Para um número de threads maior a vantagem de $CJDBC_S$ sobre $JDBC$ acentua-se.

Os valores do número de linhas por thread menores apresentam um melhor desempenho comparativo em favor de $CJDBC_S$, do que valores maiores.

5.2.1.3.3 $WJDBC$

A Figura 5.9 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $WJDBC$, no contexto Inserção.

O desempenho do $WJDBC$ é melhor para os valores do número de linhas por thread menores, enquanto que para valores maiores a vantagem é do $JDBC$.

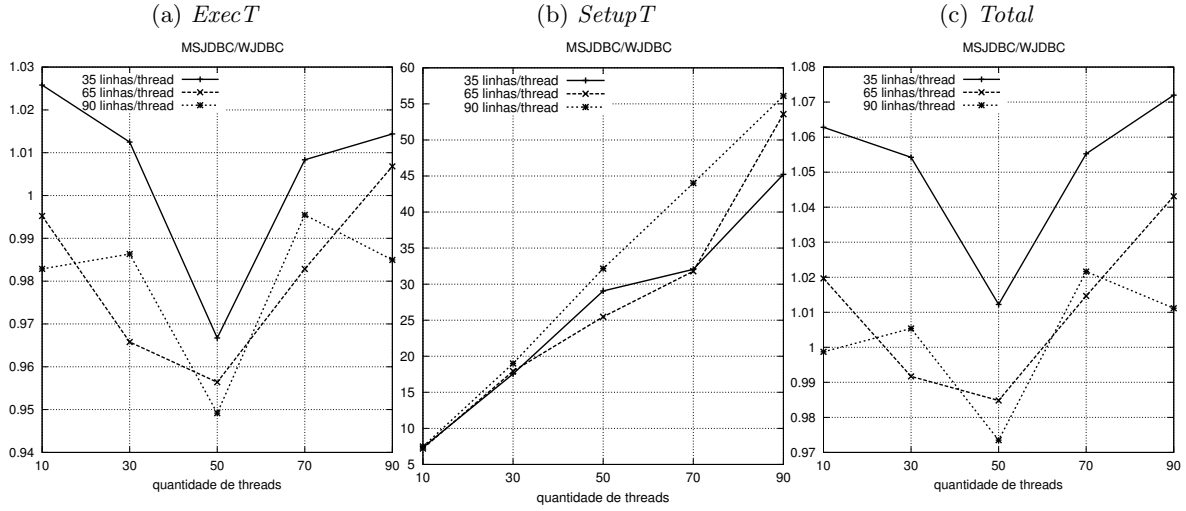
O aumento da quantidade de threads não influencia com grande peso os resultados.

O tempo de preparação tem pouca influência no tempo total.

5.2.1.4 Remoção

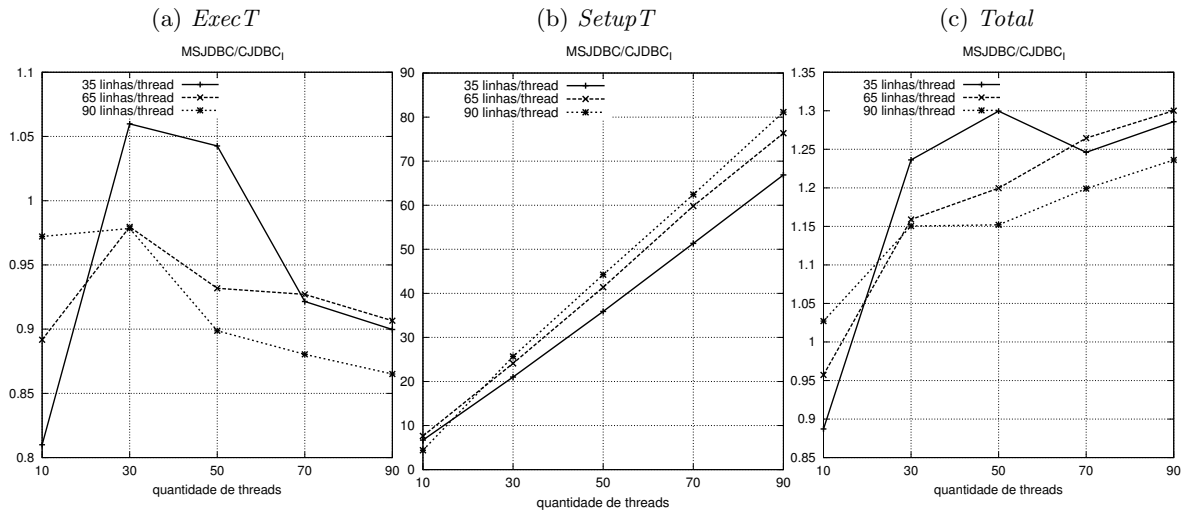
5.2.1.4.1 $CJDBC_I$

Figura 5.9: Comparação entre *MSJDBC* e *WJDBC*, no contexto **Inserção**.



A Figura 5.10 apresenta os resultados da comparação do desempenho de *MSJDBC* e de *CJDBC_I*, no contexto Remoção.

Figura 5.10: Comparação entre *MSJDBC* e *CJDBC_I*, no contexto **Remoção**.



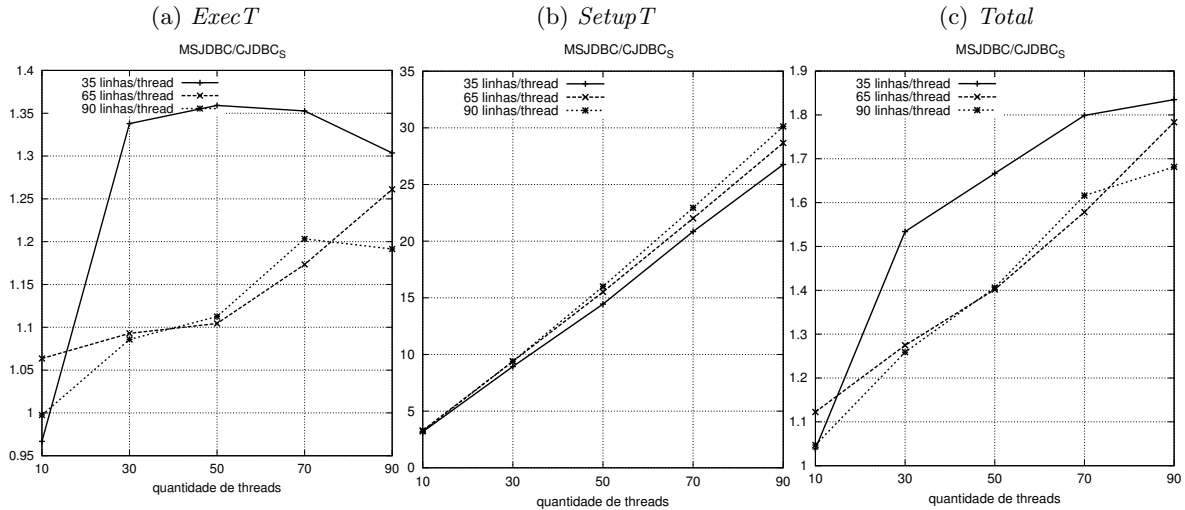
Para o tempo de execução o desempenho do *CJDBC_I* fica um pouco abaixo do desempenho do *JDBC*. Embora a influencia seja pequena, o aumento da quantidade de threads penaliza um pouco o desempenho do *CJDBC_I*, verificando-se o mesmo para os valores maiores do número de linhas por thread.

O tempo de preparação é relevante para o resultado final; para o tempo do total o *CJDBC_I* apresenta melhor desempenho, sendo praticamente sempre superior a *JDBC*. O aumento da quantidade de threads dá vantagem a *CJDBC_I* e o aumento do valor do número de linhas por thread penaliza um pouco o seu desempenho face ao desempenho do *JDBC*.

5.2.1.4.2 $CJDBC_S$

A Figura 5.11 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_S$, no contexto Remoção.

Figura 5.11: Comparação entre $MSJDBC$ e $CJDBC_S$, no contexto Remoção.



O desempenho do $CJDBC_S$ é superior ao de $JDBC$.

Para quantidades de threads menores o desempenho do $CJDBC_S$ é melhor mas a vantagem é moderada, para quantidades maiores o desempenho comparativo dispara atingindo valores muito bons.

A quantidade de linhas por thread, não incute diferenças significativas.

O tempo de preparação tem influência no resultado, acentuando ainda mais a vantagem do $CJDBC_S$ sobre o $JDBC$.

5.2.1.4.3 $WJDBC$

A Figura 5.12 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $WJDBC$, no contexto Remoção.

O desempenho do $WJDBC$ é superior ao de $JDBC$.

O aumento da quantidade de threads beneficia o desempenho do $WJDBC$, embora seja mais notório no tempo total.

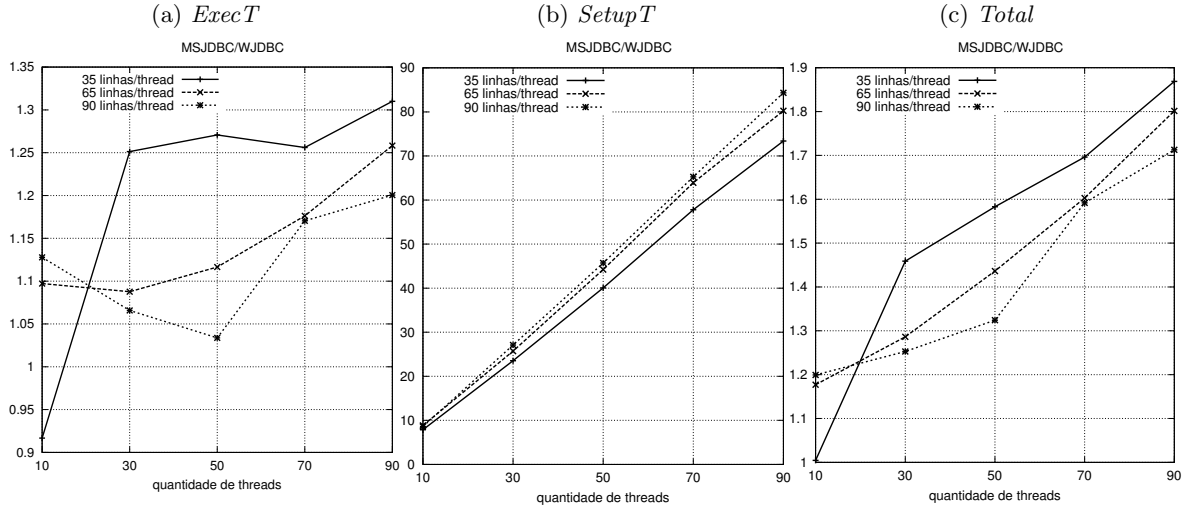
Os valores de número de linhas por thread mais altos beneficiam mais o desempenho do $JDBC$.

Os bons tempos de preparação do $WJDBC$ dão-lhe ainda mais vantagem nos tempos totais sobre o $JDBC$.

5.2.1.5 Resumo

Resumindo os resultados podemos dizer que em **geral**:

Figura 5.12: Comparação entre *MSJDBC* e *WJDBC*, no contexto **Remoção**.



- O desempenho de *CJDBC*¹ e de *WJDBC* é superior ao desempenho de *MSJDBC*;
- O desempenho das soluções *CJDBC* e *WJDBC* melhora em relação ao desempenho de *MSJDBC* com o aumento da quantidade de threads;
- O tempo de preparação de *MSJDBC* é muito superior, sendo por inúmeras vezes 80 vezes mais alto;
- Para o contexto Actualização as soluções *CJDBC_S* e *WJDBC* apresentam o melhor desempenho;
- Para o contexto Inserção a solução *CJDBC_I* apresenta o melhor desempenho;
- Para o contexto Leitura a solução *CJDBC_S* apresenta o melhor desempenho;
- Para o contexto Remoção as soluções *CJDBC_S* e *WJDBC* apresentam o melhor desempenho.

5.2.2 Comparação com *WJDBC*

Esta secção apresenta os resultados do benchmark principal (secção 4.1). Os resultados são apresentados sob a forma de uma comparação entre o desempenho registado para o *WJDBC* e as soluções *CJDBC_I* e *CJDBC_S*. A comparação é realizada através de um rácio dos tempos registados.

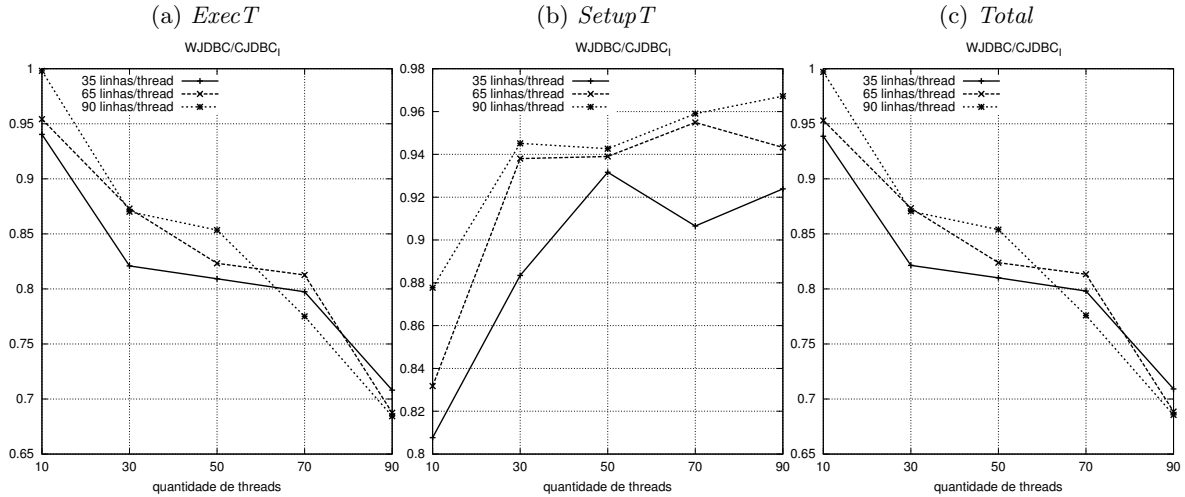
5.2.2.1 Actualização

5.2.2.1.1 *CJDBC_I*

¹*CJDBC* refere-se às soluções *CJDBC_I* e *CJDBC_S*

A Figura 5.13 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_I*, no contexto Actualização.

Figura 5.13: Comparação entre *WJDBC* e *CJDBC_I*, no contexto **Actualização**.



O desempenho do *WJDBC* é sempre melhor do que o desempenho do *CJDBC_I*.

Para os tempos de execução e total o aumento da quantidade de threads penaliza mais o desempenho do *CJDBC_I*, enquanto que para o tempo de preparação a mesma situação beneficia o desempenho do *CJDBC_I*.

Um número maior de linhas por thread não implica alterações importantes na comparação dos desempenhos.

O tempo de preparação tem pouco impacto no tempo total.

5.2.2.1.2 *CJDBC_S*

A Figura 5.14 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_S*, no contexto Actualização.

Ambos têm desempenhos semelhantes, mas no tempo de preparação a vantagem está claramente do lado do *WJDBC*, aumentando com o crescimento da quantidade de threads. Porém este tempo não altera muito o total.

O número de threads ou de linhas por thread não afecta significativamente os resultados.

5.2.2.2 Leitura

5.2.2.2.1 *CJDBC_I*

A Figura 5.15 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_I*, no contexto Leitura.

O desempenho do *CJDBC_I* é superior ao de *WJDBC* para a quantidade de threads menor e número de linhas por thread maior. O *WJDBC* é bastante superior para as restantes

Figura 5.14: Comparação entre $WJDBC$ e $CJDBC_S$, no contexto **Actualização**.

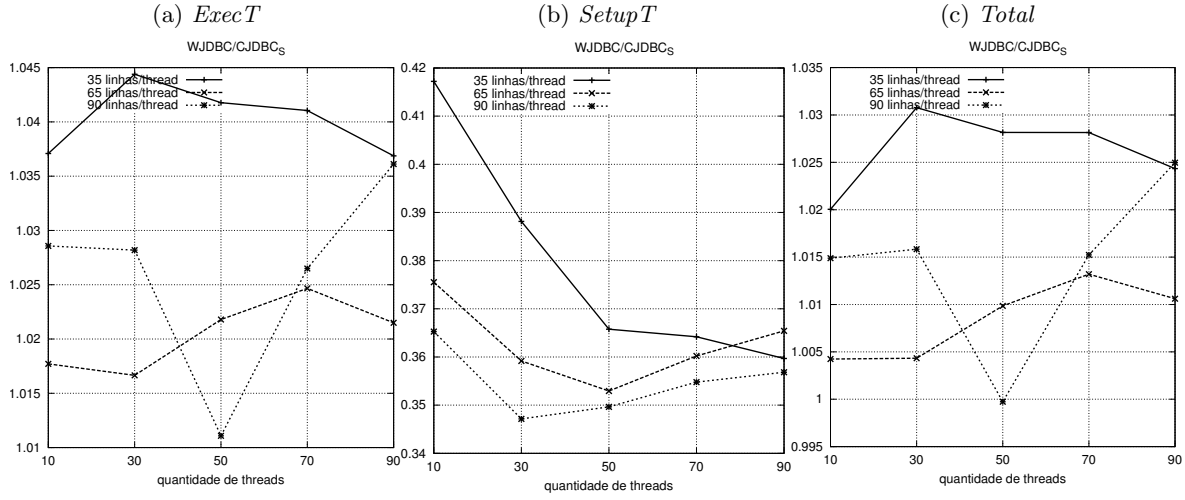
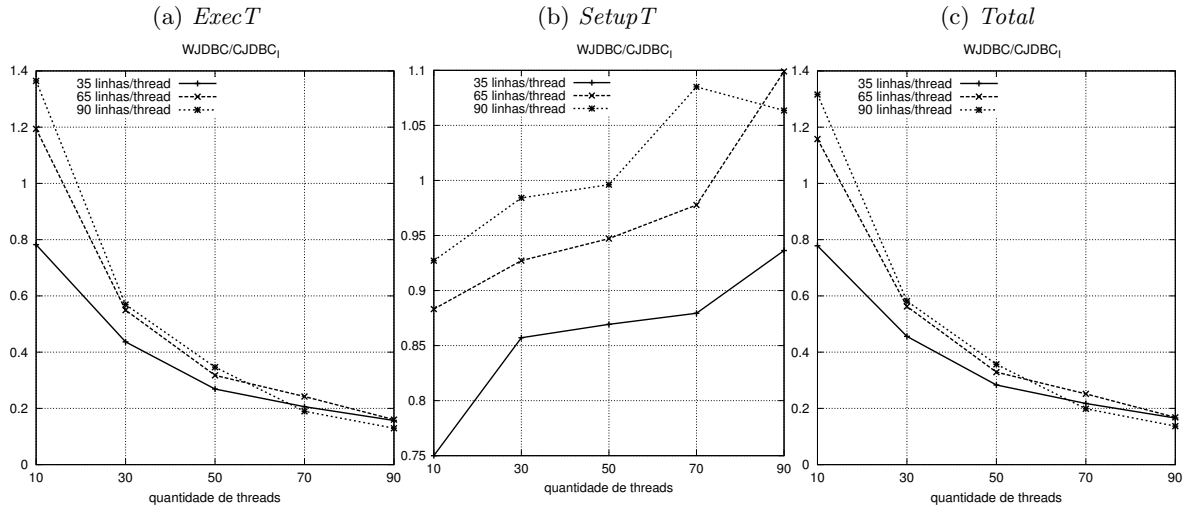


Figura 5.15: Comparação entre $WJDBC$ e $CJDBC_I$, no contexto **Leitura**.



situações.

Uma quantidade de thread maior afecta negativamente o desempenho de $CJDBC_I$.

O número de linhas por thread tem pouco efeito para as quantidades de threads maiores.

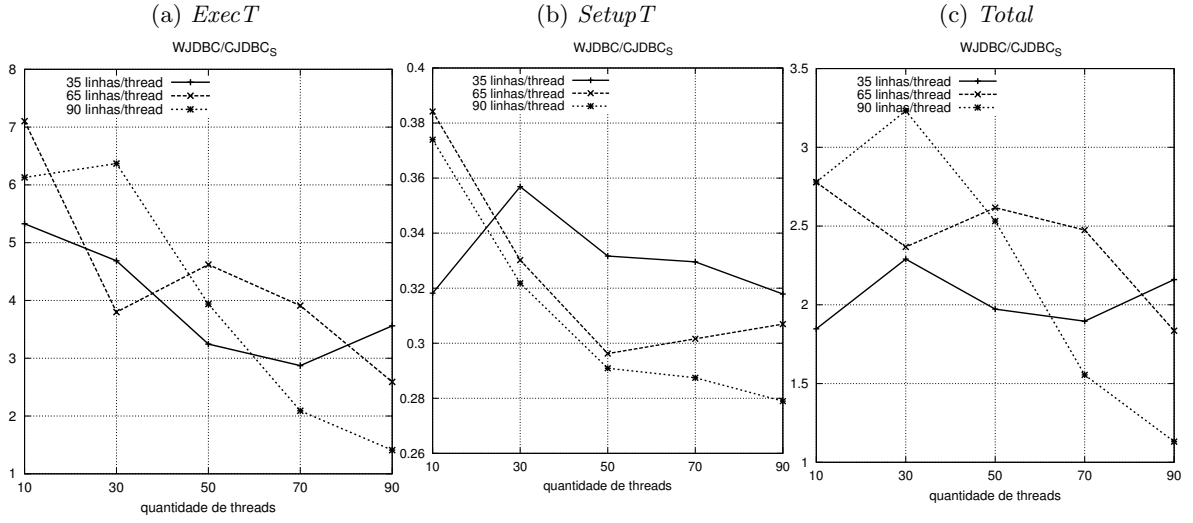
O tempo de preparação é semelhante, havendo uma ligeira vantagem para o $WJDBC$, e tem pouca preponderância no resultado total.

5.2.2.2.2 $CJDBC_S$

A Figura 5.16 apresenta os resultados da comparação do desempenho de $WJDBC$ e de $CJDBC_S$, no contexto Leitura.

O $CJDBC_S$ começa por apresentar um desempenho bastante superior, mas que se degrada rapidamente com o aumento do número de threads. Porém o desempenho de $CJDBC_S$

Figura 5.16: Comparação entre *WJDBC* e *CJDBC_S*, no contexto **Leitura**.



mantém-se superior ao de *WJDBC*.

Os números de linhas por thread maiores favorecem o desempenho do *CJDBC_S* nos tempos de execução e total, e prejudicam-no no tempo de preparação.

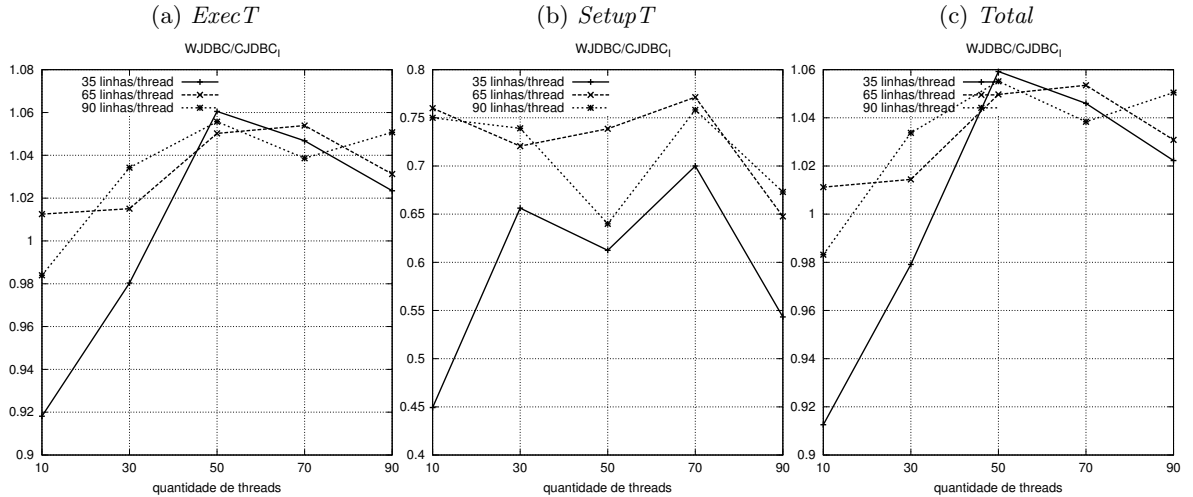
O mau tempo de preparação do *CJDBC_S* faz com que a grande vantagem que tinha no tempo de execução seja reduzida quase para metade no tempo total.

5.2.2.3 Inserção

5.2.2.3.1 *CJDBC_I*

A Figura 5.17 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_I*, no contexto Inserção.

Figura 5.17: Comparação entre *WJDBC* e *CJDBC_I*, no contexto **Inserção**.



O desempenho do *WJDBC* é superior ao de *CJDBC_I* para quantidades de threads mais baixas, enquanto que é o *CJDBC_I* que ganha para as quantidades mais altas, mas com uma ligeira vantagem.

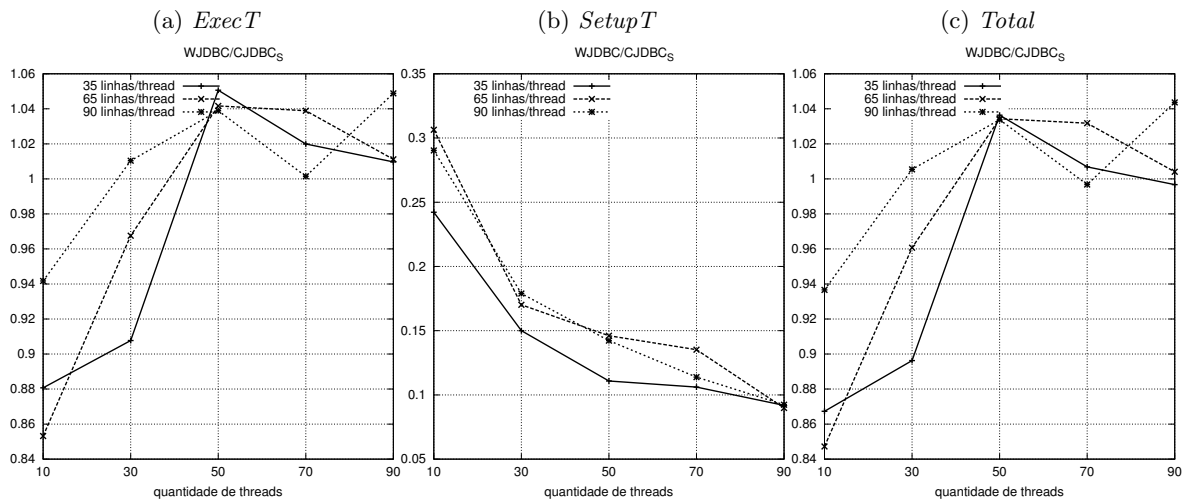
O tempo de preparação do *WJDBC* é significativamente melhor, principalmente para os valores de número de linhas mais baixos.

No geral o número de linhas por thread não afecta significativamente a comparação.

5.2.2.3.2 *CJDBC_S*

A Figura 5.18 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_S*, no contexto Inserção.

Figura 5.18: Comparação entre *WJDBC* e *CJDBC_S*, no contexto **Inserção**.



O desempenho do *WJDBC* é superior ao de *CJDBC_S* para as quantidades de threads menores, e o desempenho do *CJDBC_S* é superior para as quantidades maiores.

O tempo de preparação do *CJDBC_S* é muito pior do que o tempo do *WJDBC*, e vai-se degradando com uma quantidade de threads maior.

O tempo de preparação tem um impacto pequeno no resultado final.

O número de linhas por thread não influencia significativamente os resultados.

5.2.2.4 Remoção

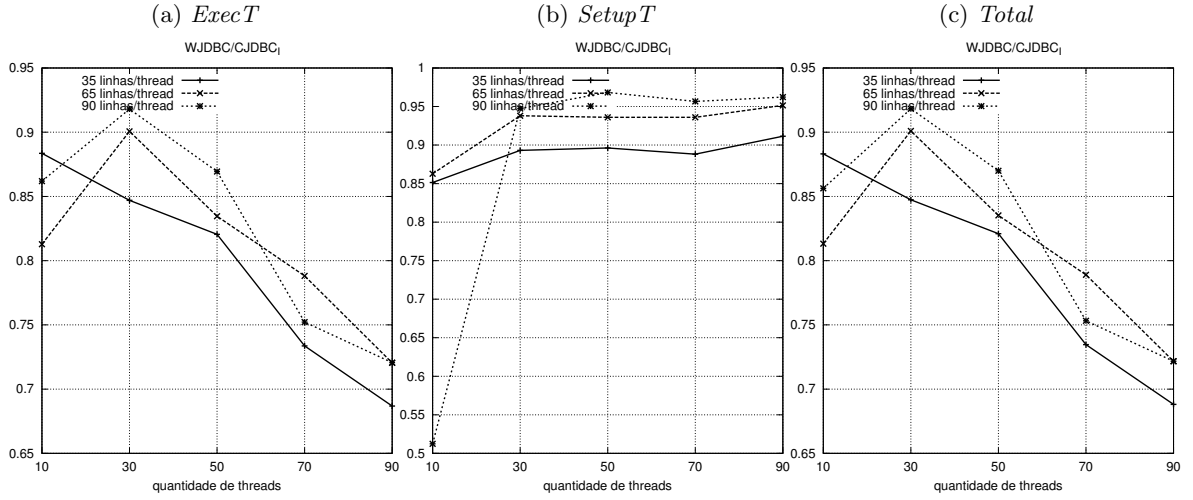
5.2.2.4.1 *CJDBC_I*

A Figura 5.19 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_I*, no contexto Remoção.

O desempenho do *WJDBC* é melhor do que o desempenho do *CJDBC_I*, cujo desempenho se degrada com o aumento da quantidade de threads.

O número de linhas por thread não afecta significativamente os resultados.

Figura 5.19: Comparação entre $WJDBC$ e $CJDBC_I$, no contexto Remoção.

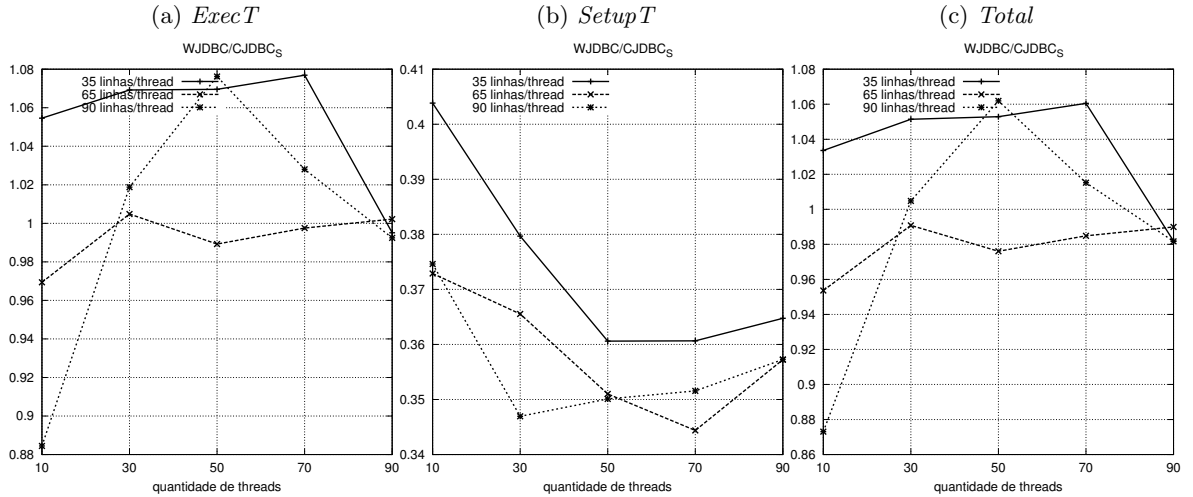


O tempo de preparação é semelhante para ambos, tendo pouco impacto no resultado total.

5.2.2.4.2 $CJDBC_S$

A Figura 5.20 apresenta os resultados da comparação do desempenho de $WJDBC$ e de $CJDBC_S$, no contexto Remoção.

Figura 5.20: Comparação entre $WJDBC$ e $CJDBC_S$, no contexto Remoção.



Nos tempos de execução e total a vantagem vai para o desempenho do $CJDBC_S$, embora seja muito pequena. O aumento da quantidade de threads não influencia significativamente os resultados e o número de linhas por thread maior é melhor para o desempenho do $WJDBC$.

O tempo de preparação do $WJDBC$ é bastante superior ao de $CJDBC_S$, e o aumento de quantidade de threads melhora a vantagem.

O tempo de preparação não tem muito peso no tempo total, pelo que os resultados da comparação dos tempos de execução e total são muito parecidos.

5.2.3 Resumo

Resumindo os resultados podemos dizer que em **geral**:

- O desempenho de $CJDBC_I$ é pior do que o de $WJDBC$, principalmente para quantidades de threads maiores. A excepção regista-se no contexto da Inserção em que $CJDBC_I$ é ligeiramente melhor;
- O desempenho de $CJDBC_S$ é melhor do que o de $WJDBC$, principalmente para quantidades de threads maiores.

5.3 Benchmark com atrasos

Nesta secção são apresentados os resultados do benchmark que introduz a simulação de processamento entre colunas ou linhas. Este benchmark foi apresentado na secção 4.2.

Uma vez que o único tempo medido que é directamente afectado pela introdução de atrasos é o tempo de execução, este será o único cujos valores serão apresentados. Para ser mais cómodo, os gráficos dos resultados com atraso e sem atraso são agrupados por contexto.

Os valores dos atrasos entre colunas e linhas foram escolhidos de forma a poderem produzir resultados que sejam comparáveis. A tabela de testes (fig. 4.1) possui oito colunas, assim no atraso entre colunas é introduzido um atraso total de $8 \times \text{atraso}_C$, em que atraso_C é valor do atraso introduzido entre cada coluna. O atraso entre linhas (atraso_L), para ser comparável, terá então o valor de $8 \times \text{atraso}_C$. Por exemplo, atraso de 0.1ms entre colunas *versus* atraso de 0.8ms entre linhas.

Os detalhes do funcionamento deste benchmark podem ser encontrados na secção 4.2.

5.3.1 Atraso entre colunas

5.3.1.1 Comparação com MSJDBC

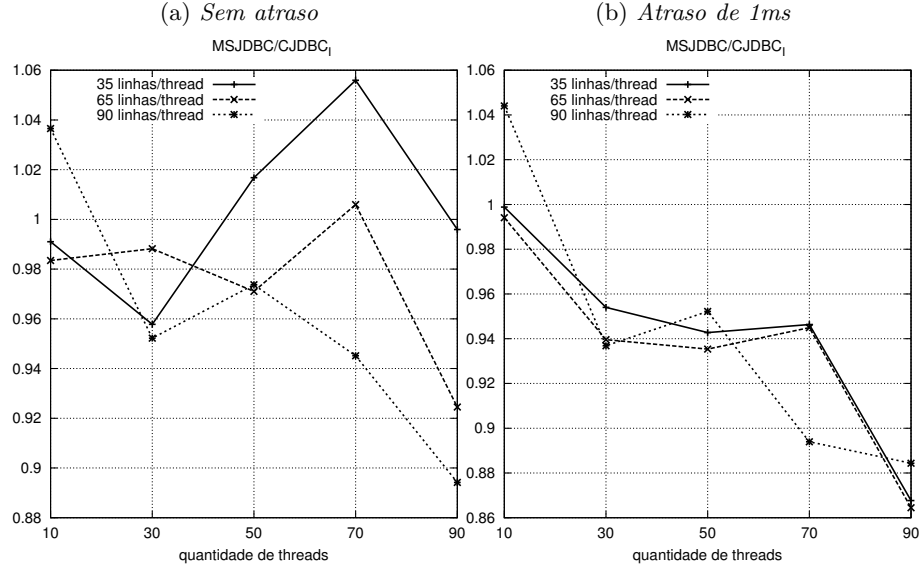
Esta secção apresenta os resultados do benchmark com atrasos (secção 4.2). Os resultados reflectem o efeito do atraso entre colunas, e são apresentados sob a forma de uma comparação entre o desempenho registado para o $MSJDBC$ e as restantes soluções. A comparação é realizada através de um rácio dos tempos registados.

5.3.1.1.1 Actualização

5.3.1.1.1.1 $CJDBC_I$

A Figura 5.21 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_I$, no contexto Actualização.

Figura 5.21: Efeito do atraso na comparação entre $MSJDBC$ e $CJDBC_I$, no contexto Actualização.



A introdução afectou ligeiramente o desempenho de $CJDBC_I$, diminuindo o desempenho em relação ao $MSJDBC$.

Os números de linhas por thread mais baixos denotam um maior efeito do atraso no desempenho de $CJDBC_I$;

Os números de linhas por thread mais altos refletem menos impacto no desempenho.

No geral a introdução de atraso entre colunas na actualização não tem muita influência nos resultados.

5.3.1.1.1.2 $CJDBC_S$

A Figura 5.22 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_S$, no contexto Actualização.

A introdução de atraso prejudica um pouco o desempenho do $CJDBC_S$.

Onde se nota um maior impacto da introdução de atraso é no número de linhas mais baixo, e para quantidades de threads maiores.

No geral a introdução de atraso entre colunas na actualização não tem muita influência nos resultados, e é ainda menos influenciante do que no caso do $CJDBC_I$.

5.3.1.1.1.3 $WJDBC$

A Figura 5.23 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $WJDBC$, no contexto Actualização.

A introdução de atraso tem uma enorme influência nos resultados, prejudicando gravemente o desempenho do $WJDBC$.

Figura 5.22: Efeito do atraso na comparação entre *MSJDBC* e *CJDBC_S*, no contexto **Atualização**.

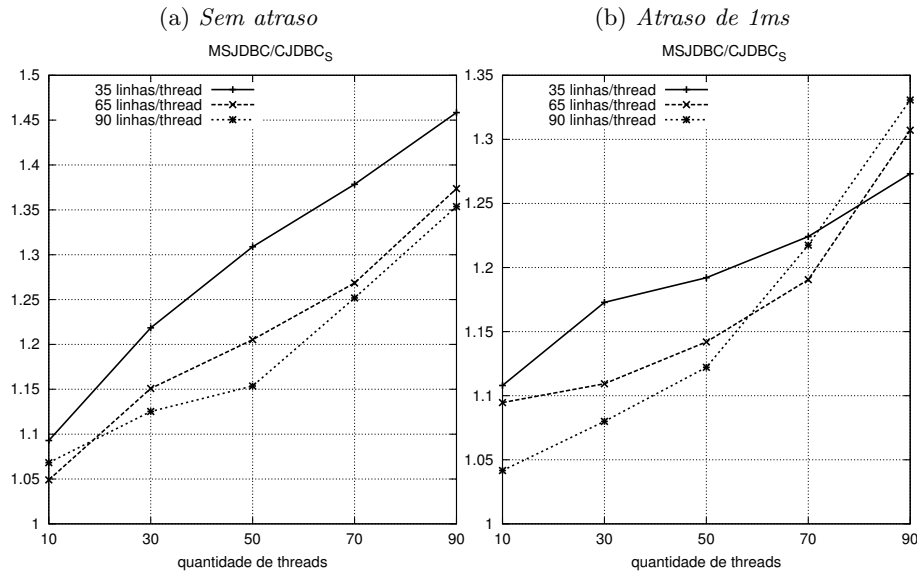
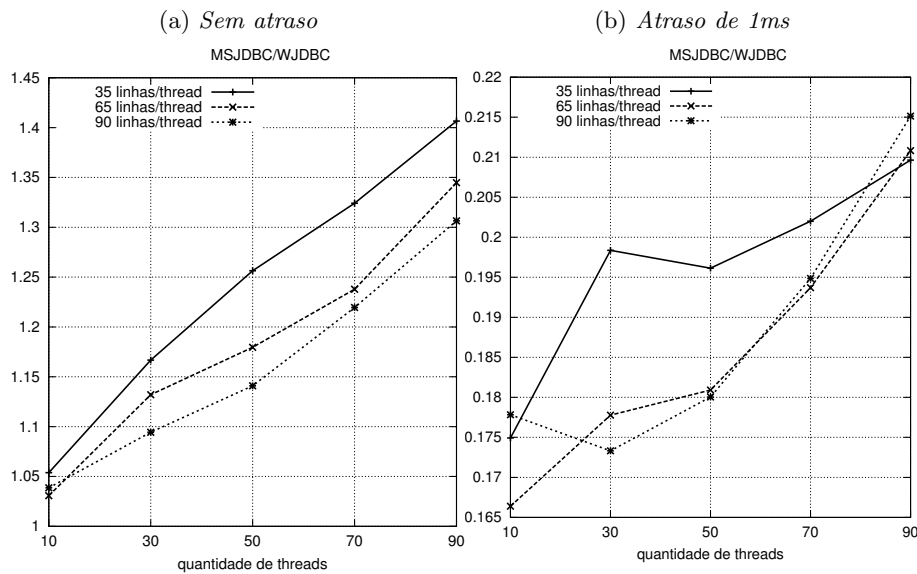


Figura 5.23: Efeito do atraso na comparação entre *MSJDBC* e *WJDBC*, no contexto **Atualização**.



Na ausência de atraso o desempenho do *WJDBC* é superior ao de *MSJDBC*, mas com a introdução de atraso o desempenho do *WJDBC* é bastante pior (cerca 8 vezes).

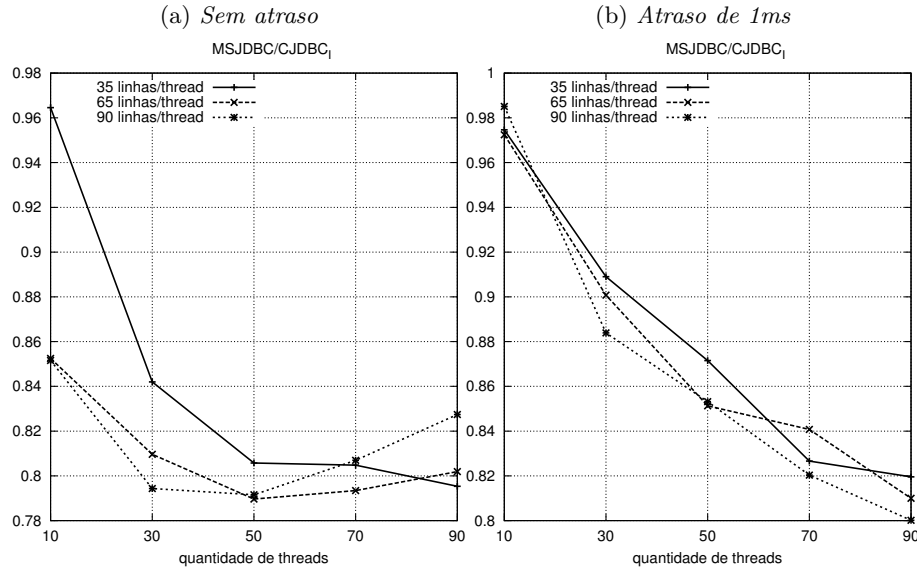
Porém comportamento em resposta à variação dos parâmetros é semelhante: os números de linhas por thread mais baixos e quantidades de threads maiores favorecem o desempenho do *WJDBC*.

5.3.1.1.2 Leitura

5.3.1.1.2.1 $CJDBC_I$

A Figura 5.24 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_I$, no contexto Leitura.

Figura 5.24: Efeito do atraso na comparação entre $MSJDBC$ e $CJDBC_I$, no contexto **Leitura**.



A introdução de atraso beneficia ligeiramente o desempenho do $CJDBC_I$ em relação ao $MSJDBC$.

São as quantidades de linhas por thread maiores que mais ganham com a introdução de atraso.

Porém o desempenho do $CJDBC_I$ degrada-se com o aumento da quantidade de threads.

5.3.1.1.2.2 $CJDBC_S$

A Figura 5.25 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_S$, no contexto Leitura.

O desempenho do $CJDBC_S$ é gravemente prejudicado pela introdução de atraso.

Ainda existe superioridade do desempenho do $CJDBC_S$ em relação ao $MSJDBC$, mas essa superioridade é cerca de 4.5 vezes mais pequena em relação à ausência de atraso.

O $CJDBC_S$ continua a beneficiar do aumento da quantidade de threads.

O número de linhas por thread varia em quase nada os resultados, pois para todos os valores os resultados são aproximados.

5.3.1.1.2.3 $WJDBC$

A Figura 5.26 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $WJDBC$, no contexto Leitura.

Figura 5.25: Efeito do atraso na comparação entre *MSJDBC* e *CJDBC_S*, no contexto **Leitura**.

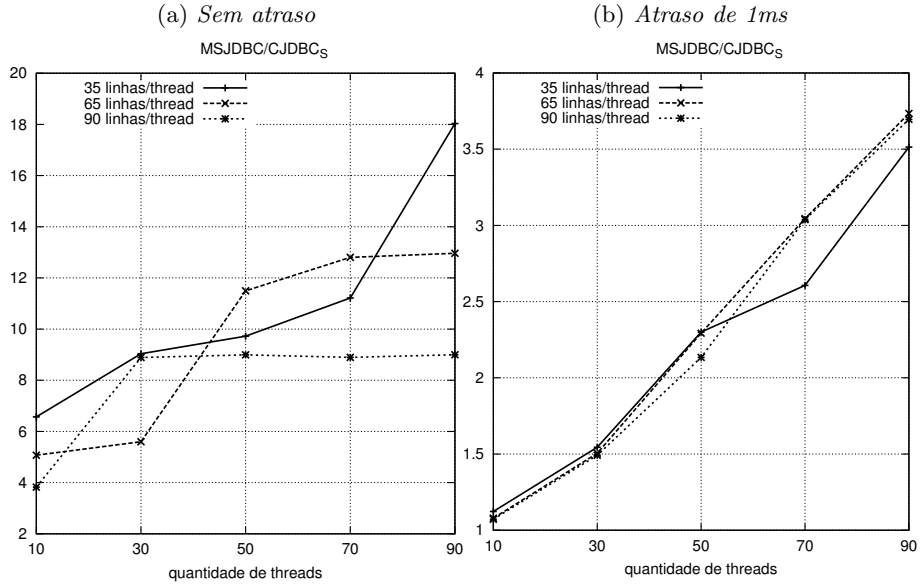
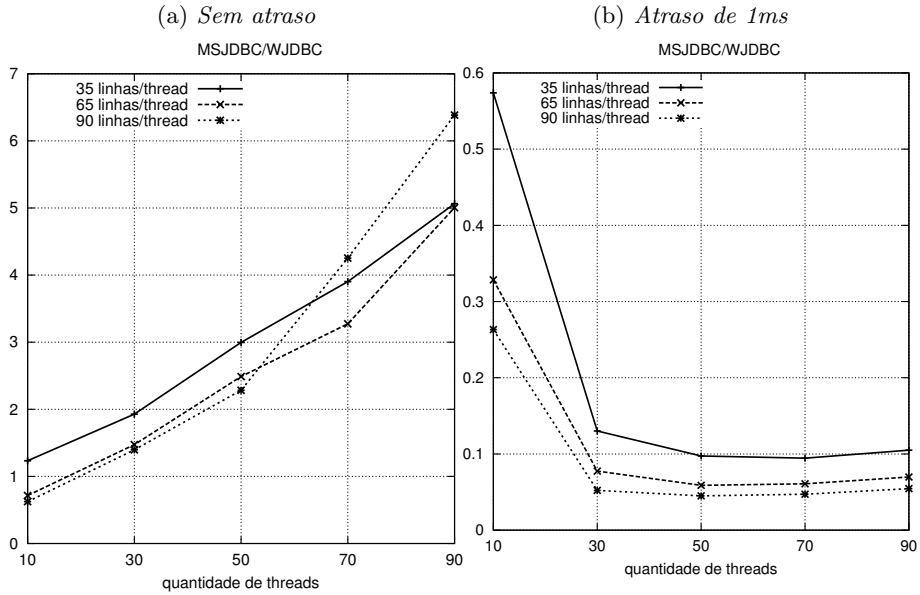


Figura 5.26: Efeito do atraso na comparação entre *MSJDBC* e *WJDBC*, no contexto **Leitura**.



O desempenho de *WJDBC* em relação ao de *MSJDBC* diminuiu drasticamente com a introdução de atraso.

Para as quantidades de threads menores a tendência é o desempenho de *WJDBC* diminuir em relação ao de *MSJDBC*. Para as quantidades maiores a relação entre as duas soluções mantém-se quase constante.

5.3.1.2 Comparação com WJDBC

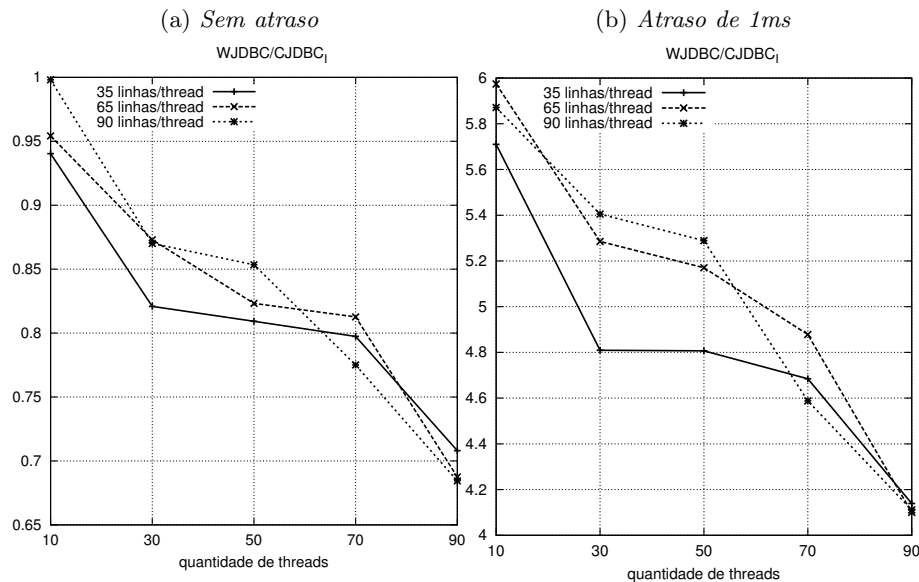
Esta secção apresenta os resultados do benchmark com atrasos (secção 4.2). Os resultados reflectem o efeito do atraso entre colunas, e são apresentados sob a forma de uma comparação entre o desempenho registado para o *WJDBC* e as restantes soluções. A comparação é realizada através de um rácio dos tempos registados.

5.3.1.2.1 Actualização

5.3.1.2.1.1 *CJDBC_I*

A Figura 5.27 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_I*, no contexto Actualização.

Figura 5.27: Efeito do atraso na comparação entre *WJDBC* e *CJDBC_I*, no contexto Actualização.



A introdução de atraso dá uma enorme vantagem ao *CJDBC_I*.

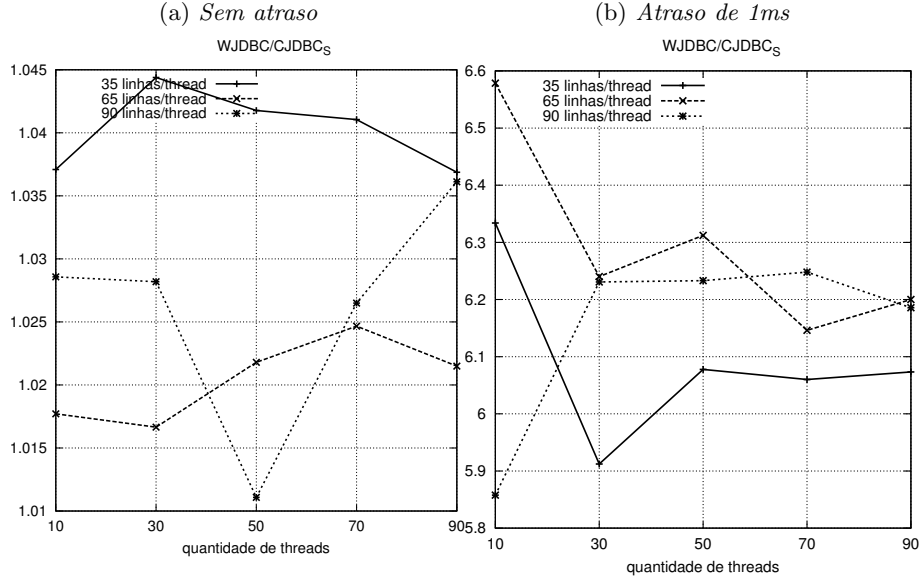
Sem o atraso o desempenho do *CJDBC_I* é no melhor caso aproximadamente igual ao de *WJDBC* e no pior caso é cerca de 0.3 vezes pior, porém com a introdução de atraso no melhor caso é 6 vezes melhor e no pior caso 4 vezes melhor.

O *CJDBC_I* continua a perder desempenho face ao *WJDBC* para quantidades de threads maiores, e os números de linhas por thread mais altos também continuam a beneficiar o *CJDBC_I*.

5.3.1.2.1.2 *CJDBC_S*

A Figura 5.28 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_S*, no contexto Actualização.

Figura 5.28: Efeito do atraso na comparação entre $WJDBC$ e $CJDBC_S$, no contexto **Atualização**.



O desempenho do $CJDBC_S$ é bastante beneficiado pela introdução de atraso, em relação ao $WJDBC$. Sem atraso, o desempenho de ambos é próximo. Com atraso, o desempenho de $CJDBC_S$ é mais de 6 vezes melhor.

A variação dos parâmetros do benchmark continuam a não influenciar significativamente os resultados.

5.3.1.2.2 Leitura

5.3.1.2.2.1 $CJDBC_I$

A Figura 5.29 apresenta os resultados da comparação do desempenho de $WJDBC$ e de $CJDBC_I$, no contexto Leitura.

O impacto da introdução de atraso é enorme, dando uma **grande** vantagem ao desempenho do $CJDBC_I$ sobre o $WJDBC$.

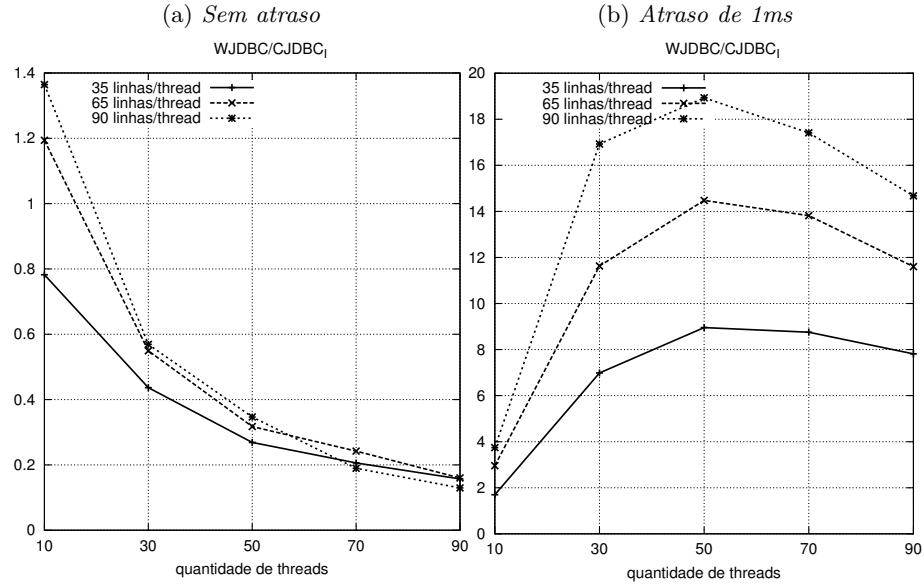
É maior a vantagem que o $CJDBC_I$ tem sobre o $WJDBC$ com atraso, do que a vantagem que existe do $WJDBC$ na ausência de atraso.

Sem atraso o desempenho do $WJDBC$ chega a ser 1.4 vezes melhor, mas com atraso o desempenho do $CJDBC_I$ chega a ser cerca de 19 vezes melhor.

A tendência é de melhoria do desempenho de $CJDBC_I$ com o aumento de quantidade de threads até se verificar uma certa estabilização para quantidades maiores.

Quanto maior é o número de linhas por thread, maior é a vantagem do $CJDBC_I$ sobre $WJDBC$.

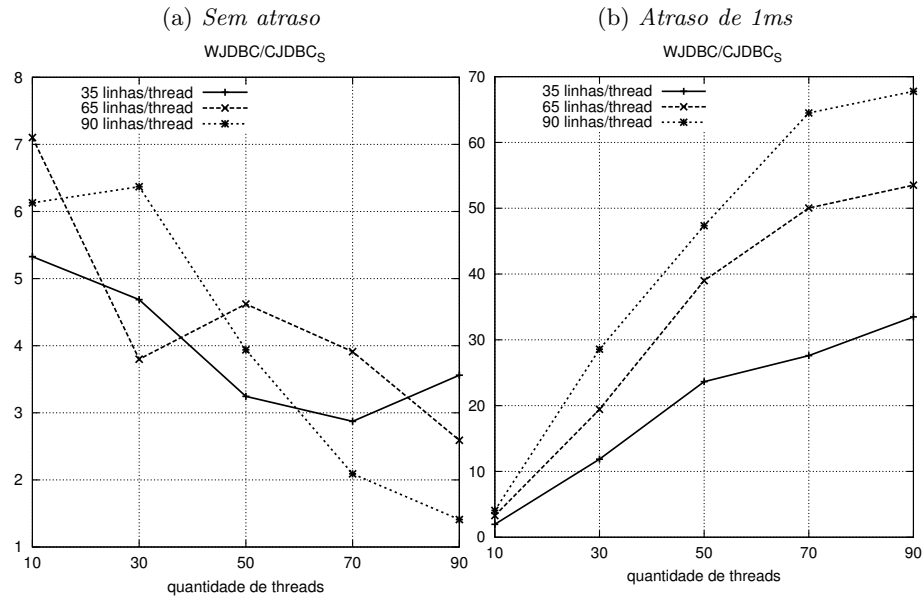
Figura 5.29: Efeito do atraso na comparação entre $WJDBC$ e $CJDBC_I$, no contexto **Leitura**.



5.3.1.2.2.2 $CJDBC_S$

A Figura 5.30 apresenta os resultados da comparação do desempenho de $WJDBC$ e de $CJDBC_S$, no contexto **Leitura**.

Figura 5.30: Efeito do atraso na comparação entre $WJDBC$ e $CJDBC_S$, no contexto **Leitura**.



Verifica-se um resultado semelhante ao de $CJDBC_I$, mas a vantagem de $CJDBC_S$ sobre $WJDBC$ é ainda mais esmagadora.

O desempenho de $CJDBC_S$ já era melhor do que o desempenho de $WJDBC$ na ausência de atraso, por isso a vantagem foi ainda maior com atraso.

A tendência de perda de desempenho do $CJDBC_S$ com o aumento da quantidade de threads foi invertida.

5.3.2 Atraso entre linhas

5.3.2.1 Comparação com MSJDBC

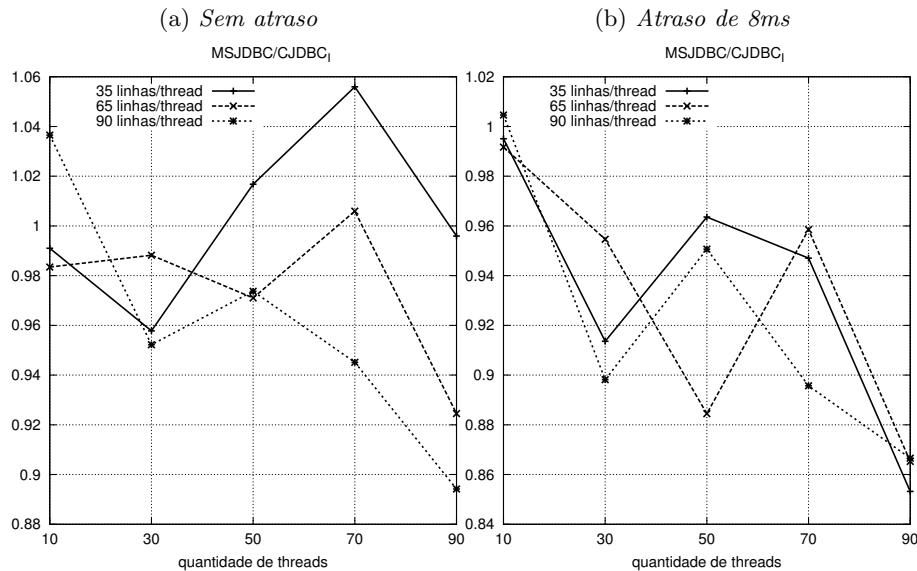
Esta secção apresenta os resultados do benchmark com atrasos (secção 4.2). Os resultados reflectem o efeito do atraso entre linhas, e são apresentados sob a forma de uma comparação entre o desempenho registado para o $MSJDBC$ e as restantes soluções. A comparação é realizada através de um rácio dos tempos registados.

5.3.2.1.1 Actualização

5.3.2.1.1.1 $CJDBC_I$

A Figura 5.31 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_I$, no contexto Actualização.

Figura 5.31: *Efeito do atraso na comparação entre $MSJDBC$ e $CJDBC_I$, no contexto Actualização.*



Os resultados do desempenho do $CJDBC_I$ com atraso são ligeiramente piores do que os resultados do $MSJDBC$.

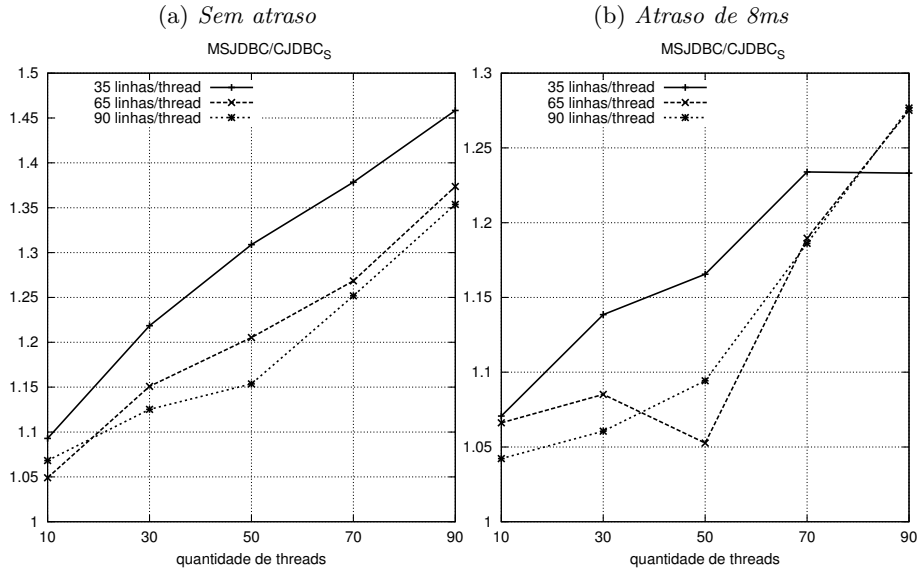
A quantidade de linhas por thread mais baixa é a que revela maior perda de desempenho por parte do $CJDBC_I$.

Tirando a ligeira descida do desempenho do $CJDBC_I$, o efeito do atraso é insignificante.

5.3.2.1.1.2 $CJDBC_S$

A Figura 5.32 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_S$, no contexto Actualização.

Figura 5.32: Efeito do atraso na comparação entre $MSJDBC$ e $CJDBC_S$, no contexto Actualização.



O desempenho do $CJDBC_S$ é prejudicado ligeiramente com introdução de atraso.

É no número de linhas por thread mais baixo que se nota a maior diferença.

No geral o efeito da introdução de atraso é insignificante.

5.3.2.1.1.3 $WJDBC$

A Figura 5.33 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $WJDBC$, no contexto Actualização.

O desempenho do $WJDBC$ sofre uma perda significativa em relação ao desempenho do $MSJDBC$.

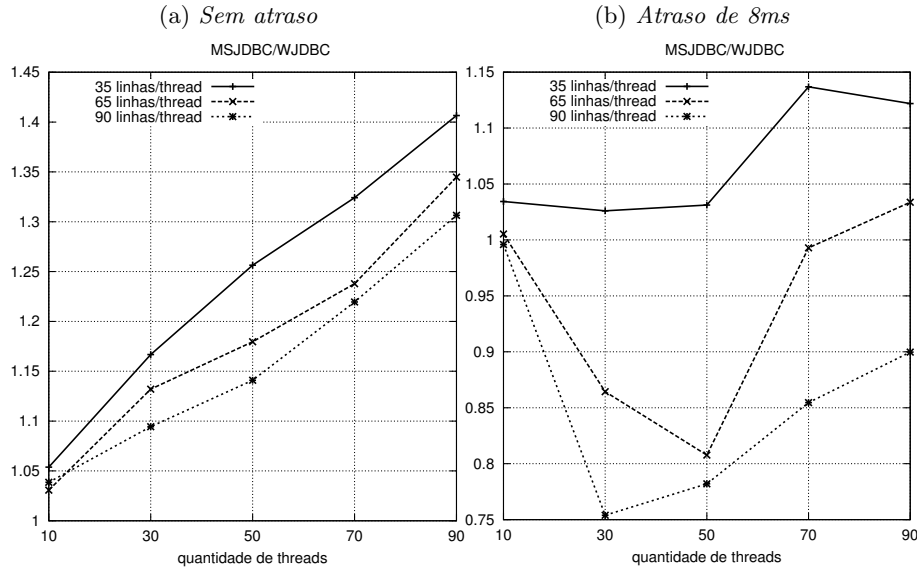
O crescimento da vantagem em relação ao $MSJDBC$ quando o número de threads aumenta já não se verifica.

Os valores do desempenho do $WJDBC$ para os números de linhas por thread maiores perdem mais, ficando mesmo abaixo do desempenho do $MSJDBC$.

5.3.2.1.2 Leitura

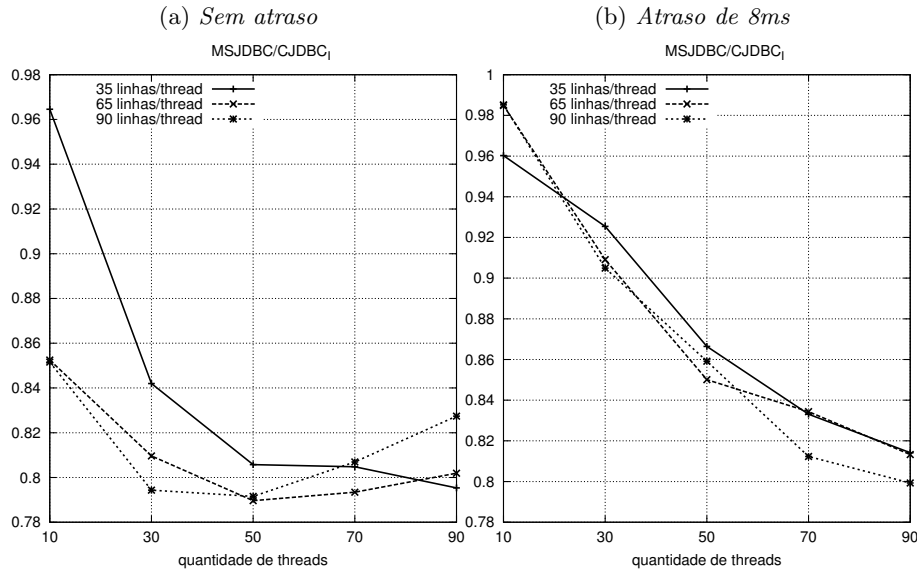
5.3.2.1.2.1 $CJDBC_I$

Figura 5.33: Efeito do atraso na comparação entre *MSJDBC* e *WJDBC*, no contexto **Actualização**.



A Figura 5.34 apresenta os resultados da comparação do desempenho de *MSJDBC* e de *CJDBC_I*, no contexto Leitura.

Figura 5.34: Efeito do atraso na comparação entre *MSJDBC* e *CJDBC_I*, no contexto **Leitura**.



A introdução de atraso favorece o desempenho do *CJDBC_I* em relação ao desempenho do *MSJDBC*.

É nos valores de linhas por thread mais altos que se verifica a vantagem da introdução de atraso de *CJDBC_I* em relação a *MSJDBC*, no entanto o desempenho do *MSJDBC* ainda é superior ao de *CJDBC_I*.

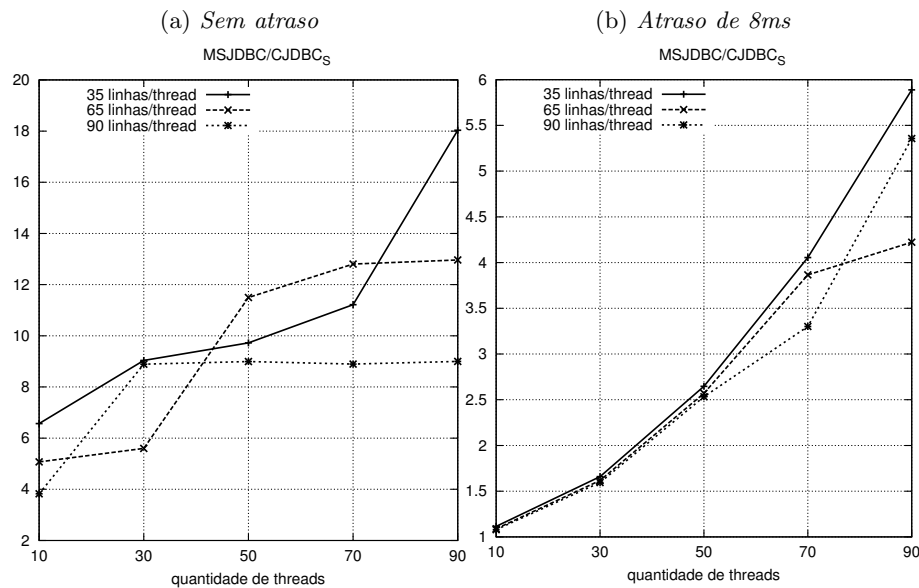
Os diferentes valores de linhas por thread apresentam resultados próximos entre si.

As quantidades maiores de threads penalizam o desempenho do $CJDBC_I$.

5.3.2.1.2.2 $CJDBC_S$

A Figura 5.35 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $CJDBC_S$, no contexto Leitura.

Figura 5.35: Efeito do atraso na comparação entre $MSJDBC$ e $CJDBC_S$, no contexto **Leitura**.



O efeito da introdução do atraso é penalizador para o desempenho do $CJDBC_S$.

Embora o $CJDBC_S$ ainda mantenha um desempenho claramente superior ao desempenho de $MSJDBC$, a vantagem é aproximadamente 3.5 vezes menor.

A diferença entre os resultados para os vários valores de linhas por thread diminuiu.

O aumento da quantidade de threads continua a beneficiar o desempenho do $CJDBC_S$.

5.3.2.1.2.3 $WJDBC$

A Figura 5.36 apresenta os resultados da comparação do desempenho de $MSJDBC$ e de $WJDBC$, no contexto Leitura.

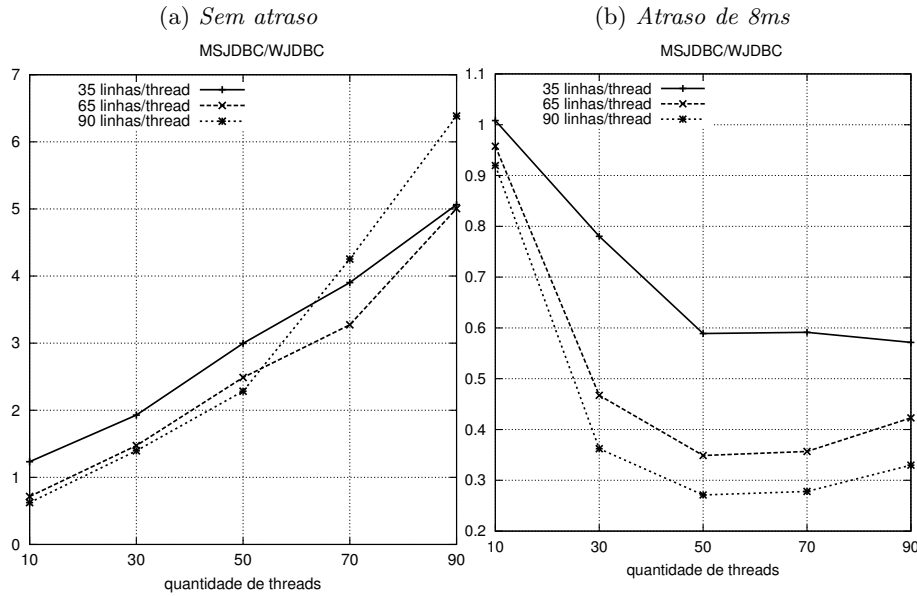
A introdução de atraso é muito penalizador para o desempenho do $WJDBC$.

O desempenho do $WJDBC$ deixou de ser superior ao de $MSJDBC$, e passou a ser bastante inferior.

$WJDBC$ perdeu mais desempenho para os valores de linhas por thread maiores.

O aumento da quantidade de thread começa por penalizar o desempenho do $WJDBC$ até que estabiliza, permanecendo relativamente constante.

Figura 5.36: Efeito do atraso na comparação entre *MSJDBC* e *WJDBC*, no contexto **Leitura**.



5.3.2.2 Comparação com *WJDBC*

Esta secção apresenta os resultados do benchmark com atrasos (secção 4.2). Os resultados reflectem o efeito do atraso entre linhas, e são apresentados sob a forma de uma comparação entre o desempenho registado para o *WJDBC* e as restantes soluções. A comparação é realizada através de um rácio dos tempos registados.

5.3.2.2.1 Actualização

5.3.2.2.1.1 *CJDBC_I*

A Figura 5.37 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_I*, no contexto Actualização.

A introdução de atraso beneficia o desempenho do *CJDBC_I* em relação ao desempenho do *WJDBC*, principalmente para os valores de linhas por thread maiores.

O valor de linhas por thread mais baixo sofreu poucas alterações.

Tirando uma fase inicial em que o aumento da quantidade de threads beneficia o desempenho do *CJDBC_I*, o aumento da quantidade de thread penaliza o desempenho do *CJDBC_I* em relação ao desempenho do *WJDBC*.

5.3.2.2.1.2 *CJDBC_S*

A Figura 5.38 apresenta os resultados da comparação do desempenho de *WJDBC* e de *CJDBC_S*, no contexto Actualização.

Figura 5.37: Efeito do atraso na comparação entre $WJDBC$ e $CJDBC_I$, no contexto **Actualização**.

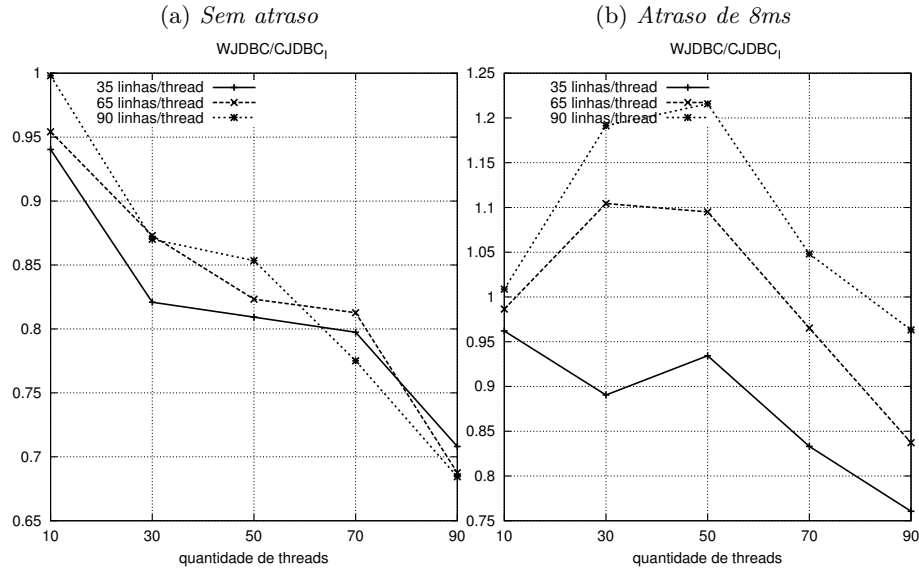
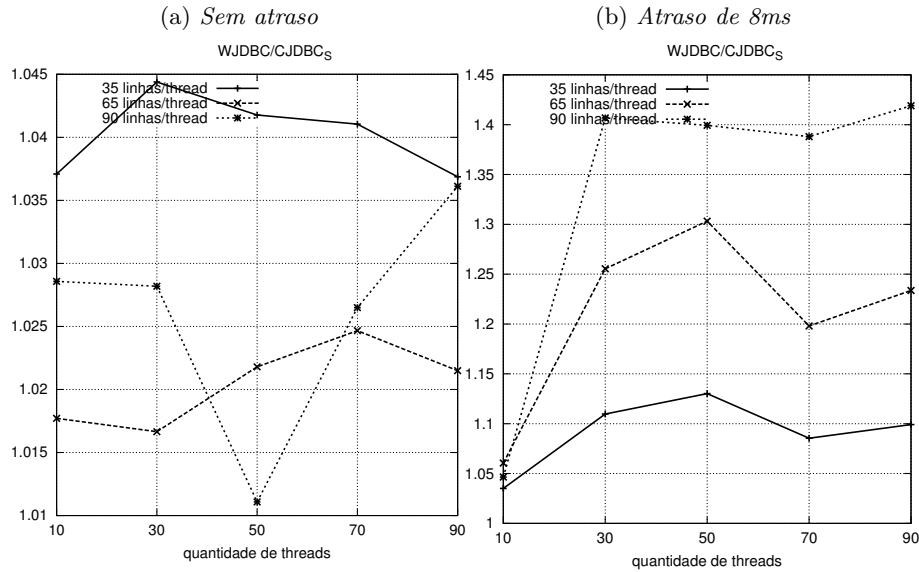


Figura 5.38: Efeito do atraso na comparação entre $WJDBC$ e $CJDBC_S$, no contexto **Actualização**.



A introdução de atraso beneficia com algum significância o desempenho do $CJDBC_S$ face ao desempenho do $WJDBC$, principalmente para os valores de linhas por thread maiores.

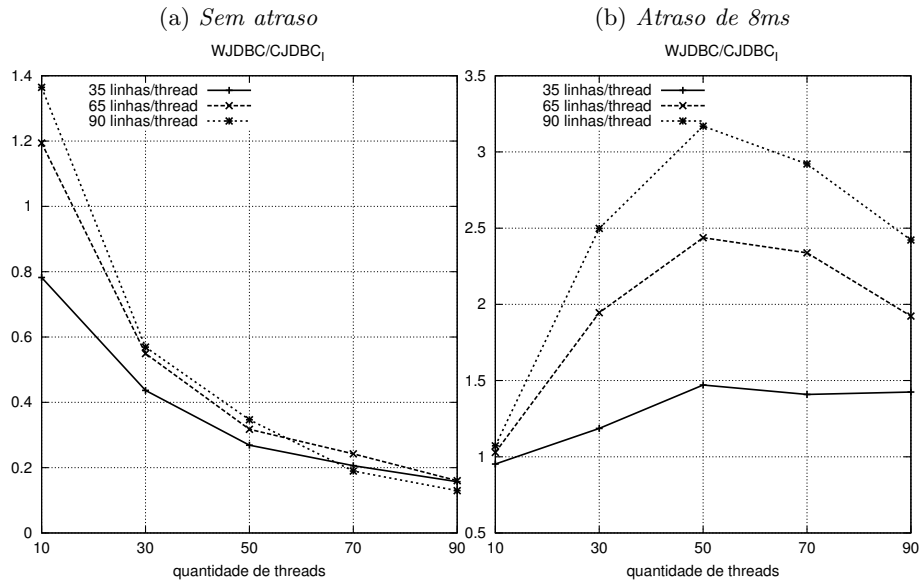
Inicialmente o aumento da quantidade threads dá alguma vantagem ao desempenho do $CJDBC_S$, até estabilizar e manter-se relativamente constante para as quantidades maiores.

5.3.2.2.2 Leitura

5.3.2.2.2.1 $CJDBC_I$

A Figura 5.39 apresenta os resultados da comparação do desempenho de $WJDBC$ e de $CJDBC_I$, no contexto Leitura.

Figura 5.39: Efeito do atraso na comparação entre $WJDBC$ e $CJDBC_I$, no contexto **Leitura**.



Com a introdução de atraso, o desempenho do $CJDBC_I$ passa a ser sempre superior ao de $WJDBC$; a vantagem que $CJDBC_I$ tem agora em relação ao $WJDBC$ é maior do que a vantagem que o $WJDBC$ tinha na ausência de atraso.

Os valores do número de linhas por thread maiores dão maior vantagem ao desempenho do $CJDBC_I$.

O aumento do quantidade de threads dá alguma vantagem ao desempenho do $CJDBC_I$, para quantidades menores.

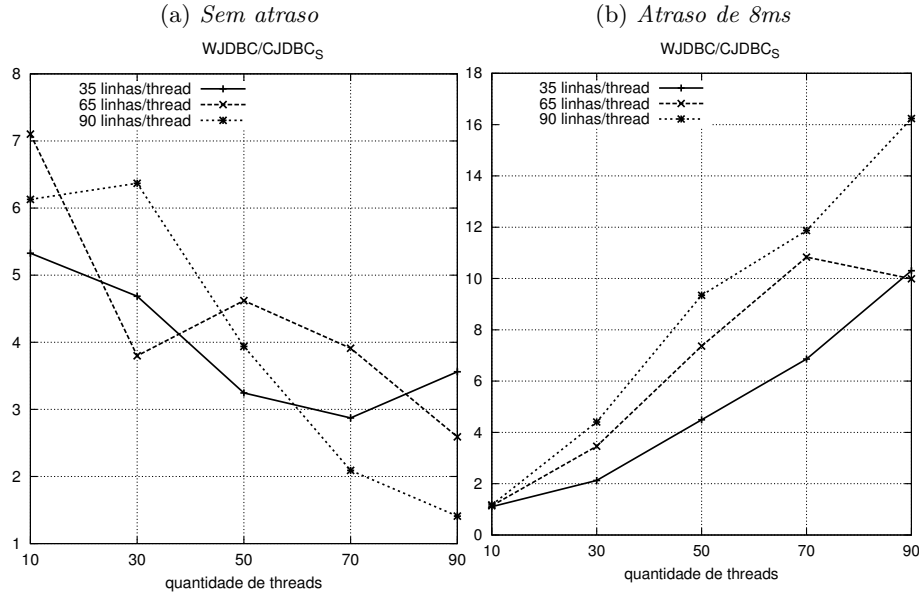
5.3.2.2.2.2 $CJDBC_S$

A Figura 5.40 apresenta os resultados da comparação do desempenho de $WJDBC$ e de $CJDBC_S$, no contexto Leitura.

O desempenho do $CJDBC_S$ em relação ao $WJDBC$ beneficia bastante da introdução de atraso.

Para além de o desempenho do $CJDBC_S$ face ao de $WJDBC$ ser ainda maior do que o que se verificava na ausência de atraso, o aumento da quantidade de threads deixa de ser penalizador para o desempenho do $CJDBC_S$ e passa a dar vantagem.

Figura 5.40: Efeito do atraso na comparação entre $WJDBC$ e $CJDBC_S$, no contexto **Leitura**.



5.3.3 Resumo

O atraso introduzido entre colunas provocou mais efeito nos resultados do que o atraso entre linhas, mas produzem conclusões semelhantes, por isso o seguinte é válido para ambos (salvo diferenças referidas explicitamente):

- A introdução de atraso afecta o $MSJDBC$ e o $CJDBC_I$ da mesma forma, pelo que os resultados com e sem atraso são semelhantes;
- A introdução de atraso não afectou significativamente os resultados da comparação de $CJDBC_S$ com $MSJDBC$ na Actualização, no entanto para a Leitura o desempenho de $CJDBC_S$ sofreu um grande decréscimo;
- O desempenho da solução $WJDBC$ diminuiu muito com a introdução de atraso. Por exemplo, comparando com a ausência de atraso entre colunas, na Leitura foi cerca de 10 vezes pior e na Actualização foi cerca de 6 vezes pior.

5.4 Cache individual vs Cache partilhado

Esta secção apresenta os resultados para o benchmark do Cache individual vs Cache partilhado (secção 4.3). Para cada tamanho do *fetch size* é realizada a comparação entre os resultados registados para o cache individual e o cache partilhado, através de um rácio. Relembro que para poder ser possível a realização deste benchmark teve de ser criada uma versão especial do cache partilhado, denominada de $CJDBC_SM$, que permite definir diferentes tamanhos para a capacidade do cache. Para se ter uma noção da diferença de desempenho entre as versões oficiais de cada tipo de cache, é realizada uma comparação adicional

entre $CJDBC_I$ e $CJDBC_S$, mas que obviamente só foi executada no contexto de **fetch size 100%**.

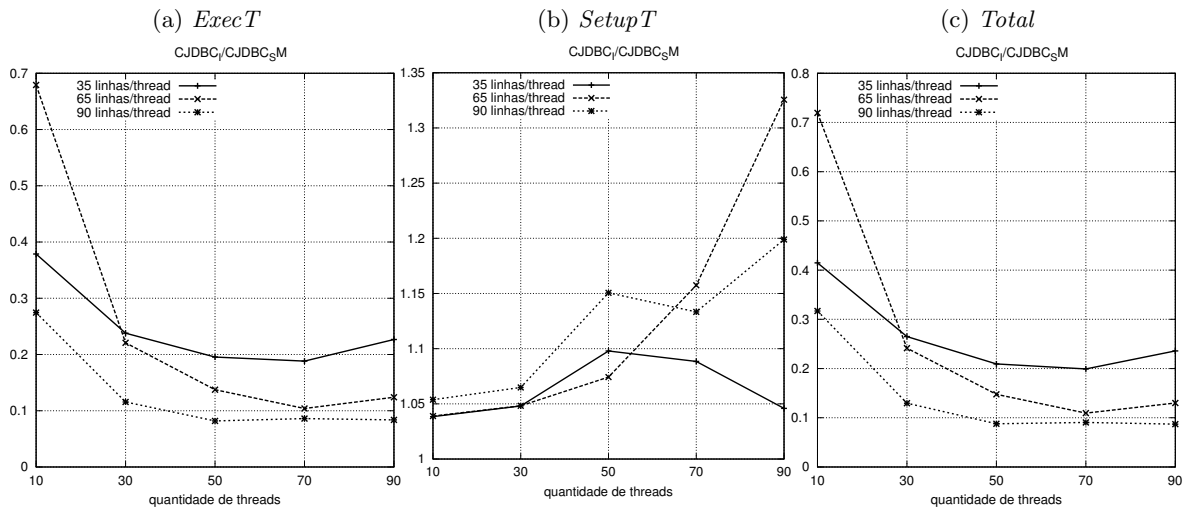
Relembro ainda que se utilizou a forma mais compacta **10**, **20**, etc, para designar os contextos **fetch size 10%**, **fetch size 20%**, etc.

5.4.1 Fetch size 10%

5.4.1.1 $CJDBC_{SM}$

A Figura 5.41 apresenta os resultados da comparação do desempenho de $CJDBC_I$ e de $CJDBC_{SM}$, no contexto Fetch size 10%.

Figura 5.41: Comparação entre $CJDBC_I$ e $CJDBC_{SM}$, no contexto 10 .



O desempenho do $CJDBC_{SM}$ é bastante inferior ao desempenho do $CJDBC_I$.

O valor do tempo de preparação tem pouco impacto, pelo que os valores do desempenho comparativo para os tempos de execução e total são essencialmente os mesmos.

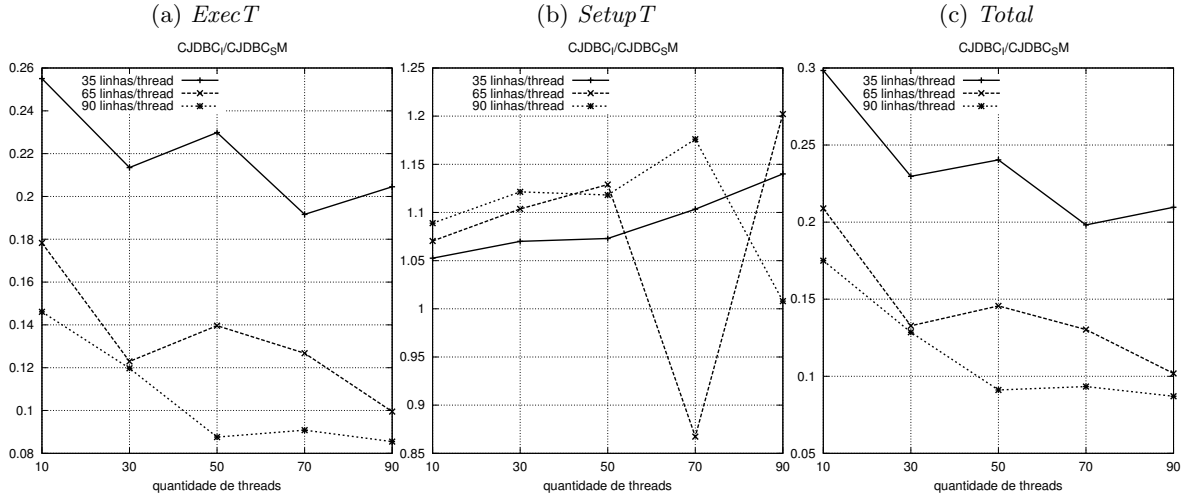
5.4.2 Fetch size 20%

5.4.2.1 $CJDBC_{SM}$

A Figura 5.42 apresenta os resultados da comparação do desempenho de $CJDBC_I$ e de $CJDBC_{SM}$, no contexto Fetch size 20%.

O desempenho do $CJDBC_{SM}$ é sempre bastante inferior ao de $CJDBC_I$. O aumento do número de linhas por thread penaliza significativamente o desempenho do $CJDBC_{SM}$ face

Figura 5.42: Comparação entre $CJDBC_I$ e $CJDBC_{SM}$, no contexto 20.



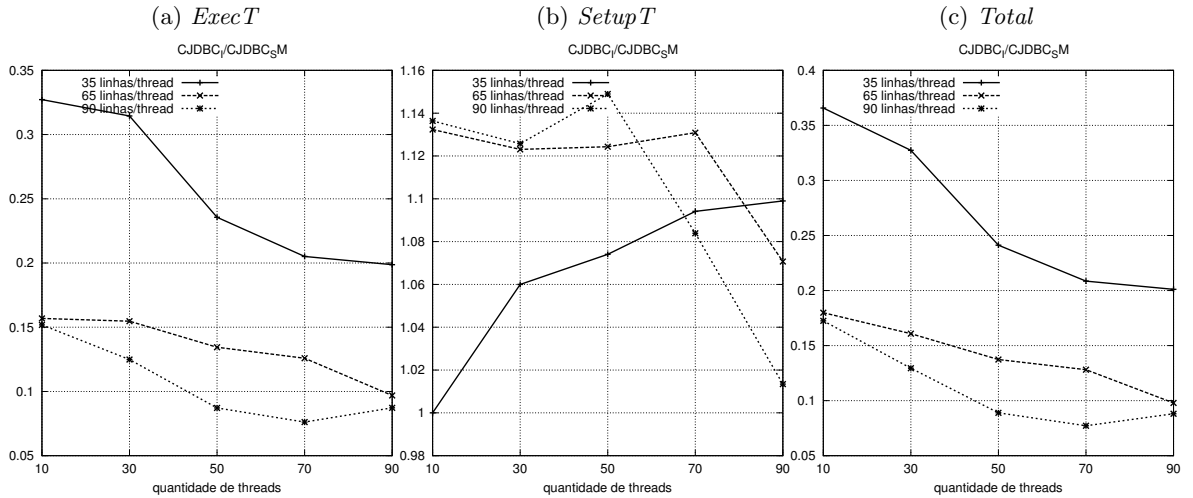
ao $CJDBC_I$. Embora o aumento da quantidade de threads também penalize o desempenho do $CJDBC_{SM}$, o peso é menor. O tempo de preparação tem pouca influência no resultado total.

5.4.3 Fetch size 50%

5.4.3.1 $CJDBC_{SM}$

A Figura 5.43 apresenta os resultados da comparação do desempenho de $CJDBC_I$ e de $CJDBC_{SM}$, no contexto Fetch size 50%.

Figura 5.43: Comparação entre $CJDBC_I$ e $CJDBC_{SM}$, no contexto 50.



A comparação de desempenho dos dois mantém-se semelhante aos contextos **fetch size**

10% e 20%, pelo que o que foi dito para esses contextos aplica-se a este.

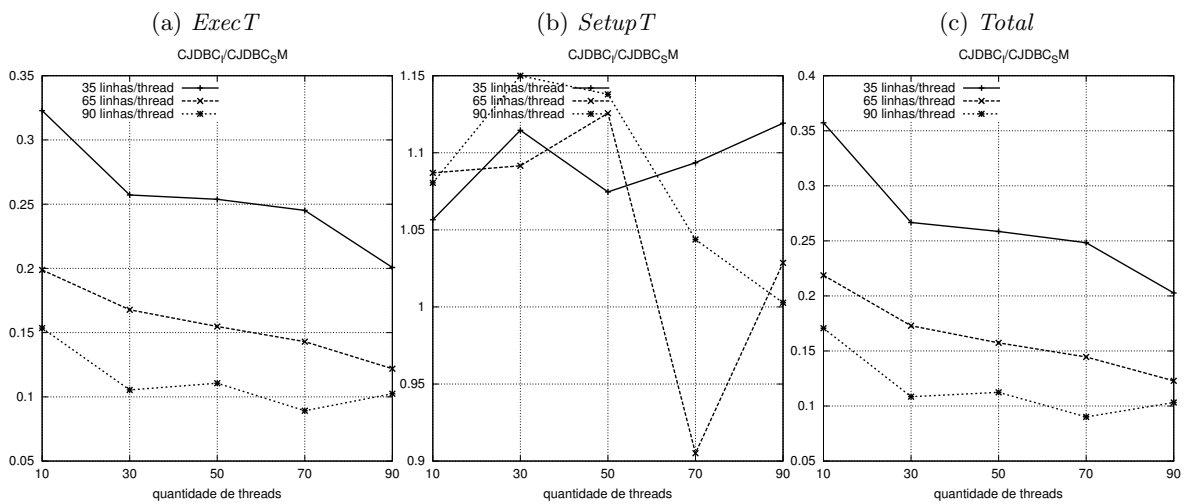
Porém nota-se uma muito ligeira melhoria em relação ao contexto **fetch size 20%**.

5.4.4 Fetch size 75%

5.4.4.1 $CJDBC_S M$

A Figura 5.44 apresenta os resultados da comparação do desempenho de $CJDBC_I$ e de $CJDBC_S M$, no contexto Fetch size 75%.

Figura 5.44: Comparação entre $CJDBC_I$ e $CJDBC_S M$, no contexto 75.



Também aqui os resultados são semelhantes aos contextos anteriores.

Nota-se, porém, uma ligeira melhoria do desempenho do $CJDBC_S M$ para os valores de número de linhas por thread maiores.

5.4.5 Fetch size 100%

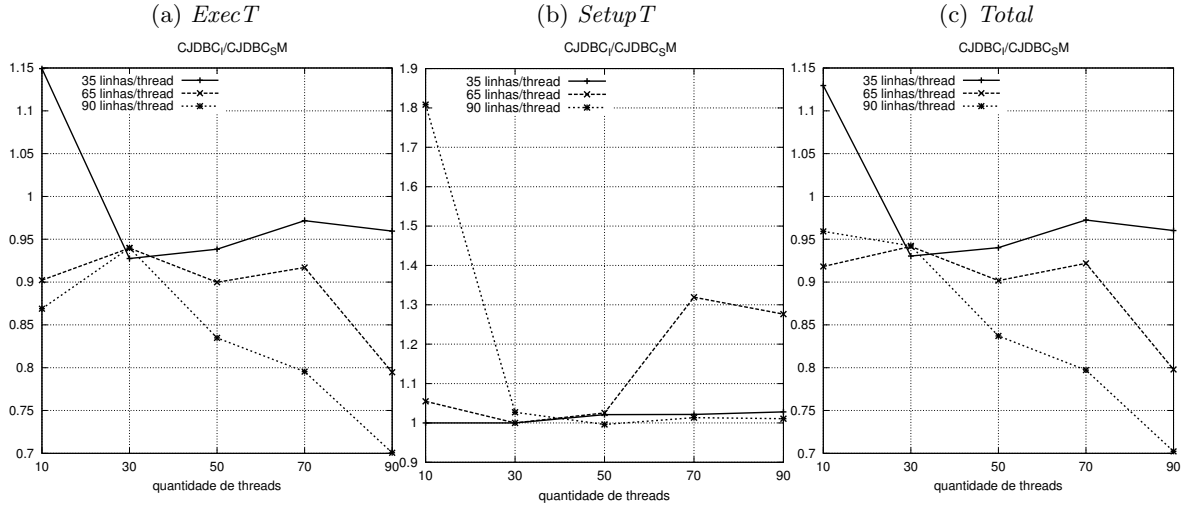
5.4.5.1 $CJDBC_S M$

A Figura 5.45 apresenta os resultados da comparação do desempenho de $CJDBC_I$ e de $CJDBC_S M$, no contexto Fetch size 100%.

O seguinte aplica-se a todos aos tempos de execução e total; o $CJDBC_S M$ para o número de linhas por thread mais baixo, praticamente iguala o desempenho de $CJDBC_I$, começando a perder terreno à medida que os valores vão aumentando.

Em relação ao tempo de preparação, para os valores mais altos de linhas existe alguma vantagem para o $CJDBC_I$, e para o restante valor o desempenho é aproximadamente igual para ambos.

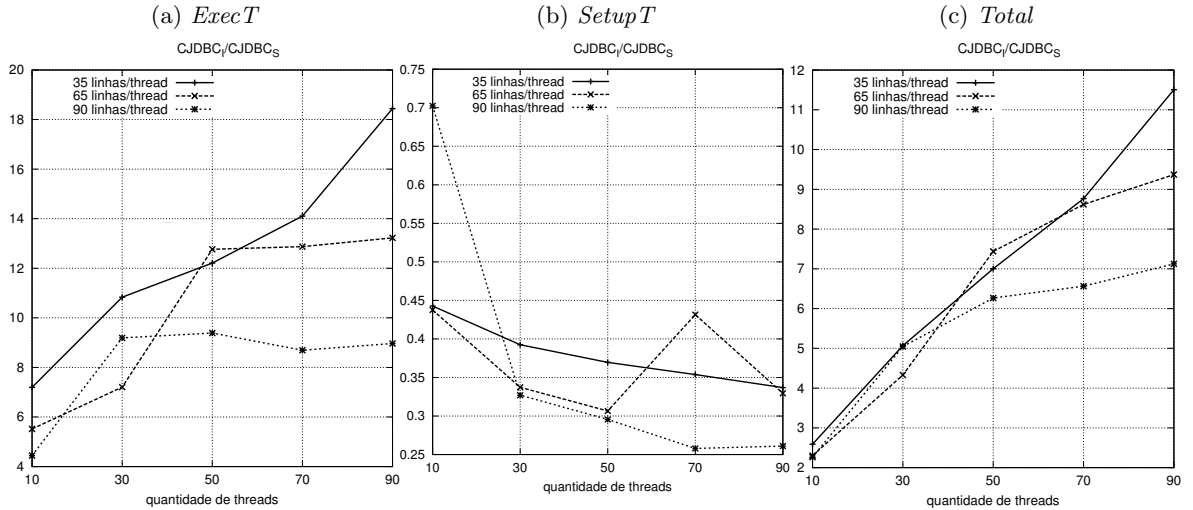
Figura 5.45: Comparação entre $CJDBC_I$ e $CJDBC_{SM}$, no contexto 100.



5.4.5.2 $CJDBC_S$

A Figura 5.46 apresenta os resultados da comparação do desempenho de $CJDBC_I$ e de $CJDBC_S$, no contexto Fetch size 100%.

Figura 5.46: Comparação entre $CJDBC_I$ e $CJDBC_S$, no contexto 100.



O desempenho geral de $CJDBC_S$ é muito superior ao de $CJDBC_I$.

Verifica-se que um número de linhas por thread mais baixo, faz com que o crescimento do desempenho relativo ao tempo de execução de $CJDBC_S$ face ao de $CJDBC_I$ seja mais acentuado, enquanto que para um número de linhas maior o crescimento é quase horizontal.

O tempo de preparação do $CJDBC_I$ é bem melhor do que o de $CJDBC_S$.

Apesar disso, no tempo total o $CJDBC_S$ tem muito melhor desempenho.

5.4.6 Resumo

Resumindo os resultados apresentados:

- O $CJDBC_I$ apresenta muito melhor desempenho do que $CJDBC_S M$ para todos os contextos à exceção de **100**, em que se encontram mais próximos, mas mantendo-se superior;
- O $CJDBC_S$ é no mínimo 2 vezes mais rápido do que $CJDBC_I$ para o contexto **100**, e chega a ser 11.5 vezes mais rápido.

Capítulo 6

Discussão

6.1 Análise de resultados

Nas subsecções seguintes são discutidos e analisados os resultados obtidos e apresentados no capítulo anterior.

6.1.1 Comparação com JDBC

Esta subsecção discute e analisa os resultados apresentados na secção 5.2.

Como já foi mencionado no capítulo dos resultados, todas as soluções apresentadas demonstraram um tempo de preparação muito melhor do que a solução *MSJDBC*. Este era o resultado esperado, na medida em que nesta solução tem que se criar uma statement e um result set para cada thread, enquanto que as restantes soluções possibilitam a partilha do mesmo objecto result set. Isto significa que num ambiente multithreaded com n threads em execução e em que o tempo de criação de um result set é dado por c_{RS} , o tempo total do tempo de preparação para *MSJDBC* será de $n \times c_{RS}$, enquanto que em *CJDBC* e em *WJDBC* será de apenas c_{RS} . Os resultados práticos comprovam esta teoria na medida em que o rácio entre os tempos de preparação do *MSJDBC* e as outras soluções é numericamente aproximado à quantidade de threads utilizado para efectuar a medição (veja-se por exemplo a Figura 5.12). Esta relação não se verifica para a solução *CJDBC_S* porque esta é a única que na preparação carrega os dados do result set para o cache. As outras soluções apenas criam o result set (declaração e abertura do cursor do servidor). A vantagem do melhor tempo de preparação revelou-se importante em diversas situações, porque onde por vezes existia uma ligeira vantagem do *MSJDBC*, esta foi anulada com a ajuda do tempo de preparação. Por exemplo na comparação entre *MSJDBC* e *CJDBC_I* para a Actualização (fig. 5.1), que para quantidades de threads e valores de número de linhas maiores o *MSJDBC* era cerca de 10% mais rápido, no tempo total a vantagem vai para o *CJDBC_I* sendo cerca de 30% mais rápido do que *MSJDBC*.

A solução **CJDBC_I** revelou-se mais eficiente do que o *MSJDBC* em todos os contextos, com destaque para os contextos de modificação (Actualização, Inserção e Remoção),

em que a tendência é haver mais vantagem para o *CJDBC_I* quantos mais threads estiverem em execução. Já no contexto da Leitura, embora exista uma clara vantagem para o *CJDBC_I*, a tendência não se verifica pois existe um ligeiro declínio em favor do desempenho do *MSJDBC*. Esta situação justifica-se pelo facto de o cursor com cache individual, do ponto de vista do cliente, ter um peso semelhante ao *MSJDBC* pois cada thread possui em cache a totalidade do dataset. Depois como se pode ver no tempo de execução da Leitura (fig. 5.4), a implementação da Microsoft é simplesmente mais eficiente do que a realizada neste trabalho. No entanto é importante ressaltar que mesmo com a tendência para diminuir o desempenho do *CJDBC_I* face ao *MSJDBC* com o aumento da quantidade de threads, é na Leitura que o *CJDBC_I* demonstra maior superioridade sendo cerca de 35% a 70% mais rápido do que *MSJDBC*. No contexto da inserção verificou-se muito equilíbrio, principalmente nos tempos de execução em que os resultados de *MSJDBC* e *CJDBC_I* são muito próximos. No tempo total consegue-se notar uma pequena superioridade de *CJDBC_I* devido ao bom desempenho do tempo de preparação. Uma outra nota vai para o facto de os resultados para os contextos da Actualização e da Remoção serem semelhantes. É compreensível que tal se suceda pois tirando o diferente valor para *optype* do RPC *sp_cursor* (ver C.1), a implementação dos métodos de actualização e remoção semelhantes. Ainda assim no contexto da remoção não há actualização de valores das colunas pelo que o desempenho do *CJDBC_I* neste contexto é melhor do que na actualização.

A solução **CJDBC_S** também apresenta um desempenho claramente superior ao desempenho de *MSJDBC*, melhor até porque apesar de não apresentar um tempo de preparação tão bom, ao nível da execução é mais eficiente. O tempo de preparação não é tão bom, porque ao contrário do *MSJDBC* e do *CJDBC_I*, o *CJDBC_S* constrói o cache no momento da criação dos cursores, refletindo-se o peso dessa operação no tempo de preparação em vez do tempo de execução. É também por esta razão que apresenta melhores resultados para a execução, pois uma vez que já tem os dados em cache não necessita de requisitar as linhas do dataset, operação essa que se revela bastante penosa para o desempenho. Este aspecto é claramente visível para o contexto da Leitura em que o *CJDBC_S* consegue ser no mínimo 4 vezes mais rápido do que *MSJDBC* e no máximo quase 18 vezes, e para além disso o desempenho comparativo melhora à medida que a quantidade de threads aumenta, ao contrário do *CJDBC_I* cujo desempenho comparativo diminui nessa situação (ver Figura 6.1).

Queria aqui referir um aspecto em relação ao contexto de Actualização. Como se pode ver na Figura 6.2 o desempenho comparativo do *CJDBC_S* em relação ao *MSJDBC* é melhor do que o do *CJDBC_I*. Mas porquê? Até se poderia pensar que aconteceria precisamente o contrário, pois o cursor de cache partilhado tem de obter acesso exclusivo ao cache para o actualizar, enquanto que o cache individual pode actualizar o seu cache sem essa necessidade. A razão surge do facto que o cache individual tem a necessidade de mover o cursor do servidor antes de efectuar a actualização, já o mesmo não acontece com o cursor com cache partilhado.

A actualização de um result set no SQL Server a partir de um cursor só é possível ao fim de se carregar (fetch) algumas linhas no *buffer* do cursor¹, e é mesmo lançado um erro caso o

Figura 6.1: Comparação entre *CJDBC* e *MSJDBC*, no contexto **Leitura**

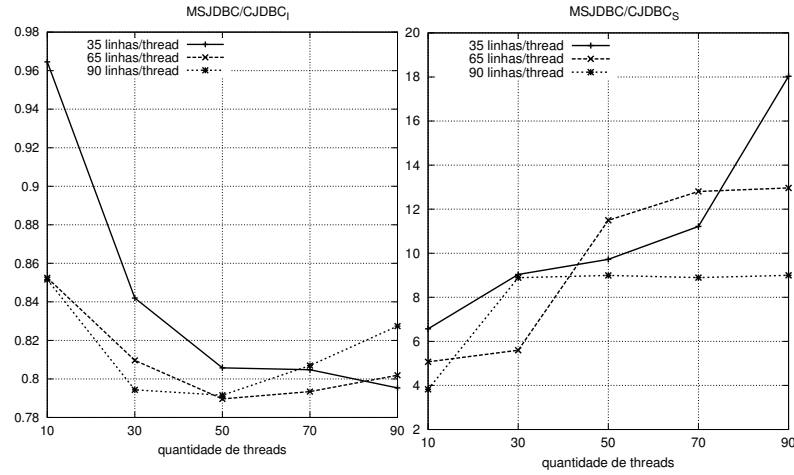
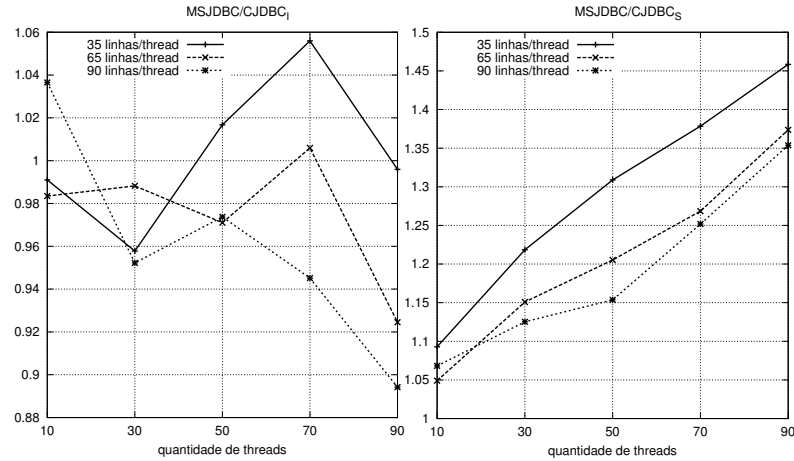


Figura 6.2: Comparação entre *CJDBC* e *MSJDBC*, no contexto **Actualização**



buffer esteja vazio. Na operação de fetch o servidor envia as linhas do buffer ao cliente, que depois cria o seu próprio cache. O índice para actualizar uma linha do result set é relativa a esse buffer e não ao result set. Isso quer dizer que o cliente pode pedir para actualizar, por exemplo, a linha 3 e estar a actualizar a linha 60 do result set.

O cursor com cache individual não pode assumir que tem todas as linhas do result set, logo assume que a sua noção de cache é diferente da noção do servidor, e por essa razão move o cursor do servidor antes de actualizar, através de um UPDATE ABSOLUTE em vez de um UPDATE normal. A diferença é que no UPDATE, o cursor utiliza o seu buffer para actualizar os dados da tabela, enquanto que no UPDATE ABSOLUTE a modificação é realizada nas tabelas. Já no cursor com cache partilhado os índices utilizados para identificar as linhas são os mesmos nos dois lados e inalteráveis, pelo que se pode actualizar directamente usando o cursor servidor.

Tal como se havia verificado com *CJDBC_I*, o desempenho de *CJDBC_S* e *MSJDBC*

¹Na documentação do protocolo é utilizada a designação de *buffer* em vez de cache.

para a Inserção, são próximos.

A solução **WJDBC** também revelou melhor desempenho do que *MSJDBC* em todos os contextos, voltando a revelar uma relação directa entre o número de threads e o ganho de desempenho para o tempo de preparação, tal como se verificou com *CJDBC_I*. Ao nível da preparação as soluções *WJDBC* e *CJDBC_I* são semelhantes na medida em que ambas essencialmente o que fazem é abrir e alocar um cursor do servidor que por sua vez cria um result set.

Nos contextos da Actualização e da Remoção os resultados foram semelhantes, porque tal como foi explicado para *CJDBC_I* as duas operações têm uma implementação semelhante. No entanto, *WJDBC* tem maior vantagem sobre *MSJDBC* do que tem *CJDBC_I*, mas isto é um assunto a discutir na comparação com *WJDBC* na secção 6.1.2.

A operação em que *WJDBC* mais se destacou foi a Leitura. Para uma quantidade reduzida de threads a vantagem é pequena, mas para quantidades maiores o desempenho de *WJDBC* é muito superior ao de *MSJDBC*, chegando a ser 10 mais rápido. A razão principal para este resultado tem a ver com as n cópias do result set que o *MSJDBC* tem que efectuar do result set. Na preparação o *MSJDBC* tem que criar n vezes mais cursores, e como é na execução que as linhas são realmente carregadas em cache, na execução têm que ser carregados n vezes mais result sets. O resultado final é a grande superioridade de desempenho do *WJDBC* sobre o *MSJDBC*.

Mais uma vez verifica-se que para o contexto da Inserção existe um maior equilíbrio nos resultados não existindo uma vantagem clara para alguma das soluções.

6.1.2 Comparação com WJDBC

No geral a solução **CJDBC_I** tem um desempenho inferior ao demonstrado pela solução *WJDBC*.

Também na generalidade, para quantidades de threads maiores a solução *CJDBC_I* perde desempenho em relação a *WJDBC*. Embora a solução *CJDBC_I* seja menos restritivo em relação ao lock e permita que mais código seja executado concorrentemente, a solução é mais pesada quanto à utilização de recursos do lado do cliente. A solução *WJDBC* partilha o mesmo objecto result set utilizando cursores cujo peso é insignificante. Um cursor no *WJDBC* não é muito mais do que uma classe que possui uma referência para um ResultSet e um inteiro que guarda a linha actual do result set. Já a solução *CJDBC_I* constrói um cache por cursor. Com a excepção da alocação de um cursor do servidor, isso é semelhante ao *MSJDBC* em que se constrói n result sets para n threads. À medida em que a quantidade de threads aumenta é cada vez mais penoso criar um cache para cada cursor de cada thread, o que se reflete numa perda de desempenho. É na Leitura que se nota mais o peso de criar os vários caches.

A situação onde existe mais equilíbrio é na inserção, existindo até uma ligeira vantagem para *CJDBC_I* que ronda os 2 a 6% para quantidades de threads maiores. Na inserção a

interacção com o servidor é mínima, não existe o fetch do result set, apenas são inseridos dados do cliente no servidor. O que se verifica no *WJDBC* para quantidades de threads maiores é que existe menos código concorrente a ser executado, pois a solução *WJDBC* bloqueia o acesso ao objecto result set quando inicia o processo de inserção. Daí surgir a vantagem para o *CJDBC_I*.

A solução **CJDBC_S** e a *WJDBC* em geral têm um desempenho semelhante, o que até era espectável pois o princípio de funcionamento de ambos é o mesmo: existe um objecto utilizado para operar sobre o dataset e que é partilhado pelos threads.

A diferença encontra-se no local onde é realizado o acesso exclusivo ao objecto partilhado: num nível mais alto para o *WJDBC*, e num nível mais baixo para o *CJDBC_S*. Obtendo o lock num nível mais alto não há tanta sensibilidade para o distinguir o código que é concorrente do que não é concorrente; todo o código executado depois do lock é executado não concorrentemente, e por isso pode acontecer estar-se a diminuir o desempenho. Obtendo o lock num nível mais baixo há a possibilidade de se identificarem as zonas do código que têm de ser executadas com acesso exclusivo e que não podem ser executadas concorrentemente (zona crítica), efectuando apenas o lock nessas zonas existe mais código concorrente. E com mais código concorrente obtemos um aumento de desempenho.

Apesar da proximidade de desempenho de ambas as soluções, nota-se uma vantagem em favor da solução *CJDBC_S*. É no contexto da Leitura que se verifica uma clara superioridade de *CJDBC_S* em relação a *WJDBC*. A razão está relacionada com o nível mais baixo de implementação dos locks e com a construção especial desta solução, em que se sabe que o cache tem todas as linhas do dataset, e por isso comunica menos com o servidor, aumentando o desempenho.

O tempo de preparação do *CJDBC_S* é mais alto porque esta solução cria o cache partilhado na instanciação, e o *WJDBC* só pede as linhas do dataset quando precisa delas, ou seja, em execução (o que também reduz o desempenho de *WJDBC* na execução, principalmente no contexto da Leitura).

6.1.3 Comparação com atrasos

O efeito que a introdução de atraso provocou na comparação dos resultados entre as soluções *MSJDBC* e *CJDBC* é quase inexistente, pelo que se pode dizer que a introdução de atraso afecta as soluções na mesma proporção, levando à obtenção de resultados semelhantes à situação de ausência de atraso.

Em relação à solução *WJDBC* notou-se uma perda de desempenho enorme, resultado que já era esperado principalmente na operação de actualização (e inserção também). O *WJDBC* bloqueia o acesso ao objecto do result set quando começa o processo de actualização de uma linha, e só o desbloqueia quando o processo termina. Com a introdução de atraso os threads mantêm o lock durante mais tempo diminuindo a concorrência, e com o resultado prático de

diminuir muito o desempenho.

Ao contrário do que se pensava inicialmente, afinal a soma das partes é maior do que o todo. Embora se introduza 8 vezes um atraso entre colunas, o atraso total verificado entre linhas usando o valor $8 \times \text{atraso}_C$ é bastante menor. A título de curiosidade o tempo total da realização do benchmark do atraso entre colunas foi de mais de 12h30 e o tempo total da realização do benchmark do atraso entre linhas foi de cerca de 6h30. O facto de a introdução de atraso entre colunas provocar mais atraso, também evidenciou mais diferenças entre os resultados na ausência de atrasos e os resultados com atraso.

6.1.4 Cache individual vs Cache partilhado

Desde o início do trabalho previu-se que a implementação de cache partilhado viria sofrer num contexto com múltiplos threads, pois numa situação em que os threads trabalhem em linhas diferentes o cache estaria continuamente a ser alterado. Este benchmark veio a confirmar esse raciocínio. O *CJDBC_SM* tem um desempenho inferior ao de *CJDBC_I*.

Do *fetch size* de 10% para 20% nota-se uma perda de desempenho do *CJDBC_S*. Isto acontece porque com um cache maior, a sua actualização torna-se numa operação mais pesada, e o maior número de linhas não chega para compensar a sua actualização. Para o *fetch size* 50% e 75% nota-se uma muito ligeira recuperação do *CJDBC_SM*, e para 100% ambos (*CJDBC_I* e *CJDBC_SM*) têm desempenhos próximos para quantidades de threads mais baixas. Este dado vem contribuir para a justificação da assunção de que a implementação do cache partilhado deveria guardar todas as linhas no cache.

Existe mais um aspecto importante a referir. Embora para o *fetch size* a 100% o *CJDBC_SM* se tenha aproximando do *CJDBC_I*, ainda ficou aquém. Mas pelo que se tinha visto na comparação com o *MSJDBC*, o *CJDBC_S* tinha melhor desempenho do que o *CJDBC_I*. Isto aconteceu porque teve que ser criada uma nova versão do *CJDBC_S* que suportasse diferentes tamanhos de *fetch size*. A versão oficial da implementação do cache partilhado está optimizada para tirar partido do facto de o cache conter todas as linhas, e não ser necessário verificar se uma linha requisitada se encontra em cache. Nos resultados da comparação da comparação entre *CJDBC_I* e *CJDBC_S* verifica-se uma grande superioridade do *CJDBC_S*, o que comprova que o cache partilhado deve conter a totalidade das linhas do result set em cache.

6.2 Conclusão

Este trabalho provou que existem soluções que permitem um acesso concorrente aos serviços da API JDBC, nomeadamente para acesso ao ResultSet. As soluções encontradas não só fornecem um mecanismo que garante um estado correcto do ResultSet num ambiente com múltiplos threads, como reduz a utilização e desperdício de recursos no cliente e no servidor.

O desempenho das soluções construídas a pensar numa execução concorrente superou o

desempenho da solução que cria um result set para cada entidade concorrente (thread).

Face aos resultados de *CJDBC_S*, *CJDBC_I* e *WJDBC* conclui-se que se houver memória disponível no cliente para conter todo o result set, deve-se utilizar a solução *CJDBC_S*, diminuindo assim a quantidade de tráfego de rede e diminuindo também a dependência sobre os cursores do servidor, que se sabe serem menos eficientes do que utilizar um result set sem cursor [16, 71]. Caso não seja possível guardar na memória do cliente todo o result set, a escolha da solução recai para o *WJDBC* se se pretender o máximo de desempenho. Porém, a solução *CJDBC_I* também garante um desempenho superior a *JDBC*.

Se tivermos em conta os resultados do benchmark que simula alguma actividade entre operações sobre o ResultSet, então conclui-se que a solução *WJDBC* deve ser evitada pois o seu desempenho sofreu um decréscimo enorme, apresentando um desempenho pior do que *JDBC*.

6.3 Trabalho relacionado

O jTDS é um driver JDBC 3.0 open-source do tipo 4 para Microsoft SQL Server (6.5, 7, 2000, 2005 e 2008) e Sybase (10, 11, 12, 15) [68]. É baseado no projecto FreeTDS (implementação em C do protocolo TDS [6]) e implementa quase a totalidade da especificação 3.0 da JDBC [67]. Quanto à concorrência suporta somente a execução concorrente de Statements.

A solução ResultSet Wrapper (*WJDBC*) teve como inspiração o trabalho realizado por Óscar Narciso Mortágua Pereira, Rui Luís Andrade Aguiar e Maribel Yasmina Campos Alves Santos, apresentado no artigo "Assessment of a Enhanced ResultSet Component for Accessing Relational Databases" [82]. A ideia principal é a mesma: encapsular um result set, controlando o acesso concorrente a ele, através da criação de cursores cliente. Na solução desse artigo a salvaguarda do contexto é realizada centralmente na entidade EResultSet. Basicamente ela possui uma memória que é indexada pelo identificador do cursor, e que para cada posição tem guardada a linha do result set para esse cursor. Se se verificar que o cursor activo foi alterado, é realizada a salvaguarda do contexto do cursor anterior e restaurado o contexto do novo. O *WJDBC* é diferente no sentido que a gestão do contexto é realizado de modo distribuído, cabendo a cada cursor guardar o seu contexto, apenas a verificação de alteração do cursor cliente é realizada centralmente. O *WJDBC* é assim uma implementação mais flexível e escalável, pois no EResultSet tem que ser definido um tamanho para a memória que guarda os contextos.

6.4 Trabalho Futuro

Foi apenas implementada uma pequena porção da API JDBC, daí a utilização do driver produzido tem que acontecer com restrições (por exemplo, só alguns tipos de dados SQL são suportados). Por isso, de modo a permitir a utilização do driver num ambiente de produção, devem ser adicionadas mais funcionalidades.

O trabalho apresentado neste documento está essencialmente centrado na implementação concorrente da interface *ResultSet*. No entanto, a API JDBC possui muitas outras que podem beneficiar de um estudo que leve a uma implementação concorrente. Por exemplo as interfaces *Statement* e *PreparedStatement*.

Um outro aspecto em que se pode trabalhar no futuro é em melhorar a implementação do TDS, e para isso seria importante saber as zonas críticas ao nível do desempenho. Para descobrir essas zonas críticas colocar-se-iam pontos de benchmark no funcionamento interno do driver. Assim em vez de se saber quanto tempo demora a realizar uma tarefa complexa, passa-se a ter a noção do tempo que cada unidade que a constitui leva a completar a sua função. A avaliação das unidades permitiria melhorar a construção das que se revelassem menos eficientes.

Glossário

ambiente multithreaded Aplicação que executa várias tarefas simultaneamente utilizando threads separados para cada tarefa. 10, 20

Application Programming Interface (API) Conjunto de regras e especificações que estabelecem o modo como um software disponibiliza as suas funcionalidades. 1, 3

batch Conjunto de uma ou mais statements Transact-SQL enviadas ao SQL Server para execução. 15, 19

boilerplate Este termo quando aplicado a código-fonte refere-se a código que pode ser re-utilizado sem sofrer alterações. 28

bulk insert Método eficiente de preenchimento de uma tabela, invocado por um cliente num servidor. 101

cache hit ratio Percentagem de acesso à cache em que o elemento procurado é lá encontrado. 28

classpath Lista com os directórios e ficheiros jar, utilizada pela Java Virtual Machine para encontrar classes e pacotes Java. 5, 29

Data Manipulation Language Linguagem que define comandos para actualizar, inserir e remover informação num base de dados. 16

database engine Serviço principal responsável pelas tarefas de armazenamento, gestão e segurança dos dados [39]. 2, 3, 13

Database Management System (DBMS) Sistema que permite criar, gerir e utilizar uma base de dados. 4, 5, 13

dataset Conjunto de dados, normalmente apresentados numa forma tabular. 1, 6, 28, 29, 33, 76, 79

fetch Pedido e carregamento de linhas de um dataset. 25, 27, 76, 79

garbage collector Thread que corre em background numa aplicação Java e que liberta a memória de objecto que já não estejam a ser utilizados. 40

- Java Virtual Machine** Máquina virtual capaz de executar bytecode Java. 3, 4
- lock** Bloqueio do acesso a um objecto partilhado, permitindo acesso exclusivo à entidade que mantêm o bloqueio. 22, 25, 27, 78, 79
- Open Database Connectivity (ODBC)** Interface de software *standard* para aceder a um DBMS. 4
- overhead** Processamento adicional requerido para executar uma determinada tarefa. 29
- override** Reimplementação de um método de uma superclasse realizada por uma das suas subclasses. 26, 28
- query** Pedido de informação a uma base de dados ou a um sistema de informação. 4
- Relational Database Management System (DBMS)** Sistema que permite criar, gerir e utilizar uma base de dados relacional. 2, 3
- Remote Procedure Call (RPC)** Procedimento executado num sistema remoto. No âmbito das bases de dados significa a invocação de um stored procedure. 28, 76
- ResultSet** Interface Java do pacote `java.sql` que permite operar sobre o resultado da execução de uma statement SQL. 13
- statement SQL** String com uma expressão numa linguagem que o servidor entende. 2, 6, 13, 16, 17, 19, 23, 25, 34
- stored procedure** Sub-rotina constituída por comandos T-SQL, disponível num sistema de base de dados relacional. 1, 101, 107
- Tabular Data Stream** Protocolo utilizado na comunicação entre a aplicação cliente e o SQL Server. 1, 13, 14, 24, 115
- User Defined Function (UDF)** Função criada pelo utilizador que pode ser utilizada em instruções SQL. 101

Acrónimos

API Application Programming Interface

CJDBC Designação genérica para a implementação concorrente do ResultSet

CJDBC_I Designação para a solução concorrente do ResultSet com cache individual

CJDBC_S Designação para a solução concorrente do ResultSet com cache partilhado

CLI Client Level Interface

DBMS Database Management System

JVM Java Virtual Machine

MSJDBC Designação para a solução que cria um ResultSet por thread

ODBC Open Database Connectivity

RBMS Relational Database Management System

RPC Remote Procedure Call

SQL Structured Query Language

TDS Tabular Data Stream

UDF User Defined Function

WJDBC Designação para a solução ResultSet Wrapper

Bibliografia

- [1] Inc. Advanced Micro Devices. Optimizing java performance in a virtual machine environment. <http://developer.amd.com/documentation/articles/pages/optimizingjavainvmenvironment.aspx>, 2009.
- [2] Scott W. Ambler. The object-relational impedance mismatch. <http://www.agiledata.org/essays/impedanceMismatch.html>, 2009.
- [3] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19:105–190, 1988.
- [4] Malcolm P. Atkinson, Laurent Daynès, Mick J. Jordan, Tony Printezis, and Susan Spence. An orthogonally persistent java, 1996.
- [5] Malcolm P. Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The Vldb Journal*, 4:319–401, 1995.
- [6] Brian Bruns. Freetds. <http://www.freetds.org/>, 2011. [Online; accessed May-2011].
- [7] Jian Chen and Qiming Huang. Eliminating the impedance mismatch between relational systems and object-oriented programming languages. In *in Proce. the 6th International Hong Kong Database Workshop*, 1995.
- [8] William R. Cook and Ali H. Ibrahim. Integrating programming languages & databases: What’s the problem? 2005.
- [9] Microsoft Corporation. Microsoft®SQL Server®2008. <http://www.microsoft.com/sqlserver/2008/en/us/>.
- [10] Microsoft Corporation. Microsoft SQL Server JDBC Driver 3.0. <http://www.microsoft.com/downloads/en/details.aspx?FamilyID=a737000d-68d0-4531-b65d-da0f2a735707>, April 2010. [Online; accessed September-2010].
- [11] Microsoft Corporation. Tabular Data Stream Protocol Specification. [http://msdn.microsoft.com/en-us/library/dd304523\(prot.13\).aspx](http://msdn.microsoft.com/en-us/library/dd304523(prot.13).aspx), 2010. [Online; accessed November-2010].

- [12] Microsoft Corporation. All headers rule definition. <http://msdn.microsoft.com/en-us/library/cc448573.aspx>, 2011. [Online; accessed May-2011].
- [13] Microsoft Corporation. Browse mode. [http://msdn.microsoft.com/en-us/library/aa936959\(SQL.80\).aspx](http://msdn.microsoft.com/en-us/library/aa936959(SQL.80).aspx), 2011. [Online; accessed May-2011].
- [14] Microsoft Corporation. Building the connection url: Sql server 2008. <http://msdn.microsoft.com/pt-pt/library/ms378428.aspx>, 2011. [Online; accessed May-2011].
- [15] Microsoft Corporation. Cursor Concurrency (Database Engine). <http://msdn.microsoft.com/en-us/library/ms191493.aspx>, 2011. [Online; accessed May-2011].
- [16] Microsoft Corporation. Cursor implementations. [http://msdn.microsoft.com/en-us/library/ms189546\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms189546(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [17] Microsoft Corporation. Cursor stored procedures. <http://msdn.microsoft.com/en-us/library/ms187801.aspx>, 2011. [Online; accessed May-2011].
- [18] Microsoft Corporation. Cursor types (database engine). [http://msdn.microsoft.com/en-us/library/ms188644\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms188644(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [19] Microsoft Corporation. Cursors (Database Engine). [http://msdn.microsoft.com/en-us/library/ms191179\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms191179(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [20] Microsoft Corporation. Cursors (Transact-SQL). [http://msdn.microsoft.com/en-us/library/ms181441\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms181441(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [21] Microsoft Corporation. Data types (transact-sql). <http://msdn.microsoft.com/en-us/library/ms187752.aspx>, 2011. [Online; accessed May-2011].
- [22] Microsoft Corporation. Deprecated database engine features in sql server 2008 r2. <http://msdn.microsoft.com/en-us/library/ms143729.aspx>, 2011.
- [23] Microsoft Corporation. Dynamic Cursors (Database Engine). [http://msdn.microsoft.com/en-us/library/ms189099\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms189099(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [24] Microsoft Corporation. Fast Forward-only Cursors (Database Engine). [http://msdn.microsoft.com/en-us/library/ms187502\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms187502(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [25] Microsoft Corporation. Fetching and Scrolling. [http://msdn.microsoft.com/en-us/library/ms187881\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms187881(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [26] Microsoft Corporation. Forward-only Cursors (Database Engine). [http://msdn.microsoft.com/en-us/library/ms178033\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms178033(v=SQL.100).aspx), 2011. [Online; accessed May-2011].

- [27] Microsoft Corporation. Keyset-driven Cursors (Database Engine). <http://msdn.microsoft.com/en-us/library/ms179409.aspx>, 2011. [Online; accessed May-2011].
- [28] Microsoft Corporation. Login7. [http://msdn.microsoft.com/en-us/library/dd304019\(v=PROT.13\).aspx](http://msdn.microsoft.com/en-us/library/dd304019(v=PROT.13).aspx), 2011. [Online; accessed May-2011].
- [29] Microsoft Corporation. Nbcrow. [http://msdn.microsoft.com/en-us/library/dd304783\(v=PROT.13\).aspx](http://msdn.microsoft.com/en-us/library/dd304783(v=PROT.13).aspx), 2011. [Online; accessed May-2011].
- [30] Microsoft Corporation. Odbc-open database connectivity overview. <http://support.microsoft.com/kb/110093/en-us>, 2011. [Online; accessed May-2011].
- [31] Microsoft Corporation. Packet data token stream definition. [http://msdn.microsoft.com/en-us/library/dd340794\(v=PROT.13\).aspx](http://msdn.microsoft.com/en-us/library/dd340794(v=PROT.13).aspx), 2011. [Online; accessed May-2011].
- [32] Microsoft Corporation. Packet header: Type. [http://msdn.microsoft.com/en-us/library/dd304214\(v=PROT.13\).aspx](http://msdn.microsoft.com/en-us/library/dd304214(v=PROT.13).aspx), 2011. [Online; accessed May-2011].
- [33] Microsoft Corporation. setresponsebuffering method (sqlserverstatement). [http://msdn.microsoft.com/en-us/library/bb879939\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/bb879939(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [34] Microsoft Corporation. Setting the connection properties: Sql server 2008. <http://msdn.microsoft.com/pt-pt/library/ms378988.aspx>, 2011. [Online; accessed May-2011].
- [35] Microsoft Corporation. sp_cursor (transact-sql). <http://msdn.microsoft.com/en-us/library/ff848759.aspx>, 2011. [Online; accessed May-2011].
- [36] Microsoft Corporation. sp_cursorclose (transact-sql). <http://msdn.microsoft.com/en-us/library/ff848800.aspx>, 2011. [Online; accessed May-2011].
- [37] Microsoft Corporation. sp_cursorfetch (transact-sql). <http://msdn.microsoft.com/en-us/library/ff848736.aspx>, 2011. [Online; accessed May-2011].
- [38] Microsoft Corporation. sp_cursoropen (transact-sql). <http://msdn.microsoft.com/en-us/library/ff848737.aspx>, 2011. [Online; accessed May-2011].
- [39] Microsoft Corporation. Sql server database engine (sql server 2008). [http://msdn.microsoft.com/en-us/library/ms187875\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms187875(v=SQL.100).aspx), 2011.
- [40] Microsoft Corporation. SQLServerResultSet Members. [http://msdn.microsoft.com/en-us/library/ms378188\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms378188(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [41] Microsoft Corporation. Sqlserverstatement class. [http://msdn.microsoft.com/en-us/library/ms378995\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms378995(v=SQL.100).aspx), 2011. [Online; accessed May-2011].

- [42] Microsoft Corporation. Static cursors (database engine). <http://msdn.microsoft.com/en-us/library/ms191286.aspx>, 2011. [Online; accessed May-2011].
- [43] Microsoft Corporation. Understanding Cursor Types. [http://msdn.microsoft.com/en-us/library/ms378405\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/ms378405(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [44] Microsoft Corporation. Using adaptive buffering. [http://msdn.microsoft.com/en-us/library/bb879937\(v=SQL.100\).aspx](http://msdn.microsoft.com/en-us/library/bb879937(v=SQL.100).aspx), 2011. [Online; accessed May-2011].
- [45] Oracle Corporation. Pjama. <http://labs.oracle.com/forest/opj.main.html>, 2000.
- [46] Oracle Corporation. JSR-000221 JDBC 4.0. <http://jcp.org/aboutJava/communityprocess/final/jsr221/index.html>, 2006. [Online; accessed September-2010].
- [47] Oracle Corporation. Java Platform Standard Ed. 6 - Package java.sql. <http://download.oracle.com/javase/6/docs/api/java/sql/package-summary.html>, 2010. [Online; accessed April-2011].
- [48] Oracle Corporation. Types of JDBC technology drivers. <http://java.sun.com/products/jdbc/driverdesc.html>, 2010. [Online; accessed November-2010].
- [49] Oracle Corporation. Class AtomicInteger. <http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/atomic/AtomicInteger.html>, 2011. [Online; accessed April-2011].
- [50] Oracle Corporation. Class reentrantlock. <http://download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/ReentrantLock.html>, 2011. [Online; accessed May-2011].
- [51] Oracle Corporation. Interface connection. <http://download.oracle.com/javase/6/docs/api/java/sql/Connection.html>, 2011. [Online; accessed May-2011].
- [52] Oracle Corporation. Jdbc overview. <http://www.oracle.com/technetwork/java/overview-141217.html>, 2011. [Online; accessed May-2011].
- [53] Oracle Corporation. Jdk 6 java database connectivity (jdbc)-related apis & developer guides. <http://download.oracle.com/javase/6/docs/technotes/guides/jdbc/>, 2011. [Online; accessed May-2011].
- [54] Oracle Corporation. Package java.sql. <http://download.oracle.com/javase/6/docs/api/java/sql/package-summary.html>, 2011. [Online; accessed May-2011].
- [55] Oracle Corporation. Package javax.sql. <http://download.oracle.com/javase/6/docs/api/javax/sql/package-summary.html>, 2011. [Online; accessed May-2011].

- [56] Oracle Corporation. `System.gc()` (java platform se 6). [http://download.oracle.com/javase/6/docs/api/java/lang/System.html#gc\(\)](http://download.oracle.com/javase/6/docs/api/java/lang/System.html#gc()), 2011. [Online; accessed May-2011].
- [57] D. Crocker and P. Overell. [RFC] Augmented BNF for Syntax Specifications: ABNF. <http://www.ietf.org/rfc/rfc4234.txt>, 2005. [Online; accessed November-2010].
- [58] Maydene Fisher, Jon Ellis, and Jonathan Bruce. *JDBC API Tutorial and Reference (Third Edition)*. Addison Wesley, 2003.
- [59] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.
- [60] Roedy Green. Garbage collection: Java glossary. <http://mindprod.com/jgloss/garbagecollection.html>, 2011. [Online; accessed May-2011].
- [61] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency In Practice*. Addison-Wesley Professional, 2006.
- [62] Qiming Huang and Jian Chen. Eliminating the impedance mismatch between relational systems and object-oriented programming language. 1995.
- [63] Sybase Inc. TDS 5.0 Functional Specification. <http://www.sybase.com/content/1040983/Sybase-tds38-102306.pdf>, 2006. [Online; accessed November-2010].
- [64] Wikimedia Foundation Inc. Tabular Data Stream. http://www.enotes.com/topic/Tabular_Data_Stream, 2010. [Online; accessed November-2010].
- [65] Wikipedia Foundation Inc. JDBC Driver. http://en.wikipedia.org/wiki/JDBC_driver, 2010. [Online; accessed November-2010].
- [66] The jTDS Project. System stored procedures (jtds documentation). <http://jtds.sourceforge.net/apiCursors.html>. [Online; accessed December-2010].
- [67] The jTDS Project. jtds feature matrix. <http://jtds.sourceforge.net/features.html>, 2011. [Online; accessed May-2011].
- [68] The jTDS Project. jTDS JDBC Driver. <http://jtds.sourceforge.net/>, 2011. [Online; accessed May-2011].
- [69] Easysoft Limited. What is odbc? http://www.easysoft.com/developer/interfaces/odbc/linux.html#what_is_odbc, 2011. [Online; accessed May-2011].
- [70] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. Implementing orthogonally persistent java. In *Workshop on Persistent Object Systems*, pages 247–261, 2000.

- [71] Brad McGehee. Performance tuning sql server cursors. http://www.sql-server-performance.com/tips/cursors_p1.aspx, January 2007. [Online; accessed May-2011].
- [72] Linda Null and Julia Lobur. *The Essentials of Computer Organization and Architecture*. Jones and Bartlett Publishers, 2003.
- [73] Scott Oaks and Henry Wong. *Java Threads, Third Edition*. O'Reilly Media, 2004.
- [74] Onstrategies.com. Evans data rates popularity of relational databases. <http://www.onstrategies.com/CURRENT-NEWS/Evans-Data-Rates-Popularity-of-Relational-Databases.html>, 2008.
- [75] Mahmoud Parsian. *JDBC Recipes: A Problem-Solution Approach (Problem-Solution Approach)*. Apress, Berkely, CA, USA, 2005.
- [76] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface (Third Edition)*. Morgan Kaufmann, 2007.
- [77] George Reese. *Database Programming with JDBC & Java, Second Edition*. O'Reilly Media, 2001.
- [78] Inc. Sybase. Embeddedsql. <http://www.sybase.com/products/archivedproducts/embeddedsql>, 2011.
- [79] ISO/IEC. Information technology. Database languages - sql - part 3: Call-level interface (sql/cli). technical report 9075-3:1995. *ISO/IEC*, 1995.
- [80] Inc. Unicode. The unicode consortium. <http://unicode.org>, 2011. [Online; accessed December-2010].
- [81] Robert Vieira. *Professional SQL Server 2005 Programming*. Wrox, 2006.
- [82] Óscar Pereira, Rui Aguiar, and Maribel Santos. Assessment of a enhanced resultset component for accessing relational databases. 2010.

Apêndice A

Estudo do `SQLServerResultSet`

Aquando da criação do objecto `statement` é definido o tipo de result set que é criado. Esse tipo determina o modo como os dados são carregados do servidor; podem ser usados cursores de servidor ou não, podem ser carregados todos os dados de uma só vez ou podem ser carregados conforme a aplicação vai os vai requisitando.

Nesta secção é apresentado um estudo da classe `SQLServerResultSet`, que corresponde à classe do driver da Microsoft que implementa a interface `java.sql.ResultSet`. O estudo concentra-se no modo de interacção do `ResultSet` com o SQL Server.

A.1 Cursores no SQL Server

As operações numa base de dados relacional actuam sobre um conjunto de linhas que satisfazem a cláusula `WHERE` de uma `statement`, no entanto muitas aplicações precisam de trabalhar com blocos mais pequenos ou até com uma linha de cada vez. Os cursores fornecem esse mecanismo, permitindo [19]:

- Posicionamento numa linha específica.
- Acesso a uma linha ou um bloco de linhas a partir da localização actual no result set.
- Modificar (actualizar, remover) linhas.
- Diferentes níveis de visibilidade às modificações realizadas por outros no result set.
- Acesso ao result set a partir de Transact-SQL em scripts, stored procedures e triggers.

A.1.1 Fetching e Scrolling

A operação de obter uma linha a partir do cursor designa-se por *fetch*. Um cursor é classificado quanto ao tipo de fetch que suporta [25]:

- Forward-only

As linhas são obtidas sequencialmente da primeira até à última.

- Scrollable
Qualquer linha pode ser obtida em qualquer direcção.

Um cursor Forward-only suporta a seguinte operação de fetch:

- FETCH NEXT
Obtém a próxima linha.

Um cursor Scrollable suporta as seguintes operações de fetch:

- FETCH NEXT
Obtém a próxima linha.
- FETCH FIRST
Obtém a primeira linha.
- FETCH PRIOR
Obtém a linha anterior.
- FETCH LAST
Obtém a última linha.
- FETCH ABSOLUTE n
Obtém a linha n a partir da primeira linha.
- FETCH RELATIVE n
Obtém a linha n a partir da linha actual.

A.1.2 Concorrência

O SQL Server suporta 4 tipos de concorrência [15]:

- READ_ONLY
A actualização usando o cursor não é permitida e não são obtidos locks nas linhas do result set.
- OPTIMISTIC WITH VALUES
Não são obtidos locks nas linhas. Quando uma actualização ocorre os valores actuais das colunas da linha são comparados com os valores anteriormente carregados, se forem iguais procede-se à actualização, caso contrário é lançado um erro.
- OPTIMISTIC WITH ROW VERSIONING
A tabela a actualizar tem que possuir uma coluna do tipo timestamp. Numa actualização valores de timestamp são comparados para determinar se a linha foi alterada ou não por outros, e em caso negativo os novos valores são guardados.

- **SCROLL LOCKS**

O cursor lê a linha obtendo um update lock. Se o cursor for aberto no decorrer de uma transacção o lock mantém-se até ocorrer um commit ou um roll-back. Se o cursor for aberto fora de uma transacção o lock da linha é liberto quando for obtida uma outra linha.

A.1.3 Tipos de cursor

O SQL Server suporta os seguintes tipos de cursor [18]:

Forward-only [26]

Não suporta scrolling, as linhas são obtidas sequencialmente da primeira para a última. As linhas só são carregadas quando são pedidas. As modificações provenientes das operações de inserção, actualização e remoção, realizadas pelo próprio ou por outros são visíveis. O SQL Server implementa uma versão designada por Fast Forward-only, com optimização de desempenho [24].

Static [42]

Quando o cursor é aberto é criada uma cópia do result set na base de dados **tempdb**. Uma vez que trabalha com uma cópia, através deste tipo de cursor as modificações não são visíveis. Também é conhecido como cursor insensitive e cursor snapshot.

Keyset-driven [27]

São usadas chaves para aceder às linhas da tabela. A tabela tem que possuir uma ou mais linhas que permitam identificar unicamente uma linha. O keyset (conjunto de chaves) é criado como uma tabela na base de dados **tempdb**. As modificações, próprias ou externas, nas colunas que não são chave são visíveis, mas as inserções externas não são visíveis.

Dynamic [23]

Todas as modificações são visíveis. Os valores e a ordem das linhas pode ser alterada em cada fetch. Este tipo de cursor é o que tem mais baixo desempenho, principalmente para maiores quantidades de dados, e têm também problemas de concorrência porque em cada fetch o result set é reconstruído, e por isso em geral deve-se evitar a sua utilização [81]. No entanto, para quantidades de dados pequenas o result set trabalha a partir da RAM, sendo nesta situação mais rápido do que o keyset, que trabalha a partir do disco (este utiliza uma tabela temporária na tempdb) [81].

A.2 Tipos de cursor por result set

As características requisitadas ao result set determinam a interacção que o SQLServerResultSet tem com o servidor. A Tabela A.1 mostra que tipo de cursor de servidor é criado para cada característica do result set [43]. A API JDBC permite definir os requisitos do result set

quanto à navegabilidade (forward-only ou scrollable) e à concorrência (actualizável ou só de leitura), mas o `SQLServerResultSet` suporta ainda mais uma característica, buffering, que é indicada na coluna com o mesmo nome na Tabela A.1 e explicada na secção A.3.

Tabela A.1: *Tipos de cursor suportados pelo driver*

Tipo	Cursor	Característica	Buffering	Descrição
TYPE_FORWARD_ONLY / CONCUR_READ_ONLY	N/A	Forward-only read-only	full	Permite apenas uma passagem, da primeira até à última linha, pelo result set. Este é o comportamento por pré-definição. O driver lê todo o result set para a memória quando a statement é executada.
TYPE_FORWARD_ONLY / CONCUR_READ_ONLY	N/A	Forward-only read-only	adaptive	Permite apenas uma passagem, da primeira até à última linha, pelo result set. O driver lê as linhas do result set conforme vão sendo pedidas, minimizando a memória gasta pelo cliente.
TYPE_FORWARD_ONLY / CONCUR_READ_ONLY	Fast Forward	Forward-only read-only	N/A	Permite apenas uma passagem, da primeira até à última linha, pelo result set usando o cursor do servidor. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .
TYPE_FORWARD_ONLY / CONCUR_UPDATABLE	Dynamic Forward-only	Forward-only updatable	N/A	Permite apenas uma passagem, da primeira até à última linha, pelo result set, permitindo também a actualização das linhas. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .
TYPE_SCROLL_INSENSITIVE	Static	Scrollable read-only	N/A	O result set não é modificável, e as modificações externas não são visíveis. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .
TYPE_SCROLL_SENSITIVE / CONCUR_READ_ONLY	Keyset	Scrollable read-only	N/A	As actualizações externas são visíveis, as remoções aparecem como dados inexistentes e as inserções externas não são visíveis. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .
TYPE_SCROLL_SENSITIVE CONCUR_UPDATABLE / CONCUR_SS_SCROLL_LOCKS CONCUR_SS_OPTIMISTIC_CC CONCUR_SS_OPTIMISTIC_CCVAL	Keyset	Scrollable updatable.	N/A	As actualizações internas e externas são visíveis, as remoções aparecem como dados inexistentes e as inserções externas não são visíveis. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .

Tabela A.1: *Tipos de cursor suportados pelo driver*

Tipo	Cursor	Característica	Buffering	Descrição
TYPE_SS_DIRECT_FORWARD_ONLY	N/A	Forward-only read-only	full or adaptive	Fornecer um cursor no cliente que não permite modificações e cujos dados do result set podem ser todos carregados na execução da statement. Não é criado uma cursor no servidor.
TYPE_SS_SERVER_CURSOR_FORWARD_ONLY	Fast Forward	Forward-only	N/A	Acende rapidamente todos os dados usando um cursor no servidor. Permite modificações de for usado com CONCUR_UPDATABLE. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> . É possível usar adaptive buffering se o método setResponseBuffering da classe SQLServerStatement for explicitamente invocado com o argumento "adaptive".
TYPE_SS_SCROLL_STATIC	Static	As actualizações externas não são reflectivas.	N/A	Esta opção é equivalente a TYPE_SCROLL_INSENSITIVE. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .
TYPE_SS_SCROLL_KEYSET / CONCUR_READ_ONLY	Keyset	Scrollable read-only	N/A	As actualizações externas são visíveis, as remoções aparecem como dados inexistentes e as inserções externas não são visíveis. Esta opção é equivalente a TYPE_SCROLL_SENSITIVE. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .
TYPE_SS_SCROLL_KEYSET / CONCUR_UPDATABLE CONCUR_SS_SCROLL_LOCKS CONCUR_SS_OPTIMISTIC_CC CONCUR_SS_OPTIMISTIC_CCVAL	Keyset	Scrollable updatable.	N/A	As actualizações internas e externas são visíveis, as remoções aparecem como dados inexistentes e as inserções não são visíveis. Esta opção é equivalente a TYPE_SCROLL_SENSITIVE. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .
TYPE_SS_SCROLL_DYNAMIC / CONCUR_READ_ONLY	Dynamic	Scrollable read-only	N/A	As actualizações e inserções externas são visíveis, e as remoções aparecem como dados inexistentes. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .

Tabela A.1: *Tipos de cursor suportados pelo driver*

Tipo	Cursor	Característica	Buffering	Descrição
TYPE_SS_SCROLL_DYNAMIC / CONCUR_UPDATABLE CON- CUR_SS_SCROLL_LOCKS CON- CUR_SS_OPTIMISTIC_CC CON- CUR_SS_OPTIMISTIC_CCVAL	Dynamic	Scrollable updatable	N/A	As actualizações e inserções internas e externas são visíveis, e as remoções aparecem como dados inexistentes. As linhas são carregadas em blocos com o tamanho <i>fetch size</i> .

O `SQLServerResultSet` para além de suportar os tipos de result set definidos pela interface *ResultSet*, adiciona alguns tipos que permitem requisitar explicitamente tipos específicos que existem no SQL Server, tanto ao nível da navegação (forward-only, scrollable), como ao nível do tipo de concorrência [40]:

- `CONCUR_SS_OPTIMISTIC_CC`
Leitura e escrita com concorrência optimística (row versioning) e sem locks de linha.
- `CONCUR_SS_OPTIMISTIC_CCVAL`
Leitura e escrita com concorrência optimística (values) e sem locks de linha.
- `CONCUR_SS_SCROLL_LOCKS`
Leitura e escrita com concorrência optimística e com locks de linha.
- `TYPE_SS_DIRECT_FORWARD_ONLY`
Cursor do tipo *fast forward-only*, só de leitura.
- `TYPE_SS_SCROLL_DYNAMIC`
Cursor do tipo *dynamic*.
- `TYPE_SS_SCROLL_KEYSET`
Cursor do tipo *keyset*.
- `TYPE_SS_SCROLL_STATIC`
Cursor do tipo *static*.
- `TYPE_SS_SERVER_CURSOR_FORWARD_ONLY`
Cursor do tipo *fast forward-only*, só de leitura.

A.3 Adaptive Buffering

O *adaptive buffering* é uma funcionalidade introduzida no Microsoft SQL Server 2005 JDBC Driver versão 1.2, que tem como finalidade o carregamento de grandes quantidades de dados sem a necessidade de utilizar cursores do servidor [44].

O que isto significa é que existem dois modos de carregamento de dados: **adaptive** e **full**. No modo adaptive é carregada a menor quantidade possível de dados, enquanto que no modo full todo o result set é lido do servidor em *run time*.

O acesso a esta funcionalidade é realiza pela utilização do método `setResponseBuffering` [33] da classe *SQLServerStatement* [41], que recebe uma String (full ou adaptive) que indica o modo desejado.

A classe *SQLServerStatement* é a implementação da interface `java.sql.Statement`.

A principal motivação é diminuir a quantidade de memória utilizada pela aplicação, e até evitar situações em que a memória esgota e é lançado o erro **OutOfMemoryError**, quando a aplicação JDBC trabalha com *queries* que produzem resultados muito grandes.

Apêndice B

Tabular Data Stream

O *Tabular Data Stream* (TDS) é um protocolo da camada aplicação utilizado para transferir informação entre um servidor de base de dados e um cliente [11, 63]. Foi desenhado e desenvolvido em 1984 pela Sybase Inc. para ser utilizado no servidor SQL da empresa, e mais tarde foi também desenvolvido pela Microsoft para ser utilizado no Microsoft SQL Server [64].

B.1 Mensagens

Como qualquer protocolo de rede, o TDS efectua a comunicação usando troca de mensagens. Existem duas categorias de mensagens: mensagens do cliente e mensagens do servidor.

Resumidamente as **mensagens do cliente** são:

Pre-login

Handshake que tem de ocorrer antes do *login*, e que configura alguns parâmetros tais como a encriptação.

Login

Mensagem que inicia o estabelecimento da comunicação com o servidor. Como resposta, o servidor informa o cliente se aceitou ou rejeitou o pedido de comunicação.

SQL Command

Mensagem que na zona de dados contém um comando SQL ou *batch* de comandos SQL, representado numa *String* codificada em Unicode [80].

SQL Command com Dados Binários

Mensagem que faz um pedido de execução de uma operação *bulk insert* usando um comando SQL seguido de dados binários. O comando também é representado numa *String* codificada em Unicode.

Remote Procedure Call (RPC)

Mensagem que faz um pedido de execução de um *stored procedure* ou uma *UDF*. A mensagem contém o nome, opções e parâmetros do RPC.

Attention signal

Mensagem que cancela a execução de um comando.

Resumidamente as **mensagens do servidor** são:

Pre-login response

Resposta a uma mensagem de *pre-login*.

Login response

Resposta a uma mensagem de *login*. Contém informação sobre as características do servidor, informação opcional e mensagens de erro.

Row data

Resposta com os dados devolvidos pela execução de um comando. Esta mensagem é precedida por uma descrição dos nomes das colunas e dos tipos de dados.

Return status

Resposta com o valor do estado de um RPC. Também é usada para enviar o estado do resultado da execução de uma instrução T-SQL.

Return parameters

Resposta com os valores dos parâmetros de saída de um RPC.

Response completion

Resposta que indica o fim de um conjunto de resultados.

Error e Info

Resposta que transmite mensagens de erro ou mensagens informativas.

Attention Acknowledgment

Resposta que confirma a recepção de um cancelamento de execução de um comando.

B.2 Pacotes

Cada mensagem é constituída por um ou mais pacotes. Cada pacote tem um tamanho máximo cujo valor é determinado na mensagem de *login*. Todos os pacotes da mensagem excepto o último têm de ter um tamanho igual ao valor do tamanho máximo negociado. Cada pacote é constituído por um cabeçalho (*packet header*) e por uma zona de dados (*packet data*).

B.2.1 Cabeçalho

Corresponde aos primeiros **8 bytes** do pacote. Os campos do cabeçalho estão representados na seguinte tabela. O valor em cima de cada campo indica o número de bytes desse campo.

Tabela B.1: *Campos do cabeçalho TDS*

1	1	2	2	1	1
Type	Status	Length	SPID	PacketId	Window

Descrição dos campos do cabeçalho:

Type

Define o tipo de mensagem.

Status

Indica o estado da mensagem (por exemplo, indica se a mensagem terminou).

Length

Indica o tamanho do pacote (incluindo o tamanho do cabeçalho).

SPID

Identifica o ID do processo no servidor correspondente à ligação actual. Este campo tem carácter opcional, pelo que nesta implementação será sempre enviado o valor 0x0000.

PacketID

Indica o número do pacote. Cada pacote enviado incrementa este valor em uma unidade.

Window

Actualmente não é utilizado, por isso tem sempre o valor 0x00.

NOTA: Todos os valores são representados em *network byte order (big-endian)* e são valores sem sinal.

B.2.2 Zona de dados

Todos os tipos de mensagens, exceptuando a *Attention signal*, a seguir ao cabeçalho têm uma zona de dados [32].

NOTA: A zona de dados também pode ser denominada de *data stream* ou apenas *stream*.

Os pacotes tem um tamanho máximo, cujo valor é determinado no *login*. O tamanho do pacote inclui o tamanho do cabeçalho.

Se uma mensagem produzir um pacote que ultrapasse o tamanho definido, terá de ser dividida por múltiplos pacotes. Cada um desses pacotes terá um cabeçalho semelhante, exceptuando os campos *Status* e *Length*. O campo *Status* terá o valor 0x0 se houverem mais pacotes da mensagem e terá o valor 0x1 se o pacote é o último da mensagem. O campo

Length terá um valor igual ao tamanho definido, para todos os pacotes excepto para o último da mensagem.

Existem dois tipos de zonas de dados: ***Token Stream*** e ***Tokenless Stream***. Um *token stream* é constituído por um ou mais *tokens*, em que cada um deles é seguido pelos dados relativos ao *token*. Um *tokenless stream* contém directamente os dados da mensagem, sem recorrer a *tokens* para os descrever.

Na tabela a seguir temos um resumo das mensagens que usam *tokens* e as que não usam.

Tabela B.2: *Indicação das mensagens que usam tokens*

Mensagem	Origem	Token
Pre-Login	Cliente	Não
Login	Cliente	Não
SQL Batch	Cliente	Não
Bulk Load	Cliente	Sim
Remote Procedure Call	Cliente	Sim
Attention	Cliente	Não
Transaction Manager Request	Cliente	Não
Pre-Login Response	Servidor	Não
Login Response	Servidor	Sim
Row Data	Servidor	Sim
Return Status	Servidor	Sim
Return Parameters	Servidor	Sim
Response Completion	Servidor	Sim
Error and Info Messages	Servidor	Sim
Attention Acknowledgment	Servidor	Não

A definição da gramática dos *streams* (*token* e *tokenless*) é especificada usando *Augmented Backus-Naur Form* [57].

B.3 *Tokenless Streams*

Um *tokenless stream* contém directamente os dados da mensagem, sem recorrer a *tokens* para os descrever.

A seguir é descrito o formato da zona de dados, das mensagens com *tokenless streams*.

B.3.1 Pre-Login

O *stream* desta mensagem é constituído por uma sequência de opções seguidas dos dados relativos a essas opções. Cada opção tem três campos: *Type*, *Position* e *Length*. *Type*

identifica a opção, *Position* indica a posição que a opção ocupa nos dados e *Length* indica o número de bytes que opção ocupa nos dados.

Tabela B.3: *Opções da mensagem de Pre-Login*

Opção	Valor	Descrição
VERSION	0x00	Versão do remetente. Normalmente usado para <i>debugging</i> .
ENCRYPTION	0x01	Negociar encriptação.
INSTOPT	0x02	Nome da instância do SQL Server.
THREADID	0x03	Id do thread da aplicação cliente. Usado para <i>debugging</i> .
TERMINATOR	0xFF	Assinala o fim da mensagem de Pre-Login.

Do que se conseguiu apurar apenas VERSION e ENCRYPTION são obrigatórios, e uma vez que as restantes opções de momento são irrelevantes, os pacotes de Pre-Login do driver só irão conter estas duas opções. Para além disso, ainda não será considerada a utilização de encriptação.

B.3.2 Login

Este *stream* define as regras de autenticação entre o cliente e o servidor. O seu tamanho não deve ultrapassar os 128-1 bytes.

A definição deste *stream* possui várias regras que podem ser consultadas em [28], das quais se destacam a *OffsetLength* e a *Data*. Estas duas regras definem os parâmetros concretos do login, tais como a base de dados a utilizar ou o nome de utilizador. A regra *Data* possui os bytes que representam os dados dos parâmetros e a regra *OffsetLength* define a posição e comprimento de cada parâmetro.

B.3.3 SQLBatch

Este *stream* define o formato de uma mensagem SQL Batch.

A definição deste stream é composta por uma regra ALL_HEADERS¹ seguida de um stream em Unicode que contém o comando SQL.

B.4 Token Streams

As mensagens mais complexas (por exemplo, os dados do result set) são construídas usando tokens. Um token consiste num byte que funciona como identificador, seguido de dados específicos ao token.

Existem os seguintes tokens [31]:

¹Alguns streams TDS podem ser precedidos de vários cabeçalhos. A regra ALL_HEADERS é utilizada para especificar esses cabeçalhos[12].

Tabela B.4: *Packet Data Token Streams*

Nome	Descrição
ALTMETADATA	Descreve o tipo de dados, tamanho e nome da coluna que resulta de uma SQL Statement que gera totais.
ALTROW	Usado para enviar uma linha com totais, cujo formato é descrito pelo token ALTMETADATA.
COLINFO	Descreve a informação da coluna em Browse Mode [13], <code>sp_cursoropen</code> e <code>sp_cursorfetch</code> .
COLMETADATA	Descreve o result set para interpretação dos tokens ROW que lhe seguem.
DONE	Indica que uma SQL Statement foi terminada.
DONEINPROC	Indica que uma SQL Statement de um stored procedure foi terminada.
DONEPROC	Indica que um stored procedure terminou.
ENVCHANGE	Notificação de uma alteração de ambiente (por exemplo, base de dados, língua).
ERROR	Usado para enviar uma mensagem de erro ao cliente.
INFO	Usado para enviar uma mensagem de informação ao cliente.
LOGINACK	Usado para enviar ao cliente a resposta a um pedido de login. A ausência deste token numa resposta de login significa que o login no servidor não foi realizado com sucesso.
NBCROW	Usado para enviar ao cliente uma linha definida pelo token COLMETADATA com compressão <i>null bitmap</i> (mais informações em [29]).
OFFSET	Usado para informar o cliente da posição onde uma palavra-chave ocorre num SQL text buffer do próprio cliente. Este token foi removido no TDS 7.2.
ORDER	Usado para informar o cliente que colunas determinam a ordem dos dados.
RETURNSTATUS	Usado para enviar ao cliente o valor do estado de um RPC.
RETURNVALUE	Usado para enviar ao cliente o valor de retorno de um RPC.
ROW	Usado para enviar ao cliente uma linha completa, que foi anteriormente definida por um token COLMETADATA.
SSPI	Token SSPI devolvido durante o processo de login.
TABNAME	Usado para enviar ao cliente o nome da tabela quando <code>sp_cursoropen</code> é utilizado ou quando em <i>browser mode</i> .
TVP ROW	Usado para enviar uma linha <i>table value parameter</i> (TVP), do cliente para o servidor.

Apêndice C

Cursor Stored Procedures

Existem instalados no SQL Server um conjunto de stored procedures que permitem a operação de um cursor sobre um *dataset* [17, 66]. Este é um tema que não está directamente relacionado com a descrição do protocolo TDS, mas do ponto de vista da implementação de um driver JDBC, estes dois temas estão intimamente ligados.

A mensagem de *rpc request* do TDS possui um campo em que se pode indicar um número de um stored procedure, esse stored procedure é um dos que se podem encontrar em [17] e cujo número identificador se pode encontrar na respectiva documentação.

Estes *stored procedures* são o elemento fundamental na implementação de um result set *scrollable* e/ou *updatable*.

Na Tabela C.1 são apresentados os stored procedures importantes para a implementação do driver. A coluna **procId** corresponde ao identificador do stored procedure.

Tabela C.1: Stored procdecures *do sistema relevantes para a implementação do driver JDBC*.

procId	Nome	Descrição
1	sp_cursor	Permite efectuar actualização, inserção ou remoção de uma ou mais linhas do <i>fetch buffer</i> do cursor.
2	sp_cursoropen	Abre um cursor definindo a <i>statement SQL</i> a ele associada e suas opções.
7	sp_cursorfetch	Carrega uma ou mais linhas para o <i>buffer</i> do cursor. Este <i>buffer</i> designa-se de <i>fetch buffer</i> .
9	sp_cursorclose	Fecha e liberta os recursos associados ao cursor.

A seguir será apresentada e explicada a sintaxe de cada um dos *stored procedures* aqui enunciados.

C.1 sp_cursor

Mais informações sobre este *stored procedure* podem ser encontradas em [35].

C.1.1 Sintaxe

```
sp_cursor cursor, optype, rownum, table  
[ , value [...n] ] ]
```

C.1.2 Argumentos

cursor

Identificador do cursor gerado pelo SQL Server na execução do `sp_cursor`.

optype

Identifica a operação a executar:

Valor	Operação	Descrição
0x0001	UPDATE	Actualiza uma ou mais linhas indicadas por <i>rownum</i> .
0x0002	DELETE	Remove uma ou mais linhas indicadas por <i>rownum</i> .
0x0004	INSERT	Insere dados.
0x0008	REFRESH	Volta a preencher o <i>fetch buffer</i> com os dados das tabelas.
0x10	LOCK	Provoca a aquisição de um SQL Server U-Lock nas páginas que contêm a linhas especificadas.
0x20	SETPOSITION	Pode ser usado numa cláusula OR com REFRESH, UPDATE, DELETE ou LOCK para mudar a posição do cursor para a última linha modificada.
0x40	ABSOLUTE	Pode ser usado numa cláusula OR com UPDATE ou DELETE para modificar a linha indicada por <i>rownum</i> , e cujo valor é referente ao <i>data set</i> criado pela <i>SQL statement</i> em vez de se referir ao <i>fetch buffer</i> .

rownum

Especifica a linha do *fetch buffer* sobre a qual se irá realizar a operação.

table Nome da tabela sobre a qual será realizada a operação. Relevante quando a *statement SQL* envolve um *join*.

value String em Unicode que indica os valores de actualização/inserção.

C.2 sp_cursoropen

Mais informações sobre este *stored procedure* podem ser encontradas em [38].

C.2.1 Sintaxe

```
sp_cursoropen cursor OUTPUT, stmt  
    [, scrollopt [ OUTPUT ] [ , ccopt [ OUTPUT ]  
    [ ,rowcount OUTPUT [ ,boundparam] [,...n] ] ] ] ]
```

C.2.2 Argumentos

cursor

Identificador do cursor gerado pelo SQL Server na execução do `sp_cursor`.

stmt

SQL statement que define o result set do cursor.

scrollopt

Indica o tipo de cursor criado:

Valor	Descrição
0x0001	KEYSET
0x0002	DYNAMIC
0x0004	FORWARD_ONLY
0x0008	STATIC
0x10	FAST_FORWARD
0x1000	PARAMETERIZED_STMT
0x2000	AUTO_FETCH
0x4000	AUTO_CLOSE
0x8000	CHECK_ACCEPTED_TYPES
0x10000	KEYSET_ACCEPTABLE
0x20000	DYNAMIC_ACCEPTABLE
0x40000	FORWARD_ONLY_ACCEPTABLE
0x80000	STATIC_ACCEPTABLE
0x100000	FAST_FORWARD_ACCEPTABLE

ccopt

Indica o tipo de concorrência do cursor criado:

Valor	Descrição
0x0001	READ_ONLY
0x0002	SCROLL_LOCKS
0x0004	OPTIMISTIC
0x0008	OPTIMISTIC
0x2000	ALLOW_DIRECT
0x4000	UPDT_IN_PLACE
0x8000	CHECK_ACCEPTED_OPTS
0x10000	READ_ONLY_ACCEPTABLE
0x20000	SCROLL_LOCKS_ACCEPTABLE
0x40000	OPTIMISTIC_ACCEPTABLE
0x80000	OPTIMISITC_ACCEPTABLE

rowcount Número de linhas do *fetch buffer* a ser usado pelo AUTO_FETCH.

boundparam Significa o uso de parâmetros adicionais.

C.3 sp_cursorfetch

Mais informações sobre este *stored procedure* podem ser encontradas em [37].

C.3.1 Sintaxe

```
sp_cursorfetch cursor
    [ , fetchtype [ , rownum [ , nrows ] ] ]
```

C.3.2 Argumentos

cursor

Identificador do cursor gerado pelo SQL Server na execução do **sp_cursor**.

fetchtype

Especifica que o *buffer* que deve ser carregado:

Valor	Nome	Descrição
0x0001	FIRST	Carrega o primeiro <i>buffer</i> de <i>nrows</i> linhas.
0x0002	NEXT	Carrega o próximo <i>buffer</i> de <i>nrows</i> linhas.
0x0004	PREV	Carrega o antecessor <i>buffer</i> de <i>nrows</i> linhas.
0x0008	LAST	Carrega o último <i>buffer</i> de <i>nrows</i> linhas.
0x10	ABSOLUTE	Carrega <i>nrows</i> linhas a partir da linha <i>rownum</i> .
0x20	RELATIVE	Carrega <i>nrows</i> linhas começando na linha <i>rownum</i> .
0x80	REFRESH	Recarrega o <i>buffer</i> com os dados das tabelas.
0x100	INFO	Obtém informação acerca do cursor.
0x200	PREV_NOADJUST	Usado como PREV, mas ao contrário de PREV, esta opção não preenche o <i>buffer</i> com linhas que se encontram na actual posição ou depois da actual posição do cursor.
0x400	SKIP_UPDT_CNCY	Quando usado os valores <i>timestamp</i> das colunas não são escritos na tabela <i>keyset</i> quando uma linha é (re)carregada. Tem que ser utilizado em conjunção com umas das outras opções à excepção de INFO.

rownum

Usado para especificar a linha de ABSOLUTE ou RELATIVE.

nrow

Especifica o número de linhas que devem ser carregadas pela operação. Por pré-definição tem o valor 20.

C.4 sp_cursorclose

Mais informações sobre este *stored procedure* podem ser encontradas em [36].

C.4.1 Sintaxe

```
sp_cursorclose cursor
```

C.4.2 Argumentos

cursor

Identificador do cursor gerado pelo SQL Server na execução do `sp_cursoropen`.

Apêndice D

Funcionalidade implementada

Este anexo apresenta uma lista da funcionalidade da API JDBC implementada pelo driver. Fica assim uma referência para a utilização do driver.

D.1 Implementação JDBC

As Tabelas D.1, D.2 e D.3 apresentam as interfaces do JDBC implementadas.

Tabela D.1: *Métodos implementados da interface Driver*

	Nome
boolean	acceptsURL(String url)
Connection	connect(String url, Properties info)
int	getMajorVersion()
int	getMinorVersion()
boolean	jdbcCompliant()

Tabela D.2: *Métodos implementados da interface Statement*

	Nome
void	addBatch(String sql)
void	clearBatch()
void	close()
int []	executeBatch()
ResultSet	executeQuery(String sql)
int	executeUpdate(String sql)

Tabela D.3: Métodos implementados da interface ResultSet

	Nome
boolean	absolute(int row)
void	afterLast()
void	beforeFirst()
void	cancelRowUpdates()
void	close()
void	deleteRow()
boolean	first()
Date	getDate(int columnIndex)
double	getDouble(int columnIndex)
int	getFetchSize()
int	getInt(int columnIndex)
int	getRow()
String	getString(int columnIndex)
void	insertRow()
boolean	isAfterLast()
boolean	isBeforeFirst()
boolean	isClosed()
boolean	isFirst()
boolean	isLast()
boolean	last()
void	moveToCurrentRow()
void	moveToInsertRow()
boolean	next()
boolean	previous()
void	refreshRow()
boolean	relative(int rows)
void	setFetchSize(int rows)
void	updateDate(int columnIndex, Date x)
void	updateDouble(int columnIndex, double x)
void	updateInt(int columnIndex, int x)
void	updateRow()
void	updateString(int columnIndex, String x)
boolean	wasNull()

D.2 Implementação TDS

O TDS especifica o formato como o SQL Server recebe e envia um valor, por isso o driver tem de conhecer esse formato para conseguir comunicar com sucesso com o servidor. Como existem vários tipos suportados pelo SQL Server [21] e o driver implementado apenas suporta alguns, a Tabela D.4 apresenta a referência dos tipos suportados. Como o facto de uma coluna permitir ou não valores nulos pode alterar o formato, a segunda coluna da Tabela D.4 indica o suporte relativo a esta característica.

Tabela D.4: *Tipos SQL suportados pelo driver.*

Nome	Null/Not Null
INT	S/S
FLOAT	S/S
DATETIME	S/S
NVARCHAR	S/S