FACULDADE · DE · CIÊNCIAS UNIVERSIDADE · DE · LISBOA
DEPARTAMENTO DE INFORMÁTICA
Bloco C6 - Piso 3 - Campo Grande, 1749-016 Lisboa
Tel & Fax: +351 217500084

RELATÓRIO DE PROJECTO

sobre

Interfaces de suporte à replicação de bases de dados

realizado no

Laboratório de Sistemas Informáticos de Grande Escala (LaSIGE)
Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

por

Susana Rodrigues Guedes

2 de Dezembro de 2006

Resumo

No contexto dos sistemas de informação, os sistemas de gestão de bases de dados assumem-se como uma ferramenta fundamental para gestão, selecção e tratamento dos dados. As técnicas que suportam uma eficiente replicação das bases de dados são extremamente importantes uma vez que aumentam a disponibilidade dos dados na presença de faltas, permitindo explorar a localidade dos mesmos e dispersar a carga entre as várias réplicas.

Este trabalho contribui para a definição de uma nova interface para suporte à replicação, denominada Interface Gorda de Reflexão, ou simplesmente Reflector. O objectivo primário consiste em desacoplar o núcleo da base de dados da lógica da replicação, de forma a que várias estratégias de replicação possam ser implementadas no topo de qualquer base de dados compatível.

De forma a validar o Reflector, este trabalho tem como objectivo concretizar uma instância do mesmo num SGBD relacional de código aberto, denominado Apache Derby. O objectivo é demonstrar que as opções de desenho do Reflector, a nível de arquitectura e funcionalidades, foram adequadas e permitem uma extensão pouco intrusiva do núcleo do SGBD.

Finalmente, pretende-se avaliar o impacto que o Reflector poderá introduzir no desempenho de um SGBD relacional que o suporte de forma nativa. A avaliação consiste em comparar o desempenho do SGBD Derby original e com a extensão de suporte ao Reflector, enquanto processam cargas semelhantes num mesmo cenário de execução.

Agradecimentos

Ao Professor Luís Rodrigues, meu orientador. Por sempre ter acreditado no meu potencial. A sua exigência, o seu rigor, a sua crítica, e o seu amor pela profissão sempre se mostraram como um exemplo a seguir, sempre puxaram as minhas capacidades ao limite, e moldaram a minha forma de ver e fazer investigação.

Aos colegas do Projecto Gorda. Agradeço, particularmente, ao Alfrânio Correia Júnior por sempre ter estado "do outro lado do messenger", ao Nuno Carvalho pela ajuda que prontamente me ofereceu, e ao José Orlando Pereira pelos conhecimentos e visões que me transmitiu.

Ao Professor António Ferreira. Pela preciosa colaboração e análise crítica. O seu rigor técnico, e a sua procura da perfeição mostraram-me uma filosofia de trabalho que procurei e pretendo seguir.

Ao Grupo de Investigação DIALNP, ao LASIGE, e ao Departamento de Informática da FCUL. Pelas condições que me proporcionaram para o desenvolvimento deste trabalho. Deixo ainda um agradecimento especial aos colegas do DIALNP, um grupo que considero enriquecido pela diversidade e qualidade dos seus elementos.

Aos meus pais, família, e amigos. Aos meus pais, por me terem posto sempre em primeiro lugar, e por me acompanharem a cada dia. À minha família, pelo orgulho que sempre demonstraram ter em mim. Aos meus amigos, por terem sido exactamente isso. E ao João, por me ter dado a força que nem sempre foi fácil ter para seguir em frente, e por estar ao meu lado sempre.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

No contexto dos sistemas de informação, os sistemas de gestão de bases de dados (SGBD) assumem-se como uma ferramenta fundamental para gestão, selecção e tratamento dos dados em tempo útil. As técnicas que suportam uma eficiente replicação das bases de dados são extremamente importantes uma vez que aumentam a disponibilidade dos dados na presença de faltas, permitindo explorar a localidade dos mesmos e dispersar a carga entre as várias réplicas. A vasta gama de aplicações de replicação de bases de dados, com requisitos diversos, levou ao desenvolvimento de múltiplos protocolos para gestão da replicação.

No entanto, a falta de suporte nativo dos SGBD para replicação por terceiros (*third-party replication*) requer que esses sistemas de replicação modifiquem o núcleo do sistema ou desenvolvam um invólucro middleware para o mesmo de forma a poderem realizar a replicação. As soluções que operam ao nível do núcleo do SGBD são as que oferecem um melhor desempenho, mas não são compatíveis entre si nem transportáveis para outros tipos de bases de dados. A falta de modularidade que caracteriza a grande maioria das soluções nativas dificulta a sua manutenção e adaptação. Por sua vez, as soluções por middleware oferecem uma maior modularidade, permitindo que o sistema replicado opere sobre sistemas de gestão de bases de dados heterogéneos. No entanto, por actuarem fora do núcleo do SGBD, estas aproximações têm muitas vezes de repetir passos realizados pela base de dados, o que geralmente se traduz numa quebra acentuada do desempenho. Existem ainda outras variantes de replicação, com características semelhantes à da aproximação por middleware, que se baseiam na utilização das interfaces cliente fornecidas pela base de dados.

Este trabalho pretende desenvolver uma solução que combine a modularidade e a portabilidade da aproximação por middleware com o desempenho da aproximação nativa, de forma a que várias estratégias de replicação possam ser concretizadas sobre qualquer base de dados compatível.

1.1 Definição do problema e objectivos

Este trabalho tem como objectivo inicial contribuir para a definição e validação de uma nova Interface de Programação (API[1]) de suporte à replicação, denominada *Interface Gorda de Reflexão*, ou simplesmente *Reflector*. O Reflector deverá ser exportado de forma nativa por SGBD relacionais e permitirá desacoplar o núcleo da base de dados, da lógica da replicação, independentemente da forma como estes componentes estarão interligados no produto final. Com esta solução poderemos beneficiar do desempenho de um acesso aos dados suportado de forma nativa pelo SGBD ao mesmo tempo que obtemos uma relação modular e portável entre o núcleo da replicação e o núcleo da base de dados.

De forma a validar o Reflector, este trabalho tem como objectivo principal concretizar uma instância do mesmo num SGBD relacional de código aberto, denominado Apache Derby. Pretende-se que a concretização permita demonstrar as potencialidades da interface definida, segundo vários cenários de replicação. De forma a facilitar a sua disseminação e aceitação pela comunidade Derby, esta extensão ao SGBD deverá ser tão pouco intrusiva quanto possível, mantendo uma compatibilidade completa com todas as suas ferramentas e interfaces clientes.

Por fim, pretende-se ainda avaliar o impacto que o Reflector poderá introduzir no desempenho de um SGBD relacional que o suporte nativamente. A avaliação consiste em comparar o desempenho do SGBD Derby original e com a extensão de suporte ao Reflector, enquanto processam cargas semelhantes num mesmo cenário de execução. Para isso, será utilizada a plataforma padrão TPC-W, que permite simular cargas de um ambiente de publicações e vendas na WEB.

1.2 Âmbito do trabalho

Este trabalho foi desenvolvido no contexto de um projecto de investigação europeu, GORDA (Open Replication of DAtabases), que procura fomentar a replicação de bases de dados como um meio de responder aos desafios de confiabilidade, desempenho, e custo dos actuais sistemas de bases de dados. O projecto é composto por um misto de parceiros académicos e industriais, incluindo U. de Lisboa, U. do Minho, U. della Svizzera Italiana, INRIA Rhône-Alpes, Emic Networks OY, e MYSQL AB. Mais pormenores sobre o projecto poderão ser consultados em [10].

O resultado deste trabalho está enquadrado no âmbito da disciplina de Projecto de Engenharia Informática do Curso de Especialização Profissional em Engenharia Informática[2], realizada no grupo de investigação DIALNP[3] do Laboratório de Sistemas Informáticos de Grande Escala (LASIGE) da Universidade de Lisboa.

[1] Application Programming Interface.
[2] CEPEI - http://cepei.di.fc.ul.pt
[3] Distributed Algorithms and Network Protocols - http://dialnp.lasige.di.fc.ul.pt

1.3 Organização do documento

O resto do documento encontra-se estruturado da seguinte forma:

Capítulo 2 - "Panorâmica sobre os Sistemas de Gestão de Bases de Dados Relacionais": Este capítulo começa por introduzir, os conceitos de sistema de gestão de bases de dados e de modelos de dados. Uma vez que este trabalho está particularmente focado nos SGBD relacionais, a Secção 2.1 aprofunda o modelo relacional apresentando ainda duas das suas linguagens formais interrogativas, a álgebra e o cálculo relacional. A linguagem SQL e suas interfaces cliente são definidas na Secção 2.2. A Secção 2.3 apresenta a arquitectura típica de um SGBD relacional. As secções seguintes focam alguns dos módulos identificados como parte da arquitectura. A Secção 2.4 apresenta um típico motor de processamento relacional com ênfase nas fases de processamento mais importantes. A Secção 2.5 apresenta o módulo de controlo de concorrência, detalhando alguns dos mecanismos utilizados para esse efeito. A Secção 2.6 apresenta o módulo de recuperação após falhas, focando os mecanismos de salvaguardas para histórico. Por fim, a Secção 2.7 resume o capítulo focando os pontos mais relevantes.

Capítulo 3 - "Panorâmica da Replicação de Bases de Dados": Este capítulo classifica e apresenta algumas das soluções existentes para replicação de bases de dados. A classificação consiste em três critérios base: a arquitectura, Secção 3.1; os modelos de coerência, Secção 3.2; e o nível ao qual os protocolos são implementados, Secção 3.3. Por fim, são ainda apresentados alguns exemplos concretos de protocolos de replicação, como tema da Secção 3.4. Por fim, a Secção 3.5 resume o capítulo focando os pontos mais relevantes.

Capítulo 4 - "Interface Gorda de Reflexão": Este capítulo apresenta a Interface Gorda de Reflexão, ou Reflector, para suporte à replicação. A Secção 4.1 realiza uma análise dos requisitos que os protocolos de replicação impõem numa interface como o Reflector. A Secção 4.2 apresenta genericamente as mais relevantes opções de desenho do Reflector, e em particular a sua arquitectura. A Secção 4.3 apresenta as fases de processamento transaccional a reflectir. A Secção 4.4 apresenta os contextos de execução que podem ser associados a cada uma das fases de processamento definidas. A Secção 4.5 agrupa e detalha os componentes do Reflector, com base nos seus diagramas de classes UML. A Secção 4.6 ilustra as potencialidades do Reflector, através de casos de estudo da sua aplicabilidade a diversos protocolos de replicação. Por fim, a Secção 4.7 resume o capítulo focando os pontos mais relevantes.

Capítulo 5 - "Concretização do Reflector": Este capítulo apresenta a concretização do Reflector, de forma abstracta num SGBDR genérico, e de forma concreta no SGBDR Apache Derby. A Secção 5.1 descreve os desafios da concretização do Reflector num SGBDR genérico. A Secção 5.2 apresenta o SGBDR Apache Derby. A Secção 5.3

detalha as soluções encontradas para a concretização de cada um dos desafios enumerados no Derby. Por fim, a Secção 5.4 resume o capítulo focando os pontos mais relevantes.

Capítulo 6 - "Avaliação": Este capítulo avalia o impacto que o Reflector introduz num SGBDR que o suporte nativamente. A Secção 6.1 apresenta a plataforma TPC-W. A Secção 6.2 descreve o ambiente de testes. A Secção 6.3 descreve os resultados obtidos. Por fim, a Secção 6.4 resume o capítulo focando os pontos mais relevantes.

Capítulo 7 - "Conclusões e Trabalho Futuro": Este capítulo conclui o trabalho e apresenta alguns dos objectivos delineados como trabalho futuro.

Capítulo 2

Panorâmica sobre os Sistemas de Gestão de Bases de Dados Relacionais

Um sistema de gestão de bases de dados (SGBD) é um software desenhado para assistir na manutenção e utilização de grandes conjuntos de dados [35]. Os SGBD apresentam vantagens face a mecanismos de armazenamento persistente mais simples, como colecções de ficheiros do sistema operativo, dado que fornecem uma variedade de técnicas para armazenar e recuperar dados locais ou remotos de forma transparente e eficiente. Estes sistemas dão a cada utilizador a ilusão de ser o único a aceder aos dados em cada momento, o que é conseguido através de técnicas de controlo de concorrência e de técnicas que mascaram as falhas provocadas por outros utilizadores do SGBD. Adicionalmente, a integridade e segurança dos dados passam a ser garantidas através de políticas de controlo de acesso. Delegar a satisfação de todos estes requisitos num SGBD traduz-se num decréscimo da complexidade das aplicações.

A colecção de conceitos que permite descrever os dados a armazenar no SGBD é definida pelo modelo de dados correspondente. Tipicamente existem dois níveis de abstracção em modelos de dados: os modelos de dados semânticos, e os modelos de dados conceptuais (ou lógicos) efectivamente suportados pelo SGBD. Um modelo de dados semântico é um modelo de alto-nível que ajuda o utilizador a desenhar uma descrição inicial e abstracta dos dados. Os modelos de classes UML[17, 33, 27] e os modelos Entidade-Associação (EA) [35] são dos mais adoptados para a definição semântica dos dados. O modelo de dados conceptual, suportado pelo SGBD, é uma colecção de conceitos de alto-nível que permite descrever os dados escondendo os pormenores do armazenamento de baixo-nível. Existe uma variedade de modelos conceptuais entre os quais: modelo relacional, modelo orientado a objectos, modelo de objectos relacional, ou modelo semi-estruturado, entre outros. Este trabalho foca-se nos SGBD baseados no modelo conceptual relacional, denominados simplesmente por SGBDR), por ser o modelo considerado no âmbito do projecto Gorda a que este trabalho se refere.

Na Secção 2.1 será detalhado o modelo relacional com referência a duas das suas linguagens formais para manipulação de dados relacionais: a álgebra e o cálculo relacionais. Na Secção 2.2 é apresentada a linguagem mais amplamente utilizada para interacção com um SGBDR, denominada Linguagem SQL. Na Secção 2.3 é descrita a arquitectura genérica de um sistema SGBDR. A Secção 2.4 descreve o modo de funcionamento genérico de um motor de processamento relacional. A Secção 2.5 apresenta o módulo de controlo de concorrência, detalhando alguns dos mecanismos existentes para controlo de concorrência. Por fim, a Secção 2.6 apresenta o módulo de recuperação após falhas, focando os mecanismos de salvaguardas para histórico.

2.1 O modelo relacional

O modelo relacional é um modelo conceptual de definição de dados que pode ser desenhado directamente pelo utilizador, ou pode ser obtido por transformação de modelos de dados semânticos, como o modelo UML ou EA[17, 33, 27, 35], segundo um conjunto de regras bem conhecidas.

O conceito central do modelo relacional é a *relação*. Uma relação é composta por um *esquema de relação* e uma *instância de relação*. O esquema de uma relação corresponde aos meta-dados de uma instância de relação. O esquema descreve o *nome* da relação, o nome de cada *atributo* (ou campo) e respectivo *domínio*. Um domínio é referenciado pelo seu nome e tem um conjunto de valores associados resultantes das *restrições de domínio* que lhe estão associadas. Por exemplo, no caso de uma base de dados de uma escola, o esquema relacional correspondente a uma disciplina pode ser:

```
Disciplina(código: string, nome: string, ano lectivo: data, vagas: inteiro)
```

Neste exemplo, *disciplina* é o nome do esquema de relação que será composto por quatro atributos: *código*, *nome*, *ano lectivo*, e *vagas*. Os atributos *código* e *nome* têm associado o domínio *string*, o que indica que os seus valores deverão ser descritos em formato de texto. O atributo *ano lectivo* apresenta o domínio *vagas*, o que restringe o seu domínio de valores a um dos possíveis formatos para uma data. Por fim, o atributo *vagas* tem o domínio *inteiro*, pelo que os seus valores deverão ser numéricos e inteiros.

Uma instância de relação é um conjunto de *tuplos* (ou registos) que pertencem a uma relação e contém todos os atributos definidos no respectivo esquema. Cada instância pode ser vista como uma *tabela* onde cada atributo representa uma *coluna* e cada tuplo constitui uma *linha*. O modelo relacional especifica ainda a noção de *vista*. Uma vista é uma relação cuja instância não está explicitamente armazenada na base de dados. Para simplificar, uma vista pode ser interpretada como sendo uma tabela temporária.

De forma a prevenir que informação incorrecta seja introduzida no sistema, o modelo relacional define *restrições de integridade* que permitem validar os tuplos de cada relação. Para além das restrições de domínio existem também:

Restrições de chave: garantem que um conjunto mínimo de atributos de uma relação identifica univocamente cada tuplo. Por exemplo, no caso do esquema de relação de uma disciplina referido anteriormente, o código poderia corresponder à restrição de chave, pois não existem duas disciplinas diferentes com um mesmo código.

Restrições de chave estrangeira: garantem que quando duas ou mais relações partilham campos, a alteração de um desses campos numa das relações é propagada e validada nas restantes. Estas restrições definem ainda o modo como as alterações são propagadas. Por exemplo, no caso do tuplo ser apagado numa relação pode configurar-se o sistema para apagar em cascata todos os tuplos que anteriormente o referenciavam.

Restrições de tabela: estão associadas a uma única tabela (ou relação) e são verificadas de cada vez que esta sofre alterações. Têm como objectivo garantir que uma determinada condição é sempre verdadeira. Por exemplo, poderá ser definida uma restrição para obrigar que um determinado campo da tabela seja de preenchimento obrigatório.

Restrições assertivas: estão associadas a uma ou mais tabelas (ou relações) e são verificadas de cada vez que uma delas sofre alterações. Têm como objectivo garantir que uma determinada condição é sempre verdadeira. Por exemplo, poderá ser definida uma restrição para obrigar que o valor de um atributo numa tabela seja sempre inferior a um valor estipulado noutra tabela.

As secções seguintes aprofundam alguns aspectos do modelo relacional. Em particular, na Secção 2.1.1, serão apresentados os níveis de abstracção fornecidos por um SGBDR e será discutida a forma como estes permitem esconder os pormenores de armazenamento e representação dos dados. Na Secção 2.1.2 será definido o processo de normalização e a sua importância na afinação de um modelo de dados relacional. Por fim, na Secção 2.1.3, serão abordadas duas linguagens formais para interacção com o modelo relacional: a álgebra e o cálculo relacionais.

2.1.1 Níveis de abstracção e independência dos dados

Ao nível da modelação relacional um SGBDR fornece uma visão dos dados em três níveis de abstracção: *esquema conceptual* (ou esquema lógico), *esquema externo* e *esquema físico*. A informação sobre os esquemas é armazenada em *catálogos do sistema* sob a forma de meta-dados.

O esquema conceptual de um SGBDR descreve todas as relações armazenadas na base de dados. O processo seguido pelo utilizador para definir este esquema designa-se *desenho conceptual da base de dados*.

O esquema físico descreve como as relações descritas no esquema conceptual são armazenadas. É necessário definir a organização dos ficheiros de dados e criar índices. Um

índice é uma estrutura que permite tornar o acesso aos dados mais eficiente. As decisões tomadas a este nível são definidas durante o *desenho físico da base de dados* e baseiam-se no conhecimento de como os dados serão tipicamente acedidos.

Por fim, o esquema externo permite o acesso aos dados ao nível do utilizador ou grupo de utilizadores. Também este esquema é definido através do modelo de dados do SGBDR. É a este nível que se definem as políticas de controlo de acesso. Cada esquema externo consiste na colecção de uma ou mais vistas do esquema conceptual moldadas para um determinado utilizador ou grupo de utilizadores.

As aplicações devem ser tão independentes quanto possível dos pormenores de representação e armazenamento dos dados. Uma importante vantagem na utilização de SGBDRé a independência dos dados conseguida através dos três níveis de abstracção acima descritos, significando que podem ser feitas alterações num esquema a um determinado nível sem ter de alterar os níveis superiores.

As relações no esquema externo (vistas) são geradas a pedido pelas relações correspondentes ao esquema conceptual. Se os dados forem alterados ou reorganizados, isto é, se o esquema conceptual for alterado, a definição de uma relação no esquema externo pode ser modificada de forma a que a vista gerada permaneça inalterada. Esta propriedade designa-se *independência lógica dos dados* e isola os utilizadores das alterações e reestruturações realizadas no nível lógico.

Por sua vez, o esquema conceptual isola os utilizadores das alterações realizadas no nível físico de armazenamento de dados. Esta propriedade de *independência física dos dados* é garantida pelo esquema conceptual que esconde pormenores como a organização dos dados em disco, a estrutura dos ficheiros e a escolha dos índices.

2.1.2 Normalização

O desenho conceptual da base de dados fornece um conjunto de esquemas relacionais e restrições de integridade que descrevem a informação de uma forma intuitiva e completa. No entanto, em alguns casos estes modelos apresentam um elevado nível de *redundância*, isto é, de armazenamento repetido da mesma informação. Um esquema não normalizado pode originar várias anomalias: anomalias de actualização, anomalias de inserção, e anomalias de remoção. As anomalias de actualização ocorrem quando apenas uma parte das cópias dos dados redundantes é actualizada, originando incoerências nos dados armazenados. As anomalias de inserção levam a que, por vezes, não seja possível armazenar um conjunto de dados sem que outro, não relacionado, seja também armazenado. As anomalias de remoção levam a que não seja possível remover um conjunto de dados sem que outro, não relacionado, seja também removido. A primeira anomalia resulta do armazenamento redundante de informação. As restantes duas anomalias resultam de uma definição instável do esquema relacional, na qual estão definidas incorrectamente algumas das dependências e relações entre os dados.

A normalização tem como objectivo avaliar a qualidade de um esquema relacional e, caso necessário, proceder à sua afinação num esquema relacional equivalente, menos redundante e mais estável. Para reduzir a redundância é usado o processo de *decomposição* que define regras para substituir um esquema de relação complexo por vários esquemas de relação mais simples. A *forma normal* é uma propriedade do esquema relacional que indica o tipo de redundância que este poderá exibir.

Por se considerar que este tópico não se enquadra no âmbito deste trabalho, remete-se o leitor interessado nos pormenores do processo de decomposição e normalização, para a bibliografia sobre o assunto [35].

2.1.3 Linguagens formais de interacção com o modelo relacional

Esta secção apresenta em termos gerais duas linguagens interrogativas formais do modelo relacional: a álgebra e o cálculo relacionais. As *linguagens interrogativas* são especializadas na declaração de pesquisas ou interrogações aos dados de uma base de dados. As interrogações partem de instâncias de relação iniciais e geram outras instâncias de relação como resultado. Na *álgebra relacional* as interrogações descrevem a sequência de operadores de álgebra relacional para calcular a relação final a partir das relações iniciais. No *cálculo relacional* a interrogação descreve as características dos tuplos que devem aparecer na relação final.

Algebra relacional

Em álgebra relacional as interrogações são compostas por uma sequência de operadores relacionais, entre os quais:

Selecção: A partir de uma relação, aplica uma condição booleana a todos os tuplos para obter apenas alguns, não considerando os restantes.

Projecção: Extrai algumas das colunas da relação inicial.

Etiquetagem: Permite alterar os nomes de uma relação e suas colunas sem alterar os dados subjacentes.

União: A união de duas relações necessariamente compatíveis (com as mesmas colunas) é a relação composta por todos os tuplos de ambas.

Intersecção: A intersecção de duas relações compatíveis é a relação composta pelas ocorrências pertencentes a ambas.

Diferença: A diferença de duas relações compatíveis é a relação composta pelas ocorrências da primeira que não pertencem à segunda.

Divisão: A partir de duas relações, devolve uma relação com os campos da primeira, que não pertencem à segunda. A relação é composta pelas ocorrências da primeira que

se combinem com todas as ocorrências da segunda, isto é, que tenham pelo menos uma ocorrência que contenha cada tuplo da segunda.

Produto Cartesiano: A partir de duas relações, devolve uma relação com os campos de ambas (mesmo que repetidos). A relação é composta pela combinação de todos os tuplos da primeira com todos os tuplos da segunda.

Junção: Restringe o resultado do produto cartesiano impondo uma condição de selecção. A junção pode ser sub-catalogada mediante o tipo de selecção a aplicar. Por exemplo, quando a condição de selecção envolve apenas igualdades então designa-se equi-junção.

A álgebra relacional é frequentemente utilizada como uma métrica do poder expressivo de linguagens interrogativas relacionais. Espera-se que a linguagem interrogativa fornecida por um SGBDR relacional seja *relacionalmente completa*, no sentido de que deverá ser capaz de exprimir todas as interrogações definidas pela álgebra relacional. Algumas destas linguagens fornecem ainda funcionalidades que permitem exprimir interrogações não suportadas pela álgebra linear.

Cálculo relacional

O *cálculo relacional* é uma alternativa à álgebra relacional. Enquanto a álgebra é uma linguagem procedimental composta por operadores relacionais, o cálculo é uma linguagem declarativa, no sentido que permite definir a resposta pretendida sem especificar a forma como esta será alcançada. A variante do cálculo relacional, apresentada nesta secção, é o *cálculo relacional sobre tuplos*.

O cálculo relacional sobre tuplos toma tuplos como valores nas suas expressões. As interrogações tomam a forma ($T \mid p(T)$) onde T é uma variável tuplo, p(T) é uma fórmula que descreve T, e a resposta é o conjunto de tuplos que satisfaz p(T).

No caso da base de dados de uma escola, uma interrogação possível seria 'Quais as disciplinas com vagas superiores a 0?'. Em TRC podemos definir a interrogação da seguinte forma: ($T \mid T \in Disciplina \wedge T.vagas > 0$).

2.2 A linguagem SQL

A interacção do utilizador com o SGBDR é tipicamente feita com através da linguagem SQL (Structured Query Language). Embora inicialmente desenvolvida pela IBM, esta linguagem expandiu-se rapidamente num leque de linguagens derivadas, o que levou à necessidade de a normalizar. Esta tarefa foi realizada pela ANSI[1] em 1986, e pela ISO[2] em 1987. Ao longo dos anos a linguagem tem sido alvo frequente de revisões e extensões, sendo as mais recentes a de 1992, 1999 e 2003.

[1] American National Standards Institute [6]
[2] International Organization for Standardization [7]

SQL é uma linguagem normalizada para definição, acesso, manipulação e controlo de bases de dados relacionais. Esta linguagem pode ser considerada uma composição de três grupos base: TML (Transaction Manipulation Language) para manipulação de transacções, DDL (Data Definition Language) para definição de dados, e DML (Data Manipulation Language) para manipulação de dados.

Conforme anteriormente referido, o modelo relacional é composto por dois conceitos base: o esquema da relação e a instância de relação. De forma simplificada podemos associar a linguagem DDL à gestão dos esquemas de relações, enquanto a linguagem DML tem como objectivo a manipulação das instâncias de relações, também designadas em SQL por tabelas (termo que utilizarei nesse contexto). A linguagem TML permite ao utilizador gerir os pedidos SQL já realizados, nomeadamente, permite cancelar as alterações realizadas ou confirmá-las com a garantia que são duráveis.

2.2.1 Linguagem TML

A linguagem TML (Transaction Manipulation Language) é a linguagem utilizada para gerir transacções. Uma *transacção* é uma sequência de operações sobre a base de dados, perfeitamente delimitada, que exibe algumas características importantes, conhecidas pelo acrónimo ACID:

Atomicidade: Uma transacção, ou termina com sucesso, ou então todas as suas operações sobre a base de dados são desfeitas dando a ilusão de nunca terem existido.

Coerência: Desde que todas as transacções terminem deixando a base de dados num estado consistente, transacção nenhuma poderá ver o SGBD num estado inconsistente.

Isolamento: Numa situação em que várias transacções concorrentes acedem aos dados, o SGBD deve garantir que o resultado final é o mesmo que seria alcançado caso as transacções fossem executadas de forma sequencial.

Durabilidade: Após uma transacção terminar com sucesso, as suas actualizações sobre a base de dados passam a ser efectivas, sobrevivendo a falhas que eventualmente possam ocorrer no sistema computacional.

Uma transacção é, portanto, uma série de leituras e escritas de dados na base de dados aos quais denominados conjunto de escrita, respectivamente. Após terminar todas as operações, uma transacção termina com sucesso ou aborta. Se uma transacção é confirmada (com commit bem sucedido), todas as suas alterações na base de dados têm de ser duráveis. Se uma transacção é cancelada (com rollback ou devido a uma falha), todas as suas alterações na base de dados têm de ser anuladas.

A linguagem TML permite iniciar, confirmar ou anular uma transacção através dos comandos BEGIN, COMMIT, e ROLLBACK, respectivamente.

2.2.2 Linguagem DDL

A linguagem DDL (Data Definition Language) tem como objectivo a definição de esquemas de relações, permitindo a criação, remoção e alteração de estruturas de dados em qualquer um dos três níveis de abstracção suportados por um SGBDR: nível conceptual, nível externo e nível físico. Relativamente ao nível conceptual a linguagem fornece funcionalidades para criação, remoção e redefinição de tabelas através dos comandos CREATE TABLE, DROP TABLE, e ALTER TABLE. No nível externo, a linguagem suporta os comandos CREATE VIEW e DROP VIEW para a definição e remoção de vistas. Por fim, o nível físico permite criar, remover, e alterar a definição de índices, para tornar mais eficiente o acesso aos dados das tabelas, através dos comandos CREATE INDEX, DROP INDEX, e ALTER INDEX.

2.2.3 Linguagem DML

A linguagem DML (Data Manipulation Language) está associada à manipulação das tabelas de um SGBDR, permitindo inserir, actualizar e apagar dados através de comandos INSERT, UPDATE, e DELETE. A funcionalidade mais poderosa da DML é o comando SELECT que permite descrever interrogações sobre os dados com diferentes graus de complexidade. A definição deste comando teve influência nos conceitos de álgebra e cálculo relacional introduzidos na Secção 2.1.3. O comando SELECT básico tem o seguinte formato:

```
SELECT [DISTINCT] lista-de-colunas
FROM  lista-de-tabelas
[WHERE lista-de-condições]
```

A cláusula SELECT denota as colunas a reter como resultado, a cláusula FROM especifica o produto cruzado das tabelas e a cláusula opcional WHERE indica as condições de selecção a aplicar às tabelas mencionadas na cláusula FROM. Intuitivamente podemos associar o comando SELECT a uma sequência de produtos cruzados, projecções, e selecções, conforme definidos pela álgebra relacional. Este comando fornece ainda uma vasta gama de cláusulas opcionais que permitem especificar todos os operadores da álgebra relacional, significando que o comando SELECT é relacionalmente completo. No entanto, a explicação pormenorizada do poder expressivo da linguagem SQL vai para além do âmbito deste trabalho, remetendo-se o leitor interessado para a bibliografia [28].

2.2.4 Interfaces SQL para Clientes

Neste tópico apresento algumas das interfaces possíveis para um cliente interagir com um SGBDR usando a linguagem SQL:

Interface SQL *interactiva:* cada SGBDR fornece uma aplicação onde os utilizadores podem introduzir comandos SQL interactivamente. Esta interface é adequada quando a utilização de comandos SQL é suficiente. No entanto, todas as outras funcionalidades estão limitadas, nomeadamente ao nível do tratamento dos dados obtidos.

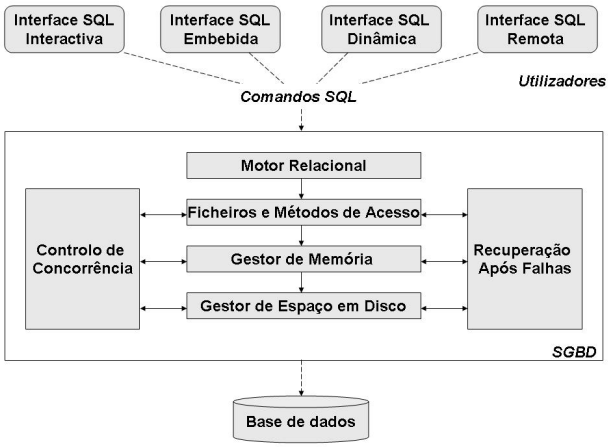


Figura 2.1: Arquitectura de um SGBD relacional [35]

SQL *embebido:* os comandos SQL podem ser embebidos e executados a partir de código numa linguagem de programação da máquina, como C ou Java. No entanto esta aproximação está limitada ao suporte que essas linguagens dão ao SQL, especialmente a nível dos tipos de dados. Em geral, estas linguagens de programação não apresentam qualquer tipo de dados que corresponda a uma colecção de tuplos como uma tabela. Este problema foi resolvido através da criação de cursores SQL que permitem obter um tuplo de cada vez.

SQL *dinâmico:* permite a interacção com o SGBDR a partir de código numa linguagem de programação da máquina, como C ou Java, sem que seja necessário ter os comandos SQL em tempo de compilação. Esta funcionalidade é possível através do uso de comandos específicos para preparar e processar pedidos SQL em tempo de execução.

Interfaces SQL *normalizadas:* à semelhança da funcionalidade de SQL embebido, as interfaces normalizadas permitem a integração da SQL com uma linguagem de programação conseguida através da introdução de um nível de indirecção entre a aplicação e o SGBDR. As interfaces ODBC (Open DataBase Connectivity) e JDBC (Java DataBase Connectivity) são dois exemplos de interfaces normalizadas. Essa camada extra fornece uma abstracção do SGBDR ao nível do executável. Isto significa que um mesmo executável pode aceder a diferentes SGBDR sem necessidade de ser alterado ou mesmo recompilado. Mais informação sobre as interfaces ODBC e JDBC pode ser consultada em [5, 4].

2.3 Arquitectura genérica de um SGBD relacional

A Figura 2.1 ilustra a arquitectura de alto nível típica de um sistema de SGBDR. Em termos gerais, um SGBDR aceita comandos na *linguagem SQL* provenientes do utilizador

sob uma variedade de interfaces possíveis, gera planos de execução dos comandos SQL, executa-os na base de dados, e retorna a resposta ao utilizador.

Quando um utilizador realiza uma interrogação, o comando é entregue ao *motor relacional* responsável por analisar a exequibilidade do comando e produzir um *plano de execução* eficiente. O plano de execução consiste numa árvore de operadores relacionais anotada com os métodos de acesso aos dados a usar e com as técnicas para passar os resultados intermédios entre os vários operadores relacionais consecutivos. Na Secção 2.4 abordam-se com mais pormenor o processamento de cada interrogação ao nível do motor relacional.

A camada do motor relacional assenta no topo da camada de *métodos de acesso*. Esta última inclui uma variedade de funcionalidades para suportar o conceito de *ficheiro* que, num SGBDR, consiste numa colecção de páginas ou registos.

Por sua vez, a camada de ficheiros e métodos de acesso assenta no topo da camada de *gestão de memória*, que é responsável por trazer as páginas do disco para memória conforme necessário em resposta às operações de leitura de dados.

A camada de *gestão de espaço em disco* fornece rotinas para reservar e libertar espaço em disco e ainda para ler e escrever páginas de dados para disco que serão usadas pelas camadas superiores.

Um SGBDR fornece ainda mecanismos para *controlo de concorrência* e *recuperação após falhas* através do escalonamento dos pedidos e da manutenção de um registo de todas as alterações à base de dados. As camadas de ficheiros e métodos de acesso, gestão de memória, e gestão de espaço em disco interagem com estes módulos. Nas secções 2.5 e 2.6 abordam-se com mais pormenor estes conceitos.

2.4 O motor relacional e o processamento de pedidos SQL

Esta secção apresenta genericamente o processo de análise e execução de pedidos SQL realizado pelo motor relacional de um SGBDR. O motor de processamento relacional foi ilustrado e relacionado com os restantes módulos de um SGBDR na Figura 2.1 da Secção 2.3.

Um motor de processamento relacional é composto por um *módulo de análise lógica e gramatical* (parser), um *módulo de optimização* (optimizer), e um *módulo de execução* (evaluator) [35], conforme ilustrado na Figura 2.4

Os pedidos formulados pelo utilizador variam desde as interrogações mais simples até às mais complexas, sendo estas últimas caracterizadas por apresentarem várias interrogações encadeadas, nas quais o resultado das primeiras é usado como entrada nas seguintes. O módulo de análise lógica e gramatical tem como objectivo receber uma interrogação SQL de alto nível e fazer a sua análise e decomposição numa estrutura de blocos organizados em forma de árvore, denominada *árvore de interrogação*. Nesta estrutura cada bloco representa uma interrogação simples com no máximo uma cláusula de cada tipo (SELECT, FROM, WHERE, entre outras). As ligações hierárquicas entre os blocos representam o

encadeamento lógico entre as execuções, indicando as relações de entrada e de saída entre as várias interrogações simples.

Numa segunda fase, a árvore de interrogação é passada ao módulo de optimização que é responsável por identificar um *plano de execução* eficiente. Encontrar um bom plano de execução para uma interrogação é uma tarefa que vai para além de escolher a implementação a usar em cada um dos operadores relacionais que a compõem. Por exemplo, a ordem de execução dos próprios operadores é um factor determinante em termos de desempenho. Genericamente, o primeiro passo na optimização de interrogações SQL consiste em transformar cada bloco da interrogação numa expressão de álgebra relacional. De seguida, para cada expressão, o módulo de optimização gera vários planos de execução alternativos. O custo total de cada plano é então estimado tendo em conta três critérios base: o custo estimado de execução, o tamanho estimado do resultado, e a necessidade de ordenar os dados. Para estimar estes custos, o módulo de optimização recorre aos meta-dados armazenados no catálogo do sistema.

Na última fase do processo de análise e execução de uma interrogação SQL, o plano de execução de menor custo é seleccionado pelo módulo de optimização que o entrega ao módulo de execução que efectivamente faz o processamento dos dados das tabelas.

2.5 Controlo de concorrência

Os SGBDR multi-utilizador devem garantir que cada utilizador ou aplicação interage com a base de dados como se fosse o único a utilizar os seus serviços. Isto significa que, os mecanismos de *controlo de concorrência* fornecidos pelo SGBDR são responsáveis por garantir a propriedade de isolamento das transacções.

Num sistema de base de dados multi-utilizador as transacções podem ser executadas sequencial ou concorrentemente. No caso da *execução sequencial*, uma transacção só se inicia quando a anterior tiver terminado, não existindo concorrência nos acessos à base de dados. O isolamento é garantido, mas os recursos do sistema são subaproveitados, o que se reflecte em perdas de desempenho. Por outro lado, o modelo de *execução concorrente* permite melhorar o desempenho permitindo que várias transacções executem concorrentemente, maximizando a utilização dos recursos do sistema. O maior desafio inerente a este modelo consiste em garantir que a concorrência entre as transacções não viola os princípios de isolamento e coerência da base de dados.

Para garantir o isolamento das transacções num modelo de execução concorrente é necessário compreender se as transacções acedem aos mesmos dados. Se duas transacções não manipulam dados em comum é possível executá-las em paralelo sem gerar conflitos. No caso de duas transacções manipularem um mesmo subconjunto de dados poderão surgir conflitos se pelo menos uma delas realizar escritas.

Existem três tipos de conflitos entre transacções: os conflitos de escrita-leitura, que originam que uma transacção possa ler dados de outras transacções que ainda não foram

confirmadas; os conflitos de leitura-escrita, que originam que uma transacção possa executar duas leituras consecutivas sobre os mesmos dados recebendo valores diferentes porque entretanto uma escrita foi realizada por outra transacção; e os conflitos de escrita-escrita, que originam que uma transacção possa sobrepor os seus dados aos de outra transacção que ainda não foi confirmada.

Existem vários métodos para controlo de concorrência entre os quais os método baseados em exclusão mútua e optimista.

O *método baseado em exclusão mútua* consiste em restringir o acesso aos dados partilhados entre as transacções através de mecanismos de exclusão mútua. Estes mecanismos consistem na associação de uma variável de estado aos dados a proteger. A variável tem dois tipos básicos: partilhada ou exclusiva. Para um mesmo objecto do SGBDR, várias transacções podem adquirir simultaneamente trincos partilhados para leitura; no entanto, apenas uma transacção pode adquirir o trinco exclusivo para escrita. O protocolo de *trincos em 2 fases estrito (2pl)* é um dos mais usados para controlo de concorrência através de mecanismos de exclusão mútua. Neste protocolo as transacções executam em duas fases: a fase de crescimento, e a fase de contracção. A fase de crescimento ocorre logo após o início da transacção e consiste na obtenção de trincos. A fase de contracção decorre depois da transacção terminar e consiste na libertação de todos os trincos obtidos. Neste protocolo, todas as operações de obtenção de trincos precedem todas as operações de libertação de trincos.

No *método optimista* as transacções avançam com o mínimo de atrasos e fazem todas as validações de uma só vez no final. Este método consiste em três fases: a fase de leitura, a fase de validação e a fase de escrita. Na *fase de leitura* todas as transacções avançam guardando as alterações em cópias locais. Na *fase de validação* certifica-se a transacção verificando se as alterações realizadas não violam a coerência do SGBDR. Por fim é realizada a *fase de escrita* que consiste em actualizar a base de dados. Este método é especialmente adequado quando a interferência entre transacções é reduzida, como por exemplo, num sistema em que a proporção de leituras é muito superior à de escritas. Quando há conflitos muitas das transacções reiniciam e o desempenho baixa significativamente.

2.6 Gestor de recuperação após falhas

O gestor de recuperação após falhas é responsável por assegurar as propriedades de atomicidade e durabilidade das transacções. Em relação à durabilidade, o maior desafio consiste em tolerar a falha do sistema antes de todos os dados serem escritos para disco. Em relação à atomicidade, o problema resulta da necessidade de escrever páginas da memória para disco que contenham alterações que ainda não foram confirmados e mais tarde vêm mesmo a ser canceladas.

Este módulo do SGBDR mantém alguma informação sobre as transacções em execução de forma a realizar as suas tarefas na eventualidade de uma falha. Em particular, é

mantido em armazenamento estável um histórico com todas as actualizações realizadas na base de dados, sendo que todas as suas alterações são escritas para disco apenas depois de terem sido armazenadas no histórico. Esta regra assegura a durabilidade dos dados mesmo em caso de falha do sistema. O histórico permite ainda desfazer as acções de transacções que abortaram, e refazer as acções de transacções que haviam sido confirmadas.

Existe uma variedade de algoritmos para recuperação da base de dados através da utilização de históricos, mas essa área ultrapassa o âmbito do meu trabalho, pelo que aconselho o leitor a consultar a bibliografia seleccionada.

No capítulo seguinte é dada uma breve panorâmica sobre o estado da replicação de bases de dados.

2.7 Sumário e discussão

Neste capítulo foi realizada uma panorâmica sobre os sistemas de gestão de bases de dados relacionais. Um sistema de gestão de bases de dados (SGBD) é um software desenhado para assistir na manutenção e utilização de grandes conjuntos de dados. O modelo relacional é um modelo conceptual de definição de dados. A interacção do utilizador com o SGBDR é feita através da linguagem SQL. A arquitectura de um SGBDR é tipicamente composta pelos módulos de gestão de memória, gestão de espaço em disco, métodos de acesso, motor relacional, controlo de concorrência, e recuperação após falhas. O motor de processamento relacional é responsável pelo processamento dos pedidos SQL e é tipicamente composto por um módulo de análise lógica e gramatical, um módulo de optimização, e um módulo de execução. O módulo de controlo de concorrência garante a propriedade de isolamento das transacções. E o módulo de recuperação após falhas garante as propriedades de atomicidade e durabilidade das transacções.

As técnicas de replicação de bases de dados são bastante importantes para melhorar a disponibilidade de um SGBDR na presença de faltas. A secção seguinte realiza uma panorâmica sobre as diferentes aproximações para replicação de bases de dados.

Capítulo 3

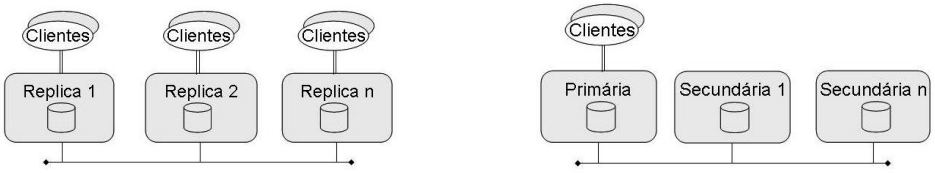
Panorâmica da Replicação de Bases de Dados

A replicação é um tópico cada vez mais importante no contexto das bases de dados, e serve sobretudo para aumentar a disponibilidade do sistema uma vez em caso de falha, permitindo redireccionar os clientes para as réplicas operacionais. Por outro lado, oferece também melhorias na capacidade de escala, ao permitir a execução paralela de pedidos de clientes nas diferentes réplicas. Finalmente, pode permitir uma menor latência no acesso, explorando a localidade dos dados. Um dos maiores desafios neste tipo de sistemas consiste em minimizar o impacto negativo que os protocolos de replicação podem introduzir ao nível do desempenho do SGBD.

As diferentes aproximações para replicação de bases de dados podem ser classificadas sob várias perspectivas. Neste trabalho serão considerados três critérios de classificação: a arquitectura, o modo como as operações são propagadas entre os servidores, e ainda o nível a que a replicação poderá ser implementada. Por fim serão descritos três dos protocolos mais usados para replicação de base de dados: máquina de estados, primária-secundárias, e baseado em certificação.

3.1 Arquitectura da replicação

Idealmente, é desejável beneficiar das vantagens da replicação, tais como acesso descentralizado aos dados, localidade e tolerância a faltas, de uma forma transparente. A arqui-



(a) Arquitectura de cópias activas. (b) Arquitectura de cópias passivas.

Figura 3.1: Arquitecturas de replicação.

tectura que fornece todas estas vantagens é denominada *cópias activas* (*multi-master*) e consiste em manter todas as réplicas como cópias equivalentes [15]. Nesta arquitectura todas as réplicas estão aptas a processar pedidos de clientes, conforme ilustrado na Figura 3.2(a). Infelizmente, quando múltiplas transacções executam concorrentemente em diferentes réplicas é necessária a sua coordenação para a coerência dos dados, conforme referido na Secção 2.5. Em certos casos, esta coordenação pode introduzir um impacto negativo no desempenho do SGBD ao ponto de inviabilizar a utilização deste modelo.

Dado este problema, algumas estratégias de replicação recorrem a arquitecturas de *cópias passivas* (*single-master*). Nesta arquitectura existe uma réplica primária onde são realizadas todas as transacções que actualizem a base de dados. Neste modelo, não é necessária coordenação entre as réplicas, uma vez que todas as actualizações são realizadas na réplica primária, conforme ilustrado na Figura 3.2(b). No entanto, os recursos das réplicas secundárias são pouco explorados e, em caso de transacções que alterem a base de dados, não é possível explorar a localidade dos dados.

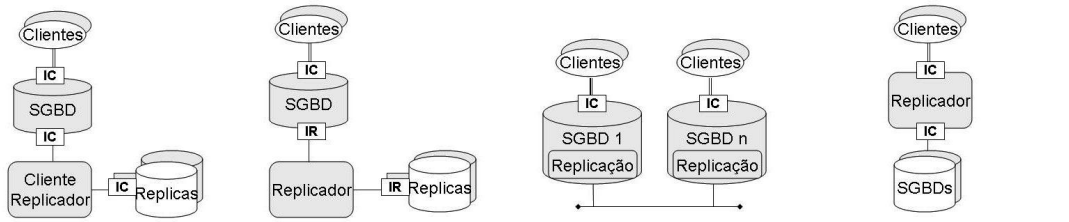
3.2 Sincronismo entre réplicas

Um dos principais objectivos da replicação é evitar perdas de dados. No melhor caso, o protocolo de replicação garante que todos os dados são duráveis, mesmo quando ocorrem falhas. O modelo de *replicação síncrona* atinge esse objectivo garantindo que todas as actualizações realizadas numa réplica são propagadas para as restantes antes da transacção ser confirmada. O tempo necessário para garantir que as actualizações foram recebidas em todas as réplicas introduz um atraso no processamento da transacção.

Para aplicações em que seja possível tolerar a perda de uma quantidade configurável de dados em benefício de uma melhoria no desempenho surge o modelo de *replicação assíncrono*. Neste modelo uma réplica pode retornar a resposta ao cliente sem garantir que os dados foram persistentemente armazenados nas restantes réplicas. A propagação das actualizações pode ser realizada de uma forma linear ou constante. A primeira consiste em enviar as actualizações a cada transacção. A segunda consiste em definir intervalos de tempo configuráveis para o envio das actualizações. Neste modelo, o total de dados que poderão ser perdidos depende da frequência com que as actualizações são propagadas.

3.3 Nível da concretização

Várias estratégias são possíveis para a concretização dos protocolos de replicação. Nesta secção vamos apresentar quatro das soluções mais adoptadas, discutindo as suas vantagens e desvantagens.



(a) Cliente de IC. (b) Cliente de IR. (c) Extensão ao SGBD. (d) Invólucro ao SGBD.

Figura 3.2: Concretização da replicação.

3.3.1 Replicação concretizada através das interfaces cliente do SGBD

Conforme ilustrado na Figura 3.3(a), nesta aproximação o módulo responsável por realizar a replicação, denominado replicador, estabelece uma ligação ao SGBD através das suas interfaces cliente (IC), como JDBC ou ODBC descritas na Secção 2.2. Geralmente estas soluções baseiam-se na instalação de gatilhos (*triggers*) no SGBD, de forma a capturar todas as alterações realizadas.

Esta aproximação é portável e bastante eficiente, nomeadamente, quando utiliza ligações do lado do servidor. No entanto, a concretização está limitada às estratégias de replicação assíncronas de cópias passivas, uma vez que as interfaces cliente não fornecem as funcionalidades necessárias à replicação síncrona.

3.3.2 Replicação concretizada através das interfaces proprietárias

Conforme ilustrado na Figura 3.3(b), alguns SGBD já incluem nativamente o suporte a algumas estratégias de replicação. Para isso, geralmente recorrem à publicação de uma interface de replicação (IR) bem definida, mas proprietária. Esta solução permite a integração com soluções de replicação concretizadas por terceiros, mas não são extensíveis às mais recentes inovações nos protocolos de replicação, como as soluções que recorrem a comunicação em grupo. Uma excepção é o sistema Oracle Streams [14], que se baseia em normalizações já existentes. No entanto, este sistema apenas suporta replicação assíncrona.

3.3.3 Replicação concretizada nativamente como uma extensão ao SGBD

Conforme ilustrado na Figura 3.3(c), uma concretização nativa é realizada dentro do núcleo do SGBD. A vantagem da concretização nativa de um algoritmo de replicação é que pode tirar partido de um acesso directo às estruturas de dados e às fases de processamento do SGBD.

Esta aproximação é a que apresenta mais potencial para oferecer o melhor desempenho. No entanto este modelo tem uma desvantagem importante na falta de modularidade, uma vez que a concretização dos protocolos de replicação fica extremamente interligada ao

núcleo do sistema. Assim, qualquer alteração ao SGBD tem de ser propagada para o protocolo de replicação. Por outro lado, a falta de modularidade limita a arquitectura de replicação à utilização de SGBD homogéneos.

3.3.4 Replicação concretizada como um invólucro middleware ao SGBD

Conforme ilustrado na Figura 3.3(d), nas soluções por *middleware* existe um nível de indirecção entre o cliente e a base de dados, responsável por realizar a replicação. O nível de indirecção intercepta os pedidos dos clientes através da concretização de invólucros para as interfaces cliente do SGBD, como JDBC ou ODBC descritas na Secção 2.2.

Estas soluções beneficiam do acesso aos pedidos dos clientes antes de estes serem processados pelo SGBD. Para além da replicação assíncrona, estas soluções suportam ainda replicação síncrona, uma vez que o pedido do cliente pode ser propagado para as restantes réplicas antes de ser entregue para execução na base de dados. Para injectar o pedido, o replicador acede à base de dados como uma caixa negra através das interfaces normalizadas já referidas. Esta aproximação tem a vantagem de permitir trocar o sistema de gestão de base de dados utilizado sem que seja necessário realizar alterações quer ao módulo cliente quer ao módulo de replicação. Para além disso, é ainda possível que o sistema replicado opere sobre sistemas de gestão de bases de dados heterogéneos, desde que estes suportem a interface cliente escolhida.

No entanto, por um lado, o nível de indirecção introduzido entre o cliente e a base de dados origina um impacto negativo no desempenho do sistema. Por outro lado, realizar a replicação num módulo externo ao SGBD dificulta a tarefa de extrair informação sobre a transacção sem que algum do trabalho realizado pela base de dados tenha de ser refeito. Em particular, para lidar com execuções não determinísticas, é necessário submeter as transacções de forma sequencial, ou concretizar um escalonador que determine quais as transacções que podem ser executados em paralelo. Nesta última solução, o escalonador tem de realizar uma análise lógica e semântica dos comandos SQL semelhante à análise que será realizada ao nível do SGBD, tema abordado na Secção 2.4. Ambas as soluções implicam perdas a nível do desempenho, a primeira devido à ausência de paralelismo, e a segunda porque requer que parte do processamento realizado no SGBD seja também realizado ao nível do replicador.

3.4 Protocolos de replicação

Nesta secção apresento alguns dos protocolos de replicação que podem ser encontrados na literatura. O objectivo não é fornecer uma listagem completa de todas as estratégias propostas, mas sim capturar os modelos mais comuns. É importante referir que não se apresentaram estratégias com requisitos a nível de trincos distribuídos já que o total de interbloqueios pode aumentar exponencialmente à medida que se adicionam réplicas ao sistema, conforme referido no artigo [22]. Por outro lado, também não serão consideradas

estratégias que forneçam modelos de coerência fracos recorrendo a técnicas de resolução manual ou automática de conflitos.

3.4.1 Máquina de estados

A máquina de estados, ou replicação activa, é uma técnica de replicação descentralizada. Neste modelo os pedidos dos clientes são enviados e entregues a todas as réplicas pela mesma ordem, em seguida são executados em todas elas pela ordem em que são recebidos e, por fim, são enviadas as respostas ao cliente. A ordenação dos pedidos é geralmente garantida através de protocolos de difusão atómica, estes protocolos são responsáveis por garantir que uma mensagem é recebida em todos os nós operacionais segundo uma mesma ordem [19]. As operações de leitura podem ser executadas independentemente por qualquer réplica. Por outro lado, o resultado obtido em qualquer uma das réplicas pode ser enviado ao cliente. Por exemplo, se o cliente executar na mesma máquina que uma das réplicas, poderá usufruir do resultado local. A maior vantagem desta aproximação é a sua simplicidade e transparência, pois mesmo que uma réplica falhe as restantes continuam o processamento e mais tarde respondem ao cliente.

Este modo de replicação baseia-se no pressuposto que, se todas as réplicas possuírem um mesmo estado inicial e executarem os mesmos pedidos, pela mesma ordem, então, cada uma delas realizará o mesmo trabalho e produzirá o mesmo resultado. Este requisito de determinismo nas réplicas nem sempre é fácil de garantir. É necessário que cada comando SQL seja reescrito de forma a substituir todas as expressões não deterministas, como expressões de tempo. Outra fonte de não determinismo prende-se com o escalonamento de transacções concorrentes realizado localmente por cada réplica, nomeadamente a ordem pela qual serão adquiridos os trincos. Para ultrapassar este problema é comum recorrer-se a um escalonador ao nível de middleware, responsável por identificar os comandos SQL que podem ser processados em paralelo, o que introduz uma carga e uma complexidade adicionais.

Este método impõe duas restrições base ao sistema de gestão de bases de dados. Em primeiro lugar, e conforme referido, é necessário evitar execuções não deterministas. Em segundo lugar, é necessário garantir que os pedidos dos clientes são interceptados e propagados para todas as réplicas antes de serem executados. Em particular, é necessário interceptar não só os comandos SQL, como os comandos de BEGIN, COMMIT, e ROLLBACK, que podem ser explicitamente indicados pelo utilizador ou implicitamente desencadeados pelo SGBD.

3.4.2 Replicação passiva

A replicação passiva, também denominada aproximação primária-secundárias, evita o problema da coordenação realizando todas as transacções numa mesma réplica, denominada primária, ou mestre.

Tipicamente este modelo opera da seguinte forma. Os clientes enviam os seus pedidos para a réplica primária. Esta executa-os garantindo a coerência através de mecanismos de controlo de concorrência locais (tal como os referidos na Secção 2.5). Quando uma transacção é confirmada, a réplica primária envia as actualizações para as secundárias que confirmam a sua recepção. Após receber as confirmações, a réplica primária retorna a resposta ao cliente.

O modelo primária-secundárias é particularmente adequado para um esquema de replicação assíncrono. Neste caso, a primária não é obrigada a propagar as alterações imediatamente após cada pedido. Em vez disso, pode acumular algumas actualizações enviando-as todas juntas em plano de fundo.

Este modelo adequa-se a servidores não deterministas, o que se traduz numa das suas maiores vantagens. Em contrapartida, todas as actualizações são centralizadas na réplica primária o que origina uma fraca capacidade de escala em sistemas com uma taxa de escritas elevada.

Este método requer que o SGBD capture os eventos de BEGIN, COMMIT, e ROLLBACK de cada transacção, tenham estes sido desencadeados de forma explícita pelo utilizador ou de forma implícita pelo SGBD. Por outro lado, é ainda necessário capturar o conjunto de escrita de cada transacção, de forma a que possa ser injectado nas restantes réplicas.

3.4.3 Replicação baseada em certificação

As aproximações baseadas em certificação operam permitindo que as transacções executem optimisticamente numa única réplica e, no momento da confirmação, executem um protocolo de certificação coordenada de forma a garantir uma coerência global, conforme descrito em [38, 36, 34]. Esta aproximação deriva dos mecanismos de controlo de concorrência optimistas, referidos na Secção 2.5. Tipicamente, a coordenação global é suportada pela difusão atómica, que estabelece uma ordenação global total entre as transacções concorrentes.

Muitas variantes a este modelo foram propostas, mas nesta secção será apresentada a aproximação cuja certificação se baseia na disseminação do conjunto de escritas realizadas pela transacção [29, 37]. No momento em que a transacção inicia, é escolhida uma réplica para a executar. Geralmente é eleita a réplica mais próxima do cliente, denominada réplica delegada. Para confirmar uma transacção, o seu identificador, versão e conjuntos de escritas é enviado para todas as réplicas com recurso a uma primitiva de difusão atómica. Ao receber esta mensagem, todas as réplicas verificam se a transacção tem a mesma versão da base de dados, caso em que pode ser confirmada. Caso contrário, é necessário verificar se a transacção não gera conflitos com transacções previamente confirmadas. Não existem conflitos se as transacções antigas não actualizaram os mesmo dados que a transacção a certificar. Se for detectado um conflito, a transacção é abortada, caso contrário é confirmada. Uma vez que este procedimento é determinista e que todas as

réplicas, incluindo a réplica delegada, recebem as mesmas transacções pela mesma ordem, a mesma decisão será tomada em todos os nós. A réplica delegada pode então retornar a resposta ao cliente.

As aproximações baseadas em certificação não requerem que todo o processamento seja determinístico. Apenas o processo de certificação tem de exibir essa propriedade. Para além disso, este modelo permite que transacções diferentes sejam executadas concorrentemente em diferentes réplicas. Se o total de conflitos for relativamente pequeno, estas aproximações podem fornecer tolerância a faltas e escalabilidade. É ainda importante referir que foram também propostas aproximações que não requerem o envio do conjunto de leitura, conforme referido em [20]. Uma vez que os conjuntos de escrita apresentam na maioria dos casos uma dimensão considerável, estas soluções poderão ser uma boa alternativa ao método aqui apresentado.

3.5 Sumário e discussão

Neste capítulo foram analisadas as diferentes aproximações para replicação de bases de dados. Relativamente à arquitectura, os sistemas podem ser de cópias activas, na qual todas as réplicas processam pedidos de actualização, ou passivas, na qual apenas uma das réplicas processa as actualizações. Relativamente ao sincronismo entre as réplicas, o modo síncrono garante que a resposta só é enviada ao cliente quando todos os dados são duráveis, enquanto o modo assíncrono tolera a perda de dados em benefício de uma melhoria no desempenho. Várias estratégias são possíveis para a concretização dos protocolos de replicação. Entre as mais adoptadas destacaram-se: a concretização através das interfaces cliente do SGBD, a concretização através de interfaces proprietárias, a concretização nativa, e a concretização como um invólucro middleware. Nesta secção foram ainda apresentados três exemplos de protocolos de replicação: o protocolo de máquina de estados, o protocolo primária-secundárias e o protocolo baseado em certificação.

Da análise realizada é possível constatar que, relativamente à arquitectura e aos modos de sincronização, existem soluções alternativas que podem ser adoptadas de acordo com as características estimadas de concorrência e os requisitos de durabilidade do sistema a replicar. No entanto, relativamente ao nível da concretização, é desejável alcançar uma solução que combine o desempenho da concretização nativa com a portabilidade das concretizações através de interfaces clientes, interfaces proprietárias, e invólucros middleware. Definir e validar essa solução face aos protocolos de replicação apresentados é um dos objectivos deste trabalho e será o tema do capítulo seguinte.

Capítulo 4

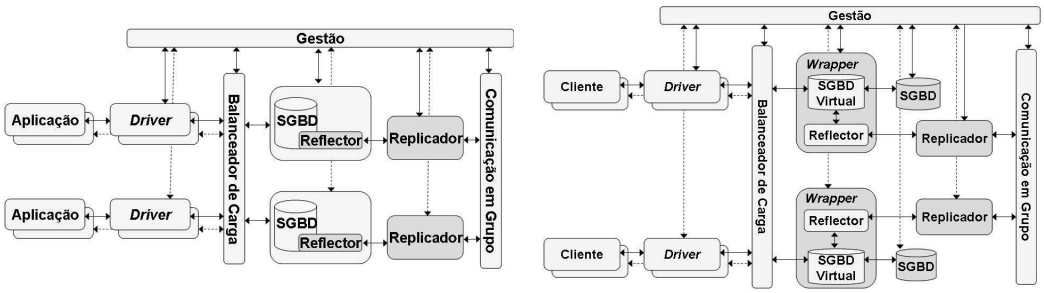
Interface Gorda de Reflexão

Um dos objectivos deste trabalho consiste em contribuir para a definição de uma Interface de Programação (API) de suporte à replicação, denominada *Interface Gorda de Reflexão*, ou simplesmente *Reflector*. A ideia baseia-se na observação que as funcionalidades necessárias ao funcionamento de diferentes protocolos de replicação podem ser fornecidas através de uma interface comum, independentemente da forma como esta será concretizada no produto final. Por exemplo, vários protocolos de replicação requerem a intercepção dos pedidos SQL dos clientes, e deverão ser capazes de realizar essa tarefa utilizando sempre a mesma interface, independentemente desta ser implementada como um invólucro ou uma extensão ao SGBDR usado.

Assim, o Reflector tem como objectivo fornecer os mecanismos necessários para os sistemas de replicação observarem, interceptarem, e modificarem eficientemente o processamento de transacções, independentemente do SGBDR usado. Aproximações baseadas em reflexão foram anteriormente utilizadas de forma viável no desenvolvimento de software configurável e extensível em diferentes áreas aplicacionais, desde linguagens de programação até aos sistemas distribuídos [25, 26]. No contexto dos SGBDR esta aproximação foi adoptada para diferentes propósitos: afinação [32], representação da meta-informação em tabelas relacionais, e intercepção das operações de actualização através de gatilhos (*triggers*), entre outros.

Ao contrário das anteriores utilizações de reflexão em SGBDR, o modelo alvo do Reflector não é o relacional. Em vez disso, a informação é reflectida na linguagem de programação Java, o que se justifica porque não se pretende que a interface seja utilizada por clientes finais no desenvolvimento das suas aplicações, mas sim por clientes ao nível de middleware. A mesma aproximação foi seguida pela Oracle, no desenvolvimento de uma interface de gestão de mensagens denominada Advanced Queuing, para o sistema Oracle Streams [14]. Note-se que, embora desenhada em Java, esta interface pode ser projectada numa variedade de outras linguagens, como C# ou C++, desde que partilhem dos mesmos padrões de desenho [13].

A integração com os sistemas distribuídos já existentes é alcançada através da reutilização, sempre que possível, de interfaces e padrões do J2EE, em especial, tratamento de



(a) SGBDR Compatível. (b) SGBDR Não Compatível.

Figura 4.1: Utilização do Reflector em diferentes cenários.

eventos, do Java AWT[2], e manipulação de dados relacionais, do JDBC[4].

A forma como os módulos de replicação podem beneficiar das funcionalidades oferecidas pelo Reflector de uma forma transparente, tanto num cenário em que o Reflector é disponibilizado nativamente pelo SGBDR, como num cenário em que este é suportado externamente por um terceiro elemento, está ilustrada na Figura 4.1.

A Figura 4.2(a) ilustra a utilização pretendida para o Reflector num cenário em que o SGBDR é compatível. Neste cenário os clientes acedem às bases de dados através de interfaces cliente normalizadas, tal como JDBC ou ODBC. O acesso poderá ser realizado para a réplica mais próxima da aplicação ou poderá ser usado um balanceador de carga, conforme ilustrado. A cada instância da base de dados estará ligado um componente do módulo de replicação, designado simplesmente por replicador. Este componente é responsável por concretizar o protocolo de coerência. O replicador utiliza o Reflector para capturar os eventos e as estruturas necessárias a garantir o nível de coerência desejado.

Embora um dos principais objectivos do projecto GORDA seja promover o desenvolvimento de sistemas de gestão de bases de dados compatíveis com o Reflector, a utilização desta interface em SGBDR não compatíveis toma um papel importante neste processo de adopção, uma vez que alarga a aplicabilidade dos protocolos de replicação. Esta arquitectura transitória é baseada num invólucro middleware, conforme ilustrado na Figura 4.2(b). Neste cenário, o SGBDR não compatível é envolvido num SGBDR *virtual* que implementa o Reflector ao nível de middleware e, com o qual, tanto os clientes como o replicador interagem. Esta aproximação limita as funcionalidades da interface de reflexão uma vez que ao nível de middleware não é possível reflectir toda a informação necessária sobre as fases processamento de transacções. Ainda assim, esta solução preserva a separação lógica entre os protocolos de replicação e a interface de reflexão, permitindo que os mesmos protocolos sejam utilizáveis tanto em SGBDR compatíveis como não compatíveis.

4.1 Análise de requisitos

No âmbito do projecto Gorda foram analisados os requisitos que os protocolos de replicação apresentam de forma a concretizar diferentes protocolos de coerência [24]. Estes foram identificados como os requisitos mínimos que o Reflector tem de suportar e estão listados de seguida:

Consulta de meta informação de objectos: Mecanismos para consultar meta informação relativa a transacções e objectos da bases de dados (como tabelas ou tuplos). Por exemplo, em vários protocolos é necessário que cada objecto tenha um identificador global que será usado nas operações de certificação e recuperação.

Captura de pedidos SQL: Mecanismos para interceptar pedidos SQL de clientes, no formato textual ou estruturados numa árvore de interrogação. Este requisito surge, por exemplo, na replicação activa, de forma a disseminar os pedidos entre as várias réplicas e optimizar a execução através de classes de conflito.

Modificação de pedidos SQL: Mecanismos para modificar ou cancelar pedidos SQL de clientes, no formato textual ou estruturados numa árvore de interrogação. Este mecanismo é necessário para remover operações não deterministas, na replicação baseada numa máquina de estados, ou para subdividir o pedido pelas várias réplicas, em cenários de replicação parcial.

Captura de conjuntos de escrita: Mecanismos para capturar as actualizações de uma transacção numa base de dados, num formato transportável e aplicável nas restantes réplicas. As aproximações de replicação baseada em certificação requerem que esta captura seja realizada dentro dos limites da transacção, enquanto nas aproximações de replicação passiva assíncrona basta apenas transferir os conjuntos de escrita periodicamente.

Captura de conjuntos de leitura: Mecanismos para capturar os identificadores globais dos objectos lidos por uma transacção numa base de dados, num formato transportável e utilizável em operações de certificação nas restantes réplicas. Este mecanismo é necessário na replicação baseada em certificação, quando se pretende detectar conflitos de transacções entre escritas e leituras.

Captura de conjuntos de resultado: Mecanismos que permitam capturar e alterar os conjuntos de resultado de um pedido do cliente. Este mecanismo é necessário num cenário de replicação baseada em certificação, quando um pedido é enviado para outra réplica para aumentar o desempenho, ou em cenários de replicação parcial, de forma a agrupar todos os resultados gerados.

Captura de eventos do ciclo de vida de uma réplica: Mecanismos para observar e controlar o ciclo de vida de uma réplica, nomeadamente, para observar os momentos em

que arranca, recupera a partir de históricos, ou está pronta a receber pedidos de clientes. Estes mecanismos são necessários para garantir a coordenação entre as réplicas. Por exemplo, quando uma réplica recupera pode ser necessário complementar o seu histórico local com dados existentes em históricos remotos, ou até mesmo ser realizada uma transferência de estado a partir de uma réplica operacional.

Captura de eventos do ciclo de vida de uma transacção: Mecanismos para observar e controlar o ciclo de vida de uma transacção, nomeadamente, para observar os momentos em que uma transacção começa, é confirmada, ou é cancelada. Este mecanismo é necessário de forma a manter números de versão, usados na replicação baseada em certificação, por exemplo.

Validação da operação de confirmação de uma transacção: Mecanismos para interceptar e validar a operação de confirmação de uma transacção. Em alguns protocolos de replicação, como os protocolos baseados em certificação, a operação de confirmação de uma transacção tem de ser certificada tendo em conta as transacções que executam nas restantes réplicas. Assim, é necessário fornecer mecanismos que permitam ao replicador, face a um pedido de confirmação, indicar se a transacção pode ser confirmada ou deve ser cancelada.

Tratamento de interbloqueios: Mecanismos de controlo de interbloqueio que possam ser geridos por um elemento externo ao SGBDR. Requisito da replicação baseada em certificação para garantir que uma transacção já certificada não é cancelada por transacções que executam localmente, de forma a resolver interbloqueios.

Captura de meta-informação de replicação: Os protocolos de replicação geralmente requerem a manutenção de meta-dados. Por vezes, os meta-dados relativos à replicação de uma base de dados são armazenados na própria base de dados. Nessa situação é necessário adicionar um passo extra ao seu ciclo de vida que permita a consulta da meta-informação em tempo de arranque, antes de ficar disponível para os utilizadores.

4.2 Arquitectura do Reflector

Os requisitos enumerados na secção anterior baseiam-se maioritariamente em funcionalidades que permitem capturar acontecimentos internos ao SGBDR, e funcionalidades que permitem tomar acções em resposta a esses acontecimentos. Por exemplo, alguns dos protocolos de replicação requerem a captura de pedidos SQL, e, em resposta a essa captura, podem optar por modificar esse mesmo pedido. Os acontecimentos a capturar ocorrem na sua maioria como resposta a estímulos ou actividades externas ao SGBDR, tal como a submissão de transacções pelos utilizadores. Para responder a esta imprevisibilidade, o Reflector foi modelado numa arquitectura orientada a eventos (EDA) [23, 21, 31].

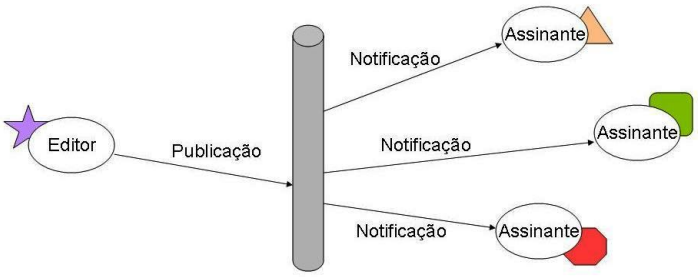


Figura 4.2: Mecanismo de publicação-subscrição numa EDA [31]

A arquitectura EDA define uma metodologia para desenhar e concretizar sistemas nos quais são transmitidos eventos entre componentes fracamente acoplados, conforme definido no padrão de desenho *editor-assinante* [18, 27]. Esta arquitectura geralmente consiste num módulo editor, que fornece funcionalidades de publicação de dados, e num conjunto de módulos assinantes, que pretendem ser notificados quando os dados são alterados. A comunicação é iniciada pela geração de um evento, tipicamente motivado pela ocorrência de um acontecimento relevante, como uma alteração de estado. Os assinantes desse evento são então notificados da sua ocorrência, directamente pelo editor, ou indirectamente através de um elemento intermediário responsável pela sua disseminação, conforme ilustrado na Figura 4.2. Nesta arquitectura o editor não depende da disponibilidade dos assinantes, o que aumenta o nível de independência entre os participantes.

Na arquitectura Gorda, o módulo do Reflector assume o papel de editor, e os seus módulos cliente (como o de replicação) o papel de assinantes. Cada evento é materializado por um objecto composto por atributos, que reflectem a informação interna do SGBDR, e métodos, que permitem aos módulos cliente interagir com o SGBDR em resposta ao evento recebido. Para cada tipo de evento a notificar foi definida uma interface cliente (*listener interface*) com um método que é invocado sempre que um evento desse tipo ocorre. Os módulos cliente devem, em tempo de programação, concretizar a interface cliente dos eventos que pretendem receber, e, em tempo de execução, registar-se no Reflector como receptores desses eventos. Apenas é suportado um assinante por cada notificação, que poderá ser o cliente final, ou um elemento intermediário responsável pela disseminação dos eventos.

As notificações fornecidas pelo Reflector podem ser feitas no modo assíncrono, por omissão, ou no modo síncrono. No modo assíncrono, as notificações são entregues ao cliente e o processamento no SGBDR prossegue. Neste modo, os atrasos introduzidos no processamento do SGBDR são mínimos, com o custo de não ser possível ao cliente modificar os dados relativos a esse evento. No modo síncrono, as notificações são entregues ao cliente e o processamento no SGBDR pára, até que o cliente indique se a execução pode prosseguir ou deve ser abortada. Enquanto o processamento no SGBDR está parado, o cliente pode

tomar as acções desejadas em resposta ao evento. Neste modo, os atrasos introduzidos no processamento do SGBDR dependem do tempo que o cliente demora a permitir que o processamento retome. Este modo de notificações é a chave para a concretização dos protocolos de replicação síncronos.

Outra importante decisão no desenho do Reflector resulta do padrão de desenho das *fachadas* [18, 27]. Num ambiente servidor-clientes, este padrão tem como objectivo envolver o servidor escondendo dos clientes a complexidade da sua concretização. Para isso, o servidor exporta uma interface bem definida que será o seu único componente visível para os clientes. Quando aplicado ao Reflector, este padrão permite que qualquer alteração na sua estrutura ou concretização seja mascarada pela interface definida e, consequentemente, seja transparente para os clientes.

Por outro lado, a interface definida pelo Reflector foi desenhada de forma a permitir a reflexão e manipulação do estado do SGBDR sem necessidade de conversões ou formatações, o que garante um melhor desempenho. A conversão para um formato independente, quando necessária por motivos de heterogeneidade dos SGBDR usados, deve ser realizada por uma camada adicional.

Em resumo, o Reflector foi desenhado como uma fachada entre o SGBDR e os módulos de replicação, responsável pela notificação de eventos com informação interna do seu estado. O desenho do Reflector apresenta as seguintes características:

Fluxo de execução baseado em eventos: O fluxo de execução é originado assincronamente pela ocorrência de eventos no SGBDR. O Reflector notifica os eventos à medida que estes vão ocorrendo no sistema, em vez de os armazenar localmente e enviar em períodos de tempo pré-estabelecidos.

Comunicação para múltiplos receptores: O Reflector suporta um assinante por cada evento, no entanto, pode ser introduzido um nível de indirecção entre ambos responsável por gerir a disseminação dos eventos para múltiplos receptores.

Não mantém estado: O Reflector tem apenas a responsabilidade de notificar os eventos, fornecendo informação que descreve e contextualiza o acontecimento, e funcionalidades que permitam reagir a esse acontecimento. As operações a realizar com resposta a esses eventos, bem como a manutenção do seu estado, são delegadas nos seus clientes.

Processamento complexo de eventos: Os clientes do Reflector compreendem e monitorizam as relações entre os eventos, como por exemplo a causalidade, que determina que evento é causado por outro.

Acoplamentos fracos: Garantidos pelos padrões de desenho do editor-assinante e da fachada.

Fachada fina: a interface definida pelo Reflector foi desenhada de forma a permitir a

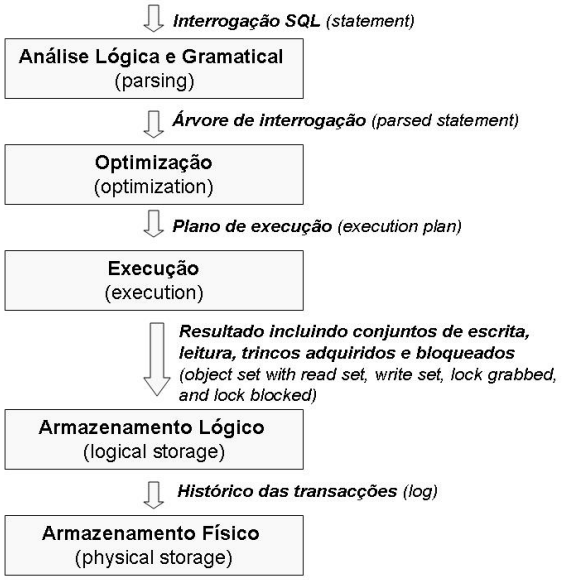


Figura 4.3: Fases de Processamento

reflexão e manipulação do estado do SGBDR sem necessidade de conversões ou formatações, o que garante um melhor desempenho.

Diferentes modos de notificação: O sistema permite que as notificações sejam recebidas em modo assíncrono, por omissão, ou no modo síncrono. No modo assíncrono, o Reflector não tem de esperar que os receptores dos eventos os processem. No modo síncrono, o Reflector pára o processamento até que o receptor indique se a execução pode prosseguir ou deve ser abortada.

4.3 Fases de processamento

Um dos maiores desafios na concretização das funcionalidades enumeradas na análise de requisitos resulta da necessidade dos protocolos de replicação interagirem com uma variedade de fases de processamento transaccional e, consequentemente, com uma variedade de módulos do SGBDR. Assim sendo, a maior parte do Reflector é dedicada à reflexão das fases de processamento transaccional como uma sequência de etapas lógicas de processamento no SGBDR.

A Figura 4.3 ilustra as fases de processamento transaccional consideradas. Genericamente, o protocolo de replicação pode ser notificado sempre que os dados avançam entre as diferentes fases de processamento. Nessa altura, é possível consultar e modificar a estrutura de dados correspondente, e mesmo atrasá-la ou cancelá-la. Em particular, cada pedido percorre as seguintes fases:

Análise lógica e gramatical: Ao receber um pedido SQL, esta fase é responsável pela sua

análise e decomposição numa estrutura de blocos organizados em forma de árvore, denominada *árvore de interrogação*. Nesta fase, podem ser fornecidas notificações do início e do fim do processamento de um pedido SQL. A primeira notificação suporta os requisitos de observação e modificação de pedidos SQL em formato de texto. Ambas as notificações dão suporte ao requisito de observação de meta informação de objectos.

Optimização: Ao receber uma árvore de interrogação, esta fase é responsável por encontrar um *plano de execução* eficiente, não necessariamente óptimo. Nesta fase, podem ser fornecidas notificações do início e do fim do processamento de uma árvore de interrogação. A primeira notificação suporta os requisitos observação e modificação de pedidos SQL em formato estruturado. Ambas as notificações dão suporte ao requisito de observação de meta informação de objectos.

Execução: Ao receber um plano de execução, esta fase é responsável por levar a cargo o seu processamento produzindo os *conjuntos de escrita*, *leitura*, *trincos adquiridos*, e *trincos bloqueados*. Nesta fase, podem ser fornecidas notificações do início e do fim da formatação de um plano de execução. Ambas as notificações dão suporte ao requisito de observação de meta informação de objectos.

Armazenamento lógico: Ao receber o resultado da execução esta fase organiza esses dados no formato lógico do histórico ao nível dos tuplos, acessível independentemente do hardware e da base de dados. Nesta fase, podem ser fornecidas notificações do início e do fim da formatação e escrita do resultado da execução. A primeira notificação suporta os requisitos de captura de conjuntos de escrita e de leitura. Ambas as notificações dão suporte ao requisito de observação de meta informação de objectos.

Armazenamento físico: Ao receber o formato lógico dos dados a escrever para o histórico, organiza-o no formato físico do histórico ao nível das páginas, esse formato é dependente da máquina. Por fim escreve ambas as formatações no histórico do SGBDR. Nesta fase, podem ser fornecidas notificações do início e do fim do processamento das escritas para o histórico. A primeira notificação suporta o requisito de consulta e gestão das operações de armazenamento. Ambas as notificações dão suporte ao requisito de observação de meta informação de objectos.

As fases de processamento transaccional definidas não são mutuamente exclusivas. Isto significa que é possível que diferentes partes da mesma transacção, ou até do mesmo pedido SQL, se encontrem em diferentes fases de processamento, característica denominada por processamento em conduta (*pipelining*). Por exemplo, alguns blocos de dados podem ser escritos para o disco, no nível de armazenamento físico, ao mesmo tempo que, a um nível superior, um novo pedido SQL está a ser processado.

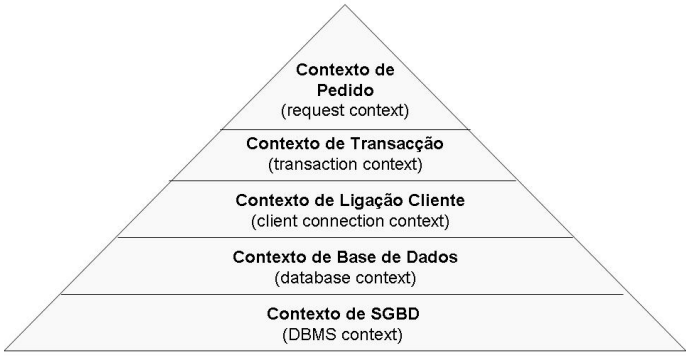


Figura 4.4: Contextos de Processamento

4.4 Contextos de execução

As fases de processamento reflectidas fornecem uma vista geral do funcionamento do SGBDR. No entanto, qualquer que seja a informação de processamento reflectida, é muitas vezes importante agrupá-la e tratá-la ao nível de um contexto de execução comum. Assim, toda a informação de processamento, incluindo eventos, é acompanhada da respectiva informação de contexto. Para além disso, são ainda definidos eventos específicos dos contextos que fornecem notificações quando um novo contexto é iniciado, sofre uma alteração de estado, ou é terminado. Conforme ilustrado na Figura 4.4, foram definidos cinco contextos de execução genéricos e encadeados onde cada nível inferior é uma generalização do superior:

Contexto de SGBDR: de todos os contextos definidos este é o mais genérico, podendo ser interpretado como a fronteira do módulo de reflexão para os restantes módulos da arquitectura Gorda, como o de replicação. Este contexto notifica o arranque e encerramento do SGBDR. A informação reflectida e notificada a este nível suporta os requisitos de observação de meta informação de objectos, e captura dos eventos do ciclo de vida de uma réplica.

Contexto de base de dados: referencia cada uma das base de dados activas. Assume-se que, a cada momento, várias bases de dados podem existir no âmbito de um mesmo SGBDR. Este contexto notifica o arranque e o encerramento de uma base de dados. A informação reflectida e notificada a este nível suporta os requisitos de observação de meta informação de objectos, captura dos eventos do ciclo de vida de uma réplica, e captura de meta-informação de replicação.

Contexto de ligação cliente: identifica a ligação de um cliente a uma base de dados. A

cada momento, várias ligações cliente podem estar activas para uma mesma base de dados. Para além da informação referente à ligação cliente, este contexto permite ainda aceder a informação sobre o utilizador. Este contexto notifica o início e o fim de uma ligação cliente. A informação reflectida e notificada a este nível suporta o requisito de observação de meta informação de objectos.

Contexto de transacção: identifica uma transacção de uma ligação cliente a uma base de dados do SGBDR. Assume-se que, a cada momento, apenas uma transacção é executada em simultâneo para um mesmo contexto de ligação cliente. Este contexto notifica o arranque, o inicio das operações de escrita, o inicio do protocolo de confirmação em duas fases, a confirmação, o cancelamento, e o fim de uma transacção. A informação reflectida e notificada a este nível suporta os requisitos de observação de meta informação de objectos, captura dos eventos do ciclo de vida de uma transacção, e validação da operação de confirmação de uma transacção.

Contexto de pedido: corresponde a cada um dos pedidos do cliente pertencentes a uma transacção. Assume-se que, a cada momento, apenas um pedido pode ser tratado no âmbito de um mesmo contexto de transacção. Este contexto notifica o início e o fim do processamento do pedido de um cliente. A informação reflectida e notificada a este nível suporta o requisito de observação de meta informação de objectos.

Os eventos fornecidos pelas fases de processamento referenciam o contexto de execução envolvente. Este, por sua vez, referencia o contexto mais genérico que lhe sucede, e assim sucessivamente. Desta forma, a partir de um evento é possível aceder a todos os contextos que lhe estão associados. Por exemplo, a partir de um pedido SQL em formato textual é possível consultar o pedido cliente no qual ele se enquadra e, a partir daí, consultar a transacção, a ligação cliente, a base de dados, e o SGBDR correspondentes. Note-se que, alguns contextos não são válidos nos níveis de abstracção mais baixos. Nomeadamente, não é possível associar os blocos de dados que são escritos para disco, ao nível do armazenamento físico, com uma transacção em particular.

Por fim, os protocolos de replicação podem anexar um objecto próprio a cada contexto de execução. Isto permite que a informação disponível em cada contexto seja extendida, conforme necessário, por cada protocolo de replicação. Por exemplo, ao receber o evento que indica o início do processamento de um novo pedido SQL, notificado pela fase de análise lógica e gramatical, o replicador obtém uma referência para o contexto de pedido cliente que lhe está associado. É então possível, ao replicador, associar um objecto seu a esse contexto. As fases de processamento seguintes, do mesmo pedido SQL, estarão associadas ao mesmo contexto de pedido cliente. Assim, sempre que forem recebidas notificações relativas às fases de processamento mais avançadas, o replicador poderá consultar e modificar a informação que associou a esse contexto.

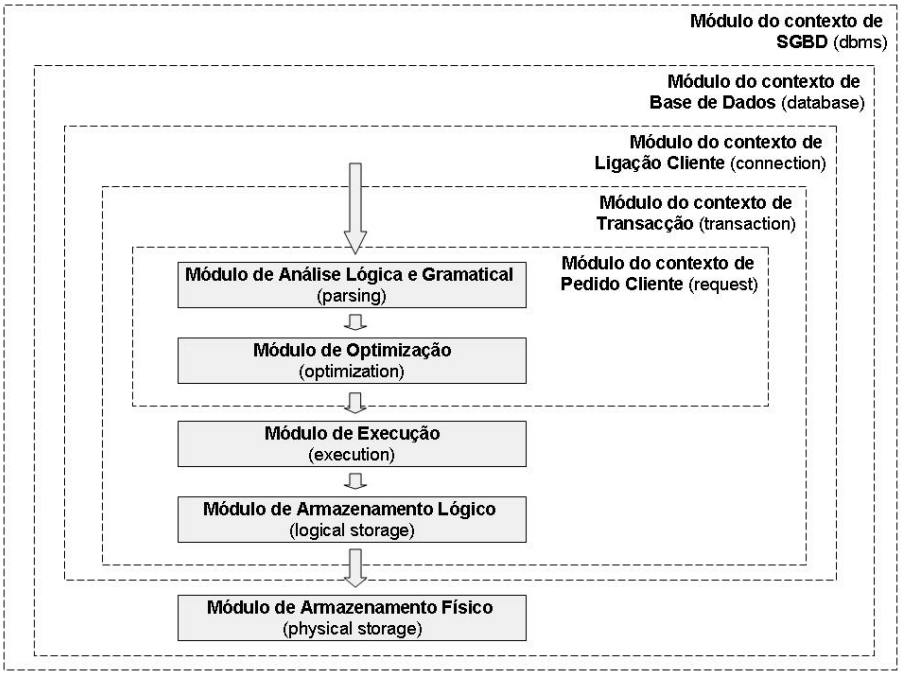


Figura 4.5: Diagrama Resumo dos Componentes da Interface Gorda

4.5 Componentes do Reflector

Uma das principais características do Reflector é que permite separar *o que é feito*, reflectido por informação de processamento, de *quem o está a fazer*, reflectido por informação de contexto. O Reflector pode ser interpretado como uma composição de módulos de reflexão, cada um com a responsabilidade de gerir um contexto de execução ou uma fase de processamento. Assim, o Reflector é composto por cinco módulos de reflexão de contextos e cinco módulos de processamento: módulo do contexto de SGBDR, módulo do contexto de base de dados, módulo do contexto de ligação cliente, módulo do contexto de pedido, módulo de análise lógica e gramatical, módulo de optimização, módulo de execução, módulo de armazenamento lógico, e módulo de armazenamento físico. A Figura 4.5 ilustra os módulos enumerados e as relações entre eles.

Como será mais claro no final do capítulo, diferentes estratégias de replicação requerem a utilização de diferentes partes da informação exposta pelo Reflector. Assim, não é necessário que todos os seus módulos estejam sempre disponíveis. Dependendo dos requisitos do protocolo de replicação e das propriedades do SGBDR a concretização de alguns dos módulos pode ser opcional.

De forma genérica, cada um dos módulos de reflexão define:

Uma interface base do contexto ou estrutura de processamento a reflectir: Em cada módulo existirá uma instância desta interface para cada um dos objectos que o módulo pretende reflectir. A partir desta é possível realizar consultas de estado, gestão de

processamento, e obter referências para outros módulos. Por exemplo, no contexto de base de dados, existirá uma instância desta interface por cada base de dados activa no sistema, esta reflecte informação sobre a base de dados, permite determinar se o seu arranque pode prosseguir ou deve ser cancelado, e permite ainda aceder ao módulos de SGBDR e de ligações cliente que lhe estão associados. Adicionalmente, os contextos de execução poderão fornecer uma outra interface especializada na reflexão da sua meta informação.

Várias interfaces cliente do módulo: Estas interfaces deverão ser implementadas pelos módulos cliente do Reflector, como o replicador, para recepção de notificações de eventos. Por exemplo, para o contexto de base de dados, estão definidas interfaces que permitem aos módulos cliente do Reflector que as implementem, receber notificações sempre que uma base de dados arranca ou encerra.

Uma interface de gestão do processamento no módulo: Em cada módulo existe uma única instância desta interface que tem como objectivo primário gerir os módulos cliente do Reflector que pretendem receber as suas notificações. Por exemplo, no contexto de base de dados, existirá uma única instância desta interface que terá o registo dos clientes que pretendem ser notificados quando uma base de dados arranca ou encerra.

Todas as interfaces base foram desenhadas de forma a permitir a reflexão e manipulação do estado do SGBDR sem necessidade de conversões ou formatações. A conversão para um formato independente, quando necessária por motivos de heterogeneidade dos SGBDR usados, deve ser realizada por uma camada adicional.

Para cada contexto é definido um identificador global, acessível pela sua classe base. No entanto, o Reflector não considera a atribuição de identificadores a cada uma das fases de processamento. Esta opção justifica-se porque as fases de processamento não deverão ser identificadas de forma independente entre si. É desejável obter um meio de identificação que permita relacionar as várias partes de uma mesma transacção, e compreender o fluxo da sua execução entre as várias fases de processamento transaccional definidas. O Reflector delega a definição desse identificador nos seus módulos cliente. A ideia é que estes possam associar informação aos contextos de execução, como o de transacção, com informação que poderá incluir ou não um identificador para as suas fases de processamento. Esse objecto é armazenado pelo Reflector como uma caixa negra e poderá ser alterado ou descartado pelo módulo cliente sempre que desejado.

As secções seguintes apresentam detalhadamente todos os componentes do Reflector através de diagramas de classes UML[17, 33, 27]. Estes descrevem, para cada componente, todas as informações que existem para consulta, actualização e notificação.

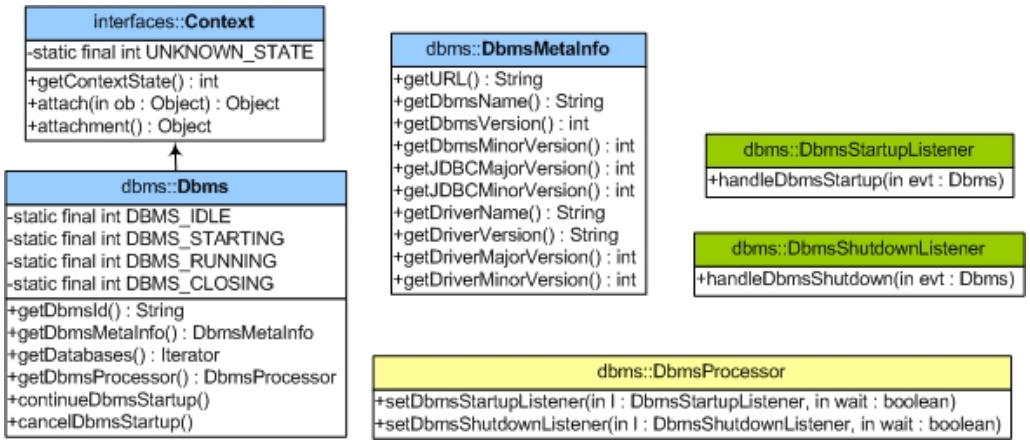


Figura 4.6: Diagrama de classes do contexto SGBD

4.5.1 Módulo do contexto de SGBD

A Figura 4.6 contém o diagrama de classes que resume a informação reflectida ao nível do contexto de SGBD. A cada SGBDR está associada uma classe *Dbms* que permite a observação e gestão da mesmo. Esta classe associa a cada SGBDR um identificador único e um de quatro estados possíveis: "arrancando", "operacional", "encerrando", e "parado". Ambos podem ser consultados através dos métodos *getDbmsId* e *getContextState*, respectivamente. Esta classe permite que os módulos clientes lhe possam anexar um objecto próprio. Esse objecto é armazenado como uma caixa negra e poderá ser alterado ou descartado pelo módulo cliente sempre que desejado, através dos métodos *attach* e *attachment*. A partir de um SGBDR é possível consultar quais as bases de dados que lhe estão associadas a cada momento, usando o método *getDatabases*.

A classe *DbmsMetaInfo* é responsável pelo acesso à meta informação do contexto. Esta informação consiste essencialmente em nomes e versões referentes ao SGBDR, através dos métodos *xVersion* e *xName*. Para além disso, é possível também consultar o URL JDBC do SGBDR[4], através do método *getURL*.

Ao nível do contexto de SGBD foram definidas duas interfaces cliente: *DbmsStartupListener*, e *DbmsShutdownListener*. Estas interfaces deverão ser implementadas pelos módulos cliente do Reflector, como o replicador, de forma a que possam a receber as notificações relativas ao contexto de SGBD. A interface *DbmsStartupListener* permite ao cliente receber notificações sobre o arranque do SGBDR, através do método *handleDbmsStartup* que é invocado pelo Reflector sempre que o estado do contexto é alterado para "arrancando" ou "operacional". A interface *DbmsShutdownListener* permite ao cliente receber notificações sobre o encerramento do SGBDR, através do método *handleDbmsShutdown* que é invocado pelo Reflector sempre que o estado do contexto é alterado para "encerrando" ou "parado". Uma vez implementadas as interfaces, os módulos cliente estão aptos a registar-se, na classe *DbmsProcessor* do Reflector, como receptores das notificações pretendidas.

A classe *DbmsProcessor* tem como objectivo permitir que os clientes do Reflector se registem como receptores das notificações do contexto SGBD. O registo é realizado através dos métodos *setDbmsStartupListener* e *setDbmsShutdownListener* e permite ainda configurar o modo de notificação pretendido: modo bloqueante, ou modo não bloqueante (por omissão). Neste contexto, apenas o evento que notifica o estado "arrancando" suporta o modo bloqueante, e o controlo da execução no SGBDR após este evento é realizado pelo cliente através dos métodos *continueDbmsStartup* e *cancelDbmsStartup* na classe *Dbms* correspondente.

4.5.2 Módulo do contexto de base de dados

A Figura 4.7 contém o diagrama de classes que resume a informação reflectida ao nível do contexto de base de dados. A cada base de dados activa no sistema está associada uma classe *Database*, que permite a observação e gestão da mesma. A classe *Database* associa a cada base de dados um identificador único e um de cinco estados possíveis: "arrancando", "operacional", "encerrando", "encerrada", e "em pânico". Estes podem ser consultados através dos métodos *getDatabaseId* e *getContextState*, respectivamente. O estado "em pânico" é atribuído através do método *panic* e indica que a base de dados está incoerente, não podendo ser acedida pelos utilizadores até que todos os conflitos sejam manual ou automaticamente resolvidos. Esta classe permite ainda gerir a versão da base de dados através dos métodos *getCurrentVersion*, *getMinimumVersion*, e *increaseVersion*. Esta classe permite que os módulos clientes lhe possam anexar um objecto próprio. Esse objecto é armazenado como uma caixa negra e poderá ser alterado ou descartado pelo módulo cliente sempre que desejado, através dos métodos *attach* e *attachment*.

A partir de uma base de dados é possível aceder ao contexto SGBD e aos contextos de ligação de cliente que lhe estão associados a cada momento, através dos métodos *getDbms* e *getConnections* respectivamente. Por outro lado, cada base de dados permite que novas ligações sejam estabelecidas através de uma *DataSource* JDBC disponibilizada pelo método *getDataSource* [4].

A classe *DatabaseMetaInfo* reflecte a meta informação de uma base de dados do sistema e está acessível a partir da classe *Database* que lhe está associada. O método *getUrl* permite consultar o URL JDBC da base de dados, o método *isReadOnly* indica se a base de dados foi arrancada no modo de leitura, e o método *isFrozen* permite consultar se esta está bloqueada ou em funcionamento.

Ao nível do contexto de base de dados foram definidas duas interfaces cliente: *DatabaseStartupListener*, e *DatabaseShutdownListener*. Estas interfaces deverão ser implementadas pelos módulos cliente do Reflector, como o replicador, de forma a que possam a receber as notificações relativas ao contexto de base de dados. A interface *DatabaseStartupListener* permite receber notificações sobre o arranque de uma base de dados através do método *handleDatabaseStartup*, que é invocado pelo Reflector sempre que o estado de

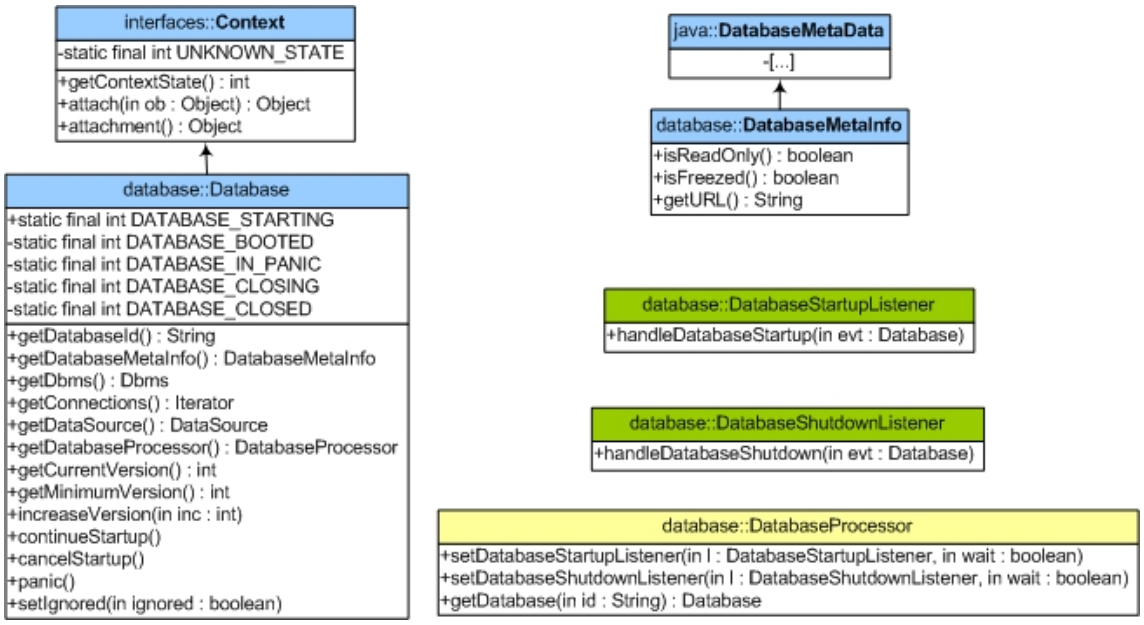


Figura 4.7: Diagrama de classes do contexto de Base de Dados

uma delas é alterado para "arrancando" ou "operacional". A interface *DatabaseShutdownListener* permite receber notificações sobre o encerramento de uma base de dados através do método *handleDatabaseShutdown*, que é invocado pelo Reflector sempre que o estado de uma delas é alterado para "encerrando" ou "encerrada". Uma vez implementadas as interfaces, os módulos cliente estão aptos a registar-se, na classe *DatabaseProcessor* do Reflector, como receptores das notificações pretendidas.

A classe *DatabaseProcessor* gere informação comum a todas as bases de dados do sistema. Em particular, esta classe tem como objectivo principal permitir que os clientes do Reflector se registem como receptores das notificações do contexto de base de dados. O registo é realizado através dos métodos *setDatabaseStartupListener* e *setDatabaseShutdownListener* e permite ainda configurar o modo de notificação pretendido: modo bloqueante, ou modo não bloqueante (por omissão). Neste contexto, apenas o evento que notifica o estado "arrancando" suporta o modo bloqueante, e o controlo da execução da base de dados após este evento é realizado pelo cliente através dos métodos *continueStartup* e *cancelStartup* na classe *Database* correspondente. A classe *DatabaseProcessor* permite ainda obter uma base de dados a partir do seu identificador, através do método *getDatabase* que retorna a instância da classe *Database* que lhe está associado.

O registo dos clientes é realizado no âmbito do contexto de base de dados, isto significa que serão recebidas notificações relativas a todas as bases de dados do sistema. De forma a filtrar as notificações desejadas, o cliente pode cancelar as notificações relativas a uma base de dados através do método *setIgnored* na classe *Database* correspondente. Este método tem um efeito de cascata que implica que todas as notificações relativas a essa

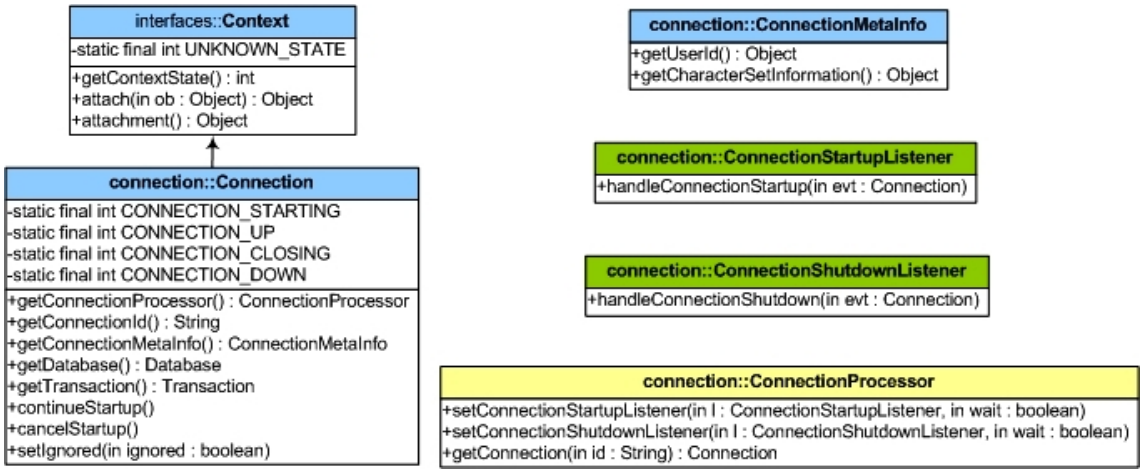


Figura 4.8: Diagrama de classes do contexto de Ligação Cliente

base de dados sejam desactivadas, em específico, não serão recebidas notificações relativas a ligações de clientes, transacções, pedidos, ou fases de processamento que lhe estejam associadas.

4.5.3 Módulo do contexto de ligação cliente

A Figura 4.8 contém o diagrama de classes que resume a informação reflectida ao nível do contexto de ligação cliente. A cada ligação cliente está associada uma classe *Connection* que permite a observação e gestão da mesma. Esta classe associa a cada ligação um identificador único e um de quatro estados possíveis: "estabelecendo", "estabelecida", "terminando", e "terminada". Ambos podem ser consultados através dos métodos *getConnectionId* e *getContextState*, respectivamente. Esta classe permite que os módulos clientes lhe possam anexar um objecto próprio. Esse objecto é armazenado como uma caixa negra e poderá ser alterado ou descartado pelo módulo cliente sempre que desejado, através dos métodos *attach* e *attachment*. A partir de uma ligação cliente é possível consultar a base de dados a que corresponde e a transacção que está activa a cada momento, através dos métodos *getDatabase* e *getTransaction*.

A classe *ConnectionMetaInfo* reflecte informação não só sobre a ligação cliente, como também sobre o utilizador. Em particular, é possível consultar o nome de utilizador e a linguagem da ligação, através dos métodos *getUserId* e *getCharacterSetInformation*, respectivamente.

Ao nível do contexto de ligação cliente foram definidas duas interfaces cliente: *ConnectionStartupListener*, e *ConnectionShutdownListener*. Estas interfaces deverão ser implementadas pelos módulos cliente do Reflector, como o replicador, de forma a que possam a receber as notificações relativas ao contexto de ligação cliente. A interface *ConnectionStartupListener* permite receber notificações quando uma ligação é estabelecida, através do método *handleConnectionStartup* que é invocado pelo Reflector sempre que o estado

de uma delas é alterado para "estabelecendo" ou "estabelecida". A interface *ConnectionShutdownListener* permite receber notificações quando uma ligação é terminada, através do método *handleConnectionShutdown* que é invocado pelo Reflector sempre que o estado de uma delas é alterado para "terminando" ou "terminada". Uma vez implementadas as interfaces, os módulos cliente estão aptos a registar-se, na classe *ConnectionProcessor* do Reflector, como receptores das notificações pretendidas.

A classe *ConnectionProcessor* gere informação comum a todas as ligações de clientes ao sistema. Em particular, esta classe tem como objectivo principal permitir que os clientes do Reflector se registem como receptores das notificações do contexto de ligação cliente. O registo é realizado através dos métodos *setConnectionStartupListener* e *setConnectionShutdownListener* e permite ainda configurar o modo de notificação pretendido: modo bloqueante, ou modo não bloqueante (por omissão). Neste contexto, apenas o evento que notifica o estado "estabelecendo" suporta o modo bloqueante, e o controlo da execução da ligação após este evento é realizado pelo cliente através dos métodos *continueStartup* e *cancelStartup* na classe *Connection* correspondente. A classe *ConnectionProcessor* permite ainda obter uma ligação cliente a partir do seu identificador, usando o método *getConnection* que retorna a instância da classe *Connection* que lhe está associada.

O registo de clientes, como o replicador, no Reflector, é realizado no âmbito do contexto de ligação cliente, isto significa que serão recebidas notificações relativas a todas as ligações de clientes ao sistema. De forma a filtrar as notificações desejadas, o módulo cliente do Reflector pode cancelar as notificações relativas a uma ligação cliente através do método *setIgnored* na classe *Connection* correspondente. Este método tem um efeito de cascata que implica que todas as notificações relativas a essa ligação cliente sejam desactivadas, em específico, não serão recebidas notificações relativas a transacções, pedidos, ou fases de processamento que lhe estejam associadas.

4.5.4 Módulo do contexto de transacção

A Figura 4.9 contém o diagrama de classes que resume a informação reflectida ao nível do contexto de transacção. A cada transacção activa no sistema está associada uma classe *Transaction*, que permite a observação e gestão da mesma. A classe *Transaction* associa a cada base de dados um identificador único e um número de versão, acessíveis através dos métodos *getTransactionId* e *getVersion*. Para cada transacção foram definidos onze estados possíveis: "iniciando", "parada", "activa", "actualizando", "preparando", "preparada", "confirmando", "confirmada", "cancelando", "cancelada", e "terminada". O estado "iniciada" indica que uma nova transacção está a começar. O estado "parada" indica que a transacção já começou mas ainda não foi executado nenhum comando. O estado "activa" indica que a transacção se encontra numa fase de leituras e ainda não foram realizadas escritas. O estado "actualizando" indica que a transacção já iniciou a sua fase de escritas. Os estados "preparando" e "preparada" são referentes à operação que prepara

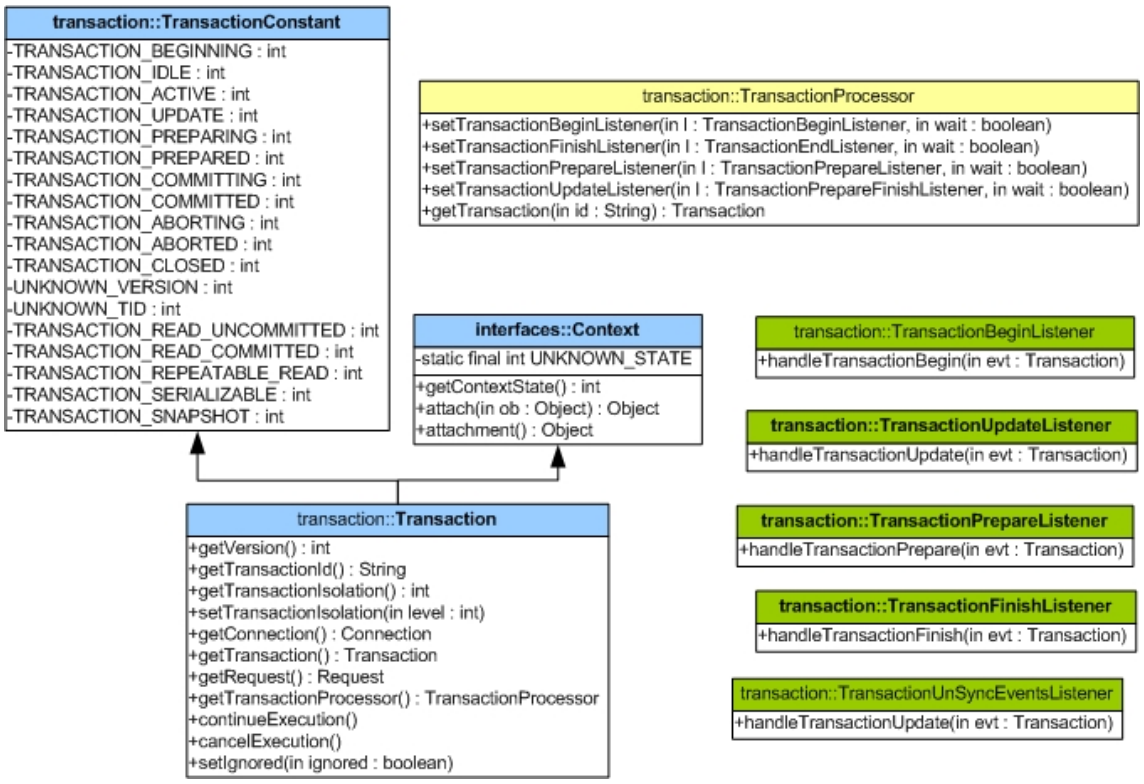


Figura 4.9: Diagrama de classes do contexto de Transacção

uma transacção para ser confirmada e só ocorrem quando está a ser usado o protocolo de 2PC (*two-phase commit*). Os estados "confirmando", "confirmada", "cancelando", "cancelada", e "terminada" ocorrem de forma a terminar a transacção. O estado de uma transacção pode ser consultado através do método *getContextState*. Esta classe permite que os módulos clientes lhe possam anexar um objecto próprio. Esse objecto é armazenado como uma caixa negra e poderá ser alterado ou descartado pelo módulo cliente sempre que desejado, através dos métodos *attach* e *attachment*.

Relativamente à execução concorrente de transacções definiu-se que poderão ocorrer três tipos de anomalias: leituras antigas, leituras não repetíveis, e actualizações perdidas (tema abordado na Secção 2.5. Assim, são considerados cinco níveis de isolamento entre transacções: READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_READ, SERIALIZABLE, e SNAPSHOT. Os primeiros quatro modos correspondem aos níveis de isolamento definidos na norma JDBC[4]. No último modo, todas as transacções vêm a mesma versão da base de dados. A versão correspondente ao estado da base de dados no momento em que a transacção inicia. Neste modo as transacções são processadas optimisticamente, esperando que na altura de serem confirmadas a versão dos dados usados seja ainda a mais actual.

A partir de uma transacção é possível consultar a ligação cliente a que corresponde e o pedido do cliente que está activo a cada momento, através dos métodos *getConnection*

e *getRequest*. No entanto, ao contrário do que acontece em todos os outros contextos, uma transacção pode estar contida dentro de outra (nested transaction). Isto acontece quando uma nova transacção (transacção interior) é iniciada explicitamente através de um comando que já está contido numa transacção (transacção exterior). Neste caso, a transacção interior é delimitada pelos comandos BEGIN TRANSACTION, explicitamente invocado, e COMMIT ou ABORT TRANSACTION, explícita ou implicitamente invocado. As alterações realizadas pela transacção interior só são visíveis pela exterior após ter sido confirmada, mediante as propriedades de isolamento entre transacções. Para cada transacção é possível obter a transacção exterior a que pertence, no caso de existir, através do método *getTransaction*.

Ao nível do contexto de transacção foram definidas quatro interfaces cliente: *TransactionBeginListener*, *TransactionUpdateListener*, *TransactionPrepareListener*, e *TransactionFinishListener*. Estas interfaces deverão ser implementadas pelos módulos cliente do Reflector, como o replicador, de forma a que possam a receber as notificações relativas ao contexto de transacção. A interface *TransactionBeginListener* permite receber notificações sobre o início de uma transacção, através do método *handleTransactionBegin* que é invocado pelo Reflector sempre que o estado de uma delas é alterado para "iniciando", "parada", ou "activa". A interface *TransactionUpdateListener* permite receber notificações sobre o início das operações de escrita numa transacção, através do método *handleTransactionUpdate* que é invocado pelo Reflector sempre que o estado de uma delas é alterado para "actualizando". A interface *TransactionPrepareListener* permite receber notificações sobre a operação que prepara uma transacção para ser confirmada, pelo protocolo 2PC (two-phase commit), através do método *handleTransactionPrepare* que é invocado pelo Reflector sempre que o estado de uma delas é alterado para "preparando" ou "preparada". Por fim, a interface *TransactionFinishListener* permite receber notificações sobre o fim de uma transacção, através do método *handleTransactionFinish* que é invocado pelo Reflector sempre que o estado de uma delas é alterado para "confirmando", "confirmada", "cancelando", "cancelada", ou "terminada". Uma vez implementadas as interfaces, os módulos cliente estão aptos a registar-se, na classe *TransactionProcessor* do Reflector, como receptores das notificações pretendidas.

A classe *TransactionProcessor* gere informação comum a todas as ligações de clientes ao sistema. Em particular, esta classe tem como objectivo principal permitir que os clientes do Reflector se registem como receptores das notificações do contexto de transacção. O registo é realizado através dos métodos *setTransactionBeginListener*, *setTransactionUpdateListener*, *setTransactionPrepareListener*, *setTransactionFinishListener*. As notificações podem ser configuradas no modo bloqueante ou não bloqueante (por omissão). Neste contexto, os eventos que notificam os estados "iniciando", "preparando", "confirmando", e "cancelando", suportam o modo bloqueante. O controlo da execução da ligação após qualquer um desses eventos é realizado pelo cliente através dos métodos *continueExecution*

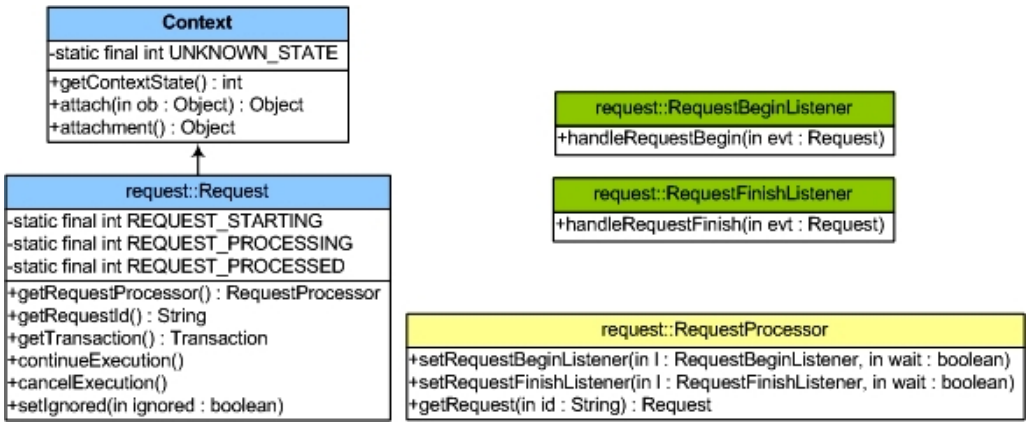


Figura 4.10: Diagrama de classes do contexto de um Pedido Cliente

e *cancelExecution* na classe *Transaction* correspondente. A classe *TransactionProcessor* permite ainda obter uma transacção a partir do seu identificador, usando o método *getTransaction* que retorna a instância da classe *Transaction* que lhe está associada.

O registo dos clientes é realizado no âmbito do contexto de transacção, isto significa que serão recebidas notificações relativas a todas as transacções do sistema. De forma a filtrar as notificações desejadas, o cliente pode cancelar as notificações relativas a uma transacção através do método *setIgnored* na classe *Transaction* correspondente. Este método tem um efeito de cascata que implica que todas as notificações relativas a essa transacção sejam desactivadas, em específico, não serão recebidas notificações relativas a pedidos, ou fases de processamento que lhe estejam associadas.

4.5.5 Módulo do contexto de pedido cliente

A Figura 4.10 contém o diagrama de classes que resume a informação reflectida ao nível do contexto de pedido. A cada pedido está associada uma classe *Request* que permite a observação e gestão da mesma. Esta classe associa a cada pedido um identificador único e um de três estados possíveis: "iniciando", "executando", e "executado". Ambos podem ser consultados através dos métodos *getRequestId* e *getContextState*, respectivamente. Esta classe permite que os módulos clientes lhe possam anexar um objecto próprio. Esse objecto é armazenado como uma caixa negra e poderá ser alterado ou descartado pelo módulo cliente sempre que desejado, através dos métodos *attach* e *attachment*. A partir de um pedido do cliente é possível consultar a transacção a que corresponde, através do método *getTransaction*.

Ao nível do contexto de pedido foram definidas duas interfaces cliente: *RequestBeginListener*, e *RequestFinishListener*. Estas interfaces deverão ser implementadas pelos módulos cliente do Reflector, como o replicador, de forma a que possam a receber as notificações relativas ao contexto de pedido. A interface *RequestBeginListener* permite receber notificações quando um novo pedido começa a ser tratado, através do método

handleRequestBegin que é invocado pelo Reflector sempre que o estado de um pedido é alterado para "iniciando" ou "executando". A interface *RequestFinishListener* permite receber notificações quando um pedido acaba de ser tratado, através do método *handleRequestFinish* que é invocado pelo Reflector sempre que o estado de um deles é alterado para "executado". Uma vez implementadas as interfaces, os módulos cliente estão aptos a registar-se, na classe *RequestProcessor* do Reflector, como receptores das notificações pretendidas.

A classe *RequestProcessor* gere informação comum a todos os pedidos ao sistema. Em particular, esta classe tem como objectivo principal permitir que os módulos cliente do Reflector se registem como receptores das notificações do contexto de pedido. O registo é realizado através dos métodos *setRequestBeginListener* e *setRequestFinishListener* e permite ainda configurar o modo de notificação pretendido: modo bloqueante, ou modo não bloqueante (por omissão). Neste contexto, apenas o evento que notifica o estado "iniciando" suporta o modo bloqueante, e o controlo da execução do pedido após este evento é realizado pelo cliente do Reflector através dos métodos *continueExecution* e *cancelExecution* na classe *Request* correspondente. A classe *RequestProcessor* permite ainda obter um pedido a partir do seu identificador, usando o método *getRequest* que retorna a instância da classe *Request* que lhe está associada.

O registo de clientes do Reflector, como o replicador, é realizado no âmbito do contexto de pedido, isto significa que serão recebidas notificações relativas a todos os pedidos de utilizadores do sistema. De forma a filtrar as notificações desejadas, o cliente do Reflector pode cancelar as notificações relativas a um pedido através do método *setIgnored* na classe *Request* correspondente. Este método tem um efeito de cascata que implica que todas as notificações relativas a esse pedido sejam desactivadas, em específico, não serão recebidas notificações relativas a nenhuma das fases de processamento que lhe estejam associadas.

4.5.6 Módulo de análise lógica e gramatical

O módulo de análise lógica e gramatical é responsável pela reflexão dos pedidos SQL em execução no sistema, e está ilustrado na Figura 4.11. A classe *Statement* reflecte um pedido SQL no formato de texto e associa-lhe um de dois estados possíveis: "iniciando", e "processado". O estado pode ser consultado através do método *getContextReference*. O pedido SQL em formato textual pode ser consultado través do método *getStatement*. A cada pedido SQL está associado um contexto de pedido de cliente, acessível através do método *getRequest*.

Este módulo define ainda uma interface cliente, denominada *StatementExecutionListener*. Esta interfaces deve ser implementada pelos módulos cliente do Reflector, como o replicador, e permite receber notificações sobre o ciclo de vida de um pedido SQL em formato de texto. Em particular, o método *handleStatementExecution* é invocado pelo Reflector sempre que o tratamento de um pedido SQL inicia ou termina. Uma vez implementada

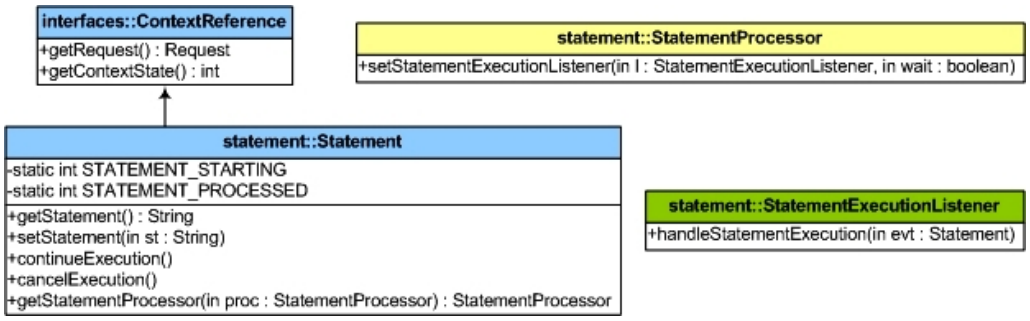


Figura 4.11: Diagrama de classes da fase de Análise Lógica e Gramatical

esta interface, os módulos cliente estão aptos a registar-se, na classe *StatementProcessor* do Reflector, como receptores das referidas notificações.

A classe *StatementProcessor* regista o módulo cliente do Reflector para o módulo de análise lógica e gramatical. O registo é realizado através do método *setStatementExecutionListener* e permite ainda configurar o modo de notificação pretendido: modo bloqueante, ou modo não bloqueante (por omissão). Neste contexto, apenas o evento que notifica o estado "iniciando" suporta o modo bloqueante, e o controlo da execução do pedido após este evento é realizado pelo módulo cliente através dos métodos *continueExecution* e *cancelExecution* na classe *Statement* correspondente.

4.5.7 Módulo de optimização

O módulo de optimização, ilustrado na Figura 4.12, é responsável pela reflexão dos pedidos SQL, no formato estruturado, que se encontram em execução no sistema. A classe *ParsedStatement* reflecte um pedido SQL estruturado e associa-lhe um de dois estados possíveis: "iniciando", e "processado". O estado pode ser consultado através do método *getContextState*. A cada pedido SQL estruturado está associado um contexto de pedido de cliente, acessível através do método *getRequest*. O padrão JDBC já define uma estrutura que representa um pedido SQL estruturado, denominada *PreparedStatement*. De forma a manter a compatibilidade com os sistemas já existentes a classe *ParsedStatement* fornece os mesmos métodos para definição dos atributos do pedido, com o formato *set[tipo]*. Para além disso, são ainda fornecidos métodos que permitem a sua consulta, com o formato *get[tipo]*.

Este módulo define ainda uma interface cliente, denominada *ParsedStatementExecutionListener*. Esta interfaces deve ser implementada pelos módulos cliente do Reflector, como o replicador, e permite receber notificações sobre o ciclo de vida de um pedido SQL em formato estruturado. Em particular, o método *handleParsedStatementExecution* é invocado pelo Reflector sempre que o tratamento de um pedido SQL inicia ou termina. Uma vez implementada esta interface, os clientes estão aptos a registar-se, na classe *ParsedStatementProcessor* do Reflector, como receptores das referidas notificações.

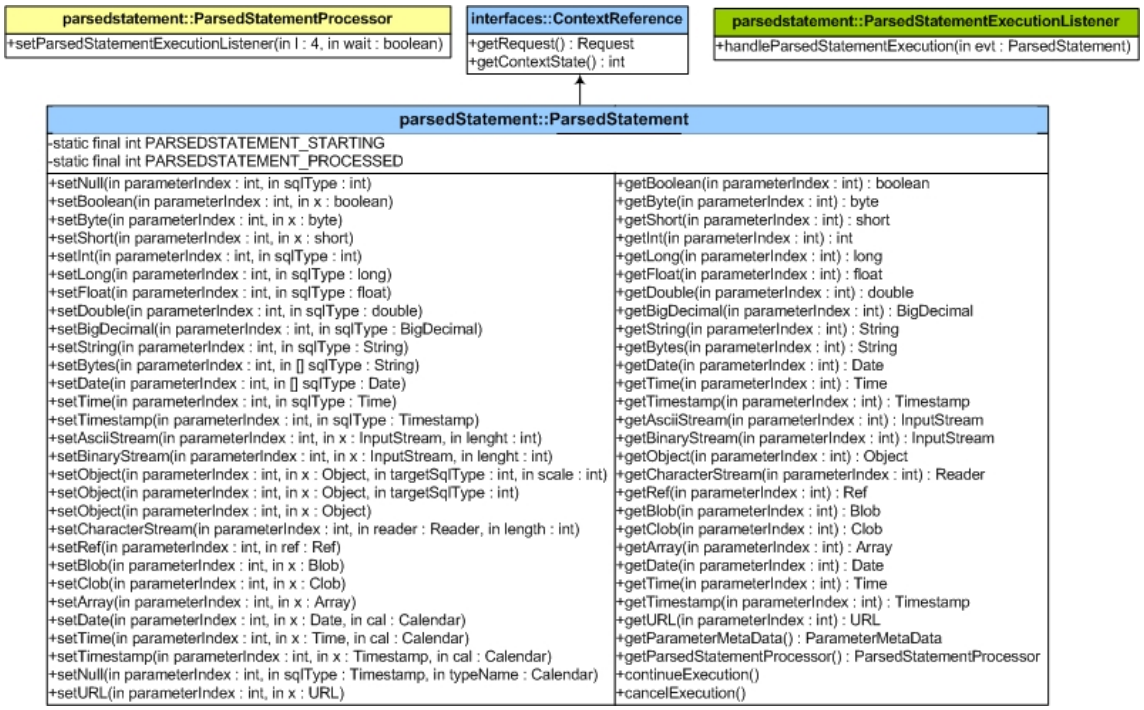


Figura 4.12: Diagrama de classes da fase de Optimização

A classe *ParsedStatementProcessor* regista o cliente do Reflector para o módulo de optimização. O registo é realizado através do método *setParsedStatementExecutionListener* e permite ainda configurar o modo de notificação pretendido: modo bloqueante, ou modo não bloqueante (por omissão). Neste contexto, apenas o evento que notifica o estado "iniciando" suporta o modo bloqueante, e o controlo da execução do pedido após este evento é realizado pelo cliente através dos métodos *continueExecution* e *cancelExecution* na classe *ParsedStatement* correspondente.

4.5.8 Módulo de execução

O módulo de execução é responsável pela reflexão dos planos de execução a processar no sistema, e está ilustrado na Figura 4.13. A classe *ExecutionPlan* reflecte um plano de execução e associa-lhe um de dois estados possíveis: "iniciando", e "processado". O estado pode ser consultado através do método *getContextState*. Cada plano de execução está associado a um contexto de pedido de cliente, acessível através do método *getRequest*.

Este módulo define ainda uma interface cliente, denominada *ExecutionPlanListener*. Esta interfaces deve ser implementada pelos módulos cliente do Reflector, como o replicador, e permite receber notificações sobre o ciclo de vida de um plano de execução. Em particular, o método *handleExecutionPlanExecution* é invocado pelo Reflector sempre que um plano de execução vai iniciar ou termina o seu processamento. Uma vez implementada esta interface, os clientes estão aptos a registar-se, na classe *ExecutionPlanProcessor* do

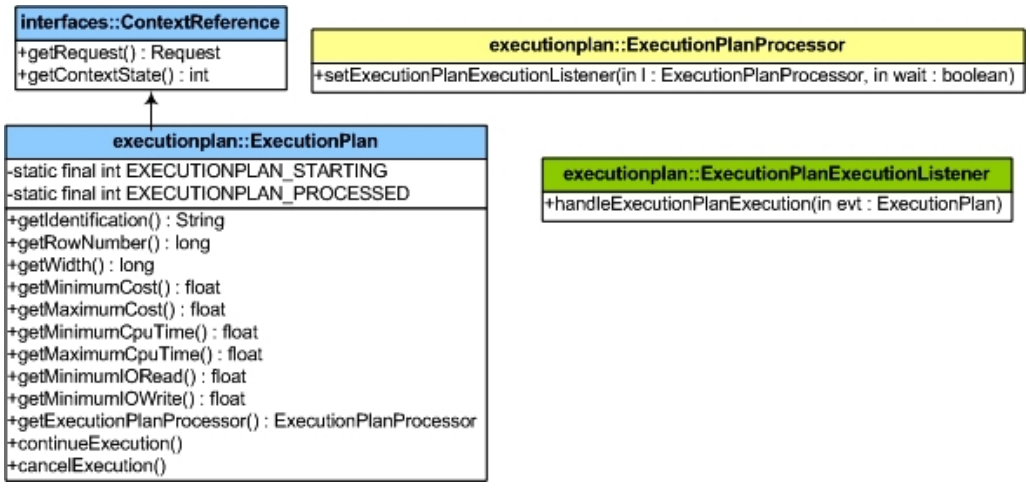


Figura 4.13: Diagrama de classes da fase de Execução

Reflector, como receptores das referidas notificações.

A classe *ExecutionPlanProcessor* regista o cliente do Reflector para o módulo de execução. O registo é realizado através do método *setExecutionPlanExecutionListener* e permite ainda configurar o modo de notificação pretendido: modo bloqueante, ou modo não bloqueante (por omissão). Neste contexto, apenas o evento que notifica o estado "iniciando" suporta o modo bloqueante, e o controlo da execução do pedido após este evento é realizado pelo módulo cliente através dos métodos *continueExecution* e *cancelExecution* na classe *ExecutionPlan* correspondente.

4.5.9 Módulo de armazenamento lógico

O módulo de armazenamento lógico é responsável pela reflexão das operações de transformação dos dados resultantes da execução no formato lógico do histórico ao nível dos tuplos. O diagrama de classes desta fase está ilustrado na Figura 4.15. A classe *ObjectSet* reflecte o conjunto de leituras ou de escritas resultante da fase de execução no formato padrão de um conjunto de resultados (*ResultSet*) do JDBC[4]. Para distinguir se estamos perante um conjunto de escritas ou de leituras é necessário compreender quais as operações realizadas em tempo de execução. Foram definidas quatro tipos de operações possíveis: "modificações de inserção", "modificações de remoção", "modificações de actualização", e "sem modificações". Os três primeiros tipos correspondem a operações de escrita na base de dados, indicando que o resultado da execução é um conjunto de escritas. O último tipo indica que não foram realizadas escritas na base de dados, indicando que o resultado da execução é um conjunto de leituras. O tipo do conjunto de resultados pode ser consultado através do método *getObjectSetType*. Para esta fase foram definidos dois estados possíveis: "iniciando", e "processado". O estado pode ser consultado através do método *getContextState*. A cada conjunto de resultados está associado a um contexto de pedido

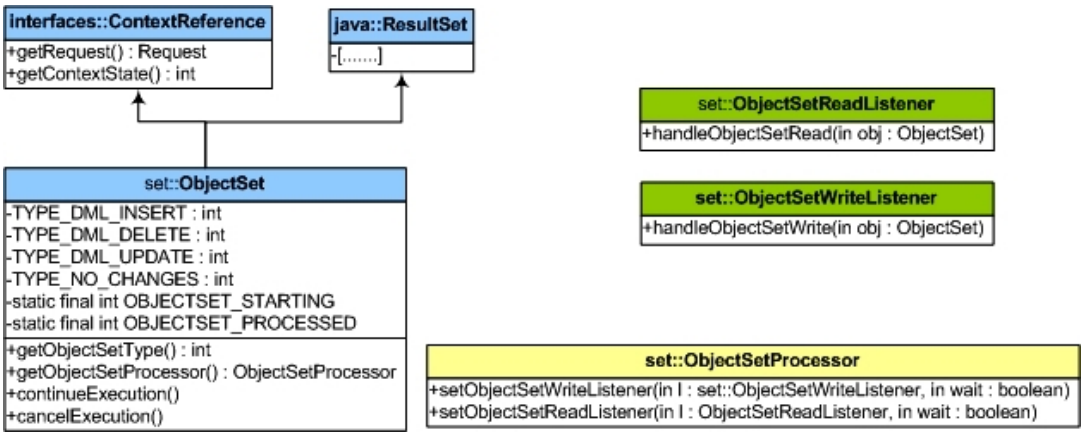


Figura 4.14: Diagrama de classes da fase de Armazenamento Lógico

de cliente, acessível através do método *getRequest*.

Ao nível do contexto de pedido foram definidas duas interfaces cliente: *ObjectSetReadListener*, e *ObjectSetWriteListener*. Estas interfaces deverão ser implementadas pelos clientes do Reflector, como o replicador, de forma a receber as notificações relativas à fase de armazenamento lógico. A interface *ObjectSetReadListener* permite receber notificações relativas a conjuntos de leitura, através do método *handleObjectSetRead* que é invocado pelo Reflector sempre que o estado da formatação de um conjunto de leitura é alterado para "iniciando" ou "processado". A interface *ObjectSetWriteListener* permite receber notificações relativas a conjuntos de escrita, através do método *handleObjectSetWrite* que é invocado pelo Reflector sempre que o estado da formatação de um conjunto de escritas é alterado para "iniciando" ou "processado". Uma vez implementadas as interfaces, os clientes estão aptos a registar-se, na classe *ObjectSetProcessor* do Reflector, como receptores das notificações pretendidas.

A classe *ObjectSetProcessor* regista os clientes do Reflector para o módulo de armazenamento lógico. O registo é realizado através dos métodos *setObjectSetReadListener* e *setObjectSetWriteListener*, e permite ainda configurar o modo de notificação pretendido: modo bloqueante, ou modo não bloqueante (por omissão). Neste contexto, apenas os eventos que notificam o estado "iniciando" suportam o modo bloqueante, e o controlo da execução após este evento é realizado pelo módulo cliente através dos métodos *continueExecution* e *cancelExecution* na classe *ObjectSet* correspondente.

4.5.10 Módulo de armazenamento físico

O módulo de armazenamento físico é responsável pela reflexão das operações de armazenamento físico, em particular, da transformação dos dados num formato físico e do seu armazenamento em disco. O diagrama de classes desta fase está ilustrado na Figura 4.15. A classe *LoggerObjectSet* reflecte o formato dos dados a escrever para o disco o e associa-lhe um de dois estados possíveis: "iniciando", e "processado". O estado pode ser consul-

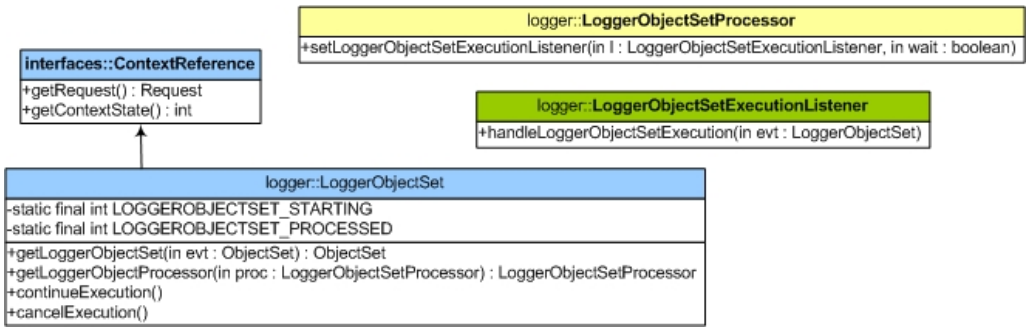


Figura 4.15: Diagrama de classes da fase de Armazenamento Físico

tado através do método *getContextState*. Tal como acontece na fase de armazenamento físico, a estrutura dos dados a armazenar é definida no formato padrão de um conjunto de resultados (*ResultSet*) do JDBC[4].

Este módulo define ainda uma interface cliente, denominada *LoggerObjectSetExecutionListener*. Esta interfaces deve ser implementada pelos módulos cliente do Reflector, como o replicador, e permite receber notificações sobre o armazenamento físico dos dados. Em particular, o método *handleLoggerObjectSetExecution* é invocado pelo Reflector sempre que a formatação e o armazenamento dos dados inicia ou termina. Uma vez implementada esta interface, os clientes estão aptos a registar-se, na classe *LoggerObjectSetProcessor* do Reflector, como receptores das referidas notificações.

A classe *LoggerObjectSetProcessor* regista o cliente do Reflector para o módulo de armazenamento físico. O registo é realizado através do método *setLoggerObjectSetListener* e permite ainda configurar o modo de notificação pretendido: modo bloqueante, ou modo não bloqueante (por omissão). Neste contexto, apenas o evento que notifica o estado "iniciando" suporta o modo bloqueante, e o controlo da execução do pedido após este evento é realizado pelo módulo cliente através dos métodos *continueExecution* e *cancelExecution* na classe *LoggerObjectSet* correspondente.

4.6 Estudo de casos

Nesta secção são ilustradas as potencialidades do Reflector através da descrição de como este poderia ser aplicado a diferentes estratégias de replicação, nomeadamente em cenários de replicação máquina de estados, replicação passiva, e replicação baseada em certificação, abordados na Secção 3.4.

4.6.1 Máquina de estados

O protocolo da máquina de estados requer que todas as réplicas recebam e processem a mesma sequência de pedidos de clientes produzindo um resultado determinista. Para concretizar este modelo é necessário interceptar os pedidos dos clientes antes de serem

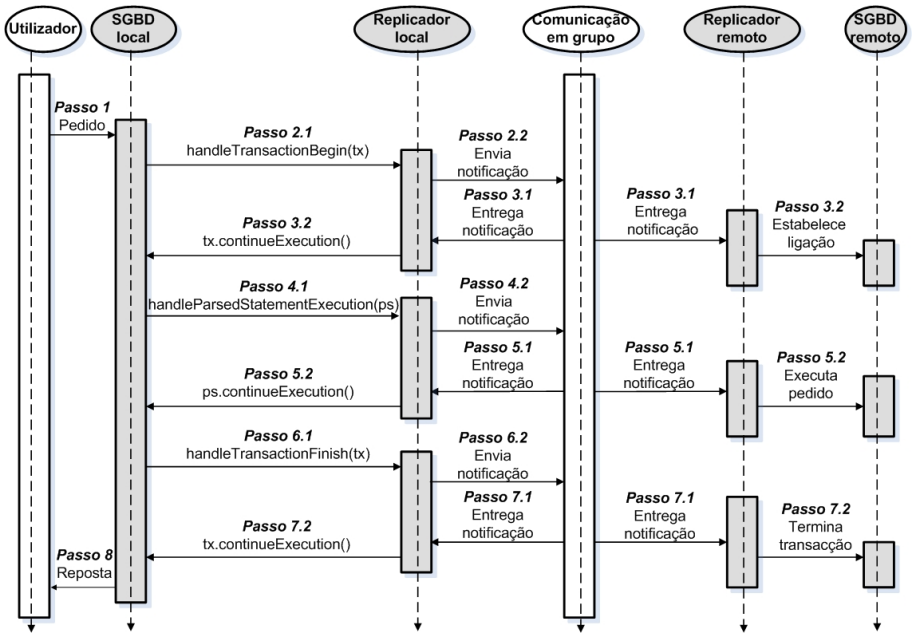


Figura 4.16: Replicação de máquina de estados através do Reflector

processados, forçando execuções deterministas. Em particular, é necessário interceptar os pedidos SQL e os comandos de BEGIN, COMMIT, e ROLLBACK, implicita ou explicitamente desencadeados pelo utilizador. A Figura 4.16 ilustra uma possível solução da utilização do Reflector para concretizar um protocolo de replicação de máquina de estados.

Componentes do Reflector a utilizar A replicação máquina de estados requer a utilização de dois componentes do Reflector: o contexto de transacção e a fase de processamento de optimização. Por um lado, o módulo de transacção, é usado para capturar os momentos em que a transacção começa, é confirmada, ou é cancelada, numa das réplicas. Por outro lado, o módulo de optimização, é usado para capturar a execução de pedidos SQL estruturados.

Algoritmo de replicação Conforme ilustrado na Figura 4.16, uma possível implementação do protocolo da máquina de estados, através do Reflector, consiste nos seguintes passos:

1: Os clientes enviam os pedidos para uma das réplicas. Essa réplica será denominada réplica delegada.

2: Através da interface *TransactionProcessor* o replicador da réplica delegada é notificado do início da transacção. O replicador propaga a notificação para todas as réplicas através de uma primitiva de difusão atómica [19].

3: Todos os replicadores recebem a notificação pela mesma ordem. A transacção é re-

tomada na réplica delegada e iniciada nas réplicas remotas, como resultado do estabelecimento de uma nova ligação à base de dados correspondente.

4: A transacção é executada na réplica delegada. Cada vez que um novo comando é iniciado o replicador é notificado através da interface *ParsedStatementProcessor*. O replicador verifica se o comando apresenta expressões que originem execuções não deterministas, como expressões que usem datas ou valores aleatórios. Se existirem, o replicador altera-as por um valor determinista. O comando resultante é então enviado para todas as réplicas pela primitiva de difusão fiável [19].

5: O comando é recebido por todos os replicadores. Estes devem concretizar um escalonador determinista capaz de garantir que dois comandos conflituosos nunca são passados em simultâneo ao SGBDR. Se existir algum conflito, o comando é mantido em espera. Na réplica delegada, quando não existir qualquer conflito, o processamento é retomado. Nas réplicas remotas, quando não existirem conflitos, o comando é injectado no sistema pela ligação estabelecida.

Passos Intermédios: Os passos 4 e 5 são repetidos tantas vezes quantos os comandos da transacção.

6: Através da interface *TransactionProcessor* o replicador da réplica delegada é notificado que a transacção está pronta a ser confirmada ou a abortar. Esta notificação é enviada para todas as réplicas por difusão atómica.

7: Ao receber a notificação para efectivar ou abortar uma transacção, as réplicas remotas executam o comando indicado e a réplica delegada deixa a sua execução prosseguir.

8: Quando o processamento termina, a réplica delegada responde ao cliente.

4.6.2 Replicação passiva

No protocolo de replicação passiva, também denominado primária-secundárias, as transacções que actualizam a base de dados são executadas numa única réplica, a primária. As actualizações podem ser disseminadas dentro ou fora dos limites temporais da transacção mediante ser escolhido um modelo de execução síncrono ou assíncrono, respectivamente. A Figura 4.17 ilustra uma possível solução de replicação passiva usando o Reflector.

Componentes do Reflector a utilizar Esta técnica de replicação requer a utilização do componente de contexto de transacção do Reflector, de forma a capturar os momentos em que as transacções começam e pretendem terminar, na réplica primária. Por outro lado, é ainda necessário o componente da fase de processamento de armazenamento lógico, de forma a extrair informação sobre os conjuntos de escrita dos comandos da transacção executados na réplica primária e injectá-los nas réplicas secundárias.

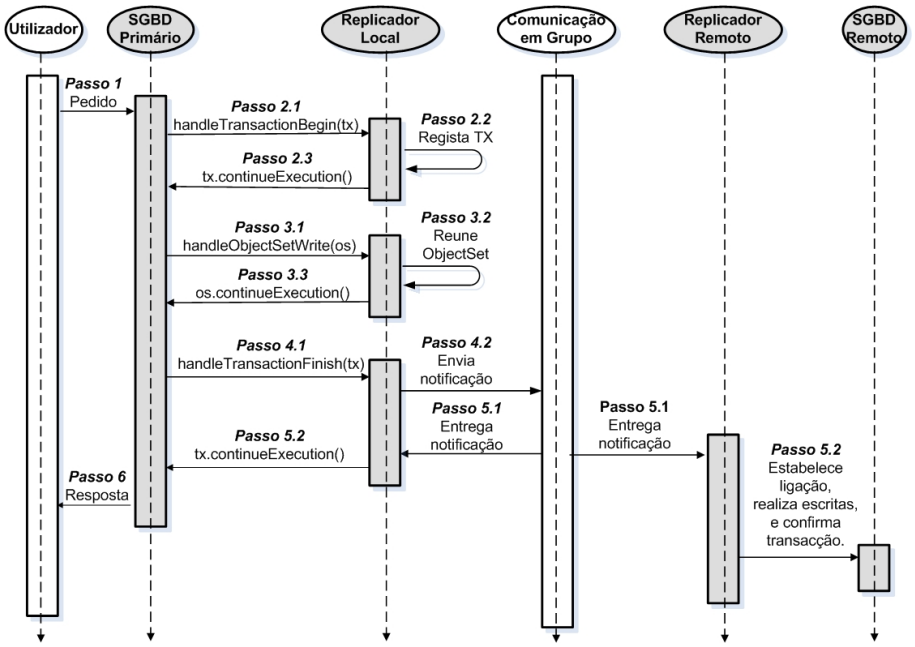


Figura 4.17: Replicação passiva através do Reflector

Algoritmo de replicação Conforme ilustrado na Figura 4.17, uma possível implementação do protocolo da replicação passiva, através do Reflector, consiste nos seguintes passos:

1: Os clientes enviam os pedidos para a réplica primária.

2: Quando uma transacção é iniciada a interface *TransactionProcessor* notifica o replicador da réplica primária que regista informação sobre o evento. Em seguida o replicador permite que a execução prossiga.

3: Assim que um comando é processado na réplica primária, a interface *ObjectSetProcessor* notifica o replicador fornecendo-lhe o *ObjectSet* da execução. Em termos gerais, o *ObjectSet* fornece uma interface para pesquisar o conjunto de escrita e leitura do comando. Neste caso, o objecto é usado para obter o conjunto de escrita. O replicador armazena esses valores num conjunto de escrita global para todos os comandos da transacção.

4: Através da interface *TransactionProcessor* o replicador da réplica primária é notificado que a transacção vai ser cancelada ou confirmada. No primeiro caso, o replicador liberta os recursos referentes a essa transacção e em seguida autoriza que esta seja abortada. No segundo caso, o replicador envia para todas as réplicas o conjunto de escrita global da transacção através de uma primitiva de difusão atómica [19].

5: O conjunto é recebido em todas as réplicas. Na réplica primária, o replicador autoriza que a transacção seja confirmada. Nas réplicas secundárias, é estabelecida uma

ligação à base de dados correspondente, são injectadas as alterações, e a transacção é confirmada.

6: Quando o processamento termina, a réplica primária responde ao cliente.

Uma variante assíncrona deste protocolo pode ser obtida alterando o passo 4 de forma a acumular os conjuntos de escrita obtidos enviando-os segundo intervalos de tempo configuráveis.

4.6.3 Replicação baseada em certificação

Os protocolos baseados em certificação requerem que todas as réplicas recebam e executem a mesma sequência de pedidos SQL de clientes, produzindo um resultado determinístico. Estas aproximações permitem que a transacção execute optimisticamente numa única réplica e, em tempo de ser confirmada, executam um protocolo de certificação coordenado que garante coerência global. Neste protocolo é necessário interceptar os pedidos dos clientes antes de serem processados, de forma a garantir o seu determinismo. Para além disso, é ainda necessário capturar os momentos em que a transacção inicia, pretende ser confirmada, ou pretende abortar. A Figura 4.18 ilustra uma solução possível para concretizar este protocolo tirando partido das funcionalidades oferecidas pelo Reflector.

Componentes do Reflector a utilizar Dadas as suas semelhanças com o modelo de replicação passiva, esta aproximação requer a utilização dos mesmos componentes de reflexão. Especificamente, serão necessários os componentes *TransactionProcessor* e *ObjectSetProcessor*.

Algoritmo de replicação Conforme ilustrado na Figura 4.18, uma possível implementação do protocolo da replicação baseado em certificação, através do Reflector, consiste nos seguintes passos:

1 - 4: Passos iguais aos definidos no cenário de replicação passiva.

5: Ao receber o conjunto de escrita, cada réplica certifica a transacção e decide se deve abortar ou ser confirmada. No primeiro caso, a réplica delegada cancela a transacção e as restantes réplicas descartam-na. No segundo caso, a réplica delegada autoriza que a transacção seja confirmada, e as restantes réplicas estabelecem uma ligação à base de dados correspondente, injectam as alterações, e confirmam a transacção.

6: Quando o processamento termina, a réplica delegada responde ao cliente.

4.7 Sumário e discussão

Neste capítulo foi apresentada a Interface Gorda de Reflexão, ou Reflector, para suporte à replicação. O Reflector fornece meios para os módulos de replicação observarem, inter-

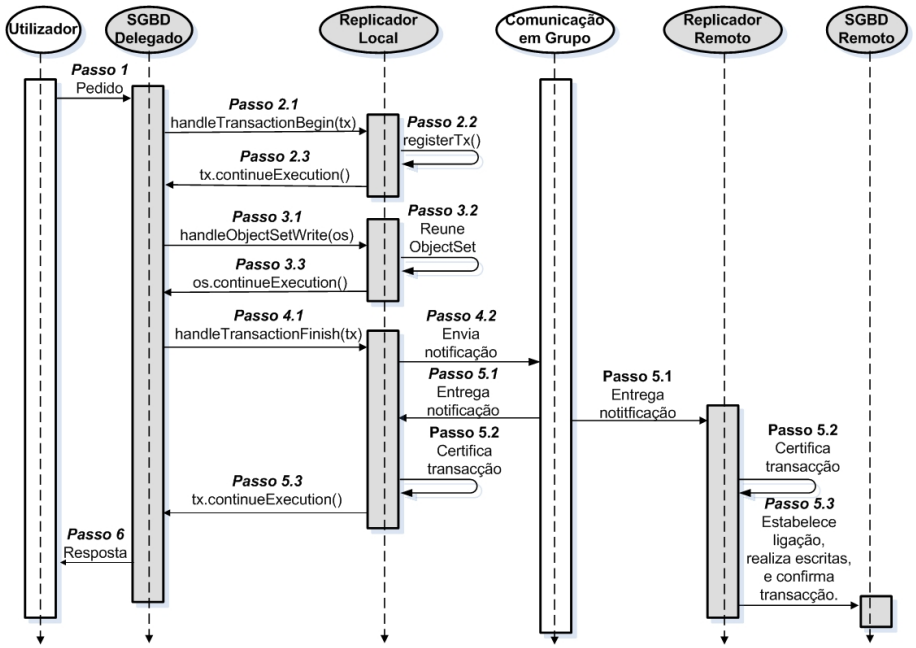


Figura 4.18: Replicação baseada em certificação através do Reflector

ceptarem, e modificarem eficientemente o processamento de transacções, independentemente do SGBDR usado. O Reflector foi modelado numa arquitectura orientada a eventos e desenhado em Java segundo os padrões de desenho do J2EE. Uma das principais características do Reflector é que permite separar *o que é feito*, reflectido por informação de processamento, de *quem o está a fazer*, reflectido por informação de contexto. No final do capítulo, foi ilustrada e validada a utilização do Reflector, através de casos de estudo da sua aplicabilidade em cenários de replicação de máquina de estados, replicação passiva, e replicação baseada em certificação.

Uma vez apresentado o Reflector, e ilustrada a sua utilização pelos protocolos de replicação, é necessário demonstrar que as decisões tomadas no seu desenho foram adequadas e permitem uma extensão pouco intrusiva do núcleo de um SGBD relacional. O capítulo seguinte apresenta a concretização de parte do Reflector no SGBDR Apache Derby.

Capítulo 5

Concretização do Reflector

Uma vez definida a arquitectura e a interface do Reflector, este trabalho tem também como objectivo concretizar uma instância do mesmo num SGBD relacional de código aberto, denominado Apache Derby. Pretende-se assim demonstrar que as opções de desenho do Reflector, a nível de arquitectura, estrutura de módulos, e funcionalidades, foram adequadas e permitem uma extensão pouco intrusiva do núcleo do SGBDR.

Infelizmente, a concretização da totalidade da interface do Reflector não é exequível no intervalo de tempo disponível para a execução de um projecto de CEPEI. Desta forma, foi necessário seleccionar um subconjunto da interface para concretizar. Esta selecção foi feita usando dois critérios.

Em primeiro lugar, as funcionalidades concretizadas devem ser suficientemente ricas para suportar o desenvolvimento de pelo menos um protocolo de replicação. Desta forma será possível validar, no curto prazo, a concretização do Reflector num sistema real.

Em segundo lugar, as funcionalidades concretizadas devem ser representativas dos vários desafios que se colocam no desenvolvimento de uma interface deste tipo. Para atingir este fim, foram analisados os desafios que se apresentam na concretização do Reflector num SGBDR genérico, apresentado na Secção 2.3. Inicialmente, identificaram-se quais as principais considerações a ter em conta de forma a planear uma concretização pouco intrusiva do núcleo do SGBDR. Depois, identificaram-se quais os componentes do SGBDR a extender de forma a suportar o Reflector, tendo em conta que um mesmo componente pode ser extendido de forma a suportar diferentes módulos do Reflector. Por fim, para cada módulo identificaram-se os requisitos da sua concretização. Garantiu-se que a concretização realizada tem em conta cada uma das considerações de planeamento identificadas, e que os módulos do Reflector concretizados solucionam os diferentes requisitos identificados obrigando à extensão parcial de todos os componentes do SGBDR. Desta forma, foram abordadas as diferentes classes de problemas que se podem levantar durante a concretização, pelo que as funcionalidades não suportadas poderão facilmente ser concretizadas reutilizando as soluções adoptadas.

Na Secção 5.1 são apresentados os desafios que se colocam na concretização do Reflector num SGBD relacional. Na Secção 5.2 é feita uma breve introdução ao sistema Derby,

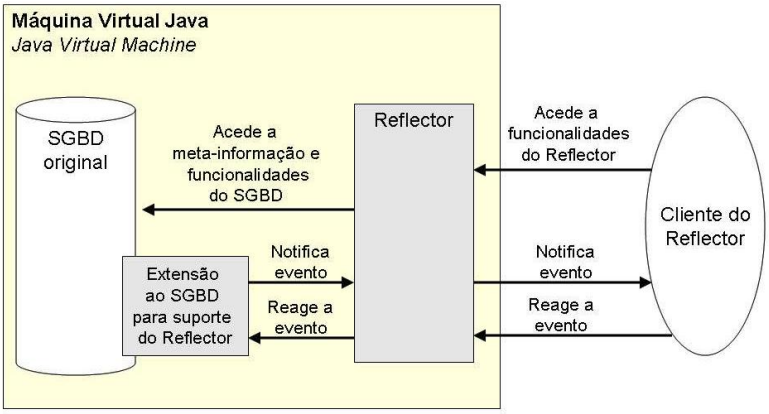


Figura 5.1: Modelo genérico para concretização do Reflector

focando os aspectos mais mais relevantes da sua concretização, que serão determinantes nas decisões tomadas a nível da concretização do Reflector. Na Secção 5.3 é apresentada a concretização do Reflector no SGBDR Apache Derby, sendo detalhadas as soluções adoptadas para cada um dos desafios enumerados. Por fim, na Secção 5.4 é apresentado um breve sumário deste capítulo.

5.1 Desafios da concretização do Reflector

A concretização de uma interface como o Reflector num SGBDR apresenta um conjunto de desafios que terão de ser resolvidos, independentemente do SGBDR escolhido. A Figura 5.1 ilustra o modelo genérico pretendido para a concretização do Reflector. Pretende-se que esta seja tão pouco intrusiva quanto possível, pelo que o código a realizar directamente no núcleo do SGBDR deverá ter como únicas responsabilidades a captura da informação necessária à concretização da lógica do Reflector, e o suporte das funcionalidades que reagem aos eventos capturados. O Reflector, propriamente dito, deverá ser concretizado num módulo separado, que terá como responsabilidade tratar a informação capturada suportando todas as restantes funcionalidades da interface.

Em seguida são apresentados os desafios da concretização do Reflector num SGBDR. Em primeiro lugar, serão apresentadas algumas considerações a ter em conta na fase de desenho do Reflector, ainda antes da concretização efectiva. Depois, são apresentados os desafios a solucionar na concretização das extensões ao SGBDR. Por fim, serão apresentados os tópicos a ter em conta na concretização das restantes funcionalidades do Reflector.

5.1.1 Opções de desenho

Antes de fazer qualquer extensão ao núcleo do SGBDR é necessário analisar a sua arquitectura e modo de funcionamento, de forma a planear uma concretização do Reflector pouco intrusiva. As principais considerações a ter em conta na fase de desenho do Reflector estão descritas a seguir:

Integração do Reflector no SGBDR: É necessário definir como adicionar o Reflector ao sistema. Por exemplo, num sistema modular o Reflector poderá ser mais um dos módulos. Devem ainda ser desenvolvidos mecanismos que permitam que em qualquer parte do código do SGBDR seja possível obter referência para o Reflector.

Localização do código fonte que concretiza o Reflector: O código fonte que concretiza o Reflector deve, sempre que possível, ser colocado fora do núcleo do SGBDR, idealmente num pacote de código fonte separado. O código a adicionar directamente sobre o núcleo do SGBDR deve ser o mínimo possível de forma a permitir a captura de toda a informação necessária. A informação capturada deve então ser entregue ao código específico do Reflector, evitando o processamento de qualquer lógica de reflexão no código fonte original do SGBDR. Esta consideração é particularmente importante uma vez que favorece uma extensão pouco intrusiva do núcleo do SGBDR.

Configuração do Reflector: É necessário permitir a configuração do Reflector através de propriedades ou ficheiros de configuração, evitando que seja necessário manipular e compilar o código fonte para esse fim.

Referências entre módulos do Reflector: É necessário definir como é que um contexto, ou fase de processamento, encontra referência para os contextos que lhe estão associados. Por outro lado, é necessário definir como é que um contexto, ou fase de processamento, encontra a classe de gestão (*Processor*) que lhe está associada. Por fim, é necessário definir como é que o gestor de um contexto, ou fase de processamento, mantém referência para todas as instâncias desse contexto, ou fase de processamento.

Gestão das notificações: É necessário definir um mecanismo de notificações que suporte paralelismo, isto é, que permita que várias notificações sejam desencadeadas em simultâneo. Para cada notificação é necessário definir como encontrar o cliente a notificar, tendo em conta que as notificações relativas a esse contexto podem ter sido canceladas, directamente, ou como consequência do efeito cascata dessa configuração ao nível de um contexto superior.

Atribuição de identificadores únicos aos contextos: É necessário definir como serão atribuídos identificadores únicos a cada contexto de execução.

5.1.2 Extensões ao SGBDR

O primeiro passo na concretização do Reflector consiste na extensão do SGBDR com a captura dos eventos necessários, e o suporte das funcionalidades que permitam reagir a esses eventos. Tanto os eventos a capturar, nomeadamente alterações de estado, como as funcionalidades de resposta aos mesmos, como cancelamento do processamento, foram identificados individualmente para cada módulo do Reflector na secção 4.5. De seguida serão identificados os componentes do SGBDR a extender:

Componentes do SGBDR correspondentes aos contextos de SGBD, ligação, e pedido cliente: Estes componentes devem ser extendidos de forma a informar o Reflector sempre que ocorre uma mudança de estado esperada, permitindo ainda que o processamento seja cancelado no estado "arrancando", "estabelecendo", e "iniciando", respectivamente.

Componente do SGBDR correspondente ao contexto de base de dados: Este componente deve ser extendido de forma a informar o Reflector sempre que ocorre uma mudança de estado esperada, permitindo ainda que o processamento seja cancelado no estado "arrancando". Para além disso, é necessário extender este componente de forma a suportar o modo de pânico de uma base de dados. Devem ser fornecidas funcionalidades que permitam consultar, colocar, e retirar a base de dados do modo de pânico.

Componente do SGBDR correspondente ao contexto de transacção: Este componente deve ser extendido de forma a informar o Reflector sempre que ocorre uma mudança de estado esperada, permitindo ainda que o processamento seja cancelado no estado "iniciando", "preparando", "confirmando", e "cancelando". Para além disso, é necessário garantir que o Reflector possa configurar o modo de isolamento de uma transacção.

Módulo de processamento relacional do SGBDR: É necessário identificar o módulo de processamento relacional de forma a capturar os respectivos pontos de passagem entre as fases de análise, optimização, e execução. É necessário definir como cancelar o processamento no início de cada uma das fases. No início da fase de análise é necessário capturar o pedido SQL em formato de texto. No início da fase de optimização é necessário capturar o pedido SQL em formato estruturado. E no início da fase de execução é necessário capturar a meta informação relativa ao plano de execução. Se necessário, este módulo deve ser extendido de forma a permitir que o pedido SQL, em formato de texto ou estruturado, seja alterado pelo Reflector.

Componente responsável pelas operações de escrita e leitura: É necessário identificar os pontos onde são lidos, inseridos, actualizados, e removidos dados, de forma a obter os valores do conjunto de escrita e leitura a reflectir pelo módulo de armazenamento lógico.

Componente responsável pelas operações de armazenamento: É necessário identificar os pontos onde os dados são armazenados para o histórico, reflectindo-os através do módulo de armazenamento físico.

5.1.3 Suporte do Reflector

Uma vez capturada a informação de processamento do SGBDR, é necessário concretizar a lógica do Reflector. Esta concretização de suporte ao Reflector deve ser realizada fora

do núcleo do SGBDR, fornecendo métodos que permitam receber, tratar, e notificar a informação capturada ao nível do núcleo. Os pontos seguintes resumem os desafios que se colocam na concretização dos diferentes módulos do Reflector:

Módulo do contexto de SGBD: Este módulo deve concretizar métodos que permitam, à extensão do SGBDR, informar a ocorrência de uma mudança de estado. É necessário permitir a consulta de meta informação do contexto, em particular, o URL, o nome, a versão do SGBD, e ainda as versões das interfaces cliente suportadas. Para além disso, devem ser concretizadas funcionalidades de consulta do identificador, consulta dos contextos associados, e gestão objecto de anexo. Por fim, é necessário concretizar os métodos que permitem, aos módulos cliente, cancelar o processamento no estado "arrancando". Esta funcionalidade é suportada pela extensão realizada ao SGBDR.

Módulo do contexto de base de dados: Este módulo deve concretizar métodos que permitam, à extensão do SGBDR, informar a ocorrência de uma mudança de estado na base de dados que lhe está associada. É necessário permitir a consulta de meta informação do contexto, em particular, o URL, o tamanho, e o modo de funcionamento da base de dados. Para além disso, devem ser concretizadas funcionalidades de consulta do identificador e dos contextos associados, de gestão da versão e do objecto de anexo, e de acesso à base de dados através de uma DataSource JDBC. Por fim, é necessário concretizar os métodos que permitem, aos módulos cliente, colocar a base de dados em modo de pânico, e cancelar o processamento no estado "arrancando". Estas funcionalidades são suportadas pela extensão realizada ao SGBDR.

Módulo do contexto de ligação cliente: Este módulo deve concretizar métodos que permitam, à extensão do SGBDR, informar a ocorrência de uma mudança de estado. É necessário permitir a consulta de meta informação do contexto, em particular, a identificação do utilizador, e a linguagem utilizada. Para além disso, devem ser concretizadas funcionalidades de consulta do identificador e dos contextos associados, e de gestão do objecto de anexo. Por fim, é necessário concretizar os métodos que permitem, aos módulos cliente, cancelar o processamento no estado "estabelecendo". Esta funcionalidade é suportada pela extensão realizada ao SGBDR.

Módulo do contexto de transacção: Este módulo deve concretizar métodos que permitam, à extensão do SGBDR, informar a ocorrência de uma mudança de estado. Para além disso, devem ser concretizadas funcionalidades de consulta do identificador e dos contextos associados, gestão do anexo do contexto, consulta da versão, e consulta do modo de isolamento da transacção. Por fim, é necessário concretizar os métodos que permitem, aos módulos cliente, configurar o modo e isolamento, e cancelar o processamento no estado "iniciando", "preparando", "confirmando", e "cancelando". Estas funcionalidades são suportadas pela extensão realizada ao SGBDR.

Módulo do contexto de pedido cliente: Este módulo deve concretizar métodos que permitam, à extensão do SGBDR, informar a ocorrência de uma mudança de estado. Para além disso, devem ser concretizadas funcionalidades de consulta do identificador e dos contextos associados, e gestão do anexo do contexto. Por fim, é necessário concretizar os métodos que permitem, aos módulos cliente, cancelar o processamento no estado "iniciando". Esta funcionalidade é suportada pela extensão realizada ao SGBDR.

Módulos de fases de processamento: Estes módulos devem concretizar métodos que permitam, às extensões do SGBDR, informar o início e o fim da fase de processamento. Para além disso, deve ser concretizada a funcionalidade que permite obter o contexto de execução correspondente. Por fim, é necessário concretizar os métodos que permitem, aos módulos cliente, cancelar o processamento no início de cada fase, e ainda consultar e alterar as estruturas de processamento correspondentes. Estas funcionalidades são suportadas pela extensão realizada ao SGBDR.

5.2 O sistema Derby

Em 1996 foi fundada, em Oakland na Califórnia, a empresa Cloudscape Inc, para desenvolver tecnologias de bases de dados em Java, lançando a primeira versão em 1997 sob o nome de JBMS. Depois disso, o nome do produto foi alterado para Cloudscape e novas versões foram colocadas no mercado a cada seis meses. Em 1999 a Cloudscape foi comprada pela Informix Software que viria a ser adquirida pela IBM em 2001. O motor de base de dados passou a ser denominado IBM Cloudscape e o lançamento das novas versões continuou, centrando-se maioritariamente numa utilização embebida em middleware e produtos Java da IBM. Em Agosto de 2004 a IBM cedeu o código para a Apache Software Foundation como Derby, um projecto da Incubator patrocinado pela Apache. Em Julho de 2005 o projecto Derby cresceu do Incubator para o projecto Apache DB.

O Derby [9] é um SGBDR baseado em Java e foi desenhado segundo o objectivo de optimizar o desempenho, com especial ênfase na poupança dos recursos. Este sistema é baseado em tecnologias normalizadas na área das bases de dados, tal como JDBC e Ansi SQL. Assim, são fornecidas as funcionalidades esperadas para um SGBDR, incluindo manipulação através da linguagem SQL, gestão de transacções, controlo de concorrência, gatilhos (*triggers*), e cópias de segurança (*backups*) enquanto a base de dados se encontra operacional. Para além disso, são ainda fornecidas funcionalidades na área de segurança, tais como, autenticação de utilizadores, políticas de controlo de acesso, cifra de ficheiros em disco, e validação de certificados de ficheiros assinados.

A Figura 5.2 ilustra a arquitectura genérica do sistema Derby. Pormenorizadamente, o SGBDR Derby é composto pelos seguintes módulos:

Monitor: O sistema Derby baseia-se na utilização de módulos, que consiste num conjunto

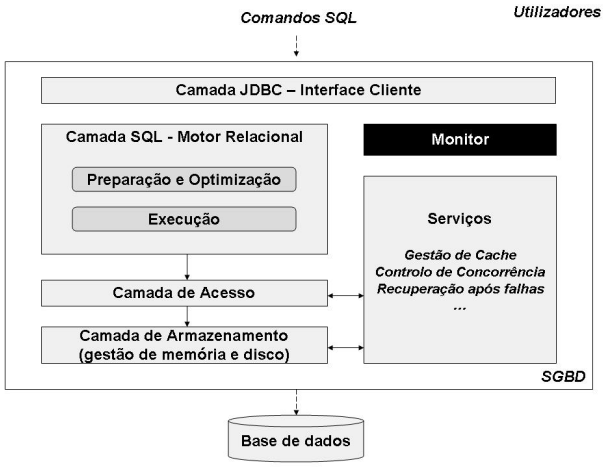


Figura 5.2: Arquitectura do Derby

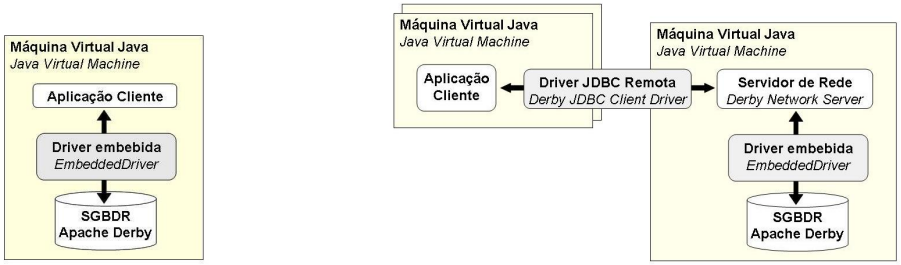
de funcionalidade utilizável com uma interface Java bem definida denominada fábrica (factory). Todos os utilizadores de um módulo fazem-no unicamente através da sua interface, o que origina uma separação lógica entre a especificação e a concretização de um módulo. O monitor gere e configura o sistema Derby associando os pedidos para um módulo com a respectiva concretização do mesmo. O monitor garante ainda que o sistema não é descartado pelo mecanismo de reciclagem de memória do Java (*garbage collector*).

Camada JDBC - interface cliente: Esta camada é o único meio que os utilizadores dispõem para aceder ao Derby, e consiste na concretização dos pacotes *java.sql* e *javax.sql* do Java correspondentes às versões JDBC 2 e JDBC 3, respectivamente.

Camada SQL - motor relacional: Motor de processamento relacional muito completo com suporte para tabelas, índices, vistas, procedimentos, funções, gatilhos, tabelas temporárias, restrições, e chaves. Permite realizar cache de dados e declarações SQL. A nível de controlo de concorrência permite a definição de vários níveis de isolamento, suporta a utilização de trincos de tabelas e de linhas e permite detectar interbloqueios.

Camada de acesso: fornece uma interface de acesso a tabelas baseada no acesso a linhas. Manipula pesquisas de dados, pesquisas de índices, pesquisas de dados usando índices, ordenação de dados, políticas de controlo de acesso, transacções, e níveis de isolamento.

Camada de armazenamento: gere o armazenamento físico dos dados em ficheiros e permite armazenar um histórico das transacções. Fornece um mecanismo de cifra de ficheiros de dados usando o protocolo JCE [3].



(a) Ambiente embebido.

(b) Ambiente cliente-servidor.

Figura 5.3: Ambientes de utilização do Derby.

Serviços: são uma colecção de módulos cooperativos com uma funcionalidade utilizável. Um dos módulos do serviço é considerado o primário e define a interface do serviço. Os serviços são sempre arrancados pelo Monitor e podem ser persistentes, definidos num ficheiro de propriedades e arrancados juntamente com o SGBDR, ou não persistentes, definidos em tempo de execução.

O sistema Derby é um produto que executa numa máquina virtual Java (JVM) [30]. Podem existir múltiplas instâncias do sistema numa mesma máquina, mas cada instância tem de executar numa JVM distinta. Embora o Derby possa ser disponibilizado tanto num ambiente embebido como num ambiente cliente-servidor, na prática o sistema funciona sempre em modo embebido, conforme ilustrado na Figura 5.3. Quando os clientes executam na mesma JVM do Derby, este pode ser embebido na própria aplicação. Neste cenário os clientes acedem ao sistema através da biblioteca (DRIVER) embebida fornecida, que invoca directamente as suas funcionalidades. Quando os clientes executam em JVM distintas, é necessário embeber o sistema numa plataforma servidora. Esta plataforma deve fornecer uma biblioteca de acesso remoto, que permita aos clientes acederem ao sistema a partir de outras JVM. Neste cenário, o servidor funciona como um nível de indirecção entre os clientes e o Derby, responsável por receber os pedidos dos clientes, encaminhá-los para o Derby através da biblioteca embebida, e devolver a resposta retornada ao cliente.

Existem três formas de instalar o sistema Derby numa plataforma servidora [12]. A forma mais simples é utilizando o servidor de rede disponibilizado, que fornece acesso ao Derby a partir da mesma JVM ou através da rede. Outra possibilidade é adquirir um servidor de rede proprietário, como o JVM WebSphere [8]. Por fim, é ainda possível concretizar um servidor próprio para embeber o Derby. Esta opção fornece uma grande flexibilidade de utilização, uma vez que cada aplicação pode definir o modo de acesso ao sistema. Por exemplo, em vez de embeber o Derby numa plataforma que comunica com o cliente através de JDBC, é possível embebê-lo numa plataforma que comunique com o cliente através de um *browser http*.

O sistema Derby arranca automaticamente quando a sua biblioteca JDBC embebida é carregada pelo gestor de bibliotecas do Java (DriverManager). No entanto, o sistema

não é persistente, o que significa que a sua localização deve sempre ser configurada no momento do arranque.

O Derby suporta a configuração dos seus componentes segundo três níveis de abstracção: nível do sistema, nível da base de dados, ou nível do armazenamento físico. A maioria das propriedades são configuradas ao nível do sistema, o que significa que a configuração afecta todos os seus componentes. Algumas propriedades, como o tratamento de erros, só podem ser configuradas a este nível. Por outro lado, podem ser configuradas propriedades especificamente para uma base de dados, como por exemplo, se esta permite escritas. Por fim, ao nível do armazenamento físico, é possível configurar propriedades sobre o modo de armazenamento, como por exemplo, o tamanho das páginas. Estas propriedades têm efeito no momento em que o armazenamento é realizado e não podem ser alteradas depois disso.

As propriedades podem ser dinâmicas ou estáticas, dependendo se é possível ou não alterar o seu valor em tempo de execução. Por exemplo, a propriedade que define a localização do sistema apenas pode ser definida durante o seu arranque. No entanto, a qualquer momento se pode determinar se uma base de dados permite ou não operações de escrita.

As propriedades de sistema podem ser definidas de três formas: na linha de comandos da JVM, no ambiente da aplicação onde o Derby está embebido, ou através do ficheiro de configuração *derby.properties*. Nos primeiros dois casos, as propriedades não são duráveis após o encerramento do sistema. A configuração de uma propriedade de sistema, como a localização do sistema, pode ser feita das seguintes formas:

```
-- Nos parâmetros de arranque da JVM
java -Dderby.system.home=/usr/app/derby AplicacaoTeste

-- No ambiente da aplicação onde o Derby está embebido:
System.getProperties().put("derby.system.home", "/usr/app/derby");

-- No ficheiro derby.properties
derby.system.home=/usr/app/derby
```

As propriedades específicas de uma base de dados são armazenadas nela própria. Esta característica permite que diferentes bases de dados de um mesmo sistema Derby sejam configuradas com diferentes propriedades, o que garante que estas se mantêm válidas mesmo quando a base de dados é copiada ou movida para outra localização. A configuração e consulta destas propriedades é realizada através da invocação de procedimentos do sistema usando a linguagem SQL:

```
-- Procedimento de consulta (recebe o nome da propriedade)
SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(<nome>)

-- Procedimento de configuração (recebe o nome e o valor da propriedade)
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(<nome> , <valor>)
```

Num ambiente cliente-servidor, as propriedades de sistema são definidas exclusivamente no servidor, enquanto as propriedades ao nível da base de dados podem ser configuradas pelo cliente, através dos procedimentos SQL.

A secção seguinte descreve a concretização do Reflector no Derby, indicando e justificando as soluções adoptadas para cada um dos desafios enumerados na secção anterior.

5.3 Concretização no Derby

Nesta secção será apresentada a concretização do Reflector no SGBDR Apache Derby. Toda a concretização foi planeada e desenhada de forma a fornecer o melhor desempenho e a menor latência possíveis. Procurou manter-se uma compatibilidade completa com os padrões de desenho do Derby e com todas as suas ferramentas e interfaces cliente.

Após analisar os desafios da concretização do Reflector num SGBDR genérico, os seus módulos foram agrupados em grupos que apresentam classes de problemas semelhantes:

Grupo 1 - Módulos de SGBD, ligação, e pedido cliente: Obrigam a uma extensão do núcleo do SGBDR que suporte apenas a captura de alterações de estado, e o cancelamento do processamento. A meta informação a capturar pode ser obtida a partir do mesmo componente do SGBDR que foi extendido para seu suporte. Para além disso, devem concretizar funcionalidades que permitam consultar o identificador, consultar os contextos associados, e gerir o objecto de anexo.

Grupo 2 - Módulo de base de dados: Para além de todos os requisitos apresentados pelos módulos do grupo 1, este módulo deve ainda permitir o acesso à base de dados a partir de uma *DataSource* JDBC, e concretizar métodos que permitam colocar a base de dados em modo de pânico.

Grupo 3 - Módulo de transacção: Para além de todos os requisitos apresentados pelos módulos do grupo 1, este módulo deve ainda permitir a consulta e alteração do modo de isolamento de uma transacção.

Grupo 4 - Módulos de análise, optimização, execução, e armazenamento físico: Obrigam a uma extensão do núcleo do SGBDR que suporte a captura de alterações de estado, e o cancelamento do processamento. Para além disso devem ainda permitir o acesso ao contexto de processamento correspondente.

Grupo 5 - Módulo de armazenamento lógico: Embora apresente requisitos iguais aos apresentados pelos módulos do grupo 4, a extensão do núcleo não se baseia apenas na captura de alterações de estado. É necessário identificar os pontos do SGBDR onde são lidos, inseridos, removidos, e alterados dados, de forma a construir o conjunto de escritas e leituras resultante da fase de execução.

De forma a endereçar as diferentes classes de problemas da concretização do Reflector num SGBDR genérico, foram concretizados, no Derby, um módulo de cada grupo identificado. Em particular, foram concretizados os módulos de SGBD, base de dados, transacção, fase de análise, e fase de armazenamento lógico. Nas secções seguintes são apresentadas as opções de desenho tomadas e as soluções encontradas para a concretização de cada um dos módulos enumerados.

5.3.1 Opções de desenho

Nesta secção são apresentadas as opções tomadas para a concretização do Reflector no Derby, tendo como base as considerações de planeamento identificadas na Secção 5.1.1.

Integração do Reflector no SGBDR:

Conforme referido na secção anterior, a concretização do Derby é organizada por módulos que se agrupam em serviços com uma funcionalidade específica. A integração do Reflector no SGBDR foi desenhada de forma a preservar a estrutura do Derby. Assim, o Reflector foi concretizado como um serviço Derby, que denominaremos serviço de reflexão. Os módulos que lhe estão associados correspondem a cada um dos componentes do Reflector, isto é, cada módulo do serviço de reflexão corresponde a um contexto de execução ou a uma fase de processamento do Reflector.

Cada módulo de serviço no Derby exporta uma interface bem definida que permite aceder às suas funcionalidades. A interface a exportar pelos módulos do serviço de reflexão é uma extensão da interface definida pelo Reflector, para esse contexto ou fase de processamento, com métodos que lhe permitem receber e instanciar a informação capturada ao nível do núcleo do Derby. Isto significa que a interface de cada módulo é composta pelos métodos disponibilizados para interacção entre o núcleo do Derby e o Reflector, bem como os métodos disponibilizados para interacção entre o Reflector e os seus clientes, como o replicador (reveja-se a Figura 5.1). A opção de incluir estes últimos métodos na interface do serviço permite que as funcionalidades do Reflector possam ser acedidas ao nível do núcleo do Derby fornecendo-lhe, assim, um meio de introspecção. Desta forma, o Derby pode tornar-se um cliente do seu próprio serviço de reflexão, observando e alterando o seu funcionamento interno conforme necessário. Esta característica permite comprovar que a aplicabilidade de um serviço como o Reflector pode ser extendida a uma variedade de cenários que vão para além da replicação.

O módulo primário do serviço de reflexão denomina-se *ReflectionModule* e é responsável pela definição da interface do serviços. Os métodos definidos na interface permitem, não só, consultar quais os contextos e fases de processamento a reflectir, como também, obter instâncias para os objectos que concretizam cada um deles. A interface define ainda o nome, *GordaReflection*, e a propriedade de configuração do serviço, *derby.service.ReflectionModule*.

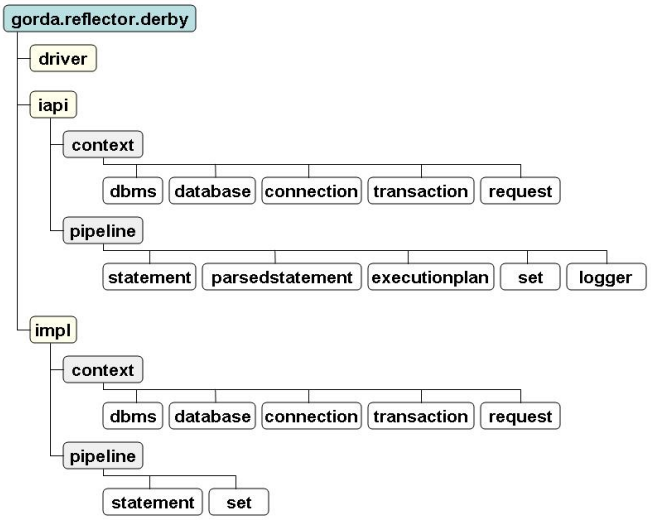


Figura 5.4: Organização do código fonte do Reflector

Localização e organização do código fonte que concretiza o Reflector:

O código fonte do Reflector foi colocado num pacote próprio, denominado *gorda.reflector.derby*, separado do código fonte do núcleo do Derby, que se encontra em *org.apache.derby*, e a sua organização está ilustrada na Figura 5.4. O pacote *gorda.reflector.derby.iapi* contém a especificação dos módulos do serviço de reflexão através de interfaces. O pacote *gorda.reflector.derby.impl* contém a concretização de cada um desses módulos.

Configuração do serviço de reflexão:

Por omissão, o serviço de reflexão está desactivado. A propriedade de configuração do serviço de reflexão permite activá-lo especificando a localização do seu módulo primário, conforme descrito a seguir:

```
-- Configuração do serviço de reflexão
derby.service.GordaReflection=gorda.reflector.derby.iapi.ReflectionModule
```

Para além disso, poderão ser configurados individualmente cada um dos módulos do serviço de reflexão, indicando se devem ou não ser activados, conforme descrito a seguir:

```
-- Configuração de cada um dos módulos de reflexão
derby.gordaReflection.Dbms            = < true / false (por omissão) >
derby.gordaReflection.Database        = < true / false (por omissão) >
derby.gordaReflection.Connection      = < true / false (por omissão) >
derby.gordaReflection.Transaction     = < true / false (por omissão) >
derby.gordaReflection.Request         = < true / false (por omissão) >
derby.gordaReflection.Statement       = < true / false (por omissão) >
derby.gordaReflection.ParsedStatement = < true / false (por omissão) >
derby.gordaReflection.ExecutionPlan   = < true / false (por omissão) >
```

```
derby.gordaReflection.ObjectSet       = < true / false (por omissão) >
derby.gordaReflection.Logger          = < true / false (por omissão) >
```

Tanto a configuração do serviço de reflexão, como a configuração de cada um dos seus módulos, é realizada ao nível do sistema. Para isso poderão ser utilizados os meios disponibilizados pelo Derby para esse efeito: na linha de comandos da JVM, no ambiente da aplicação onde o Derby está embebido, ou através do ficheiro de configuração *derby.properties*.

Referências entre módulos do Reflector:

O Derby é um SGBD baseado em módulos de serviços, pelo que já fornece meios que permitam, a cada módulo, obter referência para os restantes. As técnicas mais utilizadas, para esse fim, são através do Monitor e através do gestor de contextos. O Monitor é responsável por arrancar e gerir os módulos do sistema, permitindo obter referência para cada um deles. O gestor de contextos colecciona os vários contextos de execução à medida que vão sendo criados. O contexto de execução, de um determinado módulo, pode ser-lhe atribuído por um terceiro elemento, no momento do seu arranque, ou pode ser criado por si próprio.

No caso do Reflector, no momento em que cada um dos seus módulos é criado, é-lhe atribuído um contexto de execução com referência para os restantes módulos que lhe estão associados. Isto permite que, a nível do Reflector, todos os módulos tenham referência entre si.

Por outro lado, os componentes do Derby extendidos com código do Reflector incluem, nos seus contextos de execução, referência para o módulo do Reflector que lhe está associado. Isto significa que, as referências entre o núcleo do Derby e o Reflector, são armazenadas nos contextos dos módulos do Derby. Assim, todo o código realizado no núcleo do Derby pode obter referência para o Reflector. Por exemplo, o componente de base de dados do Derby armazena, no seu contexto, a referência para o módulo de base de dados do Reflector. Todos os módulos do Derby que herdem o contexto da base de dados, como por exemplo o módulo de transacção, poderão aceder à referência para o Reflector lá armazenada.

Gestão das notificações:

Conforme referido na Secção 4.2 a maioria dos eventos ocorre como resposta a estímulos exteriores, como por exemplo a submissão de um novo pedido SQL pelo utilizador. De forma a que as notificações possam ser realizadas em paralelo, estas são feitas a partir do próprio fio de execução (*thread*) do utilizador. Isto permite que as notificações bloqueantes mantenham parado apenas o processamento correspondente ao evento capturado, obtendo-se desta forma uma notificação síncrona.

Os clientes do Reflector, para cada notificação, são geridos pelos gestores desse contexto ou fase de processamento, conforme referido na Secção 4.5. O registo de que uma notificação deve ser ignorada, para um determinado contexto, é armazenada no próprio contexto. Para encontrar o cliente do Reflector a notificar, é necessário obter referência para todos os contextos que lhe estão associados, verificando que em nenhum deles as notificações foram configuradas para serem ignoradas. Nesse caso, o cliente a notificar pode ser consultado no módulo de gestão desse contexto ou fase de processamento.

Atribuição de identificadores únicos aos contextos:

O Reflector prevê a atribuição de identificadores únicos a cada um dos contextos de execução, conforme referido na secção 4.5. De forma a manter compatibilidade com todas as ferramentas do Derby foi necessário identificar como são gerados os seu identificadores. Concluiu-se que os identificadores, no Derby, são gerados a por um módulo especializado denominado *UUIDFactory* que os gera no mesmo formato utilizado pelo Microsoft UUIDGEN:

```
-- Identificador gerado pelo módulo UUIDFactory do Derby
E4900B90-DA0E-11d0-BAFE-0060973F0942
```

Assim, os identificadores usados pelo Reflector são gerados através da mesma plataforma.

5.3.2 Concretização e suporte do módulo de SGBD:

De forma a suportar as funcionalidades do contexto de SGBD do Reflector, apresentadas na Secção 4.5.1, é necessário identificar e extender o componente do Derby que lhe corresponde capturando os eventos de arranque e encerramento.

O componente do Derby correspondente ao contexto de SGBD será aquele que, por definição, arranca primeiro e encerra por último. Para identificar esse componente é necessário compreender como é que o sistema Derby pode ser arrancado e encerrado. Conforme referido na secção anterior, o Derby arranca automaticamente quando a sua biblioteca JDBC embebida é carregada pelo gestor de bibliotecas do Java (*DriverManager*). Analisando todo o processamento realizado quando a biblioteca é carregada é possível verificar que esta é responsável por arrancar o monitor do Derby invocando:

```
Monitor.startMonitor(bootProperties, logging);
```

Este método estático será então responsável por arrancar uma instância de uma concretização do monitor do Derby. Esta é realizada por uma classe abstracta denominada *BaseMonitor*, que concretiza o método *runWithState* para arranque do sistema, e o método *shutdown* para o encerramento. Verificando-se que nenhum dos métodos é redefinido por uma classe que extenda *BaseMonitor*, os eventos de arranque e encerramento poderão ser capturados nos métodos *runWithState* e *shutdown*, respectivamente.

No método *runWithState* são carregadas todas as definições do sistema e arrancados todos os serviços persistentes, como o serviço de reflexão se configurado para isso. O código que captura o estado "iniciando" do SGBD foi definido logo após o arranque dos serviços persistentes. Nesse código são usadas as funcionalidades do Derby para obter referência para o módulo primário do serviço de reflexão, caso tenha sido arrancado, que, conforme referido na Secção 5.3.1, permite obter novas instâncias de cada um dos restantes módulos do serviço de reflexão. Através do módulo primário é então obtida uma nova instância do módulo de SGBD e utilizado o seu método *stateChange* para notificar o estado "iniciando". Antes do método *runWithState* retornar é invocado o método *stateChange* do módulo de SGBD indicando o estado "operacional" ou "parado", consoante o sucesso ou insucesso do arranque do monitor.

No início do método *shutdown* é invocado o método *stateChange* do módulo de SGBD indicando o estado "encerrando".

Neste módulo foram concretizadas as funcionalidades que permitem suportar a comunicação entre o núcleo do Derby e os módulos cliente do Reflector. Ou seja, as alterações de estado, capturadas ao nível do núcleo do Derby, são recebidas pelo Reflector e entregues aos módulos cliente. Em resposta a certos eventos os clientes poderão cancelar o processamento no SGBD.

O objecto de anexo é armazenado como um atributo do módulo, e pode ser alterado pelos módulos cliente sempre que desejado. Para além disso, neste módulo é ainda fornecido acesso à meta informação do sistema. Esta meta informação é também definida na interface *D*atabaseMetaData definida pelo JDBC. Assim, foram reutilizadas as funcionalidades do Derby que permitem aceder a essa informação, de forma a suportar a sua consulta por parte dos módulos cliente do Reflector. Sempre que é realizada uma consulta de meta informação do sistema, o módulo do Reflector reencaminha o pedido para o Derby, usando as suas funcionalidades definidas em *EmbedDatabaseMetaData*.

5.3.3 Concretização e suporte do módulo de base de dados:

De forma a suportar as funcionalidades do contexto de base de dados do Reflector, apresentadas na Secção 4.5.2, é necessário identificar e extender o componente do Derby que lhe corresponde, definindo o modo de pânico, capturando os eventos de arranque e encerramento, e permitindo que o processamento seja cancelado no momento do arranque.

O Derby define uma interface denominada *Database*, localizada em *org.apache.derby.database*, que fornece controlo sobre uma base de dados, nomeadamente sobre os seus dados e sobre os ficheiros onde esses estão armazenados. A interface desse módulo é extendida internamente pela interface *org.apache.derby.iapi.db.Database* com operações que não estão disponíveis para os utilizadores. Apenas existe uma concretização destas interfaces, denominada *BasicDatabase*, que concretiza os métodos *boot* e *stop*, para arrancar e encerrar a base de dados, respectivamente. Esta classe corresponde ao contexto de base de dados

do Reflector.

Tal como acontece no componente de SGBD, os métodos *boot* e *stop* foram extendidos de forma a informar o reflector sobre as alterações de estado da base de dados. Por outro lado, deve ser possível cancelar o processamento no estado "arrancando". O método *boot* define o lançamento de uma excepção do tipo *StandardException* em caso de erro. Para cancelar o arranque da base de dados é lançada uma excepção desse tipo que indica que o processamento foi deliberadamente cancelado pelo Reflector. Desta forma, todos os mecanismos de gestão do Derby são desencadeados garantindo que o cancelamento não gera incoerências no sistema.

Por outro lado, a extensão deste componente deve suportar o modo de pânico de uma base de dados. Este modo indica que a base de dados está incoerente e não pode ser acedida pelos utilizadores até que todos os conflitos sejam manual ou automaticamente resolvidos. A forma menos intrusiva de suportar este modo consiste em reutilizar os procedimentos fornecidos pelo Derby para bloquear e desbloquear o funcionamento de uma base de dados:

```
CALL SYSCS_UTIL.SYSCS_FREEZE_DATABASE()
CALL SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()
```

Deste modo, quando o Reflector pretende colocar uma base de dados no modo de pânico, é usado o procedimento *freeze* do Derby que bloqueia o acesso dos utilizadores à mesma. Uma vez resolvidos os conflitos, é usado o método *unfreeze* que coloca a base de dados novamente operacional.

Neste módulo foram concretizadas as funcionalidades que permitem suportar a comunicação entre o núcleo do Derby e os módulos cliente do Reflector. Ou seja, as alterações de estado, capturadas ao nível do núcleo do Derby, são recebidas pelo Reflector e entregues aos módulos cliente. Em resposta a certos eventos os clientes poderão cancelar o processamento na base de dados.

Tal como acontece no módulo de SGBD, os meta dados fornecidos por este módulo baseiam-se na reutilização de funcionalidades já suportadas pelo Derby. Em particular, os métodos *isReadOnly* e *getURL* são também definidos pela interface *D*atabaseMetaData do JDBC, e concretizados no Derby na classe *EmbedDatabaseMetaData*.

O modo de pânico da base de dados é gerido através da classe *BasicDatabase* que foi extendida de forma a suportar essa funcionalidade. O objecto de anexo é armazenado como um atributo do módulo, e pode ser alterado pelos módulos cliente sempre que desejado. Para além disso, é também suportada a funcionalidade de gestão da versão da base de dados, guardando o número de versão atribuído e fornecendo métodos que permitam a sua alteração.

5.3.4 Concretização e suporte do módulo de transacção:

De forma a suportar as funcionalidades do contexto de transacção do Reflector, apresentadas na Secção 4.5.2, é necessário identificar e extender o componente do Derby que lhe

corresponde capturando os eventos de alteração de estado, permitindo que o processamento seja cancelado, e permitindo que seja atribuído o modo de isolamento da transacção.

O Derby define uma interface denominada *Transaction*, localizada em *org.apache.derby.iapi.store.raw*, que representa uma transacção, fornecendo métodos para a sua gestão que permitem, por exemplo, confirmar ou abortar a sua execução. Esta interface é parcialmente concretizada por uma classe abstracta denominada *RawTransaction*. Esta classe já implementa métodos que permitem que observadores externos sejam notificados quando a transacção é confirmada, abortada, ou desfeita até à última salvaguarda. No entanto, é ainda uma classe de alto nível que não permite ter acesso directo a todas as operações e mudanças de estado de uma transacção. A classe que a extende, denominada *Xact*, é que concretiza todos os estados do ciclo de vida de uma transacção, tal como definidos pelo Reflector.

Na classe *Xact* os métodos *setActiveState*, *getActiveStateTxIdString*, *setUpdateState*, *commit*, *abort*, e *close* foram identificados como os pontos de mudança de estado. Os dois primeiros métodos são relativos ao início de uma transacção, nomeadamente a passagem do estado "iniciando" ao estado "activa". O método *setUpdateState* é responsável pela passagem do estado "activa" ao estado "actualizando". O método *commit* é responsável por preparar e confirmar uma transacção, notificando os estados "preparando", "preparada", "confirmando", "confirmada". O método *abort* é responsável por cancelar uma transacção, passando-a ao estado "cancelando" e "cancelada". Por fim, o método *close* termina a transacção, colocando-a no estado "terminada".

Foi necessário permitir que o processamento fosse cancelado nos estados *iniciando*, *preparando* e *confirmando*. Os métodos onde esses estados são atribuídos são *setActiveState*, *getActiveStateTxIdString*, e *commit*. Tal como acontece na extensão do componente de base de dados, os métodos *setActiveState* e *commit* permitem que uma excepção seja levantada cancelando o processamento. No entanto, o método *setActiveState* não permite o lançamento de excepções. Neste caso, quando o Reflector deseja cancelar o processamento, a excepção é armazenada e lançada na próxima alteração de estado que permita o lançamento de excepções. Neste caso é garantido que a transacção nunca irá confirmar, uma vez que o método *commit* permite o lançamento de excepções cancelando o processamento antes confirmação.

Neste módulo foram concretizadas as funcionalidades que permitem suportar a comunicação entre o núcleo do Derby e os módulos cliente do Reflector. Ou seja, as alterações de estado, capturadas ao nível do núcleo do Derby, são recebidas pelo Reflector e entregues aos módulos cliente. Em resposta a certos eventos os clientes poderão cancelar o processamento na base de dados.

O objecto de anexo é armazenado como um atributo do módulo, e pode ser alterado pelos módulos cliente sempre que desejado. O modo de isolamento da transacção pode ser consultado e alterado através da classe *Xact* que foi extendida de forma a suportar essa funcionalidade. A consulta da versão da transacção também é suportada com recurso à

classe *Xact*.

5.3.5 Concretização e suporte do módulo de análise:

De forma a suportar o módulo de análise é necessário extender o componente de processamento relacional do Derby, capturando o início e o fim da fase de análise. Para além disso, é também necessário permitir que o processamento seja cancelado no início da fase de armazenamento lógico.

A classe *GenericStatement* corresponde a um pedido SQL em formato de texto e concretiza um método *prepMinion*, responsável por realizar a análise do pedido. Esse método foi extendido de forma a notificar o Reflector do início e do fim da fase de análise, permitindo que o processamento seja cancelado no estado "iniciando" através do lançamento de uma excepção.

Neste módulo foram concretizadas as funcionalidades que permitem suportar a comunicação entre o núcleo do Derby e os módulos cliente do Reflector. Ou seja, as alterações de estado, capturadas ao nível do núcleo do Derby, são recebidas pelo Reflector e entregues aos módulos cliente. Em resposta a certos eventos os clientes poderão cancelar o processamento na base de dados. Por fim, foi ainda concretizada a funcionalidade que permite obter o contexto de execução que está associado ao processamento.

5.3.6 Concretização e suporte do módulo de armazenamento lógico:

Para suportar o módulo de armazenamento lógico é necessário identificar os pontos onde são lidos, inseridos, actualizados, e removidos dados, de forma a obter os valores do conjunto de escrita e leitura a reflectir pelo módulo de armazenamento lógico. Dada a sua semelhança, apenas foram realizadas as extensões necessárias de forma a obter o conjunto de escritas.

No Derby, o código a ser processado pela fase de execução é gerado dinamicamente para memória num vector de código máquina. A classe gerada extende uma classe interna denominada *BaseActivation* que, por sua vez, implementa uma interface denominada *Activation*. Esta interface define métodos de armazenamento de estado, definição de parâmetros, e suporte à execução. Cada expressão SQL é definida num método da classe gerada. Durante a execução é gerada uma árvore interna de conjuntos de resultados para cada uma das operações SQL, como por exemplo, inserções, remoções, ou actualizações. É nessas classes que representam cada um dos conjuntos de resultado que os eventos serão capturados para o Reflector.

A interface *Activation* foi extendida com métodos que permitem configurar se o conjunto de escritas deve ser capturado, e, nesse caso, quais as colunas a capturar. Essa configuração é consultada nas classes *InsertResultSet*, *DeleteResultSet*, *UpdateResultSet* que foram alteradas de forma a recolher os valores inseridos, removidos, e actualizados nas colunas desejadas, respectivamente.

Para além disso, é também necessário permitir que o processamento seja cancelado no início da fase de armazenamento lógico. Esta funcionalidade foi suportada através do lançamento de uma excepção.

Neste módulo foram concretizadas as funcionalidades que permitem suportar a comunicação entre o núcleo do Derby e os módulos cliente do Reflector. Ou seja, as alterações de estado, capturadas ao nível do núcleo do Derby, são recebidas pelo Reflector e entregues aos módulos cliente. Em resposta a certos eventos os clientes poderão cancelar o processamento na base de dados. Por fim, foi ainda concretizada a funcionalidade que permite obter o contexto de execução que está associado ao processamento.

5.4 Sumário e discussão

Nesta secção foi discutida a concretização de parte do Reflector no SGBDR Apache Derby. Em primeiro lugar foram apresentados os desafios da concretização do Reflector num SGBDR genérico, quer em termos de considerações de desenho, quer em termos de componentes do SGBDR a extender, quer ainda em termos das necessidades da concretização de cada um dos módulos do Reflector. Depois, foi apresentado o SGBDR Apache Derby, focando os aspectos mais mais relevantes da sua concretização, que tomaram um papel determinante nas decisões tomadas a nível da concretização do Reflector. Por fim, os módulos do Reflector foram organizados em grupos que requerem a resolução de uma mesma classe de problemas, e apresentou-se a solução encontrada para a concretização de um módulo de cada grupo no Derby.

Uma vez apresentado o Reflector e a sua concretização no SGBDR Apache Derby, o capítulo seguinte realiza uma avaliação do impacto que o Reflector introduz no desempenho do Derby.

Capítulo 6

Avaliação

Neste capítulo avalia-se o impacto que o Reflector introduz no desempenho de um SGBD relacional que o suporte nativamente. A avaliação consiste em comparar o desempenho do SGBDR Derby original e com a extensão de suporte ao Reflector, enquanto processam cargas semelhantes num mesmo cenário de execução. Para isso, será utilizada uma plataforma de testes padrão (*benchmark*), denominada TPC-W, que gera carga normalizada para bases de dados num ambiente de publicações e vendas na Web.

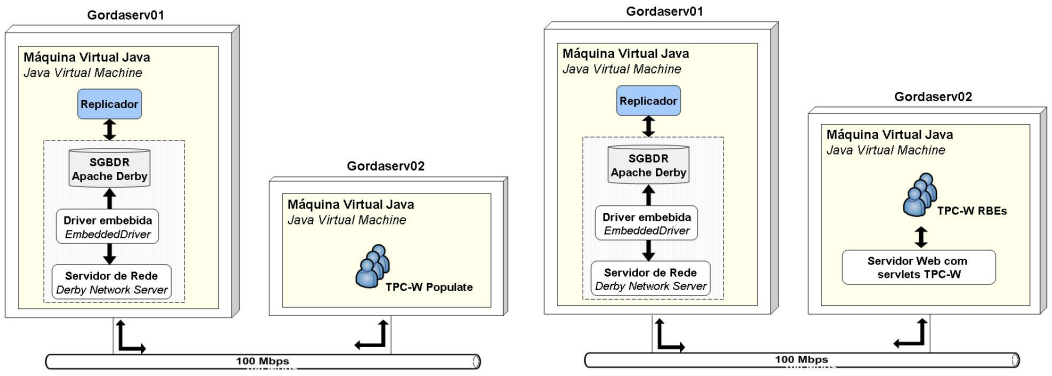
Na Secção 6.1 será definido o conceito de plataforma de testes padrão, e será apresentada a plataforma TPC-W. Na Secção 6.2 será descrita a plataforma de testes a utilizar. Na Secção 6.3 serão apresentados e discutidos os resultados obtidos, através de gráficos comparativos. Por fim, na Secção 6.4 é realizado um breve sumário do capítulo.

6.1 Ferramenta TPC-W

À medida que os sistemas computacionais se tornam mais complexos, a análise de desempenho torna-se uma actividade cada vez mais relevante e indispensável. No âmbito dos sistemas informáticos, uma ferramenta para execução de testes padrão (*benchmark*) permite realizar um conjunto de testes projectados para comparar o desempenho de um sistema computacional em relação a outros, submetendo-os a uma carga de trabalho semelhante. Uma carga de trabalho corresponde ao conjunto de tarefas, e respectivos consumos de recursos, submetido a um determinado sistema para execução durante um determinado intervalo de tempo. Por sua vez, uma tarefa é a unidade de execução do sistema computacional. Por exemplo, num cenário de bases de dados, uma tarefa pode corresponder a um pedido SQL.

Actualmente existe uma variedade de plataformas para execução de testes padrão. A *Transaction Processing Performance Council* (TPC[11]) é uma organização criada com o objectivo de estabelecer critérios e padrões de referência de desempenho do processamento de transacções numa bases de dados. Os membros da organização incluem as principais empresas de bases de dados e fornecedores de sistemas de hardware do mercado.

Uma das normas definidas pela TPC denomina-se TPC-W. Esta norma especifica uma



(a) *Populate*. (b) *RBE*.

Figura 6.1: Ambientes de testes.

carga de trabalho que simula um serviço de comércio na Web para venda de livros. O sítio do serviço na Web é composto por 14 páginas com diferentes probabilidades de acesso. A carga de trabalho é gerada através de RBE (Remote Browser Emulators) e executada num ambiente totalmente controlado. Os RBE emulam os utilizadores, simulando as cargas HTTP resultantes das suas operações, como gestão de carrinho de compras, ou pesquisa e pagamento de livros. Por outro lado, são ainda emuladas as operações de administração do sistema.

Esta ferramenta simula três perfis de interacções que variam no rácio entre pesquisas e compras: perfil de compras com 80% de pesquisas e 20% de compras, perfil de pesquisas com 95% de pesquisas e 5% de compras, e perfil de encomendas com 50% de pesquisas e 50% de compras. As operações de pesquisa correspondem operações de leitura, enquanto as operações de compra constituem escritas na base de dados. O primeiro perfil projecta a utilização padrão de um sistema de comércio electrónico na Web. O segundo perfil sobrecarrega os componentes mais alto nível do sistema, com operações de carregamento de páginas e manutenção de cache. O último perfil sobrecarrega a base de dados do sistema, com operações de compra e manutenção dos livros em armazem.

A secção seguinte apresenta o ambiente de testes utilizado, detalhando os processos que executam em cada máquina, as características de cada uma, e as características da rede.

6.2 Ambiente de testes

A avaliação comparativa consistiu na realização de dois testes distintos. O primeiro consistiu em criar e popular a base de dados. Para isso, foi utilizada a aplicação *Populate* fornecida pelo TPC-W, para criar uma base de dados com o seu esquema relacional, populando-a com dados aleatórios até atingir um tamanho configurável. O segundo consistiu em simular a utilização do sistema com uma carga padrão de vendas na Web. Para

Característica	Gordaserv01	Gordaserv02
Processador	2x AMD Opteron 2.2 Ghz 64 Bit	P4 2.8 Ghz
Memória RAM	4 Gb	2 Gb
Disco	2x Sata 250 GB em RAID1	2x Sata 250 GB em RAID1
Sistema operativo	Linux v. 2.4.21	Linux v. 2.4.21

Tabela 6.1: Características das máquinas utilizadas nos testes

isso, foram utilizados os RBE fornecidos pelo TPC-W, para gerar tráfego sobre a base de dados.

Ambos os testes foram aplicados em quatro cenários, de forma a comparar o desempenho do Derby em cada um deles. No primeiro cenário, é utilizado o Derby original. No segundo cenário, é utilizado o Derby com suporte para o Reflector, mas sem clientes. No terceiro cenário, é utilizado o Derby com suporte para o Reflector, e um replicador como cliente em modo não bloqueante (ou assíncrono). No quarto cenário, é utilizado o Derby com suporte para o Reflector, e um replicador como cliente em modo bloqueante (ou síncrono).

O replicador utilizado simula a execução de um protocolo de replicação passiva, utilizando os módulos de transacção e armazenamento lógico do Reflector, em modo bloqueante e não bloqueante. No entanto, de forma a isolar convenientemente a variável a avaliar, nomeadamente o impacto que o Reflector introduz no Derby, optou-se por concretizar um replicador que introduza o mínimo de atrasos possível no processamento do SGBD. Desta forma, o replicador não realiza qualquer computação para além de se registar como cliente do Reflector, receber os seus eventos, e, no modo bloqueante, autorizar que o processamento prossiga. Em específico, o replicador regista-se como cliente do Reflector, para os módulos de transacção e armazenamento lógico, e aguarda a recepção de eventos. Os seus métodos de recepção de eventos (*handle[notificação]*), quando invocados pelo Reflector, armazenam a notificação em memória partilhada, acordam o replicador, e retornam de imediato. No modo não bloqueante, o processamento no SGBD prossegue assim que o método de notificação retorna. O replicador, ao acordar, consulta a notificação recebida e verifica em que modo está registada. Se estiver registada em modo bloqueante, autoriza que o processamento prossiga. No modo bloqueante, o processamento no SGBD prossegue assim que o replicador autoriza. Por fim, o replicador bloqueia novamente em espera de notificações.

Os ambientes de testes utilizados encontram-se esquematizados na Figura 6.1. Conforme ilustrado foram utilizadas duas máquinas, que denominaremos *gordaserv01* e *gordaserv02*, interligadas por uma rede local a 100Mbps. Na máquina *gordaserv01* o SGBDR Derby foi embebido no seu servidor de rede, estando disponível para os clientes através de JDBC. Na máquina *gordaserv02* foram executadas as duas aplicações do TPC-W: *Populate*,

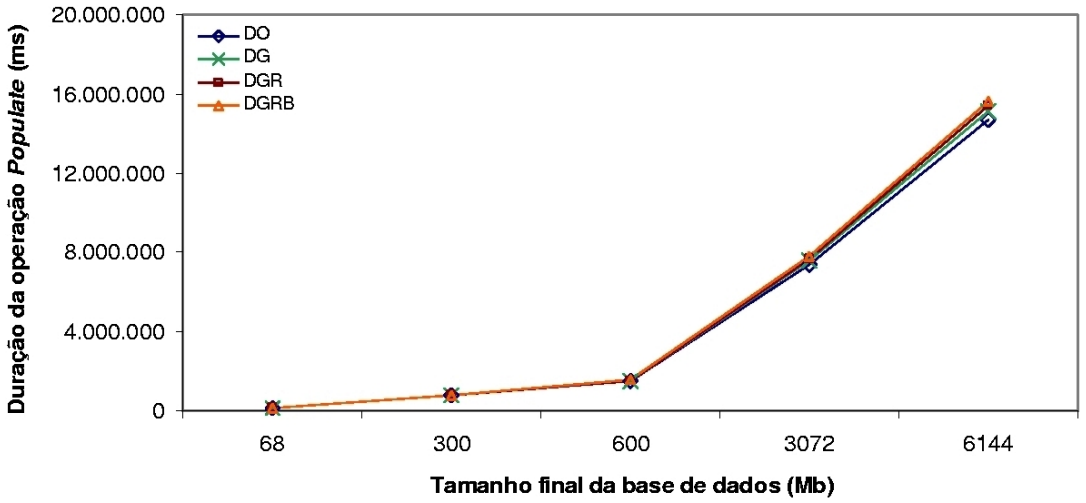


Figura 6.2: Resultados obtidos pela aplicação *Populate* do TPC-W

e RBE. Foi utilizada uma concretização do TPC-W distribuída pela Universidade de Wisconsin [16, 1]. A Tabela 6.1 resume as características de ambas as máquinas em termos de processador, memória RAM, disco, e sistema operativo.

6.3 Descrição dos resultados

Os testes realizados foram aplicados a quatro configurações do Derby: Derby original (DO), Derby com extensão Gorda mas sem replicador (DG), Derby com extensão Gorda e replicador em modo não bloqueante (DGR), e Derby com extensão Gorda e replicador em modo bloqueante (DGRB).

A avaliação de desempenho realizada através da aplicação *Populate* do TPC-W teve como objectivo calcular o tempo total da operação de *Populate* nas diferentes configurações do Derby. Neste teste fez-se variar o tamanho final pretendido para a base de dados, que se configurou em função do total de registos de cliente (variável *cli*), e do total de livros existentes no sistema (variável *items*). Foram usadas cinco configurações distintas: 10 clientes e 10000 livros, 50 clientes e 50000 livros, 100 clientes e 100000 livros, 500 clientes e 500000, e 1000 clientes e 1000000 livros. Estas originaram bases de dados de 68 Mb, 300 Mb, 600 Mb, 3072 Mb, e 6144 Mb, respectivamente.

O gráfico da Figura 6.2 apresenta o tempo total do *Populate* nas diferentes con-

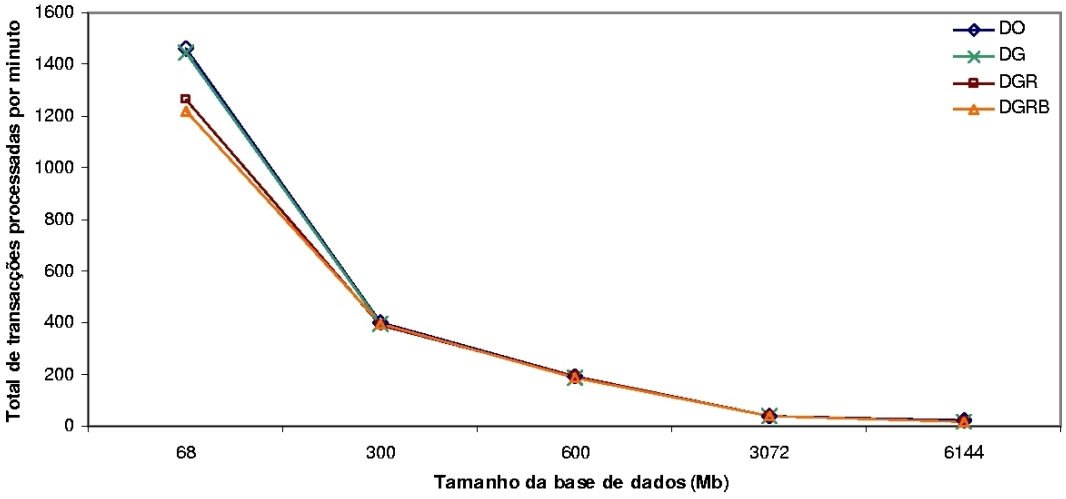


Figura 6.3: Resultados obtidos pela aplicação RBE do TPC-W

figurações do Derby, em função do tamanho final pretendido para a base de dados. Pela análise do gráfico concluímos que o tempo necessário para popular a base de dados é extremamente semelhante ao do Derby original, pelo que os resultados são bastante satisfatórios.

A avaliação de desempenho realizada através da aplicação *RBE* do TPC-W teve como objectivo calcular o total de transacções processadas por minuto (TPM) pelas diferentes configurações do Derby. Os testes decorreram em períodos de 25 minutos simulando 10 clientes a aceder concorrentemente à base de dados. Dos três padrões de tráfego gerados pela aplicação RBE, referidos na secção anterior, foi utilizado o perfil de encomendas, por ser o que apresenta a maior taxa de pedidos de escrita sobre a base de dados (cerca de 50%). Neste teste fez-se variar o tamanho da base de dados utilizada.

O gráfico da Figura 6.3 apresenta o total de TPM nas diferentes configurações do Derby, em função do tamanho da base de dados. Pela análise do gráfico é possível concluir que, quanto mais pequena é a base de dados, mais notória é a perda de desempenho originada pelo Reflector. Este resultado justifica-se porque, quanto mais pequenas são as bases de dados, menos tempo de acesso é necessário para uma mesma operação, mais operações podem ser realizadas por minuto, e, consequentemente, mais notificações são geradas pelo Reflector. No teste do *Populate*, a diferença não foi notória uma vez que o total de

operações realizadas foi constante, variando apenas o total de dados inseridos em cada uma delas. Consideramos que a perda de desempenho no total de transacções processadas por minuto é aceitável para sistemas de replicação de bases de dados que usem o Derby como SGBD

6.4 Sumário e discussão

Nesta secção foi analisado o impacto que o suporte do Reflector origina no desempenho de um SGBRD que o suporte de forma nativa. Para isso, foi utilizada a plataforma de testes padrão denominada TPC-W de forma a popular e gerar tráfego sobre a base de dados de um sítio na Web para venda de livros. O primeiro teste realizado consistiu na utilização da aplicação *Populate*, fornecida pelo TPC-W, para popular bases de dados até um tamanho configurável. Neste teste foi avaliado tempo total da operação nas diferentes configurações do Derby, em função do tamanho final pretendido para a base de dados. O segundo teste consistiu na utilização da aplicação RBE, para simular os acessos concorrentes de 10 clientes à base de dados, com uma taxa de 50% de escritas. Neste teste foi avaliado o total de transacções processadas por minuto. Os resultados mostram que o overhead causado pelo suporte adicional do Reflector é negligenciavel.

Capítulo 7

Conclusões e Trabalho Futuro

No contexto dos sistemas de informação, os sistemas de gestão de bases de dados (SGBD) assumem-se como uma ferramenta fundamental para gestão, selecção e tratamento dos dados em tempo útil. As técnicas que suportam uma eficiente replicação das bases de dados são extremamente importantes uma vez que aumentam a disponibilidade dos dados na presença de faltas, permitindo explorar a localidade dos mesmos e dispersar a carga entre as várias réplicas.

Actualmente, existe uma vasta gama de protocolos de replicação de bases de dados. No entanto, a falta de suporte nativo dos SGBD para replicação por terceiros (*third-party replication*) requer que esses sistemas de replicação modifiquem o núcleo do sistema ou desenvolvam soluções externas, como um cliente normal ou um invólucro middleware, de forma a poderem realizar a replicação. As soluções de replicação concretizadas de forma nativa são as que oferecem um melhor desempenho, no entanto, não são compatíveis entre si nem transportáveis para outros tipos de bases de dados. Por sua vez, as soluções de replicação externas ao SGBD oferecem uma maior modularidade, permitindo que o sistema replicado opere sobre sistemas de gestão de bases de dados heterogéneos. No entanto, por actuarem fora do núcleo do SGBD, estas aproximações têm muitas vezes de repetir passos realizados pela base de dados, o que geralmente se traduz numa quebra acentuada do desempenho.

Neste trabalho foi apresentada a Interface Gorda de Reflexão, ou simplesmente Reflector, que combina a modularidade e a portabilidade das aproximações externas ao SGBD, com o desempenho das aproximações nativas, de forma a que várias estratégias de replicação possam ser concretizadas sobre qualquer base de dados compatível. Em particular, o Reflector fornece os mecanismos necessários para os sistemas de replicação observarem, interceptarem, e modificarem eficientemente o processamento de transacções, independentemente do SGBDR usado.

De forma a demonstrar que as opções de desenho do Reflector foram adequadas, e permitem uma extensão pouco intrusiva do núcleo do SGBDR, foram discutidos os desafios da sua concretização num SGBDR genérico, quer em termos de considerações de desenho, quer em termos de componentes do SGBDR a extender, quer ainda em termos das neces-

sidades da concretização de cada um dos módulos do Reflector. Depois, os módulos do Reflector foram organizados em grupos que requerem a resolução de uma mesma classe de problemas, e apresentou-se a solução encontrada para a concretização de um módulo de cada grupo no SGBDR Apache Derby. Desta forma, garantiu-se que foram abordadas as diferentes classes de problemas que se podem levantar durante a concretização do Reflector, pelo que as funcionalidades não suportadas poderão facilmente ser concretizadas reutilizando as soluções adoptadas.

Por fim, avaliou-se o impacto que o Reflector introduz no desempenho de um SGBD relacional que o suporte nativamente. A avaliação consistiu em comparar o desempenho do SGBDR Apache Derby original e com a extensão Gorda de suporte ao Reflector, enquanto processavam cargas semelhantes num mesmo cenário de execução. Para isso, foi utilizada uma plataforma de testes padrão (*benchmark*), denominada TPC-W, que gera carga normalizada para bases de dados num ambiente de publicações e vendas na Web. Pela análise dos resultados obtidos, foi possível verificar que o impacto introduzido pelo Reflector é satisfatório, mesmo em cenários de replicação que se baseiem no modo de reflexão bloqueante.

Como trabalho futuro, pretende-se adicionar funcionalidades de recuperação e injecção de estado ao Reflector. Para além disso, pretende-se concretizar todas as funcionalidades do Reflector ainda não suportadas no Derby. Uma vez realizados os dois objectivos anteriores, serão realizados testes do Reflector num ambiente de replicação real, com todos os módulos identificados pela arquitectura Gorda.

Bibliografia

[1] Concretização da norma TPC-W pela Universidade de Wisconsin.
http://www.ece.wisc.edu/~pharm/tpcw.shtml.

[2] Manual de AWT da Java Sun.
http://java.sun.com/products/jdk/awt/.

[3] Manual de JCE da Java Sun.
http://java.sun.com/products/jce/.

[4] Manual de JDBC da Java Sun.
http://java.sun.com/products/jdbc/.

[5] Manual de ODBC da Microsoft.
http://www.microsoft.com/data/odbc/.

[6] Página da American National Standards Institute.
http://www.ansi.org/.

[7] Página da International Organization for Standardization.
http://www.iso.org/.

[8] Página do IBM WebSphere.
http://www-306.ibm.com/software/websphere/.

[9] Página do Projecto Apache Derby.
http://db.apache.org/derby/.

[10] Página do Projecto GORDA: Open Replication of Databases.
http://gorda.di.uminho.pt/.

[11] Página do Transaction Processing Performance Council.
http://www.tpc.org/.

[12] Derby developer's guide. 2006.
http://db.apache.org/derby/docs/dev/devguide/.

[13] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall / Sun Microsystems Press, 2001.

[14] N. Arora. Oracle Streams for near real time asynchronous replication. In *Proc. VLDB Ws. Design, Implementation, and Deployment of Database Replication*, 2005.

[15] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[16] Todd Bezenek, Trey Cain, Ross Dickson, Timothy Heil, Milo Martin, Collin McCurdy, Ravi Rajwar, Eric Weglarz, Craig Zilles, and Mikko Lipasti. Characterizing a java implementation of tpcw. 3rd Workshop On Computer Architecture Evaluation Using Commercial Workloads (CAECW), January 2000.
http://www.ece.wisc.edu/~pharm/tpcw/CAECW2000.pdf.

[17] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, 1998.

[18] Eric J. Braude. *Software Engineering - An Object-Oriented Perspective*. Wiley, 2001.

[19] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. In *ACM Computing Surveys*, 2001.

[20] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database Replication Using Generalized Snapshot Isolation. In *IEEE SRDS*, 2005.

[21] Gartner. The growing role of events in enterprise applications. 2003. ID Number: AV-20-3900.

[22] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD*, 1996.

[23] Gregor Hohpe. Programming without a call stack: Event-driven architectures. 2006.
http://www.enterpriseintegrationpatterns.com/docs/EDA.pdf.

[24] A. Correia Jr., J. Pereira, L. Rodrigues, N. Carvalho, R. Oliveira, and S. Guedes. Gorda: An open architecture for database replication. 2006.

[25] G. Kiczales. Towards a new model of abstraction in the engineering of software. In *Proc. Intl. Ws. New Models for Software Architecture*, 1992.

[26] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13:10–11, 1996.

[27] Craig Larman. *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design*. Prentice Hall PTR, 1997.

[28] Leonid Libkin. Expressive power of SQL. *Lecture Notes in Computer Science*, volume 1973, 2001.
http://citeseer.ist.psu.edu/407307.html.

[29] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jimenez-Peris. Middleware based data replication providing snapshot isolation. 2005.

[30] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley Professional, 1999.

[31] Jean-Louis Maréchaux. Combining service-oriented architecture and event-driven architecture using an enterprise service bus. 2006.

[32] P. Martin, W. Powley, and D. Benoit. Using reflection to introduce self-tuning technology into dbmss. In *Proc. Intl. Database Engineering and Applications Symp. (IDEAS'04)*, 2004.

[33] Mauro Nunes and Henrique O'Neil. *Fundamental de UML*. FCA, 2001.

[34] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach, 2002.

[35] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill International Editions, 2000.

[36] António Sousa, Fernando Pedone, Francisco Moura, and Rui Oliveira. Partial replication in the database state machine. In *n Proc. of the IEEE International Symposium on Network Computing and Applications (NCA 2001)*, pages 298–309. IEEE CS, October 2001.

[37] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE'05)*, 2005.

[38] V. Zuikeviciute and F. Pedone. Revisiting the database state machine approach. In *DIDDR'05 in conjunction with VLDB Conference*, 2005.

Índice Remissivo

Glossário

Português	**Inglês**
Biblioteca	Driver
Confirmar uma Transacção	Commit a transaction
Conjunto de escrita	Write-Set
Conjunto de leitura	Read-Set
Conjunto de resultados	Result-Set
Cópia de segurança	Backup
Ferramenta para a execução de testes padrão	Benchmark framework
Fio de execução	Thread
Gatilho	Trigger
Histórico	Log
Interbloqueio	Deadlock
Invólucro	Wrapper
Mecanismo de reciclagem de memória do Java	Garbage Collector
Modelo de Cópias Passivas	Primary-Backup Model
Modelo de Cópias Activas	One-Copy Equivalence, Update Everywhere
Pacote	Package
Processamento em Conduta	Pipelinning
Replicação Nativa	In-Core Replication
Replicação por Terceiros	Third-Party Replication
Trinco	Lock

Apêndices

Apêndice A

Extensões realizadas directamente no código fonte do Derby

Pacote	**org.apache.derby.iapi**
Ficheiro	**Alteração**
build.xml	Extendido de forma a compilar as interfaces do Reflector

Pacote	**org.apache.derby.iapi.reference**
Ficheiro	**Alteração**
Property.java	Define as propriedades de configuração do Reflector

Pacote	**org.apache.derby.iapi.services.monitor**
Ficheiro	**Alteração**
PersistentService.java	Define uma propriedade que armazena o nome de um serviço

Pacote	**org.apache.derby.iapi.sql**
Ficheiro	**Alteração**
Activation.java	Define métodos extra para definição das propriedades e do modo de captura do conjunto de escrita
ResultSet.java	Define um método extra para obter o conjunto de escrita correspondente

Pacote	**org.apache.derby.iapi.sql.conn**
Ficheiro	**Alteração**
LanguageConnectionContext.java	Define um método extra para atribuição e obtenção de referência para o serviço do Reflector
LanguageConnectionFactory.java	Define um método extra para obtenção de referência para a classe fábrica do serviço Reflector

Pacote	**org.apache.derby.impl**
Ficheiro	**Alteração**
build.xml	Extendido de forma a compilar as classes do Reflector

Pacote	**org.apache.derby.impl.db**
Ficheiro	**Alteração**
BasicDatabase.java	Extendido de forma a reflectir o contexto de base de dados

Pacote	**org.apache.derby.impl.jdbc**
Ficheiro	**Alteração**
EmbedConnection.java	Adiciona o nome da base de dados nas propriedades de arranque da mesma. Esta propriedade foi definida em Property.java

Pacote	**org.apache.derby.impl.sql**
Ficheiro	**Alteração**
build.xml	Passa a compilar este módulo com a versão 1.4 do Java

Pacote	**org.apache.derby.impl.sql.conn**
Ficheiro	**Alteração**
GenericLanguageConnectionContext.java	Concretiza os métodos extra definidos em LanguageConnectionContext.java, responsáveis pela atribuição e obtenção de referência para o serviço do Reflector
GenericLanguageConnectionFactory.java	Concretiza os métodos extra definidos em LanguageConnectionFactory.java, responsável pela obtenção de referência para a classe fábrica do serviço Reflector

Pacote	**org.apache.derby.impl.sql**
Ficheiro	**Alteração**
GenericActivationHolder.java	Concretiza os métodos extra definidos em Activation.java, responsáveis pela definição das propriedades e do modo de captura do conjunto de escrita
GenericPreparedStatement.java	Extendido de forma a reflectir a fase de processamento de armazenamento lógico
GenericStatement.java	Extendido de forma a reflectir a fase de processamento de análise

Pacote	**org.apache.derby.impl.sql.execute**
Ficheiro	**Alteração**
BaseActivation.java	Concretiza os métodos extra definidos em Activation.java, responsáveis pela definição das propriedades e do modo de captura do conjunto de escrita
BasicNoPutResultSet.java	Concretiza o método extra adicionado a ResultSet.java, para obtenção do conjunto de escrita
DeleteResultSet.java	Preenche o conjunto de escrita com os dados removidos
InsertResultSet.java	Preenche o conjunto de leitura com os dados inseridos

NoRowsResultSetImpl.java	De forma a ser visível pelo Reflector, esta classe passou de protegida a pública, para além disso, concretiza o método extra adicionado a ResultSet.java, para obtenção do conjunto de escrita
TemporaryRowHolderResultSet.java	Concretiza o método extra adicionado a ResultSet.java, para obtenção do conjunto de escrita
UpdateResultSet.java	Preenche o conjunto de escrita com os dados actualizados

Pacote	**org.apache.derby.impl.store.raw.xact**
Ficheiro	**Alteração**
Xact.java	Extendido de forma a reflectir o contexto de base de dados