Universidade do Minho
Escola de Engenharia

Adriana Manuela Ferreira Costa

Ferramenta de Gestão de Dados Históricos

Ferramenta de Gestão de Dados Históricos Adriana Manuela Ferreira Costa UMinho | 2013

outubro de 2013

Universidade do Minho
Escola de Engenharia

Adriana Manuela Ferreira Costa

Ferramenta de Gestão de Dados Históricos

Tese de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia de Comunicações

Trabalho efetuado sob a orientação do
Professor Doutor Filipe Meneses

outubro de 2013

Agradecimentos

A conclusão deste trabalho não seria possível sem a ajuda de algumas pessoas, a quem não posso deixar de agradecer.

Agradeço ao Professor Doutor Filipe Meneses por ter aceitado este desafio proposto pela PT Inovação. Agradeço-lhe pela sua disponibilidade e dedicação, bem como pelas sugestões e correções.

Agradeço ao Engenheiro Ricardo Melo por todo o empenho e esforço demonstrado ao longo do tempo em que estive na PT Inovação a desenvolver este trabalho.

Não posso deixar de agradecer à PT Inovação pela oportunidade.

Agradeço aos meus professores do curso de Mestrado Integrado em Engenharia de Comunicações, por todos os conhecimentos transmitidos. Em especial, agradeço à Professora Doutora Maria João Nicolau, pela enorme dedicação que teve durante estes anos perante o curso.

Aos meus pais e à minha irmã, agradeço principalmente toda a paciência que tiveram comigo durante os últimos anos e também pela motivação constante.

Ao Henrique agradeço por tudo.

Finalmente, agradeço a todos os meus amigos, pelo apoio e incentivo.

Resumo

Em algumas organizações, a quantidade de dados registados nas suas bases de dados é grande e continua a aumentar de forma linear ou exponencial ao longo do tempo. Face ao constante aumento do volume de dados, torna-se necessário assegurar o acesso rápido e ágil a estes, implementando novas formas de gestão.

A gestão de dados históricos é tipicamente uma tarefa que cada Sistema de Gestão de Base de Dados (SGBD) efetua de forma muito própria, otimizada para o seu motor, não existindo nenhuma norma adotada pelos vários SGBDs. Em particular, torna-se cada vez mais complexo, do ponto de vista aplicacional, gerir os dados históricos guardados pelas aplicações.

Neste trabalho é apresentada uma ferramenta que foi desenvolvida para permitir, face a um modelo numa arquitetura MVC *(Model View Controller)* implementado em Ruby on Rails, organizar a distribuição lógica dos dados para que, independentemente do SGBD usado, se possa de forma eficaz retirar os dados antigos do sistema e armazená-los de forma a serem facilmente recuperados. A ferramenta foi desenvolvida como uma biblioteca genérica, que pode ser utilizada por qualquer aplicação, e com o objetivo específico de permitir o particionamento dos dados.

Primeiro, foi realizada uma investigação sobre alguns SGBDs comerciais, como Oracle, MySQL, MS SQL Server e PostgreSQL. Posteriormente a pesquisa foi direcionada para o ORM utilizado pelo Ruby on Rails, o Active Record, de forma a perceber quais os métodos utilizados para implementar os métodos CRUD (Create-Read-Update-Delete). A fim de resolver os desafios do presente trabalho, foi desenhada e implementada uma biblioteca que, juntamente com o ORM Active Record, permite particionamento de tabelas. O trabalho realizado ao longo desta dissertação foi desenvolvido em ambiente empresarial, na Portugal Telecom Inovação. No final foram realizados e analisados alguns testes para aferir o desempenho da solução implementada.

Abstract

In some organisations, the amount of data stored in their databases is significantly large and continues to increase linearly or exponentially over time. In order to respond to the constant increase of data volume, it is necessary to ensure quick and agile access to the data, implementing new management methods.

The management of historical data is typically a task that every Database Management System (DBMS) does on its own way, optimized for its engine. However, from the application point of view, it becomes increasingly complex to manage the historical data stored by the applications.

This work presents a tool that was developed to allow, for a model in MVC architecture (Model View Controller) implemented with Ruby on Rails, to organize the logical distribution of data so that, independently of the DBMS used, it can be possible in an effective way, to remove the older data from the system and store it in an easily restorable way. The tool was developed as a generic library, which can be used by any application, and with the specific goal of implementing data partitioning.

Firstly, a study was performed based on some commercial DMBS, such as Oracle, MySQL, MS SQL Server and PostgreSQL. Afterwards, the research has been directed to the ORM used by Ruby on Rails, the Active Record, in order to determine which methods it uses to implement the CRUD (Create-Read-Update-Delete) methods. In order to solve the challenges of the presented work, a library was designed and implemented, which along with the Active Record ORM, allows table partitioning. The work done along this dissertation was developed in a business environment, at Portugal Telecom Inovação. In the end some tests were made and analyzed to assess the performance of the implemented solution.

Índice de conteúdo

Índice de figuras

Índice de tabelas

Acrónimos

Neste documento são utilizados vários acrónimos que representam abreviações de designações utilizadas. Os acrónimos adotados são:

ACID	*Atomicity, Consistency, Isolation, Durability*
AJAX	*Asynchronous Javascript and XML*
API	*Application Programming Interface*
COBOL	*Common Business Oriented Language*
CRUD	*Create, Read, Update, Delete*
DML	*Data Manipulation Language*
DRY	*Don't Repeat Yourself*
ERb	*Embedded Ruby*
GUI	*Graphical User Interface*
HTML	*HyperText Markup Language*
IBM	*International Business Machines Corporation*
IDC	*International Data Corporation*
IDE	*Integrated Development Environment*
ISO	*International Organization for Standardization*
JDBC	*Java Database Connectivity*
JSON	*JavaScript Object Notation*
MVC	*Model View Controller*
MVCC	*Multi-Version Concurrency Control*
MVP	*Model View Presenter*
MVVM	*Model View ViewModel*
NIST	*National Institute of Standards and Technologies*
ORM	*Object Relational Mapping*
REST	*Representational State Transfer*
RJS	*Ruby JavaScript*
SGBD	Sistema Gestor de Bases de Dados

SGBDR	Sistema Gestor de Bases de Dados Relacionais
SQL	*Structured Query Language*
XML	*Extensible Markup Language*
YAML	*YAML Ain't Markup Language*

1 Introdução

Neste capítulo é descrito o enquadramento do projeto de dissertação e é referida a motivação para a realização do mesmo. Também é descrita a finalidade da dissertação e os seus principais objetivos, assim como a abordagem seguida para resolução dos mesmos. Conclui-se o capítulo com a apresentação da estrutura do presente documento.

1.1 Enquadramento e motivação

Este trabalho de dissertação de mestrado está enquadrado em âmbito empresarial, na empresa de telecomunicações Portugal Telecom Inovação, e tem como objetivo o desenvolvimento de uma ferramenta de gestão de dados históricos.

Em algumas organizações a quantidade de dados registados nas suas bases de dados é grande e continua a aumentar de forma exponencial ao longo do tempo. Face ao constante aumento do volume de dados, torna-se necessário assegurar o acesso rápido e ágil a estes, implementando novas formas de gestão. Em particular, torna-se cada vez mais complexo gerir os dados históricos guardados pelas aplicações.

A gestão de dados históricos é tipicamente uma tarefa que cada Sistema de Gestão de Base de Dados (SGBD) faz à sua maneira, otimizada para o seu motor. Existem variados SGBDs comerciais, como o *Oracle*, *Postgresql* e o *MySQL* e alguns destes SGBDs disponibilizam técnicas e ferramentas específicas para fazerem a gestão de grandes quantidades de dados e para fazerem a gestão de dados históricos. Noutros casos, a gestão de dados históricos é feita diretamente pelas aplicações ou então estes dados são tratados sem diferenciação face a todos os restantes dados mantidos pelos SGBDs.

As aplicações que o grupo de *Mobile Money* da PT Inovação está a desenhar e implementar atualmente têm como um dos requisitos serem completamente independentes do SGBD utilizado, para que a escolha do SGBD seja apenas uma decisão a tomar em cada solução, consoante os requisitos de infraestrutura de cada um delas. Assim, a solução para a independência do SGBD e manutenção dos dados históricos possui ainda uma forte componente manual e específica, desenvolvida caso a caso, que se pretende automatizar.

1.2 Finalidade e objetivos de trabalho

A finalidade deste projeto de dissertação é disponibilizar uma ferramenta que permita, face a um modelo numa arquitetura MVC *(Model View Controller)*, organizar a distribuição lógica dos dados para que, independentemente do SGBD usado, se possa de forma eficaz retirar os dados antigos do sistema e armazená-los de forma a serem facilmente recuperados.

Este trabalho tem como requisito que a ferramenta seja implementada na linguagem Ruby e na plataforma Ruby on Rails.

Para concretizar este projeto de dissertação, é necessária a realização de quatro objetivos, nomeadamente:

1. Estudo da linguagem Ruby e da plataforma Ruby on Rails;
2. Estudo da técnica utilizada pelos SGBD[1] - Sistema Gestor de Bases de Dados atuais relativamente à gestão de dados históricos;
3. Estudo e análise sobre a forma como o ORM - *Object Relational Mapping* do Rails, o Active Record, implementa as ações CRUD;
4. Definição de uma biblioteca, independente do SGBD, que quando usada por um ou mais modelos permita que estes usem particionamento de dados;

A implementação destes objetivos tem de ser realizada considerando que se procura uma solução global, abrangente, independente do SGBD e que, simultaneamente, não prejudique o desempenho das aplicações nem aumente a complexidade do seu desenvolvimento ou utilização.

[1] Em inglês: DataBase Management System (DBMS).

1.3 Organização do documento

A dissertação é constituída por um total de sete capítulos e um anexo. O primeiro capítulo tem como propósito contextualizar o projeto, descrevendo alguns dos problemas inerentes, assim como a finalidade e principais objetivos do trabalho desenvolvido.

O segundo capítulo retrata as principais tecnologias, conceitos, fundamentos e metodologias que alicerçaram todo o trabalho desenvolvido. É dada ênfase aos tópicos mais relevantes para o projeto.

A gestão de dados históricos é o foco deste trabalho, sendo que no capítulo três são retratados os motivos que levam à necessidade do uso de técnicas de gestão de dados mais antigos de uma aplicação, assim como a técnica muito conhecida denominada de particionamento.

O quarto capítulo retrata a análise do problema, analisando os principais métodos utilizados pelo Active Record que permitem implementar as ações CRUD. É também apresentada neste capítulo a abordagem seguida como proposta de resolução do problema.

A implementação da solução proposta é apresentada no capítulo cinco, em que é descrito o cenário de implementação, assim como a definição da biblioteca e quais os métodos do Active Record que a mesma reescreve.

O capítulo seis apresenta os resultados obtidos, após alguns testes funcionais e de desempenho utilizando a biblioteca desenvolvida. Este capítulo termina com uma conclusão sobre os resultados obtidos.

Finalmente, as conclusões do projeto ficam remetidas para o sétimo capítulo, onde é apresentada uma síntese do trabalho realizado e algumas propostas de trabalho futuro.

2 Estado da arte

O armazenamento e processamento da informação passaram por diferentes fases ao longo do tempo. Desde os sistemas de gestão de ficheiros aos sistemas de bases de dados, há várias soluções comerciais no mercado. Neste capítulo é apresentado o estado da arte, apresentando o estado atual do mercado de sistemas de gestão de bases de dados. É também apresentada a linguagem e *framework* utilizadas no desenvolvimento deste trabalho, assim como a arquitetura utilizada pela *framework* e outras características da mesma.

2.1 Bases de dados

2.1.1 Origem das bases de dados

Após o início da era informática, rapidamente surgiu a necessidade de se armazenar a informação indispensável a cada organização. Inicialmente essa informação era guardada em ficheiros, surgindo assim os sistemas de gestão de ficheiros, destacando-se os sistemas que utilizavam a linguagem de programação COBOL (*COmmon Business Oriented Language*). É de notar que estes sistemas ainda são usados nos dias de hoje, encontrando-se ainda muitos sistemas em exploração e alguns, embora poucos, em desenvolvimento.

Inicialmente este tipo de tecnologia veio resolver alguns dos problemas das organizações, automatizando algumas tarefas previamente realizadas manualmente. Mas com a evolução dos sistemas de informação, esta solução mostrou-se pouco adequada, pois para cada nova aplicação era necessário definirem-se novos ficheiros e respetivos programas de processamento. Podem referir-se estes sistemas como sendo sistemas isolados, devido a cada aplicação ter os seus dados, não podendo assim existir qualquer relação com os restantes sistemas existentes.

Desta forma os mesmos dados são tratados por diferentes aplicações, originando assim a replicação de dados. A replicação de dados pode originar um grande problema, que

é o de incoerência nos dados (podendo até ocorrer perda de dados), visto que estes são tratados por aplicações diferentes e independentes.

O uso de sistemas de ficheiros não implica por si só um ficheiro de dados por cada aplicação, o que já resolve alguns problemas, nomeadamente os descritos anteriormente. Mas, mesmo que várias aplicações usem o mesmo ficheiro de dados, verificam-se outros problemas no contexto de partilha de dados. Neste tipo de sistemas há uma interface física entre a aplicação e o ficheiro de dados, o que implica que todas as aplicações tenham uma especificação dessa interface. Este facto resulta num grande problema para a manutenção de sistemas, pois caso haja alguma alteração num determinado ficheiro, esta tem que se propagar por todas as aplicações que o utilizem. O outro problema consiste no acesso concorrente aos dados, que é ainda mais grave pois fica à incumbência das aplicações.

Na sequência de todos os problemas detetados nos sistemas de ficheiros, nasce o conceito de Base de Dados. Uma base de dados pode ser definida como sendo uma coleção de dados estruturados, organizados e armazenados de forma persistente por uma aplicação informática [Damas, 2005]. Numa base de dados, os dados são organizados de uma forma estruturada num único conjunto, com a menor redundância possível, ao contrário dos sistemas de ficheiros.

Segundo Damas [Damas, 2005], pode-se pensar em três gerações de sistemas baseados nos modelos[2] utilizados:

1. Geração Pré-Relacional
    a. Sistemas baseados em ficheiros;
    b. Modelo Hierárquico;
    c. Modelo em rede;
2. Geração Relacional
    a. Modelo relacional.
3. Geração Pós-Relacional
    a. Modelo orientado por objetos;
    b. Modelo Objeto-Relacional;
    c. Outros modelos.

[2] Um modelo de bases de dados é um modelo lógico de representação dos dados [Damas, 2005].

Quanto aos sistemas baseados em ficheiros, como já foi referido anteriormente, têm problemas associados, como a redundância e a inconsistência dos dados, inexistência de relações entre os dados, integridade, segurança e concorrência.

O modelo hierárquico evoluiu a partir dos sistemas de ficheiros e foi o primeiro modelo associado a um sistema de base de dados. Neste modelo os dados são organizados em hierarquias ou árvores invertidas e já existem relações. No entanto, este modelo, como todos os outros, apresenta vantagens e desvantagens. Tem um grande desempenho no acesso sequencial e a informação é facilmente representada, mas o desenvolvimento de aplicações torna-se bastante lento devido à necessidade da estrutura implementada ter que ser conhecida ao pormenor, não permitir relações de muitos para muitos, não ser eficiente devido às alterações na estrutura que uma simples tarefa pode provocar e ocorre também replicação, o que leva à inconsistência dos dados e ao desperdício de espaço utilizado.

Com o objetivo de resolver alguns problemas do modelo hierárquico surgiu o modelo em rede. Este modelo é considerado uma extensão do modelo hierárquico, mas elimina o conceito de hierarquia. Os problemas de redundância e repetição da informação são resolvidos, pois neste modelo representam-se as relações como conjuntos (*sets*) ao contrário do modelo hierárquico, que se representam como hierarquias. Ao contrário também do modelo hierárquico, em que qualquer acesso aos dados passa pela raiz, no modelo em rede pode aceder-se a qualquer nó da rede. Este modelo permite relações de muitos para muitos, mas não são aconselhadas de modo a simplificar a implementação. Apesar das melhorias introduzidas este modelo não foi bem sucedido, estando já completamente ultrapassado, assim como o modelo hierárquico.

Como tentativa de solução alternativa aos modelos hierárquico e de rede, surgiu o modelo relacional. Foi a IBM - *International Business Machines Corporation* que deu o primeiro passo na área das Bases de Dados relacionais, através do seu investigador Ted Codd, que descreveu inicialmente o conceito de bases de dados relacionais numa publicação de pesquisa da IBM, intitulada de "*System R4 Relational*". Este artigo de pesquisa tratava o uso de cálculo e álgebra relacional para possibilitar o armazenamento e recuperação de informação [Damas, 2005]. Este modelo baseia-se num modelo matemático rigoroso, sendo este o seu ponto forte. A estrutura de dados neste modelo é a relação, a qual se define

através de uma tabela. Numa tabela, as colunas são os atributos (por exemplo: nome, idade, telefone) e as linhas são tuplos, os quais definem uma instância da relação.

Apesar de este modelo ser o que teve maior sucesso até aos dias de hoje, é necessário referir outros que estão a surgir: os modelos da geração pós-relacional. Estes modelos surgem na tentativa de aplicar o conceito da programação orientada por objetos aos sistemas de bases de dados (modelo orientado por objetos), e até juntar este conceito às bases de dados relacionais (modelo objeto-relacional).

2.2 Sistemas gestores de bases de dados

Um SGBD é uma aplicação informática que fornece a *interface* entre os dados que são armazenados fisicamente na base de dados e o utilizador [Damas, 2005]. Ou seja, ao contrário dos sistemas de ficheiros, todo o acesso aos dados é centralizado através do SGBD, como se pode ver na Figura 2.1.

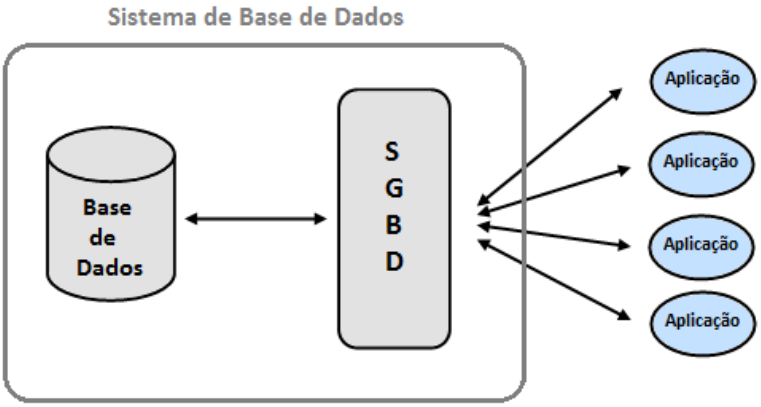

Figura 2.1 - Sistema de base de dados

Segundo Damas [Damas, 2005], um SGBD é responsável por:

- Interação com o gestor de ficheiros, de modo a minimizar os acessos a disco (operações I/O).

- Gestão de dados, que é feita exclusivamente pelo SGBD, que gere os dados e também as relações existentes entre eles.
- Integridade, para garantir que os dados não sofrem modificações que não estejam de acordo com as regras definidas.
- Segurança, onde tem que assegurar que os utilizadores só têm acesso à informação que podem aceder.
- *Backup* e *Recovery*, o que permite detetar falhas que possam ocorrer e repor a base de dados no estado em que estava antes das falhas.
- Gestão da concorrência, que permite gerir o acesso múltiplo aos dados, mantendo sempre a consistência da informação.

Um SGBD precisa de uma linguagem para permitir aos utilizadores operarem sobre a base de dados, pois é ele que está encarregue de aceder aos dados. A maioria dos SGBDs usa a linguagem SQL - *Structured Query Language*, sendo denominados por SGBDR - Sistema Gestor de Base de Dados Relacional. A linguagem SQL é uma linguagem completa para a manipulação de dados numa base de dados relacional e é hoje em dia um padrão ISO - *International Organization for Standardization*. A sua origem está no artigo de Ted Codd (anteriormente referido) sobre bases de dados relacionais.

Apesar de ter sido a IBM a dar o primeiro passo na teoria sobre bases de dados relacionais, o primeiro SGBDR a existir foi realizado pela *Relational Software Inc.*, hoje conhecida por *Oracle Corporation*, em 1979 [Damas, 2005]. Nos dias de hoje há vários SGBDs no mercado, sendo os mais conhecidos: *Oracle*, *DB2*, *MySql*, *MS SQL Server* e *PostgreSQL*.

Segundo um estudo realizado pela DB-Engines[3] [DB-Engines Ranking, 2013], em Outubro de 2013, os três SGBDs comerciais com mais quota de mercado eram o Oracle, o MySQL e o MS SQL Server. No entanto, num estudo da IDC - *International Data Corporation* [Olofson, 2008] realizado entre 2005 e 2007, os resultados apontaram que os três SGBDs com mais quota de mercado eram o Oracle, o DB2 e o MS SQL Server.

Há inúmeros estudos sobre a quota de mercado de cada SGBD, no entanto, alguns são estudos parciais, que abrangem um determinado mercado ou uma determinada região geográfica, ou até um determinado espaço de tempo, outros estudos comparam apenas

[3] A DB-Engines é uma iniciativa criada pela empresa Solid IT, que visa recolher e apresentar informação sobre os SGBDs comerciais, fazendo um ranking todos os meses. Estudo disponível em http://db-engines.com/en/ranking, consultado em Outubro de 2013.

determinadas versões (normalmente as mais atuais), e alguns estudos nem referem a forma como calculam as quotas dos SGBDs.

Nas secções seguintes apresentam-se quatro sistemas de bases de dados que são aqueles que possuem uma maior quota de mercado e, principalmente, por serem aqueles que são passíveis de ser utilizados no âmbito do trabalho desenvolvido.

Oracle

O Oracle [Oracle, 2013] foi o primeiro SGBD comercial baseado na linguagem padrão SQL, criado por Larry Ellison, Bob Miner e Ed Oates, comercializado desde Julho de 1979 [Damas, 2005]. Desde 1979 até aos dias de hoje, foram lançadas várias versões, sendo as mais recentes o Oracle 11g [Carr et al., 2008] e o Oracle 12c [Oracle Database New Features Guide, 12c Release 1, 2013].

O Oracle é composto por um completo conjunto de ferramentas que permitem o desenvolvimento de aplicações e bases de dados, e ainda integrar capacidades de inteligência de negócio[4] [Oracle, 2013]. É composto pelas seguintes ferramentas de desenvolvimento [Greenwald et al., 2007]:

- *Oracle JDeveloper:* permite o desenvolvimento de aplicações básicas sem ser necessário escrever código.
- *Oracle SQL Developer:* simplifica o desenvolvimento e gestão de bases de dados Oracle.
- *Oracle Developer Suite:* permite o desenvolvimento rápido e completo de aplicações transacionais de alta qualidade, que podem ser implementadas em diferentes canais, como portais, Web Services e dispositivos sem fios e podem ser estendidas com capacidades de inteligência de negócio.

[4] O instituto de armazenamento de dados (TDWI) define inteligência de negócio *(Business Inteligence)* como sendo os processos, tecnologias e ferramentas necessárias para tornar dados em informação, informação em conhecimento e conhecimento em planos que orientam ações de negócio rentáveis [Loshin, 2003].

A última ferramenta referida, também conhecida como Oracle Internet Developer Suite, é constituída por [Greenwald et al., 2007]:

- *Oracle Forms Developer;*
- *Oracle Reports Developer;*
- *Oracle Designer;*
- *Oracle Discoverer Administrative Edition;*
- *Oracle Portal.*

Um conceito que tem um papel muito importante neste SGBD comercial é o particionamento de dados, pelo que das últimas versões resultaram, entre outras, melhorias importantes no particionamento de dados. Neste contexto surgiu um aprimoramento, que é o particionamento por intervalos, o qual permite a criação automática de partições baseadas num intervalo predefinido da chave de partição.

Ainda no contexto do particionamento de dados, a versão 11g do SGBD Oracle tem outra nova característica, que é o particionamento de sistema, o qual permite o controlo ao nível aplicacional para particionar uma tabela. Outras da vasta gama de melhorias da versão 11g são: a compressão de tabelas, que é muito útil em bases de dados de grande escala[5] e colunas virtuais em tabelas, que basicamente são expressões armazenadas nos metadados da tabela e não ocupam espaço no disco. As colunas virtuais podem ser indexadas, usadas como chave de particionamento e podem até conter estatísticas optimizadoras [Carr *et al.*, 2008].

Já na versão 12c verificam-se melhorias em algumas características como [Oracle Database New Features Guide, 12c Release 1, 2013]:

- *Real-Time database operations monitoring*: permite aos administradores da base de dados facilmente monitorizar e solucionar problemas de desempenho enquanto correm tarefas, vendo exatamente que operações estão a ser realizadas e em que tempos;
- *Error Handling*: a forma de apresentar erros e exceções foi melhorada, de forma ao programador poder apresentar erros de uma forma mais simples para o utilizador;

[5] A compressão de tabelas permite economizar bastante o espaço em disco.

- *Partition Maintenance Operations on Multiple Partitions*: uma simples operação de manutenção de partições pode correr em várias partições ao mesmo tempo, o que facilita o desenvolvimento da aplicação e leva a uma manutenção mais eficiente das partições usando menos recursos do sistema.

Ao longo deste capítulo apenas se apresentam algumas das novas características das versões 11g e 12c do Oracle. A compreensão em detalhe de todas as características pode ser feita consultado a bibliografia existente ou o *website* da Oracle [Oracle, 2013].

De facto a Oracle tem sido recompensada por todas as melhorias realizadas nas sucessivas versões do SGBD Oracle, pois sempre foi um dos SGBDs comerciais que teve maior quota no mercado.

MS SQL Server

A primeira versão do SGBD MS SQL Server [Microsoft SQL Server, 2013] foi lançada em 1988, resultado de uma parceria entre a Microsoft com a Sybase e a Ashton-Tate. Acabada a parceria, a Microsoft continuou a melhorar o produto e a lançar novas versões.

As versões mais recentes são o SQL Server 2008 e o SQL Server 2012. Da última destacam-se grandes melhorias, entre as quais estão [Mistry & Misner, 2012]:

- *AlwaysOn Availability Groups*: esta funcionalidade permite que uma base de dados apresente uma maior disponibilidade de funcionamento, pois permite redundância e proteção de dados, dado que suporta até quatro réplicas secundárias. Das quatro réplicas secundárias, duas podem ser configuradas como síncronas, garantindo que as bases de dados estão sempre atualizadas. Sendo síncronas, estas réplicas podem ser usadas para efetuar *queries ad-hoc*, realizar cópias de segurança, gerar relatórios ou efetuar simples consultas aos dados, evitando sobrecarregar a base de dados "principal". Estas réplicas podem ser armazenadas em diferentes *datacenters*, o que incrementa a segurança dos dados e permite recuperação de desastres.

- *Columnstore indexes*: devido às quantidades enormes de dados que algumas organizações começaram a ter, foi desenvolvido o conceito de modelo de armazenamento e de otimização de *queries* avançadas. A indexação criada pelo MS SQL Server, ao contrário da indexação em árvore[6], armazena dados por cada coluna e posteriormente junta todas as colunas para completar o índice. Como resultado, os dados com características semelhantes ficam juntos e consegue-se assim uma maior compressão dos dados, diminuindo assim o *input* necessário para realizar a *query*, levando a um desempenho mais rápido na resposta. Um aspeto importante é que quando um utilizador executa uma *query* usando este mecanismo, o SQL Server procura os dados só nas colunas necessárias às *queries*.

Um conceito muito importante sempre presente no SQL Server tem sido a segurança, e por essa razão é o SGBD que menos tem reportado vulnerabilidades e exposições comuns. O gráfico apresentado na Figura 2.2 mostra as vulnerabilidades reportadas ao NIST - *National Institute of Standards and Technologies* entre Janeiro de 2002 e Janeiro de 2010.

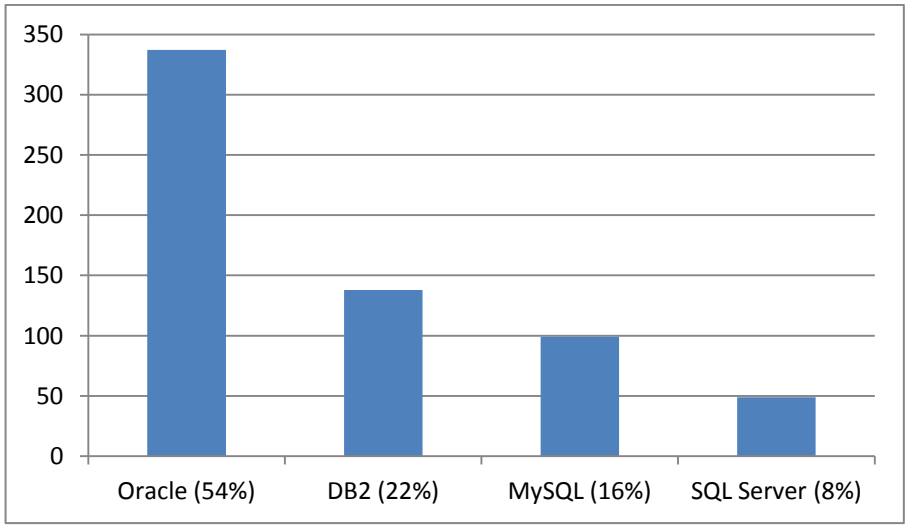

Figura 2.2 - Vulnerabilidades reportadas ao NIST de Janeiro de 2002 a Janeiro de 2010 [ITIC, 2011]

[6] Indexação é uma estrutura adicional que permite simular a ordenação de dados e consiste normalmente numa "árvore" de termos *(B-Trees)* que apontam para uma determinada posição no ficheiro ou tabela [Damas, 2005].

O MS SQL Server 2012 consiste em três edições principais [Mistry & Misner, 2012]:

- "Standard Edition"
- "Business Inteligence Edition"
- "Enterprise Edition"

Todas as três versões foram desenhadas de acordo com as necessidades de vários tipos de clientes, tendo sido feito um maior investimento em "Business Inteligence" (inteligência de negócio), o qual engloba técnicas que pode melhorar o desempenho nas pesquisas de informação.

Além das três edições principais referidas, existem edições especializadas para organizações que têm um pequeno conjunto de requisitos. Destas três distingue-se a edição MS SQL Server Express Edition, a qual é gratuita e serve para fins não profissionais.

MySQL

O MySQL [MySQL, 2013] é um SGBD relacional gratuito para fins não comerciais e o seu *software* é *open source*[7]. O seu desenvolvimento teve início em 1994, na Suécia, por David Axmark, Allan Larsson e Michael Widenius. Em 1995 foi lançada a primeira versão interna do MySQL. Em 2008 a empresa Sun comprou o MySQL. Em 2009 a Oracle comprou a Sun e todos os seus produtos, incluindo o MySQL.

O MySQL foi evoluindo ao longo do tempo, surgindo novas versões com características melhoradas. Nas primeiras versões, o MySQL não dispunha de referencial de integridade, isto é, não suportava o conceito de chave estrangeira. Este facto não impediu este SGBD de crescer no número de utilizadores, apesar de esta importante característica estar disponível nos SGBDs concorrentes. A sua mais valia foi o facto de ser um *software* livre, que conta com um rápido desenvolvimento de toda uma comunidade.

Segundo Schwartz et al. [Schwartz et al., 2012] a característica mais importante e incomum do MySQL é a sua arquitetura de motor de armazenamento *(storage-engine)*, em

[7] Um *software open source* (código aberto) é um *software* que deve ser distribuído com o seu código fonte ou em que este esteja facilmente disponível, de modo a ser possível, a qualquer pessoa que pretenda, modificar o código consoante as suas necessidades [Kavanagh, 2004].

que o processamento de *queries* e outras tarefas do servidor são separados do armazenamento e recuperação de dados.

As versões mais recentes do MySQL são as versões 5.5 e 5.6. A última versão estável lançada é a 5.6, estando em desenvolvimento e testes a versão 5.7 [MySQL, 2013].

A versão 5.5 foi a primeira versão lançada após a aquisição da Sun (e consequentemente, do MySQL) pela Oracle. Nesta versão destacam-se melhorias no desempenho, escalabilidade, replicação, particionamento, entre outras. Dentro destas melhorias foram adicionadas características como [Schwartz et al., 2012]:

- *Performance Schema:* característica desenvolvida para monitorizar a execução do MySQL Server;
- *Thread pooling:* ao contrário do modelo conhecido de uma *thread* lançada para cada conexão, esta característica consiste em conjuntos configurados de *threads* que vão recebendo tarefas. Neste modelo não existe nenhum relacionamento entre a conexão e a *thread* que a atende.
- *IPv6 support:* a partir da versão 5.5 o MySQL Server suporta conexões TCP/IP de clientes que se conectam através de IPv6.
- *Partitioning:* na versão 5.5 foram introduzidos novos tipos de particionamento de dados, os quais permitem definir gamas (*Range Columns*) ou listas (*List Columns*) de particionamento baseados nos tipos de dados *Date*, *Datetime*, *Char*, *Varchar*, e até com base em valores de múltiplas colunas. Em tabelas que usem um destes dois tipos de particionamento é possível otimizar as *queries* através do conceito de *partitioning pruning*[8]. A partir desta versão é possível também apagar todas as linhas de uma ou mais partições de uma tabela previamente particionada.

A versão corrente, 5.6 apresenta melhorias em algumas características como [MySQL Documentation, 2013]:

- *Security*: esta versão permite guardar num ficheiro, as credenciais de autenticação, devidamente encriptadas; nesta versão são também suportadas encriptações mais fortes;

[8] *Partitioning pruning* é um processo no qual uma *query* lê um subconjunto de partições e até muitas vezes uma só partição como resultado de um filtro aplicado, normalmente através da cláusula *where* [Schwartz et al., 2012].

- *Explicit partition selection*: nesta versão é permitido especificar uma ou mais partições numa *query*; relativamente ao particionamento, esta versão aumentou o número máximo de partições permitido;
- *Optimizer enhancements*: no otimizador de *queries* é permitido nesta versão fazer uma *query* retornando apenas poucas linhas de um grande conjunto de resultados, com ORDER e LIMIT; este tipo de *queries* é comum em aplicações *web*, para fazer o display de alguns registos de um grande conjunto.

PostgreSQL

Entre 1977 e 1984 foi desenvolvido na Universidade da Califórnia, em Berkeley, o Ingres que foi uma tentativa de desenvolvimento de um sistema de bases de dados relacional, por Michael Stonebraker [PostgreSQL, 2013]. Entre 1986 a 1994 foi desenvolvido o sistema de bases de dados Postgres (cujo nome surgiu de "após Ingres").

O PostgreSQL [PostgreSQL, 2013] é um sistema de base de dados objeto-relacional *open source.* Tem mais de 15 anos de desenvolvimento ativo e uma arquitetura com uma forte reputação em confiabilidade, integridade de dados e correção. Este sistema de base de dados corre na maioria dos sistemas operativos, incluindo *Linux*, *Unix* e *Windows*.

Para além de suportar os tipos de dados comuns, como por exemplo, *Integer*, suporta também o armazenamento de objetos binários grandes, incluindo imagens, sons ou vídeos.

Este sistema de base de dados é compatível com ACID - *Atomicity, Consistency, Isolation, Durability* e tem um completo suporte para chaves estrangeiras, *joins*, *views*, *triggers* e procedimentos armazenados[9] (em várias linguagens). O PostgreSQL tem também alguns recursos sofisticados, como por exemplo, *Multi-Version Concurrency Control* (MVCC), o qual assegura uma eficiente concorrência mesmo quando se verificam operações de leitura e escrita.

[9] Procedimentos armazenados (*stored procedures*) são funções que podemos criar, ficando armazenadas com a base de dados.

As versões mais recentes do PostgreSQL são as versões 9.2 e 9.3. A versão 9.2 apresentou novas características como:

- *Index-only scans:* permitem que alguns tipos de *queries* sejam respondidos através da recuperação de dados de *indexes* e não de tabelas. Isto traz um maior desempenho no tempo de resposta a *queries* características de *data warehousing,* isto é, relativamente grandes quantidades de dados estáticos, atualizados com pouca frequência, onde relatórios sobre dados históricos são frequentemente necessários.
- *Cascading replication*: quanto à replicação, até esta versão todos os clientes estavam conectados a um só servidor. Caso acontecesse uma falha era necessário todos os clientes reconectarem-se ao novo servidor, o que pode ser extremamente complicado. A partir da versão 9.2, um servidor *standby* pode aceitar conexões de replicações e efetuar *streaming replication* (constantes replicações de dados) para outros *standby.* Este facto pode diminuir o número necessário de conexões.
- *JSON data type*: o tipo de dados *json* permite o armazenamento de dados JSON - *JavaScript Object Notation*, permitindo validar os valores inseridos como sendo do tipo JSON.

A versão mais corrente do PostgreSQL, 9.3, apresenta melhorias e novas características, como por exemplo [PostgreSQL Documentation, 2013]:

- *Foreign data wrappers*: permitem dar suporte a ações de escrita (*inserts*/*updates*/*deletes*) em tabelas secundárias; permitem também aceder a outros servidores PostgreSQL;
- *Auto-updatable views*: uma *view* pode ser atualizada como uma tabela normal;
- *JSON functions and operators*: permite ler valores JSON e extrair os seus valores. Esta nova característica pode ser muito útil, por exemplo, para o desenvolvimento de uma aplicação *web*.

Conclusão

Como ficou demonstrado, há diversos SGBDs com diferentes características. Cada organização tem os seus próprios critérios que levam à adoção de um determinado SGBD. Alguns motivos para as empresas escolherem diferentes SGBDs estão relacionados com o custo, ou com a familiarização já adquirida com o SGBD que já utilizavam. Um outro possível motivo é a necessidade tecnológica; por exemplo, o PostgreSQL no ano de 2000 era dos poucos, ou mesmo o único, a suportar *queries* com informação geográfica.

2.3 Arquitetura Model-View-Controller

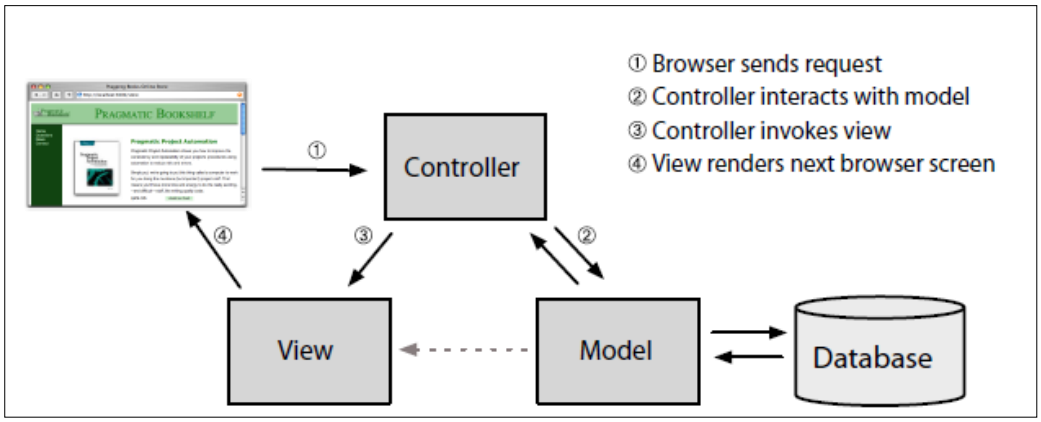

Figura 2.3 - Arquitectura MVC [Ruby et al., 2011]

A arquitetura MVC surgiu em 1979, tendo sido idealizada por Trygve Reenskaug para desenvolver aplicações interativas [Ruby et al., 2011]. Nesta arquitetura, como se pode ver na imagem anterior, as aplicações são divididas em três tipos de componentes: *models*, *views* e *controllers*.

Model

Este componente é responsável pela manutenção do estado da aplicação, sendo este por vezes transitório. Pode ser também permanente, estando armazenado normalmente numa base de dados. Não contém apenas dados, mas também regras para serem aplicadas

aos dados consoante o tipo de aplicação (por exemplo uma restrição de saldo). Isto impossibilita que alguma ação na aplicação invalide os dados.

View

Este componente é responsável por disponibilizar uma *interface* ao utilizador baseada nos dados do componente *model*, embora nunca trate dos dados de entrada na *interface*. Podem existir várias *views* que acedem aos dados presentes no mesmo modelo, mas para diferentes propósitos (por exemplo, uma para listar itens e outra para adicionar itens).

Controller

Como o próprio nome indica, este componente controla a aplicação, no sentido de receber pedidos do utilizador, interagir com o componente *model*, e apresentar a view adequada ao utilizador.

Segundo Ruby et al. [Ruby et al., 2011], com esta arquitetura torna-se muito mais simples escrever código e fazer a sua manutenção.

Existem outras arquiteturas alternativas à arquitetura MVC, nomeadamente a MVP – *Model View Presenter* e a MVVM – *Model-View-ViewModel*. Ambas arquiteturas derivam da arquitetura MVC, tendo todas o mesmo objetivo: separação de funções.

A principal diferença entre a MVC e as arquiteturas derivadas, MVP e MVVM, está na dependência que cada camada tem sobre outras camadas, bem como a forma em que estão ligadas entre si.

Verifica-se pelos dois primeiros elementos da designação de cada modelo que todas as arquiteturas têm em comum os módulos *Model* e *View*. O outro módulo difere no nome mas tem o mesmo papel: define a lógica de negócio, sendo muitas vezes chamado de *glue code*, pois tem como função unir diferentes partes do código que de outra forma não seriam compatíveis. Este módulo na arquitetura MVC é o *Controller*. Na arquitetura MVP é o *Presenter* e na arquitetura MVVM é o *View Model*.

2.4 Ruby

O Ruby é uma linguagem relativamente recente, tendo sido criada por Yukihiro Matsumoto no Japão. A primeira versão foi lançada em Dezembro de 1995, a qual foi rapidamente seguida por mais três versões nos dois dias seguintes. Esta linguagem é uma linguagem moderna de *scripting* e é orientada a objetos, pois tudo o que se manipula é um objeto.

> Segundo *Yukihiro Matsumoto* em [Thomas, 2005]*: "Eu acredito que o propósito da vida é, pelo menos em parte, ser feliz. Com base nessa crença, o Ruby é projetado para tornar a programação não só fácil, mas também divertida. O Ruby permite que se concentre no lado criativo da programação[…]."*

Segundo Thomas [Thomas, 2005] a linguagem Ruby segue o princípio da menor surpresa, isto é, as coisas funcionam da maneira que se espera que funcionem, com muito poucos casos especiais ou exceções, e é uma linguagem transparente, pois conseguimos expressar as nossas ideias sem ter que escrever extensos blocos de código para fazer algo simples. Podemos então expressar as nossas ideias de um modo simples, elegante, tornando também assim os nossos programas sempre legíveis e de fácil manutenção.

A linguagem foi concebida de forma a ocorrerem poucos erros de sintaxe. Uma anotação interessante sobre esta linguagem, a título de exemplo, é que não necessita de nenhum ponto e vírgula no final de cada instrução. A conceção do Ruby foi efetuada tendo sempre como objetivo máximo a criação de uma linguagem que facilite o desenvolvimento e manutenção de código.

Quanto ao tipo de linguagem que é o Ruby, é importante referir que esta pode fazer tudo o que linguagens de *script* fazem (como o Perl ou o Python), e até de uma forma mais limpa. Mas é mais usada como uma linguagem de programação de propósito geral, usada em aplicações GUI (*Graphical User Interface*) podendo ser usada em problemas de diversos domínios.

2.5 Ruby on Rails

O Ruby on Rails é uma *framework* escrita em Ruby que permite de uma forma mais fácil desenvolver e implementar aplicações *web* e ainda efetuar a sua manutenção [Ruby et al., 2011]. Normalmente referenciado como Rails ou RoR, tornou-se mundialmente conhecido logo após alguns anos do seu desenvolvimento, principalmente devido aos programadores de aplicações *web* que não estavam satisfeitos com as tecnologias que usavam até surgir o Rails.

As aplicações Rails são escritas em Ruby, que como já foi referido na secção anterior, devido às suas características, permite desenvolver programas que são fáceis de escrever e rever algum tempo mais tarde. Segundo Ruby et al. [Ruby et al., 2011], existem dois fundamentos para que isso aconteça. Esses dois fundamentos são o DRY (*Don't Repeat Yourself*) e o *Convention Over Configuration*. O DRY conta com o Ruby para que cada pedaço de conhecimento no sistema seja expressado apenas num sítio, normalmente sugerido pela arquitetura MVC. Quanto a *Convention Over Configuration*, tem como objetivo não se despender tempo em configurações, pois o Rails assume configurações-padrão para uma determinada convenção.

Mesmo assim é possível a cada programador definir as suas próprias convenções e configurações. Algumas convenções do Rails serão abordadas na próxima secção deste capítulo.

Todas as aplicações Rails são implementadas recorrendo à arquitetura MVC, descrita anteriormente. Esta arquitetura permite:

- Isolar a lógica do negócio da interface do utilizador;
- Facilitar a manutenção de código seguindo o princípio DRY;
- Clarificar onde pertencem diferentes e específicos bocados de código.

O Rails oferece também a escrita automatizada de testes, pois conforme vamos desenvolvendo uma aplicação Rails, vai sendo acrescentando um *test stub*[10] a cada nova funcionalidade.

[10] Um *test stub* é programa que simula o comportamento de um componente (ou módulo) de um programa [Test stub, 2012].

Quanto a recursos recentes, como por exemplo o AJAX - *Asynchronous Javascript and XML* e interfaces REST[11] - *Representational State Transfer*, o Rails também torna mais fácil a sua integração no código de uma aplicação.

De forma resumida, pode dizer-se que o Rails é uma plataforma, em que são criados à partida alguns módulos de acordo com a arquitetura MVC. Pode dizer-se também que o Rails é ágil[12], pois é construído com pedaços de código individuais e interações, e usufrui de transparência, ou seja, o que desenvolvemos é praticamente o que o cliente vê. Mesmo após finalizar o *software*, é possível fazer alterações de uma forma muito fácil, visto as aplicações serem escritas em Ruby, o que permite que o código seja escrito de uma forma precisa e concisa, tornando-o mais apto a alterações.

2.5.1 Aplicação Ruby on Rails

Uma aplicação Rails completa engloba base de dados, interface *web*, lógica de negócio e suporta os métodos CRUD – *Create*, *Read*, *Update* e *Delete*.

Para se criar uma aplicação Rails, seguindo o nome apresentado na imagem, recorre-se ao comando ***"rails new app_name"***.

Aquando a execução deste comando são criadas todas as subdiretorias visíveis dentro da diretoria base da aplicação, sendo a principal a diretoria **app**, pois é nesta que ocorre a maioria do trabalho que o programador desenvolve. Dentro desta encontram-se as subdiretorias correspondentes à arquitetura MVC – *models*, *views*, *controllers*, à exceção da diretoria app/helpers, que detém quaisquer classes auxiliares ao *model*, ao *controller* ou à *view*, de modo a manter o código destes módulos pequeno e organizado.

Esta estruturação é visível na Figura 2.4.

[11] REST permite a definição de recursos (URLs) para assim se aplicarem acções HTTP: Get, Put, Post, Delete. Por exemplo, para obtermos todos os produtos de uma base de dados, através de uma página web, basta introduzirmos o URL /products.
[12] O adjetivo ágil aparece como uma tradução livre da expressão em inglês "Agile Development".

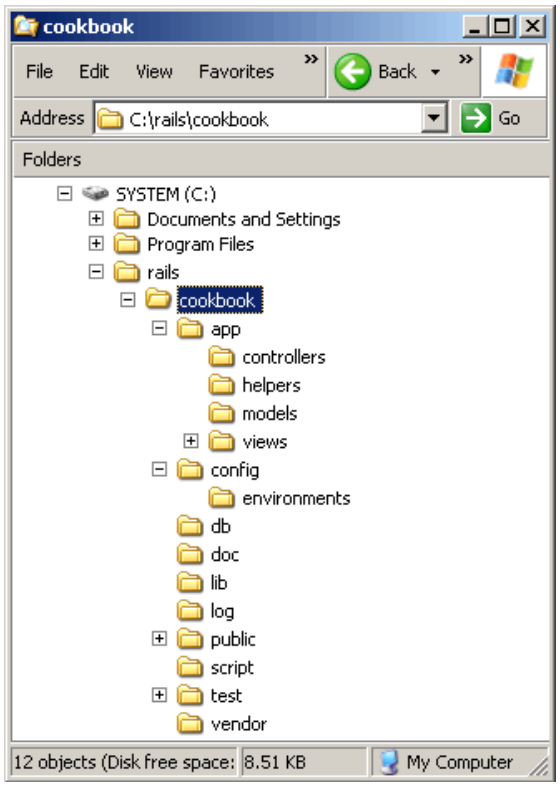

Figura 2.4 - Estrutura de diretorias de uma aplicação Ruby on Rails [O'Reilly, 2013]

Como o Rails é compatível com uma grande quantidade de bases de dados podemos escolher aquela com a qual temos mais familiaridade, ou o que já tivermos instalado na nossa máquina. Por padrão o Rails usa o SQLite3, mas se quisermos definir por exemplo o MySQL, basta executar o comando ***"rails new app_name -d mysql"***.

Todas as diretorias que o Rails cria automaticamente aquando a criação de uma nova aplicação são para utilização dos diferentes componentes envolvidos na aplicação.

Para cada novo *controller*, por defeito, o Rails segue como abordagem o protocolo REST, ou seja, usando os verbos HTTP e URLs bem definidas para lidar com os métodos CRUD. As ações geradas são [Ruby Tutorials, 2011]:

- *Index* – esta ação retorna todos os recursos. Por exemplo, GET /users chamará a ação *index* do *controller* Users, listando todos os recursos disponíveis;
- *Show* – esta ação implementa uma leitura de um único recurso. Por exemplo, GET /users/1 chamará a ação *show* do *controller* Users, mostrando os detalhes do User com id 1;

- *New* – esta ação mostra o formulário de um novo objeto. Por exemplo, GET /users/new, chamará a ação *new* do *controller* Users e mostrará o formulário para um novo User;
- *Create* – esta ação tenta criar um novo registo. Por exemplo, POST /users invocará a ação *create* do *controller* Users, com alguns dados do formulário associados.
- *Edit* – esta ação chama o formulário para editar um recurso específico. Por exemplo, GET /users/1/edit irá chamar a ação *edit* do *controller* Users e mostrará um formulário para editar o User com id 1;
- *Update* – esta ação tenta atualizar um determinado recurso. Por exemplo, PUT /users/1 chamará a ação *update* do *controller* Users e tentará atualizar o User com id 1, associando os dados do formulário obtido através da ação *edit*;
- *Destroy* – esta ação tenta destruir um dado recurso. Por exemplo, DELETE /users/1 chamará a ação *destroy* do *controller* Users e tentará destruir o User com id 1.

As características inerentes à *framework* Ruby on Rails, como por exemplo o uso da arquitetura MVC e de convenções sobre configurações estão distribuídas pelos componentes do Ruby on Rails. A Figura 2.5 mostra a interação entre alguns destes componentes:

- Action WebServices;
- Action Mailer;
- Action Pack (Action Controller, Action Dispatcher e Action View);
- Active Model:
  - Active Resource;
  - Active Support;
  - Railties;
  - Active Record.

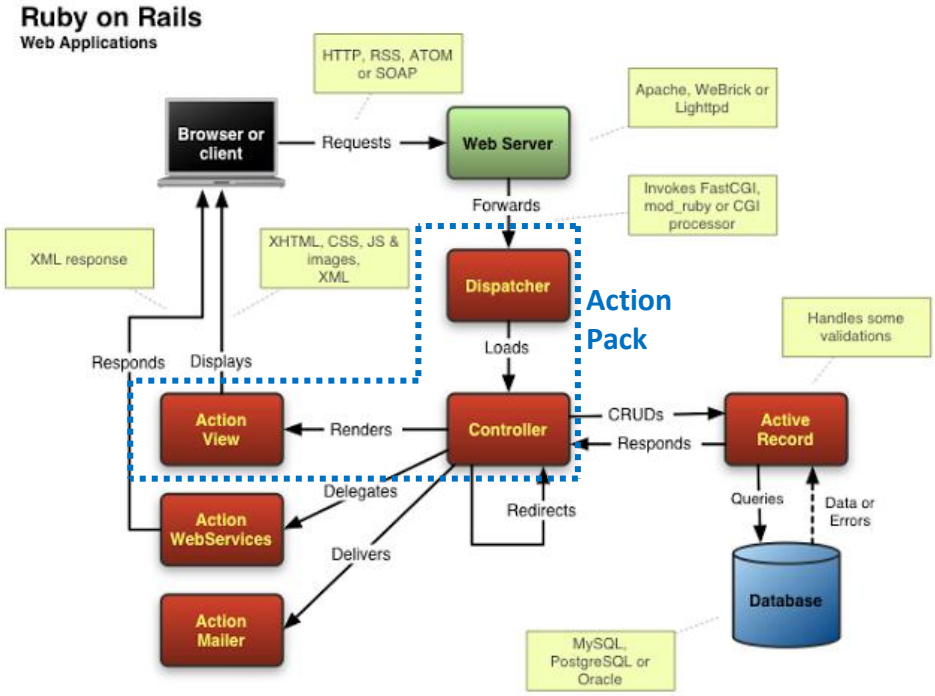

Figura 2.5 - Componentes de uma aplicação Ruby on Rails [Ruby on Rails Architectural Design, 2013]

Action WebServices

Este módulo proporciona uma forma de publicar *API - Application Programming Interface* de *Web Services* interoperáveis sem despender muito tempo investigando detalhes do protocolo.

Action Mailer

Este módulo é responsável por fornecer serviços de *emails*. Permite criar novos *emails* e processar emails recebidos. Também tem tarefas comuns embutidas de envios de mensagens de boas-vindas, *passwords* esquecidas e cumprir qualquer outra necessidade de comunicação escrita. Está relacionado com o módulo *Action Controller*. Este fornece várias maneiras de fazer *emails* com modelos, da mesma forma que o *Action View* utiliza para fazer o *rendering*[13] de páginas *web*.

[13] Foi decidido manter a terminologia inglesa, usando-se nesta dissertação a palavra *rendering* (derivada de render) dada a inexistência de uma só palavra ou expressão em Português com o mesmo significado. *Render* significa mostrar um determinado conteúdo, desenhar ou criar uma imagem conforme um modelo que se encontra no computador.

Action Pack

Este módulo faculta as camadas *View* e *Controller* da arquitetura MVC. Estes módulos capturam os pedidos realizados pelo utilizador e traduzem-nos em ações. É dividido em três sub-módulos, que são: *Action Dispatch*, *Action Controller* e *Action View*.

- *Action Dispatch:* este submódulo lida com o encaminhamento de pedidos feitos pelo utilizador através de um *browser web*.
- *Action Controller:* depois do Action Dispatch processar os pedidos, estes são encaminhados então para o controlador correspondente. Este submódulo proporciona um controlador base a partir do qual todos os controladores podem herdar. Responsabiliza-se por disponibilizar dados, controlar o *rendering* e redireccionamento das *views*. Além disso é responsável por outras funções, como por exemplo, gerir as sessões de utilizadores, o fluxo da aplicação, módulos auxiliares, etc.
- *Action View:* este submódulo é chamado pelo Action Controller. É responsável por fazer o *rendering* da página *web* solicitada, fornecendo modelos que auxiliam a geração de HTML - *HyperText Markup Language* e outros formatos de apresentação de conteúdos, como por exemplo XML - *Extensible Markup Language* e JavaScript de volta para o utilizador. Para tal o Rails fornece três esquemas de templates para definir o conteúdo da resposta da aplicação: ERb – *Embedded Ruby*, que permite gerar código HTML; *builder*, que constrói o conteúdo da resposta em XML e RJS - *Ruby Javascript* que gera código JavaScript.

Active Model

Fornece uma *interface* definida entre os módulos Action Pack e bibliotecas ORM, como por exemplo, o Active Record. Também permite ao Rails usar outras bibliotecas ORM, em vez do Active Record, tornando-o mais flexível e extensível. Pode ser usado para fazer muitas definições dentro de um modelo (validações, associações, *queries* SQL, etc.).

Active Resource

Este módulo gere as conexões entre os serviços *web* RESTful e objetos de negócio e segue o mesmo princípio do Active Record, que é reduzir a quantidade necessária de código para mapear recursos. Mapeia classe de modelos para recursos REST, da mesma forma que o Active Record mapeia classes de modelos para tabelas de base de dados.

Active Support

Este módulo consiste numa coleção de classes utilitárias e extensões da biblioteca padrão Ruby que são úteis para o desenvolvimento em Ruby on Rails. Inclui um ótimo suporte, por exemplo, para *strings multi-bytes*, fusos horários e testes.

Railties

Este módulo é o código nuclear do Rails que constrói novas aplicações. É responsável por aglomerar/unir todos os módulos acima descritos. Adicionalmente, lida com todo o processo de inicialização, gere a interface da linha de comandos e fornece o núcleo de geradores Rails. *Rake* é uma das linhas de comando usadas para executar tarefas sobre a base de dados, desenvolvimento, documentação, testes e limpezas.

O Rails também fornece uma *framework* de testes embutida que gera *stubs* de teste automaticamente aquando a geração de código, permitindo fornecer testes unitários, testes funcionais para *views* e *controllers*, fixtures[14] para testes e fornecimento de dados de testes recorrendo a YAML – *YAML Ain't Markup Language*.

Active Record

O Active Record é a biblioteca ORM fornecida com o Ruby on Rails.

Dada a relevância deste componente no trabalho desenvolvido, o mesmo é apresentado de forma mais detalhada na secção seguinte.

[14] Segundo [Marshal et al., 2007], *fixtures* são basicamente uma maneira de organizar e fornecer dados que se pretenda testar contra.

2.5.2 Active Record

O Active Record é uma biblioteca Ruby que permite aos programas transmitir dados e comandos de e para vários sistemas de armazenamentos de dados, que geralmente são bases de dados relacionais. [Marshal et al., 2007]

Esta biblioteca é a camada ORM fornecida juntamente com o Rails. O conceito chave do Active Record e de outras bibliotecas ORM é que as bases de dados relacionais possam ser representadas em código baseado em objetos, mapeando as tabelas para classes, as linhas das tabelas para objetos e os campos das tabelas para atributos desses objetos. Contudo, difere de outras camadas ORM principalmente pela forma como está configurada, não sendo necessárias praticamente quaisquer configurações iniciais.

O Active Record foi desenhado especificamente para suportar as quatro principais ações que se executam quando se lida com bases de dados, sendo elas: *create*, *read*, *update* e *delete* – CRUD. Um aspeto importante a salientar é o facto de toda a biblioteca estar escrita em Ruby, o que significa que, tudo o que é possível fazer em objetos Ruby, como por exemplo herança, reescrita de métodos e muito mais, também o é com objetos *Active Record*.

Seguindo o conceito ORM, o Active Record permite abstração da base de dados, mas tem outras funcionalidades importantes como:

- Configuração simplificada e suposições-padrão;
- Mapeamento automático entre bases de dados relacionais e objetos;
- Associações entre objetos;
- Agregações;
- Validações de dados;
- *Callbacks.*

No Rails uma associação é uma conexão entre dois modelos, ou seja, duas tabelas. Existem três tipos de associações: *one-to-one*, *one-to-many* e *many-to-many*. Para indicarmos estas associações, acrescentamos aos modelos as seguintes declarações: *has_one*, *has_many*, *belongs_to* e *has_and_belongs_to_many*.

Quanto às agregações, permitem representar atributos como objetos de valor, permitindo expressar por exemplo que o objeto Cliente é composto pelo objeto Morada, entre outros campos. A declaração normalmente usada para definir agregações é *composed_of*.

As validações, como o próprio nome indica, têm o propósito de validar campos de uma tabela, por exemplo na inserção através de um formulário e conseguem-se através de declarações como *validates_presence_of* e *validates_uniqueness_of*.

As *callbacks* permitem chamar um método antes, durante ou depois de uma ação. Estão disponíveis para todo o ciclo de vida de um objeto (instanciação, registo, destruição e validação) e são usadas através de declarações como *before_destroy*.

Existe um conjunto de suposições e convenções que o Active Record toma como padrão numa simples aplicação em que se faz a conexão à base de dados, se cria um objeto Active Record e se guarda esse objeto. Uma das convenções base é que o Active Record infere os nomes das tabelas da base de dados baseando-se nos nomes das classes (modelos). Na definição de dados, também é visível mais uma suposição do Active Record. Além dos campos que pretendemos que a tabela tenha, o Rails adiciona automaticamente um atributo chamado `id` que representa a chave primária do registo. Além deste, o Rails fornece outros atributos adicionais, como por exemplo atributos que 'acompanham' quando um registo foi atualizado a última vez. A estes atributos dá-se o nome de *timestamps*.

2.5.2.1 Interação entre o Active Record e as bases de dados

Como já foi referido, o Active Record interage com as bases de dados, permitindo abstração das mesmas por parte dos programadores. Mas, visto cada uma ter características específicas, é necessário o uso de adaptadores específicos de bases de dados. Como cada um destes adaptadores é único e específico à base de dados com que comunica, cada adaptador também tem requisitos únicos e diferentes, além dos exigidos pelo Active Record em geral.

O GitHub é um serviço, fundado em 2008, onde se pode partilhar código facilmente, criando-se repositórios. Segundo o repositório GitHub do ActiveRecord-JDBC-Adapter

[GitHub, 2013], o Active Record vem com adaptadores para conexões com as bases de dados mais populares e comumente usadas no mercado:

- *DB2;*
- *Firebird;*
- *Derby;*
- *MySQL;*
- *HSQLDB;*
- *Oracle;*
- *PostgreSQL,*
- *SQLite3;*
- *Microsoft SQL Server;*
- *H2;*
- *Informix.*

Estes adaptadores são baseados em JDBC – *Java Database Connectivity*, que é um conjunto de classes e interfaces escritas em Java que fazem o envio de instruções SQL para qualquer base de dados relacional.

Nas aplicações Rails, a linguagem usada é a linguagem Ruby pelo que é necessário conseguir que o Active Record consiga interagir com a base de dados. Na Figura 2.6 é possível ver os adaptadores necessários para que tal aconteça.

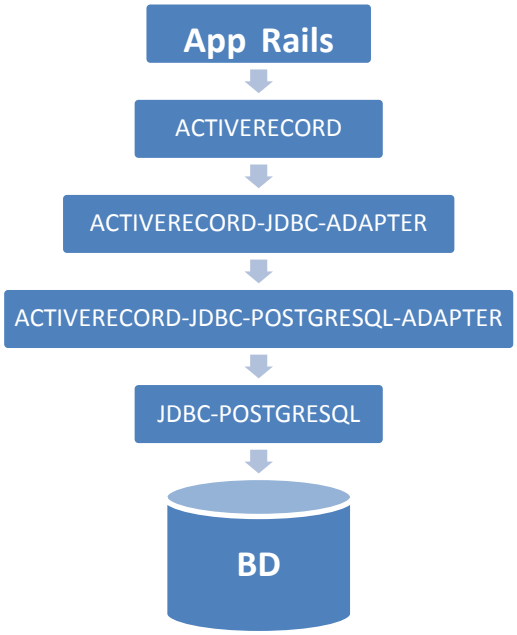

Figura 2.6 - Interação entre o Active Record e as diferentes bases de dados [JBoss.org , 2013]

Seguindo a Figura 2.6, que apresenta um exemplo para uma base de dados PostgreSQL, pode verificar-se que as aplicações Rails só interagem com a biblioteca ORM Active Record, como já foi anteriormente referido. O Active Record, por sua vez, é que interage com os adaptadores.

ACTIVERECORD-JDBC-ADAPTER é um adaptador de bases de dados para o componente Active Record, que pode ser usado com JRuby[15]. Este adaptador permite a interação entre praticamente qualquer base de dados compatível com JDBC e uma aplicação JRuby ou Rails. Embora suporte todas as bases de dados disponíveis através JDBC, é necessário o *download* e a configuração manual do driver JDBC do fornecedor da base de dados que se pretende usar (neste exemplo, JDBC-POSTGRESQL).

ACTIVERECORD-JDBC-POSTGRESQL-ADAPTER é um driver Active Record para PostgreSQL usando JDBC correndo sobre JRuby.

JDBC-POSTGRESQL permite que programas em Java se conectem a uma base de dados PostgreSQL usando código *standard* e independente da base de dados. É uma aplicação Java.

[15] Implementação da linguagem Ruby na plataforma Java.

Conclusão

O Ruby on Rails é uma linguagem muito poderosa, que envolve vários componentes, contribuindo assim que seja cada vez mais fácil e intuitiva a programação. Esta linguagem, sendo relativamente recente, está a crescer e temos como exemplo algumas aplicações conhecidas feitas em Ruby on Rails [Rubyonrails, 2013]:

- Twitter;
- Yellow Pages;
- Scribd;
- Shopify;
- Groupon;
- Jango
- Etc.

3 Gestão de dados históricos

Após a entrada das aplicações em produção, o número de registos na base de dados começa a aumentar de forma linear ou exponencial. Por exemplo, pode pensar-se na base de dados de um banco, de um hipermercado, de uma operadora de comunicações, que pode ter milhares, ou até mesmo, milhões de registos por dia.

Consequentemente, com este aumento exponencial de dados, vem a ineficiência na procura, pois o processamento de dados torna-se mais lento.

Com este aumento, se se pensar no uso dos dados após a sua inserção na base de dados, chega-se à conclusão que inicialmente são acedidos com muita frequência. Contudo, à medida que a idade dos dados aumenta, a frequência com que são acedidos diminui, chegando a ser insignificante ou mesmo nula.

Por exemplo, num sistema que engloba faturação, é provável que faturas com mais de um ano comecem a não ser acedidas com tanta frequência como faturas com um mês, possivelmente pelo motivo de ter passado a altura de reclamar algum assunto relacionado com um possível engano.

Portanto a maioria das organizações encontra-se nesta situação, em que a maior parte dos seus utilizadores acedem a dados atuais, e muito poucos utilizadores acedem a dados mais antigos.

Assim, segundo a Oracle, os dados podem ser descritos como sendo: ativos, menos ativos, históricos e prontos param serem arquivados.

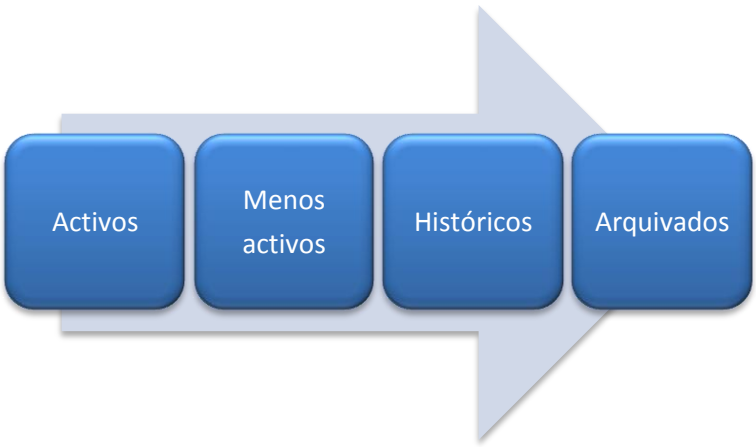

Figura 3.1 - Ciclo de vida dos dados

Podemos contrastar o aumento dos dados ao longo do tempo ao mesmo tempo com a diminuição da sua atividade, como se pode ver na Figura 3.2.

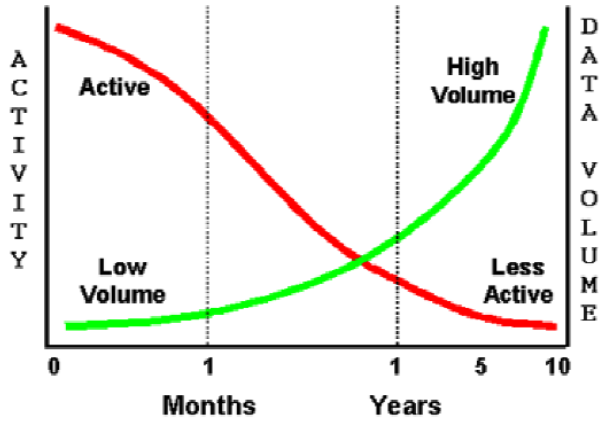

Figura 3.2 - Aumento do volume de dados versus atividade [Hobbs, 2007]

Face ao constante aumento de dados e em contrapartida à diminuição da sua utilização ao longo do tempo, são necessárias técnicas que garantam sempre a disponibilidade dos mesmos com eficiência.

Uma das técnicas usadas por muitos sistemas é o particionamento de dados, a qual é abordada na secção seguinte.

3.1 Particionamento de dados

Particionamento de dados é uma técnica que visa melhorar o desempenho das consultas sobre as bases de dados, permitindo que uma tabela, índice ou tabela organizada por índices seja subdividida em partes menores [Baer, 2007].

Cada parte do objeto de base de dados (tabela, índice ou tabela organizada por índices) é uma partição, que possui um nome próprio, e opcionalmente características próprias de armazenamento.

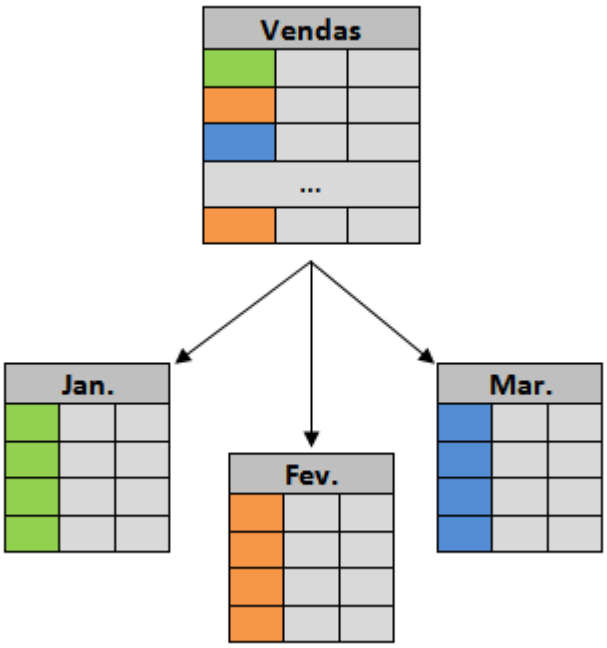

Figura 3.3 - Tabela particionada na perspetiva de uma aplicação

O mecanismo de particionamento determina a divisão dos dados em partições com base no critério estabelecido sobre os valores de uma ou mais colunas de um objeto de base de dados, conjuntamente denominadas de chave de particionamento.

Por exemplo, a tabela de vendas mostrada na Figura 3.3 está particionada por intervalos de datas de venda, em que o critério de particionamento é por mês, mas aparece a qualquer aplicação como uma tabela, pois os objetos particionados são totalmente transparentes para os sistemas de informação e instruções DML - *Data Manipulation Language*[16] padrão.

Existem vários métodos de particionamento pelos quais podemos optar:

- Particionamento por intervalo (*Range Partitioning*): as partições são criadas de acordo com um intervalo de valores. Seguindo o exemplo Figura 3.3 o intervalo de particionamento é o mês;
- Particionamento por lista (*List Partitioning*): a distribuição de dados é realizada por uma lista de valores da chave de particionamento. Por exemplo, poderia particionar-se a tabela de vendas da Figura 3.3, não por mês mas por regiões de venda;

[16] A linguagem SQL é uma linguagem DML *(Data Manipulation Language)*.

- Particionamento por *hash* (*Hash Partitioning*): é aplicada à chave de particionamento uma função *hash*[17]. Ao contrário dos dois métodos anteriores, neste não permanece qualquer lógica entre uma partição e os seus dados.

Uma tabela pode ser particionada recorrendo a particionamento composto, em que se utiliza a combinação de dois dos métodos acima descritos, podendo ser combinados da seguinte forma: intervalos-*hash*, intervalo-lista, intervalo-intervalo, lista-intervalo, lista-lista e lista-*hash*. Nestes casos, primeiro, a tabela é particionada pelo primeiro método, sendo depois cada partição ainda dividida em subpartições, aplicando o segundo método

Quanto às operações de consulta sobre tabelas particionadas, quando uma é efetuada, o optimizador determina a partição envolvida – característica denominada de *Partition Pruning* –, desde que a operação esteja condicionada pela chave de partição.

Quando uma operação de consulta não é realizada pela chave de particionamento, a mesma será feita em todas as partições da tabela correspondente, tendo assim que ser efetuado o mesmo tempo de processamento que se efetuaria se os dados não estivessem particionados, ou seja, estivessem todos numa só tabela.

Aliás, nesta situação o tempo de processamento será ligeiramente superior ao de uma tabela não-particionada, devido ao acréscimo de processamento necessário, que passa por exemplo por ser necessário um acesso mais complexo ao dicionário de dados da base de dados, ou até mesmo pelo facto de que num modelo particionado as partições podem estar distribuídas fisicamente por diferentes ficheiros de dados, o que traz um maior peso de processamento ao nível de uso do disco (com as operações I/O – *input/output*).

Ainda assim, a técnica de particionamento traz grandes vantagens para a maioria das aplicações, pois melhora a capacidade de gestão, o desempenho e a disponibilidade.

[17] Segundo [Horstmann, 2010], uma função *hash* é uma função que calcula um valor inteiro, o código *hash*, a partir de um objeto, de tal forma que diferentes objetos possivelmente reproduzirão códigos *hash* diferentes.

4 Análise do problema

A realização do trabalho proposto apresenta vários desafios: conhecer a linguagem Ruby e a *framework* Ruby on Rails; perceber a arquitetura usada, a MVC; estudar o ORM fornecido com o Ruby on Rails, o Active Record.

A manipulação de uma base de dados relacional é, tipicamente, feita através de instruções DML e em particular através do SQL. No caso do Ruby o acesso é feito por via do Active Record que disponibiliza um conjunto de métodos para acesso a uma base de dados relacional.

Ao longo deste capítulo é exposta a forma como o Active Record permite a interação e o acesso ao conteúdo de uma base de dados, explicando as principais formas de evocar as funções que implementam os métodos CRUD – Create, Read, Update e Delete.

4.1 Problema concreto e restrições

O principal objetivo desta dissertação é criar uma ferramenta de gestão de dados históricos, usando o particionamento de dados na base de dados.

A linguagem requisitada pela empresa PT-Inovação foi a linguagem Ruby, utilizando a plataforma Ruby on Rails. Apesar de a maioria dos SGBDs comerciais suportarem particionamento de dados, um dos requisitos colocados pela empresa é a necessidade de a ferramenta ser independente do SGBD usado na aplicação, pelo que deverá ser usado o ORM fornecido com o Ruby on Rails, o Active Record, para conseguir implementar esta abstração.

O Active Record não suporta particionamento de dados, pelo que deverá ser desenhada e implementada uma solução que, juntamente com o Active Record, permita particionar um modelo de dados.

A solução não será para uma aplicação específica, pois desta forma a mesma poderia ser otimizada, considerando o tipo específico de dados usados pela aplicação. A solução

deverá ser uma biblioteca que possa ser usada em qualquer aplicação, com um qualquer tipo de dados, usando um qualquer SGBD.

O Active Record disponibiliza os métodos CRUD, que suportam ações de criação, leitura, atualização e eliminação de registos de uma base de dados. A ferramenta desenvolvida dá suporte a estes métodos para um modelo particionado. Antes da implementação foi necessário fazer um levantamento de exemplos de utilização dos métodos CRUD.

4.2 Métodos CRUD

CREATE

Dado que no Active Record as tabelas são representadas como classes (modelos) e as linhas como objetos, uma linha é criada quando é criado um objeto da classe associada.

Existem duas formas principais de o fazer: através do método *new* e do método *create*. O método *new* cria um novo objeto na memória, sendo necessário usar o método *save* para guardar esse novo objeto na base de dados. Esta primeira alternativa permite três formas de implementação. A primeira consiste em chamar o método *new* sem qualquer parâmetro e posteriormente preencher os valores dos atributos (correspondente às colunas da base de dados).

```
user = User.new
user.name = "David"
user.occupation = "Code Artist"
user.save
```

A segunda forma permite fazer exatamente o mesmo mas recorrendo a um bloco de inicialização:

```
User.new do |u|
  u.name = "David"
  u.occupation = "Code Artist"
  u.save
end
```

Finalmente, a terceira e última forma de implementação do método *new*, é implementada passando uma *hash* de atributos:

```
user = User.new(:name => "David", :occupation => "Code Artist")
user.save
```

A outra alternativa para criar uma nova linha na base de dados usando o Active Record é através do método *create*, o qual instancia um novo objeto guardando-o simultaneamente na base de dados.

```
User.create(:name => "David", :occupation => "Code Artist")
```

O Active Record permite também que seja passado um *array* de *hashes* de atributos, o que criará múltiplas linhas na base de dados.

Verifica-se pelos exemplos anteriores que em nenhum deles é passado o atributo `id`, já que, como referido anteriormente, o Active Record cria automaticamente um valor único e define o atributo `id` assim que a linha é guardada.

READ

Ler dados de uma base de dados engloba sempre especificar algum critério de procura. Em SQL a declaração mais comum para pesquisas é o `select`, que no Active Record corresponde ao método `find`.

O método `find` tem diferentes formas de efetuar consultas. A forma mais simples é através da chave primária: o id. É possível passar um ou vários ids (chaves primárias), sob a forma de inteiro, *string*, ou *array*. Caso seja passado um só valor, é retornado o objeto que contém os dados da linha respetiva. Caso seja passado mais que um valor, é retornado um *array* de objetos.

```
Person.find(1)
Person.find("1")
Person.find(1, 2, 6)
Person.find([7, 17])
Person.find([1])
```

É sempre possível que não exista um registo na base de dados correspondente ao critério especificado, sendo neste caso lançada uma exceção *ActiveRecord::RecordNotFound*[18]. No caso de serem passados vários ids, basta um não ser encontrado para a exceção ser lançada.

A segunda forma de efetuar consultas utilizando o método *find* consiste em passar os argumentos *:first*, *:last, :all*, os quais permitem a obtenção do primeiro, do último ou de todos os resgistos, respectivamente.

```
Person.find(:first)
Person.find(:last)
Person.find(:all)
```

Esta variação do *find* pode ser simplificada, pois para estes argumentos, existem métodos que podem ser aplicados diretamente ao modelo, retornando exatamente os mesmos resultados.

```
Person.first
Person.last
Person.all
```

A terceira e última forma consiste em passar uma *hash* de opções, que podem corresponder a uma parte da declaração SQL *select*. As chaves válidas para o *hash* de opções são:

- `:conditions`
- `:include`
- `:order`
- `:select`
- `:group`
- `:joins`
- `:from`
- `:limit`
- `:offset`
- `:readonly`
- `:lock`

[18] http://api.rubyonrails.org/classes/ActiveRecord/RecordNotFound.html

```
Person.find(:first, :conditions => ["user_name = ?", user_name])
Person.find(:all, :order => "created_on DESC", :limit => 5)
Person.find(:all, :group => "last_name", :order => "name")
```

É possível combinar opções no mesmo *find*, como mostra o exemplo anterior, obtendo-se assim uma consulta mais específica.

Contudo, existe uma forma mais simples de aceder aos dados quando queremos efetuar procuras com outros critérios para além da chave primária. Esta forma é conseguida recorrendo aos *Dynamic Finders*. Estes métodos são gerados dinamicamente pelo Active Record para cada coluna da tabela correspondente a cada classe (modelo) e podem ser utilizados chamando os métodos dinâmicos *find_by_X* ou *find_all_by_X*, adicionando as colunas que se pretende consultar.

```
Person.find_by_name("Mary")
Person.find_all_by_name("Mary")
```

A diferença entre o método *find_by_* ou *find_all_by_* é que no primeiro é retornado o primeiro registo encontrado que responda ao critério passado, enquanto no segundo são retornados todos os registos.

Além destes dois métodos, também é possível procurar em múltiplas colunas, utilizando os métodos dinâmicos *find_by_X_and_Y* ou *find_all_by_X_and_Y*.

```
Person.find_by_name_and_age("John", 28)
Person.find_all_by_name_and_age("John", 28)
```

Assim como no método *find*, os *dynamic finders* também permitem a passagem de opções como seu último argumento, como por exemplo *:order* e *:limit*.

Adicionalmente é permitido outro tipo de cenário. Por vezes pode ser necessário criar o objeto quando não é encontrado. Por exemplo se tivermos os modelos *Product* e *Category*, e admitindo que um produto pertence a uma determinada categoria, para se garantir que o produto fique efetivamente com uma categoria e que não se insira nenhuma categoria repetida aquando a inserção de um produto, pode usar-se o método *find_or_create_by_X*.

```
p = Product.new(:name => @product_name)
p.category = Category.find_or_create_by_name(@category_name)
p.save
```

A última alternativa ao método *find* e até mesmo aos *dynamic finders*, é aplicar o método *where* diretamente ao modelo.

```
Person.where(:name => "John")
Person.where("name = 'John' AND age>20")
Person.where("name = ? AND age = ?", @name, @age)
```

Com este método é possível ainda manipular os registos retornados, utilizando algumas das opções que se apresentou no método *find* e combinando-as se necessário.

```
Person.where("age>20").order("id ASC")
Person.where("age>20").limit(10)
Person.select("name").where("age>20").group("last_name")
```

UPDATE

Depois de criado um novo registo é possível alterar um ou mais atributos e atualizá-lo. Esta ação é muito semelhante à da criação de um novo registo, no que diz respeito ao uso do método *save,* mas existem outras alternativas ao *save* para alterar um ou mais atributos: *update_attribute* e *update_attributes*.

Inicialmente é necessário saber qual o objeto a atualizar, pelo que é inevitável recorrer ao método *find*.

```
p = Person.find(1)
p.name = "Mary"
p.save
```

```
p = Person.find(1)
p.update_attribute(:name, "Mary")

p = Person.find(1)
p.update_attributes(:name => "Mary", :age => 25)
```

Estes métodos podem ser combinados nos métodos de classe *update* ou *update_all*. O método *update* leva um id e um conjunto de atributos, enquanto o *update_all* atualiza todos os registos com os atributos passados como parâmetro.

```
Person.update(1, :name => "Mary")
Person.update_all(:flag => 1)
```

DELETE

Frequentemente é necessário apagar dados de uma determinada tabela. O Active Record dispõe de duas formas de o fazer.

A primeira forma consiste no método *delete*. Como argumento pode ser passado um ou mais ids. Existe ainda o *delete_all*, que apaga todos os registos que cumprirem a condição passada como argumento.

```
Person.delete(1)
Person.delete([12,25,102])
Person.delete_all("age<15")
```

A segunda forma consiste no método *destroy*. Este método difere do delete, principalmente por existir a necessidade de se efetuar primeiro um *find* do registo que se pretende apagar.

```
p = Person.find(1)
p.destroy

Person.destroy(12)
Person.destroy([12,25,102])

Person.destroy_all(:status => "inactive")
```

Comparando os métodos *delete* e *destroy*, enquanto os métodos *delete* ignoram funções de *callback* e de validações, os métodos *destroy* asseguram que são chamadas. Apesar de os métodos *delete* serem mais rápidos por não chamarem estas funções, se se pretender uma base de dados consistente de acordo com as regras definidas nas classes dos modelos, é recomendado usar o *destroy*.

4.3 Abordagem

Sendo o objetivo deste trabalho disponibilizar uma ferramenta que permita a um programador de Ruby on Rails usar particionamento nos seus modelos, independentemente do SGBD utilizado, a abordagem consiste principalmente em desenvolver uma biblioteca que, quando chamada por uma classe (modelo) permita aplicar particionamento ao mesmo.

A abordagem seguida foi reescrever os métodos implementados pelo Active Record que auxiliam os métodos CRUD.

Tal é possível em Ruby, pois as classes são abertas, o que significa que é permitido a uma subclasse fornecer uma implementação específica de um método que já é fornecido por uma das suas superclasses.

Por exemplo, a classe *string* possui o método *to_s*, que faz com que um objeto seja apresentado como *string*.

```
user = User.new("John", 30)
u.to_s -> "#<User: 0x7c4ec2>"
```

A classe *User* possui (por herança da classe *Object*) o método *to_s*, mas este apresenta apenas a classe e o id do objeto. No entanto, este método pode ser reescrito por a classe *User*, para imprimir também os atributos do mesmo.

```
class User
   def to_s
        "User: #@name--#@age"
   end
end

user = User.new("John", 30)
u.to_s -> "User: John--30"
```

Sendo o Active Record uma biblioteca inteiramente escrita em Ruby, o que é possível fazer com objetos Ruby, também o é com objetos Active Record, ou seja, os métodos do Active Record podem ser reescritos.

Para o presente trabalho, a solução consiste então em criar uma biblioteca que reescreva para uma determinada classe (modelo neste caso) os principais métodos do Active Record que suportam os métodos CRUD.

Dado que é pretendido que a biblioteca funcione com qualquer aplicação, é necessário que a mesma saiba qual é a chave e o critério de particionamento que o programador pretende para um determinado modelo.

A chave de particionamento consiste no campo da tabela/modelo pelo qual se pretende particionar. O critério de particionamento consiste em especificar como será feito o particionamento, ou seja, para que partição irá cada registo. Por exemplo, particionando uma tabela de vendas pela data, a mesma pode ser particionada de diferentes formas: por mês, por mês do ano, por ano, por semana, etc.

Para permitir uma maior versatilidade no critério de particionamento, caberá ao programador definir a função de particionamento para cada modelo, que depois será chamada pela biblioteca sempre que for necessário calcular a partição destino de um registo. Para tal, terá que se definir um nome geral para esta função, pois esta tem que ser conhecida pela biblioteca, embora a sua definição seja diferenciada por modelo. Por exemplo, o modelo A poderá particionar a tabela por mês e o modelo B por ano.

5 Implementação

Neste capítulo é descrita a implementação da biblioteca que permite aos modelos do Ruby on Rails o uso de particionamento de dados. Primeiramente é descrito o cenário e ambiente de implementação, onde são mencionadas todas as tecnologias utilizadas e as respetivas versões, e de seguida são apresentadas as opções tomadas durante a implementação.

5.1 Descrição do cenário de implementação

Descrita a principal finalidade desta dissertação de mestrado, no capítulo 1, destaca-se que a mesma tem como requisito o desenvolvimento de uma solução/biblioteca, que permita adicionar aos modelos do Ruby on Rails a funcionalidade de particionamento de dados, que seja independente do SGBD usado.

Para o desenvolvimento da biblioteca, foi necessário trabalhar com um SGBD específico. O SGBD utilizado durante todo o desenvolvimento foi o PostgreSQL, versão 9.1. Juntamente com o PostgreSQL, utilizou-se a ferramenta gráfica pgAdmin versão 1.16.0.

A linguagem de programação, já implícita (como requisito), foi a linguagem Ruby versão 1.9.3, com a plataforma Rails, versão 3.2.13, em sistema operativo Windows 7. Para tal recorreu-se ao IDE - *Integrated Development Environment* NetBeans versão 7.3.

5.2 Implementação realizada

Nesta secção é explicado o procedimento usado para a implementação da biblioteca para particionamento de dados num determinado modelo.

Sendo um dos objetivos deste trabalho a solução ser independente do SGBD utilizado pelo programador, a solução consiste na implementação de uma biblioteca Ruby, que quando chamada pelo programador num determinado modelo, faça o necessário durante a execução da aplicação de forma a permitir particionamento dos dados. Para cumprir o

objetivo deste trabalho, a biblioteca têm que reescrever as funções do Active Record necessárias aos métodos CRUD.

5.2.1 Importar a biblioteca num determinado modelo

Tratando-se de uma biblioteca, a sua utilização segue o modelo tradicional, em que esta é adicionada à aplicação colocando-a numa diretoria específica.

Para o programador poder utilizar a biblioteca num modelo tem que:

1. Colocar a biblioteca na subdiretoria /lib;
2. No ficheiro `application.rb`, que se encontra na subdiretoria /config, acrescentar:

```
config.autoload_paths += %W(#{config.root}/lib)
```

Ou então, no próprio modelo que se pretende particionar:

```
require 'partitioning'
```

3. No modelo acrescentar:

```
include Partitioning
```

Desta forma, a biblioteca fica associada à aplicação, podendo ser chamada por um ou mais modelos, tendo cada modelo uma instância da mesma.

5.2.2 Desenvolvimento da biblioteca

A biblioteca foi desenvolvida com o objetivo de reescrever as funções do Active Record, permitindo que as aplicações usem uma Base de Dados relacional em que uma ou

mais entidades possam ser particionadas de forma transparente, isto é, sem que a aplicação seja dependente de uma tecnologia específica de um dado SGBD.

Inicialmente constatou-se que entre os métodos do Active Record necessários à resolução do problema, existem métodos de instância e métodos de classe. Por exemplo, seguindo os métodos CRUD, apresentados no capítulo 4, para criar um registo pode ser usado:

1. New & save
2. Create

Na opção 1, o método do Active Record *save*, é um método de instância, pois é aplicado a uma instância do objeto, ou seja:

```
user = User.new
...
user.save
```

Na opção 2, o método *create* é um método de classe, pois é aplicado diretamente ao objeto.

```
User.create(...)
```

Foi decidido dividir os métodos reescritos em dois submódulos dentro da biblioteca: um módulo para métodos de instância e um módulo para métodos de classe.

Para se adicionar os métodos de um módulo como métodos de instância ou de classe a um determinado modelo, o mesmo tem que invocar as primitivas *include* e *extend*, respetivamente. Com o objetivo de não deixar esta tarefa a cargo do programador, para assim tornar o uso da biblioteca o mais simples e transparente possível, foi decidido chamar estas primitivas na própria biblioteca, onde se encontram os módulos. Quando um determinado modelo faz o *include* da biblioteca, recorre-se à função *callback self.included*, que é executada de forma automática pelo Ruby, para incluir nesse modelo os métodos de instância e estender os métodos de classe. A estrutura da biblioteca é apresentada na Figura 5.1.

```
1   module Partitioning
2
3     def self.included(klass)
4
5       klass.send :include, InstanceMethods
6       klass.send :extend,  ClassMethods
7       klass.send :include, InstanceAuxMethods
8       klass.send :extend,  ClassAuxMethods
9
10      klass.send :cattr_accessor, :by
11      klass.send :cattr_accessor, :original_table
12    end
13
14    module ClassMethods{...}
696
697   module InstanceMethods{...}
731
732   module ClassAuxMethods{...}
765
766   module InstanceAuxMethods{...}
839
840 end
```

Figura 5.1 - Estrutura da biblioteca

Com a *callback self.included* consegue-se aplicar ao modelo que faz o *include* da biblioteca os *extends* e os *includes* dos métodos reescritos. Para auxiliar a reescrita destes métodos foi decidido criar dois módulos auxiliares, onde as funções neles definidas contém código que é usado em mais que um método.

Visto o objetivo de reescrever os métodos do Active Record ser o uso de particionamento, é necessário que o programador passe a chave de particionamento e o respetivo critério.

A chave de particionamento tem obrigatoriamente que ser passada à biblioteca. Em cada modelo, após fazer o *include* da biblioteca, é necessário invocar a função ***is_partitioned***, passando como argumento a respetiva chave.

O critério de particionamento é definido pelo programador, em função do módulo e do seu modelo de dados. A implementação deste critério é feita pelo próprio programador, que implementa a função ***calc_part_name***, que recebe como argumento o valor da chave de particionamento e devolve o nome da partição.

Dado o carater genérico da biblioteca, o nome da função que implementa o critério de particionamento foi definido no momento da sua implementação. Assim, a implementação desta função é obrigatória em cada modelo que use a biblioteca.

Na Figura 5.2 é apresentado um exemplo da utilização da biblioteca. O modelo é particionado pelo campo "date" (linha 4) e a função que implementa o critério de particionamento calcula o nome da partição em função do nome da tabela original e do mês e do ano da data que recebe como argumento.

```
1  class User < ActiveRecord::Base
2
3    include Partitioning
4    is_partitioned("date")
5
6    def self.calc_part_name(value)
7      partition_name = "#{self.table_name}_
8                        #{value.to_date.strftime("%m%Y")}"
9      return partition_name
10   end
11
12 end
```

Figura 5.2 - Exemplo de uma classe de um modelo particionado

Através da função ***is_partitioned*** a biblioteca sabe qual o campo de particionamento a ser usado pelo modelo a partir daquele instante. Dado que este campo é necessário em diferentes métodos, o seu valor é guardado na biblioteca, como uma variável da própria classe.

Cada modelo possui a sua própria instância da biblioteca. Assim, a variável de classe que guarda o campo de particionamento é guardada numa dada instância da biblioteca e, desta forma, o seu valor fica associado a cada classe (modelo).

De forma similar, o nome original da tabela também é guardado como uma variável de classe. Guardar o nome original da tabela é necessário para implementar a biblioteca, dado que a aplicação invoca as funções dos métodos CRUD especificando o nome da tabela original. Por exemplo, é necessário para o método `from`, pois usa sempre o nome original da tabela para todos os métodos que necessitam de construir o nome da partição. O nome da partição é sempre construído a partir do nome original da tabela. Quando uma pesquisa não recebe o campo de particionamento, para se percorrer todas as partições pertencentes ao

modelo em questão, testa-se se cada tabela obtida da conexão contém no seu nome o nome da tabela original.

Para tudo isto ser possível, no final de cada operação que altere o nome da tabela do modelo, é novamente alterado o mesmo para o nome da tabela original.

5.2.3 Métodos reescritos

Realizou-se uma análise do código de alguns módulos do Active Record, com o objetivo de identificar os métodos internos que implementam os métodos CRUD. Os módulos ou classes com métodos do Active Record essenciais a este trabalho são:

1. *Persistence:*
   - `save`
   - `save!`
   - `destroy`
2. *FinderMethods:*
   - `from(value)`
   - `find(*args)`
   - `all(*args)`
   - `first(*args)`
   - `first!`
   - `last(*args)`
   - `last!`
   - `exists?(id = false)`
   - `count(column_name = nil, options = {})`
3. *Relation:*
   - `destroy(id)`
   - `destroy_all(conditions = nil)`
   - `update(id, attributes)`
   - `update_all(updates, conditions = nil, options = {})`

4. *QueryMethods:*
   - `where(opts, *rest)`
5. *DynamicMatchers:*
   - `method_missing(method_id, *arguments, &block)`

Nas próximas secções será explicado qual ou quais destes métodos foram reescritos para cada ação CRUD.

5.2.3.1 Create

No método *Create* é possível utilizar a combinação *new & save* ou utilizar diretamente o *create.* Verificou-se que no seu fluxo de execução, em ambas situações, o sistema acaba por executar o método *save* do Active Record.

É no método *save* que de facto se guarda o registo na base de dados. De forma a existir particionamento, antes de guardar o registo, é necessário determinar a tabela de particionamento e alterar a execução deste método de forma a garantir que o registo é guardado na partição respetiva.

Antes de se guardar um registo é necessário:

1. Saber qual o campo pelo qual o programador pretende particionar;
2. Obter o valor do campo de particionamento do objeto que se pretende guardar;
3. Calcular o nome da partição para o valor obtido (*calc_part_name*);
4. Criar partição, caso esta não exista;
5. Forçar o nome da partição como tabela destino do objeto;
6. Chamar o método original do Active Record.

Para se saber qual o campo de particionamento dentro do método *save* recorre-se à variável de classe **`:by`**, que cujo valor foi definido na função *is_partitioned*, fazendo apenas:

```
part_key = self.class.by
```

O *save* é um método de instância pelo que é necessário obter a classe da respetiva instância para se poder obter o valor da variável de classe. O mesmo procedimento acontece para todos os outros métodos nos quais seja necessário obter o campo de particionamento, com a diferença que, para métodos de classe basta fazer:

```
part_key = self.by
```

Uma partição é uma tabela que apresenta a mesma estrutura que a tabela original do modelo. Ou seja, todas as partições têm necessariamente que ser constituídas pelas mesmas colunas que a tabela original.

Considerou-se recorrer ao método INHERITS, para que as partições herdassem todas as características e colunas da tabela original. Constata-se que usando este procedimento os dados inseridos nas partições são também inseridos na tabela original, contrariando o que se pretende. Por isso, resolveu-se criar as partições recorrendo ao CREATE TABLE.

É criada uma nova tabela (partição) com todos os campos existentes na tabela original, exceto os campos `id` e `timestamps` (*created_at & updated_at*), que são automaticamente acrescentados à tabela do modelo pelo Active Record.

Cada tabela (partição) possui um campo ID que é chave primária, que é gerado de forma automática pelo Active Record, é do tipo *auto-increment* e inicia a sua contagem sempre a partir do valor 1. Desta forma, a chave primária é única dentro de cada partição mas não é única no modelo global. Cada tabela (partição) possui a mesma numeração sequencial para a sua chave primária.

O ID do registo deve ser sequencial ao longo do modelo e não dentro de cada partição, como se pode ver na Figura 5.3.

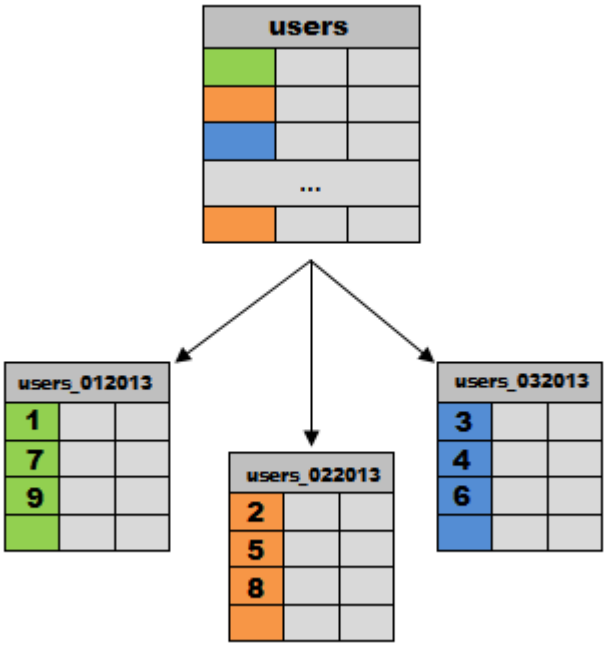

Figura 5.3 - IDs sequnciais dentro de um modelo particionado

Foi considerada a possibilidade de recorrer à criação de sequências, tal como existe no SGBD PostgresSQL. Dado a sua existência e manipulação ser específico de cada SGBD decidiu-se não usar as referidas sequências e implementar uma solução que recorre apenas à linguagem SQL *standard*.

A necessidade de possuir um ID global foi suprimida criando uma tabela que guarda os IDs de todos os modelos particionados. A tabela `sequences` apresenta a seguinte estrutura:

Campo	Tipo	Chave primária?	Não nulo?
`id`	Integer	Sim	Sim
`model`	Varchar(255)	Não	Sim
`seq`	Integer	Não	Sim

Tabela 5.1 - Modelo de dados da tabela '*sequences*'

O campo `id` é chave da tabela e é criado de forma automática pelo Active Record (tal como acontece para todas as tabelas), o campo `model` guarda o nome do modelo particionado e o campo `seq` guarda o valor do ***id*** do último registo criado para o respetivo modelo.

A implementação desta solução só é possível porque o Active Record permite que se especifique o ID do registo que está a ser criado. Quando é chamada a função *save*, este já é feito sobre o novo objeto, o registo a ser guardado, e o ID a ele associado pode ser definido antes de este ser guardado, fazendo:

```
self.id = seq + 1
```

O valor do ID atribuído a um registo é o valor guardado na tabela auxiliar `sequences` incrementado de uma unidade. Depois do registo ser guardado é necessário atualizar o contador atual, ou seja, a linha correspondente ao modelo na tabela auxiliar.

Antes de invocar o método original do Active Record que insere o registo na base de dados, é necessário definir a tabela de particionamento onde o registo é de facto guardado. O Ruby permite que se defina o nome da tabela através da instrução:

```
self.class.table_name = partition_name
```

Desta forma, o registo é inserido na partição, tal como num modelo não particionado um registo é inserido numa qualquer tabela. Após definir o nome da partição resta invocar o método original do Active Record para registar os dados no SGBD.

O método `save!` é igual ao método `save`, com a diferença que, caso o registo não seja guardado com sucesso é lançada uma exceção *ActiveRecord::RecordNotSaved*[19].

5.2.3.2 Read

No Active Record, as funções relativas ao método *Read* que foram reescritas são:

- `find(*args)`
- `method_missing(method_id, *arguments, &block)`
- `all(*args)`
- `first(*args)`

[19] http://api.rubyonrails.org/classes/ActiveRecord/RecordNotSaved.html

- `last(*args)`
- `exists?(id = false)`
- `from(value)`
- `count(column_name = nil, options = {})`
- `where(opts, *rest)`

No geral, para os métodos de pesquisa a abordagem principal é comum: se a pesquisa é realizada pelo campo de particionamento, é possível ir diretamente à partição correspondente. Caso contrário, é necessário percorrer todas as partições relativas ao modelo, chamando para cada partição o método nativo do Active Record. Tal é conseguido através da primitiva *super*, que invoca um método com o mesmo nome do método corrente na superclasse da classe corrente.

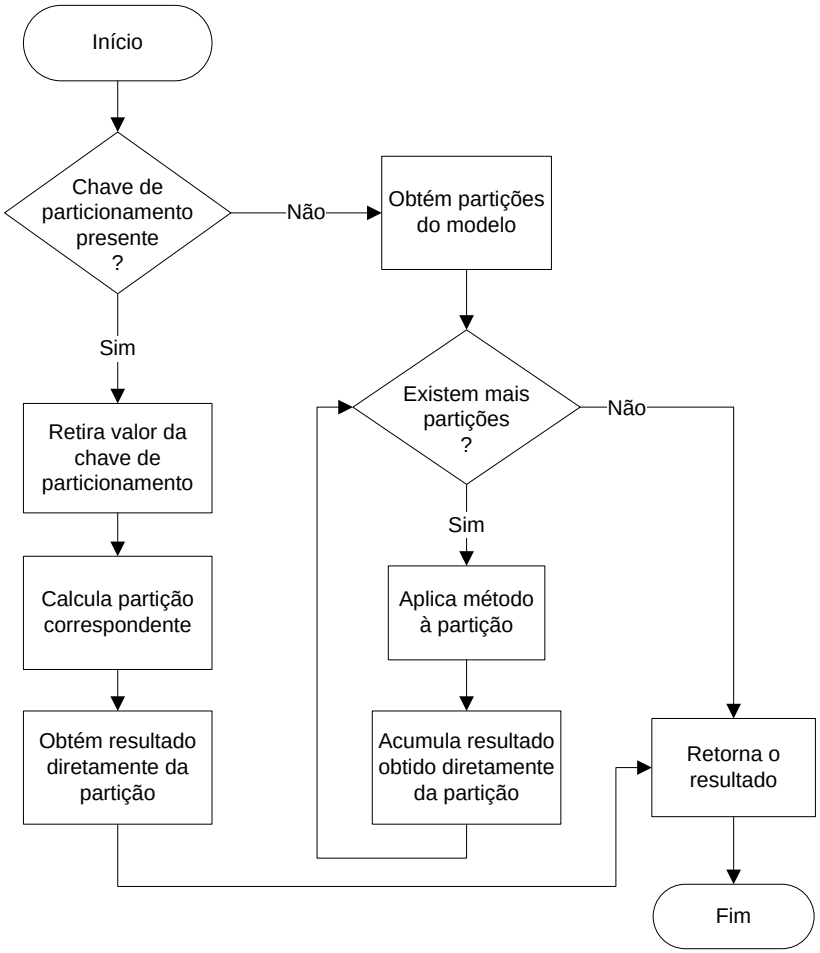

Figura 5.4 - Abordagem geral de um sistema de particionamento

O fluxograma presente na Figura 5.4 ilustra a abordagem geral de uma pesquisa em qualquer sistema de particionamento. Por exemplo, num modelo chamado `User`, particionado pela data de registo dos utilizadores, se a pesquisa pedir todos os utilizadores registados numa determinada data, com idade superior a 60, é calculada a partição e obtém-se o resultado diretamente da mesma. Se a pesquisa pedir todos os utilizadores registados com idade superior a 60, sem incluir uma data, a pesquisa tem que ser aplicada a todas as partições do modelo em questão, o que gera um grande *overhead*. Porém não existe solução para este caso, sendo desta forma que se comportam todos os sistemas de particionamento.

No entanto, no Active Record existem dois tipos de funções de pesquisa: funções que devolvem todos os registos existentes na base de dados que satisfazem um determinado conjunto de critérios e funções que devolvem apenas um registo (mesmo que na base de dados existam muitos outros registos que satisfaçam as condições especificadas). A Figura 5.4 ilustra o funcionamento do sistema no caso de uma função de pesquisa que retorna todos os registos correspondentes às condições passadas como argumento. Este é o comportamento das seguintes funções:

- All;
- Where;
- Count;
- Find_all_by_X / Find_all_by_X_and_Y.

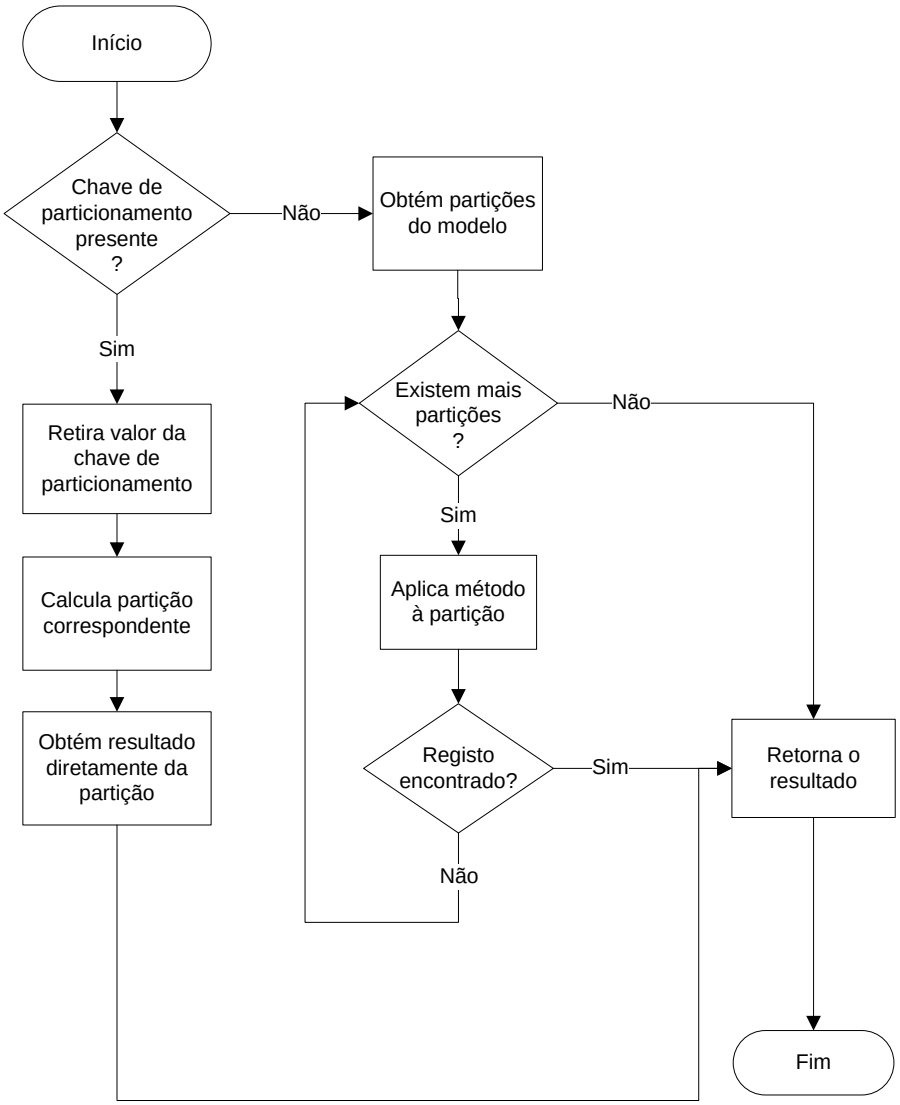

Figura 5.5 - Abordagem de uma pesquisa Active Record que retorna apenas um registo

Na Figura 5.5 é apresentado o funcionamento da biblioteca quando ocorre uma pesquisa através de um método que retorna logo após encontrar um primeiro registo. Este comportamento é característico das seguintes funções:

- First;
- Last;
- Exists;
- Dynamic Finders - Find_by_X/ Find_by_X_and_Y.

Note-se que nesta situação a pesquisa é interrompida mal é encontrado um registo, ao contrário do que acontece nos métodos que devolvem todos os registos e cujo funcionamento é apresentado na Figura 5.4.

A função *find* tem um comportamento diferente dos dois tipos de métodos referidos anteriormente. No caso de ser um *find* de um único ID, assim que o registo é encontrado, a pesquisa é terminada, pois o ID é chave primária e consequentemente é único. No caso de ser um *find* de vários IDS, a pesquisa retorna quando forem encontrados todos os IDS, tendo sempre que percorrer todos os registos para encontrar os que correspondem aos IDs passados como argumento.

Todos os métodos do Active Record que implementam os métodos CRUD têm comportamentos e tipos de retorno diferentes.

First &Last

```
first (*args)
last (*args)
```

Os métodos `first` e `last` são métodos que retornam um só registo. Como os próprios nomes indicam, o *first* retorna o primeiro registo e o *last* o último, mas não da mesma forma.

De acordo com as versões do Ruby e do Rails usadas (1.9.3 e 3.2.13, respetivamente), o *first* retorna o primeiro registo obtido através da *query*:

```
SELECT * FROM users LIMIT 1
```

O método *last* retorna o último registo de acordo com a *query*:

```
SELECT * FROM users ORDER BY id DESC LIMIT 1
```

As mesmas *queries* podem ser vistas na Figura 5.6, que consiste na captura de uma imagem da linha de comandos quando é invocado os métodos *first* e *last* sobre um modelo não particionado chamado `unpartit_users`.

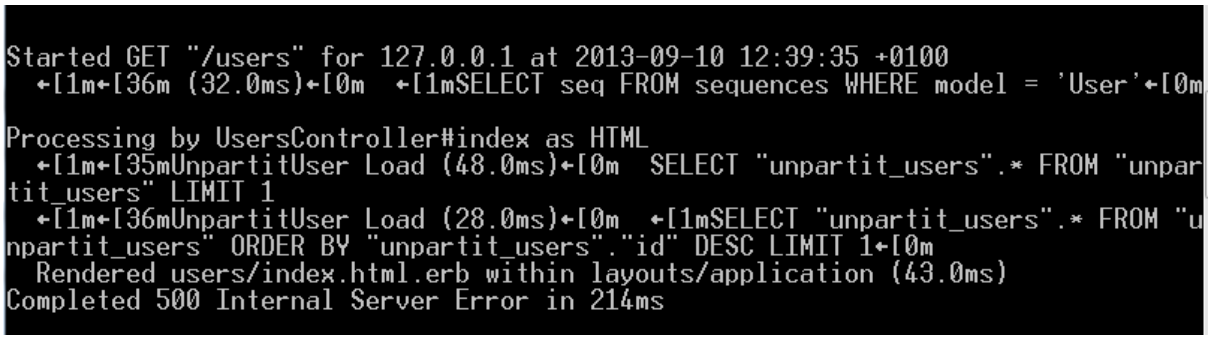

Figura 5.6 - *Query* dos métodos *first* e *last*

As *queries* implementadas em cada um destes métodos do Active Record têm critérios diferentes, o que gera alguma polémica sobre o comportamento do método *first* [Stack Overflow Questions, 2013]. Neste *website* existe alguma polémica sobre o comportamento do *first*, dizendo-se que o mesmo é incorreto. Sendo que o *first* e o *last* devem ser o contrário um do outro, o *first* deveria ser corrigido para ter o comportamento contrário ao *last*.

Foi decidido implementar o *first* de forma a que a sua implementação seja o contrário da do método *last*. Assim, os métodos *first* e *last* retornam os registos com menor e maior ID, respetivamente.

Num modelo particionado, torna-se mais complicado averiguar qual o registo com menor ou maior ID, pois num modelo não particionado só existe uma tabela, enquanto num modelo particionado a tabela principal "divide-se" em várias tabelas (partições), podendo estes registos estar em qualquer uma delas.

A abordagem principal utilizada nestes métodos é obter o mínimo/máximo ID de cada partição, ficando no final com um *array* que contém os IDs mínimos/máximos de todas as partições, sendo depois apenas necessário obter o mínimo/máximo ID do *array*.

Where

```
where (opts, *rest)
```

O método `where` foi o método ao qual foi introduzido mais processamento devido a permitir receber como parâmetro uma *hash* ou *string*, sendo sempre necessário detetar à partida que tipos de argumentos foram passados como argumento. A maior dificuldade encontrada durante a reescrita deste método do Active Record está presente no caso de se receber uma *string*.

Sendo passada uma *string* como parâmetro, é necessário averiguar se a mesma contém a chave de particionamento e o respetivo valor. Como se pode ver na declaração deste método e nos exemplos do mesmo no capítulo 4, o valor dos atributos pode não constar na *string*, mas sim ser passado em variáveis, como conteúdo de `*rest`.

Para efetuar estas verificações é necessário separar a *string* de argumentos por *`AND`* ou *`and`*.

Num modelo não particionado, este método percorre toda a tabela, retornando no final todos os registos que obedecem às condições passadas. Num modelo particionado, de forma semelhante, deve percorrer todas as partições, obtendo no final o mesmo resultado, ou, caso a chave de particionamento esteja contida na *string* de argumentos, é percorrida apenas a partição correspondente.

Find

```
find (*args)
```

O método `find` foi um dos métodos mais complexos de reescrever devido aos diferentes tipos de argumentos que permite passar. Este método pode receber como parâmetro um ID, uma lista de IDs, um array de IDs e ainda as primitivas `first`, `last` e `all`.

Admitiu-se também a possibilidade de o programador indicar o nome da partição onde pretende pesquisar, através da opção `:from`. Esta opção é dada em todos os métodos de pesquisa possíveis, pois sendo o programador a decidir o critério de particionamento, conhece a nomenclatura dos nomes das partições.

Sendo necessário um grande bloco de processamento para validar todas estas opções, na Figura 5.7 pode ver-se todo o processo do *find* reescrito.

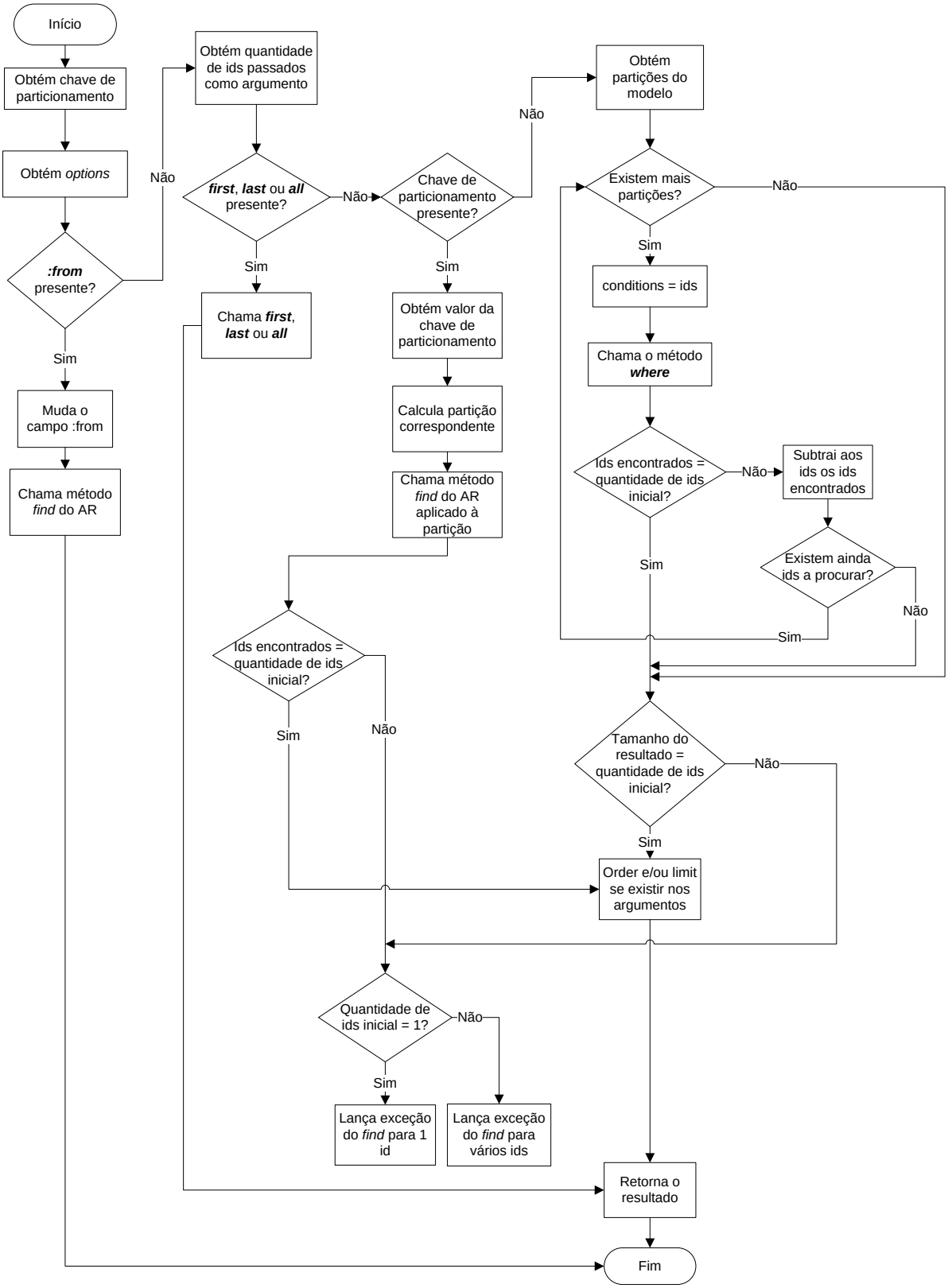

Figura 5.7 – Método *Find* reescrito

A maior dificuldade encontrada na reescrita deste método verificou-se quando o mesmo recebe como argumento vários IDs. Neste caso, como já foi referido no capítulo 4, se os registos referentes a todos os IDs passados como parâmetro não forem encontrados, é lançada uma exceção. Mas, num modelo particionado, onde existem várias partições, nunca se consegue prever em que partição ou partições se encontram os registos.

É necessário não lançar a exceção quando o método do Active Record é aplicado a uma partição, mas sim, lançá-la após percorrerem-se todas as partições. Para tal utilizou-se a primitiva `rescue`.

Por outro lado, é necessário obter em cada partição a quantidade de ***ids*** encontrados, para assim saber quantos faltam encontrar e para que após se percorrerem todas as partições se saiba se foram encontrados todos os registos, retornando-os, ou lançado a exceção.

Acontece que, quando no método *find* não são encontrados todos os registos não existe nenhuma forma de se conseguir saber se chegou a ser encontrado algum e em caso positivo quantos foram encontrados. Assim, foi decidido recorrer ao método `where`, porque neste método, quando não é encontrado nenhum registo é retornado um *Array* vazio, não lançando qualquer exceção, como acontece no *find*.

Como se pode ver no fluxograma da Figura 5.7, as primitivas `limit` e `order` não são passadas aos métodos, pois quando o método não recebe como argumento o campo de particionamento, como tem que percorrer partição a partição, chamando para cada o método nativo do Active Record, estas primitivas não retornariam o resultado esperado, pois seriam aplicadas à partição e não ao conjunto resultante de registos. Foi decidido aplicar estas primitivas ao resultado final. No entanto, é importante referir que o *order* retorna sempre o mesmo resultado que o método original do Active Record, enquanto que com o *limit* tal pode não acontecer.

Como exemplo, num *find* de todos os registos, com um limite de 100 registos, o método nativo do Active Record retorna os registos com os IDs de 1 a 100. Num modelo particionado, só pelo facto dos registos estarem em partições diferentes, possivelmente o resultado retornado já não é o mesmo, pois os registos são distribuídos pelas partições consoante o seu valor do campo de particionamento.

Se o *find* receber como argumento o campo de particionamento, este problema já não se verifica, pois só é analisada uma partição.

Dynamic Finders

```
method_missing(method_id, *arguments, &block)
```

Estes métodos fazem parte da "magia" do Ruby on Rails, pois não são declarados nem pré-definidos. São formados durante a execução de uma aplicação de acordo com os atributos de um modelo.

O método responsável pela formação destes métodos é o `method_missing`, que é chamado quando um método não é encontrado. O nome do método pretendido é recebido como `method_id`, os parâmetros do método pretendido são recebidos como `*arguments`. O campo `&block` após verificação constatou-se que não é necessário para a implementação deste trabalho.

Para reescrever os *finds* dinâmicos, é necessário verificar se o campo `method_id` é do tipo *find_by* ou do tipo *find_all_by*. Em caso negativo significa que não é um método que interesse reescrever, chamando o método `method_missing` do Active Record. Em caso positivo, é necessário averiguar se é um *find_by* ou um *find_all_by*, pois têm comportamentos diferentes.

O *find_by* funciona como um *find* normal, retornando o primeiro registo encontrado. O *find_all_by* funciona como o *where*, retornando todos os registos encontrados que obedecem às condições passadas.

A maior dificuldade encontrada na reescrita deste método foi no caso do *find_by*, que deve retornar o primeiro registo encontrado. O ideal seria chamar o *find*, mas não é possível devido ao tipo de argumentos que permite, tendo que ser passado sempre um ID ou outras primitivas.

Foi decidido utilizar o método *first*, admitindo que possivelmente não é retornado o mesmo resultado que no modelo não particionado, pois este método retornará só um registo como é pretendido mas será sempre o registo com menor ID.

Mesmo usando o método *all* com limite de 1 registo, o registo retornado possivelmente não é o mesmo que no modelo não particionado pela razão já explicada nesta secção, na implementação do método *find*.

Este facto não é relevante para um *find_by*, pois, à partida o programador nunca irá utilizar este método sabendo que existe mais que um registo, exceto se for utilizado como forma de verificar a existência de um registo por um determinado campo, funcionando assim como um `exists?`. Neste caso não interfere o resultado retornado não ser igual.

5.2.3.3 Update

Para o método CRUD `update` foram estudados três métodos do Active Record:

1. `update_attributes(attributes, options = {})`
2. `update(id, attributes)`
3. `update_all(updates, conditions = nil, options = {})`

A diferença entre os dois primeiros métodos é que o primeiro é de instância e o segundo é de classe, ou seja, o primeiro é aplicado a uma instância de um objeto, enquanto o segundo é aplicado a uma classe, tendo que indicar neste o ID do objeto que se pretende atualizar.

Não foi necessário reescrever o primeiro método, `update_attributes`, pois existe um ponto do seu fluxo de execução em que é chamado o método `save`, que foi previamente reescrito. Além da implementação explicada na reescrita do *save*, foi necessário acrescentar algum código para o *save* no caso de um *update*.

Devido ao mecanismo adotado na atribuição da chave primária, foi necessário verificar-se se a ação é um *update* ou um *create*, antes de se consultar a tabela auxiliar que implementa os contadores para o próximo ID de cada modelo e obter o ID, pois se for um *update*, não há atribuição nem incremento de ID, pois o registo já existe.

A solução passa por verificar se o campo ID está presente na instância que chega ao método *save*, pois quando se trata de um *create*, o campo ID apresenta-se com o valor NIL. NIL é um objeto como outro qualquer mas que representa a não existência de um valor.

Quanto ao segundo método, `update`, foi necessário reescrever, mas sem alterar nenhum código. A necessidade de o reescrever na biblioteca, mesmo sem ter alterado ou acrescentado algum código resulta do facto de este método receber o ID do registo que se pretende atualizar e chamar o `find` para o obter e aplicar o *update_attributes* ao objeto retornado do *find*.

Se este método não fosse reescrito na biblioteca, invocaria o *find* do Active Record, e não o *find* reescrito na biblioteca.

O último método, `update_all`, teve também que ser reescrito, alterando o seu código, pois recebendo a chave de particionamento ou a opção `:from` pode aplicar o *update_all* diretamente à partição. Tal é feito da mesma forma que no método *save*, alterando o nome da tabela do modelo e invocando posteriormente o método *update_all* do Active Record.

```
self.table_name = partition_name
```

No final, tal como no método *save*, deve definir-se o nome da tabela com o nome da tabela original do mesmo.

5.2.3.4 Delete

Para o método CRUD `delete` foram estudados três métodos do Active Record:

1. `destroy`
2. `destroy(id)`
3. `destroy_all(conditions = nil)`

Todos estes três métodos de suporte ao método `delete` foram reescritos, mas apenas o primeiro foi alterado. O `destroy`, é um método de instância, pelo que é aplicado a uma instância de um objeto, portanto, têm-se sempre acesso ao valor do campo de particionamento, podendo sempre ser aplicado diretamente à partição. O nome da partição é forçado da mesma forma que para o método *save* e o *update_all*, explicado anteriormente

nesta secção. Após o nome da partição estar definido como a tabela do modelo em questão, basta chamar-se o respetivo método do Active Record.

Os dois últimos métodos, `destroy(id)` e `destroy_all`, também foram reescritos mas exatamente com o mesmo código e apenas porque recorrem aos métodos `find` e `where`, respetivamente, e é necessário que sejam invocados os métodos reescritos e não os métodos nativos do Active Record.

6 Resultados

A implementação realizada foi sujeita inicialmente a um conjunto de testes base funcionais para aferir o seu funcionamento e verificada a inexistência de erros de implementação foram efetuados alguns testes para medição do desempenho.

Ao longo deste capítulo apresentam-se os testes realizados: os funcionais descrevem os pedidos realizados ao sistema para aferir do bom funcionamento da biblioteca e os testes de desempenho mostram a degradação que ocorre no sistema, em virtude da necessidade de executar mais operações (executar mais código Ruby e/ou efetuar mais *queries* à base de dados).

6.1 Ambiente de testes

Os testes foram realizados no mesmo computador em que foi implementada toda a solução, sendo as suas características:

- Processador: *Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz*
- Memória RAM: *4Gb*
- Sistema Operativo: *(64 bits) Windows 7 Enterprise*

O SGBD utilizado foi o mesmo utilizado durante o processo de implementação, o PostgreSQL, versão 9.1. A versão utilizada do Ruby foi a 1.9.3, tendo sido utilizada a versão do Rails 3.2.13. Para auxiliar o uso do PostgreSQL, utilizou-se o PGAdmin[20], versão 1.16.1.

Como ambiente de teste foi criada uma aplicação com dois modelos iguais:

1. `Unpartit_User`
2. `User`

[20] http://www.pgadmin.org/

Destes dois modelos, o primeiro não é particionado, enquanto o segundo é, utilizando a biblioteca desenvolvida neste trabalho.

Para o modelo particionado foi definido o critério de particionamento implementando a função `calc_part_name`, conforme se apresenta na Figura 5.2, que define um critério de particionamento por mês por ano. Desta forma obtém-se partições do tipo:

- `users_042012`;
- `users_062012`;
- `users_082013`;
- `users_102013`;
- Etc.

Cada modelo tem exatamente os mesmos atributos (ver Tabela 6.1).

Campo	Tipo
id	Integer
name	Varchar(255)
regdate	Date
age	Integer
created_at	Timestamp
updated_at	Timestamp

Tabela 6.1 - Modelo de dados das entidades de teste

Os testes foram efetuados em três bases de dados PostgreSQL, tendo cada uma quantidades diferentes de dados:

- 10 milhões de registos;
- 1 milhão de registos;
- 100 mil registos.

Em cada base de dados os dois modelos supracitados tinham a mesma dimensão.

6.2 Testes funcionais

Os métodos que o Active Record disponibiliza de suporte aos métodos CRUD aceitam muitos parâmetros/campos. Alguns dos campos são opcionais, e noutros casos nos parâmetros pode ser passada mais do que uma opção.

Foram efetuados testes funcionais aos métodos reescritos para testar todo o processamento implementado durante a sua reescrita, de forma a testar o máximo de vertentes que cada método do Active Record permite. Estes testes tinham como objetivo verificar que os resultados obtidos num modelo que utilize a biblioteca desenvolvida para particionar os seus dados são os mesmos resultados que se obtém quando se usa um modelo não particionado (implementação original do Active Record).

Adicionalmente, os testes funcionais foram efetuados com o mesmo modelo de dados implementado no SGBD MySQL.

Teste 01: Create&Save

Foram realizados 3 testes para verificar a criação de um novo registo.

`u = User.new`
`u.name = "André"`
`u.save`
	`u = User.new(:name=>"André")`
	`u.save`
		`User.create(:name=>"André")`

Teste 01-a) *new*	Teste 01-b) *new* com *hash* de atributos	Teste 01-c) *create*

Teste 02: All

O método `all` foi testado, obtendo todos os registos existentes na base de dados que satisfaçam os critérios definidos, com diferentes tipos de argumentos, como `order` e `limit`.

```
User.all
User.all (:order=>"name ASC")
User.all (:limit=>10)
User.all (:order=>"id desc", :limit=>5)
User.all (:conditions=>"id > 5")
User.all (:conditions=>"id > 5", :order=>"id desc", :limit=>5)
User.all (:select=>"id")
User.all (:conditions=>"data = '2013-07-01'")
User.all (:conditions=>"data = '2013-07-01'", :order=>"id ASC", :limit=>2)
```

Teste 02 - *all*

Teste 03: First

O método `first` foi testado sem nenhum argumento e com alguns argumentos opcionais.

```
User.first
User.first (:conditions => "name='Filipa'")
User.first (:select=>"id")
User.first (:conditions=>"data = '2013-07-01'")
```

Teste 03 - *first*

Teste 04: Last

Da mesma forma, o método `last` foi testado sem e com argumentos.

```
User.last
User.last (:conditions => "id<100")
User.last (:select=>"id")
User.last (:conditions=>"data = '2013-07-01'")
```

Teste 04 - *last*

Teste 05: Exists

A existência de um registo foi testada recorrendo a duas hipóteses.

`User.exists? (2)`
`User.exists? ('2')`
`User.exists? ("2")`
	`User.exists? (:name=>"André")`

Teste 05-a) *exists* por ID	Teste 05-b) *exists* por parâmetro passado em *hash*

Teste 06: Find

O método `find` foi testado assumindo os vários tipos de argumentos que permite como parâmetro, argumentos obrigatórios e opcionais.

`User.find (1)`
`User.find ("1")`
`User.find ([1])`
`User.find (["1"])`
	`User.find (2,4,5)`
	`User.find ("2","4","5")`
	`User.find ([2,4,5])`
	`User.find (["2","4","5"])`
		`User.find (:first)`
		`User.find (:last)`
		`User.find (:all)`

Teste 06-a) *find* de um ID	Teste 06-b) *find* de vários IDs	Teste 06-c) *find* com primitivas

```
User.find (:all, :order => "id ASC")
User.find (1,2,3, :order => "name desc")
User.find (:all, :order => "name desc", :limit=>5)
User.find ([4,5,8], :order => "name desc")
```

Teste 06-d) *find* com parâmetros opcionais

```
User.find (:all, :conditions => "id<010'", :order => "name asc", :limit=>5)
User.find (1,35, :conditions => "data='2013-07-01'", :limit=>5)
User.find (:all, :conditions => "data='2013-07-01'", :order => "name desc")
```

Teste 06-e) *find* com *:conditions*

Teste 07: Where

Foi testado o método `where`, passando argumentos do tipo *string* e do tipo *hash*.

```
User.where ("mensalidade > 50")
User.where ("nome='Adriana' AND mensalidade=55") [permite AND ou and]
User.where ("nome = ?", var1)
User.where ("name = :n", :n=>"Juliana")
User.where ("nome = ? AND mensalidade = ?", var1, var2)
User.where ("name like ?", "%Apelido")
User.where ("name like ?", "Primeiro%")
User.where ("name like ?", "%Meio%")
```

Teste 07-a) *where* com parâmetros do tipo *string*

```
User.where (:nome => "Juliana Costa")
User.where ('nome' => "Juliana Costa")
User.where (:nome =>"Adriana", :mensalidade=>55)
User.where (:mensalidade=>(50)..(55))
```

Teste 07-b) *where* com parâmetros do tipo *hash*

Teste 08: Dynamic Finders

Dentro dos *dynamic finders*, foram testados os métodos do tipo `find_by` e `find_all_by`.

```
User.find_by_name("Tiago")
User.find_by_name_and_age("Tiago", 25)
```

Teste 08-b) *find_by_X; find_by_X_and_Y*

```
User.find_all_by_name ("Tiago", :order=>"id DESC", :conditions=>"age>20")
User.find_all_by_name_and_age( "Tiago", 25)
```

Teste 08-b) *find_all_by_X; find_all_by_X_and_Y*

Teste 09: Update

Foi testada a atualização de registos realizando testes com os métodos `update_attributes`, `update` e `update_all`.

`@user = User.find(3)` `@user.update_attributes(:name="João")`	`@user = User.find(3)` `@user.update(:name="João")`
Teste 09-a) *update_attributes* com função *find*	Teste 09-b) *update* com função *find*

`@user = User.find(3)`
`@user. name="João"`
`@user.save`
	`User.update(3, :name="João")`

Teste 09-c) *update* recorrendo ao *save*	Teste 09-d) *update* passando id do registo

`User.update_all(:mensalidade=>30)` `User.update_all({:mensalidade=>25}, {:mensalidade=>0})` **[o segundo campo é uma condição]**
Teste 09-e) *update_all*

Teste 10: Delete

A eliminação de registos foi testada recorrendo aos métodos `destroy` e `destroy_all`.

`@user = User.find(3)` `@user.destroy`	`User.destroy(3)` `User.destroy(3,6,10)`
Teste 10-a) *destroy* usando a função *find*	Teste 10-b) *destroy* passando id(s) do(s) registo(s)

`User.where("id>100").destroy_all`
`User.destroy_all(:mensalidade=>0)`
`User.destroy_all("mensalidade=0")`
`User.destroy_all(["mensalidade=0", "data='02-05-2013'"])`
`User.destroy_all(:mensalidade=>0, :data=>'02-05-2013')`

Teste 10-d) *destroy_all*

Concluídos os testes funcionais, comparando os resultados que se obtém num modelo normal, que não utiliza particionamento, com os resultados obtidos de um modelo que utiliza a biblioteca desenvolvida para usar particionamento, observa-se que o sistema devolve sempre os mesmos registos.

6.3 Testes de desempenho

Após os testes funcionais, foram escritos e efetuados testes de desempenho de modo a aferir se o tempo necessário para executar o código de processamento acrescentado em alguns métodos introduz uma maior carga, que se reflete na diminuição do desempenho de uma aplicação que utilize a biblioteca desenvolvida.

Os testes foram realizados invocando os mesmos métodos para cada um dos modelos, de forma a medir o tempo de execução em cada um e poder comparar o desempenho verificada em cada modelo.

Para cada método foram realizados 20 testes idênticos. Cada teste consiste em invocar consecutivamente o método a testar 500 vezes, sendo que em cada invocação são usados dados gerados de forma aleatória. A utilização de dados aleatórios resulta da necessidade de executar testes que se aproximem da realidade e não sejam executados com um valor constante (que pode favorecer ou prejudicar o desempenho de forma contínua). Para cada teste é calculado o tempo médio de execução, dividindo por 500 o tempo total que demora a executar as referidas 500 invocações.

O racional que levou à escolha destes números prendeu-se com a capacidade de processamento da máquina utilizada em confrontação com o tempo útil disponível para a execução dos testes. Optou-se também pelos 20 testes (repetições) para que a fidedignidade dos resultados fosse maior, considerando que com um menor número de repetições existia o risco de afetar os resultados finais, dado que não é possível impedir que outros processos corram em simultâneo na máquina e tenham influência no seu desempenho, nomeadamente processos do sistema operativo.

As funções testadas foram:

1. `create(:name=>"UserTest#{i}", :regdate=>date, :age=>i)`
2. `all(:conditions=>"name='#{NAMES[rand(91)]}'")`
3. `all(:conditions=>"regdate='#{DATES[rand(91)]}'")`
4. `first`
5. `first(:conditions=>"regdate='#{DATES[rand(91)]}'")`
6. `last`
7. `last(:conditions=>"regdate='#{DATES[rand(91)]}'")`
8. `where("name='#{NAMES[rand(91)]}'")`

9. `where(:regdate=>"#{DATES[rand(91)]}")`
10. `find(rand(100000)+1)`
11. `find((rand(100000)+1),(rand(100000)+1),(rand(100000)+1), :conditions=>"age>0")`
12. `find(10000005, :conditions=>"regdate='01-06-2013'")`
13. `find_by_name(NAMES[rand(91)])`
14. `find_by_regdate(DATES[rand(91)])`
15. `find_all_by_name(NAMES[rand(91)])`
16. `find_all_by_regdate(DATES[rand(91)])`
17. `update(rand(100000)+1, :age=>rand(100)+1)`
18. `update_all({:age=>rand(100)+1}, {:name=>NAMES[rand(91)]})`
19. `update_all({:age=>rand(100)+1}, {:regdate=>DATES[rand(91)]})`
20. `exists?(rand(100000)+1)`
21. `exists?(:regdate=>DATES[rand(91)])`
22. `count(:all,:conditions=>"age=#{rand(100)+1}")`
23. `count(:all,:conditions=>"regdate='#{DATES[rand(91)]}'")`
24. `destroy_all("id>10009500 and id<=10010000")`
25. `destroy_all(:regdate=>"02-06-2013")`

Destes métodos existem alguns aos quais não é possível passar dados aleatórios. É o caso dos métodos dos pontos 12 (`find`), 24 (`destroy_all`) e 25 (`destroy_all`).

No caso do `find` do ponto 12, é passado um id e uma condição. Foi necessário garantir que os testes deste método retornem com sucesso, para não ser lançada uma exceção, de forma a se conseguir realizar as 500 invocações para cada execução.

Nos métodos apresentados nos pontos 24 e 25, `destroy_all`, não é possível passar dados aleatórios, pois pode acontecer que o método seja chamado com uma data (campo `regdate`) que já não exista, originando um erro.

Para testar o `create` e os métodos de `destroy` foi decidido usar um conjunto de 10000 registos. Este valor foi calculado considerando que 20 testes a executarem 500 pedidos realizam um total de 10000 invocações ao sistema.

O *create* foi usado para gerar a criação dos 10000 registos adicionais e os métodos de *destroy* foram usados para apagar esses mesmos registos.

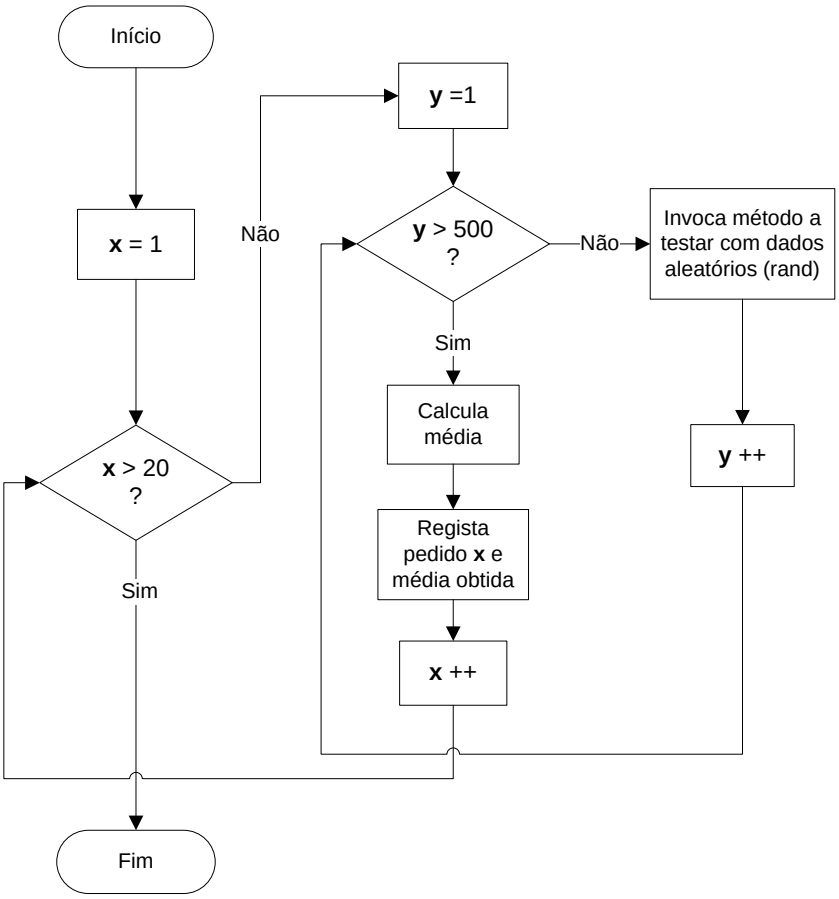

Figura 6.1 - Tipologia de um teste

A Figura 6.1 mostra o fluxograma relativo à execução de um conjunto de 20 testes. Após executar cada teste obtém-se o tempo médio que demorou a executar cada um dos pedidos. No final de um conjunto de testes obtém-se 20 valores, onde cada um representa o tempo médio de um pedido de cada série de 500 pedidos.

Esta média é calculada recorrendo ao registo do tempo antes e depois da invocação do método. Para tal utilizou-se a classe `Time` da seguinte forma:

```
start = Time.now
…
end = Time.now
```

No final a média é calculada fazendo a diferença dos dois tempos, em que o resultado é sempre retornado em segundos. Dado que algumas operações demoram menos de um segundo foi decidido converter e apresentar esta diferença em milissegundos.

Depois de calculada a média, esta é registada, juntamente com o número do pedido atual num ficheiro de texto (`.txt`).

Na Figura 6.2 pode ver-se um exemplo de testes a um método (neste caso o método `create` na base de dados de dez milhões de registos).

```
R1 - Action: CREATE       Time(ms): 1.95       Partitioned? NO
R2 - Action: CREATE       Time(ms): 1.77       Partitioned? NO
R3 - Action: CREATE       Time(ms): 1.57       Partitioned? NO
R4 - Action: CREATE       Time(ms): 1.37       Partitioned? NO
R5 - Action: CREATE       Time(ms): 1.34       Partitioned? NO
R6 - Action: CREATE       Time(ms): 1.33       Partitioned? NO
R7 - Action: CREATE       Time(ms): 1.35       Partitioned? NO
R8 - Action: CREATE       Time(ms): 1.40       Partitioned? NO
R9 - Action: CREATE       Time(ms): 1.35       Partitioned? NO
R10 - Action: CREATE      Time(ms): 1.30       Partitioned? NO
R11 - Action: CREATE      Time(ms): 1.37       Partitioned? NO
R12 - Action: CREATE      Time(ms): 1.43       Partitioned? NO
R13 - Action: CREATE      Time(ms): 1.29       Partitioned? NO
R14 - Action: CREATE      Time(ms): 1.43       Partitioned? NO
R15 - Action: CREATE      Time(ms): 1.31       Partitioned? NO
R16 - Action: CREATE      Time(ms): 1.35       Partitioned? NO
R17 - Action: CREATE      Time(ms): 1.31       Partitioned? NO
R18 - Action: CREATE      Time(ms): 1.40       Partitioned? NO
R19 - Action: CREATE      Time(ms): 1.36       Partitioned? NO
R20 - Action: CREATE      Time(ms): 1.40       Partitioned? NO

R1 - Action: CREATE       Time(ms): 2.59       Partitioned? YES
R2 - Action: CREATE       Time(ms): 2.47       Partitioned? YES
R3 - Action: CREATE       Time(ms): 2.52       Partitioned? YES
R4 - Action: CREATE       Time(ms): 2.46       Partitioned? YES
R5 - Action: CREATE       Time(ms): 2.60       Partitioned? YES
R6 - Action: CREATE       Time(ms): 2.64       Partitioned? YES
R7 - Action: CREATE       Time(ms): 2.86       Partitioned? YES
R8 - Action: CREATE       Time(ms): 2.51       Partitioned? YES
R9 - Action: CREATE       Time(ms): 2.55       Partitioned? YES
R10 - Action: CREATE      Time(ms): 2.48       Partitioned? YES
R11 - Action: CREATE      Time(ms): 2.69       Partitioned? YES
R12 - Action: CREATE      Time(ms): 2.56       Partitioned? YES
R13 - Action: CREATE      Time(ms): 2.62       Partitioned? YES
R14 - Action: CREATE      Time(ms): 2.61       Partitioned? YES
R15 - Action: CREATE      Time(ms): 2.50       Partitioned? YES
R16 - Action: CREATE      Time(ms): 2.62       Partitioned? YES
R17 - Action: CREATE      Time(ms): 2.62       Partitioned? YES
R18 - Action: CREATE      Time(ms): 2.47       Partitioned? YES
R19 - Action: CREATE      Time(ms): 2.57       Partitioned? YES
R20 - Action: CREATE      Time(ms): 2.56       Partitioned? YES
```

Figura 6.2 - Exemplo de registo dos testes de um método

Como já foi referido anteriormente nesta secção, os mesmos testes são realizados para os dois modelos criados, o particionado (*User*) e o não particionado (*Unpartit_User*), de forma a se poder comparar resultados obtidos. Os argumentos passados aos métodos contêm os mesmos atributos, embora com valores diferentes, pois são gerados aleatoriamente. Os critérios de cálculo da média e de escrita no ficheiro de texto também

são os mesmos apenas diferenciando no ficheiro o campo **Partitioned** que indica a que modelo pertencem os testes.

6.3.1 Resultados obtidos

Foram testadas as 25 funções apresentadas na secção anterior executando-as sobre três bases de dados com 10 milhões, 1 milhão e 100 mil registos respetivamente. No modelo particionado, os registos contidos nas três bases de dados foram dispersos por um total de 12 partições. A função de particionamento usava uma data para determinar a partição em função do mês e do ano. Nos dados de teste todos os registos eram do mesmo ano pelo que as partições criadas eram as correspondentes aos 12 meses do ano. As partições não tinham necessariamente o mesmo número de registos dado que o campo de particionamento (data) é gerado de forma aleatória.

Nesta secção apresenta-se os resultados obtidos nos testes efetuados, usando a biblioteca desenvolvida (modelo particionado) e usando um modelo não particionado. Os resultados apresentados mostram, em milissegundos, o tempo médio que demorou a executar uma operação. Foram realizados 20 testes e em cada teste foram executadas 500 operações.

Para uma melhor depuração dos resultados, a média calculada é uma média ponderada, tendo sido para cada um dos casos retirados do cálculo os 5 resultados melhores e os 5 piores.

Na Tabela 6.2 - Resultados dos testes ao método *Create*Tabela 6.2 são apresentados os resultados dos testes ao método `create`.

	10milhões		1milhão		100mil	
CREATE	**NP[21]**	**P [22]**	**NP**	**P**	**NP**	**P**
	1,95	2,59	1,35	2,63	1,66	2,77
	1,77	2,47	1,37	2,54	1,32	2,76
	1,57	2,52	1,48	2,80	1,56	2,68
	1,37	2,46	1,53	2,78	1,43	2,83
	1,34	2,60	1,33	2,65	1,41	2,59
	1,33	2,64	1,41	2,76	1,52	2,76
	1,35	2,86	1,48	2,61	1,45	2,91
	1,40	2,51	1,45	2,74	1,45	2,51
	1,35	2,55	1,36	2,72	1,52	2,73
	1,30	2,48	1,47	2,75	1,48	2,72
	1,37	2,69	1,41	2,80	1,40	2,72
	1,43	2,56	1,53	2,49	1,54	2,84
	1,29	2,62	1,47	2,64	1,50	2,85
	1,43	2,61	1,37	2,61	1,38	2,67
	1,31	2,50	1,40	2,72	1,45	2,78
	1,35	2,62	1,49	2,76	1,59	2,73
	1,31	2,62	1,44	2,75	1,34	2,76
	1,40	2,47	1,45	2,88	1,43	2,75
	1,36	2,57	1,38	2,81	1,41	2,62
	1,40	2,56	1,52	2,61	1,43	2,76
Média	**1,37**	**2,57**	**1,44**	**2,71**	**1,46**	**2,75**
Aumento (%)	**87,66%**		**88,86%**		**88,73%**	

Tabela 6.2 - Resultados dos testes ao método *Create*

O tempo que demora, em média, um *create* não varia de acordo com a quantidade de dados presentes na base de dados, pelo que se pode verificar que os valores das médias estão na ordem de 1 milissegundo para o modelo não particionado e na ordem dos 2 milissegundos para o modelo particionado.

Na Tabela 6.3 - Resultados dos testes para os restantes métodos é exposto um resumo dos restantes testes apresentando, para cada método, apenas as médias dos 20 testes.

[21] Modelo não Particionado.
[22] Modelo Particionado.

	10milhões		1milhão		100mil	
	NP	**P**	**NP**	**P**	**NP**	**P**
All	749,74	970,41	30,28	37,52	2,97	6,34
All Part	1636,38	1676,76	54,27	66,06	6,07	6,57
First	0,17	11,04	0,17	3,37	0,14	3,05
First Part	0,19	0,69	0,20	0,71	0,18	0,67
Last	0,19	13,02	0,19	3,58	0,17	3,97
Last Part	0,21	0,72	0,22	0,76	0,21	0,69
Where	0,03	963,81	0,04	53,42	0,03	8,88
Where Part	0,03	0,46	0,03	0,48	0,03	0,47
Find Id	0,39	19,10	0,36	2,83	0,36	2,83
Find Cond	0,52	4,29	0,58	4,55	0,55	4,45
Find Part	0,21	0,75	0,22	0,75	0,22	0,71
Find_by	0,21	4,50	0,23	4,31	0,21	4,22
Find_by Part	0,22	4,64	0,21	4,25	0,21	4,21
Find_all_by	16,11	29,80	1,59	6,83	0,35	3,96
Find_all_by Part	24,13	27,71	2,50	3,13	0,43	0,89
Update	2,33	5,69	2,17	5,44	2,23	5,56
Update_all	1057,16	484,29	412,82	13,25	35,38	4,25
Update_all Part	791,44	217,40	326,46	77,84	45,21	17,00
Exists	0,63	4,51	0,56	3,65	0,59	4,45
Exists Part	0,34	0,82	0,31	0,84	0,32	0,83
Count	0,20	2,85	0,19	2,86	0,19	2,86
Count Part	0,19	1,18	0,19	1,15	0,19	1,16
Destroy_all	0,93	1,04	0,90	0,97	0,83	0,93
Destroy_all Part	4,12	2,14	1,13	0,99	0,85	0,92

Tabela 6.3 - Resultados dos testes para os restantes métodos

Analisando a Tabela 6.3, constata-se que em determinados métodos o processamento extra, acrescentado ao código da biblioteca de modo a permitir particionamento, leva a um acréscimo do tempo de execução do modelo não particionado para o modelo particionado. O maior acréscimo do tempo de execução verifica-se no método `where`, no caso em que os argumentos não incluem o campo de particionamento. Note-se que nas tabelas 6.1, 6.2 e seguintes os resultados são apresentados usando como unidade de medida o milissegundo pelo que em todos os casos o tempo de execução médio de um pedido `where`, tanto no modelo não particionado como no modelo particionado, encontra-se abaixo de 1 segundo.

No método `all`, tanto sem o campo de particionamento, como com o campo de particionamento, o acréscimo também é evidente, mas não tão significativo como no método `where`.

Os métodos `first` e `last` consomem mais algum tempo no modelo particionado. Uma observação interessante nos resultados dos testes destes dois métodos baseia-se no facto de quando o método é chamado sem o campo de particionamento a diferença do tempo de execução no modelo não particionado para o particionado aumenta mais do que para o caso em que o método recebe como argumento a chave de particionamento.

	10milhões		**1milhão**		**100mil**	
	NP	**P**	**NP**	**P**	**NP**	**P**
First	0,17	11,04	0,17	3,37	0,14	3,05
First Part	0,19	0,69	0,20	0,71	0,18	0,67
Last	0,19	13,02	0,19	3,58	0,17	3,97
Last Part	0,21	0,72	0,22	0,76	0,21	0,69

Tabela 6.4 - Resultados dos testes dos métodos *first* e *last*

Por exemplo, observando os resultados dos métodos `first` e `last`, novamente referidos na Tabela 6.4, na base de dados de 10 milhões de registos, verifica-se que para estes métodos sem o campo de particionamento presente nos argumentos o aumento do tempo de execução é de 10.87ms e 12.83ms. Para o caso em que é passado como argumento o campo de particionamento a diferença não chega a 1ms. O mesmo acontece para as bases de dados de 1 milhão e de 100 mil registos.

No geral, para um modelo particionado, em qualquer função em que não seja passado como argumento o campo de particionamento, o seu tempo de execução é maior devido ao processamento extra que tem que efetuar. A ausência do campo de particionamento implica em alguns casos percorrer todas as partições existentes, resultando na necessidade de o SGBD executar tantas *queries* quantas as partições existentes. Quando o campo de particionamento é fornecido como um argumento de um método, o processamento extra já não é tão elevado, pelo que a diferença dos tempos de execução do modelo não particionado para o modelo particionado já não é tão significativa.

Nos métodos `find`, o tempo de execução de um pedido aumenta sempre do modelo não particionado para o modelo particionado. Para os métodos `find` dinâmicos, no caso de um `find_by`, o aumento do tempo de execução do modelo não particionado para o modelo particionado é sempre na ordem dos 4 milissegundos. Já no caso de um `find_all_by` verifica-se igualmente um aumento, embora sempre menor quando se recebe como parâmetro o campo de particionamento.

Nos métodos `update_all` e `destroy_all` os resultados mostram um comportamento contrário ao comportamento dos métodos anteriores quando recebem como argumento o campo de particionamento. No método `update_all` verifica-se uma diminuição do tempo de execução do modelo não particionado para o modelo particionado. No método `destroy_all` o tempo de execução diminui com exceção na base de dados de 100 mil registos.

Quando o campo de particionamento é passado como argumento nos métodos testados os resultados são praticamente idênticos pois são muito próximos. Algumas variações não são expectáveis (como no caso do método `destroy_all` que aumenta na base de dados de 100 mil registos) mas admite-se que alguns destes resultados possam ser justificados pela execução de outras aplicações em simultâneo em processos paralelos que possam ter ocorrido no mesmo CPU e pela execução do próprio sistema operativo.

Observando a variação do tempo médio de execução dos pedidos entre as várias bases de dados, verifica-se que, da base de dados maior (10 milhões) para a base de dados menor, existem dois tipos de comportamentos relacionados com o tipo de função, já abordado na secção 5.2.3.2. Algumas funções retornam (terminam a execução com sucesso) quando encontram um registo que verifique as condições passadas, pelo que quando o encontram não pesquisam nos restantes registos. Outras funções pesquisam todos os registos correspondentes às condições passadas como argumento, tendo obrigatoriamente que percorrer toda a tabela onde os mesmos se encontram.

Este facto traduz-se nos resultados dos testes da seguinte forma: nas funções que retornam quando encontram um registo o tempo de execução é idêntico entre as diferentes bases de dados, tal como se pode verificar na Tabela 6.5. Nas funções que percorrem toda a tabela do modelo em questão, o tamanho da base de dados já influencia

no tempo de execução, pois tem que percorrer todos os registos existentes, como mostra a Tabela 6.6.

	10milhões		1milhão		100mil	
	NP	P	NP	P	NP	P
Update	2,33	5,69	2,17	5,44	2,23	5,56
Find_by	0,21	4,50	0,23	4,31	0,21	4,22
Find_by Part	0,22	4,64	0,21	4,25	0,21	4,21

Tabela 6.5 - Métodos que retornam após encontrar um registo

	10milhões		1milhão		100mil	
	NP	P	NP	P	NP	P
All	749,74	970,41	30,28	37,52	2,97	6,34
All Part	1636,38	1676,76	54,27	66,06	6,07	6,57
Find_all_by	16,11	29,80	1,59	6,83	0,35	3,96
Find_all_by Part	24,13	27,71	2,50	3,13	0,43	0,89

Tabela 6.6 - Métodos que retornam após percorrer todos os registos

6.3.2 Conclusões

Finalizados os testes à biblioteca de particionamento desenvolvida neste trabalho, verifica-se que em determinados métodos existe um aumento do tempo de execução. Este aumento era previsível dado que:

- Em determinadas funções é necessário processar mais código, i.e., é necessário executar o código da biblioteca que foi desenvolvida, aumentando o número total de instruções a serem executadas;
- Em determinadas operações é necessário efetuar *queries* SQL sobre múltiplas tabelas. Por exemplo, no caso da inserção é necessário obter o `id` do novo registo, inserir o registo na partição e atualizar a tabela auxiliar que guarda os *Ids* de cada modelo.

No caso concreto dos métodos `update_all` e `destroy_all`, quando é incluído o campo de particionamento nos seus argumentos, verifica-se uma melhoria do tempo total de execução, o que mais uma vez está de acordo com as expectativas iniciais, dado que no modelo particionado o desenvolvimento do método é diferente do método original do Active Record, sendo utilizadas no Active Record estruturas de dados diferentes.

O objetivo do desenvolvimento desta biblioteca não era conseguir um sistema que melhorasse o desempenho face ao que já existia. De facto, ainda que em inúmeras situações se verifique uma diminuição do desempenho, o que se conclui é que o sistema em termos absolutos apresenta ainda valores totais de execução reduzidos.

A utilização da biblioteca não provoca qualquer degradação significativa do desempenho global, permitindo ter um sistema perfeitamente usável na interação com o utilizador. O tempo de resposta continua a ser reduzido ao ponto de não ser percetível quando uma operação é executada de forma interativa. Note-se que em termos absolutos o sistema apresenta tempos de resposta sempre inferiores a 1 segundo.

7 Conclusões e trabalho futuro

Este capítulo conclui a presente dissertação. Ao longo deste capítulo é apresentada uma reflexão sobre os objetivos e implicações dos resultados, apresentando as limitações da solução implementada e algumas tarefas a realizar num trabalho futuro.

7.1 Conclusão do trabalho realizado

Neste documento foi apresentada uma ferramenta de gestão de dados históricos. Em relação aos objetivos delineados inicialmente:

- Foi desenvolvido um sistema que permite gerir dados históricos recorrendo a técnicas de particionamento;
- Como solução foi implementada uma biblioteca que pode ser chamada em qualquer aplicação, sendo independente do SGBD utilizado, assim como do modelo de dados adotado em diferentes aplicações;
- A biblioteca desenvolvida reescreve os métodos nativos do ORM utilizado pelo Ruby on Rails, o Active Record.

Apesar de o objetivo deste trabalho não ser desenvolver uma biblioteca que permita particionamento com desempenho, foram realizados testes funcionais e testes de desempenho, de forma a apurar a carga introduzida pelo processamento extra introduzido pela biblioteca.

Para os testes de desempenho foram executados 20 pedidos de 500 execuções cada um, calculando a média do tempo de execução ao fim de cada 500 execuções. Assim, no final obtém-se 20 amostras, das quais são excluídas as 5 melhores e as 5 piores.

Destes resultados conclui-se que em alguns métodos o tempo de execução para um modelo particionado utilizando a biblioteca é maior do que para um modelo não particionado. Destes aumentos não existe nenhum significativo que seja percetível ao utilizador, considerando-se perfeitamente aceitáveis os resultados obtidos.

Durante a realização deste trabalho foram ultrapassados problemas e desafios, abrindo-se uma nova possibilidade de particionar modelos de dados numa aplicação Ruby, em Ruby on Rails, que utiliza o ORM Active Record para aceder a bases de dados, mapeando os dados de uma tabela em objetos. Mais do que isso, foi possível obter experiências e conhecimentos de um vasto conjunto de tecnologias e arquiteturas utilizadas, como por exemplo a linguagem Ruby, na plataforma Ruby on Rails, que implementa a arquitetura MVC.

7.2 Limitações e trabalho futuro

Apesar de todo o tempo e esforço aplicado neste trabalho, existem algumas limitações que podem ser analisadas e possivelmente eliminadas num futuro trabalho.

Relativamente à construção do nome das partições, depois de encerrados os desenvolvimentos, foi detetada a seguinte situação que pode ser problemática no futuro: na construção dos nomes das partições existe alguma convenção tomada na biblioteca, implicando que o nome de uma partição seja sempre começado por:

`nomeTabelaOriginal_`

Quando um determinado método não recebe como parâmetro a chave de particionamento e tem que percorrer todas as partições, obtém todas as partições da conexão. Para determinar quais as partições do modelo em questão, são verificados os nomes de todas as tabelas, testando se os mesmos começam pelo padrão em cima referido.

É possível existirem na base de dados tabelas normais cujo nome comecem pelo mesmo padrão. Por exemplo, é possível ter uma tabela alunos que é particionada pelo ano de nascimento, existindo partições do género:

- alunos_1988;
- alunos_1989;
- alunos_2000;
- Etc.

Contudo, pode existir uma tabela na base de dados que relacione os alunos por turma, sendo possivelmente o seu nome:

alunos_por_turma

Neste caso, quando verificadas todas as partições do modelo Aluno, a tabela alunos_por_turma será também analisada.

Este problema poderá ser facilmente contornado, criando uma tabela auxiliar que regista os nomes das partições de cada modelo particionado.

Devido ao facto de se convencionar o padrão começado nomeTabelaOriginal_ para os nomes das partições, o programador, ao escrever a sua própria implementação da função **calc_part_name** para definir o critério de particionamento, é obrigado a definir desta forma o nome da partição seguindo este início, tendo apenas liberdade no critério de particionamento. Contudo, algum tipo de convenção tem sempre que existir e o padrão acima referido é o que faz mais sentido, e é utilizado por exemplo pelo Oracle.

No entanto, para dar mais flexibilidade ao programador na definição do seu modelo de dados, poderá ser adicionada a capacidade de configurar, para cada modelo, como devem ser construídos os nomes das partições, sendo que esses nomes terão que obter obrigatoriamente o nome do modelo e o nome da partição calculado através da função **calc_part_name**.

Num trabalho futuro será interessante comparar o desempenho deste sistema com o desempenho de um sistema particionado diretamente pelo SGBD. Por exemplo, o MySQL e o Oracle possuem mecanismos de particionamento. Será interessante efetuar testes de desempenho com, por exemplo, o MySQL a fazer particionamento e comparar com os resultados da biblioteca desenvolvida neste trabalho.

Outro aspeto interessante será medir o desempenho desta biblioteca (e do modelo não particionado) noutra máquina. Por exemplo, seria interessante fazer os mesmos testes de desempenho numa máquina do tipo servidor, que certamente tem outro sistema operativo (por exemplo: Windows Server) e que tem outras capacidades de processamento.

Bibliografia

Baer, H. (2007). Particionamento no banco de dados Oracle 11g. Obtido em 02 de Junho de 2013, de Oracle: http://www.oracle.com/technetwork/pt/database/enterprise-edition/documentation/particionamento-banco-de-dados-11g-432098-ptb.pdf

Carr, B., Garmany, J., Hartmann, L., Jain, V., & Karam, S. (2008). *Oracle 11g new features - Get Started Fast with Oracle 11g Enhancements.* USA: Rampant Techpress.

Damas, L. (2005). *SQL - Structured Query Language* (Décima segunda ed.). FCA.

DB-Engines Ranking. (2013). Obtido em 18 de Outubro de 2013, de DB-Engines: http://db-engines.com/en/ranking

GitHub. (2013). Obtido em 14 de Junho de 2013, de https://github.com/jruby/activerecord-jdbc-adapter

Greenwald, R., Stackowiak, R., & Stern, J. (2007). *Oracle Essentials - Oracle Databases 11g* (Quarta ed.). O'Reilly Media.

Hobbs, L. (2007). *Information Lifecycle Management for Business Data* . Obtido em 23 de Maio de 2013, de Oracle: http://www.oracle.com/us/026964.pdf

Horstmann, C. (2010). *Big Java* (Quarta ed.). USA: John Wiley & Sons, Inc.

ITIC. (2011). *SQL Server 2008 R2 and Windows Server 2008 R2 Deliver Industry-Leading Security* .

JBoss.org . (2013). Obtido em 11 de Julho de 2013, de Torquebox: http://torquebox.org/builds/html-docs/database.html

Kavanagh, P. (2004). *Open Source Software, Implementation and Management.* USA: Elsevier Digital Press.

Loshin, D. (2003). *Business Intelligence - The Savvy manager's guide.* USA: Elsevier Science.

Marshal, K., Pytel, C., & Yurek, J. (2007). *Pro Active Record - Databases with Ruby and Rails.* USA: Apress.

Microsoft SQL Server. (2013). Obtido em 14 de Janeiro de 2013, de Microsoft: http://www.microsoft.com/sqlserver/en/us/default.aspx

Mistry, R., & Misner, S. (2012). *Introducing Microsoft SQL Server 2012.* Washington: Microsoft Press.

MySQL. (2013). Obtido em 14 de Outubro de 2013, de MySQL: http://www.mysql.com/

MySQL Documentation. (2013). Obtido em 14 de Outubro de 2013, de MySQL: http://dev.mysql.com/doc/refman/5.6/en/mysql-nutshell.html

Olofson, C. W. (2008). World wide Relational Database Management Systems. *Competitive Analysis* .

Oracle. (2013). Obtido em 26 de Setembro de 2013, de Oracle: www.oracle.com

Oracle Database New Features Guide, 12c Release 1. (Junho de 2013). Obtido em 26 de Setembro de 2013, de Oracle Database 12c Documentation: http://docs.oracle.com/cd/E16655_01/server.121/e17906.pdf

O'Reilly. (2013). Obtido em 12 de Março de 2013, de Onlamp: http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html?page=2

PostgreSQL. (2013). Obtido em 14 de Outubro de 2013, de PostgreSQL: http://www.postgresql.org/

PostgreSQL Documentation. (2013). Obtido em 14 de Outubro de 2013, de PostgreSQL: http://www.postgresql.org/docs/9.3/static/release-9-3.html

Ruby on Rails Architectural Design. (2013). Obtido em 05 de Junho de 2013, de Adrian Mejia's [code]Blog: http://adrianmejia.com/blog/2011/08/11/ruby-on-rails-architectural-design/

Ruby Tutorials. (2011). Obtido em 22 de Setembro de 2013, de Sitepoint: http://www.sitepoint.com/building-your-first-rails-application-views-and-controllers/

Ruby, S., Thomas, D., & Hansson, D. H. (2011). *Agile Web Development with Rails* (Quarta ed.). USA: Pragmatic Programmers, LLC.

Rubyonrails. (2013). Obtido em 11 de Julho de 2013, de Rails: http://rubyonrails.org/

Schwartz, B., Zaitsev, P., & Tkachenko, V. (2012). *High Performance MySQL* (Terceira ed.). USA: O'Reilly Media.

Stack Overflow Questions. (2013). Obtido em 22 de Agosto de 2013, de Stack Overflow: http://stackoverflow.com/questions/10426352/why-rails-activerecord-last-method-orders-by-id-whereas-first-does-not

Test stub. (07 de Outubro de 2012). Obtido em 07 de Fevereiro de 2013, de Wikipedia - The free encyclopedia: http://en.wikipedia.org/wiki/Test_stub

Thomas, D. (2005). *Programming Ruby - The Pragmatic Programmers' Guide* (Segunda ed.). Pragmatic Programmers, LLC.

Anexos

Anexo A: Resultados completos dos testes de desempenho

Create:

	10milhões		1milhão		100mil	
CREATE	**NP**	**P**	**NP**	**P**	**NP**	**P**
	1,95	2,59	1,35	2,63	1,66	2,77
	1,77	2,47	1,37	2,54	1,32	2,76
	1,57	2,52	1,48	2,80	1,56	2,68
	1,37	2,46	1,53	2,78	1,43	2,83
	1,34	2,60	1,33	2,65	1,41	2,59
	1,33	2,64	1,41	2,76	1,52	2,76
	1,35	2,86	1,48	2,61	1,45	2,91
	1,40	2,51	1,45	2,74	1,45	2,51
	1,35	2,55	1,36	2,72	1,52	2,73
	1,30	2,48	1,47	2,75	1,48	2,72
	1,37	2,69	1,41	2,80	1,40	2,72
	1,43	2,56	1,53	2,49	1,54	2,84
	1,29	2,62	1,47	2,64	1,50	2,85
	1,43	2,61	1,37	2,61	1,38	2,67
	1,31	2,50	1,40	2,72	1,45	2,78
	1,35	2,62	1,49	2,76	1,59	2,73
	1,31	2,62	1,44	2,75	1,34	2,76
	1,40	2,47	1,45	2,88	1,43	2,75
	1,36	2,57	1,38	2,81	1,41	2,62
	1,40	2,56	1,52	2,61	1,43	2,76
Média	**1,37**	**2,57**	**1,44**	**2,71**	**1,46**	**2,75**
Aumento (%)	**87,66%**		**88,86%**		**88,73%**	

All:

	10milhões		1milhão		100mil	
ALL	**Un**	**P**	**Un**	**P**	**Un**	**P**
	973,38	9575,07	62,45	74,57	6,61	9,88
	747,27	1166,67	30,16	37,98	2,96	6,34
	733,49	1060,62	31,48	37,46	2,91	6,41
	751,87	1050,05	30,82	37,02	2,94	6,31
	764,40	1046,44	30,35	38,18	2,89	6,32
	772,18	1024,20	30,87	37,67	2,97	6,36
	743,87	1015,68	30,16	38,07	2,95	6,32
	757,11	983,77	30,17	36,84	2,99	6,43
	748,25	956,27	30,17	37,36	2,99	6,33
	732,21	971,12	30,09	37,63	2,95	6,06
	743,19	981,54	30,44	37,10	3,04	6,37
	761,54	977,64	29,47	38,16	3,23	6,34
	740,46	931,85	30,54	37,86	3,00	6,41
	755,25	929,54	29,88	37,07	3,03	6,31
	746,38	921,23	29,22	37,46	2,93	6,39
	740,86	932,50	30,54	37,57	3,03	6,24
	748,32	903,85	30,32	36,87	2,97	6,24
	739,66	901,92	30,33	36,78	2,94	6,28
	755,87	898,26	30,16	37,88	2,95	6,20
	771,39	885,76	10,08	37,20	2,94	6,39
Média	**749,74**	**970,41**	**30,28**	**37,52**	**2,97**	**6,34**
Aumento (%)	**29,43%**		**23,91%**		**113,65%**	

All (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
ALL PART	**Un**	**P**	**Un**	**P**	**Un**	**P**
	1800,24	1773,23	83,85	68,13	8,56	7,37
	1647,32	1661,64	53,03	64,25	6,09	6,47
	1598,08	1701,22	54,16	66,43	6,24	6,43
	1733,98	1620,17	54,01	64,84	5,94	6,45
	1632,47	1705,14	56,59	64,87	6,07	6,51
	1666,00	1632,58	54,45	67,31	6,03	6,48
	1689,74	1647,30	55,49	63,57	6,03	6,55
	1600,40	1648,96	53,71	65,56	6,24	6,51
	1643,14	1647,47	56,14	66,45	6,06	6,52
	1634,36	1706,17	53,22	67,66	5,92	6,61
	1626,23	1741,58	56,27	67,66	6,14	6,78
	1616,63	1679,87	53,27	65,27	6,21	6,57
	1638,41	1670,05	54,08	67,39	6,05	6,62
	1661,78	1696,55	54,02	64,39	5,97	6,66
	1663,40	1663,35	53,61	66,72	6,13	6,62
	1617,00	1673,92	54,42	65,56	6,13	6,57
	1600,90	1683,22	54,95	66,09	6,00	6,76
	1565,86	1672,57	54,87	67,09	6,04	6,60
	1605,08	1665,17	52,99	64,77	6,00	6,58
	1646,45	1706,42	54,05	66,58	6,06	6,55
Média	**1636,38**	**1676,76**	**54,27**	**66,06**	**6,07**	**6,57**
Aumento (%)	**2,47%**		**21,72%**		**8,22%**	

First:

	10milhões		1milhão		100mil	
FIRST	**Un**	**P**	**Un**	**P**	**Un**	**P**
	0,20	12,07	0,19	3,50	0,13	3,01
	0,17	11,40	0,16	3,38	0,14	3,05
	0,15	10,99	0,17	3,31	0,19	3,06
	0,14	11,16	0,16	3,36	0,13	3,07
	0,16	11,05	0,16	3,37	0,13	3,09
	0,19	10,99	0,19	3,33	0,18	3,04
	0,15	11,25	0,14	3,34	0,14	3,07
	0,16	10,94	0,16	3,36	0,14	3,07
	0,19	10,94	0,16	3,33	0,13	3,07
	0,13	11,32	0,19	3,39	0,19	3,05
	0,16	11,10	0,16	3,55	0,14	3,04
	0,16	11,23	0,16	3,40	0,14	3,00
	0,18	11,20	0,19	3,34	0,13	3,04
	0,16	10,95	0,20	3,38	0,18	3,12
	0,22	10,97	0,16	3,48	0,14	3,06
	0,21	10,92	0,16	3,39	0,17	3,04
	0,14	10,96	0,19	3,33	0,18	3,04
	0,16	10,93	0,16	3,40	0,13	3,00
	0,16	11,00	0,17	3,37	0,13	3,02
	0,20	11,01	0,19	3,33	0,13	3,06
Média	**0,17**	**11,04**	**0,17**	**3,37**	**0,14**	**3,05**
Aumento (%)	**6552,41%**		**1904,76%**		**2079,29%**	

First (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
FIRST PART	**Un**	**P**	**Un**	**P**	**Un**	**P**
	50,57	21,71	2,35	0,88	0,44	0,79
	0,18	0,64	0,20	0,71	0,16	0,63
	0,18	0,65	0,19	0,65	0,16	0,71
	0,20	0,61	0,16	0,71	0,21	0,66
	0,18	0,74	0,21	0,72	0,16	0,67
	0,18	0,70	0,22	0,70	0,17	0,65
	0,19	0,69	0,18	0,68	0,21	0,71
	0,18	0,69	0,19	0,73	0,16	0,60
	0,18	0,74	0,20	0,72	0,17	0,64
	0,19	0,64	0,18	0,65	0,21	0,70
	0,18	0,72	0,19	0,74	0,16	0,61
	0,18	0,72	0,25	0,70	0,16	0,71
	0,25	0,66	0,20	0,68	0,23	0,61
	0,19	0,70	0,19	0,72	0,18	0,72
	0,19	0,69	0,24	0,75	0,21	0,65
	0,20	0,71	0,18	0,66	0,16	0,67
	0,19	0,66	0,25	0,75	0,16	0,66
	0,25	0,70	0,21	0,68	0,21	0,69
	0,18	0,71	0,19	0,74	0,22	0,60
	0,19	0,65	0,19	0,69	0,16	0,71
Média	**0,19**	**0,69**	**0,20**	**0,71**	**0,18**	**0,67**
Aumento (%)	**271,51%**		**259,39%**		**274,30%**	

Last:

	10milhões		1milhão		100mil	
LAST	**NP**	**P**	**NP**	**P**	**NP**	**P**
	0,27	13,09	0,18	3,57	0,20	3,97
	0,18	13,07	0,21	3,54	0,15	3,92
	0,20	13,16	0,17	3,56	0,16	4,05
	0,17	13,01	0,19	3,57	0,23	3,95
	0,18	13,00	0,21	3,59	0,16	3,96
	0,20	12,90	0,17	3,57	0,16	4,00
	0,17	13,38	0,19	3,56	0,20	4,11
	0,17	13,08	0,22	3,56	0,16	3,95
	0,20	12,97	0,19	3,51	0,16	3,99
	0,17	13,00	0,19	3,66	0,20	3,97
	0,18	12,90	0,21	3,60	0,15	4,02
	0,21	12,98	0,24	3,59	0,15	3,99
	0,24	12,86	0,19	3,64	0,22	3,93
	0,19	13,03	0,18	3,61	0,18	4,06
	0,21	13,26	0,15	3,49	0,15	3,93
	0,17	13,31	0,18	3,52	0,20	3,96
	0,17	12,91	0,26	3,58	0,15	3,98
	0,23	13,00	0,21	3,60	0,15	4,07
	0,17	13,06	0,18	3,80	0,18	3,95
	0,18	13,00	0,19	3,57	0,23	3,94
Média	**0,19**	**13,02**	**0,19**	**3,58**	**0,17**	**3,97**
Aumento (%)	**6939,46%**		**1762,50%**		**2222,81%**	

Last (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
LAST PART	**NP**	**P**	**NP**	**P**	**NP**	**P**
	25,57	7,37	2,66	1,41	0,77	0,86
	0,21	0,66	0,22	0,82	0,18	0,67
	0,21	0,70	0,21	0,80	0,23	0,68
	0,23	0,66	0,22	0,74	0,18	0,67
	0,23	0,75	0,20	0,76	0,17	0,72
	0,22	0,74	0,22	0,85	0,23	0,62
	0,20	0,72	0,22	0,75	0,17	0,73
	0,20	0,69	0,22	0,76	0,19	0,67
	0,22	0,71	0,21	0,74	0,23	0,65
	0,20	0,79	0,24	0,74	0,17	0,68
	0,31	0,69	0,28	0,78	0,23	0,73
	0,20	0,72	0,22	0,69	0,20	0,66
	0,20	0,74	0,20	0,78	0,20	0,73
	0,22	0,71	0,22	0,73	0,23	0,68
	0,20	0,71	0,26	0,81	0,17	0,74
	0,27	0,72	0,25	0,72	0,23	0,61
	0,19	0,74	0,22	0,81	0,25	0,74
	0,20	0,68	0,21	0,75	0,19	0,68
	0,21	0,75	0,23	0,76	0,24	0,68
	0,24	0,77	0,27	0,70	0,19	0,68
Média	**0,21**	**0,72**	**0,22**	**0,76**	**0,21**	**0,69**
Aumento (%)	**240,09%**		**241,70%**		**231,88%**	

Where:

	10milhões		1milhão		100mil	
WHERE	**NP**	**P**	**NP**	**P**	**NP**	**P**
	0,03	5639,55	0,05	116,94	0,03	15,57
	0,03	1122,60	0,03	52,82	0,03	8,84
	0,05	986,14	0,03	53,73	0,03	8,91
	0,03	1042,76	0,05	53,07	0,02	8,89
	0,03	1030,63	0,03	52,81	0,02	9,07
	0,05	1012,14	0,05	52,76	0,03	8,92
	0,03	1009,12	0,03	54,25	0,03	8,81
	0,06	976,02	0,03	55,01	0,03	8,84
	0,03	984,76	0,03	54,42	0,02	8,96
	0,03	962,12	0,05	54,45	0,03	8,97
	0,05	957,00	0,03	54,46	0,03	8,97
	0,03	955,38	0,03	53,80	0,03	8,65
	0,03	935,94	0,05	53,73	0,03	8,72
	0,03	946,44	0,04	52,74	0,05	8,93
	0,05	912,40	0,03	53,46	0,04	8,68
	0,03	925,21	0,05	51,50	0,04	8,74
	0,03	915,72	0,05	52,53	0,06	8,90
	0,05	901,84	0,07	53,77	0,09	8,81
	0,07	878,66	0,03	52,61	0,05	8,92
	0,03	886,20	0,03	52,59	0,03	8,78
Média	**0,03**	**963,81**	**0,04**	**53,42**	**0,03**	**8,88**
Aumento (%)	**2834644,12%**		**144278,38%**		**28535,48%**	

Where (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
WHERE PART	**NP**	**P**	**NP**	**P**	**NP**	**P**
	0,03	0,42	0,17	0,44	0,26	0,52
	0,03	0,43	0,03	0,45	0,03	0,46
	0,03	0,51	0,04	0,60	0,03	0,53
	0,03	0,44	0,03	0,46	0,03	0,44
	0,03	0,41	0,04	0,51	0,03	0,49
	0,03	0,51	0,24	0,46	0,03	0,46
	0,04	0,41	0,04	0,49	0,04	0,45
	0,03	0,48	0,04	0,50	0,03	0,46
	0,05	0,51	0,03	0,50	0,04	0,45
	0,03	0,51	0,03	0,42	0,03	0,54
	0,04	0,43	0,03	0,46	0,12	0,45
	0,03	0,51	0,03	0,54	0,03	0,48
	0,03	0,42	0,03	0,49	0,03	0,45
	0,03	0,49	0,03	0,46	0,03	0,55
	0,04	0,49	0,03	0,46	0,03	0,44
	0,03	0,50	0,08	0,56	0,03	0,48
	0,04	0,41	0,03	0,48	0,03	0,44
	0,04	0,44	0,03	0,46	0,03	0,49
	0,03	0,49	0,03	0,42	0,03	0,44
	0,04	0,43	0,03	0,49	0,03	0,50
Média	**0,03**	**0,46**	**0,03**	**0,48**	**0,03**	**0,47**
Aumento (%)	**1343,75%**		**1384,38%**		**1456,67%**	

Find (apenas com id(s)):

	10milhões		1milhão		100mil	
FIND id	**NP**	**P**	**NP**	**P**	**NP**	**P**
	5,98	30,30	0,44	3,54	0,33	2,76
	3,60	20,90	0,43	2,87	0,38	2,74
	1,57	19,75	0,36	2,57	0,34	2,85
	1,09	19,90	0,34	2,85	0,33	2,91
	0,56	18,17	0,39	2,83	0,41	2,90
	0,38	19,31	0,31	2,43	0,32	2,92
	0,40	19,05	0,45	2,74	0,40	2,91
	0,51	18,80	0,32	2,85	0,41	3,15
	0,36	18,77	0,32	2,60	0,32	3,03
	0,36	18,28	0,46	2,80	0,44	3,05
	0,34	19,15	0,34	2,76	0,32	2,64
	0,33	19,07	0,32	2,79	0,31	2,52
	0,35	18,90	0,50	2,88	0,48	2,70
	0,32	19,33	0,32	2,88	0,33	2,62
	0,41	19,31	0,36	2,71	0,37	3,00
	0,36	19,30	0,46	2,89	0,45	2,64
	0,32	19,03	0,32	2,81	0,33	2,80
	0,31	18,71	0,39	2,89	0,39	2,86
	0,45	18,62	0,33	2,95	0,40	2,81
	0,34	19,06	0,31	3,00	0,33	2,75
Média	**0,39**	**19,10**	**0,36**	**2,83**	**0,36**	**2,83**
Aumento (%)	**4771,94%**		**691,06%**		**685,83%**	

Find (passando ids mais uma condição):

	10milhões		1milhão		100mil	
FIND cond	**NP**	**P**	**NP**	**P**	**NP**	**P**
	0,73	4,02	0,49	4,62	0,48	4,52
	0,43	4,41	0,65	4,23	0,79	4,35
	0,52	4,20	0,52	4,35	0,46	4,25
	0,44	4,28	0,45	4,18	0,53	4,27
	0,44	4,36	0,63	4,29	0,63	4,24
	0,70	4,12	0,48	4,33	0,48	4,14
	0,47	4,44	0,61	4,39	0,61	4,35
	0,59	4,34	0,52	4,86	0,49	4,42
	0,46	4,17	0,63	4,80	0,61	4,36
	0,58	4,22	0,60	4,80	0,47	4,66
	0,49	4,26	0,56	4,77	0,62	4,78
	0,50	4,30	0,70	5,20	0,49	4,99
	0,60	4,27	0,58	5,02	0,57	4,88
	0,45	4,25	0,69	4,84	0,61	4,61
	0,64	4,64	0,57	4,44	0,47	4,74
	0,46	4,49	0,54	4,52	0,60	4,44
	0,52	4,44	0,67	4,37	0,46	4,54
	0,57	4,18	0,47	4,51	0,61	4,38
	0,51	4,23	0,65	4,61	0,46	4,53
	0,57	4,35	0,50	4,49	0,62	4,24
Média	**0,52**	**4,29**	**0,58**	**4,55**	**0,55**	**4,45**
Aumento (%)	**725,82%**		**690,28%**		**713,53%**	

Find (passando ids mais uma condição com campo de particionamento):

	10milhões		1milhão		100mil	
FIND PART	**NP**	**P**	**NP**	**P**	**NP**	**P**
	0,24	0,81	0,19	0,74	0,19	0,64
	0,20	0,73	0,47	0,76	0,19	0,75
	0,21	0,83	0,19	0,75	0,27	0,70
	0,24	0,75	0,25	0,82	0,19	0,72
	0,19	0,73	0,19	0,78	0,24	0,68
	0,24	0,70	0,19	0,84	0,19	0,73
	0,18	0,78	0,25	0,84	0,19	0,67
	0,18	0,64	0,19	0,72	0,27	0,74
	0,23	0,76	0,20	0,76	0,23	0,71
	0,18	0,73	0,33	0,71	0,26	0,77
	0,25	0,78	0,20	0,71	0,27	0,63
	0,21	0,73	0,24	0,73	0,19	0,78
	0,23	0,78	0,19	0,76	0,25	0,70
	0,24	0,78	0,22	0,64	0,19	0,70
	0,19	0,72	0,28	0,80	0,28	0,68
	0,26	0,72	0,19	0,72	0,19	0,76
	0,19	0,71	0,25	0,77	0,19	0,68
	0,18	0,78	0,19	0,69	0,24	0,72
	0,23	0,66	0,27	0,78	0,19	0,69
	0,19	0,78	0,27	0,74	0,29	0,77
Média	**0,21**	**0,75**	**0,22**	**0,75**	**0,22**	**0,71**
Aumento (%)	**253,30%**		**244,50%**		**226,73%**	

Find_by_X:

	10milhões		1milhão		100mil	
FIND_BY	**NP**	**P**	**NP**	**P**	**NP**	**P**
	37,32	78,02	1,65	17,89	0,36	5,21
	0,23	4,68	0,20	4,38	0,19	4,28
	0,18	4,55	0,20	4,36	0,19	4,21
	0,19	4,61	0,25	4,27	0,21	4,23
	0,23	4,50	0,19	4,24	0,24	4,27
	0,20	4,51	0,27	4,33	0,19	4,20
	0,25	4,50	0,21	4,31	0,19	4,21
	0,19	4,44	0,19	4,33	0,24	4,20
	0,19	4,49	0,24	4,30	0,18	4,22
	0,23	4,46	0,18	4,42	0,23	4,21
	0,25	4,51	0,30	4,21	0,19	4,25
	0,24	4,60	0,19	4,30	0,22	4,23
	0,19	4,45	0,31	4,35	0,24	4,26
	0,18	4,48	0,24	4,29	0,18	4,22
	0,29	4,51	0,19	4,29	0,23	4,16
	0,21	4,45	0,31	4,30	0,19	4,23
	0,24	4,50	0,24	4,24	0,19	4,21
	0,18	4,47	0,18	4,29	0,23	4,17
	0,19	4,48	0,24	4,38	0,18	4,18
	0,27	4,48	0,25	4,27	0,23	4,32
Média	**0,21**	**4,50**	**0,23**	**4,31**	**0,21**	**4,22**
Aumento (%)	**1991,16%**		**1806,64%**		**1939,61%**	

Find_by_X (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
FIND_BY PART	**NP**	**P**	**NP**	**P**	**NP**	**P**
	57,14	438,77	2,57	33,81	0,47	7,31
	0,23	4,72	0,19	4,23	0,18	4,18
	0,21	4,69	0,20	4,27	0,18	4,32
	0,22	4,62	0,25	4,13	0,24	4,15
	0,21	4,98	0,18	4,25	0,18	4,27
	0,24	4,69	0,23	4,28	0,23	4,22
	0,20	4,68	0,18	4,21	0,18	4,20
	0,21	4,68	0,18	4,20	0,18	4,20
	0,22	4,59	0,23	4,37	0,24	4,27
	0,21	4,55	0,18	4,25	0,18	4,19
	0,32	4,57	0,23	4,22	0,25	4,20
	0,21	4,68	0,26	4,24	0,28	4,24
	0,22	4,58	0,22	4,34	0,20	4,23
	0,22	4,78	0,25	4,25	0,24	4,22
	0,26	4,56	0,20	4,27	0,18	4,17
	0,24	4,61	0,29	4,28	0,27	4,19
	0,20	4,61	0,19	4,13	0,19	4,14
	0,21	4,62	0,19	4,21	0,18	4,20
	0,22	4,60	0,24	4,21	0,23	4,21
	0,25	4,62	0,18	4,27	0,18	4,20
Média	**0,22**	**4,64**	**0,21**	**4,25**	**0,21**	**4,21**
Aumento (%)	**2009,55%**		**1902,83%**		**1952,20%**	

Find_all_by_X:

	10milhões		1milhão		100mil	
FIND_ALL_BY	**NP**	**P**	**NP**	**P**	**NP**	**P**
	290,57	849,57	28,31	62,99	2,55	8,55
	16,12	30,51	1,54	6,75	0,37	4,05
	16,41	30,58	1,77	6,85	0,35	3,96
	16,58	30,73	1,73	6,86	0,29	3,94
	16,45	30,16	1,67	6,81	0,34	4,06
	15,96	28,93	1,53	6,79	0,34	3,92
	15,92	29,26	1,59	6,81	0,28	3,95
	15,84	30,19	1,63	6,76	0,42	3,97
	16,41	29,26	1,56	6,79	0,35	3,97
	15,99	29,65	1,60	6,86	0,33	4,05
	16,15	30,01	1,58	6,79	0,39	3,93
	15,81	29,56	1,54	6,76	0,35	3,96
	16,08	29,15	1,58	6,88	0,41	3,92
	16,35	29,47	1,59	6,84	0,30	4,06
	16,21	29,26	1,49	6,74	0,35	3,93
	16,07	30,06	1,63	6,68	0,39	3,89
	15,77	29,47	1,58	6,91	0,30	3,93
	16,12	29,91	1,49	6,85	0,34	4,00
	15,99	29,95	1,58	6,87	0,42	4,00
	16,03	29,79	1,67	6,90	0,31	3,94
Média	**16,11**	**29,80**	**1,59**	**6,83**	**0,35**	**3,96**
Aumento (%)	**84,99%**		**328,71%**		**1028,77%**	

Find_all_by_X (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
FIND_ALL_BY PART	**NP**	**P**	**NP**	**P**	**NP**	**P**
	285,44	98,50	27,17	5,86	2,62	1,19
	26,80	27,16	2,76	3,18	0,42	0,89
	23,82	27,09	2,44	3,12	0,41	0,89
	24,25	27,29	2,55	3,13	0,41	0,88
	24,30	27,97	2,54	3,16	0,43	0,90
	24,14	27,34	2,57	3,07	0,50	0,91
	22,92	27,11	2,54	3,08	0,45	0,88
	23,62	26,45	2,46	3,03	0,42	0,89
	23,43	28,07	2,36	3,03	0,41	0,90
	23,13	28,52	2,43	3,57	0,37	0,90
	24,53	29,20	2,50	3,15	0,44	0,79
	24,40	28,57	2,37	3,06	0,43	0,86
	24,49	27,44	2,53	3,22	0,46	0,88
	23,69	27,62	2,55	3,04	0,44	0,88
	24,32	27,20	2,48	3,23	0,41	0,88
	23,19	28,48	2,49	3,16	0,49	0,88
	25,76	28,46	2,57	3,08	0,42	0,91
	23,98	27,90	2,46	3,13	0,48	0,84
	24,20	27,80	2,46	3,09	0,42	0,88
	24,22	26,89	2,44	3,17	0,42	0,82
Média	**24,13**	**27,71**	**2,50**	**3,13**	**0,43**	**0,89**
Aumento (%)	**14,82%**		**25,03%**		**106,29%**	

Update:

	10milhões		1milhão		100mil	
UPDATE	**NP**	**P**	**NP**	**P**	**NP**	**P**
	4,92	29,23	2,28	5,41	2,36	5,80
	4,23	12,59	2,15	5,33	2,34	5,25
	3,37	8,58	2,15	5,39	2,16	5,43
	2,99	7,79	2,40	5,65	2,33	5,35
	2,77	8,20	2,92	5,29	2,30	5,62
	2,51	6,35	2,19	5,73	2,15	5,57
	2,25	5,61	2,09	5,57	2,21	5,56
	2,38	5,80	2,13	5,36	2,10	5,62
	2,41	6,02	2,18	5,26	2,29	5,57
	2,30	5,50	2,09	5,43	2,30	5,64
	2,32	5,59	2,21	5,48	2,18	5,33
	2,11	5,63	2,28	5,44	2,22	5,18
	2,16	5,47	2,06	5,70	2,20	5,71
	2,24	5,40	2,28	5,21	2,12	5,61
	2,12	5,34	2,13	5,59	2,20	5,65
	2,10	5,40	2,12	5,55	2,15	5,74
	2,19	5,27	2,40	5,44	2,27	5,86
	2,20	5,47	2,18	5,52	2,21	5,57
	2,25	5,38	2,07	5,26	2,23	5,19
	2,44	5,48	2,13	5,28	2,44	5,41
Média	**2,33**	**5,69**	**2,17**	**5,44**	**2,23**	**5,56**
Aumento (%)	**144,29%**		**150,12%**		**149,22%**	

Update_all:

	10milhões		1milhão		100mil	
UPDATE ALL	**NP**	**P**	**NP**	**P**	**NP**	**P**
	1126,60	720,82	347,94	26,99	34,30	4,28
	1089,57	569,92	405,37	25,99	35,46	4,47
	1051,99	570,19	279,15	12,76	34,45	4,13
	1120,97	576,00	576,17	13,31	35,06	4,26
	1070,50	553,73	451,69	14,00	34,01	4,26
	1091,46	535,70	396,75	12,47	34,05	4,32
	1059,89	524,52	289,09	13,64	34,61	4,20
	1091,07	508,23	524,05	12,70	35,70	4,16
	1050,52	523,02	386,95	13,16	34,98	4,20
	1086,01	485,91	497,76	14,64	35,33	4,25
	1055,20	465,96	488,93	12,88	36,74	5,61
	1052,75	450,80	347,21	14,00	35,91	4,50
	1037,37	461,88	450,88	12,62	35,59	4,15
	1055,23	448,95	367,83	12,67	36,06	4,36
	996,34	415,90	241,16	13,53	35,53	4,22
	1052,12	437,93	480,01	12,59	36,36	4,30
	1015,29	411,05	464,96	13,24	35,97	4,50
	1034,66	399,43	549,11	13,67	35,62	4,16
	997,32	413,30	370,14	12,65	35,29	4,21
	1015,33	402,95	353,62	13,63	35,21	4,21
Média	**1057,16**	**484,29**	**412,82**	**13,25**	**35,38**	**4,25**
Aumento (%)	**-54,19%**		**-96,79%**		**-87,98%**	

Update_all (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
UPDATE ALL PART	**NP**	**P**	**NP**	**P**	**NP**	**P**
	819,36	291,39	271,05	79,36	46,15	15,01
	800,63	163,53	374,20	117,41	45,67	17,68
	782,37	213,13	224,48	75,63	45,73	12,99
	801,15	183,58	279,19	116,07	45,06	17,47
	773,41	165,00	300,46	71,74	45,71	13,62
	824,41	224,70	399,25	92,82	44,82	16,55
	791,16	180,13	331,38	71,33	45,49	14,49
	797,91	255,89	248,29	78,29	45,47	17,87
	794,54	213,56	362,49	82,51	44,68	18,86
	761,47	214,15	373,54	70,85	45,98	18,82
	805,49	220,52	393,17	77,06	44,87	20,55
	811,49	182,53	309,97	96,25	45,87	14,12
	788,74	256,35	333,27	79,98	45,63	15,63
	770,12	230,48	272,18	66,80	44,84	18,77
	775,86	239,88	365,46	101,32	44,95	15,24
	778,39	297,87	329,56	74,53	44,00	22,95
	789,05	177,32	297,09	74,16	45,28	16,12
	790,45	282,98	385,33	76,96	44,63	19,06
	741,39	193,21	324,95	79,93	44,20	15,87
	818,53	240,75	309,92	72,54	44,55	19,45
Média	**791,44**	**217,40**	**326,46**	**77,84**	**45,21**	**17,00**
Aumento (%)	**-72,53%**		**-76,16%**		**-62,39%**	

Exists:

	10milhões		1milhão		100mil	
EXISTS	**NP**	**P**	**NP**	**P**	**NP**	**P**
	0,96	13,79	0,67	3,85	0,69	4,26
	0,56	6,53	0,67	3,47	1,02	4,71
	0,54	5,27	0,53	3,58	0,85	5,05
	0,57	4,46	0,57	3,66	0,68	4,69
	0,66	4,95	0,62	3,47	0,56	4,50
	0,59	4,75	0,52	3,69	0,68	4,06
	0,60	4,09	0,59	3,70	0,61	5,71
	0,67	4,24	0,45	4,02	0,62	4,47
	0,59	4,24	0,58	3,61	0,62	4,44
	0,67	4,40	0,45	3,61	0,48	5,32
	0,63	4,41	0,61	3,65	0,59	4,77
	0,62	4,14	0,61	3,77	0,54	4,81
	0,66	4,44	0,52	3,60	0,55	4,65
	0,64	4,79	0,55	3,50	0,55	4,59
	0,66	4,80	0,46	3,70	0,59	4,14
	0,68	4,78	0,58	3,54	0,55	3,98
	0,60	3,97	0,52	3,71	0,56	3,89
	0,65	3,93	0,54	3,76	0,61	3,85
	0,63	4,52	0,57	3,60	0,56	3,80
	0,59	4,27	0,49	3,76	0,56	3,95
Média	**0,63**	**4,51**	**0,56**	**3,65**	**0,59**	**4,45**
Aumento (%)	**617,52%**		**558,20%**		**656,97%**	

Exists (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
EXISTS PART	**NP**	**P**	**NP**	**P**	**NP**	**P**
	7,58	12,35	1,95	1,04	0,35	0,89
	0,40	0,73	0,29	0,80	0,34	0,82
	0,34	0,77	0,32	0,88	0,26	0,85
	0,36	0,85	0,26	0,87	0,31	0,81
	0,34	0,82	0,31	0,94	0,33	0,82
	0,27	0,79	0,32	0,87	0,27	0,77
	0,43	0,85	0,27	0,79	0,32	0,85
	0,34	0,88	0,32	0,85	0,28	0,78
	0,30	0,80	0,33	0,80	0,38	0,84
	0,39	0,83	0,31	0,90	0,35	0,85
	0,27	0,86	0,31	0,84	0,30	0,72
	0,32	0,75	0,28	0,81	0,34	0,83
	0,39	0,81	0,37	0,75	0,26	0,84
	0,27	0,85	0,26	0,88	0,33	0,84
	0,35	0,86	0,42	0,89	0,33	0,79
	0,28	0,79	0,33	0,79	0,30	0,82
	0,33	0,85	0,27	0,86	0,34	0,82
	0,32	0,83	0,35	0,79	0,33	0,84
	0,29	0,80	0,27	0,85	0,32	0,86
	0,37	0,80	0,33	0,78	0,32	0,84
Média	**0,34**	**0,82**	**0,31**	**0,84**	**0,32**	**0,83**
Aumento (%)	**144,51%**		**170,19%**		**157,28%**	

Count:

	10milhões		1milhão		100mil	
COUNT	**NP**	**P**	**NP**	**P**	**NP**	**P**
	341,49	9033,87	27,09	44,99	2,92	11,90
	0,17	237,63	1,35	2,86	0,24	2,94
	0,19	2,87	0,16	2,82	0,16	2,89
	0,23	2,88	0,16	2,87	0,17	2,82
	0,16	2,87	0,21	2,83	0,22	2,83
	0,17	2,91	0,17	2,87	0,16	2,85
	0,22	2,83	0,21	2,85	0,16	2,82
	0,17	2,85	0,22	2,86	0,21	2,92
	0,19	2,84	0,16	2,87	0,16	2,86
	0,21	2,84	0,16	3,02	0,16	2,90
	0,18	2,84	0,24	2,83	0,21	2,85
	0,23	2,87	0,22	2,87	0,23	2,98
	0,27	2,82	0,18	2,89	0,21	2,83
	0,17	2,84	0,22	2,83	0,22	2,87
	0,17	2,85	0,17	2,85	0,16	2,84
	0,23	2,84	0,17	2,87	0,16	2,88
	0,24	3,07	0,25	2,84	0,22	2,87
	0,17	2,86	0,17	2,95	0,17	2,84
	0,23	2,84	0,17	2,88	0,16	2,82
	0,18	2,85	0,22	2,85	0,22	2,85
Média	**0,20**	**2,85**	**0,19**	**2,86**	**0,19**	**2,86**
Aumento (%)	**1348,73%**		**1375,26%**		**1413,23%**	

Count (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
COUNT PART	**NP**	**P**	**NP**	**P**	**NP**	**P**
	343,46	211,26	25,64	4,28	2,74	1,56
	0,16	1,14	0,44	1,19	0,22	1,17
	0,16	1,20	0,19	1,15	0,16	1,15
	0,25	1,20	0,22	1,10	0,17	1,18
	0,16	1,19	0,16	1,19	0,22	1,14
	0,20	1,10	0,16	1,15	0,16	1,17
	0,21	1,24	0,22	1,20	0,16	1,18
	0,16	1,21	0,16	1,09	0,21	1,10
	0,16	1,17	0,16	1,17	0,16	1,12
	0,21	1,21	0,22	1,07	0,16	1,12
	0,16	1,15	0,17	1,20	0,21	1,16
	0,19	1,17	0,21	1,11	0,16	1,16
	0,28	1,11	0,22	1,19	0,24	1,15
	0,19	1,21	0,16	1,13	0,23	1,10
	0,22	1,06	0,16	1,22	0,16	1,17
	0,17	1,20	0,22	1,12	0,16	1,10
	0,21	1,15	0,17	1,12	0,22	1,16
	0,24	1,17	0,16	1,10	0,18	1,14
	0,18	1,15	0,26	1,11	0,17	1,20
	0,17	1,22	0,16	1,17	0,22	1,19
Média	**0,19**	**1,18**	**0,19**	**1,15**	**0,19**	**1,16**
Aumento (%)	**524,87%**		**511,70%**		**522,04%**	

Destroy_all:

	10milhões		1milhão		100mil	
DESTROY_ALL	**NP**	**P**	**NP**	**P**	**NP**	**P**
	1,25	1,04	0,78	0,92	0,80	0,90
	0,94	0,97	0,87	1,02	0,78	0,95
	0,97	1,28	0,81	0,98	0,85	0,87
	1,55	0,97	0,84	1,03	0,81	1,09
	0,83	0,95	0,88	0,99	0,77	0,90
	0,85	1,03	0,82	0,97	0,84	0,95
	1,08	0,98	0,82	0,94	0,79	0,97
	0,95	1,12	0,91	0,90	0,88	0,90
	0,99	1,33	0,94	0,98	0,83	0,91
	0,98	1,21	0,97	0,99	0,80	0,94
	0,88	1,02	0,85	1,03	0,84	0,92
	0,85	1,04	0,92	0,90	0,80	0,97
	0,87	0,96	0,88	1,00	0,86	0,86
	0,91	1,21	0,91	0,96	0,86	0,95
	0,90	1,30	1,45	0,94	0,82	0,92
	0,90	1,00	0,91	0,91	0,84	1,00
	0,93	0,97	0,96	1,02	0,83	0,91
	0,92	1,10	0,93	0,91	0,85	0,95
	0,88	1,08	1,22	0,96	0,81	0,92
	1,05	1,02	1,02	1,03	0,87	0,86
Média	**0,93**	**1,04**	**0,90**	**0,97**	**0,83**	**0,93**
Aumento (%)	**12,39%**		**7,89%**		**12,09%**	

Destroy_all (com campo de particionamento incluído nos argumentos):

	10milhões		1milhão		100mil	
DESTROY_ALL PART	**NP**	**P**	**NP**	**P**	**NP**	**P**
	47,14	10,47	1,02	0,98	0,80	0,83
	4,15	2,12	1,08	0,98	0,88	1,03
	4,09	2,19	1,05	1,01	0,77	1,02
	4,13	2,04	1,20	0,98	0,82	0,93
	4,13	2,22	1,10	0,99	0,89	0,96
	4,09	2,03	1,20	1,02	0,86	0,86
	4,19	2,26	1,13	0,99	0,91	0,97
	4,12	2,01	1,13	1,00	0,84	0,85
	4,24	2,23	1,17	1,05	0,86	0,92
	4,08	2,09	1,12	0,95	0,87	0,91
	4,13	2,26	1,28	0,96	0,81	0,93
	4,16	2,11	1,09	0,94	0,88	0,87
	4,27	2,08	1,13	1,25	0,94	0,96
	4,10	2,04	1,17	0,96	0,81	0,93
	4,12	2,17	1,14	1,02	1,05	0,97
	4,13	2,16	1,13	0,99	0,79	0,89
	4,05	2,07	1,15	0,91	0,84	0,96
	4,08	2,15	1,12	1,02	0,82	0,86
	4,11	2,21	1,20	0,93	0,86	0,93
	4,06	2,16	1,10	1,00	0,85	0,87
Média	**4,12**	**2,14**	**1,13**	**0,99**	**0,85**	**0,92**
Aumento (%)	**-47,97%**		**-12,72%**		**8,59%**	